CSE544
Data Management

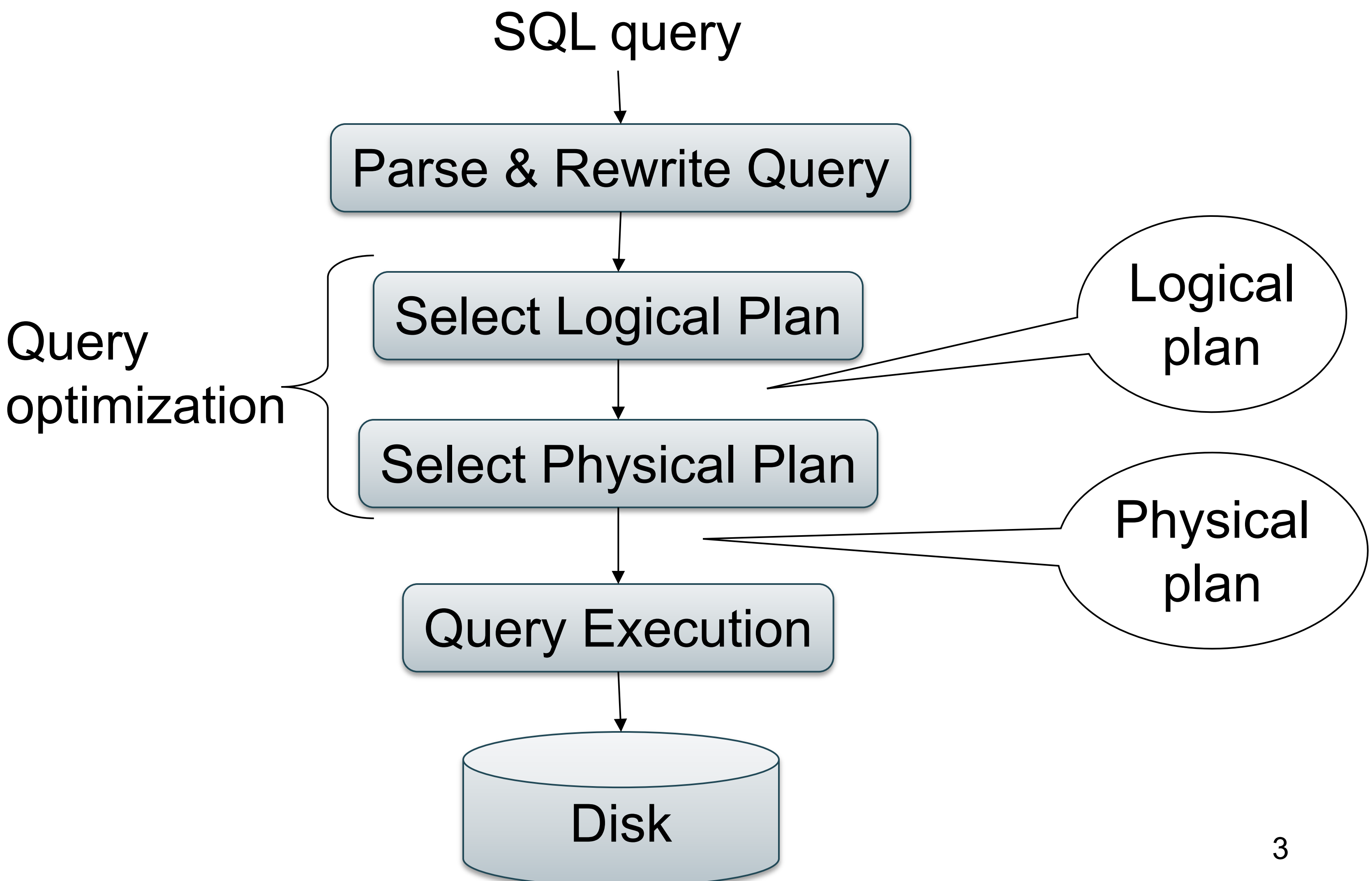
Lectures 8

Query Execution – Part 2

Announcements

- This Friday: Kyle's office hour moved
  - 1:30-2:20, Gates 274
- Project proposals due on Monday, 2/5
- HW2 due on Wednesday 2/7
- R3 pushed to Wednesday, 2/14

Lifecycle of a Query



Main Memory Operators

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)
Part(pno,pname,psize,pcolor)

1. Nested Loop Join

Logical operator:

$$\text{Supplier} \bowtie_{\text{sno}=\text{sno}} \text{Supply}$$

```
for x in Supplier do
    for y in Supply do
        if x.sno = y.sno
            then output(x,y)
```

If |R|=|S|=n,
what is the runtime?

$O(n^2)$

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)
Part(pno,pname,psize,pcolor)

1. Index Join

Logical operator:

$$\text{Supplier} \bowtie_{\text{sno}=\text{sno}} \text{Supply}$$

```
for x in Supplier do
    for y in SupplyIndex(x.sno) do
        output(x,y)
```

If |R|=|S|=n, what is the runtime?

O(n)

When data is on disk
then big difference between
clustered and unclustred

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)
Part(pno,pname,psize,pcolor)

Clustered v.s. Unclustered

Logical operator:

$$\text{Supplier} \bowtie_{\text{sno=sno}} \text{Supply}$$

```
for x in Supplier do
   for y in SupplyIndex(x.sno) do
      output(x,y)
```

Rule of thumb:
Random reading 1-2% of Supply ≈ sequential scan entire file

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)
Part(pno,pname,psize,pcolor)

2. Hash Join

Logical operator:

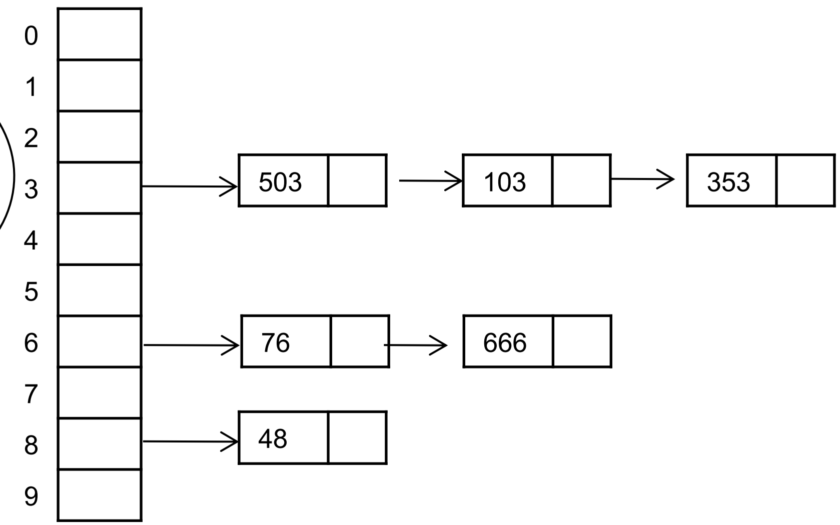
$$\text{Supplier} \bowtie_{\text{sno=sno}} \text{Supply}$$

Build phase

```
for x in Supplier do
     insert(x.sno, x)

for y in Supply do
     x = find(y.sno);
     output(x,y);
```

Probe phase



If |R|=|S|=n,
what is the runtime?

O(n)

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)
Part(pno,pname,psize,pcolor)

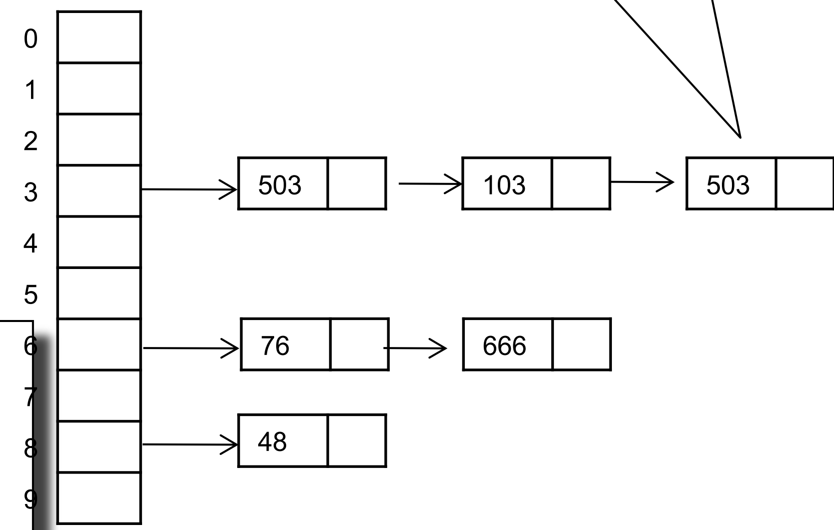
2. Hash Join

There may be duplicate sno's

Change join order

Logical operator:

$$\text{Supply} \bowtie_{sno=sno} \text{Supplier}$$




```
for y in Supply do
    insert(y.sno, y)

for x in Supplier do
    for y in find(x.sno) do
        output(x,y);
```

If |R|=|S|=n,
what is the runtime?

O(n)
But can be $O(n^2)$ why?

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)
Part(pno,pname,psize,pcolor)

3. Merge Join

Logical operator:

$$\text{Supplier} \bowtie_{\text{sno}=\text{sno}} \text{Supply}$$

```
Sort(Supplier); Sort(Supply);
x = Supplier.first();
y = Supply.first();
while y != NULL do
  case:
    x.sno < y.sno: x = x.next()
    x.sno = y.sno: output(x,y); y = y.next();
    x.sno > y.sno: y = y.next();
```

If |R|=|S|=n,
what is the runtime?

O(n log(n))

Brief Review:
Hash Tables, Sorting

Hash Tables

- Array: map indices to memory locations
  – A[0], A[1], A[2], … sequential in memory

- How to map texts to memory locations?
  – A[“alice”], A[“bob”],A[“carl”]…???

- Hash function: maps strings to indices

Separate chaining:

Hash Tables

A (naïve) hash function:

$$\boxed{h(\text{“abc”}) = (\text{‘a’}+\text{’b’}+\text{’c’}) \bmod 10}$$

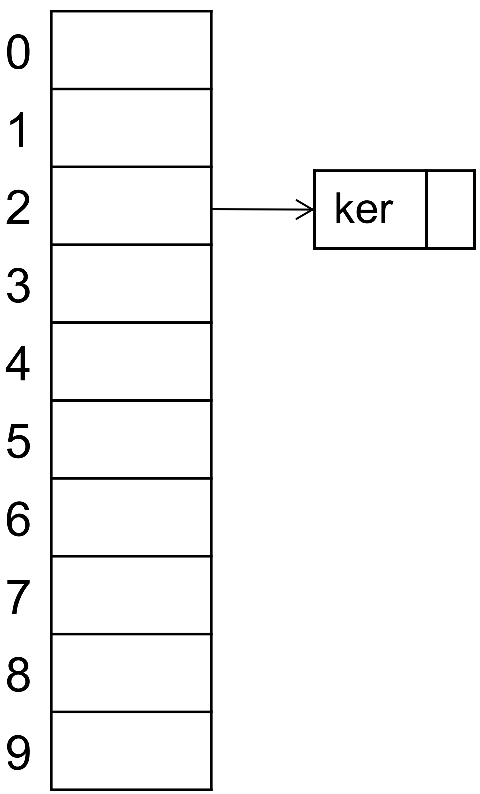
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Separate chaining:

Hash Tables

A (naïve) hash function:

h(“abc”) = (‘a’+’b’+’c’) mod 10



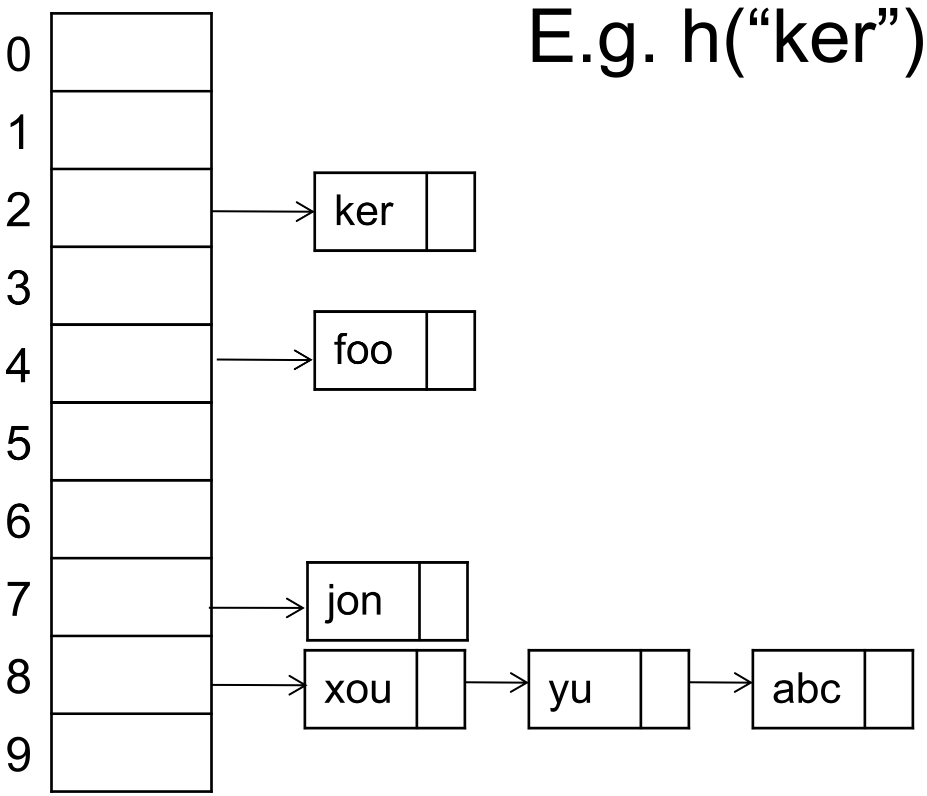
E.g. h(“ker”) = (‘k’+’e’+’r’) mod 10
= (107+101+114) mod 10
= 2

Separate chaining:

Hash Tables

A (naïve) hash function:

$$\boxed{h(\text{“abc”}) = (\text{‘a’}+\text{’b’}+\text{’c’}) \bmod 10}$$



E.g. $h(\text{“ker”}) = (\text{‘k’}+\text{’e’}+\text{’r’}) \bmod 10$

$= (107+101+114) \bmod 10$

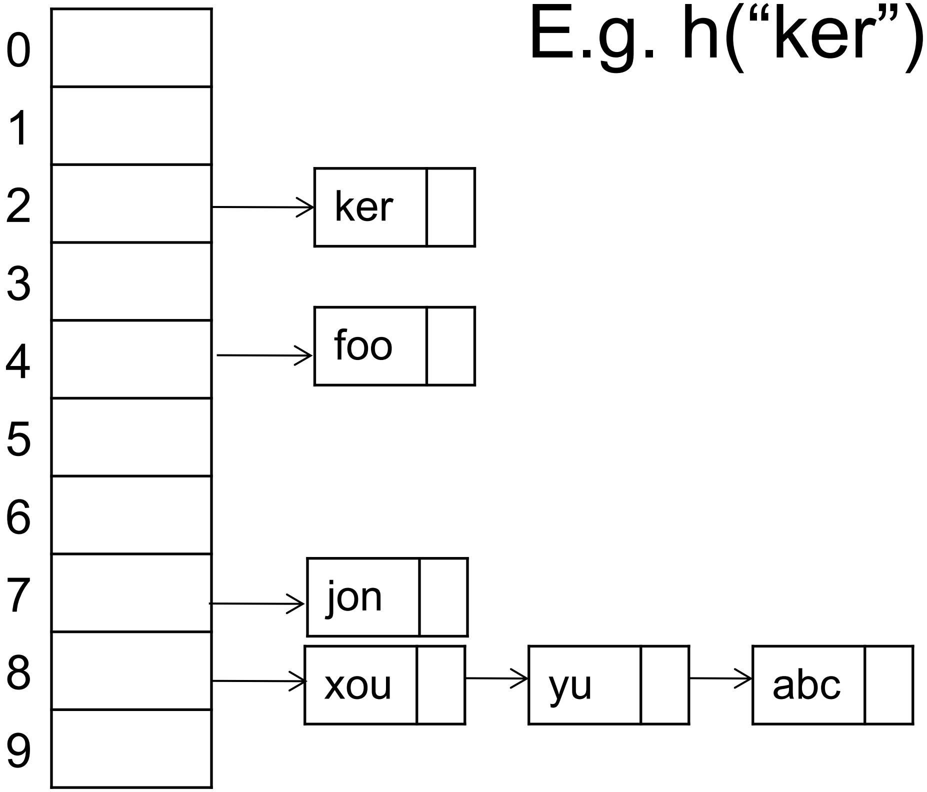
$= 2$

Separate chaining:

Hash Tables

A (naïve) hash function:

$$\boxed{h(\text{“abc”}) = (\text{‘a’}+\text{’b’}+\text{’c’}) \bmod 10}$$



E.g. $h(\text{“ker”}) = (\text{‘k’}+\text{’e’}+\text{’r’}) \bmod 10$

$= (107+101+114) \bmod 10$

$= 2$

find(“yu”) = ??

insert(“alice”) = ??

Hash Table Takeaways

- Use good hash function, never your own. E.g. https://15445.courses.cs.cmu.edu/fall2023/slides/07-hashtables.pdf
  - Low probability of collision
- The vector needs to be pre-allocated:
  - Too big: waste space
  - To small: long chains
  - Extensible hash table: doubles the vector
- Skewed data leads to collisions: O(1)→O(N)

Sorting

- Given an array, sort it in increasingly

74	7	45	99	2	90	19	87	61	82

Sorting

- Given an array, sort it in increasingly

74	7	45	99	2	90	19	87	61	82

2	7	19	45	61	74	82	87	90	99

Sorting

- Given an array, sort it in increasingly

74	7	45	99	2	90	19	87	61	82

2	7	19	45	61	74	82	87	90	99

- Simple algorithms: $O(N^2)$
- Quicksort: $O(N \log N)$
- Mergesort: $O(N \log N)$

Merge Sort: Overview

- A run is a subarray that is sorted
- Repeat:
  - Merge two runs
  - Store output in some array T
  - Switch A,T
- Stop when A is one single run

Merging two Arrays

Assume ∞ at the end

```
Merge(A, B)          // A, B are sorted
```

Merging two Arrays

Assume ∞ at the end

```
Merge(A, B)              // A, B are sorted
  i=0;  j=0;  k=0;
  while i<n or j<n
    case:
        A[i]<B[j]:  T[k++] = A[i++];
```

Merging two Arrays

Assume ∞ at the end

```
Merge(A, B)              // A, B are sorted
  i=0;  j=0;  k=0;
  while i<n or j<n
    case:
        A[i]<B[j]:  T[k++] = A[i++];
        A[i]≥B[j]:  T[k++] = B[j++]
```

Merging two Arrays

Assume ∞ at the end

```
Merge(A, B)              // A, B are sorted
  i=0;  j=0;  k=0;
  while i<n or j<n
    case:
       A[i]<B[j]:  T[k++] = A[i++];
       A[i]≥B[j]:  T[k++] = B[j++]
```

Time = O(|A|+|B|)

Merge-Sort

74	7	45	99	2	90	19	87	61	82

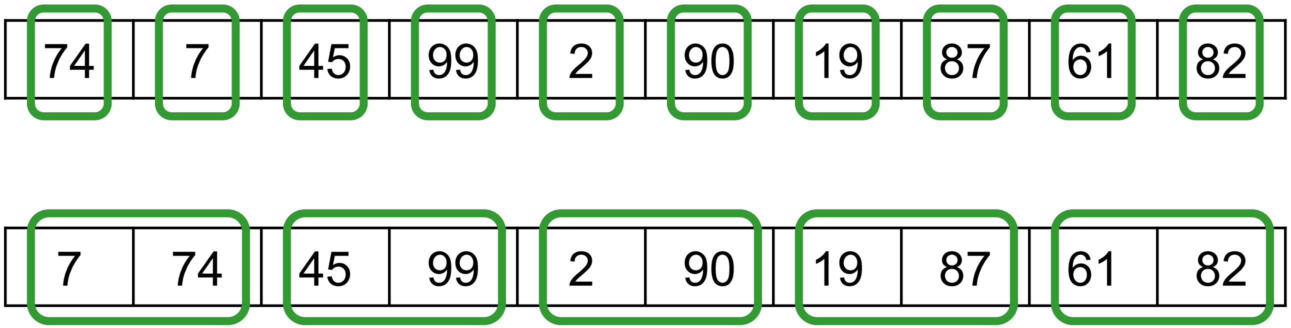
Merge-Sort

Runs

74	7	45	99	2	90	19	87	61	82

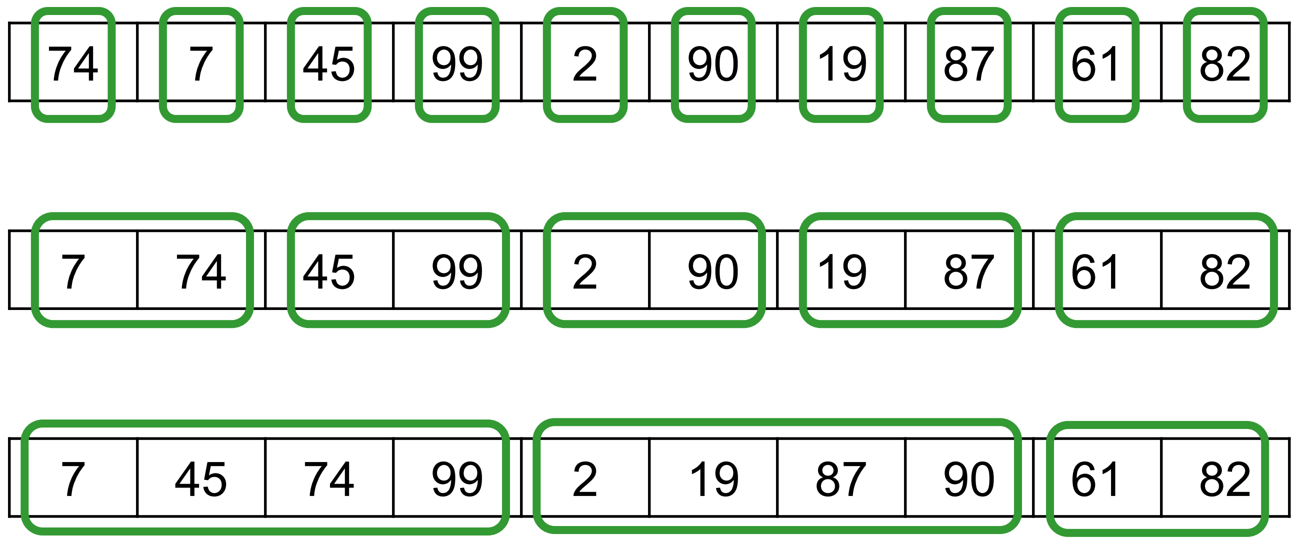
Merge-Sort

Runs



Merge-Sort

Runs



Merge-Sort

Runs

74	7	45	99	2	90	19	87	61	82

7	74	45	99	2	90	19	87	61	82

7	45	74	99	2	19	87	90	61	82

2	7	19	45	74	87	90	99	61	82

Merge-Sort

Runs

74	7	45	99	2	90	19	87	61	82

7	74	45	99	2	90	19	87	61	82

7	45	74	99	2	19	87	90	61	82

2	7	19	45	74	87	90	99	61	82

2	7	19	45	61	74	82	87	90	99

Discussion

- Main memory algorithms use quicksort
  - O(N log N) expected runtime
  - Problem: random access, not good for disk

Discussion

- Main memory algorithms use quicksort
  - O(N log N) expected runtime
  - Problem: random access, not good for disk
- Merge-sort:
  - O(N log N) runtime

Discussion

- Main memory algorithms use quicksort
  - O(N log N) expected runtime
  - Problem: random access, not good for disk
- Merge-sort:
  - O(N log N) runtime
  - log N sequential reads   will improve

Discussion

- Main memory algorithms use quicksort
  - O(N log N) expected runtime
  - Problem: random access, not good for disk
- Merge-sort:
  - O(N log N) runtime
  - log N sequential reads   will improve
  - But: O(N) memory overhead; Not used for main memory, except for python's TimSort

External Memory Operators

Setup

Main cost = number of disk I/O's
Always ignore the final write

Setup

Main cost = number of disk I/O's
Always ignore the final write

- B(R) = number of bloks used to store R
- T(R) = number of records in R

Setup

Main cost = number of disk I/O's
Always ignore the final write

- B(R) = number of bloks used to store R
- T(R) = number of records in R
- Sequential read of R: cost = B(R)
- Random read of R: cost = T(R)

Setup

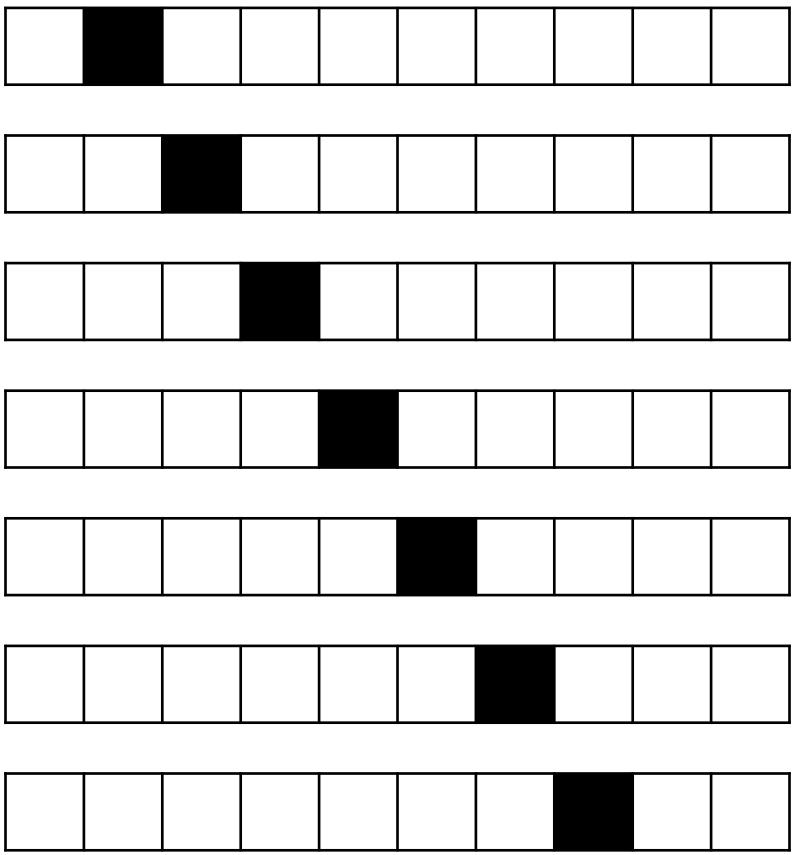
Main cost = number of disk I/O's
Always ignore the final write

- B(R) = number of bloks used to store R
- T(R) = number of records in R
- Sequential read of R: cost = B(R)
- Random read of R: cost = T(R)

$$T(R) \gg B(R)$$

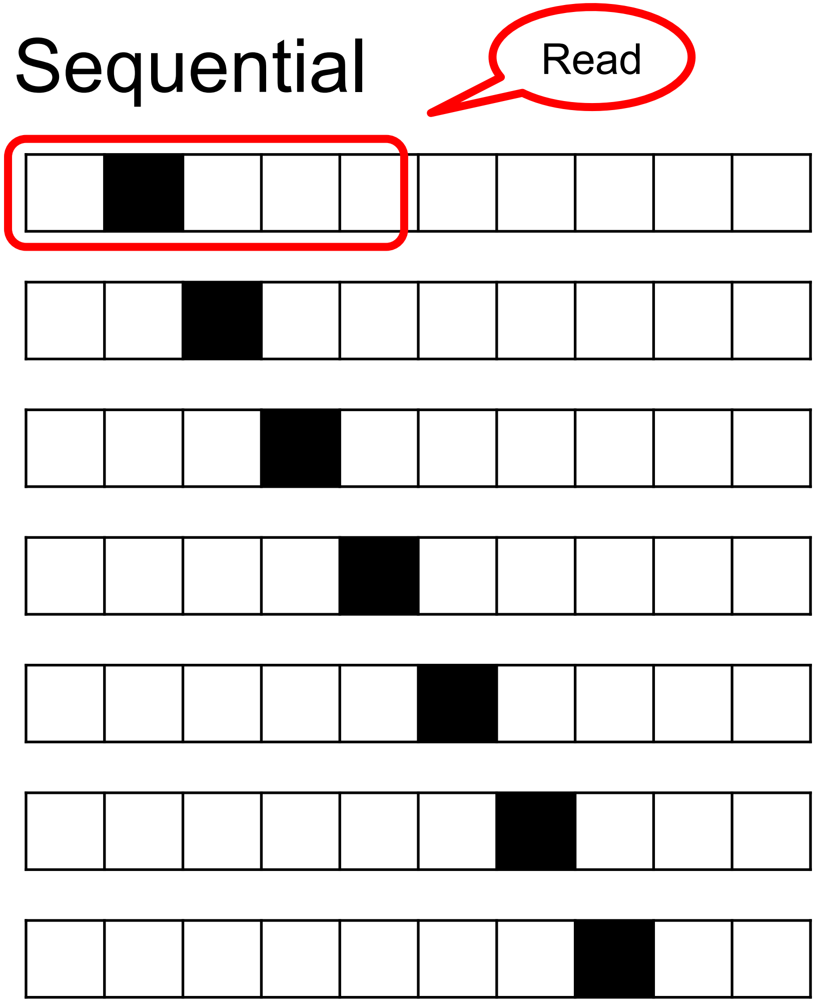
Sequential/Random Access

Sequential



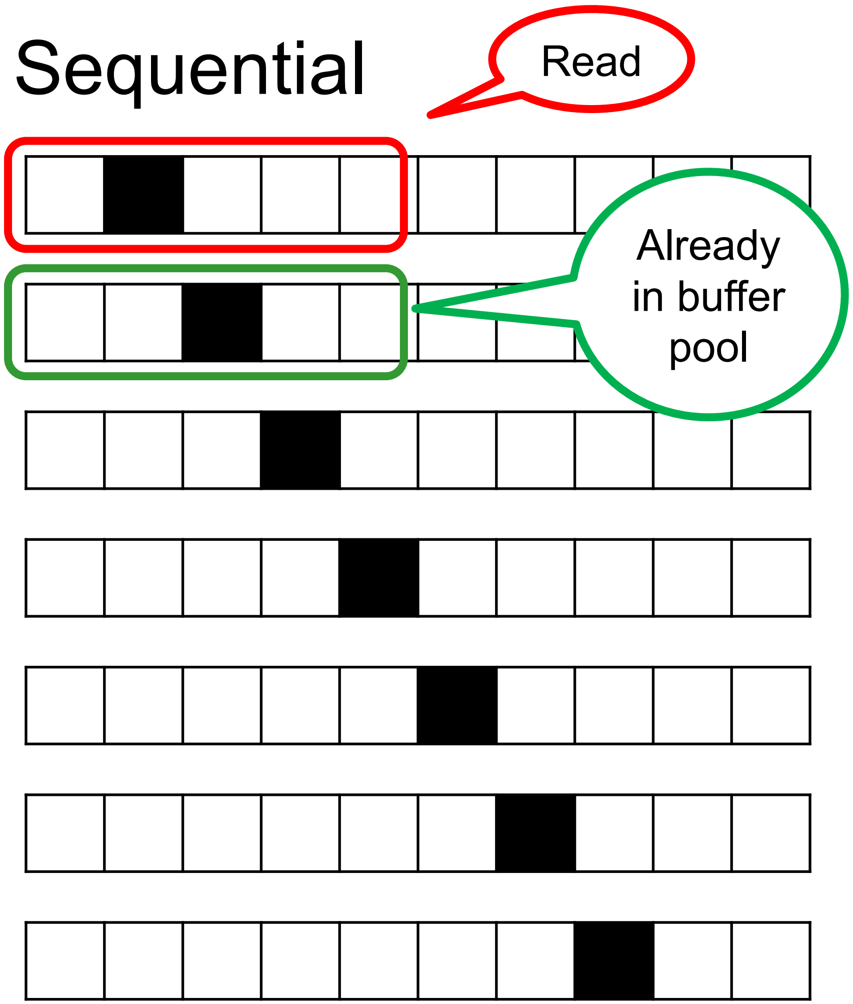
Sequential/Random Access

Sequential



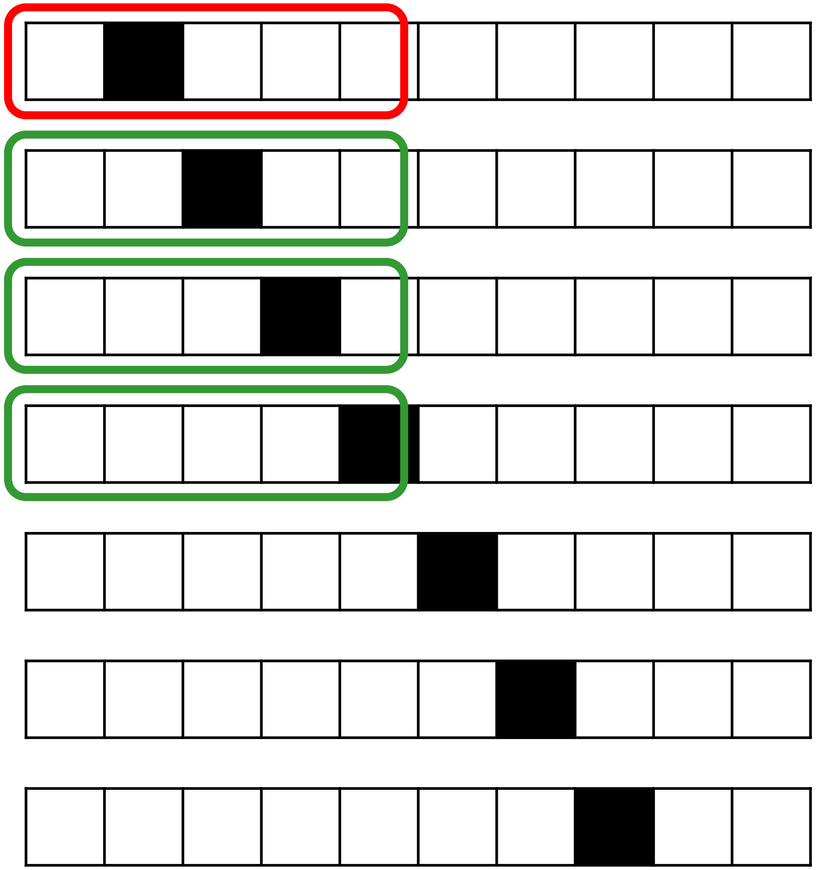
Sequential/Random Access

Sequential



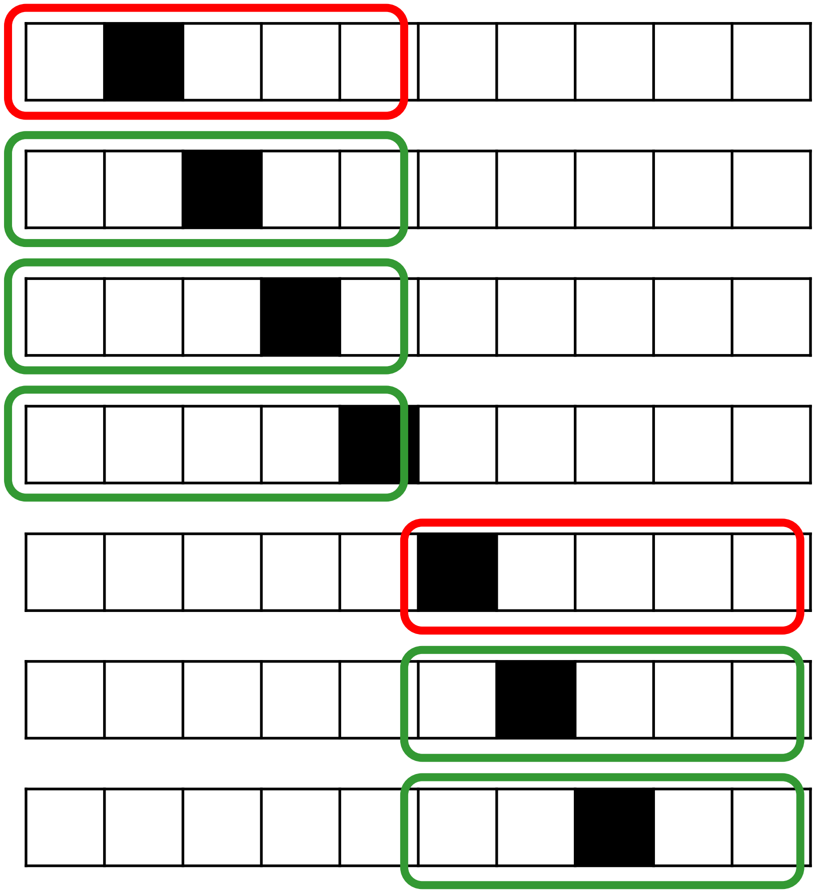
Sequential/Random Access

Sequential



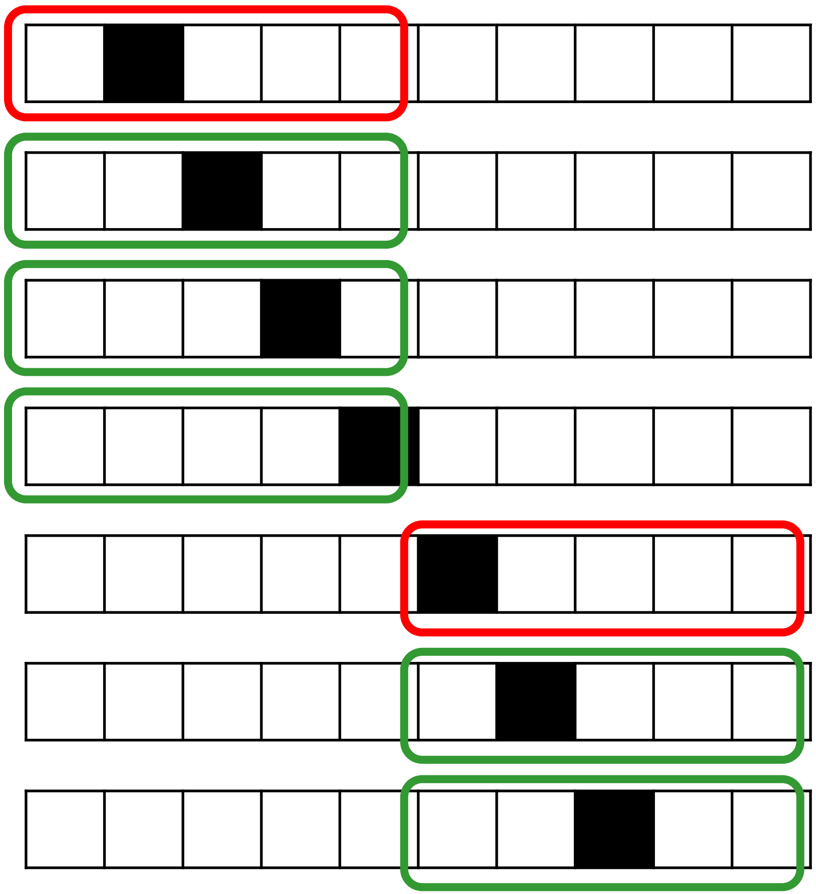
Sequential/Random Access

Sequential: B(R)=2 reads

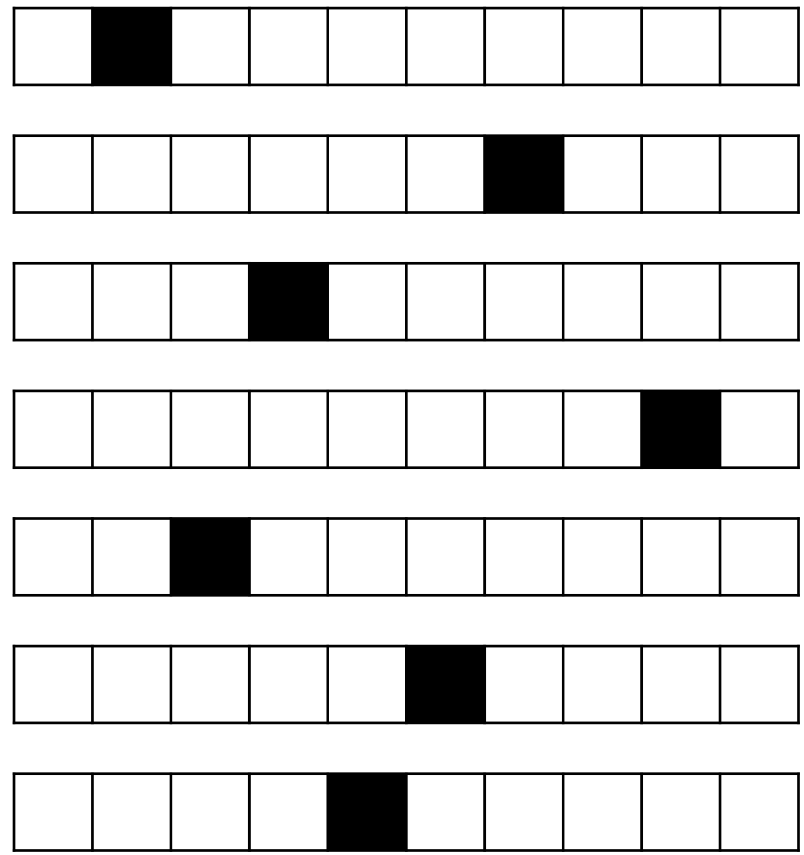


Sequential/Random Access

Sequential: B(R)=2 reads

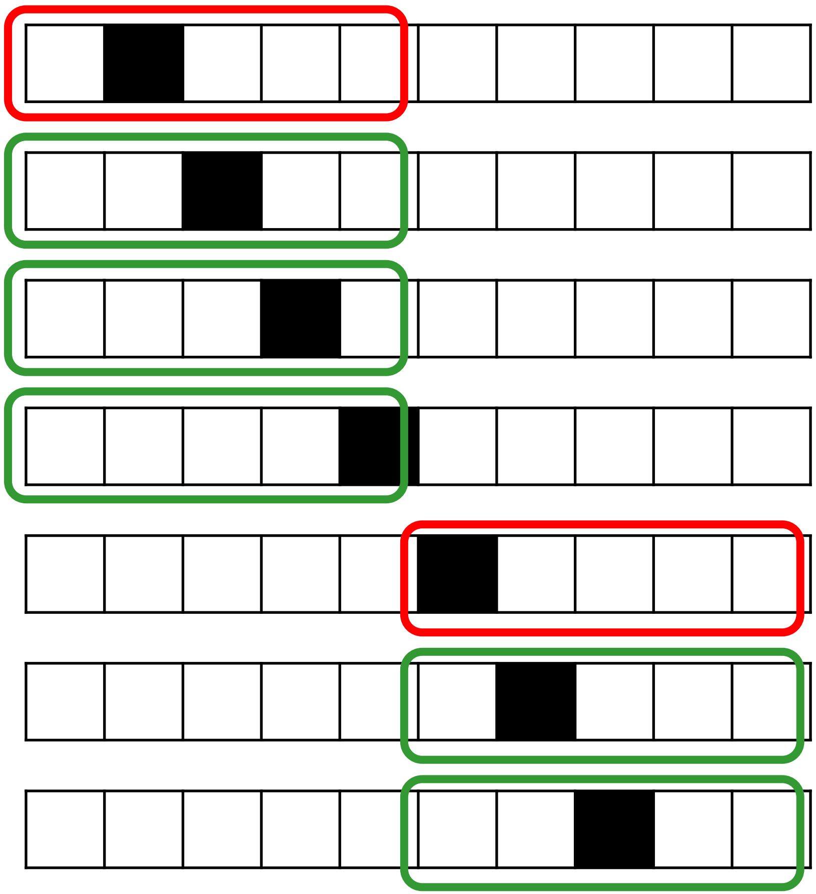


Random

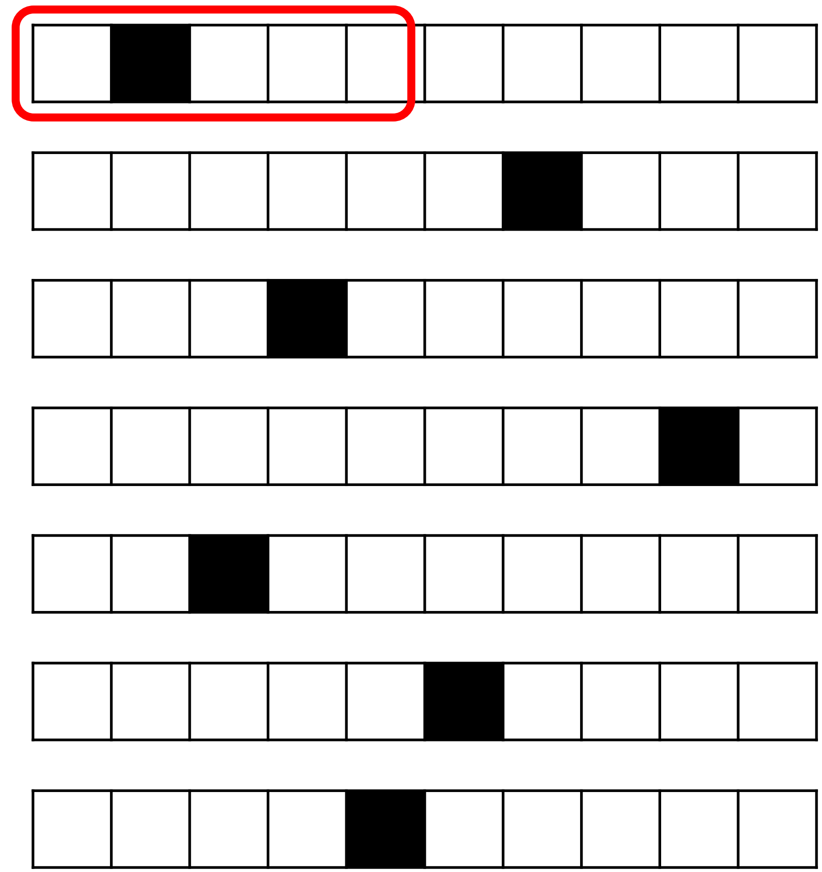


Sequential/Random Access

Sequential: B(R)=2 reads

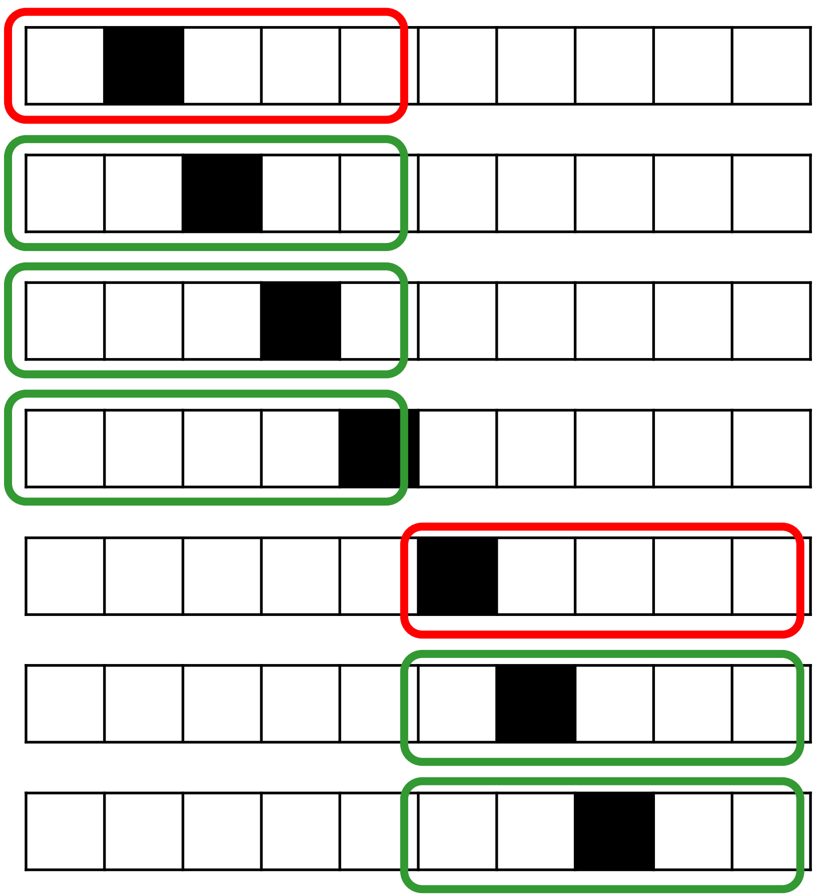


Random

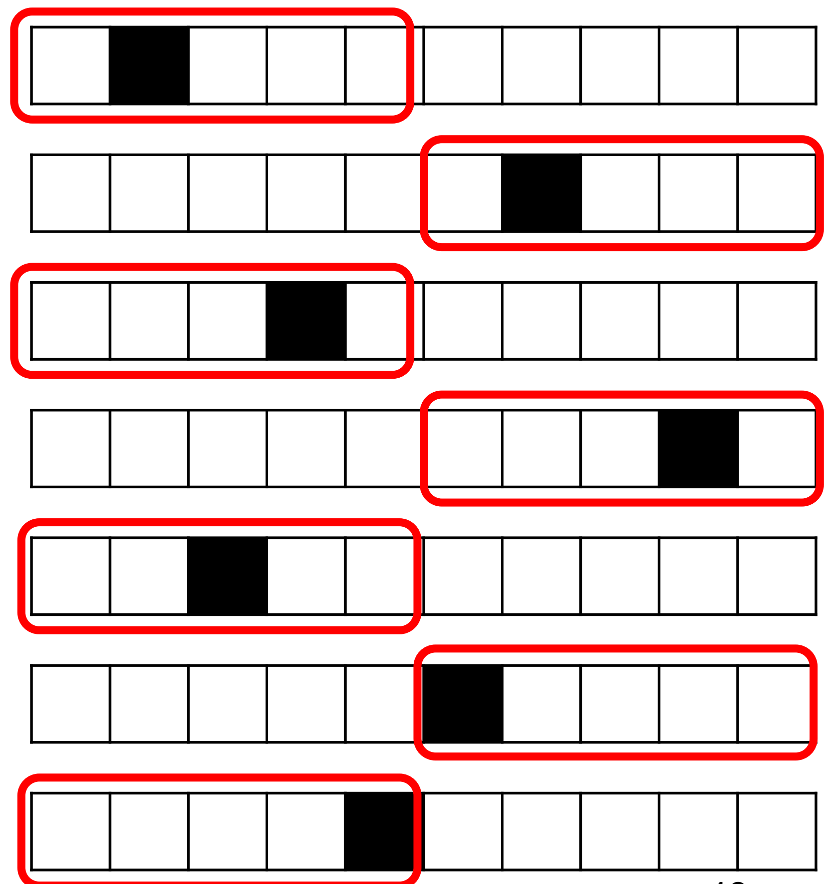


Sequential/Random Access

Sequential: B(R)=2 reads



Random: T(R)=10 reads



External Memory Operators

- Block Nested Loop Join
- Merge Join
- Partitioned Hash Join

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"

<u>for</u> each (M-2) pages PR of R <u>do</u>
    <u>for</u> each page PS of S <u>do</u>
        Main memory join: PR $\bowtie$ PS

Block Nested Loop Join

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

Why not use M-1 pages?

<u>for</u> each (M-2) pages PR of R <u>do</u>

  <u>for</u> each page PS of S <u>do</u>

    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 $\bowtie$ PS
        use the remaining page for output

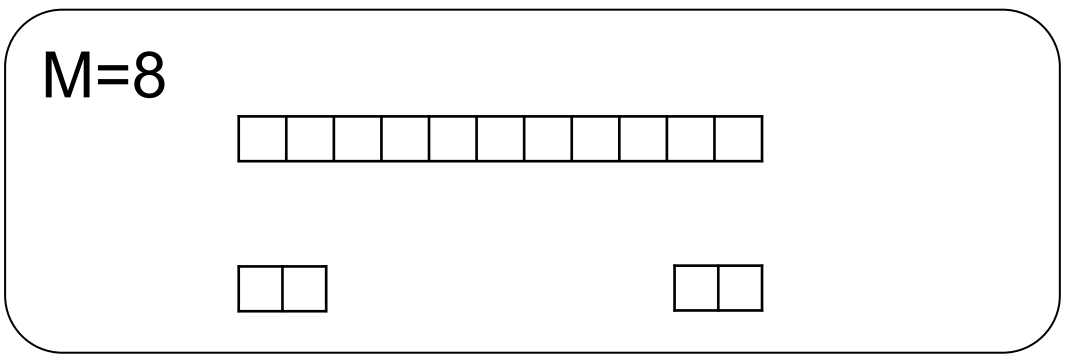
Block Nested Loop Join

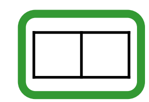
Group of (M-2) pages of R, called a "block"

<u>for</u> each (M-2) pages PR of R <u>do</u>
    <u>for</u> each page PS of S <u>do</u>
        Main memory join: PR $\bowtie$ PS
        use the remaining page for output

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

Block Nested Loop Join

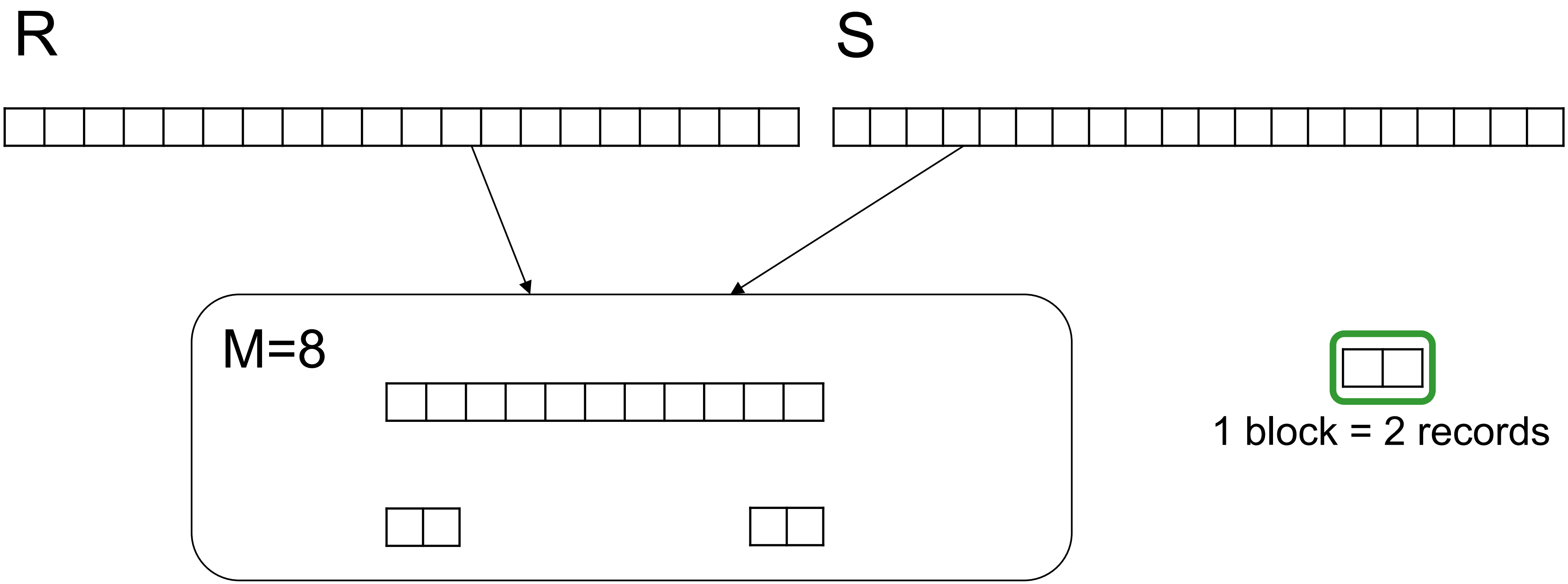




1 block = 2 records

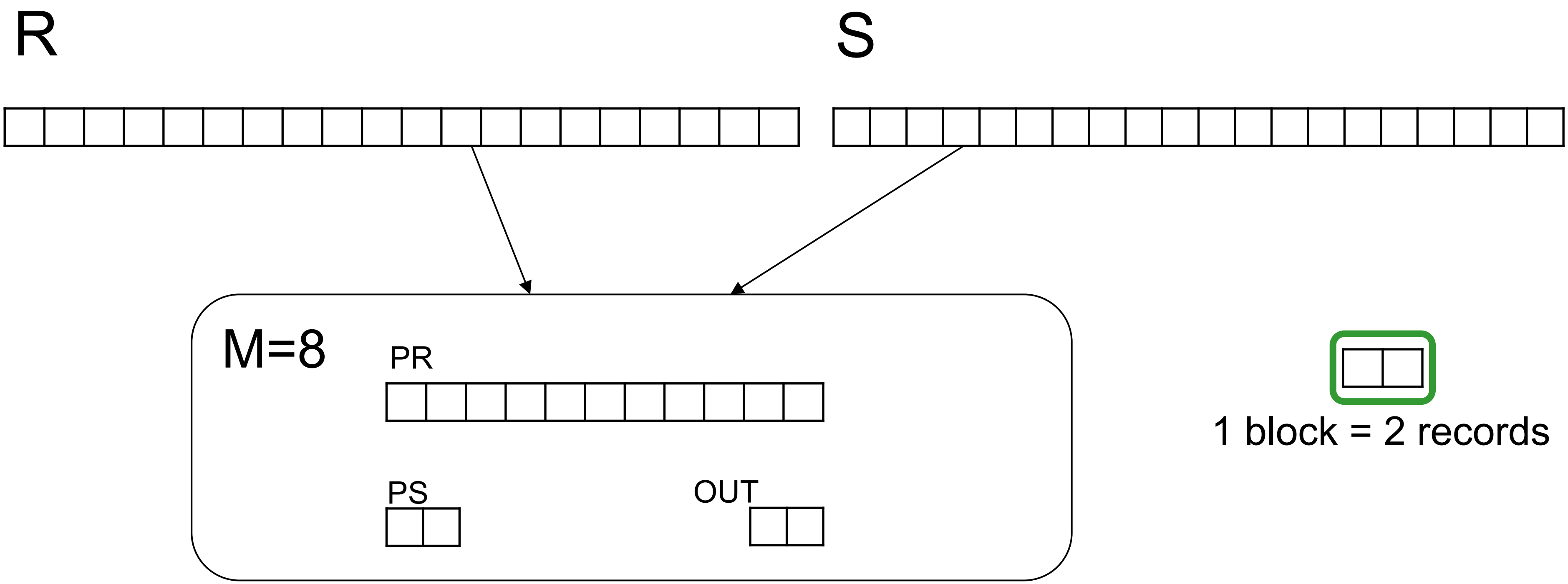
$R \bowtie S$

Block Nested Loop Join



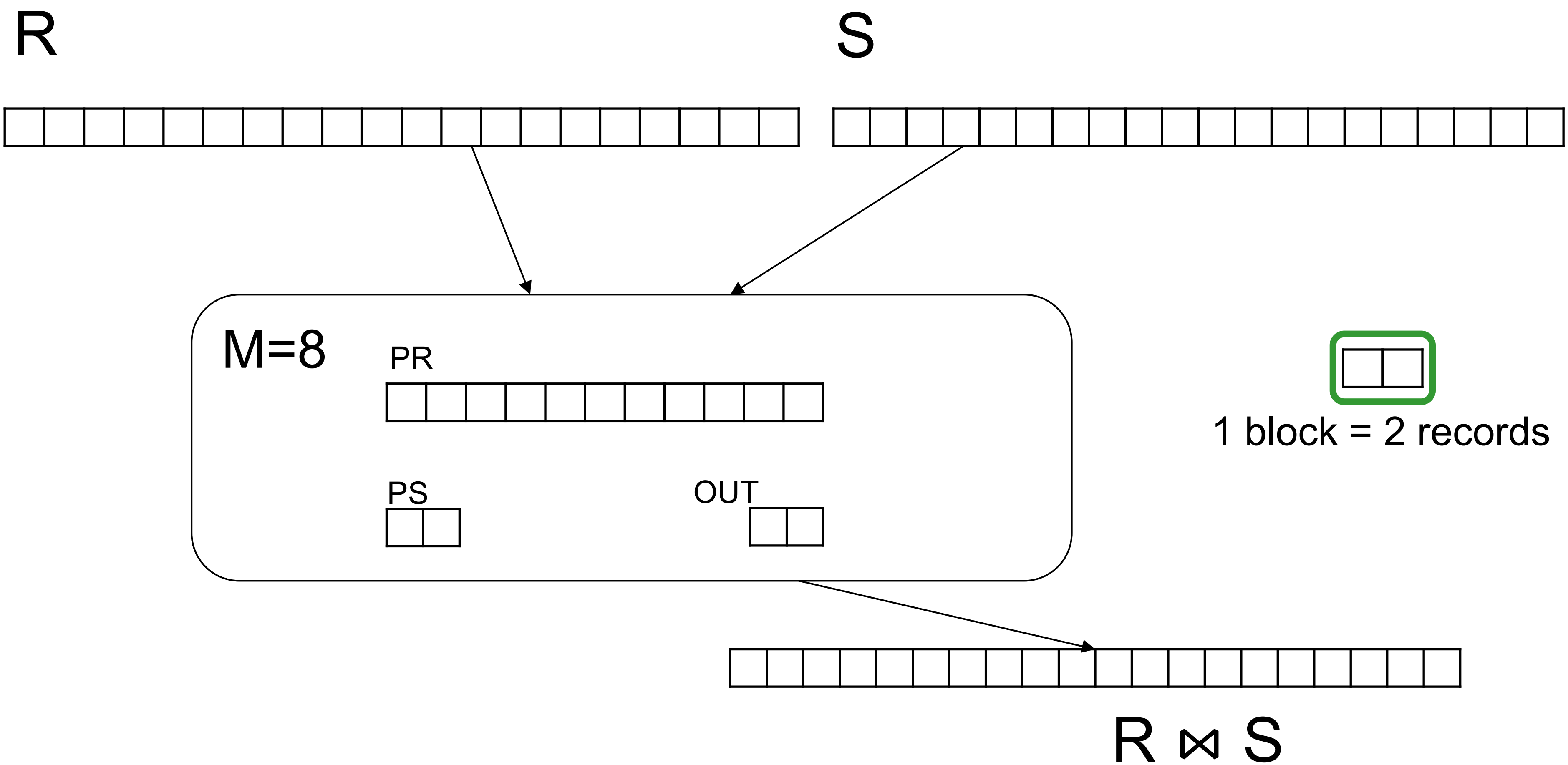
$R \bowtie S$

Block Nested Loop Join

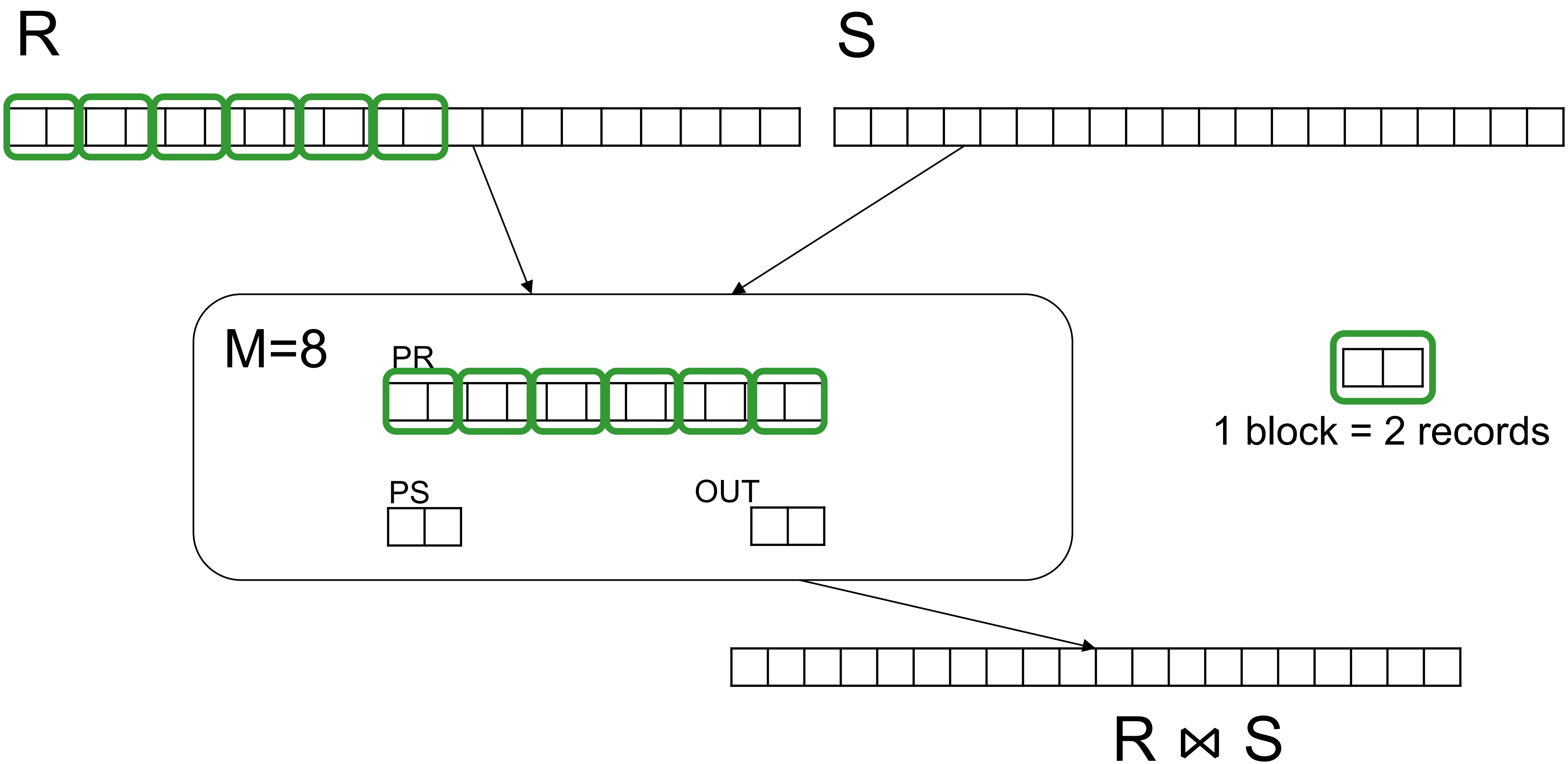


$R \bowtie S$

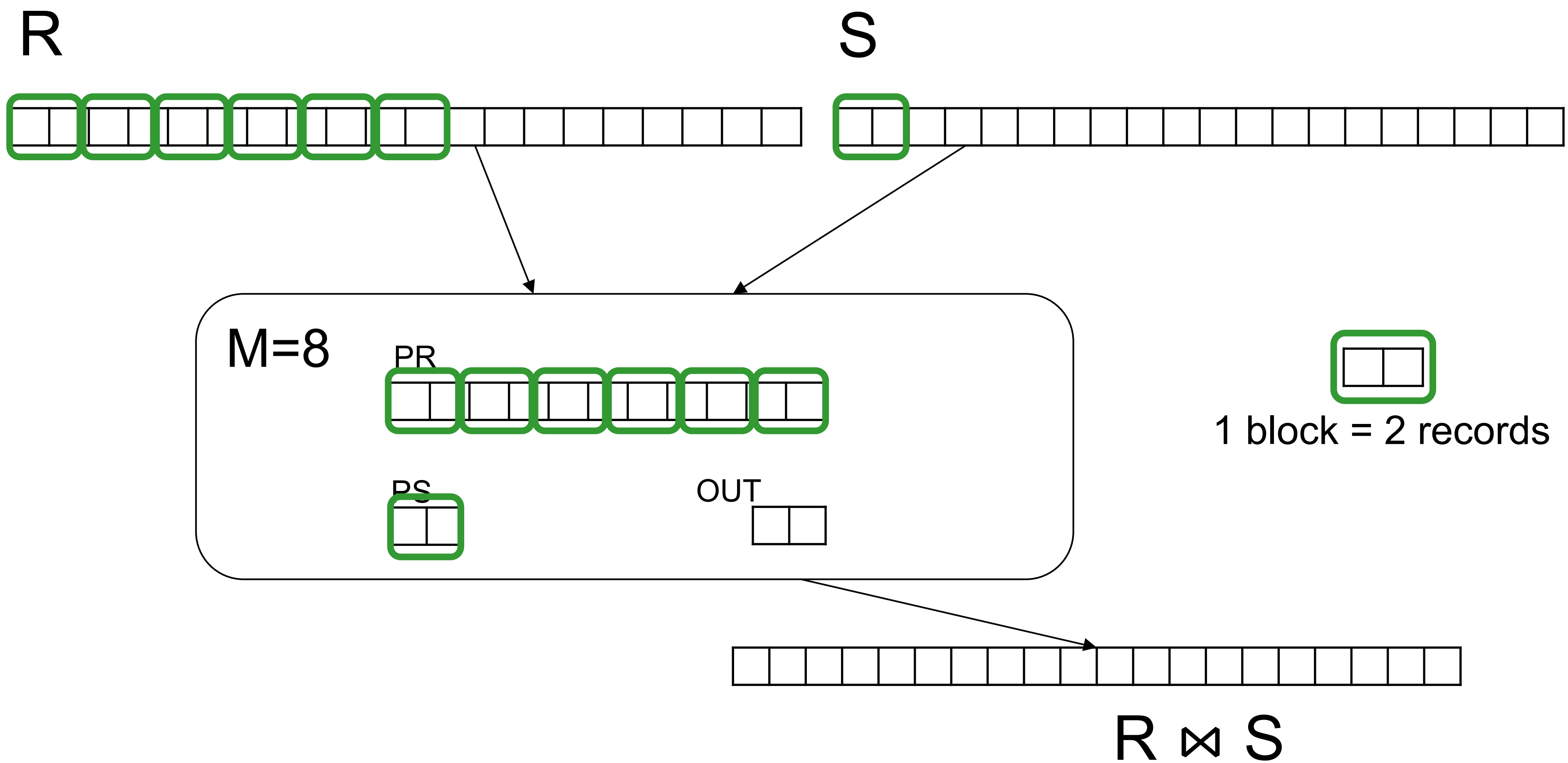
Block Nested Loop Join



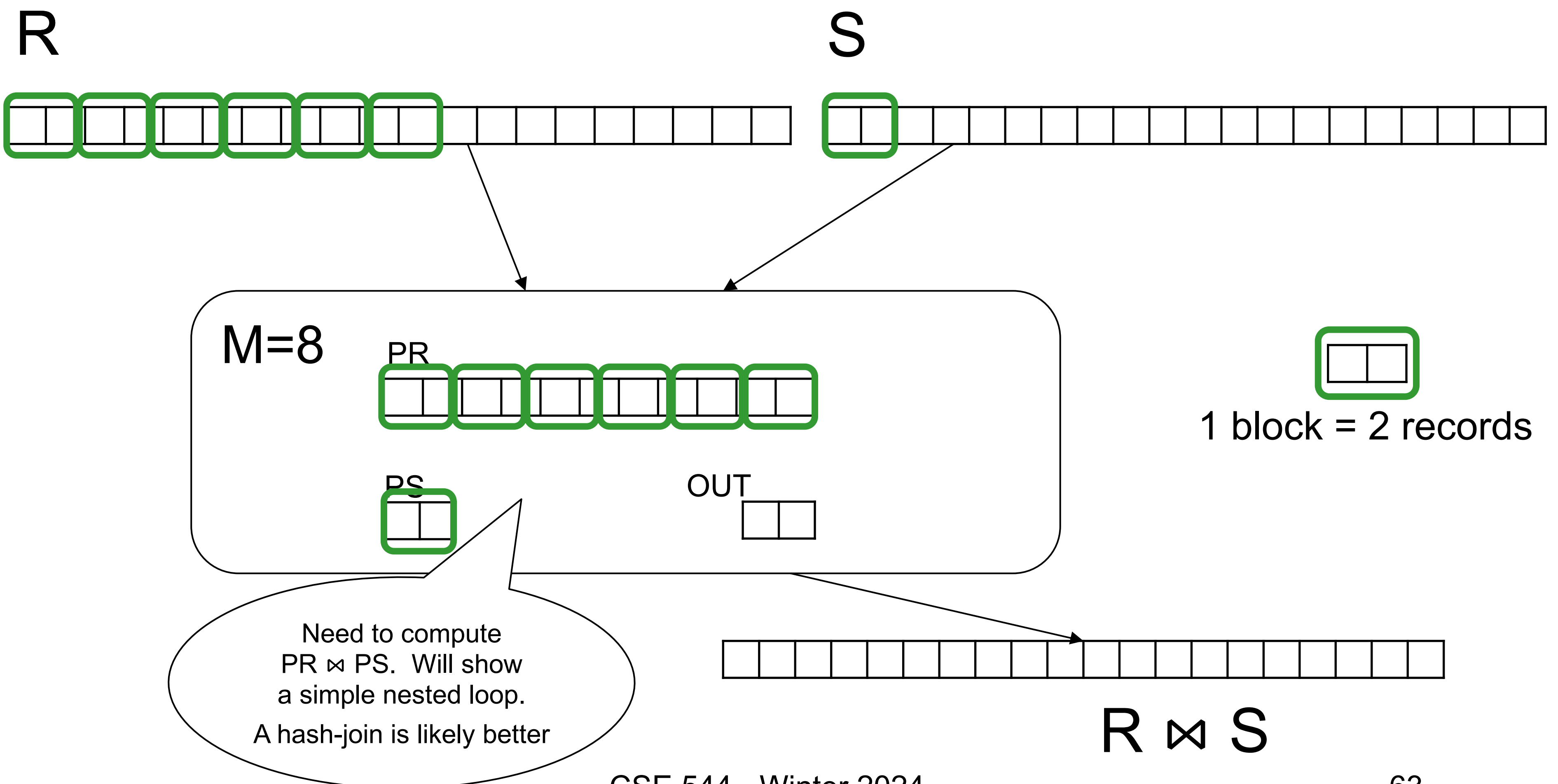
Block Nested Loop Join



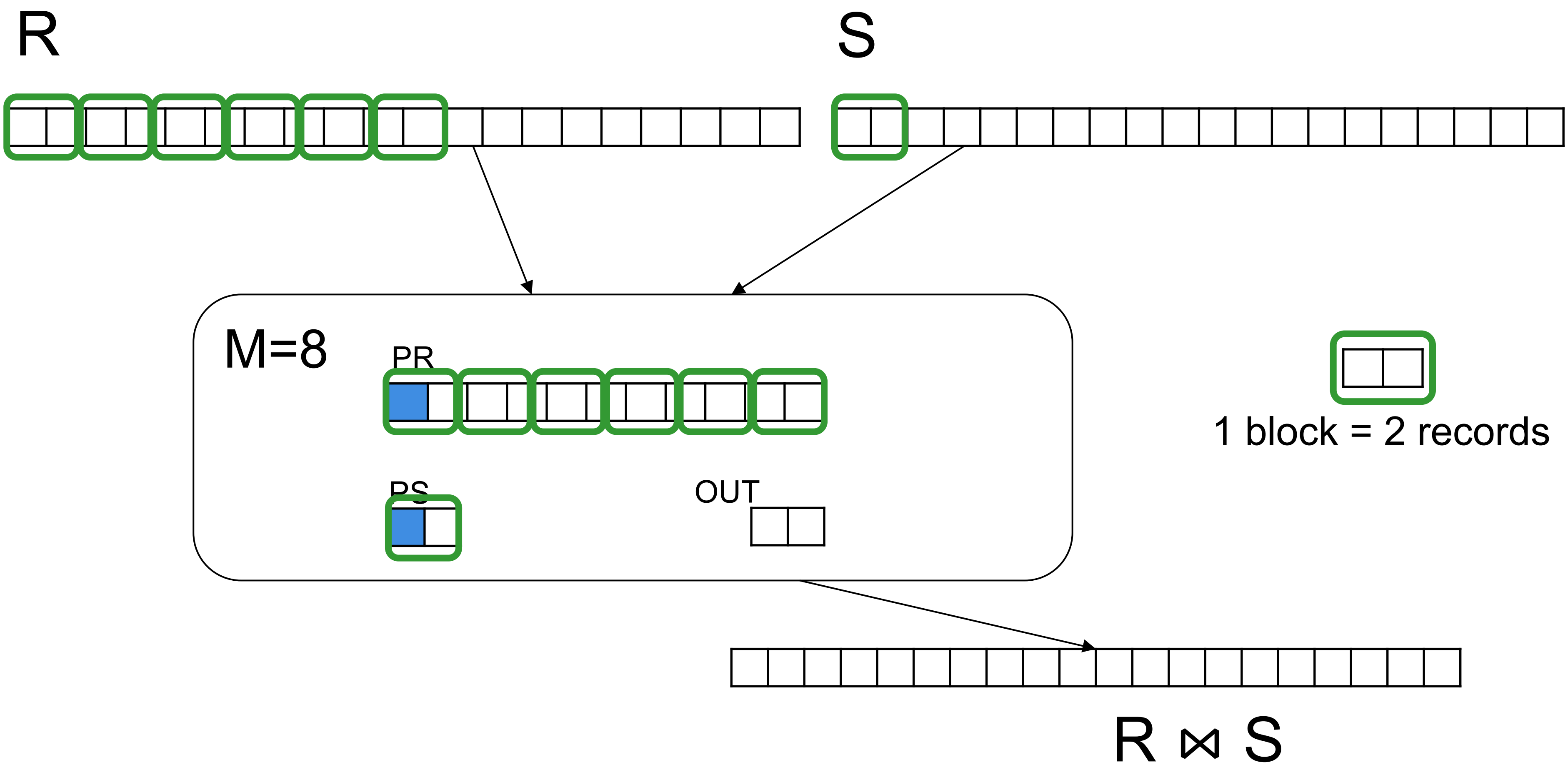
Block Nested Loop Join



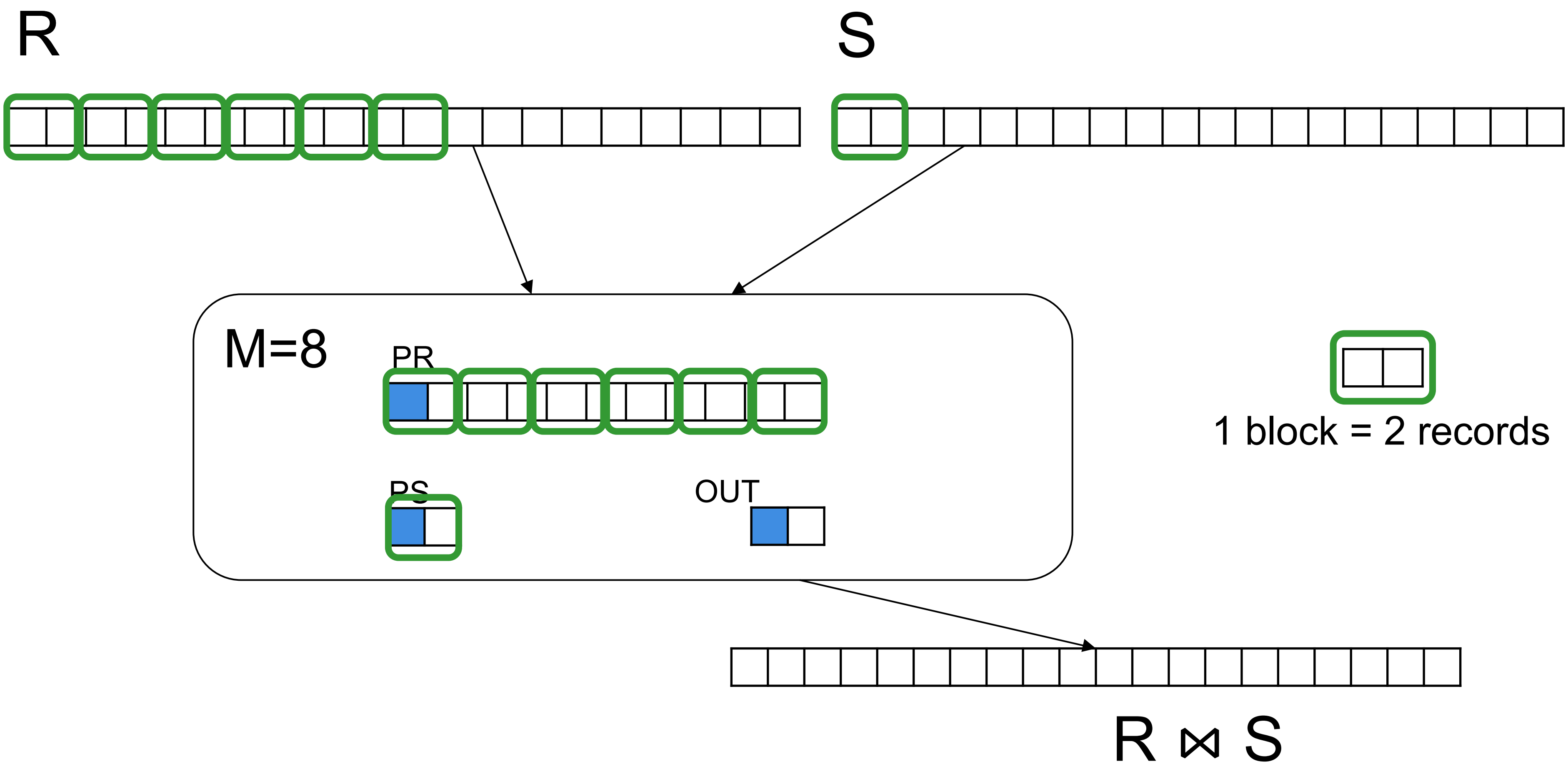
Block Nested Loop Join



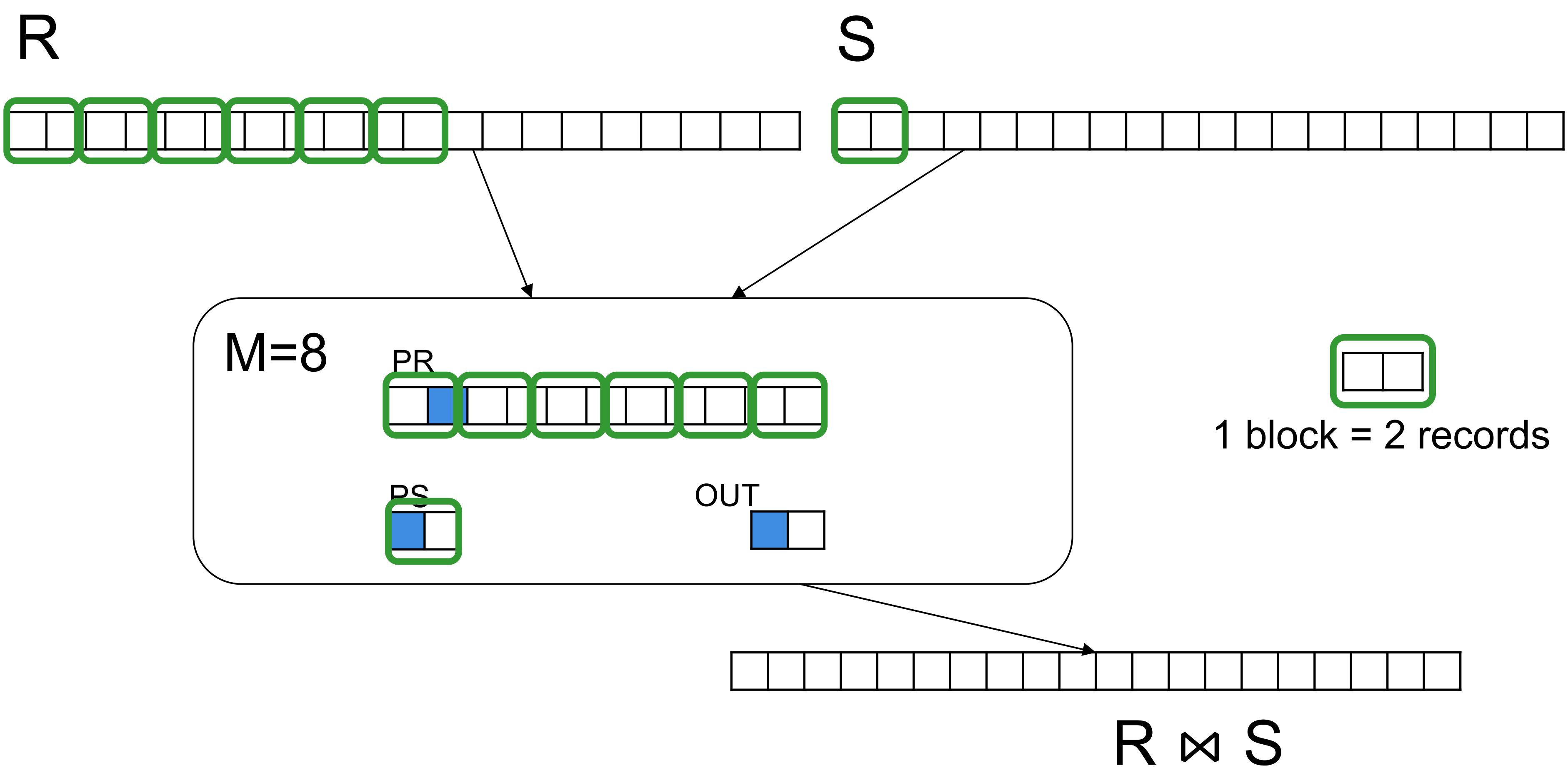
Block Nested Loop Join



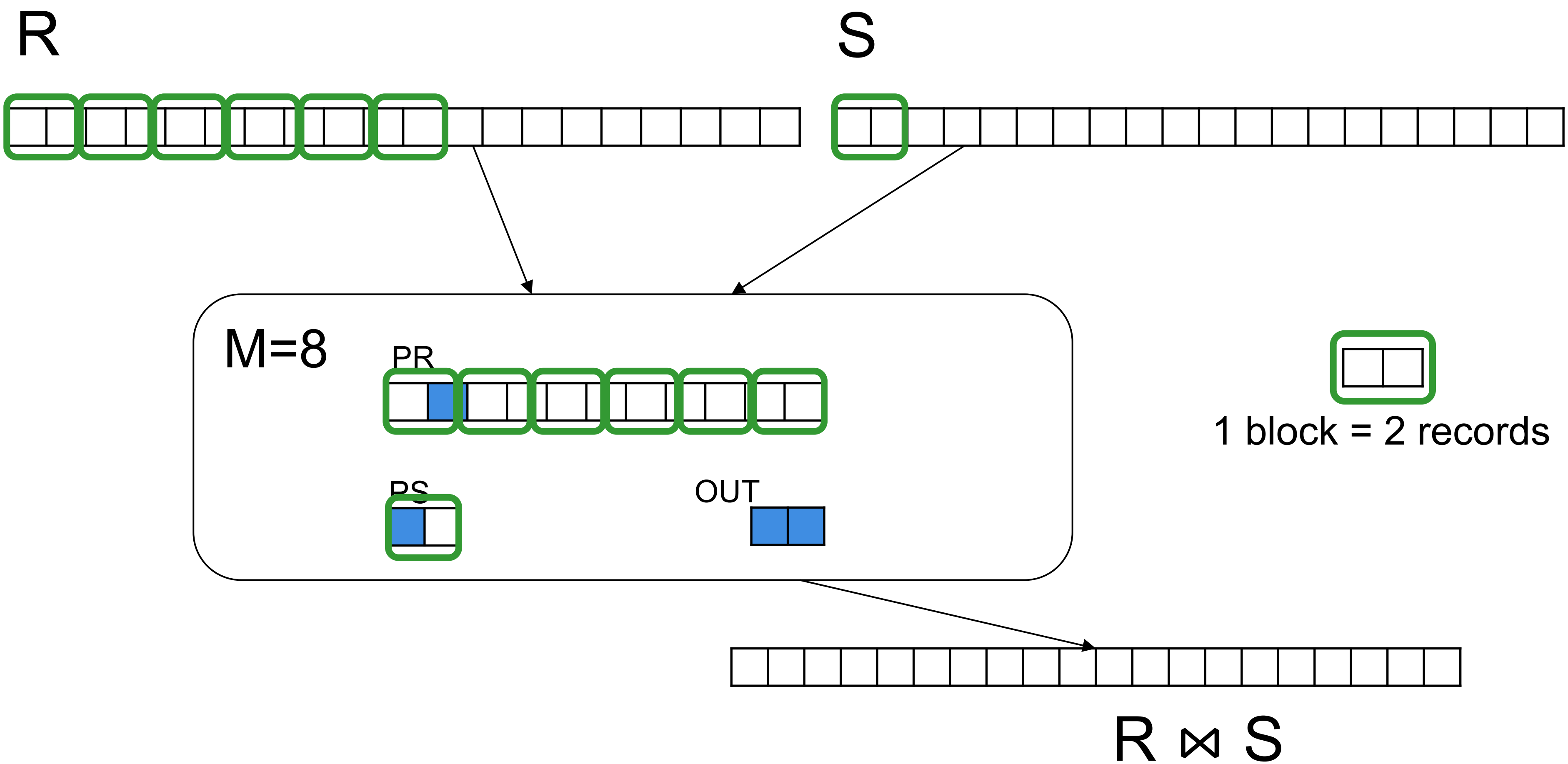
Block Nested Loop Join



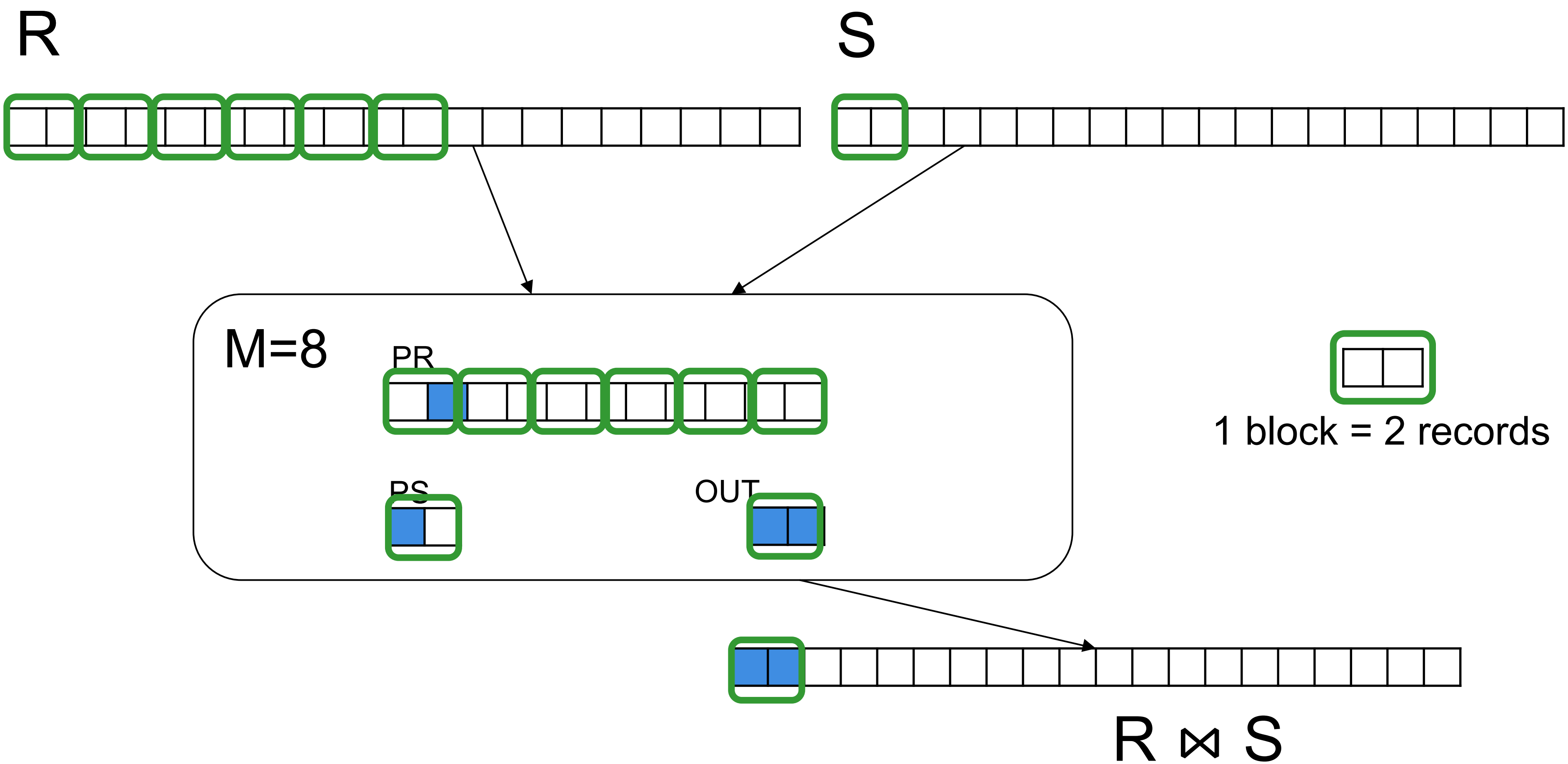
Block Nested Loop Join



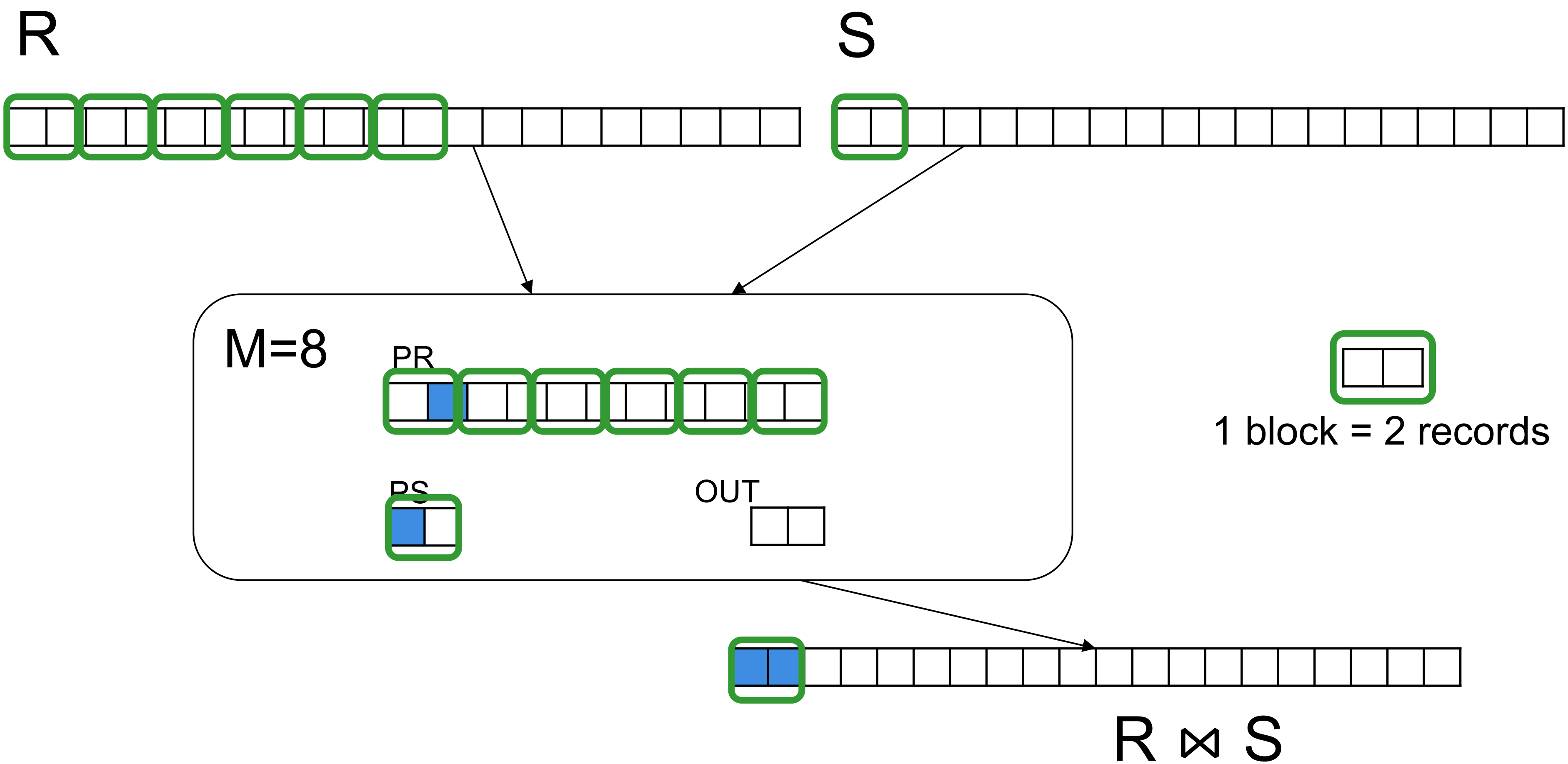
Block Nested Loop Join



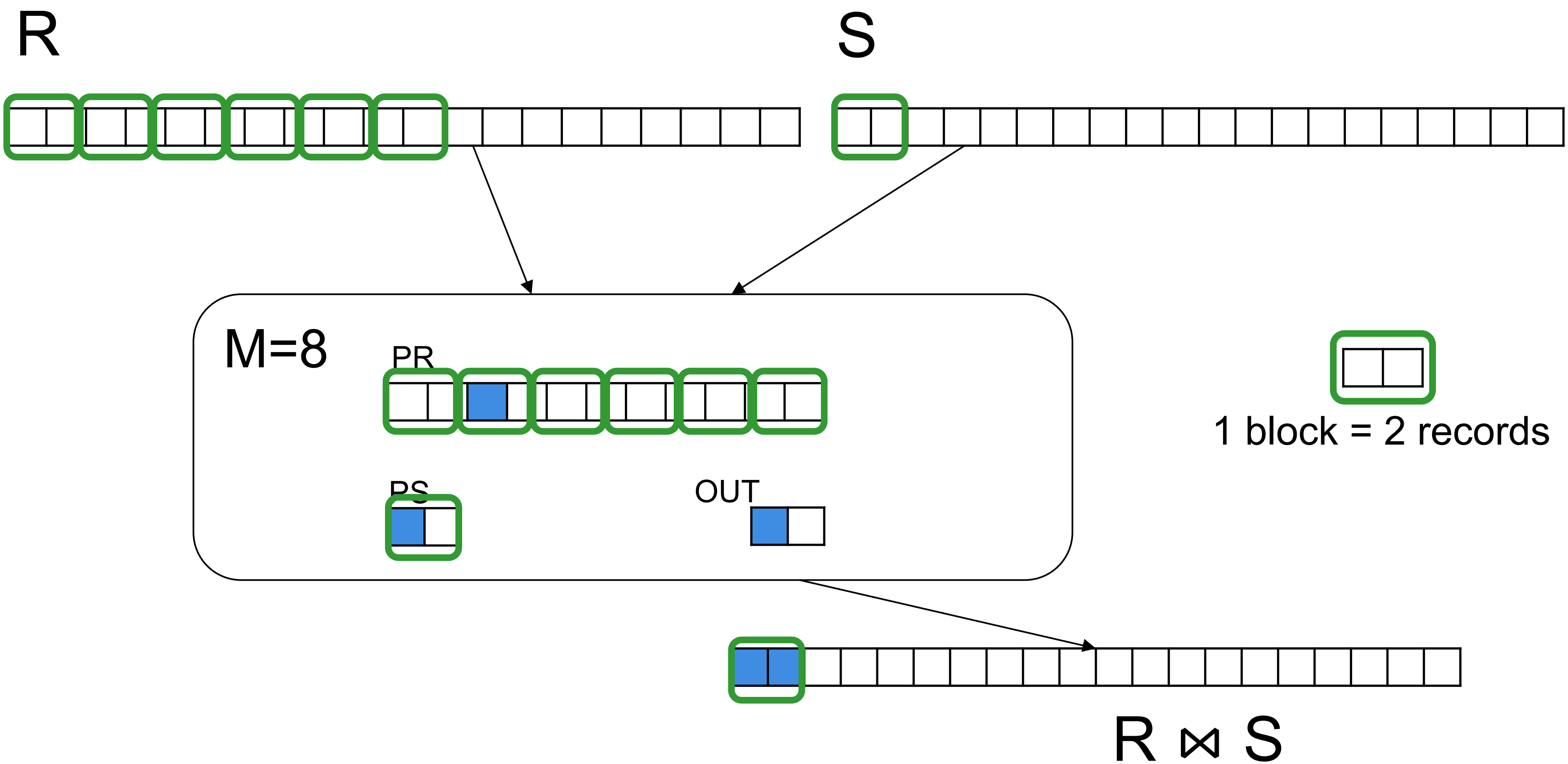
Block Nested Loop Join



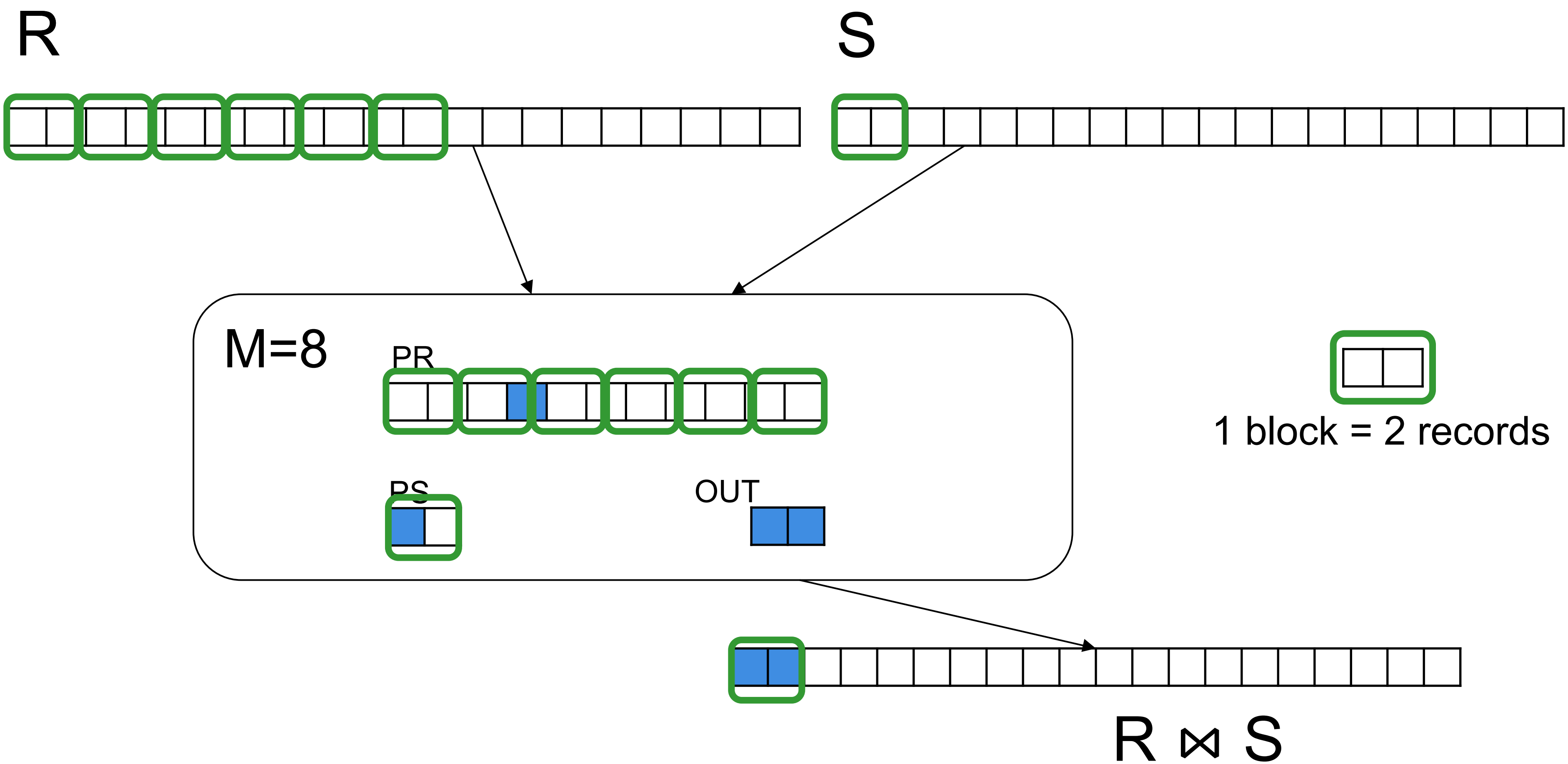
Block Nested Loop Join



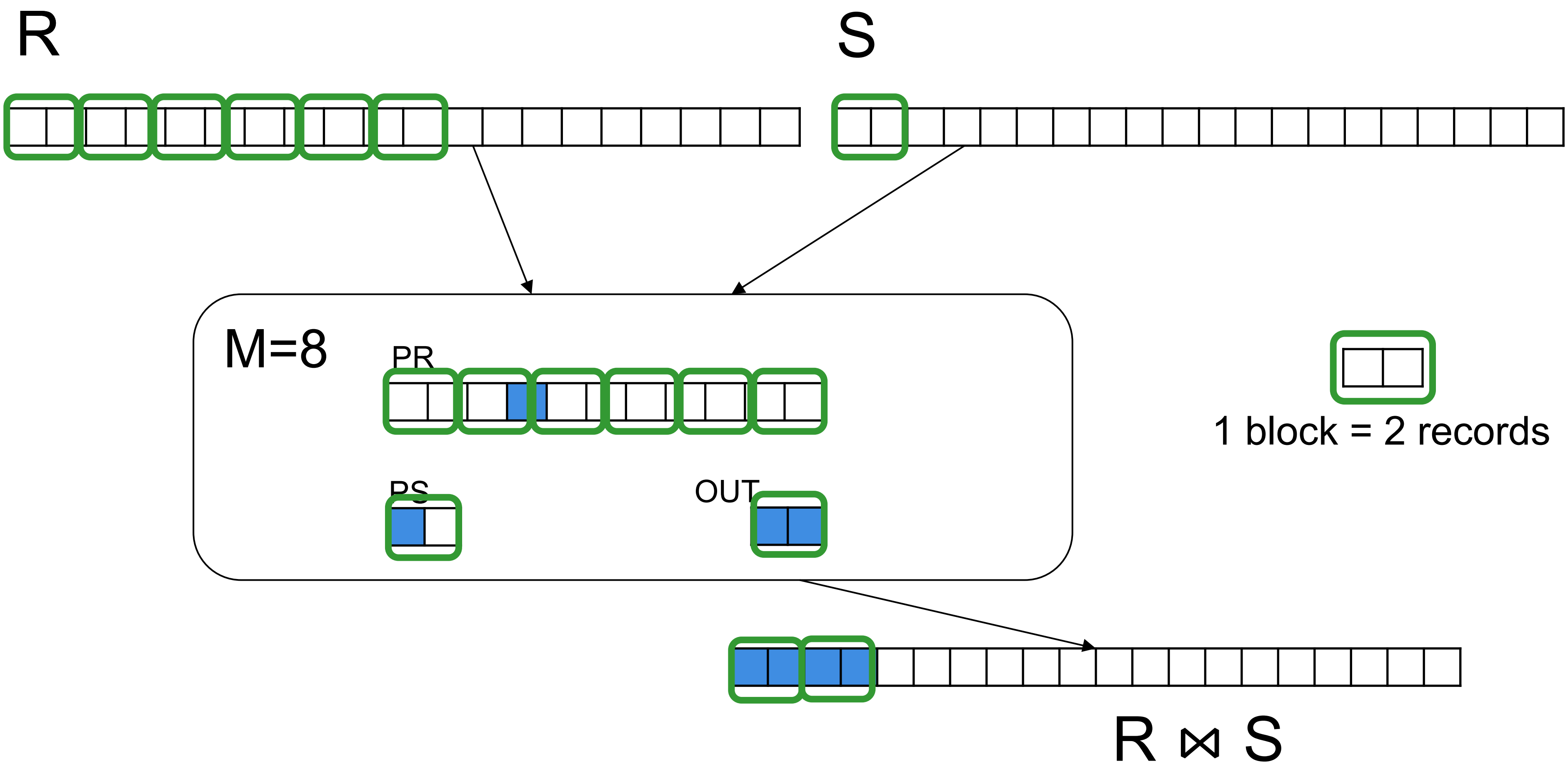
Block Nested Loop Join



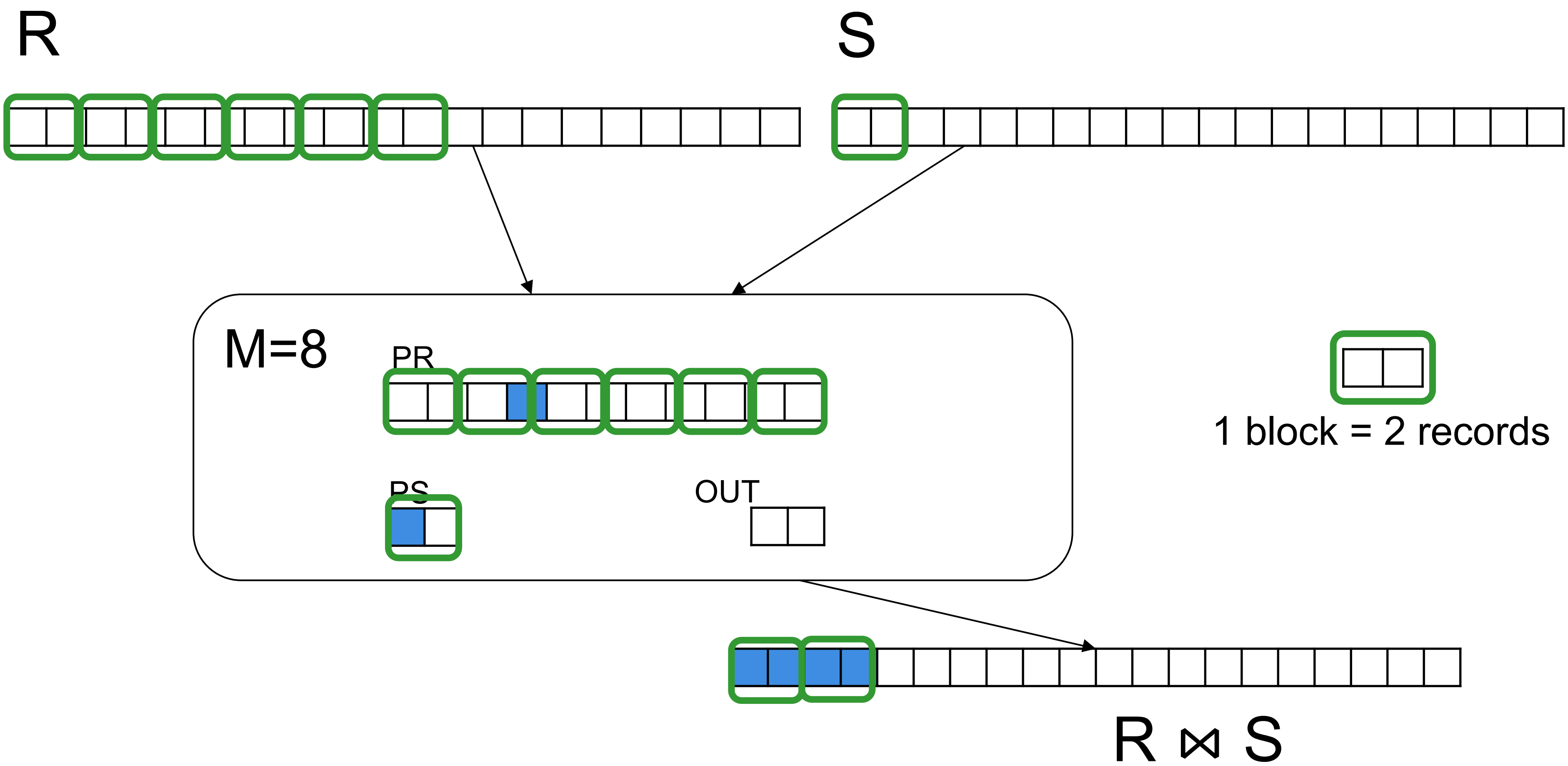
Block Nested Loop Join



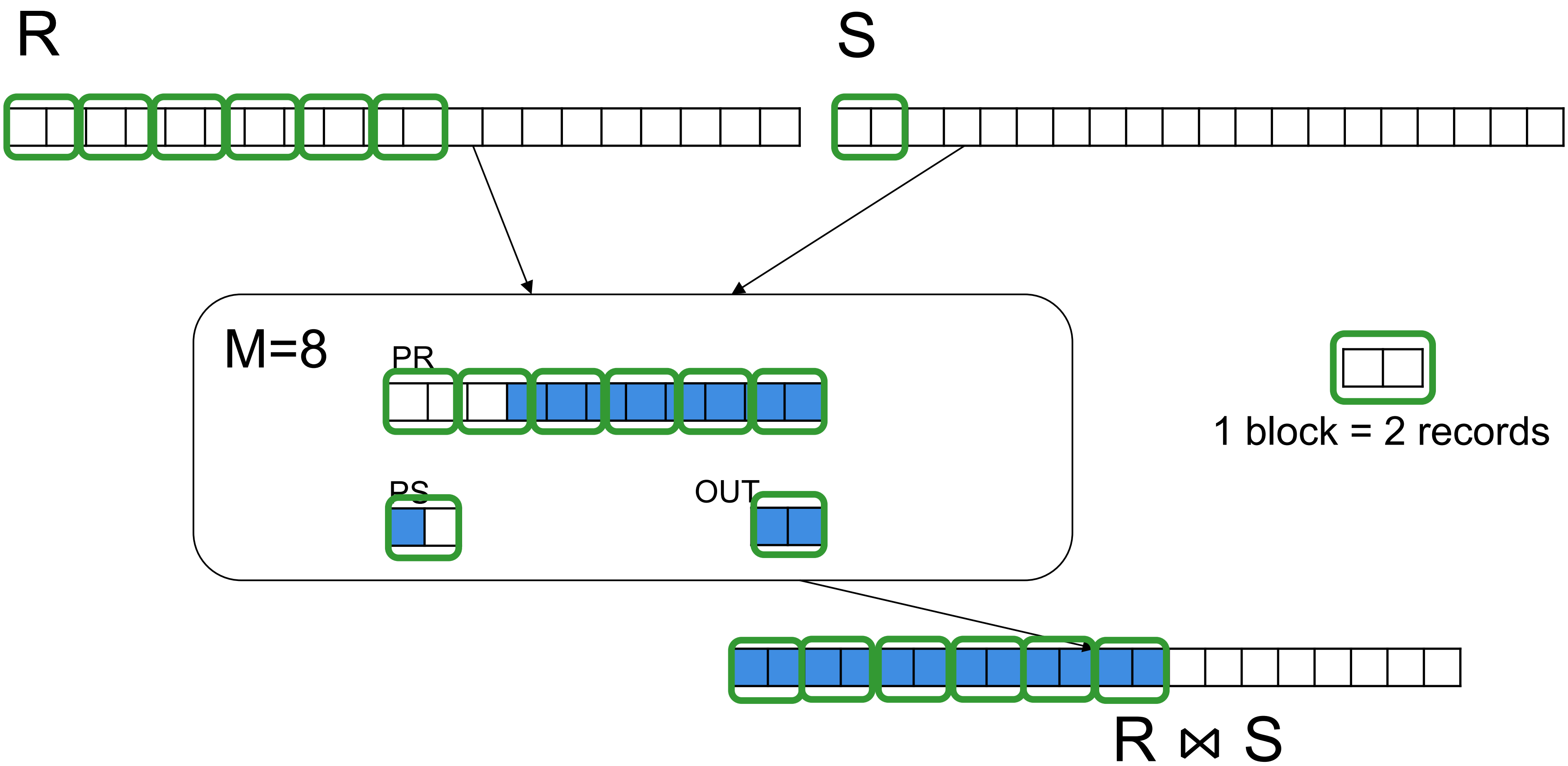
Block Nested Loop Join



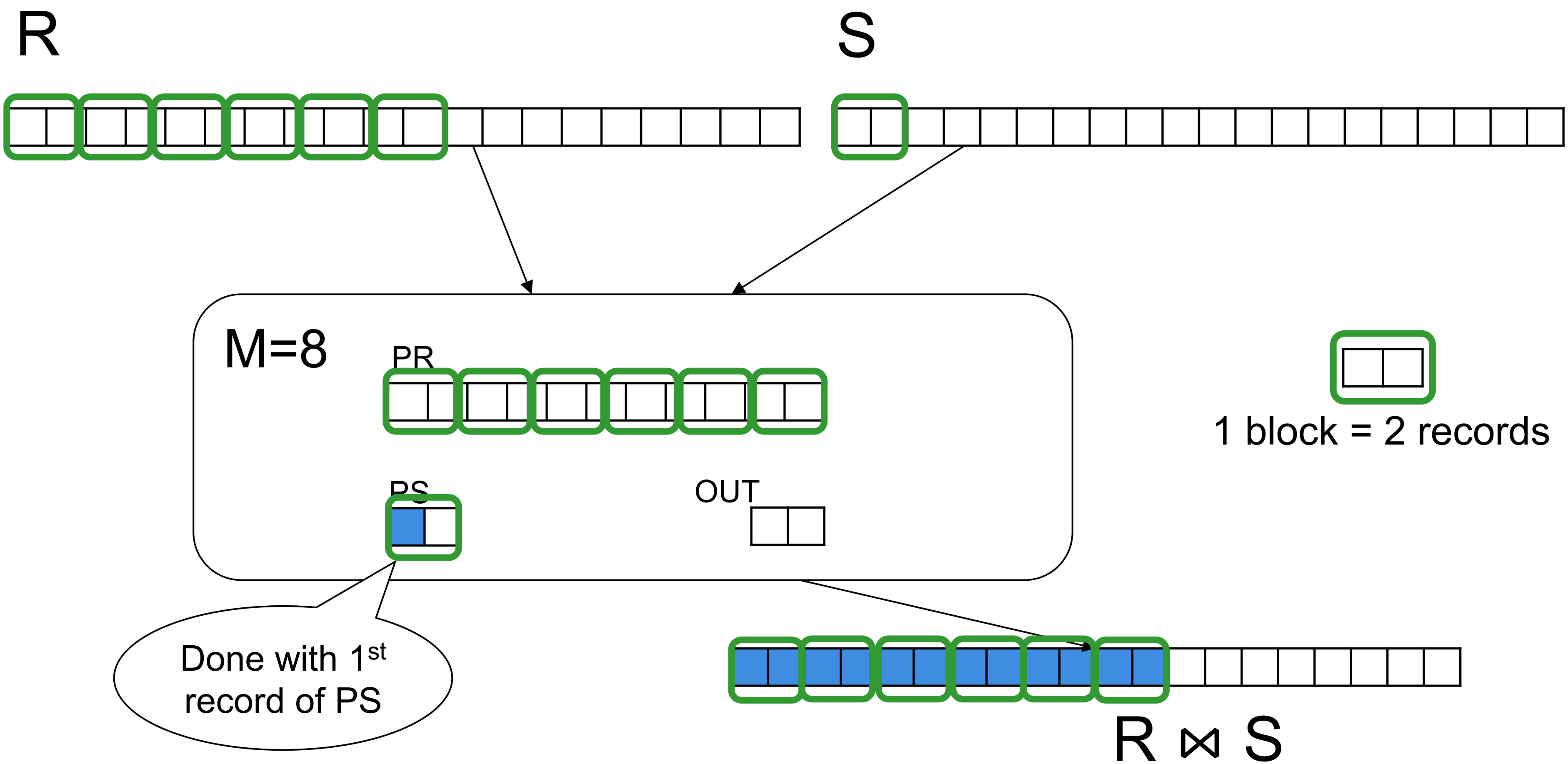
Block Nested Loop Join



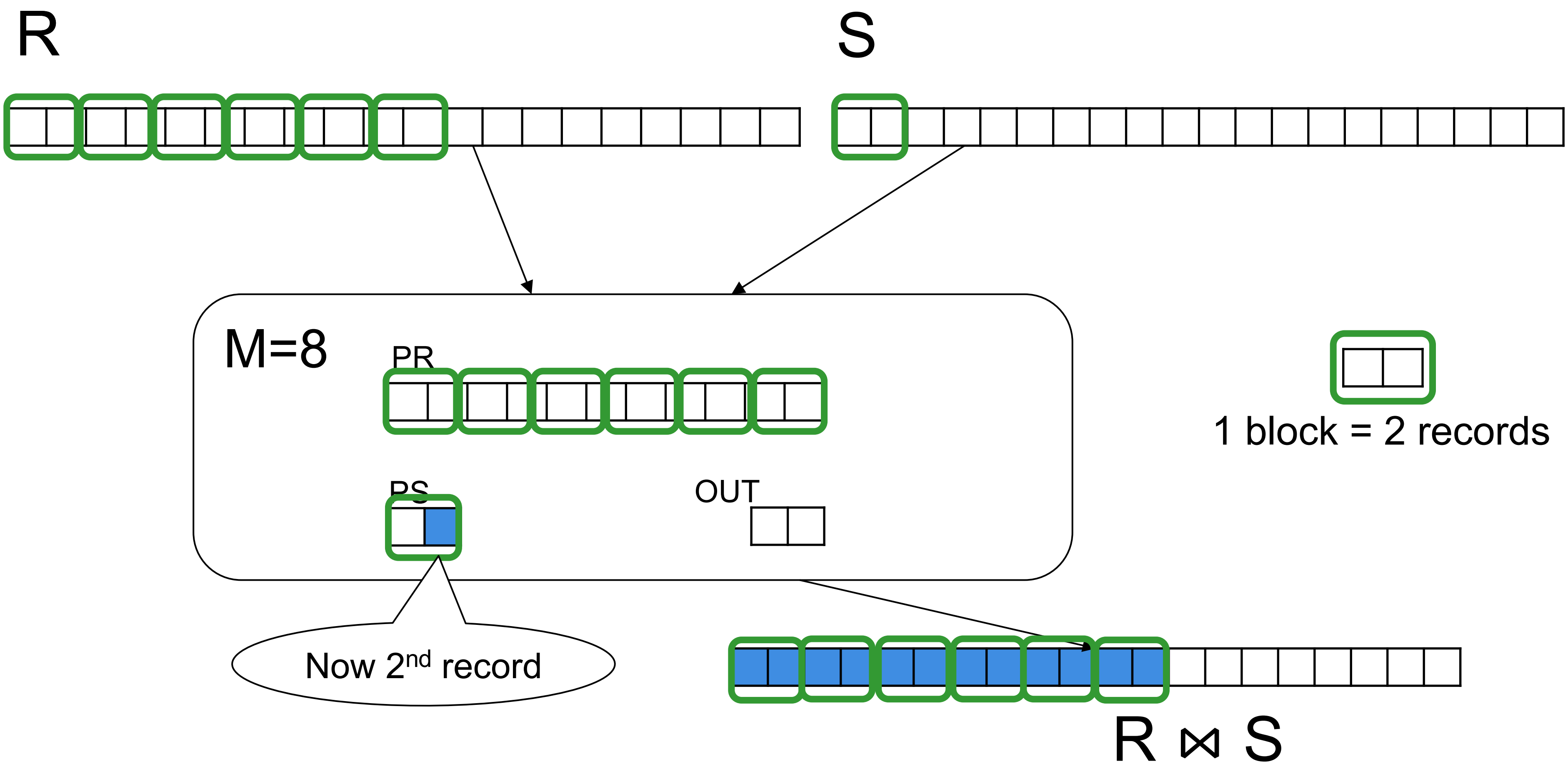
Block Nested Loop Join



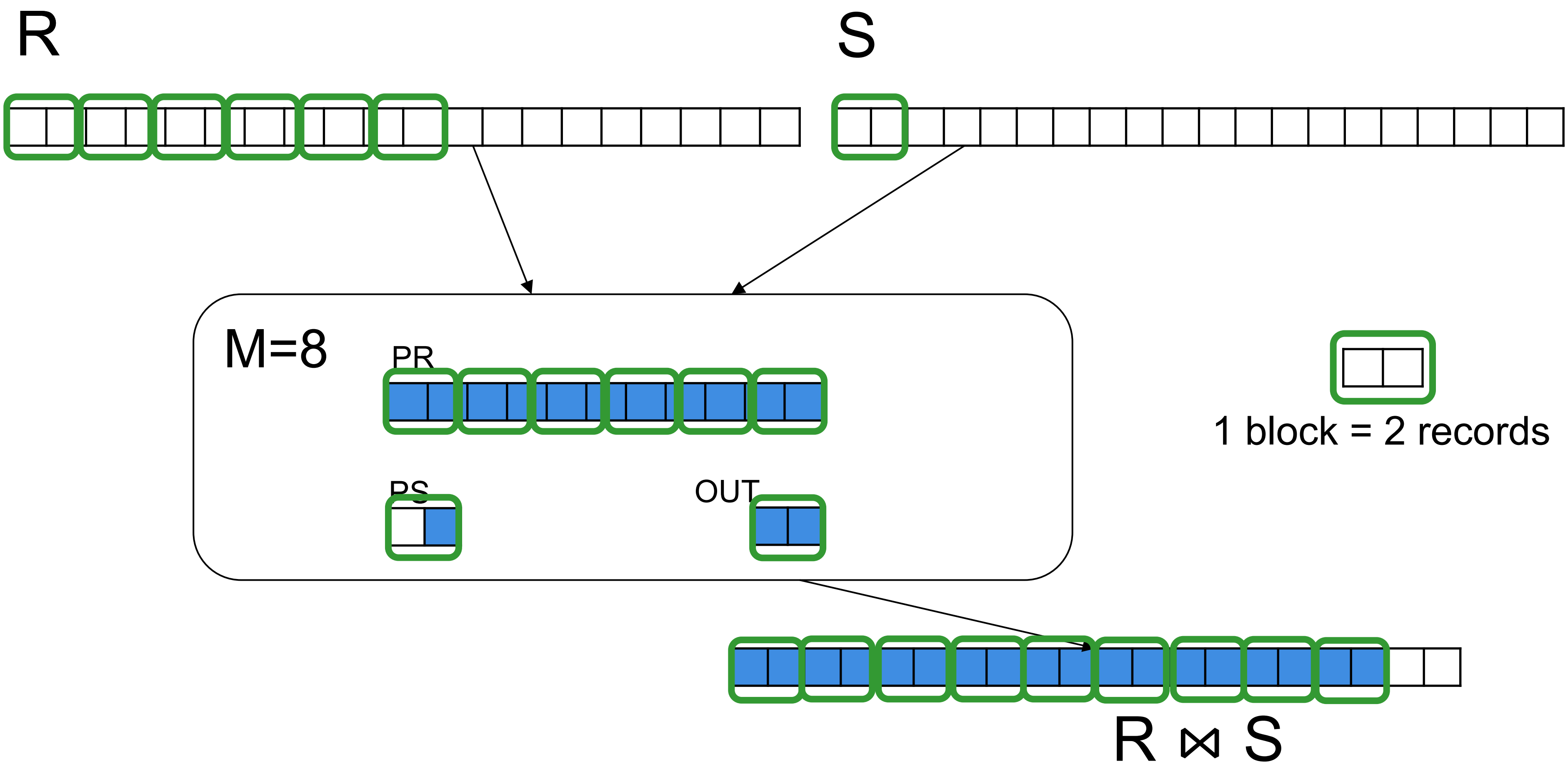
Block Nested Loop Join



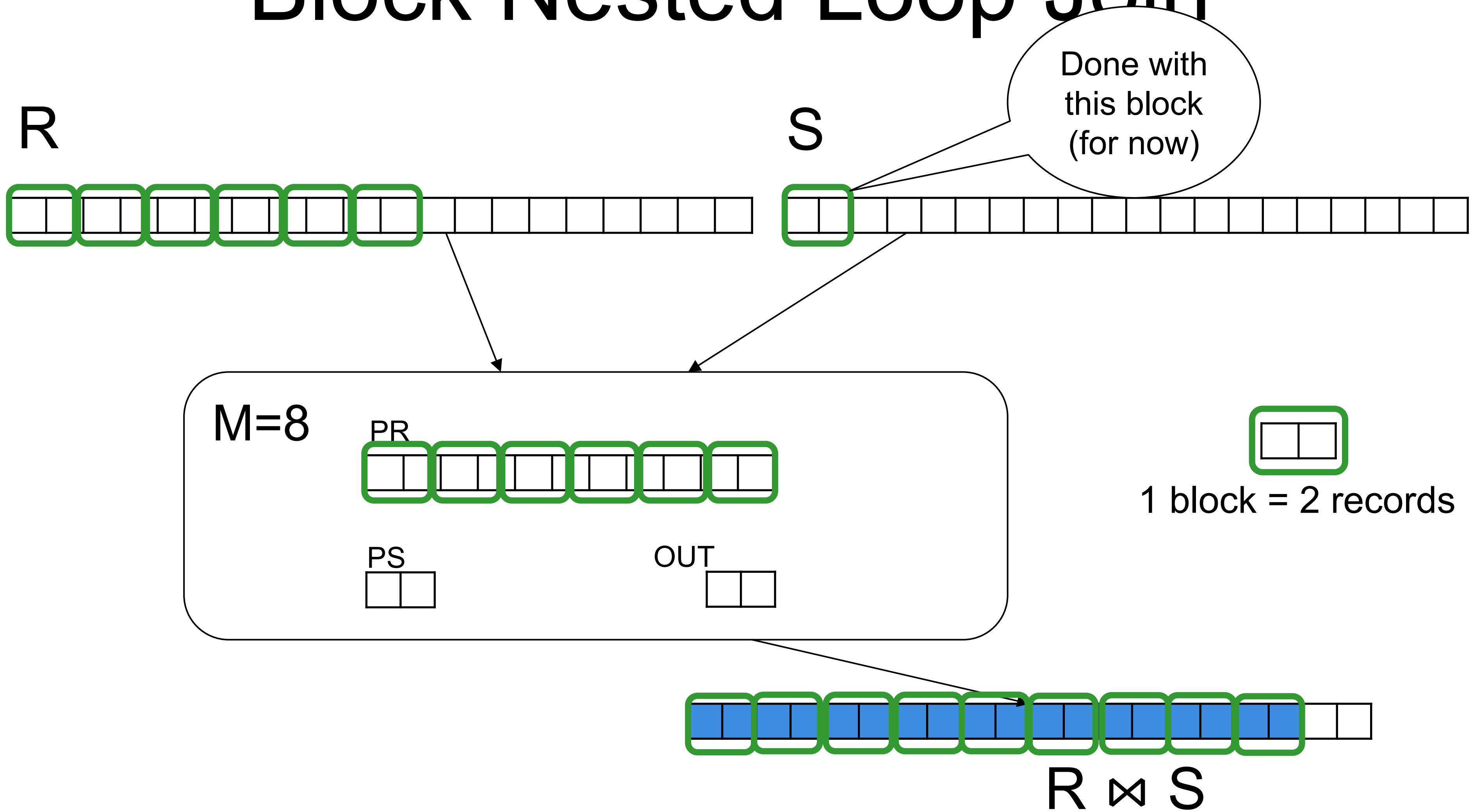
Block Nested Loop Join



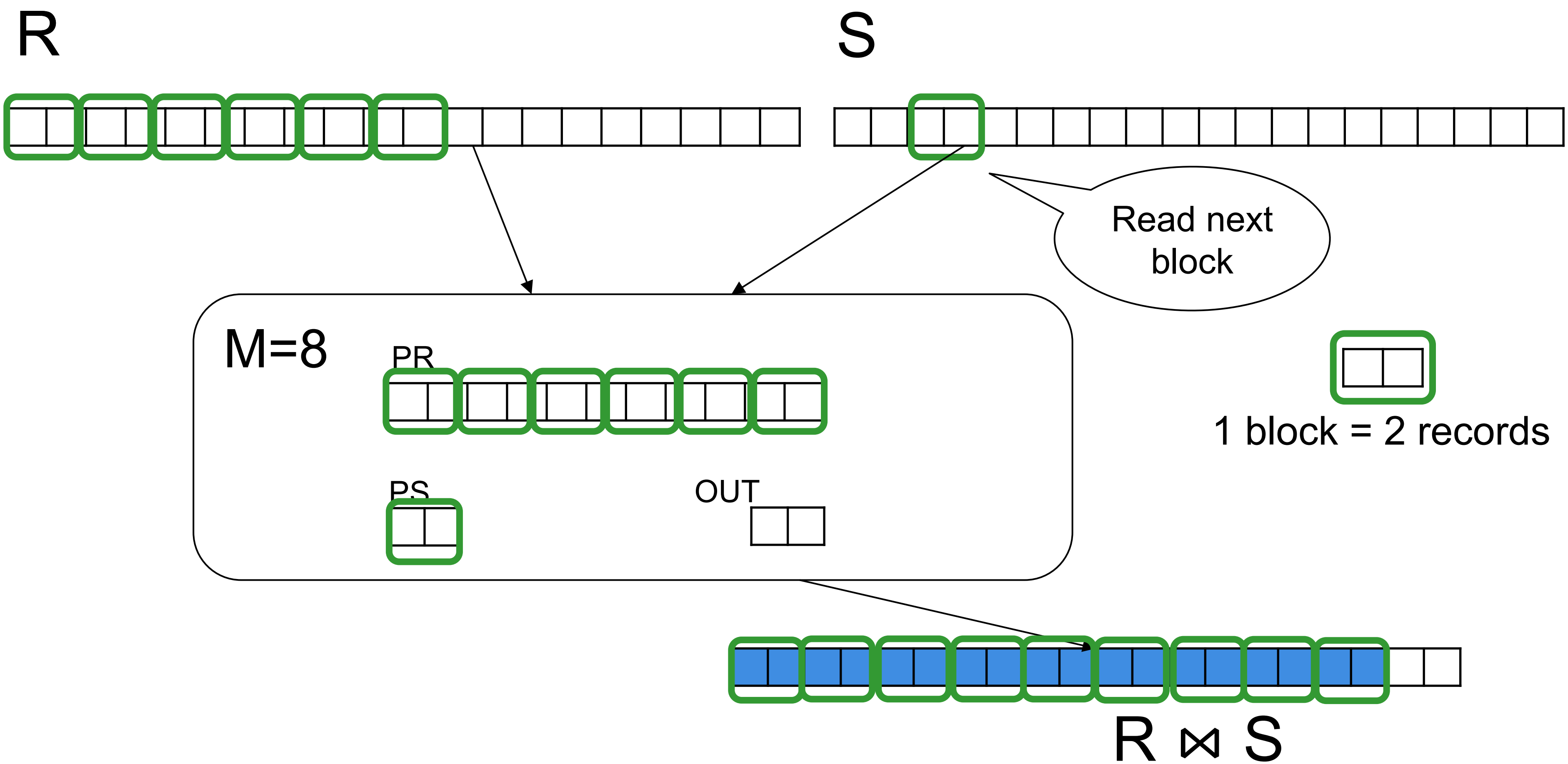
Block Nested Loop Join



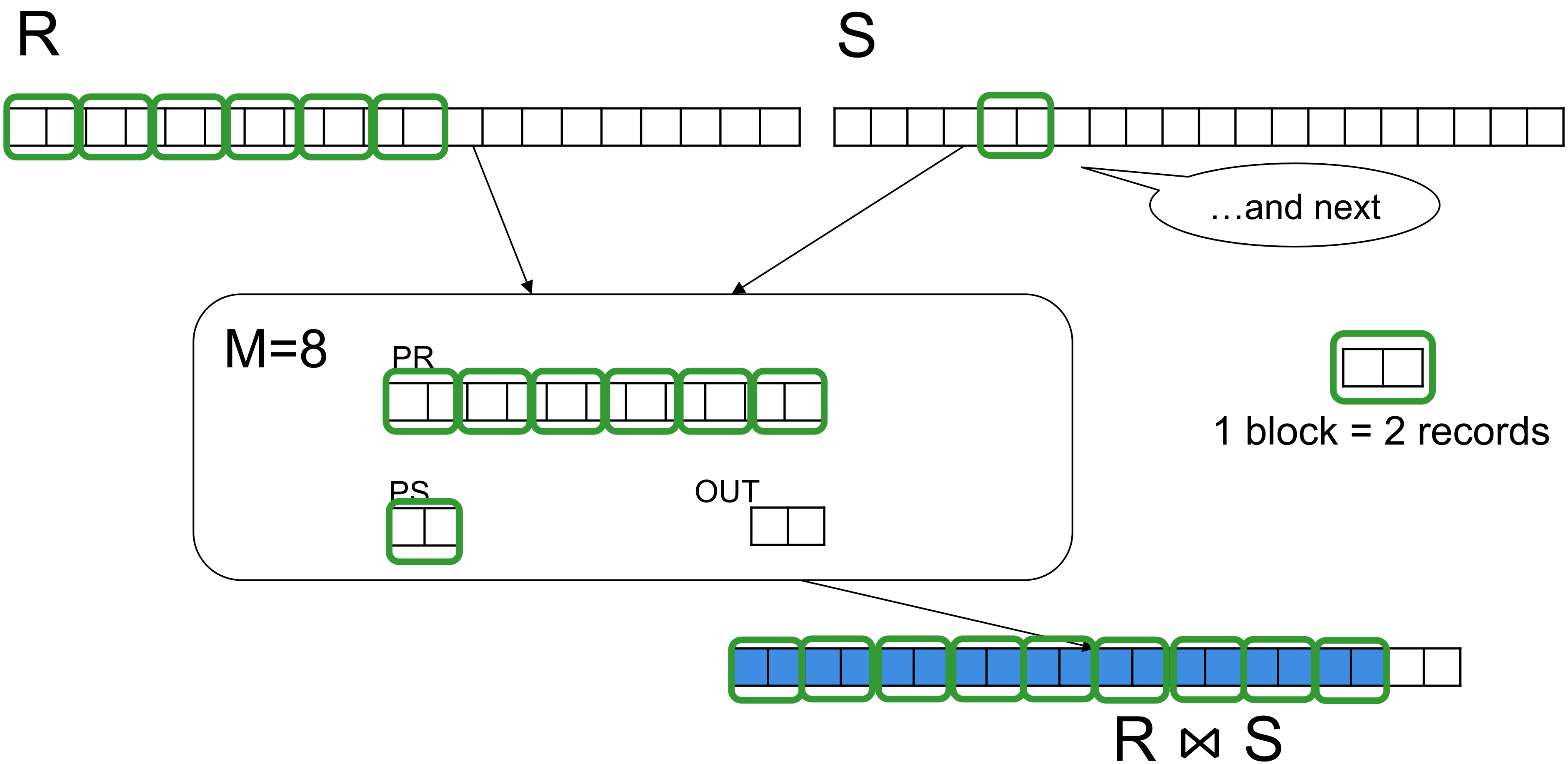
Block Nested Loop Join



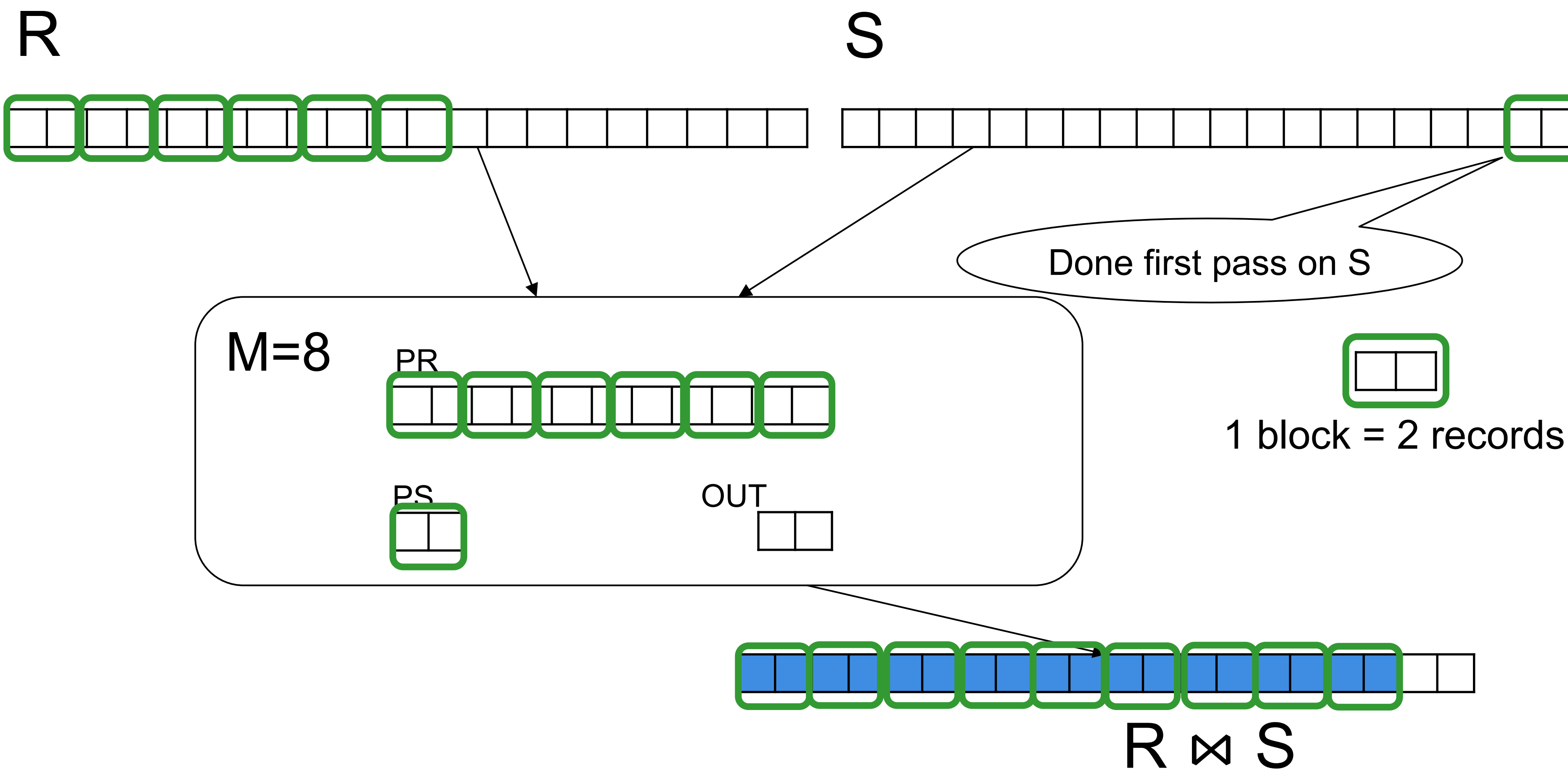
Block Nested Loop Join



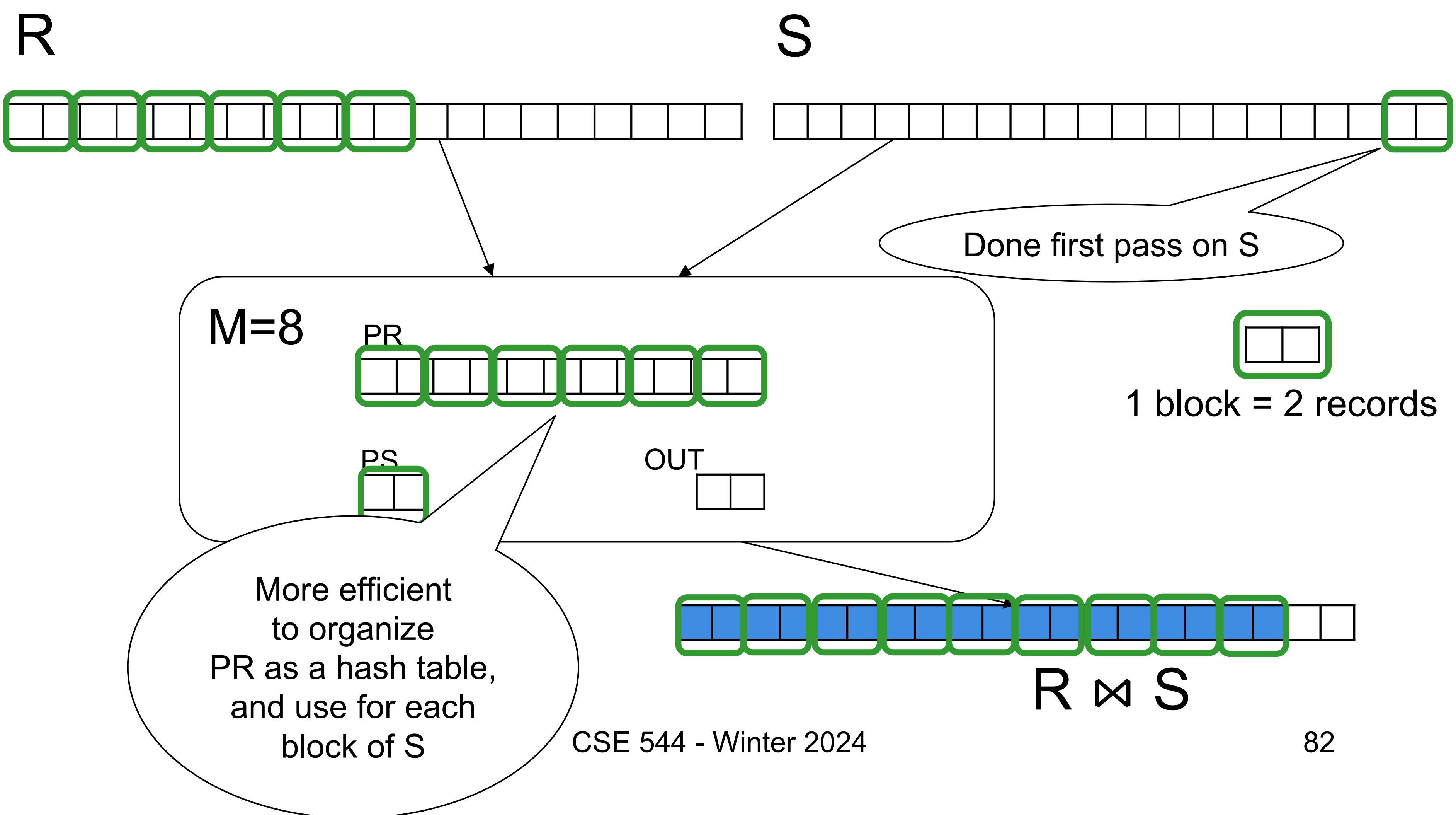
Block Nested Loop Join



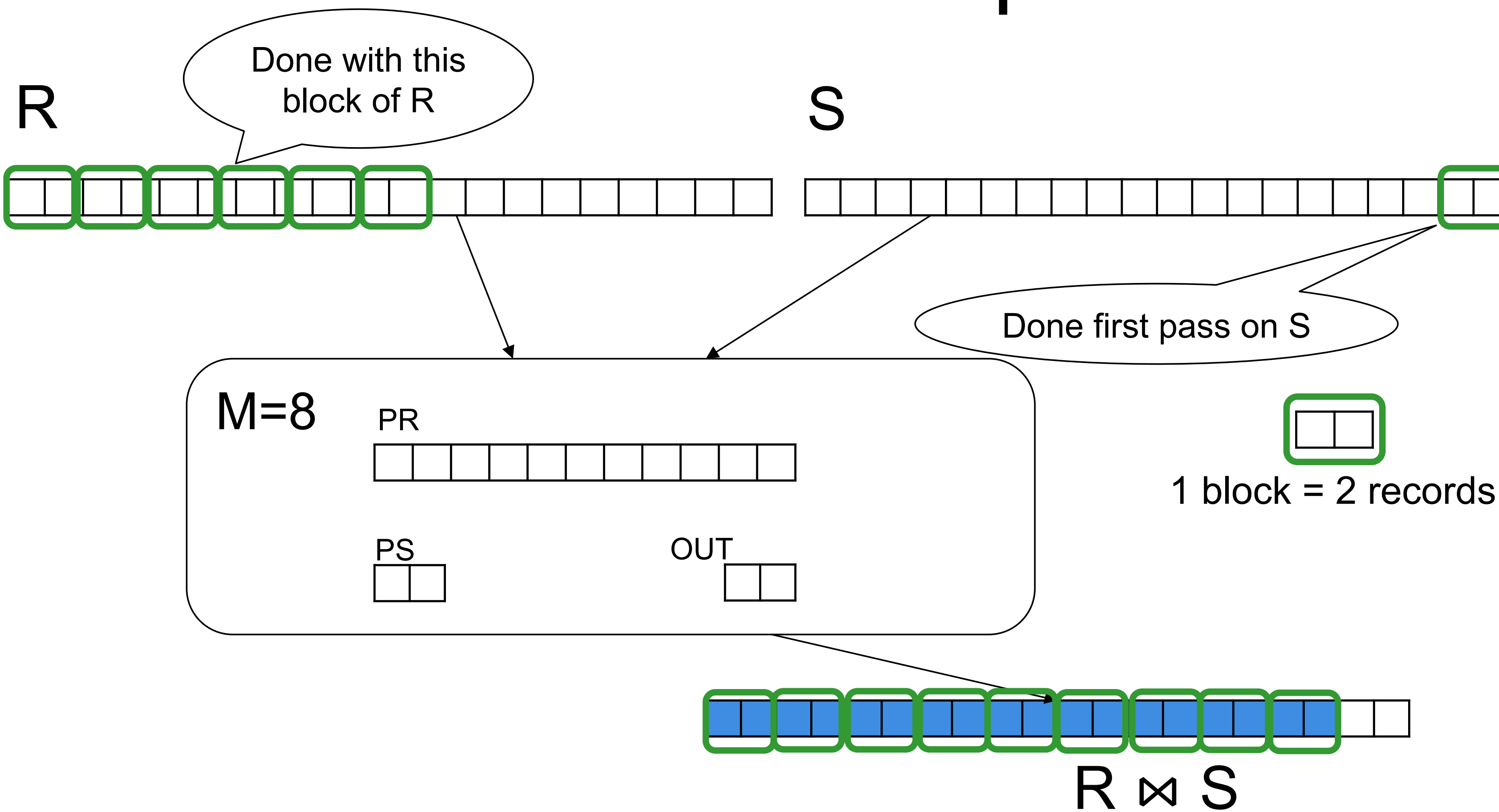
Block Nested Loop Join



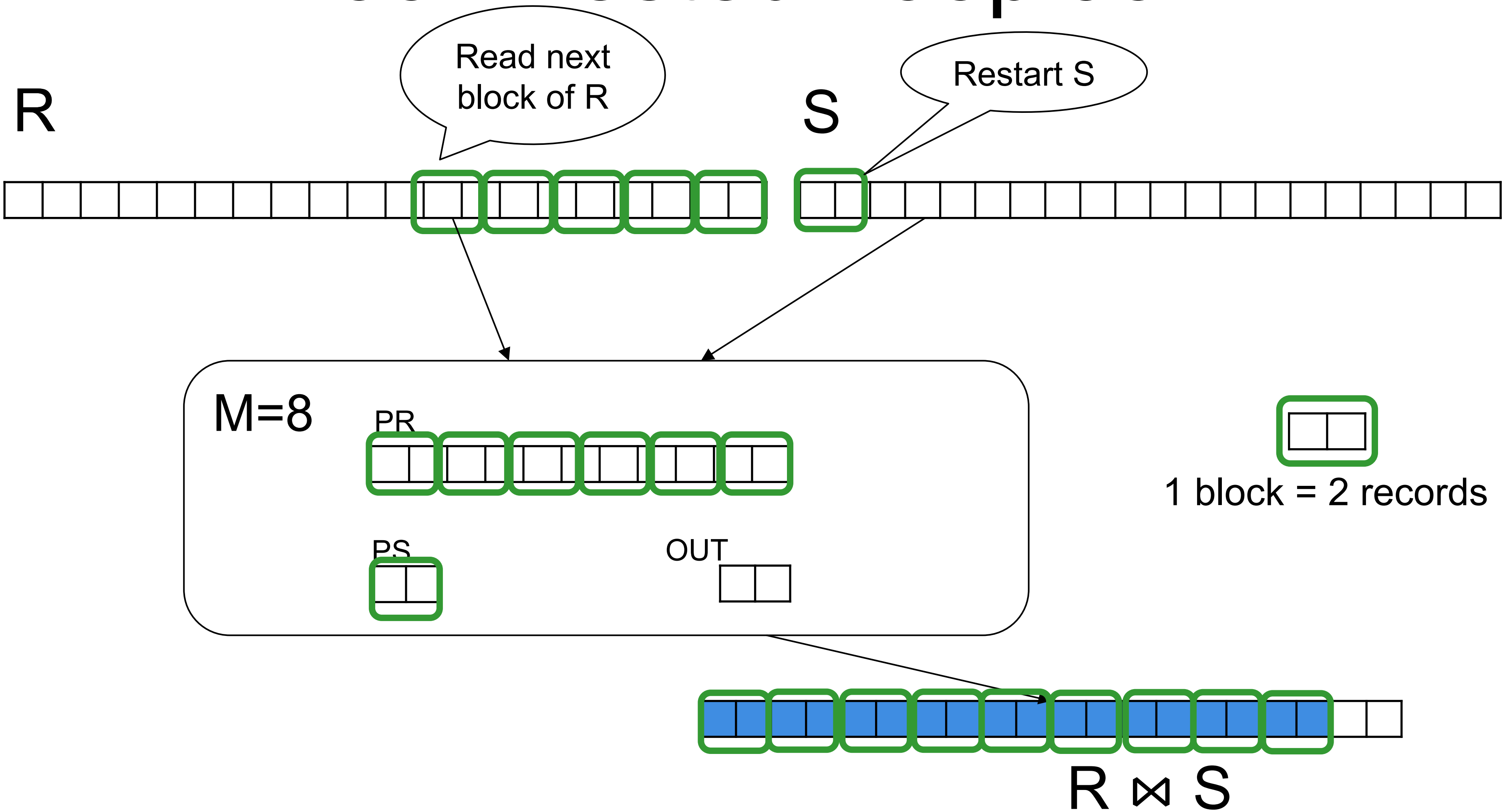
Block Nested Loop Join



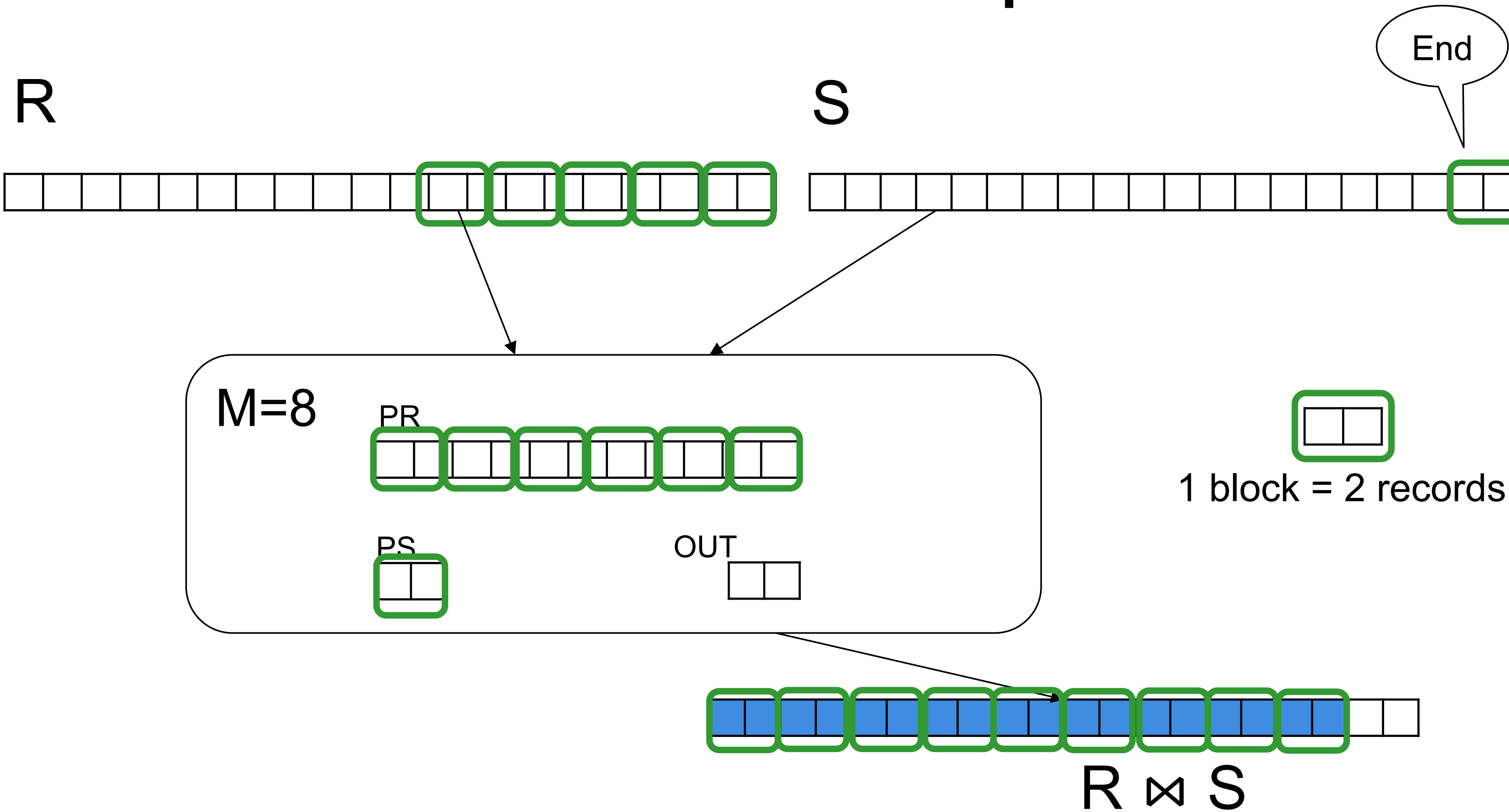
Block Nested Loop Join



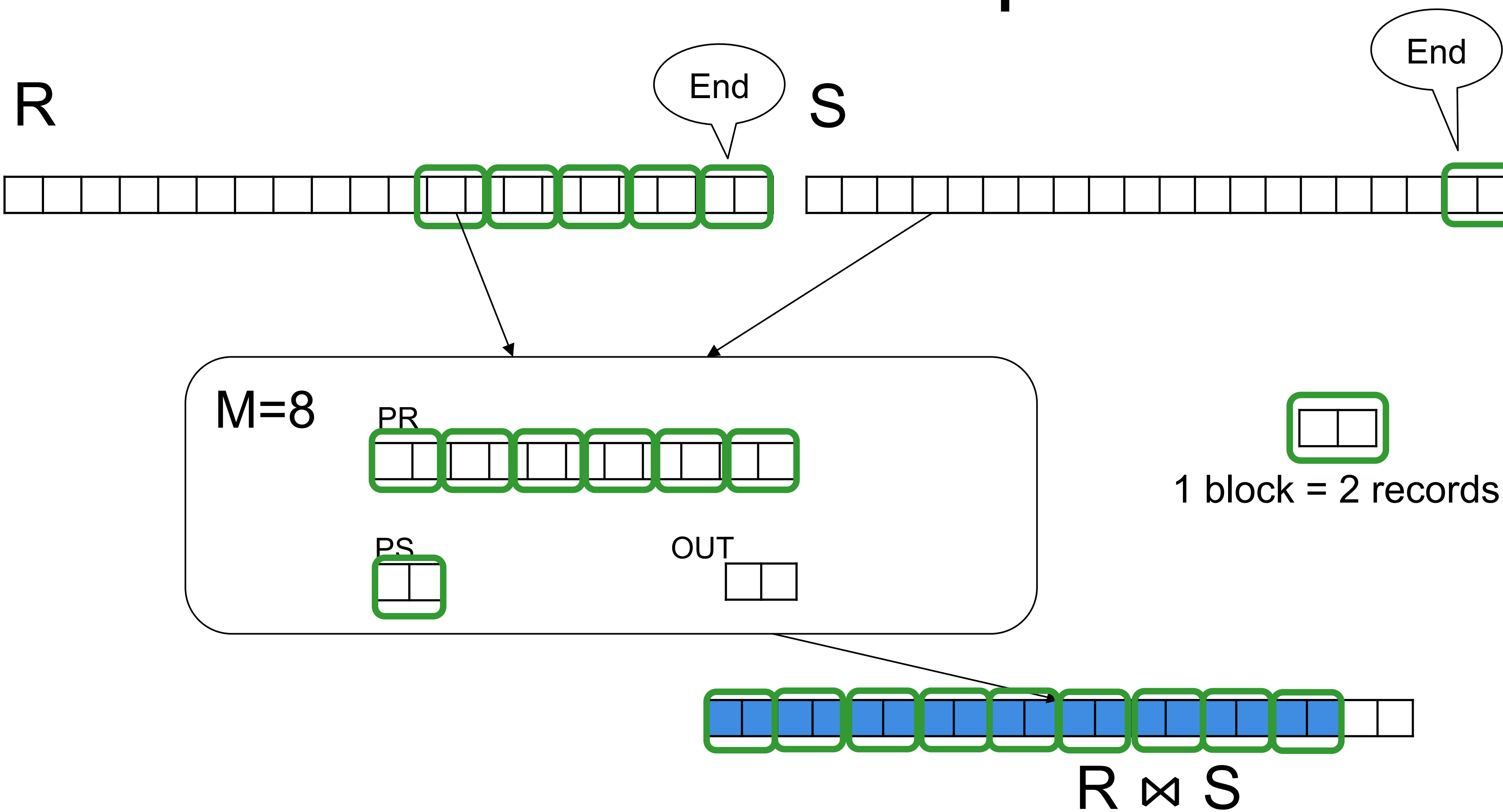
Block Nested Loop Join



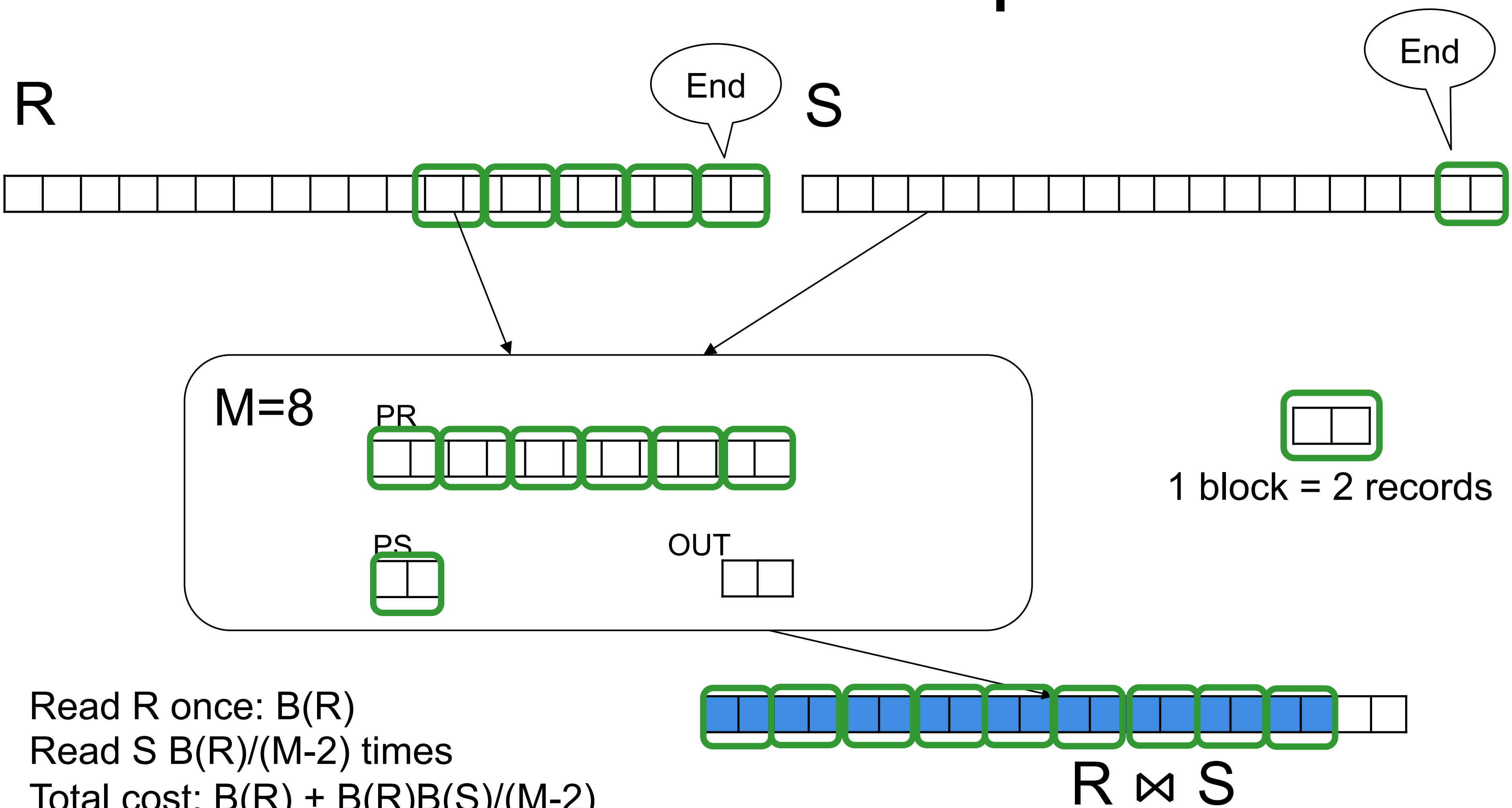
Block Nested Loop Join



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

Merge Sort, Merge Join

Merge-Sort

Merge-sort reads/writes sequentially

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

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

Multi-Way Merge-Sort

N-Way Merge Sort: use entire memory M

- Merge M-1 runs at once; run lengths: $(M\text{-}1), (M\text{-}1)^2, (M\text{-}2)^3, \ldots$
- Need $\log_{M\text{-}1} N$ reads/writes; $\log_{M\text{-}1} N \approx 2$

Cost = 3 B(R)

Ignore final write

Merging n Arrays

Merge n sorted arrays

Merge(A_1, A_2, …, A_n)   // A_1, …, A_n are sorted

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])
```

Merge(A_1, A_2, …, A_n) // A_1, …, A_n are sorted

i_1 =0; …, i_n = 0; j=0

while $i_1 \le N$ or … or $i_1 \le N$

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 ≤ 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++]
```

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++]
```

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++]
```

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++]
```

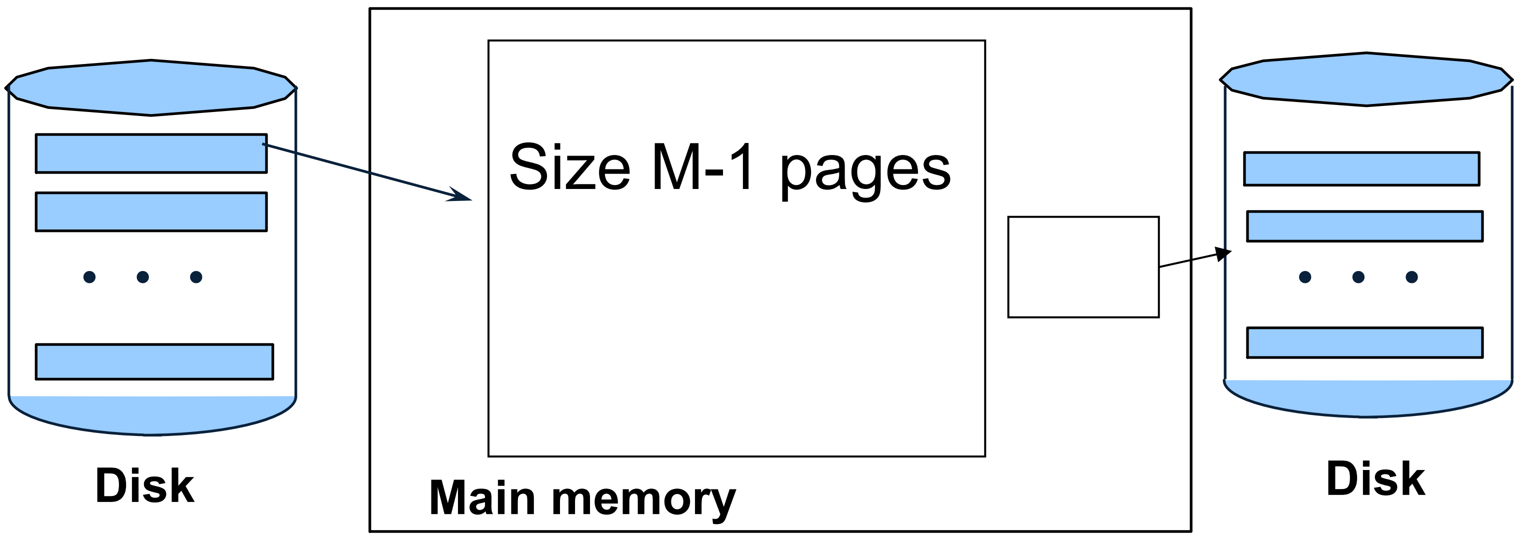
Important:
Need to read only one element from each array

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

Additional log n factor to find min using priority queue

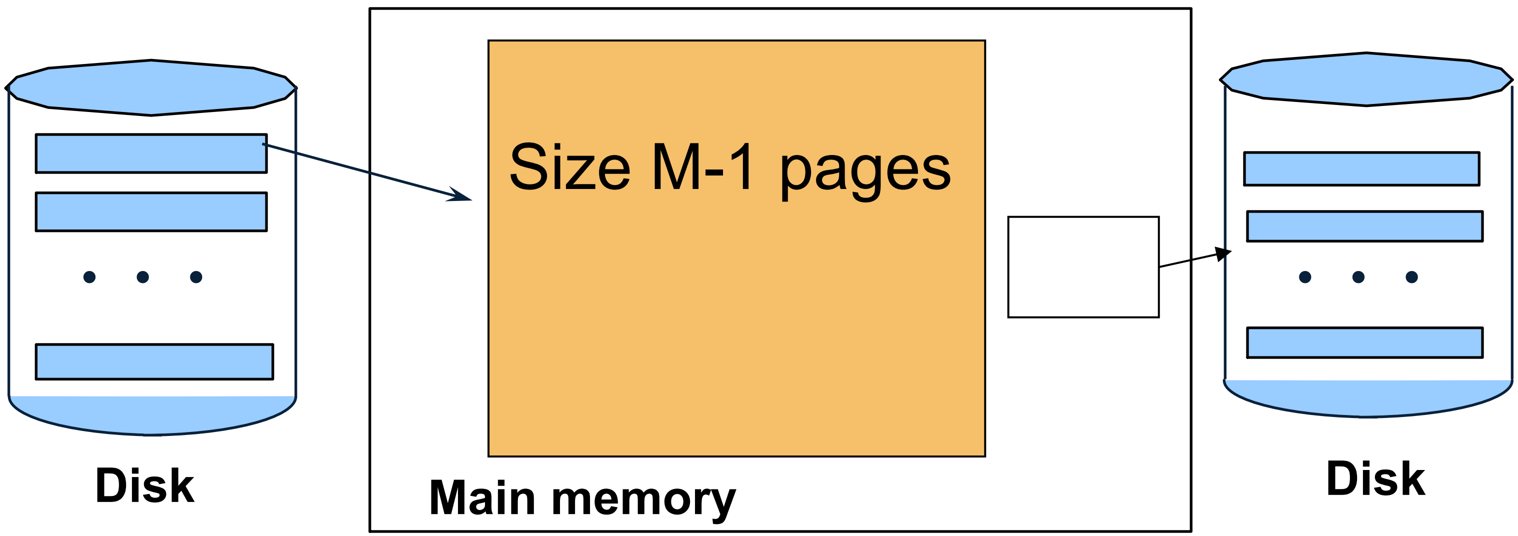
Merge-Sort: Step 1

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



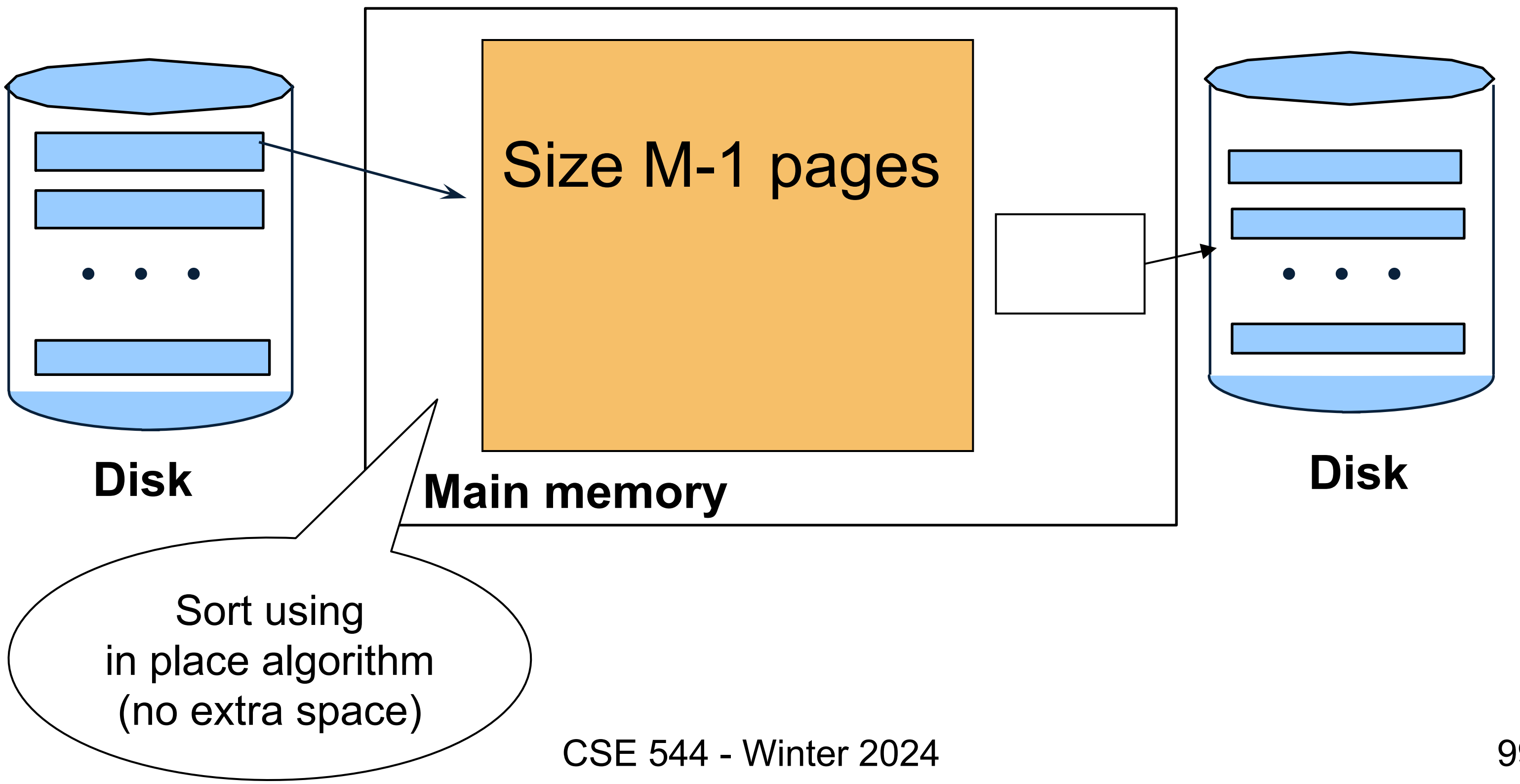
Merge-Sort: Step 1

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



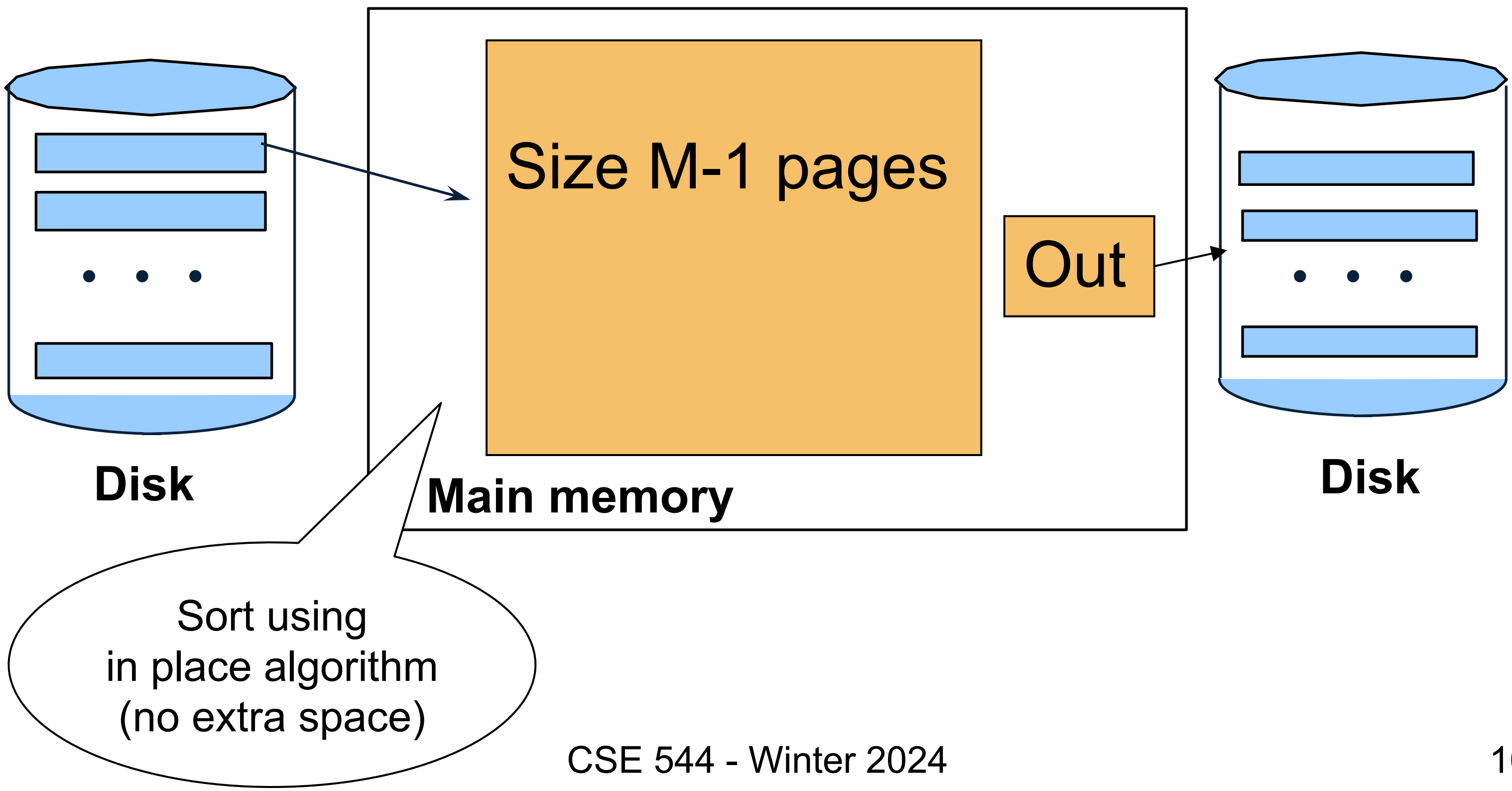
Merge-Sort: Step 1

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



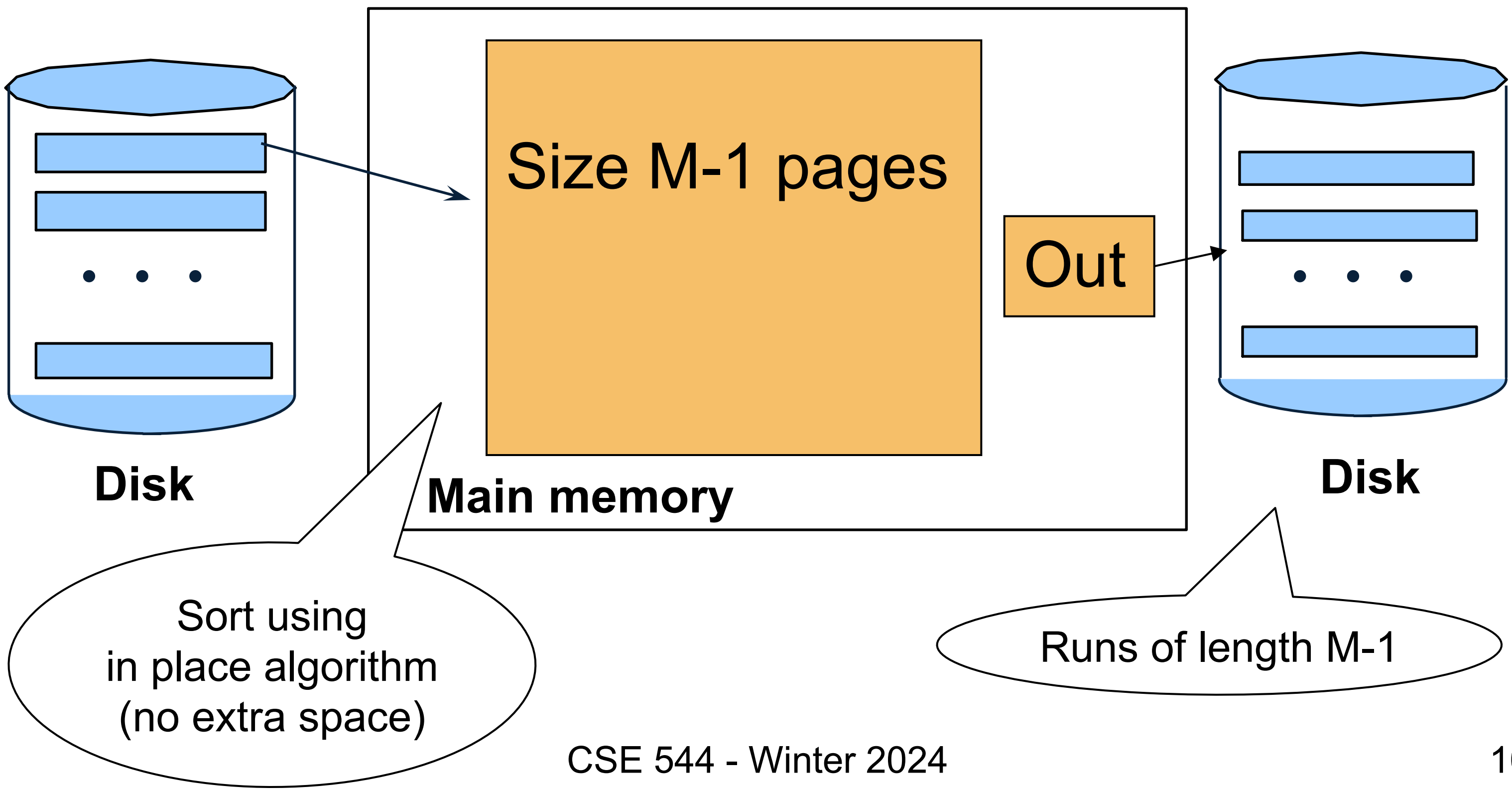
Merge-Sort: Step 1

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



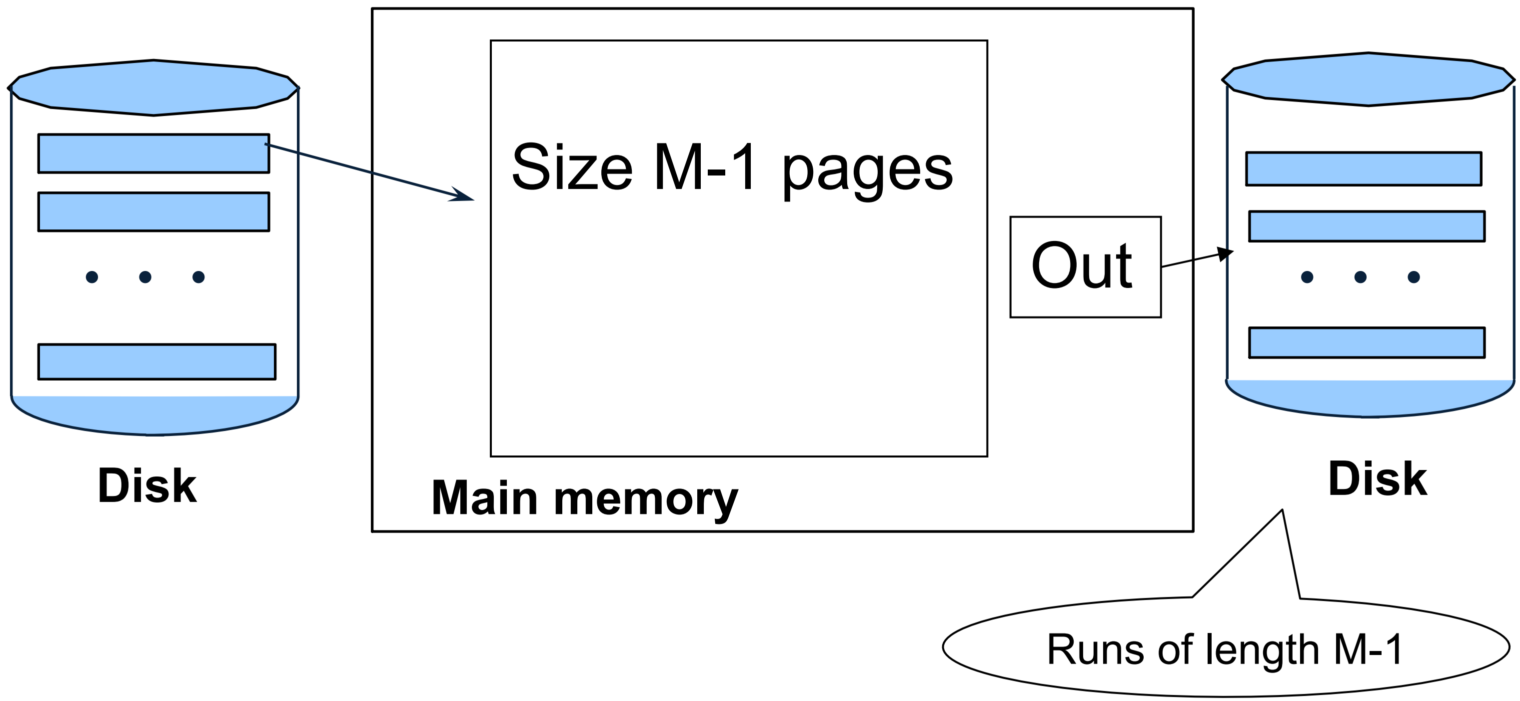
Merge-Sort: Step 1

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



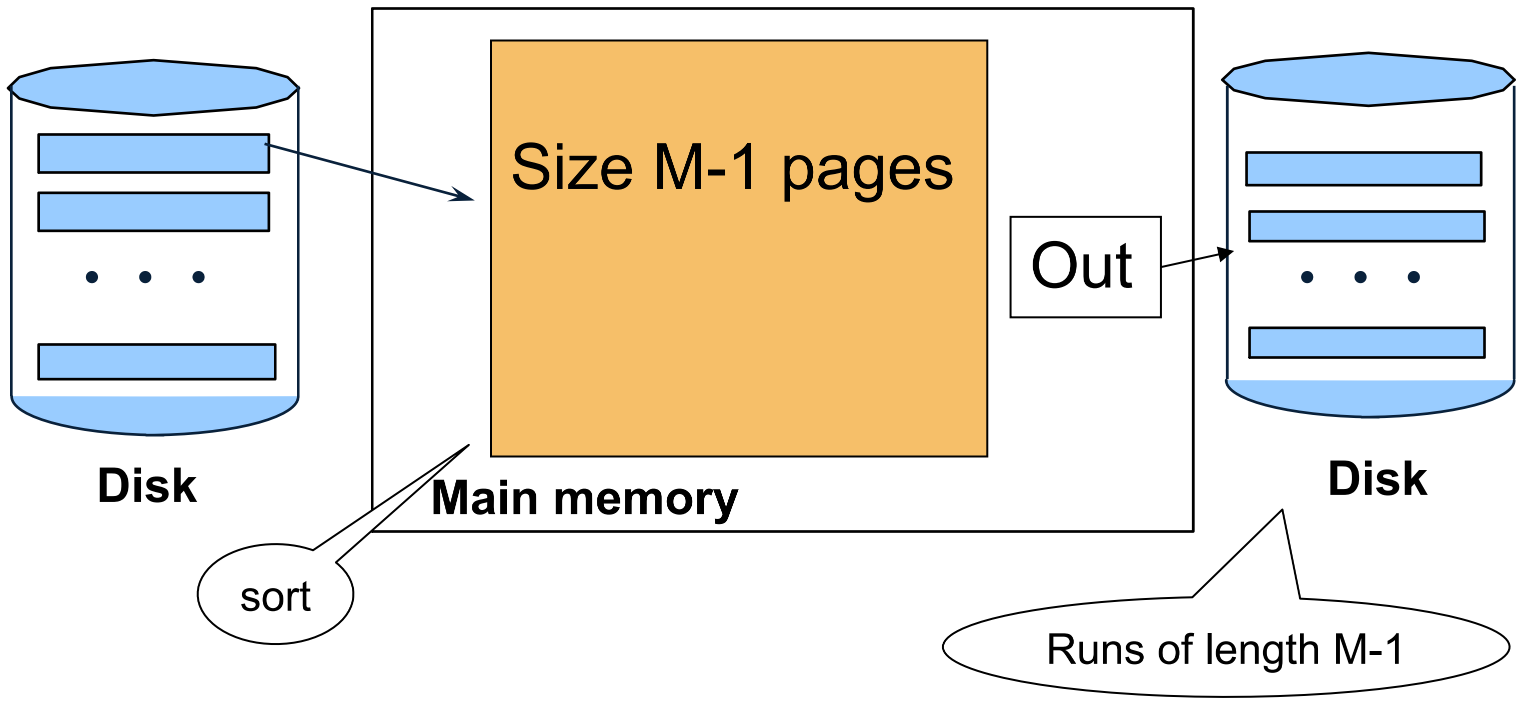
Merge-Sort: Step 1

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



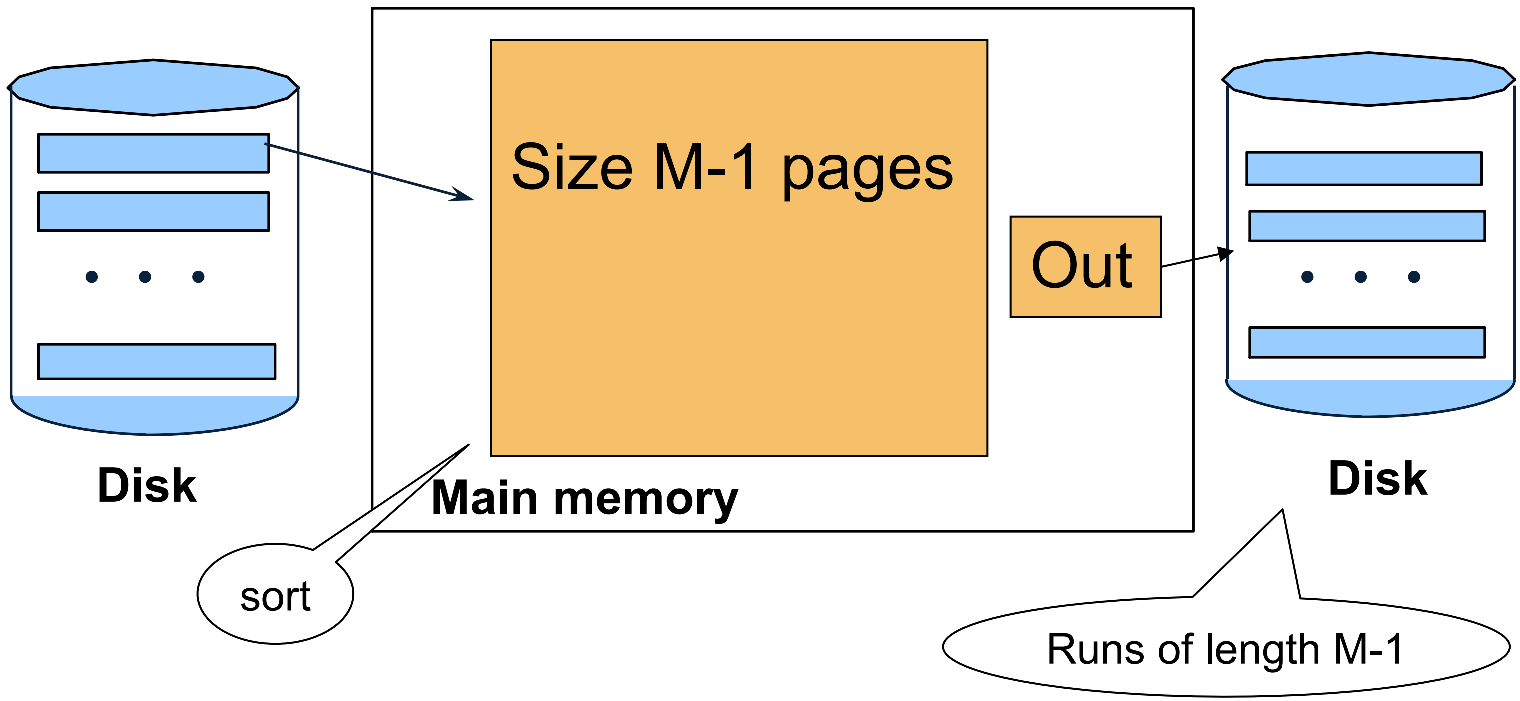
Merge-Sort: Step 1

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



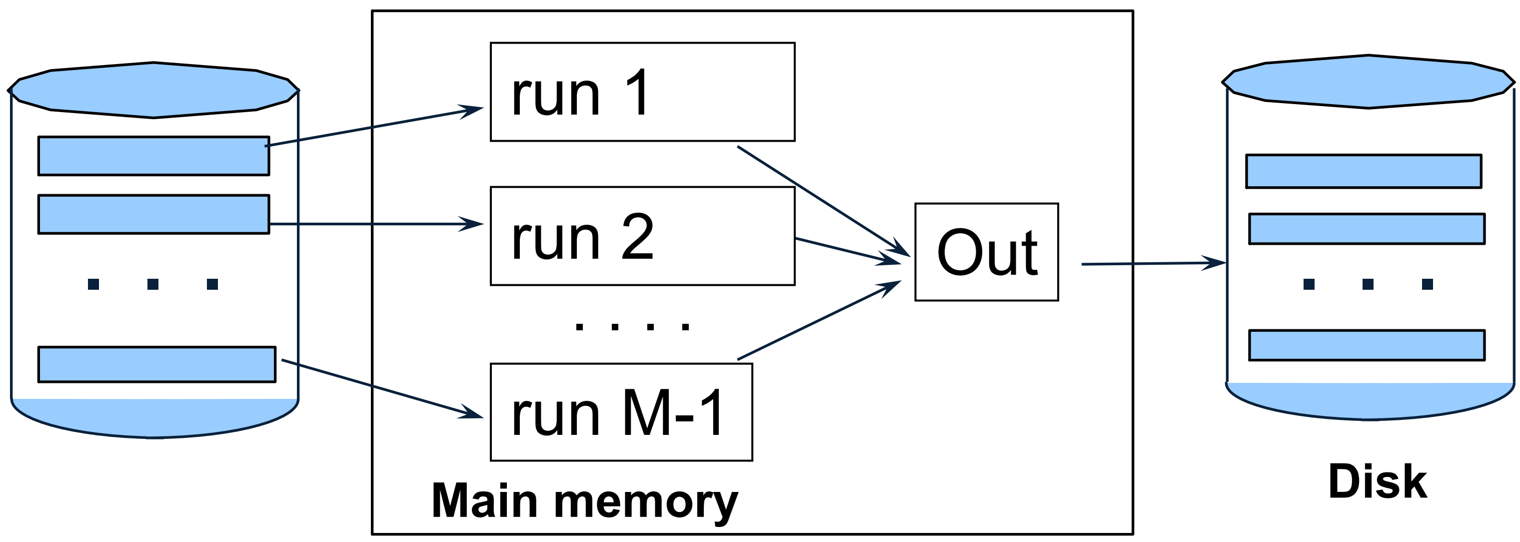
Merge-Sort: Step 1

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



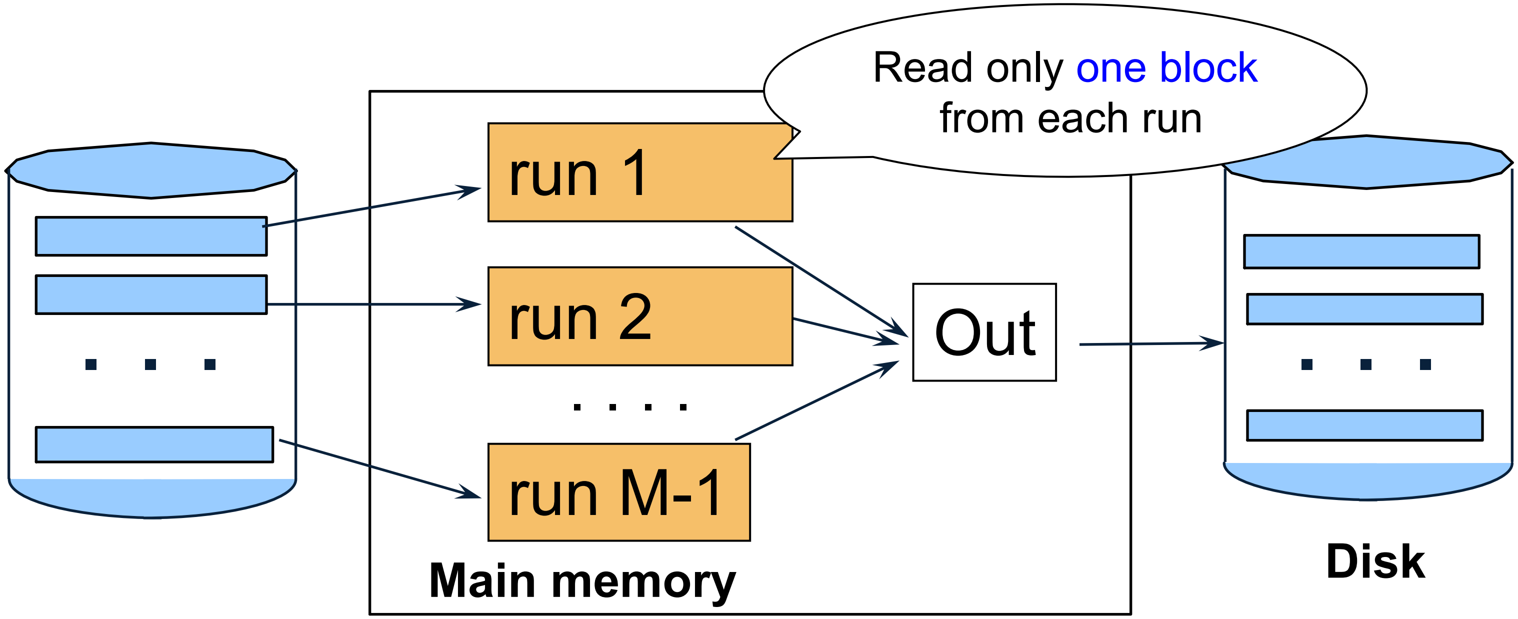
Merge-Sort: Steps 2, 3, …

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



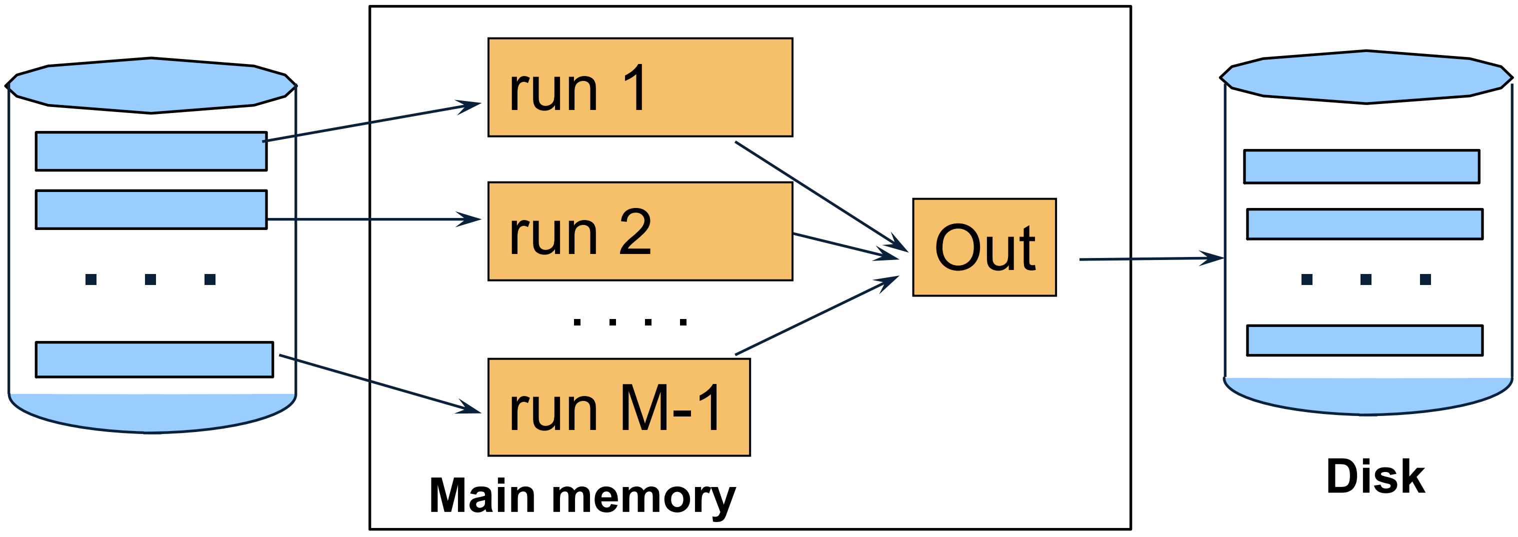
Merge-Sort: Steps 2, 3, …

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



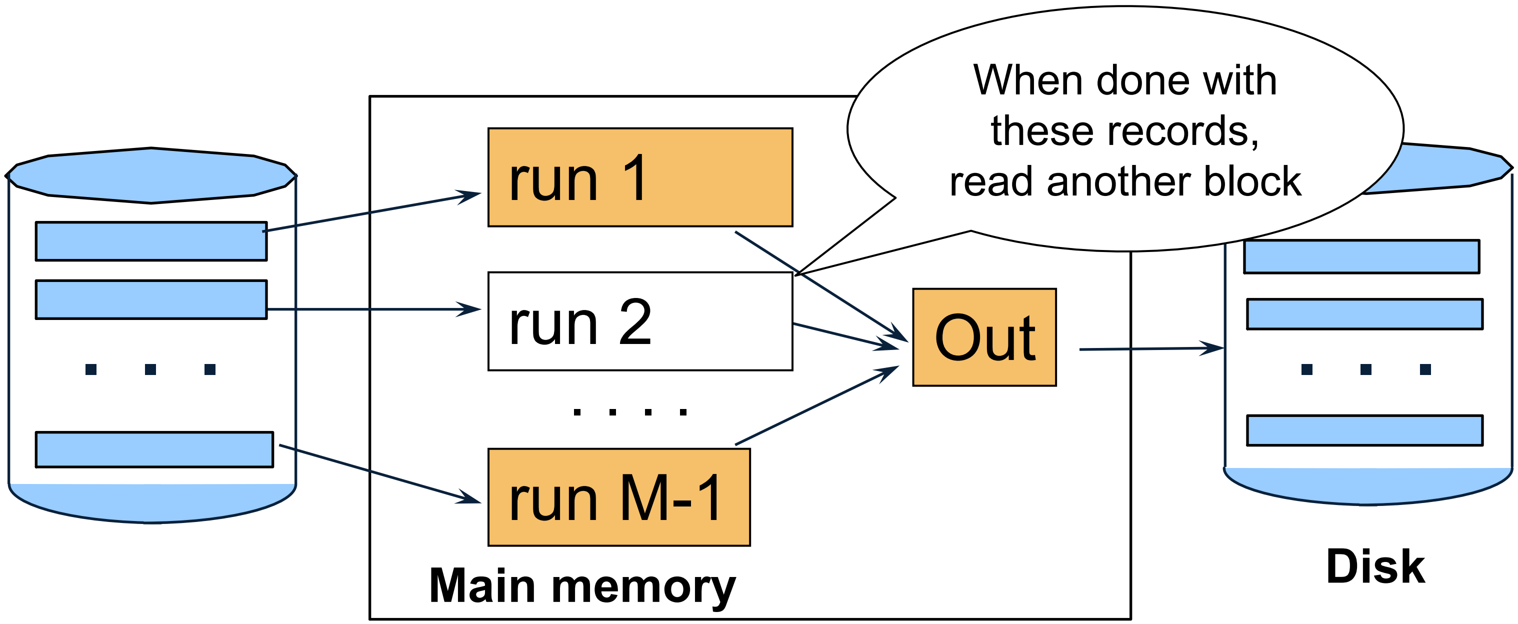
Merge-Sort: Steps 2, 3, …

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



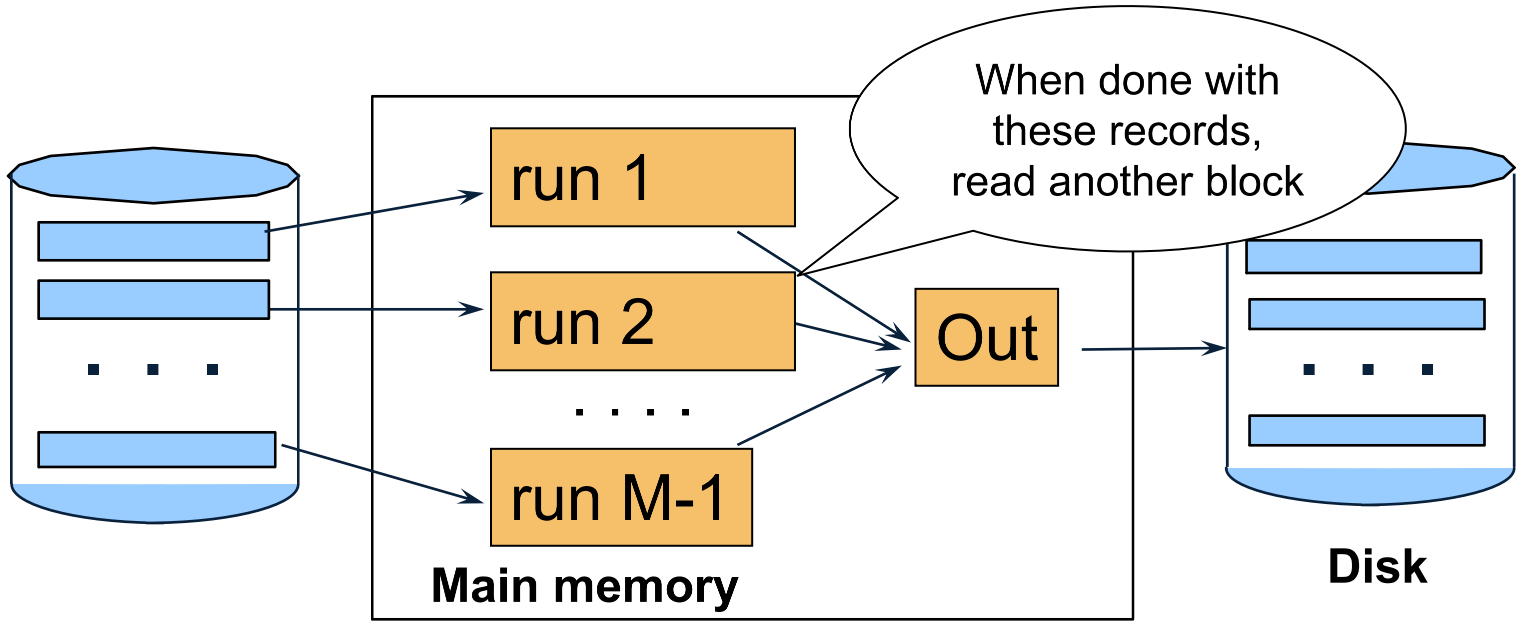
Merge-Sort: Steps 2, 3, …

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



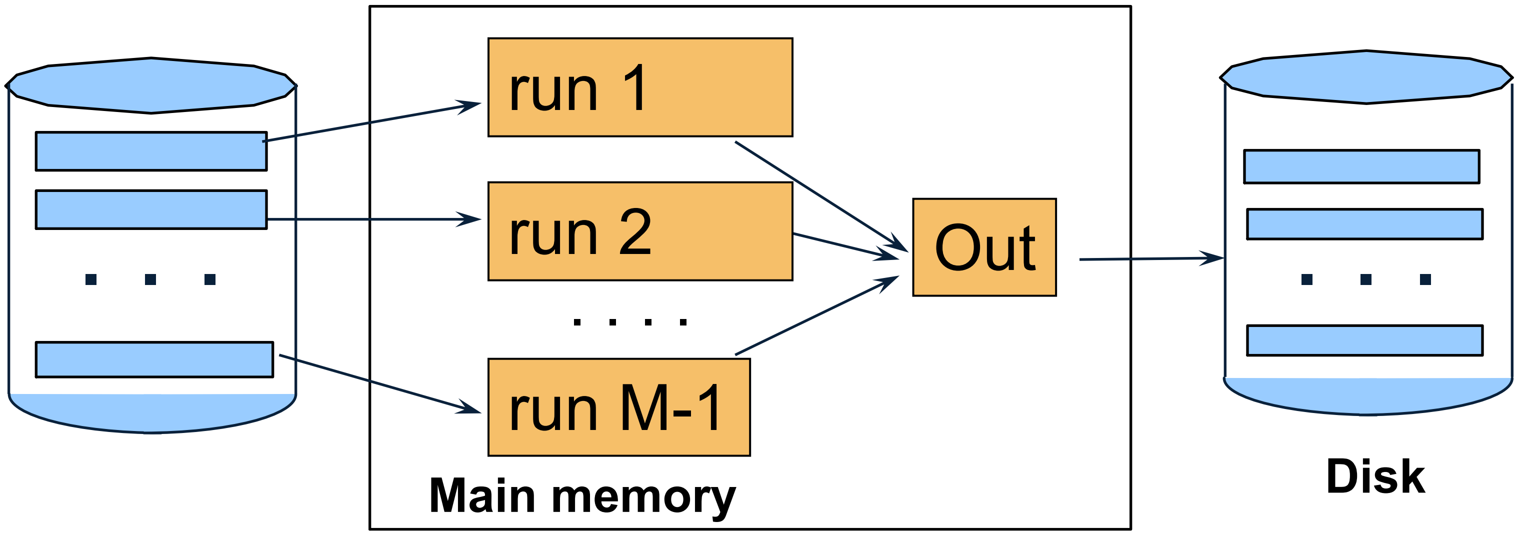
Merge-Sort: Steps 2, 3, …

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



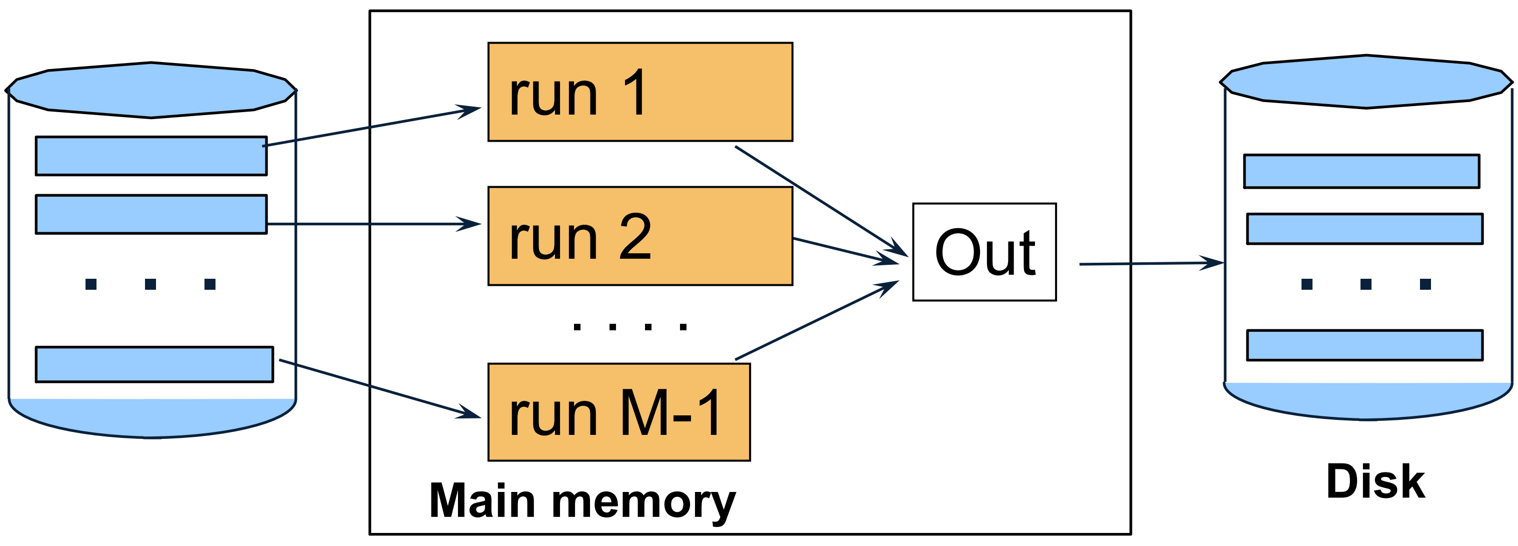
Merge-Sort: Steps 2, 3, …

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



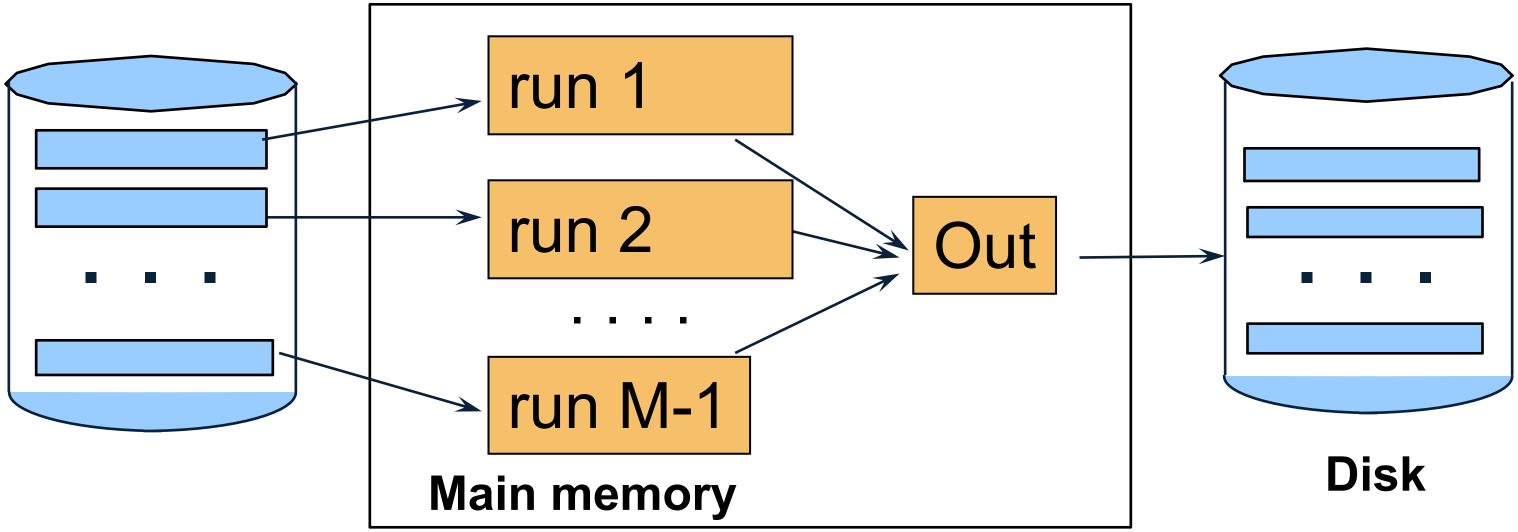
Merge-Sort: Steps 2, 3, …

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



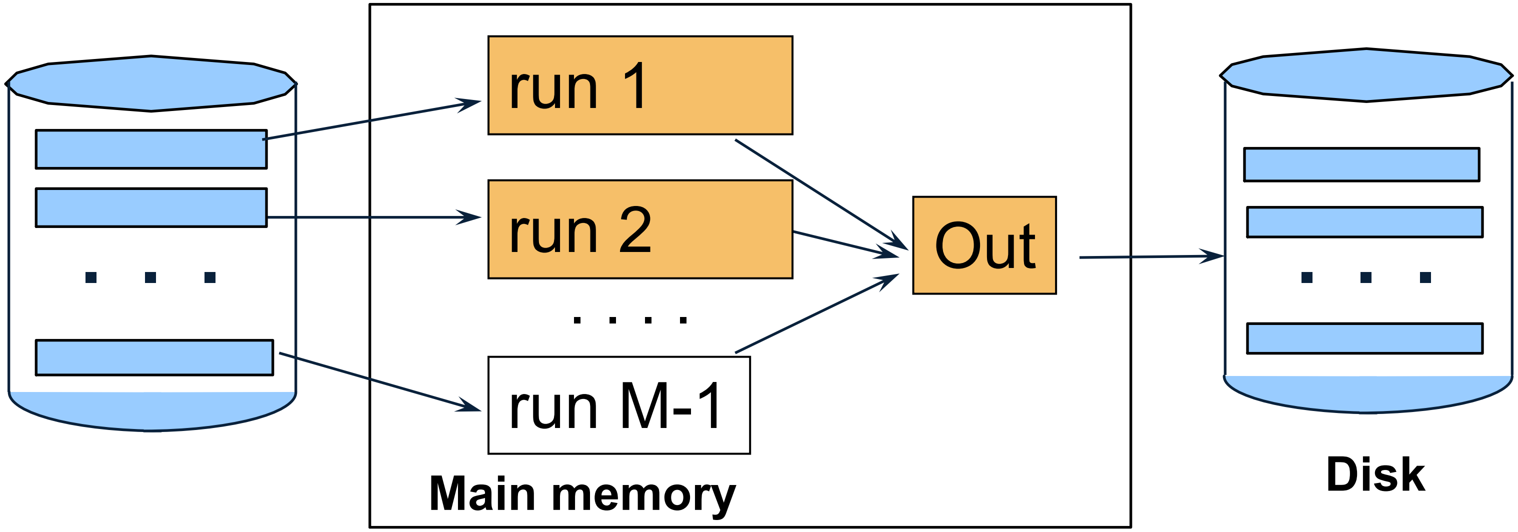
Merge-Sort: Steps 2, 3, …

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



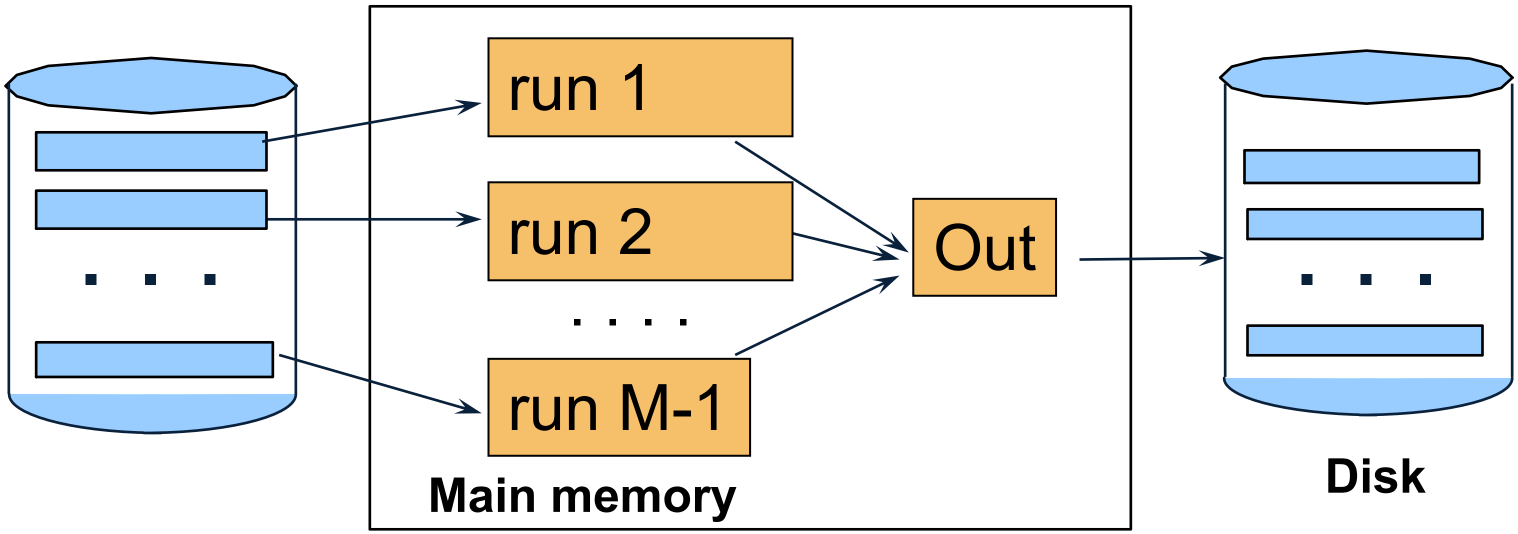
Merge-Sort: Steps 2, 3, …

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



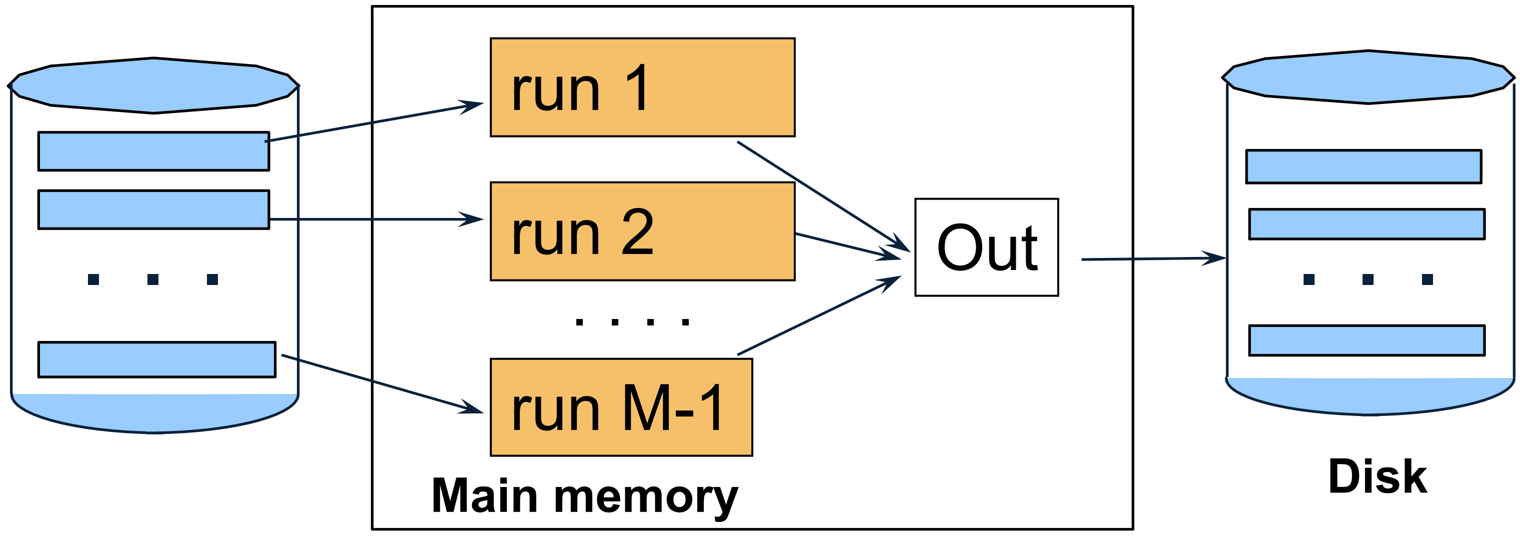
Merge-Sort: Steps 2, 3, …

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



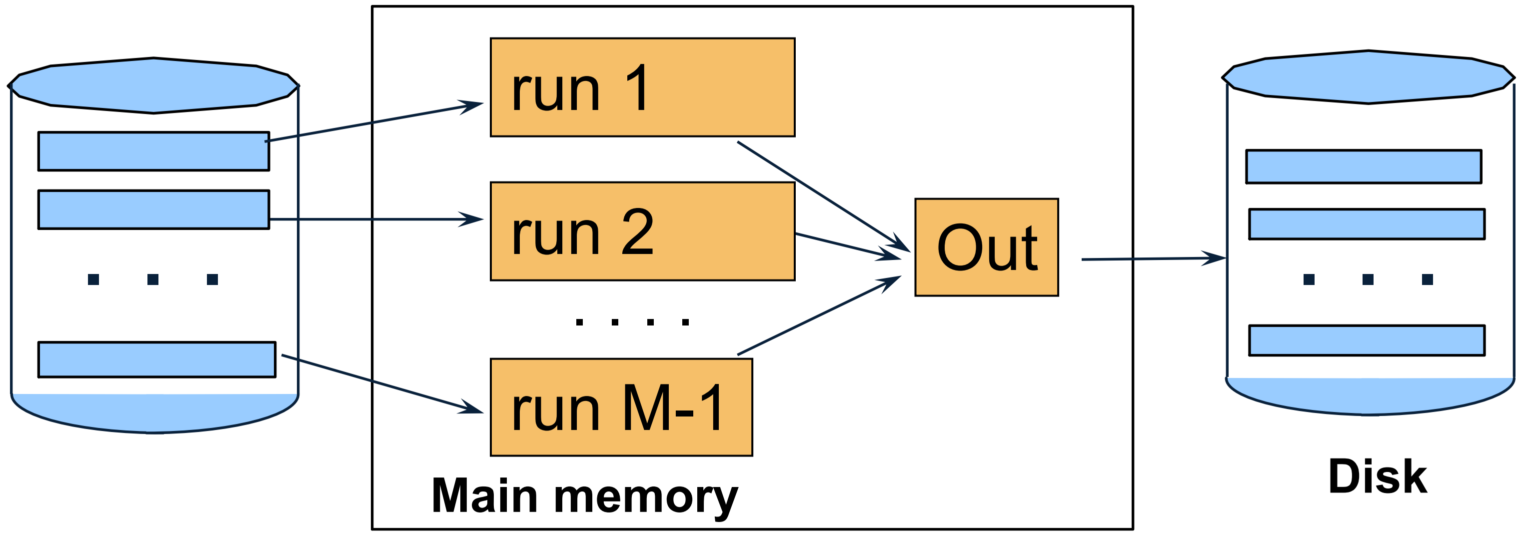
Merge-Sort: Steps 2, 3, …

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



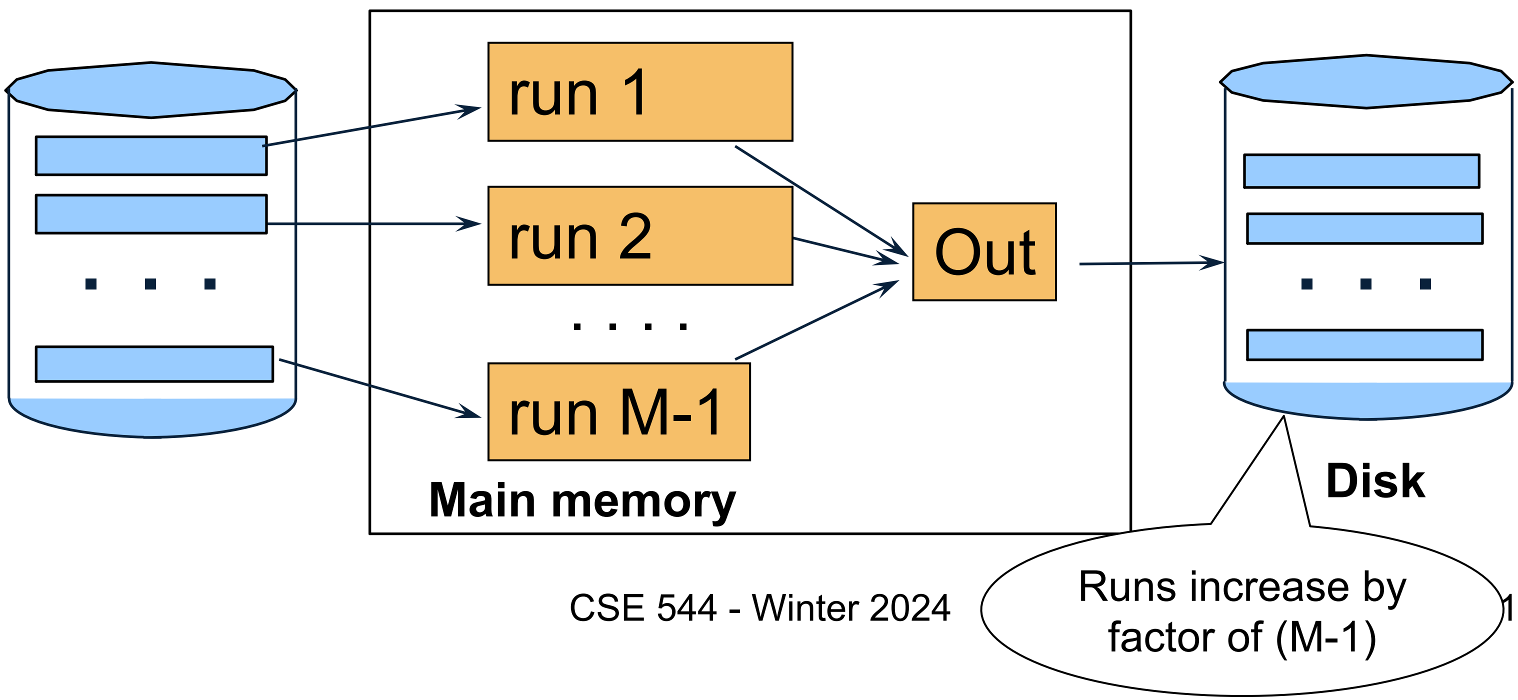
Merge-Sort: Steps 2, 3, …

- 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: Discussion

Size of the runs increases fast

- Example: B=10001

Each block is 16Kb

Merge-Sort: Discussion

Size of the runs increases fast

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

Each block is 16Kb

Merge-Sort: Discussion

Size of the runs increases fast

Each block is 16Kb

- 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

Each block is 16Kb

- 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?

$$\text{Cost} = 3(B(R)+B(S))$$

Partitioned Hash-Join

Hash-Partitioned 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

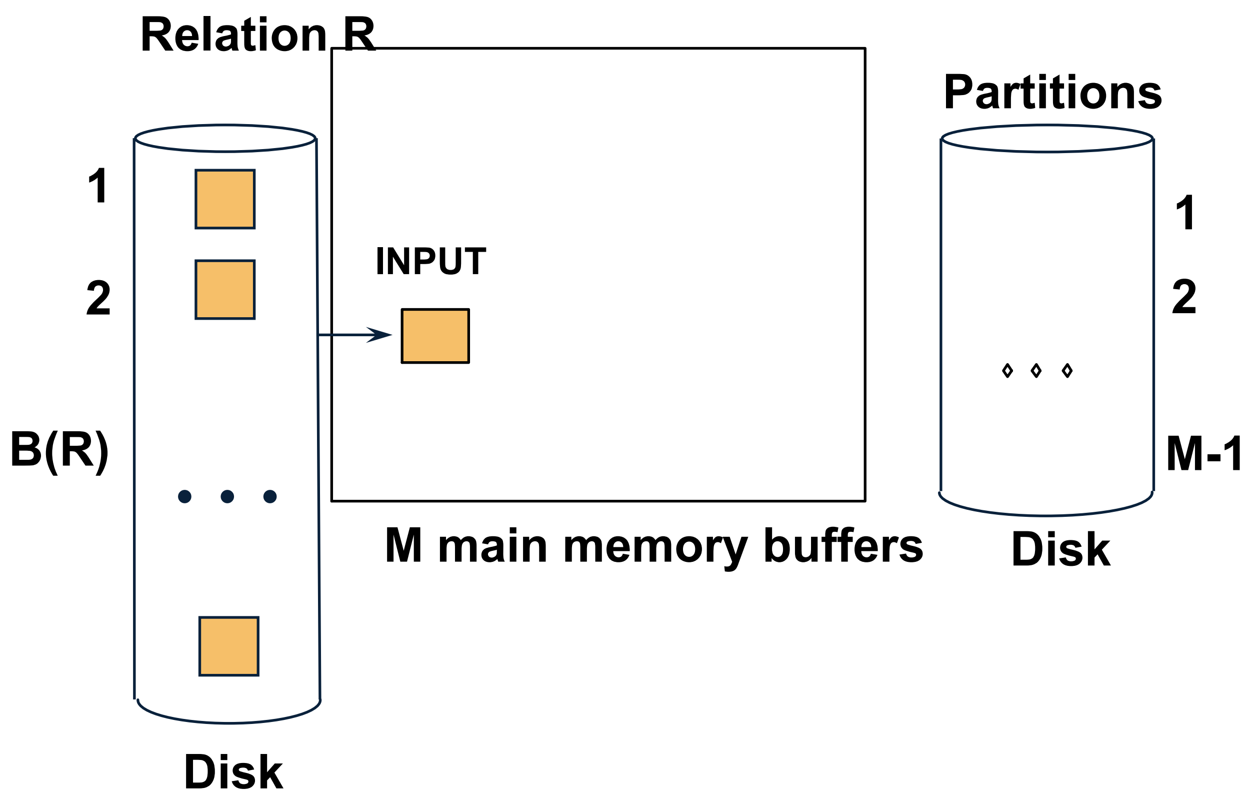
- R ⋈ S, both bigger than main memory
- Step 1:
  - Hash partition both R and S
  - Store buckets on disk

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

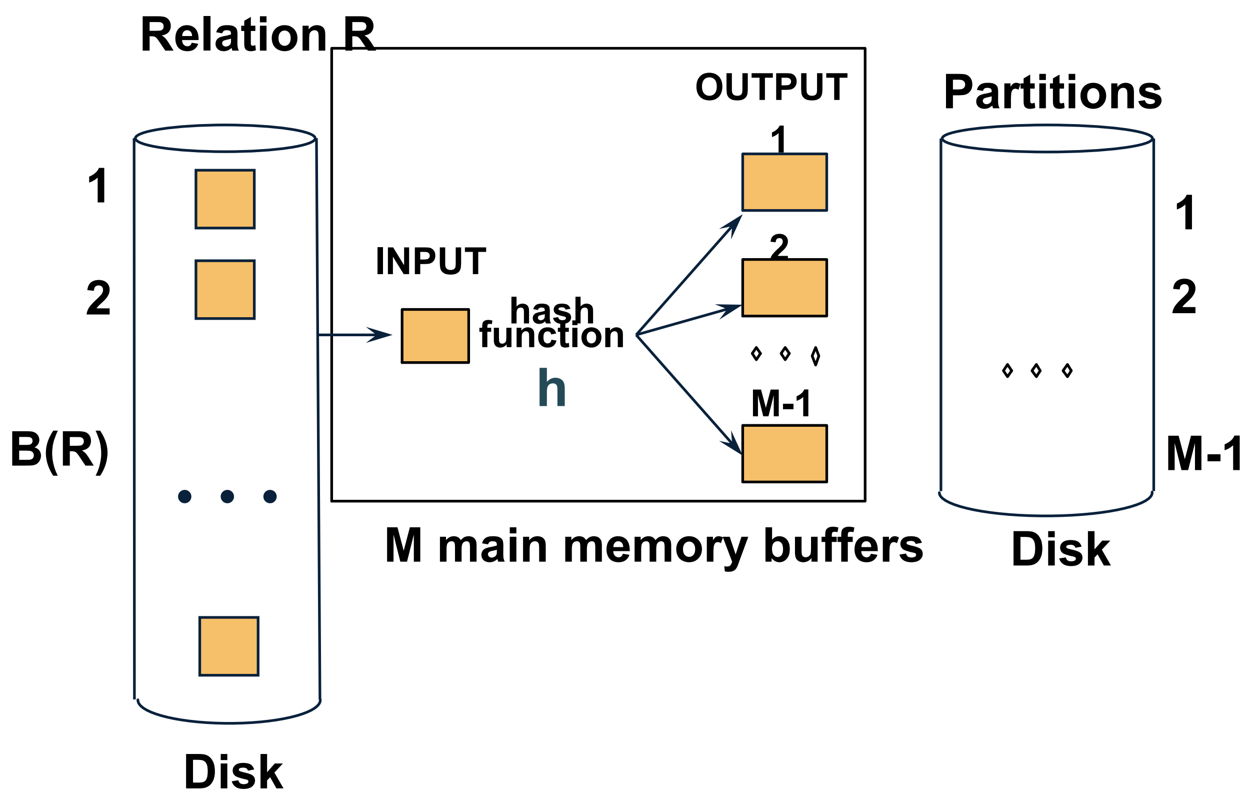
Step 1: Hash-partition

- Partition R into buckets, on disk



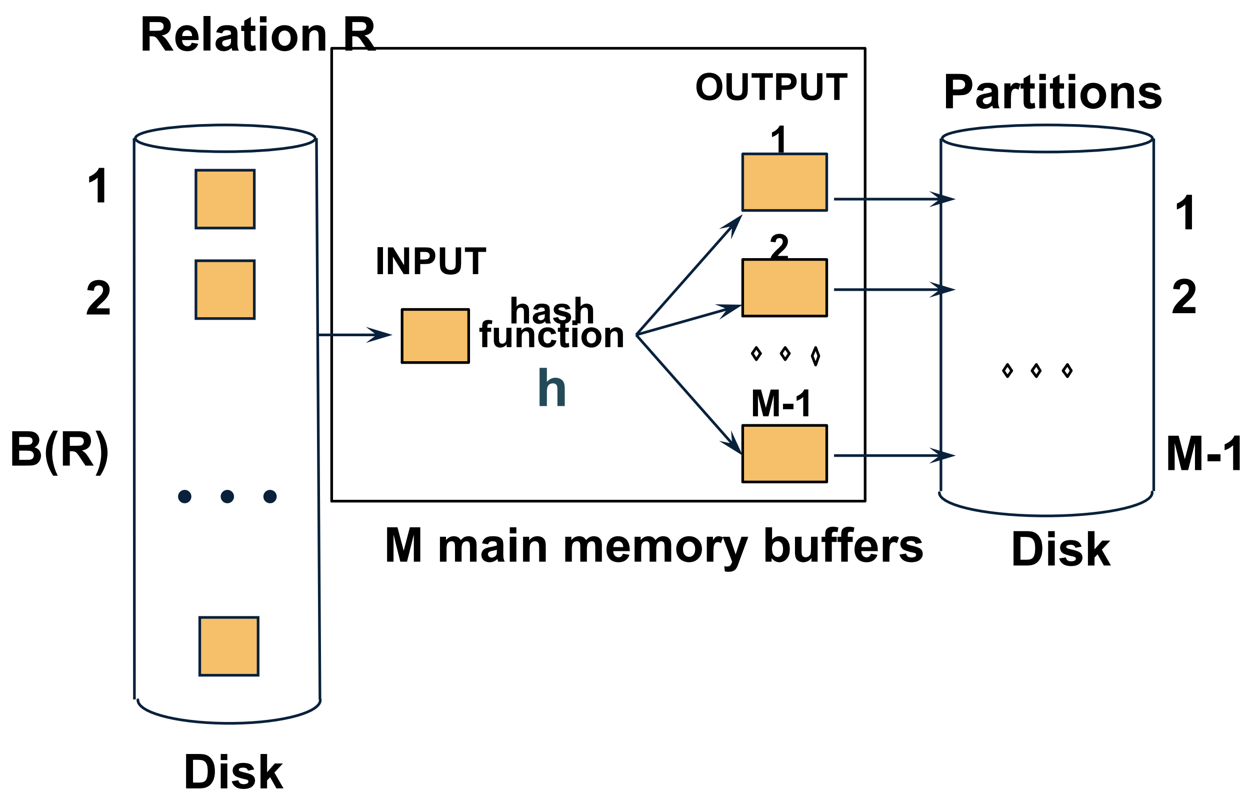
Step 1: Hash-partition

- Partition R into buckets, on disk



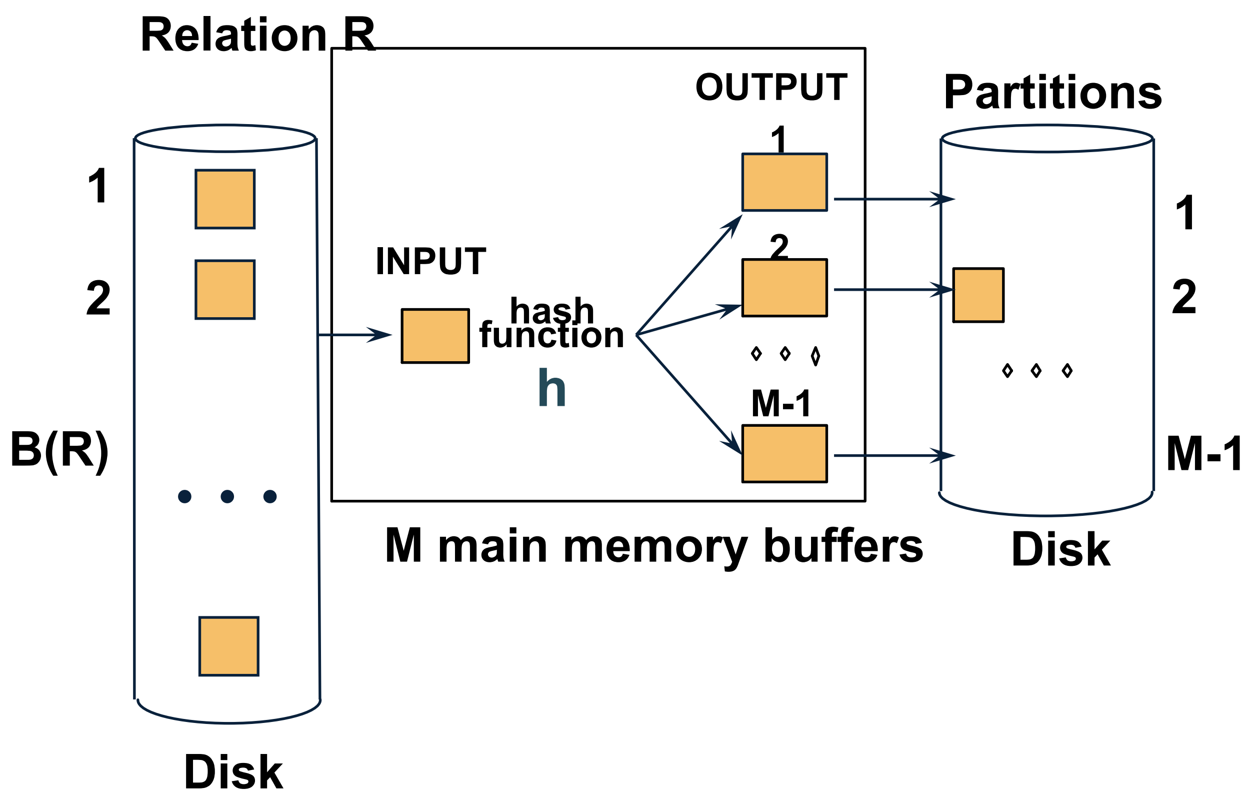
Step 1: Hash-partition

- Partition R into buckets, on disk



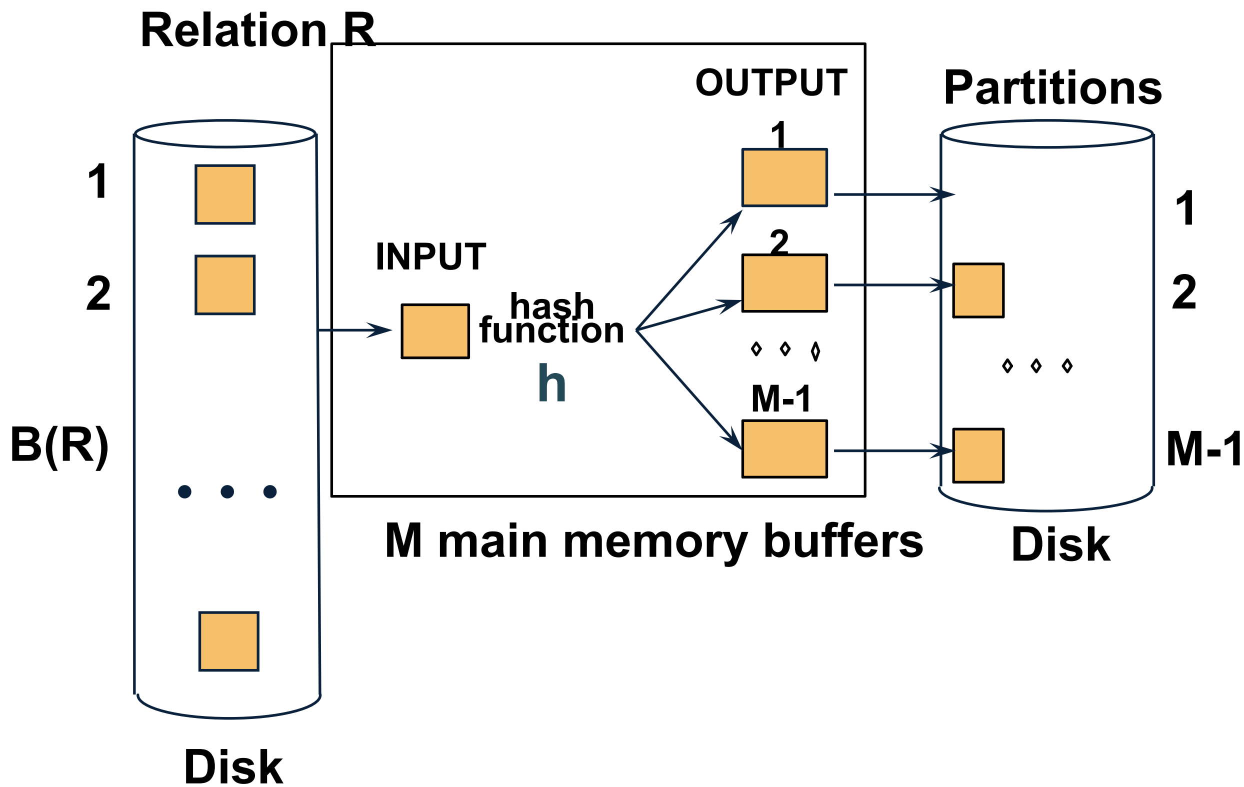
Step 1: Hash-partition

- Partition R into buckets, on disk



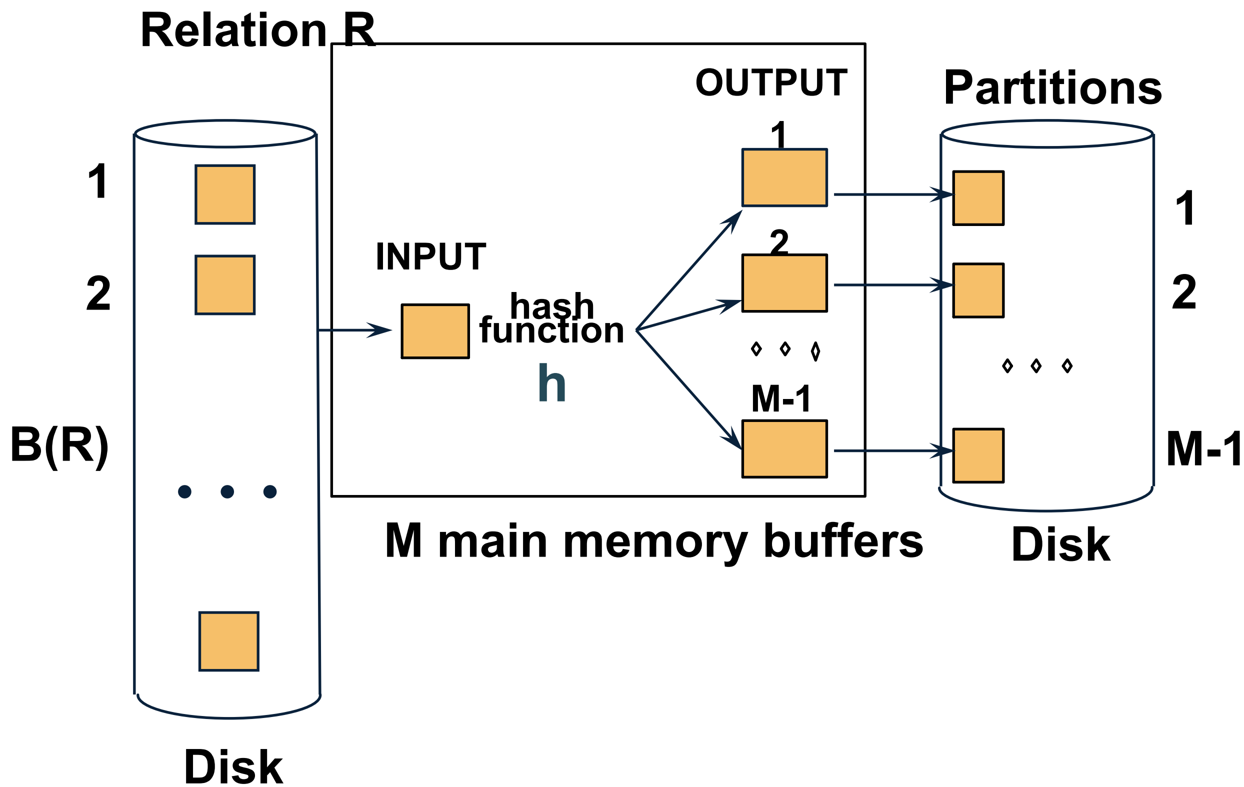
Step 1: Hash-partition

- Partition R into buckets, on disk



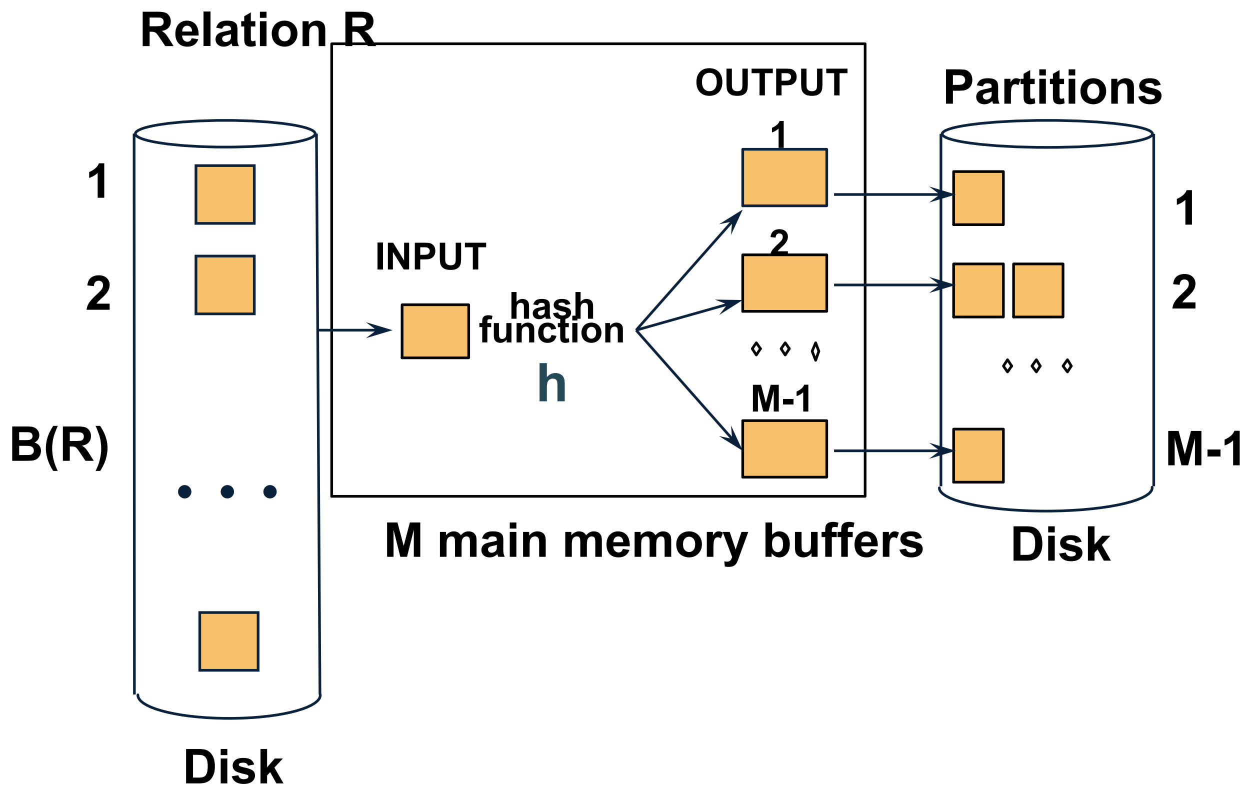
Step 1: Hash-partition

- Partition R into buckets, on disk



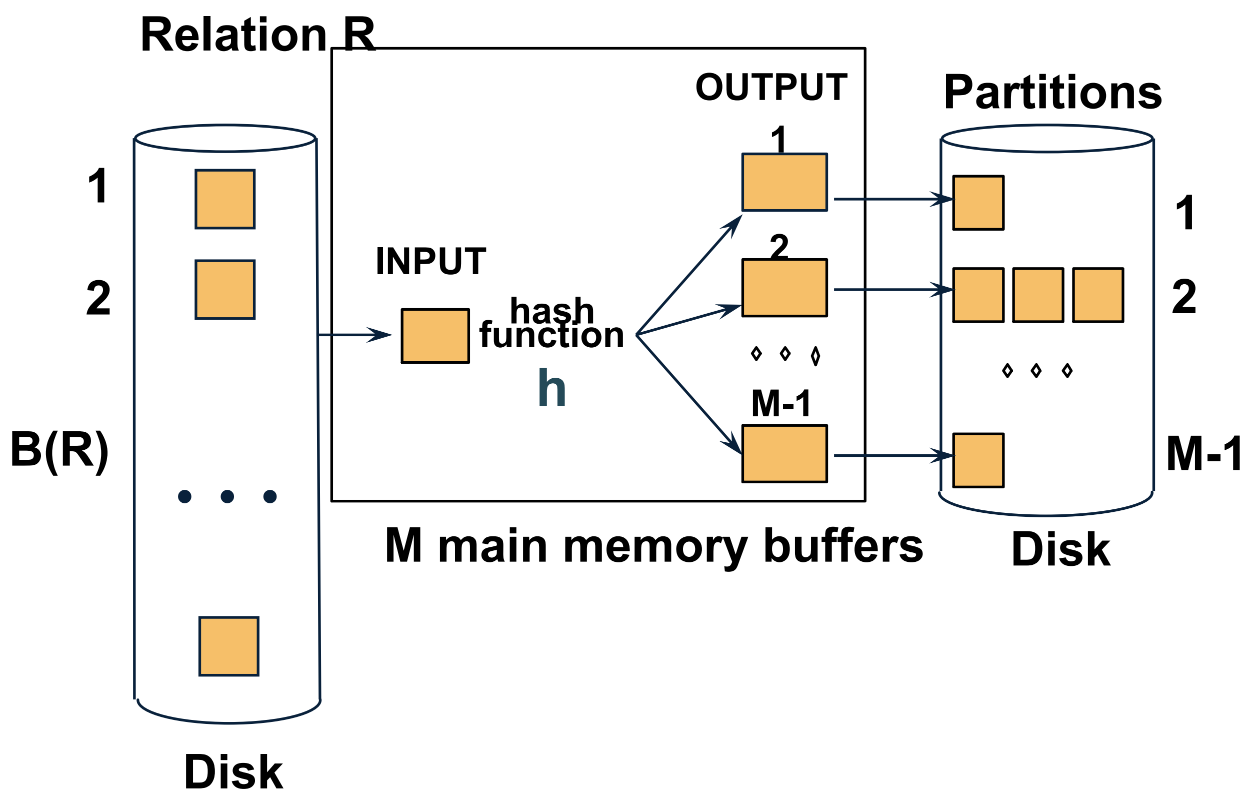
Step 1: Hash-partition

- Partition R into buckets, on disk



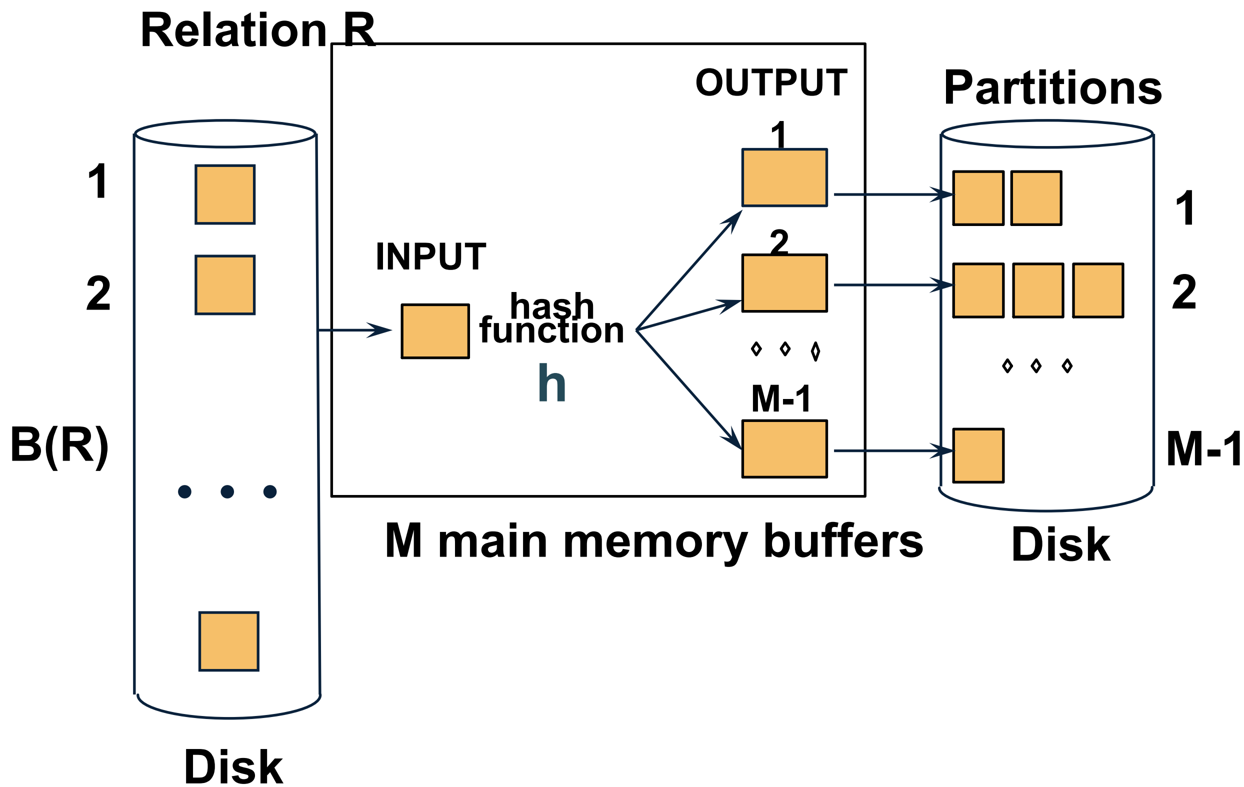
Step 1: Hash-partition

- Partition R into buckets, on disk



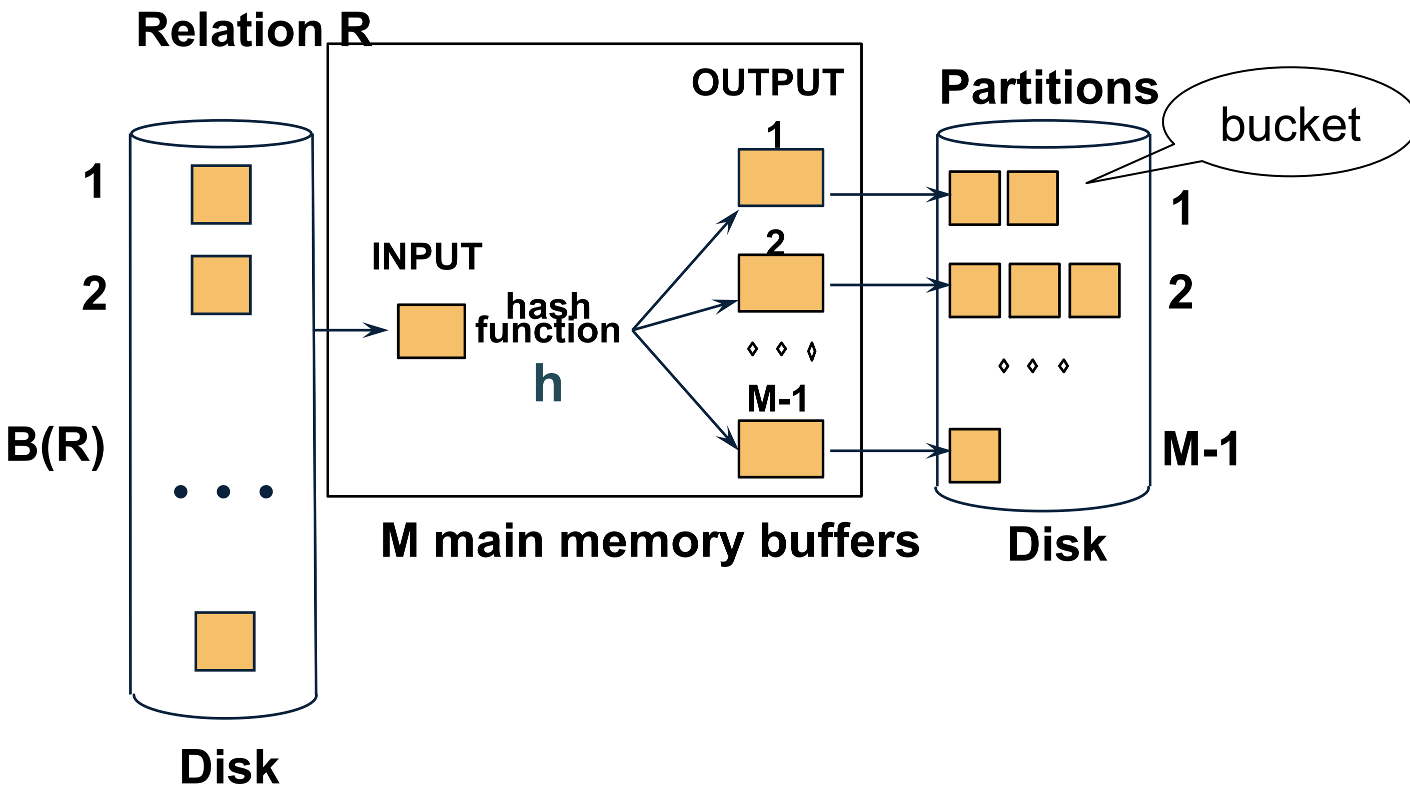
Step 1: Hash-partition

- Partition R into buckets, on disk



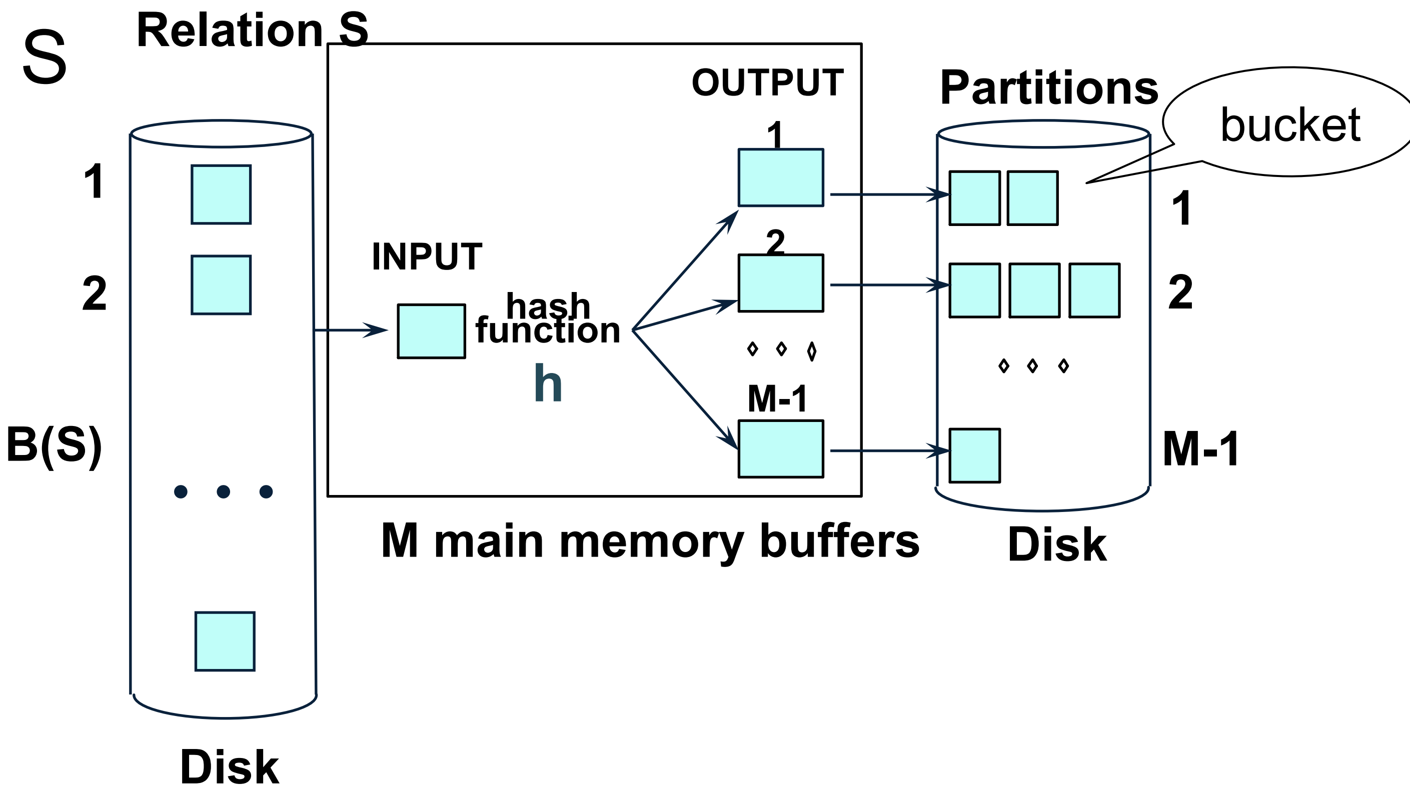
Step 1: Hash-partition

- Partition R into buckets, on disk



Step 1: Hash-partition

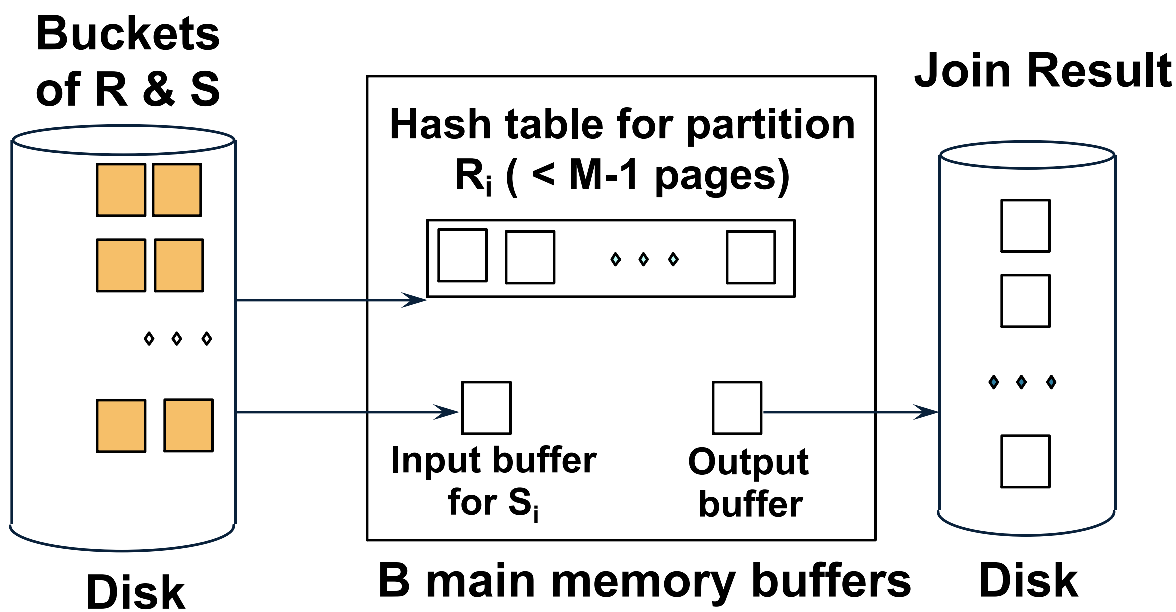
- Partition R into buckets, on disk
- Partition S



Step 2: Join Buckets

R ⋈ S

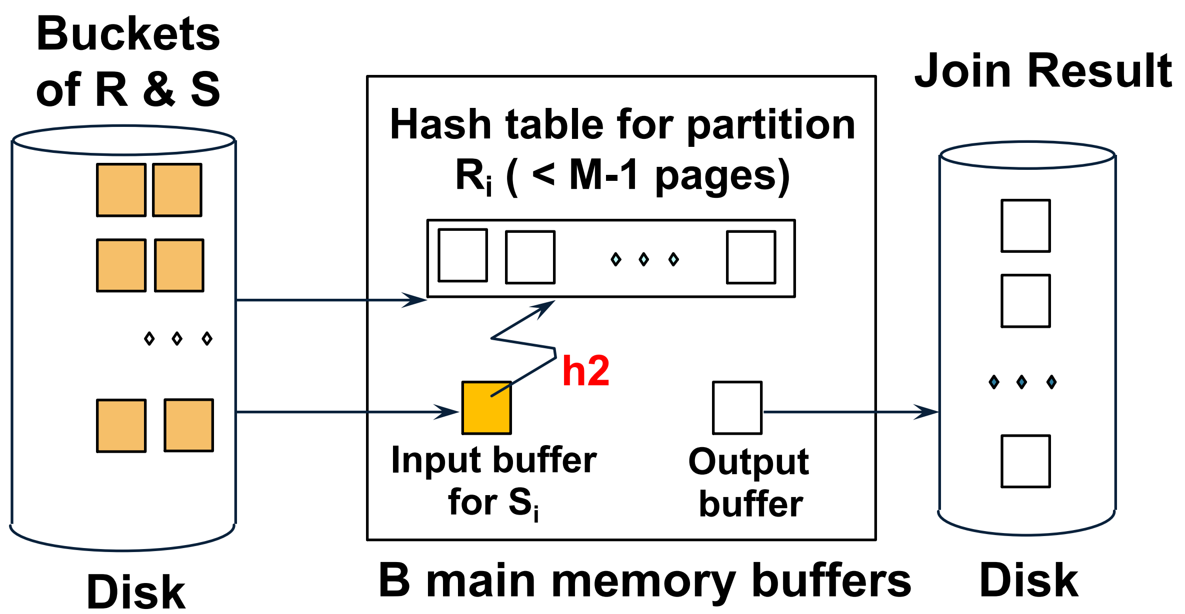
- Read one R-bucket; hash-partition it using h2 (≠ h)



Step 2: Join Buckets

R ⋈ S

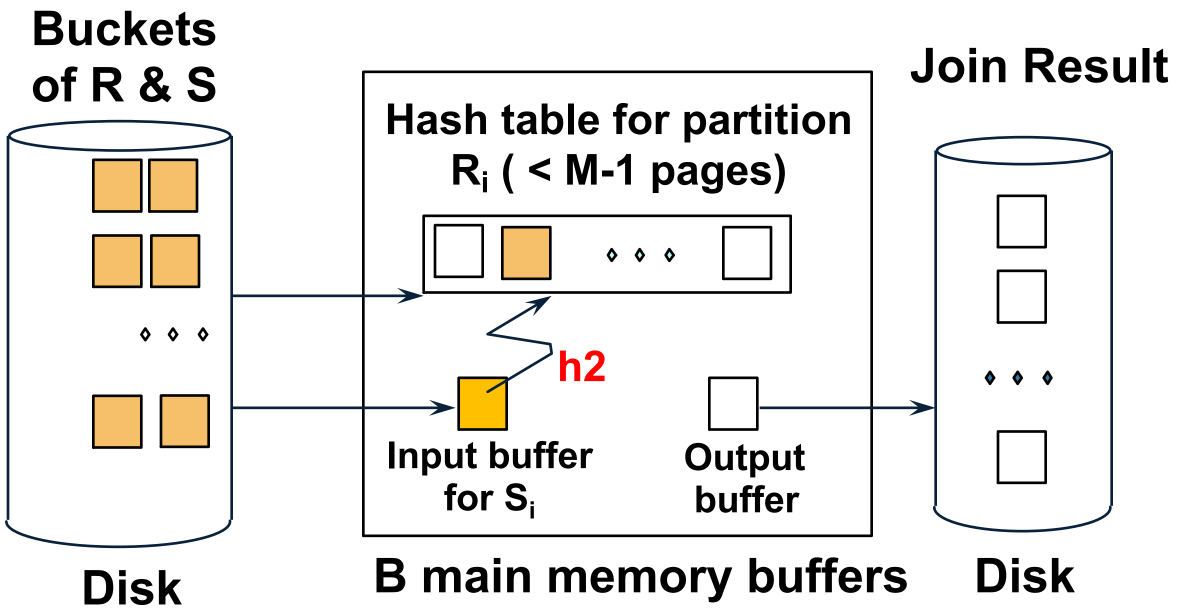
- Read one R-bucket; hash-partition it using h2 (≠ h)



Step 2: Join Buckets

R ⋈ S

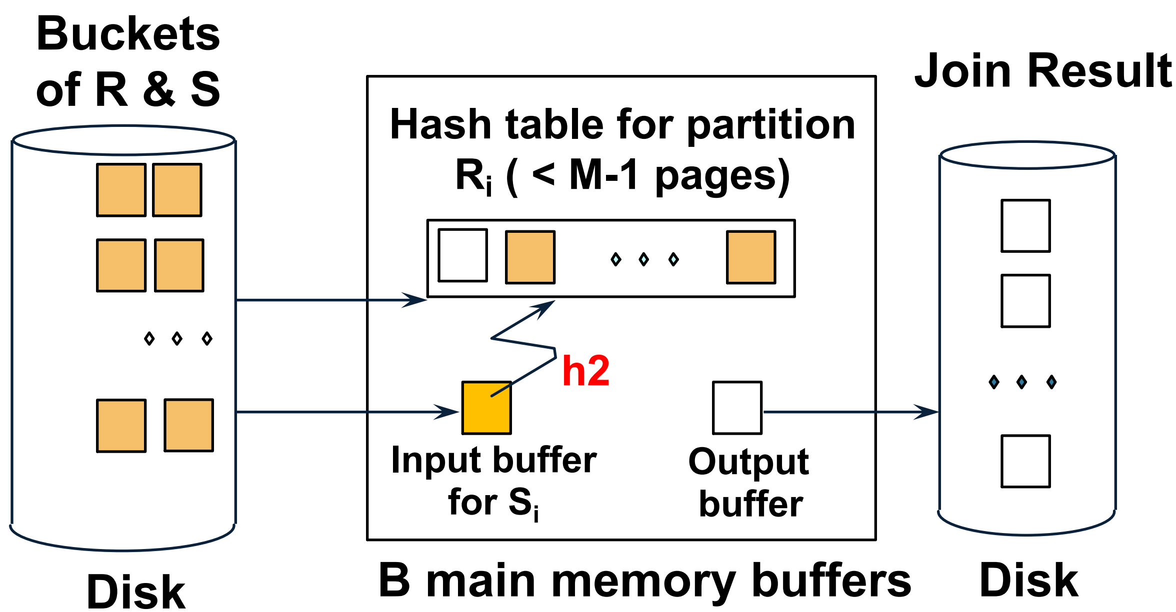
- Read one R-bucket; hash-partition it using h2 (≠ h)



Step 2: Join Buckets

R ⋈ S

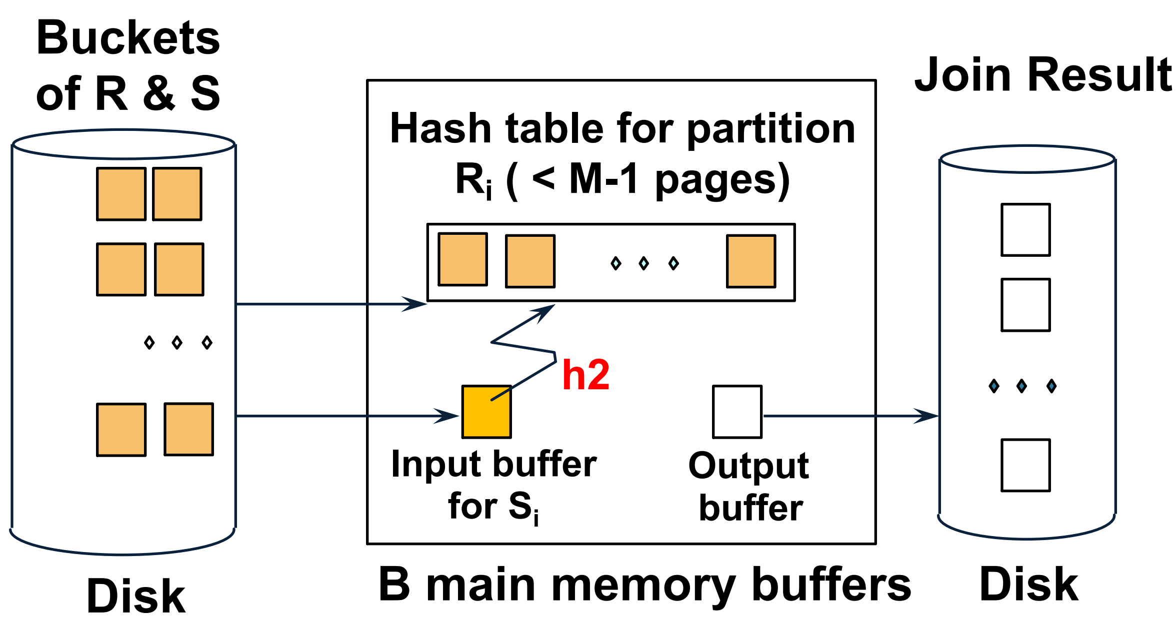
- Read one R-bucket; hash-partition it using h2 (≠ h)



Step 2: Join Buckets

$R \bowtie S$

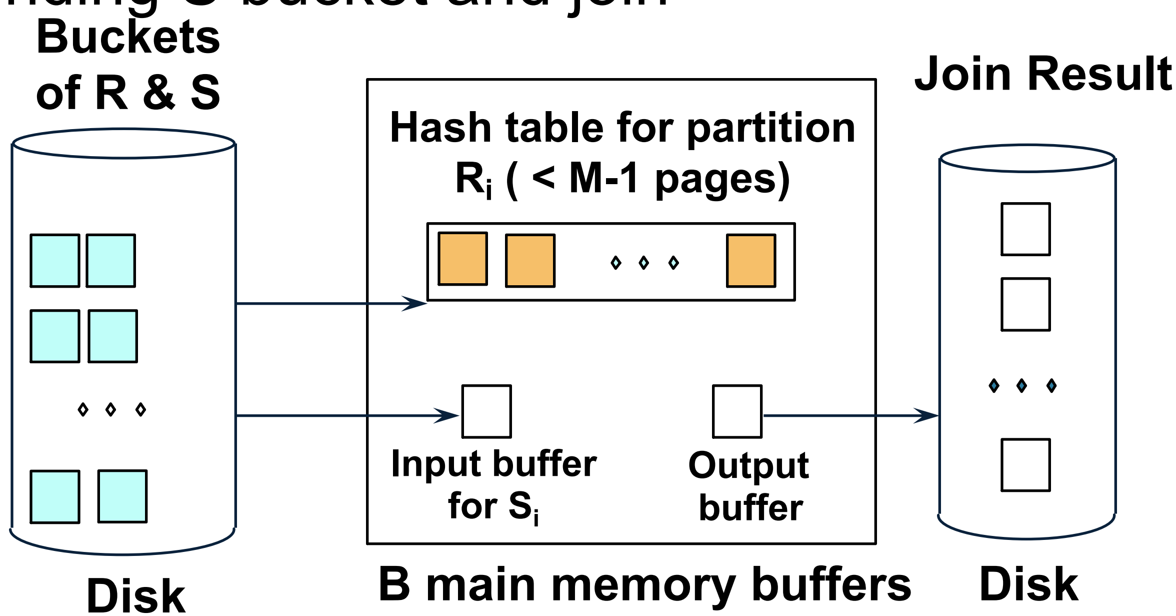
- Read one R-bucket; hash-partition it using h2 ($\neq$ h)



Step 2: Join Buckets

R ⋈ S

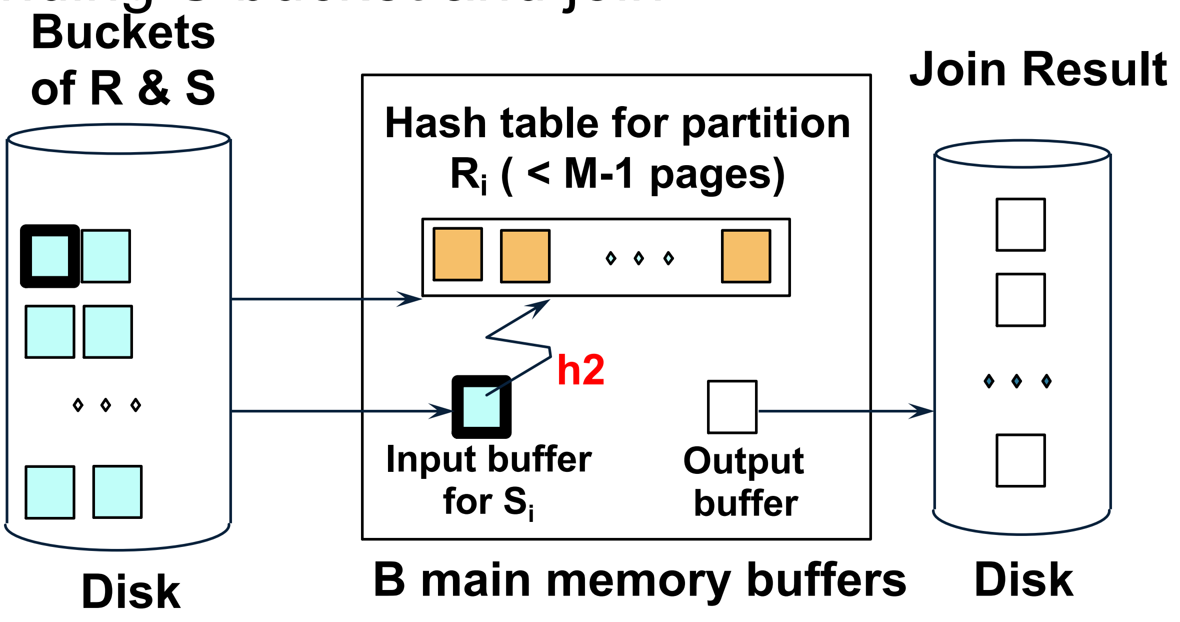
- Read one R-bucket; hash-partition it using h2 (≠ h)
- Scan corresponding S bucket and join



Step 2: Join Buckets

R ⋈ S

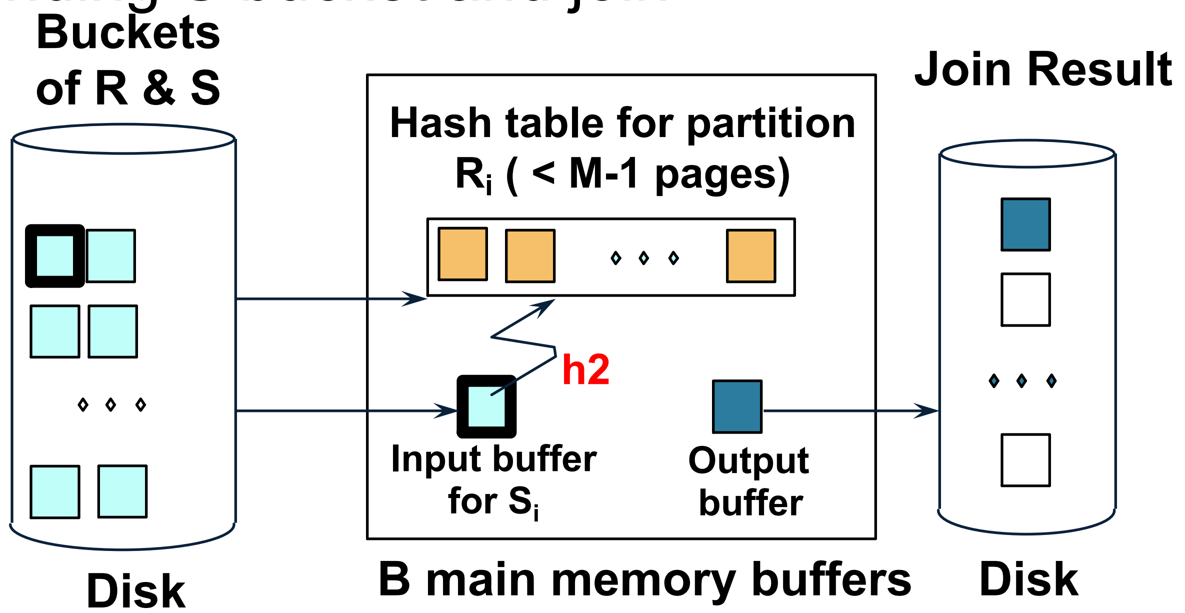
- Read one R-bucket; hash-partition it using h2 ($\neq$ h)
- Scan corresponding S bucket and join



Step 2: Join Buckets

R ⋈ S

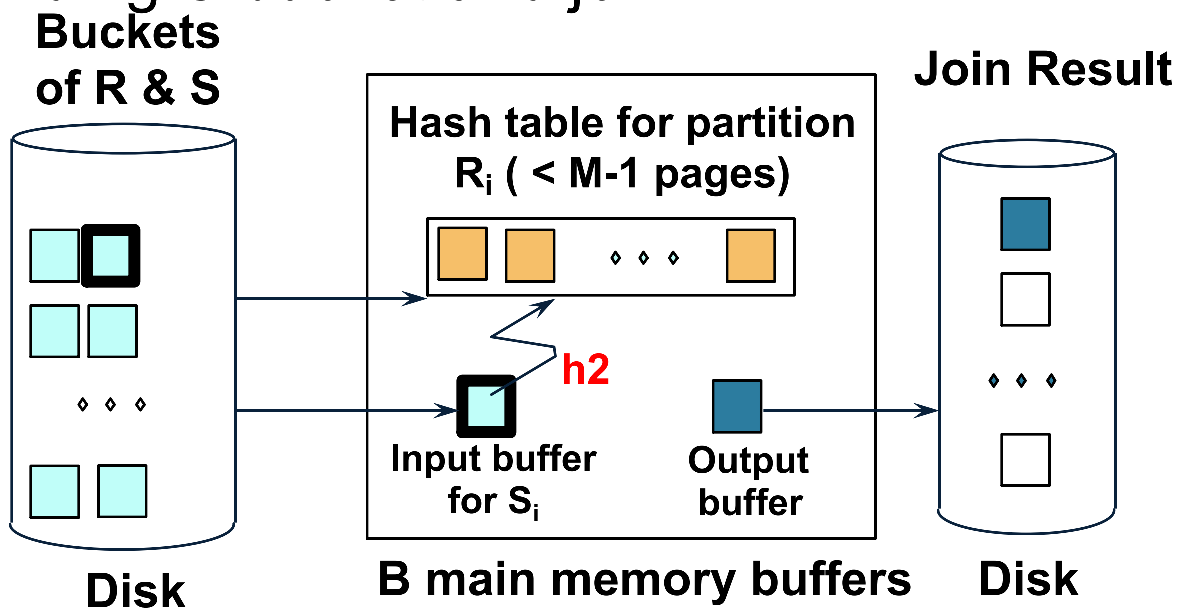
- Read one R-bucket; hash-partition it using h2 (≠ h)
- Scan corresponding S bucket and join



Step 2: Join Buckets

R ⋈ S

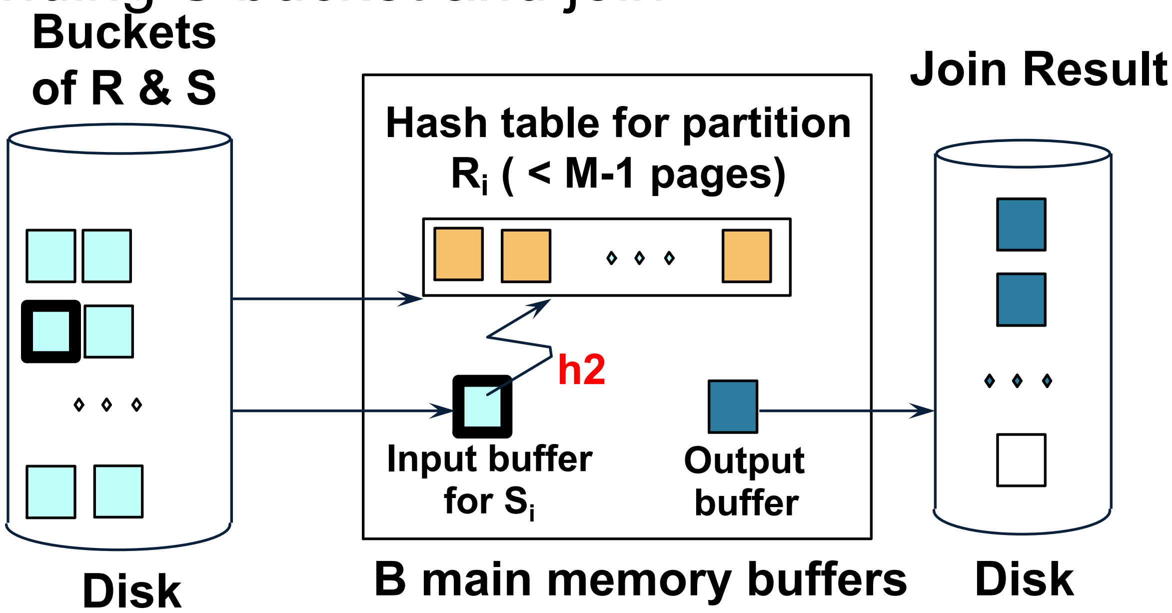
- Read one R-bucket; hash-partition it using h2 (≠ h)
- Scan corresponding S bucket and join



Step 2: Join Buckets

$R \bowtie S$

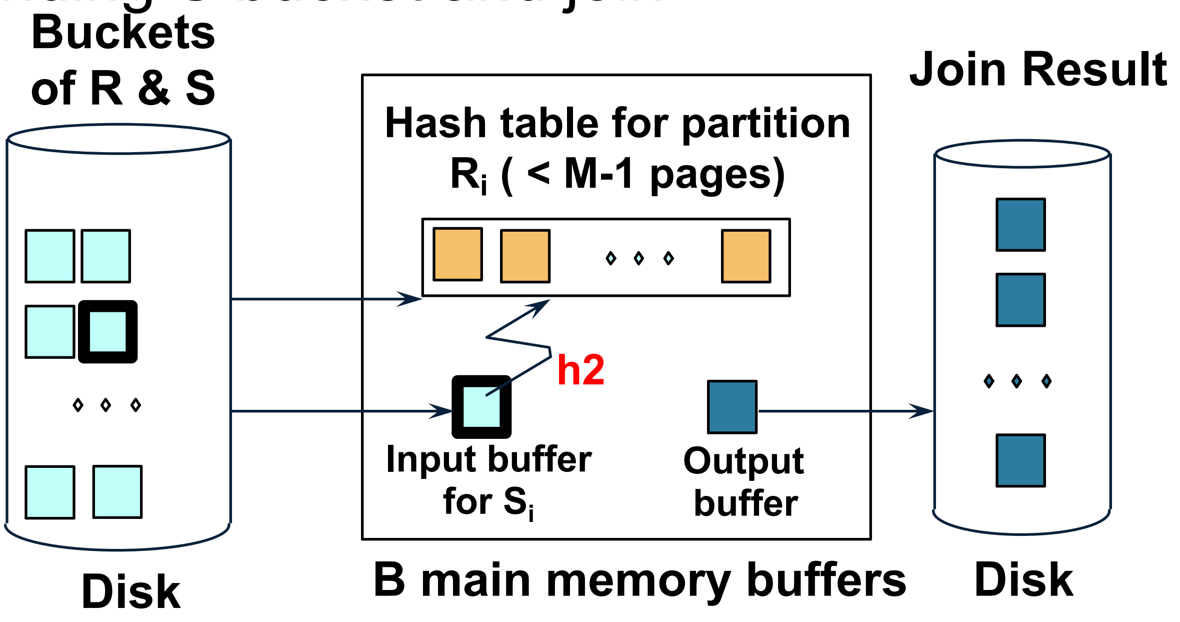
- Read one R-bucket; hash-partition it using h2 ($\neq$ h)
- Scan corresponding S bucket and join



Step 2: Join Buckets

$R \bowtie S$

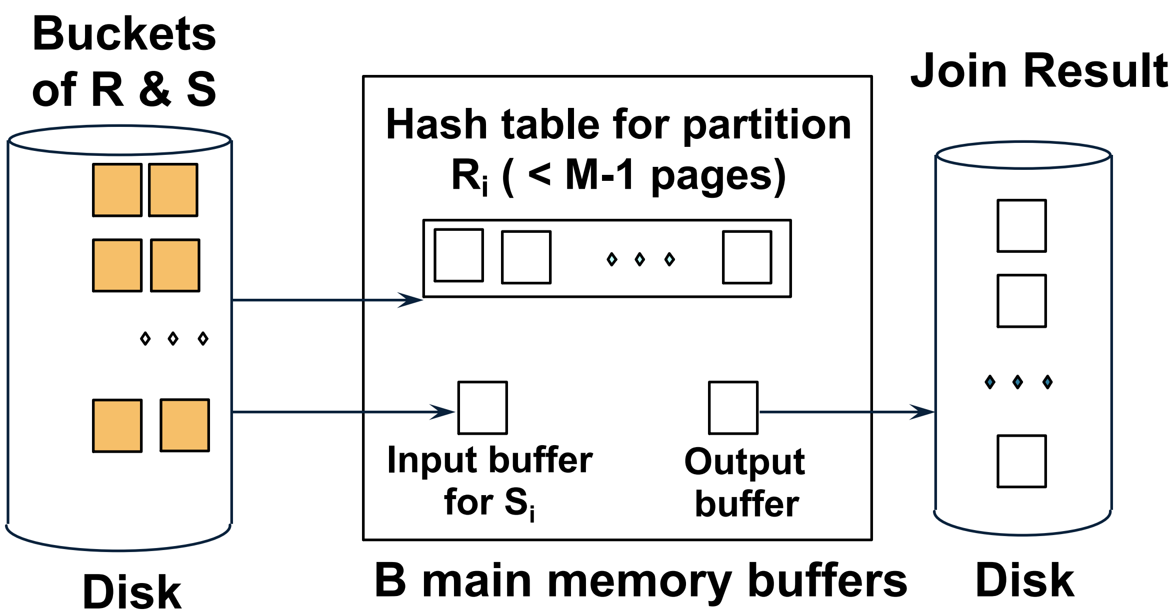
- Read one R-bucket; hash-partition it using h2 ($\neq$ h)
- Scan corresponding S bucket and join



Step 2: Join Buckets

$R \bowtie S$

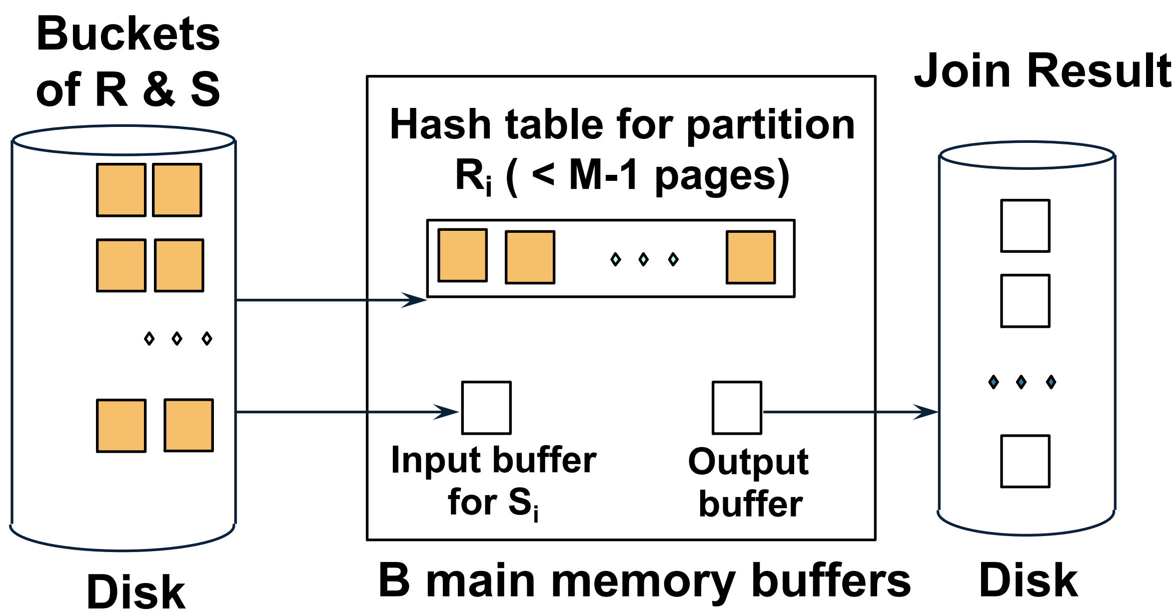
- Read the next R bucket



Step 2: Join Buckets

$R \bowtie S$

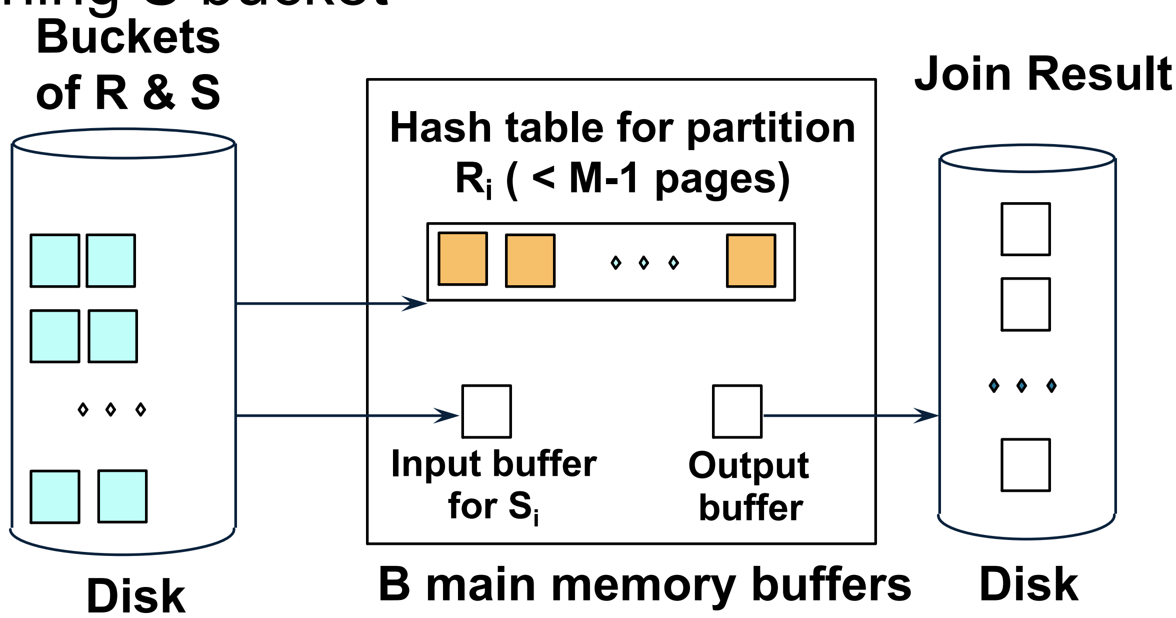
- Read the next R bucket



Step 2: Join Buckets

R ⋈ 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)) ≤ $(M-1)^2$

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