CSE544
Data Management

Lectures 9-10
Query Execution

Announcements

- HW2 due on Friday
- Project proposals due next Friday

Logical and Physical Plans

Review: Relational Algebra

Five operators:

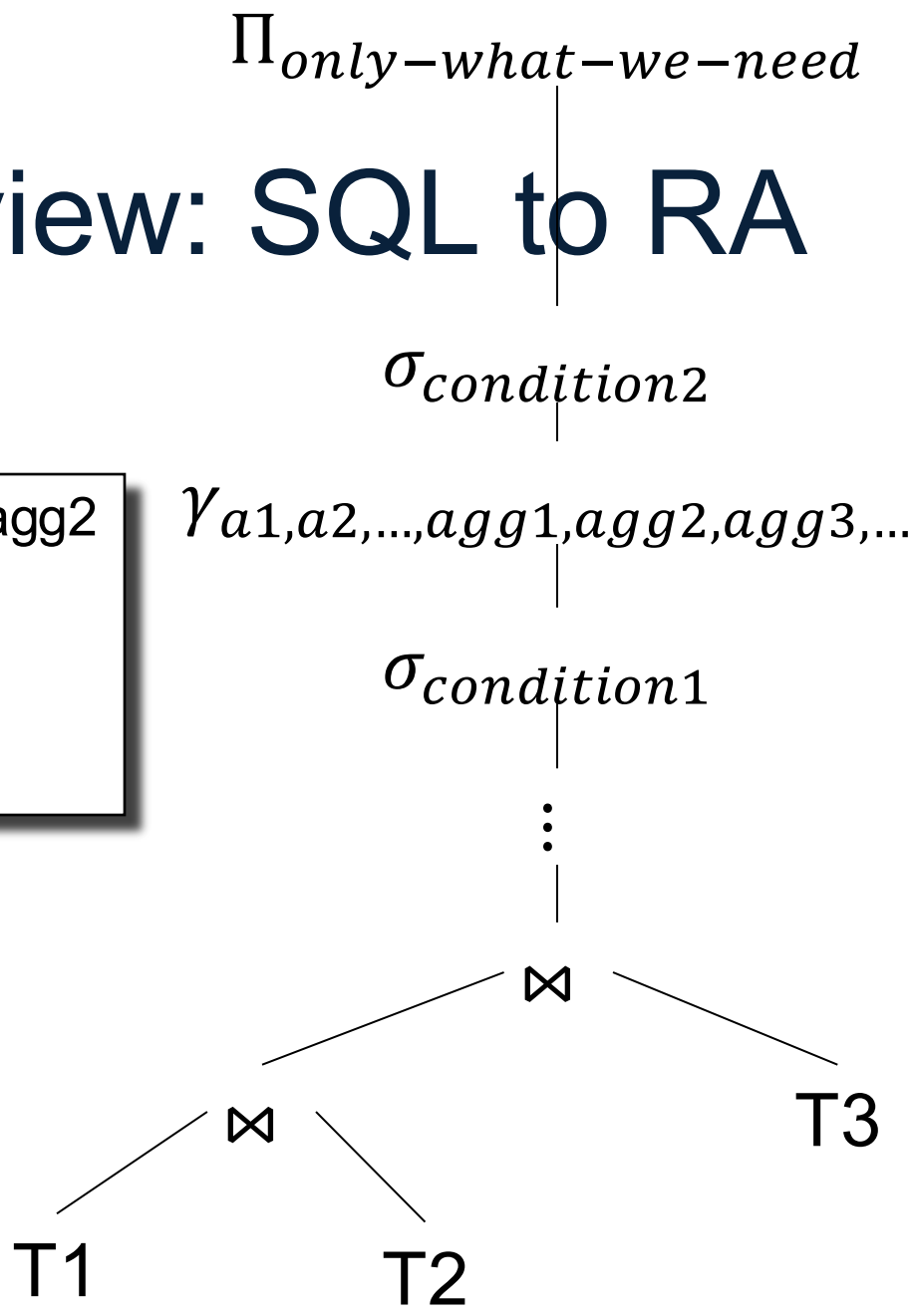
- Selection σ
- Projection Π
- Join or cartesian product $\bowtie$, $\times$
- Union $\cup$
- Difference $-$

Review: Extended RA

- Group-by and aggregate: γ
- Duplicate elimination: δ
- Sorting: τ

Review: SQL to RA

```
SELECT a1,a2,…,agg1,agg2
FROM T1, T2, …
WHERE condition1
GROUP BY a1,a2,…
HAVING condition2
```



SQL to Logical Plan

- SQL query is converted to a logical plan
- The initial plan can be naïve; we will discuss later how to optimize it

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

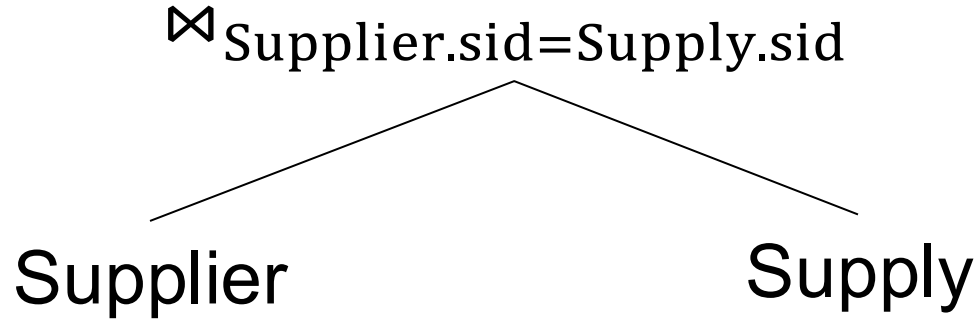
Logical Query Plan

```
SELECT DISTINCT x.sname
FROM Supplier x, Supply y
WHERE x.sno=y.sno
      and x.scity='Seattle'
      and x.sstate='WA'
      and y.pno=2
```

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

Logical Query Plan

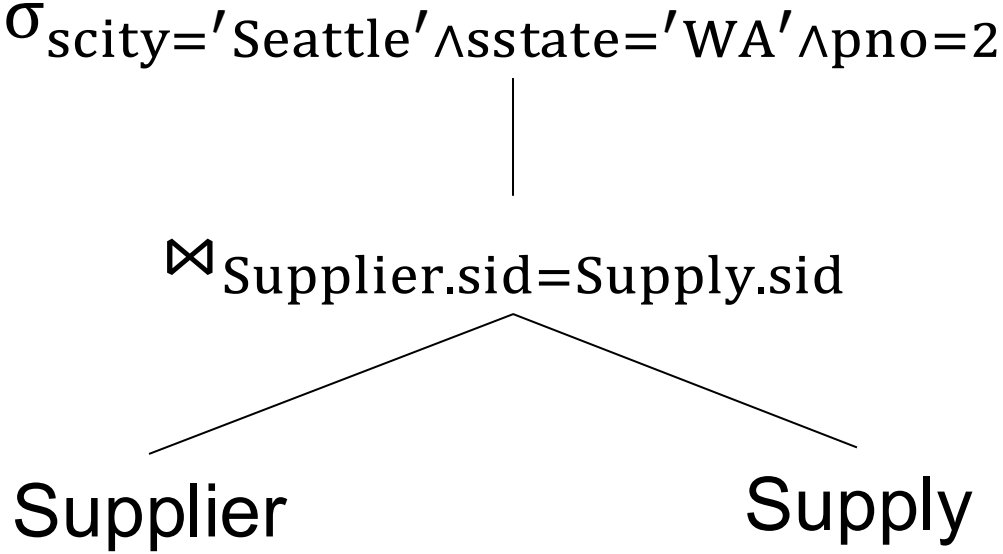
```
SELECT DISTINCT x.sname
FROM Supplier x, Supply y
WHERE x.sno=y.sno
      and x.scity='Seattle'
      and x.sstate='WA'
      and y.pno=2
```



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

Logical Query Plan

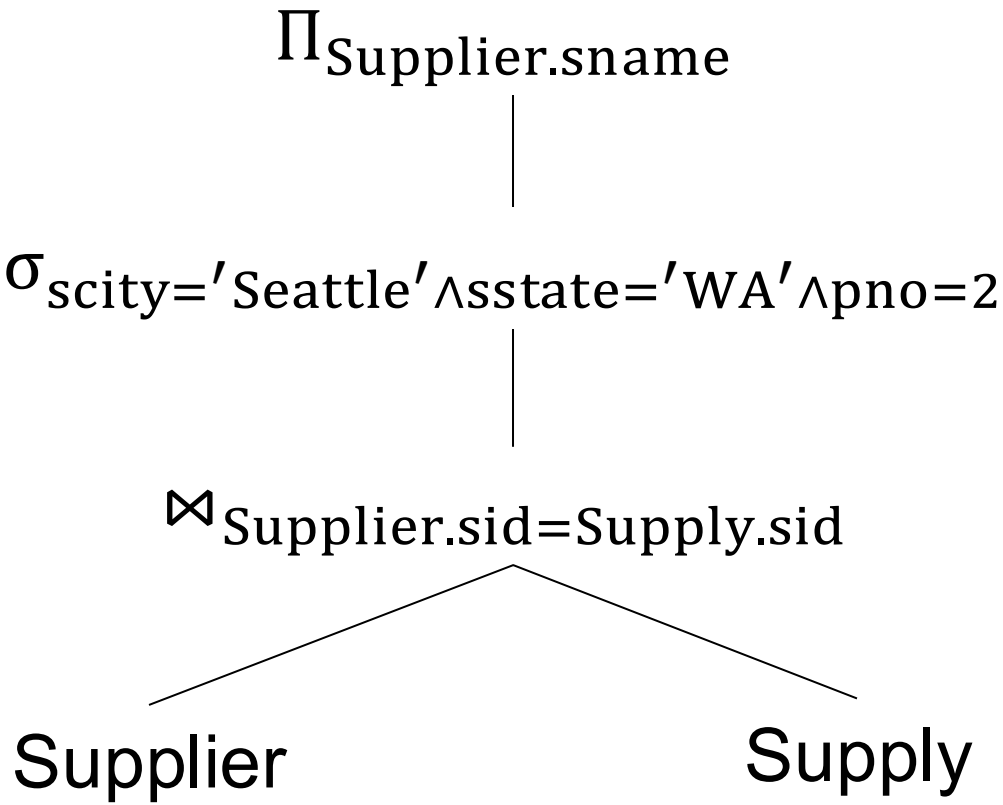
```
SELECT DISTINCT x.sname
FROM Supplier x, Supply y
WHERE x.sno=y.sno
      and x.scity='Seattle'
      and x.sstate='WA'
      and y.pno=2
```



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

Logical Query Plan

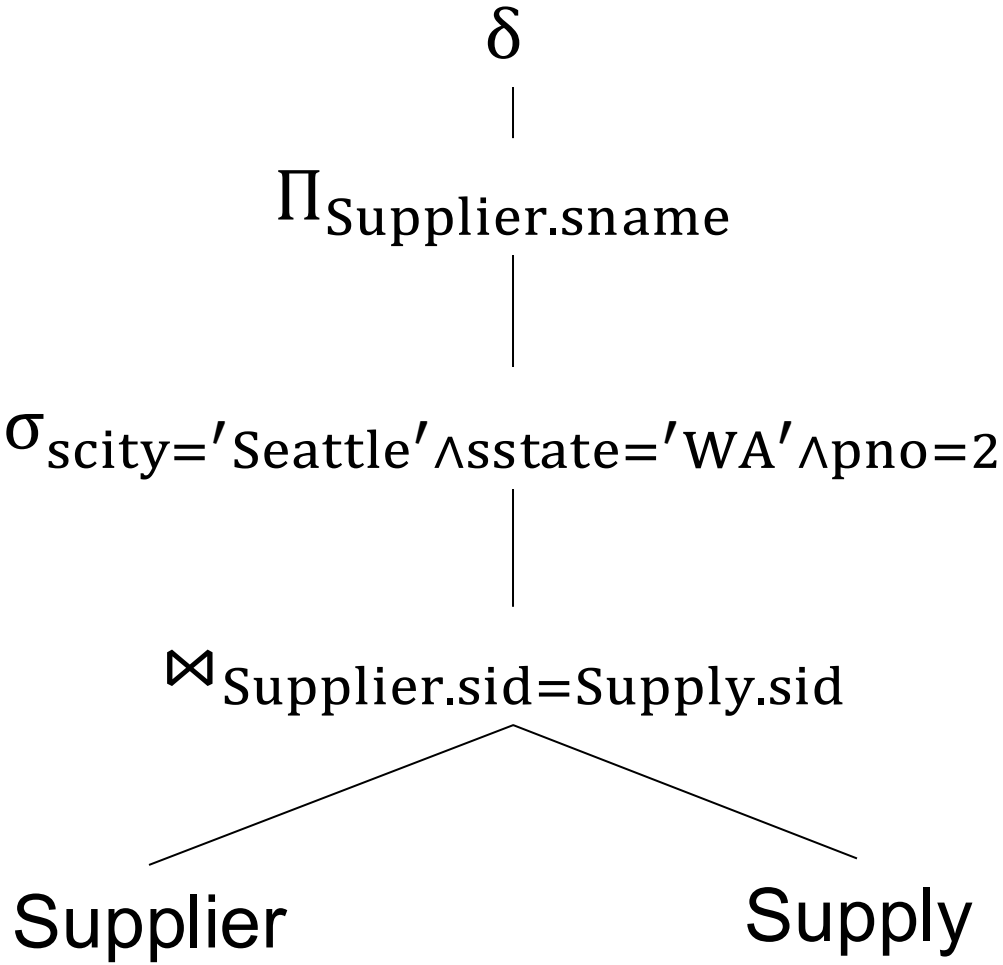
```
SELECT DISTINCT x.sname
FROM Supplier x, Supply y
WHERE x.sno=y.sno
      and x.scity='Seattle'
      and x.sstate='WA'
      and y.pno=2
```



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

Logical Query Plan

```
SELECT DISTINCT x.sname
FROM Supplier x, Supply y
WHERE x.sno=y.sno
      and x.scity='Seattle'
      and x.sstate='WA'
      and y.pno=2
```



Notation Convention

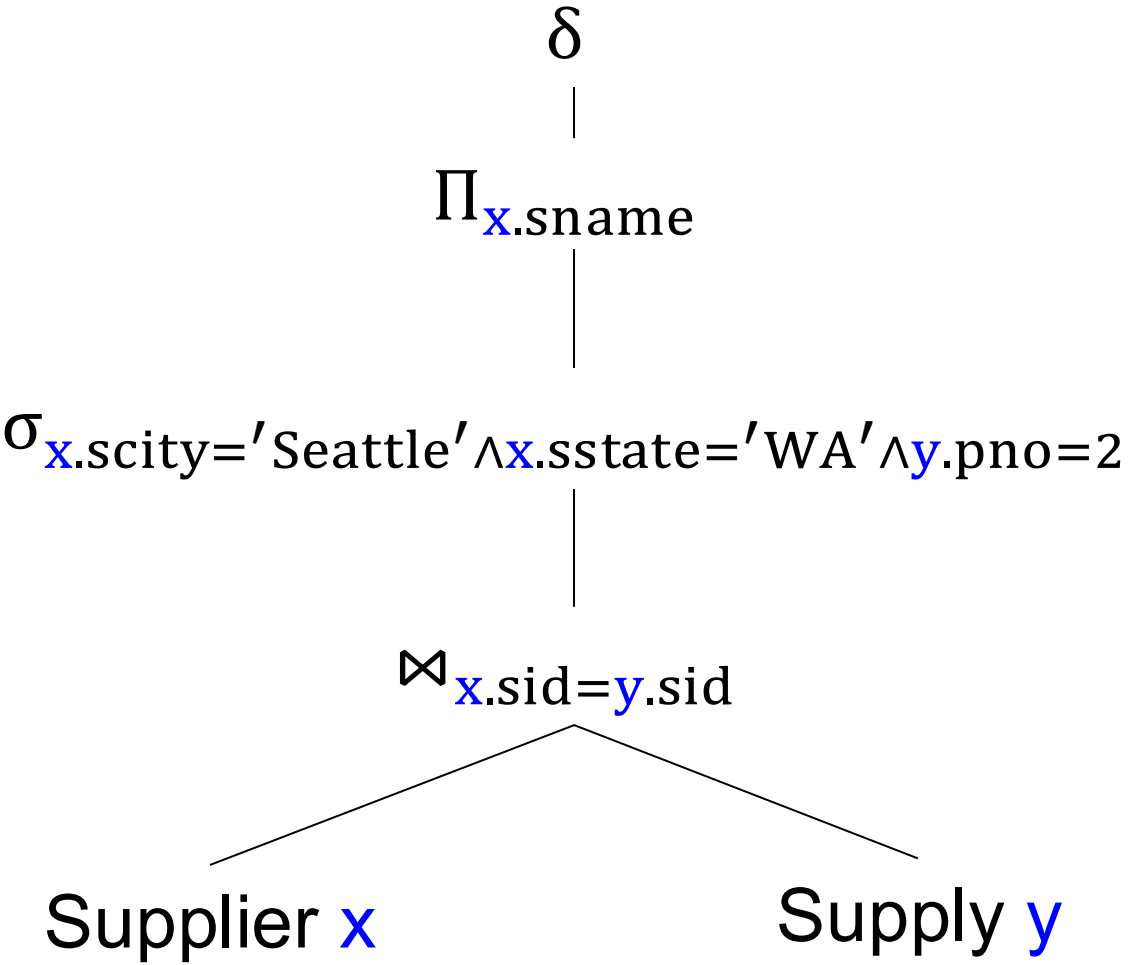
- Database systems have an internal way to represent the schema of the intermediate results, e.g. to deal with duplicate attribute names

- We use a simple convention: add tuple variables to the leaves, refer to them

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

Logical Query Plan

```
SELECT DISTINCT x.sname
FROM Supplier x, Supply y
WHERE x.sno=y.sno
      and x.scity='Seattle'
      and x.sstate='WA'
      and y.pno=2
```



Physical Query Plan

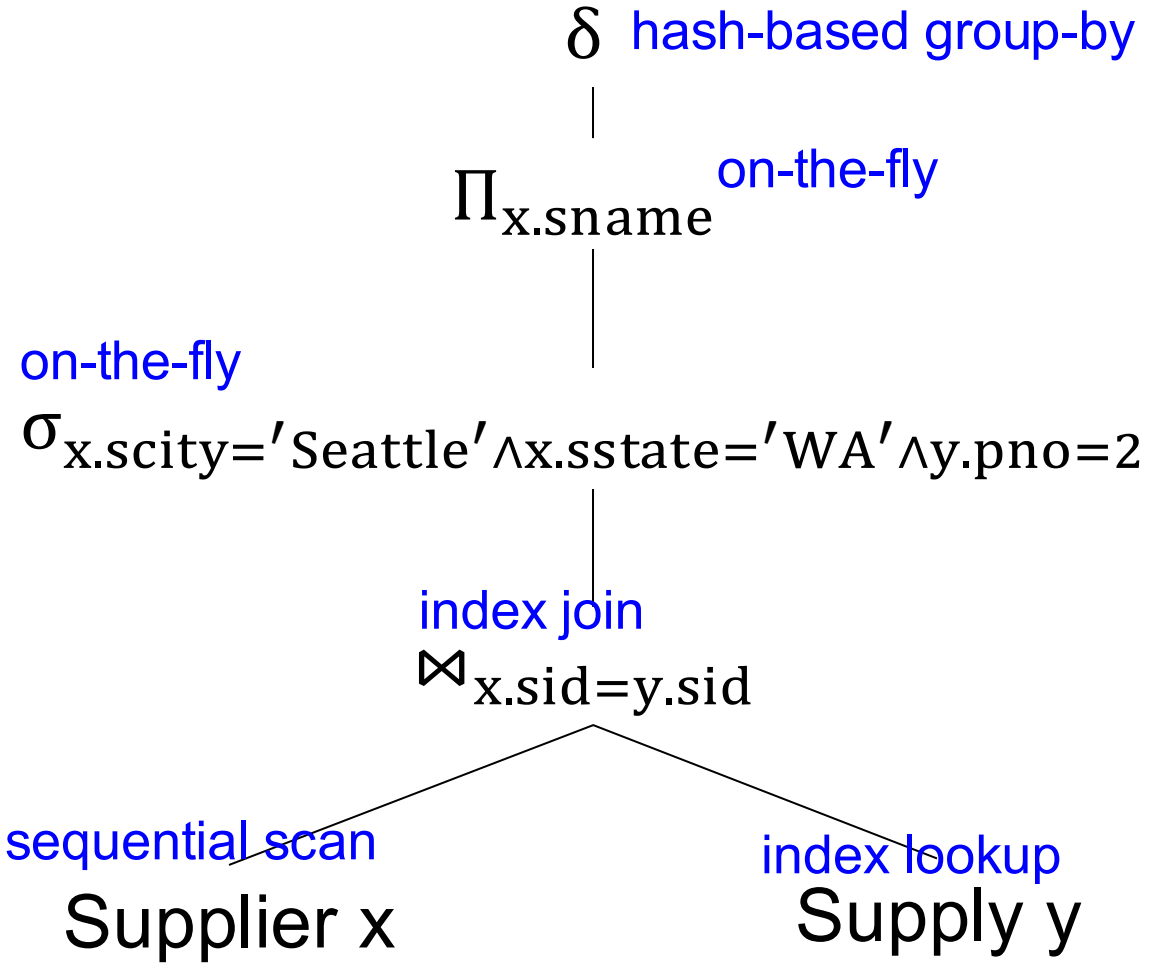
Annotate each operator with an algorithm that implements it

- Operator+algorithm = physical operator
- Annotated plan = physical plan

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

Physical Query Plan

```
SELECT DISTINCT x.sname
FROM Supplier x, Supply y
WHERE x.sno=y.sno
      and x.scity='Seattle'
      and x.sstate='WA'
      and y.pno=2
```



Reminder

- Relational algebra does not have subqueries: no subquery in σ
- Instead, optimizers needs to unnest the query

Physical Operators

Physical Operators

A few physical ops for each logical op:

- Joins (including anti-semijoin):
  - Nested loops, hash-join, merge-join
- Group-by: same…
- Selection: on-the-fly, index-based

Pedagogically, we classify physical ops into main-memory and external-memory

Two Basic Techniques: Hashing, Sorting

Hash Tables

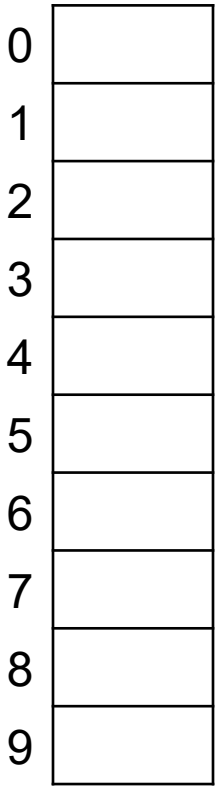
- Array: map indices to memory locations
  - A[0], A[1], A[2], … sequential in memory
- How to map texts to memory locations?
  - A[“alice”], A[“bob”],A[“carl”]…???
- Hash function: maps strings to indices

Separate chaining:

Hash Tables

A (naïve) hash function:

h(“abc”) = (‘a’+’b’+’c’) mod 10

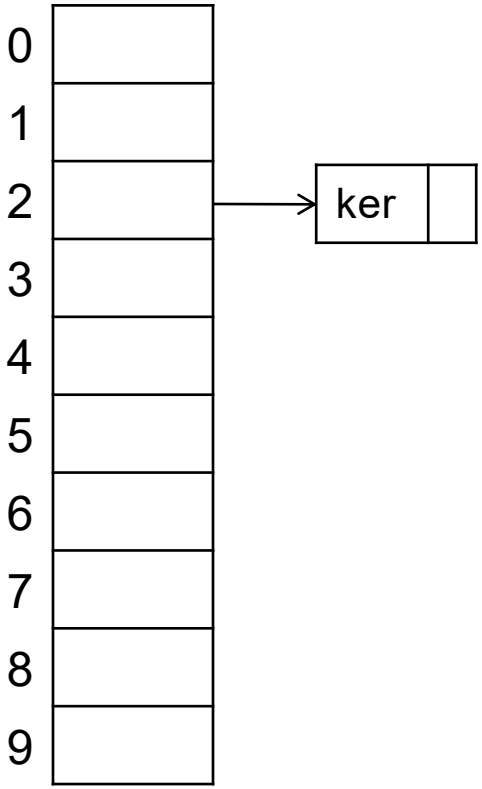


Separate chaining:

Hash Tables

A (naïve) hash function:

h(“abc”) = (‘a’+’b’+’c’) mod 10



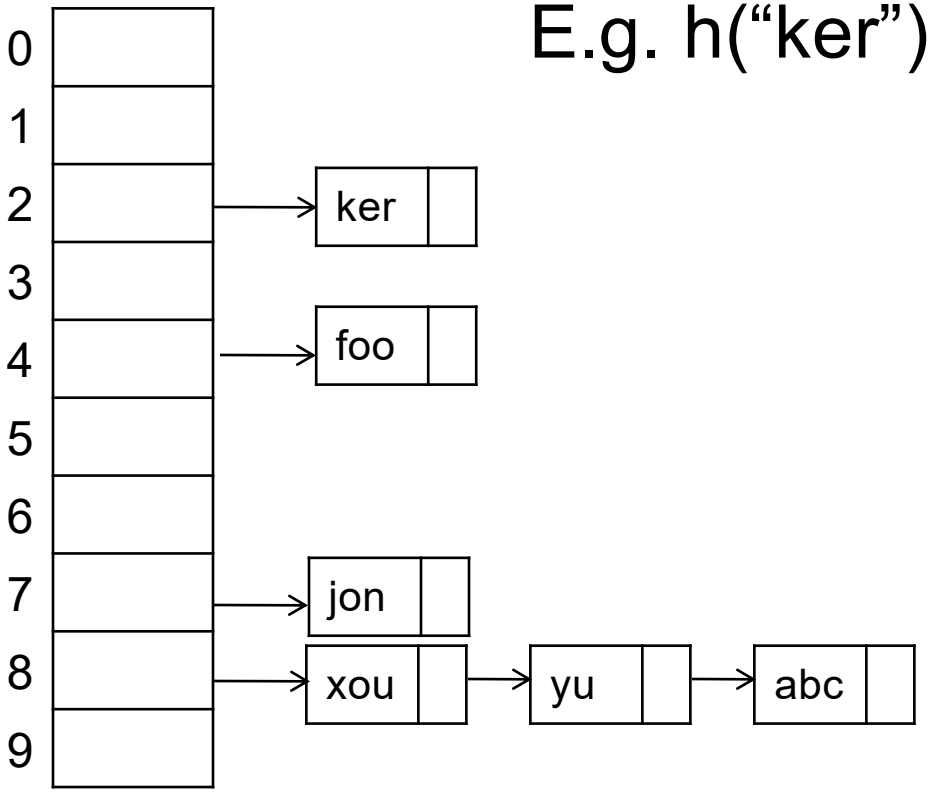
E.g. h(“ker”) = (‘k’+’e’+’r’) mod 10
= (107+101+114) mod 10
= 2

Separate chaining:

Hash Tables

A (naïve) hash function:

h(“abc”) = (‘a’+’b’+’c’) mod 10



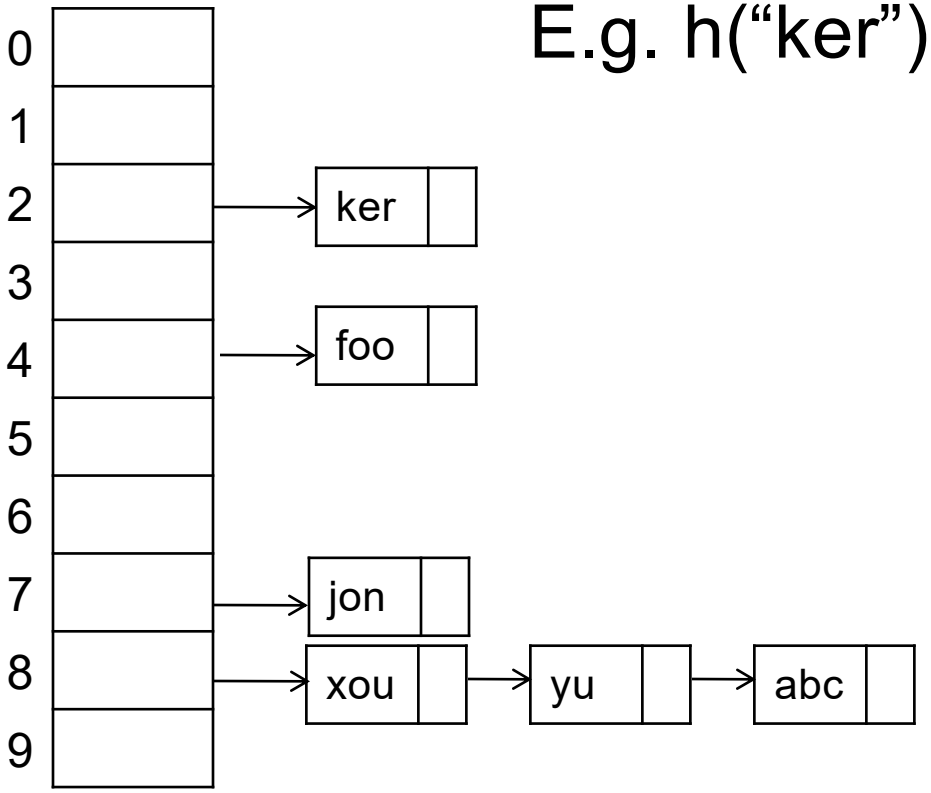
E.g. h(“ker”) = (‘k’+’e’+’r’) mod 10
= (107+101+114) mod 10
= 2

Separate chaining:

Hash Tables

A (naïve) hash function:

h(“abc”) = (‘a’+’b’+’c’) mod 10



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

find(“yu”) = ??

insert(“alice”) = ??

Hash Tables: Summary

Key/value pairs

- insert(k, v)
- find(k) = returns the ***list*** of values v
- Time O(1), but can be O(n): WHEN?

Hash Tables: Summary

Key/value pairs

- insert(k, v)
- find(k) = returns the ***list*** of values v
- Time O(1), but can be O(n): WHEN?
  - Array is too small
  - Too many collisions due to hash function
  - Data skew

Hash Tables: Summary

Key/value pairs

- insert(k, v)
- find(k) = returns the ***list*** of val
- Time O(1), but can be
  - Array is too small
  - Too many collisions due to hash function
  - Data skew

Estimate N correctly;
Or reallocate

Hash Tables: Summary

Key/value pairs

- insert(k, v)
- find(k) = returns the ***list*** of v
- Time O(1), but can be
  - Array is too small
  - Too many collisions due to hash function
  - Data skew

Estimate N correctly;
Or reallocate

Never!
(Analysis next)

Hash Tables: Summary

Key/value pairs

- insert(k, v)
- find(k) = returns the ***list*** of v
- Time O(1), but can be
  - Array is too small
  - Too many collisions due to hash function
  - Data skew

Estimate N correctly;
Or reallocate

Never!
(Analysis next)

Will analyze too

Analysis

- The time for find(k) is O(|Bucket(h(k))|)
  - How large are the buckets?
- Parameters:
  - M=array size. E.g. M=100000
  - N=data size. E.g. N=1000000
- Average bucket size = N/M E.g. size=10

What is the maximum bucket size?

Analysis

There is no deterministic "good" hash function:

- For any hash function h, we can choose N data items that all hash to the same bucket

A "good" hash function is random function

What is the probability that the max bucket is small?

The Cernoff Bound

Note: very many variants

Bernoulli r.v.: $X_1, \ldots, X_N \in \{0,1\}$

For all i, $\Pr(X_i = 1) = \mu \in (0,1)$

We are interested in $Y = X_1 + X_2 + \cdots + X_N$

Fact: $E[Y] = N\mu$

Theorem (Cernoff bound). If $Y = \sum X_i$

$$\Pr(Y > (1+\delta)E[Y]) \leq exp\left(-\frac{\delta^2}{3}E[Y]\right)$$

Theorem (Cernoff bound). If $Y = \sum X_i$

$$\Pr(Y > (1+\delta)E[Y]) \leq exp\left(-\frac{\delta^2}{3}E[Y]\right)$$

Analysis of Hash Tables

Theorem (Cernoff bound). If $Y = \sum X_i$

$$\Pr(Y > (1+\delta)E[Y]) \leq exp\left(-\frac{\delta^2}{3}E[Y]\right)$$

Analysis of Hash Tables

Bucket j: $\text{Size}(j) = X_1 + X_2 + \cdots + X_N$

Expected size: $\text{E}[\text{X}_\text{i}] = \frac{1}{M}$; $\text{E}[\text{Size}(j)] = \frac{N}{M}$

Theorem (Cernoff bound). If $Y = \sum X_i$

$$\Pr(Y > (1+\delta)E[Y]) \leq exp\left(-\frac{\delta^2}{3}E[Y]\right)$$

Analysis of Hash Tables

Bucket j: $\text{Size}(j) = X_1 + X_2 + \cdots + X_N$

Expected size: $\text{E}[\text{X}_\text{i}] = \frac{1}{M}$; $\text{E}[\text{Size}(j)] = \frac{N}{M}$

Skew at j

Cernoff: $\Pr\left(\text{Size}(j) > (1+\delta)\frac{N}{M}\right) \leq exp\left(-\frac{\delta^2}{3}\frac{N}{M}\right)$

Theorem (Cernoff bound). If $Y = \sum X_i$

$$\Pr(Y > (1+\delta)E[Y]) \leq exp\left(-\frac{\delta^2}{3}E[Y]\right)$$

Analysis of Hash Tables

Bucket j: $\text{Size}(j) = X_1 + X_2 + \cdots + X_N$

Expected size: $\text{E}[\text{X}_\text{i}] = \frac{1}{M}$; $\text{E}[\text{Size}(j)] = \frac{N}{M}$

Skew at j

Cernoff: $\Pr\left(\text{Size}(j) > (1+\delta)\frac{N}{M}\right) \leq exp\left(-\frac{\delta^2}{3}\frac{N}{M}\right)$

Union bound

$$\Pr(\delta\text{−Skew}) \leq M \cdot exp\left(-\frac{\delta^2}{3}\frac{N}{M}\right)$$

$$\Pr(\delta\text{−Skew}) \leq M \cdot exp\left(-\frac{\delta^2}{3}\frac{N}{M}\right)$$

Example 1

N=2,000,000 items, M=100 buckets:
Average bucket size = 20,000

What is the prob that max bucket size is > 10% avg size?

$$\Pr(\delta\text{−Skew}) \leq M \cdot exp\left(-\frac{\delta^2}{3}\frac{N}{M}\right)$$

Example 1

N=2,000,000 items, M=100 buckets:
Average bucket size = 20,000

What is the prob that max bucket size is > 10% avg size?

$$Pr(\delta\text{−skew}) \leq 100 \times \exp\left(-\frac{0.1^2}{3} \times 20000\right) = 100 \times \exp(-66)$$

Virtually zero!

$$\Pr(\delta\text{−Skew}) \leq M \cdot exp\left(-\frac{\delta^2}{3}\frac{N}{M}\right)$$

Example 2

N=2,000,000 items, M=10,000 buckets:
Average bucket size = 200

A 10% skew is certain

What is the prob that max bucket size is > 10% avg size?

$$Pr(\delta\text{−skew}) \leq 10000 \times \exp\left(-\frac{0.1^2}{3}200\right) = 10000\exp\left(-\frac{2}{3}\right) = 5134$$

$$\Pr(\delta\text{−Skew}) \leq M \cdot exp\left(-\frac{\delta^2}{3}\frac{N}{M}\right)$$

Example 2

N=2,000,000 items, M=10,000 buckets:
Average bucket size = 200

A 10% skew is certain

What is the prob that max bucket size is > 10% avg size?

$$Pr(\delta\text{−skew}) \leq 10000 \times \exp\left(-\frac{0.1^2}{3}200\right) = 10000\exp\left(-\frac{2}{3}\right) = 5134$$

What is the prob that max bucket size is > 100% avg size?

$$Pr(\delta\text{−skew}) \leq 10000 \times \exp\left(-\frac{1^2}{3}200\right) = 10000\exp(-66)$$

Virtually zero!

Skewed Data

- The analysis so far assumed distinct keys
- If the key(tem i) = key(item j), then $X_i = X_j$ and the r.v. are no longer independent
- If some key occurs $\gg \frac{N}{M}$ times, then SKEW
- Rigorous analysis: Chernoff bound

Hash Table Takeaways

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

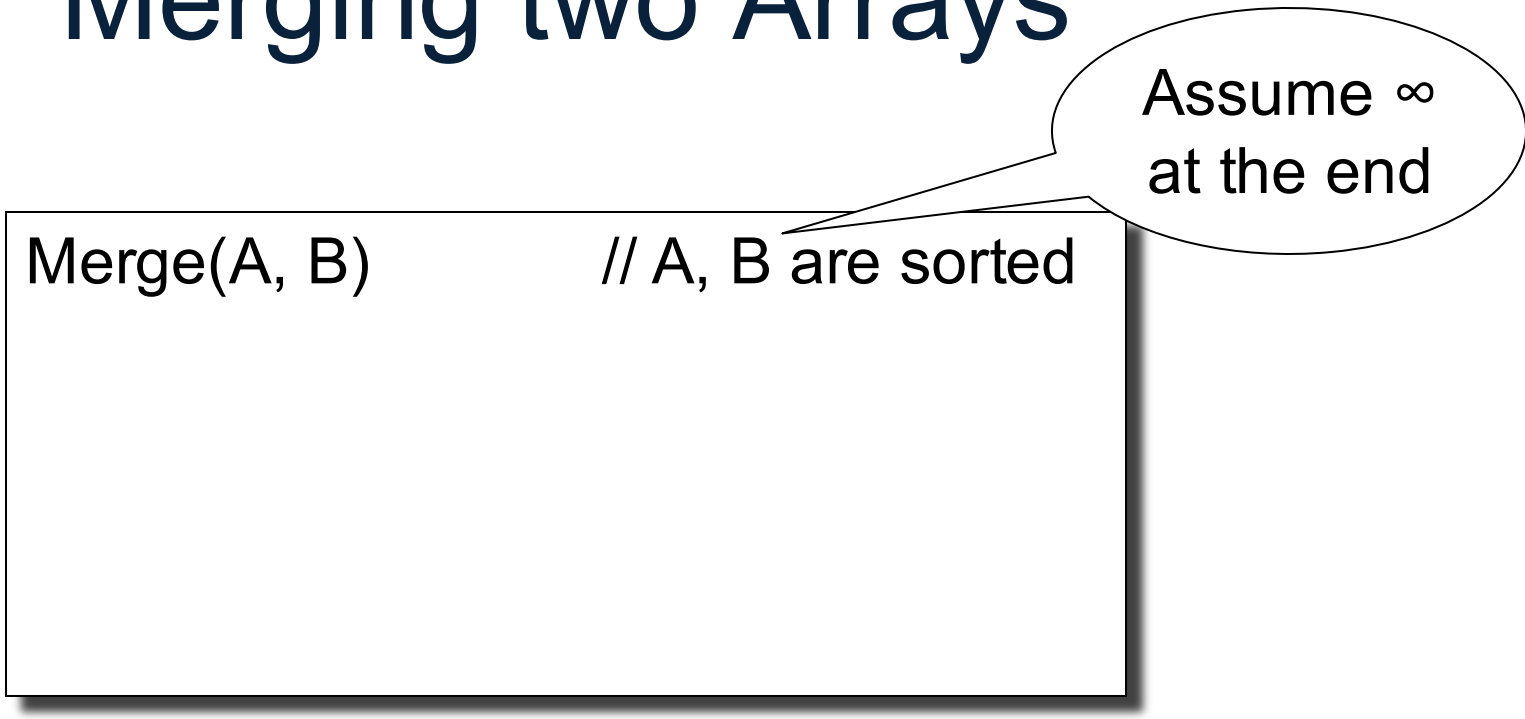
- Simple algorithms: $O(N^2)$
- Quicksort: O(N log N)
- Mergesort: O(N log N)

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



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++]
```

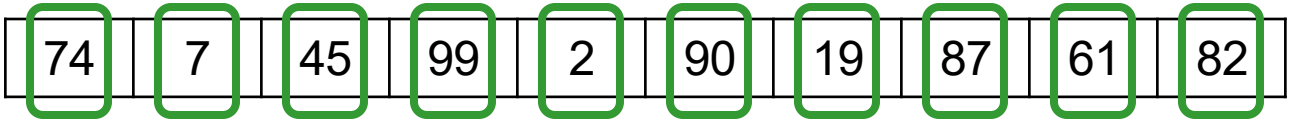
Time = O(|A|+|B|)

Merge-Sort

74	7	45	99	2	90	19	87	61	82

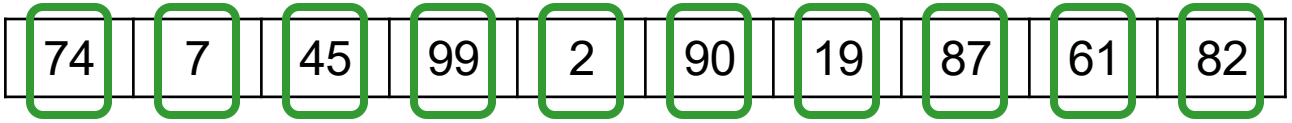
Merge-Sort

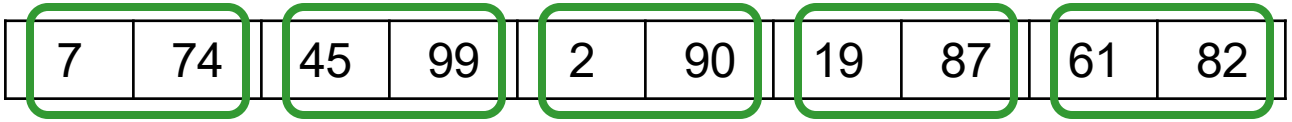
Runs



Merge-Sort

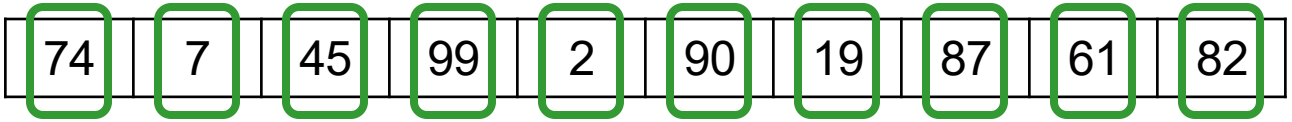
Runs

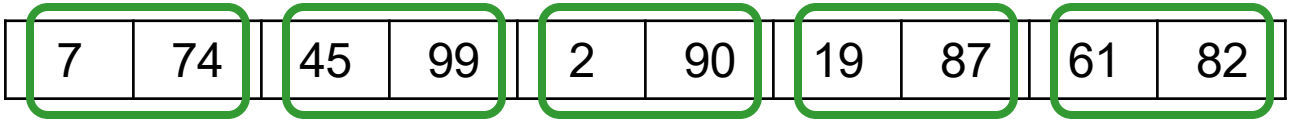


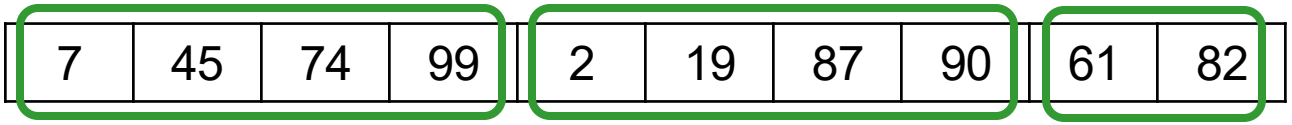


Merge-Sort

Runs

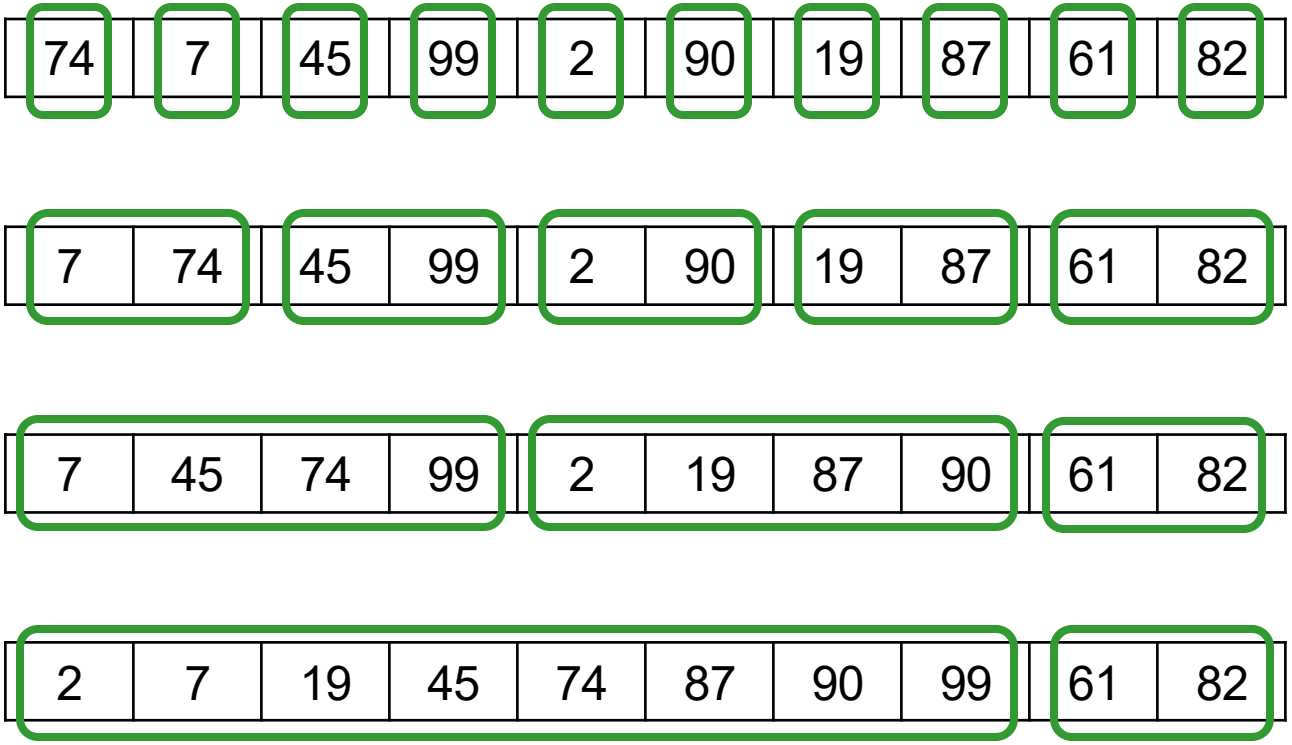






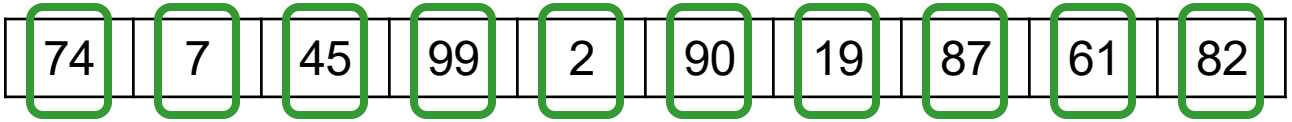
Merge-Sort

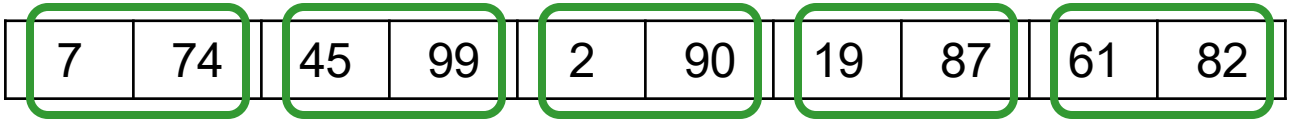
Runs

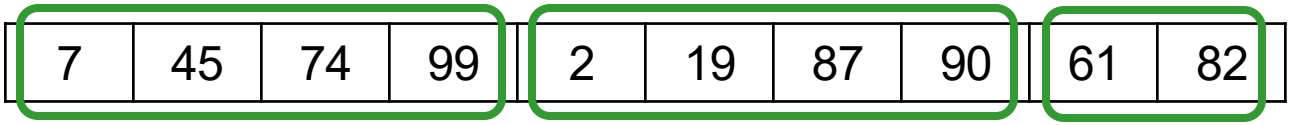


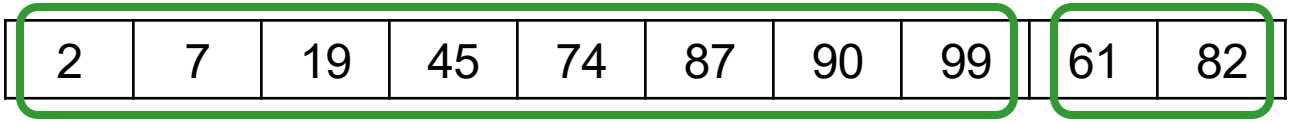
Merge-Sort

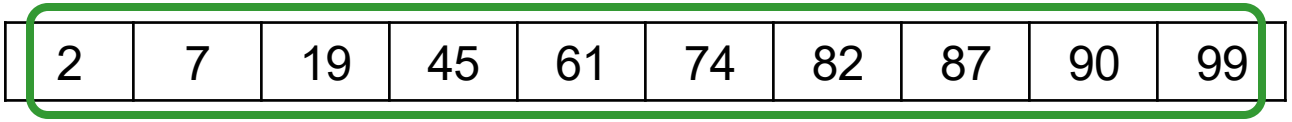
Runs











Summary

- All physical operators for joins and group-by are either nested loops, or based on some form of hashing or merge join
- We will discuss them next, starting with main memory physical operators

Main Memory Operators

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

Main Memory Algorithms

Logical operator:

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

How would you compute the join?

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

Main Memory Algorithms

Logical operator:

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

How would you compute the join?

Three physical operators:

1. Nested Loops
2. Hash-join
3. Merge-join

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

Main Memory Algorithms

Logical operator:

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

Three physical operators:

1. Nested Loops
2. Hash-join
3. Merge-join

Terminology:

$$R \bowtie S$$

R is the outer table

S is the inner table

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

1. Nested Loop Join

Logical operator:

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

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

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

1. Nested Loop Join

Logical operator:

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

```
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?

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

1. Nested Loop Join

Logical operator:

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

```
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?

$O(n^2)$

1. Nested Loop Join

- Simplest implementation of join

- Variations:
  - Index join: we have an index on inner table
  - Block nested loop join (next time)

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

1. Index Join

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)
```

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

1. Index Join

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)
```

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

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

1. Index Join

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)
```

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

O(n)

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

1. Index Join

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)
```

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

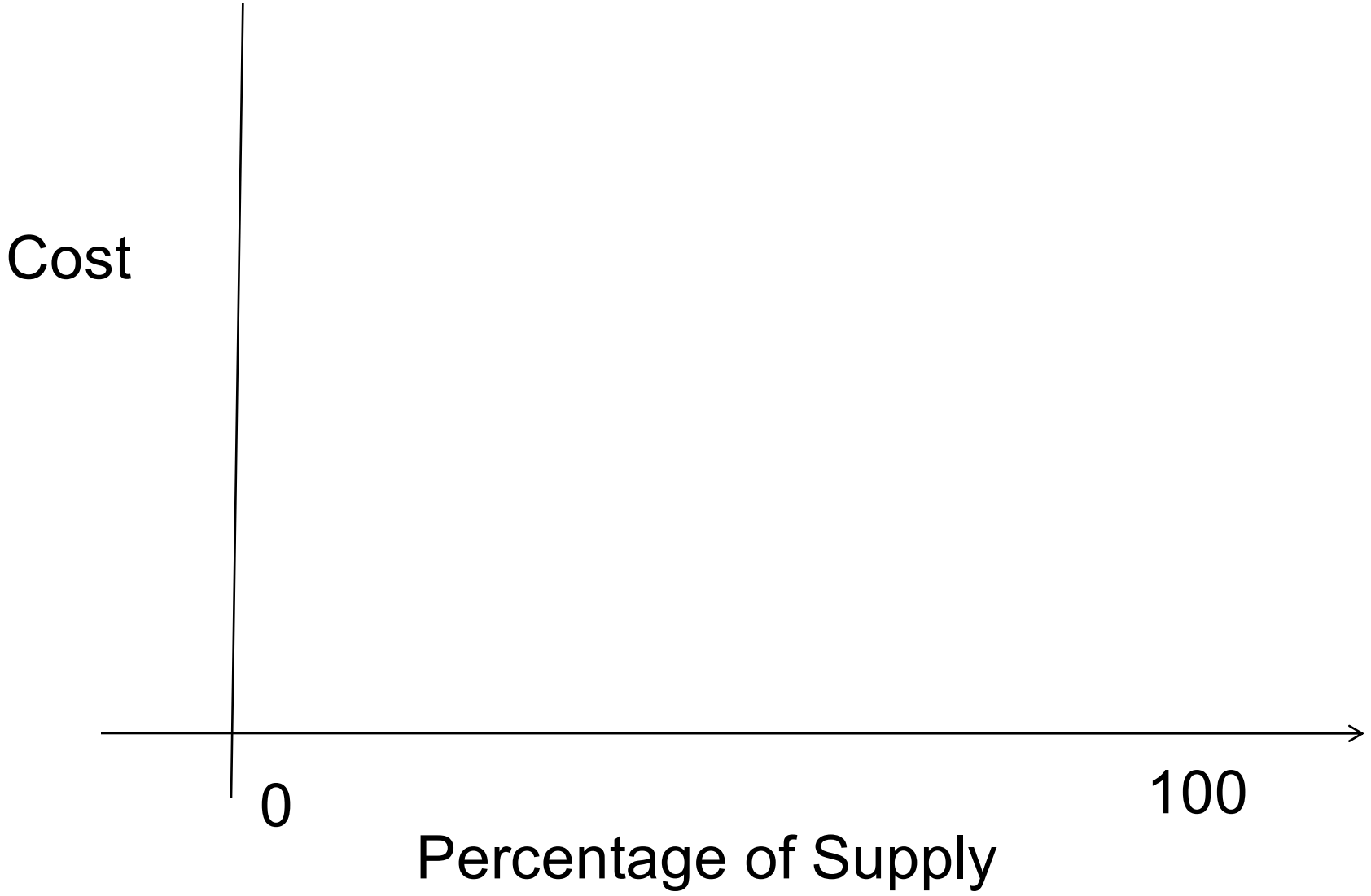
Logical operator:

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

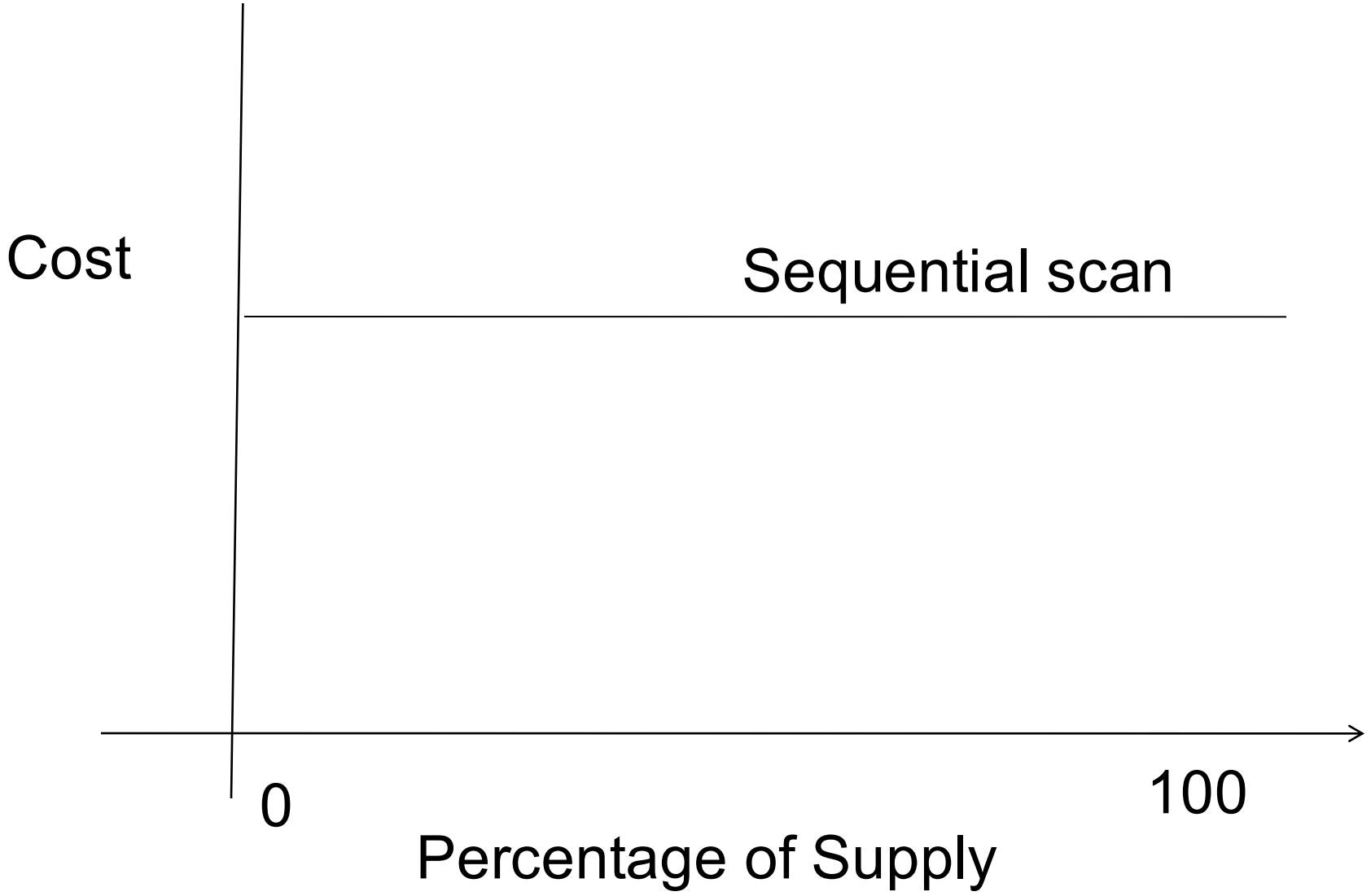
```
for x in Supplier do
   for y in SupplyIndex(x.sno) do
      output(x,y)
```

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

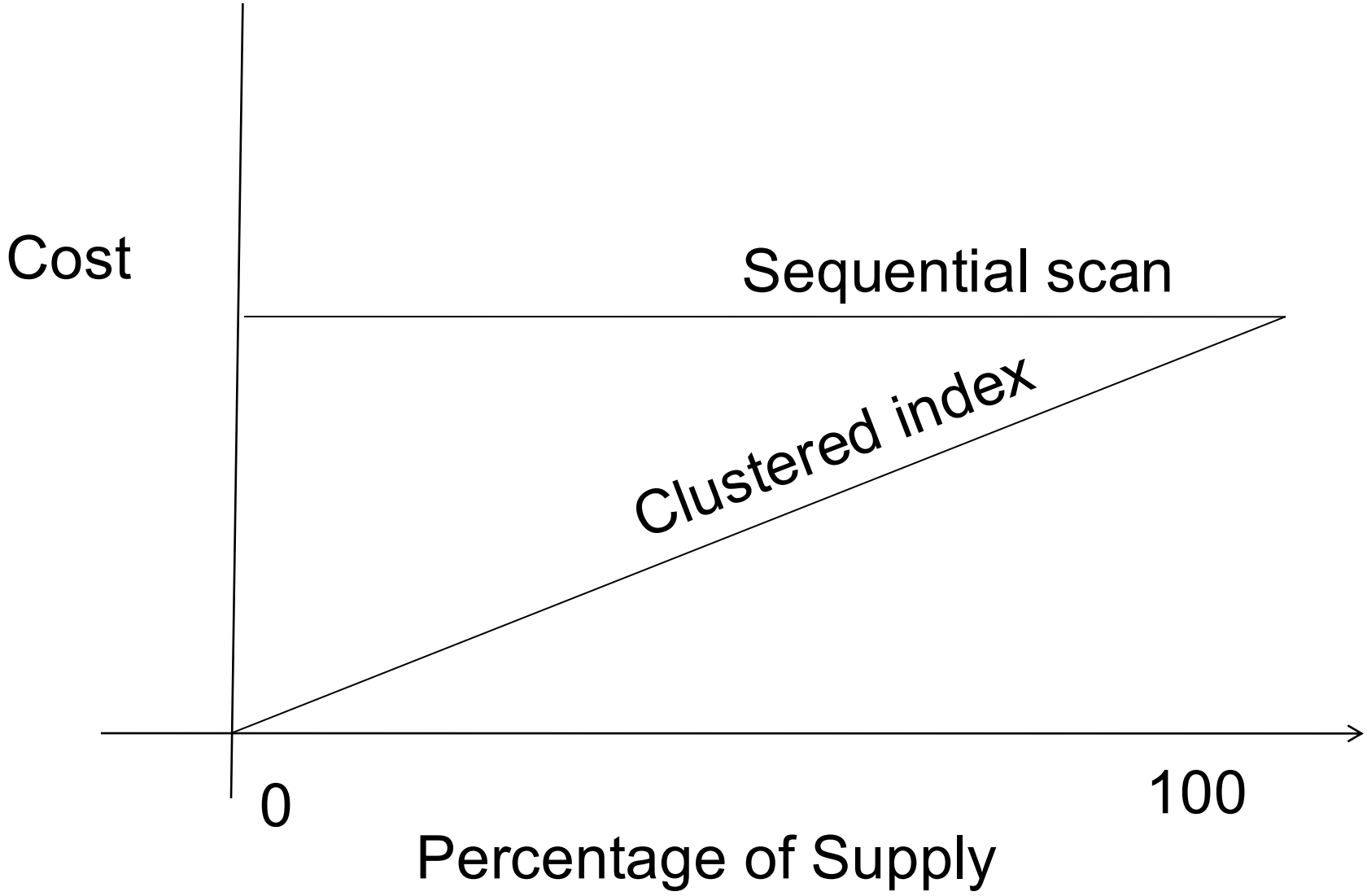
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)
Part(pno,pname,psize,pcolor)



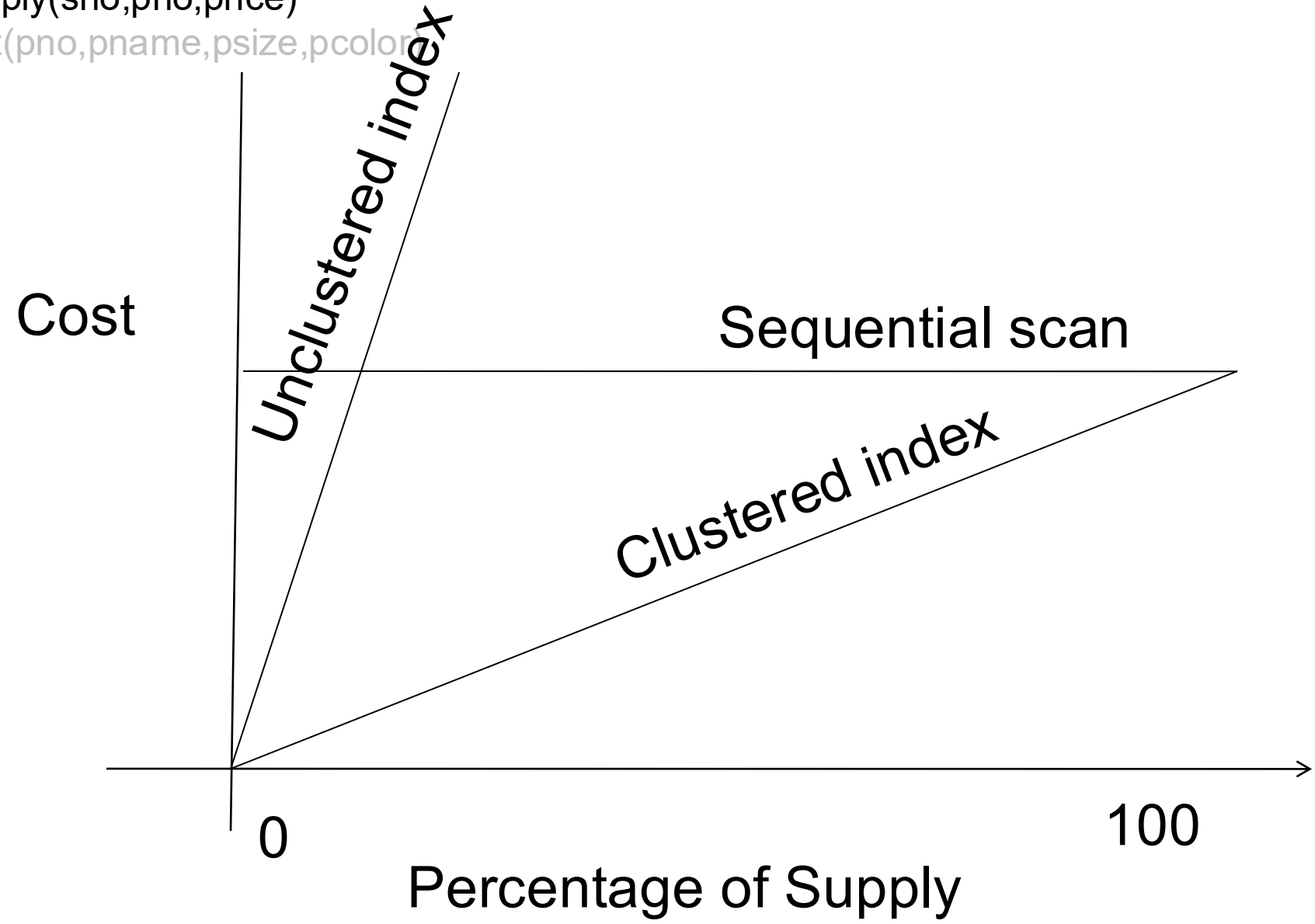
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)
Part(pno,pname,psize,pcolor)



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



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



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

2. Hash Join

Logical operator:

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

Describe it!

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)
```

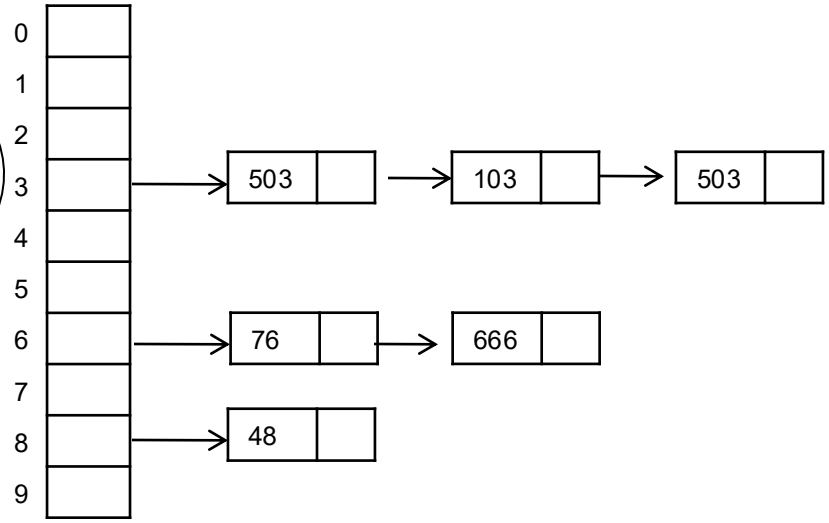
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)
Part(pno,pname,psize,pcolor)

2. Hash Join

Logical operator:

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

Build phase




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

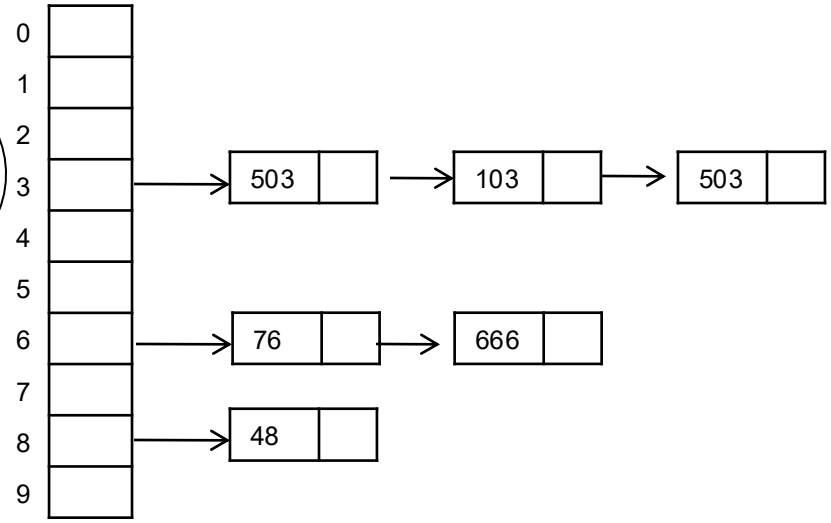
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)
Part(pno,pname,psize,pcolor)

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

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

2. Hash Join

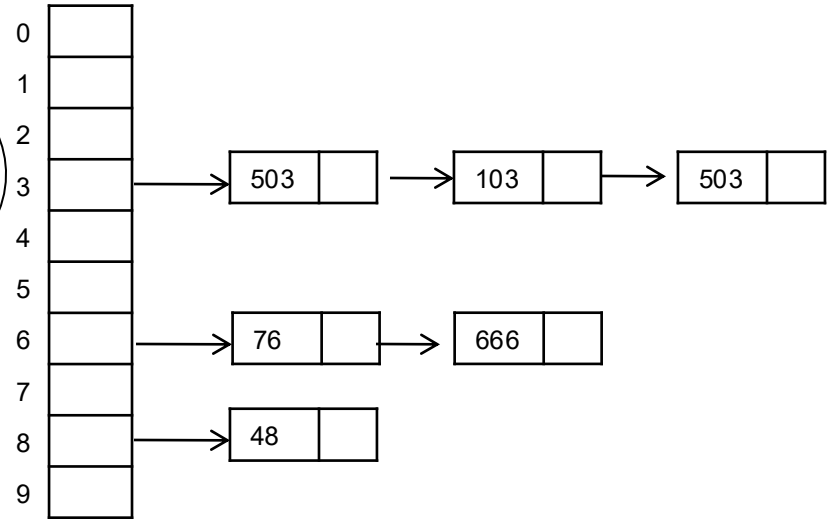
Logical operator:

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

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?

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

2. Hash Join

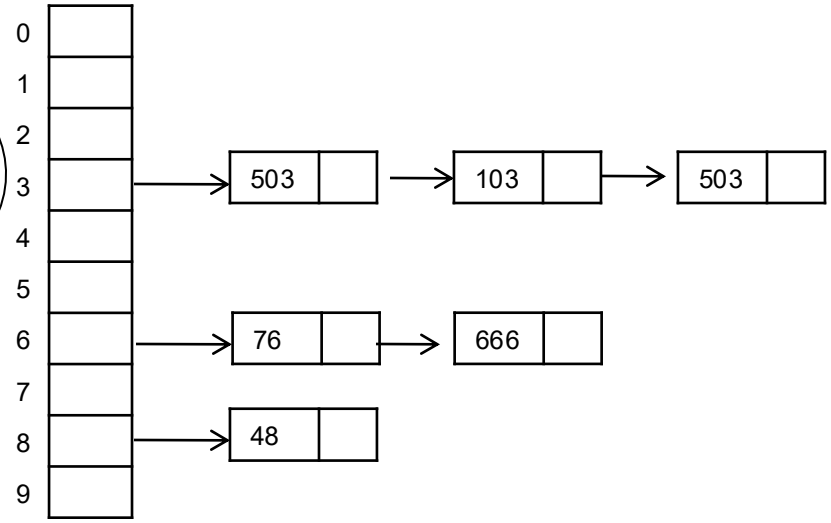
Logical operator:

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

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

Change join order

Logical operator:

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

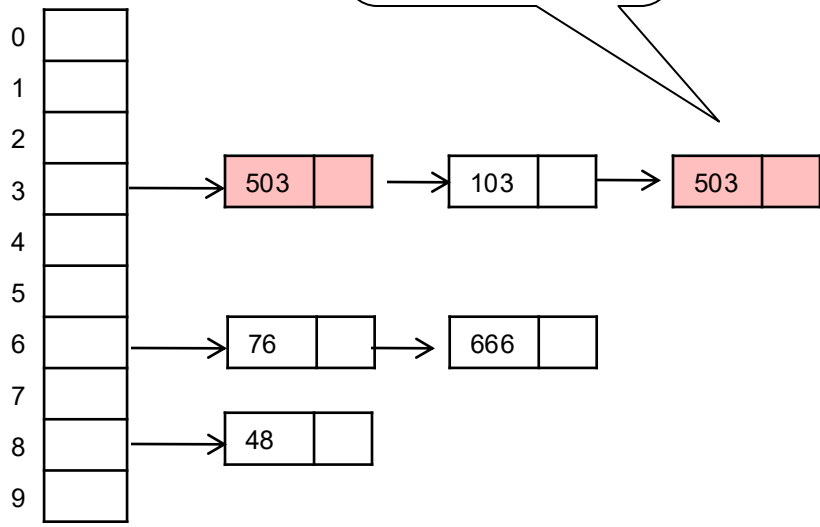
```
for y in Supply do
      insert(y.sno, y)

for x in Supplier do
    ????
```

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

2. Hash Join

Change join order

Duplicate sno's



Logical operator:

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

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

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

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

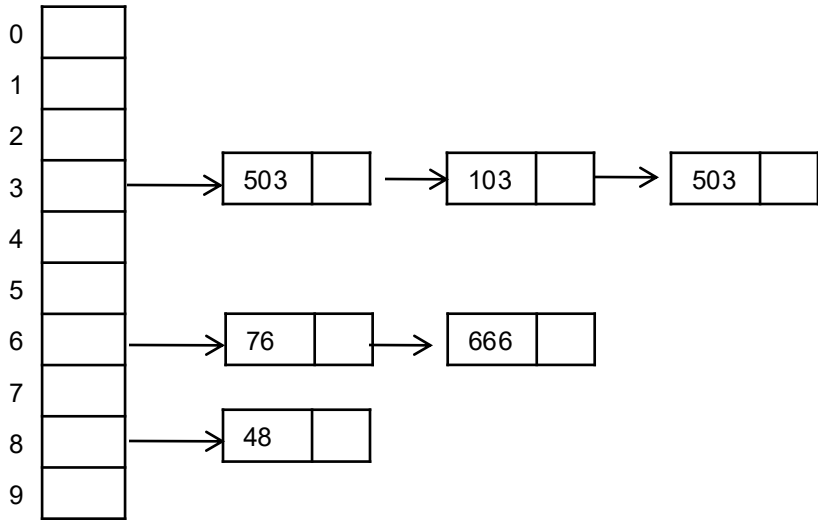
2. Hash Join

Change join order

Logical operator:

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

```
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?

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

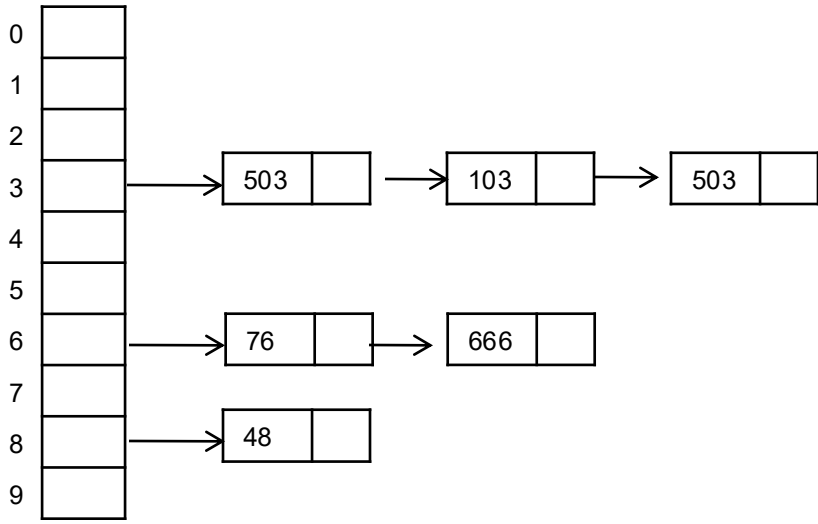
2. Hash Join

Change join order

Logical operator:

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

```
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:

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

Describe it!

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

3. Merge Join

Logical operator:

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

```
Sort(Supplier); Sort(Supply);
x = Supplier.first();
y = Supply.first();
```

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.sno = y.sno: ???
    x.sno > y.sno: ???
```

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: ???
    x.sno > y.sno: ???
```

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

3. Merge Join

Logical operator:

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

```
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: ???
```

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();
```

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

3. Merge Join

Logical operator:

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

```
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?

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

3. Merge Join

Logical operator:

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

```
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))

Summary

- Three join operators:
  - Nested loops, hash join, merge join
- How do you extend these to:
  - Left outer joins?
  - Anti semi-join?
- Group-by: similar physical operators (in class)

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 blocks that 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 blocks that store R
- T(R) = number of records in R
- Sequential read of R: cost = B(R)
- Random read of R: cost = T(R)

Setup

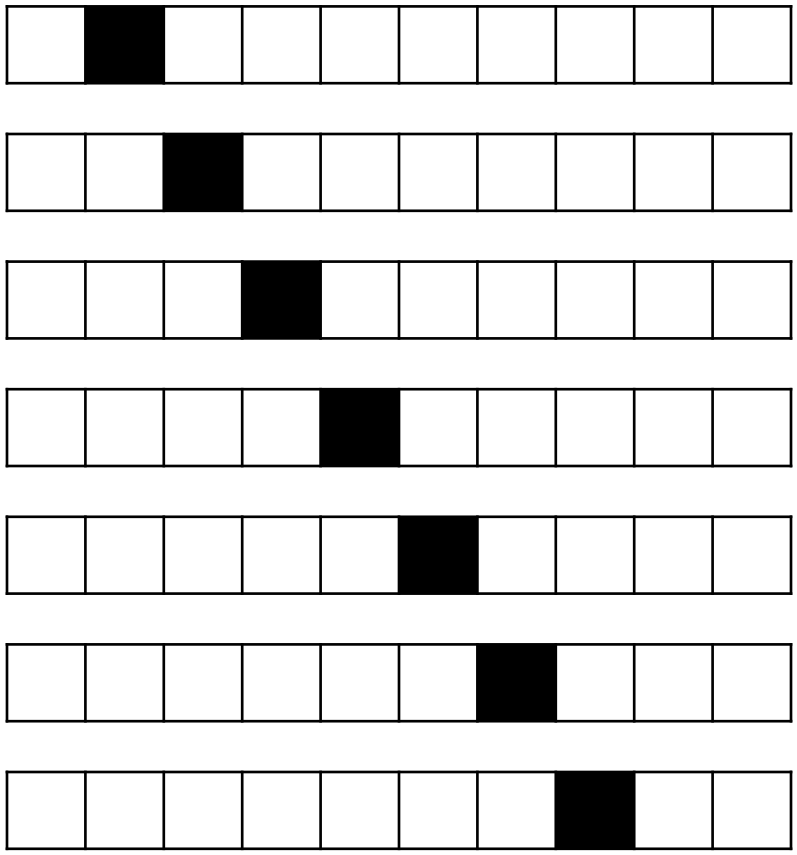
Main cost = number of disk I/O's
Always ignore the final write

- B(R) = number of blocks that store R
- T(R) = number of records in R
- Sequential read of R: cost = B(R)
- Random read of R: cost = T(R)

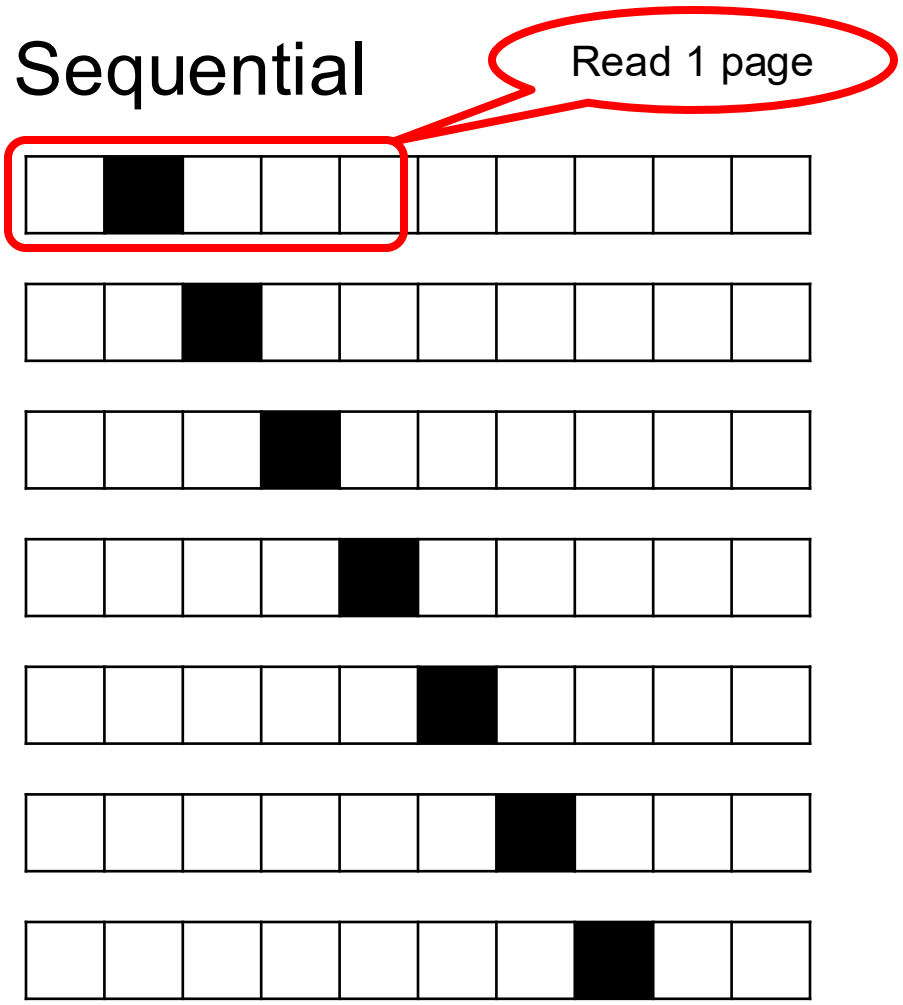
$$T(R) \gg B(R)$$

Sequential/Random Access

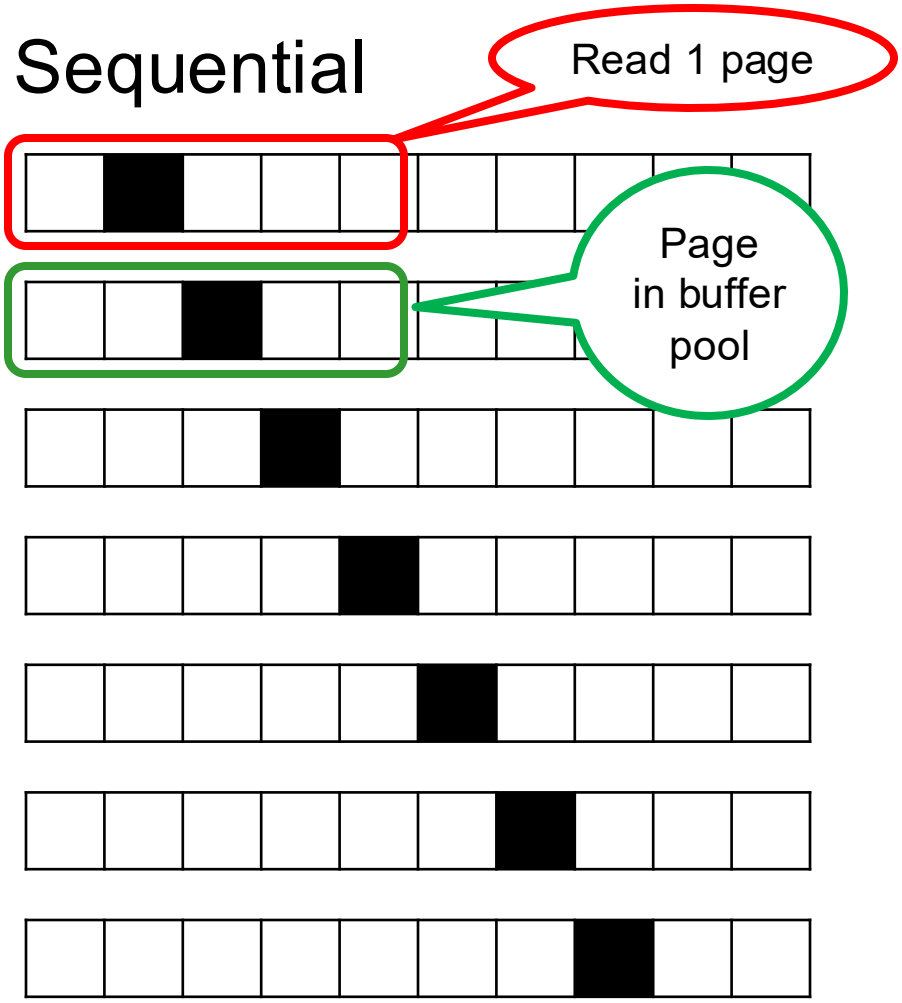
Sequential



Sequential/Random Access

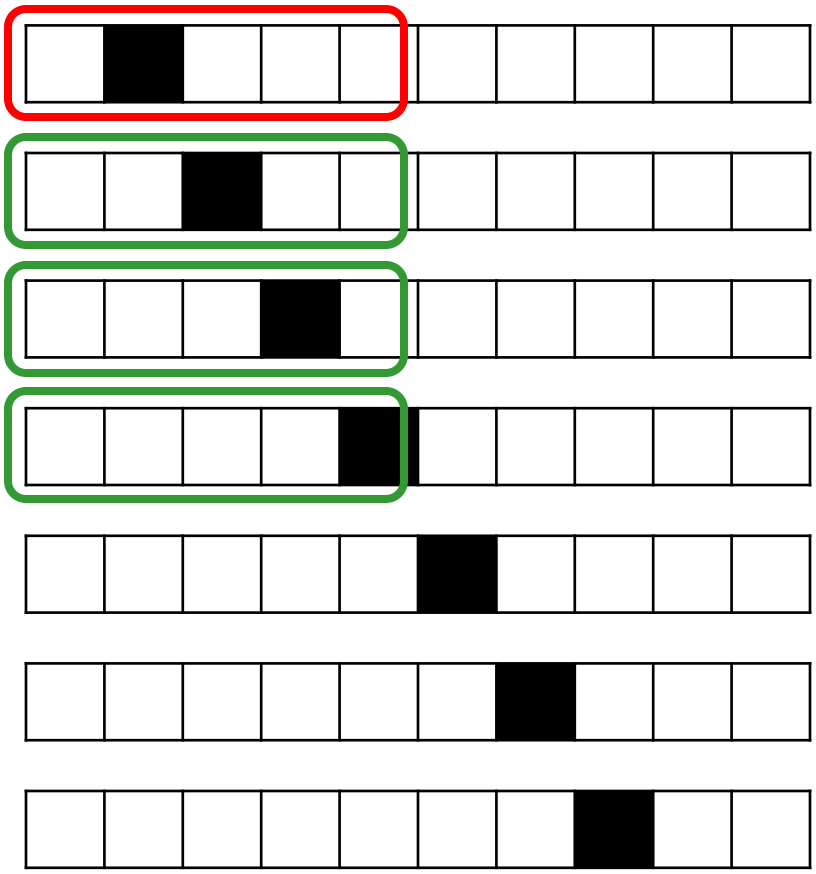


Sequential/Random Access



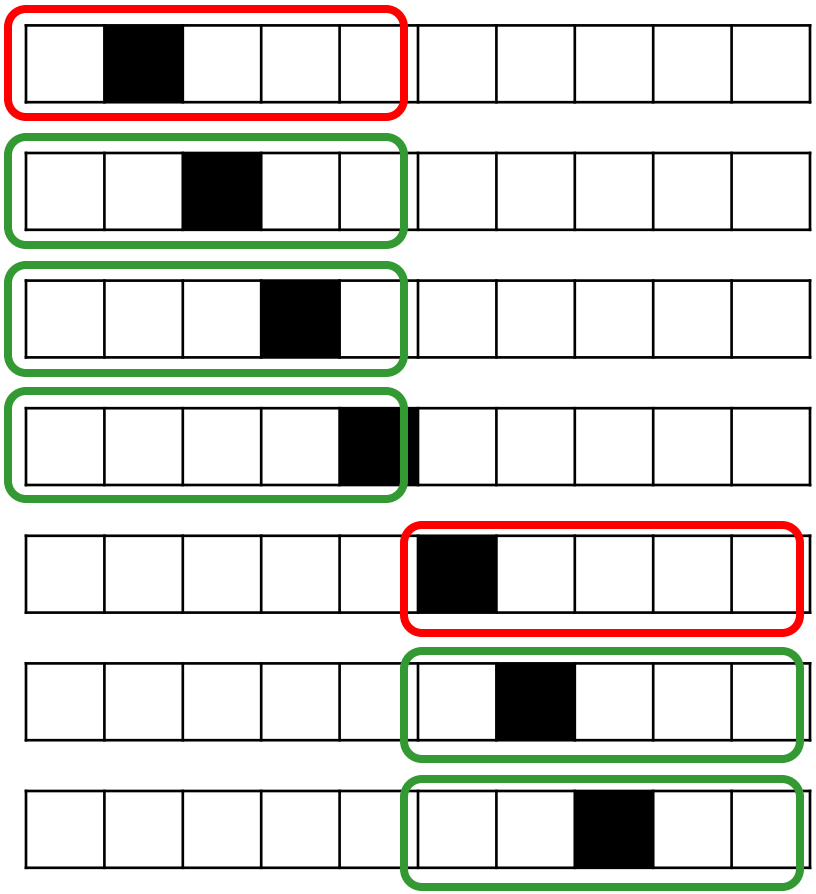
Sequential/Random Access

Sequential



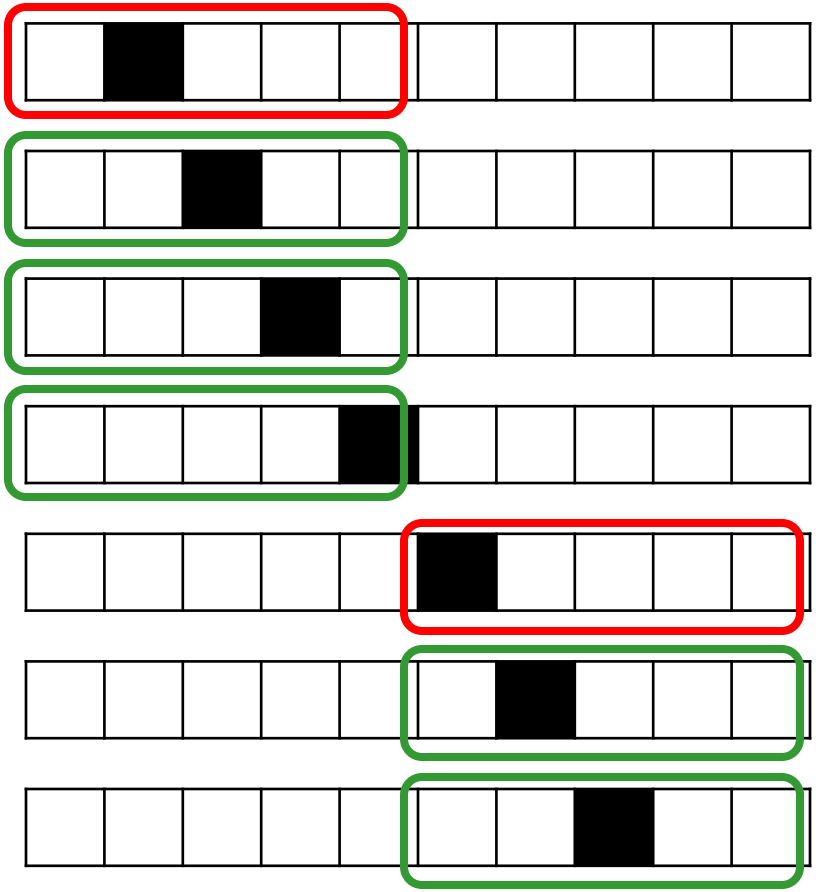
Sequential/Random Access

Sequential: B(R)=2 reads

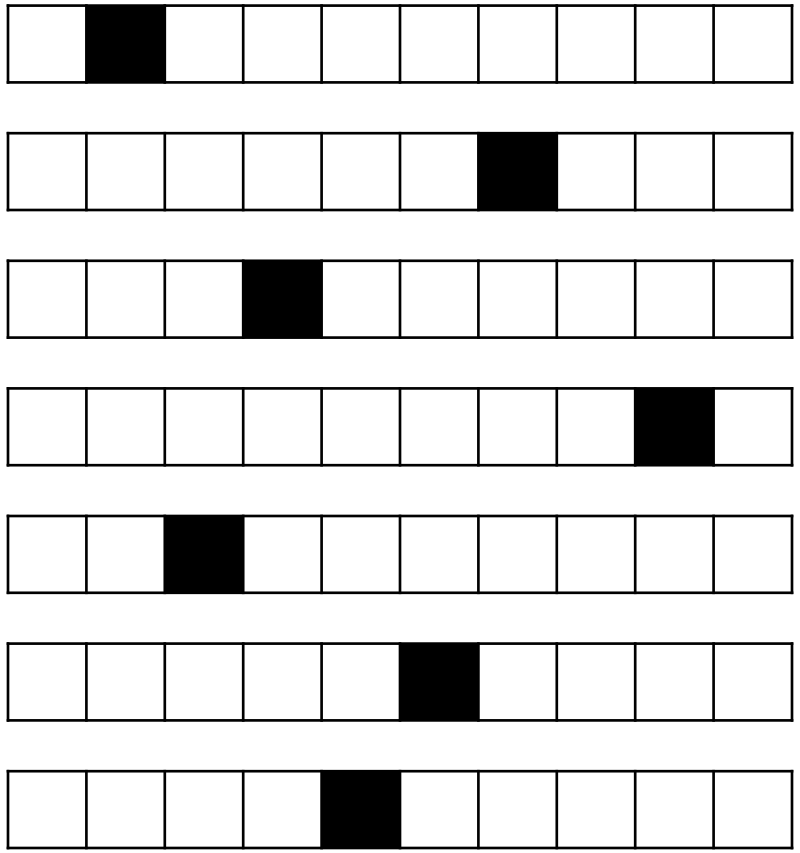


Sequential/Random Access

Sequential: B(R)=2 reads

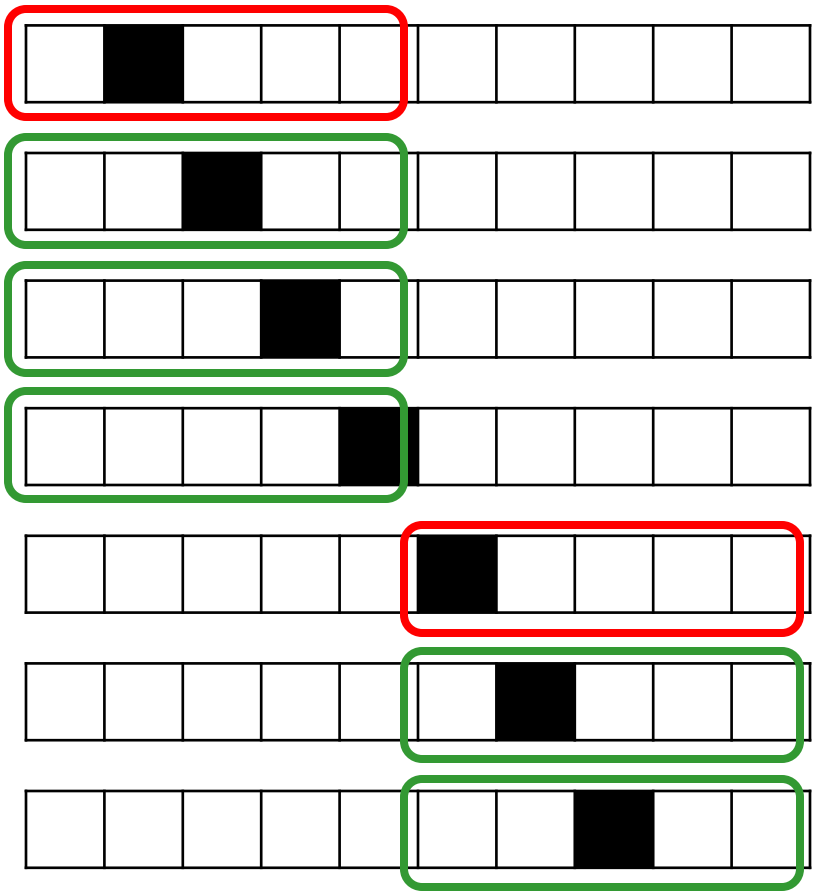


Random

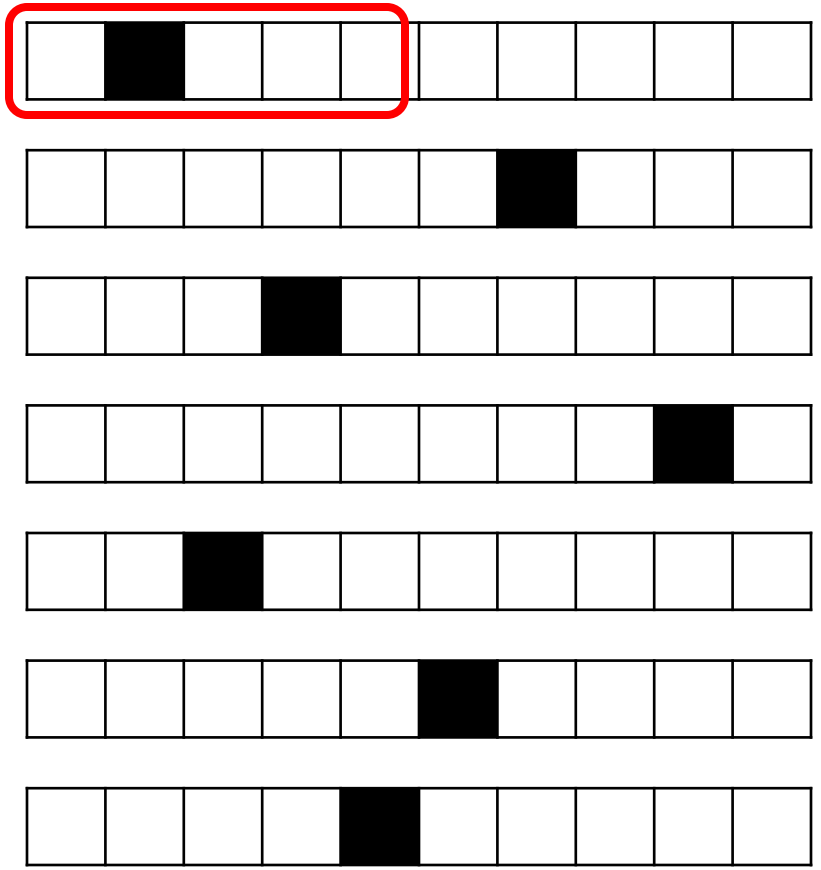


Sequential/Random Access

Sequential: B(R)=2 reads

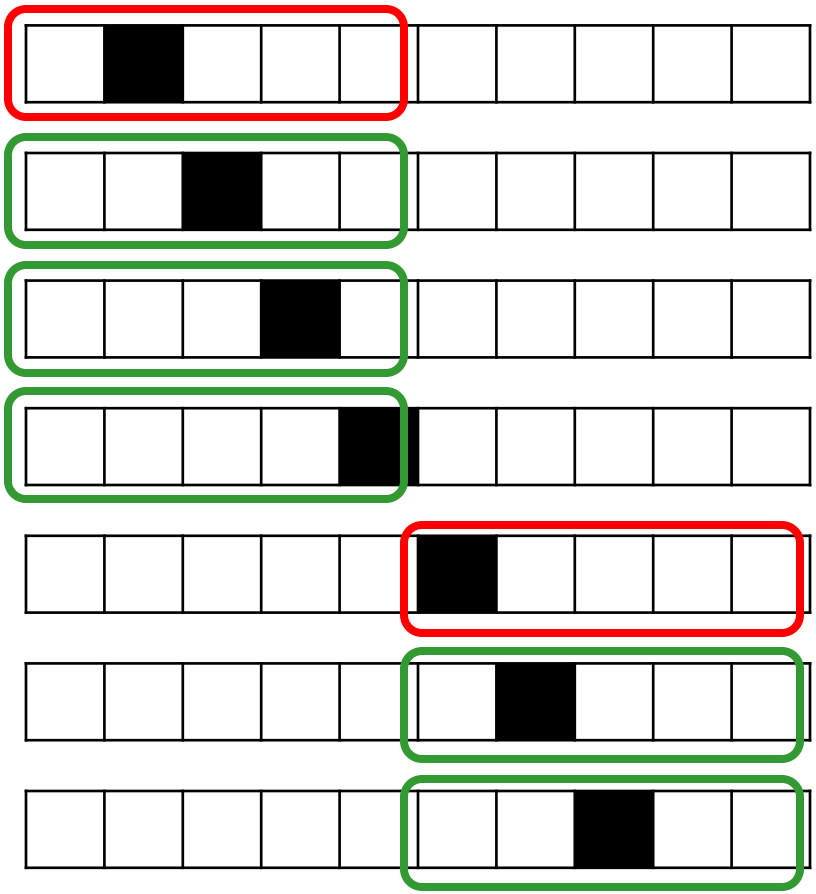


Random

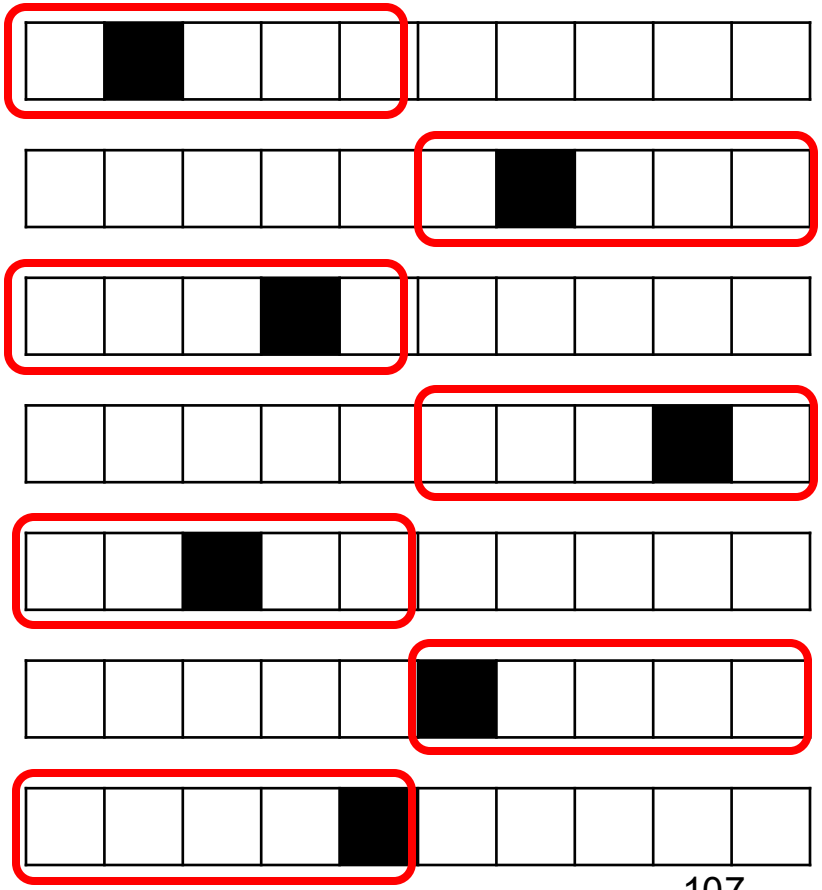


Sequential/Random Access

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)
```

Nested Loop Joins

$R \bowtie S$

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

- Given B(R), B(S), T(R), T(S):
  complexity = ????

Nested Loop Joins

$R \bowtie S$

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

- Given B(R), B(S), T(R), T(S):
  complexity = B(R) + T(R) * B(S)

Nested Loop Joins

$R \bowtie S$

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

- Given B(R), B(S), T(R), T(S):
  complexity = 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

Idea: better use of the available memory

- M = # of blocks that fit in main memory

- Example:

  M = 10,000 // buffer pool frames
  B(R) = 100,000
  T(R) = 10,000,000

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
```

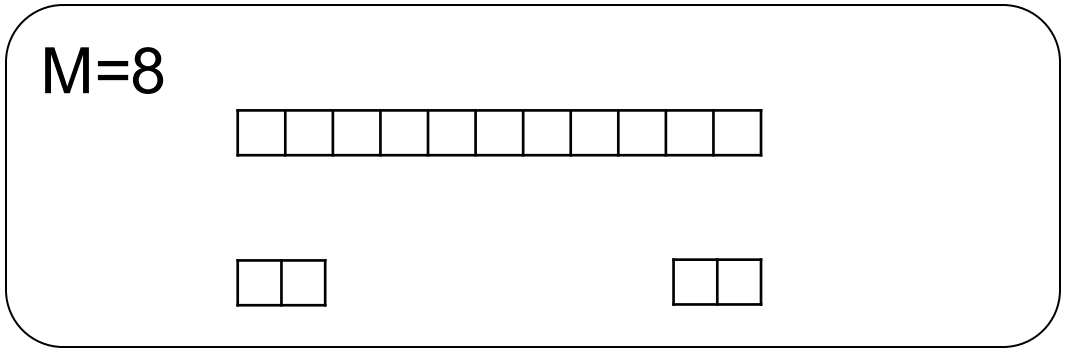
Block Nested Loop Join

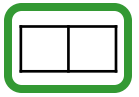
Group of (M-2) pages of R, called a “block”

```
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

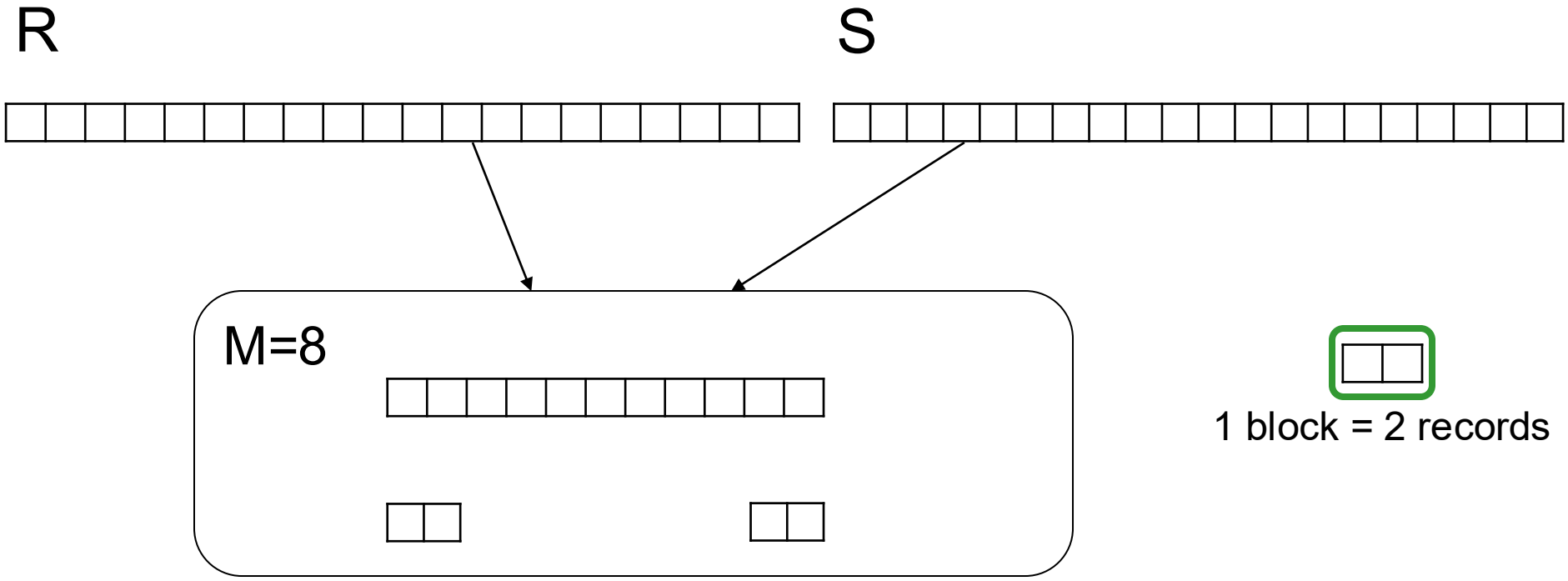




1 block = 2 records

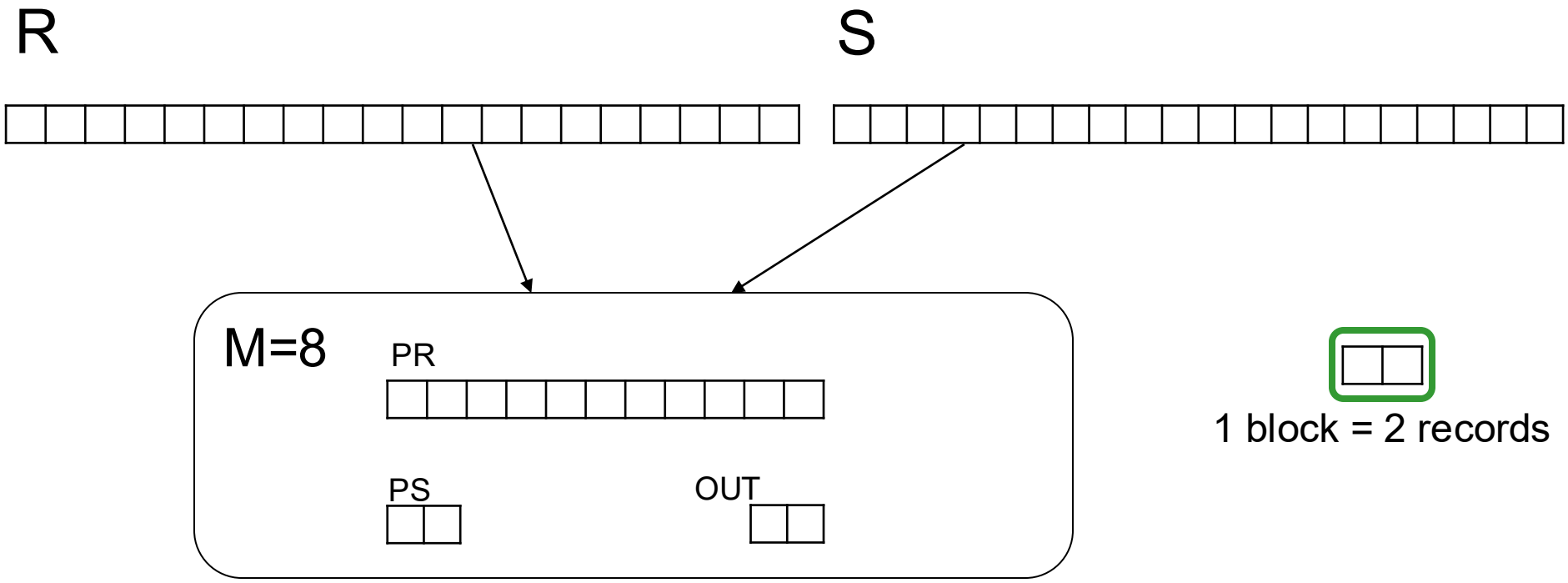
$R \bowtie S$

Block Nested Loop Join



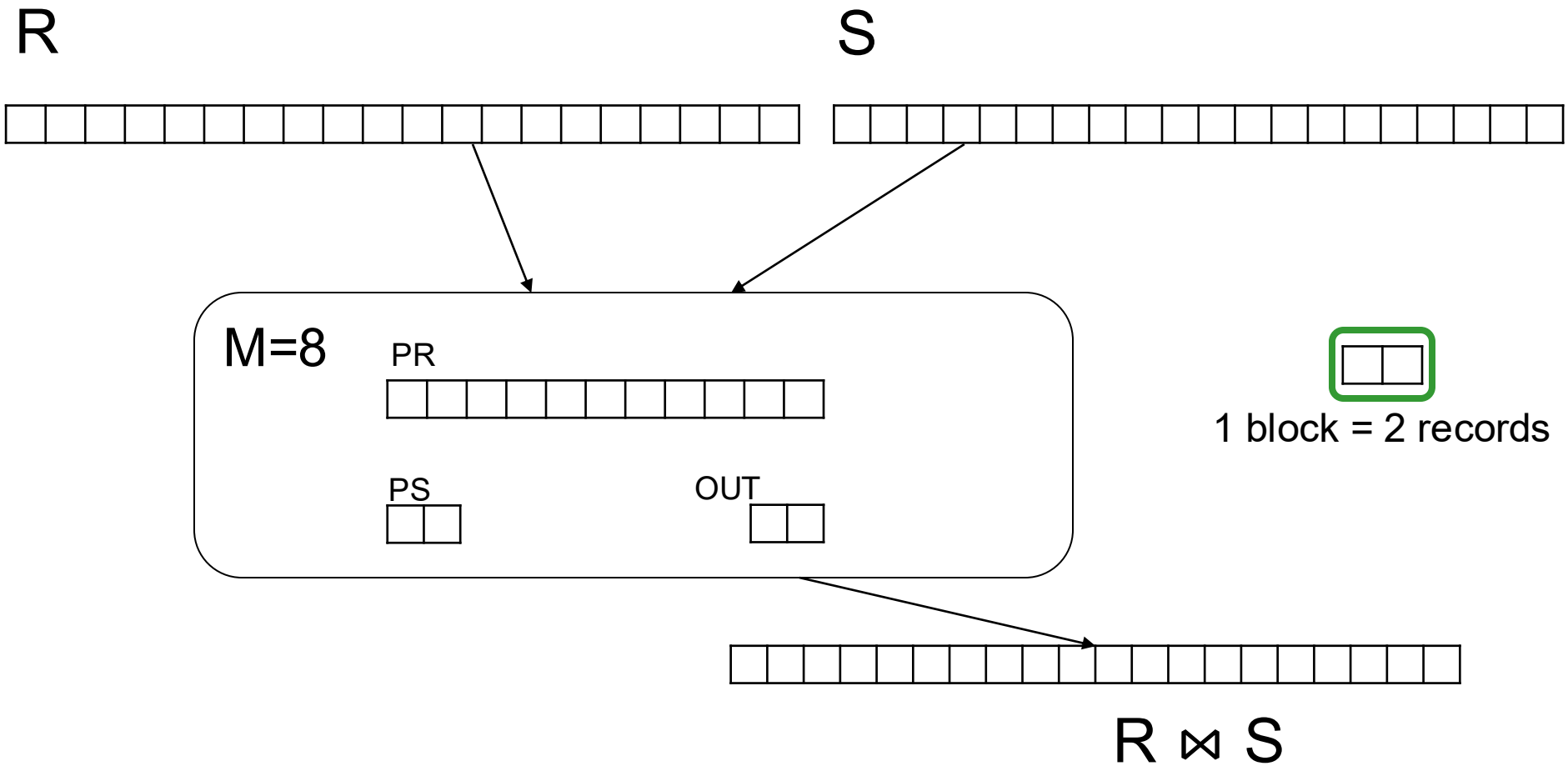
$R \bowtie S$

Block Nested Loop Join

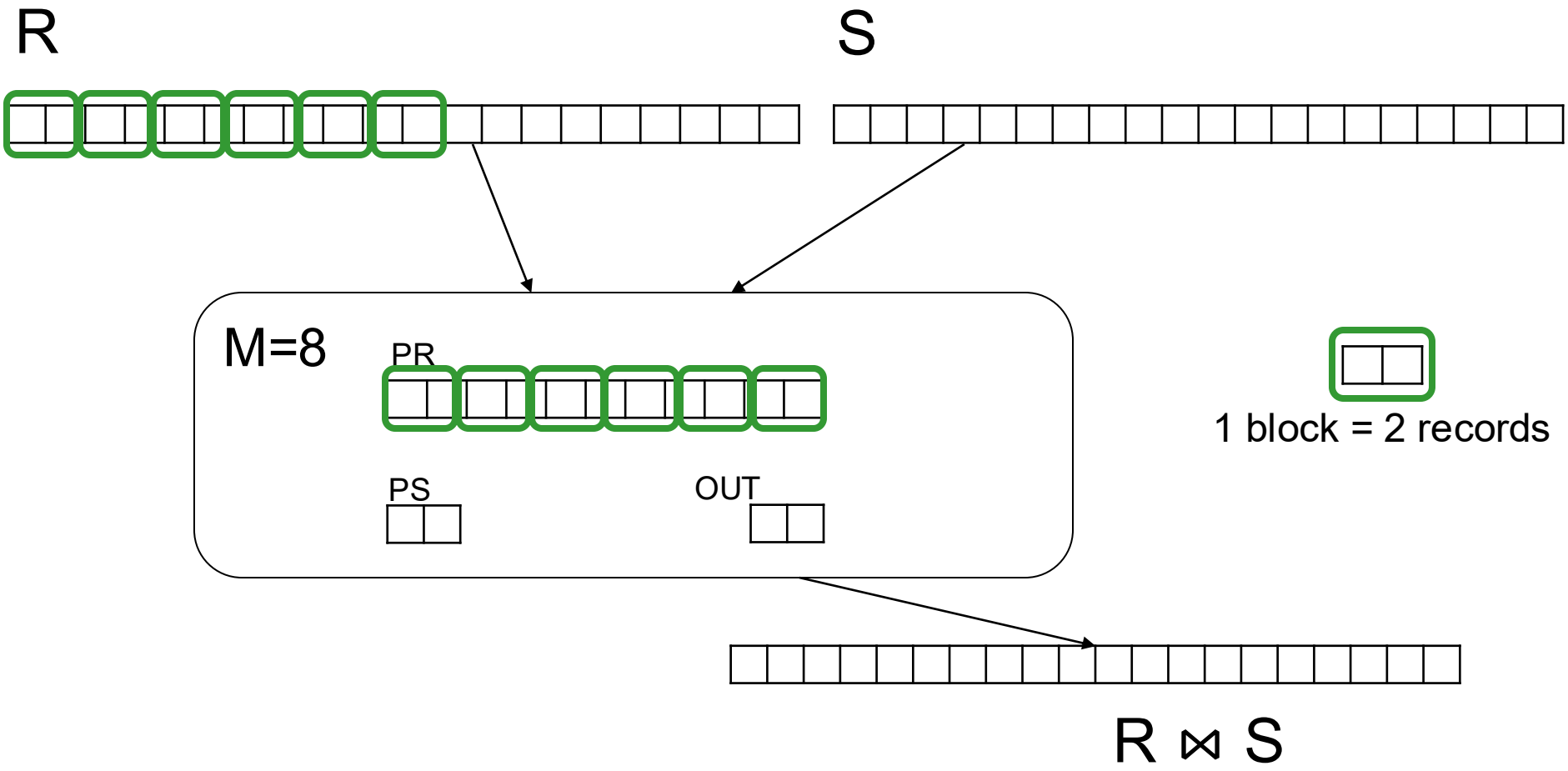


$R \bowtie S$

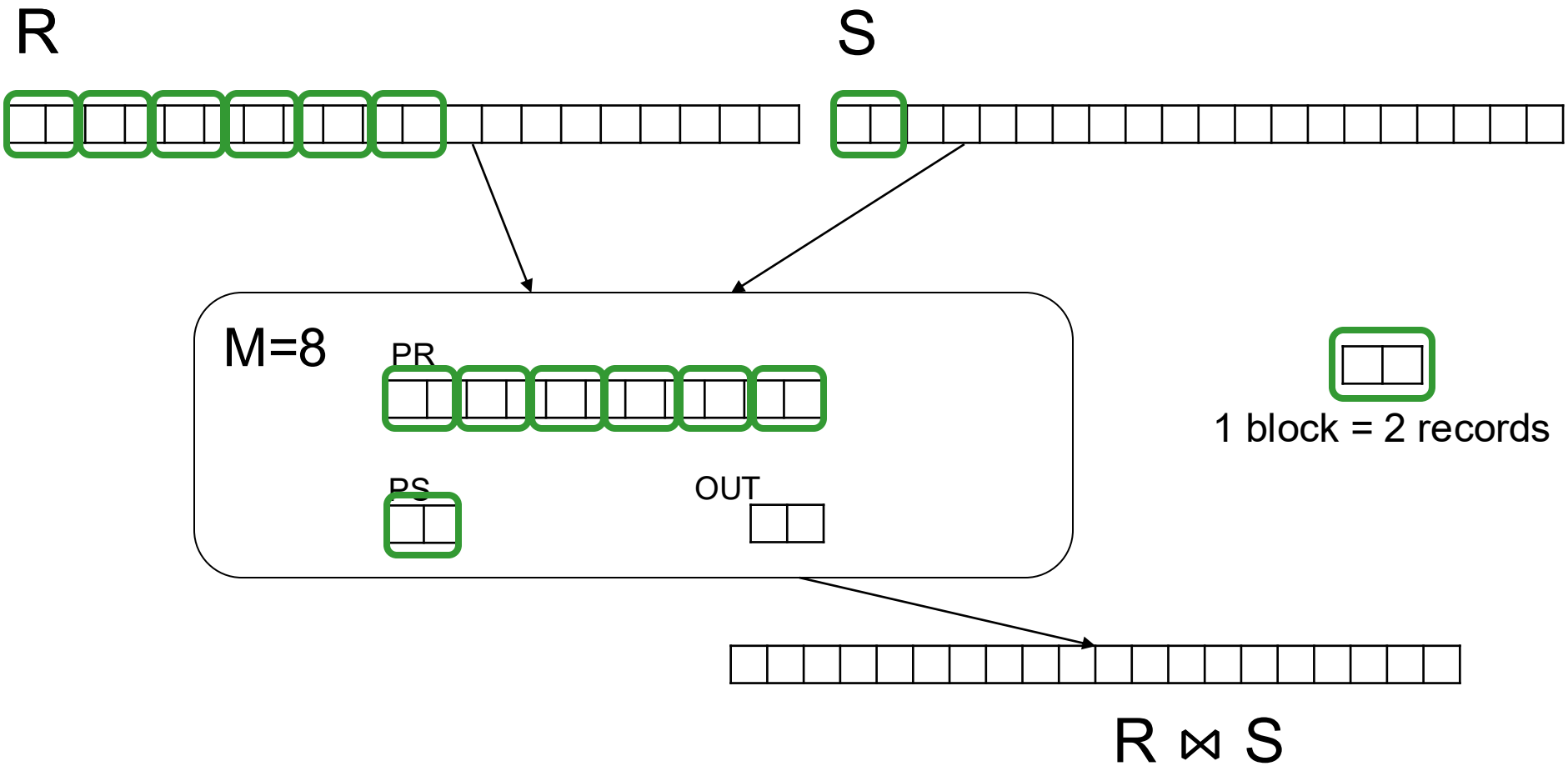
Block Nested Loop Join



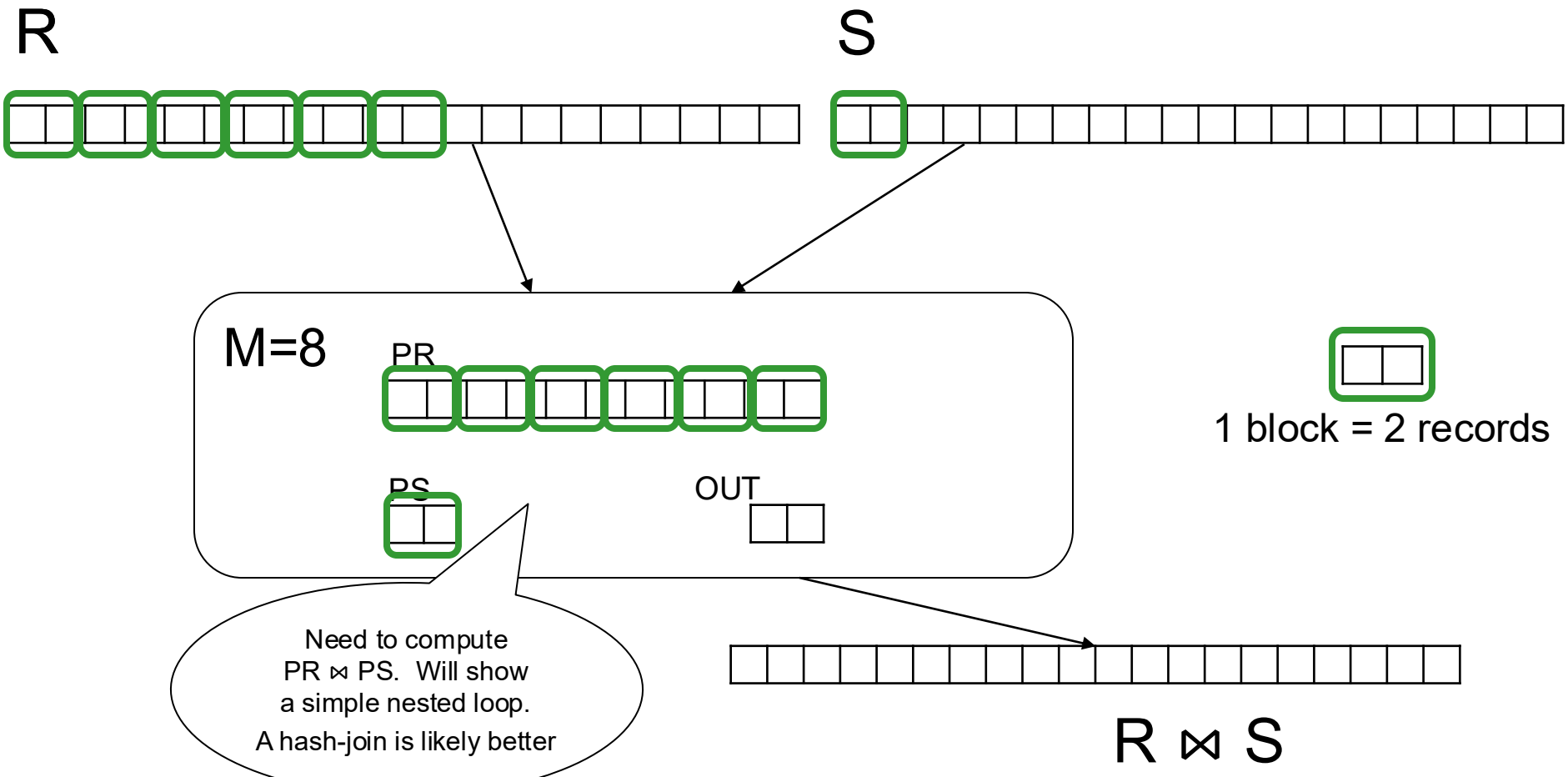
Block Nested Loop Join



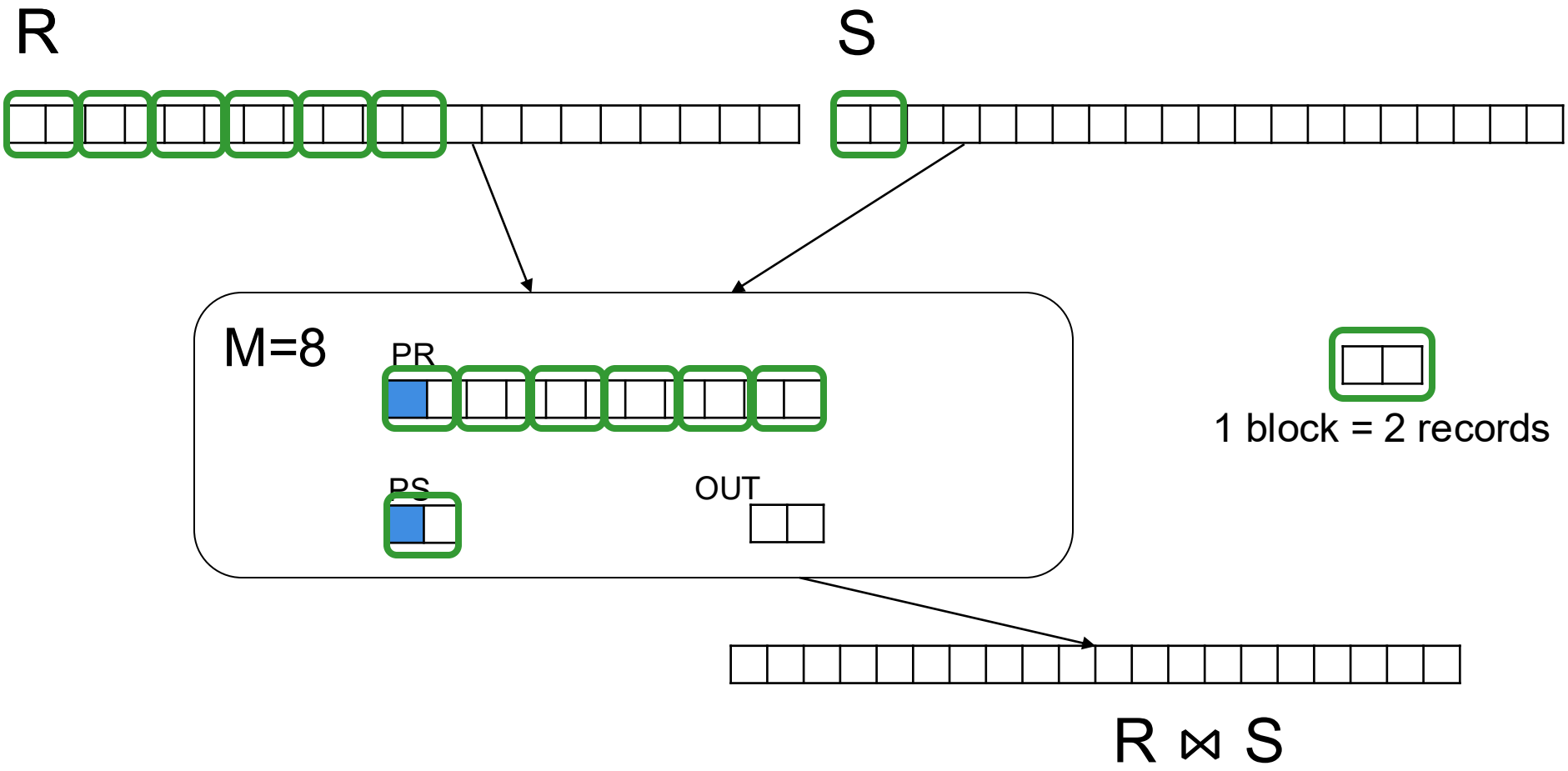
Block Nested Loop Join



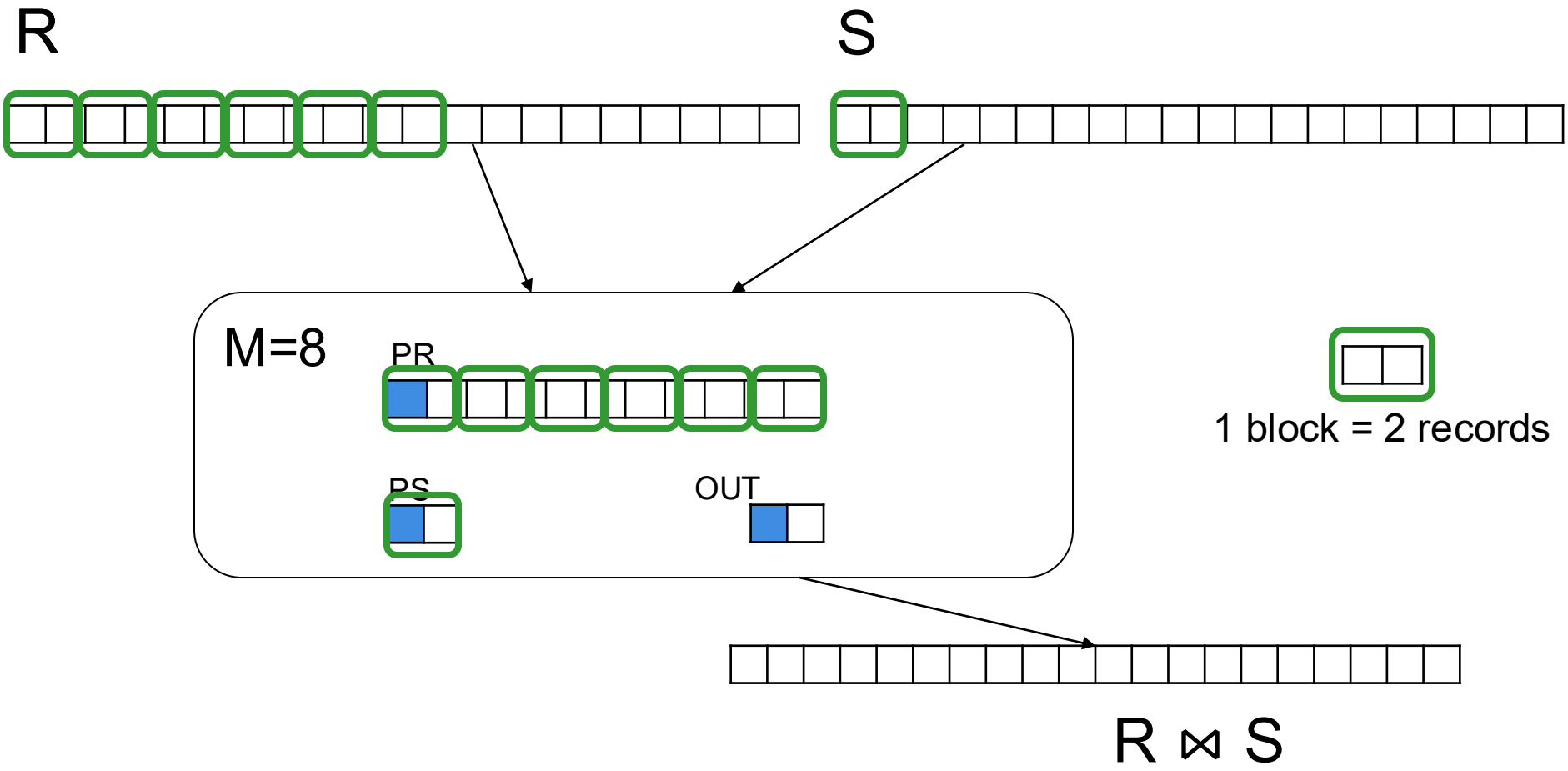
Block Nested Loop Join



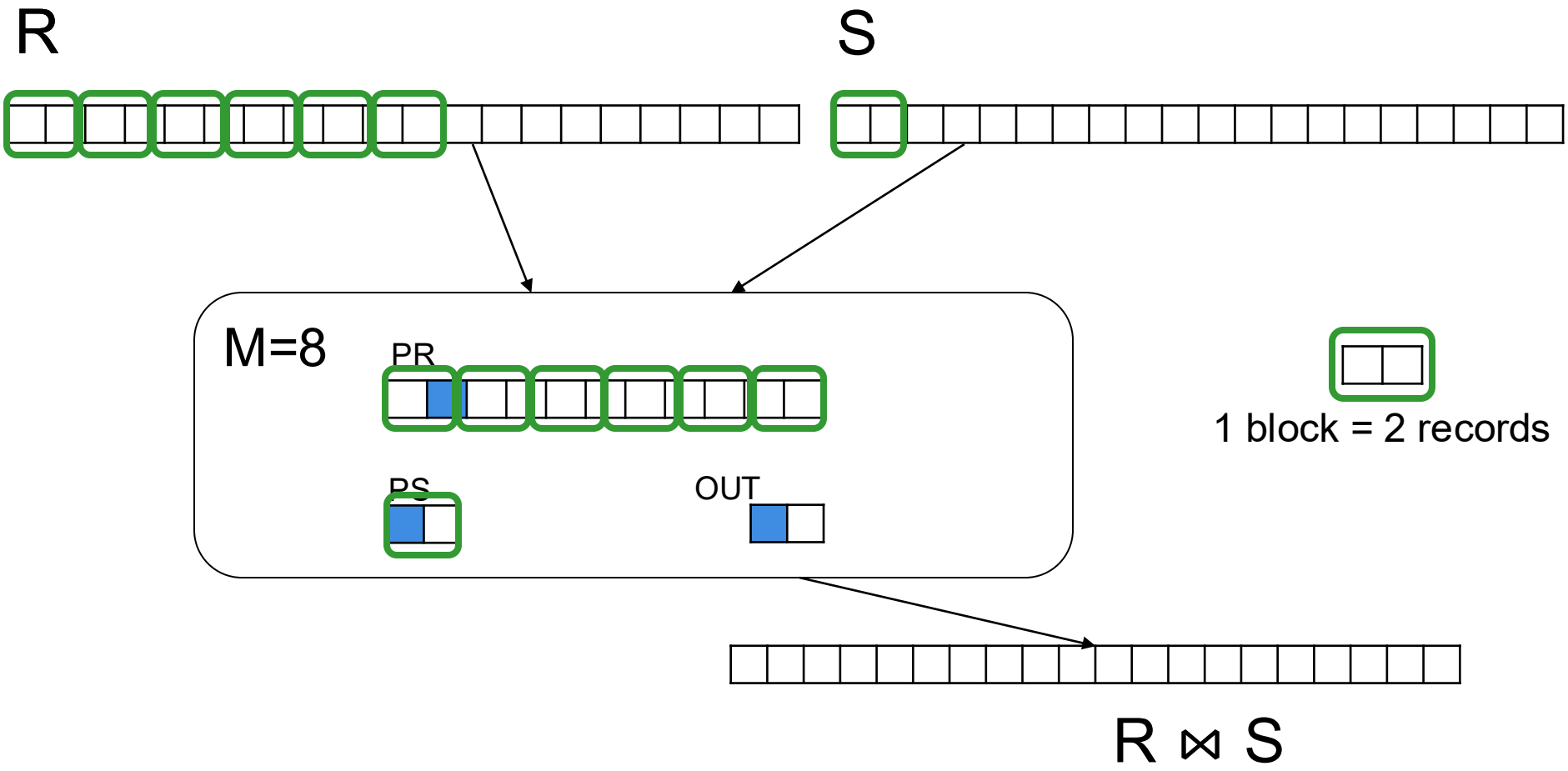
Block Nested Loop Join



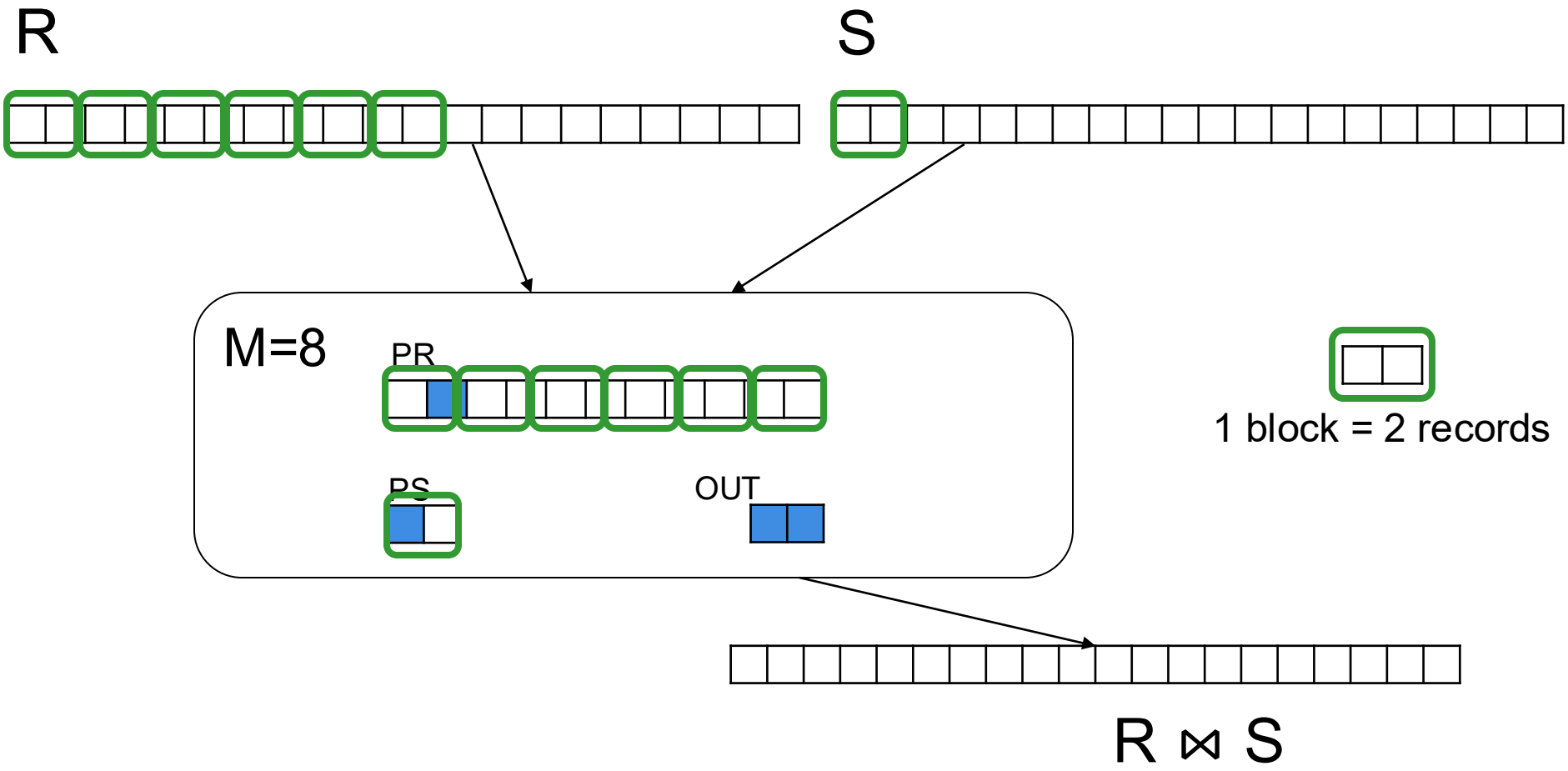
Block Nested Loop Join



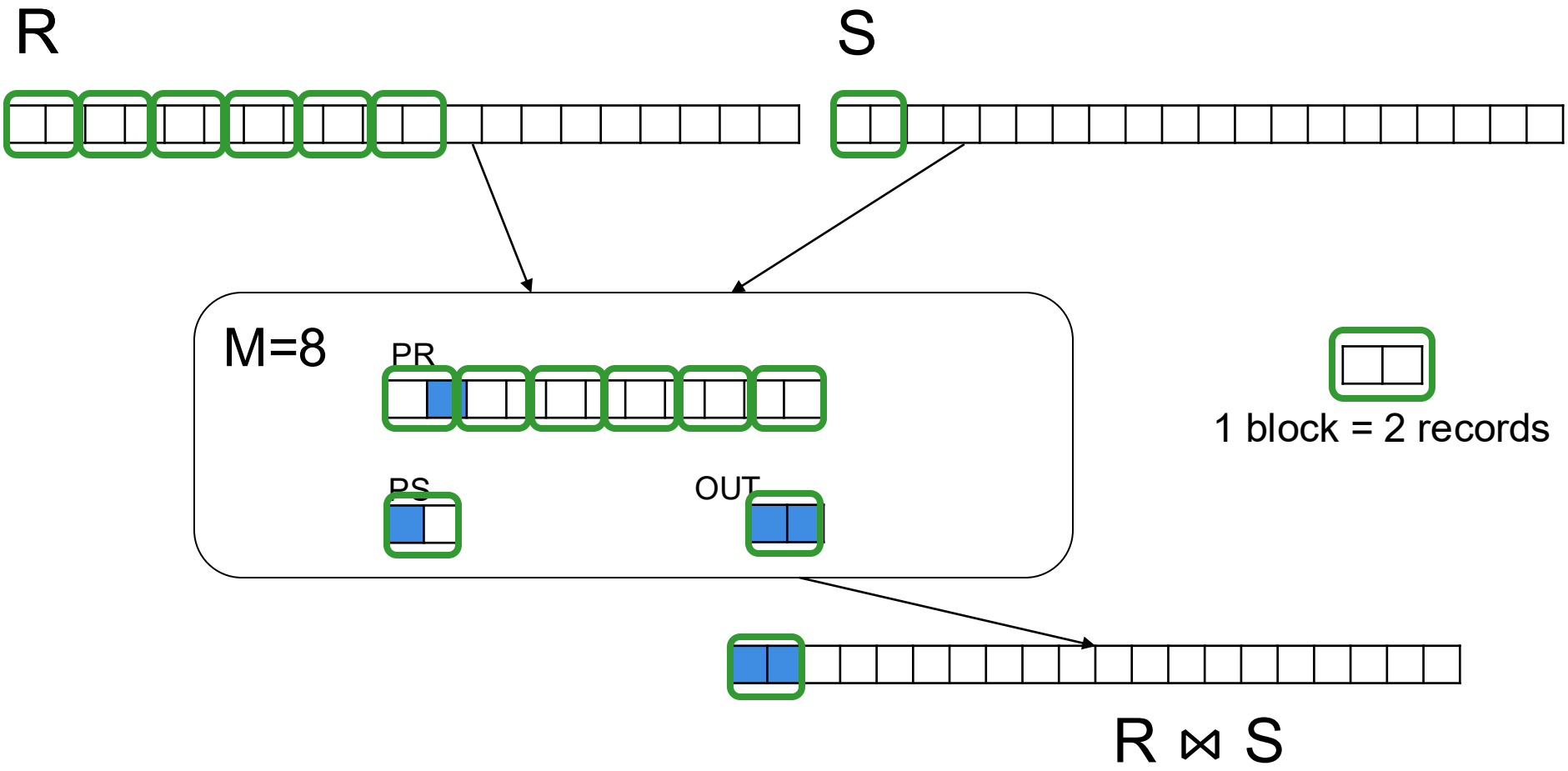
Block Nested Loop Join



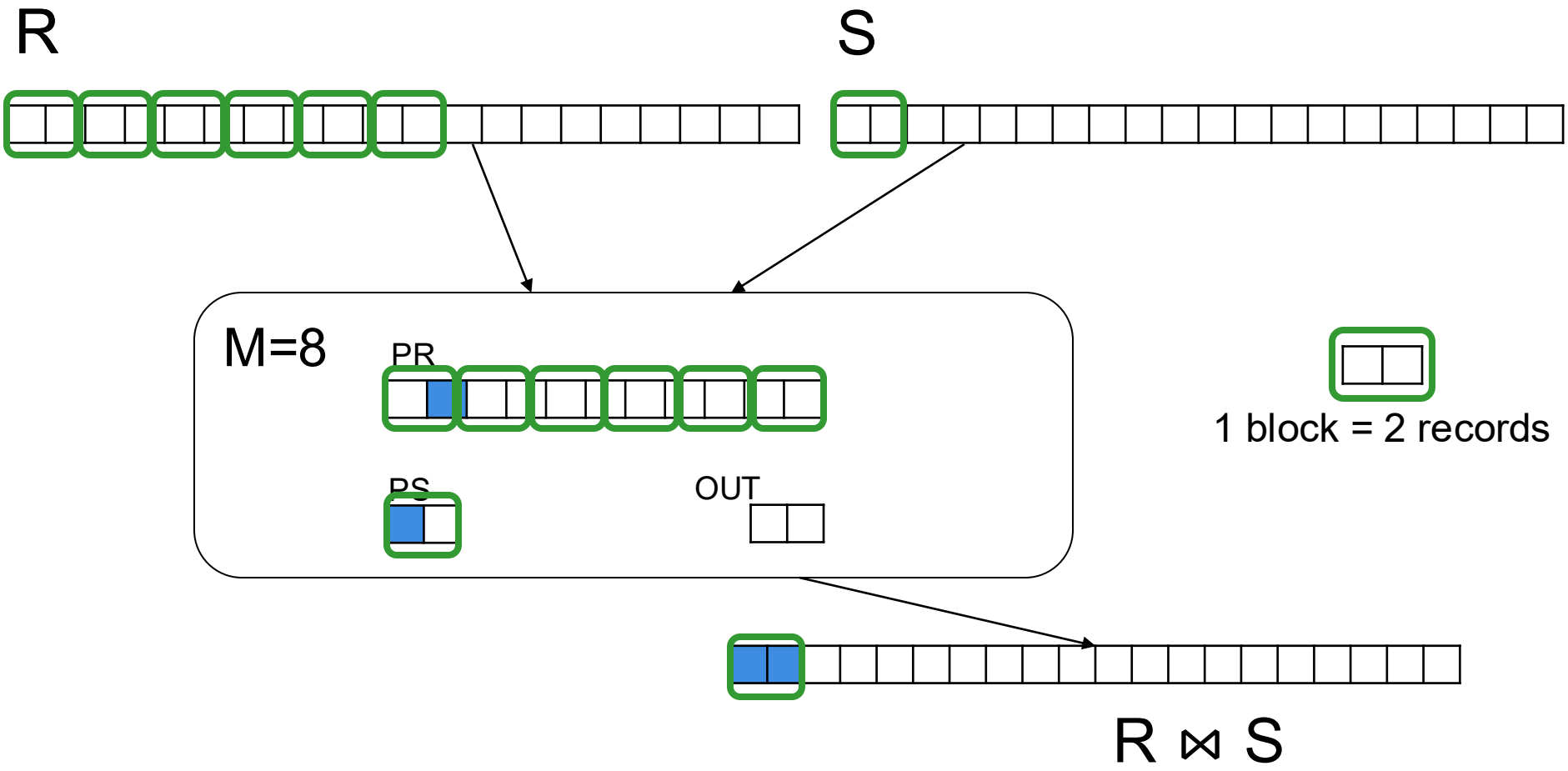
Block Nested Loop Join



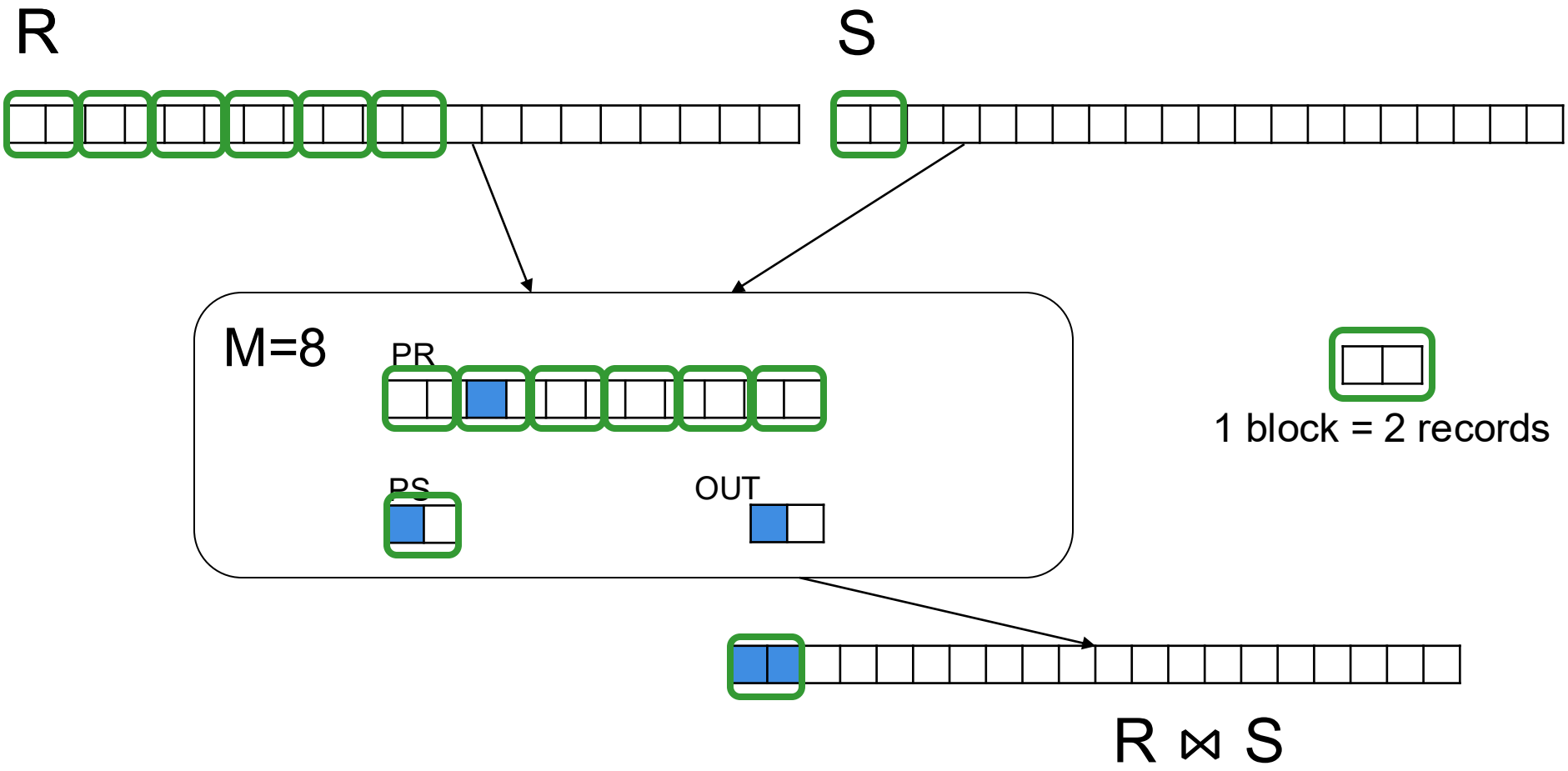
Block Nested Loop Join



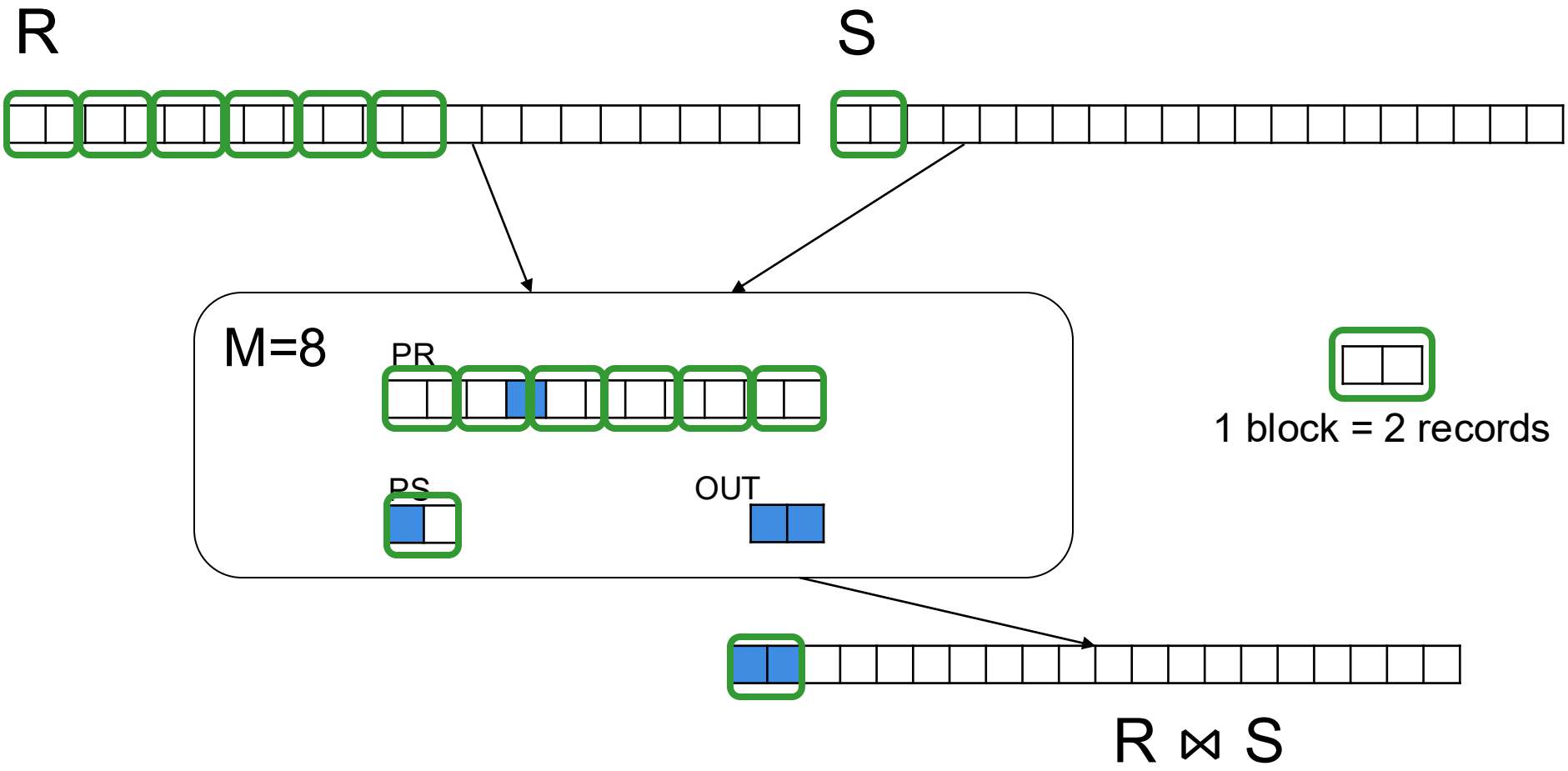
Block Nested Loop Join



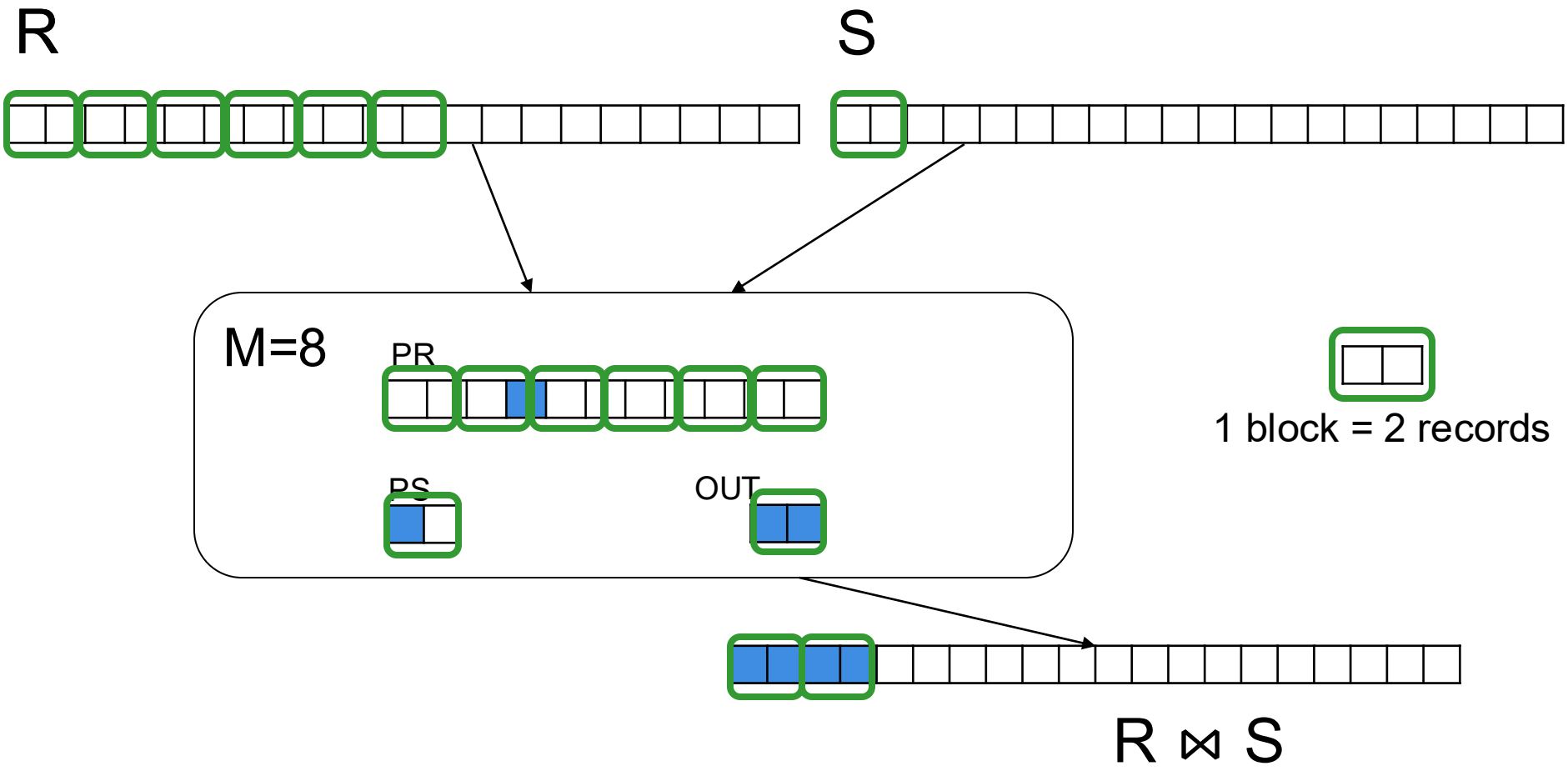
Block Nested Loop Join



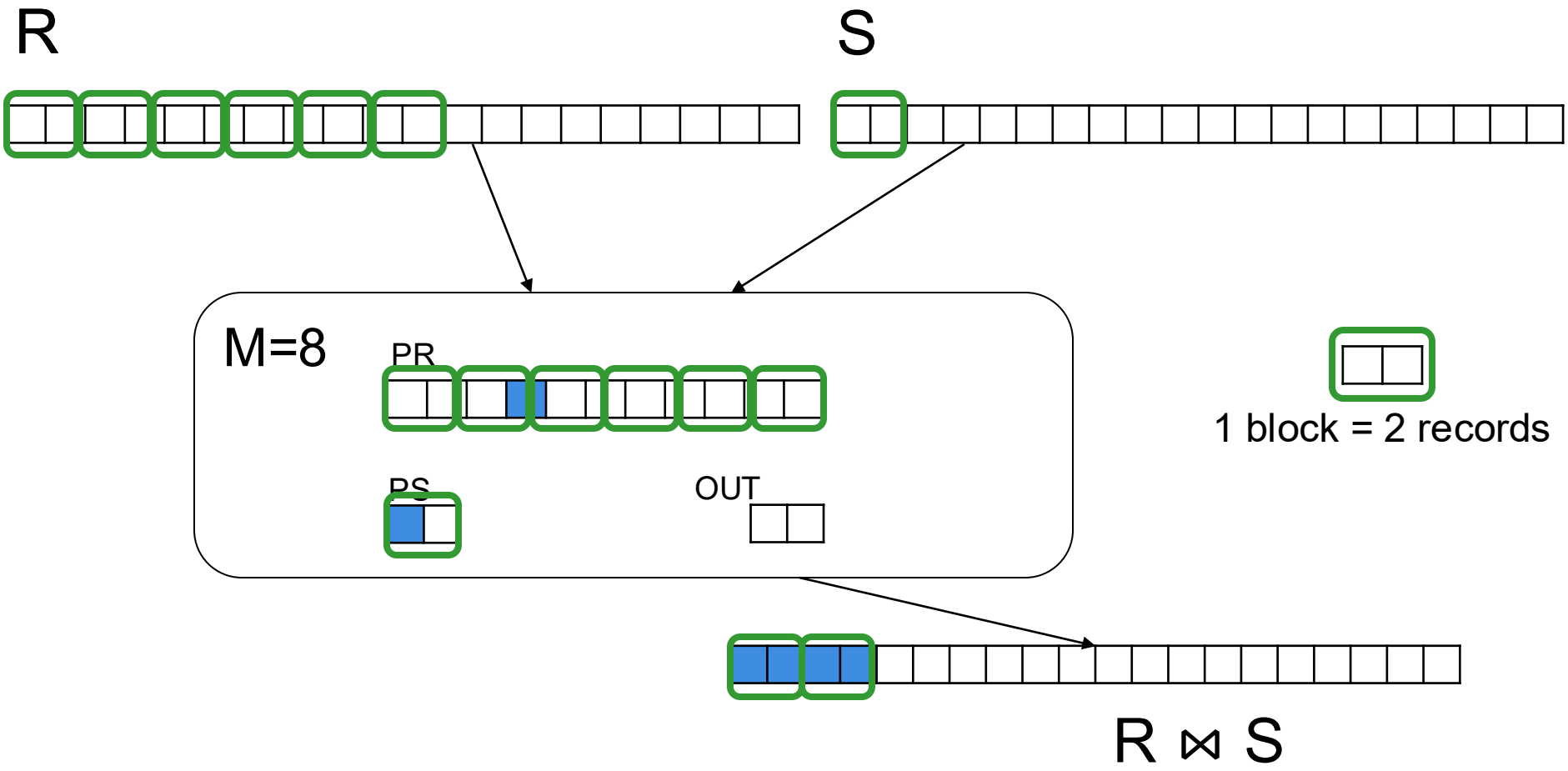
Block Nested Loop Join



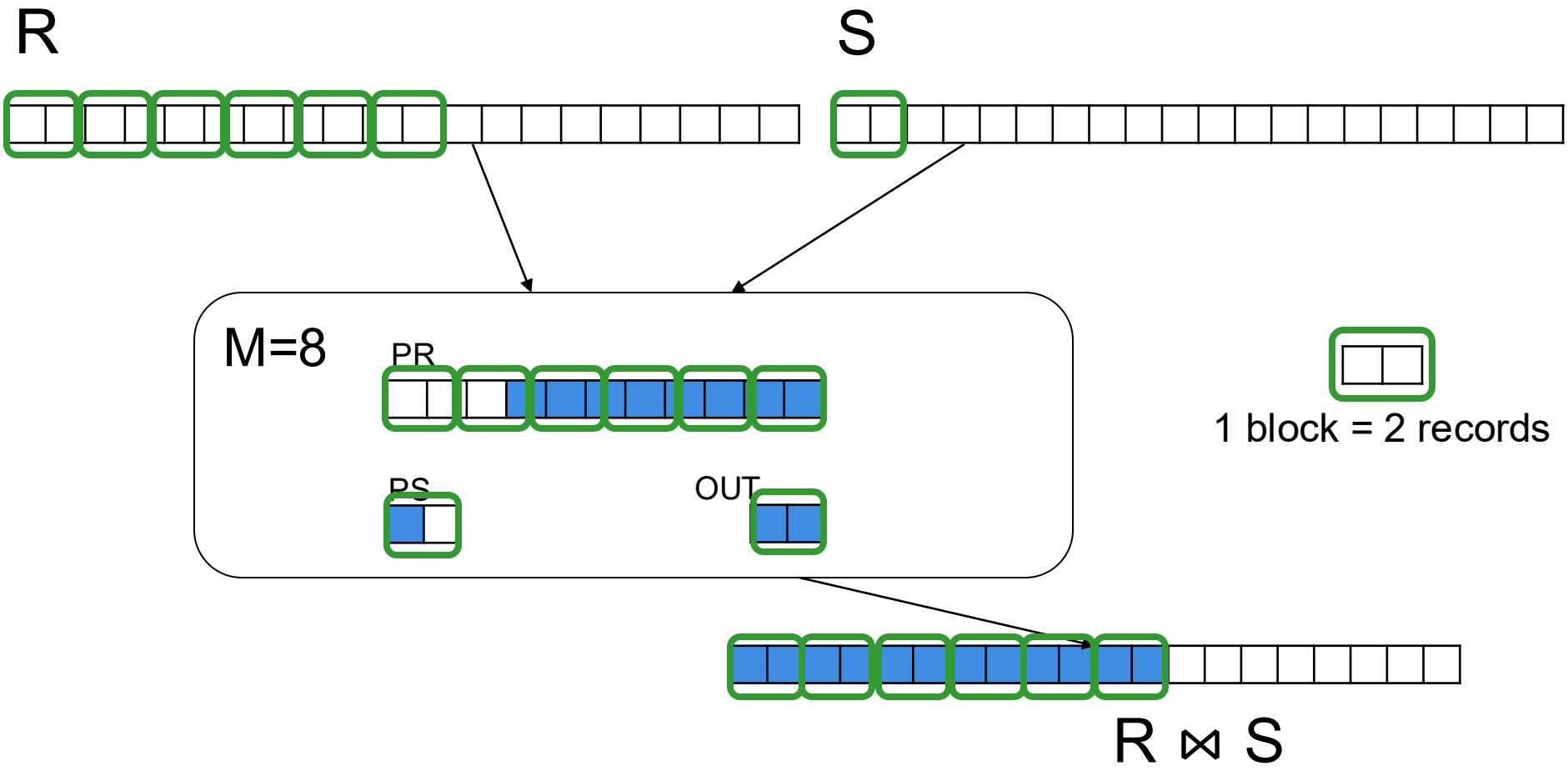
Block Nested Loop Join



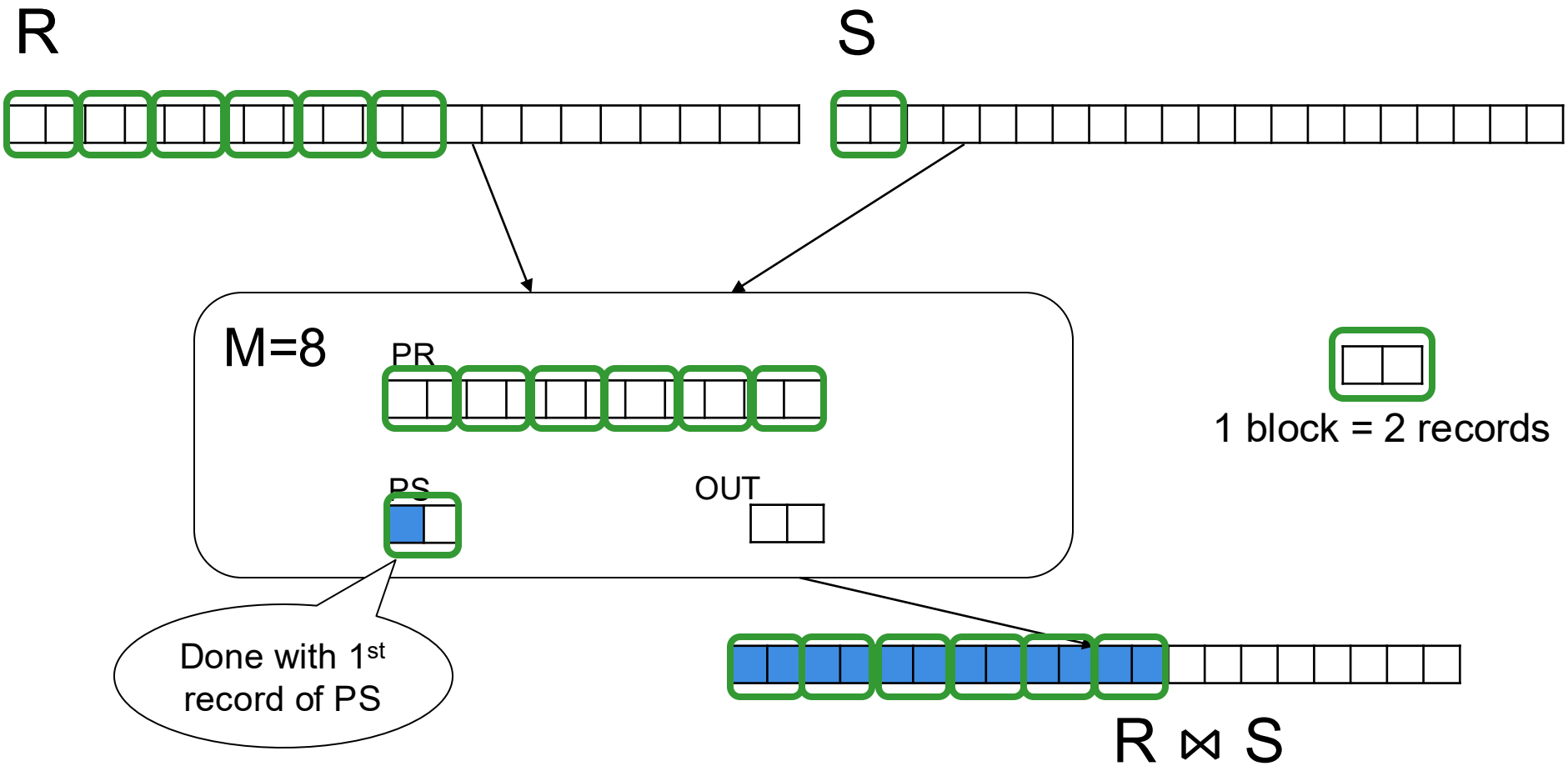
Block Nested Loop Join



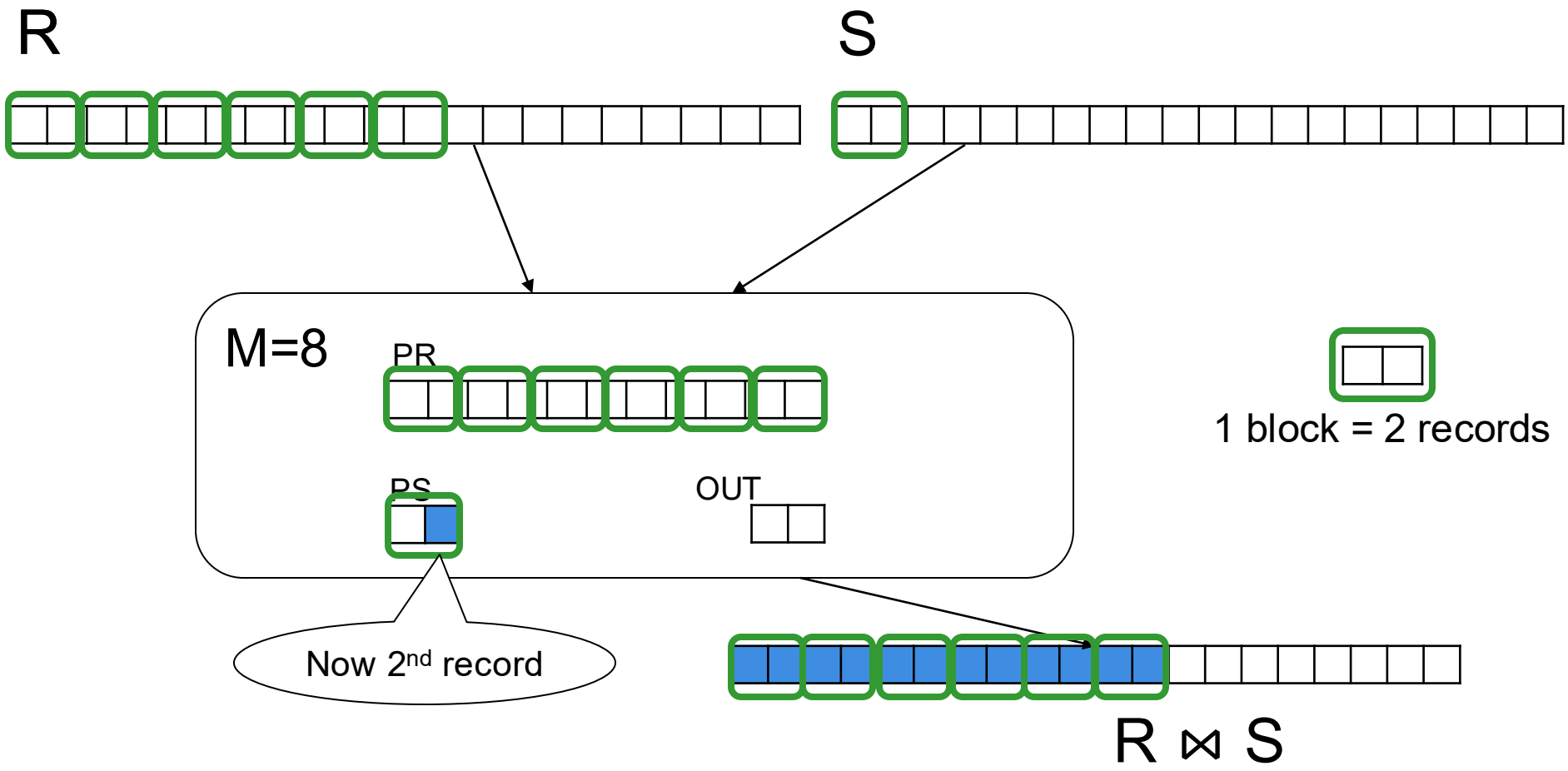
Block Nested Loop Join



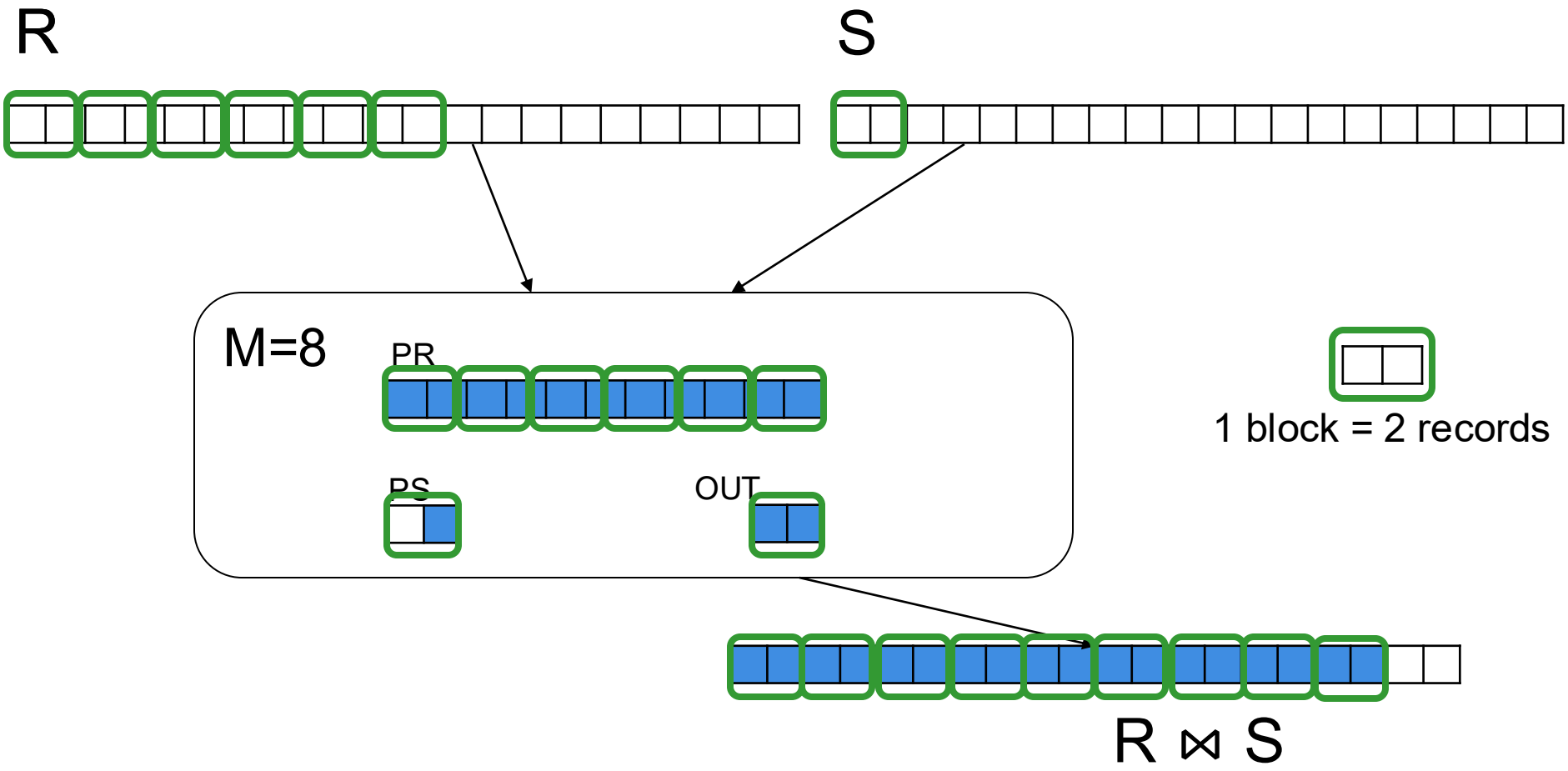
Block Nested Loop Join



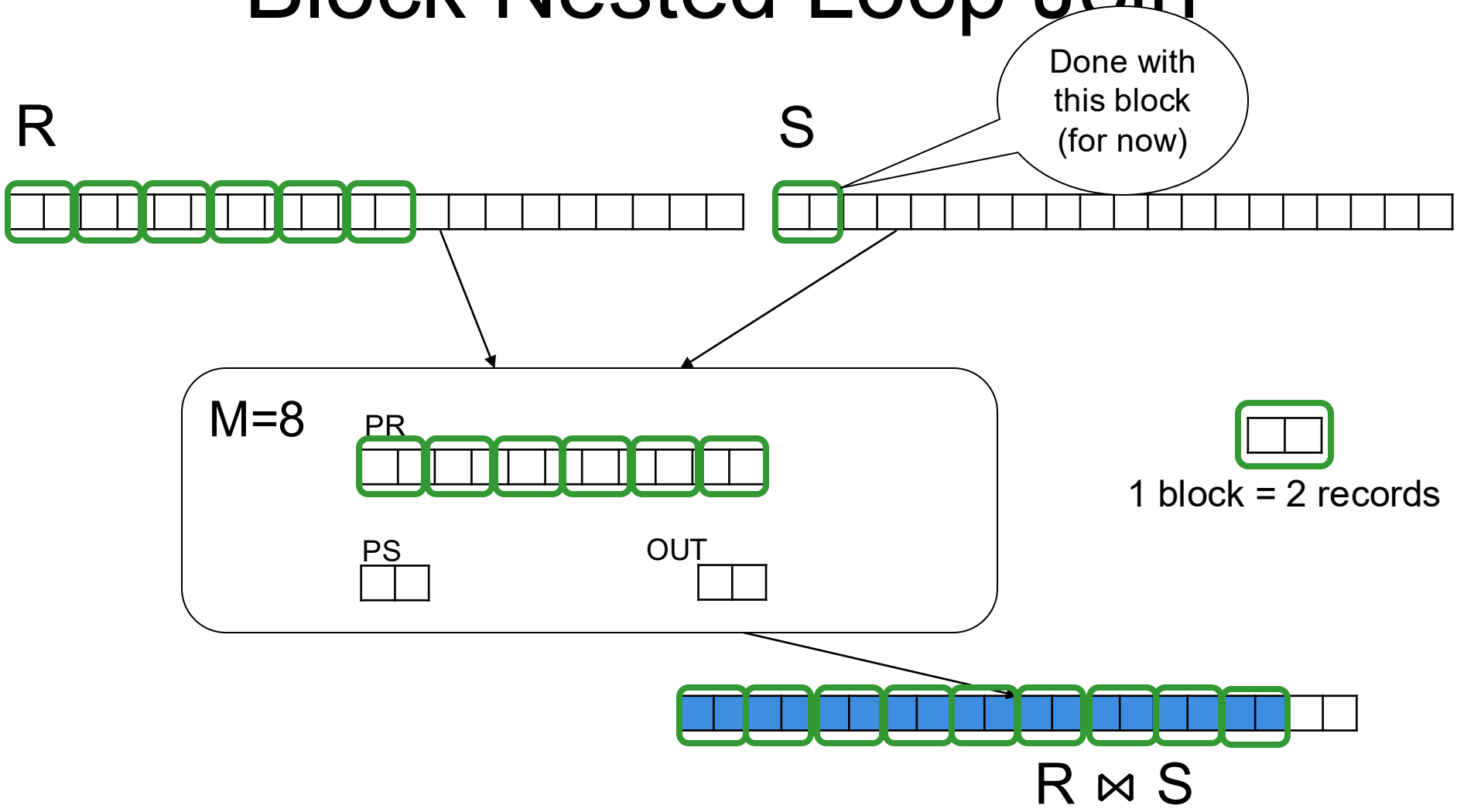
Block Nested Loop Join



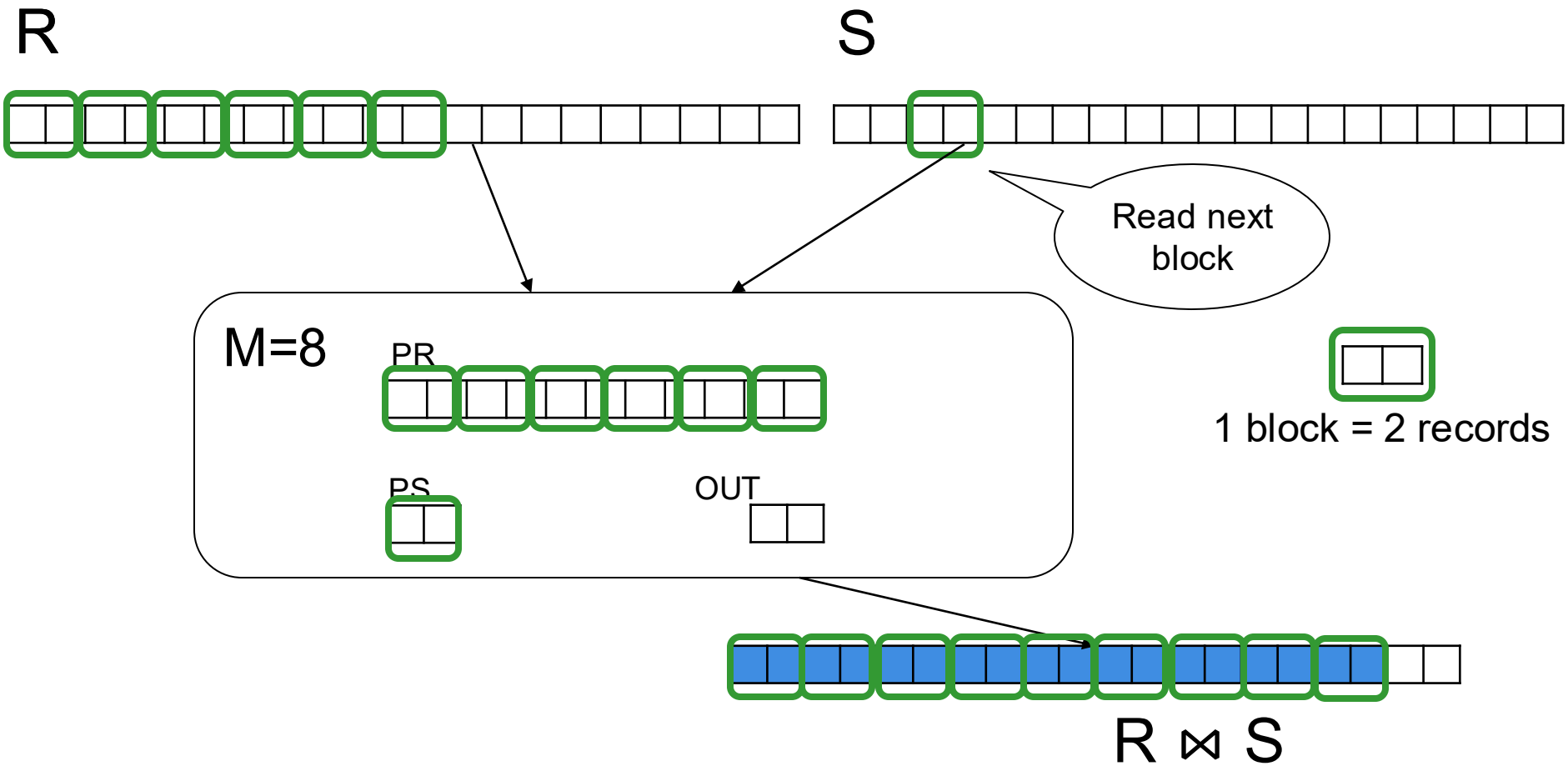
Block Nested Loop Join



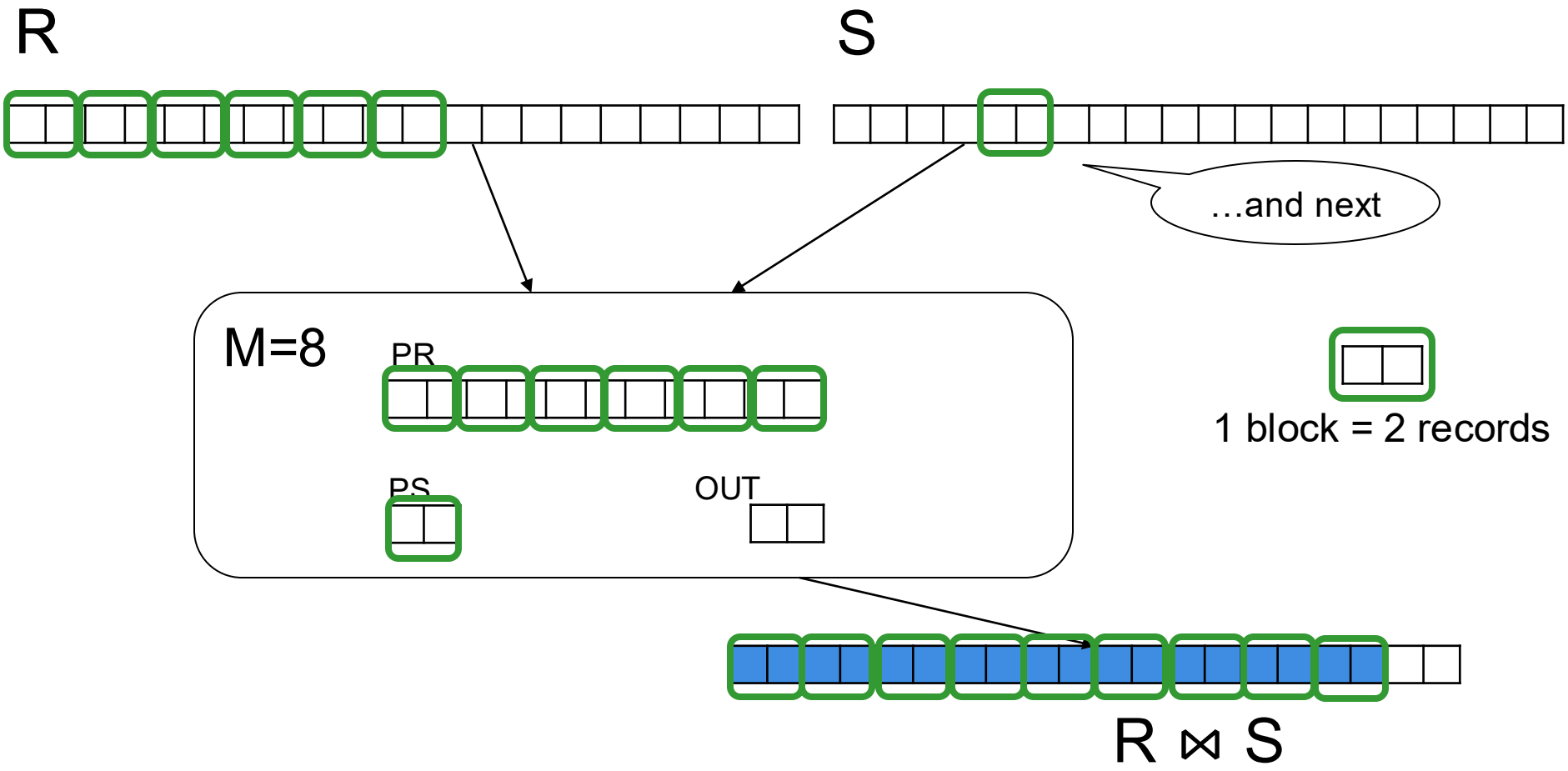
Block Nested Loop Join



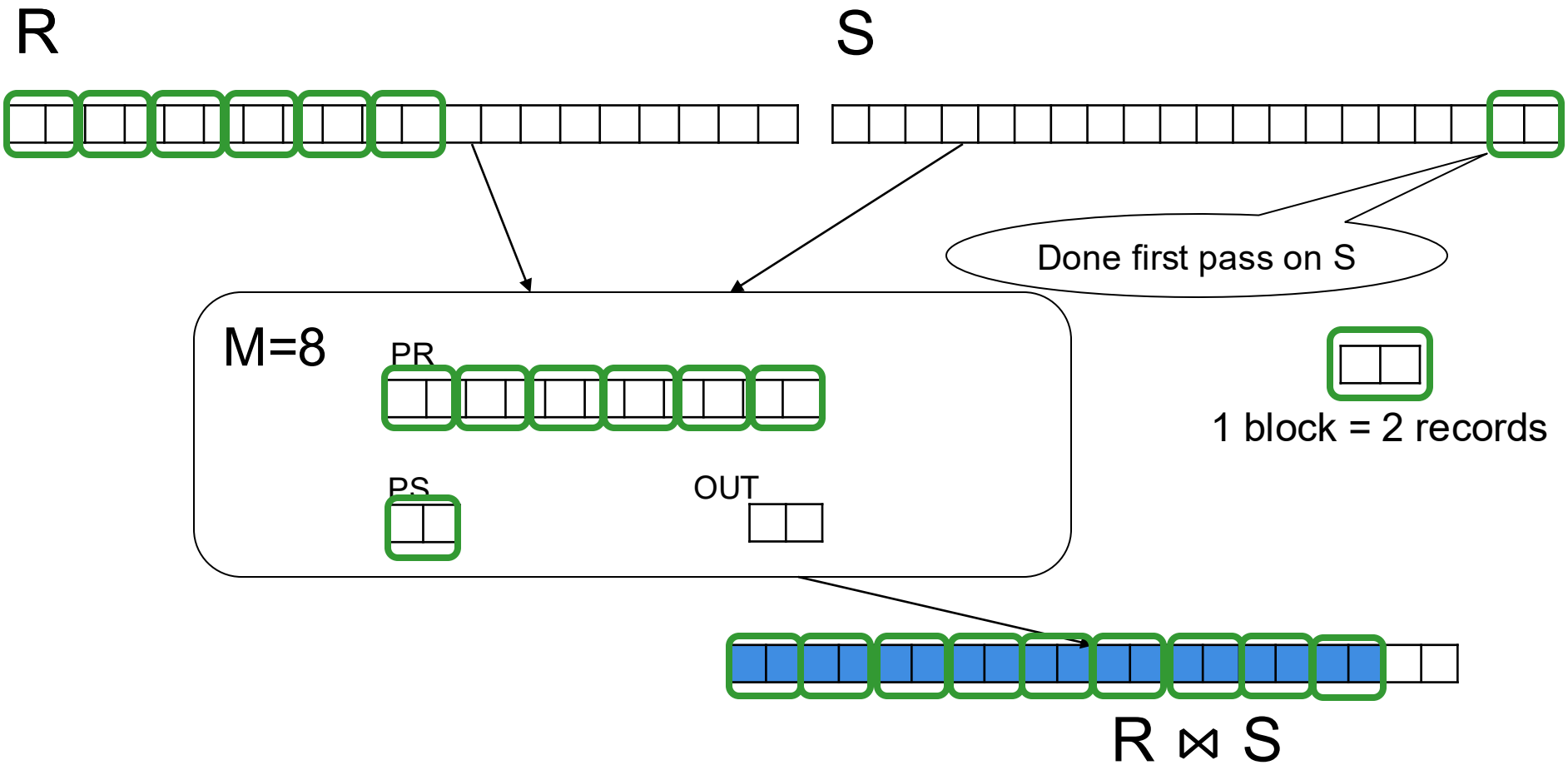
Block Nested Loop Join



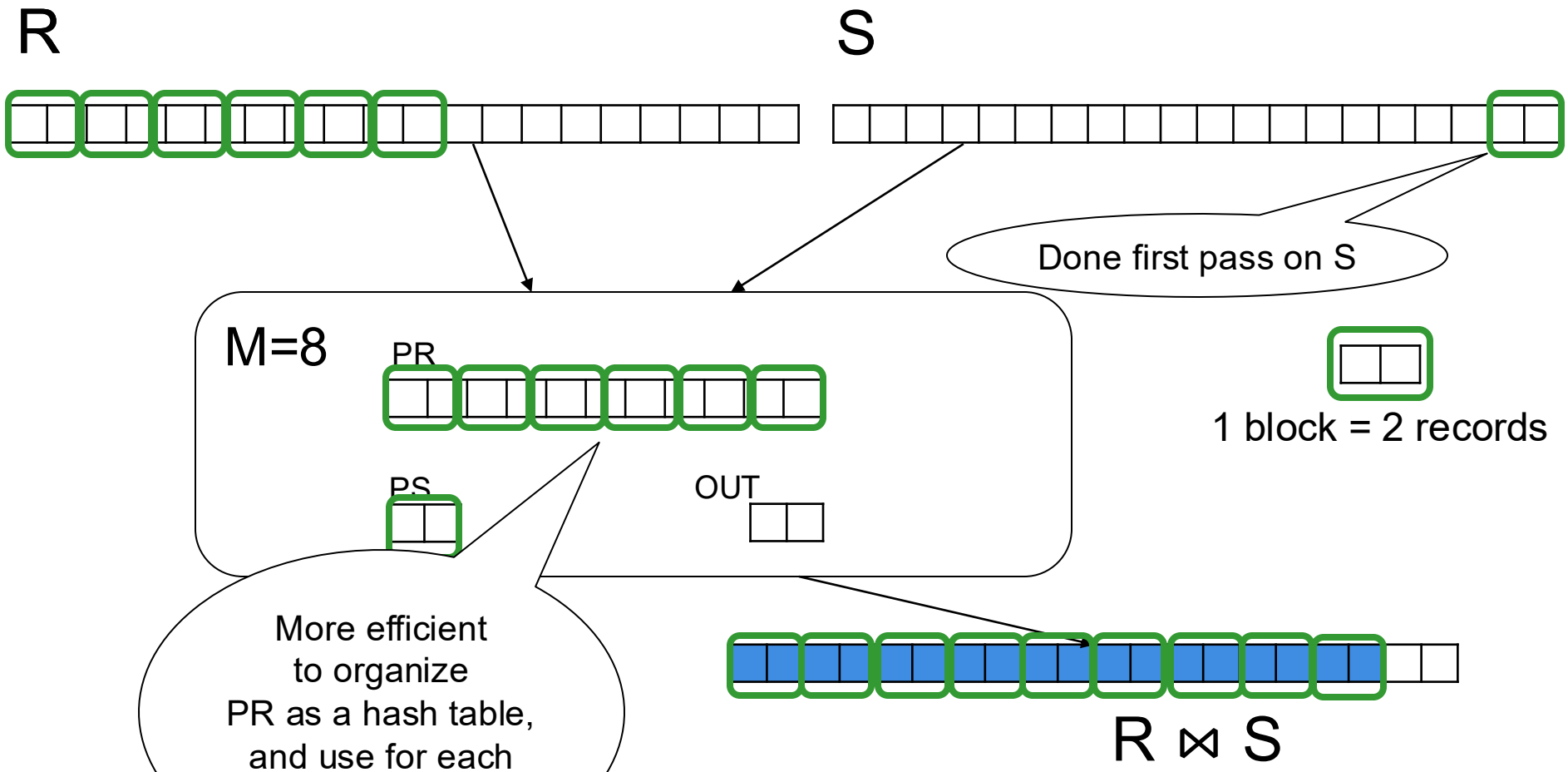
Block Nested Loop Join



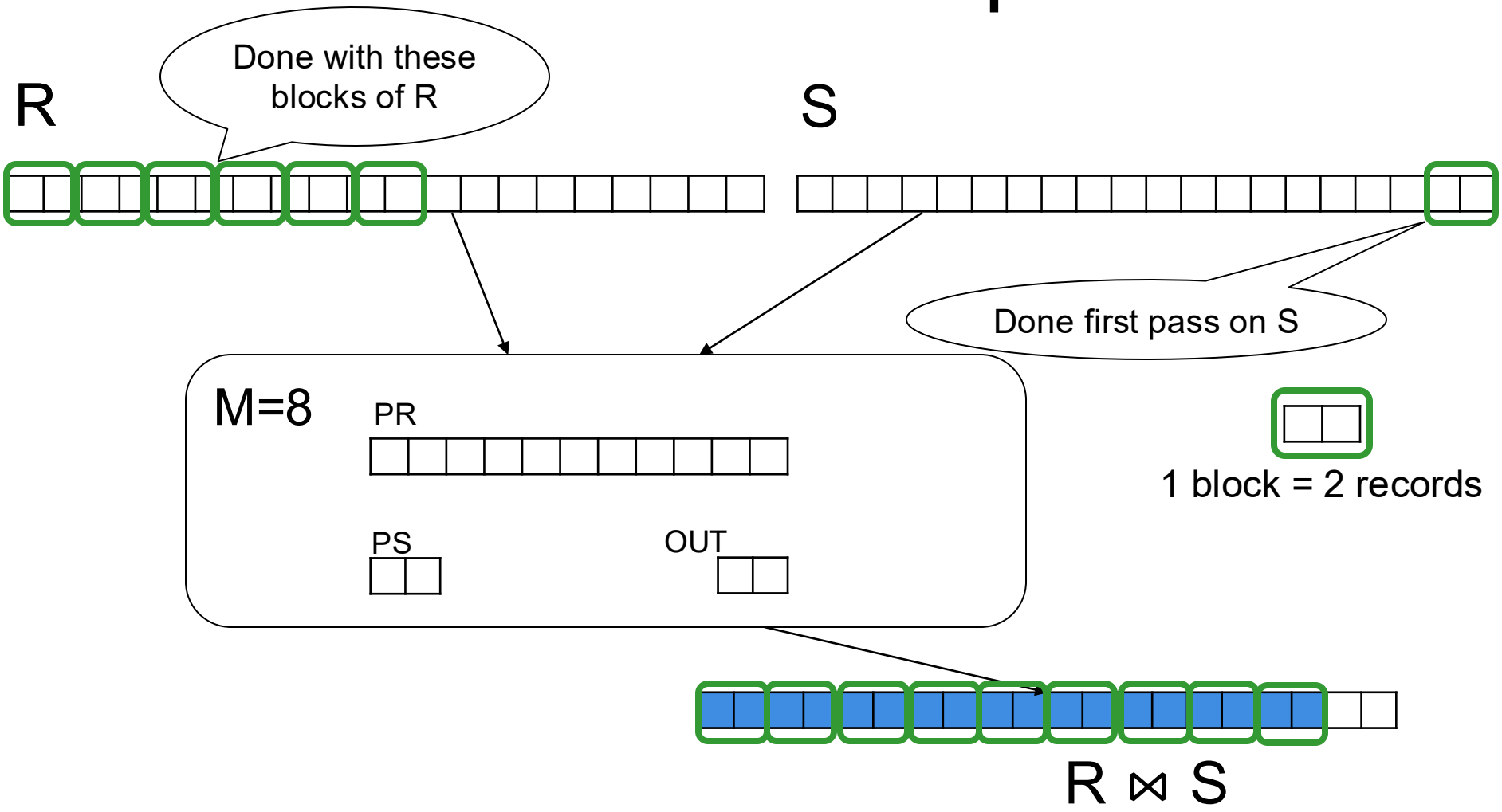
Block Nested Loop Join



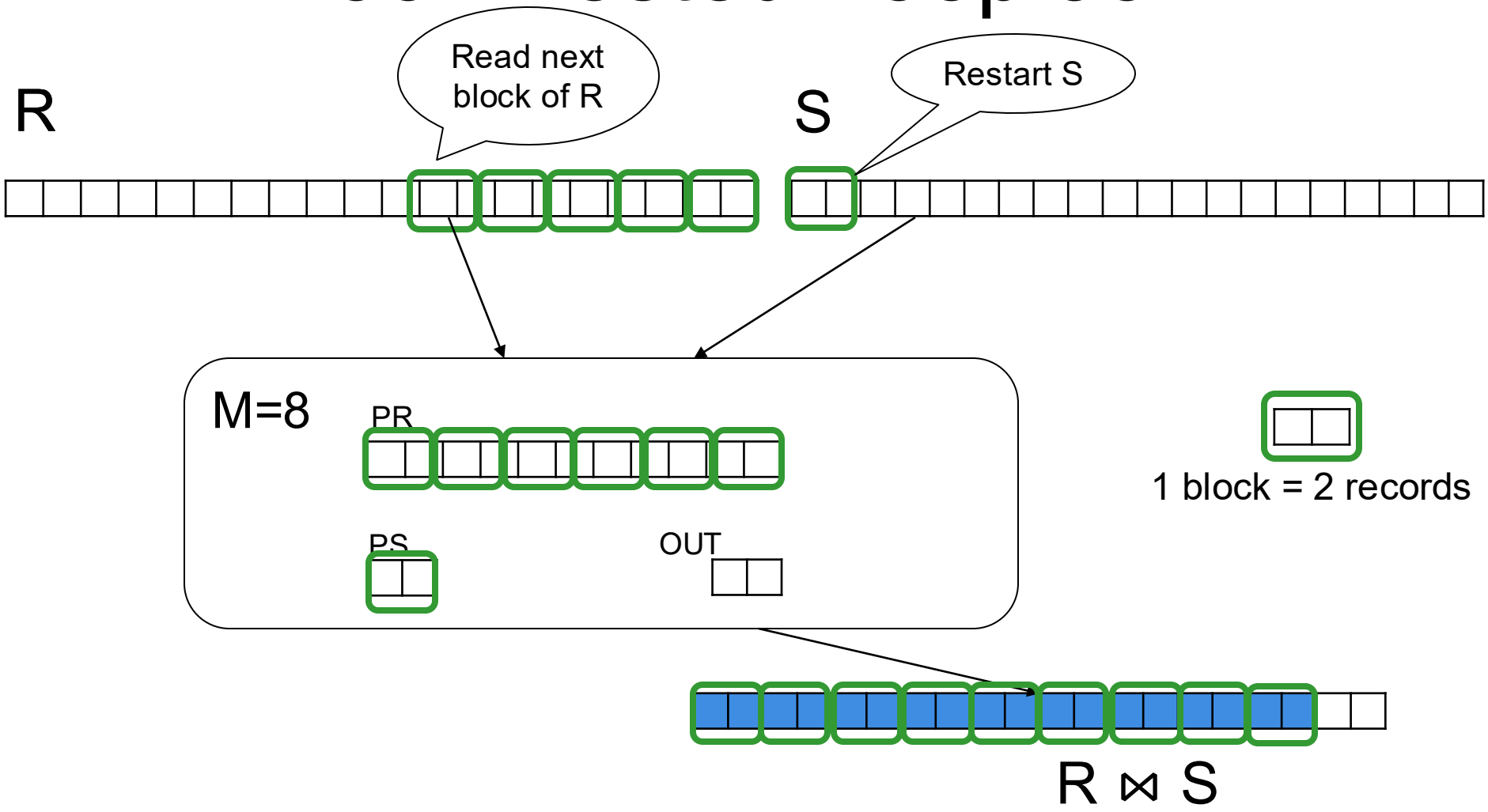
Block Nested Loop Join



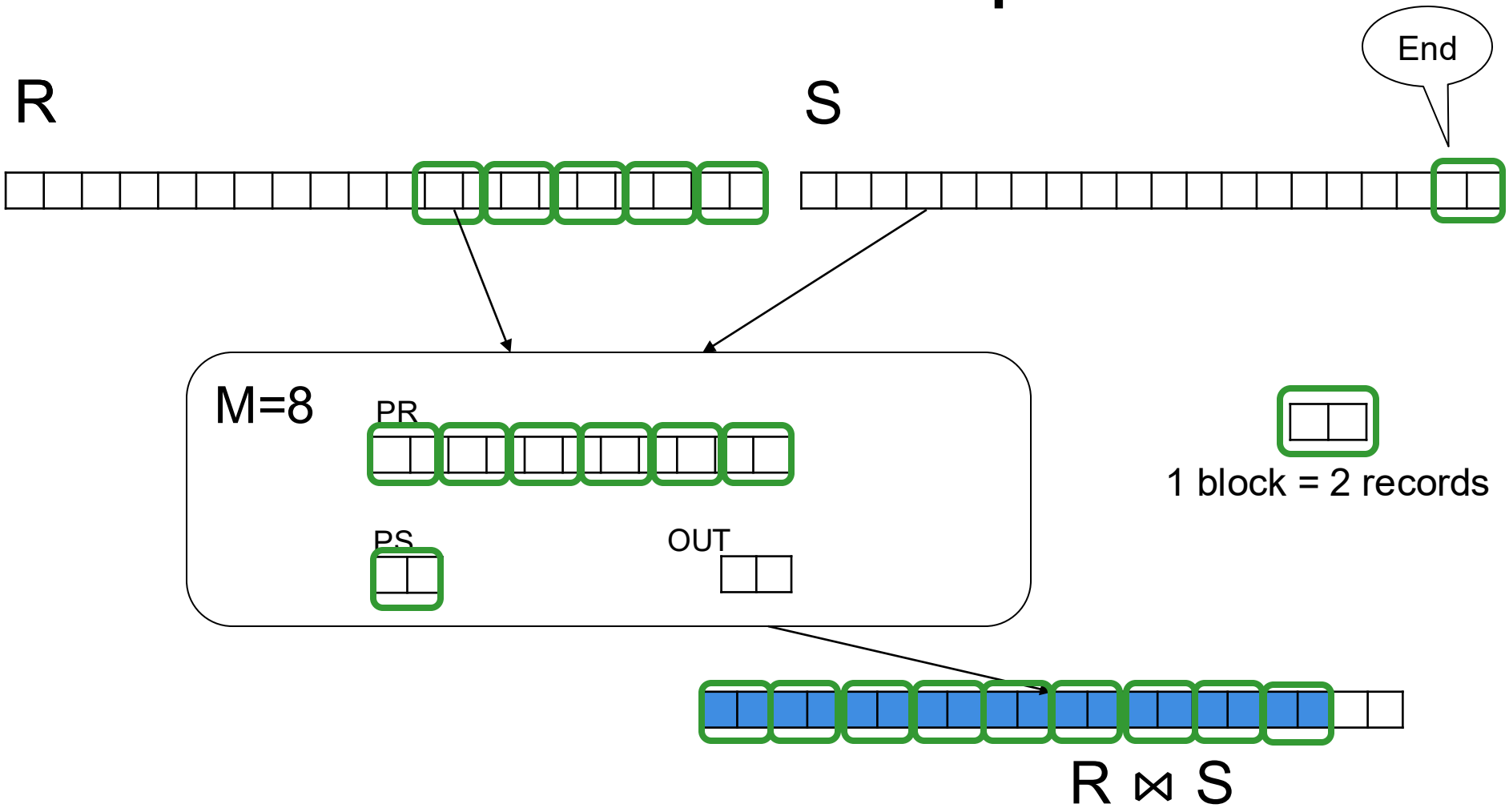
Block Nested Loop Join



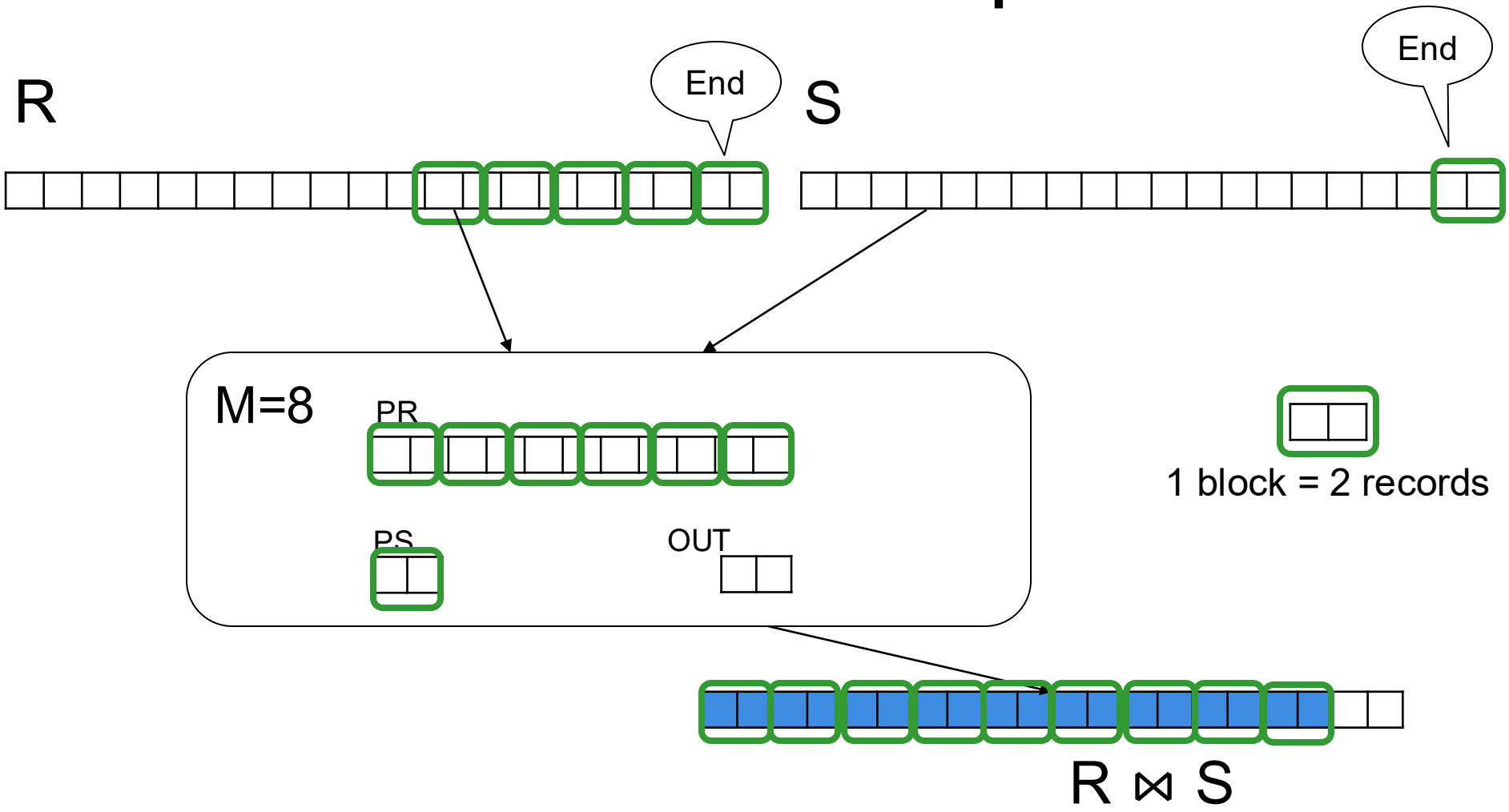
Block Nested Loop Join



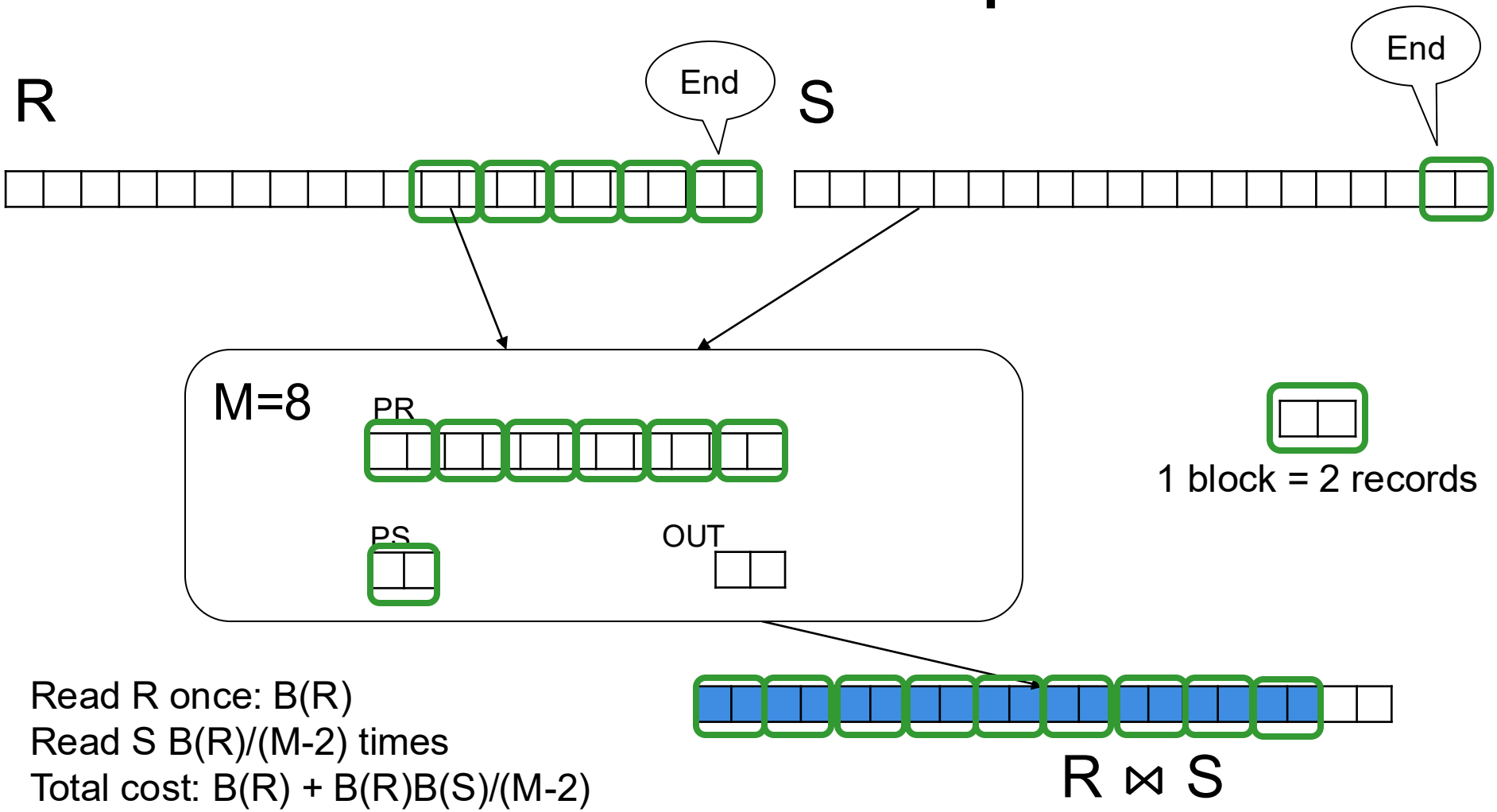
Block Nested Loop Join



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 cost of writing the output

Choice of Inner/Outer Table

- Recall terminology:

  Outer-table ⋈ Inner-table

- Cost of block nested loop join

  B(R) + B(R)B(S)/(M-2)

- Q: which table should be inner/outer?

Choice of Inner/Outer Table

- Recall terminology:

  Outer-table ⋈ Inner-table

- Cost of block nested loop join

  B(R) + B(R)B(S)/(M-2)

- Q: which table should be inner/outer?
- A: smallest table = outer, largest=inner

Final Discussion

- Block nested loop join can be very effective when the outer table is only slightly larger than M
- Usually combined with a hash-based main-memory join

External Memory
Merge Sort, Merge Join

Merge-Sort of a Relation R

Merge-sort reads/writes sequentially

- Run lengths: 2, 4, 8, 16, …

Merge-Sort of a Relation R

Merge-sort reads/writes sequentially

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

Cost = 2 log(N) B(R)

Multi-Way Merge-Sort

N-Way Merge Sort: use entire memory M

Multi-Way Merge-Sort

N-Way Merge Sort: use entire memory M

- Can read from M-1 runs
- Merge M-1 runs at once

Multi-Way Merge-Sort

N-Way Merge Sort: use entire memory M

- Can read from M-1 runs
- Merge M-1 runs at once
- The run lengths become:

  $(M-1), (M-1)^2, (M-2)^3, \ldots, B(R)$

Multi-Way Merge-Sort

N-Way Merge Sort: use entire memory M

- Can read from M-1 runs
- Merge M-1 runs at once
- The run lengths become:
  $(M-1), (M-1)^2, (M-2)^3, \ldots, B(R)$
- # Passes = $\log_{M-1} B(R)$

Multi-Way Merge-Sort

N-Way Merge Sort: use entire memory M

- Can read from M-1 runs
- Merge M-1 runs at once
- The run lengths become:
  $(M-1), (M-1)^2, (M-2)^3, \ldots, B(R)$
- # Passes = $\log_{M-1} B(R) \approx 2$

Assuming $B(R) \leq (M-1)^2$

Multi-Way Merge-Sort

N-Way Merge Sort: use entire memory M

- Can read from M-1 runs
- Merge M-1 runs at once
- The run lengths become:
  $(M-1), (M-1)^2, (M-2)^3, \ldots, B(R)$
- # Passes = $\log_{M-1} B(R) \approx 2$

Assuming $B(R) \leq (M-1)^2$

Cost = 3 B(R)

Read+write+read
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 $i_1 \leq N$ 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(A1, A2, …, An)          // A1, …, An are sorted
  i1 =0;  …, in = 0;  j=0
  while i1 ≤ N or … or i1 ≤ N
        let  Ak[ik] = min(A1[i1], …, An[in])
        T[j++] = Ak[ik++]
```

Merging n Arrays

```
Merge(A1, A2, …, An)          // A1, …, An are sorted
  i1 =0;  …, in = 0;  j=0
  while i1 ≤ N or … or i1 ≤ N
        let  Ak[ik] = min(A1[i1], …, An[in])
        T[j++] = Ak[ik++]
```

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

Merging n Arrays

```
Merge(A1, A2, …, An)          // A1, …, An are sorted
  i1 =0;  …, in = 0;  j=0
  while i1 ≤ N or … or i1 ≤ N
        let  Ak[ik] = min(A1[i1], …, An[in])
        T[j++] = Ak[ik++]
```

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

Additional log n factor to find min using priority queue

Merging n Arrays

```
Merge(A1, A2, …, An)          // A1, …, An are sorted
  i1 =0;  …, in = 0;  j=0
  while i1 ≤ N or … or i1 ≤ N
        let  Ak[ik] = min(A1[i1], …, An[in])
        T[j++] = Ak[ik++]
```

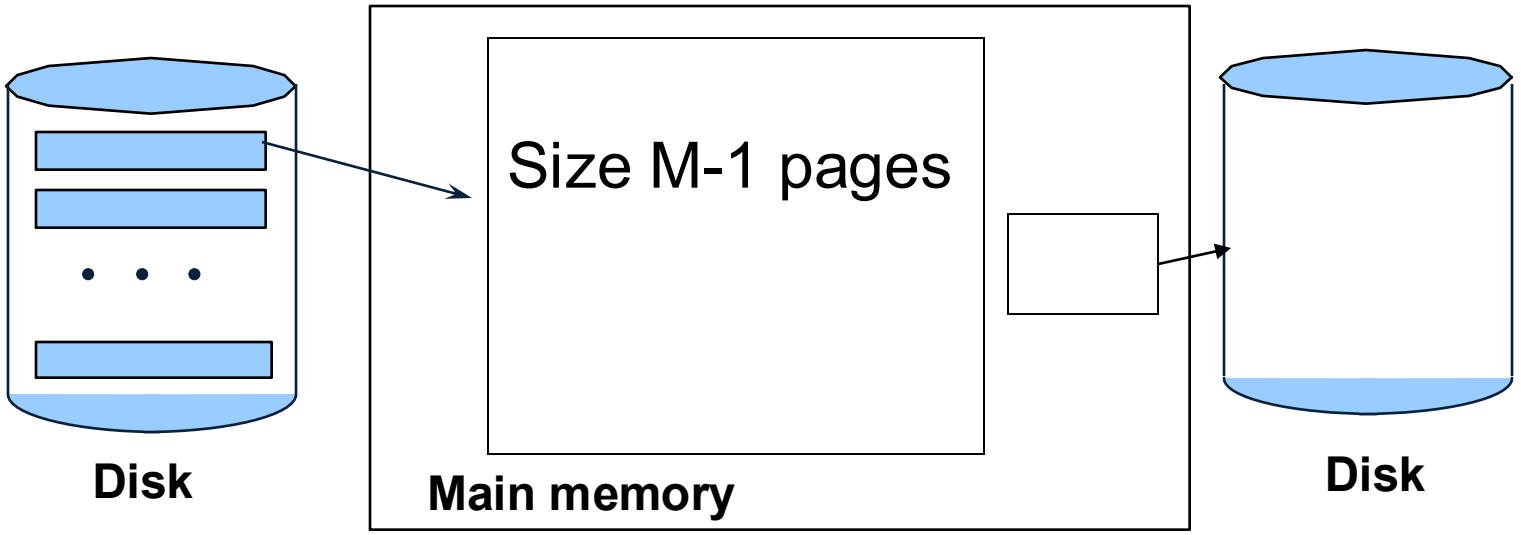
Important:
Need to read
only one element
from each array

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

Additional log n factor to find min using priority queue

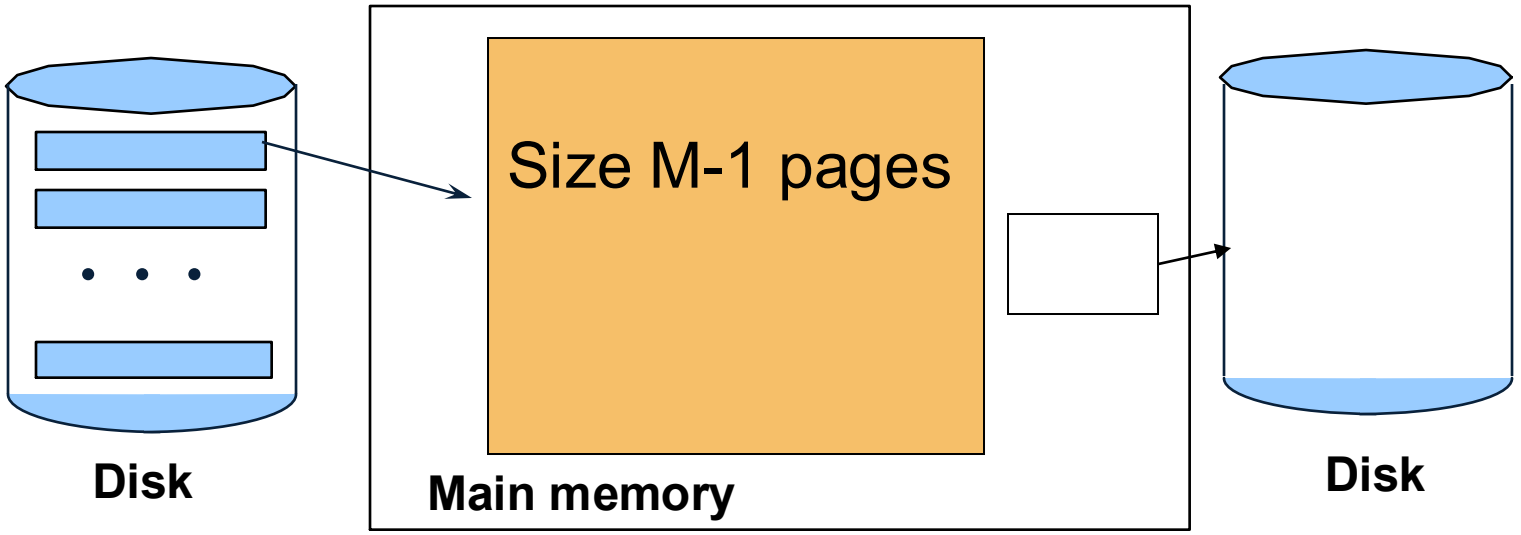
Merge-Sort: Step 1

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



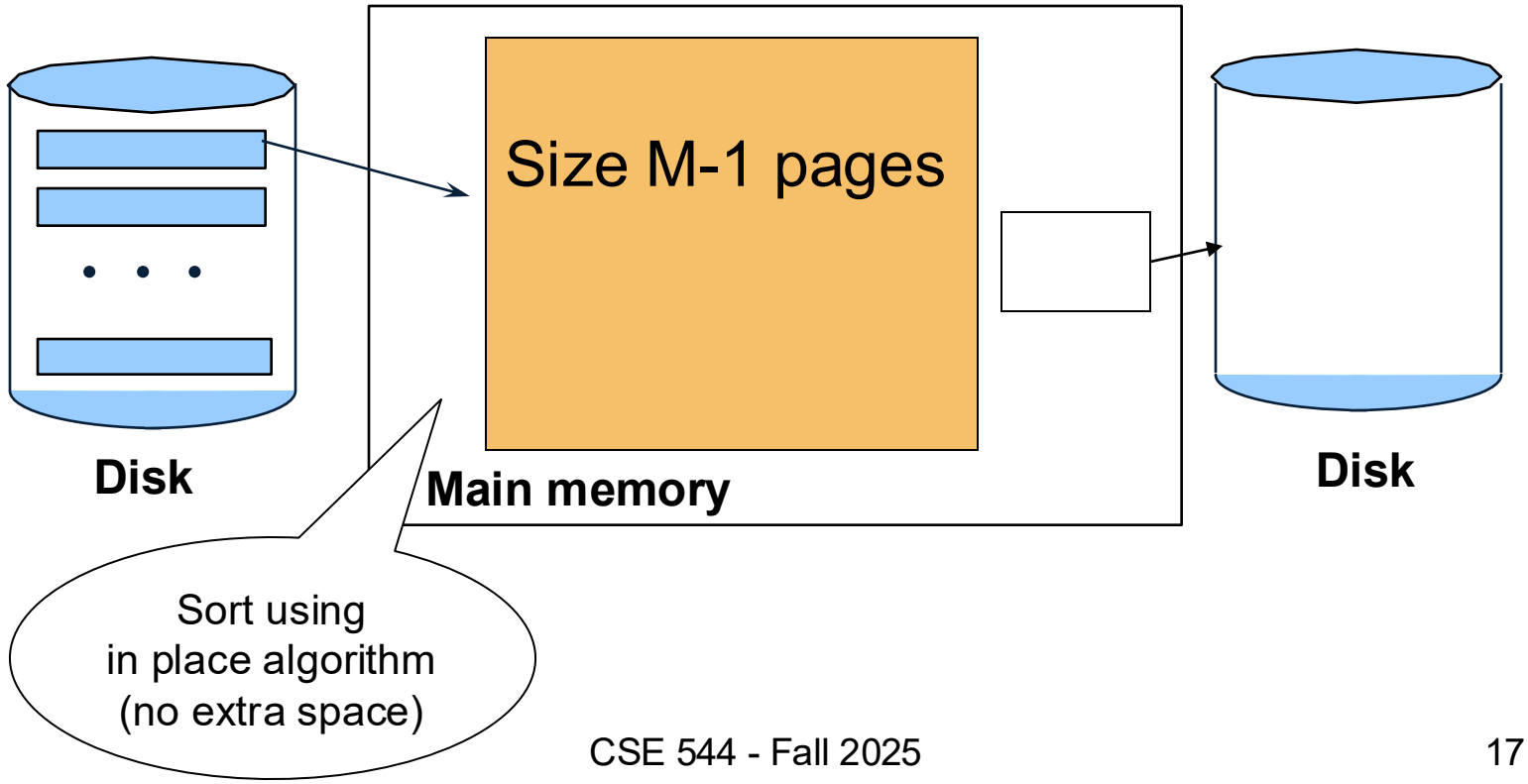
Merge-Sort: Step 1

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



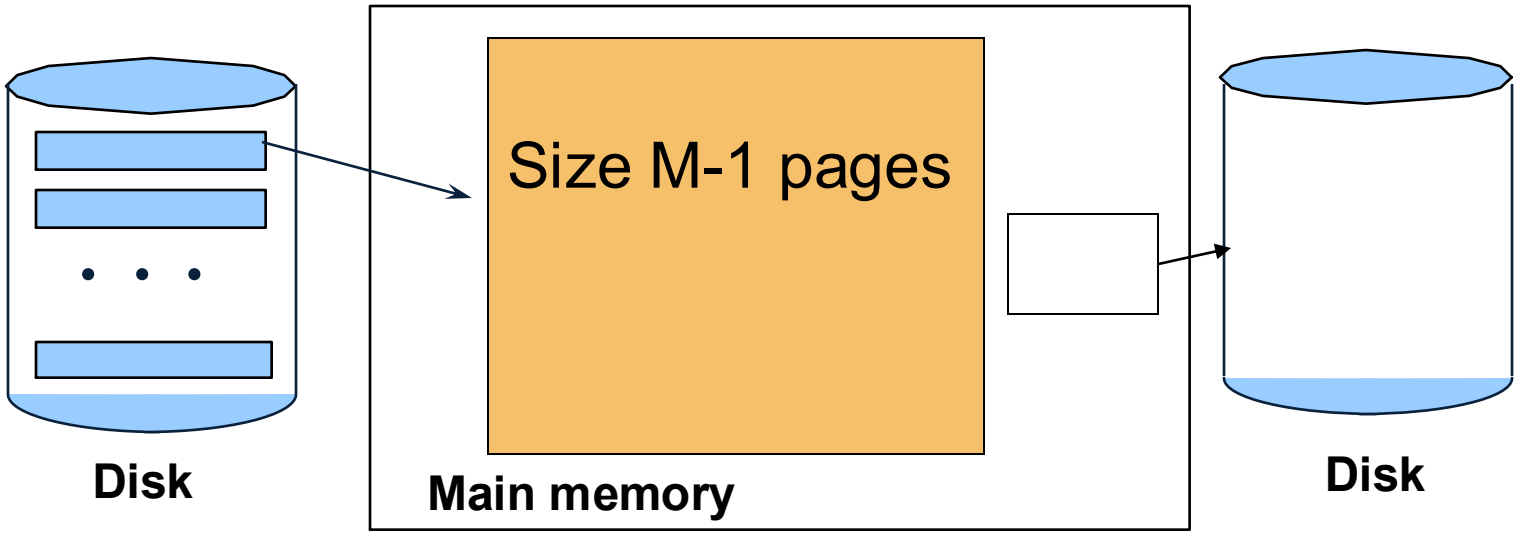
Merge-Sort: Step 1

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



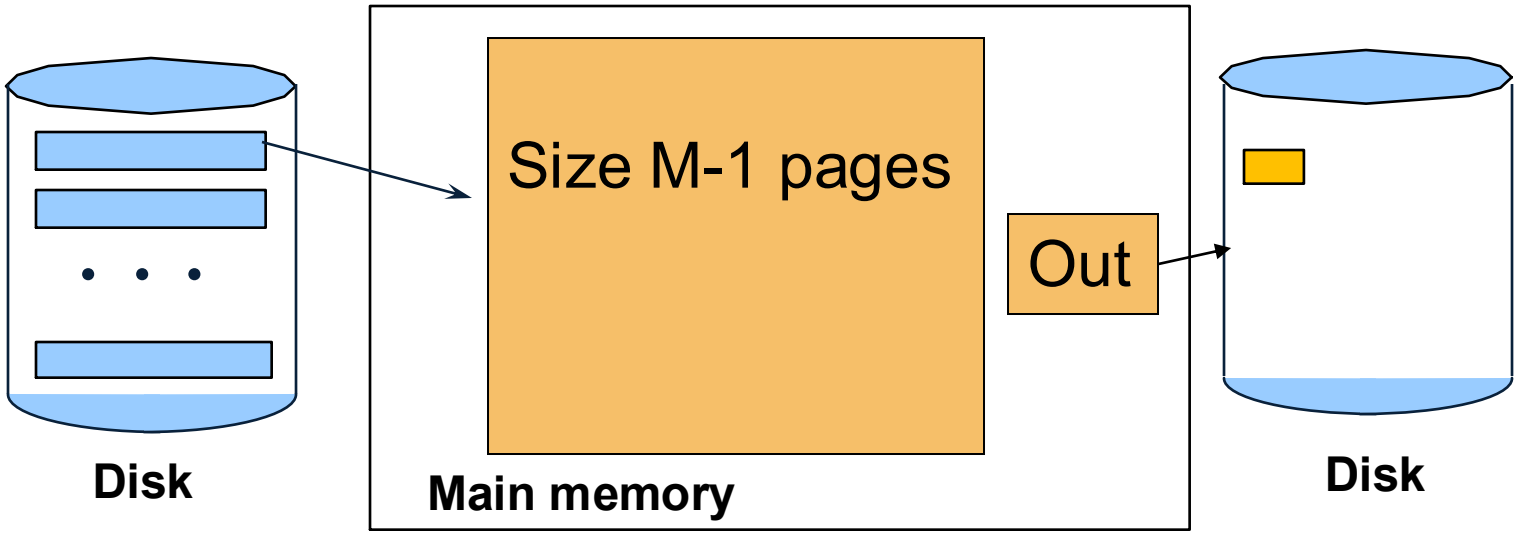
Merge-Sort: Step 1

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



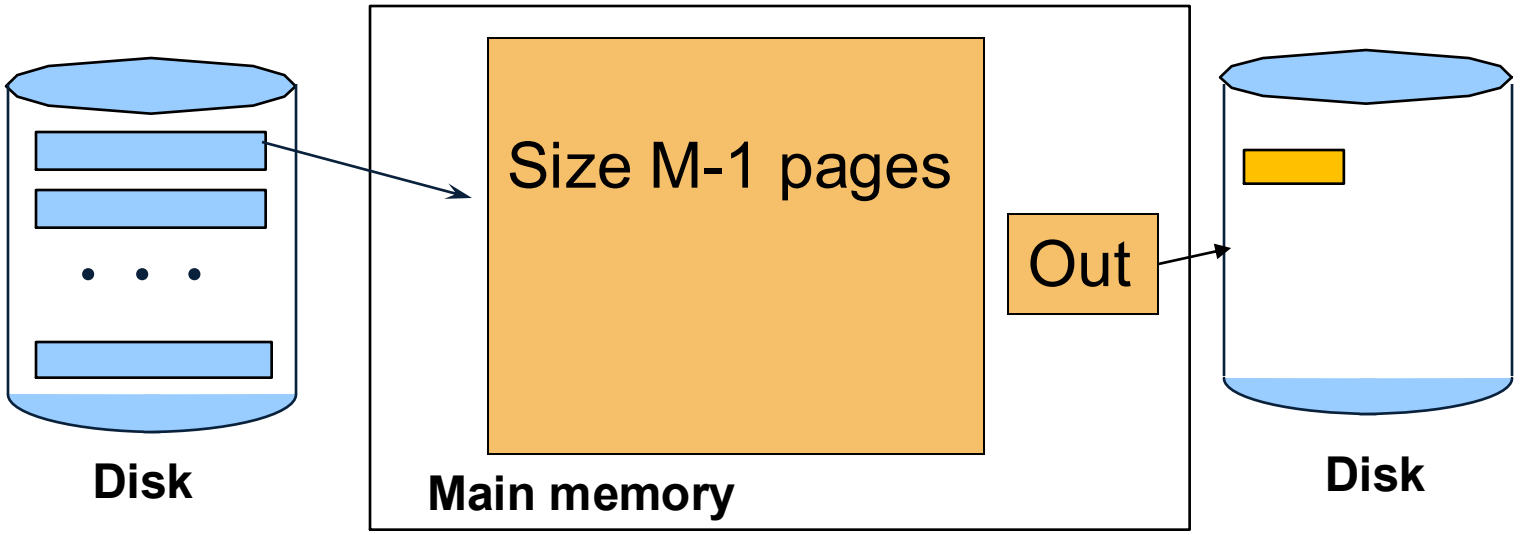
Merge-Sort: Step 1

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



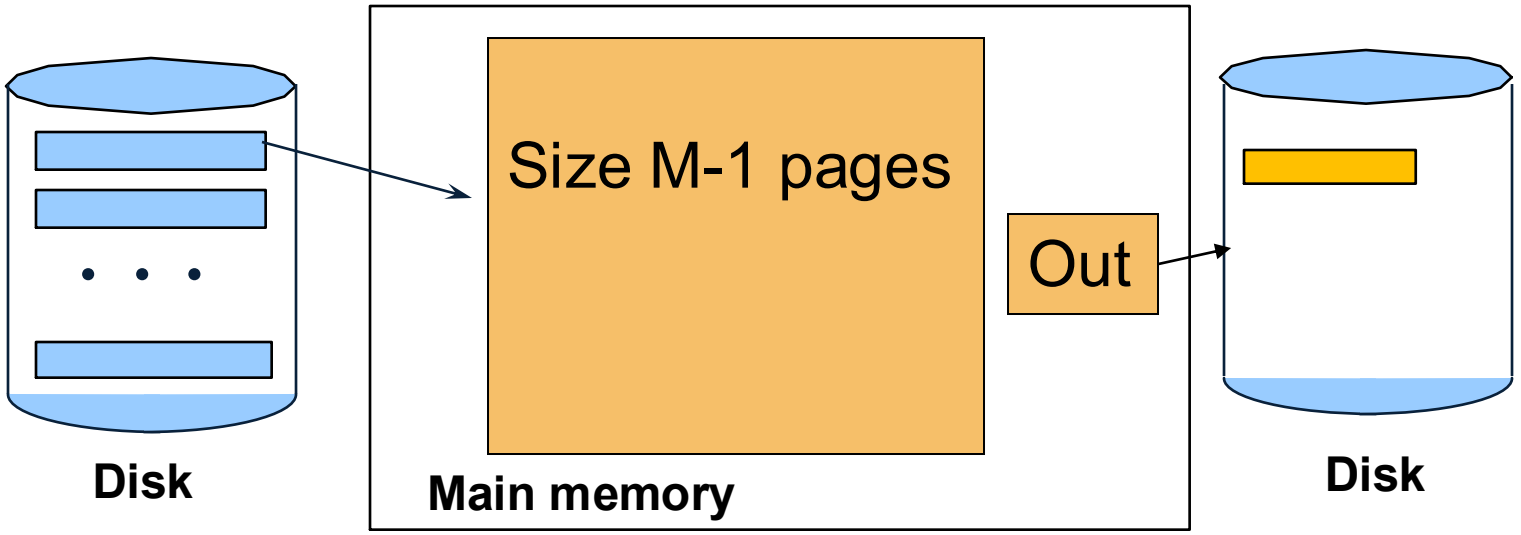
Merge-Sort: Step 1

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



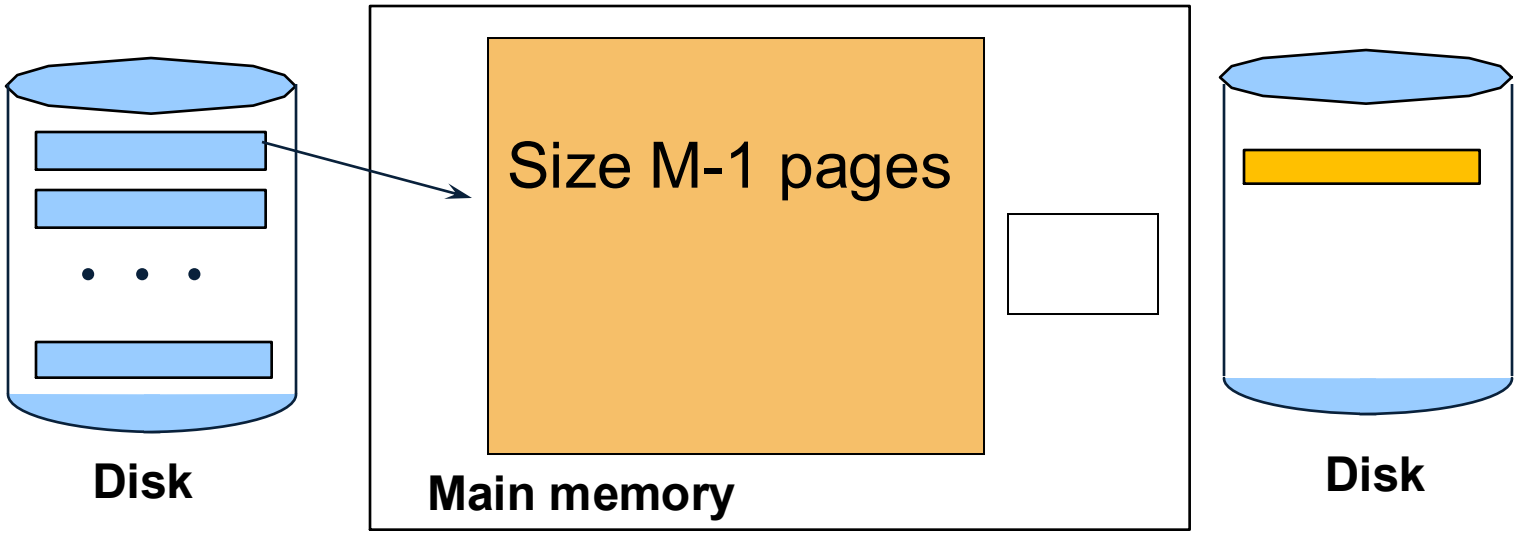
Merge-Sort: Step 1

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



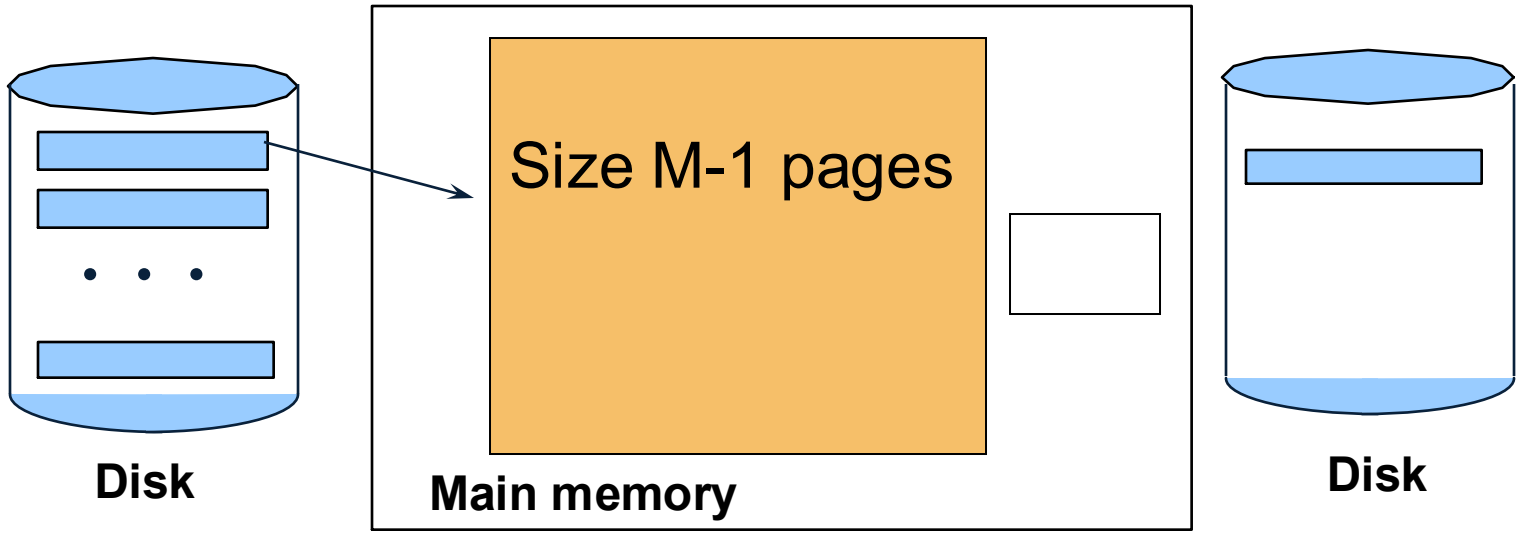
Merge-Sort: Step 1

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



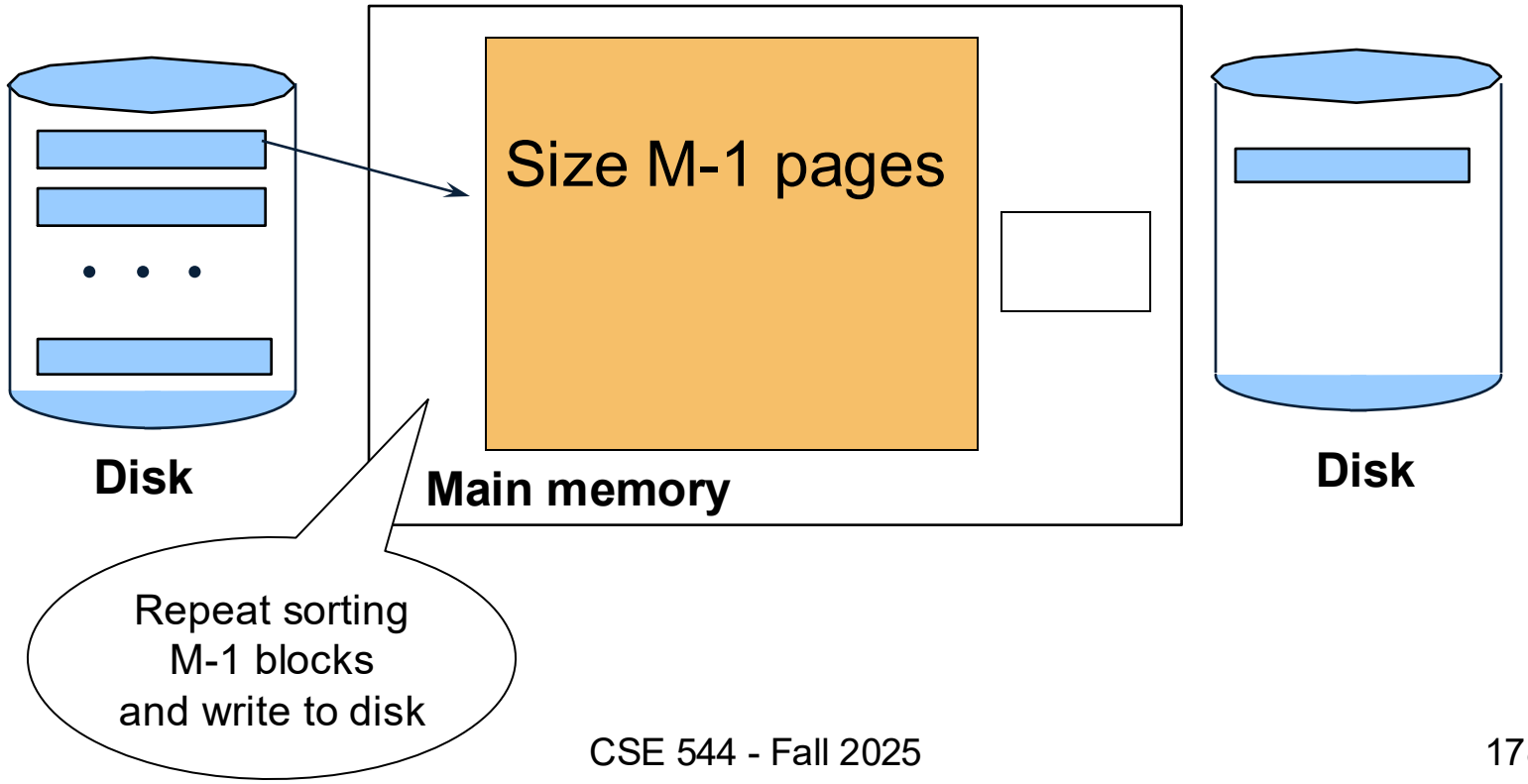
Merge-Sort: Step 1

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



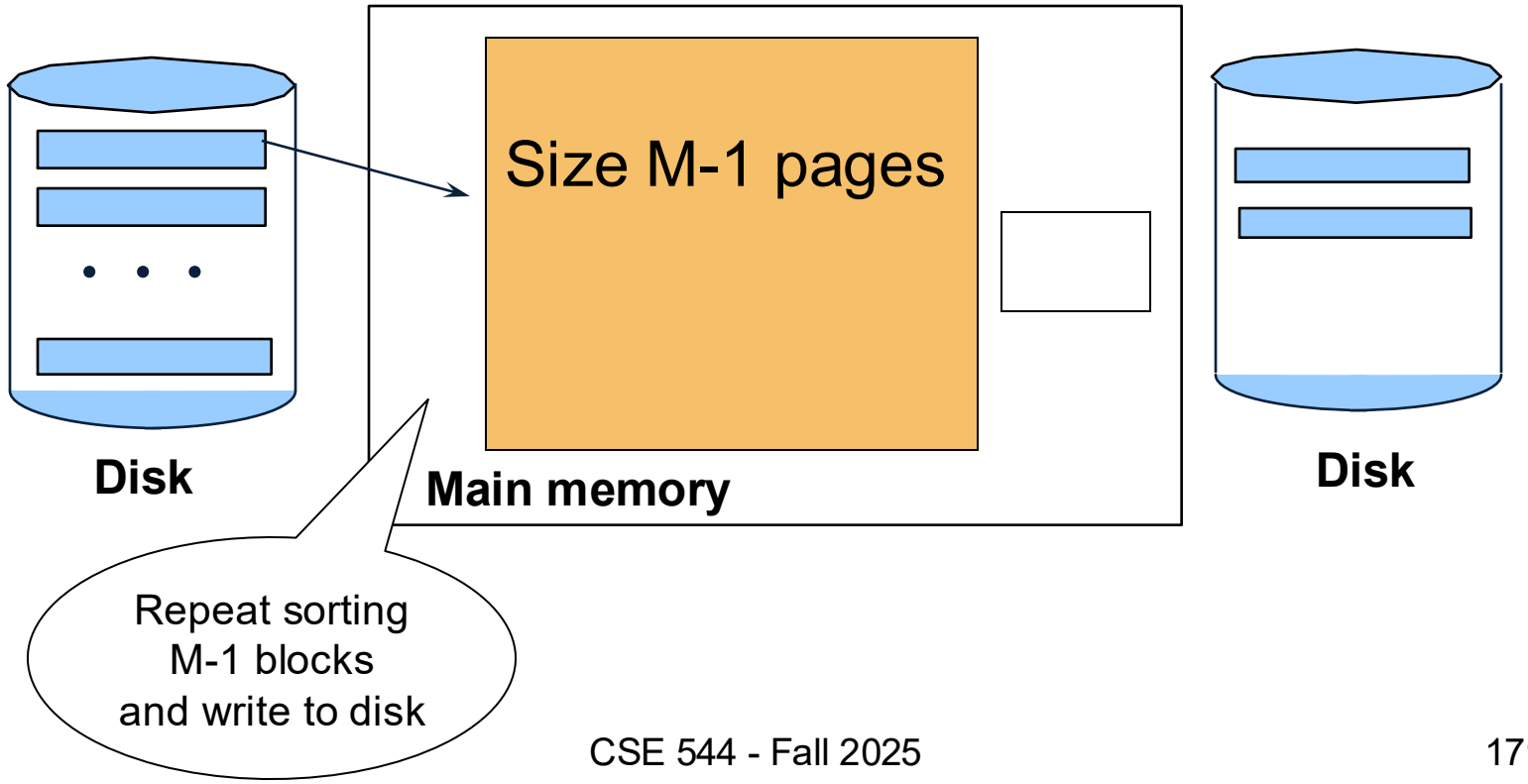
Merge-Sort: Step 1

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



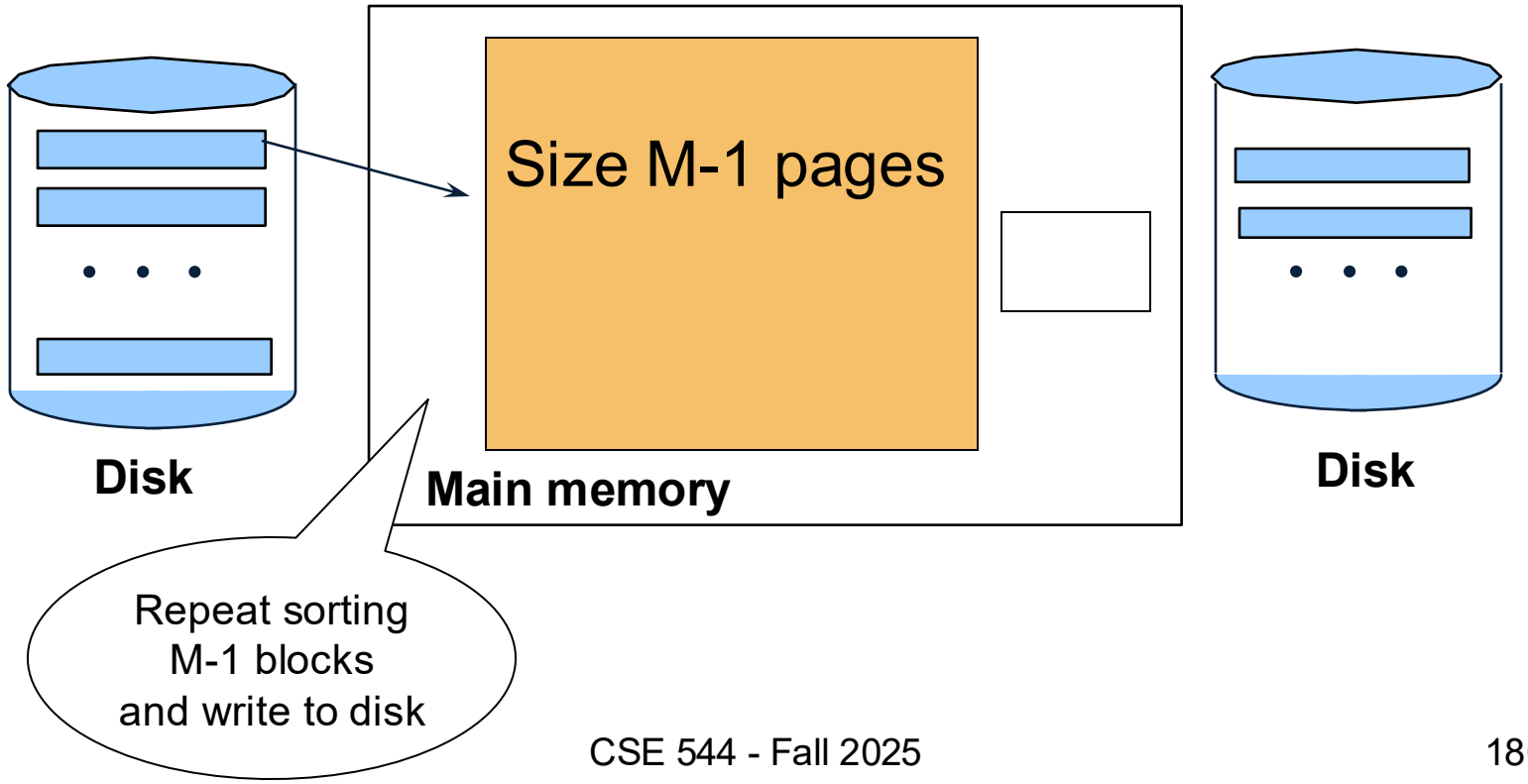
Merge-Sort: Step 1

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



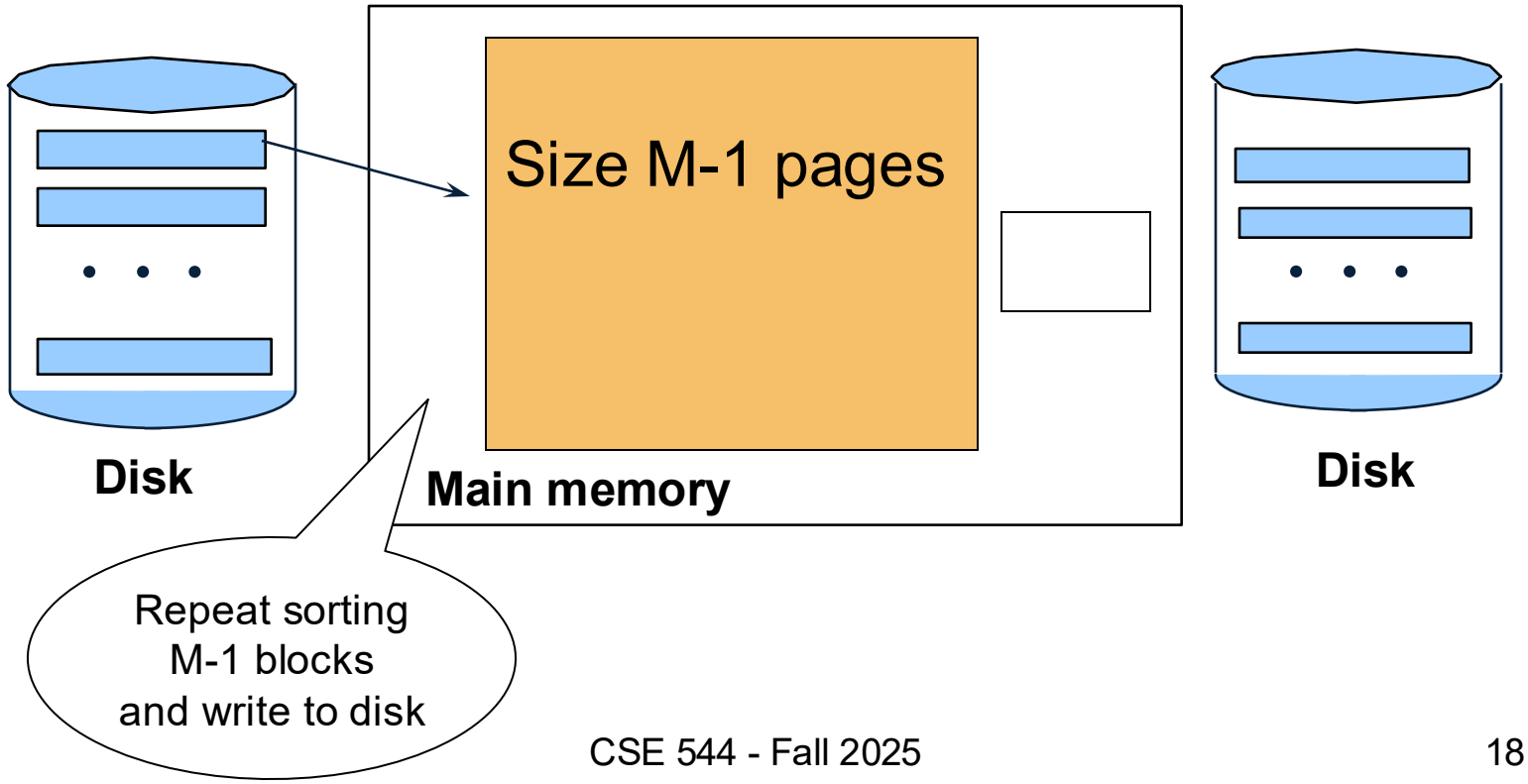
Merge-Sort: Step 1

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



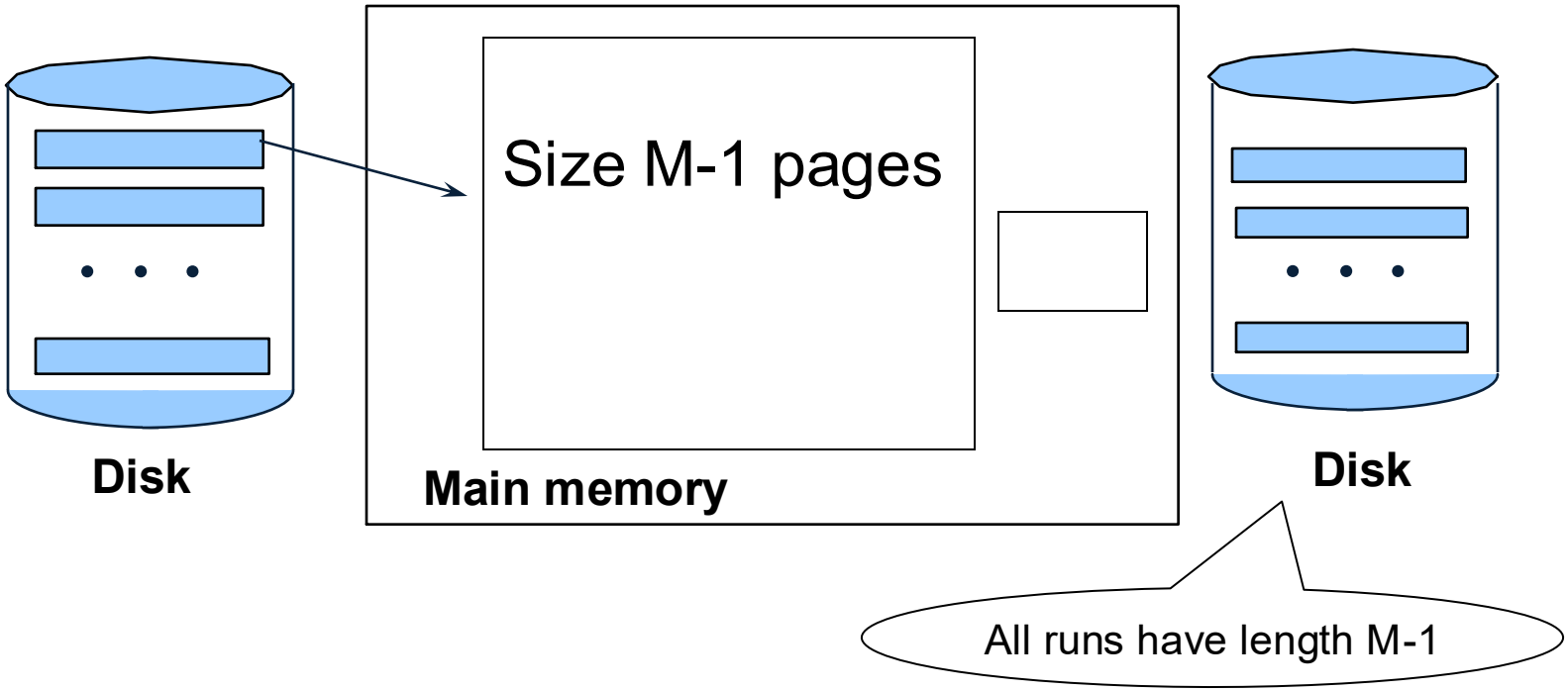
Merge-Sort: Step 1

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



Merge-Sort: Step 1

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

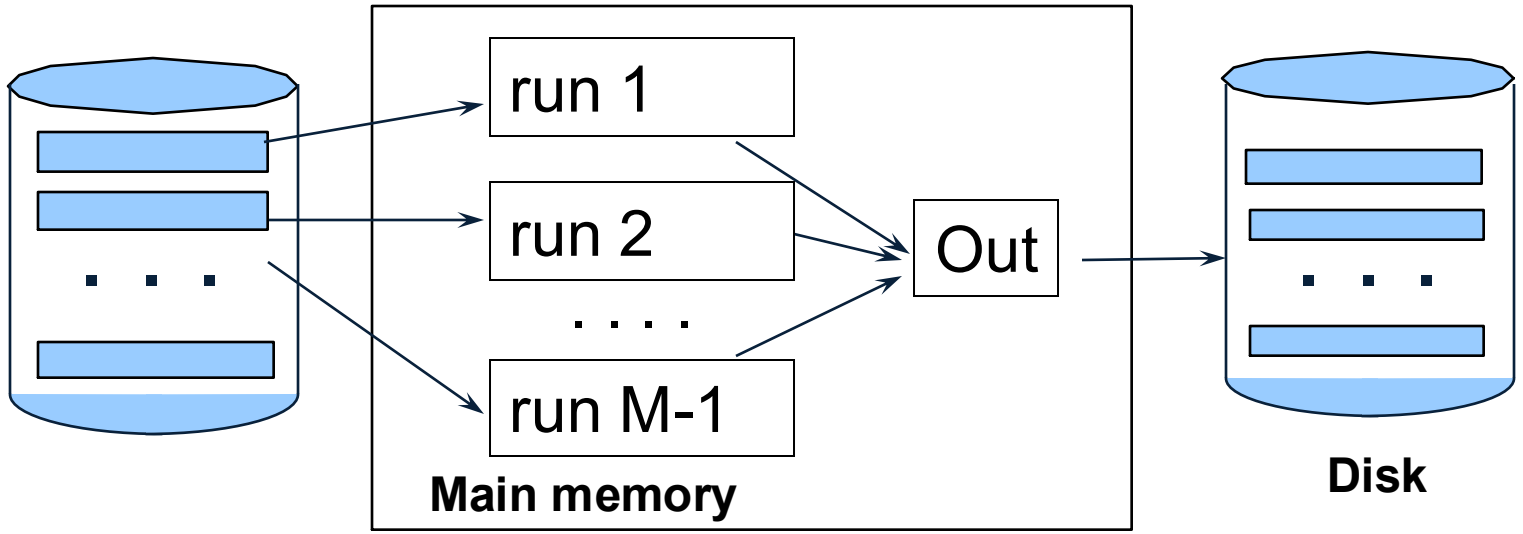


Merge-Sort: Steps 2, 3, …

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

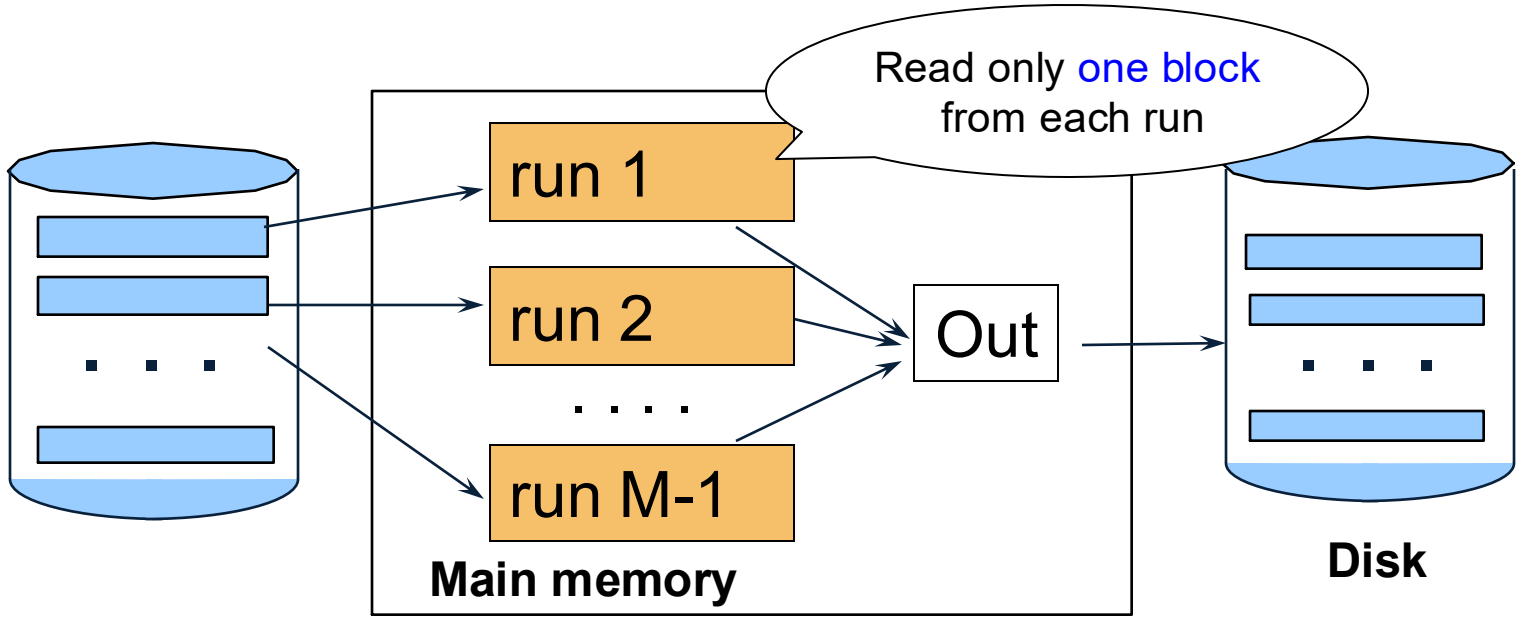
Merge-Sort: Steps 2, 3, …

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



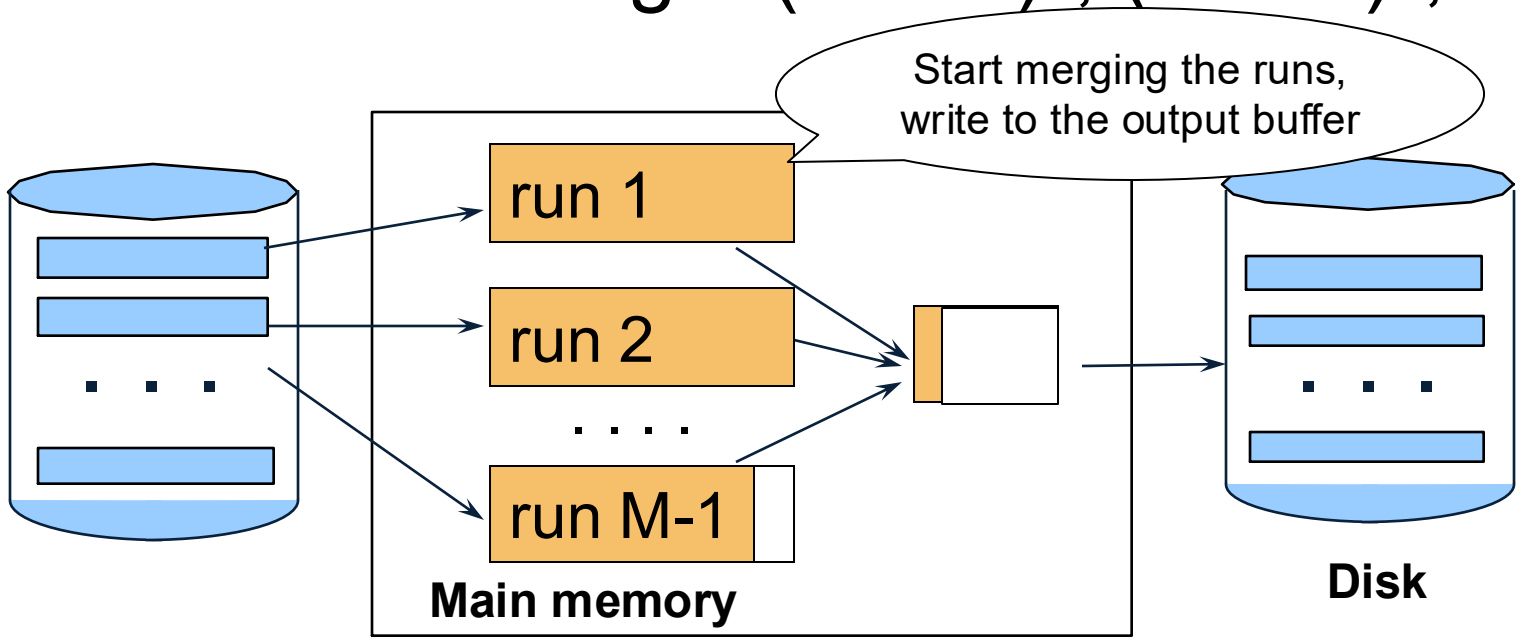
Merge-Sort: Steps 2, 3, …

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



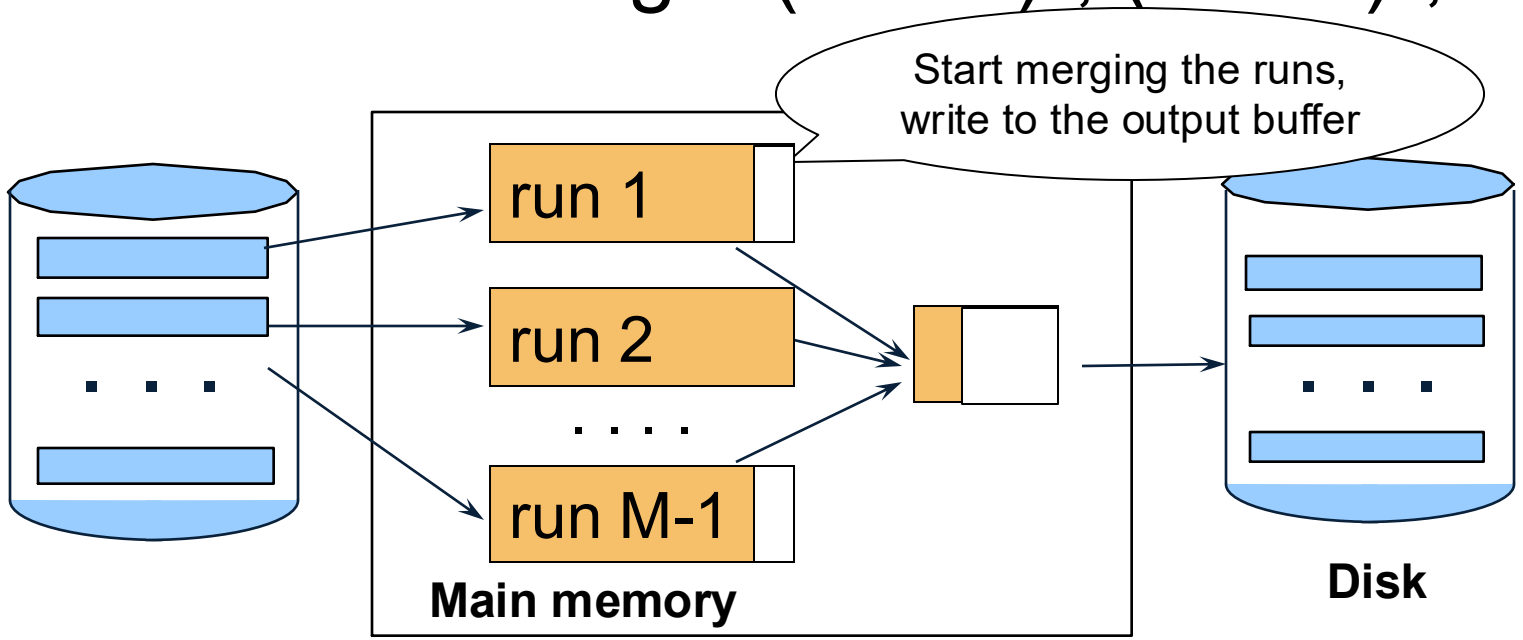
Merge-Sort: Steps 2, 3, …

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



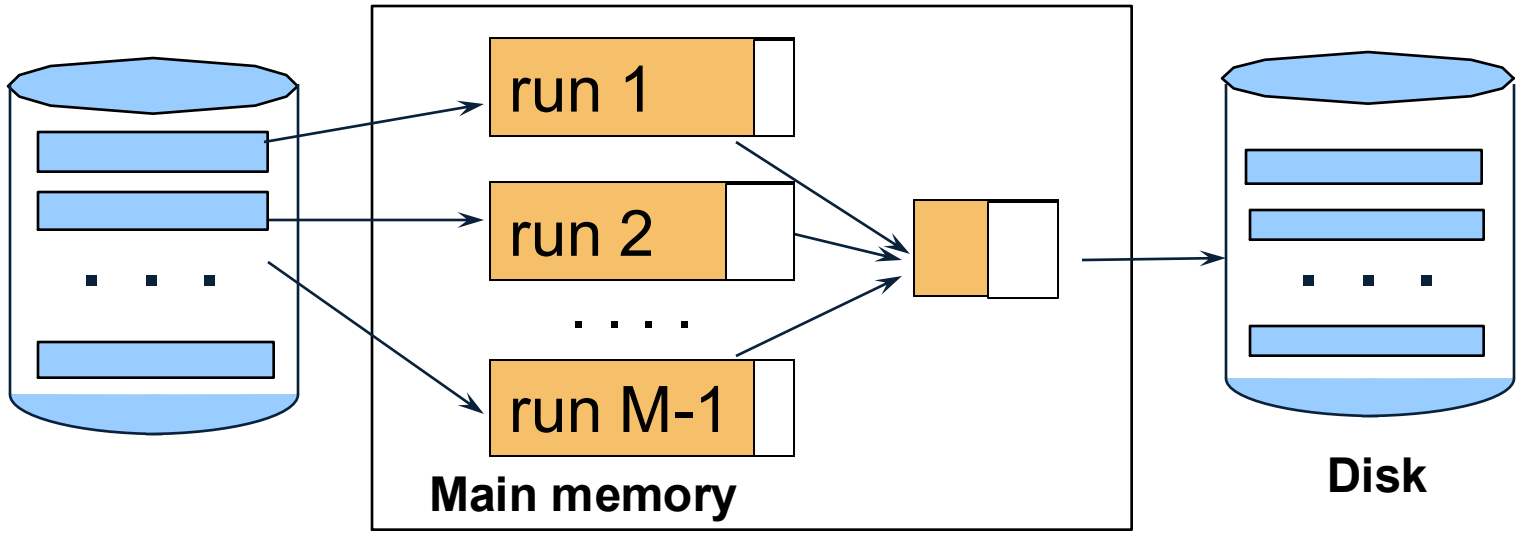
Merge-Sort: Steps 2, 3, …

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



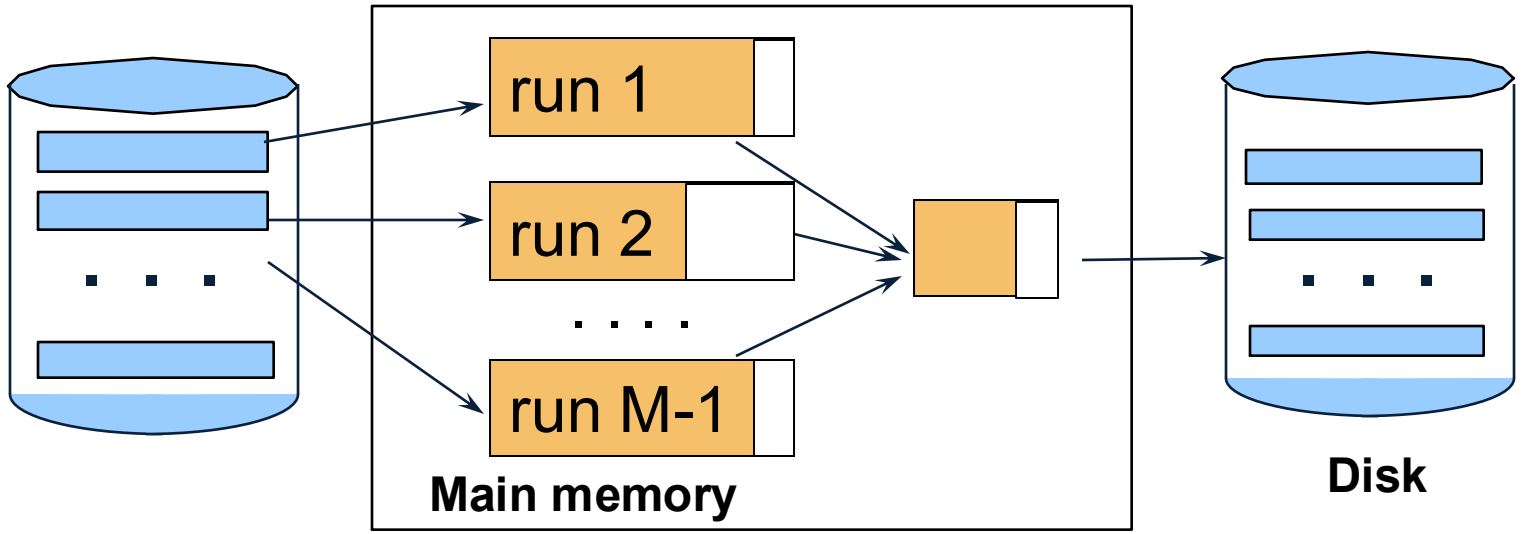
Merge-Sort: Steps 2, 3, …

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



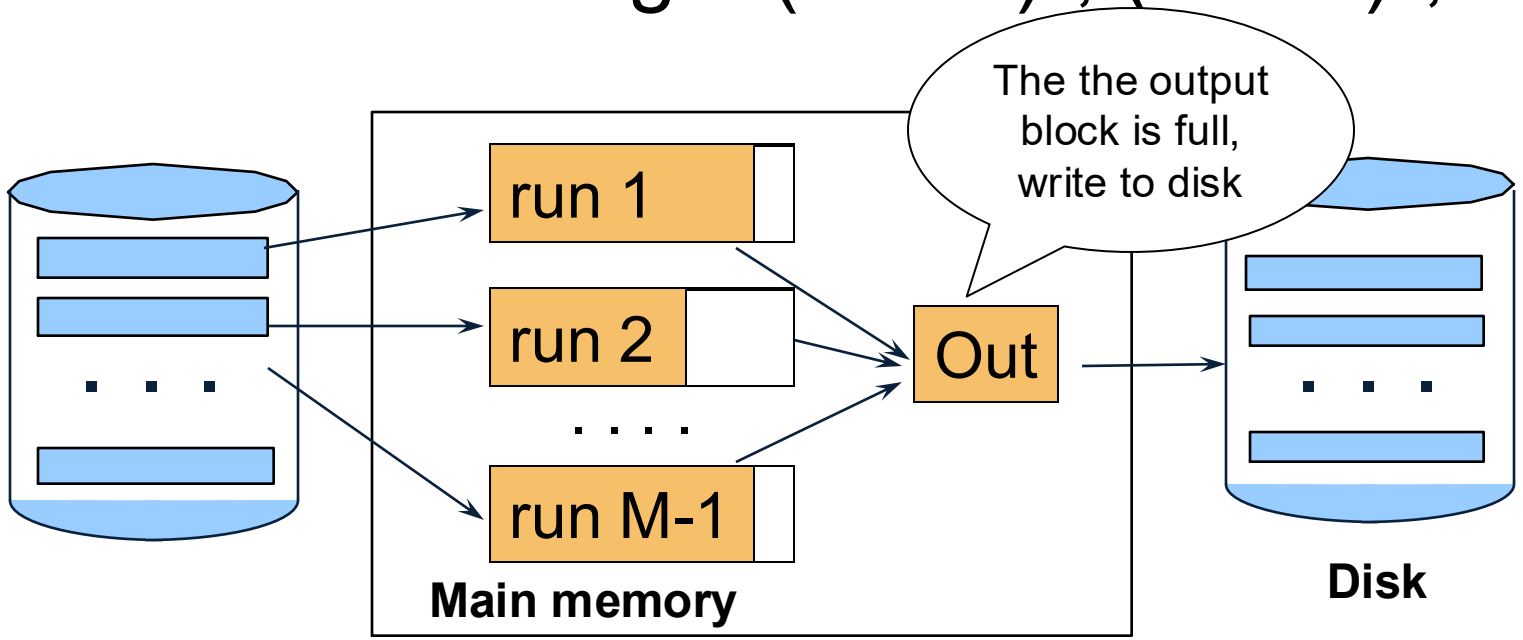
Merge-Sort: Steps 2, 3, …

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



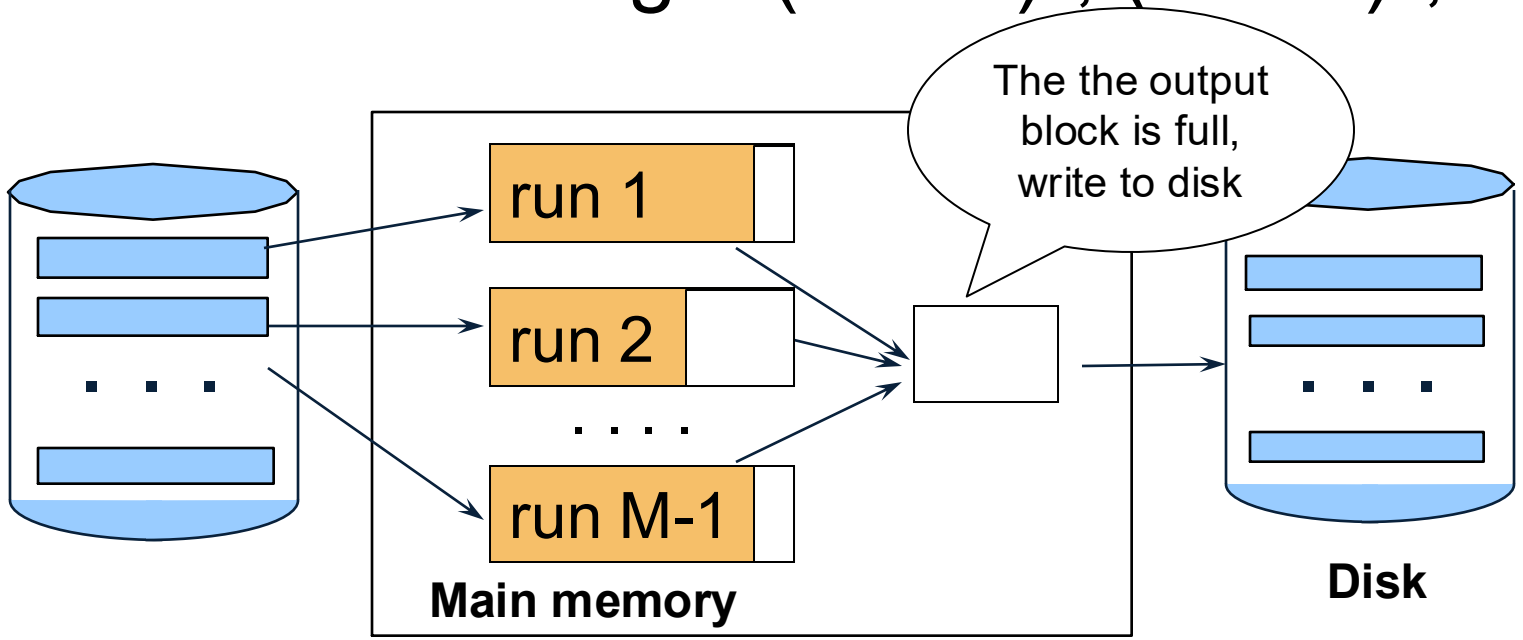
Merge-Sort: Steps 2, 3, …

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



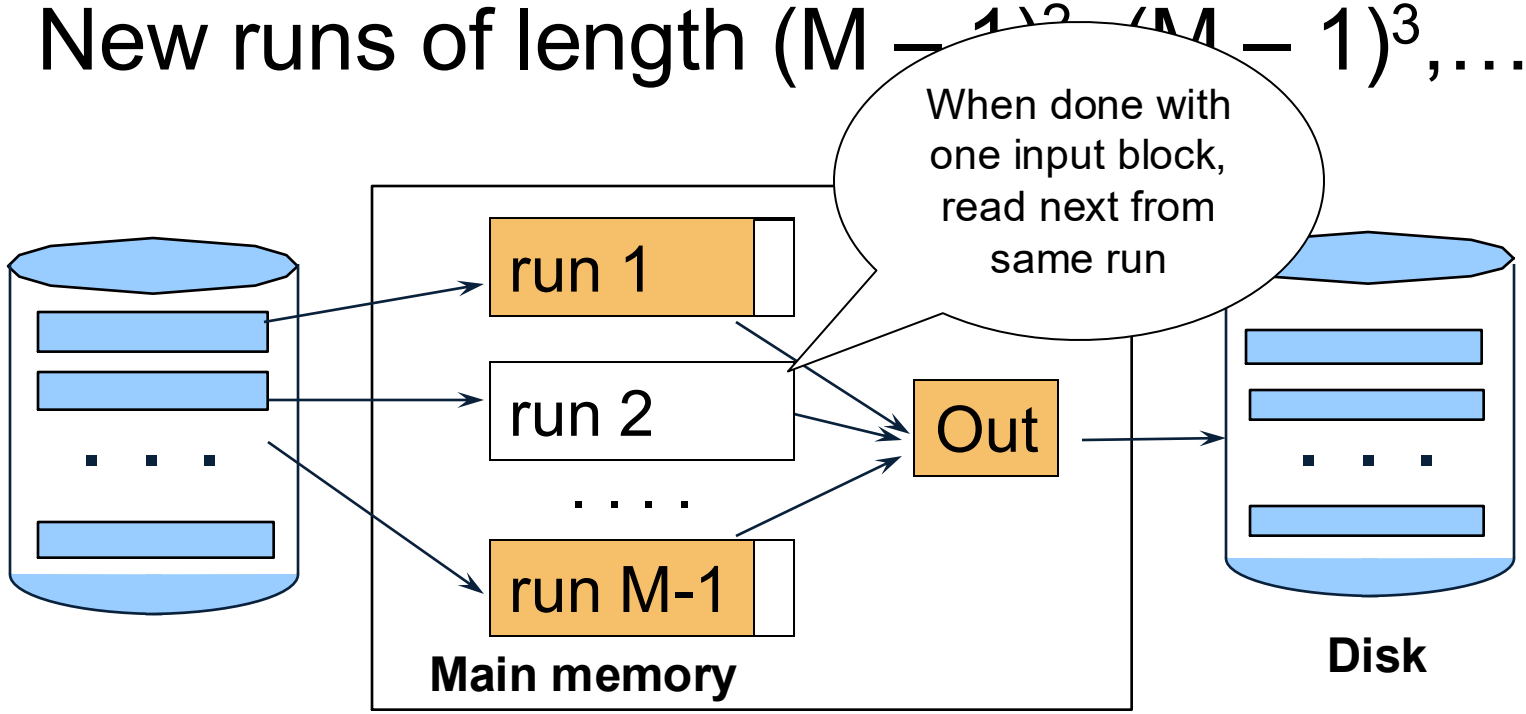
Merge-Sort: Steps 2, 3, …

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



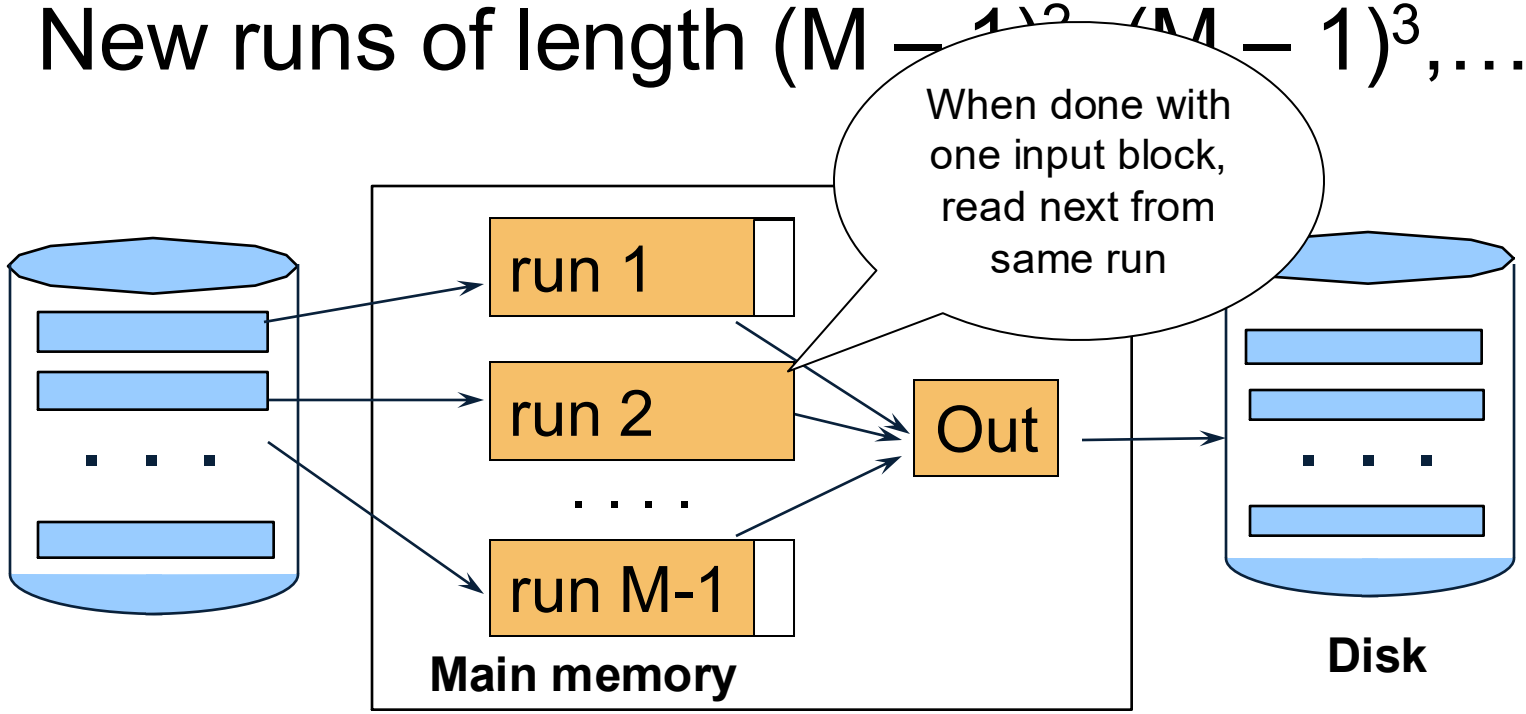
Merge-Sort: Steps 2, 3, …

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



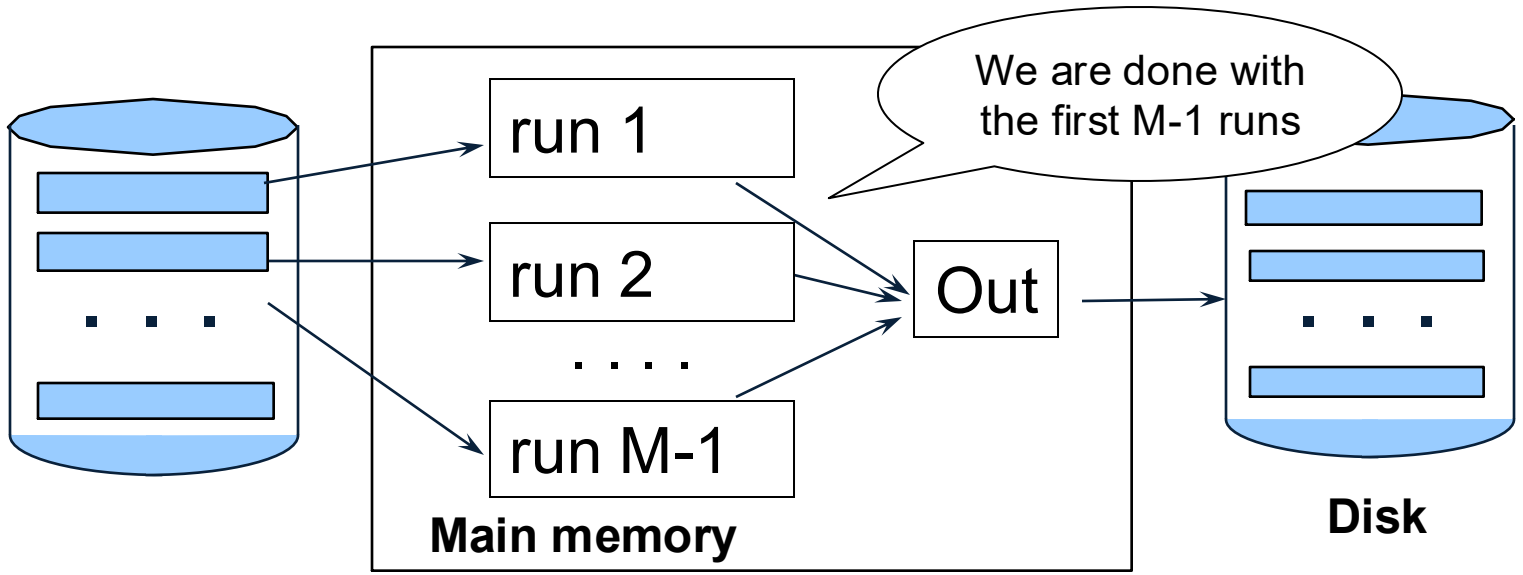
Merge-Sort: Steps 2, 3, …

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



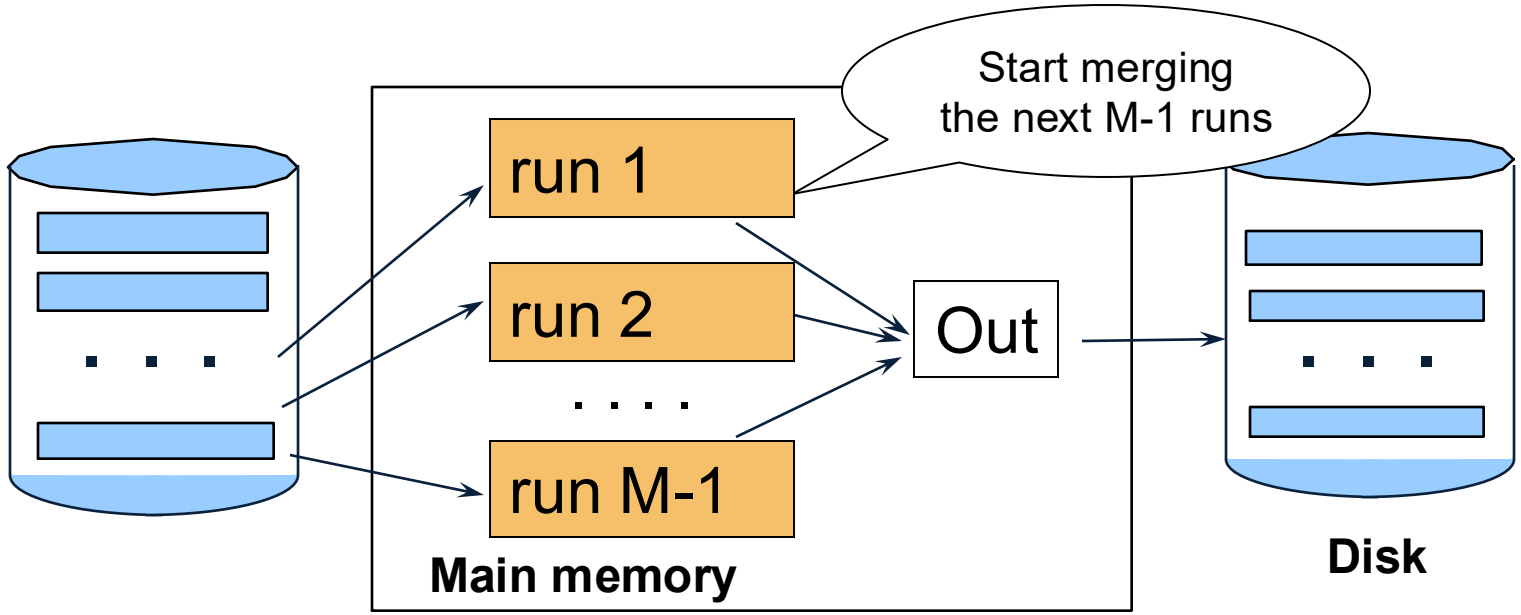
Merge-Sort: Steps 2, 3, …

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



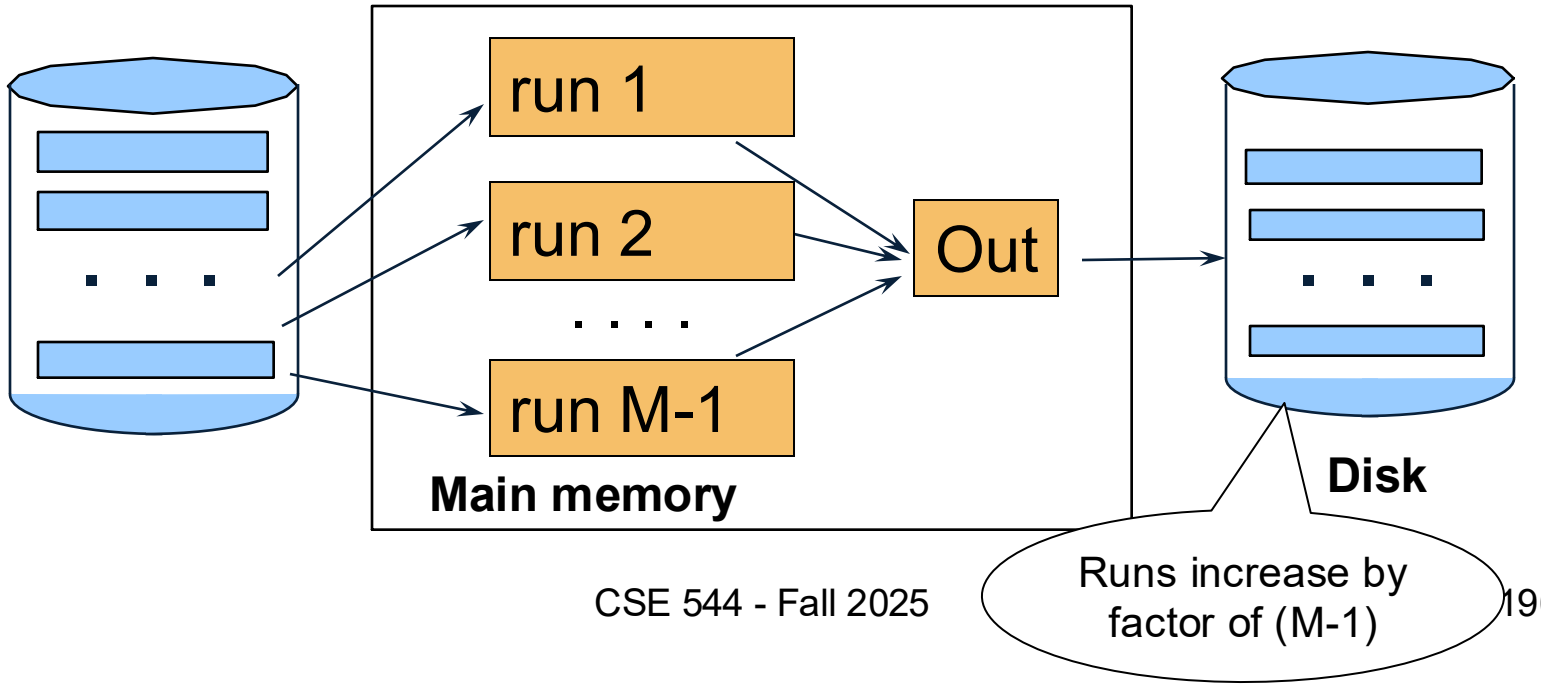
Merge-Sort: Steps 2, 3, …

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



Merge-Sort: Steps 2, 3, …

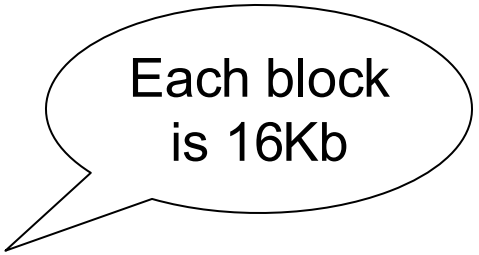
- Merge M – 1 runs into a new run
- New runs of length $(M – 1)^2$, $(M – 1)^3$,…



Merge-Sort: Discussion

Size of the runs increases fast

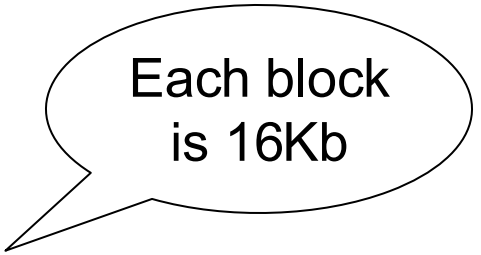
- Example: M=10001



Merge-Sort: Discussion

Size of the runs increases fast

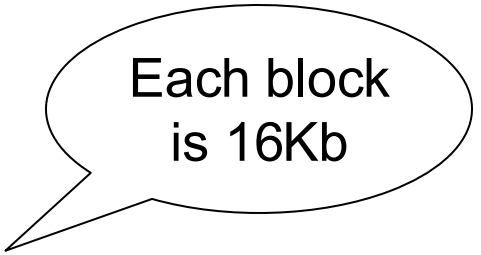
- Example: M=10001
- Step 1: run length 10000 = 160Mb



Merge-Sort: Discussion

Size of the runs increases fast

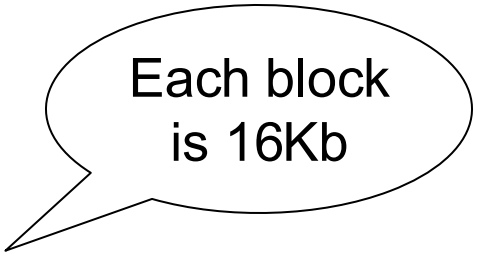
- Example: M=10001
- Step 1: run length 10000 = 160Mb
- Step 2: run length 100000000 = 1.6Tb
- …



Merge-Sort: Discussion

Size of the runs increases fast

- Example: M=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 ⋈ S

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

Finally: Merge-Join

R ⋈ 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))

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

- Cost: B(R)+B(S)

```
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 ⋈ S, both bigger than main memory

Partitioned Hash-Join

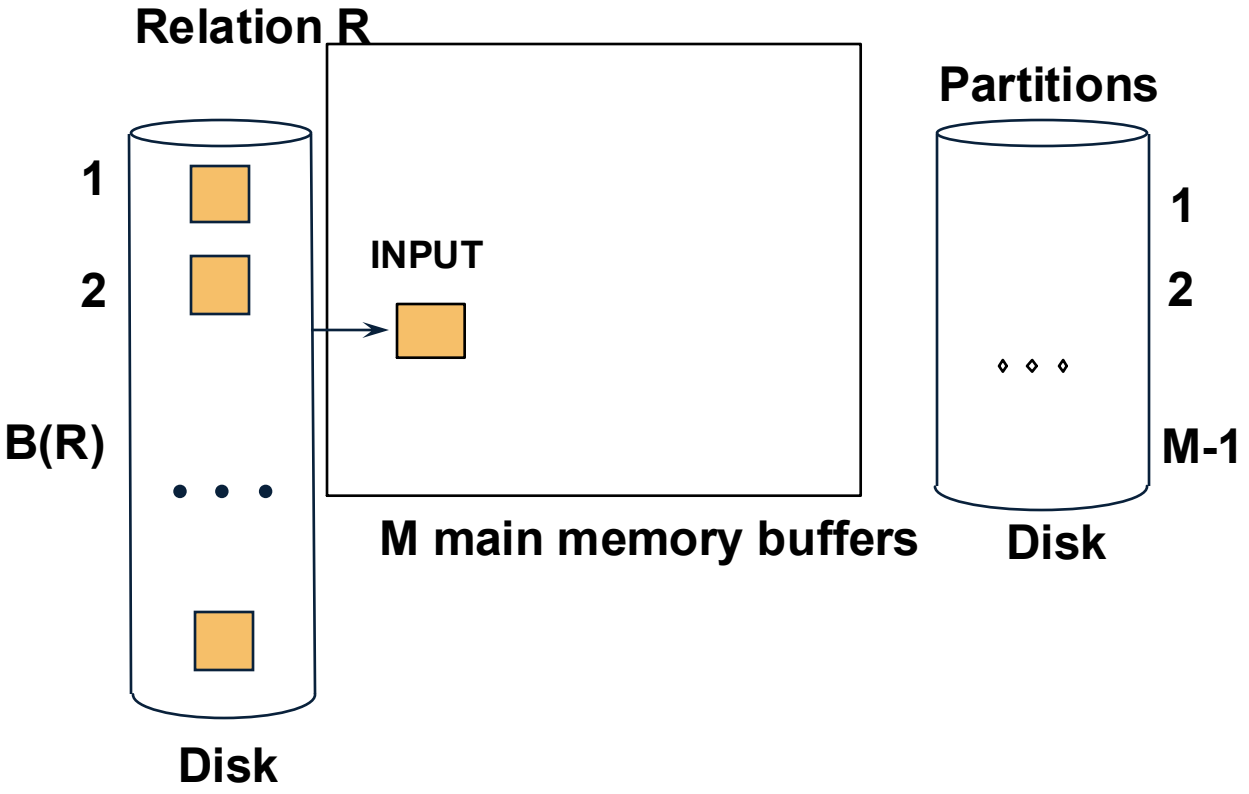
- R ⋈ S, both bigger than main memory
- Step 1:
  - Hash partition both R and S
  - Store buckets on disk

Partitioned Hash-Join

- R ⋈ 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

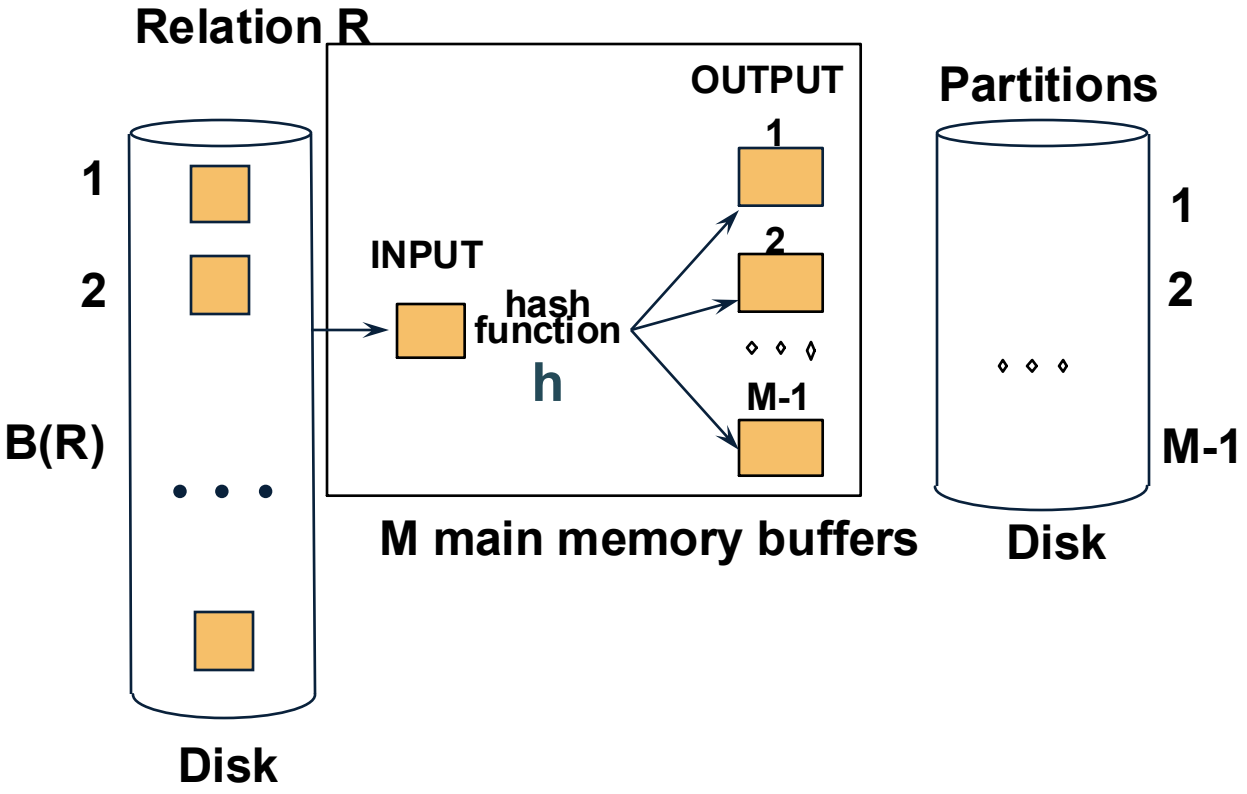
Step 1: Hash-partition

- Partition R into buckets, on disk



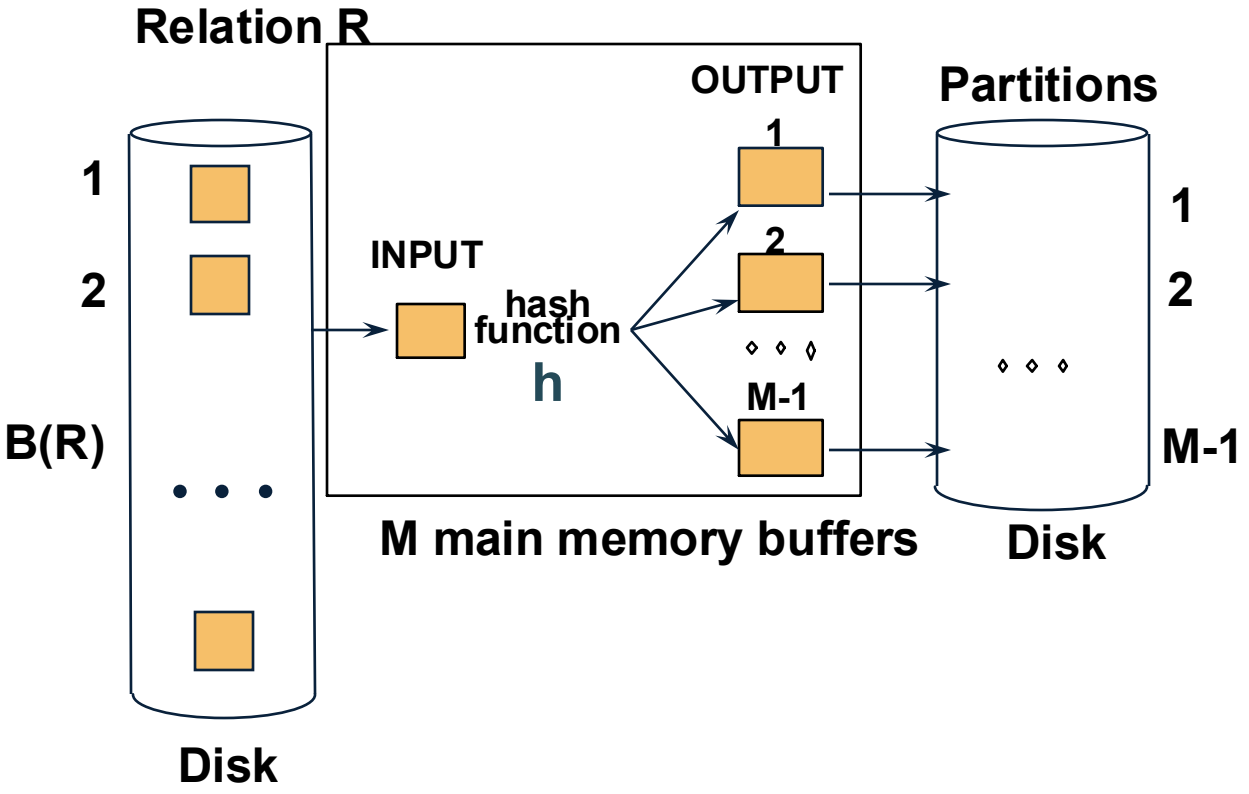
Step 1: Hash-partition

- Partition R into buckets, on disk



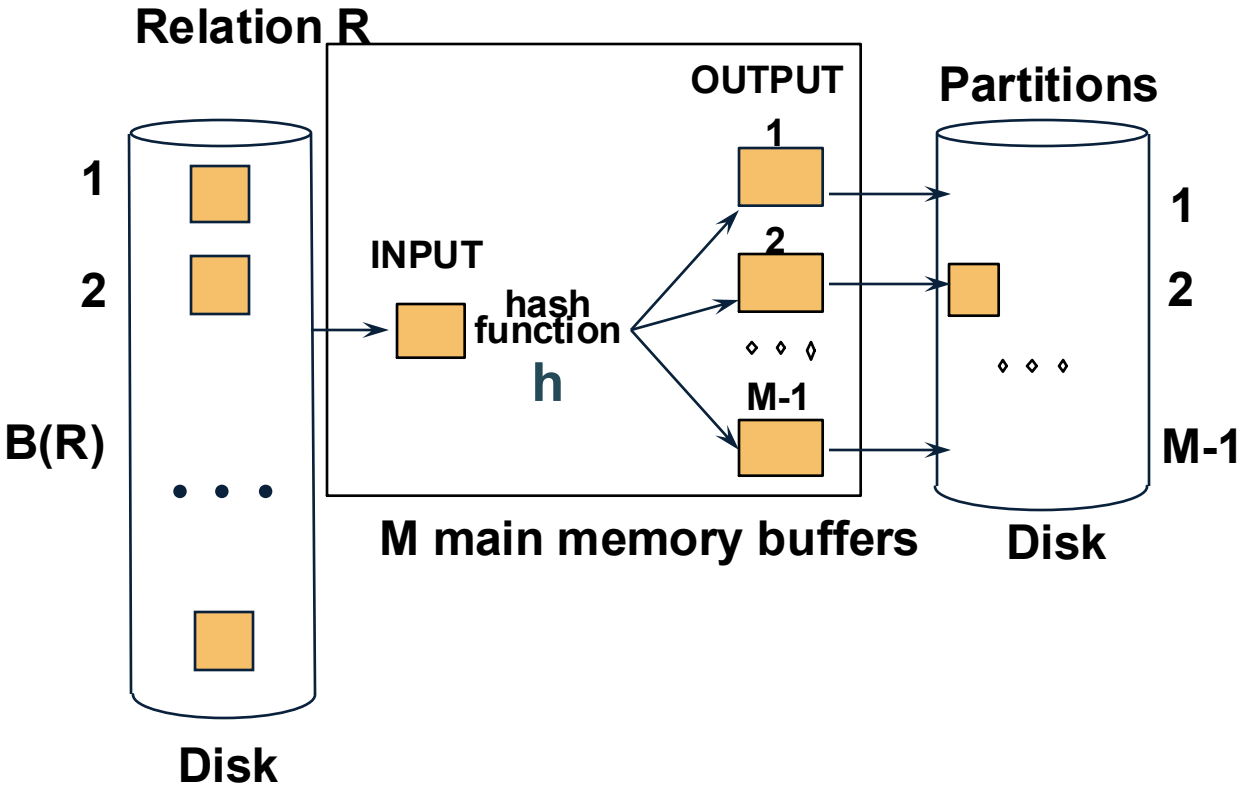
Step 1: Hash-partition

- Partition R into buckets, on disk



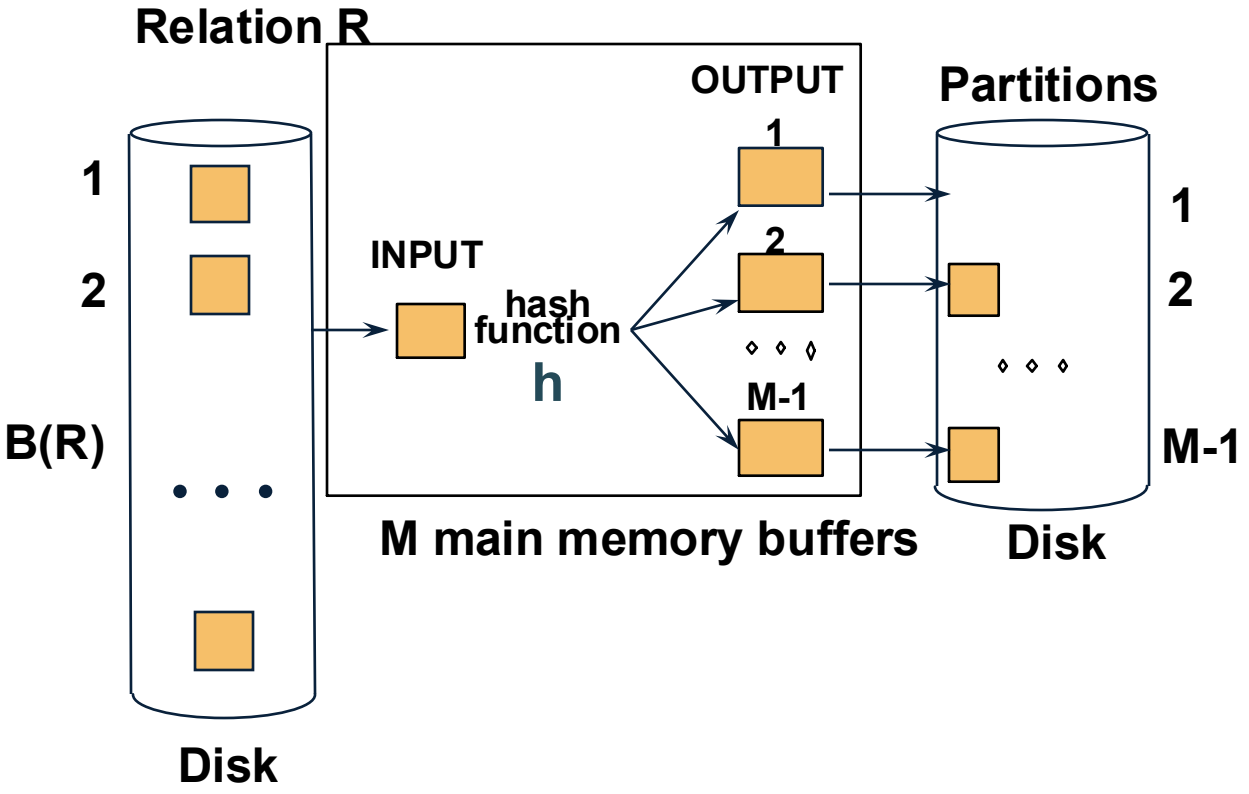
Step 1: Hash-partition

- Partition R into buckets, on disk



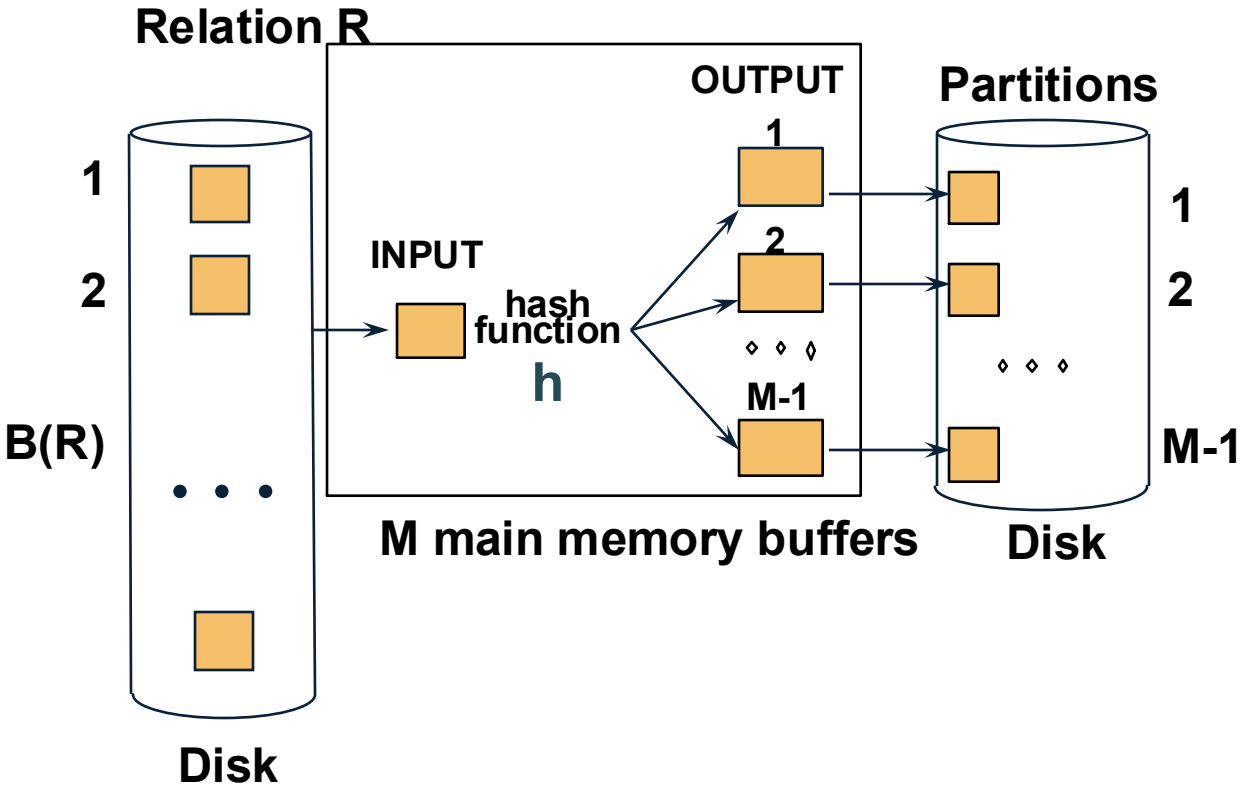
Step 1: Hash-partition

- Partition R into buckets, on disk



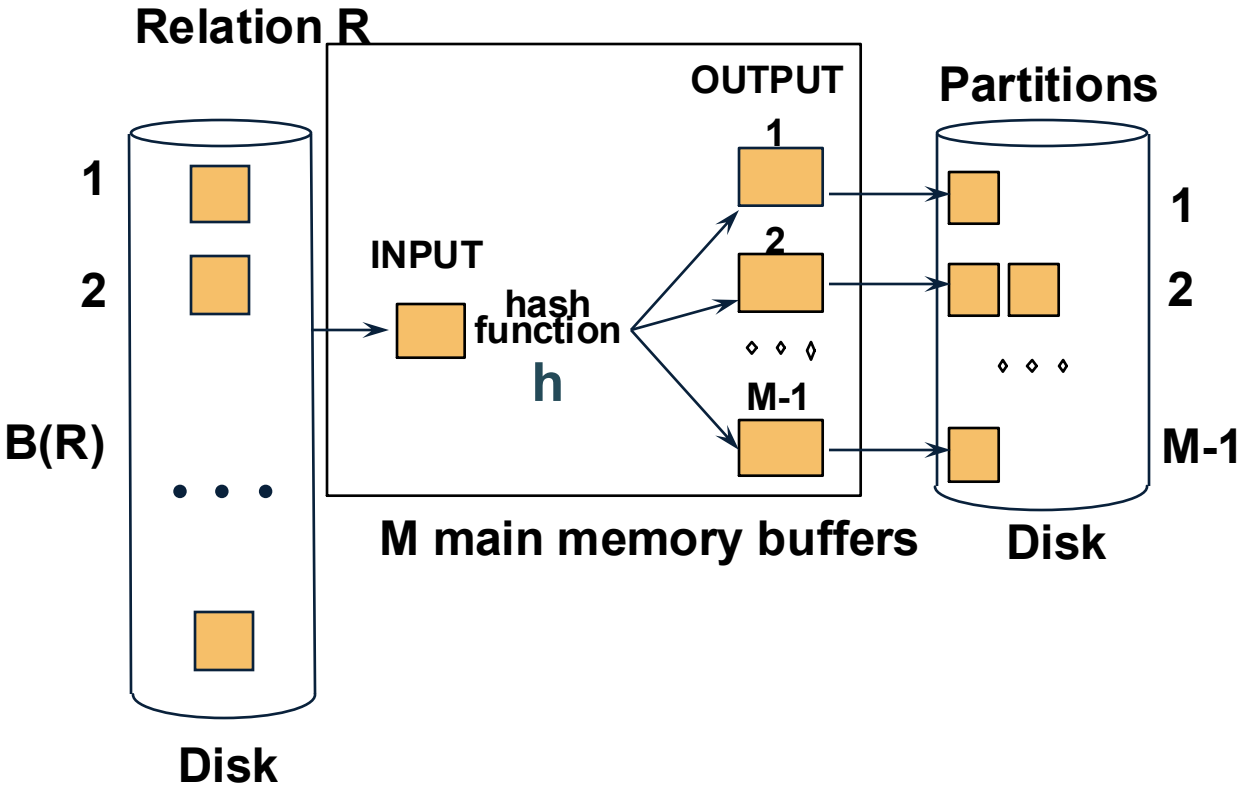
Step 1: Hash-partition

- Partition R into buckets, on disk



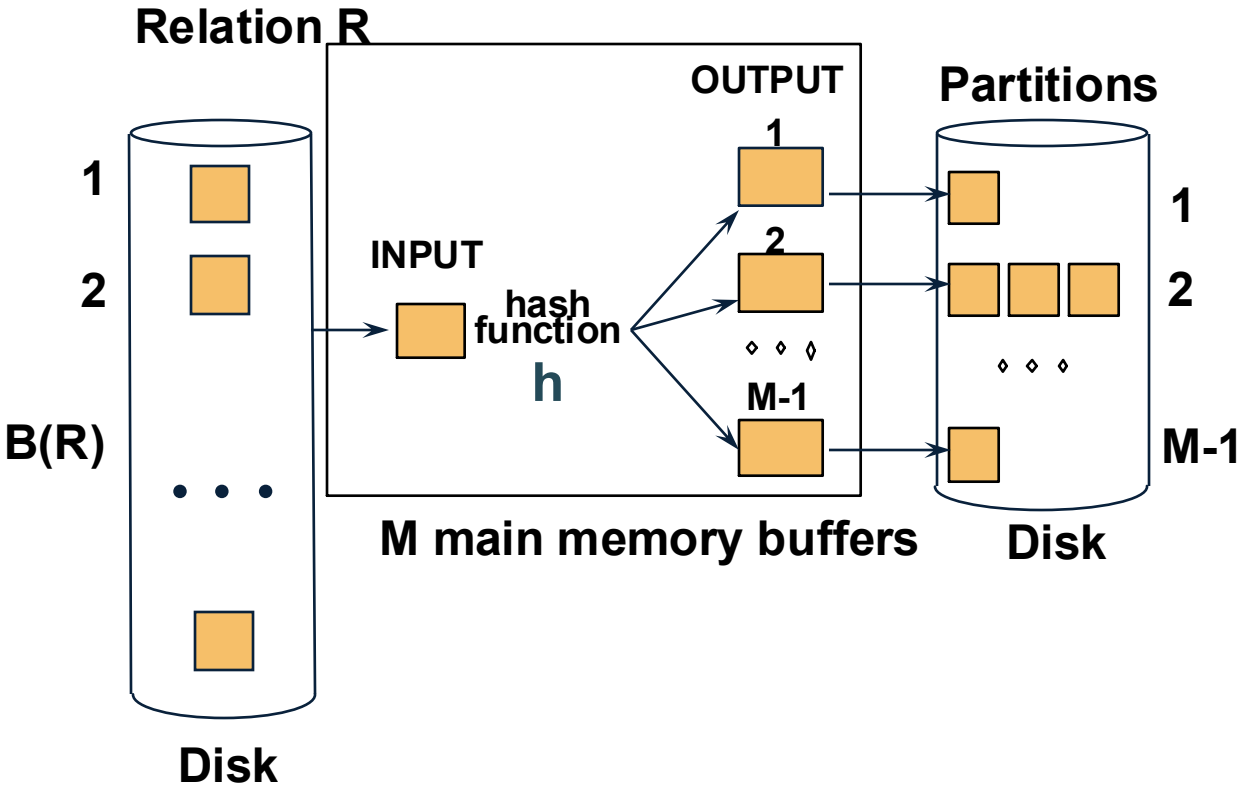
Step 1: Hash-partition

- Partition R into buckets, on disk



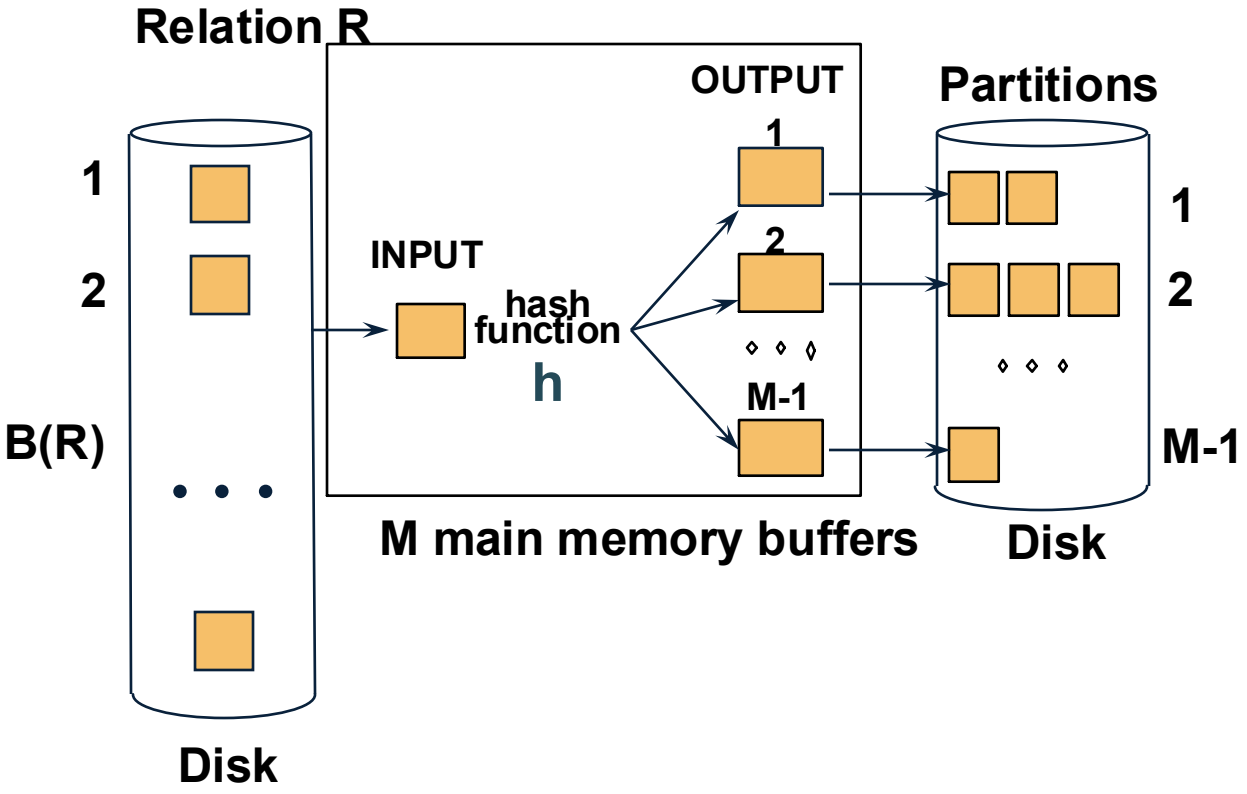
Step 1: Hash-partition

- Partition R into buckets, on disk



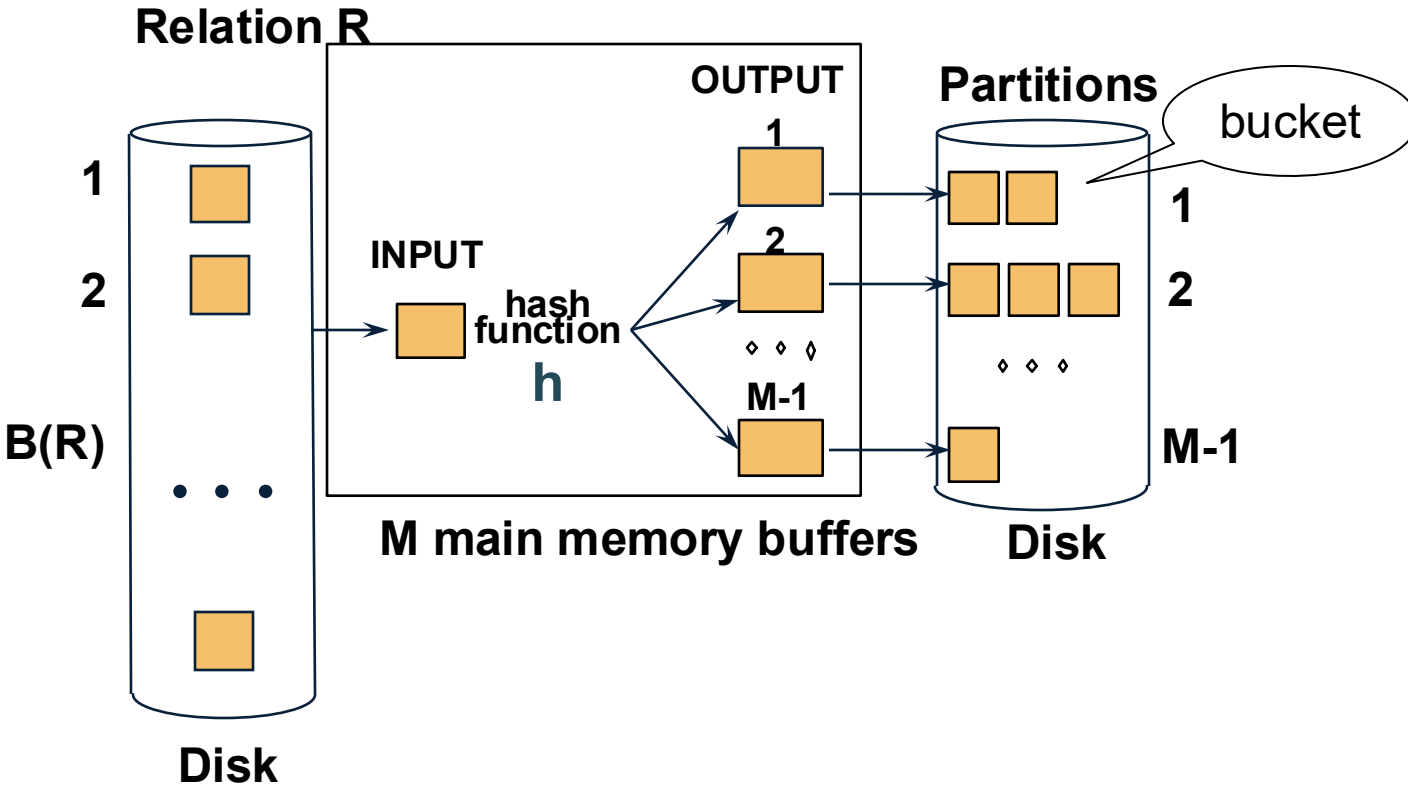
Step 1: Hash-partition

- Partition R into buckets, on disk



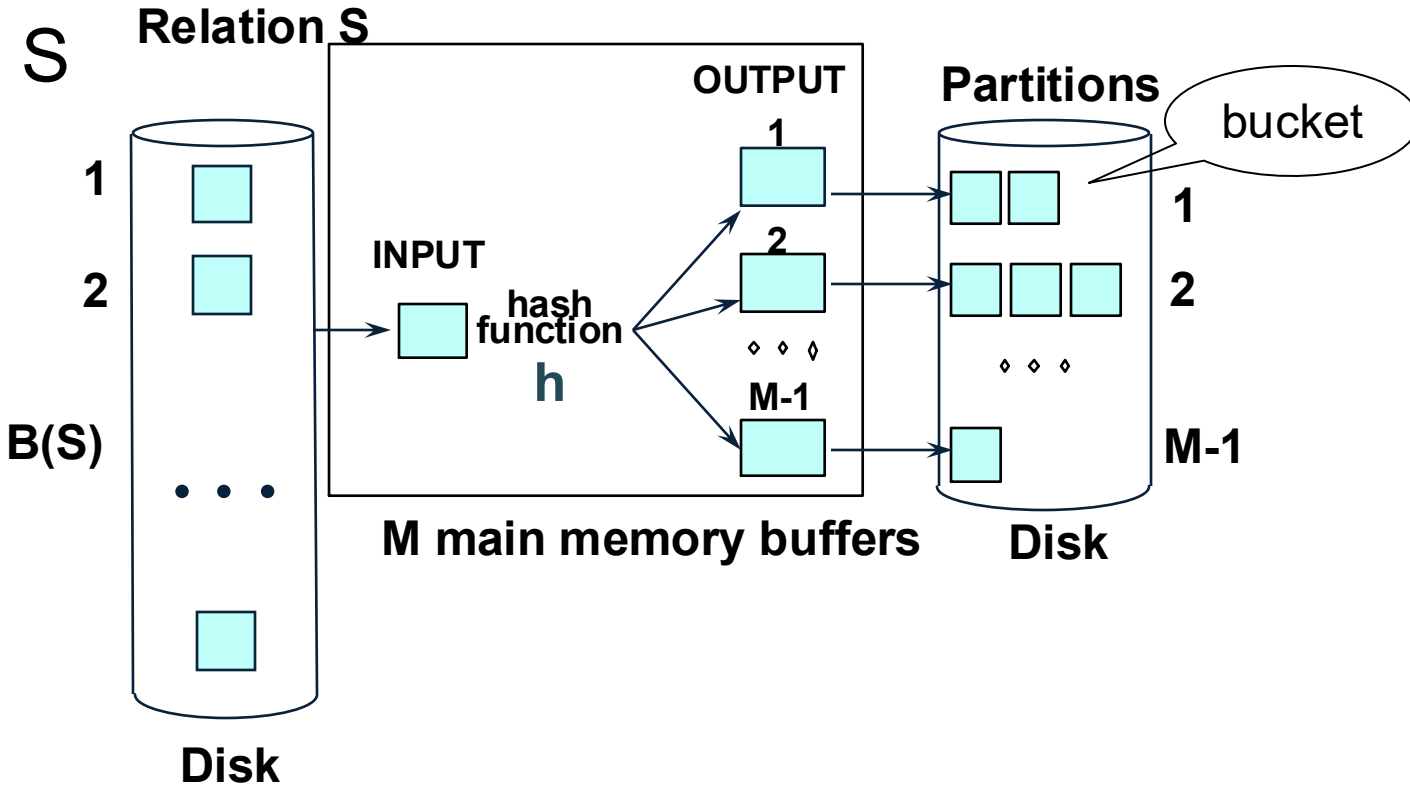
Step 1: Hash-partition

- Partition R into buckets, on disk



Step 1: Hash-partition

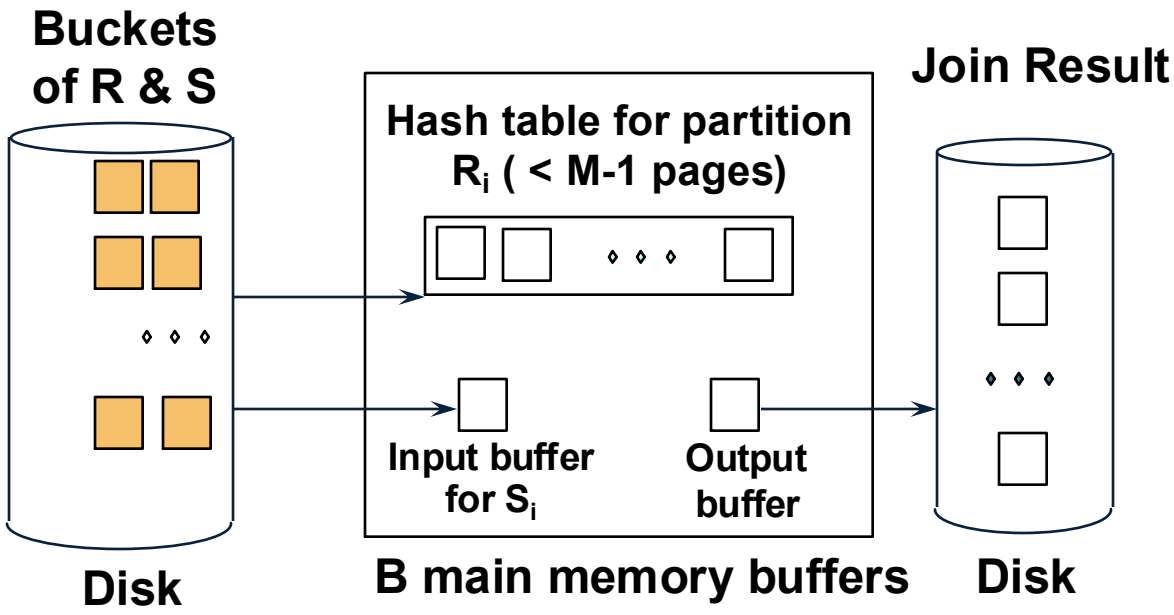
- Partition R into buckets, on disk
- Partition S



Step 2: Join Buckets

R ⋈ S

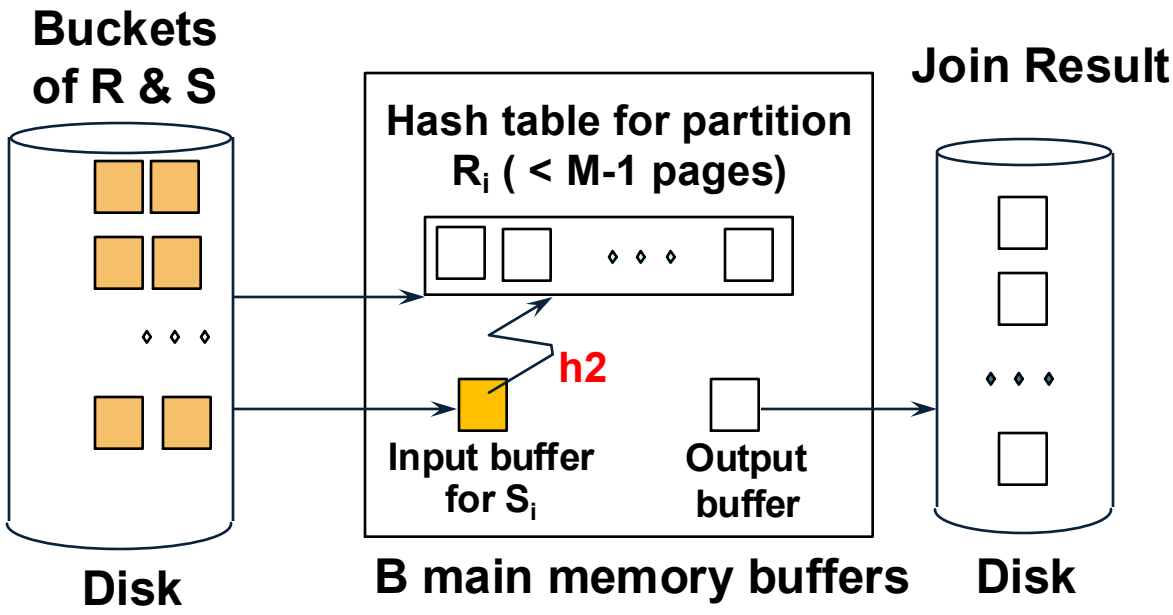
- Read one R-bucket; hash-partition it using h2 (≠ h)



Step 2: Join Buckets

R ⋈ S

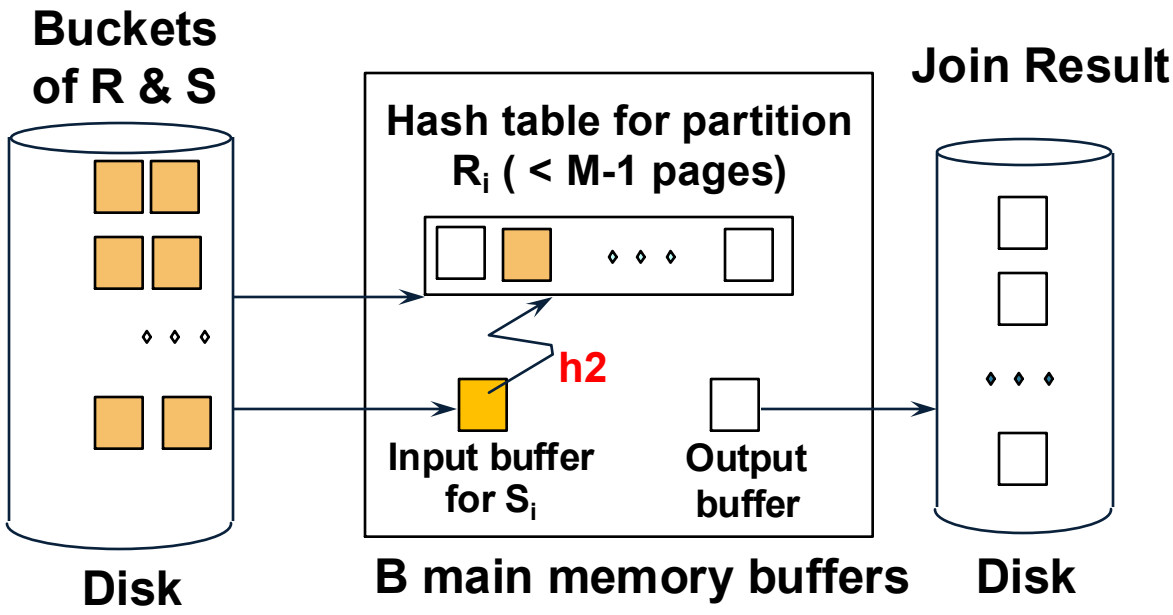
- Read one R-bucket; hash-partition it using h2 (≠ h)



Step 2: Join Buckets

R ⋈ S

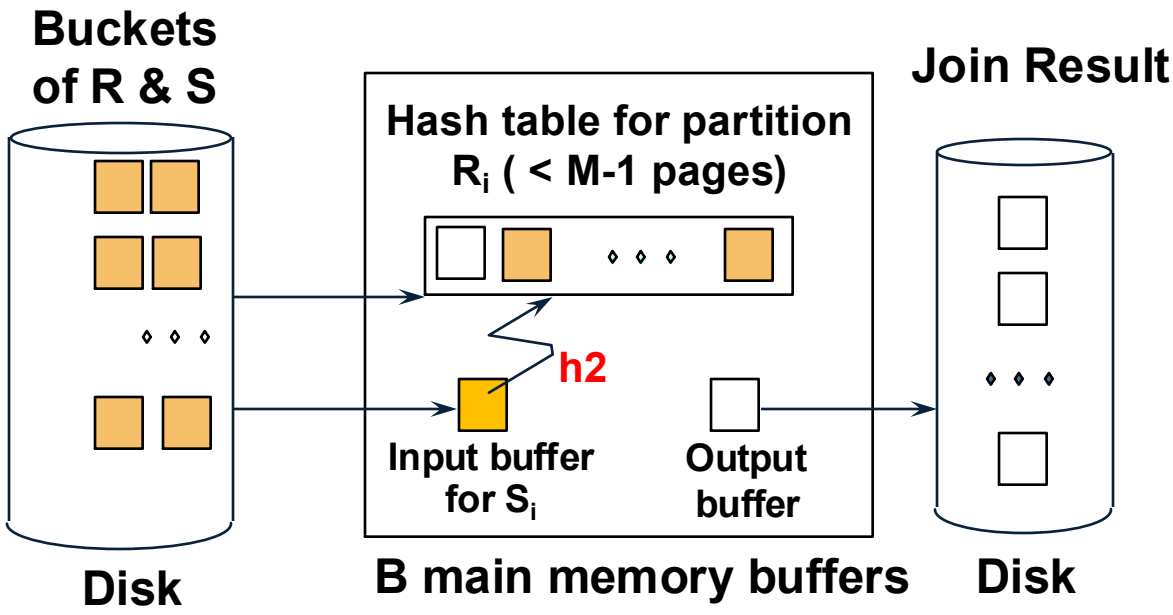
- Read one R-bucket; hash-partition it using h2 (≠ h)



Step 2: Join Buckets

R ⋈ S

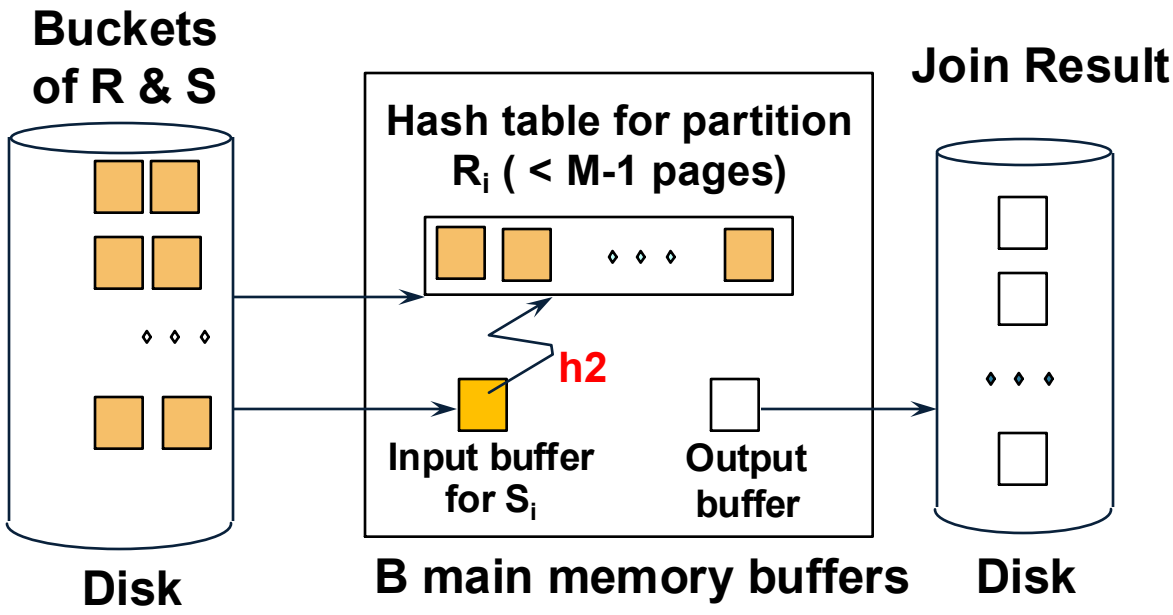
- Read one R-bucket; hash-partition it using h2 (≠ h)



Step 2: Join Buckets

R ⋈ S

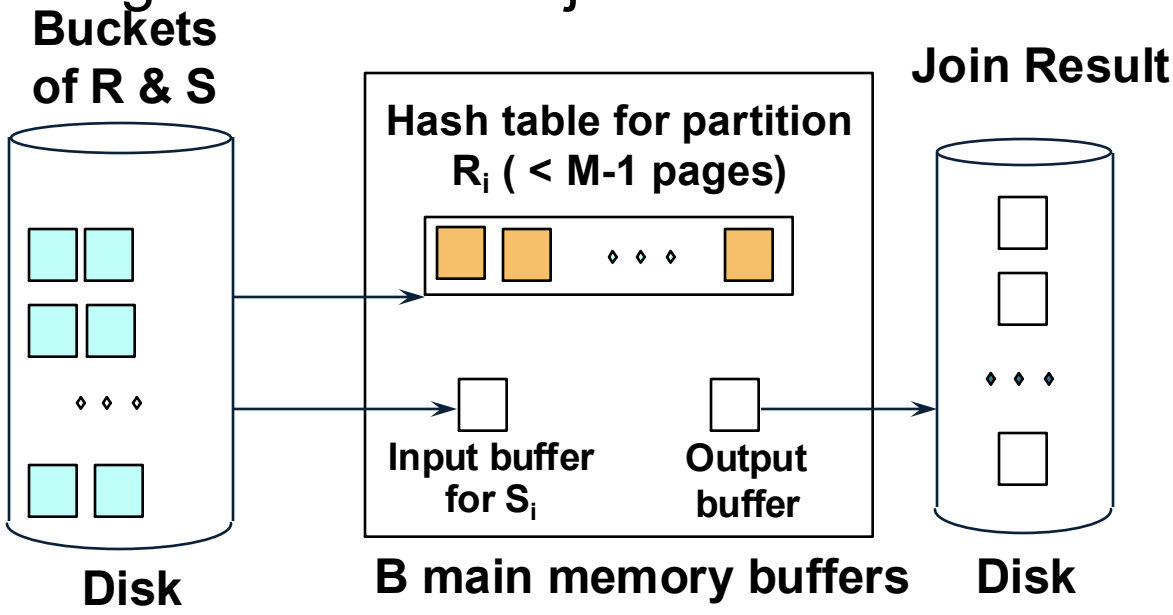
- Read one R-bucket; hash-partition it using h2 (≠ h)



Step 2: Join Buckets

R ⋈ S

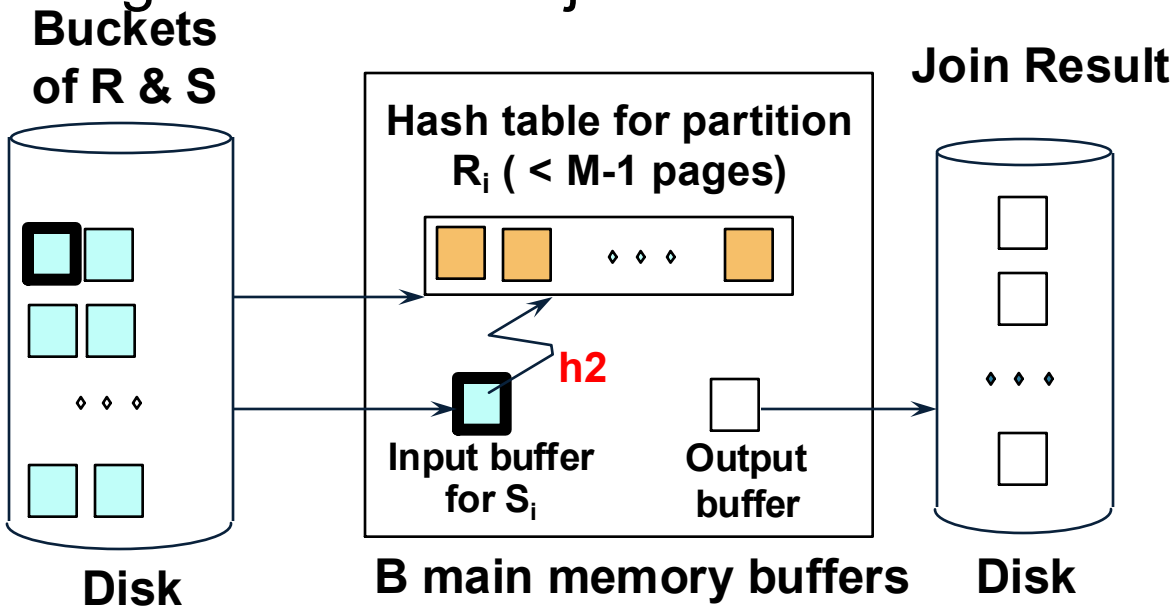
- Read one R-bucket; hash-partition it using h2 (≠ h)
- Scan corresponding S bucket and join



Step 2: Join Buckets

R ⋈ S

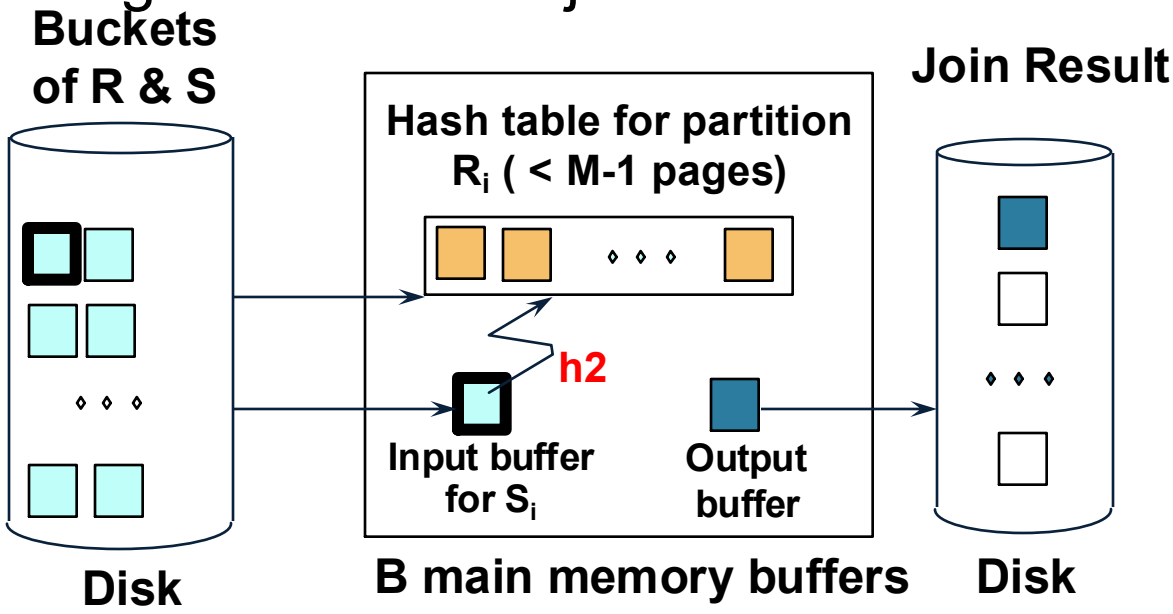
- Read one R-bucket; hash-partition it using h2 (≠ h)
- Scan corresponding S bucket and join



Step 2: Join Buckets

R ⋈ S

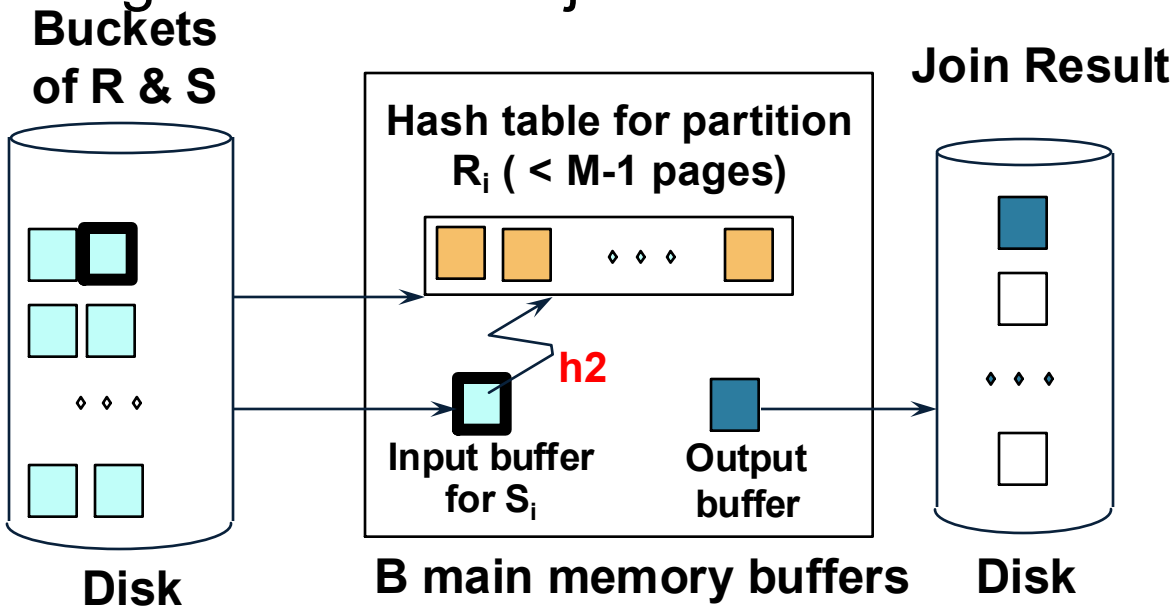
- Read one R-bucket; hash-partition it using h2 (≠ h)
- Scan corresponding S bucket and join



Step 2: Join Buckets

R ⋈ S

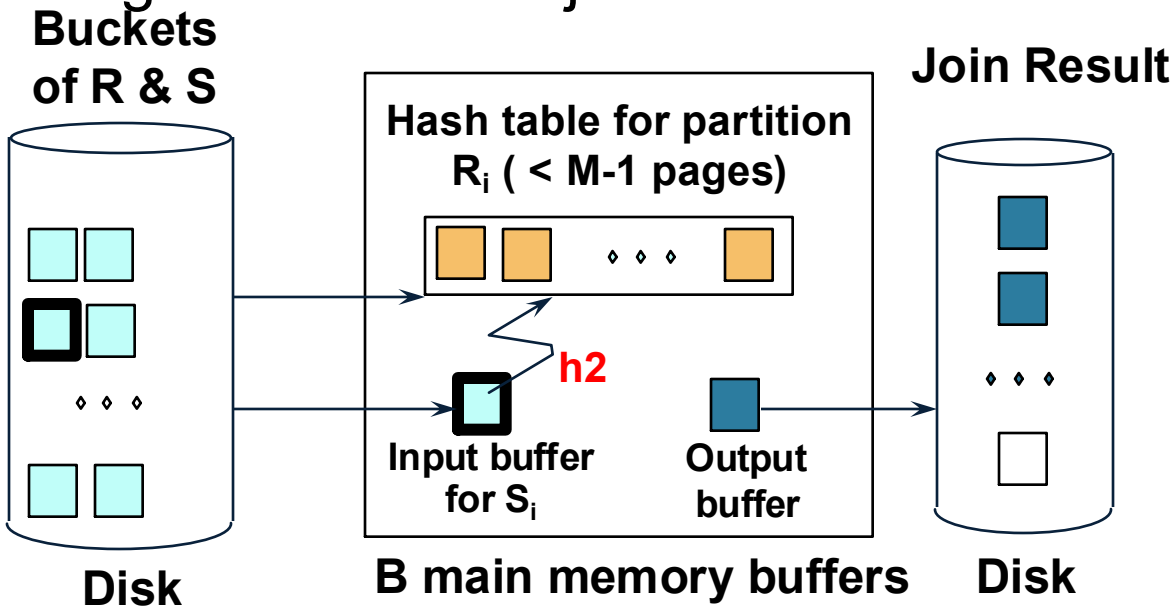
- Read one R-bucket; hash-partition it using h2 (≠ h)
- Scan corresponding S bucket and join



Step 2: Join Buckets

R ⋈ S

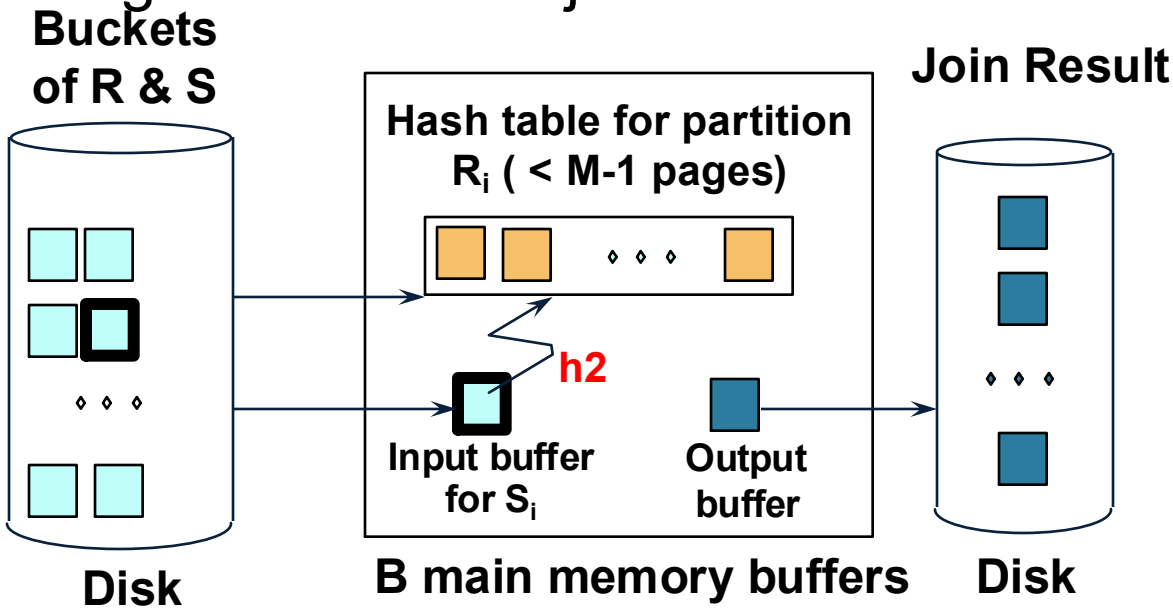
- Read one R-bucket; hash-partition it using h2 (≠ h)
- Scan corresponding S bucket and join



Step 2: Join Buckets

R ⋈ S

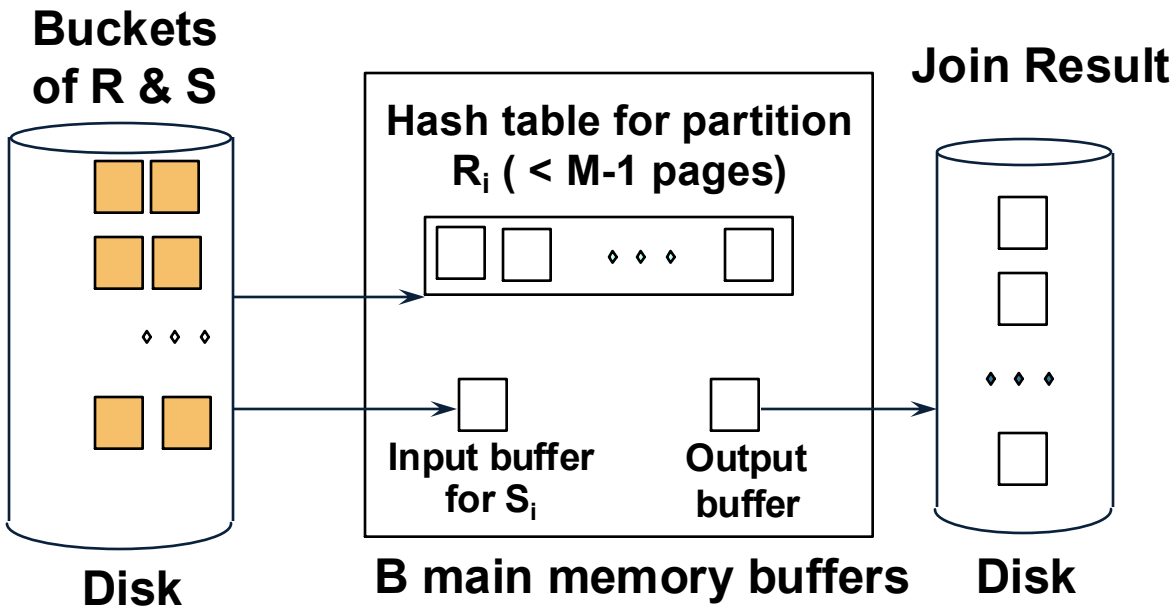
- Read one R-bucket; hash-partition it using h2 (≠ h)
- Scan corresponding S bucket and join



Step 2: Join Buckets

R ⋈ S

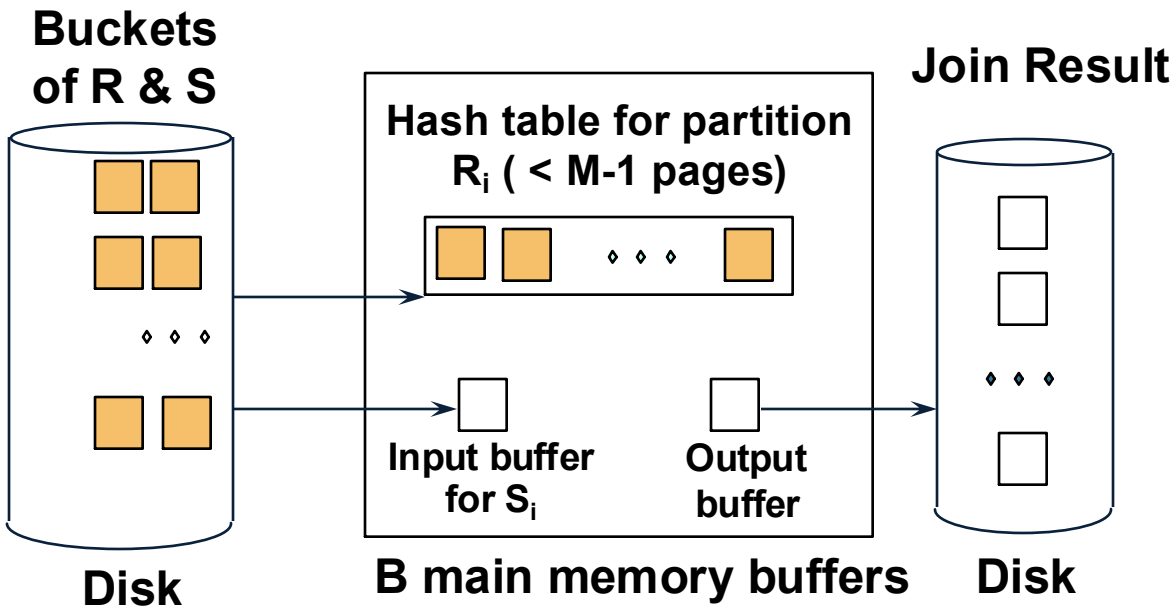
- Read the next R bucket



Step 2: Join Buckets

R ⋈ S

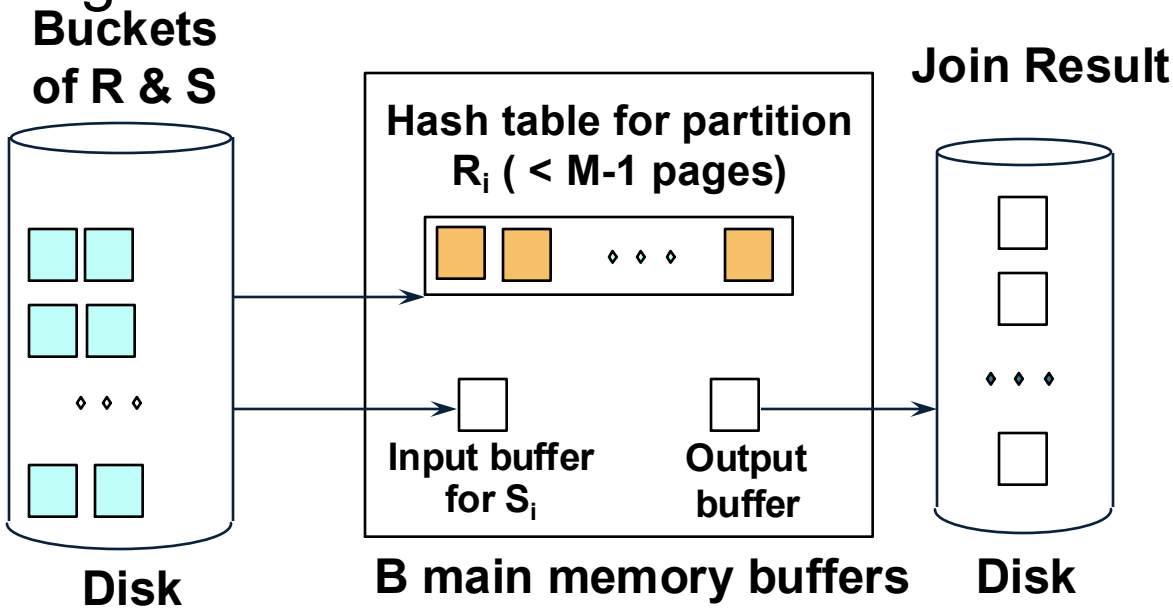
- Read the next R bucket



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

Partitioned Hash Join

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

- Problem: abrupt cost increase from Hash Join (B(R)+B(S)) to Partition Hash Join (3 times larger)
- Grace join: smoothers this transition

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

Snowflake

Snowflake

- Is an SaaS – what is this?

Snowflake

- Is an SaaS – what is this?
- SaaS = software as a service
- Other cloud services:
  - Platform as a service (PaaS): Amazon's EC
  - Infrastructure as a service (virtual machines)
  - Software as a Service
  - Function as a Service: Amazon's Lambda

Snowflake

Snowflake's Data Storage:

In class:

- S3:PUT/GET/DELETE
- Table → horizontal partition in *files*
- Blobs+PAX
- Temp storage→S3

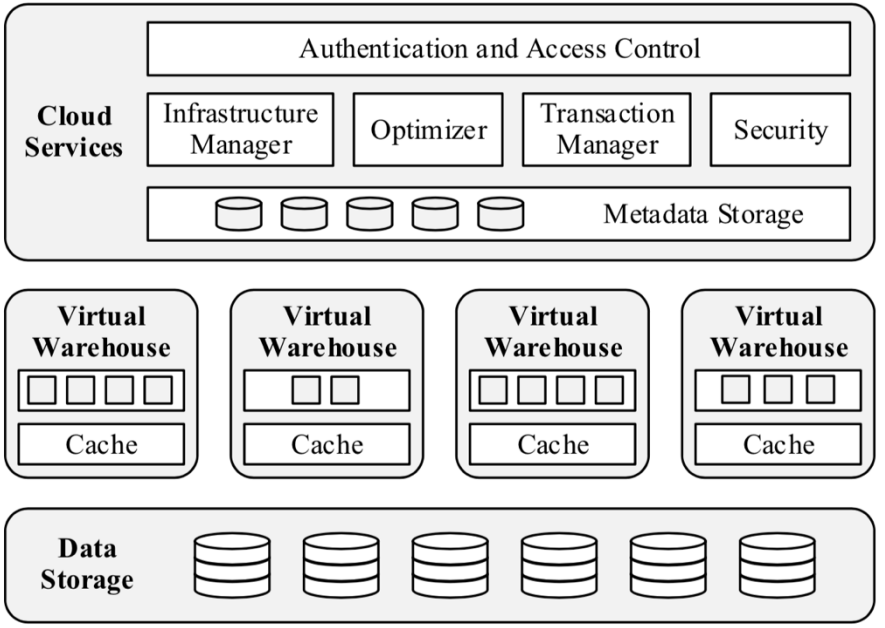


Figure 1: Multi-Cluster, Shared Data Architecture

Snowflake

- Describe Elasticity in Snowflake

- Describe failure handling in Snowflake

Snowflake

- Describe Elasticity in Snowflake
  - Virtual Warehouse (VW) serves one user
  - T-Shirt sizes: X-Small … XX-Large
  - Small query may run on subset of VW
- Describe failure handling in Snowflake

Snowflake

- Describe Elasticity in Snowflake
  - Virtual Warehouse (VW) serves one user
  - T-Shirt sizes: X-Small … XX-Large
  - Small query may run on subset of VW
- Describe failure handling in Snowflake
  - Restart the query
  - No partial retries (like MapReduce or Spark)

Snowflake

- Column-oriented (in class)
- Vectorized (“tuple batches” – in class)
- Push-based (in class)

Snowflake

- What does Snowflake use instead of indexes?

Snowflake

- What does Snowflake use instead of indexes?

- “Pruning”: for each file (recall: this is a horizontal partition of a table) and each attribute, it stores the min/max values in that column in that file; may skip files when not needed.