Discussion 11

Parallel Query Processing

Announcements

Vitamin 11 (PQP) due **Monday, April 15 at 11:59pm**

Project 4 Part 2 (Concurrency) due **April 10 at 11:59pm**

Agenda

I. Types of Parallelism
II. Partitioning Schemes
III. Parallel Query Operators (Hashing, Sorting, Joins, etc.)
IV. Aggregation
V. Misc

Types of Parallelism

Types of Query Parallelism

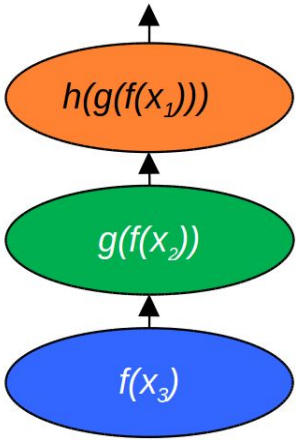
- **Inter-query parallelism** works using queries as the unit of parallelism
  - e.g. running 5 queries in parallel
  - covered later in the course
- **Intra-query parallelism** works within a single query
  - e.g. scanning two relations for a query in parallel
  - focus of this section

Types of Intra-query Parallelism

- **Inter-operator parallelism** works using operators as the unit of parallelism
  - e.g. running two scans in parallel
- **Intra-operator parallelism** works within a single operator
  - e.g. speeding up a single scan by having multiple threads or machines read different parts at the same time

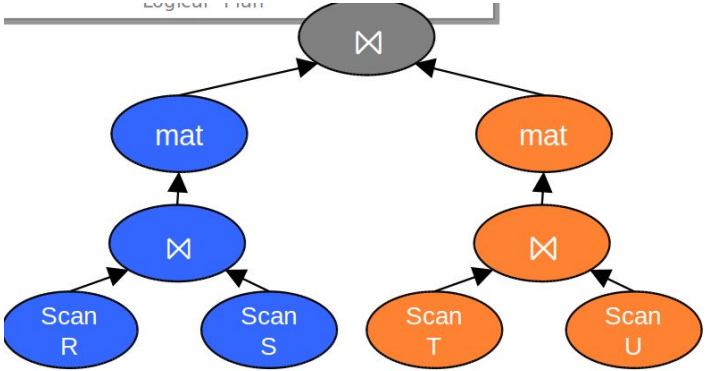
Inter-operator Parallelism

- **Pipeline parallelism** is inter-operator parallelism on operators in a pipeline
  - e.g. projection applied on selection applied on scan: selection depends on output of scan, and projection depends on output of selection



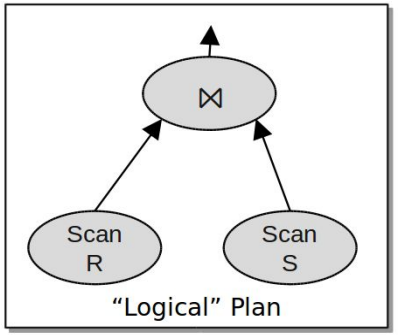
Inter-operator Parallelism

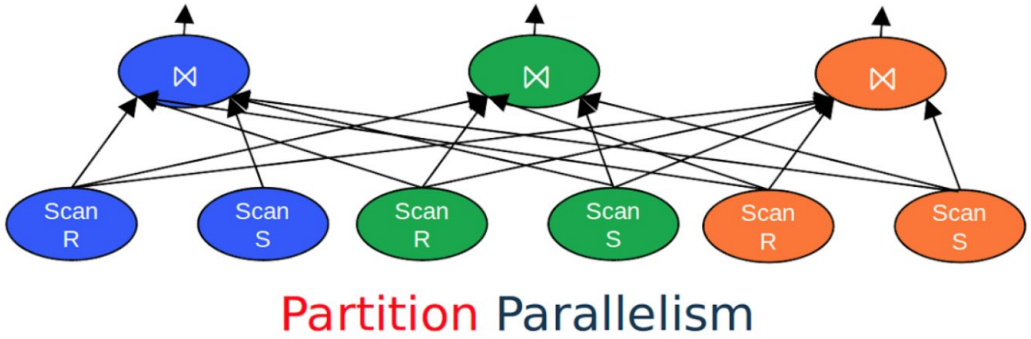
- **Bushy tree parallelism** is inter-operator parallelism on operators that don't depend on each other
  - e.g. (A ⋈ B) ⋈ (C ⋈ D) - the two parenthesized joins don't depend on each other



Intra-operator Parallelism

- **Partition parallelism** is intra-operator parallelism that works by partitioning the data and operating on partitions in parallel





Worksheet #1

What is the difference between inter- and intra- query parallelism?

Worksheet #1

What is the difference between inter- and intra- query parallelism?

Inter-query parallelism operates between multiple queries, rather than within a single query, whereas intra-query parallelism operates within a single query (parallelism of the operators that make up the query).

Partitioning Schemes

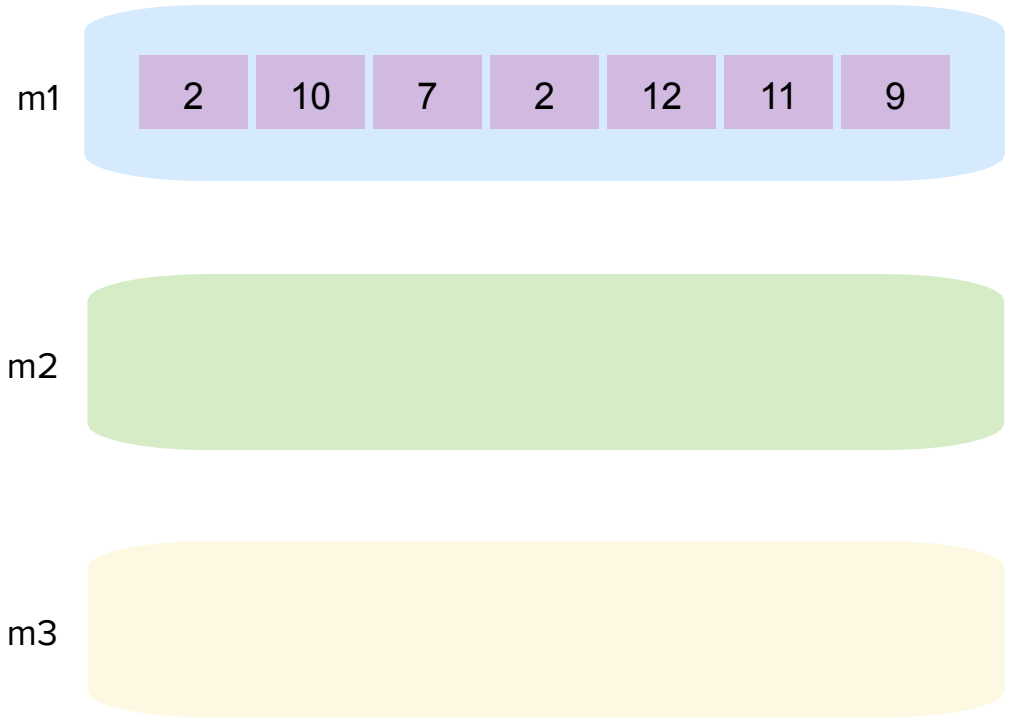
Network cost

- **Network cost:** the amount of data we need to send over the network (from one machine to another) to perform an operation
- **Units:** usually KB
- Unlike I/O cost, we do not need to send data over the network in units of pages - we can send, for example, one tuple's worth of data across the network.

Partitioning data

- **Range partitioning** on a key divides data based on which range the key belongs to
- **Hash partitioning** divides data based on a hash function
- **Round robin partitioning** just cycles through the partitions in order as data come in (*not ordered based on a key*)

Example: Range Partitioning

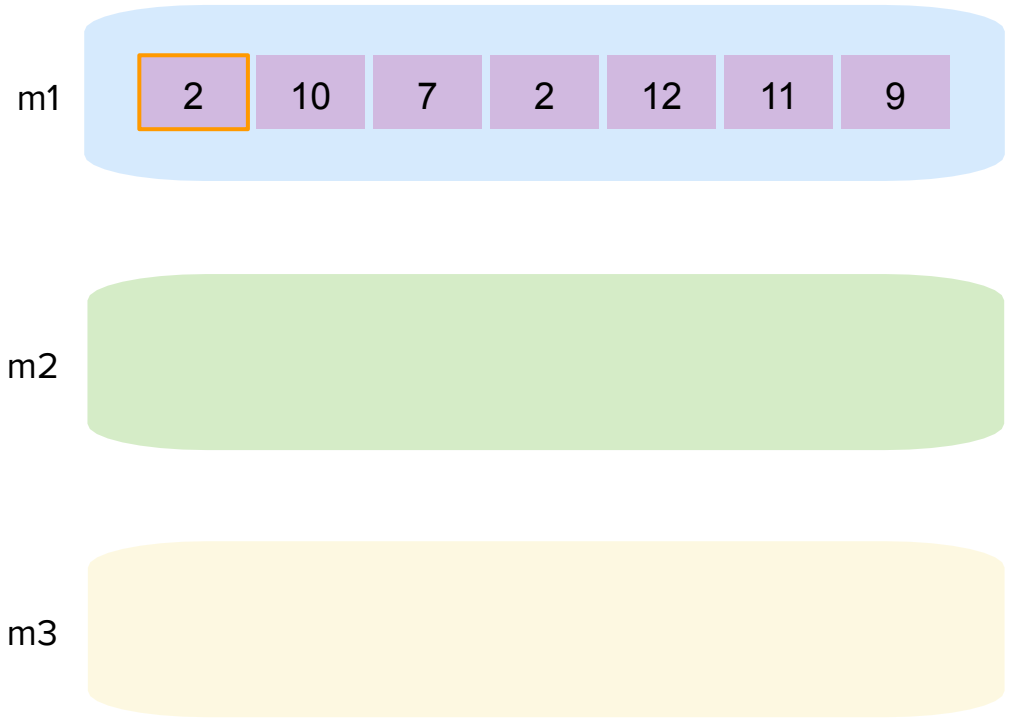


Assign data to 3 partitions using **range** partitioning
[0, 4] → 1
[5, 9] → 2
[10, 14] → 3

Assume the size of each tuple is X KB.

Total network cost: 0 KB

Example: Range Partitioning



Assign data to 3 partitions using **range** partitioning

[0, 4] → 1

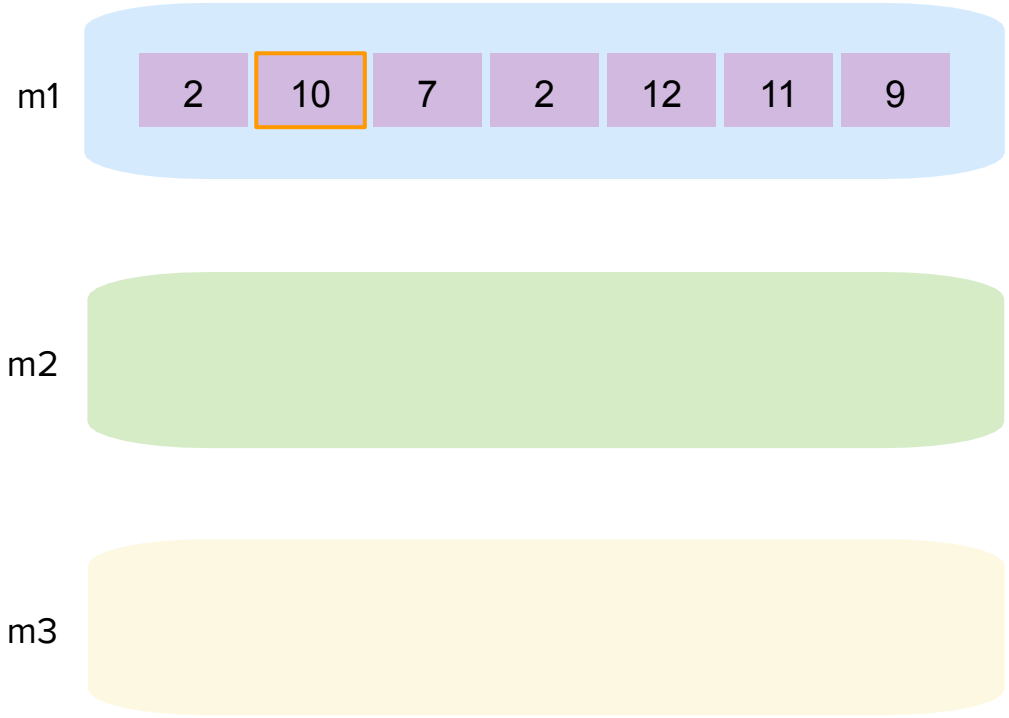
[5, 9] → 2

[10, 14] → 3

Assume the size of each tuple is X KB.

Total network cost: 0 KB

Example: Range Partitioning



Assign data to 3 partitions using **range** partitioning

[0, 4] → 1

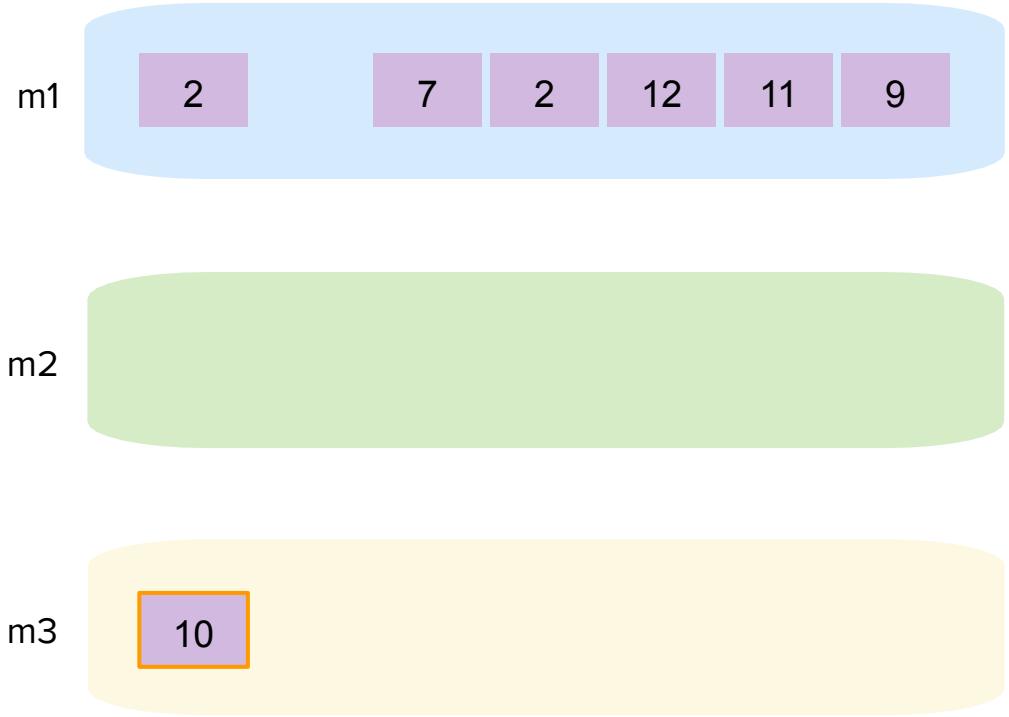
[5, 9] → 2

[10, 14] → 3

Assume the size of each tuple is X KB.

Total network cost: 0 KB

Example: Range Partitioning

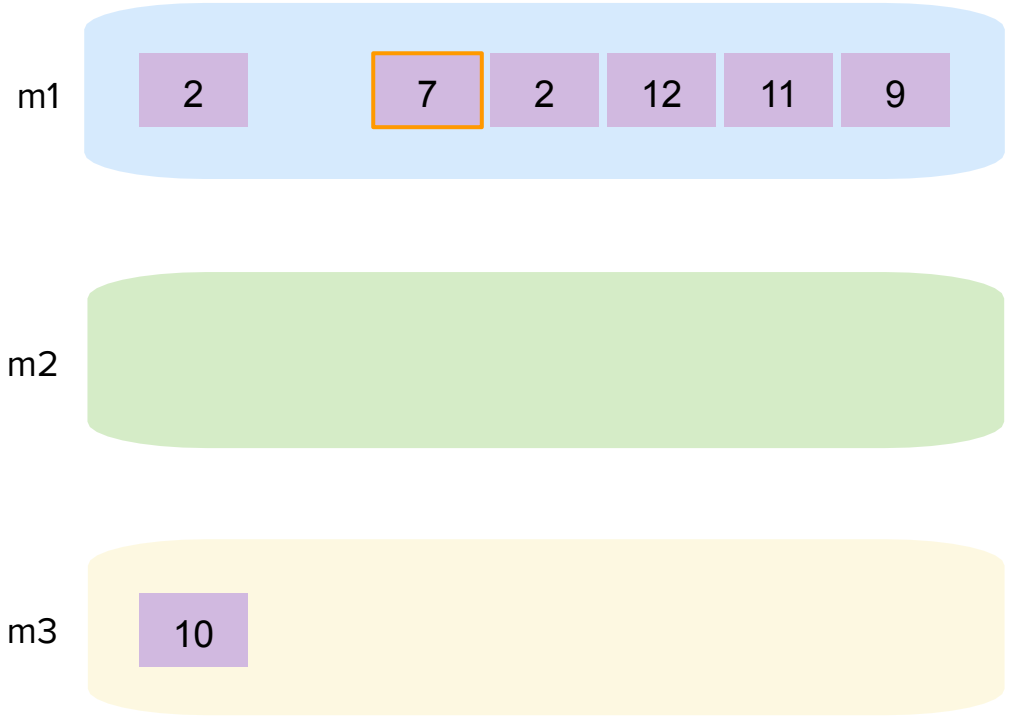


Assign data to 3 partitions using **range** partitioning
[0, 4] → 1
[5, 9] → 2
[10, 14] → 3

Assume the size of each tuple is X KB.

Total network cost: X KB

Example: Range Partitioning



Assign data to 3 partitions using **range** partitioning

[0, 4] → 1

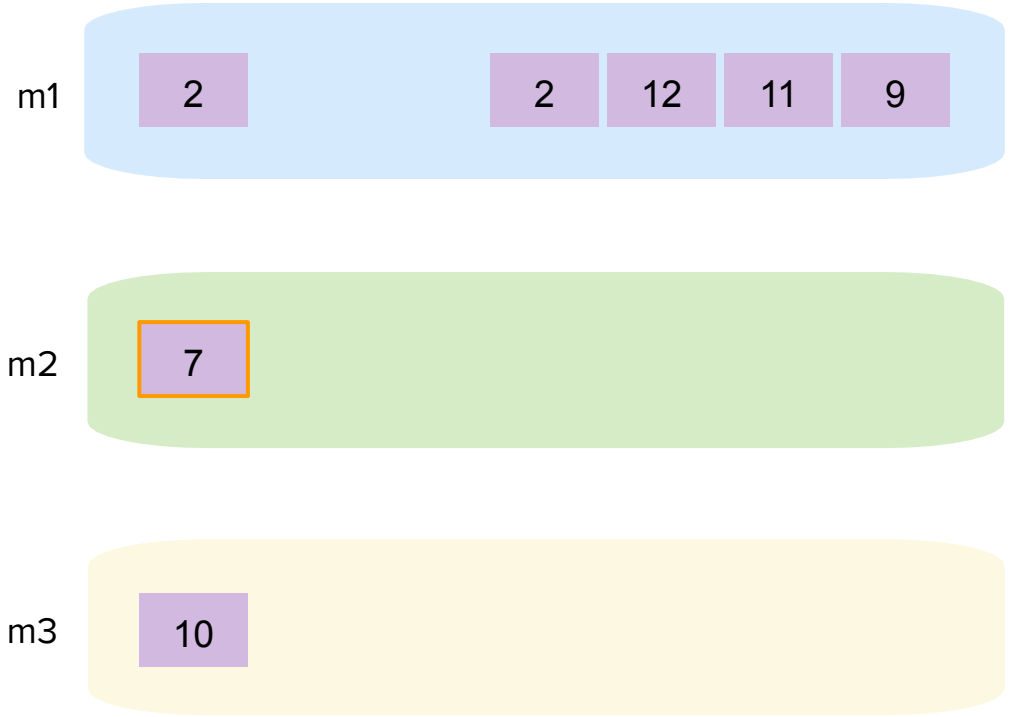
[5, 9] → 2

[10, 14] → 3

Assume the size of each tuple is X KB.

Total network cost: X KB

Example: Range Partitioning



Assign data to 3 partitions using **range** partitioning

[0, 4] → 1

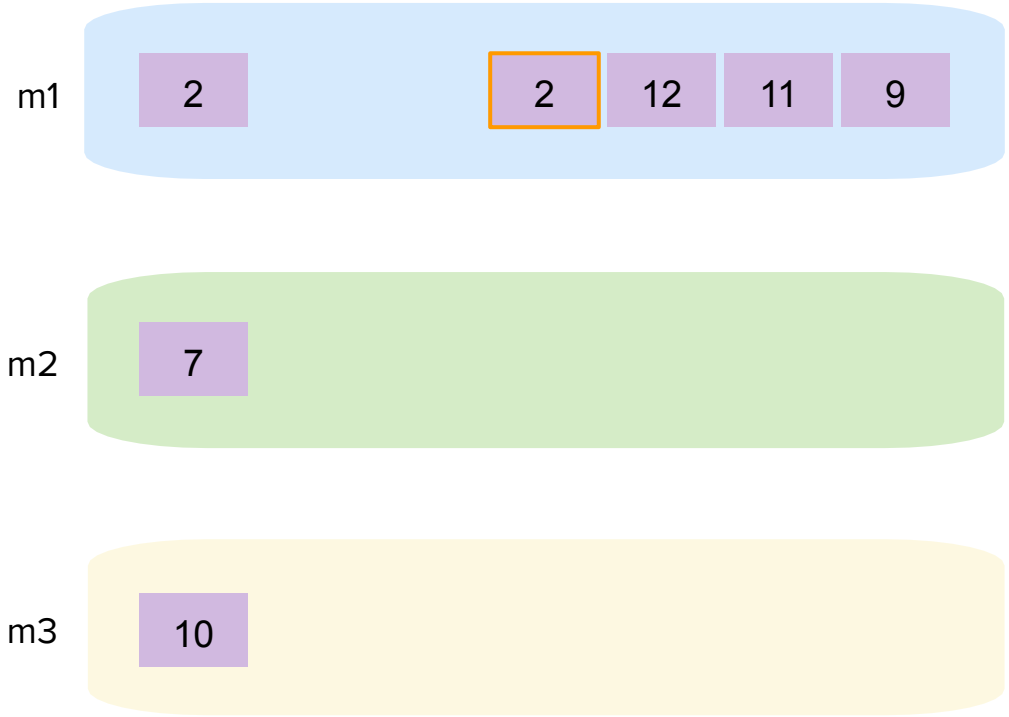
[5, 9] → 2

[10, 14] → 3

Assume the size of each tuple is X KB.

Total network cost: 2X KB

Example: Range Partitioning



Assign data to 3 partitions using **range** partitioning

[0, 4] → 1

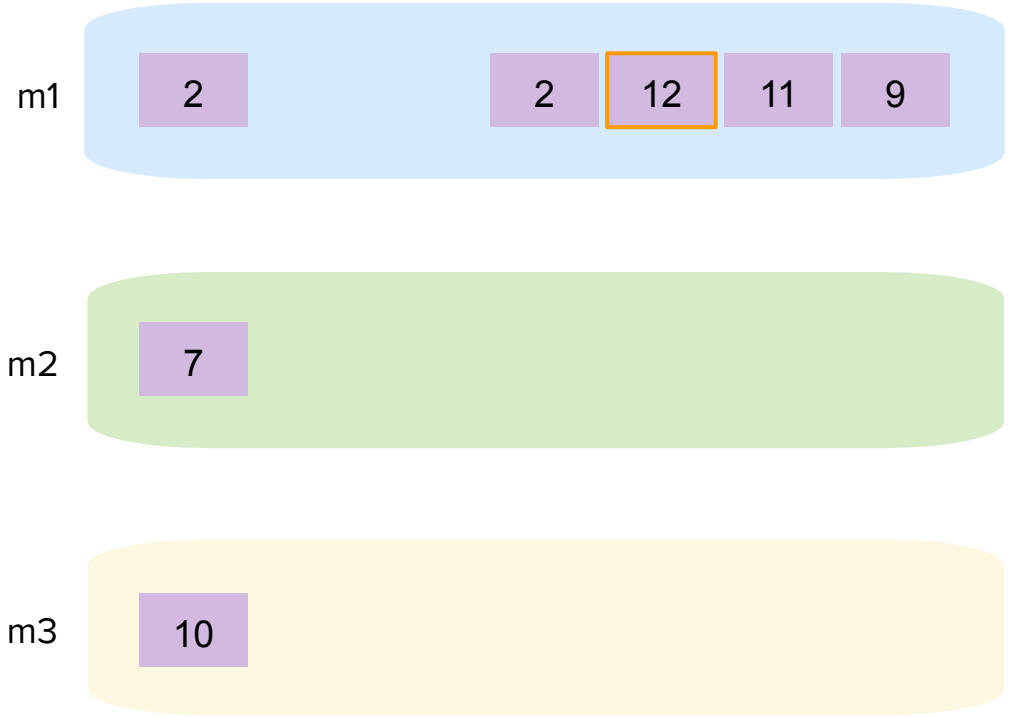
[5, 9] → 2

[10, 14] → 3

Assume the size of each tuple is X KB.

Total network cost: 2X KB

Example: Range Partitioning

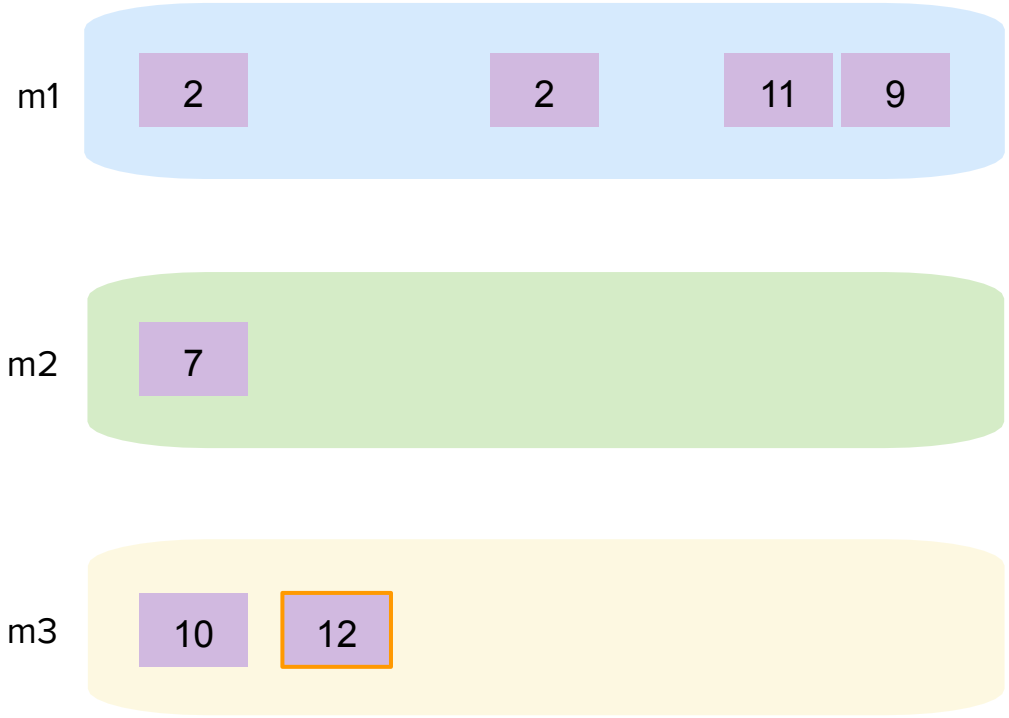


Assign data to 3 partitions using **range** partitioning
[0, 4] → 1
[5, 9] → 2
[10, 14] → 3

Assume the size of each tuple is X KB.

Total network cost: 2X KB

Example: Range Partitioning

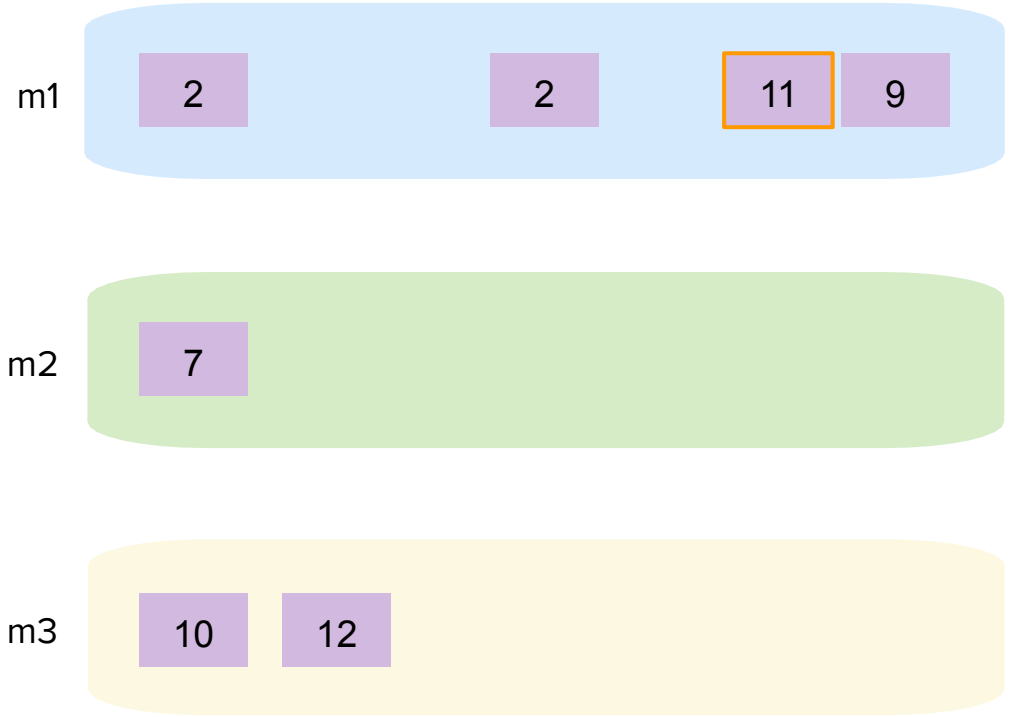


Assign data to 3 partitions using **range** partitioning
[0, 4] → 1
[5, 9] → 2
[10, 14] → 3

Assume the size of each tuple is X KB.

Total network cost: 3X KB

Example: Range Partitioning

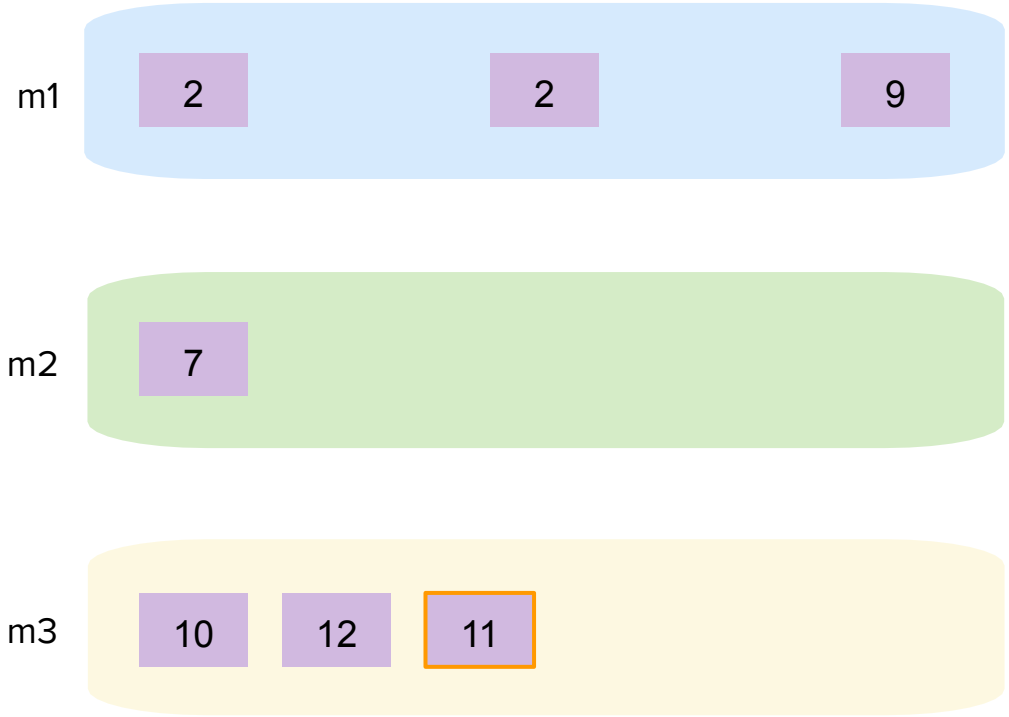


Assign data to 3 partitions using **range** partitioning
[0, 4] → 1
[5, 9] → 2
[10, 14] → 3

Assume the size of each tuple is X KB.

Total network cost: 3X KB

Example: Range Partitioning



Assign data to 3 partitions using **range** partitioning

[0, 4] → 1

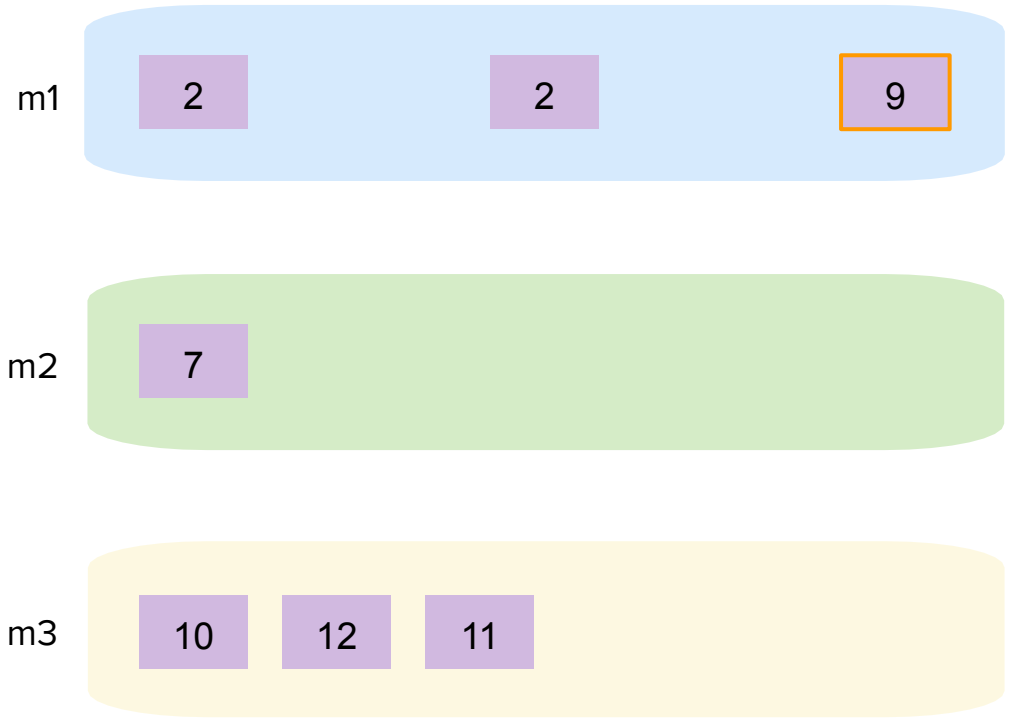
[5, 9] → 2

[10, 14] → 3

Assume the size of each tuple is X KB.

Total network cost: 4X KB

Example: Range Partitioning



Assign data to 3 partitions using **range** partitioning

[0, 4] → 1

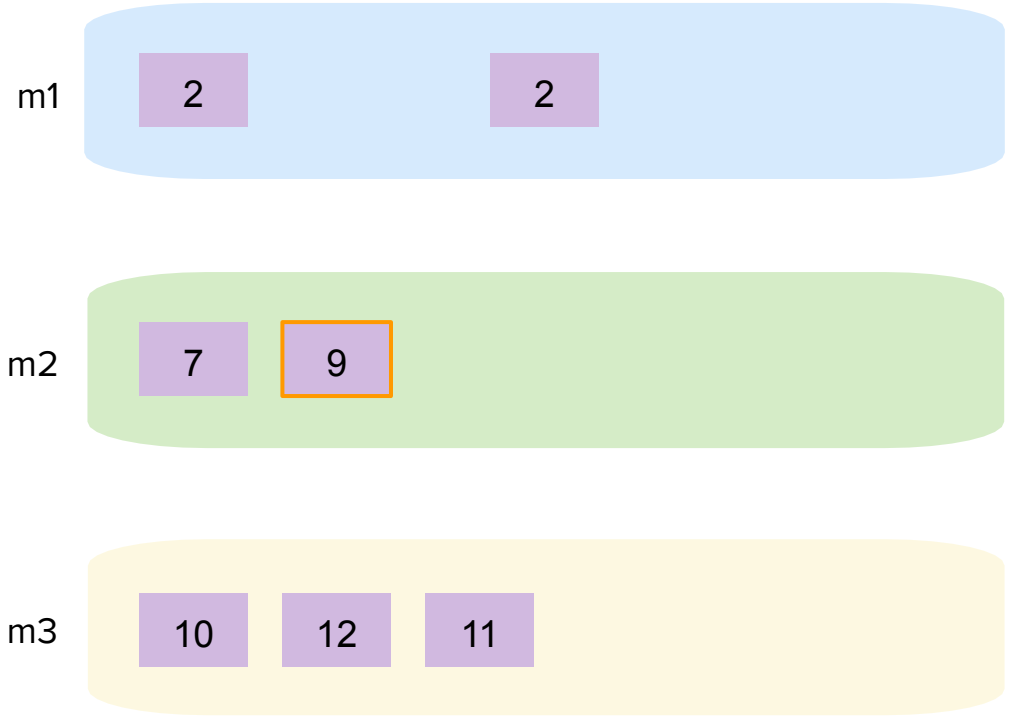
[5, 9] → 2

[10, 14] → 3

Assume the size of each tuple is X KB.

Total network cost: 4X KB

Example: Range Partitioning



Assign data to 3 partitions using **range** partitioning

[0, 4] → 1

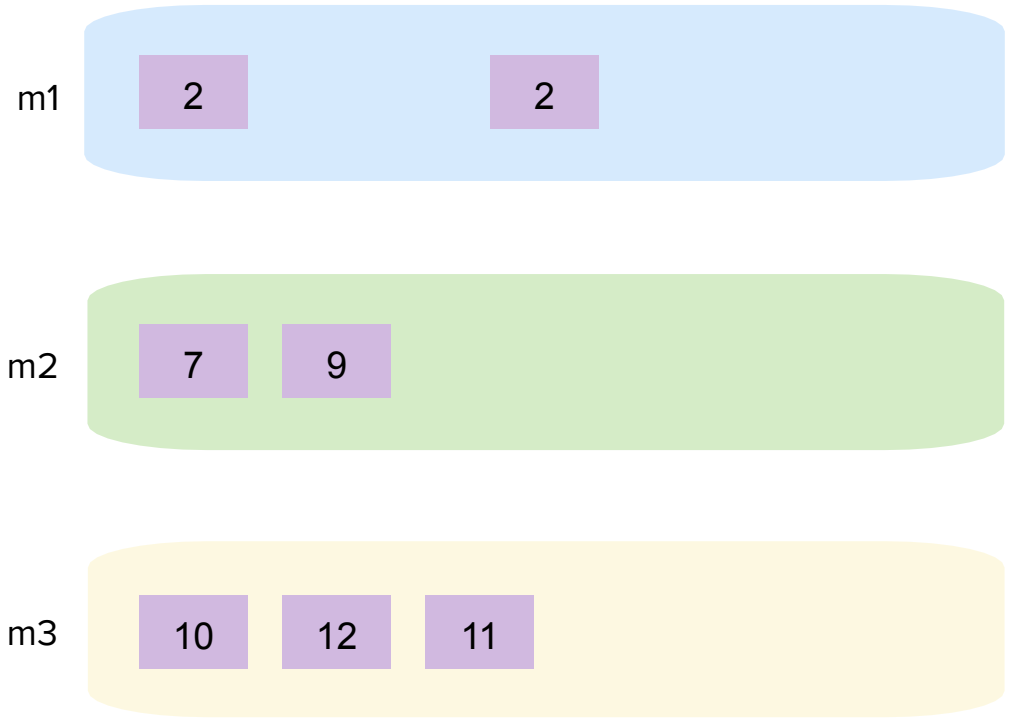
[5, 9] → 2

[10, 14] → 3

Assume the size of each tuple is X KB.

Total network cost: 5X KB

Example: Range Partitioning

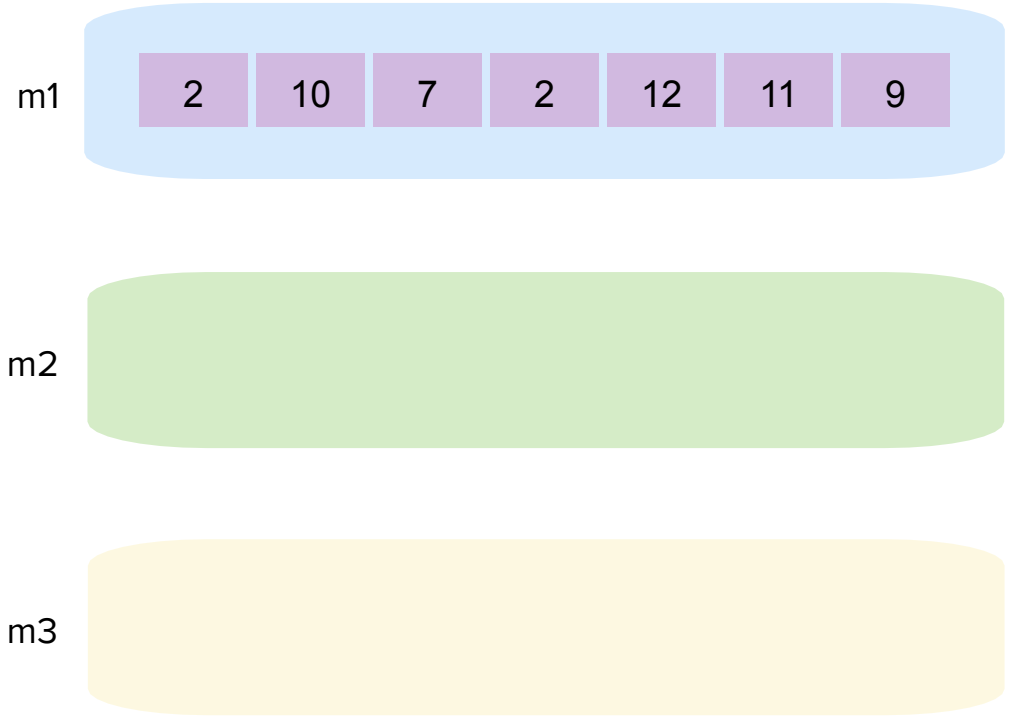


Assign data to 3 partitions using **range** partitioning
[0, 4] → 1
[5, 9] → 2
[10, 14] → 3

Assume the size of each tuple is X KB.

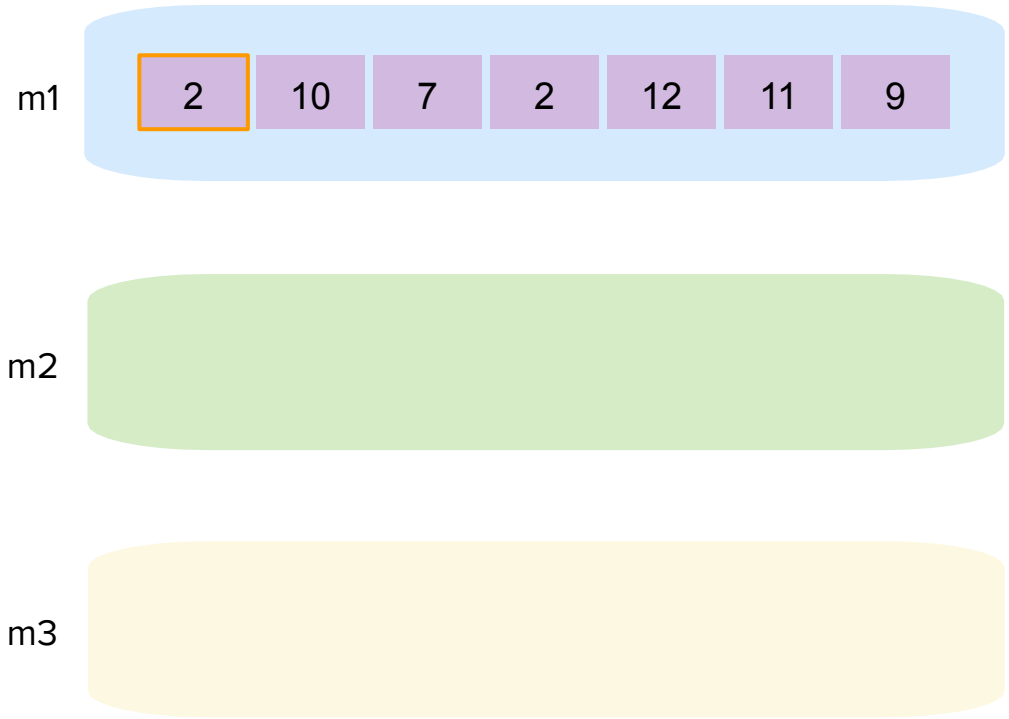
Total network cost: 5X KB

Example: Hash Partitioning



Assign data to 3 partitions using **hash** partitioning
h(k) = k % 3

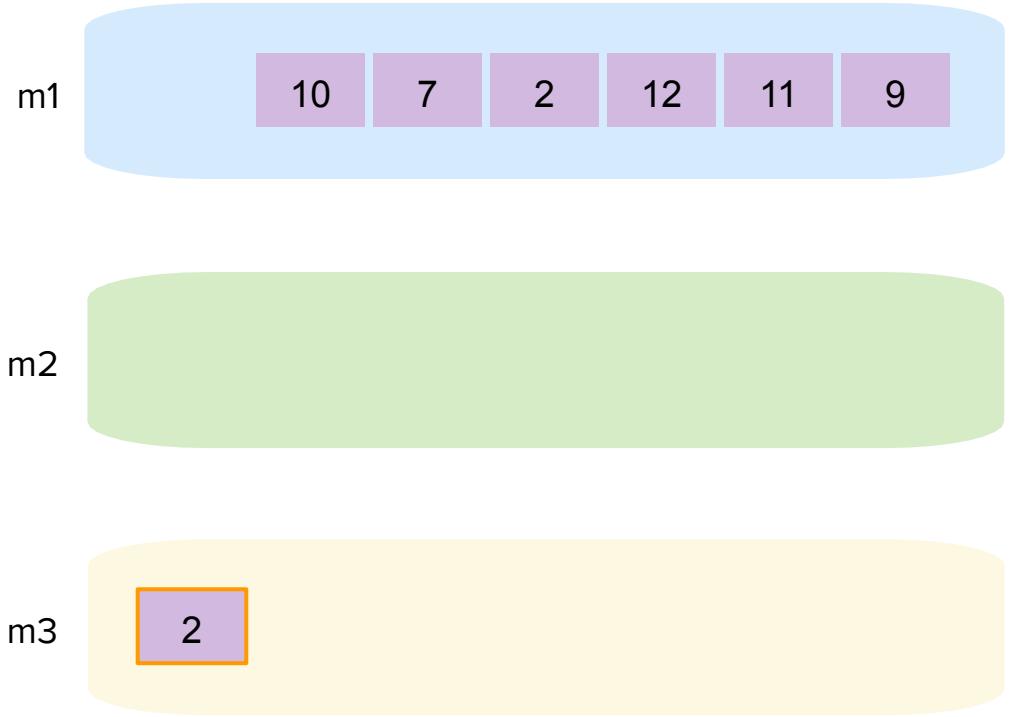
Example: Hash Partitioning



Assign data to 3 partitions using **hash** partitioning
h(k) = k % 3

2 % 3 = 2

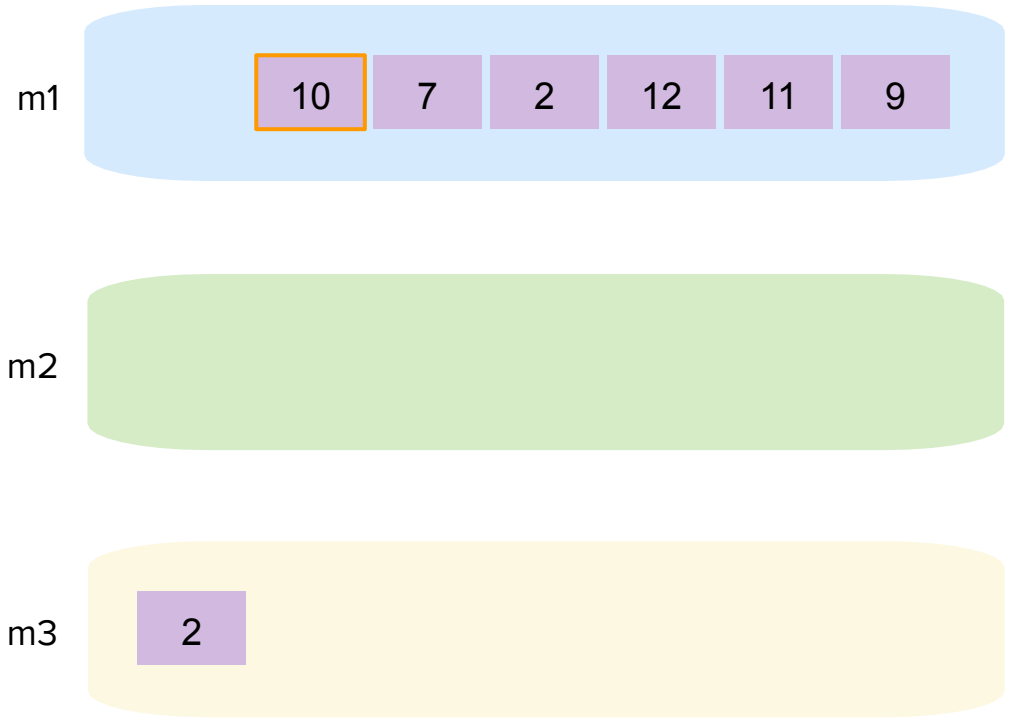
Example: Hash Partitioning



Assign data to 3 partitions using **hash** partitioning
h(k) = k % 3

2 % 3 = 2

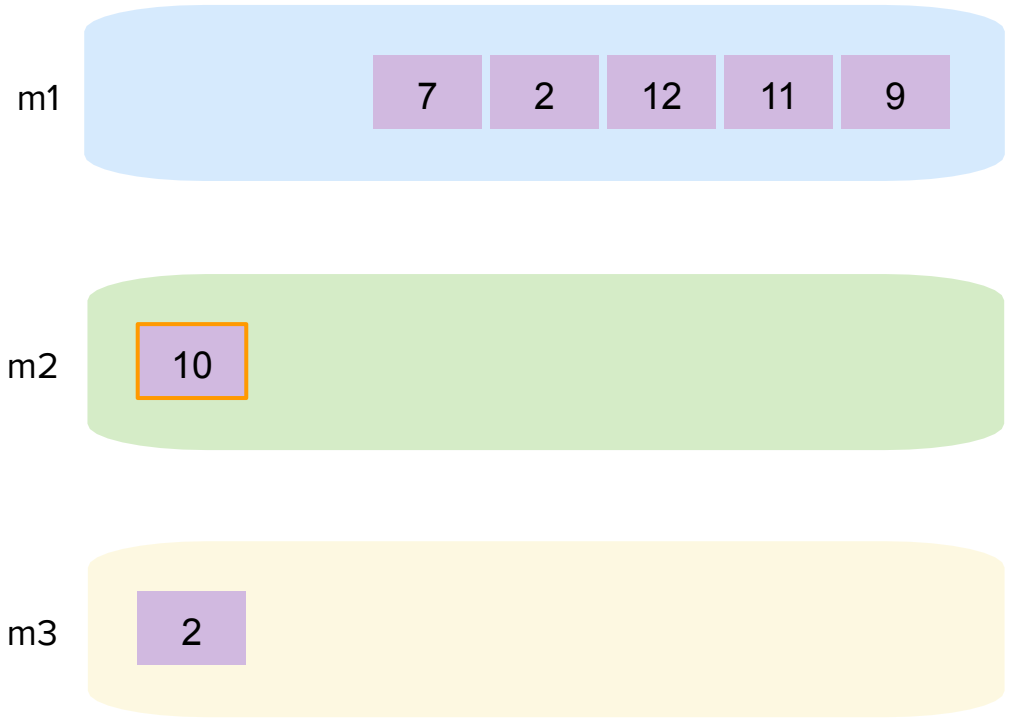
Example: Hash Partitioning



Assign data to 3 partitions using **hash** partitioning
h(k) = k % 3

10 % 3 = 1

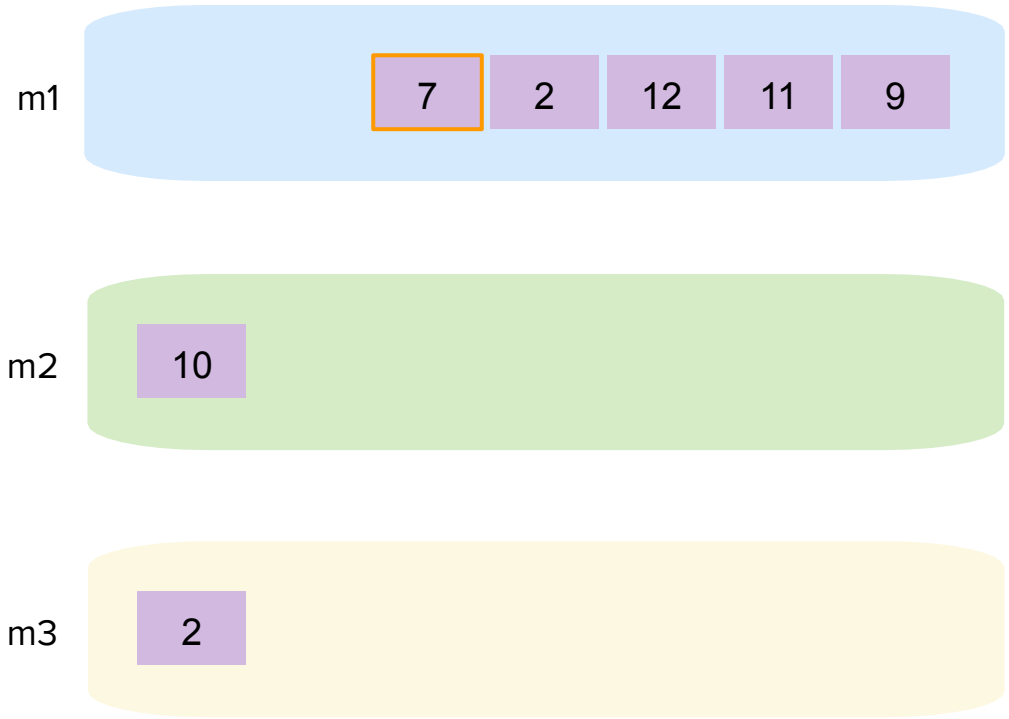
Example: Hash Partitioning



Assign data to 3 partitions using **hash** partitioning
h(k) = k % 3

10 % 3 = 1

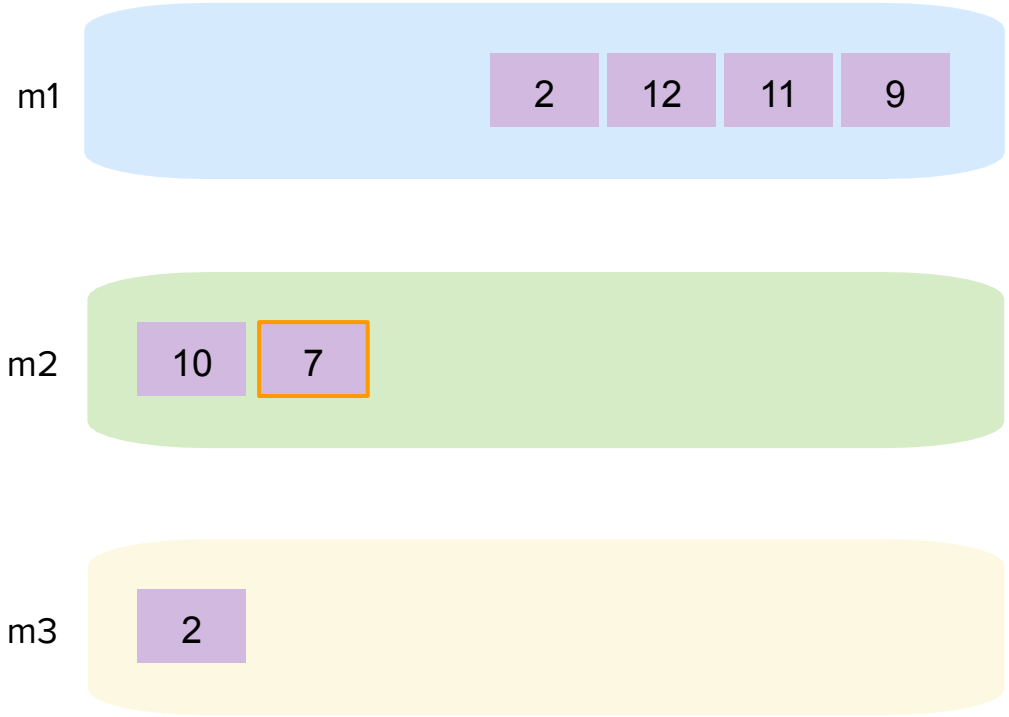
Example: Hash Partitioning



Assign data to 3 partitions using **hash** partitioning
h(k) = k % 3

7 % 3 = 1

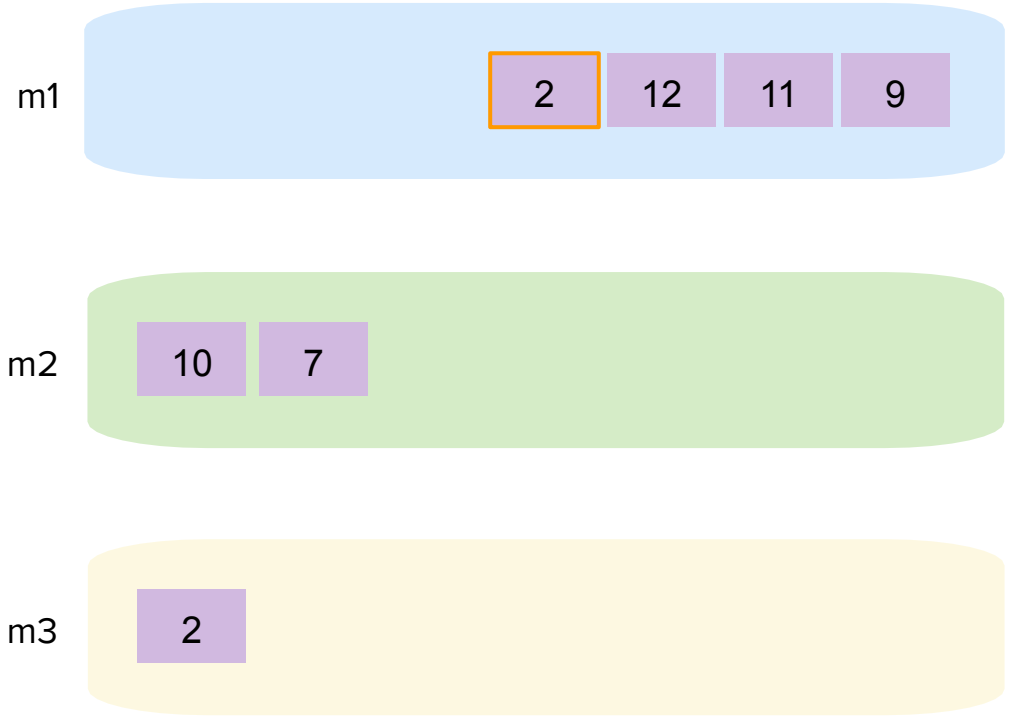
Example: Hash Partitioning



Assign data to 3 partitions using **hash** partitioning
h(k) = k % 3

7 % 3 = 1

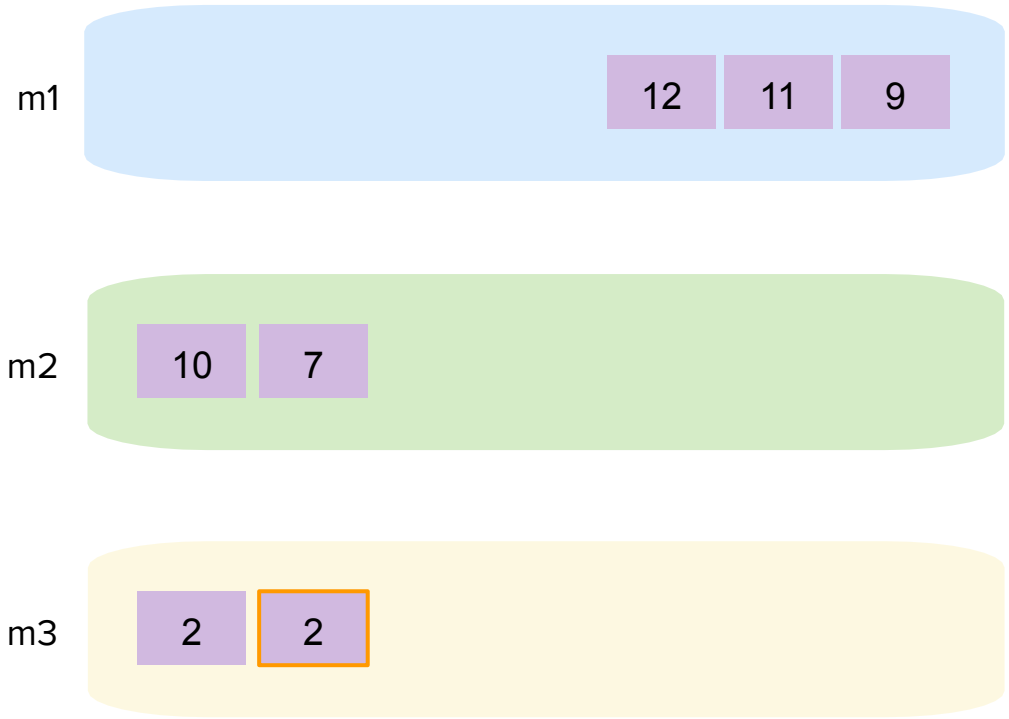
Example: Hash Partitioning



Assign data to 3 partitions using **hash** partitioning
h(k) = k % 3

2 % 3 = 2

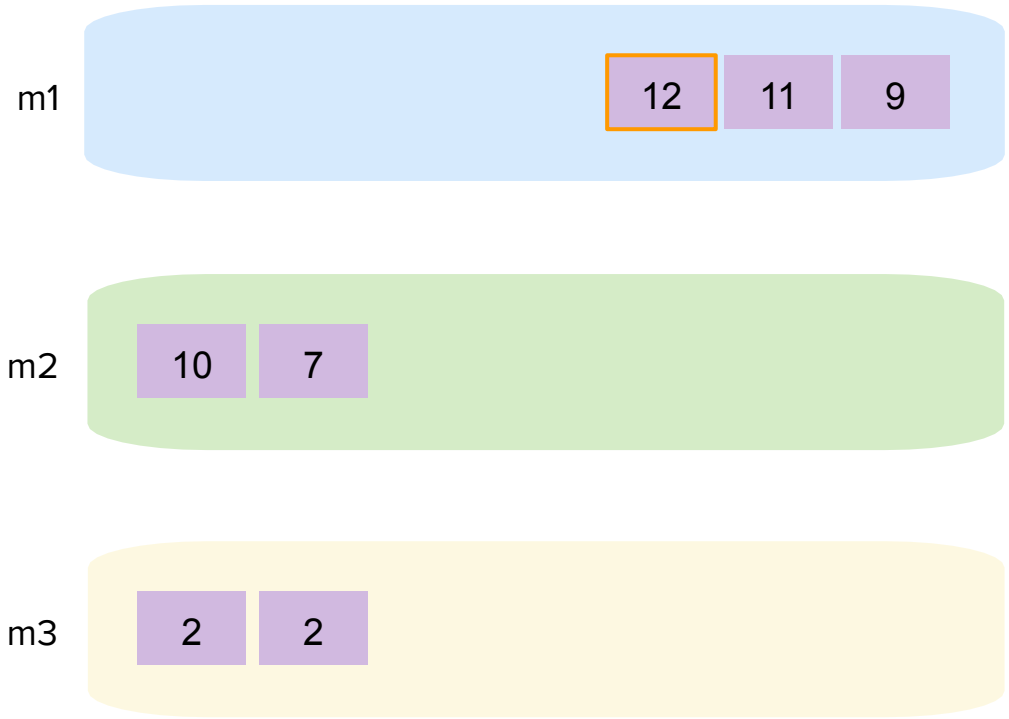
Example: Hash Partitioning



Assign data to 3 partitions using **hash** partitioning
h(k) = k % 3

2 % 3 = 2

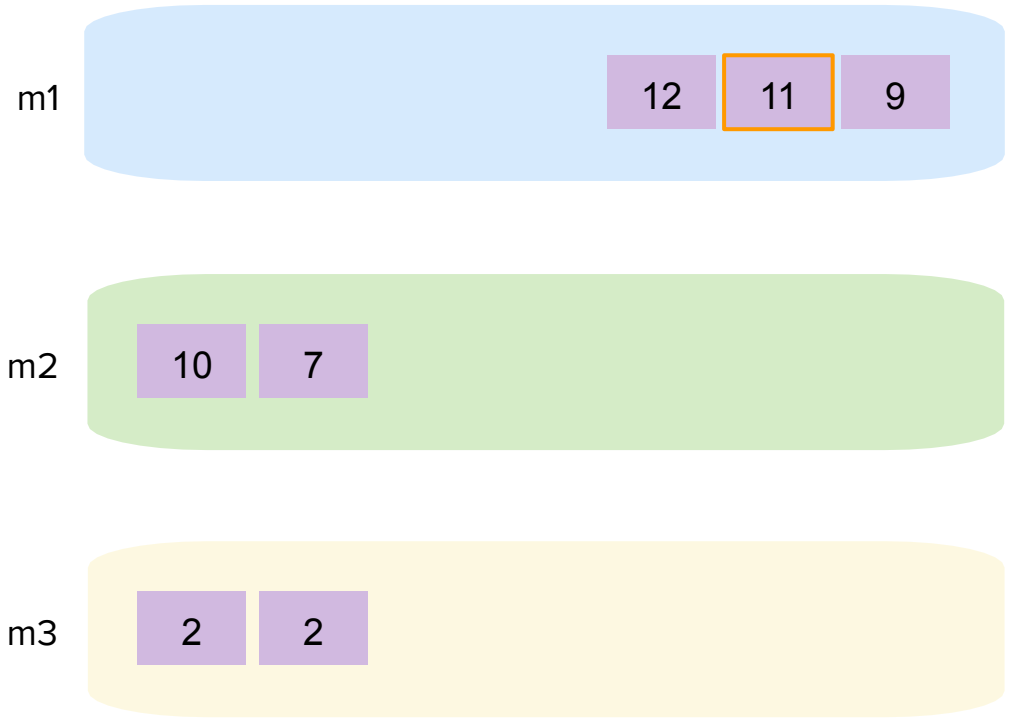
Example: Hash Partitioning



Assign data to 3 partitions using **hash** partitioning
h(k) = k % 3

12 % 3 = 0

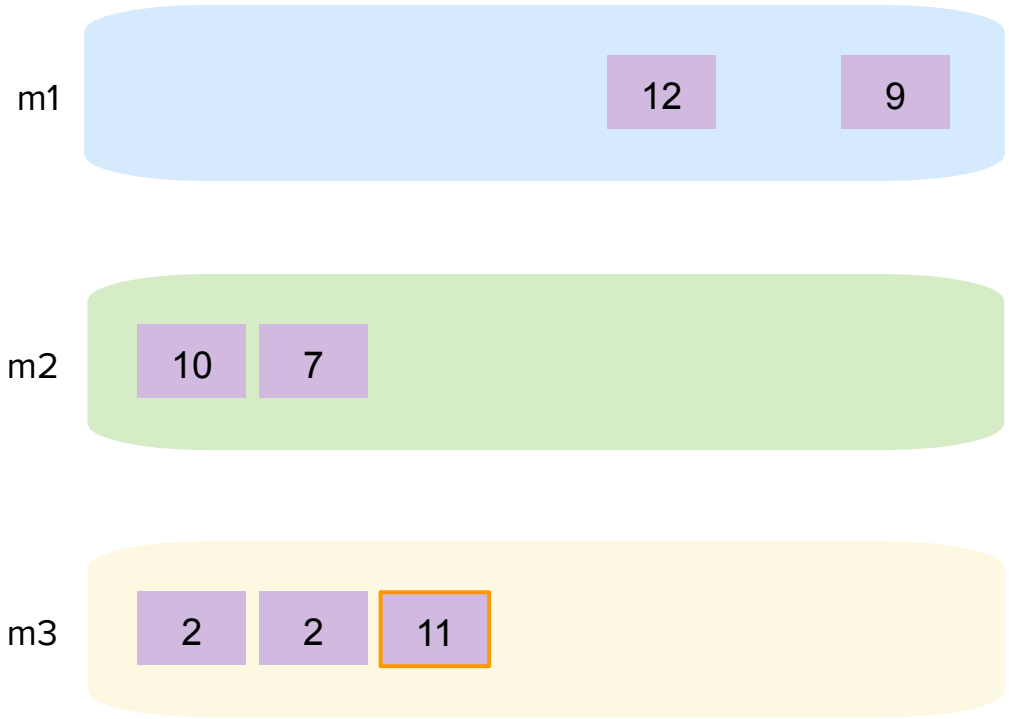
Example: Hash Partitioning



Assign data to 3 partitions using **hash** partitioning
h(k) = k % 3

11 % 3 = 2

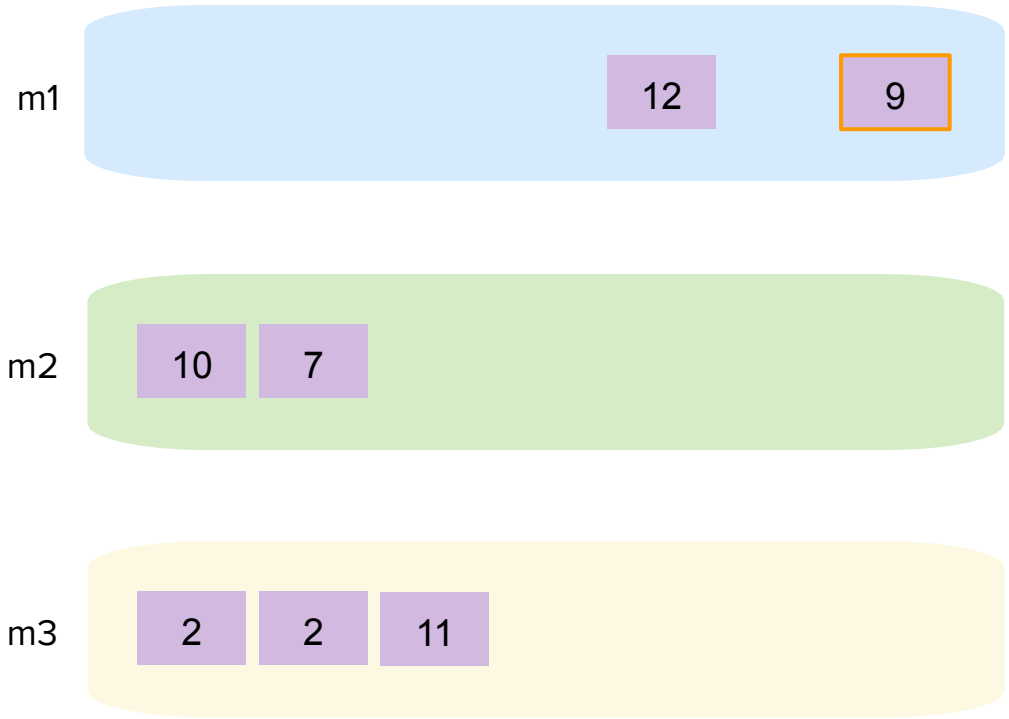
Example: Hash Partitioning



Assign data to 3 partitions using **hash** partitioning
h(k) = k % 3

11 % 3 = 2

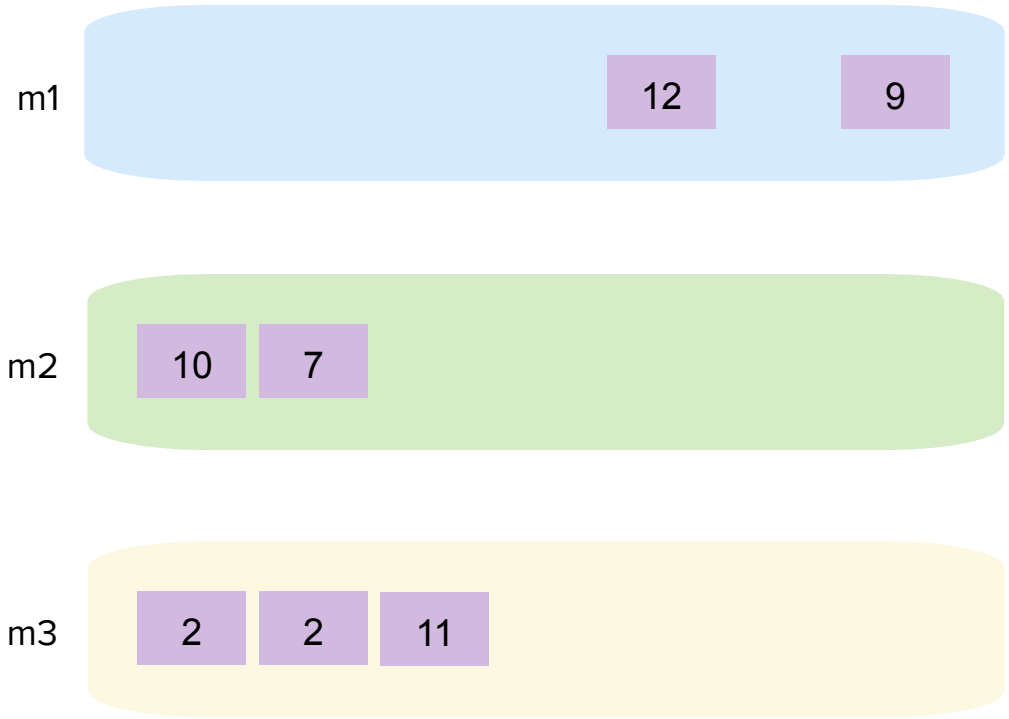
Example: Hash Partitioning



Assign data to 3 partitions using **hash** partitioning
h(k) = k % 3

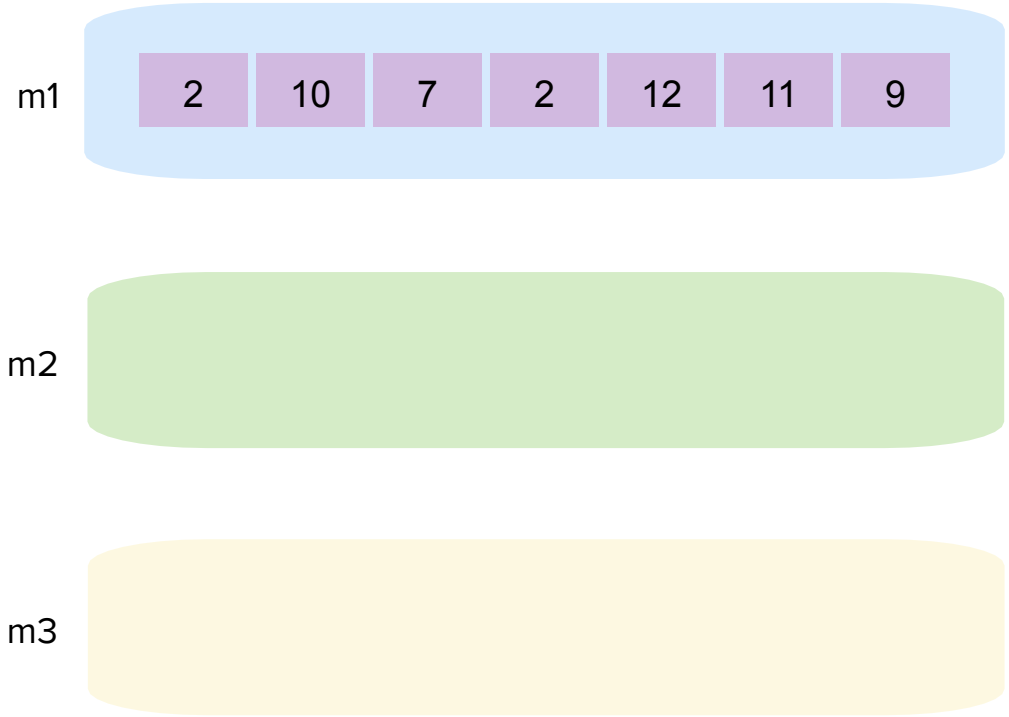
9 % 3 = 0

Example: Hash Partitioning



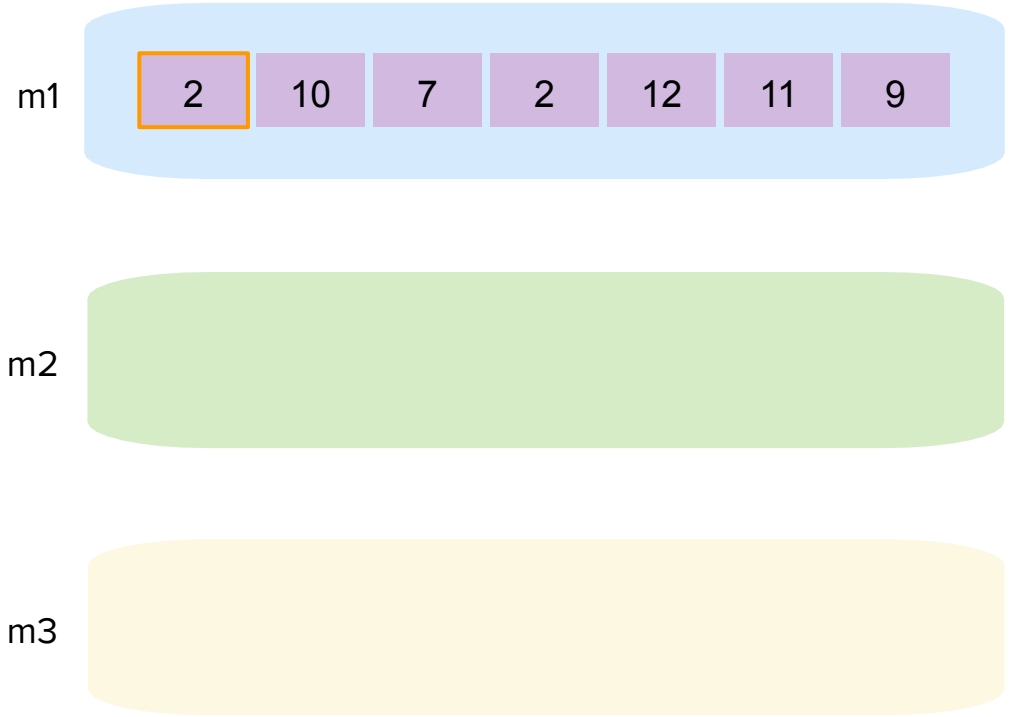
Assign data to 3 partitions using **hash** partitioning
h(k) = k % 3

Example: Round Robin Partitioning



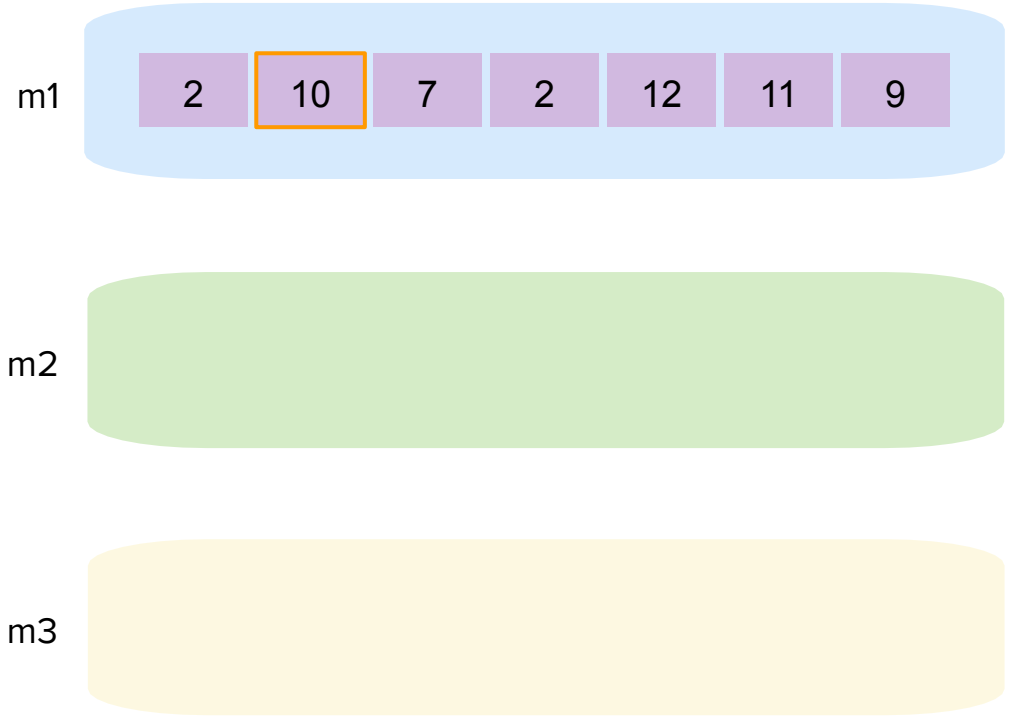
Assign data to 3 partitions using **round robin** partitioning

Example: Round Robin Partitioning



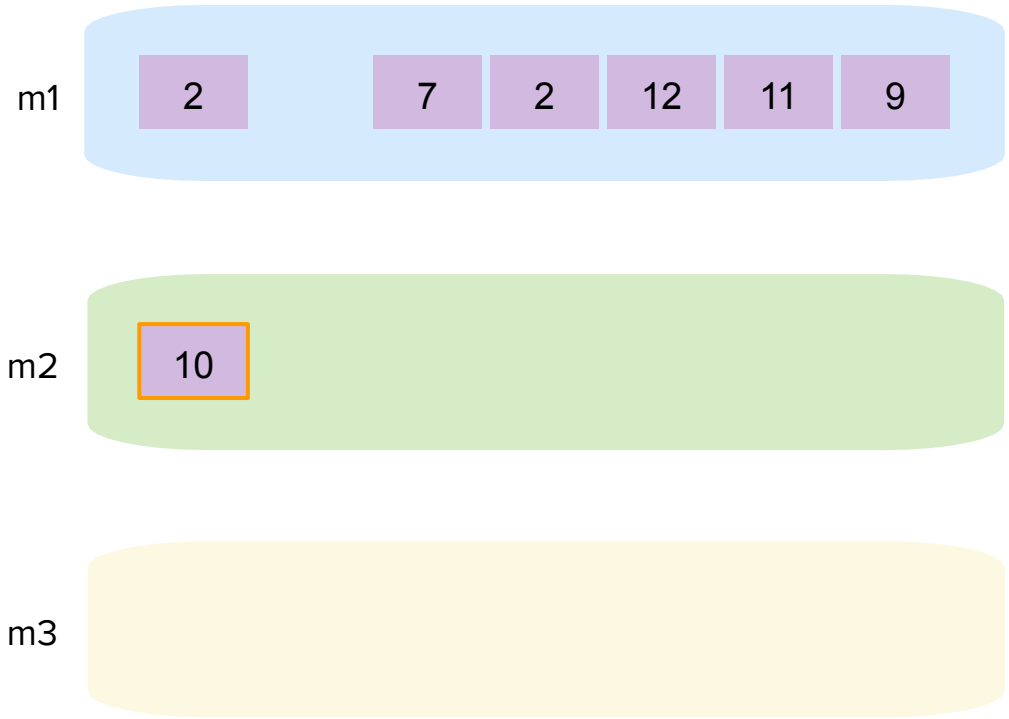
Assign data to 3 partitions using **round robin** partitioning

Example: Round Robin Partitioning



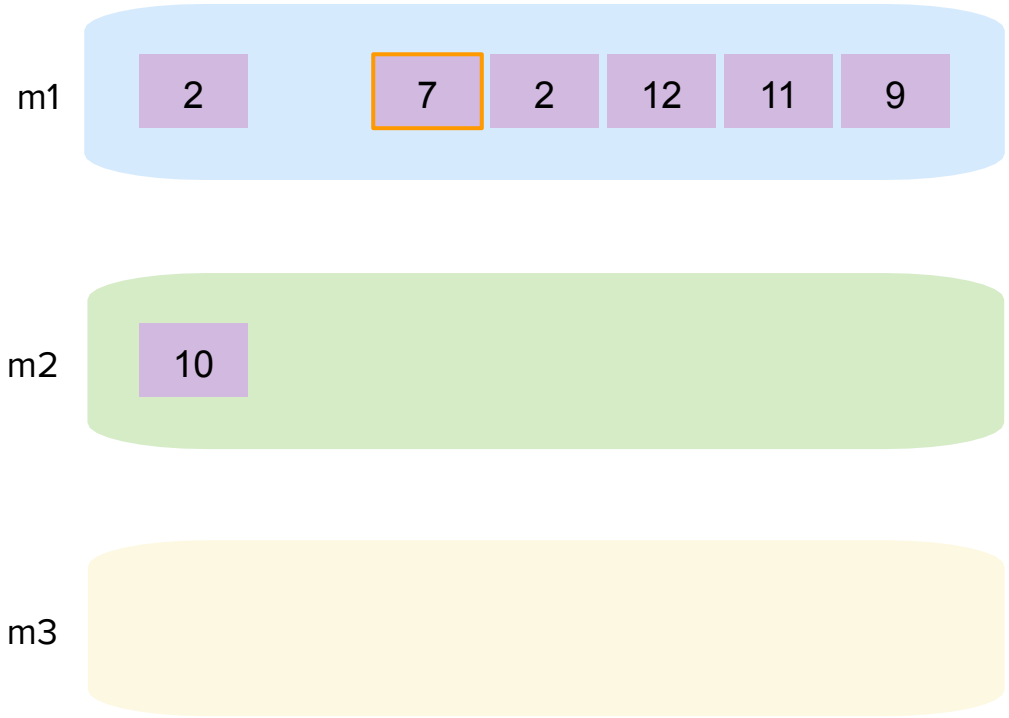
Assign data to 3 partitions using **round robin** partitioning

Example: Round Robin Partitioning



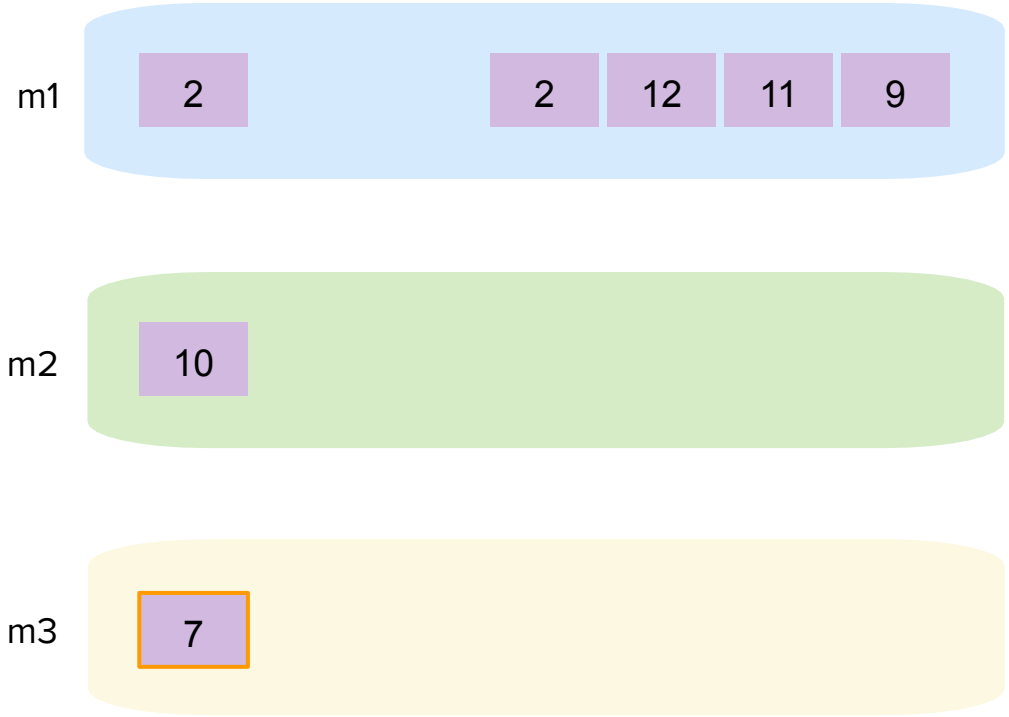
Assign data to 3 partitions using **round robin** partitioning

Example: Round Robin Partitioning



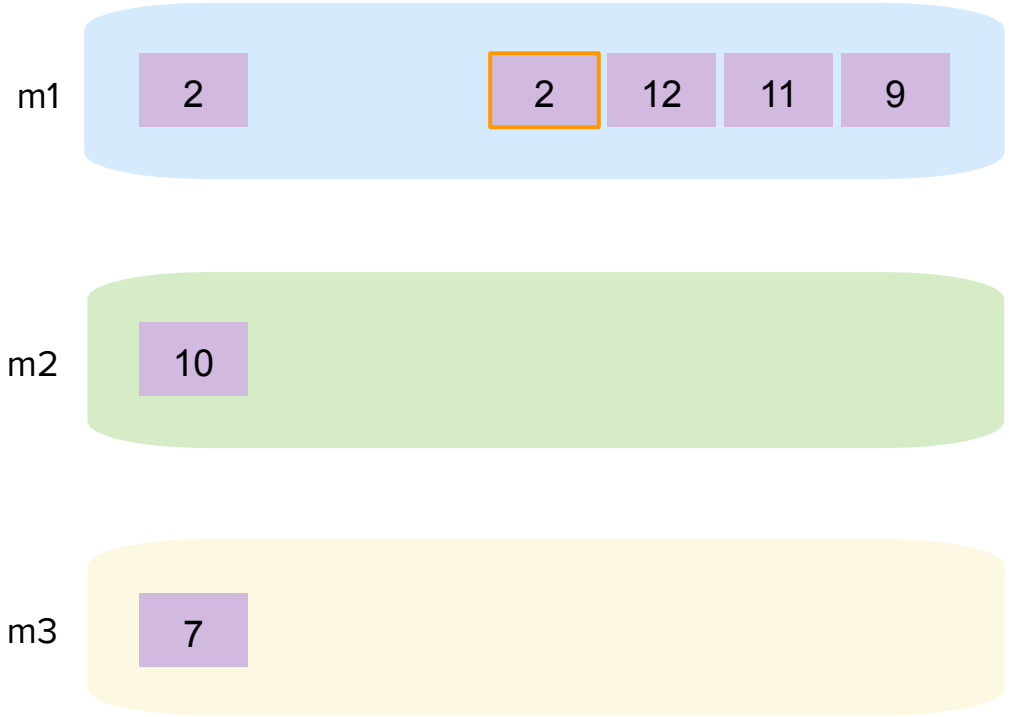
Assign data to 3 partitions using **round robin** partitioning

Example: Round Robin Partitioning



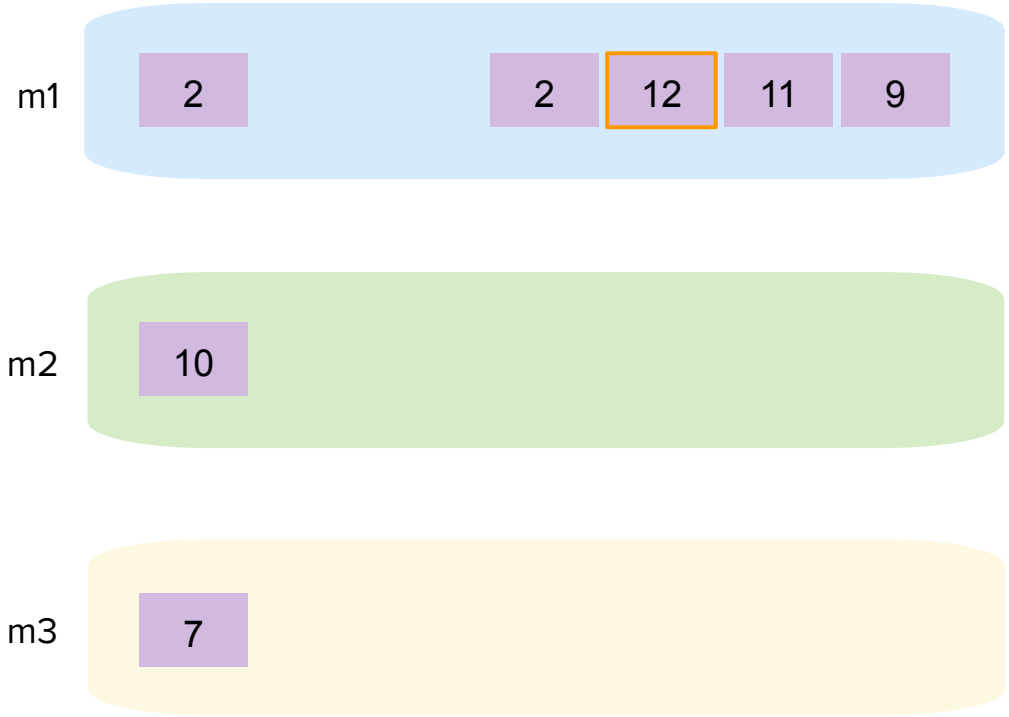
Assign data to 3 partitions using **round robin** partitioning

Example: Round Robin Partitioning



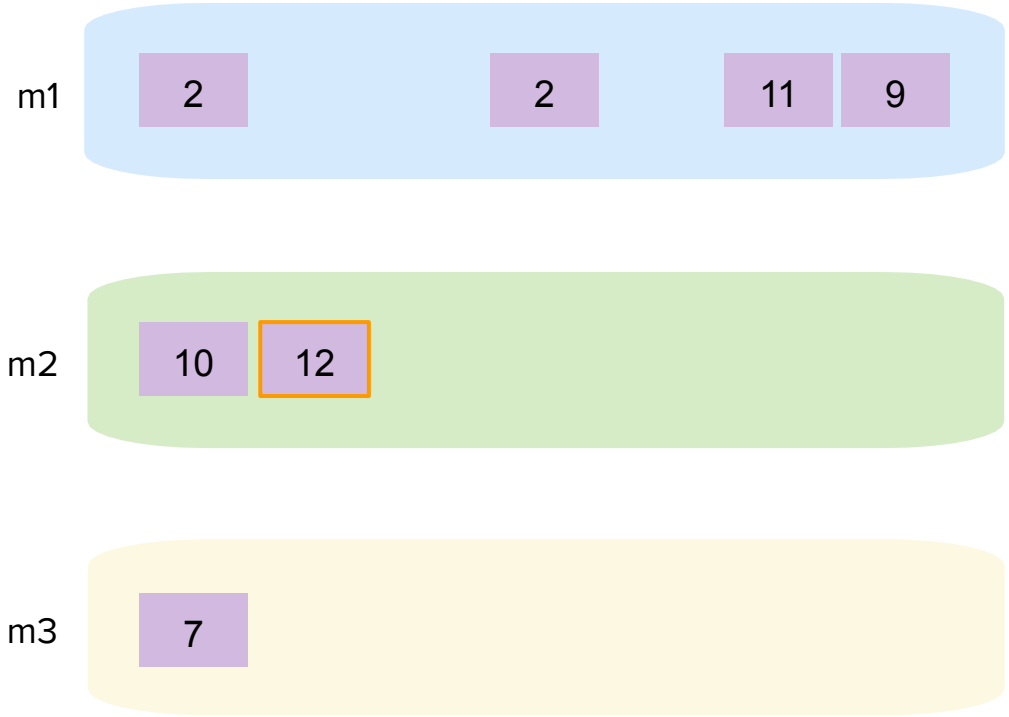
Assign data to 3 partitions using **round robin** partitioning

Example: Round Robin Partitioning



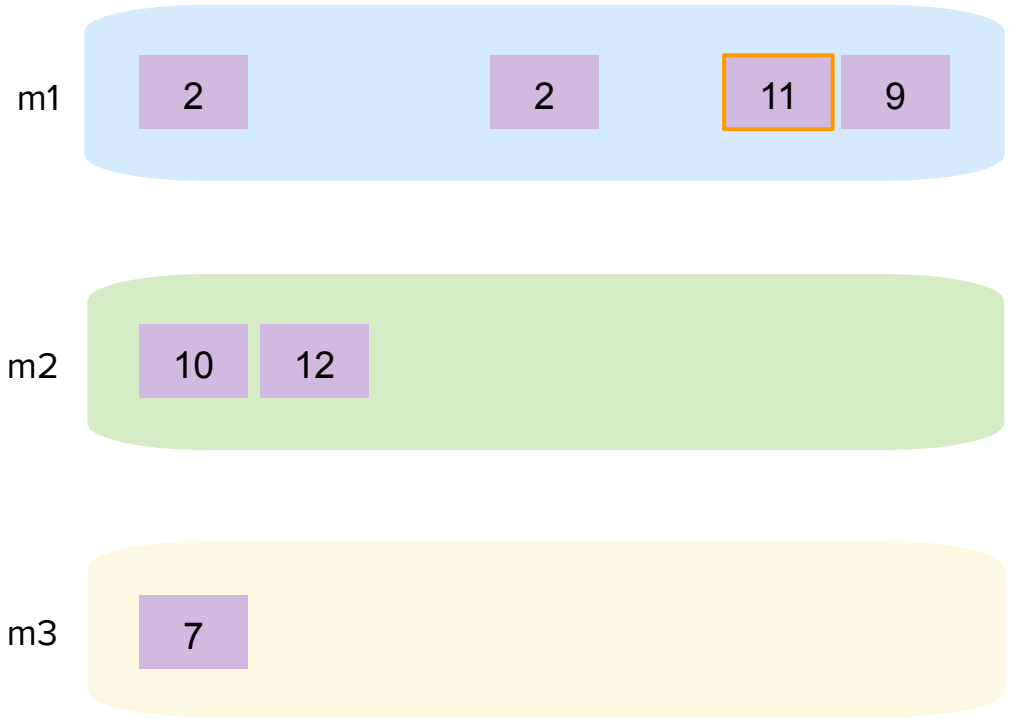
Assign data to 3 partitions using **round robin** partitioning

Example: Round Robin Partitioning



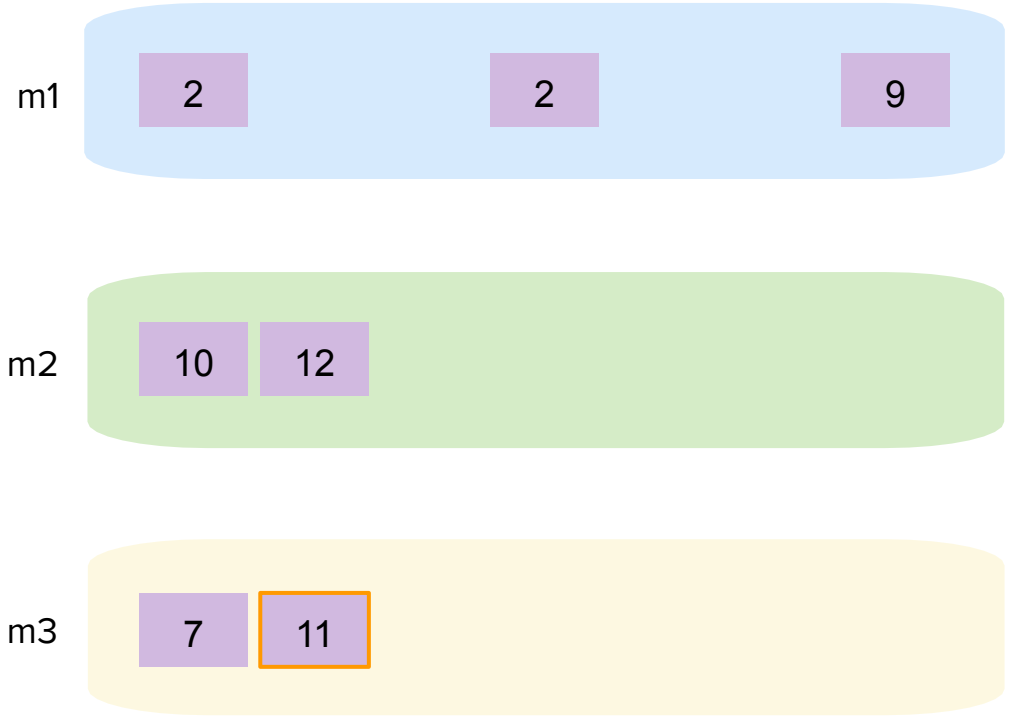
Assign data to 3 partitions using **round robin** partitioning

Example: Round Robin Partitioning



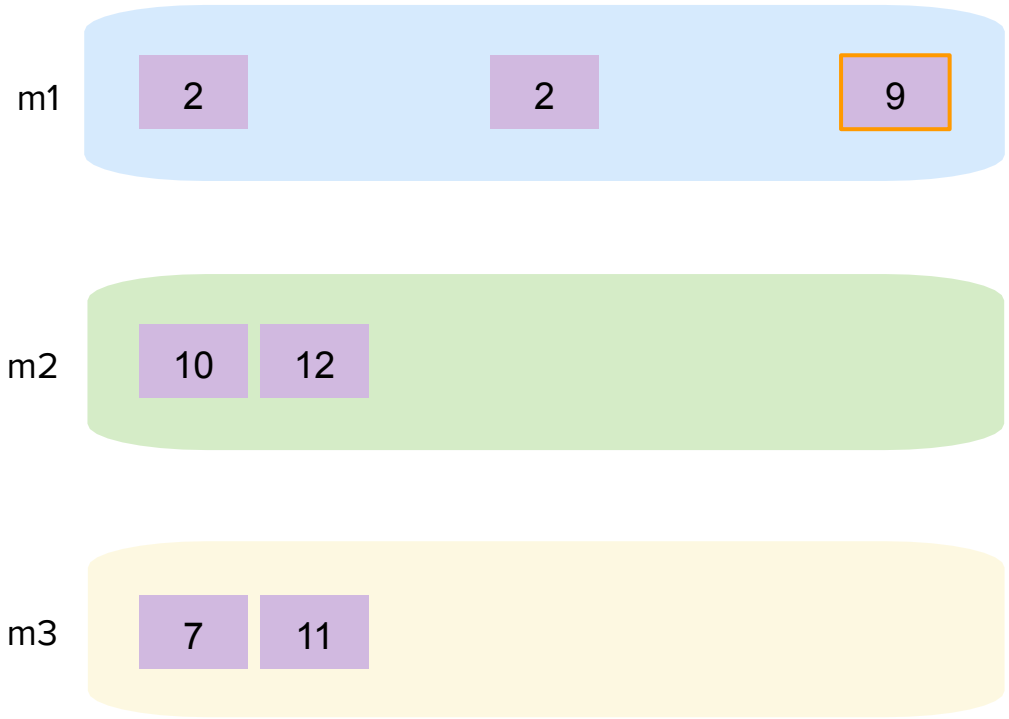
Assign data to 3 partitions using **round robin** partitioning

Example: Round Robin Partitioning



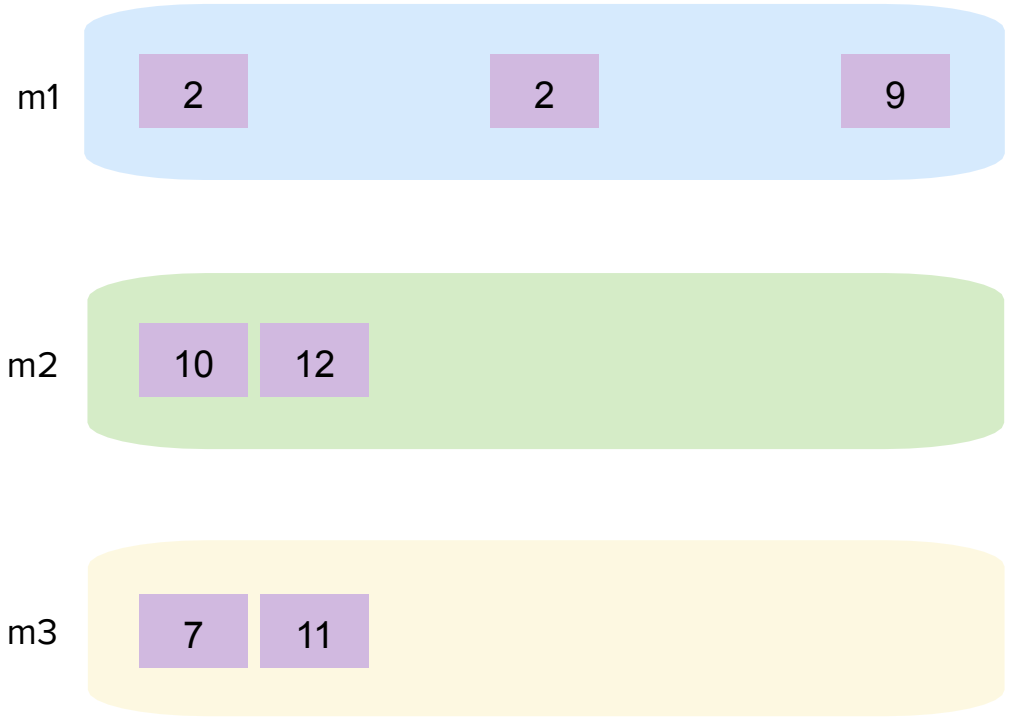
Assign data to 3 partitions using **round robin** partitioning

Example: Round Robin Partitioning



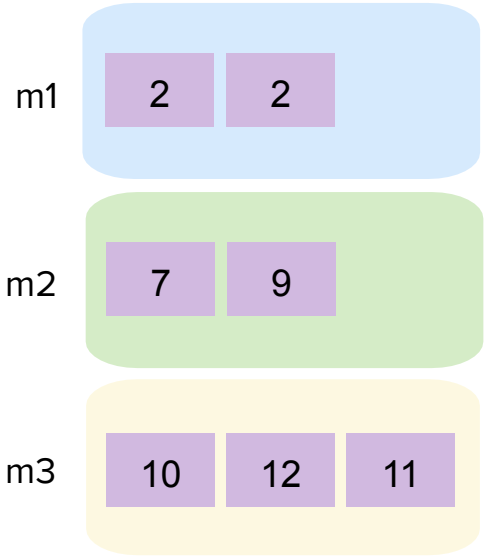
Assign data to 3 partitions using **round robin** partitioning

Example: Round Robin Partitioning

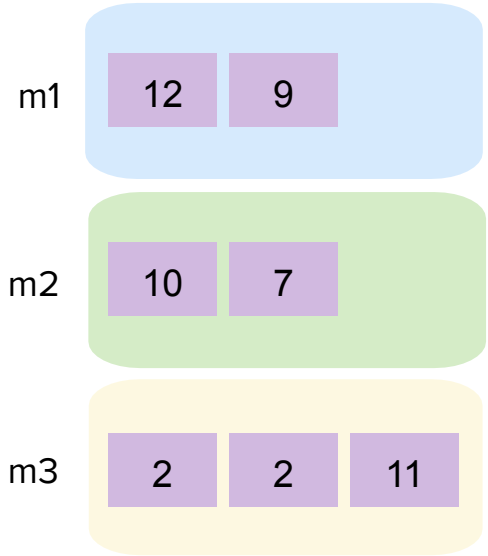


Assign data to 3 partitions using **round robin** partitioning

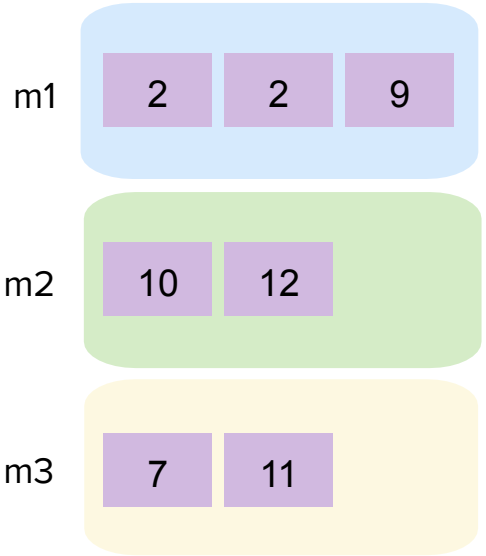
Partitioning Comparison



Range
key divides data
by range



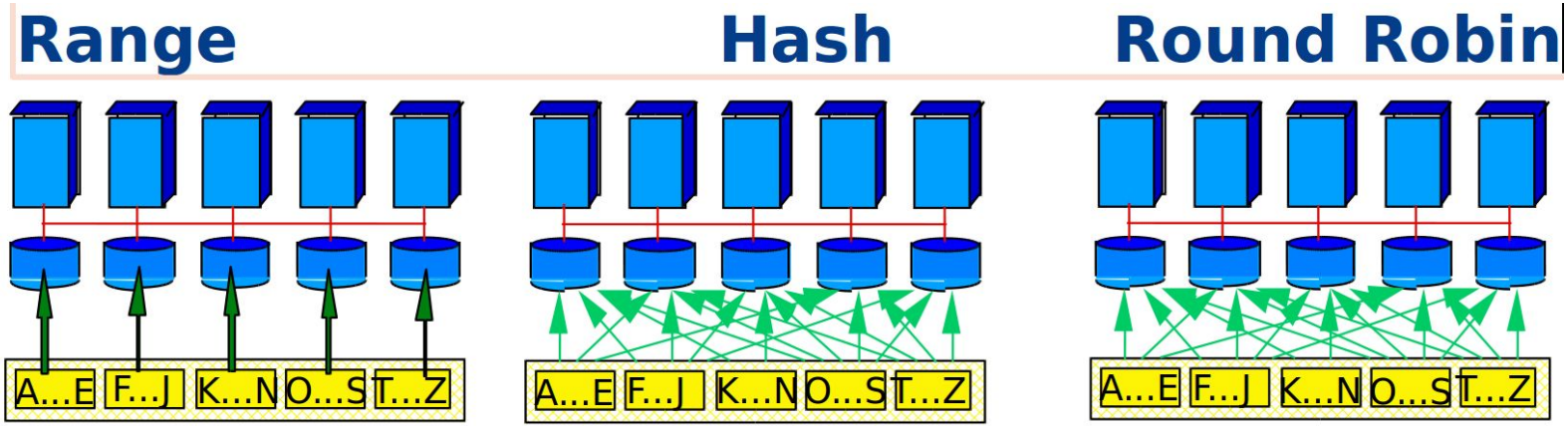
Hash
hash function
divides data



Round Robin
no key

Partitioning data

- Range: key divides data by range
- Hash: hash function divides data
- Round robin: no key



Worksheet #2

What are the advantages and disadvantages of organizing data by keys?

Worksheet #2

What are the advantages and disadvantages of organizing data by keys?

Advantages: because data is organized by keys, search and update operations (which require searching on the key) can be done more efficiently, since we have some sense of where the data must be (if it exists).

Disadvantages: we must maintain the organization, which adds overhead to insertions and updates.

Worksheet #3a

All of the data for a relation with N pages starts on one machine, and we would like to partition the data onto M machines.

How much data (in KB) would be sent over the network to partition the data through each of the following: range, hash, and round-robin partitioning?

Assume that the size of each page is S (in KB). Also, assume we use uniform hash functions and are able to construct ranges that have the same number of values in them.

Worksheet #3a

How much data (in KB) would be sent over the network to partition the data through each of the following: range, hash, and round-robin partitioning?

- The amount of data sent over the network for all three kinds of partitioning will be the same, assuming uniform spread of the data across the ranges and a uniform hash function.
- In this average case, each machine would get 1/M of the data so we would need to send data to the other M-1 machines. The total amount of data sent over the network is $S * N * (M-1) / M$ KB.

Worksheet #3b

If there are no assumptions about hash functions or data ranges, what is the best and worst case in terms of network I/O cost for the three partitioning schemes?

Worksheet #3b

If there are no assumptions about hash functions or data ranges, what is the best and worst case in terms of network I/O cost for the three partitioning schemes?

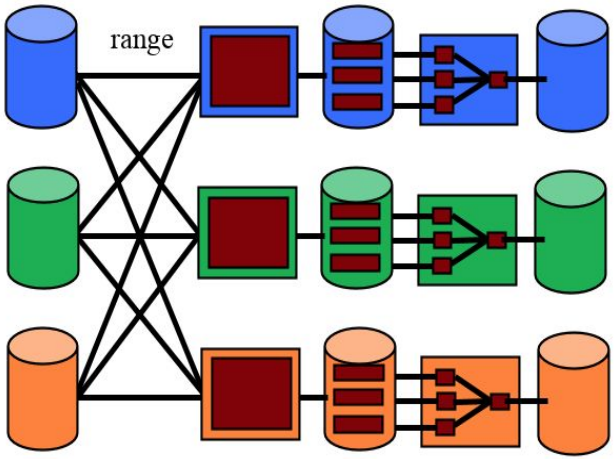
For round-robin partitioning, the best/worst case is still S * N * (M−1) / M KB, since the data will still be divided evenly.

For range and hash partitioning, the best case will be 0 KB, if all the data stays on the current machine. The worst case will be S*N KB, in the scenario where the current machine retains no data and sends all of it to the other machines.

Parallel Query Operators

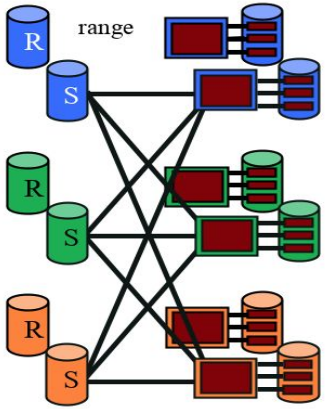
Parallel Sorting

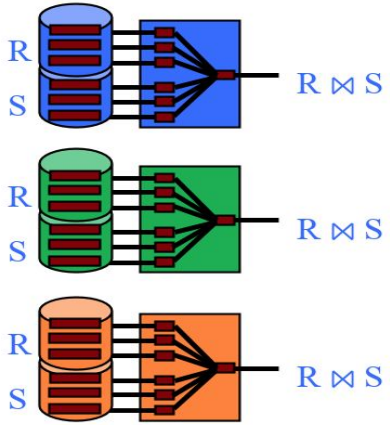
- Partition the data over machines with *range partitioning*
- Perform external sorting on each machine independently (each machine holds a different range of data)



Parallel Sort Merge Join

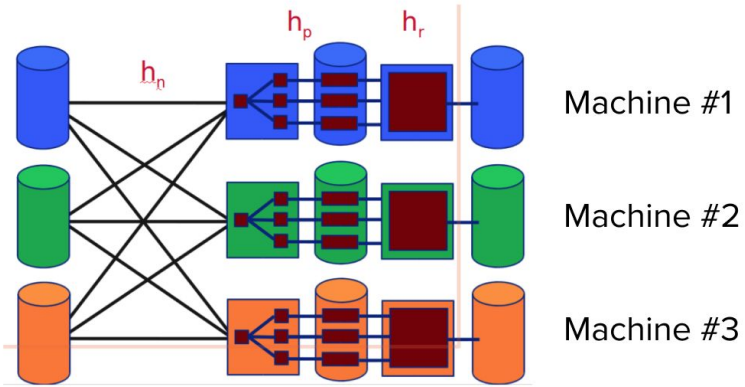
- Partition the data for both relations over machines with *range partitioning*
  - Use **the same ranges** for both relations
- Perform sort merge join on each machine independently





Parallel Hashing

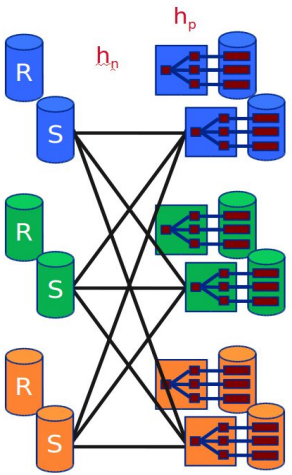
- Use a hash function to partition the data over all the machines (**hash partitioning**), then run external hashing on each machine independently
  - Similar to recursive partitioning



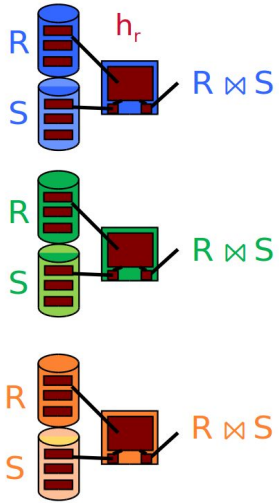
Parallel Hash Joins

- Use hash partitioning on both relations, then perform a normal hash join on each machine independently

Phase 1: Partition R and S across different machines



Phase 2: Within each machine, perform local (grace) hash joins



Join Pipelining

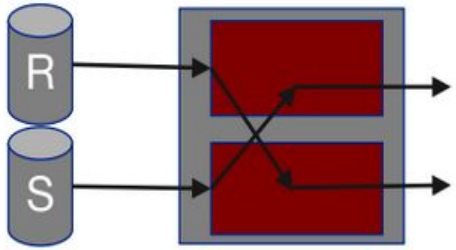
- For both parallel sort-merge and hash joins, the sort or hash must complete before completing the join!
- This “breaks” the pipeline
  - All machines need to stop and wait for all other machines to finish sorting/hashing!
  - This coordination can be costly
- Is there a join algorithm that allows pipelining?

Symmetric Hash Joins

- **Streaming** hash join algorithm
  - Does not require all tuples of one relation to be available before starting
    - Partitions for both R and S may come from multiple machines - inefficient to wait for all of the partition to arrive before joining
  - No writing to disk
  - Requires both R *and* S partitions to fit in memory (in B-1 pages)
    - Get more machines to make partitions small

Symmetric Hash Joins

- Basic idea: build two hash tables at the same time
- When a tuple from R arrives:
  - Probe hash table for S
  - Add the tuple into hash table for R
- When a tuple from S arrives
  - Probe hash table for R
  - Add the tuple into hash table for S
- Generates each joined tuple once
  - An output tuple is generated when both parts arrive



Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} ➔ 1, {B} ➔ 2, {R} ➔ 3

Table R partition 1

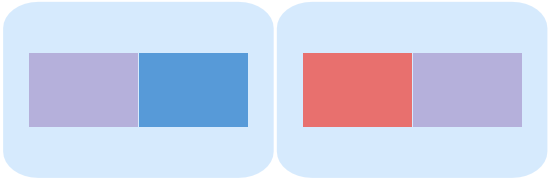
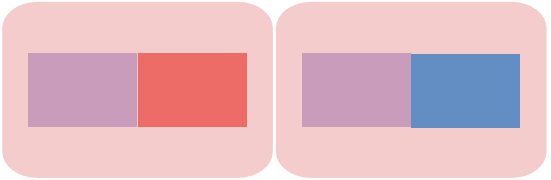
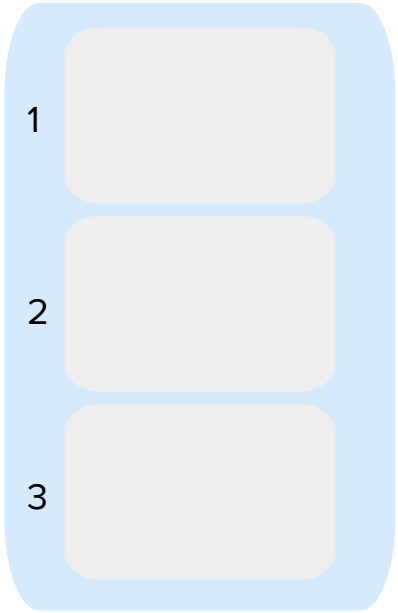
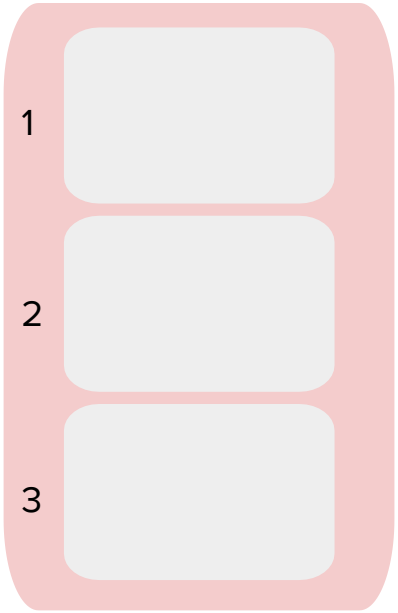


Table S partition 1



B=7

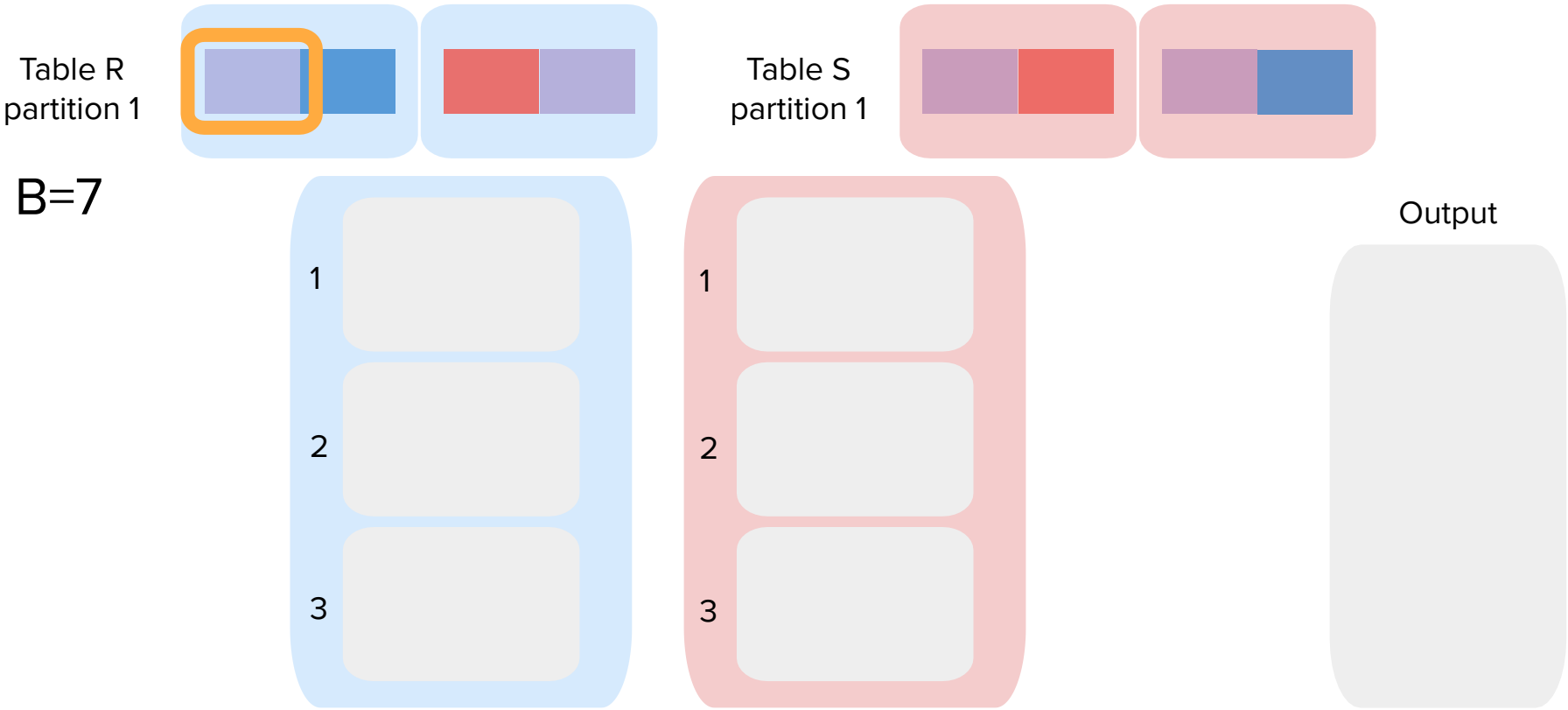




Output

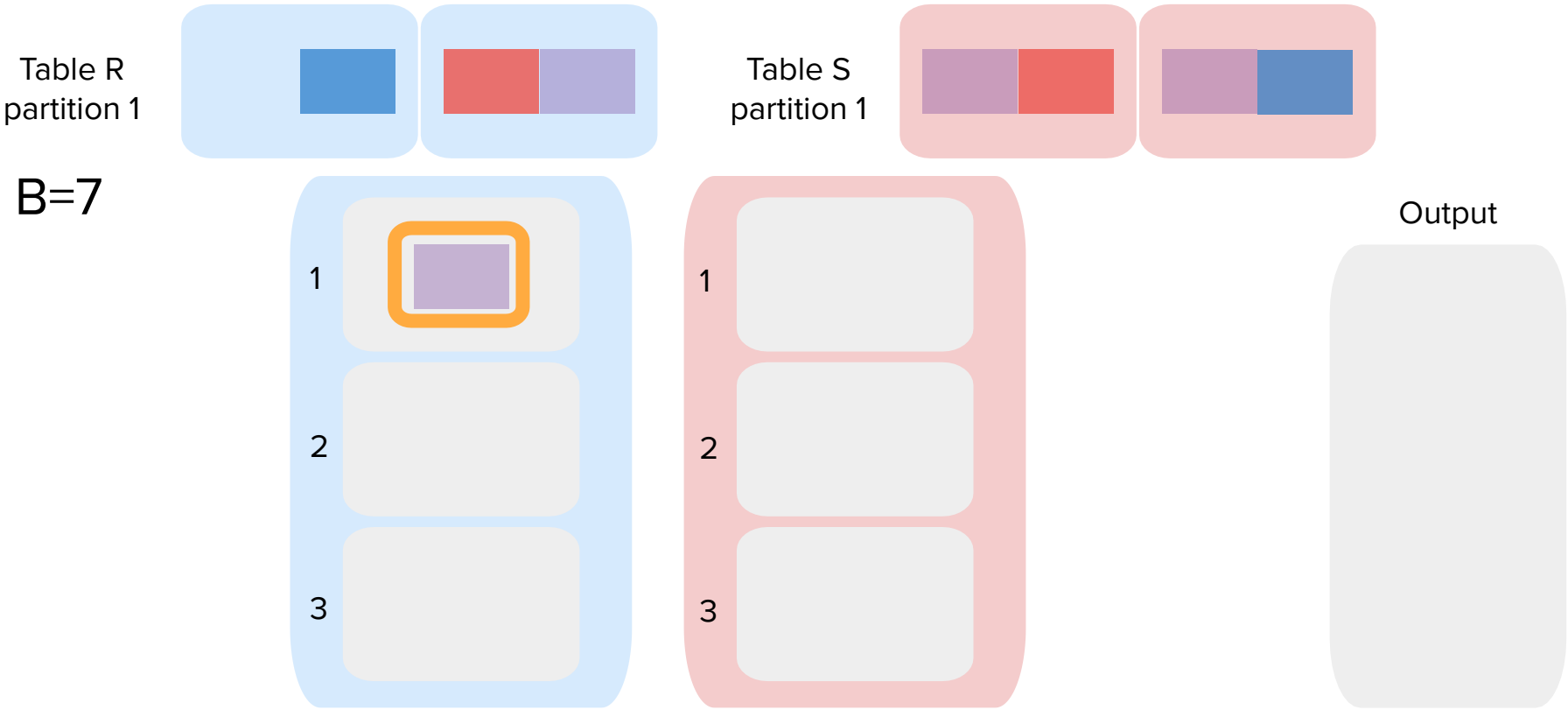
Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3



Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3



Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3

Table R partition 1

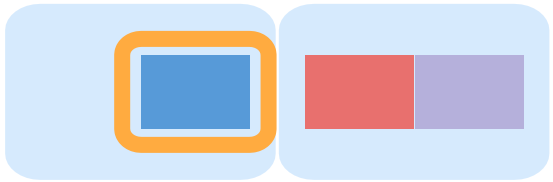
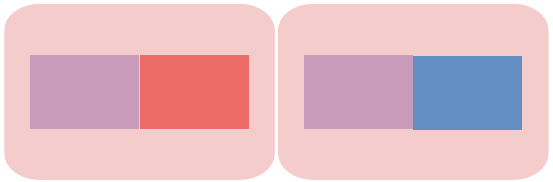
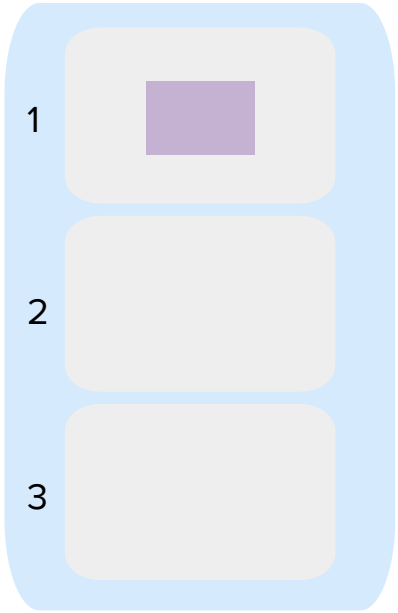
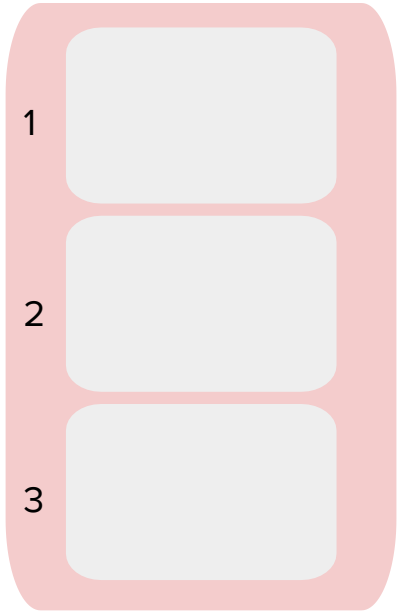


Table S partition 1



B=7





Output

Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3

Table R partition 1

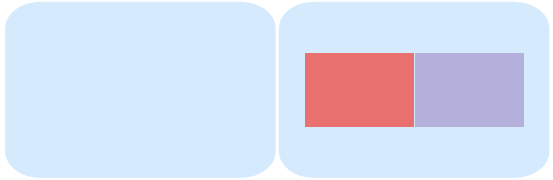
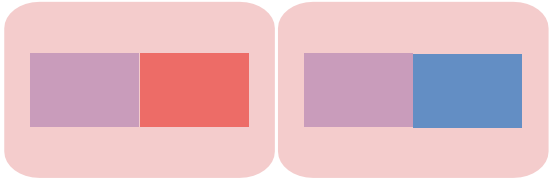
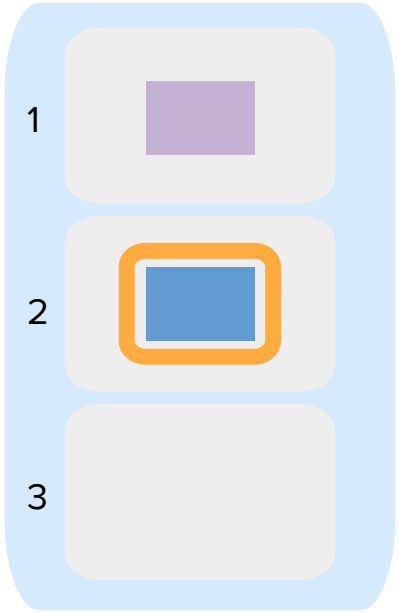
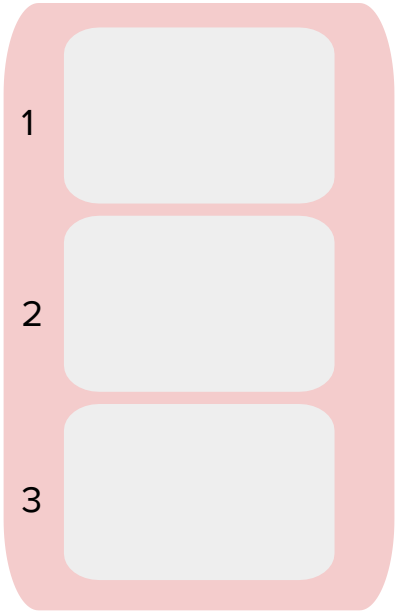


Table S partition 1



B=7

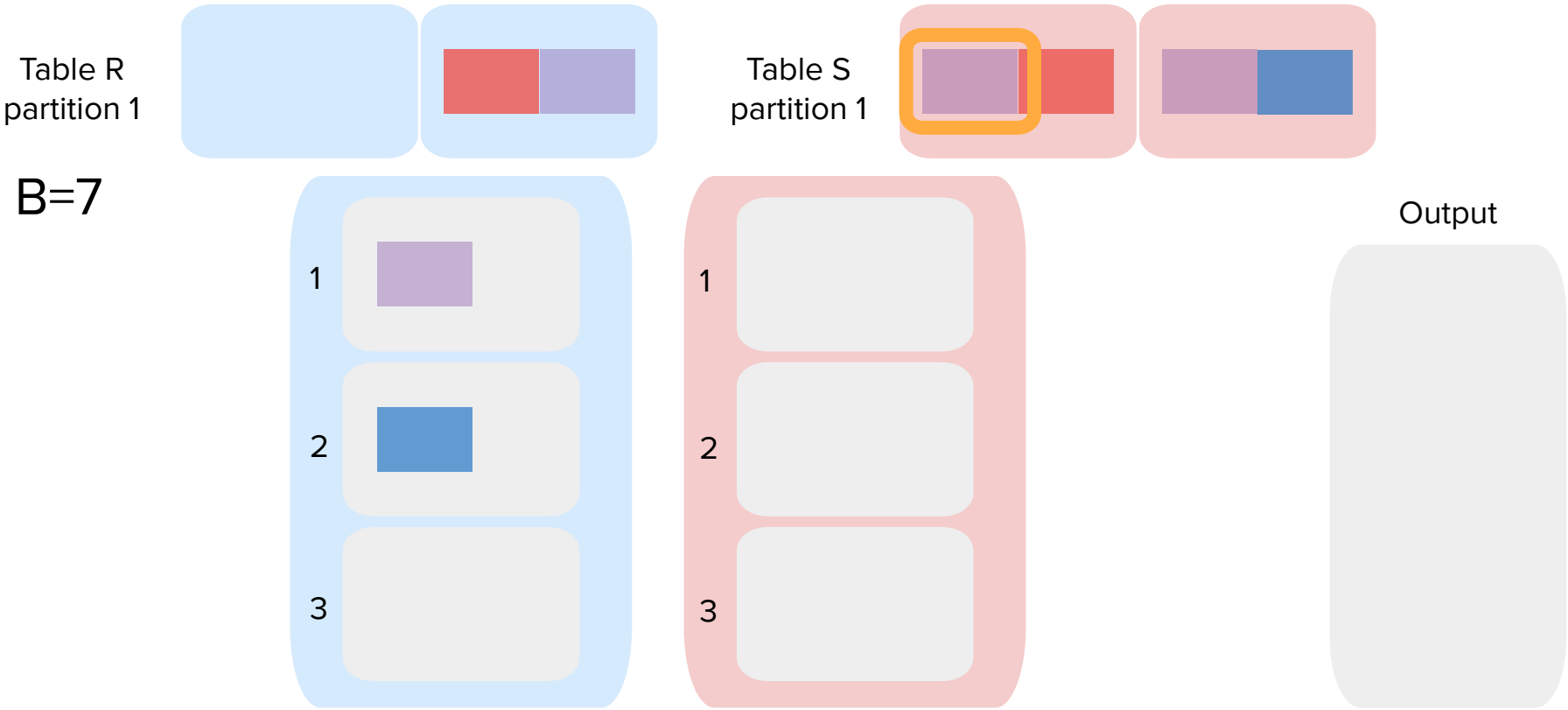




Output

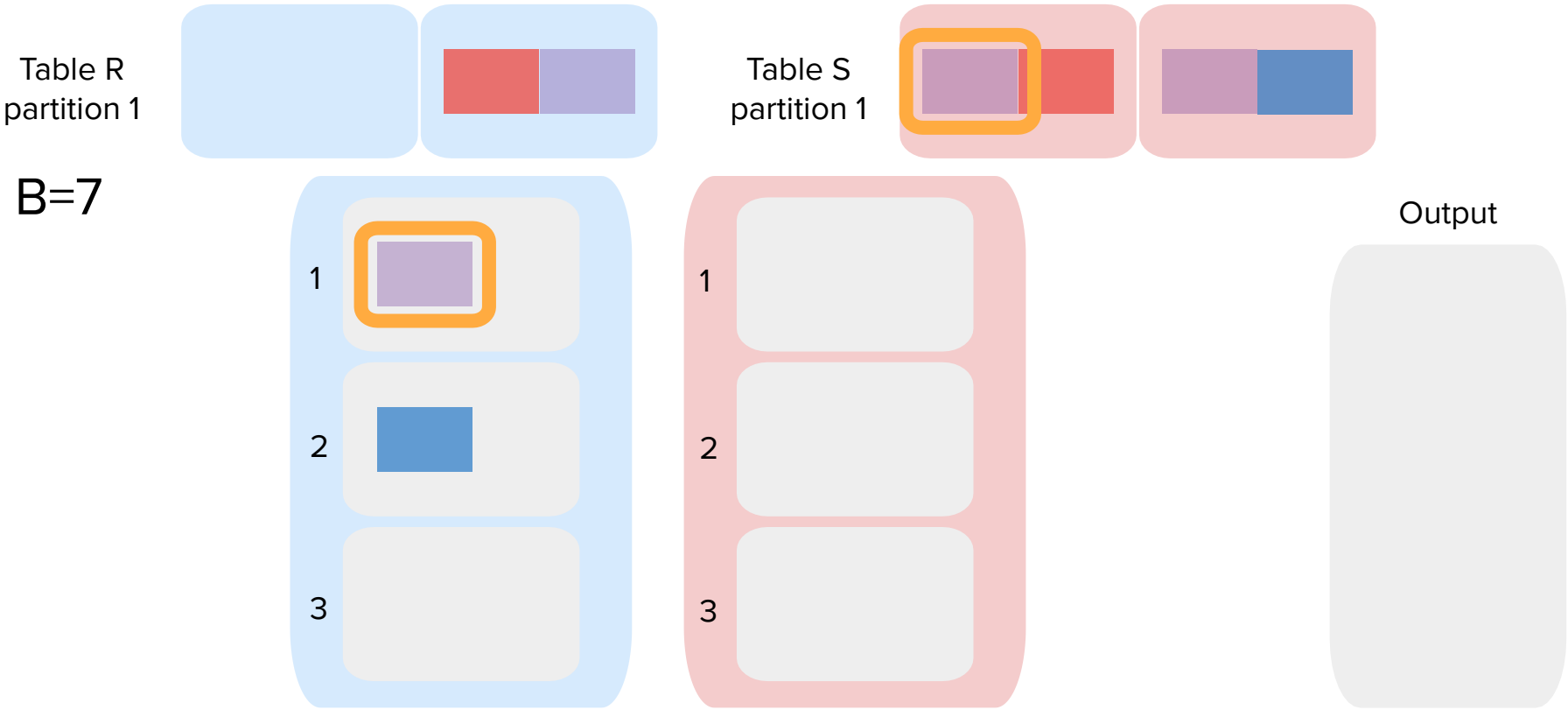
Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} ➔ 1, {B} ➔ 2, {R} ➔ 3



Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3



Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3

Table R partition 1

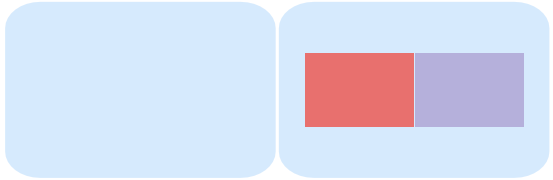
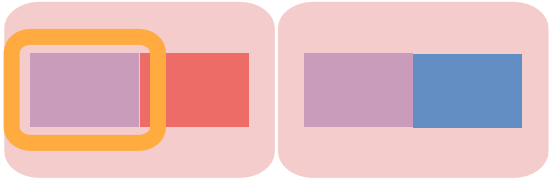
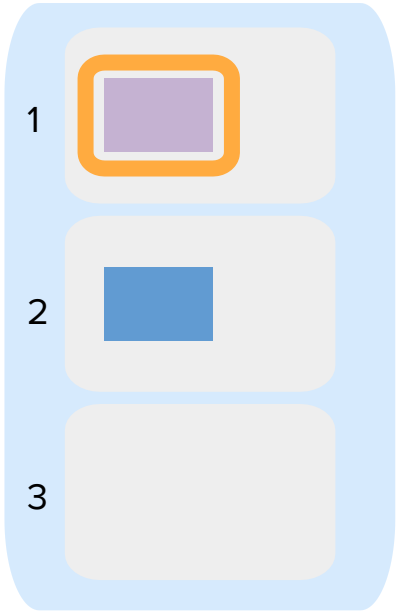
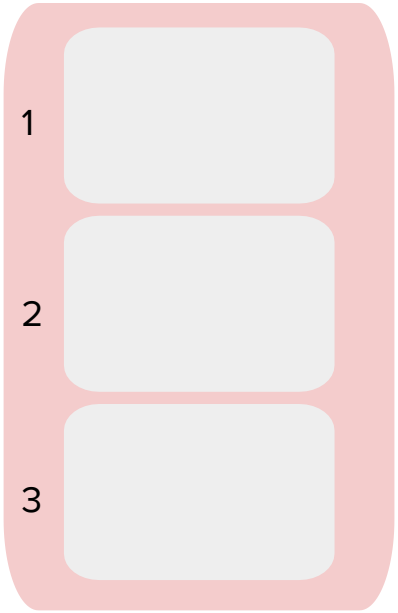


Table S partition 1

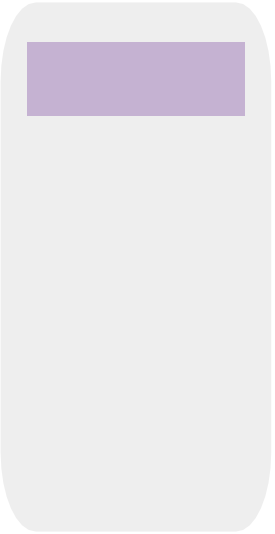


B=7





Output



Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3

Table R partition 1

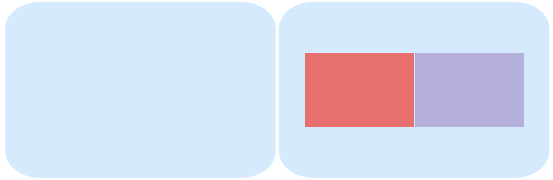
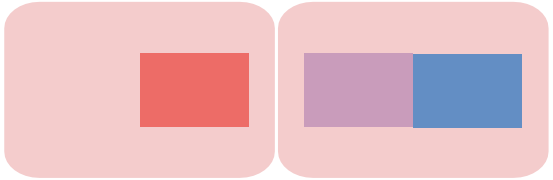
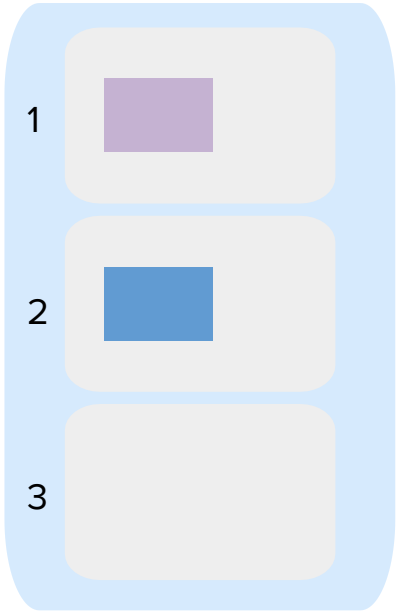
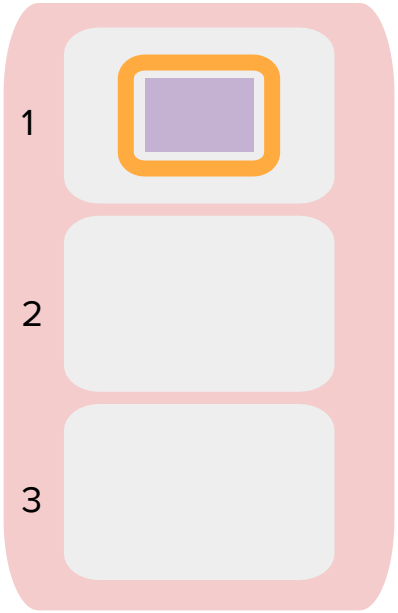


Table S partition 1

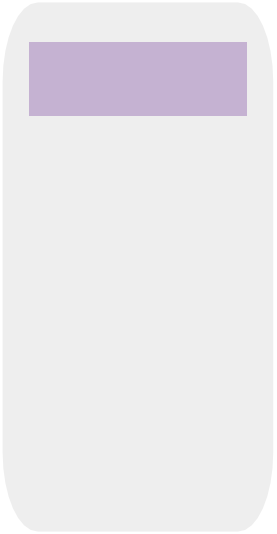


B=7





Output



Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} ➔ 1, {B} ➔ 2, {R} ➔ 3

Table R partition 1

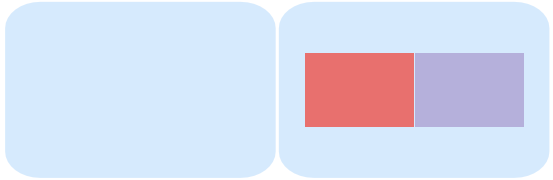
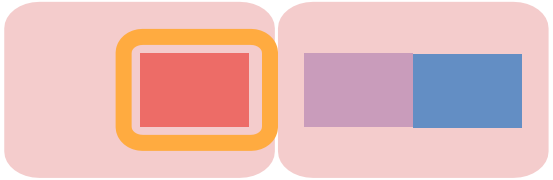
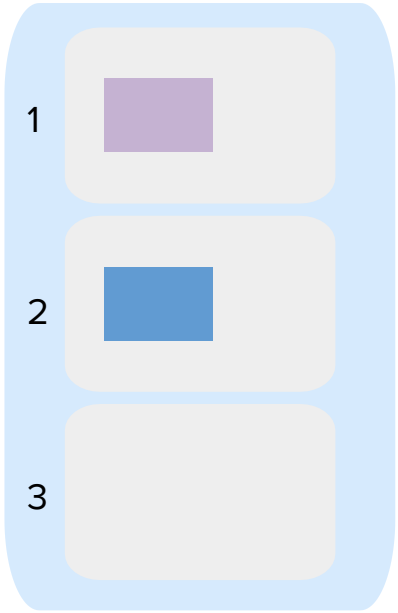
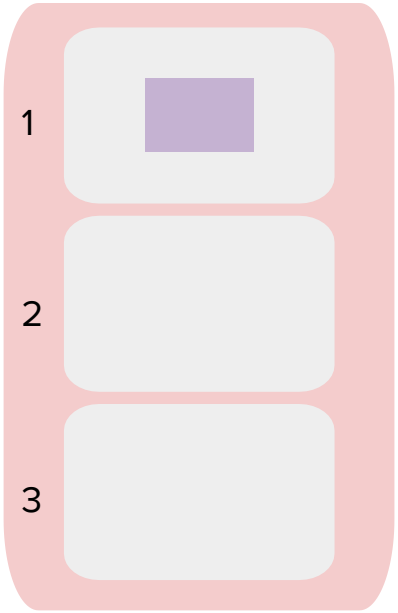


Table S partition 1

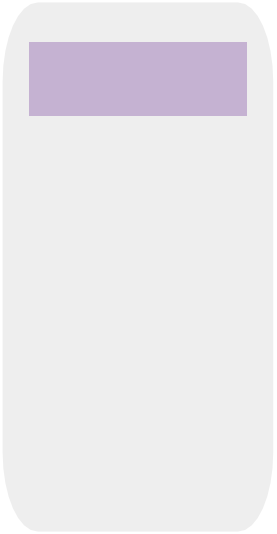


B=7



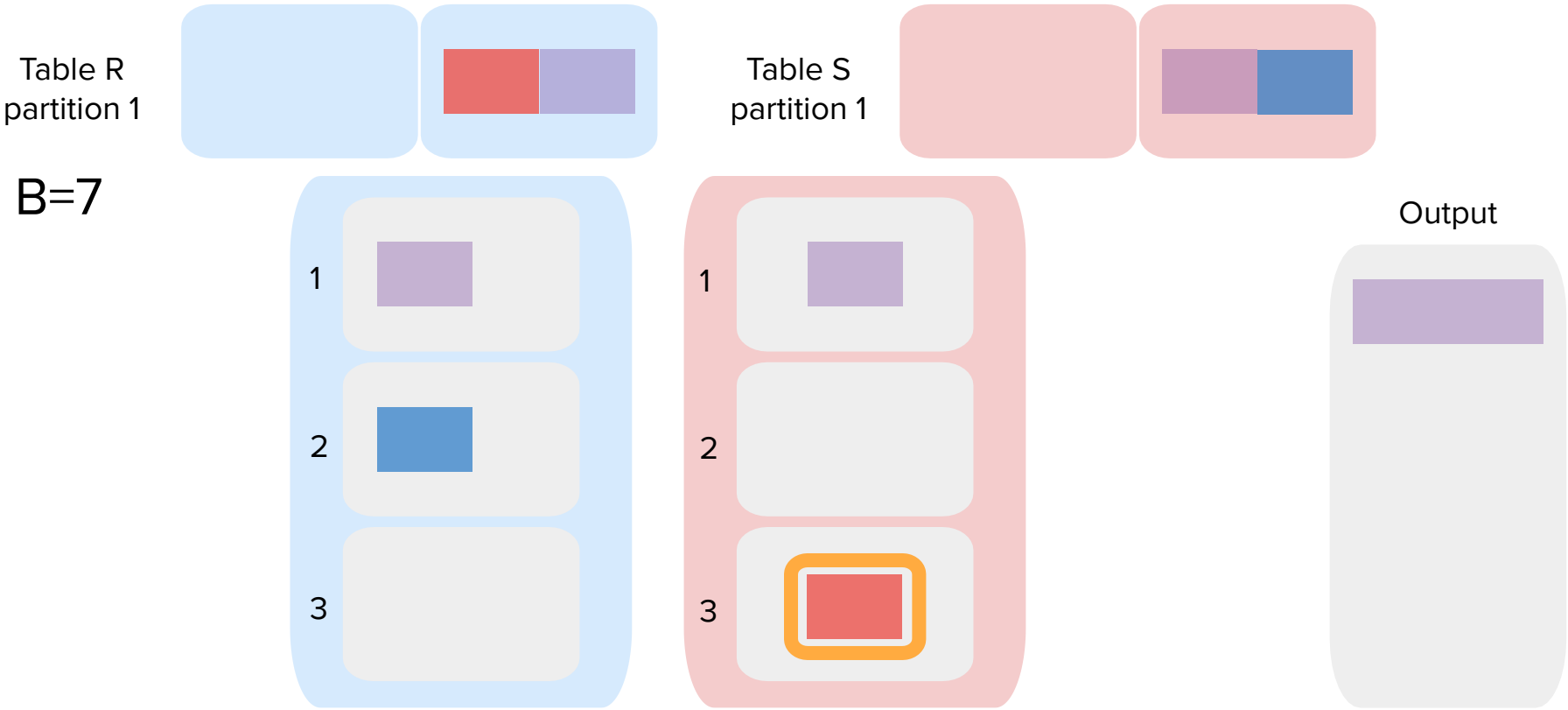


Output



Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3



Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} ➔ 1, {B} ➔ 2, {R} ➔ 3

Table R partition 1

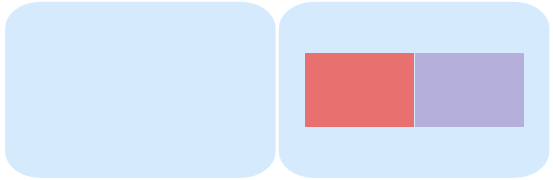
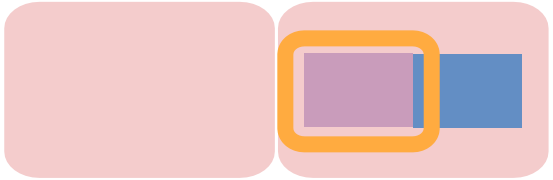
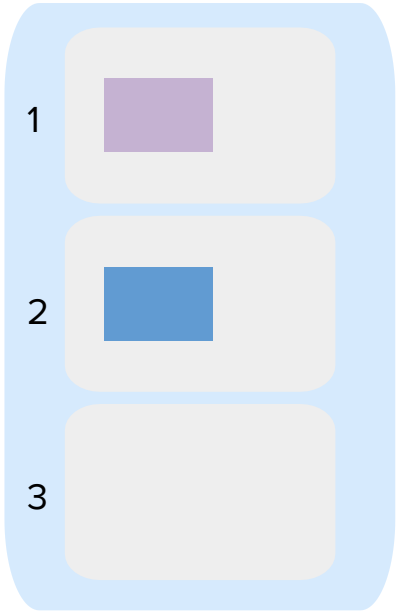
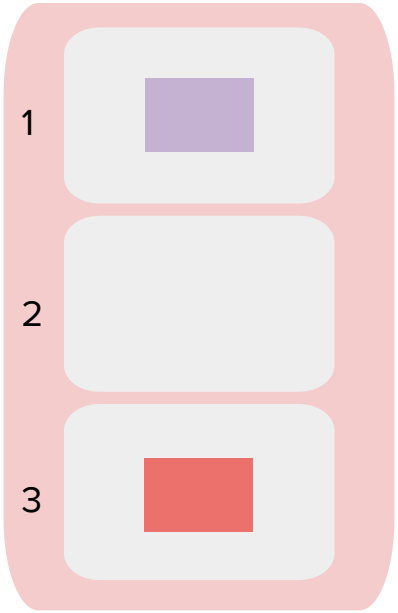


Table S partition 1

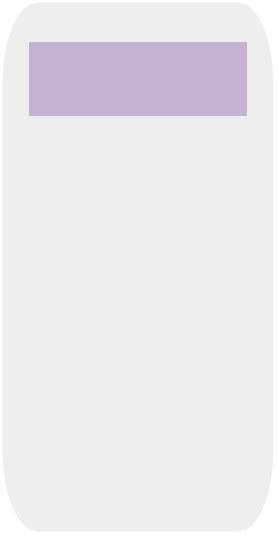


B=7





Output



Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3

Table R partition 1

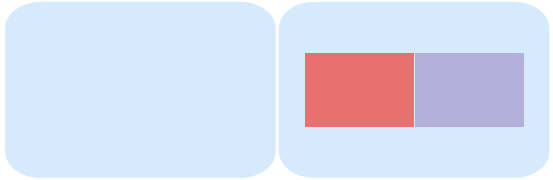
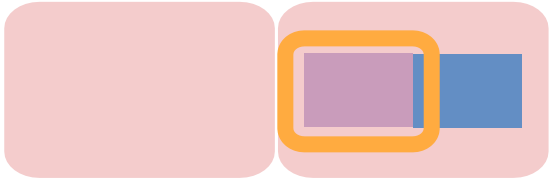
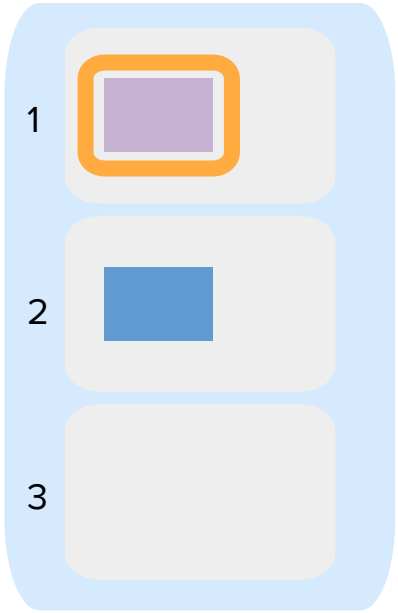
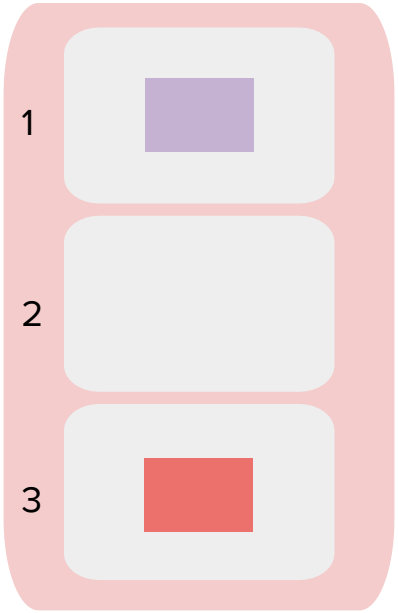


Table S partition 1

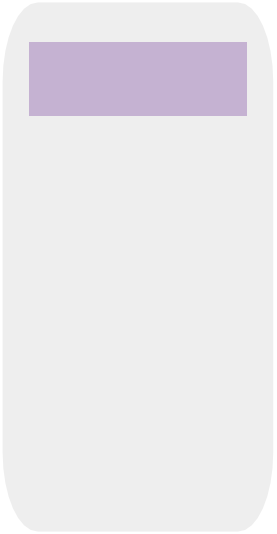


B=7





Output



Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3

Table R partition 1

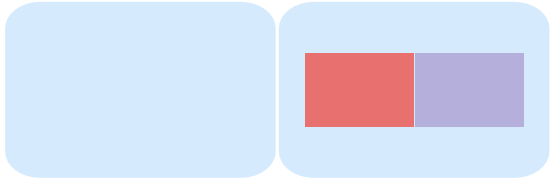
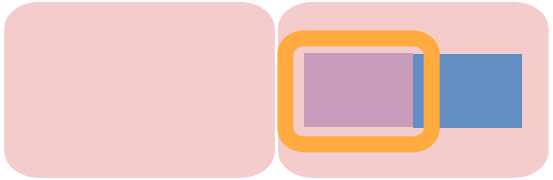
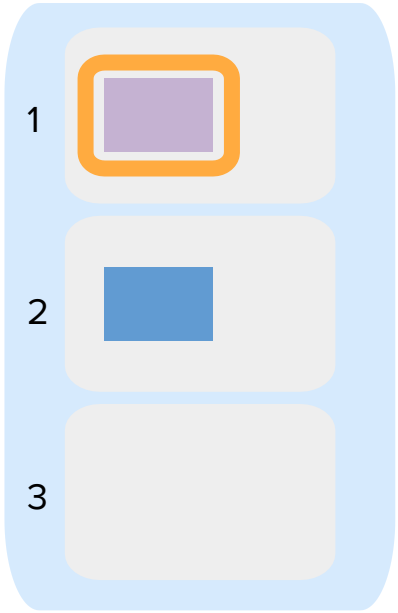
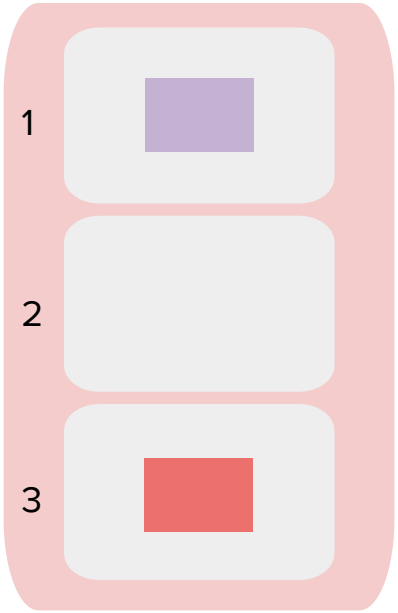


Table S partition 1



B=7

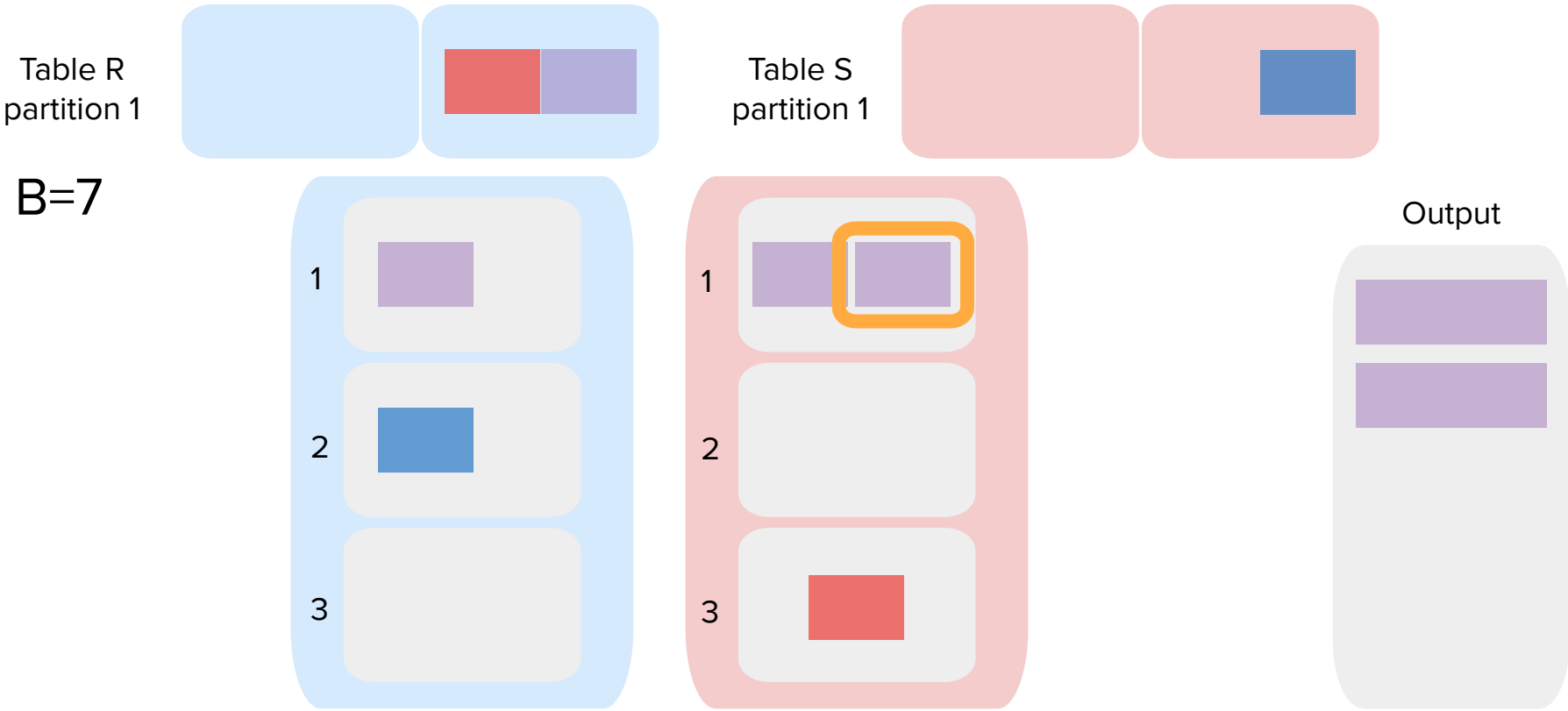




Output

Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3



Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3

Table R partition 1

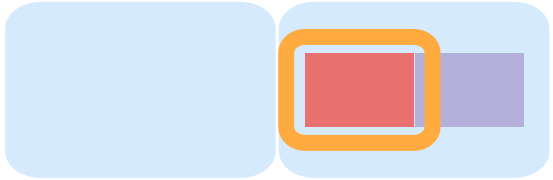
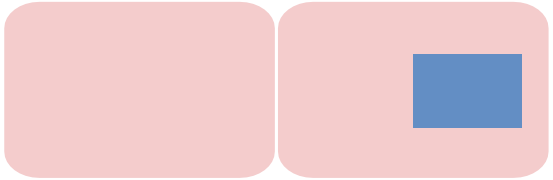
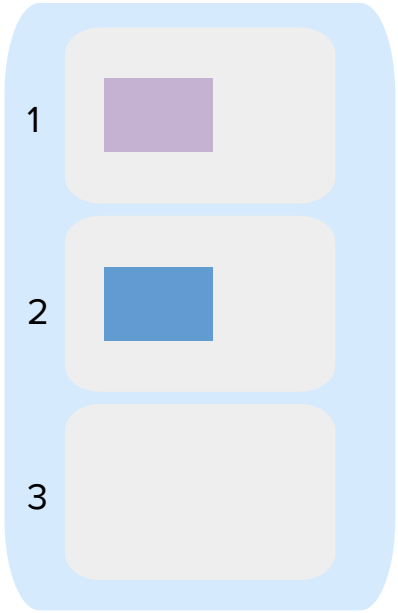
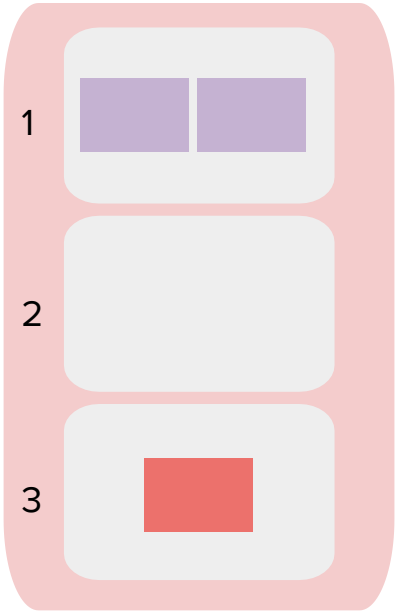


Table S partition 1



B=7

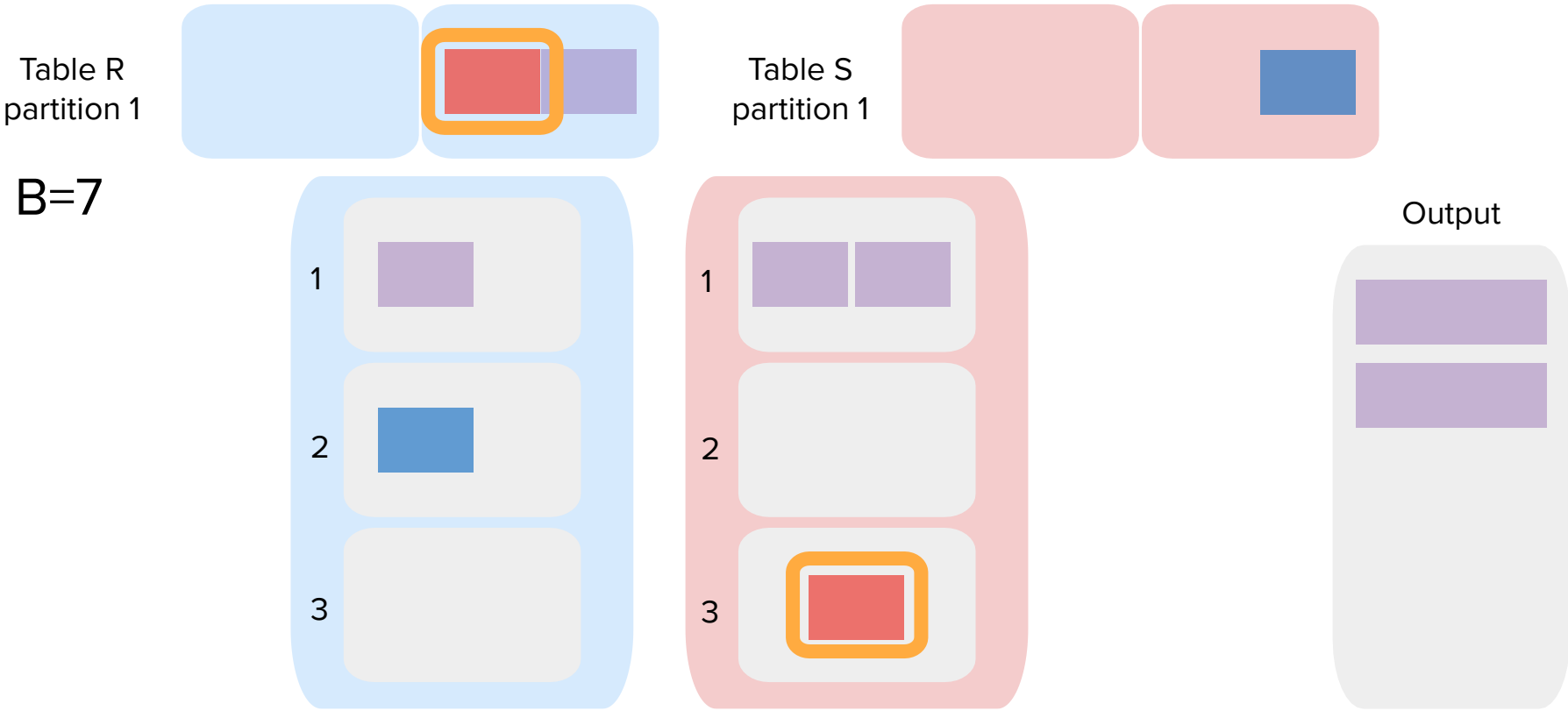




Output

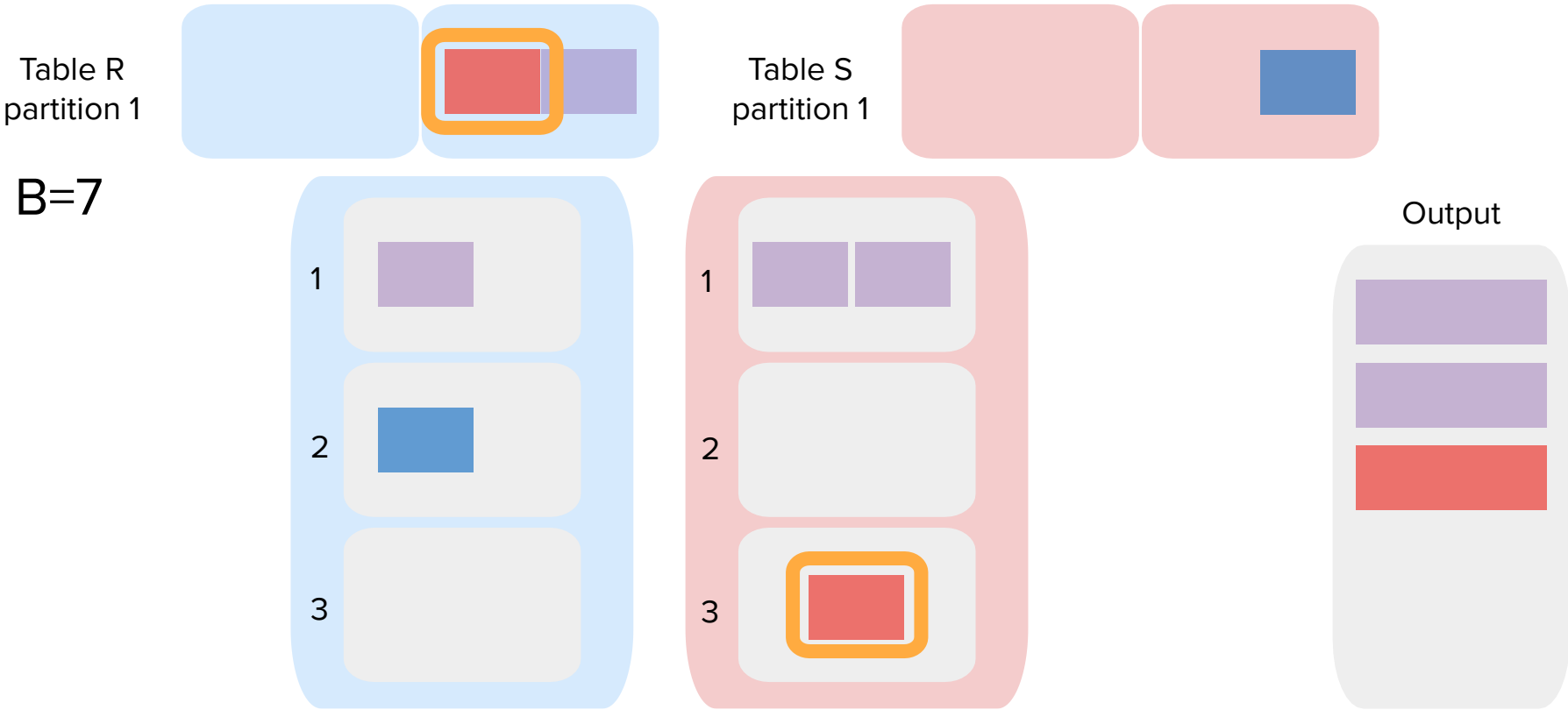
Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3



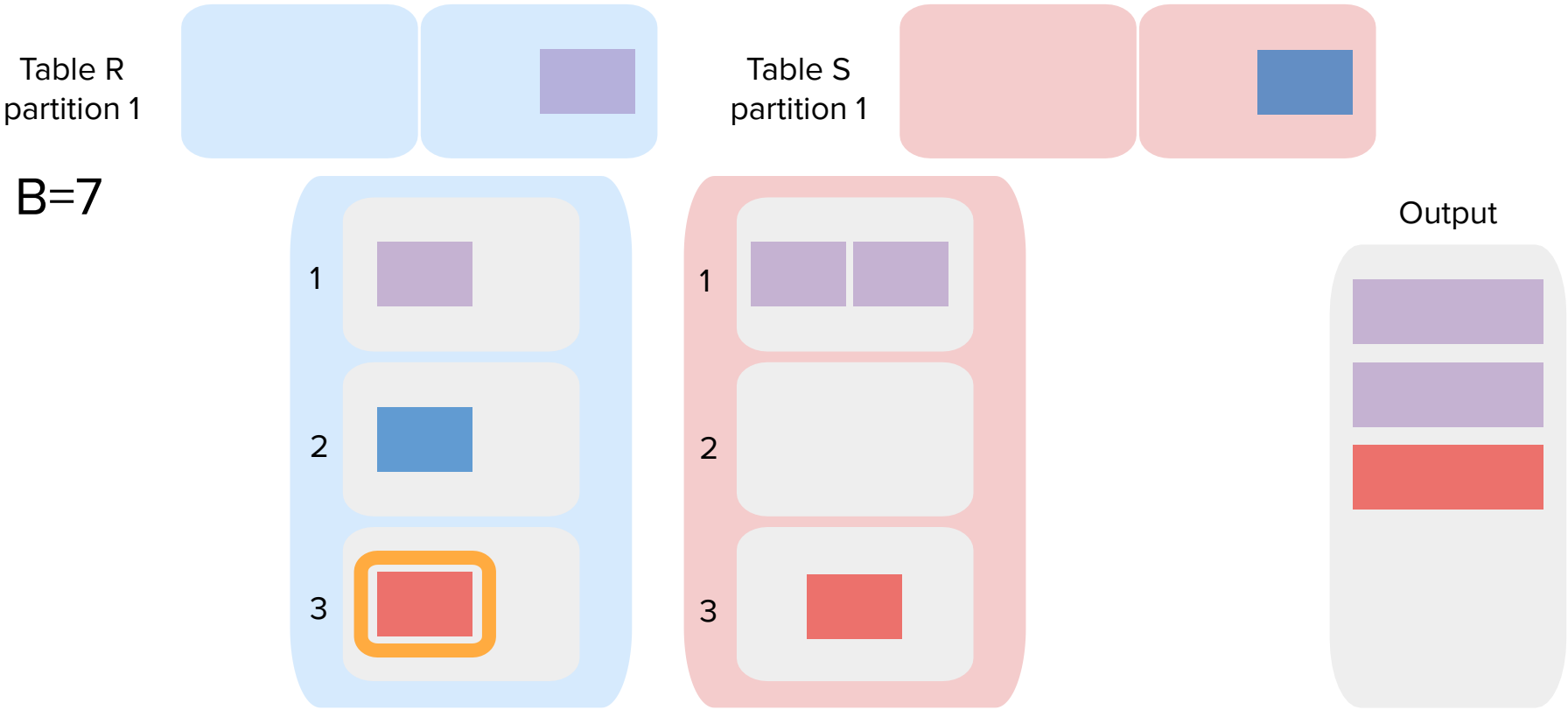
Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3



Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3

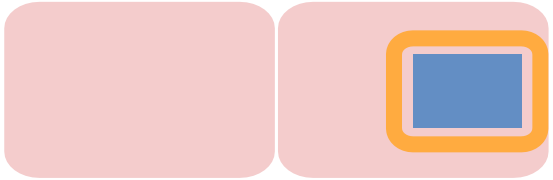


Symmetric Hash Join Example:

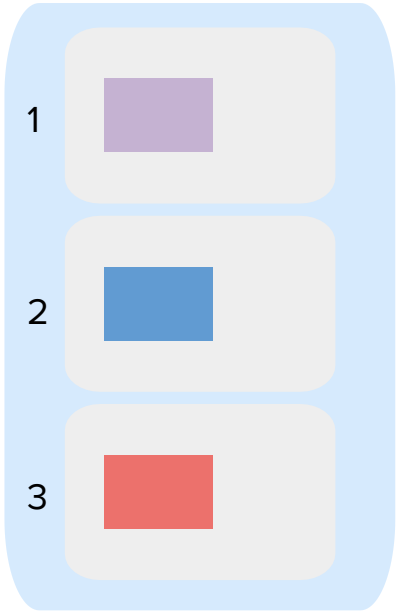
Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3

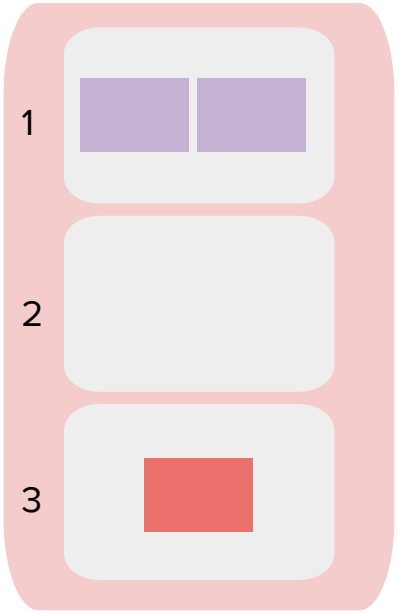
Table R partition 1

Table S partition 1

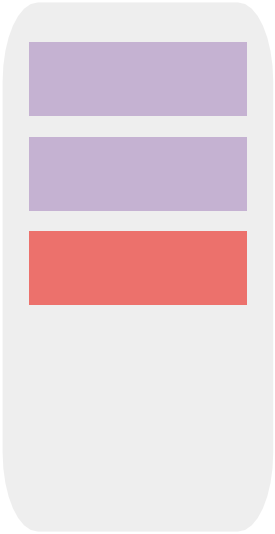


B=7





Output

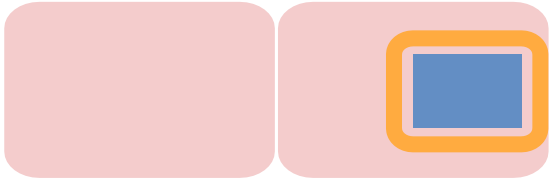


Symmetric Hash Join Example:

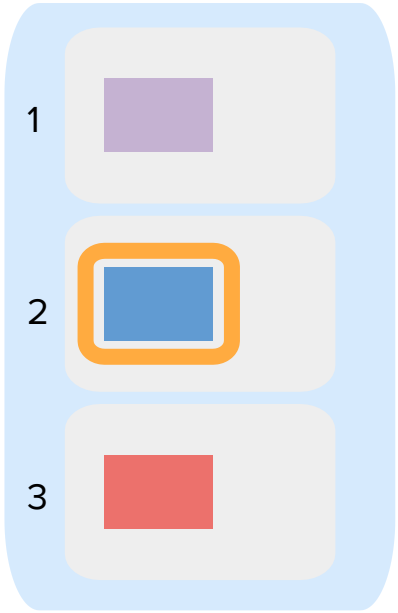
Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3

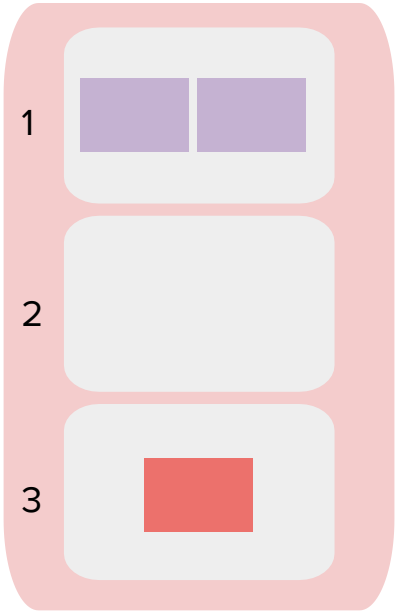
Table R partition 1

Table S partition 1

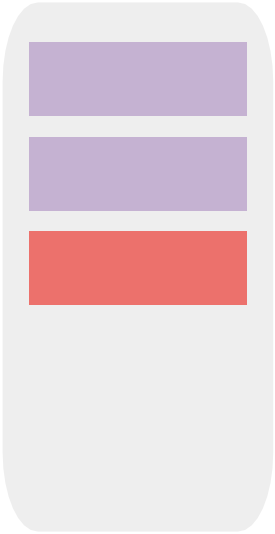


B=7





Output



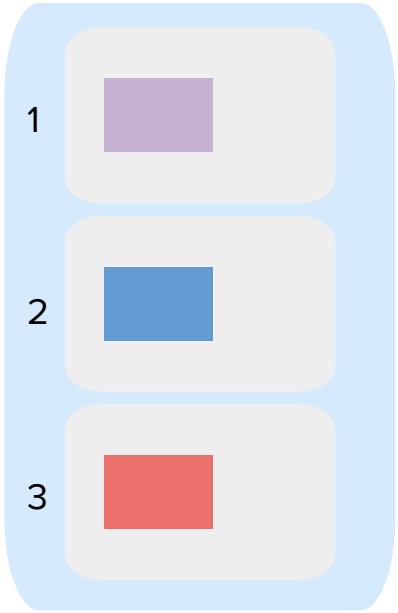
Symmetric Hash Join Example:

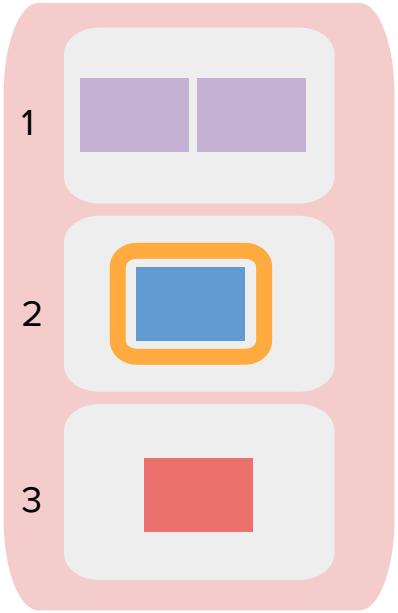
Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3

Table R partition 1

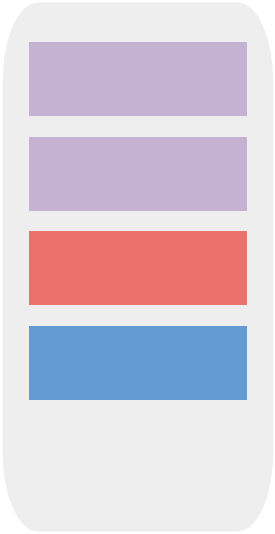
Table S partition 1

B=7





Output



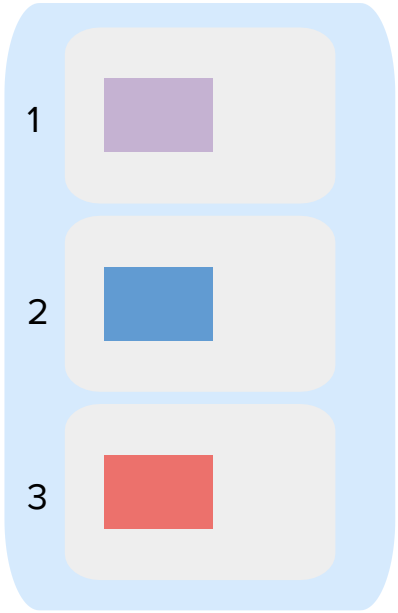
Symmetric Hash Join Example:

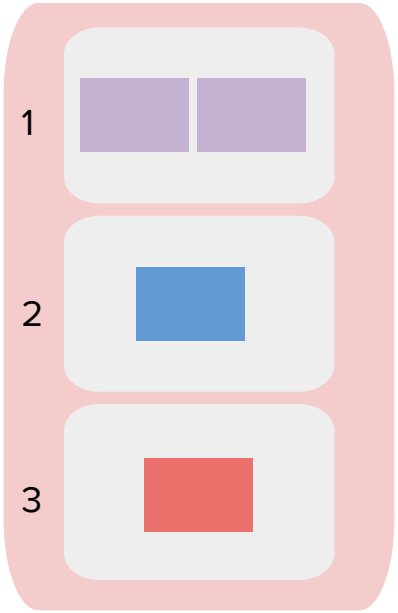
Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3

Table R partition 1

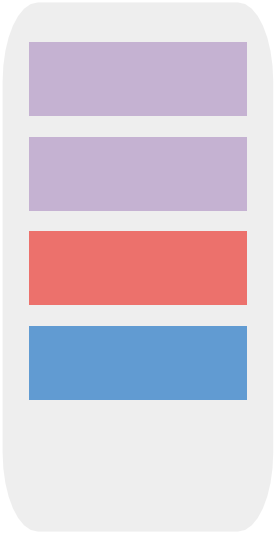
Table S partition 1

B=7





Output



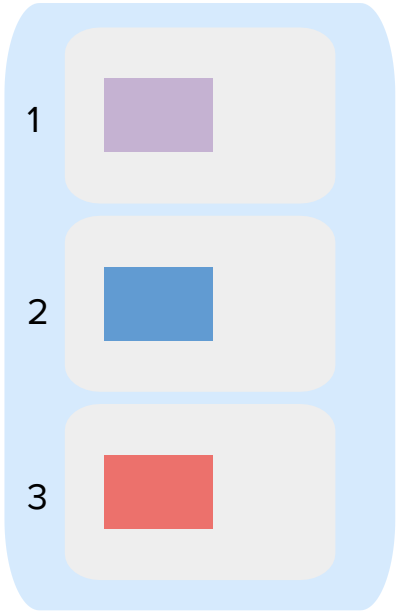
Symmetric Hash Join Example:

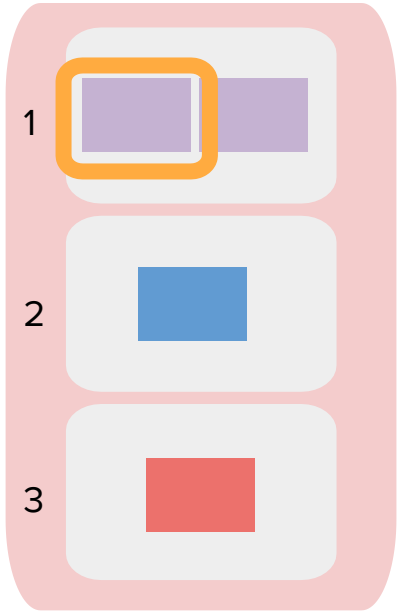
Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3

Table R partition 1

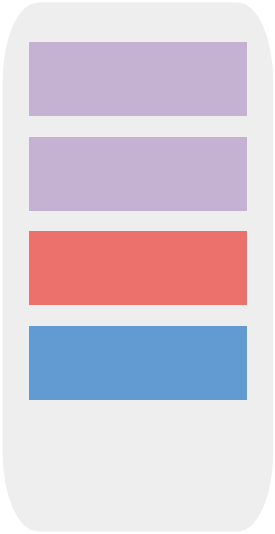
Table S partition 1

B=7



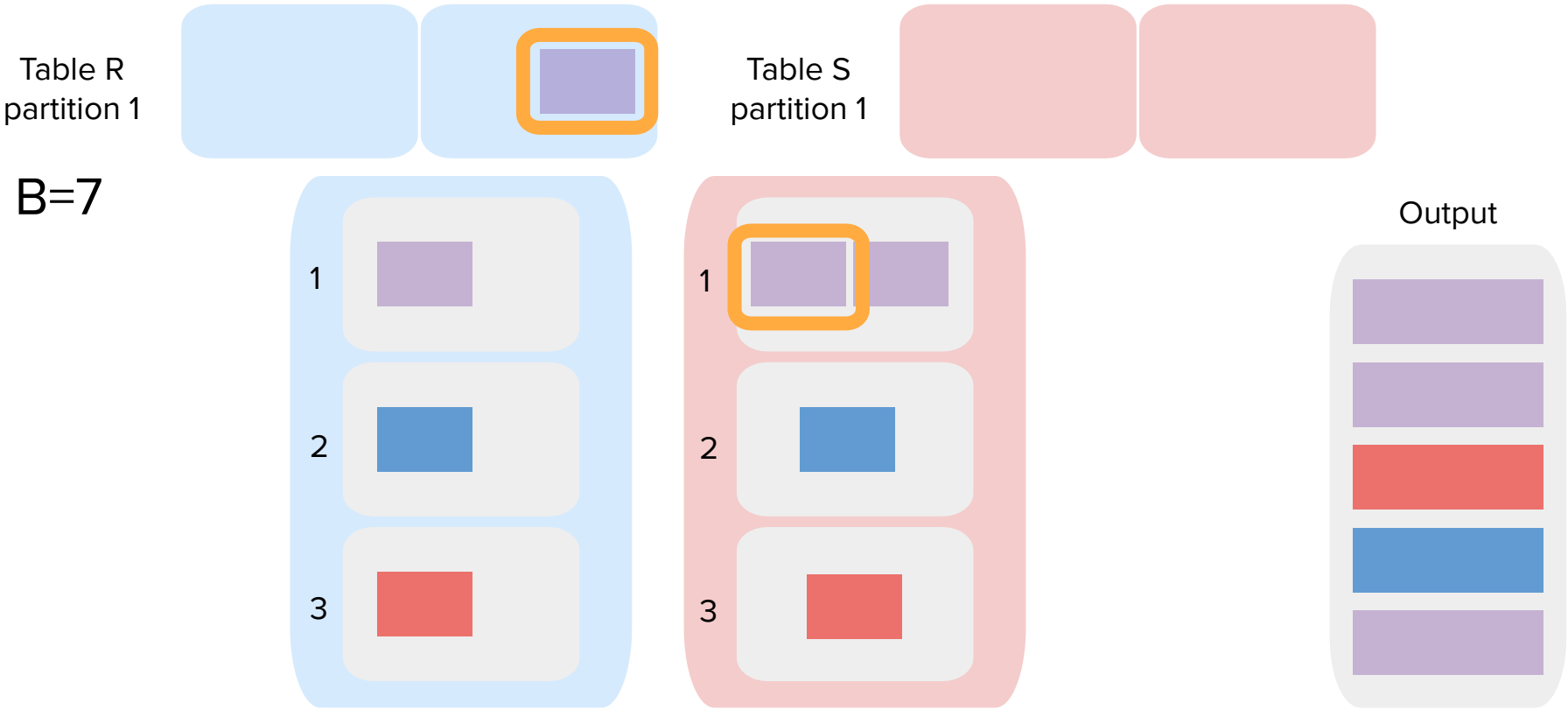


Output



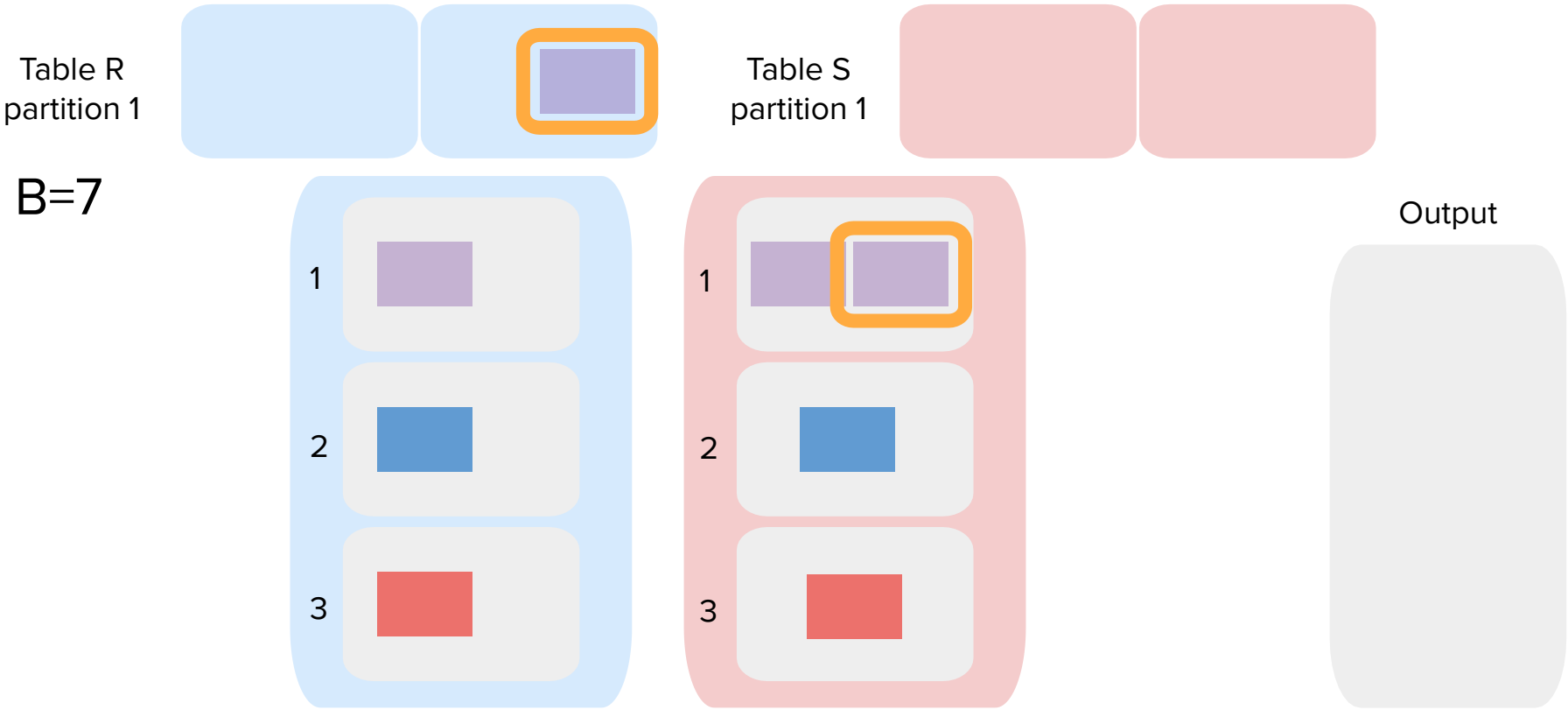
Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3



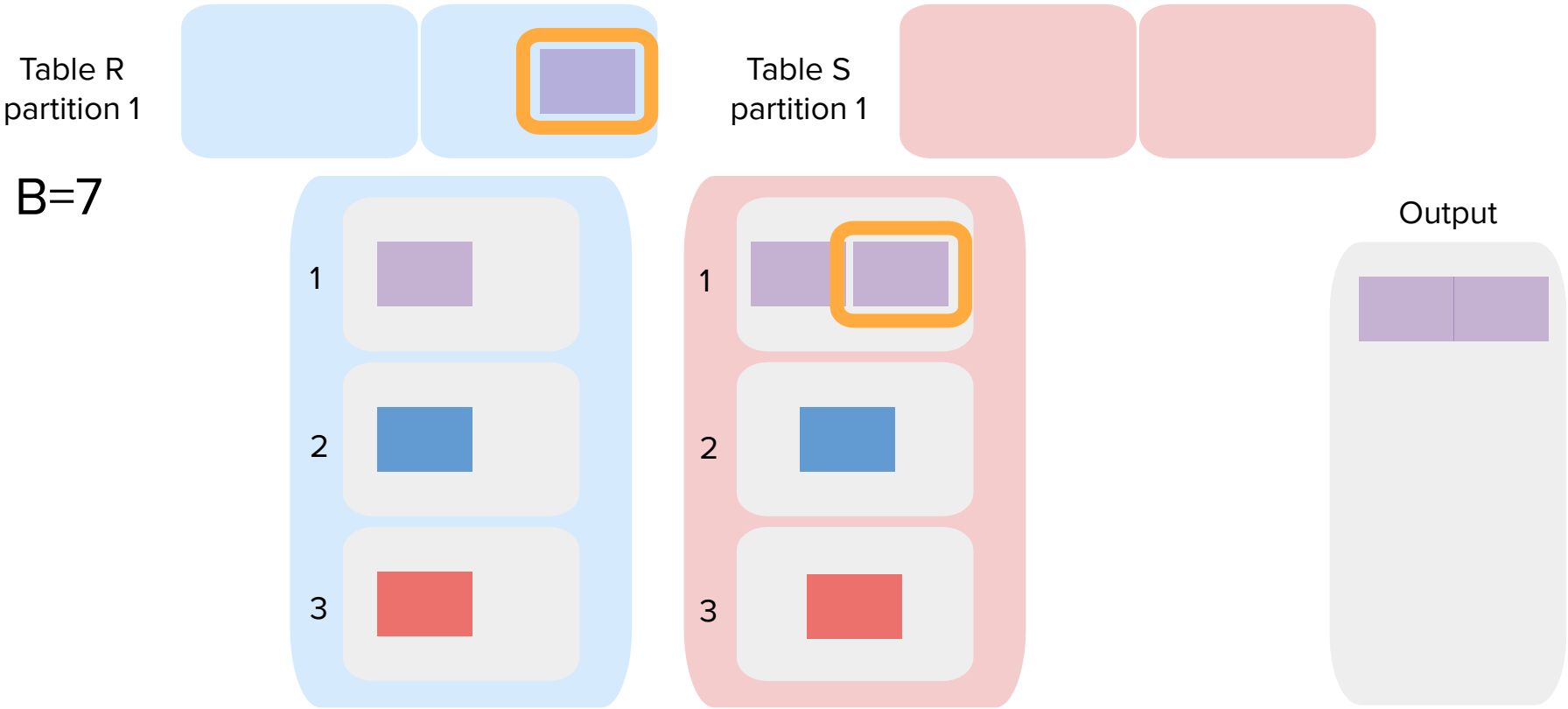
Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3



Symmetric Hash Join Example:

Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3



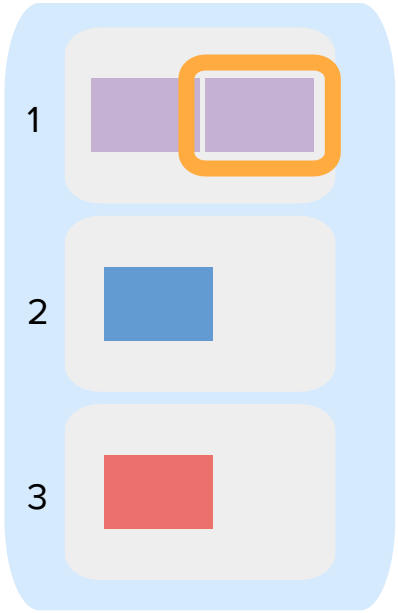
Symmetric Hash Join Example:

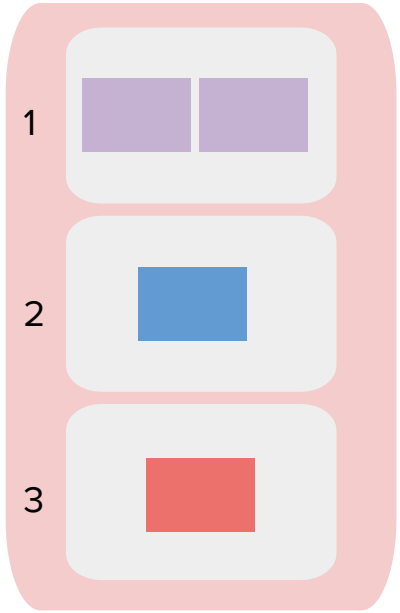
Assign colors to 3 partitions using our hash function:
{P} → 1, {B} → 2, {R} → 3

Table R partition 1

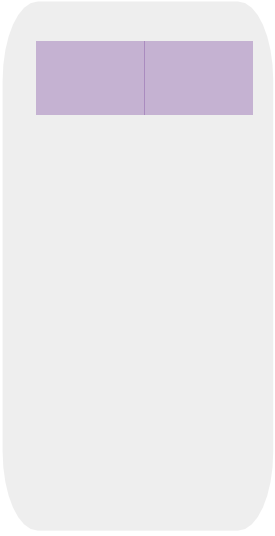
Table S partition 1

B=7





Output



Worksheet #4a

Given m=3 machines with B=5 buffer pages each, along with N=63 pages of data that don’t contain duplicates.

In the best case, what is the number of passes needed to sort the data? (Range/hash partitioning counts as 1 pass)

Worksheet #4a

With m=3 machines, B=5 buffer pages each, N=63 pages of data that don't contain duplicates: in the best case, what is the number of passes needed to sort the data?

# of passes to sort the data: (# of passes to partition the data) + (# of passes to sort each partition)

1 pass: Range partitioning data (Best case: data is uniformly partitioned, so each machine gets 21 pages.)

3 passes: Executing external sorting on each machine

- Pass 1: 21 pages => 4 runs with 5 pages, 1 run with 1 page
- Pass 2: Merge together 4 runs at a time => 2 runs
- Pass 3: Merge together to get 1 run of sorted data

Overall: 4 passes

Worksheet #4a

With m=3 machines, B=5 buffer pages each, N=63 pages of data that don't contain duplicates: in the best case, what is the number of passes needed to sort the data?

Generally, it will take $1 + \lceil 1 + \log_{B-1} \lceil N/mB \rceil \rceil$ passes to sort the data using parallel sorting in the best case.

Worksheet #4b

What is the number of passes needed to hash the data?

Examine the best case scenario, where the data will be uniformly distributed under the given hash functions.

Worksheet #4b

What is the number of passes needed to hash the data? Examine the best case scenario, where the data will be uniformly distributed under the given hash functions.

1 pass: Hash partitioning data (in best case, each machine gets 21 pages)

3 passes: Executing external hashing on each machine

- Pass 1: 21 pages => 4 partitions of ceil(21/4) = 6 pages each
- Pass 2: 4 partitions of 6 pages each => 16 partitions of 2 pages each
- Pass 3: Conquer phase, build in memory hash tables from partitions

Overall: 4 passes

Worksheet #4c

Assume that relation R has R pages of data, and relation S has S pages of data. If we have m machines with B buffer pages each, what is the number of passes in order to perform sort merge join (in terms of R, S, m, and B) in the best case?

Consider reading over either relation to be a pass.

Worksheet #4c

Assume that relation R has R pages of data, and relation S has S pages of data. If we have m machines with B buffer pages each, what is the number of passes in order to perform sort merge join (in terms of R, S, m, and B) in the best case? Consider reading over either relation to be a pass.

(1 pass/table to partition across machines) + (number of passes needed to sort R) + (number of passes to sort S) + (1 final merge sort pass, going through both tables)

$2 + \lceil 1 + \log_{B-1}\lceil R/(mB)\rceil\rceil + \lceil 1 + \log_{B-1}\lceil S/(mB)\rceil\rceil + 2$

Worksheet #4d

Can you use pipeline parallelism to implement this join (sort merge join from part c)?

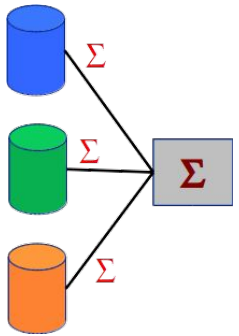
Worksheet #4d

Can you use pipeline parallelism to implement this join (sort merge join from part c)?

No, the sorting pass must complete before the merge pass can begin.

Parallel Aggregation

- To calculate aggregate functions (e.g. SUM, COUNT), we use **hierarchical aggregation**
  - Decompose aggregate into two parts: **global** and **local**
  - Apply local aggregate on each machine independently
  - Apply global aggregate on local aggregate values to get final aggregated value

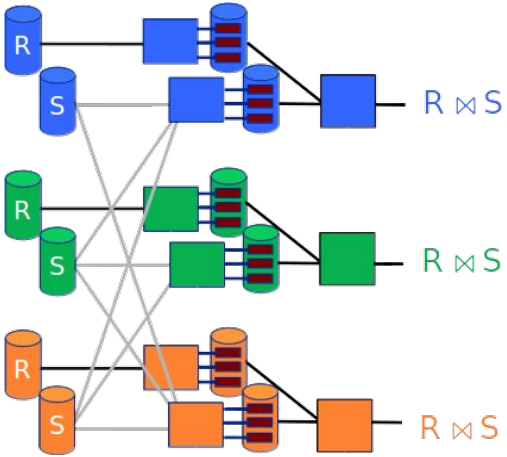


Parallel Aggregation

- To calculate aggregate functions (e.g. SUM, COUNT), we use **hierarchical aggregation**
  - AVERAGE(col)
    - local function calculates both sum and count and returns (Σ col, count(col))
    - global function takes these per-machine sums and counts, and combines: Σ((Σ col) / Σ(count(col)))

Asymmetric Shuffles

- Sometimes, data is already partitioned the way we want
  - May already be hash partitioned on a key, or range partitioned on a key
  - If relation is already partitioned the way we want, we don't need to repartition or send any data across the network for that relation.

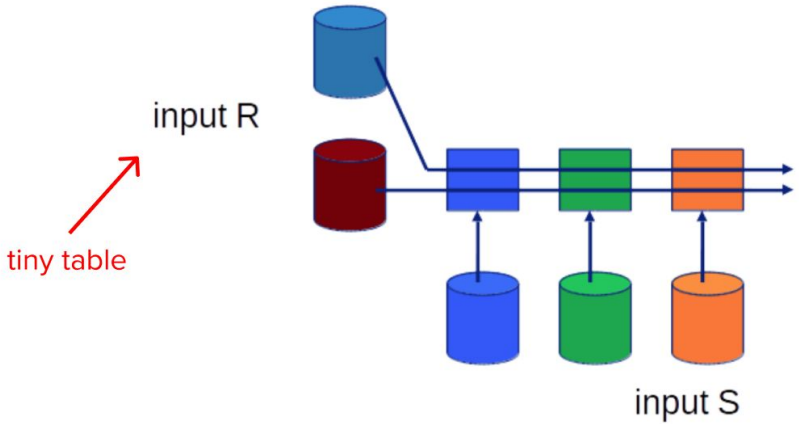


Asymmetric Shuffles

- Say we want to run Sort Merge Join on R and S
- R is already range partitioned over our m machines, and we remember what ranges were used to partition R
- We do not need to repartition R
- We will partition S with the same ranges we used to partition R and run SMJ locally on each machine

Broadcast joins

- Sometimes, one table is tiny and one is huge
  - Assume the huge table is not partitioned as we would like
  - May be much cheaper to send the entire tiny table to each machine than to partition the huge table



Worksheet #5a

Relation R has 10,000 pages, round-robin partitioned across 4 machines (M1, M2, M3, M4).

Relation S has 10 pages, all of which are only stored on M1. We want to join R and S on the condition R.col = C.col.

Assume the size of each page is 1 KB.

What type of join would be best in this scenario, and why?

Worksheet #5a

Relation R has 10,000 pages, round-robin partitioned across 4 machines (M1, M2, M3, M4).

Relation S has 10 pages, all of which are only stored on M1. We want to join R and S on the condition R.col = C.col.

Assume the size of each page is 1 KB.

What type of join would be best in this scenario, and why?

Broadcast join, because it is cheaper to send relation S to every machine rather than to partition R based on col.

Worksheet #5b

Relation R has 10,000 pages, round-robin partitioned across 4 machines (M1, M2, M3, M4).

Relation S has 10 pages, all of which are only stored on M1. We want to join R and S on the condition R.col = C.col.

Assume the size of each page is 1 KB.

How many KB of data must be sent over the network to join R and S?

Worksheet #5b

Relation R has 10,000 pages, round-robin partitioned across 4 machines (M1, M2, M3, M4).

Relation S has 10 pages, all of which are only stored on M1. We want to join R and S on the condition R.col = C.col.

Assume the size of each page is 1 KB.

How many KB of data must be sent over the network to join R and S?

The amount of data sent over the network is the amount of data required to send all pages of S to every machine that does not have it (M2, M3, and M4): 3 * 10 = 30 KB

Worksheet #5c

Relation R has 10,000 pages, round-robin partitioned across 4 machines (M1, M2, M3, M4).

Relation S has 10 pages, all of which are only stored on M1. We want to join R and S on the condition R.col = C.col.

Assume the size of each page is 1 KB.

Would the amount of data sent over the network change if R was hash partitioned among the 4 machines rather than round-robin partitioned? What if R was instead range partitioned? Assume each machine contains at least 1 tuple of R after partitioning.

Worksheet #5c

Would the amount of data sent over the network change if R was hash partitioned among the 4 machines rather than round-robin partitioned? What if R was instead range partitioned? Assume each machine contains at least 1 tuple of R after partitioning.

- If using broadcast join, network cost would remain the same
  - Still need to send S to all machines where R's tuples are located
- If R was hash partitioned, might be able to get lower network cost using parallel Grace Hash Join (if we remember hash function used to partition R). Note that this would be an asymmetric shuffle.
- If R was range partitioned, might be able to get lower network cost using parallel Sort Merge Join (if we remember ranges used to partition R). Note that this would be an asymmetric shuffle.

Attendance Link

https://cs186berkeley.net/attendance