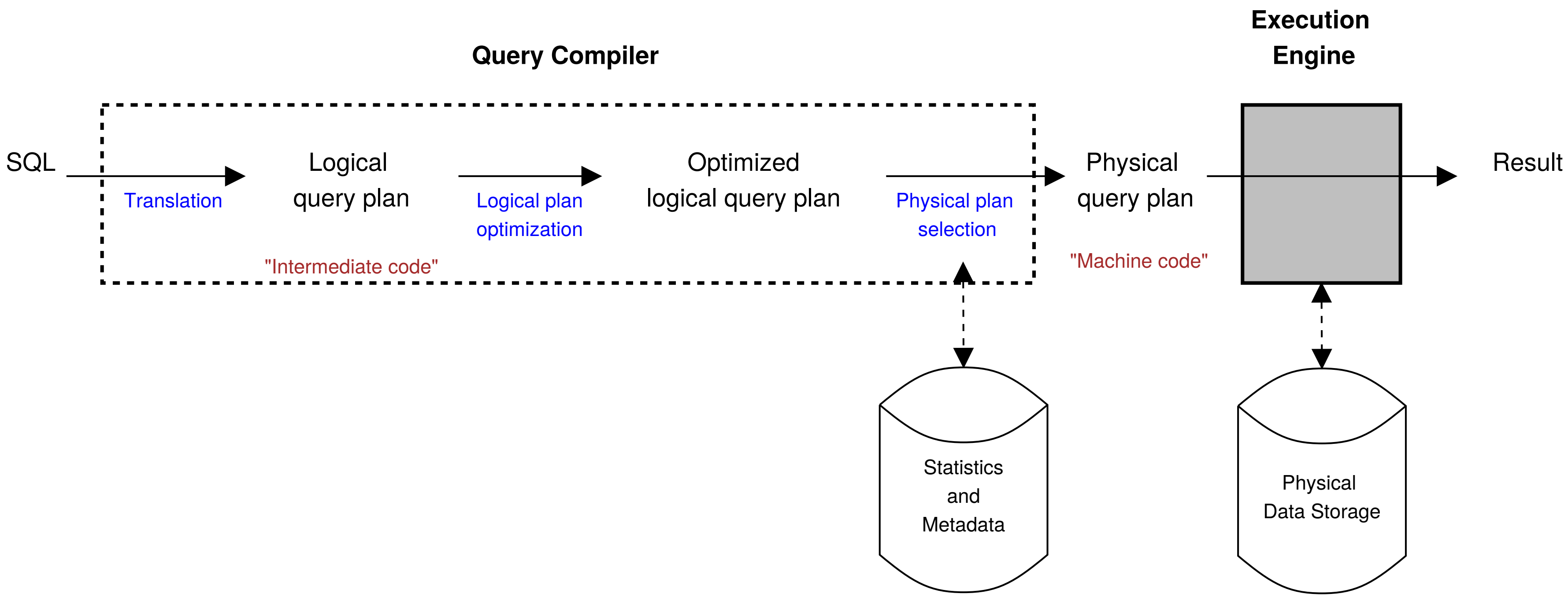
Physical Operators

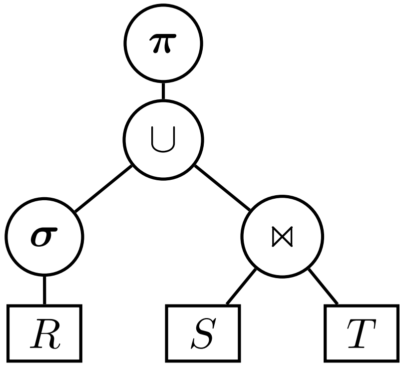
Scanning, sorting, merging, hashing

Physical Operators



Physical Operators

A logical query plan is essentially an execution tree



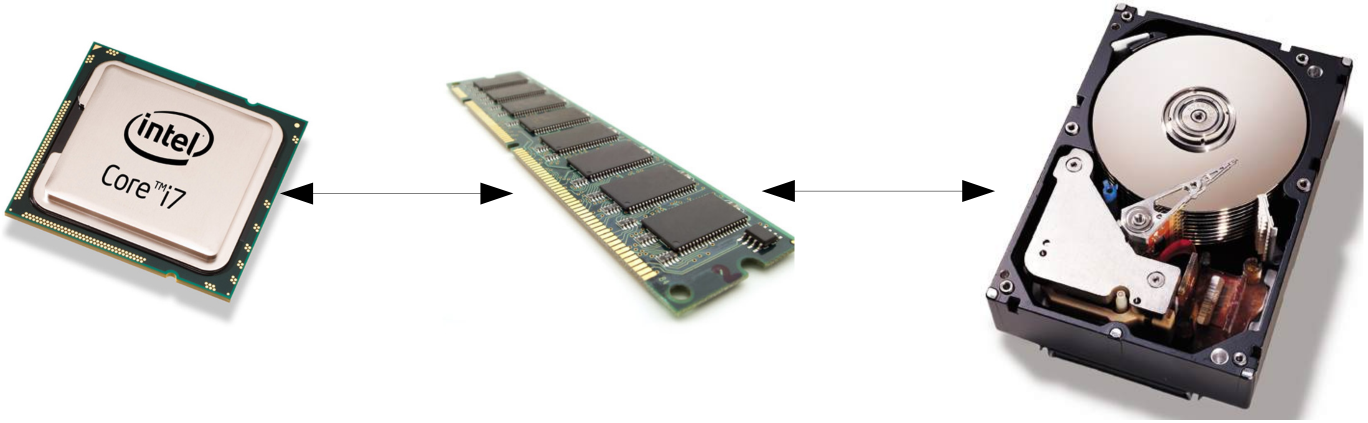
- To obtain a physical query plan we need to assign to each logical operator a physical implementation algorithm. We call such algorithms physical operators.
- In this lecture we study the various physical operators, together with their cost.

Physical Operators

Many implementations

- Each logical operator has multiple possible implementation algorithms
- No implementation is always better the others
- Hence we need to compare the alternatives on a case-by-case basis based on their costs

The I/O model of computation

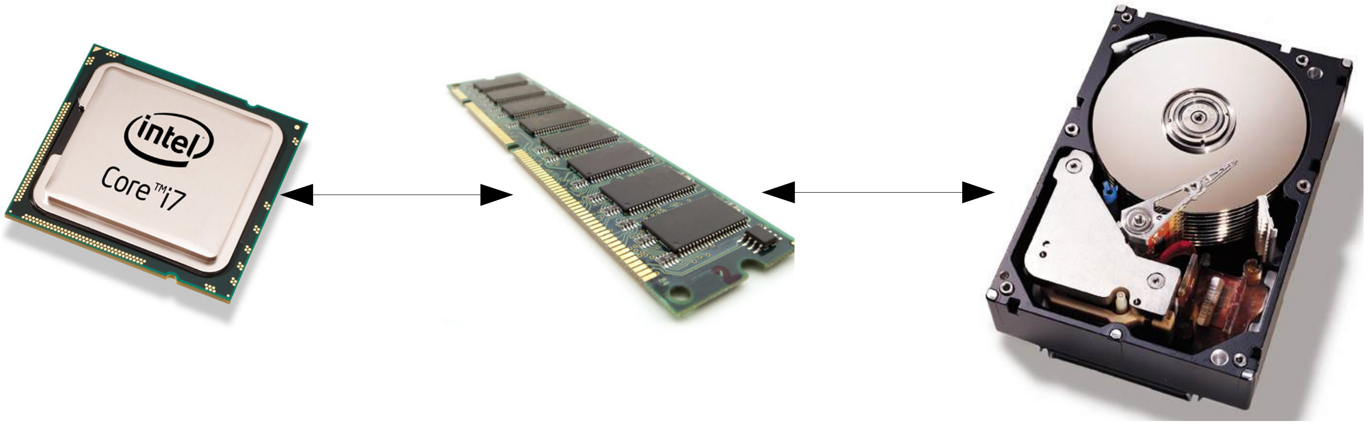


The I/O model

- Data is stored on disk, which is divided into blocks of bytes (typically 4 kilobytes)

  (each block can contain many data items)
- The CPU can only work on data items that are in memory, not on items on disk
- Therefore, data must first be transferred from disk to memory
- Data is transferred from disk to memory (and back) in whole blocks at the time
- The disk can hold D blocks, at most M blocks can be in memory at the same time (with $M << D$).

The I/O model of computation



- In-memory computation is fast (memory access $\approx 10^{-8}s$)
- Disk-access is slow (disk access: $\approx 10^{-3}s$)
- Hence: execution time is dominated by disk I/O

We will use the number of I/O operations required as cost metric

Physical Operators

To estimate the costs we will use the following parameters:

- $B(R)$: the number of blocks that R occupies on disk
- $T(R)$: the number of tuples in relation R
- $V(R, A_1, \ldots, A_n)$: the number of tuples in R that have distinct values for $A_1, \ldots, A_n$

  (i.e., $|\delta(\pi_{A_1,\ldots,A_n}(R)|$)
- M: the number of main memory buffers available

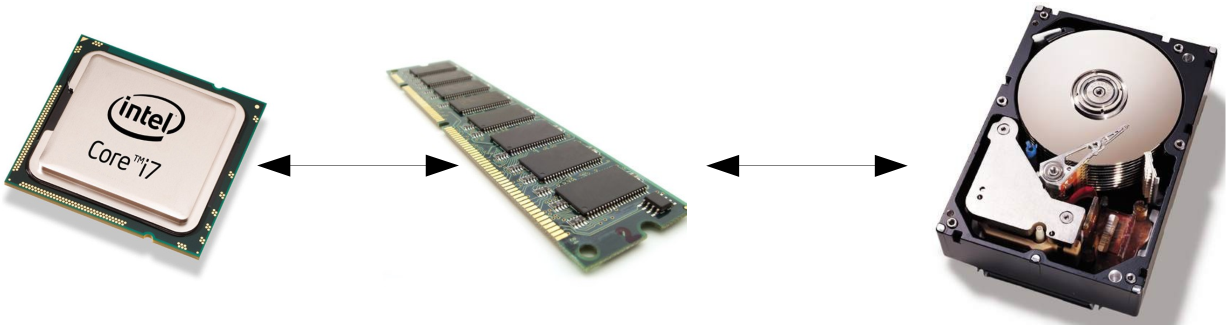
Statistics and the system catalog

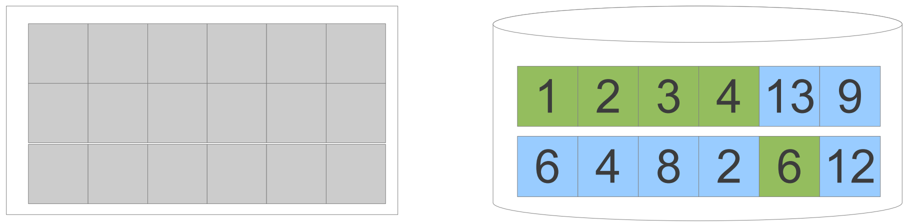
- The first three parameters are statistics that a DBMS stores in its system catalog
- These statistics are regularly collected

  (e.g., when required, at a scheduled time, . . .)

Physical Operators

Bag union $R \cup_B S$

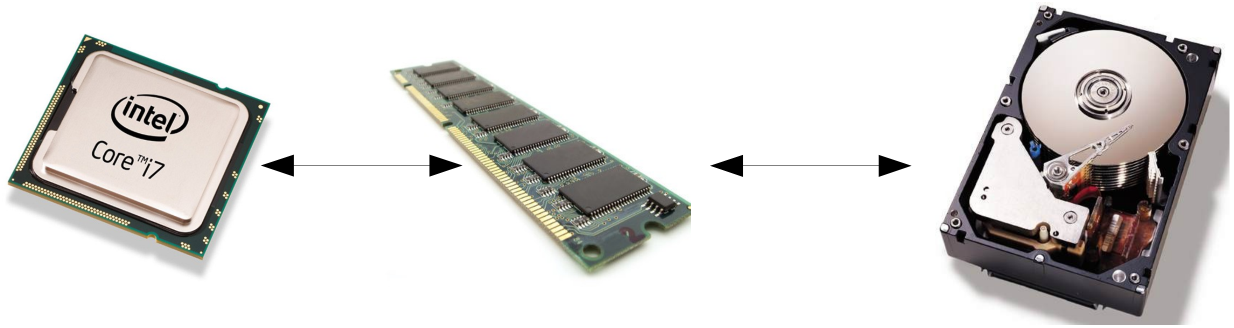


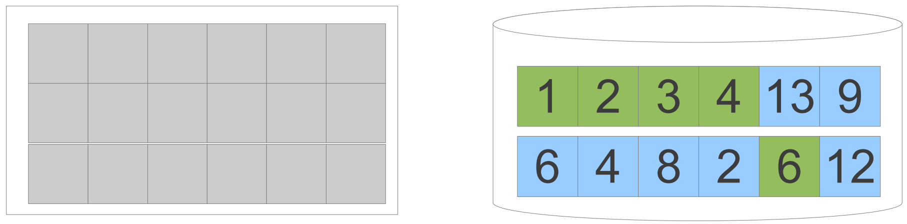


Relation R = green
Relation S = blue
1 integer per block

Physical Operators

Bag union $R \cup_B S$



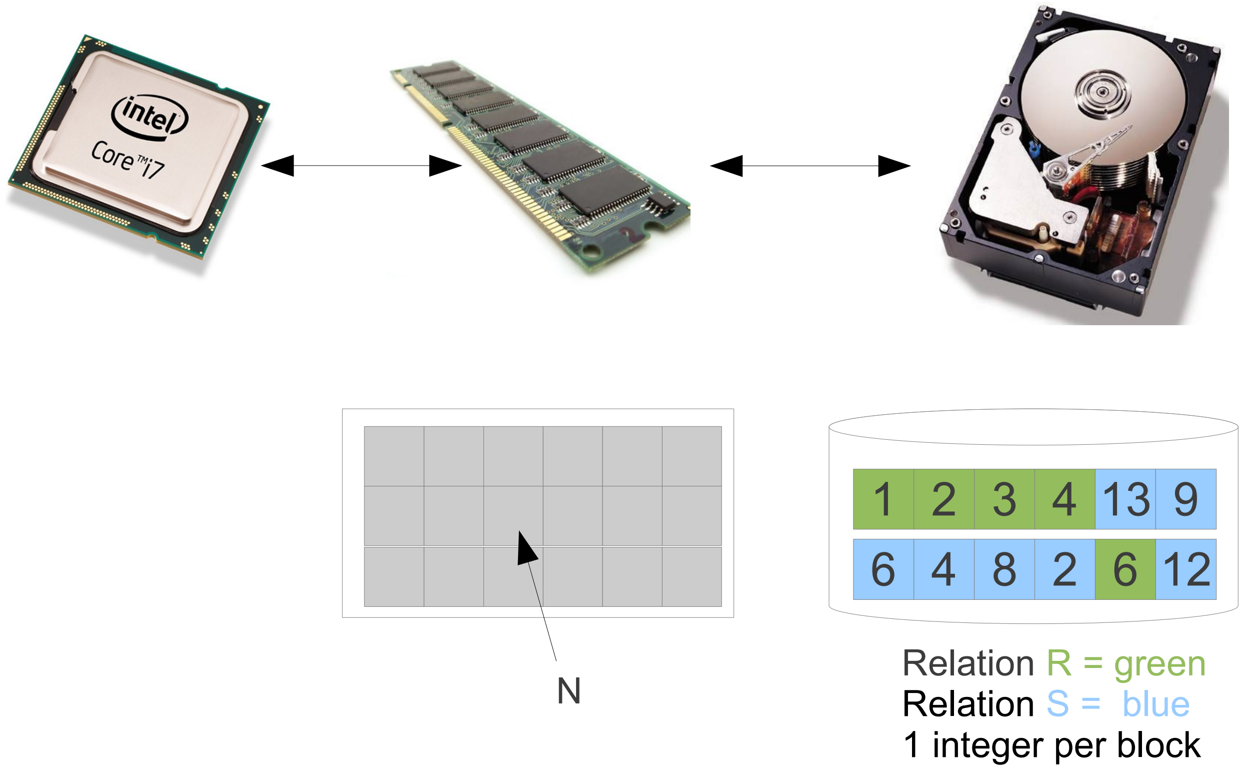


Relation R = green
Relation S = blue
1 integer per block

- Step 1: reserve 1 buffer frame, call this N

Physical Operators

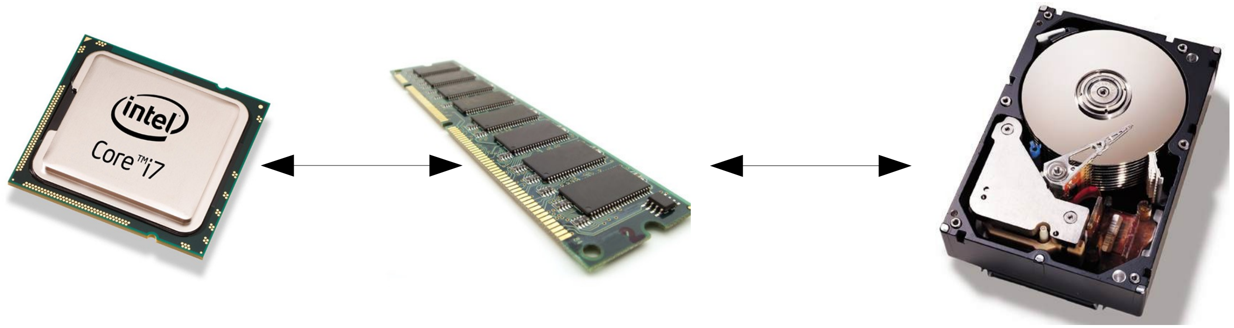
Bag union $R \cup_B S$

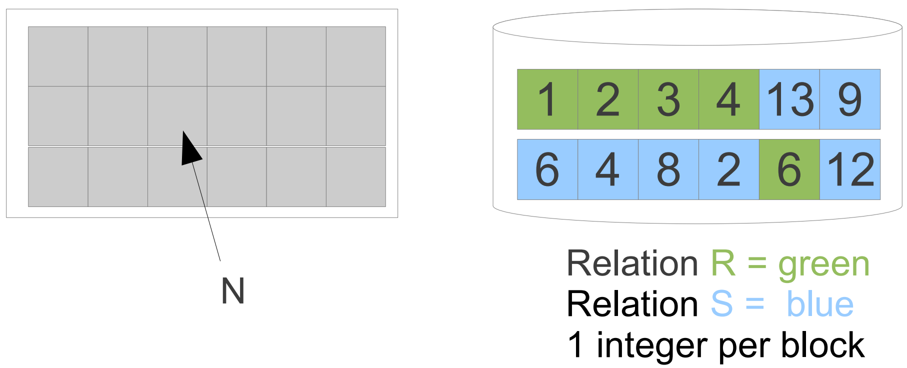


- Step 1: reserve 1 buffer frame, call this N

Physical Operators

Bag union $R \cup_B S$

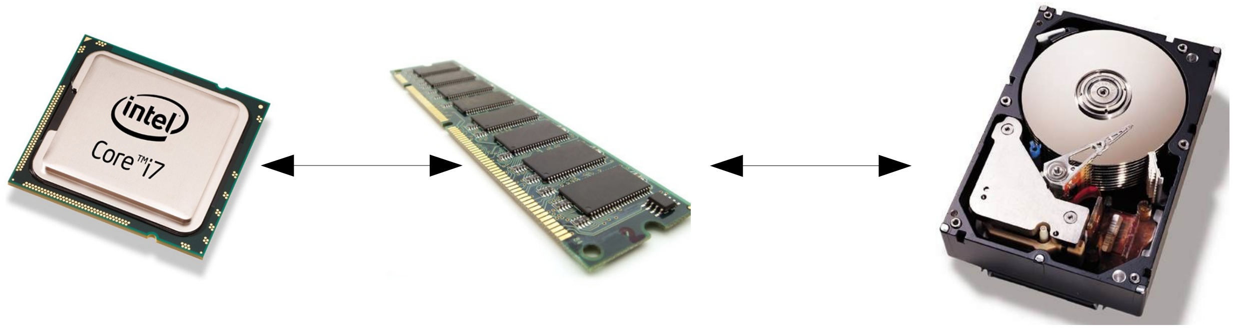


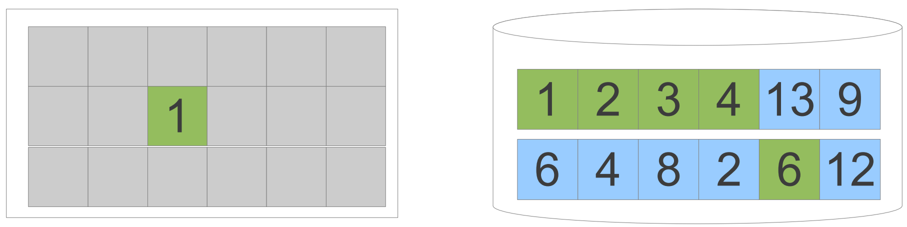


- Load 1st block of R into N, output all of its elements

Physical Operators

Bag union $R \cup_B S$



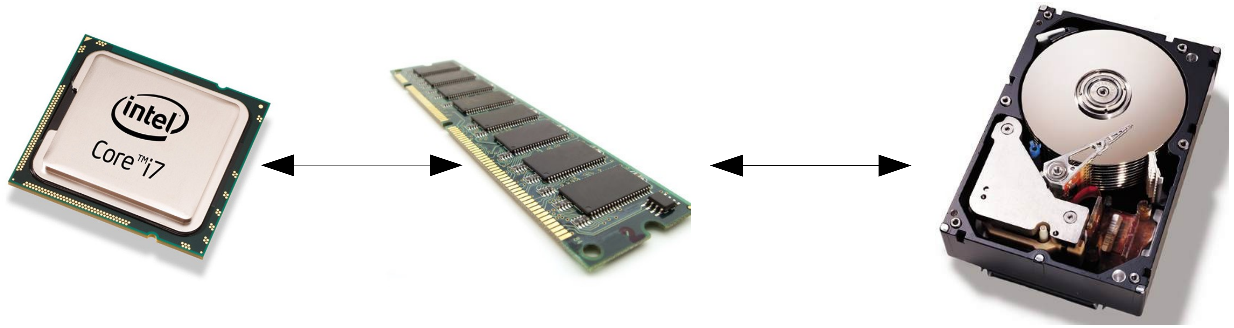


Relation R = green
Relation S = blue
1 integer per block

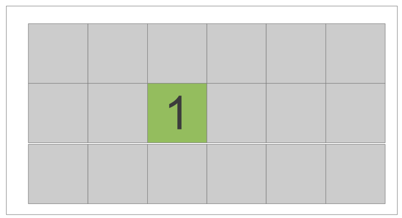
- Load 1st block of R into N, output all of its elements

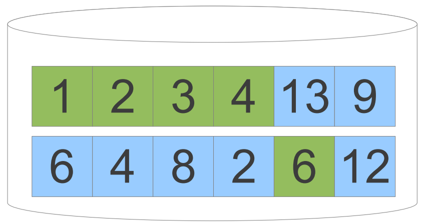
Physical Operators

Bag union $R \cup_B S$



Output:
1



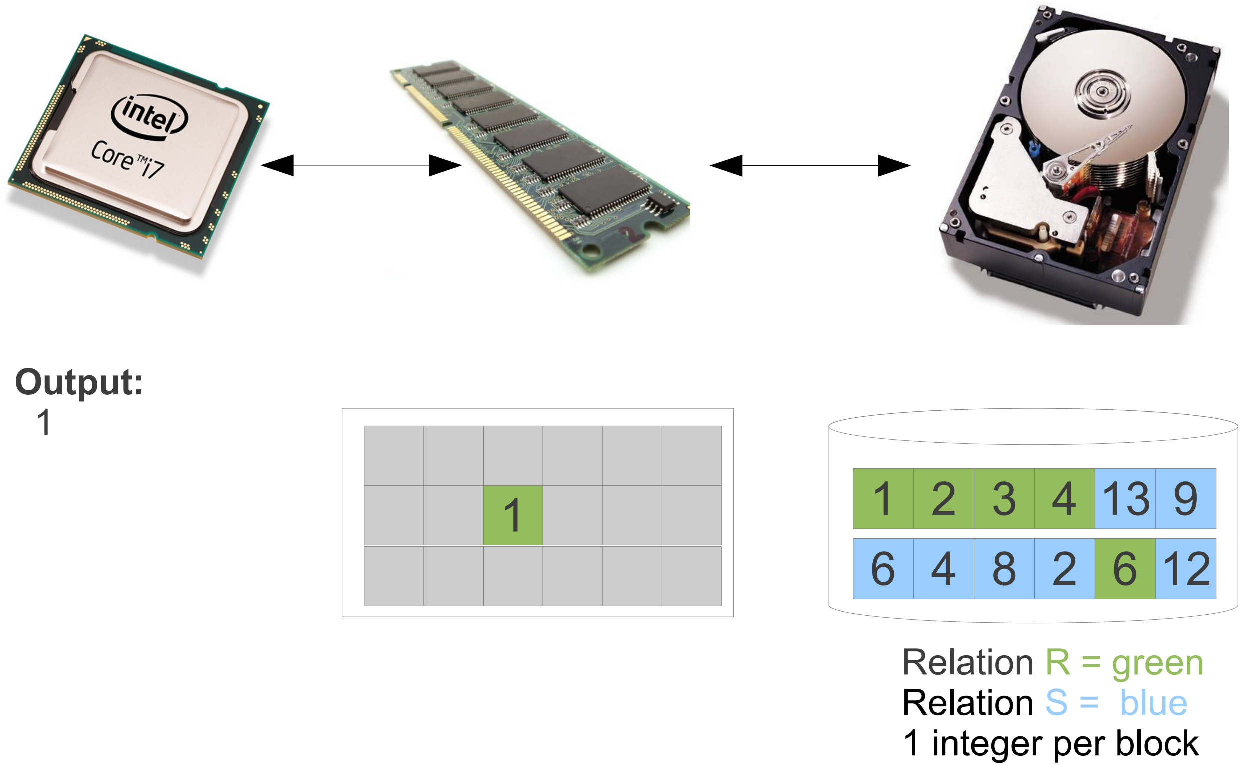


Relation R = green
Relation S = blue
1 integer per block

- Load 1st block of R into N, output all of its elements

Physical Operators

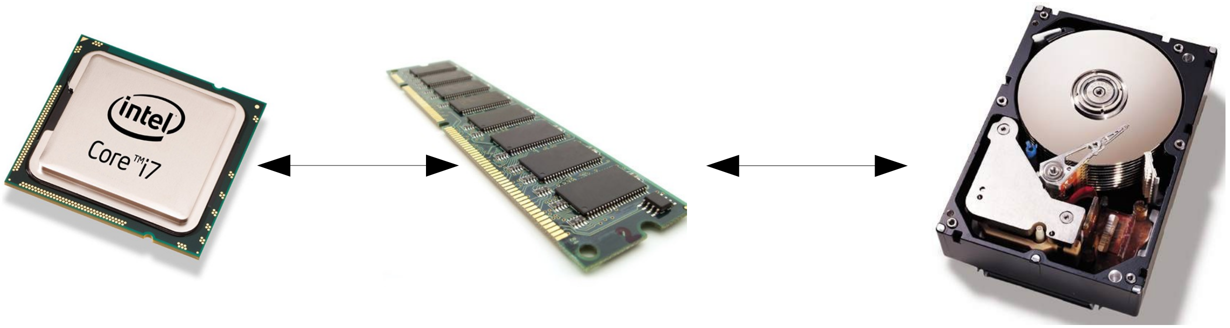
Bag union $R \cup_B S$



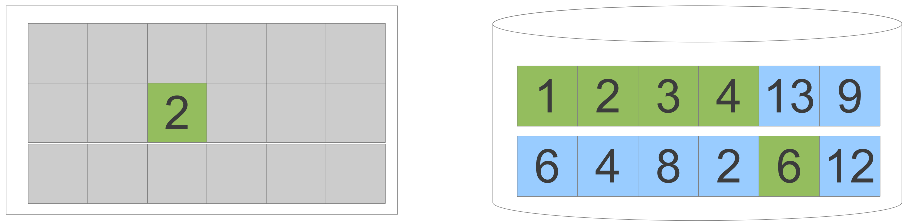
- Load 2nd block of R into N, output all of its elements

Physical Operators

Bag union $R \cup_B S$



Output:
1

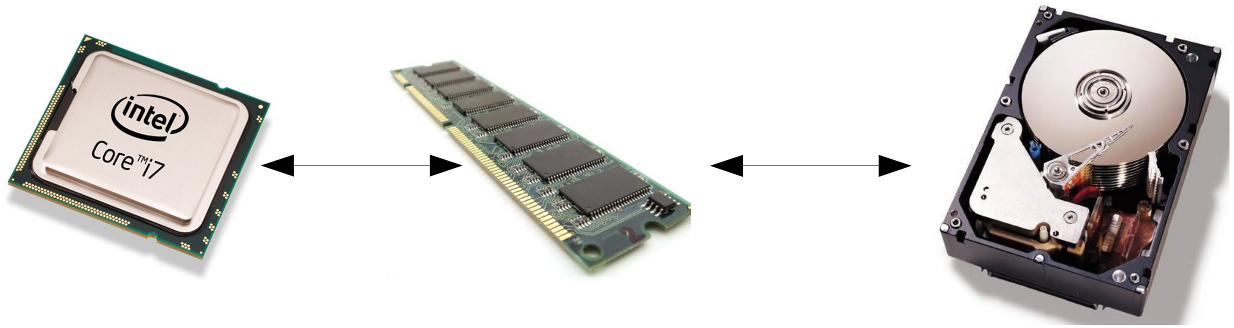


Relation R = green
Relation S = blue
1 integer per block

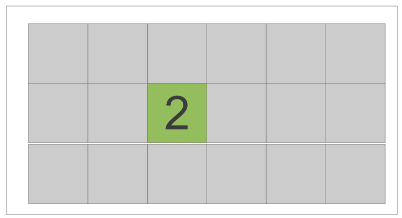
- Load 2nd block of R into N, output all of its elements

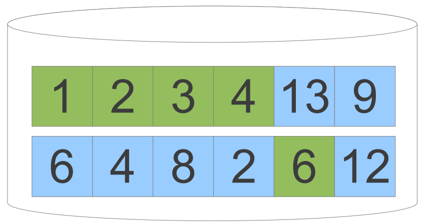
Physical Operators

Bag union $R \cup_B S$



Output:
1, 2



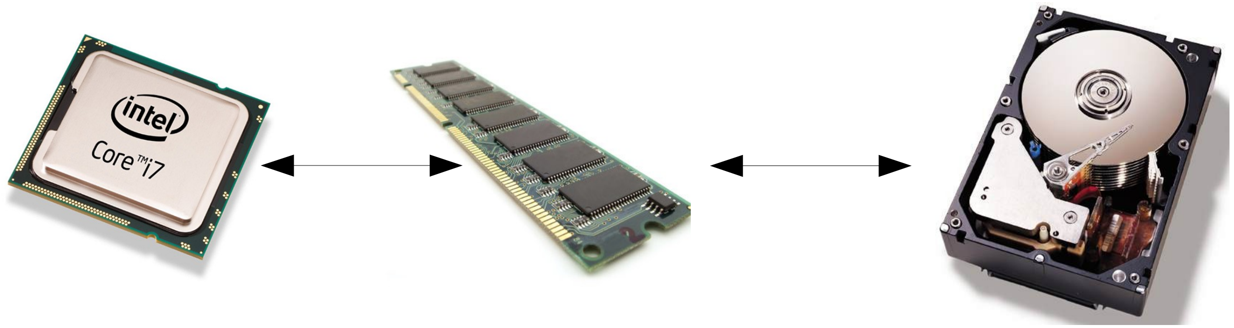


Relation R = green
Relation S = blue
1 integer per block

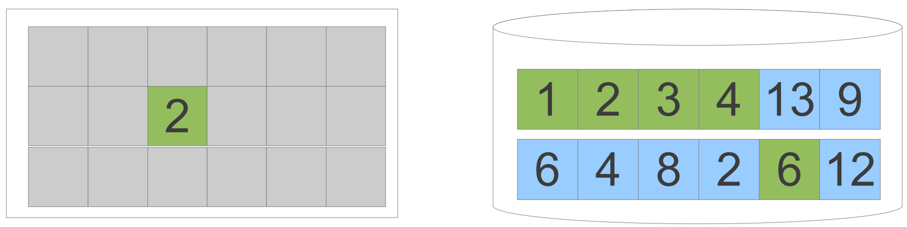
- Load 2nd block of R into N, output all of its elements

Physical Operators

Bag union $R \cup_B S$



Output:
1, 2

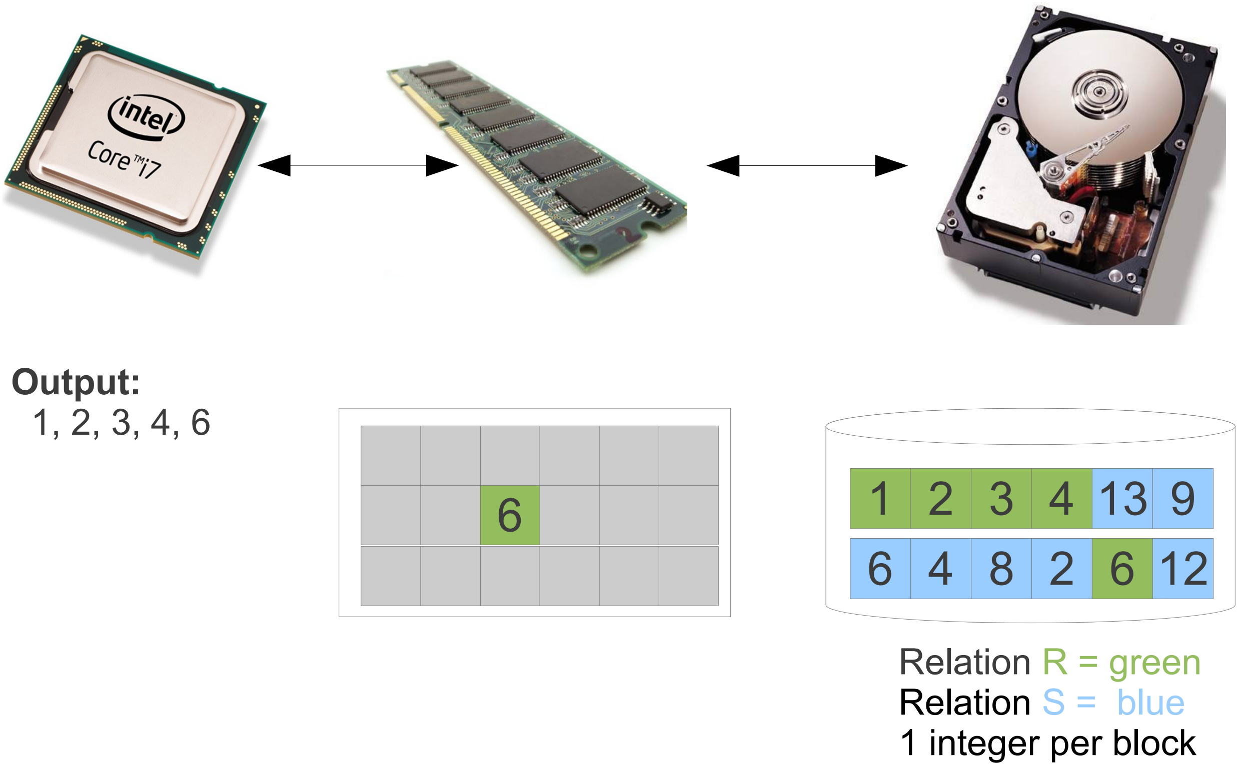


Relation R = green
Relation S = blue
1 integer per block

- . . . and repeat this for every block of R

Physical Operators

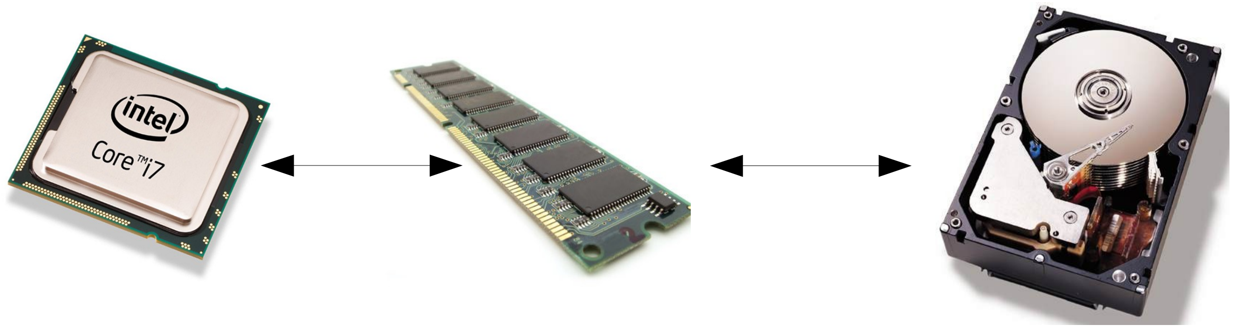
Bag union $R \cup_B S$



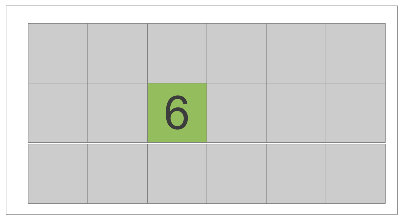
- . . . and repeat this for every block of R.

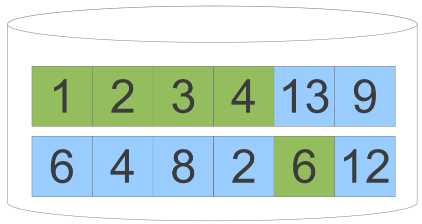
Physical Operators

Bag union $R \cup_B S$



Output:
1, 2, 3, 4, 6



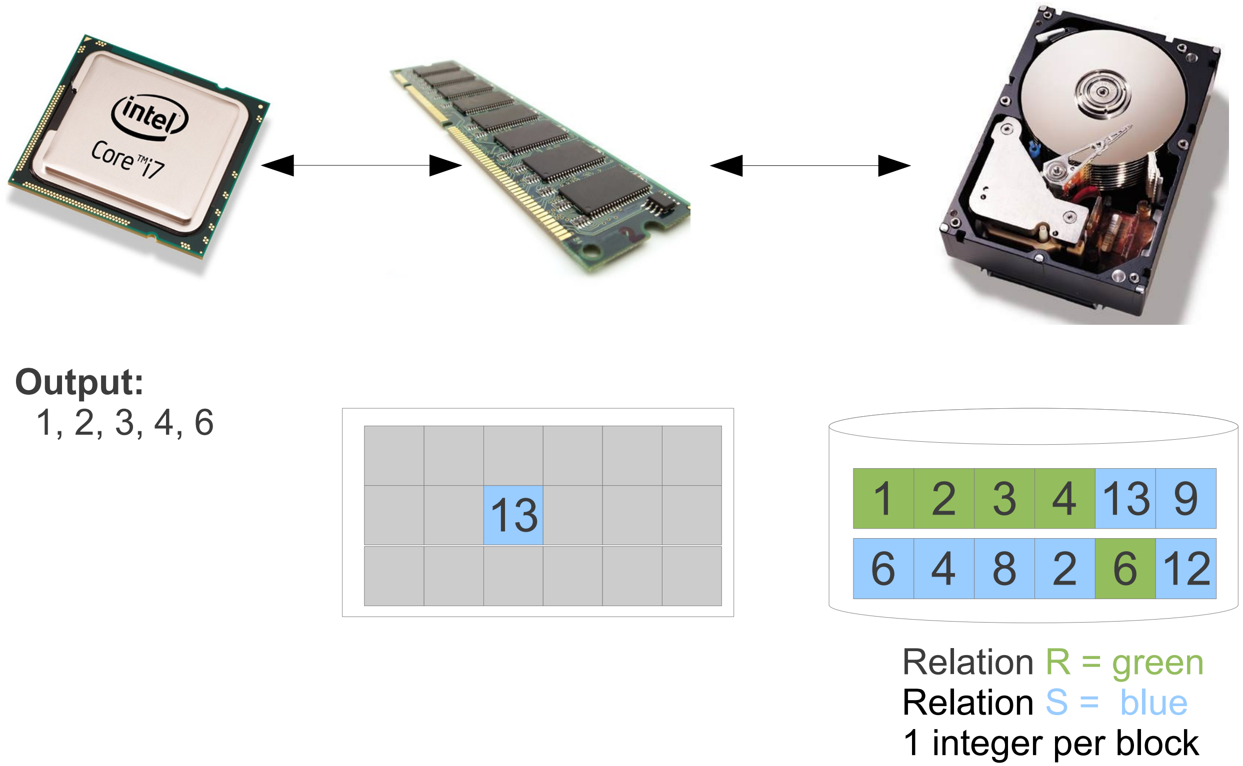


Relation R = green
Relation S = blue
1 integer per block

- Load 1st block of S into N, output all of its elements

Physical Operators

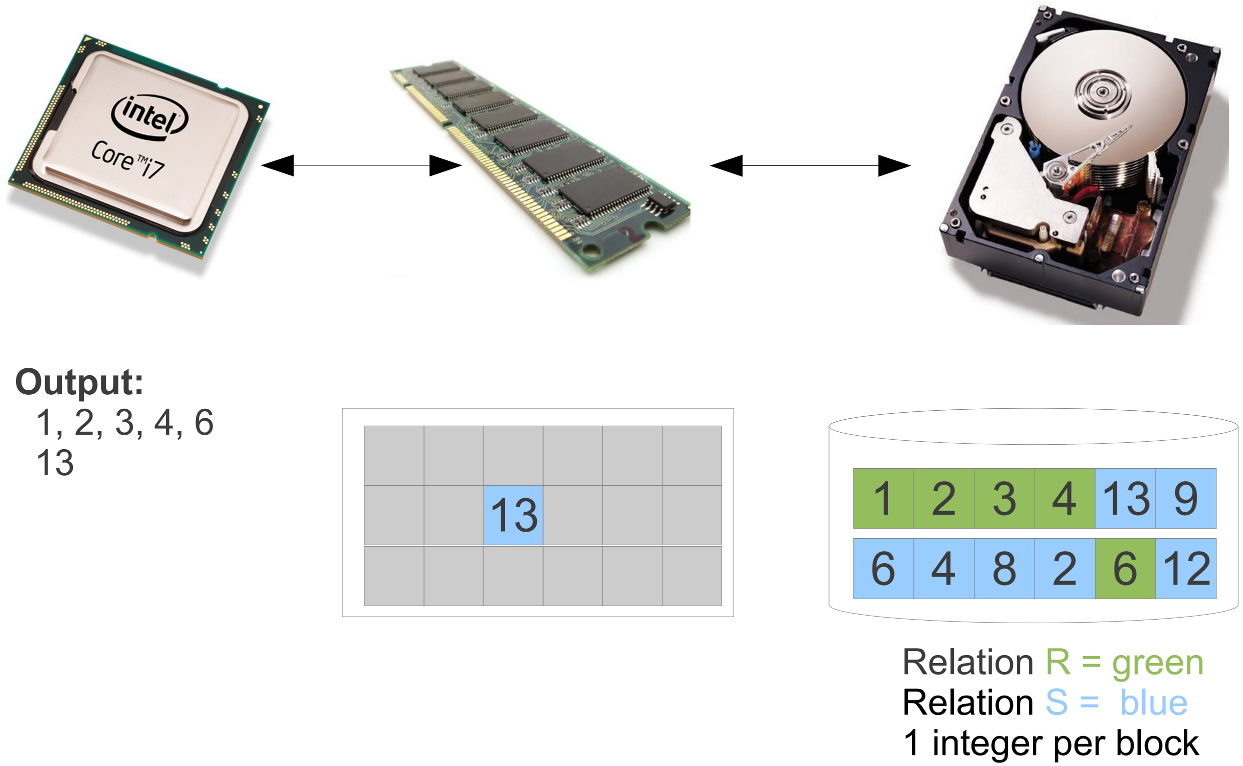
Bag union $R \cup_B S$



- Load 1st block of S into N, output all of its elements

Physical Operators

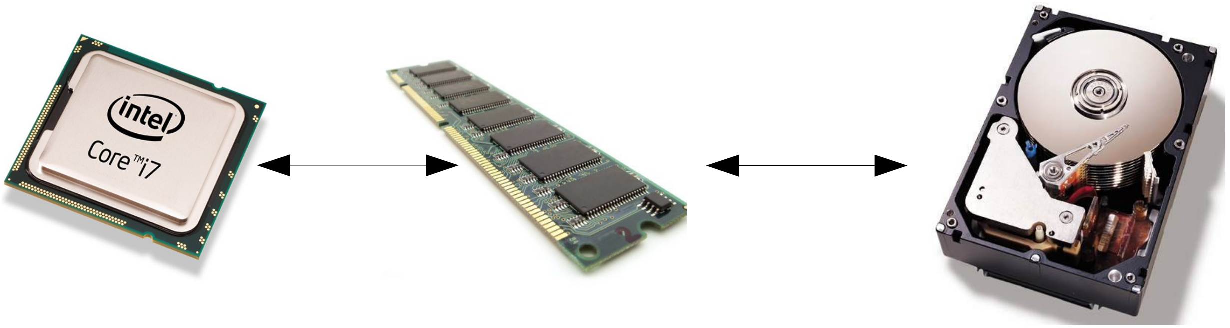
Bag union $R \cup_B S$



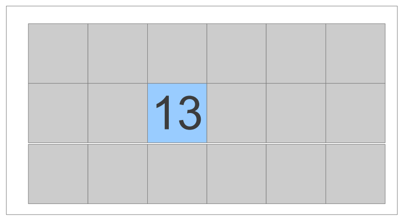
- Load 1st block of S into N, output all of its elements

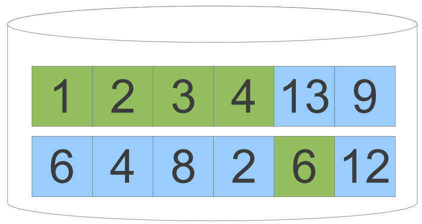
Physical Operators

Bag union $R \cup_B S$



Output:
1, 2, 3, 4, 6
13



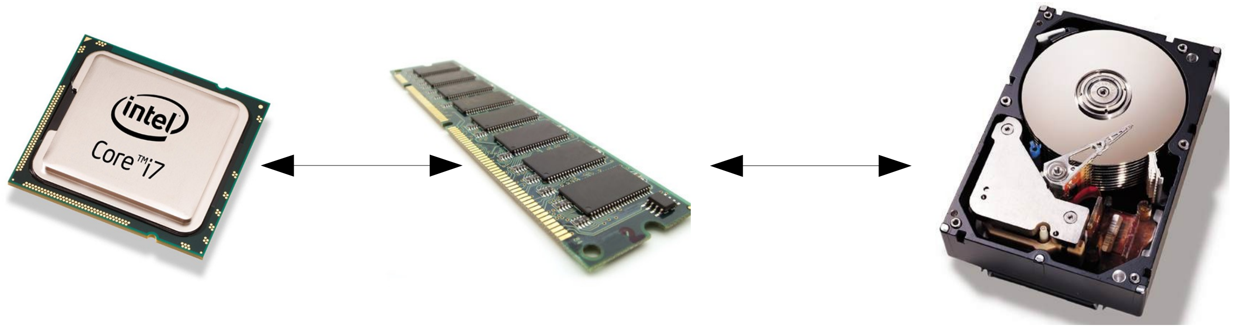


Relation R = green
Relation S = blue
1 integer per block

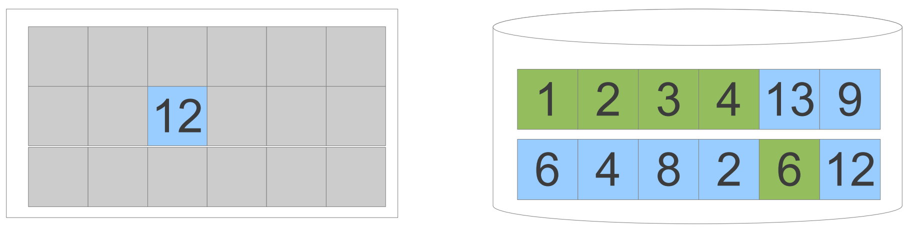
- . . . and repeat this until the last block of S

Physical Operators

Bag union $R \cup_B S$



Output:
1, 2, 3, 4, 6
13, 9, 6, 4, 8,
2, 12



Relation R = green
Relation S = blue
1 integer per block

- . . . and repeat this until the last block of S

Physical Operators

Bag union

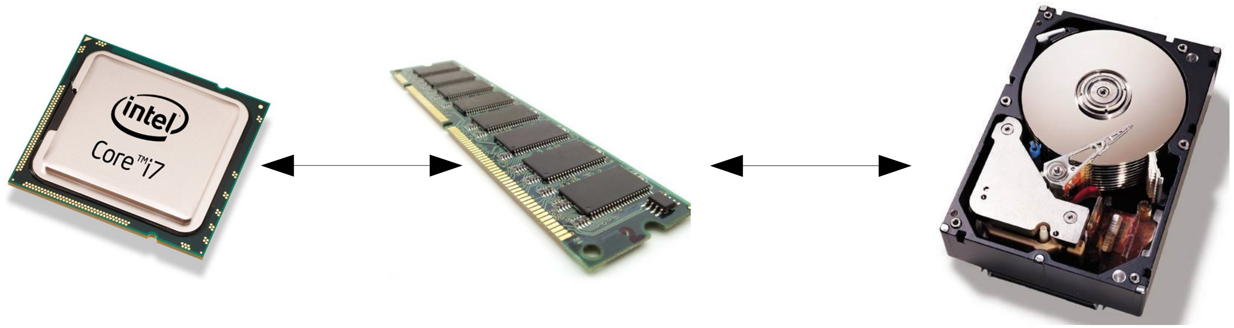
We can compute the bag union $R \cup_B S$ as follows:

for each block B_R in R **do**
    load B_R into buffer N;
    for each tuple t_R in N **do**
        output t_R;
for each block B_S in S **do**
    load B_S into buffer N;
    for each tuple t_S in N **do**
        output t_S;

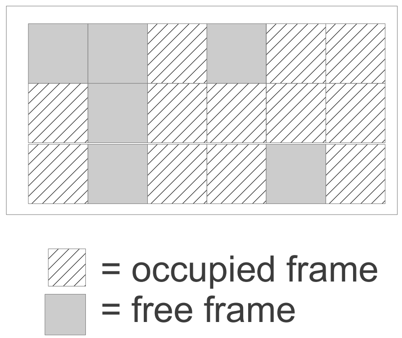
- Cost: $B(R) + B(S)$ I/O operations (we never count the output-cost)
- Requires that $M \geq 1$ (i.e., it can always be used)

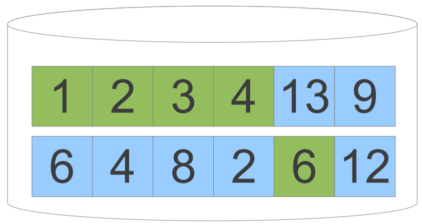
Physical Operators

One-pass set union $R \cup_S S$



Output:



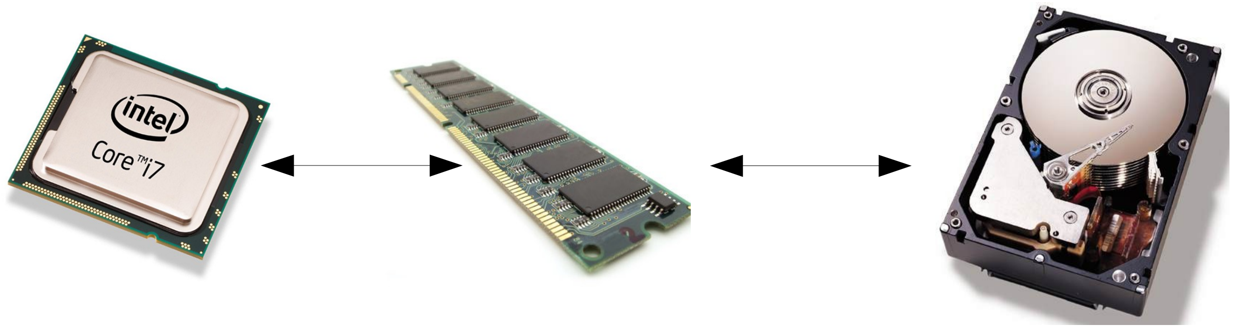


Relation R = green
Relation S = blue
1 integer per block

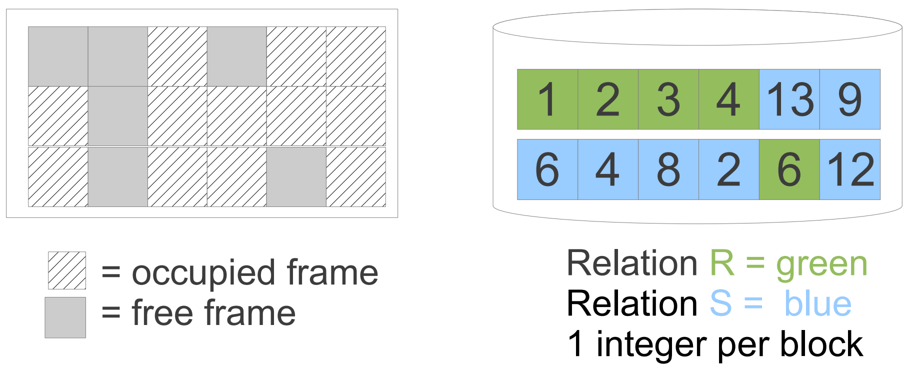
Assumption: we have $B(R) + 1$ free buffer frames

Physical Operators

One-pass set union $R \cup_S S$



Output:

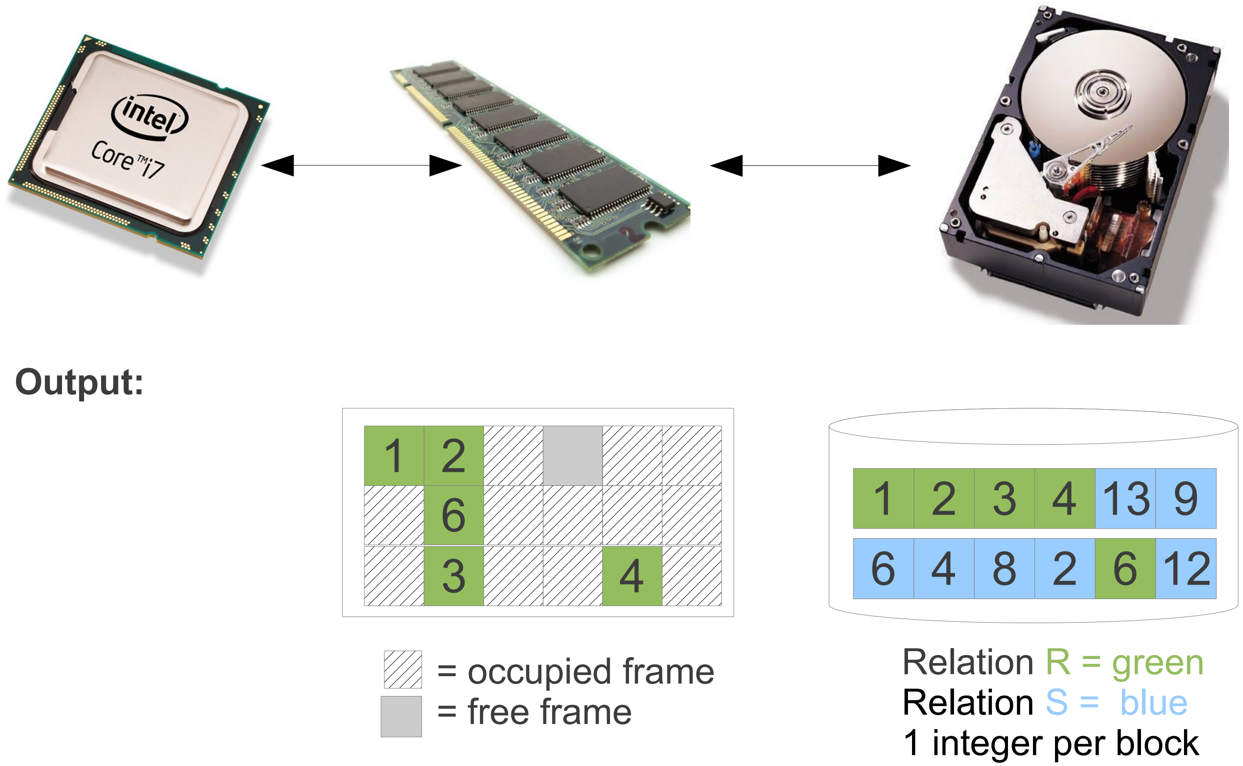


Assumption: we have $B(R) + 1$ free buffer frames

- Load all of R's blocks into memory (using $B(R)$ buffer frames) and output their elements.

Physical Operators

One-pass set union $R \cup_S S$

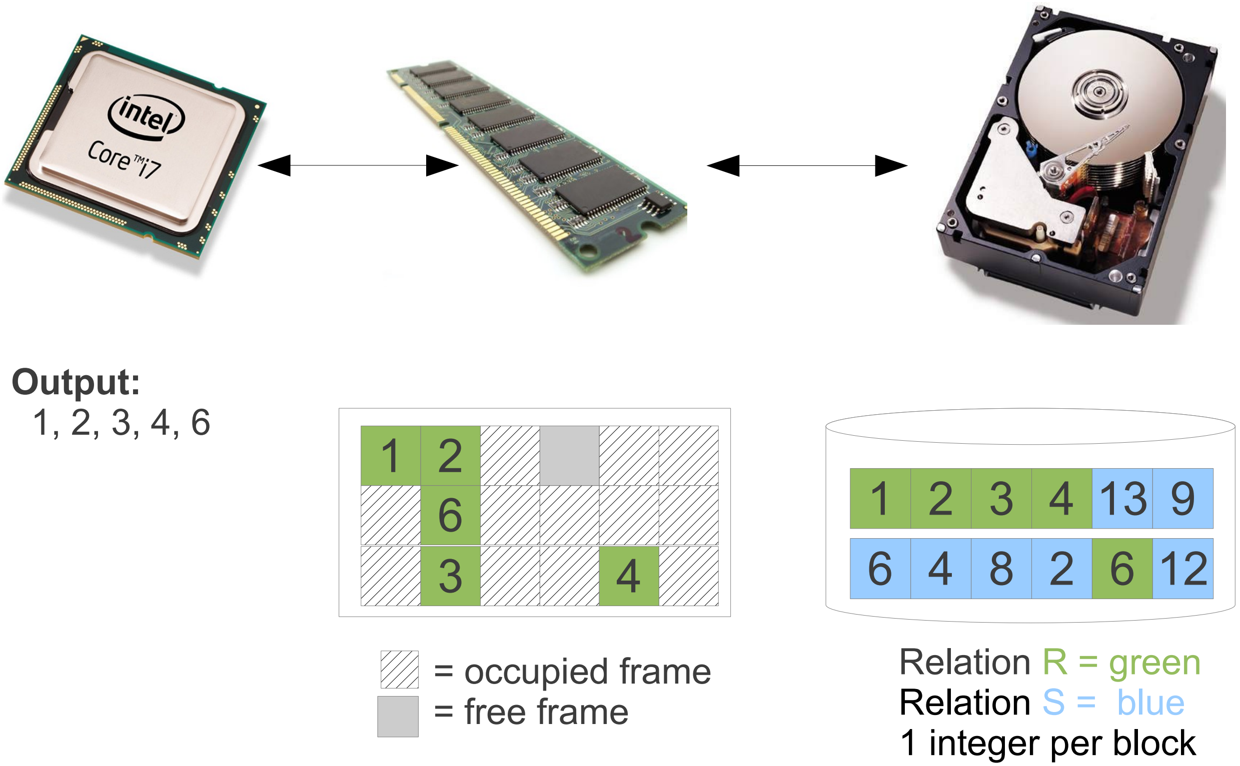


Assumption: we have $B(R) + 1$ free buffer frames

- Load all of R's blocks into memory (using $B(R)$ buffer frames) and output their elements.

Physical Operators

One-pass set union $R \cup_S S$

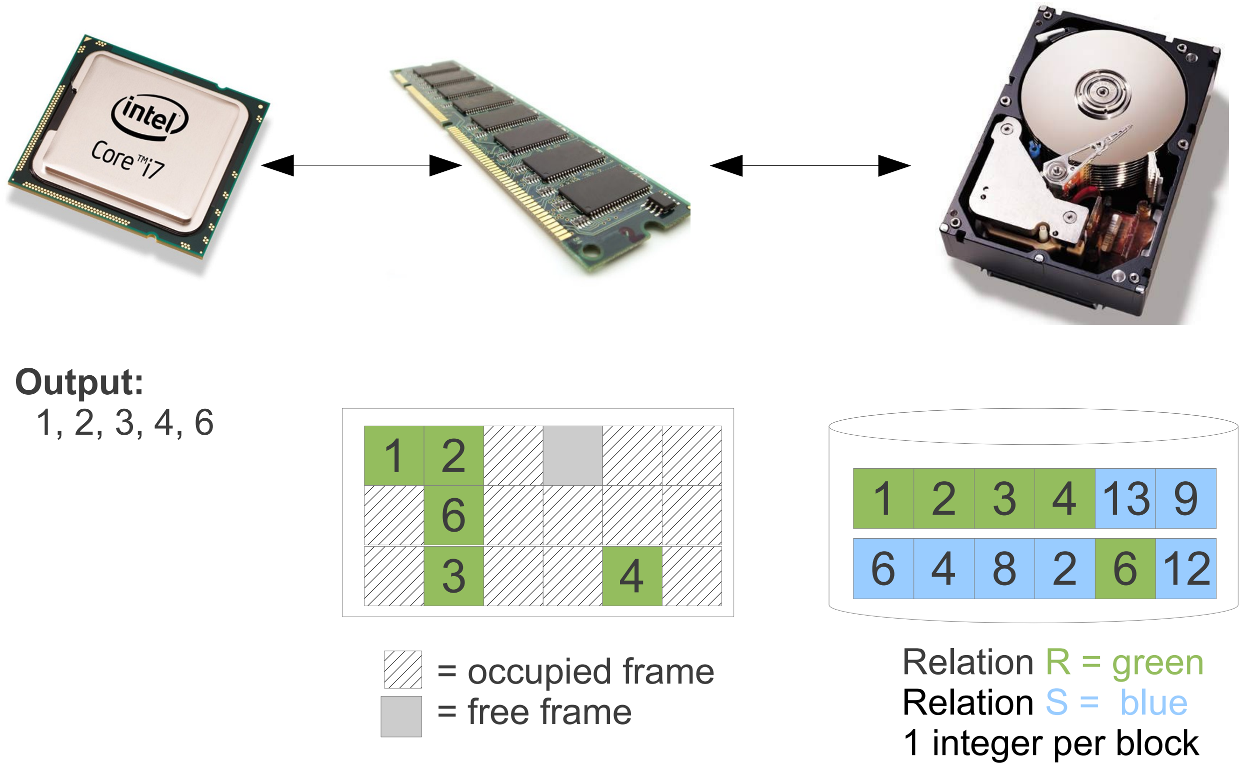


Assumption: we have $B(R) + 1$ free buffer frames

- Load all of R's blocks into memory (using $B(R)$ buffer frames) and output their elements.

Physical Operators

One-pass set union $R \cup_S S$

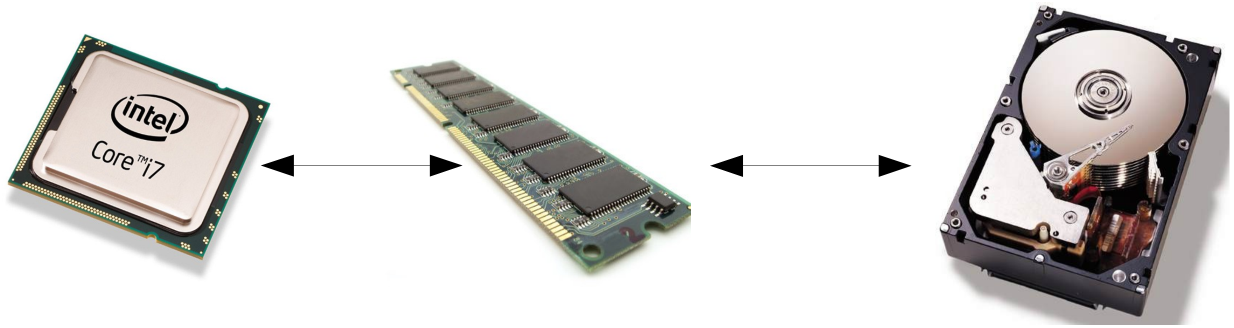


Assumption: we have $B(R) + 1$ free buffer frames

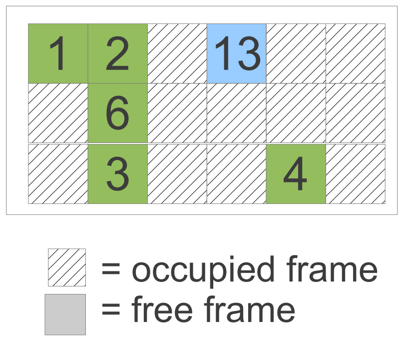
- Load 1st block of S (using 1 buffer frame), and output all of its elements that do not occur in the frames containing R.

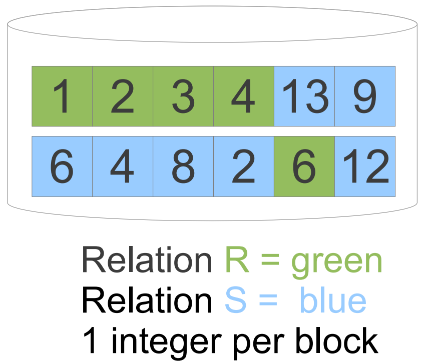
Physical Operators

One-pass set union $R \cup_S S$



Output:
1, 2, 3, 4, 6



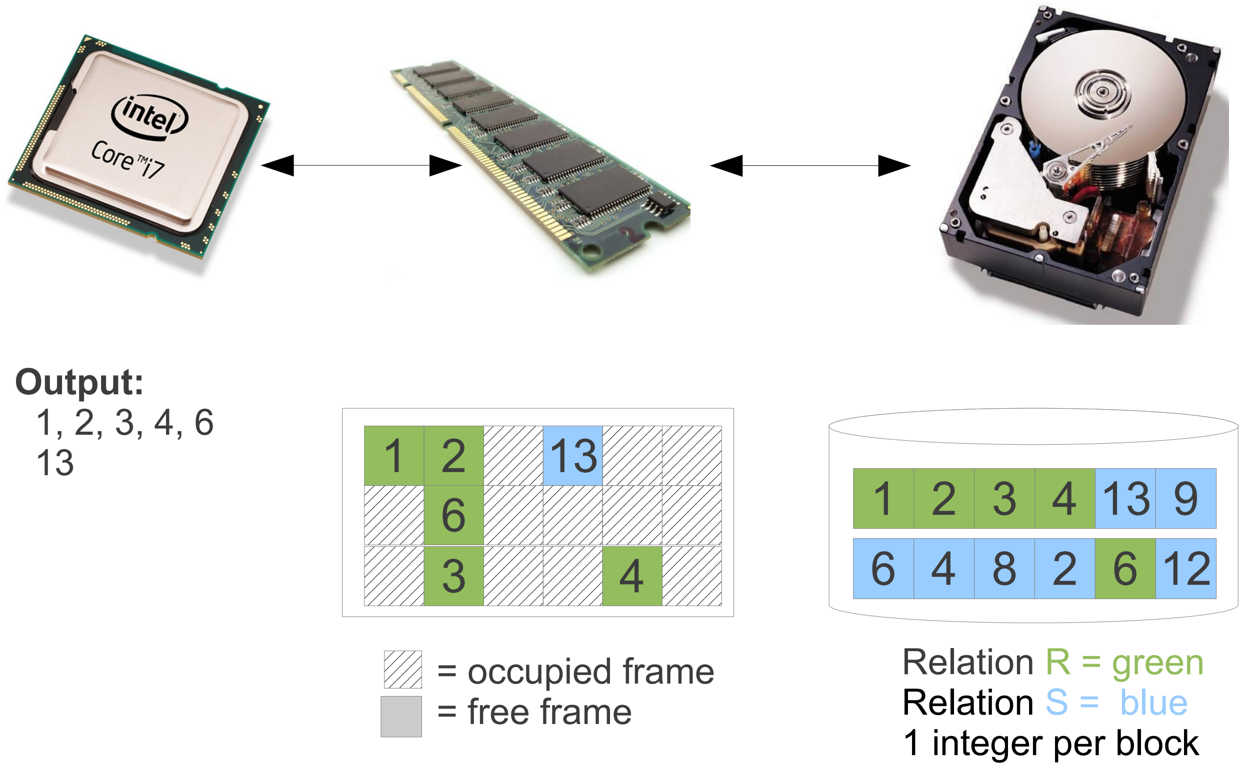


Assumption: we have $B(R) + 1$ free buffer frames

- Load 1st block of S (using 1 buffer frame), and output all of its elements that do not occur in the frames containing R.

Physical Operators

One-pass set union $R \cup_S S$

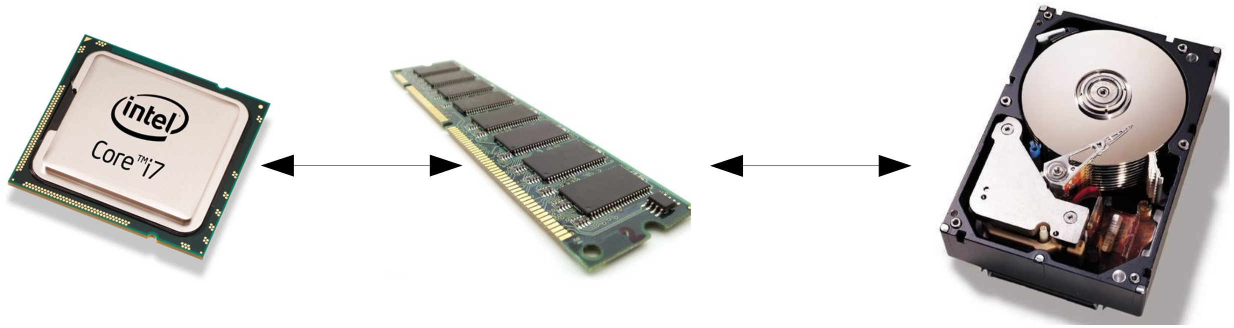


Assumption: we have $B(R) + 1$ free buffer frames

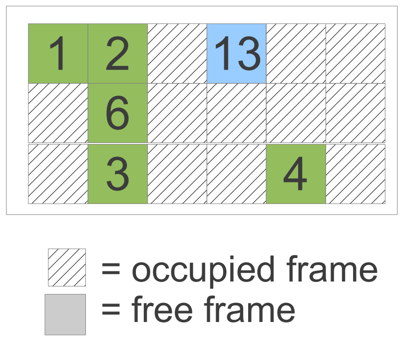
- Load 1st block of S (using 1 buffer frame), and output all of its elements that do not occur in the frames containing R.

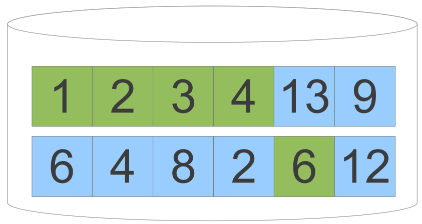
Physical Operators

One-pass set union $R \cup_S S$



Output:
1, 2, 3, 4, 6
13





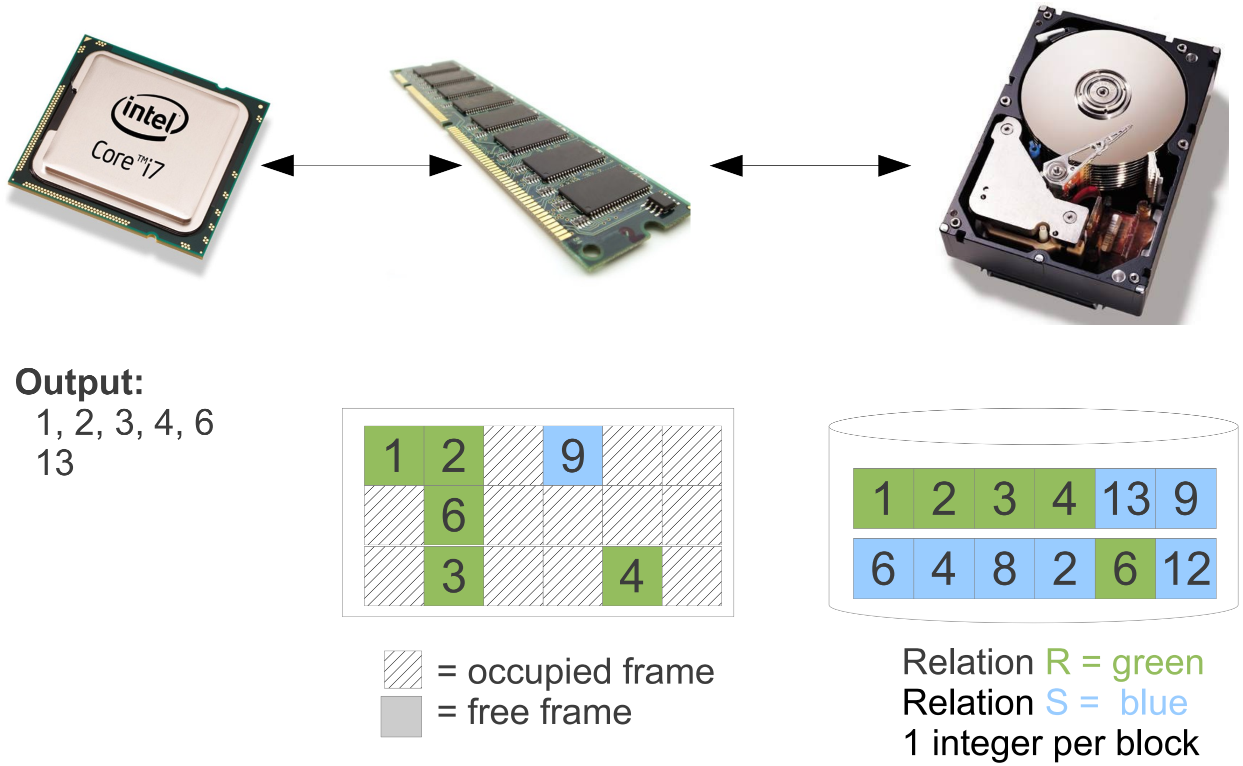
Relation R = green
Relation S = blue
1 integer per block

Assumption: we have $B(R) + 1$ free buffer frames

- Load 2nd block of S (using 1 buffer frame), and output all of its elements that do not occur in the frames containing R.

Physical Operators

One-pass set union $R \cup_S S$

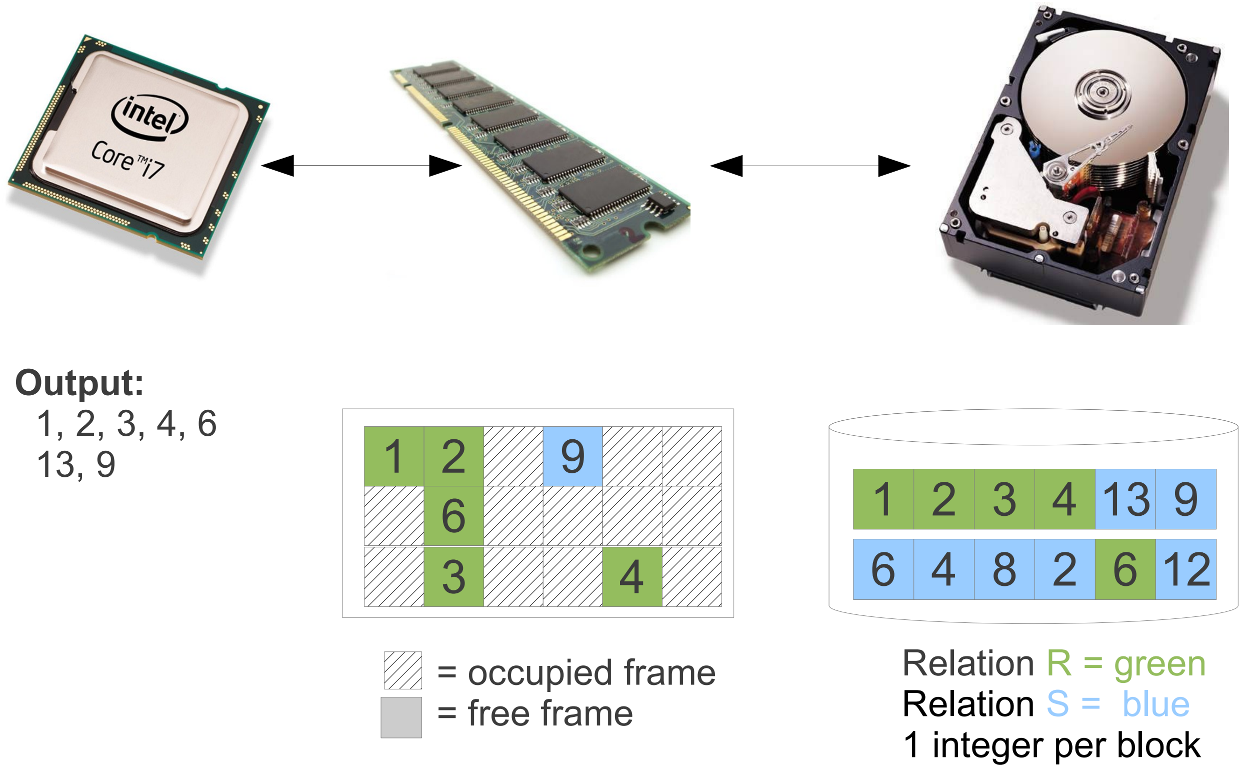


Assumption: we have $B(R) + 1$ free buffer frames

- Load 2nd block of S (using 1 buffer frame), and output all of its elements that do not occur in the frames containing R.

Physical Operators

One-pass set union $R \cup_S S$

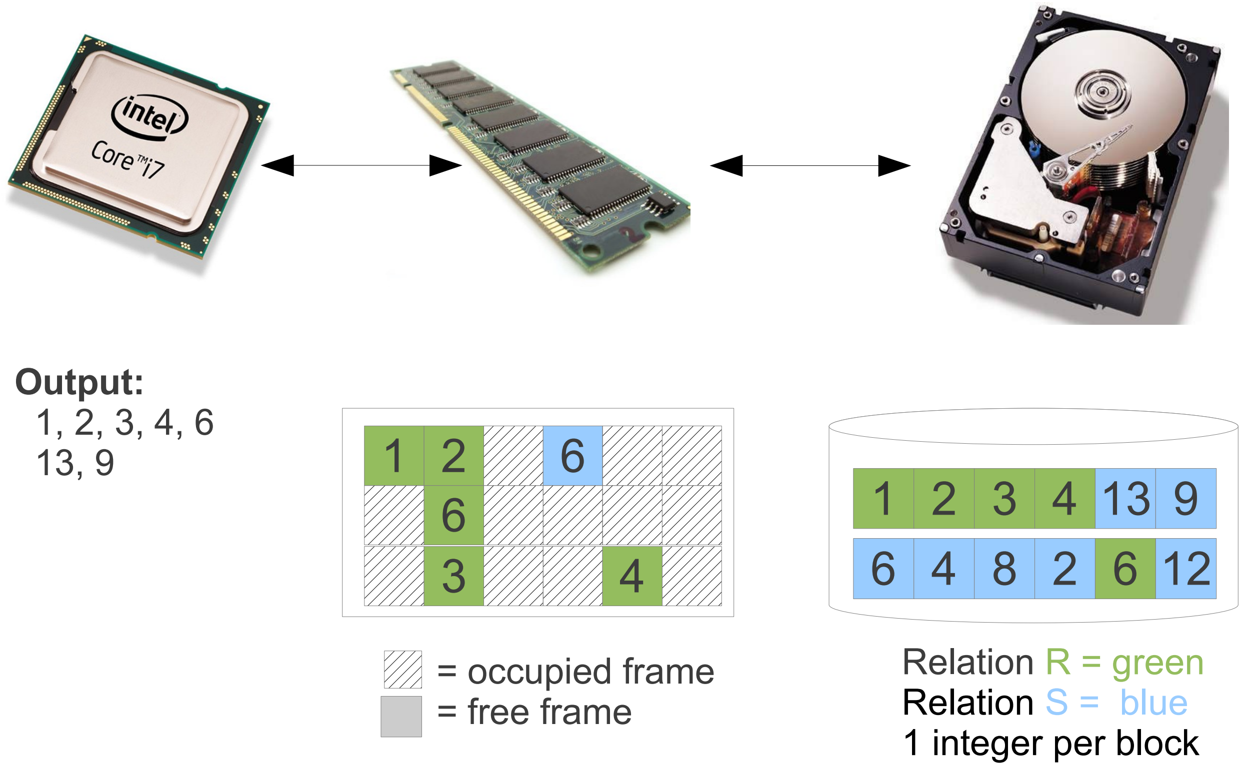


Assumption: we have $B(R) + 1$ free buffer frames

- Load 2nd block of S (using 1 buffer frame), and output all of its elements that do not occur in the frames containing R.

Physical Operators

One-pass set union $R \cup_S S$

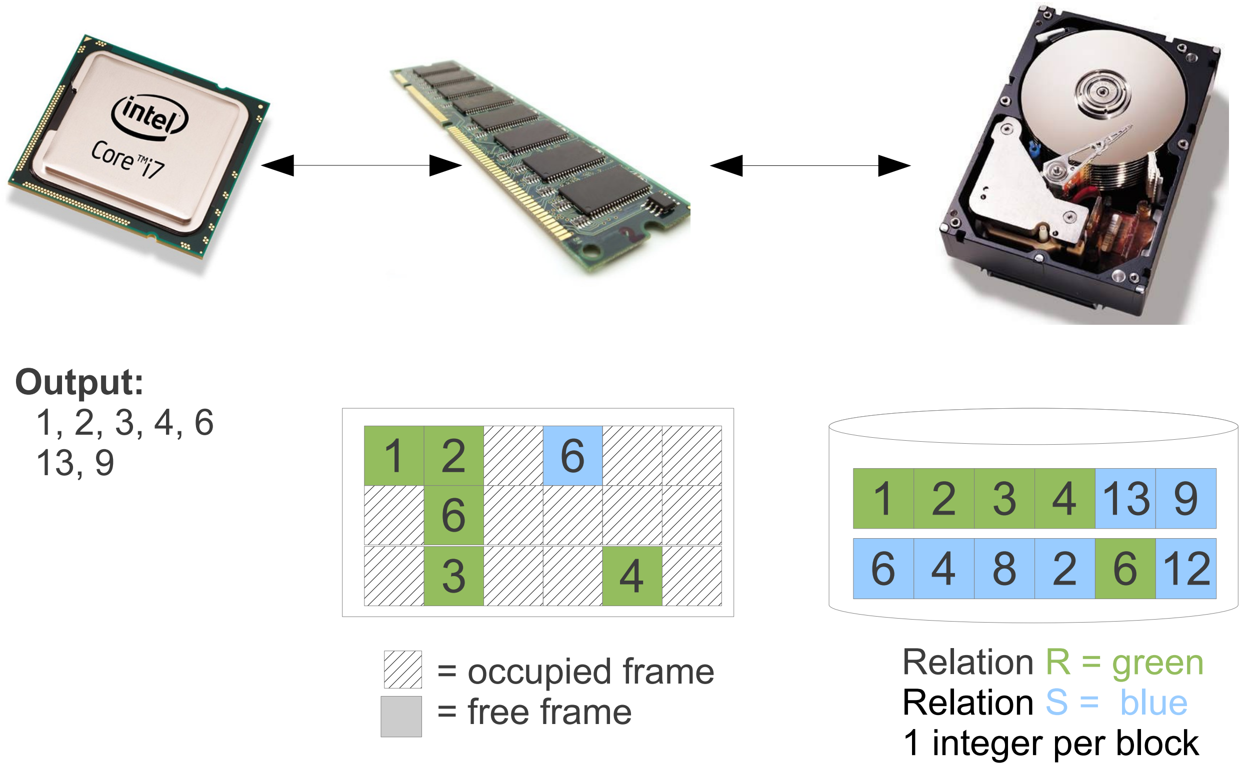


Assumption: we have $B(R) + 1$ free buffer frames

- Load 3rd block of S (using 1 buffer frame), and output all of its elements that do not occur in the frames containing R.

Physical Operators

One-pass set union $R \cup_S S$

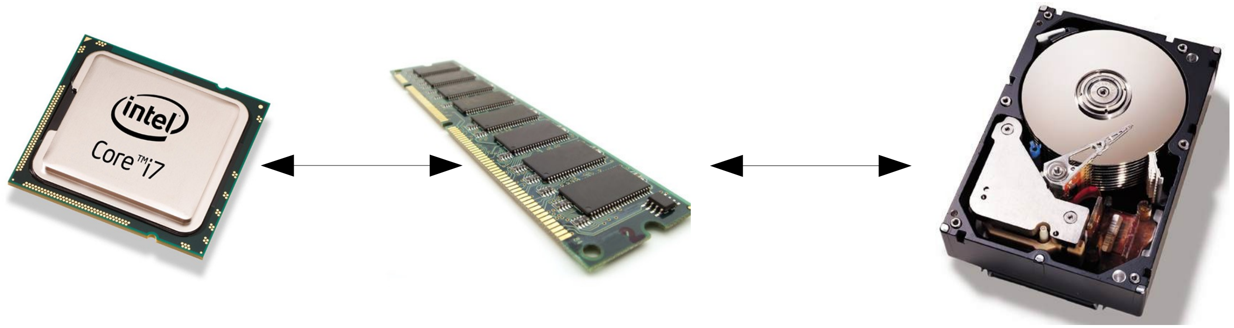


Assumption: we have $B(R) + 1$ free buffer frames

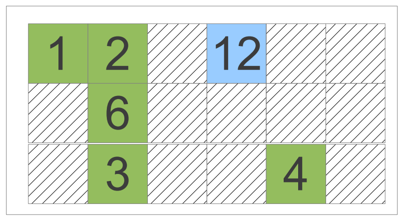
- . . . and continue doing this for until the end of S is reached.

Physical Operators

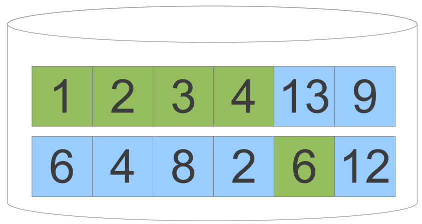
One-pass set union $R \cup_S S$



Output:
1, 2, 3, 4, 6
13, 9, 8, 12



= occupied frame
= free frame



Relation R = green
Relation S = blue
1 integer per block

Assumption: we have $B(R) + 1$ free buffer frames

- ... and continue doing this for until the end of S is reached.

Physical Operators

One-pass set union

Assume that $M - 1 \geq B(R)$. We can then compute the set union $R \cup_S S$ as follows (R and S are assumed to be sets themselves)

load R into memory buffers $N_1, \ldots, N_{B(R)}$;
  for each tuple t_R in $N_1, \ldots, N_{B(R)}$ **do**
    output t_R
for each block B_S in S **do**
  load B_S into buffer N_0;
  for each tuple t_S in N_0 **do**
    if t_S does not occur in $N_1, \ldots, N_{B(R)}$
      output t_S

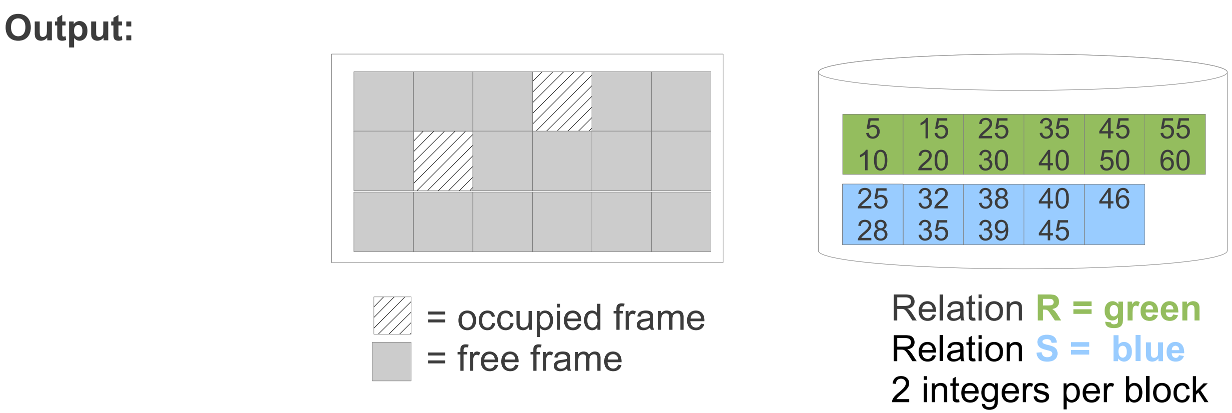
- Cost: $B(R) + B(S)$ I/O operations (ignoring output-cost)
- Note that it also costs time to check whether t_S occurs in $N_1, \ldots, N_{B(R)}$. By using a suitable main-memory data structure this can be done in $O(n)$ or $O(n \log n)$ time. We ignore this cost.
- Requires $B(R) \leq M - 1$

Physical Operators

Sort-based set union

We can also alternatively compute the set union $R \cup_S S$ as follows (again R and S are assumed to be sets):

1. Sort R
2. Sort S
3. Iterate synchronously over R and S, at each point loading 1 block of each relation in memory and inspecting 1 tuple of R and S.



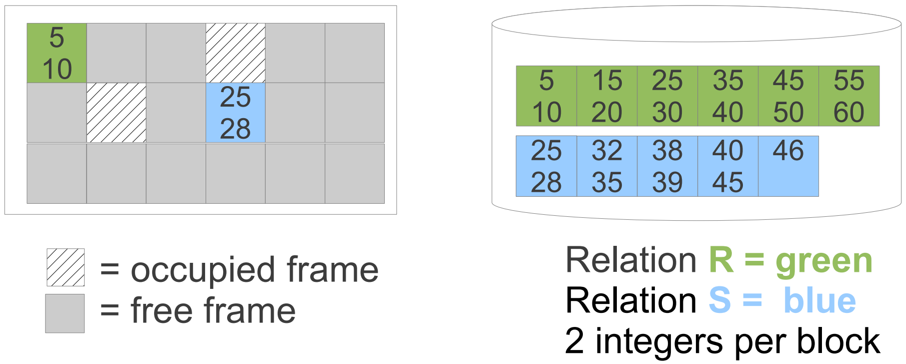
Physical Operators

Sort-based set union

We can also alternatively compute the set union $R \cup_S S$ as follows (again R and S are assumed to be sets):

1. Sort R
2. Sort S
3. Iterate synchronously over R and S, at each point loading 1 block of each relation in memory and inspecting 1 tuple of R and S.

Output:

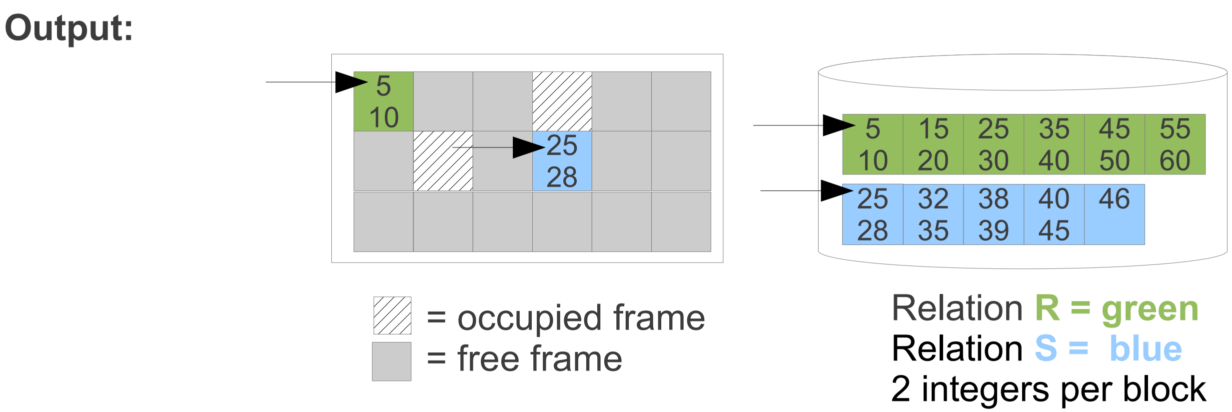


Physical Operators

Sort-based set union

We can also alternatively compute the set union $R \cup_S S$ as follows (again R and S are assumed to be sets):

1. Sort R
2. Sort S
3. Iterate synchronously over R and S, at each point loading 1 block of each relation in memory and inspecting 1 tuple of R and S.

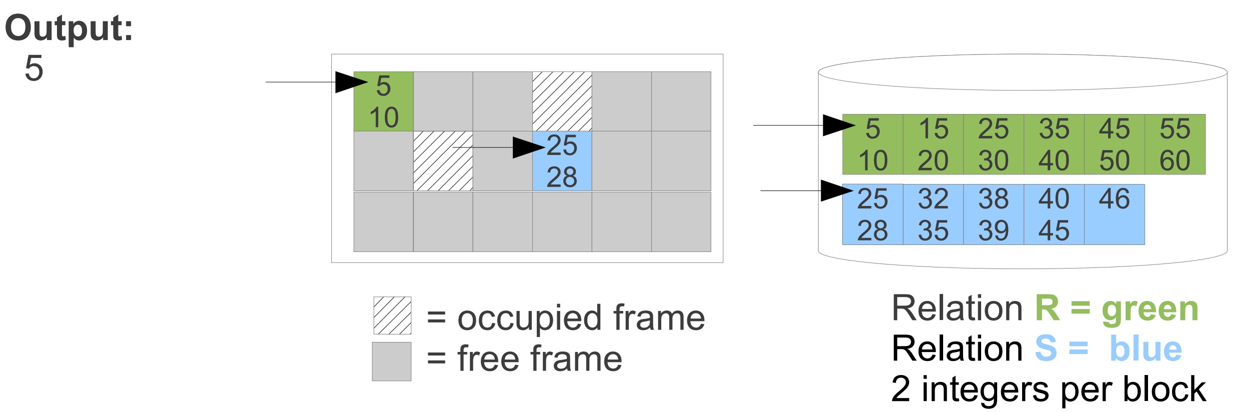


Physical Operators

Sort-based set union

We can also alternatively compute the set union $R \cup_S S$ as follows (again R and S are assumed to be sets):

1. Sort R
2. Sort S
3. Iterate synchronously over R and S, at each point loading 1 block of each relation in memory and inspecting 1 tuple of R and S.



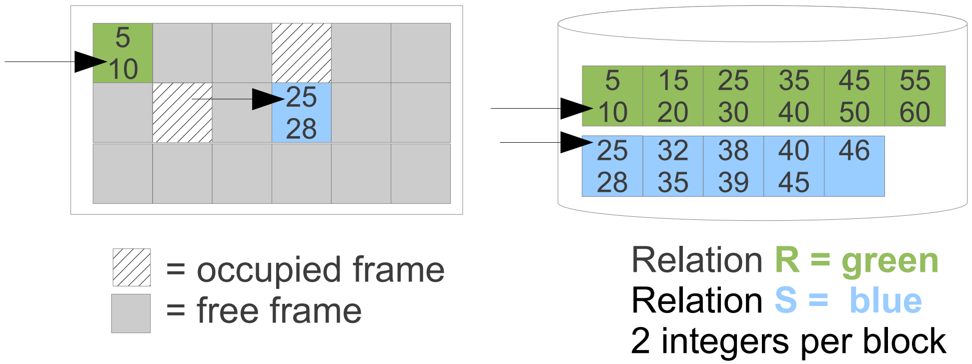
Physical Operators

Sort-based set union

We can also alternatively compute the set union $R \cup_S S$ as follows (again R and S are assumed to be sets):

1. Sort R
2. Sort S
3. Iterate synchronously over R and S, at each point loading 1 block of each relation in memory and inspecting 1 tuple of R and S.

Output:
5

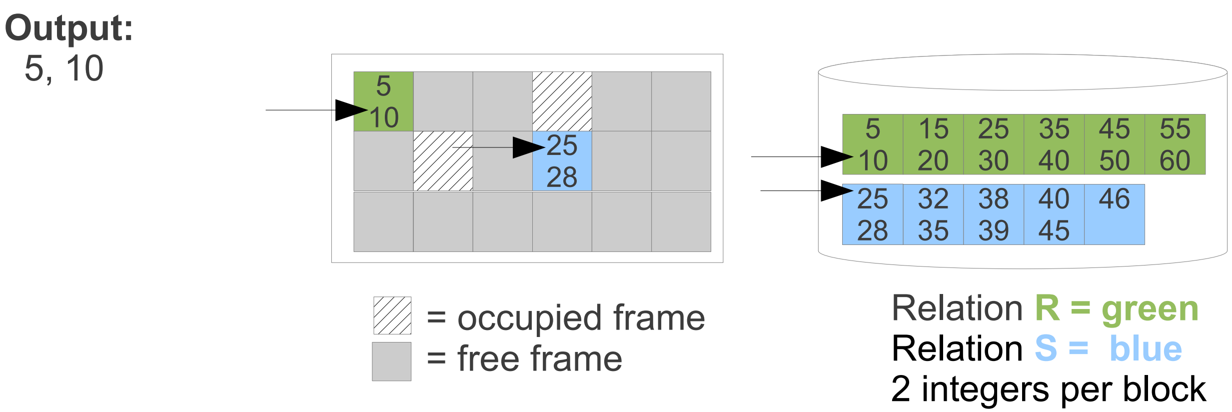


Physical Operators

Sort-based set union

We can also alternatively compute the set union $R \cup_S S$ as follows (again R and S are assumed to be sets):

1. Sort R
2. Sort S
3. Iterate synchronously over R and S, at each point loading 1 block of each relation in memory and inspecting 1 tuple of R and S.

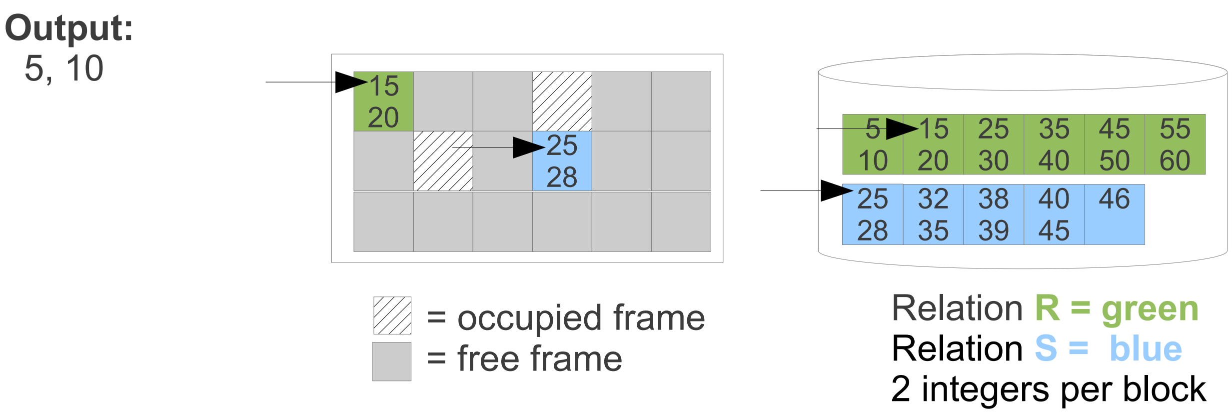


Physical Operators

Sort-based set union

We can also alternatively compute the set union $R \cup_S S$ as follows (again R and S are assumed to be sets):

1. Sort R
2. Sort S
3. Iterate synchronously over R and S, at each point loading 1 block of each relation in memory and inspecting 1 tuple of R and S.

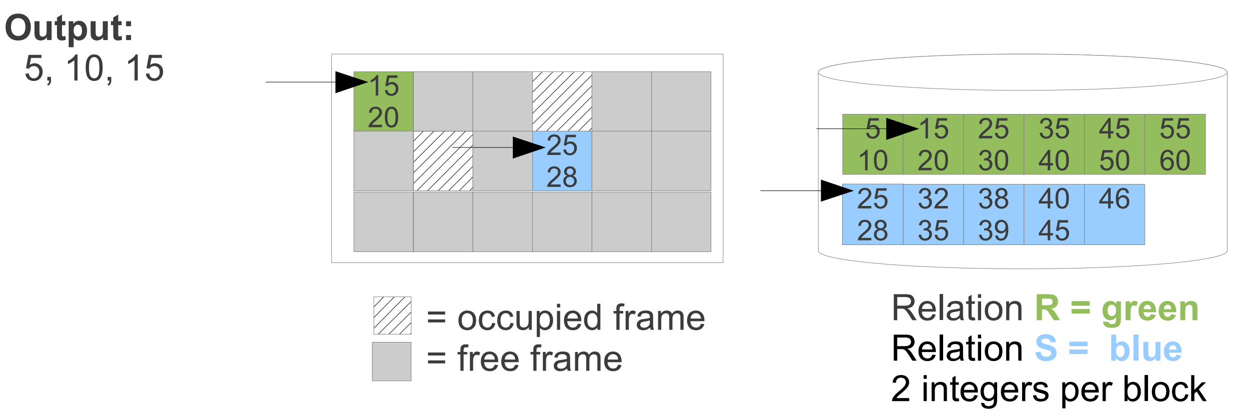


Physical Operators

Sort-based set union

We can also alternatively compute the set union $R \cup_S S$ as follows (again R and S are assumed to be sets):

1. Sort R
2. Sort S
3. Iterate synchronously over R and S, at each point loading 1 block of each relation in memory and inspecting 1 tuple of R and S.

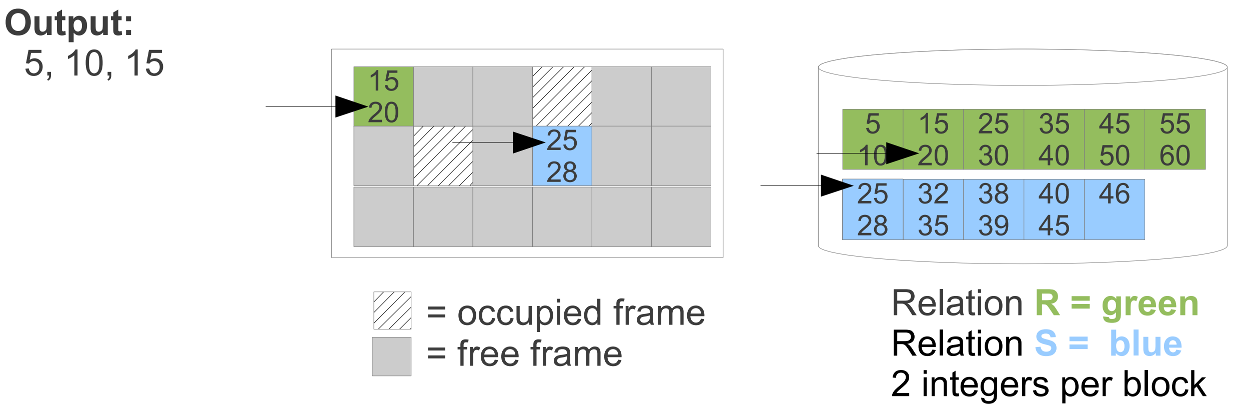


Physical Operators

Sort-based set union

We can also alternatively compute the set union $R \cup_S S$ as follows (again R and S are assumed to be sets):

1. Sort R
2. Sort S
3. Iterate synchronously over R and S, at each point loading 1 block of each relation in memory and inspecting 1 tuple of R and S.

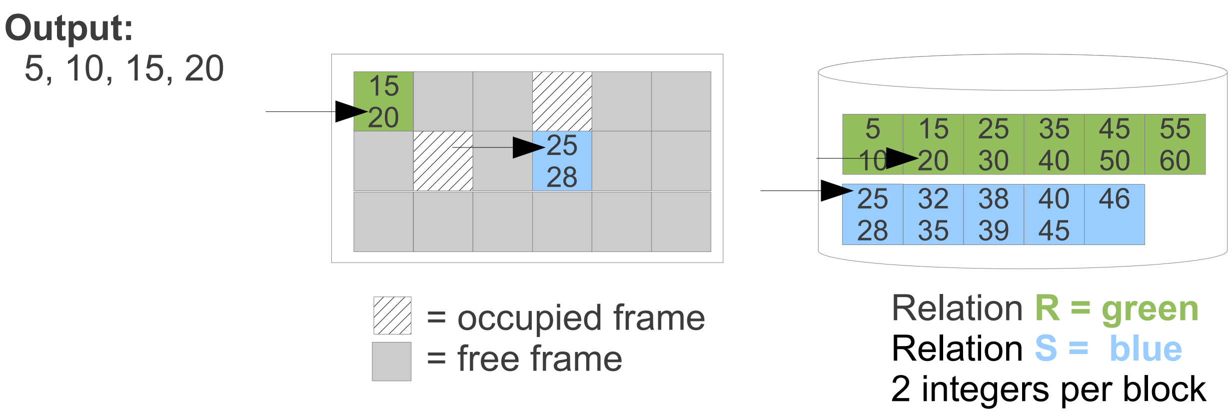


Physical Operators

Sort-based set union

We can also alternatively compute the set union $R \cup_S S$ as follows (again R and S are assumed to be sets):

1. Sort R
2. Sort S
3. Iterate synchronously over R and S, at each point loading 1 block of each relation in memory and inspecting 1 tuple of R and S.

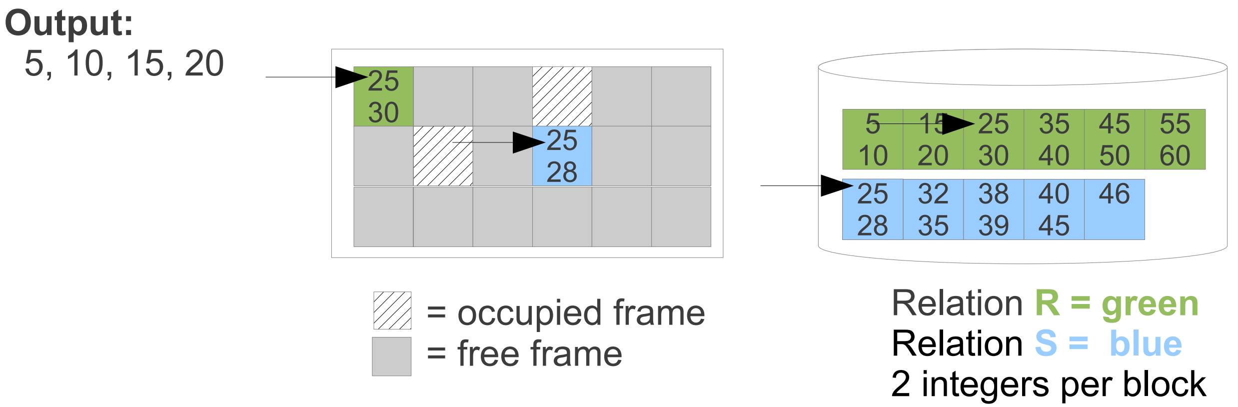


Physical Operators

Sort-based set union

We can also alternatively compute the set union $R \cup_S S$ as follows (again R and S are assumed to be sets):

1. Sort R
2. Sort S
3. Iterate synchronously over R and S, at each point loading 1 block of each relation in memory and inspecting 1 tuple of R and S.

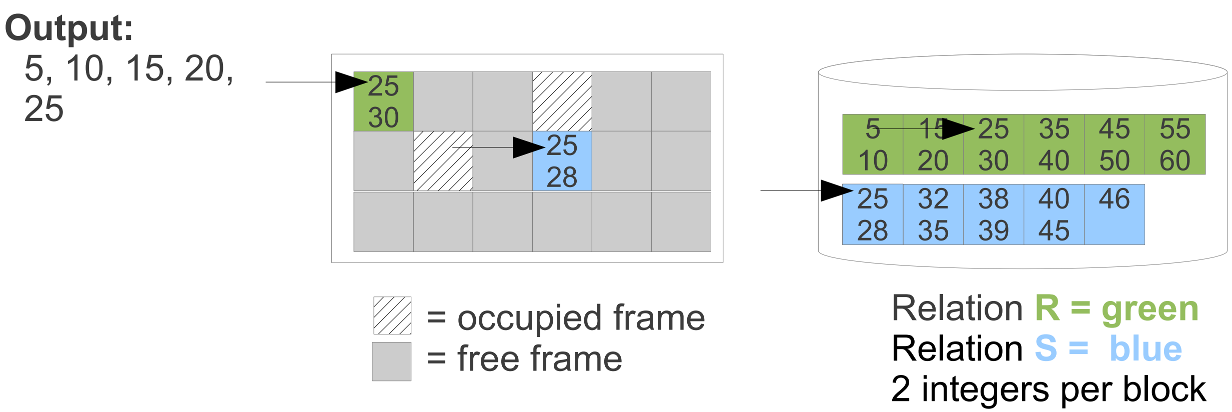


Physical Operators

Sort-based set union

We can also alternatively compute the set union $R \cup_S S$ as follows (again R and S are assumed to be sets):

1. Sort R
2. Sort S
3. Iterate synchronously over R and S, at each point loading 1 block of each relation in memory and inspecting 1 tuple of R and S.

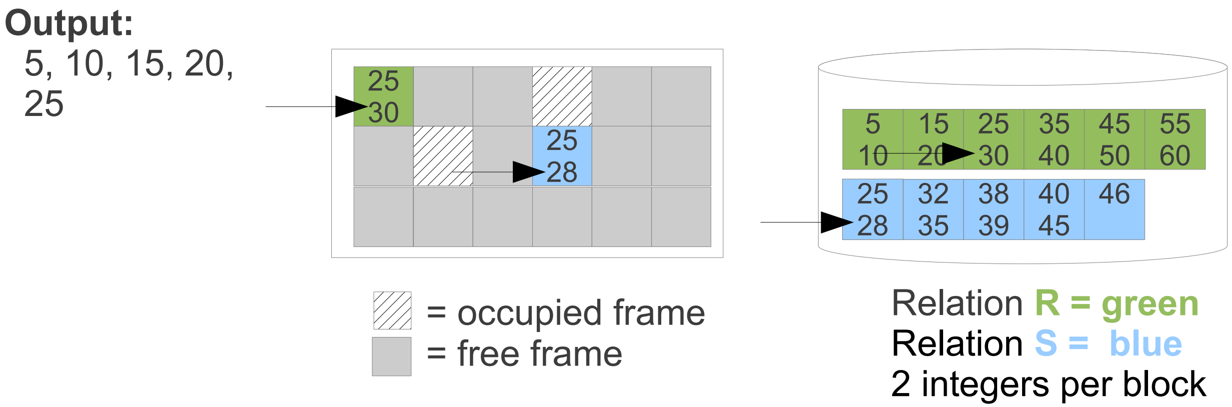


Physical Operators

Sort-based set union

We can also alternatively compute the set union $R \cup_S S$ as follows (again R and S are assumed to be sets):

1. Sort R
2. Sort S
3. Iterate synchronously over R and S, at each point loading 1 block of each relation in memory and inspecting 1 tuple of R and S.

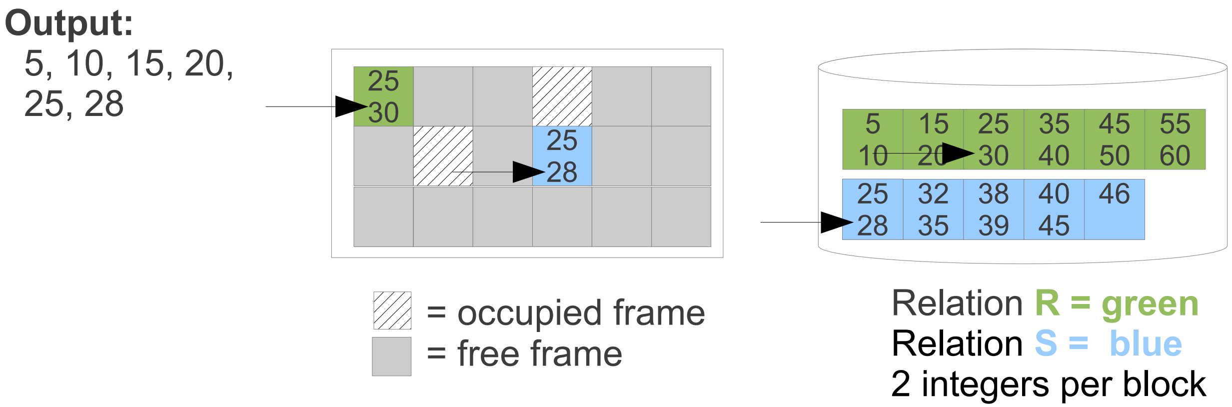


Physical Operators

Sort-based set union

We can also alternatively compute the set union $R \cup_S S$ as follows (again R and S are assumed to be sets):

1. Sort R
2. Sort S
3. Iterate synchronously over R and S, at each point loading 1 block of each relation in memory and inspecting 1 tuple of R and S.

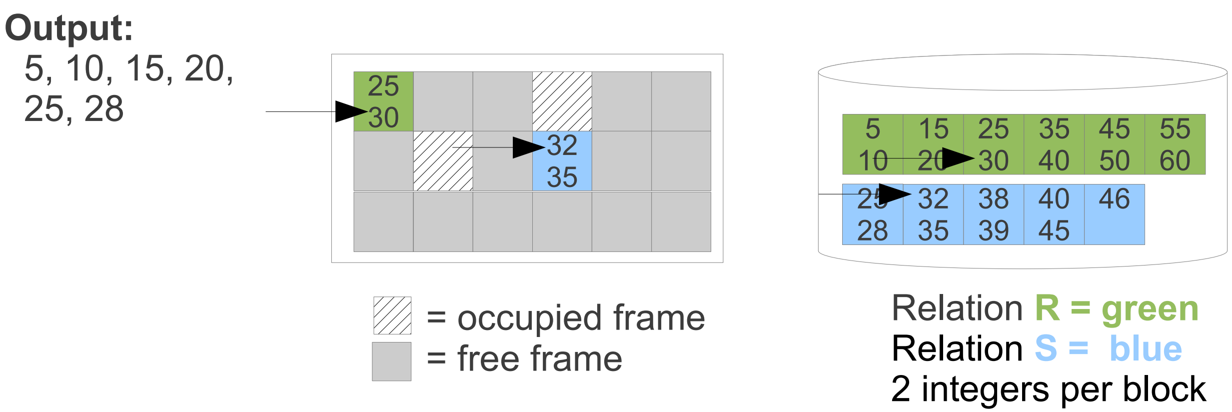


Physical Operators

Sort-based set union

We can also alternatively compute the set union $R \cup_S S$ as follows (again R and S are assumed to be sets):

1. Sort R
2. Sort S
3. Iterate synchronously over R and S, at each point loading 1 block of each relation in memory and inspecting 1 tuple of R and S.

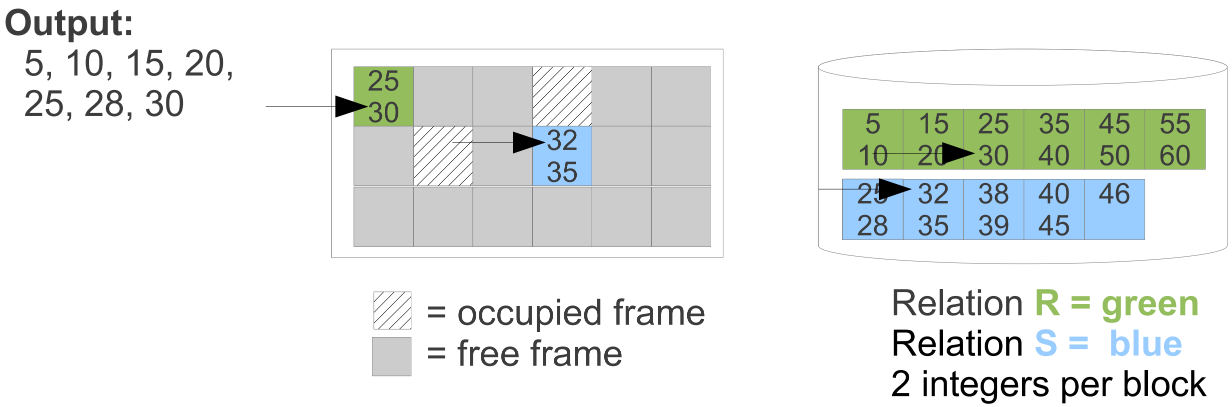


Physical Operators

Sort-based set union

We can also alternatively compute the set union $R \cup_S S$ as follows (again R and S are assumed to be sets):

1. Sort R
2. Sort S
3. Iterate synchronously over R and S, at each point loading 1 block of each relation in memory and inspecting 1 tuple of R and S.



Physical Operators

Sort-based set union

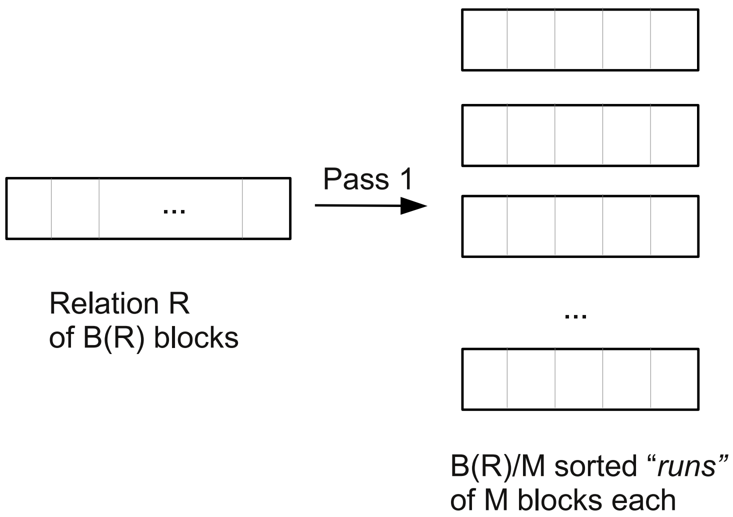
We can also alternatively compute the set union $R \cup_S S$ as follows (again R and S are assumed to be sets):

1. Sort R
2. Sort S
3. Iterate synchronously over R and S, at each point loading 1 block of each relation in memory and inspecting 1 tuple of R and S. Assume that we are currently at tuple t_R in R and tuple t_S in S:
   - If $t_R < t_S$ then we output t_R and move t_R to the next tuple in R (possibly by loading the next block of R into memory).
   - If $t_R > t_S$ then we output t_S and move t_S to the next tuple in S (possibly by loading the next block of S into memory).
   - If $t_R = t_S$ then we output t_R and move t_R to the next tuple in R and t_S to the next tuple in S (possibly by loading the next block)

Physical Operators

Sort-based set union

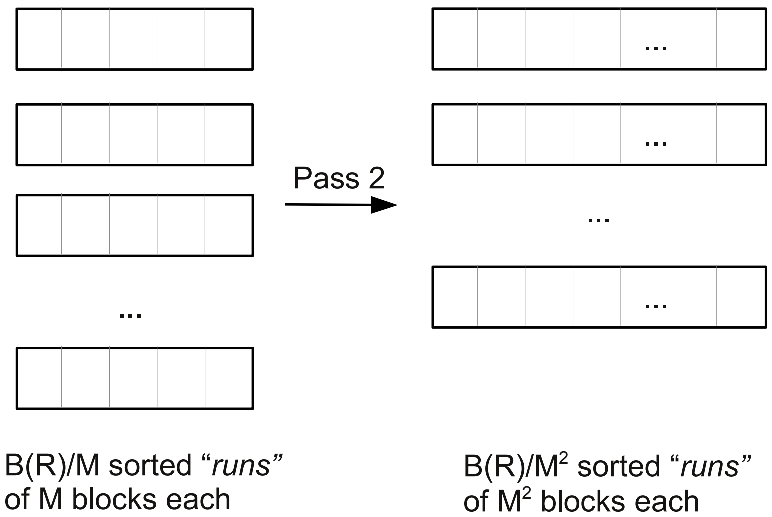
- Sorting can in principle be done by any suitable algorithm, but is usually done by Multiway Merge-Sort:
  - In the first pass we read M blocks at the same time from the input relation, sort these by means of a main-memory sorting algorithm, and write the sorted resulting sublist to disk. After the first pass we hence have $B(R)/M$ sorted sublists of M blocks each.



Physical Operators

Sort-based set union

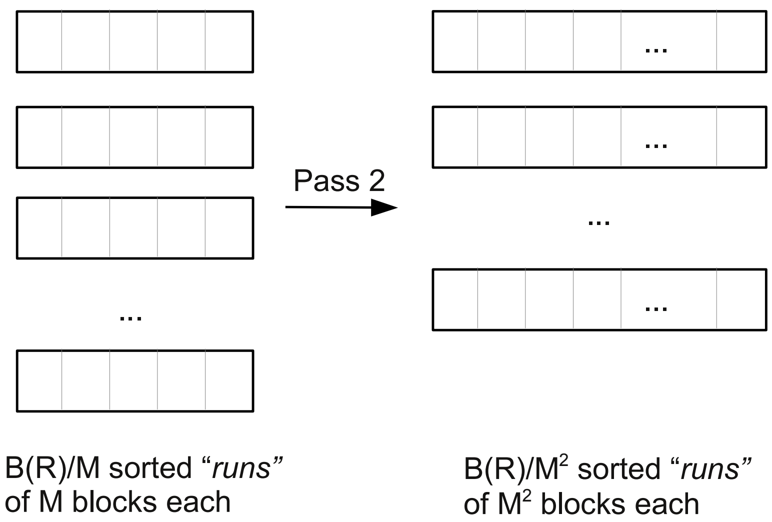
- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  - In the 2nd pass, we merge the first M sublists from the first pass into a single sublist of M^2 blocks. We do so by iterating synchronously over these M sublists, keeping 1 block of each list into memory during this iteration.



Physical Operators

Sort-based set union

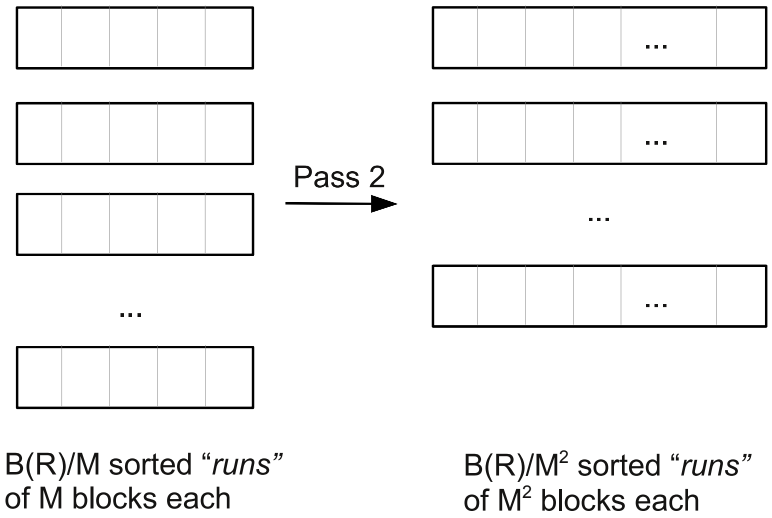
- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  - We then merge the next M sublists into a single sublist, and continue until we have treated each sublist resulting from the first pass.



Physical Operators

Sort-based set union

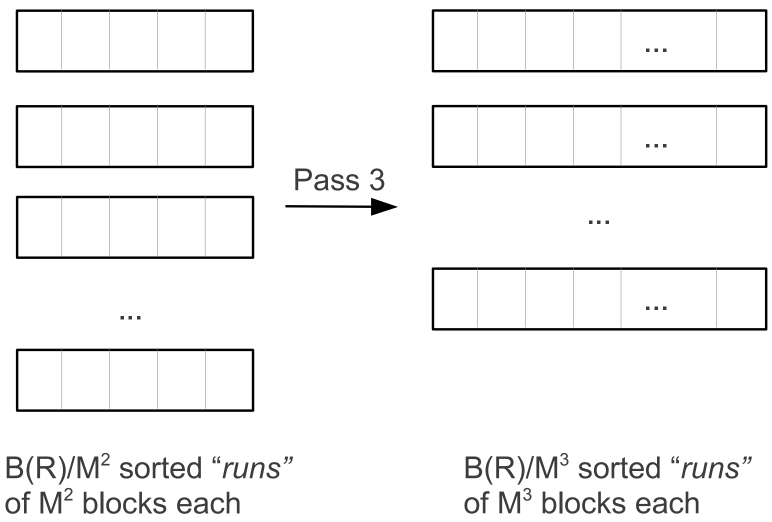
- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  - After the second pass we hence have $B(R)/M^2$ sorted sublists of M^2 blocks each.



Physical Operators

Sort-based set union

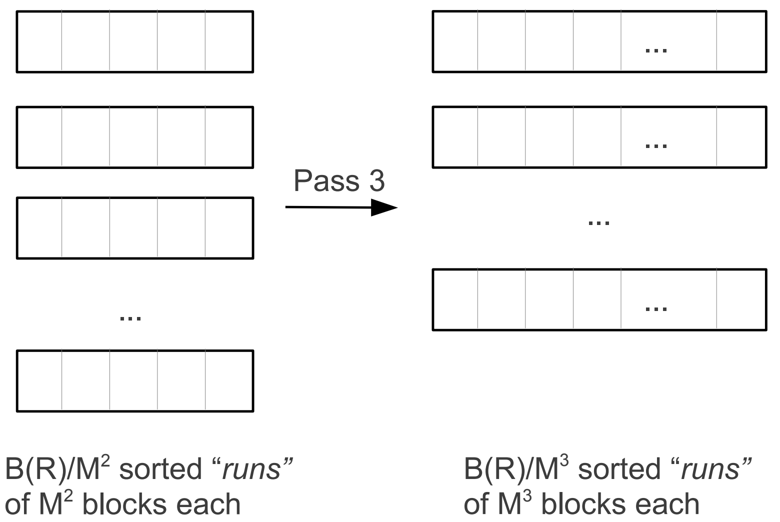
- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  - In the 3rd pass, we merge the first M sublists from the 2nd pass (each of M^2 blocks) into a single sublist of M^3 blocks. We do so by iterating synchronously over these M sublists, keeping 1 block of each list into memory during this iteration.



Physical Operators

Sort-based set union

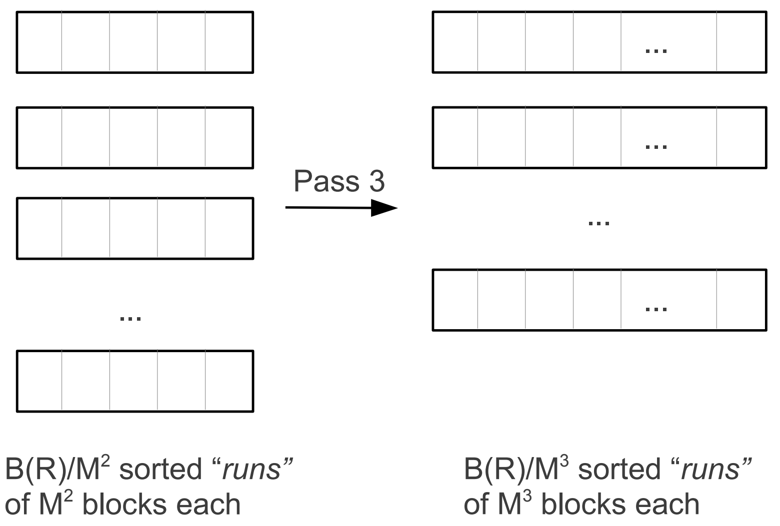
- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  - We then merge the next M sublists into a single sublist, and continue until we have treated each sublist resulting from the 2nd pass .



Physical Operators

Sort-based set union

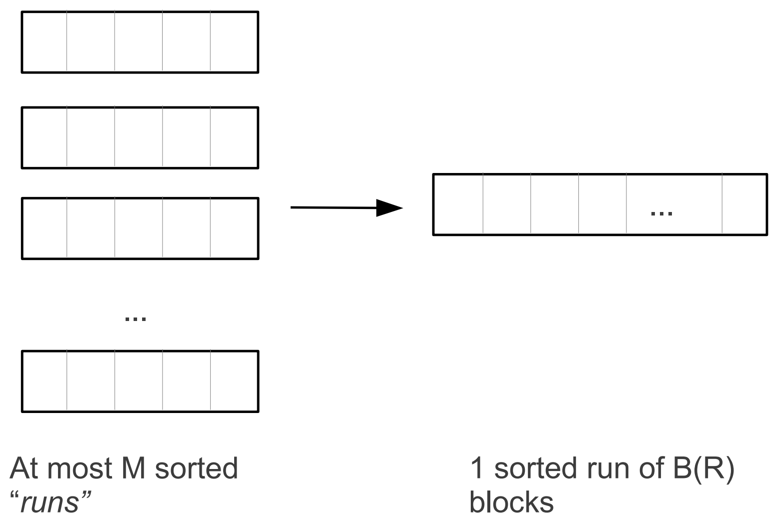
- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  - After the 3rd pass we hence have $B(R)/M^3$ sorted sublists of M^3 blocks each.



Physical Operators

Sort-based set union

- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  - We keep doing new passes until we reach a single sorted list.



Physical Operators

Sort-based set union

- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  1. In the first pass we read M blocks at the same time from the input relation, sort these by means of a main-memory sorting algorithm, and write the sorted resulting sublist to disk. After the first pass we hence have $B(R)/M$ sorted sublists of M blocks each.
  2. In the following passes we keep reading M blocks from these sublists and merge them into larger sorted sublists. (After the second pass we hence have $B(R)/M^2$ sorted sublists of M^2 blocks each, after the third pass $B(R)/M^3$ sorted sublists, . . .)
  3. We repeat until we obtain a single sorted sublist.
- What is the complexity of this?
  1. In each pass we read and write the entire input relation exactly once.
  2. There are $\lceil \log_M B(R) \rceil$ passes
  3. The total cost is hence $2B(R) \lceil \log_M B(R) \rceil$ I/O operations.

Physical Operators

Sort-based set union

- The costs of sort-based set union:
  1. Sorting R : $2B(R)\,\lceil \log_M B(R) \rceil$ I/O's
  2. Sorting S : $2B(S)\,\lceil \log_M B(S) \rceil$ I/O's
  3. Synchronized iteration: $B(R) + B(S)$ I/O's

  In Total:

$$\boxed{2B(R)\,\lceil \log_M B(R) \rceil + 2B(S)\,\lceil \log_M B(S) \rceil + B(R) + B(S)}$$

- Uses M memory-buffers during sorting
- Requires 2 memory-buffers for synchronized iteration

Physical Operators

Sort-based set union

Remark: the "synchronized iteration" phase of sort-based set union is very similar to the merge phase of multiway merge-sort. Sometimes it is possible to combine the last merge phase with the synchronized iteration, and avoid $2B(R) + 2B(S)$ I/Os:

1. Sort R, but do not execute the last merge phase. R is hence still divided in $1 < l \leq M$ sorted sublists.
2. Sort S, but do not execute the last merge phase. S is hence still divided in $1 < k \leq M$ sorted sublists.
3. If $l + k < M$ then we can use the M available buffers to load the first block of each sublist of R and S in memory.
4. Then iterate synchronously through these sublists: at each point search the "smallest" (according to the sort order) record in the $l + k$ buffers, and output that. Move to the next record in the buffers when required. When all records from a certain buffer are processed, load the next block from the corresponding sublist.

Physical Operators

Sort-based set union

The cost of the optimized sort-based set union algorithm is as follows:

1. Sort R, but do not execute the last merge phase.

$$2B(R)(\lceil \log_M B(R) \rceil - 1)$$

2. Sort S, but do not execute the last merge phase.

$$2B(S)(\lceil \log_M B(S) \rceil - 1)$$

3. Synchronized iteration through the sublists: $B(R) + B(S)$ I/O's

Total:

$$\boxed{2B(R) \lceil \log_M B(R) \rceil + 2B(S) \lceil \log_M B(S) \rceil - B(R) - B(S)}$$

We hence save $2B(R) + 2B(S)$ I/O's.

Physical Operators

Sort-based set union

Note that this optimization is only possible if $k + l \leq M$.

Observe that $k = \left\lceil \frac{B(R)}{M^{\lceil \log_M B(R) \rceil - 1}} \right\rceil$ and $l = \left\lceil \frac{B(S)}{M^{\lceil \log_M B(S) \rceil - 1}} \right\rceil$.

In other words, this optimization is only possible if:

$$\left\lceil \frac{B(R)}{M^{\lceil \log_M B(R) \rceil - 1}} \right\rceil + \left\lceil \frac{B(S)}{M^{\lceil \log_M B(S) \rceil - 1}} \right\rceil \leq M$$

Physical Operators

Sort-based set union

Example: we have 15 buffers available, $B(R) = 100$, and $B(S) = 120$.

- Number of passes required to sort R completely: $\lceil \log_M B(R) \rceil = 2$
- Number of passes required to sort S completely: $\lceil \log_M B(S) \rceil = 2$
- Can the optimization be applied?

$$\left\lceil \frac{100}{15} \right\rceil + \left\lceil \frac{120}{15} \right\rceil = 15 \leq M$$

- The optimized sort-based set union hence costs:

$$2 \times 100 \times 2 + 2 \times 120 \times 2 - 100 - 120 = 660$$

Physical Operators

Sort-based set union

- The book states that in practice 2 passes usually suffice to completely sort a relation.
- If we assume that R and S can be sorted in two passes (given the available memory M) then we can instantiate our cost formula as follows:
  - Without optimization: $5B(R) + 5B(S)$
  - With optimization: $3B(R) + 3B(S)$, but in this case we require sufficient memory:

    $$\left\lceil \frac{B(R)}{M} \right\rceil + \left\lceil \frac{B(S)}{M} \right\rceil \leq M$$

    or (approximately) $B(R) + B(S) \leq M^2$.
    $\rightarrow$ This is the formula that you will find in the book!
- Note that the book focuses on the optimized algorithm in the case where two passes suffice: the so-called "two-pass, sort-based set union". It only sketches the generalization to multiple passes.

Physical Operators

Hash-based set union

We can also alternatively compute the set union $R \cup_S S$ as follows (R and S are assumed to be sets, and we assume that $B(R) \leq B(S)$):

1. Partition, by means of hash function(s), R in buckets of at most $M - 1$ blocks each. Let k be the resulting number of buckets, and let R_i be the relation formed by the records in bucket i.
2. Partition, by means of the same hash function(s) as above, S in k buckets. Let S_i be the relation formed by the records in bucket i.

   Observe: the records in R_i and S_i have the same hash value! A record t hence occurs in both R and S if, and only if, there is a bucket i such that t occurs in both R_i and S_i.
3. We can hence compute the set union by calculating the set union of R_i and S_i, for every $i \in 1, \ldots, k$. Since every R_i contains at most $M - 1$ blocks, we can do so using the one-pass algorithm.

Note: in contrast to the sort-based set union, the output of a hash-based set union is unsorted!

Physical Operators

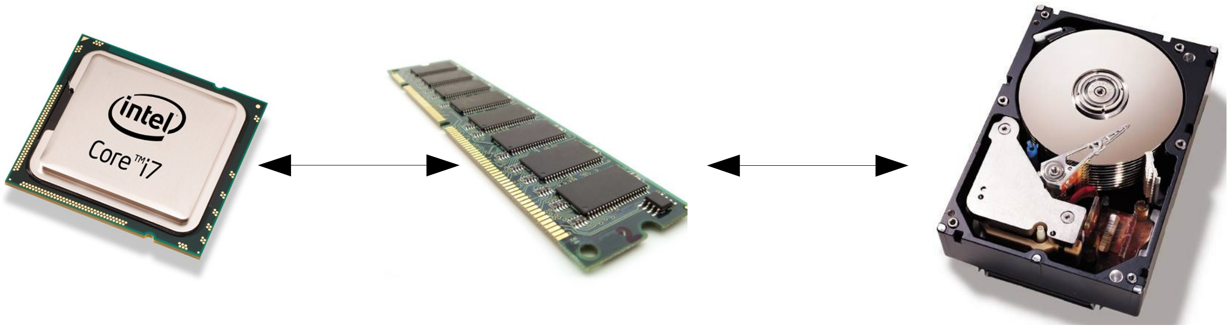
Hash-based set union

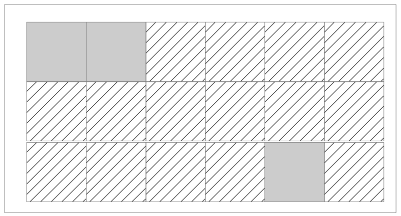
How do we partition R in buckets of at most $M - 1$ blocks?

1. Using M buffers, we first hash R into $M - 1$ buckets.
2. Subsequently we partition each bucket separately in $M - 1$ new buckets, by using a new hash function distinct from the one used in the previous step (why?)
3. We continue doing so until the obtained buckets consists of at most $M - 1$ blocks.

Physical Operators

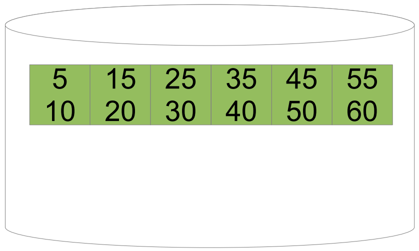
Hashing R into $M-1$ buckets using M buffers





= occupied frame

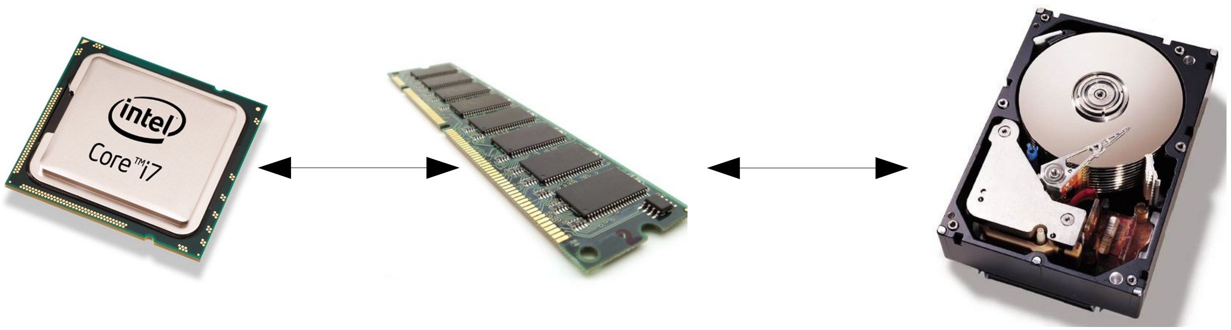
= free frame



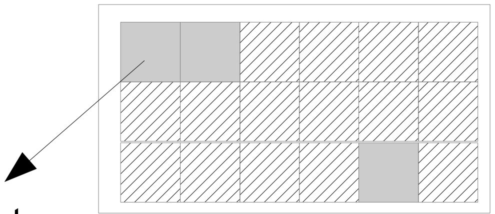
Relation R = green
2 integers per block

Physical Operators

Hashing R into $M-1$ buckets using M buffers

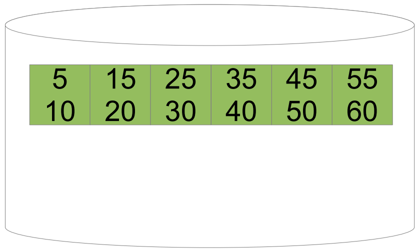


M = 3



Buffer for elements
that hash to bucket 1

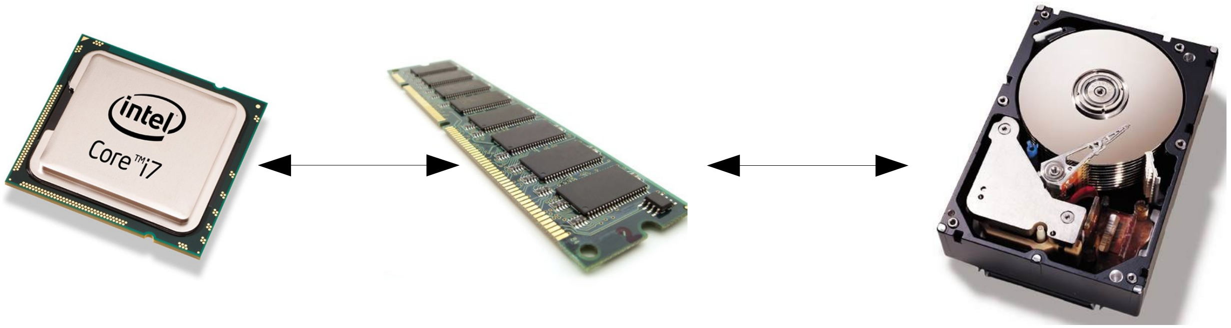
= occupied frame
= free frame

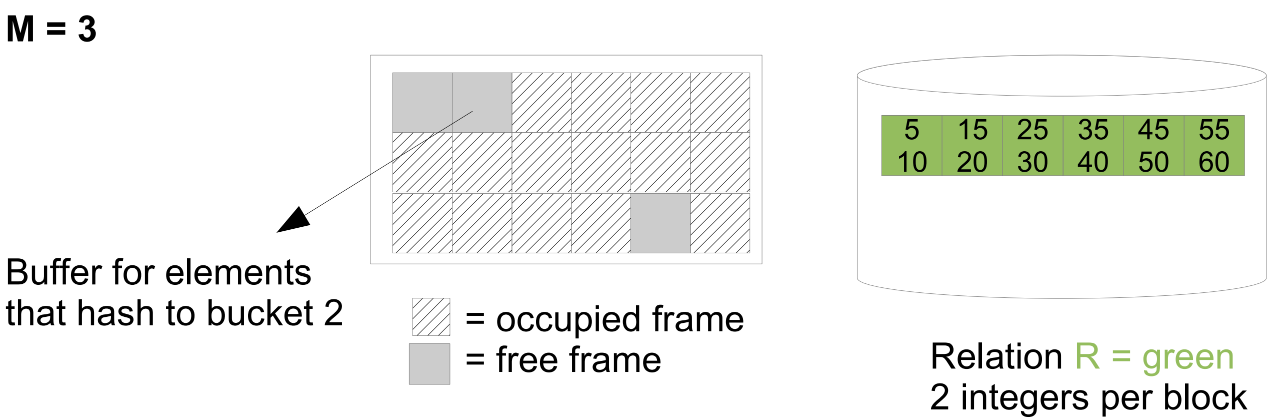


Relation R = green
2 integers per block

Physical Operators

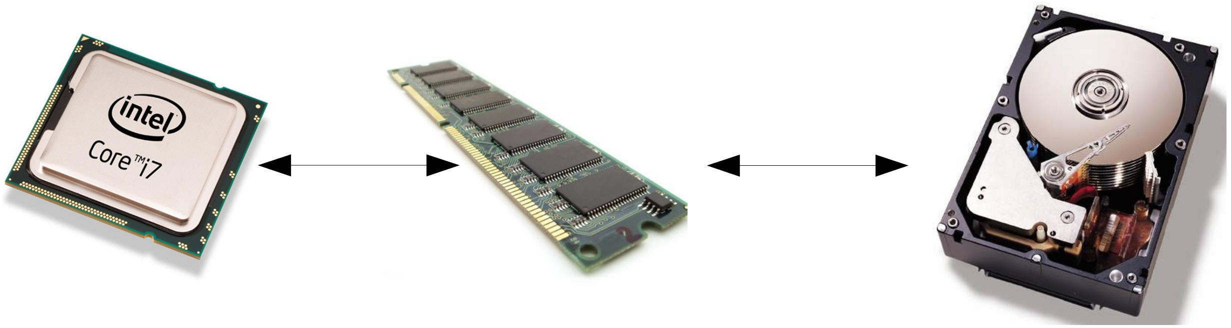
Hashing R into $M - 1$ buckets using M buffers



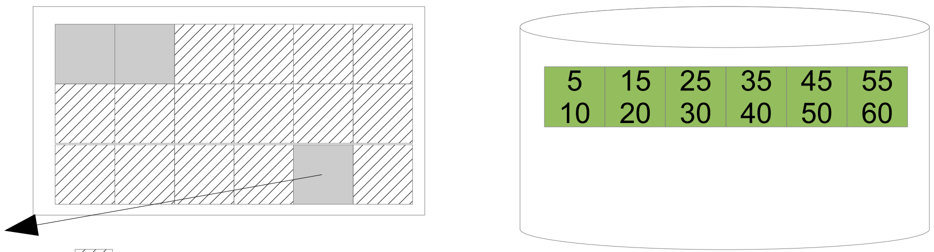


Physical Operators

Hashing R into $M - 1$ buckets using M buffers



M = 3



Buffer for loading R from disk, 1 block at a time

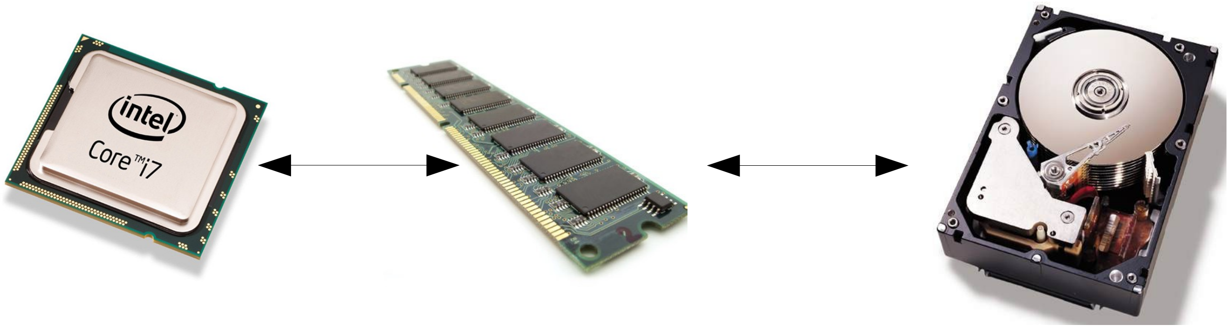
= occupied frame

= free frame

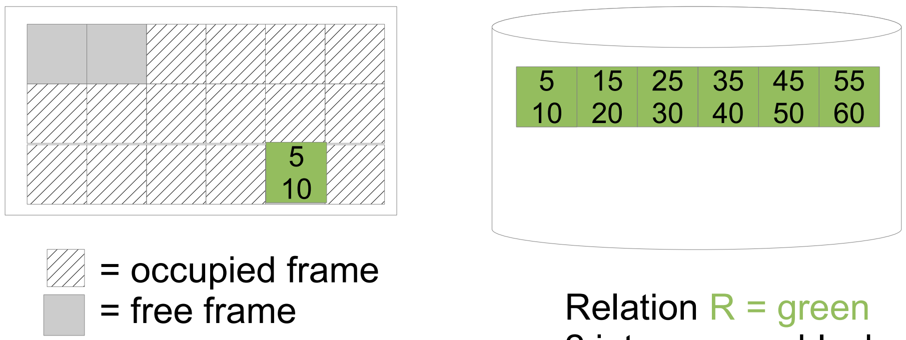
Relation R = green
2 integers per block

Physical Operators

Hashing R into $M - 1$ buckets using M buffers



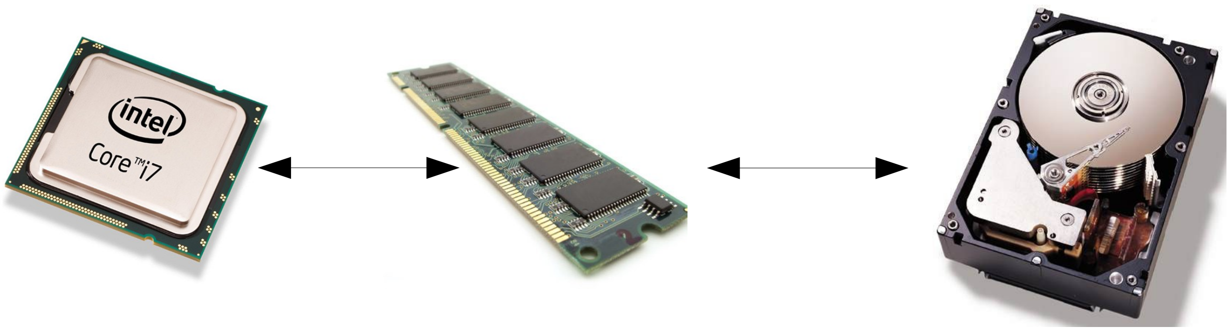
M = 3



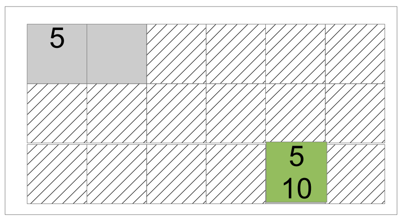
Relation R = green
2 integers per block

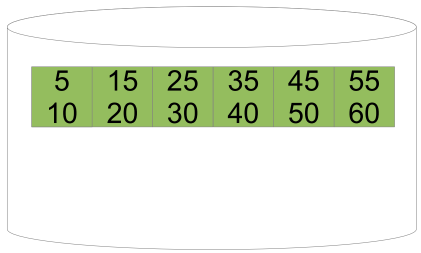
Physical Operators

Hashing R into $M - 1$ buckets using M buffers



M = 3



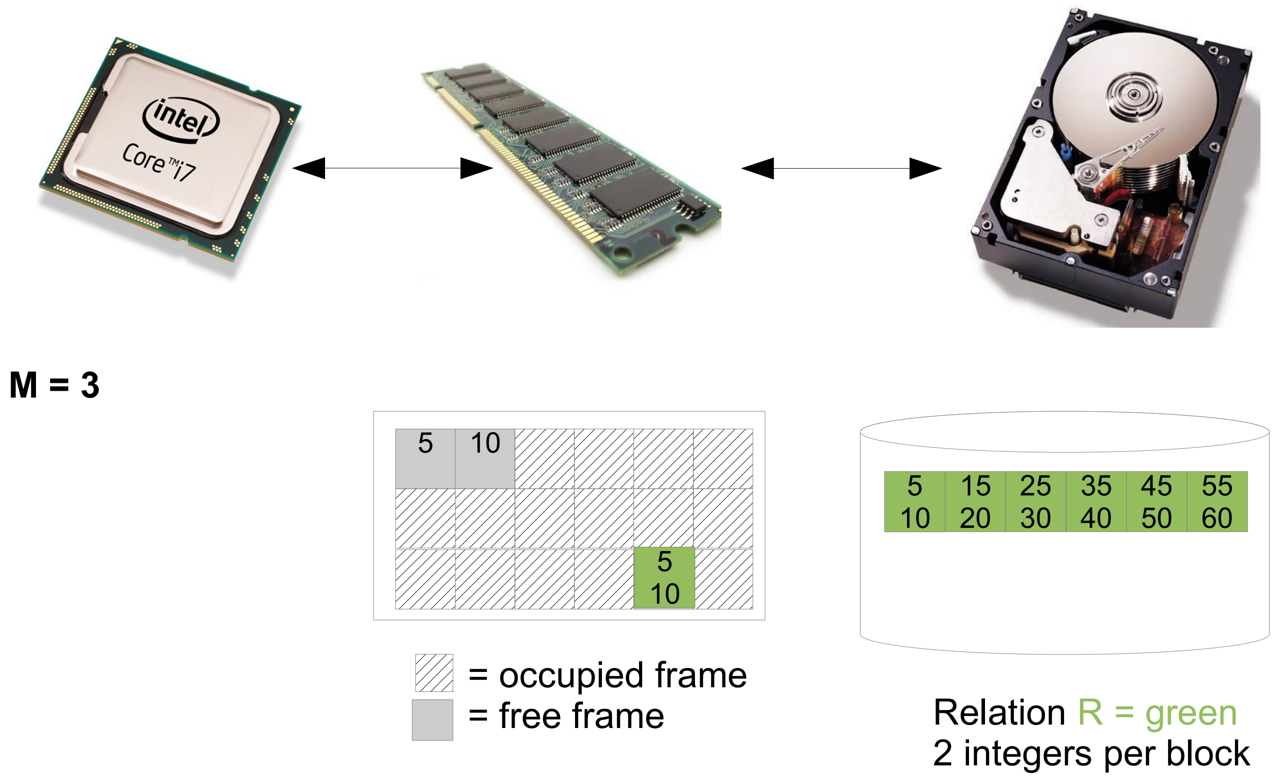


= occupied frame
= free frame

Relation R = green
2 integers per block

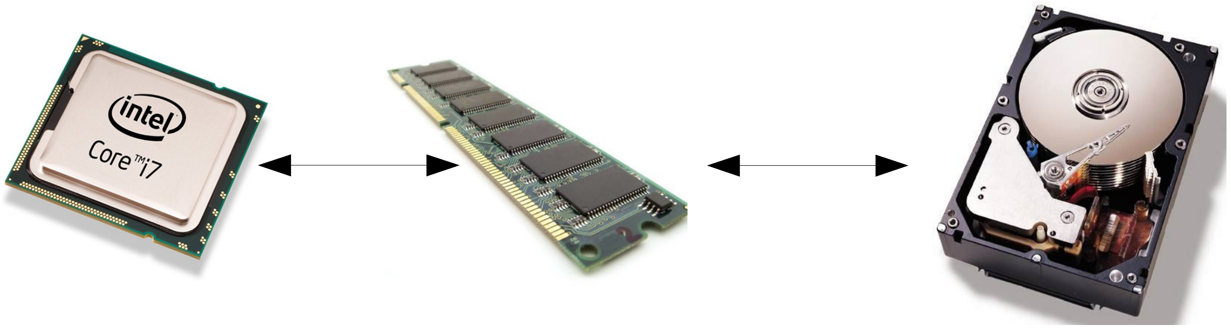
Physical Operators

Hashing R into $M - 1$ buckets using M buffers

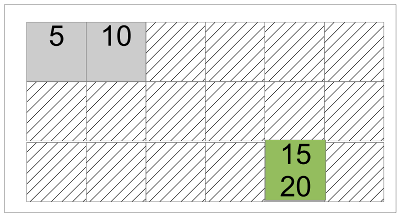


Physical Operators

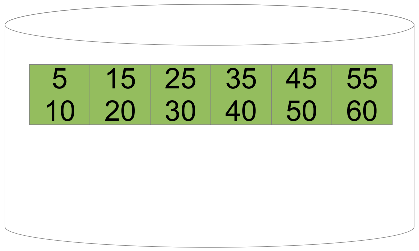
Hashing R into $M - 1$ buckets using M buffers



M = 3



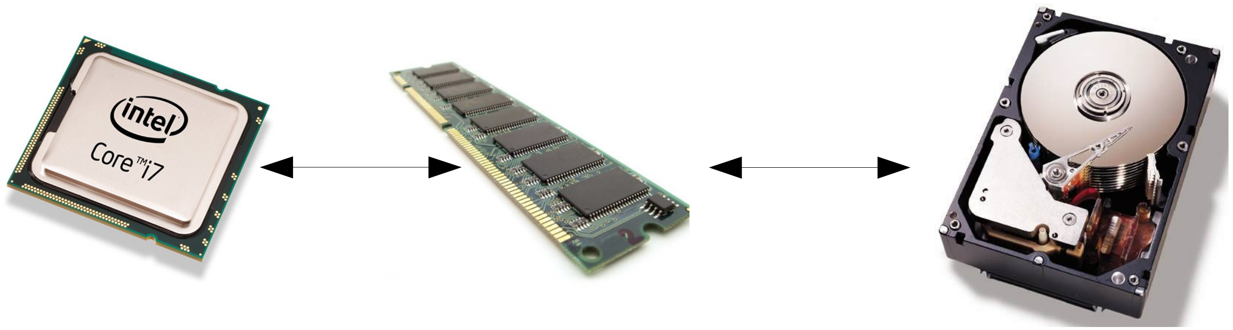
= occupied frame
= free frame



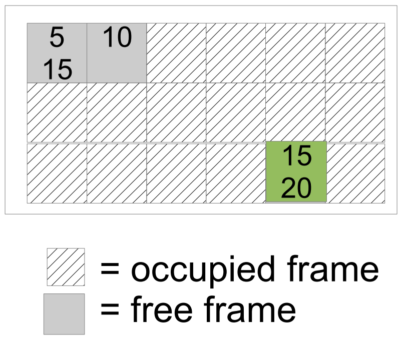
Relation R = green
2 integers per block

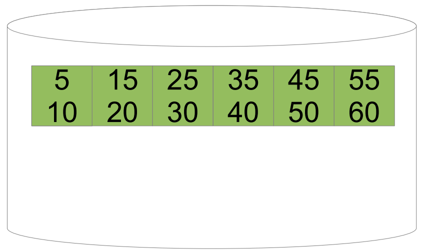
Physical Operators

Hashing R into $M-1$ buckets using M buffers



M = 3

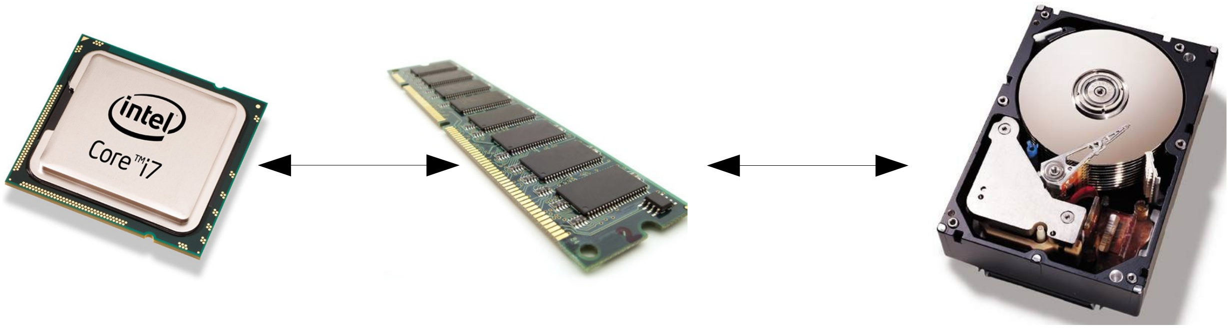




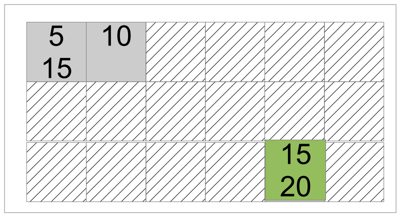
Relation R = green
2 integers per block

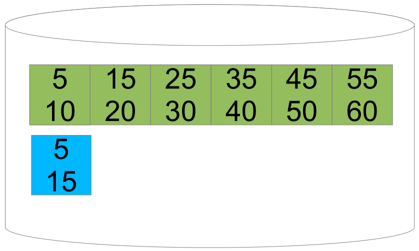
Physical Operators

Hashing R into $M-1$ buckets using M buffers



M = 3



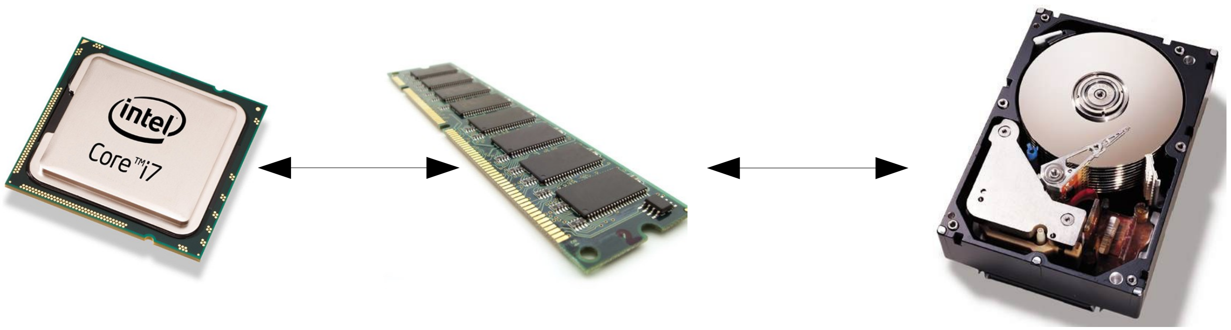


= occupied frame
= free frame

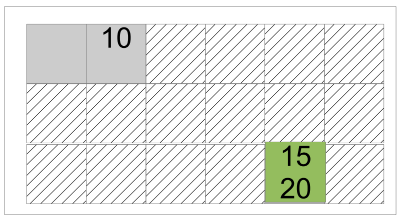
Relation R = green
2 integers per block
Bucket 1 = Blue

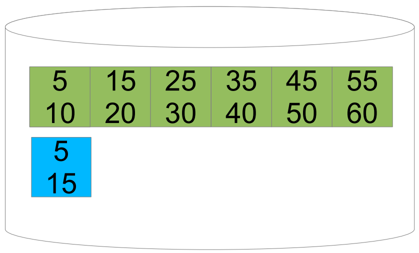
Physical Operators

Hashing R into $M - 1$ buckets using M buffers



M = 3



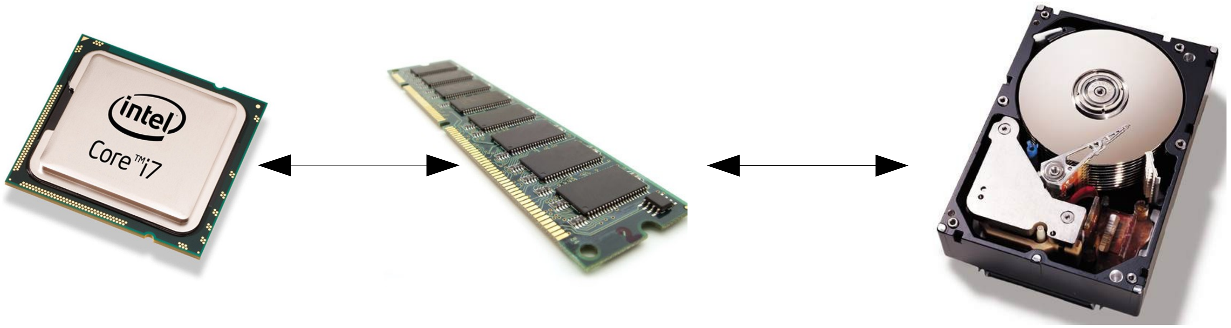


= occupied frame
= free frame

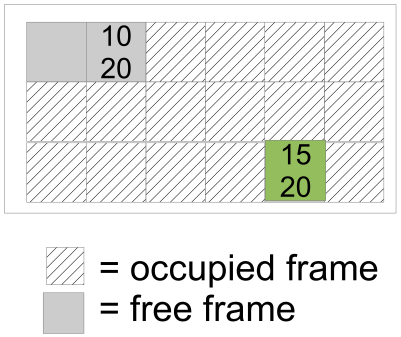
Relation R = green
2 integers per block
Bucket 1 = Blue

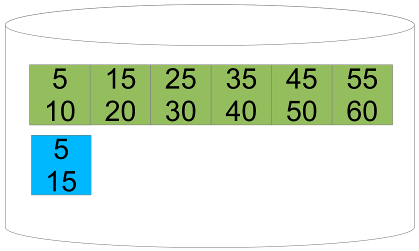
Physical Operators

Hashing R into $M-1$ buckets using M buffers



M = 3

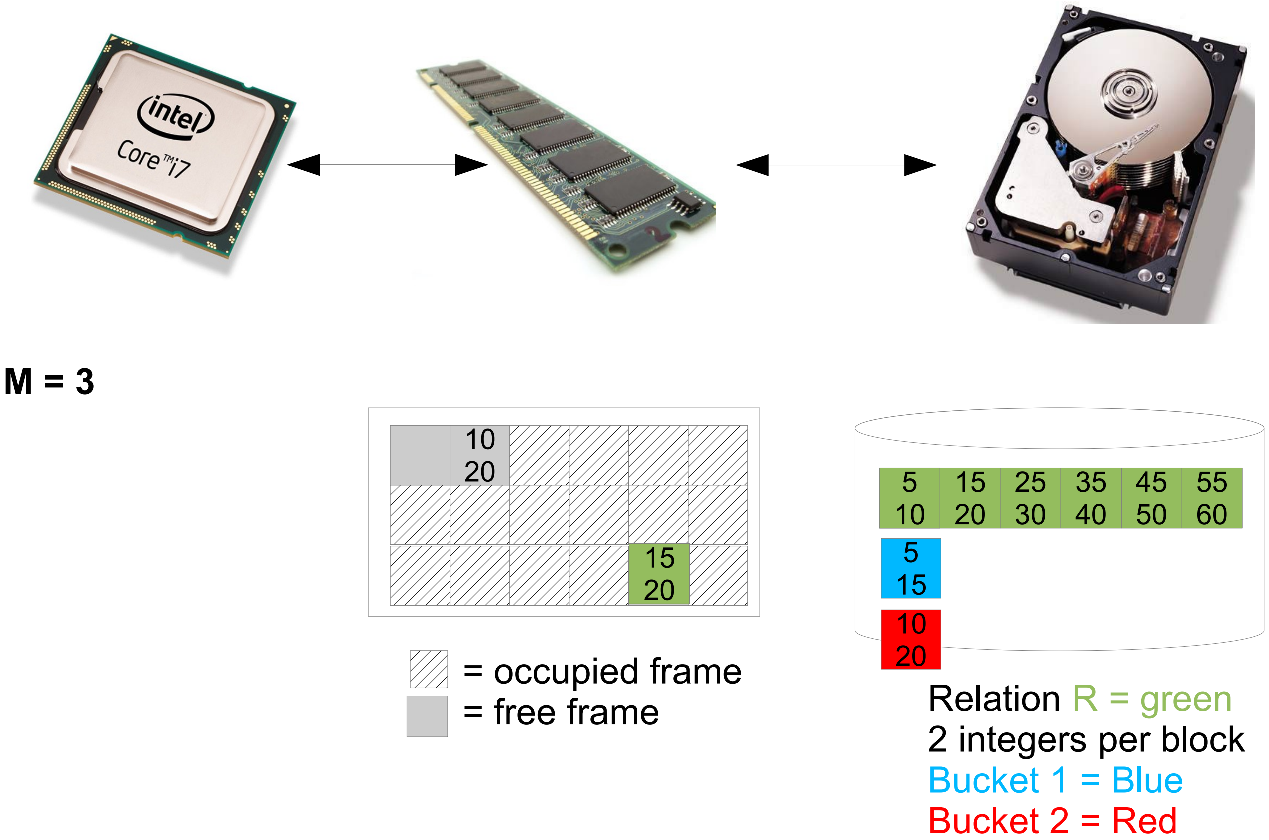




Relation R = green
2 integers per block
Bucket 1 = Blue

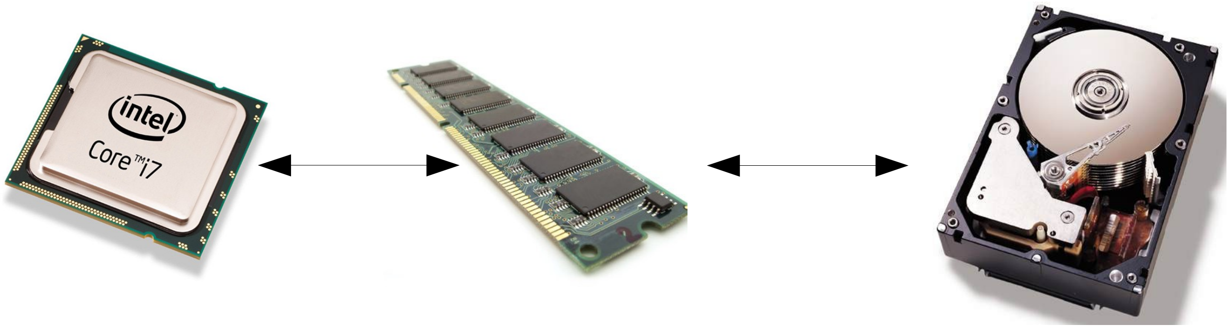
Physical Operators

Hashing R into $M-1$ buckets using M buffers

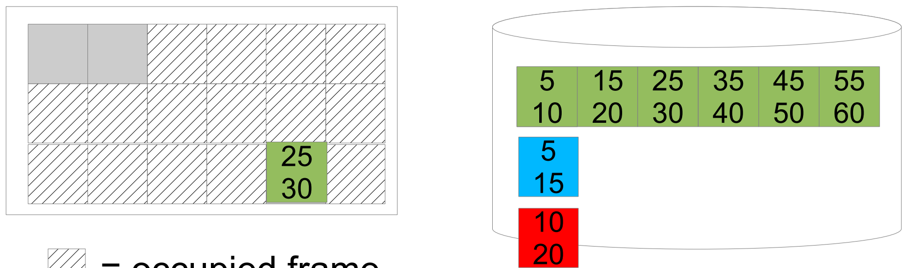


Physical Operators

Hashing R into $M - 1$ buckets using M buffers



M = 3

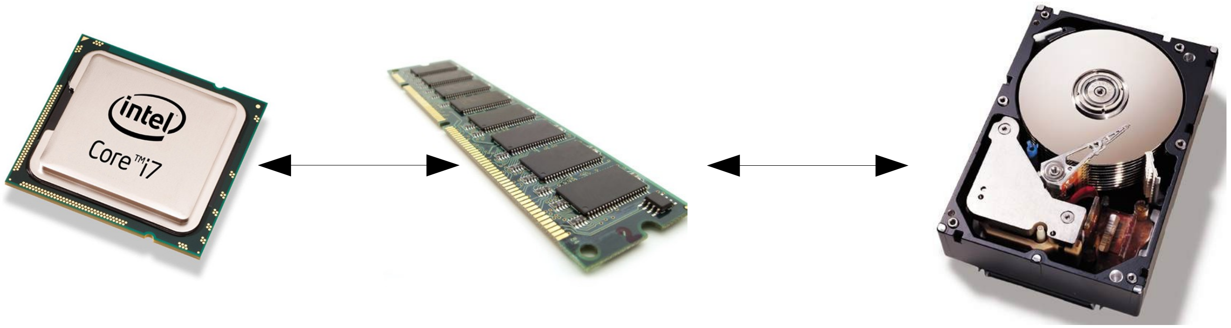


= occupied frame
= free frame

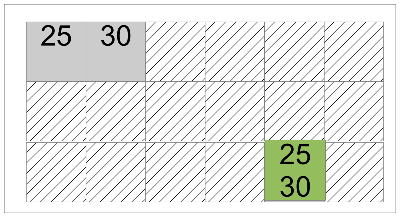
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

Physical Operators

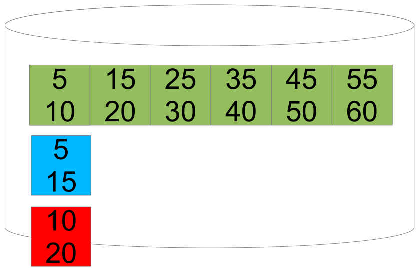
Hashing R into $M-1$ buckets using M buffers



M = 3



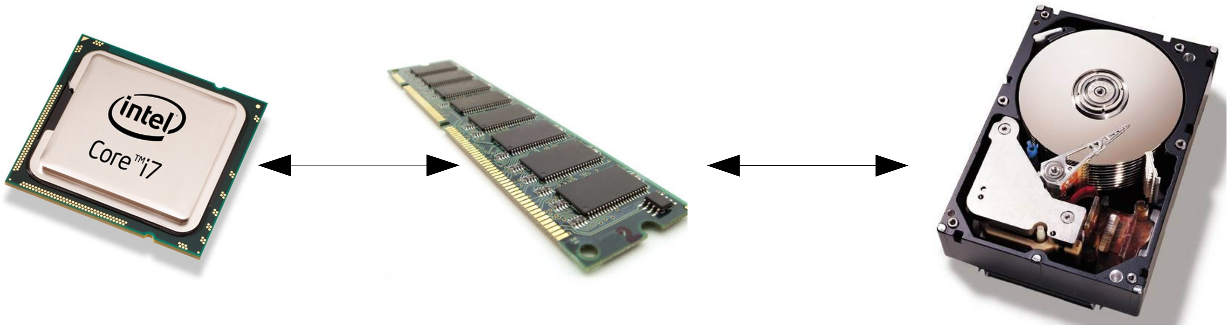
= occupied frame
= free frame



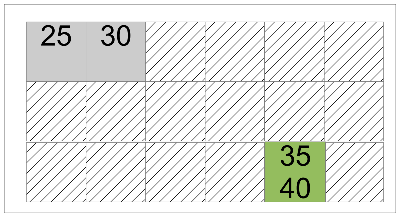
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

Physical Operators

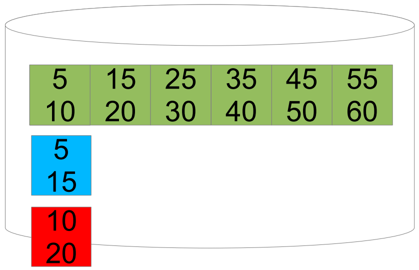
Hashing R into $M-1$ buckets using M buffers



M = 3



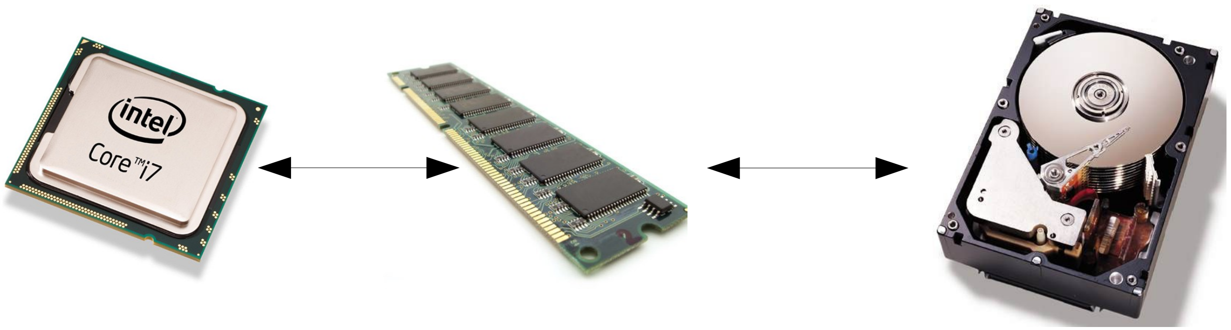
= occupied frame
= free frame



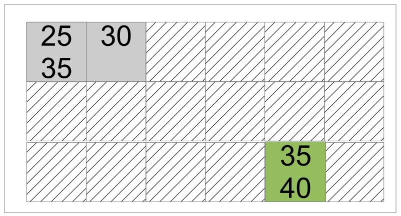
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

Physical Operators

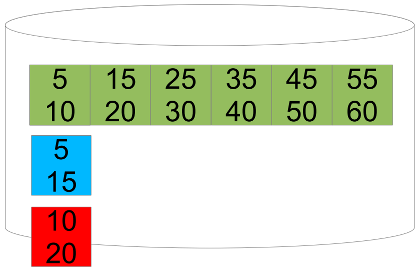
Hashing R into $M - 1$ buckets using M buffers



M = 3



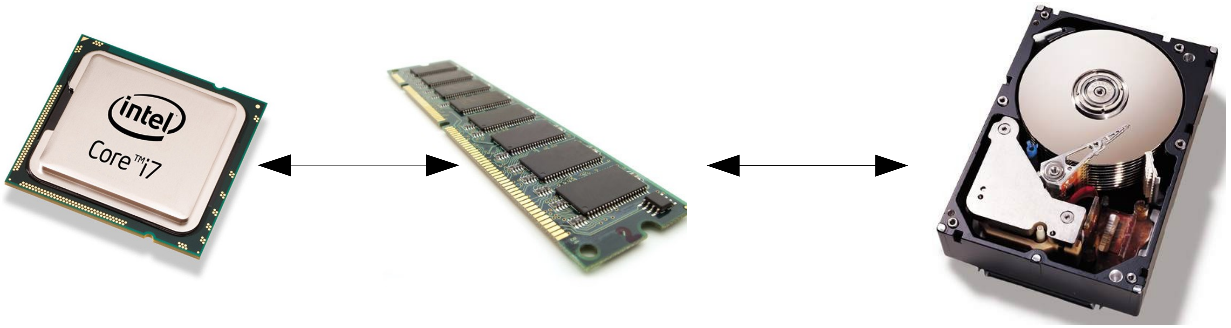
= occupied frame
= free frame



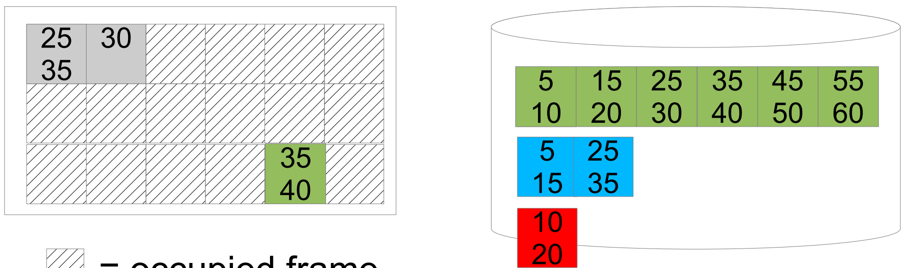
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

Physical Operators

Hashing R into $M - 1$ buckets using M buffers



M = 3

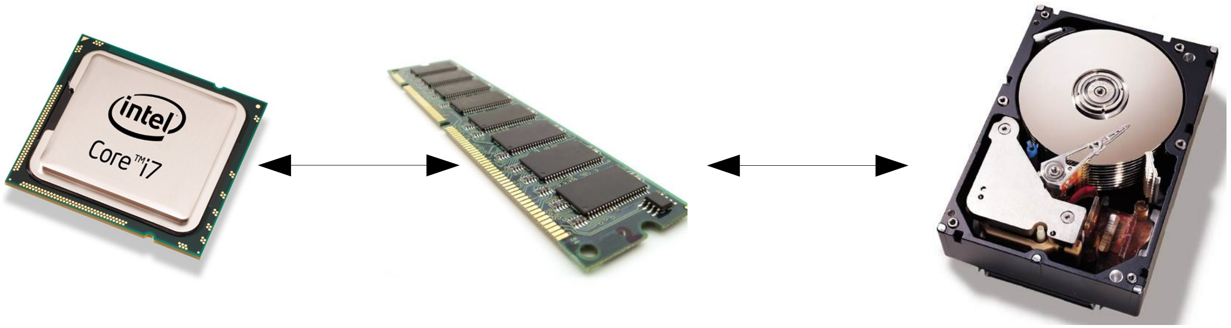


= occupied frame
= free frame

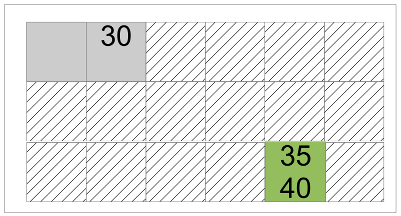
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

Physical Operators

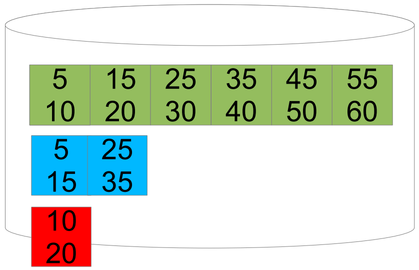
Hashing R into $M-1$ buckets using M buffers



M = 3



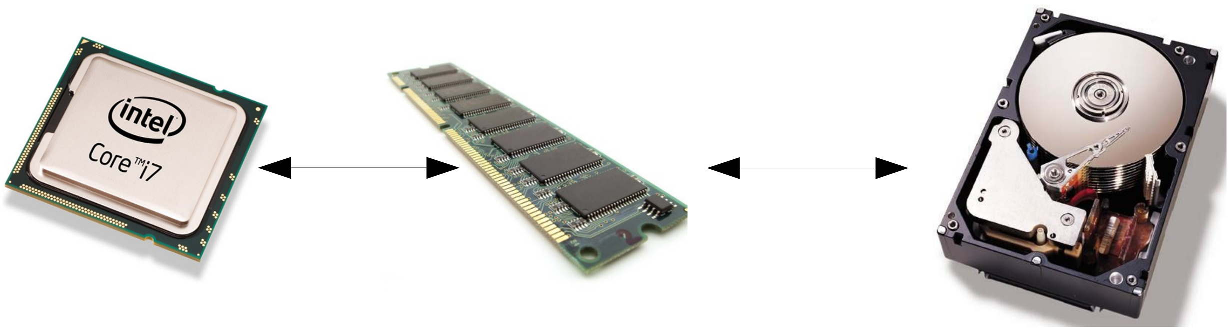
= occupied frame
= free frame



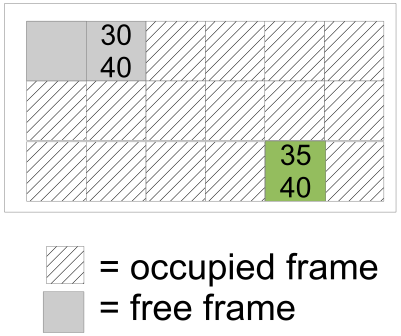
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

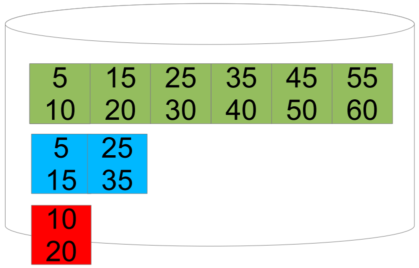
Physical Operators

Hashing R into $M-1$ buckets using M buffers



M = 3

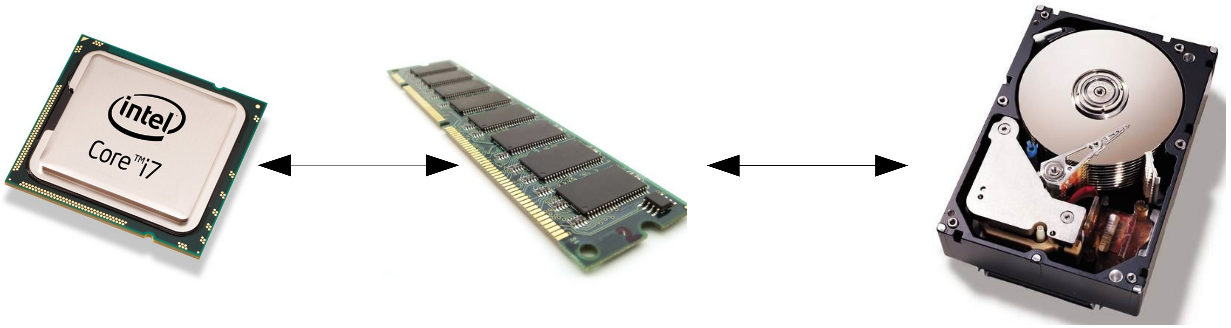




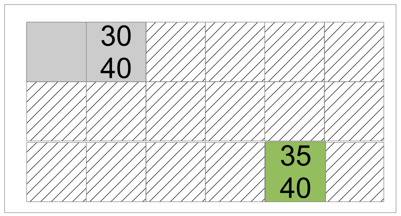
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

Physical Operators

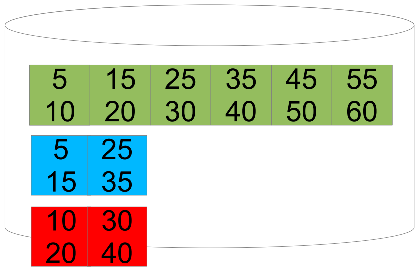
Hashing R into $M - 1$ buckets using M buffers



M = 3



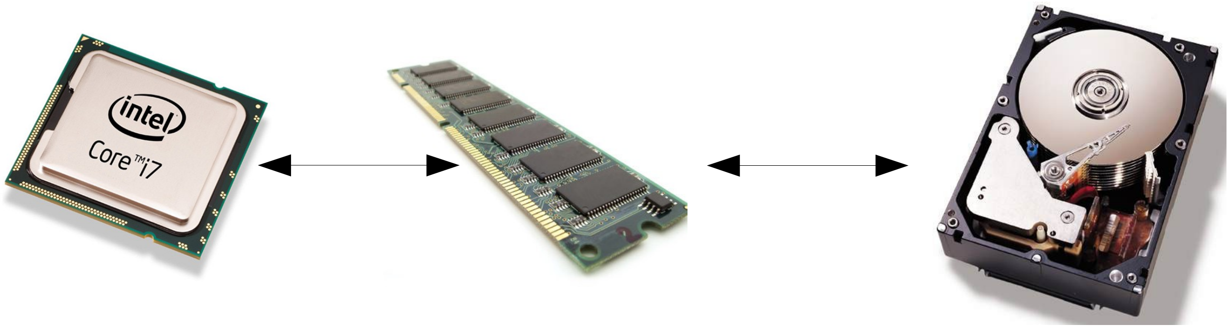
= occupied frame
= free frame



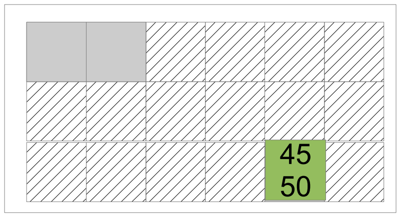
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

Physical Operators

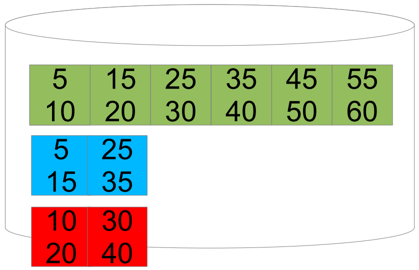
Hashing R into $M-1$ buckets using M buffers



M = 3



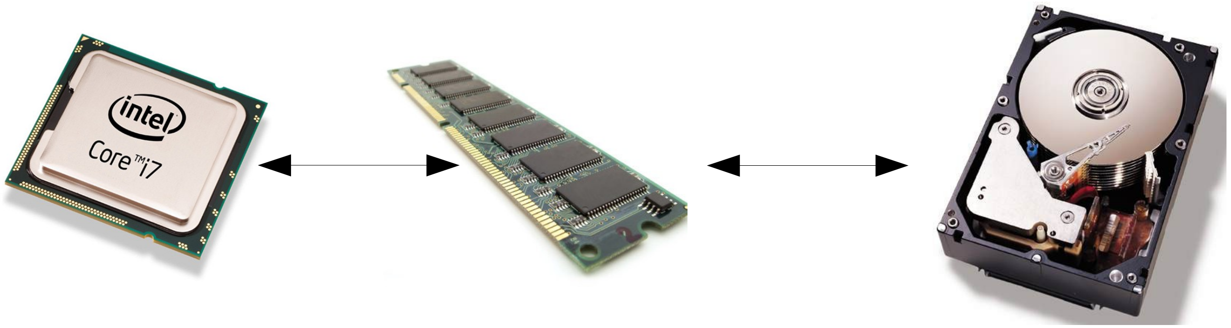
= occupied frame
= free frame



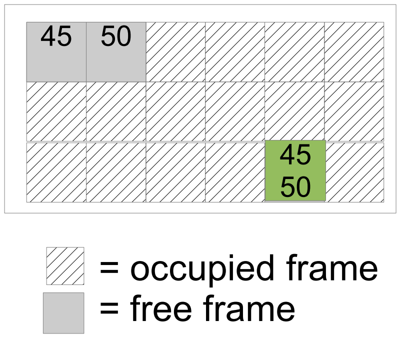
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

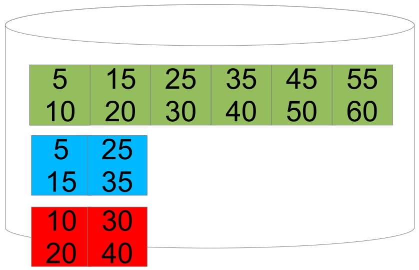
Physical Operators

Hashing R into $M - 1$ buckets using M buffers



M = 3

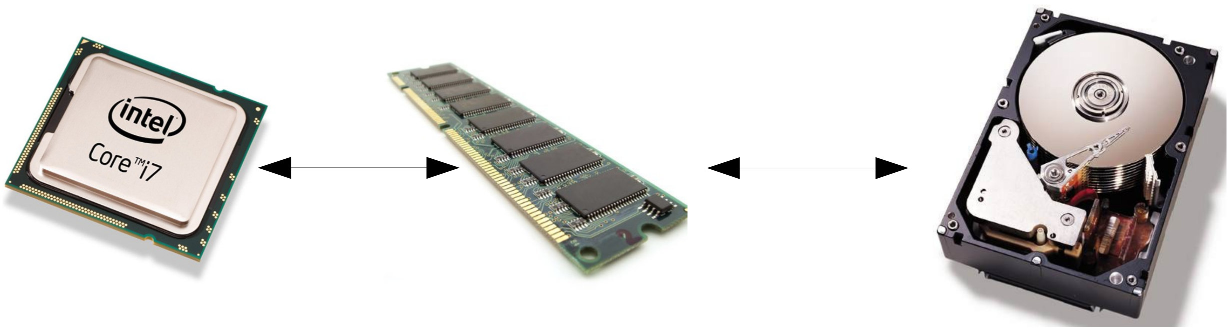




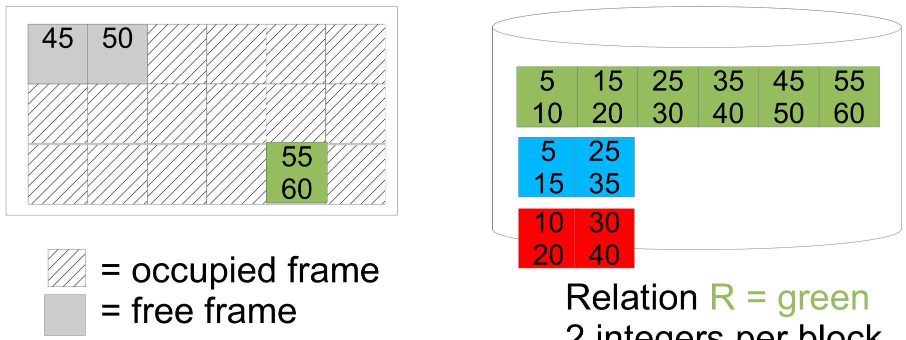
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

Physical Operators

Hashing R into $M - 1$ buckets using M buffers



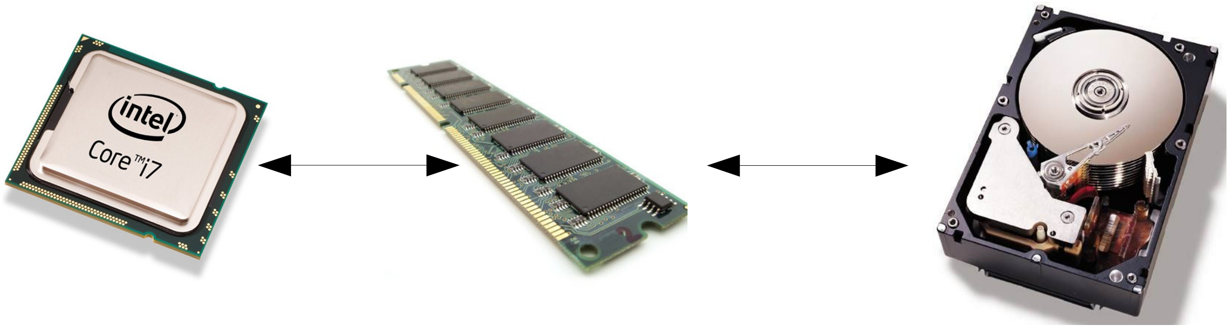
M = 3



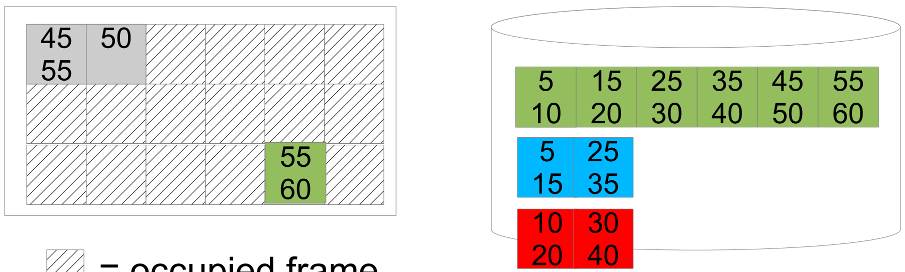
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

Physical Operators

Hashing R into $M-1$ buckets using M buffers



M = 3

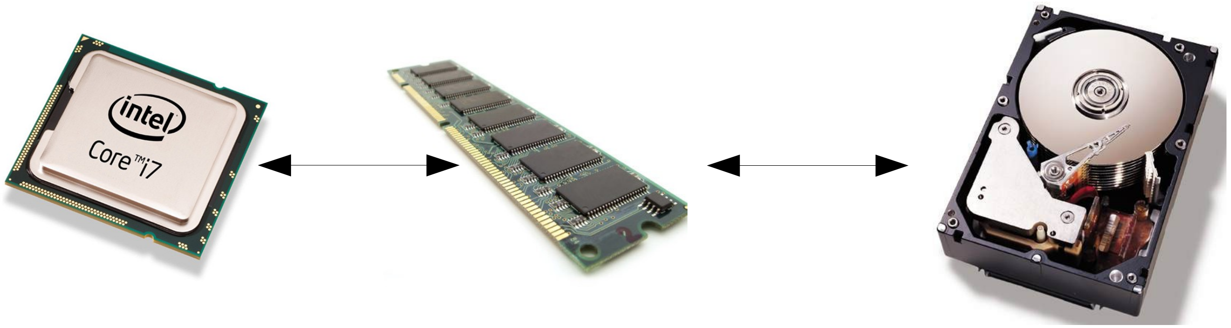


= occupied frame
= free frame

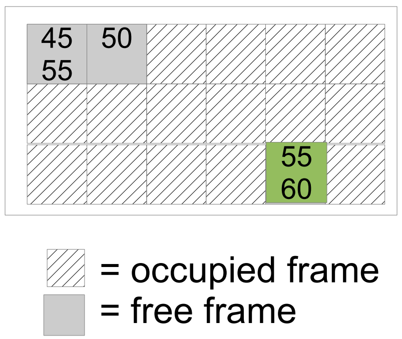
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

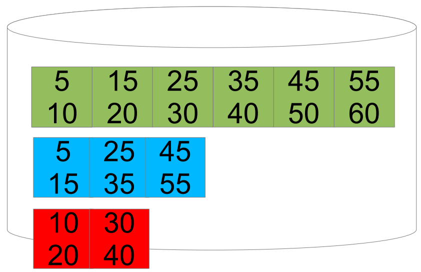
Physical Operators

Hashing R into $M-1$ buckets using M buffers



M = 3

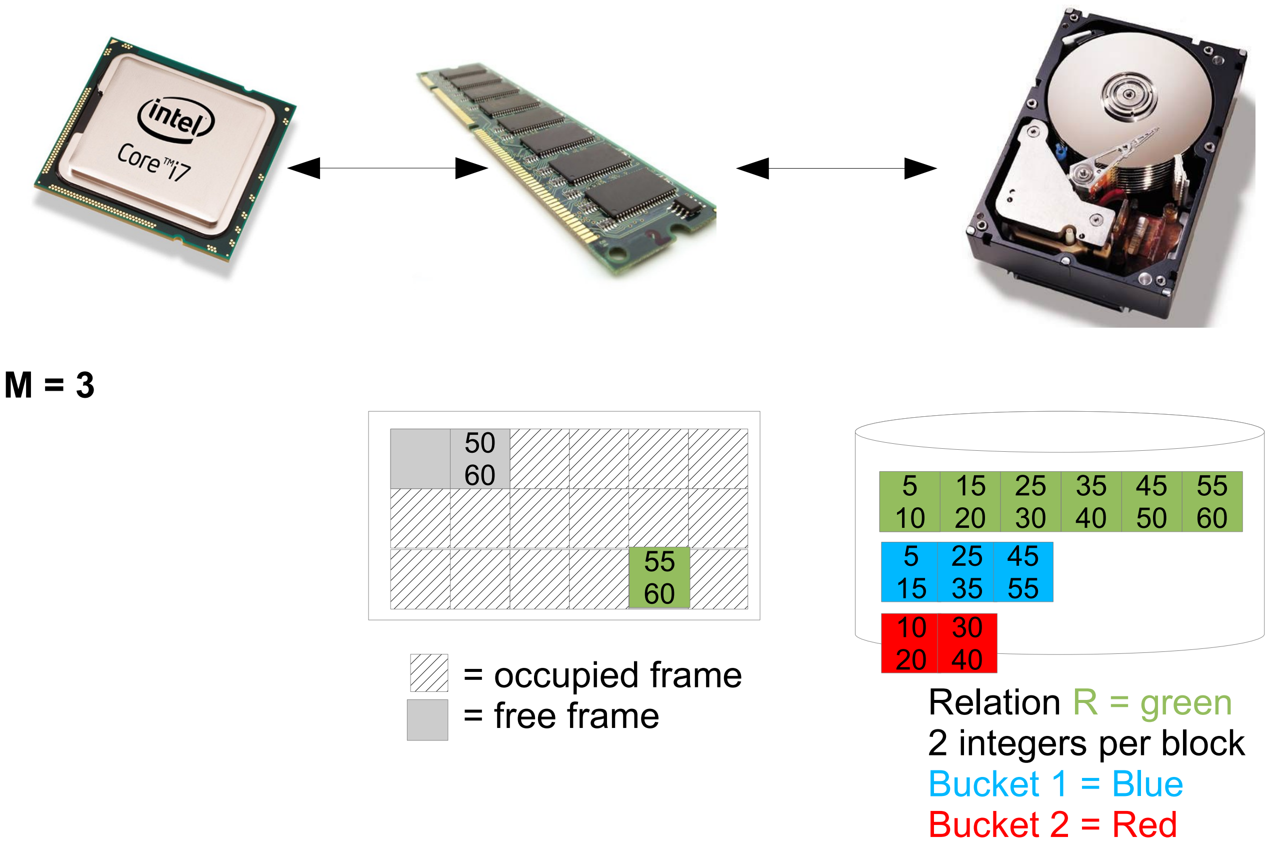




Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

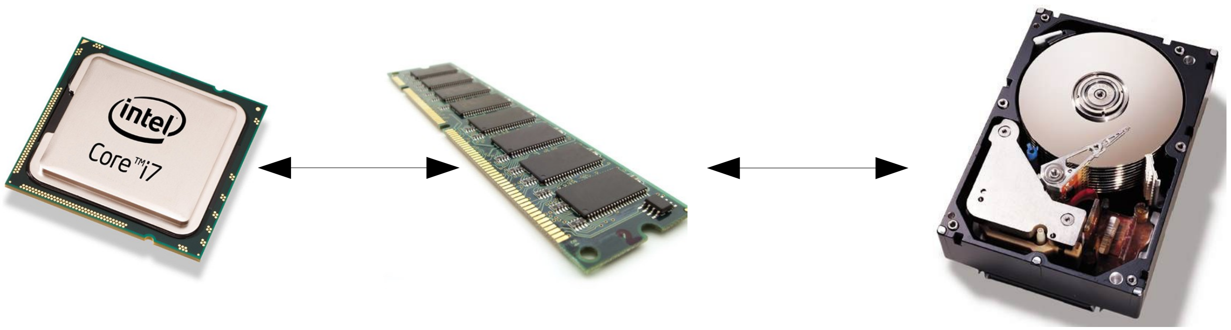
Physical Operators

Hashing R into $M-1$ buckets using M buffers

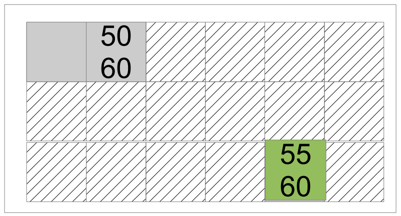


Physical Operators

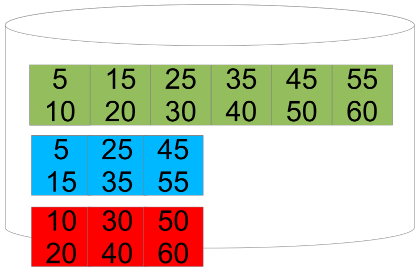
Hashing R into $M - 1$ buckets using M buffers



M = 3



= occupied frame
= free frame



Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

Physical Operators

Hash-based set union

What is the cost of partitioning?

1. Assuming that the hash function(s) distribute the records uniformly, we have $M-1$ buckets of $\frac{B(R)}{M-1}$ blocks after the first pass, $(M-1)^2$ buckets of $\frac{B(R)}{(M-1)^2}$ blocks after the second pass, and so on. Hence, if we reach buckets of at most $M-1$ blocks after k passes, k must satisfy:
$$\frac{B(R)}{(M-1)^k} \leq M-1$$
The minimal value of k that satisfies this is hence $\lceil \log_{M-1} B(R) - 1 \rceil$
2. In every pass we read and write R once.

Total cost:

$$\boxed{2B(R) \lceil \log_{M-1} B(R) - 1 \rceil}$$

Physical Operators

Hash-based set union

What is the costs of calculating hash-based set union?

1. Partition R: $2B(R)\left\lceil \log_{M-1} B(R) - 1 \right\rceil$ I/O's
2. Partition S: $2B(S)\left\lceil \log_{M-1} B(R) - 1 \right\rceil$ I/O's

   Because we "only" need to partition S in as many buckets as R.
3. The one-pass set union of each R_i and S_i: $B(R) + B(S)$

Total:

$$\boxed{2B(R)\left\lceil \log_{M-1} B(R) - 1 \right\rceil + 2B(S)\left\lceil \log_{M-1} B(R) - 1 \right\rceil + B(R) + B(S)}$$

Physical Operators

Hash-based set union

- The book states that in practice one level of partitioning suffices.
- The book hence focuses on the scenario where we only need two passes: "two-pass, hash-based set union" and only sketches the generalization to multiple passes.

  The algorithm is called two-pass because we need 1 pass through the data to partition it, and another one to do the pairwise single-pass union of the buckets
- Under the assumption that one level of partitioning suffices, our cost formula hence specializes to the cost: $3B(R) + 3B(S)$
- But: one level of partitioning only suffices if $\frac{B(R)}{M-1} \leq M - 1$, or (approximately) $B(R) \leq M^2$ (where R is the smaller relation of R and S)

  $\rightarrow$ These are the formulas introduced in the book!

Physical Operators

Other operations on relations

To compute (bag) intersection and (bag) difference we can modify the previous algorithms. The costs remain the same

Also the removal of duplicates can be done using the same techniques.

→ See book!

Physical Operators

One-pass Join

Assume that $M - 1 \geq B(R)$. We can then compute $R(X, Y) \bowtie S(Y, Z)$ as follows:

load R into memory buffers $N_1, \ldots, N_{B(R)}$;
  for each block B_S in S **do**
    load B_S into buffer N_0;
    for each tuple t_S in N_0 **do**
      for each tuple matching tuple t_R in $N_1, \ldots, N_{B(R)}$ **do**
      output $t_R \bowtie t_S$

- Cost: $B(R) + B(S)$ I/O operations
- There is also the cost of finding the matching tuples of t_S in $N_1, \ldots, N_{B(R)}$. By using a suitable main-memory data structure this can be done in $O(n)$ or $O(n \log n)$ time. We ignore this cost.
- Requires $B(R) \leq M - 1$

Physical Operators

Nested Loop Join

We can also alternatively compute $R(X, Y) \bowtie S(Y, Z)$ as follows:

for each segment G of $M - 1$ blocks of R **do**
  load G into buffers $N_1, \ldots, N_{M-1}$;
  for each block B_S in S **do**
    load B_S into buffer N_0;
    for each tuple t_R in $N_1, \ldots, N_{M-1}$ **do**
      for each tuple t_S in N_0 **do**
        if $t_R.Y = t_S.Y$ then **output** $t_R \bowtie t_S$

Cost:

$$\boxed{B(R) + B(S) \times \frac{B(R)}{M - 1}}$$

Physical Operators

Sort-merge Join

Essentially the same algorithm as sort-based set union:

1. Sort R on attribute Y
2. Sort S on attribute Y
3. Iterated synchronously through R and S, keeping 1 block of each relation in memory at all times, and at each point inspecting a single tuple from R and S. Assume that we are currently at tuple t_R in R and at tuple t_S in S.
   - If $t_R.Y < t_S.Y$ then we advance the pointer t_R to the next tuple in R (possibly loading the next block of R if necessary).
   - If $t_R.Y > t_S.Y$ then we advance the pointer t_S to the next tuple in S (possibly loading the next block of S if necessary)).
   - If $t_R.Y = t_S.Y$ then we output $t_R \bowtie t'_S$ for each tuple t'_S following t_S (including t_S itself) that satisfies $t'_S.Y = t_S.Y$. It is possible that we need to read the following blocks in S. Finally, we advance t_R to the next tuple in R, and rewind our pointer in S to t_S.

Sort-merge Join

- The cost depends on the number of tuples with equal values for Y. The worst case is when all tuples in R and S have the same Y-value. The cost is then $B(R) \times B(S)$ plus the cost for sorting R and S.
- However, joins are often performed on foreign key attributes. Assume for example that attribute Y in S is a foreign key to attribute Y in R. Then every value for Y in S has only one matching tuple in R, and there is no need to reset the pointer in S. $\rightarrow$ See book
- In this case the cost analysis is similar to the analysis for sort-based set union. Similarly, it is possible to optimize and gain $2B(R) + 2B(S)$ I/O operations (provided there is enough memory).
- The book also focuses on "two-pass sort-merge join".
- Remark: When R has a BTree index on Y, then it is not necessary to sort R (why?). The same holds for S.

Physical Operators

Hash-Join

Essentially the same algorithm as hash-based set union:

1. Partition, by hashing the Y-attribute, R into buckets of at most $M - 1$ blocks each. Let k be the number of buckets required, and let R_i be the relation formed by the blocks in bucket i.
2. Partition, by hashing the Y-attribute using the same has function(s) as above, S into k buckets. Let S_i be the relation formed by the blocks in bucket i.
   Notice: the records in R_i and S_i have the same hash value. A tuple $t_R \in R$ hence matches the Y attribute of tuple $t_S \in S$ if, and only if, there is a bucket i such that $t_R \in R_i$ and $t_S \in S_i$.
3. We can therefore compute the join by calculating the join of R_i and S_i, for every $i \in 1, \ldots, k$. Since every R_i consists of at most $M - 1$ blocks, this can be done using the one-pass algorithm.

Remark: the output of a hash-join is unsorted on the Y attribute, in contrast to the output of the sort-merge join!

Physical Operators

Hash-Join

- The cost analysis is the same as the analysis for hash-based set union
- Again the book focuses on "two-pass hash-join":

  one pass for the partitioning, one pass for the join

Physical Operators

Index-Join

Assume that S has an index on attribute Y. We can then alternatively compute the join $R(X, Y) \bowtie S(Y, Z)$ by searching, for every tuple t in R, the matching tuples in S (using the index).

Cost when the index on Y is not clustered:

$$B(R) + T(R) \times \lceil T(S)/V(S, Y) \rceil$$

Cost when the index on Y is clustered:

$$B(R) + T(R) \times \lceil B(S)/V(S, Y) \rceil$$

$\rightarrow$ See book

General comment

The book often omits the ceiling operations ($\lceil \cdot \rceil$) when calculating costs. In the exercises you must always include these operations!