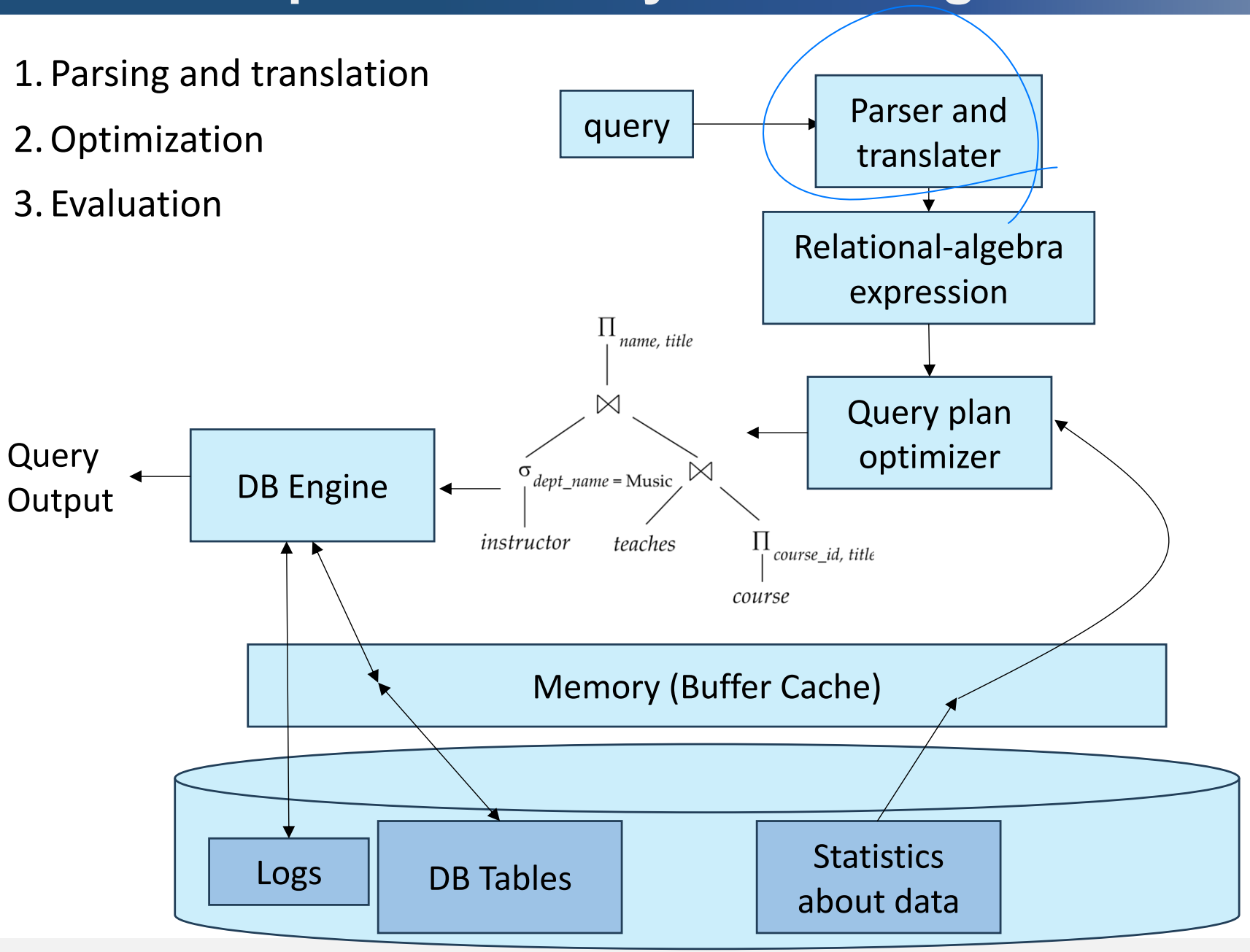
Database Systems
Lecture14 – Chapter 15: Query Planning

Beomseok Nam (남범석)
bnam@skku.edu

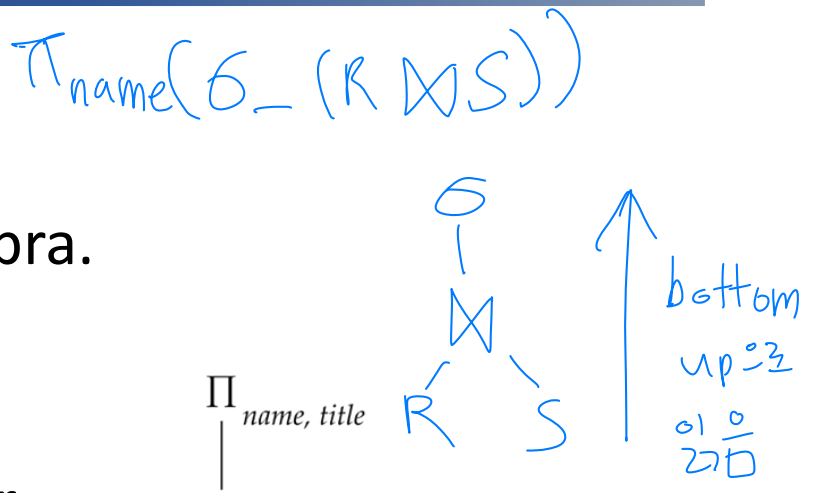
Basic Steps in DB Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation

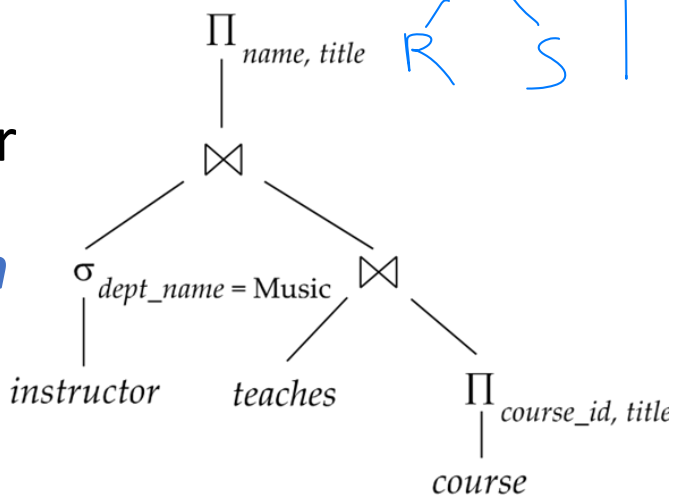


Basic Steps in DB Query Processing

- Parsing and translation
  - Parser checks syntax, verifies relations
  - translate the query into relational algebra.



- Relational algebra
  - A sequence of binary or unary operator can be transformed into a binary tree, which is called a ***query evaluation plan***



- Evaluation
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns answers to the query.

Basic Steps in Query Processing : Optimization

- A relational algebra expression has many equivalent expressions
  - E.g.,

$$\sigma_{salary<75000}(\Pi_{salary}(instructor))$$

is equivalent to

$$\Pi_{salary}(\sigma_{salary<75000}(instructor))$$

- A sequence of operations to evaluate a query is a **query-evaluation-plan**.
  - For a query, there can be multiple query-evaluation plans
  - E.g., can use an index on *salary* to find instructors with salary < 75000,
  - or can perform table scan and discard instructors with salary ≥ 75000

Basic Steps: Optimization (Cont.)

- **Query Optimization**: Choose the evaluation plan with the **lowest cost** among all equivalent plans
  - Cost is estimated using statistics from the metadata catalog
    - e.g. number of tuples, size of tuples, etc.

- This chapter covers
  - How to measure query costs
  - Algorithms for evaluating relational algebra operations
  - Combining algorithms to evaluate full expressions

Measures of Query Cost

- Query cost = total time to answer
  - Influenced by d*isk accesses and CPU*
- Disk access is the main cost and easy to estimate.
  - We ignore CPU costs for simplicity
    - Real systems do take CPU cost into account
- Disk access is measured by
  - Number of seeks * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * average-block-write-cost
    - Cost to write a block is greater than cost to read a block

Disk access Cost
(I/O 접근)

Measures of Query Cost (Cont.)

- For simplicity we just use the **number of block transfers** *from disk and the* **number of seeks** as the cost measures
  - ***transfer_time*** **– time to transfer one block**
  - ***seek_time*** **– time for one seek**
  - Cost for b block transfers plus S seeks

    *b * transfer_time + S * seek_time*

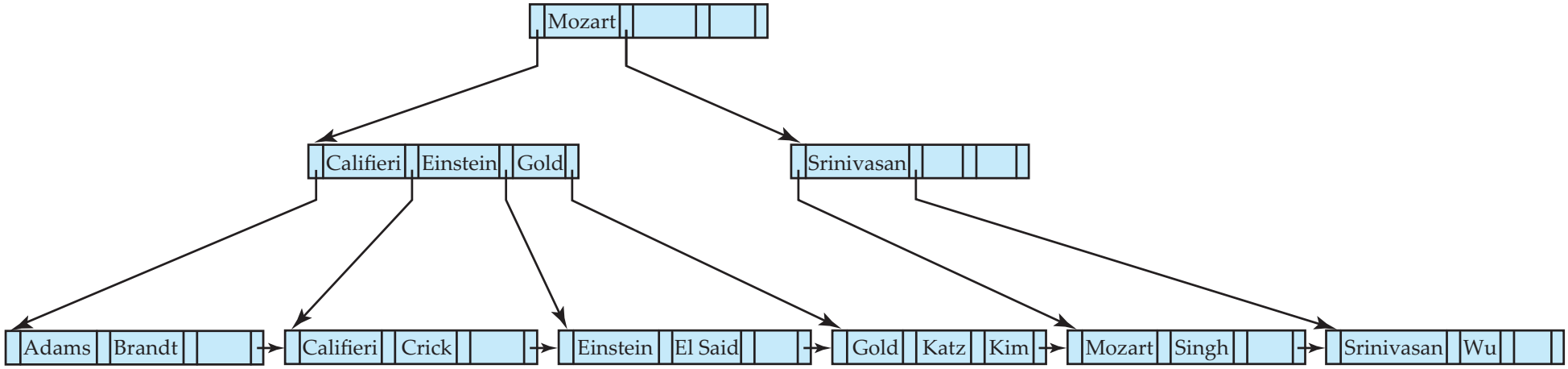
- Note that we will ignore the cost for writing output to disk

Selection Operation

- **Linear search.** Scan all file blocks and check each record for a match
  - We assume relation r is stored in contiguous b_r blocks.
  - Cost
    - non-unique attribute: **b_r block transfers + 1 seek**
    - unique attribute: **0.5 x b_r block transfers + 1 seek**
      - 0.5 because of the average-case assumption
  - Linear search can be applied regardless of
    - selection condition
    - ordering of records
    - availability of index

Selections Using Indices

- **B+tree index, equality on key**
  - *Cost* = (*h*+1)* (*transfer_time* + *seek_time*)
  - *h : height of the B+tree*
  - *+1 : table access*



  - *B+tree nodes are not contiguously stored on disks*
    - *Each node access requires a disk seek*

Implementation of Complex Selections

- **Conjunction:** $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \theta n}(r)$
- **Using one index**.
  - Select one condition θ_i that can use the most efficient algorithm (e.g., B+tree index)
  - Test remaining conditions in memory after fetching records
- **Using intersection of identifiers**.
  - Requires indices with record pointers for multiple conditions.
  - Use an index for each condition
  - Take intersection of record pointers
  - Then fetch records from file

Implementation of Complex Selections

- **Disjunction:** $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \theta n}(r)$.
- **Using union of identifiers**.
  - if *all* conditions have available indices.
    - Take the union of record pointers only
  - Otherwise
    - use linear scan.

- **Negation:** $\sigma_{\neg\theta}(r)$
  - Use linear scan on file
  - Use index if very few records satisfy $\neg\theta$ and an index is applicable to θ

Sorting

- Sorting plays an important role in DBMS.

- For relations that fit in memory, quicksort can be used.
- For relations that don’t fit in memory, do **external sort-merge**

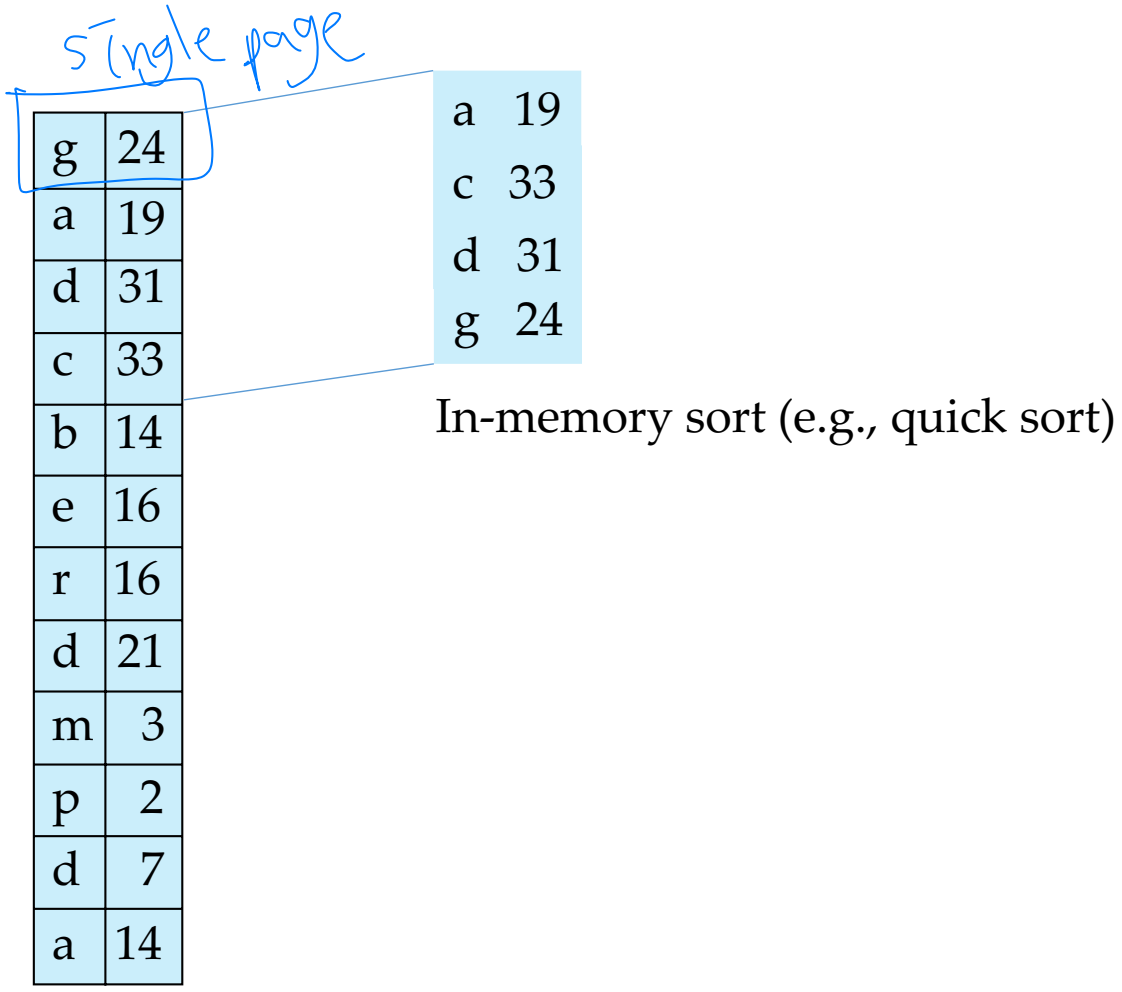
External Sort-Merge (with memory size M pages)

1. **Create sorted runs.**
   Repeat until end of data:
   (a) Read *M* blocks and sort them in memory
   (c) Write sorted data to run R_i; increment *i.*
   Size of each Run = M & Number of Runs = *N*

2. *Merge Runs (N-way merge, assuming N < M)*
   Use *N* pages for input buffers and 1 page for output buffer
   Read the first page of each run into memory
   Repeat
   1. Select the smallest record across all buffer pages
   2. Write it to the output buffer (flush to disk if full)
   3. Delete the record from its input buffer page.
      If the buffer page is empty **then**
      read the next block from the run

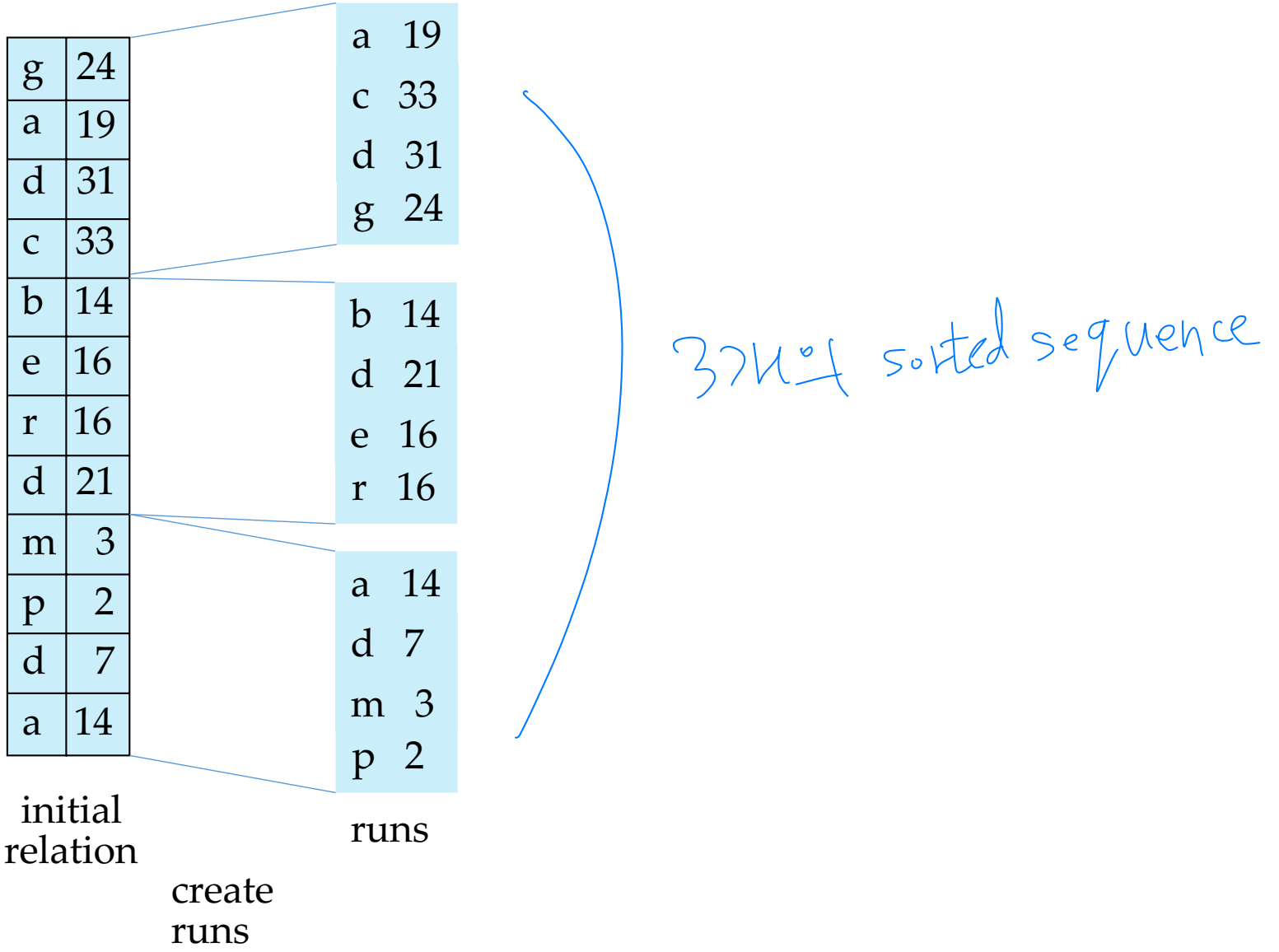
   until all input buffer pages are empty:

Example: External Sorting Using Sort-Merge



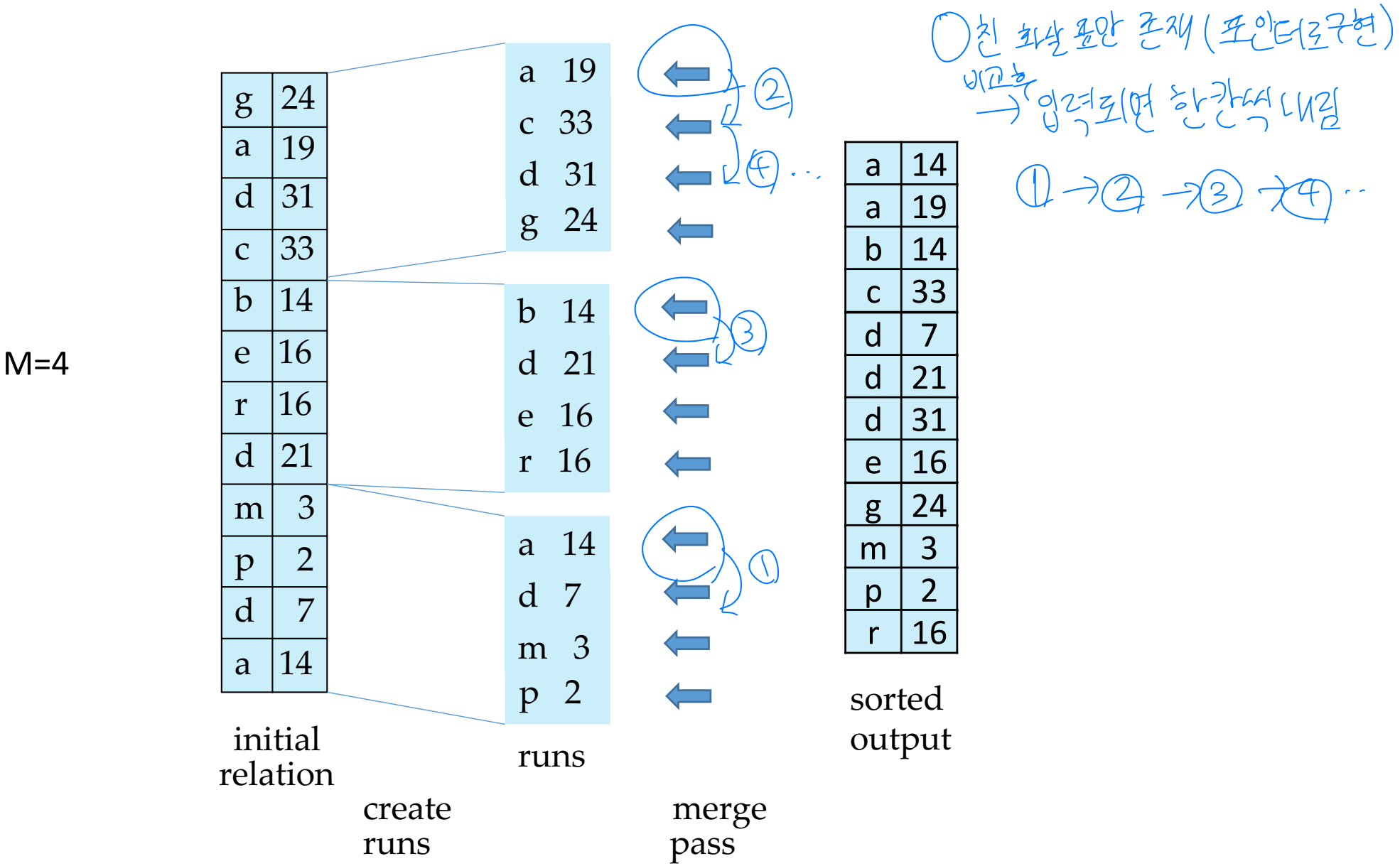
Example: External Sorting Using Sort-Merge

M=4



Example: External Sorting Using Sort-Merge

M=4



External Sort-Merge (Multi-Pass)

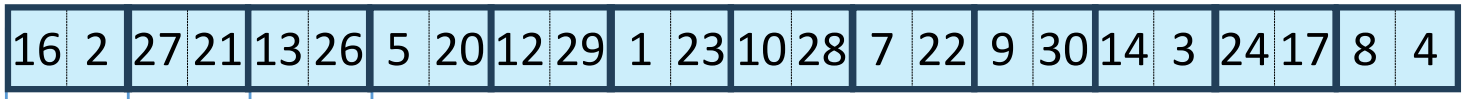
- If number of Runs $N \geq M$ *(more runs than available memory)*

  -- multiple merge *passes* are required.

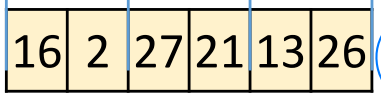
  - In each pass, merge M - 1 runs
  - Each pass:
    - Reduce number of runs by a factor of M -1
    - Increase run length by the same factor
    - E.g. If M=11, and there are 90 runs, one pass reduces the number of runs to 9, each 10 times the size of the initial runs
  - Repeat passes until all runs are merged into one.

Example: Multi-pass External Sort-Merge

- Suppose each page can hold two records
- Buffer Size M = 3 (2 for input and 1 for output)

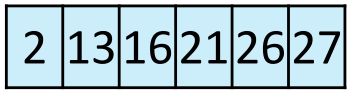


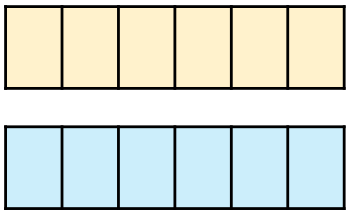
Initial input



Buffer (no need to have a separate output buffer for in-memory sort)

Sorted run



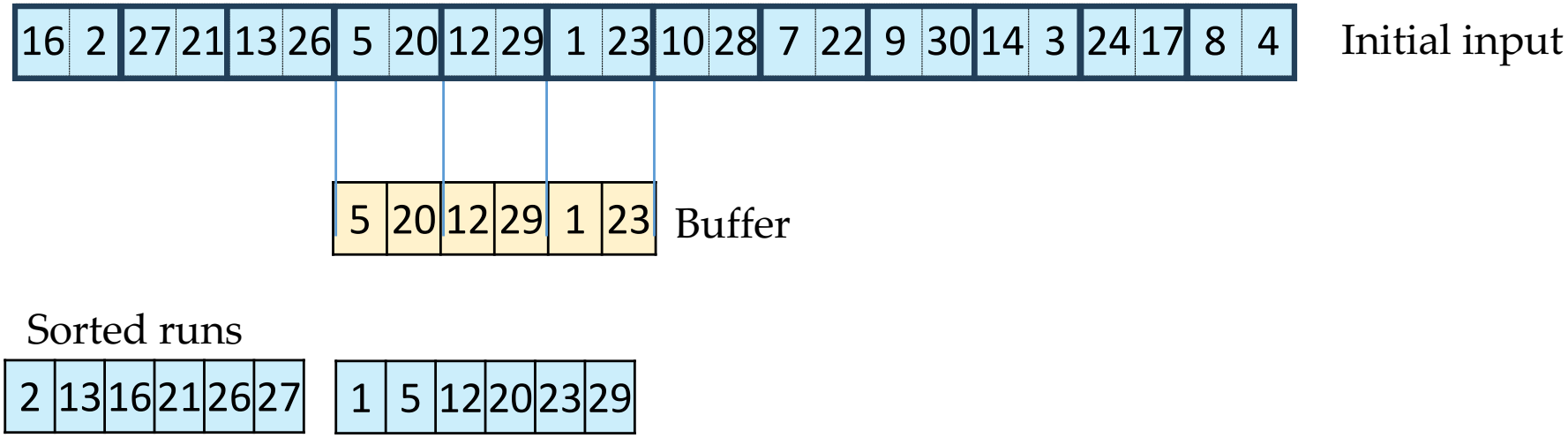


Buffer Cache in DRAM

File on Storage

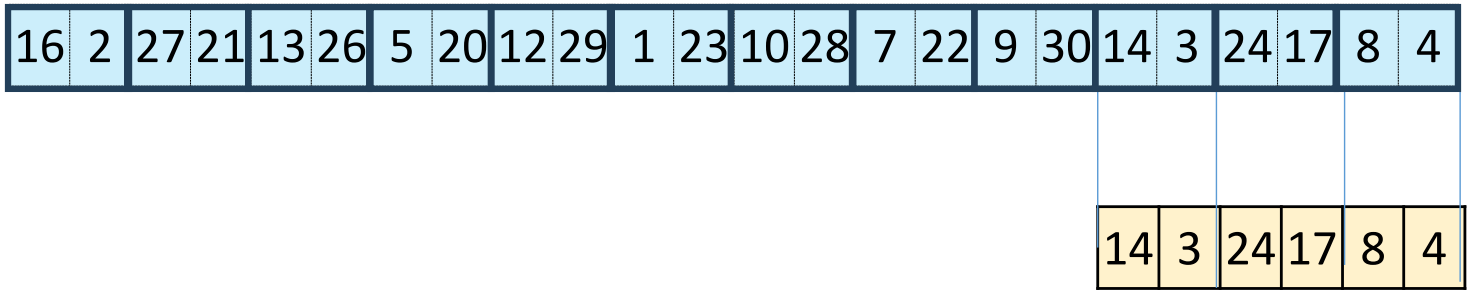
Example: Multi-pass External Sort-Merge

- Suppose each page can hold two records
- Buffer Size M = 3 (2 for input and 1 for output)

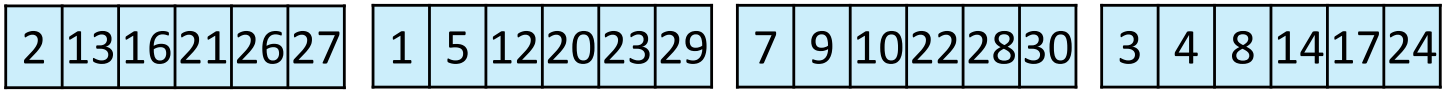


Example: Multi-pass External Sort-Merge

- Suppose each page can hold two records
- Buffer Size M = 3 (2 for input and 1 for output)



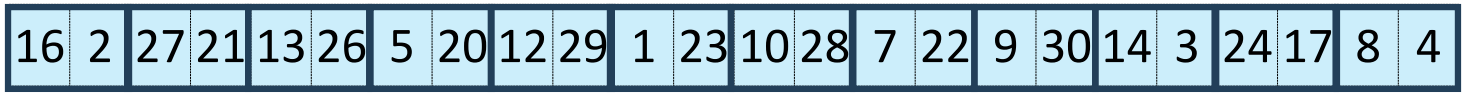
Sorted runs



Number of Block Transfer:
$2*b_r$

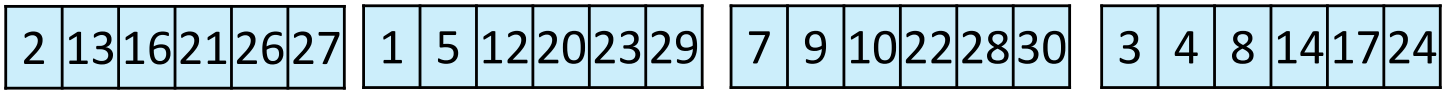
Example: Multi-pass External Sort-Merge

- Suppose each page can hold two records
- Buffer Size M = 3 (2 for input and 1 for output)



Initial input

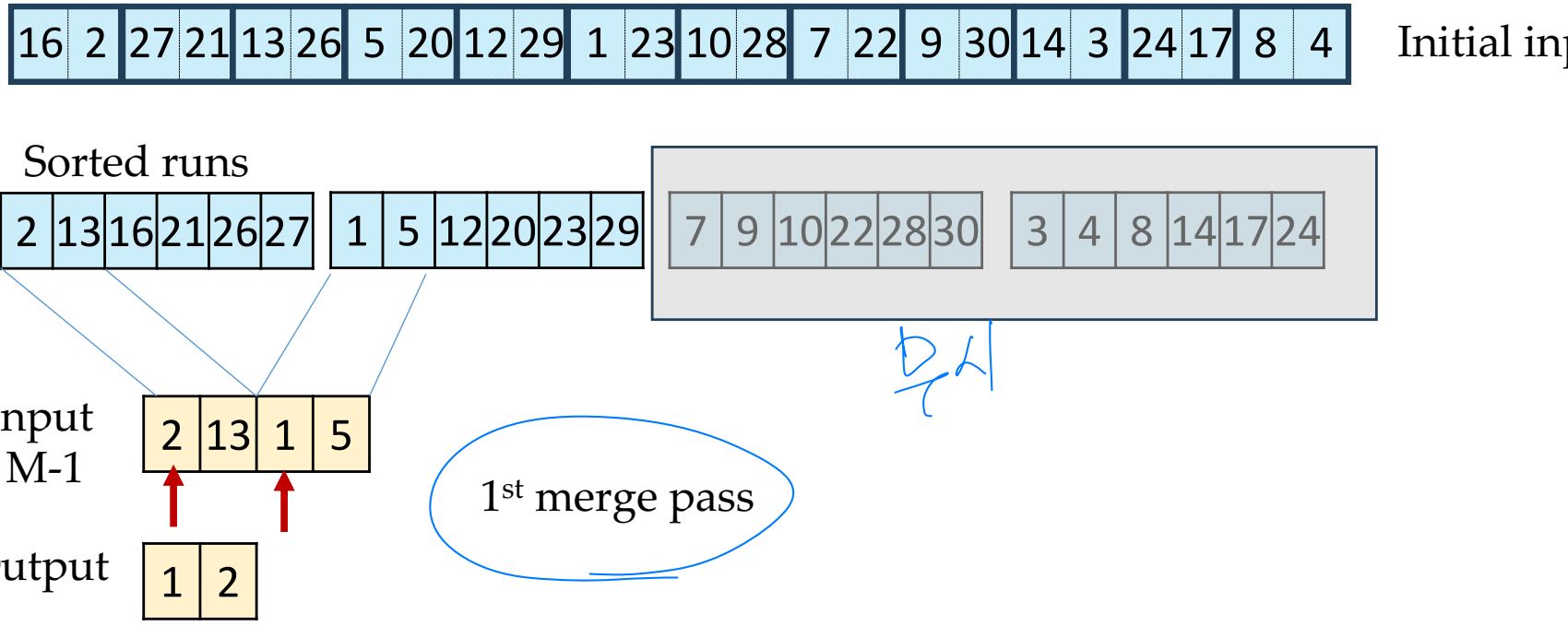
Sorted runs



4 Runs, but 2 input buffers

Example: Multi-pass External Sort-Merge

- Suppose each page can hold two records
- Buffer Size M = 3 (2 for input and 1 for output)

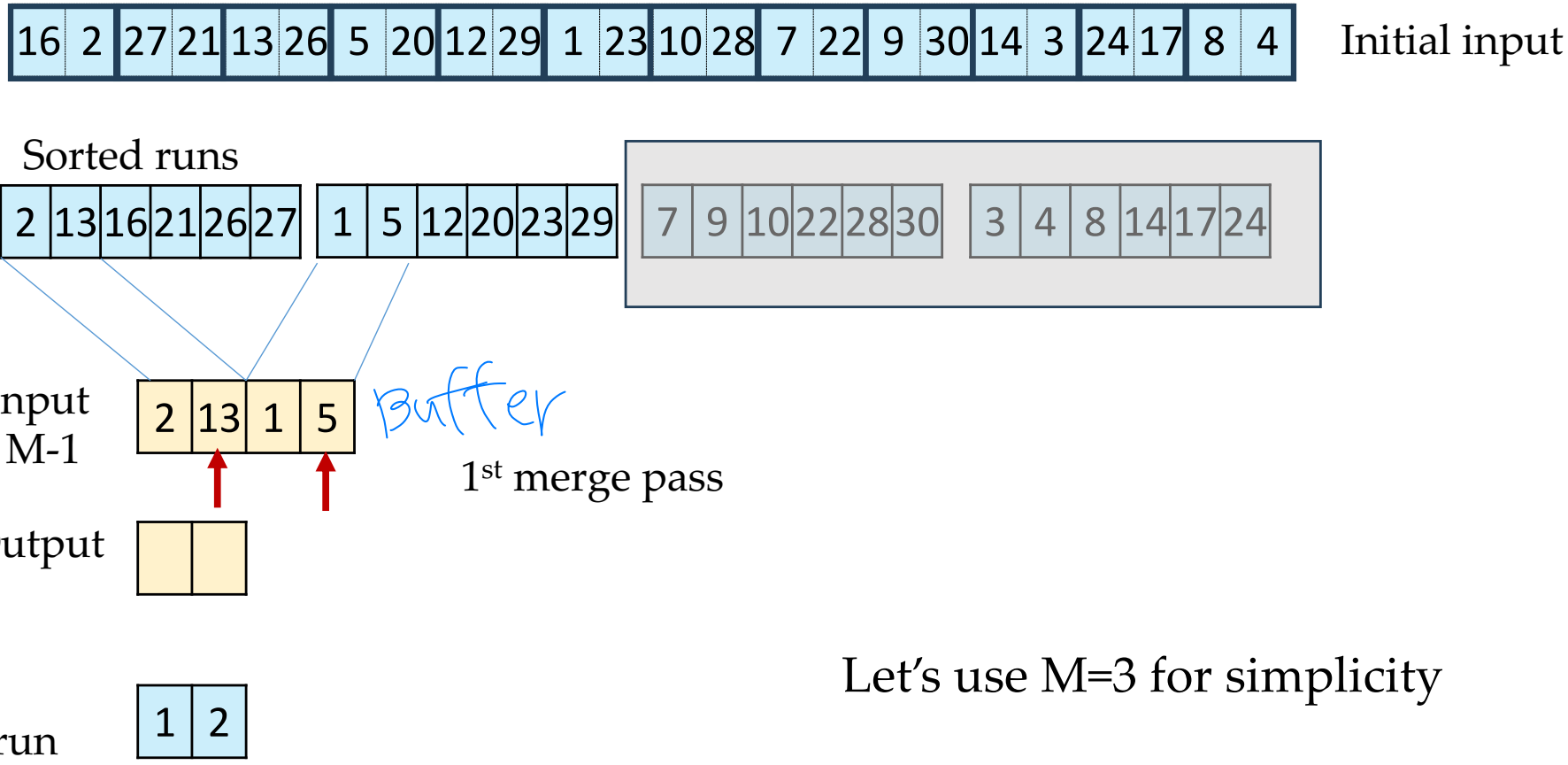


Read one block per run
If M=3, merge run#1 and #2 using 2 pages

run

Example: Multi-pass External Sort-Merge

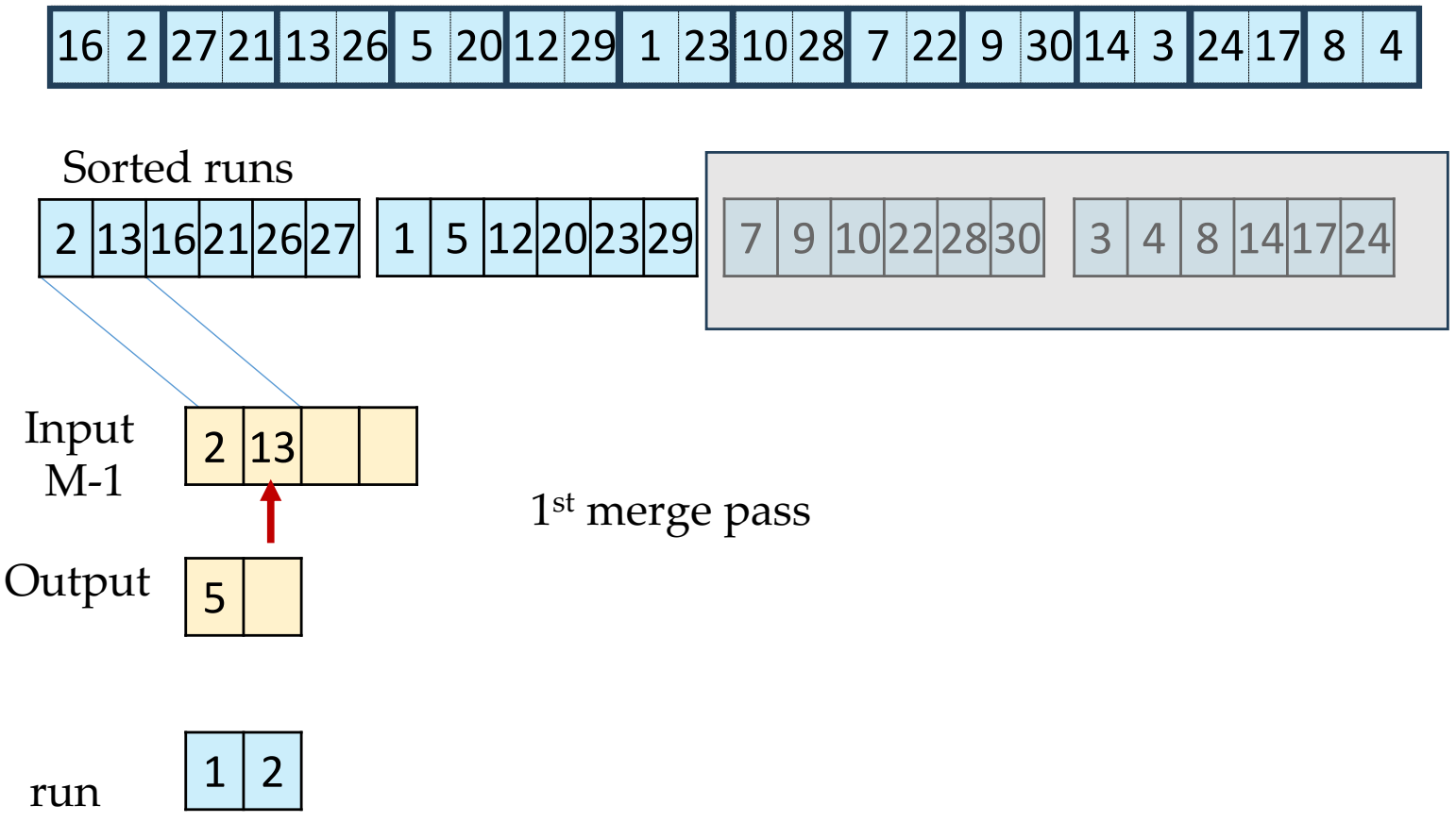
- Suppose each page can hold two records
- Buffer Size M = 3 (2 for input and 1 for output)



Let's use M=3 for simplicity

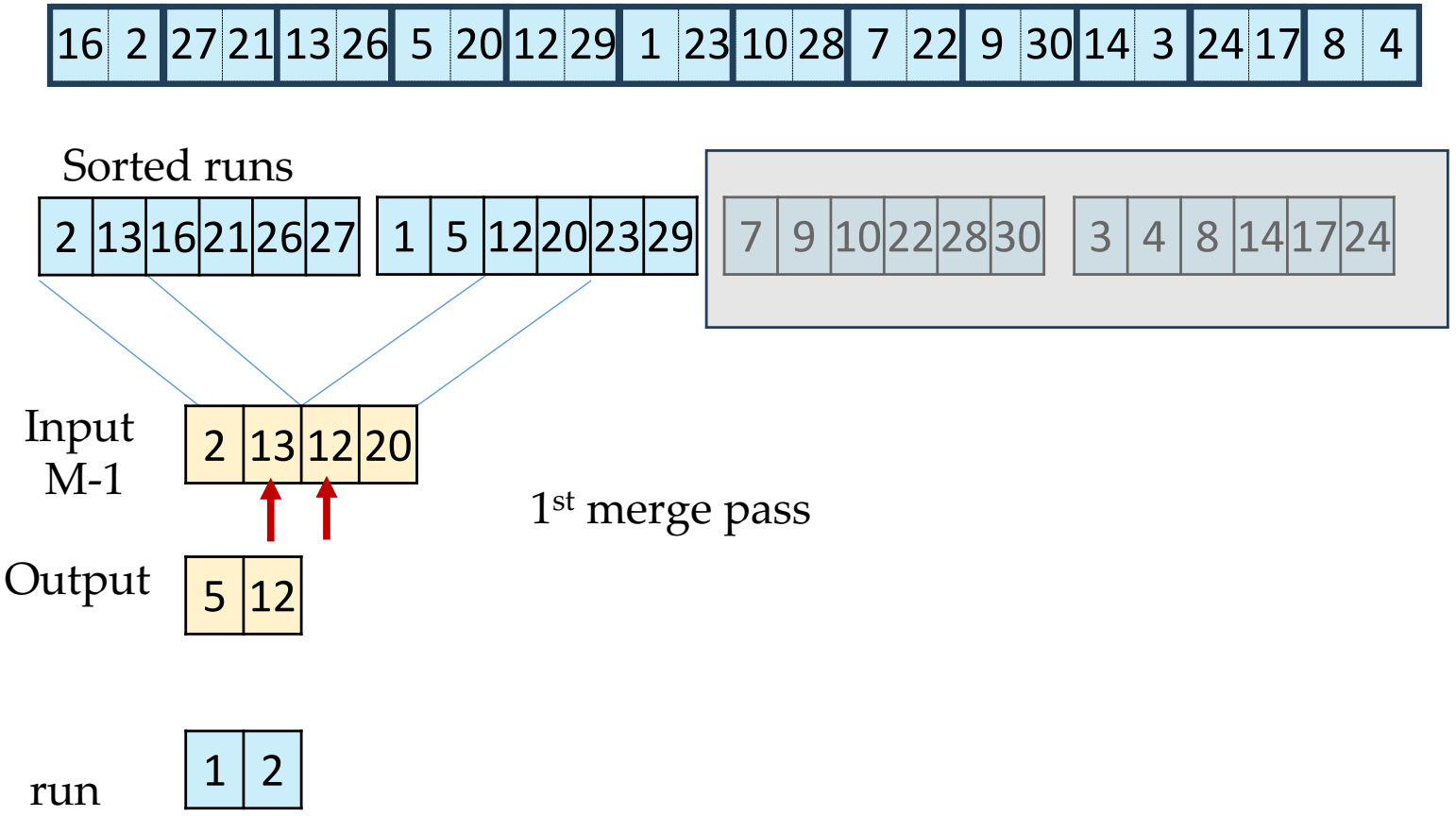
Example: Multi-pass External Sort-Merge

- Suppose each page can hold two records
- Buffer Size M = 3 (2 for input and 1 for output)



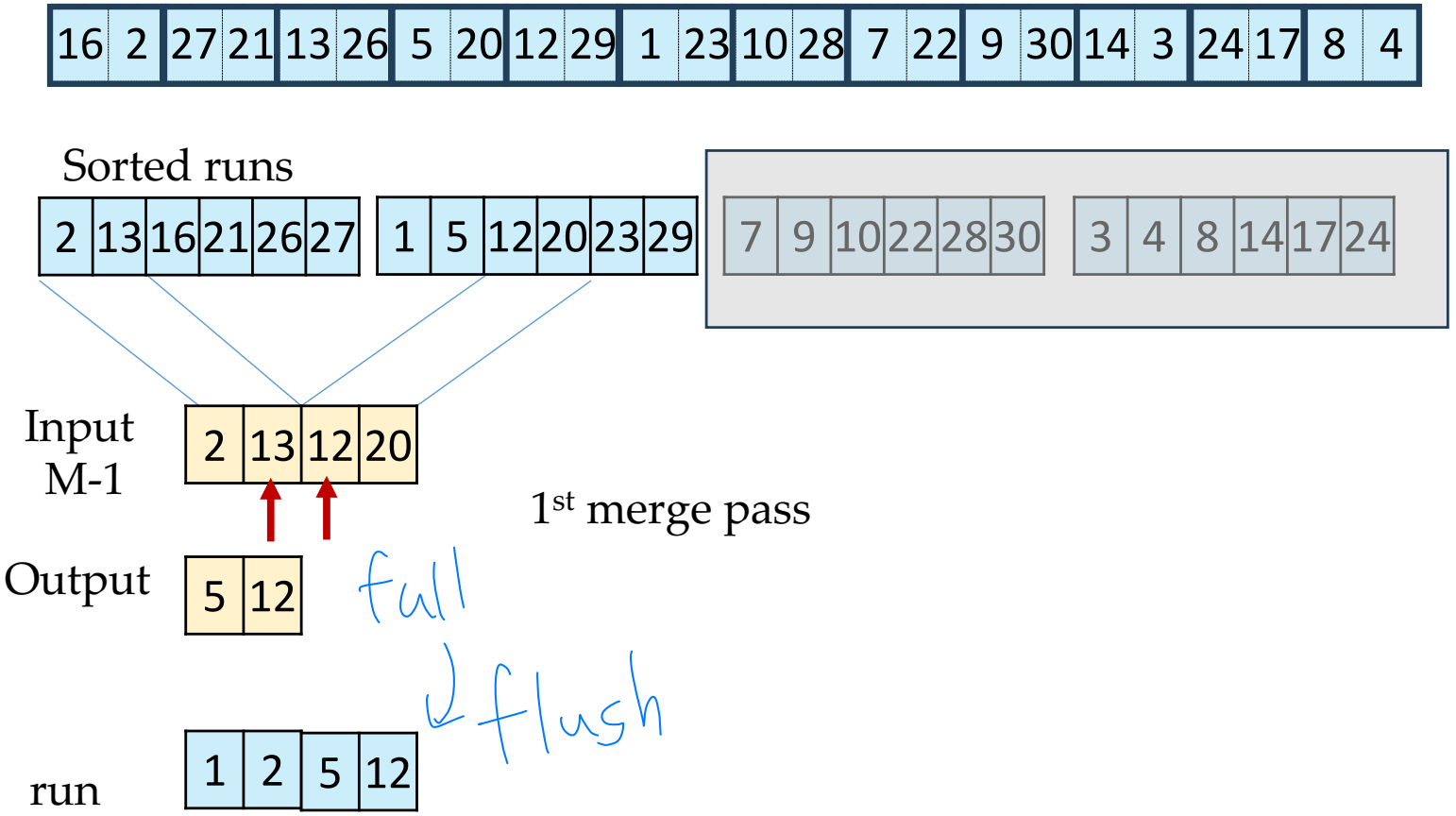
Example: Multi-pass External Sort-Merge

- Suppose each page can hold two records
- Buffer Size M = 3 (2 for input and 1 for output)



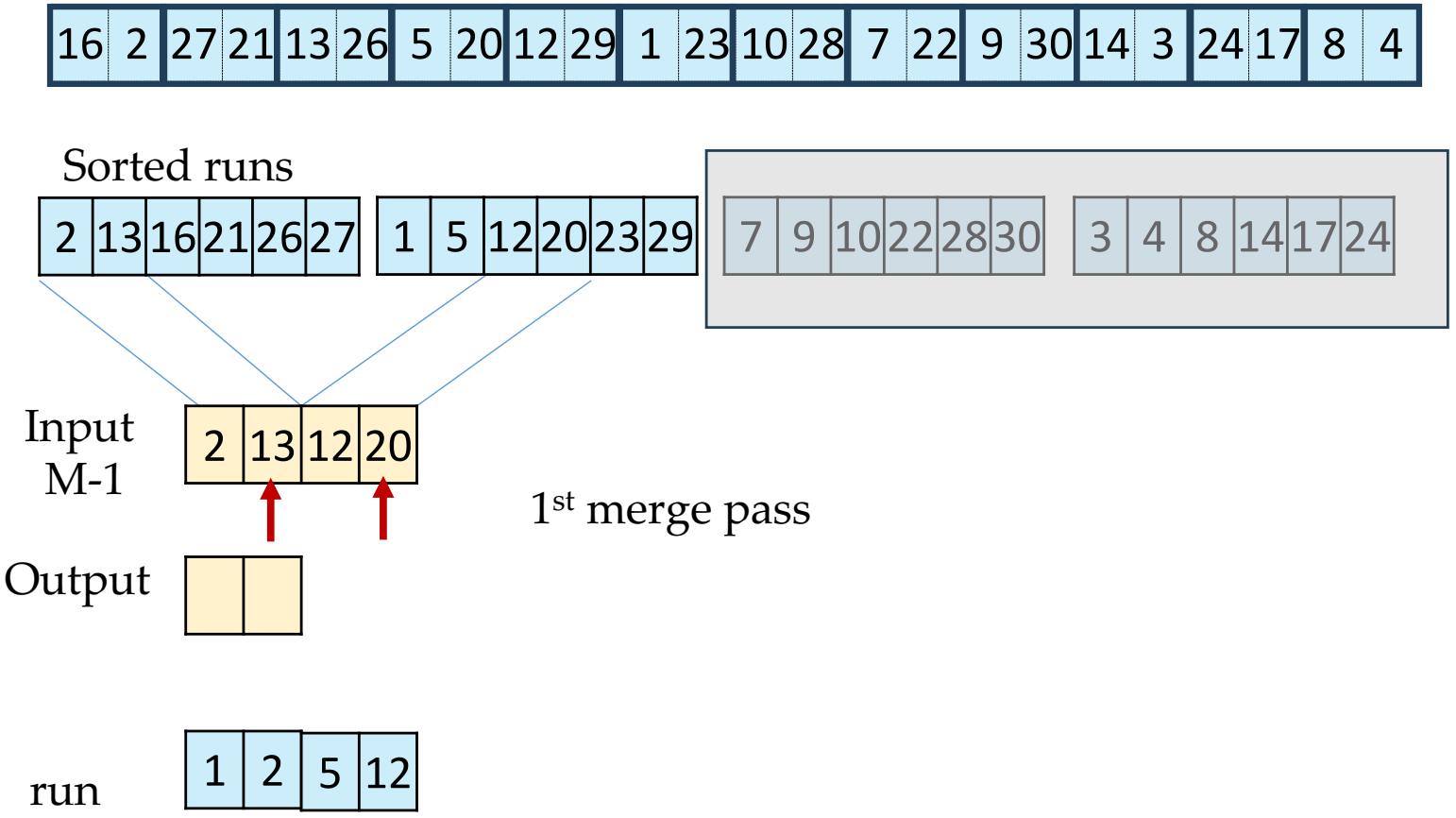
Example: Multi-pass External Sort-Merge

- Suppose each page can hold two records
- Buffer Size M = 3 (2 for input and 1 for output)



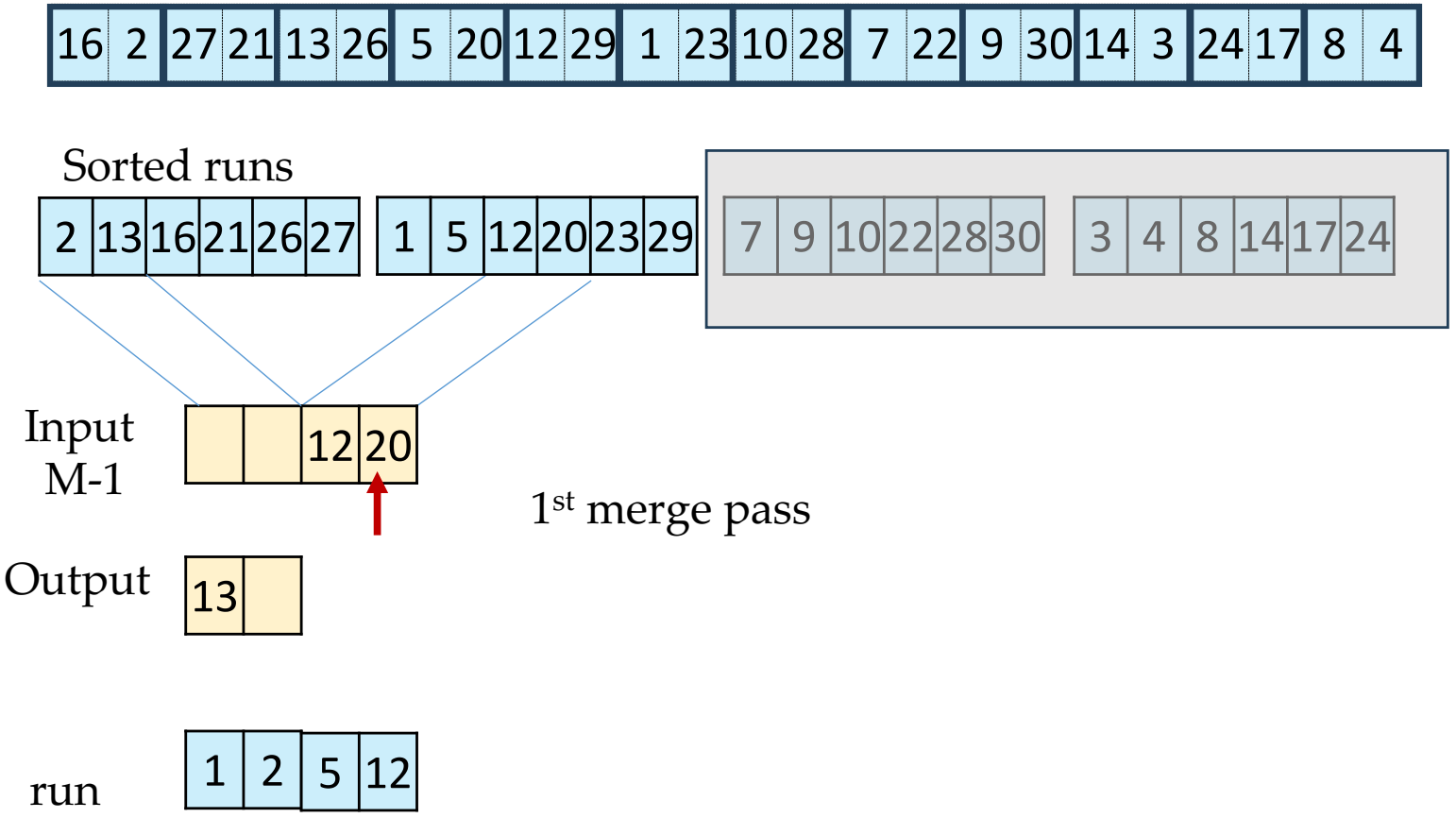
Example: Multi-pass External Sort-Merge

- Suppose each page can hold two records
- Buffer Size M = 3 (2 for input and 1 for output)



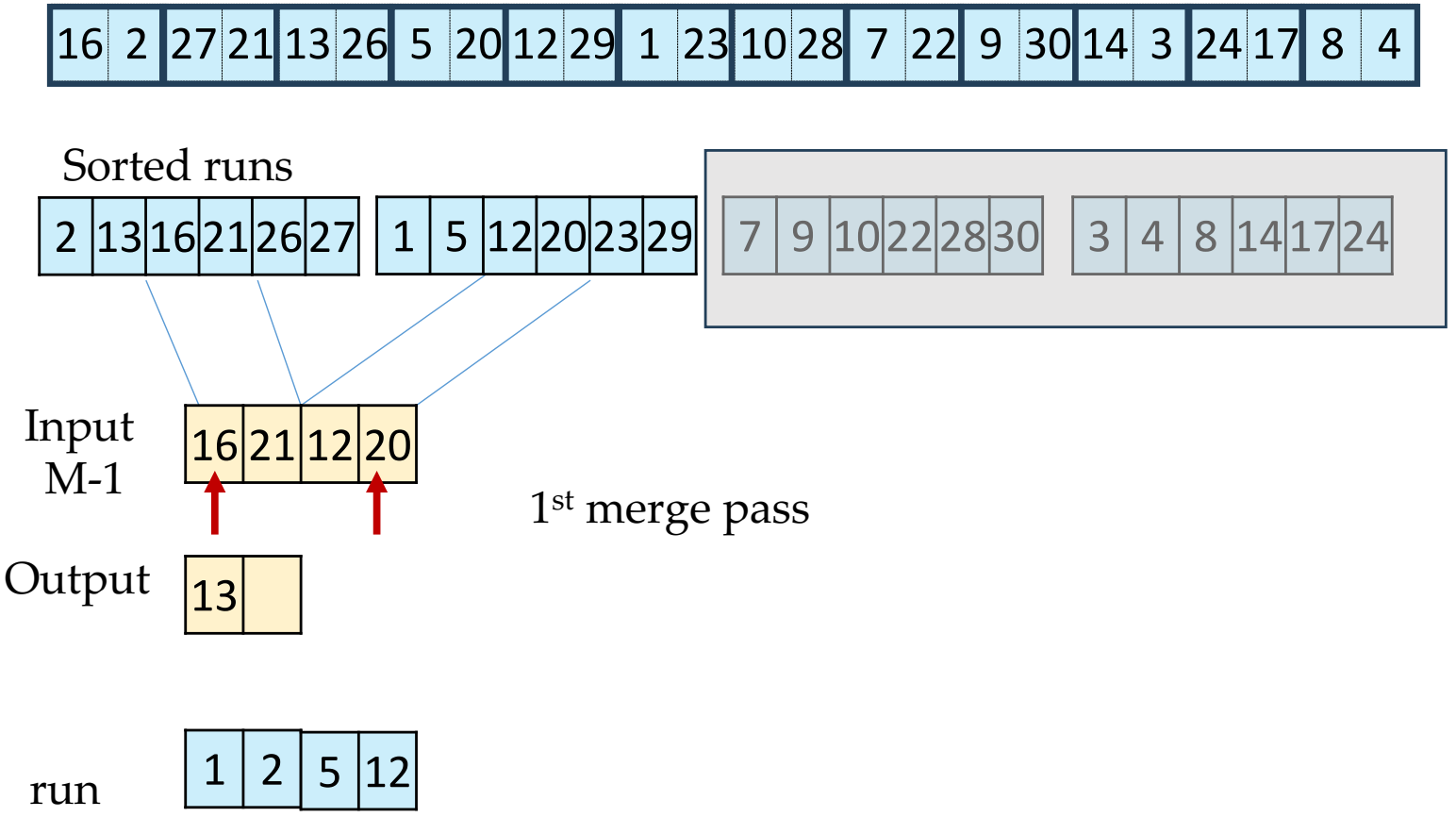
Example: Multi-pass External Sort-Merge

- Suppose each page can hold two records
- Buffer Size M = 3 (2 for input and 1 for output)



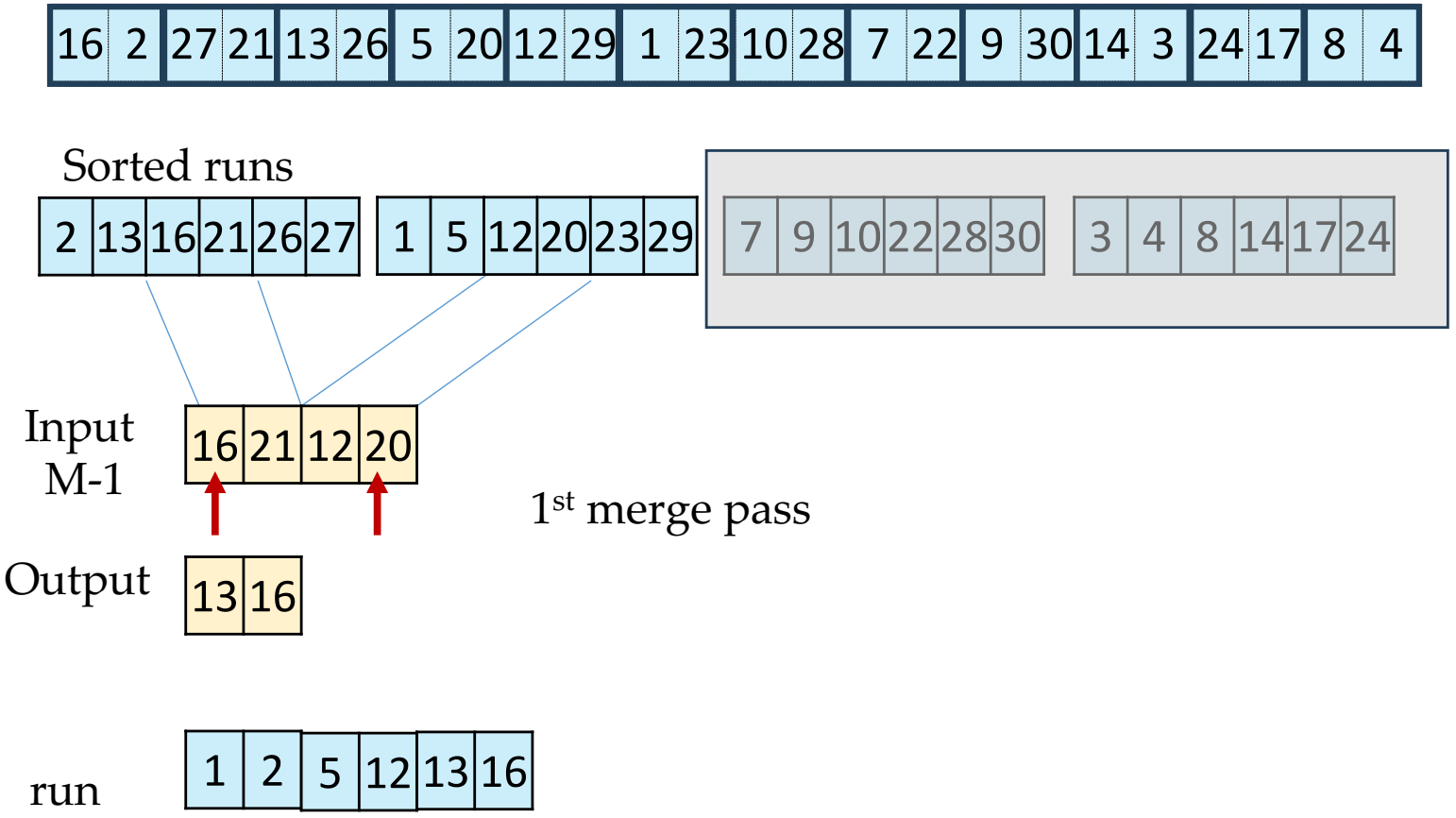
Example: Multi-pass External Sort-Merge

- Suppose each page can hold two records
- Buffer Size M = 3 (2 for input and 1 for output)



Example: Multi-pass External Sort-Merge

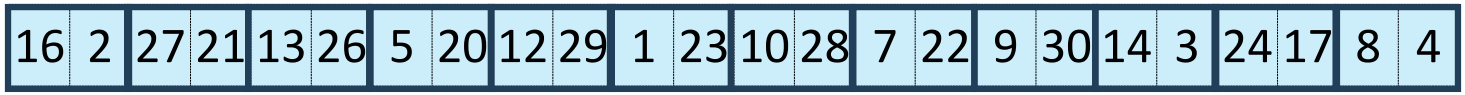
- Suppose each page can hold two records
- Buffer Size M = 3 (2 for input and 1 for output)

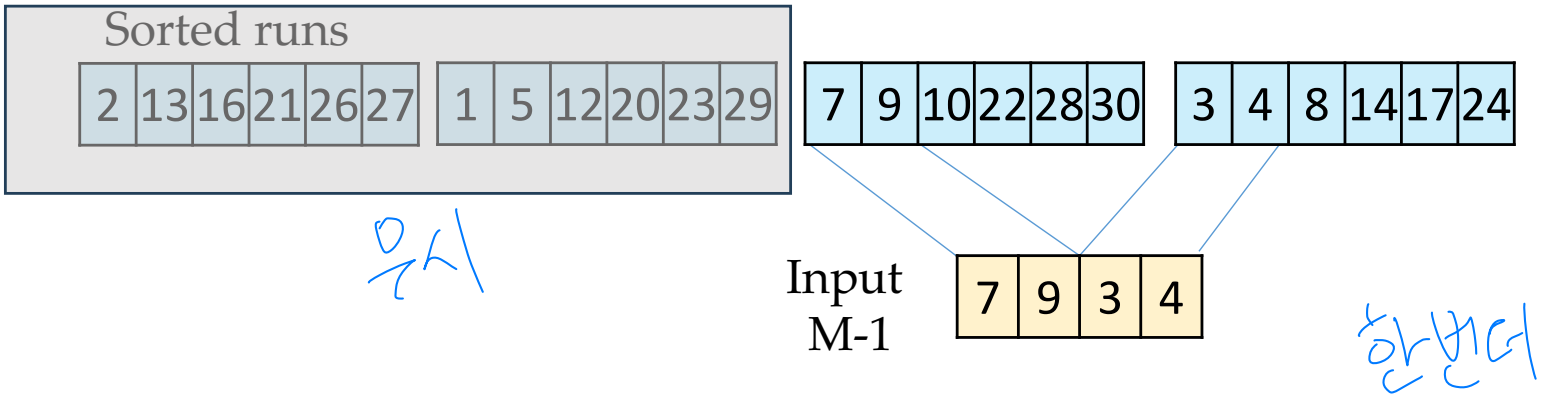


Example: Multi-pass External Sort-Merge

- Suppose each page can hold two records
- Buffer Size M = 3 (2 for input and 1 for output)

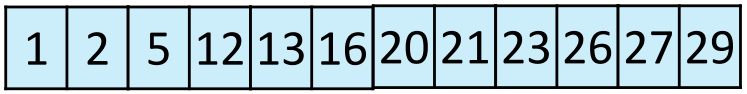
Initial input





Do The Same for the next 2 runs

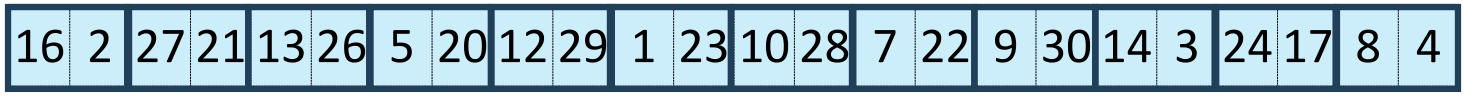
run

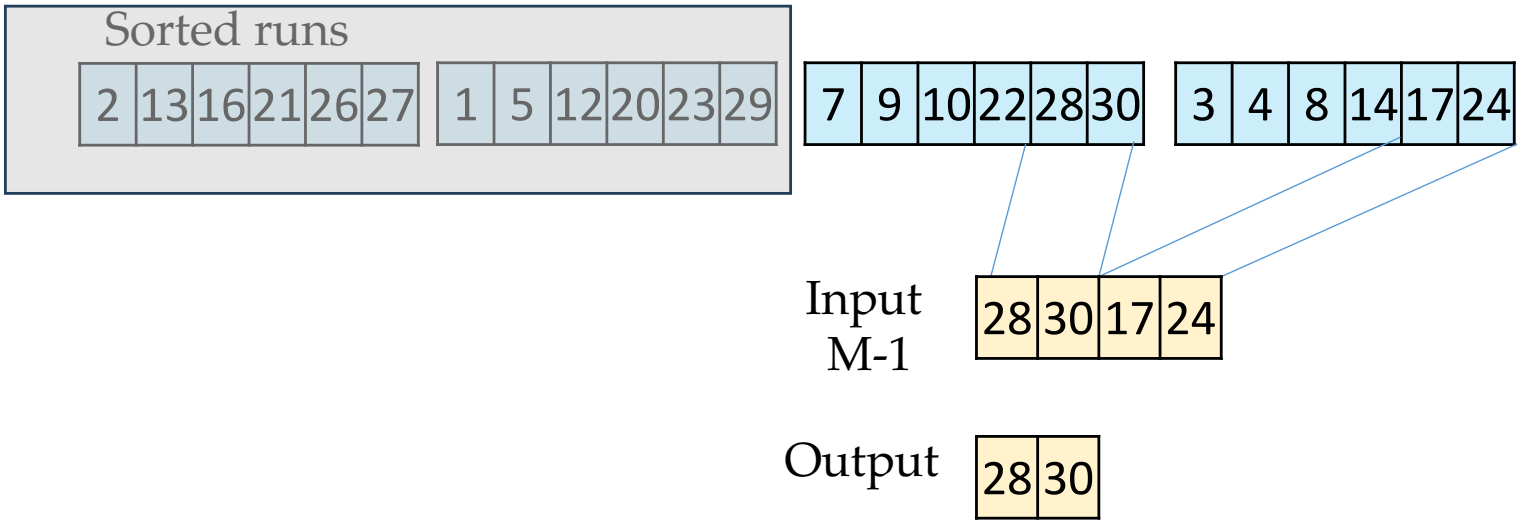


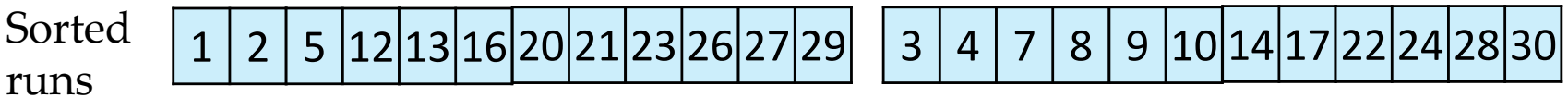
Example: Multi-pass External Sort-Merge

- Suppose each page can hold two records
- Buffer Size M = 3 (2 for input and 1 for output)

Initial input



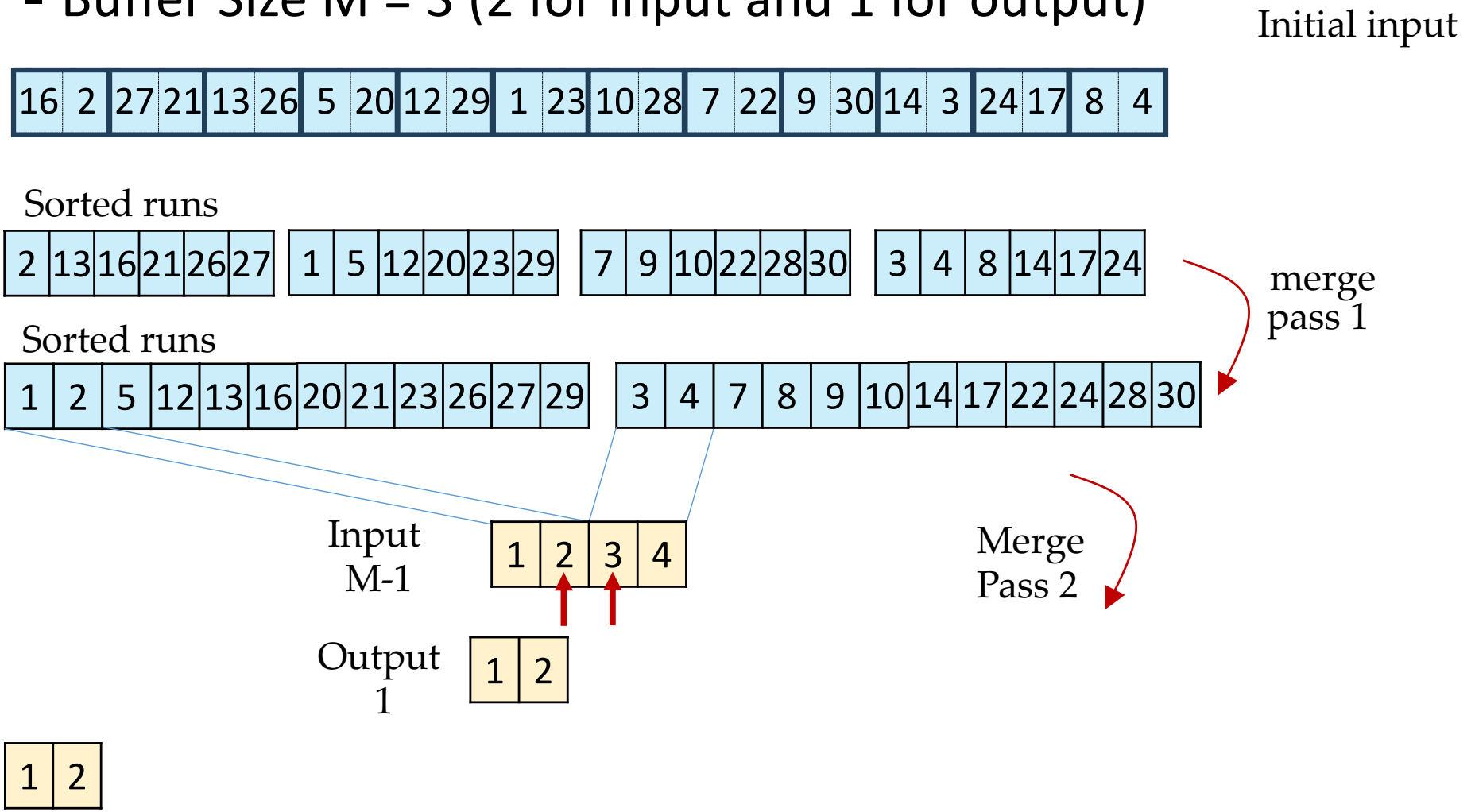




Number of Block Transfer:
$2*b_r + 2*b_r$

Example: Multi-pass External Sort-Merge

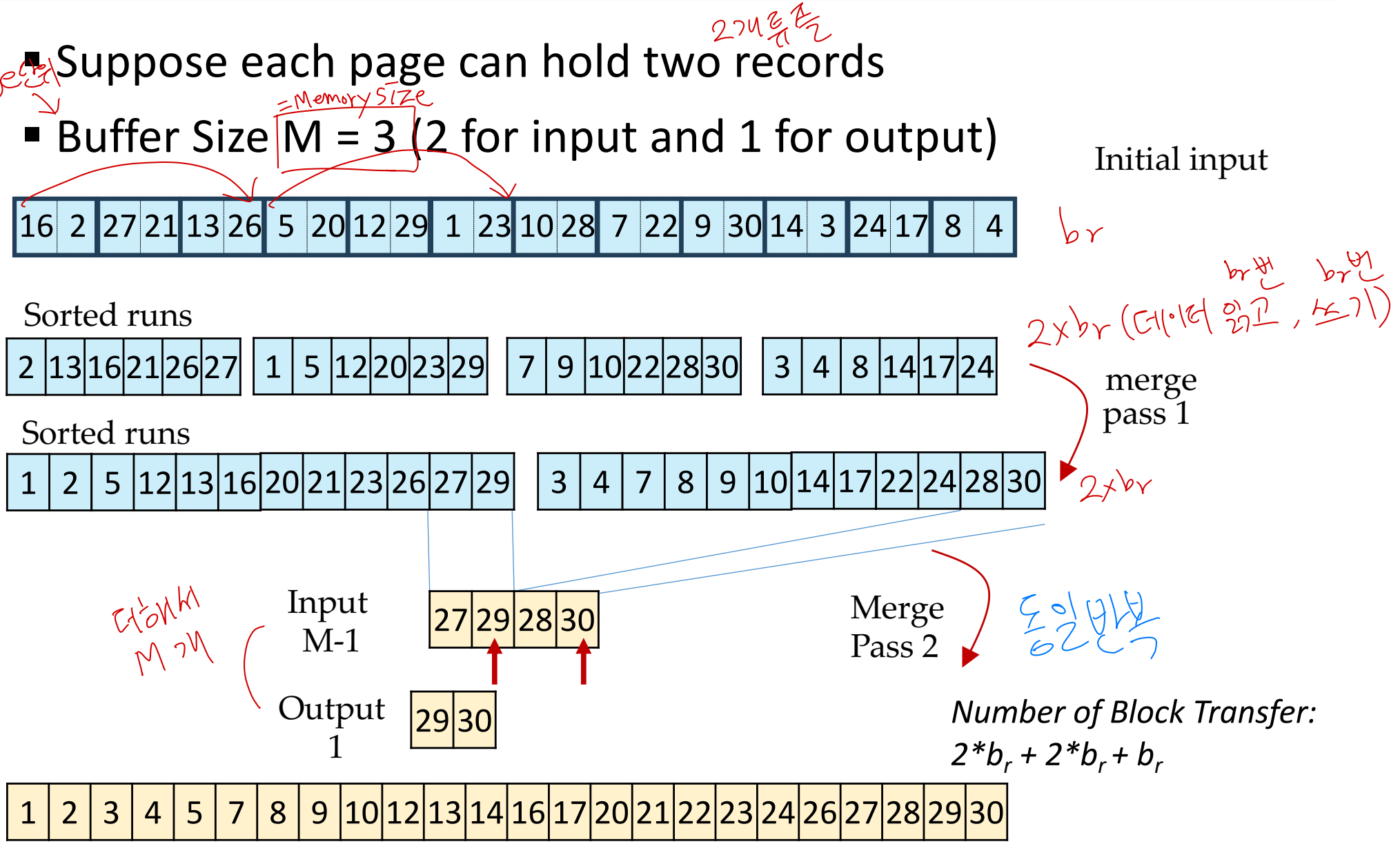
- Suppose each page can hold two records
- Buffer Size M = 3 (2 for input and 1 for output)



Output table is not stored on disks but consumed by the next level operator

Example: Multi-pass External Sort-Merge

- Suppose each page can hold two records
- Buffer Size M = 3 (2 for input and 1 for output)



Number of Block Transfer:
$2*b_r + 2*b_r + b_r$

Output table is not stored on disks but consumed by the next level operator

External Merge Sort – Cost analysis

- b_r : the number of blocks containing records of relation r.
- The initial run creation needs $2b_r$ block transfer
- The initial number of runs is $\lceil b_r/M \rceil$.
- The number of runs decreases by a factor of M-1.
  - Total number of merge passes required: $\lceil \log_{M-1}(b_r/M) \rceil$.
- The number of block transfers for each pass is $2b_r$.
  - for final pass, we don't count write cost since the output is sent to the parent operator without being written to disk
- Thus, total number of block transfers for external sorting:

$$b_r (2 \lceil \log_{M-1} (b_r / M) \rceil + 1)$$

External Merge Sort – Cost analysis

- M-way merge, i.e., 1 block per run leads to too many seeks during merge
  - Instead use b_b buffer blocks per run
    - ➔ read/write b_b blocks at a time
  - ➔ Replace M in the previous equation with $\lfloor M/b_b \rfloor$

- Cost of seeks
  - During run generation: one seek to read each run and one seek to write each run
    - $2\lceil b_r / M \rceil$
  - During the merge phase
    - Need $2\lceil b_r / b_b \rceil$ seeks for each merge pass
      - except the final one which does not require a write
    - Total number of seeks:

$$2\lceil b_r / M \rceil + \lceil b_r / b_b \rceil (2\lceil \log_{\lfloor M/bb \rfloor - 1}(b_r / M) \rceil - 1)$$

Join Operation

- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Choice based on cost estimate
- Examples use the following information
  - Number of records of *student*: 5,000
  - Number of records of *takes*: 10,000
  - Number of blocks of *student*: 100
  - Number of blocks of *takes*: 400

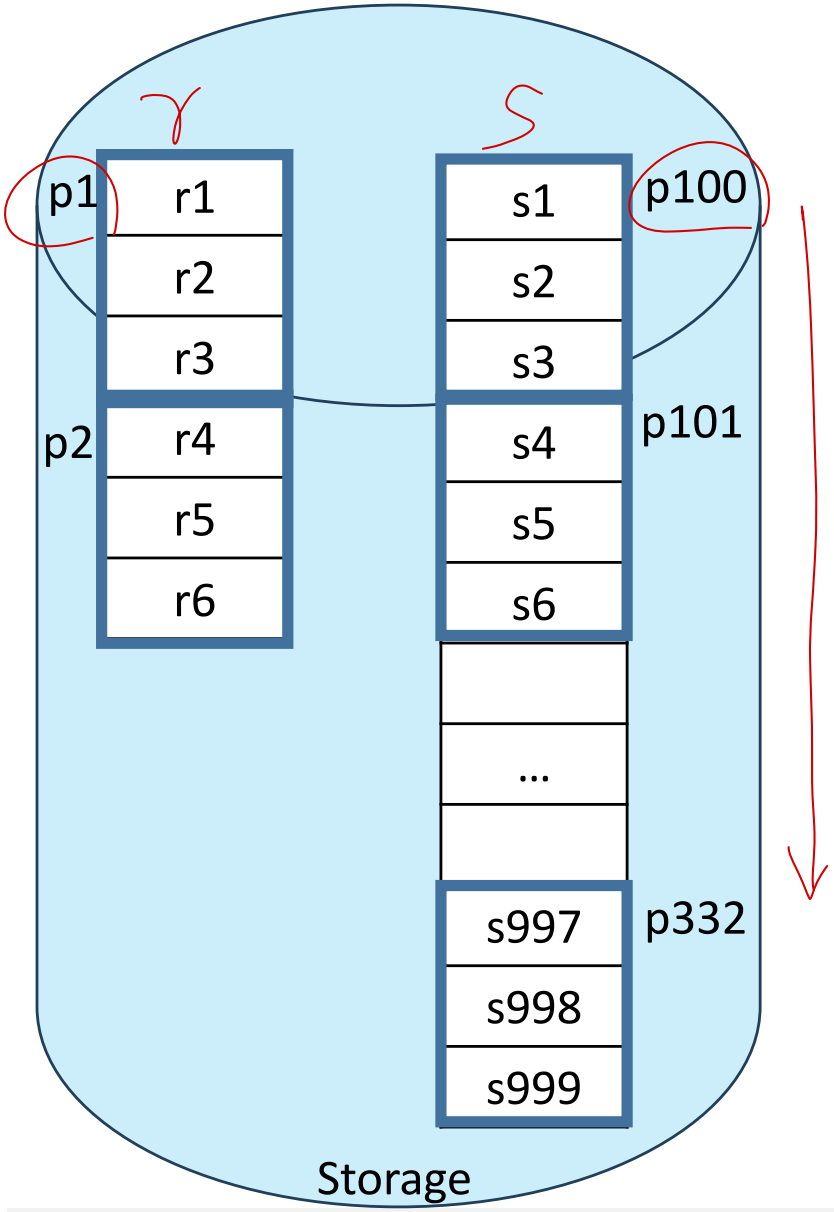
Nested-Loop Join

- To compute the theta join $r \bowtie_\theta s$

```
for each tuple t_r in r do begin
    for each tuple t_s in s do begin
        test pair (t_r, t_s) to see if they satisfy the join condition θ
        if they do, add t_r • t_s to the result.
    end
end
```

- r is called the **outer relation** and s the **inner relation** of the join.
- Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples in the two relations.

Nested-Loop Join

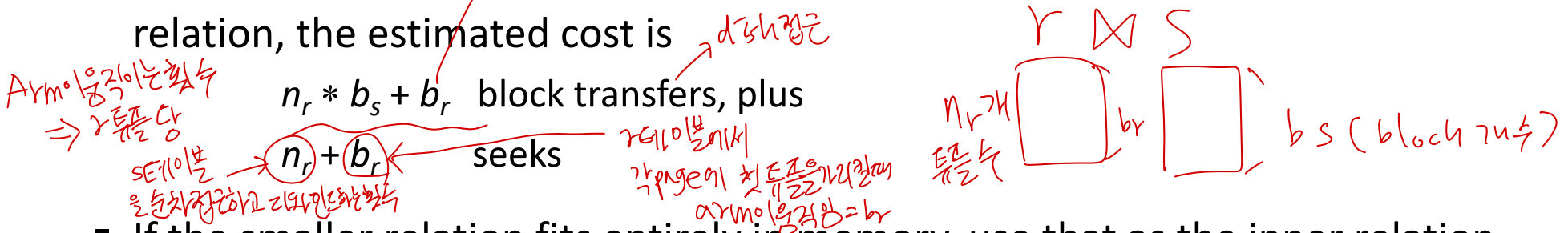


r1, s1 → read p1, read p100
r1, s2
r1, s3
r1, s4 → read p101
r1, s5
r1,s6
...
r1, s997 → read p332
r1, s998
r1, s999
r2, s1. → read p1, read p100.
r2, s2 // p1 could've been evicted from cache
r2, s3
r2, s4 → read p101
r2, s5
r2,s6
...
r2, s997 → read p332
r2, s998
r2, s999
r3, s1 → read p1, read p100
r3, s2

Main mem Capt!
cache hit
new page
new page
r1 done
not require
⇒ 탐색시스템의 분석이 $n_r \times b_s$ 2배!
r 페이지를 읽은 횟수가 고려되지 않음
순차적 읽기
S를 접근 → 233 transfer
1 seek
Arm이 접근하는 횟수

Nested-Loop Join (Cont.)

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is $n_r * b_s + b_r$ block transfers, plus $n_r + b_r$ seeks



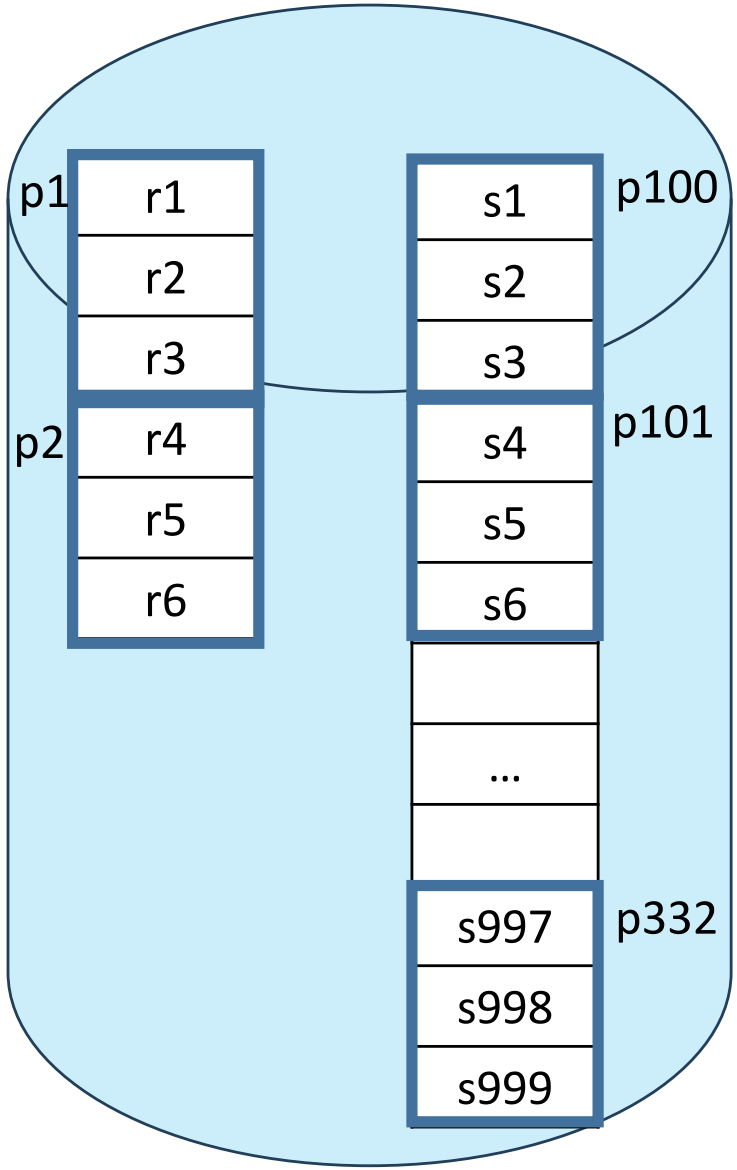
- If the smaller relation fits entirely in memory, use that as the inner relation.
  - Reduces cost to $b_r + b_s$ block transfers and 2 seeks
- Assuming worst case,
  - with *student* as outer relation:
    - 5000 * 400 + 100 = 2,000,100 block transfers,
    - 5000 + 100 = 5100 seeks
  - with *takes* as the outer relation
    - 10000 * 100 + 400 = 1,000,400 block transfers and 10,400 seeks
- If smaller relation (*student*) fits entirely in memory, the cost estimate will be 500 block transfers.

Block Nested-Loop Join

- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

```
for each block B_r of r do begin
    for each block B_s of s do begin
        for each tuple t_r in B_r do begin
            for each tuple t_s in B_s do begin
                Check if (t_r, t_s) satisfy the join condition
                if they do, add t_r • t_s to the result.
            end
        end
    end
end
```

Storage Access: Block Nested-Loop Join



Storage

r1, s1 → read p1& p100
r1, s2
r1, s3
r2, s1
r2, s2
r2, s3
r3, s1
r3, s2
r3, s3
r1, s4 → read p101
r1, s5
r1, s6
r2, s4
r2, s5
r2, s6
r3, s4
r3, s5
r3, s6
...

r2만큼 page 갯수 반복

r1, s997 → read p332
r1, s998
r1, s999
...
r3, s997
r3, s998
r3, s999
r4, s1 → read p2& p100
r4, s2
r4, s3
r5, s1
r5, s2
r5, s3
r6, s1
r6, s2
r6, s3
...

Block Nested-Loop Join (Cont.)

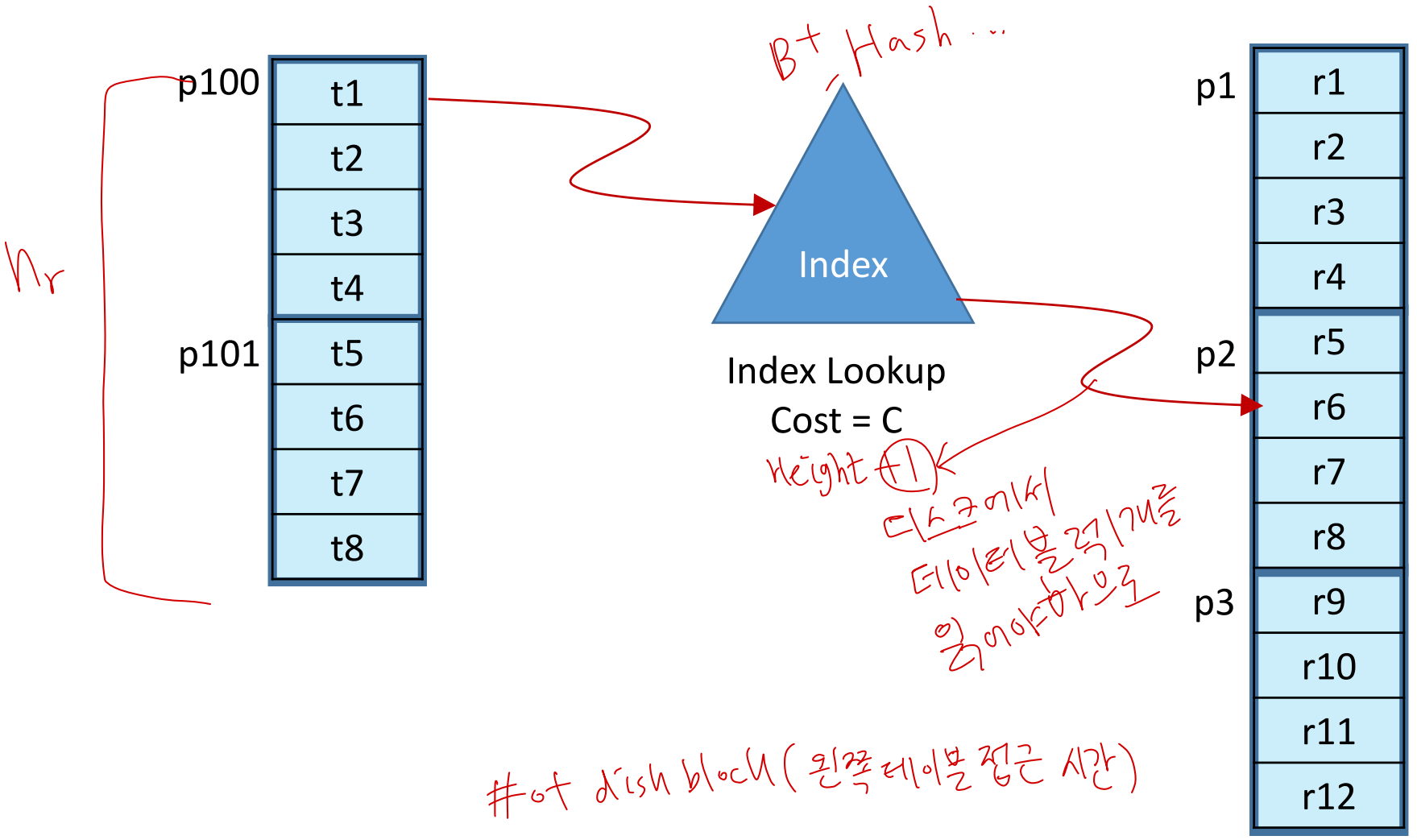
- Worst case estimate: $b_r * b_s + b_r$ block transfers + 2 * b_r seeks
  - Each block in the inner relation *s* is read once for each *block* in the outer relation
- Best case: $b_r + b_s$ block transfers + 2 seeks.

- Improvements to nested loop and block nested loop algorithms:
  - In block nested-loop, use *M* - 2 disk blocks as blocking unit for outer relations, where *M* = memory size in blocks; use remaining two blocks to buffer inner relation and output
    - reduces the number of scans of inner relation from b_r to $\lceil b_r / (M-2) \rceil$
    - Cost = $\lceil b_r / (M-2) \rceil * b_s + b_r$ block transfers + $2 \lceil b_r / (M-2) \rceil$ seeks

Indexed Nested-Loop Join

- Index lookups can replace file scans if 둘중 하나의 인덱스 갖고있을때
  - join is an equi-join or natural join and
  - an index is available on the inner relation's join attribute
    - Can construct an index just to compute a join.
- For each tuple t_r in the outer relation *r,* use the index to look up tuples in *s* that satisfy the join condition with tuple t_r.
- Worst case: buffer has space for only one page of *r*, and, for each tuple in *r*, we perform an index lookup on *s.*
- Cost of the join: b_r (*transfer_time* + *seek_time*) + n_r * *c*
  - Where *c* is the cost of traversing index and fetching all matching *s* tuples for one tuple or *r*
- If indices are available on join attributes of both *r* and *s,* use the relation with fewer tuples as the outer relation.

Indexed Nested-Loop Join



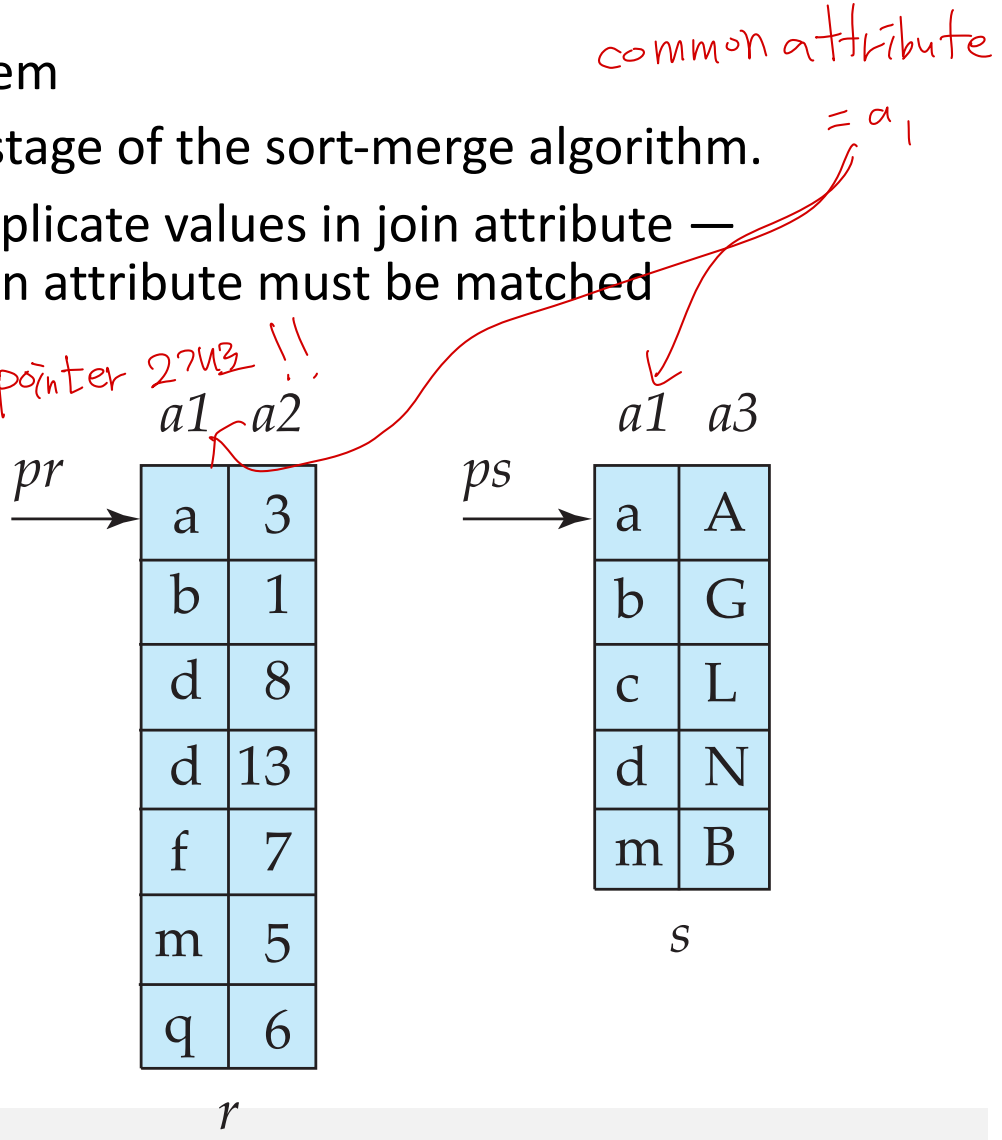
Join Cost : $b_r\,(transfer\_time + seek\_time) + n_r * c$

Example of Nested-Loop Join Costs

- Compute *student* ⋈ *takes,* with *student* as the outer relation.
- Let *takes* have a primary B$^+$-tree index on the attribute *ID,* which contains 20 entries in each index node.
- Since *takes* has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data
- *student* has 5000 tuples
- Cost of block nested loops join
  - 400*100 + 100 = 40,100 block transfers + 2 * 100 = 200 seeks
    - assuming worst case memory
    - may be significantly less with more memory
- Cost of indexed nested loops join
  - 100 + 5000 * 5 = 25,100 block transfers and seeks.
  - # of seeks increased

Sort-Merge-Join

1. Sort both relations on their join attribute (if not already sorted on the join attributes).
2. Merge the sorted relations to join them
   1. Join step is similar to the merge stage of the sort-merge algorithm.
   2. Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched
   3. Detailed algorithm in book



Sort-Merge-Join (Cont.)

- Can be used for equi-joins and natural joins
- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory)
- Thus the cost of merge join is:

  $b_r + b_s$ block transfers → 각각 테이블 전부를 1번 스캔하면 Join 종료

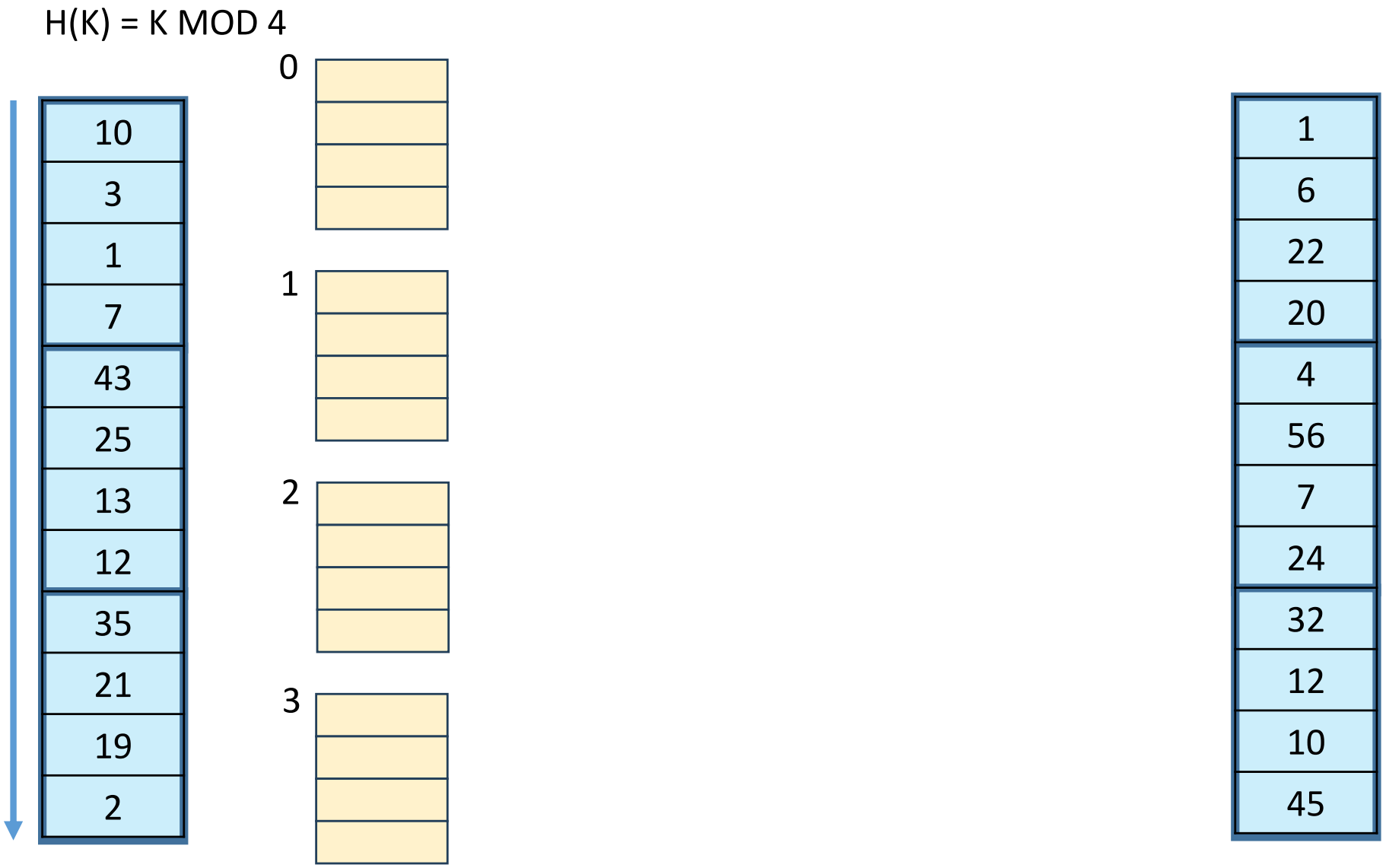
  - + the cost of sorting if relations are unsorted.

Hash-Join

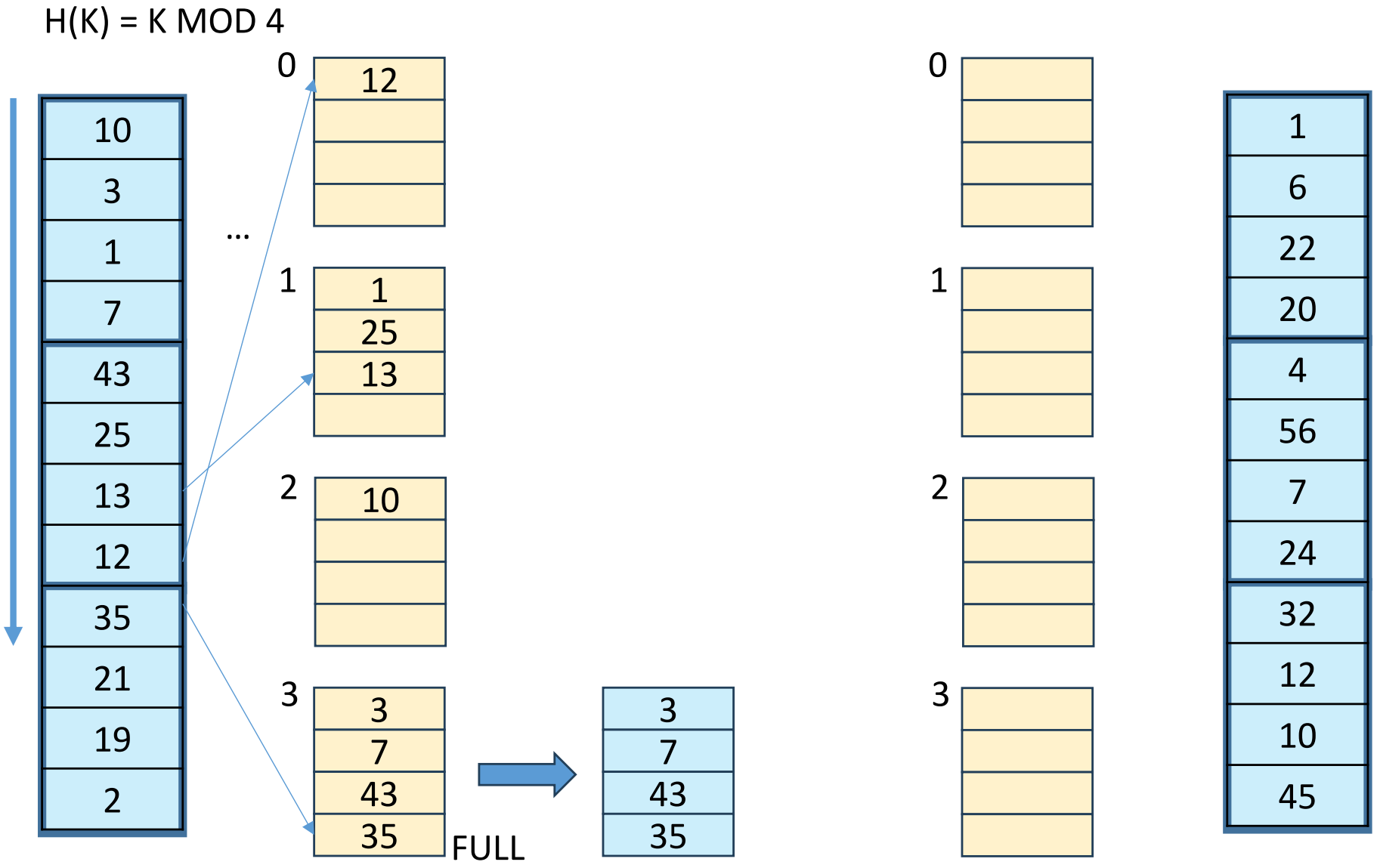
↳ range query X point query O

- Applicable for equi-joins and natural joins.
- A hash function *h* is used to partition tuples of both relations
- *h* maps *JoinAttrs* values to {0, 1, ..., *n*}, where *JoinAttrs* denotes the common attributes of *r* and *s* used in the natural join.
  - $r_0, r_1, \ldots, r_n$ denote partitions of *r* tuples
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[JoinAttrs])$.
  - $s_0, s_1 \ldots, s_n$ denotes partitions of *s* tuples
    - Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s[JoinAttrs])$.

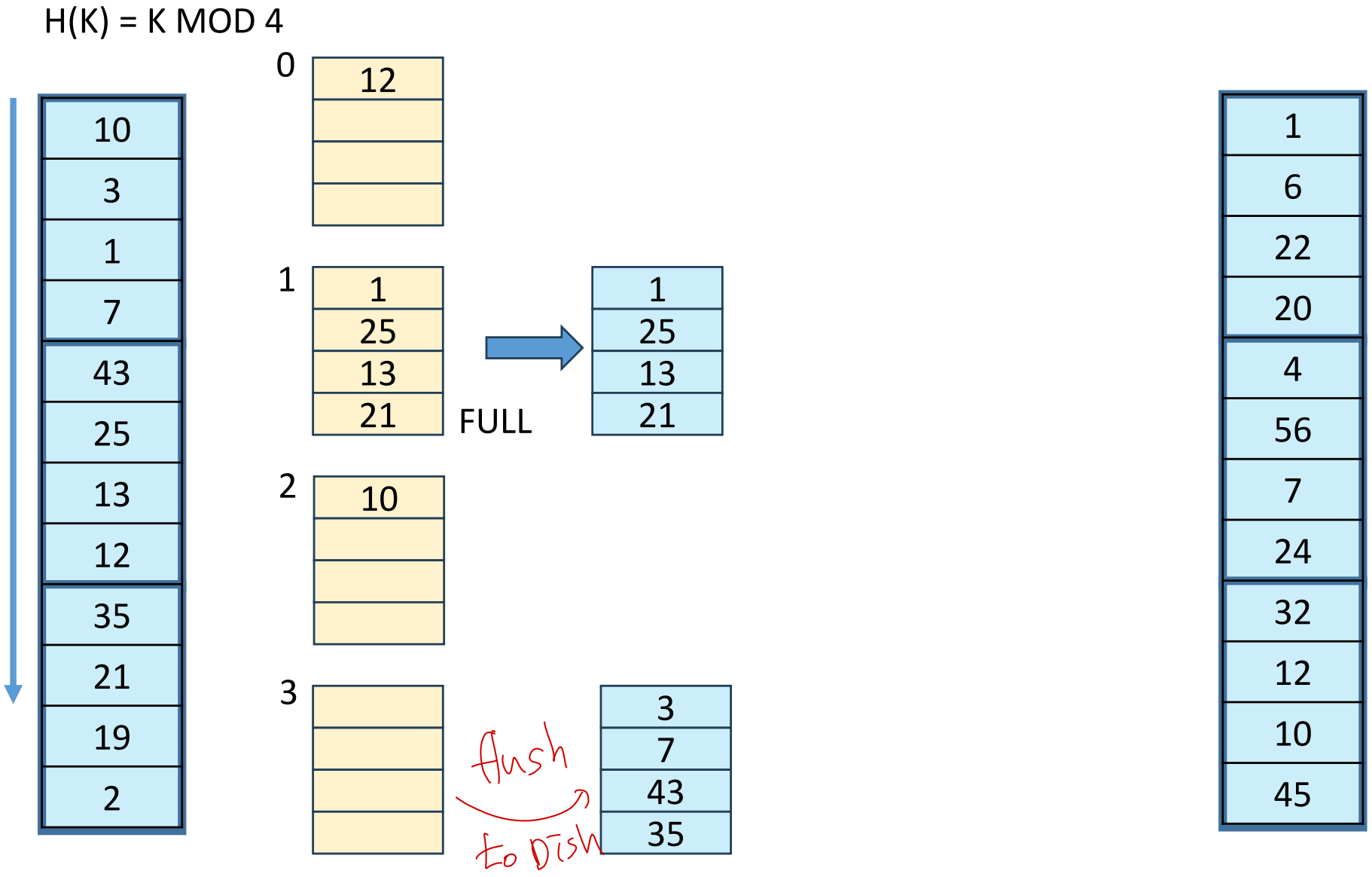
Hash-Join Example



Hash-Join Example

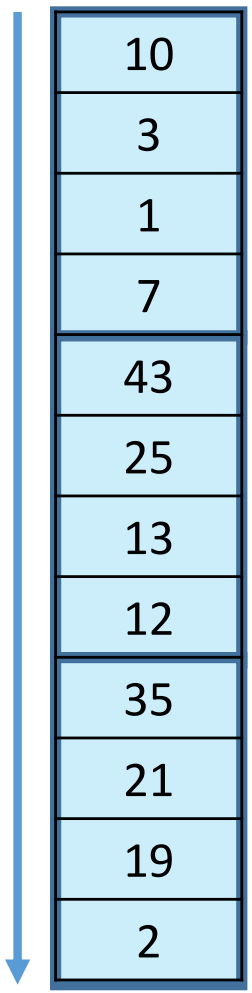


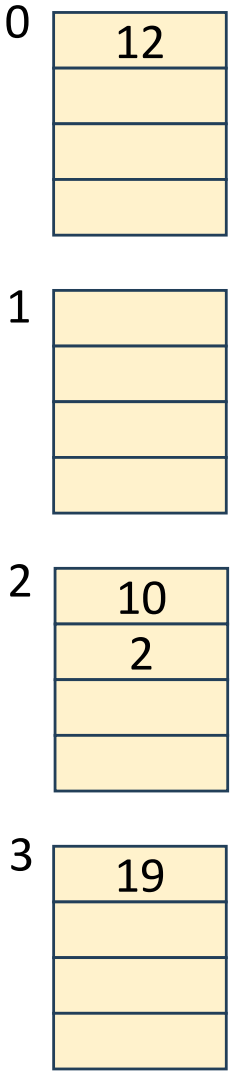
Hash-Join Example

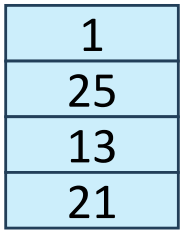


Hash-Join Example

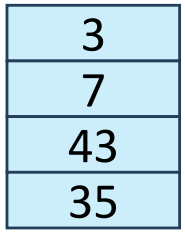
H(K) = K MOD 4

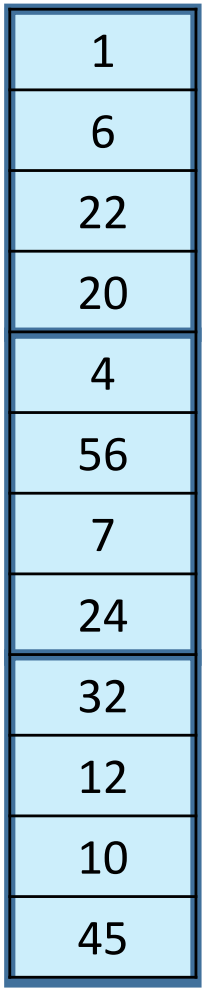






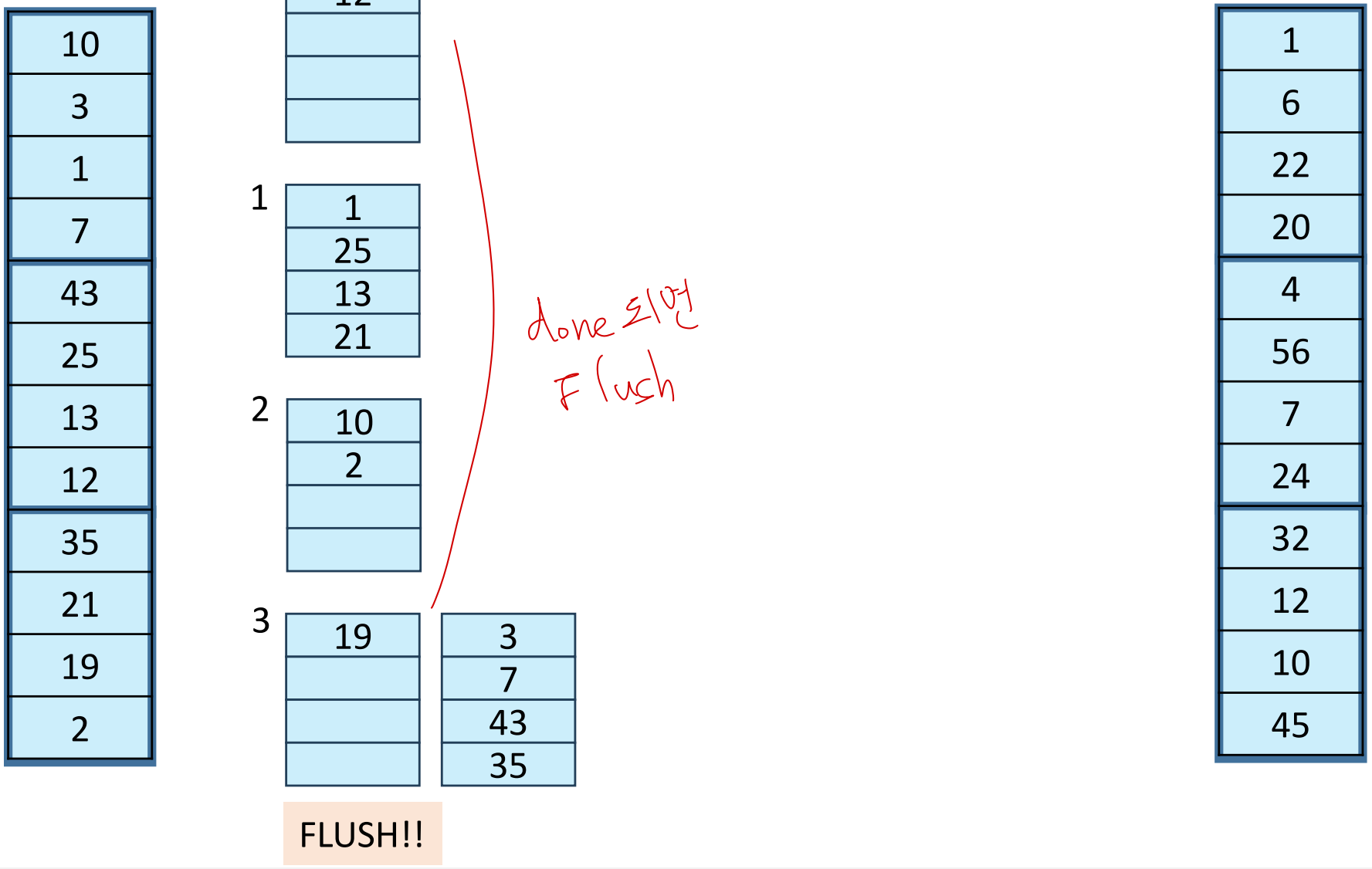
done!





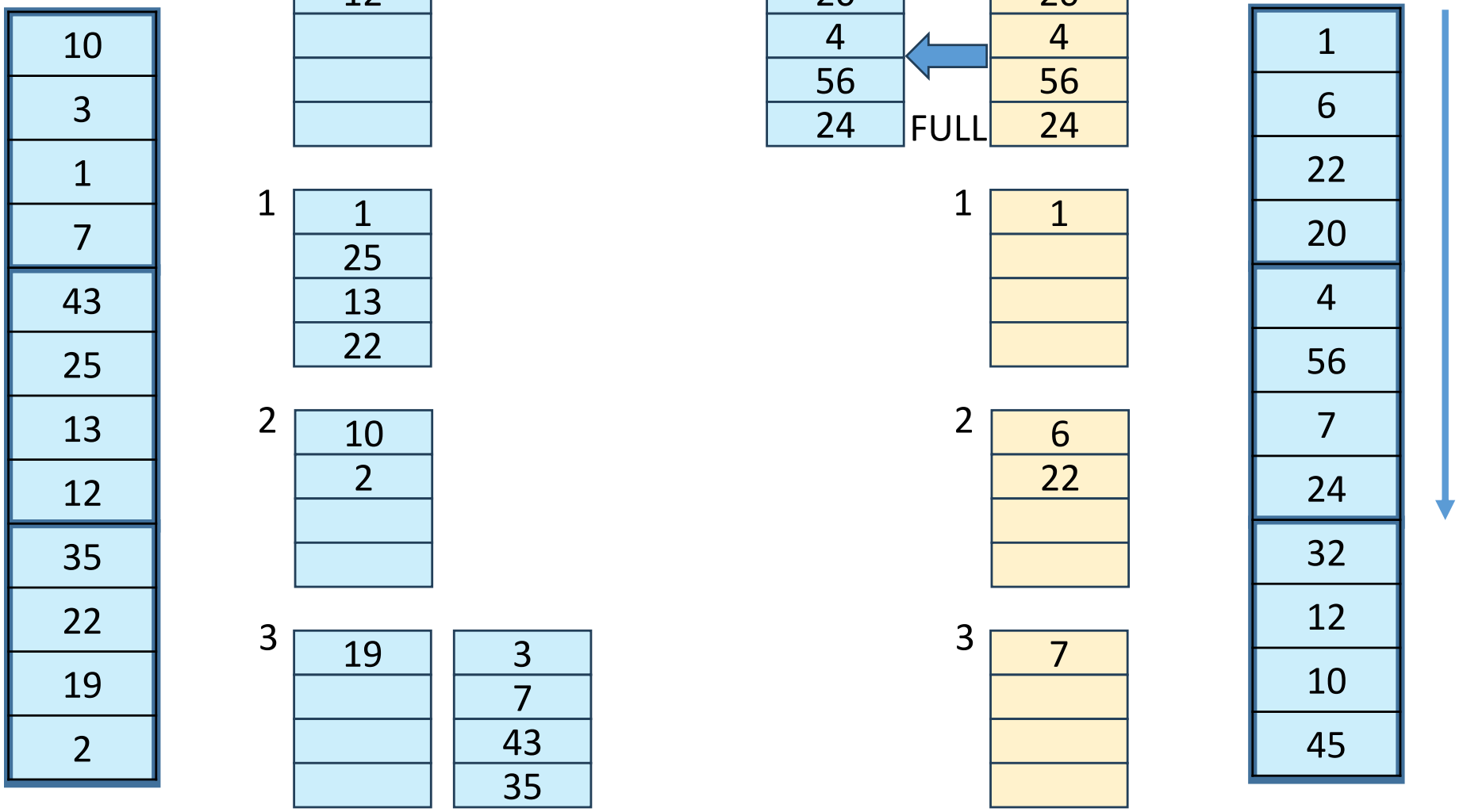
Hash-Join Example

H(K) = K MOD 4



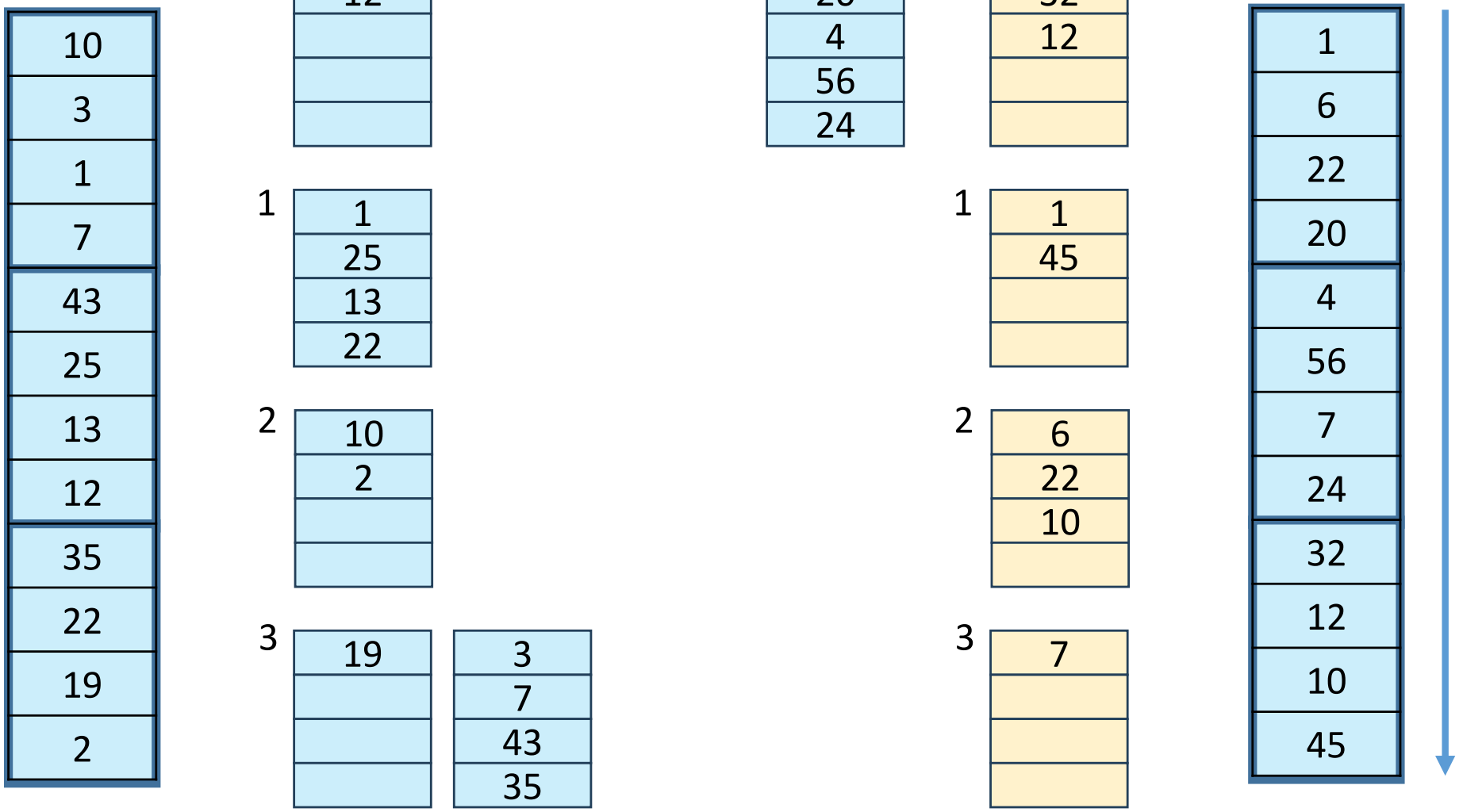
Hash-Join Example

H(K) = K MOD 4

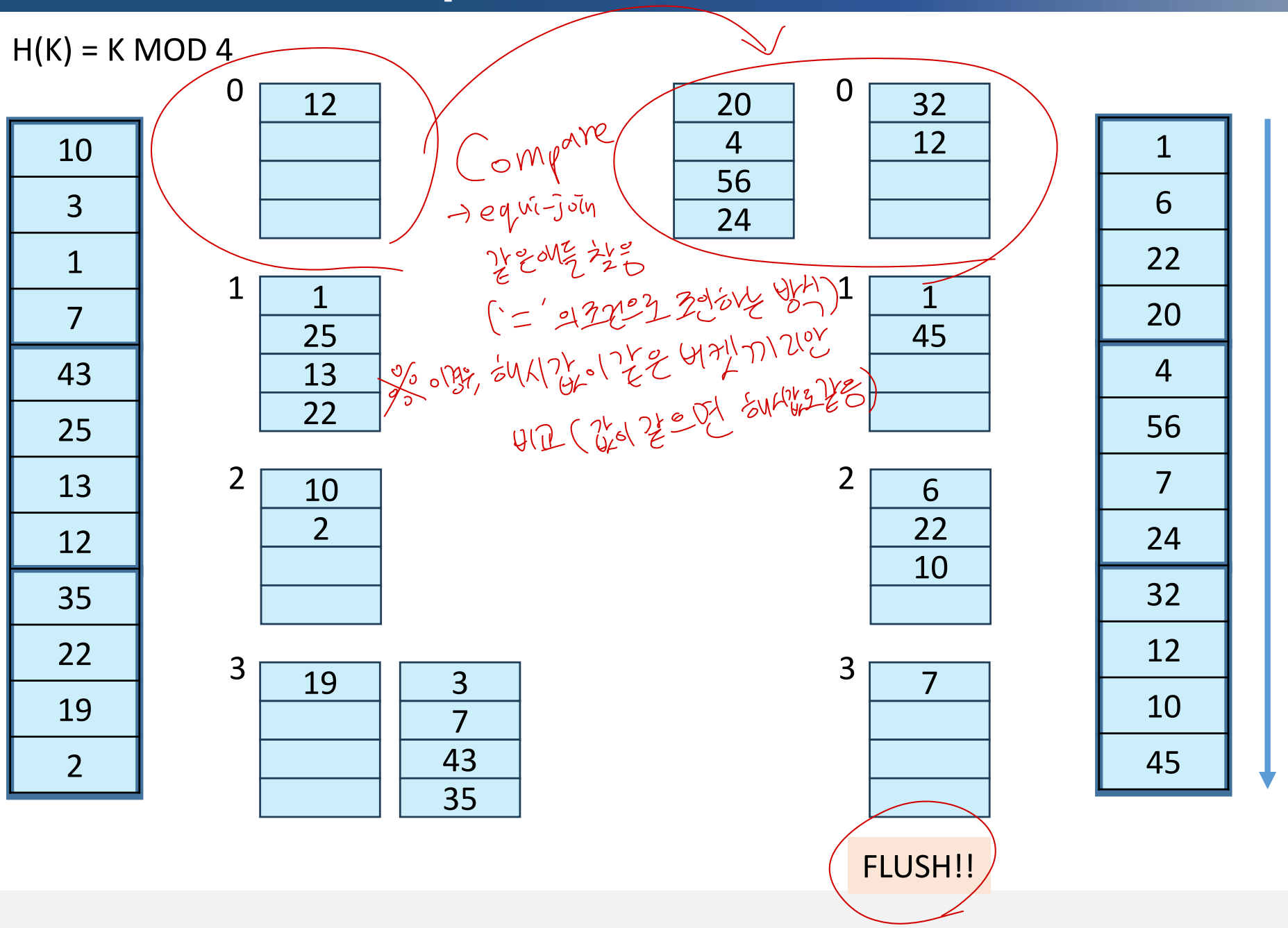


Hash-Join Example

H(K) = K MOD 4



Hash-Join Example



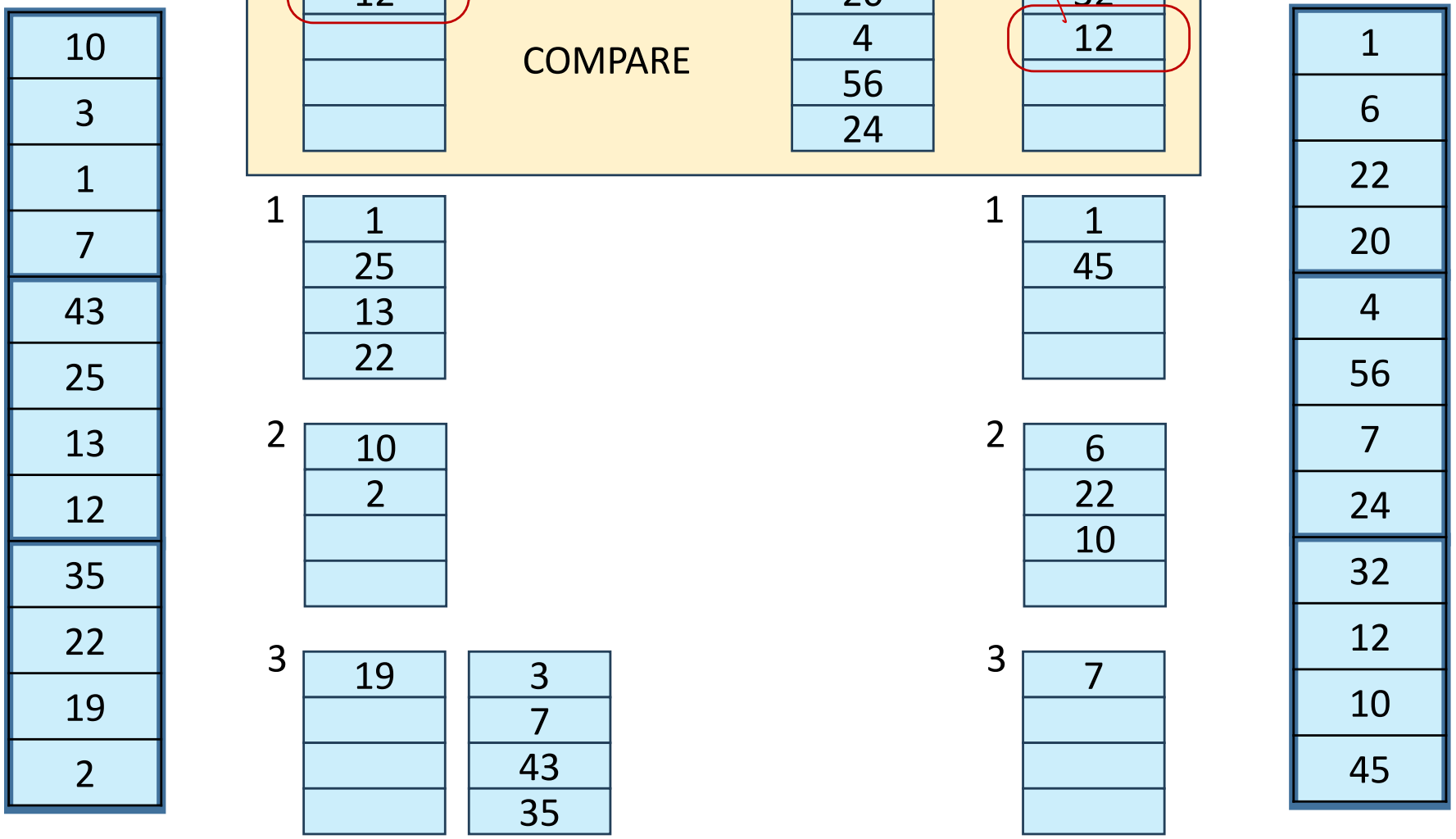
Comparison with Hash-Join

- *r* tuples in r_i need only to be compared with *s* tuples in s_i
- Need not be compared with *s* tuples in any other partition, since:
  - an *r* tuple and an *s* tuple that satisfy the join condition will have the same value for the join attributes.
  - If that value is hashed to some value *i*, the *r* tuple has to be in r_i and the *s* tuple in s_i.

Hash-Join Example

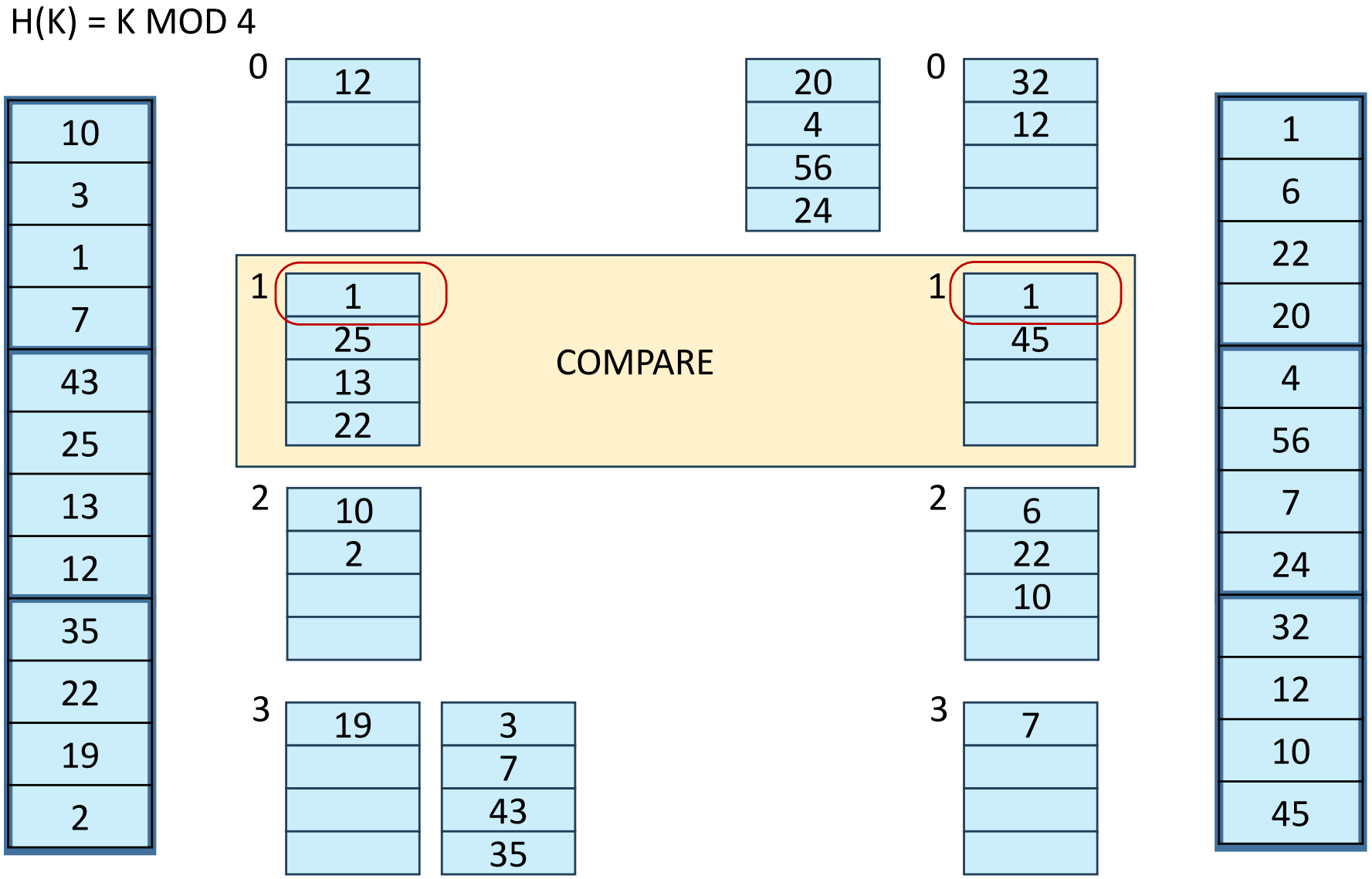
H(K) = K MOD 4

같은 값이 있으면 output 테이블에 저장



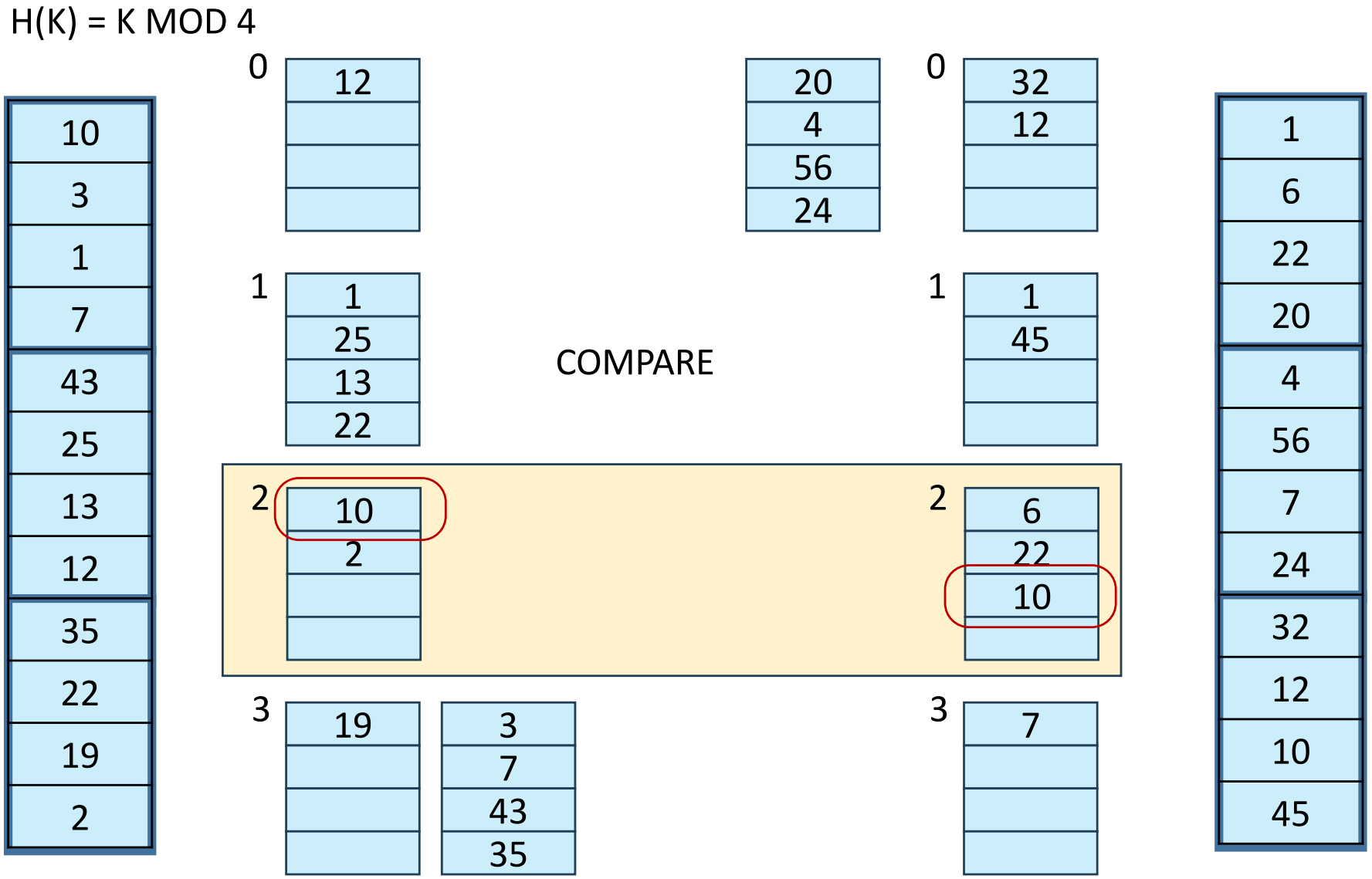
Tuples need only to be compared with tuples in the same bucket

Hash-Join Example



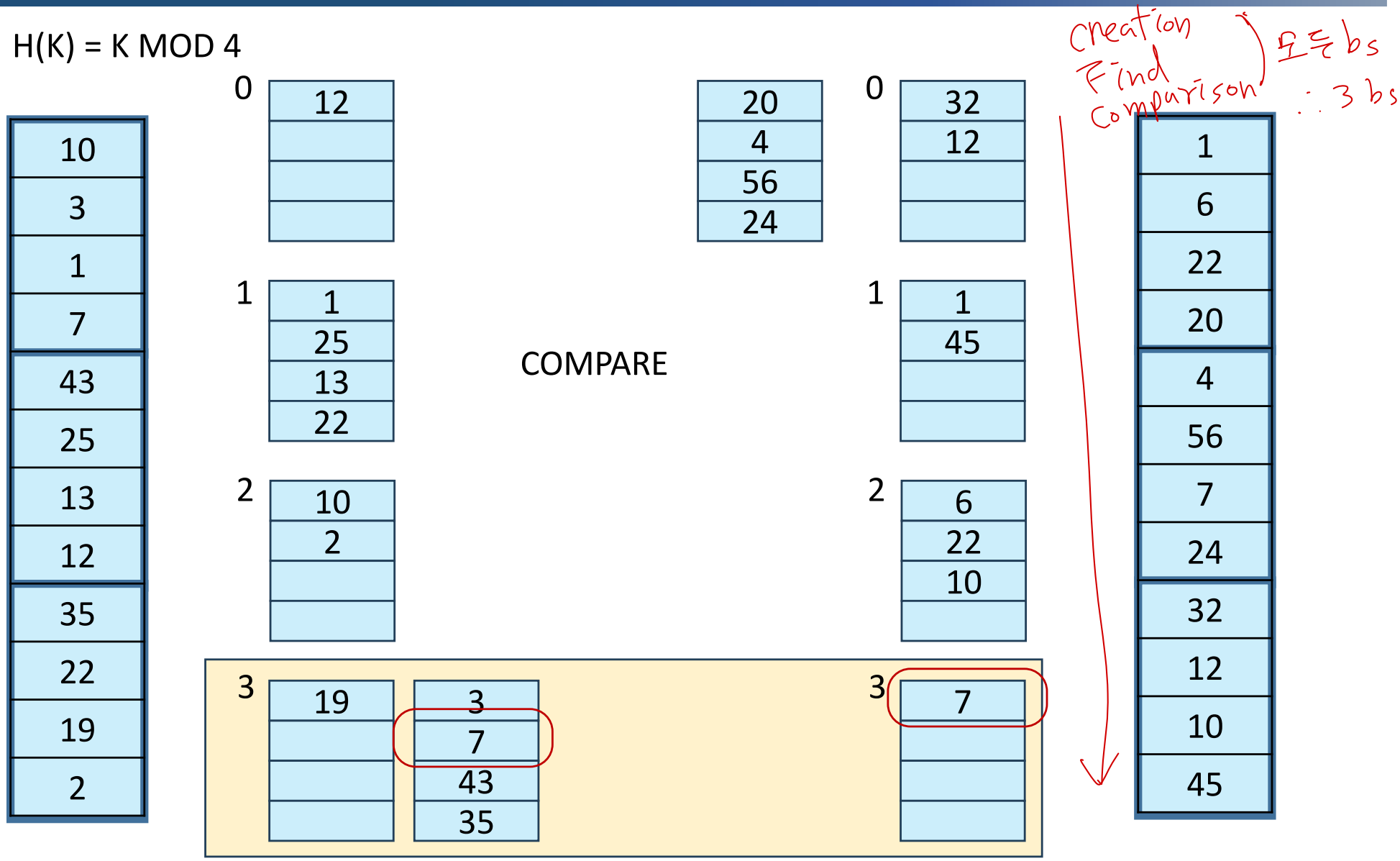
Tuples need only to be compared with tuples in the same bucket

Hash-Join Example



Tuples need only to be compared with tuples in the same bucket

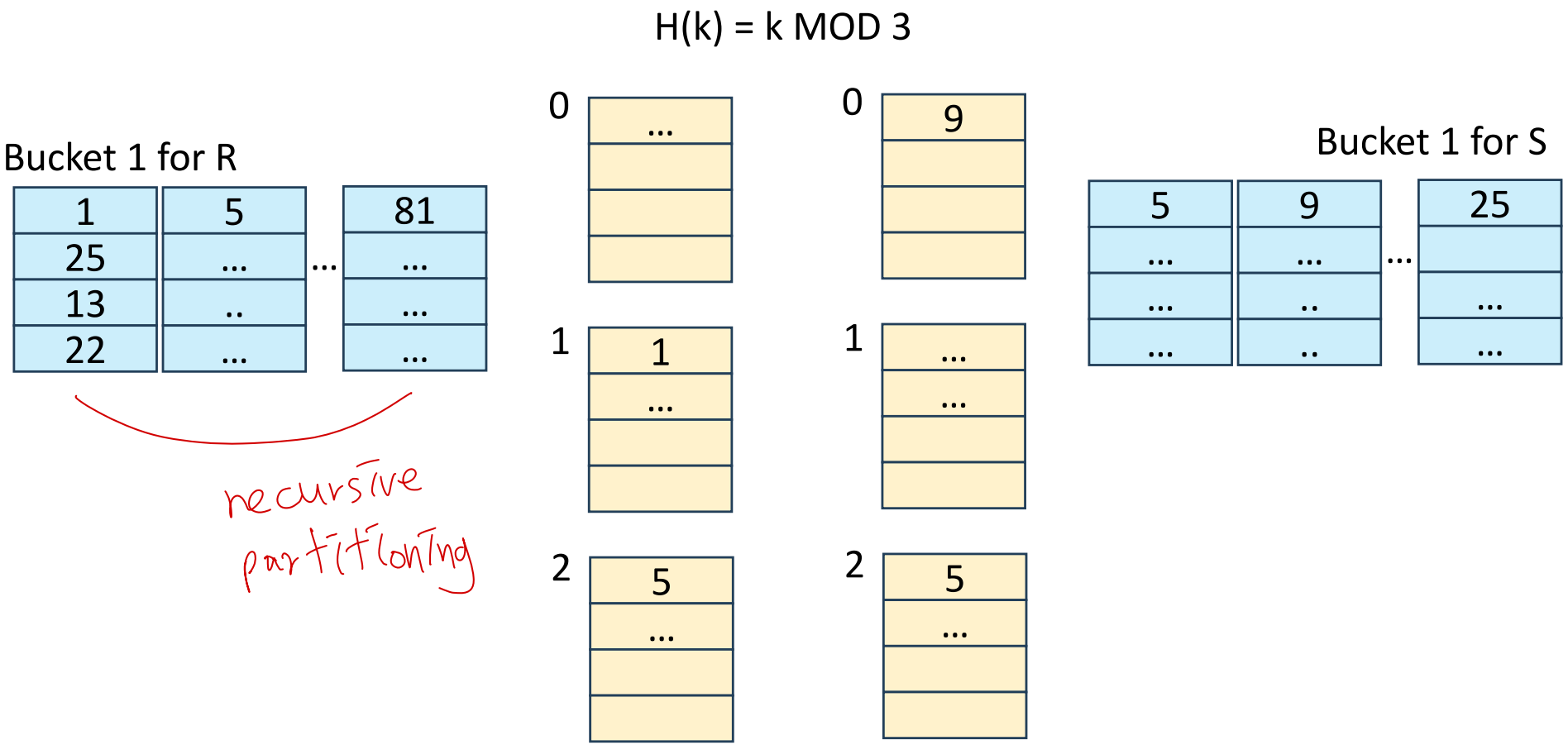
Hash-Join Example



Tuples need only to be compared with tuples in the same bucket

Hash-Join Example

- If a bucket size is too large, recursive partitioning helps
- Use a different hash function



Hash-Join Algorithm

The hash-join of *r* and *s* is computed as follows.

pseudocode

1. Partition the relation *s* using hashing function *h*. When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
2. Partition *r* similarly.
3. For each *i:*
   (a) Load s_i into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one *h.*
   (b) Read the tuples in r_i from the disk one by one. For each tuple t_r locate each matching tuple t_s in s_i using the in-memory hash index. Output the concatenation of their attributes.

   Relation *s* is called the **build input** and *r* is called the **probe input.**

Cost of Hash-Join

- If recursive partitioning is not required: cost of hash join is
  $3(b_r + b_s) + \alpha$ block transfers + $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$ seeks
  - b_b : # of blocks allocated for input/output buffer
  - Partitioning : $2(b_r + b_s)$
  - build and probe phase: $(b_r + b_s)$
  - α is an overhead for partially filled blocks

- If the entire build input can be kept in main memory no partitioning is required
  - Cost estimate goes down to $b_r + b_s$.

Example of Cost of Hash-Join

- *instructor* ⋈ *teaches*

- Assume that memory size is 20 blocks
- $b_{instructor}$= 100 and $b_{teaches}$ = 400.
- *instructor* is to be used as build input. Partition it into five partitions, each of size 20 blocks. This partitioning can be done in one pass.
- Similarly, partition *teaches* into five partitions, each of size 80.
- This is also done in one pass.
- Therefore, total cost, ignoring cost of writing partially filled blocks:
  - 3(100 + 400) = 1500 block transfers +
    2($\lceil 100/3 \rceil + \lceil 400/3 \rceil$) = 336 seeks
    - b_b : # of blocks allocated for input = 3
      # of blocks allocated for output = 5

Complex Joins

- Join with a conjunctive condition:

$$r \bowtie_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n} s$$

  - Either use nested loops/block nested loops, or
  - Compute the result of one of the simplest joins $r \bowtie_{\theta i} s$
    - final result comprises those tuples in the intermediate result that satisfy the remaining conditions

$$\theta_1 \wedge \ldots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \ldots \wedge \theta_n$$

- Join with a disjunctive condition

$$r \bowtie_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n} s$$

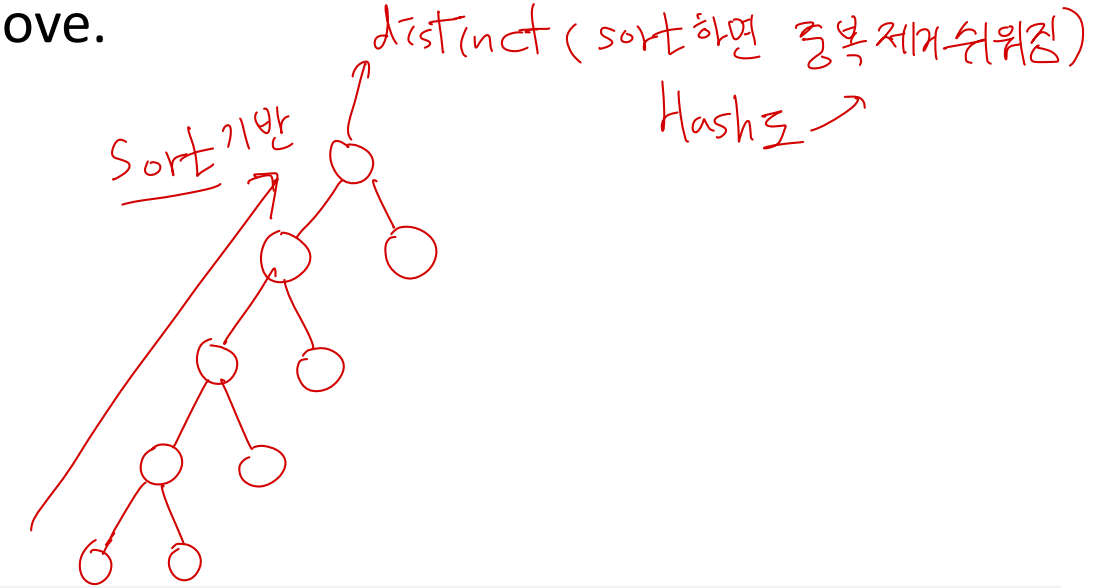
  - Either use nested loops/block nested loops, or
  - Compute as the union of the records in individual joins $r \bowtie_{\theta i} s$:

$$(r \bowtie_{\theta 1} s) \cup (r \bowtie_{\theta 2} s) \cup \ldots \cup (r \bowtie_{\theta n} s)$$

Other Operations

- **Duplicate elimination (distinct)** can be implemented via hashing or sorting.
  - On sorting duplicates will come adjacent to each other, and all but one set of duplicates can be deleted.
  - Hashing is similar – duplicates will come into the same bucket.

- **Projection:**
  - perform projection on each tuple and eliminate duplicate records by the method described above.



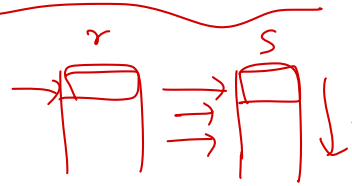
Other Operations : Set Operations

- **Set operations** ($\cup$, $\cap$ and $-$): can either use variant of merge-join after sorting, or variant of hash-join.

- E.g., Set operations using hashing:
  1. Partition both relations using the same hash function
  2. Process each partition *i* as follows.
     1. Using a different hashing function, build an in-memory hash index on r_i.
     2. Process s_i as follows
        - $r \cup s$:
          1. Add tuples in s_i to the hash index if they are not already in it.
          2. At end of s_i, add the tuples in the hash index to the result.
        - $r \cap s$:
          1. output tuples in s_i to the result if they are already there in the hash index
        - $r - s$:
          1. for each tuple in s_i, if it is there in the hash index, delete it from the index.
          2. At end of s_i add remaining tuples in the hash index to the result.

Other Operations : Outer Join

- **Outer join** can be computed either as
  - A join followed by addition of null-padded non-participating tuples.
- Modifying merge join to compute $r \mathbin{⟕} s$
  - During merging, for every tuple t_r from r that do not match any tuple in s, output t_r padded with nulls.
  - Right outer-join and full outer-join can be computed similarly.
- Modifying hash join to compute $r \mathbin{⟕} s$
  - If r is probe relation, output non-matching r tuples padded with nulls
  - If r is build relation, when probing keep track of which r tuples matched s tuples. At end of s_i output non-matched r tuples padded with nulls



Other Operations : Aggregation

- **Aggregation** can be implemented in a manner similar to duplicate elimination.
  - *E.g.)* ***select*** *dept_name,* ***avg****(salary)*
    from *instructor*
    group *by dept_name*

max, min, sum → sort 활용
(중복제거와 비슷)

  - Sorting or hashing can be used to bring tuples in the same group together, and then the aggregate functions can be applied on each group.
  - As the groups are being constructed, apply the aggregation operations on the fly.
    - For count, min, max, sum: keep aggregate values on tuples found so far in the group.
      - When combining partial aggregate for count, add up the aggregates
    - For avg, keep sum and count, and divide sum by count at the end

Evaluation of Expressions

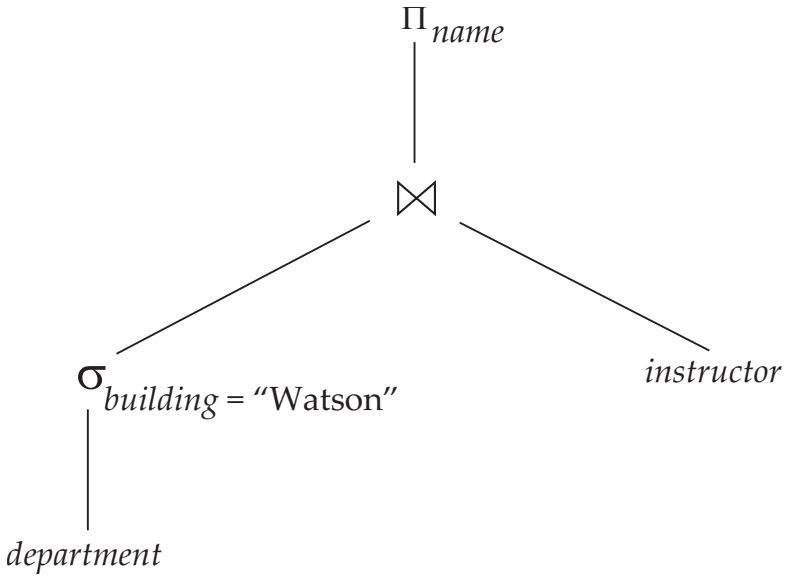
Skip

- So far: we have seen algorithms for individual operations
- Alternatives for evaluating an expression containing multiple operations
  - Materialization: generate results of an expression and reuse
  - Pipelining: evaluate several operations simultaneously

Materialization

- Materialized evaluation: evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.
  - **Materialize: store a temporary relation to disk.
- E.g., in figure below, compute and store then compute its join with *instructor*, and finally compute the projection on name.

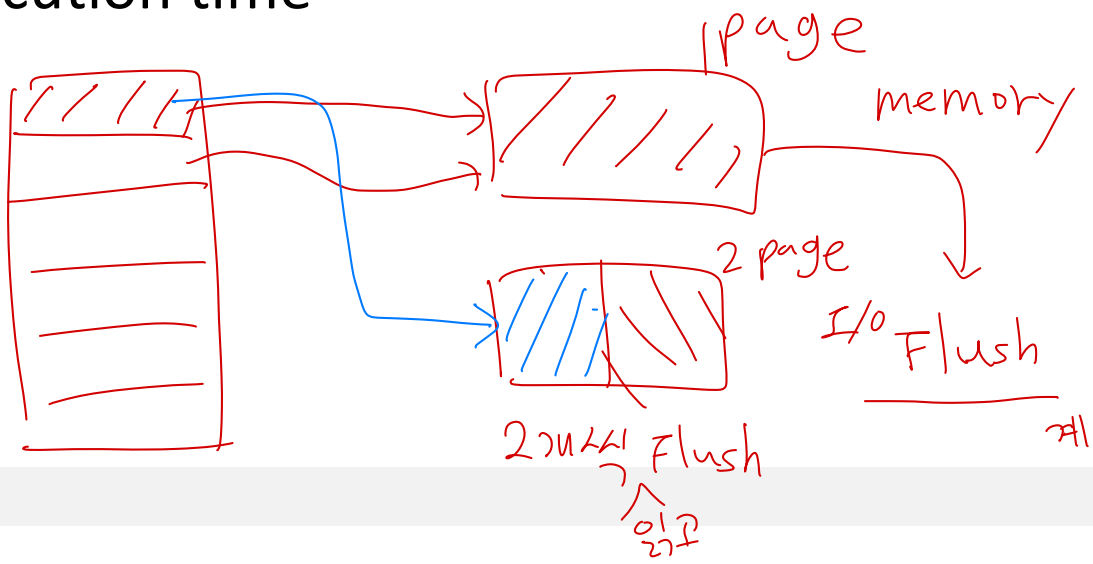
$$\sigma_{building="\text{Watson}"}(department)$$



Materialization (Cont.)

- Materialized evaluation is always applicable
  - Cost of writing results to disk and reading them back can be quite high

- Double buffering: use two output buffers for each operation, when one is full write it to disk while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time



Pipelining

<일반적 방법>
하위계층 연산이 끝나고 상위계층 연산시작
(materialize, Disk I/O => slow)

<pipeline> 여러스레드가 동시에 동작
하위계층 연산 결과를 바로 위로 보냄
위에서 연산하다가 메모리 full 되면 하위계층 stop!
(producer driven 방법 → fast)

- **Pipelined evaluation** : evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of

$$\sigma_{building="\text{Watson}"}(department)$$

  - instead, pass tuples directly to the join.. Similarly, don't store result of join, pass tuples directly to projection.

- Much cheaper than materialization: no need to store a temporary relation to disk.
- Pipelining may not always be possible – e.g., sort, hash-join.
- Pipelines can be executed in two ways: ① **demand driven** and ② **producer driven**

① demand driven
↳ 상위계층이 하위 계층에 요구할때만

② producer driven
하위계층 연산결과 즉시위로 보냄
위에서연산하다
→ 메모리 full되면 하위 stop!

※ 만약 계층간의 연산속도가 다르다면
fast가 slow를 기다린다.

Pipelining (Cont.)

- In demand driven or lazy evaluation
  - system repeatedly requests next tuple from top level operation
  - Each operation requests next tuple from children operations as required, in order to output its next tuple
  - In between calls, operation has to maintain state so it knows what to return next

Pipelining (Cont.)

- In producer-driven or eager pipelining
  - Operators produce tuples eagerly and pass them up to their parents
    - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
    - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
  - System schedules operations that have space in output buffer and can process more input tuples
- Alternative name: pull and push models of pipelining