Lecture 13
Query processing

Subject Lecturer: Kevin K.F. YUEN, PhD.

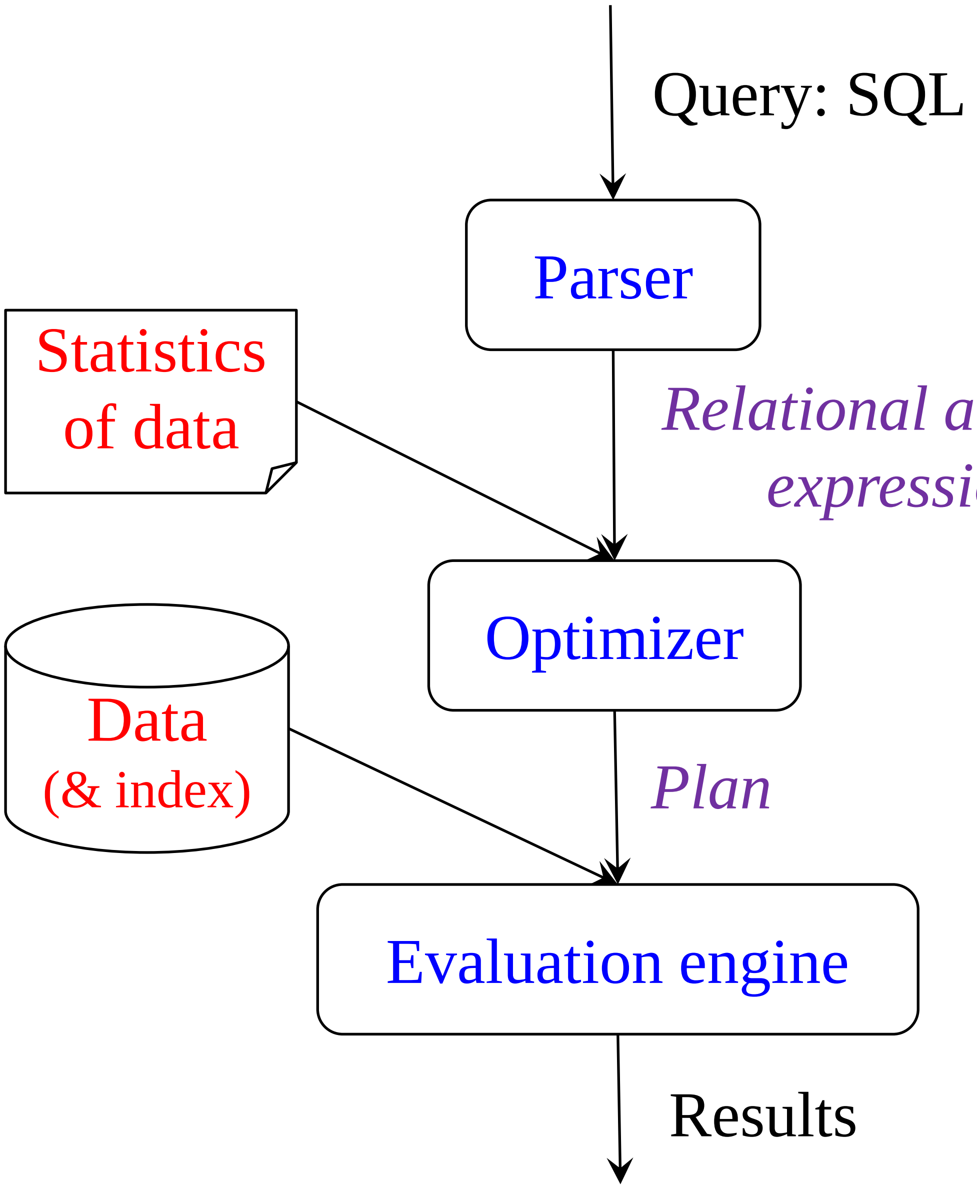
Acknowledgement: Slides were offered from Prof. Ken Yiu.
Some parts might be revised and indicated.

Slides content modified from
Database Systems Concepts

Outline

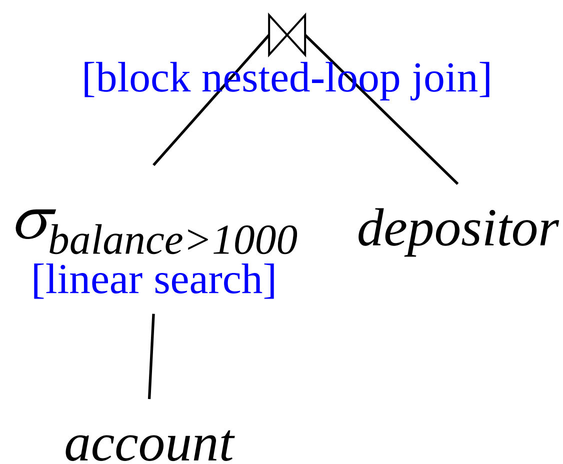
- Basic concepts for query processing
- How to process a selection?
- How to process a join?
- How to execute a plan?

How to process a query?



SELECT …
FROM …
WHERE …

$\Pi_{\ldots}(\sigma_{\ldots}(\ldots \bowtie \ldots))$



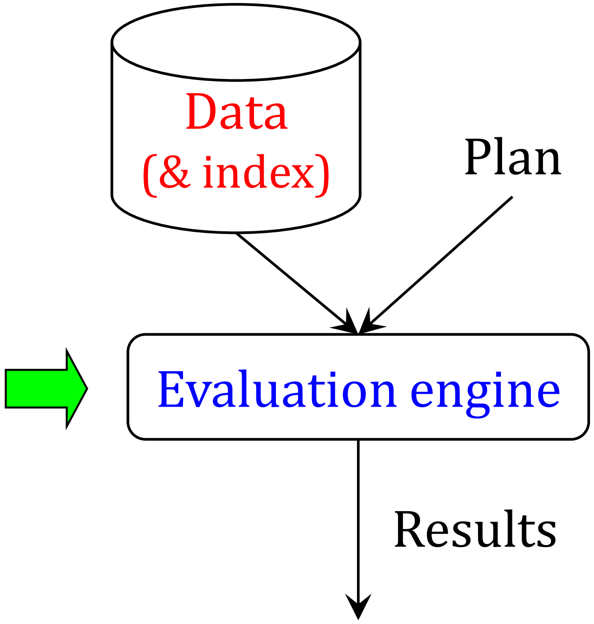
How to process a query?

- Optimizer
  - Find the evaluation plan with the lowest estimated cost
    - We skip this issue in this course

- Evaluation engine
  - Call an algorithm to evaluate a relational algebra operation
  - Combine individual operations to evaluate a complete plan

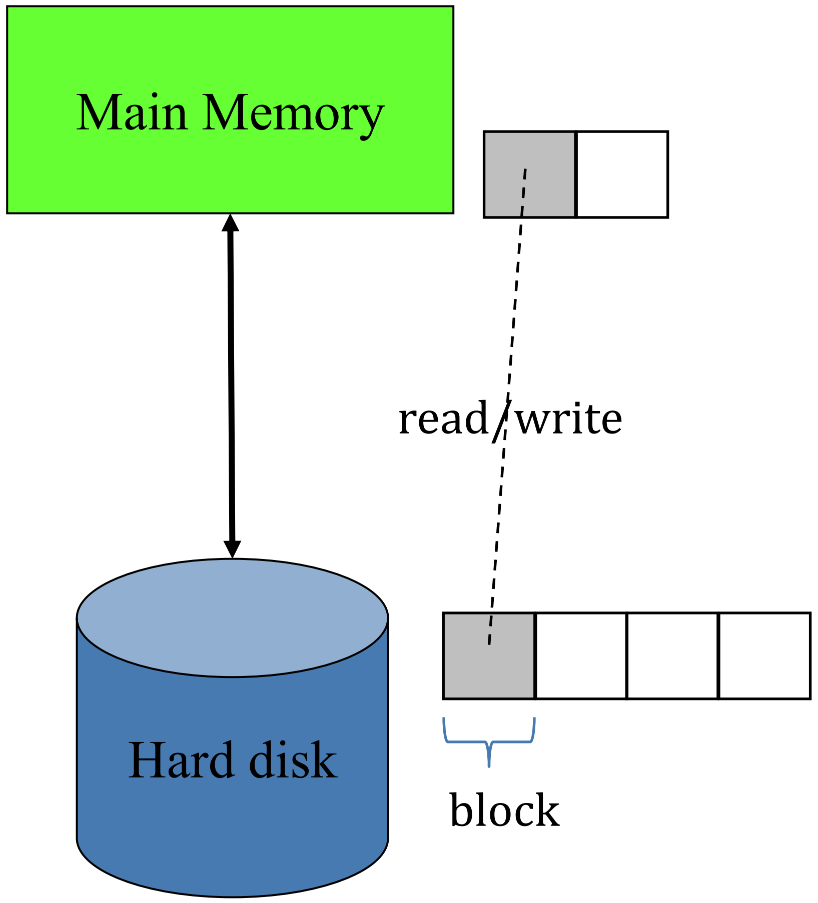
Evaluation engine: How to execute a plan?

- How to measure the cost of a plan?
- Some methods (physical operators) for executing
  - The selection operation
  - The join operation
- How to execute a plan?



Cost

- Cost = the number of disk block transfers
- Assumptions in RDBMS
  - Ignore CPU costs
  - Ignore the cost of writing the final output to disk
- Extra assumption in this lecture
  - Ignore the disk seek time, because the number of disk block transfers is much larger than the number of disk seeks



Outline

- Basic concepts for query processing
- How to process a selection?
- How to process a join?
- How to execute a plan?

Selection Operation

- *Example*: $\sigma_{balance<2500}(account)$
- Several different algorithms to implement selections
  - Usually choose the cheapest available one

Algorithm / physical operator	Cost (# disk blocks)
Linear search	b_r
Primary index, equality on key	$h_r + 1$
Primary index, equality on non-key	$h_r + b_{results}$
Secondary index, equality	$h_r + n_{results}$

b_r: size of r in blocks

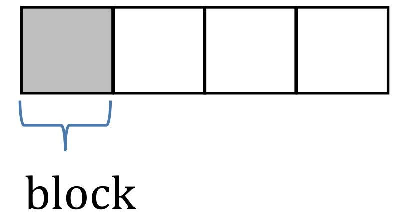
h_r: height of B^+-tree on r

$n_{results}$: # of results

$b_{results}$: result size in blocks

For simplicity, we measure the cost as the number of disk block transfers.
When using an index other than B^+-tree, replace the term h_r by the index lookup cost.

Selection



- (1) *Linear search*:

  Scan each file block and test all tuples

  - Applicable to any type of comparison condition
  - Cost = b_r blocks

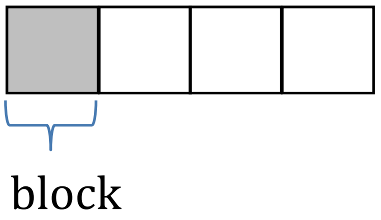
    (b_r: number of blocks occupied by relation r)

- (2) *Primary index on candidate key, equality*:

  Retrieve a single tuple that satisfies the equality condition

  - Cost = h_r + 1 blocks
    - B^+-tree height $h_r = \lceil \log_{\lceil f/2 \rceil} n_r \rceil$
    - n_r: number of records in relation r
    - f: max. number of children in a node

Selection



- (3) *Primary index on nonkey, equality:*

  Retrieve multiple (consecutive) tuples that satisfies the equality condition

  - Cost = $h_r + b_{match}$ blocks
    - This requires estimating b_{match}: number of blocks containing matching tuples

- (4) *Secondary index on nonkey, equality* :

  Retrieve multiple tuples that satisfies the equality condition

  - Cost = $h_r + n_{match}$ blocks
    - n_{match} denotes the number of matching records

Outline

- Basic concepts for query processing
- How to process a selection?
- How to process a join?
- How to execute a plan?

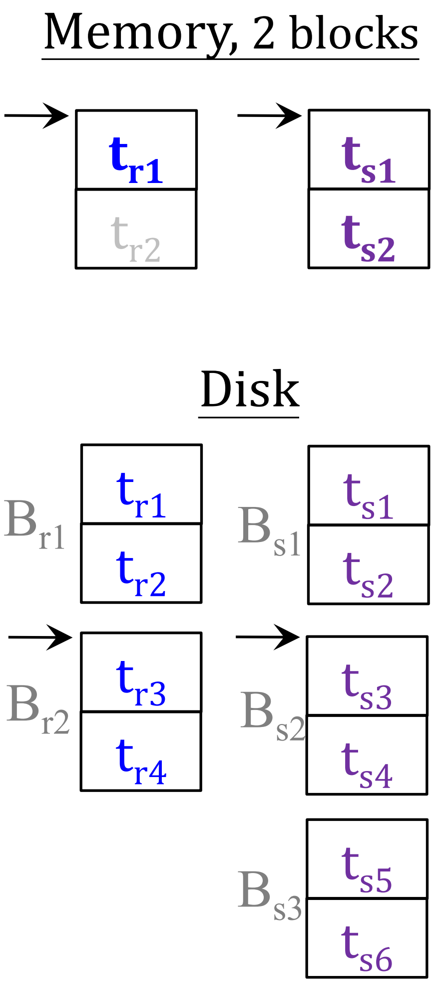
Join Operation

- *Example*: *customer* ⋈ *depositor*
- Several different algorithms to implement joins
  - Usually choose the cheapest available one

Algorithm / physical operator	Cost (# disk blocks)
Nested-loop join	$n_r * b_s + b_r$ [worst case]
Block nested-loop join	$\lceil b_r / (M–2) \rceil * b_s + b_r$
Indexed nested-loop join	$n_r * (h_s + 1) + b_r$
Merge-join	$b_r + b_s$ $+ b_r (2 \lceil \log_{M–1}(b_r/M) \rceil + 1)$ $+ b_s (2 \lceil \log_{M–1}(b_s/M) \rceil + 1)$
Hash-join	$3(b_r + b_s)$ if no recursive partitioning required

For simplicity, we measure the cost as the number of disk block transfers.

Nested-Loop Join



Outer relation

for each tuple t_r **in** r **do**

Inner relation

  for each tuple t_s **in** s **do**

    if pair (t_r, t_s) satisfies the join condition

    then add (t_r, t_s) to the result

- Applicable to any join condition, index not required
- Cost of *nested-loop join*:
  $n_r * b_s + b_r$ blocks
  - Assume the worst case: only one memory buffer block for each relation

The sequence of blocks read from the disk

Index Nested-Loop Join

for each tuple t_r **in** r **do**

    search the index on s to find tuples that match with t_r

    for each matching tuple t_s **in** s **do**

        if pair (t_r,t_s) satisfies the join condition

        then add (t_r,t_s) to the result

- Requires an index, applicable to equality condition only
- Cost of *indexed nested-loop join*:
  $n_r * (h_s + 1) + b_r$ blocks

Exercise: *customer* ⋈ *depositor*

- Number of tuples: $n_{customer}$=10000 $n_{depositor}$=5000
- Number of blocks: $b_{customer}$=400 $b_{depositor}$ =100
- Suppose that *customer* has a primary B⁺-tree index on the join attribute *customer-name*, which contains 20 child pointers per index node.

- Cost of *nested-loop join*

  If outer relation = *depositor*

  cost : ______ * ______ + ______ = _________ blocks

  If outer relation = *customer*

  cost : ______ * ______ + ______ = _________ blocks

- Cost of *indexed nested-loop join*

  E.g., the tree height (for *customer*) is 4, so the cost is:

  5000*(**4**+1)+100 = 25100 blocks

$$\lceil \log_{\lceil f/2 \rceil}(n_{customer}) \rceil$$
$$= \lceil \log_{\lceil 20/2 \rceil}(10000) \rceil = 4$$

* Try to find "log(10000)/log(10)"

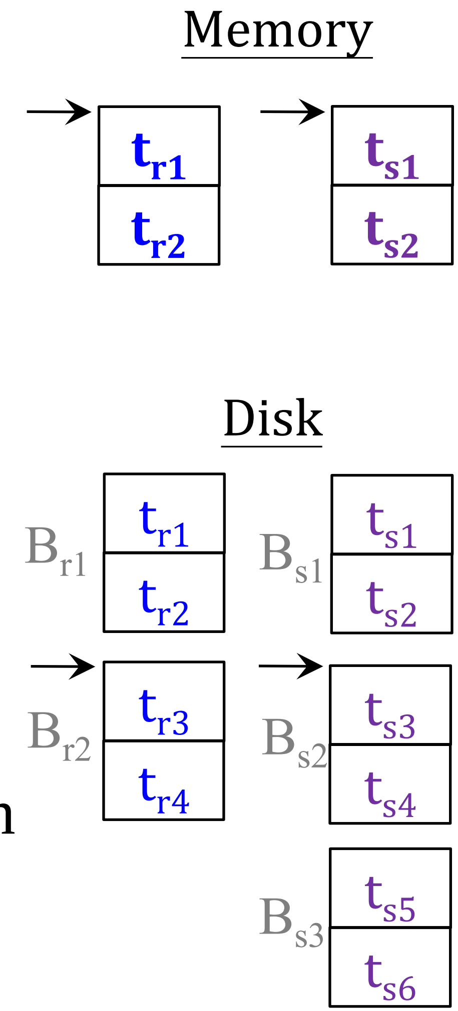
How to compute this in calculator?

* Type "10000" "log" "/" "10" "log"

Block Nested-Loop Join

- Variant of nested-loop join
- Pair every block of the inner relation with every block of the outer relation
  - In this method, the relation *s* is scanned ____ times

for each block B_r **of** *r* **do**
  for each block B_s **of** ***s*** **do**
    for each tuple t_r **in** B_r **do**
      for each tuple t_s **in** B_s **do**
        if pair (t_r, t_s) satisfies the join condition
        then add (t_r, t_s) to the result

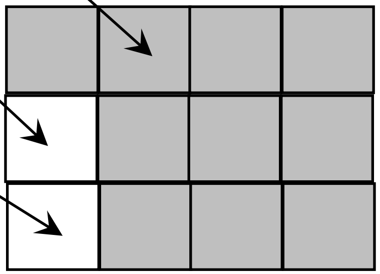


The sequence of blocks read from the disk

B_{r1}	B_{s1}	B_{s2}	B_{s3}	B_{r2}	B_{s1}	B_{s2}	B_{s3}

Block Nested-Loop Join (Cont.)

- More memory blocks may be used to reduce the cost of block nested-loop join
- If the memory has M blocks, use them as follows:
  - $M – 2$ memory blocks to buffer the outer relation
  - 1 block to buffer the inner relation
  - 1 block to buffer the output
- Cost = $\lceil b_r / (M–2) \rceil * b_s + b_r$ blocks



Merge-Join

1. Sort both relations on their join attribute <u>*(to discuss soon)*</u>
2. Merge the sorted relations to join them
    a. This step is like the merge stage of the sort-merge algorithm.
    b. The difference is to handle duplicate values in join attribute
    — every pair with same value on join attribute must be matched

- Applicable to equi-joins and natural joins
- Each block is only read once
    - assuming all tuples for any given value of the join attributes fit in memory
- Cost of merge join =

  $b_r + b_s$ blocks

  + the sorting cost (if relations are unsorted)

pr

a1	*a2*
a	3
b	1
d	8
d	13
f	7
m	5
q	6

r

ps

a1	*a3*
a	A
b	G
c	L
d	N
m	B

s

Sorting: External Sort-Merge

Example: M = 3 memory blocks

- Use it when the relation is larger than the main memory, i.e., $b_r > M$
- **External sort-merge** algorithm
  - 1. Create sorted runs
    - Read consecutive M blocks into memory, sort it, then write to a run
  - 2. Merging until only 1 run remains
- Cost: $b_r\,(\,2\lceil \log_{M-1}(b_r/M)\rceil + 1)$
  - Number of merge passes required: $\lceil \log_{M-1}(b_r/M)\rceil$
  - Block transfers for initial run and in each pass is $2b_r$

g	24
a	19
d	31
c	33
b	14
e	16
r	16
d	21
m	3
p	2
d	7
a	14

initial relation

create runs

a	19
d	31
g	24

b	14
c	33
e	16

d	21
m	3
r	16

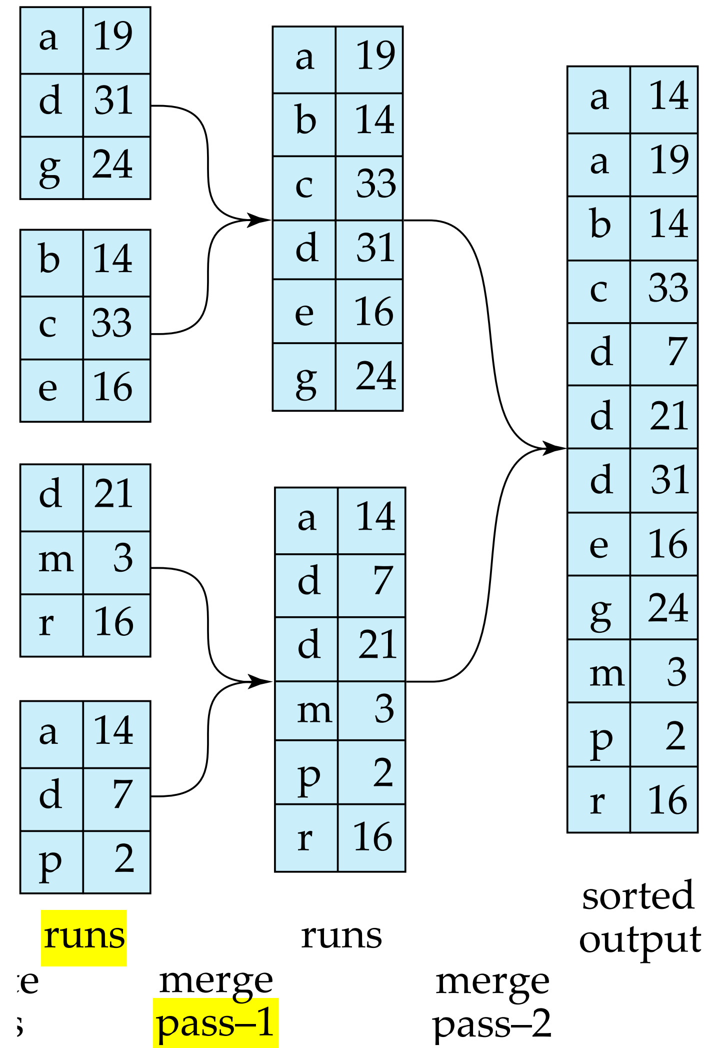
a	14
d	7
p	2

runs

Sorting: External Sort-Merge

- How to merge "sorted runs"?
- 2. Merging (for every M–1 runs)
  - Use M–1 memory blocks as input buffers, and 1 memory block as output buffer
  - Move the smallest tuple from its input buffer to the output buffer
  - An input block empty → fill it with the next disk block from its input run
  - An output block full → flush the block to its output run
- Repeat until only 1 run remains

Example: M = 3 memory blocks



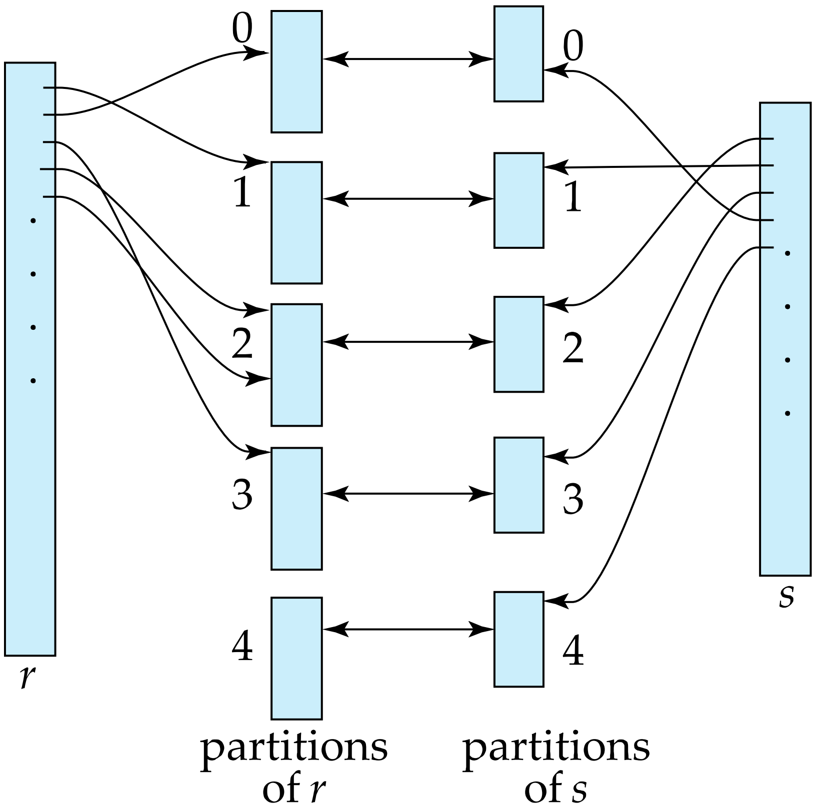
Hash-Join

Example: n_h = 5 partitions

Observation: if two records have the same key value (on the join attribute), then those two keys must have the same hash value

Idea of hash-join: use a hash function *h* to hash records into partitions

Hash-join is applicable to equi-joins and natural joins



r : **probe input**

s : **build input**

Hash-Join

Call the smaller relation s as build input

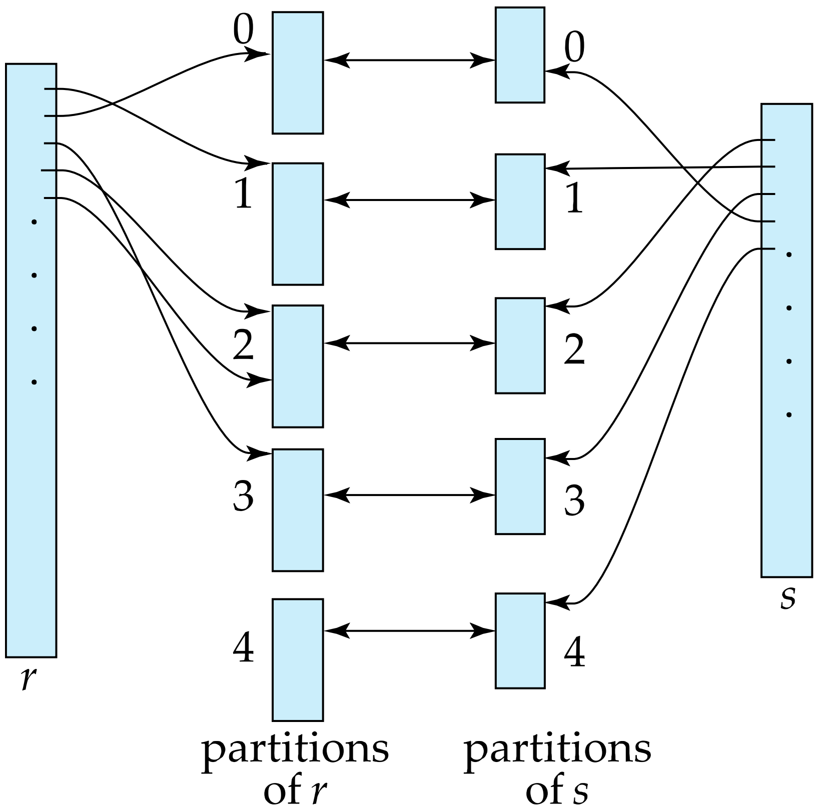
Step 1. Partition the relation s using a **hash function** h (into n_h partitions)

* note that n_h must be smaller than M

Step 2. Partition r similarly

Requirement: in step 1, we require that each partition of s must have at most M–2 blocks (i.e., can fit in memory)

Example: n_h = 5 partitions

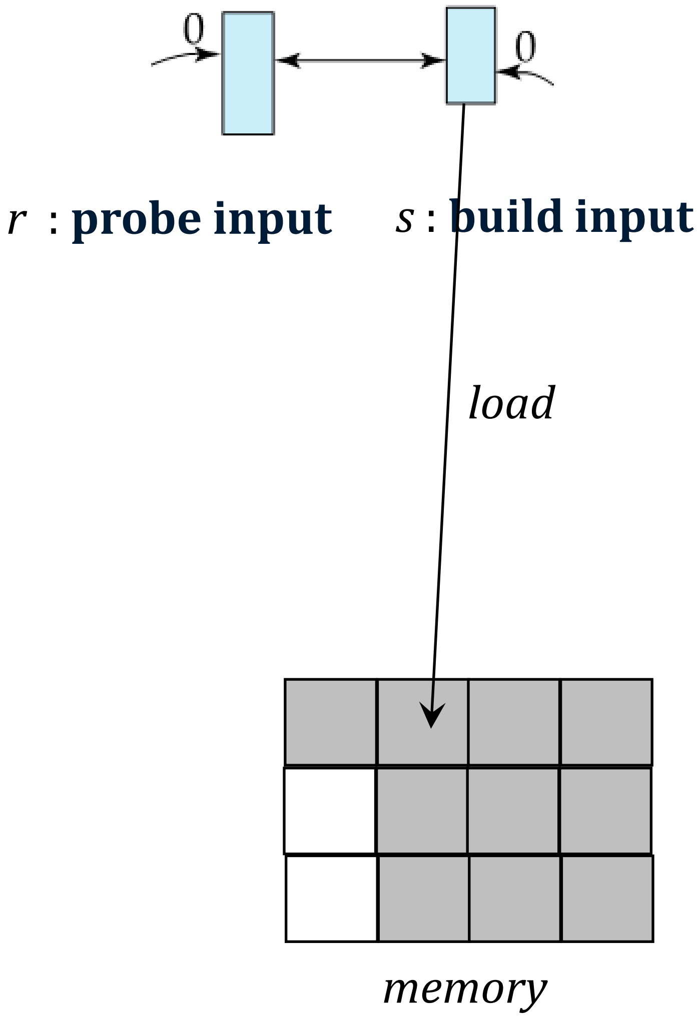


r : **probe input**

s : **build input**

Hash-Join

Example: n_h = 5 partitions



Step 3. For each partition value *i*

(a) Create another hash function *h'*

(b) Load the entire s_i into memory.

Build an in-memory hash index on it by the join attribute (by using *h'*).

(c) Read the tuples in r_i.

For each tuple t_r , find each matching tuple t_s in s_i from in-memory hash index. Output the result.

Purpose of the in-memory hash index:

reduce the computation cost on matching tuples

Hash-Join algorithm (Cont.)

- Suppose that number of partitions n_h is at most number of memory blocks M

- Cost of hash join: $3(b_r + b_s)$
  - Partitioning phase: $2(b_r + b_s)$
    - read and write each relation once
  - Build and probe phase: $b_r + b_s$
  - Partially filled blocks: $2 (2 n_h)$

small compared to other terms, can be ignored

- **Recursive partitioning** is not required if $M > n_h + 1$
  - E.g., recursive partitioning not needed for relations of 1GB with memory size of 2MB and block size of 4KB

Exercise: Cost of Hash-Join

$$customer \bowtie depositor$$

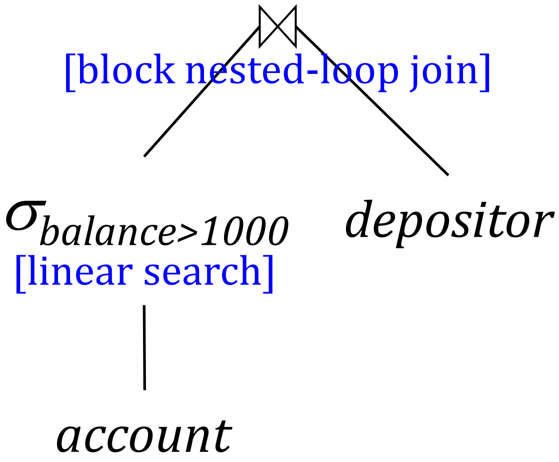
- Given that
  - memory size is 20 blocks
  - $b_{depositor}$= 100 and $b_{customer}$ = 400.
- Use the smaller relation (*depositor*) as build input
- How large should a partition be?
  - To make each partition of *depositor* fit in memory (20 blocks), we can partition it into $\lceil 100/(20-2) \rceil = 6$ partitions
  - Since 6 < 20, this partitioning can be done in **one pass**
  - Similarly, we can divide *customer* into 6 partitions
- The total cost:
  - 3(100 + 400) = 1500 blocks

Outline

- Basic concepts for query processing
- How to process a selection?
- How to process a join?
- How to execute a plan?

How to execute a plan?

- Naïve approach: execute physical operators one-by-one
  - First execute "linear search" (for selection)
    - if the intermediate result exceeds the memory size, need to write it to the disk → additional I/O cost
  - Then execute "block nested-loop join" (for natural join)



- Drawback
  - High latency for the first result record
  - Additional I/O cost for writing intermediate result to the disk

How to execute a plan?

- The **iterator approach**: each physical operator implements the "Iterator" interface
  - Open( ): open the file/index, allocate buffer
  - GetNext( ): produce a record as output
  - Close( ): close the file/index, deallocate buffer

$\sigma_{balance>1000}$
[linear search]

account

- *Example*: the iterator for linear search

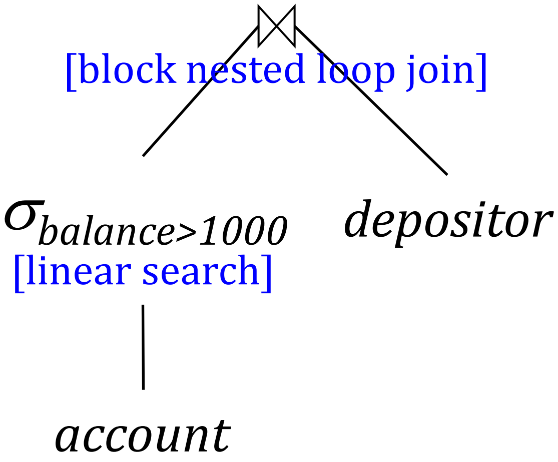
B:

Open ( )
1. allocate a memory block *B*
2. open the file

GetNext ( )
1. repeat
2. if *B* is empty, load the next block from disk to *B*
3. *t* ← pop the next tuple from *B*
4. until *t* satisfies the predicate
5. return *t*

How to execute a plan?

- Advantages of the iterator approach
  - Low latency for the first result record
  - We can pipeline intermediate results to the next/parent operator, without additional I/O cost
- Iterators incur computational overhead
  - Acceptable in traditional RDBMS



- An issue in query optimization
  - How to allocate/share the memory to different physical operators?

Summary

- How to process a selection?
  - Linear scan, several types of index scan
- How to process a join?
  - Nested-loop, block nested-loop,
    block nested-loop, merge-join, hash-join
- Query plan execution

Sample solutions

- Please be reminded to submit Assignment 2 on time
- We will post the sample solutions of Quiz 2 and Assignment 2
  at least one day before the exam

Sample types of questions

- Give an example of … such that …
- Draw … such that …
- Run … show the steps …
- Find … show the steps …
- Write an algorithm to …
- Check … explain …

Wish you good luck! ☺