Intro to DB

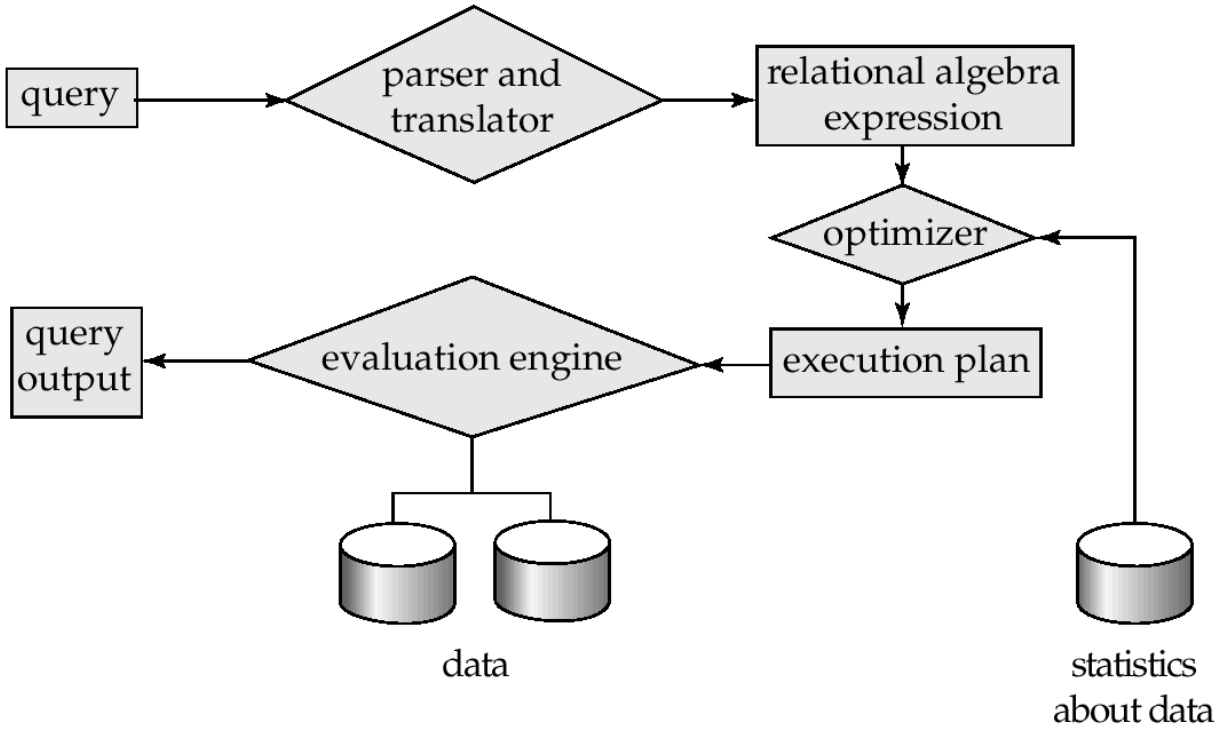
CHAPTER 12
QUERY PROCESSING

Chapter 12: Query Processing

- Overview
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions
- Cost Estimation of Expressions (Chap. 13)

Basic Steps in Query Processing

1. Parsing and translation
   - translate the query into an internal form (eg., relational algebra)
   - Parser checks syntax, verifies relations
2. Optimization
   - More than one way to evaluate a query
3. Evaluation
   - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.



Query Plan

- Evaluation primitive
  - a relational algebra expression annotated with instructions on how to evaluate it
    - $\sigma_{balance>2500}(account)$: use index 1
    - $\sigma_{balance>2500}(account)$: use table scan
-
  - a sequence of primitive operations that can be used to evaluate a query
- Example:
  - SELECT *balance* FROM *account* WHERE *balance*>2500

$\Pi_{balance}$

$\sigma_{balance>2500}$: use index 1

account

$\sigma_{balance>2500}$: seq. scan

$\Pi_{balance}$

account

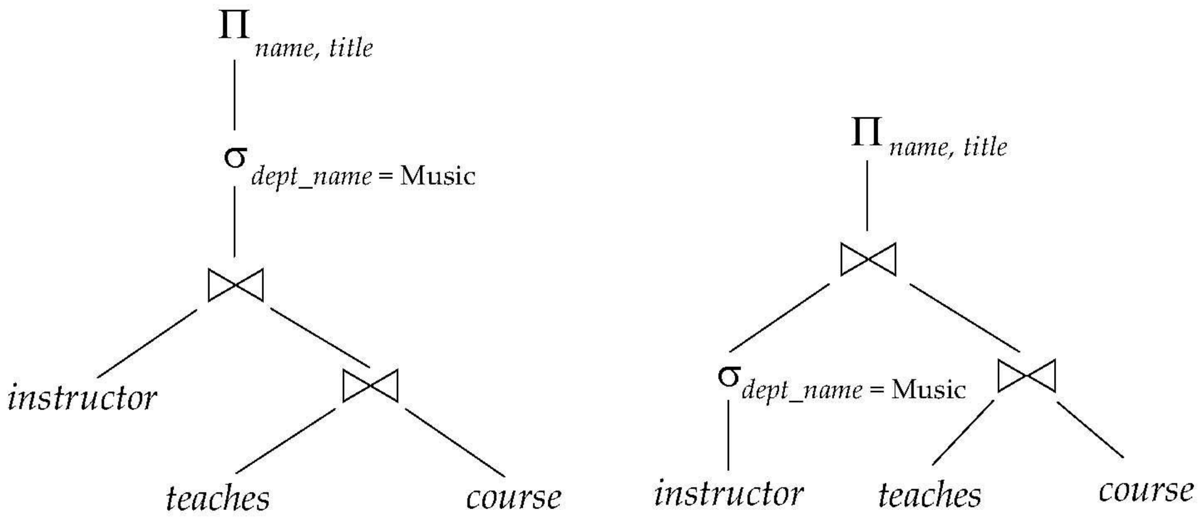
Query Optimization

- More than one way to evaluate a query
-

Given a DB schema *S*, a query *Q* on *S* is <u>*equivalent*</u> to another query *Q'* on *S*, if the answer sets of *Q* and *Q'* are the same in *any* instances of the DB.

$\Pi_{name,\ title}(\sigma_{dept=\text{"Music"}}(\ instructor \bowtie (teaches \bowtie course\ )))$ vs

$\Pi_{name,\ title}(\ (\sigma_{dept=\text{"Music"}}(instructor)) \bowtie (teaches \bowtie course)\ )$



Query Optimization

- Query optimization is the process of *selecting* the *most efficient query evaluation plan* for a given query

- Generation of Evaluation Plan
  1. Generating logically equivalent expressions
     - Use *equivalence rules* to transform an expression into an equivalent one.
  2. Annotating resulting expressions to get alternative query plans
  3. Choosing the cheapest plan based on *estimated cost*
- The overall process is called

Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
  - Many factors contribute to time cost
    - *disk accesses, CPU*, or even network *communication*
-
    - also relatively easy to estimate.
  - # of seeks $\times$ average-seek-cost
  - # of blocks read $\times$ average-block-read-cost
  - # of blocks written $\times$ average-block-write-cost

Measures of Query Cost (Cont.)

- For simplicity, we only use
  - **number of block transfers** *from disk*
    - t_T – time to transfer one block: 10~40 ms
  - **number of seeks**
    - t_S – time for one seek: 8~20 ms (disk seek + rotational delay)
  - Cost for *b* block transfers plus *s* seeks

- We ignore CPU costs
- We do not include cost of writing final output to disk
  - output of an operation may be sent to the parent operation without being written to disk

Selection Operation

- File scan
  - locate and retrieve records that fulfill a selection condition
  - Full file scan: retrieve all records of a file (relation)

- A1: *linear search*
  - Scan each file block and test all records on the selection condition
  - Cost estimate (number of disk blocks scanned) = $\boxed{}$
    - b_r : # of blocks containing records from relation *r*
  - If selection is on a key attribute, cost = $\boxed{}$
    - stop on finding record
  - Linear search can be applied regardless of
    - selection condition or
    - ordering of records in the file, or
    - availability of indices

Selections Using Indices

- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index.
- **A2** (**primary index, equality on key**). Retrieve a single record that satisfies the corresponding equality condition
  - $Cost = (h_i + 1) * (t_T + t_S)$
- **A3** (**primary index, equality on nonkey**) Retrieve multiple records.
  - Records will be on consecutive blocks
    - Let b = number of blocks containing matching records
  - $Cost = h_i * (t_T + t_S) + t_S + t_T * b$

Selections Using Indices

- **A4** (**secondary index, equality**).
  - Retrieve a single record if the search-key is a **candidate key**
    - *Cost* =
  - Retrieve multiple records if search-key is **not a candidate key**
    - each of *n* matching records may be on a different block
    - Cost =
      - Can be very expensive!

Selections Involving Comparisons

- Can implement selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using
  - a linear file scan,
  - or by using indices in the following ways:

- **A5** (**primary index, comparison**). (Relation is sorted on A)
  - For $\sigma_{A \geq V}(r)$ use index to find first tuple $\geq v$ and scan relation sequentially from there
  - For $\sigma_{A \leq V}(r)$ just scan relation sequentially till first tuple $> v$; do not use index

Selections Involving Comparisons (Cont.)

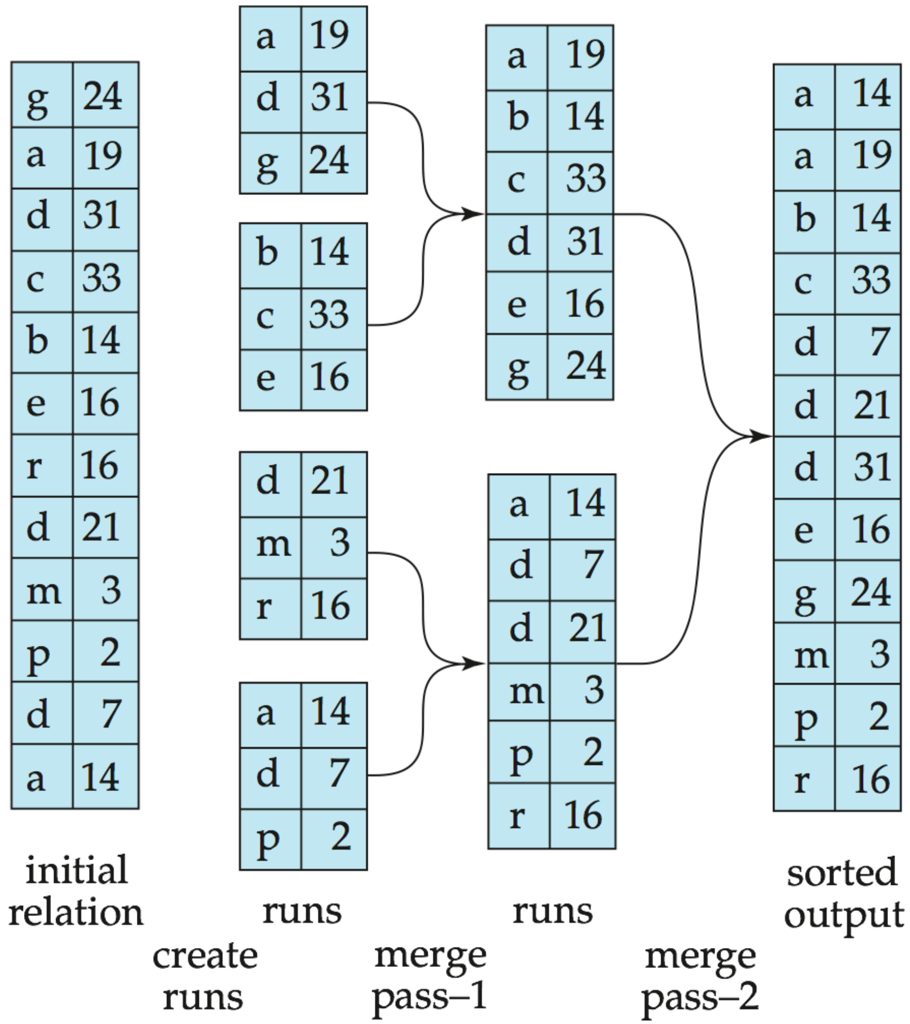
- Can implement selections of the form $\sigma_{A \le V}(r)$ or $\sigma_{A \ge V}(r)$ by using
  - a linear file scan,
  - or by using indices in the following ways:

- **A6** (**secondary index, comparison**).
  - For $\sigma_{A \ge V}(r)$ use index to find first index entry $\ge v$ and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \le V}(r)$ just scan leaf pages of index finding pointers to records, till first entry $> v$
  - In either case, retrieve records that are pointed to
    - requires an I/O for each record
    - Linear file scan may be cheaper

External Sort-Merge

- Sorting
  - Important core operation: order by, group by, join, …
  - One option: build an index, and use the index to read the relation in sorted order. May lead to one disk block access for each tuple.
- **External Sort-Merge**
  - Good choice for relations that don't fit in memory

M=3; 1 rec/block



External Sort-Merge

M=3; 1 rec/block

- Let M denote memory size (in pages).

1. **Create sorted runs**. Let i be 0 initially.

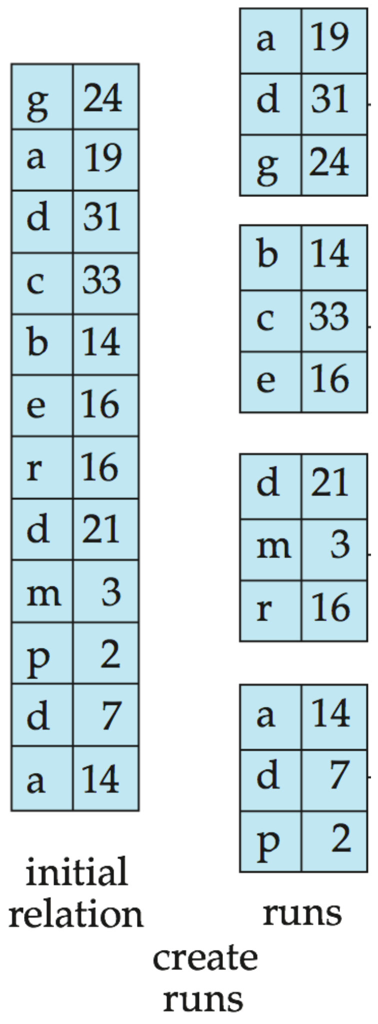
   Repeatedly do the following till the end of the relation:

   (a) Read M blocks of relation into memory

   (b) Sort the in-memory blocks

   (c) Write sorted data to run R_i ; increment i

   Let the final value of i be N

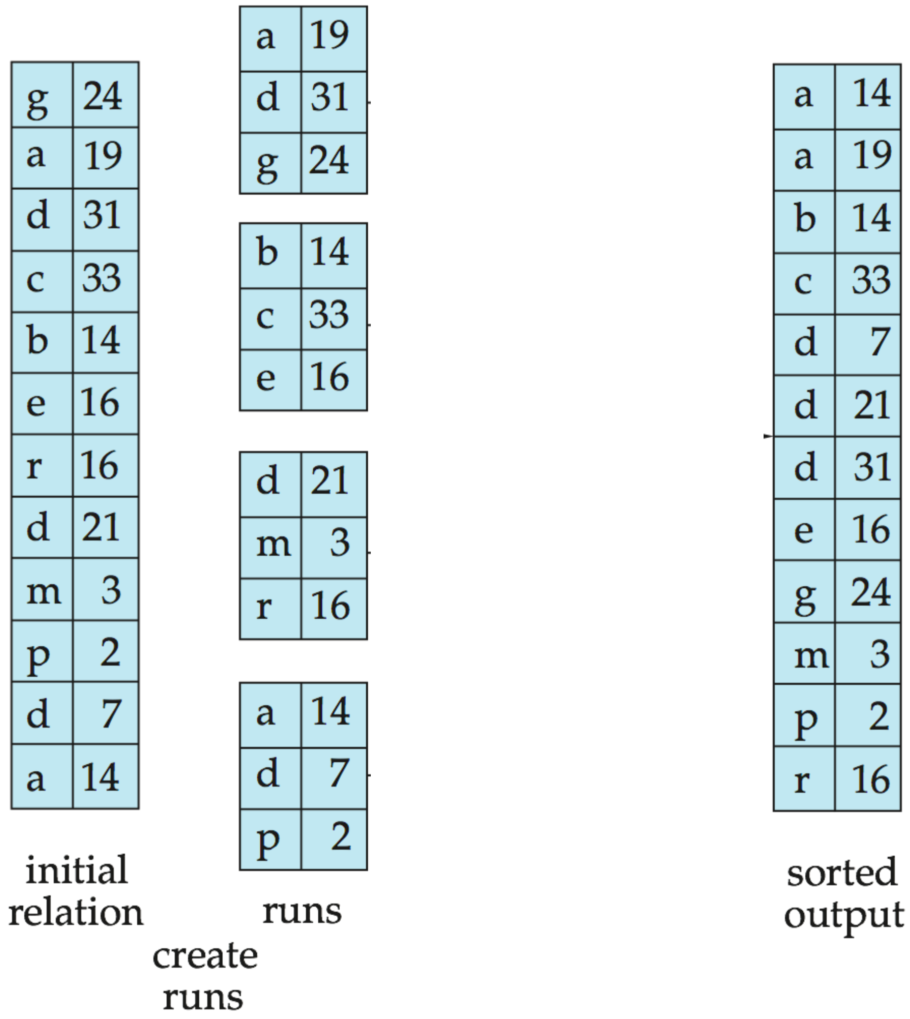


External Sort-Merge (Cont.)

2. **Merge the runs (N-way merge)**. (if $N < M$)
   1. Use N buffer blocks for input runs,

      1 buffer block for output.

      Read the first block of each run into its buffer page
   2. **repeat**
      1. Select the first record (in sort order) among all buffer pages
      2. Write the record to the output buffer. If the output buffer is full write it to disk.
      3. Delete the record from its input buffer page.
         If the buffer page becomes empty **then**
         read next block (if any) of the run into the buffer.
   3. **until** all input buffer pages are empty:

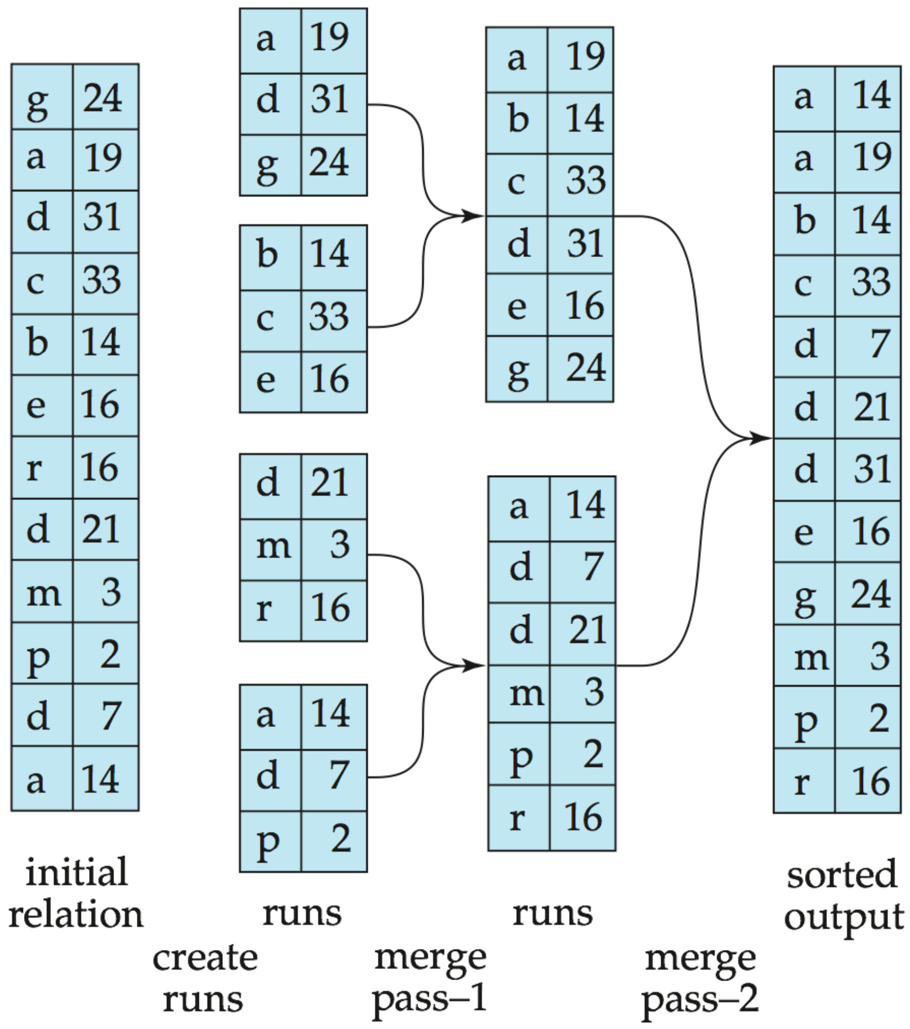


External Sort-Merge (Cont.)

M=3; 1 rec/block

2. **Merge the runs** (if $N \geq M$)

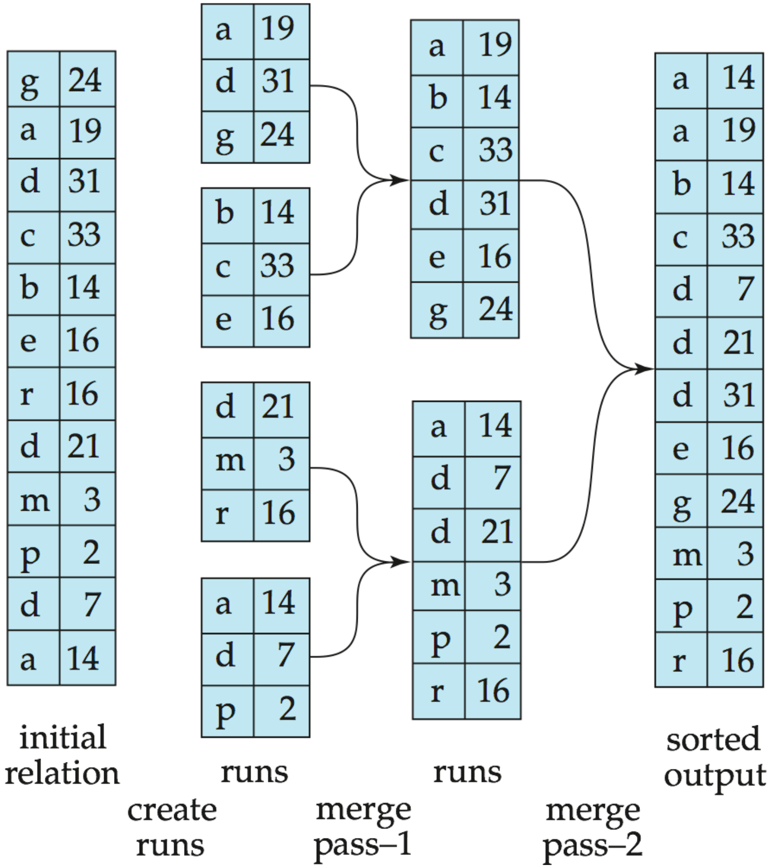
- If $N \geq M$, several merge *passes* are required.
  - In each pass, contiguous groups of M - 1 runs are merged.
  - A pass reduces the number of runs by a factor of M -1, and creates runs longer by the same factor.
    - E.g. If M=11, and there are 90 runs, one pass reduces the number of runs to 9, each 10 times the size of the initial runs
  - Repeated passes are performed till all runs have been merged into one.



External Sort-Merge – Cost Analysis

- Total number of merge passes required: $\lceil \log_{M-1}(b_r/M) \rceil$.
- Block transfers
  - for initial run creation as well as in each pass: $2b_r$
  - Thus total number of block transfers for external sorting:

    (we don't count final write cost)
- Seeks
  - Run generation: 1 seek to read and 1 seek to write each run
    - $2\lceil b_r / M \rceil$
  - During the merge phase
    - Buffer size per run: b_b (read/write b_b blocks at a time)
    - 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:



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
- Running Example
  - Number of records of *student*: 5,000 *takes*: 10,000
  - Number of blocks of *student*: 100 *takes*: 400

Nested-Loop Join

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

  for each tuple t_r in r do
  for each tuple t_s in s do
  if pair (t_r, t_s) satisfy the join condition θ
  add $t_r \bullet t_s$ to the result.

  - r is called the
  - s the
- 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 (Cont.)

- Worst case
  - there is memory only to hold one block of each relation

    $n_r * b_s + b_r$ block transfers + $n_r + b_r$ seeks
- Best case
  - the smaller relation fits entirely in memory: use it as the inner relation
- Example
  - with *student* as outer relation:
  - 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
    - 100 + 400 = 500

Block Nested-Loop Join

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

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 (t_r, t_s) satisfy the join condition
            then add $t_r \bullet t_s$ to the result

- Worst case estimate
  - Each block in the inner relation s is read once for each *block* in the outer relation
  - $\boxed{}$ block transfers + $\boxed{}$ seeks

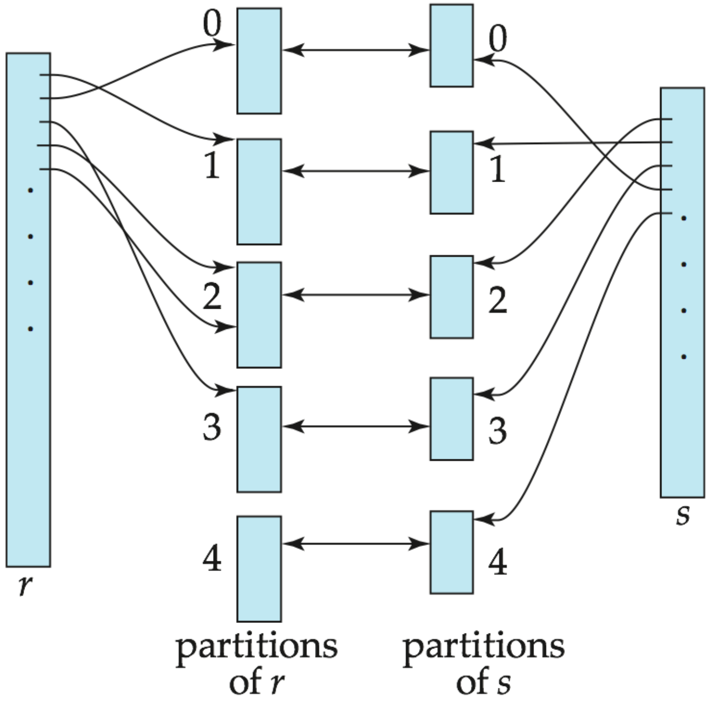
Block Nested-Loop Join (Cont.)

- Best case
  - $\boxed{}$ block transfers + 2 seeks.

- Improvements
  - Use *M*-2 disk blocks as blocking unit for outer relations, and remaining two blocks to buffer inner relation and output
    - Cost = $\boxed{}$
  - If equi-join attribute forms a key of inner relation, stop inner loop on first match
  - Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer (with LRU replacement)

Hash-Join

- Applicable for equi-joins and natural joins.
- A hash function *h* is used to partition tuples of both relations
- *h* maps *JAttrs* values to {0, 1, ..., *n*-1}
  - *JAttrs:* attributes of *r* and *s* used in the equi-join.
  - $r_0, r_1, \ldots, r_{n-1}$: partitions of *r*
    - $t_r \in r$ is put in partition r_i where $i = h(t_r[JAttrs])$.
  - $s_0, s_1, \ldots, s_{n-1}$: partitions of *s*
- *r* tuples in r_i need to be compared with *s* tuples in s_i only



(*Note:* In the textbook, r_i is denoted as H_{ri}, s_i is denoted as H_{si}, and *n* is denoted as n_h.)

Hash-Join Algorithm

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
      - build an in-memory hash index on it using the join attribute (using 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 $\boxed{}$ and
  r is called the $\boxed{}$

Hash-Join algorithm (Cont.)

- The value *n* and the hash function *h* is chosen such that each s_i should fit in memory.
- *Hash-table overflow* occurs in partition s_i if s_i does not fit in memory.
  - Many tuples in *s* with same value for join attributes or bad hash function
  - Overflow resolution can be done in build phase
    - Partition s_i is further partitioned using different hash function.
    - Partition r_i must be similarly partitioned.
  - Fails with large numbers of duplicates
    - Fallback option: use block nested loops join on overflowed partitions

Cost of Hash-Join

- Cost of hash join (without recursive partitioning)
  block transfers ($4n_h$ can be ignored) +
  seeks
  - If the entire build input can be kept in main memory, n can be set to 0 and the algorithm does not partition the relations into temporary files. Cost estimate goes down to $b_r + b_s$.
- Example:
  - *student* $\bowtie$ *takes*
  - memory size: 20 blocks; b_{stud}= 100 and b_{takes} = 400.
  - *student* is build input
  - Partition it into 5 partitions, each of size less than 20 blocks
  - Similarly, partition *takes* into 5 partitions each of size about 80
  - Therefore total cost:

$$3(100 + 400) = 1500 \quad \text{block transfers}$$
$$2(\lceil 100/3 \rceil + \lceil 400/3 \rceil) = 336 \quad \text{seeks}$$

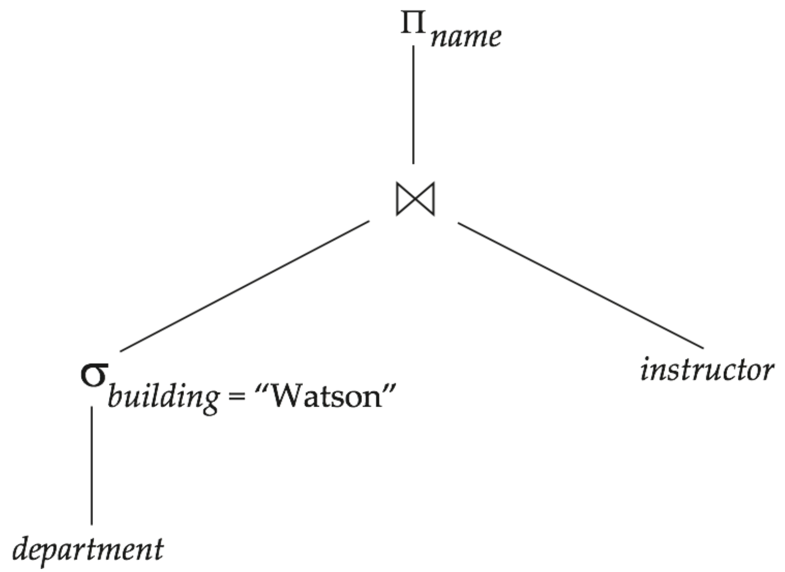
Evaluation of Expressions

- So far, we have seen algorithms for individual operations
- How do you evaluate an entire expression tree?
-
  - generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.
-
  - pass on tuples to parent operations even as an operation is being executed

Materialization

- Evaluate one operation at a time, starting at the lowest-level.
- Use intermediate results *materialized into temporary relations* to evaluate next-level operations.

- E.g.,
  - compute and store $\sigma_{building=\text{"Watson"}}(department)$
  - then compute and store its join with *instructor*
  - and finally compute the projections on *name.*



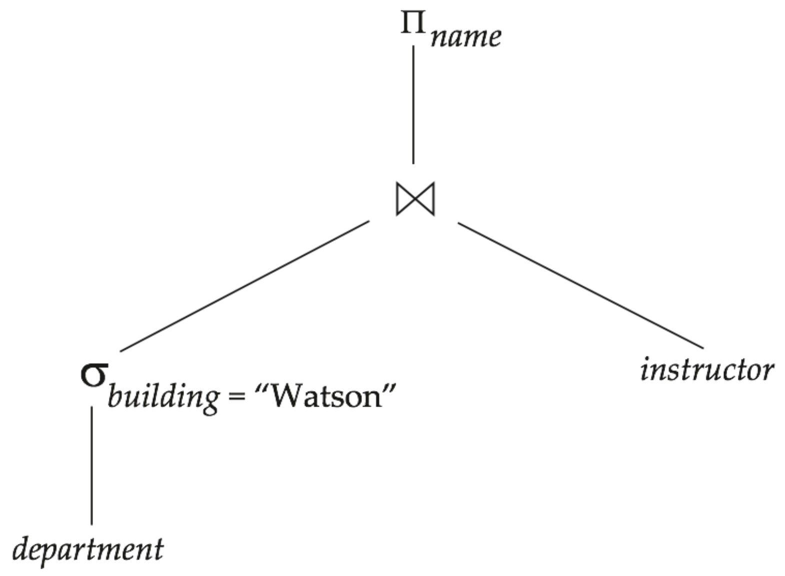
Materialization (Cont.)

- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing results to disk
  - Overall cost = Sum of costs of individual operations +

Pipelining

- Evaluate several operations simultaneously
  - passing the results of one operation on to the next.

- E.g., in expression tree
  - don’t store result of selection
  - instead, pass tuples directly to the join
  - Similarly, don’t store result of join but pass tuples directly to projection



Pipelining (cont.)

- Much cheaper than materialization
  - no need to store a temporary relation to disk.
- Pipelining may not always be possible
  - e.g., sort, hash-join: $\boxed{}$
  - Very difficult to achieve a lengthy chain of pipeline
- For pipelining to be effective
  - use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation

Cost Estimation of Expressions

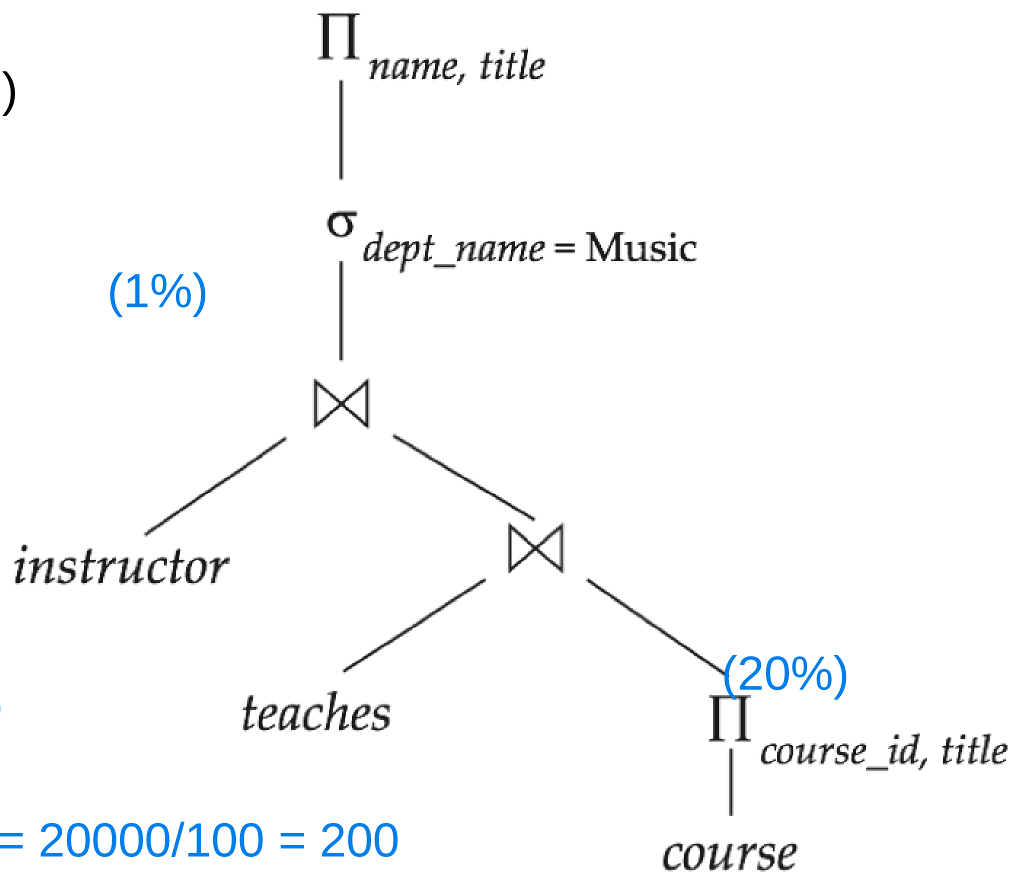
- Cost of computing each operator is as described
  - Need statistics of input relations
  - E.g. number of tuples, sizes of tuples
- Inputs can be results of sub-expressions
  -
  - Additional statistics are needed
    - number of distinct values for an attribute, histograms, …

Cost Estimation of Expressions

$\Pi_{name,\ title}(\sigma_{dept=\text{“Music”} \wedge year\ =\ 2009}$
$(instructor \bowtie (teaches \bowtie \Pi_{c\_id,\ title}\ (course))))$

M = 22

N: $\lceil b_r / (M\text{-}2) \rceil * b_s + b_r$
H: $3(b_r + b_s) + 4 * n_h$



$b_{instructor} = 2000/25 = 80$

$b_{teaches} = 20000/100 = 200$

$b_{course} = 10000/25 = 400$

(a) Initial expression tree

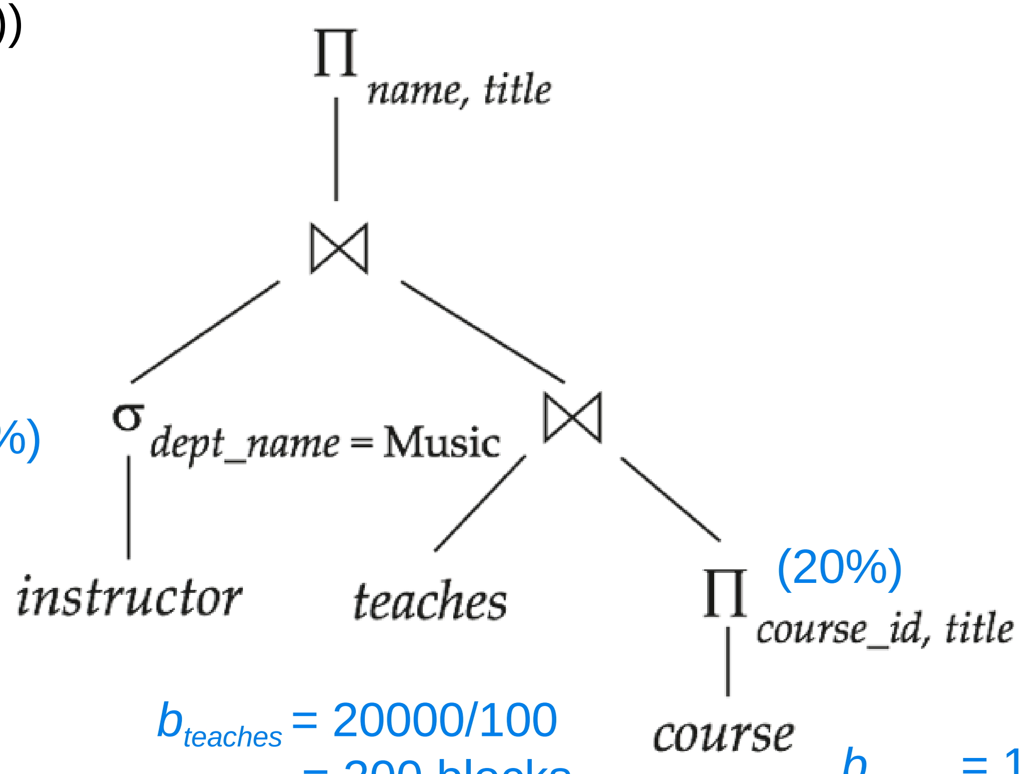
Cost Estimation of Expressions

$$\Pi_{name,\ title}(\sigma_{dept=\text{“Music”} \wedge year\ =\ 2009}$$
$$((instructor \bowtie teaches) \bowtie \Pi_{c\_id,\ title}\ (course))))$$

M = 22

N: $\lceil b_r / (M\text{-}2) \rceil * b_s + b_r$

H: $3(b_r + b_s) + 4 * n_h$



(1%)

(20%)

$b_{instructor}$ = 1000/25 = 80 blocks

$b_{teaches}$ = 20000/100 = 200 blocks

b_{course} = 10000/25 = 400 blocks

(b) Transformed expression tree

END OF CHAPTER 12