Chapter 12: Query Processing

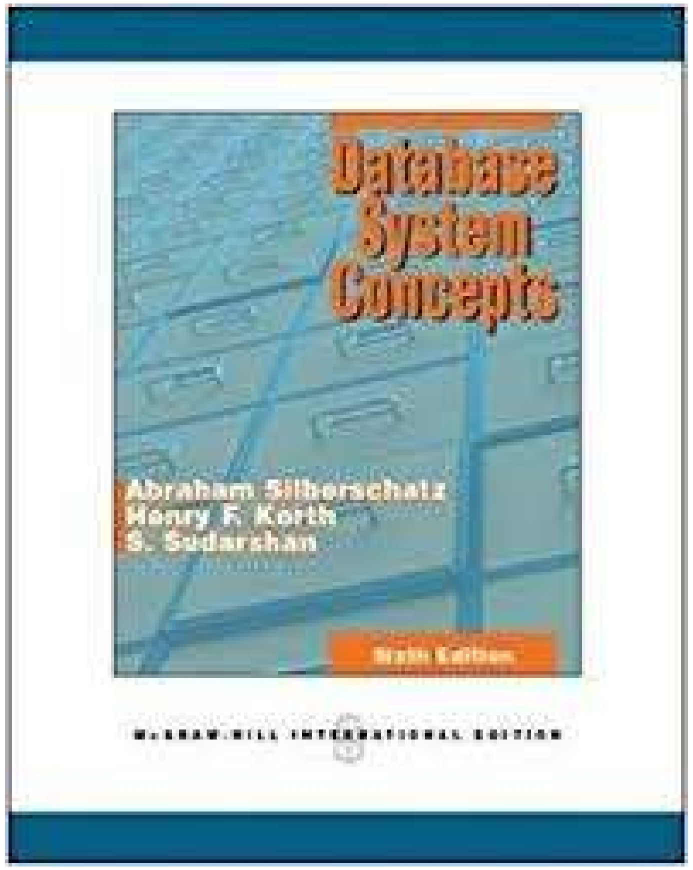
Data Management for Big Data

2018-2019 (spring semester)

Dario Della Monica

Chapter 12: Query Processing

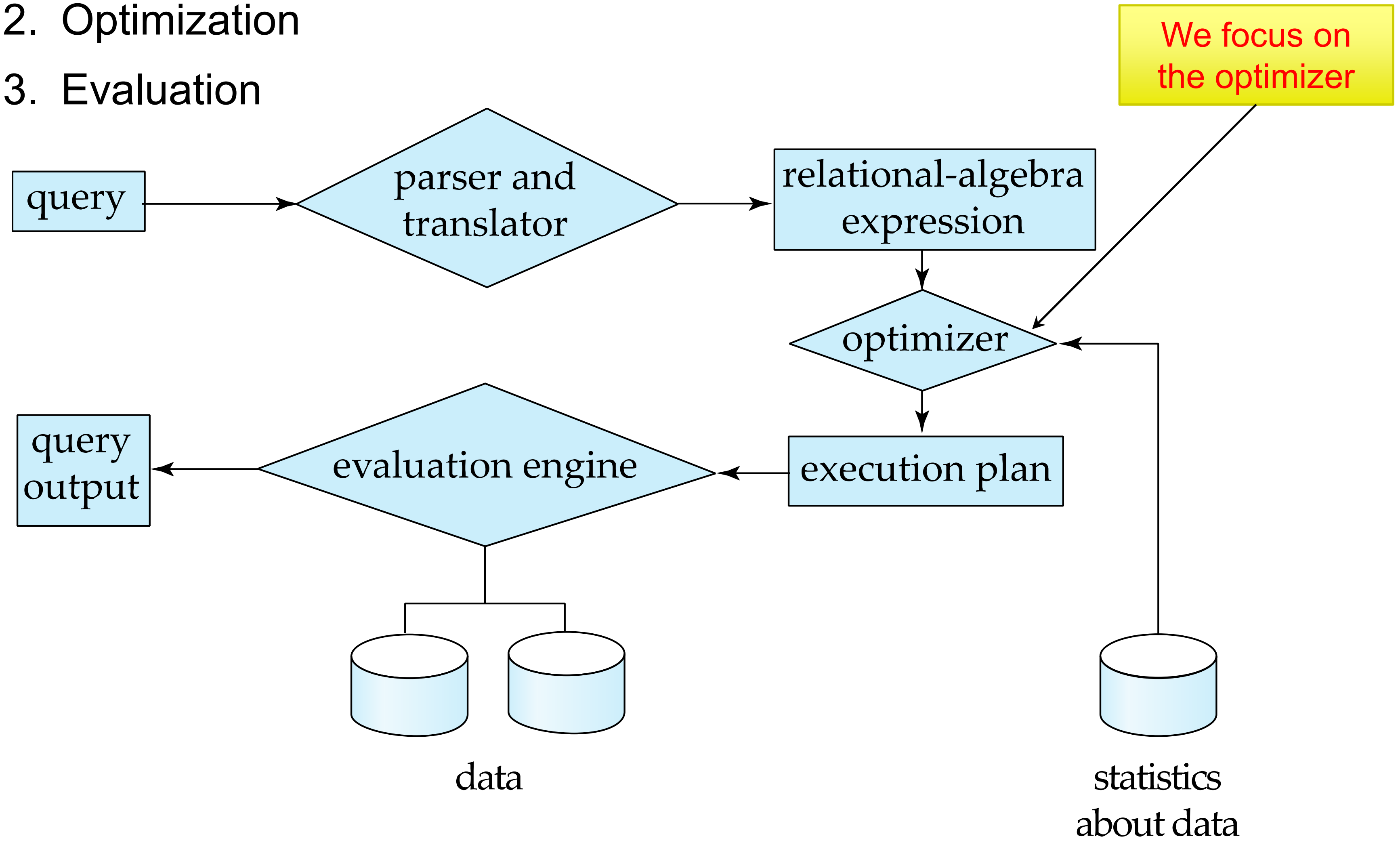
- Overview
- How to measure query costs
- Algorithms for evaluating relational algebra operations
  - Selection Operation
  - Sorting
  - Join Operation
- Evaluation of Expressions
  (How to combine algorithms for individual operations in order to evaluate a complex expression)
  - Materialization
  - Pipelining


Database System Concepts
Abraham Silberschatz
Henry F. Korth
S. Sudarshan
Sixth Edition
McGRAW-HILL INTERNATIONAL EDITION

Silberschatz, Korth, Sudarshan, *Database System Concepts*, 6° edition, 2011

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Basic Steps in Query Processing (cont.)

- Parser and translator
  - translate the (SQL) query into relational algebra
  - Parser checks syntax, verifies relations
- Evaluation engine
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query
- Optimizer (in a nutshell -- more details in the next slides)
  - Chooses the most efficient implementation to execute the query
    - Produces equivalent relational algebra expressions
    - Annotates them with instructions (algorithms)

Basic Steps: Optimization

- 1st level of optimization: an SQL query has many equivalent relational algebra expressions
  - $\sigma_{salary<75000}(\Pi_{salary}(instructor))$ *and*
    $\Pi_{salary}(\sigma_{salary<75000}(instructor))$ are equivalent
  - They both correspond to SELECT salary
    FROM instructor
    WHERE salary < 75000
- 2nd level of optimization: a relational algebra operation can be evaluated using one of several different algorithms
  - Selection: file scan VS. indices
- Output of optimization: annotated relational algebra expression specifying detailed evaluation strategy (**query evaluation plan or query execution plan - QEP**)

Basic Steps: Optimization (Cont.)

- Different query evaluation plans have different costs
  - User is not expected to specify least-cost plans
- **Query Optimization:** amongst all equivalent evaluation-plans choose the one with lowest cost.
  - Cost is estimated using statistical information from the database catalog
    - # of tuples in relations, tuple sizes, # of distinct values for a given attribute, etc.
- We study… (Chapter 12[★] – evaluation of QEP)
  - How to measure query costs
  - Algorithms for evaluating relational algebra operations
  - How to combine algorithms for individual operations in order to evaluate a complex expression (QEP)
- … and (Chapter 13[★] – choosing the best QEP)
  - How to optimize queries, that is, how to find a query evaluation plan with lowest estimated cost

[★] **Silberschatz, Korth, and Sudarshan, *Database System Concepts*, 6° ed.**

How to measure query costs

Database System Concepts, 6th Ed.

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* for distributed DBMS – later in this course)
- Typically disk access is the predominant cost, and is also relatively easy to estimate.   Measured by taking into account

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
      - data is read back after being written to ensure that the write was successful

During the whole optimization process, optimizers can make different assumptions (e.g., indices are always stored in in-memory buffer, etc.)
To be applied to concrete systems, our analysis should be adapted according to system features

Measures of Query Cost (Cont.)

- For the sake of simplicity, we ignore difference between reading and writing a block and thus we just use
  - **number of seeks** (t_T – time for one seek)
  - **number of block transfers** (t_T – time to transfer one block)
  - Example: cost for ***b*** block transfers plus ***S*** seeks
    $b * t_T + S * t_S$
  - Values of t_T and t_S must be calibrated for the specific disk system
  - Typical values (2011): t_S = 4 ms, t_T = 0.1 ms
  - Some DBMS performs, during installation, seeks and block transfers to estimate average values
- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account
- We do not include cost to writing output to disk in our cost formulae

Measures of Query Cost (Cont.)

- Response time of a QEP is hard to estimate without actually executing the plan because some runtime information is needed
  - the content of the buffer when the execution begins
  - parameter embedded in query which are resolved at runtime only

    ```
    SELECT salary
    FROM instructor
    WHERE salary < $a
    ```

    where $a is a variable provided by the application (user)
- Several algorithms can reduce disk IO by using extra buffer space
  - Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
    - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available (e.g., 1 block per relation)

Algorithms for evaluating relational algebra operations

Database System Concepts, 6th Ed.

Selection Operation

- **File scan**

  PROs: can be applied to any file, regardless of its ordering, availability of indices, nature of selection operation, etc.

  CONs: it is slow

- Algorithm **A1** (**linear search**). Scan each file block and test all records to see whether they satisfy the selection condition
  - b_r denotes number of blocks containing records from relation r
  - Cost estimate = b_r block transfers + 1 seek $= t_S + b_r * t_T$

We assume blocks are stored contiguously so 1 seek operation is enough (disk head does not need to move to seek next block

  - If selection is on a key attribute, can stop on finding record
    - cost = $(b_r/2)$ block transfers + 1 seek $= t_S + (b_r / 2) * t_T$

Selections Using Indices

- **Index scan:** search algorithms that use an index
  - selection condition must be on search-key of index
  - ***h_i* :** height of the B^+-tree (# of accesses to traverse the index before accessing the data)
- **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
  - Let b = number of blocks containing matching records
  - Records will be on consecutive blocks
  - $Cost = h_i * (t_T + t_S) + t_S + t_T * b$

There is a mistake in the book* (Fig. 12.3): the "t_S" summand is omitted

* **Silberschatz, Korth, and Sudarshan, *Database System Concepts*, 6° ed.**

Selections Using Indices

- **A4** (**secondary index, equality on key**)
  - Equal to **A2**
    - *Cost* = $(h_i + 1) * (t_T + t_S)$

- **A4** (**secondary index, equality on nonkey**)
  - Retrieve multiple records
    - each of *n* matching records may be on a different block
    - Cost = $(h_i + n) * (t_T + t_S)$
      - Can be very expensive! Can be worse than file scan

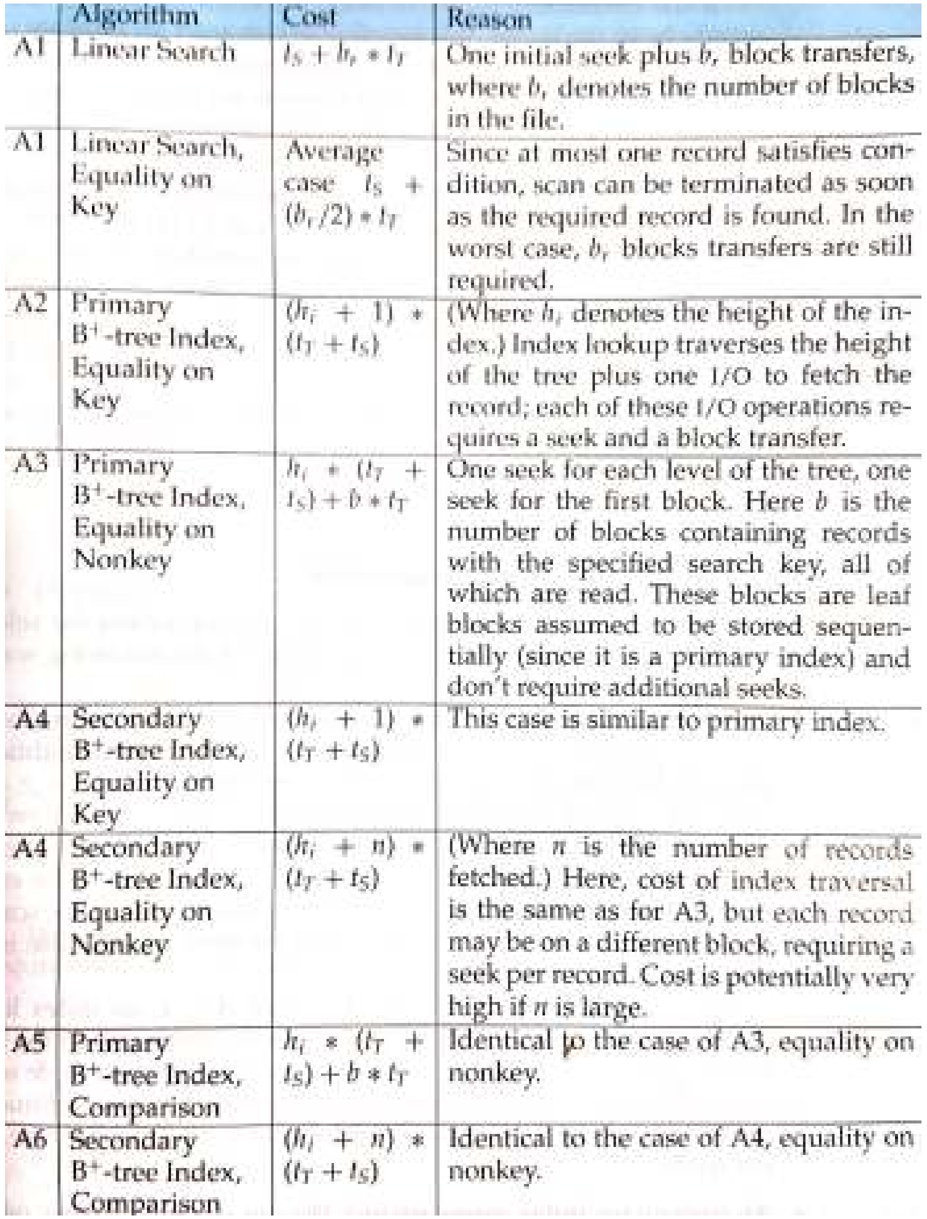
Selections Involving Comparisons

- 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:
- **A5** (**primary index, comparison**).
  - For $\sigma_{A \ge V}(r)$ use index to find first tuple $\ge$ v and scan relation sequentially from there
    - *b* is the number of blocks containing matching records
    - Equal to **A3**: *Cost* = $h_i * (t_T + t_S) + t_S + t_T * b$
  - For $\sigma_{A \le V}(r)$ just scan relation sequentially till first tuple > v; do not use the index
    - Equal to **A1**, equality on key: *Cost* = $t_S + (b_r / 2) * t_T$

Selections Involving Comparisons

- **A6** (**secondary index, comparison**).
  - For $\sigma_{A \geq V}(r)$ use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \leq V}(r)$ 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
    - Equal to **A4**, equality on nonkey: $Cost = (h_i + n) * (t_T + t_S)$
    - Linear file scan may be cheaper

Summary of costs for selections



	Algorithm	Cost	Reason
A1	Linear Search	$t_S + b_r * t_T$	One initial seek plus b_r block transfers, where b_r denotes the number of blocks in the file.
A1	Linear Search, Equality on Key	Average case $t_S + (b_r/2) * t_T$	Since at most one record satisfies condition, scan can be terminated as soon as the required record is found. In the worst case, b_r blocks transfers are still required.
A2	Primary B$^+$-tree Index, Equality on Key	$(h_i + 1) * (t_T + t_S)$	(Where h_i denotes the height of the index.) Index lookup traverses the height of the tree plus one I/O to fetch the record; each of these I/O operations requires a seek and a block transfer.
A3	Primary B$^+$-tree Index, Equality on Nonkey	$h_i * (t_T + t_S) + b * t_T$	One seek for each level of the tree, one seek for the first block. Here b is the number of blocks containing records with the specified search key, all of which are read. These blocks are leaf blocks assumed to be stored sequentially (since it is a primary index) and don't require additional seeks.
A4	Secondary B$^+$-tree Index, Equality on Key	$(h_i + 1) * (t_T + t_S)$	This case is similar to primary index.
A4	Secondary B$^+$-tree Index, Equality on Nonkey	$(h_i + n) * (t_T + t_S)$	(Where n is the number of records fetched.) Here, cost of index traversal is the same as for A3, but each record may be on a different block, requiring a seek per record. Cost is potentially very high if n is large.
A5	Primary B$^+$-tree Index, Comparison	$h_i * (t_T + t_S) + b * t_T$	Identical to the case of A3, equality on nonkey.
A6	Secondary B$^+$-tree Index, Comparison	$(h_i + n) * (t_T + t_S)$	Identical to the case of A4, equality on nonkey.

Sorting

- Reasons for sorting
  - Explicitly requested by SQL query
    - SELECT …
      FROM …
      SORT BY …
  - Needed to efficient executions of join operations
- We may build an index on the relation, and then use the index to read the relation in sorted order. May lead to one disk block access for each tuple
- For relations that fit in memory, standard sorting techniques like quick-sort can be used. For relations that don't fit in memory, **external sort-merge** algorithm is a good choice

External Sort-Merge

Let M denote number of blocks that can fit in memory.

1. **Create sorted runs** (files containing sorted pieces of relation)

   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

   (d) Increment i

   Let the final value of i be N *(number of runs)*

2. *Merge the runs (next slide)…..*

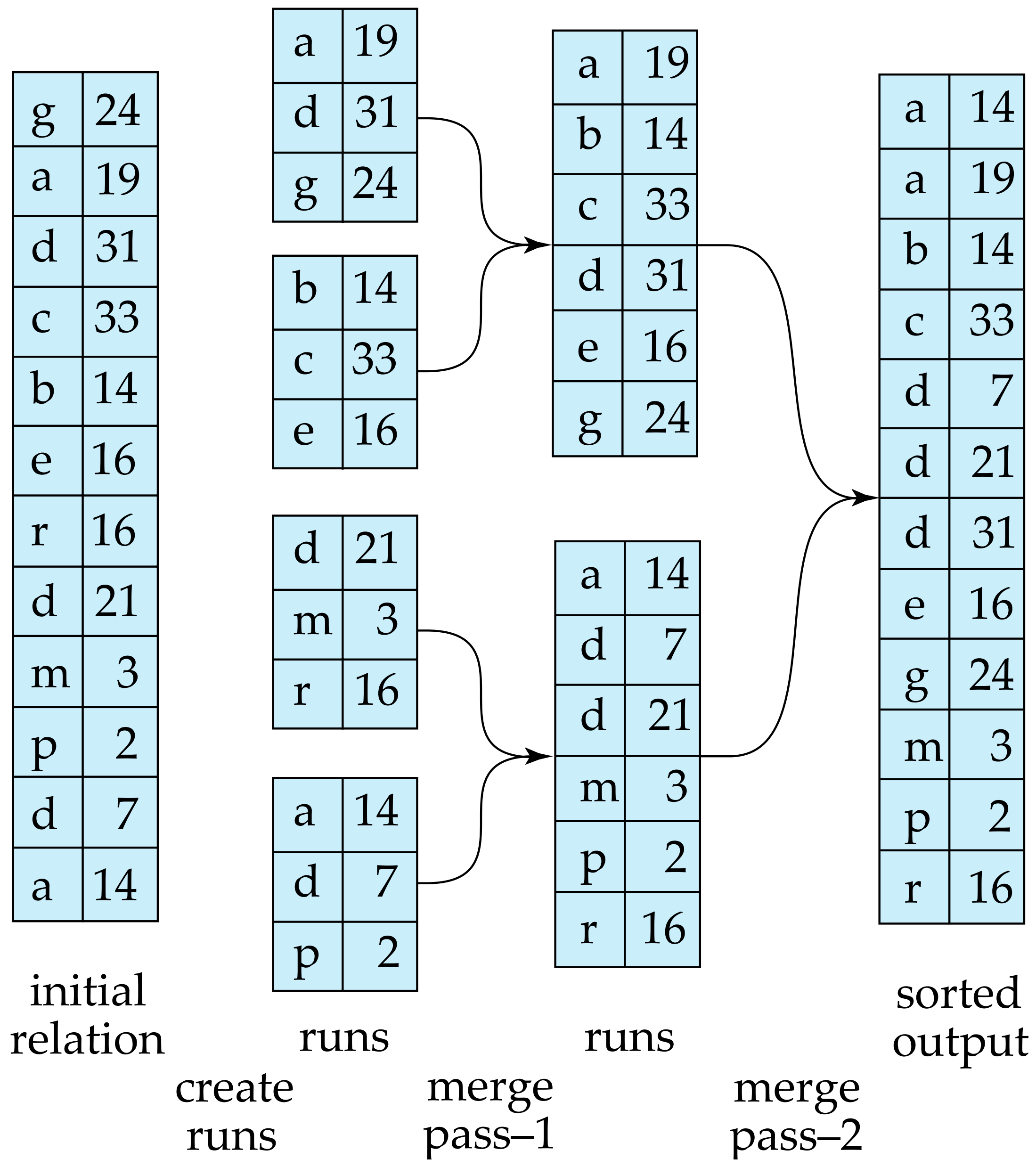
External Sort-Merge (Cont.)

2. **Merge the runs (N-way merge).** We assume (for now) that $N < M$.
   1. Use N blocks of memory to buffer input runs, and 1 block to buffer 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 the next block (if any) of the run into the buffer.
   3. **until** all input buffer pages are empty:

External Sort-Merge (Cont.)

- 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.

Example: External Sorting Using Sort-Merge



External Sort-Merge: Cost Analysis

- Cost of block transfers:
  - 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 is $2b_r$
    - for final pass, we don't count write cost
      - we ignore final write cost for all operations since the output of an operation may be sent to the parent operation without being written to disk
    - Thus total number of block transfers for external sorting:

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

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

  - Seeks: next slide

External Sort-Merge: Cost Analysis (cont.)

- 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 $2\ b_r$ seeks for each merge pass
      - except the final one which does not require a write
    - Total number of seeks:

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

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

External Sort-Merge: Cost Analysis (cont.)

An improved version of the algorithm [*]

- Number of seeks can be reduced by using $\boldsymbol{b_b}$ many blocks (instead of 1) for each run during the run merge phase
  - Using 1 block per run leads to too many seeks
  - Instead, using b_b buffer blocks per run ➔ read/write b_b blocks with only 1 seek
    - Merge together: $\lfloor \boldsymbol{M/b_b} \rfloor - \boldsymbol{1}$ runs (instead of $M - 1$)
    - Number of passes required: $\lceil \boldsymbol{\log_{\lfloor M/b_b \rfloor - 1}(b_r / M)} \rceil$ instead of $\lceil \log_{M-1}(b_r / M) \rceil$
    - During the merge phase: $\boldsymbol{2 \lceil b_r / b_b \rceil}$ seeks for each pass (instead of $2\, b_r$)
      - Except the final one (we assume final result is not written on disk)
- Thus total number of block transfers for external sorting:

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

Total number of seeks:

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

[*] In Silberschatz, Korth, and Sudarshan, Database System Concepts, 6° ed., the non-improved version of the algorithm is given only, but the cost analysis mixes elements from both versions of the algorithm

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
- Example of join used in the cost analysis: *students* $\bowtie$ *takes*

  where

Number of *records* of	*student*:	5,000
Number of *blocks* of	*student*:	100
Number of *records* of	*takes*:	10,000
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 $\theta$
     if they do, add $t_r \cdot t_s$ 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 (Cont.)

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is

  # of block transfer: $n_r * b_s + b_r$

  (b_r transfers to read relation r + $n_r * b_s$ transfers to read s for each tuple in r)

  # of seeks: $n_r + b_r$

  (b_r seeks to read relation r + n_r seeks to read s for each tuple in r)

- 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

  (same cost in the best case scenario, when both relations fit in memory)
- Assuming worst case memory availability cost estimate is
  - 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-loops algorithm (next slide) is preferable

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 $t_r \bullet t_s$ to the result.
            end
        end
    end
end
```

Block Nested-Loop Join (Cont.)

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

How to combine algorithms for individual operations in order to evaluate a complex expression

Evaluation of Expressions

- So far: we have seen algorithms for individual operations
- Alternatives for evaluating an entire expression tree
  - **Materialization**: store (**materialize**) on disk results of evaluation of sub-expressions into temporary relations for subsequent use
    - Disadvantage: several disk writing to store temporary relations
    - Always possible
  - **Pipelining**: pass on tuples to parent operations as they are generated by inner operations being executed
    - Advantage: less disk writing
    - Not always possible

Materialization

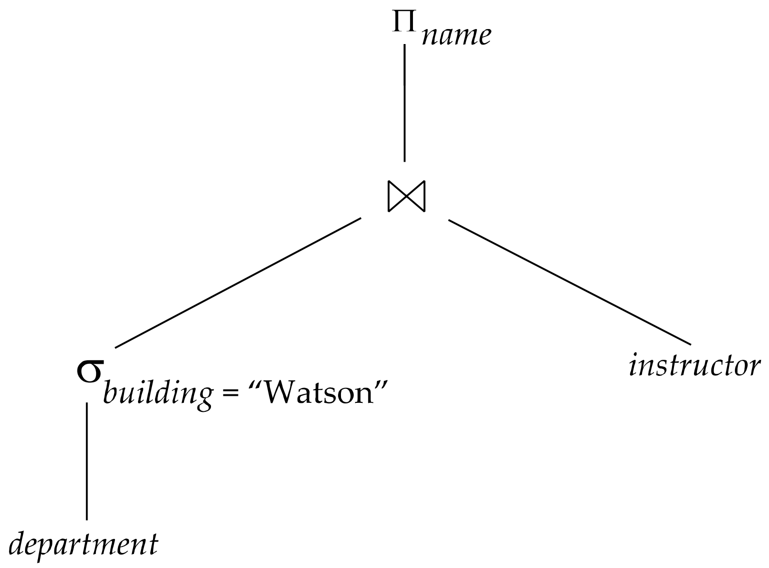
- **Materialized evaluation**: evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations

E.g., $\prod_{name}(\sigma_{building="\text{Watson}"}(department) \bowtie \text{instructor})$

1. compute and store $\sigma_{building="\text{Watson}"}(department)$

2. then compute and store its join with *instructor*

3. finally, compute the projection on *name*



Pipelining

- **Pipelined evaluation:** evaluate several operations simultaneously, passing the results of one operation on to the next and without writing on disk intermediate results
- E.g., in previous expression tree, don't store result of

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

  Instead, pass tuples directly to the join as they are found

  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., some sorting algorithms
  (it cannot output tuples early, only after all tuples have been examined)

End of Chapter

Figure 12.02

$$\begin{tikzcd}
\pi\ \textit{salary} \arrow[d, dash] \\
\sigma\ \textit{salary} < 75000\text{; use index 1} \arrow[d, dash] \\
\textit{instructor}
\end{tikzcd}$$

Selection Operation (Cont.)

- **Old-A2** *(binary search).* Applicable if selection is an equality comparison on the attribute on which file is ordered.
  - Assume that the blocks of a relation are stored contiguously
  - Cost estimate (number of disk blocks to be scanned):
    - cost of locating the first tuple by a binary search on the blocks
      - $\lceil \log_2(b_r) \rceil * (t_T + t_S)$
    - If there are multiple records satisfying selection
      - *Add transfer cost of the* number of blocks containing records that satisfy selection condition
      - Will see how to estimate this cost in Chapter 13