Chapter 12: Query Processing

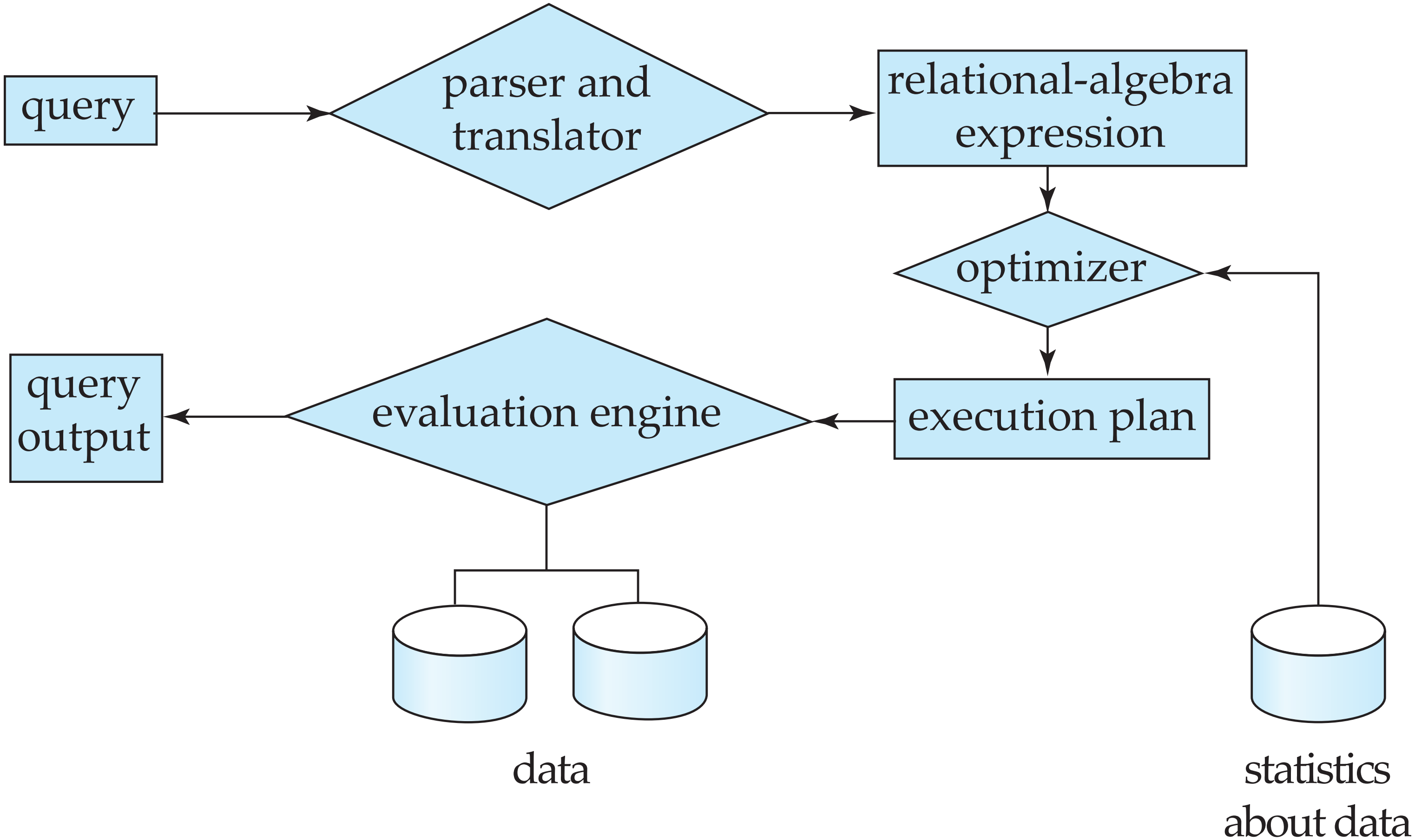
Database System Concepts, 6th Ed.

Chapter 12: Query Processing

- Overview of query processing and optimisation
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions
- Intraquery parallelism (in chapter 18 of the book)

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Basic Steps in Query Processing (Cont.)

- Parsing and translation
  - Translate the query into its internal form.
  - This is then translated into relational algebra.
    - (Extended) relational algebra is more compact, and differentiates clearly among the various different operations
  - Parser checks syntax, verifies relations
  - This is a subject for *compilers* that we will ignore here
- Evaluation
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.
    - The bulk of the problem lies in how to come up with a good evaluation plan!
    - Query execution is "simply" executing a predefined plan (or program)

Evaluation plan example

π_{salary}

$\sigma_{salary < 75000\text{; use index 1}}$

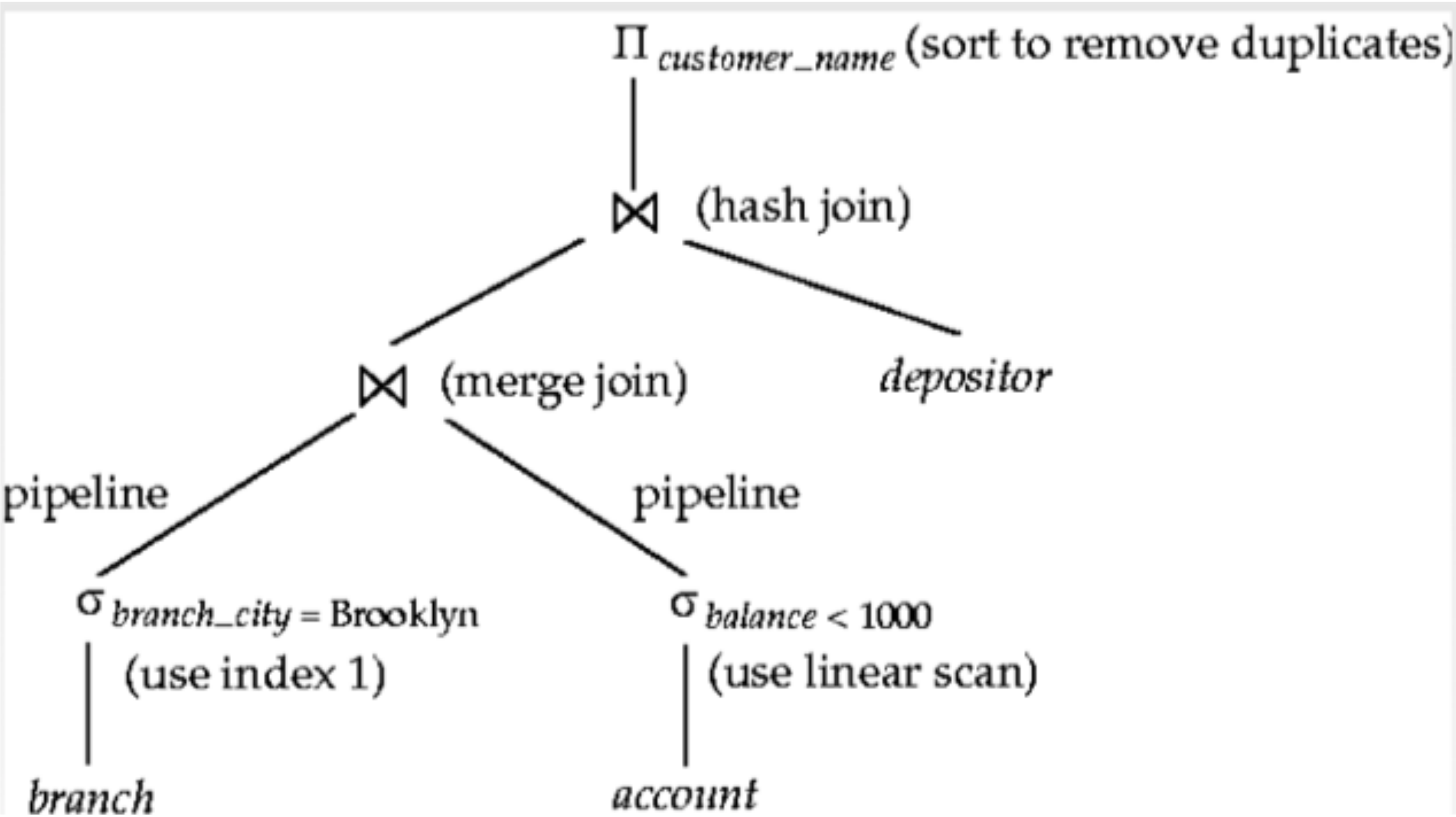
instructor

Evaluation primitive

Basic Steps in Query Processing : Optimization

- A relational algebra expression may have many equivalent expressions
  - E.g., $\sigma_{salary<75000}(\prod_{salary}(instructor))$ is equivalent to $\prod_{salary}(\sigma_{salary<75000}(instructor))$
- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.
  - E.g., can use an index on *salary* to find instructors with salary < 75000,
  - or can perform complete relation scan and discard instructors with salary ≥ 75000

A more complex evaluation-plan



Basic Steps: Optimization (Cont.)

- **Query Optimization**: Amongst all equivalent evaluation plans choose the one with lowest cost.
  - Cost is estimated using statistical information from the database catalog
    - e.g. number of tuples in each relation, size of tuples, etc.
- In this chapter we study
  - How to measure query costs (to have a measure to be able to evaluate and compare the various plans and algorithms)
  - Algorithms for evaluating (main) relational algebra operations
  - How to combine algorithms for individual operations in order to evaluate a complete expression
  - How these algorithms and combinations can be parallelised
- Later we will study how to optimize queries, that is, how to find an evaluation plan with lowest estimated cost

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*
- 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
    - The cost of a seek is usually much higher than that of a block transfer read or write (one order of magnitude)

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
  - t_T – time to transfer one block
    (0.1 ms for 4Kb blocks and 40 Mb/s transfer rate)
  - t_S – time for one seek (high-end disks 4 ms)
  - Cost for b block transfers plus S seeks
    $$b * t_T + S * t_S$$
- We do not include cost to writing output to disk in the cost formulae
- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account, but they are clearly less significant
- Evaluating the cost of an algorithm for query processing is similar to the ones learnt in "Algorithms and Data Structures" but here the measures are quite different:
  - the evaluation in terms of block transfers and seeks are substantially different than in terms of number of execution steps.

Measures of Query Cost (Cont.)

- 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
- Required data may be buffer resident already, avoiding disk I/O
  - But hard to take into account for cost estimation

Selection Operation (recall)

- Notation: $\sigma_{p}(r)$
  - p is the selection predicate
  - Defined by $\sigma_p(\boldsymbol{r}) = \{t \mid t \in r \textbf{ and } p(t)\}$
  - in which p is a formula of propositional calculus of terms connected by: $\wedge$ (and), $\vee$ (**or**), $\neg$ (**not**)
    Each term is of the form:
  - <attribute> *op* <attribute> or <constant>
  - where *op* can be one of: $=, \neq, >, \geq. <. \leq$
- Selection example:
  $\sigma_{\textit{branch-name='Perryridge'}}(\textit{account})$
- For recalling other operators, see documentation of "Bases de Dados".

Selection Operation

- **File scan** – search algorithms that locate and retrieve records that fulfill a selection condition
- Algorithm **A1** (**linear search**). Scan each file block and test all records to see whether they satisfy the selection condition.
  - Cost estimate = b_r block transfers + 1 seek
    - b_r denotes number of blocks containing records from relation r
  - If selection is on a key attribute, can stop on finding record
    - Average cost = $(b_r/2)$ block transfers + 1 seek
  - Linear search can be applied regardless of
    - selection condition or
    - ordering of records in the file, or
    - availability of indices

Binary search

- Binary search generally does not make sense since data is not stored consecutively except when there is an index available, but binary search requires more seeks than index search
- Applicable only if the selection is an equality comparison on the attribute on which file is ordered.
- Assuming that the blocks of a relation are stored contiguously, the 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
- If b_r is not too big, then most likely binary search doesn't pay.
  - Note that t_S is several (say, 50) times bigger than t_T
- Estimates on the size of the relation are needed to wisely choose which of the two algorithms is better for a specific query at hands.

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, with h_i the index height
  - $Cost = \underbrace{h_i * (t_T + t_S)}_{\text{Index search}} + \underbrace{(t_T + t_S)}_{\text{Record retrieval}} = (h_i + 1) * (t_T + t_S)$
- The height of a B+-tree is $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$, where n is the number of index entries per node and K is the number of search keys. Unless the relation is small, this algorithms "pays off" when indexes are available
  - E.g. for a relation r with 1.000.000 different search keys, and with 100 index entries per node, $h_i = 4$. Usually root node is in memory.
- **A3** (**primary index, equality on nonkey**) Retrieve multiple records.
  - Records will be on consecutive blocks
    - Let b = number of blocks containing matching records
  - $Cost = h_i * (t_T + t_S) + t_S + t_T * \text{b}$

Selections Using Indices

- **A4** (**secondary index, equality on nonkey**).
  - Retrieve a single record if the search-key is a candidate key
    - *Cost* $= (h_i + 1) * (t_T + t_S)$
  - Retrieve multiple records if search-key is not a candidate key
    - each of *n* matching records may be on a different block
    - Cost $= (h_i + n) * (t_T + t_S)$
      - Can be very expensive!

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**). (Relation is sorted on A)
  - For $\sigma_{A \ge V}(r)$ use index to find first tuple $\ge v$ and scan relation sequentially from there
  - For $\sigma_{A \le V}(r)$ just scan relation sequentially till first tuple $> v$; do not use index since it would require extra seeks on the index file
- **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
    - In worst-case requires an I/O for each record (a lot!)
    - Linear file scan may be cheaper!!!!

Implementation of Complex Selections

- **Conjunction:** $\sigma_{\theta 1 \wedge\ \theta 2 \wedge . . .\ \theta n}(r)$
- **A7** (**conjunctive selection using one index**).
  - Select a combination of θ_i and algorithms A1 through A6 that results in the least cost for $\sigma_{\theta i}\ (r)$.
  - Test other conditions on tuple after fetching it into memory buffer.
  - In this case the choice of the first condition is crucial!
    - One must use estimates to figure out which one is better.
- **A8** (**conjunctive selection using composite index**).
  - Use appropriate composite (multiple-key) index if available.
- **A9** (**conjunctive selection by intersection of identifiers***).*
  - Requires indices with record pointers (*rowids*).
  - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers.
  - Then fetch records from file
  - If some conditions do not have appropriate indices, apply test in memory.

Algorithms for Complex Selections

- **Disjunction:** $\sigma_{\theta 1} \vee_{\theta 2} \vee \ldots_{\theta n} (r)$.
- **A10** (**disjunctive selection by union of identifiers**).
  - Applicable if *all* conditions have available indices.
    - Otherwise use linear scan.
  - Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
  - Then fetch records from file
- **Negation:** $\sigma_{\neg\theta}(r)$
  - Use linear scan on file
  - If very few records satisfy $\neg\theta$, and an index is applicable to θ
    - Find satisfying records using index and fetch from file

Sorting

- Sorting algorithms are important in query processing at least for two reasons:
  - The query itself may require sorting (**order by** clause)
  - Some algorithms for other operations, like projection, join, set operations and aggregation, require previously sorted relations
- To sort a relation:
  - We may build an index on the relation, and then use the index to read the relation in sorted order.
    - This only sorts the relation logically, not physically
    - May lead to one disk block access for each tuple.
  - For relations that fit in memory sorting algorithms that you've studied before, like quicksort, can be used.
  - For relations that don't fit in memory special algorithms are required, that take into account the measures in terms of disc transfers and seeks. **External sort-merge** is a good choice.

External Sort-Merge

Let M denote memory size (in pages/blocks).

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

2. *Merge the runs (next slide)…..*

External Sort-Merge (Cont.)

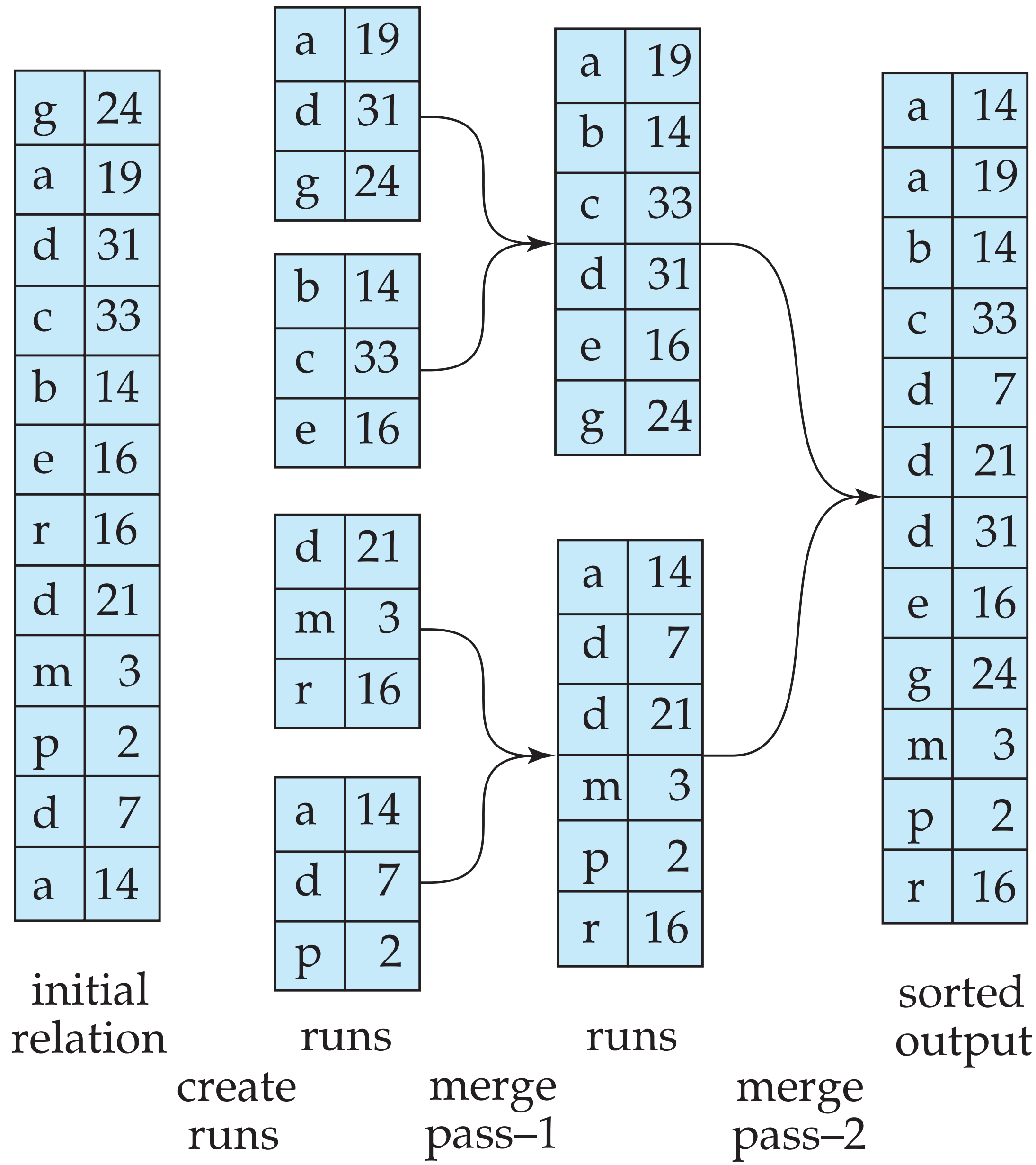
2. **Merge the runs (N-way merge).** We assume (for now) that $N < M$.
    1. Use N blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page
    2. **repeat**
        1. Select the first record (in sort order) among all buffer pages
        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.
- Note that, in practice, this is only required fore really huge relations:
  - Consider 4Gb memory and 4Kb blocks (i.e. 1M blocks fit in memory)
  - For a 2nd pass to be needed, there should be over 1M runs, i.e. 4000Tb (since each run can be circa 4Gb).

Example: External Sorting Using Sort-Merge

M=3



External Merge Sort (Cont.)

- Cost analysis (as corrected in the ERRATA):
  - 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
    - Can merge $\lfloor M/b_b \rfloor -1$ runs in one pass
  - Total number of merge passes required: $\lceil \log_{\lfloor M/bb \rfloor -1}(b_r/M) \rceil$.
  - 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:

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

  - Seeks: next slide

External Merge Sort (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 $2 \lceil b_r / b_b \rceil$ seeks for each merge pass
      - except the final one which does not require a write
    - Total number of seeks:

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

$$=$$

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

Join Operation

- Several different algorithms to implement joins, ignoring for the time being the parallel ones:
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- As for selection, choice based on cost estimate
- Examples use the following information
  - Number of records of *student*: 5,000 *takes*: 10,000
  - Number of blocks of *student*: 100 *takes*: 400

Nested-Loop Join

- The simplest algorithm that can be used always (like linear search for selection)
- 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 $t_r \cdot t_s$ 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.
- Quite expensive in general, 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

  $n_r * b_s + b_r$ block transfers, plus

  $n_r + b_r$ seeks

- In general, it is much better to have the smaller relation as the outer relation
  - The number of block transfers is multiplied by the number of tuples of the outer relation
  - The number of seeks only depends on the outer relation
- However, if the smaller relation is small enough to fit in memory, one should use it as the inner relation!
  - Reduces cost to $b_r + b_s$ block transfers and 2 seeks
- The choice of the inner and outer relation strongly depends on the estimate of the size (cardinality) of each relation

Nested-Loop Join (Example)

- 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.
- Instead of iterating over records, one could iterate over blocks. This way instead of $n_r * b_s + b_r$ we would have $b_r * b_s + b_r$ block transfers
- This is the basis of the usually preferable block nested-loop join algorithm (details in the next slide)

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 $t_r \cdot t_s$ to the result.
            end
        end
    end
end

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 (when smaller relation fits into memory):
  $b_r + b_s$ 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
    - Cost = $\lceil b_r / (M\text{-}2) \rceil * b_s + b_r$ block transfers $+ 2 \lceil b_r / (M\text{-}2) \rceil$ seeks
  - If equi-join attribute forms a key or 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)
  - Use index on inner relation if available to faster obtain the tuples that match the current tuple of the outer relation

Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - 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 block of *r*, and, for each tuple in *r*, we perform an index lookup on *s.*
- Cost of the join: $b_r\,(t_T + t_S) + n_r * c$
  - Where *c* is the cost of traversing index and fetching all matching *s* tuples for one tuple or *r*
  - *c* can be estimated as cost of a single selection on *s* using the join condition
- If indices are available on join attributes of both *r* and *s,* use the relation with fewer tuples as the outer relation.

Example of Nested-Loop Join Costs

- Compute *student* $\bowtie$ *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-loop join
  - 400*100 + 100 = 40,100 block transfers + 2 * 100 = 200 seeks (4.81 secs)
    - assuming worst case memory
    - may be significantly less with more memory
- Cost of indexed nested-loop join
  - 100 + 5000 * 5 = 25,100 block transfers and seeks (102,91 secs)
  - CPU cost likely to be less than that for block nested loops join
  - However in terms of time for transfers and seeks, in this case using the index does not pay (this is so because the relations are small)