Chapter 15: Query Processing

Sistemas de Bases de Dados 2019/20

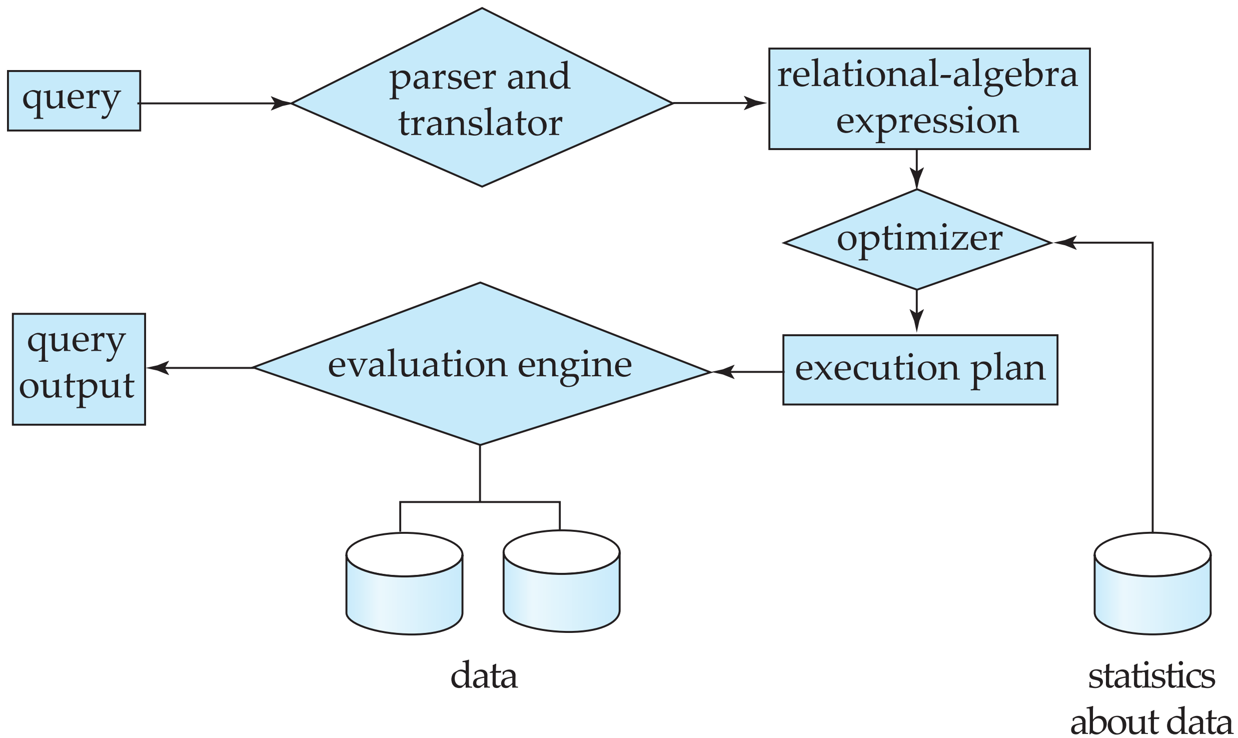
Capítulo refere-se a: Database System Concepts, 7th Ed

Chapter 15: Query Processing

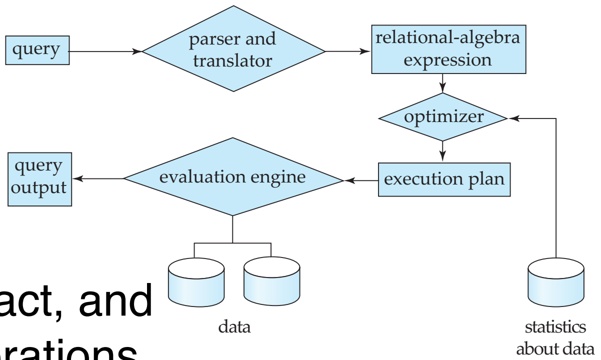
- Overview
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions

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 operations
  - Parser checks syntax, verifies relations
    - This is a subject for *compilers*
- 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 produce a good evaluation plan!
    - Query-execution is “simply” executing a predefined plan (or program)

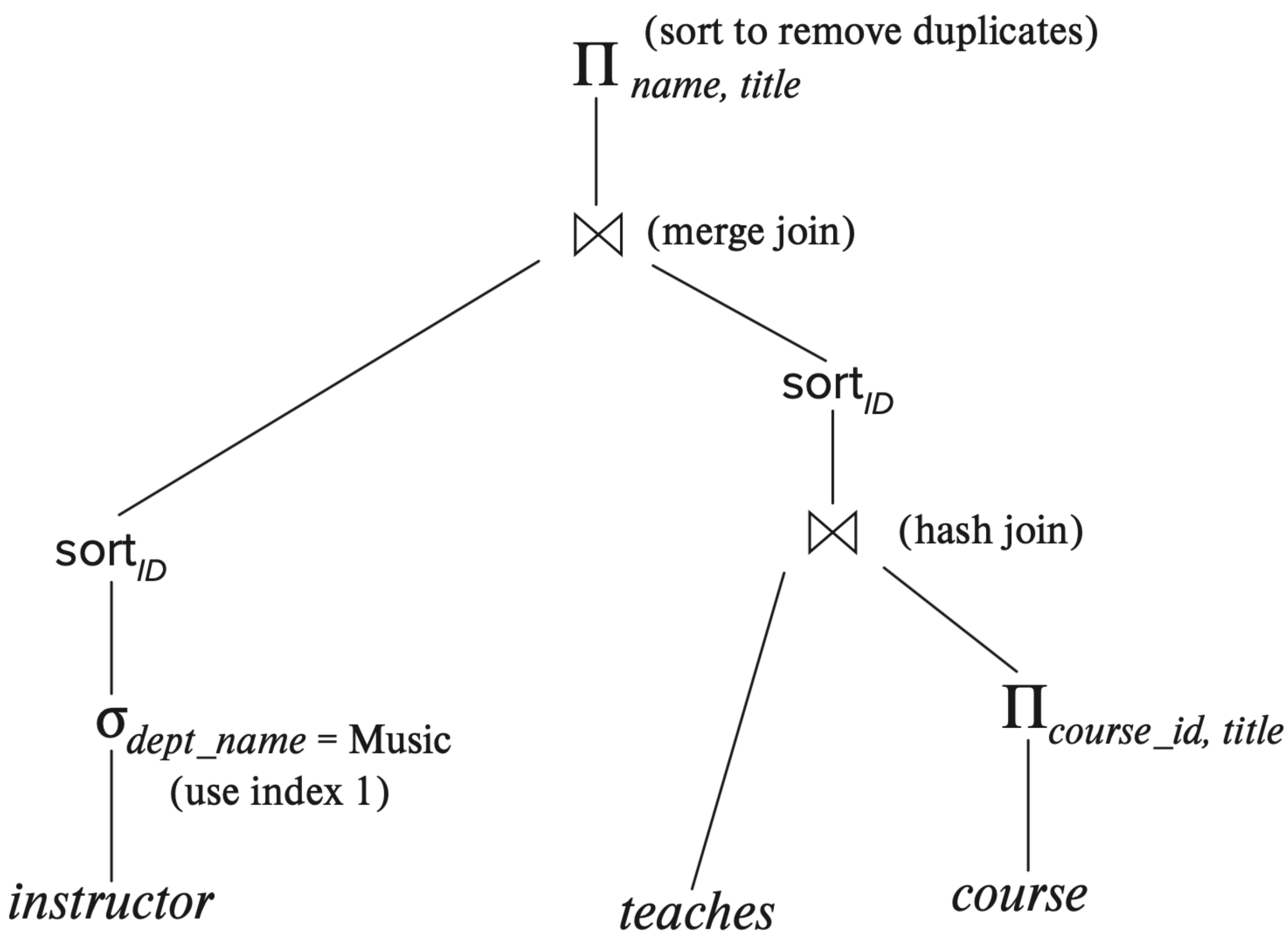
Basic Steps in Query Processing: Optimization

- A relational algebra expression may have many equivalent expressions, e.g.
  - $\sigma_{salary<75000}(\Pi_{salary}(instructor))$ is equivalent to

    $\Pi_{salary}(\sigma_{salary<75000}(instructor))$
- 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.,:
  - Use an index on *salary* to find instructors with salary < 75000,
  - Or perform complete relation scan and discard instructors with salary $\geq$ 75000

Evaluation Plan Example

An evaluation plan is a (possibly annotated) relational algebra expression



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 and the next lecture we study
  - How to measure query costs
  - Algorithms for evaluating relational algebra operations
  - How to combine algorithms for individual operations in order to evaluate a complete expression
- Immediately after that
  - We study how to optimize queries, that is, how to find an evaluation plan with lowest estimated cost

Measures of Query Cost

- Many factors contribute to time cost
  - *disk access, CPU*, and network *communication*
- Cost can be measured based on
  - **response time**, i.e. total elapsed time for answering query, or
  - total **resource consumption**
- We use total resource consumption as cost metric
  - Response time harder to estimate, and minimizing resource consumption is a good idea in a shared database
- We ignore CPU costs for simplicity, as they are usually much smaller
  - Real systems do take CPU cost into account
  - Network costs must be considered for distributed systems
- We describe how to estimate the cost of each operation

Measures of Query Cost

- Disk cost can be estimated as:
  - Number of seeks * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * average-block-write-cost
- 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
  - t_S – time for one seek
  - Cost for b block transfers plus S seeks
    $b * t_T + S * t_S$
- t_S and t_T depend on where data is stored; with 4 KB blocks:
  - High end magnetic disk: t_S = 4 msec and t_T =0.1 msec
  - SSD: t_S = 20-90 microsec and t_T = 2-10 microsec for 4KB

Measures of Query Cost (Cont.)

- Required data may be buffer resident already, avoiding disk I/O
  - But hard to consider for cost estimation
- 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
- Worst case estimates assume that no data is initially in buffer and only the minimum amount of memory needed for the operation is available
  - But more optimistic estimates are used in practice

Selection Operation (recall)

- Notation: $\sigma_{p}(r)$
- p is the selection predicate
- Defined by:

$$\sigma_p(r) = \{t \mid t \in r \textbf{ and } p(t)\}$$

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:

$$\sigma_{\mathit{branch\text{-}name}=\text{'Perryridge'}}(\mathit{account})$$

- For recalling other operators, see documentation of “Bases de Dados”.

Selection Operation

- **File scan**
- 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
    - 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
- Note: binary search generally does not make sense since data is not stored consecutively
  - except when there is an index available,
  - and binary search requires more seeks than index search

Selections Using Indices

- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index.
- **A2** (**clustering index, equality on key**). Retrieve a single record that satisfies the corresponding **equality** condition
  - $Cost = h_i * (t_T + t_S)$
- Recall that the height of a B+-tree index 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.
  - E.g. for a relation r with 1.000.000 different search key, and with 100 index entries per node, hi = 4
  - Unless the relation is really small, this algorithms (for equality condition) always "pays" when indexes are available

Selections Using Indices

- **A3** (**clustering index, equality on nonkey**) Retrieve multiple records.
  - Records will be on consecutive blocks
    - Let b = number of blocks containing matching records
  - $Cost = h_i * (t_T + t_S) + t_S + t_T * b$

- **A4** (**secondary index, equality on key/non-key**).
  - 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 if n is big!
        - Note that n multiplies by the time for seeks

Selections Involving Comparisons

- One 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** (**clustering 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 until first tuple $> v$; **do not use index**

- **A6** (**nonclustering 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 per record;
    - Linear file scan may be cheaper!!!

Implementation of Complex Selections

- **Conjunction:** $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \theta n}(r)$
- **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.
- **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 (or bitmaps)
  - Use corresponding index for each condition and take intersection of all the obtained sets of record pointers.
  - Then fetch records from file

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 that, or benefit from relations that are previously sorted

- 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
    - Sorting physically may lead to one disk access for each tuple.
  - For relations that fit in memory, techniques like quicksort can be used.
  - For relations that don't fit in memory, **external sort-merge** is a good choice.

External Sort-Merge

- If the relation does not fit in memory, divide it into runs that fit, and start by sorting those runs

Let M denote memory size (in pages).

1. **Create sorted runs**. Let i be 0 initially.
   Repeatedly do the following until 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.)

- Then merge the runs, two by two, in a sorted manner

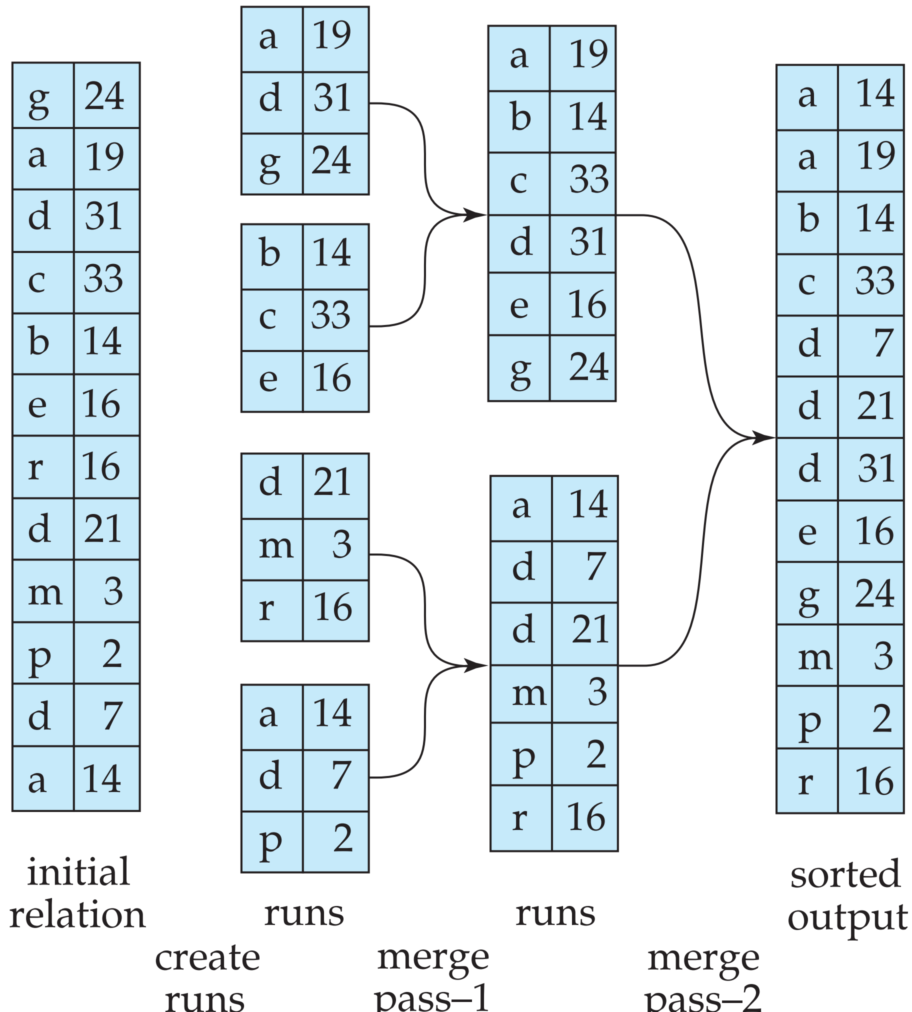
2. **Merge the runs (N-way merge)**. We assume (for now) that $N < M$.
   1. Use N blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page
   2. **repeat**
      1. Select the first record (in sort order) among all buffer pages
      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 until all runs have been merged into one.

- Note that, in practice, this is only needed for really huge relations:
  - Consider a 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)
    - A 4000TB relation is a really big relation (not found usually)!!!

Example: External Sorting Using Sort-Merge



External Merge Sort (transfer cost)

- Cost analysis:
  - Total number of merge passes required: $\lceil \log_{\lfloor M/bb \rfloor - 1}(b_r/M) \rceil$
    - This part >1 only for very very big relations
  - 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 (to be studied later)
    - Thus, total number of block transfers for external sorting:

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

      - (usually this boils down to $3b_r$)

- Legend:
  - M – size of the memory
  - b_b – number of blocks per run
  - b_r – number of blocks of relation r

External Merge Sort (seek cost)

- 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:
      $2\lceil b_r / M \rceil + \lceil b_r / b_b \rceil (2\lceil \log_{\lfloor M/bb \rfloor -1}(b_r / M) \rceil -1)$
      - (Usually, this boils down to $2\lceil b_r / M \rceil + \lceil b_r / b_b \rceil$

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 *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 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.
- Expensive since it examines every pair of tuples in the two relations.

Nested-Loop Join Costs

- 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 blocks of the inner relation
  - The number of seeks only depends on the outer relation
- However, if the smaller relation fits entirely 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 of each relation
  - Statistics on the size of the relations, in run time, can be a great help!

Nested-Loop Join Costs

- For joining *student* and *takes*, 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 and 2 seeks

- 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 block nested-loops algorithm (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 plus 2 seeks.

- In the running example the cost of *student* ⋈ *takes* is:
  - If *student* is outer: 100*400+100 = 40,100 transfer + 200 seeks
  - If *takes* is outer: 400*100+400 = 40,400 transfers + 400 seeks

- Improvements to nested loop and block nested loop algorithms:
  - If equijoin 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 (next slide)

Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - join is an equijoin or natural join and
  - an index is available on the inner relation's join attribute
    - In some cases, it pays to 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: $b_r\,(t_T + t_S) + n_r * c$
  - 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 (usually quite small compared to the joincost)
- 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* ⋈ *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
- As we've seen, the best 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.
  - 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 doesn't pay (this is so because the relations are small)