Database System Concepts

Chapter 13: Query Processing

Departamento de Engenharia Informática
Instituto Superior Técnico

1st Semester
2009/2010

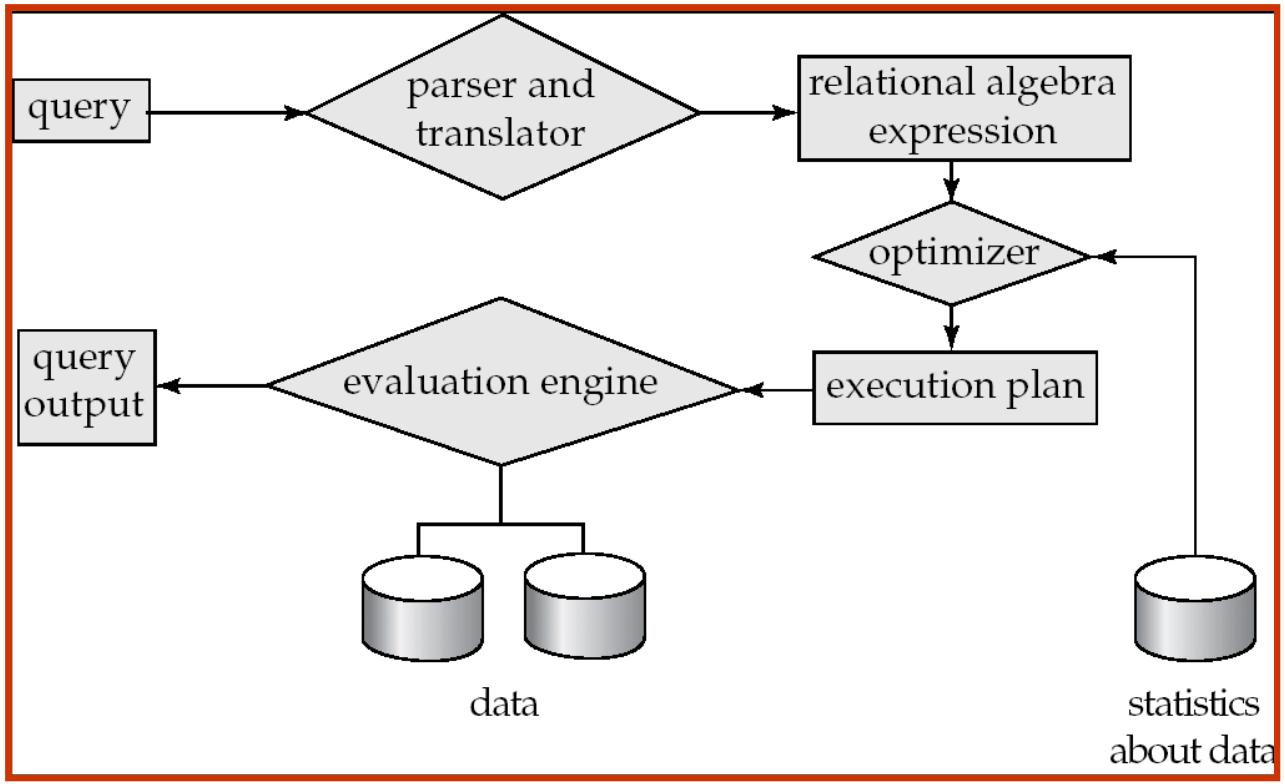
Slides (fortemente) baseados nos slides oficiais do livro "Database System Concepts"

Outline

Outline

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 and then into relational algebra
  - Parser checks syntax and verifies relations
- Optimization
  - Amongst all equivalent evaluation plans choose the one with lowest cost
  - Cost is estimated using statistical information from the database catalog, such as the number of tuples in each relation, size of tuples, etc.
- Evaluation
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query

Evaluation Plans

- A relational algebra expression may have many equivalent expressions. For instance:

$$\sigma_{balance<2500}(\pi_{balance}(account))$$
$$\Updownarrow$$
$$\pi_{balance}(\sigma_{balance<2500}(account))$$

- 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 balance to find accounts with $balance < 2500$
  - or can perform complete relation scan and discard accounts with balance ≥ 2500

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, taking into account:
  - Number of seeks $\times$ average-seek-cost
  - Number of blocks read $\times$ average-block-read-cost
  - Number of blocks written $\times$ average-block-write-cost
    - Cost to write a block is greater than cost to read a block because data is read back after being written to ensure that the write was successful

Outline

Scan Algorithms

File scan algorithms

- Linear search
  - Can be applied regardless of condition, ordering of records, availability of indices
- Binary search
  - Applicable if selection is an *equality comparison* on the attribute on which the file is *ordered*

Index scan algorithms

- Selection condition must be on search-key of index
- Depends on whether the index is primary/secondary, condition is equality/inequality, value is unique/not unique

Selections Involving Conjunctions

For selections of the form:

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n}(r)$$

- Select a combination of θ_i and algorithms that results in the least cost for $\sigma_{\theta_i}(r)$, then test other conditions on tuple after fetching it into memory
- Use appropriate composite (multiple-key) index if available
- Do a conjunctive selection by intersection of identifiers
  - 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

Selections Involving Disjunctions

For selections of the form:

$$\sigma_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n}(r)$$

- Disjunctive selection by union of identifiers
  - Applicable if all conditions have available indices
  - Otherwise use linear scan

Selections Involving Negation

For selections of the form:

$$\sigma_{\neg\theta}(r)$$

- This is simply the set of tuples that are not in $\sigma_{\theta}(r)$
- May need to use linear scan on file

Sorting

- 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, techniques like *quicksort* can be used
- For relations that don't fit in memory, external sort-merge is a good choice

External Sort-Merge

- Let M denote memory size (in blocks)
  1. Create sorted runs
  2. Merge the runs

Creating sorted runs

- Let i be 0
- Repeatedly do the following until the end of the relation:
  1. Read M blocks of relation into memory
  2. Sort the in-memory blocks
  3. Write sorted data to run R_i
  4. Increment i
- Let the final value of i be N

External Sort-Merge (cont.)

Merging the runs (N-way merge)

We assume (for now) that $N < M$

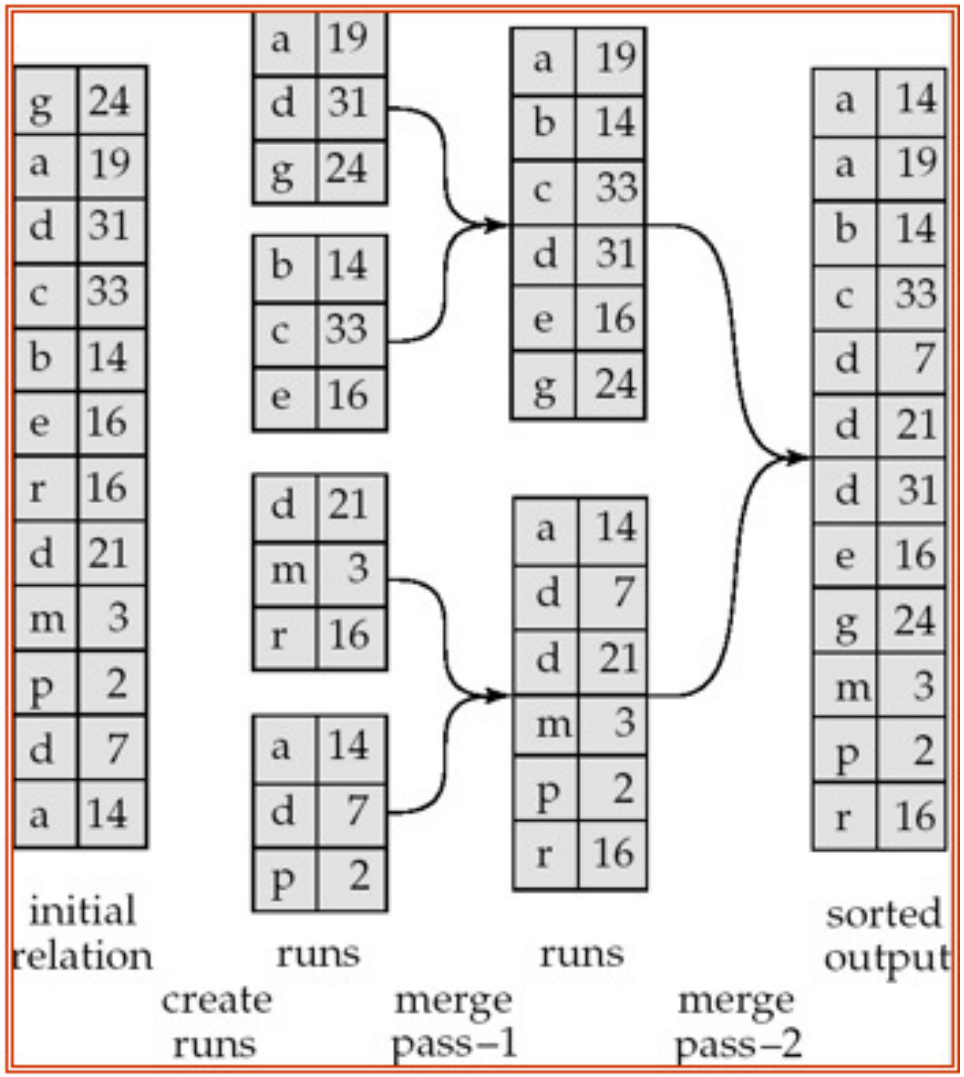
- Use N blocks of memory to buffer input runs, and 1 block to buffer output
- Read the first block of each run into its buffer page
- Repeat
  1. Select the first record (in sort order) among all buffer pages
  2. Write the record to the output buffer; if the output buffer is full write it to disk
  3. Delete the record from its input buffer page
  4. If the buffer page becomes empty read the next block (if any) of the run into the buffer

  Until all input buffer pages are empty

External Sort-Merge (cont.)

- If $N \leq M$, 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

An Example



Outline

Join Algorithms

- Several different algorithms to implement joins
  - Nested-loop join
    - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- The choice is based on cost estimates

Nested-Loop Join

To compute $r \bowtie_\theta s$

```
for each tuple t_r in r do
    for each tuple t_s in s do
        test pair (t_r, t_s) for the join condition θ
        if θ is satisfied, add (t_r, t_s) to the result
    end
end
```

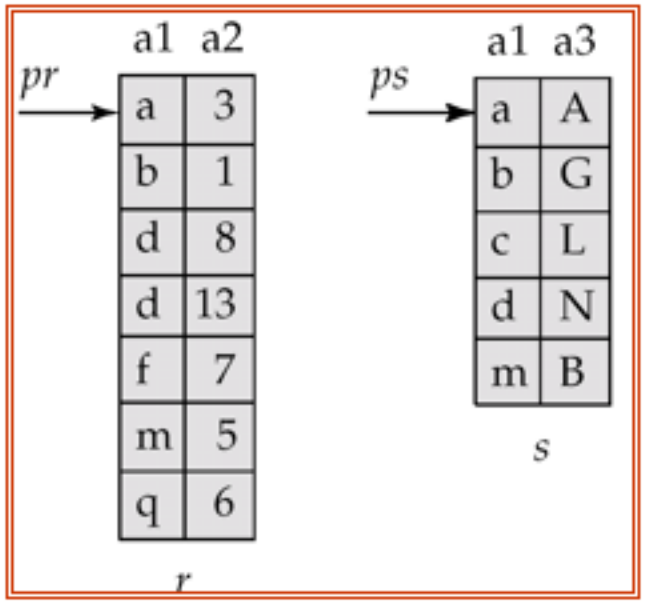
- r is called the outer relation and s the inner relation of the join
- Requires no indices and can be used with any kind of join condition
- Expensive since it examines every pair of tuples in the two relations

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.
- If indices are available on join attributes of both r and s, use the relation with fewer tuples as the outer relation

Merge-Join

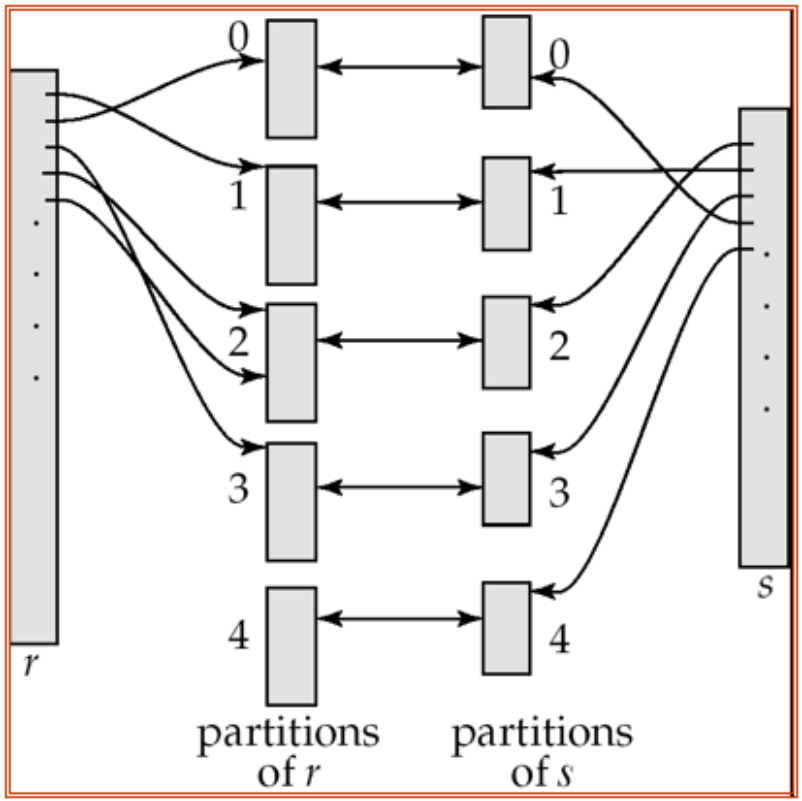
- Sort both relations on their join attribute (if not already sorted)
- Merge the sorted relations to join them
  - Join step is similar to the merge stage of the sort-merge algorithm
  - Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched



Hash Join

- Applicable for equi-joins and natural joins.
- A hash function h is used to partition tuples of both relations
- h maps A values to $\{0, 1, ..., n\}$, where A denotes the attributes of r and s used in the join
  - $r_0, r_1, \ldots, r_n$ denote partitions of r tuples
  - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[A])$
  - $s_0, s_1, \ldots, s_n$ denotes partitions of s tuples
  - Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s[A])$

Hash Join (cont.)



- r tuples in r_i need only to be compared with s tuples in s_i
- Need not be compared with s tuples in any other partition, since:
  - An r tuple and an s tuple that satisfy the join condition will have the same value for the join attributes
  - If that value is hashed to some value i, the r tuple has to be in r_i and the s tuple in s_i

Hash-Join Algorithm

Compute $r \bowtie s$

1. Partition the relation s using hashing function h.
   - One block of memory is reserved as the output buffer for each partition
2. Partition r similarly
3. For each i:
   1. Load s_i into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one h.
   2. 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 build input
- Relation r is called the probe input

Complex Joins

- Join with a conjunctive condition:

$$r \bowtie_{\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n} s$$

  - Either use nested loops/block nested loops, or
  - Compute the result of one of the simpler joins $r \bowtie_{\theta_i} s$
  - final result comprises those tuples in the intermediate result that satisfy the remaining conditions: $\theta_1 \wedge \ldots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \ldots \wedge \theta_n$

- Join with a disjunctive condition

$$r \bowtie_{\theta_1 \vee \theta_2 \vee \ldots \vee \theta_n} s$$

  - Either use nested loops/block nested loops, or
  - Compute as the union of the records in individual joins:

$$(r \bowtie_{\theta_1} s) \vee (r \bowtie_{\theta_2} s) \vee \cdots \vee (r \bowtie_{\theta_n} s)$$

Outline

Duplicate Elimination

- Duplicate elimination can be implemented via hashing or sorting
  - On sorting duplicates will come adjacent to each other, and all but one set of duplicates can be deleted
  - *Optimization*: duplicates can be deleted during run generation as well as at intermediate merge steps in external sort-merge
  - Hashing is similar—duplicates will come into the same bucket.

Aggregation

- Aggregation can be implemented in a manner similar to duplicate elimination
- Sorting or hashing can be used to bring tuples in the same group together, and then the aggregate functions can be applied on each group.
- *Optimization*: combine tuples in the same group during run generation and intermediate merges, by computing partial aggregate values
  - For *count*, *min*, *max*, *sum*: keep aggregate values on tuples found so far in the group.
  - For *avg*, keep sum and count, and divide sum by count at the end

Set Operations

- Set operations ($\cup$, $\cap$ and $-$): can either use a variant of merge-join after sorting, or a variant of hash-join

Example: set operations using hashing

- Partition both relations using the same hash function
- For $r \cup s$:
  - Add tuples in partition s_i to the result if they are not in r_i
  - At the end of s_i add remaining tuples to the result
- For $r \cap s$:
  - output tuples in s_i to the result if they are in r_i
- For $r - s$:
  - for each tuple in s_i, if it is in r_i, delete it
  - at end of s_i add remaining tuples to the result

Outer Joins

- Outer join can be computed either
  - as a join followed by addition of null-padded non-participating tuples
  - by modifying the join algorithms
- Modifying merge join to compute $r \mathbin{⟕} s$
  - In $r \mathbin{⟕} s$, non participating tuples are those in $r - \pi_R(r \bowtie s)$
  - During merging, for every tuple t_r from r that does not match any tuple in s, output t_r padded with nulls
  - Right outer-join and full outer-join can be computed similarly.
- Modifying hash join to compute $r \mathbin{⟕} s$
  - If r is probe relation, output non-matching r tuples padded with nulls
  - If r is build relation, when probing keep track of which r tuples matched s tuples. At end of s_i output non-matched r tuples padded with nulls

Outline

Evaluation of Expressions

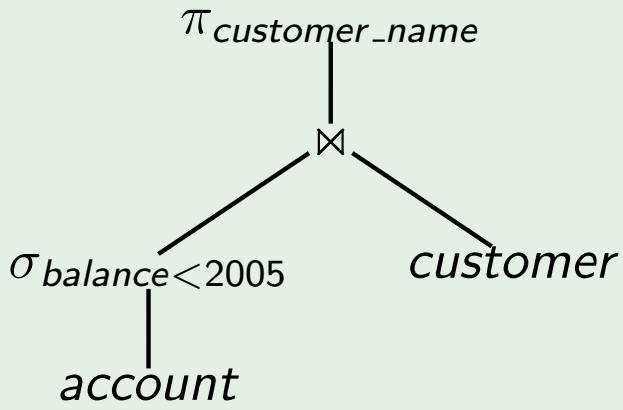
- Alternatives for evaluating an entire expression tree:
  - Materialization: generate results of an expression whose inputs are relations or are already computed, materialize (store) it on disk.
  - Pipelining: pass on tuples to parent operations even as an operation is being executed

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
- Always applicable, but cost can be quite high

Example

Compute and store $\sigma_{balance<2500}(account)$, then compute the store its join with *customer*, and finally compute the projections on *customer_name*



Pipelining

- Pipelined evaluation: evaluate several operations simultaneously, passing the results of one operation on to the next
  - E.g., in previous expression tree, don't store result of $\sigma_{balance<2500}(account)$, but instead, pass tuples directly to the join
  - 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., sort, hash-join
- Pipelines can be executed in two ways: demand driven and producer driven

End of Chapter 13