DATABASE MANAGEMENT SYSTEM

Subash Manandhar

Chapter 6 : Query Processing and Optimization

- Query processing is transforming a query written in high level language typically in SQL into correct and efficient execution strategy expressed in a low level language and to execute the strategy to retrieve required data.
- Basic steps in query processing:
  - Parsing and translation
  - Optimization
  - Evaluation

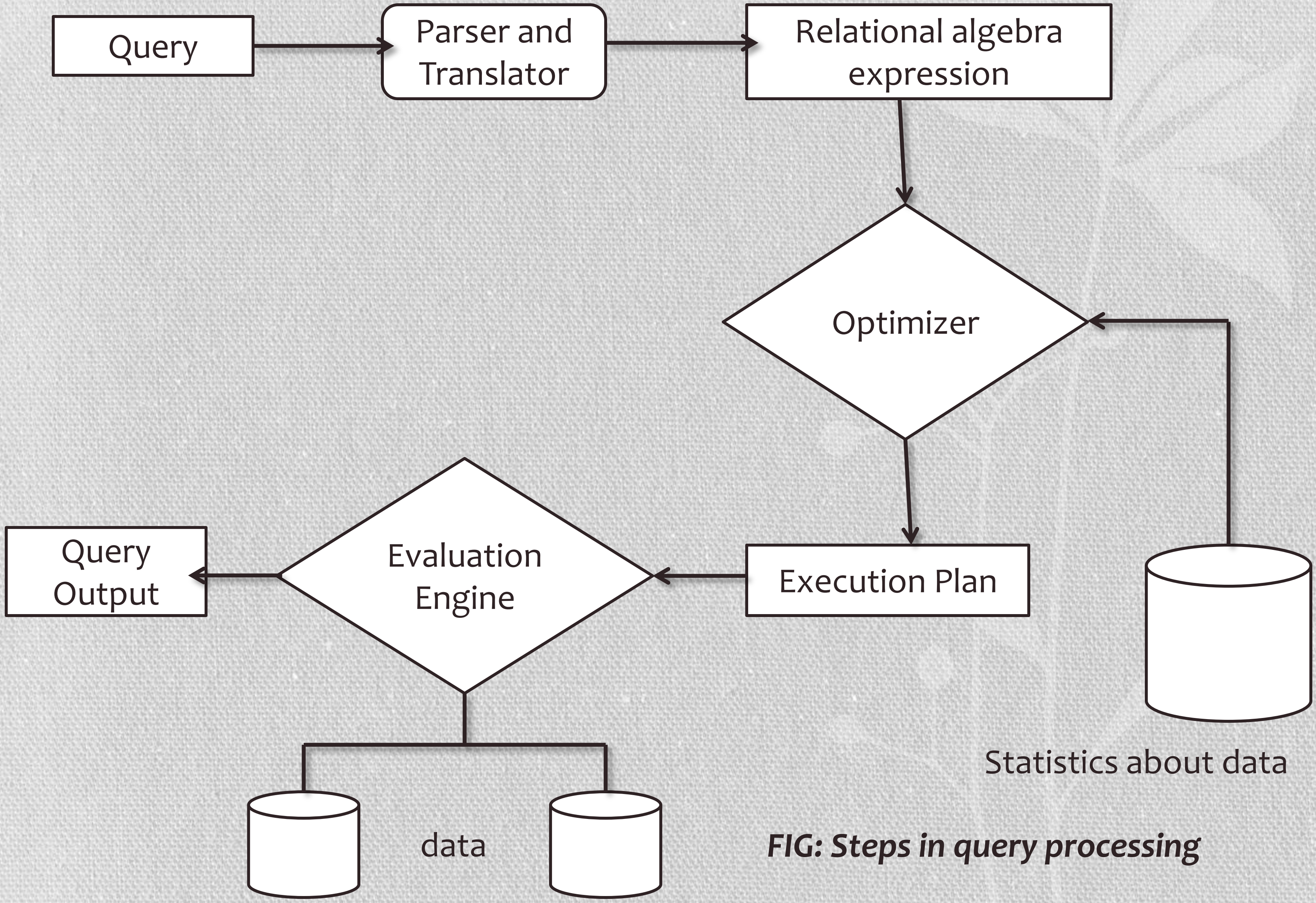


FIG: Steps in query processing

Chapter 6 : Query Processing and Optimization

- **Parsing and Translating:**
  - Translate the query into internal form. This is then translated into relational algebra.
  - Parser checks syntax, verifies relation.
  - The relational algebra representation of a query specifies only partially how to evaluate a query.
  - A relational algebra expression may have many equivalent expressions.
  - Consider a query:

  SELECT salary FROM employee WHERE salary > 20000

  Now, equivalent relational algebra expressions are

  1. $\sigma_{salary>20000}(\pi_{salary}(employee))$
  2. $\pi_{salary}(\sigma_{salary>20000}(employee))$

Chapter 6 : Query Processing and Optimization

- **Evaluation:**
  - The query execution engine takes a query evaluation plan, executes that plan and returns the answers to query.
- **Query Optimization:**
  - Amongst all equivalent evaluation plans choose the one with lowest cost.
  - Cost is estimated using statistical information from database catalog.

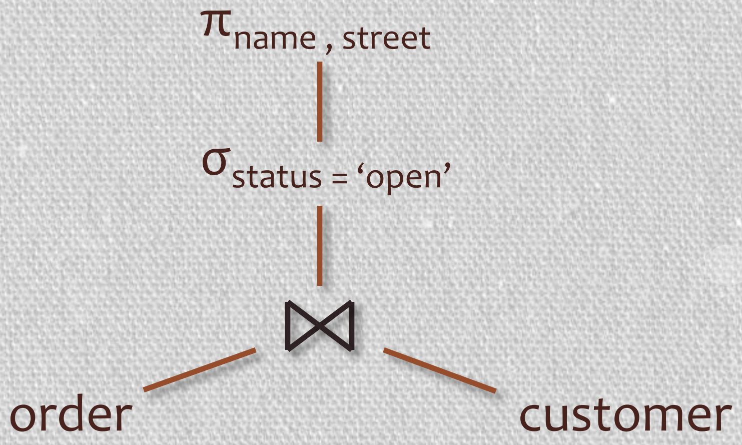
e.g. no. of tuples in each relation, size of tuples etc.

A sequence of primitive operations that can be used to evaluate a query is called query execution plan or query evaluation plan.

Once the query plan is chosen, the query is evaluated with that plan and result of query is output.

Chapter 6 : Query Processing and Optimization

- **Construction of parse tree:**
  - A leaf node is created for each base relation in query.
  - A non leaf node is created for each intermediate relation produced by relational algebra expression.
  - The root node represents result of query.
  - The sequence of operation directed from the leaves to the root.
  - E.g. $\pi_{name\ ,\ street}(\sigma_{status\ =\ 'open'}(order \bowtie customer))$



Chapter 6 : Query Processing and Optimization

- **Query Optimization:**
  - Is an important aspect of query processing.
  - Is the activity of choosing an efficient execution strategy for processing a query.
  - Aim is to choose one of many equivalent transformation that minimize resource usage.
  - Reduce the total execution time of the query, which is the sum of execution times of all individual operations that make up query.
  - There can be enormous differences in term of performance between different evaluation plans for same query.
  - E.g. seconds Vs days to execute same query.

Chapter 6 : Query Processing and Optimization

- **Cost based query Optimization:**
  - Generate logically equivalent expressions by using a set of equivalence rules.
  - Annotate the expressions to get alternative query evaluation plans.
  - Select the cheapest plan based on estimated cost.
  - Estimation of query evaluation cost based on statistical information from the catalog manager in combination with expected performance algorithms.
  - **Importance of query Optimization:**
  - Provides faster query processing
  - Requires less cost per query
  - Provides high performance of the system
  - Consumes less memory

Chapter 6 : Query Processing and Optimization

- **Measure 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
  - 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
      $b * t_T + S * t_S$

Chapter 6 : Query Processing and Optimization

- **Measure of query cost:**
  - 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
- **Query operation:**
  - Selection Operation
    - The lowest level query processing operator for accessing data is the file scan.
    - search and retrieve records for a given selection condition.
  - 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

Chapter 6 : Query Processing and Optimization

- **Selection 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
  - **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!

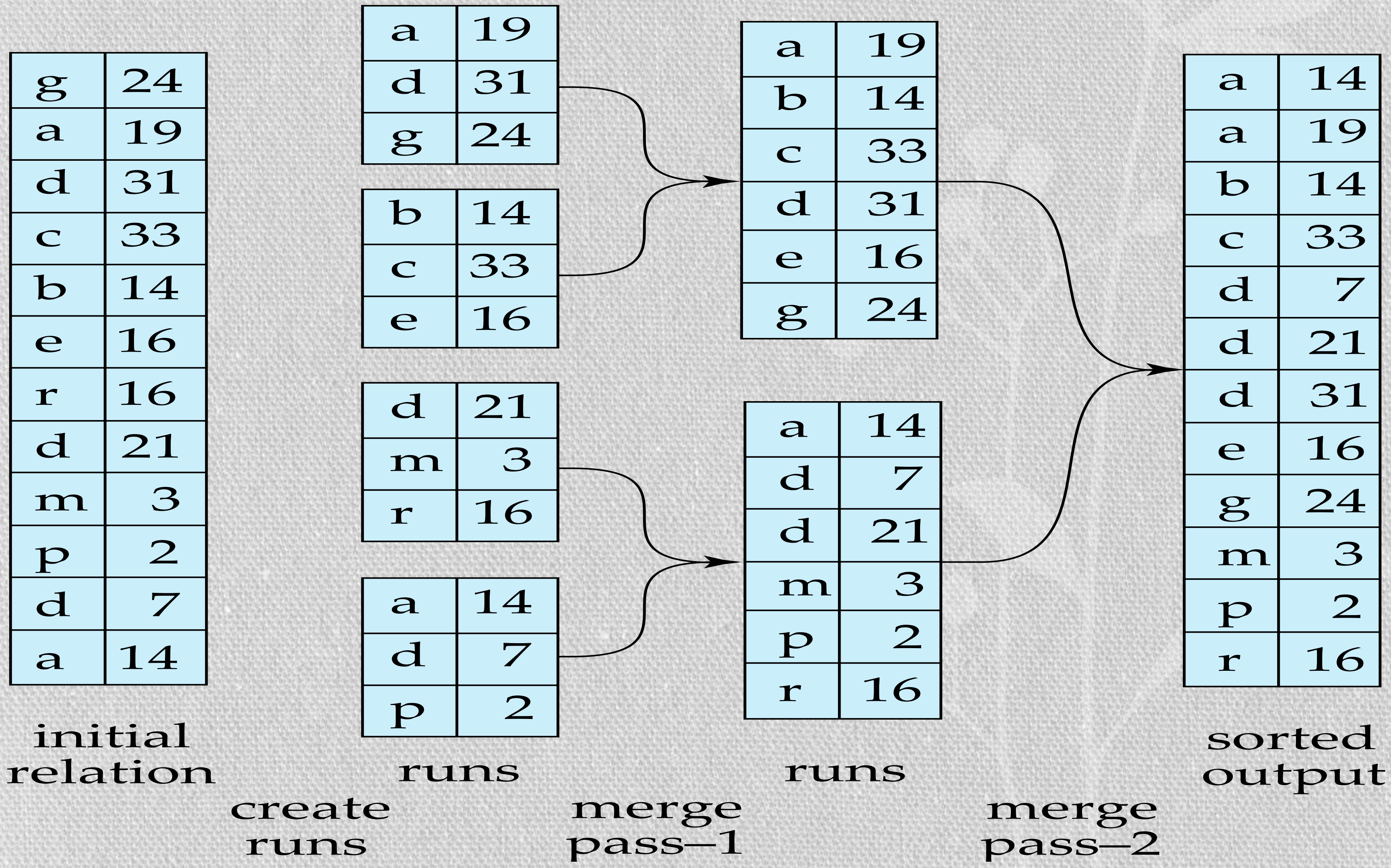
Chapter 6 : Query Processing and Optimization

- **Sorting:**
  - A query may specify that the output should be sorted.
  - The processing of some relational query operations can be implemented more efficiently based on sorted relations (join operation)
  - For relations that fit into memory, techniques like quick sort can be used.
  - For relations that do not fit into memory an external merge sort can be used.
- **External Merge Sort**
  - Let *M* denote memory size (in pages).
    - **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*

Chapter 6 : Query Processing and Optimization

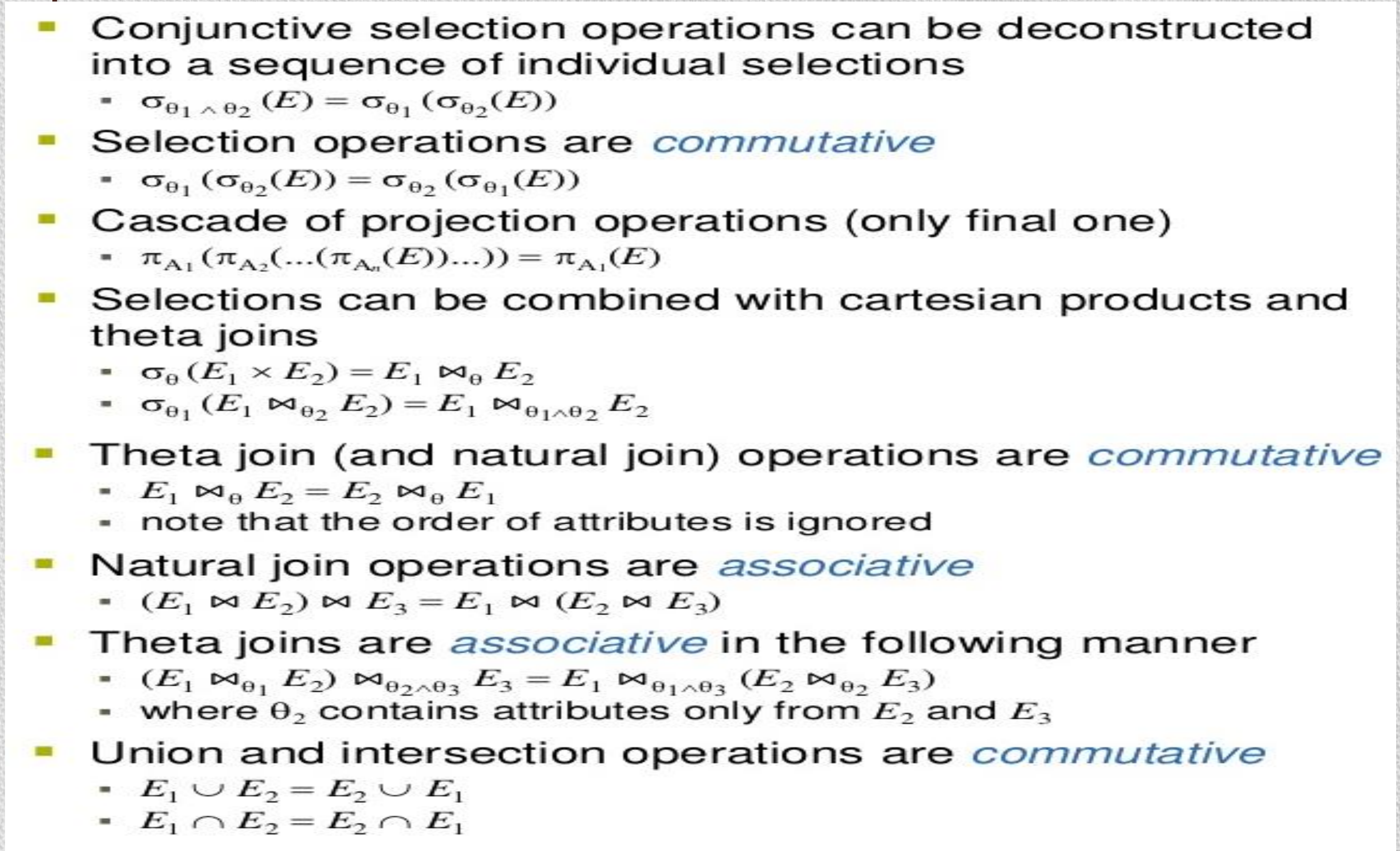
- **External Merge Sort**
  - **Merge the runs (N-way merge)**. We assume (for now) that $N < M$.
    - Use N blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page
    - **repeat**
      - Select the first record (in sort order) among all buffer pages
      - Write the record to the output buffer. If the output buffer is full write it to disk.
      - 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.
    - **until** all input buffer pages are empty:
  - 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.

Chapter 6 : Query Processing and Optimization



Chapter 6 : Query Processing and Optimization

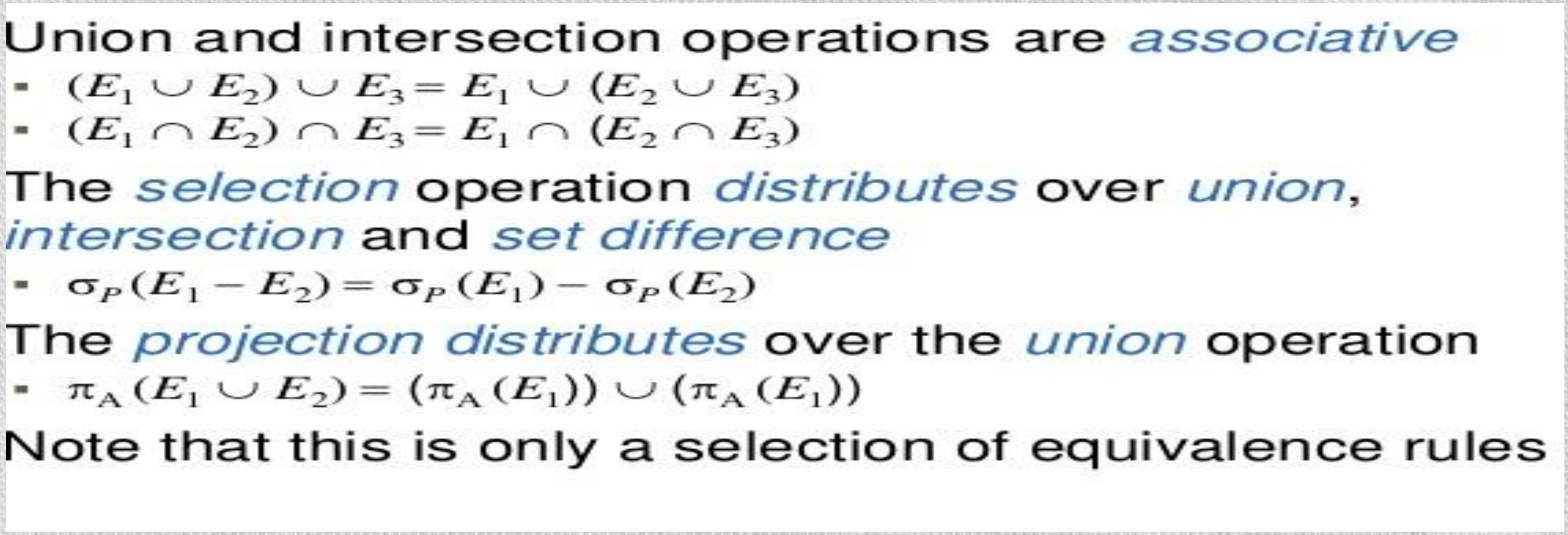
- **Equivalence Rule:**



- Conjunctive selection operations can be deconstructed into a sequence of individual selections
  - $\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$
- Selection operations are *commutative*
  - $\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$
- Cascade of projection operations (only final one)
  - $\pi_{A_1}(\pi_{A_2}(\ldots(\pi_{A_n}(E))\ldots)) = \pi_{A_1}(E)$
- Selections can be combined with cartesian products and theta joins
  - $\sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie_{\theta} E_2$
  - $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$
- Theta join (and natural join) operations are *commutative*
  - $E_1 \bowtie_{\theta} E_2 = E_2 \bowtie_{\theta} E_1$
  - note that the order of attributes is ignored
- Natural join operations are *associative*
  - $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$
- Theta joins are *associative* in the following manner
  - $(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$
  - where θ_2 contains attributes only from E_2 and E_3
- Union and intersection operations are *commutative*
  - $E_1 \cup E_2 = E_2 \cup E_1$
  - $E_1 \cap E_2 = E_2 \cap E_1$

Chapter 6 : Query Processing and Optimization

- **Equivalence Rule:**



Union and intersection operations are *associative*

- $(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$
- $(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$

The *selection* operation *distributes* over *union*, *intersection* and *set difference*

- $\sigma_P(E_1 - E_2) = \sigma_P(E_1) - \sigma_P(E_2)$

The *projection distributes* over the *union* operation

- $\pi_A(E_1 \cup E_2) = (\pi_A(E_1)) \cup (\pi_A(E_1))$

Note that this is only a selection of equivalence rules

Chapter 6 : Query Processing and Optimization

- **Heuristic Optimization:**
  - Cost-based optimization is expensive, even with dynamic programming.
  - Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion.
  - Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
    - Perform selection early (reduces the number of tuples)
    - Perform projection early (reduces the number of attributes)
    - Perform most restrictive selection and join operations before other similar operations.
    - Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

Chapter 6 : Query Processing and Optimization

- **Steps in Heuristic Optimization:**
  - Deconstruct conjunctive selections into a sequence of single selection operations .
  - Move selection operations down the query tree for the earliest possible execution .
  - Execute first those selection and join operations that will produce the smallest relations .
  - Replace Cartesian product operations that are followed by a selection condition by join operations .
  - Deconstruct and move as far down the tree as possible lists of projection attributes, creating new projections where needed .
  - Identify those sub trees whose operations can be pipelined, and execute them using pipelining.

Chapter 6 : Query Processing and Optimization

- **Performance Tuning**
  - **Database Statistics**
    - Get the correct and updated statistics
  - **Create Optimized Indexes**
    - Have a right balance of indexes on tables
  - **Specify optimizer hints in SELECT**
    - specify the index name in SELECT query
  - **Predetermine expected growth**
    - specify an appropriate value for fill factor when creating indexes
  - **Select limited data**
    - Rather than filtering on the client, push as much filtering as possible on the server-end
    - Eliminate any obvious or computed columns
  - **Drop indexes before loading data**
    - drop the indexes on a table before loading a large batch of data. This makes the insert statement run faster.
    - Recreate the indexes after insertion.
  - **Avoid foreign key constraints**
    - Foreign keys constraints ensure data integrity at the cost of performance