**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
  + *tT* – time to transfer one block
  + *tS* – time for one seek
  + Cost for b block transfers plus S seeks  
     *b \* tT + S \* tS*
* 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 Operations**

* 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 = *br* block transfers + 1 seek

* + - *br* denotes number of blocks containing records from relation *r*

If selection is on a key attribute, can stop on finding record

* + - cost = (*br* /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

**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
  + *Cost* = (*hi* + 1) \* (*tT* + *tS*)
* **A3** (**primary index, equality on nonkey**)Retrieve multiple records.
  + Records will be on consecutive blocks
    - Let b = number of blocks containing matching records
  + *Cost* = *hi \** (*tT* + *tS*)+ *tS* + *tT* \* b
* **A4** (**secondary index, equality on nonkey**)*.*
  + Retrieve a single record if the search-key is a candidate key
    - *Cost = (hi* + 1) \* (*tT* + *tS*)
  + Retrieve multiple records if search-key is not a candidate key
    - each of *n* matching records may be on a different block
    - Cost = (*hi* + *n) \** (*tT* + *tS*)
      * Can be very expensive!

**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).

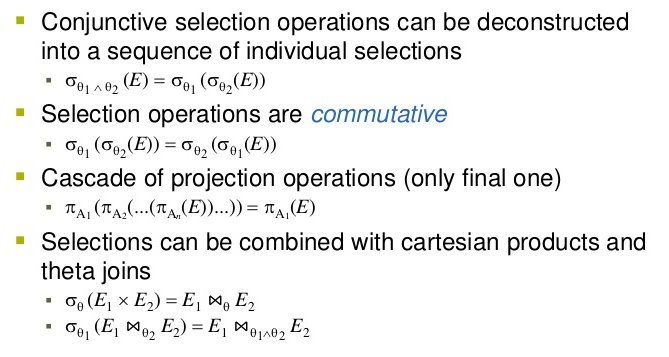
* **Create sorted** **runs**.
  + Let *i* be 0 initially.   
    Repeatedly do the following till the end of the relation:  
     (a) Read *M* blocks of relation into memory  
     (b) Sort the in-memory blocks  
     (c) Write sorted data to run *Ri*; increment *i.*Let the final value of *i* be *N*
* **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* ≥ *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.

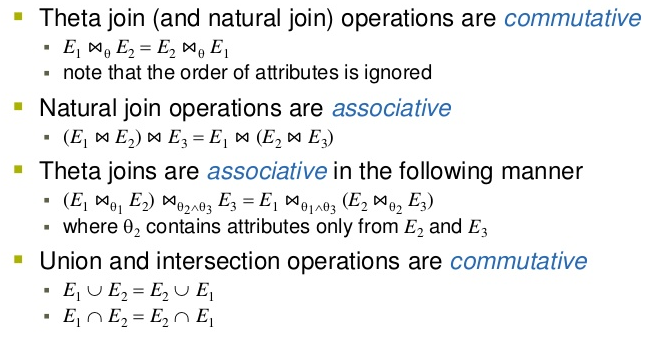


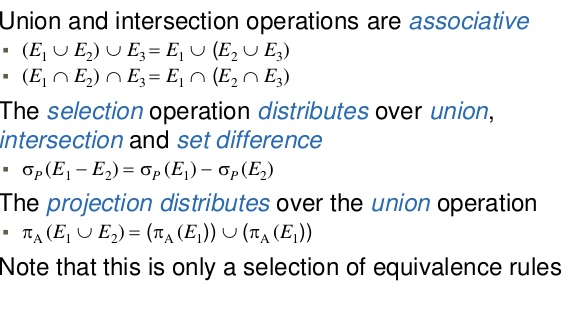
**Query Optimization**

* 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 transformations that minimizes resource usage.
* Reduce the total execution time of the query, which is the sum of execution times of all individual operations that make up the 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.
* 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 the estimated cost.
* Estimation of query evaluation cost based on statistical information from the catalogue manager in combination with the expected performance of algorithms.

**Equivalence Rule**







**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.

**Steps in Heuristic Optimization**

1. Deconstruct conjunctive selections into a sequence of single selection operations .

2. Move selection operations down the query tree for the earliest possible execution .

3. Execute first those selection and join operations that will produce the smallest relations .

4. Replace Cartesian product operations that are followed by a selection condition by join operations .

5. Deconstruct and move as far down the tree as possible lists of projection attributes, creating new projections where needed .

6. Identify those sub trees whose operations can be pipelined, and execute them using pipelining.

**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