Guidelines for Final exam

1. **Format:**

10 multiple choice questions

5 short answer questions

8 problem solving questions (some questions have multiple parts)

1. **Content of Multiple choice questions and short answers**

ER diagram

Relational model:

* relational schema
  + Relation name (Attribute: Domain)
* relational database schema,
  + A set of relational schema
* Relational instance,
  + Relation with attributes and with table
* Domain
  + Atomic value
* cardinality,
  + # of tuples in relational instance
* degree,
  + # of attributes in relational model
* foreign key constraints (why is it important)

Relational algebra:

* basic operations of RA
  + Union
  + Difference
  + Selection
  + Projection
  + Cartesian product
* special operations of RA
  + Intersection
  + Theta join
  + Natural join
  + Equi join
  + Outer join
* Query execution:

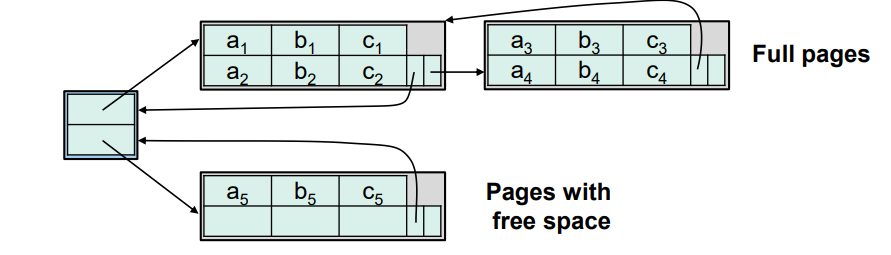
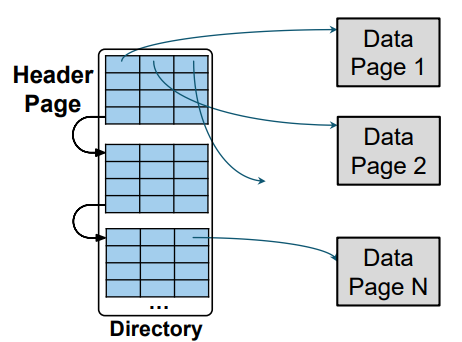
Given a set of queries in natural language and a set of relations, generate the SQL queries, and then RA. Use that RA to generate different physical plan of execution for that queries (with some given information such as join algorithm, index)

* Normalization:

Given a relation, a set of FDs, determine the candidate keys, check if the relation is 3NF or BCNF and decompose it if needed

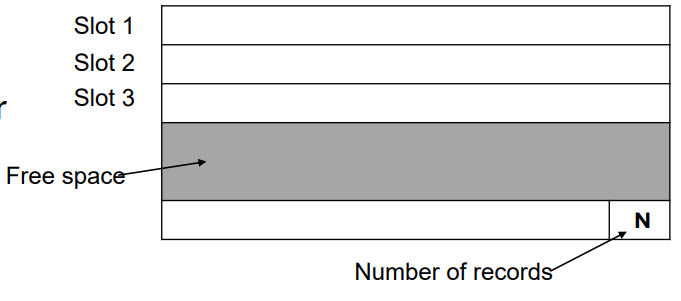
Given a relation, a set of FDs, and one way to decompose it, see if the decomposition is good

File management system:

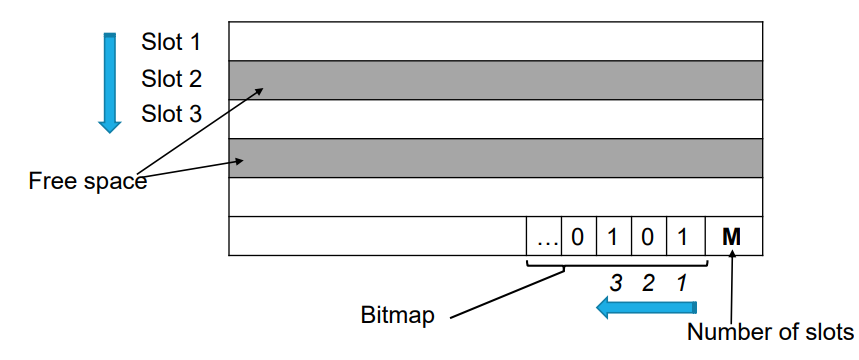
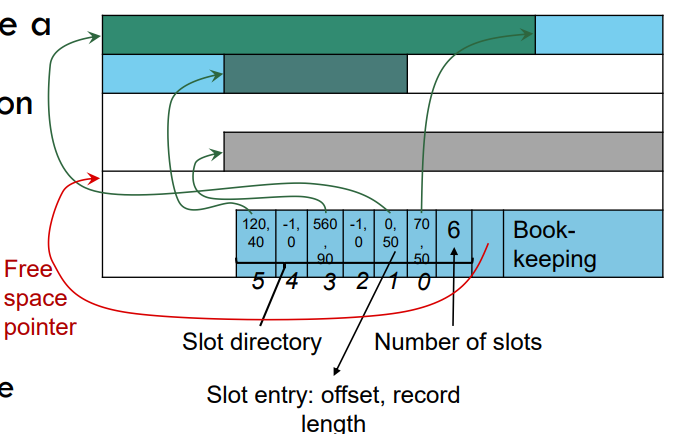
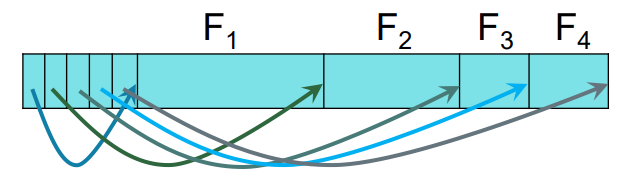
* why do we have to care
  + Relative placement of pages on disk has major impact on DBMS performance
* what are the times affecting the overall performance
  + Seek time
    - Time to move the arm to position disk head on the right track
  + Rotational delay
    - Time to wait for sector to rotate under the disk head
  + Data transfer time
    - Time to move the data to/from the disk surface
  + To low disk access time
    - Reduce seek time and/or rotational delay through optimizing the sequential arrangement of blocks
      * Block -> Blocks on same track -> Blocks on same cylinder -> Blocks on adjacent cylinders
    - For a sequential read, pre-fetching several pages at a time
* File organization
  + Disk anatomy
    - Sector
      * Portion of a track
    - Track
      * Circle around in platter
    - Platter
      * Circular hard surface on which data is stored by inducing magnetic changes
    - Spindle
      * Axis responsible for rotating platters
    - Disk head
      * Mechanism to read or write data
    - Arm assembly
      * Moves to position a head on a desired track of platter
  + Table on disk
    - Data < disk blocks/Page/Table < File
    - Field < Record < Page < File
    - Block/Page
      * Size is multiples of sector size
* File organization (Unordered/Heap file)
  + Must keep track of
    - Pages in a file
    - Free spaces on pages
    - Records on a page
  + Heap file using a List
    - 
  + Heap file using a page directory
    - * 
        + Each entry for a page keeps track of

Whether the page is full or not

Number of free bytes on the page

* + Page organization
    - Fixed length record
      * Packed organization
        + 

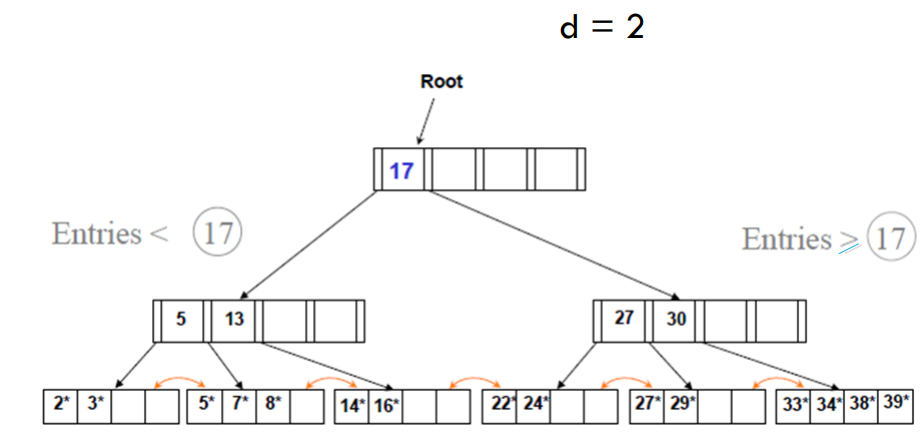
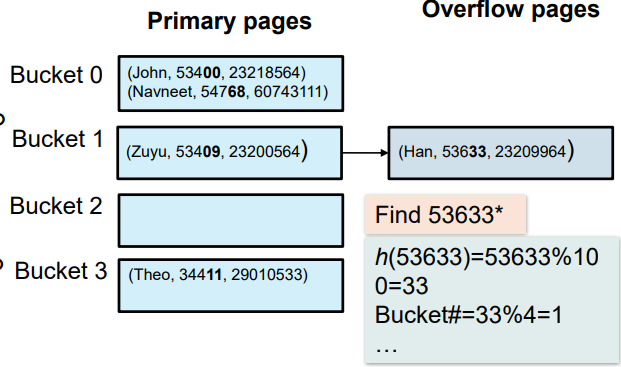
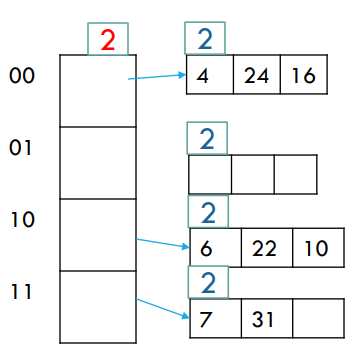
Address: Base slot + slots

* + - * Unpacked organization
        + 
    - Variable length records
      * 
        + RID does not change if moved on the same page
        + Deletion: Offset is set to -1
        + Insertion: Use any available slots. Otherwise, reorganize
        + Use delimiter to denote the end of each field
        + 

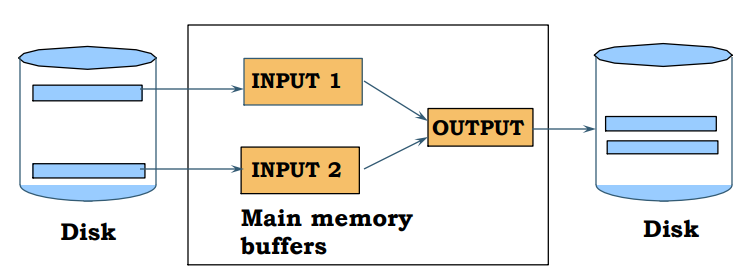
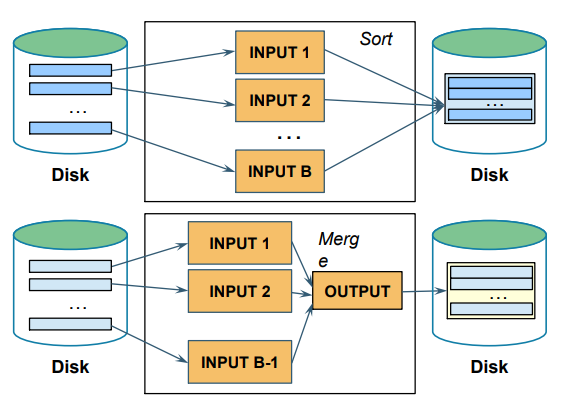
Buffer management:

* Algorithm
  + Buffer manager is in charge of bringing pages from disk to memory
  + Requests to buffer manager
    - Requests a page
    - Release a page when it is no longer needed
    - Notify the buffer manager when a page is modified
  + LRU
    - Replace least recently used frame
    - Keep a queue of pointers to frames available to be replaced
      * Add available frame pointers to the tail of the queue
      * Pick the next available frame to be replaced from the head of the queue
  + MRU
    - Replace most recently used frame
  + Sequential flooding problem
    - Situation caused by LRU policy when repeated sequential accesses happen
      * # buffer frames < # pages in file
    - Use MRU to solve the problem
* Strategies
  + When a page is requested
    - If the page is in the buffer pool
      * Return a handle to the frame in the buffer pool
      * Increment the pin count
    - If the page is not in the buffer pool
      * Choose a frame for replacement (LRU or MRU)
      * If the frame is dirty, write it to disk
      * Read requested page into the chosen frame
* PIN
  + Pin count: # of users of the page
    - Pin a page
      * Indicate the page is in use (Pin count > 0)
    - Unpin a page
      * Release the page (Pin count = 0)
      * Indicate whether the page is dirtied
    - Dirty bit
      * Indicates whether the page has been modified
      * Dirty bit 1
        + The page has been modified
        + Need to write the page to disk before release of the page
        + Pin the page and return a handle to it

Indexing:

* what is it, why is it important,
  + Index
    - Data structure that organizes data recods on disk to optimize selections on the search key fields for the index
    - Contains a collection of data entries
    - Supports efficient retrieval of all data entries with a given search key value k
* B+ tree
  + 
  + Each node must have minimum 50% occupancy
    - Each node is one disk page
  + All paths from root to leaf are of the same length
    - Log [n/2](K)
      * K: # of search key  
        N: 2 \* degree
  + Average fanout: 2 \* d \* Typical fill-factor = # of children nodes per node
  + Typical capacities: Average fanout ^ height
* Hash-based index
  + Best for equality searches but don’t support range searches
  + Static hashing
    - 
      * Need Hash function, # of buckets, # of records in a bucket
      * H(K) mod N = Bucket in which the data entry belongs
        + H(K): Hash function
        + N: # of buckets
* Extensible (dynamic) hash question(s).
  + 
  + Global depth >= Local depth
    - Global depth: # of bits to represent each entry
    - Local depth: # of bits to indicate the entry
  + Overflow Local depth = Global depth
    - Double the size of table
      * Update global depth
    - Add new bucket
      * Update local depth for new bucket and original bucket
    - Redistribute the data
    - Update pointers
  + Benefit
    - Does not degrade with growth of a file
  + Disadvantage
    - Bucket address table may become very big
    - Changing size of bucket address is an expensive operation

Internal and External sorting:

* External sorting
  + Pass
    - Collection of merges that go through entire data
  + Run
    - Sorted sub list
  + Complexity of merge sort: Nlog2(N)
    - N: cost for each pass
    - Log2(N): # of passes
  + 2-way merge sort
    - 
    - Total cost: 2N(log2N + 1)
      * 2N: Each pass read + write each page in file
      * Log2N + 1: Number of passes + Pass 0
  + Multi-way merge sort
    - 
      * Pass 0: Produce [N/B] sorted runs
      * Pass 1, … : Load B-1 runs and merges them into one run
    - # of passes: [logB-1[N/B]] + 1
      * LogB-1[N/B]: # of passes to merge runs
      * 1: Pass to create initial sorted runs
    - Total cost: 2N([logb-1[N/B]] + 1)
* Clustered B+ Tree
  + Retrieve leftmost entry, sweep through leaf pages in order
  + Cost
    - If data is not in the index: Height + # of leaf pages in index + # of data pages
    - If data is in the index: Height + # of leaf pages in index
* Unclustered B+ Tree
  + Cost
    - Worst-case: As many as I/O as the number of records
    - Slower than external merge sort

Evaluation of relational operators:

* Logical operator
  + What they do
  + Selection
    - File scan access path
      * Scan the entire file of R
      * Match selection of predicate for each tuple
      * Cost: B(R) - # of pages of R
        + B(R): # of blocks in R
    - Indexed access path
      * Use an index matching the selection predicate
      * Cost: Varies depending on the index
        + Hash Index: O(1)
        + B+ Tree index: O(logFN + X)

LogFN: Level

Unclustered Tree

X: # of selected tuples

Clustered Tree

X: (# of selected tuples) / (# of tuples per page)

* + - An index can be used to evaluate a selection operation only if matches some predicate in selection condition
      * Primary conjunct
        + Conjuncts in the selection conditions that an index matches
      * Conjunctive Normal Form
        + 
        + Hash index on search key K

All pi should be equality predicates

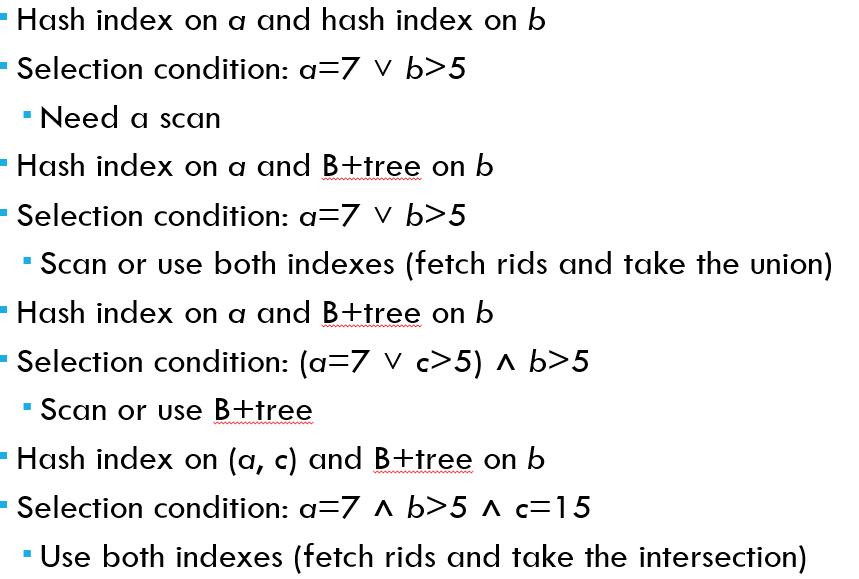
Set of attributes in K is a subset of attributes in p

* + - * + B+ Tree index on search key K

A prefix subset of K is a subset of attributes appearing in p

* + - * Disjunction
        + Can use indexes only when we have appropriate indexes for every simple predicate in the disjunction
    - Selectivity of access path
      * Fraction of pages that need to be retrieved if we use this access path
      * Hash index on all attributes
        + Selectivity is approximated by # of pages / # of keys

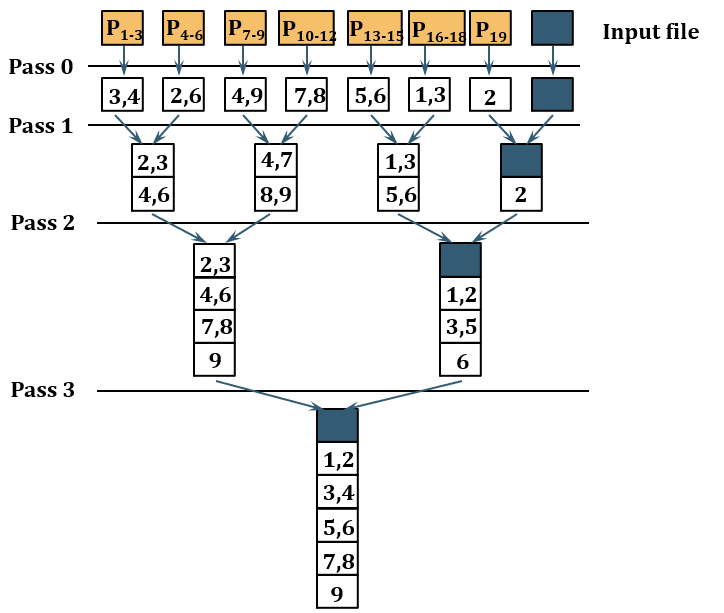
# Keys: Known from the index. Otherwise, 0.1 as default

* + - * Hash indexes on an attribute
        + Multiply the reduction factors for each primary conjunct
        + Reduction factor = # pages / # keys
  + 
  + Projection,
    - Sorting based duplication Sort R on (a,b)
      * First pass
        + Eliminate everything but a and b from each tuple
      * Later passes
        + Eliminate duplicates when encountered
        + Cost: B(R) + NT + EMrgCost(NT)

B(R): # pages in R

NT: # pages storing the results of the first pass

EMrgCost(NT): Cost of merging NT pages

* + - * 
    - Hashing based duplication
      * Create a hash table on R(a,b)
      * If the hash table fits entirely in memory
        + Cost: B(R)
      * Else use a 2 phase-algorithm
        + Partitioning

Project out attributes

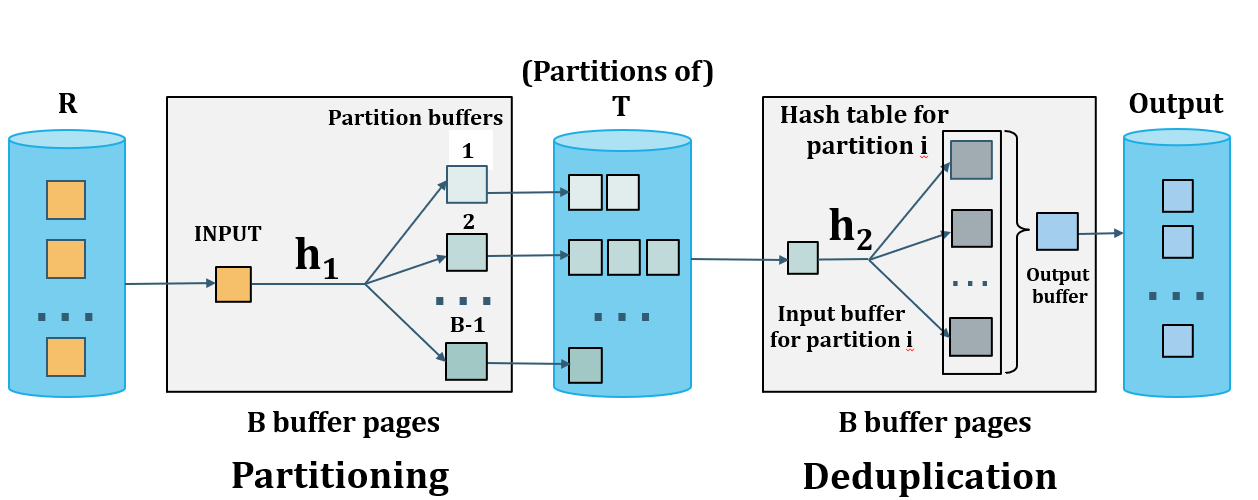
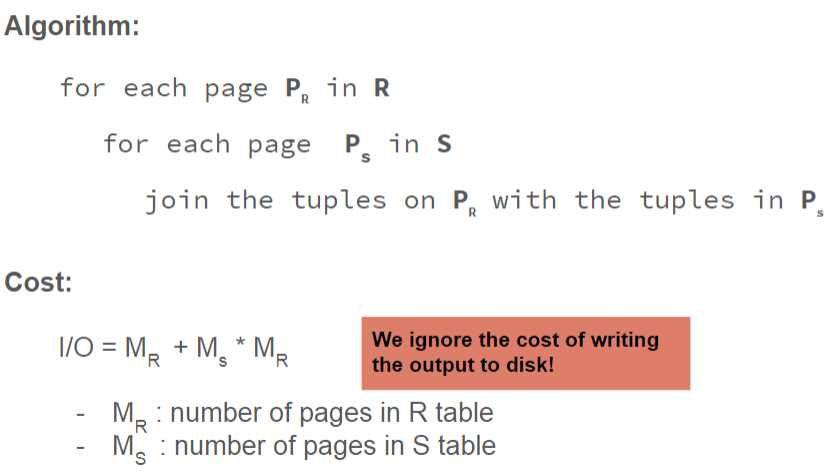
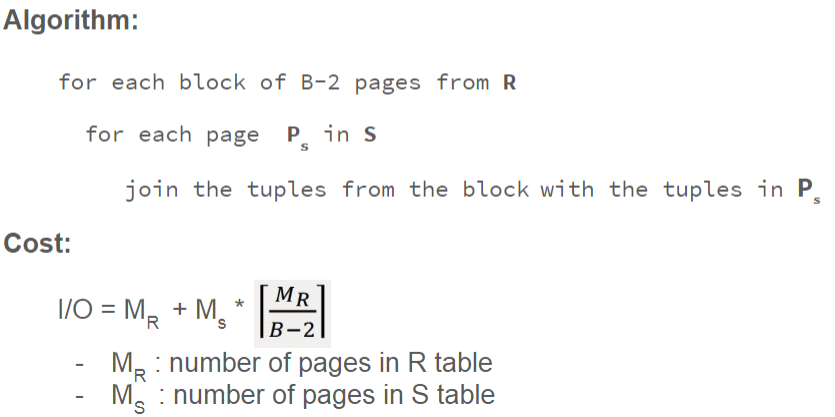
Split the input into B-1 partitions using a hash function

* + - * + Deduplication

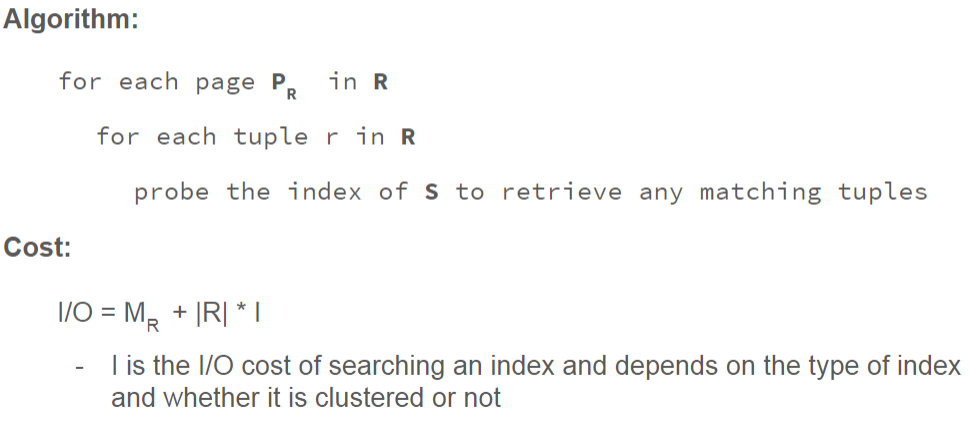
Read each partition into memory

Use an in-memory hash table to remove duplicates

Hash table with different hash function

* + - * + 
  + Join,
    - Take smaller relations as outer relation
    - **Nested loop**
      * Need only 3 buffer frames (pages)
      * 
    - **Block based nested loop**
      * Buffer collection
        + One buffer for scanning the inner S
        + One buffer for scanning for output
        + All remaining buffers for holding a block of outer R
      * 
        + Scan of outer table + #Outer blocks \* Scan of inner table

#Outer blocks = #pages outer / Block size and round up

* + - **Index Nested Loop Join**
      * 
        + I = (Cost of probing index + Cost for finding the values of matching r in R)

Cost of probing index

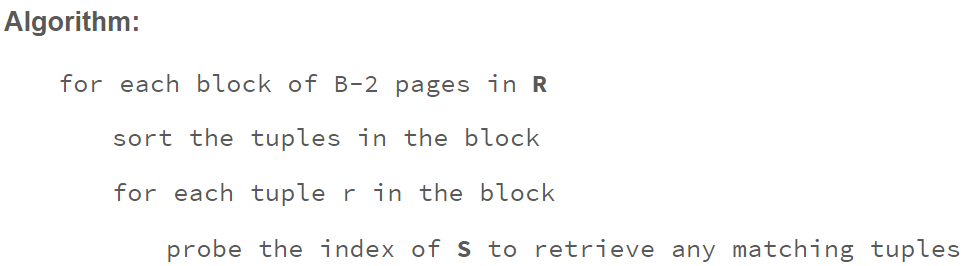
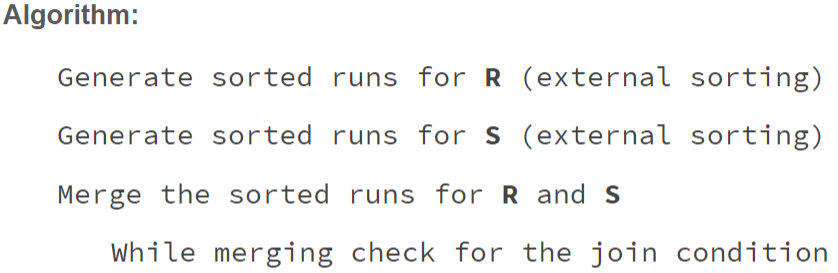
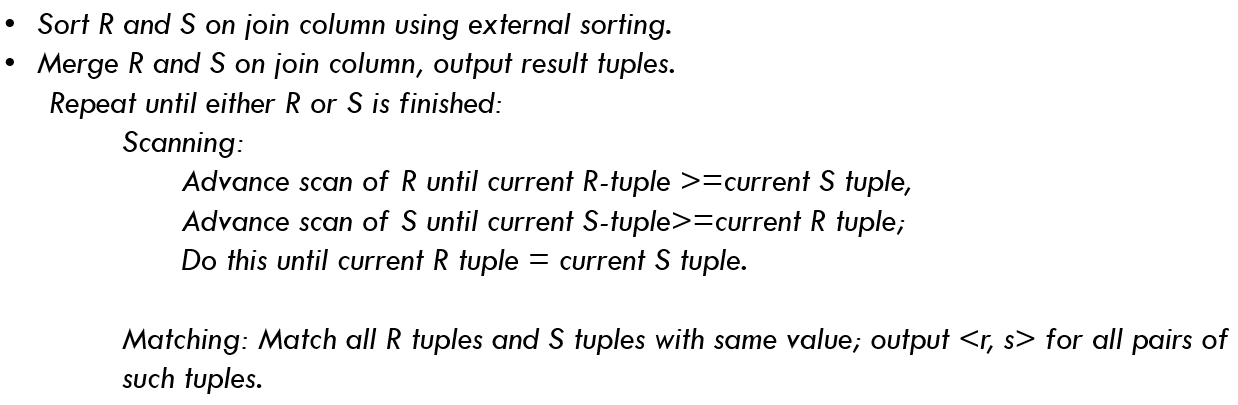
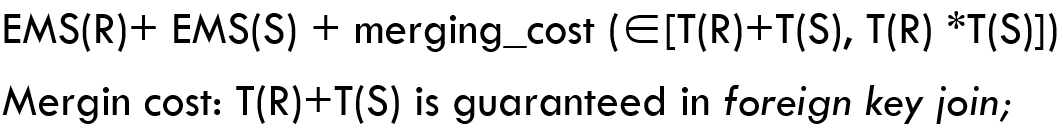
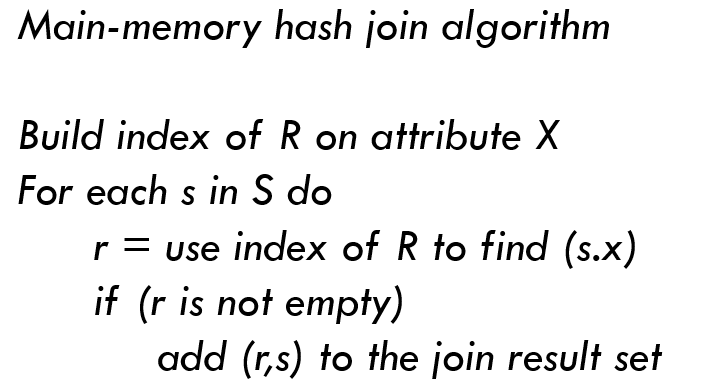
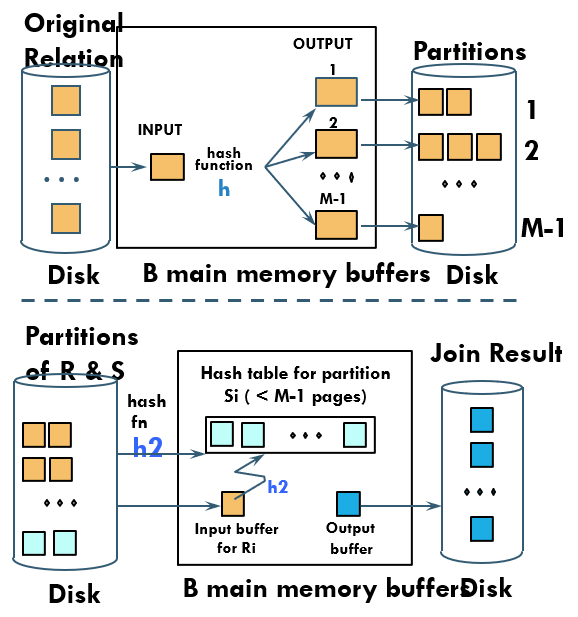
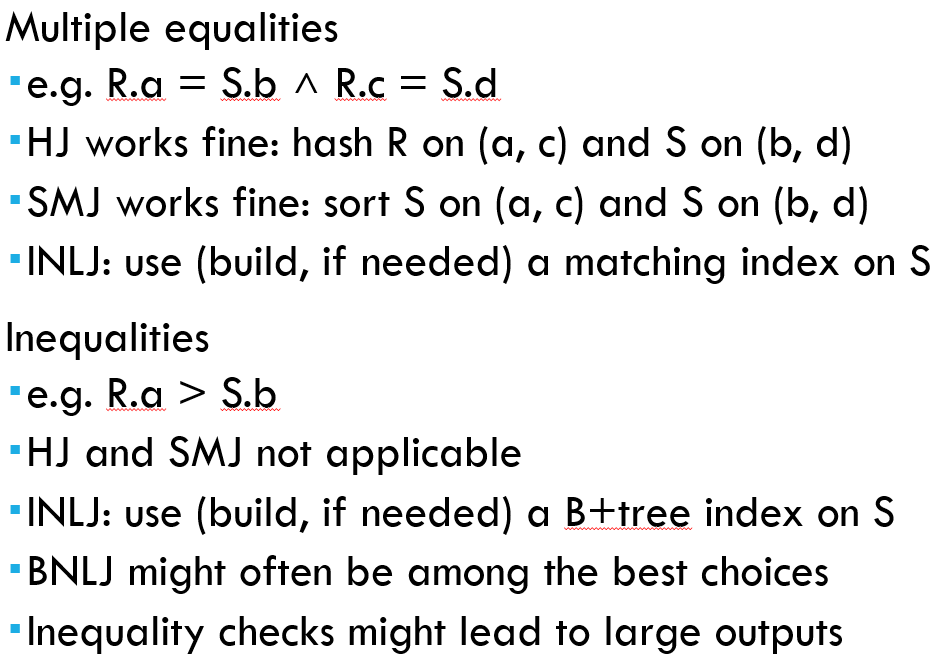
1.2 for Hash index

2.3 for B+ Tree index

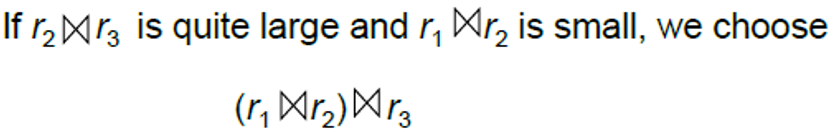
Cost for finding values of matching

1 for cluster index

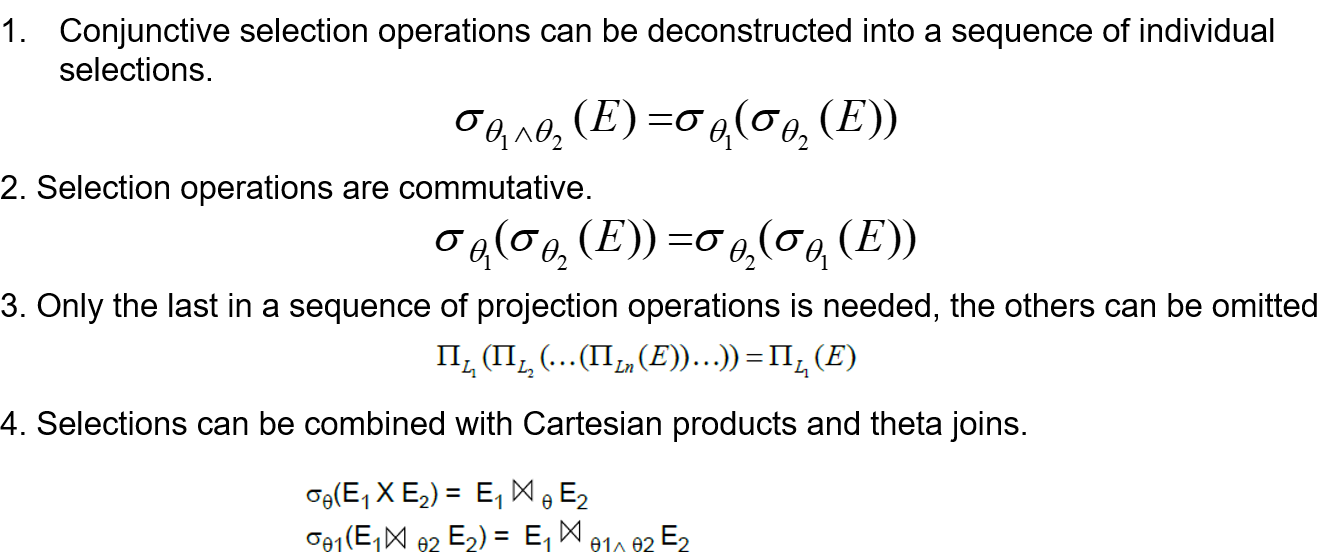
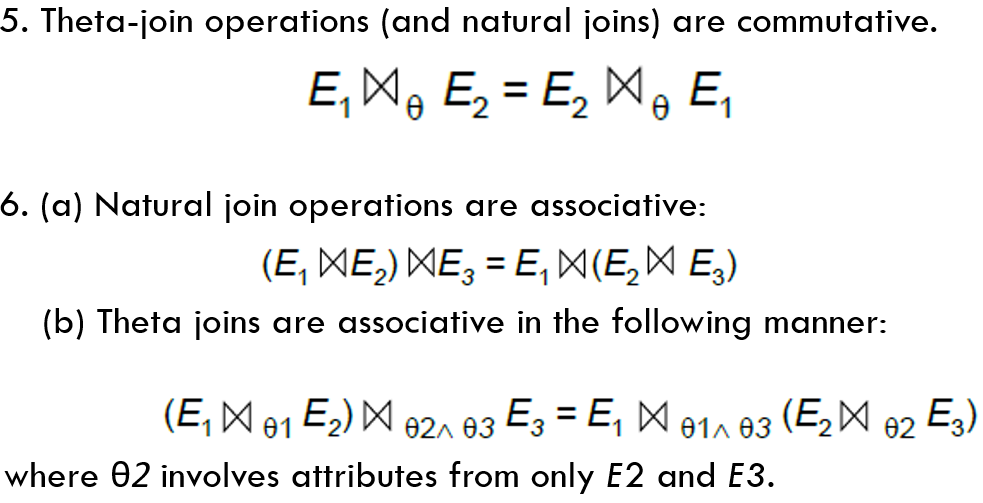
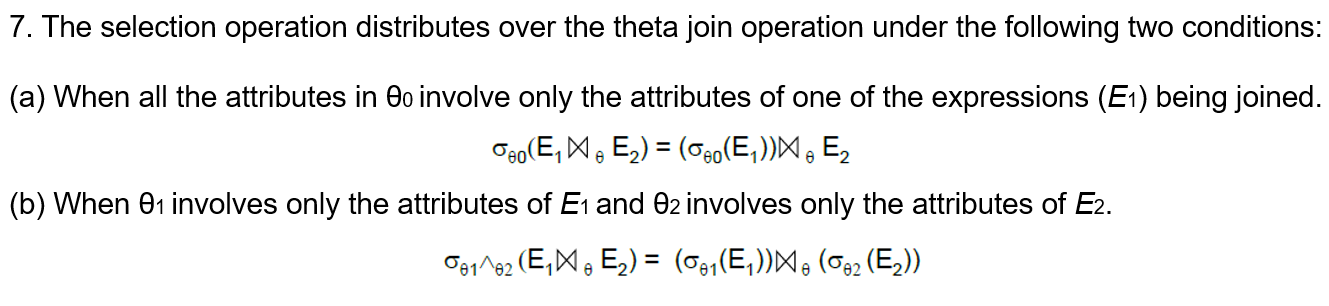
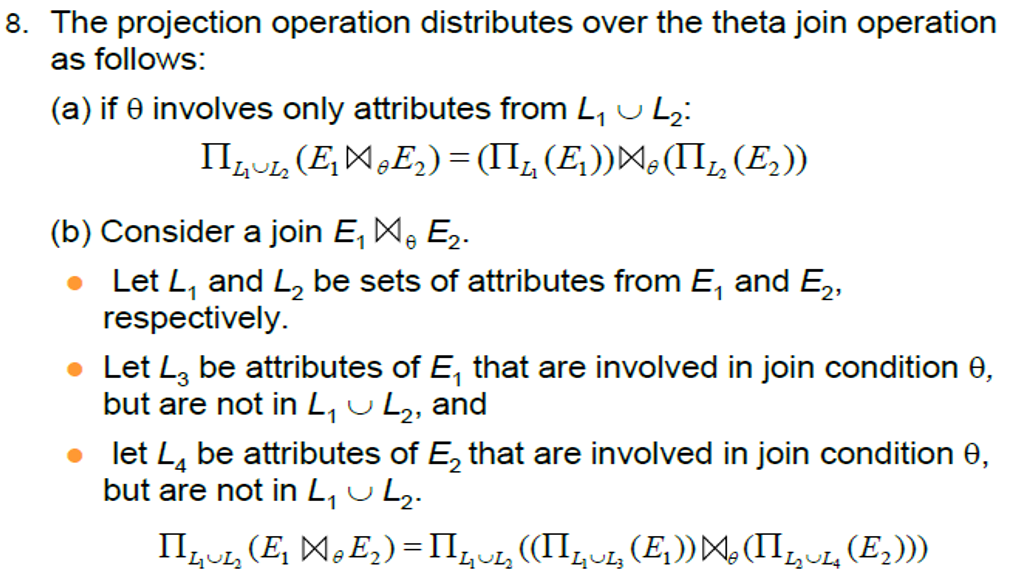
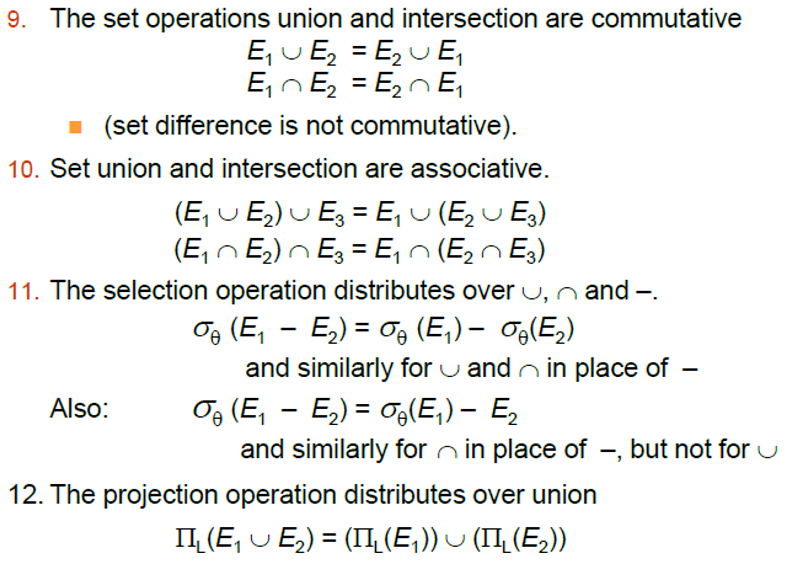
1 per matching tuple in uncluster index

* + - **Block Index Nested Loop Join**
      * 
    - **Sort merge join**
      * Given relations R and S
      * Join condition (R.x == S.x)
      * Assume X is Primary key for S, and Foreign Key for R
      * 
      * 
      * Cost: 
    - **Hash join**
      * Given two relations R and S
      * Given join condition on x (R.x == S.x)
      * R is indexed by x using a hash function h
      * 
      * Cost: Tuples of R + Tuples of S
        + Both R and S should fit into memory
    - **Partitioned Hash join**
      * Given relations R and S
      * Step 1
        + Hash S into M-1 buckets
        + Send all buckets to disk
      * Step 2
        + Hash R into M-1 buckets
        + Send all buckets to disk
      * Step 3
        + Join every pair of buckets
      * 
      * Cost: 3B(R) + 3B(S)
        + Assume min(B(R), B(S)) <= M2
  + 

Query performance:

* Transformation of Relational Expressions
  + Two RA expressions are equivalent if the two expressions generate same set of tuples on every legal database instance
* Cost-based optimization (as we discussed in class)
  + Generate logically equivalent expressions using equivalence rules
    - Join ordering in Join associativity
      * 
      * 
  + Annotate resultant expressions to get alternative query plans
  + Choose the cheapest plan based on the estimated cost
    - Estimated costs based on
      * Statistical information about relations
        + # of tuples, # of distinct values for an attribute
      * Statistical estimation for intermediate results to compute the cost of complex expressions
      * Cost formulae for algorithms, computed using statistics
* Query optimization algorithm
  + Enumerate alternative plants for evaluating the expression
    - Generated alternative plans automatically
    - Decide which physical operators are used for which logical operators
  + Estimate the cost of each enumerated plan
    - Compute I/O cost
    - Compute CPU cost
  + Choose the plan with the least estimated cost
    - Cost-Based optimization

Equivalence rules

*     

Concurrency Control and recovery

* ACID properties
  + Atomicity
    - All actions of a transaction happen, or none happen
      * Commits: All the changes are made
      * Aborts: No changes are made
  + Consistency
    - If a transaction is consistent, and the database starts from a consistent state, then it will end in a consistent state
  + Isolation
    - The execution of one transaction is isolated from other transaction
  + Durability
    - If a transaction commits, its effects persist in the database
      * By writing data to disk
  + Logging and recovery
    - Guarantees atomicity and durability
  + Concurrency control
    - Guarantees consistency and isolation, given atomicity