CS 143 Notes

Indexes
    Sparse - (key, ptr) per block, not every tuple has a (key,pointer) pair in the index
    Dense - every tuple has a (key, ptr) pair in the index
    Sparse/Primary Index/Clustering
        tuples ordered according to search key
        (key, pointer) to first tuple in "block" but not to every tuple in table
    Secondary/Non-clustering Index
        Tuples in table are not ordered by index search key
        Must always have a first level dense index
        Sparse index from second level and above
    - clustered index: rows stored physically on disk in same order as index
    - non clustered index: second list that has pointers to physical rows, can have many non clustered indexes

####################### btree insertion #########################

On Leaf insert overflow:
    Split overflowing node
    copy first key of newly split node to parent non-leaf node

On NonLeaf insert overflow
    Split overflowing node
    Move the key up into parent

A BTreeNode should have at least
    Leaf:
        ceil((n+1)/2) pointers
        ceil(n+1)/2) - 1 keys;

    NonLeaf:
        ceiling(n/2) pointers
        ceiling(n/2) - 1 keys

    Root
        2 pointers, 1 key

####################### btree deletion #########################

BTree Deletion
    Leaf
        No Underflow
            Delete value from leaf
        Coalesce leaf with neighbor
            node underflow, neighboring node not filled up yet
            Move keys from node with underflow into neighboring node (R -> L)
            Delete that node from the tree
            Delete the key of the deleted node in the parent node
        Redistribute leaf with neighbor
            node that is filled up, another underflowed when deleted from
            => overflow caused when merging neighboring nodes

Nonleaf
Coalesce non-leaf with neighbor
after deleting from leaf and from parent, parent underflows
Delete the parent node that is underflowing
Pull down the midkey of the parent's parent, merge midkey with rest of merged sibling non-leaf node
Redistribute non-leaf with neighbor
after deleting from leaf and from parent, parent underflows
Pull down mid key of the parent of underflowing node to merge with sibling non-leaf AND IT OVERFLOWS
do overflow algo by moving new sibling key to parent

If a node is deleted delete the pointer and the adjacent key (to the right) from the above parent node.

Extendible Hash Insertion:
If a hashbucket overflows
If (overflow hashbucket i == directory i)
double directory size copy existing pointers
increase directory i by 1
split overflowing bucket
update directory pointers
increase child hashbuckets by 1

Extendible Hash Merge (after deletion)
Bucket i's must be the same
First (i-1) bits must be the same

Directory Shrink
All bucket i's are smaller than directory i

####################### ER diagram ########################
ER Relationship Notes
Entity -> Rectangle
Relationship -> Diamond
Total Participation -> double Line (entity participates in relationship at least once)

General Cardinality Notation
Along the edge of a relationship
min..max
the entity participates min to max times
- arrow points at the "one" end

Multiple entities can participate in one relationship
eg(Students and TA take classes)

Roles
If a entity participates multiple times in a relationship you can label each edge as a different role
(eg students are partners and partners can take different roles)

Subclasses
Student isa foreign or domestic student
Use a triangle along with subclassing
Subclass participates in the relationships of its superclass
Subclass may participate in its own relationship
Subclass inherits all traits of superclass
Generalization: subclass -> superclass
Specialization: superclass -> subclass
Total specialization: Entity is always one of subclasses (student always either domestic of foreign) -> double line

Weak Entity Sets (WES)
entity sets without keys
Double Rectangle and Double Diamond
Discriminator: set of attributes in WES that are part of the key
- dashed underling
Owner Entity Set (OES): entity set providing part of the key
Identifying Relationship: Link between WES and OES

Converting E/R to Tables
Strong Entity Set: A table with all attributes
Relationship Set: Table contains keys of linked entities and own attributes of relationship
WES: Table with its own attributes and keys from owner ES
Do not need to translate identifying relationship

Subclass
1. Table for each subclass with attributes and superclass key
2. A big Table with all attributes with null for missing ones

####################### Normalization theory ########################
Redundancies in table result in the following anomalies:
Update anomaly
Insertion anomaly
Deletion anomaly

Functional Dependency
- each X value in R is associated with precisely one Y value in R
X -> Y
u1, u2 in R if u1[X] = u2[X] then u1[Y] = u2[Y]
Trivial FD X -> Y and Y contained in X (sid,name -> sid)
Nontrivial FD: X -> Y Y not contained in X (sid -> name, sid, addr)
Completely non-trivial FD: X -> Y but Y has no overlap between X and Y (sid -> name, addr)

Key
X is a key of R iff X+ results in all attributes in R and no smaller subset fulfills the first requirement

Lossless Decomposition occurs if
    R(X,Y,Z) -> R1(X,Y) or R(X,Z)
    if X -> Y or X->Z
Shared attributes is the key of one of the tables

////////////////////////
BCNF Form
    R is in BCNF iff every non-trivial X -> Y, X contains a key
    no redundancy from FD
BCNF Normalization Algorithm
    For any R in schema
    If non-trivial X -> Y holds in R and X does not have the key
        Compute X+
        Split R in R(X+) amd R(X,Y) where Y is elements in R except X+
ex:
ClassInstructor(dept, cnum, title, unit, instructor, office, fax)
instructor -> office
office -> fax
dept, cnum -> title, unit
dept, cnum -> instructor

R1(inst, office, fax) => R3(office, fax) R4(inst, office)
R2(dept, cnum, title, unit, inst)

////////////////////////
Multi-Valued Dependency and 4NF
    X ->> Y
    if for tuples u and v, u[X] = v[X] then there exists a tuple w s.t
    w[X] = u[X] = v[X]
    w[Y] = u[Y]
    w[Z] = v[Z]
- for all X's that are the same with different Y's, the Z's can be swapped
ex: course ->> book, course ->> lecturer

Course	Book	Lecturer
AHA	Silberschatz	John D
AHA	Nederpelt	John D
AHA	Silberschatz	William M
AHA	Nederpelt	William M
AHA	Silberschatz	Christian G
AHA	Nederpelt	Christian G
OSO	Silberschatz	John D
OSO	Silberschatz	William M

Complementation Rule
    X ->> Y then X->>Z where Z is all attributes in R except X and Y
Trivial MVD
    X->>Y is trivial if
    Y in X
    X union Y = R

Fourth Normal Form (4NF)
- R is in 4NF if for every nontrivial FD X->Y or MVD X ->> Y, X contains a key
- no redundancy from FD or MVD
- if 4NF, is BCNF

4NF Decomposition
- For any R in the schema
- If nontrivial X ->> Y holds on R and if X does not contain a key
  - Split R in R1(X,Y) and R2(X,Z) where Z is all attributes in R except (X,Y)

YUS68
ex:
Class(dept, cnum, title, ta, sid, name)
dept, cnum -> title
sid -> sname
dept, cnum -> ta

R1(sid, sname)
R2(dept, cnum, title, ta, sid)
- => R3(dept, cnum, title)
  - R4(dept, cnum, ta, sid)
    - => R5(dept, cnum, ta)
      - R6(dept, cnum, sid)

If X->Y then X ->> Y
Simplified 4NF: R is 4NF it for every nontrivial MVD X ->> Y, X contains a key

###################### Transaction #########################

ACID
- Atomicity
- Consistency
- Isolation
- Durability

Serial Schedule: all operations in any transaction are performed without any interleaving
Conflict Serializable Schedule:
- Non-Conflicting action: A pair of actions that do not change the result when we swap them
- Conflicting (two) actions are categorized by
  - Involve the same objects/variables
  - At least one of the two actions is a write
- Conflict Equivalence
  - S1 is conflict Equivalent to S2 if S1 can rearranged into S2 by a series of swaps of non-conflicting actions
- Conflict Serializability
  - S1 is conflict serializable if it conflict equivalent to to some serial schedule

Precedence Graph P(S)
- P(S) is acyclic <==> S is conflict serializable
- You can only check whether a schedule is conflict serializable not if it is conflict-equivalence
- Algo:

for each action in S
get all actions that affect same current action
draw arrow if at least 1 of pair is a write

Recoverable Schedule

S is Recoverable if Tj reads a data item written by Ti, the commit operation of Ti appears before commit of Tj

Dirty Read: reading a value that has been committed

Cascading rollback: A single transaction abort leads to a series of transaction rollback

Cascadeless Schedule:

S is cascadeless if Tj reads a data item written by Ti, the commit operations appears before the READ of Tj

Cascadeless -> Recoverable

Recoverable DOES NOT imply cascadeless

Relationship Between Schedules

Recoverable contains Cascadeless contains Serializable but Conflict Serializable (around serial) not contained in any of those

We want either Conflict Serializable + Recoverable or Conflict serializable + Cascadeless

/////////////////////////////

Locking Protocol

Rigorous Two Phase Locking Protocol (R2PL)

Rule(1): Ti has to lock tuple tk before any read/write

Rule(2): When Ti is holding the lock on tk, Tj cannot obtain the lock on tk for (j != i)

Rule(3): Release all locks at commit

Rigorous 2PL ensures a conflict-serializable and cascadeless schedule

commit vs time graph: staircase up, at commit, # locks drops to 0 abruptly

Two Phase Locking Protocol (2PL)

Rule(1): Ti lock a tuple before any read/write

Rule(2): If Ti holds the lock on A. Tj cannot access A (j!=i)

Rule(3): Two stages

(a): growing stage: Ti may obtain locks but may not release any lock

(b): shrinking stage: Ti may release lcks, but may not obtain any lock

2PL ensures a conflict-serializable schedule

commit vs time graph: staircase up then down

Conflict Serializable contains 2P,L, 2PL contains R2PL

Recovery and Logging

Log Based Recovery

Log record written to disk before the actual data is written to disk

Before commit of Ti all log records for Ti written to disk including commit

During abort DBMS gets old values from the log

During recovery, DBMS does the following

executes all actions in the log from the beginning

rolls back all actions of "non-committed" transactions in reverse order

Use checkpoint to minimize recovery time

SQL Isolation Levels

Isolation Level	dirty read	nonrepeatable read	phantom
read uncommitted	Y	Y	Y
read committed	N	Y	Y
repeatable read	N	N	Y
serializable	N	N	N

Nonrepeatable read: when Ti reads the same row multiple times, Ti may get different values
Phantom: When new tuples are inserted, some of them are seen by statements
* SET AUTOCOMMIT {0|1}
* SET TRANSACTION READ ONLY, ISOLATION LEVEL REPEATABLE READ

Only when all transactions are serializable, we guarantee ACID
Read only transactions help DBMS optimize for more concurrency

```
###################### Join Algos ########################
Nested Loop Join
     For each r in R
          for each s in S
               if r.C = s.C then output r,s
     COST: b_R + ceil(b_R / (M - 2)) * b_S
     - good for small relations

Index Join
     for each r in R                              -> |R|
          x <- index look up in S (S.c, r.c)      -> C
          for each s in X                         -> J
               write matched tuples

     COST: b_R + |R| * (C + J)
          C - avg index lookup cost
          J - # matching tuples
     - good if index exists already

Sort merge
     Sort stage: sort R and S
          COST (to sort 1 table R): 2b_R * (ceil(log_M-1 (b_R / M)) + 1)
          - don't need output buffer
     Join stage: 2 ptr technique to join based on equality
          COST: b_R + b_S
          - need output buffer
     - good for non-equi-join
     - sorted run: blocks / memory buffer
Hash join
     Hashing stage:
          hash R tuples into Gk buckets
          hash S tuples into Hk buckets
     join stage:
          for i = i to k
               match tuples in Gi, Hi buckets
     COST: 3(b_R + b_S)
     - best for equi-join and if relations not sorted and have no index
```