DBMS

Introduction:

→ Relation is described using 2 things:
(i) Relation schema; (ii) Relation instance

→ Degree/arity; Cardinality;

→ IC: PK; UNIQUE; NOT NULL; Check; Foreign key

→ Key constraints: key/CK; SK; PK;

→ FK constraints: referencing relation - child; refered relation - parent;

- Every value under FK must be under PK.
- violations: deletion/updation from parent; insertion in child.
- on del cascade; on update cascade;
- set null on deletion; set default on del;

ER-Model:

→ entity, entity set; relationship, relationship set;

→ Attributes
- simple, composite
- single val, Multi val
- stored, derived
- key
- Descriptive attribute

→ Relationship Degree: no of entities in a relationship
- Unary/recursive; binary; ternary; n-ary

ER-to-Relational:

→ Composite att - Create multiple columns

→ Multi val att - Create 2 tables (PK + non-multival) (PK + Multi-val)

→ For converting a relationship (□—◇—□) create 3 tables.

→ key constraint: Entity that uniquely identifies relationship (→, ⇒)

□—◇—□ .... 2 tables

→ total participation: (=)

→ atmost one; exactly one; atleast one; 0 or more
(→) (⇒) (=) (—)

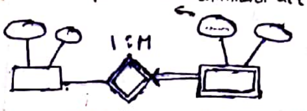
→ Cardinality ratios:

M:N	1:M	1:1	1:M
3 tables	2 table	2 tables	1 tables

A B
PK: {PK of A}

Weak Entity set:

Partial key / Discriminator att



1:M

→ weak entity set: no sufficient att to form key

→ forms key when combined (Partial key + PK of strong entity)

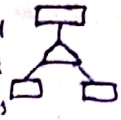
→ 2 tables

→ FK on del Cascade (think y)

Class hierarchy

→ 3 tables without covering constraint

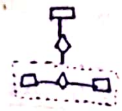
→ 2 tables with covering constraint
if overlapping is present anomalies



spe gen

Aggregation: relationship over relationship

→ 5 tables req;



Functional Dependencies:

→ Trivial FD; Non-trivial FD;

→ Properties:
- Reflexivity: $X \to Y$, $Y \subseteq X$
- Augmentation: $X \to Y \Rightarrow XZ \to YZ$
- transitivity; $X \to Y \wedge Y \to Z \Rightarrow X \to Z$
- Union: $X \to Y \wedge X \to Z \Rightarrow X \to YZ$
- Decomposition: $X \to YZ \Rightarrow X \to Y \wedge X \to Z$

→ closure set of FD: F^+

→ closure set of attributes

→ Additional FDs can be determined frm att. closures.

→ Two FDs are said to be equivalent iff ($F^+ \cong G^+$)
This similar to [(F covers G) and (G covers F)]
↳ Every FD in G can be infered from F.

→ Minimal cover:
(i) Minimize LHS: In $AB \to C$, we can replace it with $A \to C$ if $B \in A^+$
$B \to C$ if $A \in B^+$
(ii) Remove redundant productions by checking if an FD can be obtained from remaining FDs.

→ Finding keys:
- * No of SKs can be found using sum rule.
- * Independent att must be present in CK
  If AX is CK and $Y \to A$ is an FD then YX is also a CK.

Properties of Decomposition:

(i) Loss-less:

If original relationship can be obtained from child relations then it is lossless.

$R_1 \cap R_2$ must be key of R_1 or R_2

R
R_1 R_2

(ii) Dependency Preserving:

$F^+ = (F_1 \cup F_2)^+$

Scanned with CamScanner

Normalization

~~uses~~ only

→ If $X \to A$ is dependency, and A is not prime attribute

if $X \subset CK$ then $X \to A$ is Partial FD

else if A is not prime att and $X \not\subset CK$

then $X \to A$ is transitive FD

1NF: No multivalued att allowed. Only atomic values.

2NF: PFDs not allowed. Only full FDs allowed

3NF: Transitive FDs not allowed.

Every $X \to A$ (non-trivial) is

X is SK (or) A is prime att.

BCNF: Prime transitivity not allowed.

For every $X \to A$, X is SK

3NF Decomposition:

(i) Find minimal cover of F

(ii) Creat a relation for each FD

(iii) If no relation has key, then create a relation with key.

(iv) Remove redundant relations.

BCNF Decomposition:

If $X \to A$ is dependency that violates BCNF property create two reations $R_1(R-A)$ & $R_2(XA)$ and continue this process until all the smaller relations are in BCNF.

→ BCNF ~~relat~~ decomposition may not be dependency preserving.

→ Every all key relation is in BCNF.

Relational Algebra: Proceduceral; gives step by step process.

→ RA & RC eliminates duplicate tuples by default.

→ $\Pi_{att}(\sigma_{cond}\ table\_name)$;

→ ∪, ∩, − {relations must be compatible} ↳ same col & domain

→ Cross product;

→ Join

- Conditional join: $R \bowtie_c S \cong \sigma_c(R \times C)$
- Equi join: $R \bowtie_c S$ & c is equality condition (connected by ∧)
- Natural join: Equi join in which equality is implied.

→ Division: A/B is largest relation instance Q such that $Q \times B \subseteq A$ (A has 2 att, B has 1 att)

→ Rename: $\rho(R(F), E)$; E → relation expression.

R → new name; F → list of names of att (optional)

Relational Calculus: Non-procedural / Declarative.

- TRC: $\{T \mid P(T)\}$; T → tuple variable.
- DRC: $\{\langle d_1, d_2 \ldots d_n\rangle \mid P(d_1, d_2 \ldots d_n)\}$; d_1 to d_n → domain variables

→ ∀, ∃ and connectives (∧, ∨, ⇒, ⇔) can be used within expression P.

→ Unsafe Queries are possible within relational calculus.

→ Relational algebra ≅ safe relational calculus

SQL:

→ Order: from, where, groupby, having, select expressions, distinct, orderby.

→ String enclosed within single quotes.

→ operators: and, or, not | between _ and _ (inclusive) & ally with not

is null, is not null | Like: _, % | <, <=, =, >=, >, <>

in, not in | exists(): true for non empty set

↳ also works with multiple attributes (multicol subquery)

unique(): true when all values are distinct

some

any() & all() Eg: x < any(2,3,4) & x < all(2,3,4)

any(∅) = false; all(∅) = true;

union; intersect; except; (sets must be compatible)

join: from table1 join table2 on condition

natural join: from table1 natural join table2

outer join: used to join all rows even if no matching row

- left outer join (left join)
- right outer join (right join)
- full outer join (full join)

usage: T1 left join T2 on cond (or) T1 natural left join T2

Note: from T1, T2 where T1.att = T2.att(+) → left join. slly we have right join

Clauses:

WHERE: uses above ~~func~~ operators for row selection.

- conditions in where clause should not have aggregate func.

Group by & Having: group by att_list having condition;

→ select & having must contain either aggregate functions or attributes that appeared in group by

→ without group by whole table is considered a single group

Order by: ~~orderby~~ order by col1 desc/asc col2 desc/asc...

→ Think how it works when sorted with multiple attributes

with: used for creating temporary relations.

Syntax: with table_name(att_list) as (expression)

table_name(att_list) as (expression) ...

Aggregate func: MIN(), MAX(), COUNT(), avg(), sum()

→ MIN, MAX, COUNT works with both numerical & strings

→ avg & sum only with numericals. | → COUNT, SUM, AVG can appear with distinct.

→ all agg. func ignore null values.

→ avg() = sum()/count(); avg(dis) = sum(dis)/count(dis)

NULL: Denotes unknown value.
- → All mathematical operations on null result in null.
- → Any comparision with null results in false (except is null)

Sub Query:

(outer (inner))

inner is evaluated once and outer uses its o/p

→ efficient way of writing join operations

Correlated Query

(outer (inner))

→ inner is evaluated for every tuple of outer.

→ inner & outer will have a common attribute

Sub Query & Correlate Query are syntatically same

→ union, intersect & except eliminates duplicate rows.

limit x offset y

File Organization & Indexing

→ Blocking factor = $\frac{\text{block size}}{\text{record size}}$

→ spanned : no wastage; more block access; suits for variable length record

unspanned: less block access; wastage of mem; suits for fixed length records.

→ Index → ordered file

- sparse: built on ordered file; (uses block pointer) no of index records = no of blocks;
- dense: built on unordered; (uses record pointer) no of index records = no of records.

→ Index

- single level: Primary, Clustered, Secondary } avg block access time = $\log_2 Bi + 1$
- Multilevel: BTree, B+ Tree } avg block access = no of lvl + 1

Primary index: built on PK (ordered); no of index records = no of blocks; sparse

Clustered index: (NK + ordered); Each NK value forms a block. no of index rec = no of blocks; sparse

Secondary index: (NK/CK + unordered); suggestable on CK; no of ind rec = no of records; Dense

Multilevel index: 1st level may be dense/sparse.
- → next higher levels are sparse.
- → lvl are continued until last index fits in one block.

BTree: [k P$_r$] [K P$_r$] ... [K P$_r$] order: no of ptrs

$$n(BP) + (n-1)(K + P_r) <= B$$

Provides direct access

B$^+$-Tree:

internal node: $P_1 K_1 P_2 K_2 \ldots K_{n-1} P_n$ order: no of ptrs

$$n(P) + (n-1)K <= B$$

leaf node: [KP] [KP] ... [KP] Pnext → order: no of keys

$$n(K+P) + P_{next} <= B$$

→ In BTree & B$^+$ Tree each node corresponds to a block.

→ All leafs are at same level.

→ If n is order: min ptrs: $\lceil n/2 \rceil$ min keys: $\lceil n/2 \rceil - 1$ — for leaf node of B$^+$-Tree it is $\lceil n/2 \rceil$

→ If order is n (given) in B$^+$-Tree then
- order of internal node : n
- order of leaf node : n-1

→ Refer insertion, deletion in B-Tree & B$^+$-Tree

Transactions & Concurrency Control

Properties: Atomicity, Consistency, Isolation, Durability
- Atomicity ← trans manager
- Consistency ← user & App. programer
- Isolation ← concurrency control manger

Schedule
- Serial: $n!$
- Concurrent: $\frac{(m_1 + m_2 + \cdots + m_n)!}{m_1!\, m_2! \cdots m_n!}$

Problems: All the 3 problem will give diff result from serial execution

Lost update (w-w): $w_i(A) — w_j(A)$ ↳ without reading

Dirty Read (w-R): $w_i(A) — R_j(A)$ ↳ reading uncommited writes

Unrepeatable read (R-w): $R_i(A) \xrightarrow{w_j(A)} R_i(A)$ — diff values are read

Types of schedules:

Non-recoverable: T_j dirty reads frm T_i and commits first

Recoverable: Can be rolled back

NR → Dirty read

No Dirty read → R

Cascading abort → dirty read

Cascading Aborts: one Tx abort cause other Tx to abort

Non Cascading: no cascading roll back

no dirty read ↔ no Cascading abort

Cascadeless → recoverable

Strict schedule: No w-w & No w-R. ∴ recoverable & cascadeless

Serializability:

conflict serializable: conflict equivalent to a serial sche

view serializable: view equivalent to a serial sche ↳ same 1st read, w-R, last write

→ (not CS, but VS) ⇒ blind writes

Precedence graph
- No cycle (CS & VS)
- Cycle → Blind write
  - No (not CS, not VS)
  - Yes → view rules
    - not (not CS, not VS)
    - satisfied (not CS, but VS)

Locking Protocols:

- shared lock, Exclusive lock

2PL: Growing phase & shrinking phase must be present in 2PL.

order of lock points determine order of seriali

- Dead locks are possible
- If no deadlock then ensures CS.
- Edge of precedence graphs are drawn for conflicting locks
- Cycle in predence graph ⇔ Deadlock.
- Cascading rollbacks & irrecoverable schedules are possible.

Strict 2PL

- All exculsive locks are released only after commit operation.
- There will be no ww and wR conflicts

Rigorous 2PL

- All locks are released only after commit operation
- No ww, wR, Rw conflicts ~~possible~~.

- Both strict 2PL & rigrous 2PL are free from cascading rollbacks and hence recoverable. but deadlocks possible

Conservative 2PL:

→ All the locks are held at once and released after commit

∴ This is free from cascading rollbacks, recoverable & deadlock free

Deadlock Handling:

(i) Deadlock prevention:

- wound-wait: older wounds, younger waits
- wait-die: older waits, young dies.

- In both, unnecessary rollbacks are possible.
- In both, restarted Tx is started with same time stamp.

(ii) Deadlock ~~Dead~~ Detection:

→ Here a wait for graph is maintained.

→ whenever a conflicting lock is req, an edge is added and removed when lock is acquired.

→ Deadlock ⇔ cycle.

Timestamp based protocols:

- Every schedule is under this protocol is equivalent to serial schedule T_i, T_j such that $TS(T_i) < TS(T_j)$
- This is deadlock free.

2 types:

(i) Time stamp Ordering Protocol:

- Here all conflicting operations are executed in the order of their time stamp.
- If an conflicting operation is performe out of order then that Tx will be aborted and rolled back and restarted with new time stamp.
- ∴ starvation is possible.
- Ensures CS & deadlock free.

(ii) Thomas write rule:

- This is modification of TOP that minimizes the problem of starvation by avoiding obsolete write.

i.e., writes conflicting with writes of new Txs.

$w_i(A)$ is called obsolete write, if $w_i(A)$ occurs after $w_j(A)$ and $T_i < T_j$

- So here we just ignore $w_i(A)$
- TWR ensures VS.

→ Cascading rollbacks & irrecoverable schedules are possible under TOP & TWR.

TWR ⊃ TOP

Multiple Granularity:

→ If a child is locked, it's ancestor can't be locked in conflicting mode. ||ly if parent is locked child can be locked in conflicting mode.

→ locks: S, X, IS, IX, SIX

→ Multiple granularity 2PL: locking: root to leaf
unlocking: leaf to root.

Implementation of TOP:

- Every Tx has timestamp $TS(T_i)$
- Every data item A is given two timestaps (read TS & write TS)

RTS(A): highest TS of any Tx that performe read on A.

WTS(A): highest TS of any Tx that performed write on A.

Read operation on A by Tx T:

if $TS(T) \geq WTS(A)$ then read can be performed otherwise abort T and restart with new TS

write operation on A by Tx T.

if $TS(T) \geq WTS(A)$ & $TS(T) \geq RTS(A)$ then write can be performed on A. otherwise abort T and restart with new TS

If $TS(T) < WTS(A)$ & $TS(T) \geq RTS(A)$ then it is not possible under TOP but possible under TWR

~~only if TS(T) ≥ WTS(A) and TS(T) < RT~~