GATE CSE NOTES

by

Joyoshish Saha

* Topics.

→ ER Diagram
→ Conversion of E-R Diagram into relational model
→ Basics of relational model & functional dependency
→ Idea about keys / types
→ Normalization (INF – BCNF)
→ Lossless decomposition & dependency preserving
→ Indexing & Physical Structure (B, B^+ tree)
→ SQL, R.A., R.C. [Relational algebra, calculus]
→ Transition
→ Concurrency Control.

* Data : Raw & isolated facts about an entity (recorded)

Information : Processed, meaningful, usable data.

Database : Collection of similar / related data

DBMS : S/W used to create, manipulate & delete database.

* Disadvantages of File System.

1. Data redundancy
2. Data inconsistency
3. Difficulty in access
4. Data isolation
5. Security problem
6. Atomicity problem
7. Concurrent access anomalies
8. Integrity problem.
9. Representing complex relationships among data
10. Providing multiple user interfaces.

* OLAP

Online analytical processing

→ historical data
→ subject oriented
→ decision making
→ size in TBs & PBs
→ dealt by CEO, MD, GM.
→ only read

OLTP

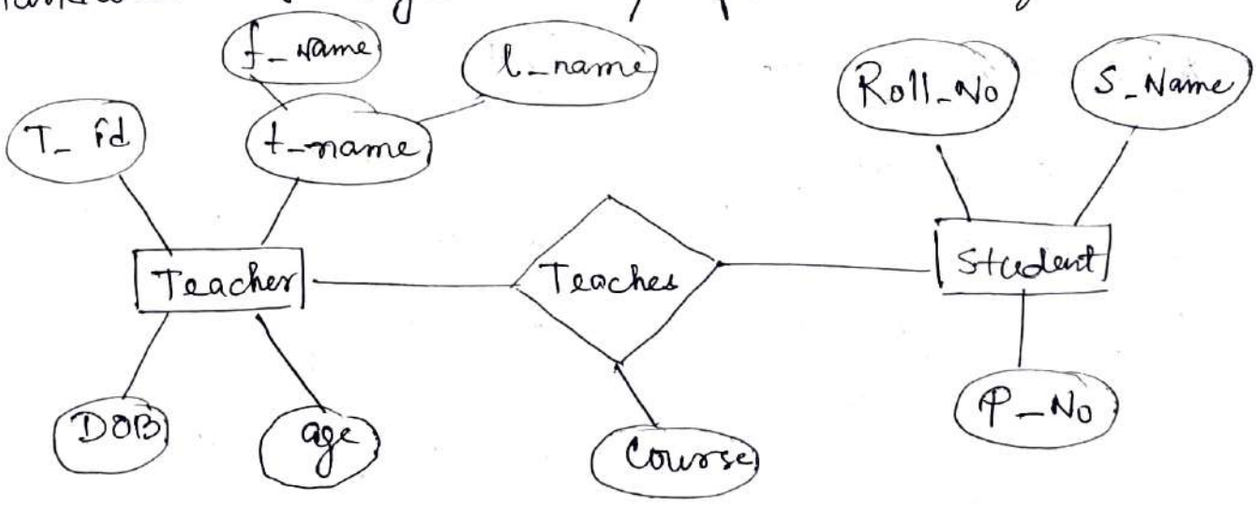
Online transaction processing

→ current data
→ application oriented
→ day to day operations
→ size in MBs, GBs
→ dealt by clerks, managers.
→ read/write.

* Entity Relationship Diagram (ER Diagram)

(Intro, Entity & entity set, attributes & types, relationship, strong & weak entity set, traps, conversion)

Non-technical design method that works on conceptual level based on the perception of real world. Consists of collection of basic objects called entities & of relationships among these objects & attributes that define their properties. Free from ambiguities & provides a standard & logical way of visualising data.



Entity: An entity is a thing or an object in the real world that is distinguishable from other objects based on the values of the attributes it possesses.

Types of entities

→ **Tangible** : Entities which physically exists in real world.
eg. car, pen, bank locker.

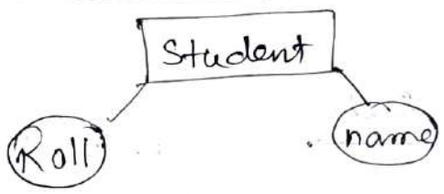
→ **Intangible** : Entities which exist logically.
eg. bank account

Entity set : Collection of same types of entities i.e. that share same properties or attributes called entity set.

ER Diagram

→ Entity can't be represented in an ER diagram as it is instance / data.

→ Entity set is represented by rectangle in an ER diagram.



Relational model

→ Entity can be represented in an relational model by row / tuple / record.

→ Entity set is represented by table in relational model.

Student

Name	Roll
A	1
B	2
⋮	⋮

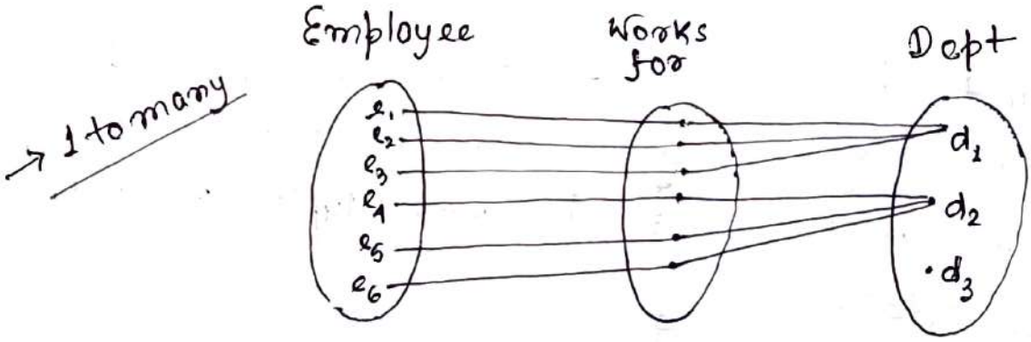
* Attributes.

Classifications –

1. Composite & Simple
2. Single valued & multi valued.
3. Stored & derived.
   (DOB) (Age ← DOB)
4. Complex attributes (composite & multivalued)

* Requirement analysis

Every employee works for exactly a dept., & a dept. can have many employees. New dept. need not have any employee.

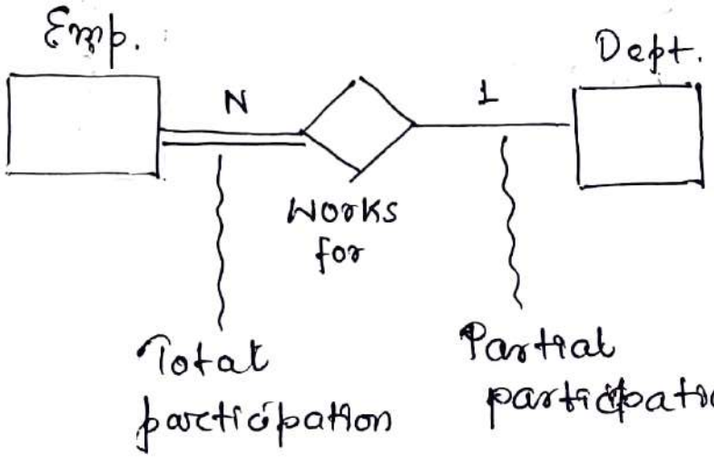


→ Degree = 2 (2 entities)

→ Cardinality ratio (Max) = 1e Nd

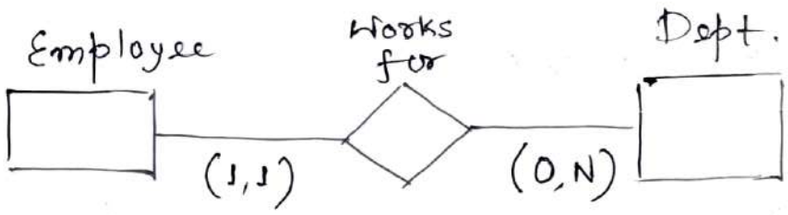
→ Participation or existence (Min) = 1e 0d

e - employee
d - dept.



Cardinality ratio/ single-double line representation

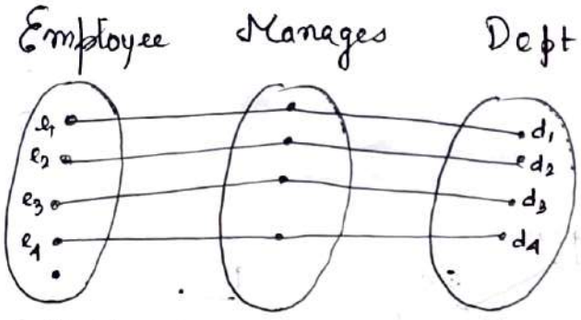
→ 1 to many relationship.



min-max representation.

→ One to one relationship.

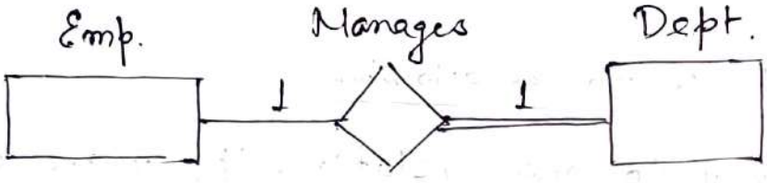
Every dept should have a manager & only one employee manages a dept.

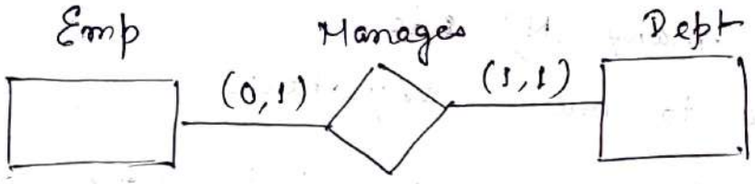


Degree 2.

Cardinality 1 1

Participation 0 1





→ Many to many relationship

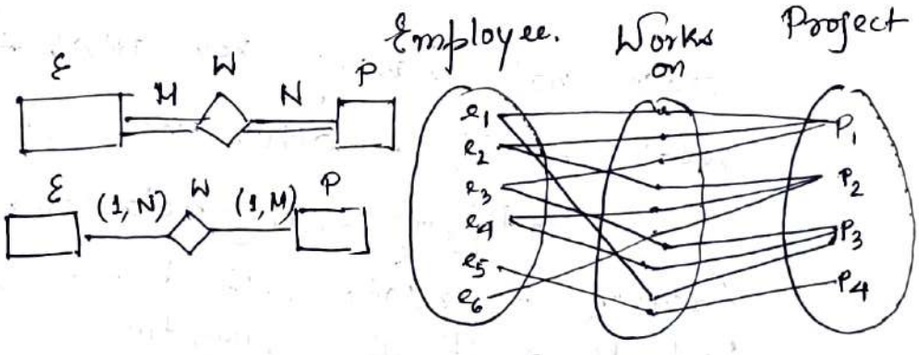
Each employee works on atleast 1 & at max all projects. Each project can have at least 1 & at max all employees.

P_1 : e_1, e_2, e_3

P_2 : e_2, e_4, e_6

P_3 : e_3, e_4, e_1

P_4 : e_5

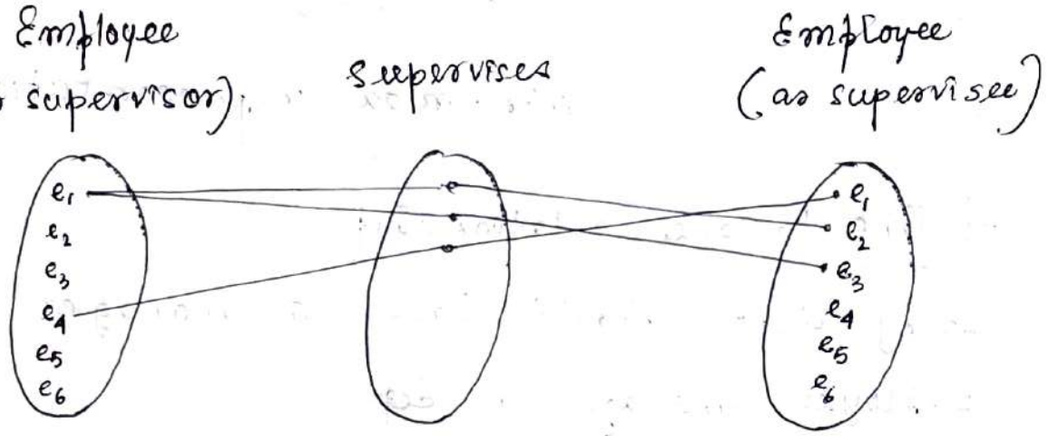


Degree 2

Cardinality N M

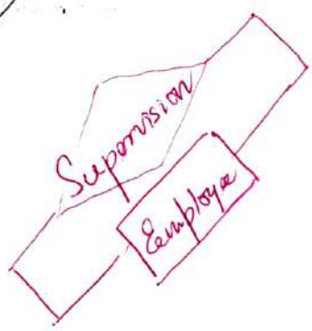
Participation 1 1

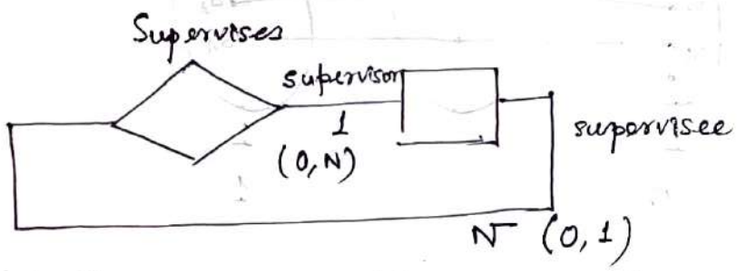
→ Recursive Relationships.



Degree 2

Cardinality	N	1
Participation	0	0





→ Attributes to relationship.

For many to one relationship, move attribute to the many (N) side.

For one to one, anywhere.

For many to many, shouldn't move attribute to entities. Keep it to relationships.

* Weak Entity.

Key attribute — uniquely identifies the entities.

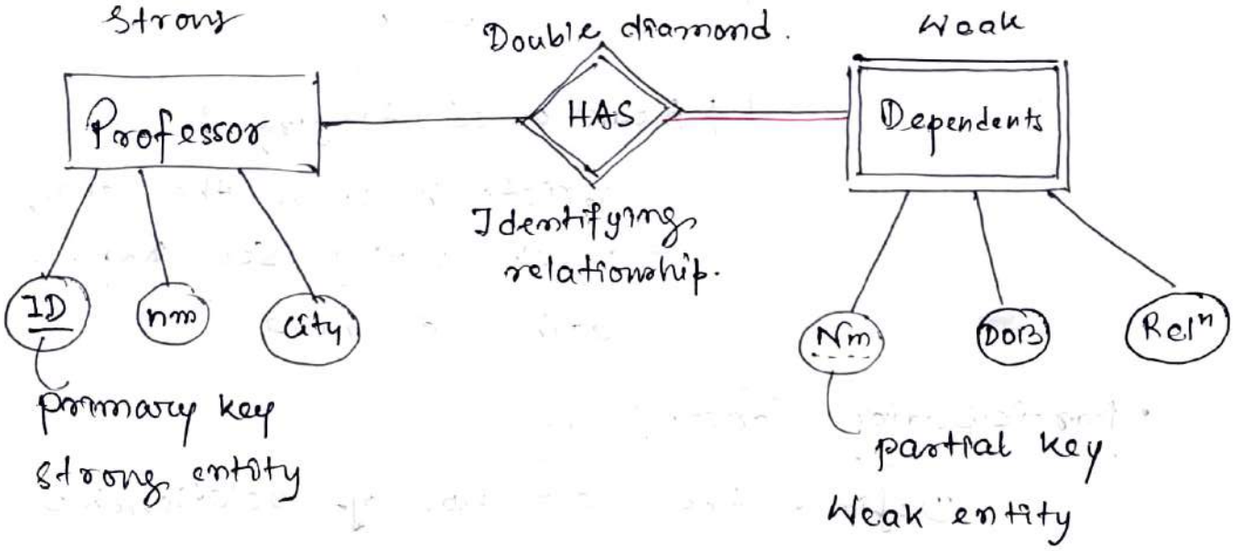
Entity not having key attributes is called weak entity, otherwise strong entity.

Identifying entity relationships are used to identify a relationship between strong & weak entity.

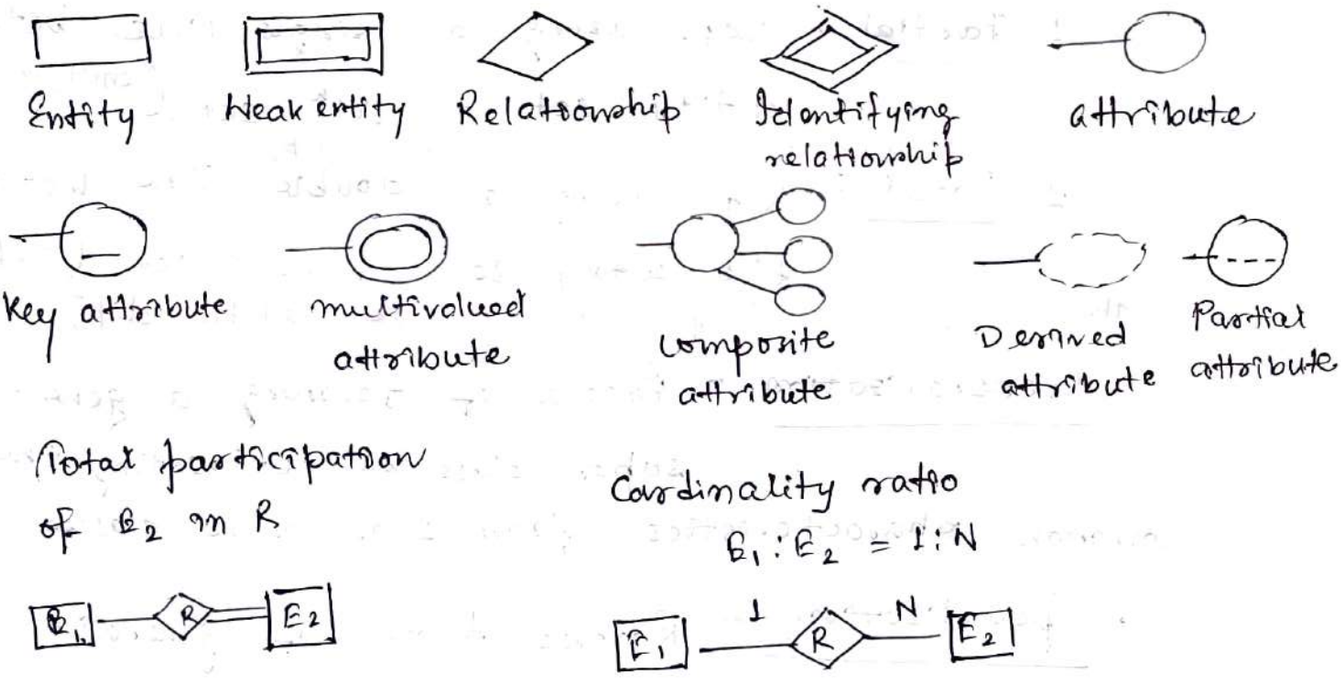
Participation of weak entity type is always total. Relationship between weak entity type & its identifying strong entity type is called identifying relationship (double diamond).

eg. Professor — Strong entity

Professor-dependents — Weake entity



ER diagram notations.



Total participation of E_2 in R

Cardinality ratio

$E_1 : E_2 = 1 : N$

- Components of ER Diagram.

i) Entity sets ii) Attributes iii) Relationship set

- Entity set → Strong : Possesses its own primary key. Represented using a single rectangle.
  → Weak : Does not possess own primary key. Rep. using a double rectangle.

- Relationship sets → Strong: Strong relationship exists between 2 strong entity sets. (Diamond symbol)
  → Weak or Identifying: Exists between the strong & weak entity set. Rep. using double diamond.

- Participation Constraints. (min)

Defines the least no. of relationship instances in which an entity has to necessarily participate.

1. Partial: Rep. using a single line between entity set & rel^n set. (entity in the entity set may or may not participate in the relationship)
2. Total: Rep. using a double line between the entity set & rel^n set. (each entity in the entity set must participate in the relationship)

- Generalisation. - Process of forming a generalised super class by extracting the common characteristics from 2 or more classes.

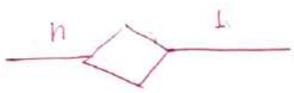
- Specialization. - Reverse process of generalization where a super class is divided into sub classes by assigning the specific characteristics of subclasses to them.

- Cardinality Constraints / ratios. (max)

Defines the max. no. of relationship instances in which an entity can participate.

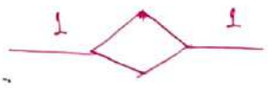
i) many to many (m:n)

ii) many to one (m : 1)



iii) one to many (1 : n)

iv) one to one. (1 : 1)



• A weak entity set is an entity set that does not contain sufficient attributes to uniquely identify its entities. It contains a partial key called discriminator. Discriminator can identify a group of entities from the entity set.

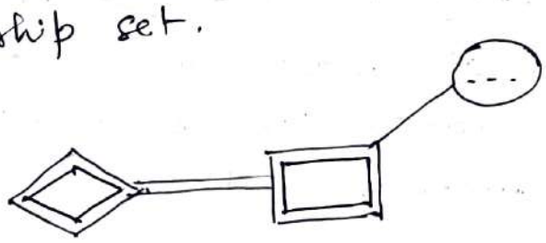
The combination of discriminator & primary key of the strong entity set makes it possible to uniquely identify all entities of the weak entity set.

Primary key of weak entity set	=	Its own discriminator	+	Primary key of strong entity set.

• Total participation may or may not exist in the strong entity set relationship.

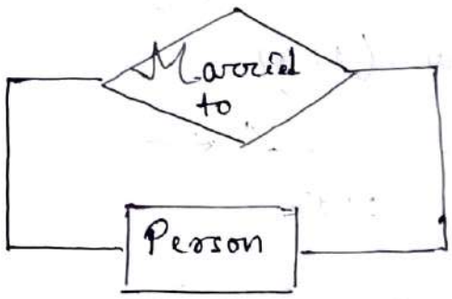
Always exists in the identifying relationship.

So, in ER diagram, weak entity set is always present in total participation with the identifying relationship set.



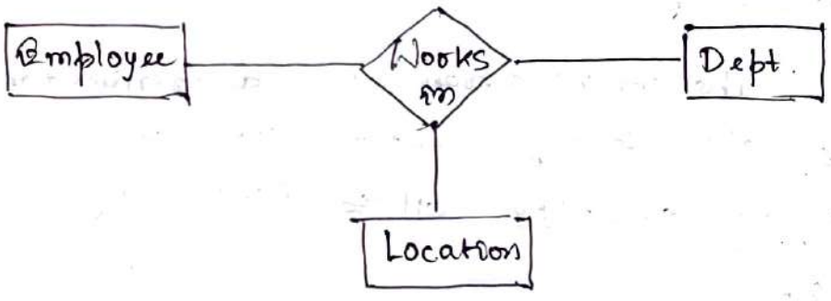
• Degree of a relationship set. : No. of entity sets participating in a relationship set.

- Unary relationship: One person married to only one person.



Binary relationship: Student enrolled in a course.

Ternary relationship:



- Cardinality Constraints: Max. no. of relationship instances in which an entity can participate.

i) m:n : An entity in set A can be associated with any no of entities of set B. Same for B.

ii) m:1 : An entity in set A can be associated with at most one entity in set B. An entity in set B can be associated with any no. of entities in set A.

iii) 1:m : Opposite of m:1.

iv) 1:1 : An entity in set A can be associated with at most one entity in set B. Same for B.

- Participation constraints.

Least no. of relationship instances in which an entity must compulsorily participate.

i) Total participation : Each entity in the entity set must compulsorily participate in at least ~~once~~ one relationship instance in the relationship set. (mandatory participation)

ii) Partial participation : Each entity ~~set~~ in the entity set may or may not participate in the relationship instance in that relationship set. (optional participation)

(i.e participation).

- Minimum cardinality tells whether the participation is partial or total.

- If minimum cardinality = 0, then it signifies partial participation.

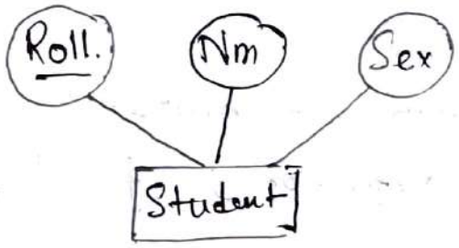
- If minimum cardinality = 1, then it signifies total participation.

Maximum cardinality tells the max. no. of entities that participates in a relationship set.

* Converting ER diagram to tables.

ER diagram is converted into the tables in relational model. This is because relational models can be easily implemented by RDBMS like MySQL, Oracle etc.

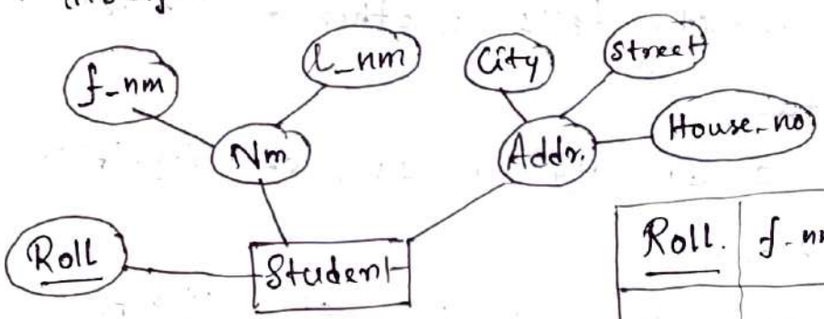
Rule 1: A strong entity set with only simple attributes will require only one table in Relational model. Attributes of the table will be the attributes of entity set. The primary key of the table will be the key attribute of the entity set.



Roll	Nm	Sex

Schema: Student

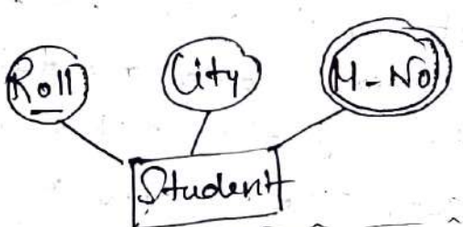
Rule 2. A strong entity set with any no. of composite attributes will require only one table in relational model. While conversion, simple attributes of the composite attributes are taken into account & not the composite attribute itself.



Roll.	f_nm.	l_nm	city	street	house_no

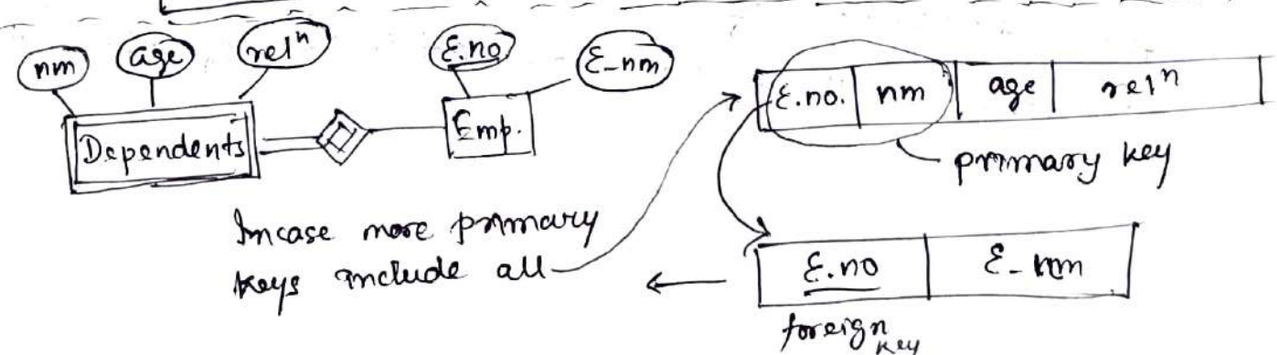
Schema : Student.

Rule 3. A strong entity set with any no. of multivalued attributes will require 2 tables in relational model. One table will contain all the simple attributes with the primary key. Other table will contain the primary key & all the multivalued variables.



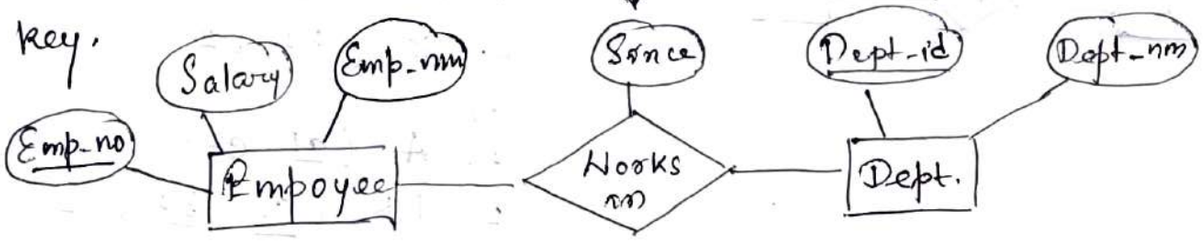
Roll	City

Roll	H_No



Rule 4. Translating relationship set into a table.

A relationship set will require one table in the relationship model. Attributes of the table are primary key attributes of the participating entity sets, its own descriptive attributes if any. Set of non-descriptive attributes will be the primary key.



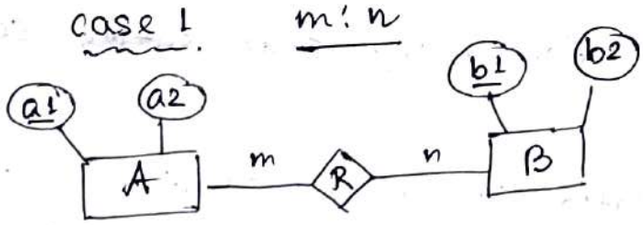
Emp_no	Dept_id	since

Schema: Works in

If we consider the overall ER diagram, three tables will be reqd. in relational model. One table for entity set 'Employee', one for 'Dept', one for rel. set 'Works in'.

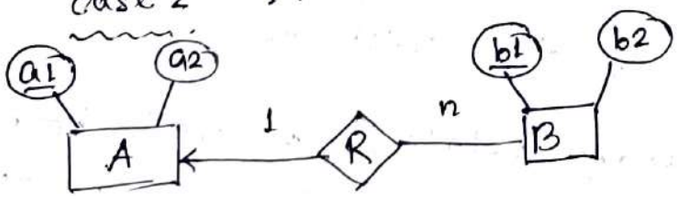
Rule 5. For binary relationships with cardinality ratios

case 1 m:n



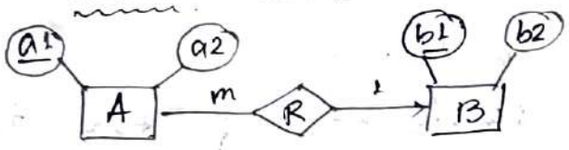
Three tables –
A (a1, a2)
R (a1, b1)
B (b1, b2)

case 2 1:n



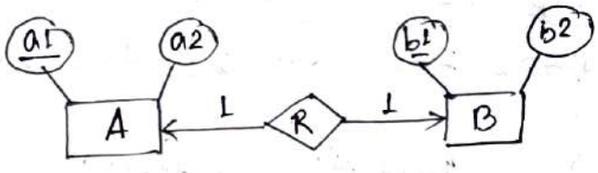
2 tables –
A (a1, a2)
BR (a1, b1, b2)
combined table

case 3 m:1



2 tables –
1) AR (a1, a2, b1)
2) B (b1, b2)

case 4 1:1



2 tables
1. AR (a1, a2, b1)
2. B (b1, b2)

or
1. A (a1, a2)
2. BR (a1, b1, b2)

• While determining the minimum number of tables reqd. for binary relationships with given cardinality ratios, following thumb rules must be kept in mind –

→ For binary relationship with cardinality ratio m:n, separate & individual tables will be drawn for each entity set & relationship.

→ For binary relationship with cardinality ratio either m:1 or 1:n, always remember many side will consume the relationship. i.e. a combined table will be drawn for many side entity set & relationship set.

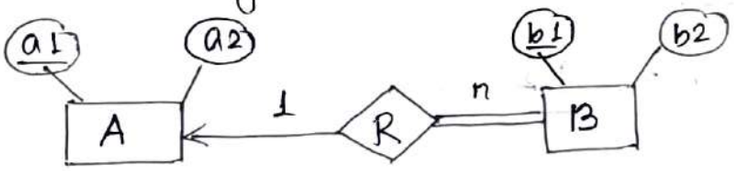
→ for binary relationship with cardinality ratio 1:1, two tables will be reqd. Combine the relationship set with any one of the entity sets.

Rule 6 For binary relationship with both cardinality constraints & participation constraints –

Cardinality constraints will be implemented as discussed in rule 5. Because of the total participation constraint, foreign key acquires NOT NULL constraints. i.e. now foreign key cannot be null. (Foreign keys are

the columns of a table that points to the primary key of another table. They act as a cross-reference between tables.)

case-1 For binary relationship with cardinality constraint & total participation constraint from one side -



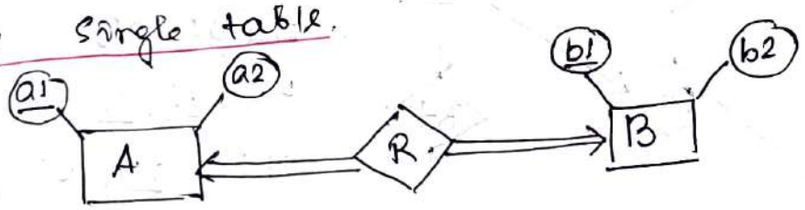
Because cardinality ratio = 1:n, so we will combine the entity set B & relationship set R. Two tables reqd. -

1. A (a1, a2)
2. BR (a1, b1, b2)

Because of total participation, foreign key a1 has acquired NOT NULL constraint, so it can't be null now.

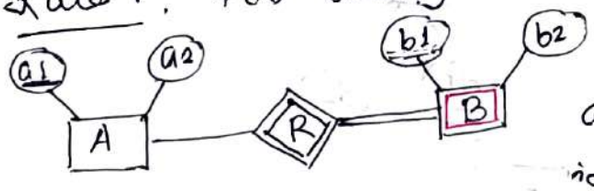
case-2 for binary relationships with cardinality constraint & total participation constraint from both sides -

If there is a key constraint from both the sides of an entity set with total participation, then that binary relationship is represented using only single table.



ARB (a1, a2, b1, b2)

Rule 7. For binary relationship with weak entity set.



Weak entity set always appears in association with identifying relationship with total participation constant.

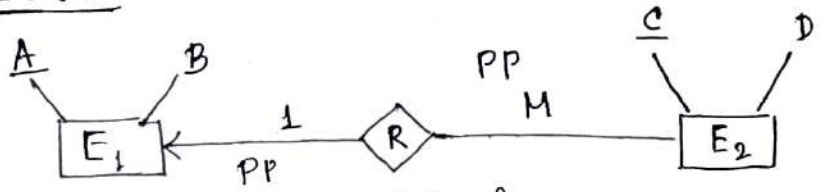
2 tables reqd.

i) A (a1, a2)

ii) BR (a1, b2, b1)

Minimum RDBMS.

① 1 : MC



(A C)

C → A

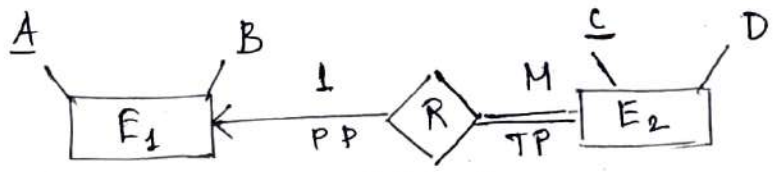
C → D.

E_1 (A B) E_2 R (C D A) [FK, should allow NULL]

Min 2 tables & 1 FK.

---

② 1 : M



R (A C)

C → A

E_1 (A B) E_2 R (C D A) [No null]

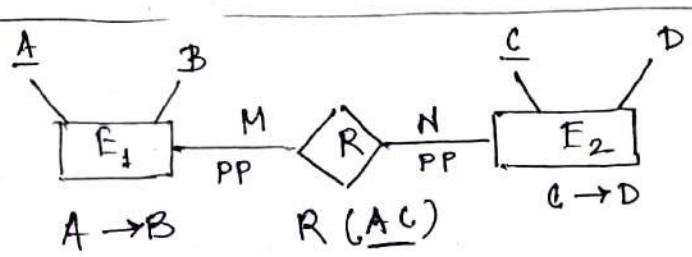
A → B

C → DA

Min 2 tables & 1 FK.

---

③ M : N



A → B

R (A C)

No nontrivial FD

C → D

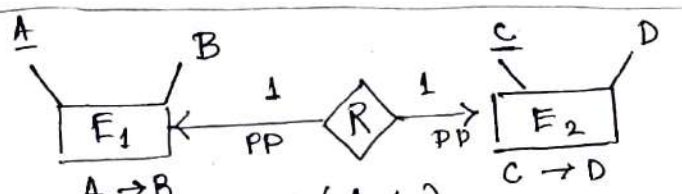
E_1 (A B) R (A C) E_2 (C D

Min 3 tables

2 FKs.

---

④ 1 : 1



A → B

R (A, C)

A → C, C → A

C → D

(i) E_1 R (A B C) E_2 (C D)

A → B

C → A

C → D.

(ii) E_1 (A B) E_2 R (C D A)

A → B

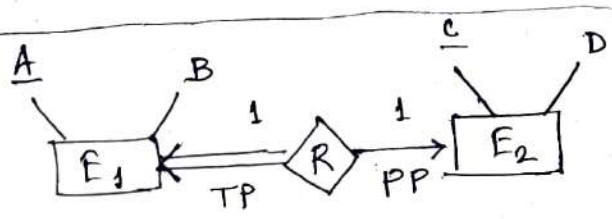
C → DA

A → C

Min 2 tables, 1 FK.

---

⑤ 1 : 1



E_1 R E_2 (A B C D)

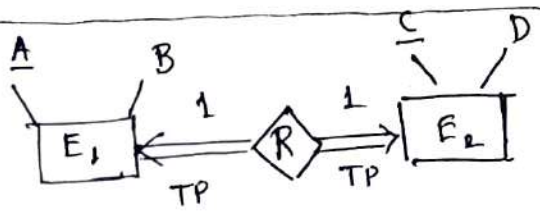
A → BC

C → AD

Min 1 table, no FK.

---

⑥ 1 : 1



E_1 R E_2 (A B C D)

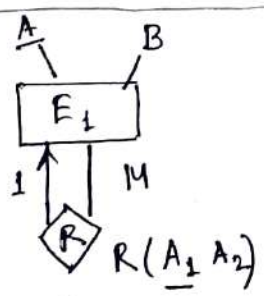
A → BC

C → AD

Min 1 table, no FK.

---

⑦ 1 : M



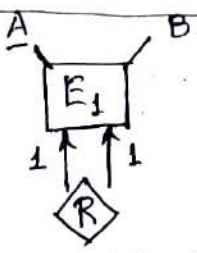
R (A1 A2)

E_1 R (A B A1)

Min 1 table, 1 FK.

---

⑧ 1 : 1



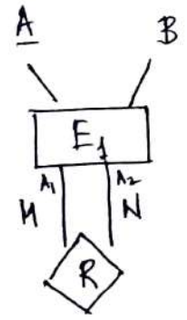
R (A1, A2)

E_1 R (A B A1) {A, A1} candidate key

Min 1 table,

1 FK.

(9)



M:N

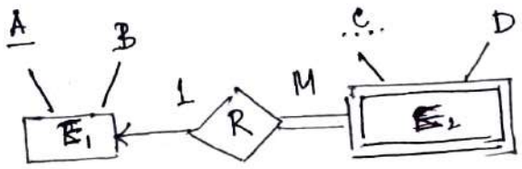
E_1 ($\underline{A}$ $\underline{B}$) R ($\underline{A_1}$ $\underline{A_2}$)

Min 2 tables

2 FKs

---

(10)



1:M

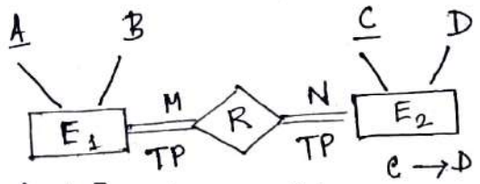
E_1 ($\underline{A}$ B) E_2 R ($\underline{A}$ C D)

Min 2 tables, 1 FK.

---

(11)

m:n



R ($\underline{A}$ $\underline{C}$)

No nontrivial FD

Min Rel^n for 1NF

E_1 R E_2 (A B C D)

A → B C → D

$\underline{A}$ $\underline{C}$ not 2NF

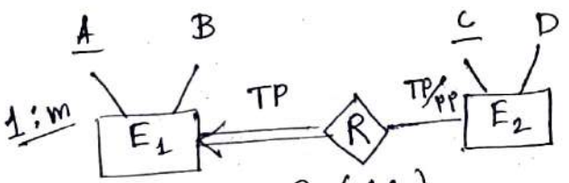
Min Rel^n for BCNF

E_1 ($\underline{A}$ $\underline{B}$) R ($\underline{A}$ $\underline{C}$)

A → B

E_2 ($\underline{C}$ D)

C → D



R ($\underline{A}$ $\underline{C}$)

E_1R (A B C D)

A → B, C → A D

$\underline{C}$: Key

1NF ✓

2NF ✓

3NF X

E_1 ($\underline{A}$ B) A → B

E_2 R ($\underline{C}$ A D) C → A D

- For not applicable or unknown values in table we set that NULL.

* Relational Constraints.

Restrictions imposed on the database contents & operations. They ensure the correctness of data in the database.

Types –

1) Domain constraint: Value taken by the attribute must be the atomic value from its domain.

2) Tuple uniqueness constraint: All the tuples must be necessarily unique in any relation.

3) Key constraint: All the values of primary key must be unique.
The value of primary key must not be NULL.

4) Entity integrity: No attribute of primary key ~~must~~ should contain a null value in any relation. Presence of null value in the primary key violates the uniqueness property.

5) Referential integrity constraint:
This constraint is enforced when a foreign key references the primary key of a rel^n. It specifies that all the values taken by the foreign key must either be available in the relation of the primary key or be null.
(Foreign key must reference a valid, existing primary key in the parent table.)

• We can't insert a record into a referencing relation if the corresponding record does not exist in the referenced rel^{n}.

• We can't delete or update a record of the referenced relation if the corresponding record exists in the referencing rel^{n}.

• Referencial Integrity Constraint Violation :

Cause-1. Insertion in a referencing relation –

It is allowed to insert only those values in the referencing ~~relationship~~ attribute which are already present in the value of the referenced attribute.

Inserting a value in the referencing attribute which isn't present in the value of the referenced attribute violates the referential integrity constraint.

Cause-2 Deletion from a referenced relation –

It is not allowed to delete a row from the referenced relation if the referencing attribute uses the value of the referenced attribute of that row. Such deletion violates referential integrity constraint.

Handling this violation –

method 1. Simultaneous deleting those tuples from the referencing relation where the referencing attribute uses the value of referenced attribute being deleted. (On delete cascade)

method 2. Aborting or deleting the request.

method 3. Setting the value to NULL or some other value in the referencing relation if the referencing attribute uses the value.

Cause-3. Updation on a referenced relation.-

It's not allowed to update a row of the referenced relation of the referencing attribute uses the value of the referenced attribute of that row. Such updates violate referential integrity comtraint.

Handling this violation-

1. Simultaneous update of those tuples of the referencing relⁿ where the referencing attribute uses the referenced attribute value being updated. (On update cascade)

2. Aborting or deleting the request.

3. Setting the value to NULL or some other value.

*Keys. A key is a set of attributes that can identify each tuple uniquely on the given relation.

| Relation ≡ Table
| Tuple ≡ Record

Different keys -

1. Super key: Set of attributes that can identify each tuple uniquely on the given relation. A super key is not restricted to have any specific no. of attributes. A super key may consist of any no. of attributes.

✓ All the attributes in a super key are definitely sufficient to identify each tuple uniquely on the given relation but all of them may not be necessary.

Any superset of a key is a super key.

Every relⁿ must have a superkey & in the worst case, whole superkey will be the key.

2. Candidate key : A minimal super key is called as a candidate key. This is a set of minimal attribute(s) that can identify each tuple uniquely in the given rel^n.

All the attributes in a candidate key are sufficient as well as necessary to identify each tuple uniquely. Removing any attribute fails in identifying each tuple uniquely. The value of candidate key must always be unique. The value of candidate key can never be NULL. It's possible to have multiple candidate keys in a relation. Those attributes which appear in some candidate key are called as prime attributes.

There can be multiple CKs for a single rel^n.

eg. Student (roll, name, sex, age, addr., class)

Super keys examples –
(roll, name, sex)
(roll, age, addr.) ... etc.

No. of candidate keys = $2^n - 1$ (possible) [when n attrib.s]

Candidate keys examples –
(class, section, roll)
(name, addr.)

each set consists of minimal attrib.s reqd. to identify each student uniquely

3. Primary Key : A candidate key that the database designer selects while designing the db.

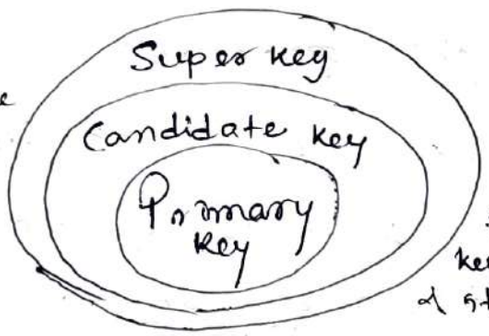
The value of primary key can never be NULL. The value of primary key must always be unique. Values of primary key can never be changed i.e. no updation possible. Value of primary key

In a single rel^n, different candidate keys may have some different set of attributes (like A, and Be) N=1

must be assigned when inserting a record.

A rel^n is allowed to have only one primary key.

There can only be one primary key for any rel^n in a schema.



One of the candidate keys gets qualified to become a primary key. Rules that a CK must have to become a PK are that the key value should never be null & it must be unique for all tuples.

4. Alternate Key: Candidate keys that are left unimplemented or unused after implementing the primary key are called as alternate keys.

5. Foreign Key: An attribute X is called a foreign key to some other attribute Y when its values are dependent on the values of Y.

The attrib. X can assume only those values which are assumed by Y.

The rel^n in which Y is present is called the referenced rel^n. The rel^n in which X is present is called the referencing rel^n. The attribute Y might be present in the same table or in some other table.

eg.

Teacher (t_no, t_nm, t_age, t_dept)

Department (dept_no, dept_nm)

t_dept → dept_no

Referencing rel^n — foreign key — Referenced rel^n.

t_dept can take only those values that are present in dept_no in department table.

Foreign key references the primary key of the table. Foreign key can take only those values that are present on the primary key of the referenced rel^n. Foreign key may have a name other than that of a primary key. Foreign key can take the NULL value. There is no restriction on a foreign key to be unique. Foreign key is not unique most of the time. Referenced rel^n may also be called as the master table or primary table. Referencing rel^n may also be called the foreign table.

6. Partial Key : A key using which all the records of the table cannot be identified uniquely. However, a bunch of related tuples can be selected from the table using partial key.

eg. Dept. (Emp_no, dependent_nm, relation)

Emp_no	dependent_nm	reldtion
e1	Suman	Mother
e1	Ajay	Father
e2	Bijay	Father
e2	Raj	Son

Using partial key Emp_no, we can't identify a tuple uniquely but we can select a bunch of tuples from the table.

7. Composite Key : A primary key comprising of multiple attribs & not just a single attrib.

8. Unique Key: Key with following properties-

- It's unique for all the records of table.
- ✓ Once assigned, its value can't be changed (non-updatable).
- ✓ It may have a NULL value.

eg. Aadhaar card numbers

9. Surrogate Key: Key with properties —

- It's unique for all the records.
- ✓ It's updatable.
- ✓ It can't be NULL.

eg. mobile no. of students on a class where every student owns a mobile phone.

10. Secondary Key: Required for the indexing purpose for better & faster searching.

Q. Schema $R(A_1, A_2, \ldots, A_n)$.

Candidate key $\{A_1\}$.

How many super keys possible?

→ Every superset of a key is a superkey.

A_2 A_3 $\ldots$ A_n

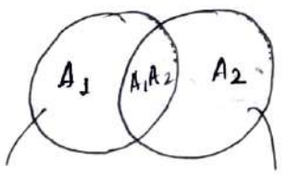
$2 \times 2 \times \ldots \times 2$ $n-1$ times

$= 2^{n-1}$ no. of superkeys.

Q. $R(A_1, A_2, \ldots, A_n)$ Relation.

Candidate key - $\{A_1, A_2\}$

How many superkeys ?

→ 

Superkeys for A_1 | Superkeys for A_2

$SK(A_1) + SK(A_2) - SK(A_1A_2)$

$= 2^{n-1} + 2^{n-1} - 2^{n-2}$

Q. $R(A_1, A_2, \ldots, A_n)$

$CK = \{A_1, A_2A_3\}$ No. of superkeys ?

→ $SK(A_1) + SK(A_2A_3) - SK(A_1A_2A_3)$.

$= 2^{n-1} + 2^{n-2} - 2^{n-3}$

Q. $R(A_1, A_2, \ldots, A_n)$

$CK = (A_1, A_2, A_3)$

No. of superkeys.?

→ $SK(A_1) + SK(A_2) + SK(A_3) - SK(A_1A_2) - SK(A_2A_3) - SK(A_3A_1) + SK(A_1A_2A_3)$

Q. $R(A, B, C, D)$

$CK = \cancel{(A_2BC)}$ (A, BC)

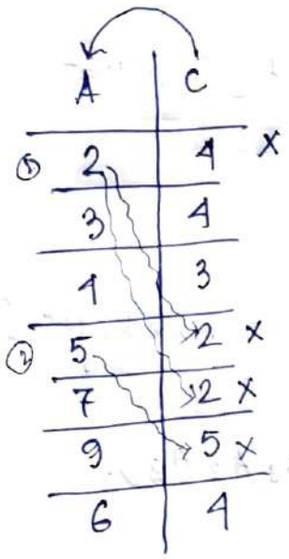
No. of superkeys?

→ $8 + 4 - 2$

$= 10.$

eg. A is the primary key & C is the foreign key referencing A with on delete cascade. The set of all tuples that must be additionally deleted to preserve referential integrity when (2,4) is deleted —

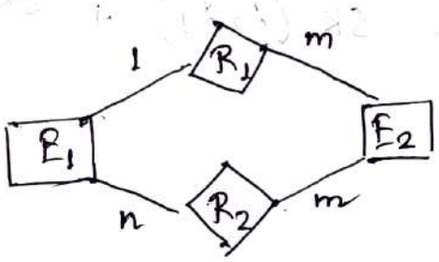
(5,2), (7,2), (9,5)



(On delete cascade in case of foreign keys)

Q. E_1, E_2 two entities. R_1, R_2 are 2 relationships b/w E_1 & E_2. R_1 is one to many & R_2 is many to many. R_1 & R_2 does not have any attributes of their own. What is min. no. of tables reqd.?

→



E_1, E_2 have simple single valued attrib.s. (assume)

E_1
E_2 R_1
1 table for R_2 (min)

} 3 tables

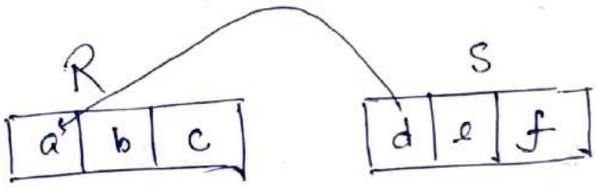
8.

Let R(a,b,c) & S(d,e,f) be two relations. 'd' is a foreign key of S that references to the primary key of R. Consider following operations on R & S —

a) Insert into R
c) Delete from R
b) Insert into S
d) Delete from S

Which of these can cause violation of referential integrity?

Assume 'a' to be primary key.



* Functional Dependency.

In any relation, a functional dependency $\alpha \rightarrow \beta$ holds if two tuples having same value of attribute α also have same value for attribute β.

If α & β are 2 sets of attribs in a relational table R where $\alpha \subseteq R$ & $\beta \subseteq R$, then for a functional dependency to exist from α to β, following condⁿ must be true

$$\text{if } t1[\alpha] = t2[\alpha], \text{ then } t1[\beta] = t2[\beta]$$

$$f_d : \alpha \rightarrow \beta.$$

- FD is a relationship that exists between 2 attributes. It typically exists between the primary key & non-key attribute within a table.

- $X \rightarrow Y$

X: determinant
Y: dependent

- FD

Trivial: $AB \rightarrow A$, $A \rightarrow A$

Non-trivial: $A \rightarrow B$ (Non trivial)

Semi-non-trivial: $AB \rightarrow BC$

→ Trivial Functional Dependency.

An FD $X \to Y$ is said to be trivial if & only if $Y \subseteq X$. If RHS of FD is a subset of LHS, it is trivial.

$AB \to A$, $A \to A$.

→ Non-trivial functional dependency

$X \to Y$ is non-trivial only if $Y \not\subseteq X$.

If there exists at least one attrib. on the RHS of a functional dependency that is not a part of LHS, then it's called non-trivial.

$AB \to BC$, $A \to B$

Q.

X	Y	Z
1	4	2
1	5	3
1	6	3
3	2	2

Which FDs are satisfied?

i) $XY \to Z$, $Z \to Y$
ii) $YZ \to X$, $Y \to Z$ ✓
iii) $YZ \to X$, $X \to Z$
iv) $XZ \to Y$, $Y \to X$

→ If all determinants (LHS) are different, then that FD is valid on that instance. (eg. $XY \to Z$ here)

* Inference Rule (IR)

The Armstrong's axioms are the basis inference rule. Armstrong's axioms are used to conclude functional dependencies on a relational database. The inference rule is a type of assertion. It can be applied to a set of FD to derive other FD. Using inference rule we can derive additional functional dependency from the initial set.

1. Reflexive rule.

If Y is a subset of X, then X determine Y.

If $X \supseteq Y$, then $X \rightarrow Y$

2. Augmentation rule / Partial dependency

If $X \rightarrow Y$, then $XZ \rightarrow YZ$

3. Transitive rule

If $X \rightarrow Y$ & $Y \rightarrow Z$, then $X \rightarrow Z$.

4. Union rule

If $X \rightarrow Y$ & $X \rightarrow Z$ then $X \rightarrow YZ$.

Proof $X \rightarrow Y$, $X \rightarrow Z$

$\Rightarrow X \rightarrow XY$ (augmentation
$XX \rightarrow XY$
$X \rightarrow XY$)

$\Rightarrow XY \rightarrow YZ$

Using transitive rule,
$X \rightarrow YZ$.

5. Decomposition rule / Project rule

It is reverse of union rule.

If $X \rightarrow YZ$, then $X \rightarrow Y$ & $X \rightarrow Z$.

proof $X \rightarrow YZ$

$YZ \rightarrow Y$ (reflexive rule)

$X \rightarrow Y$ (Transitive rule)

6. Pseudo transitive rule

If $X \rightarrow Y$ & $YZ \rightarrow W$ then $XZ \rightarrow W$

proof $X \rightarrow Y$ $YZ \rightarrow W$

$\Rightarrow XZ \rightarrow YZ$ (Augmentation)

$XZ \rightarrow W$ (Transitive rule)

* Rules for Functional Dependency

i) **Rule 01** : An FD $X \rightarrow Y$ will always hold if all the values of X are unique irrespective of the values of Y.

attribs A, B, C, D, E

All values of A attrib. are different, then

A → any combination of attrib. A, B, C, D, E.

ii) **Rule 02** : An FD $X \rightarrow Y$ will always hold if all the values of Y are same irrespective of the values of X.

All the values of attrib. C are same.

Any combination of attribs A, B, C, D, E → C

· In general, a functional dependency $\alpha \rightarrow \beta$ always holds if either all values of α are unique or if all values of β are same or both.

iii) For an FD $X \rightarrow Y$ to hold, if 2 tuples in the table agree on the value of attrib. X, then they must also agree on the value of attrib. Y.

iv) For an FD $X \rightarrow Y$, ~~to hold~~ violation will occur only when for 2 or more same values of X, the corresponding Y values are different.

* Closure of an Attribute Set.

The set of all those attributes which can be functionally determined from an attribute set is called as a closure of that attribute set.

Closure of attribute set $\{X\}$ is denoted as $\{X\}^+$.

Steps to find the closure of an attribute set –

1. Add the attributes contained in the attribute set for which closure is being calculated to the result set.

2. Recursively add the attributes to the result set which can be functionally determined from the attributes already contained in the result set.

eg. Relation R (A, B, C, D, E, F, G). with FDs

$A \to BC$, $BC \to DE$, $D \to F$, $CF \to G$.

$A^+ = \{A\} = \{A, B, C\}$ using $A \to BC$

$= \{A, B, C, D, E\}$ using $BC \to DE$

$= \{A, B, C, D, E, F\}$ using $D \to F$

$= \{A, B, C, D, E, F, G\}$ using $CF \to G$.

$D^+ = \{D\} = \{D, F\}$

$\{B, C\}^+ = \{B, C\} = \{B, C, D, E\} = \{B, C, D, E, F\}$

$= \{B, C, D, E, F, G\}$.

- Finding the keys using closure.

→ Super key. If the closure result of an attribute set contains all the attributes of the relⁿ, then that attrib. set is called a super key of that relⁿ.

Thus, the closure of a super key is the entire relation schema.

→ Candidate key. If there exists no subset of an attribute set whose closure contains all the attributes of the relⁿ, then that attrib. set is a candidate key of that relation.

eg. $R(\check{A}, \check{B}, C, \check{D}, E, F)$

FDs $(C \to F,\ E \to A,\ EC \to D,\ A \to B)$

What can be key?

a) CD b) EC ✓ c) AE d) AC.

Q. $R(E, F, \check{G}, H, \check{I}, \check{J}, \check{K}, \check{L}, \check{M}, N)$

FDs $\{E, F\} \to \{G\}$ $\{K\} \to \{M\}$

$\{F\} \to \{I, J\}$ $\{L\} \to \{N\}$.

$\{E, H\} \to \{K, L\}$

What is key?

a) $\{E, F\}$ b) $\{E, F, H\}$ ✓ c) $\{E, F, H, K, L\}$ d) $\{E\}$.

→ $(E, F, H)^+ = (EFHGIJKLMN)$

It's a key { candidate key too. (only one.)

Q. $R(\check{A}, \check{B}, \check{C}, \check{D}, E, H)$

FDs $\{A \to B,\ BC \to D,\ E \to C;\ D \to A\}$

What are candidate keys?

a) AE, BE c) AEH, BEH, BCH

b) AE, BE, DE d) AEH, BEH, DEH.

→ $(EH)^+ = \{E, H, C\}$

$(AEH)^+ = \{AEHBCD\}$ ✓

$(BEH)^+ = \{\ ,\ \}$.

$(CEH)^+ = \{CEH,\ \}$

$(DEH)^+ = \{ABCDEH\}$ ✓

Start from the attributes that are not already determined by any determinant (so they need to produce themselves)

Q. R (A B C D E F G H) G' 2013 | Normalisation obs

*** FDs = { CH → G, A → BC, B → CFH, E → A, F → EG }

How many candidate keys does the rel^n have?

(4)

→ $D^+ = \{D\}$

- ✓ $(AD)^+ = \{A\,B\,C\,D\,E\,F\,G\,H\}$
- ✓ $(BD)^+ = \{A\,B\,D\,C\,F\,H\,E\,G\}$
- $(CD)^+ = \{C\,D\}$
- ✓ $(ED)^+ = \{D\,E\,A\,B\,C\,F\,H\,G\}$
- ✓ $(FD)^+ = \{D\,F\,G\,E\,A\,B\,C\,H\}$

$(GD)^+ = \{G\,D\}$

$(HD)^+ = \{D\,H\}$

(CD) (GD) (HD)

$(GCD)^+ = \{G\,C\,D\}$

$(HCD)^+ = \{H\,C\,D\,G\}$

$(GHD)^+ = \{G\,H\,D\}$

$(CGHD)^+ = (CGHD)$

Add C, H, G.
if add A, B, E, F then will become super keys.

eg. R (A B C D E)

AB → C, C → D, B → E

✓ $(AB)^+ = \{A\,B\,C\,D\,E\}$ candidate key | 2^3 Super keys

eg. R (A B C D E F)

A → B, C → D, E → F

$(ACE)^+ = \{A\,C\,E\,B\,D\,F\}$ candidate key. | 2^3 Super keys.

eg. R = (A B C D) FDs = { AB → CD, C → A, D → B }

$A^+ = \{A\}$ $B^+ = \{B\}$ $C^+ = \{C\,A\}$ $D^+ = \{D\,B\}$

✓ $AB^+ = \{A\,B\,C\,D\}$ ✓ $BC^+ = BCAD$

$AC^+ = \{A\,C\}$ $BD^+ = BD$

✓ $AD^+ = \{A\,D\,B\,C\}$ ✓ $CD^+ = CDBA$

eg. R (ABCDEF)

FDs = {AB → C, C → D, D → E, E → F, F → A}

$B^+ = \{B\}$

$AB^+ = \{ABCDEF\}$ $EB^+ = \{EBFACD\}$

$CB^+ = \{BCDEFA\}$ $FB^+ = \{FBACDE\}$

$(DB)^+ = \{DBEFAC\}$

eg. R (ABCDEF)

FDs = {AB → C, C → D, D → EB, E → F, F → A}

$A^+ = \{A\}$ $D^+ = \{DEBFAC\}$

$B^+ = \{B\}$ $E^+ = \{EFA\}$

$C^+ = \{CDEBFA\}$ $F^+ = \{FA\}$

$AB^+ = \{ABCDEF\}$ $BE^+ = \{BEFACD\}$

$BF^+ = \{BFACDE\}$

$EF^+ = \{EFA\}$

$AE^+ = \{AEF\}$

$AF^+ = \{AF\}$

$AEF^+ = \{AEF\}$

Candidate keys.

{C, D, AB, BE, AE}.

eg. R (ABCDEF)

A → BCDEF

BC → ADEF

DEF → ABC

$A^+ = \{ABCDEF\}$

$B^+ = \{B\}$ $E^+ = \{E\}$

$C^+ = \{C\}$ $F^+ = \{F\}$

$D^+ = \{D\}$

$BC^+ = \{BCADEF\}$.

$DEF^+ = \{DEFABC\}$

Can't form 4 element CK.

therefore no 5 or 6 element CK

(A, BC, DEF) CKs

Q. $R = ABCDE$

$FDs = \{A \to B, B \to C, C \to D, D \to A\}$

$*E^+ = \{E\}$ 4 Candidate keys.

$\checkmark AE^+ = \{AEBCD\}$

$\checkmark BE^+ = \{BECDA\}$

$\checkmark CE^+ = \{CEDAB\}$

$\checkmark DE^+ = \{DEABC\}$

Q. $R = (ABCDE)$

$A \to BC$

$CD \to E$

$B \to D$

$E \to A$

$\checkmark A^+ = ABCDE$

$B^+ = BD$

$C^+ = C$

$D^+ = D$

$\checkmark E^+ = EABCD$

$\checkmark BC^+ = BCDEA$

$\checkmark CD^+ = CDEAB$

$BD^+ = BD$

~~$BCD^+ = BCDEA$~~

CKs = A, E, BC, CD.

Q. $R(ABCD)$ $\{AB \to CD, D \to A\}$

What are CKs of sub relation $R_1(BCD)$?

→ $B^+ = B$ $C^+ = C$ $D^+ = DA$

$BC^+ = BC$ $CD^+ = CDA$ $\checkmark BD^+ = BDAC$

$CK = \{BD\}$

Q $R(ABCDEF)$

$AB \to C, B \to D, AD \to F, C \to D, D \to E, E \to F, E \to D$

CKs for $R_1(DEF)$?

→ $\checkmark D^+ = DEF$ Only CKs D, E.

$\checkmark E^+ = EFD$

$F = F$

Q R(ABCDE).

$A \to BC,\ CD \to E,\ B \to D,\ E \to A$

CKs of R_1 (ABCE) ?

→ ✓ $A^+ = ABCDE$

$B^+ = BD$

$C^+ = C$

✓ $E^+ = EABCD$

✓ $BC^+ = BCDEA$

CKs = {A, E, BC}

Q R(AB)

Total no. of FDs possible.

→ $\phi \to \phi$ $\phi \to A$ $\phi \to B$ $\phi \to AB$

$A \to B$ $A \to A$ $A \to \phi$ $A \to AB$

$B \to B$ $B \to A$ $B \to \phi$ $B \to AB$

$AB \to \phi$ $AB \to A$ $AB \to B$ $AB \to AB$.

Q R(ABC)

Given FDs $\{A \to B, B \to C\}$ what are additional FDs we can determine from this ?

→ $\phi \to \phi$ 1

$A \to \{ABC\}$ $2^3 = 8$

$B \to \{BC\}$ $2^2 = 4$

$C \to \{C\}$ 2.

$AB \to \{ABC\}$ $2^3 = 8$

$BC \to \{BC\}$ $2^2 = 4$

$CA \to \{ABC\}$ $2^3 = 8$.

$ABC \to \{ABC\}$ $2^3 = 8$

$X \to Y$
$ABC \to ABC$
$2^3 \quad 2^3$
$8 \times 8 = 64$

$(1 + 8 + 4 + 2 + 8 + 4 + 8 + 8) = 43$ additional FDs (that are valid)

Q. In a schema with attributes A, B, C, D & E following set of FDs are given

$A \to B$, $A \to C$, $CD \to E$, $B \to D$, $E \to A$

Which of the following FDs is not implied by the above set? ~~$CD \to AC$~~, ~~$BD \to CD$~~ (X), ~~$BC \to CD$~~, ~~$AC \to BC$~~

→ $CD^+ = CDEAB$

$AC^+ = ACBDE$

$BD^+ = BD$

$BC^+ = BCDEA$

* Equivalence of two sets of Functional Dependencies.

Two different sets of FDs for a given relation may or may not be equivalent. If F & G are the 2 sets of FDs, then following 3 cases are possible –

1. F covers G ($F \supseteq G$)
2. G covers F ($G \supseteq F$)
3. Both F & G cover each other ($F = G$).

1. Determining whether F covers G. :

a) Take FDs of set G into consideration. For each FD $X \to Y$, find the closure of X using the FDs of set G.

b) Take FDs of set G into consideration. For each FD $X \to Y$, find the closure of X using the FDs of set F.

c) If the FDs of set F has determined all those attribs that were determined by the FDs of set G, then it means F covers G.

Then we conclude $F \supseteq G$, otherwise not.

$G \subseteq F$

2. Determining whether G covers F.

Similar to 1.

3. Determining whether both F & G cover each other.

If $F \supseteq G$ & $G \supseteq F$, we conclude $F = G$.

eg: R (A, C, D, E, H).

Two functional dependency sets F & G.

$F = \{A \to C, AC \to D, E \to AD, E \to H\}$

$G = \{A \to CD, E \to AH\}$.

Which is true?

a) $G \supseteq F$ b) $F \supseteq G$ c) $F = G$ d) all.

→ $F \supseteq G$?

$A^+ = \{A C D\}$ $E^+ = \{E A D H C\}$ using F

So, $F \supseteq G$.

$G \supseteq F$?

$A^+ = \{A C D\}$

$AC^+ = \{A C D\}$

$E^+ = \{E A H C D\}$

$E^+ = \{E A H C D\}$

using G

So, $G \supseteq F$

$\therefore F = G$.

Q F : $\{A \to B, B \to C, C \to D\}$

G : $\{A \to BC, C \to D\}$

→ F ⊇ G ?

$A^+ \to (A\ B\ C\ D)$

$C^+ = (C\ D)$

G ⊇ F ?

$A^+ = \{A\ B\ C\ D\}$

$B^+ = (B)$ ×

$C^+ = (C\ D)$

Not equivalent. As F ⊇ G but G ⊉ F.

Q. F : $\{A \to B, AB \to C, D \to AC, D \to E\}$

G : $\{A \to BC, D \to AB\}$.

→ F ⊇ G?

$A^+ = \{A\ B\ C\}$

$D^+ = \{D\ A\ C\ B\ E\}$

F ⊇ G.

G ⊇ F

$A^+ = \{A\ B\ C\}$

$AB^+ = \{A\ B\ C\}$

$D^+ = \{D\ A\ B\ C\}$

$E^+ = \{E\}$ ×.

G ⊉ F.

Q. F : $\{A \to B, B \to C, C \to A\}$

G : $\{A \to BC, B \to A, C \to A\}$.

→ F ⊇ G ? ✓

$A^+ = \{A\ B\ C\}$

$B^+ = \{B\ C\ A\}$

$C^+ = \{C\ A\ B\}$

G ⊇ F ? ✓

$A^+ = \{A\ B\ C\}$

$B^+ = \{B\ A\ C\}$

$C^+ = \{C\ A\ B\}$

F = G.

* Canonical Cover

A canonical cover is a simplified & reduced version of the given set of functional dependencies. Since it is a reduced version, it's also called irreducible set.

Canonical cover is free from all the extraneous FDs (an attrib. of a FD is extraneous if we can remove it without changing the closure of the set of FDs). The closure of canonical cover is same as that of the given set of FDs. Canonical cover is not unique & may be more than one for a given set of FDs.

Need for CC.

i) Working with the set containing extraneous FDs increases the computation time.

ii) Therefore, the given set is reduced by eliminating the useless FDs.

iii) This reduces the computation time & working with the irreducible set becomes easier.

Steps to find CC

1. Writing set of FDs in such a way that each FD contains exactly one attrib. on its right side.

eg. $X \rightarrow YZ$ to

$X \rightarrow Y$

$X \rightarrow Z$.

2. Determining each FD to be essential or non-essential.

To come to know whether an FD is essential or not, compute the closure of its left side —

- Once by considering that the particular FD is present in the set.
- Once by considering that the particular FD is not present in the set.

Then following 2 cases are possible —

Case 1 If results come out to be same, it suggests that the presence or absence of that FD does not create any difference. Thus it is non-essential. Eliminate FD from set. Eliminate non-essential FDs as soon as it is discovered. *

Case 2 If results come out to be different, it means that the FDs are essential.

3. Check if there is any FD that contains more than one attribute on its left side.

Case 1 No There contains no FD with 2 or more attrib.s on left side. Previous step result is the cc.

Case 2 Yes Consider all FDs one by one. Check if left side can be reduced. (Checking — compute closure of all the possible subsets of the left side of that FD. If any of the subsets produce the same closure result as produced by the entire left side, then replace the left side with that subset).

Set obtained now is cc.

* Do not consider it while checking the essentiality of other FDs.

eg. $R(W, X, Y, Z)$

$FDs = \{ X \to W,\ WZ \to XY,\ Y \to WXZ \}$

→ Step 1

$X \to W$ ✓, $WZ \to X$ ✗, $WZ \to Y$ ✓, $Y \to W$ ✗, $Y \to X$ ✓, $Y \to Z$ ✓.

Step 2

$X^+ = \{X\}$ without considering $X \to W$

So, $X \to W$ is essential. [as we're not getting W]

$WZ^+ = \{WZYX\}$ without considering $WZ \to X$.

So, $WZ \to X$ is non essential. [as we're getting X not considering the FD $WZ \to X$]

$WZ^+ = \{WZ\}$ essential. $WZ \to Y$

$Y^+ = \{YWXZ\}$ non essential $Y \to W$

$Y^+ = \{YZ\}$ essential $Y \to X$

$Y^+ = \{YXW\}$ essential $Y \to Z$

We get after step 2,

$X \to W,\ WZ \to Y,\ Y \to X,\ Y \to Z$

Step 3

$WZ \to Y$. None of subsets of WZ produced Y in their closure.

$W^+ = \{W\}$ So, $WZ \to Y$ essential.

$Z^+ = \{Z\}$

So, CC. ~ $X \to W$,
$WZ \to Y$
$Y \to X$
$Y \to Z$

Q $\{AB \to C, D \to E, E \to C\}$ (F) is the minimal cover of $\{AB \to C, D \to E, AB \to E, E \to C\}$ (G) T/F

→ I way here best way

FD equivalence.

$F \supseteq G$. ? $AB^+ = ABC$ not equivalent

So, can't be minimal cover.

$AB \to C, D \to E, AB \to E, E \to C$

(AB → C)

$AB^+ = \{ABEC\}$ $AB \to C$ redundant

$D^+ = \{D\}$ $D \to E$ essential

$AB^+ = \{AB\}$ $AB \to E$ essential

$E^+ = \{E\}$ $E \to C$ essential.

$D \to E, AB \to E, E \to C$.. ↳ Minimal cover.

$A^+ = \{A\}$

$B^+ = \{B\}$.

* Decomposition of a Relation.

The process of breaking up or dividing a single relation into two or more subrelations.

- Properties

i) Lossless : No information is lost from the original relation during decomposition When the subrelations are joined back, the same relation is obtained that was decomposed. Every decomposition must always be lossless.

ii) Dependency preservation : None of the FDs that holds on the original relation are lost.

- Types of Decomposition.

(1) Relation R decomposed into subrelations $R_1, R_2, \ldots, R_n$. It is lossless join decomposition when the join of the subrelations results in the same relation R that was decomposed.

For lossless join decomposition,

$$R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n = R.$$

$\bowtie$ – natural join operator

#. Lossless join decomposition is also known as non-additive join decomposition. This is because the resultant relation after joining the subrelations is same as the decomposed relation. No extraneous tuples appear after joining of the sub-relations.

eg. R (A, B, C)

A	B	C
1	2	1
2	5	3
3	3	3

Decomposed into $R_1(A,B)$ & $R_2(B,C)$

A	B
1	2
2	5
3	3

B	C
2	1
5	3
3	3

natural join

A	B	C
1	2	1
2	5	3
3	3	3

Same as original relation R

(2) Relation R decomposed into $R_1, R_2, \ldots, R_n$.

This decomposition is called lossy join decomposition when the join of the sub relations does not result in the same relation R that was decomposed. The natural join of the sub relations is always found to have some extraneous tuples.

$$R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n \supset R$$

Lossy join decomposition is known as careless decomposition. This is because extraneous tuples get introduced on the natural join of the sub relations.

eg. R (A, B, C)

A	B	C
1	2	1
2	5	3
3	3	3

R_1 (A, C)

A	C
1	1
2	3
3	3

R_2 (B, C)

B	C
2	1
5	3
3	3

$\bowtie$

A	B	C	
1	2	1	
2	5	3	
2	3	3	← Extraneous
3	5	3	←
3	3	3	

$R_1 \bowtie R_2 \supset R$

• Determining whether decomposition is lossless or lossy.

If following conditions satisfy, the decomposition is lossless. If any fails, then lossy.

1. Union of both the subrelations must contain all the attributes that are present in the original relation R.

$$R_1 \cup R_2 = R.$$

2. Intersection of both the subrelations must not be null. There must be some common attribute which is present in both the subrelations.

$$R_1 \cap R_2 \neq \phi.$$

3. Intersection of both the subrelations must be a superkey of either R_1 or R_2 or both.

$$R_1 \cap R_2 = \text{Super Key of } R_1 \text{ or } R_2 \text{ or both}$$

Q. Consider a rel^n schema $R(A, B, C, D)$ with the FDs $A \to B$ & $C \to D$. Determine whether the decomposition of R into $R_1(A, B)$ & $R_2(C, D)$ is lossless or lossy.

→ $R_1(A, B) \cup R_2(C, D)$

$= R(A, B, C, D)$

$\rightsquigarrow R_1(A,B) \cap R_2(C,D)$

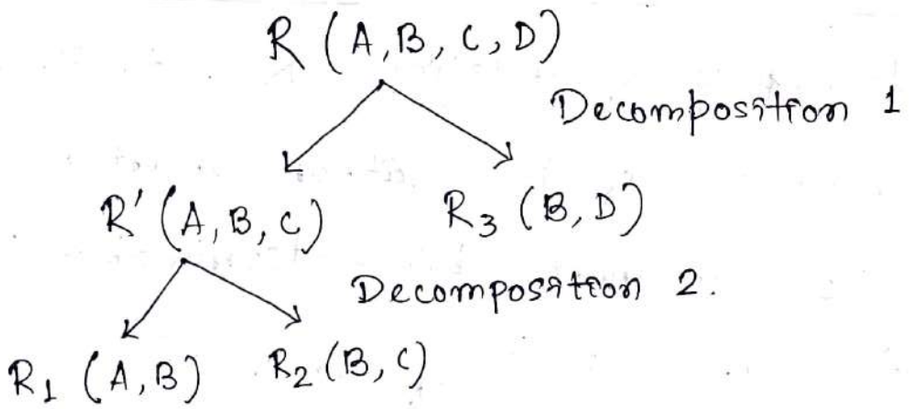
$= \phi$

Decomposition is lossy.

⑤ $R(A,B,C,D)$ with FDs $\{A \to B, B \to C, C \to D, D \to B\}$. Determine whether the decomposition of R into $R_1(A,B)$, $R_2(B,C)$, $R_3(B,D)$ is lossless or lossy.

→ # When a given relation is decomposed into more than two subrelations, then consider any possible way in which the relⁿ might have been decomposed into those subrelations. First, divide the given relⁿ into two subrelations. Then, divide the subrelations acc. to the subrelations given in problem.

As a thumb rule, any relⁿ can be decomposed only into two subrelations at a time.



Decomposition 1..

$\rightsquigarrow R'(A,B,C) \cup R_3(B,D)$

$= R(A,B,C,D)$

$\rightsquigarrow R'(A,B,C) \cap R_3(B,D)$

$= B$ Now, $B^+ = \{B, C, D\}$

$\rightsquigarrow R'(A,B,C) \cap R_3(B,D)$

$= B \neq \phi$

Attribute B can't determine attribute A of subrel^{n} R'. So, B is not super key of R'.

B can determine all attributes of R_3. Thus, it is a super key of sub relation R_3.

So, as one of the subrelations has the intersection as the super key, it is lossless.

Decomposition 2.

$\rightsquigarrow R_1(A,B) \cup R_2(B,C) = R'(A,B,C)$

$\rightsquigarrow R_1(A,B) \cap R_2(B,C) = B \neq \phi.$

$B^+ = \{B, C, D\}$

B is a super key of R_2.

So, decomposition 2 is lossless.

∴ Overall decomposition is lossless.

- Dependency preserving Decomposition.

If we decompose a relation R into R_1 & R_2, all dependencies of R either must be a part of R_1 or R_2 or must be determinable from combination of FD's of R_1 & R_2.

Q. $R(A,B,C,D)$ & FDs $\{A \rightarrow B, C \rightarrow D\}$.

Decomposition of R into $R_1(A,B)$ & $R_2(C,D)$ is —

$\rightsquigarrow$ $R_1(A,B) \cup R_2(C,D)$

$= R(A,B,C,D)$

$R_1(A,B) \cap R_2(C,D) = \phi$ lossy decomposition.

$R_1(A,B)$ $\quad A \to B,\ C \to D$

Possible FDs

- $A \to B$
- $B \to A$
- $C \to D$
- $D \to C$.

So, answer is dependency preserving but not lossless.

Q. $R(A,B,C,D)$

FDs $\{AB \to CD,\ D \to A\}$

Decomposition example 3. rbr.

(?) Decomposed into $R_1(A,D)$, $R_2(B,C,D)$.

$R_2(B,C,D)$ as A not present in $R_2(B,C,D)$.

$\rightarrow$ $R_1(A,D)$

$A \to D$

$D \to A$

$A^+ = \{A\}$, $D^+ = \{D,A\}$

$B^+ = \{B\}$, $C^+ = \{C\}$, $D^+ = \{D,A\}$

$BC^+ = \{B,C\}$, $CD^+ = \{C,D,A\}$, $BD^+ = \{B,D,A,C\}$

$R_2(B,C,D)$

$B \to C$, $C \to D$, $D \to B$

$BC \to CD$, $CD \to BD$, $BD \to BC$.

$BD \to C$

Not dependency preserving.

(as $AB \to CD$ not preserved)

Q. $R(A,B,C,D)$ & FDs $\{AB \to C,\ C \to D,\ D \to A\}$.

Decomposed into $R_1(A,B,C)$ & $R_2(C,D)$. Check dependency preservation.

$\rightarrow$ F_1 be the set of FDs for R_1 & that for R_2 is F_2.

For F_1, all combination of A, B, C.

$A^+ = \{A\}$ trivial $\quad B^+ = \{B\}$ trivial $\quad C^+ = \{C,D,A\} = \{C,A\}$ as D not present in R_1

$\therefore$ $C \to A$

$(AB)^+ = \{A, B, C, D\} = \{A, B, C\}$ remove D, as not in R_1

$AB \rightarrow C$

$(BC)^+ = \{B, C, D, A\} = \{A, B, C\}$

$BC \rightarrow A$

$(AC)^+ = \{A, C, D\} = \{A, C\}$ $AC \rightarrow AC$ trivial

$\therefore F_1 = \{C \rightarrow A, AB \rightarrow C, BC \rightarrow A\}$.

For F_2,

$C^+ = \{C, D, A\} = \{C, D\}$ $C \rightarrow D$

$D^+ = \{D, A\} = \{D\}$

$\therefore F_2 = \{C \rightarrow D\}$

Original FDs $\{AB \rightarrow C, C \rightarrow D, D \rightarrow A\}$

$D \rightarrow A$ not preserved.

∴ Decomposition is not dependency preserving.

→ Working formula - Checking FD preservation

R be decomposed to R_1, R_2 with FD set F.
R_1 & R_2 have FD set F_1, F_2.

If $R_1 = \{A, B, C\}$, $R_2 = \{C, D\}$ then to get F_1 & F_2 take every combination of attributes from R_1 & R_2 separately & compute closure of them using F.

i.e. A^+ AB^+ B^+ BC^+ C^+ CA^+ for F_1 | C^+ D^+ for F_2.

(not ABC, as it will be trivial) (not CD as it will be trivial)

While computing closures for different relations if the closure contains some attribute that is not present in the relation itself, then reject it. (i.e. if $AB^+ = \{A, B, C, D\}$ then we reject D as it is not an attribute of R_1). $\Rightarrow AB^+ = \{A, B, C\}$

We should not consider trivial FDs.

After doing this, we need to check whether $F_1 \cup F_2 = F$ or not.

If $F_1 \cup F_2 = F$ then it's FD preserving.

$F_1 \cup F_2 \subset F$ not preserving.

$F_1 \cup F_2 \supset F$ not possible.

Q $R(A, B, C, D, E, G)$.

FDs : $\{AB \to C,\ AC \to B,\ AD \to E,\ B \to D,\ BC \to A,\ E \to G\}$

Decomposed into $R_1(ABC)$, $R_2(ABDE)$, $R_3(EG)$.

→ $R(A, B, C, D, E, G)$

Decom 1

$R'(ABCDE)$ $R_3(EG)$

Decom 2

$R_1(ABC)$ $R_2(ABDE)$

Decom 1

~~$R_1(A, B, C) \cup R_2(A, B, D, E) \cup R_3(E, G)$~~

~~$= R'(A, B, C, D, E, G)$~~

~~R not needed.~~

$R_3(EG) \cup R'(ABCDE)$

$= R(A, B, C, D, E, G)$.

$R'(ABCDE) \cap R_3(EG)$

$= E \neq \phi$

Now, $E^+ = \{E, G\}$. E is a super key of R_3.

∴ Decom. 1 is lossless.

Decom. 2

$R_1(ABC) \cup R_2(ABDE) = R'(ABCDE)$

$R_1(ABC) \cap R_2(ABDE) = AB \neq \phi$

$AB^+ = \{A, B, C, D\}$

AB is a super key of R_1.

∴ Decom 2 is loss less.

Overall decomposition is lossless.

FD preservation.

F_1 A, B, C

$A^+ = \{A\}$

$B^+ = \{B, \not{D}\}$

$C^+ = \{C\}$

$AB^+ = \{A, B, C, \not{D}, \not{E}, \not{G}\}$ (AB → C)

$BC^+ = \{B, C, A, \not{D}, \not{E}, \not{G}\}$ (BC → A)

$CA^+ = \{C, A, B, \not{D}, \not{E}, \not{G}\}$ (CA → B)

F_2 A, B, D, E

$A^+ = \{A\}$

$B^+ = \{B, D\}$ (B → D)

$D^+ = \{D\}$

$E^+ = \{E, \not{G}\}$

$AB^+ = \{A, B, \not{C}, D, E, \not{G}\}$ (AB → DE)

$BD^+ = \{B, D\}$

$DE^+ = \{D, E, \not{G}\}$

$AE^+ = \{A, E, \not{G}\}$

$ABD^+ = \{A, B, D, \not{C}, E, \not{G}\}$ (ABD → E)

$BDE^+ = \{B, D, E, \not{G}\}$

$ADE^+ = \{A, D, E, \not{G}\}$

$AD^+ = \{A, D, E, \not{G}\}$ (AD → E)

$AE^+ = \{A, E, \not{G}\}$

$BE^+ = \{B, E, D, \not{G}\}$ (BE → D)

$ABD^+ = \{A, B, D, \not{C}, E, \not{G}\}$ (ABD → E)

$ABE^+ = \{A, B, E, \not{C}, D, \not{G}\}$ (ABE → D)

F_2 E, G

$E^+ = \{E, G\}$ $(E \rightarrow G)$

$G^+ = \{G\}$

$F_1 \cup F_2 \cup F_3 = \{AB \rightarrow C, BC \rightarrow A, CA \rightarrow B, B \rightarrow D, AD \rightarrow E,$
$BE \rightarrow D, ABD \rightarrow E, ABE \rightarrow D, AB \rightarrow DE, ABD \rightarrow E, E \rightarrow G\}.$

$F = \{AB \rightarrow C, AC \rightarrow B, AD \rightarrow E, B \rightarrow D, BC \rightarrow A, E \rightarrow G\}$

$F_1 \cup F_2 \cup F_3$ has all FDs from F.

So, it's FD preserving.

* Normalization or Normal form determination.

Process of making the database consistent by reducing the redundancies & ensuring the integrity of data through lossless decomposition.

Normalization is done to eliminate anomalies & redundancies.

Types

1) First Normal Form (1NF).

A given relation is called to be in 1NF if each cell of the table contains only an atomic value. The attribute of every tuple is either single valued or a null valued.

eg. Relation not in 1NF. –

Stu_id	Nm	Sbjct
100	A	CN, DAA
101	B	CD, DS
102	C	EM.

↓ brought into 1NF by rewriting each cell containing only one value

Stu-id	Nm	Sbjct
100	A	CN
100	A	DAA
101	B	CD
101	B	CD
102	C	EM.

in 1NF.

• By default, every rel^n is in 1NF. This is because formal definition of a rel^n states that value of all the attributes must be atomic.

ii) 2NF.

A relation is in 2NF iff rel^n already exists in 1NF & no partial dependency exists in the rel^n.

• Partial Dependency.

Few attributes of the candidate key determines non-prime attributes.

(An FD $X \rightarrow Y$ is a PD, if some attrib. $A \in X$ can be removed from X & the dependency still holds.
eg. $AB \rightarrow C$, $B \rightarrow C$)

$A \rightarrow B$ is partial dependency iff

1. A is a subset of some candidate key
2. B is a non-prime attribute.

To avoid partial dependency, incomplete candidate key must not determine any non-prime attribute.

eg. $R(V, W, X, Y, Z)$ with FDs $\{VW \rightarrow XY, Y \rightarrow V, WX \rightarrow YZ\}$. Possible CKs for this relⁿ are VW, WX, WY.

Prime attributes = $\{V, W, X, Y\}$

Non-prime n = $\{Z\}$

Now, as there exists no dependency where incomplete CK determines any non-prime attribute, there is no partial dependency.

Thus, relation is in 2NF.

iii) 3NF

A relⁿ is in 3NF iff relⁿ already exists in 2NF & no transitive dependency exists for non-prime attributes.

- **Transitive Dependency.**

$A \rightarrow B$ is TD iff

1. A is not a super key.
2. B is a non-prime attribute.

A rel^n is called to be in 3NF iff any one cond^n holds for each non-trivial FD $A \rightarrow B$ –

1. A is a super key
2. B is a prime attribute.

eg. $R(A, B, C, D, E)$ with FDs $A \rightarrow BC$, $CD \rightarrow E$, $B \rightarrow D$, $E \rightarrow A$. Possible CKs for this rel^n are A, E, CD, BC.

Prime attributes = $\{A, B, C, D, E\}$

No non-prime attributes.

All attributes on RHS of each FD are prime attributes. Thus, R is in 3NF.

iv) Boyce - Codd Normal Form (BCNF) / 3.5NF

Rel^n is in BCNF iff

1. Rel^n already exists in 3NF.
2. For each non trivial FD $A \rightarrow B$, A is a super key of the rel^n. (A can't be a non-prime attribute, if B is a prime attribute).

eg. $R(A, B, C)$ with FDs $\{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$.

Possible CKs $\{A, B, C\}$

→ Prime attributes = $\{A, B, C\}$

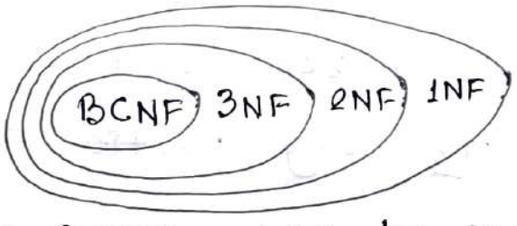
No non-prime attrib.

For FD $X \rightarrow Y$, for each prime attribute Y, X is also prime.

So, R is in BCNF.

• Important points.

→



A rel^{n} in BCNF will be in all other normal forms. A rel^{n} in 3NF will be in 2NF & 1NF. A rel^{n} in 2NF will be in 1NF.

→ BCNF is stricter than 3NF. 3NF is stricter than 2NF.

→ While determining the normal form of any given rel^{n},

i) Start checking from BCNF.

ii) This is because if it is found to be in BCNF, then it will be in all other normal forms.

iii) If the rel^{n} is not in BCNF, then check for other normal forms in the order.

→ In a relational database, a rel^{n} is always in 1NF.

→ Singleton keys are those that consist of only a single attribute. If all the candidate keys of a rel^{n} are singleton CKs, then it will always be in 2NF. This is because there will be no chances of existing any partial dependency. The CKs will either fully appear or fully disappear from the dependencies. Thus, an incomplete CK will never determine a non-prime attribute.

→ If all the attributes of a relation are prime attributes, then it will always be in 2NF at least. This is because there will be no chances of existing any partial dependency. Since there are no non-prime attributes, there will be no FD which determines a non-prime attribute.

→ If all the attributes of a rel^{n} are prime attributes, then it will always be in 3NF. Because, there will be no chances of existing any transitive dependency for non-prime attributes.

→ 3NF is considered adequate for normal relational database design.

→ Every binary rel^{n} is always in BCNF.

→ BCNF is free from redundancies arising out of FDs (zero redundancy).

→ A rel^{n} with only trivial FDs is always in BCNF.

→ BCNF decomposition is always lossless but not always dependency preserving.

→ Sometimes, going for BCNF may not preserve FDs. So, when FDs are not required, we can go for BCNF, otherwise 3NF only. 3NF is best suited for normal relational databases.

→ Lossy decomposition is not allowed in 2NF, 3NF, BCNF. So, if the decomposition of a relⁿ has been done in such a way that it is lossy, then the decomposition will never be in 2NF, 3NF & BCNF.

→ Unlike BCNF, lossless & dependency preserving decomposition into 3NF & 2NF are always possible.

→ A prime attribute can be transitively dependent on a key in a 3NF relⁿ.
A prime attribute can't be transitively dependent on a key in a BCNF relⁿ.

→ If a relⁿ consists of only singleton CKs & it's in 3NF, then it must also be in BCNF.

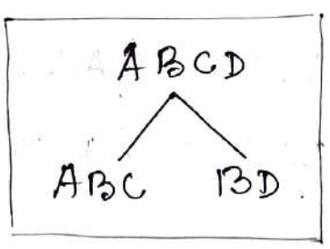
→ If a relⁿ consists of only one CK & it's in 3NF, then the relⁿ must also be in BCNF.

Q Given R(ABCD) & FDs {AB → C, B → D}.
Decompose it into 2NF.

→ $AB^+ = \{A, B, C, D\}$. | $B^+ = \{B, D\}$
AB is a candidate key.

AB → C | Prime {A, B}
B → D. | Non-prime {C, D}

Prime Non-prime.



{ 2NF is FD preserving.
2NF is lossless

Q $R(A, B, C, D, E)$ & FDs : $\{A \to B, B \to E, C \to D\}$

Decompose into 2NF.

→ Prime $\{A, B, C\}$

Non-prime $\{D, E\}$.

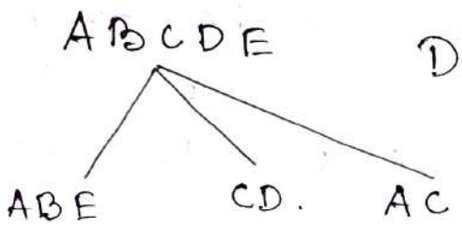
AC is a CK.

$A \to B$ partial dependency.

$A^+ = \{A, B, E\}$

$A \to B \quad C \to D \quad E$

$AC^+ = \{A, C, B, D, E\}$



Decomposition.

While merging first merge AC, ABE then ABCE & CD.

Q $R(A B C D E F G H I J)$ & FDs $\{AB \to C, BD \to EF, AD \to GH, A \to I, H \to J\}$. Decompose into 2NF.

→ RHS of FDs $\{C, E, F, G, H, I, J\}$

So, candidate key must contain ABD.

$ABD^+ = \{ABDCEFGHIJ\}$

ABD is the CK.

Non-prime = {C, E, H, F, G, I, J}

Prime = {A, B, D}

$AB \to C$ Partial

$BD \to EF$ "

$AD \to GH$ "

$A \to I$ PD

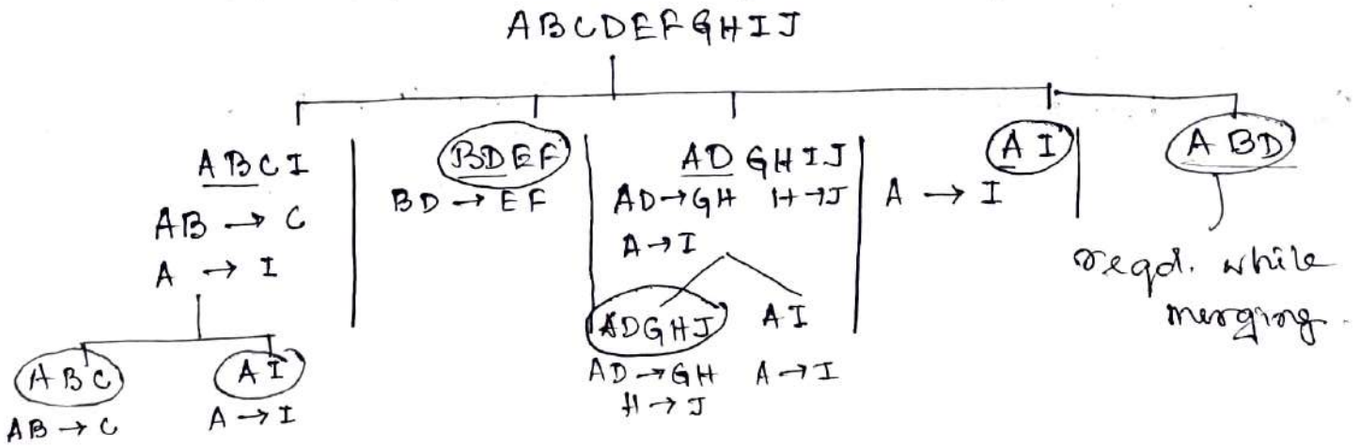
$H \to J$

$AB^+ = \{A, B, C\}$

$BD^+ = \{B, D, E, F\}$

$AD^+ = \{A, D, G, H, I, J\}$

$A^+ = \{A, I\}$



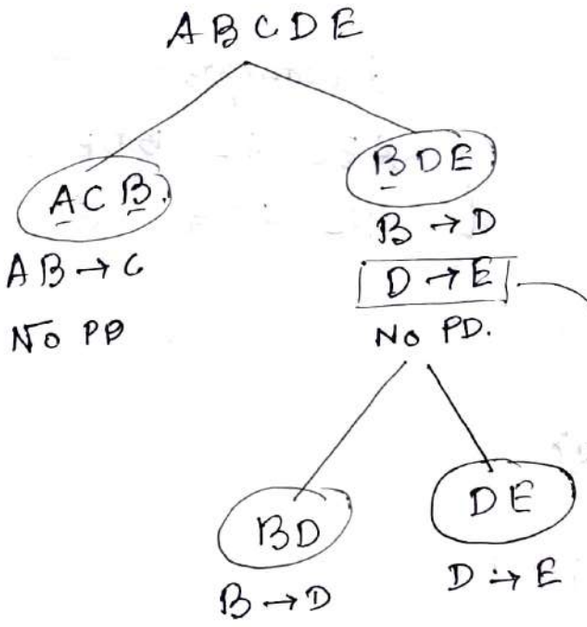
Q. R(ABCDE) FDs: $\{AB \to C, B \to D, D \to E\}$.

→ $AB^+ = \{A, B, C, D, E\}$.

AB is the CK.

$B \to D$ Partial dependency

$B^+ = \{B, D, E\}$.



in 2NF.

transitive dependency.

$D^+ = \{D, E\}$

3NF decomposition —

ABC, BD, DE.

Q. R(ABC) FDs: $\{AB \to C, C \to A\}$.

→ $B^+ = \{B\}$. $BA^+ = \{B, A, C\}$ $BC^+ = \{B, C, A\}$

BA is a CK.

BC is a CK.

No non-key attributes.

R(ABC) is in 3NF.

Q. R(ABCDEFGHIJ) FDs $\{AB \to C, BD \to EF,$

$AD \to GH, A \to I, H \to J\}$.

→ $ABD^+ = \{A, B, D, C, E, F, G, H, I, J\}$

ABD is a CK.

$AB \to C$ PD.

$BD \to EF$ PD.

$AD \to GH$ PD

$A \to I$ PD

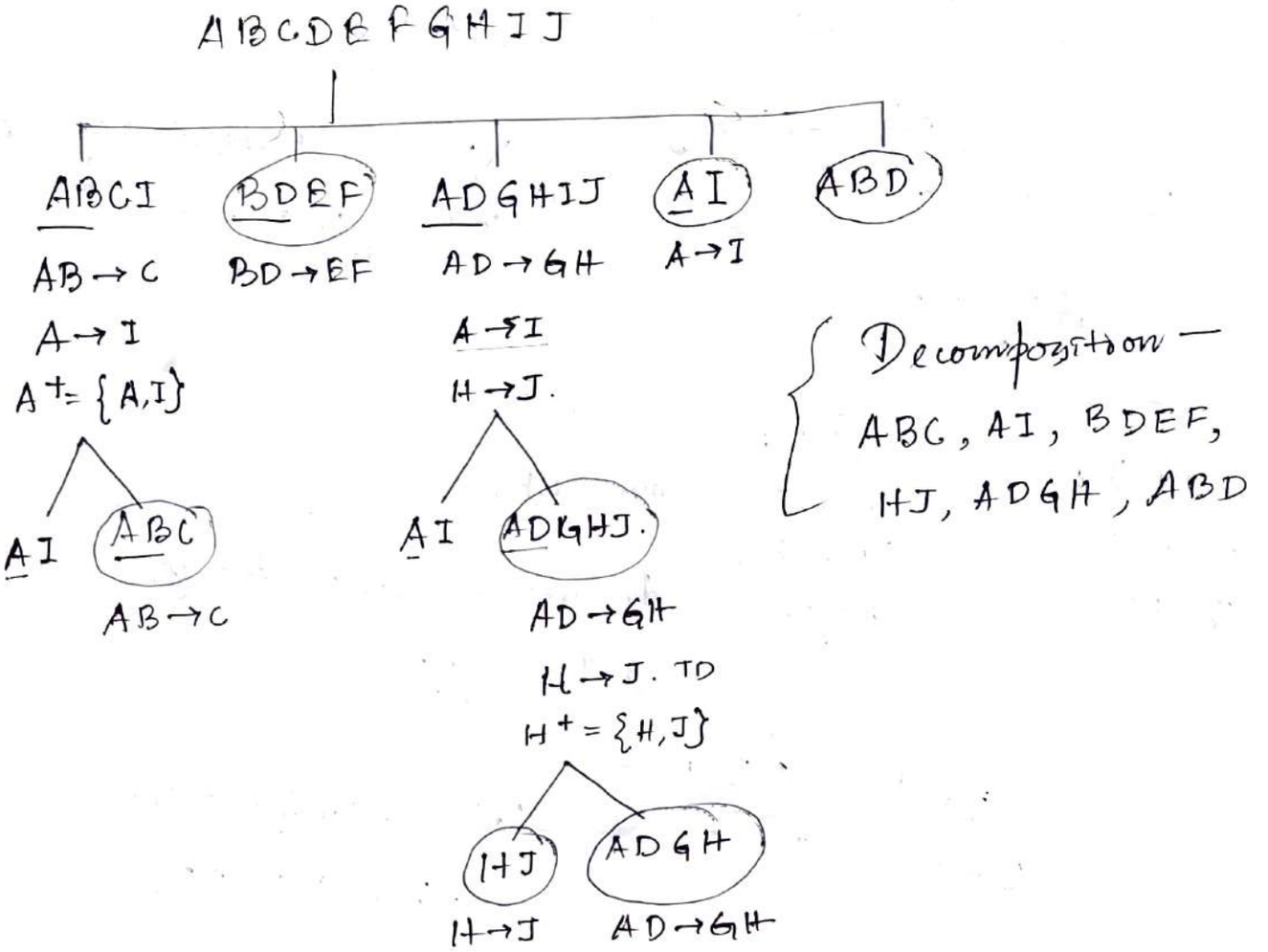
$H \to J$ TD.

$AB^+ = \{A, B, C, I\}$

$BD^+ = \{B, D, E, F\}$

$AD^+ = \{A, D, G, H, J, I\}$

$A^+ = \{A, I\}$



Q. R(ABCDE), FDs : {AB → C, B → D, D → E}

→ AB+ = {A, B, C, D, E}. AB is CK.

B → D.

B+ = {B, D, E}.

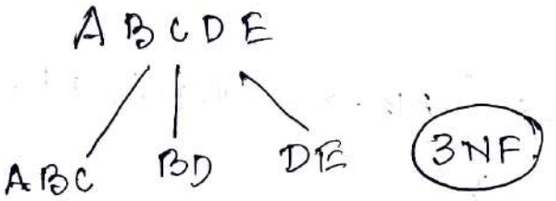
D → E

D+ = {D, E}

X → Y

X is a Super key

Y is a prime attrib.



Q. R(ABCDE) FDs {A → B, B → E, C → D} 3NF

→ AC+ = {A, B, C, D, E}.

A → B

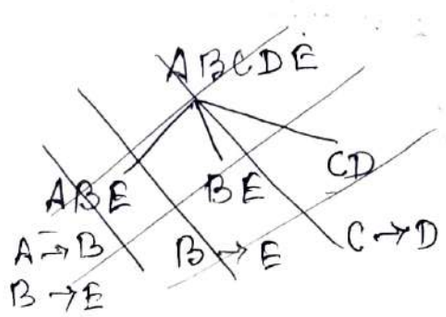
A+ = {A, B, E}

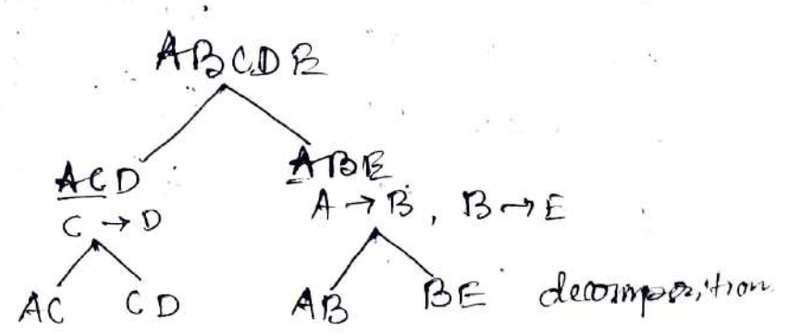
B → E

B+ = {B, E}

C → D

C+ = {C, D}





Q $R(A, B, C, D, E, F, G, H, I, J)$

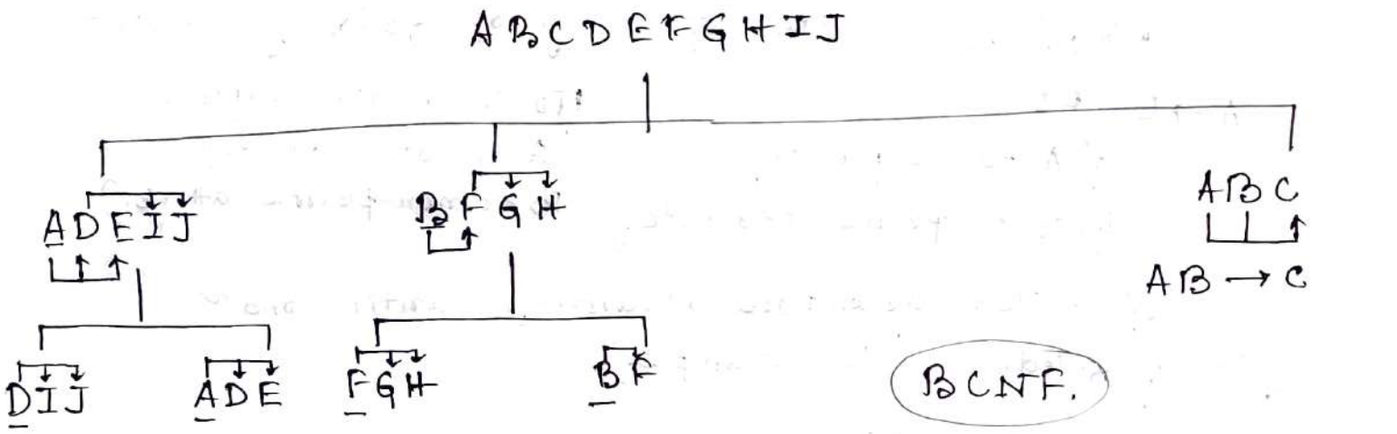
FDs : $\{ AB \to C,\ A \to DE,\ B \to F,\ F \to GH,\ D \to IJ \}$

Decompose into BCNF.

$\Rightarrow$ $AB^+ = \{A, B, C, D, E, F, G, H, I, J\}$

AB is CK.

$AB \to C$	$A \to DE$	$B \to F$	$F \to GH$	$D \to IJ$
	$A^+ = \{A, D, E, I, J\}$	$B^+ = \{B, F, G, H\}$	$F^+ = \{F, G, H\}$	$D^+ = \{D, I, J\}$



- Reducing to Normal forms - Working Technique.

1. Find out candidate key(s). [Check which are the attributes that are not dependent upon any other attributes; check which attrib.s are not on the RHS of any FD. Club them & take closure of that. If closure contains all attrib.s of rel^n then it is the CK, otherwise based on these essential keys try to know the CKs adding other attrib. to it.]

$R(ABCDE)$. $A \to BC,\ B \to D,\ D \to E$

$A^+ = ABCDE$ A is CK.

$R(ABCDE)$ $A \to B,\ B \to C,\ D \to E$

$AD^+ = ABCDE$. AD is CK.

2. Now, prime attrib.s are those who belong to at least one CK. Others are non prime.

A, D are prime attributes

B, C, E are non-prime.

And any superset of CK is super key.

3. Given any relation with a set of FDs, first check if it's in BCNF.

• Rel^n is in BCNF when every FD has a determinant which is super key.

If failed, to reduce to BCNF, we need to decompose on those portions where the FD is not holding the BCNF cond^n. Again recursively check for the cond^n to be true. (See examples).

4. Checking for 3NF ~

• Rel^n is in 3NF if any of following holds -

$A \rightarrow B$ FD

i) A is super key
ii) B is prime attribute.

No Transitive dependency
(A is not super key,
B is non-prime attrib.)

If failed decompose, recursively until cond^n gets fulfilled. (See examples).

5. Checking for 2NF ~

• Rel^n is in 2NF if there exists no partial dependency. Check partial dependency by observing that no proper subset of the CK is determining non-prime attributes.

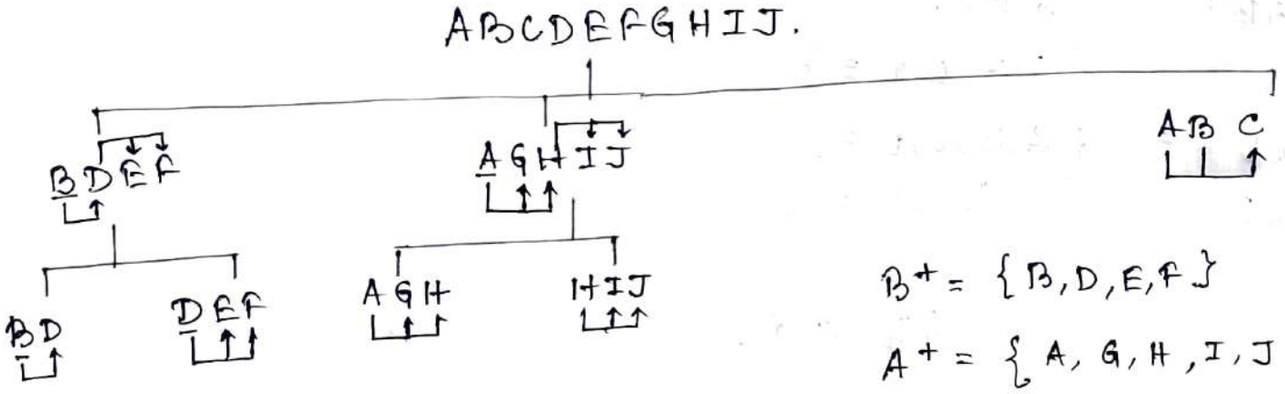
If failed decompose recursively until partial dependencies get away.

• While testing for a relation whether it is in BCNF, 3NF, 2NF or 1NF, Always start checking from BCNF. Then consecutively others. All checking must be done in the above ways.

Q R(ABCDEFGHIJ), FDs : { AB → C, B → D, D → EF, A → GH, H → IJ }

* Decompose into (BCNF)

→ . AB^+ = { A, B, C, D, E, F, G, H, I, J }



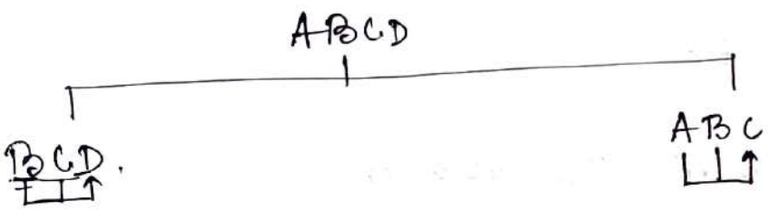
B^+ = { B, D, E, F }

A^+ = { A, G, H, I, J }

Q R(ABCD), FDs : { AB → C, BC → D }. BCNF.

→ AB^+ = { A, B, C, D } | AB CKey.

BC^+ = { B, C, D }



Q R(A, B, C, D) FDs : { AB → CD, D → A } BCNF.

→ B^+ = { B }.

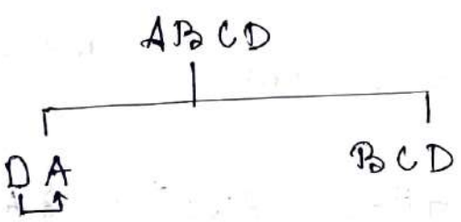
AB^+ = { A B C D }

BC^+ = { B C }

BD^+ = { C B D A }

AB as CK.

BD as CK.



BCNF

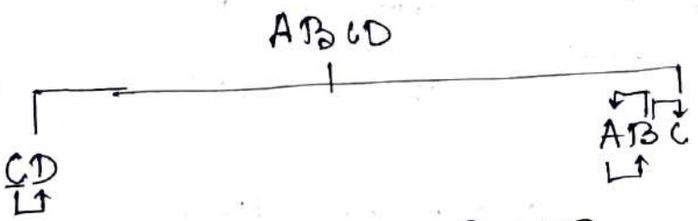
Q R(ABCD) FDs { A → B, B → A, B → C, C → D } BCNF.

→ A^+ = { A B C D }

B^+ = { A B C D }

C^+ = { C D }

D^+ = { D }

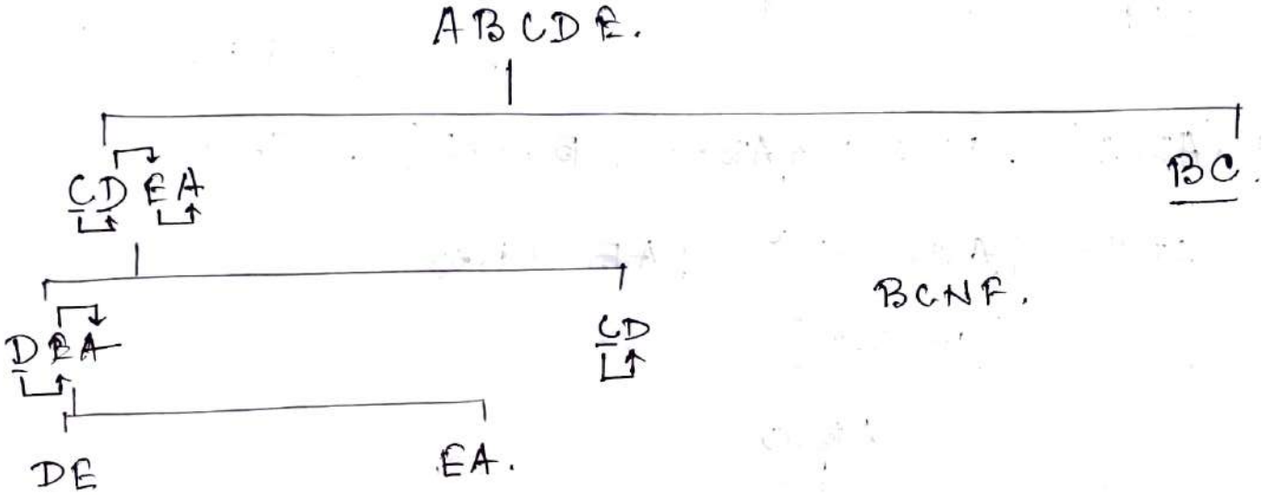


BCNF.

Q. $R(A, B, C, D, E)$; FDs : $\{AB \to C, C \to D, D \to E, E \to A\}$ BCNF.

→ $B^+ = \{B\}$. $C^+ = \{C, D, E, A\}$

$AB^+ = \{A, B, C, D, E\}$
$BC^+ = \{A, B, C, D, E\}$
$BD^+ = \{A, B, C, D, E\}$
$BE^+ = \{A, B, C, D, E\}$.



Q Check on which NF the relation is –

a) $R(ABCDEFGH)$ $AB^+ = \{ABCDEFGH\}$

$AB \to C$
$A \to DE$
$B \to F$
$F \to GH$

Checking BCNF – $A \to DE$ – A not a SK. So, not in BCNF

Checking 3NF – A is not SK, DE is not prime attrib. not in 3NF

Checking 2NF – $A \to DE$ proper subset of CK determining non-prime attrib. Not in 2NF.

∴ R is in 1NF.

b) $R(ABCDEFGHI)$ $ABD^+ = \{A, B, C, D, E, F, G, H, I\}$

$AB \to C$ Not in BCNF, as AB is not SK in $AB \to C$.

$BD \to EF$ Not in 3NF, as C is not prime attrib. in $AB \to C$.

$AD \to GH$ Not in 2NF, as $AB \to C$ is partial dependency.

$A \to I$

∴ R is in 1NF.

Check for the determinants to be CK.

c) R(ABCDE).

$AB \rightarrow CD$
$D \rightarrow A$
$BC \rightarrow DE$

$B^+ = \{B\}$.
$AB^+ = \{A, B, C, D, E\}$
$BC^+ = \{B, C, D, E, A\}$
$BD^+ = \{B, D, A, C, E\}$

CKs $\{AB, BC, BD\}$.

- Not in BCNF, as D is not a SK in $D \rightarrow A$
- In 3NF as all FDs fulfil the condⁿ that determinant is SK or dependent is a prime attribute.

d) R(ABCDEF) Finding CKs –

(*) $A \rightarrow BCDEF$
(*) $ABC \rightarrow ADEF$
$DEF \rightarrow ABC$.

✓ $A^+ = \{ABCDEF\}$ $D^+ = \{D\}$ $E^+ = \{E\}$ $F^+ = \{F\}$
$B^+ = \{B\}$ ✓ $BC^+ = \{BCADEF\}$
$C^+ = \{C\}$ ✓ $DEF^+ = \{DFFABC\}$

[We have to find all minimal super keys, by taking different combination of attrib.s].

CKs are $\{A, BC, DEF\}$.

R in BCNF as all determinants are super keys.

e) R(ABCDE)

$A \rightarrow B$
$BC \rightarrow E$
$DE \rightarrow A$

$CD^+ = \{CD\}$
$ACD^+ = \{ABCDE\}$
$BCD^+ = \{BCDEA\}$
$ECD^+ = \{ECDAB\}$

CKs $\{ACD, BCD, CDE\}$.

- Not in BCNF as in $A \rightarrow B$, A is not a SK.
- In 3NF as every dependent in the FDs is prime attribute.

f) R(VWXYZ)

$Z \rightarrow Y$
$Y \rightarrow Z$
$X \rightarrow YV$
$WV \rightarrow X$

$W^+ = \{W\}$
✓ $VW^+ = \{WVXYZ\}$
✓ $XW^+ = \{VWXYZ\}$
$YW^+ = \{YWZ\}$
$ZW^+ = \{ZWY\}$

$YWZ^+ = \{YWZ\}$

CKs $\{VW, XW\}$.

- Not in BCNF as Z not a SK in $Z \rightarrow Y$.
- Not in 3NF as Y not prime attrib in $Z \rightarrow Y$
- Not in 2NF as in $X \rightarrow YV$, proper subset of CK XW is determining non-prime attributes.
- R is in 1NF.

Q. R(ABCD) is a relation. Which does not have a lossless join, dependency preserving BCNF decomposition

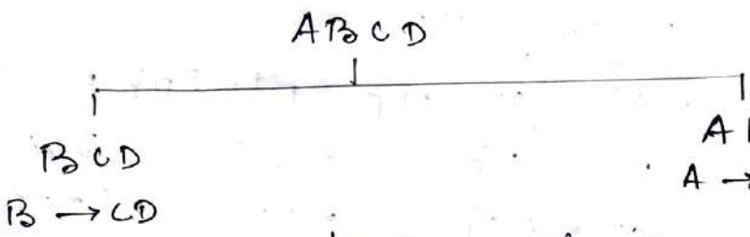
a) $A \to B$, $B \to CD$

b) $A \to B$, $B \to C$, $C \to D$

c) $AB \to C$, $C \to AD$

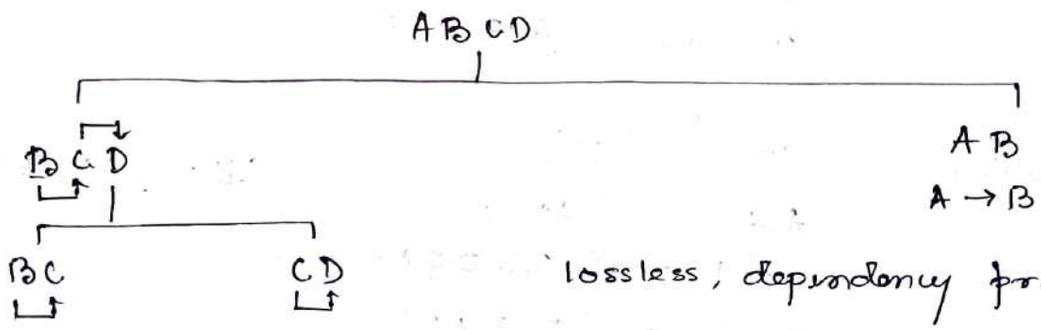
d) $A \to BCD$.

→ a) $A^+ = ABCD$. $B^+ = BCD$



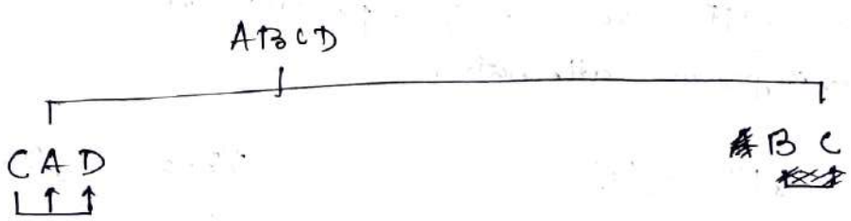
lossless & FD preserving.

b) $A^+ = ABCD$. $B^+ = BCD$.



lossless, dependency preserving

c) $B^+ = B$. $BA^+ = ABCD$ $BC^+ = BCAD$



$AB^+ = AB$. not preserving.

d) $A^+ = ABCD$.

in BCNF. No decomposition.

* Relational Algebra.

Procedural query language, that takes instances of relations as input & yields instances of relations as output. It uses operators to perform queries. An operator can be unary or binary.

Students

ID	Name	Age
1	Akon	17
2	Bkon	19
3	Ckon	13
4	Dkon	15.

Query. Select name of students with age less than 17.

Name
Ckon
Dkon.

→ Characteristics.

i) Relational operators always work on one or more relational tables.

ii) Relational operators always produce another relational table.

iii) Table produced by a relational operator has all the properties of a relational model.

Relational Algebra Operators.

- Basic
  - Unary
    - Projection (π)
    - Selection (σ)
    - Rename (ρ)
  - Binary
    - Union ($\cup$)
    - Cross product ($\times$)
    - Minus / set difference ($-$)
- Extended.
  - Join ($\bowtie$)
  - Division ($\div$)
  - Intersection ($\cap$)

→ SELECT operation.

Select a subset of tuples from R that satisfy a selection condition.

- $\sigma_{\text{<selection\_condition>}}(R_1)$
- Resulting relation R_2.

  degree (R_1) = degree (R_2)

  $|R_2| \leq |R_1|$ for any selection condition

- Commutative

  $\sigma_{C_1}(\sigma_{C_2}(R)) = \sigma_{C_2}(\sigma_{C_1}(R))$

- $\sigma_{C_1}(\sigma_{C_2}(\ldots(\sigma_{C_n}(R))\ldots)) = \sigma_{C_1 \text{ and } C_2 \text{ and } \ldots \text{ and } C_n}(R)$

→ PROJECT operation.

Project some columns (attributes) from the table (relation) & discard the other columns.

- $R_2 \leftarrow \pi_{<\text{attribute list}>}(R_1)$
- $R_2 \leftarrow \pi_{<LNAME, SALARY>}(EMPLOYEE)$.
- R_2 has only the attributes in <attribute list> with same order as they appear on the list.
- degree $(R_2) = |<\text{attribute list}>|$.
- PROJECT will remove any duplicate tuples. This happens when attribute list contains only non-key attributes.
- $|R_2| \leq |R_1|$.
  If the <attribute list> is a super key of R_1 then $|R_2| = |R_1|$.
- $\pi_{<list1>}(\pi_{<list2>}(R)) = \pi_{<list1>}(R)$ where <list2> must be a subset of <list1>.
- π is not commutative.

→ RENAME Operation.

- $\rho_{S(B_1, B_2, \ldots, B_n)}(R)$ - | order should be maintained for attributes.
- $\rho_S(R)$ | $\rho_{(B_1, B_2, \ldots, B_n)}(R)$
- S is renamed rel^n name ; B_i's attribute names

→ UNION, INTERSECTION, MINUS.

- $R_1 \cup R_2$, $R_1 \cap R_2$, $R_1 - R_2$.
  - valid iff R_1 & R_2 are union compatible.
- Two relations $R_1(A_1, A_2, \ldots, A_n)$ and $R_2(B_1, B_2, \ldots, B_n)$ are union compatible if degree (R_1) = degree (R_2) and dom (A_i) = dom (B_i) for $1 \leq i \leq n$.
- Resulting relation has the same attribute names as the first relation R_1 (convention).
- Commutative → Union, Intersection
  Associative → Union, Intersection.

→ Cartesian Product / Cross Product.

- $R_1 \times R_2$ (R_1, R_2 need not be union compatible).
- $R_1(A_1, A_2, \ldots, A_n) \times R_2(B_1, B_2, \ldots, B_m)$
  $= Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$
  where
  $|Q| = |R_1| \times |R_2|$
- eg.

A_1	A_2
a	b
c	d

$\times$

B_1
e
f
g

$=$

A_1	A_2	B_1
a	b	e
a	b	f
a	b	g
c	d	e
c	d	f
c	d	g

→ JOIN Operation.

- Used to combine related tuples from two relations onto single tuples.

- $R_1(A_1, \ldots, A_n) \bowtie_{\langle \text{join condition} \rangle} R_2(B_1, \ldots, B_m) =$

$$Q(A_1, \ldots, A_n, B_1, \ldots, B_m);$$

where each tuple in Q is the combination of tuples - one from R_1 and one from R_2 - whenever the combination satisfies the join condition.

- $R_1 \bowtie_{\langle \text{join condition} \rangle} R_2 \equiv \sigma_{\langle \text{condition} \rangle} (R_1 \times R_2)$

→ Equijoin and Natural Join. Variations.

- All JOIN operations with only '=' operator used in the conditions are called Equijoin.

- Each tuple in the resulting relation of an Equijoin has the same values for each pair of attributes listed in the JOIN cond^n.

- Natural Join (*) was created to get rid of the superfluous attributes in an Equijoin.

Natural Join requires each pair of join attributes have the same name in both relations, otherwise, a renaming operation should be applied first.

- $0 \leq |R_1(A_1, A_2, \ldots, A_n) \underset{\langle \text{cond} \rangle}{\bowtie} R_2(B_1, B_2, \ldots, B_m)| \leq n \times m$

→ UNION, INTERSECTION, MINUS.

- $R_1 \cup R_2$, $R_1 \cap R_2$, $R_1 - R_2$.
  - valid iff R_1 & R_2 are union compatible.
- Two relations $R_1(A_1, A_2, \ldots, A_n)$ and $R_2(B_1, B_2, \ldots, B_n)$ are union compatible if degree (R_1) = degree (R_2) and dom (A_i) = dom (B_i) for $1 \leq i \leq n$.
- Resulting relation has the same attribute names as the first relation R_1 (convention).
- Commutative → Union, Intersection
  Associative → Union, Intersection.

→ Cartesian Product / Cross Product.

- $R_1 \times R_2$ (R_1, R_2 need not be union compatible).
- $R_1(A_1, A_2, \ldots, A_n) \times R_2(B_1, B_2, \ldots, B_m)$
  $= Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$
  where
  $|Q| = |R_1| \times |R_2|$
- eg.

$$\begin{pmatrix} A_1 & A_2 \\ a & b \\ c & d \end{pmatrix} \times \begin{pmatrix} B_1 \\ e \\ f \\ g \end{pmatrix} = \begin{pmatrix} A_1 & A_2 & B_1 \\ a & b & e \\ a & b & f \\ a & b & g \\ c & d & e \\ c & d & f \\ c & d & g \end{pmatrix}$$

→ JOIN Operation.

- Used to combine related tuples from two relations into single tuples.

- $R_1(A_1, \ldots, A_n) \bowtie_{\langle \text{join condition} \rangle} R_2(B_1, \ldots, B_m) = Q(A_1, \ldots, A_n, B_1, \ldots, B_m)$;

where each tuple in Q is the combination of tuples - one from R_1 and one from R_2 - whenever the combination satisfies the join condition.

- $R_1 \bowtie_{\langle \text{join condition} \rangle} R_2 \equiv \sigma_{\langle \text{condition} \rangle} (R_1 \times R_2)$

→ Equijoin and Natural Join. Variations.

- All JOIN operations with only '=' operator used in the conditions are called Equijoin.

- Each tuple in the resulting relation of an Equijoin has the same values for each pair of attributes listed in the JOIN cond^n.

- Natural Join (*) was created to get rid of the superfluous attributes in an Equijoin.

Natural Join requires each pair of join attributes have the same name in both relations, otherwise, a renaming operation should be applied first.

- $0 \leq |R_1(A_1, A_2, \ldots, A_n) \bowtie_{\langle \text{cond} \rangle} R_2(B_1, B_2, \ldots, B_m)| \leq n \times m$

→ Complete Set of Relational Algebra Operations.

$(\sigma, \pi, \cup, -, \times)$

Any of the other relational algebra operations can be expressed as a sequence of these operations.

- $R \cap S \equiv (R \cup S) - ((R - S) \cup (S - R))$.
  $\equiv R - (R - S)$.
- $R \bowtie_{<cond>} S \equiv \sigma_{<cond>} (R \times S)$.
- A NATURAL JOIN can be specified as a Cross Product preceded by RENAME & followed by Select & Project operations.

→ Division Operation.

- $T(Y) \leftarrow R(Z) \div S(X)$ where
  X, Y, Z are sets of attributes & $X \subseteq Z$ and $Y = Z - X$.
- A tuple $t \in T$ if tuples $t_1 \in R$ with $t_1[Y] = t$ and with $t_1[X] = t_2$ for every tuple t_2 in S.
- $\pi_X(A) - \pi_X((\pi_X(A) \times B) - A)$
- Division operation in terms of Basic operations.

$T_1 \leftarrow \pi_{R-S}(R)$

$T_2 \leftarrow \pi_{R-S}((S \times T_1) - R)$

$T \leftarrow T_1 - T_2$.

→ Aggregate functions & Grouping.

- SUM, AVERAGE, MAXIMUM, MINIMUM and COUNT.

- $R_2 \leftarrow {}_{\langle \text{grouping attribs} \rangle}\mathcal{G}_{\langle \text{func attribs} \rangle}(R_1)$.

The resulting relation R_2 has the grouping attributes + one attribute for each element in the function list.

Each group results in a tuple in R_2.

→ OUTER JOIN:

- Left Outer Join:

$R_3(A_1, \ldots, A_n, B_1, \ldots, B_m) \leftarrow R_1(A_1, \ldots, A_n) \overset{\langle \text{Join cond} \rangle}{\bowtie} R_2(B_1, \ldots, B_m)$.

This operation keeps every tuple t in left relation R_1 in R_3 and fills NULL for attribs $B_1, \ldots, B_m$ if the join condition is not satisfied for t.

- Right Outer Join: ⟖

Keeps every tuple t in right relation R_2 in the resulting relation R_3.

- Full Outer Join ⟗

Keeps tuples from R_1 & R_2.

→ Outer Union

Make union of 2 relations that are partially compatible.

- $R_3(A_1, \ldots, A_n, B_1, \ldots, B_m, C_1, \ldots, C_p) \leftarrow$

$R_1(A_1, \ldots, A_n, B_1, \ldots B_m)$ OUTER UNION $(A_1, \ldots, A_n, C_1, \ldots, C_p)$

- List of compatible attribs are represented only once.
- $In \cdot R_3$, fill NULL when necessary.

* Tuple Relational Calculus.

Nonprocedural language : specifies what to do instead of how to do.

Expressive power of Relational Algebra & Relational Calculus is identical.

A relational query language L is considered relationally complete if we can express in L any query that can be expressed in Relational Calculus.

→ Tuple Variables and Range Relations.

A tuple variable usually range over a particular database relation : the variable may take as its value any individual tuple from that relation.

- $\{t \mid COND(t)\}$.

→ Three information should be specified in a tuple calculus expression —

a) For each tuple variable t, the range relation R of t is specified as R(t).

b) A condition to select particular combinations of tuples.

c) A set of attributes to be retrieved, the requested attributes.

→ Expressions & Formulas in TRC.

General expression form:

$$\{t_1.A_1, t_2.A_2, \ldots, t_n.A_n \mid COND(t_1, t_2, \ldots, t_n, t_{n+1}, t_{n+2}, \ldots, t_{n+m})\}$$

Where $t_1, t_2, \ldots, t_n, t_{n+1}, \ldots, t_{n+m}$ are tuple variables, each A_i is an attribute of the relation on which t_i ranges and COND is a condition or formula.

- A formula is made up of one or more atoms, which can be one of the following -

i) An atom of the form $R(t_i)$ defines the range of the tuple variable t_i as the relation R.

If the tuple variable t_i is assigned a tuple that is a member of R, then the atom is TRUE.

ii) An atom of the form $t_i.A$ op $t_j.B$, where op is one of the comparison operators

If the tuple variables t_i & t_j are assigned to tuples such that the values of the attributes $t_i.A$ and $t_j.B$ satisfy the condition, then the atom is True.

iii) An atom of the form $t_i.A$ op c or c op $t_j.B$.

If the tuple variables t_i (or t_j) is assigned to a tuple such that the value of the attribute $t_i.A$ (or $t_j.B$) satisfies the condition, then the atom is TRUE.

- A formula is made up one or more atoms connected via the logical operators and, or, not & is defined recursively as follows ~

a) Every atom is a formula.

b) If F_1 & F_2 are formulas then so are $F_1 \wedge F_2$, $F_1 \vee F_2$, $\neg F_1$, $F_1 \Rightarrow F_2$.

→ Existential & Universal Quantifiers.

- free & bound for tuple variables in formula —

1) An occurrence of a tuple variable in a formula F that is an atom is free in F.

2) An occurrence of a tuple variable is free or bound in $(F_1 \wedge F_2)$, $(F_1 \vee F_2)$, $\neg F_1$, depending on whether it is free or bound in F_1 or F_2. A tuple variable may be free in F_1 and bound in F_2.

3) All free occurrences of a tuple variable t in f are bound in formulae $f' = (\exists t)(f)$ or $f' = (\forall t)(f)$.

- A formulae with quantifiers is defined as follows ~

a) If f is a formula, then so is $(\exists t)(f)$ where t is a tuple variable.

b) If f is a formula, then so is $(\forall t)(f)$, where t is a tuple variable.

→ Transforming $\exists$, $\forall$

- $(\forall x)(P(x)) \equiv \neg(\exists x)(\neg(P(x)))$
- $(\exists x)(P(x)) \equiv \neg(\forall x)(\neg(P(x)))$
- $(\forall x)(P(x) \wedge Q(x)) \equiv \neg(\exists x)(\neg(P(x)) \vee \neg(Q(x)))$
- $(\forall x)(P(x) \vee Q(x)) \equiv \neg(\exists x)(\neg(P(x)) \wedge \neg(Q(x)))$
- $(\exists x)(P(x) \vee Q(x)) \equiv \neg(\forall x)(\neg(P(x)) \wedge \neg(Q(x)))$
- $(\exists x)(P(x) \wedge Q(x)) \equiv \neg(\forall x)(\neg(P(x)) \vee \neg(Q(x)))$
- $(\forall x)(P(x)) \Rightarrow (\exists x)(P(x))$
- $\neg(\exists x)(P(x)) \Rightarrow \neg(\forall x)(P(x))$.

→ Safe Expressions

Result is a finite number of tuples.

- $\{t \mid \neg(\text{Employee}(t))\}$ is unsafe.

- An expression is safe if all values in its result are from the domain of the expression.

* Domain Relational Calculus.

Rather than having variables range over tuples in relations, the domain variable range over single values from domains of attributes.

- General form

$$\{x_1, x_2, \ldots, x_n \mid \text{COND}(x_1, x_2, \ldots, x_n, x_{n+1}, \ldots, x_{n+m})\}$$

① Domain variables : $x_1, \ldots, x_n$ that range over the domains of attributes.

- Formula -

i) An atom of the form $R(x_1, \ldots, x_j)$ where R is the name of a relation of degree j and x_i is a domain variable.

The atom defines that $\langle x_1, \ldots, x_j \rangle$ must be a tuple in R, where the value of x_i is the value of the ith attribute of the tuple.

If the domain variables are assigned values corresponding to a tuple of R, then the atom is TRUE.

ii) An atom of the form x_i op x_j, where op is one of the comparison operators.

iii) An atom of the form x_i op c or c op x_j, where c is a constant variable.

SQL

→ SEQUEL : Structured English Query Language.

- Based on Relational algebra and Tuple Relational Calculus.

→ Creation of Schema.

```
CREATE SCHEMA FINANCE AUTHORIZATION RAVI;
```

schema_name ↙ owner_name ↙

```
GRANT SELECT ON FINANCE.* to user1;
```

→ Create table.

```
CREATE TABLE <tablename> (<col1> datatype (width),
        <col2> datatype (width), ... );
```

→ Data types

Numeric
- NUMBER
- FLOAT
- INT
- REAL
- DECIMAL
- BINARY FLOAT
- etc

Character
- CHAR
- VARCHAR2
- NCHAR
- NVARCHAR2
- LONG
- RAW
- LONG RAW

Date
- DATE
- TIME
- TIMESTAMP
- INTERVAL.

Boolean
- BOOLEAN.

→ Constraints in SQL.

1. NOT NULL
2. UNIQUE
3. Primary Key
4. Foreign Key
5. CHECK
6. DEFAULT
7. REF Constraints.

eg.

Dept : dept-id (Primary key)
dept-name (Unique)
location.

```
CREATE TABLE Dept (dept-id NUMBER PRIMARY KEY,
                   dept-name varchar2(30),
                   location varchar2(100),
                   UNIQUE (dept-name));
```

- Constraints can be defined at two levels – a) Column level
  b) Table level.

- Not Null constraint should be defined in column level.
- Composite key should be defined at table level.

eg Emp.: emp-id (P.K.), emp-name (NOT NULL),
job, dept_id (REF dept-id of Dept),
mgr, Salary (≥ 50K),
commission (if not given default 100),
email, phone } UNIQUE.

```
CREATE TABLE emp ( emp_id NUMBER PRIMARY KEY,
        emp_name varchar2(30) NOT NULL,
        dept_id NUMBER REFERENCES Dept (Dept_id),
        job varchar2(30), mgr varchar2(30),
        Salary NUMBER CHECK (Salary ≥ 50K),
        Commission NUMBER DEFAULT 100,
        email varchar2(30),
        phone varchar2(30),
        UNIQUE (email, phone)) ;
```

Column level.

Table level

- ALTER to respecify constraints.

→ Insertion in SQL.

Syntax I.

```
INSERT INTO <table-name> values (attr_value 1,
                                 attr_value 2, ...) ;
```

Attrib. values should be in the order of attr. list.

Number of attrib. values should be equal to the number of attribute in the attribute list.

Syntax II.

```
INSERT INTO <table_name> (attr1, attr2, ...)
        values (attr1_value, attr2_value, ...)
```

→ Delete

```
DELETE FROM <table_name> [ WHERE <cond> ] ;
```

DROP : Deletes both data & table

DROP TABLE <table-name> ;

→ UPDATE.

UPDATE <table-name> SET <attr> = <value>

[WHERE <condition>] ;

→ Referential Triggered Actions.

Allows us to further describe the relationship between the referential column and the object it references by attaching a referential triggered action to the foreign key.

Three triggered actions ~

ON DELETE ON UPDATE

SET NULL
SET DEFAULT
SET CASCADE.

eg. CREATE TABLE Emp (emp_id NUMBER,
dept_id NUMBER references Dept(dept_id));

i) ON DELETE SET NULL ON UPDATE SET NULL
dept_id NUMBER DEFAULT 3 References Dept(dept_id)

ii) ON DELETE SET DEFAULT ON UPDATE SET DEFAULT

iii) ON DELETE SET CASCADE ON UPDATE SET CASCADE

• Default value shouldn't be deleted.

→ ALTER.

Used to add, delete or modify columns in an existing table.

Used to add and drop various constraints on an existing table.

eg.

```
CREATE TABLE EMP (emp_id NUMBER,
                  emp_nm char(10) UNIQUE,
                  Salary NUMBER CHECK (Salary ≥ 5000),
                  phone varchar2(13));

ALTER TABLE EMP & ADD PRIMARY KEY (emp_id);
ALTER TABLE EMP DROP UNIQUE (emp_nm);
ALTER TABLE EMP MODIFY CHECK (Salary ≥ 10000);
ALTER TABLE EMP ADD Addr varchar2(30);
ALTER TABLE EMP MODIFY phone varchar2(20);
ALTER TABLE EMP DROP Column phone;
```

→ SELECT Used to retrieve data from database

TC SELECT <attr-list>

X FROM <table-list>

σ WHERE <conditions>;

→ VIEW Virtual table based on the result set of a SQL statement.

```
CREATE VIEW <viewname> AS
        SELECT <cd names>
            FROM <table_name>
                WHERE <condition>;
```

→ Aliasing

SQL aliases are used to temporarily rename a table (or) a column in a table.

<table/attrib name> AS <modified name>

→ Pattern Matching

LIKE : Used to search for a specific pattern in a column.

We have 2 wild cards here -

i) % represents any sequence of 0 or more characters.

ii) _ , used to replace a single character.

eg. SELECT * from student where name like 'R%' ;

(Students whose name starts with R)

SELECT name from student where rank like '____' ;

(Student names for whom 4 digit rank)

Select name from student where marks like '90\%' escape '\' and

email like 'abc_%@%.%' escape '\' ;

(having 90% marks and abc_...@....email)

• NOT LIKE

→ SET Operations.

UNION, INTERSECT, EXCEPT ($\cup$, $\cap$, $-$)

UNION ALL, INTERSECT ALL, EXCEPT ALL

→ Arithmetic Operators.

$+ \quad - \quad * \quad /$

SQL allows the use of arithmetic operators in queries on numeric domain.

→ CONCATENATE.

For string data type, the concatenate operator '||' can be used in a query to append two string values.

eg. SELECT FNAME||LNAME AS "FULL_NAME"
FROM EMPLOYEE;

→ BETWEEN. (text, numbers, dates)

[Select column_name(s) from Table_name WHERE column_name BETWEEN val1 and val2;]

eg. Select * from Employee where emp_nm between 'A' and 'D'

→ ORDER BY.

Used to order/sort the result of the SQL query in ascending or descending order.

SELECT <column_list> FROM
<Table_list> ORDER BY
<column_1> ASC| DESC, <column_2> ASC|DESC, ...

→ Dealing with NULL Values.

- Arithmetic expressions (Produces NULL)
- Logical expressions (Unknown)
- Two NULLs are always different.
- Select A from R where B IS NULL.

→ IN Operator.

Specify multiple values in a where clause

Select column_name(s) from Table_name
where column_name in (val1, val2, ...);

eg. Select fname from employee
where dno in (2, 3, 4).
(employees who work in dept 2, 3, 4)

⊛ Example 1, 2 on In operator. (V).

→ ANY/SOME, ALL. (>, <, >=, <=, <>, =)

• Find names of all the employees whose Salary is greater than all employees in dept no 5.

selects Fname from Employee where
salary > ALL (select salary from Employee
where Dno = 5);

• <>any ≅ <>some ≅ not in ( )

→ EXISTS / NOT EXISTS

The EXISTS condition is used in combination with a subquery and is considered to be met if the subquery returns at least 1 row.

eg. Retrieve the fnames of the employees who have dependents with same fname, sex as that of the employee.

```
select e.fname from Employee e where
    EXISTS ( select * from Dependent d where
        e.ssn = d.essn AND e.sex = d.sex AND
            e.fname = e.dependent_name );
```

eg. Retrieve the fnames of the employees who have no dependents.

```
select fname from Employee where NOT EXISTS
        (select * from Dependent where
                essn = ssn)
```

eg. List the fname of managers who have at least one dependent.

```
select e.fname from Employee e where EXISTS (select *
    from dependent d where d.essn = e.ssn)
        AND EXISTS ( select * from Department Dept
            where e.ssn = Dept.mgr_ssn) ;
```

eg. Fname of each employee who works on all the projects controlled by department no 5.

```
Select fname from Employee e where NOT EXISTS ((select
    Pnumber from project where Dnum = 5) EXCEPT
    (select Pno from works_on w where e.ssn = w.essn));
```

→ Aggregate Functions.

Take a collection (a set or multiset) of values as input and return a single value.

Average : AVG Total : SUM
Minimum : MIN Count : COUNT.
Maximum : MAX.

eg. Select AVG (Salary) from Employee;

Select AVG Distinct (Salary) from Employee;

~ AVG (marks_A + marks_B)

→ Dealing with NULL values in Aggregate functions.

All aggregate functions except count(*) ignore NULL values in their input collection.

The count of an empty collection is defined to be zero (0) and all other aggregate operations return a value of NULL when applied on an empty set.

Q. For each department that has more than 5 employees retrieve the department name & the number of its employees who are making more than 40000.

→ select Dname, count (*)
from Department, Employee where
Dnumber = Dno and salary > 40000 and
Dno in (select Dno from Employee Group by
Dno having count (*) > 5)
Group by Dname;

→ GROUP BY

eg. find the average salary in each department.

```
select Dno, AVG (salary) from Employee
        GROUP BY Dno.;
```

- All attributes used in the group by must appear in the select clause. Any attribute that is not present in the Group By clause must appeare only inside the aggregate function in the select clause.

→ HAVING Clause

eg List the department numbers with average salary greatere than 20000.

```
select Dno from Employee group by Dno
        HAVING AVG (salary) > 20000 ;
```

- An SQL query can contain a HAVING only if it has a Group By clause.

→ WITH Clause.

Allows to give a sub-query block a name, a process also called sub-query refactoring, which can be referenced in several places within the main SQL query.

The name assigned to the sub-query is treated as though it was a view or table. It is basically a drop-in replacement to the normal sub-query.

eg. Select emp_id from Employee;

Using with:

```
with Emp_tab AS (select emp_id from Employee)
    select * from Emp_tab.;
```

Multiple with clauses:

```
with Dept_temp as (select dept_id from Departments
                   where dname = 'CSE'),
     Emp_name as (select emp_name from employee E,
                  Dept_temp D where E.dept_id = D.dept_id)
     select * from Emp_name ;
```

Q. For each employee, display the number of employees working in their respective department.

→
```
with dept_count as (select dept_no , count(*) as
                    dept_count from emp GROUP BY deptno)
Select e.ename as emp_name, DC. dept_count
    as emp_dept_count From emp e , dept_count
    DC where ie. deptno = DC.deptno ;
```

→ JOINS

Normal
Natural
Left Outer
Right Outer
Full outer
Natural left outer
" right "
" full "

```
select * from (R JOIN S
               on A = C)
                / /
              attribs
```

Q G'98 Database.

FREQUENTS (Student, Parlour) giving the parlours each student visits.

SERVES (Parlour, Ice_cream)

LIKES (Student, Ice_cream)

The students that frequent at least one parlour that serves some ice cream that they have.

→ SELECT DISTINCT A.Student FROM
FREQUENTS A, SERVES B, LIKES C
WHERE A.Parlour = B.Parlour AND
B.Ice-cream = C.Ice_cream AND
A.student = C.Student.

Q. G'00 Given $r(w,x)$ and $s(y,z)$, the result of

select distinct w,x from r,s

is guaranteed to be same as r, provided

✓a) r has no duplicates and s is not empty

b) r and s have no duplicates

c) s has no duplicates and r is non-empty

d) r and s have same no of tuples.

Q. G'03 Consider following SQL query:-

select distinct $a_1, a_2, \ldots, a_n$

from $r_1, r_2, \ldots, r_m$ where p, for an arbitary p this query is equivalent to which relational algebra expressions?

a) $^{TC}_{a_1,\ldots a_n}\,\pi_p\,(r_1 \times r_2 \times \ldots \times r_m)$

b) $^{TC}_{a_1,\ldots a_n}\,\pi_p\,(r_1 \bowtie \ldots \bowtie r_m)$

c) $^{TC}_{a_1,\ldots,a_n}\,\pi_p\,(r_1 \cup \ldots, r_m)$ | d) $^{TC}_{a_1\ldots a_n}\,\pi_p\,(r_1 \cap \ldots \cap r_m)$

Q. G'11

Loan_records

Borrower	Bank_mgr	Amount
A	D	X
B	E	Y
C	D	Z

```
select count(*) from (
(Select Borrower, Bank_mgr from Loan_records) AS S
NATURAL JOIN
(select Bank_mgr, Amount from Loan_records) AS T);
```

Output ?

a) 3 b) 9 ✓ c) 5 d) 6

→

S

Borrower	Bank_mgr
A	D
B	E
C	D

T

Bank_mgr	Amount
D	X
E	Y
D	Z

A	D	X	(5)
A	D	Z	
B	E	Y	
C	D	X	
C	D	Z	

Q. G'14.

Employees (emp_id, fn, ln, hire_dt, dept_id, sal)
Departments (dept_id, dept_nm, mgr_id, loc_id)

You want to display the last names & hire dates of all latest hires in their respective departments in the location id 170. You issue the following query -

```
Select ln, hire_dt from Employees where
(dept_id, hire_dt) in (select dept_id, MAX (hire_dt)
from Employees JOIN Departments using (dept_id)
Where loc_id = 170 group by dept_id);
```

What is the outcome?

→ Executes and gives the correct result.

Q. G'14. SQL allows duplicate tuples in relations ✱ correspondingly defines the multiplicity of tuples in result of joins. Which one of the following queries always gives the same answer as the nested query -

```
Select * from R where a in (select S.a from S)
```

a) select R.* from R, S where R.a = S.a

b) select distinct R.* from R, S where R.a = S.a

c) select R.* from R, (select distinct a from S) as S_1 where R.a = S_1.a

d) select R.* from R, S where R.a = S.a and is unique R.

Q G'05. Book (title, price). Assume no 2 books have same price.

✱✱

```
Select title from Book as B where (select count (*)
from Book as T where T.price > B.Price) < 5 ;
```

a) titles of the four most expensive books.

b) titles of the fifth most expensive book

c) title of the fifth most expensive book

d) title of the five most expensive books

(Take a demo table and check).

Q. G'06. enrolled (student, course) in which (student, course) is the pk ; paid (student, amount) where student is the pk. Assume no null values & no foreign keys or integrity constraints.

Q1: select student from enrolled where student in (select student from paid)

Q2: select student from paid where student in (select student from enrolled)

Q3: select E.student from enrolled E, paid p where E.student = P.student

Q4: select student from paid where exists (select * from enrolled where enrolled.student = paid.student)

✓ a) All queries return identical row sets for any db.

✗ b) Q2 & Q4 return identical row sets for all dbs but there exist dbs for which Q1 & Q2 return different row sets.

✗ c) There exist dbs for which q3 returns strictly fewer rows than q2.

✗ d) There exist dbs for which q4 will encounter an integrity violation at runtime.

Q G'11 Consider a database table T containing 2 columns X & Y each of type integer. After the creation of table, one record (X = 1, Y = 1) is inserted.

Let MX and MY denote the respective max values at any time stamp. Using MX and MY, new records are inserted in the table 128 times with X & Y values being MX + 1, 2 * MY + 1

What will be the o/p of the query after the steps above are carried out ?

Select Y from T where X = 7 ;

a) 127 b) 255 c) 129 d) 257

→

X	Y
1	1
2	3
3	7
4	15
5	31
6	63
7	127

X value

$2^{} - 1$

$2^{7} - 1 = 127$

$$\begin{cases} T_{n+1} = 2T_n + 1 \\ T_1 = 1 \end{cases}$$

$\Rightarrow T_x = 2^x - 1$

Q G'2007 employee (empID, name, department, salary)

***

- Note In case of non-correlated nested query the subquery is replaced with the result given by the subquery and then the entire query is performed.

For correlated nested query, we have to perform subquery multiple times for every tuple from the outer query.

Q G'2009 employee, customer, rating

*

Q G'2016 Cinema (theater, address, capacity)

Q G'2016 Water-schemes.

Transactions

* A transaction is a collection of operations that forms a single logical unit of work.

* ACID Properties.

i) Atomicity : All or none.

— Transaction manager (T^n recovery subsystem)

ii) Consistency : Correctness

— User / application programmer.

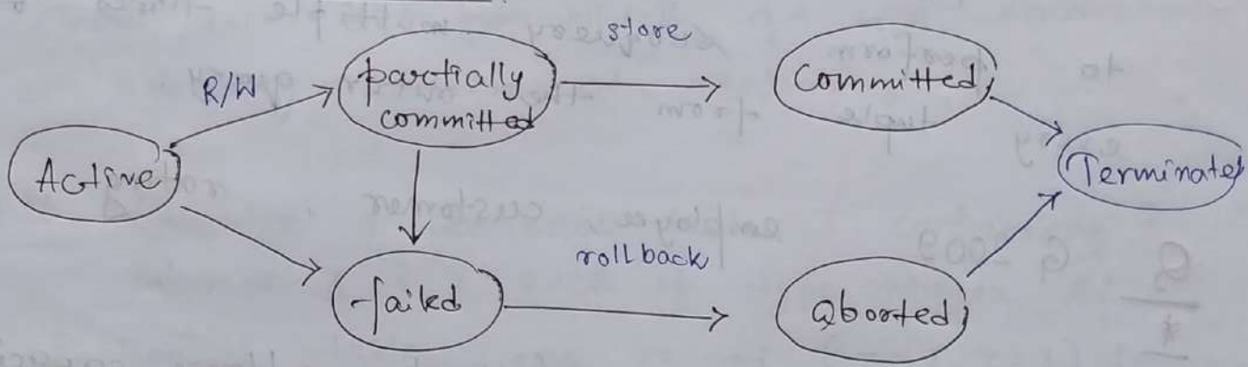
iii) Isolation : Each transaction Tx must be executed without knowing what is happening with other transactions.

— Concurrency control manager

iv) Durability : All updates done by a transaction must become permanent.

— Recovery manager.

* Transaction States :



* Problems with concurrent execution :

→ Lost update problem. (W-W conflict)

Same as race condition from Operating System.

All updation done for some transaction is lost. (2 writes ~~one~~ on same data item.)

→ Dirty Read Problem. (W-R conflict).

Reading some data that is not committed. Reading data while some transaction is rolling back (due to failure).

(Reading dirty values that can be changed later.)

→ Unrepeatable read problem (R-W conflict)

After reading some data, another transaction is changing the data.

→ Phantom tuple problem (Phantom phenomenon)

Transaction having effect for some execution of some other transaction. Same query on some data producing different results.

T_1 T_2

E.no	E.name	Sal
1	A	5000
3	C	4000

select * from Emp
where sal ≥ 3000 ---→ insert into Emp values (4, D, 3500)

E.no	E.name	Sal
1	A	5000
3	C	4000
4	D	3500

select * from Emp
where sal ≥ 3000

→ Incorrect Summary Problem.

Changing data while reading summary of some database.

* Schedule: Represents the order in which the operations of transactions are executed.

→ If there are 2 transactions T_1 and T_2 having n and m operations, possible no. of scheduling of operations is $\frac{(n+m)!}{n!\,m!}$.

(as relative ordering of n operations and m operations must be same.) relative ordering of operations doesn't change

→ If there are n transactions having no. of operations $a_1, a_2, \ldots, a_n$, possible no. of schedules =

$$\frac{(a_1 + a_2 + \ldots + a_n)!}{a_1!\, a_2! \cdots a_n!}$$

No. of serial schedules = $n!$

No. of concurrent / nonserial schedules =

$$\frac{(a_1 + a_2 + \ldots + a_n)!}{a_1!\, a_2! \cdots a_n!} - n!$$

* Types of schedules.

1. Serial Schedule: If we don't have any interleaving then that type of schedule is serial.

When transactions are serial, it ensures consistency.

If there are n transactions, possible no. of serial schedules is $n!$.

2. Complete Schedule. :

A schedule is complete if the last operation of each transaction either abort or commit.

3. Recoverable Schedule. :

For each pair of transactions T_i and T_j such that if T_j reads data item previously written by T_i, then the commit operation of T_i appears before the commit operation of T_j.

T_1 T_2
R(x)
W(x) R(x)
C_1 C_2

Only reads are allowed before write operation on same data.

Has to follow order $T_i \rightarrow T_j \Rightarrow Commit_i \rightarrow Commit_j$

4. Cascadeless Schedule. :

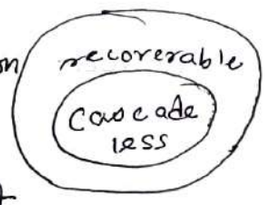
Each pair of transactions T_i & T_j such that T_j reads the data item that is written by $T_i \rightarrow$ then the commit operation of T_i should appear before the read operation of T_j.

no dirty read

If a transaction is cascadeless, it must be recoverable. Vice versa is not true.

T_1 T_2
R(x)
W(x) R(x)
C_1 C_2

Phenomenon in which a single transaction failure leads to a series of transaction rollbacks is called cascading rollback.



Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work.

5. Strict Schedule. :

If a value written by a transaction cannot be read or overwritten by other transactions until the transaction is either committed or aborted. No blind writes.

(Blind write - writing without reading data)

Every strict schedule is cascadeless (not vice versa).

* Serializability.

Helps to identify which non-serial schedulers are correct & will maintain the consistency of the database.

If a given non-serial schedule of n transactions is equivalent to some serial schedule of n transactions, then it is called as a serializable schedule.

Serializable schedules behave exactly same as serial schedules. Thus serializable schedules are always consistent, recoverable, cascadeless and strict.

→

Serial Schedules	Serializable Schedules.
i) No concurrency allowed.	i) Allowed.
ii) Leads to less resource utilization & CPU throughput.	ii) Improves resource utilization and CPU throughput.
iii) Less efficient.	iii) More efficient.

→ Result Equivalent Schedule.

S_1 and S_2 are result equivalent if they produce the same final database state.

→ Types of Serializability.

1. Conflict Serializability.

Conflict operations. –

T_i	T_j	
R(A)	W(A)	Conflict operations.
W(A)	R(A)	
W(A)	W(A)	
R(A)	R(A)	non-conflicting operations of different data

If a given non-serial schedule can be converted into a serial schedule by swapping its non-conflicting operations, then its called as a conflict serializable schedule.

2 operations are conflicting when a) both belong to different transactions, b) both Ts are on same data, c) at least one of the 2 operations is a write.

eg.

T_1	T_2
$R_1(A)$	
$W_1(A)$ ·····→	$R_2(A)$
$R_1(B)$	

$W_1(A)$ & $R_2(A)$ are conflicting.

Conflict equivalence.

eg.

S_1

T_1	T_2
$R_1(A)$	
$W_1(A)$	
	$R_2(A)$
	$W_2(A)$
$R_1(B)$	
$W_1(B)$	

$R_1(A) \longrightarrow W_2(A)$
$W_1(A) \rightarrow R_2(A)$
$W_1(A) \rightarrow W_2(A)$

Serial schedule

S_2

T_1	T_2
$R_1(A)$	
$W_1(A)$	
$R_1(B)$	
$W_1(B)$	
	$R_2(A)$
	$W_2(A)$.

$R_1(A) \longrightarrow W_2(A)$
$W_1(A) \rightarrow W_2(A)$
$W_1(A) \rightarrow R_2(A)$

Not to be used for solving problems.

S_1 & S_2 are conflict equivalent.

• Checking conflict serializability

i) Find & list all the conflicting operations.

ii) Start creating a precedence graph by drawing one node for each transaction.

iii) Draw the edge for each conflict pair such that if $X_i(v)$ & $Y_j(v)$ forms a conflict pair then draw an edge from T_i to T_j. This ensures T_i gets executed before T_j.

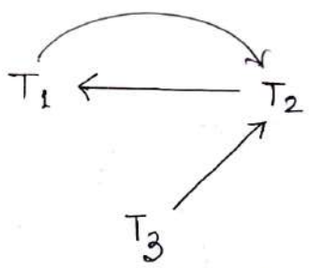
iv) Check if there is any cycle formed in the graph. If there is no cycle found, then the schedule is conflict serializable. (Perform topological sort of the DAG to find serial schedules.)

Q. Check conflict serializability :

S : $R_1(A)$, $R_2(A)$, $R_1(B)$, $R_2(B)$, $R_3(B)$, $W_1(A)$, $W_2(B)$

→

$R_2(A)$, $W_1(A)$	$T_2 \rightarrow T_1$
$R_1(B)$, $W_2(B)$	$T_1 \rightarrow T_2$
$R_3(B)$, $W_2(B)$	$T_3 \rightarrow T_2$



Cycle (T_1, T_2) exists.

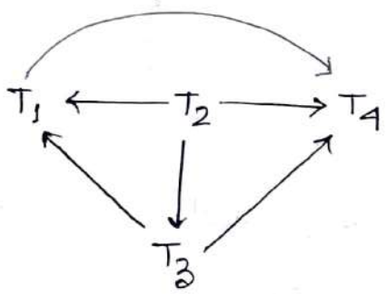
Not conflict serializable.

Q. Check conflict serializability & recoverability.

T_1	T_2	T_3	T_4
R(X)			
		W(X)	
		Commit	
W(X)			
Commit			
	W(Y)		
	R(Z)		
	Commit		
			R(X)
			R(Y)
			Commit.

→ Conflicting operations.

$R_2(x), W_3(x)$	$T_2 \to T_3$
$R_2(x), W_1(x)$	$T_2 \to T_1$
$W_3(x), W_1(x)$	$T_3 \to T_1$
$W_3(x), R_4(x)$	$T_3 \to T_4$
$W_1(x), R_4(x)$	$T_1 \to T_4$
$W_2(Y), R_4(Y)$	$T_2 \to T_4$.



No cycle.

Conflict serializable!

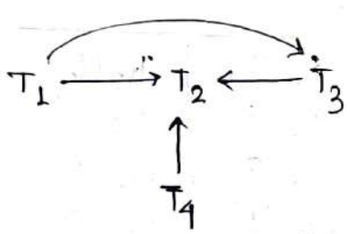
Conflict serialisable schedules are always recoverable.

Q Check whether the schedule is conflict serializable or not. If yes, then determine all possible serialized schedules.

T_1	T_2	T_3	T_4
			R(A)
	R(A)		
		R(A)	
W(B)			
	W(A)		
		R(B)	
	W(B)		

$R_4(A), W_2(A)$	$T_4 \to T_2$
$R_3(A), W_2(A)$	$T_3 \to T_2$
$W_1(B), R_3(B)$	$T_1 \to T_3$
$W_1(B), W_2(B)$	$T_1 \to T_2$
$R_3(B), W_2(B)$	$T_3 \to T_2$

No cycle ⇒ Conflict serializable.



All possible topological orderings of the precedence graph will be the possible serialized schedules.

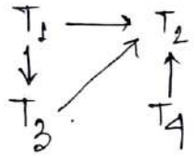
(Topological sorting - of a directed acyclic graph is a linear ordering of the vertices in such a way that if there is an edge in the DAG going from vertex u to vertex v, then u comes before v in the ordering.)

Serialized schedules - $T_1 \to T_3 \to T_4 \to T_2$

$T_1 \to T_4 \to T_3 \to T_2$

$T_4 \to T_1 \to T_3 \to T_2$.

Eg.



Serialized schedules -

$T_1\ T_3\ T_4\ T_2$

$T_1\ T_4\ T_3\ T_2$

$T_4\ T_1\ T_3\ T_2$

(Start with the node that has no incoming edge.)

Q, G'12. Data items P and Q initialised to zero.

T_1 :	T_2 :
R (P) ;	R (Q)
R (Q) ;	R (P)
if P = 0 then Q++ ;	if Q = 0 then P++
W (Q) ;	W (P) ;

Any non-serial interleaving of T_1 and T_2 for concurrent execution leads to

a) A serializable schedule

b) A schedule that is not conflict serializable

c) A conflict serializable schedule.

d) A schedule for which a precedence graph can't be drawn.

→ Conflicts

$R_1(P)$, $W_2(P)$

$W_1(Q)$, $R_2(Q)$

Non serial combinations

$R_1(P)$	$R_1(P)$	$R_2(Q)$	$R_2(Q)$
$R_2(Q)$	$R_2(Q)$	$R_1(P)$	$R_1(P)$
$W_2(P)$	$W_1(Q)$	$W_1(Q)$	$W_2(P)$
$W_1(Q)$	$W_2(P)$	$W_2(P)$	$W_1(Q)$
$T_1 \rightleftarrows T_2$	$T_1 \rightleftarrows T_2$	$T_1 \rightleftarrows T_2$	$T_1 \rightleftarrows T_2$

None is conflict serializable.

There is no schedule which is conflict equivalent to any serial schedule.

Q G'16 S:

$r_2(x), r_1(x), r_2(y), w_1(x), r_1(y), w_2(x), a_1, a_2$

a - abort.

a) Non-recoverable

b) recoverable, but has cascading abort.

c) does not have a cascading abort (cascadeless)

d) strict

T_1	T_2
	$r(x)$
$r_1(x)$	
	$r(y)$
$w(x)$	
$r(y)$	
	$w(x)$
a_1	
	a_2

- **Recoverable** -

$W_i(x) \rightarrow R_j(x)$ | Commit of T_i before commit of T_j

$\Rightarrow C_i \rightarrow C_j$

- **Cascadeless** -

T_1	T_2	T_3
$R(x)$		
$W(x)$		
	$R(x)$	
	$W(x)$	
		$R(x)$

$W_i(x) \rightarrow R_j(x)$

$\Rightarrow C_i \rightarrow R_j$

| Commit of T_i before read of T_j

↳ If T_1 fails here, T_2, T_3 must also be rolled back.

T_1	T_2
$R(x)$	
	$R(Y)$
Commit	
	$R(x)$.

-- Cascadeless. strict. too

- **Strict** -

$(\text{Commit/abort})_i \rightarrow (R/W)_j$

always.

| Neither read nor write until last transaction commits/aborts.

2. View Serializability :

If a schedule is view equivalent to some serial schedule.

~ View Equivalence

2 schedules S_1 and S_2 each consisting of 2 transactions T_1 & T_2. S_1 and S_2 are called view equivalent if 3 conditions hold true -

i) For each data item X, if T_i reads X from the db initially in S_1, then in S_2 also, T_i must perform the initial read of X from the db.

Initial readers must be the same for all the data items.

ii) If T_i reads a data item that has been updated by T_j in S_1, then in S_2 also, T_i must read same data item that has been updated by T_j.

Write-read sequence must be same.

iii) For each data item X, if X has been updated at last by T_i in S_1, then in S_2 also, X must be updated at last by T_i.

Final writers must be same for all data items.

- Checking whether view serializable or not

Method 1

If the given schedule is conflict serializable, then it is surely view serializable.

If not conflict serializable, check using other methods.

Method 2.

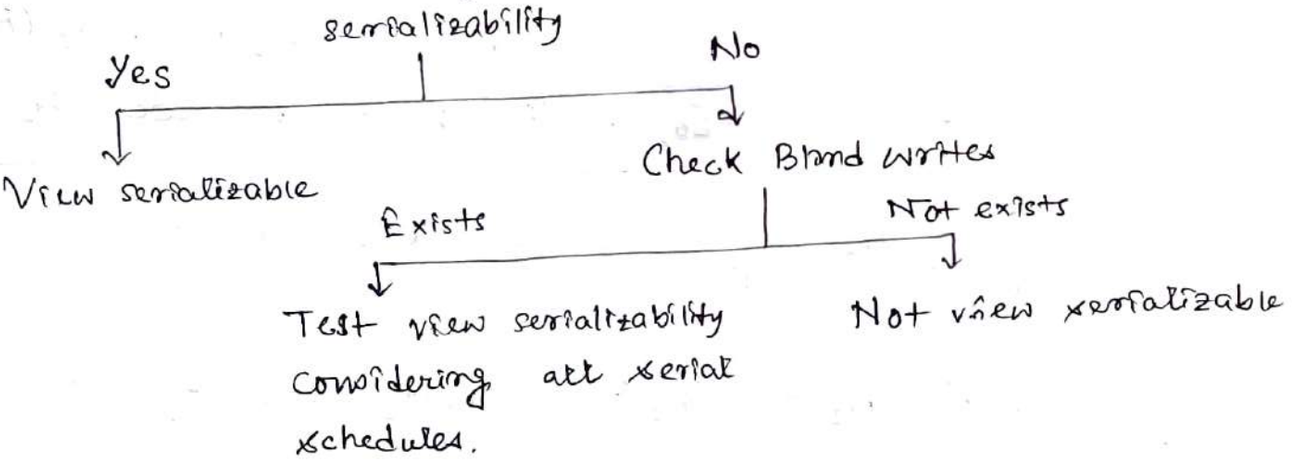
Check if there exists any blind write operation. If there is no blind write, then schedule is not view serializable.

If there is blind write, then check with other methods.

No blind write ⇒ Not view serializable.

Method 3 (Polygraph)

Try finding a view equivalent serial schedule. Using conditions for view equivalence, write all dependencies. Draw a graph using those dependencies. If no cycle, then view serializable, otherwise not. (Start from T which reads first, end at the T that writes last.)



Eg. T_1 T_2 T_3

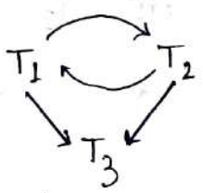
R(A) — Reads 1st

W(A)

W(A)

W(A) — Writes last

conflict serializability –

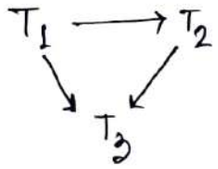


Not conflict serializable.

Blind writes – Has blind writes.

Method 3. (Polygraph)

T_1 must execute first. T_3 must execute last.



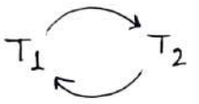
Possible schedule - T_1 T_2 T_3

There is no W-R ~~conflicts~~ dependency.

So, S is view serializable.

eg.

T_1	T_2
R(A)	
	W(A)
W(A)	



Not conflict serializable.

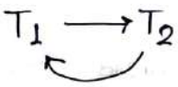
Has blind write.

$R_1 \rightarrow W_2$

$W_2 \rightarrow W_1$

Start with T_1.

End at ~~T1~~ T_1



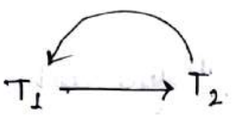
Cycle.

Not view serializable.

eg. $S: r_1(A)\ w_2(A)\ r_3(A)\ w_1(A)\ w_3(A)$.

**

T_1	T_2	T_3
r(A)		
	w(A)	
		r(A)
W(A)		
	W(A)	

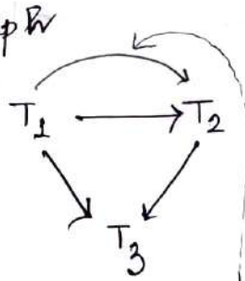


Not CS.

T_3

Blind write exists.

Polygraph



All conditions satisfied.

Write read sequence is $w_2(A), r_3(A)$.

So, in the schedule, too, the order $T_2 \rightarrow T_3$ must be there.

Also, no other transaction must not enter between $w_2(A)$ & $r_3(A)$.

So, $w_1(A)$ must precede $w_2(A)$.

So, view serializable.

T_1	T_2	T_3

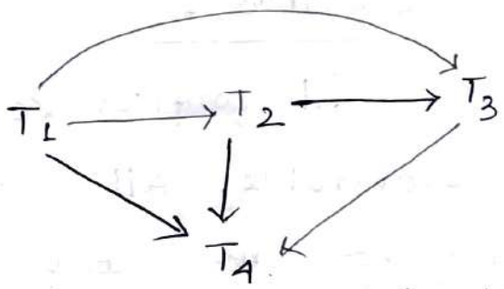
eg. $R_1(x),\ w_2(x),\ w_1(x),\ abort_2,\ commit_1$

T_1	T_2
R(x)	
	W(x)
W(x)	
	abort
commit	

As T_2 aborts, this is view serializable.

Q. Check view serializability. → Conflict serializability.

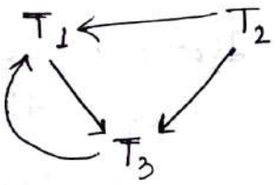
T_1	T_2	T_3	T_4
R(A)			
	R(A)		
		R(A)	
			R(A)
W(B)			
	W(B)		
		W(B)	
			W(B)



S is conflict serializable.
So, view serializable.

Q | T_1 | T_2 | T_3 | → CS.

R(A)		
	R(A)	
		W(A)
W(A)		

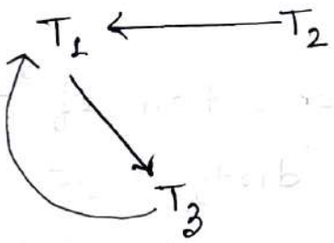


Not CS.

BW $W_3(A)$.

Dependency graph (Poly graph).

No write-read sequence.



Not view serializable.

Q. Check view serializability.

CS.  Not CS.

T_1	T_2
R(A)	
A += 10	R(A)
	A += 10
W(A)	
	W(A)
R(B)	
B += 20	
	R(B)
	B x = 1.1
W(B)	
	W(B)

BW No Blind writes.

Not View serializable.

* Checking recoverability.

Method 1. :

All conflict serializable schedules are recoverable. All recoverable schedules may or may not be conflict serializable.

So, check conflict serializability.

Method 2. :

If no dirty read, recoverable.

If dirty read exists,

i) If the commit operation of the transaction performing the dirty read occurs before the commit or abort operation of the transaction which updated the value, then irrecoverable.

ii) If the commit operation of the transaction performing the dirty read is delayed till the commit/abort operation of the transaction that updated the value, then recoverable.

* Types of Recoverable Schedules.

1. Cascading schedules. / Cascading rollback / abort

Failure of one transaction causes several other dependent transactions to roll back or abort.

T_1	T_2	T_3	T_4
R(A)			
W(A)			
	R(A)		
	W(A)		
		R(A)	
		W(A)	
			R(A)
			W(A)
failure			

Cascading recoverable schedule

2. Cascadeless : If a transaction is not allowed to read a data item until the last transaction that has written it is committed or aborted.

Allows only committed read operations.

But, allows uncommitted writes.

T_1	T_2
R(A)	
W(A)	
	W(A)
commit	

T_1	T_2	T_3
R(A)		
W(A)		
commit		
	R(A)	
	W(A)	
	commit	
		R(A)
		W(A)
		commit.

Cascadeless schedule.

3. Strict Schedule. : Allows only committed reads/ writes.

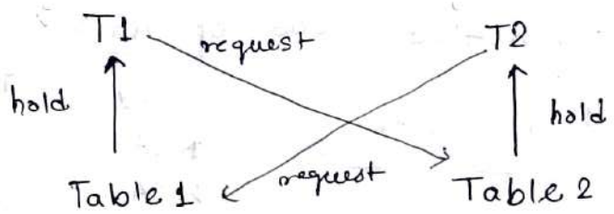
T_1	T_2
W(A)	
commit/ rollback	
	R(A)/W(A)

* Concurrency Control Protocols.

1. Lock based protocols.
   a) 2 phase locking protocols (2 PL)
      - Basic 2 PL
      - Conservative 2 PL
      - Strict 2 PL
      - Rigorous 2 PL
   b) Graph based protocol
2. Time xtamp based protocol
   a) Time xtamp ordering protocol
   b) Thomas write rule.
3. Multiple granularity protocol
4. Multi version protocol
   a) Multi version 2 PL
   b) Multi version time xtamp ordering.

• Deadlock in DBMS.

2 or more transactions are waiting indefinitely for one another to give up locks.



- Deadlock avoidance.

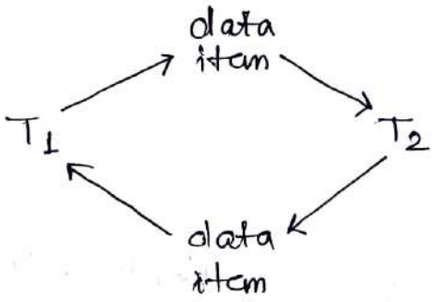
Suitable for smaller database.

T_1 waits for T_2 to release lock before it begins. When T_2 releases lock, T_1 proceeds freely.

- Deadlock detection

Wait for graph

cycle ⇒ deadlock



- Deadlock prevention

Suitable for large database.

i) Wait-die scheme.

If a transaction requests for a resource that is locked by other transaction, then the DBMS checks the timestamp of both transactions & allows the older transaction to wait until the resource is available for execution.

If a younger transaction has locked some resource & older transaction is waiting for it, then older transaction is allowed to wait for it till it is available. If the older transaction has held some resource and younger transaction waits for the

resource, then younger transaction is killed & restarted with minute delay with same timestamp

ii) Wound wait scheme.

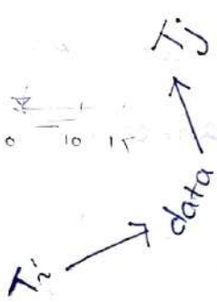
If an older transaction requests for a resource held by younger transaction, then older transaction forces younger transaction to kill itself & release resource. If the younger transaction is requesting a resource which is held by older one, then younger transaction is asked to wait till older releases it. (Older priority)

T_i requests data held by T_j	Wait die	Wound wait
T_i older than T_j ($TS(T_i) < TS(T_j)$)	T_i waits	T_j aborts
T_i younger than T_j ($TS(T_i) > TS(T_j)$)	T_j dies	T_i waits

Younger T_n holding data ⇒ older waits.
Older T_n holding data ⇒ young killed.

Wait-die

1. $TS(T_i) < TS(T_j)$
⇒ T_i is allowed to wait until data available.
2. $TS(T_i) > TS(T_j)$
⇒ T_i dies. T_i is restarted later.



Wound - wait

1. $TS(T_i) < TS(T_j)$
⇒ T_i forces T_j to roll back. T_i wounds T_j. T_j is restarted later.
2. $TS(T_i) > TS(T_j)$
⇒ T_i is forced to wait until the resource is available.

- **Starvation in DBMS.**

A transaction has to wait for an indefinite period of time to acquire a lock.

– Reasons – unfair waiting scheme, victim selection, resource leak, via denial-of-service attack.

• Concurrency Control protocols (CC protocols) allow concurrent schedules, but ensure that the schedules are conflict/view serializable & are recoverable & maybe even cascadeless.

These protocols do not examine the precedence graph as it is being created, instead a protocol imposes a discipline that avoids non-serializable schedules.

* Lock based Protocols.

A lock is a variable associated with a data item that describes a status of data item wrt possible operation that can be applied to it. It is required in this protocol that all the data items must be accessed in a mutually exclusive manner.

- Shared lock (S): Read-only lock. It can be shared between transactions because while holding this lock the transaction does not have the permission to update data on the data item. S-lock is requested using lock-S instruction.

- Exclusive lock (X): Data item can be both read as well as written. This is exclusive & cannot be held simultaneously on the same data item. X-lock is requested using lock-X instruction.

Lock compatibility matrix –

	S	X
S	Yes	No
X	No	No

A T may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other Ts.

Any no. of Ts can hold shared locks on an item, but if any T holds an exclusive lock on an item no other T may hold any lock on the item.

If a lock can't be granted, the requesting T is made to wait till all ~~~~ incompatible locks held by other Ts have been released. Then the lock is granted.

- Upgrade / downgrade Locks.

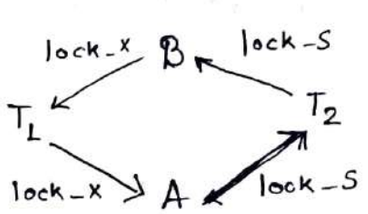
A T that holds a lock on an item A is allowed under certain condition to change the lock state from one state to another.

upgrade - A S(A) can be upgraded to X(A) if T_i is the only T holding the S-lock on element A.

downgrade - We may downgrade X(A) to S(A) when we feel that we no longer want to write on data item A. As we were holding X-lock on A, we need not check any conditions.

- Problem with simple locking.

T_1	T_2
lock-X(B)	
R(B)	
B -= 50	
W(B)	
	Lock-S(A)
	R(A)
	lock-S(B)
lock-X(A)	



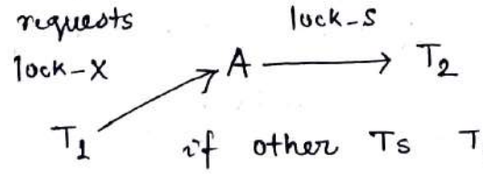
Deadlock.

Deadlock

None can proceed with execution.

Starvation

Also possible if concurrency control manager is badly designed.



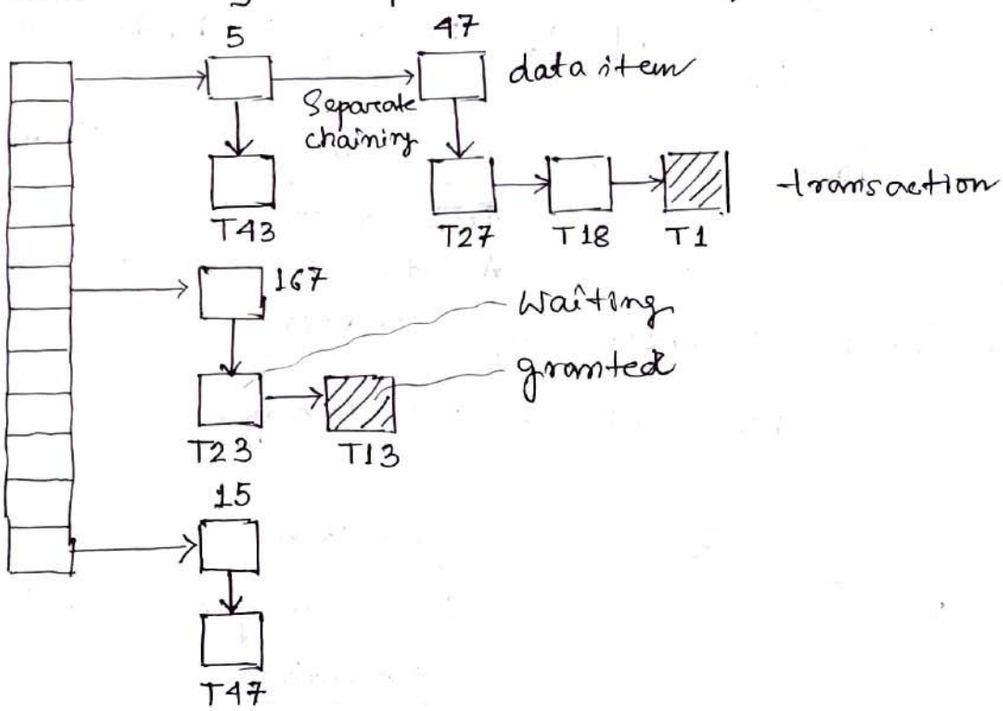
if other Ts $T_3, T_4, \ldots$ requests shared lock those will be granted and T_1 will starve.

- Implementation of locking

Mechanism to manage the locking requests made by Ts — Lock manager.

Data structure reqd. for implementation of locking is called a lock table.

Lock table is a hash table where name of data items are used as hashing index. Each locked data item has a linked list associated with it. Each node in the LL represents the T which requested for lock, mode of lock requested (S/X) & current status of the request (granted/waiting). Every new lock request for the data item will be added in the end of LL as a new node. Collisions are handled by separate chaining.



- ## 2 Phase Locking Protocol (2-PL)

A T is said to follow 2-PL protocol if locking & unlocking can be done in 2 phases:

i) Growing phase - New locks on data items may be acquired but none can be released.

ii) Shrinking phase - Existing locks may be released but no new locks can be acquired.

If lock conversion is allowed, then upgrading of lock (from S(a) to X(a)) is allowed in Growing phase & downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

Transactions implementing 2-PL.

T_1	T_2	
lock_S(A)		growing
	lock_S(A)	growing
lock_X(B)		growing
.....	.....	
unlock(A)		shrinking
	lock_X(C)	growing
unlock(B)		shrinking
	unlock(A)	shrinking
	unlock(C)	shrinking

Lock point - Point at which growing phase ends.

2-PL ensures serializability.

Drawbacks of 2-PL

i) Cascading roll back
ii) Deadlock
iii) Starvation

Cascading rollback in 2-PL.

T_1	T_2
lock_X(A)	
R(A)	
W(A)	
unlock(B)	
	lock_S(A)
	R(A)
fails.	dirty read

Deadlock in 2-PL

T_1	T_2
lock_X(A)	
	lock_X(B)
lock_X(B)	
	lock_X(A)

loop in precedence graph.

- 2PL also reduce the concurrency as a T may not be able to release a data item after it has used it.
- Basic 2-PL : Ensures conflict serializability, but does not prevent cascading rollback, deadlock.

– Enhancements to 2PL

i) Strict 2-PL
ii) Rigorous 2-PL
iii) Conservative 2-PL.

Strict 2-PL : This requires that in addition to the lock being 2-phase all exclusive locks held by the transaction be released until after the T commits, Strict 2-PL ensures recoverability and cascadelessness.

It gives freedom from cascading abort that was there in basic 2-PL & moreover guarantee strict schedules, but deadlocks are possible.

Rigorous 2-PL : In addition to the lock being 2 phase all exclusive and shared locks held by the T be released until after the transaction commits. Rigorous 2-PL ensures recoverability & cascadelessness.

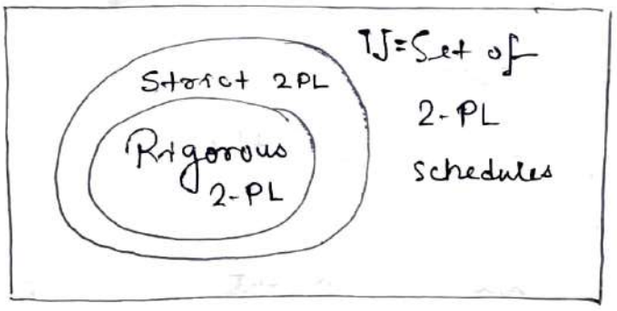
Problem of deadlocks is still there.

Conservative 2-PL. : AKA static 2-PL.

This protocol requires the T to lock all the items it accesses before the T begins execution by predeclaring its read-set & write-set. If any of the predeclared items needed can't be locked, the T does not lock any of the items, instead it waits until all the items are available for locking.

Conservative 2-PL is deadlock free but it does not ensure strict schedule. It's difficult

to use in practice as predeclaring the read-set & write-set is not always possible. In practice, most popular 2-PL is strict 2-PL.



- Schedule following conservative 2-PL will not have a growing phase. Locking before using makes it deadlock free. No hold and wait ensures deadlock free schedule.

- We only have to lock all the items beforehand, so releasing or unlocking them has no restrictions like we had in strict or rigorous 2-PL.

- Conservative 2-PL may face cascading rollbacks. So, it does not ensure strict schedules.

T_1	T_2
lock_X (A)	
lock_X (B)	
R(A)	
W(A)	
unlock (A)	
unlock (B)	
	lock_X (A)
	R(A)
	W(A)
	unlock (A)
	commit
commit	

Conservative 2-PL, not strict or rigorous

- Cascading abort in conservative 2-PL.

If there is a dirty read operation (as there is no restriction to prevent it), cascading rollback may happen.

* Graph based cc protocol.

Another way of implementing lock based protocols. Tree based protocol is a simple implementation of graph based protocol.

A prerequisite of this protocol is that we know the order to access a database item. For this, we implement a partial ordering on a set of the database items (D) $\{d_1, d_2, \ldots\}$.

Protocol following the implementation of partial ordering is stated as -

- If $d_i \rightarrow d_j$, then any transaction accessing both d_i and d_j must access d_i before accessing d_j.

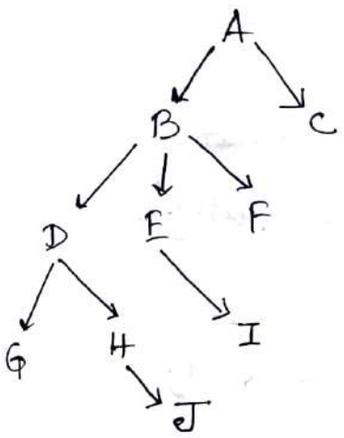
It implies that the set D may now be viewed as a DAG, called a database graph.

Tree based protocol

- Partial order on database items determines a tree like structure.
- Only exclusive locks are allowed.
- The first lock by T_i may be on any data item. Subsequently, a data Q can be locked by T_i only if the parent of Q is currently locked by T_i.
- Data items can be unlocked at any time.

Tree based protocol ensures conflict serializability and no deadlock. We need not wait for unlocking a data item as of 2-PL protocol, thus it increases concurrency.

eg. Database graph



T_1	T_2	T_3
lock_x(B)		
	lock_x(D)	
	lock_x(H)	
	unlock_x(D)	
lock_x(E)		
lock_x(D)		
unlock_x(B)		
unlock_x(E)		
		lock_x(B)
		lock_x(E)
	unlock_x(H)	
lock_x(G)		
unlock_x(D)		
		unlock_x(E)
		unlock_x(B)
unlock_x(G).		

Serializable -

- Advantages

i) Ensures conflict serializability

ii) Ensures deadlock free schedule

iii) Unlocking can be done at anytime.

- Disadvantages.

i) Unnecessary locking overheads (if we want both D & E, then at least B has to be locked)

ii) Cascading rollbacks.

* Timestamp based concurrency protocol.

Timestamp is a unique identifier created by the DBMS to identify a transaction. They are usually assigned in the order in which they are submitted to the system. Timestamp for T_i is $TS(T_i)$.

- Protocol :

Order the transactions based on their timestamps. The schedule is equivalent to the particular serial order corresponding to the order of the transaction timestamps. For each items accessed by conflicting operations in the schedule, the order in

which the item is accessed does not violate the ordering. We use 2 timestamp values relating to each database item X.

W_TS(X) : Largest TS of any T that executed W(X) successfully.

R_TS(X) : Largest TS of any T that executed R(X) successfully.

- Basic TS ordering :

Every T is issued a TS based on when it enters the system. If an old T T_i has TS $TS(T_i)$, a new T T_j is assigned $TS(T_j)$ such that $TS(T_i) < TS(T_j)$. The TS ordering protocol ensures that any conflicting read & write operations are executed in timestamp order. Whenever some T tries to issue a Read(X) or Write(X), the basic TS ordering algorithm compares the TS of T with R_TS(X) & W_TS(X) to ensure that the TS order is not violated.

→ If a T T_i issues a Read(X) -
- if $TS(T_i) < W\_TS(X)$
  operation rejected (abort T_i)
- if $TS(T_i) >= W\_TS(X)$
  operation executed (execute Read(X))
  Set R_TS(X) to larger of TS(T) & current R_TS(X)
- All data-item timestamps updated.

→ If a T T_i issues a Write(X) -
- if $TS(T_i) < R\_TS(X)$
  operation rejected & T_i rolled back
- if $TS(T_i) < W\_TS(X)$
  operation rejected & T_i rolled back
- Otherwise, operation executed. ($W\_TS(X) = TS(T)$)

Whenever the basic TO protocol detects 2 conflicting operations that occur in incorrect order, it rejects the later of the 2 operations by aborting the T that issued it. Schedules produced by basic TO are guaranteed to be conflict serializable.

- Cascading rollback in TSO.

We have a T T_1 and T_2 has used a value written by T_1. If T_1 is aborted & resubmitted to the system then T_2 must also be aborted & rolled back.

- Advantages.

i) TSOP ensures serializability since the precedence graph will be of the form -

TS^n with smaller TS ⟶ TS^n with larger TS

ii) Ensures freedom from deadlock as no T ever waits.

- Disadvantage

Cascading rollback. May not be recoverable.

- Strict TS ordering

Ensures schedules to be strict & conflict serializable.

T that issues a Read (X) or Write (X) s.t. TS (T) > W_TS (X) has its read or write operation delayed until the T that wrote the values of X has committed or aborted.

Q. G'17 In a db system unique TSs are assigned to each T using Lamport's logical clock. Let TS(T1) & TS(T2) be the TSs for T1 & T2. T1 holds a lock on R, & T_2 has requested a conflicting lock on same R. Following algorithm is used to prevent deadlocks on the db assuming that a killed T is restarted with the same TS.

if- TS(T2) < TS(T1) then // T_2 is old
    T1 is killed
else T_2 waits.

Wound-wait

Assume any Ts that is not killed terminates eventually. Which is true?

a) db system is both deadlock free & starvation free
b) deadlock free but not starvation free.
c) sf but not df
d) neither sf- nor df.

$T_1 \leftarrow R \leftarrow T_2$

→ Basic TSOP is deadlock free.

Wound-wait is also starvation free as it restarts the aborted T with same original TS.

Wound wait → no starvation → no deadlock

(90)

In wound wait, a younger transaction is allowed to wait for an older one, whereas an older T requesting an item held by a younger T preempts the younger of the 2 by aborting it.

- Remembrance.

Wait - die.

Wants resource	holds resource
A (t=7) waits	B (t=11)
B (t=11) dies.	A (t=7)

Wound - wait

Wants resource	holds resource
A (t=7) preempts	B (t=11)
B (t=11) waits	A (t=7)

- Row level of locking provides highest level of concurrency.

→ Thomas Write Rule.

Modification on the basic TSOP.

In TWR, user ignore outdated writes.

In TWR, most imp. improvement is user can achieve concurrency with view serializable schedules.

This rule states if $TS(T_i) < W\_TS(X)$, then operation is rejected and T_i is rolled back.

i) If $R\_TS(X) > TS(T)$, then abort & rollback T.

ii) If $W\_TS(X) > TS(T)$, then don't execute the write operation & continue processing. This is a case of outdated / obsolete writes.

iii) If i, ii don't occur, then only execute the Write (X) operation of T and set $W\_TS(X)$ to $TS(T)$.

Outdated write example –

$TS(T_1) < TS(T_2)$.

T_1	T_2
	$R(A)$
$W(A)$	
	$W(A)$ →

$T_1 \rightarrow T_2$ not allowed. Ignore the outdated write.

- Basic TSOP vs TWR.

$TS(T_1) < TS(T_2)$

Basic TSOP

Not allowed –

~~$R_1(X)$ $R_2(X)$~~

$R_2(X)$ $W_1(X)$

$W_2(X)$ $R_1(X)$

$W_2(X)$ $W_1(X)$

Allowed –

$R_1(A)$ $W_2(A)$

$R_2(A)$ $R_1(A)$

$W_1(A)$ $R_2(A)$

$W_1(A)$ $W_2(A)$.

TWR.

Not allowed –

$R_2(A)$ $W_1(A)$

$W_2(A)$ $R_1(A)$

Allowed –

$R_2(A)$ $R_1(A)$

$W_2(A)$ $W_1(A)$.

└ ignore.

Atomicity: All changes to data are performed as if they are a single operation. That is, all the changes are performed or none of them are. In transfer of funds, atomicity ensures if a debit is made successfully from one account, corresponding credit is made to other account.

Consistency: Data is in a consistent state when a "tran" starts & when it ends. In transfer of funds, consistency ensures that the total value of funds in both the accounts is the same at the start and end of each T.

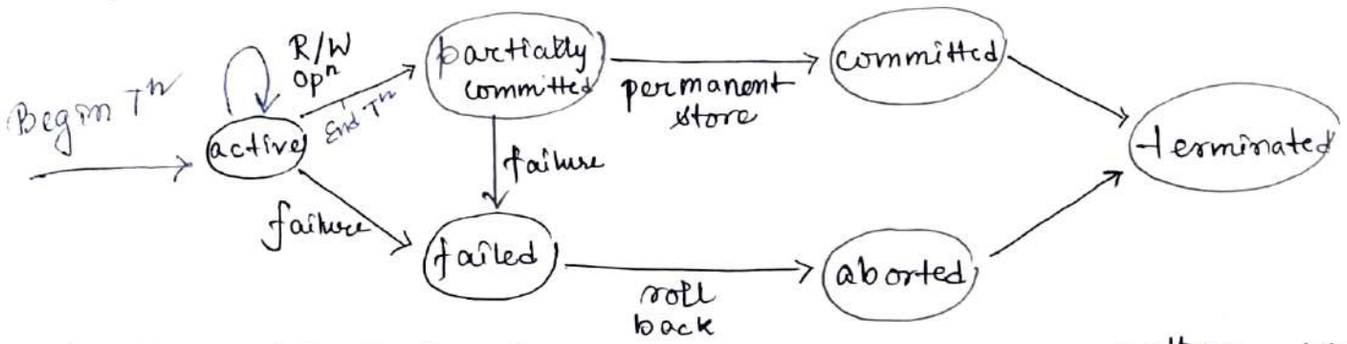
Isolation: The intermediate state of a T is invisible to other Ts. As a result, Ts that run concurrently appear to be serialized. In transfer of funds, isolation ensures that another T sees the transferred funds in one account or the other, but not in both, nor in neither.

Durability: After a T completes successfully, changes to data persist & are not undone, even in the event of system failure. In transfer of funds, durability ensures that changes made to each account will not be reversed.

Levels of isolation :

1. Level 0 : T^n in level 0 isolation if it does not overwrite the dirty reads of higher level T^n s.
2. Level 1 : No lost updates.
3. Level 2 : No lost updates and no dirty reads.
4. Level 3 : True isolation. In addition to level 2 properties, it has repeatable reads.

- Transaction states.



i) active state : As long as instructions are getting executed. All changes are stored in the buffer in MM.

ii) partially committed state : After the last instruction of transaction has executed, it enters into this state. It's not fully committed as the changes are still stored in the buffer in MM.

iii) committed : After all the changes have been stored into the database. After a transaction T has entered the committed state, it's not possible to roll back. (compensating transaction can be done to reverse operation)

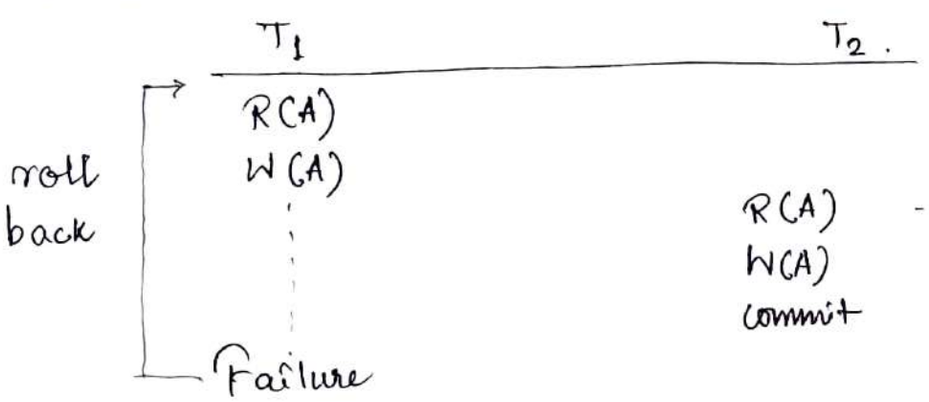
iv) failed state : When it becomes impossible to continue the execution.

v) aborted state : After the transaction has rolled back completely after going to failed state.

vi) terminated state : After entering committed or aborted, T enters terminated state.

- Concurrency problems. (Why concurrency control needed ?)

i) Dirty read problem. (Temporary update) : reading data written by an uncommitted transaction.



ii) Unrepeatable read problem.:

A transaction gets to read unrepeated i.e. different values of the same variable in its different read operations even when it has not updated its value.

T_1	T_2
R(X)	
	R(X)
W(X)	
	R(X) ... unrepeated read

iii) Lost update problem: When multiple transactions execute concurrently & updates from one or more transactions get lost.

T_1	T_2
R(A)	
W(A)	
⋮	W(A)
⋮	commit
commit	

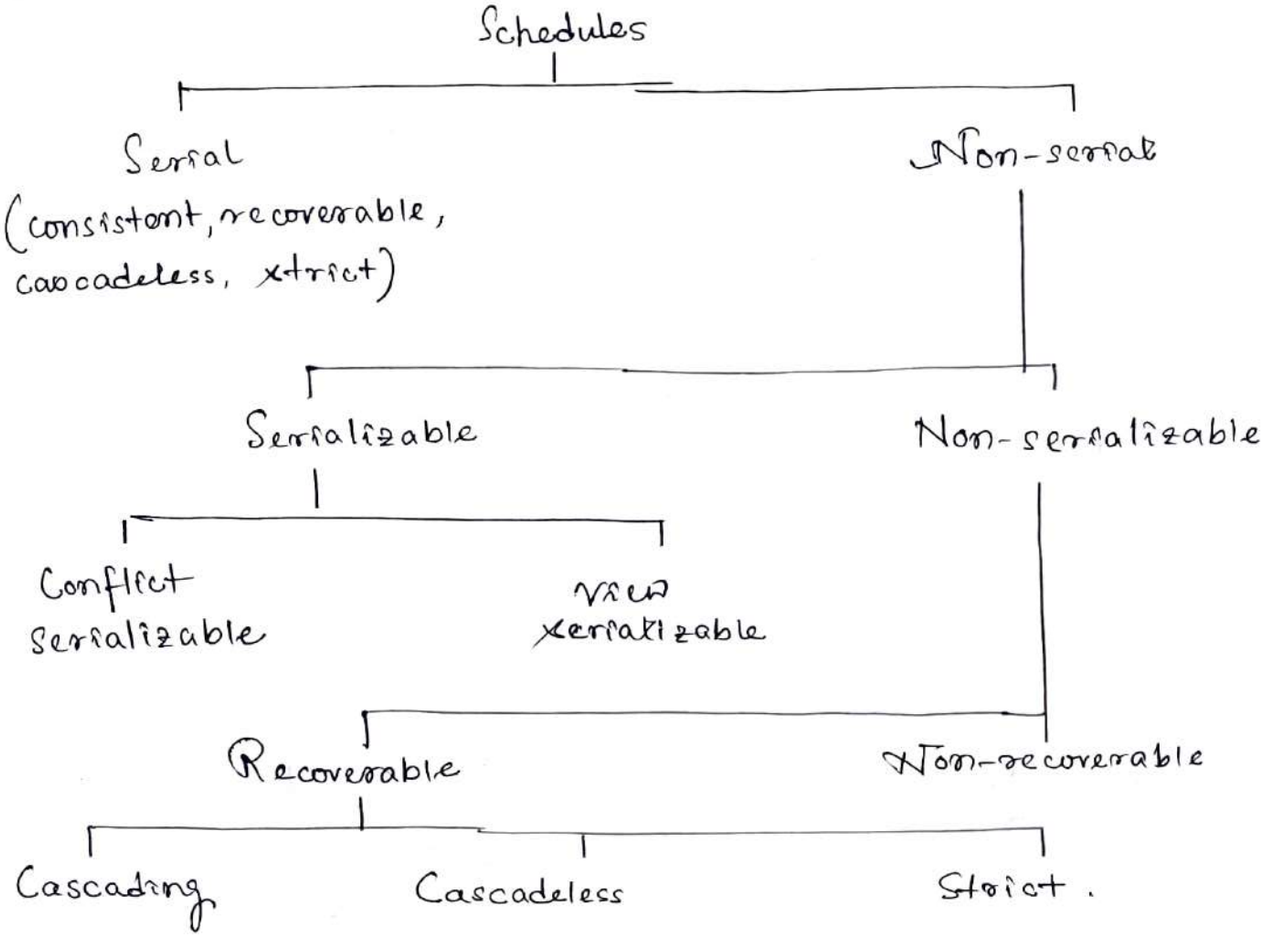
Occurs whenever there is a write-write conflict.

iv) Phantom read problem.: When a transaction reads some variable from the buffer & when it reads the same variable later it finds the variable does not exist.

T_1	T_2
R(X)	
	R(X)
Delete(X)	
	R(X)

T_2 wonders who deleted X as according to it, it's running in isolation.

* Types of schedules.



1. Serial.

T_1	T_2
R(A)	
W(A)	
R(B)	
	W(B)
	R(A)
	R(B)

2. Non-serial.

2.a. Serializable

2.a.i. Conflict Serializable. (Can be transformed into a serial schedule by swapping non-conflicting operations)

2.a.ii) View Serializable. (If it's view equal to a serial schedule.)

2.b. Non-serializable.

2.b.i. Recoverable. (Schedules in which transactions commit only after all transactions whose changes they read commit)

T_1	T_2
R(A)	
W(A)	
	W(A)
	R(A)
C_1	
	C_2

2.b.i. A. Cascading

B. Cascadeless (disallow transaction from reading uncommitted changes

C. Strict.

eg.

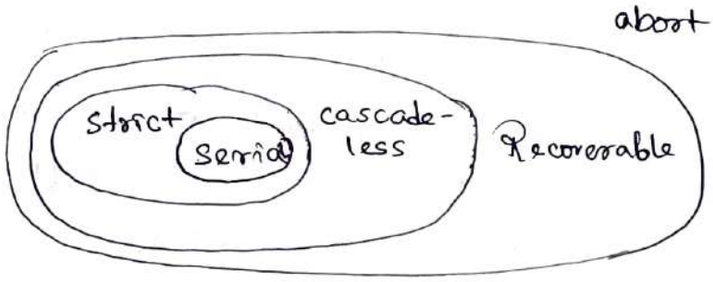
T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
abort	
	abort

Recoverable but not cascadeless.

2.b.ii. Non-recoverable.

(Student copying from another student analogy)

T_1	T_2
R(A)	
W(A)	
	W(A)
	R(A)
	commit
abort	



- Cascadeless schedules are stricter than recoverable schedules or are a subset of recoverable sched
- Strict schedules are a subset of cascadeless schedules.
- Serial schedules satisfy constraints of all recoverable, cascadeless, & strict schedules.

* Equivalence of Schedules.

i) Result equivalence. If any 2 schedules generate the same result (→ database state) after their execⁿ.

This equivalence is considered of least significance, as some schedules might produce same results for some set of values & different results for some other set of values.

ii) Conflict equivalence. If 2 schedules satisfy -

a) Set of transactions is same.

b) Order of pairs of conflicting operations of both the schedules is same.

iii) View equivalence. If initial read and final write are same and also write-read sequence must match.

* 2 operations in a schedule are said to conflict if they satisfy all 3 -

1. They belong to different transactions.
2. They access the same item X.
3. At least one of the operations is a write_item(X).

2 operations are conflicting if changing their order can result in a different outcome.

* Complete schedule: S of n Ts $T_1, \ldots, T_n$ is complete if -

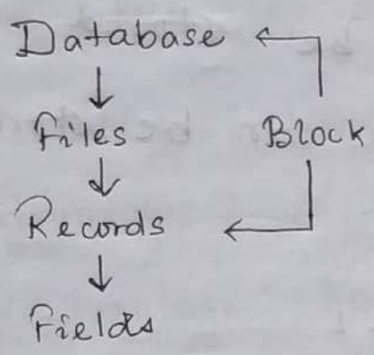
i) Operations in S are exactly those opⁿs in $T_1, T_2, \ldots, T_n$ including a commit or abort opⁿ as the last opⁿ for each T_n. ii) for any pair of opⁿs for the same T_n T_i, their relative order of appearance in S is same as order of op. in T_i.

iii) For any 2 conflicting opⁿs one of the 2 must occur before the other in schedule.

A complete schedule will not contain any active Tns at the end of schedule.

File Structures.

* File organization.



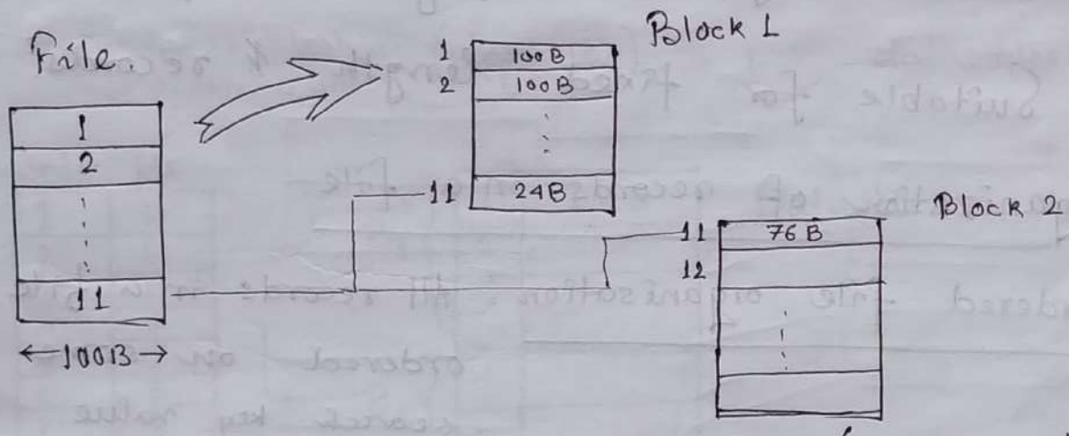
Blocking factor ~
average number of records per block.

* Mapping Strategies for file records into blocks.

A file is a collection of related records stored on the secondary storage.

i) Spanned mapping :

In spanned mapping, the record of a file is stored inside the block even if it can only be stored partially & hence, the record is spanned over 2 blocks giving it the name.



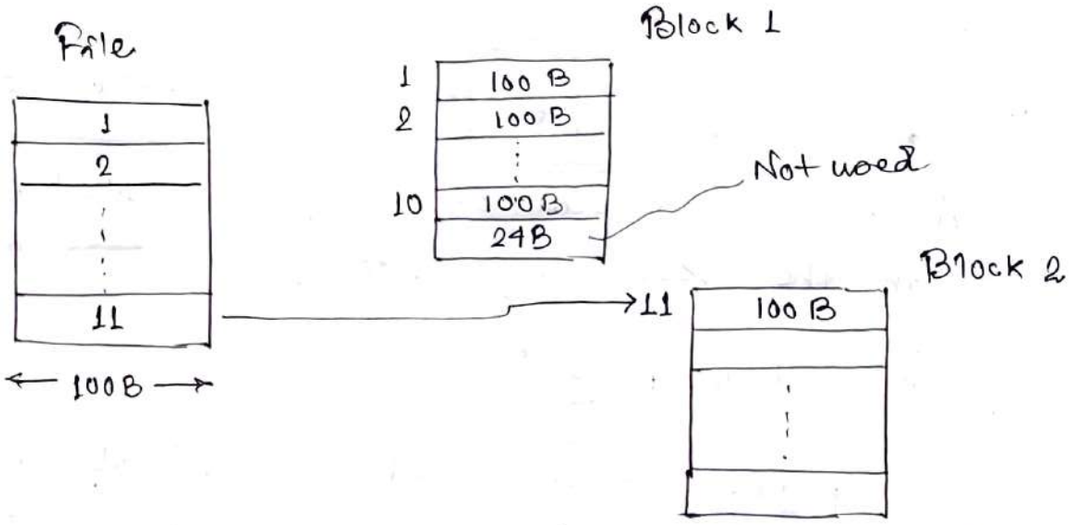
Adv. : No wastage of memory (no internal fragmentation)

Disadv. : No. of block accesses to access a record increases.

This mapping is suitable for variable length records.

ii) Unspanned Mapping

The record of a file is stored inside the block only if it can be stored completely inside it. No record can be stored in more than 1 blocks.



Adv.: Access time of a record is less.

Disadv.: Wastage of memory (internal fragmentation).

Suitable for fixed length records.

* Organization of records in a file.

i) Ordered file organisation: All records in a file are ordered on some search key value.

Searching: Binary search

Adv.: Searching a record is efficient

Disadv.: Insertion is expensive due to reorganisation of the entire file.

SSN	
1	
2	
3	
⋮	

ii) Unordered file organization. : All records in file are inserted wherever the place is available, usually at the end of file.

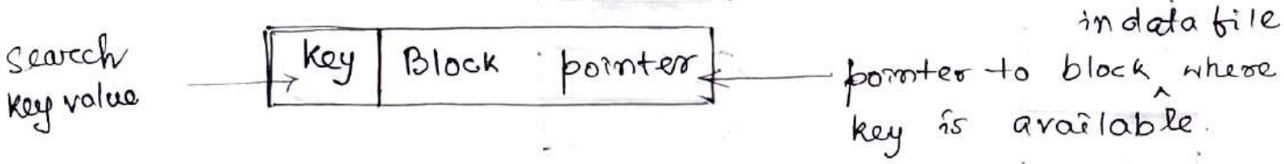
Searching : Linear search

Advantages : Inserting a record is efficient

Disadvantage : Searching a record is inefficient compared to ordered file organisation.

* Structure of index files.

→ Index record consists of 2 fields.

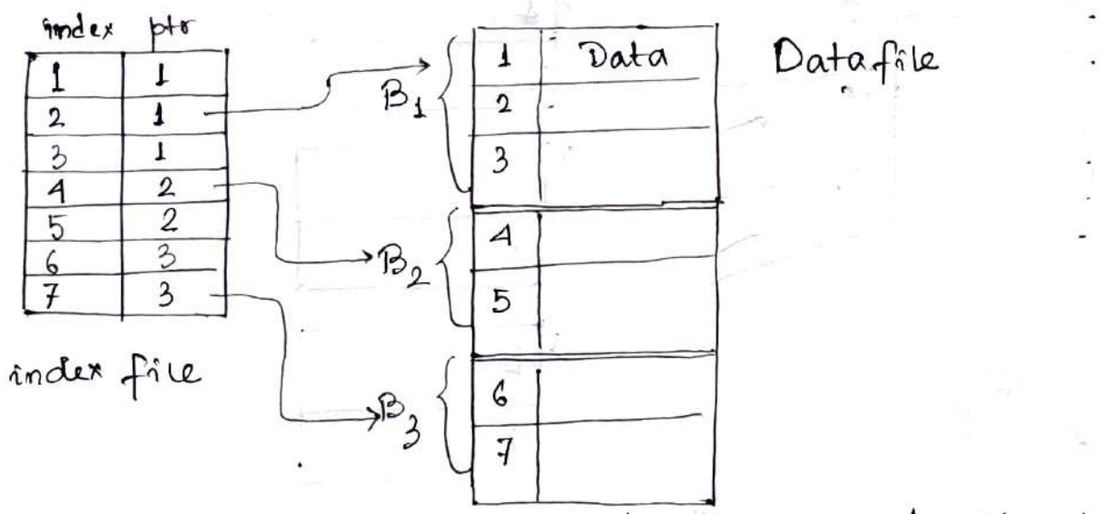


→ Index is an ordered file.

→ Searching : Binary search

→ To access a record using index, the average number of block access =

index block access count $(\log_2 B_i) + 1$ data block access

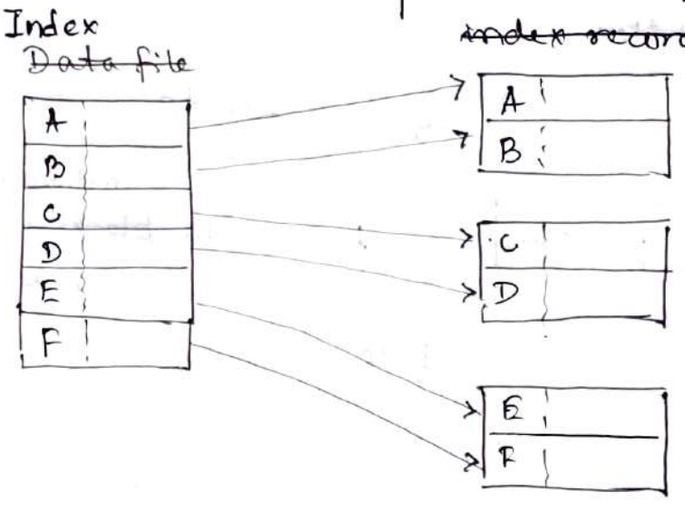


→ Index can be created on any field of relation (Primary key, non key, candidate key).

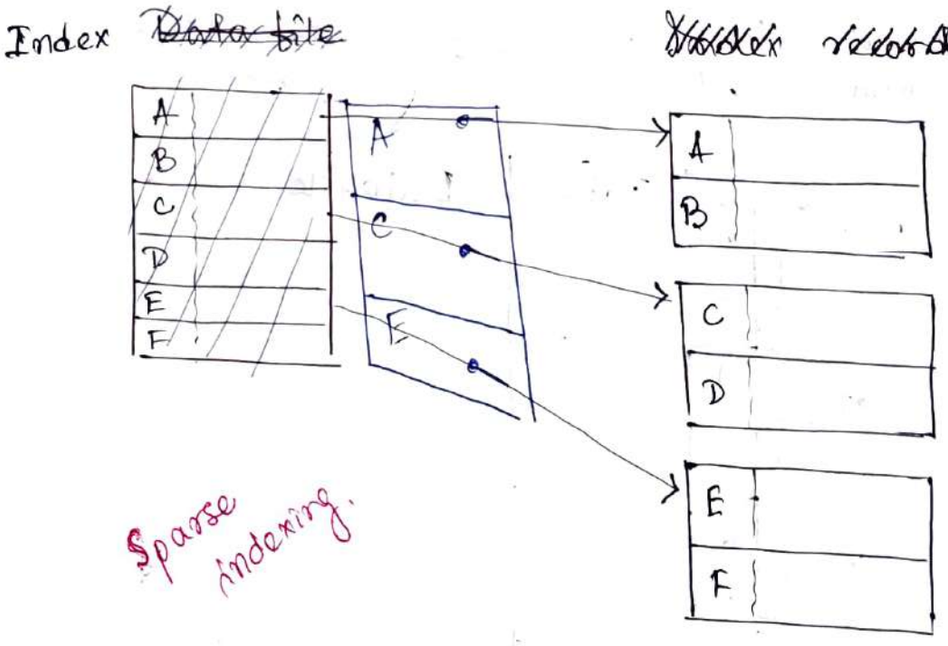
* Classification of indexing.

1. Dense indexing: If an index entry is created for every search key value, then it is called dense indexing.

2. Sparse indexing: If an index entry is created only for some records, then it is called sparse indexing.

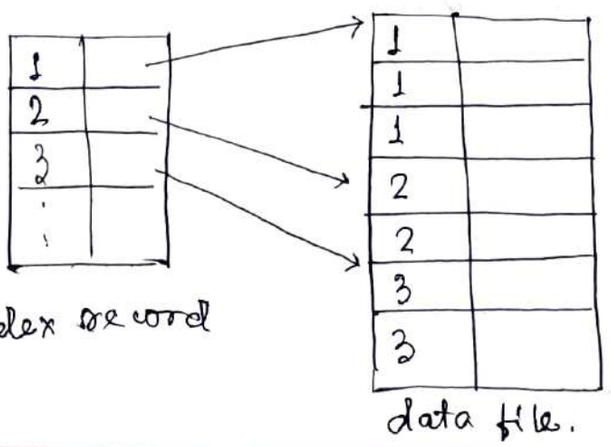


Dense indexing



Sparse indexing.

Both dense and sparse indexing.



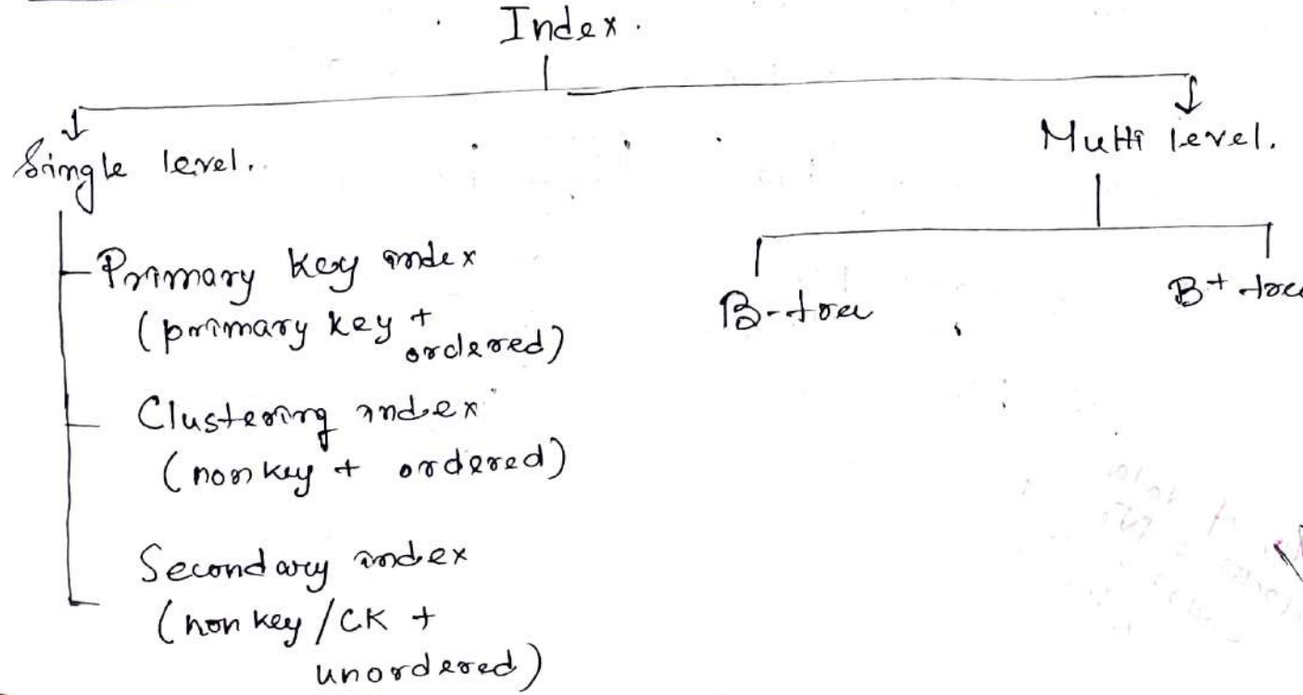
— Dense indexing: The record contains the search key & also a reference to the first data record with that search key value.

— Sparse indexing: The index record appears only for a few times in the data file. Each item points to a block. To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for. We start at the record pointed by the index record, & proceed along with the pointers in the file, until we find the desired record.

* Attributes of indexing:

1. Access types. : Value based search, range access etc.
2. Access time : Time needed to find particular data element or set of elements.
3. Insertion time : Time taken to find appropriate space & insert a new data.
4. Deletion time : finding item & deleting it.
5. Space overhead : Additional space required by the index.

* Types of indexing



* Primary Index. (Ordered file on primary key)

Ordered file whose records are of fixed length with 2 fields. The first field is same as primary key of data file & the second field is a pointer to data block, where the key is available.

- Index created for first record of each block - block anchors. (Sparse indexing)
- The number of index entries is equal to the number of data blocks.
- The average number of block accesses using index $= (\log_2 B_i) + 1$ | B_i - No. of index blocks.

Q. We have an ordered file of 30000 records on a disk with a block size of 1024 B. Records are of fixed size and are unspanned (size 100B). Suppose we have created primary index on the key field of the file of size 9 B & a block ptr. of size 6B. Find the avg no. of block accesses to search a record using with & without index?

→ Block size = 1024 B

Data record size = 100B

blocking factor

Data records / block $= \left\lfloor \frac{1024}{100} \right\rfloor \approx 10$ (unspanned)

No. of data blocks in data file $= \frac{30000}{10} = 3000$

No. of block accesses $= \lceil \log_2 3000 \rceil = 12$ (Ans)

Without indexing

No. of data blocks = No. of entries in index record.

Index record = (9+6) = 15B.

Index records/block = $\left\lfloor \frac{1024}{15} \right\rfloor$ = 68. — blocking factor in case of indexing

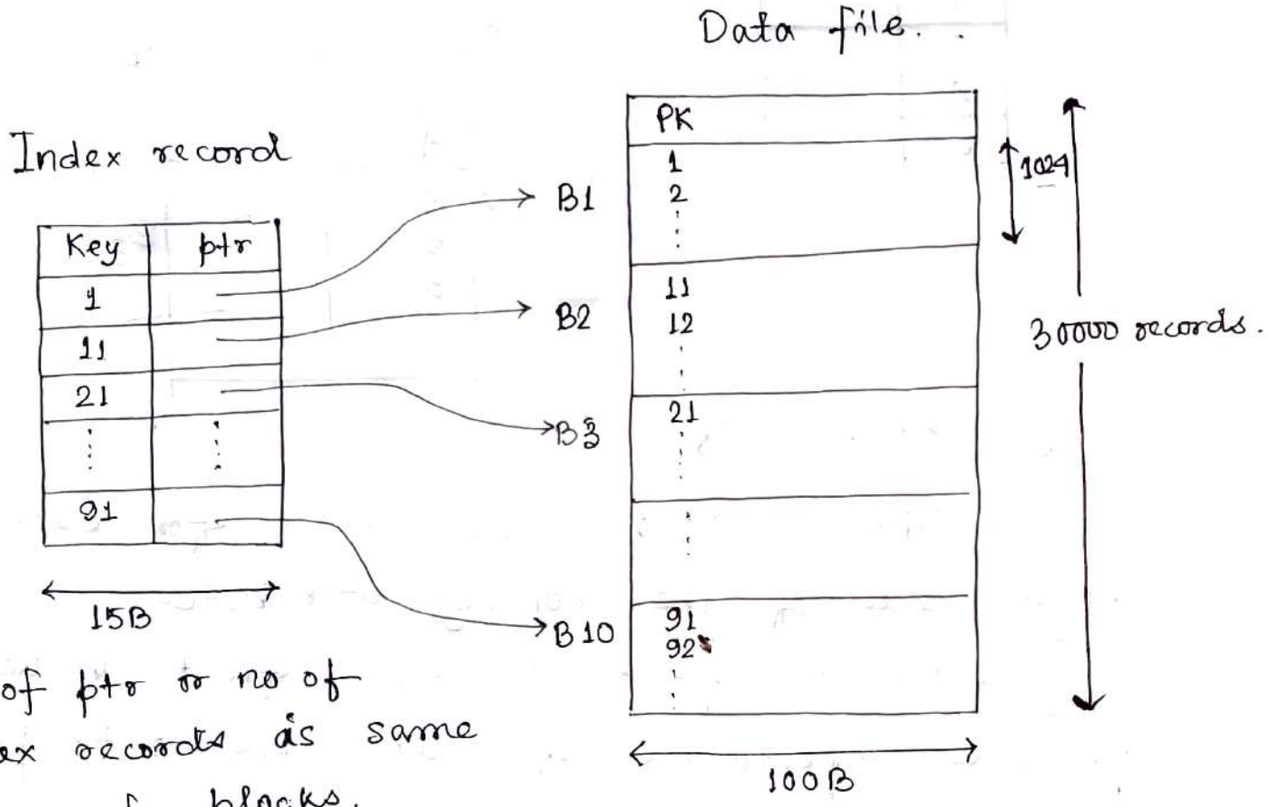
Total index records = 3000 (sparse)

No. of blocks reqd = $\left\lceil \frac{3000}{68} \right\rceil$ = 45.

No. of block accesses = $(\log_2 45) + 1$ = 7. Ans

With indexing

- Data file is sorted on primary indexing.



No. of ptr or no of index records is same as no. of blocks.

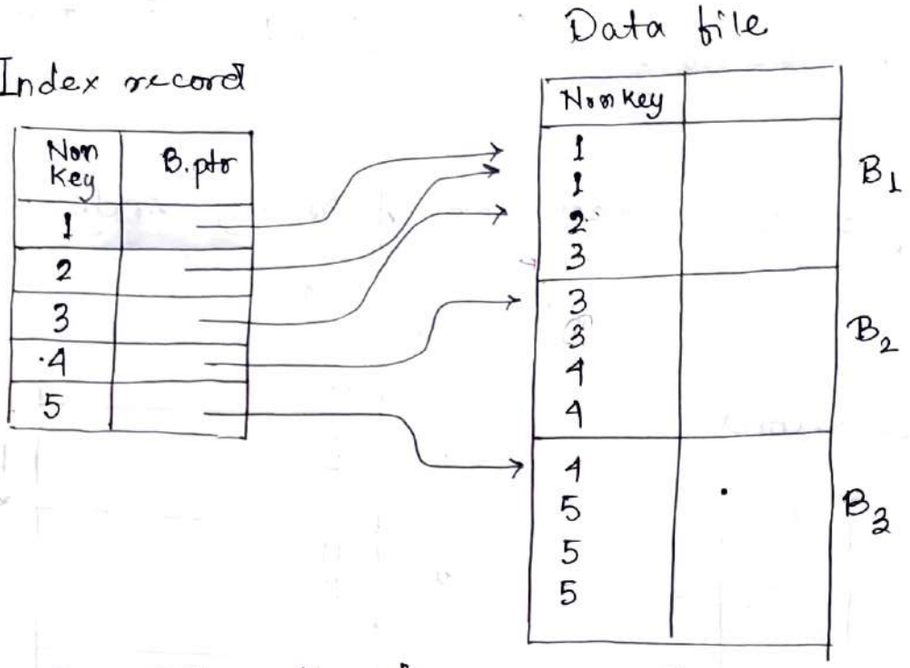
- Blocking factor = No. of records in a block

= $\left\lfloor \frac{\text{block size}}{\text{record size}} \right\rfloor$ [UnSpanned].

- No. of blocks in main data file = $\left\lceil \frac{\text{no. of records in data file}}{\text{Blocking factor}} \right\rceil$

* Clustering Index. (Ordered file on non-key)

Created on data file whose records are physically ordered on a non key field which does not have distinct value for each record; the field is called clustering field.



- Example of dense as well as sparse indexing.

- There will be one entry for each unique value of the non-key attributes.

- Pointer points to the first value; for next repeatations of value we don't point to them.

- If no. of blocks acquired by index file is n, then no. of block accesses req. $\geq = (\log_2 n) + 1$

> because we may need to search at some other blocks as block ptr points to the first iteration of some value.

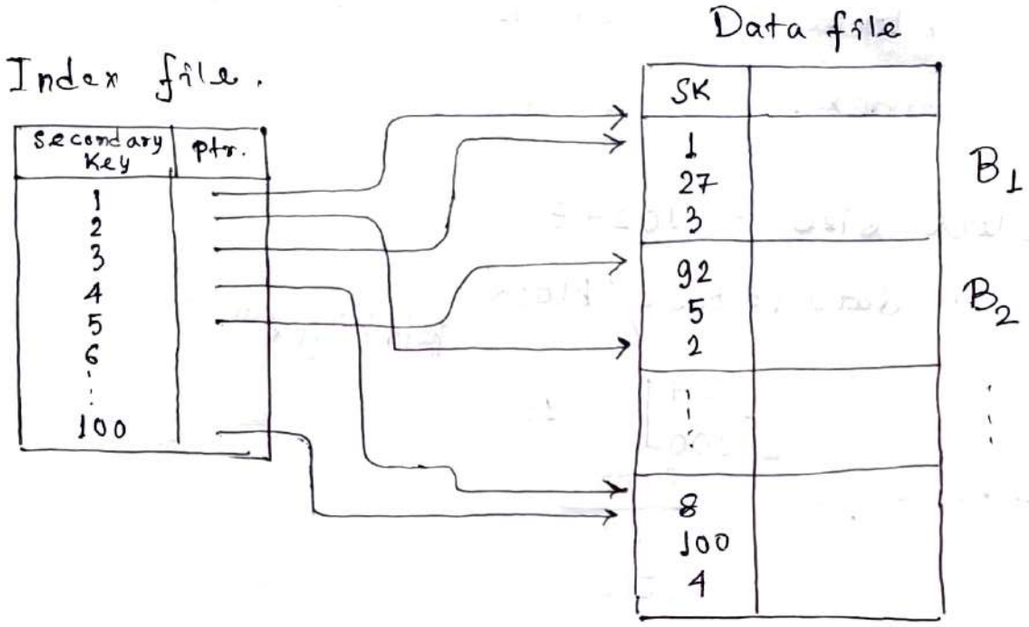
* Secondary Index. (Data file unsorted)

Provides a secondary means of accessing a file for which primary access already exists.

Index is created for every record in a file.

It is an example of dense indexing.

No. of index entries = No. of records.



Data file can be sorted wrt some attribute. But, wrt some other attribute data file is unsorted.

- Called secondary because normally one indexing is already done (wrt some attribute). To get record on some other attribute, we use secondary indexing.

- Index entries required for each record (as they're unordered).

- No. of block accesses reqd. =

$$\lceil \log_2 n \rceil + 1.$$

- Generally applied on data file that is sorted wrt some attribute, but we want indexing wrt some non-key.

- For duplicate values, we can put all the references of some value inside the block pointer an index entry.

Q. We have an ordered file of 30000 records on a disk with a block size 1024 B. Records are of fixed length & are unspanned of size 100 B. Suppose, we have created secondary index on the key field of the file of size 9 B & a block ptr of size 6 B. Find avg no. of blocks to search a record with or without index.

→ Block size = 1024 B

No. of data records / block

Blocking factor.

$$= \left\lfloor \frac{1024}{100} \right\rfloor = 10$$

Blocks reqd. = $\frac{30000}{10}$

$= 3000$

Block accesses without indexing =

best case $\left(\frac{1 + 3000}{2}\right) \cong 1500$ worst case

| linear search

Size of index records = $9 + 6 = 15$ B

No. of index records = No. of data records

$= 30000$

No. of index records / block = $\left\lfloor \frac{1024}{15} \right\rfloor = 68$

No. of blocks reqd. by the index file =

$$\left\lceil \frac{30000}{68} \right\rceil = 441 + 1$$

No. of block accesses = $\lceil \log_2 442 \rceil + 1$

$= 10.$

* B - Trees and B+ Trees.

- Generalization of - Multilevel Indexing.
  Multilevel indexes with little modifications.
- Node pointer / Block pointer
- Record pointer : Points to the record we are searching for.

• B - Trees.

In B - Trees at every level we are going to have key & data pointer & that data pointer actually points to either block or record.

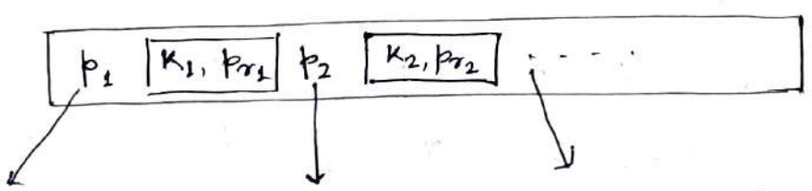
- Properties.

root - a root can have children between 2 and p, (p is called order of tree).

internal nodes -

- internal node can have children between $\lceil p/2 \rceil$ and p.
- can have keys between $\lceil p/2 \rceil - 1$ and $p-1$
- internal nodes arranged as

$$\langle P_1 \langle K_1, Pr_1 \rangle P_2 \langle K_2, Pr_2 \rangle \cdots \langle K_{p-1}, Pr_{p-1} \rangle, P_p \rangle$$

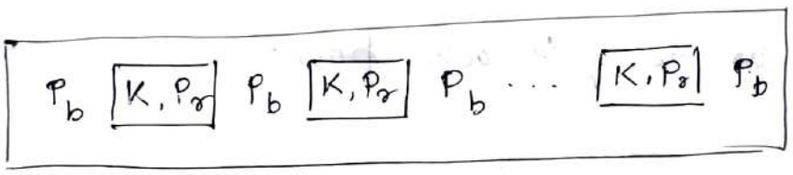


internal node.

Leaf node - can have keys between -
$\lceil p/2 \rceil - 1$ and $p-1$

Q. B-tree with key size 10 B, block size 512 B, data ptr is 8 B & block pointer is 5 B. What is order of B-tree ?

→



P_b – node ptr
P_r – record ptr

$$n P_b + (n-1)(K + P_r) \leq 512$$

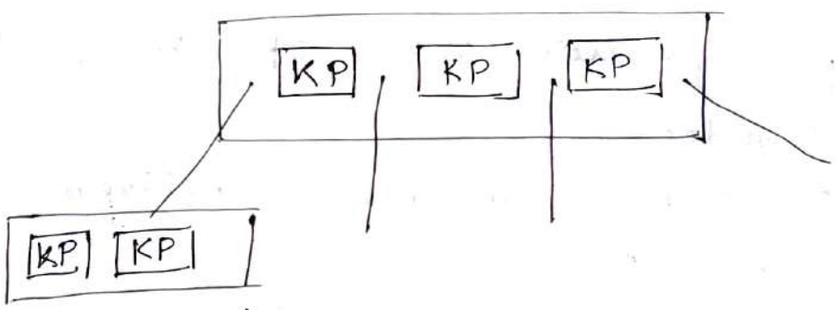
n order (say)

$$\Rightarrow n \times 5 + (n-1)(10 + 8) \leq 512$$

$$\Rightarrow n \leq \frac{530}{23} \quad \therefore n = 23 \text{ order}$$

Q. Order of a B-tree is 23. How many ~~may~~ index records can be stored in 4 levels (including root as 1-level) across the B-tree?

→



index records

① 22

② 23×22

③ $23 \times 23 \times 22$

④ $23 \times 23 \times 23 \times 22$

– Underflow & overflow in B-tree

B-tree of order p, if no. of search key values in a B-tree exceeds $p-1$, then it's overflow.

When node has less than what is actually required.

↳ no. of search key values $< \lceil p/2 \rceil - 1$

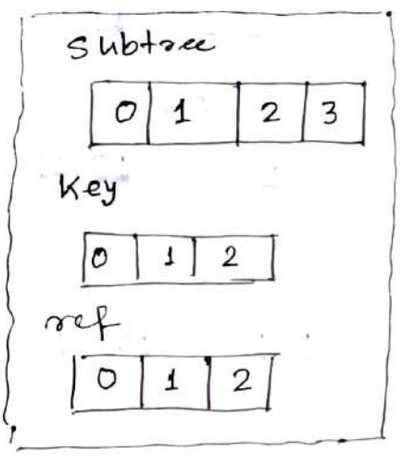
— Searching in B-tree.

Similare to BST, however rather than moving left or right at each node we need to perform a p-way search to see which subtree to probe.

target key, trace node to search, order

```
Search_btree (tk, node, p) {
int s = 0
while s < p-1
    if tk == node.key[s] then
        return node.ref[s];
    else if tk < node.key[s] then
        break;
    else
        s++
    end
end
if node.subtree[s] ≠ null then
    return search_btree (tk, node.subtree[s], p);
else return -1;
end }
```

Node data structure.



Time complexity $\log_p n$

n - no. of search keys

p - order of B-tree.

— Insertion in B-tree

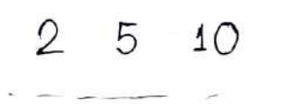
1. Search to determine which leaf node will hold a key.
2. If leaf node has space, insert key in ascending order.
3. Otherwise split leafnode's keys into 2 parts & promote median key to the parent.
4. If the parent node is full, recursively split & promote median key to its parent.
5. If a promotion is made to a full root node, split & create a new root node holding only the promoted median key.

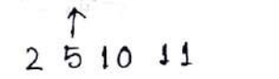
eg. Insert keys in B. Tree of order 4.

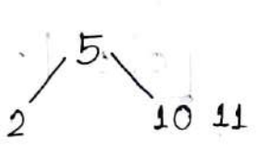
2, 5, 10, 11, 1, 6, 9, 4, 3, 12, 18, 20, 25

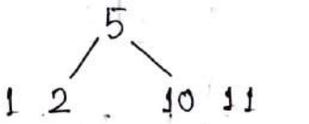
→ Root (1, 3) - keys (min, max)

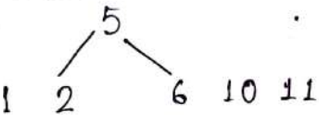
Non-root (1, 3) - keys (min, max)

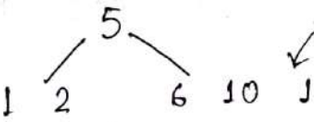




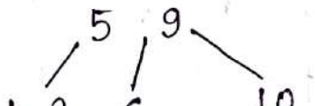


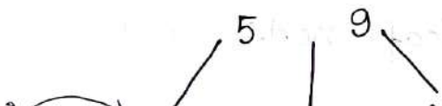




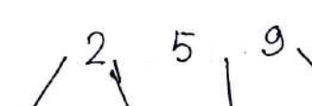


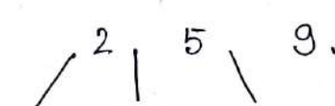
(9 between 6 & 10 then take middle element)

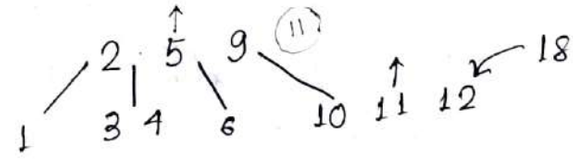




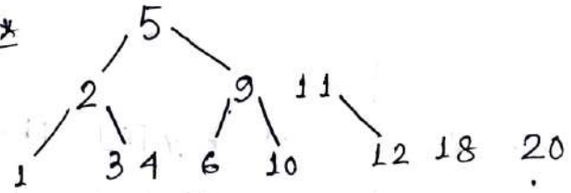
**

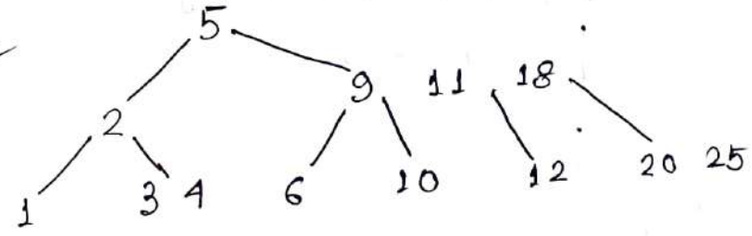






***

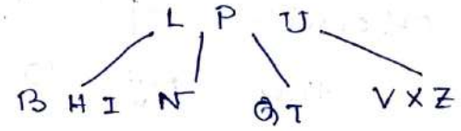




If a B-tree has n levels, inserting a new key may create maximum n+1 new nodes.

Q. Consider following 2-3-4 tree (B Tree '03 with min degree 2) in which each data item is a letter. The usual alphabetical ordering of letters is used in constructing the tree -

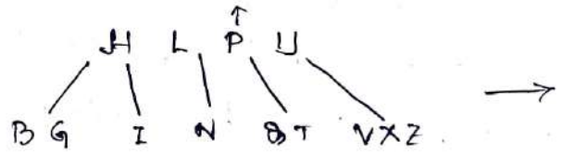
Max 3 keys



What is the result of inserting G?

→ B G H I

middle element (based on n/2 / n/2+1 choice)



— Deletion from a B Tree.

1. If the key to be deleted is not in a leaf, swap it with successor (or predecessor) under the natural order of the keys. Then delete key from the leaf.

2. If leaf contains more than the minimum number of keys then one can be deleted with no further action.

3. Otherwise, if the node contains the minimum no. of keys consider the 2 immediate siblings of the node.

4. If one of the siblings has more than the minimum number of keys, then redistribute one key from this sibling to the parent node & one key from the parent to the deficient node.

5. If both immediate siblings have exactly the minimum no. of keys, then merge deficient node with one of the sibling node, & one entry from the parent node.

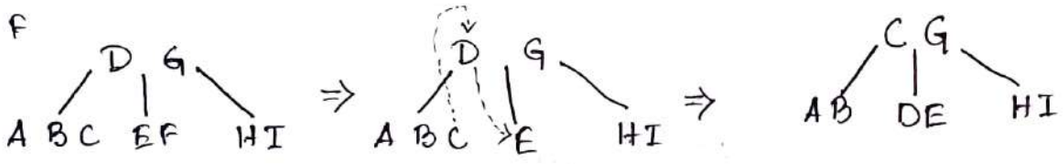
6. If this leaves the parent node with too few keys, then the process is propagated upward.

• Removing a key from a leaf node leaving l keys in the leaf node.—

if $l \geq \lceil p/2 \rceil - 1$ the we can stop. If $l < \lceil p/2 \rceil - 1$, we rebalance tree to correct this.

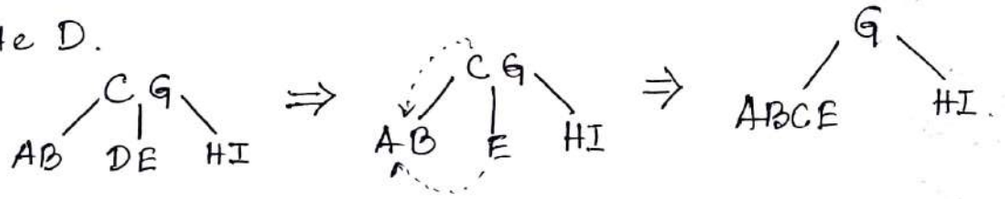
Borrow: Order 5 B-tree. $\lceil p/2 \rceil - 1 = 2$

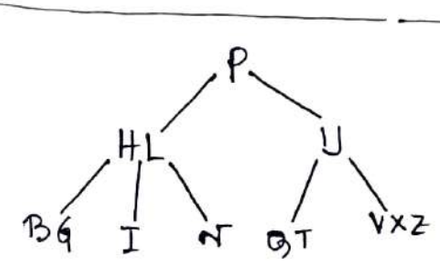
Delete F



Coalesce:

Delete D.





B-tree.

→ Balanced p-way tree (order p)

→ Generalisation of BST in which a node can have more than one key & more than 2 children.

→ Maintains sorted data.

→ All leaf nodes must be at same level.

→ Order p has following properties ~

- Every node has max p children.
- Min children ~
  - leaf → 0
  - root → 2
  - internal nodes → $\lceil p/2 \rceil$
- Every node has max (p-1) keys
- Min keys ~
  - root → 1
  - all other nodes → $\lceil p/2 \rceil - 1$

• Deletion in B-tree.

Case 1 If target key is in leaf node.

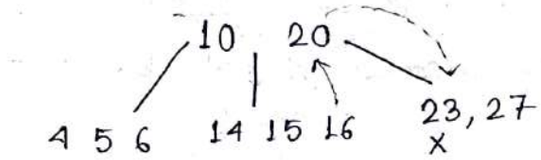
1a. Leaf node contains more than min no. of keys $\left[> \lceil p/2 \rceil - 1 \right]$

⇒ Simply delete the key & then stop.

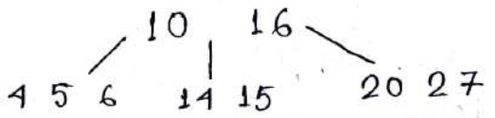
1b Leaf node contains ≤ min. no of keys $\left[\leq \lceil p/2 \rceil - 1 \right]$

1b.i. Borrow from immediate left/r sibling

| p = 5
| min keys = 2

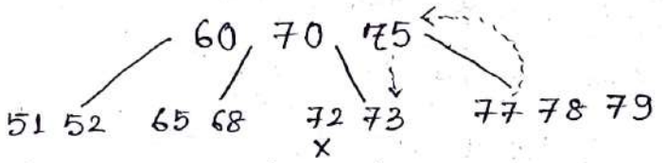


Delete 23. ⇓

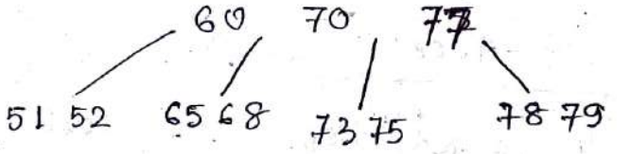


1b.ii Borrow from immediate right sibling.

| p = 5

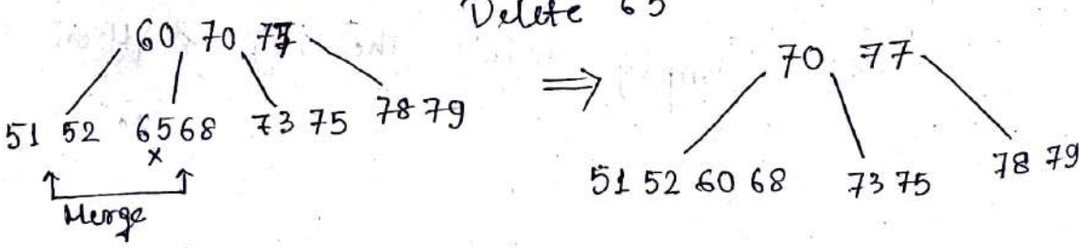


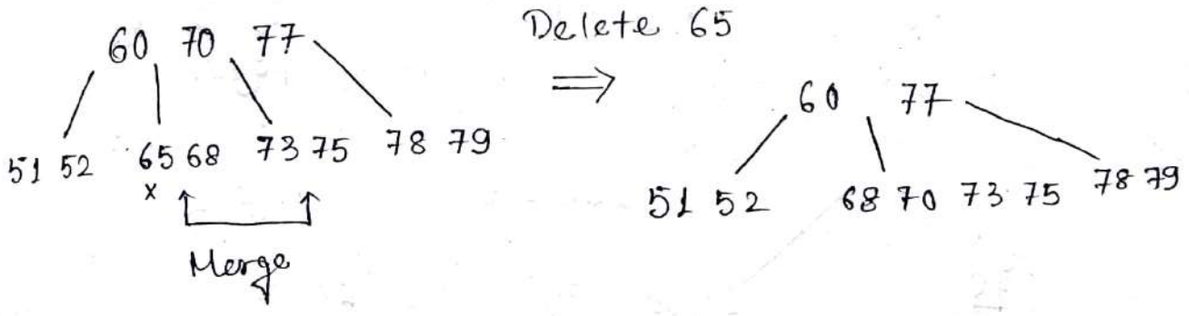
Delete 72 ⇓



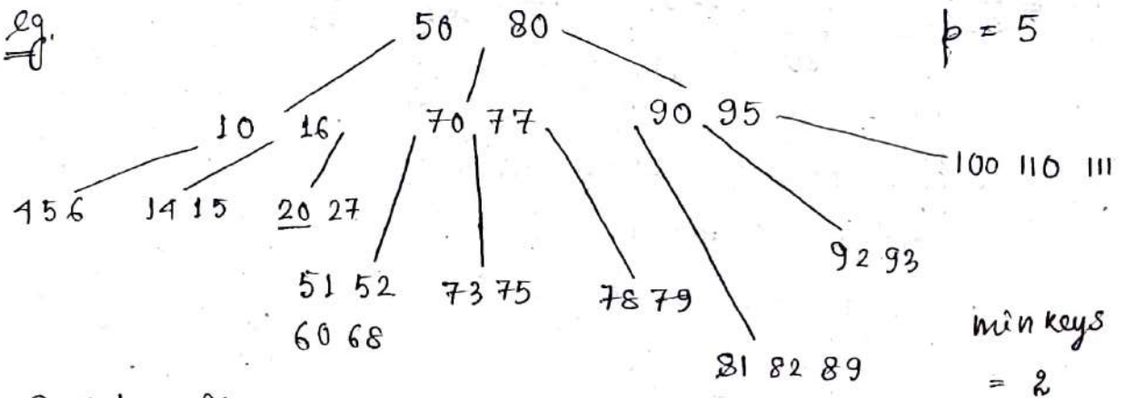
1b.iii Neither right or left sibling has more than min. no of keys. Then merge with l/r sibling. (Coalesce)

Delete 65

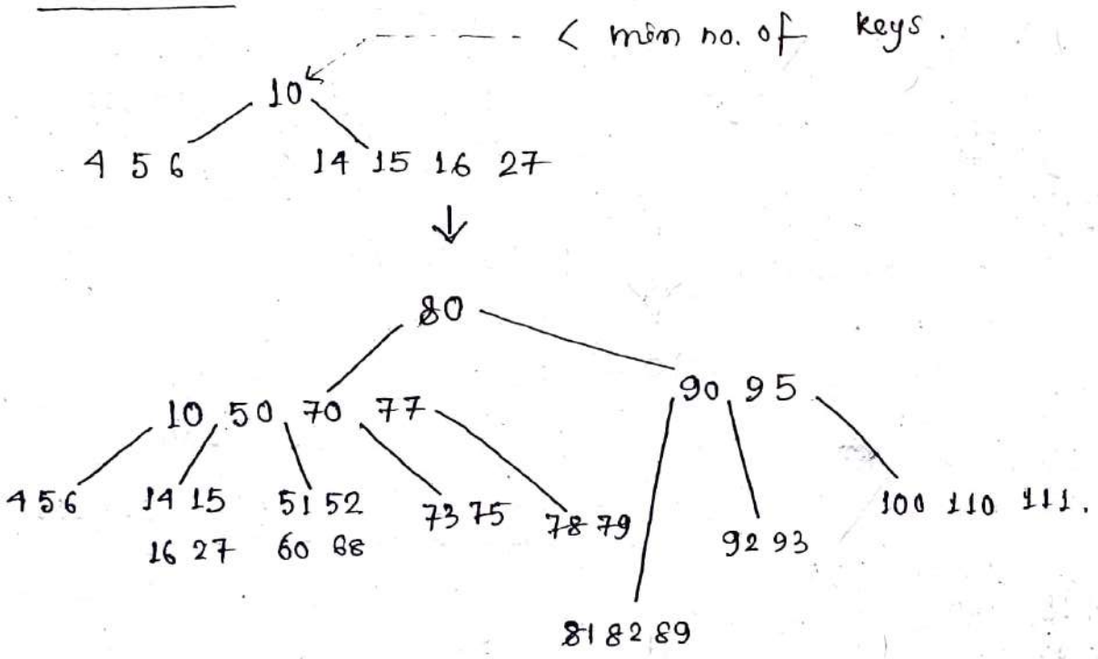




Eg.



Delete 20.

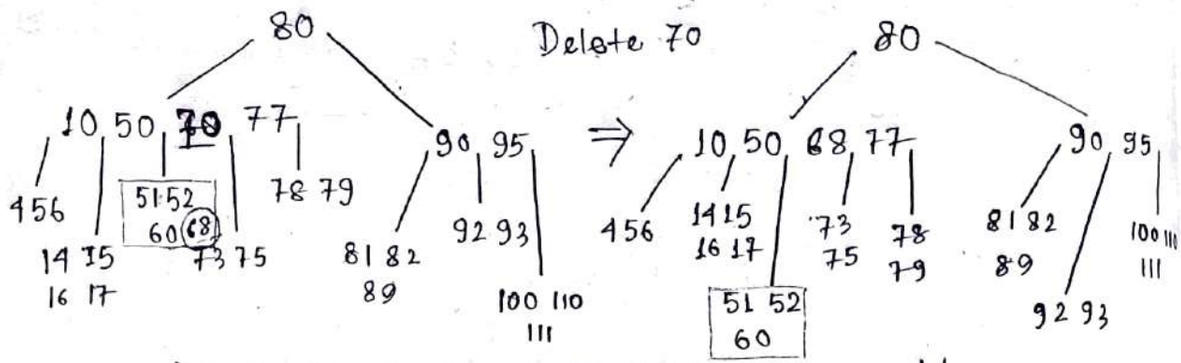


Case 2. If target key is an internal node.

Internal node can have

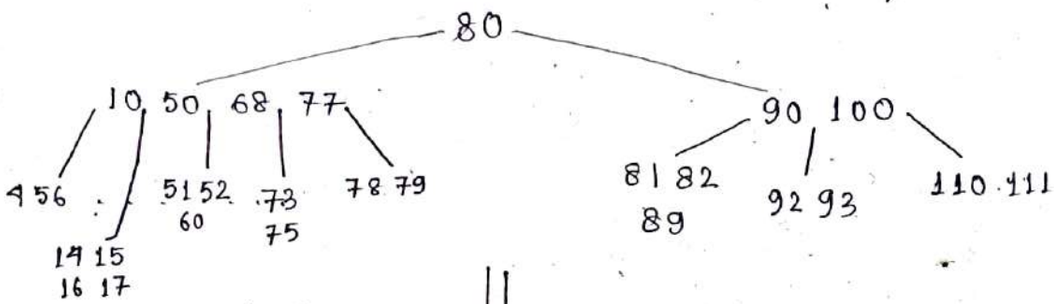
- Inorder predecessor
- Inorder successor.

• Has inorder predecessor and container node has > min. no. of keys.



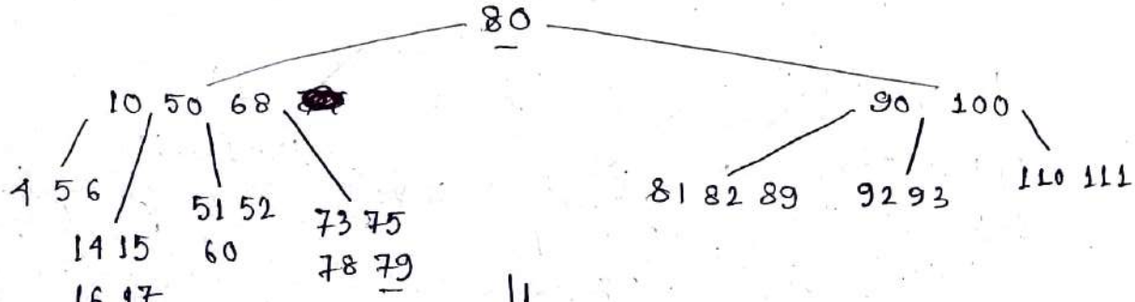
• Has inorder successor of container node has > min no. of keys.

Delete 95



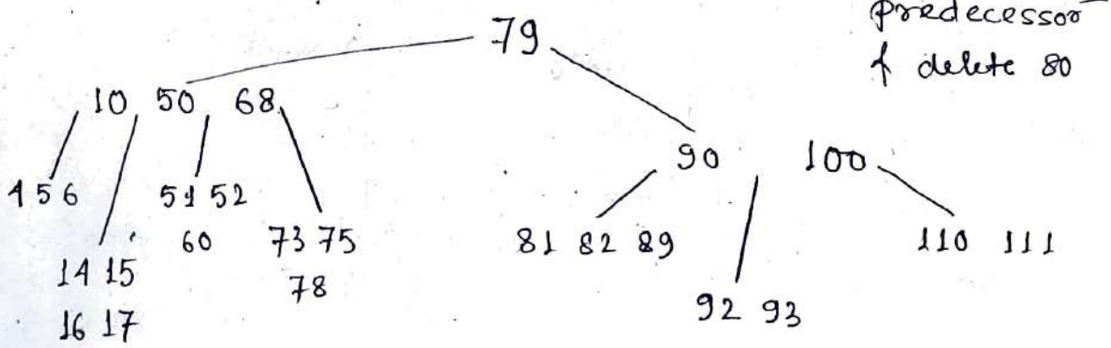
• Coalesce

Delete 77

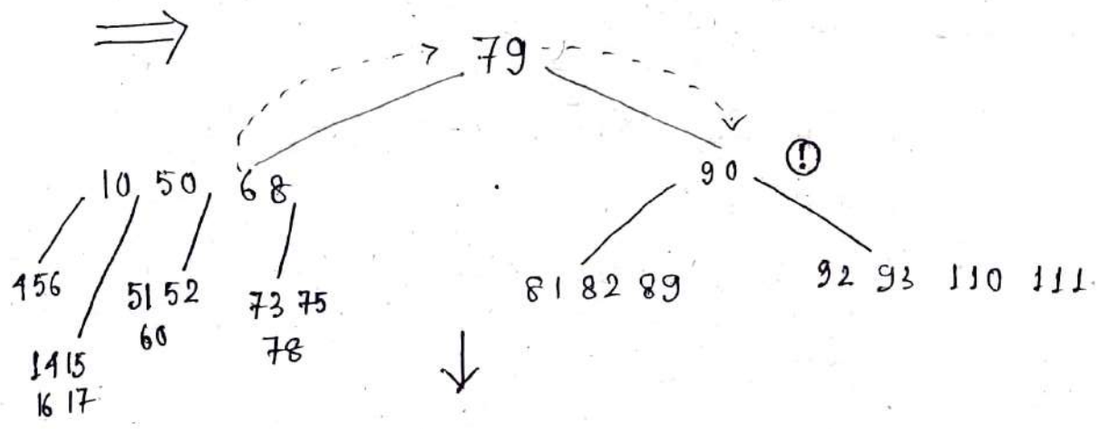


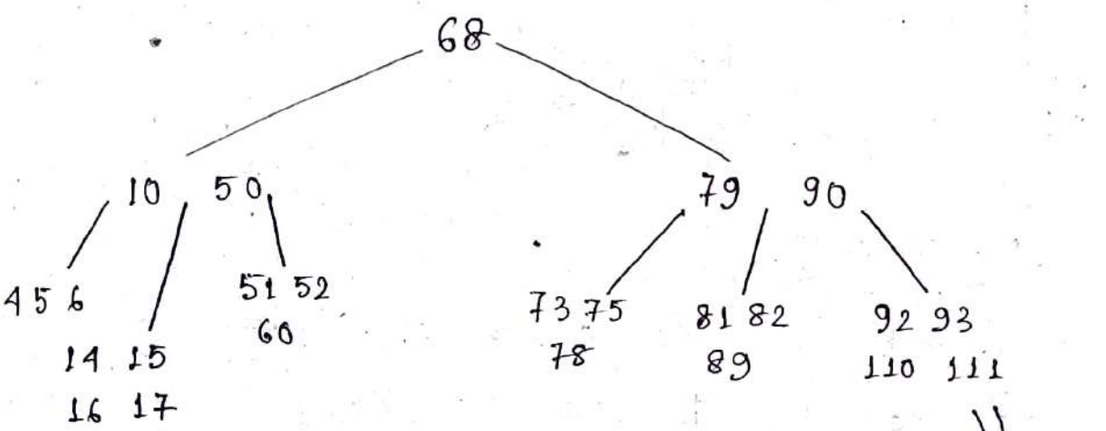
eg

Delete 80. [Swap with inorder predecessor & delete 80]

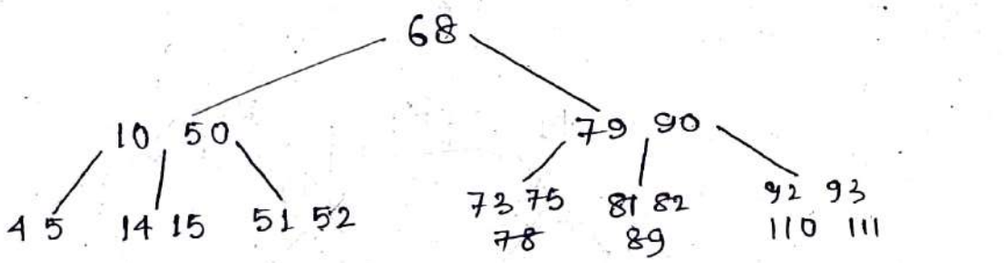


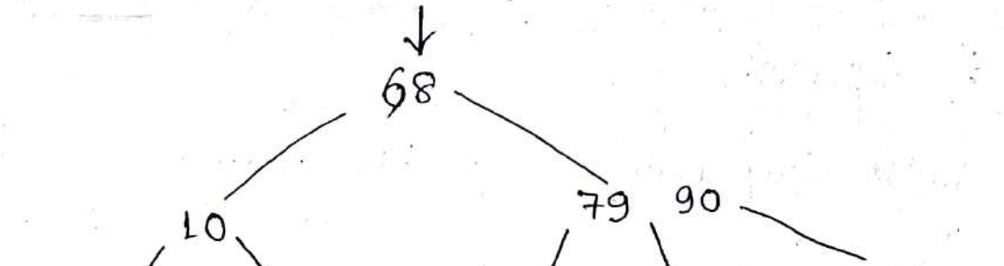
eg
Delete 100 (Coalesce)



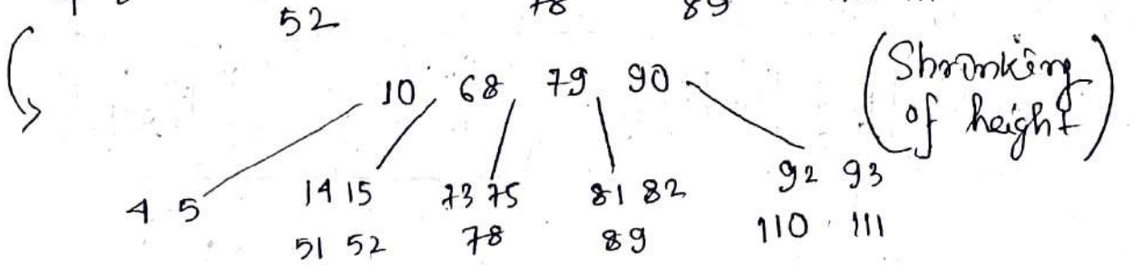


eg. Delete 6, 17, 60, 16 and then 50.





*



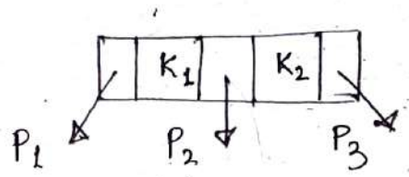
* B^+-Tree

Order of internal node is p.

⇒ Internal node has $\lceil p/2 \rceil$ to p children

⇒ Internal node has $\lceil p/2 \rceil - 1$ to $p-1$ search key values.

$p = 3$

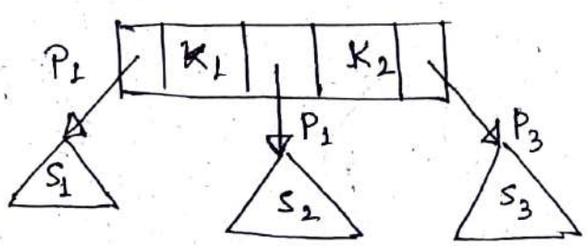


- Non-leaf node.

Each key-search value in subtree S pointed by P_i, $K_i >= K_{i-1}$

Key values in $S_1 < K_1$

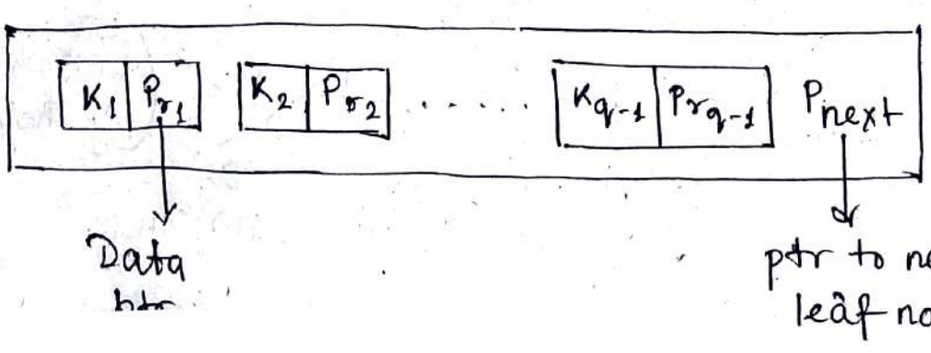
$K_1 \leq$ key values in $S_2 < K_2$



- Leaf node.

Each leaf node is of the form-

$\langle \langle K_1, P_{r_1} \rangle, \langle K_2, P_{r_2} \rangle, \ldots, \langle K_{q-1}, P_{r_{q-1}} \rangle, P_{next} \rangle$



eg. Search key field is $V = 9B$, the block size $B = 512B$; record ptr $P_r = 7B$, block ptr $P = 6B$. What is the order of internal node & leaf node?

→ $\boxed{P_{b_1} K_1 P_{b_2} K_2 P_{b_3} \cdots K_{n-1} P_n}$ Non leaf

$$n(P_b) + (n-1)K \leq 512$$

$$\Rightarrow n \times 6 + (n-1) \cdot 9 \leq 512$$

$$\Rightarrow n \leq \frac{512}{15} \qquad n = 34.$$

Max no. of nodes ptr that the non-leaf can have.

Order of non-leaf 34.

$\boxed{(K_1, P_{r_1})(K_2, P_{r_2}) \cdots (K_m P_{r_m}) P_{next}}$ Leaf

Order here is max no. of record ptrs it can have.

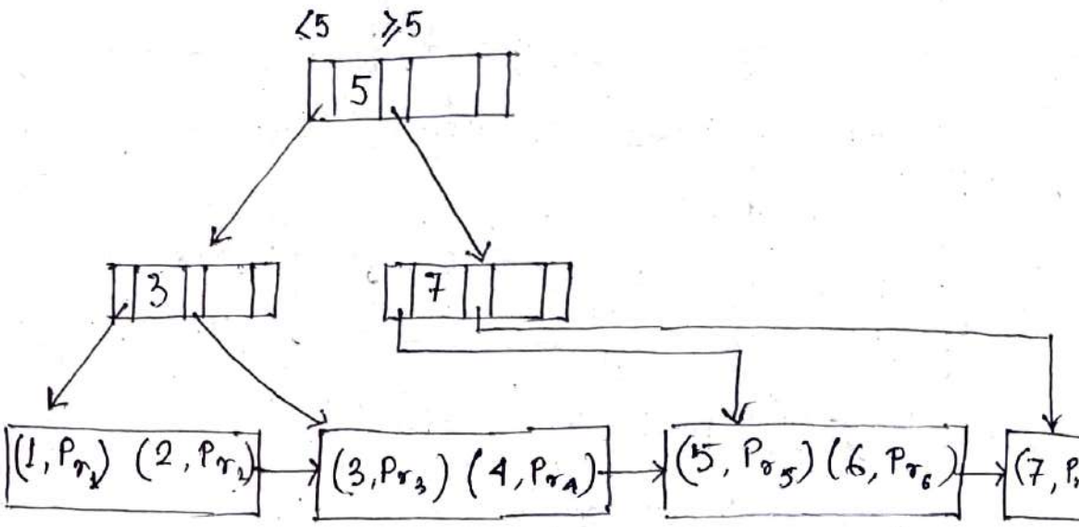
$$m(K + P_r) + P_b \leq 512$$

$$\Rightarrow m(9+7) + 6B \leq 512$$

$$\Rightarrow m \leq \frac{506}{16} \qquad m = 31.$$

Order of leaf node 31.

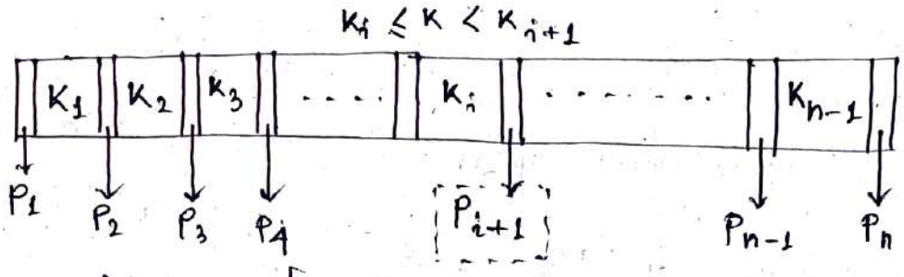
eg. B+ Tree with $P = 3$ and $P_{leaf} = 2$.



- Searching a key value K

- Start from the root, look for the largest key value (K_i) in the node $\leq K$.

- Follow the pointer P_{i+1} to next value until reach the leaf node



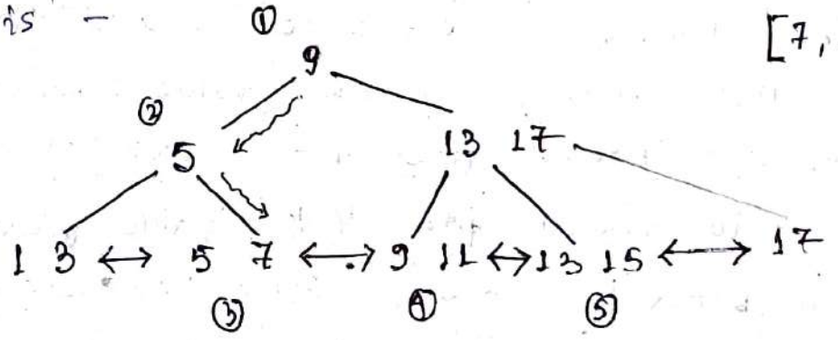
- If K is found to be equal to K_i in the leaf, follow P_{r_i} to search record.

~ Assuming order to be p, (order of non-leaf), $\log_p n$

- In B+ tree, leaf level is a sorted linked list of index entries.

G'15 With reference to the B+ tree index of order 3 shown, the minimum number of nodes (including root node) that must be fetched in order to satisfy the following query - 'Get all records with search key greater than or equal to 7 & less than 15 is' -

[7, 15)



Ans - 5.

G'15 Consider a B+-tree in which the search key is 12B long, block size 1024B, record ptr 10B, block ptr 8B. Max. no of keys that can be accomodated in each non-leaf node of the tree is - (50)

$$n(K) + (n+1)(P) \leq 1024$$

$$\Rightarrow n(12) + (n+1)(8) \leq 1024 \Rightarrow n \leq 50...$$

Q. G'16 B+ Trees are considered balanced because-

✓ a) The lengths of the paths from the root to all leaf nodes are all equal.

b) The lengths of the paths from the root to all leaf nodes differ from each other by at most 1.

c) The no of children of any 2 non-leaf siblings differ by at most 1.

d) The no. of records in any 2 leaf nodes differ by at most 1.

Q G'07 The order of a leaf node in a B+ tree is the maximum no. of (value, record ptr) pair it can hold. Given that block size is 1KB, data record ptr 7 B, value field 9B & a block ptr 6 B. What is the order of leaf node? (63)

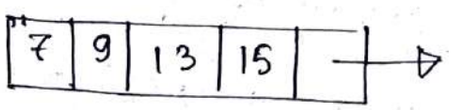
$$n(9+7) + 6 \leq 1024$$

$$\Rightarrow n \leq 63 \ldots$$

- Insertion in B+ tree.

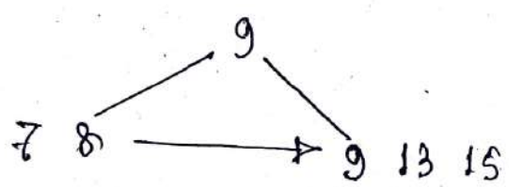
Order of B+ Tree.

leaf node only



Insert 8..

7 8 9 13 15.



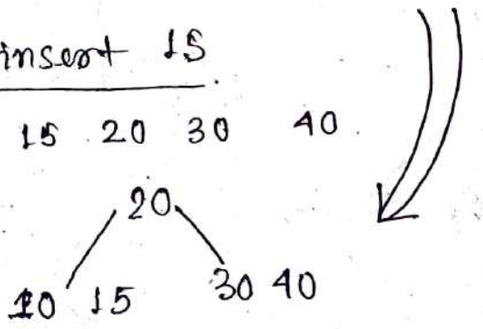
Overflow. When no. of search key values exceed p-1.

Leaf node. Split into 2 nodes.

- 1st node contains $\lceil \frac{p-1}{2} \rceil$ values
- 2nd node contains remainings
- copy the smallest search key value ~~10~~ of the 2nd node to the parent.

Parent non leaf.

Order 5. 10 20 30 40



insert 15

10 15 20 30 40

20

10 15 30 40

non-leaf node.

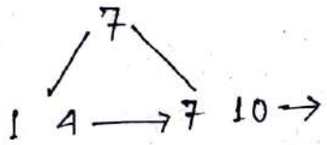
Split into 2 nodes.

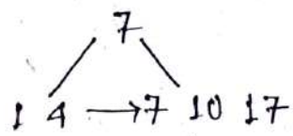
- 1st node contains $\lceil n/2 \rceil - 1$ keys
- Move the smallest of the remaining keys to the parent
- 2nd node contains remaining keys.

eg. Construct B+ tree (1, 4, 7, 10, 17, 21, 31, 25, 19, 20, 28, 42). p = 4. order of leaf & non-leaf 4.

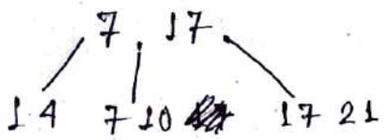
→ leaf ↘

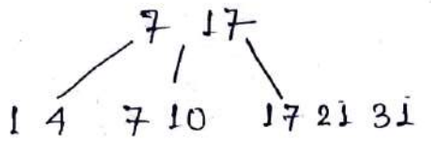
1 4 7 10.



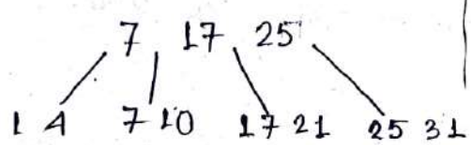


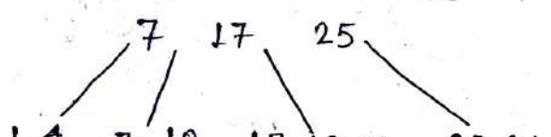
7 10 17 21.



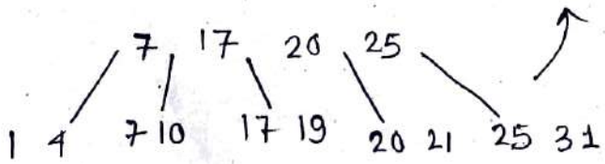


17 21 25 31.

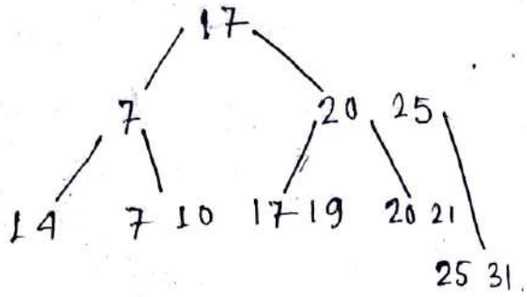


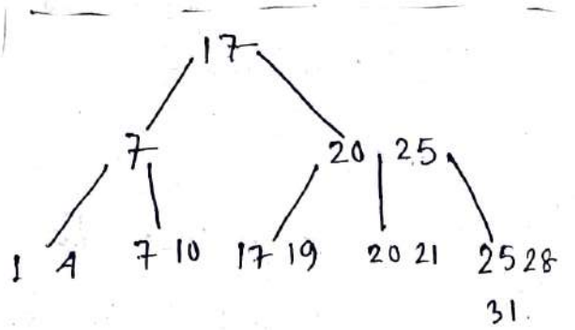


17 19 20 21

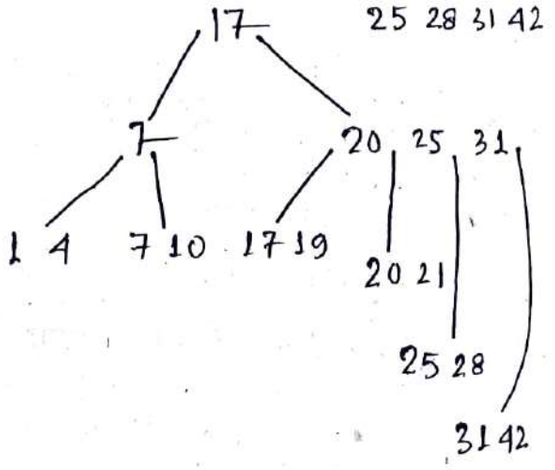


$\lceil p/2 \rceil - 1 = 1$





25 28 31 42



• Why B/B+ trees preferred over BST like (AVL, Red Black T) for storing index records & accessing them?

→ In Binary trees every node can have only 2 children therefore they need to grow in height.

Because of this, even database is small, the no. of levels we might have to access before we get to the final level is actually large.

• Deletion in B+ tree.

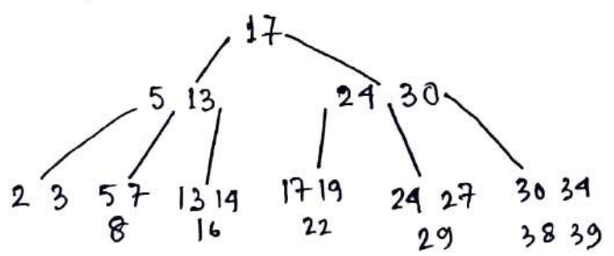
- Start at root, find leaf L where entry belongs.
- Remove the entry
  - ⇒ If L contains at least $\lceil P/2 \rceil - 1$ entries, done!
  - ⇒ If L has only $\lceil P/2 \rceil - 2$ entries,
    - ~ Try to redistribute, borrowing from sibling.
    - ~ If redistribution fails, merge L & a sibling.

If merge occurred, then corresponding entry from parent must be deleted.

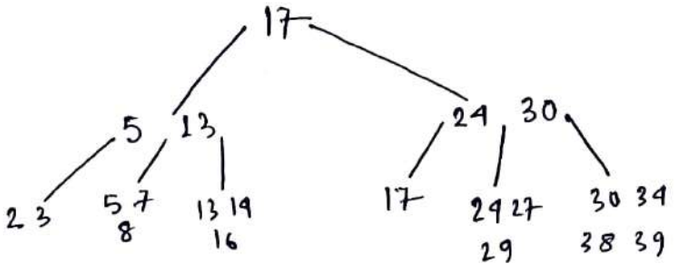
Merge could propagate to root decreasing height.

If the deleted entry present in the internal node, replace it with inorder successor.

eg. B+tree of order 5.

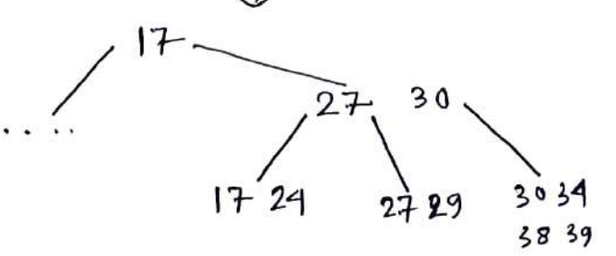


⇒

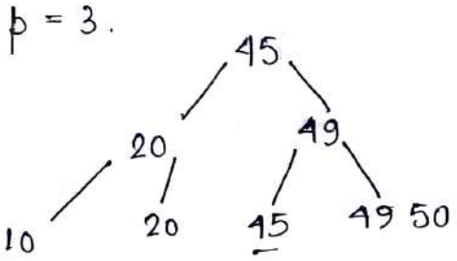


Delete 19, 29

⇓

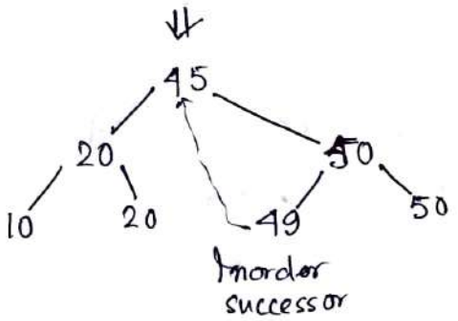


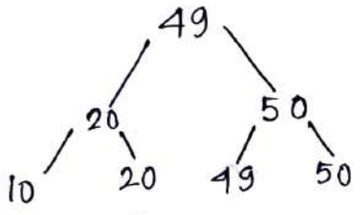
eg. $p = 3$.



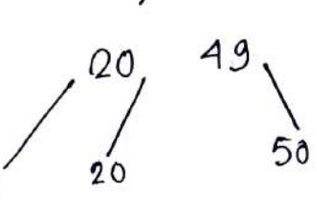
Delete 45

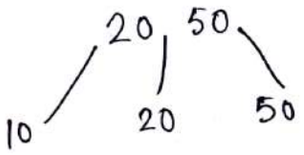
$\lceil p/2 \rceil - 1$ = 1 key at least





eg. Delete 49.





**