DATABASE MANAGEMENT SYSTEM COURSE BY

GO CLASSES

Lectures by Deepak Poonia

References :

- Fundamentals of Databases systems by Navathe
- Database System Concepts by Henry F. Korth 7th edition

1. The Relational Model and Normalization

//Lecture 1

1.1) Introduction to DBMS and Relational Model :

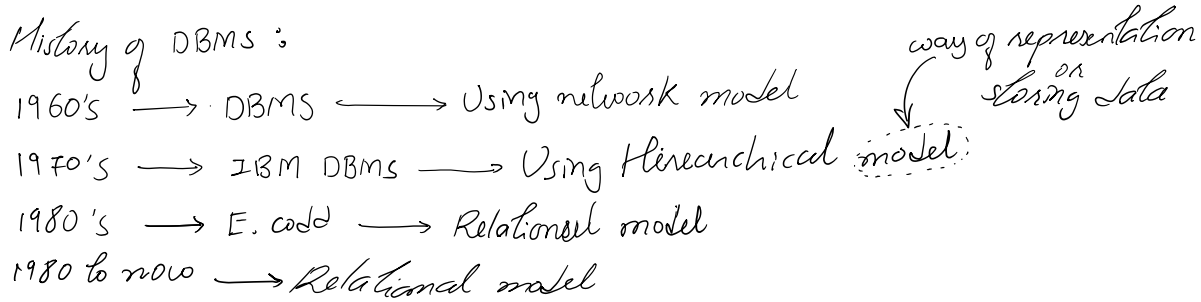
Data : can be stored, recorded.

Database : A collection of related data representing facts that have meaning in the real world.

Database management system : it is a software package designed to store and manage databases.

Database system (DBS) = Database + DBMS

It is computerized system; whose overall purpose is to maintain the information and to make that information available on demand.



Q : Do you use DBMS ? – No, in computer, OS provides you a file system. So, DBMS are required by large organizations. File system has main disadvantage of redundancy. We will study more in upcoming chapter.

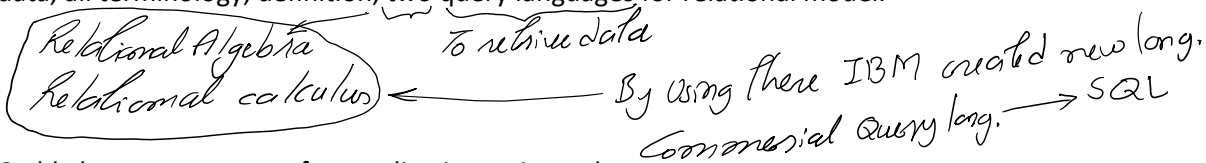
DBS design process :

1) *Requirement analysis phase* : gather information
2) *Conceptual design phase* : representing gathered info in user-friendly model (i.e. ER model)
3) *Logical design phase* : ER model is diagram based so we convert it to logical design (implementation friendly model also called relational model)
4) *Implementation* : Finally, we implement.

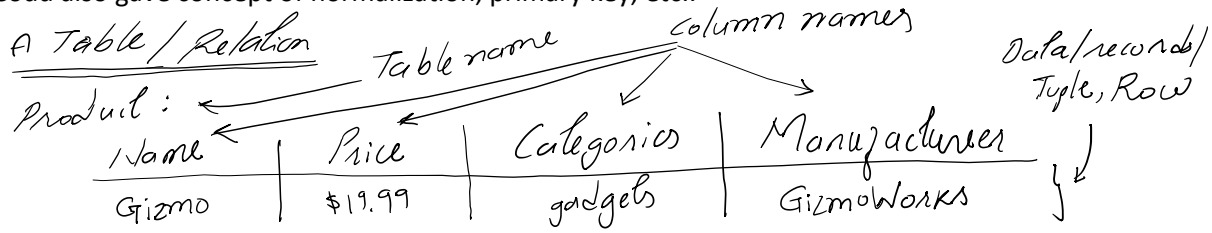
Data model : set of rules, conventions to represent/store/manage the data. For example, relational mode, ER model.

1.1.1) The relational model :

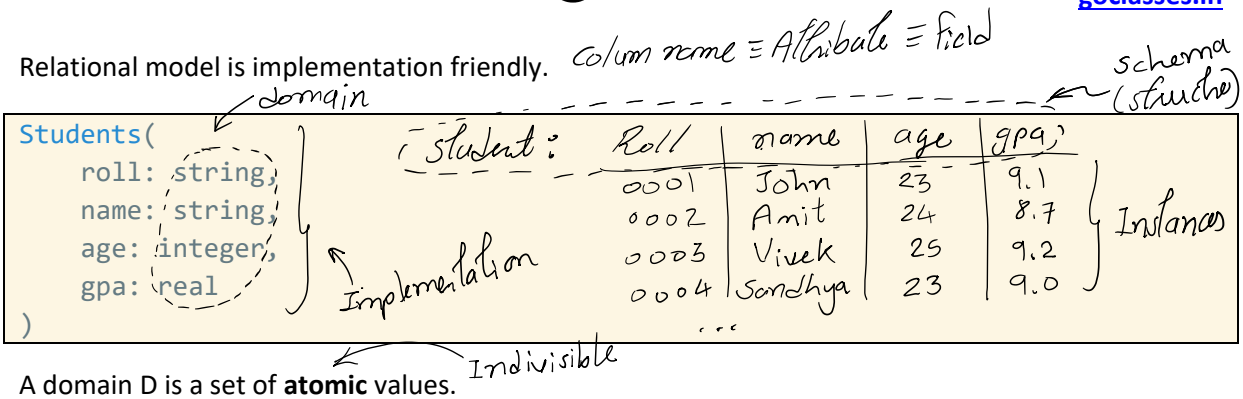
It is table/relation-based model proposed by E.F. Codd in a research paper. It covers how to store the data, all terminology, definition, two query languages for relational model.



Codd also gave concept of normalization, primary key, etc..



Relational model is implementation friendly.



A domain D is a set of **atomic** values.

Atomic means say if our row contains some string "ravi kumar" then we cannot get "kumar" out of this string because it is defined to be atomic i.e. indivisible.

Degree (or arity) of a relation : the number of attributes n of its relation schema.

Cardinality : number of rows in any given instance of a relation.

Q : Why this model is called "relational" ? – this is same as relation we saw in discrete math. Each attribute can be considered as a set. For example, previously

Relation (Student) $R \subseteq$ string $\times$ string $\times$ Integer $\times$ Real

→ every record → ordered n-tuple

In general, **n-ary relation** defines as

$$R \subseteq A_1 \times A_2 \times \ldots \times A_n$$

so, column order matters

$$A_1 \times A_2 \times \ldots \times A_n = \{(a_1, a_2, \ldots, a_n) | a_j \in A_j\}$$

Attribute values are atomic (as explained earlier) have a known domain and can sometimes be "NULL" and meaning of "NULL" is depend on context.

1.1.2) Key and Candidate key :

Some attributes which is used to uniquely identify tuples is called **key**. For example, contact no. of student in school register.

Superkey : *A set of attributes (can be singleton also) which can identify any record uniquely*. In relational model, at least one superkey is definitely present because as two records can never be entirely same so at worst all attributes combined will work as superkey.

In relational model, set of all attributes is always a superkey

Candidate key (CK) : *It is a minimal superkey* meaning if $CK = \{A_1, A_2, \ldots, A_n\}$ then $\forall_i (CK - A_i)$ is not a superkey.

Primary key (PK) : *One of the candidate keys is chosen as (selected as) to be a primary key (which must be not NULL)*. This is chosen by database owner you can say. Other keys apart from primary key in candidate key is called alternate keys.

If PK is multiple attributes, then no attribute of PK can have NULL value

PK is one of the CKS which must be not NULL

Every relational model has exactly one primary key therefore, no tuple can be fully NULL.

Q : Given $R(a_1, a_2, a_3, \dots, a_n)$ then max no. of superkey ? – to get max no. of superkey each a_i must be candidate key. Therefore, $2^n - 1$ superkey at max.

Q : Given $R(a_1, a_2, \dots, a_n)$ then max no. of candidate key ? – All keys cannot be candidate key then it won't be maximum. For example, R(a, b, c, d) in which max number of CK would be {ab, ac, ad, bc, bd, cd} and not {a, b, c, d}. we know that candidate key is subset of superkey so we want to find subset which has highest number of keys. $C\left(n, \frac{n}{2}\right)$.

Q : Given $R(a_1, a_2, \dots, a_8)$ then max no. of candidate key if it is known that $a_1a_2a_3$ is CK. Other CKs we don't know ? – we know that max no. of candidate key with 8 attributes would be C(8, 4) (from previous question) but this count $a_1a_2a_3$. So, C(8, 4) – 5 + 1 = 66. Why 5? Because at max we are selecting 4 attributes and we know $a_1a_2a_3$ is already selected thus, from remaining we can get 5 and +1 because we also have to count $a_1a_2a_3$ also which we have never counted.

//Lecture 4

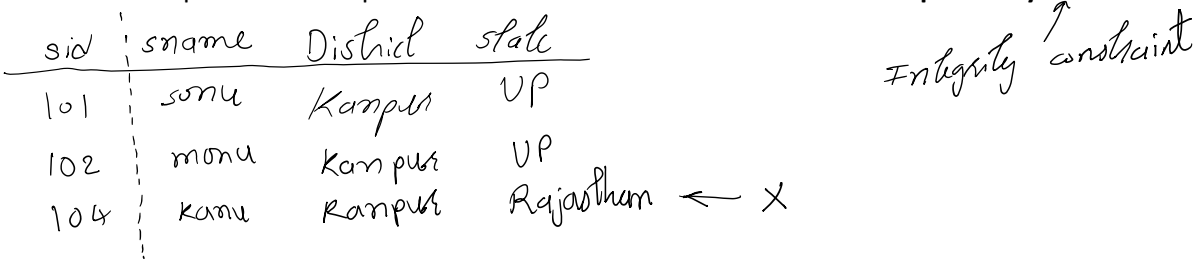
Note that relational model is theoretical/mathematical model and SQL is practical language loosely based on relational model.

Relational model considers table as a set of rows so no two tuples can be same. And SQL on the other hand considers table as a multiset of rows. We will explore more of this in SQL chapter.

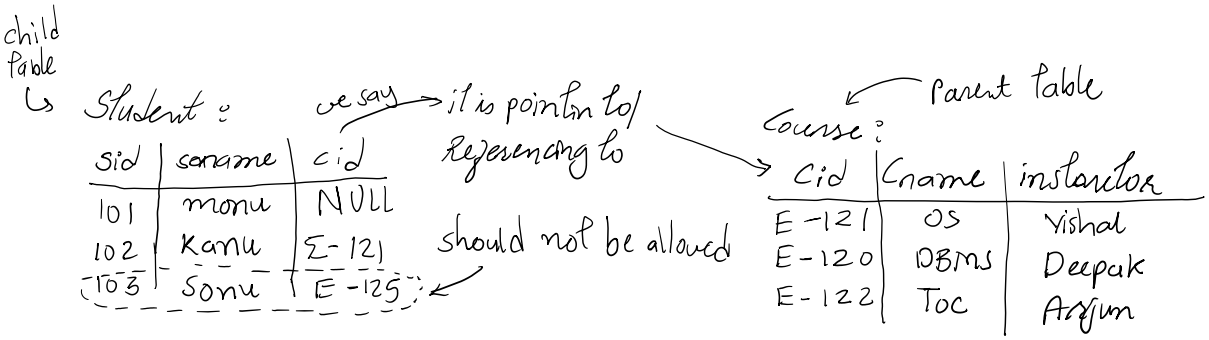
1.1.3) Integrity constraints in relational model : FF*KED

Let's say we want to create database for student studying in specific school. We should have some restriction/ constraints for example,

a) Sid (student id) must be integer : this type of restriction is often known as **domain integrity constraint**.
b) Sid (primary key) must be unique and never NULL : because if sid is NULL then we cannot distinguish two students. This is called **Entity integrity constraint** also known as **uniqueness constraint**.
c) Sid must be different for any two students : this type of restriction is known as **key integrity constraint**.
d) Consider one example, in this Kanpur must be the state of UP so we cannot allow Rajasthan with Kanpur. State is dependent on district. This is called **functional dependency IC**.



e) There should be some restriction on false information for example, consider following database. Here each student must take course provided by institute or don't take course then we will consider that student as registered but not enrolled (this is okay we don't have any problem) but we have problem with student selected course is not available in database. For example,



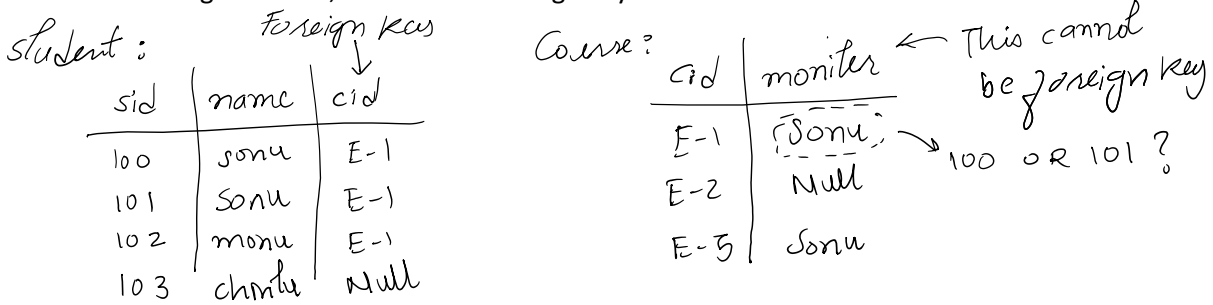
This is called **referential IC** or **foreign IC**.

In summary we can say,

IC name	Applies to whom ?	Applies where ?
Domain IC	For individual attributes	Within the same table
Entity IC	For primary key (not null, unique)	
Key IC	For candidate keys (Unique)	
FD-IC	Between two set of attributes	
Tuple Unique IC	Between tuple – Tuple should be unique	
Referential/ Foreign IC	Between two set of attributes of same domain	Between two tables OR within a table

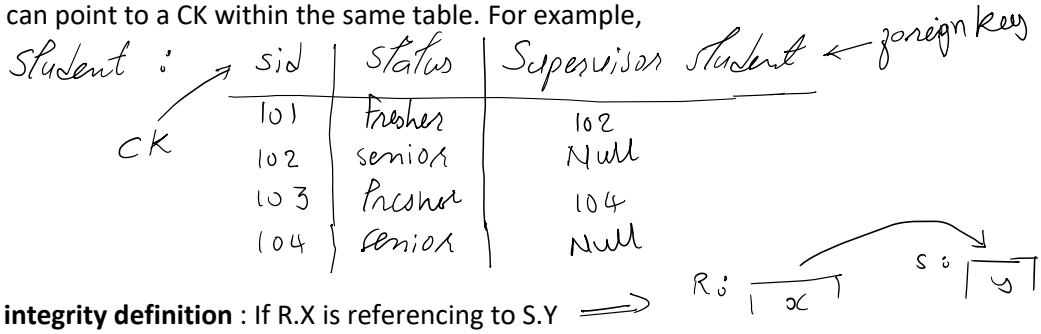
Foreign key always refers to candidate key

Consider following database, If Monitor is foreign key then Who is monitor of E1 ?



Therefore, <u>foreign need not be CK. FK may or may not be NULL, by default FK is allowed to be NULL. FK need not be a single attribute in other words FK can consists of more than one attribute also</u>.

Foreign key can point to a CK within the same table. For example,



Referential integrity definition : If R.X is referencing to S.Y ⟹

NOTE : Parent table = Referenced table = Containing CK

Child table = Referencing table = Containing FK = Referential table

Check assertion is a constraint used in SQL, it is used for limiting the value range for any column.

Then Referential IC says that R.X can either be NULL OR a value already presents in S.Y.

Q : When referential integrity violation happens; then what to do ? – In the *referential* table,

Deletion of row : No violation, insert a row : violation possible, modify a row : violation possible

Thus, if violation happens when we insert/modify a row then that action/operation is rejected/not allowed.

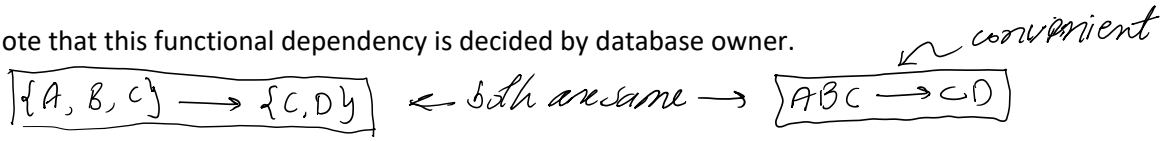
But in *referenced* table, insertion : no violation possible. But if Delete/modify : violation possible and if violation happens then 1) Cascade delete (meaning delete all the rows from referential table) 2) Cascade modify (meaning modify all the rows from referential table) 3) Put NULL (if we want that value to exists) 4) Ignore operation.

//Lecture 5

1.1.4) More about functional dependencies :

Definition : Let A, B be two attributes in a relation R then A → B iff for any two tuples t_1, t_2 such that $t_1.A = t_2.A$ then $t_1.B = t_2.B$.

Note that this functional dependency is decided by database owner.



Example,

A	B	C	D
a1	b1	c1	d1
a1	b2	c2	d2
a2	b2	c2	d3
a3	b3	c4	d3

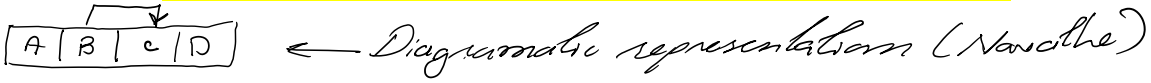
Here

$A \to B$ = Definitely does not hold

$A \to D$ = Definitely does not hold

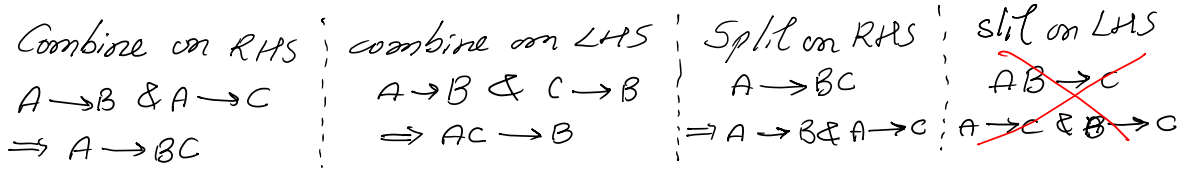
$B \to C$ = May or may not hold on R because it is true for this instant

Functional dependency defined only for schema and not for instances



1) **Transitive rule** : if A → B and B → C then A → C.

We can combine on LHS, RHS. We can split on RHS but we cannot split on LHS.



Trivial Functional dependency : $X \to Y$ is trivial FD iff $Y \subseteq X$. AB → A is trivial FD

All the FD apart from trivial is called non-trivial FD.

Now we will see Armstrong's axioms :

2) **Reflexivity** : $X \to Y$ for any $Y \subseteq X$.

3) **Augmentation** : If $X \rightarrow Y$, then $XZ \rightarrow YZ$.
4) **Pseudo-transitivity** : If $X \rightarrow Y$ and $WY \rightarrow Z$, then $WX \rightarrow Z$
5) **Composition :** If $X \rightarrow Y$ and $Z \rightarrow W$, then $XZ \rightarrow YW$.

Closure of a set of attributes : Let X : set of attributes then Closure of X = X^+ = {W | X → W}

X is closed iff X^+ = X

Q : R(ABC); AB → C; C → B which of the following is closed ? –

1) A $A^+ = \{A\}$ 3) C $C^+ = CB$ 5) AC $AC^+ = ABC$ 7) ABC $ABC^+ = \{ABC\}$
2) B $B^+ = \{B\}$ 4) AB $AB^+ = ABC$ 6) BC $BC^+ = \{BC\}$ 8) ϕ $\phi^+ = \{\phi\}$

NOTE : If every set of attributes is closed then there will be no non-trivial FD's thus always in BCNF.

We wish to test whether ad → b since b is a member of the closure of ad we conclude that **ad→b follows from the set of FD's**.

Q : Given FD set : {A→B, B→C} on R(A, B, C, D) then what will be CKS of R ? –

$A^+ = ABC$ but A cannot be ck because it cannot generate D
$B^+ = BC$; $C^+ = C$; $D^+ = D$ smallest possible set is AD^+ is CK

Any attribute y which is not present in any closure of other then y then y must be present in every key (Candidate key = key).

Cover : Let two FD set F, G then we can say F covers G iff $G^+ \subseteq F^+$. F is a minimal cover of G if F is the smallest set of functional dependencies that cover G.

And they are **equivalent** iff $F^+ = G^+$.

//lecture 6

Definition of non-trivial FD : $X \rightarrow Y$ is non-trivial iff $Y \not\subseteq X$ **OR** $X \rightarrow Y$ is non-trivial iff $X \cap Y = \emptyset$.

From now on we will consider only the nontrivial FDs with RHS being a single attribute.

Example, if $A \rightarrow BC$ is given then we will slit as $A \rightarrow B$, $A \rightarrow C$.

Types of functional dependencies : (partial and full)

- ***Full FD*** : Z is fully dependent on X and Y meaning $XY \rightarrow Z$ and $X \rightarrow Z$, $Y \rightarrow Z$ should not be allowed. Formally we can say that $\boldsymbol{X \rightarrow Y}$ **is full FD iff** $\forall_{a \subset X}\, \boldsymbol{a \nrightarrow Y}$.
- ***Partial FD*** : $X \rightarrow Y$ is partial FD iff $\exists_{a \subset X} a \rightarrow Y$. It means we must have more than 2 attributes for partial FD to exists because with two or less attributes we cannot create proper subset.

Q : consider relation R(A, B, C, D, E) FD set : $\{CD \rightarrow E, D \rightarrow A, A \rightarrow E\}$. Is $CD \rightarrow E$ a partial FD and is $BD \rightarrow C$ a partial or full FD ? – we check C+ = {C} and D+ = {D, A, E} we can see that D alone can determine E so $CD \rightarrow E$ is a partial FD. Talking about $BD \rightarrow C$ so this is not even valid FD so it can't be both.

General definition transitive FD : $X \rightarrow A$ is transitive FD if $\exists y$ $X \rightarrow Y \rightarrow A$ & $A \notin X$, $A \notin Y$

//Lecture 7

1.2) Normal forms :

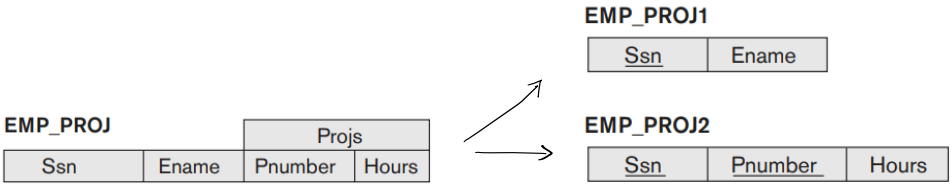
A relation (relation schema/scheme) can be in the following normal forms like 1NF, 2NF, 3NF and BCNF,...

1NF : Trivial goes Relational model
↳ Defined wrt Domain

2NF : } Defined wrt Non-trivial
3NF FD's
BCNF }

1.2.1) 1NF and 2NF :

Domain is **atomic** if its elements are considered to be indivisible units. *A relational schema R is in **first normal form** if the domains of all attributes of R are atomic.* Which means every relation is in 1NF.



*A relation is in 2NF if every non-prime attribute is fully dependent on every **candidate key OR** a relation is in 2NF if no non-prime attribute is partially dependent on some candidate key.*

NOTE : here in definition non-prime attribute is used meaning if some relation has no non-prime attribute then it is in 2NF.

Q : Which of the following relations is in 2NF ? –

a) R(ABCD) FD's : A → C; B → A; A → D; AD → C B is PK
Note that B → ACD means all non PA is fully dependent on PK B Non PA = ACD

b) R(ABCD FD's : C → D; CD → A; AB → C; BD → A
Hence CK = {BD, BA, BC} No non prime attribute

c) R(BCD) FD's: A → D; C → A; D → B; AC → B PK = C
same as option A so this is in 2NF

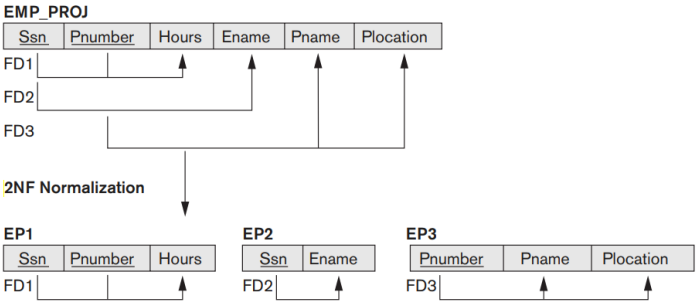
d) R(ABCD) FD's : BD → C; AB → D; AC → B; BD → A
CK = {AB, BD, AC} no non PA so it is 2NF

Which means violation of 2NF happens when...

Proper subset of some candidate key → some non-prime attributes

Most common mistake in checking whether some part of candidate key determines some non-prime attribute is only checking FDs in given FD set but we actually have to check FDs in FD closure.

Steps : 1) Find CKs and find non-prime attributes. 2) For every proper subset of CK, find closure and see if it contains non-prime attribute.



1.2.2) 3NF :

A relation R is in 3NF if no non-prime attribute is transitive dependent on some CK ***OR*** *a relation R is in 3NF if every non-prime attribute is not transitive dependent on every candidate key*. Meaning violation happens when

some CK —transitively→ Some non prime attribute

Say α is not a super key meaning it cannot determine any candidate key and say A is non-prime attribute. Therefore, if $\alpha \rightarrow A$ then $CK \rightarrow \alpha \rightarrow A$ which is the condition of transitive FD.

Which means another variation of violation can be $\boldsymbol{\alpha(which\ is\ not\ a\ superkey) \rightarrow A(non\ PA).}$

NOTE : In definition of 3NF non-prime attribute is mentioned so if no non-prime attribute is present then it is in 3NF.

//Lecture 8

Q : In 3NF; can we have transitive dependency ? – Surprisingly yes, because definition says no non-prime attribute is transitively dependent upon some candidate key but prime attribute can transitively dependent upon each other. For example, AB ⟷ CD and A → C. Here there is no non-prime attribute and $AB \rightarrow C$ & $AB \rightarrow A \rightarrow C$ Therefore, AB → C is transitive dependency.

Now,

Consider 2NF violation condition, It says

This means (Proper subset of some candidate key) → Some non-prime attribute

Not a superkey

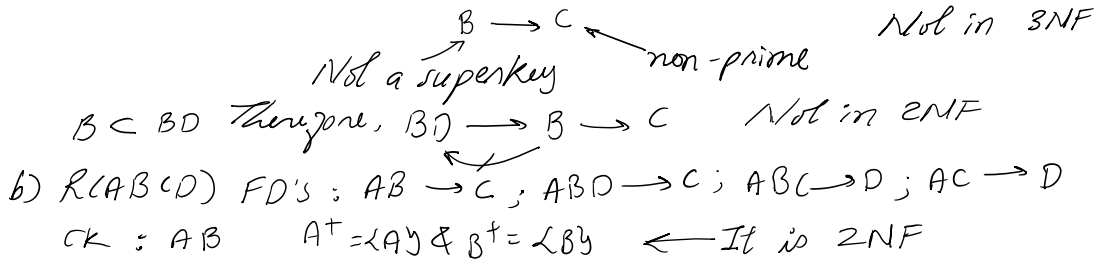
Therefore, Not a superkey → some non-prime attribute

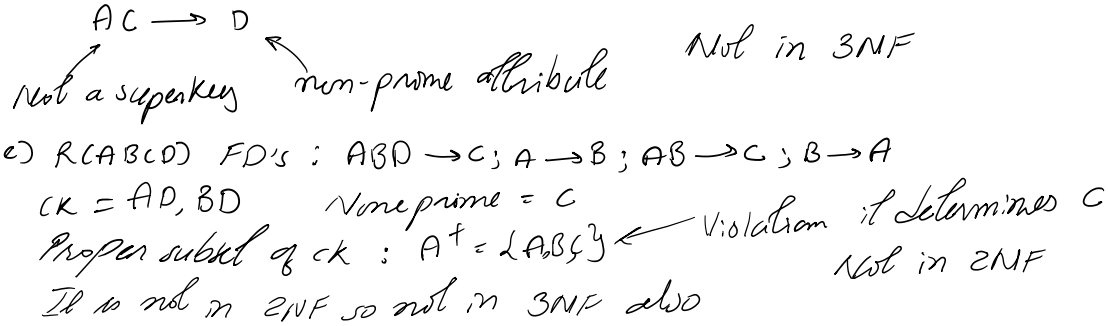
Walla, this is also violation condition of 3NF so we can say that 2NF violation → 3NF violation. Contraposition will be 3NF → 2NF.

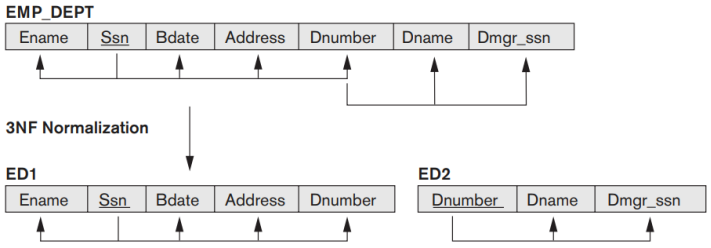
Q : Which of the following relations is in third normal form ? –

a) R(ABCD) FD's : $B \rightarrow C$; $AC \rightarrow D$; $ABD \rightarrow C$; $BCD \rightarrow A$

CK : BD, BA



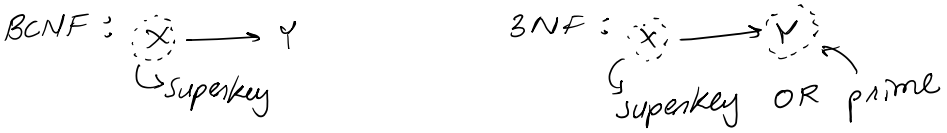




1.2.3) Boyce-Codd normal form (BCNF) :

Definition : A relation R is in BCNF iff for all non-trivial FDs $X \rightarrow Y$, X should be superkey.

3NF is saying this should not happen : NOT SK → Non-prime. Which means for every non-trivial FD X → A this should happen : X is either SK or A is prime.



Which means R is in BCNF → R is in 3NF

Q : What makes a relation to be in 3NF but not in BCNF ? – When for some non-trivial FD, X → A where X is not a superkey and A is prime.

If a relation is in 3NF and not in BCNF, then there must be a non-trivial FD A→B, where B is a prime attribute and A is not a super key (assume A is the minimal attribute set which determines B). This means B is not a super key as otherwise A must also be a super key. But since, B is a prime attribute, for some non-empty set of attributes Y, BY is a candidate key. Since A→B, and A is assumed to be minimal, this means AY is also a candidate key. Thus, we got two candidate keys AY and BY and Y is common. This proofs below sentence.

" A necessary (but not sufficient) condition for a relation to be in 3NF but not in BCNF is that it should have overlapping candidate keys – two candidate keys of two or more attributes and at least one common attribute "

Note Q is necessary condition for P == P->Q == *if 3NF but not in BCNF then overlapping cand. key*

Q : A prime attribute can be transitively dependent on a key in a BCNF relation ? –

$\alpha \rightarrow \beta \rightarrow A$ $\quad A \not\subseteq \alpha$, $A \not\subseteq \beta$ } by definition of transitive FD

But In BCNF if $X \rightarrow Y$ then X should be superkey so $\beta \rightarrow A$ here β should be superkey meaning $\beta \rightarrow C$ but this is contradiction to transitively FD definition. So above statement is false

//Lecture 9

1.3) Decomposition and Normalization :

Before starting this, we will explore some concept regarding relation.

Natural Join : Natural way of joining two relations/tables

Suppose you have two relation student and pay-grade and you want to know answer of following question or queries.

S	name	SSN	DOB	Grade
	A	121	2367	2
	A	132	3678	3
	A	101	3498	4
	A	106	2987	2

T	Grade	Salary
	2	80
	3	70
	4	70

Queries : Salary of (SSN == 132) = 70

Sometimes, we must join two relations to get some information and make one relation/table based on following :

- Equating attributes of the same name, and
- Projecting out one copy of each pair of equated attributes.

We denote this operation by a symbol ⋈ and in above case we write $S \bowtie T$:

$S \bowtie T$	Name	SSN	DOB	Grade	Salary
	A	121	2367	2	80
	A	132	3678	3	70
	B	101	3498	4	70
	C	106	2987	2	80

R:

A	B
X	Y
X	Z
Y	Z
Z	V

S:

B	C
Z	U
V	W
Z	V

No matching

R ⋈ S :

A	B	C
X	Z	U
X	Z	V
Y	Z	U
Y	Z	V
Z	V	W

column = Degree = Arity = 3

Q : What if no common attribute with same name in two relations… how to do natural join ? – Combine every row of one relation with every row of other relation. This is same as *cartesian product*.

Anomalies/Problems (due to redundancy) :

1) ***Update Anomaly*** : after modification of some row we have to change the other row also because of FD. Or may be other table have some attributes of previous row then it also invalidates.
2) ***Insertion Anomaly*** : Suppose we want to add new course to one department and we don't have student entry till now so we cannot put NULL in primary column.
3) ***Deletion Anomaly*** : Suppose there is one student in some courses and he passed now we need to delete that entry but course must remain as we don't want to delete course. But after deleting name we can't put NULL in primary column also.
4) ***Redundant storage*** : some information is stored repeatedly.

Solution : Decomposition ! and **BCNF (which removes the anomalies)**

1.3.1) Decomposition :

Suppose that relation R contains attributes A1,…, An. A **decomposition** of R consists of replacing R by two or more relations such that :

- Each new relation scheme contains a subset of the attributes of R, and
- Every attribute of R appears as an attribute of at least one of the new relations.

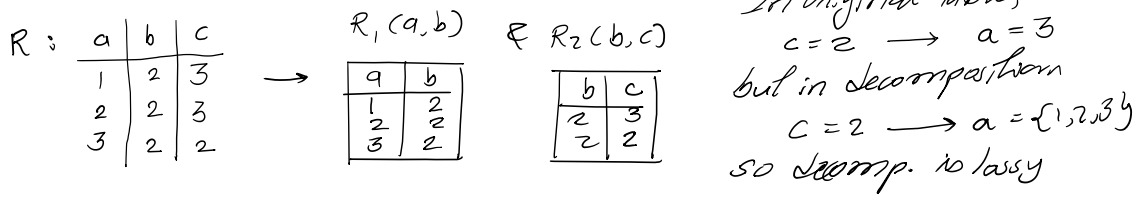
Consider, Relation $R(a,b,c,d,e)$ then $D_1(ab,cd,e)$ valid decomp. $D_2(ab,cd)$ not valid

1) **Lossless Vs lossy decomposition** :

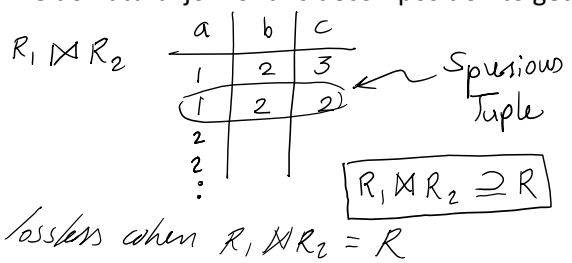
Lossless decomposition : Will never give wrong information

Lossy decomposition : for some instance will give us wrong information.

Example, Binary decomp.



If we do natural join of this decomposition to get original relation, we will get



In Lossy decomposition, we are not losing any original tuple instead getting spurious tuples which is loss of storage

But Decomposition R1(a, b) and R2(a, c) is lossless. Now, we can conclude that a **binary** decomposition R (yes this condition is only for binary) into (R1, R2) is

Lossless : iff common attributes of R1, R2 is a superkey in at least one of R1 or R2 meaning

$$R_1 \cap R_2 \rightarrow R_1 \textbf{ or } R_1 \cap R_2 \rightarrow R_2$$

Lossy : iff common attributes of R1, R2 is not a superkey in any of the R1 or R2 meaning

$$R_1 \cap R_2 \nrightarrow R_1 \textbf{ and } R_1 \cap R_2 \nrightarrow R_2$$

//Lecture 10

Therefore, to check lossless and lossy decomposition first find common attributes then find if it is key of one of the relations. For this you take closure of common attributes and see if its closure contains all the columns of one of the relations.

2) Dependencies preservation :

Suppose, some dependencies associated to R relation and now you decompose R into R1 and R2. Then some dependencies will be for R1 (let's say F1) and some will be for R2 (Let's say F2). Then you again combine both relation into one can it happens that we get some new dependencies apart of F i.e. $F^+ \subset (F_1 \cup F_2)^+$ - Answer is no, you can never have.

Example,

$R = \{a, b, c, d, e\}, F = \{a \rightarrow bc, cd \rightarrow e, b \rightarrow d, e \rightarrow a\}$ and $R_1 = \{a, b, c\}, R_2 = \{a, d, e\}$

First, It is lossless. Let F_1 = FDs Holding on R_1 ∴ $F_1 = \{a \rightarrow bc, bc \rightarrow a\}$

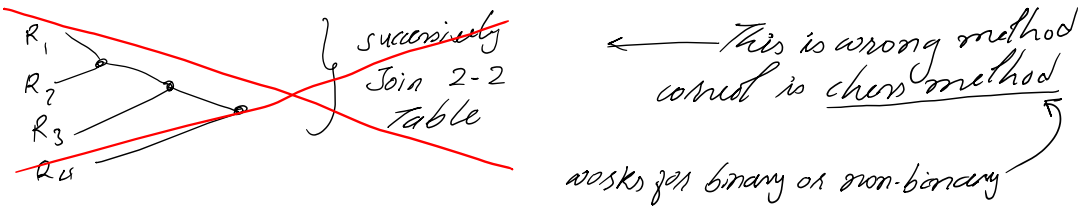
Similary, $F_2 = \{a \rightarrow de; e \rightarrow a\}$

In $F_1 \cup F_2$, $b^+ = \{b\}$ & $cd^+ = \{cd\}$ so R_1 & R_2 is not dependencies preservation

So, $b \rightarrow d$ this FD without joining tables cannot be enforced. That is why we say not Dep. Preserving.

Here F1 is called **FDs preserved by R1** only and similarly for F2. This also explains perseveration of FD's after decomposition. After finding F1, F2, F3, …. If $(F_1 \cup F_2 \cup F_3)^+ = F^+$ then decomposition is DP.

Q : How to check lossy/lossless for non-binary decomposition ? – you must have seen this last year,



//GO YouTube playlist decomposition of a relation

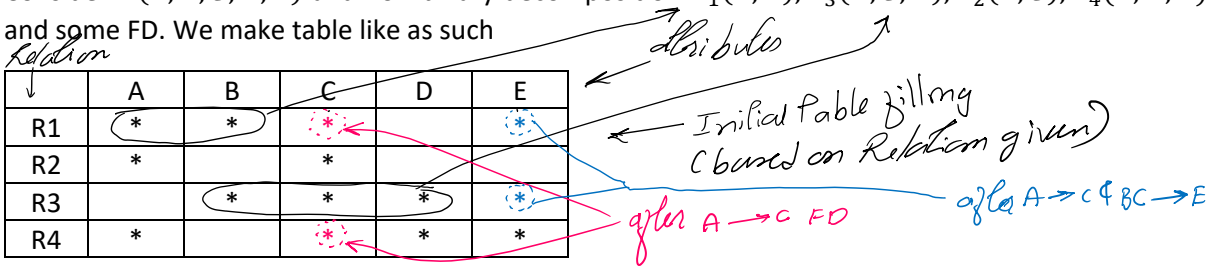
Lossless join ≡ Non-additive join

1.3.2) Non-binary decomposition :

We have wrong method which was successive combining method to check lossless/lossy decomposition. One thing to note that if this method answers lossless then decomposition will be lossless. But answers lossy then decomposition may or may not be lossy.

Chase algorithm : (by example)

Consider $R(A,B,C,D,E)$ and non-binary decomposition $R_1(A,B), R_3(B,C,D), R_2(A,C), R_4(A,D,E)$ and some FD. We make table like as such

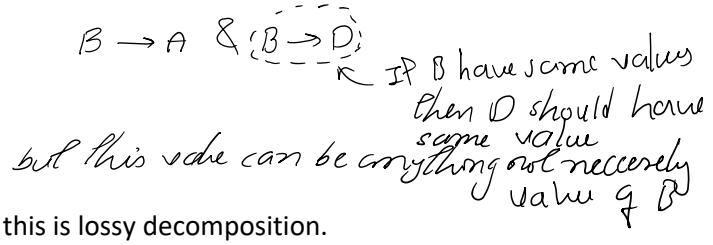


	A	B	C	D	E
R1	*	*	*		*
R2	*		*		
R3		*	*	*	*
R4	*		*	*	*

Now, suppose we have FD $A \rightarrow C$ then we will fill all the star place in C which is present in A but not in C. Suppose we also have $BC \rightarrow E$ then we will apply procedure. Similarly, for the rest FD we do this **repeatedly** and at last we check if there exists a row with all * entries then lossless else lossy.

NOTE : Even after applying some FD if there exists a row with all * entries you can conclude result.

Q : Consider the relation R(A, B, C, D) with the FD : B → AD and the proposed decomposition {A, B}, {B, C}, and {C, D}. -

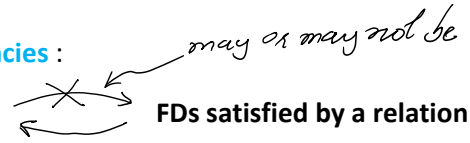


	A	B	C	D
R1	*	*		#
R2	*	*	*	#
R3			*	*

So, there is no row containing *. Thus, this is lossy decomposition.

//Lecture 12

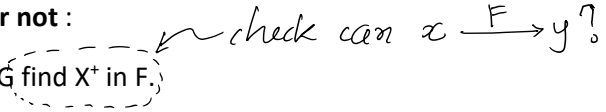
1.4) Minimal cover of FDs and redundancies :



FDs satisfied by a relation instance **FDs satisfied by a relation schema**

Thus, superkey of a particular instance r(R) are the **potential candidate keys** for the relation R. Potential candidate key exists in relation instance.

Checking F covers G or not :



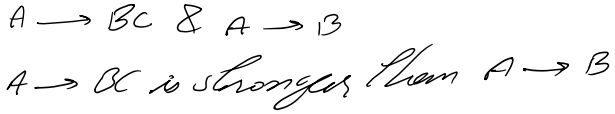
For every FD X → Y in G find X^+ in F.

Functional dependencies set F is equivalent to G iff F covers G and G covers F

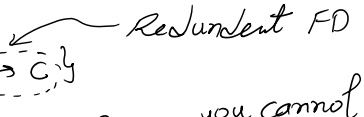
1.4.1) Minimal cover of FDs :

Minimal cover of FDs = minim**al** cover/ canonical cover/ irreducible FD set

For Two FDs F1, F2 we say F1 is stronger than F2 iff F1 → F2.



Back to minimal cover of FDs



FD set : $F = \{a \rightarrow b, b \rightarrow c, a \rightarrow c\}$

$\therefore F_c = F_m = F_{IR} = \{a \rightarrow b, b \rightarrow c\}$ ← you cannot reduce further

Before this we have to cover few small topic :

Extraneous attribute on LHS of a FD :

$FD : \{A \to C; AB \to C\ y \equiv \{A \to C\ y$

↖ extraneous attribute

Definition : *$XA \to Y$ FD in F1 we say A is extraneous/redundant in FD F1 if $X \to Y$.*

A common algorithm to produce a minimal cover consists of three steps :

- Split the right part, producing FDs with only one attribute on the right part. And also remove trivial FD.
- For all FDs on LHS find a redundant (extraneous) attribute
- Eliminate all redundant FDs

//Lecture 13

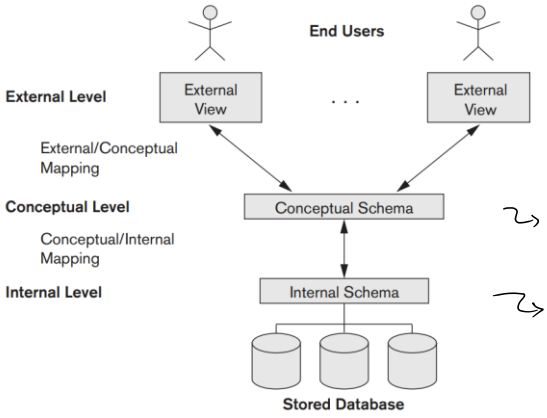
1.4.2) Three-Schema Architecture and Data Independence :

Three levels of schema : feature provided by DBMS

We know that all the data shouldn't be visible to all the types of users.

stud (schema, P(C))
↖ Not stored in DB
It is derived from CS

student (sid, DOB, name)
↖ stored on DB



↳ Use of data

↳ what data in DB

↳ How data is actually stored

Q : Why we need Three-levels of schema ? –

The DBMS can enforce access controls that govern which data is visible to different classes of users.

Another reason is **data independence** :

There are two types of data independence namely, physical data independency and logical data independency.

- **Physical data independency** means independence of conceptual schema from physical change. For example, if you change data structure from array to list it will not affect conceptual schema. It tells you how much physical schema you can change without affecting conceptual schema.
- **Logical data independency** means External schema independence from conceptual schema changes.

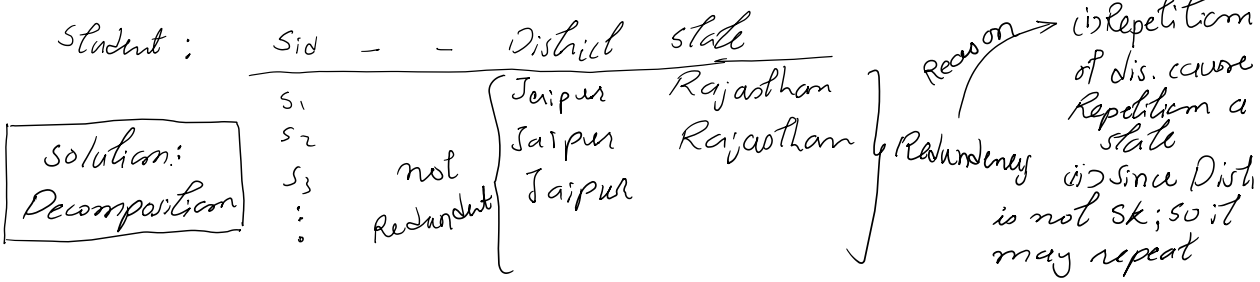
//Lecture 15

1.4.3) Redundancies :

Trivial FDs never create redundancies in the database because those are trivial, and they always hold on all relations.

Non-trivial FDs may create redundancies in the database

For example, students(sid, ….., district, state) and FD district → state



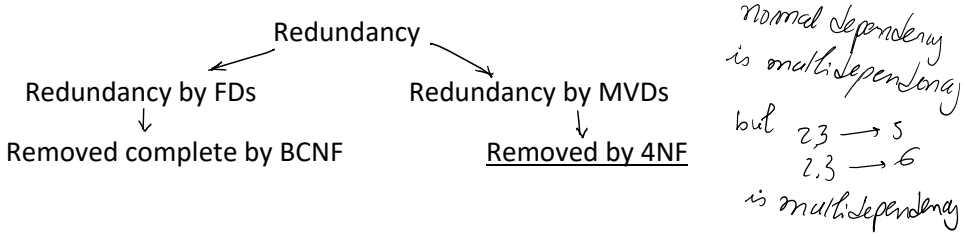
Non-trivial FDs in which LHS is a superkey never create redundancies in the database

For example, in BCNF, every non-trivial FD has LHS as SK which means

- Free from redundancies ? ✗
- Or Free from redundancy caused by FDs. ✓

Which means you cannot have redundancy caused by FDs in BCNF. but is it free from all redundancies ? – No, there is still redundancies. Because we put more than one "many to many" relationships in single table. For example, Students(Sid, cid, movie) here sid to cid (here many students may have taken may course and many courses may be taken by many students.

Such situation gives birth to a new concept of **MVDs** (**multi valued dependencies**)



4NF : A relation is in Fourth Normal Form (4NF) if and only if for every nontrivial multi-valued dependency, A↠B, A is a super key of the relation.

//Lecture 16

1.4.4) 3NF, BCNF decomposition :

NOTE : A database $\{R_1, \dots, R_n\}$ is in 3NF if each relation schema R_i is in 3NF and similarly for BCNF.

Q : If R is not in BCNF, then how to make it into BCNF ? – decompose R into sub-relations such that Each sub-relation is in BCNF.

Do BCNF decomposition : give a lossless decomposition(must) if possible, you can also give me DP.

If it is a BCNF decomposition : just check each individual sub-relation if they are in BCNF or not.

BCNF decomposition algo :

Which guarantees a lossless decomposition into BCNF sub-relations,

- If R is already in BCNF stop algo

- If R is not in BCNF : take a violation of BCNF (meaning take X → α. Where X is not SK) Find X^+ = R1 and do R2 = (R - X^+) ∪ (X)

Example, $R(A, B, C, D)$ with $A \to B; B \to C$.

Here AD = CK meaning both FD are violation
we take 1st violation, A → B ; A^+ = {ABC}
$R_1 = (R - A^+) \cup R$ = {A, D} so $R_1(A, D)$ & $R_2(A, B, C)$
only two attributes meaning BCNF

Take 2nd violation, B → C & B^+ = {B, C}
$R_2 = (R - B^+) \cup B$ = {B, A}
Final BCNF decomposition : $R_1(A, D)$, $R_2(B, C)$, $R_3(A, B)$

You may get different BCNF decompositions depending on the order of violations that we take.

NOTE : for ALL relations, lossless BCNF decomposition is ALWAYS possible. But for some relations, lossless dependency preserving BCNF decomposition does NOT exist.

3NF decomposition (synthesis algorithm) :

Guarantees a lossless, as well as dependency preserving decomposition into 3NF sub-relations.

- Find minimal over of FDs
- In fd : if LHS is same then merge those FDs (for example, $X \to A_1; X \to A_2 \implies X \to A_1A_2$
- For every FD, make table/relation (for example, for $X \to A_1A_2 \Rightarrow R_1(XA_1A_2)$
- If none of the relations from step 3 contains a CK take any one CK and make one separate relation for it in case if they don't have CK.
- Delete some relations : If $R_2 \subset R_1$ then delete R_2.

NOTE : For ALL relations, lossless 3NF decomposition is ALWAYS possible. For ALL relations, lossless dependency preserving 3NF decomposition is always possible.

Q : A decomposition D of relation R (with FD set F) is dependency preserving then – each dependency of F can be enforced by dealing with an individual relation in D.

Third Normal form which is always preferred.

Silly mistakes : below is wrong find mistake

Consider relation $R(A, B, C, D, E, F, G, H)$ and the functional dependencies $\{AB \to C, BD \to EF, A \to G, F \to H\}$. If $2\ NF$ decomposition of R requires minimum 'a' relations, and $3\ NF$ decomposition requires minimum 'b' relations, the value of $a \times b =$ ____ First do 2NF procedure ⟹ a = 3 (A, B, C) (B, D, E, F, H) (A, G)
Then by using 2NF Relations do 3NF procedure ⟹ b = 4 (A, B, C) (B, D, E) (F, H) (A, G)

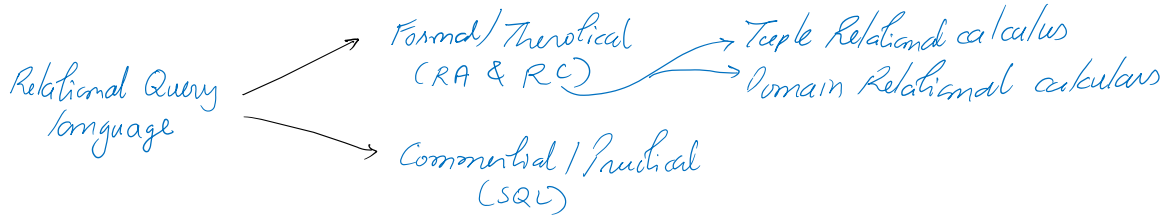
Your Answer: 16 Correct Answer: 12 Incorrect Discuss

2. Queries

//mod2-Lecture 1

Query languages are specialized languages for asking questions, or queries, that involve the data in a database.

Along with relational model Codd. Gave two query languages namely relational algebra and relational calculus. These are theoretical query languages.



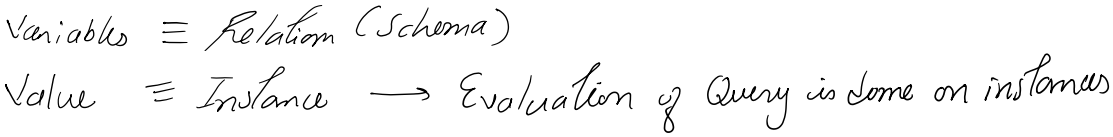
In **relation algebra** query, we need to specify everything from what you want? To how to get it ? (step by step procedure in what order) we can say it is **procedural query language**.

Whereas in **relational calculus** query, we need to specify just what you want. We don't care how it will get it. It is more abstract. It is also known as **declarative query language**. SQL is one such example.

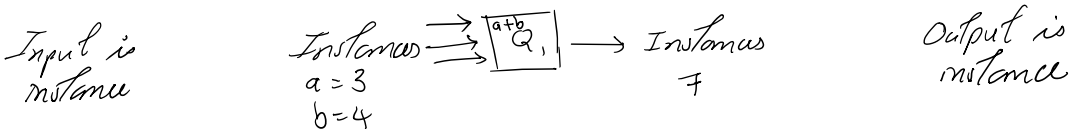
From programmer's point of view : relational calculus is easier. But from QL compiler point of view, relational algebra is easier and more useful.

2.1) RELATIONAL ALGEBRA :

Just like math algebra. We have variable and values.



An algebra whose operands are relations or variables that represent relations. and operators are designed to do the most common things that we need to do with relations in a database.



2.1.1) Basic Operators :

Six basic operators in relational algebra,

Select	σ	*Selects a subset of tuples from reln*
Project	π	*Deletes unwanted columns from reln*
Cartesian product	$\times$	*Allows to combine two relations*
Set-difference	$-$	*Tuples in reln 1, but not in reln 2*
Union	$\cup$	*Tuples in reln 1 plus tuples in reln 2*
Rename	ρ	*Renames attribute(s) and relation*

Additional, derived operators : join, natural join, intersection, etc.

1) **Selection Operator** :

Select your desired tuples and discard remaining tuples. Example,

Users with popularity higher than 0.5

$S = \sigma_{pop>0.5} User$ ← Relation table; pop>0.5 ← condition

User :

vid	name	age	pop
142	Bart	10	0.9
123	Milhouse	10	0.2
857	Lisa	8	0.7

→ $\sigma_{pop>0.5}$ →

S

vid	name	age	pop
142	Bart	10	0.9
857	Lisa	8	0.7

We can observe one thing that it is unary operator, schema not changes, cardinality may change. It is unary because we check every tuple independently and individually. For example,

Find the rows with maximum marks.

Student :

Sid	marks
a	2
b	3
a	4

~~$\sigma_{marks \geq \forall marks}$ (Student)~~

Because we can see only 1 row a time

I.e. <selection condition> is applied independently to each individual tuple t in relation.

2) Project operation :

Selecting desired attributes and rejecting rest. Example,

Projected Relation

R :

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

→ $\Pi_{A,C}(R)$ →

A	C
α	1
α	1
β	1
β	2

≡

A	C
α	1
β	1
β	2

Here cardinality changes but it never increases.

If the projection list is a superkey of R that is, it includes some key of R – the resulting relation has the same number of tuples as R.

NOTE :

When you apply any of this operation then in output you will get new table with no name.

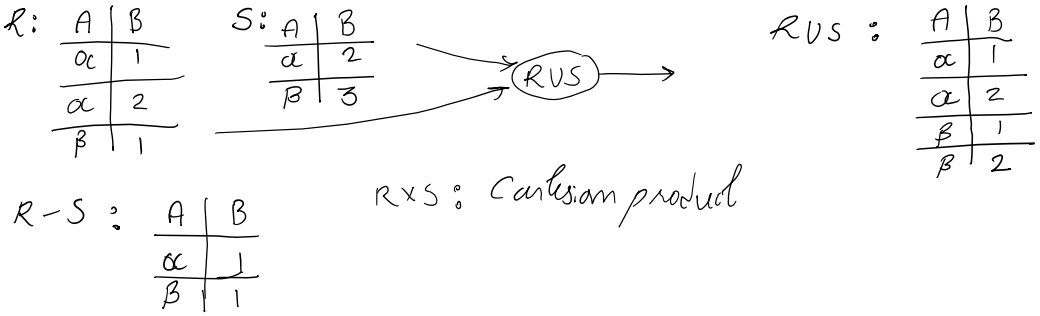
The result of the PROJECT operation has only the attributes specified in <attribute list> in the same order as they appear in the list. for example, $\pi_{gpa,name}S$ then output relation contain attribute in gpa, name order.

Property of operation learned till now…

- *Selection operation is commutative*. $\sigma_{p1}\left(\sigma_{p2}(R)\right) = \sigma_{p2}\left(\sigma_{p1}(R)\right)$ this is true because we can write this as $\sigma_{p1 \wedge p2} = \sigma_{p2 \wedge p1}$.
- *But project operation is not commutative.* Because there are changes that after applying, we do not get any attributes.

//Lecture 2

3) Set operations : (Union, intersection, set difference)



Union, intersection, set difference ← to define these operations relations must have compatible scheme. (union compatible schema)

R is **Union-compatible** with S iff

- # attributes (R) = # attributes (S)
- ith attribute of R must have some domain/type as ith attributes of S.

× ← cartesian product is not union-compatible.

4) Assignment operator :

Temp ← $\sigma_p(R)$ and Trans2 ← $\pi_A(temp\ 2)$

5) Renaming :

Input : a table R and S.

Output : a table with the same rows or its columns differently.

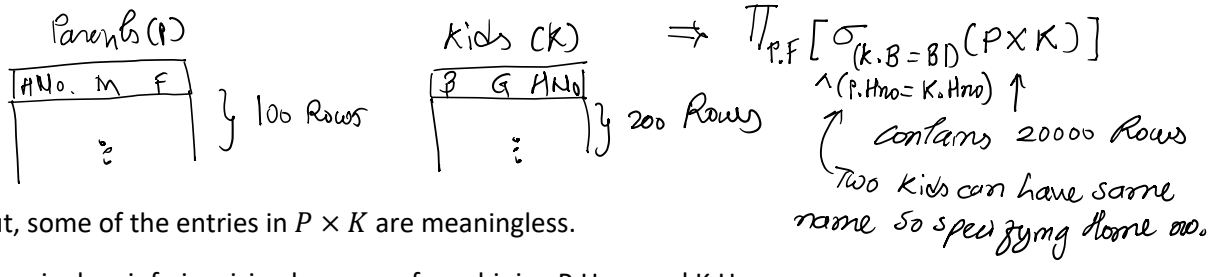
Notation : $\rho(A_1, A_2, \ldots A_n) R_1$ or $\rho S(A_1, A_2, \ldots)$

//Lecture 3

2.1.2) Derived Operators :

Sometimes we need to combine two or more tables (same or different) to retrieve some information.

For example, find father of B1 : to do this we do cartesian product between parents table and kids table.



But, some of the entries in $P \times K$ are meaningless.

Meaningless info is arising because of combining P.Hno. and K.Hno.

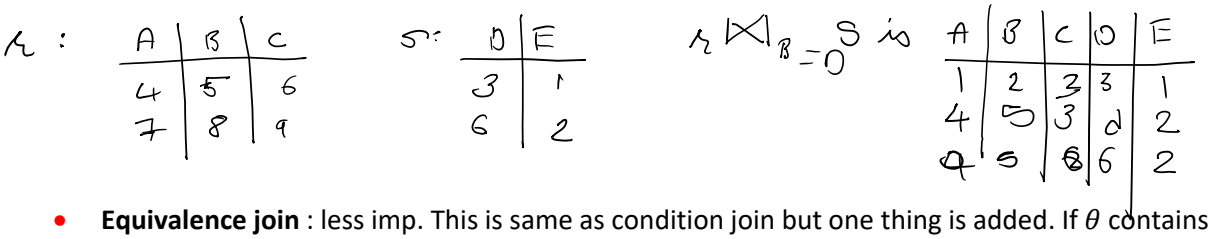
For this reason, we can create one separate operation. In general, say column C is common in both table T_1 and T_2 then **conditional join** is defined as $\boldsymbol{T_1 \bowtie_c T_2 = \sigma_c(T_1 \times T_2)}$.

1) Join Operation :

Two types of join operation : **inner join** (by default) and **outer join** operation.

In Inner join there are 3 types of inner join.

- **Condition join** : $R \bowtie_C S$ also called theta join $R \bowtie_\theta S$.



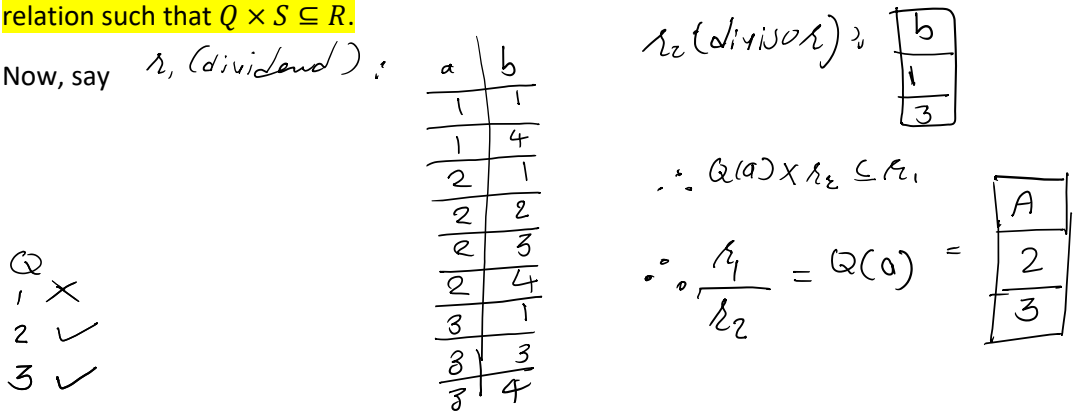
- **Equivalence join** : less imp. This is same as condition join but one thing is added. If θ contains = condition not containing $<, \leq, >, \geq$.
- **Natural join** : $R \bowtie S$. Equality of all common attributes.

//Lecture 4

2) Division operator :

In math, for integer A and B, A/B is the largest integer Q such that Q*B <= A.

Similarly, in Queries, for relation R and S, R/S = Q, here Q is also relation. This means Q is the **largest** relation such that $Q \times S \subseteq R$.



To find, we check which attributes of "a" are related with all the value of "b" in r2. By doing so we are guarantying that $Q(a) \times r_2 \subseteq r_1$.

//Lecture 5

Shortcut : R(Kids, Game) ; S(Game)

R/S ? = kids who play all the S Games = All kids – kids who do not play some S games.

Kids who don't play some games = π_{kids}(all possible pair of students – all existing pair)

$(\Pi_{kids} R) \times S$ $\quad$ R

$$\boldsymbol{R/S = \pi_{kids}(R) - \left(\pi_{kids}\left((\pi_{kids}R) \times S - R\right)\right)}$$

If $S = \phi$ then $R/S = \Pi_{kids}(R)$

In general, this is called *Healy implementation*

$$\boldsymbol{R(A,B)/S(A) = Q(B) = \pi_B(R) - \pi_B[S \times \pi_B(R) - R]}$$

$A_1 A_2 \ldots A_n$ $\quad$ $B_1 B_2 \ldots B_m$ $\quad$ $A_1 A_2 \ldots A_n$

Another implementation,

Q : R(kids, Game); S(Game); for every S.Game gi, we find the set of kids Ki who play that S.Game gi…

What can we say about intersection of all Ki ? – This is same as division operation i.e. R(Kids, Game)/S(Game)

K1 = all the kids who plays g1

K2 = all the kids who plays g2…. Kn = all the kids who plays g3

$K_1 \cap K_2 \ldots \cap K_n$ = all the kids who play g1, g2, g3, g4, …, gn = kids who play all S games

Meaning $\boldsymbol{R(A,B)/S(B)} = \cap_{\forall b_i \in B} \underbrace{\boldsymbol{\pi_A(\sigma_{B=b_i} R)}}_{A_i}$

Max. value of $|R/S| = \left\lfloor \frac{|R|}{|S|} \right\rfloor$ and min. value of R/S is 0.

3) Outer join operation :

Suppose we join $R \bowtie S$. A tuple of R which doesn't join with any tuples of S is said to be **dangling of R**. and Similarly, A tuples of S which doesn't join with any tuples of R is said to be **dangling of S**.

If R and S are two relations then $R \bowtie S$ may contains some dangling tuples in R, S. Outer join includes dangling tuples in the final answer with help of NULL.

Outer join : Inner Join + dangling tuples

Left outer join ($\bowtie$) **:** Dangling tuples of R are included in the final result of $R \bowtie S$ with the help of NULL.

Right outer join ($\bowtie$) : Dangling tuples of S are included in the final result of $R \bowtie S$ with the help of NULL.

FULL outer join ($\bowtie$) : Dangling types of R, S are included in the final result of $R \bowtie S$ with the help of NULL. By default, outer join ≡ full outer join.

For exmple,

R =

name	phone
A	B
C	D

Dangling Tuple of R

S =

name	email
A	F
G	H

Dangling Tuples

R ⋈ S (Natural Join) =

name	phone	email
A	B	F

R ⟕ S (Left Join) =

name	phone	email
A	B	F
C	D	Null

R ⟖ S (Right Join) :

name	phone	email
A	B	F
G	Null	H

R ⟗ S (full) =

name	phone	email
A	B	F
C	D	Null
G	Null	H

//Lecture 6

In short, we can say that,

$$\boldsymbol{R \bowtie S = R \bowtie S + R_{Dangling}; R \bowtie S = R \bowtie S + S_{dangling}}$$

$$\boldsymbol{R \bowtie S = R \bowtie S + R_{Dangling} + S_{dangling}}$$

Expressing outer join in terms of basic operations :

all r which not part of r ⋈ s

$$R \bowtie S = R \bowtie S + R_{Dangling}$$

$R - (\Pi_R(R \bowtie S))$

$$R \bowtie S = R \bowtie S \cup \left[(R - \pi_r(R \bowtie S)) \times \{\underbrace{\boldsymbol{null}, \boldsymbol{null}, \ldots}\}\right]$$

# of Nulls = # attributes(S) - common attribute

//Lecture 7

Sample database for upcoming content :

Sailors(*sid:* **integer**, *sname:* **string**, *rating:* **integer**, *age:* **real**)
Boats(*bid:* **integer**, *bname:* **string**, *color:* **string**)
Reserves(*sid:* **integer**, *bid:* **integer**, *day:* **date**)

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Figure 4.15 An Instance S3 of Sailors

sid	*bid*	*day*
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Figure 4.16 An Instance R2 of Reserves

bid	*bname*	*color*
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Figure 4.17 An Instance *I1* of Boats

Q : Find the sid's of sailors who have reserved boat 103. – $\Pi_{sid}(\sigma_{bid=103}\ Reserves)$

~~$\sigma_{bid=103}(\Pi_{sid}\ Reserves)$~~

NOTE : $\sigma_c(\pi_A R) \equiv \pi_A(\sigma_c R)$ iff condition c only applies on attributes of A. Because if $A \cap c = \emptyset$ then $\sigma_c(\pi_A R)$ would be invalid.

Whenever you find "all" in queries (in some), we use division operator.

Q : Find the names of sailors who have reserved boat 103. – $\Pi_{sname}(\sigma_{bid=103}\ R \bowtie S)$

But this gives us more useless tuples so to reduce that we can just first select 103 from S…

$\Pi_{sname}(S \bowtie (\sigma_{bid=103}\ R))$

So, to save computational time apply selection as early as possible; if possible…

Q : find the names of sailors who have reserved a red boat. – $\Pi_{sname}(S \bowtie (R \bowtie (\sigma_{color=red}\ B)))$

Q : find the colors of boats reserved by Lubber. – $\Pi_{color}[B \bowtie \Pi_{bid}(R \bowtie (\sigma_{sname=Lubber}\ S))]$

Q : find the names of sailors who have reserved at least one boat. – $\Pi_{sname}(S \bowtie \Pi_{sid}(R))$

Q : find the names of sailors who have reserved a red and a green boat. –

~~Snames of red boat people ∩ Snames of green boat people~~ – this will not work in some relations consider following counter…

Sid	colors	Same
21	Red	John
20	Green	John

} Two different person but we are considering them same

But we know sid is unique so first we can find intersection in sid then find name that would make sense.

$\Pi_{sname}(S \bowtie (\text{sid of red boat people} \cap \text{sid of green boat people}))$

Q : Find the names of sailors who have reserved at least two boats. –

$S \bowtie R_1 \bowtie R_2 \Rightarrow \Pi_{sname}((S \bowtie R_1) \bowtie_{R_1.Bid \neq R_2.Bid} R_2)$

S(sid, sname), R_1(sid, Bid), R_2(Sid, Bid); S.sid = R_1.sid, R_1.Bid ≠ R_2.Bid

//Lecture 8

If we not write R1.bid ≠ R2.bid then query will return name of sailors who have not reserved no boats. Meaning at least 1. This is happening because tuple of S can select same boat from R1 and R2. So, we need not equal condition between boat id.

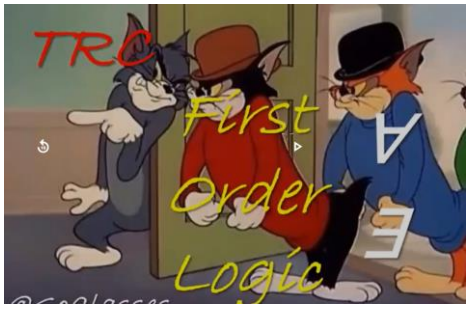
MIN and MAX # of tuples of $R \bowtie S$:

When all the attributes of R are different from S then we have max tuples = $|R \times S|$.

And when all the attributes of R and S is same (meaning same domain) but there is no matching between any instances then the result will be 0.

//Lecture 9

2.2) TUPLE RELATIONAL CALCULUS :



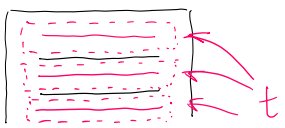
In relational algebra, we have to provide what we want and how it will be computed… hence, procedural query language…

$\Pi_A(\sigma_c(R \bowtie S))$ & $\Pi_A(\sigma_c R \bowtie S)$

Here order of operations given by user. And its output is dependent upon order of operation.

In TRC (tuple relational calculus), we just have to specify what we want… Hence, declarative language.

R (A, B)



R(t) OR t∈R } means t is a tuple variable in Relation R so; t can take tuples of R

$\{t \mid P(t)\}$ → Predicate/condition on t

Those tuples which satisfy Property P

There are two ways to solve problem related to TRC...

way 1	way 2
logical	~~logical~~
logic → Result	Result → logic
	Gives us feel of why TRC is Declarative

Operation on relation R(A, B)	**Relational algebra**	**Relational calculus**
Selection operator of RA	$\sigma_{B=17}(r)$	$\{t \mid t \in R \wedge t[B] = 17\}$
Projection operator of RA	$\pi_A(r)$	$\{t.A \mid t \in r\} = \{t[A] \mid t \in r\}$
	$\pi_{A,C}(r) \neq \pi_{C,A}(r)$	$\{t.A, t.C \mid t \in r\} = \{t.C, t.A \mid t \in r\}$

This was way 1

Now way 2... we go result to logic

$\Pi_A(1) \Rightarrow \{t \mid \exists x \in r\ (x.A = t.A)\}$

r:	A	B	C
	α	10	1
	α	20	1
	β	30	1
	β	40	2

Suppose,

Result

A
α
β

$\{t \mid \exists_{x \in r}\ (x.A = t.A)\}$

there exists some tuple whose A value is same as A value of t

Similarly, $\pi_{A,C}(r) \equiv \{t \mid \exists_{x \in r}(t.A = x.A \wedge t.c = x.c)\}$

$\sigma_{B=17}(r) \equiv \{t \mid \exists_{x \in r}(t.A = x.A \wedge t.B = x.B \wedge t.B = 17\}.$

way 1 way 2

$$\pi_A(\sigma_{B=10}(r)) \equiv \{t.A \mid t \in r \wedge t.B = 10\} = \{t \mid \exists_{x \in r}(x.B = 10 \wedge t.A = x.A)\}$$

$\Pi_{A,B}$

$$\pi_{A,B}(\sigma_{c=1}(r)) \equiv \{t.A, t.B \mid t \in r \wedge t.C = 1\} = \{t \mid \exists_{x \in r}(x.C = 1 \wedge t.A = x.A \wedge t.B = x.B)\}$$

Cross product :

Relations *r*, *s*:

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

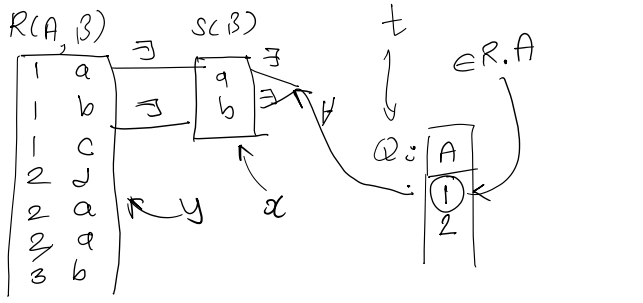
s

r x *s*:

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

$\{t.A, t.B, S.C, S.D, S.E \mid t \in R \wedge S \in S\}$

$= \{t \mid \exists_{x \in R}\ (t.A = x.A \wedge t.B = x.B) \wedge (t.C = x.C \wedge t.D = x.D \wedge t.E = x.E)\}$



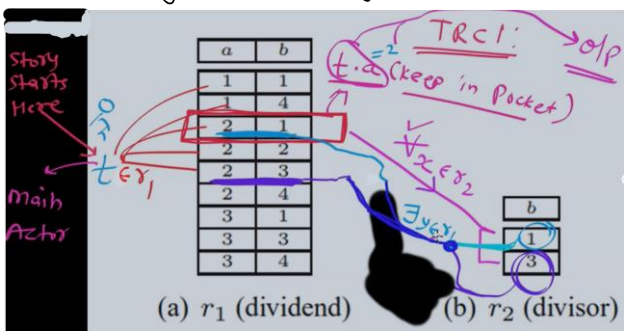
//Lecture 11

Division of relational algebra in TRC :

For every tuple t in the result (i.e. in Q) :

For all tuples of S; there is a tuple y of R such that y = tx.

$\{t \mid \forall x \in S \; \exists y \in R \; (t.A = y.A \wedge x.B = y.B)\}$ ← way 2



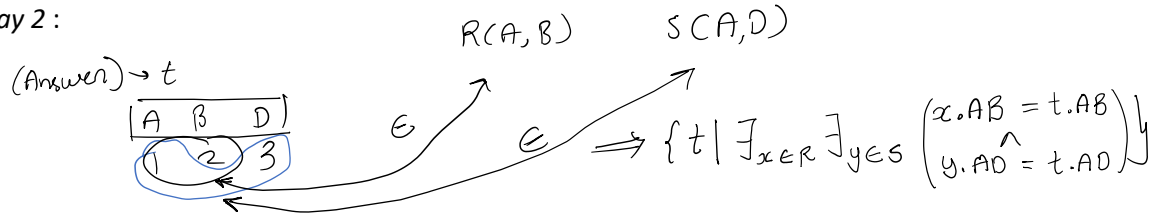
Way 1 : $\{ t.A \mid t \in R \wedge \forall x \in S \; \exists y \in R (t[A] = y[A] \wedge y[B] = x[B]) \}$

Natural join in TRC :

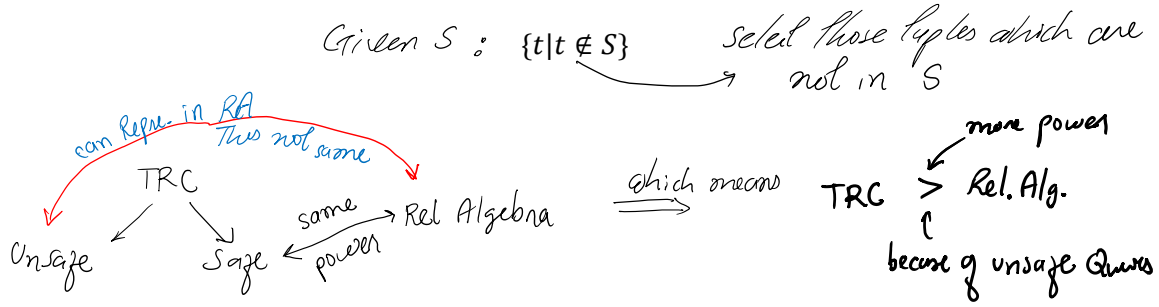
Consider, $R(A, B), S(A, D)$

Way 1 : $\{x, y.D \mid x \in R \wedge y \in S \wedge (x[A] = y[A])\}$

Way 2 :



⋆ **Unsafe query (only for TRC)** : If a TRC query provides any data which is not in the database then that TRC query is unsafe. For example, most popular



For every RA there exists RC but converse is not true !

⇨ SQL has more power than relational algebra, and the main source of this additional power is its aggregation and grouping constructs, together with arithmetic operations on numerical attributes.

//Lecture 13

NOTE : Whatever TRC query, { LHS | RHS }

all free variables → All variable must be bounded except free variables from LHS

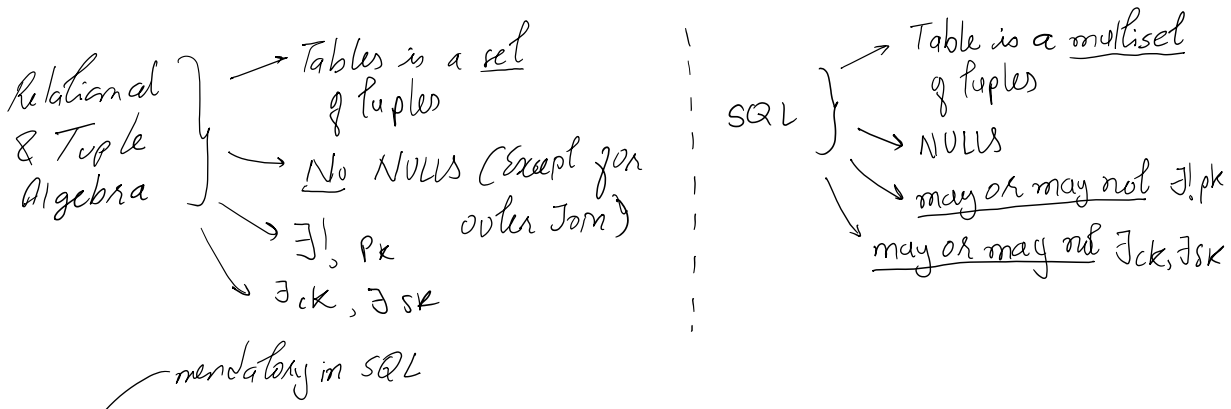
//Lecture 14

2.3) Structured Query Language (SQL) :

SQL is declarative, not procedural.

⋆ ***SQL sublanguages*** :

1) **The data definition language (DDL)** : This subset of SQL supports the creation, deletion, and modification of definitions for tables and views.
2) **The data manipulation language (DML)** : This subset of SQL allows users to pose queries and to insert, delete, and modify rows.



SQL is case insensitive. The basic form of an SQL query is as follows :

```
SELECT  [DISTINCT] select-list
FROM    from-list
WHERE   qualification
```

Eliminate Duplicate Tuples from final Result. It is optional because by default tuples are not removed

optional

Consider, Table R(A, B), S(C, D)

```
SELECT  A, D
FROM    R, S
WHERE   A > 10 ∧ C ≤ 5
```

not similar to : $\Pi_{A,D}\left(\sigma_{A>10 \wedge C\leq 5}(R \times S)\right)$

Because output of above SQL may contain duplicate tuples.

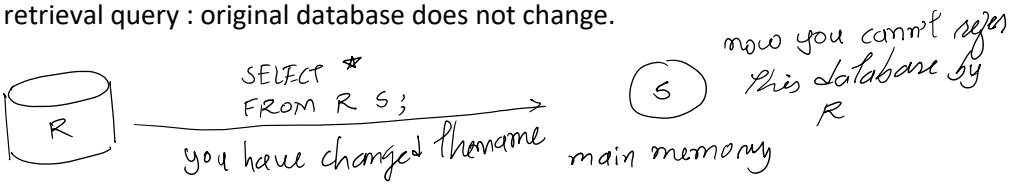
But we add [Distinct] just after SELECT we get same output.

NOTE : In SQL also, if AB is primary key then A, B (one of them) cannot be NULL & no duplicates tuples in AB (so in relation) so if primary key is specified then it is useless to write [Distinct].

To retrieve all data of R(A, B) as it is : ~~R~~

Select * ← means "all columns"
from R ✓

In SQL data retrieval query : original database does not change.



2.3.1) Writing SQL queries :

Consider following data scheme,

Sailors(*sid:* **integer**, *sname:* **string**, *rating:* **integer**, *age:* **real**)
Boats(*bid:* **integer**, *bname:* **string**, *color:* **string**)
Reserves(*sid:* **integer**, *bid:* **integer**, *day:* **date**)

Q : Find names of all sailors with a rating above 7 –

```
SELECT sname FROM Sailors WHERE rating > 7
```

Sailors As S ≡ Sailors S

Q : find the names of sailors who have reserved boat number 103 –

In SQL we don't have ∧ we use "and"

```
SELECT sname FROM Sailors S, Reserves AS R WHERE (bid = 103) ∧ (S.sid = R.sid)
```

But renaming the tables are useless here it is useful when we are using same table two times for example,

Emp (empid, Supervisor)

a	b
b	c
c	c
d	c

Q: Supervisor of Supervisor of a :

⇒ SELECT E2.Supervisor
FROM Emp E1, Emp E2
WHERE (E1.empid = a) ∧ (E1.Sup... = E2.empid)

1) UNION, EXCEPT (minus), INTERSECT :

Note that these three operations are same as relational algebra because they are union compatible and will not return multiset (i.e. no duplicates).

R : A
a1
a2
a3
a4

S : A
a1
a2
a4
a4

SELECT R UNION S ← Invalid

⇒ (SELECT * FROM R) UNION (SELECT * FROM S) ⇒

A
a1
a2
a4

REALITY : SQL first eliminates duplicates from input tables then it will produce result and then again eliminates duplicates.

But what if we want duplicates then

Remove Duplicates →

UNION	UNION all
EXCEPT	EXCEPT all
INTERSECT	INTERSECT all

Not Remove Duplicates

Bag1	Bag2
fruit	fruit
apple	apple
apple	orange
orange	orange

```
(SELECT * FROM Bag1)
   UNION ALL
(SELECT * FROM Bag2);
```

fruit
apple
apple
orange
apple
orange

```
(SELECT * FROM Bag1)
   EXCEPT ALL
(SELECT * FROM Bag2);
```

fruit
apple

```
(SELECT * FROM Bag1)
   INTERSECT ALL
(SELECT * FROM Bag2);
```

fruit
apple
orange

AND is trickier than OR. Because if we want some name of kids who wares red and green T-shirt we cannot just write (colors = "red" and colors = "green") this will always give you empty result. This is happening because colors cannot be red and green at the same time, we have to apply INTERSECT operator.

//Lecture 17

2) Note about Where clause :

In SQL, we do not have $\wedge, \vee, \neg, \neq$ instesad we have AND, OR, NOT, <>.

Select ≡ Select all

//Lecture 18

2.3.2) Aggregate operators :

Average : avg

Total : sum

Minimum : min

Count : count

Maximum : max

These functions take multiset and returns single value.

AVG (DISTICNT A)

SUM (DISTICNT A)

COUNT (DISTICNT A)

} There will remove Duplicate first

MIN (DISTICNT A)

MAX (DISTICNT A)

} same as min(A) max(A)

COUNT(*) : count number of rows in the table.

COUNT (DISTINCT *) : WTH ? it's an error

//Lecture 19

AGGREGATE OPERATORS IN CASE OF NULL VALUES :

Null # ◯ = Null

Arithmatic operation (+, -, x...) anything

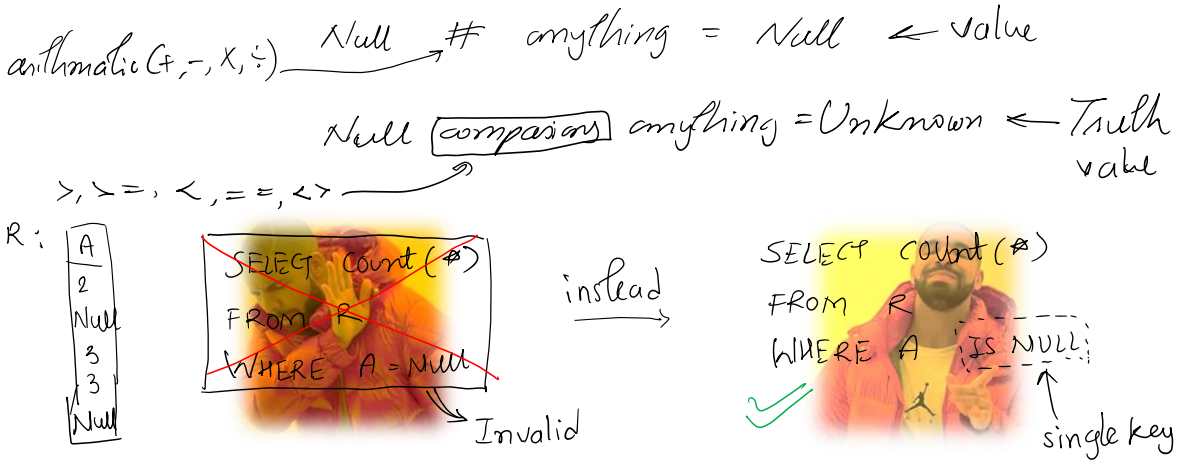
Aggregate attribute always removes NULLs and then calculates. But note that COUNT(*) is different as it gives no. of rows so it cannot ignore NULL entries.

But, R : [A] → empty Relation ⇒ count(A) = 0, count(*) = 0
sum(A) = Null
min(A) = Null
...
} By Definition it is null

In SQL, we have 3-truth value system namely, True, False and Unknown (So when to use unknown?)

Any comparison with NULL results in Unknown

Which means WHERE clause passes a tuple only when "TRUE" and fails when "FALSE" or "UNKNOWN".



//Lecture 20

Logical Connectives in Three-Valued Logic

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN
OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN
NOT			
TRUE	FALSE		
FALSE	TRUE		
UNKNOWN	UNKNOWN		

2.3.3) Group by, having, order by clause :

GROUPING TUPLES : Consider students(id, name, state, marks)

It is easy to find avg marks of all student but how about average marks of all students of Gujrat ?

No, we can find that by `SELECT avg(marks) FROM students WHERE state = 'Gujrat'`

How about finding average marks of all students state wise.

`SELECT avg(marks) FROM students GROUP BY state` – invalid query (as per SQL – 92)

Now, consider same schema, Output the average marks of students, state wise, but only for those states where average marks > 5.

```
SELECT state, avg(marks) FROM students GROUP BY state HAVING avg(marks) > 5
```

Applies to each group

This is fine for group as it is aggregate not attribute

filtering Groups

Those groups which HAVING Avg(marks) > 5

WHERE (filters tuples)	HAVING (filters group)
Works on **tuple** after **tuple**	Works on **group** after **group**
Check every **tuple** individually, independently	Check every **group** individually, independently

NOTE : GROUP BY without aggregation is equivalent to simple SELECT DISTINCT

```
SELECT A
FROM R
GROUP BY A
```
≡
```
SELECT DISTINCT A
FROM R
```

Something interesting : If the GROUP BY clause is omitted, the entire table is regarded as a single group. That's why, aggregation without GROUP BY clause gives SINGLE tuple in the output. 😱

```
GROUP BY A, B HAVING ..... Aggregated attributes
```

Unaggregated attributes must be subset of group by clause

any attributes

```
SELECT X, aggregation
GROUP BY X
```

→ set of attribute

If NULLs exist in the grouping attribute, then a separate group is created for all tuples with a NULL value in the grouping attribute.

ORDER BY CLAUSE : by default, it is ASC

ORDER BY A ≡ ORDER BY A ASC

But in case of ORDER BY A DES, B ASC – first A is sorted in descending order and in case of tie it is sorted by B in ascending order

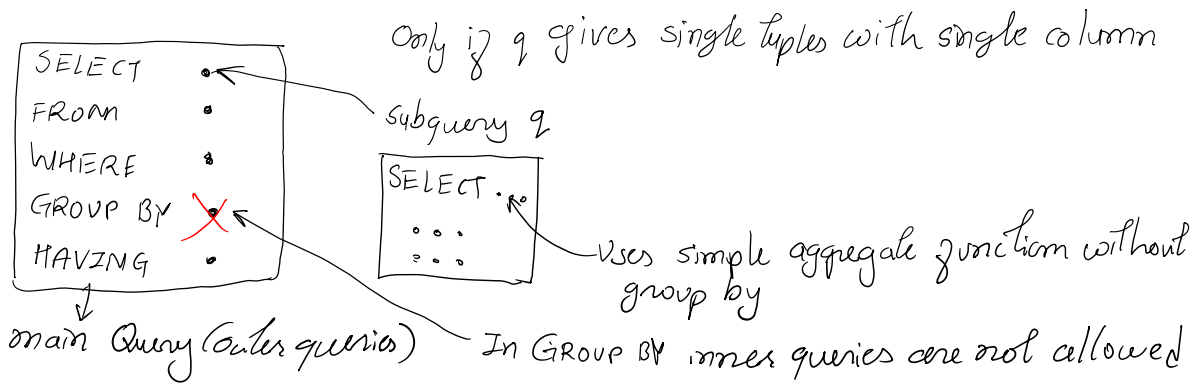
Table Alias → From student AS S

Column Alias → SELECT Avg(marks) As A

Order of evaluation : FROM > WHERE > GROUP BY > HAVING > SELECT > ORDER BY

//Lecture 22

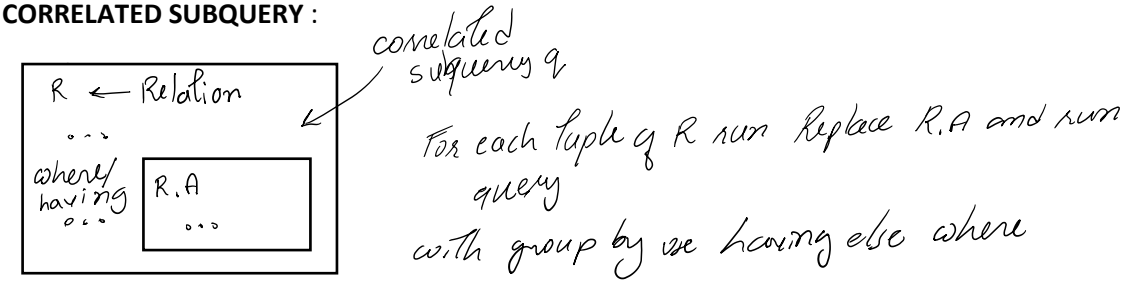
2.3.4) Nested Queries :



There are two types of subqueries :

1) *Simple Subquery* : subquery without co-relation with outer query
2) *Co-related subquery* : subquery using attributes of outer query

CORRELATED SUBQUERY :



⋆ **SOME SQL KEYWORDS** : ALL, EXISTS, ANY, UNIQUE, IN, NOT IN, NOT ALL, NOT EXISTS, NOT ANY, NOT UNIQUE

- **IN**, **NOT IN** – set membership keywords

Consider tuple S and subquery R (*relation R must be result of a subquery*)

S **IN** R = true iff tuple S is in R

S **NOT IN** R = true iff tuple S is not in R

Q : Find the names of sailors who have reserved some boat. –

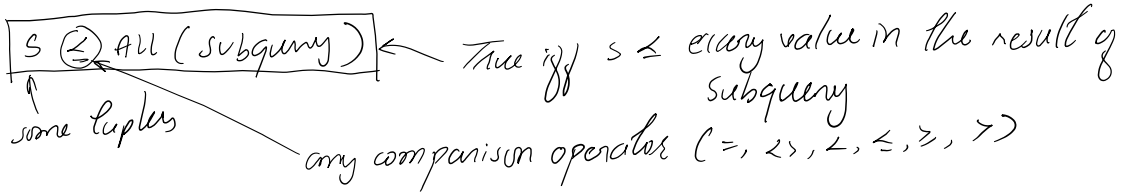
```
SELECT  S.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
                FROM Reserves R)
```

- **EXISTS** – True iff subquery result is non-empty. NOT EXISTS – True iff subquery result is empty.

```
SELECT  S.sname
FROM Sailors S
WHERE S.sid EXISTS (SELECT COUNT(*)
                    FROM R)
```

if R is empty set then count(*) = 0
Thus inner query is not empty

- **UNIQUE** – True iff no duplicates in the result of subquery. NOT UNIQUE – some duplicates tuples should be there in the result of subquery.
- **ALL, ANY/SOME** – In SQL, Any ≡ Some.



ALL this is same as ∀ meaning if subquery returns empty set then value will be true.

ANY/SOME is same as ∃ meaning if subquery returns empty set then value will be false.

NOTE : SQL, RA, RC does not permit attributes to be repeated in the same relation. And if present then it is ambiguity.

```
-- Syntax For Creating A View

CREATE VIEW ViewName
AS
SELECT Column1, Column2, Column3 ...
FROM TableName
WHERE Condition;
```

2.3.5) RANGE QUERIES :

SELECT * **FROM** emp **WHERE** salary **BETWEEN** 3000 **AND** 15000

3. ER MODEL

//Lecture 1

Database design process :

We need to convert ER (**E**ntity **R**elational) model to Relational model because there is no database software which uses ER model.

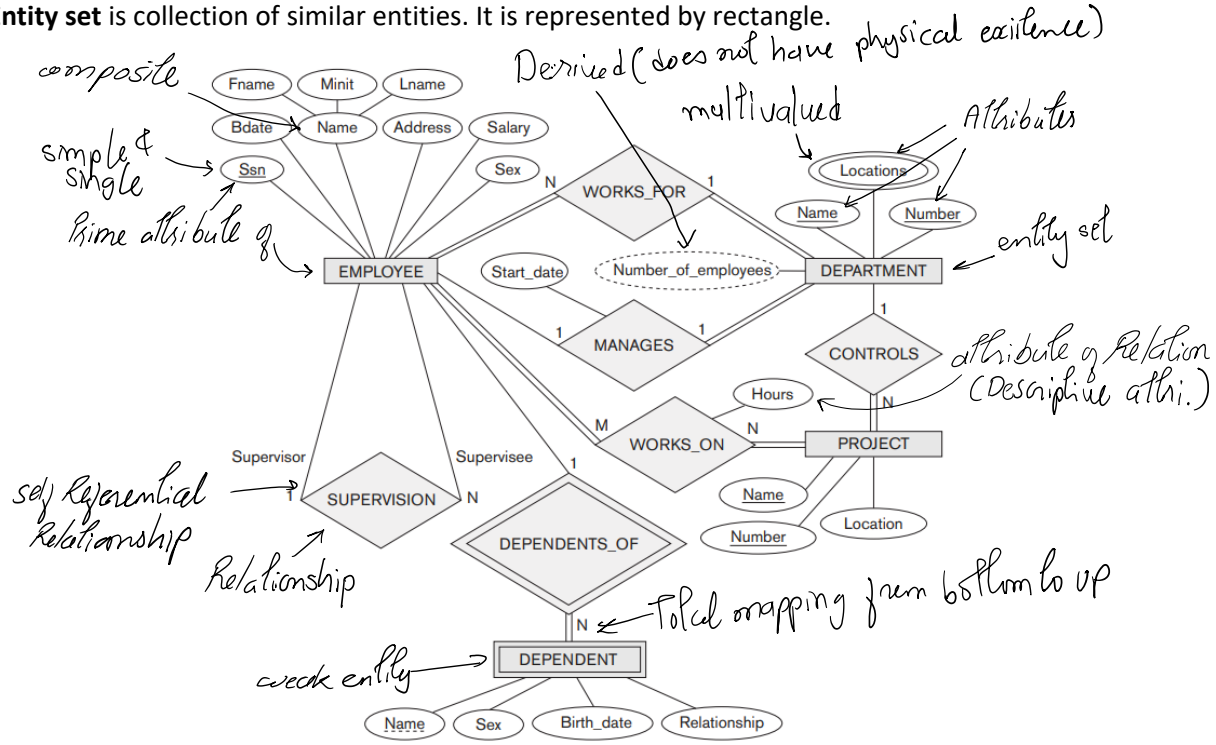
Requirements Analysis ⟩ Conceptual Design ⟩ Logical, Physical, Security ,etc. ⟩

Interaction with user english — ER model —convert→ Relational model

3.1) ENTITY TYPES, ENTITY SETS, ATTRIBUTES, AND KEYS :

Entity : which is a thing or object in the real world with an independent existence. Each entity has set of **attributes**—the particular properties that describe it. For example, an EMPLOYEE entity may be described by the employee's name, age, address, salary, and job.

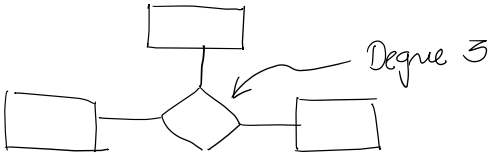
Entity set is collection of similar entities. It is represented by rectangle.



Relational Model	ER Model
1) Data structure : Relation	1) Data structure : Entity set Relationship set
2) Data manipulation languages (to access information) - Relational Algebra - Relational calculus	2) No language (agreed upon) to access data
3) Entity	3) Tuple / Row / Record
4) Entity type / set	4) Relation / Table

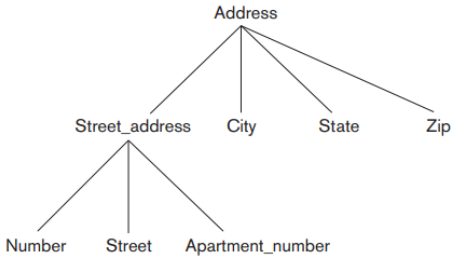
Relationship : Association between entities. Relationship set is set of all similar relationship between same attributes.

Degree of relationship set : relationship between how many entities, is called *degree of relationship set.*



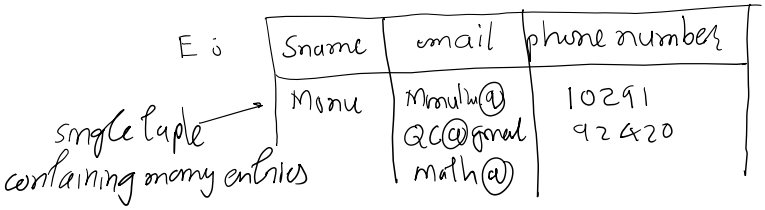
3.1.1) TYPES OF ATTRIBUTES IN ER MODEL :

SIMPLE (ATOMIC) Vs COMPOSITE : Attributes that are not divisible are called simple or atomic attributes. Composite attributes can form a hierarchy; for example, Street_address can be further subdivided into three simple component attributes: Number, Street, and Apartment_number, as shown in Figure.



SINGLE-VALUED Vs MULTIVALUED ATTRIBUTE : Most attributes have a single value for a particular entity; such attributes are called single-valued. For example, Age is a single-valued attribute of a person.

One person may not have any college degrees, another person may have one, and a third person may have two or more degrees; therefore, different people can have different numbers of values for the College_degrees attribute. Such attributes are called multivalued.



STORED Vs DERIVED ATTRIBUTES : For a particular person entity, the value of Age can be determined from the current (today's) date and the value of that person's Birth_date. The Age attribute is hence called a *derived attribute* and is said to be *derivable from* the Birth_date attribute, which is called a *stored attribute*.

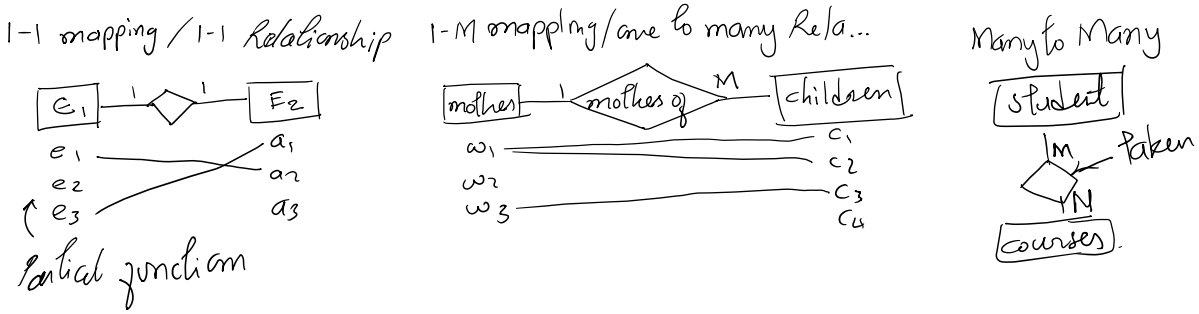
Multivalued + composite = complex attribute

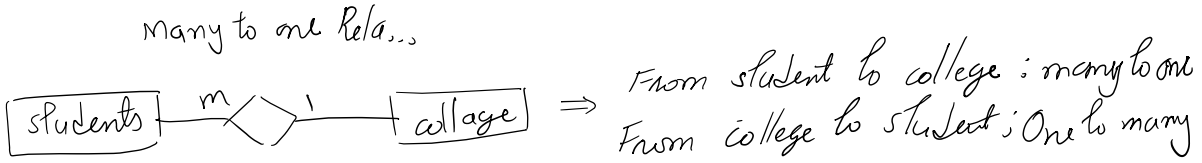
3.1.2) TYPES OF RELATIONSHIPS :

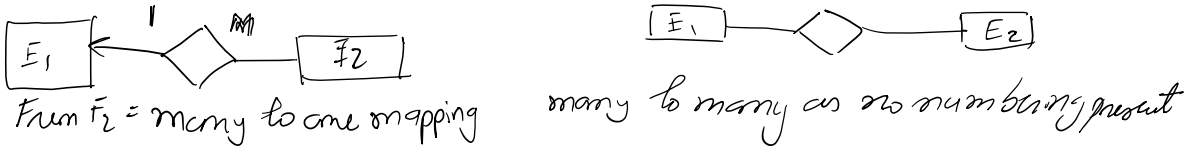
⋆ **Relationship strength** :

A **weak** or **non-identifying** relationship exists between two entities when the primary key of one of the related entities does not contain a primary key component of the other related entities.

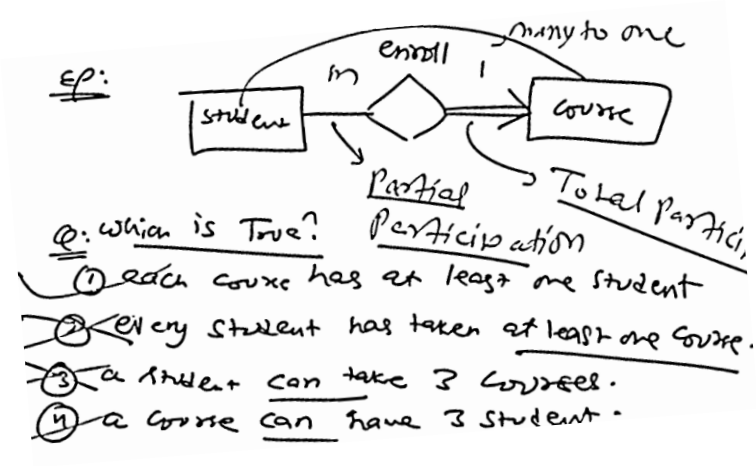
A **strong** or **identifying** relationship is when the primary key of the related entity contains the primary key of the "parent".







//Lecture 2



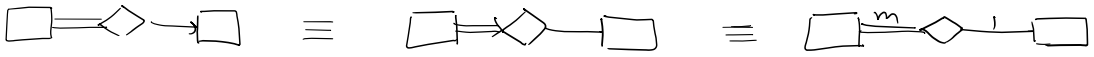
NOTE :

Total participation : 

Partial participation : 

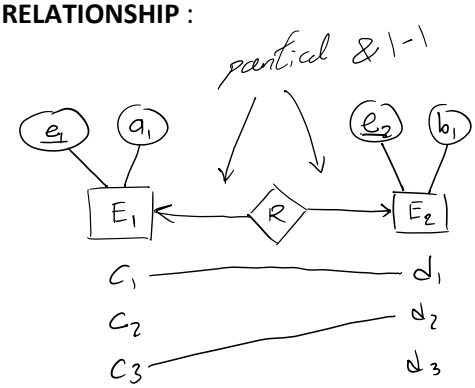
One : 

Many : 



3.1.3) KEY OF RELATION : (NULLs are allowed)

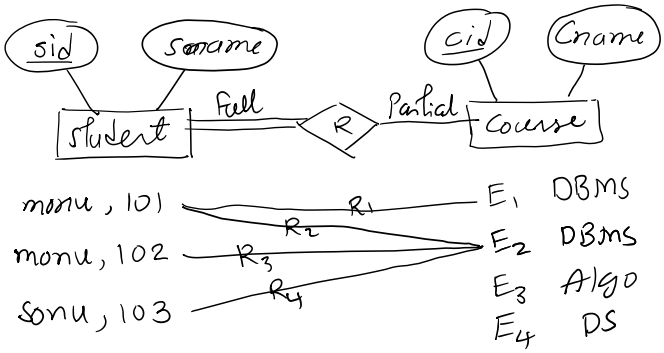
1-1 RELATIONSHIP :



Two relationship : (c_1, d_1)
(c_2, d_3)

Primary key of R : e_1 or e_2
because if is 1 to 1 so no attributes of E_1 can repeat & so for E_2

M-N RELATIONSHIP :



4 Relationship : $R_1(101, E_1)$
$R_2(101, E_2)$
$R_3(102, E_2)$
$R_4(103, E_2)$

Primary key of R : e_1 & e_2
because cannot uniquely identify

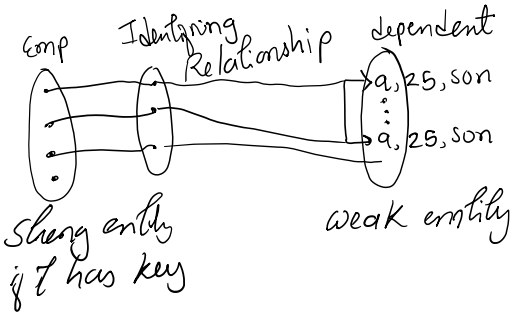
1-M RELATIONSHIP : R(c1, r1), R(c1, r2), R(c2, r3) thus e2 must be primary key of R as it is not repeating.

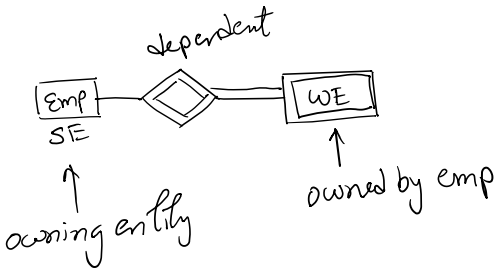
M-1 RELATIONSHIP : Primary key of R : e1.

3.1.4) WEAK ENTITY SET :

Strong entity set are those entity set which has its own key.

Entity set without any key is called weak entity set. What is the need of this entity set ? -

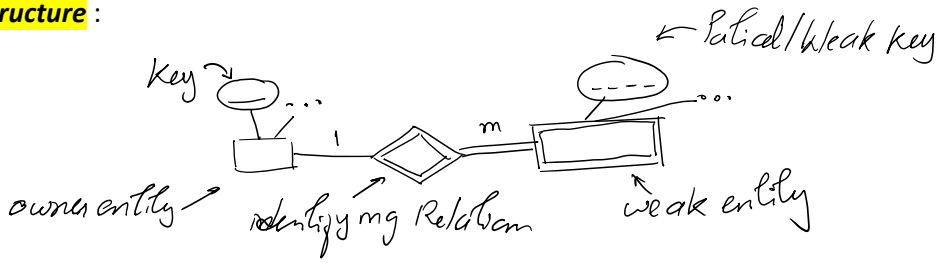




Weak entity set does not have any key and its existence depends on existence of owner entity

And the relation here is called **identifying/Strong relation** as without this relation it is not possible to identify weak entity.

Common weak entity structure :



Partial ≡ Discriminator.

Here, *primary key of identifying relationship* : (PK of owner ES, Partial key of WES)

//Lecture 3

Note that every entity must be uniquely identifiable. Reason being by the definition of entity (a distinguishable things or object). There are two ways

- In case of strong entity set : attributes in the same ES
- In case of weak entity set : Identifying relationship with other identifying/owner entity set

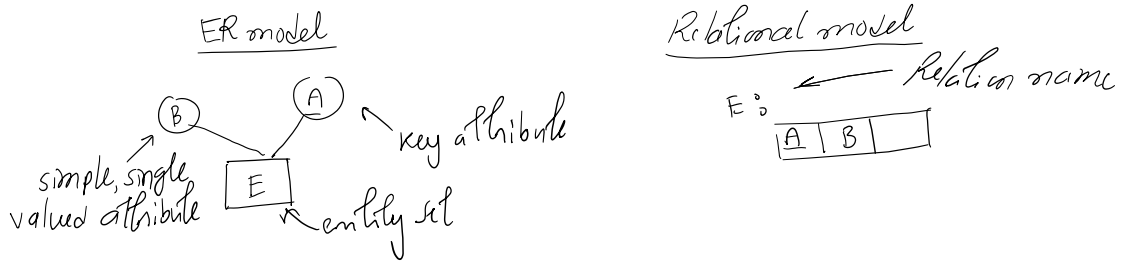
NOTE : In ER model, the following concepts not present : Functional dependencies and foreign key.

Relationship set (represented as diamond) is not an entity set; it is association between entity sets. Thus, it cannot contain attributes of other entity set. But relationship set can have its own attributes which are describing its property.

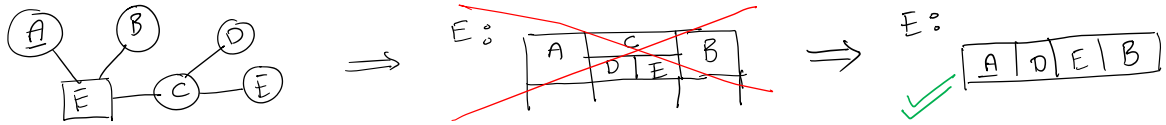
3.2) ER MODEL TO RELATIONAL MODEL :

While converting ER model to relational model we should keep in mind few things :

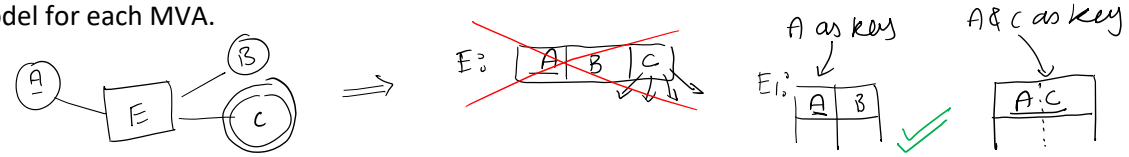
- Good design : 3NF, no null problem
- Minimum no. of tables



In case of composition attributes => There is no composite attributes in relational model.

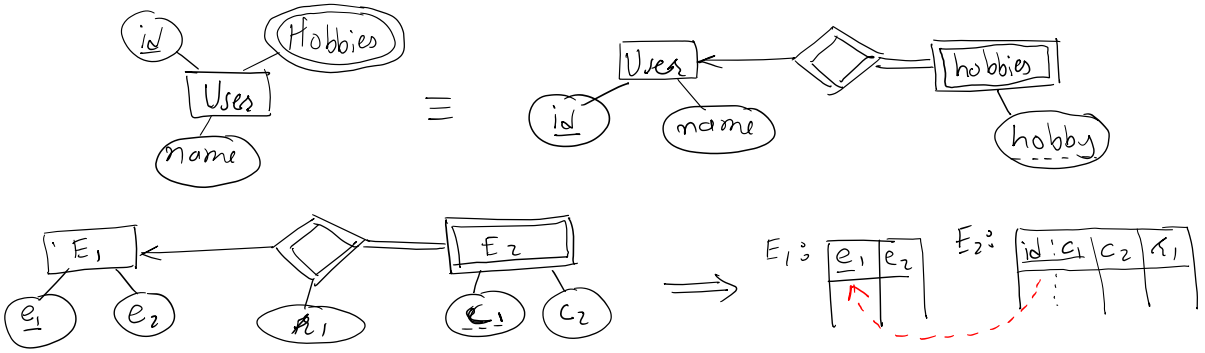


Q : How to handle multivalued attribute ? – for each MVA, we create a separate relation in relational model for each MVA.



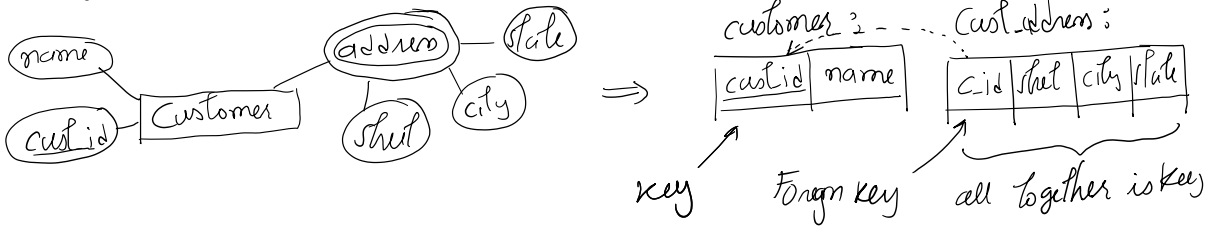
Every MVA can be converted into a weak ES

For example,



If owner entity (i.e. user) will gone then weak entity (hobbies) will be gone.

For ***complex attribute***,

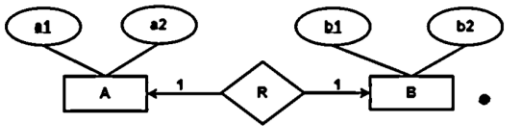


⋆ **Specialization** is the process of classifying a class of objects into more specialized subclasses. **Generalization** is the inverse process of generalizing several classes into a higher-level abstract class that includes the objects in all these classes. We call the relationship between a subclass and its superclass an **IS-A-SUBCLASS-OF** relationship, or simply an **IS-A** relationship. Denoted by triangle.

3.2.1) Mapping relationship :

First, we will explore binary relationship,

1-1 MAPPING :



In ***One-to-One relationship***, **two tables will be required. Either combine 'R' with 'A' or 'B'**

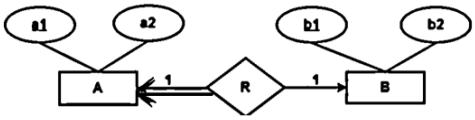
Way-01:

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

Way-02:

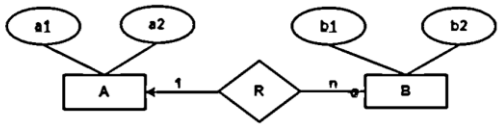
1. A (a1 , a2)
2. BR (b1 , b2, a1)

best design : AR (a1, a2, b1) (FK: b1)
B (b1, b2)



If both sides total participation is there then we need only one table because there is no problem of NULL to be appear on any side. so, ARB(a1, a2, b1, b2) or ARB(a1, a2, b1, b2).

1-M RELATIONSHIP :



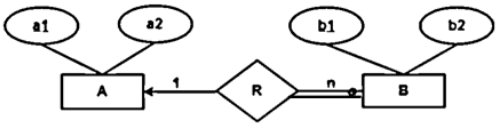
In ***One-to-Many relationship***, two tables will be required-

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

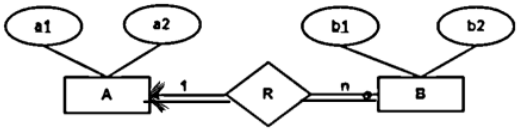
↑ when null values are allowed then this is correct

A(a1, a2)
R(a1, b1)
B(b1, b2)
} best design (No Null)

best design : RB (b1, b2, a1)
A (a1, a2)



In case of total on both sides we cannot merge all together ex. ~~ARB(a1, a2, b1, b2)~~



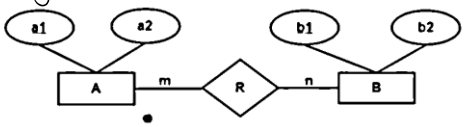
best design :
$E_1(\underline{a_1}, a_2)$
$E_2R(\underline{b_1}, b_2, a_1)$ ← FK

If we merge all together then b1 will only be key of whole relation and $a1 \rightarrow a2$ can happen which violets 3NF.

$b1 \longrightarrow a_1 \longrightarrow a_2$ (✗)

M-N RELATIONSHIP :

a_1b_1 is primary key
i) $ARB(\overline{a_1, b_1}, b_2, a_2)$
$a_1 \rightarrow a_2$ & $b_1 \rightarrow b_2$
So not in 2NF



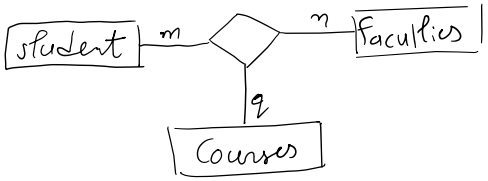
$AR(\overline{a_1, b_1}, a_2)$
$a_1 \rightarrow a_2$
Not in 2NF

In ***Many-to-Many relationship***, three tables will be required-

1. A (a1 , a2)
2. R (a1 , b1)
3. B (b1 , b2)

Whatever be the mapping (whether total or partial) one table per entity and relation must require.

3.2.2) Non-binary relationship :



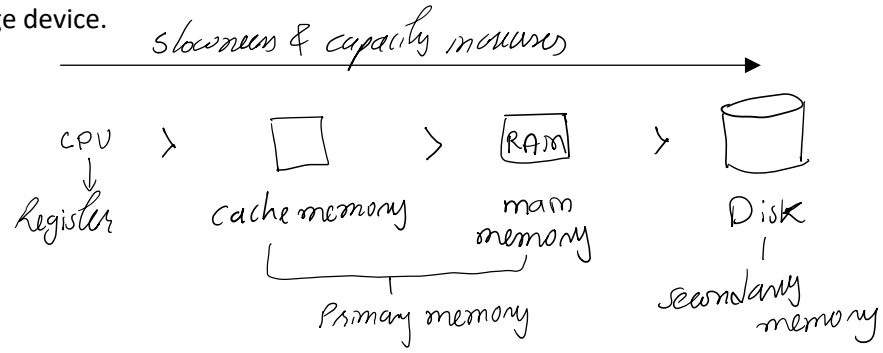
any combination from student, courses to faculties is possible also, courses faculties to student.

enroll : | sid | fid | cid | ← all together is key

4. MAGNETIC DISK

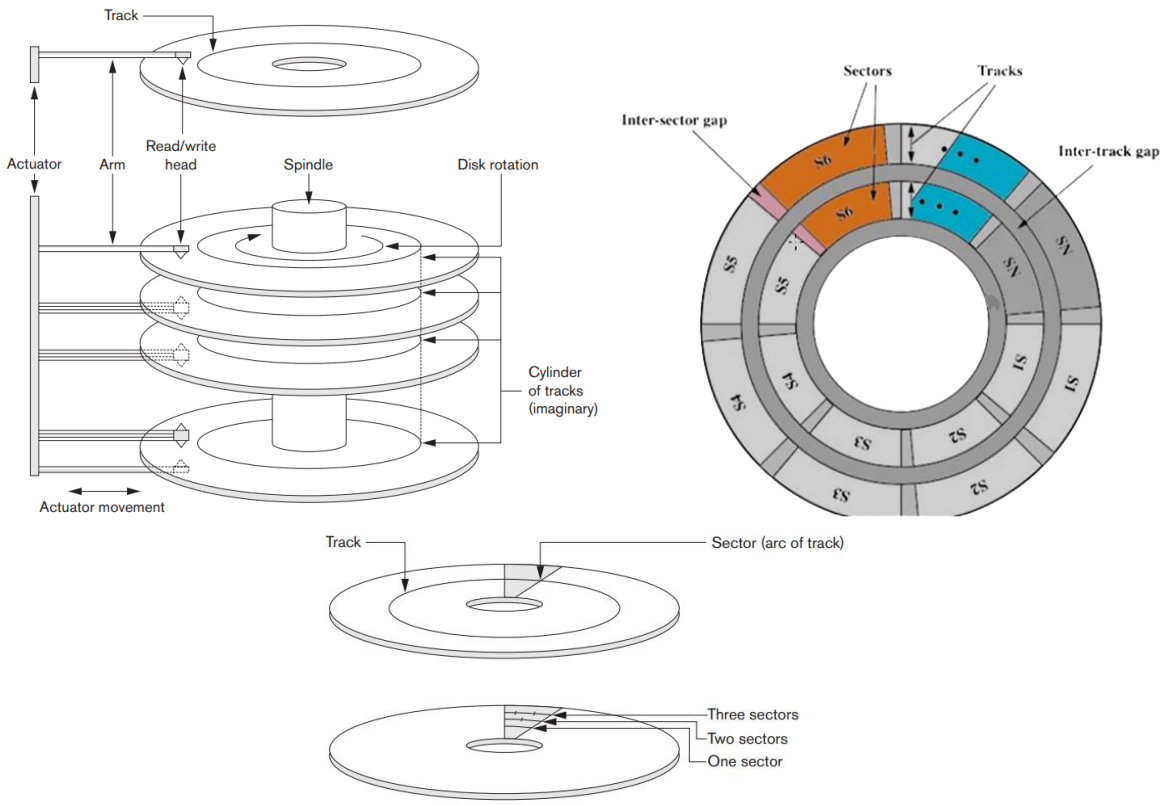
//Mod4-Module1

Although a database system provides a high-level view of data, ultimately data have to be stored as bits on a storage device.



4.1) DISK PHYSICAL STRUCTURE :

Block : It is collection of sectors. Note that it is not a property of disk but it is property of a DBMS software. DBMS decides how many consecutive sectors to be considered as one sectors.



All R/W head move together and vertically in same position. And only one R/W head is active at any time. Usually, head movement is slower than disk rotation.

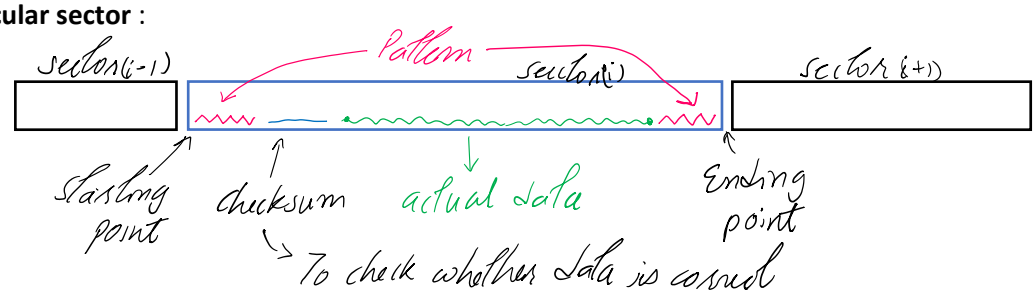
To Read data,

Step 1 : Put R/W head on desired track (this is also called **seek time**).

Step 2 : Rotate starting point of desired sector under R/W head (this is called **rotational latency**).

Step 3 : Rotate complete desired sector under R/W head (this is called **transfer time**).

A particular sector :



The main measure of the qualities of a disk are

- **Capacity** : how much data can we store on a disk

Same as # of tracks

$$Capacity = \#Recoding\ surface \times \#\frac{Tracks}{surface} \times \#\frac{sectors}{tracks} \times Sector\ size$$

- **Access time** : It is the time from when a read or write request is issued to when data transfer begins.
- **Data-transfer rate** : How much data(Bytes) per second can we read/write.
- **Reliability**.

Q : 1000 Cylinders, 10 sectors/track, head assembly at cylinder 0 initially, head moves 10 mics/cylinder, disk rotates 100 times/second, Controller bus perfect, no previous request. What is the average time to read a randomly chosen byte is ? –

Average seek time = $\frac{0+1000}{2} \times \frac{10\mu s}{cyln} = 5ms$

Average rotational latency = $\frac{0+1}{2} \times 10ms = 5ms$

{ 100 times → 1 second
1 time → $\frac{1}{100}$ second = 10 ms

Transfer time (one sector) = $\frac{1}{10} \times 10ms = 1ms$

{ 1 track needs 10 ms
1 sector → $\frac{track}{10}$ → $\frac{10\ ms}{10}$

No of track = Recording width / inner space between track.

Recording width = (Outer Diameter - Inner Diameter) / 2

//Lecture 2

4.1.1) ORDER OF CLOSENESS :

The closest that two records can be on a disk to be on the same sector (because we need only one seek and one rotation)

In decreasing order of closeness :

They could be on the same block >> they could be on the same track >> the same cylinder >> an adjacent cylinder.

As we have natural notion of “closeness” for sectors, we can extend to a notion of next and previous sector.

Q : How data is stored in disk ? –

- It is first stored on the first sector of the first surface of the first cylinder. Then in the next sector, and next until all the sectors on the first track are exhausted.

- Then it moves on to the first sector of the second surface (remains at the same cylinder), then next sector and so on. It exhausts all available surfaces for the first cylinder in this way.
- After that, it moves on to repeat the process for the next cylinder.

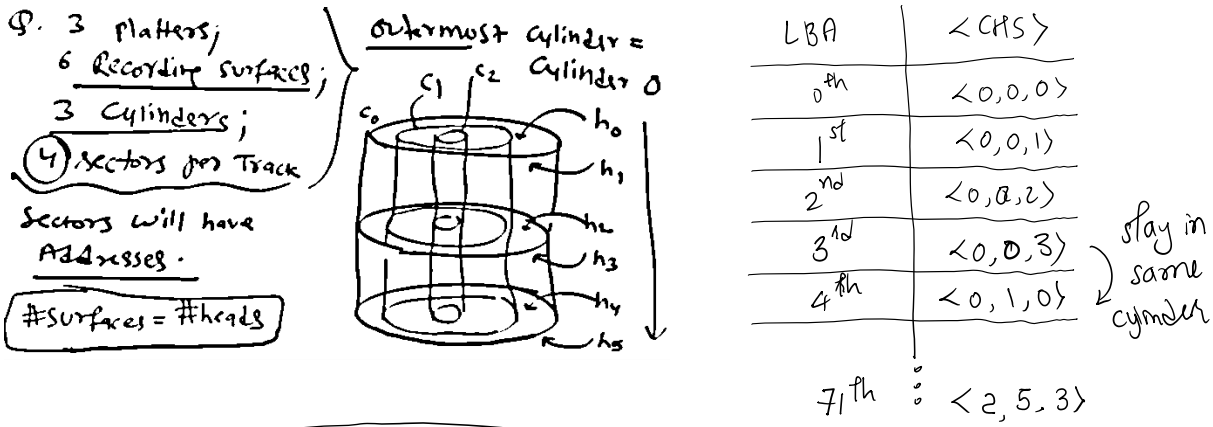
4.1.2) SECTOR ADDRESSING :

<C, H, S> ← surface No
simple integer number

There are two types of addressing : **Cylinder-head-sector (CHS)** and **Logical block addressing (LBA)**

NOTE : by default, track number start at 0, and track 0 is the outermost track of the disk. The highest numbered track is next to the spindle

Sectors are the smallest unit that we can access so, sectors are given addresses, meaning each sector has its own address.



CHS to LBA : $<2, 3, 2> = 2 \times (6 \times 4) + 3 \times 4 + 2 = 62$

LBA to CHS : 35th sector

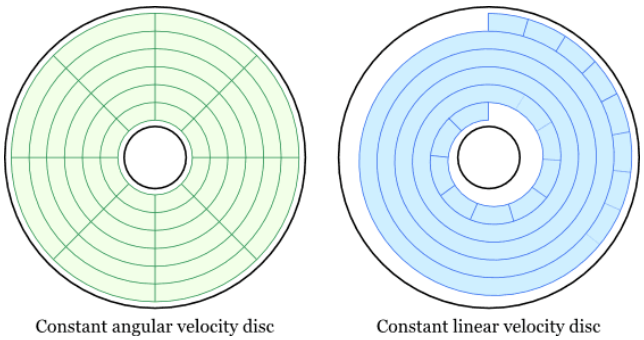
1 cylinder → 24 sectors
1 surface → 4 sectors
$\Rightarrow \left\lfloor \frac{35}{24} \right\rfloor = 1 = C$

$\left\lfloor \frac{11}{4} \right\rfloor = 2$ ← 35 − 1×24
11 − 2×4 → <1, 2, 3>

In <C, H, S> each number represents how many cylinders, surfaces, sectors are gone respectively (because indexing starts from 0).

For magnetic disks, reads and writes are equally fast

⋆ **EFFECT OF CLV & CAV on DISC** :



Constant angular velocity disc | Constant linear velocity disc

With **Constant linear velocity (CLV)**, the density of bits is uniform from cylinder to cylinder, because there are more sectors in outer cylinders, the disk spins slower when reading those cylinders, causing the rate of bits passing under the read write head to remain constant. This is the approach used by modern CDs and DVDs. Here ***outer track data capacity = k x inner data capacity where k is proportional to radius of track.***

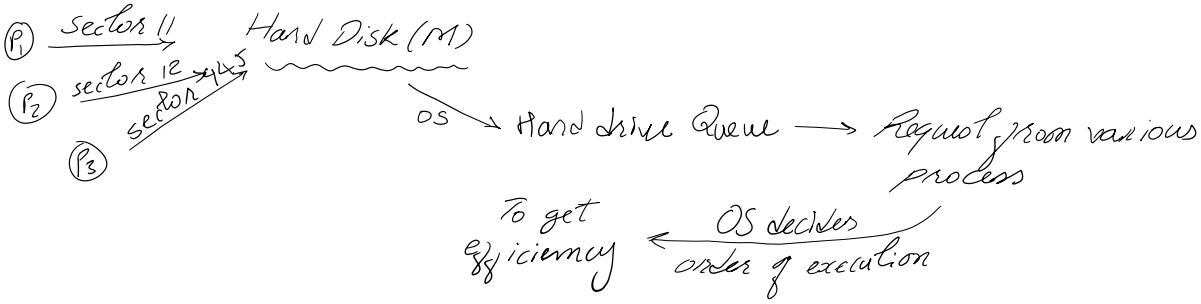
With **Constant Angular Velocity (CAV)**, the disk rotates at a constant angular speed, with the bit density decreasing on outer cylinders (these disks would have a constant number of sectors per track on all cylinders). Here ***inner track data capacity = outer track data capacity***

//Lecture 3

4.2) DISK SCHEDULING ALGORITHMS :

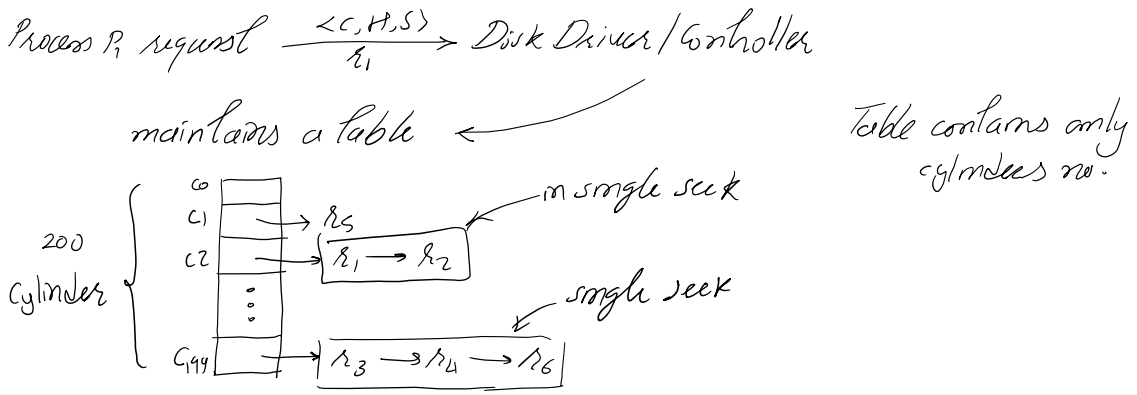
We now describe disk-scheduling algorithms, which schedule the order of disk I/O to maximize performance.

Now, we know nowadays we have multiprogramming OS meaning there can be many processes in ready queue. And some process request to perform some I/O operation on some I/O devices. And it is responsibility of OS to manage resources (this includes I/O) efficiently.



Q : What matters most to improve performance (fast performance) ? – total seek time and total Rotational latency (this also includes actual data transfer time as data transfer requires rotation to read or write).

We can see that **seek time >>> rotational latency** (in most cases) thus, aim of disk scheduling is to minimize the overall seek time. *Seek time is affected by no. of cylinder* thus all disk scheduling algo only cares about cylinder number of the disk requests.



No scheduling algo is optimal.

FCFS scheduling : (not preferred)

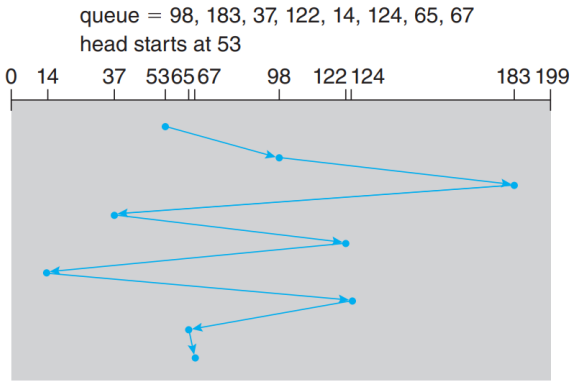


Figure 11.6 FCFS disk scheduling.

SSTF (shortest-seek-time-first algorithm) :

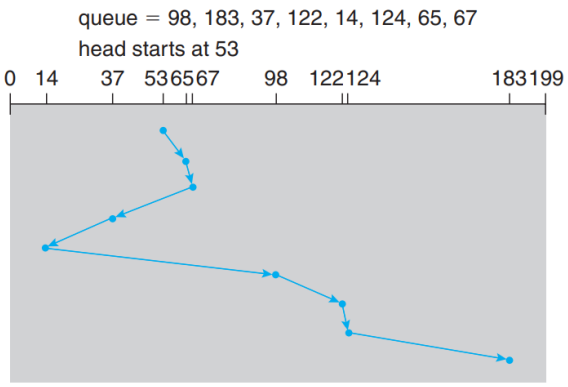


Figure 10.5 SSTF disk scheduling.

SCAN scheduling : elevator

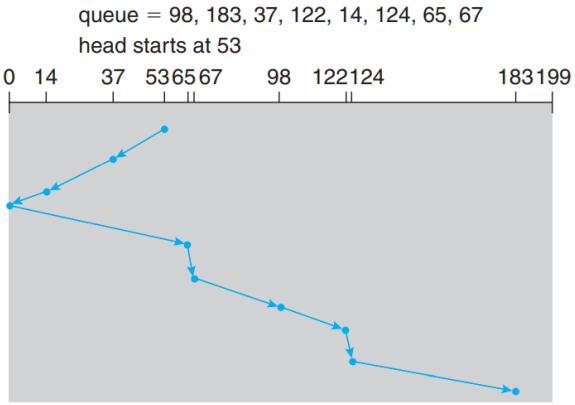


Figure 10.6 SCAN disk scheduling.

C-SCAN scheduling : maintain same direction

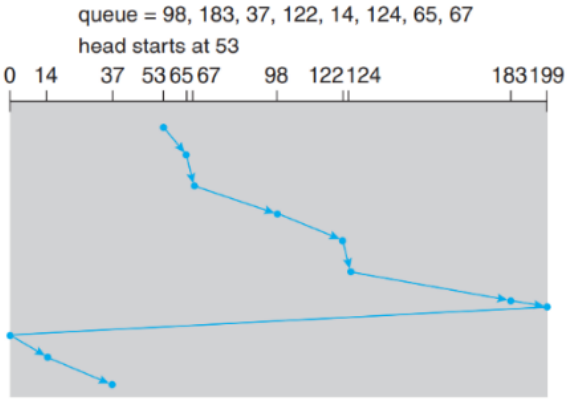
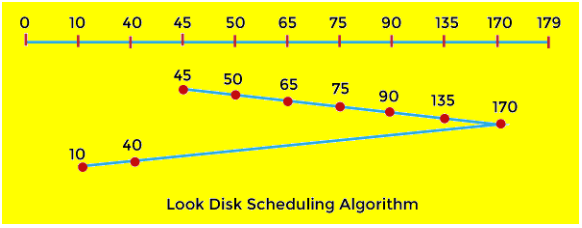


Figure 10.7 C-SCAN disk scheduling.

LOOK scheduling : (improved SCAN)



C-LOOK scheduling : (improved C-SCAN)

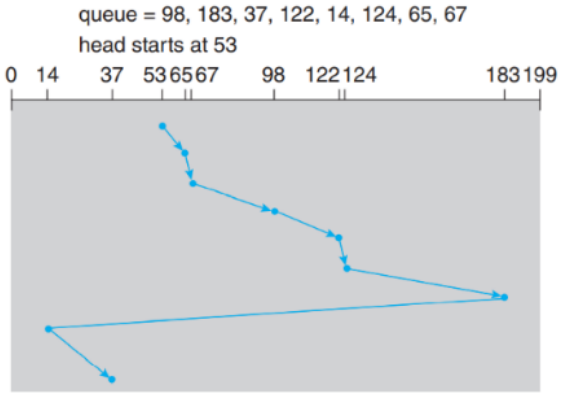


Figure 10.8 C-LOOK disk scheduling.

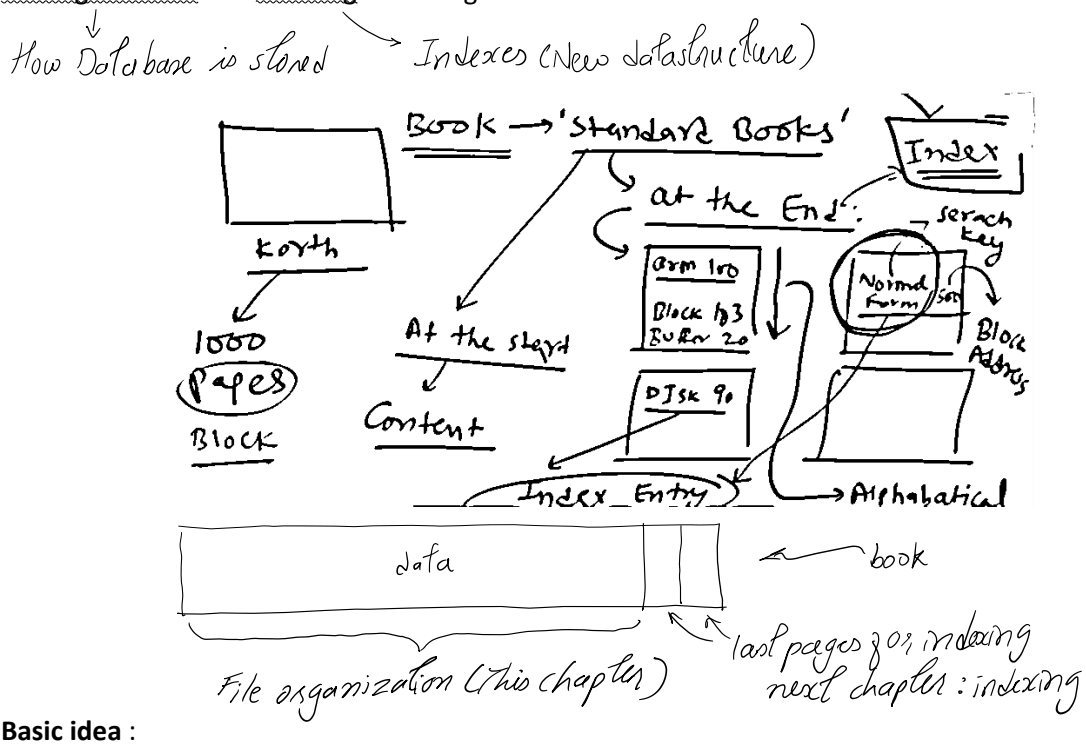
All doubt clear problem : https://gateoverflow.in/3846

5. FILE ORGANIZATION & INDEXING

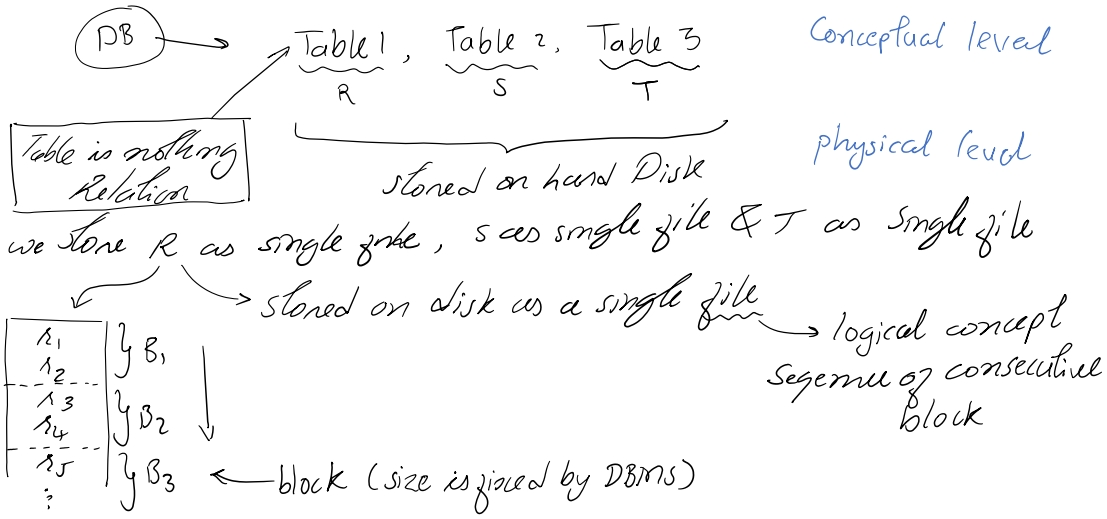
//Lecture 1

5.1) FILE ORGANIZATION :

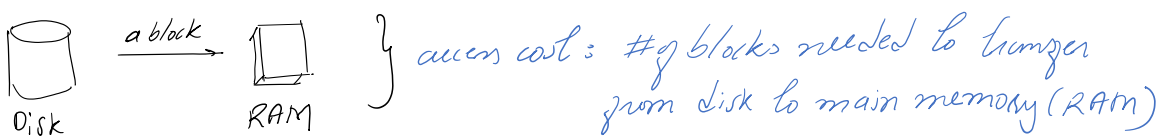
File organization and indexing two things are different.



Basic idea :



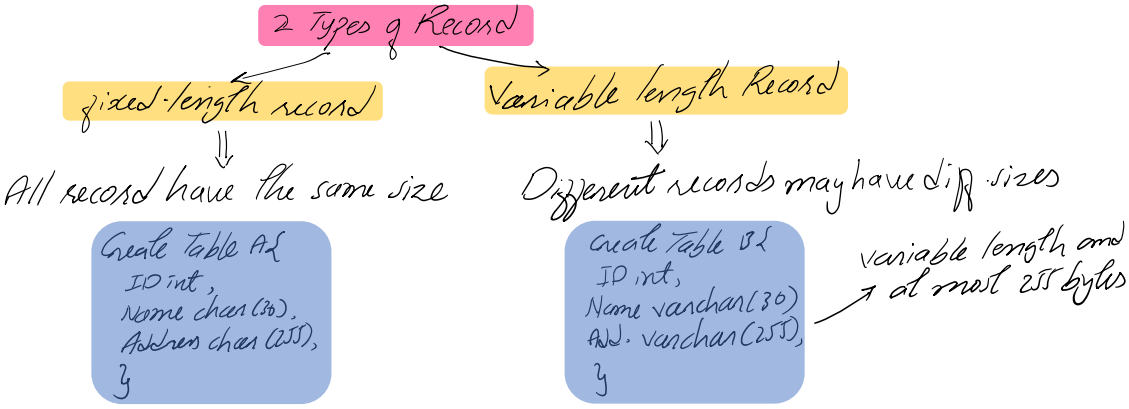
Whenever a certain portion of the data is needed, it must be located on disk, copied to main memory for processing, and then rewritten to the disk if the data is changed.



NOTE : block is also called page in DBMS. This is not same as the page in paging. But we use block.

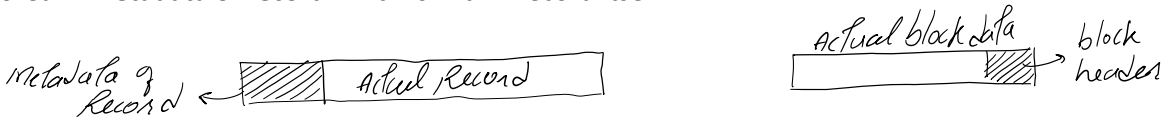
BLOCK : *Each file is also logically partitioned into fixed-length storage units called blocks*, which are the units of both storage allocation and data transfer. A block may contain several records. No record is larger than a block. No block size is larger than a track size.

5.1.1) TYPES OF RECORDS :



Q : How to find a particular field value inside a record ? –

Before answering we explore record property (time stamp of record, size of record, field value, etc) and how and where these properties are store. Time stamp and all properties related to record are stored in **metadata of record** which is within record itself.



And Field value are properties of file so they are stored at in **block header**.

For size of field, if size of field is of variable size the it is stored in metadata of record. And if fixed length records then we will store in block header.

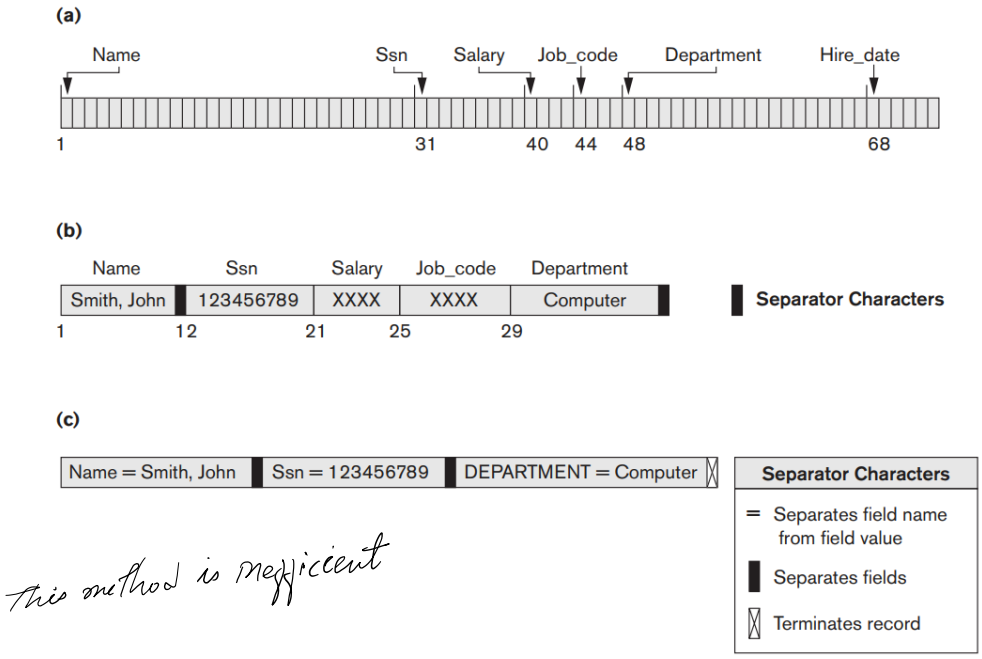


Figure 16.5
Three record storage formats. (a) A fixed-length record with six fields and size of 71 bytes. (b) A record with two variable-length fields and three fixed-length fields. (c) A variable-field record with three types of separator characters.

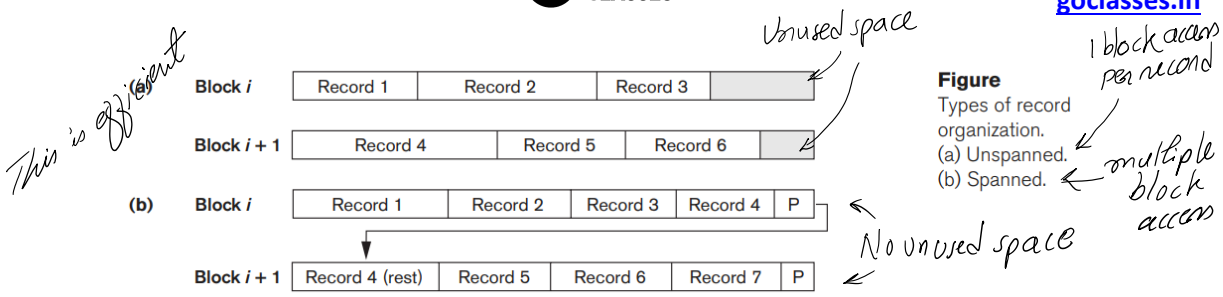


Figure Types of record organization. (a) Unspanned. (b) Spanned.

Blocking factor : No. of record that can be fitted into per block. If B is block size and R is record size then, if fixed length record then *Bfr* = $\left\lfloor \frac{B}{R} \right\rfloor$, In variable length *Bfr* represents average number of records per block for the file. Thus, *Bfr* = $\frac{B}{R}$.

NOTE : Unspanned Organization is generally used with fixed-length records because it makes each record start at known location in the block, simplifying record processing. For variable-length records, either a spanned or an unspanned organization can be used.

Thus, Unspanned organization is by default taken. And fixed-length records are by default considered.

Q : For a variable-length records, different records have different sizes. Let the average record size in a file be R, and number of records in the file be r. Assume that block size is B bytes, and K bytes are used for block header. Using spanned organization, find the number of blocks needed to store this file ? – As variable length records are given meaning two block can contain same record info. Total no. of records = r. meaning if we somehow find record in per block. Then we can take $\lceil r/something \rceil$ = # of block.

This is what we need !

Note that bfr in variable length records gives average no. of record per block.

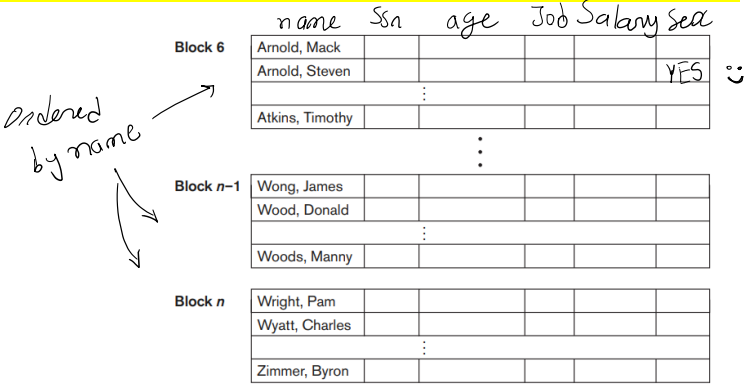
Thus, bfr = $\frac{B-K}{R}$; # of blocks needed to store file = $\left\lceil \frac{r}{bfr} \right\rceil$.

5.1.2) ORGANIZATION OF RECORDS IN FILES :

Determine how the file records are physically placed on the disk.

worst

1) **Heap files (files of unordered records)** : records are placed in the file in the order in which they are inserted, so new records are inserted at the end of the file.
2) **Sequential or sorted files** : Put ordered based on some field (some field means some attributes like sid or pid something). This field is called **ordering field**.



Average Access Times for a File of b Blocks under Basic File Organizations

Type of Organization	Access/Search Method	Average Blocks to Access a Specific Record
Heap (unordered)	Sequential scan (linear search)	$b/2$
Ordered	Sequential scan	$b/2$
Ordered	Binary search	$\log_2 b$

Block address : Address of the first sector of that block. <C, H, S> address OR LBA address.

Record address : a record can be identified by giving its block address and the offset.

//Lecture 3

5.1.3) FREE DISK SPACE MANAGEMENT :

We know that block size is decided by DBMS, meaning us

Having a large block size means that even a 1-byte file, ties up an entire cylinder. On the other hand, a small block size means that most files will span multiple blocks and thus need multiple seeks and rotational delay.

To keep track of free disk space, the system maintains a free-space list. The **free-space** list records all free disk blocks-those not allocated to any file.

We can keep track of free blocks by two methods :

BITMAP (BIT VECTOR) METHOD : 1 → free , 0 → occupied

For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be

00111100111111000110000001110000... ← we keep this block in main memory

LINKED LIST METHOD (FREE LIST) :

Way 1 : within free block we maintain one pointer pointing to next free block and so on. And we keep one pointer to first free block in RAM. But this is stupid as we are storing pointer in free block, they are not free space anymore.

Way 2 : Store addresses of n free blocks in the first free block. The first n – 1 of these blocks are actually free. The last block contains the addresses of another n free blocks, and so on.

Q : when does the linked- list scheme require fewer blocks than the bitmap ? – When no. of free block is few OR in other words when disk is nearly full then liked list scheme require fewer blocks than the bitmap.

NOTE : one thing to realize that linked list is actually does not wasting space. Free blocks are used to hold the free list (address), so the storage is essentially free. Thus, liked free list method doesn't eat up our useful space… (थोड़ी जगह हे तोह देदो। नहीं हे तोह कोई बात नहीं)

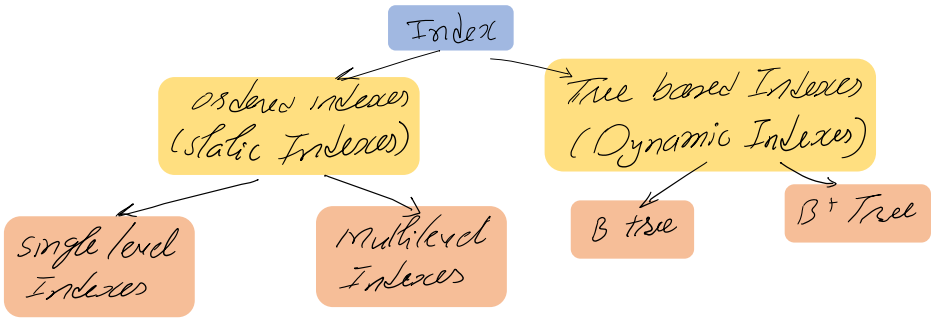
Meaning "taking space" and "wasting space" is different thing.

//Lecture 4

5.2) INDEXING :

Index is *overhead*. Entire book is data file and pages consist of only indexing is called index file. So both are different thing.

Index file = index Vs Data file = main file = indexed file or file



Index consists of a particular entry called index entry consist of

$\langle index\ field\ value\ v, block/record\ address\ of\ the\ record\ containing\ v \rangle$

Search key value V

Record address Vs Block address

Block address + offset

To gain fast random access to records in a file, we can use an index structure. Each index structure is associated with a particular search key.

A file may have several indices, on different search keys

5.2.1) SINGLE LEVEL ORDERED INDEXES :

Sorted

Note that ordered indexes means index entries are ordered by some key but file may or may not be ordered.

In GATE, by default index file is ordered.

SINGLE LEVEL ORDERED INDEXES :

//Lecture 5

- ***PRIMARY INDEX*** :

If data file (original) is sorted on a candidate key attribute A then index on A is called *primary index*.

same as "Sequentially ordered by a key field A"

Q : From DBMS point of view, index at the end of Navathe book, is primary index ? – Answer is NO, search key is topic name meaning they are sorted by (Ordered by) topic name but data file (book) is not sorted on Topic name. So, topic name is index but it is not primary index.

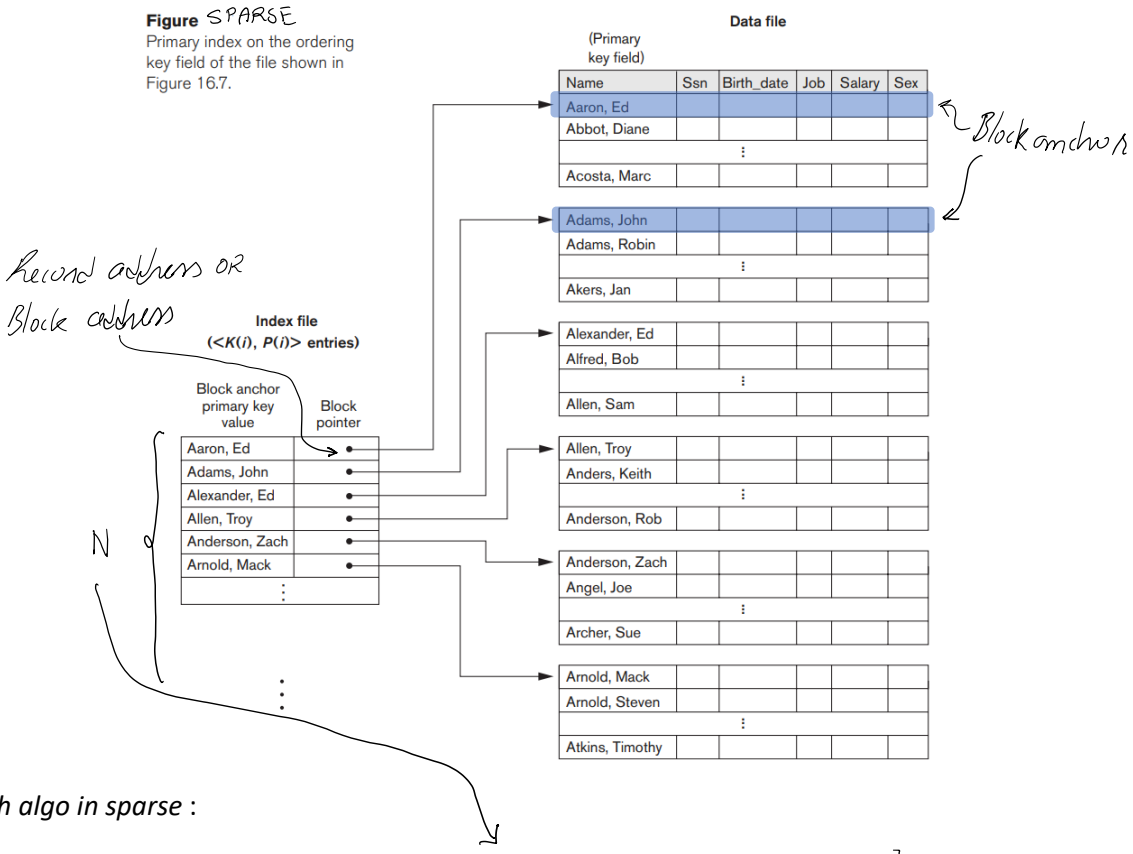
If A is key ⟹ Key field

If file is ordered (sorted) by A ⟹ Ordering field

If A is key and file is ordered (sorted) by A ⟹ Ordering key field

It has two implementations : (i) For each data record one index record ..... this is called *Dense*

(ii) For each block one index record ..... this is called *sparse*

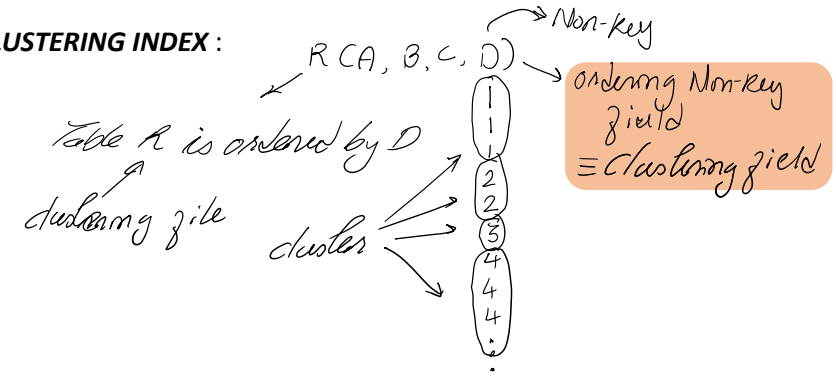


Search algo in sparse :

- Binary search on index blocks (suppose N index blocks) → $\lceil \log_2 N \rceil$ } $\lceil \log_2 N \rceil + 1$
- 1 more access for data block

//Lecture 6

- ***CLUSTERING INDEX*** :



Q : From DBMS point of view, index at the end of Navathe book, is clustering index ? – No, again same reason because data file is not sorted on search key (i.e. on topic name)

NOTE : To create clustering index on a file R : R must be physically ordered by a NON-KEY FIELD (say F), then an index on F is called clustering index.

Implementation of clustering index :

index entries = # of clusters ….. thus, this is ***sparse***

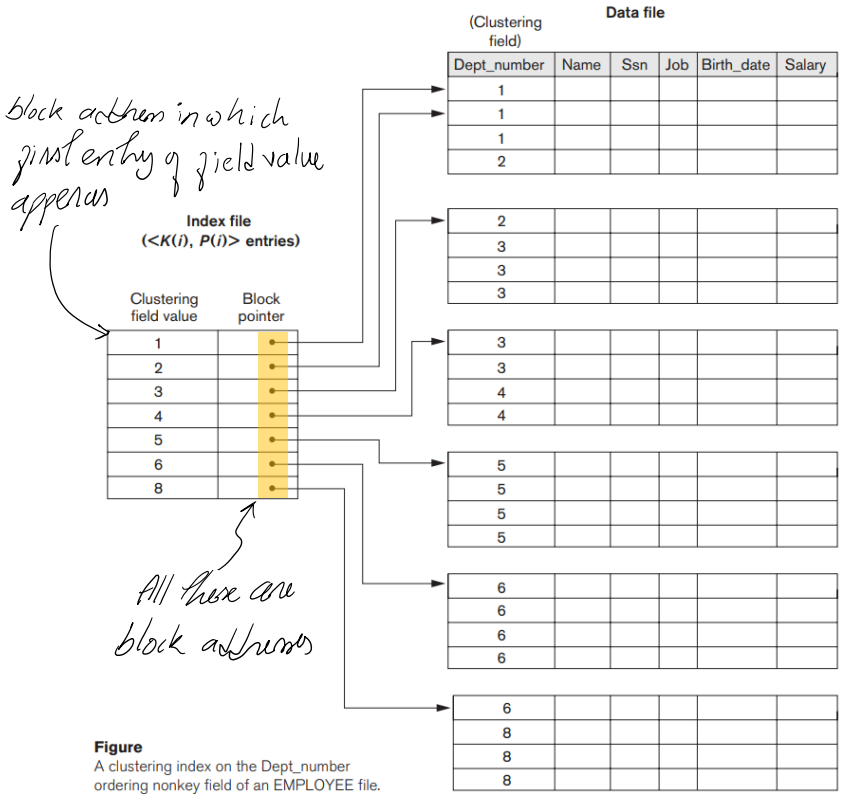
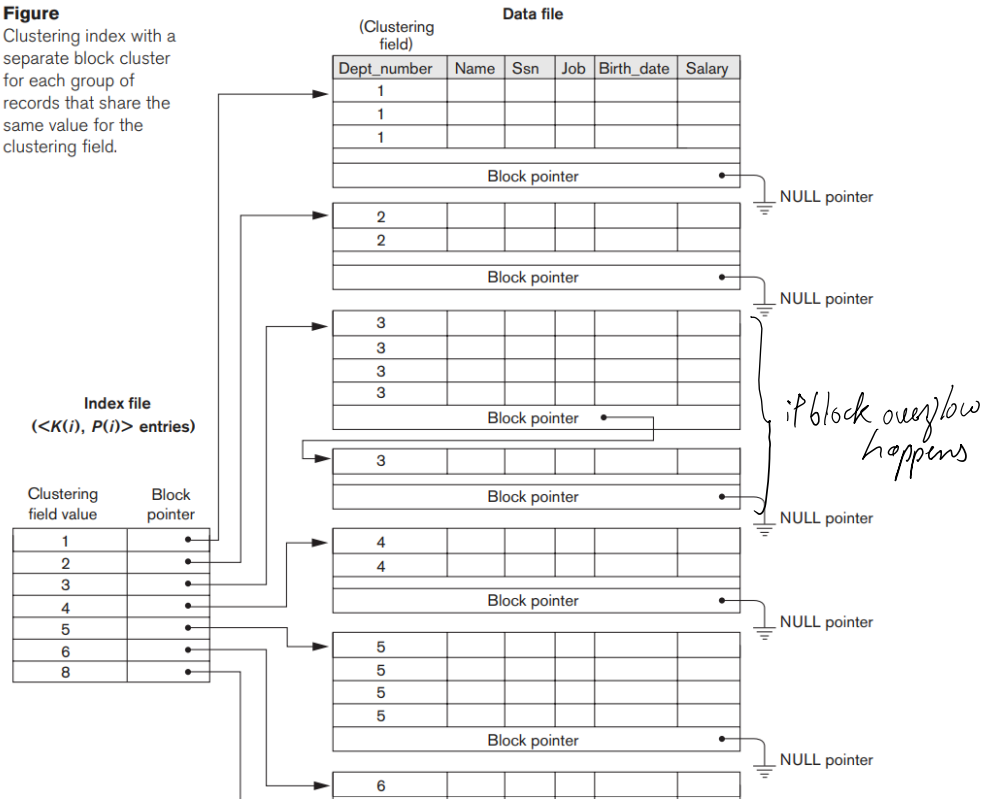


Figure
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

Figure
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.



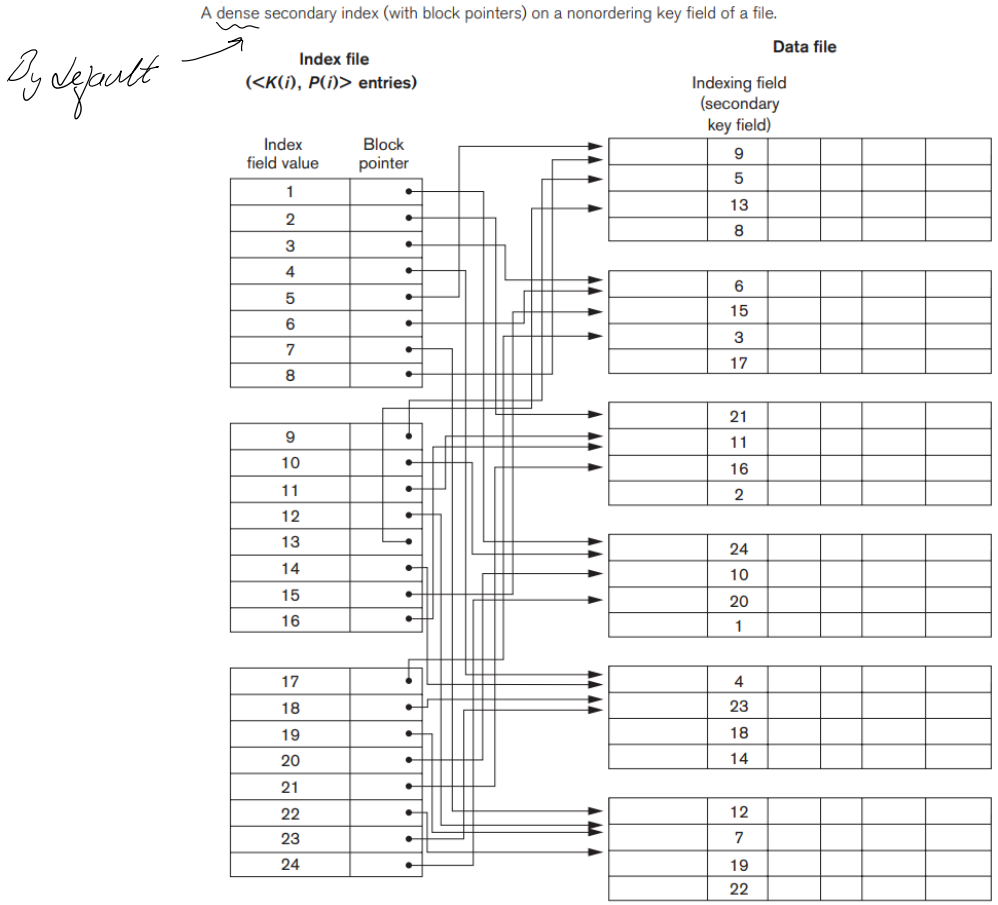
Primary index	Clustering index
Data file is sorted according to key	Data file is sorted according to non-key
Block pointer per block	Block pointer per cluster

- ***SECONDARY INDEX*** : indexing on non-ordering field (say field B). Now, if B is key then we say "secondary index on key" and if B is not key then we say "secondary index on non-key".

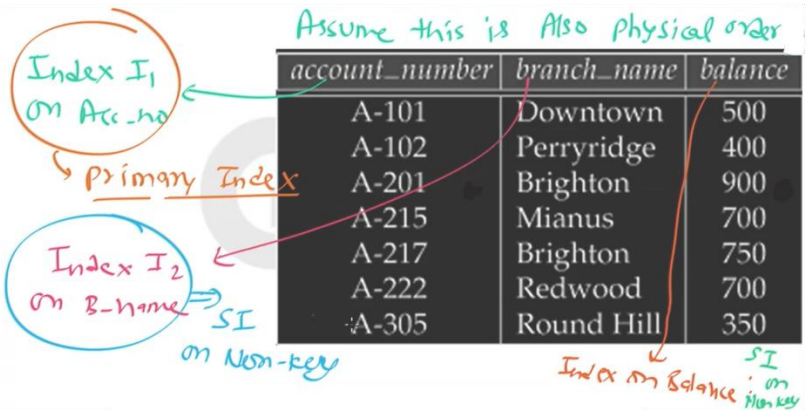
Meaning file is not ordered by index is always ordered. *In every case whether index is primary, secondary, clustering index file is always ordered.*

We know that heap file is unordered thus we can do indexing on heap using secondary index.

#entries in index = #records in data file



Data access always happens block by block



A you can see there is 700 entry so in such cases we create extra level of block. That extra level consists of only block addresses or block pointers which points to addresses of block which contains same nonkey value…

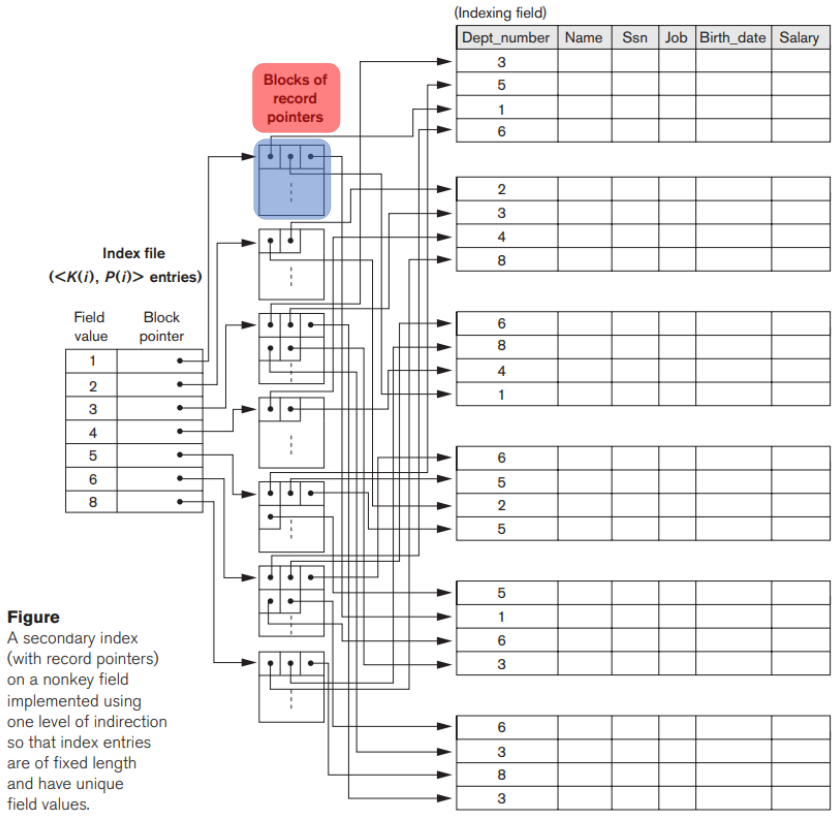


Figure
A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

We found that indexing done in "Navathe" was secondary indexing on topic name !!

- *Access cost using SI (Dense)* : $\lceil \log_2 I \rceil + 1$ (Blocks in index)
- *Access cost without SI* : can't do binary search as file is not ordered by search key so B/2 (on average) if worst case is asked then B.

In short,

Table Types of Indexes Based on the Properties of the Indexing Field

	Index Field Used for Physical Ordering of the File	Index Field Not Used for Physical Ordering of the File
Indexing field is key	Primary index	Secondary index (Key)
Indexing field is nonkey	Clustering index	Secondary index (NonKey)

NOTE : SI is useful when we want a particular record which can exists in few numbers of data blocks. Example, people with age = 50. (SI on age very good). SI is worst when we have to access to many data blocks. Example, range queries. Here worst case happens when all record satisfies queries but you are still accessing data block + index block.

//Lecture 9

Till now we saw single level indexing now we will see multilevel indices and more.

5.2.2) MULTILEVEL INDEX AND ISAM :

To reduce I/O cost : we can put index in main memory and can-do search in main memory only,

- Problem 1 : index larger than main memory
- Problem 2 : Even if index size < main memory then RAM has many things to accommodate.

But this idea of putting index in RAM is good only when index file is very small or sparse.

Thus, we introduce multilevel index…

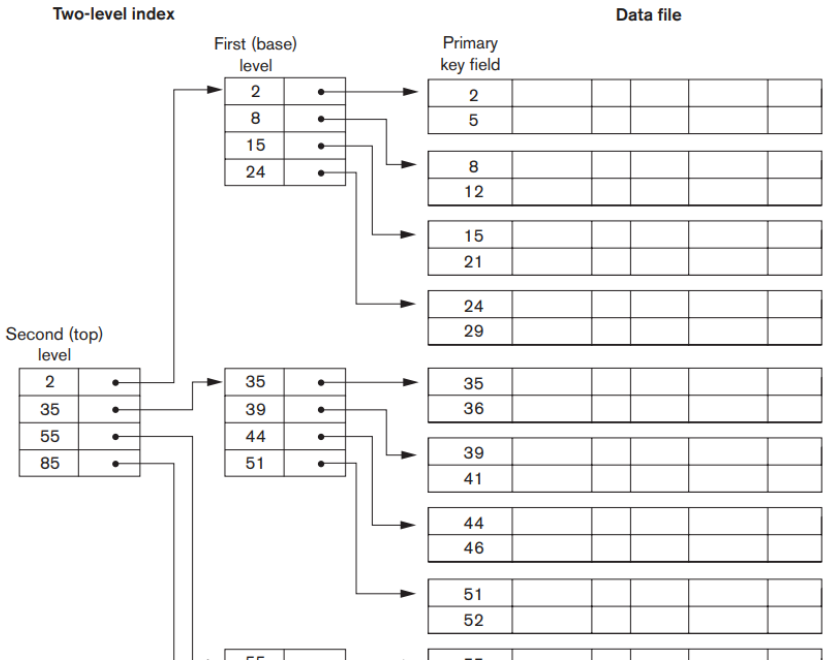
Once we decide to go for multilevel index then by default outer most level has one block.

ISAM (Index sequential access method) : Sequential file ordered by key + Multilevel PI

Multilevel CI + sequential file ordered by non-key is also called ISAM.

This is default

Figure
A two-level primary index resembling ISAM (indexed sequential access method) organization.



Primary index
Clustering index
Secondary indexes
Multilevel indexes
ISAM Files

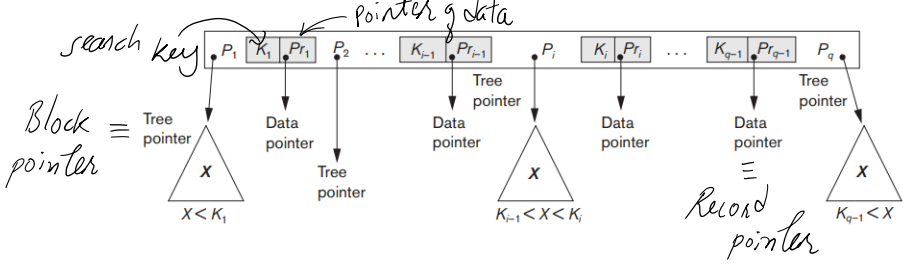
⇒ static indexes
Modifying these indices is costly. So, used when data in file is static. i.e. rarely changed. So, for dynamic files these are not good options

//Lecture 11

5.3) B TREE AND B+ TREE:

These are dynamic multilevel tree-based indexes. Modification is less costly and modification friendly.

Q : But why B and B+ tree and not other data structure like BST, AVL tree and Heap ? – because these are in-memory data structure. Not suitable for disk data structure. And In case of AVL tree we have one pointer per block so there is lots of storage waste we can instead use B tree and B+ tree.



5.3.1) B TREE :

Order of B tree (p) = Maximum number of tree (block or Node) pointers per node.

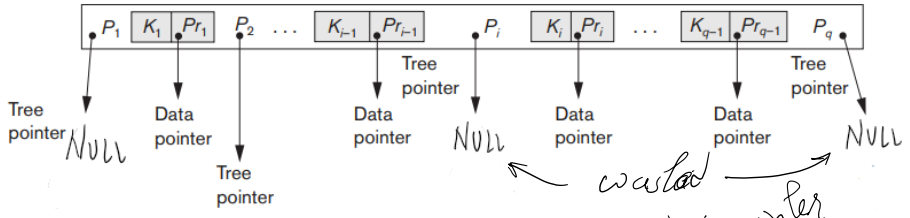
Some condition must be satisfied for B – tree :

1) Basic balanced binary search tree property
2) All leaves must be on the same level.
3) Space utilization rule : (≥ 50%)

   For root node : 2 BP to p BP; *For non-root node :* $\left\lceil \frac{p}{2} \right\rceil$ *to p BP*

   For keys root node : 1 to p – 1 *For keys non-root node :* $\left\lceil \frac{p}{2} \right\rceil - 1$ *to p – 1*

4) Leaf node structure,



At max, there are p block pointer and p – 1 <key, pointer> pair,

block size without block header → $Blocksize \geq p \times BP + (p-1) \times (keysize + DP)$

NOTE : Above formula is for all nodes (including leaf) but we know that leaf node block pointers points to NULL. So, in exam variation may be possible. Ex. Since leaf nodes require no pointers to children, they could conceivable store a different (larger) number of keys than internal nodes for the same disk page size. In this case, leaf node size ≥ $(\boldsymbol{p-1}) \times (\boldsymbol{keysizse + DP})$

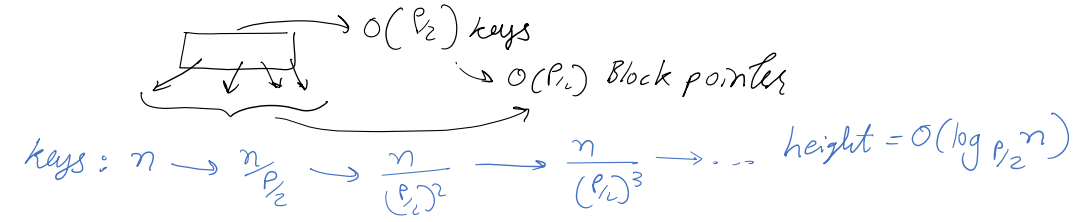
5) Keys inside one node should be sorted in increasing order.

//Lecture 12

Within node binary search

SEARCHING IN B TREE : same as binary search tree. TC with n keys = $O\left(\log_{\frac{p}{2}} n \times \log_2 p\right) = O(\log_2 n)$

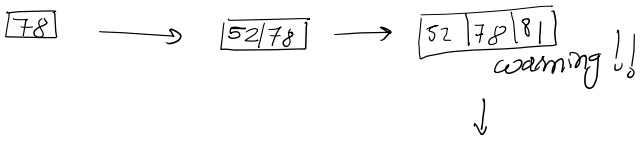
All node must have $\lceil P/2 \rceil$ BP in order to have max height

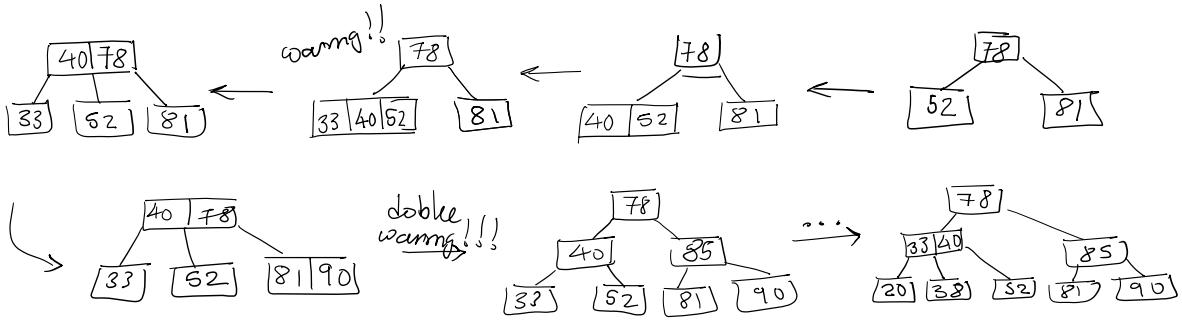


INSERTION IN B TREES : Insert the keys 78, 52, 81, 40, 33, 90, 85, 20 and 38 with order 3

Insertion always happens at leaf

Root : 1 to 2 key
Other node : 1 to 2 key





Sometimes if no. of keys in node is even then and overflow occurs then it is hard to decide which node to take as root in that case, we follow left and right biased split.

NOTE : Always stick to left or right biased but only one of them consistently. But by default, right biased.

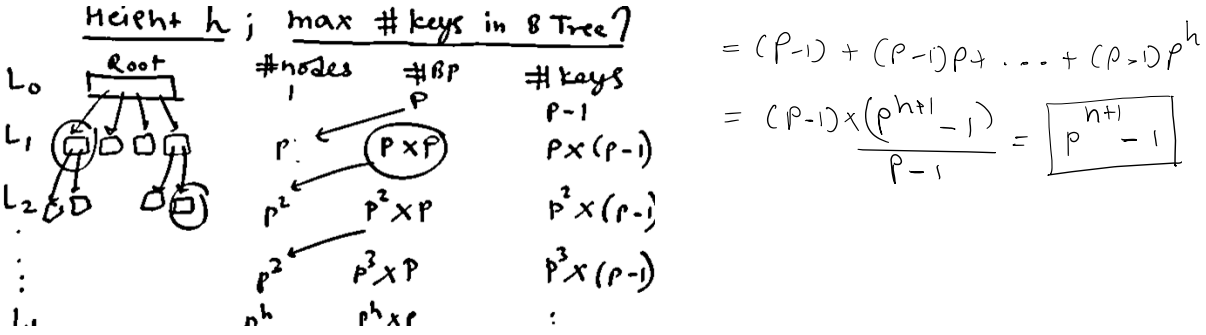
Insertion in B tree can cause split from leaf to root node.

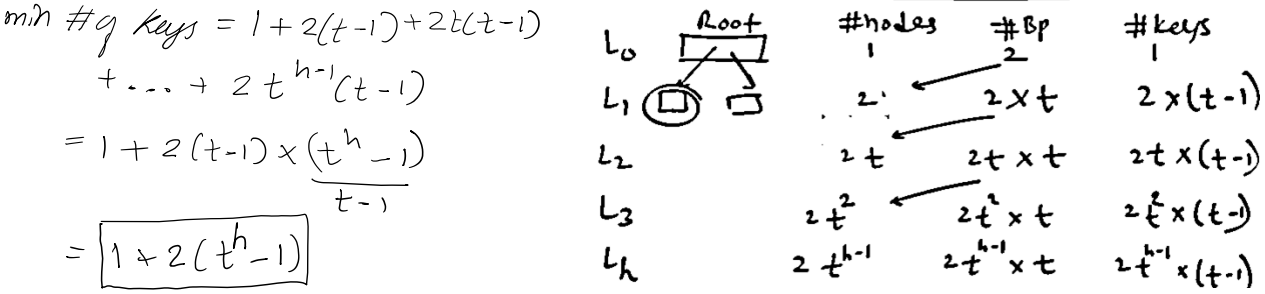
#leaf nodes = 1 + #keys in non-leaf nodes … idea is each non-leaf have some immediate predecessor in leaf node only. But there is rightmost leaf node which is not contain any immediate predecessor of any non-key thus we have added +1. This satisfy for B tree as well as B+ tree.

//Lecture 14

MAX AND MIN QUESTIONS : ***Note here h starts from 0***

Consider a B tree with height h and order p, then max # of keys in B tree will be





In B tree we can observe that some keys are in non-leaf node and some in leaf. In B+ tree all keys are in leaf node.

//Lecture 16

5.3.2) B+ TREE :

B+ tree keep all keys (Key + Data pointer) in leaf node while it keeps only key in non-leaf node. And leaf-nodes have block pointer at the end which points to next leaf node. There are no data pointer in non-leaf node.

Thus, in B+ tree we have duplicate key (only key not data pointer)

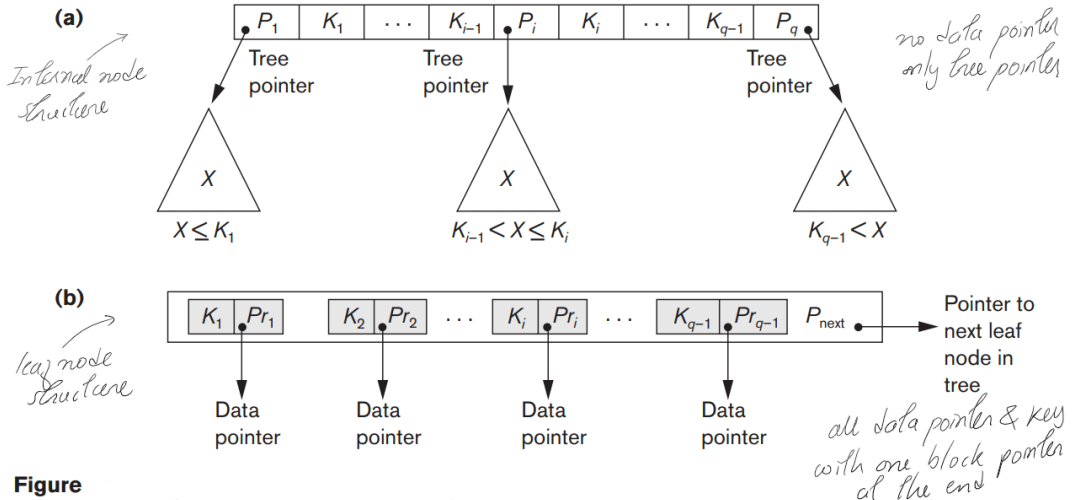


Figure
The nodes of a B^+-tree. (a) Internal node of a B^+-tree with $q-1$ search values. (b) Leaf node of a B^+-tree with $q-1$ search values and $q-1$ data pointers.

At max one key can have two duplications, one key at non-leaf and duplicate key in leaf node with data pointer. B+ tree never contains duplicates in same or distinct non-leaf or leaf.

Order of B+ tree = maximum number of total pointers node = O'(leaf) = O'(internal)

Here total pointer is used where as in B tree, tree pointers is used. This is because, leaf node in B+ tree contains data pointer and one block or tree pointer.

In B+ tree,

Node type	#of Block pointer	#of keys
Root	2 BP to p BP	1 Keys to p-1 keys
Leaf	$\lceil \frac{p-1}{2} \rceil + 1$ to p total pointer	$\lceil \frac{p-1}{2} \rceil$ to p-1 keys
Remaining internal node	$\lceil \frac{p}{2} \rceil$ to p BP	$\lceil \frac{p}{2} \rceil - 1$ to p-1 keys

While in B tree,

Node type	#of block pointer	#of keys
Root	2 BP to p BP	1 Keys to p-1 keys
Non-root	$\lceil \frac{p}{2} \rceil$ to p total pointer	$\lceil \frac{p}{2} \rceil - 1$ to p-1 keys

Thus, all things same except leaf node.

For leaf node : $Blocksize \geq (p-1) \times (keysize + DP) + BP$

For internal node : $Blocksize \geq (p-1) \times keysize + p \times BP$ ... provided block contains no header

FEW OBSERVATIONS :

Case 1 : Given Block size, if DP = BP then O'(leaf) = O'(internal)

Case 2 : Given block size, if DP ≠ BP then O'(leaf) ≠ O'(internal)

If DP > BP then O'(leaf) < O'(internal)

If DP < BP then O'(leaf) > O'(internal)

Case 3 : If B+ tree has order p means all nodes have order p

If RP > BP then internal nodes are not fully utilized

If RP < BP then leaf nodes are not fully utilized

Searching a B+ Tree

- For exact key values:
  - Start at the root
  - Proceed down, to the leaf

```
SELECT name
FROM   people
WHERE  age = 25
```

- For range queries:
  - As above
  - *Then sequential traversal*

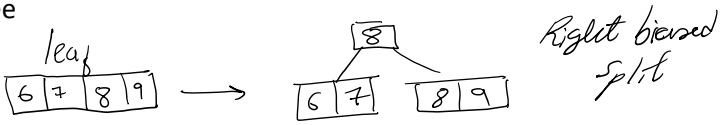
```
SELECT name
FROM   people
WHERE  20 <= age
  AND  age <= 30
```

//Lecture 18

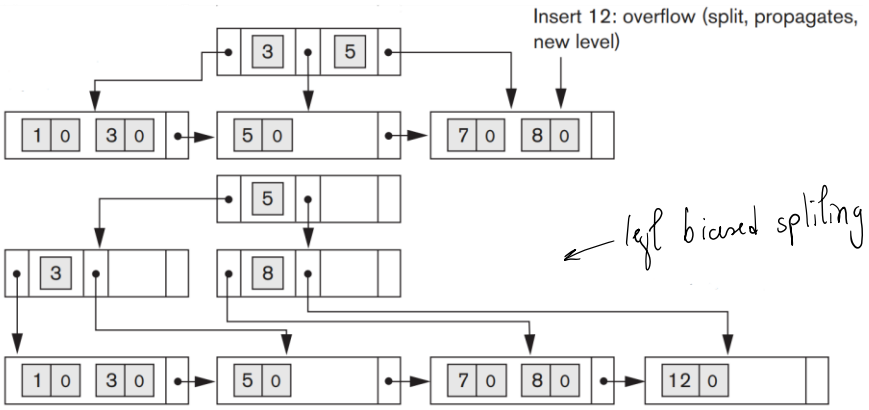
INSERTION IN B+ TREE :

Split of internal nodes : same as B Tree

Split of leaf nodes : order 4, max keys 3



Same as B tree, in B+ tree insertion starts at leaf node



Order in which keys are inserted in B+ tree matters

All searches (successful OR unsuccessful) in B+ tree take exactly same amount of time (index I/O cost = # of levels) i.e. **O(h), with n nodes and order p it will take O($\log_p n$)**

Similar to B tree we can also find no. max or min number of keys…

Max # keys	Min # keys

$L(I+1)^h$
L = Max # keys per leaf node
I = Max # keys per internal node
	$2(i+1)^{h-1}l$
	l = min # keys per leaf node
	i = min # keys per internal nodes

For the same system, same file : i.e. block size same, for n keys : then B+ tree will take less number of level and less number of nodes.

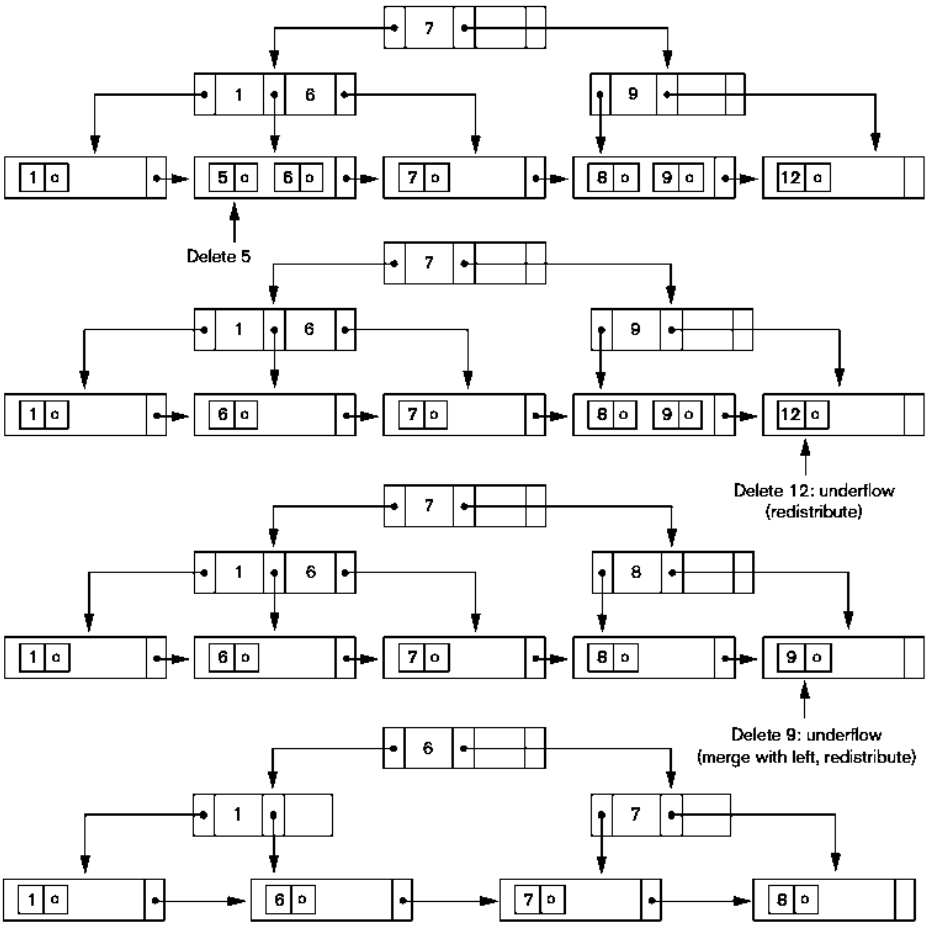
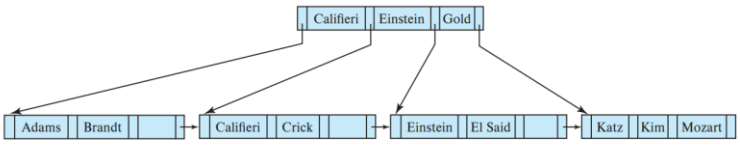


Figure 17.13
An example of deletion from a B^+-tree.

NOTE : During searching in B+tree we need not to go till leaf node because key might be present in internal node. Because we are doing searching not data access. If data access were given then we definitely go to leaf.



Deletion of "Gold" from the B^+-tree of Figure 14.19.

It is worth noting that, as a result of deletion, a key value that is present in a nonleaf node of the B+-tree may not be present at any leaf of the tree. For example, in Figure, the value "Gold" has been deleted from the leaf level but is still present in a nonleaf node.

//Korth page 733

5.4) Algorithm for join operation :

We study several algorithms for computing the join of relations, and we analyze their respective costs.

A running example the expression: student ⋈ takes

We assume the following information about the two relations:

- Number of records of student: $n_{student}$ = 5000.
- Number of blocks of student: $b_{student}$ = 100.
- Number of records of takes: n_{takes} = 10, 000.
- Number of blocks of takes: b_{takes} = 400.

5.4.1) Nested-loop join :

outer relation

```
for each tuple t_r in r do begin          ← Inner Relation
    for each tuple t_s in s do begin
        test pair (t_r, t_s) to see if they satisfy the join condition θ
        if they do, add t_r · t_s to the result;
    end
end
```

Above code shows a simple algorithm to compute the theta join, $r \bowtie_\theta s$, of two relations r and s. This algorithm is called the **nested-loop join algorithm**, since it basically consists of a pair of nested **for** loops.

Extending the algorithm to compute the natural join is straightforward, since the natural join can be expressed as a theta join followed by elimination of repeated attributes by a projection. The only change required is an extra step of deleting repeated attributes from the tuple tr · ts, before adding it to the result.

For each record in r, we have to perform a complete scan of s. We assume that **each relation must fit into one track**.

In worst case,

- *Block transfer* : **nr ∗ bs + br** … Only one buffer (to hold block) is present for each relation.
- *No. of Seeks* : **nr + br** … S is stored sequentially so one seek. Total nr seeks for s. and br seeks for r in total.

In best case,

- Block transfer : **br + bs** … Large MM to hold both table
- No. of seeks : **2** … Since large MM to hold both table, we need one seek for r and one for s.

To try running example take two cases for inner relation as student and same for takes.

5.4.2) Block Nested-Loop Join :

```
for each block B_r of r do begin
    for each block B_s of s do begin
        for each tuple t_r in B_r do begin
            for each tuple t_s in B_s do begin
                test pair (t_r, t_s) to see if they satisfy the join condition
                if they do, add t_r · t_s to the result;
            end
        end
    end
end
```

A variant of the nested-loop join where every block of the inner relation is paired with every block of the outer relation. Within each pair of blocks, every tuple in one block is paired with every tuple in the other block, to generate all pairs of tuples.

In the worst case,

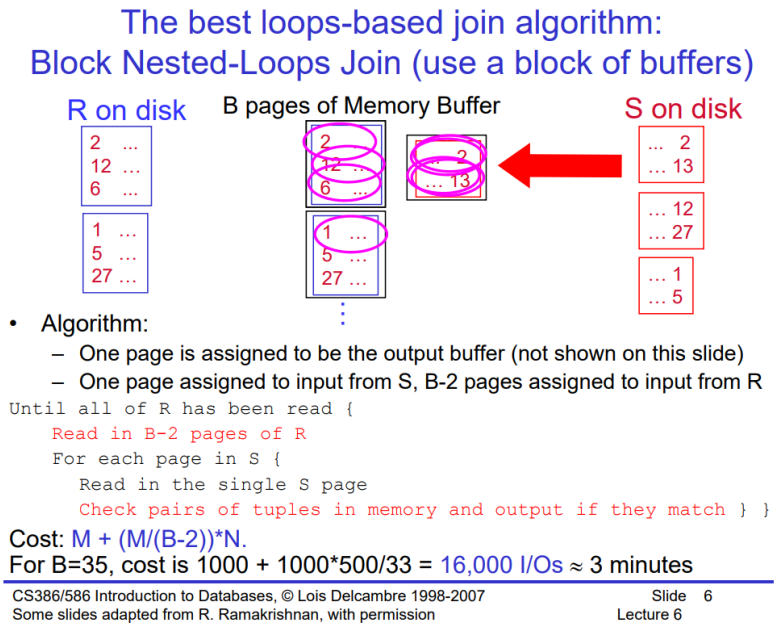
- block transfer : **br ∗ bs + br** … each block in the inner relation s is read only once for each block in the outer relation, instead of once for each tuple in the outer relation.
- No. of seeks : **2 ∗ br** … br for outer relation and another br for inner relation being sequential and access br times.

In best case : same reason as nested loop join,

- Block transfer : **br + bs**
- No. of seeks : **2**

VARIATION : If instead of 2 buffers what if we have B buffers

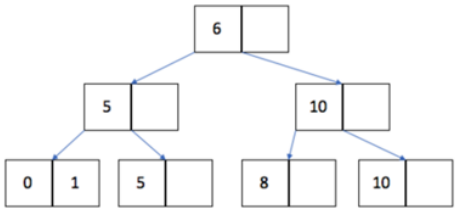
Here, M = br, N = bs, pr = nr, ps = ns



Algorithm	Number of times to read inner table	Cost formula (with example M = 1000, p_r = 100, b = 52, N = 500)
simple nested loops join	Once for each row in outer table	M + (M*p_r) * N 1000 + (1000*100) * 500 1000 + 100,000 * 500 1000 + 50,000,000
page-oriented nested loops join	Once for each PAGE of rows in outer table	M + M*N 1000 + 1000 * 500 1000 + 500,000
block nested loops join	Once for each b-2 pages of rows in outer table	M + (M/(b-2) * N 1000 + (1000/50) * 500 1000 + 20 * 500 1000 + 10,000

Silly Mistakes :

Consider the following B+ tree with order d = 3 (Maximum number of children a node can have is 3):



Here you were putting only natural numbers but fractional values are also valid

At max we can put 18 value in leaf node

5 is already there so (18 - 5)

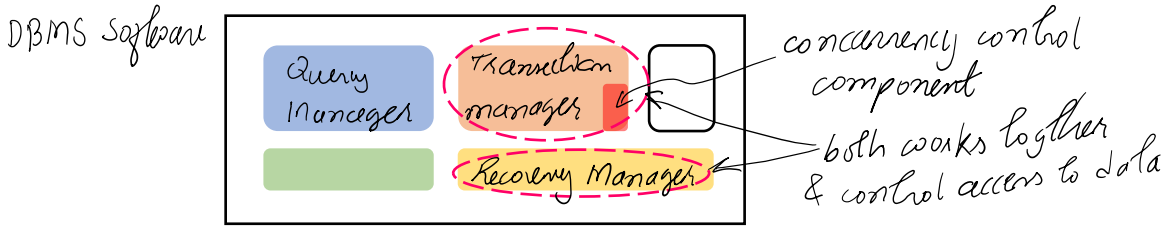
What is the maximum number of inserts we can do without changing the height of the tree? _____

6. TRANSACTION MANAGEMENT & RECOVERY

//Lecture 1

A **transaction** is seen by the DBMS as a series, or list, of actions. The actions that can be executed by a transaction include reads and writes of database objects.

Problem can happen when several transactions executed concurrently for example, two people booking same seat. This will be handled by transaction manager and recovery manager.

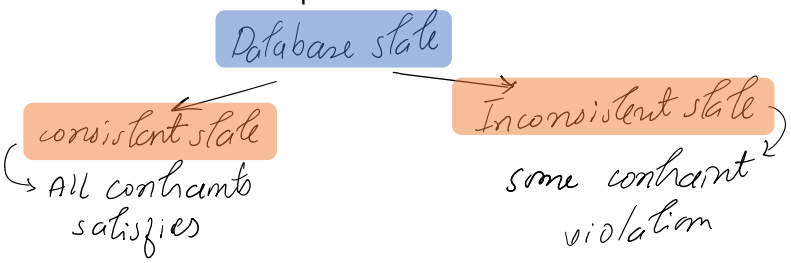


Data item can be database record, but it can also be a larger unit such as a whole disk block, or even a smaller unit such as an individual field (attribute) value of some record in the database.

The size of a data item is called its **granularity**.

Each data item has a unique name. if the item is a single record then the record id can be the item name etc.

State of database : A database has a state, which is value for each of its elements. Basically, database state = database instance = database snapshot



//Lecture 2

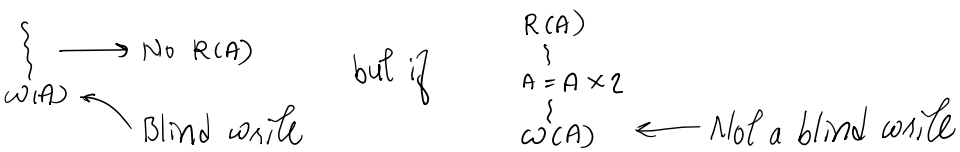
Read(A, t) : Read(A) ← conversion notation

- Block containing A comes in MM (buffer of transaction) from Disk
- T : temp program variable, do t = A

Write(A, t) :

- Block containing A comes in MM (buffer of transaction) from disk
- In memory, transfer t to A
- Store block containing A in disk

Blind write :

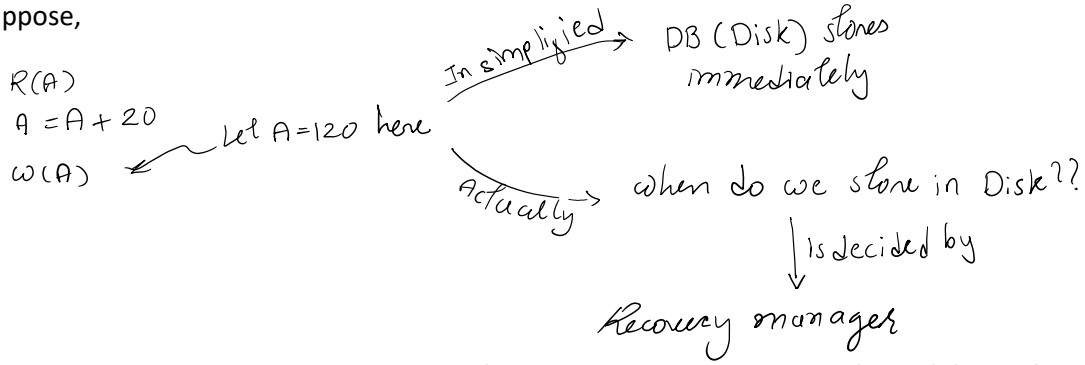


In each transaction each write should be followed by read on same object. If T1 : r(x), w(x), w(x) here second write is blind write because it is not followed by any read.

Commit : commit means end of transaction.

Here in all cases we are taking simplified version of read and write but actual implementation is different which we will study in recovery management chapter.

Suppose,



Meaning in actual implementation, value of A will be stored immediately after W(A) or before commit or after commit !!

NOTE : write(X, t) brings the block containing X BUT write(X, t) doesn't read X. It simply puts value of t into X.

Abort or Rollback : whatever you have done do it again (Undo changes by T may be some failure occur)

Concurrent control : database must go from one consistent state to another consistent state after concurrent execution of transaction.

//Lecture 3

6.1) ACID PROPERTIES :

Atomic, Consistency, Isolation, Durability

Transaction or set of transaction should satisfy these properties.

6.1.1) ATOMICITY :

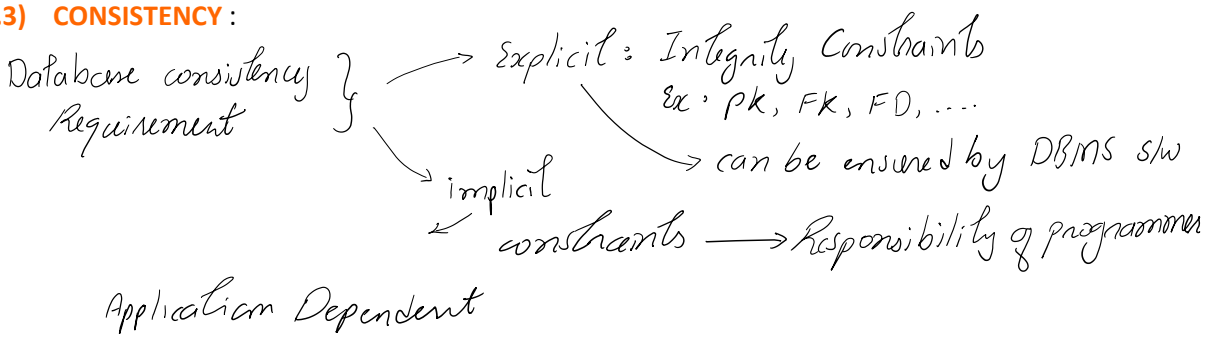
A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.

If a transaction begins to execute but fails for whatever reason, any changes to the database that the transaction may have made must be undone.

6.1.2) DURABILITY :

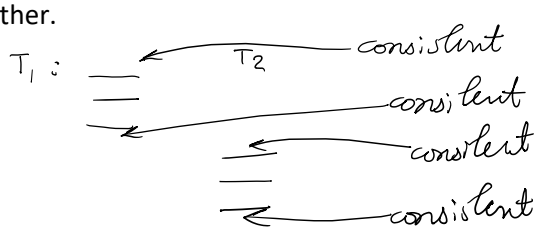
After commit, despite of any failure, all changes by T must be reflect in DB (must be made permanent in DB).

6.1.3) CONSISTENCY :



If transaction is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another. In sort, individual transaction (running in isolation) must take database from one consistent state to another.

Q : Is serial execution always correct ? in other words does serial execution always preserves database consistency ? – Yes, because of consistency property, individual transaction must take database from one consistent state to another.

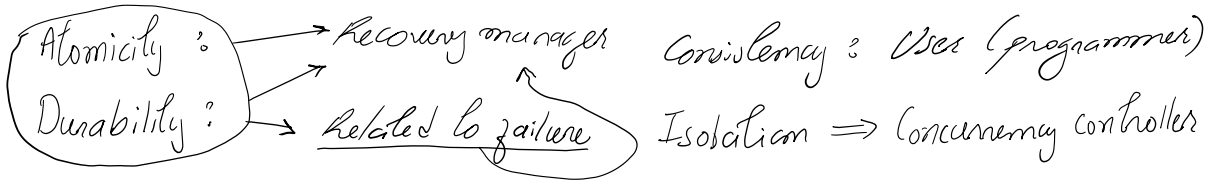


6.1.4) ISOLATION :

A transaction should not see the changes made by other transactions. Pseudo feel that you are executing alone.

There are many ways to ensure isolation but strongest way is serializability.

Now, we saw components of DBMS in previous discussion,



//Lecture 5

6.2) SERIALIZABILITY :

When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from all the various transactions is known as a **schedule** (or history).

But within transaction order of operation should not change.

Every serial schedule is also concurrent schedule

Throughput : No. of transaction that can execute in per unit time.

Aim : to ensure **isolation property**

The idea of serializability is allow those concurrent execution whose effect is equivalent to some serial schedule.

S (schedule) is **serializable** iff for *every* initial Database state, effect of S is same as some serial schedule. If for Some initial Database state, effect of S is not same as some serial schedule then S is not serializable.

T1	R(B)	R(A)	A=A+100	B=B+A	W(A)	B=B+100	W(B)	A=A+B		COM

Here As W(A) is not done after A=A+B then content of database is A = 200, B = 400. Instead of A = 600.

Each data value maintains one buffer in which it stores its updated value and on W(A) it stores into DB.

//Lecture 6

6.2.1) DIFFERENT NOTIONS OF SERIALIZABILITY :

Serializability (in general) – very hard or almost impossible to implement practically

Conflict serializability – very easy to implement

View serializability – NP-complete implementation i.e. exponential time complexity.

In this section we will focus on serializability (in general),

We say a schedule S is serializable if there is a serial schedule S' such that for *every* initial *consistent* database state, the effects (final database state) of S on Database is same as effect of some serial schedule on Database.

Problem with general notion of serializability : we have to consider all computations which is hard to implement and expensive.

Solution : Don't rely in computations, only see read, write not computations.

//Lecture 7

6.2.2) CONFLICT SERIALIZABILITY :

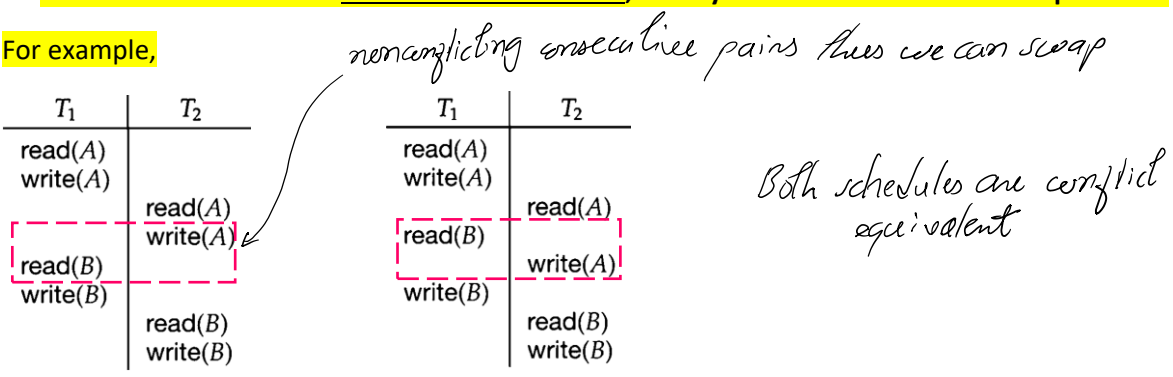
Conflict serializability purely based on read and write operation it ignores other computation for example,

CONFLICT PAIRS : We say that two operations I and J in a schedule conflict if they are operations :

- By different transactions AND
- On the same data item, AND
- At least one of these instructions is a write operation.

If a schedule S can be transformed into a schedule S' by a series of swaps of nonconflicting consecutive instructions of different transaction, we say that S and S' are conflict equivalent.

For example,



But why this concept here ? – because we can check serializability,

Idea 1 : given schedule S; to check conflict serializability of S :

S → Repeatedly Swap consecutive non-conflicting pair of diff. transaction → some serial schedule

Every serial schedule is conflict serializable

NOTE : There is no effect of commit in conflict serializable because by default by the end of transaction if nothing is written it is automatically committed. So, there is no effect of commit operations on serializability.

If abort is coming in some transaction simply consider that, that transaction was never existed (ignore transaction completely).

Idea 2 : test by **precedence graph** (an easy test)

Represent transaction by node and conflicting pairs by edges

For example, $T_1 \rightarrow T_2$ implies that there is conflicting pair between T1 and T2 but it T1's operation coming first in sequence.

Transaction T_1	Transaction T_2	Transaction T_3

	read_item(Z);
	read_item(Y);
	write_item(Y);	

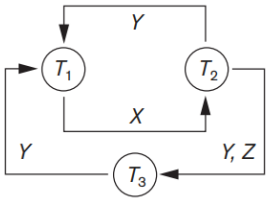
		read_item(Y); read_item(Z);
read_item(X); write_item(X);		
		write_item(Y); write_item(Z);
	read_item(X);	
read_item(Y); write_item(Y);		
	write_item(X);	

Time

But here Topological order is not possible thus ESS not possible

we follow topological order

To find this



Equivalent serial schedules

None

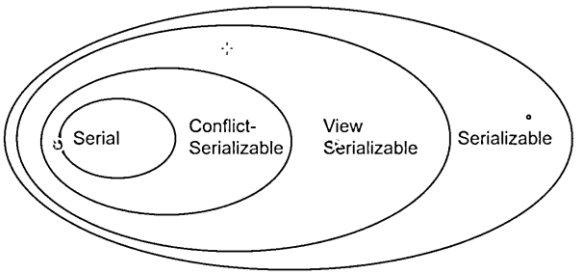
Reason

Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$
Cycle $X(T_1 \rightarrow T_2), YZ(T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

A schedule S is conflict-serializable iff precedence graph of S is Acyclic

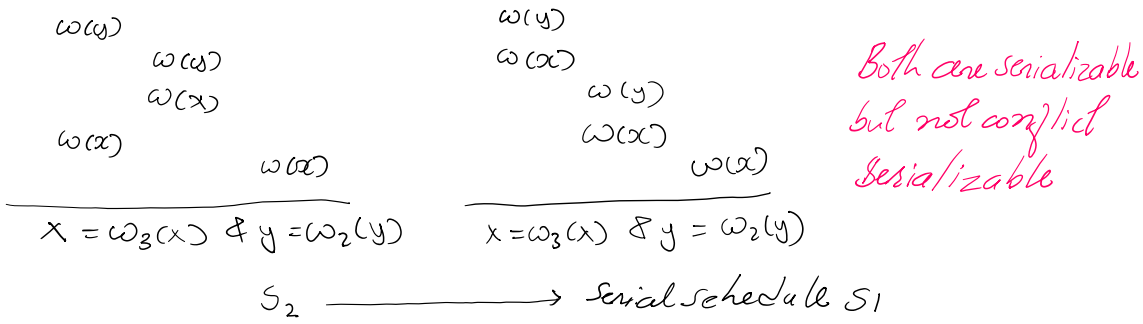
//Lecture 8

6.2.3) VIEW SERIALIZABILITY :



Consider the two schedules :

S_2 : T_1 T_2 T_3 S_1 : T_1 T_2 T_3 all are blind write

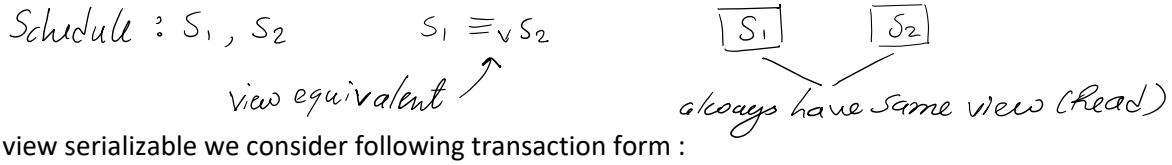


Every conflict serializable schedule is also a serializable

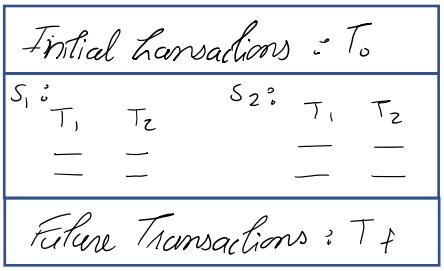
Conflict-serializability is sufficient but not necessary for serializability

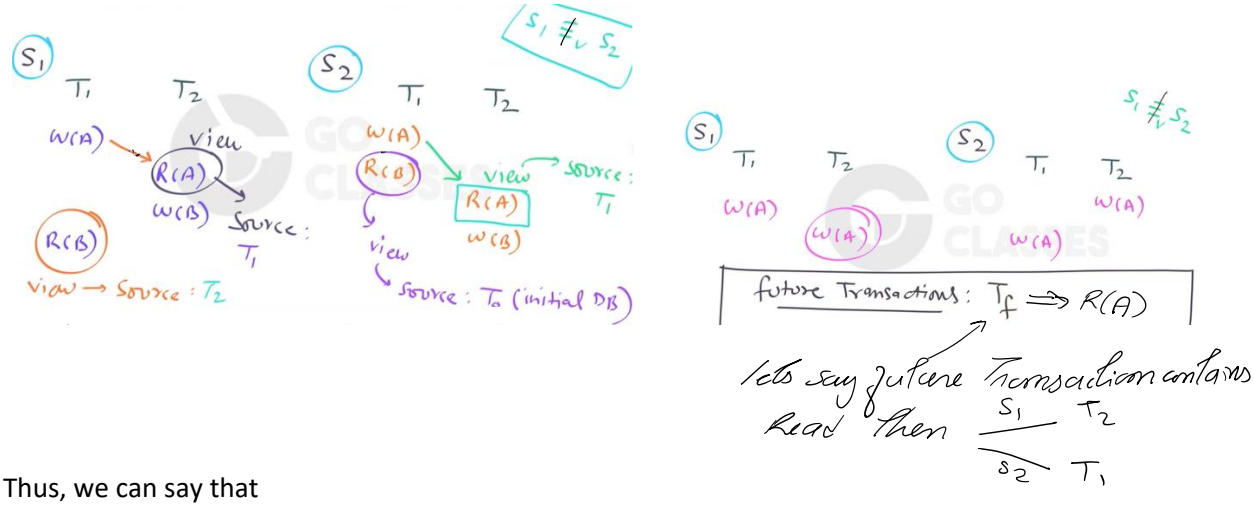
Q : Can we cover more of serializable schedules ? – yes ! by using view serializability.

View serializability is slightly similar to conflict serializability because it is purely based on read, write operations only. But <u>for view serializability schedule, it should be view equivalent to some serial schedule.</u>



In view serializable we consider following transaction form :





Thus, we can say that

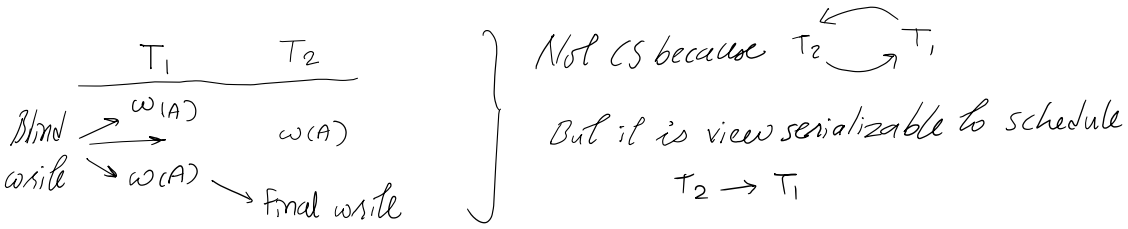
Two schedules S1, S2 on same set of transactions, $S1 \equiv_v S2$ *if and only if*

- *Same read operations in S1, S2 must have same view*
- *To future transactions, for every element; S1 and S2 should provide same view*

Here same view means by same transaction and same write

A schedule S is said to be view serializable if it is view equivalent to a serial schedule.

Consider following example,



This is happening because View serializability uses the blind writes carefully and conflict serializability doesn't take benefit of the blind writes.

Which means If schedule is non-conflict serializable and no blind writes then not view serializable but if schedule is non-conflict serializable and blind write is present then it can be view serializable.

If a schedule is VS but not CS then there is blind write. But if there is blind write then schedule may or may not be VS.

View serializability : less strict, more time complexity (exponential TC)

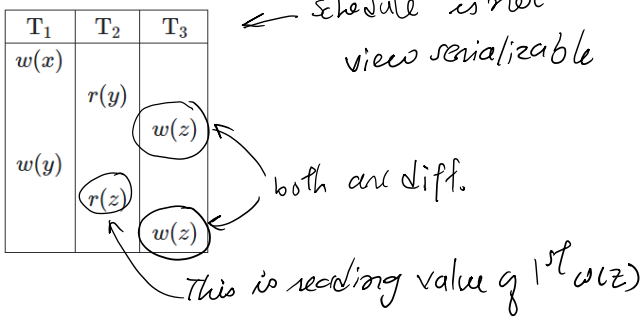
Conflict serializability : Stricter, less time complexity (polynomial TC)

If a schedule only contains read, write then VS = serializability but if we some computations are also given then VS <> serializability

For example,

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s+200		
	WRITE(A,s)	325 ✓	
	READ(B,s)		
	s := s+200		
	WRITE(B,s)		225
READ(B,t)			
t := t+100			
WRITE(B,t)			325 ✓

This is serializable but it is not VS & CS



//Lecture 9

6.3) RECOVERABILITY :

So, by far we have studied schedule while assuming implicitly that there are no transaction failures. We now address the effect of transaction failures during concurrent execution.

Serializability concept helps us ensure (if not failure, then)

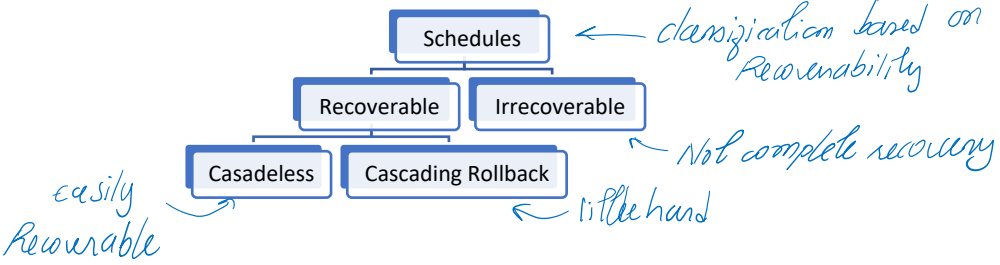
- Isolation
- Consistency of database

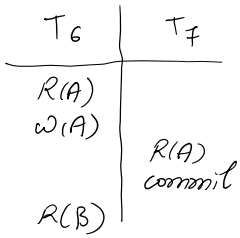
- Consistency of transaction
- Consistency of schedule
- Atomicity
- Durability

NOTE : Showing wrong result/values to user is also considered violation of consistency of database.

System : T_1 → Committed
T_2 → Uncommitted
failure occurs

Action : Rollback (Undo) T_2
T_1 must persists in DB
T_1 must not be undone



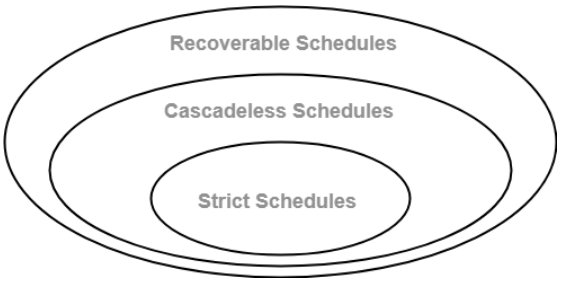


Here if for some reason T6 aborts after R(B) then T7 has read incorrect value of A & committed thus this schedule is nonrecoverable Atomicity violated!!

NOTE : serializability and recoverability are not related (nothing to do with each other)

To preserve atomicity in above case we have to abort T6 and all the transaction who have read from T6. But as T7 already committed so is this not possible thus violation of atomicity.

⋆ **RECOVERABLE SCHEDULE MAY BE ONE OF THESE KINDS** :



Cascading schedule or **Cascading recoverable schedule** : If in a schedule, failure of one transaction causes several other dependent transactions to rollback or abort.

Cascadeless scheduling or **Cascadeless Recoverable schedule** : A schedule is said to be **cascadeless**, or to **avoid cascading rollback**, if every transaction in the schedule reads only items that were written by committed transactions. – don't read from uncommitted write. (dirty read)

Strict schedule or **Strict recoverable schedule** : Transactions can *neither read nor write* an item X until the last transaction that wrote X has committed (or aborted). – don't read/write from uncommitted write.

//From korth page chapter 19

6.3.1) RECOVERABILITY USING THE LOG TO REDO AND UNDO TRANSACTIONS :

- Transaction Ti needs to be undone if the log contains the record **<Ti start>** but does not contain either the record **<Ti commit>** or the record **<Ti abort>.**
- Transaction Ti needs to be redone if the log contains the record **<Ti start>** and either the record **<Ti commit>** or the record **<Ti abort>**.

When a system crash occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. In principle, we need to search the entire log to determine this information to reduce these types of overhead, we introduce checkpoints.

Checkpoint is performed as follows

1) Output onto stable storage all log records currently residing in main memory.
2) Output to the disk all modified buffer blocks.
3) Output onto stable storage a log record of the form <checkpoint L>, where L is a list of transactions active at the time of the checkpoint.

Example, As an illustration, consider the set of transactions {T0, T1, …, T100}. Suppose that the most recent checkpoint took place during the execution of transaction T67 and T69, while T68 and all transactions with subscripts lower than 67 completed before the checkpoint. Thus, only transactions T67, T69, …, T100 need to be considered during the recovery scheme. Each of them needs to be redone if it has completed (i.e., either committed or aborted); otherwise, it was incomplete and needs to be undone.

NOTE : Undo means we have to delete all updates made by that transaction (even before checkpoint). Redone means we again have to do operation after checkpoint.

//Lecture 10

6.4) CONCURRENCY CONTROL :

It basically means ensuring isolation property while concurrency execution of transactions.

Till now we are doing “completing schedule” then check for serializability. But better idea would be “prepare beforehand”

Thus, while database is in execution, we ensure that only conflict serializability schedule generate, for that we need set of different rules.

Q : Why concurrency control is needed ? –

- **The Lost Update Problem** : This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect.

In Serializable schedules (Conflict Serializable), We DO NOT have lost update problem because when there is lost update problem, there will be cycle in the precedence graph, So Not Conflict serializable.

- **The Temporary update (dirty read) problem** : One transaction reads from write of uncommitted transaction and commits.
- **The Incorrect Summary Problem** : If one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items.
- **The Unrepeatable Read Problem** : Another problem that may occur is called unrepeatable read, where a transaction T reads the same item *twice* and then item is changed by another transaction T' between the two reads.

So, *each transaction*, individually, follow these rules... Automatically system ONLY generates conflict serializable schedules.

Two sets of rules :

- Lock based protocols (kinds of obvious rules)
- Time stamp-based protocols

6.4.1) LOCK BASED PROTOCOLS :

In this data items are locked in two modes :

- ⇨ **Exclusive (X) mode** : Data item can be both read as well as written. X-lock is requested using lock-X instruction
- ⇨ **Shared (S) mode** : Data item can only be read. S-lock is requested using lock-S instruction.

Note that all these requests are handed by CCM (concurrency control manager) in database s/w.

Scenario 1 : *Early unlocking*

But by using these general S, X locking protocol, non-serializable schedules are possible (can be generated).

Scenario 2 : *Delayed unlocking*. In this case also it gives us inconsistent results.

Thus, *using locks(read/write locks) in transactions, as described earlier, does not guarantee serializability of schedules on its own*. **Deadlock possible, Starvation possible, inconsistency possible. Serializability not guarantee.**

We need more rules !!

//Lecture 11

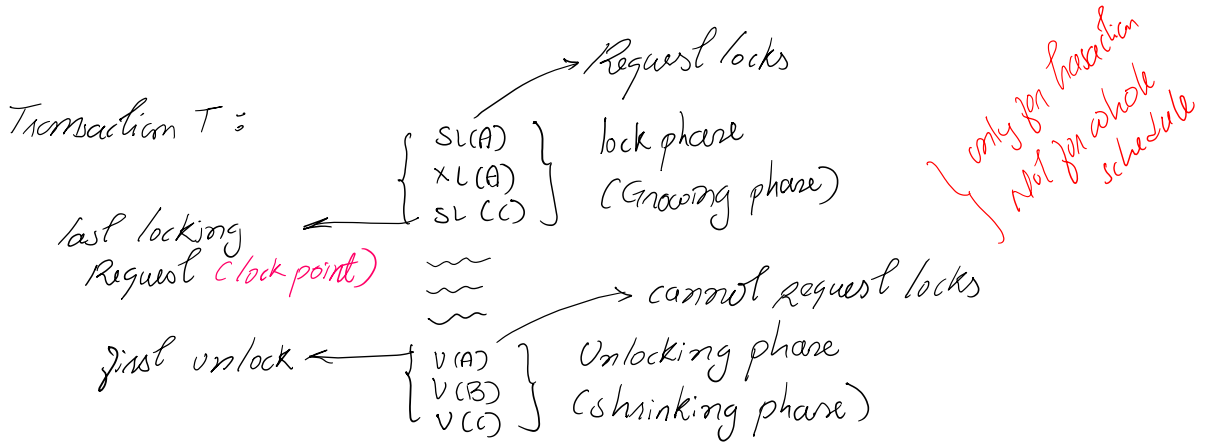
6.4.2) TWO PHASE LOCKING PROTOCOL (2PL) :

There are many two-phase locking protocols :

- Basic 2PL (by default 2PL)
- 2PL with lock conversion
- Conservative 2PL
- Strict 2PL
- Rigorous two-phase locking

Set of rules :

1) General locking rules as stated previously (like S, X)
2) Lock phase (growing phase)
3) Unlocking phase (shrinking phase)



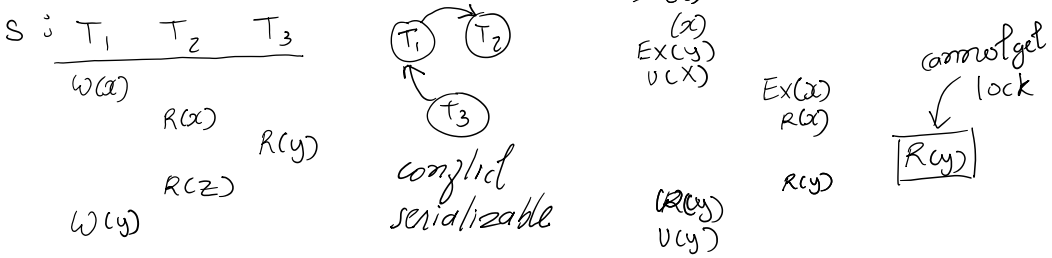
This protocol ensures that *if every transaction individually follow 2PL rules* *then* ***every schedule that will be generated will be conflict serializable***. And the equivalent serial schedule is in the order of **lock point.**

Which means if 2PL → conflict serializable. Thus, if not 2PL then it may or may not be conflict serializable. Thus, not all conflict serializable schedule is produced by 2PL.

Not conflict serializable → schedule is not allowed by 2PL

Again, *transaction can be called 2PL not schedule.*

For example,



Thus, above schedule is CS but T3 is not following 2PL.

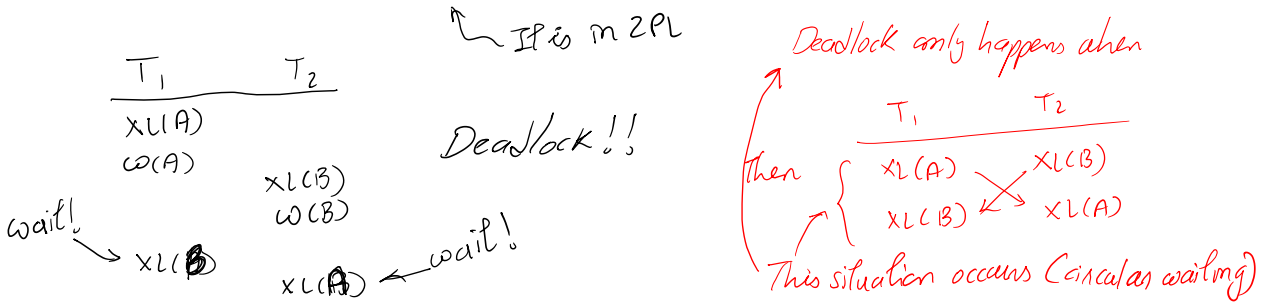
But this suffers from deadlock !?!?!

For example,

If is in 2PL

T_1 : W(A) , W(B) → T_1 : XL(A) ; W(A) ; XL(B) ; W(B) ; U(A) ; U(B)

T_2 : W(B) , W(A) → T_2 : XL(B) ; W(B) ; XL(A) ; W(A) ; U(A) ; U(B)



//Lecture 12

Not only deadlock but it does not guarantee recoverability, cascade less schedule.

VARIATION OF 2PL : ensures 2PL + recoverability

Not true for strict 2PL

Strict 2PL : Release Exclusive lock only after commit/abort of all transactions which uses them,

Rigorous 2PL : Release **all** locks only after commit/abort, transaction can be serialized in commit order

Conservative 2PL : Take all the locks before you starting transaction

	Conflict serializability	Deadlock freedom	Recoverability
2PL	✅	❌	❌
Strict 2PL	✅	❌	✅
Rigorous 2PL	✅	❌	✅
Conservative 2PL	✅	✅	❌

Recall the size of a data item is called its ***granularity***.

Q : Does granularity of data item affect amount of concurrency we can achieve ? – Yes, if entire Database is data item then only serial execution of transaction is possible because at a time, we can lock entire DB then unlock then lock again unlock, so only serial execution is possible. Thus, less concurrency.

The larger the data-item, the lesser the amount of concurrency we can achieve

⋆ **2PL WITH LOCK CONVERSION** :

Two new instructions :

- **Upgrade (A)** : Converts shared lock to exclusive lock ……. Applicable only in growing phase
- **Downgrade (A)** : Convert exclusive lock to shared lock ……. Applicable only in shrinking phase

But because of lock conversion, deadlock is possible.

//Lecture 13

6.4.3) DEADLOCK HANDLING :

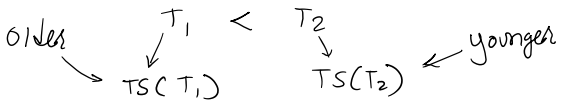
System with many transactions … Deadlock happens when there exists a set of transaction such that every transaction in set is waiting for other transaction to release locks.

If system is already in deadlock → Abort some of the transactions

DEADLOCK AVOIDANCE : You have to periodically check for deadlock and avoid.

DEADLOCK PREVENTION : If you implement me then deadlock never occur and you don't have to periodically check for deadlock.

- *Conservative approach* : Take all the locks before transaction starts execution. Deadlock never possible.
- *Partial ordering the data-items* : Given then order and access them according to this order only.
- ***Transaction Timestamp*** : Time at which transaction was created.



We will always give more priority to older as these transactions had done more computation.

Deadlock prevention schemes which uses transaction timestamp :

1) **Wait-Die scheme** : Ti wants to lock A but Tj holds lock on A

If TS(Ti) < TS(Tj) then Ti waits for Tj to release

If TS(Ti) > TS(Tj) then Ti dies (Aborts)

Non-preemptive
Don't Disturb executive Transaction

In short *older wait, younger die !*

2) **Wound-wait** : Ti wants lock A but Tj holds lock on A

If TS(Ti) < TS(Tj) then Ti wounds(Aborts) Tj and snatch lock A from Tj

If TS(Ti) > TS(Tj) then Ti waits for Tj

Preemptive

In both preventions when a transaction aborts, it will restart after some time but with old timestamp to avoid starvation.

If we don't implement deadlock prevention then deadlock possible thus, we periodically **detect** deadlock and recover.

DEADLOCK DETECTION : (Dynamically changing because Ti request and release)

We use **wait-for graph**, where nodes are transaction and Ti → Tj edge implies Ti waiting for Tj to release lock.

The system is in deadlock state if and only if the wait-for graph has a cycle.

Again, as we say earlier, to remove deadlock we abort some process who have done less computation.

//Lecture 14

6.4.4) TIMESTAMP BASED PROTOCOLS FOR CONCURRENCY CONTROL PROTOCOLS :

We again focus our mind into Concurrency control protocols. We already saw lock-based protocol then 2PL. Now, we will see timestamp-based protocols.

There are two types of people in the world :

- Your actions decide your fate (future).
- Your fate is already decided (you are just acting according to it)

Locking protocol : *Your actions(locking) decide your fate (serializability)*

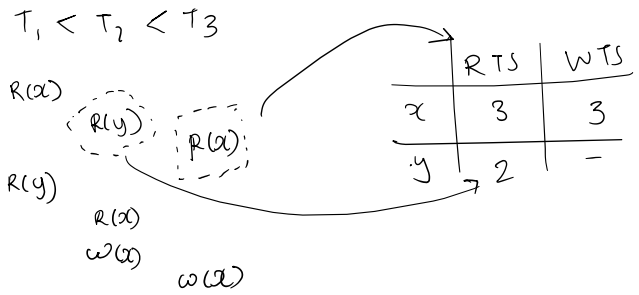
Timestamp based protocol : *Your fate (serializability) is already decided (you are just acting (locking) according to it)*

TIMESTAMP BASED PROTOCOL (TSP) :

In timestamp protocol, when a transaction T is born (submitted/created), T is assigned a timestamp. In this protocol serializability order is decided in advance (in order you're born).

This protocol has *one rule* : ***If TS(T) < TS(T') then order of conflict operations in any schedule between T, T' should be in the order of T → T'. If not then the transaction who is LATE, is aborted & restarted with a NEW FRESH Timestamp.***

NOTE : For every element x protocol uses : Read time stamp RTS and Write Time stamp WTS (both tells us who is the youngest transaction to perform these operation).



Transaction who came first assigned a lower number then the transaction who came later.

1) Whenever a transaction T issues a write_item(X) operation, the following check is performed : If **read_TS(X) > TS(T)** or if **write_TS(X) > TS(T)**, then abort and roll back T and reject the operation. **T Write करने के लिये गया ओर T से छोटा कोइ Read/Write करके चला गया तोह reject operation and abort & rollback T**.

If above does not occur, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).

2) Whenever a transaction T issues a read_item(X) operation, the following check is performed : If **write_TS(X) > TS(T)**, then abort and roll back T and reject the operation.

If **write_TS(X) ≤ TS(T)**, then execute the read_item(X) operation of T. And set read_item(X) = Youngest timestamp.

In short, we can say that

All conflicting operations must be in this order of transaction timestamp

THOMAS WRITE RULE : Only modifies conditions for write operations as follows :

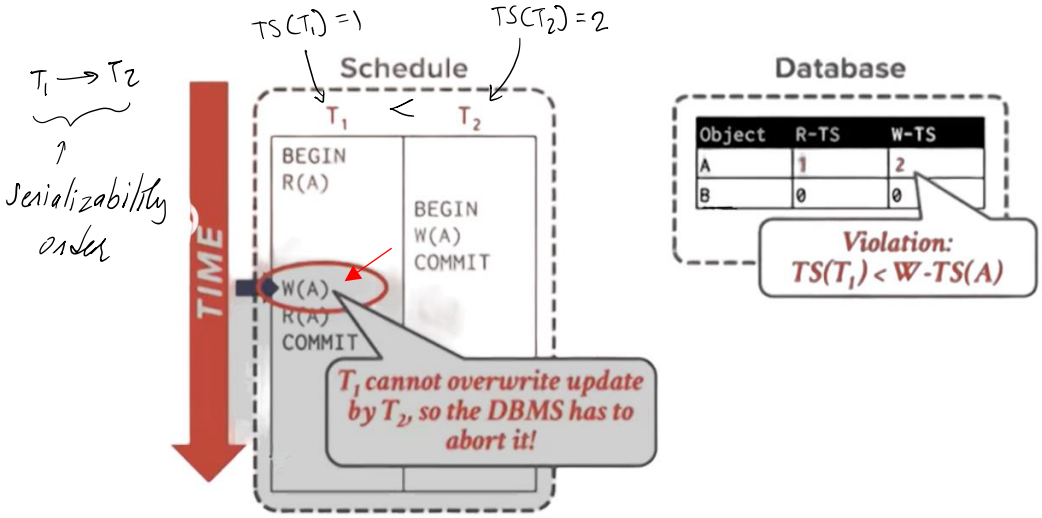
Whenever a transaction T issues a write_item(X) operation, the following check is performed :

1) Read_TS(X) > TS(T) → abort and roll back T and reject the operation. (same as timestamp)
2) Write_TS(X) > TS(T) → ignore T's request but continue processing.
3) If neither of this satisfy then set write_TS(X) = TS(T)

By default, we do not use Thomas write rules

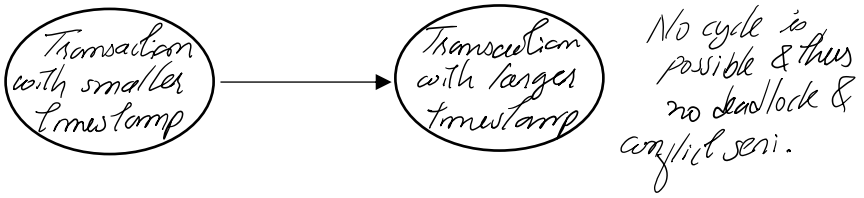
Using Thomas write rule, we can allow/generate **some** VS and non-CS schedules.

Example of TSP,



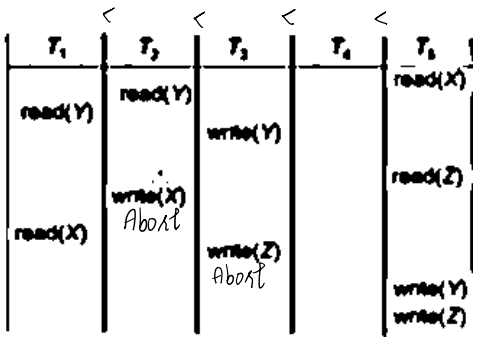
TSP ensures conflict serializability in order of timestamp and avoids deadlock but *does not avoids cascading rollback*.

Reason : All the arcs in the precedence graph are of the form,



NOTE :

- **If you are using Thomas write rule then some non-conflict serializable (but correct) are also allowed.**



- **As soon as conflicting in opposite direction then abort it and do not consider in upcoming pair making.**
- **2PL is not subset of TSP and vice versa.**

Silly mistake :

Two transactions T1 and T2 are given as

T1: r1(X);w1(X);r1(Y);w1(Y);

T2: r2(Y);w2(Y);r2(Z);w2(Z);

The total number of interleaving that can be formed by T1 and T2 is ____

if CS interleaving were asked then take W1(Y) & start placing into T2

interleaving can be CS or not CS we don't care

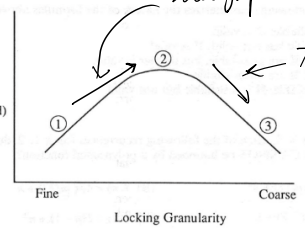
In the database systems, the size of a data item is called its granularity. A data item can be a database record, but it can also be a larger unit such as a whole disk block, or even a smaller unit such as an individual field (attribute) value of some record in the database.

The granularity of the data items is the portion of the database a data item represents. Granularity(size) of data items can affect the amount of Concurrency we can achieve with concurrent execution of transactions.

In some database systems, data items are protected by locks to ensure correct behavior in the presence of concurrency. Locking is said to be "fine-grained" if each lock protects only a few data items; it is said to be "coarse-grained" if each lock protects many data items. The following performance graph is typical.

locking per data → high overhead due to locking

locking more data → low overhead due to locking



Throughput increases because locking overhead decreases

Throughput decreases because concurrency decreases due to high granularity