(6) SELECT A, B, Sum(B) FROM R GBY A;

This is invalid. For every group at most one tuple should be there, so what would be "B" per group.

- Note: (1) For any attr. (or attrs) X, Y, Z on which GroupBy is performed must be there in Select clause, i.e

SELECT X, Y, Z, ... GROUP BY X, Y, Z

(2) For NULLs in grouping attr, a separate group is created.

(3) For a query with GROUP BY, at most one tuple in o/p per group.

(4) In a query with GROUP BY, SELECT clause can have
(1) Grouping attr. X, Y, Z..
(2) Aggregated attr.
(3) or Both.

i.e.

SELECT X, Y, Z, Sum(A), Min(B)..... GBY X, Y, Z

(5) In queries w/ GroupBy; no. of tuples in output:
- equal to no. of groups if no HAVING clause.
- less than equal to no. of groups if HAVING clause.

(6) Queries without GroupBy can be thought of as single group.

(7) SELECT X, Y, Z FROM .. GBY X Y Z; behaves as
SELECT DISTINCT X, Y, Z FROM ...;

Ex. R(A, B), S(B, C)

R = {(1,2), (1,2), (1,3), (N,N), (N,5), (7,2)}

S = {(1,2), (1,4), (4,3), (N,N), (2,1)}

SELECT A, C, COUNT(R.B) FROM R, S WHERE A <> S.B GBY A, C;

R × S: (where A <> S.B)

A	R.B	S.B	C
1	2	4	3
1	2	2	1
1	2	4	3
1	2	2	1
1	3	4	3
1	3	2	1
7	2	1	2
7	2	1	4
7	2	4	3
7	2	2	1

Output:

A	C	Count(R.B)
1	3	3
1	1	3
7	2	1
7	4	1
7	3	1
7	1	1

We could have done it in alternate way. Since this query has GBy clause, the no. of tuples in o/p would be same as no. of groups. Also since WHERE clause is there we first have to remove rows.

So, possible groups: (1, 3), (1, 1) (7,2) (7,4) (7,3) (7, 1)

∴ No. of tuples in o/p: 6.

- In SQL-92, we can't have HAVING clause without GROUP BY clause.
- In HAVING clause the attr. used should either be aggregated or subset of attr. used for GROUP BY.

Ex. Find the age of the youngest sailor for each rating level above 7.
Which of the following queries gives desired result?
(A) SELECT rating, min(age) FROM Sailors WHERE rating > 7 GR BY rating;
(B) SELECT rating, min(age) FROM Sailors GROUP BY rating HAVING rating > 7;
Both give correct answer.

Ex. In Emp table, count of total no. of employees whose salary exceeds $40,000, in each dept, but only for dept with more than 5 employees. Is this query correct:
SELECT Dept, COUNT(*) FROM Emp WHERE salary > 40000 GR BY Dept HAVING COUNT(*) > 5;

No. The WHERE clause executes before GBY & HAVING, so in the above query the HAVING clause is only checking for each dept if no. of (emp. with Sal > 40000) > 5 or not.

- Order By
After everything is executed, ORDER BY can be used to sort results.
Ex. ... ORDER BY Desc A, Asc. B, ...;
By default ordering is ascending.
Ex. ... ORDER BY A, B;
Ascendingly sort by A, then sort by B ascendingly.

- Order of execution:

(1) FROM
(2) WHERE
(3) GROUP BY
(4) HAVING
(5) SELECT
(6) ORDER BY

```
SELECT ...  FROM ...  WHERE ...
GROUP BY ...  HAVING ...
ORDER BY ...;
```

- Nested Query
  - SELECT ...
  - FROM ...
  - WHERE ...
  - GROUP BY ...
  - HAVING ...

  can put subquery.

- Most common is to have subquery in WHERE clause.
- If subquery is put in SELECT clause, then result of the subquery must be single row single column. So the subquery must use single aggregate func. without groupby.
- There are two types of Nested Query:
  - Simple Subquery: The subquery doesn't depend on attr. of the outer query. (You can run the subq. indep. of outer query)
  - Correlated Subquery: The subquery depends on attr. of the outer query.
- When subquery is used, some set comparators are used:
  - IN & NOT IN: Used to check set membership, on "subquery".

    Ex. S IN R

    where R is a nested subquery. (not relation directly.)

    (SELECT ... FROM ...)

Ex. Find name of Sailors who've reserved some boat.

✗ SELECT Sname FROM Sailors WHERE Sid IN Reserves

First, Reserves is multiattr & can't compare with sid (single attr)

Also, Reserves is a relationname, not subquery.

✓ SELECT Sname FROM Sailors WHERE sid IN (SEL sid FR Reserves);

It is simple subquery, since it doesn't use anything from outer query. without outer query also it makes sense.

- For any set comparator, we can't use relation directly, only subquery.

Ex. Find the names of sailors who have not reserved any boat.

SELECT Sname FROM Sailors WHERE sid NOTIN (SEL sid FROM Reserves);

Ex. Find the names of sailors who've not reserved boat 103.

✗ SEL sname ..... WHERE sid IN (SEL sid FROM Reserves R WHERE R.bid <> 103);

This is wrong. Counterexer:

sid	sid	bid
1	1	103
2	1	104
	2	102

SEL sname .... WHERE sid NOT IN (SEL sid ... WHERE R.bid=103);

Ex. Find the names of sailors who've reserved a red boat.

```
SEL Sname FROM Sailors WHERE sid IN (
        SEL sid FROM Reserves R WHERE R.bid IN (
                SEL B.bid FROM Boats B WHERE color = 'red'
        )
);
```

- The queries in case of simple subqueries is executed from inwards to outwards.
- The subquery is calculated once & used again & again.
- Scoping Rule: An attr. without the table mentioned, (sname, not R.sname) is scoped to the nearest enclosing relation which has that attribute.

Ex. Q SEL S.sname FROM Sailors S WHERE S.sid X
SQ1 ← (SEL R.sid FROM Reserves R WHERE R.bid Y
SQ2 ← (SEL B.bid FROM Boats B WHERE B.color = 'red')
);

What will the query output if

(1) X: IN Y: NOT IN
(2) X: IN Y: IN
(3) X: NOT IN Y: IN
(4) X: NOT IN Y: NOT IN

(1) SQ2: Bid of all red boats
SQ1: Sid of all sailors who've reserved some (atleast one) non-red boat.
Q: Sname of all sailors who've reserved some non-red boat.

(2) Q: Sname of all sailors who've booked some red boat.
(3) SQ1: Sid of sailors who've booked some red boat.
Q: Sname of sailors who've booked no red boat.
(4) Q: Sname of sailors who've booked no non-red boats, i.e., either booked no boats or booked only red boats.

Correlated Subquery

Ex: SELECT * FROM R WHERE R.A = (SELECT count(*) FROM R WHERE S.A = R.C);

Here the inner query depends on the attributes of the outer query. So the evaluation of subquery depends on the outer query, i.e., the result of inner query can't just be calculated once & used again & again, rather for each tuple in R, the inner query is supplied 'C' attr. which is used to eval. the inner query for each tuple.

- So, conceptual evaluation for above query will be like:
  For every tuple in R, supply C attr to subq., & evaluate its result & then use in WHERE clause.

Note that, its "For every tuple.." since the subquery is in WHERE clause.

If the nested query was in "HAVING" clause, it would be "For every group" & the supplied attr X would either be aggr. or subset of GROUP BY attr.

Ex. R:

A	B
1	2
3	4
5	6

S:

B	C	D
2	4	6
4	6	8
4	7	9

SELECT * FROM R WHERE A > (SELECT count(*) FROM S WHERE R.B = S.B);

Nested query. Since WHERE clause evaluates for each tuple:

A	B
3	4
5	6

- EXISTS : EXISTS Subquery
  This evaluates T iff the output of subquery is non-empty.
- NOTEXISTS : NOT EXISTS Subquery
  Evaluates True iff the output of subquery is empty.

Ex. R:

A	B
1	2
3	4
5	6
7	8

S:

C	D
2	10
4	12
6	14
8	16

(1) SELECT * FROM R WHERE EXISTS (SELECT * FROM S WHERE B < C);

A	B
1	2
3	4
5	6

(2) SEL * FROM R WHERE EXISTS (SEL count(*) ...);
Output: R (since count(*) never gives empty rel.)

Ex. Find the name of sailors who've not reserved boat 103.
SELECT sname FROM Sailors S WHERE NOT EXISTS (SELECT * FROM Reserves R WHERE S.sid = R.sid AND R.bid = 103);

SELECT ... WHERE (SELECT count(*) ...) = 0;

- UNIQE : UNIQUE Subquery
  Is True iff the o/p of the subquery has no duplicates.
- NOT UNIQUE : NOT UNIQUE Subquery
  Is True iff the output of the subquery has some duplicate.

- ANY or SOME : A op ANY subquery, where A is tuple of attr. & op is comparision operator (=, <>, >, <, >=, <=)
- ALL : A op ALL subquery.

Ex. Find sailors with rating better than

(1) Some sailor with age > 18.

```
SELECT * FROM Sailors S1 WHERE S1.rating > ANY (SELECT
    rating FROM Sailors WHERE age > 18);
```

(2) Every sailor with name 'Horatio'.

```
SELECT * FROM Sailors S1 WHERE S1.rating > ALL (SELECT
    rating FROM Sailors WHERE name = 'Horatio');
```

(3) rating of everybody else.

```
SELECT * FROM Sailors S1 WHERE S1.rating >= ALL (SELECT
    rating FROM Sailors);
```

- A IN subq. ≡ A = ANY subq.
  A NOT IN Subq. ≡ A <> ALL Subq.
  A <> SOME Subq ≡ A <> ANY Subq. = True when there is any/some tuple X in Subq. which is not equal to A. (So always remember ANY means SOME.)

Ex. Find the name of Sailors who've reserved all the boats. (Division opⁿ)

(1)
```
SELECT sname FROM Sailors S WHERE (SELECT COUNT(*)
FROM Boats B WHERE (S.sid, B.bid) IN (SEL sid, bid FROM Reserves))
= (SELECT COUNT(*) FROM Boats);
```

(2)
```
SEL sname FROM Sailors S WHERE sid IN (SELECT sid FROM
Reserves GROUP BY sid HAVING COUNT (DISTINCT bid) = (SEL COUNT(*)
FROM Boats));
```

(3)
```
SEL sname FROM Sailors S WHERE NOT EXISTS ((SEL bid FROM boats) EXCEPT
(SEL bid FROM Reserves R WHERE R.sid = S.sid));
```

(4) SELECT sname FROM Sailors S WHERE NOT EXISTS (SELECT bid FROM Boats B WHERE (S.sid, B.bid) NOT IN (SELECT sid, bid FROM Reserves));

For every sid check that no bid exists such that the record (sid, bid) isn't in reservation.

1	1
2	3
3	7
4	15
5	31
6	63
7	127

A

classmate
Date
Page

Entity Relationship Model

- The very first step of Database design is to "Analyze Requirements".
- Once the requirements are collected, the Conceptual Design is done. This design is both userfriendly, also acts as base for the logical design.
- Relational Model is used for logical design.
- For Conceptual level, one of the famous models is Entity Relationship Model (ER Model).

* ER Model vs Relational Model

- In RM, database is set of relations, and each relation is a set of tuple (unique). Attr. domains are by def^n atomic.

- In ER model, two entities may be same. Attr. can be of different types and may/maynot be atomic. For ER model there is no agreed upon query language & hence is not implemented by any realworld DB.

* - Entity: Any distinguishable object. For ex. Steve, both 'Steve Martin' & 'Steve Jobs' have that name, but its just one of the properties of them & they are distinguishable.

- Relationship: Association of entities.
- Every entity has some attributes which describe the entity.
- Entity set: Collection of similar entities. Similar means, same set of attributes.

Ex: (Ram, 24yrs, 'M') (Shyam, 30Y, 'L') (R1, 10cm, 5cm)
e_1 e_2 e_3

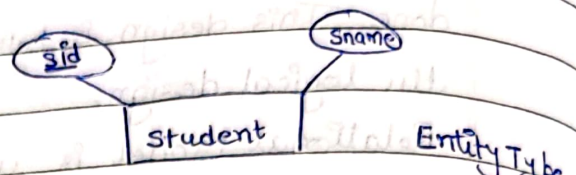
So, we can collect (e_1, e_2) in one set & e_3 in other entity set.

- ER model shares the concept of SK, CK & PK, with Relational Model.

Ex.

Student (sid, sname)

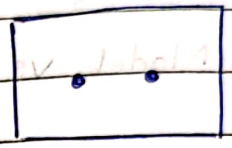
Schema.



Visualization : As Instance

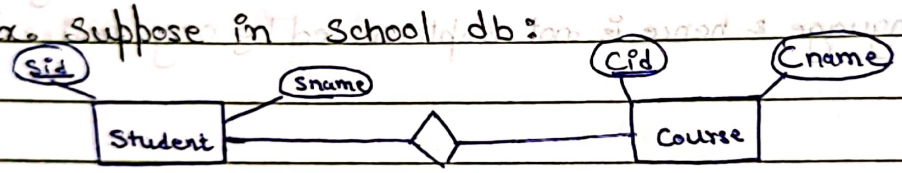
sid	sname
101	A B
102	C D

Visualization : As Entity Set



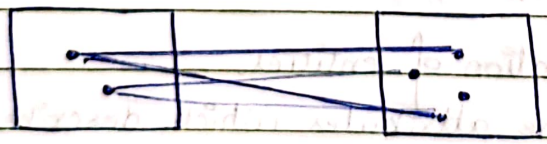
- Entity Type name must be unique
- The concept of domain differs b/w ER model & Relational Model
- Entity in ER model is equivalent to tuple in Relational model.

Ex. Suppose in School db:



Now suppose a student entity <101, A B> has taken the course <CS, DBMS>. This is called relationship.

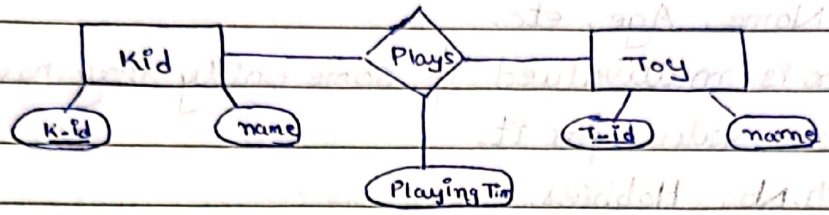
Visualization:



- So, diamond is used to represent relationship b/w Entity Set.
- The set of all similar relationships is called Relationship Set. Similar means b/w same Entity Types.

- Relationships can also have attributes.

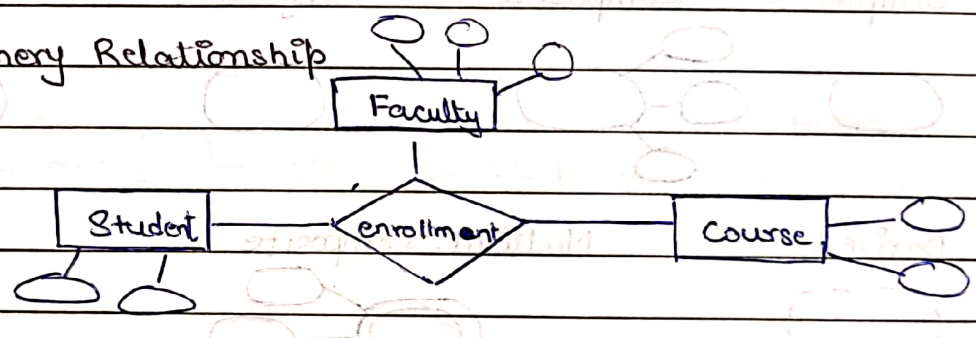
Ex.



So, playing time is property of relationship, not Kid, not Mutoy.

- Degree of Entity Set: No. of attributes of Entity set.
- Degree of Relationship Set: No. of entity sets participating in the relation.
- Cardinality of Entity Set: No. of entities in the Entity set.

Ex. Ternary Relationship



So, degree of relationship : 3

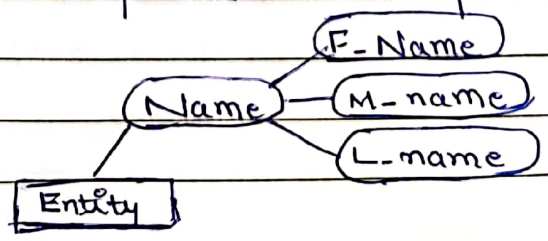
* Types of Attributes

- Simple vs Composite: Simple attr can't be divided further (atomic). Composite attr. can be divided into multiple simple attr.

Ex. Name = First Name + Mid. Name + Last Name.

{ We can choose to def^n Name as composite or simple attr. }

Composite:

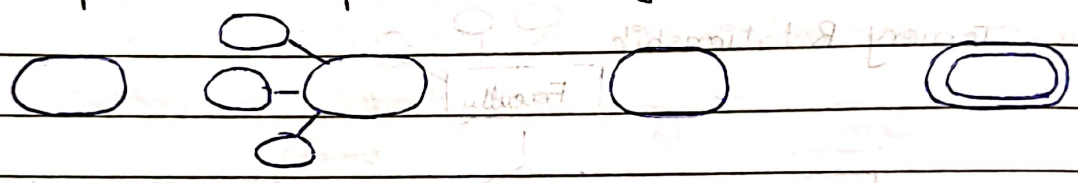


- Single Valued vs Multivalued: An attr. is singlevalued if all entities have a single value for that attr.
  Ex: Name, Age, etc.
  An attr. is multivalued if some entity may have more than one value for it.
  Ex. Ph.No, Hobbies.

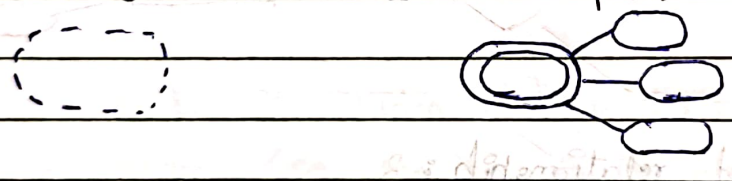
- Derived Attribute: Attribute which can be derived from other attr. of the entity type.
  Ex. Age from DOB of User.
  Profit/Loss from CP & SP.

- Representation:

Simple Composite SingleVal. Multival



Derived Multival. Composite



- Identifying Attr. aka. key Attribute

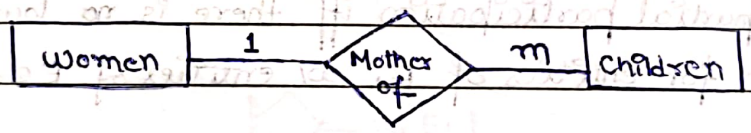
* Types of Relationship
(based on association)

- **One-to-One:** A relationship b/w entity type E_1 & E_2 is called one-to-one iff every entity of E_1 can map to atmost one entity of E_2 & vice-versa.

Ex.

- **One-to-Many:** A rel. b/w E_1 & E_2 is called one-to-many iff every entity of E_1 can map to many (atmost) entities of E_2, but every entity of E_2 must be mapped to atmost one entity of E_1.

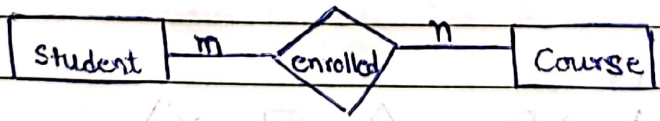
Ex.



1 to many from E_1 to E_2 & many to 1 from E_2 to E_1.

- **Many-to-Many:** A relationship b/w E_1 & E_2 is called Many-to-Many, iff every entity of E_1 can map to many entities of E_2 & vice versa.

Ex.



- One to One Relationship is Partial Function, (i.e. every element of domain is mapped to atmost one elemenent in codomain), from E_1 to E_2, also from E_2 to E_1.

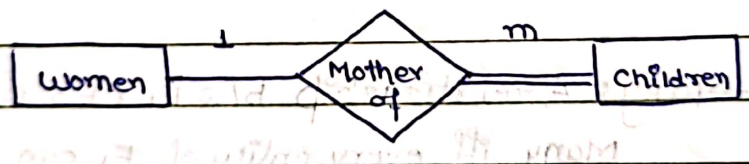
- 1-to-many relationship is Partial function from E_2 to E_1. From E_1 to E_2, it can be called a relation.

- The above classification of relationships put the restriction on the "maximum" (atmost) association of two entities, not on the min. By default an entity from E_1 can associate w/ no entity of E_2 & vice-versa.

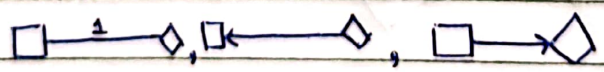
- The restriction can be put on the min. participation of an entity in a relationship:
  - Total Participation: We say that in relationship b/w E_1 & E_2, E_1 is having total participation iff every entity in E_1 associates w/ at least one entity from E_2.

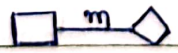
  - Partial Participation: We say that in relationship b/w E_1 & E_2, E_1 has partial participation iff there is no lower limit of association for entities of E_1 w/ entities of E_2.

Ex.

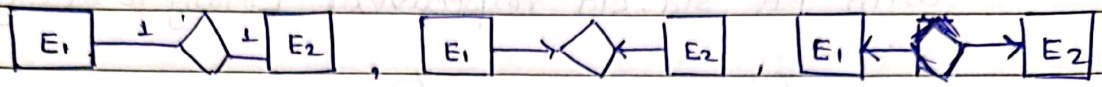


So, a woman can have • 0 or more children.
A child can have at least one & atmost one ⇒ exactly one mother

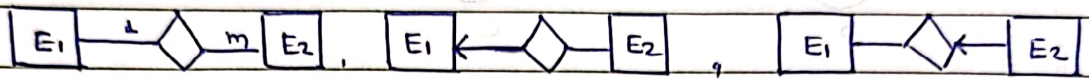
- Representation:
  - One: 

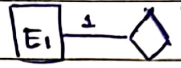
  - Many: 

Ex. From E_1 to E_2 1 : 1

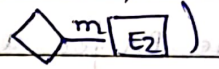


Ex. From E_1 to E_2: 1 : m



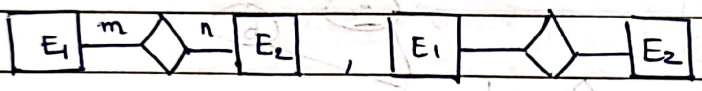
(Arrow towards E_1 means 

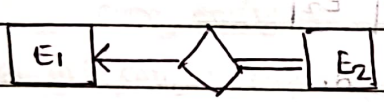
)

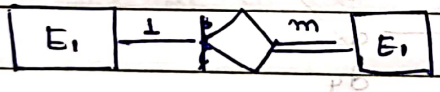
(No arrow towards E_2 means 

)

Ex. From E_1 to E_2 m : n



Ex 

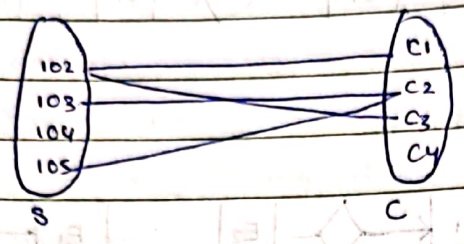
≡ 

1 to many from E_1 to E_2.
E_1 partial participation. E_2 total participation.

* Primary Key / Candidate Key for Relationship

- We know the PK/CK for a Entity Type.
- For a relationship b/w two entities, all you need to describe the relationship is their Primary / Candidate Key.

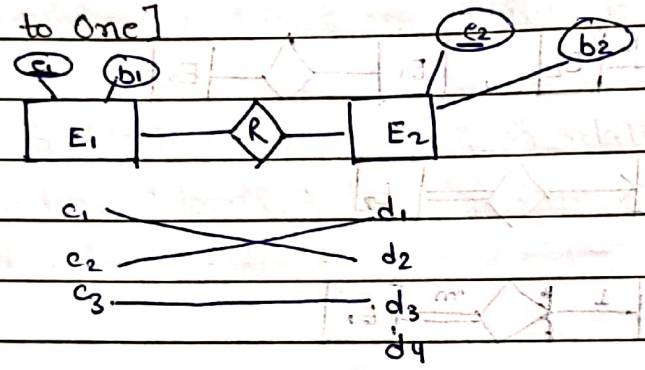
Ex. Suppose Enrolled Relationship b/w Students & Course, with PK sid, cid respectively. [Many to Many]



$R_1(102, C_1)$ $R_2(102, C_3)$ $R_3(103, C_2)$ $R_4(105, C_2)$

- Hence the PK for a many-to-many relationship would be PK of both entities together.

Ex. [One to One]



- Since the association b/w any entity of E_1 & any entity of E_2 is unique (c_1 can only be assoc. with one entity of E_2), the PK for relation could be either entity's PK, c_1 or e_2.

- In one-to-many relationship from E_1 to E_2, the entities of E_2 don't repeat, so PK of relationship would be e_2.

- If the relationship has simple single valued attr., nothing would change. To describe the relation now you need $\langle e_1, e_2, a, b \ldots \rangle$ where $a, b \ldots$ are relationship attr.

- So, possible CKs for rel. R from E_1 to E_2 with e_1 & e_2 as PK; when R is:

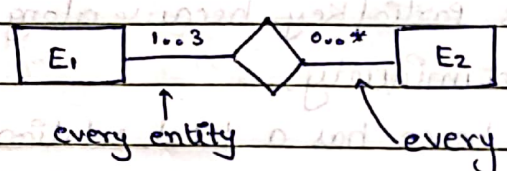
(1) 1:1 : e_1, e_2 (#=2)

(2) 1:m : e_2 (#=1)

(3) m:1 : e_1 (#=1)

(4) m:n : $e_1 e_2$ (#=1)

Note: Alternate notation of "mapping cardinality" & "participation"

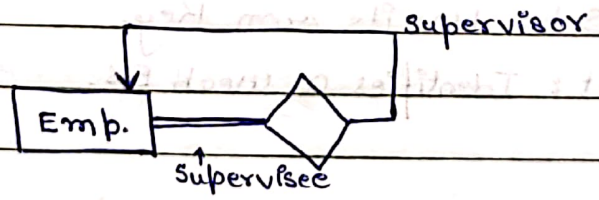


every entity in E_1 can associate with min 1, max 3 entities of E_2.

every entity of E_2 can associate with 0 or more entities of E_1.

* Self Referential Relationship (Recursive Relationship)

Ex. Employee & their supervisors.

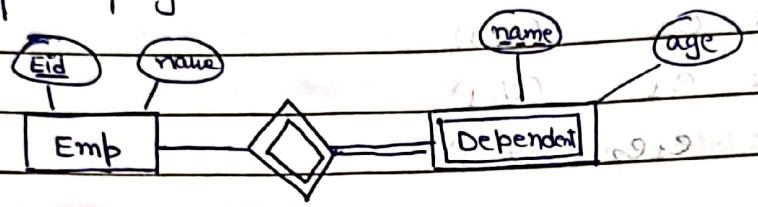


Every employee must have exactly one supervisor.
A supervisor can manage 0 or more employees.

* Weak Entity set

- Entity set without any key.

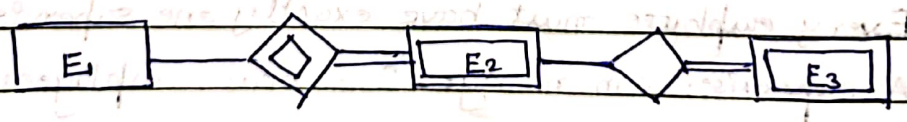
Ex. Employee & their dependents. Employee get id, their dependents don't. To identify the dependent, we require the PK of Employee.



- So, Dependent is weak entity.
- Dependent name is partial key, because along with Eid, it can identify dependent uniquely.
- The Dependent hence has a total relation (always) with Employee. Dependent is weak entity. Employee is owner entity.
- The relation b/w a strong/Owner entity & a Weak entity is called identifying relationship. (Repr. w/ double diamond)
- Weak entity always has total participation in relation w/ strong entity.

- Strong Entity Set: Has its own Key.
  Owner Entity Set: Identifies a weak ES.

Ex.



Here, both E_1 & E_2 are owner ES of E_2 & E_3 respectively, but only E_1 is strong ES.

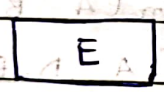
Identifying Relationship

- Primary key of ~~weak ES~~, is the PK of owner entity set + partial key of weak ES.
  (Weak ES by itself doesn't have a Primary Key).

- Misconception: In weak ES, same entity ~~of~~ can appear twice.
  Correction: In weak ES, entities are different, we just can't identify them uniquely.
  Ex. X has dependent <A, 20y> & Y has <A, 20yr>.
  Both <A, 20y> represent two different people. You need association w/ X & Y to uniquely identify them.

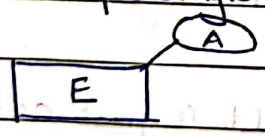
* Converting ER Model to Relational Model

- Entity Set



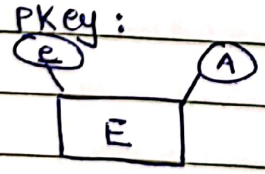
⇒ Make Table. E | | | |

- If simple Single Val Attr.



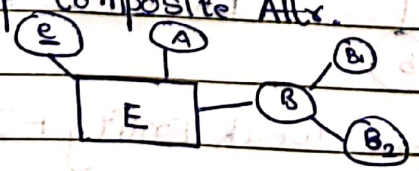
⇒ E (A,)

- If PKey:



⇒ E (e, A)

- If Composite Attr.

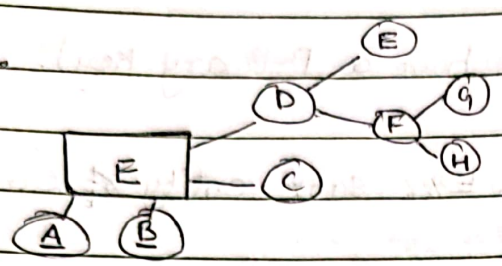


⇒ E (e, A, B1, B2)

Store the atomic attr.

- The composite attr. info is lost. It can be derived later.

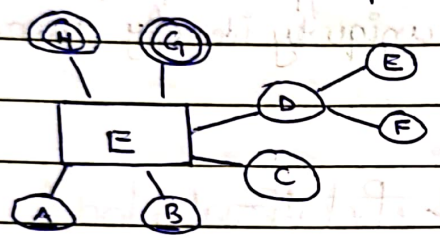
Ex.



E (A, B, C, E, G, H)

{AB is PKey}

- If Multivalued attr.: For each multivalued attr. of E, we create a separate table, with E's PKey & the attr.

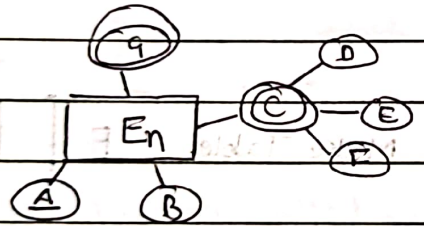


⇒ E (A, B, C, E, F)

E_G (A, B, G)

E_H (A, B, H)

Ex.



⇐ En (A, B)

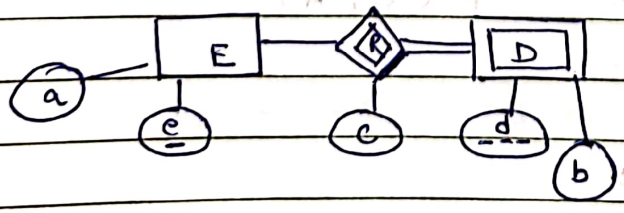
Enc (A, D, E, F)

Eng (A, G)

A is FK in Enc & Eng

- Derived attr. is not stored. It is just calculated from source attribute.

- Weak Entity (& Identifying Rel.)



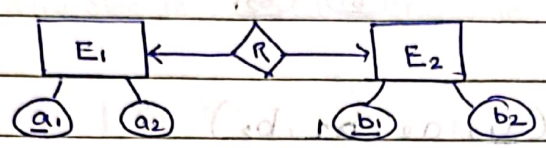
⇒ E (e, a)

DR (e, d, b, c)

So, a table for Weak Entity + Identifying relationship, w/ PK as (PK of owner + Partial key of weak).

1:1 Mapping Cardinality

(1) Both partial participation.

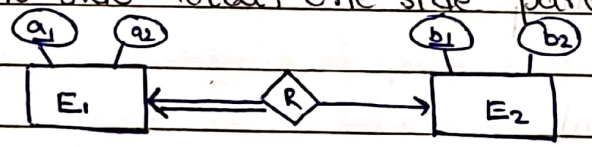


☆ Sol1: $E_1(\underline{a_1}, a_2)$ $E_2(\underline{b_1}, b_2)$ $R(a_1, b_1)$
{or $R(a_1, \underline{b_1})$}

Sol2: $E_1R(\underline{a_1}, a_2, b_1)$ $E_2(\underline{b_1}, b_2)$
{This sol. is only allowed if NULL values are allowed, since there can be NULL values in E_1R, for attr. b_1.}
{OR $E_1(a_1, a_2)$ $E_2R(\underline{b_1}, b_2, a_1)$}

Overall Sol1 is better design in general.
{Sol2 is better in terms of #relations, if NULL allowed}

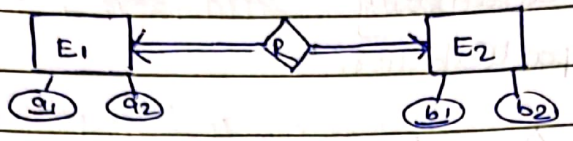
(2) One side Total, One side partial participation



Sol1: Same as above Sol1.

☆ Sol2: $E_1R(a_1, a_2, \underline{b_1})$ $E_2(\underline{b_1}, b_2)$
This is better design, as NULL values are not possible due to total participation of E_1.
{If E_2 had total & E_1 had partial
$E_1(\underline{a_1}, a_2)$ $E_2R(\underline{b_1}, b_2, a_1)$}

(3) Both side Total Participation



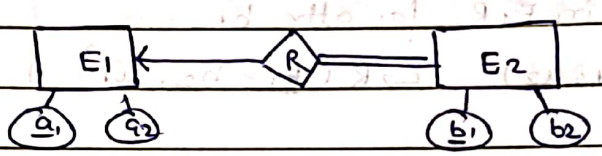
★ Sol. $E_1 R E_2 (a_1, a_2, b_1, b_2)$

{ $CK_s : \{a_1, a_2\}$ }

This design is best, since no Null values on either side.

- 1 to Many Mapping Cardinality

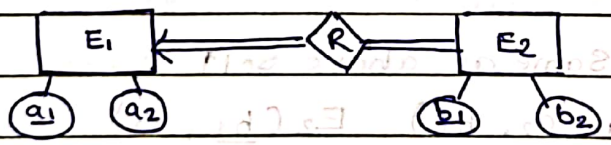
(1) One side has partial, many side has total participation.



★ Sol. $E_1 (a_1, a_2)$ ∴ $E_2 R (b_1, b_2, a_1)$

{ Best design. a_1 can never be NULL, since total participation of E_2. }

(2) Both side Total.



Sol. 1. Is this possible design? Is it good design?

$E_1 R E_2 (a_1, a_2, b_1, b_2)$

Yes, it is possible design. No null problem.

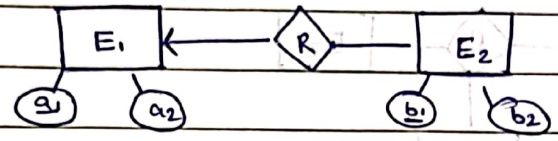
No, it isn't good design (3NF).

$b_1 \rightarrow b_2 a_1 a_2$ $\quad a_1 \rightarrow a_2$

Not in 3NF.

★ Sol.2. $E_1(\underline{a_1}, a_2)$ $E_2R(\underline{b_1}, b_2, a_1)$

(3) Many side is Partial (one side can be either)



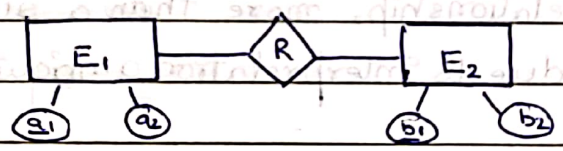
★ Sol. $E_1(a_1, a_2)$ $E_2(b_1, b_2)$ $R(b_1, a_1)$

{Since E_2R merging would have null problem for a_1 in E_2R as FKey.}

Sol.2 [If Null Values explicitly allowed]

$E_1(a_1, a_2)$ $E_2R(b_1, b_2, a_1)$

- Many - To - Many



In all variations of participation:

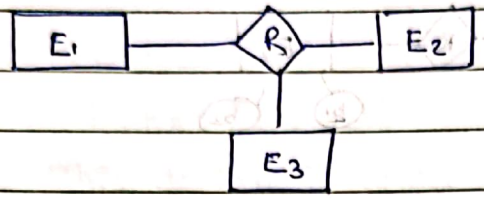
★ $E_1(a_1, a_2)$ $E_2(b_1, b_2)$ $R(a_1, b_1)$

Note: First priority: Good Design (3NF + No NULL Problem)

Second: Minimum no. of tables.

* N-ary Relationships (N>2)

Ex. Ternary Relationship



- If say all sides are 'many' then all combinations are possible & PK of R would be (e_1, e_2, e_3).

- If, say E_3 has '1'. This means for a particular (E_1, E_2) pair, we have at most one E_3.
  So, $e_1 e_2 \rightarrow e_3$.

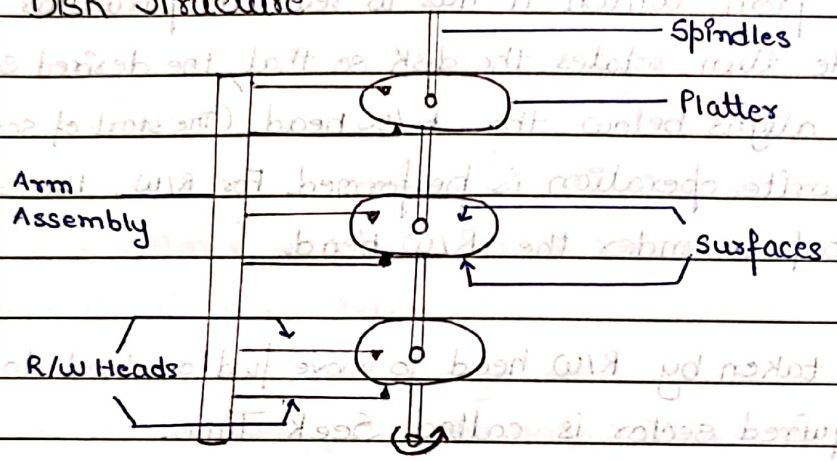
- In non-binary relationship, more than a single '1' side is not allowed, due to interpretation ambiguity.

classmate
Date
Page

Magnetic Disk

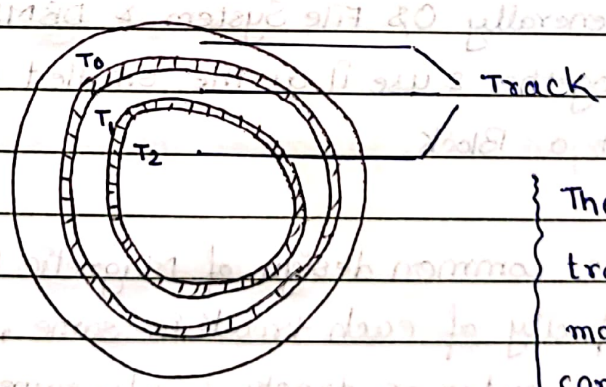
- Disk is secondary storage, an IO device used for persisting data.
- The key reason to study Magnetic disk is that different areas of the disk take different access time.
- In Disk, bits are stored as north & south pole.

* Disk Structure



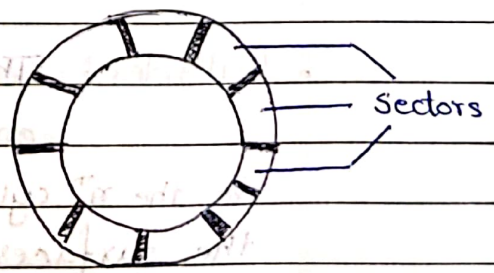
- Each surface is logically divided as:

Tracks are numbered starting from outermost. $T_0, T_1, T_2 \ldots$



The black areas are track gaps, so that magnetic interaction can be avoided.

- A track is further divided as:
(The black areas are sector gaps to avoid magnetic interactions)



Note: Track gaps & sector gaps are too small to add any overhead.

- The "sector" is the smallest unit of access of Magnetic Disk, i.e in a single read / write the disk accesses one complete sector as a whole.

- For a sector to be read:
  (1) The R/W head moves fwd or backward to align itself to the track from which it has to read the particular surface.
  (2) The spindle then rotates the disk so that the desired sector on the track aligns below the R/W head. (The start of sector)
  (3) Read or write operation is performed. For R/W, the sector just has to pass under the R/W head.

- The time taken by R/W head to move fwd or bwd to align with the required sector is called Seek Time.
- The time taken by the sector to align below the R/W head is called Rotational Lateney (or just Lateney).

- Block: Generally OS File System & DBMS s/ws club 2-3 sectors together & use it as the smallest unit of R/W, and call it a Block.

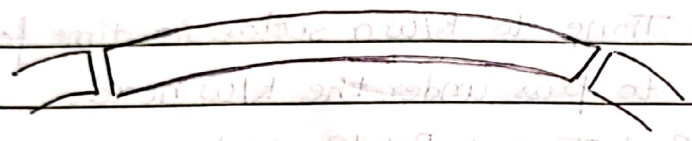
- Note: The common design of Magnetic Disk is such that the capacity of each track is same, and the capacity of each sector on tracks is also same.

- Cylinder: The parallal tracks of multiple surfaces alinged on top of each other is called a cylinder.
  So, the nth cylinder would contain ~~all the~~ n^{th} tracks from all the surfaces.

- Although all R/W head move simultaneously & are vertically in same position, only one of them is active at a time.

- Since for any area to be read by R/W head, it has to just move under a R/W head, for a track to be read a single rotation of the surface is enough.

* Sector



- Each sector's start & end is marked by putting some special pattern at each end.

- Each sector along w/ the data, also stores a Checksum value, used to verify if the sector is corrupt or not.

- Apart from actual data, any info in the sector is called as "Disk Formatting".

- The checksum calculation is one of the responsibilities of Disk Controller.

* Disk Performance

The main measures of the disk performance are:

(1) Capacity
(2) Access Time
(3) Data Transfer Rate.
(4) Reliability.

- Disk Capacity
  - ⇒ No. of surfaces x #tracks/surface x #sectors/track x sector capacity.
  - ⇒ No. of cylinders x #surfaces x #sectors/track x sector capacity.

- Access Time
  - When a disk is operational it is constantly rotating at a constant velocity.
  - Transfer Time: Time to R/W a sector, i.e. time for sector to pass under the R/W head.
  - Access Time = Seek Time + Rotational latency
  - Sector Access Time = Access Time + Transfer Time

- Data Transfer Rate (Throughput)
  - How much data (bytes) per second can be read or written.

Ex. 1000 cylinders. 10 sectors/track. Head assembly at cylinder 0 initially. Head moves 10 μs/cylinder. Disk rotates at 100 rps. Find the average time to read a randomly chosen byte.

Even to read a byte, you'll have to read a complete sector.
On avg, you'll have to move 1000/2 = 500 tracks.
On a track, on avg you have to move half a track, so half a rotation.

Seek Time = 10 μs x 500 = 5 msec.

Rotational Latency = $\frac{1}{2}$ x 1 Rot time = $\frac{1}{2} \times \frac{1}{100} = \frac{5}{1000}$ sec = 5ms

For a sector to be read, it'll have to move below R/W head.

10 sector $\equiv$ 1 rotation $\equiv \frac{1}{100}$ sec.

$\Rightarrow$ 1 sector $\equiv \frac{1}{1000}$ sec = 1 ms.

$\therefore$ Avg time to read a byte = 5 ms + 5 ms + 1 ms

$\Rightarrow$ 11 ms.

Note: • Seek Time is always taken as avg, unless you have to read specific sector on specific track, then calculate.

• Rotational latency is also taken as avg.
(a.k.a. Latency) (a.k.a Rotational Delay)

Ex. Consider a disk having 6 platters, with avg. seek time of 4 ms, rotation speed of 15000 rpm & 512 B sectors with 500 sectors/track. The track-to-track access time is 1 ms. Suppose we wish to read a file consisting of 2500 sectors for a total of 1.28 MB. Estimate the total time of transfer when:

(1) The file is stored consecutively on a cylinder.

(2) The file is stored consecutively on a platter.

(3) The file is stored randomly throughout the disk.

Rot lat = 2 ms

(1) # sectors in a cylinder = 6 x 2 x 500 = 6000 sectors.

So, we only need 5 surfaces in a cylinder to store 2500 sectors.

So, seek time = (only once req. to get to the cylinder) = 4 ms.

rot. latency = (5 times for 5 surf) = 5 x 2 = 10 ms.

transfer time = 5 x 1 rot time = 5 x 4 ms = 20 ms.

So, total transfer time = 34 ms.

(2) One track capacity = 500 sectors.

tracks on a platter for the file (2500 sectors) = 5.

So, seek time = ST to get to get to first ~~sector~~ track + 4 × ST to get to next track

$\Rightarrow 4 + 4 \times 1 = 8$ ms.

rot. latency = $5 \times 2 = 10$ ms.

transfer time = $5 \times 4 = 20$ ms.

Total time = $(8 + 10 + 20) = 38$ ms.

(3) In worst case for each of 2500 sectors of the file: $4 + 2 + 4/500 = 6 + 4/500$

$\therefore$ Total time = $\cancel{2500}^{5}\left(\frac{3004}{\cancel{500}}\right) = 15020$ msec

$\Rightarrow 15.020$ secs.

* Two sectors are close (in decreasing order) when:

(1) They are on same track. (single rotation).

(2) They are on same cylinder (single seek)

(3) They are on adjacent cylinder (low seek time).

Two data records are called close when (decreasing order):

(1) They are on same block.

(2) They are on same track.

(3) On same cylinder.

(4) On consecutive cylinder.

So, data is stored on disk as:

(1) Try to put in single sector.

(2) If larger, try to put in a single track.

(3) If larger, try to put in a single cylinder.

(4) If even larger, go to the next cylinder.

* Disk Addressing

- Based on the concept of ~~closeness~~ closeness of sectors, we come up with Cylinder-Head-Sector addressing scheme.

- By default, the outermost track (& outermost cylinder) starts w/ index 0.

- Since sector is the most atomic unit of access in disk, addresses are given to each sector.

- Cylinder-Head-Sector Addressing: In CHS scheme, each sector is addressed with a three tuple <C,H,S>, where C is the Cylinder no. (starting w/0), H is head/surface no. (starting w/ 0) & S is the Sector no. on particular head (starting from 0).

  Ex. <20, 15, 2>

- 2nd sector on 15th surface of 20th cylinder.
  {Think of it as D[20][15][2]}

- Logical Block Addressing: Starting from <0,0,0>, it gives addresses 0, 1, 2, ... to each sector in the disk.

  Ex. 450 means 450th block on the disk.

Ex. 3 platters, 3 cylinders, 4 sectors / track. {6 surfaces}

(CHS) <0,0,0> $\longrightarrow$ $0 \times (6 \times 4) + 0 \times 4 + 0 = 0$ (LBA)

(CHS) <1, 5, 2> $\longrightarrow$ $1 \times (6 \times 4) + 5 \times 4 + 2 = 46$ (LBA)

(LBA) 62 $\longrightarrow$ $\langle 62/24,\ (62\%24)/4,\ (62\%24)\%4 \rangle$

$\Rightarrow \langle 2, 3, 2 \rangle$

* Disk Scheduling

- Modern OSes are multiprocessing systems. So, multiple processes in the system can be in the ready queue.

- Hence more than one process (one-after another) can issue disk requests, to read from various sectors.

- So, the Disk requests are collected by the OS in a queue. Now the OS can decide the order in which these requests will be executed, in order to get good performance.

- When we have sector R/W, the most important component is the Seek (placement of R/W head over required cylinder) since the head movement is very slow compared to rotation of disk. So, overall, through different scheduling Algorithms, we want to minimize the overall head movement, i.e. minimize Seek time.
  Hence all disk algo. only consider the cylinder no. of a Sector R/W request.

- FCFS: Service the request in the order they arrive in the disk queue.
  Problem w/ FCFS is that it doesn't care if there is a request nearby which can easily be serviced, it will service distant request if it is next in queue.

- Shortest Seek Time First: From current head position, service the request closest, to the head position.
  The problem w/ SSTF is that it may starve requests on one side of the surface.

Starved

- Scan / Elevator : Given the initial head position & current direction of movement (up or down), move to the very end on the way servicing the request, reverse the direction & go to the very end, servicing all the requests on the way. The problem w/ this algo. is that we have to go till end even if no request is pending in that direction.

- Look: Same as Scan algo., but in either direction, only go till the last ~~address~~ cylinder requiring service.
  - The problem w/ both Scan & look algo. in case of dynamic request queue where requests keep coming in is that the request for same cylinder may get service multiple times before other cylinders getting serviced.

- C-Scan: Given the initial head pos. & current direction, go to the very end & service all requests in path. Now reverse the direction & go till very last. Reverse direction & now service remaining requests.

- C-Look: From current pos, go to ~~the~~ till ~~the~~ the last request in direction of movement servicing requests on the way. reverse direction & go to the last req. in other direction. Reverse direction, & serve remaining requests.

classmate
Date
Page

File Organization & Indexing

- Block: Some consecutive sectors within a track of a disk.

- File Organization concerns w/ storage of Database data on the disk.

* File Organization

- For a DB w/ relations $R_1, R_2, \ldots$, we store one relation in one file each.

- File is a logical concept. It is stored in blocks of the disk. So, file is a sequence of consecutive blocks.

- So, the records of a relation R are divided into blocks.

$$R: \underbrace{r_1 \; r_2}_{B_1} \; \underbrace{r_3 \; r_4}_{B_2} \; \underbrace{r_5}_{B_3}$$

{Block Size ≥ Record Size}

So, when, say r_4 is to accessed, block B_2 will be transferred to the main m/m. Although for the disk, sector is smallest unit of access, DBMS uses block as unit of access.

- Access Cost: # blocks to be transferred from disk to main m/m in order to access desired records.

Note: Sometimes "Page" is used to refer to "Block". Not to confuse w/ OS Paging concepts. Block & "Virtual m/m page" are not related concepts.

Ex. On a disk, sector size is 512 B, #sectors per track = 50, #tracks / surface = 2000. Which of the following is/are valid block size:

(A) 256 B (B) 2048 B (C) 51200 B.

Track size = 512 B × 50 = 25 KB.

(A) 256 B < sector size.

(B) 2048 B = 4 × sector size ∴ Valid block size.
& 2048 B < Track size

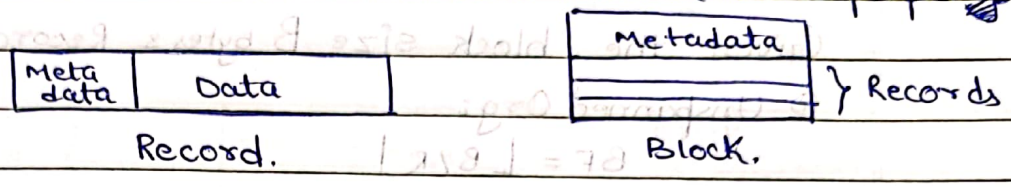
(C) 51200 B = 100 × sector size. ∴ Invalid
but 51200 B > Track size

Block size must be multiple of sector size & less than track size.

- Types of Records: Within a relation (file), there are two types of records:
  - Fixed length Records: All records of the relation are of fixed length.
  - Variable length Records: All records of the relation may have variable sizes.

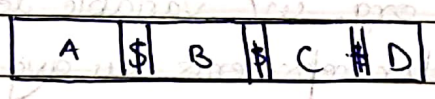
- Most of the time we only want a particular field within a record.
  - In Fixed length record, we can easily find the fields just by knowing the size of each field. But where is the info. about field sizes is stored? This info., such

as, size of fields, how many fields, last access time of the record. ~~are stored within the record (Record property)~~

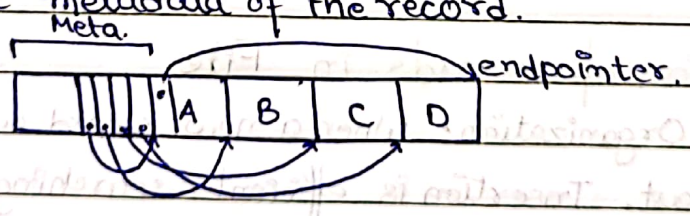


- (a) Last Access Time: Within a record. (Record Property)
  (b) # fields in record: Store once per block (file property).
  (c) Size of fields: If the size of fields can vary over time, then store within record metadata, else if the size of fields never vary, then store in block meta-data.

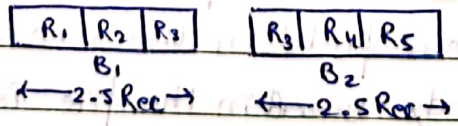
- One other approach to mark field boundaries of Variable length record is to use special marker symbol.



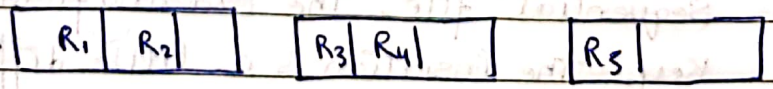
- Alternate sol2 is to store pointers (offset) for each field in the metadata of the record.



- Spanned Organization: Allows for a record to be stored in two portions, in two blocks.



Unspanned: A record can be put in a block as whole or none.



Some space may get wasted per block.

- Blocking Factor: The no. of ~~blocks~~ records that can be put in a block.

Given the block size B bytes & Record Size R bytes.

① Unspanned Org:

$$BF = \lfloor B/R \rfloor$$

② Spanned Org:

$$BF = B/R$$

- For accesses, Unspanned organization incurs lower cost of 1 block access for 1 record, compared to spanned organization which can have 2 block accesses for 1 record.

Note: • By default, Unspanned organization w/ Fixed length records is used.

- For spanned org. w/ variable length records, the blocking factor represents the avg no. of records per block.

- Organization of records in File

- Heap Organization: When a new record comes in, put it in the last. Insertion is efficient, searching is hard. Suppose a file has B blocks, then to search, on avg. $\Theta(B)$ blocks.
- Ordered Org. based on some field: The records can be ordered based on some field in the records. The field used to order records ~~based~~ in the file is known as "Ordering field", & a file using such org. is called "Ordered / Sequential" file. The ordering field may/maynot be key. The insertion is a little difficult but searching based on ordering field becomes better. $\Theta(\log B)$

Binary search

for a file with B blocks. From point-of-view of non-ordering field, it still looks like Heap organization.

Orgniz.	Searching Based On	Search Method	Avg. Block Access Required
Heap	Any field	Linear / Sequential	B/2
Sequential	Ordering field	Binary Search	log (B)
Sequential (a.k.a. Ordered)	Non-Ordering f.	Linear / Seq.	B/2

- Block Address: The address of first sector of the Block is used as Block Address.

- Record Address: A record can be identified by the <block addr, offset> pair.

* Indexing

- Index is a way to make accessing data more efficient.
- Index entries consist of <Search Key, Block Pointer (or Record Pointer)> pairs, and they are arranged in sorted order of search key (index field) although the data file may not actually be sorted by that field.
- Why is Indexing more efficient way to access data:
  (1) Index structure is significantly smaller in size compared to the data file, so it takes less no. of block access (IO oper.) to search index.
  (2) Creating Index on a commonly searched field makes it efficient to search based on that field, $O(\log B)$ for B no. of blocks.

- Data File ≡ Main File ≡ Relation ≡ Indexed File ≡ File
- Index File ≡ Index

Ex. Student (sname, sid, marks, class):
How many indexes can be created simultaneously on this relation?
4. One index by "sid", one by "sname", by "marks" & by "class".
{Multiattr. indices are Advance level concept}

- Search Key: The field of the relation, on which Index is created.

- Indexes
  - Ordered (Static)
    - Single Level
    - Multilevel
  - Tree based (dynamic)
    - B Tree
    - B+ Tree.

Note :. There'd also hash based indexes, which are not sorted.

- By default index files are sorted.
- We assume that index is created on single field.

* Single Level Ordered Index.

- The index file is ordered (sorted) based on some search key. The data file may/maynot be ordered based on the 'search key', or maynot be ordered at all, (i.e. Heap or Sequential file organization).

* Primary Index

- A relation R is physically stored on the disk, either as a heap file, or as a sequential file based on some ordering field.

- Primary index can only be created on sequential file.
- If data file is sorted on a Key attr. A, then index on A is called Primary index.

- So, Primary index is an index created on a field F, where F is a Key attr. of the relation & the relation must be physically stored as a sorted (sequential) file, ordered by F.

- Hence to create a Primary Index for a relation R, first of all R must be physically ordered by a Key field F, then we can use this field F to create Primary Index.

- A field F can be used for Primary index only if :
(1) It is ordering field of R physically.
(2) It is Key field of R
∴ (Ordering + Key) field. ≡ Ordering Key field.
So, F must be the Ordering Key field of R to be candidate for Primary Index.

- Atmost one field F in relation R can be the ordering key field of R, at a time.

- So, at a time, atmost one Primary Index can be created on a data file.

- Since Index file is ordered on search key, & in Primary Index is on a ordering key field, both the data & index file have the same ordering.

- Dense Implementation: For every record in the relation (data record) there is an Index record. So for every data record, in the index file:

  <search key, pointer>

Now, this pointer can be both:

(1) Block pointer to the block of the record.

(2) OR Record pointer {Block Ptr + offset} to the record.

By default, its Record Pointer.

Note: For questions, if Record pointer is mentioned, take record pointer for Dense Implementation.

- Sparse Implementation: For every block in the data file there would be one Index record. So, for every data file block, the index file would have:

  <search key, pointer>

where search key for a block is the value of ordering key field of the "anchor record / Block anchor" for that block. Generally first record of the block is chosen as the block anchor. So the pointer can only be block pointer.

- Searching: To search in single level Primary Index file having I blocks, we first perform Binary Search on the Index blocks, then atmost one more access to the data block.
So, no. of block accesses $= \lceil \log_2 I \rceil + 1$

- Although Dense implementation is possible, but by default Primary Index has sparse implementation.

Note: In ques. if the implemenentation detail of Primary Index not mentioned then Sparse implementation.

Ex. Data file has:

No. of blocks = 9500 Ordering Key field len = 15 B

~~Block size = 4KB~~. Block Ptr length = 6 B.

Block size = 4KB.

For the Primary index on the sequentially ordered data file (ordered on key field) find:

① No. of records
② Size of an entry.
③ Blocking Factor
④ No. of blocks
(By default sparse impl.)

⑤ Max. no. of block access to search record based on ordering key field
(a) w/ index.
(b) without index.

Size of each entry = 15 B + 6 B = 21 B.

No. of records in index file = # blocks of data file = 9500.

BFr of Index file $= \left\lfloor \frac{4 \times 2^{10} B}{21 B} \right\rfloor = 195$

index blocks $= \left\lceil \frac{9500}{195} \right\rceil = 49.$

Max # block accesses to search record

(a) w/ index $= \lceil \log_2 49 \rceil + 1 = 7.$

(b) without index $= \lceil \log_2 9500 \rceil = 14$

* Clustering Index

- If the data file is ordered physically by a Nonkey field F, then the index created using F is called Clustering Index.

- So, Clustering index can be created for R based on field F, only if:
  (1) F is non-Key attr of R.
  (2) R is physically ordered by F.

So, (Ordering + NonKey) attr. ⇒ Ordering NonKey field
= Clustering field.

∴ So, If Clustering Index is created for R then R has a Clustering field.

- A relation R, ordered by a non-key attr, is called Clustered file & the attr is called Clustering field.
Now if you create index on the clustering file, it is called Clustering Index.

- Clustering Index can't be created on Heap file, since such files aren't physically ordered.

- At any time, a data file can be ordered by at most one field, so, at a time atmost one Clustering Index can be created for a data file.

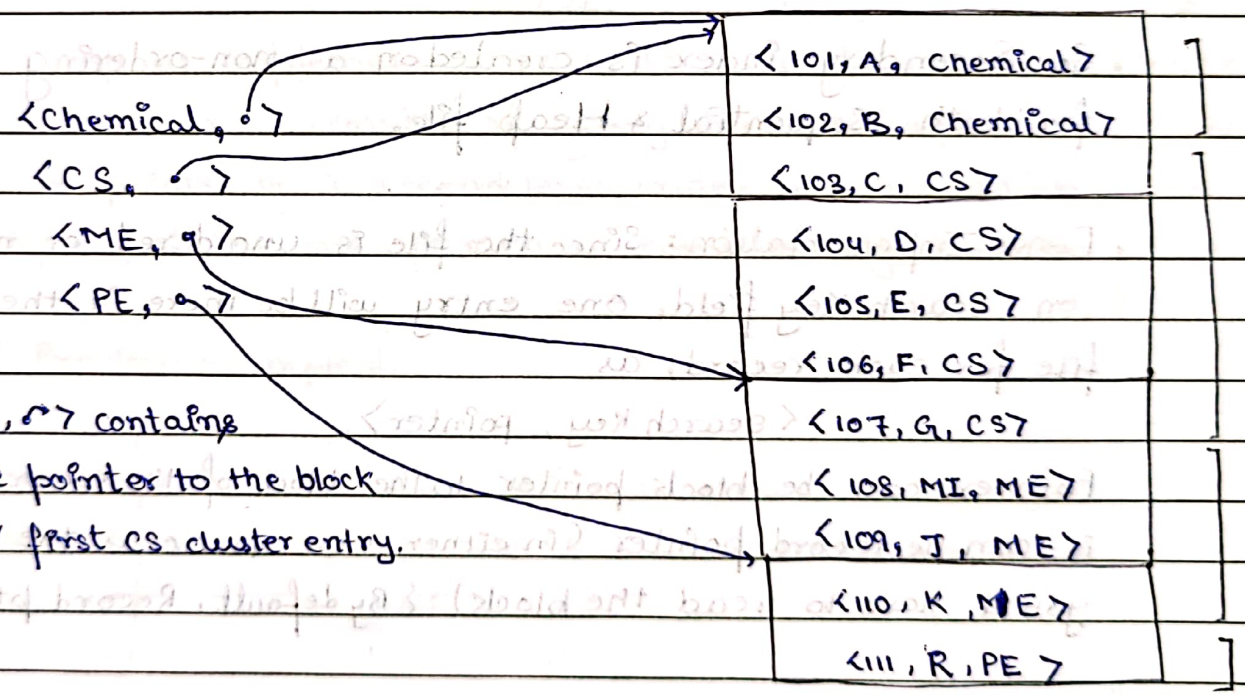
Ex. On a file can we create both primary & clustering index simultaneously?
No. At a time file can be ordered by at most one field, and the same field can't be key & nonkey both.

- Sparse Implementation : Since file is ordered by non key field, there would be one entry per cluster in the index file :

  < search key, pointer >

  where search key will be a value from clustering field, and the pointer will be block pointer which points to that block which contains the first data record w/ the respective search key value.

Ex. Student (Sid, Sname, Subject)



The reason to use block ptr & not record ptr is that the cluster can span to more than one block & if we have record ptr we can't go to next block, but block ptr allows us to check i^{th} block's header & go to $(i+1)^{th}$ block.

- So, "clustering index" is "Sparse index"
- If a file is sequential (by defn on some ordering field) then either a primary index {if file sequenced on key attr} or clustering index {if file sequenced on non-key attr} is possible, not both.

* Secondary Index.

- If a file is stored physically on the disk, such that the field B is not stored in ordered manner, then if we create an index on B, it is called Secondary Index.

- 'B' can be both Key or non Key:
  (1) If B is Key, it is called "Secondary Index on Key".
  (2) If B is non-Key, it is called "Secondary Index on NonKey"

- So, Secondary index is created on a non-ordering field for both Sequential & Heap file.

- Dense Implementation: Since the file is unordered or not ordered on search key field, one entry will be there in the index file for each record, as
  <search key, pointer>.

Pointer can be block pointer to the block of the record, or it can be record pointer (in either case to access the record you'll have to read the block) {By default, Record ptr}

Ex. Assume the following is Physical order:

acct_no.	branch_name	balance	
A101	Dow	500	B_1
A102	Per	400	
A201	Bri	900	B_2
A 215	Mia	700	
A 217	Bri	750	B_3
A222	Red	350	
A. 305	Rou	750	B_4

What Indexes can be created on this file?

(1) acct_no is ordering key field. So, Primary Index can be created on it.

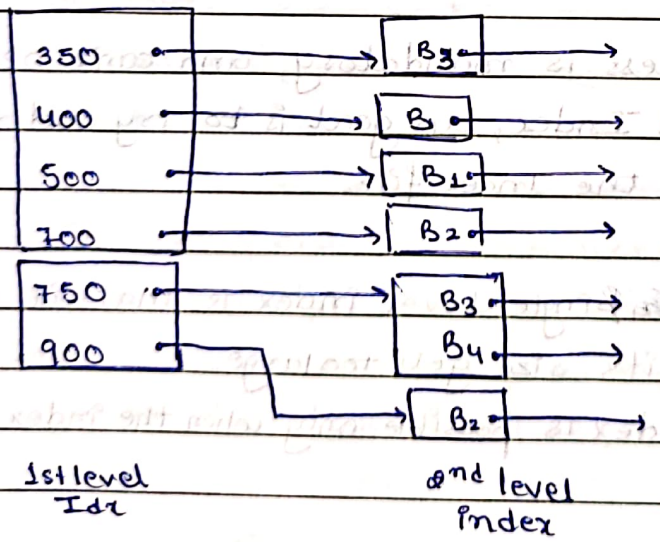
(2) branch-name is non-ordering non-key field. So, Secondary Index on non-key field.

(3) Similarly, secondary index on non-key field for balance field.

- There can be various ways to store index records for Secondary Index on non-key field (due to duplicate search key value possibility).

The prefered implementation is to have two level of index. First level index consists of <Key value, block ptr>. The block ptr points to a second level index block which has multiple pointer one for each record having that key value.

Ex. [Previous example].



- So, secondary index can be created on unordered field of the file.

Ex. How many secondary indices can be created on file w/ n attr: (1) min (2) max?

At min: n-1 {Sequential file}
At max: n {Clustering file}

- Secondary Index ≡ Unclustered Index.

* Multilevel Ordered Index

- So far we've seen Singlelevel Ordered index, with I index blocks & B data blocks, in general, no. of block accesses for a search is:

$$\lceil \log I \rceil + 1$$

index blocks — data block

The one data block access is mandatory, and can't be avoided. With Multilevel Ordered Index, our goal is to try and reduce the cost of searching the Index file.

- Another problem w/ ~~Fn~~ Single level index is that it can't be cached in m/m, if its size gets too large.
So, caching Singlelevel index is possible only when the index size is small.

- Although Multilevel ~~paging~~ indexing ~~doe~~ isn't done so that Index file is stored in one ~~file~~ block, but if we do it, we do it so that the top level fits within one block.

- All our index types from single level indexing are such that the index file has ordered unique entries on the search key value. So the 1st level index itself is acting like a "file w/ ordering key field", so we can create a primary index on first level index. So, we can repeat this strategy either until the desired no. of levels are reached or the top level index fits in one block.

- The first level: (primary or clustering or secondary)
  - #entries = # data blocks or # clusters or # records
  - could be dense or sparse

The ith level ($i \geq 2$): (Primary)
- #entries = # $i-1^{th}$ level index blocks.
- could only be sparse. (otherwise there's no point of multilevel)

- So, for any data file, we have three options
  - No index.
  - Single level index
  - Multilevel index.

Ex. Consider a disk w/ Block Size = 512 B. A block pointer is P = 6 B long & a record ptr. P_R = 7 B. A file has r = 30000 records of fixed length. Each record has following fields: Name (30 B), Ssn (9 B), Dept (9 B), Addr (40 B), PhNo (10 B), DoB (8 B), Sex (1 B), Job-code (4 B) & Sal (4 B). An additional byte is used as deletion marker.
{Navathe 7th ed., Q. 17.18, Pg. 648}

(a) Calculate record size R in bytes.
(b) Calc. BFr & no. of file blocks b assuming unspanned org.

(a) R = 116 B

(b) $BFr = \left\lfloor \frac{512}{116} \right\rfloor = 4$ records $\qquad b = \left\lceil \frac{30000}{4} \right\rceil = 7500$ blocks.

(c) (i) Index record len = 9B + 6B = 15B, bfr_i = $\lfloor \frac{512}{15} \rfloor = 34$

(ii) #index records = 7500

1st level index blocks = $\lceil \frac{7500}{34} \rceil = 221$.

(iii) # 2nd level indx rec. = 221

2nd level index blocks = $\lceil \frac{221}{34} \rceil = 7$.

3rd level idx recs = 7

3rd level idx blocks = $\lceil \frac{7}{34} \rceil = 1$.

∴ no. of index levels = 3.

(iv) Total no. of index blocks = 229.

(v) #block accesses = 3 + 1 = 4.

index blocks, data block

(d) The index type is : Secondary on Key field.

(i) Idx record len (1st level) = 9B + 7B = 16B

bfr_i = $\lfloor 512/16 \rfloor = 32$

(ii) # indx records = # data records = 30000

1st leve Idx (I_1) blocks = $\lceil \frac{30000}{32} \rceil = 938$.

(iii) Bfr from second level onwards = $\lfloor \frac{512B}{9B+6B} \rfloor = \lfloor \frac{512B}{15B} \rfloor = 34$

I_2 entries = 938.

I_2 blocks = $\lceil 938/34 \rceil = 28$.

I_3 entries = 28.

I_3 blocks = 1

(iv) Total # index blocks = 938 + 28 + 1 = 967

(v) #block accesses = 3 + 1 = 4

Idx blocks, data blocks

(e) Type of Index: Secondary Index on non key.

Impl. Type: 2 level implementation.

unique val = 1000 # records per unique val = $\frac{30000}{1000} = 30$

(i) Index record len ⇒ 9B + 6B = 15 B (Key, block ptr) 7 B (record ptr)

Bfr ⇒ $\left\lfloor \frac{512}{15} \right\rfloor = 34$ $\left\lfloor \frac{512}{7} \right\rfloor = 73$.

(ii) # 1st level blocks = # blocks for (Key, bptr) + # blocks per Key

⇒ $\lceil 1000/34 \rceil + \lceil 30/73 \rceil \times 10000$

⇒ 30 + 1000

⇒ 1030.

(iii) # I_2 entries = 1030 $bfr_{I_2} = \left\lfloor \frac{512}{9+6} \right\rfloor = 34$

I_2 blocks = $\lceil 30/34 \rceil = 1$

~~# I_3 entries = 1~~

~~# I_3 blocks = 1~~

∴ 2 level index.

(iv) # Total index blocks = 1030 + 1 = 1031

(v) # Accesses = 2 + 1 + 1

idx blocks, rcd ptr block, data block.

- Indexed Sequential Access Method: Just a terminology used for
  1. Multilevel Primary Index + Seq. file ordered by key field
  2. OR MLvl Clustering Idx + Seq. file ordered by nonkey field.

  By default it refers to 1.

ISAM = Indexed Seq. file

* The problem with ordered index is that modification (insertion/del.) is unoptimized & hard, if records are dynamic. So, we use Tree based indexes instead.

* Tree Based Index

- The problem w/ Static indexes that we've seen so far is when the data file changes, its difficult (more complex) to change the index structure.

- Tree based indexes are dynamic since they allow for easier modification.

- The two most popular choice for Tree based index are:
  - B-Tree
  - B+Tree

- The reason to use specific tree based data structure is that, DStructs like BST, AVL Tree, etc are in m/m DStructs, since they are suitable when the node has to only store small amounts of data, & accessing it is easy.
  DStructs like BTree & B+Tree are suitable because they can pack a lot of data in single nodes, and hence less no. of IO operations will result in higher data access.
  Also, this decreases the no. of levels. So, they grow horizontally more than they grow vertically.
- In the disk, one node would mean one block. B & B+ Trees utilize the block space as much as possible.

Note: [Convention]
- Root is at level 1.
- Height of a node = #edges from root to node.

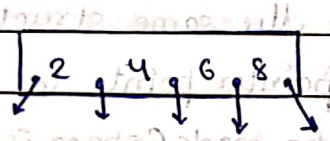
- B & B+ Tree are Search trees so, for any node its left subtree has value less than equal to node's value & vice-versa for right. (Although there can be more than 2 children.)

- For a disk related Dstruct, performance parameters : Spaced occupied; No. of IOs.
- B & B+ Tree are multiway Search Trees, i.e. one node stores more than one & more than two, Key & pointer respectively.

* B Tree

- There is a parameter associated w/ BTree & B+ Tree, Order.

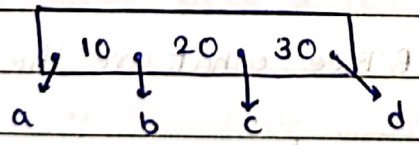
- Order : The ~~max~~ maximum number of child pointers (Block Ptr) a node of the tree can have.

Ex. 

∴ Order = 5

- Pointer can be :
  - Block (a.k.a. Node) Pointer ≡ Child Ptr ≡ Tree Ptr
  - OR Record pointer

- Some Multiway Search Tree rules :
  (1) Within a node, the values are ordered.
  (2)



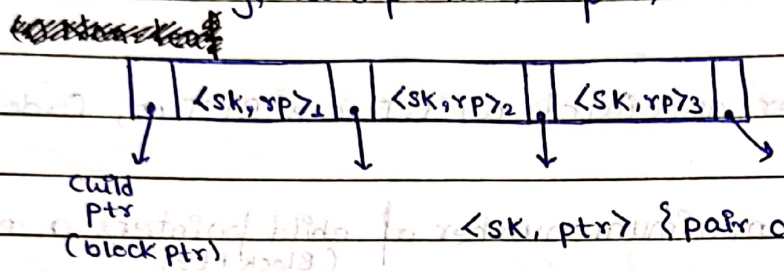
values in region a: < 10
values in region b: > 10 & < 20
c: > 20 & < 30
d: > 30

- All leaves must be on the same level.

- [Space Utilization Rule] In a BTree, for:
  (order : p)

- Root Node: 2 block ptrs to p block ptrs.
- Non Root Node: $\lceil p/2 \rceil$ block ptrs to p block ptrs.

- Node Structure: Every node (block) contains multiple <search Key, record pointer> pairs.



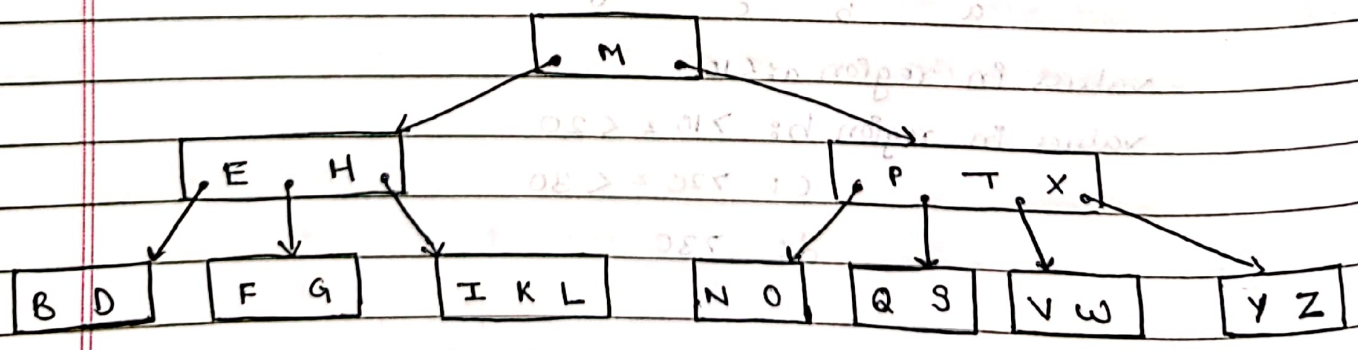
<sk, ptr> {pair can be record or block ptr}
child ptr is always block ptr.

Both leaf & non-leaf nodes have the same structure. In the leaf node, although the child pointer points to nothing, but the space for them is there in the node (space is wasted).

- So, for a B Tree w/ order P, no. of items {<Key, Ptr>}:
  - Root: $(2-1)$ to $(P-1)$ ⇒ 1 to $(P-1)$
  - Nonroot: $(\lceil P/2 \rceil - 1)$ to $(P-1)$

So, minimum 50% occupany for all nodes (except root).

Ex. For the following B Tree what are the possible orders:



(a) 2 (c) 4 (e) 6
(b) 3 (d) 5 (f) 7

Order can't be 2 or 3, since a node has four pointers. So, order p must be atleast 4.

For $p=4$: Root: 2 to 4 ptrs Nonroot: 2 to 4 ptrs.

So, $p=4$ is valid.

For $p=5$: Root: 2 to 5 Nonroot: $\lceil 5/2 \rceil = 3$ to 5 child ptrs.

So, 5 is valid.

For $p=6$: Root: 2 to 6 child ptrs Nonroot: $\lceil 6/2 \rceil = 3$ to 6 child ptrs.

So, 6 is valid.

For $p=7$: Root: 2 to 7 Nonroot: $\lceil 7/2 \rceil = 4$ to 7 child ptrs.

So, 7 can't be order, since a nonroot node is there w/ just three child ptrs.

We can also solve it analytically. Let's say p is the order.

The root has 2 child ptrs. $2 \leq \underline{\underline{2}} \leq p$ So, $p \leq 2$.

The min. no. of ptrs a non-root has $= 3$

w.r.t ——— " ——— $= \lceil p/2 \rceil$

So, $3 \geq \lceil p/2 \rceil$ {i.e. 3 must be atleast 50% of order}

$\Rightarrow 6 \geq p$

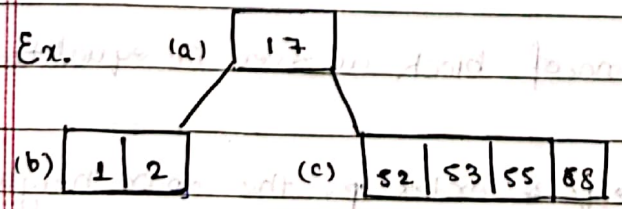
Also, max. no. of ptrs a nonroot has $= 4$.

w.r.t ——— " ——— $= p$.

So, $p \geq 4$. {i.e. 4 must be smaller than order by defn}

So, $\boxed{4 \leq p \leq 6}$

Ex.



for root: $2 \leq$ #child ptr $\leq p$.

So, $2 \leq p$.

for nonroot:

p must atleast be 5 {since (c) can have 5 child ptr or 4 items}

$\lceil p/2 \rceil$ should at most be 3 {since (b) can have 3 child ptr or 2 items}

$\therefore 5 \leq p \leq 6$.

Ex. Suppose block are 4096 B. Keys are 4B & pointers are 8B. Find order of B tree.

Order p must be such that:

$p \times len_{ptr} + (p-1) \times len_{\langle key, ptr \rangle} \leq 4096B$

$\Rightarrow 8p + (p-1)(4+8) \leq 4096$

$\Rightarrow 20p - 12 \leq 4096$

$\Rightarrow p \leq 205.4$

$\therefore p = 205$

- Searching in B Tree is similar to BST. For any node for a value b/w Key i & Key i+1 take pointer TreePtr i. For a node, to get record of a value equal to Key i, take dataptr i.

Note that all nodes contain key & corresponding pointer to record in data file: <Key, dataptr>

So, for a successful search, the search can end at any level w/ the matching key.
For unsuccessful search, the search will always go to the last level (leaves).

So, in the worst case, the no. of block accesses is equal to #levels or height of BTree.
With a BTree w/ n keys & order p, the max. height will happen when we store then min. possible keys in every node, i.e. $\lceil p/2 \rceil - 1$. So, appx height: $\log_{p/2} n$. So the

time complexity for search is $O(\log_{p/2} n * \underbrace{\log_2 p/2}_{\text{searching in a block}})$

$$\Rightarrow O\left(\frac{\log_2 n}{\cancel{\log_2 p/2}} * \cancel{\log_2 p/2}\right)$$

$$\Rightarrow O(\log_2 n)$$

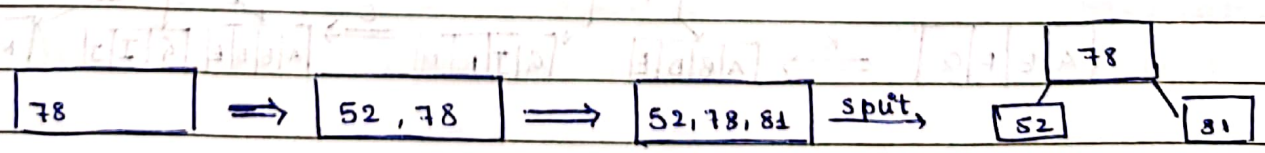
So, ultimately, the worst case search complexity of B Tree is as much as BST, but we get the benefit of storing more than a single key in a node.

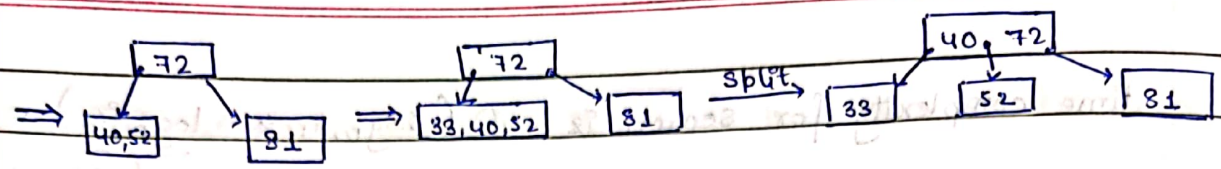
Insertion

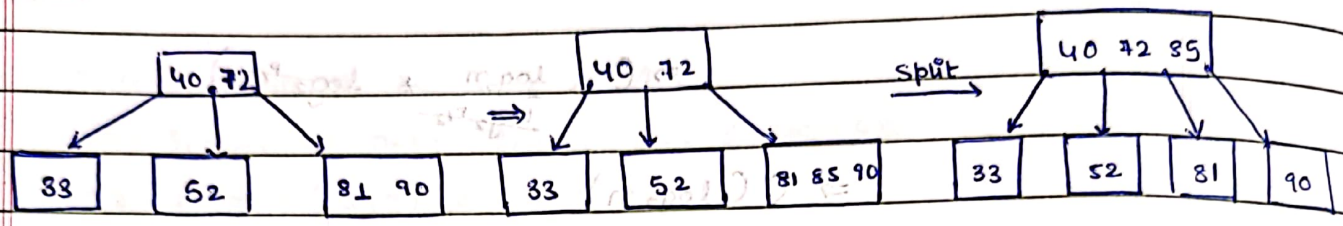
- Insertion in B Tree always happens from the leaf level.
- There can be multiple cases when inserting a key K at leaf of B Tree w/ order p $\{(\lceil p/2 \rceil - 1) \leq \#\text{Keys} \leq (p-1)\}$:
  - C1: Leaf isn't full. Just insert in sorted order.
  - C2: Leaf is full. Insert key in sorted way. Split the node in two halves & send centermost key to the parent, & repeat.

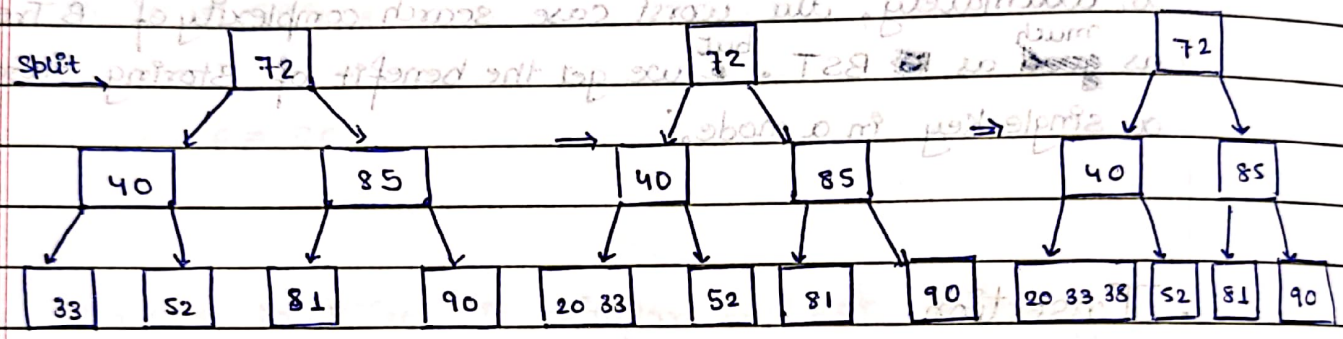
Ex. Inset in an initially empty B Tree: 78, 52, 81, 40, 33, 90, 85, 20 & 38.
Order = 3.

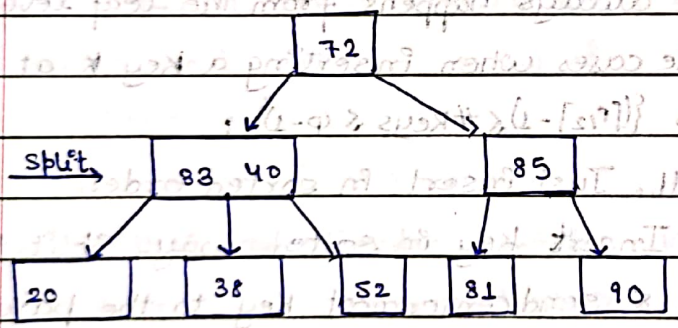
Root: 2 to 3 ptrs. 1 to 2 keys.
NonRoot: 2 to 3 ptrs. 1 to 2 keys.







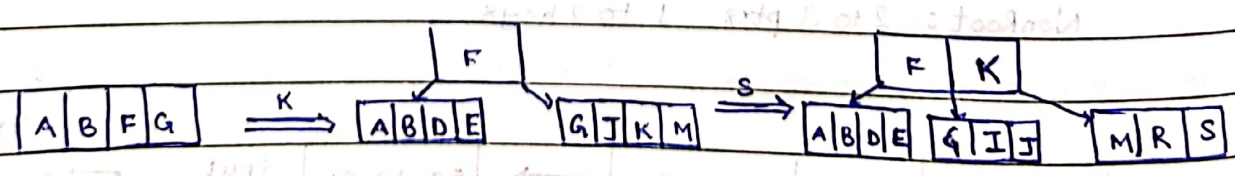




Ex. Last Example: A, G, F, B, K, D, M, J, E, S, I, R

Order: 5.

Root: 2 to 4 Keys. Nonroot: 2 to 4 Keys.

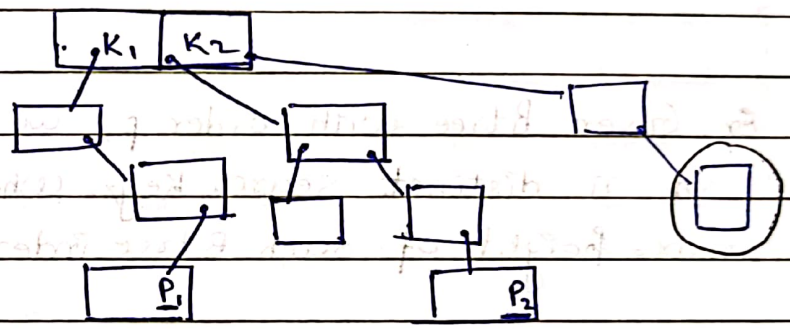


- The structure of B Tree depends on the order in which the Keys are inserted in the tree.

- Insertion in B tree can result in at max one node at every level splitting & atmost an increase of one level.

- #leaf nodes = #Keys in all non-leaf combined + 1.

Proof: For every non-leaf, we'll have a Key's immediate predecesor on a leaf node & a unique leaf node.



So, per non-leaf Key, one leaf node.
But for there is one extra leaf node. Rightmost, which has no predecessor.

Ex. Consider a DBMS w/ following characteristics:
- 2KB fixed size blocks.
- 12B ptrs.
- 56B Headers.

We want to build an index on 8B long search Key. Calc. the max. no. of records that can be indexed, with a 3 level Btree.

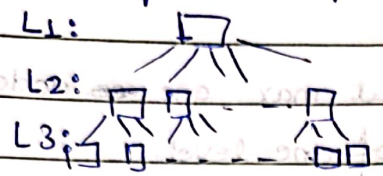
We first find max no. of Keys per block possible:

$(8+12)(p-1) + 12(p) + 56 \leq 2048$

$\Rightarrow p \leq 62.875 \Rightarrow p = 62$

$\therefore$ # Keys per block = 61

So, to get max no. of records, we'll have each of the three levels of Btree full.



L1: 1

L2: 62

L3: 62^2

∴ max. no. of ~~records~~ nodes = 3907

So, max. no. of keys in BTree = 3907 × 61

⇒ 238327

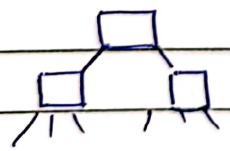
Ex. Given Btree with order p, we want to create index on n distinct search keys. What could be the max. height of such Btree index.

Order: p.

Root: 2 TreePtr to P TreePtr { 1 to (P-1) Keys

Nonroot: $\underbrace{\lceil p/2 \rceil}_{(\text{say } t)}$ to P TreePtrs { $\lceil p/2 \rceil - 1$ to (P-1) Keys. $(t-1)$

For max. height, we'd want to fill all nodes as little as possible.



height	#nodes	#Keys per node
0	1	1
1	2	t-1
2	2×t	t-1
3	$2 \times t^2$	t-1
⋮	⋮	⋮
h	$2 \times t^{h-1}$	t-1

⇒ #total nodes in h height = $1 + (2 \times t^0 + 2 \times t^1 + 2 \times t^2 + \dots + 2 \times t^{h-1}) \times (t-1)$

$\Rightarrow 1 + 2 \times \frac{(t^h - 1)}{\cancel{(t-1)}} \times \cancel{(t-1)}$

$\Rightarrow \quad 1 + 2(t^h - 1)$

So,

$$1 + 2(t^h - 1) = n.$$

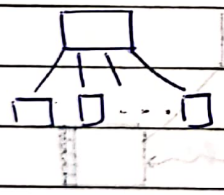
$$\Rightarrow \quad h = \log_t \left(\frac{n+1}{2}\right)$$

If h_{max} comes out to be some fraction, we'll take floor. So,

$$h_{max} = \left\lfloor \log_{\lceil p/2 \rceil} \left(\frac{n+1}{2}\right) \right\rfloor$$

{Floor, since to go one level more, the n nodes won't be enough, so we fill $(h_{max} - 1)$ levels as less as possible & last level w/ whatever remains}

- For min height, we'll fill all nodes as much as possible.



height	nodes	Keys per node
0	1	$p-1$
1	p	$p-1$
2	p^2	$p-1$
$\vdots$	$\vdots$	$\vdots$
h	p^h	$p-1$

So, max. no. of keys w/ h levels $= (1 + p + p^2 + \dots p^h) \times (p-1)$

$$\Rightarrow (p^{h+1} - 1)$$

So, $(p^{h+1} - 1) = n$

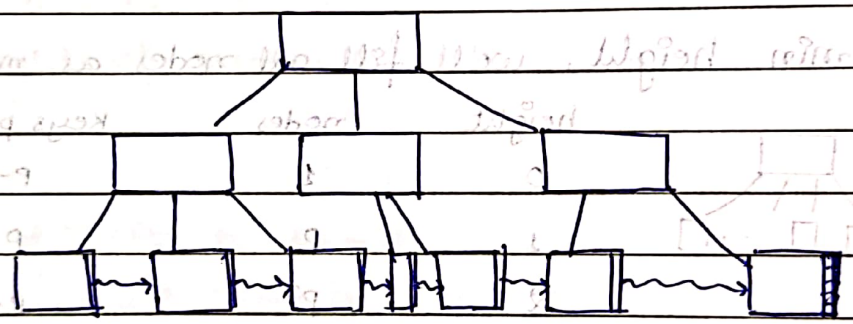
$$\Rightarrow \quad h = \lceil \log_p (n+1) - 1 \rceil$$

* B+ Tree

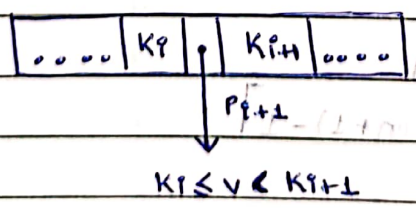
- In Btree some keys were on leaf, some on non-leaf. In B+Tree all keys will be on the leaf level.

- In B+ Tree, since all keys are on leaf level, we put <key, dataptr> together on leaf level. On any other level, we only keep keys (repition) since they are necessary for building the tree.

- In B+ Tree, every leaf node has exactly one TreePtr which points to the next leaf node.



- In a node, with keys K_i & K_{i+1}, to get values v. b/w $K_i \leq v < K_{i+1}$, take treeptr P_{i+1}.

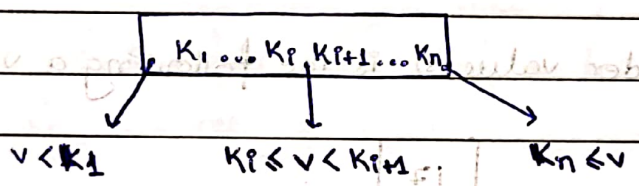


- The duplication of key values in non-leaf nodes is only for searching purpose, hence dataptrs are not necessary to keep on non-leaf nodes.

- Note that repetition only means that among all the Keys at leaf level {all distinct Keys}, some will also appear in non-leaf nodes & appear atmost once. So same Key can't repeat twice in non-leaf.

- In B+ Tree also, #leaf nodes = #nonleaf Keys + 1.

- Order of B+ Tree: Max. no. of total pointers per node.
  - For non-leaf, we'll only have treeptrs.
  - For leaf, we'll have multiple recordptrs & exactly one treeptr.

- B+ Tree rules:
  - Within a node items are in sorted order
  - 

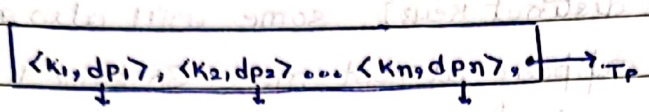
    {This is right-biased. We can also have left biasing.}

  - All leaves on same level.
  - For B+ Tree w/ order P
    - Root Node: 2 to P treeptrs | 0 dataptrs
    - Intermidiate: $\lceil P/2 \rceil$ to P treeptrs | 0 dataptrs
    - Leaf: 1 treeptr | $\lceil \frac{P-1}{2} \rceil$ to P-1 dataptrs.

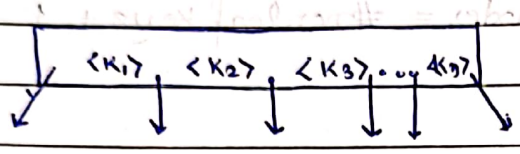
~~For B+ Tree w/ order P, no. of~~

- For B+ tree w/ K ~~[illegible]~~ total ptrs, no. of Keys = K-1

- Leaf Node structure : (Data ptr by default Record ptr).



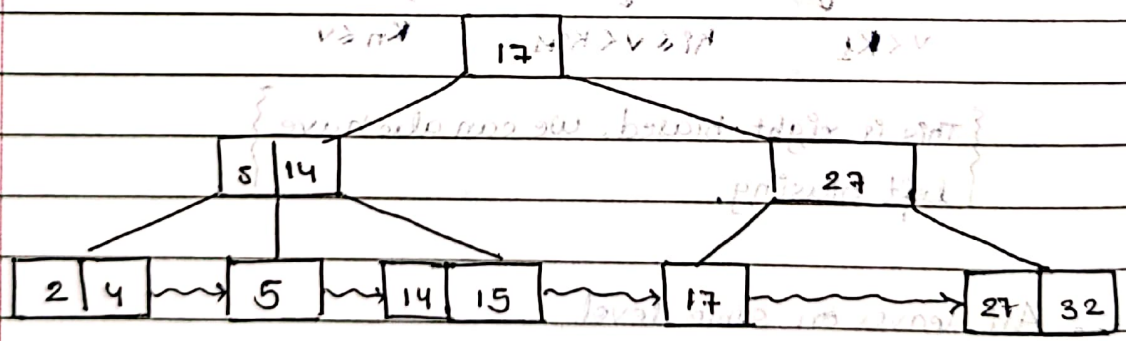
- Internal Node Structure :



-

	# TPtrs	# D Ptrs	# Keys
Root	2 to P	0	1 to P-1
Internal	$\lceil \frac{P}{2} \rceil$ to P	0	$\lceil \frac{P}{2} \rceil - 1$ to P-1
Leaf	1	$\lceil \frac{P-1}{2} \rceil$ to (P-1)	$\lceil \frac{P-1}{2} \rceil$ to P-1

Ex. For what order value(s) is the following a valid B+ Tree:



P must satisfy:

$1 \geq \left\lceil \frac{P-1}{2} \right\rceil$ $\quad$ $3 \leq P$

least no. of Leaf Keys

max. no. of inter. tree ptr

$\Rightarrow P \leq 3$ $\quad$ & $\quad$ $\Rightarrow P \geq 3$

$\therefore P = 3.$

Ex. Suppose block size is 4096 B. Keys are 4B, & let the pointers be 8B. Find the order of B+ Tree, where order is defn as the max. no. of pointers per block.

Say order = p.

Leaf node: (p-1) <Key, dptr> pairs & 1 tree ptr.

$$(p-1)(4+8) + 1 \times 8 \leq 4096.$$

$$12p - 12 + 8 \leq 4096$$

$$p \leq 341.66$$

Non leaf node: p tree ptrs & (p-1) <Key>.

$$(p-1) \times 4 + p \times 8 \leq 4096$$

$$12p - 4 \leq 4096$$

$$p \leq 341.66$$

So, since order for both leaf & internal node is same, the order of B+ Tree is 341.

- In B+ Tree, the order of non-leaf & leaf can differ if the size of tree & dataptr are different, since here we are calc. the max. possible order for each type of node.
But if we are given a fixed 'p', then the order is fixed, i.e., it doesn't matter if full space of block is utilized or not, only p ptrs can be there at max.

- If the B+ tree has only single node, i.e. only the root, then the root behaves as leaf.

Note: In case the order of internal & leaf nodes aren't same then we can't report the order of B+ Tree.

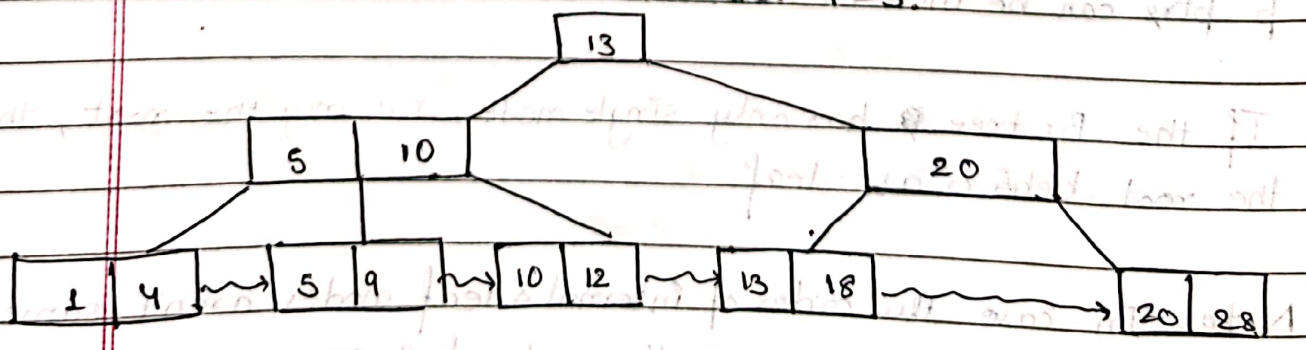
- **Searching**: In B+ Tree, the search must go on till the leaf for successful or unsuccessful search. So, to search for a particular Key's record, the no. of block access $= \Theta(h)$, where h is the height of index tree.
{For successful search, # block accesses $= h + 1_{data}$}

This also makes range search, i.e., searching for all Keys k b/w x & y, $x \leq K \leq y$. Search for $K = x$, get to the leaf. On the leaf then traverse linearly till you find $K > y$. So search cost : h {to get to first leaf node} + $(b-1)$ where b is the # blocks w/ at least one Key K, such that $x \leq K \leq y$.

$\therefore$ IO cost $= h + b - 1$

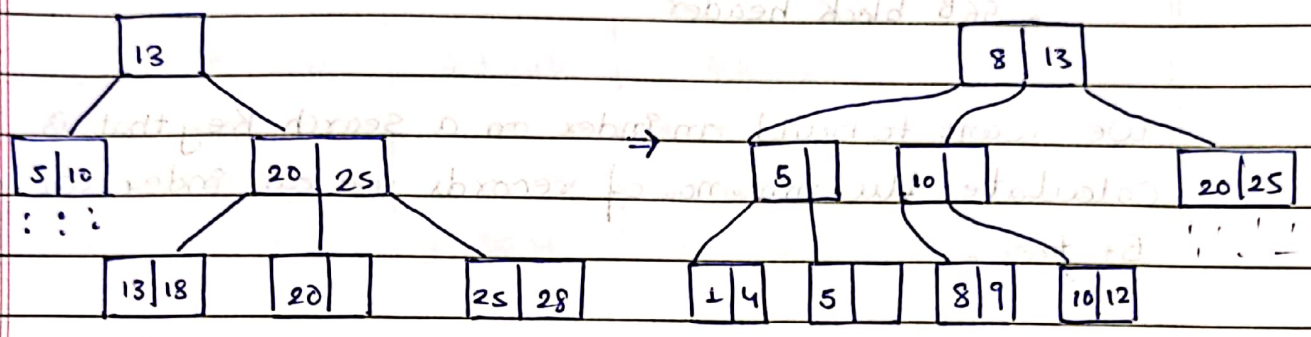
- **Insertion**: Insertion happens at leaf level only.
  - The leaf node if full while inserting, split in two, send the middlemost key to the parent node, while also keeping the copy on the leaf.
    [Follow either left or right biasing, but be consistent.]
  - If internal node is full, split in two, send middlemost key to parent, don't keep a copy.

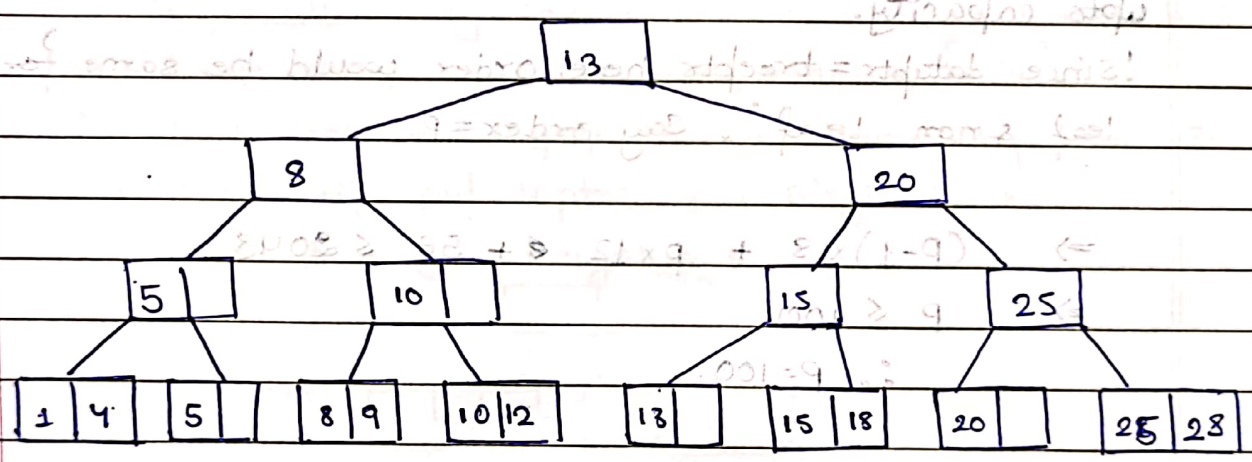
Ex. Consider a B+ Tree w/ order $p = 3$.



Insert: 25, 8, 15.

	# Tree P	# Data P	# Keys
Root	2 to 3	0	1 to 2
Int.	2 to 3	0	1 to 2.
Leaf	1	1 to 2	1 to 2





- For a single insertion, at most h splits might happen and the height at most increases by 1.

- For B+ Tree also, order of insertion determines the structure of the tree.

Ex. Consider a DBMS w/ following characteristics:

- 2 KB fixed size blocks
- 12 B ptrs
- 56 B block header

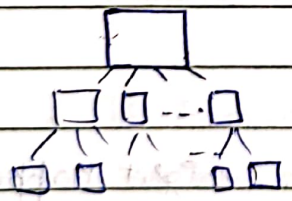
We want to build an index on a search key that is 8 B long. Calculate the max. no. of records we can index w/ a 3 level B+ tree.

To index as many as possible records, we'll fill everything upto capacity.
{Since dataptr = treeptr here, order would be same for both leaf & non leaf}. Say order = p.

$\Rightarrow \quad (p-1) \times 8 + p \times 12 + 56 \leq 2048$

$\Rightarrow \quad p \leq 100$

$\therefore \quad p = 100.$



level	#nodes	#keys/n	#dptrs/n	#tptrs/n
0	1	99	0	100
1	1×100	99	0	100
2	1×100×100	99	99	1

$\therefore$ # indexed records $= 99 \times 10^4$.

- We can also find min.

0	1	1	0	2
1	1×2	49	0	50
2	1×2×50	49	49	1

$\therefore$ # index records $= 1 \times 2 \times 50 \times 49 = 4900$

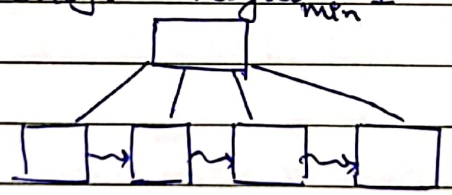
Ex. [Bulkloading] Create B+ tree of order 5 w/ given Keys of
(a) min height (b) max height.
92, 24, 6, 7, 11, 8, 22, 4, 5, 16, 19, 20, 78

	# tree p	# data p	# Keys
root	2 to 5	0	2 to 4
internal	3 to 5	0	2 to 4.
leaf	1	2 to 4	2 to 4.

Bulkloading goes bottom up.

(a) 1. At leaf, all 13 Keys should be there. We want min. height, so put max. no. of Keys. ∴ # leaf nodes = $\lceil 13/4 \rceil = 4$.

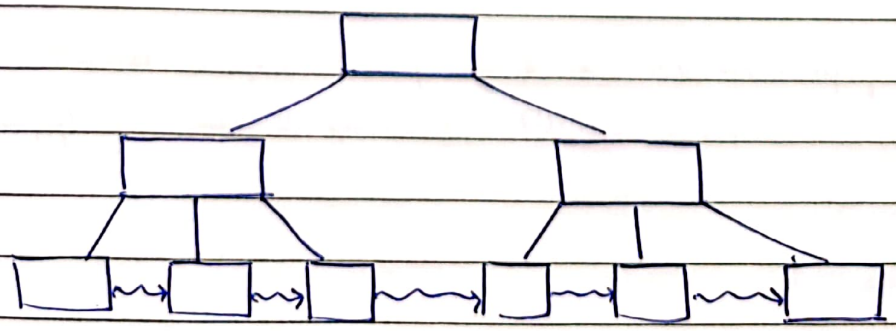
2. So, there shold be 4 ptrs in one above leaf level. So, we can easily put 4 ptrs in a single block.
∴ 2 levels only. $\text{Height}_{min} = 1$



(b) 1. At leaf, fill min. for max height. $\lfloor \frac{13}{2} \rfloor = 6$.

2. So, we'll require, 6 ptrs in one above leaf level. Again put min. pointers per node, i.e., 3. $\lfloor \frac{6}{3} \rfloor = 2$

3. At two above leaf level, $\lfloor \frac{2}{2} \rfloor = 1$.
∴ Atmost 3 levels. $\text{Height}_{max} = 2$



classmate
Date
Page

Transaction Management & Recovery

- Whenever we access data in the database, the most fundamental operations involved are "read" & "write". When data is read from disk to m/m, the entire block containing the desired item is copied to buffer. If no write op^n is performed, the in-memory copy doesn't need to be copied back to the disk.
  If the data has to be both read & written to, it is copied (block) to the main m/m, changes are made to the in-memory copy, then block is copied back to disk.

Ex. Suppose you want to double the salary of employee E_1.
- Copy the block w/ E_1's record to buffer.
- From the buffer read the salary attr. to some program variable, say 't'.
- Modify t as, t = 2*t. Store the value of t in buffer.
- Copy the block back to disk.

Ex. Suppose you want to set validity to 365 days for E_1.
- Copy block w/ E_1's rec. to buffer.
- Assign to some ~~value~~ temp. var. in program, the val. 365.
- Copy the value of t to validity of E_1 in buffer.
- Copy the block back to disk.

The second sequence of op^{n}s is not reading from buffer before modifying. This is called, Blind write.

- Transactions in database are sequence of read, write & data manipulation operations on database items, at the very lowest level.

Ex. Transaction to transfer $50 from account A to B.

```
read (A)
A.bal = A.bal - 50
write (A)
read (B)
B.bal = B.bal + 50,
write (B)
```

⇐

```
UPDATE accounts
SET balance = balance - 50
WHERE acct_id = 213;
```

- Problems can happen when a single transaction execution, as well as concurrent transaction execution. Transaction Management is the task of ensuring that these problems are handled.

In DBMS s/w, Transaction Manager & Recovery manager are the components working in coordination to achieve this task.

Ex. Multiple Users (concurrently) try to book airplane/ railway seat.

Ex. Balance debited from source account but not credited to destⁿ account.

* - Database Elements: Everything related to db, such as, relation, record, block, some attribute, particular value of a record's attribute is element.

Each element (item) has unique name.

- State / Snapshot of Database: From transaction's point of view, database is a collection of elements. State of DB at any point of time is the value of all the elements of the database.

- There can be various constraints that can be applied on a database, such as Domain, Key, Entity Integrity, NULL, Referential constraints & more. These are explicit.
Along w/ these constraints, there can be application dependent constrains.
Ex. Net balance of accounts involved in monetary transactions, must be in check, as; "Source + dest is constant".
These are implicit constraints & harder to enforce.

- A database state which ~~has~~ fulfills all the constraints (explict + implicit), then it will be called to be Consistent. Otherwise the state will be Inconsistent.

Ex. Same seat shouldn't be booked by more than one people. (Application Dependent requirement)

* [Simplified]

- Assumption: Every transaction has its own buffer.
- Read opⁿ: t = Read(A) or Read(A, t) or Read(A)
  - Block containing DB element 'A' is copied from disk to m/m & assigned to temp var. 't' or temp var 'A'.
- Write opⁿ: Write(A, t) or Write(A)
  - Block containing DB element 'A' is copied from disk to m/m.
  - In-m/m copy of A is modified.
  - Block is copied back from m/m to disk.

Note: Showing wrong results to the user is also violation of "DB Consistency".

- Assumption: write op^n stores the written block back to the disk immediately.

Ex.

T_1	T_2
Read(A); A ← A+100;	
Write(A);	
Read(B); B ← B+100;	
Write(B);	
	Read(A); A ← A×2; Write(A);
	Read(B); B ← B×2; Write(B);

Consistency Requirement: A = B {This is implicit requirement}
If the state is consistent before transactions T_1 & T_2 execute, will the state remain consistent after both execute?

Yes. The execution of the transactions is serial. So, at the end of execution state remains consistent.

- Blind Write: A "write" op^n on DB element "X" which isn't preceeded by "read" op^n of "X", i.e., "X" is never bought into m/m buffer of transaction before the transaction brings in block of X & assigns (writes) to it, and stores it back.

Ex. Read(x); Write(x);

Copy X's block (Disk to m/m) } Read(x)
t ← X

Copy X's block (Disk to m/m)
X ← t } Write(x)
Copy X's block (m/m to disk)

This is not blind write. X bought in buffer of transaction before assigning to it.

Ex. Write(X); Read(X);
This is blind write.

Copy from ~~main~~ disk to m/m X's block.
X ← someval / var.
Copy back X's block } Write(X)
Copy from disk to m/m X's block.
t ← X } Read(X)

Ex. Same Consistency Requirement as before

T_1	T_2
Read(A); A←A+10;	
Write(A);	
	Read(A); A←A×10;
	Write(A);
Read(B); B←B+10;	
Write(B);	
	Read(B); B←B×10
	Write(B)

A

T_1	T_2
Read(A); A←A+10;	
Write(A);	
	Read(A); A←A×10;
	Write(A);
	Read(B); B←B×10;
	Write(B);
Read(B); B←B+10;	
Write(B)	

B

These are interleaved execution sequence. At the end of A, the DB state remains consistent. B can make state inconsistent (for atleast one value)

* [Actual]
- Fetch block only when block not in main m/m.
- There is a common buffer for all trans., but each have their own vars.
- Write opⁿ may/maynot lead to immediate copy back of block. (It is generally decided by Recovery Manager).

Note: The simplified def^{n}s of Reads Write op^{n}s are good enough for all the analysis of transactions. ~~Bufpse~~
The actual def^n & simplified def^n are equivalent.
The actual def^n would be useful for study of Recovery Manager.

* Transaction

- Transaction is a collection of op^{n}s, which forms a single logical unit of work. Transaction is the unit of execution of DB operation.

So, every query in SQL, any op^{n}s such as checks, constraints, etc. are all internally transactions.

- For every transaction, the last op^n is either commit or abort.
  - Commit: Tells the transaction manager & recovery manager that the transaction is over. All op^{n}s have been performed.
    Commit op^n doesn't guarentee ~~that~~ that all the changes done by transaction are reflected (stored) in the actual database. The changes may reflect after some time or at time of commit. {Depends on Trans. Manager & Recovery Manager}

  - Abort or Rollback: Tells the TM & RM, that some problem has occured during the execution of the transaction & the changes made by transaction ~~may~~ should be undone.

(Think of Abort as Try-Catch block, which can only decide if catch will be executed during runtime.)

Note: By default, if neither given, assume last action to be "Commit".

- Concurrent Execution: Multiple transactions execute by interleaving their op^{n}s, on CPU. [We assume single CPU]

- Concurrency Control: Ensuring that concurrent execution of transactions doesn't lead to inconsistent state of the database.

- Remember, it may happen that $T_1 \rightarrow T_2$ & $T_2 \rightarrow T_1$ executions may give different results, but that isn't important. It's important that the result preseves consistency.

* ACID Properties

- These are desirable properties of transaction execution.

- Atomic Property: A transaction must be the atomic unit of execution in the DB. So, any particular transaction should either execute as whole or not execute at all. Hence, if a transaction begins execution, but for some reason it fails, in middle, then any change reflected in DB must be undone.

- Durability Property: At the time, when "Commit" op^n executes for a transaction, the Atomicity property is satisfied since all of the transaction is executed, but we know "Commit" doesn't guarentee that the changes reflect in DB immediately.
  Durability property says that after a transaction commits, the changes may/may not reflect immediately in DB, but the changes must not be lost, (even if any failure of DB, disk, OS, etc happens).

- Consistency Property : We know that the database can have both implict & explicit consistency requirement. Explicit constraints can easily be enforced by the DBMS s/w, but the implicit constraints are application dependent.

Consistency property says that, when assuming that a transaction is executing alone, starting w/a consistent state, then after the transaction executes, the state of the the system should still remain consistent, i.e. no constraint should be violated (implicit or explicit).

The task of ensuring Application dependent constraints don't get violated falls upon the programmer, within a single transaction. DB can handle explicit constraints.

So, if the single transaction's code itself is wrong (i.e. makes DB inconsistent) then TM & RM can't help even if each transaction executes one after another.

If the transaction code is correct, then the TM & RM take care of :

(1) Failure within transaction. (Atomic Prop.)

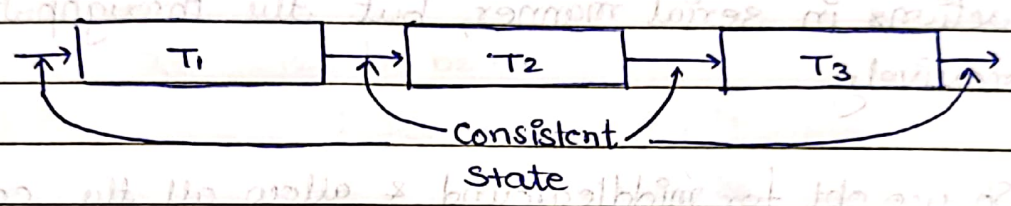
(2) Failure after commit. (Durability)

(3) Failure due to concurrent execution. (Isolation)

Note: We assume that the transaction author is smart enough to write transactions which don't violate ~~transaction~~ consistency requirements. So we assume that Consistency Property (of individual transactions) hold.

Ex. Does Serial execution of Transaction always holds DB consistency?

Assuming that each transaction holds consistency property, we can say that serial execution of transactions always hold DB consistency.



- Isolation Property: (Assuming single CPU) Multiple transactions can be executing on the CPU concurrently (interleaved), (i.e one opn at a time on CPU but of different transactions).

When transactions execute serially, every one of them is executing in isolation, i.e., independently, without affect of other transactions.

When transactions execute ~~in~~ concurrently, situations can arise where execution of one transaction affects the ~~transaction~~ execution of other transaction & may lead to inconsistent state.

Ex. Consider the following transactions

T_1	T_2	T_1's Buffer	T_2's Buffer	
R(A) ←		t = A		
A ← A-10 ←		t = t-10		
	R(A) ←		t = A	①
W(A) ←		A = t		
	R(A) ←		t = A	②

At point one & two T_2 will read different values of A & will know that there is some other transaction affecting its execution.

This violates isolation property.

Isolation property says that no matter serial or concurrent execution, every transaction should execute as if it is execution in isolation & execution of other transactions don't affect it.

One extreme way to ensure isolation is to execute transactions in serial manner, but the throughput suffers massively.

So, we opt for middleground & allow all the concurrent transaction executions whose effect is equivalent to some serial execution. This is called Serializability.

- Recovery Manager is responsible for holding Atomicity & Durability property.

Concurrency Manager is responsible for holding Isolation property. CManager is subpart of Transaction Manager.

For writing transactions which hold Consistency property the programmer is responsible.

- If a transaction's code individually fails to preserve some constraint (implicit or explicit) then DB consistency is violated.

But if DB consistency is violated then it doesn't necessarily mean that it's because some transaction code didn't preserve all constraints.

* Serializability

- Schedule: When transactions are executing concurrently in interleaved fashion then the order of execution of various operations of the transactions is known as the schedule.

Ex. $T_1: a_1\ a_2$ $\quad T_2: b_1\ b_2$

T_1	T_2
a_1	
a_2	
	b_1
	b_2

T_1	T_2
a_1	
	b_1
a_2	
	b_2

T_1	T_2
a_1	
	b_1
	b_2
a_2	

T_1	T_2
	b_1
a_1	
	b_2
a_2	

All are schedules. #possible schedules $= \frac{4!}{2!\,2!} = 6.$

- The order of op^{n}s for multiple transactions can interleave, but the order of op^{n}s within a transaction can't change.

Suppose you have T_1 with n_1 op^{n}s, T_2 w/ n_2 ops ... T_m w/ n_m op^{n}s then:

$$\#\text{schedules} = \frac{(n_1+n_2+\dots n_m)!}{n_1!\,n_2!\dots n_m!}$$

$$\#\text{serial schedules} = m!$$

So,

$$\#\text{non-serial schedules} = \#\text{schedules} - \#\text{serial schedules}$$

Note: Every serial schedule is also concurrent.

- Since we assume every transaction preserves consistency, when executed in full successfully, we can say that every serial schedule results in consistent DB.

Concurrent schedule may/maynot lead to inconsistent DB.

Ex.

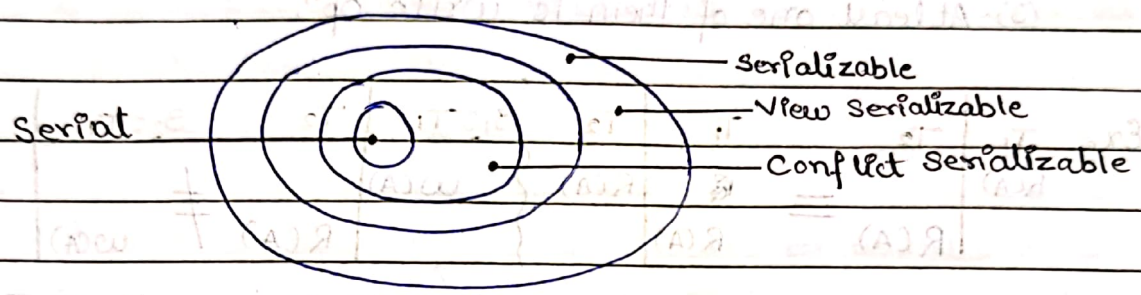
T_1	T_2	
	R(A)	①
A←A-10		②
	A←A+10	③

When T_2 is executing ①, it leads to I/O op^n (reading block from disk). For that duration T_1 can be scheduled & allowed to execute ②. This leads to both better CPU utilization & disk utilization.

- Serializability: We know that serial schedules preserve DB consistency. The idea of allowing only those concurrent schedules which have effect similar to some serial schedule is called Serializability.

- Serializable Schedule: A schedule S is serializable iff for every initial DB state (consistent), effect of S is same as some serial schedule.
  Every serial schedule is serializable.
  ($\exists_{\text{serial schedule } S'}$ $\forall_{\text{consistent DB state}}$ effect of S = effect of S')
- So, a schedule S is serializable, iff there exists a serial schedule S', such that for all possible consistent initial DB state, the result of S & S' (the final DB state) must be same.

- The def^n of Serializability is almost impossible to implement in practical systems, so we use restricted def^{n}s of serializability : Conflict Serializable & View Serializable.
A schedule S has the following hierarchy:



The (set of serial schedules) is proper subset of (set of CS schedules).
Similarly (set of CS schedules) is proper subset of (set of VS schedules) & (set of VS schedules) is proper subset of (set of all serializable schedules).

The def^n of Conflict serializability is easy to implement. The def^n of View serializability has NP-Complete exponential time implementation.

* Conflict Serializability

- The idea of Conflict Serializability is based on only read write op^{n}s of DB elements, not on their actual values or the computation on them.

- For CSerializability, we only look at read & write of data elements, and we assume that b/w a read (Q) & write (Q) on data item Q, a transaction may perform an arbitrary sequence of op^{n}s on the copy of Q in the local buffer.

- **Conflict Pairs:** Two operations I & J (both Read/Write op^n) in a schedule have conflict if they are operations:
  (1) By different transactions.
  (2) On the same data item.
  (3) Atleast one of them is write op^n.

Ex.

T_1	T_2	T_1	T_2	S_1: T_1	T_2	S_2: T_1	T_2
R(A)			R(A)	W(A)			R(A)
	R(A)	R(A)			R(A)	W(A)	

$=$ $\neq$

Here T_2 reads value written by T_1.

Here T_2 reads some value of A which may/may not have been same as value of A in the other ~~trans~~ schedule S_1.

Suppose A initially is 3, & W(A) in T_1 makes it 5. So in S_1, T_2 reads 5, in S_2, T_2 reads 3.

Ex. If we swap two consecutive non-conflicting op^n of two transactions in a schedule, does it change the effect of the schedule on the database?
No. If they are non-conflicting op^n, swapping them doesn't change the effect of schedule on database.

- **Conflict Equivalent Schedules:** Two schedules S & S' are conflict-equivalent iff by repeatedly swapping non-conflicting consecutive op^{n}s of different transactions in S, we can obtain S'.

$$S \equiv_C S'$$

Ex. Can we swap non-consecutive non-conflicting opⁿs two transactions without changing the effect of S on DB?

No. Suppose the opⁿs are a_i & b_i of trans. T_1 & T_2 & they are non consecutive, i.e., there's at least one opⁿ X b/w them.

$$a_i, \ldots X \ldots, b_i$$

swap

if we swap them, then there would be problem, due to X. If X is opⁿ of T_1; a_i & X shouldn't be swapped. If X is opⁿ of T_2, X & b_i shouldn't change order.

Ex.	T_1	T_2	T_1	T_2	T_1	T_2
1	R(A)		R(A)		R(A)	
2	W(A)		W(A)		W(A)	
3		R(A)		R(A)	R(B)	
4		W(A)	R(B)		W(B)	
5	R(B)			W(A)		R(A)
6	W(B)		W(B)			W(A)
7		R(B)		R(B)		R(B)
8		W(B)		W(B)		W(B)
	S_1		S_2		S_3	

Which are Conflict Equivalent?

In S_2, if we swap 4 & 5 we get S_1 {4 & 5 don't have conflict}.

∴ S_1 & S_2 are conflict equivalent.

In S_1, w/ following swaps: 4 & 5, 3 & 4, 6 & 5, 4 & 5, {none have any conflict} we get S_3.

∴ S_1, S_2 & S_3 are all conflict equivalent.

Also, S_3 is serial. So, S_1 & S_2 are conflict equivalent to a serial schedule.

- Conflict Serializable: A schedule S is conflict serializable iff there exists a serial schedule S' with which S is conflict serializable.

Note: If for a transaction it isn't specified that last last opⁿ is commit or abort, we assume it to be commit.
- For Serializability, Commit opⁿ is irrelevant.
- If trans. has abort at last, we can ignore the transaction from schedule.

Ex.	T_1	T_2	T_3
1	R(A)		
2		W(A)	
3	R(B)		
4			R(C)
5		R(C)	
6	W(B)		
7	W(A)		
8			W(C)

Not conflict serializable. 2 can't be swapped by 1, nor can it be swapped w/ 7, so T_2's at least one instⁿ must be b/w T_1's opⁿs. ∴ T_1 & T_2 can not be made serial.

Ex. Which is/are conflict serializable:

(A) T_1: R(X) T_2: W(X) T_1: W(X) T_2: Abort T_1: Commit

(B) T_1: R(X) T_2: W(X) T_1: W(X) T_2: Commit T_1: Commit
1 2 3 4 5

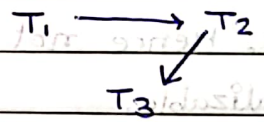
A has abort for T_2. Remove T_2. Only one transaction in schedule. ∴ Serializable. {Conflict Serializable}

B is not CS. 1 & 2, 2 & 3 can't be swapped.

- Precedence Graph Test: Create a Directed graph, w/ all transactions as node. Add an edge from T_i to T_j iff there is a conflict b/w T_i & T_j, say op^{n}s X & Y resp, & X precedes Y.

Iff the precedence graph has a cycle, then the schedule is not Conflict Serializable.

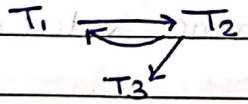
Ex. S: $R_2(X)$, $R_1(Y)$, $W_2(X)$, $R_3(X)$, $W_1(Y)$, $W_3(X)$, $R_2(Y)$, $W_2(Y)$



Since no cycle, hence conflict serializable.

Order of serial execution: $T_1 \rightarrow T_2 \rightarrow T_3$ (Topological Sort)

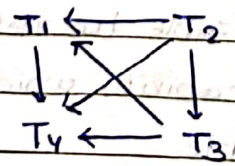
Ex. S: $R_2(A)$, $R_1(B)$, $W_2(A)$, $R_2(B)$, $R_3(A)$, $W_1(B)$, $W_3(A)$, $W_2(B)$



Since cycle, hence not conflict serializable.

- If the Precedence graph has no cycles, it is DAG, & the topological sort of the graph gives the conflict equivalent serial schedule.

Ex. S: $R_2(X)$, $W_3(X)$, $Commit_3$, $W_1(X)$, $Commit_1$, $W_2(Y)$, $R_2(Z)$, $Commit_2$, $R_4(X)$, $R_4(Y)$, $Commit_4$



No cycle. Order: $T_2 \rightarrow T_3 \rightarrow T_1 \rightarrow T_4$.
Conflict Serializable.

{Ignoring all Commits.}

- Time Complexity for checking Conflict Serializability:
  - Creating precedence graph : $O(n^2)$
  - Checking for cycle : $O(V+E)$

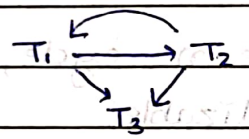
So, in worst case : $O(V^2)$ where V is no. of transaction.

Ex. S_1: $w_1(Y)$; $w_1(X)$; $w_2(Y)$; $w_2(X)$; $w_3(X)$;

S_2: $w_1(Y)$; $w_2(Y)$; $w_2(X)$; $w_1(X)$; $w_3(X)$;

S_1 is serial & hence Serializable.

S_2:



Cycle. Hence not Conflict Serializable.

But if we check the op^{n}s the value of Y written by w_2 finally persists & its blind write, i.e the value of Y is not used.

All writes in S_2 are blind & hence value of X & Y are never used.

Similarly the value of X set by T_3 persists.

So, in effect S_2 is similar to:

S_2': $w_1(Y)$; $w_1(X)$; $w_2(Y)$; $w_2(X)$; $w_3(X)$;

as this produces the same state as S_2 for all initial states.

$\therefore$ S_2 is serializable.

- So there are schedules which are serializable but not conflict serializable. We want a def^n of serializability which covers more schedules & still is implementable.

* View Serializability

- View serializability also, takes only read & write op^{n}s into account.

- View means "Read".

Ex. Suppose you have two transactions : T_1 & T_2

S_1:

T_1	T_2
R(A)	
	W(A)
	R(B)

S_2:

T_1	T_2
	W(A)
R(A)	
	R(B)

In S_1, R(A) gets the value of A from initial DB state (or we can assume the initial state to be output of some transaction Tinit).
So, source of A for $R_1(A)$ in S_1 is Tinit.
Similarly for Tfinal, source of A will be T_2 in S_1.
Source of $R_2(B)$ is Tinit.
But in S_2, source of $R_1(A)$ is T_2. Source of $R_2(B)$ is Tinit.
Hence we can say, S_1 & S_2 are not view equivalent, since $R_1(A)$ has different views in them.

Ex. S_1:

T_1	T_2
W(A)	
	W(A)

S_2:

T_1	T_2
	W(A)
W(A)	

Not view equivalent. For Tfinal source (view) of A in S_1 is T_2, in S_2 is T_1.

Ex. S_1:

	T_1	T_2
1	W(A)	
2		R(A)
3	W(A)	

S_2:

T_1	T_2
W(A)	
W(A)	
	R(A)

In S_1, T_2: R(A) reads (views) from first write T_1: W(A)[1]. For future transactions the source is T_1: W(A)[3].

In S_2; T_2: R(A) reads (views) from second write T_1: W(A)[3]. For future trans. the source is T_1: W(A)[3].

∴ Not view equivalent, since source for T_2: R(A) is different.

Ex. S_1:

T_1	T_2	T_3
R(A)		
	W(A)	
W(A)		
		W(A)

S_2:

T_1	T_2	T_3
R(A)		
W(A)		
	W(A)	
		W(A)

R_1(A) has same view in both S_1 & S_2, T_{init}.
For T_{final} view of A is T_3: W(A)[4] in both S_1 & S_2.
∴ $S_1 \equiv_V S_2$.

- **View Equivalence:** Two schedules S_1 & S_2 on same set of transactions are view equivalent iff:
  (1) Same read opⁿs in S_1, S_2, must have the same view.
  (2) To future transactions, both S_1 & S_2 provide the same view, i.e., final write for an element X is by same transaction in both S_1 & S_2.

Read R(X)[i] in S_1 & S_2 have the same view iff they both read values written by same transaction's same write opⁿ.

Ex. S_1:

	T_1	T_2
1	W(A)	
2		R(A)
3	W(A)	

S_2:

T_1	T_2
W(A)	
W(A)	
	R(A)

View source for $R_2(A)$ in S_1 is first write of T_1.
View source for $R_2(A)$ in S_2 is second write of T_1.
Hence although the transactions in both S_1 & S_2 are same, the write opⁿs are different. Hence,

$$S_1 \not\equiv_v S_2$$

- **View Serializable:** A schedule S is view serializable iff it is view equivalent to some serial schedule S' on same set of transactions.

Ex. Check if view serializable:

T_1	T_2	T_3
R(Q)		
	W(Q)	
W(Q)		
		W(Q)

If yes, then give the View Equivalent Serial Schedule (VESS)

Element	Read by [Source]	Written By	Final Write
Q	T_1 [Tinit]	Tinit, T_1, T_2, T_3	T_3

Final write of Q must be same. So, T_3 must be in last, else T_2 or T_1 would have final write.
T_1 must read Q from Tinit.
So, order should be T_1, T_2, T_3.
Also, since the write opⁿ of T_2 is blind, it can be in b/w T_1 & T_3.

$\therefore$ VESS: $T_1 \rightarrow T_2 \rightarrow T_3$.

Ex.

T_1 :		R(A)		W(B)		
T_2 : R(B)	W(A)				W(B)	
T_3 :			R(A)			W(B)

Elem	Read By [source]	Written By	Final Write
A	T_1[T_2], T_3[T_2]	Tinit, T_2	T_2
B	T_2[Tinit]	Tinit, T_1, T_2, T_3	T_3.

- Since T_2 reads B from initial state, it must be first, else T_1 or T_3 will become source of B for T_2.
- T_1 & T_3 must be after T_2, since T_2 is source of A for T_1 & T_3.
- T_3 must be at last, since it has last write of B, else T_1 or T_2 will be come final write.
- T_2 being at first doesn't affect final write of A.

$\therefore$ VESS: $T_2 \rightarrow T_1 \rightarrow T_3$.

We can also write the four cond^{n}s as eq^{n}s:

[R(B)] ① ... $T_2 \rightarrow$ ... (T_1, T_3) ... ③ ... (T_1, T_2) ... $\rightarrow T_3$ [W(B)]

[R(A)] ② ... $T_2 \rightarrow$ ... (T_1, T_3) ... ④ ... $T_{init} \rightarrow$ ... T_2 [W(A)]

{ Reads } { Final writes }

Ex.

T_1		R(A)	W(C)		W(B)							
T_2	R(A)								W(D)	R(B)		
T_3				R(C)			W(A)					
T_4						R(B)		R(C)			W(A)	W(B)

Elem	Readby [source]	written By	Final write
A	T_2[init], T_1[init]	T_3, T_4	T_4
B	$T_4[T_1]$, $T_2[T_1]$	T_1, T_4	T_4
C	$T_3[T_1]$, $T_4[T_1]$	T_1	T_1
D	–	T_2	T_2

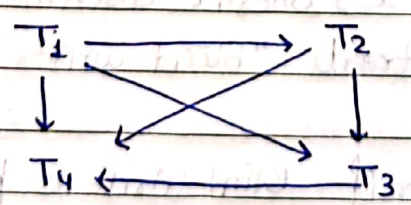
Eqn^s: ① T_3 ... T_4 [A] ③ (T_2, T_1) ... (T_3, T_4) [A]

② T_1 ... T_4 [B] ④ (T_1) ... (T_2, T_4) [B]

{Final write} ⑤ (T_1) ... (T_3, T_4) [C]

{ Reads }

We can draw precedence graph from this.



Since this is acyclic, we can get Topological order:

$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$.

{ Note that no transaction is overwritting some element X, so we don't get situation like: $W_1(X)$, $R_2(X)$, $W_1(X)$ }

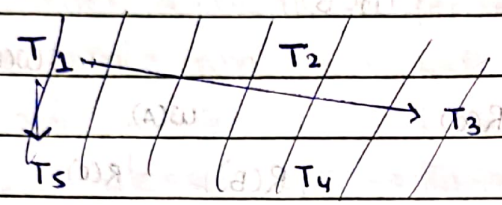
Ex. $R_1(A)$; $R_3(D)$; $W_1(B)$; $R_2(B)$; $W_3(B)$; $R_4(B)$; $W_2(C)$; $R_5(C)$; $W_4(E)$; $R_5(E)$; $W_5(B)$;

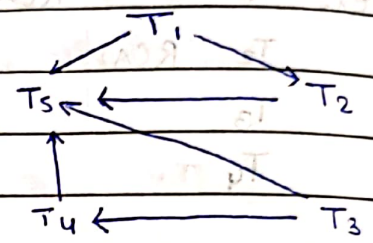
(No overwrite by any Transaction) [Multiple write]

Eq^{n}s: ① (1, 3) ... 5 ④ 2 ... 5

② 1 ... 2 (No 3, 5) ⑤ 4 ... 5

③ 3 ... 4 (No 1, 5)





So, possible topological orders:

① T_1 T_2 T_3 T_4 T_5

② T_1 T_3 T_2 T_4 T_5

③ T_1 T_3 T_4 T_2 T_5

④ T_3 T_4 T_1 T_2 T_5

⑤ T_3 T_1 T_4 T_2 T_5

⑥ T_3 T_1 T_2 T_4 T_5

Out of these the orders that satisfy the above conds: 1 ... 2 (No 3, 5) & 3 ... 4 (No 1, 5) are:

① T_1 T_2 T_3 T_4 T_5

② T_3 T_4 T_1 T_2 T_5

∴ We have two VESS.

- The only difference b/w Conflict Serializability & View Serializability is that CS doesn't discriminate b/w "types of write op^{n}s", but VS treats "blind writes" differently & takes advantage of it.

So, VS & CS only differ if Blind Write is present in Schedule.

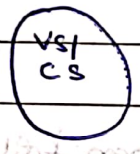
Ex. T_1: $W(x)$ $W(x)$

T_2: $W(x)$

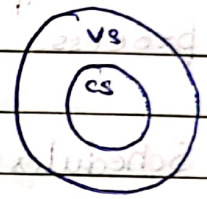
The schedule is not CS. But it is view serializable w/ VESS: $T_2 \rightarrow T_1$, since the write op^{n}s are blind.

So, if blind write is present in schedule, it may be Conflict Serializable, or it may be (not CS but VS), or neither. If no blind write, then it may be (View S. & Conflict S.) or neither, i.e. if no blind write allowed, then set of all VS schedules & set of all CS schedules are same.

No Blind Write Allowed.



In General.



Ex. T_1: W(A) Blind write. CS. VS.
T_2: R(B)

Ex. T_1: W(X) W(X) Blind Write. Not CS. VS.
T_2: W(X)

Ex. T_1: R(A) W(A) No Blind Write. CS. VS.
T_2: R(A)

- Although, the def^n of View Serializability covers a lot more schedules {by taking advantage of Blind Writes}, it still only focuses on Read/write op^{n}s & not all op^n.
So, (Set of View Serializable Schedule) is proper subset of (set of Serializable schedules).

* Recovery & Recoverability

- Once a transaction commits (i.e. execute "Commit" opⁿ), its changes must persist in the database.
(This is said by Durability Property of ACID)

- If a system failure occurs, we can search for all the transactions that have not executed the "Commit" opⁿ yet. These transactions may have to be rolled back to undo their effect on the database. This is called Recovery process.

- Schedules can be of two types. When failure occurs, schedule may be recoverable or irrecoverable. Recoverable schedules can further be devided as Cascadeless Recoverable (easier) & Cascading Rollback (difficult).
Irrecoverable schedules are the one which can't guarentee that in case of failure occurs, recovery will be possible.

Ex.

	0	1	2	3	4	5	
T_1:	R(A)	W(A)			R(B)		...
T_2:			R(A)	Commit			

[when last opⁿ not specified; Commit is assumed.]

When T_2 commits, it shows value of A read after T_1 writes A.
Suppose at 5 failure happens. Now since T_1 isn't commited, its write opⁿ on A will be rolled back. There is no way to make T_2 aware that T_1 has rolled back & the value of A it gave to user, doesn't represent true database state.
∴ This schedule makes DB inconsistent (by showing wrong result) & the schedule is irrecoverable, since DB can't be made consistent on some failure.

Note: When commit is executed, assume:

- Transaction executed successfully.
- Changes made permanent (immediately or eventually).
- Result is shown to user.

- **Irrecoverable Schedule:** A schedule is irrecoverable iff there are two transactions T_i & T_j such that T_j reads some DB element written by T_i & commits before T_i executes its Commit / Abort opⁿ.

T_i	T_j	
⋮	⋮	
W(x)		{Dirty Read}
⋮	R(x)	
⋮	COMMIT	
⋮		
COMMIT/ABORT		

This violates Atomicity property. If T_i fails, its effect should be nullified on the DB, but since T_j already read & persists the changes made by T_i, T_i violates Atomicity.

- **Recoverable Schedule:** A schedule is recoverable iff, all such pair of transactions T_i & T_j, such that T_j reads some DB element written by T_i, must allow T_i to commit / Abort before T_j

Ex. Is it Recoverable:

T_1: W(A)

T_2: R(B) Commit

Yes. T2 isn't dependent on T_1.

Ex. Is it recoverable:

T_1: W(A)

T_2: W(A) R(A) W(A) Commit

T_2's R(A) is reading value of A written by T_2 itself.
∴ T_2 isn't dependent on T_1.

Ex. Recoverable?

T_1: W(A) R(B) Abort

T_2: W(A) Commit

Yes. W(A) by T_2 is blind write.

Ex. Recoverable?

T_1: W(A) Com.

T_2: R(A) Com.

T_3: R(A) W(A) Com.

T_3 reads A written by T_1 & commits before T_1's Commit / Abort. ∴ Irrecoverable.

Ex. T_1: W(A) R(B) C

T_2: W(B) W(A) C

Recoverable.
Not Conflict Serializable. Not View Ser.

Ex. T_1: W(A) W(B) C

T_2: W(A) R(B) C

Irrecoverable. B is read by T_2 from T_1 & commited before
Conflict Serializable

- Serializability & Recoverability have nothing to do with each other, i.e., both concepts are independent.

- To preserve 'Atomicity' when schedule is recoverable, Undo operation needs to be performed.
Say for T_i, we want to preserve Atomicity when failure happens, then we Undo T_i & all transactions that are dependent on T_i (i.e. read changes made by T_i).

Ex.

T_1	T_2	T_3	T_4
W(A)			
R(A)			
	R(A)		
		R(A)	
		W(A)	
			R(A)
Abort			

So, which transactions are dependent on T_1? T_2, T_3, T_4.
T_4 is transitively dependent on T_1.
∴ If T_1 is aborted, T_2, T_3 & T_4, all must be aborted.

- Cascading Rollback: In case of failure, if a transaction has to rollback, which results in rollback of dependent transactions, which inturn results in further rollbacks & so on. This situation is called Cascading Rollback. This is expensive.
So, for this to not happen, we have to make sure that schedules don't have transactions reading from uncommited transaction. This type of schedule is called Cascadeless schedule.

Notice that Cascadeless schedules are always recoverable.

- So, a recoverable schedule can be Cascadeless schedule or Cascading Rollback schedule.

- Strict Schedule: No transaction should read or write to a data item X, if some other transaction has written to X, but hasn't commited yet. Strict schedules are always cascadeless.

T_i	T_j
⋮	
W(X)	
⋮	No Read or Write on X
Commit	

Ex. T_1: W(A) C

T_2: W(A) R(A) C

This is Recoverable, & cascadeless, since R(A) of T_2 isn't from T_1: W(A), so no dirty read.
This isn't Strict schedule, since b/w T_1: W(A) & T_1: Commit, T_2 writes & reads A.
Cascadeless since T_2: R(A) doesn't read A from any other uncommited transaction.

Ex. T_1: R(X) R(Z) W(X) C

T_2: R(Z) R(Y) W(Z) W(Y) C

T_3: R(X) R(Y) W(Y) C

This is strict, hence cascadeless, hence recoverable.

Ex. T_1: R(x) R(z) W(X) C

T_2: R(z) R(Y) W(z) W(Y) C

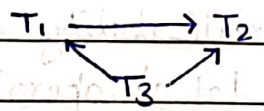
T_3: R(X) R(Y) W(Y) C

Not Strict. T_2 reads Y b/w T_3: W(Y) & T_3: C.

Not Cascadeless. —— " ——.

Not recoverable. T_2 reads Y written by T_3 & commits before T_3.

C Serial.:



∴ Conflict Serializable.

CESS: T_3 T_1 T_2.

∴ View Serializable.

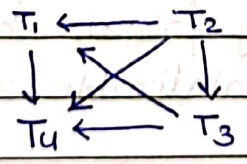
Ex. T_1: W(x) C

T_2: R(X) W(Y) R(Z) C

T_3: W(X) C

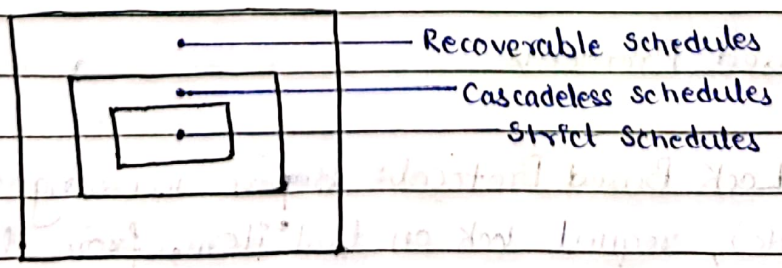
T_4: R(X) R(Y) C.

Strict schedule.



∴ Conflict Serializable.

CESS: T_2, T_3, T_1, T_4



* Concurrency Control Schemes

- Concurrency Control ensures the Isolation property when transactions are executing concurrently.

- Till now, we've only "checked" serializability given a schedule, not ensured it.
  Ensuring means, to make sure that at no point the schedule becomes non-serializable.
  If we only "check" serializability after schedule completes, we'll have to undo a lot of operations, if schedule is found to be non-serializable.
  So we rather ensure that schedule doesn't become non-serializable as the schedule executes.

- One simple idea is to only allow serial execution. But this scheme yeilds no good result since it results in serial schedule.

- Concurrency Control Schemes are set of protocols that every transaction has to individually follow, which inturn automatically ensures that only conflict serializable schedules are generated.

* Lock Based Protocols

- General Lock Based Protocol: Before accessing any item, (read or write), request lock on that item, from the concurrency control manager".
  If many transactions only want to read the item, then exclusivity isn't required, so, a "Shared Lock" is used.

If any transaction wants to write (& maybe also read) so only one transaction should be allowed to do so. So, exclusivity is required. So, "Exclusive Lock" is used.

T_i already holds \ T_j wants	Lock_S(A)	Lock_X(A)
Lock_S(A)	Grant	Deny
Lock_X(A)	Deny	Deny

Ex. Say, a transaction has to read & after say 500 op^{n}s it has to write to A. Can it first take shared lock on A & when has to write, it will take exclusive lock? No. This is called Lock Conversion. Although lock conversion can be implemented, it is not trivial, i.e., special considerations have to be made to allow lock conversion. So, by default, the transaction would initially have to take exclusive lock.

- Remember, that these rules are followed by individual transactions (not by the schedule).

Ex. T_1: R(A) W(B)

T_2: W(A) R(B)

- Scenario 1: Unlock as soon as work is done.

T_1: L_S(A), R(A), U(A), L_X(B), W(B), U(B)

T_2: L_X(A), W(A), U(A), L_S(B), W(B), U(B)

But this strategy can lead to the schedule becoming non-serial.

Say,

T_1	T_2	Concurrency Manager
L_S(A)		
		Grant(S, A, T_1)
R(A)		
U(A)		
	L_X(A)	
		Grant(X, A, T_2)
	W(A)	
	U(A)	
L_X(B)		
		Grant(X, B, T_1)
W(B)		
U(B)		
	L_S(B)	
		Grant(X, B, T_2)
	R(B)	
	U(B)	

So, this is a Conflict Serializable Schedule.

But,

T_1	T_2	CM
L_S(A)		This leads to
R(A)		$R_1(A)\ W_2(A)\ R_2(B)\ W_2(B)$
U(A)		This is not serializable.
	L_X(A)	
	W(A)	So, early unlocking
	U(A)	strategy can lead to
	L_S(B)	a non-serializable schedule.
	R(B)	
	U(B)	
L_X(B)		
W(B)		
U(B)		

- So, the "General Locking Protocol" can lead to non-serializable schedules, so isolation is violated. This can also lead to deadlock. (So DB consistency violated)

Ex. [... contd]

Strategy 2: Delayed Unlock.

T_1: L_X(A) W(A) L_S(B) R(B) U(A) U(B)

T_2: L_X(B) W(B) L_S(A) R(A) U(B) U(A)

This can lead to a schedule which is deadlocked

T_1	T_2
L_X(A)	
W(A)	
	L_X(B)
	W(B)
L_S(B)	
	L_S(A)

"Consistency of DB maintained, always, but prone to deadlock & starvation."

. . . . . At this point T_1 holds lock on A & T_2 has requested. So T_2 waits for T_1. T_2 holds lock on B & T_1 has requested. So, T_1 waits for T_2. ∴ Deadlock.

- Hence, General Locking Protocol can lead to Deadlock, Starvation, violation of isolation, consistency of DB, etc. So, this isn't a good protocol.

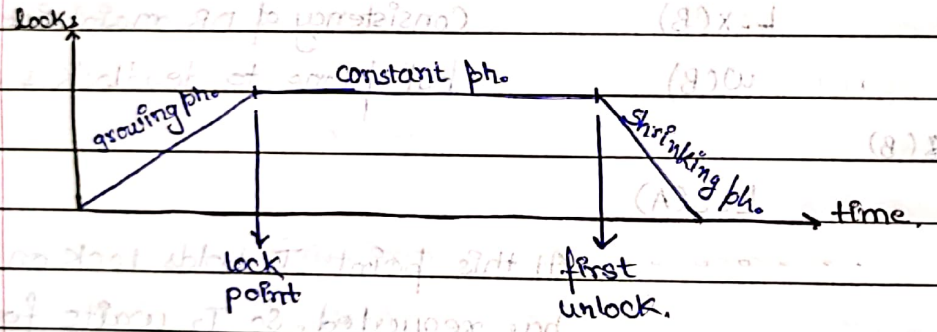
- We generally prefer "deadlocked but always consistent" over "sometimes inconsistent but never deadlocked", since deadlocks are rare, and can be broken easily.

Most locking based protocols are prone to deadlock & starvation, but they must maintain consistency.

Two Phase Locking Protocol : Each transaction follows all the rules of General locking protocol, and additionally, locking & unlocking only happen in two phases. In the first phase the transaction can only "acquire" locks. Once the transaction unlocks the first lock, second phase starts & the transaction can now only unlock.

The first phase is called locking phase or growing phase. The ~~last~~ second phase is called unlocking phase or shrinking phase.

The point where the transaction obtains the last lock is called as "Lock Point".



2PL guarentees that ~~the schedule~~ if every transaction follows 2PL Protocol, then every schedule generated by those transactions would be Conflict Serializable.

Ex. T_1: L_X (B); R(B); B=B-50; W(B); L_X(A); R(A); A=A+50; W(A); U(B); U(A);

first unlock — Lock Point

Ex. T_1: L_X(B); W(B); L_X(A); R(A); U(B); A=A+50; W(A); U(A);

Lock point — first unlock

- If every transaction follows 2PL, then generated schedule would be Conflict Serializable & the CESS would be in the order of the transaction's' lock points.

Note: Given a schedule to check if it can be generated by 2PL or not, try to generate it following the rules of 2PL.

Ex. Can the following schedule be generated by 2PL scheduler:

T_1:		W(A)		W(B)		
T_2:			R(B)		R(D)	
T_3:	R(D)			W(D)		
	LX(D)	LX(A)	LS(B)	U(D)	U(A) U(B)	U(D)

Here LX is req. but it won't be granted, since T_2 holds LS on B & can't unlock before R(D)
So, the foll. op^{n}s need to be executed:
T_2: LS(D); U(B); T_1: LX(B)

So, the op^{n}s are:
$LX_3(D)$ $R_3(D)$ $LX_1(A)$ $W_1(A)$ $LS_2(B)$ $R_2(B)$ $W_3(D)$ $U_3(D)$ $LS_2(D)$ $U_2(B)$ $LX_1(B)$ $W_1(B)$ $U_1(A)$ $U_1(B)$ $U_2(D)$.
This schedule is guarenteed to be Conflict S.
We can also check the CESS by checking lock point order:
$T_3 \rightarrow T_2 \rightarrow T_1$
(We can verify this by looking at schedule itself).

Ex. T_1: W(A) W(B)
T_2: R(A) W(A)
T_3: W(B) R(B)

Ex.

	0	1	2	3	4	5	6	7	8
T_1:	W(A)			W(B)					
T_2:		R(A)				W(A)		R(B)	
T_3:			W(B)		W(B)		R(B)		

0: LX(A)

1: Here T_2 req. LS(A) but it can't get it until T_1 does U(A), which T_1 can't do before T_3 locks & unlocks B.

∴ This can't be produced following ~~strict~~ 2PL.
Also, we can easily show this is not Conflict S.

- w.k.t If (all trans. follow 2PL) then (generated schedule is Conflict Serializable)

∴ If (generated schedule not CS) then (not all tran. follow 2PL)

Ex.

	0	1	2	3	4	5
T_1:	W(X)				W(Y)	
T_2:		R(X)		R(Z)		
T_3:			R(Y)			

This is CS, so we'll have to check by "trying to generate following 2PL rules."

0: $LX_1(X)$

1: Here T_2 needs $LS(X)$, which T_1 holds, so T_1 must first lock Y to start unlocking. ∴ $LX_1(Y)\ U_1(X)\ LS_2(X)$

2: T_3 requires $LS(Y)$, but T_1 won't unlock Y untill 5. So, this can't be generated by all trans. following 2PL.

Ex.

	0	1	2	3	4	5
T_1	W(A)			W(B)		
T_2		R(B)			R(D)	
T_3			W(D)			

CS. So check:

0: $LX_1(A)$

1: $LS_2(B)$

2: $LX_3(D)$

3: $U_3(D)$; $LS_2(D)$; $U_2(B)$; $LX_1(B)$;

4: $U_1(A)$; $U_1(B)$;

5: $U_2(D)$

Lock Point order : $T_3 \rightarrow T_2 \rightarrow T_1$

- 2PL protocol also suffers from deadlock (& hence starvation).
- 2PL doesn't guarentee Recoverability, so DB consistency may get violated, (due to Atomicity violation).

Ex.

T_1	T_2
R(A);	R(B);
R(B);	R(A);
if A=0 then B=B+1;	if B=0 then A=A+1;
W(B);	W(A);

Add lock & unlock instn to T_1 & T_2 such that they follow 2PL. Can their execution lead to deadlock?

T_1: $LS(A)$; $R(A)$; $LX(B)$; $U(A)$; $R(B)$; if A=0 then B=B+1; $W(B)$; $U(B)$;

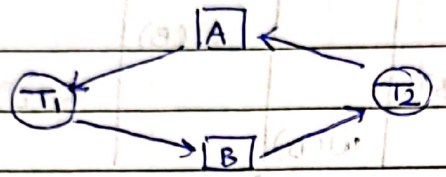
T_2: $LS(B)$; $R(B)$; $LX(A)$; $U(B)$; $R(A)$; if B=0 then A=A+1; $W(A)$; $U(A)$;

There can be a possible situation of deadlock if T_1 holds $LS(A)$ & requests

LX(B), while T_2 holds LS(B), while requesting LX(A).
i.e.

$LS_1(A)$; $R_1(A)$, $LS_2(B)$; $R_2(B)$; $LX_1(B)$; $LX_2(A)$;

This now is deadlocked.



Ex. Is the following schedule recoverable? Can 2PL generate it?

$W_1(A)$; $R_2(A)$; $W_1(B)$; $W_2(B)$; C_2; C_1;

This isn't recoverable. T_2 reads A written by T_1 & commits before.

$T_1 \longrightarrow T_2$

This is Conflict S, so we check 2PL manually.

$LX_1(A)$ $W_1(A)$ $LX_1(B)$ $U_1(A)$ $LS_2(A)$ $R_2(A)$ $W_1(B)$ $U_1(B)$ $LX_2(B)$ $W_2(B)$ $U_2(A)$ $U_2(B)$ C_2 C_1

So, the schedule can be generated with each trans. following 2PL.

(Hence 2PL can generate irrecoverable schedules).

- So, 2PL doesn't really guarentee anything related to Recoverability, i.e., 2PL can lead to irrecoverable, cascading rollback, cascadeless schedule, etc.

- For 2PL to guarentee both Serializability & Recoverability, there are two variations:
  - Strict 2PL
  - Rigorous 2PL.

(Both are still prone to deadlock [& hence starvation])

- Strict 2PL: In strict schedule we ensured that a trans. T_j can R/W on db element X, previously written by T_i only after T_i commits.
To enforce this in 2PL, we can make sure that every transaction that acquires exclusive lock on db element X, then it only unlocks X after it commits.

So, every transaction releases exclusive locks only after commit or abort.

- Rigorous 2PL: Every transaction releases any lock that it holds only after commit or abort. So, the CESS would be in order of commit opⁿ, since unlocking starts in that order only.

- Both strict & Rigorous 2PL ensure Conflicts. & strict schedules Strict schedules are by default cascadeless & recoverable.

- Conservative 2PL: To ensure deadlock never occurs (prevention) we have an approach where we take all resources before starting only.
In Conservative 2PL, each transaction acquires all required locks before starting any other opⁿ. Once the actual opⁿ start, they are in Shrinking phase.
So, C2PL ensures deadlock never occurs. Starvation still possible. Recoverability not guarenteed.

Note: All trans. follow 2PL → Schedule C.Serializable (one way).

	Conflict Serializability	Recoverability	Deadlock Freedom	Starvation Freedom
2PL	✓	✗	✗	✗
S2PL	✓	✓	✗	✗
R2PL	✓	✓	✗	✗
C2PL	✓	✗	✓	✗

- Deadlock: In a system w/ multiple transactions, if there exists a set of transactions such that every one of them is waiting for some other transaction.

If the system is already in deadlock, then one or more transaction will have to be rolled back.

We already saw a deadlock prevention scheme w/ Conservative 2PL. This approach is not very practical since not always known which resource used. Another prevention scheme is ordering the data items & allowing transactions to acquire lock on data items only in specific order. This is also not always practical since transaction author may not always know the ordering

- Timestamp Based Approach: Each transaction is assigned a unique timestamp when created, such that if T_i is created before T_j then $TS(T_i) < TS(T_j)$
  - Scheme 1 (Wait-Die Approach): If a trans. T_i wants to lock X, while T_j holds lock on X,

    C1: $TS(T_i) < TS(T_j)$: Wait for T_j to unlock X

    C2: $TS(T_j) < TS(T_i)$: T_i aborts.

    [This is non-preemptive approach]

- Scheme 2 (Wound-Wait): If a trans. T_i wants to lock X, while T_j holds lock on X:
  - C1: $TS(T_i) < TS(T_j)$: Snatch from T_j, i.e. abort T_j, lock X.
  - C2: $TS(T_j) < TS(T_i)$: wait for T_j to unlock X.
  - [This is preemptive approach]
- When a transaction aborts (always the younger), it restarts w/ original TS only so as to age & avoid starvation.

* Timestamp Based Protocols

- In 2 Phase locking, the serializability order is decided during execution time (based on lock point).
- In TS Based protocol, the order of serializability is already decided before it is executed (based on the timestamp).
- Two transactions can never have same timestamp. Hence the order of serializability is same as the order in which the transactions come into the system.
- TS Ordering Protocol says: "If you are late, then you rollback".

Ex. Suppose T_1 comes before T_2: So, $TS(T_1) < TS(T_2)$. Serializability order: $T_1 \rightarrow T_2$.

Scenario 1:

T_1	T_2
R(X)	
	R(X)

This has no problem.

Scenario 2:

T_1	T_2
	R(X)
R(X)	

This also has no problem.

S3:

T_1	T_2
W(x)	
	R(X)

This is fine, since serially T_1 should come first & hence T_2 reads value written by T_1.

S4:

T_1	T_2
	R(x)
W(x)	

This is wrong since T_1 should serially write to X first, but it's late, so T_1 rollbacks.

- So, to implement TS ordering, for each DB element, its latest read & latest write timestamp is maintained.
  For X we maintain RTS(X) & WTS(X). These values are set equal to the TS of youngest (latest) transaction that R/W to it.

Ex. Suppose we have T_1, T_2, T_3 w/ TS 1, 2 & 3 resp.

T_1	T_2	T_3	X RTS	X WTS	Y RTS	Y WTS
R(x)			1	-	-	-
		W(Y)	1	-	-	3
	W(Y)		1	-	-	3
	R(Y)		1	-	2	3
R(Y)			1	-	2	3
	W(x)		1	2	2	3
		W(x)	1	3	2	3
		R(Y)	1	3	3	3

- So, for a DB element X,
  - If T_i successfully reads X: $RTS(x) = \max[RTS(x), TS(T_i)]$
  - If T_i successfully writes to X: $WTS(x) = \max[WTS(x), TS(T_i)]$

- Reading Rules: Suppose for DB element X, transaction T is trying to read X:
  - If TS(T) < WTS(X), then abort T. Restart T with new timestamp.
  - If TS(T) ≥ WTS(X), then allow T to read X. Also update RTS(X) = max[RTS(X), TS(T)]

- Writing Rules: Suppose for DB element X, transaction T is trying to write to X:
  - If TS(T) < RTS(X), then abort T. Restart T with new timestamp.
  - If TS(T) < WTS(X), then abort T & restart T with new timestamp.
  - Else, allow T to write to X. Also update WTS(X) = max[WTS(X), TS(T)] = TS(T).

Note: Above rules are called "TS ordering rules" or "Basic TS protocol".

- Thomas Write Rule: This rule tries to optimize one of the write rules of basic TS protocol.
  The second write rule means that, X has already been written by a younger transaction, so don't allow T to write to X. Additionally, T is rolled back.
  But this rollback is unnecessary. We can just ignore this write opⁿ of T to X, since anyway this value should have been over-written by the younger transaction.

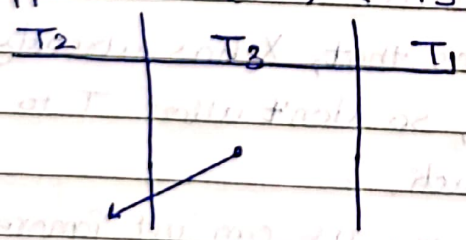
Ex.	T_1	T_2	A RTS	A WTS	B RTS	B WTS
0:			0	0	0	0
1:		R(B)	0	0	2	0
2:	R(B)		0	0	2	0
3:		R(B)	0	0	2	0
4:		W(A)	0	2	2	0
5:	W(A)					

So under basic TS protocol op^n 5 creates problem. A has already been read by T_2 & T_1 is tring to write to A. So, T_1 aborts, rolls back & starts w/ new TS.

If Thomas Write Rule is used, then op^n 5 can simply be ignored & no rollbacks required.

Trick: In the schedule, write the transaction in increasing order of TS. If then there is a conflict happens in right to left manner then the schedule is not allowed by Basic TS protocol.

- Suppose $TS(T_2) < TS(T_3) < TS(T_1)$, then schedule:



If such a conflict happens ↙, then not allowed by basic TS protocol.

- With Thomas Write Rule only write write conflict of ↙ type is allowed.

- Also since ↙ type of conflict isn't possible under TS ordering, the system can never have circular wait. So, deadlock is prevented.

Basic

- ↑TS ordering ensures conflict serializability & hence serializability.
- (TS ordering + TW Rule) ~~doesn't guarantee con~~ allows non-conflict serializable schedule as well. But it ensures correctness.

Ex.

T_1	T_2
R(X)	
	W(X)
W(X)	

This $w^{\swarrow w}$ conflict is allowed by Thomas Write rule but the schedule is not conflicts.