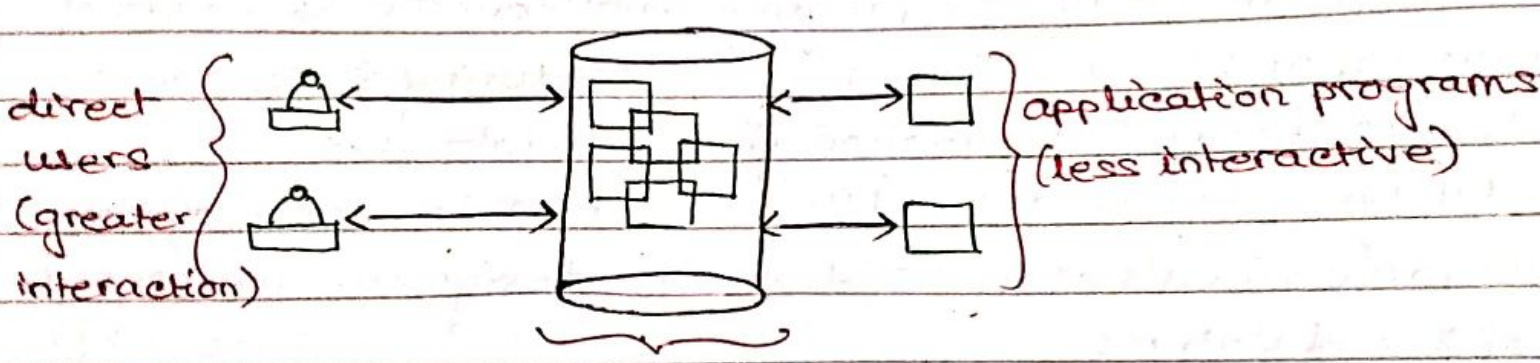
DATABASE MANAGEMENT SYSTEM

Database:
A database is a collection of stored operational data used by the application systems of some particular enterprise
(R.W. Engles)

Database components: The components of a database are:
database: A large collection of data, stored in secondary storage.
application programs: Run against the data, operating on in all usual ways.
on-line users: Interacting through terminals, performing all functions (mainly retrieval)



(overlapped units: portions of database viewed by single users / groups - based on requirement / access rights etc.)

- Interaction with a database, primarily includes - retrieval, insertion, deletion, updation (via direct users application programs)

The database is integrated, so that the database contains data for many users, and not just one. This implies:

i) Any one user will be concerned with just a small portion of it.

ii) Different user's portion, will overlap in various ways, i-e., individual pieces of data may be shared by different users.

Online users:

end-users: Most interactive.

- casual users: users accessing the database with some commercial query language.
- naive users: users accessing the database through menues.

application programmers: Write the menu applications used by naive users. The program must foresee the needs of the users, and be able to pose queries during execution, to retrieve desired information from the database.

database administrator (DBA): The DBA is a team of computer professionals responsible for the design and maintenance of the database.

An enterprise is a reasonably large organization - a manufacturing company, university, hospital, government organization that maintain data that can form a database.

Operational data: It does not include any purely transient data (I/o, work queue etc.). Transactions may cause a change to the operational data, but are not a part of the database. (Transactions in hotel management systems, airline reservation systems etc. - may be required to be stored

as operational data, as customers may request a detailed transaction listing.) Associations or relationships between data, must also be stored as part of operational data. E.g: For a manufacturing company - details of projects handled, use of parts - supplied by suppliers - warehouse locations - employees : each being a distinguishable entity, has relationships / associations with one another.

Centralized database:

i) Centralized control of operational data.

ii) the DBA has the central responsibility of the operational data (taking care of critical section issues).

Tasks of a DBA:

i) Deciding information content.

ii) Designing the database.
-organize views, restrictions etc.

iii) Recovery mechanism (frequency of backup issues) and backups.

iv) Authorization checks.

v) Monitor and upgrade performance issues.

Advantages / implications of a DBMS:

i) The amount of "redundancy" in the stored data can be reduced. Independent applications may have their own private files leading to redundancy and wastage of storage space. With centralized control, this redundancy can be reduced.

ii) Inconsistency: The problem of inconsistency can also be

avoided to some extent. This property directly follows from redundancy removal, implying, two entries of the same information may lead to inconsistency if one is updated.

Remark: Sometimes, there are technical reasons for maintaining several distinct copies of the same data. Any such redundancy should be carefully controlled.

iii) Data sharing: Data can be shared, i.e., not only the existing applications can share the data in the database, but also, new applications can be developed to operate against the same shared data.

iv) Standards: Industry standards can be enforced and maintained by the DBA. This simplifies problems of maintenance and data interchange between installations.

v) Security: Security restrictions can be applied. The DBA can ensure that the only means of access to the database is through proper channels and hence can define authorization checks to be carried out, when access to sensitive data is attempted. This also implies privacy across various user departments.

vi) Integrity: Integrity can be maintained. The data values stored in the database, must satisfy certain types of consistency constraints. The problem of integrity is the problem of ensuring that the data in the database is accurate.

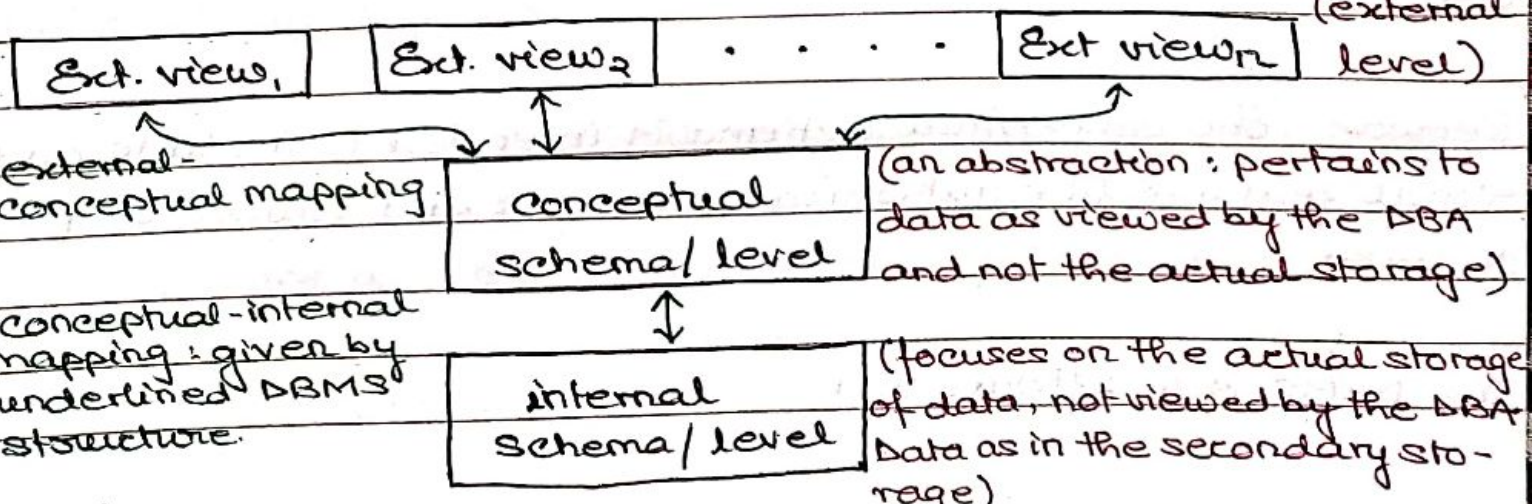
vii) Ease of application development: Cost and time for developing new applications are reduced. Studies show that, a programmer can develop an application 2-4 times faster. The reason being that the programmer is free from designing, building and maintaining master files.

viii) Data independence: Separation of data from the application program environment. Organization of data can change and evolve, without any change in the application program. E.g: field-size change, file-organization change etc. It is a major objective of database systems and may be defined as the immunity of applications to change in the storage structure and access strategy, i.e., the applications concerned donot depend on any particular storage structure or access strategy. May be viewed as a 2-stage independence

- logical data independence: capacity to change the conceptual schema, without having to change external schemas or application programs.
- physical data independence: capacity to change the internal schema, without having to change the conceptual schema.

Schema of a database, stands for the description of a database.

Database architecture: 3-schema architecture (ANSI/SPARC architecture):



Internal level: This level has an internal schema, which defines the physical storage structure of the DBMS, i-e., how the database is actually stored (depends on the data model of the DBMS). It describes the complete details of data storage and access paths for the database.

Conceptual level: This level is defined by a conceptual schema through the Data Definition Language (DDL) of the DBMS package used. This is done by the DBA, who decides what information is to be kept in the database. This is a representation of the entire information content in the database, in a form that is somewhat abstract, in comparison to the way in which the data is actually (physically) stored.
(E.g: physically a B-tree, but conceptually a table) It may be completely different from the way data is viewed by the user. It describes what data are actually stored in the database, and the relationships that exist among the data.

External level: This level is defined by a number of external schemas or views. Most database users will not be concerned with the entire database, but need just a part of it. Thus, there are many external views of it, which basically consists of definitions of various external record types in that view.

Remark: The conceptual schema is intended to include additional features like authorization checks and validation procedures.

The DBMS is a software that:

i) Allows the DBA to define the conceptual and external model through the DDL. In most cases, the physical model definition is part of the package.

ii) Allows users to manipulate data through Data Manipulation Language (DML) commands and routines.

iii) Handles all accesses to the database, i.e., controls the overall operation of the database.

. Data Definition Language: A high-level non-procedural language. It is a notation for describing the entities and relationships among the data entities, in terms of a particular data model. It is used to:

i) Express the design of the database.

ii) Modify the design.

iii) Describe in abstract terms what the physical layout of the database should be.

[Specify domain constraints, referential integrity, assertions, authorizations]

Data Manipulation Language: Used to manipulate data, i.e, perform storage (insert, deletion, retsriève etc.) operations.

Remark: The DDL and the DML depend on the DBMS package which in turn depends on the data model.

Data Models: In a centralized database, there are 3 types of data models which form the heart of a database:

i) Relational ii) Hierararchial iii) Network.

• Relational Model:

Supplier: (entity: provides complete information)

S#	S_name	City
S1	B. Raghuram	Mumbai
S2	A.K. Sharma	Kolkata

- table synonymous with relation
- record synonymous with instance or tuple.

Parts: (entity)

P#	P_name	colour	wt.	City.
P_1	Nut	Red	12	Kolkata
P_2	Bolt	Green	17	Mumbai
P_3	Screw	Blue	17	Delhi

SP : (A table that symbolizes a relation more than an entity)

S#	P#	Qty
S_1	P_1	300
S_1	P_2	200
S_1	P_3	400
S_2	P_1	300
⋮	⋮	⋮

The statement: Supplier no. S1 of name B. Raghuram stays in Mumbai: depicts a relationship which is best explained in the SUPPLIER table.

i) Entities and associations are viewed as tables (or relationships)

ii) Most convenient form for the users.

iii) For every tuple, a relation exists between every attribute of the table, and the same relation holds for every tuple within a relation. And thus the name relational model.

Within a given relation, there is one attribute (may be a composite attribute of 2 or more attributes) with values that

uniquely identifies the tuples of the relation. This is called the primary key of the relation.
In the given tables: S#, P# and (S#, P#) are primary keys of the tables supplier, parts and SP respectively.

Advantages:

i) User friendly representation.

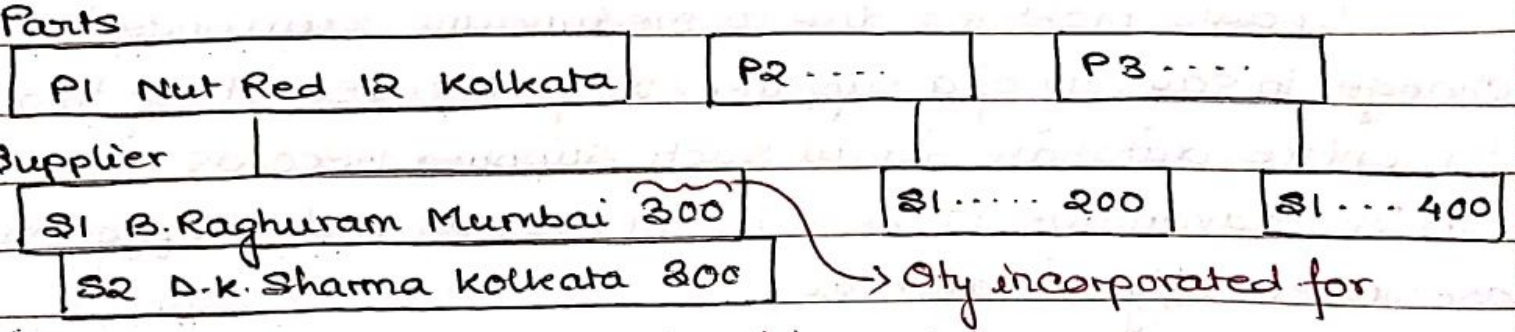
ii) Given a query:

- find S# for suppliers who supply P# = 'P1'.
- find p# for parts supplied by 'S# = 'S1".

Both the queries incorporate similar search types and are symmetric. In the relational model, symmetric query solving is symmetric and poses no problem.

iii) Storage operations (insertion, deletion, updation) are simple to make.

- Hierarchial Model:



(we might have placed "supplier" at the root instead of "parts")

i) Entities are represented as tree structures. The structure implies the relationship.

ii) In this particular relation, "part" is superior to "supplier". For each part, there may be more than one supplier record occurrences. Each supplier record also contains the shipment

quantity. The association is represented by the parent-child relationship, hence the name.

iii) In general, the root may have any number of dependance. Each of these may again have any number of dependances and so on.

iv) Asymmetric structure - not convenient for user

Advantages and disadvantages:

i) Symmetric query solving is not symmetric.

ii) The hierarchial model possesses certain undesirable properties with respect to storage operations.

Insert: to introduce a new supplier, dummy parts are to be introduced unless the supplier supplies some part.

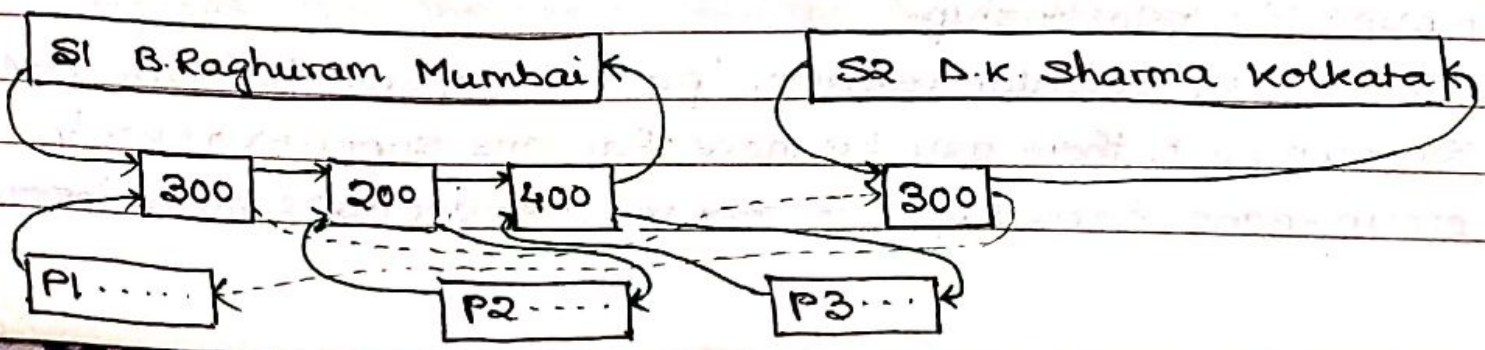
Delete: deleting a shipment information is to delete the supplier record, implying that we may lose all information about the supplier, if it happens to be the only shipment for the supplier.

Update: problems due to redundant occurrences. Any change, in say city of a supplier, requires searching through the entire database for all such supplier records.

iii) The advantage is the naturalness, as natural processes are inherrently hierarchial.

• Network Model:

An extension of the hierarchial model.



Remark: Data is represented as records and links. Entity occurs as records and associations as links. More general than hierarchial structure. Connector chain represents shipment quantities. No concepts of superiors and dependance. Each connector occurrence links only one supplier and only one part.

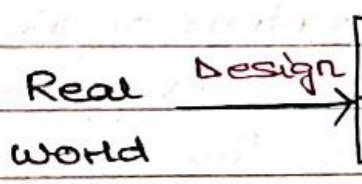
Advantages and disadvantages:

i) Symmetric query solving is symmetric.

ii) Storage operations are simple.

iii) Disadvantage is the complexity of links.

Relational database design:



• The output of a database design is the user views.

It is the responsibility of the DBA to design the database, assigning the related data items of the database to columns of tables (with respect to a relational model), in a manner that preserves desirable properties.

The database designer has to consider many issues at the same time. The final output of the logical database design process is the user's schema, since the user's schema represents the database designer's solution. The user's schemata are usually difficult to understand and change. The designer is constrained by the limited data-structure types supported by the database system (since the designer has to keep in mind the access paths. The search strategies are dependant on

the conceptual schema.)

The designer may have to consider the access paths of the record, i.e., how to access a particular record type. The designer may have to consider how to make the retrieval and updating more efficient.

There are two common technologies Entity-Relationship approach and Normalization approach towards database design. However, it turns out that the relational design based on either approach transform into relational form having nearly identical results, and in fact, the two approaches reinforce each other.

E-R approach:

The key idea to the E-R approach is to concentrate on the conceptual schema. At this stage, the designer should view the data from the point of view of the whole enterprise. This description is called enterprise conceptual schema or enterprise schema. This should be a pure representation of the real world and independent of storage and efficiency considerations. The design process can be viewed as a 2-phase process –

- Design enterprise schema.
- Translate enterprise schema to user schema for the database system.

Advantages of the 2-phase approach:

i) The database design process becomes simpler and better organized.

ii) The enterprise schema is easier to design than the final schema, since, it neednot be restricted by the capabilities of the data

base system, and is independent of storage and efficiency Considerations.

iii) Enterprise schema is more stable than the user's schema. If one wants to change from one database system to another, one would probably have to change the user's schema, but not the enterprise schema.

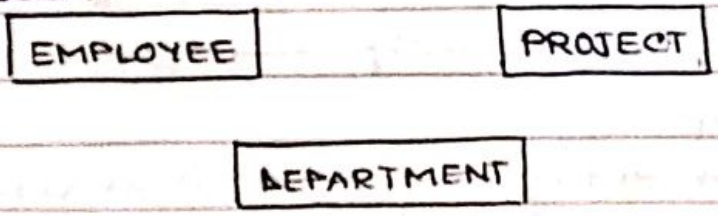
iv) The enterprise schema represented by the E-R diagram is more easily understood by non-EDP (electronic data processing) people.

E-R concepts :

The E-R approach defines a number of data classification objects. Three fundamental data classification objects are:

- entities
- relationships.
- attributes

Entity :- An entity is a collection of distinguishable real world objects, having common properties, and is of interest to the enterprise. Is represented as rectangular boxes in an E-R diagram.

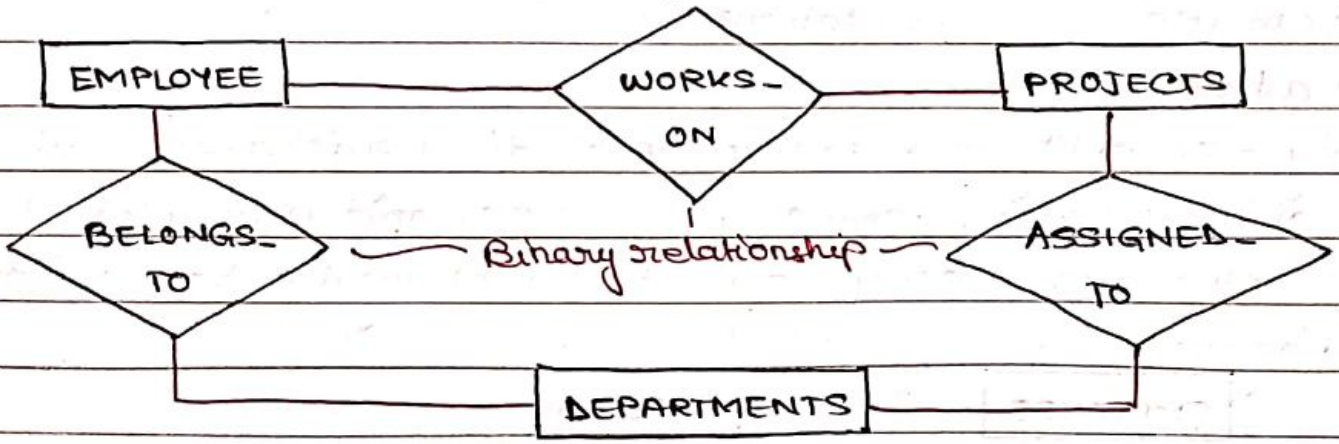


An entity is usually mapped to an actual table, and each row of the table corresponds to one of the distinguishable objects that make up the entity, called an entity occurence / entity instance.

Remark: There are many things in the real world, and only some of them are of interest to the enterprise. It is the responsibility of the database designer to select the entities, which are important.

Remark: Choice of entities is a key step to database design.

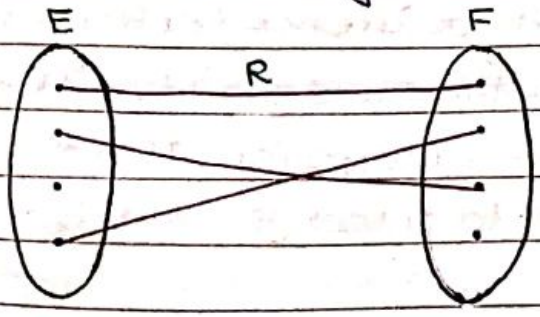
Relationship :- Given an ordered set of entities $E_1, E_2, \ldots, E_m$, (may not be distinct) a relationship R defines a rule of correspondence between the instances of these entities. Specifically, R represents a set of m tuples $\{(e_1, e_2, \ldots, e_m) \mid e_i \in E_i, 1 \leq i \leq m\}$ which is basically a subset of the cartesian product of the entities: $E_1 \times E_2 \times \cdots \times E_m$ (rather the entity instances). A particular occurence of a relationship, corresponding to a tuple of entity instances $(e_1, e_2, \ldots, e_m)$, where $e_i \in E_i$ and $1 \leq i \leq m$, is called a relationship instance / relationship occurence. m is the degree of the relationship. Is represented as diamond-shaped boxes in an E-R diagram.



Types of relationships:

a) one-to-one: For each entity instance in either entity, there is atmost one associated member of the other entity.

E.g: HEAD_OF (employee, department)

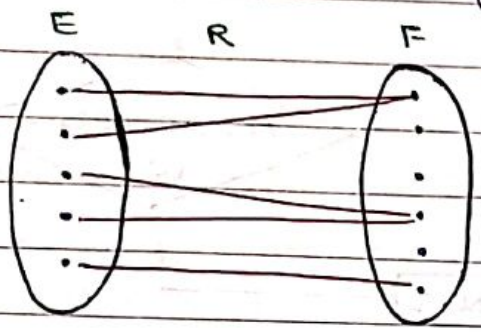


minimum_cardinality (E,R) = 0
maximum_cardinality (E,R) = 1
minimum_cardinality (F,R) = 0
maximum_cardinality (F,R) = 1

$f(x) =$
$f^{-1}(y) =$

b) many-to-one: A relationship is many-to-one, from entity E to entity F, if one entity instance in F is associated to 0 or more entity instances in E, but each instance in E is associated with almost one entity instance in F.

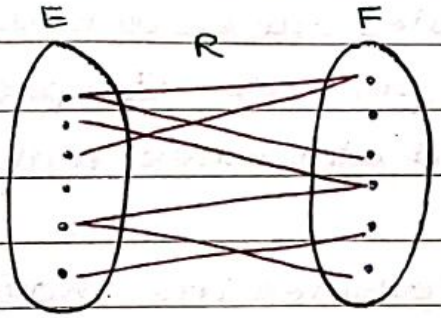
E.g: BELONGS_TO (employee, department)



minimum_cardinality (E,R) = 1
maximum_cardinality (E,R) = 1
minimum_cardinality (F,R) = 0
maximum_cardinality (F,R) = n.

c) many-to-many: A relationship is many-to-many from entity E to entity F, if one entity occurrence in F is associated with any number of entity occurrences of E, and each entity occurrence in E is associated with any number of entity occurrences of F.

E.g: WORKS_ON (employee, project)



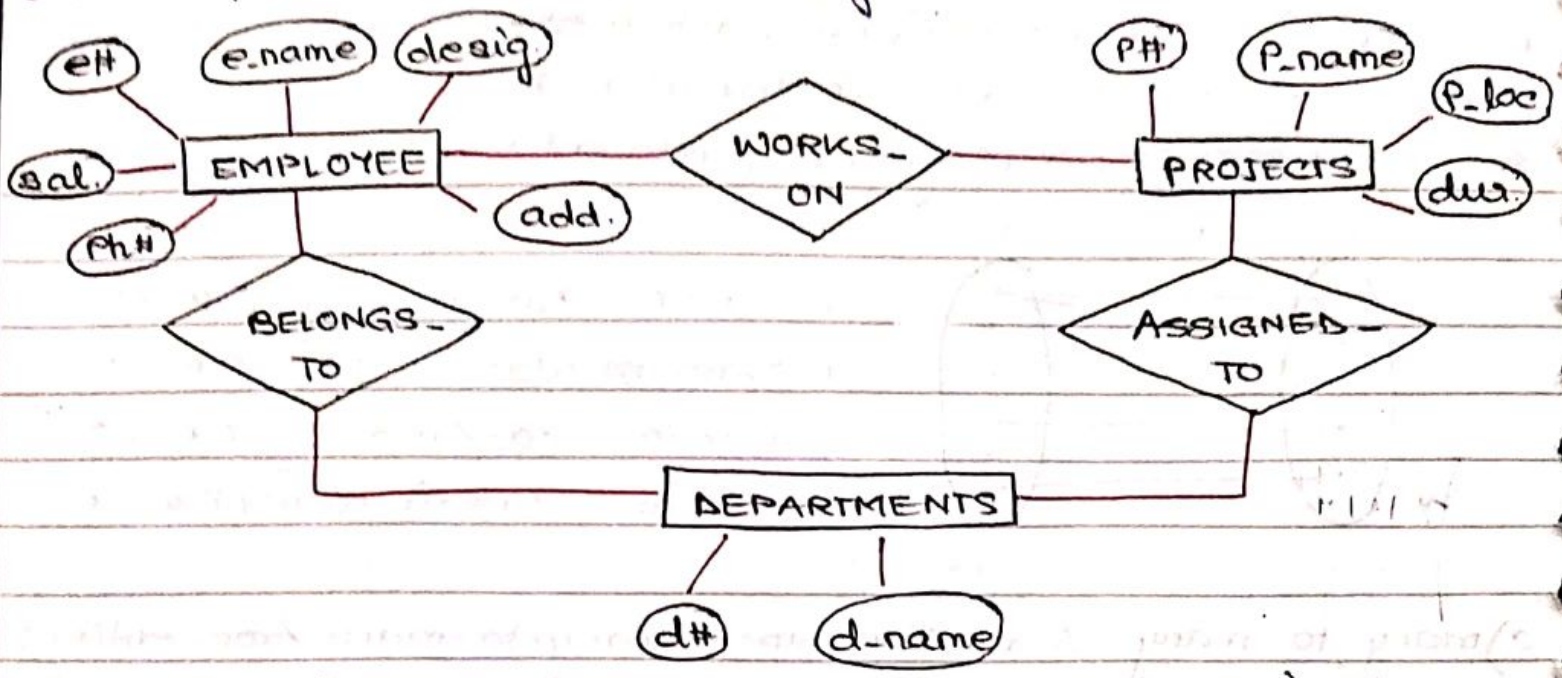
minimum_cardinality (E,R) = 0
maximum_cardinality (E,R) = n
minimum_cardinality (F,R) = 0
maximum_cardinality (F,R) = n.

Remark: There may be many types of relationships between entities, and some of them may not be of interest to the enterprise. The database designer is responsible for the selection of relationships relevant to the enterprise.

Remark: Choice of relationships is a key step to database design.

Attributes :- An attribute is a data item that describes a

property of an entity or a relationship. It is represented as oval-shaped boxes in an ER diagram.



(A bare outline of the ER diagram of the database)

Types of attributes:

a) Simple single-valued attributes: Simple single-valued attributes are called simple attributes. E.g: e_name, duration.

b) Composite attributes: Composite attributes can be divided into subparts. E.g: address = street + state + pin_code. Composite attributes help us to group together related attributes, making the modeling cleaner.

c) Multivalued attributes: multivalued attributes are those that can take on multiple values for a single entity instance. E.g: phone#, hobby.

keys:

a) Superkey: A superkey is a set of one or more attributes, that when taken collectively allows us to identify uniquely an entity instance in an entity. Say if e# were absent in the above database, then (e_name, address) could be treated as a superkey.

Again (e_name, designation, address) is also a superkey. Thus clearly, there may exist many superkeys; any superset of a superkey is also a superkey in an entity.

b) minimal super-key or candidate key: A superkey for which no proper subset is a superkey is said to be a candidate key. It is possible that several distinct set of attributes could serve as a candidate key. E.g: e# (simple), (e_name, add.) (composite)

c) primary key: The primary key is a candidate key that is chosen by the database designer as the principal means of identifying entity occurrences, within an entity. Usually, a primary key is used in references from other tables.

E.g: e#, p#, d# in employee, project and department respectively. (Not considering phone# as an attribute.)

Transformation Rules:

i) Each entity in an E-R diagram is mapped to a single table in a relational database. The table is named after the entity. Columns of the tables represent all the single-valued simple attributes that are attached to the entity. A primary key is selected for the entity. Entity occurrences are mapped to the rows of the table.

ii) Given an entity E, with primary key attribute p; a multi-valued attribute a attached to E in an E-R diagram, is mapped to a table of its own. The table is named after the plural multivalued attribute (E.g: PHONES). The columns of the new table are named after p (e#) and a (phone #), and the rows of the table correspond to (p,a) value pairs - representing all pairings of attribute values of a, associated with entity occurrences relative to p in E. The primary

key attribute for this table is the set of columns in p and a.

iii) When two entities E and F, take part in a many-to-one binary relation R, and the entity F represents the many side of the relationship, the relational table T, transformed from entity F should include columns constituting the primary keys from the table transformed from E. This is known as a foreign key in T

Remark: This foreign key cannot take on null values.

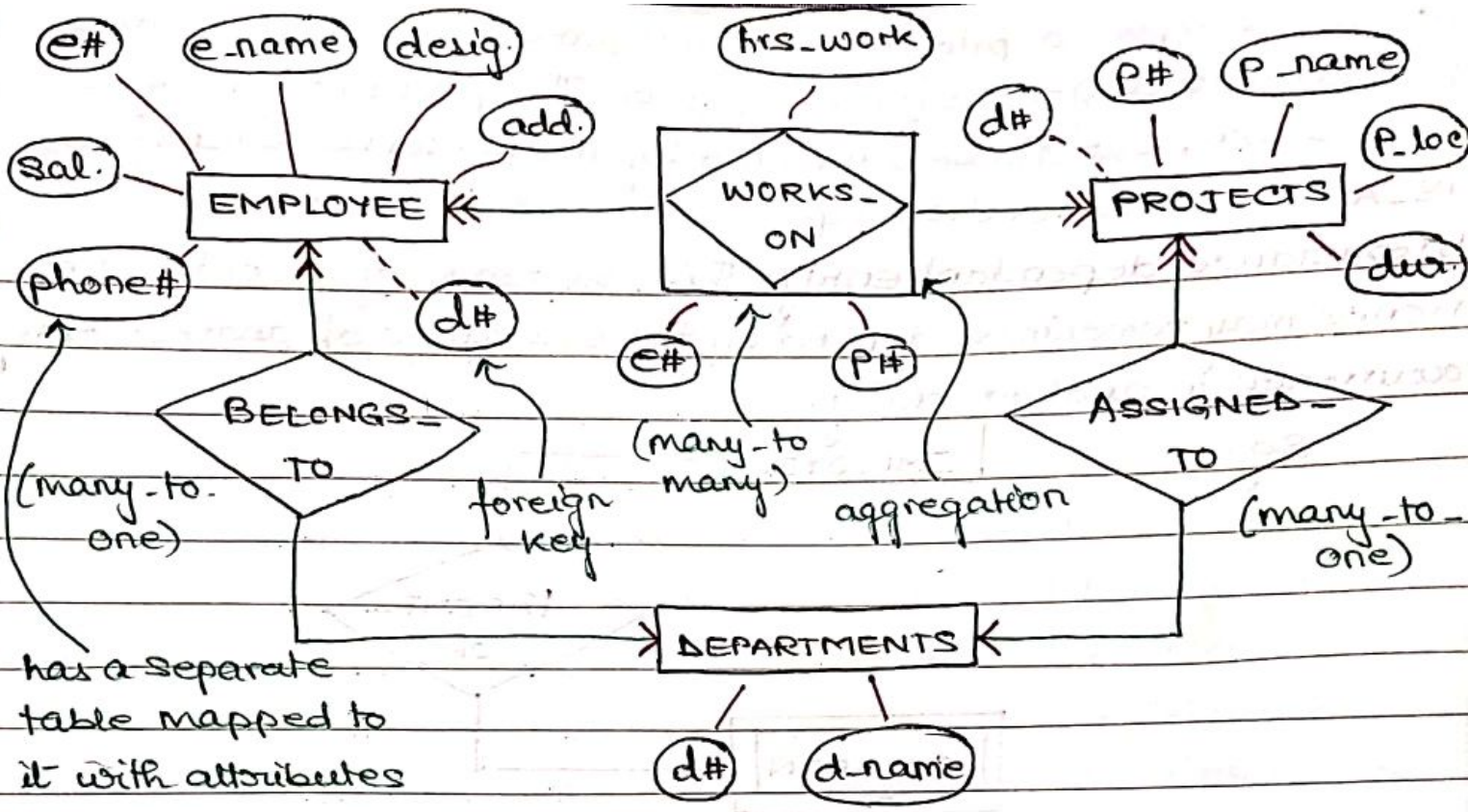
iv) When two entities E and F take part in a many-to-many binary relationship R, the relationship is mapped to a representative table T, in the related relational database design. The table contains columns for all attributes in the primary keys of both tables transformed from entities E and F, and this set of columns forms the primary key for T. It also contains columns for all attributes attached to the relationship

(Here, primary keys of either tables in the relationship are not attached to the transformed tables of the opposite entities, as this leads to data redundancy.

The phenomenon of treating a relationship like an entity, as described above, is termed aggregation.

Refinement allows answering queries pertaining to cross entity references in a relational database. Refinement brings forth the true essence of "relationships" in a relational database design.)

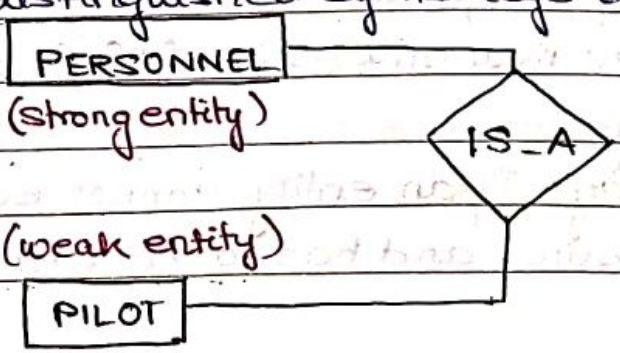
Refined E-R diagram, as compared to the bare outline of the E-R diagram:



has a separate table mapped to it with attributes e# and phone #

Definitions:

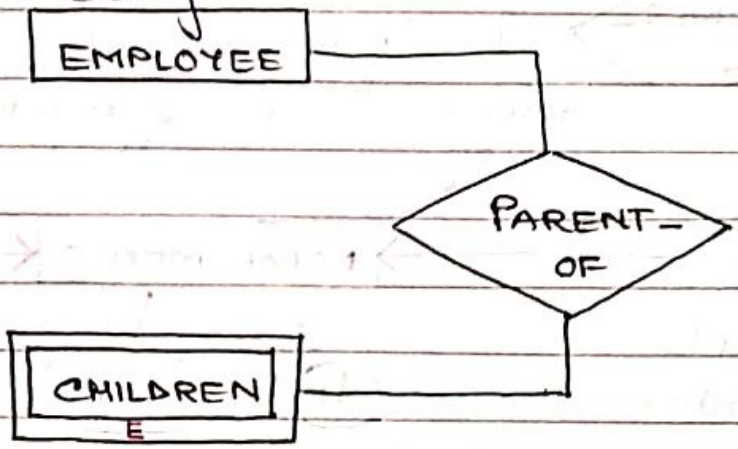
a) weak entity: A weak entity is an entity whose occurrences are dependant for their existence through a relationship R on the occurrence of another entity called strong entity. There may be occasional cases in which the entity occurrence of an entity are not distinguished by their attributes but rather by their relationships to entities of another type. These entities are called weak entities. Whereas, the entities whose entity occurrences can be distinguished by its keys are called strong entities. E.g:



In the example, a pilot is a specialization of personnel, or a personnel is a generalization of pilot. The pilot entity may not have a key, but can be identified by the personnel.identifier. IS_A is a weak relationship

b) existance dependant entity: The existance of an entity occurrence may sometimes depend on the existance of another entity occurrence in another entity.

e.g:



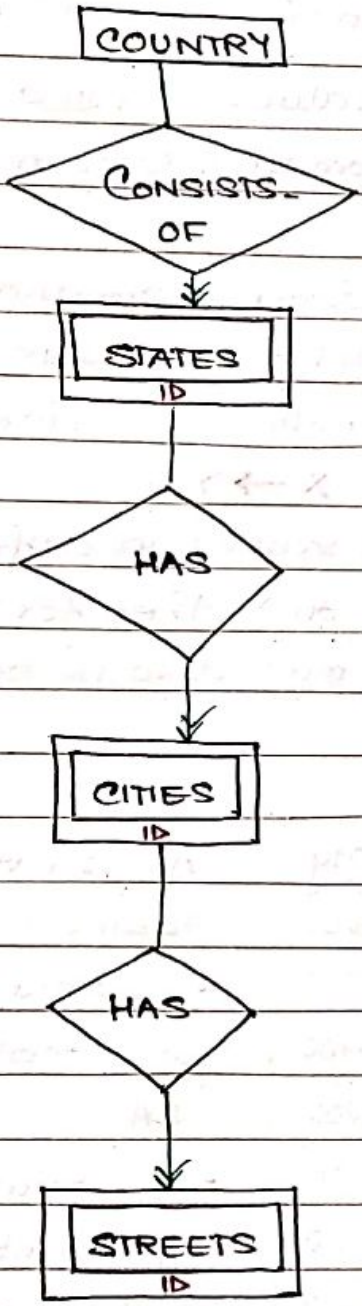
(existance dependant)

In this particular case, the existance of children entity depends on the existance of the associated employee. If an employee leaves the company, the database shall not keep track of the children. Thus "CHILDREN" is an existance dependant entity. This is also a weak entity, though the weak entity children may have a key attribute children#. The relationship PARENT_OF is also a weak-relationship (many-to-many). It is possible that the existance dependant relationship is a many-to-many mapping. e.g: If the father leaves the company, the children entity occurrence may still exist if their mother is an employee of the company.

c) ID dependancy: If an entity cannot be uniquely identified by its own attribute, and has to be identified by its relati-

onship with other entity(s), then we say that it has an ID dependancy on other entities.

E.g: A street is unique only within a city, a city is unique only within a state and a state is unique only within a country.



In order to uniquely identify the address of a location, we have to specify the names of city, state and country, in addition to the name of the street.

Remark: An ID dependancy is automatically existance constrained, but an existance constrained is not necessary an ID dependancy, since, the existance dependent entity can still be uniquely identified by its key.

Remark: Choice of attributes is a key step to database design.

Dependancies and normal form:

The goal in relational design is to choose relations that remain consistant and have minimum inconsistency. Such relations are said to be in the normal form. In a normalized relation, at every row and column position in the table, there exists precisely one value, and never a set of values, i.e., in a normalized relation, each of the underlined domains contain atomic values only.

Functional dependancy (FD): Given a relation R, the attribute Y of R is functionally dependant on attribute X of R, iff each X value in R has associated with it precisely one Y value in R. Is usually represented as: $X \rightarrow Y$

Note: There may be same X values in different tuples of R. If Y is functionally dependant on X, then for the tuples having same X values, the Y values must also be same.

Considering a relation:

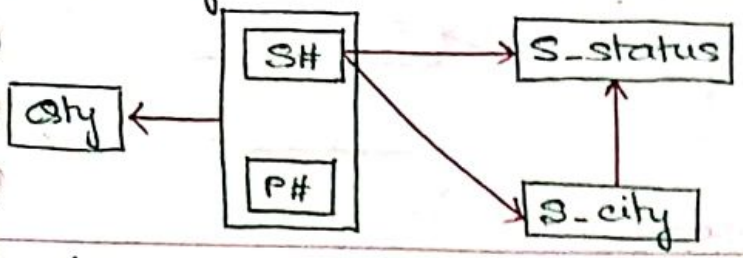
FIRST:

S#	S_status	s_city	P#	Qty
S1	20	London	P1	200
S1	20	"	P2	200
S1	20	"	P3	400
S1	20	"	P4	200
S1	20	"	P5	100
S1	20	"	P6	100
S2	10	Paris	P1	300
S2	10	"	P2	400
S3	10	"	P2	200
S4	20	London	P2	200
S4	20	"	P4	300
S4	20	"	P5	400

As is evident from the table,

- S_status and s_city are functionally dependant on S#
- S_status is functionally dependant on s_city.
- (S#, P#) form the key of the table.

FD diagram:



Difficulties in storage operations:

- INSERTION – If we were to insert a record for S5, who currently supplies no part; the operation would not be possible as P# cannot hold a null value.
- DELETION – If we were to delete record S3 – 10 – Paris – ···, rather, if we were to delete the transhipment of S3; all records corresponding to S3 would be deleted.
- UPDATION – If we were to update S_city of S1 from London to San Francisco; we would have to search through all the records and update accordingly. A very tedious task, prone to inconsistencies.

Cause of the above problems: Absence of full functional dependency (FFD)

Full functional dependancy (FFD): Attribute Y is FFD on attribute X, if it is FD on X and not FD on any proper subset of the attributes of X.

In the above FD diagram, a partial dependancy is revealed.

Resolution of the difficulties:

Projection of FIRST as follows, creating two new relations:

SECOND: (S# : key)

S#	S_status	S_city
S1	20	London
S2	10	Paris
S3	10	"
S4	20	London

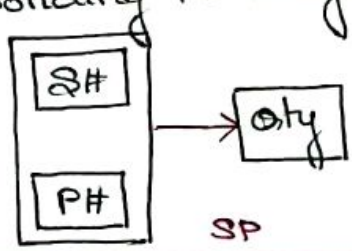
≈ supplier information

SP: ((S#, P#): key)

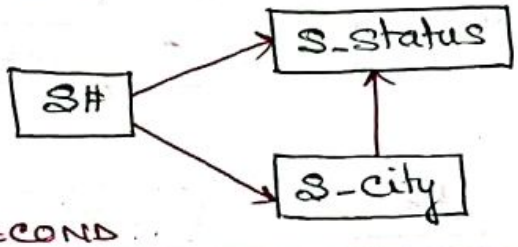
S#	P#	Qty
S1	P1	300
⋮	⋮	⋮
S2	P1	300
⋮	⋮	⋮

≈ relation between supplier and parts

Corresponding FD diagram:



SP



SECOND

Now, the previously mentioned storage operations are rendered easier.

However, some other difficulties are exposed:

- INSERTION: Rome : st_status = 50 cannot be inserted as S# for the record would have null values.
- DELETION: Say, there exists a record S5 – Rome - 50, if it be deleted, the record Rome - 50 stands deleted too.
- UPDATION: Due to the redundancy in S_city, updation of S_status is error-prone.

The above difficulties arise due to the transitivity in the dependancies in SECOND, i.e., the dependancy of supplier status on S#, though functional is transitive. This transitivity leads to difficulties over storage operations. Thus, we replace SECOND by its projections as follows:

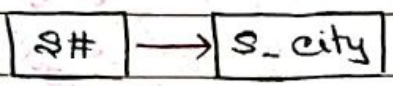
SC : (SC : key)

S#	S_city
S1	London
S2	Paris
S3	"
S4	London

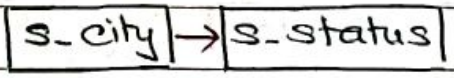
CS : (S_city : key)

S_city	S_status
London	20
Paris	10
Rome	50

FD diagram: (the FD of SP stands unchanged and is to be included)



SC



CS

**Transitive dependancy: A functional dependancy $X \rightarrow Y$ in a relation R is a transitive dependancy if there is a set of attributes Z, that is neither a candidate key nor a subset of any key of R, and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold.

First Normal Form (1NF): A relation is in 1NF, iff all the underlined domains contain atomic values only. Eg: FIRST

Second Normal Form (2NF): A relation is in 2NF, if it is in 1NF, and every non-key attribute is FFD on the primary key.

E.g: FIRST is not in 2NF, but SECOND and SP are in 2NF.

Third Normal Form (3NF): A relation is in 3NF, if it is in 2NF and every non-key attribute is non-transitively dependant on the primary key. E.g: SC, CS and SP are in 3NF.

▶.. Summary of normal forms, and the corresponding normalization technique:

NORMAL FORM	TEST	NORMALIZATION
• 1NF	Relation should have no nonatomic attributes, or nested relations.	Form new relations for each nonatomic attribute or nested relation.
• 2NF	For relations, where the primary key contains multiple attributes, no non-key attribute should be FD on part of the primary key.	Decompose and setup a new relation for each partial key with its dependant attribute(s). A relation must exist with the original primary key and all its FDs
• 3NF	There should be no transitive dependancy of a non-key attribute on the primary key.	Decompose and setup a relation that includes the non-key attribute(s) that FDetermine others.

Determinant: Any attribute(s) on which any one or more attributes are FFD is (are) called a determinant, i.e., if $X \xrightarrow[FFD]{} Y$, X is termed as the determinant.

In the relations covered, the determinants are as follows:

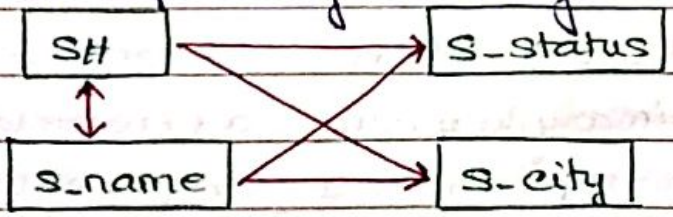
FIRST	SECOND	SP	SC	CS
S#	S#	(S#, P#)	S#	S_city
S_city	S_city			
(S#, P#)				

Boyce-Codd Normal Form (BCNF): A normalized relation R, can be said to be in BCNF if every determinant of R is a candidate key. If R is in BCNF, it must be in 3NF, but the reverse is not always true. Here, SP, SC and CS are in BCNF

E.g: Note: Any binary relation is in BCNF

- Given a relation: SUPP, with attributes: S#, S_name, S_status and S_city, where:
  - i) S_status and s_city are independent
  - ii) Both S# and s_name are candidate keys.

Thus, the corresponding FA diagram is:



Here there exists no transitivity, and the determinants are: S#, S_name and (S#, S_name) and they are all candidate keys. Thus, SUPP is in BCNF.

Here, the determinants are disjoint.

- Given a relation: SSP, with attributes: S#, S_name, P#, qty,

and the candidate keys are: (S#, P#), (P#, S_name). Now as is evident, the candidate keys are not disjoint, and the determinants are: (S#, P#), (S_name, P#), S#, S_name. Here all the determinants are not candidate keys, and thus SSP is not in BCNF.

Moreover, S_name is a redundant attribute, and updation of its values poses difficulties. Thus, resolution of the problem is through the projections:

SN (S# or S_name : key)

S#	S_name
⋮	⋮

PS ((S#, P#) : key)

S# (or S_name)	P#	Qty
⋮	⋮	⋮

- Given a relation: SJF, with attributes : Student, Subject, teacher. The rules pertaining to the relation are as follows:

i) For each subject, each student of that subject is taught by only one teacher.

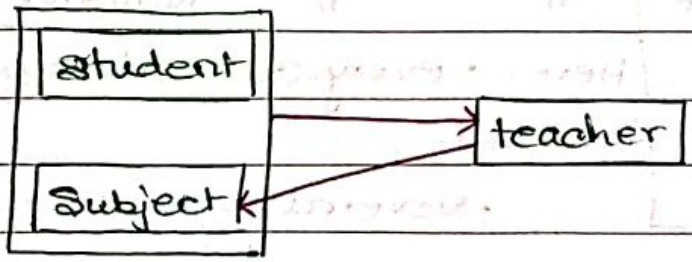
ii) Each teacher teaches only one subject.

iii) Each subject is taught by several teachers.

SJF

Student	subject	teacher
Anil	maths	Sen
"	physics	Saha
Madhu	maths	Sen
Madhu	physics	Dutta

Here, (student, subject) is a candidate key. (Student, teacher) forms an overlapping candidate too. Thus, an FD diagram would be:



Here, the determinants are: (student, subject), (student, teacher), teacher; but since teacher cannot be a candidate key, the relation is not in BCNF.

Here, if the tuple <anil - physics - saha> by deleted, then the record physics - saha is removed. Thus, STF is replaced by its projections: i) ST (student, teacher)

ii) TS (teacher, subject)

• Given a relation : EXAM, with attributes Student, subject, position; and i) No two students obtain the same position in the same subject.

Thus, the candidate keys are (student, subject) and (subject, position). Here, the determinants are (student, subject) and (subject, position). Now, since all the determinants are candidate keys, the given relation is in BCNF.

Though the determinants and the candidate keys overlap, the absence of a subset of the candidate keys as a determinant results in none of the previous difficulties arising.

• Given a relation :

CTX

Course	teacher	text
AI	KND	Luger
AI	"	Nillson
"	"	Winston.
DBMS	SC	Korth
"	"	Navathe
"	SKS	Korth
"	"	Navathe
"	NC	Korth
"	"	Navathe.

Here, • every course has a given text set.

• Several teachers can teach

a Course.

• Assume that no matter who actually teaches a particular Course, the same texts are used.

Meaning of CTX:

CTX is an all-key relation. A tuple $<c,t,x>$ appears in CTX, iff c can be taught by t who uses x as a reference. Note that in the relation CTX, if the tuples $<c, t_1, x_1>$ and $<c, t_2, x_2>$ both appear, then tuples $<c, t_1, x_2>$ and $<c, t_2, x_1>$ also appear. Clearly, CTX contains a great deal of redundancy. This leads to problems in update operations. E.g. To add the record that DBMS uses a new text Conoly, it is necessary to add one tuple for each teacher of DBMS, as CTX is an all-key relation and is in BCNF.

The lack of dependancy of teacher on text is the source of the difficulties.

Improvement:

Replacement of CTX by its all-key BCNF projections:

CT (Course, teacher),

CX (Course, text)

Extension of FD:

The definition of FD may be extended by: the following representations:

$X \xrightarrow[FD]{} Y \quad \approx \quad X \rightarrow$ (Y: y_1, y_2) — a well-defined set having multiple values of Y for a value of X.

In the second representation, the concept of multi-valued dependancy (MVD) is brought forth.

Remark: Although a given Course doesnot have a single corr-

esponding teacher (teacher is not FD on course), nevertheless, each course has a well-defined set of corresponding teachers. This is called an MVD.
There is an MVD of teacher on course and an MVD of text on course.

Clearly, FD is a special case of MVD, and hence, MVD is a generalization of FD, i.e, an FD is an MVD in which the set of dependant values, actually consists of a single value.
[FD is a specialization of MVD].

Theorem:
Lossless decomposition: A relation R with attributes (A, B, C) can be lossless decomposed into two projections: R_1 (A, B) and R_2 (A, C), iff, there is an MVD of B on A and an MVD of C on A in R.

Fourth Normal Form (4NF): A normalized relation R is said to be in 4NF, whenever there exists an MVD in R, say of attribute CTX is not in 4NF, but CT and CX are in 4NF. [B on attribute A, then all attributes of R are also FD on A]
Relational model and query language:
The relational data model, represents the database as a collection of tables, and there is a direct correspondence between the concept of a table and the concept of a mathematical relation.
Formal query language: A query language is a language through which a user requests information from the database.
These are typically high level languages compared to standard programing languages. Query languages are classified as

follows:

- Procedural: Specification of a sequence of operations. E.g: relational algebra.
- Nonprocedural: the sequence of operations is not specified, but the information desired is. E.g: relational calculus.

Relational Calculus is subdivided as follows:

- Tuple relational calculus: A variable represents a tuple of a relation.
- Domain relational calculus: A variable represents a data item or field or attribute (simple single-valued)

Commercial query language: Takes features of both procedural and non-procedural query languages. Eg: Structured Query Language (SQL)

- Relational Algebra:

Relational algebra, essentially encompasses:

i) 5 fundamental operations - that allow construction of queries of our choice. The fundamental operations are SELECT (σ), PROJECT (π), CARTESIAN PRODUCT ($\times$), UNION ($\cup$) and DIFFERENCE ($-$)

ii) 4 additional operations - that are dependant on the fundamental operations. The additional operations are INTERSECTION ($\cap$), THETA-JOIN. ($\bowtie_\theta$), NATURAL-JOIN ($\bowtie$) and DIVISION ($\div$)

Use of any of the above operations on relations, yields a new operation.

Fundamental operations:

• SELECT: A unary operation (i.e., it operates on a single relation) resulting in a horizontal subset of that relation.

Syntax: $\sigma_{\text{Predicate}}(\text{relation\_name})$

≈ Select rows of relation_name, satisfying conditions in the predicate. The predicate may comprise of relational operators (=, !=, <, <=, >, >=), logical operators/connectors (AND: ∧, OR: V, NOT: ¬) or a combination of both.

• PROJECT: A unary operation resulting in a vertical subset of the relation under consideration.

Syntax: $\pi_{attrib_1, attrib_2, \ldots attrib_k}(\text{relation\_name})$

≡ Select columns, corresponding to the attribute-list specified, of relation_name.

• CARTESIAN PRODUCT: A binary operation (i.e, it operates on two relations) resulting in the following:

R = relation1 X relation2 where,

if relation_1 has m attributes and t_1 tuples and relation_2 has n attributes and t_2 tuples,

R has (m+n) attributes (common attributes are repeated) and $(t_1 \times t_2)$ tuples.

• UNION: A binary operation with the following constraints:

i) The two relations, R1 and R2, under consideration must be compatible, i.e., should have the same number of attributes.

ii) The ith attributes of R1 and R2, must be defined in the same

domain, i.e., represent the same values, though they may be identified by different attribute names.
Syntax: R1 ∪ R2.
Results in a relation with tuples of R1 and R2, no repeatitions.

- DIFFERENCE: A binary relation resulting in tuples of R1 that are not present in R2. Follows constraints of compatibility and
Syntax: R1 − R2. (Set difference) Same domain values.

The 5 fundamental operations/operators allow us to give a complete definition of an expression in relational algebra. Let E_1 and E_2 be two relational algebra expressions. Then, each of the following are relational algebra expressions:

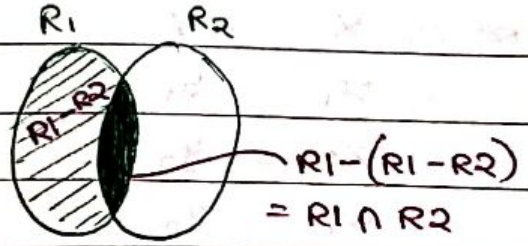
- $E_1 \cup E_2$
- $E_1 - E_2$
- $E_1 \times E_2$
- $\sigma_p(E_1)$, where p = predicate on attribute(s) of E_1.
- $\pi_s(E_1)$, where s = a list of some of the attributes of E_1.

Additional operations:

The five fundamental operations of relational algebra, σ, π, ∪, −, × are sufficient to express any relational algebra query. However, some common queries are lengthy to express with just these 5 operations. The four additional operations, each of which can be expressed in terms of the five fundamental operations, make the query length smaller in some cases.

- INTERSECTION: Syntax: $R_1 \cap R_2$, results in a relation with tuples common to both R1 and R2.

$R1 \cap R2 = R1 - (R1 - R2)$



Remark: It is possible to write several equivalent relational algebra expressions, that are quite different from one another.

- THETA JOIN: A binary operation with syntax: $R1 \bowtie_\theta R2$

$$\equiv \sigma_\theta (R1 \times R2)$$

- NATURAL JOIN: A binary operation with syntax: $R1 \bowtie R2$

$$\equiv \pi_{R1' \cup R2'} \left(\sigma_{R1.A_1 = R2.A_1 \wedge R1.A_2 = R2.A_2 \wedge \ldots \wedge R1.A_k = R2.A_k} (R1 \times R_2)\right)$$

where A_i ($i = 1$ to k) represents the common attribute set of the relations. The common attributes occur only once in the resultant relation. ($R1'$, $R2'$ => attribute sets of R1 and R2 respectively)

- DIVISION: A binary operation with syntax: $R1 \div R2$

$$\equiv \pi_{R1'-R2'}(R1) - \pi_{R1'-R2'}((\pi_{R1'-R2'}(R1) \times R2) - R1)$$

($R1'$, $R2'$ = attribute sets of R1 and R2 respectively). The constraints of the operation are: If R1 has $(m+n)$ attributes and R2 has the last n attributes, i.e, the $(m+i)$th attribute of R1 coincides with the ith attribute of R2, then the division operation yields a relation ($R = R1 \div R2$) that has m attributes. Usually satisfies the "for-all" clause in a query.

[Proof of the equivalence:

Clubbing/Concatenating together the 'm' attributes to a single attribute 'X' and the 'n' attributes to 'Y' for R1 and R2, Let,

R1: X	Y	R2: Y	then R1 ÷ R2
x_1	y_1	y_1	$= x_1$
x_2	y_2	y_2	($\because x_1$ is associated with
x_1	y_2		each value of Y)

[A tuple t is in $r \div s$ iff: {r(R), s(S) are relations, $S \subseteq R$; $r \div s$ is a relation on R-S}
a) t is in $\pi_{R-S}(r)$ b) For every t_s in s, there is t_r in r: $t_r[S] = t_s[S]$, $t_r[R-S] = t$]

Now, if $\pi_{R1'-R2'}(R1)$ by named A, i.e.,

$A = \pi_{R1'-R2'}(R1)$, then $A \times R_2$ $\Rightarrow (A \times R_2) - R_1$

X
x_1
x_2

X	Y
x_1	y_1
x_2	y_2
x_1	y_2
x_2	y_1

X	Y
x_2	y_1

$\therefore B = \pi_{R1'-R2'}((A \times R_2) - R1)$ $\therefore A - B =$

X
x_1

X
x_2

$= R1 \div R2.$

Relational algebra queries:

Given a schema: "ABC"

EMP (FNAME, MINIT, LNAME, SSN, BDATE, ADDR, SEX, SALARY, SUPERSSN, DNO) — FOREIGN KEY.

DEPT (DNAME, DNUMBER, MGRSSN, MGRSTARTDATE)

DEPT_LOC (DNUMBER, DLOCATION)

WORKS_ON (ESSN, PNO, HOURS)

PROJECT (PNAME, PNUMBER, PLOC, DNUM)

DEPENDENT (ESSN, DNAME, SEX, BDATE, RELATION)

- Retrieve the name and address of all employees who work in the "RESEARCH" department.

$RSCH\_DEPT \leftarrow \sigma_{DNAME = 'RESEARCH'}(DEPT)$

RSCH_DEPT_EMP ← RSCH_DEPT $\bowtie_{DNUMBER=DNO}$ EMP

RESULT ← $\pi_{FNAME, LNAME, ADDRESS}$ (RSCH_DEPT_EMP)

- For every project located in Stafford, list the project number, the controlling department number and the last name, address and birth date of the department manager.

PJCT_STAFFORD ← $\sigma_{PLOC='STAFFORD'}$ (PROJECT)

CNTRL_DEPT ← PJCT_STAFFORD $\bowtie_{DNUM=DNUMBER}$ DEPT

RESULT ← $\pi_{LNAME, ADDR, BDATE}$ (CNTRL_DEPT $\bowtie_{MGRSSN=SSN}$ EMP)

- Find the names of employees who work on all the projects controlled by department numbered '5'.

PJCT5 (PNO) ← $\pi_{PNUMBER}$ ($\sigma_{DNUM=5}$ (PROJECT))

SSNS (SSN, PNO) ← $\pi_{ESSN, PNO}$ (WORKS_ON)

SSNS_PJCT5 ← SSNS ÷ PJCT5

RESULT ← $\pi_{FNAME, LNAME}$ (SSNS_PJCT5 ⋈ EMP)

- Retrieve the names of employees who have no dependants.

SSN_DEPN (SSN) ← π_{ESSN} (DEPENDENT)

SSNS ← π_{SSN} (EMP)

$SSN\_NDEPN \leftarrow SSNS - SSN\_DEPN$

$RESULT \leftarrow \pi_{FNAME, LNAME} (SSN\_NDEPN \bowtie EMP)$

- List the names of managers who have atleast one dependant.

$MANAGER(SSN) \leftarrow \pi_{MGRSSN} (DEPT)$

$SSN\_DEPN(SSN) \leftarrow \pi_{ESSN} (DEPENDENT)$

$MANAGER\_DEPN \leftarrow MANAGER \cap SSN\_DEPN$

$RESULT \leftarrow \pi_{FNAME, LNAME} (MANAGER\_DEPN \bowtie EMP)$

- Make a list of project numbers or projects that involve an employee whose last name is 'SMITH' - either as a worker or as a manager of the department that controls the project.

$SSN\_SMITH(ESSN) \leftarrow \pi_{SSN} (\sigma_{LNAME='SMITH'} (EMP))$

$SMITH\_WORKER \leftarrow \pi_{PNO} (SSN\_SMITH \bowtie WORKS\_ON)$

$MGR\_SMITH(DNUM) \leftarrow \pi_{DNUMBER} (SSN\_SMITH \bowtie_{MGRSSN = ESSN} DEPT)$

$SMITH\_MANAGER(PNO) \leftarrow \pi_{PNUMBER} (MGR\_SMITH \bowtie PROJECT)$

$RESULT \leftarrow SMITH\_WORKER \cup SMITH\_MANAGER.$

- Relational Calculus:

Relational algebra is a procedural language, since we write a relational algebra expression, by providing a sequence of operations that generates the answers to our query. In relational calculus we give a formal description of the information desired, without specifying how to obtain the information. Relational calculus

is non-procedural. The two forms of relational calculus are:

Tuple relational calculus – A variable represents a tuple in a relation.

Domain relational calculus – A variable represents a column/field in a relation.

Tuple Relational Calculus (TRC):

A general query of TRC is of the form $\{t \mid P(t)\}$, i.e., we would like to obtain all tuples "t", satisfying the predicate "P(t)".

An attribute of a typical tuple t, is stated as t.A, "A" being an attribute in 't'.

Formal definition of TRC:

A tuple variable is a free variable, unless quantified by $\exists$ (there exists) or $\forall$ (for all), else they are bound variables. Any number of tuple variables may appear in a formula. A TRC formula is built up of atoms. An atom has the following forms:-

a) $s \in r$, where s is a tuple variable and r is a relation (the use of the $\notin$ operator is not permitted)

b) $s.x \odot u.y$, where s and u are tuple variables, x is an attribute on which s is defined, y is an attribute on which u is defined, and $\odot$ is a comparison operator ($<, <=, =, \neq (!=), >, >=$); it is required that attributes x and y have domains whose members can be compared by $\odot$.

c) $s.x \odot c$, where s is a tuple variable, x is an attribute on which s is defined, $\odot$ is a comparison operator, and c is a constant in the domain of attribute x.

Formulae are built up of atoms, using the following rules:

a) An atom is a formula.

b) If P_1 is a formula, then so are: $\neg P_1$ and (P_1)

c) If P_1 and P_2 are formulae, then so are: $P_1 \vee P_2$, $P_1 \wedge P_2$.

d) If $P_1(s)$ is a formula containing a free tuple variable s, and r is a

relation, then $\exists s \in r(P_1(s))$, $\forall s \in r(P_1(s))$ are also formulae.

Safe TRC:

A TRC expression may generate an infinite relation. e.g: $\{t \mid \neg R(t)\}$ may not be a finite set, and may contain values that donot belong to the database. Thus, such TRC expressions are unsafe.

To define a restriction on the TRC expressions, we introduce the concept of domain of a tuple relational formula P. The domain of P [dom(P)] is the set of all values that appear explicitly in P or that appear in one or more relations whose names appear in P. An expression $\{t \mid P(t)\}$ is safe, if all the following holds :-

a) All values that appear in tuples of the expressions, are values from dom(P).

b) For every expression like $\exists s(P_1(s))$ in P, the subformula is true, iff there is a tuple S, with values from $dom(P_1)$, such that $P_1(s)$ is true.

c) For every expression like $\forall s(P_1(s))$ in P, the subformula is true, iff $P_1(s)$ is true for every tuple S, with values from $dom(P_1)$.

Safe TRC expressions are equivalent to relational algebra expressions.

Queries:

Using schema: "ABC"

- Retrieve the birth date and address of employees named 'JOHN B SMITH'.

$\{t.BDATE, t.ADDRESS \mid EMP(t)$ AND $t.FNAME = 'JOHN'$ AND $t.MINIT = 'B'$ AND $t.LNAME = 'SMITH'\}$

- Retrieve the name and address of all employees who work for the 'RESEARCH' department.

$\{t.FNAME, t.LNAME, t.ADDRESS \mid EMP(t)$ AND $(\exists d)($ DEPT(d) AND $d.DNAME = 'RESEARCH'$ AND $d.DNUMBER = t.DNO)\}$

- For every project located in Stafford, list the project number, the controlling department number and the department manager's last name, birthdate and address.

$\{P.PNUMBER, P.DNUM, t.LNAME, t.BDATE, t.ADDRESS \mid$
PROJECT(P) AND EMPLOYEE(t) AND $P.PLOC = 'STAFFORD'$ AND
$(\exists d)$ (DEPT(d) AND $d.DNUMBER = P.DNUM$ AND $d.MGRSSN = t.SSN)\}$

Domain Relational Calculus (DRC):

In this form, we use domain variables that take on values from an attribute domain, rather than values from an entire tuple. An expression in DRC is of the form:

$\{\langle x_1, x_2, \ldots, x_n \rangle \mid P(x_1, x_2, \ldots, x_n)\}$, where x_i $(1 \leq i \leq n)$ represent domain variables and P represents a formula composed of atoms. An atom in DRC has one of the following forms:

a) $\langle x_1\ x_2 \ldots x_n \rangle \in r$, where r is a relation on n attributes and x_i are domain variables or domain constants.

b) $x \odot y$, where x and y are domain variables, and $\odot$ is a comparison operator, and the attributes x and y have domains that can be compared by $\odot$

c) $x \odot c$, where x is a domain variable, $\odot$ is a comparison operator and c is a constant in the domain of the attribute for which x is a domain variable.

The DRC formulae are built up of atoms, using the following rules:

a) An atom is a formula.
b) If P_1 is a formula, then so are $\neg P_1$, (P_1)
c) If P_1 and P_2 are formulae, then so are $P_1 \vee P_2$, $P_1 \wedge P_2$.

d) If $P_1(x)$ is a formula in x, where x is a free domain variable, then $\exists x\,(P_1(x))$ and $\forall x\,(P_1(x))$ are also formulae.

Notational shorthand:

$\exists a, b, c\,(P(a,b,c)) \approx \exists a\,(\exists b\,(\exists c\,(P(a,b,c))))$.

Safe DRC:

DRC expressions like $\{\langle x_1, x_2, \ldots, x_n\rangle \mid \neg P(x_1, x_2, \ldots, x_n)\}$ are unsafe, as they allow values in the result that are not in the domain of the expression. Thus, a DRC expression $\{\langle x_1, x_2, \ldots, x_n\rangle \mid P(x_1, x_2$ is safe, if all of the following hold:

a) All values that appear in tuples of the expression, are values from dom (P)

b) For every expression of the form $\exists x\,(P_1(x))$, the subformula is true iff, there is a value x in dom (P_1) such that $P_1(x)$ is true.

c) For every expression of the form $\forall x\,(P_1(x))$, the subformula is true iff, $P_1(x)$ is true for all values of x from dom (P_1).

Queries: (using schema 'ABC')

- Retrieve the birthdate and address of employees named 'JOHN B SMITH'

$\{uv \mid (\exists q)(\exists r)(\exists s)(\text{EMP}(qrstuvwxyz) \text{ AND } q = \text{'JOHN' AND } r = \text{'B' AND } s = \text{'SMITH'})\}$

- Retrieve the name and address of all employees who work for the 'RESEARCH' department

$\{qsv \mid (\exists z)(\exists l)(\exists m)(\text{EMP}(qrstuvwxyz) \text{ AND DEPT}(lmno) \text{ AND } l = \text{'RESEARCH' AND } m = z)\}$

- For every project located in Stafford, list the project number, the controlling department number and the department manager's last name, birth date and address.

$\{iksuv \mid (\exists j)(\exists m)(\exists n)(\exists t)$ (PROJECT (hijk) AND DEPT (lmno) AND EMPLOYEE (qrstuvwxyz) AND j = 'STAFFORD' AND $k = m$ AND $n = t$)$\}$

• Structured Query Language (SQL):

The general format of a data retrieval SQL is :

```
SELECT   <attribute_list>
FROM     <relation_list>
WHERE    <predicate>
```

The output of the above is again a relation having an attribute list and corresponding values.

Queries: (using the schema 'ABC')

• Retrieve the birthdate and address of the employee, whose name is 'JOHN B SMITH'.

```
SELECT  BDATE, ADDR
FROM    EMP
WHERE   FNAME = 'JOHN' AND MINIT = 'B' AND LNAME = 'SMITH'.
```

• Retrieve the name and address of all employees working in the 'RESEARCH' department.

```
SELECT  FNAME, LNAME, ADDR
FROM    EMP, DEPT
WHERE   DNAME = 'RESEARCH' AND
        DNO = DNUMBER.
```

• For every project located in Stafford, list the project number, controlling department number and the department manager's last name, b-date and address.

```
SELECT  PNUMBER, DNUM, LNAME, BDATE, ADDR
FROM    PROJECT, DEPT, EMP
```

```
WHERE   PROJECT.PLOC = 'STAFFORD'  AND
        PROJECT.DNUM = DEPT.DNUMBER  AND
        DEPT.MGR.SSN = EMP.SSN
```

- Retrieve all employee SSN

```
SELECT  SSN
FROM    EMP
```

- Retrieve all combinations of employee SSN and department name.

```
SELECT  SSN, DNAME
FROM    EMP, DEPT
```

(Results received from cartesian product of EMP and DEPT)

- Retrieve whole records of employees working in department 5

```
SELECT  *  ← all records
FROM    EMP
WHERE   DNO = '5'
```

- Retrieve the salary of every employee

```
SELECT  DISTINCT SALARY
FROM    EMP
```

- For each employee, retrieve the FNAME, LNAME and the same for his/her supervisor

```
SELECT  E.FNAME, E.LNAME, S.FNAME, S.LNAME
FROM    EMP E, EMP S        (⇒ E x S)
WHERE   S.SSN = E.SUPERSSN
```

• Make a list of project numbers for projects that involve an employee whose LNAME = 'SMITH', either as a worker or as a manager of the department controlling the project.

```
(SELECT PNO
 FROM WORKS_ON, EMP, PROJECT
 WHERE ESSN = SSN AND PNUMBER = PNO AND
       LNAME = 'SMITH')
UNION
(SELECT PNUMBER
 FROM PROJECT, EMP, DEPT
 WHERE DNUM = DNUMBER AND
       MGR_SSN = SSN AND
       LNAME = 'SMITH')
```

OR,

```
SELECT DISTINCT PNUMBER
FROM PROJECT
WHERE PNUMBER IN (SELECT PNO
                  FROM WORKS_ON, PROJECT, EMP
                  WHERE ESSN = SSN AND
                        PNUMBER = PNO AND
                        LNAME = 'SMITH')
               OR (SELECT PNUMBER
                   FROM PROJECT, DEPT, EMP
                   WHERE DNUM = DNUMBER AND
                         MGR_SSN = SSN AND
                         LNAME = 'SMITH').
```

** The IN operator can also compare tuples of values in parantheses.

• Retrieve SSN of all employees who work the same (hours, project) combination, as some project employee with SSN = '123456789'.

```
SELECT DISTINCT ESSN
FROM WORKS_ON
WHERE (HOURS, PNO) IN ( SELECT HOURS, PNO
                        FROM   WORKS_ON
                        WHERE  ESSN = '123456789' )
```

• Retrieve the names of all employees whose salary is greater than the salary of all employees in department: '5'.

```
SELECT FNAME, LNAME
FROM   EMP
WHERE  SALARY >all (SELECT DISTINCT SALARY
                    FROM   EMP
                    WHERE  DNO = '5').
```

** If X be a member of a set, while Y be a set, the expression "$X >_{all} Y$" returns true, if the value of X is greater than that of all the members in Y.

• Retrieve the name of each employee, who has a dependant of the same name and same sex as the employee.

```
SELECT FNAME, LNAME
FROM   EMP
WHERE  SSN IN (SELECT ESSN
               FROM   DEPENDANT D
               WHERE D.ESSN = SSN AND
                     D.DNAME = FNAME AND
                     D.SEX = SEX)
```

OR,
```
SELECT E.FNAME, E.LNAME
FROM   EMP E, DEPENDANT D
WHERE  D.ESSN = E.SSN AND
       D.DNAME = E.FNAME AND
       D.SEX = E.SEX.
```

** In general, a query written in nested blocks and using the "=" or "IN" operator, can always be expressed as a single block query.

The constructs with "IN", "$>_{any}$", "$>_{all}$", allow us to test a single value against members of a set (an entire set). SQL also allows "$<_{any}$", "<=any", ">=any", "=any", "!=any", "!=all".

Since a SELECT generates a set of tuples, we may at times want to compare sets to determine if one set contains all the members of some other set. Such comparisons are made in SQL using CONTAINS and NOT CONTAINS.

X CONTAINS Y => $Y \subseteq X$

X NOT CONTAINS Y => $Y \not\subseteq X$.

• Retrieve the names of all employees who work on all the projects controlled by department "5".

```
SELECT FNAME, LNAME
FROM   EMP
WHERE ((SELECT PNO
        FROM WORKS_ON
        WHERE SSN = ESSN)
        CONTAINS
       (SELECT PNUMBER
        FROM PROJECT
        WHERE DNUM = '5'))
```

** EXISTS (Q) returns true, if atleast one tuple exists in the result of the tuple Q. Similarly, we have NOT EXISTS (Q) as opposed to EXISTS (Q), which returns true if Q is empty.

• An alternative approach to the query pertaining to the retrieval of names of employees who have dependants of the same name and sex. Here, for each tuple of employee, the nested query that retrieves all dependant tuples with same SSN, name and sex as the employee tuple, is evaluated.

```
SELECT FNAME, LNAME
FROM   EMP E
WHERE  EXISTS (SELECT *
               FROM DEPENDANT
               WHERE E.SSN = ESSN AND
                     E.FNAME = DNAME AND
                     E.SEX = SEX).
```

If atleast one tuple exists in the result of the nested query, then select that employee tuple for output.

• Retrieve the names of employees who have no dependants.

```
SELECT FNAME, LNAME
FROM   EMP
WHERE  NOT EXISTS (SELECT *
                   FROM DEPENDANT
                   WHERE ESSN = SSN)
```

• List the names of managers who have atleast one dependant.

```
SELECT FNAME, LNAME
FROM   EMP E
WHERE  EXISTS (SELECT *
```

```
                FROM DEPENDANT
                WHERE ESSN = SSN)
AND EXISTS      (SELECT *
                FROM DEPT
                WHERE MGR_SSN = SSN)
```

** X CONTAINS Y ⇒ $Y \subseteq X$

⇒ Y EXCEPT X ⇒ $Y - X = \phi$

i.e., "Set X contains set Y" is logically equivalent to "Y except (set difference) X" is empty.

- An alternative approach to the query pertaining to the retrieval of the names of all employees working in all projects controlled by department '5'.

```
SELECT FNAME, LNAME
FROM   EMP
WHERE  NOT EXISTS ((SELECT PNUMBER
                    FROM   PROJECT
                    WHERE  DNOM = '5')
                    EXCEPT
                   (SELECT PNO
                    FROM   WORKS_ON
                    WHERE  SSN = ESSN)).
```

- Retrieve the SSNs of all employees who work on project: 1, 2 or 3.

```
SELECT ESSN
FROM   WORKS_ON
WHERE  PNO IN (1, 2, 3)
```

• Retrieve the names of all employees who donot have any supervisors.

```
SELECT  FNAME, LNAME
FROM    EMP
WHERE   SUPER_SSN IS NULL
```

← missing / undefined / unapplicable.

• Retrieve the last name of each employee and his/her supervisor, while renaming the attribute names as "EMP_NAME" and "SUPERVISOR_NAME" respectively.

```
SELECT  E.LNAME AS EMP_NAME,
        S.LNAME AS SUPERVISOR_NAME
FROM    EMP AS E , EMP AS S.
WHERE   E.SUPERSSN = S.SSN
```

• Retrieve the name and address of each employee working in the research department.

```
SELECT  FNAME, LNAME, ADDR
FROM    (EMP JOIN DEPT ON DNO = DNUMBER)
WHERE   DNAME = 'RESEARCH'
```

← θ join operation.

** Aggregate functions and grouping: There are a number of built in aggregate functions like COUNT, SUM, MAX, MIN, AVG etc.

• Find the sum of salaries of all employees, the maximum salary, the minimum salary and the average salary.

```
SELECT  SUM (SALARY), MAX (SALARY), MIN (SALARY),
        AVG (SALARY)
FROM    EMP.
```

• Find the sum of the salaries of the employees in the research department, as well as the max, min and avg salary in this department.

```
SELECT SUM (SALARY), MAX (SALARY), MIN (SALARY), AVG (SALARY)
FROM   EMP, DEPT
WHERE  DNO = DNUMBER AND
       DNAME = 'RESEARCH'.
```

• Find the total number of employees in the company.

```
SELECT COUNT (*)
FROM   EMP.
```

• Find the number of employees in the research department.

```
SELECT COUNT (*)
FROM   EMP, DEPT
WHERE  DNO = DNUMBER AND
       DNAME = 'RESEARCH'.
```

• Count the number of distinct salary values.

```
SELECT COUNT (DISTINCT SALARY)
FROM   EMP.
```

• Retrieve the names of employees, who have 2 or more dependants.

```
SELECT FNAME, LNAME
FROM   EMP
WHERE  (SELECT COUNT (*)
        FROM   DEPENDANT
        WHERE  ESSN = SSN) >= 2.
```

- For each department retrieve department number, number of employees in that department and their average salary.

```
SELECT DNO, COUNT (*), AVG (SALARY)
FROM   EMP
GROUP BY DNO.
```

- For each project, retrieve the project number, project name and the number of employees working on the project.

```
SELECT PNO, PNAME, COUNT (*)
FROM   PROJECT, WORKS_ON
WHERE  PNO = PNUMBER
GROUP BY PNO
```

← HAD PNAME BEEN A KEY, GROUP BY PNAME WOULD BE POSSIBLE.

(OR GROUP BY (PNO, PNAME))

- For each project, on which more than 2 employees work, retrieve the project number, project name and number of employees who work on the project.

```
SELECT PNO, PNAME, COUNT (*)
FROM   PROJECT, WORKS_ON
WHERE  PNUMBER = PNO
GROUP BY PNUMBER, PNAME
HAVING COUNT (*) > 2
```

← usually used where COUNT(*) has already been computed.

- For each project, retrieve the project number, project name and number of employees of department '5' who work on the project.

```
SELECT PNO, PNAME, COUNT (*)
FROM   PROJECT, WORKS_ON, EMP
WHERE  PNUMBER = PNO AND
       ESSN = SSN AND DNO = '5'
GROUP BY PNUMBER,
         PNAME.
```

• For each department having more than 5 employees, retrieve the department name, and the number of employees getting more than Rs. 40,000.

```
SELECT DNAME, COUNT(*)
FROM   DEPT, EMP
WHERE  DNUMBER = DNO AND
       SALARY > 40000 AND
       DNO IN (SELECT DNO
               FROM   EMP
               GROUP BY DNO
               HAVING COUNT (*) > 5).
GROUP BY DNAME.
```

• Retrieve a list of employees and the projects they are working on, ordered by department, and within each department alphabetically by the lastname and the firstname.

```
SELECT DNAME, LNAME, FNAME, PNAME
FROM   DEPT, WORKS_ON, PROJECT, EMP.
WHERE  PNUMBER = PNO AND
       ESSN = SSN AND
       DNO = DNUMBER.
ORDER BY DNAME, LNAME ASC, FNAME.
```

↖ ordered in the ascending order by default.

DESC ⇒ descending order.

Database Manipulation:

Insert ; Delete ; Update.

- INSERT INTO EMP
  VALUES ('RICHARD', 'K', 'MARINI'...)

- INSERT INTO EMP (FNAME, LNAME, SSN) ← other fields hold default/null values.
  VALUES ('RICHARD', 'MARINI', '235892851G')

- DELETE DEPENDANT ← the structure remains, while all constituent data are removed. Insertion hereafter into the table is possible.

- DELETE FROM EMP
  WHERE LNAME = 'DEY'

- DELETE FROM EMP
  WHERE DNO IN (SELECT DNUMBER
  FROM DEPT
  WHERE DNAME = 'RESEARCH')

- Delete all employees whose salary is less than the average salary.

  DELETE FROM EMP ← marks all records to be deleted, during the course of the execution
  WHERE SALARY < (SELECT AVG (SALARY)
  FROM EMP)

of the query; and the marked records are deleted physically at the end of the query execution. It may so seem that AVG (SALARY) is reevaluated after each record is marked, but practically the value is evaluated only once.

An UPDATE clause is used to change the value of an attribute in a tuple. The WHERE clause is used to select the tuple to be updated (if this clause is not present => all tuples), and a SET clause is used to no-

minate the attribute whose value is to be changed and to specify the new value.

- UPDATE PROJECT
  SET PLOCATION = 'MUMBAI', DNUM = 5
  WHERE PNUMBER = 10.

- Increase the salary by 10% of all employees in the research department.

```
UPDATE EMP
SET SALARY = SALARY + (1.1 * SALARY)
WHERE DNO IN (SELECT DNUMBER
              FROM DEPT
              WHERE DNAME = 'RESEARCH')
```

Views in SQL:

A view is a single table that is derived from other tables. It does not necessarily exist in its physical form. It is considered as a virtual table, in contrast to the base tables, whose tuples are actually stored in the database. We can think of a view as a way of specifying a table that we need to reference frequently, even though it may not exist physically.

Eg: We may frequently use queries that retrieve the employee name and the project names that the employee works on; rather than having to specify the join of 'EMP', 'WORKS_ON' and 'PROJECT' tables each time we issue the query, we can define a view that is a result of these joins. We can then issue queries on the view, which are specified as "Single table retrievals", rather than as retrievals involving 2 joins on three tables.

A view is defined in SQL, using the CREATE VIEW command:

```
CREATE VIEW V AS <QUERY>
```

E.g: CREATE VIEW WORKS_ON1 AS
SELECT FNAME, LNAME, PNAME, HOURS
FROM EMP, WORKS_ON, PROJECT
WHERE SSN = ESSN AND
PNO = PNUMBER.

← Same attribute names as the tables are used in the view

- Here, we need not specify any new attribute names. It inherits the names of the view attributes from the defining base tables : EMP, WORKS_ON, PROJECT.

- CREATE VIEW DEPT_INFO (DEPT_NAME, NO_OF_EMP, TOT_SAL)
AS SELECT DNAME, COUNT (*), SUM (SALARY)
FROM EMP, DEPT
WHERE DNO = DNUMBER
GROUP BY DNAME.

- Retrieve the first name and last name of employees who work on PNAME = 'PROJECT X'.

```
SELECT FNAME, LNAME
FROM WORKS_ON1
WHERE PNAME = 'PROJECT X'.
```

** A view is always uptodate, i.e., if we modify the tuples in the base tables, on which the view is defined, the view automatically reflects these changes. This is done by the underlined DBMS.

If we donot need a view anymore, we can use the DROP VIEW command to dispose it off. E.g: DROP VIEW DEPT_INFO.

**Updating views:

Considering the WORKS_ON1 view, and suppose we issue the command to update attribute PNAME of JOHN SMITH from PROJECT X to PROJECT Y.

```
UPDATE WORKS_ON1
SET PNAME = 'PROJECT Y'
WHERE FNAME = 'JOHN' AND
      LNAME = 'SMITH' AND
      PNAME = 'PROJECT X';
```

- Possible reflections of change on base:

This query can be mapped into several updates in the base relations to give the desired update on the view.

a)
```
UPDATE WORKS_ON
SET PNO = (SELECT PNUMBER
           FROM PROJECT
           WHERE PNAME = 'PROJECT Y')
WHERE ESSN = (SELECT SSN
              FROM EMP
              WHERE FNAME = 'JOHN' AND
                    LNAME = 'SMITH')
AND PNO = (SELECT PNUMBER
           FROM PROJECT
           WHERE PNAME = 'PROJECT X')
```

b)
```
UPDATE PROJECT
SET PNAME = 'PROJECTY'
WHERE PNAME = 'PROJECT X'
```

In cases where instances pertaining to PROJECT Y do not exist.

** In general, we cannot guarantee that any view can be updated. Whenever, an update on the view can be mapped to more than one update on the underlying base, we must have a certain procedure to choose the desired update. We can make the following obser-

vations:

a) A view with a single defining table is updateable, if the view attributes contain the primary key, or some other candidate key of the base relation, as this maps each view tuple to a single base tuple.

b) Views defined on multiple tables using joins, are generally not updateable.

c) Views defined using grouping and aggregate functions are not updateable.

As a result of these observations, most database systems impose the following constraints:

A modification is permitted through views, only if the view in question is defined on terms of one relation of the actual relational database.

Transaction Processing:

We consider an interleaved model of concurrent execution. The execution of a program that accesses or changes the content of the database is called a transaction. In our case, it refers to a program execution that updates the database unless we explicitly state otherwise.

Basic database operations:

READ (x, X) : x : program variable X : database item. : $x \leftarrow X$.

$\approx$ READ (X) : $x \leftarrow X$.

WRITE (x) : $X \rightarrow X$

program variable database item.

Primitive operations of READ (X):-

- locate data items in disk block.
- read block into main memory buffer, if not present already.
- locate data item in buffer and assign to program variables.

Primitive operations of WRITE(X)

- locate data item in disk block.
- read block into main memory buffer, if not present already.
- modify data items in buffer (from program variables) and write buffer

content into disk.

** The recovery manager is a subsystem of the DBMS package. It decides if the buffer content is to be transferred to the disk immediately or later.

• We have a variable (x) holding the number of bookings made on airline A_1, while (Y) holds the same for airline A_2. Transaction (T_1) is concerned with discarding n seats of A_1 and booking those seats onto A_2; while T_2 is concerned with booking m seats of A_1.

T_1	T_2	
READ (x)	READ (x)	
X = X - n	X = X + m	consistency check: X+Y = constant
WRITE (x)	WRITE (x)	
READ (Y)		• We are unaware of the order of transactions Considering arbitrary.
Y = Y + n		
WRITE (Y)		

Let, initial values be as follows: X = 80 ; n = 5 ; m = 4

A) T_1 and T_2 serially executed ⇒ X = 79

B) Considering a schedule: S_a

	READ (x)
x = 75 ↗	X = X - n
	READ (x)
	X = X + M ← x = 84
x = 75 ↗ written	WRITE (x)
	READ (Y)
	WRITE (x) ← x = 84
	Y = Y + n
	WRITE (Y)

overwritten (i.e., last update lost)

'LOST UPDATE PROBLEM'

The lost update problem occurs when two transactions, that access the same database items, have their operations interleaved in a way,

that makes the value of some database item incorrect.

c) Considering a schedule: S_b

READ (X)	
X = X − n	
WRITE (X)	
	READ (X)
	X = X + M
	WRITE (X)
READ (Y)	
failure (T_1)	

x = 75 → WRITE (X)

READ (X) ← value read is dirty since T_1 is not yet complete, ⇒ scope of change to X.

WRITE (X) ← x = 79

Y's value remains unchanged. T_1 not complete, might need to undo all operations.

"DIRTY READ PROBLEM"

The dirty read problem occurs when one transaction updates a database item and the transaction fails for some reason in future. In between, the updated item is accessed by another transaction, before it is changed to its original / final value.

d) Considering a schedule that primarily aims at summarizing the total transactions of the day: S_c

"INCORRECT SUMMARY PROBLEM"

Sum = 0	
READ (A)	
Sum += A	
READ (B)	
Sum += B	
⋮	
	READ (X)
	X = X − n
	WRITE (X)
READ (X)	
Sum += X	
READ (Y)	
Sum += Y	
	READ (Y)
	Y = Y + n
	WRITE (Y)

reads incorrect value of y.

∴ Summary = (actual − n)

c) UNREPEATABLE READ: This problem occurs when another transaction modifies a data item, read continuously by another transaction - thus leading to read two different values of the same data item, when a single value is expected.

The five aforementioned problems depict the need of concurrency control.

Transaction Properties (ACID properties):

a) Atomicity - A transaction is an atomic unit of processing, either performed in entirity or not performed at all.

b) Consistency - Transactions must take the database from one consistent state to another.

c) Isolation - A transaction should not make its updates visible to other transactions, until it is committed.

d) Durability - Once a transaction changes the database, and the changes are committed, these changes must never be lost because of subsequent failures. (Both T_1 and T_2 need to be aborted and the transactions undone in S_b)

Transaction Operations : Transaction States.

Start. transaction : Active

Marks the beginning of transaction execution. The transaction enters the active state.

read / write

These specify read/write operations on the database items, that are executed as part of transactions.

End. transaction : Partially committed

Specifies read/write operations have ended, and marks the end limit of transaction execution.

At this point it is necessary to check whether the changes introduced

by the transaction can be permanently applied to the database, or that the transaction has to be aborted if it has violated concurrency checks or for any other reason.
The concurrency control techniques require to ensure that the transaction didnot interfere with other executing transactions. Also, some recovery protocol needs to ensure that a system failure will not result in an inability to record the changes in a transaction permanently.

Commit : Committed

This signals successful end of the transaction, so that any changes executed by the transaction, can be safely committed to the database.
If all the concurrency checks are successful, the transaction is said to have reached its commit point and enters the committed state.

Abort / rollback : failed

This signals that the transaction has ended unsuccessfully, so that any changes/effects that the transaction may have applied to the database, must be undone. A transition can go to the failed state, if one of the checks fails or if it is aborted during its active state. The transaction may then have to be rolled back.

terminated.

This state corresponds to the transaction leaving the system.

Transaction log:

The recovery manager (system) maintains a log to keep track of all transaction operations. It permits recovery from transaction failures. The log is periodically backed up to archival storage, to guard against disk/catastrophic failures.
Typical log entries: (Donot concern the operations).
[start. transaction T] ⇒ T (transaction identifier) has started execu-

tion.

[Write.T, x, old_value, new_value] {x: database item, presence of "old_value" in the log entry allows undo transaction operations}.

[Read.T, x] (not a necessary log entry)

[Commit T] ⇒ T has successfully completed and affirms that its effect can be committed to the database.

[Abort T] ⇒ T has been aborted.

(Sometimes, we might have to redo - execution of the log - to acquire the present or latest database scenario — due to the occurrences of failures / catastrophe.)

** Some recovery mechanisms perform deferred updates while others perform immediate updates.

Schedules: A schedule S of n transactions: $T_1, \ldots, T_n$, is an ordering of the operations of the transactions, so that for each transaction T_i in S, the operations of T_i in S, must appear in the same order in which they appear in T_i. However, the operations from other transactions T_j can be interleaved with the operations of T_i in S. [S_a, S_b, S_c].

S_a	S_b
$r_1(x)$	$r_1(x)$
$r_2(x)$	$w_1(x)$
$w_1(x)$	$r_2(x)$
$r_1(y)$	$w_2(x)$
$w_2(x)$	c_2
c_2	$r_1(y)$
$w_1(y)$	a_1
c_1	

conflicting operations

r_i ⇒ read in T_i

w_i ⇒ write in T_i

c_i ⇒ commit T_i

a_i ⇒ abort T_i

Conflicting operations:

Two operations are said to conflict if:

a) they access the same database item.

b) must belong to different transactions.

c) atleast one operation is a write operation.

Recoverable schedule:

A schedule is said to be recoverable, if no transaction T in S, commits until all transactions T', that have written an item, that T reads, have committed.

Following S_b, $T' = T_1$, $T = T_2$,

we have the sequence, $w_1(x)$

$r_2(x)$

However, T commits prior to T', and thus S_b is not recoverable [S_a on the other hand is recoverable].

$r_1(x), w_1(x), r_2(x), r_1(y), w_2(x), w_1(y), a_1, a_2$

↳ postponed to after commit(1) in case...

Cascading rollback:

When an uncommitted transaction has to be rolled back, since it read an item written by a transaction that failed, the phenomenon is termed cascading rollback.

As in S_b, T_2 is rolled back since T_1 has failed and needs to be rolled back. ("the "isolation" property ceases to hold).

** A schedule is said to avoid cascading rollback, if every transaction in the schedule only reads items that were written by committed transactions.

• cascadeless – ∀ transactions ∈ schedule, read items only in committed transactions.

$x=9$; $w_1(x,5)$, $w_2(x,8)$, a_1

⇒ cascadeless but not strict

Strict schedule:

Another restricted schedule called strict schedule, where transactions can neither read or write an item X, until the last transaction that wrote X has committed / aborted.

All strict are cascadeless, but not vice-versa.

Serializability Theory:

The database system must control concurrent execution of transactions, to ensure that the database state remains consistent.

All transactions are mutually independent as we have assumed. Say, we have transactions $T_1, T_2, \ldots, T_n$, then serially we can have n! serial schedules. As we have assumed that the transactions are mutually independent, then any schedule is a correct schedule. So,

Every serial schedule is correct.

For schedule S_b, which is a non-serial schedule, it is serializable as it is equivalent to a serial schedule.

Definition: A schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions.

** Clearly schedule S_a is not serializable, whereas S_b is serializable (provided all operations work).

Equivalence of schedules:

- Result equivalence: (as shown before)

Consider a schedule S_1 and S_2 with one transaction each:

S_1	S_2
READ (x)	READ (x)
X = X+10	X = X * 1.1
WRITE (x)	WRITE (x)

Say initially x=100; then both the transactions/schedules produce 110 each. But this doesnot state their equivalence, so in this case we discard result equivalence.

- Conflict equivalence:

Now, if we rewrite S_a as:

S_a	
$r_1(x)$	
$r_2(x)$	conflicting operations
$w_1(x)$	conflicting operations
$r_1(y)$	conflicting operations
$w_2(x)$	conflicting operations
c_2	
$w_1(y)$	
c_1	

Say we have a serial schedule S:

S_c	S_d
$r_1(x)$	$r_2(x)$
$w_1(x)$	$w_2(x)$
$r_1(y)$	c_2
$w_1(y)$	$r_1(x)$
c_1	$w_1(x)$
$r_2(x)$	$r_1(y)$
$w_2(x)$	$w_1(y)$
c_2	c_1

If for 2 schedules (with the same transactions) all the conflicting operations are in the same order, then it is termed conflicting equivalence.

Now, if a non-serial schedule is conflicting equivalent to a serial schedule, then this non-serial schedule is always correct, as a serial schedule is always correct.

Here, it is evident that S_a is neither conflict equivalent to S_c nor to S_d, as the order of their conflicting operations are not same. So, it is not serializable, hence not correct.

Considering another schedule S_b:

S_b	S_c
$r_1(x)$	✓ $r_1(x)$
$w_1(x)$	• $w_1(x)$ ✓
$r_2(x)$	$r_1(y)$
$w_2(x)$	$w_1(y)$
c_2	c_1
$r_1(y)$	• $r_2(x)$
$w_1(y)$	✓ $w_2(x)$ ✓
	c_2

conflicting operations

Here, we see that S_b is conflict equivalent to the serialized schedule S_c; thus, S_b is correct.

Definition:

- Two schedules are said to be conflict equivalent if the order of the conflicting operations is the same in both schedules. (any two conflicting op.)
- A schedule S is conflict-serializable if it is conflict equivalent to some serial schedule S'. [S_a is not conflict serializable to either of S_c or S_d]

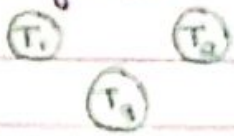
Test of conflict serializability:

Consider the following schedule with 3 transactions:

T_1	T_2	T_3
	r(z)	
	r(y)	
	w(y)	
		r(y)
		r(z)
r(x)		
w(x)		
		w(y)
		w(x)
	r(x)	
r(y)		
w(y)		
	w(x)	

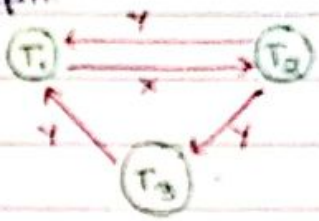
Construct a preceedence graph as follows :-

* For each transaction T_i in schedule S, create a node labelled T_i in the preceedence graph.

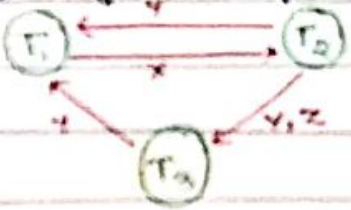


For each case in S :

* Where T_j executes a read (x) after a write (x) executed by T_i, create an edge from T_i to T_j in the preceedance graph



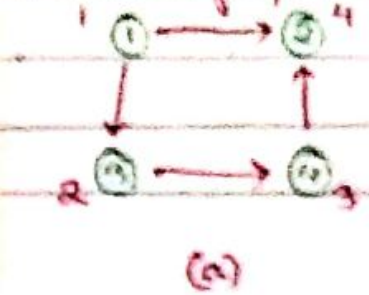
* When T_j executes a write (x) after T_i executes a read (x), create an edge from T_i to T_j in the preceedance graph



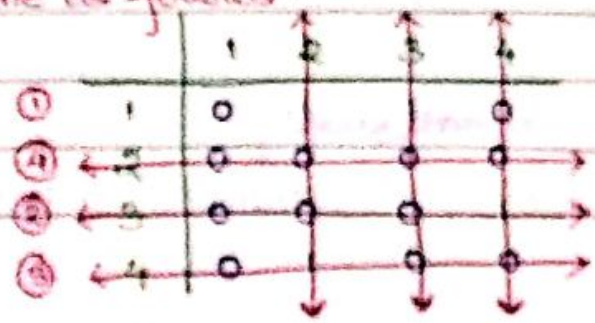
* When T_j executes a write (x) after T_i executes a write (x), create an edge from T_i to T_j in the preceedance graph

* The schedule S is serializable iff the preceedance graph has no cycles.

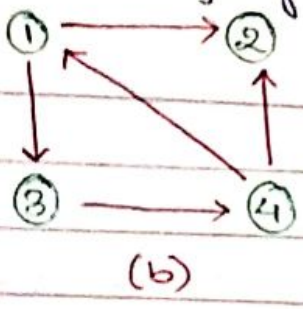
Numbering of the nodes is done as follows :



(a)



At each step we find out the terminating node in the remaining graph, i.e., the node whose row has all 0's; and number / label that node with the next highest number. We repeat this till all the nodes are successfully numbered.



(b)

	1	2	3	4
1	0		1	0
2	0		0	0
3	0		0	1
4	1		0	0

(4)

Here, after this step we donot find any other terminating node in the remaining graph; none of the rows have all 0 entries. Thus, we can conclude that the remaining graph has a cycle.

** Graph (a) has no cycles, so if it is a precedence graph of transactions, we can say that it is serializable; and its serializable schedule is of the order T_1, T_3, T_4, T_2.

Whereas, graph (b) has cycles, and if it is a schedule graph, then it is not serializable.

- View serializability and view equivalence:

Two schedules S, S' are said to be view equivalent if the following holds:

1) the same set of transactions participate in S and S'.

2) for any operation $r_i(x)$ of T_i in S, if the value of x, read by the operation, has been written by $w_j(x)$ of T_j (or if it is the original value of x, before the schedule started), the same conditions must hold for the value of x read by $r_i(x)$ of T_i in S'.

3) if $w_k(y)$ of T_k is the last operation to write y in S, then $w_k(y)$ of T_k must also be the last operation to write y in S'.

The concept behind view equivalence is that, as long as each read

operation of a transaction reads the result of the same write operation in both schedules, the write operations of each transaction must produce the same results. The read operations are hence said to see the same view in both schedules. Condition 3 ensures that the final write operation on each data item is the same in both schedules, so the database state should be the same at the end of both schedules.

A schedule S is said to be view serializable, if it is view equivalent to a serial schedule. $[r_1(x), w_2(x), w_1(x), w_3(x), c_1, c_2, c_3]$

blind write $\approx T_1 \to T_2 \to T_3$: view s. but not conflict ser.

**a) A serial schedule represents inefficient processing, since there is no interleaving of operations from different transactions.

b) A serializable schedule gives us the benefits of concurrent execution, without giving up any correctness.

c) It is practically impossible to determine, how the operations of a schedule will be interleaved beforehand (since they are typically determined by the OS scheduler).

d) If transactions are executed at will, and then the resultant schedule is tested for serializability, we must cancel the effects of the schedules if it does not turn out to be serializable. [-clearly an impractical approach].

e) A more practical approach is to determine protocols or sets of rules, that if followed by every individual transaction or if enforced by a DBMS concurrency control subsystem will ensure serializability of all the schedules, in which the transactions participate.

Protocols : (Ensures the isolation property of concurrent execution)

• Lock Based Protocols :-

These protocols employee the technique of locking data items to prevent multiple transactions from accessing the items concurrently. A lock is a variable associated with a data item Lock(x) that describes

the status of the item with respect to possible operations that can be applied to it. Generally, there is one lock for each data item in the database. Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

a) Binary locking:

A binary lock can have two states or values : locked (1) and unlocked (0). A distinct lock is associated with each database item X. If the value of the lock on X = 1, X cannot be accessed by a database operation that requests the item ; If the value of the lock on X = 0, the item can be accessed when requested. The following two atomic operations are used with binary locking:

```
LOCK_ITEM (X) :
   L : IF (LOCK (X) == 0)
          LOCK (X) = 1 ;
       ELSE
          {
             WAIT (UNTIL LOCK (X) == 0 AND THE LOCK-MANAGER WAKES
                UP THE TRANSACTION) ;
             GOTO L ;
          }
UNLOCK_ITEM (X) :
   LOCK (X) = 0 ;
   IF THERE ARE ANY WAITING TRANSACTIONS,
      WAKEUP ONE OF THEM ;
```

The DBMS has a lock-manager subsystem to keep track of and control access to locks.

The binary locking scheme protocol:

A transaction T-

a) Must issue LOCK_ITEM (X) before any read (X) or write (X) operations are performed in T.

b) Must issue UNLOCK_ITEM (X) after all read (X) or write (X) ope-

rations are completed in T.

c) will not issue LOCK_ITEM(X) if it already holds a lock on X.

d) will not issue UNLOCK_ITEM(X) unless it already holds a lock on X.

The binary locking scheme is restrictive, since atmost one transaction can hold a lock on a given item at an instant. If more than one transaction tries to access X only for reading, we should allow them to access X simultaneously.

b) Multiplemode locking:

In this scheme, a lock associated with an item X has three states: read_locked/shared_locked (1), write_locked/exclusive_locked (2) and unlocked (0). The following three atomic operations are used in multiplemode locking:

```
READ_LOCK(X):
  L: IF (LOCK(X)==0)
     {
        LOCK(X)=1;
        NO_OF_READS(X)=1;        keep track of the number of transactions holding a shared_lock on an item.
     }
     ELSE IF (LOCK(X)==1)
        NO_OF_READS(X)++;
     ELSE
     {
        WAIT (UNTIL LOCK(X)==0 AND THE LOCK_MANAGER WAKES
              UP THE TRANSACTION);
        GOTO L;
     }
WRITE_LOCK(X):
  L: IF (LOCK(X)==0)                WAIT (UNTIL LOCK(X)==0 AND THE LOCK_
     LOCK(X)=2;                           MANAGER WAKES UP THE TRANSAC-
     ELSE                                 TION);
     {                                 GOTO L;
                                    }
```

```
UNLOCK (X):
    IF (LOCK(X) == 2)
        {
          LOCK(X) = 0;
          WAKE UP ONE OF THE WAITING TRANSACTIONS, IF ANY;
        }
    ELSE IF (LOCK(X) == 1)
            {
              NO_OF_READS(X)--;
              IF (NO_OF_READS(X) == 0)
                 {
                   LOCK(X) = 0;
                   WAKE UP ONE OF THE WAITING TRANSACTIONS,
                       IF ANY;
                 }
            }
```

The multimode locking scheme protocol:

A transaction T:

a) Must issue READ_LOCK(X) or WRITE_LOCK(X) before any r(X) is performed in T

b) Must issue WRITE_LOCK(X) before any w(X) is performed in T.

c) Must issue UNLOCK(X) after all r(X) and w(X) are completed in T.

d) Will not issue READ_LOCK(X) and/or WRITE_LOCK(X) if it already holds a read/write lock on X. (May be relaxed in certain cases).

e) Will not issue UNLOCK(X) unless it already holds a read/write lock on X.

An example of transactions that maintain the aforementioned protocol:

T_1	T_2
rlock (Y)	
r(Y)	
unlock (Y)	
	rlock (X)
	r(X)
	unlock (X)
	wlock (Y)
	r(Y)
	Y = X+Y
	w(Y)
	unlock (Y)
wlock (X)	
r(X)	
X = X+Y	
w(X)	
unlock (X)	

Initially: X = 20 ; Y = 30.

S_1

T_1 $\Rightarrow$ X = 50 ; Y = 80

T_2

S_2

T_2 $\Rightarrow$ X = 70 ; Y = 50.

T_1

} serial schedules

Now following the schedule shown alongside: X = 50 ; Y = 50.

As is evident, the schedule alongside is not equivalent to any of the serial schedules ; and is thus incorrect.

Observations:

a) Using binary or multiplemode locking schemes donot guarantee serializability of schedules.

b) An additional protocol, concerning the positioning of the locking and the unlocking operations, in every transaction must be followed.

c) Two-phase Locking protocol (2PL) :

(Basic) A transaction is said to follow the 2PL protocol if all locking operations preceed the first unlock operation in the transaction. There are two phases in a transaction :

a) An expanding / growing phase, during which new locks on items can be acquired, but none can be released.

b) A shrinking phase, during which existing locks can be released but no new locks can be acquired.

Neither T_1 nor T_2 follow the 2PL. The 2PL equivalent stands as:

T_1'	T_2'
rlock (Y)	rlock (X)
r(Y)	r(X)
wlock (X)	wlock (Y)
unlock (Y)	unlock (X)
r(X)	r(Y)
X := X+Y	Y := Y+X.
w(X)	w(Y)
unlock (X)	unlock (Y)

* It can be proved that, if every transaction in a schedule follows the 2PL protocol, the schedule is guaranteed to be serializable.

Two-phase locking may limit the amount of concurrency that can occur in a schedule. This is because, a transaction T may not be able to release an item X after it is through using it, if T must lock an additional item Y later on; or conversely, T must lock the additional item Y before it needs it so that it can release X. Hence X must remain locked by T until all items that the transaction needs to read or write have been locked; only can then X be released by T. Meanwhile, another transaction seeking to access X may be forced to wait, even though T is done with X; conversely, if Y is locked earlier, than it is needed, another transaction seeking to access Y is forced to wait even though T is not using Y yet. (Deadlock and starvation)

(Conservative / static): This scheme requires a transaction to lock all the items it accesses, before the transaction begins to execute, by predeclaring its readset and writeset. The readset of a transaction is the set of all items that the transaction reads, and the writeset is the set of all items that it writes. If anyone of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking. Is a deadlock-free protocol; difficult to use in practise because of the need to predeclare the readset and the writeset.

(strict) The most popular 2PL scheme. It guarantees strict schedules. Here, a transaction T does not release any of its exclusive (write)

locks until after it commits or aborts. Hence, no other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability. Is not deadlock free. (Rigorous) A more restrictive variation of strict 2PL, which also guarantees strict schedules. Here, a transaction T does not release any of its locks until after it commits or aborts, and so is easier to implement than strict 2PL.

** The conservative 2PL must lock all its items before it starts, so once the transaction starts it is in its shrinking phase, whereas rigorous 2PL does not unlock any of its items until after it terminates (by committing or aborting), so the transaction is in its expanding phase until it ends.

2PL - basic ; conservative, strict, rigorous.
(dead) (x) (recov but dead) (no deadlock)

** A deadlock situation:

T_1	T_2
wlock(Y)	
r(Y)	
	wlock(X)
	r(X)
wlock(X)	
wait states.	wlock(Y)

Deadlock handling mechanisms:

a) Deadlock prevention - Not allowing deadlocks to occur at all. Restrictive => less throughput.

b) Deadlock detection - Allow deadlocks → identify reason → abort.

T_1	T_2
rlock(Y)	
read(Y)	
	rlock(X)
	read(X)
wlock(X)	
	wlock(Y)

• Deadlock prevention protocols :-

Is used in situations concerning heavy transaction load. The various deadlock prevention schemes are:

a) Conservative 2PL.

b) Assignment of an arbitrary linear ordering to each data item, and ask the transaction authors to lock items only in that order (not a practical assumption - as the programmer or the system requires to be aware of the chosen order of the items.

c) Use of transaction timestamps. (Might be implemented by counters - limited by permissible maximum value.) Here, if TS(T) indicates the timestamp of the transaction T, then $TS(T_i) < TS(T_j)$ if T_i is an older transaction. The various schemes under this methodology are as follows :-

i) Wait-die scheme: If T_i tries to lock X but is not able since X is locked by T_j; Here, (waiting for younger)

IF $TS(T_i) < TS(T_j)$

THEN T_i IS ALLOWED TO WAIT

ELSE

ABORT T_i (T_i dies) AND RESTART IT LATER WITH THE SAME TIMESTAMP.

ii) Wound-wait scheme: If T_i tries to lock X but is not able since X is locked by T_j; Here, (waiting for older)

IF $TS(T_i) < TS(T_j)$

THEN ABORT T_j (T_i wounds T_j) AND RESTART IT LATER WITH THE SAME TIMESTAMP.

ELSE

T_i IS ALLOWED TO WAIT.

** a) It can be shown that both schemes are deadlock free.

b) Both schemes end up aborting the younger of the two transactions, that may be involved in a deadlock.

c) Both techniques cause some transactions to be aborted and re-started, even though those transactions may never actually cause a deadlock.

d) Both schemes avoid starvation, i.e., no transaction gets aborted repeatitively, since timestamps always increase and transactions are not alloted new timestamps when aborted. A transaction that is aborted, will eventually have the smallest timestamp.

d) No-waiting algorithm - In this algorithm, if a transaction is unable to obtain a lock, it is immediately rolled back and restarted after a certain

time delay, without checking whether a deadlock will actually occur or not.

**Transaction rollback and restart occurs unnecessarily.

e) Cautious waiting approach – If T_i tries to lock X but is unable as X is locked by T_j.

IF T_j IS BLOCKED (waiting for some locked item)

THEN ABORT T_i AND RESTART IT LATER.

ELSE

{

SET STATUS OF T_i AS BLOCKED AND ALLOW IT TO WAIT

}

** It can be shown that this scheme is deadlock free, by considering the time b(T) at which each blocked transaction T was blocked. If the two transactions T_i and T_j both become blocked, and T_i is waiting on T_j, then $b(T_i) < b(T_j)$, since T_i can only wait at a time when T_j is not blocked. Hence, the blocking times form a total ordering on all blocked transactions, so no cycle that causes a deadlock can occur.

f) Lock timeout – If a transaction waits longer than a system defined timeout, the system assumes that the transaction is deadlocked and aborts it, regardless of whether a deadlock situation actually exists.

** In wait-die, an older transaction is allowed to wait on a younger transaction, whereas, a younger transaction requesting an item held by an older transaction is aborted and restarted. Whereas, in the wound-wait scheme, a younger transaction is allowed to wait on an older one, and an older transaction requesting an item held by a younger transaction preempts the younger transaction by aborting it. Both schemes result in aborting the younger of the two transactions,

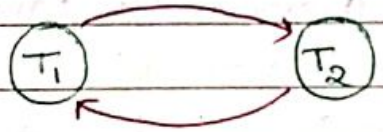
that may be involved in a deadlock. These two techniques are deadlock-free, since in wait-die transactions, transactions only wait on younger transactions so no cycle is created. Similarly, in wound-wait, transactions only wait on older transactions so no cycle is created.

- Deadlock detection protocols:

Is used in situations concerning low transaction load.

Deadlocks can be described precisely in terms of a directed graph, called a wait-for graph. A node is created in the wait-for graph for each transaction that is currently executing in the schedule. Whenever a transaction T_i is waiting to lock an item x that is currently locked by a transaction T_j, create an edge $T_i \rightarrow T_j$. When T_j releases locks on the items that T_i was waiting for, the directed edge is dropped from the wait-for graph. We have a state of deadlock iff the wait-for graph has a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for graph and periodically to invoke an algorithm that searches for a cycle in the graph.

Following the deadlock situation:- (as per the transaction stated before)



(An edge is drawn for each dependancy. The graph is updated regularly).

- Recovery from deadlock:

a) Victim selection: Given a set of deadlock transactions, the victim selection algorithm determines which transactions to rollback to break the deadlock. It should rollback those transactions that will incur minimum cost, which depends on factors like :-

- How long the transaction has computed, and how much longer it

will compute to complete its designated task.

- How many data items the transaction has used, and how many more the transaction needs for it to complete.
- How many transactions will be involved in the rollback.
- How many times a single transaction has been rolled back. (Starvation prevention).

b) Rollback: Once the victim selection is over, it must be determined to how far this transaction should be rolled back. The simplest solution is total rollback (abort) and restart.

However, it is more effective to roll back the transaction only as far as necessary to break the deadlock. But, this method requires the system to maintain additional information about the state of all the running transactions.

c) Starvation: In a system where the victim selection is based on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task. This situation is called starvation.

One solution to avoid starvation is to include the number of rollbacks in the cost factor.

d) Livelock: Another problem that may occur when it uses locking is livelock. A transaction is in a state of livelock, if it cannot proceed for an indefinite period of time, while other transactions in the system continue normally. This may occur if the waiting scheme for locked items is unfair, giving priority to some transactions over others.

The standard solution to livelock, is to follow a scheme that uses a FIFO queue, where transactions are enabled to lock an item in the order in which they originally requested to lock the item.

- Basic Timestamp Ordering (TO) protocol: Another approach that guarantees serializability, involves using transaction timestamps to order transaction executions for an equivalent serial schedule.

The basic TO protocol, associates with each database item x, two time-stamp values:

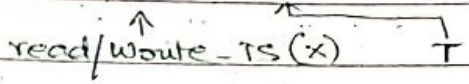
READ_TS(x) - The read timestamp of x. This is the largest time-stamp, amongst all transactions that have successfully read item (x).

WRITE_TS(x) - The write timestamp of x. This is the largest time stamp, amongst all transactions that have successfully written item (x).

The TO protocol is as follows:

We have transactions T_1, T_2, T_3, T_4, T_5 where $TS(T_1) < TS(T_2) < \cdots < TS(T_5)$.

If there be a schedule : T_2, T_1, T_5, T_4, T_3 where:



Scenario 1 :

T requests a WRITE(x)

- If (read_TS(x) > TS(T))

⇒ the value of x that T was producing, was needed previously, and the system assumed that, that value would never be produced. Hence, the WRITE(x) operation is rejected and T is aborted.

- If (write_TS(x) > TS(T))

⇒ that T is attempting to write an obsolete value of x. Hence, this WRITE(x) operation is rejected and T is rolled back.

- Otherwise

the WRITE(x) operation is executed and WRITE_TS(x) is set to TS(T)

Scenario 2 :

T requests a READ(x)

- If (write_TS(x) > TS(T))

⇒ T needs to read a value of x, that was already overwritten. Hence READ(x) is rejected and T is rolled back.

- If (write_TS(x) ≤ TS(T))

⇒ the READ(x) operation is executed and Read_TS(x) = max{Read_TS(x), TS(T)}.

Transaction T that is rolled back by the concurrency control scheme, as a result of either a read/write operation being issued is assigned a new timestamp

and is restarted. * locking + timestamp: pessimistic :: force wait/rollback
② Validation protocol (local page)

**This protocol guarantees conflict serializability since it ensures that, any conflicting read and write operations are executed in timestamp order.

** One of the problems associated with this protocol is that, it does not avoid cascading rollback.

• A modification of the basic TO protocol - known as Thomas' Write Rule - does not enforce conflict serializability, but it rejects fewer write operations by modifying the checks for WRITE(X) operations as follows. If (WRITE_TS(X) > TS(T)) then T is attempting to write an obsolete value of X. Therefore this write operation is ignored, and processing is continued without aborting T. [The other two checks, as in the basic TO protocol, remain identical in this case too.] write_item(x) issued:
If read_TS(X) > TS(T) then abort, rollback T, reject operation. If write_TS(x) > TS(T)
Continue processing :: write_item(x) already outdated. Any conflicts resolved by (1).

** Thomas Write Rule makes use of view serializability by deleting obsolete write operations by transactions that issue them.

* strict TO: If read_(x)/write_(x) issued, such that TS(T) > write_TS(x); then r/w delayed until T' (that wrote X) has committed (i.e., WRITE_TS(x) = TS(T'))

** These protocols ensure deadlock avoidance, as no locks are issued.

$w_1(A)$,	$w_1(B)$	$w_2(A)$	$r_2(B)$	c_1 c_2 → recoverable, serializable
$w_2(A)$	$w_1(B)$	$w_1(A)$	$r_2(B)$	c_1 c_2 → ~~deadlock~~ recov, nonserializable (inconsistent)
$w_1(A)$	$w_1(B)$	$w_2(A)$	$r_2(B)$	c_2 c_1 → non recov, serializ ⇒ If no failures then consistent else incon.
$w_1(A)$	$w_2(B)$	$w_2(A)$	$r_1(B)$	c_2 c_1 → non recov, nonseriali

• Recovery techniques:

If a transaction fails after executing some of its operations, but before executing all of them, the database system must have a recovery scheme for restoration of the database to a consistent state that existed prior to the occurence of the failure.

To make a successful recovery from transaction failures, the system log keeps the information about the changes to data items during transaction execution, outside the database.

a) Database backup and recovery from catastrophic failures : The main

Technique used to handle catastrophic failures is that of data... backup. The whole database and the log are periodically copied on a stable storage media. In case of catastrophic failures, the latest backup copy can be reloaded to disk, and the system can be restarted. The system log is usually substantially smaller than the database itself, and hence can be backed up more frequently. A new system log is started after each backup operation. Hence, to recover from disk failures :

a) the database is first recreated on disk from its latest backup copy.

b) then the effects of all the committed transactions whose operations have been entered in the backup copy of the system log, are then reconstructed.

b) Other techniques :-

- Deferred update (NO-UNDO / REDO algorithm)
- Immediate Update (UNDO / REDO algorithm)

[Checkpoint entries in a system log - Checkpoints are another entry type in a system log. The recovery manager decides at what intervals to take a checkpoint. Taking checkpoints consists of the following actions : i) Suspend execution of transaction temporarily.

ii) Force write all update operations of committed transactions from main memory buffer to disk.

iii) Write a checkpoint record in the log and force write the log to disk. [log checkpoint entry : CHECKPOINT]

iv) Resume transaction execution.

Recovery based on deferred update : The idea behind deferred update technique is to defer/postpone any updates to the database itself until the transaction completes its execution successfully and reaches its commit point. The typical deferred update protocol is as follows :-

1. A transaction cannot change a database until it reaches its commit point.

2. A transaction does not reach its commit point until all its update operations are recorded in the log and the log is force written to disk.

If a transaction fails before reaching its commit point, it will not have changed the database in any way. Thus UNDO is not needed. It may be necessary to REDO the effect of the operations of a committed transaction from the log, since their effect may not have been recorded in the database [i.e., changes reflected in the cache but not in the database – checkpoint not taken]. Thus, such an algorithm is named a NO-UNDO/REDO algorithm.

We consider a system in which concurrency control uses the two-phase locking protocol. To combine deferred update with this protocol, we keep all the locks on items in effect until the transaction reaches its commit point, after which all the locks are released. This ensures a strict and serializable schedule. Assume that checkpoint entries are included in the log.

Recovery using deferred update in the multiuser environment:

PROCEDURE RDU-M

Use two lists of transactions by the system

- Committed transactions list (T) since last checkpoint.
- Active transactions list (T')

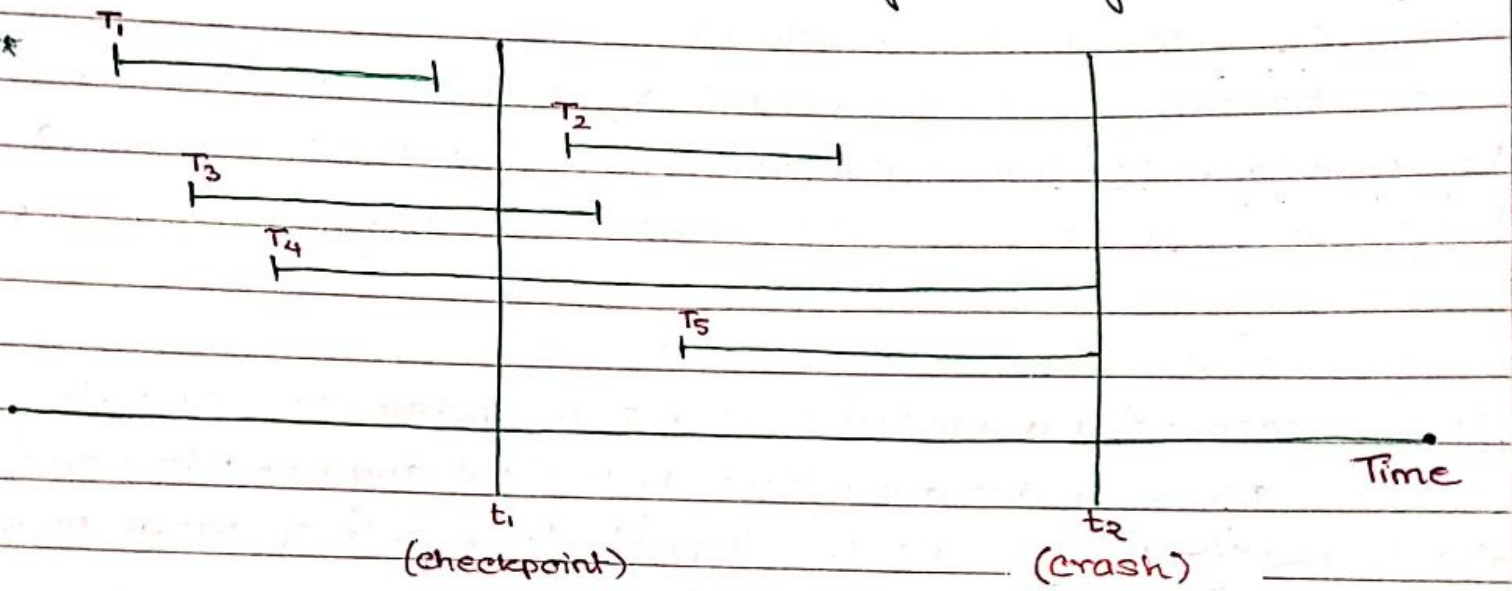
REDO all the write operations of the committed transactions from the log, in the order in which they were written into the log. The transactions that are active and did not commit, are effectively cancelled and must be resubmitted.

REDO (write_op)

Redoing a write operation consists of examining its log entry [WRITE T, x, new_val] and setting the value of item(x) in the database to the new_val.

(NO-UNDO, since old_val is not stored).

** the REDO operation needs to be idempotent. In fact, the whole recovery process should be idempotent. Clearly, if the system fails during the recovery process the next recovery might redo certain write operations that had already been redone. The result of recovery from a crash during recovery should be the same as the result of recovery when there is no crash during recovery.

** 

a) T_1 has committed before checkpoint was taken at time t_1, but T_3 and T_4 have not committed yet.

b) Before the system crash, at time t_2, T_3 and T_2 have committed but T_4 and T_5 have not.

c) Thus REDO of T_1 is not necessary.

d) But REDO of write-op of T_2 and T_3 is required, as they have committed but after checkpoint time.

e) T_4 and T_5 are ignored. They are effectively cancelled / rolled back. They are to be resubmitted.

** Instead of redoing every modification on the same data item X, it would be economical to set X to its final modified value and maintaining a REDONE list, such that during the bottom up traversal of the REDO list, X is not re-redone.

Disadvantages of deferred update:
Limits the concurrent execution of transactions, since all items remain locked until the transaction reaches its commit point.

Advantages of deferred update:
Transaction operations never need to be undone; the reasons being:

a) A transaction does not record its changes in the database until it reaches its commit point (no rollback)

b) A transaction will never read the value of an item, that is written by an uncommitted transaction, since items remain locked until a transaction reaches its commit point. (no cascading rollback).

• Recovery based on immediate update:
In this case, when a transaction issues an update command, the database can be updated immediately without any need to wait for the transaction to reach its commit point. An update operation must be recorded in the log on disk, before it is applied to the database, so that a recovery can be easily made. The two types of immediate update are:

a) If the recovery technique ensures that all updates of a transaction are recorded in the database on disk, before the transaction commits, there is never a need to REDO any operation of committed transactions. Such an algorithm is called UNDO (active) / NO-REDO (committed). Algorithm

b) If the transaction is allowed to commit before all its changes are written to the database, then we use UNDO (active) / REDO (on commit list) Algorithm.

Considering a system in which concurrency control uses the two-phase locking protocol in conjunction with the immediate update technique. Assumption - checkpoints are included in the log.

Recovery using immediate update in the multiuser environment:

PROCEDURE RIU-M

Use two lists of transactions maintained by the system
- a list of committed transactions (T) since last check point
- a list of active transactions (T')

UNDO all the write-op of active transactions using UNDO in the reverse order in which they were written into the log.

REDO all write-op of committed transactions using REDO in the order in which they were written into the log.

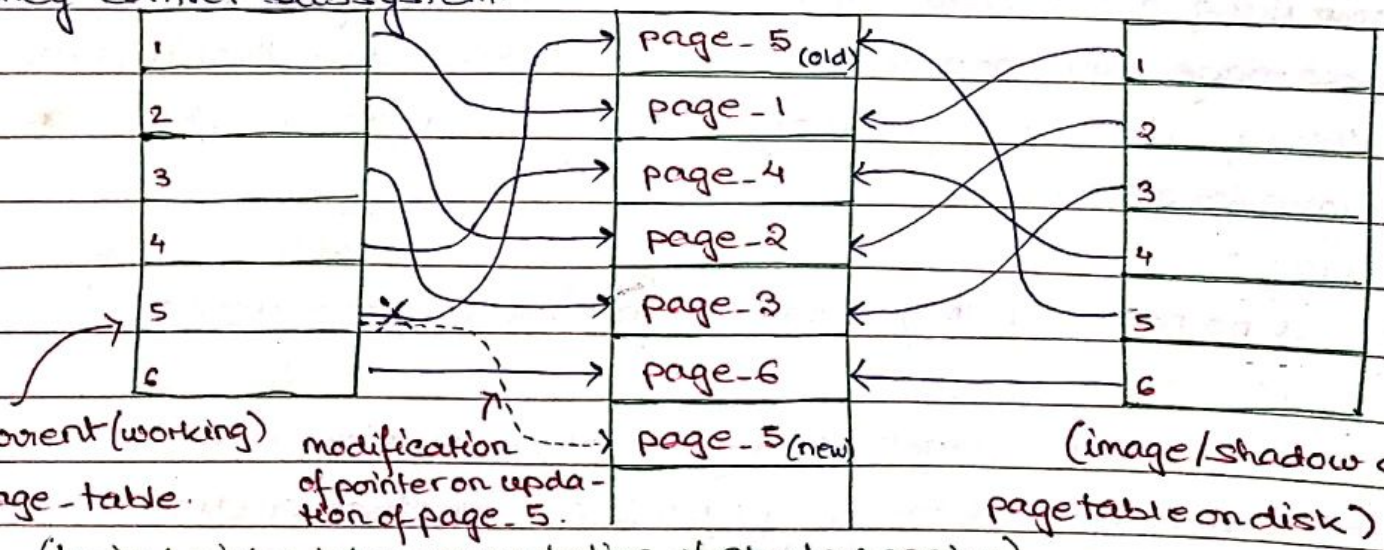
UNDO (write-op)

Undoing a write-op consists of examining its log entry [WRITE T, X, old-val, new-val] and setting the value of X in the database to the old-val.

Undoing a number of write-op from one/more transactions from the log must proceed in the reverse order from the order in which the operations were written in the log.

• Shadow Paging:

A recovery technique that uses a NO-UNDO/NO-REDO scheme. It does not use a log, but may require a log if required by the concurrency control subsystem.



(typical pictorial representation of shadow paging)

Shadow paging considers the database to be made up of a number of fixed size disk pages or blocks, say n.

A page_table with n entries is constructed, where the ith page_table entry points to the ith database page on disk. The page_table is kept in main memory if it is not too large. When a transaction begins execution, the current page_table, whose entries point to the most recent/current database pages on disk, is copied into a shadow page_table, and this shadow page_table is then saved on disk.

During transaction execution, the shadow page_table is never modified. When a write_op is performed, a new copy of the modified database is created, but the old copy of the page is not overwritten. The current page_table entry is modified to point to the new disk block, whereas, the shadow page table is not modified and continues to point to the old disk blocks. For pages updated by the transaction, two versions are kept – the old version is referenced by the shadow page table and the new version by the current page table.

To recover from a failure during transaction execution, it is sufficient to free the modified database pages and to discard the current page table. The state of the database before transaction execution is recovered by reinstating the shadow page table, so that it becomes the current page table once more.

Committing a transaction corresponds to discarding the previous shadow page table and freeing the old pages on disk that it references. Clearly, this technique may be categorized as NO-UNDO/NO-REDO technique for recovery.

Advantages:

There is no need to UNDO/REDO any transaction operations.

Disadvantages:

i) Updated database pages change location on disk. Hence it is difficult to keep related database pages close together on disk without complex storage management strategies.

ii) If the page table is large, the overhead of writing shadow page tables to disk, as transactions commit, is significant.

iii) Each time that a transaction commits, the database pages containing the old version of data becomes inaccessible. Such pages are considered garbage. (Garbage may be created also as a side-effect of crashes.)

Periodically it is necessary to find all the garbage pages and to add them to the list of free pages. This process called garbage collection, imposes additional overhead and complexity on the system.

** Concurrent transactions - sharing pages - are difficult to maintain as simultaneous page-tables need to be updated when such a page is updated. Complex schemes for shadow page-table maintenance required.

- referential integrity
- entity integrity.

} Navathe : 107.

- Integrity Constraint - N (105, 15)
- Consistency constraint - K(10)

- Bucket overflows - korth 509.
- External hashing.

optimistic concurrency control

- Validation Protocol (deadlock prevention - transactions)

- Each transaction T_i executes in 2/3 different phases in its lifetime - depending on whether it is a read-only / update transactions.

A) Read phase ⇒ T_i reads values & stores in local variables & updates on local variables.

B) Validation ⇒ Determination if T_i to move to write phase without serializability violation; If failure here, then transaction abort.

C) Write ⇒ Temporary local updates → written to database.

~~$TS(T_i)$ = Validation TS (T_i) ; If $TS(T_j) < TS(T_k)$ ⇒ Serial equivalent to T_j, T_k.~~

Given $TS(T_k) < TS(T_i)$

① Finish (T_k) < Start (T_i) ⇒ T_k finishes before T_i ∴ Serial OR

② Start (T_i) < Finish (T_k) < Validation (T_i) ⇒ T_k completes before T_i starts validation ⇒ write phases non overlapping ∴ serializability maintained.

Read B	Read(B)	guards against cascading rollback
	B = B - 50	∵ actual write only after transaction
	Read(A)	issuing write has committed. Starvation
Read A	A = A + 50	possible. ∴ conflicting transactions temporarily blocked to enable long transactions
Validate	<validate>	to finish.
Display(A+B)	write(B) write(A)	