Introduction to DBMS

A DBMS consists of a collection of interrelated data and a set of programs to access those data. The set of data normally referred to as the database, contains information about one particular enterprise. The primary goal of DBMS is to provide an environment that is both convenient and efficient to use in retrieving and storing database information.

Library (database) R#, B# => primary key

R#	Name	Age	B#	Bname	Subject	Price
1	Amit	20	B1	Algebra	Math	100
1	Amit	20	B2	Co-ordinate	Math	150
2	Mita	19	B3	Optics	Physics	200
2	Mita	19	B4	Sound	Physics	250
3	Tina	21	B5	Organic	Chem.	175

Library (R#, Name, Age, B#, Bname, Subject, Price)

Student

R#	Name	Age
1	Amit	20
2	Mita	19
3	Tina	21

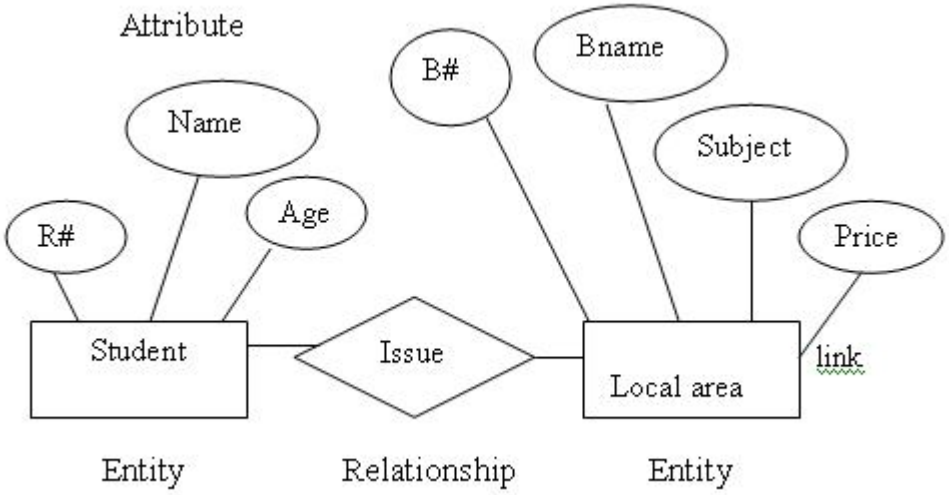
Book

B#	Bname	Subject	Price
B1	Algebra	Math	100
B2	Co-ordinate	Math	150
B3	Optics	Physics	200
B4	Sound	Physics	250
B5	Organic	Chem	175

ISSUE :

R#	B#
1	B1
1	B2
2	B3
2	B4
3	B5

Empname	Dname	Relationship
Abir	Tripti	Mother
Abir	Sankha	Son
Abir	Briti	Daughter
Soumen	Jayita	Sister
Soumen	Aritra	Brother
Soumen	Aritra	Daughter



Terminologies

- DBMS
- Tuple/Record
- Attribute/Field
- Domain
- Instance
- Schema
- Normalization(1 nf to 5 nf) and BCNF,DKNF/PJNF
- Anomaly
  - Insertion
  - Deletion
  - Updation
- Key → Foreign
- Primary
- Super
- Candidate
- Partial
- Secondary → Alternate key
- ER Diagram
- View
- Database Language
- DDL
- DML
  - Procedural (what,how)
  - Nonprocedural (what)
- Multivalued Attribute
- Derived Attribute
- Entity Set
- Relationship Set
- Links
- DDBMS
- CS Database (Client Server)

- Tuple / record- Each row wise entry in a database is called tuple/record.
- Attribute/field- Each tuple is consisting of set of attributes.
- Domain- It specifies the set of values that may be assigned to that attribute for each individual entity.
- Instance- The collection of information stored in the database at a particular moment is called an instance of the database.
- Schema- The logical description and the overall design of a database is called schema.
- ER diagram- The pictorial representation of the database.
- Normalization- It is a process to remove anomalies, redundancy and inconsistency of a database.
- Anomaly- Anomaly means difficulty.
- Multivalued attribute- It has a set of values of a specific entity e.g. dependent name of an employee.
- View- It is the permission through which a user will be allowed to see the portion of the database but not the full version to enhance the security measures.
- Super Key- It is a collection of one or more attributes which allows users to identify a tuple uniquely.
- Candidate Key-If all proper subsets of a super key is not a super key then that super key is called a candidate key i.e. minimal super key is called a candidate key.
- Primary Key- That super key selected by a DataBase Administrator (DBA) for implementation.
- Partial Key- Set of attributes that can uniquely identify weak entities related to the same owner entity. If no two dependent names (dname) are same then dname is the partial key.

Branch-name	Account number	Balance
Downtown	A-101	500
Mianus	A-215	700
Perryridge	A-102	400
Roundhill	A-305	350
Brighton	A-201	900
Redwood	A-222	700
Brighton	A-217	750

3.1 The Account relation

Customer-name	Account-number
Johnson	A-101
Smith	A-215
Hayes	A-102
Turner	A-305
Johnson	A-201
Jones	A-217
Lindsay	A-222

3.4 The Depositor relation

Branch-name	Branch-city	Assets
Downtown	Brooklyn	9000000
Redwood	Palo Alto	2100000
Perryridge	Horseneck	1700000
Mianus	Horseneck	400000
Roundhill	Horseneck	8000000
Pownal	Bennington	300000
North town	Rye	3700000
Brighton	Brooklyn	7100000

3.2 The Branch relation

Customer-name	Loan-number
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17
Adams	L-16

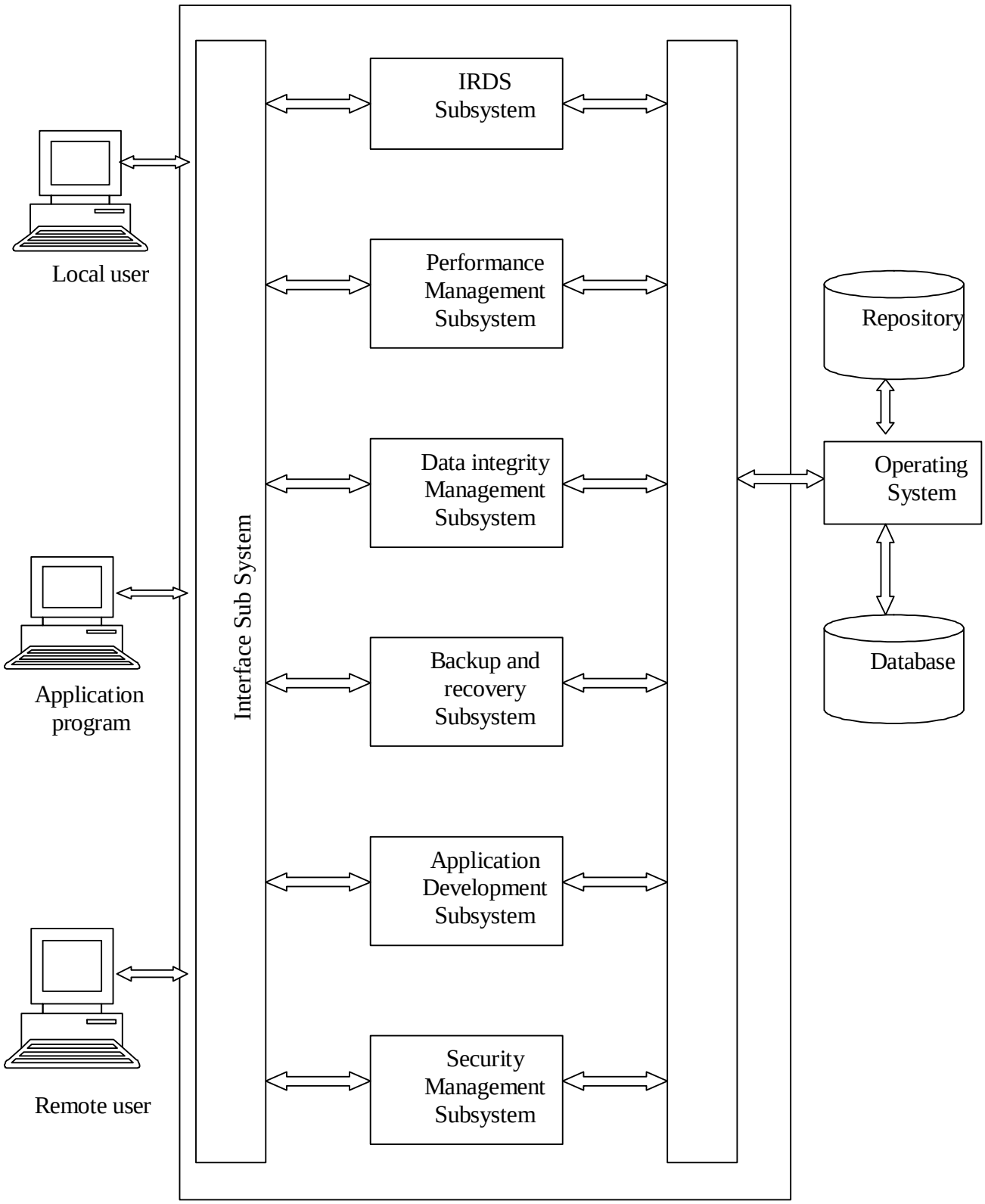
3.6 The Borrower relation

Customer name	Customer street	Customer city
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stanford
Williams	Nassan	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stanford

3.3 The Customer relation

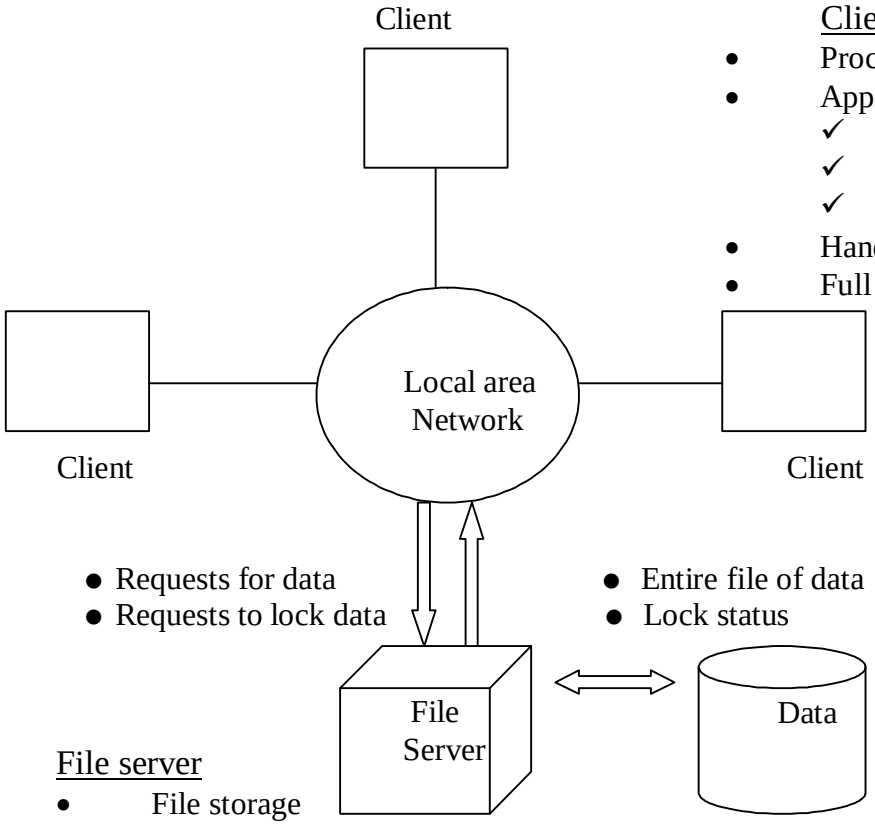
Branch name	Loan number	Amount
Downtown	L-17	1000
Redwood	L-23	2000
Perryridge	L-15	1500
Downtown	L-14	1500
Mianus	L-93	500
Roundhill	L-11	900
Perryridge	L-16	1300

3.5 The Loan relation



Components of a DBMS

IRDS means Information Repository Directory Subsystem i.e. a computer software tool that is used to manage and control access to the information repository.



Client
- Process scan table
- Application program
  - ✓ User interface
  - ✓ Database processing
  - ✓ Generate queries
- Handle integrity and security
- Full DBMS

➢ File server
- File storage
- Record locking
- Acts like extra hard disk to client
- Not very busy
- Significant LAN traffic

Fig.4.7 File Server model

☞ A form of LAN in which a central database server or engine perform all database commands sent to it from client workstations and application programs on each client concentrate on user interface functions.

.

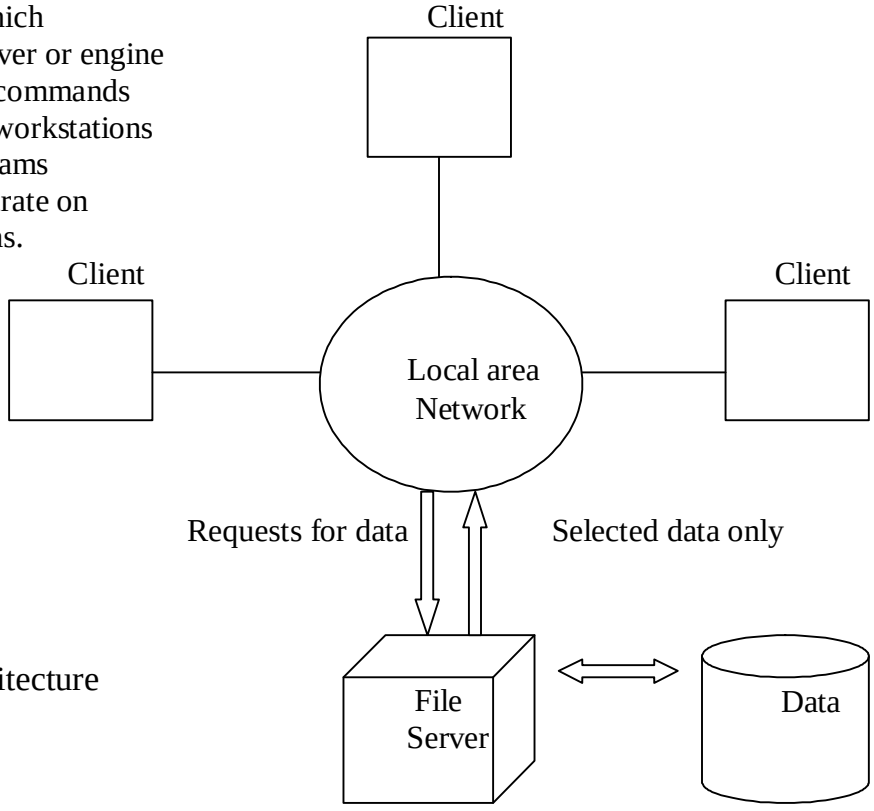


Fig.4.8 Client / Server architecture

- A client has the following characteristics:-
  - The client presents the user interface; that is, the client application program controls the PC screen, interprets data sent to it by the server and presents the results of database queries.
  - The client forms queries or commands in a specified language(often SQL) to gather the data needed by the application, and then sends these queries to the server; often, the preparation of queries is transparent to the user, since the user interface does not resemble this query language.
  - The client functions are performed on a separate computer from the database server functions; hence, neither the client nor the database server are complete application environments themselves.

- The server has the following characteristics:
  - The server responds to queries from clients, checks the syntax of these commands, verifies the access rights of the user, executes these commands, and responds with desired data or error messages.
  - The server hides the server system from the client and from the end user, so that the user is completely unaware of the server's hardware and software.

- Client/Server advantages:
  1. It allows companies to leverage the benefits of microcomputer technology. Today's workstations deliver impressive computing power at a fraction of the costs of mainframe.
  2. It allows most processing to be performed close to the source of data being processed, thereby improving response times and reducing network traffic.
  3. It facilitates the use of graphical user interfaces (GUIs) and visual presentation techniques commonly available for workstations.
  4. It allows for and encourages the acceptance of open systems.

DISTRIBUTED DATABASE

A single logical database that is spread physically across computers in multiple locations that are connected by a data communication link.

Options for distributed network: tree, star, ring, fully or partially connected.

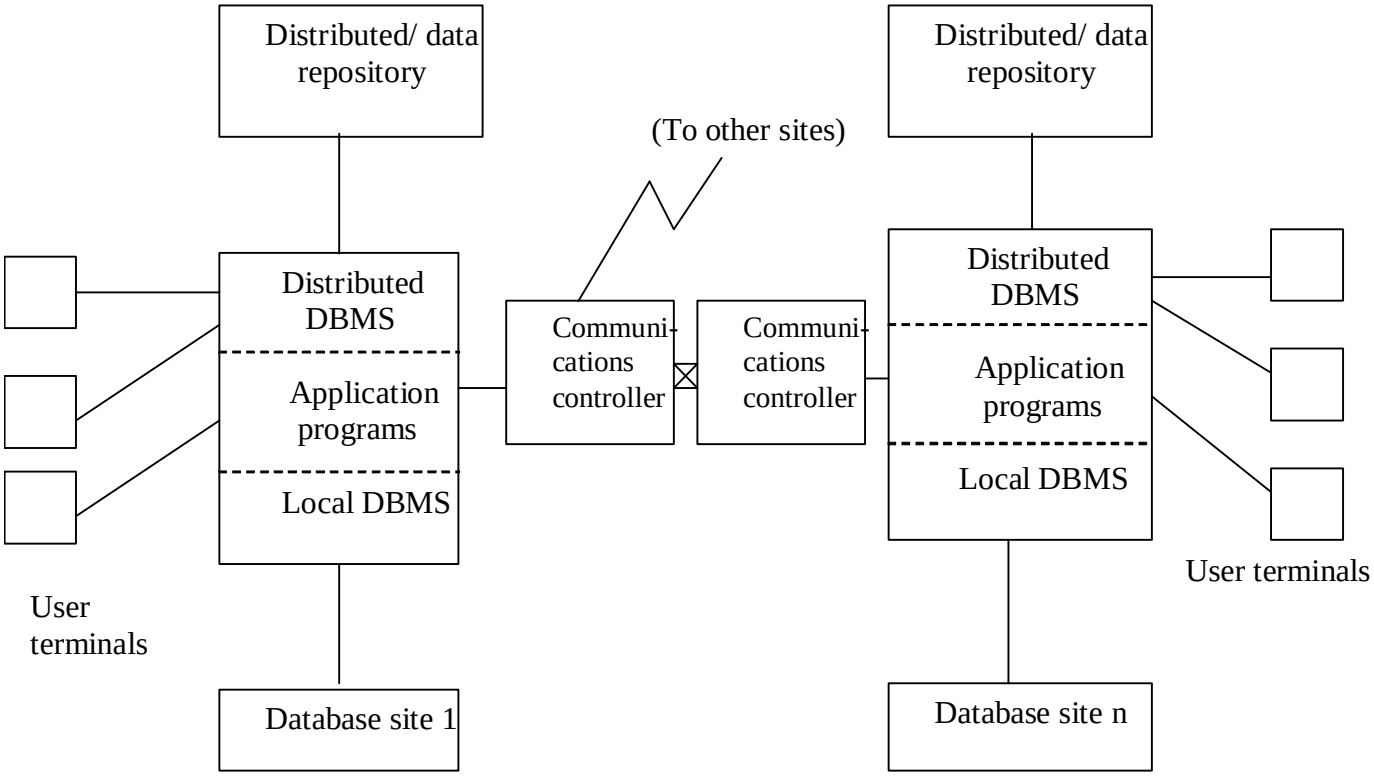


fig.4.6 Distributed DBMS architecture

- ✓ Advantages: Increased reliability & availability, local control, modular growth, lower communication cost, faster response.
- ✓ Disadvantages: Software cost and complexity, processing overhead, data integrity, slow response.
- ✓ Options for distributing database: Data replication, horizontal partitioning, vertical partitioning, combinations of the above.
- ✓ Features: Local & global transaction, location transparency replication transparency, failure transparency, Transaction manager, two phase commit protocol, concurrency transparency, time stamping, query optimization.

NORMALIZATION

In an unnormalized relation non-atomic values exist. In normalized forms, all anomalies (insertion, deletion and updation) and all inconsistencies and resolved in the database.

❋ Functional dependency (FD): It is a particular relationship between two attributes. For any relation R, attribute B is functionally dependent on attribute A if, for every valid instance of A that value of A uniquely determines the value of B. It is expressed as A→B. We can describe it in two ways.

- A determines B
- B is determined by A.

Example: Consider the following relation

A	B
a_1	b_1
a_1	b_1
a_2	b_1
a_2	b_1
a_3	b_2
a_1	b_1
a_3	b_2
a_2	b_1

It shows that A→B
because for each value/instance of A there is a particular value for attribute B. But the reverse is not true.

Example:

- ISBN → BKTITLE
- SSN → NAME ADDRESS DOB
- VIN → MAKE MODEL COLOR

The attribute(s) on the left hand side of a FD will be called as determinant. In A→B, A is the determinant.

➢ A key has two properties.

- Unique identification (can identify each tuple uniquely)
- Non-redundancy (Each attribute in the key is essential and sufficient).

When a key contains more than one attribute then that key is called a Composite Key. e.g. to identify a student (Class, Section, Rollno) is a key and can also be called as composite key.

➢ Rules of functional dependency (FD):

- Reflexivity rule:
  If α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.
- Union rule:
  If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds.
- Decomposition rule:
  If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds.
- Augmentation rule:
  If $\alpha \rightarrow \beta$ holds and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.
- Transitivity rule:
  If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \alpha$ holds, then $\alpha \rightarrow \gamma$ holds. e.g. Roll $\rightarrow$ Dept and Dept $\rightarrow$ HOD then Roll $\rightarrow$ HOD.
- Pseudo transitivity rule:
  If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds. e.g. Roll $\rightarrow$ Dept. and Dept Year $\rightarrow$ TIC then Roll Year $\rightarrow$ TIC. (TIC denotes Teacher in Charge).

➢ First Normal Form (1NF):

A relation will be in 1NF if there is no repeating groups at the intersection of each row and column. That means values at the intersection of each row and column is atomic.

Student ID	Student Name	Campus Address	Major	Course ID	Course Title	Instructor Name	Instructor Location	Grade
111	Anil	208 West	IS	IS 11 IS 22	DBMS SAD	S.B. A.R.	203 East 204 West	A B
222	Adhir	104 East	IT	IS 11 IT 22 IT 33	DBMS COA C	S.B. S.B. H.K.	203 East 203 East 209 South	C B A

Table for grade report

Against the student name Anil we have two repeating groups IS 11 and IS 22. So the above table is not in INF.

☞ So to avoid the repeating groups we use the following relation:

Student ID	Student Name	Campus Address	Major	Course ID	Course Title	Instructor Name	Instructor Location	Grade
111	Anil	208 West	IS	IS 11	DBMS	S.B.	203 East	A
111	Anil	208 West	IS	IS 22	SAD	A.R.	204 West	B
222	Adhir	104 East	IT	IS 11	DBMS	S.B.	203 East	C
222	Adhir	104 East	IT	IT 22	COA	S.B.	203 East	B
222	Adhir	104 East	IT	IT 33	C	H.K.	209 South	A

Grade report relation (INF)

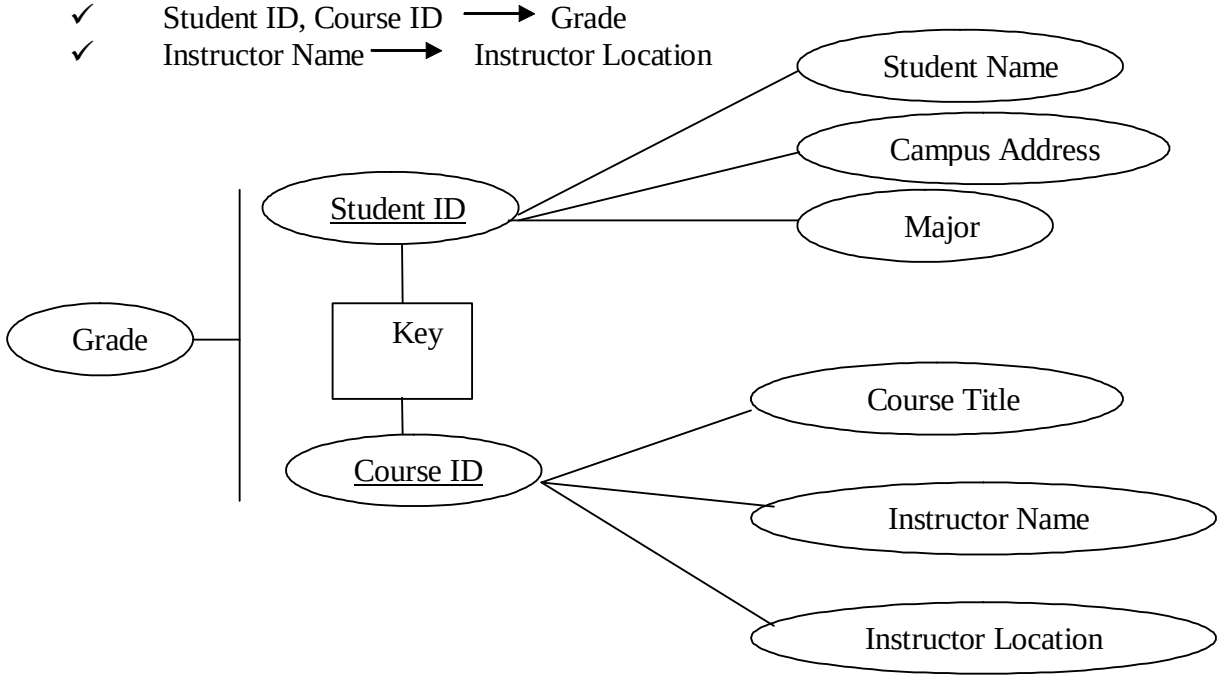
➢ Second Normal Form (2NF):

Relation is in 2NF if it is in 1NF and every non-key attribute is fully functionally dependent on the primary key. Thus no non key attribute is functionally dependent on part (but not all) of the primary key. That means no partial dependency exists.

In partial dependency in which one or more non-key attributes are functionally dependent on part but not all of the primary key.

☞ In the above-relation there are following four FDs.

- ✓ Student ID ⟶ Student Name, Campus Address, Major
- ✓ Course ID ⟶ Course Title, Instructor Name, Instructor Location.
- ✓ Student ID, Course ID ⟶ Grade
- ✓ Instructor Name ⟶ Instructor Location



☞ The above table is in INF but not in 2NF as partial dependency exists for the table. So also anomalies exist.

☞ So to resolve anomalies and put them in 2NF we split the table as shown below.

Student

Student ID	Student Name	Campus Address	Major
111	Anil	208 West	IS
222	Adhir	104 East	IT

It is also in 3NF

Course-Instructor

Course ID	Course Title	Instructor Name	Instructor Location
IS 11	DBMS	S.B.	203 East
IS 22	SAD	A.R.	204 West
IS 11	DBMS	S.B.	203 East
IT 22	COA	S.B.	203 East
IT 33	C	H.K.	209 South

Only in 2NF

Registration

Student ID	Course ID	Grade
111	IS 11	A
111	IS 22	B
222	IS 11	C
222	IS 22	B
222	IS 33	A

Also in 3NF

➢ Third Normal - Form (3NF):

A relation will be in 3NF if it is in 2NF and no transitive dependency exists. In transitive dependency a functional dependency exists between two or more non key attributes.

In the above tables only the 2nd table is not in 3NF and thus having anomalies (insertion, deletion and updation). So the table requires splitting into :

- Course (Course ID, Course Title, Instructor Name)
- Instructor (Instructor Name, Instructor Location).

☞ So ultimately up to 3NF total 4 tables are formed:

Student

Student ID	Student Name	Campus address	Major
111	Anil	208 West	IS
222	Adhir	104 East	IT

Course

Course ID	Course Title	Instructor Name
IS 11	DBMS	S.B.
IS 22	SAD	A.R.
IT 22	COA	S.B.
IT 33	C	H.K.

Instructor

Instructor Name	Instructor Location
S.B.	203 East
A.R.	204 West
H.K.	209 South

Registration

Student ID	Course ID	Grade
111	IS 11	A
111	IS 22	B
222	IS 11	C
222	IS 22	B
222	IS 33	A

➢ Boyce-codd Normal Form (BCNF):

A relation is in BCNF if every determinant is a candidate Key.

☞ The semantic rules for the following relation are:

1. Each student may major in several subjects.
2. For each major, a given student has only one advisor.
3. Each major has several advisors.
4. Each advisor advises only one major.
5. Each advisor advices several students in one major.

☞ There are two FDs in this relation:

- Student ID, Major ⟶ Advisor
- Advisor ⟶ Major

ST_MAJ_ADV

Student ID	Major	Advisor
123	Physics	Einstein
123	Music	Mozart
456	Bio	Darwin
789	Physics	Bohr
999	Physics	Einstein

☞ This relation is in 3NF but

still it has all anomalies, so

to bring it in BCNF we have the following splitting:

- St_Adv (Student ID, Advisor)
- Adv_Maj (Advisor, Major)

☞ So all anomalies are solved as they are now in BCNF.

➢ Fourth normal form (4NF):

A relation is in 4NF if it is in BCNF and contains no multivalued dependencies.

☞ A multivalued dependency is a type of dependency that exists where there are at least 3 attributes (e.g. A, B, C) in a relation. And for each value of A there is a well defined set of values for B and a well defined set of values for C, but the set of values of B is independent of set C.

☞ To demonstrate multivalued dependency consider the example:

OFFERING

Course	Instructor	Text book
Management	White Green Black	Drucker Peters
Finance	Gray	Weston Gilford

As in this table MVD exists so this table is suffering from all anomalies so to solve this splitting of table is required.

TEACHER

Course	Instructor
Management	White
Management	Green
Management	Black
Finance	Gray

TEXT

Course	Text book
Management	Drucker
Management	Peters
Finance	Weston
Finance	Gilford

Thus the above relations are in 4NF

➢ Fifth Normal Form (5NF):

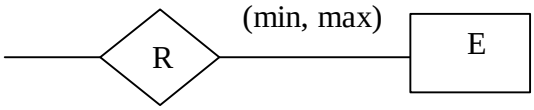
A relation is in 5NF if it is in 4NF and does not have a join dependency and the join should loss less.

A relation that has a join dependency cannot be divided into two (or more) relations such that the resulting tables can be recombined to form the original table.

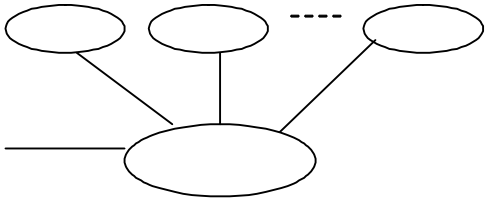
➢ Domain Key Normal Form (DKNF)

A relation is in DKNF if and only if every content of the relation is a logical consequence of key constraints and domain constraints.

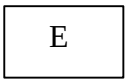
- A relation which is in DKNF is automatically in 5NF, 4NF and so on.
- DKNF is also called as PJNF (Project Join Normal Form).



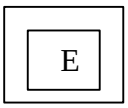
Structural constraint (min, max) on participation of E in R.



Composite attribute.

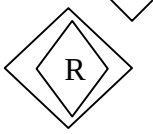


Entity set



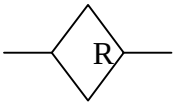
Weak entity set

Relationship set

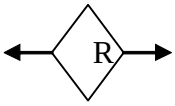


Identifying relationship set for weak entity set

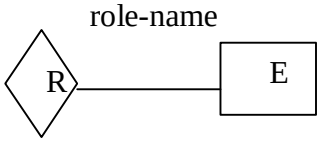
Primary key



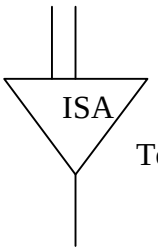
many-to-many relationship



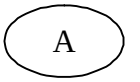
one-to-one relationship



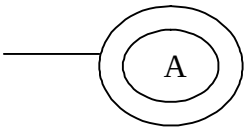
role indicator



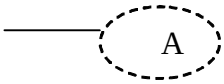
Total generalization



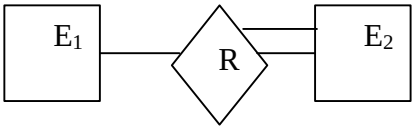
Attribute



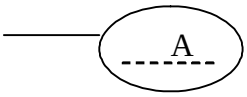
Multivalued attribute



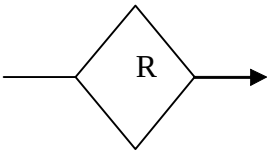
Derived attribute



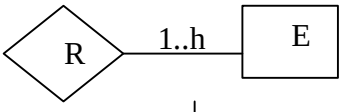
total participation of entity set in relationship



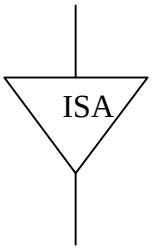
discriminating attribute of weak entity set



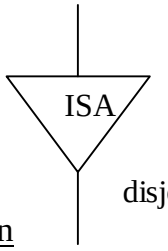
many-to-one relationship



Cardinality limits



ISA (specialization or generalization)



disjoint generalization

disjoint

Symbols used in the E-R notation

Codd's Rules for RDBMS

1 Codd's Rules for RDBMS

In 1985, **Edgar Codd** published a set of 13 rules which he defined as an evaluation scheme for a product which claimed to be a **Relational DBMS**.
Although these rules were later extended – and they now number in the 100s – they still serve as a foundation for a definition of a **Relational DBMS**.

1.1 The foundation rule

This is Codd's **Rule Zero** which is to foundation for the other rules.
The rules states that:

> **any system which claims to be a relation database management system must be able to manage databases entirely through its relational capabilities**

and means that the RDBMS must support:

- **A data definition language**
- **A data manipulation language**
- **A data integrity language**
- **A data control language**

all of which must work on several records at a time, that is, on a **relation**.

1.2 Information rule

All information in RDB must be represented explicitly at logical level in just one way, i.e., by **values** in **tables**.

1.3 Guaranteed access rule

Each item of data in a RDB must be guaranteed to be logically accessible by using a combination of **table name**, **primary key name**, and **primary key value**.

1.4 Systematic nulls rule

Null values are distinct from an empty character string, a string of blank characters, a zero or any other number, and must be supported by representing missing and/or inapplicable information in a systematic way.

1.5 Dynamic catalogue rule

The description of the DB must be represented at the logical level just like ordinary data. Authorized users must be able to use the data manipulation language to interrogate the DB in the same way that they interrogate ordinary data.

1.6 Comprehensive data sub-language rule

A RDB must support at least one language which is capable of supporting:

- **Data definition**
- **View definition**
- **Data manipulation**
  - **Interactively**
  - **By program in a conventional host language**
- **Integrity constraints**
- **Authorization**
- **Transaction boundaries**

1.7 View updating rule

All views of the data which are theoretically updatable must be updatable in practice by the DBMS.

1.8 High-level language rule

1.8 High-level language rule

The capability of handling a relation as a single operation applies to:

- **The retrieval of data**

and also:

- **The insertion of data**
- **The deletion of data**
- **The updating of data**

and these must be possible:

- **Interactively**
- **By program in a conventional host language**

1.9 Physical data independence rule

Application systems and terminal activities must be logically unimpaired whenever any changes are made to the way in which the physical data is stored, the storage structures which are used, or the access methods.

1.10 Logical data independence rule

Application systems and terminal activities must be logically unimpaired whenever any changes are made to the way in which the logical data is organized, such as when a table is dispersed to several tables to optimized storage or performance.

1.11 Integrity independence rule

Integrity constraints must be definable in the RDB sub-language and stored in the system catalogue and not within individual application programs.

1.12 Distribution independence rule

Application systems and terminal activities must be logically unimpaired whenever the data is redistributed amongst several locations on a data communications network.

1.13 Non-subversion rule

If the DB has any means of handling a single record at a time, that low-level of working must not be able to subvert or avoid the integrity rules which are expressed in a higher-level language which handles multiple records at a time

SQL Advantages and Disadvantages

SQL is both the data definition and data manipulation language of a number of relational database systems. SQL is based on tuple calculus. SQL resembles relational algebra in some places and tuple calculus in others. SQL supports the basic relational algebraic operations of Union, Difference, Cartesian product with the where clause indicating the joining attributes and the type of join.

SQL also provides for a wide variety of set of operators to allow the expression of relational calculus types of predicates and the testing of the membership of a tuple in a set. In addition, the use of aggregate functions provides SQL with additional data manipulation features not included in relational algebra or calculus. SQL is easier to use and more powerful than as data sublanguages than the ones used in DBMS based on the network and hierarchical models. However, these languages do not fully support some of the basic features of the relational data model; the concept of domains, entity and referential integrity and hence the concept of primary and foreign keys.

SQL is non-orthogonal i.e. concepts and constructs are not designed independently and can not be used consistently in uniform manner. The current SQL standard is viewed as one that tried to reconcile the various commercial implementations and came up with one that is, in effect, the lowest common denominator. An attempt is currently underway to upgrade the SQL standard.

2-tier Versus 3-tier Client/Server architecture

Features	2-tier	3-tier
1. System administration	Complex (More logic on the client to manage)	Less complex (Server manages the application)
2. Security	Low (data level security)	High (fine tuned at the service, method or object type level)
3. Encapsulation of data	Low (data tables are exposed)	High (the client invokes services or methods)
4. Performance	Poor (data sent through network and down loaded to client for analysis against SQL statements from client)	Good (only service requests and responses are sent between the client and server)
5. Scale	Poor (limited management of client communications link)	Excellent (can distribute loads across multiple servers)
6. Application reuse	Poor (monolithic application on client)	Excellent (can reuse services and objects)
7. Ease of development	High	Getting better (standard tools are used for both client and server side applications)
8. Server-to-server infrastructure	No	Yes (via server side middle ware)
9. Legacy application integration	No	Yes (via gateways encapsulated by services or objects)
10. Internet support	Poor (Internet bandwidth limitations make it harder to download fat clients)	Excellent (thin clients are downloaded and remote service invocations distribute the application load to the server)
11. Heterogeneous database support	No	Yes (3 tier applications can use multiple databases in a transaction)
12. Rich communication choices	No (only synchronous connection oriented RPC-like calls) **RPC => Remote Procedure Call*	Yes (supports RPC like calls, connectionless messaging, queued delivery, publish and subscribe, and broadcast)
13. Hardware architecture flexibility	Limited (we have a client and a server)	Excellent (3-tiers may reside on different, same computers, middle tier may involve multiple computers)
14. Availability	Poor	Excellent (can restart middle tier components on other servers)

Example :

Given R(A,B,C,D) with the functional dependencies F{A→B, A→C, C→D} are decomposed into R1(A,B,C) and R2(C,D). Check whether the decomposition is dependency-preserving and lossless or not.

	A	B	C	D
R1	α_A	α_B	α_C	β_{1D}
R2	β_{2A}	β_{2B}	α_C	α_D

	A	B	C	D
R1	α_A	α_B	α_C	α_D
R2	β_{2A}	β_{2B}	α_C	α_D

As after all updations, the first row contains all αs, so the decomposition is lossless.

Algorithm :

Algorithm to check if a Decomposition is Loss less
Input : A relation scheme R(A_1, A_2, A_3…, A_k), decomposed into the relation schemes R_1, R_2, R_3,…, R_i,…, R_n.
Output : Whether the decomposition is loss less or lossy.

(*A table, TABLE_LOSSY(1:n, 1:k) is used to test for the type of decomposition. Row i is for relation scheme R_i of the decomposed relation and column j is for attribute A_j in the original relation.*)
for each decomposed relation R_i do
if an attribute A_j is included in R_i,
then TABLE_LOSSY(i,j) := α_A (* place a symbol α_A in row i, column j *)
else TABLE_LOSSY(i,j) := β_{iA} (* place a symbol β_{iA} *)
change := true
while(change) do
for each FD X→Y in F do
if rows i and j exist such that the same symbol appears in each column corresponding to the attributes of X
then if one of the symbols in the Y column is α_r
then make the other α_r
else if the symbols are β_{pm} and β_{qm}
then make both of them, say, β_{pm}
else change := false
i := 1
lossy := true
while (lossy and i≤n) do
for each row i of TABLE_LOSSY
if all symbols are αs
then lossy := false
else i := i + 1 ;

Example

Given R(A,B,C,D,E) with the functional dependencies F {AB→CD, A→E, C→D}, the decomposition of R into R1 (A,B,C), R2(B,C,D) and R3(C,D,E) is lossy.

	A	B	C	D	E
R1	α_A	α_B	α_C	β_{1D}	β_{1E}
R2	β_{2A}	α_B	α_C	α_D	β_{2E}
R3	β_{3A}	β_{3B}	α_C	α_D	α_E

	A	B	C	D	E
R1	α_A	α_B	α_C	α_D	β_{1E}
R2	β_{2A}	α_B	α_C	α_D	β_{2E}
R3	β_{3A}	β_{3B}	α_C	α_D	α_E

As after all updations, no row exists with all αs, so the decomposition is lossy.

Algorithm : Lossless and Dependency-Preserving Third Normal Form Decomposition

Input : A relation scheme R, a set of canonical (minimal) functional dependencies F_c, and K, a candidate key of R.

Output : A collection of third normal form relation schemes (R_1, R_2, …, R_n) that are dependency-preserving and loss less

```
i := 0
Find all the attributes in R that are not involved in any FDs in Fc either on the left or right side.
If any such attributes {A} are found then
 begin
  i : = i + 1;
  Form a relation Ri{A};            (* involving attributes not in any FDs *)
  R := R – {A};                     (* remove the attributes {A} from R *)
 end
 If there is a dependency X→Y in Fc such that all the attributes that remain
 in R are included in it
 then
    begin
     i := i + 1;
     output R as Ri{X,Y};
   end
 else
    begin
      for each FD X→A in Fc do
        begin
          i := i + 1;
          Form Ri <{X,A}, F{X→A}>
        end;
      combine all relation schemes corresponding to FDs with the same LHS
       (* i.e., <(X,A), {X→A}> and <(X,B), {X→B}>
              could be replaced by <(X,AB), {X→AB}> *)
      if none of left side of FD in
       Fj for 1 ≤ j ≤ i satisfies K ⊆ X
         then begin
           i := i + 1;
           form Ri<{K}>;            (* make sure that a relation contains the candidate key of R *)
        end;
  end;
```

Example :

Find a loss less join and dependency–preserving decomposition of the following relation and given FDs :

SHIPPING(Ship,Capacity,Date,Cargo,Value)

with Ship→Capacity, ShipDate→Cargo, CargoCapacity→Value

The FDs are simple since each has a single attribute on the RHS and they are not redundant. No FD contains extraneous attributes on the LHS. Hence they are in canonical form. A candidate key of the relation is Ship Date. As { Ship Date }$^+$ contains all attributes.

There is no attribute not appearing in any FD. There is no single FD containing all remaining attributes in SHIPPING, so we proceed to form a relation for each FD in the canonical cover.

R_1 (Ship, Capacity) with the FD Ship→Capacity

R_2 (Ship, Date, Cargo) with the FD Ship Date→Cargo

R_3 (Cargo, Capacity, Value) with the FD Cargo Capacity→Value

As a candidate key is included in the determinant of the FD of the decomposed relation scheme R2, we need not include another relation scheme with only a candidate key. The decomposition of SHIPPING into R_1, R_2, and R_3 is both loss less and dependency preserving.

Example :

Consider the relation scheme STUDENT_INFO(Student(S), Major(M), Student_Department(S_d), Advisor(A), Course(C), Course_Department(C_d), Grade(G), Professor(P), Prof_Department(P_d), Room(R), Day(D), Time(T)) with the following functional dependencies:

S→M	**each student is in a unique major**
S→A	**each student has a unique advisor**
M→S_d	**each major is offered in an unique department**
S→S_d	**each student is in one department**
A→S_d	**each advisor is in an unique department**
C→C_d	**each course is offered by a single department**
C→P	**each course is taught by one professor**
P→P_d	**each professor is in an unique department**
RTD→C	**each room has on a given day and time only one course scheduled in it**
RTD→P	**each room has on a given day and time one professor teaching it it**
TPD→R	**a given professor on a given day and time is in one room**
TSD→R	**a given student on a given day and time is in one room**
TDC→R	**a course can be in only one room on a given day and time**
TPD→C	**on a given day and time a professor can be teaching only one course**
TSD→C	**on a given day and time a student can be attending only one course**
SC→G	**each student in a given course has a unique grade**

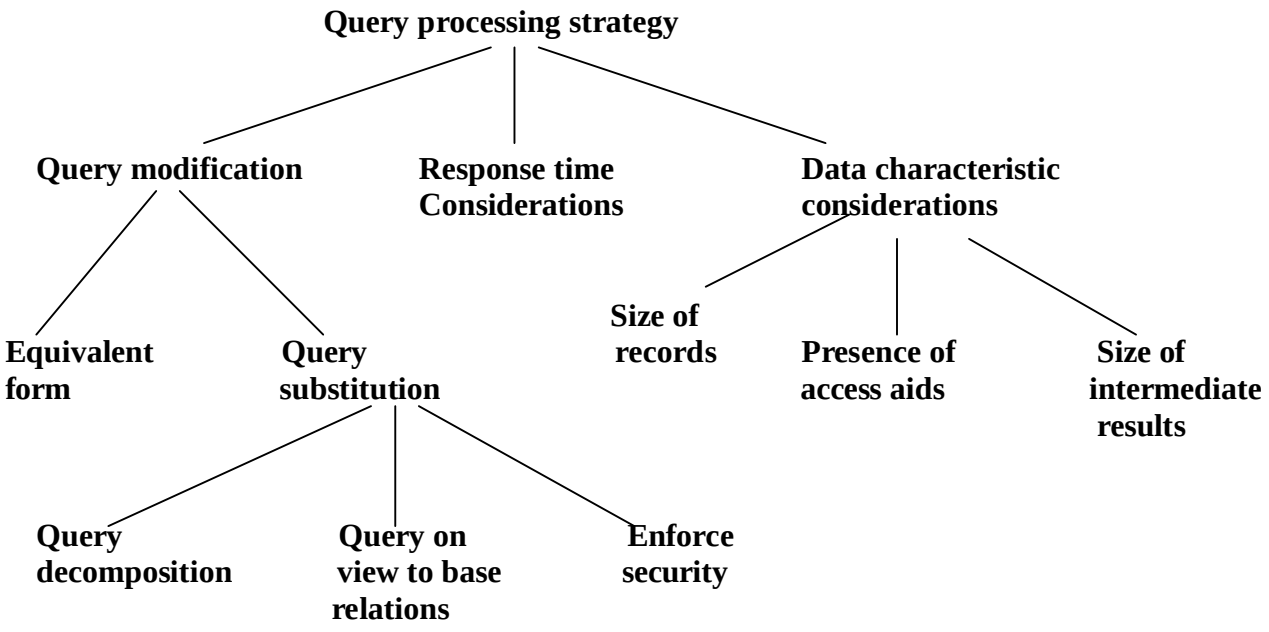
A canonical cover of this set of functional dependencies will not contain the dependencies {S→S_d as S→M and M→S_d, RTD→P as RTD→C and C→P, TDC→R as RTD→C and TDC→R follows RTD→R which is obvious, TPD→C as TPD→R and RTD→C, TSD→C as TSD→R and RTD→C follows TSD→C}. The key is TSD. Thus the decomposition in 3NF results :

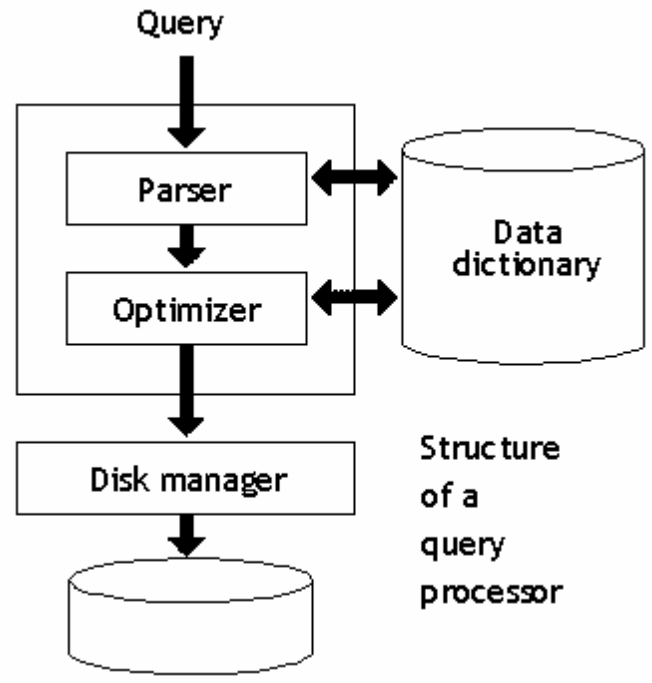
R1(SMA)	**with the FD S→MA**	**R6(RTDC)**	**with the FD RTD→C**
R2(MS_d)	**with the FD M→S_d**	**R7(TPDR)**	**with the FD TPD→R**
R3(AS_d)	**with the FD A→S_d**	**R8(TSDR)**	**with the FD TSD→R**
R4(CC_dP)	**with the FD C→C_dP**	**R9(SCG)**	**with the FD SC→G**
R5(PP_d)	**with the FD P→P_d**		

(Note: Since all the attributes in the original relation scheme are involved with some FD we do not have to create a relation scheme with attributes not so involved. Also, the relation scheme R8 includes a candidate key; consequently we don't need to create an explicit relation scheme for the key.)

R1 through R9 form a loss less and dependency-preserving decomposition of STUDENT_INFO.

Query Processing Strategies





Query processing (also called query optimizing) is the procedure of selecting the best plan or strategy to be used in responding to a database request. The plan is then executed to generate a response. Query processor is responsible to generate this strategy. Parser in query processor verifies and checks the syntax and feasibility of the query. Then access plans are generated for the query transformed into relational algebraic expressions. Query modification is called for when the query is based on a view. Such queries have to be replaced by appropriate queries on the base relations.

The aims of query processing are to transform a query written in a high level language, typically in SQL, into a correct and efficient execution strategy expressed in a low level language (implementing the relational algebra), and to execute the strategy to retrieve the required data.

Example :

We concentrate on the database consisting of the following four relations:

STUDENT (Std#, Std_Name) : *student details (40,000 tuples)*
REGISTRATION (Std#, Course#) : *courses the students are currently registered in (4,00,000 tuples)*
GRADE (Std#, Course#, Grade) : *grades in courses already completed by a student (6,00,000 tuples)*
COURSE (Course#, Course_Name, Instructor) : *course details (5,000 tuples)*
(Let us assume that there are 500 courses whose course number is higher than COMP300.)
Consider the following query:

"List the names of courses higher than COMP300 and all students registered in them."

The following are some different ways of stating this query in SQL and relational algebra.

Category No : 1

In SQL:

```
Select Std_Name, Course_Name
From STUDENT, REGISTRATION, COURSE
Where STUDENT.Std# = REGISTRATION.Std# and
        COURSE.Course# = REGISTRATION.Course# and
        REGISTRATION.Course# > COMP300
```

In Relational Algebra :

$\Pi_{Std\_Name,\ Course\_Name}(\sigma_{Course\# > COMP300}(STUDENT \underset{Std\#}{\triangleright\triangleleft} REGISTRATION \underset{Course\#}{\triangleright\triangleleft} COURSE))$

Operation	Processing cost if relations		Estimated size of result
	not sorted	sorted	
Join of STUDENT And REGISTRATION	40,000 * 4,00,000	40,000 + 4,00,000	4,00,000 tuples
Join of this result With COURSE	5,000 * 4,00,000	5,000 + 4,00,000	4,00,000 tuples
Selection from result Of course# > COMP300	4,00,000	4,00,000	40,000 tuples
Projection on Std_Name, Course_Name	40,000	40,000	40,000 tuples

Category No : 2

In SQL:

```
Select Std_Name, C1.Course_Name
From STUDENT, REGISTRATION, COURSE Cl
Where STUDENT.Std# = REGISTRATION.Std#
   And REGISTRATION.Course#  in
        (Select C2.Course#
         From COURSE c2
         Where C2.Course# > COMP300 and
                C1.Course# = C2.Course#)
```

In Relational Algebra :

$\Pi_{Std\_Name,\ Course\_Name}(STUDENT \underset{Std\#}{\triangleright\triangleleft} (\sigma_{Course\# > COMP300}(REGISTRATION \underset{Course\#}{\triangleright\triangleleft} COURSE))$

Operation	Processing cost if relations		Estimated size of Result
	Not sorted	Sorted	
Join of REGISTRATION and COURSE	5,000 * 4,00,000	5,000 + 4,00,000	4,00,000 tuples
Selection from result of Course# > COMP300	4,00,000	4,00,000	40,000 tuples
Join of STUDENT and result above	40,000 * 40,000	40,000 + 40,000	40,000 tuples
Projection on Std_Name, Course_Name	40,000	40,000	40,000 tuples

STUDENT (Std#, Std_Name) : ***student details (40,000 tuples)***
REGISTRATION (Std#, Course#) : ***courses the students are currently registered in (4,00,000 tuples)***
GRADE (Std#, Course#, Grade) : ***grades in courses already completed by a student (6,00,000 tuples)***
COURSE (Course#, Course_Name, Instructor) : ***course details (5,000 tuples)***

(Let us assume that there are 500 courses whose course number is higher than COMP300.)

Consider the following query:

"List the names of courses higher than COMP300 and all students registered in them."

Category No : 3

In SQL:

```
Select Std_Name, Cl.Course_Name
From STUDENT, COURSE Cl
Where STUDENT.Std# in
  (Select REGISTRATION.Std#
   From REGISTRATION
  Where REGISTRATION. Course# in
    (Select C2.Course #
     From COURSE C2
    Where C2.Course# > COMP300 and
      C1.Course# = C2.Course#))
```

In Relational Algebra :

$$\Pi_{\text{Std\_Name, Course\_Name}}(\text{STUDENT} \underset{\text{Std\#}}{\triangleright\triangleleft} (\sigma_{\text{Course\# > COMP300}}(\text{REGISTRATION}) \underset{\text{Course\#}}{\triangleright\triangleleft} (\sigma_{\text{Course\# > COMP300}}\text{COURSE})$$

Operation	Processing cost if relations		Estimated size of Result
	Not sorted	Sorted	
Selection from COURSE Course# > COMP300	5,000	5,000	500 tuples
Selection from REGISTRATION Course# > COMP300	4,00,000	4,00,000	40,000 tuples
Join of selected tuples from COURSE and REGISTRATION	500 * 40,000	500 + 40,000	40,000 tuples
Join of STUDENT with result above	40,000 * 40,000	40,000 + 40,000	40,000 tuples
Projection on Std_Name, Course _Name	40,000	40,000	40,000 tuples

The above illustrates a considerable processing (and I/O cost) reduction when one from of the query is used as opposed to another equivalent one. This indicates that some from of query processing is necessary if the DBMS is to provide an acceptable response. The intent of the query processor is to find a more efficient form of a user-supplied query expression. A query can be improved in a number of ways before its evaluation is performed. The improvements are basically concerned with minimizing, if not altogether removing, redundancy from expressions and results. This in turn involves simplifying and transforming the query, taking into account the characteristics of the data contained in the database. For example, relations may be supported by some access aid on certain attributes. Such access aids could be in the form of an index using a B+ -tree, ISAM, or a hash. Furthermore, the tuples of the relation may be stored in some particular order to aid their retrieval. The system must exploit these access aids and storage schemes to come up with an optional access plan.

Query Representation :

Internally a query will be represented by relational calculus, relational algebra, object graph or by mostly using operator graph or tableau.

$\Pi_{Std\_Name}(\sigma_{Course\_Name = 'Database'}(STUDENT \underset{Std\#}{\triangleright\!\triangleleft} REGISTRATION \underset{Course\#}{\triangleright\!\triangleleft} COURSE))$

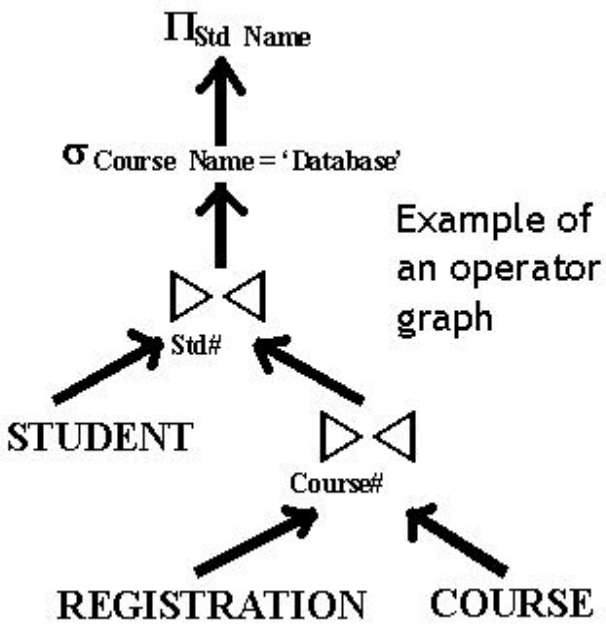
An operator graph depicts how a sequence of operations can be performed. Here operations are depicted by nodes and directed edges denote flow of data. The relational algebra expression for the query "List the names of the students registered in the Database course" is

$\Pi_{Std\_Name}(\sigma_{Course\_Name = 'Database'}(STUDENT \underset{Std\#}{\triangleright\!\triangleleft} REGISTRATION \underset{Course\#}{\triangleright\!\triangleleft} COURSE))$

The corresponding operator graph is shown below :

Steps in Query Processing :

- Convert to a standard starting point : The query is of the form $p_1 \vee p_2 \vee \ldots$ (conjunctive normal form), where each p_i consists of $t_{11} \wedge t_{12} \wedge \ldots$
- Transform the query : It is to enhance the performance.
- Simplify the query : Remove all redundant useless operations.
- Prepare alternate access plans : Optimal access plan is chosen.



General Processing Strategies :

- Perform selection as early as possible : As it reduces the cardinality and subsequent processing time.
- Combine a number of unary operations : As example consider following example
  $\sigma_{C1}(\sigma_{C2}(R)) \equiv \sigma_{C1 \wedge C2}(R)$ $\quad \Pi_X(\Pi_Y(R)) \equiv \Pi_{X \cap Y}(R)$
  If $X \subseteq Y$, then $\Pi_X(\Pi_Y(R)) \equiv \Pi_X(R)$
- Convert cartesian product with certain subsequent selection into a join : $\sigma_{R.A = S.B} R \times S \equiv R \underset{A \theta B}{\triangleright\!\triangleleft} S$
- Compute common expression once : A common expression that appears more than once should be computed once and then stored, and then reused.
- Preprocess the relations : It includes sorting and index creation on join attributes

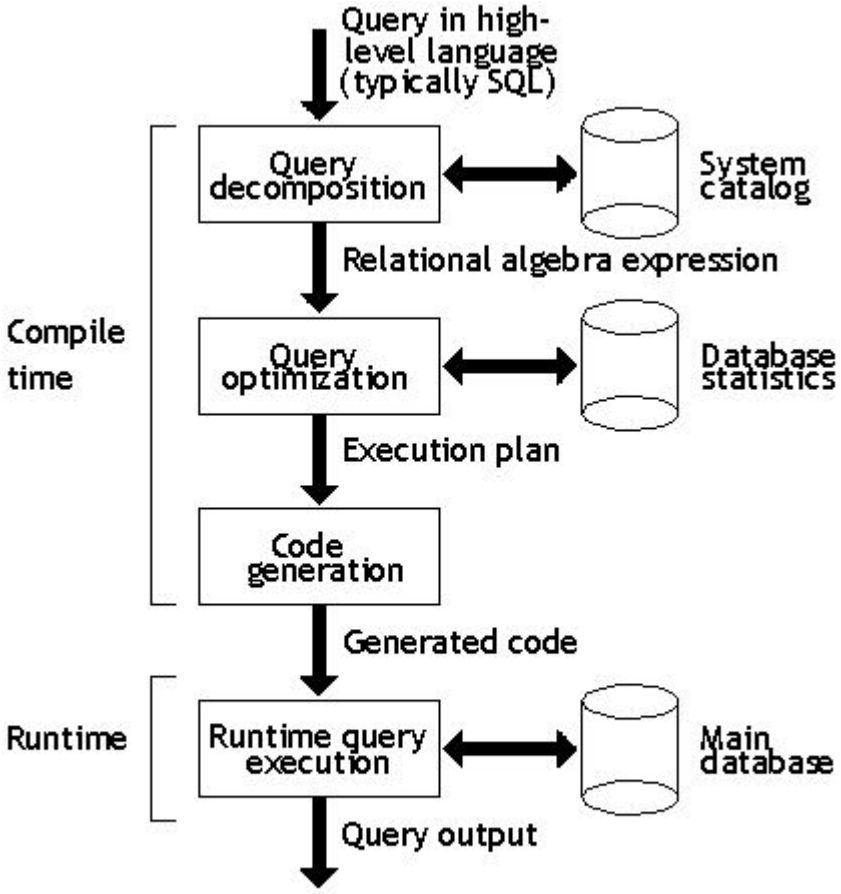
Transformation into an equivalent Expression :

- Commutative law
  $R \cup S \equiv S \cup R$
  $R \cap S \equiv S \cap R$
  $R \underset{}{\triangleright\!\triangleleft} S \equiv S \underset{}{\triangleright\!\triangleleft} R$
  $R * S \equiv S * R$
- Associative law
  $R \underset{}{\triangleright\!\triangleleft} (S \underset{}{\triangleright\!\triangleleft} T) \equiv (R \underset{}{\triangleright\!\triangleleft} S) \underset{}{\triangleright\!\triangleleft} T$
  $R * (S * T) \equiv (R * S) * T$
- Idempotent law
  $R \cup R \equiv R$
  $R \cap R \equiv R$
- Other laws
  $R \underset{}{\triangleright\!\triangleleft} R \equiv R$
  $R - R \equiv \Phi$
  $R \cup \Phi \equiv R$
  $R \cap \Phi \equiv \Phi$
  $R \underset{}{\triangleright\!\triangleleft} \Phi \equiv \Phi$
  $R - \Phi \equiv R$
  $\Phi - R \equiv \Phi$

Dynamic versus Static Optimization :

In dynamic query optimization in the runtime it is done. And main advantage is that all information required is up to date. And the disadvantage is that each time the query is to be parsed, validated and optimized before execution.

In static query optimization the query is parsed, validated and optimized once. So runtime overhead is removed. The disadvantage is that the query, which was optimized once, may not remain when in run later.



Phases of query processing

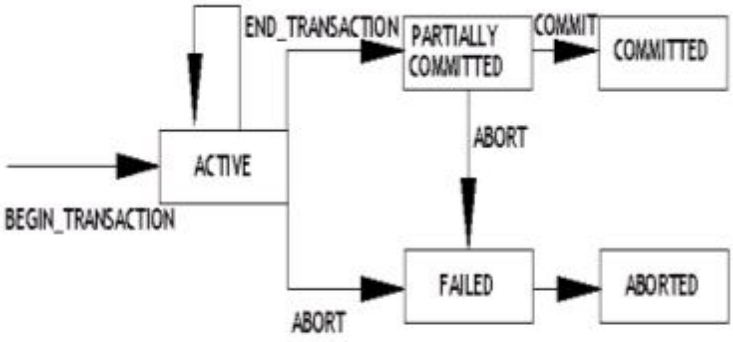
TRANSACTION MANAGEMENT AND CONCURRENCY CONTROL

To keep DBMS in reliable and consistent states so that it can provide three services :

- **Transaction support**
- **Concurrency control services**
- **Recovery services.**

If concurrent operations are not controlled then accesses may interfere to make inconsistent database. Hence we require concurrency control to solve it. Database recovery is the process of restoring the database to a correct state following a failure (e.g. H/W system crash, S/W error in application, logical error in program etc.)

Transaction : An action or a series of actions, carried out by a single user or application program, which reads or updates the contents of the database. A transaction is a logical unit of work. It may an entire or part of a program or a command (SQL for INSERT or UPDATE) and it may involve a large number of operations on a database. A transaction is said to be committed and the database reaches to a new consistent state. It may also be rolled back or undone. A compensating transaction can reverse the effect of a transaction.



State transition diagram for a transaction

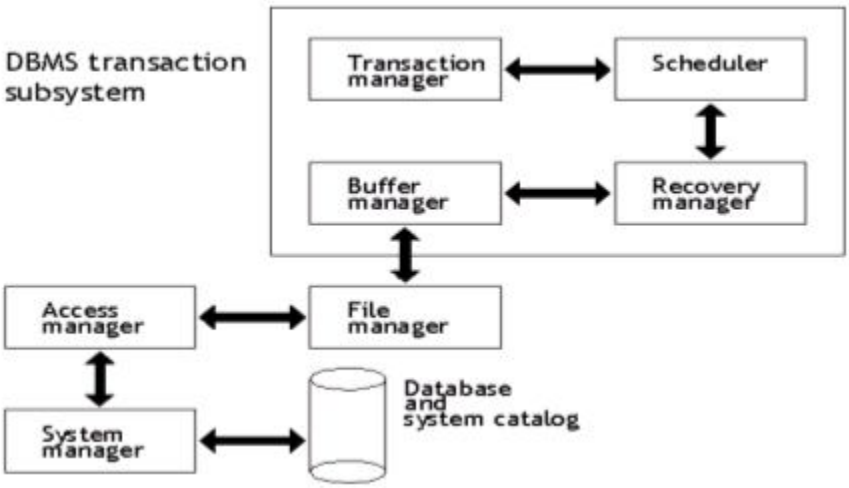
PARTIALLY_COMMITTED : A transaction is partially committed if it has violated serializability or integrity constraint or secondary storage failure for record updation, then the transaction has to be aborted. If the transaction has been successful, any updates can be safely recorded and the transaction can go to the COMMITTED state.

FAILED : It occurs if the transaction can not be committed or the transaction is aborted while in the ACTIVE state, perhaps due to the user aborting the transaction or as a result of the concurrency control protocol aborting the transaction to ensure the serializability.

Properties of Transaction

There are four acid properties of transaction.

- **Atomicity : A transaction is an indivisible unit. The recovery subsystem is responsible for atomicity.**
- **Consistency : It is the responsibility of the DBMS and application developer so that each transaction must transform the database from one consistent state to another consistent state.**
- **Isolation : It is the responsibility of the concurrency control subsystem to ensure isolation where each transaction can execute independently so that partial effects of each transaction should not be visible to other transactions.**
- **Durability : It is the responsibility of the recovery manager to record permanently the effects of a successfully completed transaction. And must not be lost due to subsequent failure.**



Database Architecture

The transaction manager coordinates transactions on behalf of the application programs. It communicates with scheduler (also called lock manager if it uses locking protocol), the module responsible for implementing strategy for concurrency control. Recovery manager manages failure and ensures the database is restored to the initial state same as prior to transaction. The buffer manager is responsible for the transfer of data between disk storage and main memory.

Concurrency Control

The concurrency control is a process of managing simultaneous operations on the database without having them interfere with one another. A transaction continues until it reaches to some I/O operations. Then another transaction begins in interleaved fashion enhancing overall CPU utilization.

There are three potential problems of concurrency:

- **Lost update problem: Update operation of one user over written by other user.**

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	read(bal_x)	100
t_3	read(bal_x)	$bal_x = bal_x + 100$	100
t_4	$bal_x = bal_x - 10$	write(bal_x)	200
t_5	write(bal_x)	commit	90
t_6	commit		90

- **Uncommitted dependency (or dirty read) problem: When a transaction uses result of a partially completed transaction.**

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2		read(bal_x)	100
t_3		$bal_x = bal_x + 100$	100
t_4	begin_transaction	write(bal_x)	200
t_5	read(bal_x)	……	200
t_6	$bal_x = bal_x - 10$	rollback	100
t_7	write(bal_x)		190
t_8	commit		190

- **Inconsistent analysis problem: When a transaction reads several values from the database but a second transaction updates some of them during the execution of the first.**

Time	T_1	T_2	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	read(bal_x)	read(bal_x)	100	50	25	0
t_4	$bal_x = bal_x - 10$	sum = sum + bal_x	100	50	25	100
t_5	write(bal_x)	read(bal_y)	90	50	25	100
t_6	read(bal_z)	sum = sum + bal_y	90	50	25	150
t_7	$bal_z = bal_z + 10$	…..	90	50	25	150
t_8	write(bal_z)	…..	90	50	35	150
t_9	commit	read(bal_z)	90	50	35	150
t_{10}		sum = sum + bal_z	90	50	35	185
t_{11}		commit	90	50	35	185

Another problem can occur when a transaction T rereads a data item it has previously read but, in between, another transaction has modified it. Thus, T receives two different values for the same data item. This is nonrepeatable (or fuzzy) read. A similar problem can occur if transaction T executes a query that retrieves a set of tuples from a relation satisfying a certain predicate, re-executes the query at a later time but finds that the retrieved set contains an additional (phantom) tuple that has been inserted by another transaction in the meantime. This is sometimes referred to as phantom read.

Serializability and Recoverability

Schedule: A sequence of operations by a set of concurrent transactions that preserves the order of the operations in each of the individual transactions.

Serial schedule: A schedule where the operations of each transaction are executed consecutively without any interleaved operations from other transactions.

Non-serial schedule: A schedule where the operations from a set of concurrent transactions are interleaved.

Serializable schedule: The objective of serializability is to find non-serial schedules that allow transactions to execute concurrently with out interfering one another, and thereby produce a database state that could be produced by a serial execution. We say that a non-serial schedule is serializable and correct if it produces the same results as some serial execution.

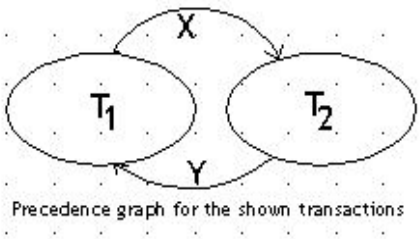
- If two transactions perform read then they are not conflicting and they do not require any ordering.
- If two transactions perform read or write on two separate data items then they are not conflicting and they do not require any ordering.
- If one transaction performs write and other performs read or write on same data item then their order of execution is very important.

Time	T_1	T_2	T_1	T_2	T_1	T_2
t_1	begin_tran		begin_tran		begin_tran	
t_2	read(bal_x)		read(bal_x)		read(bal_x)	
t_3	write(bal_x)		write(bal_x)		write(bal_x)	
t_4		begin_tran		begin_tran	read(bal_y)	
t_5		read(bal_x)		read(bal_x)	write(bal_y)	
t_6		write(bal_x)	read(bal_y)		commit	
t_7	read(bal_y)			write(bal_x)		begin_tran
t_8	write(bal_y)		write(bal_y)			read(bal_x)
t_9	commit		commit			write(bal_x)
t_{10}		read(bal_y)		read(bal_y)		read(bal_y)
t_{11}		write(bal_y)		write(bal_y)		write(bal_y)
t_{12}		commit		commit		commit
	Non-serial schedule S1		Non-serial schedule S2 same as S1		Serial schedule S3 same as S1, S2	

Whether a schedule is conflict serializable or not can be tested by constrained write rule (i.e. a transaction updates a data item based on its old value, which is first read by the transaction) or by a precedence (or serialization) graph. This is a directed graph G = (N,E) constructed with following rules:

- Create a node for each transaction.
- Create a directed edge $T_i \rightarrow T_j$, if T_j reads the value of an item written by T_i.
- Create a directed edge $T_i \rightarrow T_j$, if T_j writes a value into an item after it has been read by T_i.
- Create a directed edge $T_i \rightarrow T_j$, if T_j writes a value into an item after it has been written by T_i.

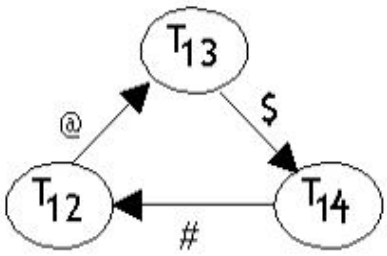
Time	T_1	T_2
t_1	begin_tran	
t_2	read(bal_x)	
t_3	$bal_x=bal_x+100$	
t_4	write(bal_x)	begin_tran
t_5		read(bal_x)
t_6		$bal_x=bal_x*1.1$
t_7		write(bal_x)
t_8		read(bal_y)
t_9		$bal_y=bal_y*1.1$
t_{10}		write(bal_y)
t_{11}	read(bal_y)	commit
t_{12}	$bal_y=bal_y-100$	
t_{13}	write(bal_y)	
t_{14}	commit	
Two concurrent update transactions		



Precedence graph for the shown transactions

Time	Schedule	T_{12}	T_{13}	T_{14}
t_1	Read(A)	Read(A)		
t_2	Read(B)		Read(B)	
t_3	A := f_1(A)	A := f_1(A)		
t_4	Read(C)			Read(C)
t_5	B := f_2(B)		B := f_2(B)	
t_6	Write(B)		Write(B) $	
t_7	C := f_3(C)			C := f_3(C)
t_8	Write(C)			Write(C) #
t_9	Write(A)	Write(A)@		
t_{10}	Read(B)			Read(B) $
t_{11}	Read(A)		Read(A) @	
t_{12}	A := f_4(A)		A := f_4(A)	
t_{13}	Read(C)	Read(C) #		
t_{14}	Write(A)		Write(A)	
t_{15}	C := f_5(C)	C := f_5(C)		
t_{16}[	Write(C)	Write(C)		
t_{17}	B := f_6(B)			B := f_6(B)
t_{18}	Write(B)			Write(B)

An execution schedule involving three transactions



T_i WRITE (X)

↓ Rule No : 2

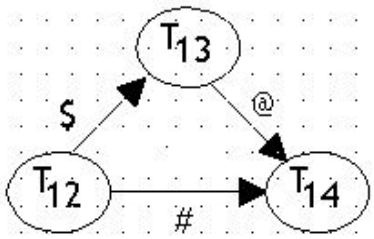
T_j (earliest) READ (X)

T_i READ (X) or WRITE(X)

↓ Rules No : 3 and 4

T_j WRITE (X)

Time	Schedule	T_{12}	T_{13}	T_{14}
t_1	Read(A)	Read(A)		
t_2	A := f_1(A)	A := f_1(A)		
t_3	Read(C)	Read(C)		
t_4	Write(A)	Write(A) $		
t_5	A := f_2(C)	A := f_2(C)		
t_6	Read(B)		Read(B)	
t_7	Write(C)	Write(C) #		
t_8	Read(A)		Read(A) $	
t_9	Read(C)			Read(C) #
t_{10}	B := f_3(B)		B := f_3(B)	
t_{11}	Write(B)		Write(B) @	
t_{12}	C := f_4(C)			C := f_4(C)
t_{13}	Read(B)			Read(B) @
t_{14}	Write(C)			Write(C)
t_{15}	A := f_5(A)		A := f_5(A)	
t_{16}[	Write(A)		Write(A)	
t_{17}	B := f_6(B)			B := f_6(B)
t_{18}	Write(B)			Write(B)



A precedence graph is acyclic if it does not contain any cycle. A precedence graph is serializable if it is acyclic. If we find a cycle in the graph then the schedule is non-serializable. So the first example is non-serializable as it contains a cycle and the second one is serializable as it is acyclic.

Recoverable Schedule

A schedule where, for each pair of transactions T_i and T_j, if T_i reads a data item previously written by T_j, then the commit operation of T_i precedes the commit operation of T_j.

Locking, time-stamping (pessimistic and conservative) and optimistic are the techniques for concurrency control.

Granularity of Data items

The size of data items chosen as the unit of protection by a concurrency control protocol. Granularity can range form small to large data items e.g. the entire database, a file, a page (sometimes called an area or database space – a section of physical disk in which relations are stored), a record or a field value of a record. This size has a very significant effect in the designing of the concurrency algorithms.

Locking Methods

Locking: A procedure used to control concurrent access to data. When one transaction is accessing the database, a lock may deny access to other transactions to prevent incorrect results. Locks may be of two types: 1> Shared lock and 2> Exclusive lock

Shared lock: If a transaction has a shared lock on a data item, it can read the item but not update it.

Exclusive lock: If a transaction has an exclusive lock on a data item, it can both read and update the item.

Locks are used in the following way:

- **Any transaction that needs to access a data item must first lock the item, requesting a shared lock for read only access or an exclusive lock for both read and write access.**
- **If the item is not already locked by another transaction, the lock will be granted.**
- **If the item is currently locked, the DBMS determines whether the request is compatible with the existing lock. If a shared lock is requested on an item that already has a shared lock on it, the request will be granted; otherwise, the transaction must wait until the existing lock is released.**
- **A transaction continues to hold a lock until it explicitly releases it either during execution or when it terminates (aborts or commits). It is only when the exclusive lock has been released that the effects of the write operation will be made visible to other transactions.**
- **Due to some needs a system may support upgrade of locks from shared to exclusive lock. Or in certain cases downgrade of locks from exclusive to shared lock.**

To guarantee serializability, we must follow an additional protocol concerning the positioning of the lock and unlock operations in every transaction. The best-known protocol is two-phase locking (2PL).

Two-Phase Locking (2PL)

According the rules of this protocol, every transaction can be divided into two phases: first a growing phase, in which it acquires all the locks needed but can not release any locks, and then a shrinking phase, in which it releases its locks but can not acquire any new locks. The rules are:

- **A transaction must acquire a lock on an item before operating on the item. The lock may be read or write, depending on the type of access needed.**
- **Until the transaction releases a lock, it can never acquire any new locks.**

Preventing the Lost Update Problem using 2PL

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	write_lock(bal_x)	100
t_3	write_lock(bal_x)	read(bal_x)	100
t_4	WAIT	$bal_x = bal_x + 100$	100
t_5	WAIT	write(bal_x)	200
t_6	WAIT	commit/unlock(bal_x)	200
t_7	read(bal_x)		200
t_8	$bal_x = bal_x - 10$		200
t_9	write(bal_x)		190
t_{10}	commit/unlock(bal_x)		190

Preventing the Uncommitted Dependency Problem using 2PL

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2		write_lock(bal_x)	100
t_3		read(bal_x)	100
t_4	begin_transaction	$bal_x = bal_x + 100$	100
t_5	write_lock(bal_x)	write(bal_x)	200
t_6	WAIT	rollback/unlock(bal_x)	100
t_7	read(bal_x)		100
t_8	$bal_x = bal_x - 10$		100
t_9	write(bal_x)		90
t_{10}	commit/unlock(bal_x)		90

Preventing the Inconsistent Analysis Problem using 2PL

Time	T_1	T_2	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	write_lock(bal_x)		100	50	25	0
t_4	read(bal_x)	read_lock(bal_x)	100	50	25	0
t_5	$bal_x = bal_x - 10$	WAIT	100	50	25	0
t_6	write(bal_x)	WAIT	90	50	25	0
t_7	write_lock(bal_z)	WAIT	90	50	25	0
t_8	read(bal_z)	WAIT	90	50	25	0
t_9	$bal_z = bal_z + 10$	WAIT	90	50	25	0
t_{10}	write(bal_z)	WAIT	90	50	35	0
t_{11}	commit/unlock(bal_x, bal_z)	WAIT	90	50	35	0
t_{12}		read(bal_x)	90	50	35	0
t_{13}		sum = sum + bal_x	90	50	35	90
t_{14}		read_lock(bal_y)	90	50	35	90
t_{15}		read(bal_y)	90	50	35	90
t_{16}		sum = sum + bal_y	90	50	35	140
t_{17}		read_lock(bal_z)	90	50	35	140
t_{18}		read(bal_z)	90	50	35	140
t_{19}		sum = sum + bal_z	90	50	35	175
t_{20}		commit/unlock(bal_x, bal_y, bal_z)	90	50	35	175

Cascading Rollback Problem using 2PL

In a situation where a single transaction leads to a series of rollbacks, is called cascading rollback.

Time	T_1	T_2	T_3
t_1	begin_transaction		
t_2	write_lock(bal_x)		
t_3	read(bal_x)		
t_4	write_lock(bal_y)		
t_5	read(bal_y)		
t_6	$bal_x = bal_x + bal_y$		
t_7	write(bal_x)		
t_8	unlock(bal_x)	begin_transaction	
t_9	....	write_lock(bal_x)	
t_{10}	....	read(bal_x)	
t_{11}	....	$bal_x = bal_x + 100$	
t_{12}	....	write(bal_x)	
t_{13}	....	unlock(bal_x)	
t_{14}	....	....	
t_{15}	rollback	....	
t_{16}		....	
t_{17}		....	begin_transaction
t_{18}		....	read_lock(bal_x)
t_{19}		rollback	....
t_{20}			rollback

In a situation where T_2 can would get its exclusive lock until after T_1 has performed its rollback is called rigorous 2PL. In strict 2PL a transaction can only hold exclusive locks until the end of the transaction.

Latches: **It is another type of lock, which is held for a shorter duration than a normal lock. A latch can be used before a page is read from and written to, disk to ensure that the operation is atomic. As the latch is simply to prevent conflict for this type of access, latches do not need to conform to the normal concurrency control protocol such as 2PL.**

Time	T_1	T_2
t_1	begin_transaction	
t_2	write_lock(bal_x)	begin_transaction
t_3	read(bal_x)	write_lock(bal_y)
t_4	$bal_x = bal_x - 10$	read(bal_y)
t_5	write(bal_x)	$bal_y = bal_y - 10$
t_6	write_lock(bal_y)	write(bal_y)
t_7	WAIT	write_lock(bal_x)
t_8	WAIT	WAIT
t_9	WAIT	WAIT
t_{10}	WAIT	WAIT
t_{11}	....	....

Deadlock and Livelock

Deadlock is a situation that may result when two (or more) transactions are each waiting for locks to be released that are held by the other. In deadlock transactions can wait for locks on data items. In live lock a transaction is left in a wait state indefinitely, unable to acquire any new locks, although the DBMS is not in deadlock. To avoid live lock a priority system can be used, whereby the longer a transaction has to wait, the higher its priority. For example, FCFS queue can be used for waiting transactions.

Time Stamping Methods

Timestamp: **A unique identifier created by the DBMS that includes the relative starting time of a transaction. Timestamps can be generated by simply using the system clock at the time the transaction started, or, more normally, by incrementing a logical counter every time a new transaction starts.**

Timestamping: **A concurrency control protocol that orders transactions in such a way that older transactions, transactions with smaller timestamps, get priority in the event of conflict.**

Beside transactions, timestampings are also done for data items. Each data items have read_timestamp and write_timestamp, giving the timestamp of the last transaction to read and write the item respectively.

Thomas's write rule:

1. If T wants write on item(x) and younger transaction has already been read item(x) i.e. ts(T)<read_timestamp(x). Then restart the transaction T using later timestamp.
2. If T wants write on item(x) and younger transaction has already been written item(x) i.e. ts(T)<write_timestamp(x). Then write operation is to be simply ignored. (ignore obsolete write rule)
3. Otherwise, as before, the write operation can proceed. We set write_timestamp(x) = ts(T).

Time	Operation	T_1	T_2	T_3
t_1		begin_transaction		
t_2	read(bal_x)	read(bal_x)		
t_3	$bal_x = bal_x + 10$	$bal_x = bal_x + 10$		
t_4	write(bal_x)	write(bal_x)	begin_transaction	
t_5	read(bal_y)		read(bal_y)	
t_6	$bal_y = bal_y + 20$		$bal_y = bal_y + 20$	begin_transaction
t_7	read(bal_y)			read(bal_y)
t_8	write(bal_y)		write(bal_y) Φ	
t_9	$bal_y = bal_y + 30$			$bal_y = bal_y + 30$
t_{10}	write(bal_y)			write(bal_y)
t_{11}	$bal_z = 100$			$bal_z = 100$
t_{12}	write(bal_z)			write(bal_z)
t_{13}	$bal_z = 50$	$bal_z = 50$		commit
t_{14}	write(bal_z)	write(bal_z) ϑ	begin_transaction	
t_{15}	read(bal_y)	commit	read(bal_y)	
t_{16}	$bal_y = bal_y + 20$		$bal_y = bal_y + 20$	
t_{17}	write(bal_y)		write(bal_y)	
t_{18}			commit	

Φ At time t_8, the write by transaction T_2 violates the first time stamping write rule described above and there fore is aborted and restarted at time t_{14}.

ϑ At time t_{14}, the write by transaction T_1 can safely by ignored using the ignored obsolete write rule, as it would have been overwritten by the write of transaction T_3 at time t_{12}.

Optimistic Techniques

Optimistic techniques are based on the assumption that conflict is rare, and that it is more efficient to allow transactions to proceed without imposing delays to ensure serializability.

There are two or three phases to an optimistic concurrency control protocol, depending on whether it is a read only or an update transaction:

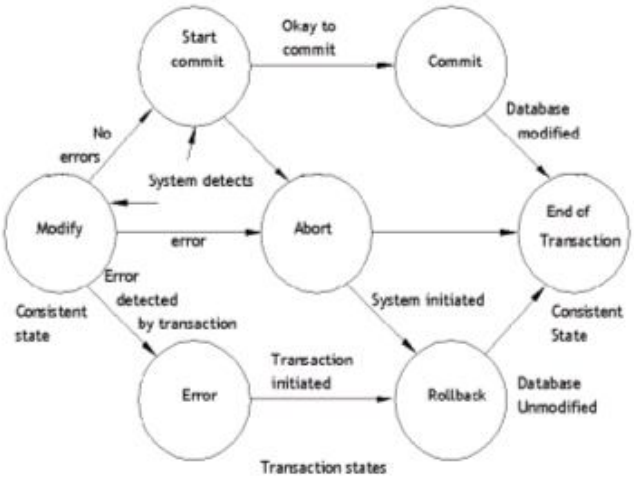
- **Read phase: This extends from the start of the transaction until immediately before and commit. The transaction reads the values of all data items it needs from the database and stores them in local variables. Updates are applied to a local copy of the data not to be database itself.**
- **Validation phase: This follows the read phase. Checks are performed to ensure serializability is not violated if the transaction updates are applied to the database. For a read only transaction, this consists of checking that the data values read are still the current values for the corresponding data items. If no interference occurred, the transaction is committed. If interference occurred, the transaction is aborted and restarted. For a transaction that has updates, validation consists of determining whether the current transaction leaves the database in a consistent state, with serializability maintained. If not, the transaction is aborted and restarted.**
- **Write phase: This follows the successful validation phase for update transactions. During this phase, the updates made to the local copy are applied to the database.**

To pass validation test, one of the following must be true:

1. **All transactions S with earlier time stamps must have finish before transaction T started; i.e., *finish(S)<start(T)*.**
2. **If transaction T starts before an earlier one S finishes, then:**
   - **A. The set of data items written by the earlier transaction are not the ones read by the current transaction; and**
   - **B. The earlier transaction completes its write phase before the current transaction enters its validation phase, i.e. *start(T)<finish(S)<validation(T)***

Compatibility of Locking

		Current state of locking of data items		
Lock mode of request		Unlocked	Shared	Exclusive
	Unlocked		Yes	Yes
	Shared	Yes	Yes	No
	Exclusive	Yes	No	No



Definition of Transaction: A transaction is a program we need whose execution may change the contents of a database. If the database was in a consistent state before a transaction then on the completion of the execution of the program unit corresponding to the transaction, the database will be in a consistent state. This requires the transaction be considered atomic..

States of a transaction: A transaction can end in three possible ways. It can end after a commit operation (a successful termination). It can detect an error during it's processing and decide to abort itself by performing a roll back operation (a suicidal termination). The DBMS or the operating system can force it to be aborted for one reason or another (a murderous termination).

Recovery and Reliability

- **A system is considered reliable if it functions as per its specifications and produces a correct set of output values for a given set of input values.**
- **The failure of a system occurs when the system does not function according to its specifications and fails to deliver the service for which it was intended.**
- **An error in the system occurs when a component of the system assumes a state that is not desirable; the fact that the state is undesirable is a subjective judgment.**
- **A fault is detected either when an error is propagated from one component to another or the failure of the component is observed.**
- **A fault-tolerant system, in addition to the fault-detection scheme, has redundant components and subsystem built in. On detection of a fault, these redundant components are used to replace the faulty components.**
- **Reliability is a measure used to indicate how successful a system is in providing the service it was intended for.**
- **A number of measures are used to define the reliability of a system. These include the mean time between failures (MTBF), the mean time to repair (MTTR), and the system availability, which is the fraction of time that a system performs according to its specifications.**

Types of Failures

1. **Hardware Failure: Hardware errors are having following sources:**

- **Design errors: These could include a design that did not meet the required specifications of performance and/or reliability; the use of components that are of poor quality or insufficient capacity; poor error detection and correction schemes; and failure to take into account the errors that can occur in the error detection and correction subsystems.**
- **Poor quality control (during fabrication): This could include poor connections, defective subsystems, and electrical and mechanical misalignments.**
- **Over utilization and overloading: Using a component or subsystem beyond its capacity. This could be a design error or utilization error where mismatching subcomponents may be used, or due to unforeseen circumstances a system is simply used beyond its capacity.**
- **Wear out: The system, especially its mechanical parts, tends to wear with usage causing it to divert from its design performance. Solid-state electrical parts do not wear out, but insulation on wire could undergo chemical changes with age and crack, leading to eventual failure.**

2. **Software Failure: Its sources are**

- **Design errors: Not all possible situations can be accounted for in the design process. This is particularly so in software design where it is hard to foresee all possible modes of operation, including the combinations and the sequence of usage of various components of a software system. However, the design should allow for the most serious types of errors to be detected and appropriate corrective action to be incorporated. In situations that could result in loss of life or property, the design must be fail-safe.**
- **Poor quality control: This could include undetected errors in entering the program code. Incompatibility of various modules and conflict of conventions between versions of the operating system are other possible causes of failure in software.**
- **Over utilization and overloading: A system designed to handle a certain load may be swamped when loading on it is exceeded. Buffers and stacks may overrun their boundaries or be shared erroneously.**
- **Wear out: There are no known errors caused by wear out of software; software does not wear out. However, the usefulness of a software system may become obsolete due to the introduction of new versions with additional features.**

3. Storage Medium Failure: It can be classified in the following categories

- **Volatile storage: An example of this type of storage is the semiconductor memory requiring an uninterruptible power source for correct operation. A volatile storage failure can occur due to the spontaneous shutdown of the computer system, sometimes referred to as a system crash.**
- **Nonvolatile storage: Examples of this type of storage are magnetic tape and magnetic disk systems. These types of storage devices do not require power for maintaining the stored information. A power failure or system shutdown will not result in the loss of information stored on such devices. However, nonvolatile storage devices such as magnetic disks can experience a mechanical failure in the form of a read/write head crash (i.e., the read/write head comes in contact with the recording surface instead of being a small distance from it), which could result in some loss of information.**
- **Permanent or Stable storage: Permanency of storage, in view of the possibility of failure of the storage medium, is achieved by redundancy. Thus, instead of having a single copy of the data on a nonvolatile storage medium, multiple copies of the data are stored. Each such copy is made on a separate nonvolatile storage device. Since these independent storage devices have independent failure modes, it is assumed that at least one of this multiple copies will survive any failure and be usable.**

4. Implementation of Stable Storage: Stable storage is implemented by replicating the data on a number of separate nonvolatile storage devices and using a careful writing scheme. Errors and failures occurring during transfer of information and leading to inconsistencies in the copies of data on stable storage can be arbitrated.

A write to the stable storage consists of writing the same block of data from volatile storage to distinct nonvolatile storage devices two or more times. If the writing of the block is done successfully, all copies of data will be identical and there will be no problems.

Types of Errors in Database Systems and Possible Detection Schemes

- **User Error: This includes errors in application programs as well as errors made by online users of the database. One remedy is to allow online users limited access rights to the database (e.g. read only). Validation routines are to be in built in the application programs for insertion or update operations. The routines will flag any values that are not valid and prompt the user to correct these errors.**
- **Consistency error: The database system should include routines that check for consistency of data entered in the database. A simple distinction between validity and consistency errors is that validity establishes that the data is of the correct type and within the specified range; consistency establishes that it is reasonable with respect to itself or to the current values of other data-items in the database.**
- **System error: This encompasses errors in the database system or the operating system, including situations such as deadlocks. Such errors are fairly hard to detect and require reprogramming the erroneous components of the system software or working with the DBMS vendor. Situations such as deadlocks are catered for in the DBMS by allowing appropriate locking facilities. Deadlocks are also catered to in the operating system by deadlock avoidance, prevention, or detection schemes.**
- **Hardware failure: This refers to hardware malfunctions including storage system failures.**
- **External environmental failure: Power failure is one possible type. Others are fire, flood, and other natural disasters, or malicious acts.**

Audit Trails

The DBMS also has routines that maintain an audit trail or a journal. An audit trail or a journal is a record of an update operation made on the database. The audit trail records who (user or the application program and a transaction number), when (time and date), (from) where (location of the user and/or the terminal), and what (identification of the data affected, as well as a before-and-after image of that portion of the database that was affected by the update operation). In addition, a DBMS contains routines that make a backup copy of the data that is modified. This is done by taking a "snapshot" of the before-and-after image of that portion of the database that is modified. For obvious reasons, the backups are produced on a separate storage medium.

Recovery Schemes

- **Forward error recovery: In this scheme, when a particular error in the system is detected, the recovery system makes an accurate assessment of the state of the system and then makes appropriate adjustments based on the anticipated result had the system been error free. The adjustments obviously depend on the error; consequently the error types have to be anticipated by the designers of the recovery system. The aim of the adjustment is to restore he system so that the effects of the error are canceled and the system can continue to operate. This scheme is not applicable to unanticipated errors.**
- **Backward error recovery: in this scheme no attempt is made to extrapolate what the state of the system would have been had the error not occurred. Instead, the system is reset to some previous correct state that is known to be free of any errors. The backward error recovery is a simulated reversal of time and does not try to anticipate the possible future state of a system.**

Failure Anticipation and Recovery:

- **Failures without loss of data: This type of failure is due to errors that the transaction discovers before it reaches the start to commit state. It can also be due to the action of the system, which resets its state to that which existed before the start of the transaction. No loss of data is involved in this type of failure, especially in the case where the transactions are run in a batch mode; these transactions can be returned later in the same sequence.**
- **Failure with loss of volatile storage: Such a failure can occur of a result of software or hardware errors. The processing of an active transaction is terminated in an unpredictable manner before it reaches its commit or rollback state and the contents of the volatile memory are lost.**
- **Failure with loss of nonvolatile storage: This is the sort of failure that can occur after the failure of a nonvolatile storage system; for example, a head crash on a disk drive or errors in writing to a nonvolatile device.**
- **Failure with a loss of stable storage: This type involves loss of data stored on stable storage. The cause of the loss could be due to natural or man-made disasters. Recovery from this type of failure requires manual regeneration of the database. The probability of such a failure is reduced to a very small value by having multiple copies of data in stable storage. Stored in physically secure environments in geographically dispersed locations.**

Recovery in a Centralized DBMS

To implement database transaction in the field of failures using data redundancy may be in the form of logs, checkpoints and archival copies of the database.

Logs

The log, which is usually written to stable storage, contains the redundant data required to recover from volatile storage failures and also from errors discovered by the transaction or the database system. For each transaction the following data is recorded on the log:

- **A start-of-transaction marker.**
- **The transaction identifier, which could include the who and where information referred to in section**
- **The record identifiers, which include the identifiers for the record occurrences.**
- **The operation(s) performed on the records (insert, delete, modify).**
- **The previous value(s) of the modified data. This information is required for undoing the changes made by a partially completed transaction; it is called the undo log. Where the modification made by the transaction is the insertion of a new record, the previous values can be assumed to be null.**
- **The updated value(s) of the modified record(s). This information is required for making sure that the changes made by a committed transaction are in fact reflected in he database and can be used to redo these modifications. This information is called the redo part of the log. In case the modification made by the transaction is the deletion of a record, the updated values can be assumed to be null.**
- **A commit transaction marker if the transaction is committed; otherwise an abort or rollback transaction marker.**

The log is written before any problem is made to the database. This is called the write-ahead log strategy.

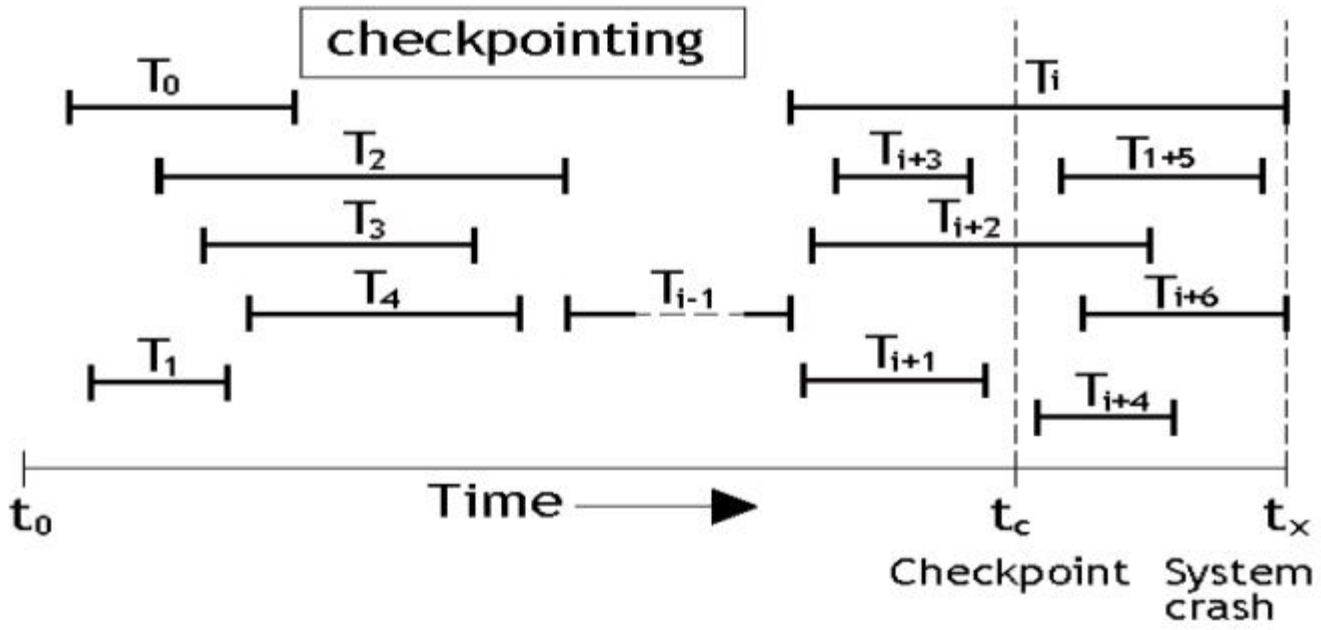
Checkpoints

A scheme called checkpoint is used to limit the volume of log information that has to be handled and processed in the event of a system failure involving the loss of volatile information. This checkpoint is additional to the log scheme.

In the case of a system crash, the log information being collected in buffers will be lost. A checkpoint operation, performed periodically, copies log information onto stable storage. The information and operations performed at each checkpoint consist of the following:

- **A start-of-checkpoint record giving the identification that it is a checkpoint along with the time and date of the checkpoint is written to the log on a stable storage device.**
- **All log information from the buffers in the volatile storage is copied to the log on stable storage.**
- **All database updates from the buffers in the volatile storage are propagated to the physical database.**
- **An end-of-check-point record is written and the address of the checkpoint record is saved on a file accessible to the recovery routine on start-up after a system crash.**

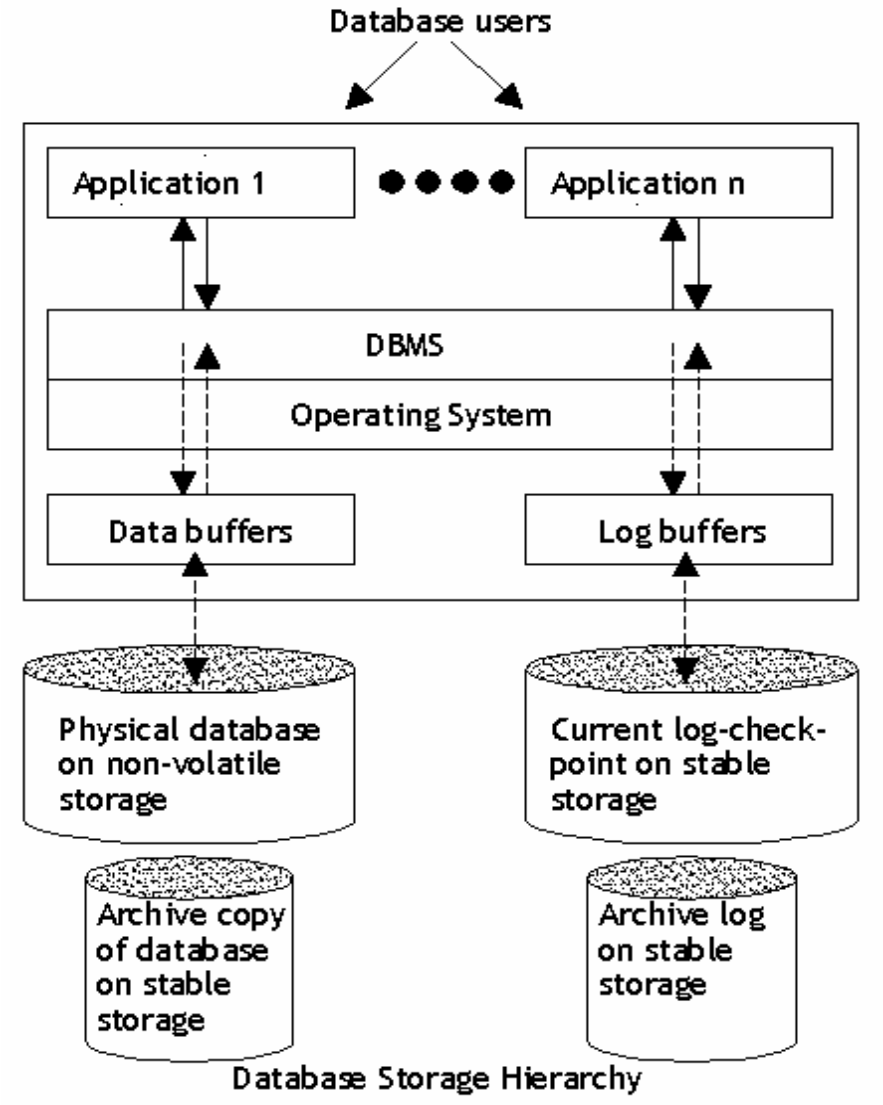
The frequency of check pointing is a design consideration of the recovery system. A checkpoint can be taken at fixed intervals of time (say, every 15 minutes). If this approach is used, a choice has to be made regarding what to do with the transactions that are active when a system timer generates the checkpoint signal. In one alternative, called transaction-consistent checkpoint the transactions that are active when the system timer signals a checkpoint are allowed to complete, but no new transactions (requiring modifications to the database) are allowed to be started until the checkpoint is completed. This scheme, though attractive, makes the database unavailable at regular intervals and may not be acceptable for certain online applications. In addition, this approach is not appropriate for long transactions. In the second variation, called action consistent checkpoint, active transactions are allowed to complete the current step before the checkpoint and no new actions can be started on the database until the checkpoint is completed; during the checkpoint no actions are permitted on the database. Another alternative, called transaction oriented checkpoint, is to take a checkpoint at the end of each transaction by forcing the log of the transaction onto stable storage. In effect, each commit transaction is a checkpoint.



At first all uncompleted transactions started before t_c and all that started after t_c are placed in the UNDO list UNDO List: (T_i, T_{i+2}, T_{i+4}, T_{i+5}, T_{i+6}). After system crash at t_x the recovery system scans backward. It finds transactions already committed and keep them in the REDO list and others in the UNDO list.
So REDO List: (T_{i+2}, T_{i+4}, T_{i+5}) and UNDO List: (T_i, T_{i+6}).

Archival Database and Implementation of the Storage Hierarchy of a Database System

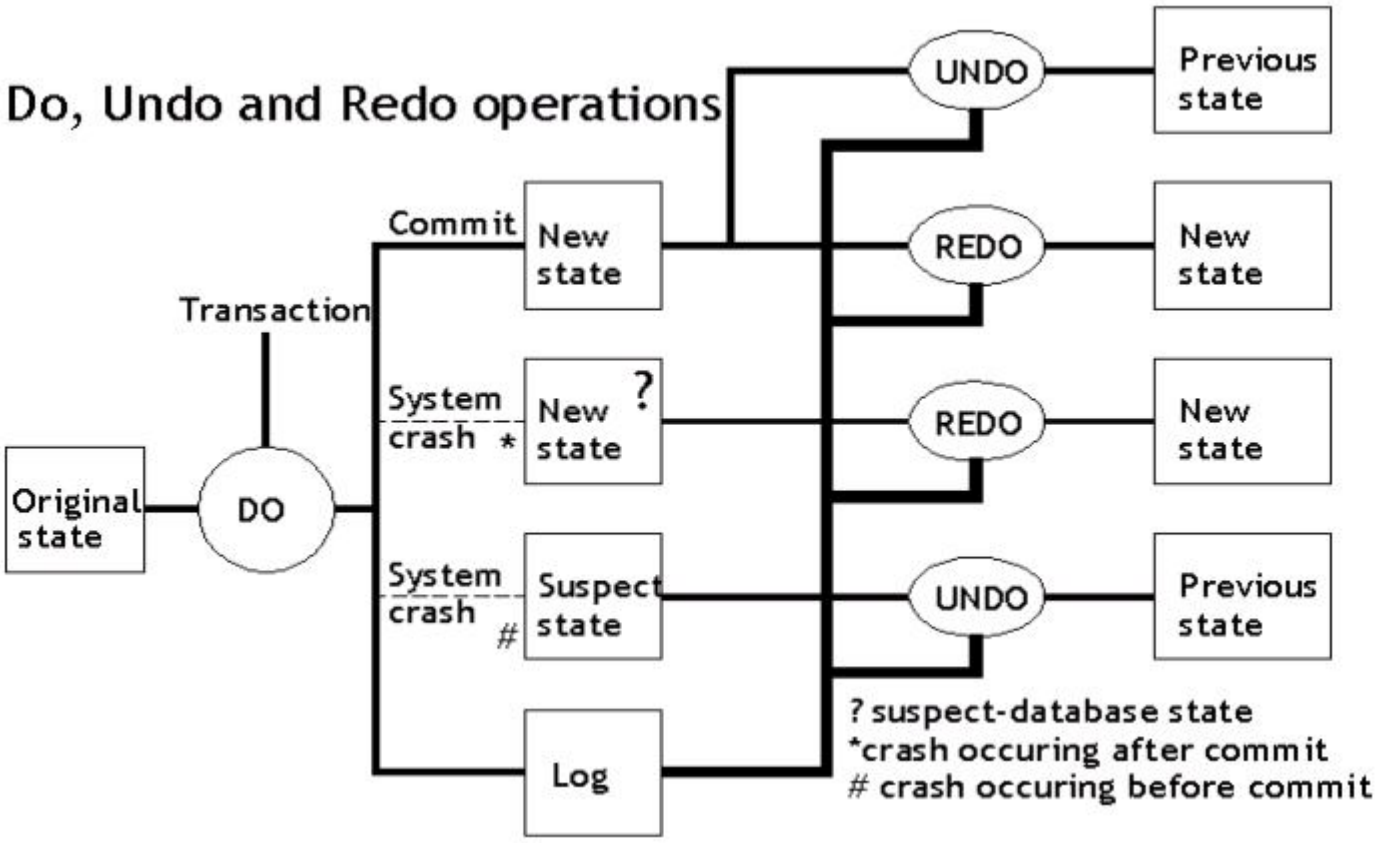
Different categories of data are used (e.g. archival database, physical database, archival log, and current log) in the database system. Physical database is the online copy of the database that is stored in nonvolatile storage and used by all active transactions. Current database is the current version of the database is made up of the physical database plus modifications implied by buffers in the volatile storage. Archival database in the stable storage is the copy of database in the stable storage in the quiescent mode (i.e. no transaction were active when the database was copied on the stable storage). It holds the effect of the committed transactions only. Current log contains the log information (including the check point) required for recovery from system failures involving loss of volatile information. Archival log is used for failures involving loss of nonvolatile information on all transactions made on the database from the time of archival copy in the chronological order. The on-line or current database is made up of all the records, indices accessible to the database during operation. It contains data stored in the nonvolatile storage and also in buffers, which have not been moved to nonvolatile storage yet. The materialized database is that portion of the database that is still intact after a failure. The intent is to limit the amount of lost data and the loss of completed transactions.



Database Storage Hierarchy

Do, Undo and Redo

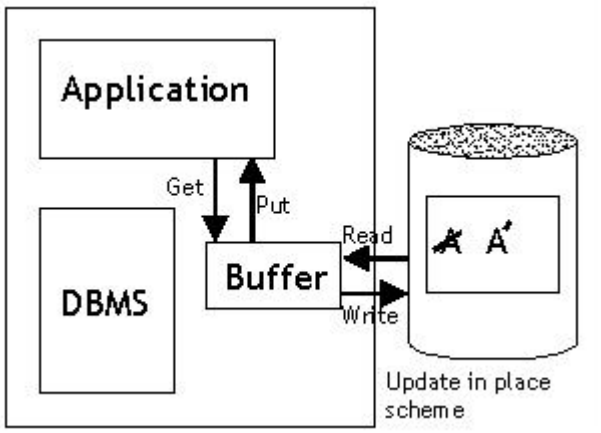
- **Transaction Do:** A transaction on the current database transforms it from the current state to a new state. This is so called Do-operation.
- **Transaction Undo:** A transaction that discovers an error while it is in progress and consequently needs to abort itself and rollback any changes made by it uses the transaction undo feature. A transaction also has to be undone when the DBMS forces the transaction to abort. A transaction undo removes all database changes, partial or otherwise, made by the transaction.
- **Transaction Redo:** Transaction redo involves performing the changes made by a transaction that committed before a system crash. With the write-ahead log strategy, a committed transaction implies that the log for the transaction would have been written to nonvolatile storage, but the physical database may or may not have been modified before the system failure. A transaction redo modifies the physical database to the new values for a committed transaction. Since the redo operation is idem potent, redoing the partial or complete modifications made by a transaction to the physical database will not pose a problem for recovery.
- **Global Undo:** Transactions that are partially complete at the time of a system crash with loss of volatile storage need to be undone by undoing nay changes made by the transaction. The global undo operation, initiated by the recovery system, involves undoing the partial or otherwise updates made by all uncommitted transactions at the time of a system failure.
- **Global Redo:** The global redo operation is required for recovery from failures involving nonvolatile storage loss. The archival copy of the database is used and all transactions committed since the time of the archival copy are redone to obtain a database updated to a point as close as possible to the time of the nonvolatile storage loss. The effects of the transaction in progress at the time of the nonvolatile loss will not be reflected in the recovered database. The archival copy of the database could be any where from months to days old and the number of transactions that have to be redone could be large. The log for the committed transactions needed for performing a global redo operation has to be stored on stable storage so that they are not lost with the loss of nonvolatile storage containing the physical database.



Reflecting updates to the database and recovery

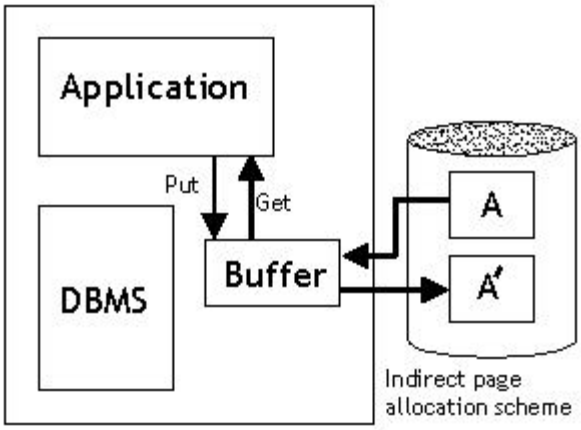
Update in place:

In this approach the modifications appear in the database in the original locations and, in the case of a simple updates, the new values will replace the old values.



In-direct update with careful replacement:

In this approach the modifications are not made directly on the physical database. Two possibilities can be considered the first scheme, called the shadow page scheme, makes the changes on a copy of that portion of the database being modified. The other scheme is called update via log. In this strategy of indirect update, the update operations of a transaction are logged and the log of a committed transaction is used to modify the physical database.



Savepoint:

It is used to emulate nested transactions. It is an identifier point in a flat transaction representing some partially consistent state, which can be used as an internal restart point for the transaction if a subsequent problem is detected. User can establish a save point, for example using a SAVE WORK statement. This generates an identifier that the user can subsequently use to roll the transaction back to, e.g. using a ROLLBACK WORK <savepoint_identifier> statement.

Sagas:

It is a sequence of (flat) transactions that can be interleaved with other transactions. The DBMS guarantees that either all the transactions in a saga are successfully completed or compensating transactions are run to recover from partial execution. Unlike a nested transaction, which has an arbitrary level of nesting. Further, for every sub-transaction that is defined, there is a corresponding compensating transaction that will semantically undo the sub-transaction's effect. Let a saga contains n transactions T_1, T_2, …. , T_n with corresponding compensating transactions C_1, C_2, … , C_n, then the outcome may have following sequence:

T_1, T_2, …. , T_n	**if the transaction completes successfully**
T_1, T_2, …. , T_i,C_{i-1}, …., C_2, C_1	**if sub-transaction T fails and is aborted**