Unit 3

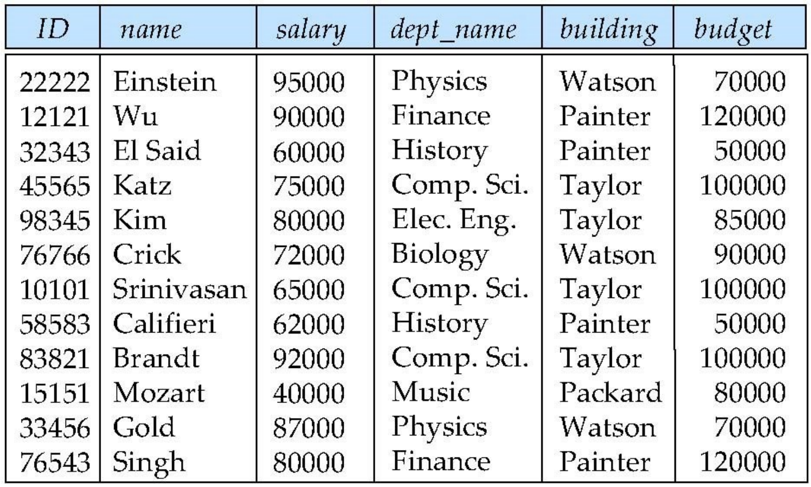
Functional Dependencies: Basic concepts, closure of set of functional dependencies, closure of attribute set, canonical cover, Decomposition: lossless join decomposition and dependency preservation, The Process of normalization, 1NF, 2NF, 3NF,BCNF, 4NF, 5NF.

Combine Schemas?

Result is possible repetition of information

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

Redundancy is Bad!



ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

Department

  - Efficiency + potential for errors
- Delete Physics Department
  - update multiple tuples
  - Efficiency + potential for errors
- Departments without instructor or instructors without departments
  - Need dummy department and dummy instructor
  - Makes aggregation harder and error prone.

A Combined Schema Without Repetition

- Combining is not always bad!
- Consider combining relations
  - *sec_class(course_id, sec_id, building, room_number)*

  and
  - *section(course_id, sec_id, semester, year)*

  into one relation
  - *section(course_id, sec_id, semester, year, building, room_number)*
- No repetition in this case

What About Smaller Schemas?

Suppose we had started with *inst_dept.* How would we know to split up (**decompose**) it into *instructor* and *department*?

- Write a rule "if there were a schema (*dept_name, building, budget*), then *dept_name* would be a candidate key"
- Denote as a **functional dependency**:

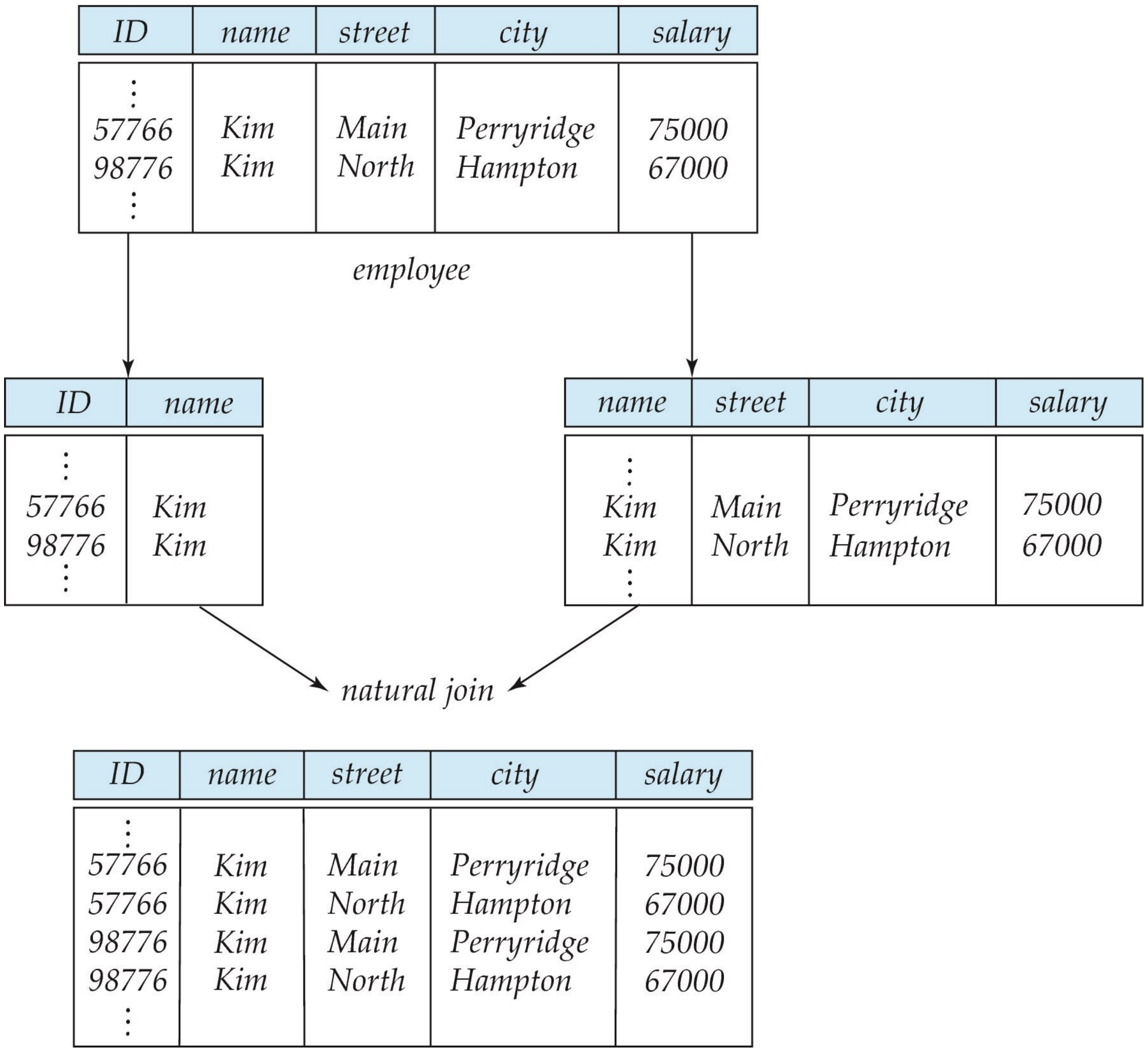
  dept_name → *building, budget*

- In *inst_dept*, because *dept_name* is not a candidate key, the building and budget of a department may have to be repeated.
  - This indicates the need to decompose *inst_dept*
- Not all decompositions are good. Suppose we decompose *employee(ID, name, street, city, salary)* into

  employee1 (*ID*, *name*)

  employee2 (*name*, *street, city, salary*)

A Lossy Decomposition



ID	name	street	city	salary
⋮				
57766	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

employee

ID	name
⋮	
57766	Kim
98776	Kim
⋮	

name	street	city	salary
⋮			
Kim	Main	Perryridge	75000
Kim	North	Hampton	67000
⋮			

natural join

ID	name	street	city	salary
⋮				
57766	Kim	Main	Perryridge	75000
57766	Kim	North	Hampton	67000
98776	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

- The above resulting tuples using the schemas resulting from the decomposition, and the result if we attempted to regenerate the original tuples using a natural join.
- As we see in the figure, the two original tuples appear in the result along with two new tuples that incorrectly mix data values pertaining to the two employees named Kim.
- Although we have more tuples, we actually have less information in the following sense.
- We can indicate that a certain street, city, and salary pertain to someone named Kim, but we are unable to distinguish which of the Kims.
- Thus,our decomposition is unable to represent certain important facts about the university employees. Clearly, we would like to avoid such decompositions. We shall refer to such decompositions as being lossy decompositions, and, conversely, to those that are not as lossless decompositions

Goals of Lossless-Join Decomposition

- Lossless-Join decomposition means splitting a table in a way so that we do not loose information
  - That means we should be able to reconstruct the original table from the decomposed table using joins

A	B	C
α	1	A
β	2	B

r

A	B
α	1
β	2

$\Pi_{A,B}(r)$

B	C
1	A
2	B

$\Pi_{B,C}(r)$

$\Pi_{A}(r) \bowtie \Pi_{B}(r)$

A	B	C
α	1	A
β	2	B

Goal — Devise a Theory for the Following

- Decide whether a particular relation *R* is in "good" form.
- In the case that a relation *R* is not in "good" form, decompose it into a set of relations $\{R_1, R_2, ..., R_n\}$ such that
  - each relation is in good form
  - the decomposition is a lossless-join decomposition
- Our theory is based on:
  - **1)** Models of dependency between attribute values
    - **functional dependencies**
    - multivalued dependencies
  - **2)** Concept of **lossless decomposition**
  - **3) Normal Forms** Based On
    - Atomicity of values
    - Avoidance of redundancy
    - Lossless decomposition

Functional Dependencies

The functional dependency is a relationship that exists between two attributes. It typically exists between the **primary key and non-key attribute within a table.**

$$X \rightarrow Y$$

The left side of FD is known as a **determinant**, the right side of the production is known as a **dependent**.

Assume we have an employee table with attributes:

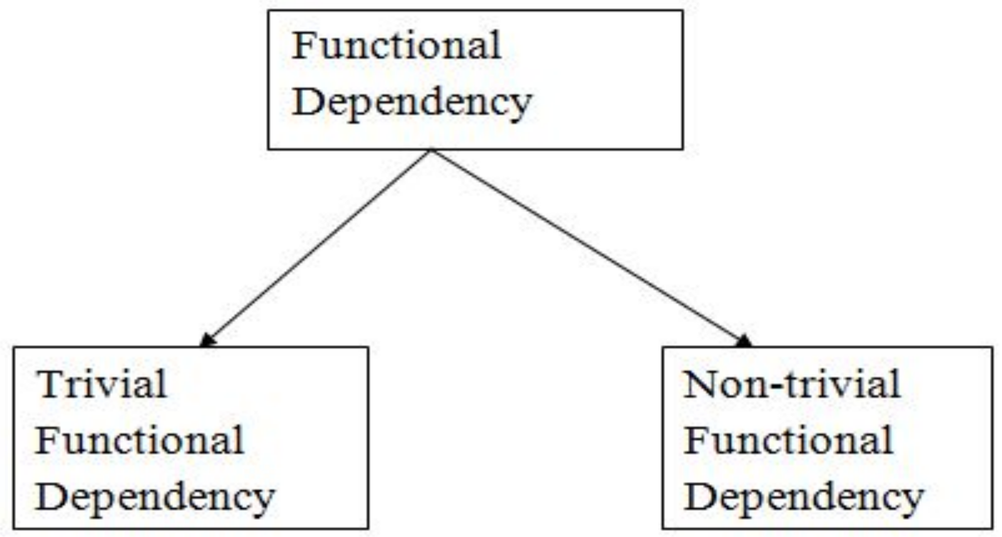
Emp_Id, Emp_Name, Emp_Address.

Here Emp_Id attribute can uniquely identify the Emp_Name attribute of employee table because if we know the Emp_Id, we can tell that employee name associated with it.

Functional dependency can be written as:

Emp_Id → Emp_Name

Types of Functional dependency


Functional Dependency
Trivial Functional Dependency
Non-trivial Functional Dependency

1. Trivial functional dependency

- $A \rightarrow B$ has trivial functional dependency if B is a subset of A.
- The following dependencies are also trivial like: $A \rightarrow A$, $B \rightarrow B$

Consider a table with two columns Employee_Id and Employee_Name.

{Employee_id, Employee_Name} → Employee_Id
is a trivial functional dependency as

Employee_Id is a subset of {Employee_Id, Employee_Name}
.

Also,
Employee_Id → Employee_Id and
Employee_Name → Employee_Name
are trivial dependencies too.

2. Non-trivial functional dependency

- A → B has a non-trivial functional dependency if B is not a subset of A.
- When A intersection B is NULL, then A → B is called as complete non-trivial.

1. ID → Name,
2. Name → DOB

Functional Dependencies (Cont.)

Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

inst_dept (*ID, name, salary, dept_name, building, budget*).

We expect these functional dependencies to hold:

dept_name → *building*

and *ID* *building*

but would not expect the following to hold:

dept_name → *salary*

Functional Dependencies (Cont.)

- *A* functional dependency is **trivial** if it is satisfied by all instances of a relation
  - Example*:*
    - *ID, name* → *ID*
    - *name* → *name*
  - In general, $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$

bor_loan(customer_id, loan_number, amount)

- loan_number -> amount,
- but loan_number -> customer_id does not hold
- The amount information repeats unnecessarily

Inference Rule (IR):

- The Armstrong's axioms are the basic inference rule.

- Armstrong's axioms are used to conclude functional dependencies on a relational database.

- The inference rule is a type of assertion. It can apply to a set of FD(functional dependency) to derive other FD.

- Using the inference rule, we can derive additional functional dependency from the initial set.
  The Functional dependency has 6 types of inference rule:

1. **Reflexive Rule (IR_1)**

In the reflexive rule, if Y is a subset of X, then X determines Y.

If X ⊇ Y then X → Y

Example
X = {a, b, c, d, e}
Y = {a, b, c}

2. **Augmentation Rule (IR_2)**

The augmentation is also called as a partial dependency. In augmentation, if X determines Y, then XZ determines YZ for any Z.

If X → Y then XZ → YZ

Example
For R(ABCD), **if** A → B then AC → BC

3. Transitive Rule (IR_3)

In the transitive rule, if X determines Y and Y determine Z, then X must also determine Z.

If X $\rightarrow$ Y and Y $\rightarrow$ Z then X $\rightarrow$ Z

4. Union Rule (IR_4)

Union rule says, if X determines Y and X determines Z, then X must also determine Y and Z.

If X $\rightarrow$ Y and X $\rightarrow$ Z then X $\rightarrow$ YZ

5. Decomposition Rule (IR_5)

Decomposition rule is also known as project rule. It is the reverse of union rule.

This Rule says, if X determines Y and Z, then X determines Y and X determines Z separately.

If X $\rightarrow$ YZ then X $\rightarrow$ Y and X $\rightarrow$ Z

6. Pseudo transitive Rule (IR_6)

In Pseudo transitive Rule, if X determines Y and YZ determines W, then XZ determines W.

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

If $X \rightarrow Y$ and $X \rightarrow Z$ then $X \rightarrow YZ$

1. $X \rightarrow Y$ (given)
2. $X \rightarrow Z$ (given)
3. $X \rightarrow XY$ (using IR_2 on 1 by augmentation with X. Where XX = X)
4. $XY \rightarrow YZ$ (using IR_2 on 2 by augmentation with Y)
5. $X \rightarrow YZ$ (using IR_3 on 3 and 4)

If $X \rightarrow YZ$ then $X \rightarrow Y$ and $X \rightarrow Z$

1. $X \rightarrow YZ$ (given)
2. $YZ \rightarrow Y$ (using IR_1 Rule)
3. $X \rightarrow Y$ (using IR_3 on 1 and 2)

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

1. $X \rightarrow Y$ (given)
2. $WY \rightarrow Z$ (given)
3. $WX \rightarrow WY$ (using IR_2 on 1 by augmenting with W)
4. $WX \rightarrow Z$ (using IR_3 on 3 and 2)

Closure of a Set of Functional Dependencies

- Given a set *F* of functional dependencies, there are certain other functional dependencies that are logically implied by *F*.
  - For example: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- The set of **all** functional dependencies logically implied by *F* is the **closure** of *F*.
- We denote the *closure* of *F* by $\mathbf{F^+}$.
- F^+ is a superset of *F*.

Example

$R = (A, B, C, G, H, I)$

$F = \{ A \rightarrow B, A \rightarrow C , CG \rightarrow H ,CG \rightarrow I, B \rightarrow H\}$

some members of F^+

- $A \rightarrow H$
  - by transitivity from $A \rightarrow B$ *and* $B \rightarrow H$
- $AG \rightarrow I$
  - by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$ and then transitivity with $CG \rightarrow I$
- $CG \rightarrow HI$
  - by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$, and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$, and then transitivity

Closure of Attribute Sets

Given a set of attributes α, define the *closure* of α under *F* (denoted by α^+) as the set of attributes that are functionally determined by α under *F*

Algorithm to compute α^+, the closure of α under *F*

```
result := α;
while (changes to result) do
    for each β → γ in F do
        begin
            if β ⊆ result then  result := result ∪ γ
        end
```

Example of Attribute Set Closure

$R = (A, B, C, G, H, I)$

$F = \{A \rightarrow B$

$A \rightarrow C$

$CG \rightarrow H$

$CG \rightarrow I$

$B \rightarrow H\}$

$(AG)^+$

1. *result* = *AG*
2. *result* = *ABCG* ($A \rightarrow C$ and $A \rightarrow B$)
3. *result* = *ABCGH* ($CG \rightarrow H$ and $CG \subseteq AGBC$)
4. *result* = *ABCGHI* ($CG \rightarrow I$ and $CG \subseteq AGBCH$)

Is *AG* a candidate key?

1. Is AG a super key?
   1. Does $AG \rightarrow R$? == Is $(AG)^+ \supseteq R$
2. Is any subset of AG a superkey?
   1. Does $A \rightarrow R$? == Is $(A)^+ \supseteq R$
   2. Does $G \rightarrow R$? == Is $(G)^+ \supseteq R$

Procedure for Computing F^+

To compute the closure of a set of functional dependencies F:

```
$F^+ = F$
**repeat**
    **for each** functional dependency $f$ in $F^+$
        apply reflexivity and augmentation rules on $f$
        add the resulting functional dependencies to $F^+$
    **for each** pair of functional dependencies $f_1$ and $f_2$ in $F^+$
        **if** $f_1$ and $f_2$ can be combined using transitivity
            **then** add the resulting functional dependency to $F^+$
**until** $F^+$ does not change any further
```

NOTE: We shall see an alternative more efficient procedure for this task later

Canonical Cover of Functional Dependencies in DBMS

Whenever a user updates the database, the system must check whether any of the functional dependencies are getting violated in this process. If there is a violation of dependencies in the new database state, the system must roll back. Working with a huge set of functional dependencies can cause unnecessary added computational time. This is where the canonical cover comes into play.

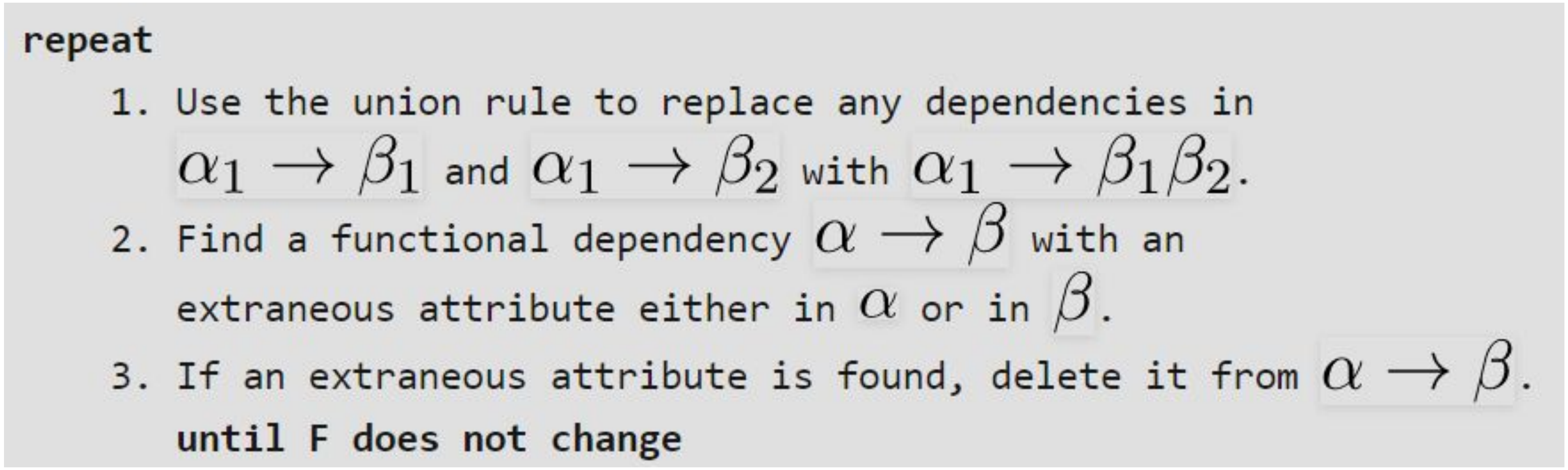
A canonical cover of a set of functional dependencies F is a simplified set of functional dependencies that has the same closure as the original set F.

Important definitions:

Extraneous attributes: An attribute of a functional dependency is said to be extraneous if we can remove it without changing the closure of the set of functional dependencies.

Canonical cover: A canonical cover of a set of functional dependencies F such that ALL the following properties are satisfied:

Algorithm to compute canonical cover of set F:



```
repeat
    1. Use the union rule to replace any dependencies in
       $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1\beta_2$.
    2. Find a functional dependency $\alpha \rightarrow \beta$ with an
       extraneous attribute either in $\alpha$ or in $\beta$.
    3. If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$.
       until F does not change
```

Lossless Join-Decomposition Dependency Preservation

Decomposition of a Relation-

The process of breaking up or dividing a single relation into two or more sub relations is called as decomposition of a relation.

Properties of Decomposition-

The following two properties must be followed when decomposing a given relation-

1. Lossless decomposition-

Lossless decomposition ensures-

- No information is lost from the original relation during decomposition.
- When the sub relations are joined back, the same relation is obtained that was decomposed.

Every decomposition must always be lossless.

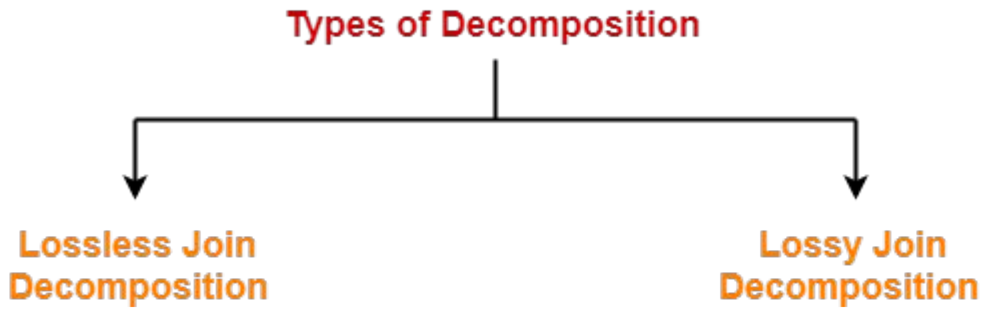
2. Dependency Preservation-

Dependency preservation ensures-

- None of the functional dependencies that holds on the original relation are lost.
- The sub relations still hold or satisfy the functional dependencies of the original relation.

Types of Decomposition-

Decomposition of a relation can be completed in the following two ways-



<u>1. Lossless Join Decomposition-</u>

- Consider there is a relation R which is decomposed into sub relations R_1 , R_2 , …. , $R_{n.}$
- This decomposition is called lossless join decomposition when the join of the sub relations results in the same relation R that was decomposed.
- For lossless join decomposition, we always have-

where $\bowtie$ is a natural join operator

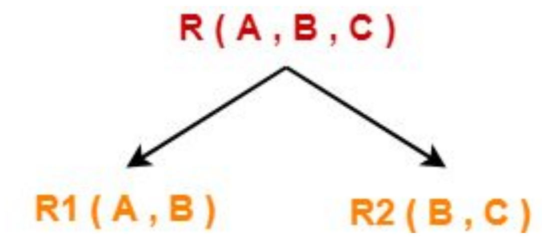
$$\boxed{R_1 \bowtie R_2 \bowtie R_3 \ldots\ldots \bowtie R_n = R}$$

Example-

Consider the following relation R(A , B , C)-

R(A , B , C)

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



A	B
1	2
2	5
3	3

B	C
2	1
5	3
3	3

Now, let us check whether this decomposition is lossless or not.

For lossless decomposition, we must have-

$\mathbf{R_1 \bowtie R_2 = R}$

Now, if we perform the natural join ($\bowtie$) of the sub relations R_1 and R_2 , we get-

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

This relation is same as the original relation R.

Thus, we conclude that the above decomposition is lossless join decomposition.

2. Lossy Join Decomposition-

- Consider there is a relation R which is decomposed into sub relations R_1 , R_2 , …. , $R_{n.}$
- This decomposition is called lossy join decomposition when the join of the sub relations does not result in the same relation R that was decomposed.
- The natural join of the sub relations is always found to have some extraneous tuples.
- For lossy join decomposition, we always have-

$$\boxed{R_1 \bowtie R_2 \bowtie R_3 \ldots\ldots\ldots \bowtie R_n \supset R}$$

where ⋈ is a natural join operator

Example-

Consider the following relation R(A , B , C)-

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

R(A , B , C)

$$\begin{tikzcd}
 & R(A,B,C) \arrow[dl] \arrow[dr] & \\
R1(A,C) & & R2(B,C)
\end{tikzcd}$$

A	C
1	1
2	3
3	3

R_1(A , B)

B	C
2	1
5	3
3	3

R_2(B , C)

Now, let us check whether this decomposition is lossy or not.
For lossy decomposition, we must have- $R_1 \bowtie R_2 \supset R$
Note:-A $\supset$ B means:
A is a superset of B

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

This relation is not same as the original relation R and contains some extraneous tuples.
Clearly, $R_1 \bowtie R2 \supset R$.
Thus, we conclude that the above decomposition is lossy join decomposition.

Dependency Preservation

- Let F_i be the set of dependencies F^+ that include only attributes in R_i.
  - A decomposition is **dependency preserving**, if

    $$(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$$

  - If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive.

Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B$
  $B \rightarrow C\}$
  Key = $\{A\}$
- Decomposition $R_1 = (A, B)$, $R_2 = (B, C)$
  - Lossless-join decomposition
  - Dependency preserving

Normal Forms

So Far

- **Theory of dependencies**
- **Decompositions and ways to check whether they are "good"**
  - Lossless
  - Dependency preserving
- **What is missing?**
  - Define what constitutes a good relation
    - Normal forms
  - How to check for a good relation
    - Test normal forms
  - How to achieve a good relation
    - Translate into normal form
    - Involves decomposition

Normal Forms in DBMS
Normalization

- **Prerequisite –** Database normalization and functional dependency concept.

- Normalization is the process of organizing the data in the database.

- Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate the undesirable characteristics like Insertion, Update and Deletion Anomalies.

- Normalization divides the larger table into the smaller table and links them using relationship.

- The normal form is used to reduce redundancy from the database table.

Normal Form	Description
1NF	A relation is in 1NF if it contains an atomic value.
2NF	A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key.
3NF	A relation will be in 3NF if it is in 2NF and no transition dependency exists.
BCNF	A relation R is in BCNF if R is in Third Normal Form and for every FD, LHS is super key. A relation is in BCNF iff in every non-trivial functional dependency X –> Y, X is a super key.
4NF	A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency.
5NF	A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.

First Normal Form (1NF)

- A relation will be 1NF if it contains an atomic value.
- It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attribute.
- First normal form disallows the multi-valued attribute, composite attribute, and their combinations.

- Note-An atomic value is a value that cannot be divided

Example: Relation EMPLOYEE is not in 1NF because of multi-valued attribute EMP_PHONE.

EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385, 9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389, 8589830302	Punjab

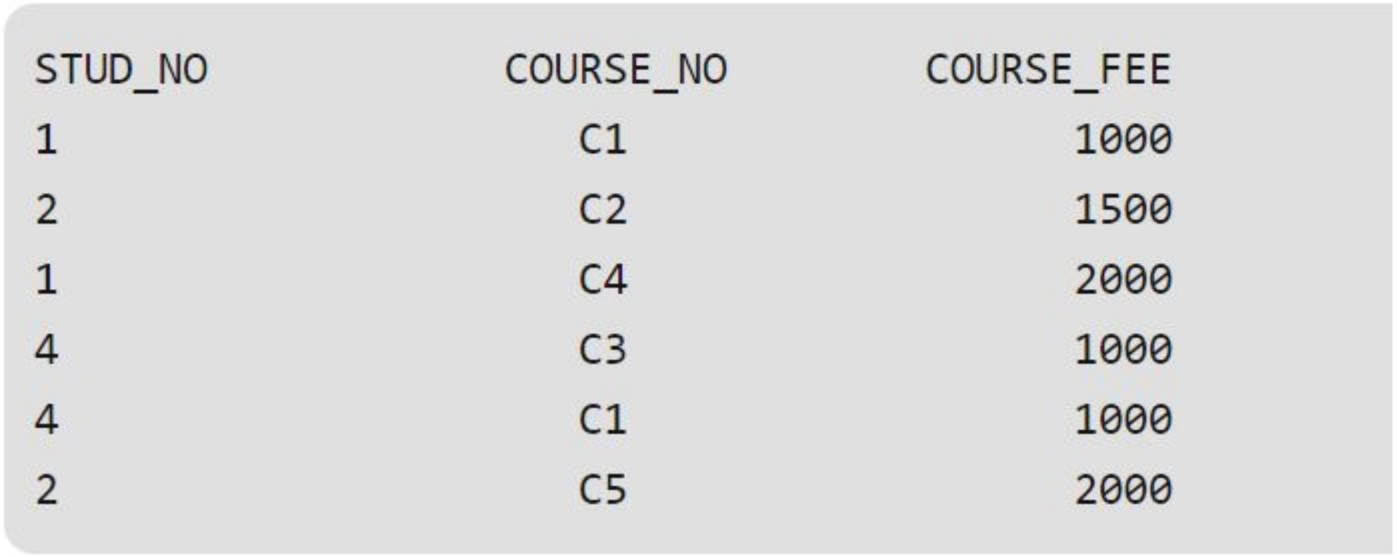
The decomposition of the EMPLOYEE table into 1NF has been shown below:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385	UP
14	John	9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389	Punjab
12	Sam	8589830302	Punjab

Second Normal Form (2NF)

- In the 2NF, relational must be in 1NF.
- In the second normal form, all non-key attributes are fully functional dependent on the primary key.

Example 1 – Consider table as following below



STUD_NO	COURSE_NO	COURSE_FEE
1	C1	1000
2	C2	1500
1	C4	2000
4	C3	1000
4	C1	1000
2	C5	2000

{Note that, there are many courses having the same course fee. }
Here,
1.COURSE_FEE cannot alone decide the value of COURSE_NO or STUD_NO;
2.COURSE_FEE together with STUD_NO cannot decide the value of COURSE_NO;
3.COURSE_FEE together with COURSE_NO cannot decide the value of STUD_NO;

Hence,
COURSE_FEE would be a non-prime attribute, as it does not belong to the one only candidate key {STUD_NO, COURSE_NO} ;

But, COURSE_NO -> COURSE_FEE , i.e., COURSE_FEE is dependent on COURSE_NO, which is a proper subset of the candidate key.
Non-prime attribute COURSE_FEE is dependent on a proper subset of the candidate key,
which is a partial dependency and so this relation is not in 2NF.
To convert the above relation to 2NF,
we need to split the table into two tables such as :
Table 1: STUD_NO, COURSE_NO
Table 2: COURSE_NO, COURSE_FEE

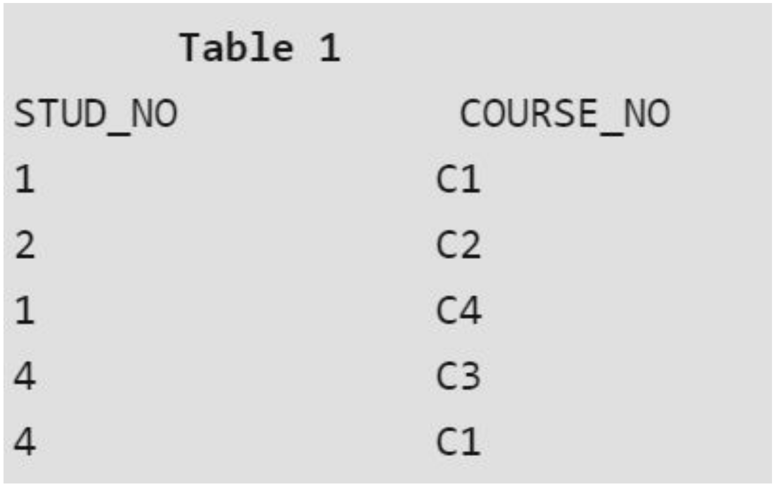


Table 1	
STUD_NO	COURSE_NO
1	C1
2	C2
1	C4
4	C3
4	C1

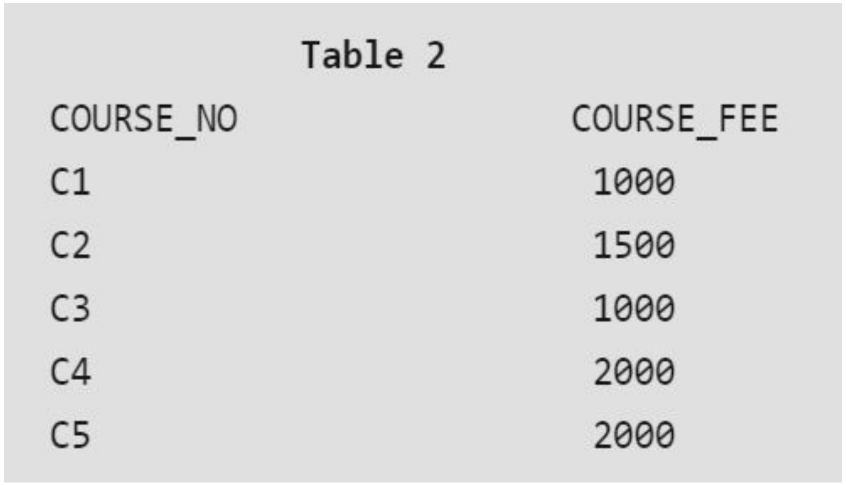


Table 2	
COURSE_NO	COURSE_FEE
C1	1000
C2	1500
C3	1000
C4	2000
C5	2000

Third Normal Form (3NF)

- A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency.

- 3NF is used to reduce the data duplication. It is also used to achieve the data integrity.

- If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.

A relation is in third normal form if it holds at least one of the following conditions for every non-trivial function dependency X → Y.

1. X is a super key.
2. Y is a prime attribute, i.e., each element of Y is part of some candidate key.

Note-Prime attributes are those attributes which are present in the primary key of that table. While Non -primary attributes are those that are not there in the primary key.

Example:
EMPLOYEE_DETAIL table:

EMP_ID	EMP_NAME	EMP_ZIP	EMP_STATE	EMP_CITY
222	Harry	201010	UP	Noida
333	Stephan	02228	US	Boston
444	Lan	60007	US	Chicago
555	Katharine	06389	UK	Norwich
666	John	462007	MP	Bhopal

Super key in the table above:

1. {EMP_ID}, {EMP_ID, EMP_NAME}, {EMP_ID, EMP_NAME, EMP_ZIP}.

…so on

Candidate key: {EMP_ID}

Non-prime attributes: In the given table, all attributes except EMP_ID are non-prime.

Here, EMP_STATE & EMP_CITY dependent on EMP_ZIP and EMP_ZIP dependent on EMP_ID.

The non-prime attributes (EMP_STATE, EMP_CITY) transitively dependent on super key(EMP_ID).

It violates the rule of third normal form.

That's why we need to move the EMP_CITY and EMP_STATE to the new <EMPLOYEE_ZIP> table, with **EMP_ZIP as a Primary key.**

EMPLOYEE table:

EMP_ID	EMP_NAME	**EMP_ZIP**
222	Harry	201010
333	Stephan	02228
444	Lan	60007
555	Katharine	06389
666	John	462007

EMPLOYEE_ZIP table:

EMP_ZIP	EMP_STATE	EMP_CITY
201010	UP	Noida
02228	US	Boston
60007	US	Chicago
06389	UK	Norwich
462007	MP	Bhopal

Boyce Codd normal form (BCNF)

1.BCNF is the advance version of 3NF. It is stricter than 3NF.
2.A table is in BCNF if every functional dependency X → Y, X is the super key(Group of attribute) of the table alone attribute not allowed .
3.For BCNF, the table should be in 3NF, and for every FD, LHS is super key.

Example: Suppose there is a company wherein employees work in **more than one department**. They store the data like this:

emp_id	emp_nationality	emp_dept	dept_type	dept_no_of_emp
1001	Austrian	Production and planning	D001	200
1001	Austrian	stores	D001	250
1002	American	design and technical support	D134	100
1002	American	Purchasing department	D134	600

Functional dependencies in the table above:
emp_id -> emp_nationality
emp_dept -> {dept_type, dept_no_of_emp}
Candidate key: {emp_id, emp_dept}
The table is not in BCNF as neither emp_id nor emp_dept alone are keys.
To make the table comply with BCNF we can break the table in three tables like this:

emp_nationality table:

emp_id	emp_nationality
1001	Austrian
1002	American

emp_dept table:

emp_dept	dept_type	dept_no_of_emp
Production and planning	D001	200
stores	D001	250
design and technical support	D134	100
Purchasing department	D134	600

emp_dept_mapping table:

emp_id	emp_dept
1001	Production and planning
1001	stores
1002	design and technical support
1002	Purchasing department

Functional dependencies:
emp_id -> emp_nationality
emp_dept -> {dept_type, dept_no_of_emp}
Candidate keys:
For first table: emp_id
For second table: emp_dept
For third table: {emp_id, emp_dept}
This is now in BCNF as in both the functional dependencies left side part is a key.

BCNF

❖ **Definition**

One of the more desirable normal forms that we can obtain is **Boyce–Codd normal form** (**BCNF**). A relation schema *R* is in BCNF with respect to a set *F* of functional dependencies if, for all functional dependencies in *F*+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency (that is, $\beta \subseteq \alpha$).
- α is a superkey for schema *R*.

- A database design is in BCNF if each member of the set of relation schemas that constitutes the design is in BCNF.
- Consider the following relation schemas and their respective functional dependencies:

✔ • *Customer-schema* = (*customer-name*, *customer-street*, *customer-city*)
customer-name → *customer-street customer-city*

✔ • *Branch-schema* = (*branch-name*, *assets*, *branch-city*)
branch-name → *assets branch-city*

✔ • *Loan-info-schema* = (*branch-name*, *customer-name*, *loan-number*, *amount*)
loan-number → *amount branch-name*

BCNF Decomposition Algorithm

```
    result := {R };
done := false;
compute $F^+$;
while (not done) do
    if (there is a schema $R_i$ in result that is not in BCNF)
        then begin
            let $\alpha \rightarrow \beta$ be a nontrivial functional dependency that holds on $R_i$
                      such that $\alpha \rightarrow R_i$ is not in $F^+$,
                      and $\alpha \cap \beta = \emptyset$;
            result := (result – $R_i$) $\cup$ ($R_i$ – $\beta$) $\cup$ ($\alpha$, $\beta$ );
        end
        else done := true;
```

Note: each R_i is in BCNF, and decomposition is lossless-join.

- *Lending-schema* = (*branch-name, branch-city, assets, customer-name, loan-number, amount*)

The set of functional dependencies that we require to hold on *Lending-schema* are

branch-name → assets branch-city
loan-number → amount branch-name

- A candidate key for this schema is *{loan-number, customer-name}*.

- The functional dependency *branch-name → assets branch-city* holds on *Lending-schema*, but *branch-name* is not a superkey.

- Thus, *Lendingschema* is not in BCNF.

- We replace *Lending-schema* by
  Branch-schema = (*branch-name, branch-city, assets*)
  Loan-info-schema = (*branch-name, customer-name, loan-number, amount*)

• The only nontrivial functional dependencies that hold on *Branch-schema* include *branch-name* on the left side of the arrow. Since *branch-name* is a key for *Branch-schema*, the relation *Branch-schema* is in BCNF.

- The functional dependency
  loan-number → *amount branch-name*

holds on *Loan-info-schema*, but *loan-number* is not a key for *Loan-info-schema*.

- We replace *Loan-info-schema* by
  Loan-schema = (*loan-number, branch-name, amount*)
  Borrower-schema = (*customer-name, loan-number*)

• *Loan-schema* and *Borrower-schema* are in BCNF.
Thus, the decomposition of *Lending-schema* results in the three relation schemas *Branchschema*, *Loan-schema*, and *Borrower-schema*, each of which is in BCNF.

Example of BCNF Decomposition

$R = (A, B, C)$

$F = \{A \rightarrow B$

$\quad B \rightarrow C\}$

Key = $\{A\}$

R is not in BCNF ($B \rightarrow C$ but B is not superkey)

Decomposition

- $R_1 = (B, C)$
- $R_2 = (A,B)$

Example of BCNF Decomposition

Original relation *R* and functional dependency *F*

R = (*branch_name, branch_city, assets,*
customer_name, loan_number, amount)

F = {*branch_name* → *assets branch_city*
loan_number → *amount branch_name* }

Key = {*loan_number, customer_name*}

Decomposition

R_1 = (*branch_name, branch_city, assets*)
R_2 = (*branch_name, customer_name, loan_number, amount*)
R_3 = (*branch_name, loan_number, amount*)
R_4 = (*customer_name, loan_number*)

Final decomposition

R_1, R_3, R_4

BCNF and Dependency Preservation

It is not always possible to get a BCNF decomposition that is dependency preserving

$R = (J, K, L)$
$F = \{JK \rightarrow L$
$\quad L \rightarrow K\}$
Two candidate keys = JK and JL
R is not in BCNF
Any decomposition of R will fail to preserve
$$JK \rightarrow L$$
This implies that testing for $JK \rightarrow L$ requires a join

Fourth normal form (4NF)

A relation will be in 4NF if it is in Boyce Codd normal form
and **has no multi-valued dependency.**
For a dependency A → B, if for a single value of A, multiple values of B exists, then the relation will be a multi-valued dependency.

Example

STUDENT

STU_ID	COURSE	HOBBY
21	Computer	Dancing
21	Math	Singing
34	Chemistry	Dancing
74	Biology	Cricket
59	Physics	Hockey

The given STUDENT table is in 3NF, but the COURSE and HOBBY are two independent entity.
Hence, there is no relationship between COURSE and HOBBY.
In the STUDENT relation, a student with STU_ID, **21** contains two courses, **Computer** and **Math** and two hobbies, **Dancing** and **Singing**.
So there is a Multi-valued dependency on STU_ID, which leads to unnecessary repetition of data.
So to make the above table into 4NF, we can decompose it into two tables:

STUDENT_COURSE

STU_ID	COURSE
21	Computer
21	Math
34	Chemistry
74	Biology
59	Physics

STUDENT_HOBBY

STU_ID	HOBBY
21	Dancing
21	Singing
34	Dancing
74	Cricket
59	Hockey

Fifth normal form (5NF)

A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.
5NF is satisfied when all the tables are broken into as many tables as possible in order to avoid redundancy.
5NF is also known as Project-join normal form (PJ/NF).

SUBJECT	LECTURER	SEMESTER
Computer	Anshika	Semester 1
Computer	John	Semester 1
Math	John	Semester 1
Math	Akash	Semester 2
Chemistry	Praveen	Semester 1

In the above table, John takes both Computer and Math class for Semester 1 but he doesn't take Math class for Semester 2.
In this case, combination of all these fields required to identify a valid data.
Suppose we add a new Semester as Semester 3 but do not know about the subject and who will be taking that subject so we leave Lecturer and Subject as NULL.
But all three columns together acts as a primary key, so we can't leave other two columns blank.
So to make the above table into 5NF, we can decompose it into three relations P1, P2 & P3:

P1

SEMESTER	SUBJECT
Semester 1	Computer
Semester 1	Math
Semester 1	Chemistry
Semester 2	Math

P2

SUBJECT	LECTURER
Computer	Anshika
Computer	John
Math	John
Math	Akash
Chemistry	Praveen

P3

SEMSTER	LECTURER
Semester 1	Anshika
Semester 1	John
Semester 1	John
Semester 2	Akash
Semester 1	Praveen

References

Abraham Silberschatz ,HenryKorth , S.Sudarshan,"Database System concepts",5th Edition ,McGraw Hill International Edition
http://www.timeconsult.com/TemporalData/TemporalDB.html
http://punarvasi.com/different-types-of-database-users/

CODD's Rules

Codd, after his extensive research on the Relational Model of database systems, came up with twelve rules of his own, which according to him, a database must obey in order to be regarded as a true relational database.

These rules can be applied on any database system that manages stored data using only its relational capabilities. This is a foundation rule, which acts as a base for all the other rules.

- **Rule 1: Information Rule**

  The data stored in a database, may it be user data or metadata, must be a value of some table cell. Everything in a database must be stored in a table format.

- **Rule 2: Guaranteed Access Rule**

  Every single data element (value) is guaranteed to be accessible logically with a combination of table-name, primary-key (row value), and attribute-name (column value). No other means, such as pointers, can be used to access data.

- **Rule 3: Systematic Treatment of NULL Values**

  The NULL values in a database must be given a systematic and uniform treatment. This is a very important rule because a NULL can be interpreted as one of the following – data is missing, data is not known, or data is not applicable.

- **Rule 4: Active Online Catalog**

  The structure description of the entire database must be stored in an online catalog, known as **data dictionary**, which can be accessed by authorized users. Users can use the same query language to access the catalog which they use to access the database itself.

- **Rule 5: Comprehensive Data Sub-Language Rule**

  A database can only be accessed using a language having linear syntax that supports data definition, data manipulation, and transaction management operations. This language can be used directly or by means of some application. If the database allows access to data without any help of this language, then it is considered as a violation.

- **Rule 6: View Updating Rule**

  All the views of a database, which can theoretically be updated, must also be updatable by the system.

- **Rule 7: High-Level Insert, Update, and Delete Rule**

A database must support high-level insertion, updation, and deletion. This must not be limited to a single row, that is, it must also support union, intersection and minus operations to return sets of data records.

- **Rule 8: Physical Data Independence**

The data stored in a database must be independent of the applications that access the database. Any change in the physical structure of a database must not have any impact on how the data is being accessed by external applications.

- **Rule 9: Logical Data Independence**

The logical data in a database must be independent of its user's view (application). Any change in logical data must not affect the applications using it. For example, if two tables are merged or one is split into two different tables, there should be no impact or change on the user application. This is one of the most difficult rule to apply.

- **Rule 10: Integrity Independence**

A database must be independent of the application that uses it. All its integrity constraints can be independently modified without the need of any change in the application. This rule makes a database independent of the front-end application and its interface.

- **Rule 11: Distribution Independence**

The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only. This rule has been regarded as the foundation of distributed database systems.

- **Rule 12: Non-Subversion Rule**

If a system has an interface that provides access to low-level records, then the interface must not be able to subvert the system and bypass security and integrity constraints.