CHAPTER - 5

Database Constraints and Relational Database Design

5.1 Introduction

Constraints are the rules enforced on data columns on table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database. Constraints could be column level or table level. Column level constraints are applied only to one column, whereas table level constraints are applied to the whole table.

In SQL, we have the following constraints:

- **NOT NULL** - Indicates that a column cannot store NULL value
- **UNIQUE** - Ensures that each row for a column must have a unique or different value
- **PRIMARY KEY** - A combination of a NOT NULL and UNIQUE. Ensures that a column (or combination of two or more columns) have an unique identity which helps to find a particular record in a table more easily and quickly
- **FOREIGN KEY** - Ensure the referential integrity of the data in one table to match values in another table
- **CHECK** - Ensures that the value in a column meets a specific condition
- **DEFAULT** - Specifies a default value when specified none for this column.
- **INDEX:** Use to create and retrieve data from the database very quickly.

Constraints can be specified when a table is created with the CREATE TABLE statement or you can use ALTER TABLE statement to create constraints even after the table is created.

SQL CREATE TABLE + CONSTRAINT Syntax

```
CREATE TABLE table_name
(
column_name1 data_type(size) constraint_name,
column_name2 data_type(size) constraint_name,
column_name3 data_type(size) constraint_name,
....
);
```

Dropping Constraints:

Any constraint that you have defined can be dropped using the ALTER TABLE command with the DROP CONSTRAINT option. For ***example***, to drop the primary key constraint in the EMPLOYEES table, we can use the following command:

```
ALTER TABLE EMPLOYEES DROP CONSTRAINT EMPLOYEES_PK;
```

Some implementations may provide shortcuts for dropping certain constraints. For ***example***, to drop the primary key constraint for a table in Oracle, we can use the following command:

```
ALTER TABLE EMPLOYEES DROP PRIMARY KEY;
```

ome implementations allow you to disable constraints. Instead of permanently dropping a constraint from the database, we may want to temporarily disable the constraint and then enable it later.

5.2 Integrity Constraints

Integrity constraints are those constraints in database system which guard against invalid database operations or accidental damage to the database, by ensuring that authorized changes to the database. It does not allow to loss of data consistency in database, it ensures database consistency. In fact, integrity constraints provide a way of ensuring that changes made to the database by authorized users do not result in a loss of data consistency.

Example of integrity constraints in E-R model

- Key declaration: candidate key, primary key
- Form of relationship: mapping cardinalities: one to one, one to many etc.

In database management system we can enforce any arbitrary predicate as integrity constraints but it adds overhead to the database system so its cost should be evaluated, as far as possible integrity constraint should with minimal overhead.

Domain Constraints

A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types). Declaring an attribute to be of a particular domain acts as a constraint on the values that it can take. Domain constraints are the most elementary form of integrity constraint. They are tested easily by the system whenever a new data item is entered into the database.

Domain types in SQL

SQL standard supports a variety of built in domain types including:

- Char (n): A fixed length character string with user specified length n.
- Varchar(n): A variable length string with user specified maximum length n.
- Int: An integer (Machine dependant).
- Smallint: A small integer.
- Numeric (p, d): A fixed point number with user specified precision. Where (.) is counted in p.
- Real, double precision: Floating point and double precision floating point numbers.
- Float (n): A floating point number with precision of at least n digits.
- Date: A calendar date containing a four digit year, month and day of the month.
- Time: The time of a day, in hours, minutes and seconds.
- Timestamp: A combination of date and time.

New domains can be created from existing data types. For e***xample:***

```
create domain Dollars numeric(12, 2)
      create domain Pounds numeric(12,2)
```

Note: The **check** clause in SQL allow domains to be restricted

Example: The domain constraint ensures that the hourly-rate must greater than 5000

```
create domain salary-rate  numeric(5)
constraint value-test check(value > = 5000)
```

The clause **constraint** *value-test is* optional but useful to indicate which constraint an update violated.

Example:

```
create domain AccountType char(10)
    constraint account-type-test
        check (value in ('Checking', 'Saving'))
```

Example:

```
create domain account-number char(10)
constraint account-number-null-test check(value not
null)
```

5.3 Referential Integrity

Referential integrity is a condition which ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.

Example: Consider two relation department and employee as follows

```
department(deptno#,dname)
employee(empno#,ename,deptno)
```

Deletion of particular department from department table also need to delete records of employees they belongs to that particular department or delete need not be allow if there is any employee that is associated to that particular department that we are going to delete. Any update made in `deptno` in department table `deptno` in employee must be updated automatically. This implies primary key acts as a referential integrity constraint in a relation. When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.

Formal Definition

Let $r_1(R_1)$ and $r_2(R_2)$ be relations with primary keys K_1 and K_2 respectively. The subset α of R_2 is a ***foreign key*** referencing K_1 in relation r_1, if for every t_2 in r_2 there must be a tuple t_1 in $r1$ such that $t_1[K_1] = t_2[\alpha]$. Referential integrity constraint also called subset dependency since its can be written as $\prod_{\alpha} (r_2) \subseteq \prod K_1 (r_1)$.

Database modification

The following tests must be made in order to preserve the following referential integrity constraint: $\prod_{\alpha} (r_2) \subseteq \prod_{K} (r_1)$

- **Insert:** If a tuple t_2 is inserted into r_2, the system must ensure that there is a tuple t_1 in r_1 such that $t_1[K] = t_2[\alpha]$. That is $t_2 [\alpha] \in \prod_K (r_1)$
- **Delete:** If a tuple, t_1 is deleted from r_1, the system must compute the set of tuples in r_2 that reference $t1$: $\sigma_{\alpha} = t_1[K] (r_2)$
- If this set is not empty i.e. either the delete command is rejected as an error, or the tuples that reference t_1 must themselves be deleted (cascading deletions are possible).
- **Update:** There are two cases:
  - If a tuple t_2 is updated in relation r_2 and the update modifies values for foreign key α, then a test similar to the insert case is made: Let t_2' denote the new value of tuple t_2. The system must ensure that $t_2'[\alpha] \in \prod K(r_1)$
  - If a tuple t_1 is updated in r_1, and the update modifies values for the primary key (K), then a test similar to the delete case is made:
    - The system must compute $\sigma\alpha = t_1[K] (r_2)$ using the old value of t_1 (the value before the update is applied).
    - If this set is not empty
      - the update may be rejected as an error, or
      - the update may be cascaded to the tuples in the set, or
      - the tuples in the set may be deleted.

Referential integrity in SQL

Using the SQL Create table statement we can enforce: Primary key, Unique and Foreign key.

Syntax:

```
create table account
(       . . .
     foreign key(branch-name) references branch
     on delete cascade
     on update cascade
. . . )
```

- **on delete cascade**: if a delete of a tuple in *branch* results referential-integrity constraint violation, it also delete tuples in relation account that refers to the branch that was deleted.
- **on update cascade:** if a update of a tuple in *branch* results referential-integrity constraint violation, it updates tuples in relation account that refers to the branch that was updated.

Example:

create table customer (customer-name char(20)**,** customer-street char(30), customer-citychar(30), **primary key** (customer-name))	**create table** branch (branch-name char(15)**,** branch-city char(30), assets integer, **primary key** (branch-name))
create table account (account-number char(10)**,** branch-name char(15), balance integer, **primary key** (account-number), **foreign key** (branch-name) **references** branch) *Cascading actions*	**create table** depositor (customer-name char(20)**,** account-number char(10)**,** **primary key** (customer-name, account-number), **foreign key** (account-number) **references** account, **foreign key** (customer-name) **references** customer)

Example:

```
create table Sale
(
     OrderID Dec( 7, 0 ) Not Null
     Constraint SaleOrderIdChk Check( OrderID > 0 ),
     SaleDate Date Not Null,
     ShipDate Date Default Null,
     SaleTot Dec( 7, 2 ) Not Null,
     CrdAutNbr Int Default Null,
     CustID Dec( 7, 0 ) Not Null,
     primary Key( OrderID ),
     constraint SaleCrdAutNbrUK Unique ( CrdAutNbr ),
     constraint SaleCustomerFK Foreign Key ( CustID )
     references Customer ( CustID )
     on Delete Cascade
     on Update Restrict,
     constraint SaleShipDateChk Check( ShipDate Is Null
     OR ShipDate >= SaleDate )
)
```

5.4 Assertions and Triggers

An assertion is a condition or predicate expressing a condition we wish the database to always satisfy. Domain constraints, functional dependency and referential integrity are special forms of assertion. However, there are many constraints that we cannot express by using only these special forms. For example, "Every department must have at least five courses offered every semester" must be expressed as an assertion. When an assertion is created, the system tests it for validity. If the assertion is valid, then any future modification to the database is allowed only if it does not cause that assertion to be violated. Assertion testing may introduce a significant amount of overhead, especially if the assertions are complex; hence assertions should be used with great care. General **syntax** for creating assertion in SQL is:

```
create assertion <assertion-name> check <predicate>
```

Example: sum of loan amounts for each branch is less than the sum of all account balances at the branch.

```
create assertion sum-constraint check
  (not exists (select * from branch
     where (select sum(amount) from loan
       where loan.branch-name = branch.branch-name)
         >= (select sum(amount) from account
            where loan.branch-name = branch.branch-name)))
```

Example: every customer must have minimum balance 1000 in an account who are loan holder

```
create assertion balance-constraint check
 (not exists (
  select * from loan
     where not exists (
     select *from borrower, depositor, account
     where loan.loan-number = borrower.loan-number
      and borrower.customer-name = depositor.customer-name
       and depositor.account-number = account.account-number
       and account.balance >= 1000)))
```

Trigger

A trigger is a statement that is automatically executed by the system as a side effect of a modification to the database. While writing a trigger we must specify *conditions under which the trigger is executed* and *actions to be taken when trigger executes*. Triggers are useful mechanism to perform certain task automatically when certain condition/s met. Sometime trigger is also called *rule* or *action rule*.

Syntax:

```
CREATE OR REPLACE TRIGGER <TRIGGER NAMR>
{BEFORE, AFTER}
{INSERT|DELETE|UPDATE [OF column, . .]} ON <table name>
[REFERENCING {OLD AS <old>, NEW AS <new>}]
[FOR EACH ROW [WHEN <condition>]]
DECLARE
Variable declaration;
BEGIN
 . . .
END;
```

Example: If employee salary increased by more than 10%, then increment rank field by 1.

```
Create or Replace Trigger EmpSal
Before Update Of salary On Employee
For Each Row
Begin
 IF (:new.salary > (:old.salary * 1.1)) Then
  :new.rank := :old.rank + 1; // The assignment operator has ":"
 End IF;
End;
/
```

Note:

- Make sure to have the "/" to run the command
- If the trigger exists, then drop it first using `replace trigger` command.
- Since we need to change values, then it should be "Before" event

Example: Suppose that instead of allowing negative account balances, the bank deals with overdrafts by setting the account balance to zero and creating a loan in the amount of the overdraft providing same loan number as an account number of the overdrawn account.

```
create trigger overdraft-trigger
after update on account
referencing new row as nrow
    for each row
       when nrow.balance < 0
begin
   insert into borrower
   (select customer-name, account-number
   from depositor
   where nrow.account_number =  depositor.account-number);
   insert into loan values
      (nrow.account_number, nrow.branch-name, nrow.balance);
   update account
      set balance = 0
      where account.account_number = nrow.account_number
end;
```

Note: Authorization

We may want to differentiate among the users as far as the type of access they are permitted on various data values in the database. These differentiations are expressed in terms of **authorization**, the most common being: **read authorization**, which allows reading, but not modification, of data; **insert authorization**, which allows insertion of new data, but not modification of existing data; **update authorization**, which allows modification, but not deletion, of data; and **delete authorization**, which allows deletion of data. We may assign the user all, none, or a combination of these types of authorization.

5.5 Normalization

The basic objective of normalization is to reduce redundancy which means that information is to be stored only once. Storing information several times leads to wastage of storage space, and increase in the total size of the data stored. Relations are normalized so that when relations in a database are to be altered during the lifetime of the database, we do not lose information or introduce inconsistencies. Normalization theory is based on: *Functional dependencies* and *Multi valued dependencies*. Normalization eliminate anomalies [update, delete & insert problem] that results from there redundancies. The type of alterations (*anomalies*) normally needed for relations are:

- **Update anomalies:** if data items are scattered and are not linked to each other properly, then there may be instances when we try to update one data item that has copies of it scattered at several places, few instances of it get updated properly while few are left with their old values. This leaves database in an inconsistent state.
- **Deletion anomalies:** we tried to delete a record, but deleting some data cause other information to be lost (i.e. one relation related to another).
- **Insert anomalies:** we tried to insert data in a record that does not exist at all.

Normalization is the process of removing redundant data from relational tables by decomposing (splitting) a relational table into smaller tables by projection. The goal is to have only primary keys on the left hand side of a functional dependency. In order to be correct, decomposition must be lossless. That is, the new tables can be recombined by a natural join to recreate the original table without creating any spurious or redundant data.

Properties of Normalized Relations

The idea of normalizing relations to higher and higher normal forms is to attain the goals of having a set of ideal relations meeting the above criteria. Ideal relations after normalization should have the following properties:

1. No data value should be duplicated in different rows unnecessarily.
2. A value must be specified (and required) for every attribute in a row.
3. Each relation should be self-contained. In other words, if a row from a relation is deleted, important information should not be accidentally lost.
4. When a row is added to a relation, other relations in the database should not be affected.
5. A value of an attribute in a tuple may be changed independently of other tuples in the relation and other relations.

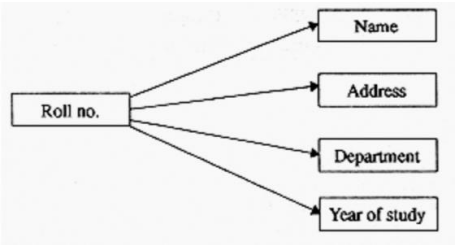
Functional Dependency

The concept of functional dependency (aka normalization) was introduced by professor Codd in 1970 when he defined the first three normal forms (first, second and third normal forms). Functional dependencies are constraints on the set of legal relations. It defines attributes of relation, how they are related to each other. It determines unique value for a certain set of attributes to the value for another set of attributes that is functional dependency is a generalization of the notation of key. Functional dependencies are interrelationship among attributes of a relation. There is no fool-proof algorithmic method of identifying dependency. We have to use our commonsense and judgment to specify dependencies.

Definition

For a given relation R with attribute X and Y, Y is said to be functionally dependent on X, if given value for each X uniquely determines the value of the attribute in Y. X is called determinant of the functional dependency (FD) and functional dependency denoted by X→ Y. For ***example***, given the value of item code, there is only one value of item name for it. Thus item name is functionally dependent on item code. This is shown as: `Item code → item name`

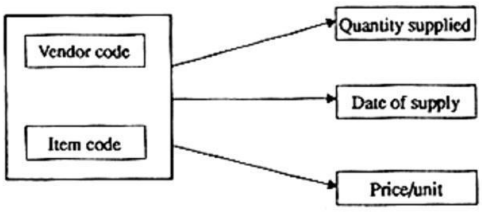
Example: Functional dependency on relation student using attribute *Roll no.*



Functional dependency may also be based on a composite attribute. For example, if we write: X, Z → Y; It means that there is only one value of Y corresponding to given values of X, Z. In other words, Y is functionally dependent on the composite X, Z. In ***example*** below, for example, Order no., and Item code together determine Qty. and Price. Thus: `Order no., Item code → Qty., Price`

Order no.	Order date	Item code	Quantity	Price/unit
1456	260289	3687	52	50.40
1456	260289	4627	38	60.20
1456	260289	3214	20	17.50

Example: In the relation "Supplies" (Vendor code, Item code, Qty supplied, Date of supply, Price/unit) Vendor code and Item code together form the key.



Example: consider a relation *supplier*: Supplier(supplier_id#, sname, status,city)
Here, `sname`, status and city are functionally dependent on `supplier_id`. Meaning is that each `supplier_id` uniquely determines the value of attributes supplier `name`, `supplier status` and `city`. This can be express by

`Supplier.supplier_id→supplier.sname`
`Supplier.supplier_id→supplier.status`
`Supplier.supplier_id→supplier.city`
	OR	`supplier_id→ sname`
		`supplier_id→ status`
		`supplier_id→city`

Is functional dependency valid?	Valid case: sname unique	Invalid case: sname not unique
sname→status sname→city	*sname* | *status* X | Good Y | Good	*sname* | *status* X | Good Y | Good X | Bad

Armstrong's Axioms
If F is set of functional dependencies then the closure of F, denoted as F^+, is the set of all functional dependencies logically implied by F. Armstrong's Axioms are set of rules, when applied repeatedly generates closure of functional dependencies.

- **Reflexive rule:** If alpha is a set of attributes and beta is_subset_of alpha, then alpha holds beta.
- **Augmentation rule:** if a → b holds and y is attribute set, then ay → by also holds. That is adding attributes in dependencies, does not change the basic dependencies.
- **Transitivity rule:** Same as transitive rule in algebra, if a → b holds and b → c holds then a → c also hold. a → b is called as a functionally determines b.

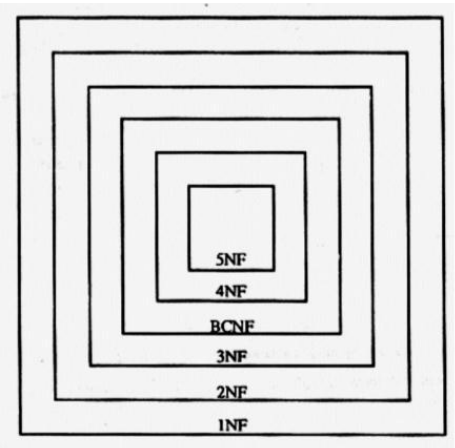
Trivial Functional Dependency

- **Trivial:** If an FD X → Y holds where Y subset of X, then it is called a trivial FD. Trivial FDs are always hold.
- **Non-trivial:** If an FD X → Y holds where Y is not subset of X, then it is called non-trivial FD.
- **Completely non-trivial:** If an FD X → Y holds where x intersect Y = Φ, is said to be completely non-trivial FD.

Application of Functional dependencies
Functional dependencies are applicable to:

1. To test the relation whether they are legal under a given set of functional dependency. For ***example:*** Let r is a relation and F is a given set of functional dependencies. If r satisfies F, then we determine that r is legal under a given set of functional dependency F.
2. To specify the constraints for the legal relation. For ***example:*** We say that f holds on R if all legal relations on R satisfy the set of functional dependencies F.

5.6 Normal Forms



Normalization is the process of reducing redundant data / information in the database by decomposing the long relation. It is an approach for designing reliable database system. Normalization theory is built under the concept of the normal form. There are several normal form called

- First Normal Form. (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)
- Boyce Codd Normal Form (BCNF)
- Fourth Normal Form (4NF)
- Fifth Normal Form (5NF)

A relation is said to in particular normal form if it satisfy the set of this particular normal form's constraints. In practical, third normal form is sufficient to design reliable database system.

A. First Normal Form

A relation is in first normal form if the domain of each attribute contains only atomic values, and the value of each attribute contains only a single value from that domain i.e. they contain no repeating values.

For ***example***: The sample data taken from any programming institute is:

Course	Content
Programming	Java, c++
Web	HTML, PHP, ASP

We re-arrange the relation (table) as below, to convert it to First Normal Form. Each attribute must contain only single value from its pre-defined domain.

Course	Content
Programming	Java
Programming	c++
Web	HTML
Web	PHP
Web	ASP

Major *drawback* of 1NF is data redundancy. Redundancy causes what are called *update anomalies*. Update anomalies are problems that arise when information is inserted, deleted, or updated.

B. Second Normal Form

Before we learn about second normal form, we need to understand the following:

- **Prime attribute:** an attribute, which is part of prime-key or candidate key, is prime attribute.
- **Non-prime attribute:** an attribute, which is not a part of prime-key or candidate key, is said to be a non-prime attribute.

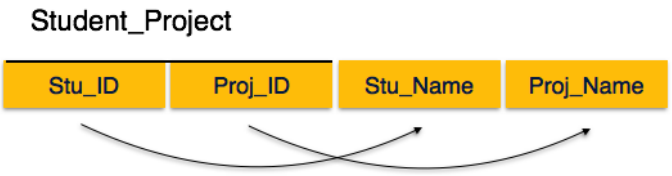
Second normal form says, that every non-prime attribute should be fully functionally dependent on prime key attribute. That is, if $X \rightarrow A$ holds, then there should not be any proper subset Y of X, for that $Y \rightarrow A$ also holds.

Put simply, a table is in 2NF if and only if it is in 1NF and every non-prime attribute of the table is dependent on the whole of a candidate key. A non-prime attribute of a table is an attribute that is not a part of any candidate key of the table.

The process for transforming a 1NF table to 2NF is:

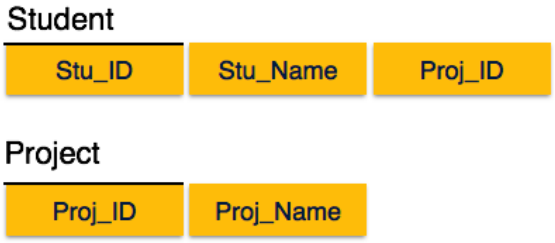
1. Identify any determinants other than the composite key, and the columns they determine.
2. Create and name a new table for each determinant and the unique columns it determines.
3. Move the determined columns from the original table to the new table. The determinate becomes the primary key of the new table.
4. Delete the columns you just moved from the original table except for the determinate which will serve as a foreign key.
5. The original table may be renamed to maintain semantic meaning.

For ***example:*** In *Student_Project* relation that the prime key attributes are *Stu_ID* and *Proj_ID*. According to the rule, non-key attributes, i.e. *Stu_Name* and *Proj_Name* must be dependent upon both and not on any of the prime key attribute individually.



But we find that *Stu_Name* can be identified by *Stu_ID* and *Proj_Name* can be identified by *Proj_ID* independently. This is called partial dependency, which is not allowed in Second Normal Form.

We broke the relation in two as depicted in the above picture. So there exists no partial dependency.



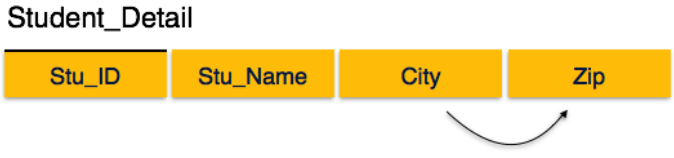
C. Third Normal Form

Third normal form is the third step in normalizing a database design to reduce the duplication of data and ensure *referential integrity* by ensuring that the entity is in second normal form and all the attributes in a table are dependent on the primary key and only the primary key.

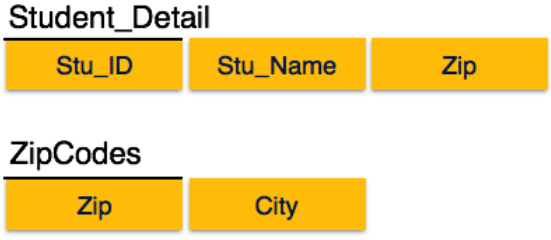
A relational table is in third normal form (3NF) if it is already in 2NF and every non-key column is none transitively dependent upon its primary key. In other words, all non-key or non-prime attributes are functionally dependent only upon the primary key.

The advantage of having relational tables in 3NF is that it eliminates redundant data which in turn saves space and reduces manipulation anomalies (INSERT and DELETE).

For example: In *Student_detail* relation, *Stu_ID* is key and only prime key attribute. We find that City can be identified by *Stu_ID* as well as *Zip* itself. Neither *Zip* is a superkey nor *City* is a prime attribute. Additionally, *Stu_ID* → *Zip* → *City*, so there exists ***transitive dependency***.



We broke the relation as above depicted two relations to bring it into 3NF.



For ***example:*** Consider a relation 'course_detail'

Course_detail=(#course,prof,#room,enroll_limit)

There is a transitive dependency, course→room→enroll_limit). We can eliminate this transitive dependency by decomposing the relation 'course_detail' into the following relations.

Now:

Given relation in 1NF

Course_detail=(#course,prof,room,room_capacity,enroll_limit)

FDs={course→(prof,room,room_capacity,enroll_limit),room→room_capacity}

Where, FDs = Functional dependencies

Relations in 2NF

Course_detail=(#course,prof,room,emroll_limit)

FDs={course→prof,course→room,course→enroll_limit,room→enroll_limit}

Room_detail=(#room,room_capacity)

Fds={room→room_capacity}

Relations in 3NF

Course_prof_info=(#course.prof,enroll_limit)

FDs={course→prof, course→enroll_limit}

Course_room_info=(course,room)

No FDs

Room_detail=(#room,room_capacity)

FDs={room→eoom_capacity}

D. Boyce Codd Normal Form (BCNF)

BCNF (or 3.5NF) is a normal form used in database normalization. It is a slightly stronger version of the third normal form (3NF). BCNF was developed in 1974 by Raymond F. Boyce and Edgar F. Codd to address certain types of anomaly not dealt with by 3NF as originally defined.

If a relational schema is in BCNF then all redundancy based on functional dependency has been removed, although other types of redundancy may still exist. A relational schema R is in Boyce–Codd normal form if and only if for every one of its dependencies $X \rightarrow Y$, at least one of the following conditions hold:

- $X \rightarrow Y$ is a trivial functional dependency (Y ⊆ X)
- X is a superkey for schema R & X called determinant of functional dependency X→R)

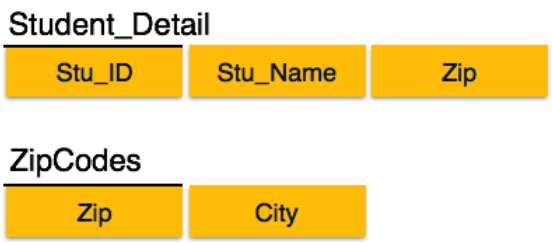
For ***example:*** Stu_ID is super-key in Student_Detail relation and Zip is super-key in ZipCodes relation. So,

`Stu_ID → Stu_Name, Zip`

And

`Zip → City`

Confirms, that both relations are in BCNF.



Example: Let us consider the relation schema "grade" which is in 3NF.

Grade=(#name,student_id,course,grade)

Assume that, each student has unique name and unique student_id then set of FD

FDs={name,course →grade, student_id,course→grade,name→student_id,student_id→name}

Here this relation has two candidate keys {name,course} and {student_id,course}. Each of those composite key has common attribute course. Here the relation grade is not in BCNF because the dependencies

{student_id→name} and {name→student_id} are non-trivial and there determinants are not superkey of relation grade.

Drawback of this relation (which is not 3NF)

- ***Data redundancy:*** The association between name and corresponding student_id are repeated.
- ***Update problem:*** Any changes in one of these attribute value (name or student_id) has to reflected all tuples; otherwise there will be inconsistency in the database.
- ***Insertion problem:*** The student_id can not be associated with the student name unless the student has registered in a course.
- ***Deletion problem:*** The association between student_id and student name is lost if the student deletes all courses he/she is reistered in.

Solution:

student_info=(#student_id,name)

grade(#student_id,course,grade)

These problems of relation schema in 3NF occur since relation may have overlapping candidate keys. BCNF removes this problem. So it is stronger than 3NF.

E. Fourth normal form (4NF)

4NF is a normal form used in database normalization. Introduced by Ronald Fagin in 1977, 4NF is the next level of normalization after Boyce–Codd normal form (BCNF). Whereas the second, third, and Boyce–Codd normal forms are concerned with functional dependencies, 4NF is concerned with a more general type of dependency known as a multivalued dependency.

- A Table is in 4NF if and only if, for every one of its non-trivial multivalued dependencies $X \twoheadrightarrow Y$, X is a superkey—that is, X is either a candidate key or a superset thereof
- A relation is in 4NF if it is in Boyce-Codd normal form and contains no nontrivial multi-valued dependencies.

Multivalued dependencies

It is a type of functional dependency where the determinant can determine more than one value. Multi-valued dependency (MVD) represents a dependency among at least three attributes (for example, A, B and C) in a relation, such that for each value of A there is a set of values for B and a set of value for C. However, the set of values for B and C are independent of each other.

A multi-valued dependency can be further defined as being trivial or nontrivial. A MVD A $\twoheadrightarrow$ B in relation R is defined as being trivial *if B is a subset of A* or *A U B = R*. A MVD is defined as being nontrivial if neither of the above two conditions is satisfied.

For ***example:***
Student has one or more majors; i.e. `StudentID ↠ Majors`
Student can participate in one or more activities; i.e. `StudentID ↠ Activities`

Example:

No multi value dependency

Eid	Language	Skill
100	English	Teaching
100	Kurdish	Politic
100	French	cooking
200	English	cooking
200	Arabic	Singing

No multi value dependency, therefore R is in fourth normal form.

Multi-value dependency

Eid	Language	Skill
100	English	Teaching
100	Kurdish	Politic
100	English	politic
100	Kurdish	Teaching
200	Arabic	Singing

Since:
`Eid ---->> Languages`
`Eid ------>> Skills`
Languages and Skills are dependent and hence multivalued dependency.

Note:

- A ***trivial multivalued dependency*** $X \twoheadrightarrow Y$ is one where either *Y* is a subset of *X*, or *X* and *Y* together form the whole set of attributes of the relation.
- A ***functional dependency*** is a special case of multivalued dependency. In a functional dependency $X \rightarrow Y$, every *x* determines *exactly one y*, never more than one.
- **Fifth Normal Form (5NF):** It eliminates a lossless-join dependency.

5.7 Decomposition of relational schema

The idea of decomposition is break down large and complicated relation in no. of simple and small relations which minimized data redundancy. It can be consider principle to solve the relational model problem.

Definition

The decomposition of relation schema R= (A_1, A_2, . ,An) is a set of relation schema { R_1, R_2, ---- - R_m}, such that $R_i \subseteq R \ \forall \ 1 \leq i \leq m$ and $R_1 \bigcup R_2 \bigcup \ . \ . \bigcup R_m = R$.

- All attributes of an original schema (R) must appear in the decomposition (R_1, R_2).
- $R= R_1 \bigcup R_2$. if $R \neq R_1 \bigcup R_2$ then such decomposition called lossey join decomposition.
- $R \neq \prod_{R1}(R) \bowtie \prod_{R2}(R)$. Decomposition should ***lossless join decomposition***.

Example: Consider the relation schema to store the information a student maintain by the university. **Student_info** (#name, course, phone_no, major, prof, grade)

name	**course**	**phone_no**	**major**	**prof**	**grade**
John	353	374537	Computer Science	Smith	A
Scott	329	427993	Mathematics	James	S
John	328	374537	Computer Science	Adams	A
Allen	432	729312	Physics	Blake	C
Turner	523	252731	Chemistry	Miller	B
John	320	374537	Computer Science	Martin	A
Scott	328	727993	Mathematics	Ford	B

Problems: Redundancy, complicates updating, complicate insertion, and deletion problem. The problems in this relation schema student_info can be resolved if we it with the following relation schemas.

```
Students (#name, phone_no, major)
Transcript (#name, course, grade)
Teacher (course, prof)
```

- Here, first relation schema ***'Student'*** gives the phone number and major subject of each student such information will be stored only once for each student. Thus, any changes in the phone number will thus require changes in only one tuple of this relation
- The second relation schema ***'Transcript'*** stores the grade of each student in each course. So, to insert information about student phone number, major and the course teaches by which the professor.
- Third relation schema records the teacher of each course.

What is the problem of such of such decomposition?

- One of the disadvantages of original relation schema 'student_info' with these true relational schemas is that retrieval of certain information required to performed natural join operation.
- We know that to resolve the problem of bad relational database design we need to decompose relation but the problem is that how to decompose relation. That is, we require to process decomposition of relation that may give good relational database. Normalization gives the approach for designing the best relational database under the five normal forms.