M.Sc. (Computer Science)
UNIT – III
DATE: 26-03-2020

Functional Dependencies

In relational database theory, a **functional dependency** is a **constraint** between two sets of attributes in a relation from a database. In other words, functional dependency is a constraint that describes the relationship between attributes in a relation.

Given a relation *R*, a set of attributes *X* in *R* is said to **functionally determine** another set of attributes *Y*, also in *R*, (written $X \rightarrow Y$) if, and only if, each *X* value in *R* is associated with precisely one *Y* value in *R*; *R* is then said to *satisfy* the functional dependency $X \rightarrow Y$. Equivalently,

the projection is a function, i.e. *Y* is a function of *X*.[1][2] In simple words, if the values for the *X* attributes are known (say they are *x*), then the values for the *Y* attributes corresponding to *x* can be determined by looking them up in *any* tuple of *R* containing *x*. Customarily *X* is called the *determinant* set and *Y* the *dependent* set. A functional dependency FD: $X \rightarrow Y$ is called *trivial* if *Y* is a subset of *X*.

In other words, a dependency FD: $X \rightarrow Y$ means that the values of *Y* are determined by the values of *X*. Two tuples sharing the same values of *X* will necessarily have the same values of *Y*.

The determination of functional dependencies is an important part of designing databases in the relational model, and in database normalization and de-normalization. A simple application of functional dependencies is **Heath's theorem**; it says that a relation *R* over an attribute set *U* and satisfying a functional dependency $X \rightarrow Y$ can be safely split in two relations having the lossless-join decomposition property, namely into where $Z = U - XY$ are the rest of the attributes. (Unions of attribute sets are customarily denoted by mere juxtapositions in database theory.) An important notion in this context is a candidate key, defined as a minimal set of attributes that functionally determine all of the attributes in a relation. The functional dependencies, along with the attribute domains, are selected so as to generate constraints that would exclude as much data inappropriate to the user domain from the system as possible.

Basic Structure of SQL

1. Basic structure of an SQL expression consists of **select, from** and **where** clauses.
    - **select** clause lists attributes to be copied - corresponds to relational algebra **project**.
    - **from** clause corresponds to Cartesian product - lists relations to be used.
    - **where** clause corresponds to selection predicate in relational algebra.
2. Typical query has the form
3. **select** $A_1, A_2, \ldots, A_n$
4.
5. **from** $r_1, r_2, \ldots, r_m$
6.
7. **where** *P*

8. where each A_i represents an attribute, each r_i a relation, and *P* is a predicate.
9. This is equivalent to the relational algebra expression

$$\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

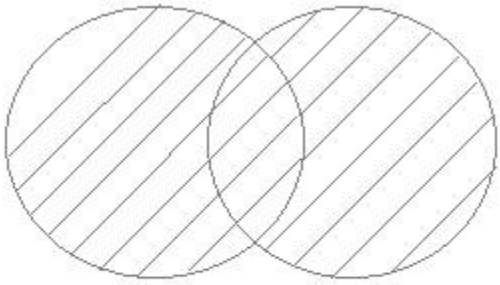
   - If the where clause is omitted, the predicate *P* is true.
   - The list of attributes can be replaced with a * to select all.
   - SQL forms the Cartesian product of the relations named, performs a selection using the predicate, then projects the result onto the attributes named.
   - The result of an SQL query is a relation.
   - SQL may internally convert into more efficient expressions.

Set Operation in SQL

SQL supports few Set operations to be performed on table data. These are used to get meaningful results from data, under different special conditions.

Union

UNION is used to combine the results of two or more Select statements. However it will eliminate duplicate rows from its result set. In case of union, number of columns and datatype must be same in both the tables.



Example of UNION

The **First** table,

ID	Name
1	abhi
2	adam

The **Second** table,

ID	Name

2	adam
3	Chester

Union SQL query will be,

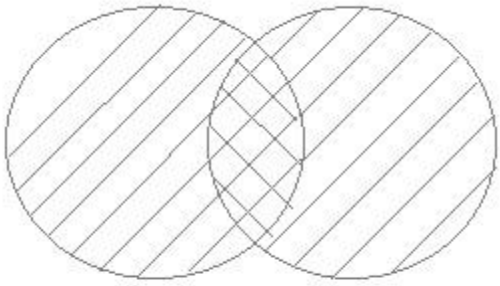
```
select * from First
UNION
select * from second
```

The result table will look like,

ID	NAME
1	abhi
2	adam
3	Chester

Union All

This operation is similar to Union. But it also shows the duplicate rows.



Example of Union All

The **First** table,

ID	NAME
1	abhi
2	adam

The **Second** table,

ID	NAME

2	adam
3	Chester

Union All query will be like,

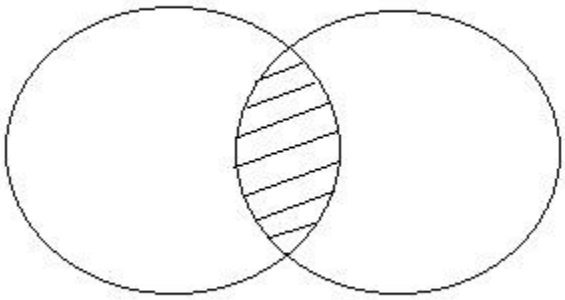
```
select * from First
UNION ALL
select * from second
```

The result table will look like,

ID	NAME
1	abhi
2	adam
2	adam
3	Chester

Intersect

Intersect operation is used to combine two SELECT statements, but it only retuns the records which are common from both SELECT statements. In case of **Intersect** the number of columns and datatype must be same. MySQL does not support INTERSECT operator.



Example of Intersect

The **First** table,

ID	NAME
1	abhi
2	adam

The **Second** table,

ID	NAME
2	adam
3	Chester

Intersect query will be,

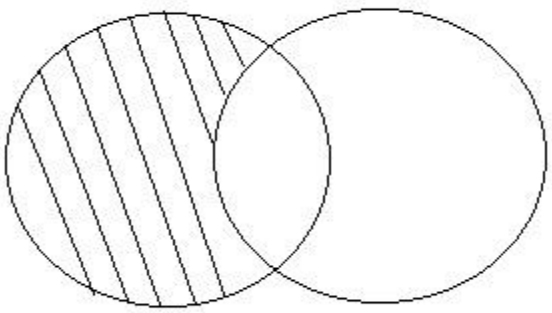
```
select * from First
INTERSECT
select * from second
```

The result table will look like

ID	NAME
2	adam

Minus

Minus operation combines result of two Select statements and return only those result which belongs to first set of result. MySQL does not support INTERSECT operator.



Example of Minus

The **First** table,

ID	NAME
1	abhi
2	adam

The **Second** table,

ID	NAME

2	adam
3	Chester

Minus query will be,

```
select * from First
MINUS
select * from second
```

The result table will look like,

ID	NAME
1	abhi

SQL has many built-in functions for performing calculations on data.

SQL Aggregate Functions

SQL aggregate functions return a single value, calculated from values in a column.

Function	Description
AVG()	Returns the average value
COUNT()	Returns the number of rows
FIRST()	Returns the first value
LAST()	Returns the last value
MAX()	Returns the largest value
MIN()	Returns the smallest value
ROUND()	Rounds a numeric field to the number of decimals specified
SUM()	Returns the sum

What is a NULL Value?

A field with a NULL value is a field with no value.

If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.

Note: It is very important to understand that a NULL value is different from a zero value or a field that contains spaces. A field with a NULL value is one that has been left blank during record creation!

How to Test for NULL Values?
It is not possible to test for NULL values with comparison operators, such as =, <, or <>.
We will have to use the IS NULL and IS NOT NULL operators instead.

IS NULL Syntax

SELECT ***column_names***
FROM ***table_name***
WHERE ***column_name*** **IS NULL;**

IS NOT NULL Syntax

SELECT ***column_names***
FROM ***table_name***
WHERE ***column_name*** **IS NOT NULL;**

SQL CREATE VIEW Statement

In SQL, a view is a virtual table based on the result-set of an SQL statement.
A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.
You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.

CREATE VIEW Syntax

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Note: A view always shows up-to-date data! The database engine recreates the data, using the view's SQL statement, every time a user queries a view.

SQL CREATE VIEW Examples
If you have the Northwind database you can see that it has several views installed by default.
The view "Current Product List" lists all active products (products that are not discontinued) from the "Products" table. The view is created with the following SQL:

```
CREATE VIEW [Current Product List] AS
SELECT ProductID, ProductName
FROM Products
WHERE Discontinued = No;
```

What is a SQL join?

A SQL join is a Structured Query Language (**SQL**) instruction to combine data from two sets of data (e.g. two tables). Before we dive into the details of a SQL join, let's briefly discuss what SQL is, and why someone would want to perform a SQL join.

SQL is a special-purpose programming language designed for managing information in a relational database management system (**RDBMS**). The word relational here is key; it specifies that the database management system is organized in such a way that there are clear relations defined between different sets of data.

Relational Database Example

Imagine you're running a store and would like to record information about your customers and their orders. By using a relational database, you can save this information as two tables that represent two distinct entities: customers and orders.

Customers

customer_id	first_name	last_name	email	address	city	state	zip
1	George	Washington	gwashington@usa.gov	3200 Mt Vernon Hwy	Mount Vernon	VA	22121
2	John	Adams	jadams@usa.gov	1250 Hancock St	Quincy	MA	02169
3	Thomas	Jefferson	tjefferson@usa.gov	931 Thomas Jefferson Pkwy	Charlottesville	VA	22902
4	James	Madison	jmadison@usa.gov	11350 Constitution Hwy	Orange	VA	22960
5	James	Monroe	jmonroe@usa.gov	2050 James Monroe Parkway	Charlottesville	VA	22902

Here, information about each customer is stored in its own row, with columns specifying different bits of information, like their first name, last name, and email address. Additionally, we associate a unique customer number, or primary key, with each customer record.

Orders

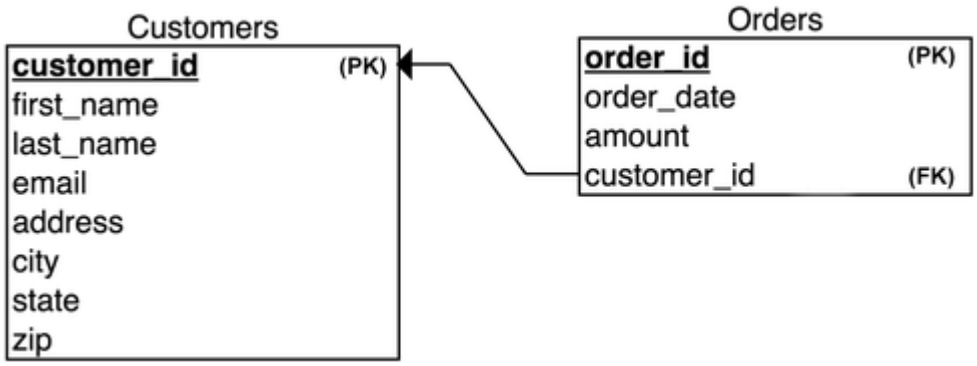
order_id	order_date	amount	customer_id
1	07/04/1776	$234.56	1
2	03/14/1760	$78.50	3
3	05/23/1784	$124.00	2

4	09/03/1790	$65.50	3
5	07/21/1795	$25.50	10
6	11/27/1787	$14.40	9

Again, each row contains information about a specific order. Each order has its own unique identification key order_idassigned to it as well.

Relational Model

You've probably noticed that these two examples share similar information. You can see these simple relations diagrammed below:



Note that the orders table contains two keys: one for the order and one for the customer who placed that order. In scenarios when there are multiple keys in a table, the key that refers to the entity being described in that table is called the **Primary Key** (PK) and other key is called a **Foreign Key** (FK).

DDL, DML

In our example, order_id is a primary key in the orders table, while customer_id is both a primary key in the customers table, and a foreign key in the orders table. Primary and foreign keys are essential to describing relations between the tables, and in performing SQL joins.

Basic SQL statements: DDL and DML

In the first part of this tutorial, you've seen some of the SQL statements that you need to start building a database. This page gives you a review of those and adds several more that you haven't seen yet.

• SQL statements are divided into two major categories: **data definition language (DDL)** and **data manipulation language (DML)**. Both of these categories contain far more statements than we can present here, and each of the statements is far more complex than we show in this introduction. If you want to master this material, we strongly recommend that you find a SQL reference for your own database software as a supplement to these pages.

Data definition language

DDL statements are used to build and modify the structure of your tables and other objects in the database. When you execute a DDL statement, it takes effect immediately.

• The create table statement does exactly that:

CREATE TABLE <table name> (
<attribute name 1> <data type 1>,
...
<attribute name n> <data type n>);

The **data types** that you will use most frequently are character strings, which might be called VARCHAR or CHAR for variable or fixed length strings; numeric types such as NUMBER or INTEGER, which will usually specify a precision; and DATE or related types. Data type syntax is variable from system to system; the only way to be sure is to consult the documentation for your own software.

• The alter table statement may be used as you have seen to specify primary and foreign key constraints, as well as to make other modifications to the table structure. Key constraints may also be specified in the CREATE TABLE statement.

ALTER TABLE <table name>
ADD CONSTRAINT <constraint name> PRIMARY KEY (<attribute list>);

You get to specify the constraint name. Get used to following a convention of tablename_pk (for example, Customers_pk), so you can remember what you did later. The attribute list contains the one or more attributes that form this PK; if more than one, the names are separated by commas.

• The foreign key constraint is a bit more complicated, since we have to specify both the FK attributes in this (child) table, and the PK attributes that they link to in the parent table.

ALTER TABLE <table name>
ADD CONSTRAINT <constraint name> FOREIGN KEY (<attribute list>)
REFERENCES <parent table name> (<attribute list>);

Name the constraint in the form childtable_parenttable_fk (for example, Orders_Customers_fk). If there is more than one attribute in the FK, all of them must be included (with commas between) in both the FK attribute list and the REFERENCES (parent table) attribute list.

You need a separate foreign key definition for each relationship in which this table is the child.

• If you totally mess things up and want to start over, you can always get rid of any object you've created with a drop statement. The syntax is different for tables and constraints.

DROP TABLE **<table name>;**

ALTER TABLE <table name>
DROP CONSTRAINT **<constraint name>;**

This is where consistent constraint naming comes in handy, so you can just remember the PK or FK name rather than remembering the syntax for looking up the names in another table. The DROP TABLE statement gets rid of its own PK constraint, but won't work until you separately drop any FK constraints (or child tables) that refer to this one.

It also gets rid of all data that was contained in the table—and it doesn't even ask you if you really want to do this!

• All of the information about objects in your schema is contained, not surprisingly, in a set of tables that is called the **data dictionary**. There are hundreds of these tables most database systems, but all of them will allow you to see information about your own tables, in many cases with a graphical interface. How you do this is entirely system-dependent.

Data manipulation language

DML statements are used to work with the data in tables. When you are connected to most multi-user databases (whether in a client program or by a connection from a Web page script), you are in effect working with a private copy of your tables that can't be seen by anyone else until you are finished (or tell the system that you are finished). You have already seen the SELECT statement; it is considered to be part of DML even though it just retreives data rather than modifying it.

• The insert statement is used, obviously, to add new rows to a table.

INSERT INTO <table name>
VALUES (<value 1>, ... <value n>);

The comma-delimited list of values must match the table structure exactly in the number of attributes and the data type of each attribute. Character type values are always enclosed in single quotes; number values are never in quotes; date values are often (but not always) in the format 'yyyy-mm-dd' (for example, '2006-11-30').

Yes, you will need a separate INSERT statement for every row.

• The update statement is used to change values that are already in a table.

UPDATE <table name>
SET <attribute> = <expression>
WHERE <condition>;

The update expression can be a constant, any computed value, or even the result of a SELECT statement that returns a single row and a single column. If the WHERE clause is omitted, then the specified attribute is set to the same value in every row of the table (which is usually not what you want to do). You can also set multiple attribute values at the same time with a comma-delimited list of attribute=expression pairs.

• The **delete** statement does just that, for rows in a table.

DELETE FROM <table name>
WHERE <condition>;

If the WHERE clause is omitted, then every row of the table is deleted (which again is usually not what you want to do)—and again, you will not get a "do you really want to do this?" message.

• If you are using a large multi-user system, you may need to make your DML changes visible to the rest of the users of the database. Although this might be done automatically when you log out, you could also just type:

COMMIT;

• If you've messed up your changes in this type of system, and want to restore your private copy of the database to the way it was before you started (this only works if you haven't already typed COMMIT), just type:

ROLLBACK;

Although single-user systems don't support **commit** and **rollback** statements, they are used in large systems to control **transactions**, which are sequences of changes to the database. Transactions are frequently covered in more advanced courses.

DDL

Data Definition Language (DDL) statements are used to define the database structure or schema. Some examples:

- CREATE - to create objects in the database
- ALTER - alters the structure of the database
- DROP - delete objects from the database
- TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed
- COMMENT - add comments to the data dictionary
- RENAME - rename an object

DML

Data Manipulation Language (DML) statements are used for managing data within schema objects. Some examples:

- SELECT - retrieve data from the a database
- INSERT - insert data into a table
- UPDATE - updates existing data within a table
- DELETE - deletes all records from a table, the space for the records remain
- MERGE - UPSERT operation (insert or update)
- CALL - call a PL/SQL or Java subprogram
- EXPLAIN PLAN - explain access path to data
- LOCK TABLE - control concurrency

DCL

Data Control Language (DCL) statements. Some examples:

- GRANT - gives user's access privileges to database
- REVOKE - withdraw access privileges given with the GRANT command

TCL

Transaction Control (TCL) statements are used to manage the changes made by DML statements. It allows statements to be grouped together into logical transactions.

- COMMIT - save work done
- SAVEPOINT - identify a point in a transaction to which you can later roll back
- ROLLBACK - restore database to original since the last COMMIT
- SET TRANSACTION - Change transaction options like isolation level and what rollback segment to use

Assertions

1. An **assertion** is a predicate expressing a condition we wish the database to always satisfy.
2. Domain constraints, functional dependency and referential integrity are special forms of assertion.
3. Where a constraint cannot be expressed in these forms, we use an assertion, e.g.
    - Ensuring the sum of loan amounts for each branch is less than the sum of all account balances at the branch.
    - Ensuring every loan customer keeps a minimum of $1000 in an account.
4. An assertion in DQL-92 takes the form,
5. **create assertion** assertion-name **check** predicate
6.
7. Two assertions mentioned above can be written as follows.
   Ensuring the 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.bname = branch.bname* >=

   (**select sum**)*amount*) **from** *account*

   where (*account.bname = branch.bname*)))

8. Ensuring every loan customer keeps a minimum of $1000 in an account.
9. **create assertion** *balance-constraint* **check**
10.
11. (**not exists** (**select * from** *loan L*
12.
13. (**where not exists** (**select ***
14.
15. **from** *borrower B, depositor D, account A*

16.
17. **where** *L.loan# = B.loan#* **and** *B.cname = D.cname*
18.
19. **and** *D.account# = A.account#*
20. **and** *A.balance* >= 1000)))
21.
22. When an assertion is created, the system tests it for validity.
    If the assertion is valid, any further modification to the database is allowed only if it does not cause that assertion to be violated.
    This testing may result in significant overhead if the assertions are complex. Because of this, the **assert** should be used with great care.
23. Some system developer omits support for general assertions or provides specialized form of assertions that are easier to test.

Trivial Functional Dependency

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

Normalization in DBMS: 1NF, 2NF, 3NF and BCNF in Database

Normalization is a process of organizing the data in database to avoid data redundancy, insertion anomaly, update anomaly & deletion anomaly. Let's discuss about anomalies first then we will discuss normal forms with examples.

Anomalies in DBMS

There are three types of anomalies that occur when the database is not normalized. These are – Insertion, update and deletion anomaly. Let's take an example to understand this.

Example: Suppose a manufacturing company stores the employee details in a table named employee that has four attributes: emp_id for storing employee's id, emp_name for storing employee's name, emp_address for storing employee's address and emp_dept for storing the department details in which the employee works. At some point of time the table looks like this:

emp_id	emp_name	emp_address	emp_dept
101	Rick	Delhi	D001
101	Rick	Delhi	D002

123	Maggie	Agra	D890
166	Glenn	Chennai	D900
166	Glenn	Chennai	D004

The above table is not normalized. We will see the problems that we face when a table is not normalized.

Update anomaly: In the above table we have two rows for employee Rick as he belongs to two departments of the company. If we want to update the address of Rick then we have to update the same in two rows or the data will become inconsistent. If somehow, the correct address gets updated in one department but not in other then as per the database, Rick would be having two different addresses, which is not correct and would lead to inconsistent data.

Insert anomaly: Suppose a new employee joins the company, who is under training and currently not assigned to any department then we would not be able to insert the data into the table if emp_dept field doesn't allow nulls.

Delete anomaly: Suppose, if at a point of time the company closes the department D890 then deleting the rows that are having emp_dept as D890 would also delete the information of employee Maggie since she is assigned only to this department.
To overcome these anomalies we need to normalize the data. In the next section we will discuss about normalization.

Normalization

Here are the most commonly used normal forms:

- First normal form(1NF)
- Second normal form(2NF)
- Third normal form(3NF)
- Boyce & Codd normal form (BCNF)

First normal form (1NF)

As per the rule of first normal form, an attribute (column) of a table cannot hold multiple values. It should hold only atomic values.

Example: Suppose a company wants to store the names and contact details of its employees. It creates a table that looks like this:

emp_id	emp_name	emp_address	emp_mobile
101	Herschel	New Delhi	8912312390
102	Jon	Kanpur	8812121212 9900012222

103	Ron	Chennai	7778881212
104	Lester	Bangalore	9990000123 8123450987

Two employees (Jon & Lester) are having two mobile numbers so the company stored them in the same field as you can see in the table above.

This table is **not in 1NF** as the rule says "each attribute of a table must have atomic (single) values", the emp_mobile values for employees Jon & Lester violates that rule.

To make the table complies with 1NF we should have the data like this:

emp_id	emp_name	emp_address	emp_mobile
101	Herschel	New Delhi	8912312390
102	Jon	Kanpur	8812121212
102	Jon	Kanpur	9900012222
103	Ron	Chennai	7778881212
104	Lester	Bangalore	9990000123
104	Lester	Bangalore	8123450987

Second normal form (2NF)

A table is said to be in 2NF if both the following conditions hold:

- Table is in 1NF (First normal form)
- No non-prime attribute is dependent on the proper subset of any candidate key of table.

An attribute that is not part of any candidate key is known as non-prime attribute.

Example: Suppose a school wants to store the data of teachers and the subjects they teach. They create a table that looks like this: Since a teacher can teach more than one subjects, the table can have multiple rows for a same teacher.

teacher_id	subject	teacher_age
111	Maths	38
111	Physics	38
222	Biology	38
333	Physics	40
333	Chemistry	40

Candidate Keys: {teacher_id, subject}

Non prime attribute: teacher_age

The table is in 1 NF because each attribute has atomic values. However, it is not in 2NF because non prime attribute teacher_age is dependent on teacher_id alone which is a proper subset of candidate key. This violates the rule for 2NF as the rule says "**no** non-prime attribute is dependent on the proper subset of any candidate key of the table".

To make the table complies with 2NF we can break it in two tables like this:

teacher_details table:

teacher_id	teacher_age
111	38
222	38
333	40

teacher_subject table:

teacher_id	subject
111	Maths
111	Physics
222	Biology
333	Physics
333	Chemistry

Now the tables comply with Second normal form (2NF).

Third Normal form (3NF)

A table design is said to be in 3NF if both the following conditions hold:

- Table must be in 2NF
- **Transitive functional dependency** of non-prime attribute on any super key should be removed.

An attribute that is not part of any **candidate key** is known as non-prime attribute.

In other words 3NF can be explained like this: A table is in 3NF if it is in 2NF and for each functional dependency X-> Y at least one of the following conditions hold:

- X is a **super key** of table
- Y is a prime attribute of table

An attribute that is a part of one of the candidate keys is known as prime attribute.

Example: Suppose a company wants to store the complete address of each employee, they create a table named employee_details that looks like this:

emp_id	emp_name	emp_zip	emp_state	emp_city	emp_district
1001	John	282005	UP	Agra	Dayal Bagh

1002	Ajeet	222008	TN	Chennai	M-City
1006	Lora	282007	TN	Chennai	Urrapakkam
1101	Lilly	292008	UK	Pauri	Bhagwan
1201	Steve	222999	MP	Gwalior	Ratan

Super keys: {emp_id}, {emp_id, emp_name}, {emp_id, emp_name, emp_zip}…so on
Candidate Keys: {emp_id}
Non-prime attributes: all attributes except emp_id are non-prime as they are not part of any candidate keys.
Here, emp_state, emp_city & emp_district dependent on emp_zip. And, emp_zip is dependent on emp_id that makes non-prime attributes (emp_state, emp_city & emp_district) transitively dependent on super key (emp_id). This violates the rule of 3NF.
To make this table complies with 3NF we have to break the table into two tables to remove the transitive dependency:
employee table:

emp_id	emp_name	emp_zip
1001	John	282005
1002	Ajeet	222008
1006	Lora	282007
1101	Lilly	292008
1201	Steve	222999

employee_zip table:

emp_zip	emp_state	emp_city	emp_district
282005	UP	Agra	Dayal Bagh
222008	TN	Chennai	M-City
282007	TN	Chennai	Urrapakkam
292008	UK	Pauri	Bhagwan
222999	MP	Gwalior	Ratan

Boyce Codd normal form (BCNF)

It is an advance version of 3NF that's why it is also referred as 3.5NF. BCNF is stricter than 3NF. A table complies with BCNF if it is in 3NF and for every **functional dependency** X->Y, X should be the super key of the table.

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.

Decomposition

1. The previous example might seem to suggest that we should decompose schema as much as possible.
   Careless decomposition, however, may lead to another form of bad design.
2. Consider a design where *Lending-schema* is decomposed into two schemas
3. *Branch-customer-schema = (bname, bcity, assets, cname)*
4.
5. *Customer-loan-schema = (cname, loan#, amount)*
6.
7. We construct our new relations from *lending* by:
8. *branch-customer* = $\Pi_{bname,bcity,assets,cname}(lending)$
9.
10. *customer-loan* = $\Pi_{cname,loan\#,amount}(lending)$

bname	bcity	assets	cname
SFU	Burnaby	2M	Tom
SFU	Burnaby	2M	Mary
Downtown	Vancouver	8M	Tom

cname	loan#	amount
Tom	L-10	10K
Mary	L-20	15K
Tom	L-50	50K

Figure 7.2: The decomposed *lending* relation.

11. It appears that we can reconstruct the *lending* relation by performing a natural join on the two new schemas.
12. Figure 7.3 shows what we get by computing *branch-customer* **1** *customer-loan*.

bname	bcity	assets	cname	loan#	amount
SFU	Burnaby	2M	Tom	L-10	10K
SFU	Burnaby	2M	Tom	L-50	50K
SFU	Burnaby	2M	Mary	L-20	15K
Downtown	Vancouver	8M	Tom	L-10	10K
Downtown	Vancouver	8M	Tom	L-50	50K

Figure 7.3: Join of the decomposed relations.

13. We notice that there are tuples in *branch-customer* **1** *customer-loan* that are not in *lending*.
14. How did this happen?
    - The intersection of the two schemas is *cname*, so the natural join is made on the basis of equality in the cname.
    - If two lendings are for the same customer, there will be four tuples in the natural join.
    - Two of these tuples will be spurious - they will not appear in the original *lending* relation, and should not appear in the database.
    - Although we have **more** tuples in the join, we have **less** information.
    - Because of this, we call this a **lossy** or **lossy-join decomposition**.
    - A decomposition that is not lossy-join is called a **lossless-join decomposition**.
    - The only way we could make a connection between *branch-customer* and *customer-loan* was through *cname*.
15. When we decomposed *Lending-schema* into *Branch-schema* and *Loan-info-schema*, we will not have a similar problem. Why not?
16. *Branch-schema = (bname, bcity, assets)*
17.
18. *Branch-loan-schema = (bname, cname, loan#, amount)*
19.
    - The only way we could represent a relationship between tuples in the two relations is through *bname*.
    - This will not cause problems.
    - For a given branch name, there is exactly one assets value and branch city.
20. For a given branch name, there is exactly one assets value and exactly one bcity; whereas a similar statement associated with a loan depends on the customer, not on the amount of the loan (which is not unique).
21. We'll make a more formal definition of lossless-join:
    - Let *R* be a relation schema.
    - A set of relation schemas $\{R_1, R_2, \ldots, R_n\}$ is a **decomposition** of *R* if
    $$R = R_1 \cup R_2 \cup \ldots \cup R_n$$
    - That is, every attribute in *R* appears in at least one R_i for $1 \leq i \leq n$.
    - Let *r* be a relation on *R*, and let
    $$r_i = \Pi_{R_i}(r) \text{ for } 1 \leq i \leq n$$

- That is, $\{r_1, r_2, \ldots, r_n\}$ is the database that results from decomposing *R* into $\{R_1, R_2, \ldots R_n\}$.
- It is always the case that:

$$r \subseteq r_1 \; 1 \; r_2 \; 1 \ldots 1 \; r_n$$

- To see why this is, consider a tuple $t \in r$.
  - When we compute the relations $\{r_1, r_2, \ldots, r_n\}$, the tuple *t* gives rise to one tuple t_i in each r_i.
  - These *n* tuples combine together to regenerate *t* when we compute the natural join of the r_i.
  - Thus every tuple in *r* appears in $1_{i=1}^{n} r_i$.
- However, in general,

$$r \neq r_1 \; 1 \; r_2 \; 1 \ldots 1 \; r_n$$

- We saw an example of this inequality in our decomposition of *lending* into *branch-customer* and *customer-loan*.
- In order to have a lossless-join decomposition, we need to impose some constraints on the set of possible relations.
- Let *C* represent a set of constraints on the database.
- A decomposition $\{R_1, R_2, \ldots, R_n\}$ of a relation schema *R* is a **lossless-join decomposition** for *R* if, for all relations *r* on schema *R* that are legal under *C*:

$$r = \Pi_{R_1}(r) \; 1 \; \Pi_{R_2}(r) \; 1 \ldots 1 \; \Pi_{R_n}(r)$$

22. In other words, a lossless-join decomposition is one in which, for any legal relation *r*, if we decompose *r* and then ``recompose'' *r*, we get what we started with - no more and no less.

Multivalued dependency

In database theory, a **multivalued dependency** is a full constraint between two sets of attributes in a relation.

In contrast to the functional dependency, the **multivalued dependency** requires that certain tuples be present in a relation. Therefore, a multivalued dependency is a special case of *tuple-generating dependency*. The multivalued dependency plays a role in the 4NF database normalization.

A multivalued dependency is a special case of a join dependency, with only two sets of values involved, i.e. it is a binary join dependency.

A multivalued dependency exists when there are at least 3 attributes (like X,Y and Z) in a relation and for value of X there is a well defined set of values of Y and a well defined set of values of Z. However, the set of values of Y is independent of set Z and vice versa.

Fifth Normal Form (5NF)

A database is said to be in 5NF, if and only if,

- It's in 4NF

- If we can decompose table further to eliminate redundancy and anomaly, and when we re-join the decomposed tables by means of candidate keys, we should not be losing the original data or any new record set should not arise. In simple words, joining two or more decomposed table should not lose records nor create new records.

Consider an example of different Subjects taught by different lecturers and the lecturers taking classes for different semesters.

Note: Please consider that Semester 1 has Mathematics, Physics and Chemistry and Semester 2 has only Mathematics in its academic year!!

In above table, Rose takes both Mathematics and Physics class for Semester 1, but she does not take Physics class for Semester 2. In this case, combination of all these 3 fields is required to identify a valid data. Imagine we want to add a new class - Semester3 but do not know which Subject and who will be taking that subject. We would be simply inserting a new entry with Class as Semester3 and leaving Lecturer and subject as NULL. As we discussed above, it's not a good to have such entries. Moreover, all the three columns together act as a primary key, we cannot leave other two columns blank!