

Database Management System

(18IS5DCDBM)

Unit 4

(SQL, Database Design)

Mrs. Bhavani K
Assistant Professor
Dept. of ISE, DSCE

Unit 4

- **SQL:**
 - Complex Integrity Constraints in SQL
 - Triggers and active Databases
 - Accessing Databases from Applications
 - Stored Procedures
- **Database Design:**
 - Informal Design Guidelines for Relation Schemas
 - Functional Dependencies
 - Normal Forms Based on Primary Keys

SQL

Integrity Constraints

- An IC describes conditions that every legal instance of a relation must satisfy.
 - Inserts, deletes, updates that violate IC's are disallowed.
 - Can be used to ensure application semantics (e.g., sid is a key), or prevent inconsistencies (e.g., sname has to be a string, age must be < 200)
- Types of IC's: Domain constraints, primary key constraints, foreign key constraints, general constraints.
 - Domain constraints: Field values must be of right type.
 - Always enforced.

Constraints over a Single Table

- We can specify complex constraints over a single table using table constraints, which have the form
CHECK conditional-expression.
- The CHECK constraint is used to limit the value range that can be placed in a column.
- If you define a CHECK constraint on a single column it allows only certain values for this column.
- If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

SQL CHECK on CREATE TABLE

- The following SQL creates a CHECK constraint on the “Age” column when the “Persons” table is created. The CHECK constraint ensures that you can not have any person below 18 years:

```
CREATE TABLE Persons (  
  ID int NOT NULL,  
  LastName varchar(255) NOT NULL,  
  FirstName varchar(255),  
  Age int,  
  CHECK (Age>=18) );
```

To allow naming of a CHECK constraint, and for defining a CHECK constraint on multiple columns, use the following SQL syntax:

```
CREATE TABLE Persons (  
  ID int NOT NULL, LastName varchar(255) NOT NULL,  
  FirstName varchar(255),  
  Age int,  
  City varchar(255),  
  CONSTRAINT CHK_Person CHECK (Age>=18 AND  
  City='Sandnes') );
```

SQL CHECK on ALTER TABLE

- To create a CHECK constraint on the “Age” column when the table is already created, use the following SQL:

```
ALTER TABLE Persons ADD CHECK (Age>=18);
```

- To allow naming of a CHECK constraint, and for defining a CHECK constraint on multiple columns, use the following SQL syntax:

```
ALTER TABLE Persons
```

```
ADD CONSTRAINT CHK_PersonAge CHECK (Age>=18 AND City='Sandnes');
```

- **DROP a CHECK Constraint**

- To drop a CHECK constraint, use the following SQL:

```
ALTER TABLE Persons DROP CONSTRAINT CHK_PersonAge;
```

OR

```
ALTER TABLE Persons DROP CHECK CHK_PersonAge;
```

Domain Constraints and Distinct Types

- A user can define a new domain using the CREATE DOMAIN statement, which uses CHECK constraints.

CREATE DOMAIN ratingval INTEGER DEFAULT 1

CHECK (VALUE >= 1 AND VALUE <= 10)

- INTEGER is the underlying, or *source, type for the domain ratingval*, and every ratingval value must be of this type. Values in ratingval are further restricted by using a CHECK constraint.
- Once a domain is defined, the name of the domain can be used to restrict column values in a table; we can use the following line in a schema declaration, for example:
rating ratingval
- The optional DEFAULT keyword is used to associate a default value with a domain.
- SQL's support for the concept of a domain is limited

Domain Constraints and Distinct Types

- CREATE TYPE ratingtype AS INTEGER
 - This statement defines a new *distinct type* called *ratingtype*, with *INTEGER* as its source type.
 - Values of type ratingtype can be compared with each other, but they cannot be compared with values of other types.
 - In particular, ratingtype values are treated as being distinct from values of the source type, INTEGER--we cannot compare them to integers or combine them with integers

Assertions: ICs over Several **Tables**

- Table constraints are associated with a single table, although the conditional expression in the CHECK clause can refer to other tables.
 - Table constraints are required to hold *only if the associated table is nonempty*. Thus, when a constraint involves two or more tables, the table constraint mechanism is sometimes cumbersome and not quite what is desired.
- To cover such situations, SQL supports the creation of assertions, which are constraints not associated with anyone table.

Assertions: ICs over Several Tables

- Enforce the constraint that the number of boats plus the number of sailors should be less than 100.

```
CREATE TABLE Sailors ( sid INTEGER,  
    sname CHAR ( 10) ,  
    rating INTEGER,  
    age REAL,  
    PRIMARY KEY (sid),  
    CHECK ( rating >= 1 AND rating <= 10)  
    CHECK ( ( SELECT COUNT (S.sid) FROM Sailors S )  
        + ( SELECT COUNT (B. bid) FROM Boats B )  
        < 100 ))
```

- This solution suffers from two drawbacks.
 - It is associated with Sailors, although it involves Boats in a completely symmetric way.
 - if the Sailors table is empty, this constraint is defined (as per the semantics of table constraints) to always hold, even if we have more than 100 rows in Boats!
 - we could extend this constraint specification to check that Sailors is nonempty, but this approach becomes cumbersome.
- The best solution is to create an assertion

Assertions: ICs over Several **Tables**

```
CREATE ASSERTION smallClub  
CHECK (( SELECT COUNT (S.sid) FROM Sailors S )  
        + ( SELECT COUNT (B. bid) FROM Boats B)  
        < 100 )
```

Assertions: ICs over Several Tables

- SQL assertions can be used to implement what's commonly called **cross-row constraints**, or **multi-table check constraints**.
- Syntax:

```
CREATE ASSERTION <NAME>  
CHECK ( <condition>);
```
- An assertion is satisfied if and only if the specified <condition> is not false.
- Domain constraints, functional dependency and referential integrity are special forms of assertion.
- 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 Rs.1000 in an account.

TRIGGERS AND ACTIVE DATABASES

- A trigger is a procedure that is automatically invoked by the DBMS in response to specified changes to the database, and is typically specified by the DBA.
- A database that has a set of associated triggers is called an active database.
- A trigger description contains three parts:
 - Event: A change to the database that activates the trigger.
 - Condition: A query or test that is run when the trigger is activated.
 - Action: A procedure that is executed when the trigger is activated and its condition is true.
- A trigger can be thought of as a 'daemon' that monitors a database, and is executed when the database is modified in a way that matches the *event specification*.
- An insert, delete, or update statement could activate a trigger, regardless of which user or application invoked the activating statement
- A condition in a trigger can be a true/false statement (e.g., all employee salaries are less than \$100,000) or a query. A query is interpreted as *true if the answer set is nonempty and false if the query has no answers*.
 - If the condition part evaluates to true, the action associated with the trigger is executed.

Uses of Triggers

- Cascade Changes Through Related Tables in a Database
- Enforce More Complex Data Integrity than a CHECK Constraint
- Define Custom Error Messages
- Automatically update redundant data
- Compare Before and After States of Data Under Modification

Creating Triggers

- **Syntax**

```
Create Trigger Trigger_Name  
(Before | After) [ Insert | Update | Delete]  
on [Table_Name]  
[ for each row | for each column ]  
[ trigger_body ]
```

- **Create Trigger**

These two keywords are used to specify that a trigger block is going to be declared.

- **Trigger_Name**

It specifies the name of the trigger. Trigger name has to be unique and shouldn't repeat.

- **(Before | After)**

This specifies when the trigger will be executed. It tells us the time at which the trigger is initiated, i.e, either before the ongoing event or after.

- *Before Triggers* are used to update or validate record values before they're saved to the database.
- *After Triggers* are used to access field values that are set by the system and to effect changes in other records.

Creating Triggers

- **[*Insert / Update / Delete*]**

These are the DML operations and we can use either of them in a given trigger.

- ***on [Table_Name]***

We need to mention the table name on which the trigger is being applied. Don't forget to use **on** keyword and also make sure the selected table is present in the database.

- **[*for each row | for each column*]**

- Row-level trigger gets executed before or after *any column value of a row* changes.
- Column Level Trigger gets executed before or after the *specified column* changes

- **[*trigger_body*]**

It consists of queries that need to be executed when the trigger is called.

Example for Trigger

To calculate the percentage of the student as soon as the details are inserted to the database:

```
create table Student(  
    sid int primary key,  
    name varchar(20),  
    subj1 int,  
    subj2 int,  
    subj3 int,  
    total int,  
    per int );  
  
create trigger stud_marks  
    before INSERT  
    on  
    Student  
    for each row  
    set new.total = new.subj1 + new.subj2 + new.subj3,  
    new.per = new.total * 60 / 100;  
  
insert into Student values(1, 'Anu', 20, 20, 20, 0, 0);  
  
select * from Student;
```

sid	name	subj1	subj2	subj3	total	per
1	Anu	20	20	20	60	36

Here the “NEW” keyword refers to the row that is getting affected.

Reference link:

Triggers :

<https://youtu.be/f6VWSInHGCE>

Accessing Databases from Applications

- Access to databases via programming languages :
 - SQL is a direct query language; as such, it has limitations.
 - via programming languages :
 - Complex computational processing of the data.
 - Specialized user interfaces.
 - Access to more than one database at a time.

Introduction to SQL

Programming Techniques

- **Database applications**
 - Host language
 - Java, C/C++/C#, COBOL, or some other programming language
 - Data sublanguage
 - SQL
- SQL standards
 - Continually evolving
 - Each DBMS vendor may have some variations from standard

Database Programming: Techniques and Issues

- **Interactive interface**
 - SQL commands typed directly into a monitor
- **Execute file of commands**
 - *@<sql filename> (Oracle SQLPlus)*
 - *source <sql filename> (MySQL)*
- **Application programs or database applications**
 - Used as canned transactions by the end users access a database
 - May have **Web interface**

Approaches to Database Programming

- **Embedding** database commands in a general-purpose programming language
 - Database statements identified by a special prefix
 - **Precompiler** or **preprocessor** scans the source program code
 - Identify database statements and extract them for processing by the DBMS
 - Called **embedded SQL**

Approaches to Database Programming (cont'd.)

- Using a library of database functions
 - **Library of functions** available to the host programming language
 - **Application programming interface (API)**
- Designing a brand-new language
 - **Database programming language** designed from scratch
- First two approaches are more common

SQL in Application Code

- SQL commands can be called from within a host language (e.g., C++ or Java) program.
 - SQL statements can refer to **host variables** (including special variables used to return status).
 - Must include a statement to **connect** to the right database.

SQL in Application Code (Contd.)

Impedance mismatch:

- SQL relations are (multi-) sets of records, with no *a priori* bound on the number of records. No such data structure exist traditionally in programming languages.
 - SQL supports a mechanism called a cursor to handle this.

Impedance Mismatch

- Differences between database model and programming language model
- **Binding** for each host programming language
 - Specifies for each attribute type the compatible programming language types
- Cursor or iterator variable
 - Loop over the tuples in a query result

Desirable features

- Ease of use.
- Conformance to standards for existing programming languages, database query languages, and development environments.
- Interoperability: the ability to use a common interface to diverse database systems on different operating systems

Vendor specific solutions

- Oracle PL/SQL: A proprietary PL/1-like language which supports the execution of SQL queries:
- Advantages:
 - Many Oracle-specific features, not common to other systems, are supported.
 - Performance may be optimized to Oracle based systems.
- Disadvantages:
 - Ties the applications to a specific DBMS.
 - The application programmer must depend upon the vendor for the application development environment.
 - It may not be available for all platforms.

Vendor Independent solutions based on SQL

There are three basic strategies which may be considered:

- Embed SQL in the host language (Embedded SQL, SQLJ)
- SQL modules : invocations to SQL are made via libraries of procedures
- SQL call level interfaces : A call-level interface provides a library of functions for access to DBMS's.
 - The DBMS drivers are stored separately; thus the library used by the programming language is DBMS independent.

Typical Sequence of Interaction in Database Programming

- Open a connection to database server
- Interact with database by submitting queries, updates, and other database commands
- Terminate or close connection to database

Embedded SQL

- Embedded SQL allows execution of parametrized static queries within a host language
- Approach: Embed SQL in the host language.
 - A preprocessor converts the SQL statements into special API calls.
 - Then a regular compiler is used to compile the code.
- Language constructs:
 - Connecting to a database:
`EXEC SQL CONNECT`
 - Declaring variables:
`EXEC SQL BEGIN (END) DECLARE SECTION`
 - Statements:
`EXEC SQL Statement;`

Embedded SQL: Variables

```
EXEC SQL BEGIN DECLARE SECTION  
char c_sname[20];  
long c_sid;  
short c_rating;  
float c_age;  
EXEC SQL END DECLARE SECTION
```

- Two special “error” variables:
 - SQLCODE (long, is negative if an error has occurred)
 - SQLSTATE (char[6], predefined codes for common errors)

SQLCODE and SQLSTATE

- **SQLCODE** and **SQLSTATE** are communication variables
 - Used by DBMS to communicate exception or error conditions
- **SQLCODE** variable
 - 0 = statement executed successfully
 - 100 = no more data available in query result
 - < 0 = indicates some error has occurred
- **SQLSTATE**
 - String of five characters
 - '00000' = no error or exception
 - Other values indicate various errors or exceptions
 - For example, '02000' indicates 'no more data' when using SQLSTATE

Retrieving Single Tuples with Embedded SQL

```
//Program Segment E1:
0) loop = 1 ;
1) while (loop) {
2)     prompt("Enter a Social Security Number: ", ssn) ;
3)     EXEC SQL
4)         select Fname, Minit, Lname, Address, Salary
5)         into :fname, :minit, :lname, :address, :salary
6)         from EMPLOYEE where Ssn = :ssn ;
7)     if (SQLCODE == 0) printf(fname, minit, lname, address, salary)
8)     else printf("Social Security Number does not exist: ", ssn) ;
9)     prompt("More Social Security Numbers (enter 1 for Yes, 0 for No): ", loop) ;
10) }
```

C program segment with embedded SQL.

Cursors

- Cursor mechanism allows retrieval of one record at a time and bridges impedance mismatch between host language and SQL
- Can declare a cursor on a relation or query statement (which generates a relation).
- Can *open* a cursor, and repeatedly *fetch* a tuple then *move* the cursor, until all tuples have been retrieved.
 - Can use a special clause, called **ORDER BY**, in queries that are accessed through a cursor, to control the order in which tuples are returned.
 - Fields in ORDER BY clause must also appear in SELECT clause.
 - The **ORDER BY** clause, which orders answer tuples, is *only* allowed in the context of a cursor.
- Can also modify/delete tuple pointed to by a cursor.

Retrieving Multiple Tuples with Embedded SQL Using Cursors

- **Cursor**
 - Points to a single tuple (row) from result of query
- **OPEN CURSOR** command
 - Fetches query result and sets cursor to a position before first row in result
 - Becomes current row for cursor
- **FETCH** commands
 - Moves cursor to next row in result of query

C program segment that uses
cursors with embedded SQL for update purposes.

```
//Program Segment E2:
0) prompt("Enter the Department Name: ", dname) ;
1) EXEC SQL
2)     select Dnumber into :dnumber
3)     from DEPARTMENT where Dname = :dname ;
4) EXEC SQL DECLARE EMP CURSOR FOR
5)     select Ssn, Fname, Minit, Lname, Salary
6)     from EMPLOYEE where Dno = :dnumber
7)     FOR UPDATE OF Salary ;
8) EXEC SQL OPEN EMP ;
9) EXEC SQL FETCH from EMP into :ssn, :fname, :minit, :lname, :salary ;
10) while (SQLCODE == 0) {
11)     printf("Employee name is:", Fname, Minit, Lname) ;
12)     prompt("Enter the raise amount: ", raise) ;
13)     EXEC SQL
14)         update EMPLOYEE
15)         set Salary = Salary + :raise
16)         where CURRENT OF EMP ;
17)     EXEC SQL FETCH from EMP into :ssn, :fname, :minit, :lname, :salary
18) }
19) EXEC SQL CLOSE EMP ;
```

Retrieving Multiple Tuples with Embedded SQL Using Cursors (cont'd.)

- **FOR UPDATE OF**
 - List the names of any attributes that will be updated by the program
- **Fetch orientation**
 - Added using value: NEXT, PRIOR, FIRST, LAST, ABSOLUTE *i*, and RELATIVE *i*

```
DECLARE <cursor name> [ INSENSITIVE ] [ SCROLL ] CURSOR  
[ WITH HOLD ] FOR <query specification>  
[ ORDER BY <ordering specification> ]  
[ FOR READ ONLY | FOR UPDATE [ OF <attribute list> ] ] ;
```

Cursor that gets names of sailors who've reserved a red boat,
in alphabetical order

```
EXEC SQL DECLARE sinfo CURSOR FOR  
  SELECT S.sname  
  FROM Sailors S, Boats B, Reserves R  
  WHERE S.sid=R.sid AND R.bid=B.bid AND B.color='red'  
  ORDER BY S.sname
```


Embedding SQL in C: An Example

```
char SQLSTATE[6];  
EXEC SQL BEGIN DECLARE SECTION  
char c_sname[20]; short c_minrating; float c_age;  
EXEC SQL END DECLARE SECTION  
c_minrating = random();  
EXEC SQL DECLARE sinfo CURSOR FOR  
    SELECT S.sname, S.age  
    FROM Sailors S  
    WHERE S.rating > :c_minrating  
    ORDER BY S.sname;  
do {  
    EXEC SQL FETCH sinfo INTO :c_sname, :c_age;  
    printf("%s is %d years old\n", c_sname, c_age);  
} while (SQLSTATE != '02000');  
EXEC SQL CLOSE sinfo;
```

Specifying Queries at Runtime Using Dynamic SQL

- **Dynamic SQL**
 - Execute different SQL queries or updates dynamically at runtime
- Dynamic update
- Dynamic query

```
//Program Segment E3:  
0) EXEC SQL BEGIN DECLARE SECTION ;  
1)  varchar sqlupdatestring [256] ;  
2) EXEC SQL END DECLARE SECTION ;  
   ...  
3)  prompt("Enter the Update Command: ", sqlupdatestring) ;  
4) EXEC SQL PREPARE sqlcommand FROM :sqlupdatestring ;  
5) EXEC SQL EXECUTE sqlcommand ;  
   ...
```

C program segment
that uses dynamic SQL for updating a table.

Dynamic SQL

- Dynamic SQL allows execution of completely ad-hoc queries within a host language
- SQL query strings are not always known at compile time (e.g., spreadsheet, graphical DBMS frontend): Allow construction of SQL statements on-the-fly

- Example:

```
char c_sqlstring[]=
    {"DELETE FROM Sailors WHERE rating>5"};
EXEC SQL PREPARE readytogo FROM :c_sqlstring;
EXEC SQL EXECUTE readytogo;
```

Disadvantages

- It is a real pain to debug preprocessed programs.
- The use of a program-development environment is compromised substantially.
- The preprocessor must be vendor and platform specific.

Stored Procedures

- Stored procedures execute application logic directly at the server
- What is a stored procedure:
 - Program executed through a single SQL statement
 - Executed in the process space of the server
- Advantages:
 - Can encapsulate application logic while staying “close” to the data
 - Reuse of application logic by different users
 - Avoid tuple-at-a-time return of records through cursors

Database Stored Procedures and SQL/PSM

- **Stored procedures**
 - Program modules stored by the DBMS at the database server
 - Can be functions or procedures
- **SQL/PSM (SQL/Persistent Stored Modules)**
 - Extensions to SQL
 - Include general-purpose programming constructs in SQL

Database Stored Procedures and Functions

- **Persistent stored modules**
 - Stored persistently by the DBMS
- **Useful:**
 - When database program is needed by several applications
 - To reduce data transfer and communication cost between client and server in certain situations
 - To enhance modeling power provided by views by allowing more complex types of derived data

Database Stored Procedures and Functions (cont'd.)

- Declaring stored procedures:

```
CREATE PROCEDURE <procedure name> (<parameters>)  
<local declarations>  
<procedure body> ;
```

declaring a function, a return type is necessary,
so the declaration form is

```
CREATE FUNCTION <function name> (<parameters>)  
RETURNS <return type>  
<local declarations>  
<function body> ;
```


Database Stored Procedures and Functions (cont'd.)

- Each parameter has parameter type
 - **Parameter type:** one of the SQL data types
 - **Parameter mode:** IN, OUT, or INOUT
- Calling a stored procedure:

```
CALL <procedure or function name>  
(<argument list>) ;
```

Stored Procedures: Examples

```
CREATE PROCEDURE ShowNumReservations
  SELECT S.sid, S.sname, COUNT(*)
  FROM Sailors S, Reserves R
  WHERE S.sid = R.sid
  GROUP BY S.sid, S.sname
```

Stored procedures can have parameters:

- Three different modes: IN, OUT, INOUT

```
CREATE PROCEDURE IncreaseRating(
  IN sailor_sid INTEGER, IN increase INTEGER)
UPDATE Sailors
  SET rating = rating + increase
  WHERE sid = sailor_sid
```

Stored Procedures: Examples (Contd.)

Stored procedure do not have to be written in SQL:

```
CREATE PROCEDURE TopSailors(  
    IN num INTEGER)  
LANGUAGE JAVA  
EXTERNAL NAME "file:///c:/storedProcs/rank.jar"
```

Calling Stored Procedures

```
EXEC SQL BEGIN DECLARE SECTION
```

```
Int sid;
```

```
Int rating;
```

```
EXEC SQL END DECLARE SECTION
```

```
// now increase the rating of this sailor
```

```
EXEC CALL IncreaseRating(:sid,:rating);
```

Calling Stored Procedures (Contd.)

JDBC:

```
CallableStatement cstmt=  
    con.prepareCall("{call  
        ShowSailors});  
ResultSet rs = cstmt.executeQuery();  
while (rs.next()) {  
    ...  
}
```

SQLJ:

```
#sql iterator ShowSailors(...);  
ShowSailors showsailors;  
#sql showsailors={CALL  
    ShowSailors};  
while (showsailors.next()) {  
    ...  
}
```

SQL/PSM

SQL/PSM is a standard for writing stored procedures.

Most DBMSs allow users to write stored procedures in a simple, general-purpose language (close to SQL) → SQL/PSM standard is a representative

Declare a stored procedure:

```
CREATE PROCEDURE name(p1, p2, ..., pn)  
    local variable declarations  
    procedure code;
```

Declare a function:

```
CREATE FUNCTION name (p1, ..., pn) RETURNS sqlDataType  
    local variable declarations  
    function code;
```

SQL/PSM: Extending SQL for Specifying Persistent Stored Modules

- Conditional branching statement:

```
IF <condition> THEN <statement list>
ELSEIF <condition> THEN <statement list>
...
ELSEIF <condition> THEN <statement list>
ELSE <statement list>
END IF ;
```

SQL/PSM (cont'd.)

- Constructs for looping

```
WHILE <condition> DO  
    <statement list>
```

```
END WHILE ;
```

```
REPEAT  
    <statement list>
```

```
UNTIL <condition>
```

```
END REPEAT ;
```

```
FOR <loop name> AS <cursor name> CURSOR FOR <query> DO  
    <statement list>
```

```
END FOR ;
```


Main SQL/PSM Constructs (Contd.)

- Local variables (DECLARE)
- RETURN values for FUNCTION
- Assign variables with SET
- Branches and loops:
 - IF (condition) THEN statements;
ELSEIF (condition) statements;
... ELSE statements; END IF;
 - LOOP statements; END LOOP
- Queries can be parts of expressions
- Can use cursors naturally without “EXEC SQL”

SQL/PSM: Example

Declaring a function in
SQL/PSM.

```
//Function PSM1:  
0) CREATE FUNCTION Dept_size(IN deptno INTEGER)  
1) RETURNS VARCHAR [7]  
2) DECLARE No_of_emps INTEGER ;  
3) SELECT COUNT(*) INTO No_of_emps  
4) FROM EMPLOYEE WHERE Dno = deptno ;  
5) IF No_of_emps > 100 THEN RETURN "HUGE"  
6)     ELSEIF No_of_emps > 25 THEN RETURN "LARGE"  
7)     ELSEIF No_of_emps > 10 THEN RETURN "MEDIUM"  
8)     ELSE RETURN "SMALL"  
9) END IF ;
```

Main SQL/PSM Constructs

```
CREATE FUNCTION rate Sailor
  (IN sailorId INTEGER)
  RETURNS INTEGER
DECLARE rating INTEGER
DECLARE numRes INTEGER
SET numRes = (SELECT COUNT(*)
              FROM Reserves R
              WHERE R.sid = sailorId)
IF (numRes > 10) THEN rating =1;
ELSE rating = 0;
END IF;
RETURN rating;
```

Database Design

Informal Design Guidelines for Relational Databases

- What is relational database design?
 - The grouping of attributes to form "good" relation schemas
- Two levels of relation schemas
 - The logical "user view" level
 - The storage "base relation" level
- Design is concerned mainly with base relations

1. Semantics of the Relational Attributes must be clear

- Informally, each tuple in a relation should represent one entity or relationship instance. (Applies to individual relations and their attributes).
 - Attributes of different entities (EMPLOYEEs, DEPARTMENTs, PROJECTs) should not be mixed in the same relation
 - Only foreign keys should be used to refer to other entities
 - Entity and relationship attributes should be kept apart as much as possible.

GUIDELINE 1

- If a relation schema corresponds to one entity type or one relationship type,
 - it is straightforward to explain its meaning.
- If the relation corresponds to a mixture of multiple entities and relationships,
 - semantic ambiguities will result and the relation cannot be easily explained.

Bottom Line: *Design a schema that can be explained easily relation by relation. The semantics of attributes should be easy to interpret.*

Example

EMP_DEPT

Ename	<u>SSN</u>	Bdate	Address	Dnumber	Dname	DmgrSsn
-------	------------	-------	---------	---------	-------	---------

Employee Related info.

Dept. Specific info.

EMP_PROJ

<u>SSN</u>	<u>Pnumber</u>	Hours	Ename	Pname	LOcation
------------	----------------	-------	-------	-------	----------

Employee Related to project

Attributes which violate the guideline

Guideline 1

- Although there is nothing wrong logically with these two relations,
 - they are considered poor designs because
 - they violate Guideline 1 by mixing attributes from distinct entities
- They may be used as views, but they cause problems when used as base relations

A simplified COMPANY relational database schema

EMPLOYEE F.K.

Ename	<u>Ssn</u>	Bdate	Address	Dnumber
-------	------------	-------	---------	---------

P.K.

DEPARTMENT F.K.

Dname	<u>Dnumber</u>	Dmgr_ssn
-------	----------------	----------

P.K.

DEPT_LOCATIONS
F.K.

<u>Dnumber</u>	<u>Dlocation</u>
----------------	------------------

P.K.

PROJECT F.K.

Pname	<u>Pnumber</u>	Plocation	Dnum
-------	----------------	-----------	------

P.K.

WORKS_ON
F.K. F.K.

<u>Ssn</u>	<u>Pnumber</u>	Hours
------------	----------------	-------

P.K.

P.K –Primary Key
F.K. – Foreign key

2. Redundant Information in Tuples and Update Anomalies

- Information is stored redundantly
 - Wastes storage
 - For example, compare the space used by the two base relations EMPLOYEE and DEPARTMENT with that for an EMP_DEPT base relation in, which is the result of applying the NATURAL JOIN operation to EMPLOYEE and DEPARTMENT.
 - Causes problems with update anomalies
 - Insertion anomalies
 - Deletion anomalies
 - Modification anomalies

Storage Wastage

Redundant data

Ename	SSN	Ddate	Address	Dnumber	Dname	DMGRManager
Salah	111	1965-01-09	Khobar	5	Research	222
Wael	222	1955-12-08	Khobar	5	Research	222
Zaher	333	1968-07-19	Dammam	4	Administration	444
Walid	444	1941-06-20	Dhahran	4	Administration	444
Nasser	555	1962-09-15	Jubail	5	Research	222
Essam	666	1972-07-31	Khobar	5	Research	222
Jabber	777	1969-03-29	Khobar	4	Administration	444
Bader	888	1937-11-10	Khobar	1	Headquarters	888

EMP_DEPARTMENT

Insertion Anomalies

- To insert a new employee tuple into EMP_DEPT:
 - Include the attribute values for the department that the employee works for, or
 - nulls (if the employee does not work for a department as yet).

Insertion Anomalies

EMP_DEPARTMENT

Ename	<u>SSN</u>	Ddate	Address	Dnumber	Dname	DMGRManager
Salah	111	1965-01-09	Khobar	5	Research	222
Wael	222	1955-12-08	Khobar	5	Research	222
Zaher	333	1968-07-19	Dammam	4	Administration	444
Walid	444	1941-06-20	Dhahran	4	Administration	444
Nasser	555	1962-09-15	Jubail	5	Research	222
Essam	666	1972-07-31	Khobar	5	Research	222
Jabber	777	1969-03-29	Khobar	4	Administration	444
Bader	888	1937-11-10	Khobar	1	Headquarters	888
Ali	999	1967-10-15	Dammam			
				6	Accounting	111

A null primary Key

Newly inserted rows

Deletion Anomalies

- If we delete from EMP_DEPT, an employee tuple that represents the last employee working for a particular department:
 - The information concerning that department is lost from the database.

Deletion Anomalies

EMP_DEPARTMENT

Ename	SSN	Ddate	Address	Dnumber	Dname	DMGRManager
Salah	111	1965-01-09	Khobar	5	Research	222
Wael	222	1955-12-08	Khobar	5	Research	222
Zaher	333	1968-07-19	Dammam	4	Administration	444
Walid	444	1941-06-20	Dhahran	4	Administration	444
Nasser	555	1962-09-15	Jubail	5	Research	222
Essam	666	1972-07-31	Khobar	5	Research	222
Jabber	777	1969-03-29	Khobar	4	Administration	444
<i>Bader</i>	<i>888</i>	<i>1937-11-10</i>	<i>Khobar</i>	<i>1</i>	<i>Headquarters</i>	<i>888</i>

Deleting **Bader** will delete the only information of department 1 from the database

Modification Anomalies

- In EMP_DEPT, if we change the value of one of the attributes of a particular department-say, the manager of department 5 :
 - we must update the tuples of all employees who work in that department;
 - otherwise, the database will become inconsistent
- If we fail to update some tuples, the same department will be shown to have two different values for manager in different employee tuples, which would be wrong

Modification Anomalies

Updating the manager Salah to 444 will cause inconsistency



Ename	SSN	Ddate	Address	Dnumber	Dname	DMGRManager
Salah	111	1965-01-09	Khobar	5	Research	444
Wael	222	1955-12-08	Khobar	5	Research	222
Zaher	333	1968-07-19	Dammam	4	Administration	444
Walid	444	1941-06-20	Dhahran	4	Administration	444
Nasser	555	1962-09-15	Jubail	5	Research	222
Essam	666	1972-07-31	Khobar	5	Research	222
Jabber	777	1969-03-29	Khobar	4	Administration	444
Bader	888	1937-11-10	Khobar	1	Headquarters	888

EMP_DEPARTMENT

GUIDELINE 2

- Design a schema that does not suffer from the insertion, deletion and update anomalies.
- If there are any anomalies present, then note them so that applications can be made to take them into account.

3. Null Values in Tuples

- Reasons for nulls:
 - Attribute not applicable or invalid
 - Attribute value unknown (may exist)
 - Value known to exist, but unavailable
- Problems with null values:
 - Waste of disk space
 - Problem of understanding the meaning of attributes
 - Problems in specifying JOIN operations
 - Problems in applying some aggregate functions
 - May have multiple interpretations (not applicable, unknown, unavailable)

GUIDELINE 3

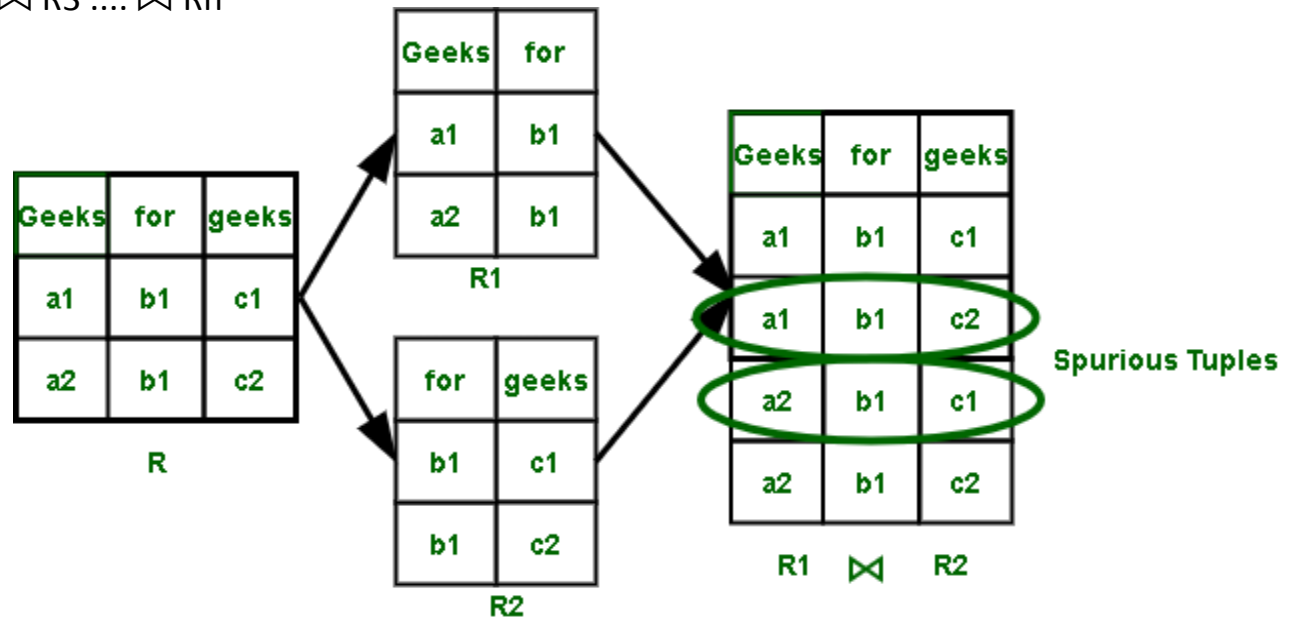
- Relations should be designed such that their tuples will have as few NULL values as possible
- Attributes that are NULL frequently could be placed in separate relations (with the primary key)

4. Generation of Spurious Tuples – avoid at any cost

- Avoid relations that contain matching attributes that are not (foreign key, primary key) combinations,
 - because joining on such attributes may produce spurious tuples.
 - This informal guideline is called the non additive (or lossless) join property, that guarantees that certain joins do not produce spurious tuples.
- Design relation schemas so that they can be joined with equality conditions on attributes that are either primary keys or foreign keys
 - in a way that guarantees that no spurious tuples are generated.

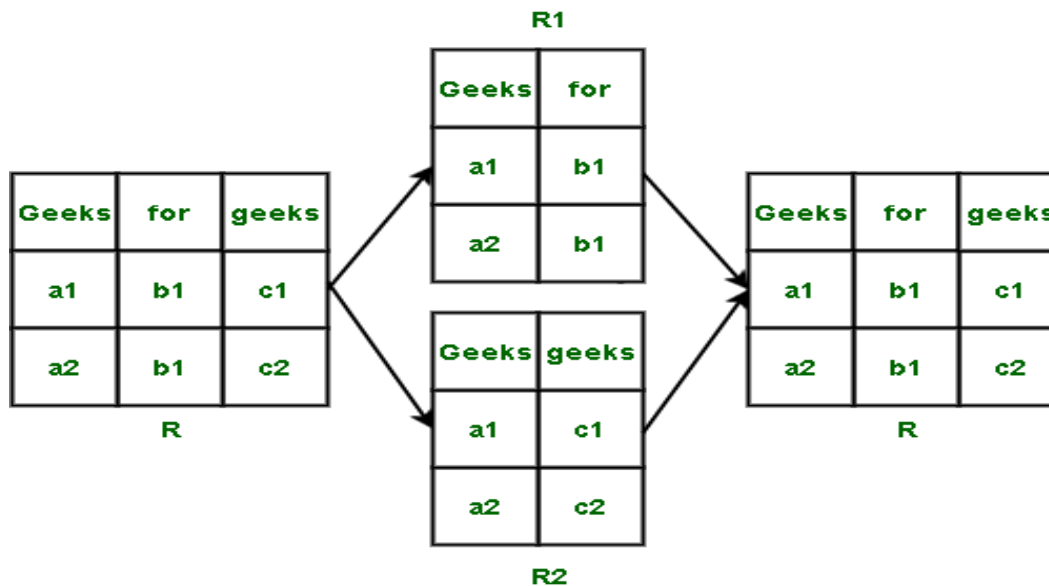
Spurious Tuples

- Spurious Tuples are those rows in a table, which occur as a result of joining two tables in wrong manner. They are extra tuples (rows) which might not be required.
- If relation is denoted by R, and its decomposed relations are denoted by R1, R2, R3....Rn, then, condition for not getting any Spurious Tuple is denoted by,
 - $R1 \bowtie R2 \bowtie R3 \dots \bowtie Rn = R$
- Whereas condition for getting Spurious Tuples is denoted by,
 - $R \subset R1 \bowtie R2 \bowtie R3 \dots \bowtie Rn$



Non Spurious Tuples

- The condition for no spurious tuples, $R1 \bowtie R2 = R$, is met.



GUIDELINE 4

- Bad designs for a relational database may result in erroneous results for certain JOIN operations
- The "lossless join" property is used to guarantee meaningful results for join operations
- **GUIDELINE 4:**
 - The relations should be designed to satisfy the lossless join condition.
 - No spurious tuples should be generated by doing a natural-join of any relations.

Spurious Tuples

- There are two important properties of decompositions:
 - a) Non-additive or losslessness of the corresponding join
 - b) Preservation of the functional dependencies
- Note that:
 - Property (a) is extremely important and cannot be sacrificed.
 - Property (b) is less stringent and may be sacrificed.

Informal Design Guidelines for Relational Databases

- Four informal measures of quality for relation schema design are:
 1. Imparting clear semantics to attributes in Relations
 2. Reducing the redundant values in tuples
 3. Reducing the null values in tuples
 4. Disallowing the possibility of generating spurious tuples.

Functional Dependencies

- Functional dependencies (FDs)
 - Are used to specify *formal measures* of the "goodness" of relational designs
 - And keys are used to define **normal forms** for relations
 - Are **constraints** that are derived from the *meaning* and *interrelationships* of the data attributes
- A set of attributes *X functionally determines* a set of attributes *Y* if the value of *X* determines a unique value for *Y*

Defining Functional Dependencies

- A functional dependency, $X \rightarrow Y$ holds if whenever two tuples have the same value for X , they must have the same value for Y
 - For any two tuples t_1 and t_2 in any relation instance $r(R)$: If $t_1[X]=t_2[X]$, then $t_1[Y]=t_2[Y]$
- $X \rightarrow Y$ in R specifies a constraint on all relation instances $r(R)$
 - This means that the values of the Y component of a tuple in r depend on, or are determined by, the values of the X component
- The values of the X component functionally determines the values of Y component.
- FDs are derived from the real-world constraints on the attributes
- The main use of FD is to describe R by specifying constraints on its attributes that must hold at all times.

Graphical representation of Functional Dependencies

Two relation schemas suffering from update anomalies.

(a) EMP_DEPT and
(b) EMP_PROJ.

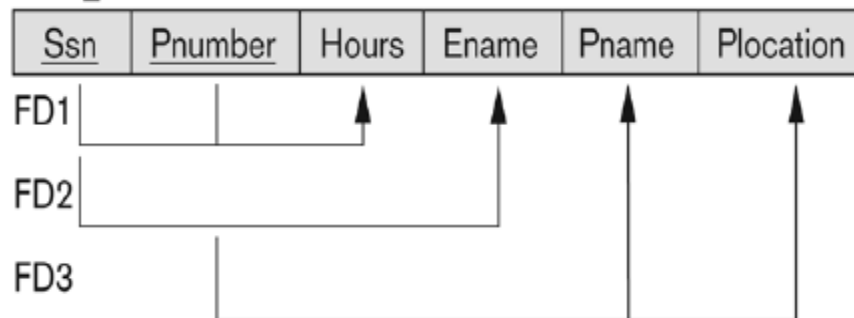
(a)

EMP_DEPT



(b)

EMP_PROJ



Examples of FD constraints

- Social security number determines employee name
 - $SSN \rightarrow ENAME$
- Project number determines project name and location
 - $PNUMBER \rightarrow \{PNAME, PLOCATION\}$
- Employee ssn and project number determines the hours per week that the employee works on the project
 - $\{SSN, PNUMBER\} \rightarrow HOURS$

FD constraints

- An FD is a property of the attributes in the schema R
- The constraint must hold on *every* relation instance $r(R)$
- If K is a key of R, then K functionally determines all attributes in R
 - (since we never have two distinct tuples with $t1[K]=t2[K]$)

Defining FDs from instances

- Note that in order to define the FDs, we need to understand the meaning of the attributes involved and the relationship between them.
- An FD is a property of the attributes in the schema R
- Given the instance (population) of a relation, all we can conclude is that an FD may exist between certain attributes.
- What we can definitely conclude is – that certain FDs do not exist because there are tuples that show a violation of those dependencies.

Ruling Out FDs

TEACH

Teacher	Course	Text
Smith	Data Structures	Bartram
Smith	Data Management	Martin
Hall	Compilers	Hoffman
Brown	Data Structures	Horowitz

Note that given the state of the TEACH relation, we can say that the FD: Text \rightarrow Course may exist. However, the FDs Teacher \rightarrow Course, Teacher \rightarrow Text and Course \rightarrow Text are ruled out.

What FDs may exist?

- A relation $R(A, B, C, D)$ with its extension.
- Which FDs may exist in this relation?

A	B	C	D
a1	b1	c1	d1
a1	b2	c2	d2
a2	b2	c2	d3
a3	b3	c4	d3

What FDs may exist?

- The following FDs may hold $B \rightarrow C$, $C \rightarrow B$

A	B	C	D
a1	b1	c1	d1
a1	b2	c2	d2
a2	b2	c2	d3
a3	b3	c4	d3

Inference Rules for Functional Dependencies

- F is the set of functional dependencies that are specified on relation schema R .
- Schema designers specifies the most obvious FDs.
- The other dependencies can be inferred or deduced from FDs in F .

Example of Closure

- Department has one manager
(DEPT_NO \rightarrow MGR_SSN)
- Manager has a unique phone number
(MGR_SSN \rightarrow MGR_PHONE)
- These two dependencies together imply that
(DEPT_NO \rightarrow MGR_PHONE)

Closure

- This defines a concept called as closure that includes all possible dependencies that can be inferred from the given set F .
- The set of all dependencies that include F as well as all dependencies that can be inferred from F is called the closure of F denoted by (F^+)

Example

- $F = \{SSN \rightarrow \{ENAME, BDATE, ADDRESS, DNUMBER\}, DNUMBER \rightarrow \{DNAME, DMGRSSN\}\}$
- The inferred functional dependencies are
 - $SSN \rightarrow \{DNAME, DMGRSSN\}$
 - $SSN \rightarrow SSN$
 - $DNUMBER \rightarrow DNAME$
- To determine a systematic way to infer dependencies, a set of inference rules has to be discovered that can be used to infer new dependencies from a given set of dependencies. This is denoted by $F \models X \rightarrow Y$

Inference Rules for FDs

- Given a set of FDs F , we can infer additional FDs that hold whenever the FDs in F hold
- Armstrong's inference rules:
 - IR1. (Reflexive) If Y subset-of X , then $X \rightarrow Y$
 - IR2. (Augmentation) If $X \rightarrow Y$, then $XZ \rightarrow YZ$
 - (Notation: XZ stands for $X \cup Z$)
 - IR3. (Transitive) If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

Inference Rules for FDs

- IR1, IR2, IR3 form a sound and complete set of inference rules
- *By sound we mean, any dependency that we can infer from F by using IR1 through IR3 satisfies the dependencies in F . In other words, if the axioms are correctly applied they cannot derive false dependencies*
- *By complete we mean that by using IR1 through IR3 repeatedly to infer dependencies until no more dependencies can be inferred results in complete set of all possible valid dependencies that can be inferred from F*

Inference Rules for FDs

- **Some additional inference rules that are useful:**
 - Decomposition: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
 - Union or additive: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
 - Pseudotransitive: If $X \rightarrow Y$ and $WY \rightarrow Z$, then $WX \rightarrow Z$
- **The last three inference rules, as well as any other inference rules, can be deduced from IR1, IR2, and IR3 (completeness property)**

Normal Forms Based on Primary Keys

- Normalization of Relations
- Practical Use of Normal Forms
- Definitions of Keys and Attributes Participating in Keys
- First Normal Form
- Second Normal Form
- Third Normal Form

Normalization of Relations

- **Normalization:**

- The process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations

- **Normal form:**

- Condition using keys and FDs of a relation to certify whether a relation schema is in a particular normal form

Normalization of Relations

- 2NF, 3NF, BCNF
 - based on keys and FDs of a relation schema
- 4NF
 - based on keys, multi-valued dependencies : MVDs
- 5NF
 - based on keys, join dependencies : JDs
- Additional properties may be needed to ensure a good relational design (lossless join, dependency preservation)

Practical Use of Normal Forms

- **Normalization** is carried out in practice so that the resulting designs are of high quality and meet the desirable properties
- The practical utility of these normal forms becomes questionable when the constraints on which they are based are *hard to understand* or to *detect*
- The database designers *need not* normalize to the highest possible normal form
 - (usually up to 3NF and BCNF. 4NF rarely used in practice.)
- **Denormalization:**
 - The process of storing the join of higher normal form relations as a base relation—which is in a lower normal form

Definitions of Keys and Attributes Participating in Keys

- A **superkey** of a relation schema $R = \{A_1, A_2, \dots, A_n\}$ is a set of attributes S *subset-of* R with the property that no two tuples t_1 and t_2 in any legal relation state r of R will have $t_1[S] = t_2[S]$
- A **key** K is a **superkey** with the *additional property* that removal of any attribute from K will cause K not to be a superkey any more.

Definitions of Keys and Attributes Participating in Keys

- If a relation schema has more than one key, each is called a **candidate** key.
 - One of the candidate keys is *arbitrarily* designated to be the **primary key**, and the others are called **secondary keys**.
- A **Prime attribute** must be a member of *some* candidate key
- A **Nonprime attribute** is not a prime attribute—that is, it is not a member of any candidate key.

First Normal Form (1NF)


- 1NF states that the domain of an attribute must include only atomic (simple, indivisible) values.
- Thus, 1NF disallows
 - composite attributes
 - multivalued attributes
 - **nested relations**; attributes whose values for an *individual tuple* are non-atomic
- Considered to be part of the definition of a relation
- Most RDBMSs allow only those relations to be defined that are in First Normal Form

Normalization into 1NF

(a)

DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	Dlocations



Normalization into 1NF

(a) A relation schema that is not in 1NF.

(b)

DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(b) Sample state of relation DEPARTMENT.

(c) 1NF version of the same relation with redundancy.

(c)

DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	<u>Dlocation</u>
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Normalizing nested relations into 1NF

(a)

EMP_PROJ

Ssn	Ename	Projs	
		Pnumber	Hours

(b)

EMP_PROJ

Ssn	Ename	Pnumber	Hours
123456789	Smith, John B.	1	32.5
		2	7.5
666884444	Narayan, Ramesh K.	3	40.0
453453453	English, Joyce A.	1	20.0
		2	20.0
333445555	Wong, Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zelaya, Alicia J.	30	30.0
		10	10.0
987987987	Jabbar, Ahmad V.	10	35.0
		30	5.0
987654321	Wallace, Jennifer S.	30	20.0
		20	15.0
888665555	Borg, James E.	20	NULL

(c)

EMP_PROJ1

Ssn	Ename
-----	-------

EMP_PROJ2

Ssn	Pnumber	Hours
-----	---------	-------

Normalizing nested relations into 1NF. (a) Schema of the EMP_PROJ relation with a nested relation attribute PROJS. (b) Sample extension of the EMP_PROJ relation showing nested relations within each tuple. (c) Decomposition of EMP_PROJ into relations EMP_PROJ1 and EMP_PROJ2 by propagating the primary key.

Normalization into 1NF

item	colors	price	tax
T-shirt	red, blue	12.00	0.60
polo	red, yellow	12.00	0.60
T-shirt	red, blue	12.00	0.60
sweatshirt	blue, black	25.00	1.25

This table is not in first normal form because the “Colour” column contains multiple Values.

Normalization into 1NF

item	color	price	tax
T-shirt	red	12.00	0.60
T-shirt	blue	12.00	0.60
polo	red	12.00	0.60
polo	yellow	12.00	0.60
sweatshirt	blue	25.00	1.25
sweatshirt	black	25.00	1.25

Table is now in first normal form

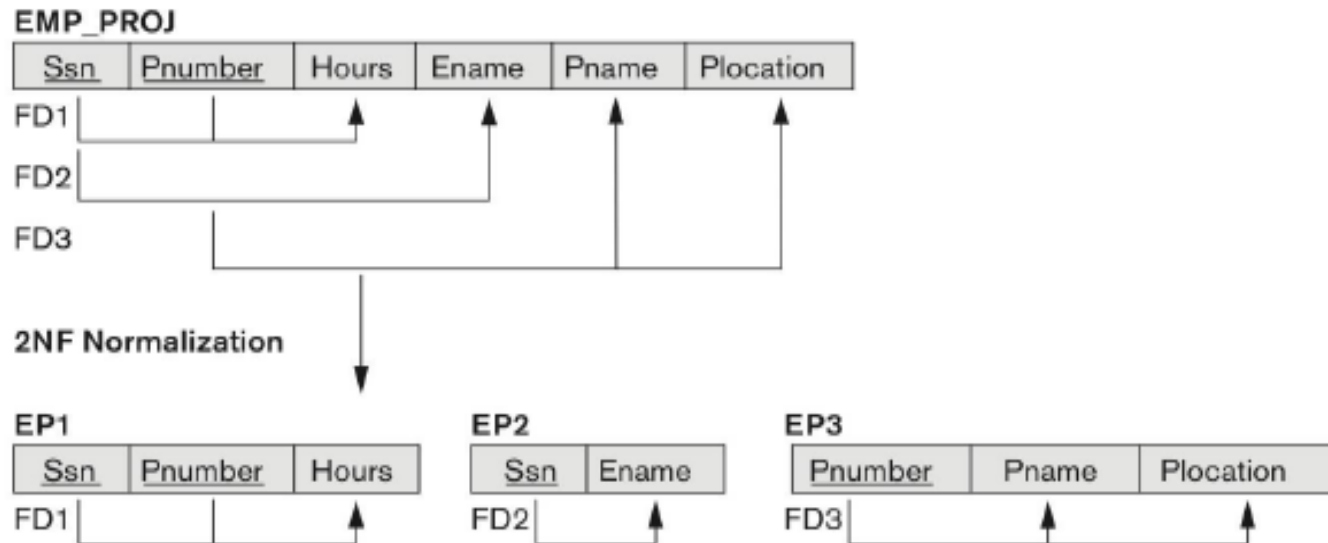
Second Normal Form (2NF)

- 2NF is based on the concepts of FDs, and primary key
- Definitions
 - Prime attribute: An attribute that is member of the candidate key K
 - Full functional dependency: a FD $Y \rightarrow Z$ where removal of any attribute from Y means the FD does not hold any more (or if no proper subset of Y also determines Z).
- Examples:
 - $\{SSN, PNUMBER\} \rightarrow HOURS$ is a full FD since neither $SSN \rightarrow HOURS$ nor $PNUMBER \rightarrow HOURS$ hold
 - $\{SSN, PNUMBER\} \rightarrow ENAME$ is not a full FD (it is called a partial dependency) since $SSN \rightarrow ENAME$ also holds
- A Relation schema R is in 2NF if every nonprime attribute (not a member of primary or candidate key) A in R is FFD (fully functionally Dependent) on the primary or candidate key of R.

Second Normal Form

- The Test for 2NF involves testing for FD's whose left hand side attributes are part of the primary key.
 - Eg. The EMP_PROJ relation is not in 2NF
 - Emp_Proj(Ssn, Pnumber, Hours, Ename, Pname, Plocation) has the following FDs
 - FD1: (Ssn,Pnumber)->Hours
 - FD2: Ssn-> Ename
 - FD3: Pnumber-> Pname, Plocation
- EMP_PROJ is not in 2NF because from FD2 and FD3:
 - the nonprime (non key) attributes Ename, Pname and Plocation are determined by a subset of the primary key of EMP_PROJ(Ssn, Pnumber) thus violating the 2NF test stating that all non key attributes should be FFD on the key.
- To place in 2NF, break into a number of relations in which non key attributes are associated only with the part of the primary key on which they are FFD on.
- Thus, decomposition of EMP_PROJ into EP1, Ep2 and Ep3 in 2NF.
 - EP1(Ssn, pnumber, Hours)
 - Ep2(Ssn, Ename)
 - EP3(Pnumber, Pname, Plocation)

Second Normal Form (2NF) Example



Normalizing EMP_PROJ into 2NF relations

Second Normal Form (2NF) Example

item	color	price	tax
T-shirt	red	12.00	0.60
T-shirt	blue	12.00	0.60
polo	red	12.00	0.60
polo	yellow	12.00	0.60
sweatshirt	blue	25.00	1.25
sweatshirt	black	25.00	1.25

Table is not in second normal form because:
price and tax depend on item, but not color

Second Normal Form (2NF) Example

item	color
T-shirt	red
T-shirt	blue
polo	red
polo	yellow
sweatshirt	blue
sweatshirt	black

item	price	tax
T-shirt	12.00	0.60
polo	12.00	0.60
sweatshirt	25.00	1.25

Tables are now in second normal form

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.
- All non-key attributes depend on all components of the primary key.
 - Guaranteed when primary key is a single attribute.

Third Normal Form (3NF)

- Definition:
 - **Transitive functional dependency:** is a FD $X \rightarrow Z$ that can be derived from two FDs $X \rightarrow Y$ and $Y \rightarrow Z$
- Examples:
 - $SSN \rightarrow DMGRSSN$ is a **transitiveFD**
 - Since $SSN \rightarrow DNUMBER$ and $DNUMBER \rightarrow DMGRSSN$ hold
 - $SSN \rightarrow ENAME$ is **non-transitive**
 - Since there is no set of attributes X where SSN

Third Normal Form (3NF)

- A relation (R) is in 3NF if it is in 2NF and no non prime attribute of R is transitively dependent on the primary/candidate key. That is, when both of these conditions hold:
 - (a) R is fully functionally dependent on every key of R
 - (b) R is non-transitively dependent on every key of R
- In $X \rightarrow Y$ and $Y \rightarrow Z$, with X as the primary key, we consider this a problem only if Y is not a candidate key.
- When Y is a candidate key, there is no problem with the transitive dependency
- E.g., Consider EMP (SSN, Emp#, Salary)
 - Here, $SSN \rightarrow Emp\# \rightarrow Salary$ and Emp# is a candidate key

Third Normal Form (3NF) Example

item	price	tax
T-shirt	12.00	0.60
polo	12.00	0.60
sweatshirt	25.00	1.25

Table is not in third normal form because:
tax depends on price, not item



item	price
T-shirt	12.00
polo	12.00
sweatshirt	25.00

price	tax
12.00	0.60
25.00	1.25

Tables are now in third normal form

Exercise

- Normalize the given relation to 3NF

Name	Assignment 1	Assignment 2
Jeff Smith	Article Summary	Poetry Analysis
Nancy Jones	Article Summary	Reaction Paper
Jane Scott	Article Summary	Poetry Analysis

Solution

Assignment ID	Description
1	Article Summary
2	Poetry Analysis
3	Reaction Paper

Student ID	First Name	Last Name
1	Jeff	Smith
2	Nancy	Jones
3	Jane	Scott

Assignment ID	Student ID
1	1
1	2
1	3
2	1
2	3
3	2

Thank you