25CS305

Database Management Systems

M. NARMADHA,
Assistant Professor,
Department of Computer Science and Engineering

NARSIMHA REDDY ENGINEERING COLLEGE
UGC AUTONOMOUS INSTITUTION
Maisammaguda (V), Kompally - 500100, Secunderabad, Telangana State, India
Accredited by NBA & NAAC with 'A' Grade
Approved by AICTE
Permanently affiliated to JNTUH

UNIT-II

What is SQL

- When a user wants to get some information from a database file, he can issue a **query**.
- A query is a user–request to retrieve data or information with a certain condition.
- SQL is a query language that allows user to specify the conditions.

DDL (Data Definition Language)

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
- RENAME – rename an object

DML (Data Manipulation Language)

- **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

INSERT COMMAND

Des:

This command is used to enter (input) data into the created table.

Syntax:

INSERT INTO

<TableName> (<ColumnName1>,<ColumnName2>)
VALUES(<Expression1>,<Expression2>);

Ex:

INSERT INTO

STUDENT(ROLL_NO,NAME,BRANCH,PERCENT)

VALUES('CS05111', 'AAA', 'CO.SC', 84.45);

Modification of the Database – Insertion

- Add a new tuple to *account*

  insert into *account*
  values ('A-9732', 'Perryridge', 1200)

  or equivalently

  insert into *account* (*branch_name, balance, account_number*)
  values ('Perryridge', 1200, 'A-9732')

- Add a new tuple to *account* with *balance* set to null

  insert into *account*
  values ('A-777','Perryridge', *null*)

Modification of the Database – Updates

- Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.
  - Write two **update** statements:

update *account*
set *balance* = *balance* * 1.06
where *balance* > 10000

update *account*
set *balance* = *balance* * 1.05
where *balance* ≤ 10000

Case Statement for Conditional Updates

- Same query as before: Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.

```
update account
set balance =  case
                    when balance <= 10000
then balance *1.05
                    else   balance * 1.06
                end
```

DCL (Data Control Language)

Data Control Language (DCL) statements. Some examples:

- GRANT – to allow specified users to perform specified tasks.
- REVOKE – to cancel previously granted or denied permissions.
- COMMIT – save work done
- ROLLBACK – restore database to original since the last COMMIT

- The following privileges can be GRANTED TO or REVOKED FROM a user or role:
- SELECT
- INSERT
- UPDATE
- DELETE

Priviledge

The rights that allow the use of some or all Oracle's resources on the server are called Priviledge.

Granting the Priviledge

Objects that are created by the user are owned and controlled by the user. If a user wishes to access any of the objects belonging to other user , the owner of the object will have to give the permissions for such access. This is called Granting the Priviledge.

Revoking of Priviledges

Privileges once given can be taken by the owner of the object . This is called evoking of priviledge.

SQL GRANT Command

- SQL GRANT is a command used to provide access or privileges on the database objects to the users.

The Syntax for the GRANT command is:

- GRANT <privilege_name>
  ON <object_name >
  TO {user_name }
  [WITH GRANT OPTION];

- ***privilege_name*** is the access right or privilege granted to the user. Some of the access rights are ALL, EXECUTE, and SELECT.
- ***object_name*** is the name of an database object like TABLE, VIEW, STORED PROC and SEQUENCE.
- ***user_name*** is the name of the user to whom an access right is being granted.
- ***user_name*** is the name of the user to whom an access right is being granted.
- ***WITH GRANT OPTION*** – allows a user to grant access rights to other users.

SQL REVOKE Command:

- The REVOKE command removes user access rights or privileges to the database objects.
- The Syntax for the REVOKE command is:
- REVOKE privilege name
  ON object name
  FROM {username }

- The set operations **union, intersect,** and **except** operate on relations and correspond to the relational algebra operations ∪, ∩, −.
- Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multi set versions **union all, intersect all** and **except all.**

Set Operations

eg. 15 Find all customers who have a loan, an account, or both:

(select *customer_name* **from** *depositor*)
union
(select *customer_name* **from** *borrower)*

eg. 16 Find all customers who have both a loan and an account.

(**select** *customer_name* **from** *depositor*)
intersect
(select *customer_name* **from** *borrower)*

eg. 17 Find all customers who have an account but no loan.

(**select** *customer_name* **from** *depositor*)
except
(select *customer_name* **from** *borrower)*

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

avg: average value
min: minimum value
max: maximum value
sum: sum of values
count: number of values

Aggregate Functions (Cont.)

eg. 18 Find the average account balance at the Perryridge branch.

select avg *(balance)*
from *account*
where *branch_name* = 'Perryridge'

- **eg. 19** Find the number of tuples in the *customer* relation.

select count (*) **from** *customer*

- **eg. 20** Find the number of depositors in the bank.

select count (distinct *customer_name)*
from *depositor*

Group by clause

- This is used to produce summary information for a subset of the table .
- This will produce a single row for all the rows that met a particular condition.

Employee

ID	F_NAME	L_NAME	DOB	SALARY
1	vaibhav	bharade	19-OCT-90	30000
2	pradeep	verma	19-OCT-91	30000
3	navin	singh	19-OCT-92	30000
4	vibhu	bharade	19-OCT-93	30000
5	anish	bharadwaj	19-OCT-94	30000
6	minu	choudhary	27-JUN-86	68888
7	ankita	choudhary	27-JAN-86	68888
1	vaibhav	bharade	19-OCT-80	5000
2	pradeep	verma	20-OCT-80	5000

Grouping

eg. 21 To sum the salary of each employee group by id.

SELECT id, sum(salary) FROM employee GROUP BY id;

Result

ID	SUM(SALARY)
1	35000
6	68888
2	35000
4	30000
5	30000
3	30000

Grouping

eg. 22 Find the average of the salary of each employee group by id.

```
SELECT id, avg(salary) FROM employee GROUP
BY id;
```

Result

ID	AVG(SALARY)
1	17500
6	68888
2	17500
4	30000
5	30000
3	30000

Grouping

eg. 23 Find the minimum of the salary of each employee group by id.

```
SELECT id, min(salary) FROM employee
group by id;
```

Result

ID	MIN(SALARY)
1	5000
6	68888
2	5000
4	30000
5	30000
3	30000

Grouping

eg. 24 Find the maximum of the salary of each employee group by id.

SELECT id, max(salary) FROM employee group by id;

Result

ID	MAX(SALARY)
1	30000
6	68888
2	30000
4	30000
5	30000

Grouping

eg. 25 List the number of id present in employee table.

SELECT id, count(id) FROM employee group by id;

ID	COUNT(ID)
1	2
6	1
2	2
4	1
5	1
3	1
7	1

Aggregate Functions – Having Clause

eg. 26 Find the names of all branches where the average account balance is more than $1,200.

select *branch_name,* **avg** (*balance*)
from *account*
group by *branch_name*
having avg (*balance*) > 1200

Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The predicate **is null** can be used to check for null values.

eg. 27 Find all loan number which appear in the *loan* relation with null values for *amount.*

select *loan_number*
from *loan*
where *amount* **is null**

Null Values and Aggregates

- Total all loan amounts

  select sum (*amount*)**from** *loan*
  - Above statement ignores null amounts
  - Result is *null* if there is no non-null amount
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes.

Nested Sub queries

- SQL provides a mechanism for the nesting of sub queries.
- A **sub query** is a **select-from-where** expression that is nested within another query.
- A common use of sub queries is to perform tests for set membership, set comparisons, and set cardinality.

Example Query

eg. 28 Find all customers who have both an account and a loan at the bank.

```
select distinct customer_name
        from borrower
        where customer_name in (select customer_name
                                from depositor )
```

eg. 29 Find all customers who have a loan at the bank but do not have an account at the bank

```
select distinct customer_name
        from borrower
        where customer_name not in (select customer_name
                                    from depositor )
```

Example Query

eg. 30 Find all customers who have both an account and a loan at the Perryridge branch

```
select distinct customer_name
        from borrower, loan
        where borrower.loan_number = loan.loan_number and
          branch_name = 'Perryridge'  and
          (branch_name, customer_name ) in
                  (select branch_name, customer_name
                    from depositor, account
                    where depositor.account_number =
                        account.account_number )
```

- Note: Above query can be written in a much simpler manner. The formulation above is simply to illustrate SQL features.

Set Comparison

eg. 31 Find all branches that have greater assets than some branch located in Brooklyn.

```
select distinct  T.branch_name
        from branch as T, branch as S
        where  T.assets > S.assets and
                S.branch_city = 'Brooklyn'
```

- Same query using > **some** clause

```
select branch_name
        from branch
        where assets > some
                (select assets
                 from branch
                 where branch_city = 'Brooklyn')
```

Example Query

eg. 32 Find the names of all branches that have greater assets than all branches located in Brooklyn.

```
select branch_name
        from branch
        where assets > all
                (select assets
                from branch
                where branch_city = 'Brooklyn')
```

Example Query

eg. 33 Find all customers who have an account at all branches located in Brooklyn.

```
select distinct S.customer_name
        from depositor as S
        where not exists (
                (select branch_name
                from branch
                where branch_city = 'Brooklyn')
                except
                (select R.branch_name
                from depositor as T, account as R
                where T.account_number = R.account_number and
                        S.customer_name = T.customer_name ))
```

With Clause

- The **with** clause provides a way of defining a temporary view whose definition is available only to the query in which the **with** clause occurs.

eg. 34 Find all accounts with the maximum balance

```
with max_balance (value) as
select max (balance)
from account
select account_number
from account, max_balance
where account.balance =
max_balance.value
```

Complex Queries using With Clause

eg. 35 Find all branches where the total account deposit is greater than the average of the total account deposits at all branches.

```
with branch_total (branch_name, value) as
        select branch_name, sum (balance)
        from account
        group by branch_name
with branch_total_avg (value) as
        select avg (value)
        from branch_total
select branch_name
from branch_total, branch_total_avg
where branch_total.value >= branch_total_avg.value
```

DATABASE TRIGGERS

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

avg: average value
min: minimum value
max: maximum value
sum: sum of values
count: number of values

Aggregate Functions (Cont.)

eg. 18 Find the average account balance at the Perryridge branch.

select avg *(balance)*
from *account*
where *branch_name* = 'Perryridge'

- **eg. 19** Find the number of tuples in the *customer* relation.

select count (*) **from** *customer*

- **eg. 20** Find the number of depositors in the bank.

select count (distinct *customer_name)*
from *depositor*

Group by clause

- This is used to produce summary information for a subset of the table .
- This will produce a single row for all the rows that met a particular condition.

Employee

ID	F_NAME	L_NAME	DOB	SALARY
1	vaibhav	bharade	19-OCT-90	30000
2	pradeep	verma	19-OCT-91	30000
3	navin	singh	19-OCT-92	30000
4	vibhu	bharade	19-OCT-93	30000
5	anish	bharadwaj	19-OCT-94	30000
6	minu	choudhary	27-JUN-86	68888
7	ankita	choudhary	27-JAN-86	68888
1	vaibhav	bharade	19-OCT-80	5000
2	pradeep	verma	20-OCT-80	5000

Grouping

eg. 21 To sum the salary of each employee group by id.

```
SELECT id, sum(salary) FROM employee GROUP
BY id;
```

Result

ID	SUM(SALARY)
1	35000
6	68888
2	35000
4	30000
5	30000
3	30000

Grouping

eg. 22 Find the average of the salary of each employee group by id.

```
SELECT id, avg(salary) FROM employee GROUP
BY id;
```

Result

ID	AVG(SALARY)
1	17500
6	68888
2	17500
4	30000
5	30000
3	30000

Grouping

eg. 23 Find the minimum of the salary of each employee group by id.

SELECT id, min(salary) FROM employee group by id;

Result

ID	MIN(SALARY)
1	5000
6	68888
2	5000
4	30000
5	30000
3	30000

Grouping

eg. 24 Find the maximum of the salary of each employee group by id.

SELECT id, max(salary) FROM employee group by id;

Result

ID	MAX(SALARY)
1	30000
6	68888
2	30000
4	30000
5	30000

Grouping

eg. 25 List the number of id present in employee table.

SELECT id, count(id) FROM employee group by id;

ID	COUNT(ID)
1	2
6	1
2	2
4	1
5	1
3	1
7	1

Aggregate Functions – Having Clause

eg. 26 Find the names of all branches where the average account balance is more than $1,200.

select *branch_name,* **avg** (*balance*)
from *account*
group by *branch_name*
having avg (*balance*) > 1200

Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

Introduction

- The Oracle server allows the user to define procedures that are implicitly executed, when an insert, update or delete is issued against a table.
- These procedures are called database triggers.
- The major issue that make these triggers stand-alone is that they are fired implicitly(i.e. internally) by the Oracle server itself and not explicitly called by the user.

Use of database triggers

- A trigger can permit DML statements against a table only if they are issued, during regular business hours or on predetermined weekdays.
- A trigger can also be used to keep an audit trail of a table along with the operation performed and the time on which the operation was performed.
  - It can be used to prevent invalid transactions.
  - Enforce complex security authorizations.

How to apply database triggers

- A trigger has three basic parts
  - A triggering event or statement
  - A trigger restriction
  - A trigger action

A triggering event or statement

- It is a SQL statement that causes a trigger to be fired.
- It can be INSERT, UPDATE or DELETE statement for a specific table.

A trigger restriction

- A trigger restriction specifies a Boolean (logical) expression that must be TRUE for the trigger to fire.
- It is an option available for triggers that are fired for each row.
- Its function is to conditionally control the execution of a trigger.
- A trigger restriction is specified using a WHEN clause.

A trigger action

- A trigger action is the PL/SQL code to be executed when a triggering statement is encountered and any trigger restriction evaluates to TRUE.
- The PL/SQL block can contain SQL and PL/SQL statements, can define PL/SQL language constructs and can call stored procedures.

Row triggers

- A row trigger is fired each time a row in the table is affected by the triggering statement.
- For example, if an UPDATE statement updates multiple rows of a table, a row trigger is fired once for each row affected by the UPDATE statement.
- If the triggering statement affects no rows, the trigger is not executed at all.
- Row triggers should be used when some processing is required whenever a triggering statement affects a single row in a table.

Statement triggers

- A statement trigger is fired once on behalf of the triggering statement, independent of the number of rows the triggering statement affects.
- Statement triggers should be used when a triggering statement affects rows in a table but the processing required is completely independent of the number of rows affected.

Before V/s after triggers

- When defining a trigger it is necessary to specify the trigger timing, i.e. specifying when the triggering action is to be executed in relation to the triggering statement.
- Before and after apply to both row and the statement triggers.

Before triggers

- Before triggers execute the trigger action before the triggering statement. These types of triggers are commonly used in the following situation:
  - Before triggers are used when the trigger action should determine whether or not the triggering statement should be allowed to complete. By using a before trigger, one can eliminate unnecessary processing of the triggering statement.
  - Before triggers are used to drive specific column values before completing a triggering INSERT or UPDATE statement.

After triggers

- AFTER trigger executes the trigger action after the triggering statement is executed. These types of triggers are commonly used in the following situation:
  - AFTER triggers are used when you want the triggering statement to complete before executing the trigger action.
  - If a before trigger is already present, an AFTER trigger can perform different actions on the same triggering statement.

Combinations triggers

- **BEFORE statement trigger**: before executing the triggering statement, the trigger action is executed.
- **BEFORE row trigger**: before modifying each row affected by the triggering statement and before appropriate integrity constraints, the trigger is executed if the trigger restriction either evaluated to TRUE or was not included.
- **AFTER statement trigger**: after executing the triggering statement and applying any deferred integrity constraints, the trigger action is executed.
- **AFTER row trigger**: after modifying each row affected by the triggering statement and possibly applying appropriate integrity constraints, the trigger action is executed for the current row if the trigger restriction either evaluates to TRUE or was not included. Unlike BEFORE row triggers, AFTER row triggers have rows locked.

Creating a Triggers

CREATE OR REPLACE TRIGGER triggername
{ BEFORE, AFTER}
{DELETE, INSERT,UPDATE [OF column,…..]}
ON tablename
[REFERENCING {OLD AS old, NEW AS new}]
[FOR EACH ROW [WHEN condition]]

Keyword and parameters

- **OR REPLACE**: recreates the trigger if it already exists. This option can be used to change the definition of an existing trigger without first dropping it.
- **Triggername**: is the name of the trigger to be cretated.
- **BEFORE**: indicates that the Oracle server fires the trigger before executing the triggering statement.
- **AFTER**: indicates that the Oracle server fires the trigger after executing the triggering statement.

Keyword and parameters

- **DELETE**: indicates that the Oracle server fires the trigger whenever a DELETE statement removes a row from the table.
- **INSERT**: indicates that the Oracle server fires the trigger whenever an INSERT statement adds a row to table.
- **UPDATE**: indicates that the Oracle server fires the trigger whenever an UPDATE statement changes a value in any column of the table.

Keyword and parameters

- **ON**: Specifies the name of the table, which the trigger is to be created.
- **REFERENCING**: specifies correlation names.
- **FOR EACH ROW**: designates the trigger to be a row trigger.
- **WHEN**: specifies the trigger restriction.

Creating Triggers

- CREATE OR REPLACE TRIGGER Print_salary_changes BEFORE DELETE OR INSERT OR UPDATE ON Emp_tab FOR EACH ROW WHEN (new.Empno > 0)

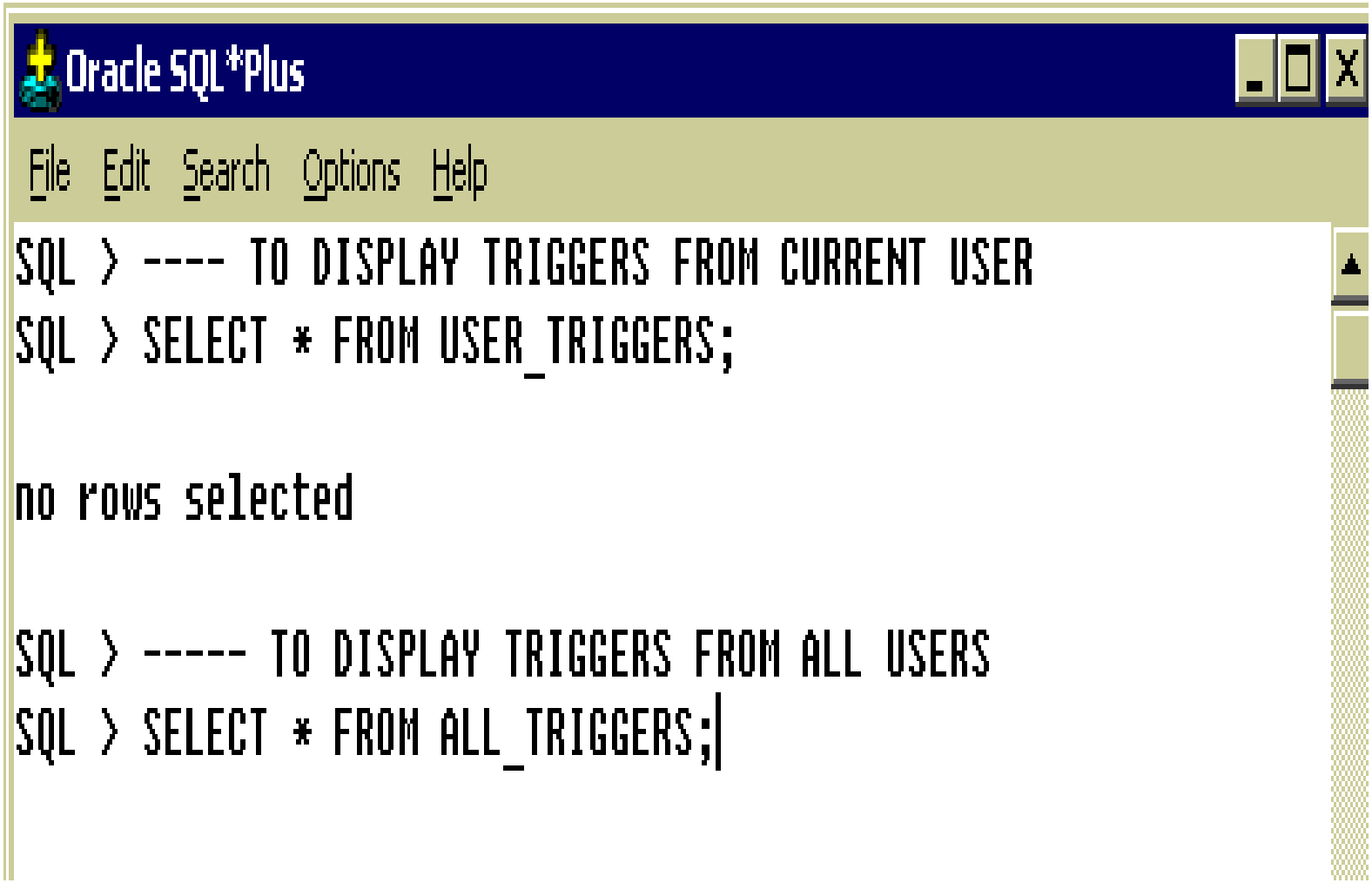
  DECLARE sal_diff number;

  BEGIN sal_diff := :new.sal - :old.sal;
dbms_output.put('Old salary: ' || :old.sal);
dbms_output.put(' New salary: ' || :new.sal);
dbms_output.put_line(' Difference ' || sal_diff);

  END;

Points to ponder

- A trigger cannot include COMMIT, SAVEPOINT and ROLLBACK.
- We can use only one trigger of a particular type .
- A table can have any number of triggers.


Oracle SQL*Plus
File Edit Search Options Help
SQL > ---- TO DISPLAY TRIGGERS FROM CURRENT USER
SQL > SELECT * FROM USER_TRIGGERS;
no rows selected
SQL > ----- TO DISPLAY TRIGGERS FROM ALL USERS
SQL > SELECT * FROM ALL_TRIGGERS;

Viewing Defined Triggers

To view a list of all defined triggers, use:

- select trigger_name from user_triggers;
- For more details on a particular trigger:
- select trigger_type, triggering_event, table_name, referencing_names, trigger_body from user_triggers where trigger_name = '<trigger_name>';

Disabling Triggers

- To disable or enable a trigger:
- alter trigger <trigger_name> {disable|enable};

Schema Refinement Problems caused by Redundancy

INFORMAL DESIGN GUIDELINES FOR RELATIONAL SCHEMAS

- Semantics of attributes.
- Reducing the redundant values in tuples.
- Reducing the null values in tuples.
- Disallowing the possibility of generating spurious tuples.

Semantics of attributes

- Semantics – specifies how to interpret the attribute values stored in a tuple of the relation.
- Name of the attribute must have some meaning.
- Relationship among the relations must be clear.

Employee	Emp_id	Emp_name	Street	City

Company	Company_id	Company_name	C_City

Works	Emp_id	Company_id	Salary

Manages	Manager_id	Manager_name	Emp_id

GUIDELINES 1:

- Design a relation schema so that it is easy to explain its meaning.
- Do not combine attributes from multiple entity types into a single relation.

Reducing the redundant values in tuples

- One goal of schema design is to minimize the storage space that the base relations occupy.
- Grouping attributes into relation schemas has a significant effect on storage space.
- Mixing attributes of multiple entities may cause problems
- Information is stored redundantly wasting storage
- Problems with update anomalies
  - Insertion anomalies
  - Deletion anomalies
  - Modification anomalies

Emp_com

Emp_id	Emp_name	Street	City	Company_id	Company_name	C_City

Combining the 2 relations ‘Employee’ and ‘Company’ into one relation namely: ‘Emp_com’

Insertion Anomalies

Insertion anomalies can be differentiated into two types:

- To insert a new tuple into emp_com, either include the attribute values for the company that the employee works for, or null if the employee does not work for a company yet.
- To insert a new company that has no employees as yet in the emp_com relation. Place the null values in the attributes for employee.

Deletion Anomalies

- If we delete from emp_com, an employee tuple that represent the last employee working for a particular company, the information concerning that company, is lost from the database.

Modification Anomalies

In Emp_com if the value of the attributes for a particular company is to be changed, the update is required for all tuples of 'emp_com' relation who work in that company, otherwise database will become inconsistent.

GUIDELINE 2 : Design a schema that does not suffer from the insertion, deletion and update anomalies. If there are any present, then ensure that applications that update the database will operate correctly.

Null Values in Tuples

If many of the attributes do not apply to all tuples in the relation , we end up with many nulls in those tuples. Nulls can have multiple interpretations such as,

- Attribute not applicable or invalid
- Attribute value unknown (may exist)
- Value known to exist, but unavailable

For example, if only 10 percent of employees have individual offices, there is little justification for including an attribute office_number in the employee relation.

GUIDELINE 3: Relations should be designed such that their tuples will have as few NULL values as possible

Generation of SpuriousTuples

- At the time of joining of two tables the extra tuples which are included in the join are extra, i.e. which are not required are spurious tuples or dangling tuples or wrong tuple.

GUIDELINE 4 : 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.

Functional Dependencies

- Functional dependencies (FDs) are used to specify formal measures of the "goodness" of relational designs.
- A Functional dependency define the properties of the database schema.
- FDs are **constraints** that are derived from the meaning and interrelationships of the data attributes.
- An FD is a property of the attributes in the schema R.
- The constraint must hold on *every relation instance* r(R).

FUNCTIONAL DEPENDENCIES

- A Functional dependency denoted by X ⟹ Y between two sets of attributes X and Y that are subsets of R specifies a constraint on the possible tuple that can form a relation state r of R.
- The constraint is for any two tuples t1 and t2 in r if t1[X] = t2[X] then they have t1[Y] = t2[Y].
- This means the value of X component of a tuple uniquely determines the value of component Y.

Full functional dependency

- For a relation schema R and a FD X->Y is a **full functional dependent** if you remove any attribute from X the dependency does not hold any more.
- {X-A}-> Y is no longer true.

Partial dependency

- A **partial dependency** X → Y, then there is some attribute on X that can be removed from X and yet the dependency stills holds.

Transitive dependency

- Given a relation R with the functional dependencies F, if there a set of attribute Z that are neither a primary or candidate key and both X → Y and Y → Z holds, , then Z is **transitively dependent** on X.
- (X->Z) is transitive dependent

TRIVIAL AND NON-TRIVIAL FD

- If X -> Y hold and X is a superset of Y, then this is called a Trivial Functional Dependency. All FDs which are not trivial is called Non-trivial FD.

Example:

- {Employee ID, EmployeeAddress} → {EmployeeAddress} is trivial.

PRIME & NON–PRIME ATTRIBUTES

If an attribute is a candidate key or a subset of candidate key then it is called a prime attribute.

For ex,

A	B	C	D

'A' is a candidate key.

Combination of C & D is a candidate key.

Hence; A, C, D are prime attributes and B is a non-prime attribute.

Inference Rules for FDs(Armstrong Axioms)

- Axioms, or rules of inference, provide a simpler technique for reasoning about functional dependencies.
- **Armstrong's axioms** are a set of axioms (inference rules) used to infer all the functional dependencies on a relational database.
- The very first Armstrong is the person who gives a set of inference rules.

Armstrong's inference rules

The following six rules (IR1 through IR6) are well-known inference rules for FDs.

1. IR1 – Reflective rule: $Y \subseteq X$ then $X \rightarrow Y$.
2. IR2 – Augmentation rule: $X \rightarrow Y$, then $XZ \rightarrow YZ$.
3. IR3 – Transitive rule: $\{X \rightarrow Y, Y \rightarrow Z\}$ then $X \rightarrow Z$

Inference Rules for FDs (contd…

Some **additional inference rules** that are useful:

(**Union**) If X -> Y and X -> Z, then X -> YZ

(**Decomposition**) If X -> YZ, then X -> Y and X -> Z

(**Psuedotransitivity**) If X -> Y and WY -> Z, then WX -> Z

Questions on inference rules for FD

1. Let set of FDs are given

F={ A->B,C->X, BX->Z}

Prove or disprove that AC->Z

Soln:

C->X and BX->Z then **CB->Z**

By rule A6: Pseudo transitivity

A->B(given) and **CB->Z then AC->Z**

By rule A6: Pseudo transitivity

Questions on inference rules for FD

2. Let set of FDs are given

F={ A->B,C->D, C is subset of B}

Prove or disprove that A->C

Soln:

C is subset of B then **B->C**

By rule A1: Reflexivity

A->B(given) and **B->C then A-> C**

By rule A3: Transitivity

Questions on inference rules for FD

3. Let set of FDs are given

F={ A->B,BC->D}

Prove or disprove that AC->D

Sol^{n}:

A->B and BC->D then **AC->D**

By rule A6: Pseudo transitivity

Questions on inference rules for FD

4. Let set of FDs are given

F={ A->B,BC->D}

Prove or disprove that AD-> B

Soln:

A->B then **AD->BD**

By rule A2: Augmentation

AD->BD then AD->B and AD->D

By rule A5: Decomposition

Questions on inference rules for FD

5. Let set of FDs are given

F={ A->B,A->C,BC->D}

Prove that A->D

Soln:

A->B and A->C then **A->BC**

By rule A4: union

A->BC and BC->D(given) **then A->D**

By rule A3: Transitivity

UN-NORMALIZED RELATION

A table is said to be un-normalized if each row contains multiple set of values for some of the columns, these multiple values in a single row are also called non-atomic values.

UN-NORMALIZED RELATION

SID	Sname	Saddress	Phoneno	DOB
101	Animesh	Bhopal	276666	17-08-87
102	Ravi	Delhi	566666	25-06-78
103	Nishant	Indore	665555 453333 656566	04-3-56
104	Ganesh	Bhopal	555555 876544	12-5-78

Introduction to Normalization

- **Normalization** is the process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations while ensuring data integrity, eliminating data redundancy and minimizing the insertion, deletion and update anomalies.

Objective of normalization

- To make it feasible to represent any relation in the database.
- To free relations from undesirable insertion, update and deletion anomalies.

Benefits of Normalization

- Improve storage efficiency
- Quicker updates
- Less data inconsistency
- Clearer data relationships
- Easier to add data
- Flexible Structure

Normal form

- Basically the normal form of data indicate how much redundancy is in that data.
- Types of normal form
  - 1NF
  - 2NF
  - 3NF
  - BCNF **(Boyce-Codd Normal Form)**
  - 4NF
  - 5NF (PJNF)
  - DKNF(Domain Key Normal Form)

Normal Forms

1NF	*Keys; No repeating groups*
2NF	*No partial dependencies*
3NF	*No transitive dependencies*
BCNF	*Determinants are candidate keys*
4NF	*No multivalued dependencies*

First Normal Form

- It states that the value of any attribute in any relation must be a **single atomic value**.
- In other words, only one value is associated with each attribute and the value is not a set of values or a list of values (No composite Attributes).

(a)

DEPARTMENT

(b)

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATIONS
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATION
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

DNAME	DNUMBER	DMGRSSN	DLOCATION1	DLOCATION2	DLOCATION3
Research	5	333445555	Bellaire	Sugarland	Houston
Administration	4	987654321	Stafford	Null	Null
Headquarters	1	888665555	Houston	Null	Null

DNAME	DNUMBER	DMGRSSN
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

DNUMBER	DLOCATION
5	Bellaire
5	Sugarland
5	Houston
4	Stafford
1	Houston

SECOND NORMAL FORM

- A relation schema R is in second normal form (2NF) if it is in the 1NF and if all non-prime attributes of R are fully functionally dependent on the primary keys.
- R can be decomposed into 2NF relations via the process of 2NF normalization.

Steps to convert 1NF to 2NF

- First find all candidate keys.
- Determine all the prime and nonprime attributes separately.
- Check that no nonprime attribute is partially dependent on any key.

Sales order relation

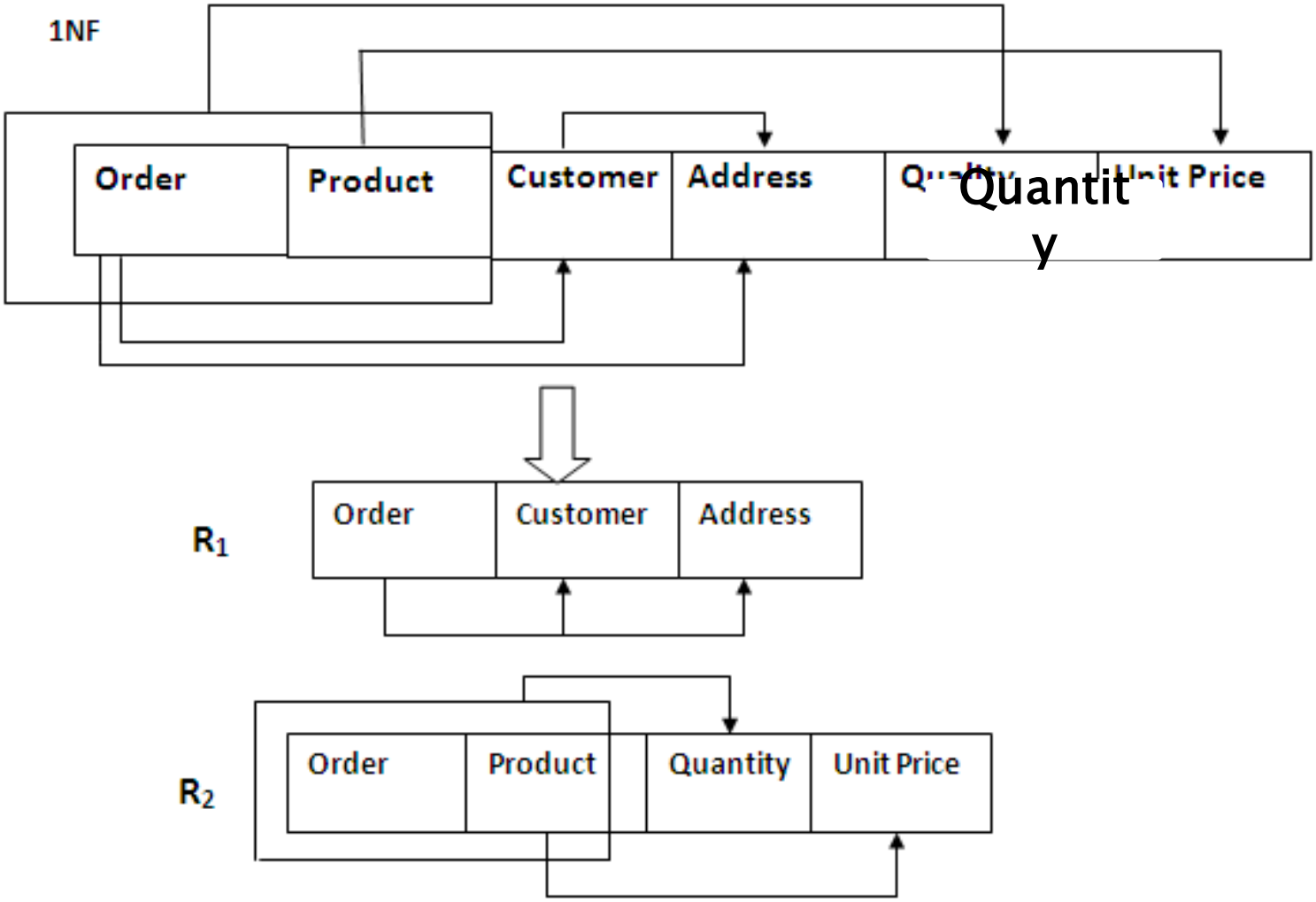
Order	Product	Customer	Address	Qua-ntity	Unit price
S1	P1	Animesh	Bhopal	300	500
S1	P2	Animesh	Bhopal	100	700
S2	P3	Ganesh	Raipur	500	200
S3	P4	Rahul	Guna	200	1000
S3	P1	Rahul	Guna	450	500
S4	P5	Nishant	Indore	400	900
S5	P2	Ranjit	Jabalpur	250	700

2 NF

- 2NF means no partial dependencies on the key. But we have
- {order}-> {customer , address}
- {product}-> {unit price}

So the relation is in 1NF but not in 2NF. So we convert the relation sale order in 2NF.

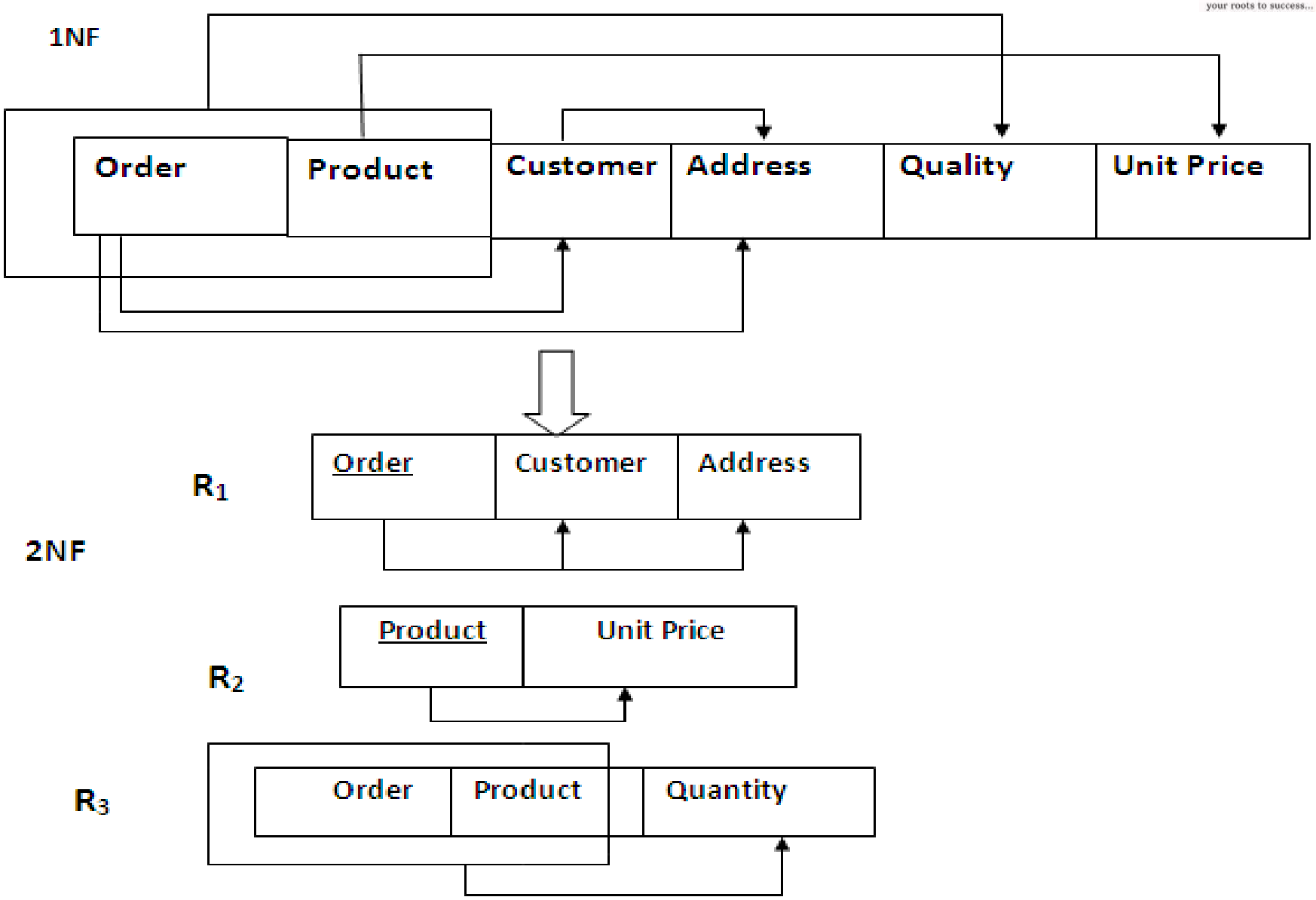
Conversion of 1NF to 2NF



Continued…….

- R_1 is now in 2NF, but there is still partial FD in R_2.
- Product-> unit price

Relation in 2NF



Sample value

Order	Customer	Address
S_1	Animesh	Bhopal
S_2	Gnesh	Raipur
S_3	Rahul	Guna
S_4	Nishant	Indore
S_5	Rajit	Jabalpur

(a) Relation R_1

Product	Unit-price
P_1	500
P_2	700
P_3	200
P_4	1000
P_5	900

(b) Relation R_2

Order	Product	uantity
S_1	P_1	300
S_1	P_2	100
S_2	P_3	500
S_3	P_4	200
S_3	P_1	450
S_4	P_5	400
S_5	P_2	250

(c) Relation R_3

Lossy decomposition

- There should be no loss of information due to decomposition.

Lossless Join decomposition

Recombination using relational join produces exactly same as pre decomposition.

Lossless-Join

Condition for lossless join.

1) All attributes of an original schema (R) must appear in the decomposition(R_1 , R_2)

$R = R_1 \cup R_2$

2) At least one of the following functional dependencies holds

- If $R_1 \cap R_2 = R_1$
- If $R_1 \cap R_2 = R_2$

Dependency preservation

- Dependency preservation is another important requirement since a dependency is a constraint on the database and if X-> Y holds than we know that the two attributes are closely related and it would be useful if both attributes appeared in the same relation so that the dependency can be checked easily.

Dependency preservation

- Dependency preservation is another important requirement since a dependency is a constraint on the database and if X-> Y holds than we know that the two attributes are closely related and it would be useful if both attributes appeared in the same relation so that the dependency can be checked easily.

Dependency preservation Condition

- If R is decomposed into X and Y then

$$(F_x \cup F_y)^+ = F^+$$

Irreducible sets(minimal sets or canonical cover) of FD

- A canonical cover F_c for F is a set of dependencies such that F logically implies all dependencies in F_c, and F_c logically implies all dependencies in F.
- A set of FDs S is irreducible if and only if
  - The right-hand side (the dependent) of each FD in S involves just one attribute.
  - The left-hand side (the determinant) of each FD in S is irreducible.

Normalization

- Normalization is the process of putting one fact in one appropriate place.
- This optimizes updates at the expense of retrievals.
- When a fact is stored in only one place, then data retrieving is easy. But, When a fact is stored in different places, then retrieving many different but related facts usually requires going to many different places.
- This tends to slow the retrieval process.
- There are two strategies for dealing in this type of situation.

1st method

- The preferred method is to keep the logical design normalized, but allow the DBMS to store additional redundant information on disk to optimize query response.
- In this case it is the DBMS software's responsibility to ensure that any redundant copies are kept consistent.
- This method is often implemented in SQL as indexed views(Microsoft SQL) or materialized views(oracle).

2nd method

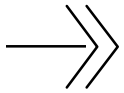
- The more usual approach is to denormalize the logical data design.
- With care this can achieve a similar improvement in query response, but at a cost-it is now the database designer's responsibility to ensure that the denormalized database does not become inconsistence.
- This is done by creating rules in the database called constraints, that specify how the redundant copies of information must be kept synchronized.

Denormalization

- A denormalized data model is not the same as a data model that has not been normalized, denormalization should only take place after a satisfactory level of normalization has taken and that any required constrainits and/ or rules have been created to deal with the inherent anomalies in the design.

Multivalued Dependencies

- A multivalued dependency (MVD) exists between two fields X and Y, when a distinct value of X is directly associated with two or more values of Y.
- An MVD is represented as X Y and can be read as:
  - "the value of X determines multiple values of Y"

    Or 
  - "multiple values of Y are functionally dependent on the value of X.

Teaching database

Course	Book	Lecturer
AHA	Silberschatz	John D
AHA	Nederpelt	William M
AHA	Silberschatz	William M
AHA	Nederpelt	John D
AHA	Silberschatz	Christian G
AHA	Nederpelt	Christian G
OSO	Silberschatz	John D
OSO	Silberschatz	William M

Types of MVD

- A multi-valued dependency can be further defined as being trivial or nontrivial.
  - A MVD A ->> B in relation R is defined as being trivial if (a) B is a subset of A *or* (b) $A \cup B = R$.
  - A MVD is defined as being nontrivial if neither (a) nor (b) are satisfied.

Definition MVD

- The multivalued dependency X->> Y holds in a relation R if whenever we have two tuples of R that agree in all the attributes of X, then we can swap their Y components and get two new tuples that are also in R.
- If two tuples t_1 and t_2 exist in 'r' such that $t_1[X] = t_2[X]$, then two tuples t_3 and t_4 should also exist in 'r' with the following properties,
  - $t_1[X] = t_2[X] = t_3[X] = t_4[X]$
  - $t_1[Y] = t_3[Y]$ and $t_2[Y] = t_4[Y]$
  - $t_1[Z] = t_4[Z]$ and $t_2[Z] = t_3[Z]$

Inference Rules for Multivalued Dependencies

Rule 1: **Complementation Rule**

It states that if in a relation R, X->-> Y exists then X->->R-(X UY) is true.

Rule 2: **Augmentation Rule**

It states that if in a relation R, X->->Y and T $\subseteq$ R And P $\subseteq$ T, then TX->-> PY

Rule 3: **Transitive Rule**

If X->->Y and Y->->P, then X->->(P-Y)

Rule 4: **Replication Rule**

If X -> Y then X->> Y but not vice versa is not true .

Inference Rules for Multivalued Dependencies

Rule 5: **Coalescence Rule**

If X->->Y and $P \subseteq Y$, then there exists Q such that
(a) $Q \subseteq R$ (b) $Q \cap Y$ is empty (c) Q->P then X->P

Rule 6: **Union Rule**

If X->->Y and X->->P, then X->-> $P \cup Y$ or X->->PY

Rule 7: **Intersection Rule**

If X->->Y and X->->P, then X->-> $Y \cap P$

Rule 8: **Difference Rule**

If X->->Y and X->->P, then X->-> (Y – P) and X->-> (P- Y)

Fourth Normal Form (4NF)

A relation schema *R* is in **4NF** with respect to a set of dependencies *F* if,

- It is in BCNF and
- For every nontrivial multivalued dependency *X* —>> *Y* , *X* is a superkey — that is, *X* is either a candidate key or a superset thereof.

Pizza Delivery Permutations

Restaurant	Pizza Variety	Delivery Area
Vincenzo's Pizza	Thick Crust	Springfield
Vincenzo's Pizza	Thick Crust	Shelbyville
Vincenzo's Pizza	Thin Crust	Springfield
Vincenzo's	ThinCrust	Shelbyville
Elite Pizza	Thin Crust	Capital City
Elite Pizza	Stuffed Crust	Capital City
A1 Pizza	Thick Crust	Springfield
A1 Pizza	Thick Crust	Shelbyville
A1 Pizza	Thick Crust	Capital City
A1 Pizza	Stuffed Crust	Springfield
A1 Pizza	Stuffed Crust	Shelbyville
A1 Pizza	Stuffed Crust	Capital City

- The table has no non-key attributes because its only key is {Restaurant, Pizza Variety, Delivery Area}.
- The problem is that the table features two non-trivial multivalued dependencies on the {Restaurant} attribute (which is not a superkey).
- The dependencies are:

{Restaurant} →→ {Pizza Variety}

{Restaurant} →→ {Delivery Area}

To satisfy 4NF, we must place the facts about varieties offered into a different table from the facts about delivery areas:

Varieties By Restaurant

Restaurant	Pizza Variety
Vincenzo's Pizza	Thick Crust
Vincenzo's Pizza	Thin Crust
Elite Pizza	Thin Crust
Elite Pizza	Stuffed Crust
A1 Pizza	Thick Crust
A1 Pizza	Stuffed Crust

Delivery Areas By Restaurant

Restaurant	Delivery Area
Vincenzo's Pizza	Springfield
Vincenzo's Pizza	Shelbyville
Elite Pizza	Capital City
A1 Pizza	Springfield
A1 Pizza	Shelbyville
A1 Pizza	Capital City