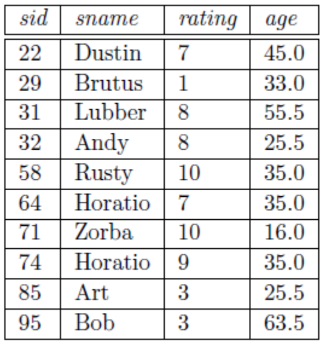
UNIT-3

OVERVIEW:

- The SQL language has several aspects to it.
    - DML (Data Manipulation Language)
    - DDL (Data Definition Language)
    - Triggers and Advanced Integrity Constraints
    - Embedded and Dynamic SQL
        - SQL code to be called from a host language such as C or COBOL or JAVA
        - Dynamic SQL (Query to be constructed (and executed) as run-time
    - Client-Server Execution and Remote Database Access:
        - These commands control how a client application program can connect to an SQL db server; or access db over a network.
    - Transaction Management (explicitly control aspects of how a transaction is to be executed)
    - Security: mechanisms to control users access to data objects such as tables & views
    - Advanced Features: OO features, recursive queries, DS queries, DM, spatial data etc.

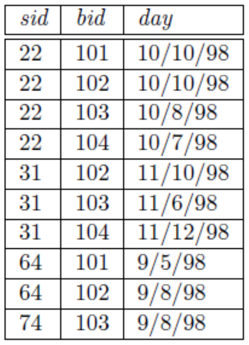
THE FORM A BASIC SQL QUERY

- The basic form of an SQL query is as follows:
- SELECT [DISTINCT] select-list
- FROM from-list
- WHERE qualification
- Every query must have a SELECT clause, which specifies columns to be retained in the result, and a FROM clause, which specifies a cross-product of tables.
- The optional WHERE Clause specifies selection conditions on the tables mentioned in the FROM clause.



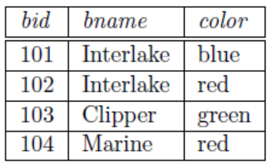
sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Figure 5.1 An Instance *S*3 of Sailors



sid	*bid*	*day*
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Figure 5.2 An Instance *R*2 of Reserves



bid	*bname*	*color*
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Figure 5.3 An Instance *B*1 of Boats

(Q15) Find the names and ages of all sailors.
SELECT DISTINCT S.sname, S.age FROM Sailors S

(Q11) ***Find all sailors with a rating above 7.***
SELECT S.sid, S.sname, S.rating, S.age FROM Sailors AS S WHERE S.rating > 7

SELECT clause used to do *projection* Whereas selections in the relational algebra sense are expressed using the WHERE clause
The **from-list** in the FROM clause is a list of table names.
Table name can be followed by a range variable; (useful when same table name appears more than once)

The select-list is a list of column names of tables named in the from-list.
The qualification in the WHERE clause is a Boolean combination (using connectives AND, OR and NOT) of Form expression op expression, where op is comparison operators. Expression is column name or constants or an expression

Conceptual Evaluation Strategy:
1. Compute the cross-product of the tables in the **from-list**
2. Delete rows in the cross-product that fail the qualification conditions.
3. Delete all columns that do not appear in the select-list.
4. If DISTINCT is specified, eliminate duplicate rows.

(Q1) Find the names of sailors who have reserved boat number 103.

SELECT S.sname FROM Sailors S, Reserves R WHERE S.sid = R.sid AND R.bid = 103

(Q16) ***Find the sids of sailors who have reserved a red boat.***
SELECT R.sid FROM Boats B, Reserves R WHERE B.bid = R.bid AND B.color = ‘red’

(Q2) ***Find the names of sailors who have reserved a red boat***
SELECT S.sname FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = ‘red’

(Q3) ***Find the colors of boats reserved by Lubber*****.**
SELECT B.color FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid and R.bid = B.bid AND S.sname = ‘Lubber’

(Q4) ***Find the names of sailors who have reserved at least one boat.***
SELECT S.sname FROM Sailors S, Reserves R WHERE S.sid = R.sid

Expressions and Strings in the SELECT Command

SQL supports a more general version of the select-list than just a list of columns.

Each item in a select-list can be of the form expression AS column name, where expression is any arithmetic or string expression over column names and constants, and column name is a new name for this column in the output of the query.

(Q17) Compute increments for the ratings of persons who have sailed two different boats on the same day

SELECT S.sname, S.rating+1 AS rating FROM Sailors S, Reserves R1, Reserves R2 WHERE S.sid = R1.sid AND S.sid = R2.sid AND R1.day = R2.day AND R1.bid <> R2.bid

In addition, SQL provides support for pattern matching through the LIKE operator, along with the use of the wild-card symbols %

Thus ‘AB%’ denotes a pattern matching every string that contains at least three characters, with the second and third characters being A and B respectively.

(Q18) Find the ages of sailors whose name begins and ends with B and has at least three characters

SELECT S.age FROM Sailors SWHERE S.sname LIKE ‘B_%B’

UNION, INTERSECT, AND EXCEPT

SQL provides three set-manipulation constructs that extend the basic query form presented earlier.
UNION
INTERSECT
EXCEPT
IN (to check if an element is in a given set)
EXISTS(to check if a set is empty).
NOT

(Q5) Find the names of sailors who have reserved a red or a green boat.

SELECT S.sname FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid and R.bid = B.bid AND (B.color = ‘red’ OR B.color = ‘green’)

Or Q5 can be rewritten as follows:

SELECT S.sname FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = ‘red’

UNION

SELECT S2.sname FROM Sailors S2, Boats B2, Reserves R2 WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = ‘green’ AND B1.color = ‘red’ AND B2.color = ‘green’

(Q6) Find the names of sailors who have reserved both a red and a green boat

SELECT S.sname FROM Sailors S, Reserves R1, Boats B1, Reserves R2, Boats B2 where S.sid = R1.sid AND R1.bid = B1.bid AND S.sid = R2.sid AND R2.bid = B2.bid

OR Q6 can be rewritten as follows:

SELECT S.sname FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = ‘red’

INTERSECT

SELECT S2.sname FROM Sailors S2, Boats B2, Reserves R2 WHERE S2.sid = R2.bid AND R2.bid = B2.bid AND B2.color = ‘green’

(Q19) ***Find the sids of all sailors who have reserved red boats but not green boats.***

SELECT S.sid FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = ‘red’
EXCEPT
SELECT S2.sid FROM Sailors S2, Reserves R2, Boats B2 WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = ‘green’

OR

SELECT R.sid FROM Boats B, Reserves R WHERE R.bid = B.bid AND B.color = ‘red’
EXCEPT
SELECT R2.sid FROM Boats B2, Reserves R2 WHERE R2.bid = B2.bid AND B2.color = ‘green’

20. Find all sailors who have a rating of 10 or reserved boat 104.

```
SELECT S.sid FROM   Sailors S WHERE S.rating = 10
UNION
SELECT R.sid FROM   Reserves R WHERE R.bid = 104
```

NESTED QUERIES

A **Nested Query** is a query that has another query within it; the embedded query is called a **subquery.**

1. Find the names of sailors who have reserved boat 103.

```
SELECT S.sname FROM Sailors S WHERE S.sid IN ( SELECT R.sid FROM Reserves R WHERE R.bid = 103 )
```

2. Find the names of sailors who have reserved a red boat.

```
SELECT S.sname FROM Sailors S WHERE S.sid IN ( SELECT R.sid FROM Reserves R
WHERE R.bid IN ( SELECT B.bid FROM Boats B WHERE B.color = `red' )
```

(Q21) Find the names of sailors who have not *reserved a red boat.*

```
SELECT S.sname FROM Sailors S WHERE S.sid NOT IN ( SELECT R.sid
FROM Reserves R WHERE R.bid IN ( SELECT B.bid FROM Boats B WHERE B.color = `red' )
```

(Q1) Find the names of sailors who have reserved boat number 103.

```
SELECT S.sname FROM Sailors S WHERE EXISTS ( SELECT * FROM Reserves R
WHERE R.bid = 103 AND R.sid = S.sid )
```

(Q22) Find sailors whose rating is better than some sailor called Horatio.

```
SELECT S.sid FROM Sailors S WHERE S.rating > ANY ( SELECT S2.rating
FROM Sailors S2 WHERE S2.sname = `Horatio' )
```

(Q24) Find the sailors with the highest rating.

```
SELECT S.sid FROM Sailors S WHERE S.rating >= ALL (SELECT S2.rating FROM Sailors S2 )
```

(Q6) Find the names of sailors who have reserved both a red and a green boat.

```
SELECT S.sname FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = R.bid AND B.color = ‘red’
          AND S.sid IN (SELECT S2.sid
                              FROM Sailors S2, Boats B2, Reserves R2
                              WHERE S2.sid = R2.sid AND R2.bid = B2.bid
                                        AND B2.color = ‘green’)
```

(Q9) Find the names of sailors who have reserved all boats.

```
SELECT S.sname FROM Sailors S
WHERE NOT EXISTS (( SELECT B.bid FROM Boats B )
EXCEPT
(SELECT R.bid FROM Reserves R WHERE R.sid = S.sid ))
```

AGGREGATE OPERATIONS

1. COUNT ([DISTINCT] A): The number of (unique) values in the A column.
2. SUM ([DISTINCT] A): The sum of all (unique) values in the A column.
3. AVG ([DISTINCT] A): The average of all (unique) values in the A column.
4. MAX (A): The maximum value in the A column.
5. MIN (A): The minimum value in the A column.

(Q25) Find the average age of all sailors.

SELECT AVG (S.age) FROM Sailors S.

(Q26) Find the average age of sailors with a rating of 10.

SELECT AVG (S.age) FROM Sailors S WHERE S.rating = 10

(Q27) Find the name and age of the oldest sailor.

SELECT S.sname, MAX (S.age) FROM Sailors S

(Q28) Count the number of sailors.

SELECT COUNT(*) FROM Sailors S

(Q29) Count the number of different sailor names.

SELECT COUNT (DISTINCT S.sname) FROM Sailors S

(Q30) Find the names of sailors who are older than the oldest sailor with a rating of 10.

SELECT S.sname FROM Sailors S
WHERE S.age > (SELECT MAX (S2.age) FROM Sailors S2 WHERE S2.rating = 10)

The GROUP BY and HAVING Clauses

(Q31) Find the names of the youngest sailor for each rating level.
If we know the that ratings are integers in the range 1 to 10, we could write 10 queries of the form:

SELECT MIN (S.age) FROM Sailors S WHERE S.rating = i;

Where I = 1,2….10. Writing 10 such queries is tedious. More important, we may not know what rating levels exists in advance.

To write such queries, we need a major extension to the basic SQL query form, namely, the GROUP BY clause.

The extension also includes an option HAVING clause that can be used to specify qualifications over groups.

The general form of an SQL query with these extensions is:

SELECT [DISTINCT] **select-list**
FROM **from-list**
WHERE **qualification**
GROUP BY **grouping-list**
HAVING **group-qualification**

Using the GROUP BY clause, we can write Q31 as follows:

SELECT S.rating, MIN (S.age) FROM Sailors S GROUP BY S.rating

(Q32) Find the age of the youngest sailor who is eligible to vote (i.e., is at least 18 years old) for each rating level with at least two such sailors
SELECT S.rating, MIN (S.age) AS minage FROM Sailors S WHERE S.age >= 18 GROUP BY S.rating
HAVING COUNT (*) > 1

(Q33) For each red boat, find the number of reservations for this boat
SELECT B.bid, COUNT (*) AS sailorcount FROM Boats B, Reserves R WHERE R.bid = B.bid AND B.color = `red'
GROUP BY B.bid

(Q34) Find the average age of sailors for each rating level that has at least two sailors
SELECT S.rating, AVG (S.age) AS avgage FROM Sailors S GROUP BY S.rating HAVING COUNT (*) > 1

(Q35) Find the average age of sailors who are of voting age (i.e., at least 18 years old)
for each rating level that has at least two sailors.

(Q36) Find the average age of sailors who are of voting a ge (i.e., at least 18 years old) for each rating level that has at least two such *sailors*
SELECT S.rating, AVG (S.age) AS avgage FROM Sailors S WHERE S. age > 18 GROUP BY S.rating
HAVING 1 < (SELECT COUNT (*) FROM Sailors S2 WHERE S.rating = S2.rating AND S2.age >= 18)

(Q37) Find those ratings for which the average age of sailors is the minimum overall Ratings
SELECT S.rating FROM Sailors S WHERE AVG (S.age) = (SELECT MIN (AVG (S2.age)) FROM Sailors S2
GROUP BY S2.rating)

COMPLEX INTEGRITY CONSTRAINTS IN SQL

Constraints over a Single Table

We can specify complex constraints over a single table using **table constraints**, which have the form CHECK *conditional-expression.* For example, to ensure that *rating* must be an integer in the range 1 to 10, we could use:

```
CREATE TABLE Sailors ( sid INTEGER,
                       sname CHAR(10),
                       rating INTEGER,
                       age REAL,
                       PRIMARY KEY (sid),
                       CHECK ( rating >= 1 AND rating <= 10 ))
```

To enforce the constraint that Interlake boats cannot be reserved, we could use:

```
CREATE TABLE Reserves (     sid INTEGER,
                            bid INTEGER,
                            day DATE,
                            FOREIGN KEY (sid) REFERENCES Sailors
                            FOREIGN KEY (bid) REFERENCES Boats
                            CONSTRAINT noInterlakeRes
                            CHECK ( `Interlake' <>
                                    ( SELECT B.bname
                                    FROM Boats B
                                    WHERE B.bid = Reserves.bid )))
```

Domain Constraints and Distinct Types

A user can de_ne a new domain using the CREATE DOMAIN statement, which makes uses of CHECK constraints.

```
CREATE DOMAIN ratingval INTEGER DEFAULT 0 CHECK ( VALUE >= 1 AND VALUE <= 10 )
```

INTEGER is the **base 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; in de_ning this constraint, we use the keyword VALUE to refer to a value in the domain.

The optional DEFAULT keyword is used to associate a default value with a domain. If the domain ratingval is used for a column in some relation, and no value is entered for this column in an inserted tuple, the default value 0 associated with ratingval is used.

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 any one table.

As an example, suppose that we wish to 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 ))
```

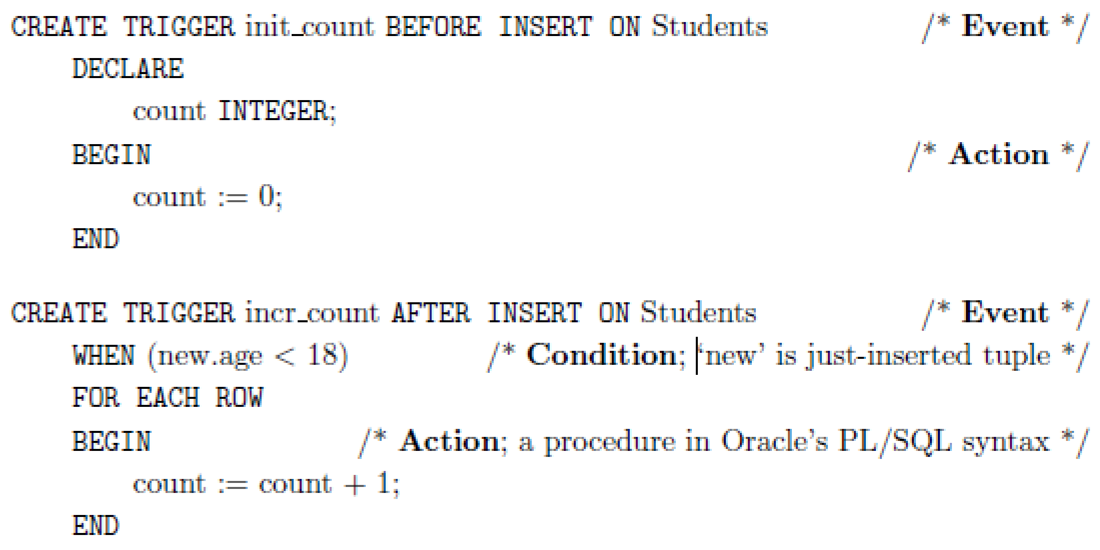
TRIGGERS AND ACTIVE DATABASES

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 databasethat 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; users may not even be aware that a trigger was executed as a side effect of their program.

Examples of Triggers in SQL



```
CREATE TRIGGER init_count BEFORE INSERT ON Students          /* Event */
    DECLARE
        count INTEGER;
    BEGIN                                                     /* Action */
        count := 0;
    END

CREATE TRIGGER incr_count AFTER INSERT ON Students           /* Event */
    WHEN (new.age < 18)         /* Condition; 'new' is just-inserted tuple */
    FOR EACH ROW
    BEGIN               /* Action; a procedure in Oracle's PL/SQL syntax */
        count := count + 1;
    END
```

Figure 5.19 Examples Illustrating Triggers

DESIGNING ACTIVE DATABASES

Triggers offer a powerful mechanism for dealing with changes to a database, but they must be used with caution. The effect of a collection of triggers can be very complex and maintaining an active database can become very difficult. Often, a judicious use of integrity constraints can replace the use of triggers.

In an active database system, when the DBMS is about to execute a statement that modifies the database, it checks whether some trigger is activated by the statement. If so, the DBMS processes the trigger by evaluating its condition part, and then (if the Condition evaluates to true) executing its action part.

If a statement activates more than one trigger, the DBMS typically processes all of them, in some arbitrary order. An important point is that the execution of the action part of a trigger could in turn activate another trigger. In particular, the execution of the action part of a trigger could again activate the same trigger; such triggers are called
recursive triggers. The potential for such *chain* activations, and the unpredictable order in which a DBMS processes activated triggers, can make it difficult to understand the effect of a collection of triggers.

SCHEMA REFINEMENT AND NORMAL FORMS

INTRODUCTION TO SCHEMA REFINEMENT

Problems Caused by Redundancy

Storing the same information redundantly, that is, in more than one place within a database, can lead to several problems.

- Redundant Storage
- Update Anomalies
- Insertion Anomalies
- Deletion Anomalies

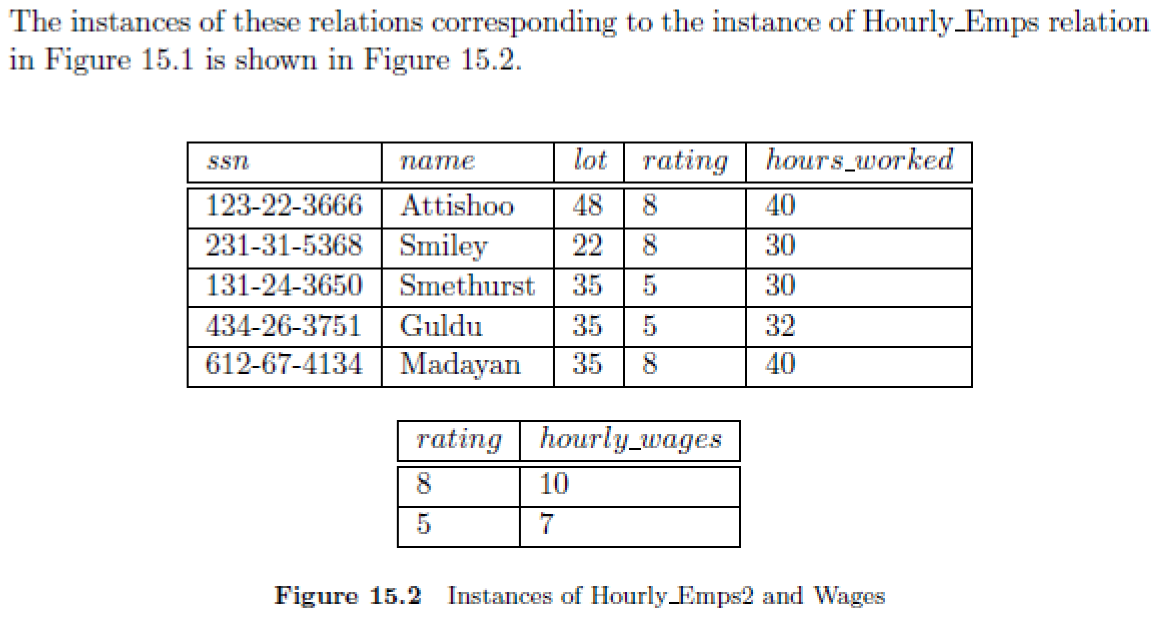
Decompositions

The essential idea is that many problems arising from redundancy can be addressed by replacing a relation with a collection of 'smaller' relations.

A decomposition of a relation schema R consists of replacing the relation schema by two (or more) relation schemas that each contain a subset of the attributes of R and together include all attributes in R. Institutively, we want to store the information in any given instance of R by storing projections of the instance.

We can decompose Hourly_Emps into two relations:

Hourly Emps2(*ssn*, *name*, *lot*, *rating*, *hours worked*)
Wages(*rating*, *hourly wages*)



The instances of these relations corresponding to the instance of Hourly_Emps relation in Figure 15.1 is shown in Figure 15.2.

ssn	*name*	*lot*	*rating*	*hours_worked*
123-22-3666	Attishoo	48	8	40
231-31-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

rating	*hourly_wages*
8	10
5	7

Figure 15.2 Instances of Hourly_Emps2 and Wages

Problems Related to Decomposition

Two important questions must be asked repeatedly:

1. Do we need to decompose a relation?
2. What problems (if any) does a given decomposition cause?

To help with the first question, several normal forms have been proposed for relations. If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise.

W.r.t the second question, two properties of decompositions are of particular interest. The lossless-join property enables us to recover any instance of the decomposed relation from corresponding instances of the smaller relations. The dependency-preservation property enables us to enforce any constraint on the original relation by simply enforcing some constraints on each of the smaller relations.

Functional Dependencies

A **functional dependency** (FD) is a kind of IC that generalizes the concept of a *key*.
Let *R* be a relation schema and let *X* and *Y* be nonempty sets of attributes in *R*. We say that an instance *r* of *R* satis_es the FD *X ! Y* 1 if the following holds for every pair of tuples t_1 and t_2 in *r*:

If *t*1*:X* = *t*2*:X*, then *t*1*:Y* = *t*2*:Y* .

We use the notation *t*1*:X* to refer to the projection of tuple t_1 onto the attributes in *X*, in a natural extension of our TRC notation (see Chapter 4) *t:a* for referring to attribute *a* of tuple *t*. An FD *X->Y* essentially says that if two tuples agree on the values in attributes *X*, they must also agree on the values in attributes *Y.*

Figure 15.3 illustrates the meaning of the FD *AB* -> *C* by showing an instance that satisfies this dependency. The first two tuples show that an FD is not the same as a key constraint: Although the FD is not violated, *AB* is clearly not a key for the
relation. The third and fourth tuples illustrate that if two tuples differ in either the *A* field or the *B* field, they can differ in the *C* field without violating the FD. On the other hand, if we add a tuple *ha*1*; b*1*; c*2*; d*1*i* to the instance shown in this figure, the resulting instance would violate the FD; to see this violation, compare the first tuple
in the figure with the new tuple

A	*B*	*C*	*D*
a1	b1	c1	d1
a1	b1	c1	d2
a1	b2	c2	d1
a2	b1	c3	d1

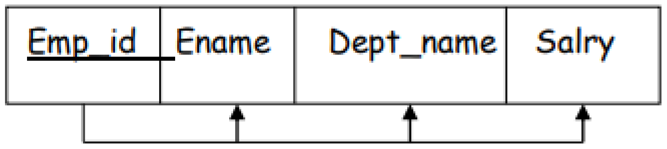
Figure 15.3 An Instance that Satisfies $AB \rightarrow C$

What is Relation?
A relation is a named two–dimensional table of data. Each relation consists of a set of named columns and an arbitrary number of unnamed rows.
For example, a relation named Employee contains following attributes, emp-id, ename, dept name and salary.
marketing 42000

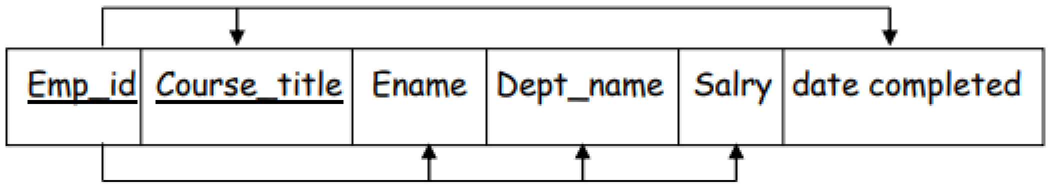
What are the Properties of relations?
The properties of relations are defined on two dimensional tables. They are:
θ Each relation (or table) in a database has a unique name.
θ An entry at the intersection of each row and column is atomic or single. These can be no multiplied atttributes in a relation.
θ Each row is unique, no two rows in a relation are identical.
θ Each attribute or column within a table has a unique name.

θ The sequence of columns (left to right) is insignificant the column of a relation can be interchanged without changing the meaning use of the relation.
θ The sequence of rows (top to bottom) is insignificant. As with column, the rows of relation may be interchanged or stored in any sequence.

A functional dependency is a constraint between two attributes (or) two sets of attributes.
For example, the table EMPLOYEE has 4 columns that are Functionally dependencies on EMP_ID.



Partial functional dependency: It is a functional dependency in which one or more nonkey attributes are functionally dependent on part of the primary key. Consider the following graphical representation, in that some of the attributes are partially depend on primary key



In this example, Ename, Dept_name, and salary are fully functionally depend on Primary key of Emp_id. But Course_title and date_completed are partial functional dependency. In this case, the partial functional dependency creates redundancy in that relation

What is Normal Form? What are steps in Normal Form?

NORMALIZATION: Normalization is the process of decomposing relations to produce smaller, well-structured relation. To produce smaller and well structured relations, the user needs to follow six normal forms

Steps in Normalization:
A normal form is state of relation that result from applying simple rules from regarding functional dependencies relationships between attributes to that relation. The normal form are

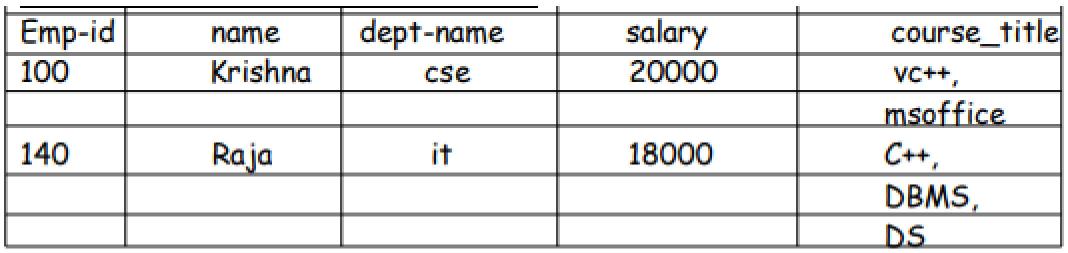
1. First normal form
2. Second normal form
3. Third normal form
4. Boyce/codd normal form
5. Fourth normal form
6. Fifth normal form

1) First Normal Form: Any multi-valued attributes (also called repeating groups) have been removed,
2) Second Normal Form: Any partial functional dependencies have been removed.
3) Third Normal Form: Any transitive dependencies have been removed.
4) Boyce/Codd Normal Form: Any remaining anomalies that result from functional dependencies have been removed.
5) Fourth Normal Form: Any multi-valued dependencies have been removed.
6) Fifth Normal Form: Any remaining anomalies have been removed.

Advantages of Normalized Relations Over the Un-normalized Relations:
The advantages of normalized relations over un-normalized relations are
1) Normalized relation (table) does not contain repeating groups whereas, unnormalized relation (table) contains one or more repeating groups.
2) Normalized relation consists of a primary key. There is no primary key presents in un-normalized relation.

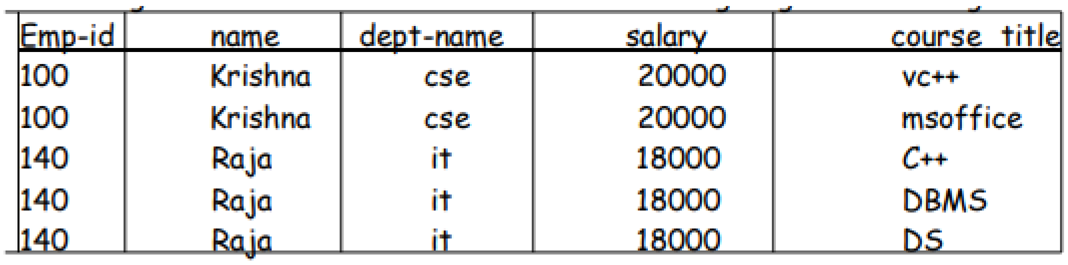
3) Normalization removes the repeating group which occurs many times in a table.
4) With the help of normalization process, we can transform un-normalized table to First Normal Form (1NF) by removing repeating groups from un-normalized tables.
5) Normalized r0000000
elations (tables) gives the more simplified result whereas unnormalized relation gives more complicated results.
6) Normalized relations improve storage efficiency, data integrity and scalability. But un-normalized relations cannot improvise the storage efficiency and data integrity.
7) Normalization results in database consistency, flexible data accesses.

FIRST NORMAL FORM (1NF):
A relation is in first normal form (1NF) contains no multi-Valued attributes. Consider the example employee, that contain multi valued attributes that are removing and converting into single valued attributes
<u>Multi valued attributes in course title</u>



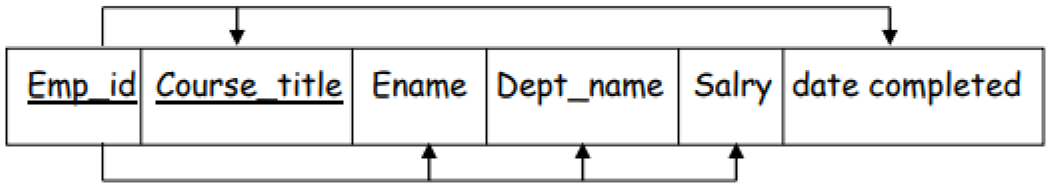
Emp-id	name	dept-name	salary	course_title
100	Krishna	cse	20000	vc++,
				msoffice
140	Raja	it	18000	C++,
				DBMS,
				DS

Removing the multi valued attributes and converting single valued using First NF

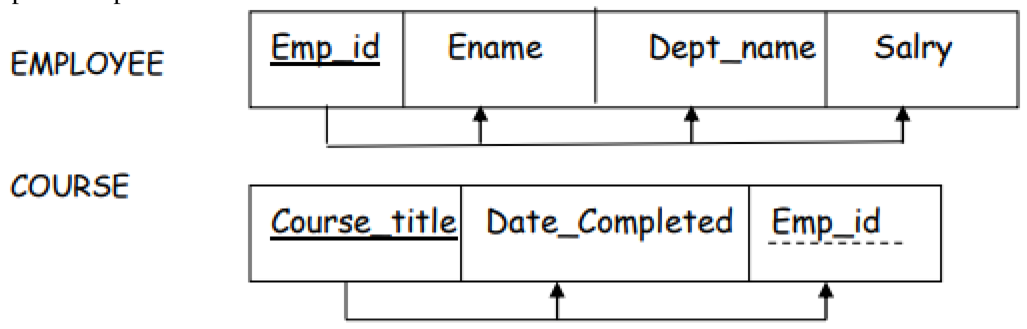


Emp-id	name	dept-name	salary	course_title
100	Krishna	cse	20000	vc++
100	Krishna	cse	20000	msoffice
140	Raja	it	18000	C++
140	Raja	it	18000	DBMS
140	Raja	it	18000	DS

SECOND NORMAL FORM (2NF):
A relation in Second Normal Form (2NF) if it is in the 1NF and if all non-key attributes are fully functionally dependent on the primary key. In a functional dependency X -> Y, the attribute on left hand side (i.e. x) is the primary key of the relation and right side attributes on right hand side i.e. Y is the non-key attributes. In some situation some non-key attributes are partial functional dependency on primary key. Consider the following example for partial functional specification and also that convert into 2 NF to decompose that into two relations.



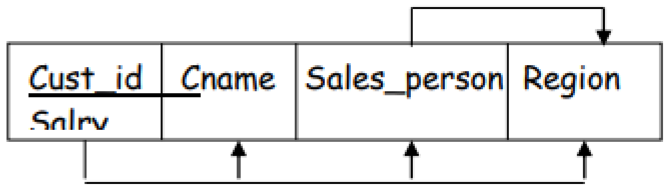
To avoid this, convert this into Second Normal Form. The 2NF will decompose the relation into two relations, shown in graphical representation



In the above graphical representation
¬ the EMPLOYEE relation satisfies rule of 1 NF in Second Normal form and
¬ the COURSE relation satisfies rule of 2 NF by decomposing into two relation

THIRD NORMAL FORM(3NF): A relation that is in Second Normal form and has no transitive dependencies present.

Transitive dependency: A transitive is a functional dependency between two non-key attributes. For example, consider the relation Sales with attributes cust_id, name, sales person and region that shown in graphical representation.



CUST_ID	NAME	SALESPERSON	REGION
1001	Anand	Smith	South
1002	Sunil	kiran	West
1003	Govind	babu rao	East
1004	Manohar	Smith	South
1005	Madhu	Somu	North

In this example, to insert, delete and update any row that facing Anomaly

a) Insertion Anomaly: A new salesperson is assigned to North Region without assign a customer to that salesperson. This causes insertion Anomaly.

b) Deletion Anomaly: If a customer number say 1003 is deleted from the table, we lose the information of salesperson who is assigned to that customer. This causes, Deletion Anomaly.

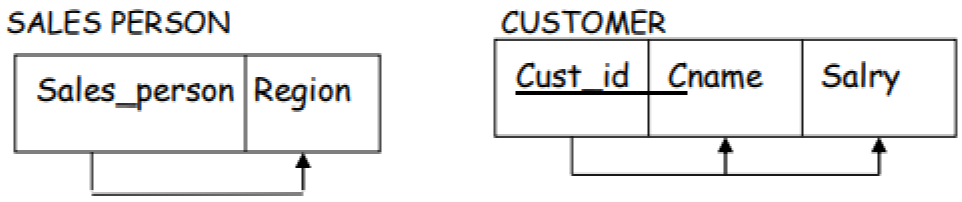
c) Modification Anomaly: If salesperson Smith is reassigned to the East region, several rows must be changed to reflect that fact. This causes, update anomaly.

To avoid this Anomaly problem, the transitive dependency can be removed by decomposition of SALES into two relations in 3NF.

Consider the following example, that removes Anomaly by decomposing into two relations

CUST_ID	NAME	SALESPERSON
1001	Anand	Smith
1002	Sunil	Kiran
1003	Govind	Babu rao
1004	Manohar	Smith
1005	Madhu	Somu

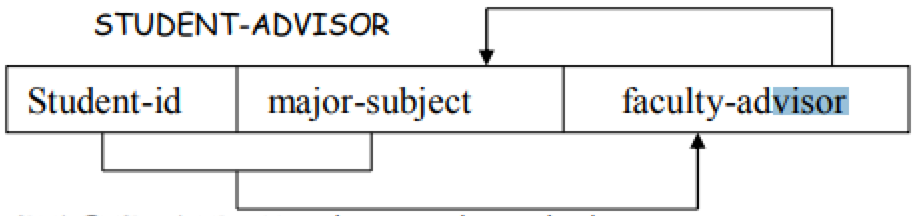
SalesPerson	Region
Smith	South
Kiran	West
Babu Rao	East
Smith	South
Somu	North



BOYCE/CODD NORMAL FORM(BCNF): A relation is in BCNF if it is in 3NF and every determinant is a candidate key.

FD in F+ of the form X -> A where X c S and A є S, X is a super key of R.

Boyce-Codd normal form removes the remaining anomalies in 3NF that are resulting from functional dependency, we can get the result of relation in BCNF.
For example, STUDENT-ADVIDSOR IN 3NF



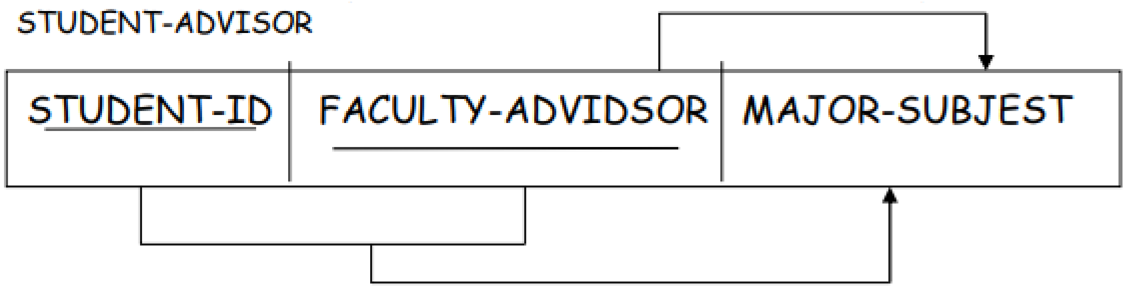
STUDENT-ADVISOR relstion with simple data.

STYDENT-ID	MAJOR-SUBJECT	FACULTY-ADVISOR
1	MATHS	B
2	MATHS	B
3	MATHS	B
4	STATISTICS	A
5	STATISTICS	A

In the above relation the primary key in student-id and major-subject. Here the part of the primary key major-subject is dependent upon a non-key attribute faculty–advisor. So, here the determinant the faculty-advisor. But it is not candidate key.

Here in this example there are no partial dependencies and transitive dependencies. There is only functional dependency between part of the primary key and non key attribute. Because of this dependency there is anomaly in this relation. Suppose that in maths subject the advisor' B' is replaced by X. this change must be made in two or more rows in this relation. This is an updation anomaly.

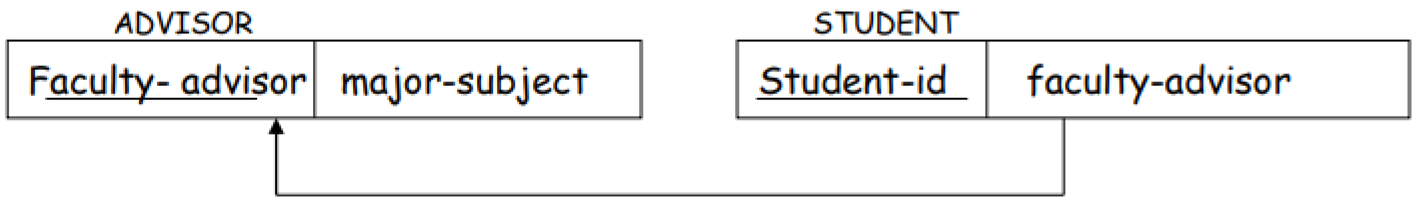
To convert a relation to BCNF the first step in the original relation is modified that the determinant (non key attributes) becomes a component of the primary key of new relation. The attribute that is dependent on determinant becomes a non-key attributes.



The second step in the conversion process is decompose the relation to eliminate the partial functional dependency.

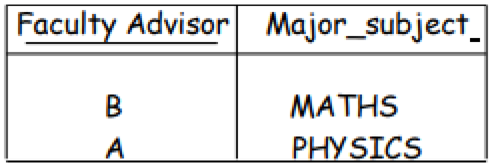
This results in two relations. These relations are in 3NF and BCNF. since there is only one candidate key. That is determinant.

Two relations are in BCNF.

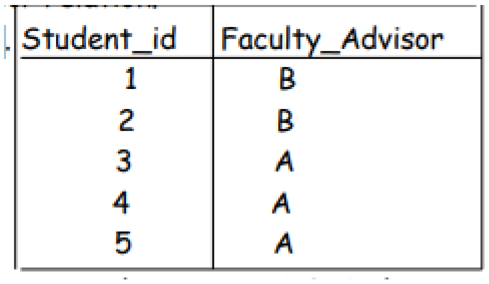


In these two relations the student relation has a composite key, which contains attributes student-id and faculty-advisor. Here faculty–advisor a foreign key which is referenced to the primary key of the advisor relation.

Two relations are in BCNF with simple data



Faculty Advisor	Major_subject_
B	MATHS
A	PHYSICS



Student_id	Faculty_Advisor
1	B
2	B
3	A
4	A
5	A

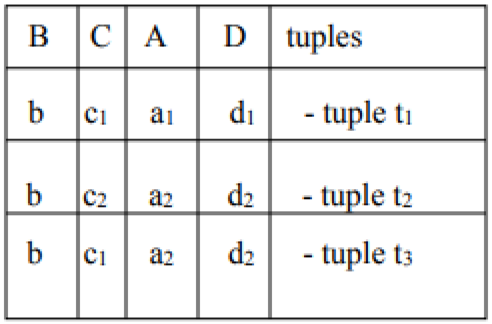
Fourth Normal Form (4 NF):

A relation is in BCNF that contain no multivalued dependency. In this case, 1 NF will repeated in this step. For example, R be a relation schema, X and Y be attributes of R, and F be a set of dependencies that includes both FDs and MVDs. (i.e. Functional Dependency and Multi-valued Dependencies). Then R is said to be in Fourth Normal Form (4NF) if for every MVD X ->-> Y that holds over R, one of the following statements is true.

1) Y c X or XY = R, or 2) X is a super key

Example: Consider a relation schema ABCD and suppose that are FD A -> BCD and the MVD B -> -> C are given as shown in Table

It shows three tuples from relation ABCD that satisfies the given MVD B -> -> C. From the definition of a MVD given tuples t1 and t2, it follows that tuples t3 must also be included in the above relation. Now, consider tuples t2 and t3. From the given FD A -> BCD and the fact that these tuples have the same A-value, we can compute



B	C	A	D	tuples
b	c_1	a_1	d_1	- tuple t_1
b	c_2	a_2	d_2	- tuple t_2
b	c_1	a_2	d_2	- tuple t_3

the c1 = c2. Therefore, we see that the FD B -> C must hold over ABCD whenever the FD A-> BCD and the MVD B-> -> C holds. If B -> C holds, the relation is not in BCNF but the relation is in 4 NF.

The fourth normal from is useful because it overcomes the problems of the various approaches in which it represents the multi-valued attributes in a single relatio00n.

Fifth Normal Form (5 NF): Any remaining anomalies from 4 NF relation have been removed.

A relation schema R is said to be in Fifth Normal Form (5NF) if, for every join dependency
* (R1, ... Rn) that holds over R, one of the following statements is true.

*Ri = R for some I, or

* The JD is implied by the set of those FDs over R in which the left side is a key for R. It deals with a property loss less joins

LOSSELESS-JOIN DECOMPOSITION:

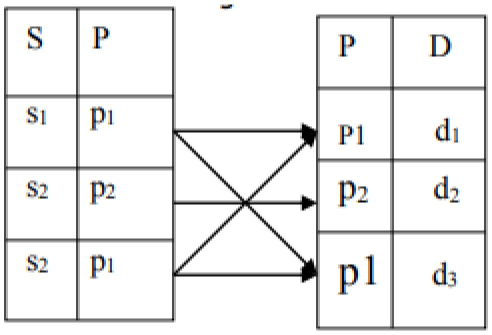
Let R be a relation schema and let F be a set FDs (Functional Dependencies) over R. A decomposition of R into two schemas with attribute X and Y is said to be lossless-join decomposition with respect to F, if for every instance r of R that satisfies the dependencies in Fr.

$$\pi_x (r) \bowtie \pi_y (r) = r$$

In simple words, we can recover the original relation from the decomposed relations

In general, if we take projection of a relation and recombine them using natural join, we obtain some additional tuples that were not in the original relation

S	P	D
s_1	p_1	d_1
s_2	p_2	d_2
s_2	p_1	d_3



S	P
s_1	p_1
s_2	p_2
s_2	p_1

P	D
P1	d_1
p_2	d_2
p1	d_3

S	P	D
s_1	p_1	d_1
s_1	p_1	d_3
s_2	p_2	d_2
s_2	p_1	d_1
s2	p_1	d_3

The decomposition of relation schema r i.e. SPD into SP i.e. PROJECTING πsp (r) and PD i.e., projecting πPD (r) is therefore lossless decomposition as it gains back all original tuples of relation ‘r’ as well as with some additional tuples that were not in original relation ‘r’