Databases 2022

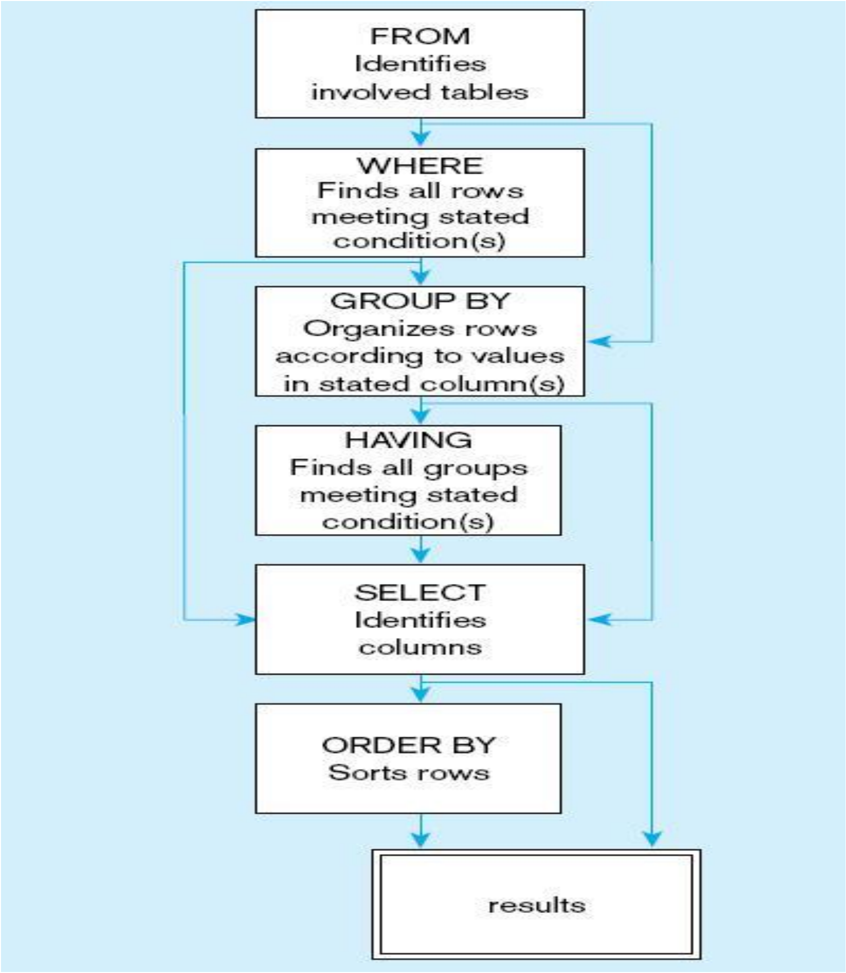
Darko Bozhinoski,
Ph.D. in Computer Science

Agenda

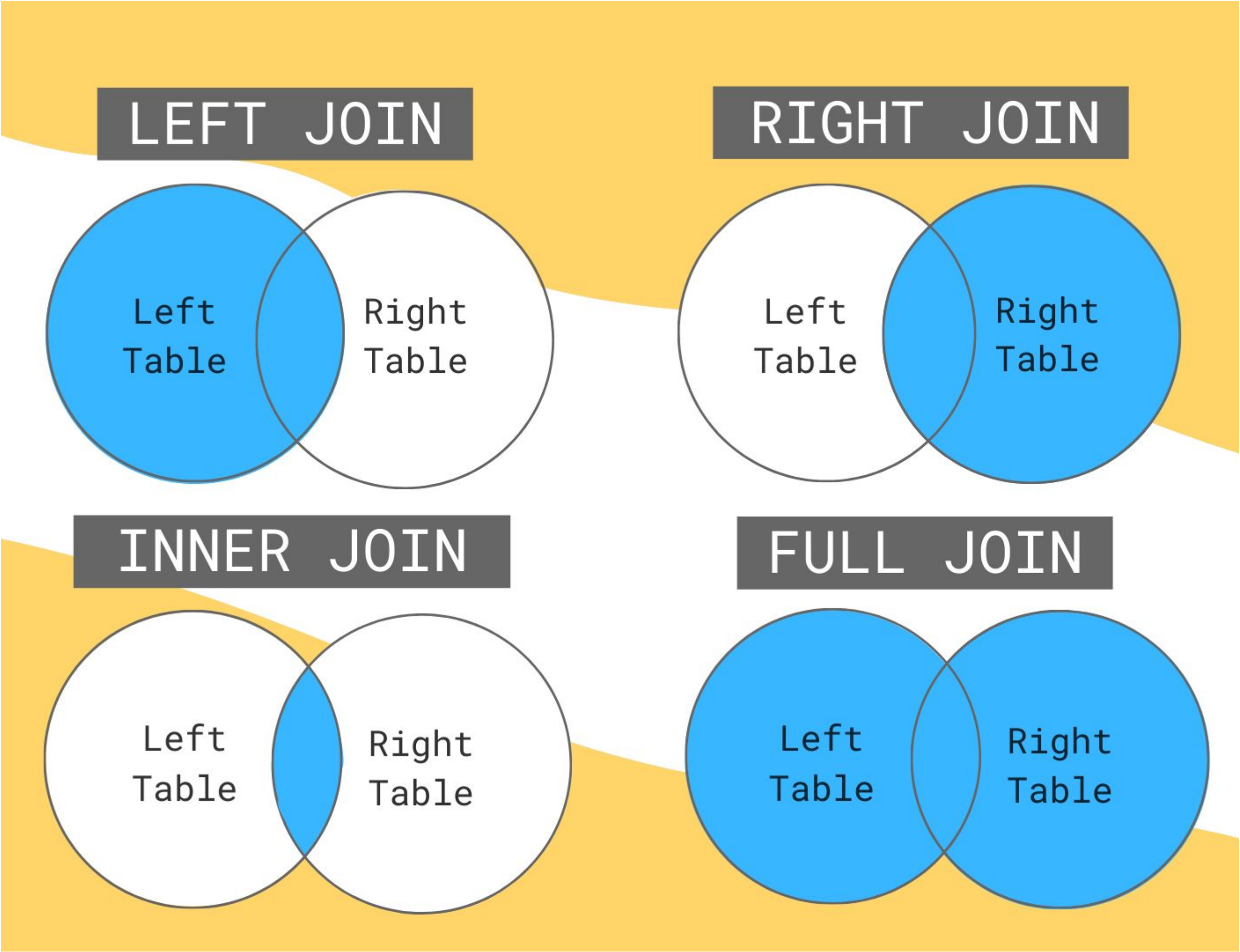
- SQL Advanced Concepts (Recap)
- Database Normalization

SELECT STATEMENT SUMMARY

- Clauses of the SELECT statement:
  - **SELECT**
    - List the columns (and expressions) to be returned from the query
  - **FROM**
    - Indicate the table(s) or view(s) from which data will be obtained
  - **WHERE**
    - Indicate the conditions under which a row will be included in the result
  - **GROUP BY**
    - Indicate categorization of results
  - **HAVING**
    - Indicate the conditions under which a category (group) will be included
  - **ORDER BY**
    - Sorts the result according to specified criteria


FROM
Identifies
involved tables
WHERE
Finds all rows
meeting stated
condition(s)
GROUP BY
Organizes rows
according to values
in stated column(s)
HAVING
Finds all groups
meeting stated
condition(s)
SELECT
Identifies
columns
ORDER BY
Sorts rows
results

JOINS

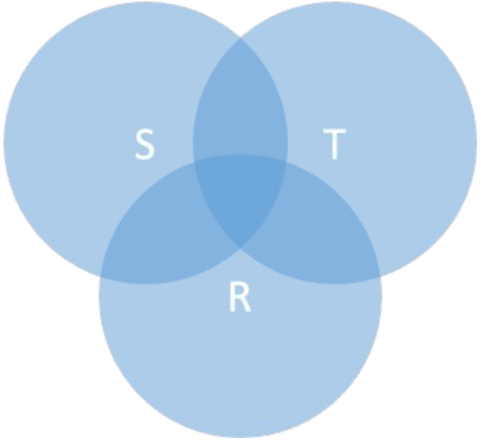

LEFT JOIN
Left Table
Right Table
RIGHT JOIN
Left Table
Right Table
INNER JOIN
Left Table
Right Table
FULL JOIN
Left Table
Right Table

```
SELECT  DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A    OR   R.A=T.A
```

What does it compute ?

```
SELECT  DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A    OR   R.A=T.A
```

What does it compute ?



Computes R ∩ (S ∪ T)

Subqueries Returning Relations

Company(<u>name</u>, city)
Product(<u>pname</u>, maker)
Purchase(<u>id</u>, product, buyer)

Return cities where one can find companies that manufacture products bought by Ivan Ivanov

```
SELECT  Company.city
FROM    Company
WHERE   Company.name  IN
            (SELECT Product.maker
             FROM   Purchase , Product
             WHERE Product.pname=Purchase.product
                AND Purchase .buyer = ‘Ivan Ivanov‘);
```

Subqueries Returning Relations

```
SELECT  Company.city
FROM    Company, Product, Purchase
WHERE   Company.name= Product.maker
        AND  Product.pname  = Purchase.product
        AND  Purchase.buyer = 'Ivan Ivanov'
```

Subqueries Returning Relations

```
SELECT  Company.city
FROM    Company, Product, Purchase
WHERE   Company.name= Product.maker
        AND  Product.pname  = Purchase.product
        AND  Purchase.buyer = ‘Ivan Ivanov’
```

Beware of duplicates !

Removing Duplicates

```
SELECT DISTINCT Company.city
FROM     Company
WHERE  Company.name  IN
                (SELECT Product.maker
                  FROM   Purchase , Product
                  WHERE Product.pname=Purchase.product
                      AND Purchase .buyer = ‘Joe Blow‘);
```

```
SELECT DISTINCT Company.city
FROM      Company, Product, Purchase
WHERE   Company.name= Product.maker
        AND  Product.pname  = Purchase.product
        AND  Purchase.buyer = ‘Joe Blow’
```

Subqueries Returning Relations

You can also use: **s > ALL R**

s > ANY R

EXISTS R

Product (pname, price, category, maker)

Find products that are more expensive than all those produced By "Gizmo-Works"

```
SELECT  name
FROM    Product
WHERE   price >  ALL (SELECT price
                      FROM    Purchase
                      WHERE   maker='Gizmo-Works')
```

Question

- Can we express this query as a single SELECT-FROM-WHERE query, without subqueries ?

- Select - From - Where queries are monotone.
  - A monotonic query is one that does not lose any tuples it previously made output, with the addition of new tuples in the database.
- A query with **ALL** is not monotone

Correlated Queries

Movie (title, year, director, length)

Find movies whose title appears more than once.

correlation

```
SELECT DISTINCT title
FROM   Movie AS x
WHERE  year <> ANY
                (SELECT  year
                 FROM    Movie
                 WHERE   title =  x.title);
```

Note (1) scope of variables (2) this can still be expressed as single Select From WHERE query

Complex Correlated Query

Product (pname, price, category, maker, year)

- Find products (and their manufacturers) that are more expensive than all products made by the same manufacturer before 1972

Complex Correlated Query

Product (pname, price, category, maker, year)

- Find products (and their manufacturers) that are more expensive than all products made by the same manufacturer before 1972

```
SELECT DISTINCT  pname, maker
FROM     Product AS x
WHERE  price > ALL  (SELECT  price
                                  FROM    Product AS y
                                  WHERE  x.maker = y.maker AND y.year < 1972);
```

Aggregation Operations

Aggregation

```
SELECT  avg(price)
FROM    Product
WHERE   maker="Toyota"
```

```
SELECT  count(*)
FROM    Product
WHERE   year > 1995
```

SQL supports several aggregation operations:

sum, count, min, max, avg

Except count, all aggregations apply to a single attribute

Aggregation: Count

COUNT applies to duplicates, unless otherwise stated:

```
SELECT  Count(category)
FROM    Product
WHERE   year > 1995
```

same as Count(*)

We probably want:

```
SELECT  Count(DISTINCT category)
FROM    Product
WHERE   year > 1995
```

Examples

Purchase(product, date, price, quantity)

Product	Date	Price	Quantity
Bagel	10/21	1	20
Banana	10/3	0.5	10
Banana	10/10	1	10
Bagel	10/25	1.50	20

```
SELECT  Sum(price * quantity)
FROM    Purchase
```

```
SELECT  Sum(price * quantity)
FROM    Purchase
WHERE   product = 'bagel'
```

Grouping and Aggregation

Purchase(product, date, price, quantity)

Find total sales after 10/1/2005 per product.

```
SELECT      product, Sum(price*quantity) AS TotalSales
FROM        Purchase
WHERE       date > '10/1/2005'
GROUP BY    product
```

What this means?

Grouping and Aggregation

1. Compute the FROM and WHERE clauses.

2. Group by the attributes in the GROUPBY

3. Compute the SELECT clause: grouped attributes and aggregates.

1&2. FROM-WHERE-GROUPBY

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

3. SELECT

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

Product	TotalSales
Bagel	50
Banana	15

```
SELECT      product, Sum(price*quantity) AS TotalSales
FROM        Purchase
WHERE       date > '10/1/2005'
GROUP BY    product
```

HAVING Clause

Consider products that had at least 100 buyers.

```
SELECT      product, Sum(price *
quantity)
FROM        Purchase
WHERE       date > '10/1/2005'
GROUP BY    product
HAVING      Sum(quantity) > 30
```

HAVING clause contains conditions on aggregates.

General form of Grouping and Aggregation

SELECT S
FROM $R_1,\ldots,R_n$
WHERE C1
GROUP BY $a_1,\ldots,a_k$
HAVING C2

Evaluation steps:

1. Evaluate FROM-WHERE, apply condition C1
2. Group by the attributes $a_1,\ldots,a_k$
3. Apply condition C2 to each group
4. Compute aggregates in S and return the result

Advanced Concepts

1. Quantifiers

2. Aggregation v.s. subqueries

Quantifiers

Product (pname, price, company)
Company(cname, city)

Find all companies that make <u>some</u> products with price < 100

```
SELECT DISTINCT  Company.cname
FROM     Company, Product
WHERE  Company.cname = Product.company and Product.price < 100
```

Quantifiers Example

1. Find *the other* companies: i.e. s.t. <u>some</u> product $\geq$ 100

2. Find all companies s.t. <u>all</u> their products have price < 100

Quantifiers Example Solution

1. Find *the other* companies: i.e. s.t. <u>some</u> product ≥ 100

```
SELECT DISTINCT  Company.cname
FROM     Company
WHERE  Company.cname IN (SELECT Product.company
                          FROM Product
                          WHERE Product.price >= 100
```

2. Find all companies s.t. <u>all</u> their products have price < 100

```
SELECT DISTINCT  Company.cname
FROM     Company
WHERE  Company.cname NOT IN (SELECT Product.company
                              FROM Product
                              WHERE Produc.price >= 100
```

GROUP BY v.s. Nested Query

Author(<u>login</u>,name)

Wrote(login,url)

- Find authors who wrote ≥ 10 documents:
- Attempt 1: with nested queries

```
SELECT DISTINCT Author.name
FROM        Author
WHERE       count(SELECT Wrote.url
                  FROM Wrote
                  WHERE Author.login=Wrote.login) > 10
```

Group-by v.s. Nested Query

- Find all authors who wrote at least 10 documents:
- Attempt 2: SQL style (with GROUP BY)

```
SELECT      Author.name
FROM        Author, Wrote
WHERE       Author.login=Wrote.login
GROUP BY    Author.name
HAVING      count(wrote.url) > 10
```

No need for DISTINCT: automatically from GROUP BY

Group-by v.s. Nested Query (Example)

Author(<u>login</u>,name)

Wrote(login,url)

Mentions(url,word)

Find authors with vocabulary ≥ 10000 words:

Group-by v.s. Nested Query (Example)

Author(login,name)

Wrote(login,url)

Mentions(url,word)

Find authors with vocabulary ≥ 10000 words:

```
SELECT      Author.name
FROM        Author, Wrote, Mentions
WHERE       Author.login=Wrote.login AND Wrote.url=Mentions.url
GROUP BY    Author.name
HAVING      count(distinct Mentions.word) > 10000
```

Example

Store(sid, sname)
Product(pid, pname, price, sid)

Find all stores that sell *only* products with price > 100

same as:

Find all stores s.t. all their products have price > 100)

```
SELECT Store.name
FROM    Store, Product
WHERE  Store.sid = Product.sid
GROUP BY  Store.sid, Store.name
HAVING 100 < min(Product.price)
```

```
SELECT Store.name
FROM    Store
WHERE
   100 < ALL (SELECT
Product.price
                    FROM product
                    WHERE Store.sid =
Product.sid)
```

```
SELECT Store.name
FROM    Store
WHERE  Store.sid NOT IN
               (SELECT Product.sid
                FROM Product
                WHERE  Product.price <= 100)
```

NULLS in SQL

- Whenever we don’t have a value, we can put a NULL
- Can mean many things:
  - Value does not exists
  - Value exists but is unknown
  - Value not applicable
  - Etc.
- The schema specifies for each attribute if can be null (*nullable* attribute) or not

Null Values

Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

```
SELECT *
FROM    Person
WHERE   age < 25  OR  age >= 25 OR age IS NULL
```

Normalization

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
- What are the criteria for "good" base relations?

Informal and Formal Design Guidelines for Relational Databases

- We first discuss informal guidelines for good relational design
- Then we discuss formal concepts of functional dependencies and normal forms
  - *1NF (First Normal Form)*
  - *2NF (Second Normal Form)*
  - *3NF (Third Normal Form)*
  - *BCNF (Boyce-Codd Normal Form)*

Clear Semantics of the Relation Attributes

- **GUIDELINE 1: Informally, each tuple in a relation should represent one entity or relationship instance.**
  - 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.
- <u>Bottom Line:</u> *Design a schema that can be explained easily relation by relation. The semantics of attributes should be easy to interpret.*

Reduce Redundant Information in Tuples

- Redundant information
  - Wastes storage
  - Causes problems with update anomalies
    - Insertion anomalies
    - Deletion anomalies
    - Modification anomalies

EXAMPLE OF AN UPDATE ANOMALY

- Consider the relation:
  - EMP_PROJ(Emp#, Proj#, Ename, Pname, No_hours)
- Update Anomaly:
  - If we change the name of project number P1 from “Billing” to “Customer-Accounting”, we must update all employees that work on project P1.

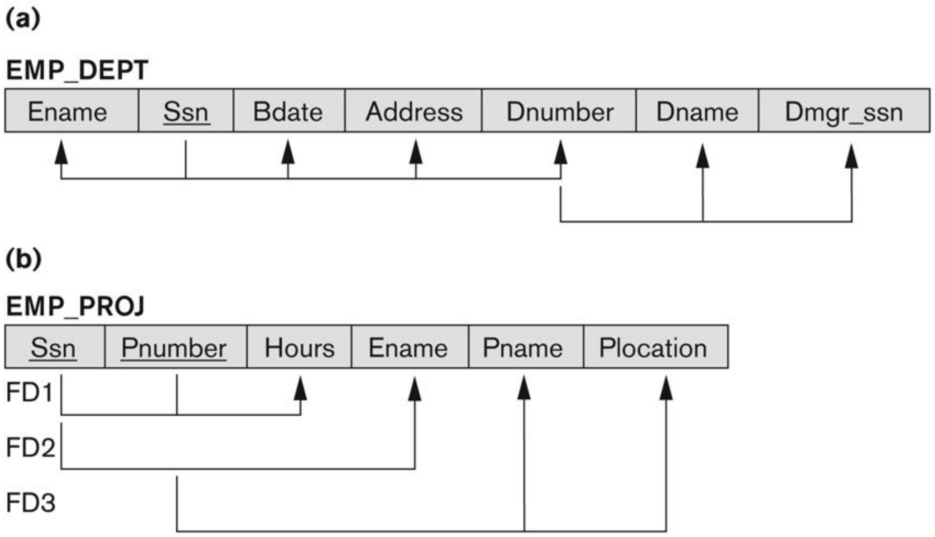
EXAMPLE OF AN INSERT ANOMALY

- Consider the relation:
  - EMP_PROJ(Emp#, Proj#, Ename, Pname, No_hours)
- Insert Anomaly:
  - Cannot insert a project unless an employee is assigned to it.
- Conversely
  - Cannot insert an employee unless an he/she is assigned to a project.

EXAMPLE OF AN DELETE ANOMALY

- Consider the relation:
  - EMP_PROJ(Emp#, Proj#, Ename, Pname, No_hours)
- Delete Anomaly:
  - When a project is deleted, it will result in deleting all the employees who work on that project.
  - Alternately, if an employee is the sole employee on a project, deleting that employee would result in deleting the corresponding project.

Figure 10.3
Two relation schemas suffering from update anomalies.
(a) EMP_DEPT and
(b) EMP_PROJ.



Two relation schemas suffering from update anomalies

GUIDELINE 2

- **GUIDELINE:**
  - 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.

GUIDELINE 3

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

- **GUIDELINE:**
  - 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)

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:**
  - 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.**

Functional Dependencies (1)

- 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**

Functional Dependencies (2)

- X -> Y holds if whenever two tuples have the same value for X, they *must have* the same value for Y
  - For any two tuples t1 and t2 in any relation instance r(R): If t1[X]=t2[X], *then* t1[Y]=t2[Y]
- X -> Y in R specifies a *constraint* on all relation instances r(R)
- FDs are derived from the real-world constraints on the attributes

Examples of FD constraints (1)

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

Examples of FD constraints (2)

- 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])

Normalization of Relations (1)

- **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 (2)

- 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, BCNF or 4NF)
- **Denormalization**:
  - The process of storing the join of higher normal form relations as a base relation—which is in a lower normal form

Reading Material

- Fundamentals of Database Systems. Ramez Elmasri and Shamkant B. Navathe. Pearson. **Chapter 7 and Chapter 14.**


Q & A