DBMS

Module 05

Content

- Data Modeling
- Normalization, and Star Schema
- ACID transactions
- Select, insert, update & delete (DML and DQL)
- Join operations
- Window functions (rank, dense rank, row number etc)
- Data Types, Variables and Constants
- Conditional Structures (IF, CASE, GOTO and NULL)
- Integrating python with SQL

Database Systems and the life cycle

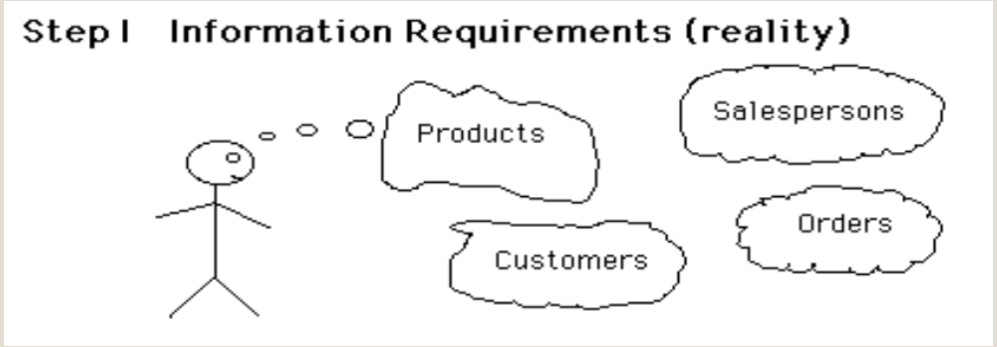
- Introductory Concepts
  - data — a fact, something upon which an inference is based.
  - **data item** — smallest named unit of data that has meaning in the real world (examples: lastname, address, ssn)
  - **data aggregate (or group)** -- a collection of related data items that form a whole concept;
  - **record** — group of related data items treated as a unit by an application program (examples: students, presidents, elections)
  - **file**—collection of records of a single type (examples: student, president, election)
  - **database**—computerized collection of interrelated stored data that serves the needs of multiple users within one or more organizations, i.e. interrelated collections of records of potentially many types.
  - **database management system (DBMS**) -- a generalized software system for manipulating databases. Includes logical view (schema, sub-schema), physical view(access methods, clustering), data manipulation language, data definition language, utilities- security, recovery, integrity, etc.
  - **database administrator (DBA)** -- person or group responsible for the effective use of database technology in an organization or enterprise. he has control over all phases of the lifecycle.

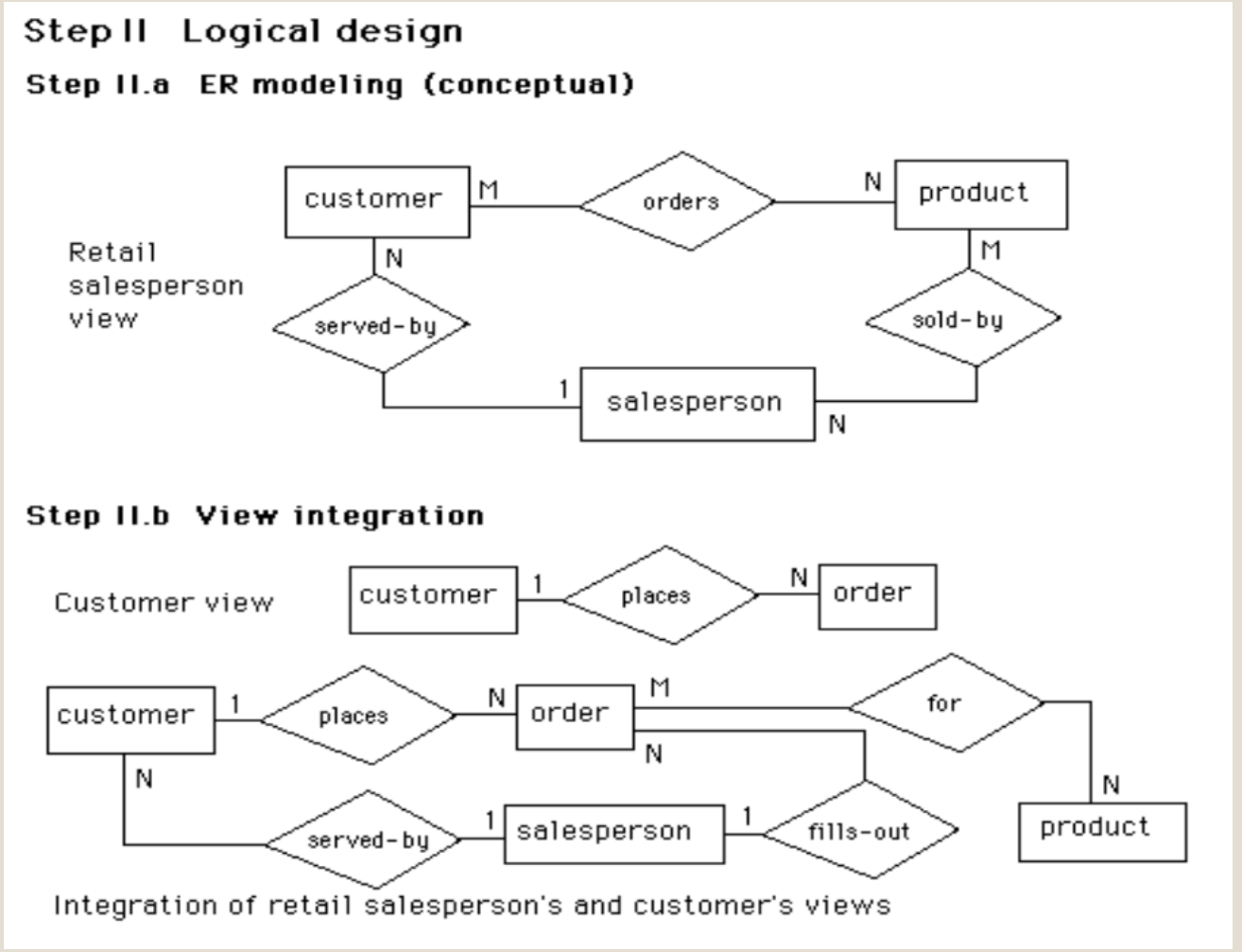
Database Systems and the life cycle
Objectives of DBMS

1. Data Availability - make an integrated collection of data avaialble to a wide variety of users.
    a) at reasonable cost
    b) in meaningful format
    c) easy access
2. Data Integrity - ensure correctness and validity
3. Privacy and security - schema/sub-schema, passwords
4. Management Control - DBA, lifecycle control, training, maintenance.
5. Data independence - avoids reprogramming of applications, allows easier conversion and reorganization.
    a) Physical data independence - program unaffected by changes in the storage structure or access methods
    b) logical data independence - program unaffected by changes in the schema.

Social Security Administration example (1980ís)
- changed benefit checks from $999.99 to $9999.99 format
- had to change 600 application programs
- 20,000 work hours needed to make the changes (10 work years)

Database Life Cycle





Step II.c Transformation of the ER diagram to SQL tables

Customer

cust- no	cust- name	.....

Product

prod-no	prod-name	qty-in-stock

```
create table customer
   (cust_no integer,
   cust_name char(15),
   cust_addr char(30),
   sales_name chor(15),
   prod_no integer,
   primary key (cust_no),
   foreign key (sales_name)
      references salesperson,
   foreign key (prod_no)
      references product);
```

Salesperson

sales-name	addr	dept	job-level	vacation-days

Order

order-no	sales-name	cust-no

Order-product

order-no

Step II.d Normalization of SQL tables (3NF, BCNF, 4NF, 5NF)

Decomposition of tables and removal of update anomalies.

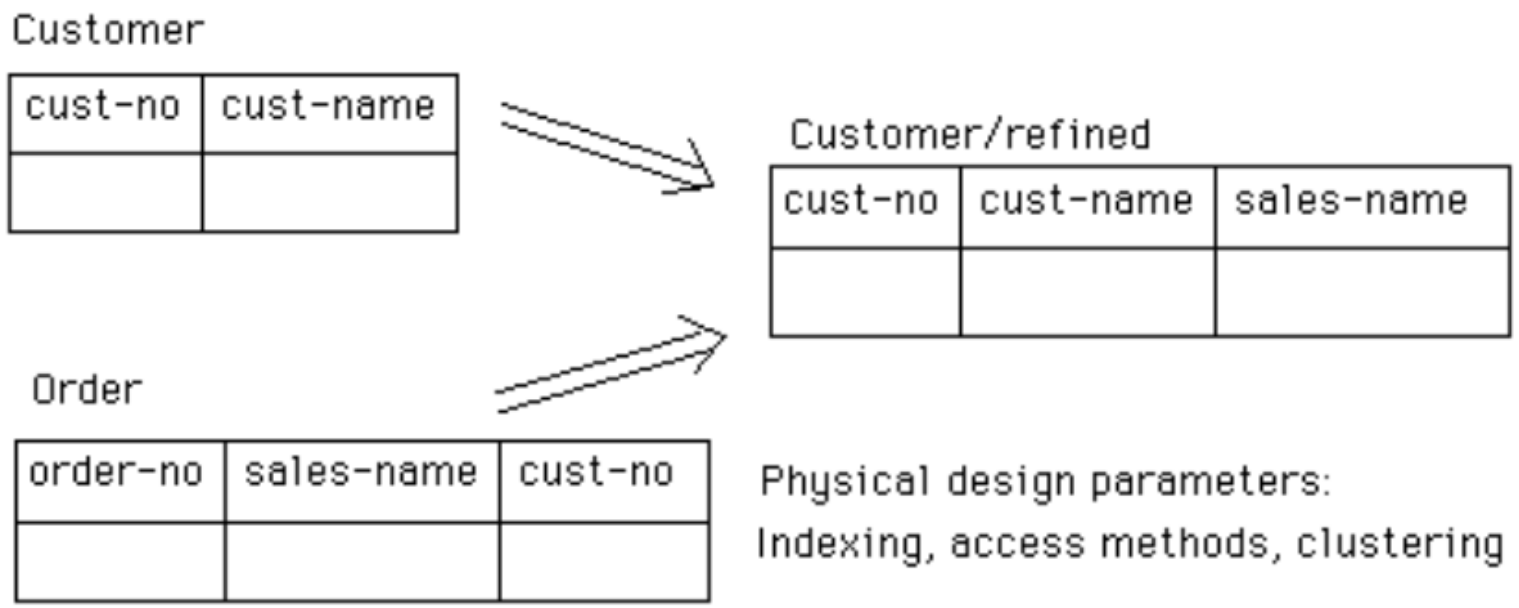
Salesperson

sales-name	addr	dept	job-level

Sales-vacations

job-level	vacation-days

Step III Physical Design (including denormalization)



What is Data Modelling

- Data modeling is the process of creating a data model for the data to be stored in a Database.
- This data model is a conceptual representation of Data objects, the associations between different data objects and the rules.
- Data modeling helps in the visual representation of data and enforces business rules on the data.
- Data model emphasizes on what data is needed and how it should be organized instead of what operations need to be performed on the data.
- Data Model is like architect's building plan which helps to build a conceptual model and set the relationship between data items.

Why to use Data Modelling?

- Ensures that all data objects required by the database are accurately represented. Omission of data will lead to creation of faulty reports and produce incorrect results.
- A data model helps design the database at the conceptual, physical and logical levels.
- Data Model structure helps to define the relational tables, primary and foreign keys and stored procedures.
- It provides a clear picture of the database and can be used by database developers to create a physical database.
- It is also helpful to identify missing and redundant data.
- Though the initial creation of data model is labor and time consuming, in the long run, it makes your IT infrastructure upgrade and maintenance cheaper and faster.

Types of Data Models

1. Conceptual:
   - This Data Model defines WHAT the system contains.
   - This model is typically created by Business stakeholders and Data Architects.
   - The purpose is to organize, scope and define business concepts and rules.
2. Logical:
   - Defines HOW the system should be implemented regardless of the DBMS.
   - This model is typically created by DataArchitects and Business Analysts.
   - The purpose is to develop technical map of rules and data structures.
3. Physical:
   - This Data Model describes HOW the system will be implemented using a specific DBMS system.
   - This model is typically created by DBA and developers.
   - The purpose is actual implementation of the database.

Conceptual Model

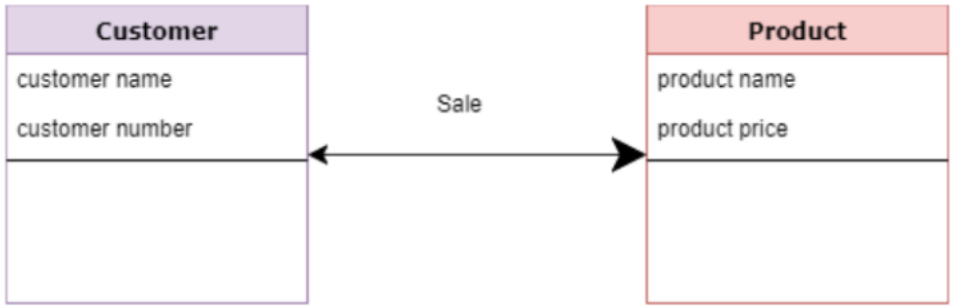
The main aim of this model is to establish the entities, their attributes, and their relationships. In this Data modeling level, there is hardly any detail available of the actual Database structure.

The 3 basic tenants of Data Model are

Entity: A real-world thing

Attribute: Characteristics or properties of an entity

Relationship: Dependency or association between two entities



For example:

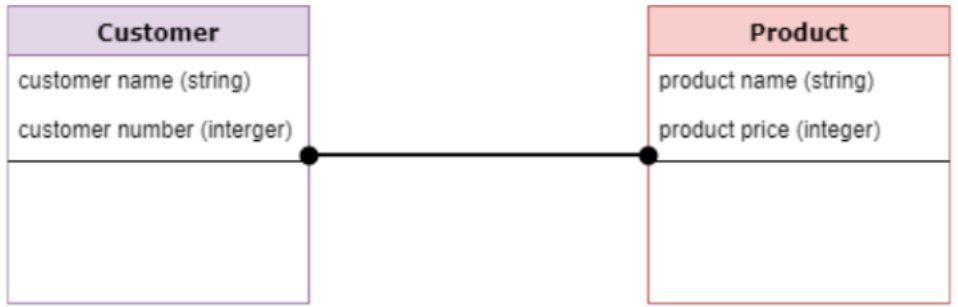
- Customer and Product are two entities. Customer number and name are attributes of the Customer entity
- Product name and price are attributes of product entity
- Sale is the relationship between the customer and product

Logical Data Model

Logical data models add further information to the conceptual model elements. It defines the structure of the data elements and set the relationships between them.

The advantage of the Logical data model is to provide a foundation to form the base for the Physical model. However, the modeling structure remains generic.

At this Data Modeling level, no primary or secondary key is defined. At this Data modeling level, you need to verify and adjust the connector details that were set earlier for relationships.



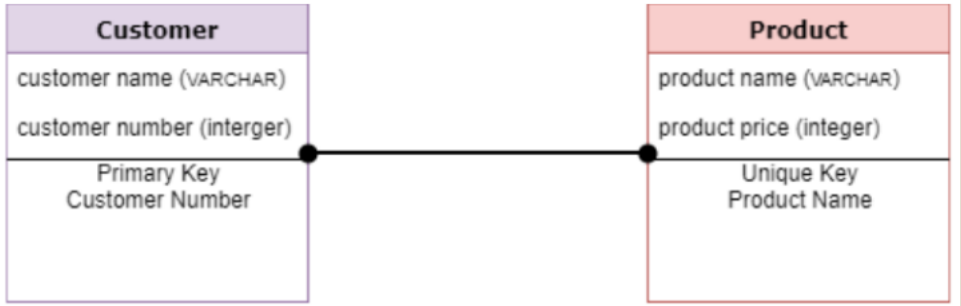
Characteristics of a Logical data model

- Describes data needs for a single project but could integrate with other logical data models based on the scope of the project.
- Designed and developed independently from the DBMS.
- Data attributes will have datatypes with exact precisions and length.
- Normalization processes to the model is applied typically till 3NF.

Physical Data Model

A Physical Data Model describes the database specific implementation of the data model. It offers an abstraction of the database and helps generate schema. This is because of the richness of meta-data offered by a Physical Data Model.

This type of Data model also helps to visualize database structure. It helps to model database columns keys, constraints, indexes, triggers, and other RDBMS features.



Characteristics of a physical data model:

- The physical data model describes data need for a single project or application though it maybe integrated with other physical data models based on project scope.

- Data Model contains relationships between tables that which addresses cardinality and nullability of the relationships.
- Developed for a specific version of a DBMS, location, data storage or technology to be used in the project.
- Columns should have exact datatypes, lengths assigned and default values.
- Primary and Foreign keys, views, indexes, access profiles, and authorizations, etc. are defined.

Introduction to Schema Refinement

- Schema Refinement is intended to address "Redundancy" Problem in database design and the refinement approach is based on decomposition.
- Redundancy : Duplicate copies of same data stored in different location.
- Redundant storage of information is the root cause of several problems in database design.
- Problems caused by redundancy
  - Redundant storage
  - Update / Insertion / Deletion Anomalies

Introduction to Schema Refinement

Sid	SName	Cld	CName	Fld	FName	Fee
S1	A	C1	C	F1	X	5K
S2	B	C1	C	F1	X	5K
S3	B	C2	C++	F2	Y	10K
S4	C	C1	C	F1	X	5K

Consider the relation

What is the key?

Hourly_emps(ssn, name, lot, rating, hourly_wages, hours_worked)

ssn	name	lot	Rating	Hourly_wages	Hours_worked
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

Is there any redundancy?

In the above table redundancy is there because of the constraing *rating* → *hourlywage*

What are the problems we face here?

What is the solution?

Introduction to Schema Refinement

- Decomposition
  - Technique used to eliminate the problems caused by redundancy.
  - Def: 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 of R.
  - in simple words, Splitting the relation into two or more sub tables
- Problems related to Decomposition
  - Decomposing a relationshema can create more problems than it solves
  - Two important questions to be addressed while decomposing are
    1. Do we need to decompose a relation?
    2. What problems(if any) does a given decomposition cause?

Sid	SName	Cld	CName	Fld	FName	Fee
S1	A	C1	C	F1	X	5K
S2	B	C1	C	F1	X	5K
S3	B	C2	C++	F2	Y	10K
S4	C	C1	C	F1	X	5K

Sid	S name	Cid
S1	A	C1
S2	B	C1
S3	B	C2
S4	C	C1

Fid	F name	Cid	C name	Fee
F1	X	C1	C	5K
F2	Y	C2	C++	10K

Anomalies are removed.

Decomposition must satisfy some properties

Suggest a decomposition for Hourly_emps(ssn, name, lot, rating, hourly_wages, hours_worked

It can be decomposed into two relations
Hourly_Emps2(ssn, name lot, rating, hours_worked)
Wages(rating, hourly_wages)

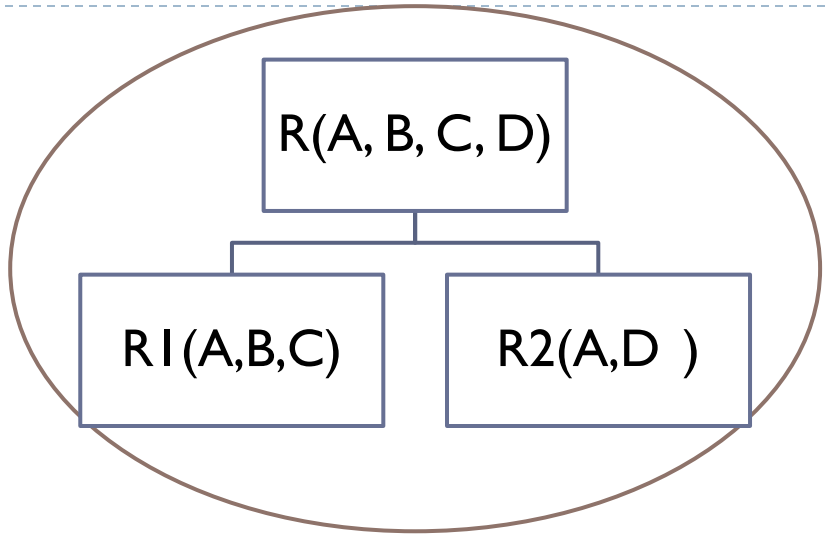
ssn	name	lot	Rating	Hours_worked
123-22-3666	Attishoo	48	8	40
231-31-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

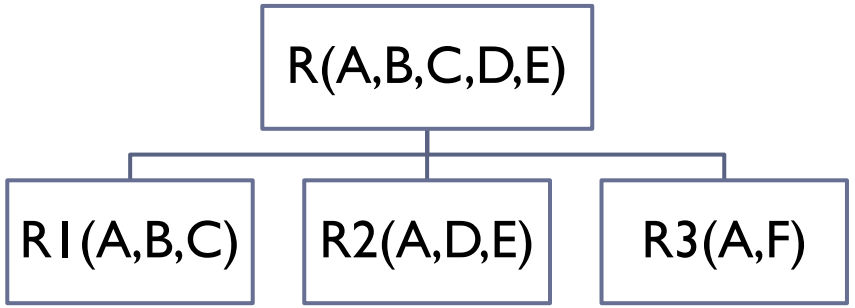
Rating	Hourly_wages
8	10
5	7

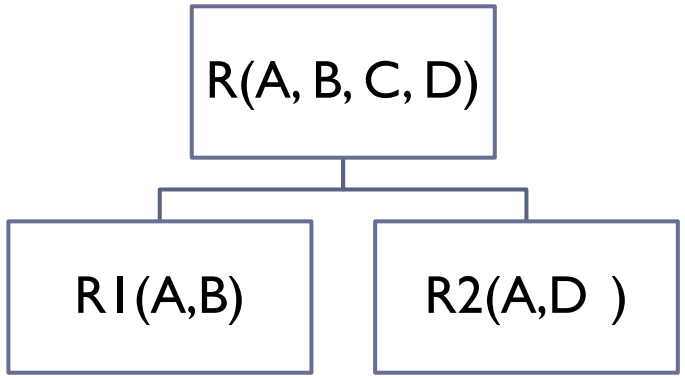
Do we need to decompose a relation?

- Normal forms have been proposed for the relations.
- NF of a given schema help us to decide whether or not to decompose it further.

Is it a valid decomposition?







What is normalization?

Normalization is a technique of organizing the data into multiple related tables, to minimize DATA REDUNDANCY.

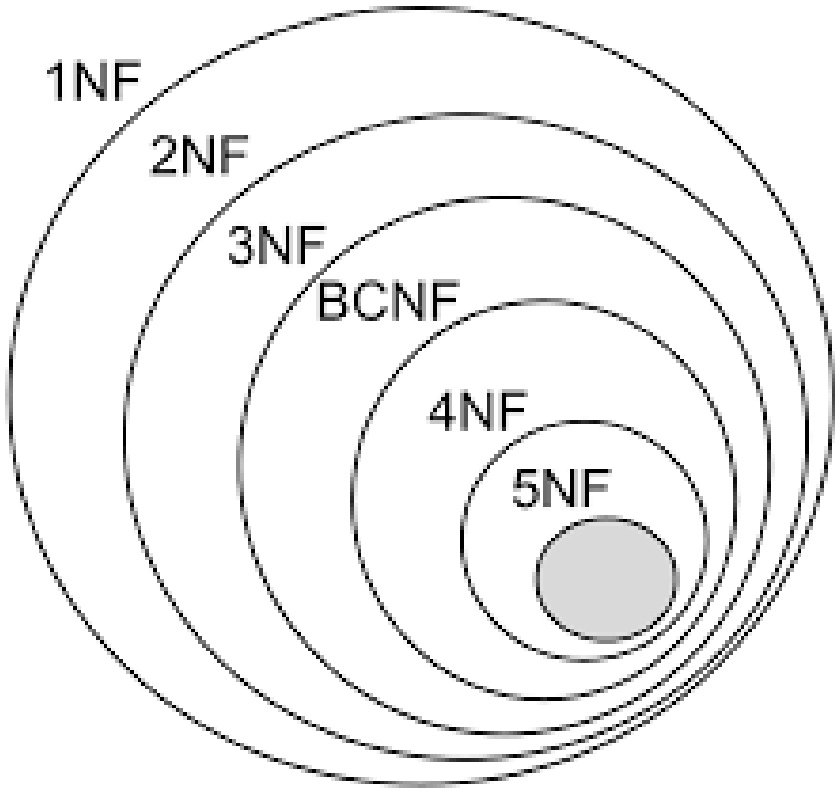
Normal Forms

- Provides a guidance to decide whether a database design is good or needs decomposition.
- Can be used to identify the presence of redundancy in relations.
- Normal forms based on FDs are
  - First normal form (1 NF)
  - Second normal form (2 NF)
  - Third normal form (3 NF)
  - Boyce-Codd normal form (BCNF)
- A relational table is said to be in a particular normal form if it satisfied a certain set of constraints.

Normal Forms

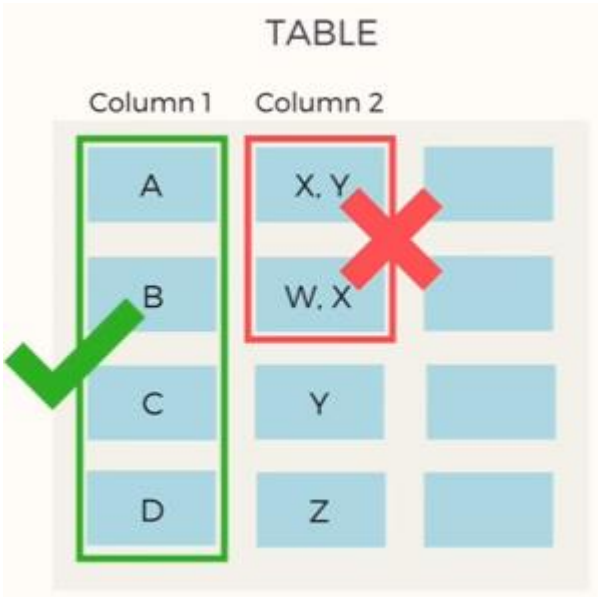
Each higher level is a subset of the lower level

- If a relation is in one the of the higher normal forms, then it will be automatically in all lower normal forms
- The higher the normal form the lower the redundancy
- Therefore higher normal forms are preferable



First Normal Form (1 NF)

- A schema is in First normal form
  - if the domain of every attribute is atomic
    - That is, no composite values (lists or sets)
  - All entries in any column must be of the same kind.
  - Each column must have a unique name
  - No two rows are identical



Time for an example

STUDENTS TABLE

rollno	name	subject
101	Akon	OS, CN
103	Ckon	JAVA
102	Bkon	C, C++

Non-atomic

rollno	name	subject
101	Akon	OS
101	Akon	CN
103	Ckon	JAVA
102	Bkon	C
102	Bkon	C++

Partial Dependency

SCORE TABLE

student_id	subject_id	marks	teacher
10	1	82	Mr. J
10	2	77	Mr. C++
11	1	85	Mr. J
11	2	82	Mr. C++
11	4	95	Mr. P

What is the primary key?

student_id + subject_id

Student_id, subject_id → marks

Is teacher name also depend on both student_id and subject_id?

Teacher column only depends on subject and not on student

This is Partial Dependency

Full Functional Dependency

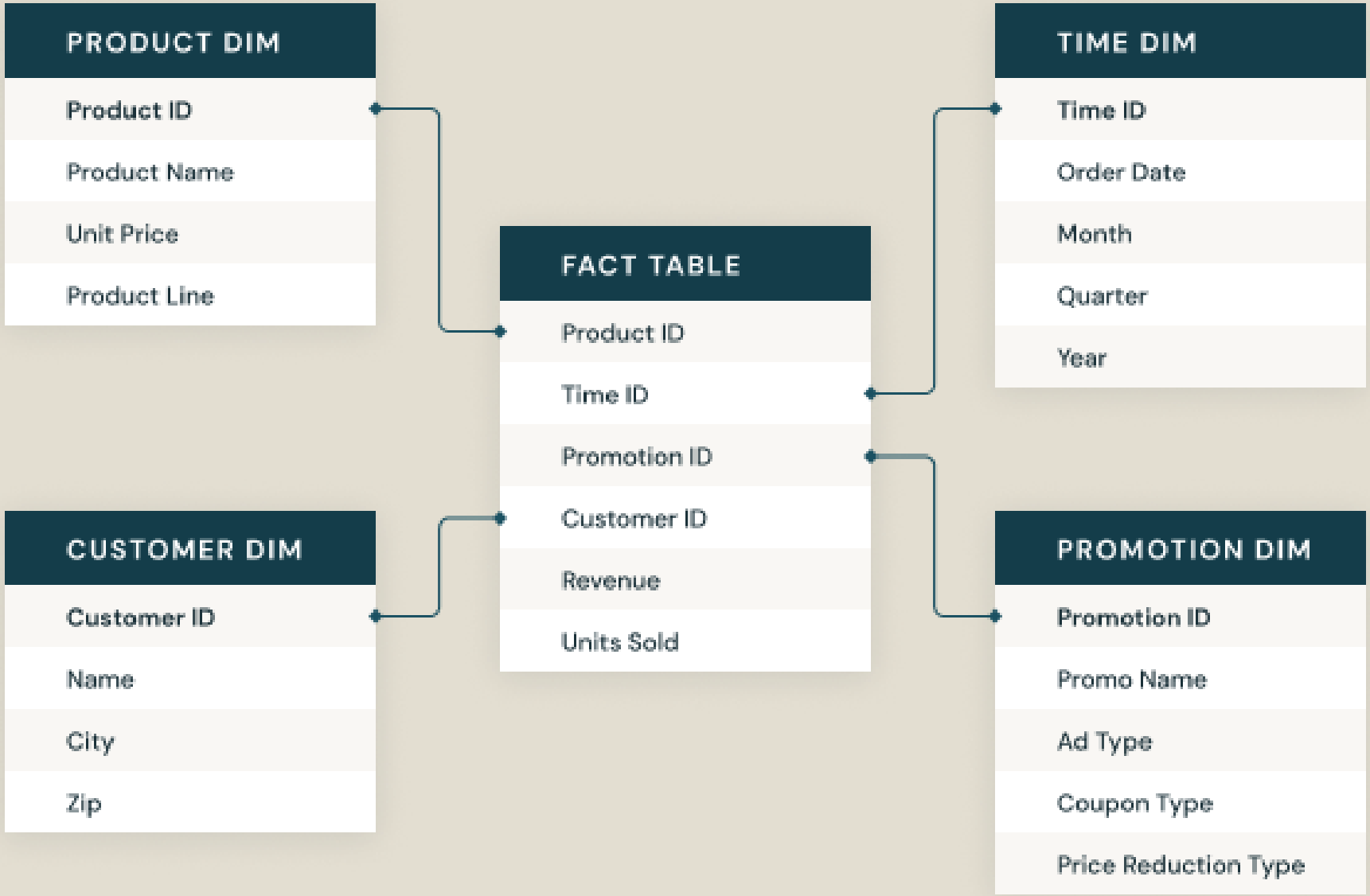
- In X → Y,

  if Y cannot be determined by any of subsets of X, then Y is said to be fully functionally dependent on X

Second Normal Form

- A relation is said to be in 2 NF
  - If it is in 1 NF and
  - All non prime attributes are fully functionally dependent on any key of R (it should not have any partial dependencies)

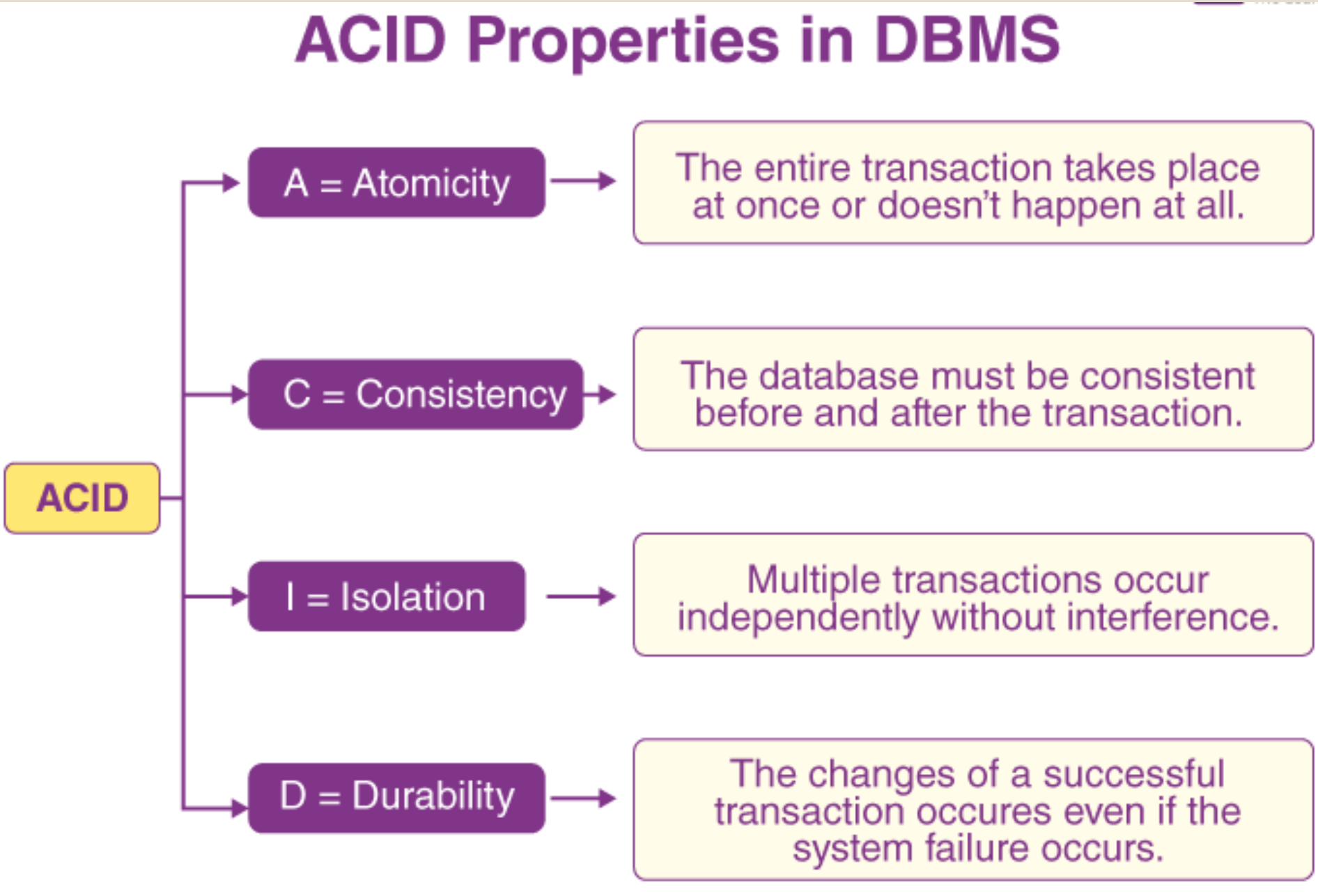
Star schema

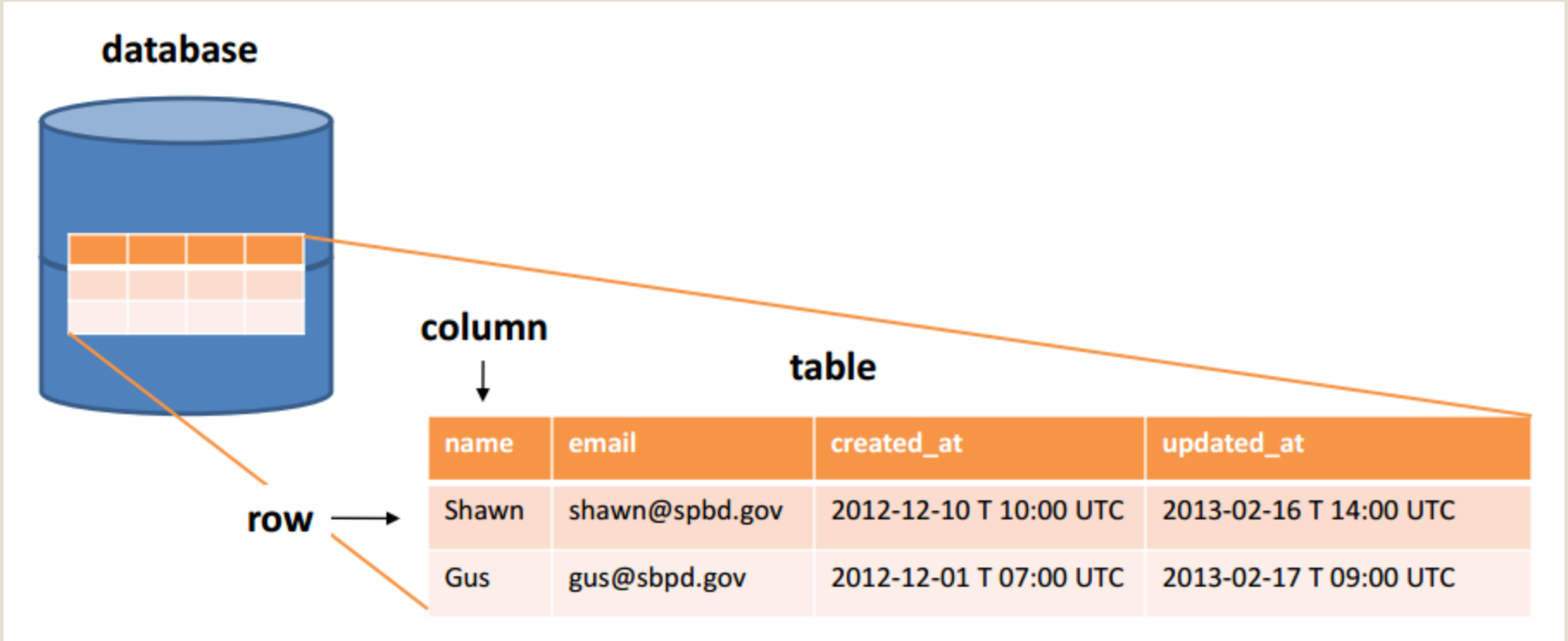

PRODUCT DIM
Product ID
Product Name
Unit Price
Product Line
TIME DIM
Time ID
Order Date
Month
Quarter
Year
FACT TABLE
Product ID
Time ID
Promotion ID
Customer ID
Revenue
Units Sold
CUSTOMER DIM
Customer ID
Name
City
Zip
PROMOTION DIM
Promotion ID
Promo Name
Ad Type
Coupon Type
Price Reduction Type

Star Schema

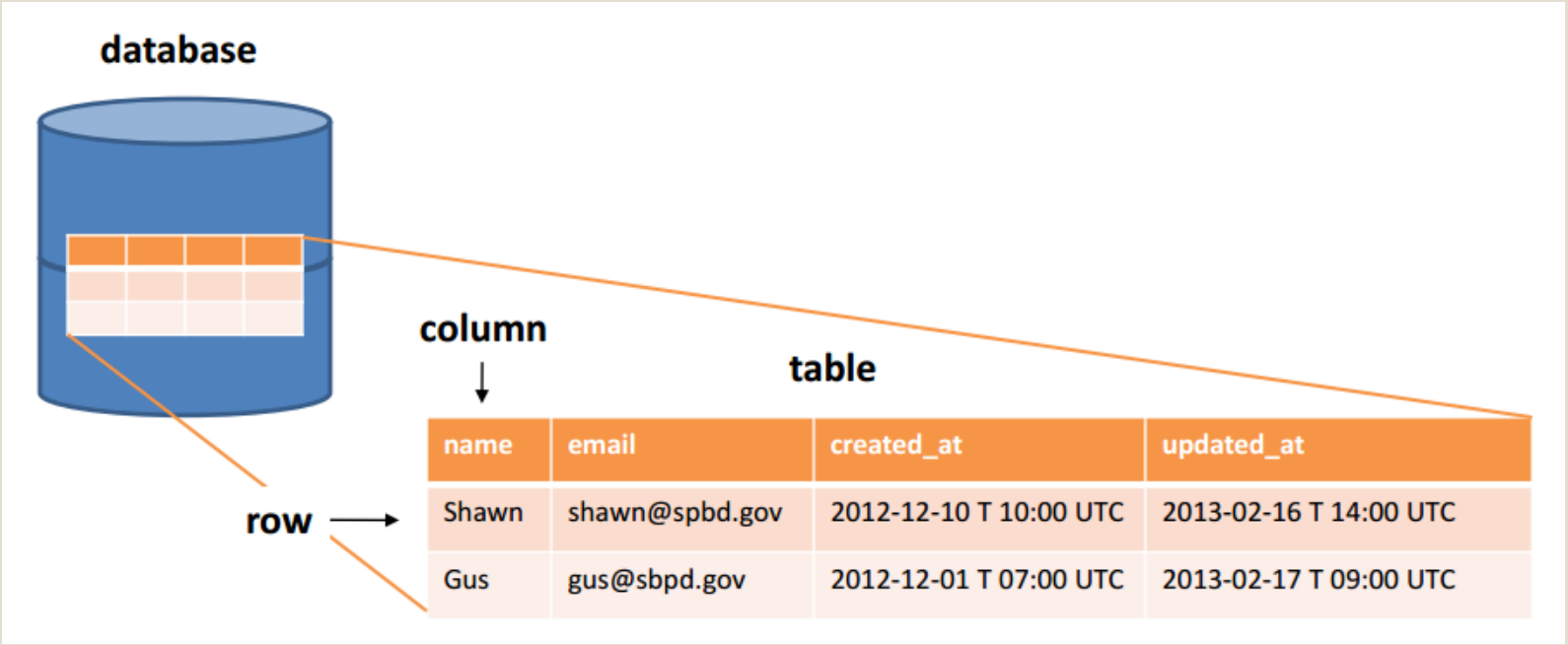
- A star-schema is a multi-dimensional data model used to organize data in a database so that it is easy to understand and analyze.
- The star schema design is efficient at storing data and optimized for querying large datasets.
- It denormalizes the data into dimenstions and facts.
- It has a single fact table in the center, containing facts. The fact table connects to multiple other dimension tables.
- Star schemas denormalize the data, which means adding redundant columns to some dimension tables to make querying and working with the data faster and easier.

ACID Properties in DBMS



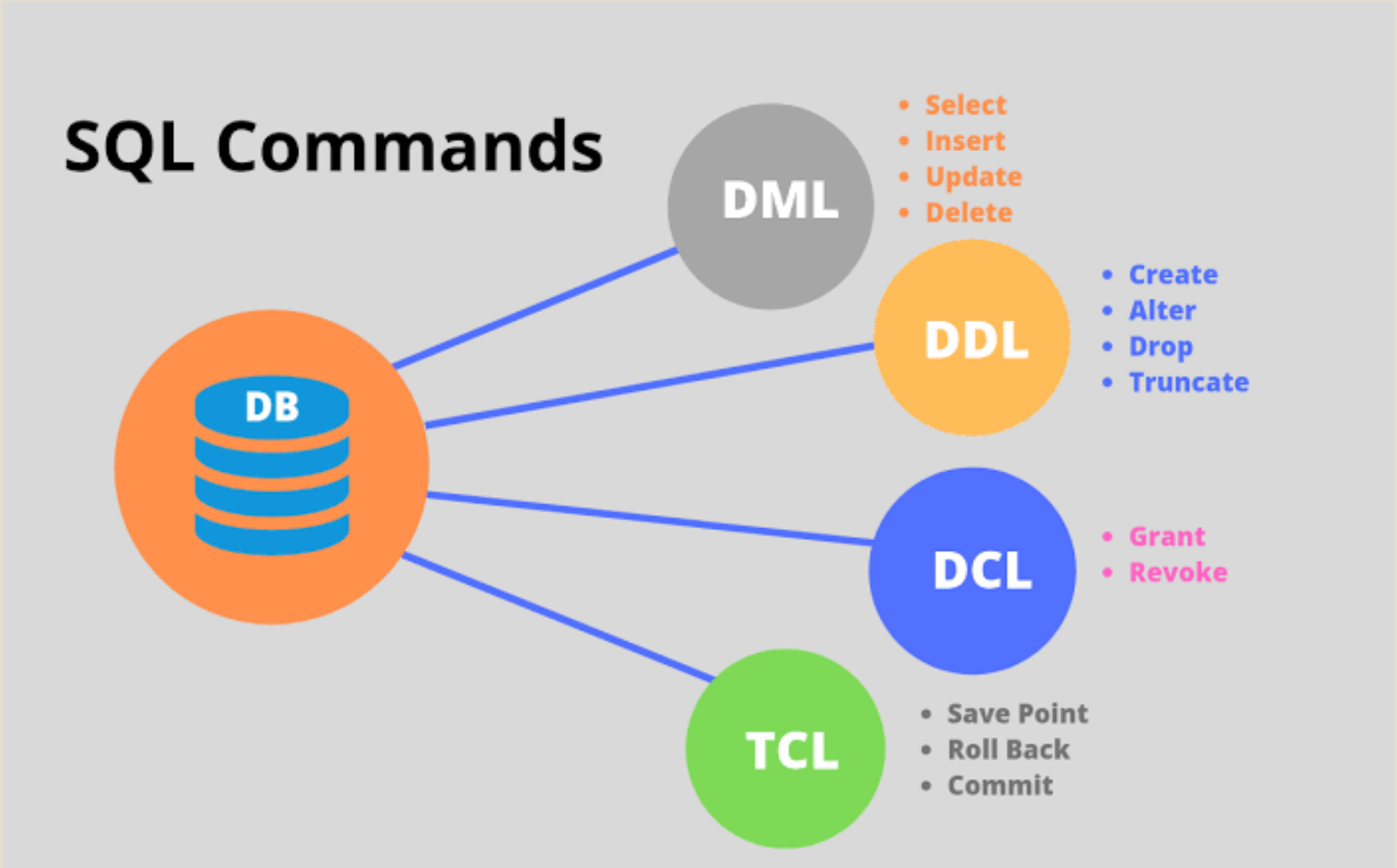

database
column
table
row
name
email
created_at
updated_at
Shawn
shawn@spbd.gov
2012-12-10 T 10:00 UTC
2013-02-16 T 14:00 UTC
Gus
gus@sbpd.gov
2012-12-01 T 07:00 UTC
2013-02-17 T 09:00 UTC

- **RDBMS –**
  - Relational Database Management System used to maintain a relational database.
  - It is the basis for all modern database systems such as MySQL, Microsoft SQL Server, Oracle, and Microsoft Access.
  - RDBMS uses SQL queries to access the data in the database.
- **Table**
  - A collection of related data entries, and it consists of columns and rows.
  - A column holds specific information about every record in the table. A row is a record is an individual entry that exists in a table.



Structured Query Language (SQL)

- SQL is a standard language for accessing and manipulating databases.
- SQL
  - stands for Structured Query Language
  - lets you access and manipulate databases
  - can execute queries against a database
  - can retrieve data from a database
  - can insert / update records in a database
  - can delete records from a database
  - can create new databases and tables in a database
  - can create stored procedures and views in a database
  - can set permissions on tables, procedures, and views


SQL Commands
DB
DML
Select
Insert
Update
Delete
DDL
Create
Alter
Drop
Truncate
DCL
Grant
Revoke
TCL
Save Point
Roll Back
Commit

Querying

- The SELECT statement allows you to select data from one or more tables. To write a SELECT statement in MySQL, you use this syntax:

  SELECT select_list

  FROM table_name;

- How to
  - Retrieve a single column
  - retrive multiple columns
  - retrive all columns
- MySQL has many built-in functions like string, date, and Math functions. And you can use the SELECT statement to execute these functions.
  - SELECT NOW();
  - SELECT CONCAT('John',' ','Doe');
- To change a column name of the result set, you can use a column alias:
  - SELECT expression AS column_alias;
  - SELECT CONCAT('John',' ','Doe') AS name;

Querying contd…

- **Order by** - to sort the rows in the result set
- syntax of the order by clause:

```
SELECT    select_list
FROM    table_name
ORDER BY    column1 [ASC | DESC],    column2 [ASC | DESC],    ...;
```

- The **WHERE clause** allows you to specify a search condition for the rows returned by a query. The following shows the syntax of the WHERE clause:

```
SELECT    select_list
FROM    table_name
WHERE    search_condition;
```

- AND and OR operator can be used to combine two conditions
- The BETWEEN operator returns TRUE if a value is in a range of values:
  - expression BETWEEN low AND high
  - SELECT * from customer where address_id between 30 and 40
- The LIKE operator evaluates to TRUE if a value matches a specified pattern.
  - To form a pattern, you use the % and _ wildcards.
- The IN operator returns TRUE if a value matches any value in a list.
- LiMIT – to limit the number of rows
- DISTINCT – to get the distinct values

Managing Databases

- Managing the databases:
- Select database(); - to display the current database
- use database_name; - to select the database to work with
- show databases; - to list the available databases
- Create database database_name; - creates a database
- Drop database database_name; - drops all tables in the database and deletes the database permanently.
- create table - allows to create a new table in a database
- Describe table – to display the structure of the table
- Drop table – is used to drop a table.

Creating tables

- syntax :

```
CREATE TABLE table_name(
   column_1_definition,
   column_2_definition,
   ...,
   table_constraints
)
```

- The syntax of column definition:

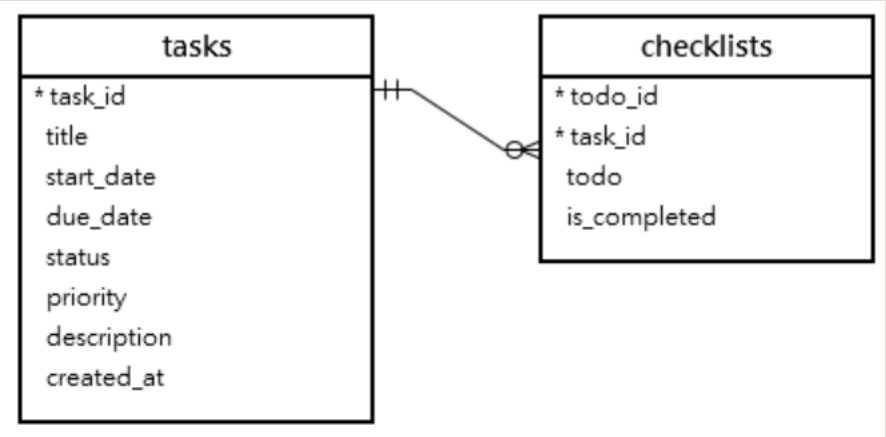
```
column_name data_type(length) [NOT NULL]
[DEFAULT value] [AUTO_INCREMENT]
column_constraint;
```

- Example:

```
CREATE TABLE  tasks (
    task_id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    start_date DATE,
    due_date DATE,
    status TINYINT NOT NULL,
    priority TINYINT NOT NULL,
    description TEXT,
    created_at TIMESTAMP DEFAULT
CURRENT_TIMESTAMP
);
```

Create table with primary and foreign key

```
CREATE TABLE IF NOT EXISTS checklists (
    todo_id INT AUTO_INCREMENT,
    task_id INT,
    todo VARCHAR(255) NOT NULL,
    is_completed BOOLEAN NOT NULL DEFAULT FALSE,
    PRIMARY KEY (todo_id , task_id),
    FOREIGN KEY (task_id)
        REFERENCES tasks (task_id)
        ON UPDATE RESTRICT ON DELETE CASCADE
);
```



MySQL DATA TYPES

DATE TYPE	SPEC	DATA TYPE	SPEC
CHAR	String (0 - 255)	INT	Integer (-2147483648 to 214748-3647)
VARCHAR	String (0 - 255)	BIGINT	Integer (-92233720368547758 08 to 9223372036854775807)
TINYTEXT	String (0 - 255)	FLOAT	Decimal (precise to 23 digits)
TEXT	String (0 - 65535)	DOUBLE	Decimal (24 to 53 digits)
BLOB	String (0 - 65535)	DECIMAL	"DOUBLE" stored as string
MEDIUMTEXT	String (0 - 16777215)	DATE	YYYY-MM-DD
MEDIUMBLOB	String (0 - 16777215)	DATETIME	YYYY-MM-DD HH:MM:SS
LONGTEXT	String (0 - 4294967295)	TIMESTAMP	YYYYMMDDHHMMSS
LONGBLOB	String (0 - 4294967295)	TIME	HH:MM:SS
TINYINT	Integer (-128 to 127)	ENUM	One of preset options
SMALLINT	Integer (-32768 to 32767)	SET	Selection of preset options
MEDIUMINT	Integer (-8388608 to 8388607)	BOOLEAN	TINYINT(1)

INSERT INTO Syntax

It is possible to write the `INSERT INTO` statement in two ways:

1. Specify both the column names and the values to be inserted:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

2. If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. Here, the `INSERT INTO` syntax would be as follows:

```
INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

Joins

- A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

OrderID	CustomerID	OrderDate
10308	2	1996-09-18
10309	37	1996-09-19
10310	77	1996-09-20

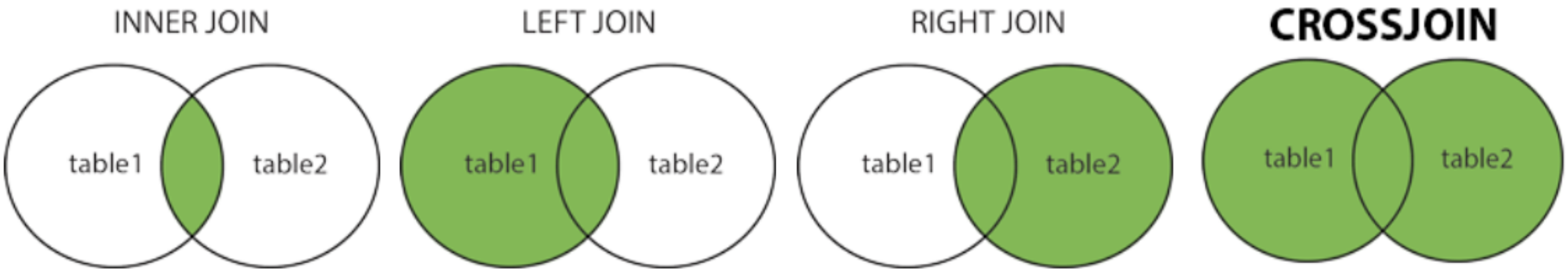
CustomerID	CustomerName	ContactName	Country
1	Alfreds Futterkiste	Maria Anders	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mexico

```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;
```

OrderID	CustomerName	OrderDate
10308	Ana Trujillo Emparedados y helados	9/18/1996
10365	Antonio Moreno Taquería	11/27/1996
10383	Around the Horn	12/16/1996
10355	Around the Horn	11/15/1996
10278	Berglunds snabbköp	8/12/1996

Supported Types of Joins in MySQL

- `INNER JOIN` : Returns records that have matching values in both tables
- `LEFT JOIN` : Returns all records from the left table, and the matched records from the right table
- `RIGHT JOIN` : Returns all records from the right table, and the matched records from the left table
- `CROSS JOIN` : Returns all records from both tables



Set the sample tables

- Create two tables with the name members and committees and insert some data as shown below.

```sql
CREATE TABLE members (
    member_id INT AUTO_INCREMENT,
    name VARCHAR(100),
    PRIMARY KEY (member_id)
);

CREATE TABLE committees (
    committee_id INT AUTO_INCREMENT,
    name VARCHAR(100),
    PRIMARY KEY (committee_id)
);
```

```sql
INSERT INTO members(name)
VALUES('John'),('Jane'),('Mary'),('David'),('Amelia');

INSERT INTO committees(name)
VALUES('John'),('Mary'),('Amelia'),('Joe');
```

```sql
SELECT * FROM members;
```

```
+-----------+--------+
| member_id | name   |
+-----------+--------+
|         1 | John   |
|         2 | Jane   |
|         3 | Mary   |
|         4 | David  |
|         5 | Amelia |
+-----------+--------+
```

```sql
SELECT * FROM committees;
```

```
+--------------+--------+
| committee_id | name   |
+--------------+--------+
|            1 | John   |
|            2 | Mary   |
|            3 | Amelia |
|            4 | Joe    |
+--------------+--------+
4 rows in set (0.00 sec)
```

Inner Join

- Syntax

```
SELECT column_list
FROM table_1
INNER JOIN table_2 ON join_condition;
```

- example:

```
SELECT
    m.member_id,
    m.name AS member,
    c.committee_id,
    c.name AS committee
FROM
    members m
INNER JOIN committees c ON c.name = m.name;
```

member_id	member	committee_id	committee
1	John	1	John
3	Mary	2	Mary
5	Amelia	3	Amelia

Left join

- syntax:

```
SELECT column_list
FROM table_1
LEFT JOIN table_2 ON join_condition;
```

(or)

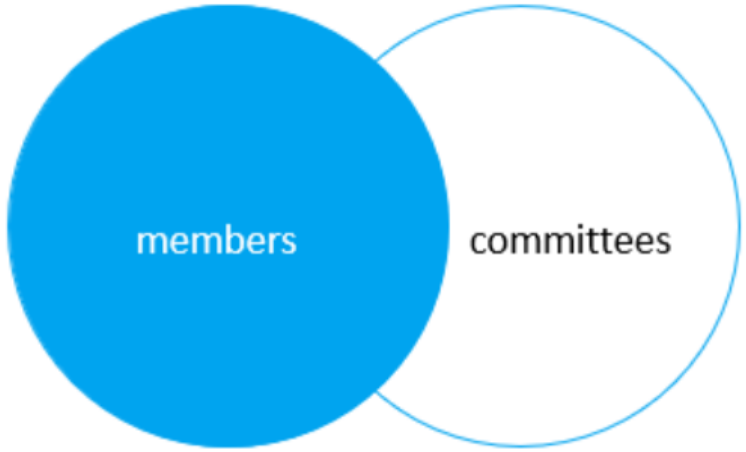
```
SELECT column_list
FROM table_1
LEFT JOIN table_2 USING (column_name);
```

- example:

```
SELECT
    m.member_id,
    m.name AS member,
    c.committee_id,
    c.name AS committee
FROM
    members m
LEFT JOIN committees c USING(name);
```

```
+-----------+--------+--------------+-----------+
| member_id | member | committee_id | committee |
+-----------+--------+--------------+-----------+
|         1 | John   |            1 | John      |
|         2 | Jane   |         NULL | NULL      |
|         3 | Mary   |            2 | Mary      |
|         4 | David  |         NULL | NULL      |
|         5 | Amelia |            3 | Amelia    |
+-----------+--------+--------------+-----------+
5 rows in set (0.00 sec)
```

The following Venn diagram illustrates the left join:



Right Join

- Syntax

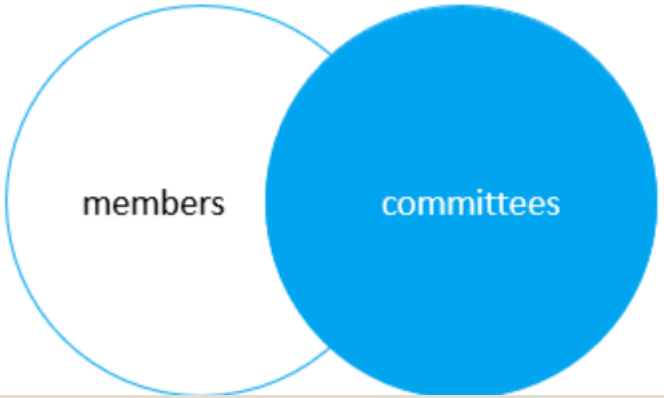
```
SELECT column_list
FROM table_1
RIGHT JOIN table_2 ON join_condition;
```

- Example:

```
SELECT
    m.member_id,
    m.name AS member,
    c.committee_id,
    c.name AS committee
FROM
    members m
RIGHT JOIN committees c on c.name = m.name;
```

```
+-----------+--------+--------------+-----------+
| member_id | member | committee_id | committee |
+-----------+--------+--------------+-----------+
|         1 | John   |            1 | John      |
|         3 | Mary   |            2 | Mary      |
|         5 | Amelia |            3 | Amelia    |
|      NULL | NULL   |            4 | Joe       |
+-----------+--------+--------------+-----------+
4 rows in set (0.00 sec)
```

This Venn diagram illustrates the right join:



Cross join

- does not have a join condition
- The cross join makes a Cartesian product of rows from the joined tables.
- Syntax:

```
SELECT select_list
FROM table_1
CROSS JOIN table_2;
```

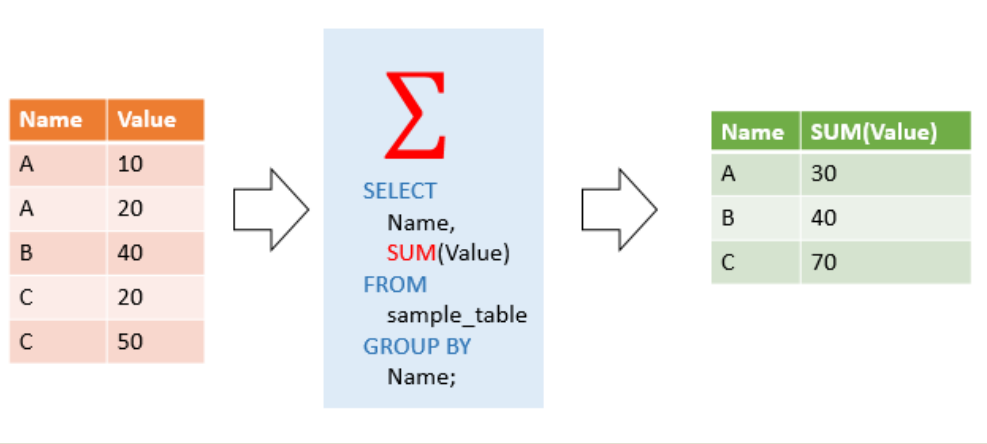
- Example:

```
SELECT
    m.member_id,
    m.name AS member,
    c.committee_id,
    c.name AS committee
FROM
    members m
CROSS JOIN committees c;
```

member_id	member	committee_id	committee
1	John	4	Joe
1	John	3	Amelia
1	John	2	Mary
1	John	1	John
2	Jane	4	Joe
2	Jane	3	Amelia
2	Jane	2	Mary
2	Jane	1	John
3	Mary	4	Joe
3	Mary	3	Amelia
3	Mary	2	Mary
3	Mary	1	John
4	David	4	Joe
4	David	3	Amelia
4	David	2	Mary
4	David	1	John
5	Amelia	4	Joe
5	Amelia	3	Amelia
5	Amelia	2	Mary
5	Amelia	1	John

Aggregate functions

- Aggregate functions perform a calculation on multiple values and returns a single value
- Examples : AVG(), COUNT(), MAX(), MIN(), SUM()




```sql
SELECT
    AVG(buyPrice) average_buy_price
FROM
    products;
```

```sql
SELECT
    COUNT(*) AS total
FROM
    products;
```

```sql
SELECT
     MAX(buyPrice) highest_price
FROM
     products;
```

Window functions (set the environment)

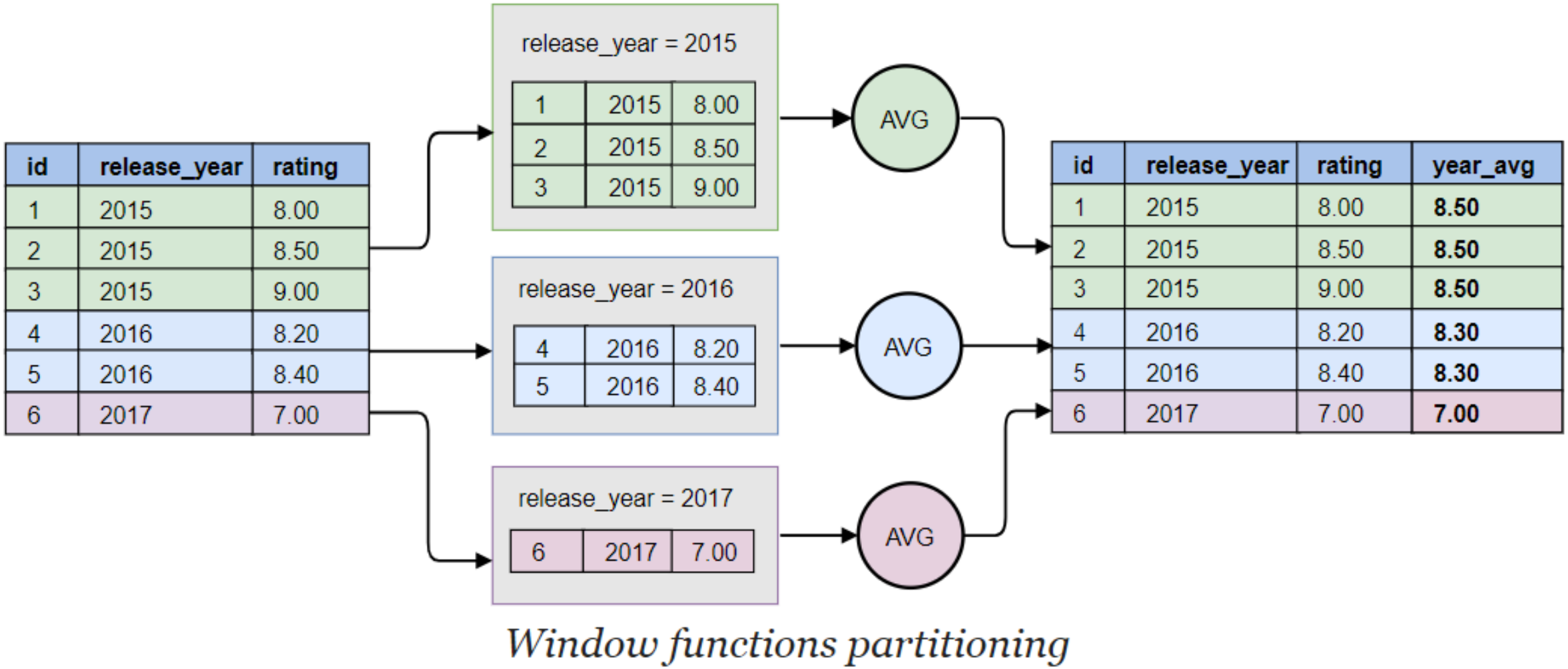
```
CREATE TABLE sales(
    sales_employee VARCHAR(50) NOT NULL,
    fiscal_year INT NOT NULL,
    sale DECIMAL(14,2) NOT NULL,
    PRIMARY KEY(sales_employee,fiscal_year)
);

SELECT * FROM sales;
SELECT
    SUM(sale)
FROM
    sales;
```

```
INSERT INTO
sales(sales_employee,fiscal_year,sale)
VALUES('Bob',2016,100),
      ('Bob',2017,150),
      ('Bob',2018,200),
      ('Alice',2016,150),
      ('Alice',2017,100),
      ('Alice',2018,200),
       ('John',2016,200),
      ('John',2017,150),
      ('John',2018,250);
```

Window functions

- A window function performs a calculation across a set of table rows that are somehow related to the current row.
- This is comparable to the type of calculation that can be done with an aggregate function.
- However, window functions do not cause rows to become grouped into a single output row. Instead, the rows retain their separate identities



Window functions partitioning

Window function syntax

- The general syntax of calling a window function is as follows:

```
window_function_name(expression) OVER (
  [partition_defintion]
  [order_definition]
  [frame_definition]
)
```

- The partition_clause breaks up the rows into chunks or partitions.
- The window function is performed within partitions and re-initialized when crossing the partition boundary.
- The ORDER BY clause specifies how the rows are ordered within a partition.
- A frame is a subset of the current partition. To define the subset, you use the frame clause as follows:
  - frame_unit {<frame_start> | <frame_between>}

Rank() function

- assigns a rank to each row within the partition of a result set.
- syntax:

```
RANK() OVER (
    PARTITION BY <expression>[{,<expression>...}]
    ORDER BY <expression> [ASC | DESC], [{,<expression>...}]
)
```

```
CREATE TABLE t (
    val INT
);

INSERT INTO t(val)
VALUES(1),(2),(2),(3),(4),(4),(5);

SELECT * FROM t;

SELECT
    val,
    RANK() OVER (
        ORDER BY val
    ) my_rank
FROM
    t;
```

	val	my_rank
▶	1	1
	2	2
	2	2
	3	4
	4	5
	4	5
	5	7

Run this

```
SELECT
    sales_employee,
    fiscal_year,
    sale,
    RANK() OVER (PARTITION BY
                    fiscal_year
                ORDER BY
                    sale DESC
                ) sales_rank
FROM
    sales;
```

Row_number

- The ROW_NUMBER() is a window function that assigns a sequential number to each row in the result set. The first number begins with one.
- The following shows the syntax of the ROW_NUMBER()function:

- ROW_NUMBER() OVER (<partition_definition> <order_definition>)

```
SELECT *,
    ROW_NUMBER() OVER(PARTITION BY val) AS row_num
FROM t;
```

val	row_num
1	1
2	1
2	2
3	1
4	1
4	2
5	1

Dense_rank

- The DENSE_RANK() is a window function that assigns a rank to each row within a partition or result set with no gaps in ranking values.
- The rank of a row is increased by one from the number of distinct rank values which come before the row.

```
SELECT
    sales_employee,
    fiscal_year,
    sale,
    DENSE_RANK() OVER (PARTITION BY
                    fiscal_year
                ORDER BY
                    sale DESC
                ) sales_rank
FROM
    sales;
```

	val	my_rank
▶	1	1
	2	2
	2	2
	3	3
	4	4
	4	4
	5	5

IF

- The IF() function returns a value if a condition is TRUE, or another value if a condition is FALSE.

- Syntax
  - IF(condition, value_if_true, value_if_false)

```
SELECT OrderID, Quantity, IF(Quantity>10, "MORE", "LESS")
FROM OrderDetails;
```

CASE

- The CASE statement goes through conditions and return a value when the first condition is met (like an IF-THEN-ELSE statement). So, once a condition is true, it will stop reading and return the result.
- If no conditions are true, it will return the value in the ELSE clause.
- If there is no ELSE part and no conditions are true, it returns NULL.
- Syntax:

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    WHEN conditionN THEN resultN
    ELSE result
END;
```

```
SELECT OrderID, Quantity,
CASE
    WHEN Quantity > 30 THEN "The quantity is greater than 30"
    WHEN Quantity = 30 THEN "The quantity is 30"
    ELSE "The quantity is under 30"
END
FROM OrderDetails;
```