Practical-Web-Database-Design

Chapter 2

---

1. data model :

- **what is data model ?**
  - it is a formalized method of structuring & manipulating data .
  - every data model should define :
    1. structure : how data is organized (table, rows, columns) .
    2. integrity : how to ensure correctness (primary key, not null, foreign key).
    3. operations : how to work with data (SQL operations) .

---

2. Relational Model (table, row, column):

- it is a logical model used to represent data .
- easy to use & understand .
- **what is relation ?**
  - A logical table made of rows (tuples) and columns (attributes), representing a set of related data .
  - suppose you have the following *sets* :
    - Employee name , position, location, salary .
  - each Employee has only one value of each set .
EMPLOYEE			
Name	**Position**	**Location**	**Salary**
Jane Smith	Manager	London	35000
Richard Jones	Analyst	New York	34000
Anna Horton	Programmer	London	22000
Alex de Tocqueville	Programmer	Paris	21000
  - Row (tuple) = Employee (Real world component) .
  - column = Properties (attributes) of the things we want to store in the database .
  - table = relation .
  - no specific order, each row is unique .
- **Notes about table (relation) :**
  1. each table has :
     - a name -> Employee table .
     - columns, rows .
  2. each column has :
     - name -> "Position" column .
     - set of allowed values called *domain* -> "Manager, Analyst, Programmer" .
  3. each cell has only one single value .
  4. Number of rows unlimited .
  5. row :
     - must be unique cause each row represents unique instance -> need of primary key .
     - rows order not important .

---

3. Domains & Data Types :

- **Data types :**
  - **what is it ?**
    - it is the type of data that will be stored inside a specific column (numbers, text, date) and hide physical details from the user .
  - **is it useful ?**
    - it helps DBMS knowing how to (Store & retrieve) data .
  - **examples ?**
    - numbers -> (INTEGER , DECIMAL) .
    - date/time -> (DATE, TIME, DATETIME) .
    - text -> (CHAR, VARCHAR) .
    - Booleans -> (TRUE, FALSE) .
- **Domain :**
  - **what is it ?**
    - set of allowed values for specific columns .
  - **example ?**
    - salary can be any number bigger than 4000 .

- we can get the effect of domains via Constrains or data entry mechanisms .

---

4. missing data : nulls

- sometimes you don't have a value to put in a column. Two reasons for this are:
  1. unknowing or missing value :
     - customer didn't provide an email address .
  2. inapplicable value :
     - commission rate for a salaried employee with no commission .
- the previous situations represented with *NULL* in Database .
- NULL is a flag means "no value", not zero or empty .
- can we rely on other values like -1 or "" (empty text)?
  - -1 : no, it may lead to some issues when dealing with some mathematical operations like (avg, sum),
  - "" empty text : no, it doesn't mean no value it is still a value .
- NULL can't be compared with other values .
- When should you use a default value instead of NULL?
  - When there is a sensible and meaningful default that fits the business context.

---

5. Primary Key :

- Why each row must be unique ?
  - in DB each row identifies a real component like "employee".
  - imagine that we have 2 identical rows for to different persons, how could we know tell them apart ? we can't cause they are identical so there must be something that identify each row .
- **what is primary key ?**
  - it is a unique key which can be one column or multiple columns (composite), used to identify each row in the table .
  - not changeable .
  - not duplicated .
  - each table must have only one primary key, tables may have unique columns which called "alternate key" .
- **why PK is important ?**
  - restrict duplication of rows .
  - used for relationships among tables .
  - for fast search .
  - used for data integrity .

---

6. Foreign Key :

- **what is it ?**
  - it is a column or set of columns contains Primary key of another table sometimes it is called reference .
- **is it important ?**
  - use it is used to link related tables with each others, make relations between tables .
  - join related rows from different tables .
- **notes about foreign key :**
  - table contains foreign key called "Child table", table refers to called "parent table" .
  - foreign key values doesn't have to be unique .
  - it must refer to an existing value in parent table .

---

7. Normalization :

- **why we don't put all data in one big table ?**
  - this will lead to more problems cause of *data redundancy* :
    1. waste space :
       - duplicating same data will take more space .
    2. update anomaly :
       - when we want to update data of specific row we must update it in all duplicated rows to avoid inconsistency .
    3. Delete anomaly :
       - if we deleted data of (order), we will remove entire row with all data inside it which may be all data of customer .
    4. insert anomaly .
- **Normalization :**
  - it is an operation of splitting big tables into smaller related tables to avoid data redundancy and anomaly problems .
  - **levels of normalization :**
    - 1NF : No multivalued attributes (each column must have atomic/single values) .
    - 2NF : no partial dependency (non-key column must depend on the whole composite primary key) (in composite PK) .
    - 3NF : All non-key columns must depend **directly** on the primary key only .
- **First Normal Form (1NF)**
  - **What is it?**
    - A table is in 1NF if it has **no repeating groups** or **multivalued attributes**.
  - **Goal:**

        - Ensure that each column contains only a single (atomic) value.
    - How to know you're in 1NF?
        - Each column contains only one value.
        - Each row represents a single entity instance.
- Second Normal Form (2NF)
    - What is it?
        - A table is in 2NF if it has no partial dependency, meaning non-key attributes depend on the entire composite primary key, not just part of it.
    - Goal:
        - Ensure that all non-key columns depend on the whole primary key.
        - Eliminate dependencies on part of a composite key.
- Third Normal Form (3NF)
    - What is it?
        - A table is in 3NF if it has no transitive dependency, meaning non-key columns depend only on the primary key, and not on another non-key column.
    - Goal:
        - Eliminate indirect dependencies by making sure every column is directly dependent on the primary key only.
- Fourth Normal Form (4NF)
    - What is it?
        - A table is in 4NF if it has no multi-valued dependencies — i.e., no column contains a set of values related to a single key.
    - Goal:
        - Ensure that the primary key determines only single values, not sets of values (even across rows).
المرحلة	الهدف الأساسي	نوع المشكلة التي تحلها	الحل
1NF	جعل الأعمدة تحتوي على قيمة واحدة فقط (Atomic)	Multivalued Attributes / Repeating Groups	تفكيك القيم إلى صفوف منفصلة
2NF	منع Partial Dependency في المفاتيح المركبة	Column يعتمد على جزء من المفتاح	تقسيم الجدول حسب المجموعات الوظيفية
3NF	منع Transitive Dependency	Column يعتمد على Column آخر غير المفتاح	فصل الأعمدة التي تعتمد على غير المفتاح لجدول منفصل
4NF	منع Multi-valued Dependency	المفتاح يحدد أكثر من قيمة لنفس العمود	فصل القيم المتعددة لعلاقات مستقلة (مثلًا many-to-many علاقة)

---

8. Data Integrity :

- What is it?
    - Integrity means that data is valid and internally consistent.
    - Integrity is achieved by defining a set of rules that the data must comply with.
- Why is it important?
    - To prevent invalid or inconsistent data from entering the system.
    - Without enforcement of integrity rules, unexpected behaviour and bugs can occur.
    - Assumptions like "a customer can't have more than 30% discount" are useless if not enforced.
- When is it defined?
    - Integrity rules are usually defined during the database design phase.
- Types of Integrity Rules
    1. Built-in Integrity Rules (enforced by the DBMS automatically):
        - Example: Primary key must be unique (Entity Integrity Rule).
    2. User-Defined Integrity Rules (defined by the developer):
        - Example:
            - Two employees can't have the same SSN.
            - Employee name can't be null.
            - Gender must be either `'m'` or `'f'`.
            - Credit limit must be between 0–5000 or null.
- Best Practices
    - User-defined rules should be:
        - Well-documented.
        - Implemented inside the database when possible.
        - Enforced at the lowest layer to avoid being bypassed by applications.
    - In web apps, you can:
        - Use JavaScript to enforce client-side integrity (client-side validation) rules before submission.
        - Make sure rules are downloaded from a trusted source (e.g., rule server).
- Built-in Integrity Rules (Relational Model) :
    1. Entity Integrity
        - Every row in a table must be uniquely identifiable.
        - Each table must have a primary key.
        - A primary key must:
            - Be unique.

        - Not contain nulls in any part of it.
    - Why? Because nulls cannot be compared, so uniqueness can't be guaranteed.
2. Referential Integrity
    - Any foreign key value must:
        - Refer to an existing value in the primary key of the parent table.
        - Or be null.
    - 🧠 Example: In the SALE table, a `CustomerNo` being null is valid. But if it has a value that doesn't exist in the CUSTOMER table, that breaks referential integrity.
3. Domain Integrity
    - All values in a column must belong to a predefined set of valid values (called a domain).
    - Examples:
        - Gender must be either `'m'` or `'f'`.
        - Age must be a positive number.
        - Salary must be within a specific range.
- ⚠️ Note:
    - Integrity rules ensure that data is *valid*, but not necessarily *correct*.
    - Example: `"Brain Watlers"` may be valid by the system, but it's a typo for `"Brian Walters"`.
    - ✅ Best practices:
        - Use data verification (e.g., double entry).
        - Combine integrity rules + verification for more reliable data.

---

9. Metadata & the Data Dictionary :

- What do we store in a database?
    - Mainly data, but also metadata.
- What is Metadata?
    - Metadata = "data about data"
    - It includes definitions and structures of the database, not the actual business data (like sales or customers).
    - Example: When you create a table, the database stores info about that table (its name, columns, keys, etc.).
- Where is metadata stored?
    - In a special part of the database called the catalogue or data dictionary.
- What's stored in the Data Dictionary?
    1. Table definitions
        - Table name, number of columns, primary keys, storage info
    2. Column info
        - Column names, data types, sizes, and which table they belong to
    3. Integrity rules
        - Such as primary keys, foreign keys, constraints
    4. User info
        - Usernames, passwords
    5. Access privileges
        - Who can read/write/update/delete what
    6. Indexes
        - Details about indexes used for faster searching
    7. Database statistics
        - Sizes of tables, average row sizes, number of queries, etc.
- Note:
    - The data dictionary is mostly used internally by the DBMS.
    - In relational databases, the dictionary itself is made up of tables.
    - Example: In Oracle, the data dictionary contains hundreds of system tables.

---

10. Physical Data access models :

📌 Why This Matters

- Performance is critical for multi-user or web-based databases with many concurrent transactions.
- Physical data storage and access are central to database performance, not the logical design (like the relational model).

🔽 Performance Bottleneck: Disk Access

- Accessing data in RAM takes nanoseconds.
- Accessing data on disk takes milliseconds (i.e., ~1 million times slower).
- With millions of transactions, even milliseconds add up → response time matters (users get frustrated >3s wait).

🧱 Data Storage Basics

- Data is stored in files, divided into blocks (pages), which are the smallest unit of I/O.
- Each block can contain many records.

- Disk I/O performance = minimizing **disk accesses**.

🔍 Physical Access Methods

1. Sequential Access

- Search starts from the beginning of the file.
- Goes **record by record** until the desired one is found.
- **Very slow** for large tables (millions of records = thousands of disk reads).
- **Analogy:** Reading a phone book from cover to cover.

---

2. Indexed Access

- Uses an **index** (like a book's index) to **quickly find** a record.
- Most common: **B-tree index** (hierarchical, sorted structure).
- Fast because:
  - Few disk accesses needed (often 1–4).
  - Index often cached in RAM.
- **Range queries** (e.g., "names starting with C") are efficient.
- **Downsides:**
  - Slower if lots of updates/deletes.
  - Tree needs rebalancing = expensive operation.
  - Not useful on very small tables.

---

3. Direct Access (Hashing)

- Uses a **hash function** to calculate **exact disk location**.
- Fastest method – often **1 disk access**.
- Best for:
  - **Very large tables** (millions of records).
  - **Frequent single-record lookups**.
- **Bad for range queries**, because similar values aren't stored together.
- Think of it like assigning each key a storage "slot" based on its hash.

⚠️ Key Notes

- **Indexes** can be:
  - **Unique**: no duplicate keys.
  - **Non-unique**: allows duplicates.
- **Primary keys** often get indexed automatically.
- Use **indexing wisely**
  - Great for reads.
  - Can **slow down writes/updates**.
- Avoid indexing **tiny tables** – may not help.

✅ When to Use What

Access Method	Best Use Case	Speed	Range Queries?	Drawbacks
Sequential	Small tables, no index available	Slow	Yes	Very inefficient on large tables
Indexed (B-tree)	General-purpose, mixed queries	Fast	Yes ✅	Costly to maintain on updates
Direct (Hashing)	Huge tables, frequent exact lookups	Fastest (1 I/O)	No ❌	Poor for ranges, insert overhead

---

Chapter 3

What is SQL ?

- Structured Query language .
- declarative not procedural .
- Main purpose: retrieve data from databases, but also used to create, modify, and delete database structures and records.
- standard for interact with relational database .

SQL in web :

- in web we don't use sql .
- we use stored procedures instead :

- more security .
- reduce network usage .

Syntax Components :

- Statements "Commands" in sql (used to tell RDBMS what we want) split to 3 types :
  - *DDL (Data Definition Language) :*
    - used to define structure (creation) of tables .
    - contains five main statements (CREATE table, DROP table, ALTER table, CREATE index, DROP index).
  - *DML (Data Manipulation Language) :*
    - used to manipulate & view data .
    - contains 4 statements (UPDATE, DELETE, INSERT, SELECT) .
  - *DCL (Data Control Language) :*
    - used to manage access, permissions, ownership of database objects .

Create table :

🔷 What does `CREATE TABLE` do?

- Used to define the structure of a **new table** inside an existing database.
- Syntax is **portable** across RDBMSs (unlike creating the database itself).
- Can be used من خلال:
  - **SQL command line**
  - **Scripts**

🔷 Basic Syntax

```
CREATE TABLE <table-name> (
  <column-name> <data-type> [DEFAULT <value>] [<column-constraints>],
  ...
  [<table-constraints>]
);
```

✅ Syntax Breakdown:

Element	Description
`<table-name>`	اسم الجدول
`<column-name>`	اسم العمود
`<data-type>`	نوع البيانات (زي CHAR, INT...)
`DEFAULT <value>`	(اختياري) قيمة افتراضية للعمود
`<column-constraints>`	(اختياري) قيود العمود (زي NOT NULL, CHECK...)
`<table-constraints>`	(اختياري) قيود على الجدول بالكامل (زي PRIMARY KEY...)
`;`	نهاية الأمر (مهم جدًا في SQL)

> ❗ *SQL is not case-sensitive, but capitalizing keywords improves readability.*

🧪 Example – CUSTOMER Table

CustomerNo	First	Last	Address	CreditLimit
CHAR(9)	CHAR(20)	CHAR(20)	VARCHAR(100)	SMALLINT

💻 SQL Code:

```
CREATE TABLE CUSTOMER (
  CustomerNo   CHAR(9)      NOT NULL,
  First        CHAR(20)     NOT NULL,
  Last         CHAR(20)     NOT NULL,
  Address      VARCHAR(100),
  CreditLimit  SMALLINT     CHECK (CreditLimit > 0),
  PRIMARY KEY (CustomerNo)
);
```

🔍 Explanation of Constraints Used:

Constraint	Purpose
`NOT NULL`	يمنع أن يكون العمود فارغًا
`CHECK (...)`	يحدد شرطًا منطقيًا يجب أن يتحقق لقيمة العمود

Constraint	Purpose
`PRIMARY KEY (...)`	يحدد العمود الذي يمثل المفتاح الأساسي (قيم فريدة + غير فارغة)

- `CREATE TABLE` defines table schema.
- Use data types carefully.
- Constraints like `NOT NULL`, `CHECK`, `PRIMARY KEY` ensure data integrity.
- Separate items with commas, and finish with a semicolon.

SQL-92 Data Types

🔹 1. Numeric Data Types

Type	Use for	Max Value (تقريبًا)
`SMALLINT`	أرقام صغيرة	حتى 32,767
`INTEGER`	أرقام صحيحة أكبر	حتى 2.1 مليار
`NUMERIC(p,s)`	أرقام عشرية بدقة ثابتة	`p`: digits total, `s`: digits after decimal
`FLOAT`	أرقام عشرية بفاصلة عائمة	غير دقيق 100%، مناسب للقياسات العلمية

> ⚠ *بعض المنتجات (زي Oracle) بتستخدم أنواعها الخاصة زي `NUMBER` بدل `NUMERIC`.*

🔹 2. Date and Time

Type	Use for
`DATE`	التاريخ فقط (يوم/شهر/سنة)

🔹 3. Character Data Types

Type	وصف	خصائص
`CHAR(n)`	نص بطول ثابت (Fixed-length)	يُملأ بمسافات إضافية إذا كانت القيمة أقصر من `n`
`VARCHAR(n)`	نص بطول متغير (Variable-length)	لا يُهدر مساحة، لكن أبطأ في المعالجة

🔍 Example from CUSTOMER Table:

```
First CHAR(20) Last CHAR(20) Address VARCHAR(100)
```

✅ **Interpretation:**

- First/Last: أسرع للبحث → حرف 20 دائمًا.
- Address: أفضل لتوفير المساحة → طول متغير حسب الحاجة.

🔄 CHAR vs VARCHAR – أيهما تختار؟

الحالة	النوع الأنسب
القيم غالبًا بنفس الطول	`CHAR(n)`
الأعمدة يتم البحث فيها كثيرًا	`CHAR(n)`
نصوص طويلة / نادرة الاستخدام	`VARCHAR(n)`
توفير مساحة على حساب السرعة	`VARCHAR(n)`

> ✳ *مثال: أرقام الهواتف أو الأكواد تستخدم `CHAR`، بينما العناوين تستخدم `VARCHAR`*

Column & Table Constraint :

- Column Constraints: تنطبق على عمود واحد.
- Table Constraints: تنطبق على جدول كامل (وممكن تشتغل على أكثر من عمود مع بعض).

🔹 ما هي القيود (Constraints)؟

> ***قيود = قواعد تحكم البيانات***
> *تمنع إدخال بيانات خاطئة أو غير متوافقة مع منطق النظام.*

النوع	يطبق على	مثال
Column Constraint	عمود واحد فقط	`CreditLimit SMALLINT NOT NULL`
Table Constraint	جدول كامل (عدة أعمدة)	`PRIMARY KEY (A, B)`

🔑 أنواع القيود الأساسية:

Constraint	الوظيفة	مثال
`PRIMARY KEY`	يحدد المفتاح الأساسي. يمنع التكرار والـ null	`PRIMARY KEY (CustomerNo)`
`FOREIGN KEY`	يحافظ على التكامل المرجعي (ربط بين جداول)	`FOREIGN KEY (OrderID) REFERENCES Orders(ID)`
`NOT NULL`	يمنع القيم الفارغة (null)	`FirstName CHAR(20) NOT NULL`
`UNIQUE`	يمنع التكرار في العمود (يستخدم كمفتاح بديل)	`Email VARCHAR(50) UNIQUE`
`CHECK (...)`	يفرض شرط منطقي على القيم	`CHECK (CreditLimit >= 0)`
`CONSTRAINT <name>`	تسمية القيد لسهولة التعامل معه لاحقًا	`CONSTRAINT CreditCheck CHECK (CreditLimit > 0)`

🔍 مثال عملي - جدول `CUSTOMER` :

```
CREATE TABLE CUSTOMER (
  CustomerNo   CHAR(9) PRIMARY KEY,
  First        CHAR(20) NOT NULL,
  Last         CHAR(20) NOT NULL,
  Address      VARCHAR(100) NOT NULL,
  CreditLimit  SMALLINT CHECK (CreditLimit >= 0)
);
```

✅ شرح المثال:

- `CustomerNo` : تلقائيًا ⇐ مفتاح أساسي `NOT NULL + UNIQUE`.
- `First` , `Last` , `Address` : لا يُسمح بـ null.
- `CreditLimit` : يُسمح بـ null، 0 ≤ لازم ⇐ لكن لو دخلت قيمة.

⚠️ ملاحظات مهمة:

- القيود المركبة (Composite Key):
تستخدم أكثر من عمود كمفتاح، مثل:

```
PRIMARY KEY (A, B, C)
```

- null ≠ 0 أو ""
  - `CHECK (x > 0)` لا تمنع `NULL`

- الحل: استخدم `NOT NULL` بجانب `CHECK` إذا لزم الأمر.
- عدم تكرار NOT NULL مع PRIMARY KEY:
  - `PRIMARY KEY` بالفعل يمنع null ⇐ تكرار `NOT NULL` غير ضروري ويبطّئ الأداء.
- CONSTRAINT :
  - تسمية القيود مهم لو عايز تعدّلها أو تحذفها باستخدام `ALTER TABLE` .

✅ أفضل الممارسات:

الحالة	النصيحة
عمود ممكن يدخل في عمليات منطقية (AND/OR)	لا تسمح بـ null
تريد تقييد القيمة ضمن نطاق معين	استخدم `CHECK (value BETWEEN x AND y)`
عمود لازم يكون فريد ومفتاح بديل	استخدم `UNIQUE`

DROP table :

- used to delete table data & structure from data base .
- DROP table customer
- This completely removes the table — data, columns, constraints — everything .

ALTER table :

يُستخدم لتعديل هيكل الجدول الموجود دون الحاجة لحذفه وإعادة إنشائه.

📌 أنواع استخدامات `ALTER TABLE`:

1. ➕ ADD Form

لإضافة عمود جديد أو قيد (Constraint) جديد.

📄 صيغة:

```
ALTER TABLE table_name ADD COLUMN column_name data_type [DEFAULT value] [NOT NULL];
ALTER TABLE table_name ADD CONSTRAINT constraint_name ...;
```

🧠 ملاحظات:

- الأعمدة الجديدة تُملأ بـ `NULL` أو بالقيمة الافتراضية (إذا تم تحديدها).
- لا يمكنك إضافة عمود `NOT NULL` إلا إذا أعطيته قيمة `DEFAULT` .
- لا يمكن في SQL-92 إضافة قيد `NOT NULL` على عمود موجود مباشرة (نستخدم طرق بديلة أو خصائص مخصصة في RDBMS حديثة).

🧪 مثال:

```
ALTER TABLE EMPLOYEE ADD COLUMN ManagerID INTEGER DEFAULT 0 NOT NULL;
ALTER TABLE EMPLOYEE ADD FOREIGN KEY (ManagerID) REFERENCES EMPLOYEE(EmployeeID);
```

2. 🛠️ ALTER Form

لتعديل الخصائص مثل القيمة الافتراضية.

📄 صيغة:

```
ALTER TABLE table_name ALTER COLUMN column_name SET DEFAULT value;
ALTER TABLE table_name ALTER COLUMN column_name DROP DEFAULT;
```

🧪 مثال:

```
ALTER TABLE EMPLOYEE ALTER COLUMN ManagerID SET DEFAULT 0;
```

---

3. ❌ DROP Form

لحذف عمود أو قيد (Constraint) من الجدول.

📄 صيغة:

```
ALTER TABLE table_name DROP COLUMN column_name;
ALTER TABLE table_name DROP CONSTRAINT constraint_name;
```

🧠 ملاحظات:

- البيانات داخل العمود المحذوف تُفقد.
- لا يمكن حذف عمود مشارك في قيد (مثل Foreign Key) دون حذف القيد أولًا.
- يجب معرفة اسم القيد إذا أردت حذفه.

🧩 مثال تطبيقي شامل:

🥕 تعديل عمود `CreditLimit` من **nullable** إلى **NOT NULL**:

1. أضف عمود جديد `TempLimit` :

```
ALTER TABLE CUSTOMER ADD COLUMN TempLimit SMALLINT DEFAULT 0 NOT NULL CHECK (TempLimit >= 0);
```

2. انسخ البيانات إليه:

```
UPDATE CUSTOMER SET TempLimit = CreditLimit WHERE CreditLimit IS NOT NULL;
```

3. احذف العمود القديم:

```
ALTER TABLE CUSTOMER DROP COLUMN CreditLimit;
```

4. أنشئ العمود من جديد بنفس الاسم لكن بقيود جديدة:

```
ALTER TABLE CUSTOMER ADD COLUMN CreditLimit SMALLINT DEFAULT 0 NOT NULL CHECK (CreditLimit >= 0);
UPDATE CUSTOMER SET CreditLimit = TempLimit;
```

```
ALTER TABLE CUSTOMER DROP COLUMN TempLimit;
```

CREATE & DROP INDEX

🧠 ما هو الـ Index؟

الـ **Index** هو هيكل بيانات يُستخدم لتسريع عمليات البحث (SELECT) في الجداول، مثل الفهرس في كتاب لتسهيل الوصول للمحتوى.

📌 أولًا: `CREATE INDEX`

📄 الصيغة:

```
CREATE [UNIQUE] INDEX index_name ON table_name (column1, column2, ...);
```

🔷 **`UNIQUE` (اختياري):**

- يستخدم إذا كنت تريد منع تكرار القيم في الأعمدة المفهرسة.
- يُستخدم غالبًا مع الأعمدة التي تمثل **Primary Key** أو **قيود فريدة (Unique Constraints)**.

🧪 **أمثلة:**

```
-- فهرس عادي (غير فريد) على اسم العميل:
CREATE INDEX CustomerNameIndex ON CUSTOMER (Last, First);

-- فهرس فريد على رقم الموظف (بما أنه Unique):
CREATE UNIQUE INDEX EmployeeIDIndex ON EMPLOYEE (EmployeeID);
```

🔎 **ملاحظات:**

- لا يُشترط أن تكون الأعمدة مفاتيح أساسية أو فريدة لتتم فهرستها.
- معظم قواعد البيانات **تُنشئ فهارس تلقائيًا على الـ Primary Keys** و الـ Unique constraints.

❌ ثانيًا: `DROP INDEX`

📄 **الصيغة:**

```
DROP INDEX index_name;
```

🧪 **مثال:**

```
DROP INDEX CustomerNameIndex;
```

⚠️ **تحذير:**

- حذف الفهرس **لا يحذف الأعمدة أو البيانات**، فقط يوقف تسريع الوصول المرتبط به.

🧠 ملاحظات عامة:

- **الفهارس تُسرّع عمليات SELECT**، لكن قد تُبطئ عمليات `INSERT`, `UPDATE`, `DELETE` لأن النظام يحتاج إلى تحديث الفهرس أيضًا.
- **اختر الأعمدة المفهرسة بعناية** — لا تفهرس كل شيء.

✅ `INSERT` — إدخال بيانات جديدة

🔷 **الغرض:**

إضافة صف جديد (Row) في جدول موجود مسبقًا.

📄 **الصيغة العامة:**

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

- يمكن حذف الجزء الخاص بالأعمدة، لكن **يفضّل دائمًا كتابته لتجنب الأخطاء**.
- الترتيب مهم جدًا إذا حذفت أسماء الأعمدة.

🧪 **أمثلة:**

✅ **مثال مكتمل:**

```sql
INSERT INTO EMPLOYEE (EmployeeID, Name, Position, Location, Salary)
VALUES (617258, 'Jane Smith', 'Manager', 'London', 35000);
```

✅ مثال بدون إدخال لقيمة المفتاح الأساسي (Auto-generated):

```sql
INSERT INTO EMPLOYEE (Name, Position, Location, Salary)
VALUES ('Heather Ralaton', 'Manager', 'Melbourne', 33000);
```

- في هذا المثال، قاعدة البيانات تولد `EmployeeID` تلقائيًا (مثلاً باستخدام `AUTO_INCREMENT` في MySQL أو `SERIAL` في PostgreSQL).

🔍 استخدام القيم الافتراضية أو `NULL`

✅ إدخال مع قيمة افتراضية أو `NULL`:

```sql
-- يتم إدخال NULL تلقائيًا إذا سمح العمود بها:
INSERT INTO CUSTOMER (CustomerNo, First, Last, Address)
VALUES ('5794-3467', 'Eric', 'Wilberforce', '9558 Great South Road');

-- يتم إدخال NULL يدويًا:
INSERT INTO SALE (SaleNo, SaleDate, CustomerNo, ProductNo, Qty, Amount, SalesRep)
VALUES (12348, '2002-08-13', NULL, 'DHU69863', 50, 118.5, 'Sara Thompson');
```

> 💡 *إذا تم تقييد العمود بعدم قبول NULL، وكان له **قيمة افتراضية** (`DEFAULT`)، فسيتم استخدام هذه القيمة بدلًا من NULL.*

💬 ملاحظات مهمة:

الملاحظة	الشرح
🔤 النصوص توضع بين علامات تنصيص `'`	مثل `'Jane Smith'`
❗ لا تخلط بين `'` و `"`	لأن بعضها يستخدم لأسماء الأعمدة (حسب الـ RDBMS)
⚠️ ترتيب القيم مهم جدًا	خصوصًا إذا لم تذكر أسماء الأعمدة
📅 التواريخ	يجب أن تكون بصيغة `YYYY-MM-DD` (وقد تختلف حسب قاعدة البيانات)

ممتاز، دخلت الآن في ثاني عملية من عمليات CRUD وهي `UPDATE`، الخاصة بتعديل البيانات الموجودة بالفعل في قاعدة البيانات.

دعني ألخص لك هذا الدرس بشكل منظم ومبسط 👇

✅ تعديل البيانات — `UPDATE`

🔷 الغرض:

تعديل بيانات موجودة في صفوف من جدول معين.

📄 الصيغة العامة:

```sql
UPDATE table_name
SET column1 = value1, column2 = value2, ...
[WHERE condition];
```

> 🟡 *`WHERE` اختيارية، ولكن بدونها سيتم تعديل **كل الصفوف** في الجدول!*

🧪 أمثلة:

🔶 تعديل كل الرواتب (جميع الصفوف):

```sql
UPDATE EMPLOYEE SET Salary = 37000;
```

> ⚠️ *هذا يعدل **جميع** الموظفين!*

🔶 تعديل صف واحد باستخدام المفتاح الأساسي:

```sql
UPDATE EMPLOYEE SET Salary = 37000
WHERE EmployeeID = 617258;
```

> ✅ *هذا آمن لأنه يستخدم مفتاح فريد.*

🔸 تعديل باستخدام الاسم (غير مفضل لأنه غير فريد):

```
UPDATE EMPLOYEE SET Salary = 37000
WHERE Name = 'Jane Smith';
```

> ❌ *قد يغيّر أكثر من صف!*

🔸 تعديل أكثر من عمود:

```
UPDATE CUSTOMER
SET CreditLimit = 0, Address = 'Iowa State Penitentiary'
WHERE CustomerNo = '4649-4673';
```

> ✅ *تغيير أكثر من عمود في نفس العملية.*

🔸 استخدام NULL في التحديث:

```
UPDATE CUSTOMER
SET CreditLimit = NULL, Address = 'Xowa State Penitentiary'
WHERE CustomerNo = '4649-4673';
```

> ✅ *يمكنك استخدام `NULL` كقيمة جديدة، ولكن للمقارنة لا تستخدم `=` بل `IS NULL`.*

🔸 تعديل باستخدام عملية حسابية:

```
UPDATE PRODUCT
SET Price = Price * 1.1;
```

> ✅ *زيادة كل الأسعار بنسبة 10%. نستخدم نفس العمود كـ "مصدر" و"وجهة".*

💡 ملاحظات مهمة:

الملاحظة	الشرح
⚠️ بدون WHERE = كارثة	يحدث تعديل لكل الصفوف
👄 استخدم مفتاح فريد في WHERE	مثل `ID` أو `CustomerNo`
🧮 يمكن استخدام تعبيرات رياضية	مثل `Price * 1.1`
❌ لا تستخدم `=` مع NULL للمقارنة	بل استخدم `IS NULL`

ممتاز، الآن وصلت لآخر عملية في CRUD وهي `DELETE`، المسؤولة عن حذف البيانات من قاعدة البيانات.
دعني ألخص لك هذا الدرس بشكل واضح ومنظم:

🗑️ حذف البيانات — `DELETE`

🔹 الغرض:

حذف صف (أو أكثر) من جدول موجود.

📄 الصيغة العامة:

```
DELETE FROM table_name
[WHERE condition];
```

> 🟡 *مثل `UPDATE`، إذا **لم تكتب** `WHERE` سيتم حذف **جميع الصفوف** في الجدول!*

🧪 أمثلة:

🔸 حذف منتج باستخدام رقم المنتج:

```
DELETE FROM PRODUCT
```

```sql
WHERE ProductNo = 'AAD62726';
```

> ✅ *أفضل استخدام العمود الفريد مثل* `ProductNo`.

◆ حذف موظفين براتب ≥ 40000 في نيويورك:

```sql
DELETE FROM EMPLOYEE
WHERE Location = 'New York' AND Salary >= 40000;
```

> ✅ *استخدام شرطين بربط* `AND`.

◆ حذف مبيعات بدون رقم زبون:

```sql
DELETE FROM SALE
WHERE CustomerNo IS NULL;
```

> ✅ *عند التعامل مع* `NULL`، *نستخدم* `IS NULL` *بدلًا من* `= NULL`.

◆ حذف كل المنتجات (بدون شرط):

```sql
DELETE FROM PRODUCT;
```

> ⚠ ***يحاول حذف كل الصفوف***، *لكن:*

- إذا كان هناك قيود مفتاح خارجي (foreign key) من جدول آخر (مثلاً `SALE` يعتمد على `PRODUCT`)، العملية قد تفشل للحفاظ على سلامة البيانات (referential integrity).

🔒 ماذا تعني Referential Integrity؟

- يعني أن الصفوف المرتبطة بمفاتيح خارجية يجب أن تشير إلى صفوف صحيحة موجودة.
- إذا حاولت حذف صف من جدول أساسي (مثل `PRODUCT`) وكان هناك صف في جدول آخر (مثل `SALE`) يعتمد عليه، فإن قاعدة البيانات ستمنع الحذف تلقائيًا.

💡 ملاحظات سريعة:

الملاحظة	الشرح
حذف شامل = `WHERE` بدون `DELETE` ⚠	لا تستخدمه إلا وأنت متأكد
✅ استخدم مفاتيح فريدة في الشرط	مثل `ProductNo` , `CustomerID`
`= NULL` بدلًا من `IS NULL` ❗	عند التعامل مع بيانات مفقودة
🔄 يتم التحقق من السلامة المرجعية	لمنع حذف بيانات يعتمد عليها جداول أخرى

المقارنة	`DELETE`	`TRUNCATE`
ينتمي لـ SQL رسميًا؟	✅ نعم	❌ لا (لكن مدعوم من أغلب قواعد البيانات)
الأداء	أبطأ — يسجل كل حذف	أسرع — لا يسجل كل حذف
يمكن استخدام `WHERE`؟	✅ نعم	❌ لا — يحذف كل الصفوف دائمًا
يسجل في Log؟	✅ نعم	❌ لا
يعيد تسلسل ID؟	❌ لا	✅ غالبًا نعم

🔍 WHERE في SQL

📌 ما هي؟

هي الجملة التي تُستخدم لتحديد أي الصفوف (rows) يجب أن تتأثر بالأمر (`SELECT` , `DELETE` , `UPDATE`).

💥 لماذا WHERE مهمة جدًا؟

✅ بدون `WHERE`:

```sql
DELETE FROM EMPLOYEE;
```

- هذا الأمر يحذف جميع الموظفين من الجدول!

✅ مع `WHERE`:

```
DELETE FROM EMPLOYEE WHERE Department = 'Sales';
```

- هذا يحذف فقط الموظفين في قسم المبيعات.

⚠️ تحذير للمبتدئين

- نسيان جملة `WHERE` هو أحد أكثر الأخطاء شيوعًا في SQL!
- حتى إن كتبتها، فقد تكون غير دقيقة وتؤدي لتعديل أو حذف خاطئ.

✅ كيف تختبر جملة WHERE بأمان؟

> *لا تجرب جملة WHERE مباشرة مع `UPDATE` أو `DELETE` على قاعدة بيانات حقيقية (production).*

🧪 اختبرها أولًا هكذا:

```
SELECT * FROM EMPLOYEE WHERE Department = 'Sales';
```

- هذا يريك ما هي الصفوف التي ستتأثر دون حذف أو تعديل فعلي.

🧠 كيف تعمل WHERE خلف الكواليس؟

- يتم تنفيذ التعبير الموجود بعد `WHERE` على كل صف (row) في الجدول.
- إذا أعطى **True** → الصف يُؤثّر عليه.
- إذا أعطى **False** → الصف يُتجاهل.

🧷 يمكنك استخدام شروط منطقية مركبة:

```
UPDATE PRODUCT
SET Price = Price * 2
WHERE ((QtyInStock > 5000) OR (Price <= 10))
AND ReorderLevel < 1000;
```

هذا الشرط المعقد يعني:

- زد السعر للضعف إذا:
  - كان المخزون > 5000 أو السعر ≤ 10
  - و مستوى إعادة الطلب أقل من 1000

🔥 مثال على DELETE مع شرط مركب:

```
DELETE FROM SALE
WHERE (
  (SaleDate BETWEEN '2002-08-13' AND '2002-08-14') AND (Salesrep = 'Li Qing')
  OR (ProductNo <> 'DHU69863')
  OR (Qty > 50)
);
```

سيتم حذف أي صف يُحقق:

- مبيعات قام بها Li Qing بين 13 و 14 أغسطس 2002.
- أو المنتج ليس هو DHU69863.
- أو كمية البيع > 50.

🧪 رموز مفيدة في WHERE

الرمز	المعنى
`=`	يساوي
`<>`	لا يساوي (قياسي)
`!=`	لا يساوي (بعض الأنظمة)
`BETWEEN`	بين قيمتين

الرمز	المعنى
`IS NULL`	القيمة فارغة
`AND`	كلا الشرطين يجب أن يتحققا
`OR`	يكفي أن يتحقق أحد الشرطين
`NOT`	نفي الشرط

🧠 ملخص ونصائح ذهبية:

- ✅ دائمًا اختبر `WHERE` باستخدام `SELECT`.
- ⚠️ لا تنس `WHERE` عند استخدام `DELETE` أو `UPDATE`.
- 🧪 استخدم قاعدة بيانات اختبارية عند تجربة شروط معقدة.
- 💾 خذ نسخ احتياطية قبل أي تعديل كبير.

🔷 Querying in SQL: SELECT

🟦 شرح مبسط:

كل الاستعلامات (queries) في SQL تتم عن طريق الأمر `SELECT`.
الهدف من الاستعلام هو إنك تطلب بيانات من قاعدة البيانات، ومين بيحدد شكل البيانات اللي راجعة؟ أنت، من خلال تحديد الأعمدة (columns)، الجداول (tables)، والشرط (conditions).

🛠️ الصيغة الأساسية:

```
SELECT <columns>
FROM <tables>
[WHERE <conditions>];
```

✳️ مثال:

لو عايز تطبع أسماء ورواتب الموظفين اللي شغالين في باريس:

```
SELECT Name, Salary
FROM EMPLOYEE
WHERE Location = 'Paris';
```

📌 ملاحظات:

- **SELECT**: بتحدد الأعمدة اللي عايزها في النتيجة.
- **FROM**: بتحدد الجدول اللي هتستعلم منه.
- **WHERE**: شرط لتصفية النتائج (مش إجباري، ممكن تستغنى عنه).

🔷 شرح SELECT * (الكل)

🟦 شرح مبسط:

لما تستخدم `SELECT *`، معناها "هات كل الأعمدة من الجدول".
مثال:

```
SELECT * FROM EMPLOYEE;
```

✳️ نتيجة الاستعلام:

كل صفوف وأعمدة جدول EMPLOYEE هتظهر.

⚠️ تحذير:

- استخدام `SELECT *` يعتبر **ممارسة سيئة (bad practice)** في عالم قواعد البيانات.
- بيستهلك **وقت وسعة سيرفر** لو الجدول كبير.
- ممكن يعرض **بيانات حساسة** لو تم إضافة أعمدة جديدة لاحقًا بدون قصد.

🔷 فهم ترتيب التنفيذ في SELECT

🟦 ترتيب التنفيذ المنطقي:

1. **FROM**: تحديد الجداول.
2. **WHERE**: تصفية الصفوف.
3. **SELECT**: تحديد الأعمدة اللي هتظهر.

> *بالتالي، تقدر تستخدم أعمدة في WHERE حتى لو مش ظاهرة في SELECT.*

🔷 أمثلة تطبيقية على SELECT:

✅ استعلام: الموظفين في لندن

```
SELECT Name FROM Employee
WHERE Location = 'London';
```

✅ استعلام: رصيد منتج اسمه **Grunge nut**

```
SELECT ProductNo, Description, QtyinStock
FROM PRODUCT
WHERE ProductNo = 'FGE91822';
```

✅ استعلام: العملاء اللي عندهم حد ائتماني أكبر من 6000

```
SELECT CustomerNo, First, Last
FROM CUSTOMER
WHERE CreditLimit > 6000;
```

🔷 استعلام مبيعات خلال يومين

✅ باستخدام OR:

```
SELECT SaleNo, SaleDate, Amount
FROM SALE
WHERE SaleDate = '2002-08-12' OR SaleDate = '2002-08-13';
```

✅ باستخدام BETWEEN:

```
SELECT SaleNo, SaleDate, Amount
FROM SALE
WHERE SaleDate BETWEEN '2002-08-12' AND '2002-08-13';
```

✅ باستخدام IN:

```
SELECT SaleNo, SaleDate, Amount
FROM SALE
WHERE SaleDate IN ('2002-08-12', '2002-08-13');
```

💡 ملحوظة:

- لازم تكتب التواريخ بصيغة تناسب قاعدة البيانات بتاعتك (مثلاً `YYYY-MM-DD`).
- **IN** مفيد جدًا لو عندك لستة قيم.

🔷 استعلام مبيعات عميل اسمه Amelia Waverley

❌ الطريقة الغلط:

```
SELECT SaleNo, SaleDate, Amount, Salesrep
FROM CUSTOMER, SALE
WHERE First = 'Amelia' AND Last = 'Waverley';
```

ده بيطلع كل التركيبات الممكنة بين جدول CUSTOMER وSALE → يعني بيانات كتير جدًا ومغلوطة.

🟦 ليه؟

لأن استخدام أكتر من جدول في `FROM` من غير `JOIN` بيعمل **Cartesian Product** (الضرب الديكارتي).

✅ الطريقة الصحيحة: باستخدام JOIN

> *نستخدم العلاقة بين الجداول (مفتاح أساسي ومفتاح أجنبي) للربط بينهم.*

```
SELECT SaleNo, SaleDate, Amount, Salesrep
FROM CUSTOMER
JOIN SALE ON CUSTOMER.CustomerNo = SALE.CustomerNo
WHERE First = 'Amelia' AND Last = 'Waverley';
```

🔍 التوضيح:

- `JOIN` : بيربط جدول CUSTOMER بجدول SALE على أساس CustomerNo.
- `WHERE` : بيحدد اسم العميل اللي احنا بندور عليه.

JOINS :

📌 أولًا: لماذا نحتاج Joins؟

لأن قواعد البيانات مصممة بطريقة التطبيع (Normalization)، فالمعلومات موزعة على أكثر من جدول. لذلك إذا أردنا الحصول على بيانات متكاملة (مثلاً: اسم العميل وتفاصيل المشتريات)، نحتاج إلى ضم الجداول (Join) للحصول على نتيجة موحدة.

🔗 أنواع الـ Joins:

1. Condition-Based Joins (الانضمام بشرط)

وهو النوع الأكثر استخدامًا والأكثر وضوحًا.

📜 Pre-SQL-92 Style (الأسلوب القديم):

- نضع شروط الربط في جملة `WHERE`.
- مثال:

```
SELECT SaleNo, SaleDate, Amount, Salesrep
FROM CUSTOMER, SALE
WHERE CUSTOMER.CustomerNo = SALE.CustomerNo
  AND First = 'Amelia' AND Last = 'Waverley';
```

✅ SQL-92 Style (الأسلوب الحديث الموصى به):

- نستخدم الكلمة المفتاحية `JOIN`، وشرط الربط باستخدام `ON`.
- أكثر وضوحًا وسهولة في القراءة.
- مثال:

```
SELECT SaleNo, SaleDate, Amount, Salesrep
FROM CUSTOMER
JOIN SALE ON CUSTOMER.CustomerNo = SALE.CustomerNo
WHERE First = 'Amelia' AND Last = 'Waverley';
```

> ✨ *الأفضل دائمًا استخدام SQL-92 لأنه أكثر قابلية للفهم والصيانة.*

---

3. Outer Joins (الانضمام الخارجي)

تُستخدم عندما تريد عرض جميع الصفوف من أحد الجداول، حتى لو لم يكن لها تطابق في الجدول الآخر.

⬅ Left Outer Join:

- يعرض كل الصفوف من الجدول الأيسر حتى لو لم يكن لها تطابق في الجدول الأيمن.
- مثال:

```
SELECT PRODUCT.ProductNo, PRODUCT.Description, SALE.SaleNo, SALE.Amount
FROM PRODUCT LEFT OUTER JOIN SALE
ON PRODUCT.ProductNo = SALE.ProductNo;
```

➡ Right Outer Join:

- يعرض كل الصفوف من الجدول الأيمن حتى لو لم يكن لها تطابق في الجدول الأيسر.

⬌ Full Outer Join:

- يعرض كل الصفوف من كلا الجدولين، سواء تطابقت أم لا (نادراً ما يُستخدم).

✅ نصائح عامة:

- استخدم أسماء أعمدة واضحة لتجنب اللبس.
- من الأفضل دائمًا تحديد أسماء الجداول مع الأعمدة (مثل `SALE.SaleNo`) حتى يكون الكود واضحًا.
- احرص على أن تكون العلاقات بين الجداول واضحة (عبر مفاتيح أساسية وأجنبية).
- تجنب استخدام `SELECT *` و `NATURAL JOIN` في الإنتاج، فهي قد تسبب نتائج غير متوقعة.

🎯 ملخص سريع:

النوع	هل موصى به؟	ماذا يفعل؟
`NATURAL JOIN`	❌	يربط الجداول تلقائيًا حسب الأعمدة المشتركة (خطير)
`INNER JOIN`	✅	يعرض فقط الصفوف المتطابقة في الجدولين
`LEFT OUTER JOIN`	✅	يعرض كل الصفوف من الجدول الأيسر، والمتطابقة من الأيمن
`RIGHT OUTER JOIN`	✅	يعرض كل الصفوف من الجدول الأيمن، والمتطابقة من الأيسر

النوع	هل موصى به؟	ماذا يفعل؟
`FULL OUTER JOIN`	🔁	يعرض كل الصفوف من كلا الجدولين، مع Nulls في حال عدم وجود تطابق

🔍 المشكلة الأساسية: Joins مقابل الأداء

- ضرورية في قواعد البيانات **المطبقة لل (normalization) لتجنب التكرار.
- لكن في بيئة الويب، الأداء له أولوية قصوى لأن سرعة استجابة الموقع تؤثر مباشرة على تجربة المستخدم وربما أرباح الشركة.
- الانضمامات المعقدة (Joins) خاصةً مع بيانات ضخمة (مثل آلاف العملاء وملايين المبيعات) قد تبطئ التطبيق.

💡 الحل: Pre-compute joins

✅ عندما يكون الأداء أهم من التطبيع الكامل:

- إذا كانت البيانات ثابتة أو شبه ثابتة (مثل سجل مبيعات قديمة أو درجات طلاب تم اعتمادها)، يمكن:
  - تحضير نتائج الـ Join مسبقًا.
  - تخزين النتيجة في جدول جديد يحتوي على بيانات مكررة لكنها جاهزة للعرض بسرعة.

📦 مثال عملي: جدول SALE_HISTORY

1. يتم إنشاؤه مرة واحدة:

```
CREATE TABLE SALE_HISTORY (
  CustomerNo CHAR(9),
  First CHAR(20) NOT NULL,
  Last CHAR(20) NOT NULL,
  Address VARCHAR(100) NOT NULL,
  ProductNo CHAR(8),
  Description VARCHAR(50) NOT NULL,
  Price NUMERIC(5,2) NOT NULL CHECK (Price >= 0 and Price < 999.99),
  SaleNo SMALLINT,
  SaleDate DATE NOT NULL,
  Qty INTEGER NOT NULL CHECK (Qty > 0),
  Amount NUMERIC(6,2) NOT NULL CHECK (Amount >= 0),
  PRIMARY KEY (SaleNo)
);
```

2. ثم تعبئته ببيانات مشتقة من Join بين الجداول الأصلية:

```
INSERT INTO SALE_HISTORY (SaleNo, SaleDate, Qty, Amount, ProductNo,
                          Description, Price, CustomerNo, First, Last, Address)
SELECT SALE.SaleNo, SALE.SaleDate, SALE.Qty, SALE.Amount,
       PRODUCT.ProductNo, PRODUCT.Description, PRODUCT.Price,
       CUSTOMER.CustomerNo, CUSTOMER.First, CUSTOMER.Last, CUSTOMER.Address
FROM SALE, PRODUCT, CUSTOMER
WHERE SALE.ProductNo = PRODUCT.ProductNo
  AND SALE.CustomerNo = CUSTOMER.CustomerNo;
```

🔁 بعدها، يُحدَّث هذا الجدول دوريًا:

- باستخدام Triggers أو Scheduled Jobs.
- أو باستخدام مفهوم Materialized View.

🎓 أمثلة تطبيقية شائعة

- سجل مشتريات العميل في موقع تسوّق.
- كشف درجات طالب في جامعة.
- فواتير العميل في موقع خدمات.

📝 ملاحظات إضافية:

- بعض قواعد البيانات مثل SQL Server تدعم اختصارات مثل `SELECT INTO` لإنشاء جداول مباشرة من الاستعلام (لكن هذا غير قياسي).
- التكرار في البيانات هنا مبرر إذا كان الهدف تحسين سرعة استرجاعها دون الحاجة لـ Join في كل مرة.
- هذا لا يلغي أهمية التطبيع، لكنه يدعو إلى حل وسط ذكي.

📊 1. Reporting with SQL: Key Features

🔷 A. Sorting with `ORDER BY`

- Purpose: Arrange the output rows in a desired sequence.
- Syntax:

```
SELECT <columns>
FROM <table>
ORDER BY <column1> [ASC|DESC], <column2> [ASC|DESC];
```

- **Examples**:
  - Sort by position (ascending):

    ```
    SELECT Name, Position FROM EMPLOYEE ORDER BY Position;
    ```

  - Sort by salary (descending):

    ```
    SELECT Name, Salary FROM EMPLOYEE ORDER BY Salary DESC;
    ```

  - Sort by position (descending), then salary (ascending):

    ```
    SELECT Name, Position, Salary
    FROM EMPLOYEE
    ORDER BY Position DESC, Salary;
    ```

🔷 B. Removing Duplicates with `DISTINCT`

- **Purpose**: Get unique values from a column (or combination of columns).
- **Syntax**:

```
SELECT DISTINCT <column> FROM <table>;
```

- **Example**:
  - Get unique sale dates:

    ```
    SELECT DISTINCT SaleDate FROM SALE;
    ```

📚 2. Real-World Use in Web Applications

✅ Why It Matters:

- **Efficient display**: Reduces redundant data shown to users (e.g., repeated dates, names, etc.).
- **Better UX**: Sorting improves readability of reports or dashboards.
- **Performance-aware**: Aggregation and sorting can be expensive on large datasets — pre-computing or caching results is often needed.

🛠️ Tips for Web Apps:

- **Use sorting and filtering on backend (SQL) when possible**, instead of fetching raw data and sorting in the front end.
- **Combine `DISTINCT` with `ORDER BY`** when you want a clean, sorted list of unique values.

📊 2. Aggregation in SQL

🧮 Built-in Aggregate Functions

Aggregate functions summarize data from multiple rows into a single result:

Function	Description
`COUNT(*)`	Total number of rows
`SUM(column)`	Total sum of values in a column
`MIN(column)`	Smallest value in a column
`MAX(column)`	Largest value in a column
`AVG(column)`	Average value in a column

📜 Basic Example

> *How many employees do we have? What's the total, minimum, maximum, and average salary?*

```
SELECT
  COUNT(*) AS total_employees,
  SUM(Salary) AS total_salary,
  MIN(Salary) AS min_salary,
  MAX(Salary) AS max_salary,
```

```
  AVG(Salary) AS avg_salary
FROM EMPLOYEE;
```

🧠 **Notes:**

- All functions return a **single result**.
- These apply to the **entire table** unless restricted with `WHERE`.
- Null values are **ignored** by all except `COUNT(*)`.

🧮 Using Expressions Inside Aggregates

> *What's the total value of our inventory?*

```
SELECT SUM(QtyInStock * Price) AS total_inventory_value
FROM PRODUCT;
```

You can use arithmetic expressions inside aggregate functions!

🧍 COUNT with DISTINCT

> *How many unique job positions are there?*

```
SELECT COUNT(DISTINCT Position) AS unique_positions
FROM EMPLOYEE;
```

✅ **Use Cases:**

- Avoid counting duplicate values.
- Helps generate cleaner reports.

⚠️ Important Rule

You **can't mix** aggregate functions and regular columns unless you use a `GROUP BY` clause.

❌ Invalid:

```
SELECT COUNT(*), Name FROM EMPLOYEE; -- ❌ Syntax Error
```

✅ Valid only with grouping:

```
SELECT Position, COUNT(*) FROM EMPLOYEE GROUP BY Position;
```

🔍 What Does `COUNT(*)` Mean?

- Counts **all rows**, even if some columns have `NULL`.
- Doesn't rely on any specific column's values.
- Fastest way to get total row count.

Use `COUNT(DISTINCT column)` if:

- You want to **ignore duplicates**.
- You only want to **count non-null unique** values.

✅ Why Aggregation Matters in Web Apps

- Useful for dashboards: totals, averages, summaries.
- Reduces the need for app-side calculations.
- Helps with performance if done in SQL instead of code.

📂 3. Grouping Rows with `GROUP BY`

🧠 What is `GROUP BY`?

`GROUP BY` is used to **group rows** that have the **same value** in one or more columns, often to apply **aggregate functions** (`COUNT`, `SUM`, etc.) **per group**.

🔢 Basic Example

> *How many employees do we have **in each position**?*

```sql
SELECT Position, COUNT(*) AS employee_count
FROM EMPLOYEE
GROUP BY Position;
```

Position	employee_count
Analyst	1
Manager	1
Programmer	2

Explanation:

- SQL groups by `Position`, then counts rows in each group.
- Every group produces **one row** in the result.

🧱 Grouping by Multiple Columns

> *Group employees by **position and location**.*

```sql
SELECT Position, Location
FROM EMPLOYEE
GROUP BY Position, Location;
```

This returns unique combinations like:

- Analyst + New York
- Manager + London
- Programmer + London
- Programmer + Paris

Rule: Any "regular" column in `SELECT` **must** be in `GROUP BY`.

❌ You Can't Use Aggregate Functions in `WHERE`

> *Find sales reps with total sales > 2000:*

❌ Invalid:

```sql
SELECT Salesrep
FROM SALE
WHERE SUM(Amount) > 2000
GROUP BY Salesrep;
```

Error: You can't use `SUM` or other aggregates in `WHERE`.

✅ Use `HAVING` Instead of `WHERE` for Group Conditions

✔ Valid:

```sql
SELECT Salesrep
FROM SALE
GROUP BY Salesrep
HAVING SUM(Amount) > 2000;
```

Explanation:

- `GROUP BY` makes groups.
- `HAVING` filters **groups** based on aggregate conditions.

🧠 Summary of Filtering:

Clause	Filters On	Works With Aggregates?
`WHERE`	Individual rows	❌ No
`HAVING`	Grouped rows	✅ Yes

⚡ Performance Tip

If a `GROUP BY` is used **frequently** on large tables:

- You can **pre-compute** the result.

- Save it in a **separate summary table**.
- Useful if data is **static or slow-changing**.

💡 Real-Life Use Cases

- Total sales per product or rep.
- Users per country or plan type.
- Page views per day/week.
- Inventory value per category.

✅ Rules Recap

1. Columns in `SELECT` must be in `GROUP BY` (unless inside an aggregate).
2. Use `HAVING` to filter grouped results.
3. Don't mix aggregates and regular columns without `GROUP BY`.

ملاحظات مهمه :

✅ التعليقات في SQL

تقدر تكتب تعليقات داخل أو بجانب أي سطر كود باستخدام `--` .
دي مفيدة جدًا لو بتحفظ الكود ده علشان تستخدمه لاحقًا، زي لما تكتبه داخل **stored procedures** (هنتكلم عنها في الفصل ٨).

✅ التعبيرات (Expressions) والثوابت (Constants) في SELECT

مش لازم تستخدم أسماء أعمدة أو دوال فقط في `SELECT`، تقدر تستخدم تعبيرات رياضية أو حتى ثوابت.

مثال على تعبير رياضي:

```
SELECT Description, QtyInStock - ReorderLevel FROM PRODUCT;
```

دي بتحسب الفرق بين الكمية الموجودة وكمية إعادة الطلب (يعني فاضل قد إيه قبل ما تحتاج تطلب تاني).

مثال على ثوابت:

```
SELECT Salesrep, 'has sold to', First, Last FROM SALE JOIN CUSTOMER ON ...;
```

كلمة `'has sold to'` هتظهر زي ما هي في كل صف، كأنها جملة ثابتة بتوضح العلاقة.

✅ إعادة تسمية الجداول (Table Aliasing)

ممكن تستخدم أسماء مختصرة للجداول علشان تسهّل الكتابة وتخلي الكود أوضح، خاصة لما تعمل `JOIN` .

مثال:

```
SELECT S.SaleNo, C.First FROM SALE AS S JOIN CUSTOMER AS C ON S.CustomerNo = C.CustomerNo;
```

> *ملاحظة: `AS` اختيارية. ممكن تكتب `SALE S` بدل `SALE AS S` .*

✅ الـ Self Join

أحيانًا عايز تربط الجدول بنفسه، زي مثلاً لما يكون عندك موظف ومديره في نفس جدول `EMPLOYEE` .

المشكلة:

```
SELECT Name, Name FROM EMPLOYEE JOIN EMPLOYEE ON ManagerID = EmployeeID;
```

ده هيطلعلك خطأ لإن النظام مش عارف أن الـ EMPLOYEE هنا معناها حاجتين مختلفتين.

الحل باستخدام alias:

```
SELECT E.Name, M.Name
FROM EMPLOYEE E JOIN EMPLOYEE M ON E.ManagerID = M.EmployeeID;
```

- هنا بتمثل الموظف `E`
- بتمثل المدير `M`

> *ملاحظة مهمة: **الـ RDBMS ما بيعملش نسخة من الجدول لما تستخدم alias**. هي مجرد أسماء مؤقتة مش أكتر.*

✅ إعادة تسمية الأعمدة (Column Aliasing)

مفيدة جدًا خصوصًا لما تستخدم دوال تجميعية أو تعبيرات، علشان تطلع بأسماء مفهومة.

مثال بدون alias:

```
SELECT AVG(Price), AVG(QtyInStock) FROM PRODUCT;
```

ده هيطلعلك عمودين من غير أسماء واضحة.

مثال باستخدام alias:

```
SELECT AVG(Price) AS AveragePrice, AVG(QtyInStock) AS AverageQty FROM PRODUCT;
```

> *كمان تقدر تسمي ناتج أي تعبير رياضي:*

```
SELECT Description, (QtyInStock - ReorderLevel) AS QtyRemaining FROM PRODUCT;
```

🧠 إزاي تكتب SELECT Query ؟

أنت مش بس محتاج تكتب الكويري، كمان لازم تتأكد إنه بيرجع النتيجة الصح.
عشان كده، في خطوات وتقنيات تقدر تتبعها:

✅ أولًا: ابدأ بـ FROM

> *"ابدأ دايمًا بـ الجداول اللي هتشتغل عليها."*

- اسأل نفسك: "أنا هحتاج أنهي جدول/جداول؟"
- لو أكتر من جدول → أكيد في JOIN.
  - اتأكد إنك كاتب join condition صح.
  - واختار النوع المناسب من الـ join (INNER / LEFT / ...).
- لو هتعمل join لنفس الجدول → استخدم alias (تسمية مؤقتة) للجداول.

✅ ثانيًا: حدد الصفوف المطلوبة → WHERE

> *"هسترجع أي صفوف؟"*

- حدد شروط الفلترة اللي محتاجها → حطها في WHERE.
- مثلًا: `WHERE salary > 5000 AND department = 'IT'`
- اختبر الشروط دي كويس جدًا، لأن أي خطأ هنا هيجيبلك نتايج غلط.
  - جرّب كل شرط لوحده الأول.
  - بعدين ضيفهم واحد ورا التاني، واختبر كل مرة.

✅ ثالثًا: هل في تجميع؟ → GROUP BY

> *"هل محتاج أجمع بيانات؟ زي عدد الطلبات لكل عميل؟"*

- لو أه → استخدم GROUP BY.
- الأعمدة اللي بتجّمع بيها، لازم تكتبها في SELECT.
- لو محتاج تفلتر بعد التّجميع → استخدم HAVING.

مثال:

```
SELECT department, COUNT(*)
FROM employees
GROUP BY department
HAVING COUNT(*) > 5;
```

✅ رابعًا: ايه الأعمدة اللي هتظهر؟ → SELECT

> *"محتاج أعرض انهي بيانات؟"*

- اختار الأعمدة اللي محتاجها في النتيجة.
- ممكن تستخدم دوال، aliases، expressions، إلخ.

✅ خامسًا: عايز النتيجة تكون مترتبة؟ → ORDER BY

> *"هل محتاج أرتب النتائج؟"*

- استخدم `ORDER BY` لترتيب النتيجة.

- ممكن ترتب تصاعدي (ASC) أو تنازلي (DESC).

💡 نصائح ذهبية لبناء Queries صح:

1. استخدم بيانات اختبار كويسة.
2. قسّم الكويري لمراحل، واشتغل عليهم واحدة واحدة.
   - ابدأ مثلًا بـ FROM + WHERE بس.
   - شوف النتيجة، لو تمام → كمّل GROUP BY.
3. اختبر، اختبر، اختبر.
   - جرب كويري بسيط.
   - بعدين زوّد شرط شرط.
   - كل خطوة لازم تتأكد إن النتيجة مظبوطة.

🧪 مثال عملي:

المطلوب:

"عايز أعرف عدد الموظفين في كل قسم، بشرط إن المرتب أكبر من 4000، وارتبهم حسب الاسم تنازليًا."

خطوات التحليل:

1. **FROM** → جدول الموظفين `employees`.
2. **WHERE** → `salary > 4000`.
3. **GROUP BY** → `department`.
4. **SELECT** → `department`, `COUNT(*)`.
5. **ORDER BY** → `department DESC`.

الكويري:

```sql
SELECT department, COUNT(*) AS NumEmployees
FROM employees
WHERE salary > 4000
GROUP BY department
ORDER BY department DESC;
```
