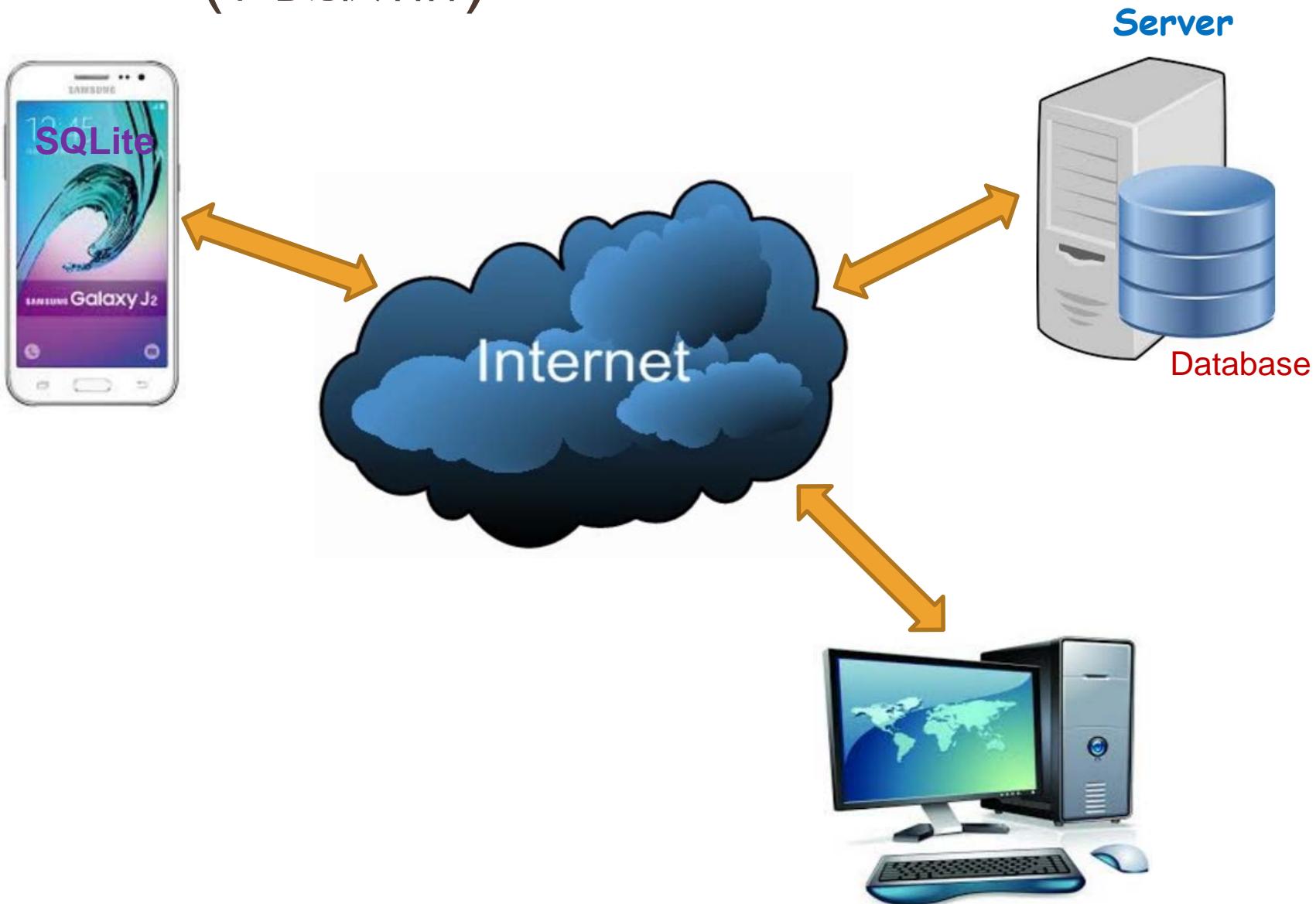


# 資料庫+SQL操作

---

William Wang

# Server (伺服器)



# phpmyadmin

- Run <http://localhost/phpmyadmin/> on your browser

The screenshot shows the phpMyAdmin configuration interface. At the top, there's a navigation bar with tabs for 資料庫, SQL, 狀態, 使用者帳號, 噴出, 備援, 變數, 字元集, 引擎, and 附加元件. Below the navigation bar, there are two main configuration sections: '一般設定' (General Settings) and '外觀設定' (Appearance Settings). In '一般設定', the '伺服器連線編碼與排序' dropdown is set to 'utf8mb4\_unicode\_ci'. In '外觀設定', the '語言 - Language' dropdown is set to '中文 - Chinese traditional', the '佈景主題' dropdown is set to 'pmahomme', and the '字體大小' (Font Size) dropdown is set to '82%'. To the right of these sections, there are three summary boxes: '資料庫伺服器' (Database Server), '網頁伺服器' (Web Server), and 'phpMyAdmin'. The '資料庫伺服器' box contains information about the MySQL connection, including the host (127.0.0.1 via TCP/IP), type (MariaDB), and version (10.4.6-MariaDB). The '網頁伺服器' box lists the web server (Apache/2.4.39), PHP version (7.3.8), and MySQL version (5.0.12-dev). The 'phpMyAdmin' box shows the current version (4.9.0.1) and links to the documentation, homepage, and other resources.

phpMyAdmin

伺服器: 127.0.0.1

資料庫 SQL 狀態 使用者帳號 噴出 備援 變數 字元集 引擎 附加元件

一般設定

伺服器連線編碼與排序: utf8mb4\_unicode\_ci

外觀設定

語言 - Language: 中文 - Chinese traditional

佈景主題: pmahomme

字體大小: 82%

更多設定

資料庫伺服器

- 伺服器: 127.0.0.1 via TCP/IP
- 伺服器類型: MariaDB
- 伺服器連線: 未有使用 SSL
- 伺服器版本: 10.4.6-MariaDB - mariadb.org binary distribution
- 協定版本: 10
- 使用者: root@localhost
- 伺服器字元集: cp1252 West European (latin1)

網頁伺服器

- Apache/2.4.39 (Win64) OpenSSL/1.1.1c PHP/7.3.8
- 資料庫用戶端版本: libmysql - mysqlnd 5.0.12-dev - 20150407 - \$Id: 7cc7cc96e675f6d72e5cf0f267f48e167c2abb23 \$
- PHP 擴充套件: mysqli curl mbstring
- PHP 版本: 7.3.8

phpMyAdmin

- 版本資訊: 4.9.0.1
- 說明文件
- 官方首頁
- 貢獻
- 技術支援
- 版本沿革
- 授權

# Exercise-Load schema

1. create a database name as “ksu\_database”
2. Import “ksu\_database.sql” schema for the database “ksu\_database”

# Insert data



在資料表 ksu\_database.ksu\_std\_table 執行 SQL 查詢: [?](#)

```
1 INSERT INTO `ksu_std_table`(`ksu_std_id`, `ksu_std_name`, `ksu_std_age`, `ksu_std_department`, `ksu_std_signin`) VALUES ('4070E001', 'John wish', 21, 'IE', 2018)
```



顯示第 0 - 0 列 (總計 1 筆, 查詢用了 0.0040 秒。)

```
SELECT * FROM `ksu_std_table`
```

效能分析

全部顯示

資料列數 :

25

篩選資料列:

+ 選項

←→

ksu\_std\_id

ksu\_std\_name

ksu\_std\_age

ksu\_std\_department

ksu\_std\_signin

編輯

複製

刪除

4070E001

John wish

21

IE

2018



上一頁

下一頁

前一頁

後一頁

首頁

尾頁

# Exercise

- Ch12.ppt - page10
- Column name: Name, Address, phone

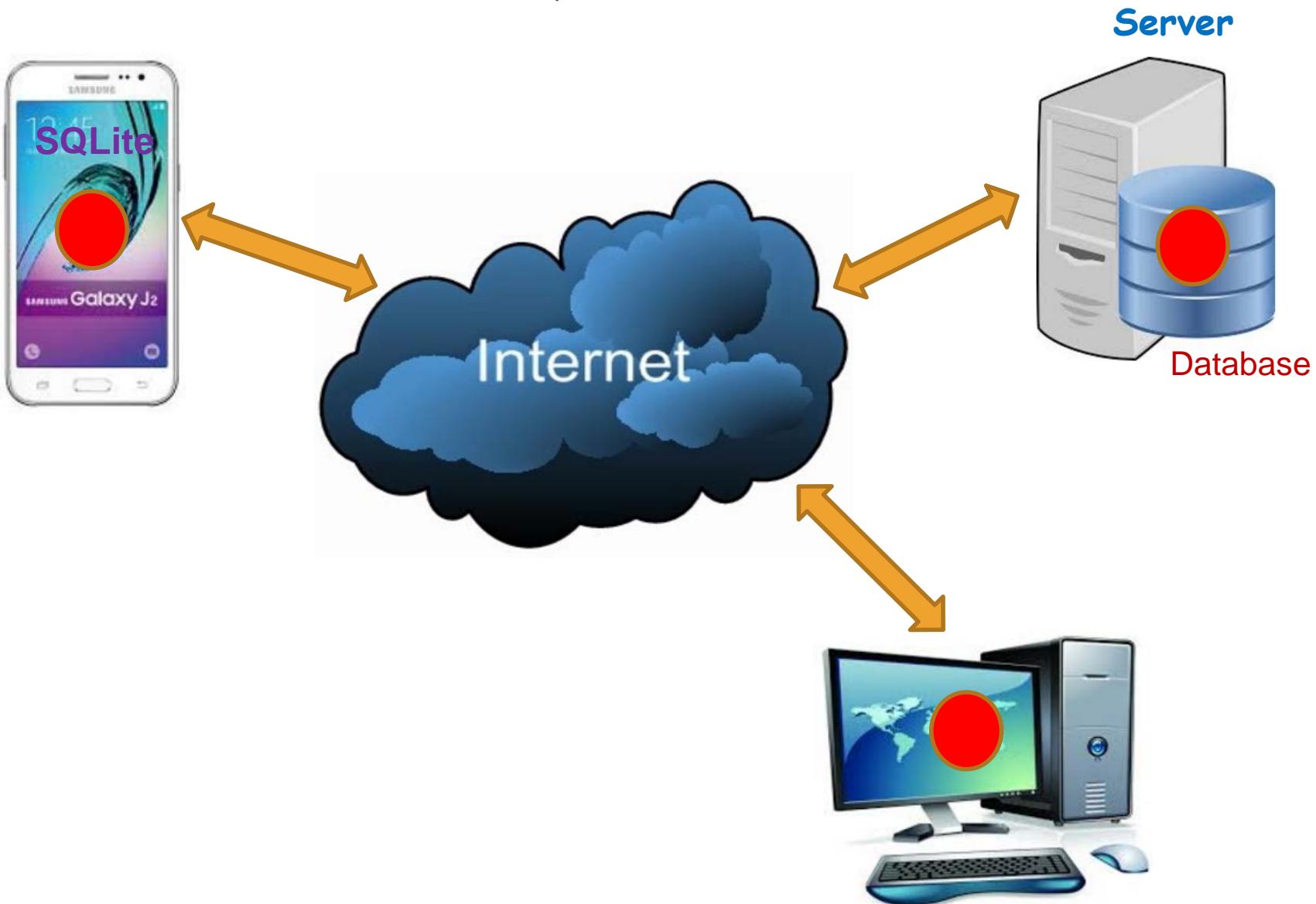
# Exercise

- Ch12.ppt - page12
- 2 tables

# SQL

<https://www.1keydata.com/tw/sql/sql.html>

# Where is the SQLs?



# SQL

- Oracle SQL 的分類 **DML**、**DCL**、**DDL** 、**TCL**
- **DDL** 資料定義語言 Data Definition Language (**DDL**)
- 相關指令：
  - CREATE - to create objects in the database
  - ALTER - alters the structure of the database
  - DROP - delete objects from the database
  - TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed
  - COMMENT - add comments to the data dictionary
  - RENAME - rename an object

# SQL

- DML 資料操作語言 Data Manipulation Language (**DML**)
- 相關指令：
  - SELECT** - retrieve data from the a database
  - INSERT** - insert data into a table
  - UPDATE** - updates existing data within a table
  - DELETE** - deletes all records from a table, the space for the records remain
  - MERGE** - UPSERT operation (insert or update)
  - CALL** - call a PL/SQL or Java subprogram
  - EXPLAIN PLAN** - explain access path to data
  - LOCK TABLE** - control concurrency

# SQL

- DCL 資料控制語言 Data Control Language (**DCL**)

- 相關指令 :

GRANT - gives user's access privileges to database

REVOKE - withdraw access privileges given with the  
GRANT command

# SQL

- TCL 交易控制語言 Transaction Control (TCL)
- 相關指令：

COMMIT - save work done

SAVEPOINT - identify a point in a transaction to which you can later roll back

ROLLBACK - restore database to original since the last COMMIT

SET TRANSACTION - Change transaction options like isolation level and what rollback segment to use

# SQL-SELECT – stage 1

<https://www.1keydata.com/tw/sql/sql.html>

# SQL - SELECT

- SQL 是用來做什麼的呢？一個最常用的方式是將資料從資料庫中的表格內選出。從這一句回答中，我們馬上可以看到兩個關鍵字：**從 (FROM) 資料庫中的表格內選出 (SELECT)**。(表格是一個資料庫內的結構，它的目的是儲存資料。在表格處理這一部分中，我們會提到如何使用 SQL 來設定表格。) 我們由這裡可以看到最基本的 SQL 架構：
- **SELECT “欄位名” FROM “表格名”;**

# An Example

- 我們用以下的例子來看看實際上是怎麼用的。假設我們有以下這個表格：

Store_Name	Sales	Txn_Date
Los Angeles	1500	2019-10-05
San Diego	250	2019-10-07
Los Angeles	300	2019-10-08
Boston	700	2019-10-08

- 若要選出所有的店名 (Store\_Name)，我們就鍵入：

**SELECT Store\_Name FROM Store\_Information;**

結果:

✓ 顯示第 0 - 3 列 (總計 4 筆, 查詢用了 0.0020 秒。)

```
SELECT Store_Name FROM Store_Information
```

全部顯示

資料列數 :

25 ▾

篩選資料列:

+ 選項

**Store\_Name**

Los Angeles

San Diego

Los Angeles

Boston



# SELECT Distinct

- **SELECT** 指令讓我們能夠讀取表格中一個或數個欄位的所有資料。這將把所有的資料都抓出，無論資料值有無重複。在資料處理中，我們會經常碰到需要找出表格內的不同資料值的情況。換句話說，我們需要知道這個表格/欄位內有哪些不同的值，而每個值出現的次數並不重要。這要如何達成呢？在 SQL 中，這是很容易做到的。我們只要在 **SELECT** 後加上一個 **DISTINCT** 就可以了。  
**DISTINCT** 的 語法如下：

```
SELECT DISTINCT "欄位名"  
FROM "表格名";
```

# An Example

Store_Name	Sales	Txn_Date
Los Angeles	1500	2019-10-05
San Diego	250	2019-10-07
Los Angeles	300	2019-10-08
Boston	700	2019-10-08

```
SELECT DISTINCT Store_Name FROM Store_Information
```

全部顯示 | 資料列數 : 25 ▾

篩選資料列:

+ 選項

Store\_Name

Los Angeles

San Diego

Boston



# SQL Where

- 我們並不一定每一次都要將表格內的資料都完全抓出。在許多時候，我們會需要選擇性地抓資料。就我們的例子來說，我們可能只要抓出營業額超過 \$1,000 的資料。要做到這一點，我們就需要用到 **WHERE** 這個指令。這個指令的語法如下：

```
SELECT "欄位名"  
FROM "表格名"  
WHERE "條件";
```

# An Example

Store_Name	Sales	Txn_Date
Los Angeles	1500	2019-10-05
San Diego	250	2019-10-07
Los Angeles	300	2019-10-08
Boston	700	2019-10-08

✓ 顯示第 0 - 0 列 (總計 1 筆, 查詢用了 0.0020 秒。)

```
SELECT Store_Name FROM Store_Information WHERE Sales > 1000
```



全部顯示

| 資料列數 :

25 ▾

篩選資料列:

+ 選項

**Store\_Name**

Los Angeles



# About the above example

- How to tell the output “Los Angeles” comes from?

One of the solutions

```
SELECT Store_Name, Sales FROM Store_Information WHERE Sales > 1000
```

全部顯示

資料列數：

25 ▾

篩選資料列：搜尋此資料表

+ 選項

Store_Name	Sales
Los Angeles	1500



# An Example

Store_Name	Sales	Txn_Date
Los Angeles	1500	2019-10-05
San Diego	250	2019-10-07
Los Angeles	300	2019-10-08
Boston	700	2019-10-08

```
SELECT Store_Name, Sales FROM Store_Information WHERE Txn_Date > "2019-10-7"
```

全部顯示 | 資料列數 : 25 ▾

篩選資料列:

+ 選項

Store_Name	Sales
Los Angeles	300
Boston	700



# An Example

Store_Name	Sales	Txn_Date
Los Angeles	1500	2019-10-05
San Diego	250	2019-10-07
Los Angeles	300	2019-10-08
Boston	700	2019-10-08

```
SELECT sales, Txn_Date FROM Store_Information WHERE Store_Name ="Los Angeles"
```



全部顯示

資料列數：

▼

篩選資料列:



+ 選項

sales	Txn_Date
1500	2019-10-05
300	2019-10-08

# SQL AND OR

- 我們看到 **WHERE** 指令可以被用來由表格中有條件地選取資料。這個條件可能是簡單的(像上一頁的例子)，也可能是複雜的。複雜條件是由二或多個簡單條件透過 **AND** 或是 **OR** 的連接而成。一個 SQL 語句中可以有無限多個簡單條件的存在。

複雜條件的語法如下：

```
SELECT "欄位名"  
FROM "表格名"  
WHERE "簡單條件"  
{[AND|OR] "簡單條件"}+;
```

{ }+ 代表{ }之內的情況會發生一或多次。在這裡的意思就是 **AND** 加簡單條件及 **OR** 加簡單條件的情況可以發生一或多次。另外，我們可以用( )來代表條件的先後次序。

舉例來說，我們若要在 **Store\_Information** 表格中選出所有 Sales 高於 \$1,000 或是 Sales 在 \$500 及 \$275 之間的資料的話，

# An Example

Store_Name	Sales	Txn_Date
Los Angeles	1500	2019-10-05
San Diego	250	2019-10-07
Los Angeles	300	2019-10-08
Boston	700	2019-10-08

```
SELECT Store_Name FROM Store_Information WHERE Sales > 1000 OR (Sales < 500 AND Sales > 275)
```

效能

全部顯示 | 資料列數 : 25 ▾

篩選資料列:

+ 選項

**Store\_Name**

Los Angeles

Los Angeles



# SQL IN

- 在 SQL 中，在兩個情況下會用到 **IN** 這個指令；這一頁將介紹其中之一：與 **WHERE** 有關的那一個情況。在這個用法下，我們事先已知道至少一個我們需要的值，而我們將這些知道的值都放入**IN** 這個子句。**IN** 指令的語法為下：

```
SELECT "欄位名"  
FROM "表格名"  
WHERE "欄位名" IN ('值一', '值二', ...);
```

在括弧內可以有一或多個值，而不同值之間由逗點分開。值可以是數目或是文字。若在括弧內只有一個值，那這個子句就等於

```
WHERE "欄位名" = '值一'
```

# An Example

Store_Name	Sales	Txn_Date
Los Angeles	1500	2019-10-05
San Diego	250	2019-10-07
Los Angeles	300	2019-10-08
Boston	700	2019-10-08

```
SELECT * FROM Store_Information WHERE Store_Name IN ('Los Angeles', 'San Diego')
```

全部顯示 | 資料列數： 25 ▾

篩選資料列： 搜尋此資料表

+ 選項

Store_Name	Sales	Txn_Date
Los Angeles	1500	2019-10-05
San Diego	250	2019-10-07
Los Angeles	300	2019-10-08

# SQL BETWEEN

- **IN** 這個指令可以讓我們依照一或數個不連續 (discrete) 的值的限制之內抓出資料庫中的值，而 **BETWEEN** 則是讓我們可以運用一個範圍 (range) 內抓出資料庫中的值。  
**BETWEEN** 這個子句的語法如下：

```
SELECT "欄位名"  
FROM "表格名"  
WHERE "欄位名" BETWEEN '值一' AND '值二';
```

這將選出欄位值包含在值一及值二之間的每一筆資料。

# An Example

Store_Name	Sales	Txn_Date
Los Angeles	1500	2019-10-05
San Diego	250	2019-10-07
Los Angeles	300	2019-10-08
Boston	700	2019-10-08

```
SELECT * FROM Store_Information WHERE Txn_Date BETWEEN '2019-10-06' AND '2019-10-08'
```

全部顯示

資料列數 :

25 ▾

篩選資料列:

+ 選項

Store_Name	Sales	Txn_Date
San Diego	250	2019-10-07
Los Angeles	300	2019-10-08
Boston	700	2019-10-08



# SQL 萬用字元

有的時候，我們需要依照由字串模式中找出相符的資料。要滿足這個需求，我們就需要用到萬用字元 (wildcard) 的做法。SQL 中有兩個萬用字元：

% (百分比符號)：代表零個、一個、或數個字母。

\_ (底線)：代表剛好一個字母。

萬用字元是與 **LIKE** 關鍵字一起使用的。

以下是幾個萬用字元的例子：

- 'A\_Z'：所有以 'A' 起頭，另一個任何值的字原，且以 'Z' 為結尾的字串。'ABZ' 和 'A2Z' 都符合這一個模式，而 'AKKZ' 並不符合（因為在 A 和 Z 之間有兩個字元，而不是一個字元）。
- 'ABC%'：所有以 'ABC' 起頭的字串。舉例來說，'ABCD' 和 'ABCABC' 都符合這個模式。
- '%XYZ'：所有以 'XYZ' 結尾的字串。舉例來說，'WXYZ' 和 'ZZXYZ' 都符合這個模式。
- '%AN%'：所有含有 'AN' 這個模式的字串。舉例來說，'LOS ANGELES' 和 'SAN FRANCISCO' 都符合這個模式。
- '\_AN%'：所有第二個字母為 'A' 和第三個字母為 'N' 的字串。舉例來說，'SAN FRANCISCO' 符合這個模式，而 'LOS ANGELES' 則不符合這個模式。

# SQL LIKE

- **LIKE** 是另一個在 **WHERE** 子句中會用到的指令。基本上，**LIKE** 能讓我們依據一個模式 (pattern) 來找出我們要的資料。相對來說，在運用 **IN** 的時候，我們完全地知道我們需要的條件；在運用 **BETWEEN** 的時候，我們則是列出一個範圍。**LIKE** 的語法如下：

```
SELECT "欄位名"  
FROM "表格名"  
WHERE "欄位名" LIKE {模式};
```

{模式} 經常包括萬用字元 (wildcard)。

# An Example

Store_Name	Sales	Txn_Date
Los Angeles	1500	2019-10-05
San Diego	250	2019-10-07
Los Angeles	300	2019-10-08
Boston	700	2019-10-08

```
SELECT * FROM Store_Information WHERE store_name LIKE '%AN%'
```

全部顯示 | 資料列數 : 25 ▾

篩選資料列:

+ 選項

Store_Name	Sales	Txn_Date
Los Angeles	1500	2019-10-05
San Diego	250	2019-10-07
Los Angeles	300	2019-10-08



# SQL ORDER BY

- 到目前為止，我們已學到如何藉由 **SELECT** 及 **WHERE** 這兩個指令將資料由表格中抓出。不過我們尚未提到這些資料要如何排列。這其實是一個很重要的問題。事實上，我們經常需要能夠將抓出的資料做一個有系統的顯示。這可能是由小往大 (ascending) 或是由大往小 (descending)。在這種情況下，我們就可以運用 **ORDER BY** 這個指令來達到我們的目的。

**ORDER BY** 的語法如下：

```
SELECT "欄位名"
FROM "表格名"
[WHERE "條件"]
ORDER BY "欄位名" [ASC, DESC];
```

- [] 代表 **WHERE** 子句不是一定需要的。不過，如果 **WHERE** 子句存在的話，它是在 **ORDER BY** 子句之前。**ASC** 代表結果會以由小往大的順序列出，而 **DESC** 代表結果會以由大往小的順序列出。如果兩者皆沒有被寫出的話，那我們就會用 **ASC**。

我們可以照好幾個不同的欄位來排順序。在這個情況下，**ORDER BY** 子句的語法如下(假設有兩個欄位)：

```
ORDER BY "欄位一" [ASC, DESC], "欄位二" [ASC, DESC]
```

# An Example

```
SELECT Store_Name, Sales, Txn_Date FROM Store_Information ORDER BY Sales DESC
```

全部顯示

資料列數 : 25 ▾

篩選資料列: 搜尋此資料表

+ 選項

Store_Name	Sales	Txn_Date
Los Angeles	1500	2019-10-05
Boston	700	2019-10-08
Los Angeles	300	2019-10-08
San Diego	250	2019-10-07

# Database Design

# 為何正規化

- 正規化主要是對表格做分割的動作。
- 沒有正規化會造成：
  - 容易有**資料重覆儲存**的浪費情形
  - 資料在做**插入、刪除或更新**動作時產生**異常(Anomalies)**情形
- 正規化的目的：
  - 降低**資料重覆性(Data Redundancy)**
  - 避免產生**插入、刪除或更新可能的異常(Anomalies)**

## 公司對某供應商之採購資料檔

供應商代號	供應商名稱	聯絡電話	產品代號	產品名稱	單價	採購量
12345	連店	0800000XXX	p147	X檯燈	1000	10
			s201	30cm燈管	150	300
02314	台雞店	0800111△△△	s201	30cm燈管	160	100
			s199	插座	20	100
22138	廣答	0800123〇〇〇	u001	電鍋	750	20

- 更新產品代號s201為30cm直式燈管，因為資料不只一筆，可能有些資料未被更新到，而造成資料不一致。
- 插入代號為00133的供應商“滑碩”，但插入不允許!! 除非此供應商已賣給我們公司至少一項產品。
- 刪除u001電鍋的銷售記錄，亦會刪除供商“廣答”之資訊。

- 功能相依 (Functional Dependence; FD) 的概念在資料庫設計上，尤其是資料庫正規化，非常重要。
  - 在資料正規化的過程中，每個階段都是以不同相依性 (Dependence) 之類型做為分割表格的依據。

## ■ 功能相依性 (Functional Dependency, FD)

- 假如  $R(A_1, A_2, \dots, A_n)$  代表一個關聯，其中  $X$  與  $Y$  為關聯  $R$  中屬性的子集合，則  $Y$  功能相依於  $X$  可以利用符號寫成：

$Y \propto X$  ( $Y$  功能相依於  $X$ )

或

$X \rightarrow Y$  ( $X$  決定  $Y$ )

其中， $X$  為 **決定因素 (Determinant)** 或 **left-hand side**， $Y$  為 **相依因素 (Dependent)** 或 **right-hand side**

- 一個關聯表格中的數個屬性間之功能相依性可能如：

- 所在地可決定分公司電話區碼** (即: 所在地  $\rightarrow$  區碼)
- 地址所在之區域可決定郵遞區號**
- 員工編號可決定員工姓名** (即: 員工編號  $\rightarrow$  姓名)

□ Def：給定一個關聯R，R的屬性子集Y功能相依於R的屬性子集X，則：

- 若且唯若(if and only if) 無論何時R的兩個Tuples若有相同的X值時，則必有相同的Y值。
- 對於關聯R中的每個X值，均有唯一的Y值來對應。

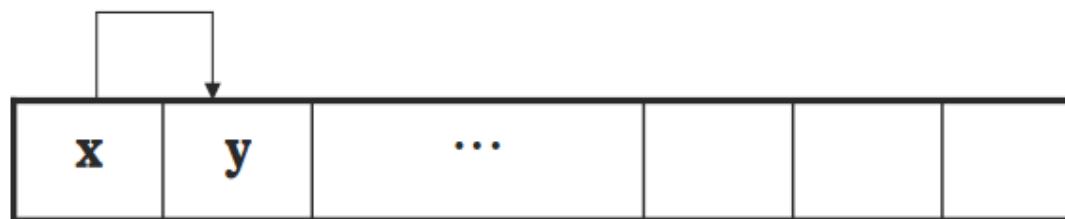
□ Ex: Ar功能相依於A1 ( $A1 \rightarrow Ar$ )

R		決定					
		A1	A2	...	Ar	...	An
t1	X	K	...	Y	...	Z	
	R	G		C		A	
t2	X	T	...	Y	...	L	
	K	H		W		X	

## □ 特性：

- 若在關聯R中 $X \rightarrow Y$ 成立，並不表示 $Y \rightarrow X$ 也成立
  - 功能相依性是一個**多對一**的關係
    - 多個不同的決定因素X可能決定出一個相依因素，反之則否。
  - 關聯R中的X若為主鍵(候選鍵)，則關聯R中所有其它屬性必功能相依於X
- ## □ 找出功能相依性是從事正規化的第一步。

## ■ 在一個關聯上，功能相依的表示方式：



屬性y功能相依於屬性x，或稱x決定了y ( $x \rightarrow y$ )

## ■ 範例：

供應商代號	供應商名稱	聯絡電話	產品代號	產品名稱	單價	採購量
12345	連店	0800000XXX	p147	X檯燈	1000	10
			s201	30cm燈管	150	300
02314	台雞店	0800111△△△	s201	30cm燈管	160	100
			s199	插座	20	100
22138	廣答	0800123〇〇〇	u001	電鍋	750	20

- {供應商代號} → {供應商名稱，聯絡電話}
- {產品代號} → {產品名稱}
- {供應商代號，產品代號} → {單價，採購量}



## Trivial(沒價值)及Non-Trivial(非沒價值)相依性

### ■ 沒價值 (Trivial)

- 若功能相依右邊之相依因素，為左邊決定因素的部份集合時，稱此功能相依為沒價值的。
- Ex: {供應商代號，產品代號}→供應商代號

### ■ 非沒價值 (Non-Trivial)

- 若功能相依右邊之相依因素，不為左邊決定因素的部份集合時，稱此功能相依為非沒價值的。
- Ex: {供應商代號，產品代號}→單價
- Non-Trivial的功能相依性，在從事正規化時才具有意義。

## 功能相依性的種類

- 功能相依性有幾種型式：

- 完全功能相依 (Full Functional Dependency)
- 部份功能相依 (Partial Functional Dependency)
- 遷移相依 (Transitive Dependency)

- 鍵值屬性(Key Attribute)

- 能構成主鍵或候選鍵的所有屬性。反之，則稱為非鍵值屬性(Non-Key Attribute)。
- 在實務上，鍵值屬性通常是指主鍵。

## ■ Key Attribute範例：

學號	姓名	系別	年級	生日	地址	身份証字號
001	張三	資管	2	3.18	台北	F11111111
002	李四	企管	3	3.19	台中	M22222222
003	王五	人管	4	3.20	台南	K33333333

- 鍵值屬性：學號，身份証字號
- 非鍵值屬性：姓名，系別，年級，生日，地址

主鍵(primary key)特性：

1. unique: 唯一性
2. one or multiple columns: 可為多數欄位組合
3. meaningful: 對該 table 有極相關且有意義

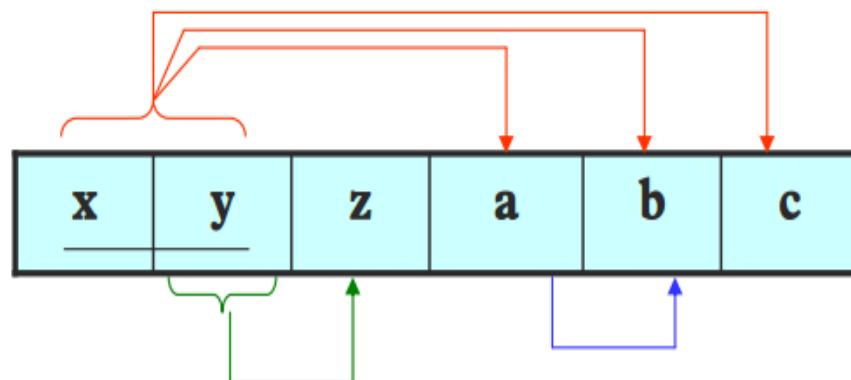
# An example

- What's the primary key for the following table?
- If so, how to setup the primary key?

Store_Name	Sales	Txn_Date
Los Angeles	1500	2019-10-05
San Diego	250	2019-10-07
Los Angeles	300	2019-10-08
Boston	700	2019-10-08

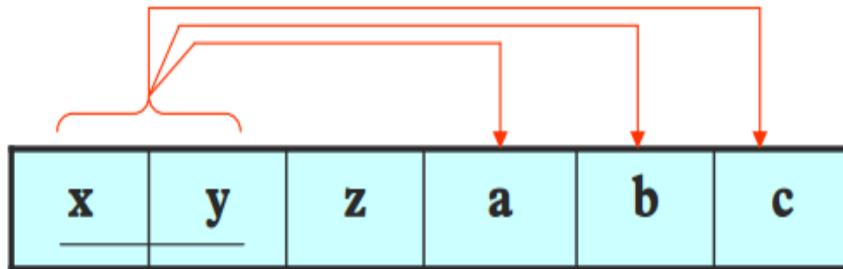
## 功能相依性範例

- 假設現在有下面的相依性範例：



## 完全功能相依 (Full Functional Dependency)

- 若主鍵是由**多個屬性**組成，且某非鍵值屬性依賴主鍵之**全部**而非部分時，則稱該非鍵值屬性“完全相依”於主鍵。
- 例如：



屬性a, b, c**完全相依**於由屬性x, y所共同構成的主鍵。

- 若主鍵**僅由一個屬性**所組成，則任一非鍵值屬性必“完全相依”於主鍵。

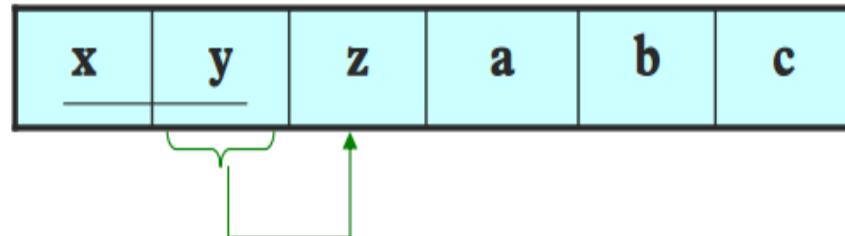
# An Example

- What's the primary key?

#	名稱	類型	編碼與排序	屬性	空值(Null)	預設值	備註	額外資訊	動作
1	ksu_std_id	varchar(6)	latin1_swedish_ci	否	無			 修改  刪除  更多	
2	ksu_std_name	varchar(20)	utf8_unicode_ci	否	無			 修改  刪除  更多	
3	ksu_std_age	int(2)		否	無			 修改  刪除  更多	
4	ksu_std_department	char(2)	latin1_swedish_ci	否	無			 修改  刪除  更多	
5	ksu_std_signin	int(4)		否	無			 修改  刪除  更多	
6	ksu_std_grade	int(1)		否	100			 修改  刪除  更多	

## 部份功能相依 (Partial Functional Dependency)

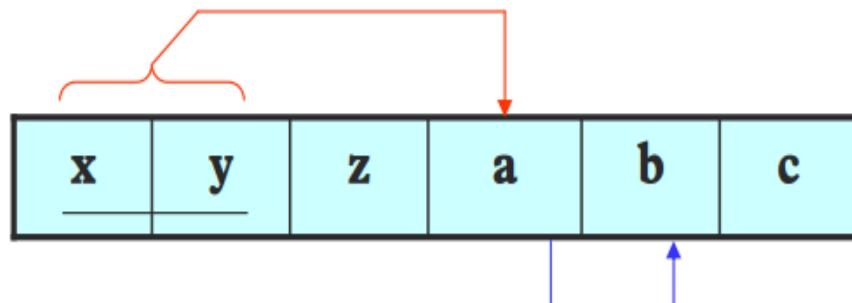
- 若主鍵是由**多個屬性**組成，而某非鍵值屬性是依賴主鍵之部分屬性時，則稱該非鍵值屬性“**部份相依**”於主鍵。
- 例如：



屬性z與屬性x, y所構成的主鍵部份相依。

# 遞移相依 (Transitive Dependency)

- 若存在一個非鍵值屬性 T，使得  $I \rightarrow T$  且  $T \rightarrow J$  的功能相依性均成立，則稱之 J 遷移相依於 I。
  - 即：I、T、J 三個屬性，形成 T 依賴 I ( $I \rightarrow T$ )，J 依賴 T ( $T \rightarrow J$ )，即稱為遞移相依。
- 例如：



$\{x,y\} \rightarrow b$  是透過非鍵值屬性 a 遷移的

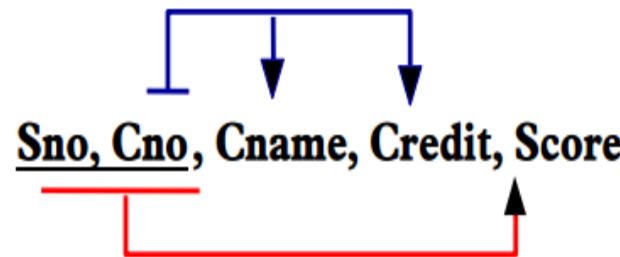
# 功能相依圖(FD Diagram)

- 學生選課資料的功能相依性(FD)為：

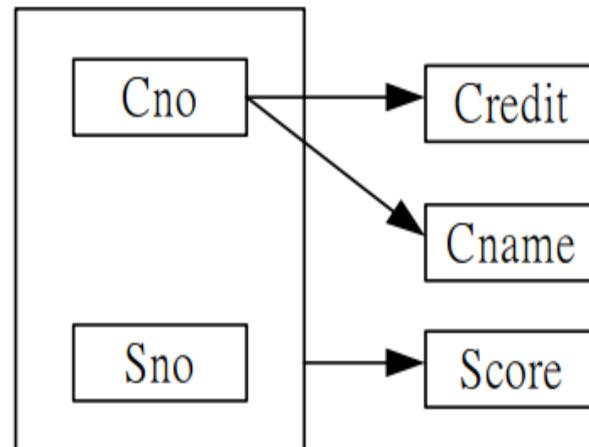
$(Sno, Cno) \rightarrow Score$

$Cno \rightarrow Cname$

$Cno \rightarrow Credit$



其功能相依圖如圖：



學生選課資料的功能相依圖

Q&A

# Foreign key (外來鍵)

SELECT \* FROM `student\_detail`

效能分析 [

全部顯示 | 資料列數： 25 ▾ 篩選資料列: 搜尋此資料表 依主鍵排序: [

+ 選項

	← T →	▼	std_id	std_name	std_city_id	std_cell	std_address	
<input type="checkbox"/>				4070E001	John Mike	A1	910971234	skyline Dr. no. 120
<input type="checkbox"/>				4070E002	Wom Try	A2	91029123	station Dr. no. 11
<input type="checkbox"/>				4070E003	Mike Fire	A1	912123456	Fire Street no 11.
<input type="checkbox"/>				4070E004	Dave Van	A3	901233333	Make Street no. 22

SELECT \* FROM `city\_detail`

全部顯示 | 資料列數： 25 ▾

+ 選項

std_city_id	std_city_name
A1	Taipei
A2	Tainan
A3	Kaoshiung
A4	Taichung

# An Example

Join

SELECT \* FROM `student\_detail`

效能分析

全部顯示 | 資料列數 :

25 ▾

篩選資料列:  搜尋此資料表

依主鍵排序:

+ 選項

	<input type="checkbox"/>	<input type="checkbox"/> 編輯	<input type="checkbox"/> 複製	<input type="checkbox"/> 刪除	std_id	std_name	std_city_id	std_cell	std_address
	<input type="checkbox"/>				4070E001	John Mike	A1	910971234	skyline Dr. no. 120
	<input type="checkbox"/>				4070E002	Wom Try	A2	91029123	station Dr. no. 11
	<input type="checkbox"/>				4070E003	Mike Fire	A1	912123456	Fire Street no 11.
	<input type="checkbox"/>				4070E004	Dave Van	A3	901233333	Make Street no. 22

SELECT \* FROM `city\_detail`

全部顯示 | 資料列數 :

25 ▾

+ 選項

std_city_id	std_city_name
A1	Taipei
A2	Tainan
A3	Kaoshiung
A4	Taichung

select std\_id, std\_name, std\_city\_name from student\_detail, city\_detail where student\_detail.std\_city\_id = city\_detail.std\_city\_id

輸出

效能分析 [行內編輯] [編輯] [SQL語句分析]

全部顯示 | 資料列數 :

25 ▾

篩選資料列:  搜尋此資料表

+ 選項

std_id	std_name	std_city_name
4070E001	John Mike	Taipei
4070E003	Mike Fire	Taipei
4070E002	Wom Try	Tainan
4070E004	Dave Van	Kaoshiung

# Extra talk

```
SELECT std_id as "學號", std_name as "姓名" FROM `student_detail`
```

學號	姓名
4070E001	John Mike
4070E002	Wom Try
4070E003	Mike Fire
4070E004	Dave Van



# 功能相依性的推導法則－阿姆斯壯定理 (Armstrong's Axioms)

- 目的：利用已知的功能相依性，透過一些性質，推導出以下可能項目：
  - 其它隱含(Implicit) 的功能相依性。
  - 一個關聯中的候選鍵或主鍵、
  - 最簡功能相依性(Irreducible Functional Dependence)
- 最簡功能相依性的特性：
  - ① 相依因素僅有一個屬性
  - ② 沒有多餘的決定因素
  - ③ 沒有多餘的功能相依性

- 假設A, B, C, D為關聯R的四個任意屬性子集，且AB表示A聯集B。則阿姆斯壯定理如下：
  - 反身性(Reflexivity)：若B是A的子集合，則 $A \rightarrow B$
  - 擴張性(Augmentation)：若 $A \rightarrow B$ ，則 $AC \rightarrow BC$
  - 遞移性(Transitivity)：若 $A \rightarrow B$ 且 $B \rightarrow C$ ，則 $A \rightarrow C$
  - 自身決定性(Self-determination)： $A \rightarrow A$
  - 分解性(Decomposition)：若 $A \rightarrow BC$ ，則 $A \rightarrow B$ 且 $A \rightarrow C$
  - 聯集性(Union)：若 $A \rightarrow B$ 且 $A \rightarrow C$ ，則 $A \rightarrow BC$
  - 合成性(Composition)：若 $A \rightarrow B$ 且 $C \rightarrow D$ ，則 $AC \rightarrow BD$
  - 虛擬遞移性(Pseudo-transitivity): 若 $A \rightarrow B$ 且 $BC \rightarrow D$ ，則 $AC \rightarrow D$

■ 例：某一關聯  $R=\{A, B, C, D\}$ ，且有以下功能相依：

$$\{C \rightarrow BD, B \rightarrow D, C \rightarrow B, BC \rightarrow D, CD \rightarrow A\}$$

請找出最簡功能相依性，並決定  $R$  的鍵值為何。

■ Ans:

① 相依因素僅有一個屬性

■ 透過分解性，將帶有一個以上屬性之相依因素分解掉。故可得到

$$\{C \rightarrow B, C \rightarrow D, B \rightarrow D, C \rightarrow B, BC \rightarrow D, CD \rightarrow A\}$$

■ 上述功能相依集合中， $C \rightarrow B$  有重覆，保留一個即可!!故可得到

$$\{C \rightarrow B, C \rightarrow D, B \rightarrow D, BC \rightarrow D, CD \rightarrow A\}$$

## ② 沒有多餘的決定因素

- $C \rightarrow D$  與  $CD \rightarrow A$  呈虛擬遞移，可得到  $C \rightarrow A$ 。 $\therefore$  上述功能相依可改為：

$$\{C \rightarrow B, C \rightarrow D, B \rightarrow D, BC \rightarrow D, \textcolor{red}{C \rightarrow A}\}$$

- $C \rightarrow B$  與  $BC \rightarrow D$  呈虛擬遞移，可得到  $C \rightarrow D$ 。 $\therefore$  上述功能相依可改為：

$$\{C \rightarrow B, C \rightarrow D, B \rightarrow D, \textcolor{red}{C \rightarrow D}, C \rightarrow A\}$$

- 上述功能相依集合中， $C \rightarrow D$  有重覆，保留一個即可!!故可得到

$$\{C \rightarrow B, C \rightarrow D, B \rightarrow D, C \rightarrow A\}$$

## ③ 去除多餘的功能相依

- $\because$  由遞移性得知， $C \rightarrow D$  可由  $C \rightarrow B$  與  $B \rightarrow D$  得到。 $\therefore$  上述功能相依可改為：

$$\{C \rightarrow B, B \rightarrow D, C \rightarrow A\}$$

- 由上述推導得知，最簡相依性是  $\{C \rightarrow B, B \rightarrow D, C \rightarrow A\}$ 。且因為屬性 C 可以直接或間接決定其它所有屬性， $\therefore C$  是鍵值。

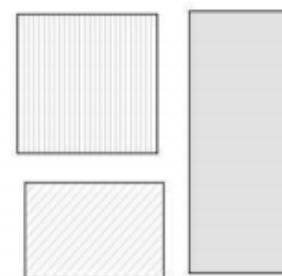
## ■ 正規化(Normalization)

- 正規化的理論首先由 E. F. Codd 於 1971 年提出，目的是用來設計「良好」的關聯式資料模式。
  - 原因：解決**資料重覆**及一些**異常現象**。
  - 方法：根據不同的相依性問題來**分割關聯**
  - 前題：**分割後的關聯不能有資訊遺失**的情況(無損失分解；Lossless decomposition)



單一的大資料表

(正規化的 4 個步驟)  
 $\rightarrow 1NF \rightarrow 2NF \rightarrow 3NF \rightarrow BCNF \rightarrow$



多個獨立但相關  
聯的小資料表

## 正規化的類型

- 基本上，正規化是以漸進的方式逐步地進行表格分割，每一個步驟都會滿足某項「正規化型式」的條件。
- 正規化的型式有以下幾種：
  - 低階正規化
    - 第一正規化型式 (1NF; First Normal Form)
    - 第二正規化型式 (2NF; Second Normal Form)
    - 第三正規化型式 (3NF; Third Normal Form)
    - BC正規化型式 (BCNF; Boyce-Codd Normal Form)
  - 高階正規化 (比較少用)
    - 第四正規化型式 (4NF; Forth Normal Form)
    - 第五正規化型式 (5NF; Fifth Normal Form)

## All Relations

(最寬鬆)

1NF

2NF

3NF

BCNF

4NF

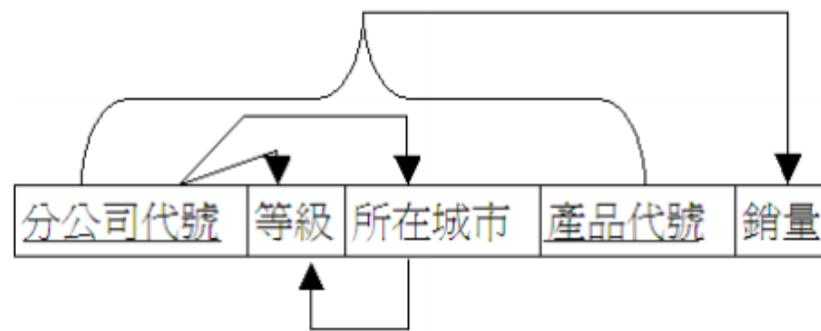
5NF

(最嚴謹)

- 理論上，表格經由正規化分割得愈細，愈可避免資料重覆及一些更新上的異常現象。然而，在實務上，分割太細的表格，可能會產生以下問題：
  - 在查詢資料時，可能會用到許多**合併 (Join)** 運算，此運算易造成查詢動作非常沒有效率。
  - 可能產生**有損分解**。
- 因此，在實作上，為保持“執行效率”與“避免異常”間的平衡，通常正規化進行到**3NF** (最多至**BCNF**) 即可。

# 非正規化 表格

- 假設有一個**非正規化表格**與它的屬性間之相依關係如下：



有 **非atomic value** 的  
屬性(欄位)

分公司代號	等級	所在城市	產品代號	銷量
S1	20	台北	P1	300
			P2	200
			P3	400
			P4	200
			P5	100
			P6	100
S2	10	高雄	P1	300
			P2	400
S3	10	高雄	P2	200
S4	20	台北	P2	200
			P4	300
			P5	400

# 第一正規化(First Normal Form , 1NF)

- 定義：一個關聯表為**第一正規化表格**，若且唯若關聯表中的每一個屬性其值皆為**基元值**(Atomic Value)。
- 作法：
  - 複合屬性：拆為簡單屬性
  - 多值屬性：分解成多個值組

分公司代號	等級	所在城市	產品代號	銷量
S1	20	台北	P1	300
			P2	200
			P3	400
			P4	200
			P5	100
			P6	100
S2	10	高雄	P1	300
			P2	400
S3	10	高雄	P2	200
S4	20	台北	P2	200
			P4	300
			P5	400

多值屬性

## ■ 問題：

- 產生**資料重複性**

■ 需要靠**2NF, 3NF, BCNF**  
來降低重複性。

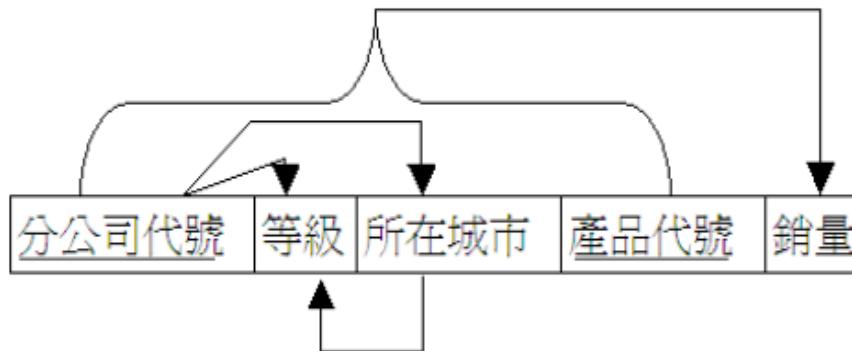
分公司代號	等級	所在城市	產品代號	銷量
S1	20	台北	P1	300
	20	台北	P2	200
	20	台北	P3	400
	20	台北	P4	200
	20	台北	P5	100
	20	台北	P6	100
S2	10	高雄	P1	300
	10	高雄	P2	400
S3	10	高雄	P2	200
S4	20	台北	P2	200
	20	台北	P4	300
	20	台北	P5	400

## 第一正規化型式所產生的異常 (Anomalies)

- 新增記錄時 (輸入資料時，必須等主鍵輸入才可進行)
  - 當有一新的分公司加入時，但未有任何交易經歷時，將造成產品代號為NULL。
- 更新資料時 (需要一起修改許多相關的值組)
  - 若單一分公司有多筆記錄在檔案中，此分公司若有資料異動，則必須更動多筆資料。如此不僅浪費空間，而且更新時亦浪費時間，也容易造成資料不一致的情況。
- 刪除記錄時 (刪除資料時，會把過多的資訊刪掉)
  - 若某分公司只有一筆交易經歷，若我們要刪除此分公司的交易資料時，則將連帶刪除該分公司資訊。

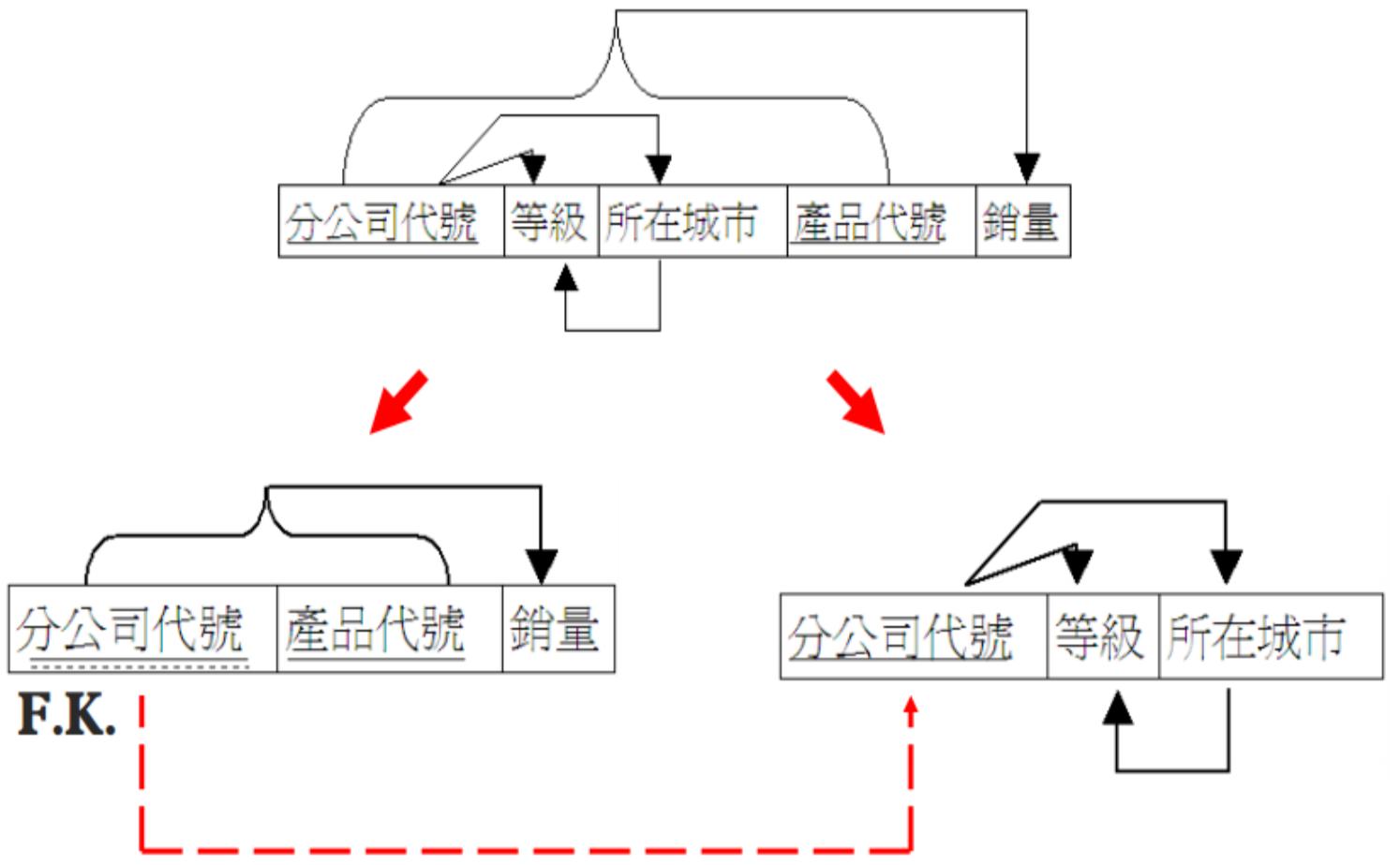
## 第二正規化(Second Normal Form，2NF)

- 定義：一個關聯表為第二正規化表格，若且唯若關聯表中，所有**非鍵值屬性皆完全功能相依於主鍵**



- 作法：
  - **拆解關聯**：將**部份相依於主鍵之非鍵值屬性取出，並與所屬之決定因素建立新的關聯，並將相依因素自原關聯刪除。**
  - **決定外來鍵**

## ■ 去除部份功能相依



分公司代號	等級	所在城市
S1	20	台北
S2	10	高雄
S3	10	高雄
S4	20	台北

分公司代號	產品代號	銷量
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

## 第二正規化型式所產生的異常 (Anomalies)

### ■ 新增記錄時 (輸入時必須等主鍵輸入才可進行)

- 當公司想評估一個新的點時(如：苗栗)，無法加入這個新的點的評估等級，除非已經在那裡設分公司了(∴在當地沒有實際設公司，沒有主鍵值可以對應!!)。

### ■ 更新資料時 (需要一起修改許多相關的值組)

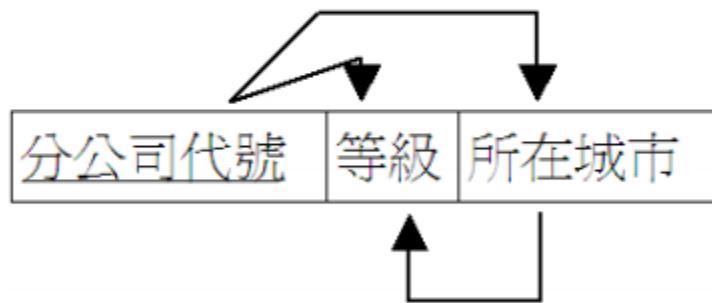
- 如果要把高雄的評等做更動，就要先將所有高雄的記錄給找出來(∵高雄的資料有很多筆，怕有些會沒改到，產生資料不一致)。

### ■ 刪除記錄時 (刪除時會把過多的資訊刪掉)

- 假設高雄分公司的評等資料只有一筆。如果要刪除高雄分公司的資料時，則必須連帶地刪除高雄的評估資料 (∵遞移相依)，如果以後還想在高雄設點，就要重新做評估了。

## 第三正規化(Third Normal Form , 3NF)

- 定義：一個關聯表為第三正規化表格，若且唯若該關聯表中，**不存在非鍵值屬性遞移相依於主鍵**
  - (即：非鍵值屬性間，不存在著功能相依性)。



- 作法：
  - **拆解關聯**：將遞移相依中，非鍵值之決定因素與相依因素取出，建立新的關聯，並將相依因素自原關聯中刪除。
  - **決定外來鍵**

## ■ 去除(非主鍵屬性)遞移功能相依。



分公司代號	所在城市
S1	台北
S2	高雄
S3	高雄
S4	台北

等級	所在城市
20	台北
10	高雄

分公司代號	產品代號	銷量
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

# 正規化過程



# SQL-SELECT – stage 2

<https://www.1keydata.com/tw/sql/sql.html>

# ORDER BY

- 我們經常需要能夠將抓出的資料做一個有系統的顯示。這可能是由小往大 (ascending) 或是由大往小 (descending)。在這種情況下，我們就可以運用 **ORDER BY** 這個指令來達到我們的目的。**ORDER BY** 的語法如下：

```
SELECT "欄位名"  
FROM "表格名"  
[WHERE "條件"]  
ORDER BY "欄位名" [ASC, DESC];
```

- [] 代表 **WHERE** 子句不是一定需要的。不過，如果 **WHERE** 子句存在的話，它是在 **ORDER BY** 子句之前。**ASC** 代表結果會以由小往大的順序列出，而 **DESC** 代表結果會以由大往小的順序列出。如果兩者皆沒有被寫出的話，那我們就會用 **ASC**。

我們可以照好幾個不同的欄位來排順序。在這個情況下，**ORDER BY** 子句的語法如下(假設有兩個欄位)：

```
ORDER BY "欄位一" [ASC, DESC], "欄位二" [ASC, DESC]
```

# An Example

```
SELECT Store_Name, Sales, Txn_Date FROM Store_Information ORDER BY Sales DESC
```



全部顯示

資料列數 :

25 ▾

篩選資料列: 搜尋此資料表

+ 選項

Store_Name	Sales	Txn_Date
Los Angeles	1500	2019-10-05
Boston	700	2019-10-08
Los Angeles	300	2019-10-08
San Diego	250	2019-10-07

```
SELECT Store_Name, Sales, Txn_Date FROM Store_Information ORDER BY 2 DESC
```



# SQL 函數

- **AVG** (平均)
- **COUNT** (計數)
- **MAX** (最大值)
- **MIN** (最小值)
- **SUM** (總合)

運用函數的語法是：

```
SELECT "函數名"("欄位名")
FROM "表格名";
```

# AVG( ) 函數 + Example

- 來計算平均值

```
SELECT AVG(Sales) FROM Store_Information
```

全部顯示 | 資料列數 : 25 ▾

+ 選項

**AVG(Sales)**

687.5000

+ 選項

Store_Name	Sales	Txn_Date
Los Angeles	1500	2019-10-05
San Diego	250	2019-10-07
Los Angeles	300	2019-10-08
Boston	700	2019-10-08



# An Example

+ 選項

Store_Name	Sales	Txn_Date
Los Angeles	1500	2019-10-05
San Diego	250	2019-10-07
Los Angeles	300	2019-10-08
Boston	700	2019-10-08

```
SELECT COUNT(Store_Name) FROM Store_Information WHERE (Store_Name IS NOT NULL) and Sales > 700
```

效能分析

+ 選項

COUNT(Store\_Name)

1



# COUNT( ) 函數

- 能夠數出在表格中有多少筆資料被選出來

```
SELECT COUNT("欄位名")
FROM "表格名";
```

+ 選項			
Store_Name	Sales	Txn_Date	
Los Angeles	1500	2019-10-05	
San Diego	250	2019-10-07	
Los Angeles	300	2019-10-08	
Boston	700	2019-10-08	

```
SELECT COUNT(Store_Name) FROM store_information
```

+ 選項

COUNT(Store\_Name)

4

```
SELECT COUNT(1) FROM store_information
```



# MAX( ) 函數 + Example

- 來計算一個欄位的最大值。

```
SELECT MAX ("欄位名")
FROM "表格名";
```

```
SELECT MAX(Sales) FROM Store_Information
```

全部顯示

資料列數：

25 ▾

+ 選項

**MAX(Sales)**

1500

+ 選項

Store_Name	Sales	Txn_Date
Los Angeles	1500	2019-10-05
San Diego	250	2019-10-07
Los Angeles	300	2019-10-08
Boston	700	2019-10-08



# MIN( ) 函數 + Example

- 來計算一個欄位的最小值

```
SELECT MIN ("欄位名")
FROM "表格名";
```

+ 選項		
Store_Name	Sales	Txn_Date
Los Angeles	1500	2019-10-05
San Diego	250	2019-10-07
Los Angeles	300	2019-10-08
Boston	700	2019-10-08

```
SELECT MIN(Sales) FROM Store_Information
```

全部顯示 | 資料列數 : 25 ▾

+ 選項  
**MIN(Sales)**  
250



# SUM( ) 函數 + Example

- 來計算一個欄位的總合

```
SELECT SUM ("欄位名")
FROM "表格名";
```

+ 選項		
Store_Name	Sales	Txn_Date
Los Angeles	1500	2019-10-05
San Diego	250	2019-10-07
Los Angeles	300	2019-10-08
Boston	700	2019-10-08

```
SELECT SUM(Sales) FROM Store_Information
```

全部顯示

| 資料列數：

25 ▾



+ 選項

**SUM(Sales)**

2750

# Group by

- 記得我們用 **SUM** 這個指令來算出所有的 Sales (營業額) 吧！如果我們的需求變成是要算出每一間店 (Store\_Name) 的營業額 (Sales)，那怎麼辦呢？在這個情況下，我們要做到兩件事：第一，我們對於 Store\_Name 及 Sales 這兩個欄位都要選出。第二，我們需要確認所有的 Sales 都要依照各個 Store\_Name 來分開算。這個語法為：

# An Example

```
SELECT Store_Name, SUM(Sales) FROM Store_Information GROUP BY Store_Name
```

全部顯示

資料列數 :

25 ▾

篩選資料列:

+ 選項

Store_Name	SUM(Sales)
Boston	700
Los Angeles	1800
San Diego	250



# Having

- 那我們如何對函數產生的值來設定條件呢？舉例來說，我們可能只需要知道哪些店的營業額有超過 \$1,500。在這個情況下，我們不能使用 **WHERE** 的指令。那要怎麼辦呢？很幸運地，SQL 有提供一個 **HAVING** 的指令，而我們就可以用這個指令來達到這個目標。**HAVING** 子句通常是在一個 SQL 句子的最後。一個含有 **HAVING** 子句的 SQL 並不一定要包含 **GROUP BY** 子句。**HAVING** 的語法如下：

```
SELECT "欄位1", SUM("欄位2")
FROM "表格名"
GROUP BY "欄位1"
HAVING (函數條件);
```

請讀者注意：如果被 **SELECT** 的只有函數欄，那就不需要 **GROUP BY** 子句。

# An example

- 若我們要找出 Sales 大於 1,500 的 Store\_Name，我們就鍵入，

+ 選項

Store_Name	Sales	Txn_Date
Los Angeles	1500	2019-10-05
San Diego	250	2019-10-07
Los Angeles	300	2019-10-08
Boston	700	2019-10-08

```
SELECT Store_Name, SUM(Sales) FROM Store_Information GROUP BY Store_Name HAVING SUM(Sales) > 1500
```

效能分析

全部顯示

資料列數：

25 ▾

篩選資料列:

+ 選項

Store\_Name | SUM(Sales)

Los Angeles 1800



# Discuss

+ 選項

Store_Name	Sales	Txn_Date
Los Angeles	1500	2019-10-05
San Diego	250	2019-10-07
Los Angeles	300	2019-10-08
Boston	700	2019-10-08

```
SELECT Store_Name, SUM(Sales) FROM Store_Information GROUP BY Store_Name
```

全部顯示

資料列數：

25 ▾

篩選資料列：搜尋此資料表



```
SELECT Store_Name, SUM(Sales) FROM Store_Information GROUP BY Store_Name HAVING SUM(Sales) > 1500
```

全部顯示

資料列數：

25 ▾

篩選資料列：搜尋此資料表



+ 選項

Store_Name	SUM(Sales)
Boston	700
Los Angeles	1800
San Diego	250

# How about?

- ```
SELECT Store_Name, SUM(Sales)
FROM Store_Information
where SUM(Sales) > 1500
GROUP BY Store_Name
```
- ```
SELECT Store_Name, SUM(Sales)
FROM Store_Information
where SUM(Sales) > 1500
```

# SQL-SELECT – stage 3

<https://www.1keydata.com/tw/sql/sql.html>

# Create a table

- 表格是資料庫中儲存資料的基本架構。在絕大部份的情況下，資料庫廠商不可能知道您需要如何儲存您的 資料，所以通常您會需要自己在資料庫中建立表格。雖然許多資料庫工具可以讓您在不需用到 SQL 的情況下 建立表格，不過由於表格是一個最基本的架構，我們決定包括 **CREATE TABLE** 的語法在這個網站中。

- 在我們跳入 **CREATE TABLE** 的語法之前，我們最好先對表格這個東西 有些多一點的瞭解。表格被分為欄位 (column) 及列位 (row)。每一列代表一筆資料，而每一欄代表一筆資料的一部份。舉例來說，如果我們有一個記載顧客資料的表格，那欄位就有可能包括姓、名、地址、城市、國家、生日 . . . 等等。當我們對表格下定義時，我們需要註明欄位的標題，以及那個欄位的資料種類。

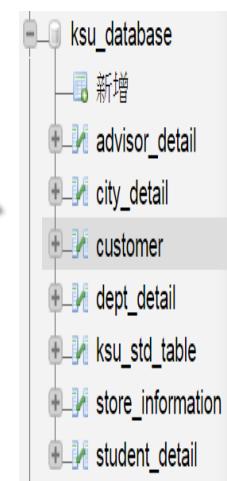
- 那，資料種類是什麼呢？資料可能是以許多不同的形式存在的。它可能是一個整數 (例如 1)、一個實數 (例如 0.55)、一個字串 (例如 'sql')、一個日期/時間 (例如 '2000-JAN-25 03:22:22')、甚至是 以二進法 (binary) 的狀態存在。當我們在對一個表格下定義時，我們需要對每一個欄位的資料種類下定義。(例如 '姓' 這個欄位的資料種類是 char(50)——代表這是一個 50 個字元的字串)。我們需要注意的一點是 不同的資料庫有不同的資料種類，所以在對表格做出定義之前最好先參考一下資料庫本身的說明。

**CREATE TABLE** 的語法是：

```
CREATE TABLE "表格名"  
("欄位 1" "欄位 1 資料種類",  
"欄位 2" "欄位 2 資料種類",  
...);
```

# Example

```
1 CREATE TABLE Customer  
2   (First_Name  char(50),  
3    Last_Name   char(50),  
4    Address     char(50),  
5    City        varchar(50),  
6    Country     char(25),  
7    Birth_Date  datetime);
```



First_Name	Last_Name	Address	City	Country	Birth_Date
------------	-----------	---------	------	---------	------------

查詢結果選項

新增檢視表

# ALTER TABLE

- 在表格被建立在資料庫中後，我們常常會發現，這個表格的結構需要有所改變。常見的改變如下：
  - 加一個欄位
  - 刪去一個欄位
  - 改變欄位名稱
  - 改變欄位的資料種類

以上列出的改變並不是所有可能的改變。ALTER TABLE 也可以被用來作其他的改變，例如改變主鍵定義。

**ALTER TABLE** 的語法如下：

```
ALTER TABLE "table_name"  
[改變方式];
```

[改變方式] 的詳細寫法會依我們想要達到的目標而有所不同。再以上列出的改變中，[改變方式] 如下：

- 加一個欄位：ADD "欄位 1" "欄位 1 資料種類"
- 刪去一個欄位：DROP "欄位 1"
- 改變欄位名稱：CHANGE "原本欄位名" "新欄位名" "新欄位名資料種類"
- 改變欄位的資料種類：MODIFY "欄位 1" "新資料種類"

# Example

第一，我們要加入一個叫做 "Gender" 的欄位。這可以用以下的指令達成：

```
ALTER TABLE Customer ADD Gender char(1);
```

這個指令執行後的表格架構是：

Customer 表格

欄位名稱	資料種類
First_Name	char(50)
Last_Name	char(50)
Address	char(50)
City	char(50)
Country	char(25)
Birth_Date	datetime
Gender	char(1)

# Example

接下來，我們要把 "Address" 欄位改名為 "Addr"。這可以用以下的指令達成：

```
ALTER TABLE Customer CHANGE Address Addr char(50);
```

這個指令執行後的表格架構是：

Customer 表格

欄位名稱	資料種類
First_Name	char(50)
Last_Name	char(50)
Addr	char(50)
City	char(50)
Country	char(25)
Birth_Date	datetime
Gender	char(1)

# Example

再來，我們要將 "Addr" 欄位的資料種類改為 `char(30)`。這可以用以下的指令達成：

```
ALTER TABLE Customer MODIFY Addr char(30);
```

這個指令執行後的表格架構是：

Customer 表格

欄位名稱	資料種類
First_Name	char(50)
Last_Name	char(50)
Addr	char(30)
City	char(50)
Country	char(25)
Birth_Date	datetime
Gender	char(1)

# Example

最後，我們要刪除 "Gender" 欄位。這可以用以下的指令達成：

```
ALTER TABLE Customer DROP Gender;
```

這個指令執行後的表格架構是：

Customer 表格

欄位名稱	資料種類
First_Name	char(50)
Last_Name	char(50)
Addr	char(30)
City	char(50)
Country	char(25)
Birth_Date	datetime

# INSERT

- 到目前為止，我們學到了將如何把資料由表格中取出。基本上，我們有兩種作法可以將資料輸入表格中內。一種是一次輸入一筆，另一種是一次輸入好幾筆。一次輸入一筆資料的語法如下：

```
INSERT INTO "表格名" ("欄位1", "欄位2", ...)  
VALUES ("值1", "值2", ...);
```

假設我們有一個架構如下的表格：

### Store\_Information 表格

欄位名稱	資料種類
Store_Name	char(50)
Sales	float
Txn_Date	datetime

而我們要加以下的這一筆資料進去這個表格：在 January 10, 1999，Los Angeles 店有 \$900 的營業額。我們就打入以下的 SQL 語句：

```
INSERT INTO Store_Information (Store_Name, Sales, Txn_Date)
VALUES ('Los Angeles', 900, 'Jan-10-1999');
```

- 第二種 **INSERT INTO** 能夠讓我們一次輸入多筆的資料。  
跟上面剛的例子不同的是，現在我們要用 **SELECT** 指令來  
指明要輸入表格的資料。如果您想說，這是不是說資料是  
從另一個表格來的，那您就想對了。

一次輸入多筆的資料的語法是：

```
INSERT INTO "表格1" ("欄位1", "欄位2", ...)
SELECT "欄位3", "欄位4", ...
FROM "表格2";
```

以上的語法是最基本的。這整句 SQL 也可以含有 **WHERE**、**GROUP BY**、及 **HAVING** 等子句，以及表格連接及別名等等。

舉例來說，若我們想要將 1998 年的營業額資料放入 **Sales\_Information** 表格，而我們知道資料的來源是可以由 **Sales\_Information** 表格取得的話，那我們就可以鍵入以下的 SQL：

```
INSERT INTO Store_Information (Store_Name, Sales, Txn_Date)
SELECT Store_Name, Sales, Txn_Date
FROM Sales_Information
WHERE Year(Txn_Date) = 1998;
```

在這裡，我用了 SQL Server 中的函數來由日期中找出年。不同的資料庫會有不同的語法。舉個例來說，在 Oracle 上，您將會使用 **WHERE TO\_CHAR (Txn\_Date,'yyyy') = 1998**。

# update

- 我們有時候可能會需要修改表格中的資料。在這個時候，我們就需要用到**UPDATE** 指令。這個指令的語法是：

```
UPDATE "表格名"  
SET "欄位1" = [新值]  
WHERE "條件";
```

```
UPDATE "表格"  
SET "欄位1" = [值1], "欄位2" = [值2]  
WHERE "條件";
```

- 最容易瞭解這個語法的方式是透過一個例子。假設我們有以下的表格：

Store\_Information 表格

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

我們發現說 Los Angeles 在 1999 年 1 月 8 號的營業額實際上是 \$500，而不是表格中所儲存的 \$300，因此我們用以下的 SQL 來修改那一筆資料：

```
UPDATE Store_Information  
SET Sales = 500  
WHERE Store_Name = 'Los Angeles'  
AND Txn_Date = 'Jan-08-1999';
```

現在表格的內容變成：

Store\_Information 表格

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	500	08-Jan-1999
Boston	700	08-Jan-1999

# Delete

- 在某些情況下，我們會需要直接由資料庫中去除一些資料。這可以藉由 **DELETE FROM** 指令來達成。它的語法是：

```
DELETE FROM "表格名"  
WHERE "條件";
```

以下我們用個實例說明。假設我們有以下這個表格：

Store\_Information 表格

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

而我們需要將有關 **Los Angeles** 的資料全部刪除。在這裡我們可以用以下的 **SQL** 來達到這個目的：

```
DELETE FROM Store_Information  
WHERE Store_Name = 'Los Angeles';
```

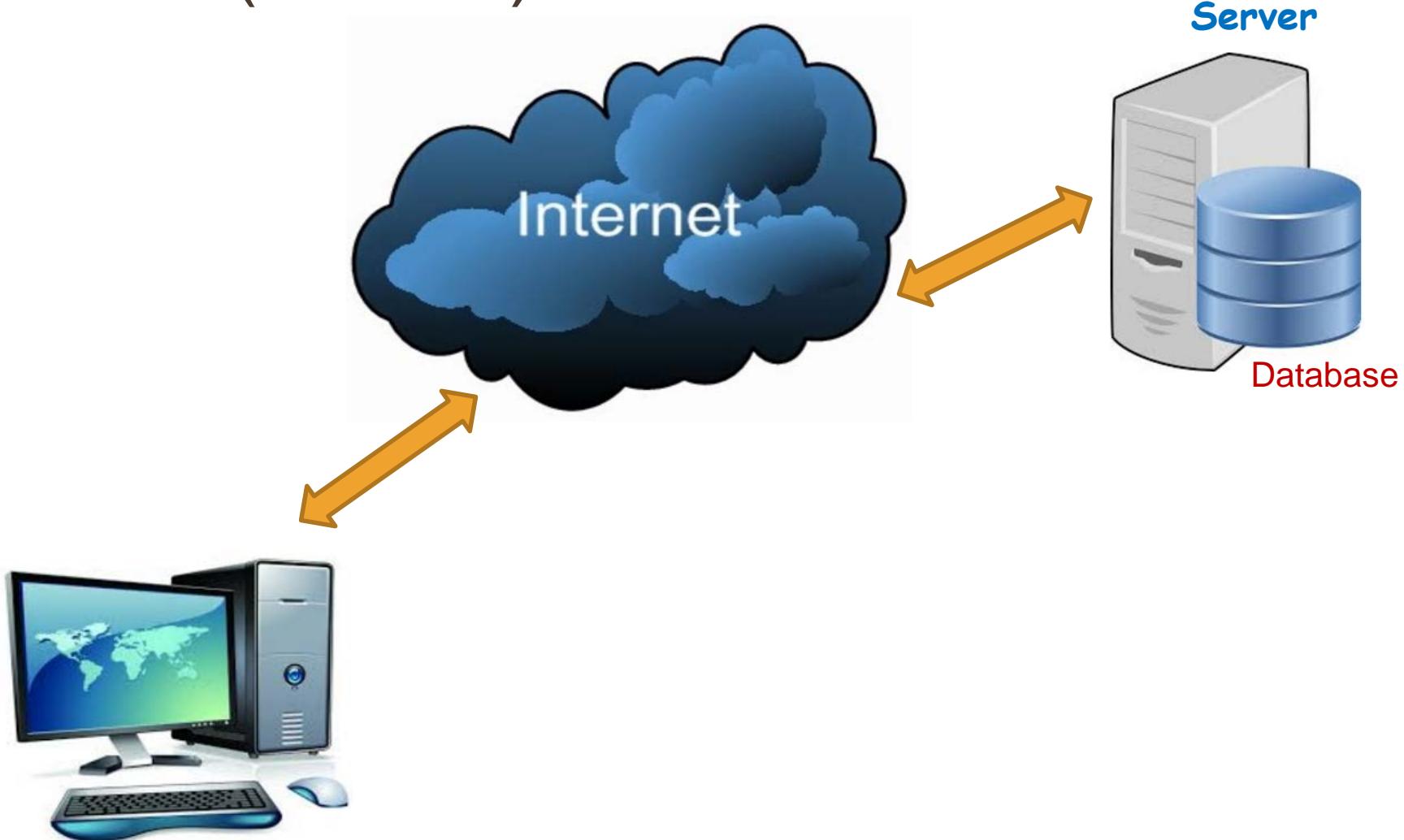
現在表格的內容變成：

Store\_Information 表格

Store_Name	Sales	Txn_Date
San Diego	250	Jan-07-1999
Boston	700	Jan-08-1999

# Connect to DB from php

# Server (服务器)

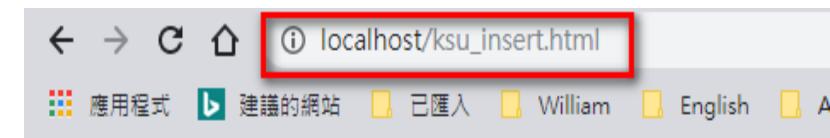
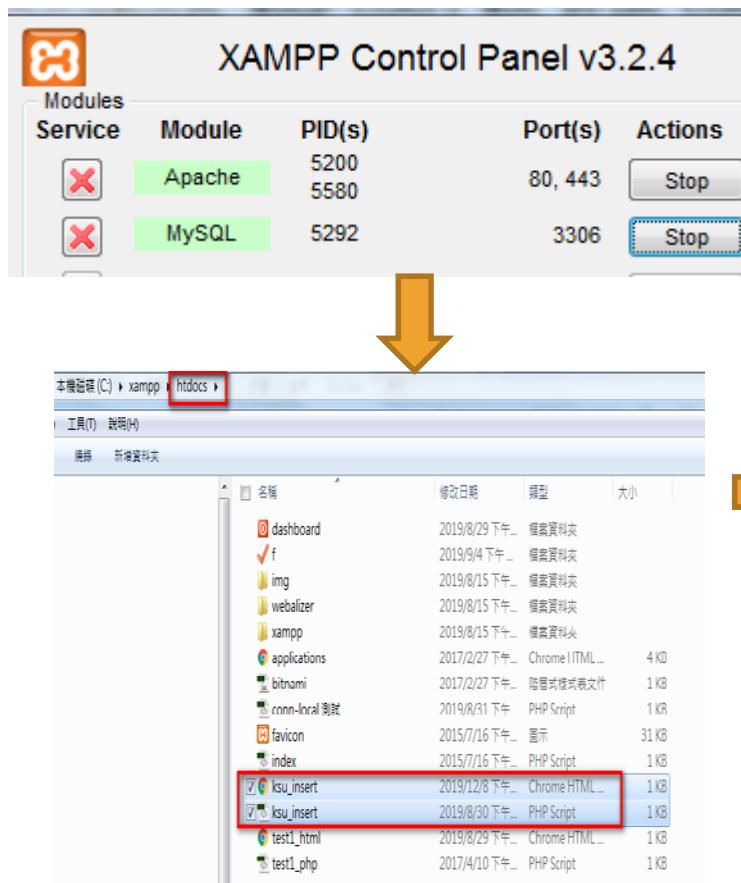


# Using html+php

- Html: for your frontend
- Php: for your backend
- SQL: for your DB

# Example

- The location for you to run your code (.html+.php)
- Run your code under XAMPP server



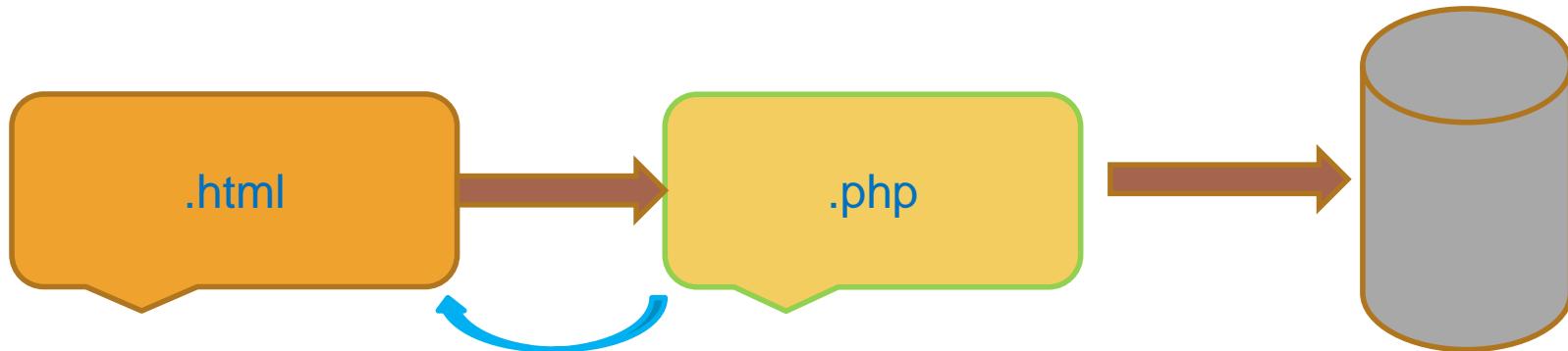
## SQL DML operation: Insert

姓名:

學號:

年齡:

# Run html+php



## SQL DML operation: Insert

姓名:

學號:

年齡:

↓ ↑

here! 1 record added



ksu_std_id	ksu_std_name	ksu_std_age	ksu_std_department
A12345	獨孤一劍	30	
aaaa	a1	0	
E012	無添	20	

# ksu\_insert.html

```
<!doctype html>
<html lang="zh_tw">
<head>
    <meta charset="utf-8">
    <title> Insert Option </title>
</head>
<body>
    <h3> SQL DML operation: Insert </h3>
    <form enctype="multipart/form-data" method="post" action="ksu_insert.php">
        姓名:<input type="text" name="name" size="10"/><br/>
        學號:<input type="text" name="id" size="10"/><br/>
        年齡:<input type="text" name="age" size="10"/><br/>
        |   <input type="submit" name="sub" value="新建"/>
    </form>

</body>
</html>
```

# php things

1.

把字符串"Hello world!" 中的字符"world" 替換為"Shanghai" :

```
<?php  
echo str_replace("world","Shanghai","Hello world!");  
?>
```

2.

PHP 的 `$_GET`, `$_POST`, 與 `$_REQUEST` 測試

最常用的 HTTP 方法是 GET 與 POST, 當我們提交表單時, 後端伺服器上的 PHP 程式可以分別用 `$_GET["param"]` 與 `$_POST["param"]` 來取得前端傳來的參數 param, 如果不知道前端會用哪一種方法, 則可用 `$_REQUEST["param"]`, 不管前端用 GET 還是 POST, `$_REQUEST` 都可以擷取到參數.

# ksu\_insert.php

```
<?php
$name=str_replace("","",$_REQUEST['name']);
$id=str_replace("","",$_REQUEST['id']);
$age=str_replace("","",$_REQUEST['age']);
print ("here!");

$db_host = "localhost";
$db_name = "ksu_database";
$db_table = "ksu_std_table";
$db_user = "root";
$db_password = "";
$conn = mysqli_connect($db_host, $db_user, $db_password);
if(empty($conn)){
    print mysqli_error ($conn);
    die ("無法對資料庫連線！");
    exit;
}
if(!mysqli_select_db( $conn, $db_name)){
    die("資料庫不存在!");
    exit;
}
mysqli_set_charset($conn,'utf8');

$insert_sql="INSERT INTO $db_table (ksu_std_id,ksu_std_name,ksu_std_age) values('$id','$name','$age')";
if (!mysqli_query($conn,$insert_sql))
{   die("無法新增資料!");
    exit;
}
echo "1 record added";
?>
<form enctype="multipart/form-data" method="post" action="ksu_insert.html">
    <input type="submit" name="sub" value="返回"/>
</form>
```

# Q&A