

課程：SQL 結構化查詢語言
講師：李昱賞

Agenda(一)

- 一. 資料庫
- 二. 關聯式資料庫
- 三. 資料庫術語
- 四. 資料庫常見物件
- 五. What is SQL ?

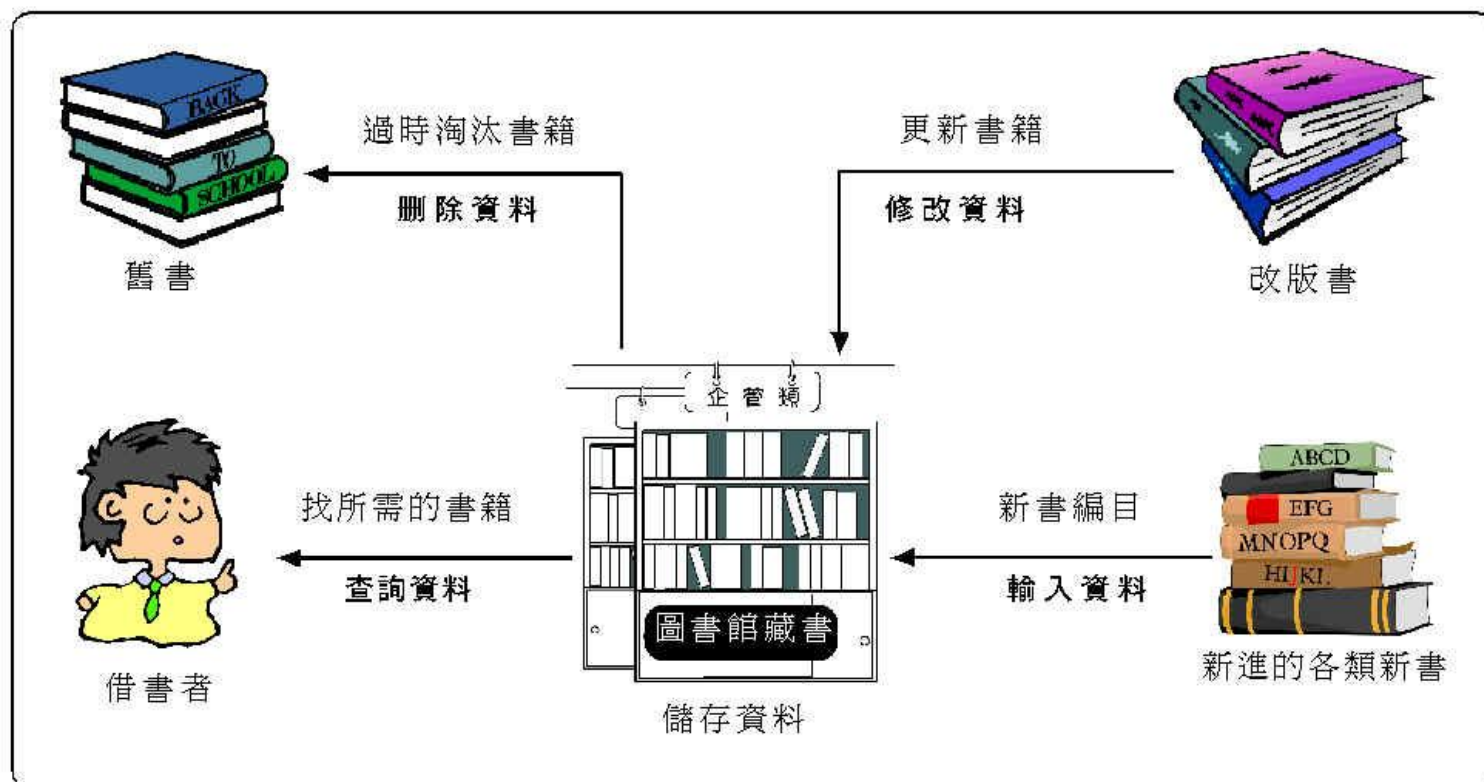
資料庫

- **資料庫**則是指一群有組織、有系統的資料集合，這集合是經由系統化設計組織的。
- **Oracle**、**MS SQL Server**、**MySQL**、**Postgre DB**、**Access**，都是一套管理資料庫的應用軟體，也是一般常常聽說的**資料庫管理系統** (DataBase Management System ; DBMS)

資料庫特性

- 用Structured Query Language (SQL)命令(command)去存取及修改在資料庫(Database)的物件。
- SQL 是American National Standards Institute (ANSI)所制訂的標準，用來操作在資料上的一種標準語言(standard language)。
- 使用SQL command 可對資料庫的物件在線上進行更改。
- 存放許多特有的物件(object)，且都有個名字，例如表格(table)。
- 所有的資料都是彼此獨立的(independence)。

生活中的資料庫－以圖書館為例

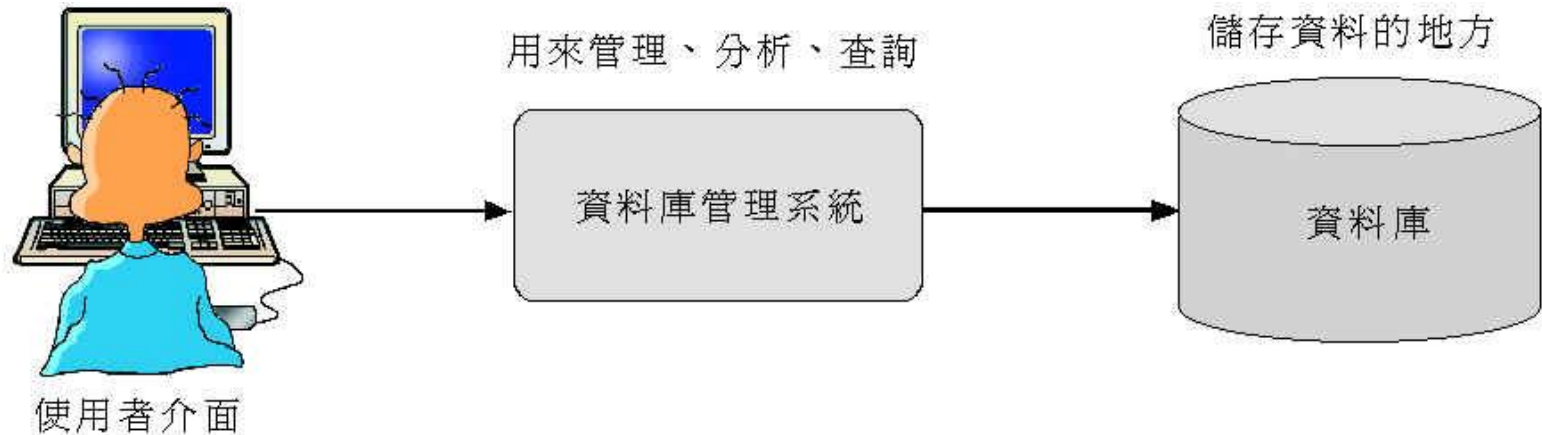


生活中的資料庫

1. 電話簿？
2. 畢業紀念冊？
3. 餐廳菜單？
4. ？？

資料庫與資料庫管理系統的關係

- 資料庫只是儲存與維護資料的地方，資料庫管理系統則用來操作及運用管理資料。



何謂關連式資料庫？

1. 最早由E.F. CODE Dr. 在1970年提出的資料庫的關聯式模型，作為日後發展管連是資料庫的基礎。
2. 儲存了許多的物件，而物件與物件間有關聯性存在。
3. 資料是以Value存在資料庫中，不以指標Point來指向Data。
4. 可操作(operation)與維護(maintain)在資料庫的資料。
5. 資料與應用程式間是各自獨立的。
6. 保有資料的完整性(integrity)與一致性(consistency)

何謂關連式資料庫？

- **關聯式資料庫 (Relational Database)** 是以 2 維的矩陣來儲存資料，而儲存在行、列裡的資料必會有所 "關聯"。

這整個是一個資料表

姓名	地址	電話
孫小小	台北市民生東路	(02)21219999
盧拉拉	台北市民族西路	(02)25444444
陳章章	台北市民權南路	(02)26669666

橫的一列稱為記錄

縱的一行稱為欄位

每一個都是一個資料項目

何謂關連式資料庫？

- 若我們要從以上的資料表尋找“盧拉拉”的地址，則是由橫向的『盧拉拉』與縱向的『地址』，交相關聯而得來：

姓名	地址	電話
孫小小	台北市民生東路	(02)21219999
盧拉拉	台北市民族西路	(02)25444444
陳章章	台北市民權南路	(02)26669666

何謂關連式資料庫？

資料表之間的關聯

- 關聯式資料庫裡的資料表通常也互有關聯，其好處是可以透過資料表的關聯，找到在另一個資料表中的資料：

訂單序號	日期	客戶編號	是否付款
1	2003/7/1	6	1
②	2003/7/1	③	1
3	2003/7/3	2	0

訂單資料表

客戶編號	客戶名稱	聯絡人	性別	地址
1	十全書店	陳圓圓	女	台北市
2	大發書店	陳季暄	女	台北市
③	好看書店	趙飛燕	女	台中市

客戶資料表

經由客戶編號欄的關聯，可知道
訂單序號 2 的客戶為好看書店

資料庫術語

- Table 表格
- Column 直欄(欄)
- Row 橫列(列)
- Field 欄位
- Unique Key 唯一鍵
- Primary Key 主鍵
- Foreign 外來鍵

Table 表格

- 表格是關聯式資料庫系統(RDBMS)中最基本的儲存結構(或稱物件)。
- 由一個以上的直欄(column)及零個或更多的橫列(row)所構成。
- 在表格(table)中每一筆資料(row)必須是唯一的(unique)。
- 在一個表格中，沒有兩筆以上的資料列是相同的。

Column 直欄

- 直欄(column)的特徵為相同的屬性值(value)。
- 同一直欄，其資料型態(data type)和長度(size)都相同。

Row 橫列(列)

- 橫列的特徵為一個以上的直欄所構成，有時亦稱為record。
- 規定在所有的橫列必須可以找到唯一的一筆橫列(row)。

Field 欄位

- 直欄(column)和橫列(row)所交叉的地方稱欄位(field)。
- 欄位(field)所存放的是該直欄(column)的屬性值(value)。
- 若該欄位(field)沒有資料，可以稱為該直欄(column)的屬性值(value)為空值(null)。
- Null是個很特殊的值，任何算式中若含有null值時，其結果均為null。

Unique Key 唯一鍵

- 可以是一個直欄(column)或一組直欄(a set of column 又稱複合鍵)所組成。
- 其鍵值在表格中是唯一(unique)的，也就是同一表格中沒有第二筆以上的鍵值是一樣的。
- 建立唯一鍵的同時，資料庫也會建立索引鍵值。
- 若沒有給指定名稱時，資料庫會主動命予 SQLn，其中n表時間戳記。

Primary Key 主鍵

- 為可在一個表格中能識別出每一筆資料 (row) 的鍵(Key)。
- 是由一個直欄(column)或一組直欄(a set of column 又稱複合鍵)所組成。
- 主鍵的值是唯一(unique)、不可重複性；並且不可為空值(not null)。
- 主鍵通常不可更改。
- 建立主鍵的同時，資料庫也會建立索引鍵值。

Foreign 外來鍵

- 是由一個直欄(column)或一組直欄(a set of column 又稱複合鍵)所組成。
- 外來鍵是參考同一表格或是另一表格中的主鍵或唯一鍵。
- 其值可為null。

What is SQL?

- SQL stands for **S**tructured **Q**uery **L**anguage
- SQL allows you to access a database
- SQL is an ANSI standard computer language
- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert new records in a database
- SQL can delete records from a database
- SQL can update records in a database
- SQL is easy to learn

Agenda(二)

- SQL 指令:
 - SQL 如何被用來儲存、讀取、以及處理資料庫之中的資料。
- 表格處理:
 - SQL 如何被用來處理資料庫中的表格。
- 進階 SQL:
 - 介紹 SQL 進階概念，以及如何用 SQL 來執行一些較複雜的運算。

資料查詢語言 DQL

- 資料查詢語言 **DQL**(**D**ata **Q**uery **L**anguage)

對資料庫進行資料查詢的動作，也是SQL裡支援語法最多寫法最具變化的語句，以下列出經常使用到的查詢語法：

1. 條件查詢：WHERE、AND、OR、IN、Between、Like
2. 群組、函數查詢：Group By、AVG、SUM、Having
3. 表格連接查詢：Join、INNER JOIN、LEFT JOIN、RIGHT JOIN
4. 進階查詢：Sub Query、UNION、UNION ALL、INTERSECT、MINUS、EXISTS、CASE WHEN、RANK、Hierarchical Queries

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)。

最基本的 SQL 架構：

SELECT "欄位名" **FROM** "表格名"

SQL SELECT

- 用以下的例子來看看實際上是怎麼用的…
- 假設有以下這個表格：

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

SQL SELECT

若要選出所有的店名 (store_name), 就鍵入:

```
SELECT store_name FROM Store_Information
```

結果:

store_name
Los Angeles
San Diego
Los Angeles
Boston

我們一次可以讀取好幾個欄位，也可以同時由好幾個表格中選資料。

SQL 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：

SELECT DISTINCT "欄位名" **FROM** "表格名"

SQL Distinct

舉例來說，若要在以下的表格，Store_Information，找出所有不同的店名時，

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

SQL Distinct

我們就鍵入：

```
SELECT DISTINCT store_name FROM  
Store_Information
```

結果：

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：

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

SQL WHERE

若我們要以以下的表格抓出營業額超過 \$1,000 的資料

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

SQL WHERE

我們就鍵入：

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

結果: store_name
Los Angeles

SQL And 、OR

- 在上一頁中，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。

SQL And 、 OR

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

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
San Francisco	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

SQL And 、 OR

我們就鍵入：

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

結果：

store_name
Los Angeles
San Francisco

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'**

SQL In

舉例來說，若我們要在 Store_Information 表格中找出所有含蓋 Los Angeles 或 San Diego 的資料

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

SQL In

我們就鍵入：

```
SELECT * FROM Store_Information
```

```
WHERE store_name IN ('Los Angeles', 'San Diego')
```

結果：

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999

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。

SQL Between

舉例來說，若我們要由 Store_Information 表格中找出所有介於 January 6, 1999 及 January 10, 1999 中的資料

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

SQL Between

我們就鍵入：

SELECT * FROM Store_Information

WHERE Date BETWEEN 'Jan-06-1999' AND 'Jan-10-1999'

請注意：在不同的資料庫中，日期的儲存法可能會有所不同。在這裡我們選擇了其中一種儲存法。同

結果：

store_name	Sales	Date
San Diego	\$250	Jan-07-1999
San Francisco	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

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}

SQL Like

- * **'A_Z'**: 所有以 **'A'** 起頭，另一個任何值的字元(中間底線表示任何字串)，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。

PS：若要找尋的資料包含萬用字元，就要使用 **ESCAPE** 關鍵字並指定適當的「跳脫字元」

SQL Like

我們將以上最後一個例子用在**Store_Information**表格上:

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

SQL Like

我們就鍵入：

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

結果：

store_name	Sales	Date
LOS AN GELES	\$1500	Jan-05-1999
SAN FR AN CISCO	\$300	Jan-08-1999
SAN DIEGO	\$250	Jan-07-1999

SQL 練習題(一)

- 1. 「且」 找出屬於西區的商店
- 2. 「且」 營業額大於**300**(包含**300**)
- 3. 「且」 商店名稱“L”開頭
- 4. 「或」 營業日介於**2018年3月至4月**
- 查詢結果：

	STORE_ID	STORE_NAME	SALES	STORE_DATE	FK_GEOGRAPHY_ID
1	1	Boston	700	2018-03-09 00:00:00 0000000000	1
2	2	Los Angeles	1500	2018-04-05 00:00:00 0000000000	2
3	4	Los Angeles	300	2018-02-07 00:00:00 0000000000	2



EXP_01_DATA.sql



EXP_01_Answer.sql

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]
```

SQL Order By

- [] 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**(預設由小到大排序)。
- 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**]

- 若我們對這兩個欄位都選擇由小往大的話，那這個子句就會造成結果是依據 "欄位一" 由小往大排。若有好幾筆資料 "欄位一" 的值相等，那這幾筆資料就依據 "欄位二" 由小往大排

SQL Order By

舉例來說，若我們要依照 **Sales** 欄位的由大往小列出 **Store_Information** 表格中的資料

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

SQL Order By

我們就鍵入：

```
SELECT store_name, Sales, Date FROM Store_Information  
ORDER BY Sales DESC
```

結果：

store_name	Sales	Date
LOS ANGELES	\$1500	Jan-05-1999
Boston	\$700	Jan-08-1999
SAN FRANCISCO	\$300	Jan-08-1999
SAN DIEGO	\$250	Jan-07-1999

在以上的例子中，我們用欄位名來指定排列順序的依據。除了欄位名外，我們也可以用欄位的順序(依據 **SQL** 句中的順序)。在 **SELECT** 後的第一個欄位為 **1**，第二個欄位為 **2**，以此類推。在上面這個例子中，我們用以下這句 **SQL** 可以達到完全一樣的效果：

```
SELECT store_name, Sales, Date FROM Store_Information  
ORDER BY 2 DESC
```

SQL Aggregate Functions 聚合函數

- 既然資料庫中有許多資料都是已數字的型態存在，一個很重要的用途就是要能夠對這些數字做一些運算，例如將它們總合起來，或是找出它們的平均值。SQL 有提供一些這一類的函數。它們是：
 - **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"

舉例來說，若我們由我們的範例表格中求出 **Sales** 欄位的總合

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

SQL 函數

我們就鍵入：

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

結果: **SUM(Sales)**
\$2750

\$2750 代表所有 Sales 欄位的總合: $\$1500 + \$250 + \$300 + \700 .

除了函數的運用外，SQL 也可以做簡單的數學運算，例如加(+)和減(-)。對於文字類的資料，SQL 也有好幾個文字處理方面的函數，例如文字相連 (concatenation)，文字修整 (trim)，以及子字串 (substring)。不同的資料庫對這些函數有不同的語法，所以最好是參考您所用資料庫的資訊，來確定在那個資料庫中，這些函數是如何被運用的。

SQL Count

- 在前一個指令有提到， **COUNT** 是函數之一。由於它的使用廣泛，我們在這裡特別提出來討論。基本上， **COUNT** 讓我們能夠數出在 表格中有多少筆資料被選出來。

它的語法是：

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

SQL Count

舉例來說，若我們要找出我們的範例表格中有幾筆
`store_name` 欄不是空白的資料時

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

SQL Count

我們就鍵入：

```
SELECT COUNT(store_name) FROM Store_Information  
WHERE store_name is not NULL
```

結果：
Count(store_name)
4

"is not NULL" 是 "這個欄位不是空白" 的意思。

SQL Count

COUNT 和 **DISTINCT** 經常被合起來使用，目的是找出表格中有多少筆不同的資料（至於這些資料實際上是什麼並不重要）。舉例來說，如果我們要找出我們的表格中有多少個不同的 **store_name**，我們就鍵入：

```
SELECT COUNT(DISTINCT store_name)
FROM Store_Information
```

結果: Count(store_name)
3

SQL Group By

- 我們現在回到函數上。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**) 的營業額總合 (**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。

GROUP 的語法是：**GROUP BY** 群組欄位

SELECT "欄位1", **SUM**("欄位2") **FROM** "表格名"

GROUP BY "欄位1"

SQL Group By

在範例上

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

我們就鍵入 ,

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

SQL Group By

結果:

<u>store_name</u>	<u>SUM(Sales)</u>
Los Angeles	\$1800
San Diego	\$250
Boston	\$700

當我們選不只一個欄位，且其中至少一個欄位有包含函數的運用時，我們就需要用到 **GROUP BY** 這個指令。在這個情況下，我們需要確定我們有 **GROUP BY** 所有其他的欄位。換句話說，除了有包括函數的欄位外，我們都需要將其放在 **GROUP BY** 的子句中。

ORACLE LISTAGG 函數

- 有時候我們只是想查尋 **GROUP BY** 之後的「其它欄位值」，那麼 Oracle **LISTAGG** 函數就可以滿足你的需求!

註：Oracle 11g R2 才支援

LISTAGG 語法如下：

LISTAGG (expr, 'delimiter') **WITHIN GROUP** (order_by_clause)

expr：用於字串連結之欄位(必要參數)

delimiter：字串連結之分隔符號(選擇參數)

order_by_clause：字串連結之順序(必要參數)

ORACLE LISTAGG 函數

我們就鍵入

```
SELECT STORE_NAME, SUM(SALES) SUM_SALES,  
LISTAGG(STORE_DATE , ',')  
WITHIN GROUP (ORDER BY STORE_DATE desc) AS STORE_DATE  
FROM STORE_INFORMATION  
GROUP BY STORE_NAME
```

結果:

STORE_NAME	SUM_SALES	STORE_DATE
Albany, Crossgates	2500	2016-06-15 00:00:00
Boston	700	1999-01-09 00:00:00
Buffalo, Walden Galleria	3000	2016-08-10 00:00:00
Los Angeles	1800	1999-01-08 00:00:00,1999-01-05 00:00:00
San Diego	250	1999-01-07 00:00:00

MySQL GROUP CONCAT 函數

MySQL 群組清單語法：

GROUP_CONCAT(FIELD **ORDER BY** FIELD **SEPARATOR** '/')

ORDER BY(排序) 及 SEPARATOR(指定分隔符號) 非必填

SELECT STORE_NAME, SUM(SALES) SUM_SALES,

GROUP_CONCAT(SALES **ORDER BY** SALES **SEPARATOR** '/') GROUP_SALES

FROM STORE_INFORMATION

GROUP BY STORE_NAME

STORE_NAME	SUM_SALES	GROUP_SALES
Albany, Crossgates	2500	2500
Boston	3700	1500/2200
Buffalo, Walden Galleria	3000	3000
Los Angeles	3300	300/1400/1600
San Diego	750	250/500

SQL Having

- 那我們如何對函數產生的值來設定條件查尋呢？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 的語法是：

SELECT "欄位1", **SUM**("欄位2") **FROM** "表格名"

GROUP BY "欄位1" **HAVING** (函數條件)

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

在我們 Store_Information 表格這個例子中,

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

SQL Having

若我們要找出 Sales 大於 \$1,500 的 store_name ,
我們就鍵入：

```
SELECT store_name, SUM(sales)
FROM Store_Information GROUP BY store_name
HAVING SUM(sales) > 1500
```

結果:	<u>store_name</u>	<u>SUM(Sales)</u>
	Los Angeles	\$1800

SQL Alias

- 接下來，我們討論 **alias (別名)** 在 **SQL** 上的用處。最常用到的別名有兩種：**欄位別名**及**表格別名**。簡單地來說，欄位別名的目的是為了讓 **SQL** 產生的結果易讀。在之前的例子中，每當我們有營業額總合時，欄位名都是 **SUM(sales)**。雖然在這個情況下沒有什麼問題，可是如果這個欄位不是一個簡單的總合，而是一個複雜的計算，那欄位名就沒有這麼易懂了。若我們用欄位別名的話，就可以確認結果中的欄位名是簡單易懂的。第二種別名是**表格別名**。要給一個表格取一個別名，只要在 **FROM** 子句中的表格名後空一格，然後再列出要用的表格別名就可以了。這在我們要用 **SQL** 由數個不同的表格中獲取資料時是很方便的。這一點我們在之後談到連接 (**join**) 時會看到。

SQL Alias

先來看一下欄位別名和表格別名的語法：

```
SELECT "表格別名"."欄位1" "欄位別名"  
FROM "表格名" "表格別名"
```

基本上，這兩種別名都是放在它們要替代的物件後面，而它們中間由一個空白分開。我們繼續使用 Store_Information 這個表格來做例子：

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

SQL Alias

我們用跟 [SQL GROUP BY](#) 那一頁 一樣的例子。這裡的不同處是我們加上了欄位別名以及表格別名：

```
SELECT A1.store_name Store, SUM(A1.Sales) "Total Sales"
```

```
FROM Store_Information A1 GROUP BY A1.store_name
```

結果：

<u>Store</u>	<u>Total Sales</u>
Los Angeles	\$1800
San Diego	\$250
Boston	\$700

在結果中，資料本身沒有不同。不同的是欄位的標題。這是運用欄位別名的結果。在第二個欄位上，原本我們的標題是 "Sum(Sales)"，而現在我們有一個很清楚的 "Total Sales"。很明顯地，"Total Sales" 能夠比 "Sum(Sales)" 更精確地闡述這個欄位的含意。用表格別名的好處在 這裡並沒有顯現出來，不過這在 [下一頁](#) 就會很清楚了。

SQL 表格連接(Join)

- 現在我們介紹連接(join 利用where字句)的概念。要瞭解連接，我們需要用到許多我們之前已介紹過的指令。我們先假設我們有以下的兩個表格。

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

Geography 表格

region_name	store_name
East	Boston
East	New York
West	Los Angeles
West	San Diego

SQL 表格連接(Join)

而我們要知道每一區 (region_name) 的營業額 (sales)。**Geography** 這個表格告訴我們每一區有哪些店，而 **Store Information** 告訴我們每一個店的營業額。若我們要知道每一區的營業額，我們需要將這兩個不同表格中的資料串聯起來。當我們仔細瞭解這兩個表格後，我們會發現它們可經由一個相同的欄位，**store_name**，連接起來。我們先將 SQL 句列出，之後再討論每一個子句的意義：

```
SELECT A1.region_name REGION, SUM(A2.Sales) SALES
FROM Geography A1, Store_Information A2
WHERE A1.store_name = A2.store_name
GROUP BY A1.region_name
```

結果:

<u>REGION</u>	<u>SALES</u>
East	\$700
West	\$2050

SQL 表格連接(Join)

在第一行中，我們告訴 SQL 去選出兩個欄位：第一個欄位是 Geography 表格中的 region_name 欄位 (我們取了個別名叫做 REGION)；第二個欄位是 Store_Information 表格中的 sales 欄位 (別名為 SALES)。請注意在這裡我們有用到表格別名：Geography 表格的別名是 A1，Store_Information 表格的別名是 A2。若我們沒有用表格別名的話，第一行就會變成

```
SELECT Geography.region_name REGION,  
SUM(Store_Information.Sales) SALES
```

很明顯地，這就複雜多了。在這裡我們可以看到表格別名的功用：它能让 SQL 句容易被瞭解，尤其是這個 SQL 句含蓋好幾個不同的表格時。

接下來我們看第三行，就是 **WHERE** 子句。這是我們闡述連接條件的地方。在這裡，我們要確認 **Geography** 表格中 store_name 欄位的值與 **Store_Information** 表格中 store_name 欄位的值是相等的。這個 **WHERE** 子句是一個連接的靈魂人物，因為它的角色是確定兩個表格之間的連接是正確的。如果 **WHERE** 子句是錯誤的，我們就極可能得到一個笛卡兒連接(Cartesian join)。笛卡兒連接會造成我們得到所有兩個表格 每兩行之間所有可能的組合。在這個例子中，笛卡兒連接會讓我們得到 $4 \times 4 = 16$ 行的結果。

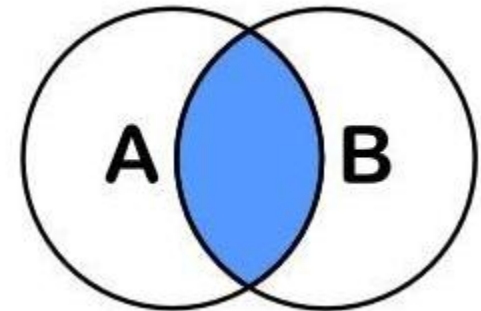
SQL 內部連接

- 內部連接 (**INNER JOIN**)
- 為等值連接，必需指定等值連接的條件，而查詢結果只會返回符合連接條件的資料。
- Syntax

```
SELECT table1.column1 , table2.column1 ...  
FROM table1 INNER JOIN table2  
ON table1.column1 = table2.column2
```

```
SELECT table1.column1 , table2.column1 ...  
FROM table1 INNER JOIN table2  
USING (table1.column1)
```

PS : table1與table2的 join 連接欄位必須同名稱



SQL 内部连接

- 範例：

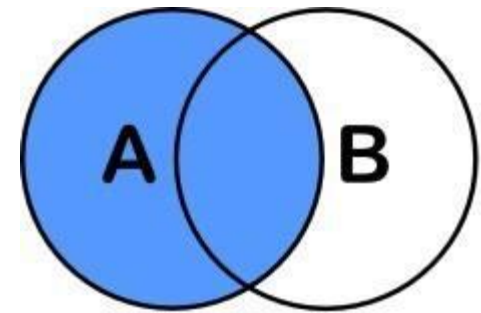
```
select * from geography a  
inner join store_information b  
on a.store_name = b.store_name
```

R2	REGION_NAME	R2	STORE_NAME	R2	STORE_NAME_1	R2	SALES	R2	S_DATE
1	West		Los Angeles		Los Angeles		1500		10-DEC-10
2	West		San Diego		San Diego		250		22-DEC-10
3	West		Los Angeles		Los Angeles		300		15-DEC-10
4	East		Boston		Boston		700		30-DEC-10

SQL (LEFT JOIN)

- 左外部連接(**LEFT JOIN** or **LEFT OUTER JOIN**)
- 查詢的 SQL 敘述句 **LEFT JOIN** 左側資料表的所有記錄都會加入到查詢結果中，即使右側資料表中的連接欄位沒有符合的值也一樣。
- Syntax

```
SELECT table1.colum1 , table2.column1 ...  
FROM table1  
LEFT JOIN table2  
ON table1.column1 = table2.column2
```



SQL (LEFT JOIN)

- 範例：

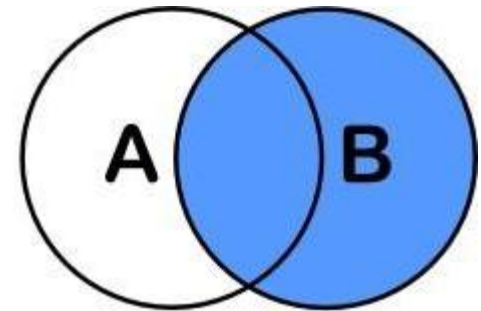
```
select * from geography a  
left join store_information b  
on a.store_name = b.store_name
```

	REGION_NAME	STORE_NAME	STORE_NAME_1	SALES	S_DATE
1	West	Los Angeles	Los Angeles	1500	10-DEC-10
2	West	San Diego	San Diego	250	22-DEC-10
3	West	Los Angeles	Los Angeles	300	15-DEC-10
4	East	Boston	Boston	700	30-DEC-10
5	East	New York	(null)	(null)	(null)

SQL (RIGHT JOIN)

- 右外部連接 (**RIGHT JOIN** or **RIGHT OUTER JOIN**)
- 查詢的 SQL 敘述句 **RIGHT JOIN** 右側資料表的所有記錄都會加入到查詢結果中，即使左側資料表中的連接欄位沒有符合的值也一樣。
- Syntax

```
SELECT table1.colum1 , table2.column1 ...  
FROM table1  
RIGHT JOIN table2  
ON table1.column1 = table2.column2
```



SQL (RIGHT JOIN)

- 範例：

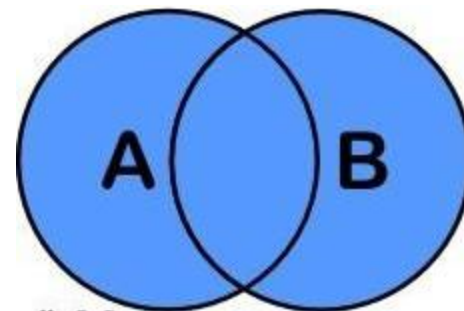
```
select * from geography a  
right join store_information b  
on a.store_name = b.store_name
```

	A2	REGION_NAME	A2	STORE_NAME	A2	STORE_NAME_1	A2	SALES	A2	S_DATE
1	East		Boston		Boston			700		30-DEC-10
2	West		Los Angeles		Los Angeles			300		15-DEC-10
3	West		Los Angeles		Los Angeles			1500		10-DEC-10
4	West		San Diego		San Diego			250		22-DEC-10

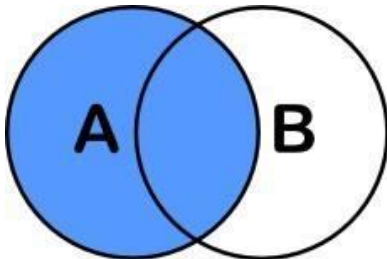
SQL (FULL JOIN)

- 全外部連接 (**FULL JOIN** or **FULL OUTER JOIN**)
- 查詢的 SQL 敘述句 **FULL JOIN** 左側資料表及右側資料表的所有記錄都會加入到查詢結果中。
- Syntax

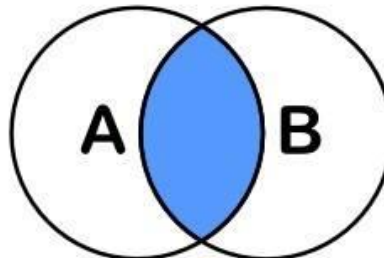
```
SELECT table1.column1 , table2.column1 ...  
FROM table1  
FULL JOIN table2  
ON table1.column1 = table2.column2
```



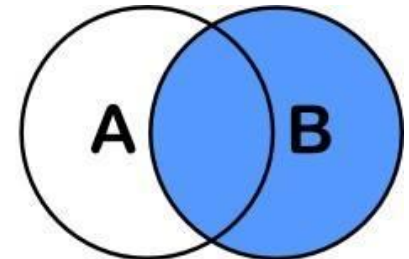
SQL JOINS



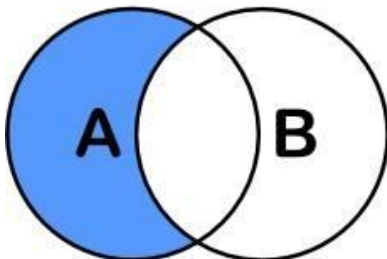
```
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key
```



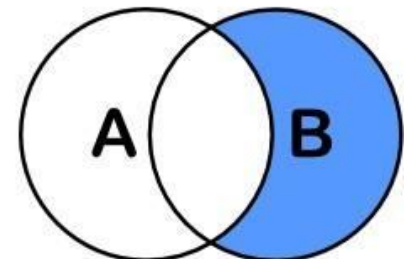
```
SELECT <auswahl>
FROM tabelleA A
INNER JOIN tabelleB B
ON A.key = B.key
```



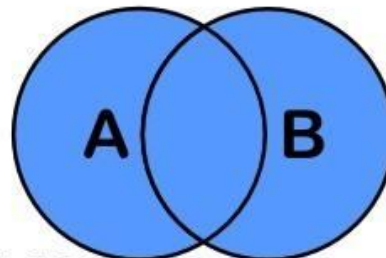
```
SELECT <auswahl>
FROM tabelleA A
RIGHT JOIN tabelleB B
ON A.key = B.key
```



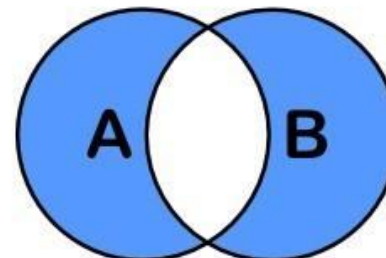
```
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key
WHERE B.key IS NULL
```



```
SELECT <auswahl>
FROM tabelleA A
RIGHT JOIN tabelleB B
ON A.key = B.key
WHERE A.key IS NULL
```



```
SELECT <auswahl>
FROM tabelleA A
FULL OUTER JOIN tabelleB B
ON A.key = B.key
```



```
SELECT <auswahl>
FROM tabelleA A
FULL OUTER JOIN tabelleB B
ON A.key = B.key
WHERE A.key IS NULL
OR B.key IS NULL
```

SQL 外部連接

- 外部連接的語法是依資料庫的不同而有所不同的。
- 舉例來說，在 **Oracle** 上，我們會在 **WHERE** 子句中要選出所有資料的那個表格之後加上一個 **"(+)"** 來代表說這個表格中的所有資料我們都要。

SQL 外部連接

假設我們有以下的兩個表格：

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

Geography 表格

region_name	store_name
East	Boston
East	New York
West	Los Angeles
West	San Diego

我們需要知道每一間店的營業額。如果我們用一個普通的連接，我們將會漏失掉 'New York' 這個店，因為它並不存在於 Store_Information 這個表格。

SQL 外部連接

所以，在這個情況下，我們需要用外部連接來串聯這兩個表格：

```
SELECT A1.store_name, SUM(A2.Sales) SALES  
FROM Geography A1, Store_Information A2  
WHERE A1.store_name = A2.store_name (+)  
GROUP BY A1.store_name
```

我們在這裡是使用了 Oracle 的外部連接語法。

結果:	<u>store_name</u>	<u>SALES</u>
	Boston	\$700
	New York	
	Los Angeles	\$1800
	San Diego	\$250

請注意：當第二個表格沒有相對的資料時，SQL會傳回NULL值。在這一個例子中，'New York'並不存在於 Store_Information表格，所以它的 "SALES" 欄位是 NULL。

SQL (CROSS JOIN)

- 交叉連接(cross join)
- 交叉連接為兩個資料表間的笛卡兒乘積 (Cartesian product) ，兩個資料表在結合時，不指定任何條件，即將兩個資料表中所有的可能排列組合出來

- Syntax

```
SELECT table1.colum1 , table2.column1 ...  
FROM table1 , table2
```

```
SELECT table1.colum1 , table2.column1 ...  
FROM table1 JOIN table2
```

SQL (CROSS JOIN)

- 範例：`select * from geography , store_information`

R2	REGION_NAME	R2	STORE_NAME	R2	STORE_NAME_1	R2	SALES	R2	S_DATE
1	East		Boston		Los Angeles		1500		10-DEC-10
2	East		New York		Los Angeles		1500		10-DEC-10
3	West		Los Angeles		Los Angeles		1500		10-DEC-10
4	West		San Diego		Los Angeles		1500		10-DEC-10
5	East		Boston		San Diego		250		22-DEC-10
6	East		New York		San Diego		250		22-DEC-10
7	West		Los Angeles		San Diego		250		22-DEC-10
8	West		San Diego		San Diego		250		22-DEC-10
9	East		Boston		Los Angeles		300		15-DEC-10
10	East		New York		Los Angeles		300		15-DEC-10
11	West		Los Angeles		Los Angeles		300		15-DEC-10
12	West		San Diego		Los Angeles		300		15-DEC-10
13	East		Boston		Boston		700		30-DEC-10
14	East		New York		Boston		700		30-DEC-10
15	West		Los Angeles		Boston		700		30-DEC-10
16	West		San Diego		Boston		700		30-DEC-10

SQL Concatenate 函數

- 有的時候，我們有需要將由不同欄位獲得的資料串連在一起。每一種資料庫都有提供方法來達到這個目的：
- MySQL: CONCAT()
- Oracle: CONCAT(), ||
- SQL Server: +

CONCAT() 的語法：**CONCAT(字串1, 字串2, 字串3, ...):**

將字串1、字串2、字串3，等字串連在一起。

請注意，Oracle的CONCAT()只允許兩個參數；換言之，一次只能將兩個字串串連起來。不過，在Oracle中，我們可以用'||來一次串連多個字串。

SQL Concatenate 函數

來看一個例子。假設我們有以下的表格：

Geography 表格

region_name	store_name
East	Boston
East	New York
West	Los Angeles
West	San Diego

例子1: MySQL/Oracle:

```
SELECT CONCAT(region_name,store_name) FROM Geography  
WHERE store_name = 'Boston';
```

結果: 'EastBoston'

SQL Concatenate 函數

例子2: Oracle:

```
SELECT region_name || ' ' || store_name  
FROM Geography  
WHERE store_name = 'Boston';
```

結果: 'East Boston'

例子3: **SQL Server:**

```
SELECT region_name + ' ' + store_name  
FROM Geography  
WHERE store_name = 'Boston';
```

結果: 'East Boston'

SQL SUBSTRING 函數

- SQL 中的 **substring** 函數是用來抓出一個欄位資料中的其中一部分。這個函數的名稱在不同的資料庫中不完全一樣：

MySQL: SUBSTR(), SUBSTRING()

Oracle: SUBSTR() 由1起始

SQL Server: SUBSTRING()

最常用到的方式如下 (在這裡我們用 SUBSTR() 為例)：

SUBSTR(str,pos): 由 <str> 中，選出所有從第 <pos> 位置開始的字元。請注意，這個語法不適用於 SQL Server 上。

SUBSTR(str,pos,len): 由 <str> 中的第 <pos> 位置開始，選出接下去的 <len> 個字元。

SQL SUBSTRING 函數

假設我們有 Geography 表格：

region_name	store_name
East	Boston
East	New York
West	Los Angeles
West	San Diego

例1: **SELECT SUBSTR(store_name, 3)** 結果: 's Angeles'
FROM Geography
WHERE store_name = 'Los Angeles';

例2: **SELECT SUBSTR(store_name,2,4)** 結果: 'an D'
FROM Geography
WHERE store_name = 'San Diego';

SQL Trim 函數

- SQL 中的 **TRIM** 函數是用來移除掉一個字串中的字頭或字尾。最常見的用途是移除字首或字尾的空白。這個函數在不同的資料庫中有不同的名稱：
 - MySQL: TRIM(), RTRIM(), LTRIM()
 - Oracle: RTRIM(), LTRIM()
 - SQL Server: RTRIM(), LTRIM()

SQL Trim 函數

各種 trim 函數的語法如下：

TRIM([[位置] [要移除的字串] **FROM**] 字串):

[位置] 的可能值為 **LEADING** (起頭), **TRAILING** (結尾), or **BOTH** (起頭及結尾)。這個函數將把 **[要移除的字串]** 從字串的起頭、結尾, 或是起頭及結尾移除。如果我們沒有列出 **[要移除的字串]** 是什麼的話, 那空白就會被移除。

LTRIM(字串):

將所有字串起頭的空白移除。

RTRIM(字串):

將所有字串結尾的空白移除。

SQL Trim 函數

例 1: **SELECT TRIM(' Sample ');**

結果: **'Sample'**

例 2: **SELECT LTRIM(' Sample ');**

結果: **'Sample '**

例 3: **SELECT RTRIM(' Sample ');**

結果: **' Sample'**

SQL 練習題(二)

- 計算和統計「個別商店」的以下三項資料：
「總合營業額」、「商店資料個數」、「平均營業額」

搜尋或排除條件如下：

排除「平均營業額」1000(含)以下的商店資料

排除「商店資料個數」1(含)個以下的商店資料

依照「平均營業額」由大至小排序

PS:使用別名語法簡化「表格名稱」及查詢結果「欄位名稱」



EXP_02_DATA.sql



EXP_02_03_Answer.sql

	STORE_NAME	SUM_SALES	COUNT_STORE	AVG_SALES
1	Boston	3700	2	1850
2	Los Angeles	3300	3	1100

SQL 練習題(三)

- 查詢各區域的營業額總計
資料結果依營業額總計由大到小排序
(不論該區域底下是否有所屬商店)

	REGION_NAME	SUM_SALES
1	West	6550
2	East	3700
3	North	0

- 查詢各區域的商店個數
資料結果依區域的商店個數由大至小排序
(依據商店名稱,不包含重覆的商店)
(不論該區域底下是否有所屬商店)

	REGION_NAME	STORE_COUNT
1	West	3
2	East	1
3	North	0

SQL 練習題(四)

- HR DB 資料查詢

查詢每個部門高於平均部門薪資的員工
(結果依部門平均薪資降冪排序)

SQL 所有擷取的資料列: 38 費時 0.015 秒						
	EMPLOYEE_ID	FIRST_NAME	SALARY	DEPARTMENT_ID	DEPARTMENT_NAME	AVG_SALARY
1	100	Steven	24000	90	Executive	19333
2	205	Shelley	12008	110	Accounting	10154
3	201	Michael	13000	20	Marketing	9500
4	150	Peter	10000	80	Sales	8955
5	174	Ellen	11000	80	Sales	8955
6	170	Taylor	9600	80	Sales	8955
7	169	Harrison	10000	80	Sales	8955
8	168	Lisa	11500	80	Sales	8955
9	163	Danielle	9500	80	Sales	8955
10	162	Clara	10500	80	Sales	8955
11	158	Allan	9000	80	Sales	8955
12	157	Patrick	9500	80	Sales	8955
13	156	Janette	10000	80	Sales	8955
14	152	Peter	9000	80	Sales	8955
15	151	David	9500	80	Sales	8955
16	149	Eleni	10500	80	Sales	8955
17	148	Gerald	11000	80	Sales	8955
18	147	Alberto	12000	80	Sales	8955
19	146	Karen	13500	80	Sales	8955
20	145	John	14000	80	Sales	8955
21	109	Daniel	9000	100	Finance	8601
22	108	Nancy	12008	100	Finance	8601
23	104	Bruce	6000	60	IT	5760
24	103	Alexander	9000	60	IT	5760
25	114	Den	11000	30	Purchasing	4150
26	141	Trenna	3500	50	Shipping	3475



EXP_02_04_Answer.sql

SQL 練習題(五)

- HR DB 資料查詢

查詢所有部門資訊如下：

- 1.所在地(國家、洲省、城市)
- 2.部門(部門編號、部門名稱)
- 3.部門管理者(員工編號、員工姓名、員工職稱)



EXP_02_05_Answer.sql

SQL 所有擷取的資料列: 27 費時 0.005 秒							
COUNTRY_ID	STATE_PROVINCE	CITY	DEPARTMENT_ID	DEPARTMENT_NAME	EMPLOYEE_ID	FIRST_NAME	JOB_TITLE
1 US	Washington	Seattle	10	Administration	200	Jennifer	Administration Assistant
2 CA	Ontario	Toronto	20	Marketing	201	Michael	Marketing Manager
3 US	Washington	Seattle	30	Purchasing	114	Den	Purchasing Manager
4 UK	(null)	London	40	Human Resources	203	Susan	Human Resources Representative
5 US	California	South San Francisco	50	Shipping	121	Adam	Stock Manager
6 US	Texas	Southlake	60	IT	103	Alexander	Programmer
7 DE	Bavaria	Munich	70	Public Relations	204	Hermann	Public Relations Representative
8 UK	Oxford	Oxford	80	Sales	145	John	Sales Manager
9 US	Washington	Seattle	90	Executive	100	Steven	President
10 US	Washington	Seattle	100	Finance	108	Nancy	Finance Manager
11 US	Washington	Seattle	110	Accounting	205	Shelley	Accounting Manager
12 US	Washington	Seattle	120	Treasury	100	Steven	President
13 US	Washington	Seattle	130	Corporate Tax	100	Steven	President
14 US	Washington	Seattle	140	Control And Credit	100	Steven	President
15 US	Washington	Seattle	150	Shareholder Services	100	Steven	President
16 US	Washington	Seattle	160	Benefits	100	Steven	President

資料定義語言DDL

- 資料定義語言DDL (Data Definition Language)

1. 建立資料表

主鍵、外來鍵、唯一鍵、NOT NULL、CHECK、
資料表欄位、View

2. ALTER 變更資料表

加一個欄位、刪去一個欄位、改變欄位名稱、改
變欄位型態

3. DROP 刪除資料表、Truncate 清空資料表

SQL 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 **TABLE** 的語法在這個網站中。
- 在我們跳入 **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 (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。

Column data type

- Column data type (欄位資料類型)

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')、或甚至是以二進法 (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。

Column data type

- 字元

資料類型	大小範圍	說明
CHAR	2000 (byte、char)	固定長度字串資料型別 (資料長度不足補空白)
VARCHAR	4000 (byte、char)	可變長度字串資料型別
VARCHAR2	4000 (byte、char)	可變長度字串資料型別 Oracle自行開發 資料可儲存NULL
CLOB	資料大小(4 GB)	大型文字資料 Character Large Object

PS：一個中文字佔 3 byte、一個英文字佔 1 byte
一個中文字、英文字皆是1 char

Column data type

- 數字

資料類型	數值範圍	說明
TINYINT	0 ~ 255	位元組
SMALLINT	-32,768 ~ 32,767 $-2^{15} \sim 2^{15}-1$	短整數
INTEGER	$-2^{31} \sim 2^{31}-1$	整數
BIGINT	$-2^{63} \sim 2^{63}-1$	長整數
REAL	$-3.40E + 38 \sim 3.40E + 38$	浮點數
FLOAT	$-1.79E + 308 \sim 1.79E + 308$	浮點數(可調整大小)
DOUBLE PRECISION	$-1.79769E + 308 \sim 1.79769E + 308$	倍精度浮點數
NUMBER (Oracle) NUMERIC (MS SQL)	NUMBER(P, S)	P : 數字位數 (1 ~ 38) S : 小數位數 (-84 ~ 127)

Column data type

- 日期時間

資料類型	說明
DATE	日期格式(最小單位/秒) 年、月、日、時、分、秒 日期範圍： 西元前 4712 年元月 1 日 西元 9999 年 12 月 31 日
TIMESTAMP	日期格式(最小單位/毫秒)
TIMESTAMP WITH LOCAL TIME ZONE	沒有世界地區時差
TIMESTAMP WITH TIME ZONE	時區及整個世界地區時差

- 進階資料型別

資料類型	資料大小	說明
BLOB	4 GB	大型二位元資料(影像檔、圖片檔) Binary Large Object

SQL Table

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", ..)
```

若我們要建立上面提過的顧客表格，就鍵入以下的 SQL：

```
CREATE TABLE customer  
(First_Name char(50),  
Last_Name char(50),  
Address char(50),  
City char(50),  
Country char(25),  
Birth_Date date)
```



TABLE_DDL.sql



Normalization

SQL Table

- 我們可以限制哪一些資料可以存入表格中。這些限制可以在表格初創時藉由 **TABLE** 語句來指定，或是之後藉由 **ALTER TABLE** 語句來指定。常見的限制有以下幾種：
 - **NOT NULL** 不能為空值
 - **UNIQUE** 唯一值
 - **CHECK**
 - 主鍵 (**Primary Key**) 有**UNIQUE**特性
 - 外來鍵 (**Foreign Key**)

以下對這幾種限制分別做個介紹：

SQL Table

NOT NULL

在沒有做出任何限制的情況下，一個欄位是允許有 NULL 值得。如果我們不允許一個欄位含有 NULL 值，我們就需要對那個欄位做出 NOT NULL 的指定。

舉例來說，在以下的語句中，

```
CREATE TABLE Customer  
(SID integer NOT NULL,  
Last_Name varchar (30) NOT NULL,  
First_Name varchar(30));
```

"SID" 和 "Last_Name" 這兩個欄位是不允許有 NULL 值，而 "First_Name" 這個欄位是可以有 NULL 值得。

SQL Table

UNIQUE

UNIQUE 限制是保證一個欄位中的所有資料都是不一樣的值。舉例來說，在以下的語句中，

```
CREATE TABLE Customer (SID integer Unique, Last_Name varchar (30),  
First_Name varchar(30));
```

"SID" 欄位不能有重複值存在，而 "Last_Name" 及 "First_Name" 這兩個欄位則是允許有重複值存在。請注意，一個被指定為主鍵的欄位也一定會含有 UNIQUE 的特性。相對來說，一個 UNIQUE 的欄位並不一定會是一個主鍵。

SQL Table

CHECK

CHECK 限制是保證一個欄位中的所有資料都是符合某些條件。舉例來說，在以下的語句中，

CREATE TABLE Customer

(SID integer CHECK (SID > 0),

Last_Name varchar (30), First_Name varchar(30));

"SID" 欄只能包含大於 0 的整數。請注意，CHECK 限制目前尚未被執行於 MySQL 資料庫上。[主鍵](#) and [外來鍵](#) 將於下頁中討論。

SQL 主鍵

- 主鍵 (Primary Key) 中的每一筆資料都是表格中的唯一值。換言之，它是用來獨一無二地確認一個表格中的每一行資料。主鍵可以是原本資料內的一個欄位，或是一個人造欄位 (與原本資料沒有關係的欄位)。主鍵可以包含一或多個欄位。當主鍵包含多個欄位時，稱為組合鍵 (Composite Key)。主鍵可以在建置新表格時設定 (運用 TABLE 語句)，或是以改變現有的表格架構方式設定 (運用 ALTER TABLE)。

SQL 主鍵

以下舉幾個在建置新表格時設定主鍵的方式：

MySQL:

```
CREATE TABLE Customer (SID integer, Last_Name varchar(30),  
First_Name varchar(30), PRIMARY KEY (SID));
```

Oracle:

```
CREATE TABLE Customer (SID integer PRIMARY KEY,  
Last_Name varchar(30), First_Name varchar(30));
```

SQL Server:

```
CREATE TABLE Customer (SID integer PRIMARY KEY,  
Last_Name varchar(30), First_Name varchar(30));
```

SQL 主鍵

以下則是以**ALTER**改變現有表格架構來設定主鍵的方式：

MySQL:

```
ALTER TABLE Customer ADD PRIMARY KEY (SID);
```

Oracle:

```
ALTER TABLE Customer ADD PRIMARY KEY (SID);
```

SQL Server:

```
ALTER TABLE Customer ADD PRIMARY KEY (SID);
```

請注意，在用ALTER TABLE語句來添加主鍵之前，我們需要確認被用來當做主鍵的欄位是設定為『NOT NULL』；也就是說，那個欄位一定不能沒有資料。

SQL 外來鍵

- 外來鍵(**Foreign Key**)是一個(或數個)指向另外一個表格主鍵的欄位。外來鍵的目的是確定資料的參考完整性(**referential integrity**)。換言之，只有被准許的資料值才會被存入資料庫內。

舉例來說，假設我們有兩個表格：一個 **CUSTOMER** 表格，裡面記錄了所有顧客的資料；另一個 **ORDERS** 表格，裡面記錄了所有顧客訂購的資料。在這裡的一個限制，就是所有的訂購資料中的顧客，都一定是要跟在 **CUSTOMER** 表格中存在。在這裡，我們就會在 **ORDERS** 表格中設定一個外來鍵，而這個外來鍵是指向 **CUSTOMER** 表格中的主鍵。這樣一來，我們就可以確定所有在 **ORDERS** 表格中的顧客都存在 **CUSTOMER** 表格中。換句話說**ORDERS**表格中，不能有任何顧客是不存在於**CUSTOMER**表格中的資料。

SQL 外來鍵

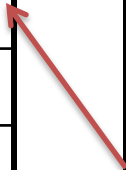
這兩個表格的結構將會是如下：

CUSTOMER 表格 **One**

欄位名	性質
SID	主鍵
Last_Name	
First_Name	

ORDERS 表格 **Many**

欄位名	性質
Order_ID	主鍵
Order_Date	
Customer_SID	外來鍵
Amount	



在以上的例子中，ORDERS 表格中的 customer_SID 欄位是一個指向 CUSTOMERS 表格中 SID 欄位的外來鍵。

SQL 外來鍵

以下列出幾個在建置 **ORDERS** 表格時指定外來鍵的方式：

MySQL:

```
CREATE TABLE ORDERS (Order_ID integer, Order_Date date,  
Customer_SID integer, Amount double, Primary Key (Order_ID),  
Foreign Key (Customer_SID) references CUSTOMER(SID));
```

Oracle:

```
CREATE TABLE ORDERS (Order_ID integer primary key, Order_Date date,  
Customer_SID integer references CUSTOMER(SID), Amount double);
```

SQL Server:

```
CREATE TABLE ORDERS (Order_ID integer primary key,  
Order_Date datetime, Customer_SID integer references CUSTOMER(SID),  
Amount double);
```

SQL 外來鍵

以下的例子則是藉著改變表格架構來指定外來鍵。這裡假設 **ORDERS** 表格已經被建置，而外來鍵尚未被指定：

MySQL:

```
ALTER TABLE ORDERS ADD FOREIGN KEY (customer_sid)  
REFERENCES CUSTOMER(sid);
```

Oracle:

```
ALTER ORDERS  
ADD CONSTRAINT CUS_SID  
FOREIGN KEY (customer_sid)  
REFERENCES CUSTOMER (sid);
```

SQL Server:

```
ALTER TABLE ORDERS ADD FOREIGN KEY (customer_sid)  
REFERENCES CUSTOMER(sid);
```

Table Relationships 資料表關係

- **One to One** 一對一關係

A資料表中的**單筆資料**記錄**同時只能對應**到B資料表的一**筆**記錄

例如一個洲只能有一位洲長

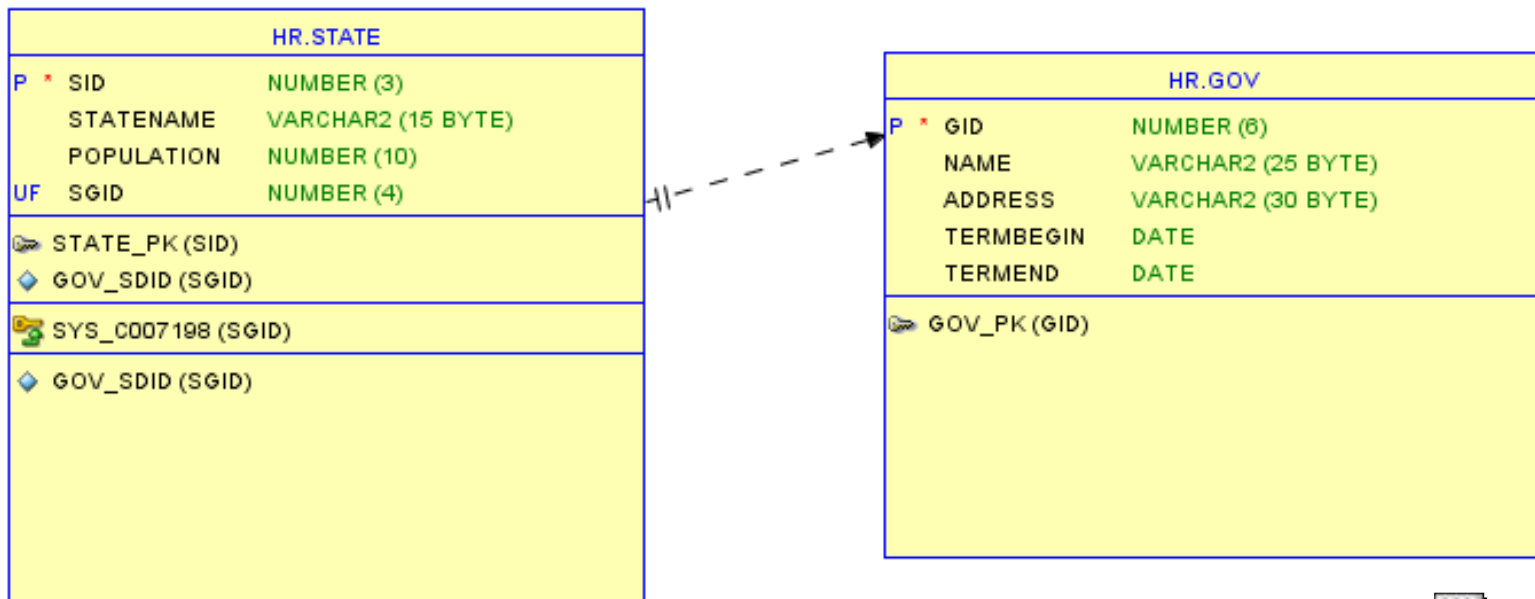


Table Relationships 資料表關係

- **One to Many** 一對多關係

A資料表中的**單筆資料**記錄同時可以對應到B資料表的**多筆記錄**

例如一間**供用商**可同時有**多個商品**，但一個商品只能屬於一間**供應商**

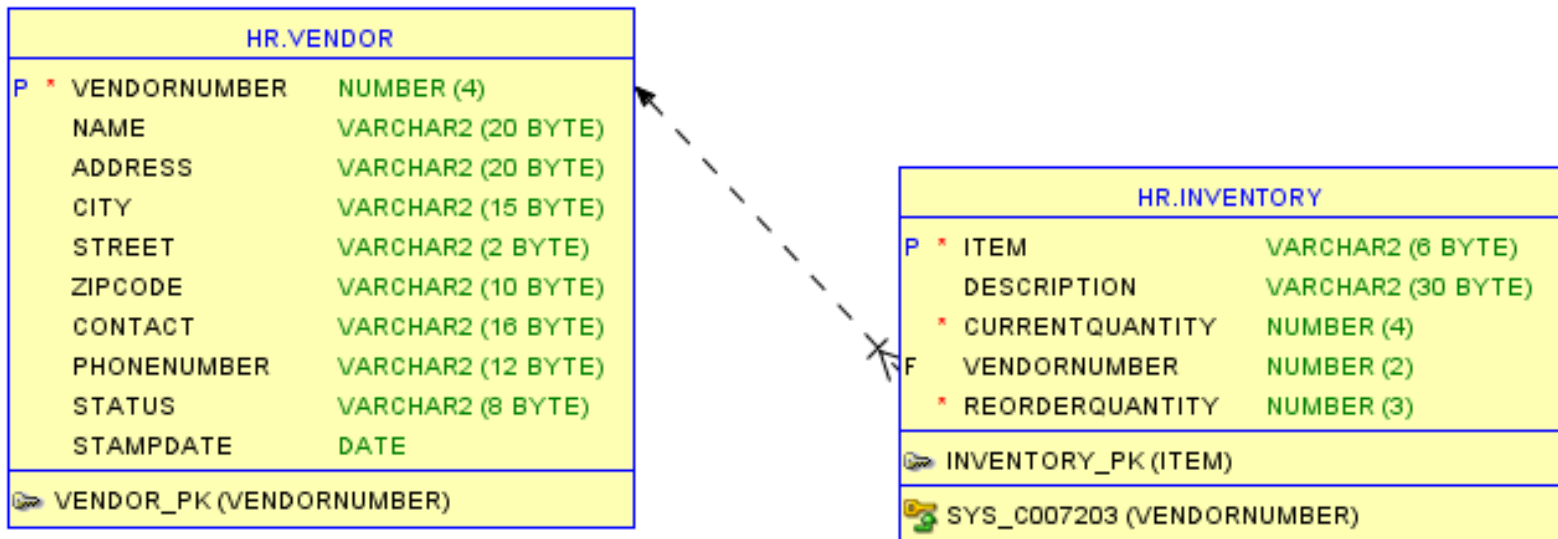
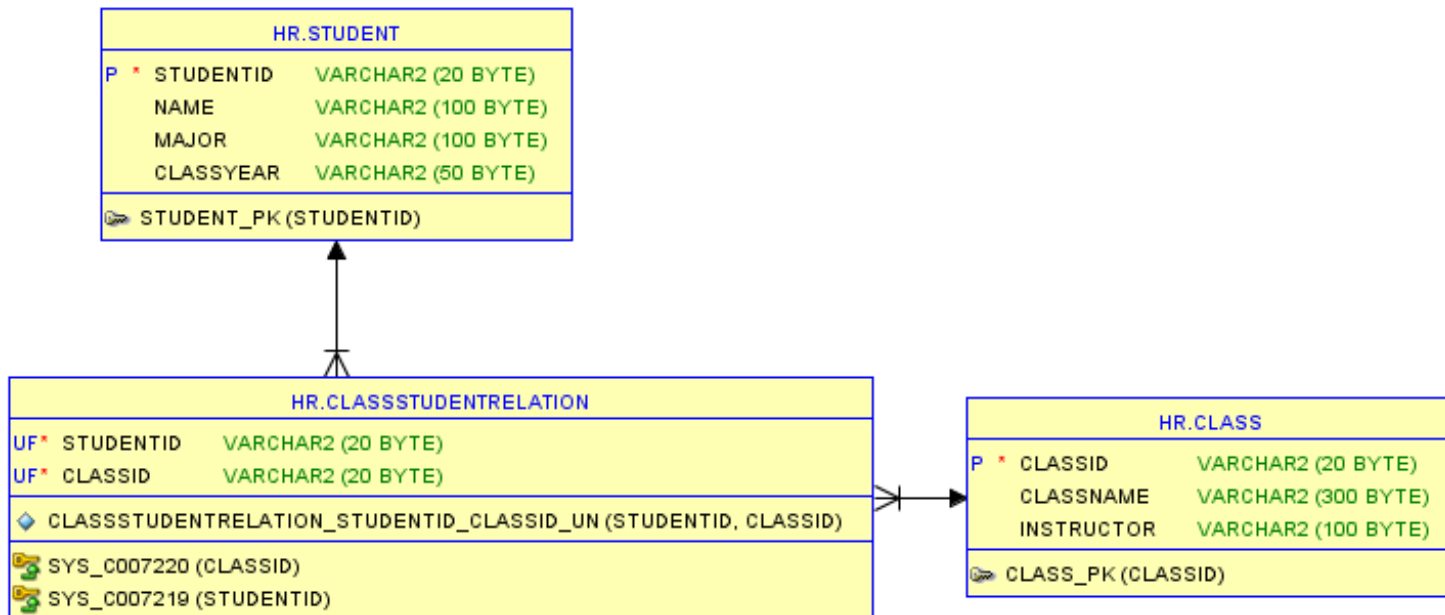


Table Relationships 資料表關係

- **Many to Many** 多對多關係

A資料表中的**多筆資料**記錄同時可以對應到B資料表的**多筆記錄**

例如**一位學生**可以選擇**多門課**，**一門課**也可以同時被**多位學生**選擇



SQL View

- 視觀表 (**Views**) 可以被當作是虛擬表格。它跟表格的不同是，表格中有實際儲存資料，而視觀表是建立在表格之上的一個架構，它本身並不實際儲存資料。

建立一個視觀表的語法如下：

```
CREATE VIEW "VIEW_NAME" AS
```

SQL View

來看一個例子。假設我們有以下的表格：

```
TABLE Customer  
First_Name char(50),  
Last_Name char(50),  
Address char(50),  
City char(50),  
Country char(25),  
Birth_Date date
```

若要在這個表格上建立一個包括 First_Name, Last_Name, 和 Country 這三個欄位的視觀表，我們就打入，

```
CREATE VIEW V_Customer  
AS SELECT First_Name, Last_Name, Country  
FROM Custome
```

SQL View

現在，我們就有一個叫做 *V_Customer* 的視觀表：
View V_Customer
(First_Name char(50),
Last_Name char(50),
Country char(25))

我們也可以用視觀表來連接兩個表格。在這個情況下，使用者就可以直接由一個視觀表中找出她要的資訊，而不需要由兩個不同的表格中去做一次連接的動作。假設有以下的兩個表格：

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

Geography 表格

region_name	store_name
East	Boston
East	New York
West	Los Angeles
West	San Diego

SQL View

我們就可以用以下的指令來建一個包括每個地區 (region) 銷售額 (sales) 的視觀表：

```
CREATE VIEW V_REGION_SALES
AS SELECT A1.region_name REGION, SUM(A2.Sales) SALES
FROM Geography A1, Store_Information A2
WHERE A1.store_name = A2.store_name
GROUP BY A1.region_name
```

這就給我們有一個名為 V_REGION_SALES 的視觀表。這個視觀表包含不同地區的銷售額。如果我們要從這個視觀表中獲取資料，我們就打入，

```
SELECT * FROM V_REGION_SALES
```

結果：

<u>REGION</u>	<u>SALES</u>
East	\$700
West	\$2050

SQL Index

- **索引 (Index)** 可以幫助我們從表格中快速地找到需要的資料。舉例來說，假設我們要在一本園藝書中找如何種植青椒的訊息。若這本書沒有索引的話，那我們是必須要從頭開始讀，直到我們找到有關種直青椒的地方為止。若這本書有索引的話，我們就可以先去索引找出種植青椒的資訊是在哪一頁，然後直接到那一頁去閱讀。很明顯地，運用索引是一種有效且省時的方式。
- 從資料庫表格中尋找資料也是同樣的原理。如果一個表格沒有索引的話，資料庫系統就需要將整個表格的資料讀出 (這個過程叫做'**table scan**'). 若有適當的索引存在，資料庫系統就可以先由這個索引去找出需要的資料是在表格的什麼地方，然後直接去那些地方抓資料。這樣子速度就快多了。

SQL Index

因此，在表格上建立索引是一件有利於系統效率的事。一個索引可以涵蓋一或多個欄位。建立索引的語法如下：

```
CREATE INDEX "INDEX_NAME" ON "TABLE_NAME"  
(COLUMN_NAME)
```

現在假設我們有以下這個表格，

```
CREATE TABLE Customer  
(First_Name char(50),  
Last_Name char(50),  
Address char(50),  
City char(50),  
Country char(25),  
Birth_Date date)
```

SQL Index

若我們要在 Last_Name 這個欄位上建一個索引，我們就打入以下的指令，

```
CREATE INDEX IDX_CUSTOMER_LAST_NAME  
on CUSTOMER (Last_Name)
```

我們要在 City 及 Country 這兩個欄位上建一個索引，就打入以下的指令，

```
CREATE INDEX IDX_CUSTOMER_LOCATION  
on CUSTOMER (City, Country)
```

索引的命名並沒有一個固定的方式。通常會用的方式是在名稱前加一個字首，例如 "IDX_"，來避免與資料庫中的其他物件混淆。另外，在索引名之內包括表格名及欄位名也是一個好的方式。

請讀者注意，每個資料庫會有它本身的 **INDEX** 語法，而不同資料庫的語法會有不同。因此，在下指令前，請先由資料庫使用手冊中確認正確的語法。

SQL 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：
 1. 加一個欄位
 2. 刪去一個欄位
 3. 改變欄位名稱
 4. 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]

SQL Alter Table

[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: **RENAME** "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"

以下我們用在 **TABLE** 一頁建出的 customer 表格來當作例子:

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	date

SQL Alter Table

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

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	date
Gender	char(1)

SQL Alter Table

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 rename column Address to Addr

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	date
Gender	char(1)

SQL Alter Table

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	date
Gender	char(1)

SQL Alter Table

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	date

SQL Drop Table

- 有時候我們會決定我們需要從資料庫中清除一個表格。事實上，如果我們不能這樣做的話，那將會是一個很大的問題，因為資料庫管理師（Database Administrator -- DBA）勢必無法對資料庫做有效率的管理。還好，SQL 有提供一個 **DROP TABLE** 的語法來讓我們清除表格。

DROP TABLE 的語法是：**DROP TABLE** "表格名"

我們如果要清除在上一頁中建立的顧客表格，我們就鍵入：

DROP TABLE customer.

SQL Truncate Table

- 有時候我們會需要清除一個表格中的所有資料。要達到這個目的，一種方式是我們在上一頁看到的 **DROP TABLE** 指令。不過這樣整個表格就消失，而無法再被用了。另一種方式就是運用 **TRUNCATE TABLE** 的指令。在這個指令之下，表格中的資料會完全消失，但表格本身會繼續存在。

TRUNCATE TABLE 的語法為：**TRUNCATE TABLE** "表格名"

所以，我們如果要清除在 SQL Table 那一頁建立的顧客表格之內的資料，我們就鍵入：

TRUNCATE TABLE customer

資料操作語言 DML

- 資料操作語言 **DML** (**D**ata **M**anipulation **L**anguage)
 1. **INSERT** 新增資料到資料表中
 2. **UPDATE** 更改資料表中的資料
 3. **DELETE** 刪除資料表中的資料

SQL Insert Into

- 到目前為止，我們學到了將如何把資料由表格中取出。但是這些資料是如果進入這些表格的呢？這就是這一頁 (**INSERT INTO**) 和下一頁 (**UPDATE**) 要討論的。
- 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。我們先來看一次輸入一筆的方式。

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", ...)
```

SQL Insert Into

假設我們有一個架構如下的表格：

Store_Information 表格

Column Name	Data Type
store_name	char(50)
Sales	float
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, Date)  
VALUES ('Los Angeles', 900, 'Jan-10-1999')
```

SQL Insert Into

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。

SQL Insert Into

舉例來說，若我們想要將 1998 年的營業額資料放入 Store_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, Date)
SELECT store_name, Sales, Date
FROM Sales_Information
WHERE Year(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，將會使用 **WHERE to_char(date,'yyyy')=1998**。

SQL 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}

最容易瞭解這個語法的方式是透過一個例子。

假設我們有 Store_Information 表格：

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

SQL UPDATE

我們發現說 Los Angeles 在 01/08/1999 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 Date = "Jan-08-1999"
```

現在Store_Information表格的內容變成：

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$500	Jan-08-1999
Boston	\$700	Jan-08-1999

SQL UPDATE

在這個例子中，只有一筆資料符合 **WHERE** 子句中的條件。如果有**多筆資料符合條件**的話，**每一筆符合條件的資料都會被修改**的。

我們也可以同時修改好幾個欄位。這語法如下：

UPDATE "表格"

SET "欄位1" = [值1], "欄位2" = [值2]

WHERE {條件}

SQL Delete From

- 在在某些情況下，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。

這個指令的語法是：

DELETE FROM "表格名" **WHERE** {條件}

以下我們用個實例說明。

假設我們有Store_Information表格：

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

SQL Delete From

而我們需要將有關 Los Angeles 的資料全部去除。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	Date
San Diego	\$250	Jan-07-1999
Boston	\$700	Jan-08-1999

資料控制語言 DCL

- 資料控制語言 **DCL** (**D**ata **C**ontrol **L**anguage)

1. **COMMIT** 資料提交(完成交易)

如進行資料異動操作後最後須執行交易提交**commit**的動作資料方可異動成功

交易隔離性：資料庫交易與交易之間彼此獨立，一個交易是看不到另一個交易所異動中的資料

2. **ROLLBACK** 資料回滾(交易取消)

可對交易異動中的資料在資料未提交**commit**前，進行**rollback**取消這次交易所所有的資料**DML**異動指令

進階SQL

在這一部分，我們將介紹
以下的 SQL 概念及關鍵字：

- SQL UNION 聯集(不包含重覆值)
- SQL UNION ALL 聯集(包含重覆值)
- SQL INTERSECT 交集
- SQL MINUS 排除(不包含重覆值)
- SQL SubQuery 子查詢
- SQL EXISTS 存在式關聯查詢
- SQL CASE WHEN 條件查詢

我們並介紹如何用 SQL 來做出
以下的運算：

- Rank (排名函數)
- Aggregate Functions with OVER clause
- Analytic Functions with OVER clause
- Median (中位數)
- Running Total (累積總計)
- Percent to Total (總合百分比)
- Cumulative Percent to Total (累積總合百分比)
- Hierarchical Queries (階層式查詢)

SQL Union

- **UNION** 指令的目的是將兩個 SQL 語句的結果合併起來。從這個角度來看，**UNION** 跟 JOIN 有些許類似，因為這兩個指令都可以由多個表格中擷取資料。**UNION** 的一個限制是兩個 SQL 語句所產生的欄位需要是同樣的資料種類。另外，當我們用 **UNION** 這個指令時，我們只會看到不同的資料值 (類似 SELECT DISTINCT)。

UNION 的語法如下：

[SQL 語句 1] **UNION** [SQL 語句 2]

SQL Union

假設我們有以下的兩個表格，

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

Internet_Sales 表格

Date	Sales
Jan-07-1999	\$250
Jan-10-1999	\$535
Jan-11-1999	\$320
Jan-12-1999	\$750

而我們要找出來所有有營業額 (sales) 的日子。

SQL Union

要達到這個目的，我們用以下的 SQL 語句：

```
SELECT Date FROM Store_Information
```

```
UNION
```

```
SELECT Date FROM Internet_Sales
```

結果：

<u>Date</u>
Jan-05-1999
Jan-07-1999
Jan-08-1999
Jan-10-1999
Jan-11-1999
Jan-12-1999

有一點值得注意的是，如果我們在任何一個 SQL 語句 (或是兩句都一起) 用

```
"SELECT DISTINCT Date"
```

的話，那我們會得到完全一樣的結果。

SQL Union All

- **UNION ALL** 這個指令的目的也是要將兩個 SQL 語句的結果合併在一起。 **UNION ALL** 和 **UNION** 不同之處在於 **UNION ALL** 會將每一筆符合條件的資料都列出來，無論資料值有無重複。

UNION ALL 的語法如下：

[SQL 語句 1] UNION ALL [SQL 語句 2]

我們用和上一頁同樣的例子來顯示出 **UNION ALL** 和 **UNION** 的不同。

SQL Union All

同樣假設我們有以下的兩個表格，

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

Internet_Sales 表格

Date	Sales
Jan-07-1999	\$250
Jan-10-1999	\$535
Jan-11-1999	\$320
Jan-12-1999	\$750

而我們要找出有店面營業額以及網路營業額的日子。

SQL Union All

要達到這個目的，我們用以下的 SQL 語句：

```
SELECT Date FROM Store_Information
```

```
UNION ALL
```

```
SELECT Date FROM Internet_Sales
```

結果:

<u>Date</u>
Jan-05-1999
Jan-07-1999
Jan-08-1999
Jan-08-1999
Jan-07-1999
Jan-10-1999
Jan-11-1999
Jan-12-1999

SQL Intersect

- 和 **UNION** 指令類似，**INTERSECT** 也是對兩個 SQL 語句所產生的結果做處理的。不同的地方是，**UNION** 基本上是一個 **OR** (如果這個值存在於第一句或是第二句，它就會被選出)，而 **INTERSECT** 則比較像 **AND** (這個值要存在於第一句和第二句才會被選出)。

UNION 是「聯集」，而 **INTERSECT** 是「交集」。

INTERSECT 的語法為：

[SQL 語句 1] INTERSECT [SQL 語句 2]

SQL Intersect

假設我們有以下的兩個表格，

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

Internet_Sales 表格

Date	Sales
Jan-07-1999	\$250
Jan-10-1999	\$535
Jan-11-1999	\$320
Jan-12-1999	\$750

而我們要找出哪幾天有店面交易和網路交易。

SQL Intersect

要達到這個目的，我們用以下的 SQL 語句：

```
SELECT Date FROM Store_Information
```

```
INTERSECT
```

```
SELECT Date FROM Internet_Sales
```

結果:

<u>Date</u>
Jan-07-1999

請注意，在 **INTERSECT** 指令下，不同的值只會被列出一次。

SQL Minus

- **MINUS** (Oracle)、**EXCEPT** (MS SQL)指令是運用在兩個 SQL 語句上
它先找出第一個 SQL 語句所產生的結果，
然後看這些結果「有沒有在第二個 SQL 語句的結果中」。
- 如果「有」的話，那這一筆資料就被「**去除**」，而不會在最後的結果中出現。
- 如果「沒有」的話，那這一筆資料就被「**保留**」，而就會在最後的結果中出現。

MINUS 的語法為：

[SQL 語句 1] MINUS [SQL 語句 2]

SQL Minus

我們繼續使用同樣的例子，

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

Internet_Sales 表格

Date	Sales
Jan-07-1999	\$250
Jan-10-1999	\$535
Jan-11-1999	\$320
Jan-12-1999	\$750

而我們要知道有哪幾天是

有店面營業額

而

沒有網路營業額的。

SQL Minus

要達到這個目的，我們用以下的 SQL 語句：

SELECT Date FROM Store_Information

MINUS

SELECT Date FROM Internet_Sales

結果:

<u>Date</u>
Jan-05-1999
Jan-08-1999

“Jan-05-1999”, “Jan-07-1999”, and “Jan-08-1999” 是 “**SELECT Date FROM Store_Information**” 所產生的結果。在這裡面，"Jan-07-1999" 是存在於 “**SELECT Date FROM Internet_Sales**” 所產生的結果中。因此 “Jan-07-1999” 並不在最後的結果中。

請注意，在 **MINUS** 指令下，不同的值只會被列出一次。

SQL 子查詢

- 我們可以在一個 SQL 語句中放入另一個 SQL 語句。當我們在 **WHERE** 子句或 **HAVING** 子句中插入另一個 SQL 語句時，我們就有一個子查詢 (subquery) 的架構。子查詢的作用是什麼呢？第一，它可以被用來連接表格。另外，有的時候子查詢是唯一能夠連接兩個表格的方式。

子查詢的語法為：

```
SELECT "欄位1" FROM "表格"  
WHERE "欄位2" [比較運算素]  
(SELECT "欄位1" FROM "表格" WHERE [條件])
```

[比較運算素] 可以是相等的運算素，例如 =, >, <, >=, <=。這也可以是一個對文字的運算素，例如 "LIKE"。

綠色的部分代表外查詢，紅色的部分代表內查詢。

SQL 子查詢

我們就用剛剛在闡述 SQL 連接時用過的例子：

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

Geography 表格

region_name	store_name
East	Boston
East	New York
West	Los Angeles
West	San Diego

我們要運用 subquery 來找出所有在西部的店的營業額。

SQL 子查詢

要達到這個目的，我們用以下的 SQL 語句：

```
SELECT SUM(Sales) FROM Store_Information  
WHERE Store_name IN  
(SELECT store_name FROM Geography  
WHERE region_name = 'West')
```

結果:

<u>SUM(Sales)</u>
2050

在這個例子中，我們並沒有直接將兩個表格連接起來，然後由此直接算出每一間西區店面的營業額。我們做的是先找出哪些店是在西區的，然後再算出這些店的營業額總共是多少。

SQL 子查詢

在以上的例子，內部查詢本身與外部查詢沒有關係。
這一類的子查詢稱為『簡單子查詢』 (Simple Subquery)。
如果內部查詢是要利用到外部查詢提到的表格中的欄位，
那這個子查詢就被稱為『相關子查詢』 (Correlated Subquery)。
以下是一個相關子查詢的例子：

```
SELECT SUM(a1.Sales) FROM Store_Information a1  
WHERE a1.Store_name IN  
(SELECT store_name FROM Geography a2  
WHERE a2.store_name = a1.store_name)
```

紅色部分即是外部查詢提到的表格中的欄位。

SQL 子查詢

每個子查詢都可視為一個資料查詢的結果集
查詢與查詢之間彼此獨立不能互相使用對方的欄位
只能在查詢的最後再做連接(join)查詢
以下是一個子查詢與子查詢之間join查詢的例子：

```
SELECT * FROM (  
    SELECT GEOGRAPHY_ID, REGION_NAME FROM GEOGRAPHY  
) G,  
(  
    SELECT GEOGRAPHY_ID, STORE_NAME  
    FROM STORE_INFORMATION  
) S  
WHERE G.GEOGRAPHY_ID = S.GEOGRAPHY_ID
```

SQL 子查詢

WITH (Common Table Expressions)

子查詢與子查詢之間join查詢，有個支援更為合適的SQL專用語法 **WITH AS**，使用SQL子查詢在撰寫更有結構性及閱讀性！
且查詢與查詢之間可以相互使用欄位做關聯式查詢
PS：注意只能下面的查詢使用上面查詢的欄位

```
WITH G AS (  
    SELECT GEOGRAPHY_ID, REGION_NAME FROM GEOGRAPHY  
) ,  
S AS (  
    SELECT GEOGRAPHY_ID, STORE_NAME FROM  
    STORE_INFORMATION  
)  
SELECT * FROM G, S  
WHERE G.GEOGRAPHY_ID = S.GEOGRAPHY_ID
```

SQL 子查詢相關應用(分頁功能)

- 我們可以經常的在一般網站上看到資料分頁的功能，那麼這是怎麼辦到的呢？也就是SQL是如何辦到資料分頁的功能。
- 其實是利用SQL的**子查詢**語法**搭配ROWNUMBER**取得資料編號以達到**取得指定區間資料**的功能效果。

來看一個例子。假設我們有以下表格：

STORE_ID	STORE_NAME	SALES	STORE_DATE
1	Boston	2200	2018-03-09 00:00:00
2	Los Angeles	1400	2018-04-05 00:00:00
3	San Diego	250	2018-01-07 00:00:00
4	Los Angeles	300	2018-02-07 00:00:00
5	Albany, Crossgates	2500	2018-05-15 00:00:00
6	Buffalo, Walden Galleria	3000	2018-06-10 00:00:00
7	San Diego	500	2018-02-15 00:00:00
8	Los Angeles	1700	2018-02-07 00:00:00
9	Boston	1600	2018-03-09 00:00:00

SQL 子查詢相關應用(分頁功能)

我們要**第二頁**資料(假設**一頁3筆**)就鍵入以下SQL：

PS：若SQL本身有GROUP BY子句，就必須要再多包覆一層子查詢了

```
SELECT * FROM (  
    SELECT ROWNUM ROW_NUM, S.* FROM STORE_INFORMATION S  
) WHERE ROW_NUM > 3 AND ROW_NUM <= 6
```

結果：

ROW_NUM	STORE_ID	STORE_NAME	SALES	STORE_DATE
4	4	Los Angeles	300	2018-02-07 00:00:00
5	5	Albany, Crossgates	2500	2018-05-15 00:00:00
6	6	Buffalo, Walden Galleria	3000	2018-06-10 00:00:00

SQL EXISTS

- 在上一頁中，我們用 **IN** 來連接內查詢和外查詢。另外有數個方式，例如 >，<，及 =，都可以用來連接內查詢和外查詢。**EXISTS** 也是其中一種方式。這一頁我們將討論 **EXISTS** 的用法。
- **EXISTS** 是用來測試「內查詢」有沒有產生任何結果。如果有的話，系統就會執行「外查詢」中的 SQL。若是沒有的話，那整個 SQL 語句就不會產生任何結果。

EXISTS的語法為：

SELECT "欄位1" FROM "表格1" WHERE

EXISTS

(SELECT * FROM "表格2" WHERE [條件])

在內查詢中，我們並不一定要用 * 來選出所有的欄位。我們也可以選擇表格2中的任何欄位。這兩種做法最後的結果是一樣的。

SQL EXISTS

來看一個例子。假設我們有以下的兩個表格：

Store_Information 表格

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
Los Angeles	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

Geography 表格

region_name	store_name
East	Boston
East	New York
West	Los Angeles
West	San Diego

而我們打入的 SQL 是：

```
SELECT SUM(Sales) FROM Store_Information WHERE  
EXISTS  
(SELECT * FROM Geography WHERE region_name = 'West')
```

SQL EXISTS

我們會得到以下的答案：
SUM(Sales)
2750

乍看之下，這個答案似乎不太正確，因為內查詢有包含一個 `[region_name = 'West']` 的條件，可是最後的答案並沒有包含這個條件。實際上，這並沒有問題。在這個例子中，內查詢產生了超過一筆的資料，所以 **EXISTS** 的條件成立，所以外查詢被執行。而外查詢本身並沒有包含 `[region_name = 'West']` 這個條件

SQL CASE

- **CASE** 是 SQL 用來做為 if-then-else 之類邏輯的關鍵字
- 若是在單純只有條件判斷的情況需求下，接在CASE之後的欄位名可以不寫

CASE的語法為：

```
SELECT CASE ("欄位名")  
    WHEN "條件1" THEN "結果1"  
    WHEN "條件2" THEN "結果2"  
    ...  
    [ELSE "結果N"]  
END  
FROM "表格名"
```

"條件" 可以是一個數值或是公式。 **ELSE** 子句則並不是必須的。

SQL CASE

在我們的 *Store_Information* 中：

store_name	Sales	Date
Los Angeles	\$1500	Jan-05-1999
San Diego	\$250	Jan-07-1999
San Francisco	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

若我們要將 'Los Angeles' 的 Sales 數值乘以2，以及將 'San Diego' 的 Sales 數值乘以1.5，我們就鍵入以下的 SQL：

```
SELECT store_name, CASE store_name  
    WHEN 'Los Angeles' THEN Sales * 2  
    WHEN 'San Diego' THEN Sales * 1.5  
    ELSE Sales END "New Sales", Date  
FROM Store_Information
```

"New Sales" 是用到 **CASE** 那個欄位的欄位名。

SQL CASE

結果:

store_name	New Sales	Date
Los Angeles	\$3000	Jan-05-1999
San Diego	\$375	Jan-07-1999
San Francisco	\$300	Jan-08-1999
Boston	\$700	Jan-08-1999

SQL 排名

- 列出每一行的排名是一個常見的需求，要以 SQL 列出排名，基本的概念是要做一個表格自我連結 (self join)，將結果依序列出，然後算出每一行之前(包含那一行本身)有多少行數。這樣講大家聽得可能有點困惑，所以最好的方式是用一個實例來介紹。

假設我們有 **Total_Sales** 表格：

Name	Sales
John	10
Jennifer	15
Stella	20
Sophia	40
Greg	50
Jeff	30

SQL 排名

要找出每一行的排名，我們就打入以下的 SQL 語句：

```
SELECT a1.Name, a1.Sales, COUNT(a2.sales) Sales_Rank
FROM Total_Sales a1, Total_Sales a2
WHERE a2.Sales >= a1.Sales
GROUP BY a1.Name, a1.Sales
ORDER BY a1.Sales DESC, a1.Name DESC;
```

結果:

<u>Name</u>	<u>Sales</u>	<u>Sales_Rank</u>
Greg	50	1
Sophia	40	2
Stella	20	3
Jeff	20	3
Jennifer	15	5
John	10	6

SQL 排名

我們來看 **WHERE** 子句。

在字句的部分 (**a2.Sales >= a1.Sales**)

算出**有多少筆**資料 Sales 欄位的值是比较**自己本身的值高或是相等**。如果在 Sales 欄位中**沒有同樣大小的資料**，
那這部分的 **WHERE** 子句本身就可以產生出正確的排名。

SQL 排名函數

SQL當然也支援排名的函數，讓你可以更方便的達成一些排名上的需求議題(例:分組排名、現實中的名次規則)當然若你僅僅只是要依序的每一列資料紀錄的編號，則只要使用ROWNUMBER取得就好，不需要用到排名函數

SQL語法：

RANK() OVER ([query_partition_clause] order_by_clause)

RANK(必填)：排名函數,當有同名次時(排名結果是不連續的)

EX:雙冠軍就不會有亞軍

query_partition_clause(選填)：資料分群排名劃分欄位

order_by_clause (必填)：資料排序欄位

SQL 排名函數

假設我們有以下**STORE_INFORMATION**表格：

STORE_ID	STORE_NAME	SALES
1	Boston	2200
2	Los Angeles	1400
3	San Diego	250
4	Los Angeles	300
5	Albany, Crossgates	2500
6	Buffalo, Walden Galleria	3000
7	San Diego	500
8	Los Angeles	1600
9	Boston	1600

要依照營業額做排名，我們就鍵入：

```
SELECT STORE_ID, STORE_NAME, SALES,  
RANK( )OVER(ORDER BY SALES DESC) RANK_SALES  
FROM STORE_INFORMATION
```

SQL 排名函數

結果：

STORE_ID	STORE_NAME	SALES	RANK_SALES
6	Buffalo, Walden Galleria	3000	1
5	Albany, Crossgates	2500	2
1	Boston	2200	3
9	Boston	1600	4
8	Los Angeles	1600	4
2	Los Angeles	1400	6
7	San Diego	500	7
4	Los Angeles	300	8
3	San Diego	250	9

SQL 排名函數

若要依照**各別商店**做**分群(資料劃分)**營業額**排名**，我們就鍵入：

```
SELECT STORE_ID, STORE_NAME, SALES,  
RANK( ) OVER (  
    PARTITION BY STORE_NAME ORDER BY SALES DESC  
) RANK_SALES  
FROM STORE_INFORMATION
```

結果：

STORE_ID	STORE_NAME	SALES	RANK_SALES
5	Albany, Crossgates	2500	1
1	Boston	2200	1
9	Boston	1600	2
6	Buffalo, Walden Galleria	3000	1
8	Los Angeles	1600	1
2	Los Angeles	1400	2
4	Los Angeles	300	3
7	San Diego	500	1
3	San Diego	250	2

SQL 排名函數

其它排名函數：

Analytic Functions with OVER Clause (分析函數)

DENSE_RANK () : 當有同名次時(排名結果是連續的)

PERCENT_RANK () : 名次所佔的百分比

公式：(RANK() - 1) / (總資料列筆數 - 1)

ROW_NUMBER () : 依序編號

```
SELECT STORE_NAME, SALES,  
RANK() OVER (ORDER BY SALES DESC) RANK_SALES,  
PERCENT_RANK() OVER (ORDER BY SALES DESC) PERCENT_RANK  
FROM STORE_INFORMATION
```

SQL 排名函數

結果：

STORE_NAME	SALES	RANK_SALES	PERCENT_RANK
Buffalo, Walden Galleria	3000	1	0
Albany, Crossgates	2500	2	0.125
Boston	2200	3	0.25
Boston	1600	4	0.375
Los Angeles	1600	4	0.375
Los Angeles	1400	6	0.625
San Diego	500	7	0.75
Los Angeles	300	8	0.875
San Diego	250	9	1

Aggregate Functions with OVER Clause (聚合函數)

```
SELECT STORE_ID, STORE_NAME, SALES, GEOGRAPHY_ID,  
-- 依「區域劃分」取營業額"最小值"  
MIN(SALES) OVER (PARTITION BY GEOGRAPHY_ID) MIN_SALES,  
-- 依「區域劃分」取營業額"最大值"  
MAX(SALES) OVER (PARTITION BY GEOGRAPHY_ID) MAX_SALES,  
-- 依「區域劃分」取商店"數量"  
COUNT(STORE_ID) OVER (PARTITION BY GEOGRAPHY_ID) COUNT_STORE_ID,  
-- 依「區域劃分」取營業額"總和"  
SUM(SALES) OVER (PARTITION BY GEOGRAPHY_ID) SUM_SALES,  
-- 依「區域劃分」取營業額"平均"  
AVG(SALES) OVER (PARTITION BY GEOGRAPHY_ID) AVG_SALES  
FROM STORE_INFORMATION  
ORDER BY GEOGRAPHY_ID, SALES;
```

STORE_ID	STORE_NAME	SALES	GEOGRAPHY_ID	MIN_SALES	MAX_SALES	COUNT_STORE_ID	SUM_SALES	AVG_SALES
9	Boston	1500	1	1500	2200	2	3700	1850
1	Boston	2200	1	1500	2200	2	3700	1850
3	San Diego	250	2	250	2500	6	6550	1091.66...
4	Los Angeles	300	2	250	2500	6	6550	1091.66...
7	San Diego	500	2	250	2500	6	6550	1091.66...
2	Los Angeles	1400	2	250	2500	6	6550	1091.66...
8	Los Angeles	1600	2	250	2500	6	6550	1091.66...
5	Albany, Crossgates	2500	2	250	2500	6	6550	1091.66...
6	Buffalo, Walden Galleria	3000	(null)	3000	3000	1	3000	3000

Analytic Functions with **OVER** Clause (分析函數)

```
SELECT STORE_ID, STORE_NAME,  
       ROW_NUMBER() OVER (ORDER BY SALES) ROWNO_STORE,  
       SALES,  
       -- 依「營業額」排序取"上一個"營業額  
       LAG(SALES) OVER (ORDER BY SALES) PREV_SALES,  
       -- 依「營業額」排序取"下一個"營業額  
       LEAD(SALES) OVER (ORDER BY SALES) NEXT_SALES  
FROM STORE_INFORMATION  
ORDER BY SALES;
```

STORE_ID	STORE_NAME	ROWNO_STORE	SALES	PREV_SALES	NEXT_SALES
3	San Diego	1	250	(null)	300
4	Los Angeles	2	300	250	500
7	San Diego	3	500	300	1400
2	Los Angeles	4	1400	500	1500
9	Boston	5	1500	1400	1600
8	Los Angeles	6	1600	1500	2200
1	Boston	7	2200	1600	2500
5	Albany, Crossgates	8	2500	2200	3000
6	Buffalo, Walden Galleria	9	3000	2500	(null)

SQL 中位數

- 要**算出中位數**，我們必須要能夠達成以下幾個目標：
 - 將資料依序排出，並找出每一行資料的排名。
 - 找出『中間』的排名為何。舉例來說，如果總共有 9 筆資料，那中間排名就是 5 (有 4 筆資料比第 5 筆資料大，有 4 筆資料比第 5 筆資料小)。
 - 找出中間排名資料的值。

來看看以下的例子。

SQL 中位數

假設我們有以下 **Total_Sales** 表格：

Name	Sales
John	10
Jennifer	15
Stella	20
Sophia	40
Greg	50
Jeff	20

要找出中位數，我們就鍵入：

```
SELECT Sales Median FROM
(SELECT a1.Name, a1.Sales, COUNT(a1.Sales) Rank
FROM Total_Sales a1, Total_Sales a2
WHERE a1.Sales <= a2.Sales
group by a1.Name, a1.Sales
order by a1.Sales desc) a3
WHERE Rank = (
    SELECT CEIL( COUNT(*) / 2) FROM Total_Sales
);
```

SQL 中位數

結果：

<u>Median</u>
20

大家將會發現，第 2 行到第 6 行是跟產生 排名 的語句完全一樣。第 7 行則是算出中間的排名。

CEIL (無條件進位)是在 Oracle 中算出商的方式。

在不同的資料庫中會有不同的方式求商。

第 1 行則是列出排名中間的資料值。

SQL 累積總計

- 算出**累積總計**是一個常見的需求，可惜以 SQL 並沒有一個很直接的方式達到這個需求。要以 SQL 算出累積總計，基本上的概念與列出排名類似：第一是先做個表格**自我連結 (self join)**，然後將結果依序列出。在做列出排名時，我們算出每一行之前(包含那一行本身)有多少行數；而在做累積總計時，我們則是算出每一行之前(包含那一行本身)的總合。

來看看以下的例子。

SQL 累積總計

假設我們有以下 **Total_Sales** 表格：

Name	Sales
John	10
Jennifer	15
Stella	20
Sophia	40
Greg	50
Jeff	20

要找出累積總計，我們就鍵入：

```
SELECT a1.Name, a1.Sales, SUM(a2.Sales) Running_Total  
FROM Total_Sales a1, Total_Sales a2  
WHERE a1.Sales <= a2.sales  
GROUP BY a1.Name, a1.Sales  
ORDER BY a1.Sales DESC, a1.Name DESC;
```

SQL 累積總計

結果：

<u>Name</u>	<u>Sales</u>	<u>Running_Total</u>
Greg	50	50
Sophia	40	90
Stella	20	110
Jeff	20	130
Jennifer	15	145
John	10	155

在以上的 SQL 語句中， **WHERE** 子句和 **ORDER BY** 子句讓我們能夠在有重複值時能夠算出正確的累積總計。

SQL 總和百分比

- 要用 SQL 算出**總和百分比**，我們需要用到算排名和累積總計的概念，以及運用子查詢的做法。在這裡，我們把子查詢放在外部查詢的 **SELECT** 子句中。

來看例子。

Total_Sales 表格

Name	Sales
John	10
Jennifer	15
Stella	20
Sophia	40
Greg	50
Jeff	20

SQL 總和百分比

要算出總合百分比，我們鍵入：

```
SELECT a1.Name, a1.Sales,  
a1.Sales / ( SELECT SUM(Sales) FROM Total_Sales ) Pct_To_Total  
FROM Total_Sales a1, Total_Sales a2  
WHERE a1.Sales <= a2.sales  
GROUP BY a1.Name, a1.Sales  
ORDER BY a1.Sales DESC, a1.Name DESC;
```

結果：

<u>Name</u>	<u>Sales</u>	<u>Pct_To_Total</u>
Greg	50	0.3226
Sophia	40	0.2581
Stella	20	0.1290
Jeff	20	0.1290
Jennifer	15	0.0968
John	10	0.0645

"SELECT SUM(Sales)
FROM Total_Sales" 這一段子
查詢是用來算出總合。總合算
出後，我們就能夠將每一行一
一除以總合來求出每一行的總
合百分比。

SQL 累積總和百分比

- 要用 SQL 累積總和百分比算出，我們運用類似總和百分比的概念。兩者的不同處在於在這個情況下，我們要算出到目前為止的累積總和是所有總和的百分之幾，而不是光看每一筆資料是所有總和的百分之幾。

來看以下例子。 ***Total_Sales*** 表格

Name	Sales
John	10
Jennifer	15
Stella	20
Sophia	40
Greg	50
Jeff	20

SQL 累積總和百分比

要算出累積總和百分比，我們鍵入：

```
SELECT a1.Name, a1.Sales,  
SUM(a2.Sales) / (SELECT SUM(Sales) FROM Total_Sales) Pct_To_Total  
FROM Total_Sales a1, Total_Sales a2  
WHERE a1.Sales <= a2.sales  
GROUP BY a1.Name, a1.Sales  
ORDER BY a1.Sales DESC, a1.Name DESC;
```

結果：

<u>Name</u>	<u>Sales</u>	<u>Pct_To_Total</u>
Greg	50	0.3226
Sophia	40	0.5806
Stella	20	0.7097
Jeff	20	0.8387
Jennifer	15	0.9355
John	10	1.0000

"SELECT SUM(Sales)
FROM Total_Sales" 這一段子查詢
是用來算出總和。我們接下來用累
積總計 "SUM(a2.Sales)" 除以總和
來求出每一行的累積總和百分比。

SQL Hierarchical Queries (Oracle)

一般用來查尋存在父子關係的資料，也就是樹狀的資料結構。

其返還的數據也能夠明確的區分出每一層數據的「**階層式查詢**」

- **start with** condition1 :

是用來限制第一層的數據，或者叫根節點數據，以這部分數據為基礎來查找第二層數據，然後以第二層數據查找第三層數據以此類推。

- **connect by** [prior] :

這部分是用來查找時以怎樣的一種關係去查找，比如說查找第二層的數據時用第一層的ID去跟資料表裡面記錄的parentId的欄位進行匹配，如果這個條件成立那麼查找出來的數據就是第二層數據，同理查找第三層第四層.....等等都是按這樣去匹配 **recursive queries**(遞迴查詢)。

SQL語法：

```
select * from table
```

```
[start with condition1]
```

```
connect by [prior] id = parentid
```


SQL Hierarchical Queries (Oracle)

由父階層「由上至下」查詢所有子階層：
Level 表示為「層級數字」

```
Select Employee_Id, First_Name, Manager_Id, Level
From Employees
-- START WITH 父階層 EMPLOYEE_ID
Start With Employee_Id = 100
-- 由父階層 EMPLOYEE_ID 尋找所屬下一個子階層 MANAGER_ID
Connect By Prior Employee_Id = Manager_Id
```

PS：第一層根節點的 **Manager_Id** 值必須是 NULL
若值為100 會有 ORA-01436: CONNECT BY loop in user data 的錯誤

SQL Hierarchical Queries (Oracle)

結果:

EMPLOYEE_ID	FIRST_NAME	MANAGER_ID	LEVEL
100	Steven		1
101	Neena	100	2
108	Nancy	101	3
109	Daniel	108	4
110	John	108	4
111	Ismael	108	4
112	Jose Manuel	108	4
113	Luis	108	4
200	Jennifer	101	3
203	Susan	101	3
204	Hermann	101	3
205	Shelley	101	3
206	William	205	4
102	Lex	100	2
103	Alexander	102	3
104	Bruce	103	4
105	David	103	4
106	Valli	103	4
107	Diana	103	4

SQL Hierarchical Queries (Oracle)

由子階層「由下至上」查詢所有父階層：

```
Select Employee_Id, First_Name, Manager_Id, Level  
From Employees
```

```
-- START WITH 子階層 EMPLOYEE_ID
```

```
Start With Employee_Id = 206
```

```
-- 由子階層 MANAGER_ID 尋找所屬上一個父階層 EMPLOYEE_ID
```

```
Connect By Employee_Id = Prior Manager_Id
```

結果：

EMPLOYEE_ID	FIRST_NAME	MANAGER_ID	LEVEL
206	William	205	1
205	Shelley	101	2
101	Neena	100	3
100	Steven		4

SQL Hierarchical Queries (Oracle)

由父階層「由上至下」查詢所有子階層，搭配 **SYS_CONNECT_BY_PATH** 函數可查詢列出該階層所屬之所有父階層「資料路徑」

```
Select Employee_Id, First_Name, Manager_Id,  
-- 列出所有父階層資料  
SYS_CONNECT_BY_PATH(Employee_Id, '-') DATA_PATH,  
Level  
From Employees  
-- START WITH 父階層 EMPLOYEE_ID  
Start With Employee_Id = 100  
-- 由父階層 EMPLOYEE_ID 尋找所屬下一階層的子階層 MANAGER_ID  
Connect By Prior Employee_Id = Manager_Id
```

SQL Hierarchical Queries (Oracle)

結果：

EMPLOYEE_ID	FIRST_NAME	MANAGER_ID	DATA_PATH	LEVEL
100	Steven		-100	1
101	Neena	100	-100-101	2
108	Nancy	101	-100-101-108	3
109	Daniel	108	-100-101-108-109	4
110	John	108	-100-101-108-110	4
111	Ismael	108	-100-101-108-111	4
112	Jose Manuel	108	-100-101-108-112	4
113	Luis	108	-100-101-108-113	4
200	Jennifer	101	-100-101-200	3
203	Susan	101	-100-101-203	3
204	Hermann	101	-100-101-204	3
205	Shelley	101	-100-101-205	3
206	William	205	-100-101-205-206	4
102	Lex	100	-100-102	2
103	Alexander	102	-100-102-103	3
104	Bruce	103	-100-102-103-104	4
105	David	103	-100-102-103-105	4
106	Valli	103	-100-102-103-106	4
107	Diana	103	-100-102-103-107	4

SQL Recursive Queries (MySQL)

MySQL並未有直接支援recursive queries(遞迴查詢)的語法
但仍可以透過一般 Table join 的方式來達到遞迴查詢的效果




-- 由父階層「由上至下」查詢所有子階層：

-- 由父階層 EMPLOYEE_ID 尋找所屬下一個子階層 MANAGER_ID

```
SELECT E1.EMPLOYEE_ID E1_ID, E1.MANAGER_ID E1_MID,  
       E2.EMPLOYEE_ID E2_ID, E2.MANAGER_ID E2_MID,  
       E3.EMPLOYEE_ID E3_ID, E3.MANAGER_ID E3_MID,  
       E4.EMPLOYEE_ID E4_ID, E4.MANAGER_ID E4_MID  
FROM EMPLOYEES E1  
JOIN EMPLOYEES E2 ON E1.EMPLOYEE_ID = E2.MANAGER_ID  
JOIN EMPLOYEES E3 ON E2.EMPLOYEE_ID = E3.MANAGER_ID  
JOIN EMPLOYEES E4 ON E3.EMPLOYEE_ID = E4.MANAGER_ID  
WHERE E1.EMPLOYEE_ID = 100 -- 啟始的父階層編號  
ORDER BY E4.EMPLOYEE_ID
```

SQL 練習題(八)

- 使用案列說明:異業結盟策略行銷，欲針對貢獻度較高的顧客提供現金折價卷
- 查詢條件：
 - 1.顧客：查詢結果排除黑名單顧客
 - 2.顧客：同時有在「星巴克」、「阿馬龍」購買
 - 3.消費金額：購買**平均金額**在「星巴克」大於或等於 500
「阿馬龍」大於或等於 1000
 - 4.日期區間：2017-10 ~ 2017-12
- 搜尋結果欄位：顧客身份證字號、姓名、手機號碼

	 CUS_IDENTIFIER_ID	 CUS_NAME	 CUS_PHONE_ENUMBER
1	A123213967	曹操	0923910594
2	B122123109	黃忠	0928777849
3	F123423426	諸葛亮	0939485948
4	U192039489	典韋	0923948288
5	Y140394950	孫權	0910293403
6	Y203948377	貂蟬	0919288890



EXP_06_Table.sql



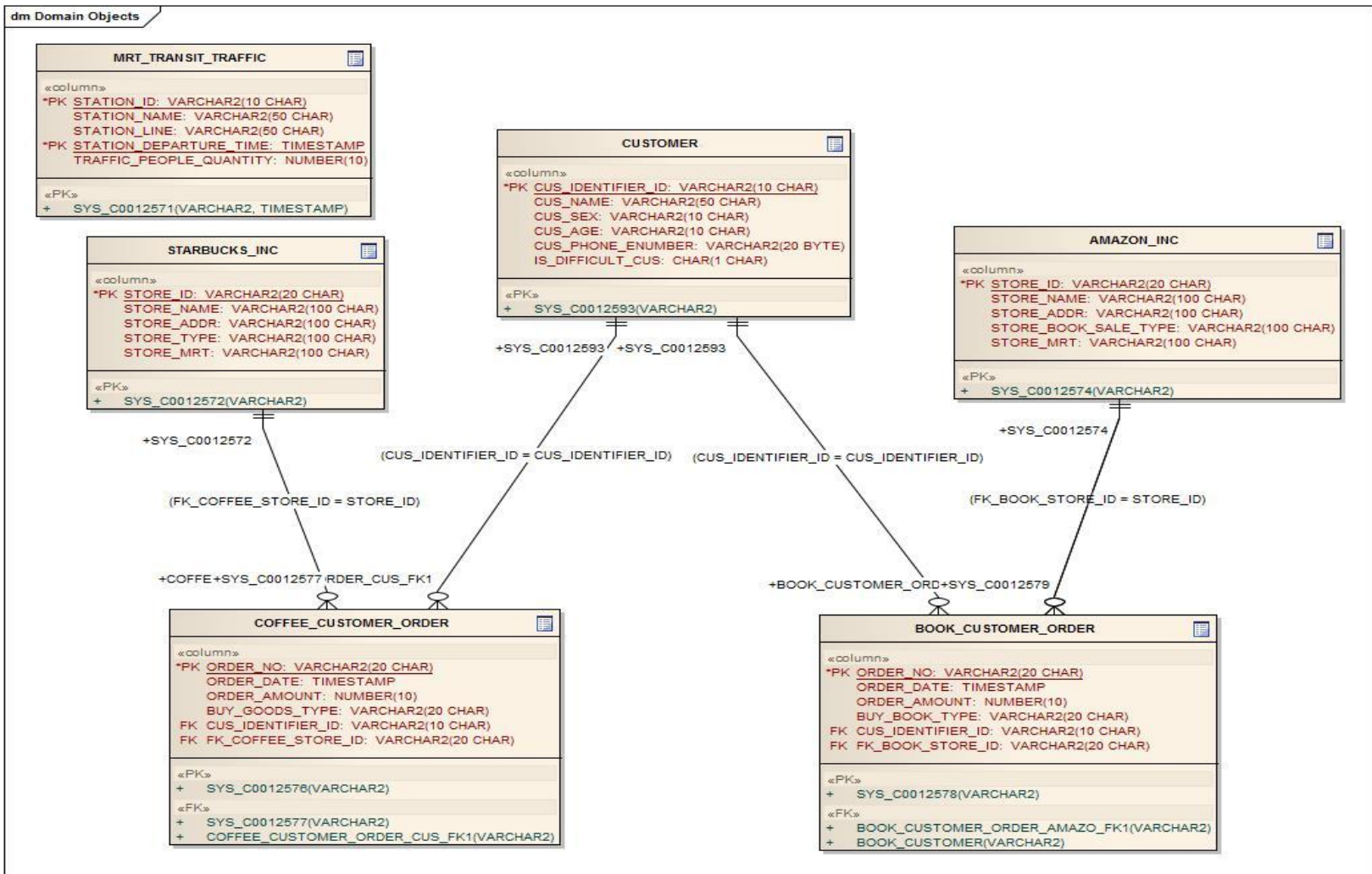
EXP_06_Data.sql



EXP_06_Answer.sql

SQL 練習題(八)

資料庫表格 Data Modeling



常用的SQL FUNCTION

□ 字符函數：返回「**字符**」值

1. CHR(x) : 返回ASCII碼表示的字符，與ASCII相反
2. CONCAT(string1, string2) : 返回串接上string2後的string1，與 || 運算符相同
3. INITCAP(string) : 返回string中每個單詞第一個字母大寫，而其它字母小寫的string，不是字母的字符不受影響。
4. LOWER(string) : 返回小寫形式的string，不是字母的不受影響。
5. UPPER(string) : 返回大寫形式的string
6. LPAD(string1, x , [string2]) : 在左邊填充string2, 使string1的長度為x, 如果未指定string2, 則以空格填充
7. RPAD(string1, x , [string2]) : 與LPAD相同，唯填充在右邊

常用的SQL FUNCTION

8. LTRIM (string1, [string2]) : 刪除string1中最左邊与string2相同的字串，string2若未給值預設為單個空格
9. RTRIM(string1, [string2]) : 與LTRIM類似，只是刪除最右邊.
10. REPLACE(string, search_str , [replace_str]) : 用replace_str替換string中所有為search_str的字串並將結果返回, 如果未指定replace_str, 則刪除string中所有為search_str的字串
11. SUBSTR(string , a , [b]) :
返回string中從 a 位置開始取 b 個字符長度的部分字串
如果a = 0，就被認為是1，如果a > 0，則從左往右起定位a,
如果a < 0，則從右往左起定位a
如果未指定b, 返回定位a後的所有部分, 如果b < 1，返回null.

常用的SQL FUNCTION

- 12. SUBSTRB(string, a, [b]) : 与SUBSTR相類似，唯substrb以字節為單位而不是以字符為單位
- 13. TRANSLATE(string, from_str, to_str) : 返回將string中的from_str替換為to_str，to_str不能為空，如果有任何參數為空，則返回null.

常用的SQL FUNCTION

□ 字符函數：返回「**數字**」值

1. ASCII(string)：返回字串的ASCII值

2. INSTR(string1, string2 [a, [b]])：

返回string2在string1中的位置

a > 0從**左邊**開始掃描，**a** < 0從**右邊**開始掃描

返回第b次出現的位置，a和b預設值為1

3. INSTRB(string1, string2 [a, [b]])：

類似於INSTR但是以字節為單位(一個中文字三個字節)
進行表達的

常用的SQL FUNCTION

4. LENGTH(string) :

以字符為單位返回STRING的長度(注意CHAR問題)

5. LENGTHB(string) :

類似於LENGTH以字節為單位進行表達的.

常用的SQL FUNCTION

□ 數字函數:

- 1.ABS(x)：返回x的絕對值
- 2.CEIL(x)：返回大於或等於x的最大整數值(無條件進位)
- 3.FLOOR(x)：返回小於或等於x的最小整數值(無條件捨去)
- 4.MOD(x,y)：返回x除以y的餘數, 如果y為0則返回x
- 5.ROUND(x,[y])：
返回(四捨五入)到小數點右邊y位的x值, y預設值為0
如果y是負數，則捨入到小數點左邊相應的整數位上

常用的SQL FUNCTION

- 6. SIGN(x) : 如果 $x < 0$ 則返回-1, 如果 $x = 0$ 則返回0, 如果 $x > 0$ 則返回1
- 7. TRUNC(x , [y]) : 返回截尾到Y位小數的X值，不做捨入處理, Y預設值為0, 將X截尾為一個整數值, 如果Y是負數則結尾到小數點左邊相應的位上

常用的SQL FUNCTION

□ 日期函數

- 1.ADD_MONTHS(d,x)：返回日期d加上x個月的結果
- 2.LAST_DAY(d)：返回日期d的月份的最後一天的日期
- 3.MONTHS_BETWEEN(date1,date2)：返回在 date1 和 date2 相差的月份數. 如果‘日’相同或date1和date2都是所在月最后一天，則返回整數，否則返回的結果將包含一個分數部分(Oracle以每月31天為計算結果的小數部分)

常用的SQL FUNCTION

4. SYSDATE：返回當前的日期和時間

5. TRUNC(date, [format])：對日期作無條件捨去運算(oracle)
MONTH(月捨去)、YEAR(年捨去)、不帶FORMAT(日捨去)

□ MySQL：YEAR(date)取年、MONTH(date)取月份、DAY(date)取日
HOUR(date)取小時、MINUTE(date)取分鐘、SECOND(date)取秒

常用的SQL FUNCTION

□ 日期算術(Oracle):

1. $d1 - d2$: 返回 $d1$ 和 $d2$ 之間相差的天數 (兩個日期不能相加)
2. $d1 + n$: 在 $d1$ 上加上 n 天並作為date類型返回結果
3. $d1 - n$: 從 $d1$ 上減去 n 天並作為date類型返回結果
4. $d - \text{INTERVAL '1' UNIT}$: 日期間隔計算

常用的 unit 單位有：

SECOND、MINUTE、HOUR、DAY、MONTH、YEAR

□ 日期算術(MySQL):

日期加減的函數有**DATE_ADD()**、**DATE_SUB()**

DATE_SUB()語法(Syntax): **DATE_SUB**(datetime, **INTERVAL** expr **UNIT**)

expr 用來指定你要減去的時間間隔，UNIT 是 expr 的單位。

常用的 unit 單位有：

MINUTE、HOUR、DAY、WEEK、MONTH、YEAR

常用的SQL FUNCTION

□ 轉換函數：

1. `TO_CHAR(d, 'YYYY-mm-DD HH24:MI:SS')`

將日期轉換字符串

2. `TO_DATE(string, 'YYYY-mm-DD HH24:MI:SS')`

字符串轉換日期

3. `TO_TIMESTAMP(string, 'YYYY-MM-DD HH24:MI:SS.FF3')`

字符串轉換日期

4. `TO_NUMBER(string)`：字符串轉數字

常用的SQL FUNCTION

□ 分組函數：

1.AVG([DISTINCT | ALL] col):返回平均值

2.COUNT(* | [DISTINCT | ALL] col) :

返回查詢中行的數

3.MAX([DISTINCT | ALL] col) :

返回選擇列表項目的最大值, DISTINCT和ALL不起作用

4.MIN([DISTINCT | ALL] col) : 返回最小值

5.SUM([DISTINCT | ALL] col) :

返回選擇列表項目的數值的總和

常用的SQL FUNCTION

□ 其它函數

1. DECODE(base_expr, compara1, value1, compara2, value2, , default)

類似相同於CASE...WHEN...THEN...ELSE...END 功能

2. NVL(expr1, expr2) 適用於 Oracle

IFNULL(expr1, expr2) 適用於 MySQL

如果expr1是NULL, 則返回expr2, 否則返回expr1

3. UID : 返回唯一標識當前數據庫用戶的整數

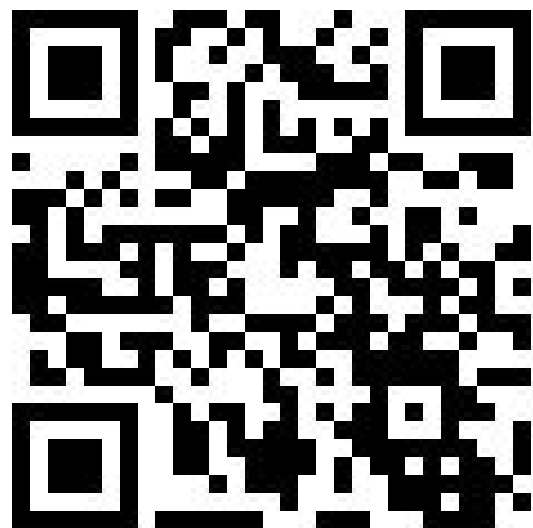
4. USER : 返回當前ORACLE用戶的名字

附錄：Sqldeveloper 快捷鍵

- 轉換大小寫(CTRL + SHIFT + 雙引號)
- 註解(CTRL + /)
- 自動排版格式(CTRL + F7)
- 執行語句(CTRL + ENTER)
- COMMIT(F11)
- ROLLBACK(F12)

講師資訊

- FB伯樂粉絲團



- FB個人臉書

