

无论您是一位 SQL 的新手，或是一位只是需要对 SQL 复习一下的资料仓储业界老将，您就来对地方了。这个 SQL 教材网站列出常用的 SQL 指令,包含以下几个部分：

- ◆ SQL 指令: SQL 如何被用来储存、读取、以及处理数据库之中的资料。
- ◆ 表格处理: SQL 如何被用来处理数据库中的表格。
- ◆ 进阶 SQL: 介绍 SQL 进阶概念，以及如何用 SQL 来执行一些较复杂的运算。
- ◆ SQL 语法: 这一页列出所有在这个教材中被提到的 SQL 语法。

对于每一个指令，我们将会先列出及解释这个指令的语法，然后用一个例子来让读者了解这个指令是如何被运用的。当您读完了这个网站的所有教材后，您将对 SQL 的语法会有一个大致上的了解。另外，您将能够正确地运用 SQL 来由数据库中获取信息。笔者本身的经验是，虽然要对 SQL 有很透彻的了解并不是一朝一夕可以完成的，可是要对 SQL 有个基本的了解并不难。希望在看完这个网站后，您也会有同样的想法。

SQL 指令

SELECT

是用来做什么的呢？一个最常用的方式是将资料从数据库中的表格内选出。从这一句回答中，我们马上可以看到两个关键字： 从 (FROM) 数据库中的表格内选出 (SELECT)。(表格是一个数据库内的结构，它的目的是储存资料。在表格处理这一部分中，我们会提到如何使用 SQL 来设定表格。) 我们由这里可以看到最基本的 SQL 架构：

SELECT "栏位名" FROM "表格名"

我们用以下的例子来看看实际上是怎么用的。假设我们有以下这个表格：

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

若要选出所有的店名 (store_Name)，我们就打入：

```
SELECT store_name FROM Store_Information
```

结果:

store_name
Los Angeles
San Diego
Los Angeles
Boston

我们一次可以读取好几个栏位，也可以同时由好几个表格中选资料。

DISTINCT

SELECT 指令让我们能够读取表格中一个或数个栏位的所有资料。这将把所有的资料都抓出，无论资料值有无重复。在资料处理中，我们会经常碰到需要找出表格内的不同资料值的情况。换句话说，我们需要知道这个表格/栏位内有哪些不同的值，而每个值出现的次数并不重要。这要如何达成呢？在 **SQL** 中，这是很容易做到的。我们只要在 **SELECT** 后加上一个 **DISTINCT** 就可以了。**DISTINCT** 的语法如下：

```
SELECT DISTINCT "栏位名"  
FROM "表格名"
```

举例来说，若要在以下的表格，Store_Information，找出所有不同的店名时，

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

我们就键入，

```
SELECT DISTINCT store_name FROM Store_Information
```

结果:

```
store_name  
Los Angeles
```

San Diego
Boston

WHERE

我们并不一定每一次都要将表格内的资料都完全抓出。在许多时候，我们会需要选择性地抓资料。就我们的例子来说，我们可能只要抓出营业额超过 \$1,000 的资料。要做到这一点，我们就需要用到 **WHERE** 这个指令。这个指令的语法如下：

```
SELECT "栏位名"  
FROM "表格名"  
WHERE "条件"
```

若我们要由以下的表格抓出营业额超过 \$1,000 的资料，

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

我们就键入，

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

结果：

```
store_name  
Los Angeles
```

AND OR

在上一页中，我们看到 **WHERE** 指令可以被用来由表格中有条件地选取资料。这个条件可能是简单的（像上一页的例子），也可能是复杂的。复杂条件是由二或多个简单条件透过

AND 或是 OR 的连接而成。一个 SQL 语句中可以有无限多个简单条件的存在。

复杂条件的语法如下：

```
SELECT "栏位名"  
FROM "表格名"  
WHERE "简单条件"  
{[AND|OR] "简单条件"}+
```

{ }+ 代表 {} 之内的情况会发生一或多次。在这里的意思就是 AND 加简单条件及 OR 加简单条件的情况可以发生一或多次。另外，我们可以用 () 来代表条件的先后次序。

举例来说，我们若要在 Store_Information 表格中选出所有 Sales 高于 \$1,000 或是 Sales 在 \$500 及 \$275 之间的资料的话，

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

我们就键入，

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

结果：

```
store_name  
Los Angeles  
San Francisco
```

IN

在 SQL 中，在两个情况下会用到 IN 这个指令；这一页将介绍其中之一：与 WHERE 有关的那一个情况。在这个用法下，我们事先已知道至少一个我们需要的值，而我们将这些知道的值都放入 IN 这个子句。IN 指令的语法为下：

```
SELECT "栏位名"
```

```
FROM "表格名"  
WHERE "栏位名" IN ('值一', '值二', ...)
```

在括号内可以有一或多个值，而不同值之间由逗号分开。值可以是数目或是文字。若在括号内只有一个值，那这个子句就等于

```
WHERE "栏位名" = '值一'
```

举例来说，若我们要在 Store_Information 表格中找出所有含盖 Los Angeles 或 San Diego 的资料，

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

我们就键入，

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

结果:

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

BETWEEN

IN 这个指令可以让我们依照一或数个不连续 (discrete) 的值的限制之内抓出资料库中的值，而 BETWEEN 则是让我们可以运用一个范围 (range) 内抓出资料库中的值。BETWEEN 这个子句的语法如下：

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

这将选出栏位值包含在值一及值二之间的每一笔资料。

举例来说，若我们要由 Store_Information 表格中找出所有介于 January 6, 1999 及 January

10, 1999 中的资料,

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

我们就键入,

```
SELECT *  
FROM Store_Information  
WHERE Date BETWEEN 'Jan-06-1999' AND 'Jan-10-1999'
```

请读者注意: 在不同的数据库中, 日期的储存法可能会有所不同。在这里我们选择了其中一种储存法。

结果:

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

LIKE

LIKE 是另一个在 WHERE 子句中会用到的指令。基本上, LIKE 能让我们依据一个模式 (pattern) 来找出我们要的资料。相对来说, 在运用 IN 的时候, 我们完全地知道我们需要的条件; 在运用 BETWEEN 的时候, 我们则是列出一个范围。LIKE 的语法如下:

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

{模式} 经常包括野卡 (wildcard)。以下是几个例子:

'A_Z': 所有以 'A' 起头, 另一个任何值的字原, 且以 'Z' 为结尾的字符串。'ABZ' 和 'A2Z' 都符合这一个模式, 而 'AKKZ' 并不符合 (因为在 A 和 Z 之间有两个字原, 而不是一个字原)。

'ABC%': 所有以 'ABC' 起头的字符串。举例来说, 'ABCD' 和 'ABCABC' 都符合这个模式。

'%XYZ': 所有以 'XYZ' 结尾的字符串。举例来说, 'WXYZ' 和 'ZZXYZ' 都符合这个模式。

'%AN%': 所有含有 'AN'这个模式的字符串。举例来说, 'LOS ANGELES' 和 'SAN FRANCISCO' 都符合这个模式。

我们将以上最后一个例子用在我们的 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
SAN FRANCISCO	\$300	Jan-08-1999
BOSTON	\$700	Jan-08-1999

我们就键入,

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

结果:

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

ORDER BY

到目前为止, 我们已学到如何藉由 **SELECT** 及 **WHERE** 这两个指令将资料由表格中抓出。不过我们尚未提到这些资料要如何排列。这其实是一个很重要的问题。事实上, 我们经常需要能够将抓出的资料做一个有系统的显示。这可能是由小往大 (**ascending**) 或是由大往小 (**descending**)。在这种情况下, 我们就可以运用 **ORDER BY** 这个指令来达到我们的目的。

ORDER BY 的语法如下:

```
SELECT "栏位名"  
FROM "表格名"  
[WHERE "条件"]  
ORDER BY "栏位名" [ASC, DESC]
```

[] 代表 **WHERE** 子句不是一定需要的。不过, 如果 **WHERE** 子句存在的话, 它是在 **ORDER BY** 子句之前。ASC 代表结果会以由小往大的顺序列出, 而 DESC 代表结果会以

由大往小的顺序列出。如果两者皆没有被写出的话，那我们就会用 `ASC`。

我们可以照好几个不同的栏位来排顺序。在这个情况下，`ORDER BY` 子句的语法如下(假设有两个栏位)：

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

若我们对这两个栏位都选择由小往大的话，那这个子句就会造成结果是依据 "栏位一" 由小往大排。若有好几笔资料 "栏位一" 的值相等，那这几笔资料就依据 "栏位二" 由小往大排。

举例来说，若我们要依照 `Sales` 栏位的由大往小列出 `Store_Information` 表格中的资料，

`Store_Information` 表格

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

我们就键入，

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

结果：

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，我们用栏位名来指定排列顺序的依据。除了栏位名外，我们也可以用栏位的顺序 (依据 `SQL` 句中的顺序)。在 `SELECT` 后的第一个栏位为 1，第二个栏位为 2，以此类推。在上面这个例子中，我们用以下这句 `SQL` 可以达到完全一样的效果：

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

函数

既然数据库中有许多资料都是已数字的型态存在,一个很重要的用途就是要能够对这些数字做一些运算,例如将它们总合起来,或是找出它们的平均值。SQL 有提供一些这一类的函数。它们是:

AVG (平均)

COUNT (计数)

MAX (最大值)

MIN (最小值)

SUM (总合)

运用函数的语法是:

```
SELECT "函数名"("栏位名")
```

```
FROM "表格名"
```

举例来说,若我们要由我们的范例表格中求出 Sales 栏位的总合,

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

我们就键入,

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

结果:

```
SUM(Sales)
```

```
$2750
```

\$2750 代表所有 Sales 栏位的总合: $\$1500 + \$250 + \$300 + \700 .

除了函数的运用外,SQL 也可以做简单的数学运算,例如加(+)和减(-)。对于文字类的资料,SQL 也有好几个文字处理方面的函数,例如文字相连 (concatenation),文字修整 (trim),以及子字符串 (substring)。不同的数据库对这些函数有不同的语法,所以最好是参考您所用数据库的信息,来确定在那个数据库中,这些函数是如何被运用的。

COUNT

在上一页有提到, COUNT 是函数之一。由于它的使用广泛,我们在这里特别提出来讨论。

基本上， COUNT 让我们能够数出在表格中有多少笔资料被选出来。它的语法是：

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

举例来说，若我们要找出我们的范例表格中有几笔 store_name 栏不是空白的资料时，

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

我们就键入，

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

结果：

```
Count(store_name)  
4
```

"is not NULL" 是 "这个栏位不是空白" 的意思。

COUNT 和 DISTINCT 经常被合起来使用，目的是找出表格中有多少笔不同的资料 (至于这些资料实际上是什么并不重要)。举例来说，如果我们要找出我们的表格中有多少个不同的 store_name，我们就键入，

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

结果：

```
Count(DISTINCT store_name)  
3
```

GROUP BY

我们现在回到函数上。记得我们用 `SUM` 这个指令来算出所有的 `Sales` (营业额)吧！如果我们的需求变成是要算出每一间店 (`store_name`) 的营业额 (`sales`)，那怎么办呢？在这个情况下，我们要做到两件事：第一，我们对于 `store_name` 及 `Sales` 这两个栏位都要选出。第二，我们需要确认所有的 `sales` 都要依照各个 `store_name` 来分开算。这个语法为：

```
SELECT "栏位 1", SUM("栏位 2")
FROM "表格名"
GROUP BY "栏位 1"
```

在我们的范例上，

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

我们就键入，

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

结果：

```
store_name  SUM(Sales)
Los Angeles $1800
San Diego   $250
Boston      $700
```

当我们选不只一个栏位，且其中至少一个栏位有包含函数的运用时，我们就需要用到 `GROUP BY` 这个指令。在这个情况下，我们需要确定我们有 `GROUP BY` 所有其他的栏位。换句话说，除了有包括函数的栏位外，我们都需要将其放在 `GROUP BY` 的子句中。

HAVING

那我们如何对函数产生的值来设定条件呢？举例来说，我们可能只需要知道哪些店的营业额有超过 \$1,500。在这个情况下，我们不能使用 **WHERE** 的指令。那要怎么办呢？很幸运地，SQL 有提供一个 **HAVING** 的指令，而我们就可以用这个指令来达到这个目标。 **HAVING** 子句通常是在一个 SQL 句子的最后。一个含有 **HAVING** 子句的 SQL 并不一定要包含 **GROUP BY** 子句。**HAVING** 的语法如下：

```
SELECT "栏位 1", SUM("栏位 2")
FROM "表格名"
GROUP BY "栏位 1"
HAVING (函数条件)
```

请读者注意： 如果被 **SELECT** 的只有函数栏， 那就不需要 **GROUP BY** 子句。

在我们 **Store_Information** 表格这个例子中，

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

若我们要找出 **Sales** 大于 \$1,500 的 **store_name**，我们就键入，

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

结果:

```
store_name  SUM(Sales)
Los Angeles  $1800
```

ALIAS

接下来，我们讨论 **alias** (别名) 在 SQL 上的用处。最常用到的别名有两种：栏位别名及表

格别名。

简单地来说，栏位别名的目的是为了让 SQL 产生的结果易读。在之前的例子中，每当我们有营业额总合时，栏位名都是 SUM(sales)。虽然在这个情况下没有什么问题，可是如果这个栏位不是一个简单的总合，而是一个复杂的计算，那栏位名就没有这么易懂了。若我们用栏位别名的话，就可以确认结果中的栏位名是简单易懂的。

第二种别名是表格别名。要给一个表格取一个别名，只要在 FROM 子句中的表格名后空一格，然后再列出要用的表格别名就可以了。这在我们要用 SQL 由数个不同的表格中获取资料时是很方便的。这一点我们在之后谈到连接 (join) 时会看到。

我们先来看一下栏位别名和表格别名的语法：

```
SELECT "表格别名"."栏位 1" "栏位别名"  
FROM "表格名" "表格别名"
```

基本上，这两种别名都是放在它们要替代的物件后面，而它们中间由一个空白分开。我们继续使用 Store_Information 这个表格来做例子：

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

我们用跟 SQL GROUP BY 那一页一样的例子。这里的不同处是我们加上了栏位别名以及表格别名：

```
SELECT A1.store_name Store, SUM(A1.Sales) "Total Sales"  
FROM Store_Information A1  
GROUP BY A1.store_name
```

结果：

Store	Total Sales
Los Angeles	\$1800
San Diego	\$250
Boston	\$700

在结果中，资料本身没有不同。不同的是栏位的标题。这是运用栏位别名的结果。在第二个栏位上，原本我们的标题是 "Sum(Sales)"，而现在我们有一个很清楚的 "Total Sales"。很明显地，"Total Sales" 能够比 "Sum(Sales)" 更精确地阐述这个栏位的含意。用表格别名的好

处在这里并没有显现出来，不过这在 下一页就会很清楚了。

表格链接

现在我们介绍连接(join)的概念。要了解连接，我们需要用到许多我们之前已介绍过的指令。我们先假设我们有以下的两个表格，

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

而我们要知道每一区 (region_name) 的营业额 (sales)。Geography 这个表格告诉我们每一区有哪些店，而 Store_Information 告诉我们每一个店的营业额。若我们要知道每一区的营业额，我们需要将这两个不同表格中的资料串联起来。当我们仔细了解这两个表格后，我们会发现它们可经由一个相同的栏位，store_name，连接起来。我们先将 SQL 句列出，之后再讨论每一个子句的意义：

```
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
```

结果:

```
REGION  SALES
East     $700
West     $2050
```

在第一行中，我们告诉 SQL 去选出两个栏位：第一个栏位是 Geography 表格中的 region_name 栏位 (我们取了一个别名叫做 REGION)；第二个栏位是 Store_Information 表格中的 sales 栏位 (别名为 SALES)。请注意在这里我们有用到表格别名：Geography 表格的别名是 A1，Store_Information 表格的别名是 A2。若我们没有用表格别名的话，第一行

就会变成

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

很明显地，这就复杂多了。在这里我们可以看到表格别名的功用：它能让 SQL 句容易被了解，尤其是这个 SQL 句含盖好几个不同的表格时。

接下来我们看第三行，就是 WHERE 子句。这是我们阐述连接条件的地方。在这里，我们要确认 Geography 表格中 store_name 栏位的值与 Store_Information 表格中 store_name 栏位的值是相等的。这个 WHERE 子句是一个连接的灵魂人物，因为它的角色是确定两个表格之间的连接是正确的。如果 WHERE 子句是错误的，我们就极可能得到一个笛卡儿连接 (Cartesian join)。笛卡儿连接会造成我们得到所有两个表格每两行之间所有可能的组合。在这个例子中，笛卡儿连接会让我们得到 $4 \times 4 = 16$ 行的结果。

外部链接

之前我们看到的左连接 (left join)，又称内部连接 (inner join)。在这个情况下，要两个表格内都有同样的值，那一笔资料才会被选出。那如果我们想要列出一个表格中每一笔的资料，无论它的值在另一个表格中有没有出现，那该怎么办呢？在这个时候，我们就需要用到 SQL OUTER JOIN (外部连接) 的指令。

外部连接的语法是依数据库的不同而有所不同的。举例来说，在 Oracle 上，我们会在 WHERE 子句中要选出所有资料的那个表格之后加上一个 "(+)" 来代表说这个表格中的所有资料我们都要。

假设我们有以下的两个表格：

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

我们需要知道每一间店的营业额。如果我们用一个普通的连接，我们将会漏失掉 'New York'

这个店，因为它并不存在于 `Store_Information` 这个表格。所以，在这个情况下，我们需要用外部连接来串联这两个表格：

```
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
```

我们在这里是使用了 `Oracle` 的外部连接语法。

结果:

store_name	SALES
Boston	\$700
New York	
Los Angeles	\$1800
San Diego	\$250

请注意：当第二个表格没有相对的资料时，`SQL` 会传回 `NULL` 值。在这一个例子中，'New York' 并不存在于 `Store_Information` 表格，所以它的 "SALES" 栏位是 `NULL`。

CONCATENATE

有的时候，我们需要将由不同栏位获得的资料串连在一起。每一种数据库都有提供方法来达到这个目的：

MySQL: `CONCAT()`

Oracle: `CONCAT()`, `||`

SQL Server: `+`

`CONCAT()` 的语法如下：

`CONCAT(字符串 1, 字符串 2, 字符串 3, ...)`: 将字符串 1、字符串 2、字符串 3，等字符串连在一起。请注意，`Oracle` 的 `CONCAT()` 只允许两个参数；换言之，一次只能将两个字符串串连起来。不过，在 `Oracle` 中，我们可以用 `||` 来一次串连多个字符串。

来看一个例子。假设我们有以下的表格：

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';
```

结果:

'EastBoston'

例子 2:

Oracle:

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

结果:

'East Boston'

例子 3:

SQL Server:

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

结果:

'East Boston'

SUBSTRING

SQL 中的 `substring` 函数是用来抓出一个栏位资料中的其中一部分。这个函数的名称在不同的数据库中不完全一样:

MySQL: `SUBSTR()`, `SUBSTRING()`

Oracle: `SUBSTR()`

SQL Server: SUBSTRING()

最常用到的方式如下 (在这里我们用 SUBSTR()为例):

SUBSTR(str,pos): 由<str>中, 选出所有从第<pos>位置开始的字符。请注意, 这个语法不适用于 SQL Server 上。

SUBSTR(str,pos,len): 由<str>中的第<pos>位置开始, 选出接下去的<len>个字符。

假设我们有以下的表格:

Geography 表格

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

例 1:

```
SELECT SUBSTR(store_name, 3)
FROM Geography
WHERE store_name = 'Los Angeles';
```

结果:

's Angeles'

例 2:

```
SELECT SUBSTR(store_name,2,4)
FROM Geography
WHERE store_name = 'San Diego';
```

结果:

'an D'

TRIM

SQL 中的 TRIM 函数是用来移除掉一个字符串中的字头或字尾。最常见的用途是移除字首或字尾的空白。这个函数在不同的数据库中有不同的名称:

MySQL: TRIM(), RTRIM(), LTRIM()

Oracle: RTRIM(), LTRIM()

SQL Server: RTRIM(), LTRIM()

各种 trim 函数的语法如下：

TRIM([[位置] [要移除的字符串] FROM] 字符串): [位置] 的可能值为 LEADING (起头), TRAILING (结尾), or BOTH (起头及结尾)。这个函数将把 [要移除的字符串] 从字符串的起头、结尾，或是起头及结尾移除。如果我们没有列出 [要移除的字符串] 是什么的话，那空白就会被移除。

LTRIM(字符串): 将所有字符串起头的空白移除。

RTRIM(字符串): 将所有字符串结尾的空白移除。

例 1:

```
SELECT TRIM(' Sample ');
```

结果:

'Sample'

例 2:

```
SELECT LTRIM(' Sample ');
```

结果:

'Sample '

例 3:

```
SELECT RTRIM(' Sample ');
```

结果:

' Sample'

表格处理

CREATE TABLE

表格是数据库中储存资料的基本架构。在绝大部份的情况下，数据库厂商不可能知道您需要如何储存您的资料，所以通常您需要自己在数据库中建立表格。虽然许多数据库工具可以让您在不需用到 SQL 的情况下建立表格，不过由于表格是一个最基本的架构，我们决定包括 CREATE TABLE 的语法在这个网站中。

在我们跳入 CREATE TABLE 的语法之前，我们最好先对表格这个东西有些多一点的了解。表格被分为栏位 (column) 及列位 (row)。每一列代表一笔资料，而每一栏代表一笔资料的一部份。举例来说，如果我们有一个记载顾客资料的表格，那栏位就有可能包括姓、名、地址、城市、国家、生日...等等。当我们对表格下定义时，我们需要注明栏位的标题，以及那个栏位的资料种类。

那，资料种类是什么呢？资料可能是以许多不同的形式存在的。它可能是一个整数 (例如 1)、一个实数 (例如 0.55)、一个字符串 (例如 'sql')、一个日期/时间 (例如 '2000-JAN-25 03:22:22')、或甚至是以二进制法 (binary) 的状态存在。当我们在对一个表格下定义时，我们需要对每一个栏位的资料种类下定义。(例如 '姓' 这个栏位的资料种类是 char(50)——代表这是一个 50 个字符的字符串)。我们需要注意的一点是不同的数据库有不同的资料种类，所以在对表格做出定义之前最好先参考一下数据库本身的说明。

CREATE TABLE 的语法是：

```
CREATE TABLE "表格名"  
("栏位 1" "栏位 1 资料种类",  
"栏位 2" "栏位 2 资料种类",  
...)
```

若我们要建立我们上面提过的顾客表格，我们就键入以下的 SQL：

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

CONSTRAINT

我们可以限制哪一些资料可以存入表格中。这些限制可以在表格初创时藉由 `CREATE TABLE` 语句来指定，或是之后藉由 `ALTER TABLE` 语句来指定。

常见的限制有以下几种：

`NOT NULL`

`UNIQUE`

`CHECK`

主键 (Primary Key)

外来键 (Foreign Key)

以下对这几种限制分别做个介绍：

NOT NULL

在没有做出任何限制的情况下，一个栏位是允许有 `NULL` 值得。如果我们不允许一个栏位含有 `NULL` 值，我们就需要对那个栏位做出 `NOT NULL` 的指定。

举例来说，在以下的语句中，

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

"SID" 和 "Last_Name" 这两个栏位是不允许有 `NULL` 值，而 "First_Name" 这个栏位是可以有 `NULL` 值得。

UNIQUE

`UNIQUE` 限制是保证一个栏位中的所有资料都是有不一样的值。

举例来说，在以下的语句中，

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

"SID" 栏位不能有重复值存在, 而 "Last_Name" 及 "First_Name" 这两个栏位则是允许有重复值存在。

请注意, 一个被指定为主键的栏位也一定会含有 **UNIQUE** 的特性。相对来说, 一个 **UNIQUE** 的栏位并不一定会是一个主键。

CHECK

CHECK 限制是保证一个栏位中的所有资料都是符合某些条件。

举例来说, 在以下的语句中,

```
CREATE TABLE Customer
(SID integer CHECK (SID > 0),
Last_Name varchar (30),
First_Name varchar(30));
```

"SID" 栏只能包含大于 0 的整数。

请注意, **CHECK** 限制目前尚未被执行于 **MySQL** 数据库上。

主键 **and** 外来键 将于下两页中讨论。

主键

主键 (**Primary Key**) 中的每一笔资料都是表格中的唯一值。换言之, 它是用来独一无二地确认一个表格中的每一行资料。主键可以是原本资料内的一个栏位, 或是一个人造栏位 (与原本资料没有关系的栏位)。主键可以包含一或多个栏位。当主键包含多个栏位时, 称为组合键 (**Composite Key**)。

主键可以在建置新表格时设定 (运用 **CREATE TABLE** 语句), 或是以改变现有的表格架构方式设定 (运用 **ALTER TABLE**)。

以下举几个在建置新表格时设定主键的方式：

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));
```

以下则是以改变现有表格架构来设定主键的方式：

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);
```

请注意，在用 ALTER TABLE 语句来添加主键之前，我们需要确认被用来当做主键的栏位是设定为『NOT NULL』；也就是说，那个栏位一定不能没有资料。

外来键

外来键是一个(或数个)指向另外一个表格主键的栏位。外来键的目的是确定资料的参考完整性(referential integrity)。换言之，只有被准许的资料值才会被存入数据库内。

举例来说，假设我们有两个表格：一个 CUSTOMER 表格，里面记录了所有顾客的资料；另一个 ORDERS 表格，里面记录了所有顾客订购的资料。在这里的一个限制，就是所有的订购资料中的顾客，都一定是要跟在 CUSTOMER 表格中存在。在这里，我们就会在 ORDERS 表格中设定一个外来键，而这个外来键是指向 CUSTOMER 表格中的主键。这样一来，我们就可以确定所有在 ORDERS 表格中的顾客都存在 CUSTOMER 表格中。换句话说，ORDERS 表格之中，不能有任何顾客是不存在于 CUSTOMER 表格中的资料。

这两个表格的结构将会是如下：

CUSTOMER 表格

栏位名	性质
SID	主键
Last_Name	
First_Name	

ORDERS 表格

栏位名	性质
Order_ID	主键
Order_Date	
Customer_SID	外来键
Amount	

在以上的例子中，ORDERS 表格中的 customer_SID 栏位是一个指向 CUSTOMERS 表格中 SID 栏位的外来键。

以下列出几个在建置 ORDERS 表格时指定外来键的方式：

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);
```

以下的例子则是藉着改变表格架构来指定外来键。这里假设 **ORDERS** 表格已经被建置，而外来键尚未被指定：

MySQL:

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

Oracle:

```
ALTER TABLE ORDERS  
ADD (CONSTRAINT fk_orders1) FOREIGN KEY (customer_sid) REFERENCES  
CUSTOMER(sid);
```

SQL Server:

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

CREATE VIEW

视观表 (Views) 可以被当作是虚拟表格。它跟表格的不同是，表格中有实际储存资料，而视观表是建立在表格之上的一个架构，它本身并不实际储存资料。

建立一个视观表的语法如下：

```
CREATE VIEW "VIEW_NAME" AS "SQL 语句"
```

"SQL 语句" 可以是任何一个我们在这个教材中有提到的 SQL。

来看一个例子。假设我们有以下的表格：

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

若要在这个表格上建立一个包括 First_Name, Last_Name, 和 Country 这三个栏位的视图表，我们就打入，

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

现在，我们就有一个叫做 V_Customer 的视图表：

```
View V_Customer
(First_Name char(50),
Last_Name char(50),
Country char(25))
```

我们也可以用视图表来连接两个表格。在这个情况下，使用者就可以直接由一个视图表中找出她要的信息，而不需要由两个不同的表格中去做一次连接的动作。假设有以下的两个表格：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

我们就可以用以下的指令来建一个包括每个地区 (region) 销售额 (sales) 的视图表：

```
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
```

这就给我们有一个名为 V_REGION_SALES 的视图表。这个视图表包含不同地区的销售哦。如果我们要从这个视图表中获取资料，我们就打入，

```
SELECT * FROM V_REGION_SALES
```

结果：

```
REGION  SALES
East    $700
```

CREATE INDEX

索引 (Index) 可以帮助我们从表格中快速地找到需要的资料。举例来说, 假设我们要在一本园艺书中找如何种植青椒的讯息。若这本书没有索引的话, 那我们是必须要从头开始读, 直到我们找到有关种直青椒的地方为止。若这本书有索引的话, 我们就可以先去索引找出种植青椒的信息是在哪一页, 然后直接到那一页去阅读。很明显地, 运用索引是一种有效且省时的方式。

从数据库表格中寻找资料也是同样的原理。如果一个表格没有索引的话, 数据库系统就需要将整个表格的资料读出 (这个过程叫做'table scan')。若有适当的索引存在, 数据库系统就可以先由这个索引去找出需要的资料是在表格的什么地方, 然后直接去那些地方抓资料。这样子速度就快多了。

因此, 在表格上建立索引是一件有利于系统效率的事。一个索引可以涵盖一或多个栏位。建立索引的语法如下:

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

现在假设我们有以下这个表格,

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

若我们要在 Last_Name 这个栏位上建一个索引, 我们就打入以下的指令,

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

我们要在 City 及 Country 这两个栏位上建一个索引, 我们就打入以下的指令,

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

索引的命名并没有一个固定的方式。通常会用的方式是在名称前加一个字首，例如 "IDX_"，来避免与数据库中的其他物件混淆。另外，在索引名之内包括表格名及栏位名也是一个好的方式。

请读者注意，每个数据库会有它本身的 CREATE INDEX 语法，而不同数据库的语法会有不同。因此，在下指令前，请先由数据库使用手册中确认正确的语法。

ALTER TABLE

在表格被建立在数据库中后，我们常常会发现，这个表格的结构需要有所改变。常见的改变如下：

- 加一个栏位
 - 删去一个栏位
 - 改变栏位名称
 - 改变栏位的资料种类
- 以上列出的改变并不是所有可能的改变。ALTER TABLE 也可以被用来作其他的改变，例如改变主键定义。

ALTER TABLE 的语法如下：

```
ALTER TABLE "table_name"  
[改变方式]
```

[改变方式] 的详细写法会依我们想要达到的目标而有所不同。再以上列出的改变中，[改变方式] 如下：

- 加一个栏位: ADD "栏位 1" "栏位 1 资料种类"
 - 删去一个栏位: DROP "栏位 1"
 - 改变栏位名称: CHANGE "原本栏位名" "新栏位名" "新栏位名资料种类"
 - 改变栏位的资料种类: MODIFY "栏位 1" "新资料种类"
- 以下我们用在 CREATE TABLE 一页建出的 customer 表格来当作例子：

customer 表格

栏位名称	资料种类
First_Name	char(50)
Last_Name	char(50)
Address	char(50)
City	char(50)
Country	char(25)

Birth_Date	date
------------	------

第一，我们要加入一个叫做 "gender" 的栏位。这可以用以下的指令达成：

```
ALTER table customer add Gender char(1)
```

这个指令执行后的表格架构是：

customer 表格

栏位名称	资料种类
First_Name	char(50)
Last_Name	char(50)
Address	char(50)
City	char(50)
Country	char(25)
Birth_Date	date
Gender	char(1)

接下来，我们要把 "Address" 栏位改名为 "Addr"。这可以用以下的指令达成：

```
ALTER table customer change Address Addr char(50)
```

这个指令执行后的表格架构是：

customer 表格

栏位名称	资料种类
First_Name	char(50)
Last_Name	char(50)
Addr	char(50)
City	char(50)
Country	char(25)
Birth_Date	date
Gender	char(1)

再来，我们要将 "Addr" 栏位的资料种类改为 char(30)。这可以用以下的指令达成：

```
ALTER table customer modify Addr char(30)
```

这个指令执行后的表格架构是：

customer 表格

栏位名称	资料种类
First_Name	char(50)
Last_Name	char(50)
Addr	char(30)
City	char(50)

Country	char(25)
Birth_Date	date
Gender	char(1)

最后，我们要删除 "Gender" 栏位。这可以用以下的指令达成：

```
ALTER table customer drop Gender
```

这个指令执行后的表格架构是：

customer 表格

栏位名称	资料种类
First_Name	char(50)
Last_Name	char(50)
Addr	char(30)
City	char(50)
Country	char(25)
Birth_Date	date

DROP TABLE

有时候我们会决定我们需要从数据库中清除一个表格。事实上，如果我们不能这样做的话，那将会是一个很大的问题，因为数据库管理师 (Database Administrator -- DBA) 势必无法对数据库做有效率的管理。还好，SQL 有提供一个 DROP TABLE 的语法来让我们清除表格。DROP TABLE 的语法是：

```
DROP TABLE "表格名"
```

我们如果要清除在上一页中建立的顾客表格，我们就键入：

```
DROP TABLE customer.
```

TRUNCATE TABLE

有时候我们会需要清除一个表格中的所有资料。要达到这个目的，一种方式是在 上一页看到 的 DROP TABLE 指令。不过这样整个表格就消失，而无法再被用了。另一种方式

就是运用 `TRUNCATE TABLE` 的指令。在这个指令之下，表格中的资料会完全消失，可是表格本身会继续存在。 `TRUNCATE TABLE` 的语法为下：

```
TRUNCATE TABLE "表格名"
```

所以，我们如果要清除在 `SQL Create Table` 那一页建立的顾客表格之内的资料，我们就键入：

```
TRUNCATE TABLE customer.
```

INSERT INTO

到目前为止，我们学到了如何将资料由表格中取出。但是这些资料是如何进入这些表格的呢？这就是这一页 (`INSERT INTO`) 和下一页 (`UPDATE`) 要讨论的。

基本上，我们有两种作法可以将资料输入表格中内。一种是一次输入一笔，另一种是一次输入好几笔。我们先来看一次输入一笔的方式。

依照惯例，我们先介绍语法。一次输入一笔资料的语法如下：

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

假设我们有一个架构如下的表格：

Store_Information 表格

Column Name	Data Type
store_name	char(50)
Sales	float
Date	datetime

而我们要加以下的这一笔资料进去这个表格：在 January 10, 1999, Los Angeles 店有 \$900 的营业额。我们就打入以下的 `SQL` 语句：

```
INSERT INTO Store_Information (store_name, Sales, Date)
VALUES ('Los Angeles', 900, 'Jan-10-1999')
```

第二种 `INSERT INTO` 能够让我们一次输入多笔的资料。跟上面刚的例子不同的是，现在我们要用 `SELECT` 指令来指明要输入表格的资料。如果您想说，这是不是说资料是从另一个表格来的，那您就想对了。一次输入多笔的资料的语法是：

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

以上的语法是最基本的。这整句 SQL 也可以含有 WHERE、GROUP BY、及 HAVING 等子句，以及表格连接及别名等等。

举例来说，若我们想要将 1998 年的营业额资料放入 Store_Information 表格，而我们知道资料的来源是可以由 Sales_Information 表格取得的话，那我们就可以键入以下的 SQL：

```
INSERT INTO Store_Information (store_name, Sales, Date)
SELECT store_name, Sales, Date
FROM Sales_Information
WHERE Year(Date) = 1998
```

在这里，我用了 SQL Server 中的函数来由日期中找出年。不同的数据库会有不同的语法。举个例子来说，在 Oracle 上，您将会使用 WHERE to_char(date,'yyyy')=1998。

UPDATE

我们有时候可能会需要修改表格中的资料。在这个时候，我们就需要用到 UPDATE 指令。这个指令的语法是：

```
UPDATE "表格名"
SET "栏位 1" = [新值]
WHERE {条件}
```

最容易了解这个语法的方式是透过一个例子。假设我们有以下的表格：

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

我们发现说 Los Angeles 在 01/08/1999 的营业额实际上是 \$500，而不是表格中所储存的 \$300，因此我们用以下的 SQL 来修改那一笔资料：


```
UPDATE Store_Information
SET Sales = 500
WHERE store_name = "Los Angeles"
AND Date = "Jan-08-1999"
```

现在表格的内容变成：

Store_Information 表格

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

在这个例子中，只有一笔资料符合 **WHERE** 子句中的条件。如果有多笔资料符合条件的话，每一笔符合条件的资料都会被修改的。

我们也可以同时修改好几个栏位。这语法如下：

```
UPDATE "表格"
SET "栏位 1" = [值 1], "栏位 2" = [值 2]
WHERE {条件}
```

DELETE FROM

在某些情况下，我们会需要直接由数据库中去删除一些资料。这可以藉由 **DELETE FROM** 指令来达成。它的语法是：

```
DELETE FROM "表格名"
WHERE {条件}
```

以下我们用个实例说明。假设我们有以下这个表格：

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
--------	-------	-------------

而我们需要将有关 Los Angeles 的资料全部去除。在这里我们可以用以下的 SQL 来达到这个目的:

```
DELETE FROM Store_Information
WHERE store_name = "Los Angeles"
```

现在表格的内容变成:

Store_Information 表格

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

进阶 SQL

在这一部分, 我们将介绍以下的 SQL 概念及关键字:

SQL UNION

SQL UNION ALL

SQL INTERSECT

SQL MINUS

SQL Subquery

SQL EXISTS

SQL CASE

我们并介绍如何用 SQL 来做出以下的运算:

排名 (Rank)

中位数 (Median)

累积总计 (Running Total)

总合百分比 (Percent to Total)

累积总合百分比 (Cumulative Percent to Total)

UNION

UNION 指令的目的是将两个 SQL 语句的结果合并起来。从这个角度来看，UNION 跟 JOIN 有些许类似，因为这两个指令都可以由多个表格中撷取资料。UNION 的一个限制是两个 SQL 语句所产生的栏位需要是同样的资料种类。另外，当我们用 UNION 这个指令时，我们只会看到不同的资料值 (类似 SELECT DISTINCT)。

UNION 的语法如下：

```
[SQL 语句 1]
UNION
[SQL 语句 2]
```

假设我们有以下的两个表格，

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

而我们要找出来所有有营业额 (sales) 的日子。要达到这个目的，我们用以下的 SQL 语句：

```
SELECT Date FROM Store_Information
UNION
SELECT Date FROM Internet_Sales
```

结果:

```
Date
Jan-05-1999
Jan-07-1999
Jan-08-1999
Jan-10-1999
Jan-11-1999
Jan-12-1999
```

有一点值得注意的是，如果我们在任何一个 SQL 语句（或是两句都一起）用 "SELECT DISTINCT Date" 的话，那我们会得到完全一样的结果。

UNION ALL

UNION ALL 这个指令的目的也是要将两个 SQL 语句的结果合并在一起。UNION ALL 和 UNION 不同之处在于 UNION ALL 会将每一笔符合条件的资料都列出来，无论资料值有无重复。

UNION ALL 的语法如下：

```
[SQL 语句 1]
UNION ALL
[SQL 语句 2]
```

我们用和上一页同样的例子来显示出 UNION ALL 和 UNION 的不同。同样假设我们有以下两个表格，

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

而我们要找出有店面营业额以及网络营业额的日子。要达到这个目的，我们用以下的 SQL 语句：

```
SELECT Date FROM Store_Information
UNION ALL
SELECT Date FROM Internet_Sales
```

结果:

Date
Jan-05-1999
Jan-07-1999
Jan-08-1999
Jan-08-1999
Jan-07-1999
Jan-10-1999
Jan-11-1999
Jan-12-1999

INTERSECT

和 UNION 指令类似， INTERSECT 也是对两个 SQL 语句所产生的结果做处理的。不同的地方是， UNION 基本上是一个 OR (如果这个值存在于第一句或是第二句，它就会被选出)，而 INTERSECT 则比较像 AND (这个值要存在于第一句和第二句才会被选出)。UNION 是联集，而 INTERSECT 是交集。

INTERSECT 的语法如下：

[SQL 语句 1]
INTERSECT
[SQL 语句 2]

假设我们有以下的两个表格，
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

而我们要找出哪几天有店面交易和网络交易。要达到这个目的，我们用以下的 SQL 语句：

```
SELECT Date FROM Store_Information
INTERSECT
SELECT Date FROM Internet_Sales
```

结果：

```
Date
Jan-07-1999
```

请注意，在 INTERSECT 指令下，不同的值只会被列出一次。

MINUS

MINUS 指令是运用在两个 SQL 语句上。它先找出第一个 SQL 语句所产生的结果，然后看这些结果有没有在第二个 SQL 语句的结果中。如果有的话，那这一笔资料就被去除，而不会在最后的結果中出现。如果第二个 SQL 语句所产生的结果并没有存在于第一个 SQL 语句所产生的结果内，那这笔资料就被抛弃。

MINUS 的语法如下：

```
[SQL 语句 1]
MINUS
[SQL 语句 2]
```

我们继续使用一样的例子：

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

而我们要知道有哪几天是有店面营业额而没有网络营业额的。要达到这个目的，我们用以下的 SQL 语句：

```
SELECT Date FROM Store_Information
MINUS
SELECT Date FROM Internet_Sales
```

结果：

```
Date
Jan-05-1999
Jan-08-1999
```

"Jan-05-1999", "Jan-07-1999", and "Jan-08-1999" 是 "SELECT Date FROM Store_Information" 所产生的结果。在这里面，"Jan-07-1999" 是存在于 "SELECT Date FROM Internet_Sales" 所产生的结果中。因此 "Jan-07-1999" 并不在最后的结果中。

请注意，在 MINUS 指令下，不同的值只会被列出一次。

子查询

我们可以在一个 SQL 语句中放入另一个 SQL 语句。当我们在 WHERE 子句或 HAVING 子句中插入另一个 SQL 语句时，我们就有一个子查询 (subquery) 的架构。子查询的作用是什么呢？第一，它可以被用来连接表格。另外，有的时候子查询是唯一能够连接两个表格的方式。

子查询的语法如下：

```
SELECT "栏位 1"
FROM "表格"
WHERE "栏位 2" [比较运算符]
(SELECT "栏位 1"
FROM "表格"
WHERE [条件])
```

[比较运算符] 可以是相等的运算符，例如 =, >, <, >=, <=。这也可以是一个对文字的运算符，例如 "LIKE"。绿色的部分代表外查询，红色的部分代表内查询。

我们就用刚刚在阐述 SQL 连接时用过的例子：

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

我们要运用 subquery 来找出所有在西部的店的营业额。我们可以用下面的 SQL 来达到我们的目的：

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

结果：

```
SUM(Sales)
2050
```

在这个例子中，我们并没有直接将两个表格连接起来，然后由此直接算出每一间西区店面的营业额。我们做的是先找出哪些店是在西区的，然后再算出这些店的营业额总共是多少。

在以上的例子，内部查询本身与外部查询没有关系。这一类的子查询称为『简单子查询』(Simple Subquery)。如果内部查询是要利用到外部查询提到的表格中的栏位，那这个字查询就被称为『相关子查询』(Correlated Subquery)。以下是一个相关子查询的例子：

```
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)
```


红色部分即是外部查询提到的表格中的栏位。

EXISTS

在上一页中，我们用 `IN` 来连接内查询和外查询。另外有数个方式，例如 `>`，`<`，及 `=`，都可以用来连接内查询和外查询。`EXISTS` 也是其中一种方式。这一页我们将讨论 `EXISTS` 的用法。

基本上，`EXISTS` 是用来测试内查询有没有产生任何结果。如果有的话，系统就会执行外查询中的 `SQL`。若是没有的话，那整个 `SQL` 语句就不会产生任何结果。

`EXISTS` 的语法是：

```
SELECT "栏位 1"
FROM "表格 1"
WHERE EXISTS
(SELECT *
FROM "表格 2"
WHERE [条件])
```

在内查询中，我们并不一定要用 `*` 来选出所有的栏位。我们也可以选择表格 2 中的任何栏位。这两种做法最后的结果是一样的。

来看一个例子。假设我们有以下的两个表格：

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

而我们打入的 `SQL` 是：

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

我们会得到以下的答案：

```
SUM(Sales)
2750
```

乍看之下，这个答案似乎不太正确，因为内查询有包含一个 `[region_name = 'West']` 的条件，可是最后的答案并没有包含这个条件。实际上，这并没有问题。在这个例子中，内查询产生了超过一笔的资料，所以 `EXISTS` 的条件成立，所以外查询被执行。而外查询本身并没有包含 `[region_name = 'West']` 这个条件。

CASE

`CASE` 是 SQL 用来做为 `if-then-else` 之类逻辑的关键字。 `CASE` 的语法如下：

```
SELECT CASE ("栏位名")
  WHEN "条件 1" THEN "结果 1"
  WHEN "条件 2" THEN "结果 2"
  ...
  [ELSE "结果 N"]
END
FROM "表格名"
```

"条件" 可以是一个数值或是公式。 `ELSE` 子句则并不是必须的。

在我们的 `Store_Information` 中

`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

若我们要将 'Los Angeles' 的 Sales 数值乘以 2, 以及将 'San Diego' 的 Sales 数值乘以 1.5, 我们就键入以下的 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 那个栏位的栏位名。

结果:

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

算排名

列出每一行的排名是一个常见的需求, 可惜 SQL 并没有一个很直接的方式达到这个需求。要以 SQL 列出排名, 基本的概念是要做一个表格自我连结 (self join), 将结果依序列出, 然后算出每一行之前 (包含那一行本身) 有多少行数。这样讲读者听得可能有点困惑, 所以最好的方式是用一个实例来介绍。假设我们有以下的表格:

Total_Sales 表格

Name	Sales
John	10
Jennifer	15
Stella	20
Sophia	40

Greg	50
Jeff	20

要找出每一行的排名，我们就打入以下的 SQL 语句：

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

结果：

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

我们先来看 WHERE 子句。在字句的第一部分 (a1.Sales <= a2.Sales)，我们算出有多少笔资料 Sales 栏位的值是自己本身的价值或是相等。如果在 Sales 栏位中没有同样大小的资料，那这部分的 WHERE 子句本身就可以产生出正确的排名。

子句的第二部分，(a1.Sales=a2.Sales and a1.Name = a2.Name)，则是让我们在 Sales 栏位中有同样大小的资料时 (像 Stella 及 Jeff 这两笔资料)，仍然能够产生正确的排名。

算中位数

要算出中位数，我们必须能够达成以下几个目标：

将资料依序排出，并找出每一行资料的排名。

找出『中间』的排名为何。举例来说，如果总共有 9 笔资料，那中间排名就是 5 (有 4 笔资料比第 5 笔资料大，有 4 笔资料比第 5 笔资料小)。

找出中间排名资料的值。

来看看以下的例子。假设我们有以下的表格：

Total_Sales 表格

Name	Sales
John	10
Jennifer	15

Stella	20
Sophia	40
Greg	50
Jeff	20

要找出中位数，我们就键入：

```
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 OR (a1.Sales=a2.Sales AND a1.Name <= a2.Name)
group by a1.Name, a1.Sales
order by a1.Sales desc) a3
WHERE Rank = (SELECT (COUNT(*)+1) DIV 2 FROM Total_Sales);
```

结果:

Median
20

读者将会发现，第 2 行到第 6 行是跟产生 排名 的语句完全一样。第 7 行则是算出中间的排名。DIV 是在 MySQL 中算出商的方式。在不同的数据库中会有不同的方式求商。第 1 行则是列出排名中间的资料值。

算累积总计

算出累积总计是一个常见的需求，可惜以 SQL 并没有一个很直接的方式达到这个需求。要以 SQL 算出累积总计，基本上的概念与列出排名类似：第一是先做个表格自我连结 (self join)，然后将结果依序列出。在做列出排名时，我们算出每一行之前 (包含那一行本身) 有多少行数；而在做累积总计时，我们则是算出每一行之前 (包含那一行本身) 的总合。

来看看以下的例子。假设我们有以下的表格：

Total_Sales 表格

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

要算出累积总计，我们就键入：

```
SELECT a1.Name, a1.Sales, SUM(a2.Sales) Running_Total
FROM Total_Sales a1, Total_Sales a2
```

```
WHERE a1.Sales <= a2.sales or (a1.Sales=a2.Sales and a1.Name = a2.Name)
GROUP BY a1.Name, a1.Sales
ORDER BY a1.Sales DESC, a1.Name DESC;
```

结果:

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

在以上的 SQL 语句中， WHERE 子句和 ORDER BY 子句让我们能够在有重复值时能够算出正确的累积总计。

算总合百分比

要用 SQL 算出总合百分比，我们需要用到算排名和累积总计的概念，以及运用子查询的做法。在这里，我们把子查询放在外部查询的 SELECT 子句中。让我们来看以下的例子：

Total_Sales 表格

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

要算出总合百分比，我们键入：

```
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 or (a1.Sales=a2.Sales and a1.Name = a2.Name)
GROUP BY a1.Name, a1.Sales
ORDER BY a1.Sales DESC, a1.Name DESC;
```

结果:

Name	Sales	Pct_To_Total
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" 这一段子查询是用来算出总合。总合算出后，我们就能够将每一行一一除以总合来求出每一行的总合百分比。

算累积总合百分比

要用 SQL 累积总合百分比算出，我们运用类似总合百分比的概念。两者的不同处在于在这个情况下，我们要算出到目前为止的累积总合是所有总合的百分之几，而不是光看每一笔资料是所有总合的百分之几。让我们来看看以下的例子：

Total_Sales 表格

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

要算出累积总合百分比，我们键入：

```
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 or (a1.Sales=a2.Sales and a1.Name = a2.Name)
GROUP BY a1.Name, a1.Sales
ORDER BY a1.Sales DESC, a1.Name DESC;
```

结果:

Name	Sales	Pct_To_Total
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" 这一段子查询是用来算出总合。我们接下来用累积总计 "SUM(a2.Sales)" 除以总合来求出每一行的累积总合百分比。

SQL 语法

Select

```
SELECT "栏位" FROM "表格名"
```

Distinct

```
SELECT DISTINCT "栏位"  
FROM "表格名"
```

Where

```
SELECT "栏位"  
FROM "表格名"  
WHERE "condition"
```

And/Or

```
SELECT "栏位"  
FROM "表格名"  
WHERE "简单条件"  
{[AND|OR] "简单条件"}+
```

In

```
SELECT "栏位"  
FROM "表格名"  
WHERE "栏位" IN ('值 1', '值 2', ...)
```

Between

```
SELECT "栏位"  
FROM "表格名"  
WHERE "栏位" BETWEEN '值 1' AND '值 2'
```


Like

```
SELECT "栏位"  
FROM "表格名"  
WHERE "栏位" LIKE {模式}
```

Order By

```
SELECT "栏位"  
FROM "表格名"  
[WHERE "条件"]  
ORDER BY "栏位" [ASC, DESC]
```

Count

```
SELECT COUNT("栏位")  
FROM "表格名"
```

Group By

```
SELECT "栏位 1", SUM("栏位 2")  
FROM "表格名"  
GROUP BY "栏位 1"
```

Having

```
SELECT "栏位 1", SUM("栏位 2")  
FROM "表格名"  
GROUP BY "栏位 1"  
HAVING (函数条件)
```

Create Table

```
CREATE TABLE "表格名"  
("栏位 1" "栏位 1 资料种类",  
"栏位 2" "栏位 2 资料种类",  
... )
```

Drop Table

```
DROP TABLE "表格名"
```

Truncate Table

```
TRUNCATE TABLE "表格名"
```

Insert Into

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

Update

```
UPDATE "表格名"  
SET "栏位 1" = [新值]  
WHERE {条件}
```

```
Delete From  
DELETE FROM "表格名"  
WHERE {条件}
```

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