雲端資料庫

PostgreSQL9.0

Ricky Liao

CHAPTER 1

PostgreSQL 9 概觀

本章內容大綱如下：

- PostgreSQL9 介紹
- 安裝資料庫軟體
- 開始使用 PostgreSQL9
- 設定資料庫遠端存取
- 連接 PostgresSQL9 伺服器
- 圖型化管理工具
- psql 命令模式管理工具
- 資料庫伺服器密碼檔
- 資料庫遠端存取控制檔
- 資料庫存取故障排除

前言

PostgreSQL9 是一套功能非常強大且複雜的資料庫管理系統；當我們第一次使用時資料庫時，不僅可使用圖型化介面操作，也可透過命令模式 psql 工具。而它提供線上查詢功能，可隨時查詢命令查用法來加快操作。

PostgreSQL9 介紹

PostgreSQL9 是一套先進的資料庫管理系統，可安裝在各種作業系統上運行。

其中最吸引人們使用的主因在於 PostgreSQL9 是一套開源資料庫管理系統，代表著安裝使用 PostgreSQL9 是不需要任何授權，我們並不需要付任何費用 PostgreSQL9 經歷了長時間的演變，開始於在 UC Berkeley 的 Ingres 計劃，在開源資料管理系統中相當知名。它在其他開放原始碼資料庫系統（比如 MySQL 和 Firebird），和商業軟體如 Oracle、Sybase、IBM DB2 和 Microsoft SQL Server 之外，為使用者又提供了一種選擇；整體而言使用 PostgreSQL9 當做內部資料庫管系統將會大大降低公司成本。

PostgreSQL9 主要功能特色如下：

- 出色的資料庫軟體媲美 MS SQL2008。
- 主從式架構(Client-Server)。
- 可負載大量使用者同時寫入及讀取。
- 高效能、高安全性及高可用性。
- 高延展性及多種效能調校工具。

PostgreSQL9 有哪裏不一樣？

PostgreSQL9 開源專案遵循以下目標：

- 建立強大高品質且易於維護原始碼的資料庫軟體。
- 於企業中降低維護管理成本。
- 遵循 SQL 標準，可安裝並相容於各種作業平台。
- 高效能、高安全性及高可用性。

PostgreSQL9 最令人感到意外的事它相容於其它知名資料庫管理系統如 Oracle、SQL Server 和 MySQL。雖然 PostgreSQL9 與 MySQL 都是知名的開源資料庫管系統但功能卻大大的不同。

有哪些企業使用 PostgreSQL9 呢？擁有企業如 Apple, BASF, Genentech, IMDB.com, Skype, NTT, Yahoo。至目前為止 PostgreSQL9 每年至少超過 1 千萬次下載；PostgreSQL9 是眾多資料庫管理系統中最多人選擇的。

PostgreSQL 不尋常的名字導致一些讀者停下來嘗試拼讀它，特別是那些把 SQL 拼讀為"sequel"的人。PostgreSQL 開發者把它拼讀為"post-gress-Q-L"。它也經常被簡略唸為"postgres"。

PostgreSQL 經歷了長時間的演變，開始於在 UC Berkeley 的 Ingres 計劃。這個計劃的領導者 Michael Stonebraker 在 1982 年離開 Berkeley 去商業化 Ingres，但是最後還是返回了學術界。在 1985 年返回 Berkeley 之後，Stonebraker 開始了 post-Ingres 計劃來致力於在 1980 年代早期變得日益清楚的、當代資料庫系統的問題。Postgres 和 Ingres 的代碼函式庫開始（並保持）完全分離了。

結果的計劃叫 Postgres，標的是介入增加對型別的完全支援所需要的最小數目的特徵。這些特徵包括定義型別的能力，還有完全描述至今廣泛使用但由使用者完全維護著的聯繫的能力。在 Postgres 中資料庫"理解"聯繫，並可以使用規則以自然的方式在有關的表中檢索資訊。

從 1986 年開始專案組發表了一些描述系統基本原理的論文，並在 1988 年這項計劃建成並執行了一個原型版本。專案組在 1989 年六月向少數使用者發行了版本 1，隨後在 1990 年六月發行了帶有重寫後的規則系統的版本 2。1991 年的版本 3 再次重寫了規則系統，並增加了對多個儲存管理器和改進的查詢引擎的支援。在 1993 年就有大量的使用者存在了，並開始用對支援和特徵的要求淹沒這個計劃。在發行了主要作為最後清理的版本 4 之後計劃就終止了。

儘管Postgres計劃正式的終止了，BSD許可證(Berkeley在其下發行的Postgres）確使開放原始碼開發者獲得複本並進一步開發系統。在 1994 年兩個 UC Berkeley 大學的研究生，Andrew Yu 和 Jolly Chen，增加了一個 SQL 語言直譯器來替代早先的基於 Ingres 的 QUEL 系統，建立了 Postgres95。代碼隨後被發行到 web 上來在世界上找尋它自己的出路。在 1996 年計劃被重新命名了：為了反映資料庫的新 SQL 查詢語言，Postgres95 變成了 PostgreSQL。

第一次 PostgreSQL 發行形成了版本 6.0。隨後來自世界各地的一組資料庫開發者和志願者，透過 Internet 協作起來，維護著這套軟體。自從版本 6.0 之後，出現了很多後續發行，在系統中也出現了很多改進；在 2005 年 1 月 19 日，版本 8.0 成為當前發行。由 8.0 後，PostgreSQL 以原生(Native)的方式，執行於 Windows 視窗系統。

儘管許可證允許 Postgres 的商業化，Postgres 代碼沒有像 Ingres 那樣快速的被商業開發-- 考慮到 Postgres 提供的好處這有些令人驚奇。主要分支起源於 Paula Hawthorn（從 Ingres 轉移來的最初的 Ingres 專案組成員）和 Michael Stonebraker 組建 Illustra 資訊科技公司來商業化 Postgres 的時候。

在 2005 年一月，PostgresQL 收到了它來自資料庫廠商的第一份援助。Pervasive Software 宣布了商業支援和社群參與。

2000 年，前 Red Hat 投資者籌組了一間名為 Great Bridge 的公司來商業化 PostgreSQL，以和其他商用資料庫廠商競爭。Great Bridge 資助了好幾位 PostgreSQL 開發者，並且貢獻回饋了許多資源給社群。然而到了 2001 年末，Great Bridge 終止了營運，因為像是 Red Hat...等等公司面臨了嚴酷的競爭，且市場狀況不佳。

2001 年，Command Prompt, Inc.釋出了 Mammoth PostgreSQL，最老牌的 Postgresql 商業套件。他們透過對開發者的贊助和對如同 PL/Perl、PL/php、維護 PostgreSQL Build Farm ...等等方式來支援 PostgreSQL 社群。

2005 年 1 月，PostgreSQL 接到了來自另一間資料庫廠商 Pervasive Software 的支援，他們以十分常見於 Novell NetWare 平台的 Btrieve 產品而聞名。他們宣布了進行商業支援和對社群的參與。在他們得到成功一段時間後的 2006 年 7 月，Pervasive Software 離開了 PostgreSQL 的支援市場。

在 2005 年中，兩間其他的公司宣佈商業化 PostgreSQL，分別進入不同的利基市場。EnterpriseDB 宣布將專注於讓使用 Oracle 的應用程式能更容易的在 PostgreSQL 上執行。Greenplum 則專注貢獻在資料倉儲和商業智慧的應用程式，尤其以 BizGres 專案著稱。

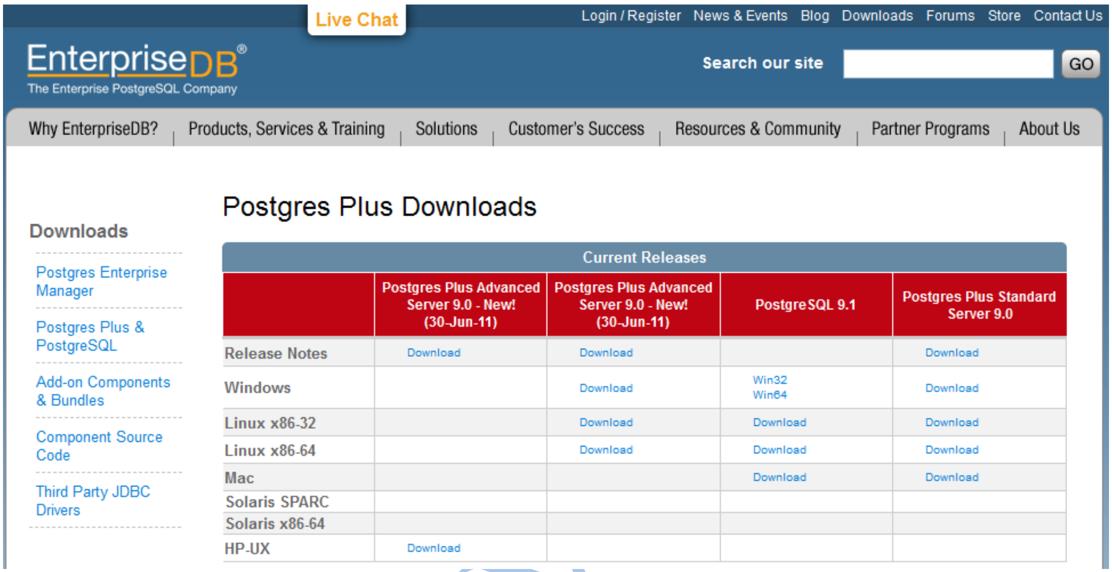
2005 年 10 月，昇陽的軟體部門執行副總裁 John Loiacono 談論到："我們不會去 OEM 微軟的產品，我們正關注著 PostgreSQL"，儘管當時並沒有任何規格釋出。到了 2005 年 11 月，昇陽宣布將支援 PostgreSQL。2006 年 6 月，Solaris 10 包含 PostgreSQL 一起發佈。

至於 PostgreSQL 專案本身，他繼續著每年一個主要版本發佈，以及次要的除錯版本發佈，全都可以在 BSD 授權底下取得。這些都是基於商業化廠商、支援公司、和開放原始碼駭客。

關於 PostgreSQL 本節部份截取維基百科欲獲得更多資訊可參閱網址，

http://zh.wikipedia.org/wiki/PostgreSQL

直到近年來由 EnterpriseDB 公司包裝 PostgreSQL 成為企業版本，針對原本社群版做了更多功能加強，如相容 Oracle 資料庫、MSSQL 資料庫、提出異質資料庫轉移方案如 Oracle，PostgreSQL 組織也將部份交由 EnterpriseDB 維護，如 PostgreSQL 社群版本 Binary 安裝套件，現行 PostgreSQL 版本可分為社群版本與商業版本，商業版本稱為 Postgres Plus 分為標準版(Standard)及進階版(Advanced)，更多資訊可參考以下網址： http://www.enterprisedb.com



安裝 PostgreSQL

首先選擇 PostgreSQL 安裝版本，可選擇 Binary 套件 Source code 安裝另也有個各作業系統所維護的安裝版本如 Redhat 與 Suse 所出的 RPM 版本；Debian 與 Ubuntu 所出的 deb 版本，有由各作業系統所維護的版本都有稍許差異且安裝方式也較為簡單，就不多作介紹，本書將介紹如何安裝 Binary 套件或 Source code 編譯。

下載 PostgreSQL 安裝程式，上一節已介紹過所有 Binary 套件已由 EnterpriseDB 維護，故請至此網址下載，此網站中有企業版、標準版及社群版，請選擇社群版本下載：

http://www.enterprisedb.com/downloads/postgres-postgresql-downloads

✧ Binary 套件安裝

選擇 Binary 套件在安裝時比較簡單且快速，在安裝時可時可選擇圖型化介面或是命令模式安裝。

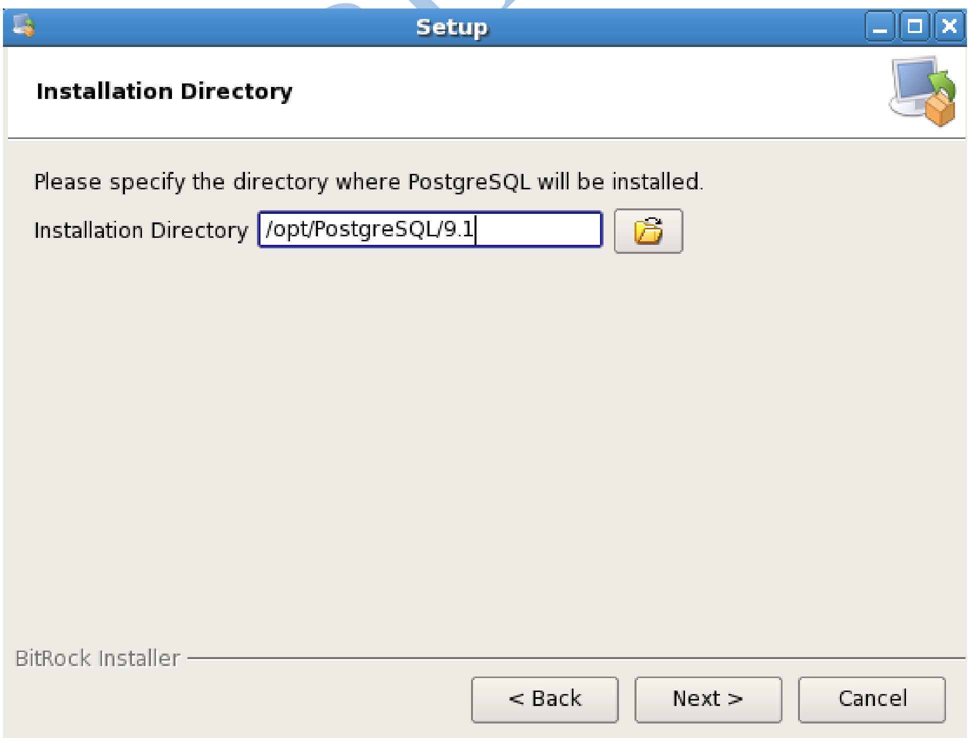
當執行安裝動作時使用者帳號須擁有作業系統 root 權限，將下載下來的安裝程式檔案增加可執行權限，並執行此安裝程式如下，安裝模式會自動偵測是否進入圖型模式安裝：

```
# chmod 755 postgresql-9.1.1-1-linux-x64.bin
# ./postgresql-9.1.1-1-linux-x64.bin
```

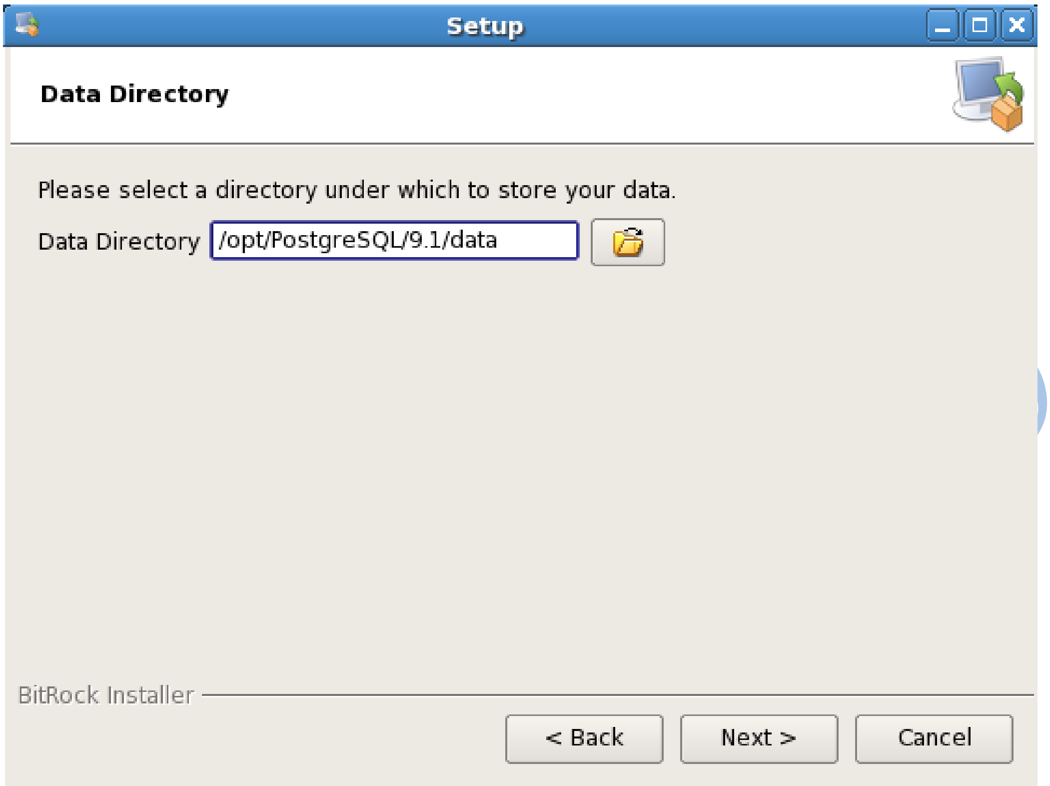
➢ 圖型模式安裝

PostgreSQL 安裝歡迎畫面：

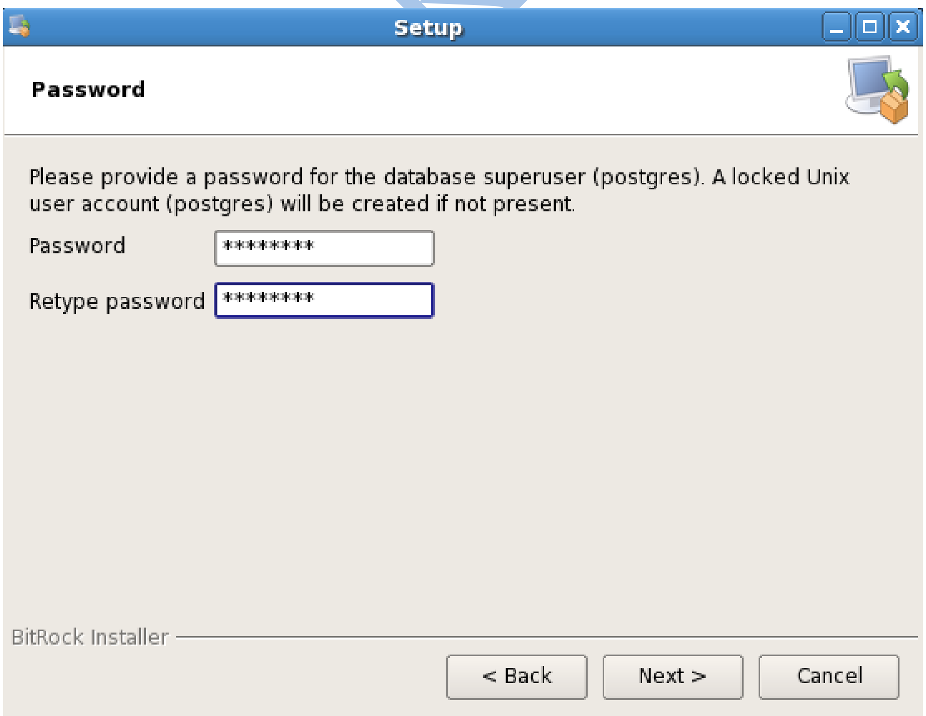
選擇 PostgreSQL 軟體安裝目錄：



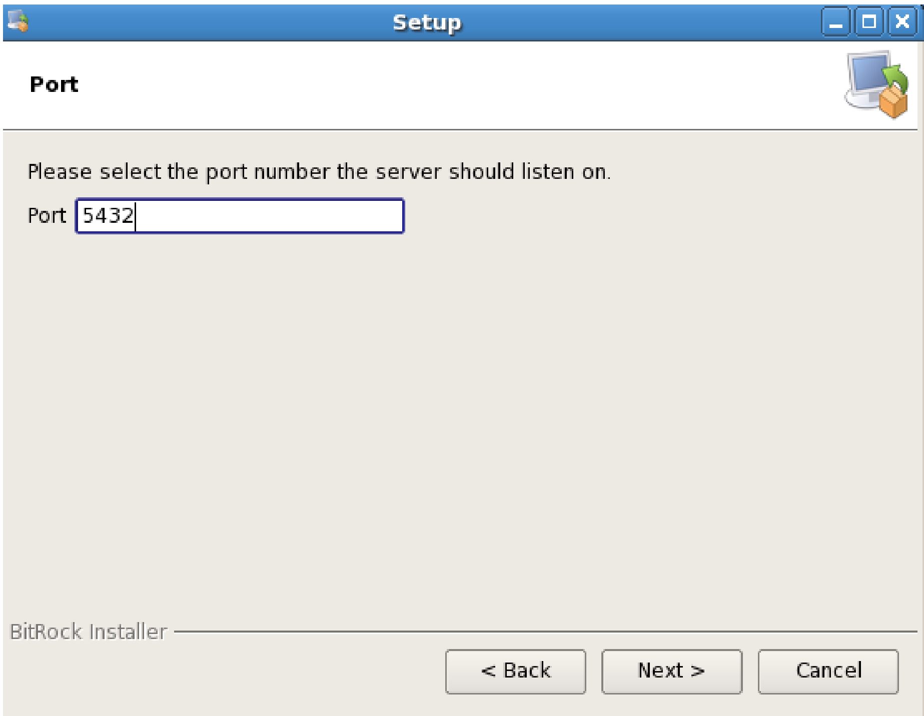
選擇伺料庫伺服器資料目錄：



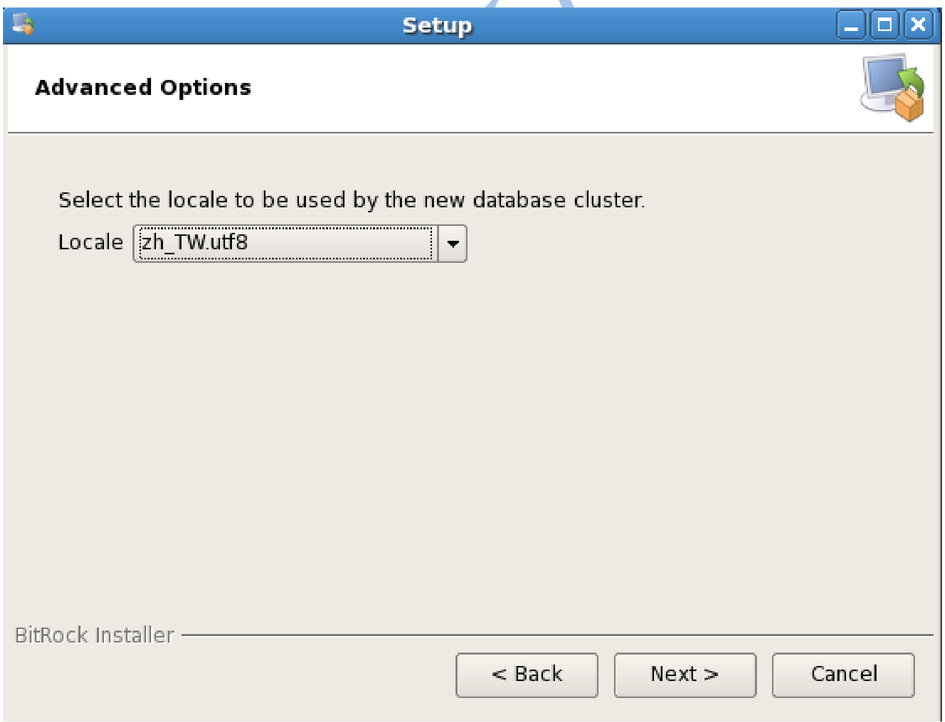
輸入資料庫伺服器 Superuser 使用者帳號 postgres 密碼，如作業系統帳號並未建立 postgres 帳號也會一併建立：



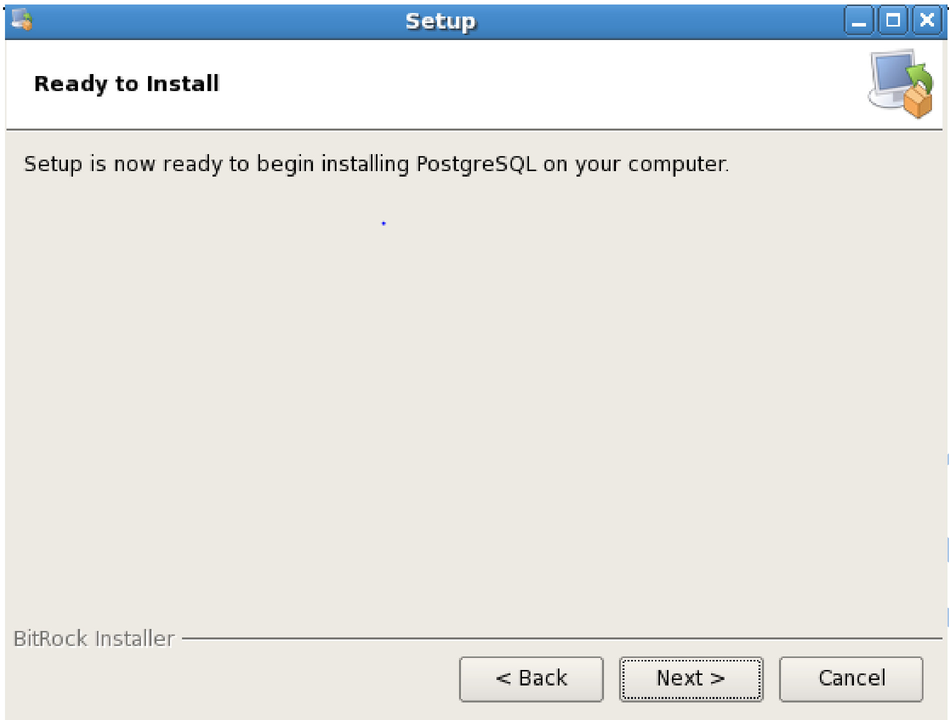
輸入資料庫伺服器監聽埠號碼：



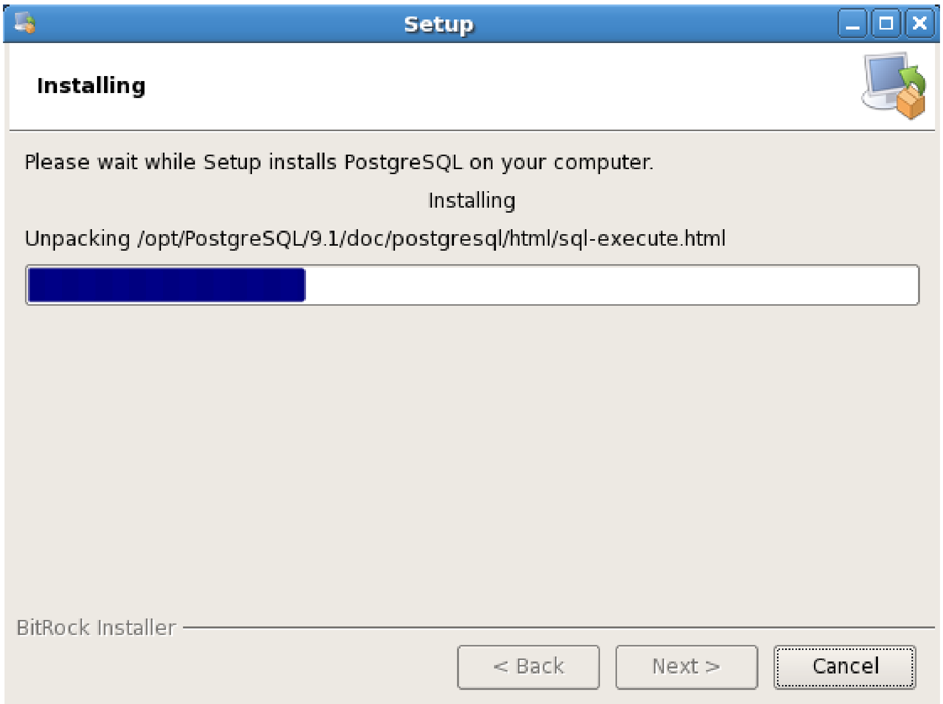
選擇資料庫預設語系及字元集：



當一切選擇完畢後，準備安裝：



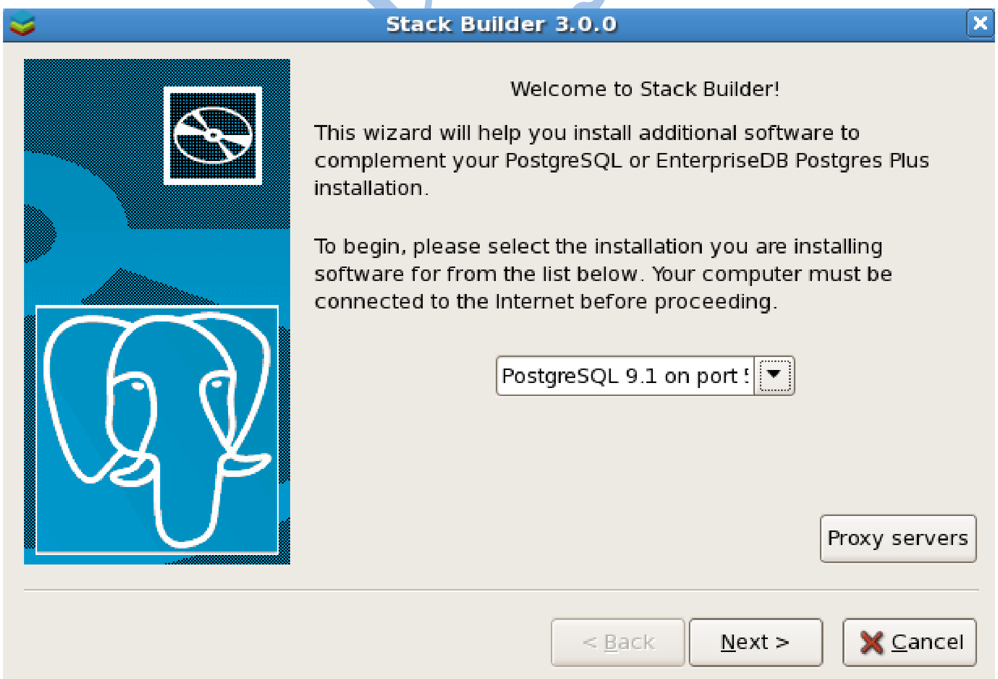
安裝 PostgreSQL 時畫面：



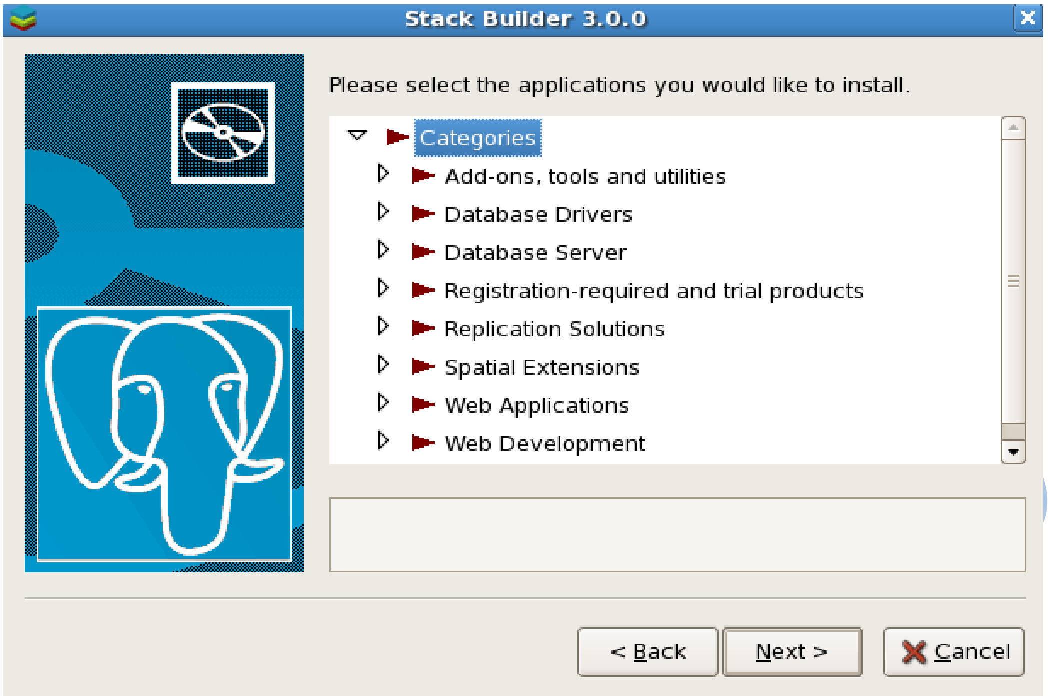
安裝 PostgreSQL 後，會提示是否開啟 Stack Builder 精靈，透過此精靈可安裝

額外其它工具，預設開啟 Stack Builder 精靈：

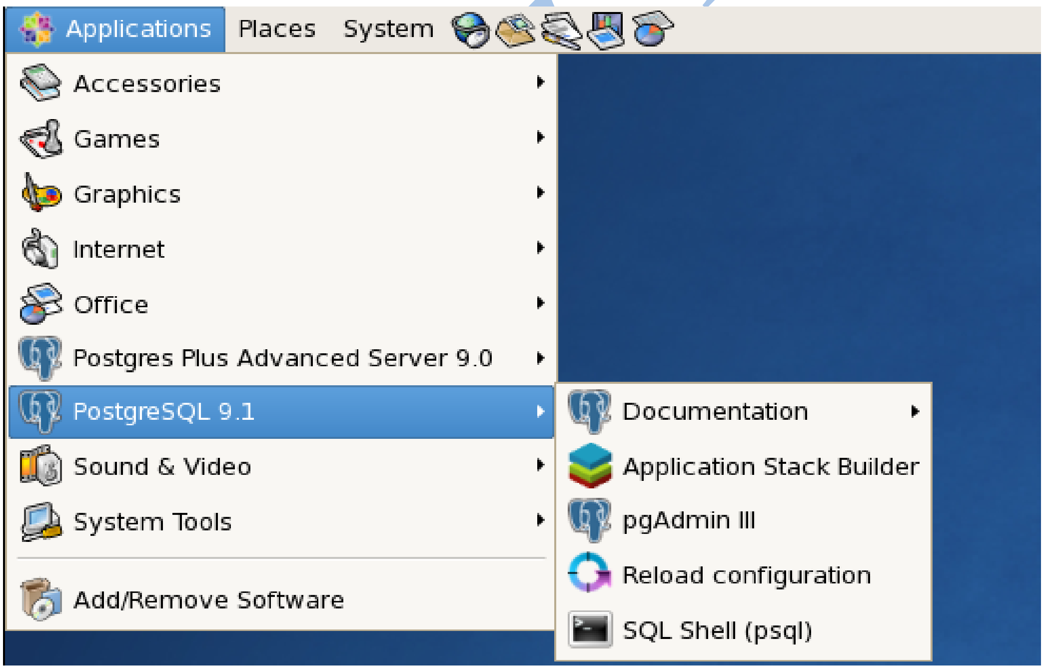
開啟 Stack Builder 精靈後，選擇作業系統中已安裝 PostgreSQL 版本來進行額外工具安裝：



以下為 Stack Builder 精靈所提供的額外工具列表，如 Database Drivers。



當一切安裝完成後，可透過作業系統工能表啟用 PostgreSQL 程式或開啟說明文件：



命令模式安裝：

選擇命令模式安裝更簡便及快速，基本上與圖型化模式安裝相同，故就不多介紹囉！

啟動命令模式安裝並選擇安裝目錄及資料庫伺服器目錄等資訊。

```
[root@db1 ~]# ./postgresql-9.1.1-1-linux-x64.bin
----------------------------------------------------------------------------
Welcome to the PostgreSQL Setup Wizard.

----------------------------------------------------------------------------
Please specify the directory where PostgreSQL will be installed.

Installation Directory [/opt/PostgreSQL/9.1]:

----------------------------------------------------------------------------
Please select a directory under which to store your data.

Data Directory [/opt/PostgreSQL/9.1/data]:

----------------------------------------------------------------------------
Please provide a password for the database superuser (postgres). A locked Unix
user account (postgres) will be created if not present.

Password :
Retype password :
----------------------------------------------------------------------------
Please select the port number the server should listen on.

Port [5432]:
```

選擇資料庫預設語系及字元集：

```
[622] zh_CN
[623] zh_CN.gb2312
[624] zh_CN.utf8
[625] zh_HK.utf8
[626] zh_SG
[627] zh_SG.gb2312
[628] zh_SG.utf8
[629] zh_TW.euctw
[630] zh_TW.utf8
[631] zu_ZA
[632] zu_ZA.iso88591
[633] zu_ZA.utf8
Please choose an option [1] : 630
```

安裝 PostgreSQL 時畫面

```
----------------------------------------------------------------------------
Setup is now ready to begin installing PostgreSQL on your computer.

Do you want to continue? [Y/n]: y

----------------------------------------------------------------------------
Please wait while Setup installs PostgreSQL on your computer.

 Installing
 0% ______________ 50% ______________ 100%
 ########################
```

✧ Source code 安裝

使用源始碼(Source code)進行安裝時比較複雜且困難，可隨時依自己需求進行調整，如安裝好 PostgreSQL 資料庫伺服後，資料庫伺服器交易檔(WAL, Write Ahead Log)預設為 16MB 是不能更改的，卻可在透過編譯原始碼時進行彈性修改，然而如在編譯原始碼時如一小步驟設定錯誤可能導致日後上維護時困難或發生不可預期的錯誤，使用源始碼安裝屬於較進階且熟悉 PostgreSQL 人士所使用，故本書就不多做說明。

連接 PostgreSQL9 伺服器

當安裝完 PostgreSQL 不是迫不急待想進入連接至資料庫伺服器體驗一下呢？

在存取資料庫伺服器前，須擁有以下資訊方可連接：

- ✧ 主機名稱或位址。
- ✧ 連接埠(預設埠號為 5432)。
- ✧ 資料庫名稱。
- ✧ 使用者。
- ✧ 密碼 。

當擁有以上資訊後接著可使用用戶端工具存取資料庫伺服器，如 PostgreSQL 內建工具 psql 進入命令列模式或 JDBC 方式連接，但本書主要為 DBA 管理，故以 psql 工具為主，將在「使用 psql 命令模式及如何載入腳本」章節詳細介紹。

```
$ psql -Ufoo -W -h localhost -p 5432 store
Password for user foo:
psql (9.1rc1)
Type "help" for help.

store=>
```

可執行 psql --help 列出所有參數說明及用法。

連接資料庫時不想指定參數，可改透過設定作業系統環境達成：

- ✧ PGHOST, PGHOSTADDR。
- ✧ PGPORT (預設為 5432)
- ✧ PGDATABASE
- ✧ PGUSER
- ✧ PGPASSWORD (不安全故不建議使用)。

當有些特殊情況希望不用輸入密碼即可登入，可參考章節「資料庫伺服器密碼」，將會有詳細介紹。

查詢基本資訊：

身為 DBA 可能需管理非常多的資料庫伺服器，甚至一次連接二台以上資料庫伺服器，在眾多管理視窗中可能造成混淆，而發生執行錯指令而造成不可預期錯誤，故身為 DBA 務必經常執行以下 SQL 語法來獲得這些有用資訊：

查詢現在所連接資料庫：

```
postgres=# SELECT current_database();
 current_database
------------------
 postgres
```

查詢現在已連接使用者：

```
postgres=# SELECT current_user;
 current_user
--------------
 postgres
```

查詢資料庫伺服器 IP 位址：

```
postgres=# SELECT inet_server_addr();
   inet_server_addr
--------------------------
        5432
```

查詢資料庫伺服器埠號：

```
postgres=# SELECT inet_server_port();
   inet_server_port
--------------------------
        5432
```

查詢資料庫伺服器版本：

```
postgres=# select version();
     version
-----------------------------------------------------------------------
 PostgreSQL 9.1rc1 on x86_64-unknown-linux-gnu, compiled by gcc (GCC) 4.1.2
20080704 (Red Hat 4.1.2-50), 64-bit
```

資料庫伺服器認証檔

安裝好 PostgreSQL 資料庫伺服器後預設並未允許遠端連線存取，故首先須啟用遠端連線存取，並設定資料庫伺服器認証檔來設定用戶端如何存取及限制。

- 啟用遠端連線存取：
  編輯資料庫伺服器設定檔參數：

```
        listen_addresses = '*'
```

- 編輯資料庫伺服器設定檔 pg_hba.conf：

```
➢   # TYPE      DATABASE        USER     CIDR-ADDRESS     MTEHOD
host    all             all     0.0.0.0/0       md5
```

  - TYPE：使用何種類型
    local：使用 Unix Socket 網路類型
    host：使用 IPv4、IPv6網路類型
    hostssl：使用 SSL 加密網路類型

  - DATABASE：允許連線的資料庫
    設定資料庫名稱。若設定為 all 代表為所有資料庫

  - USER
    使用者名稱或是群組名稱，all 代表所有使用者

➢ METHOD

認證方式，說明如表格所示：

認証方式	說明
trust	不進行驗證，直接信任
reject	符合設定條件時拒絕連線
md5	使用 md5 進行密碼驗證
crypt	以系統資料表 pg_shadow 進行驗證
password	使用明碼進行驗證
kbr5	以 kerberos V5 進行驗證
pam	插入 PAM 模組進行驗證
ldap	使用 LDAP 進行驗證

更多…

資料庫伺服器認証檔規則會由上而下依序套用。如下，表示資料庫伺服器拒絕192.168.0.x 網段連線，其餘網段皆允許。

```
host    all     all     192.168.0.1/24  reject
host    all     all     0.0.0.0/0       trust
```

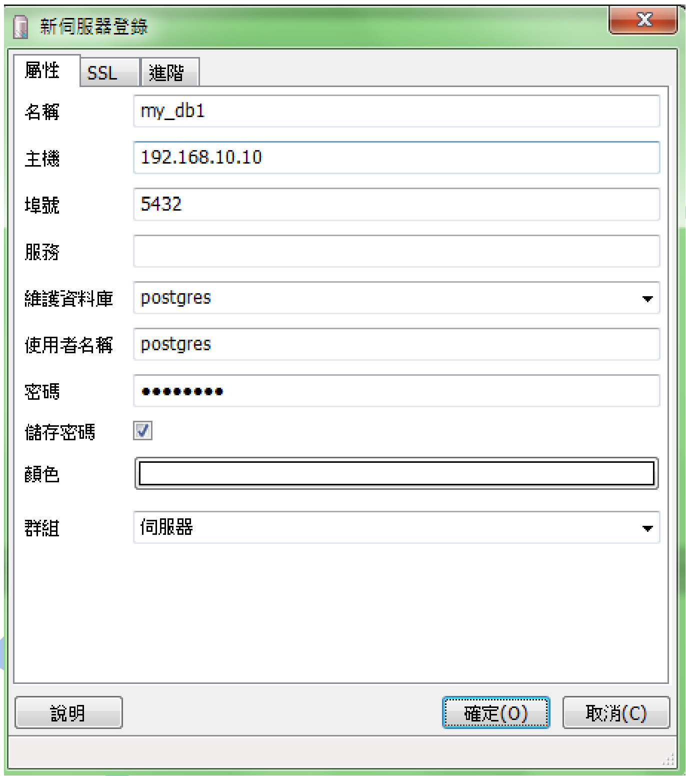
使用圖型化管理工具

圖型化管理工具會協助 DBA 減輕許多管理上的工作，本節介紹 PostgreSQL 最有名的二套免費的管理工具：

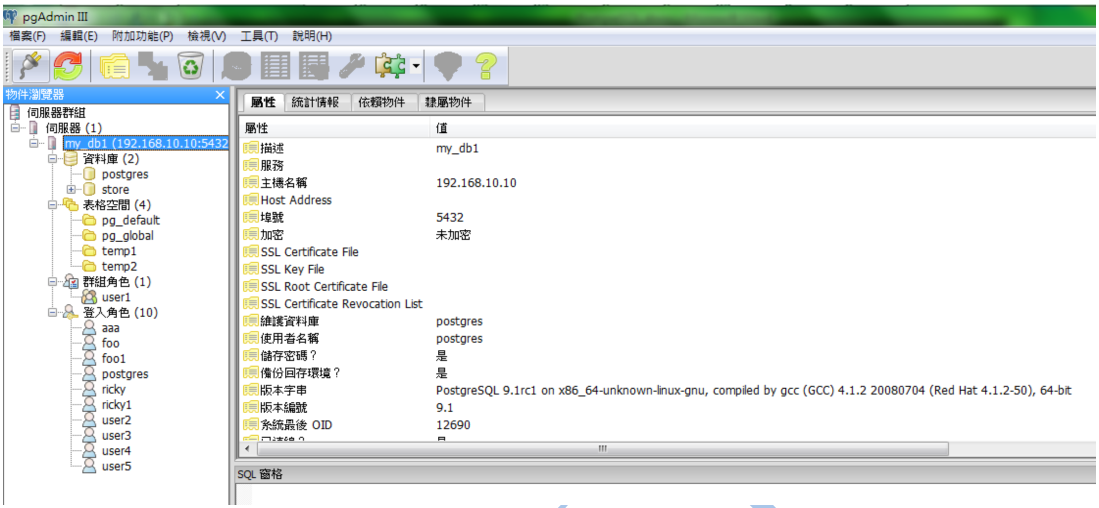
- pgAdmin III(Postgres Studio)
- phpPgadmin

pgAdmin III：

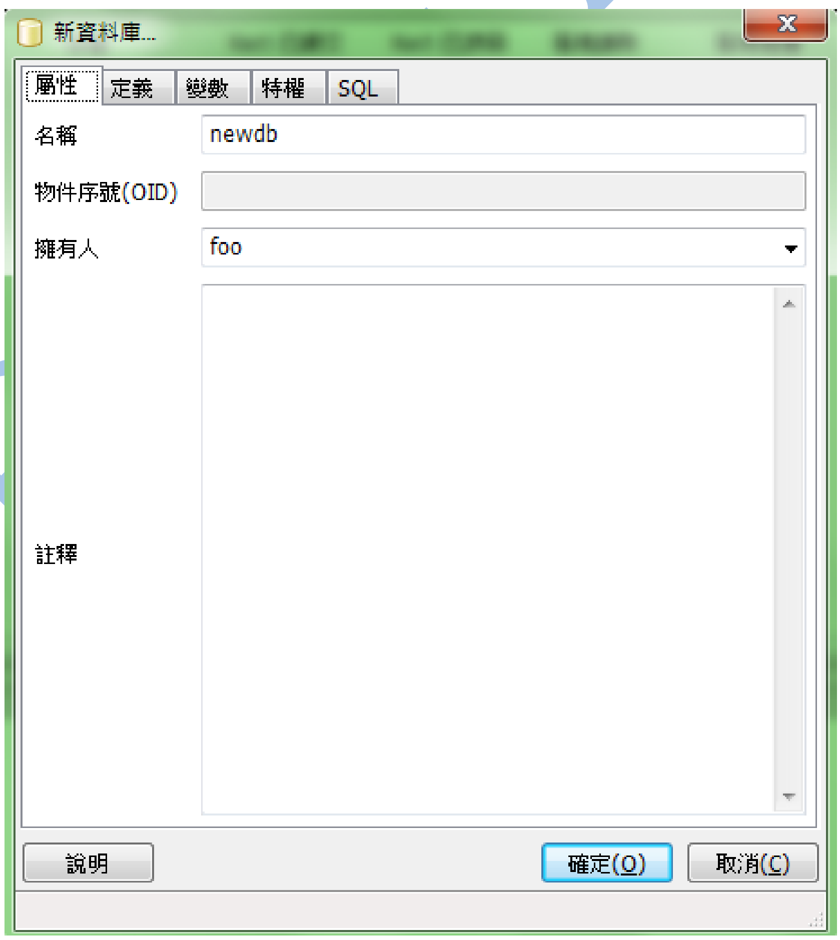
此管理工具在安裝完 PostgreSQL 後，就內建於資料庫管理系統中，首次開啟 pgAdmin III，須註冊新伺服器，才能進行管理。

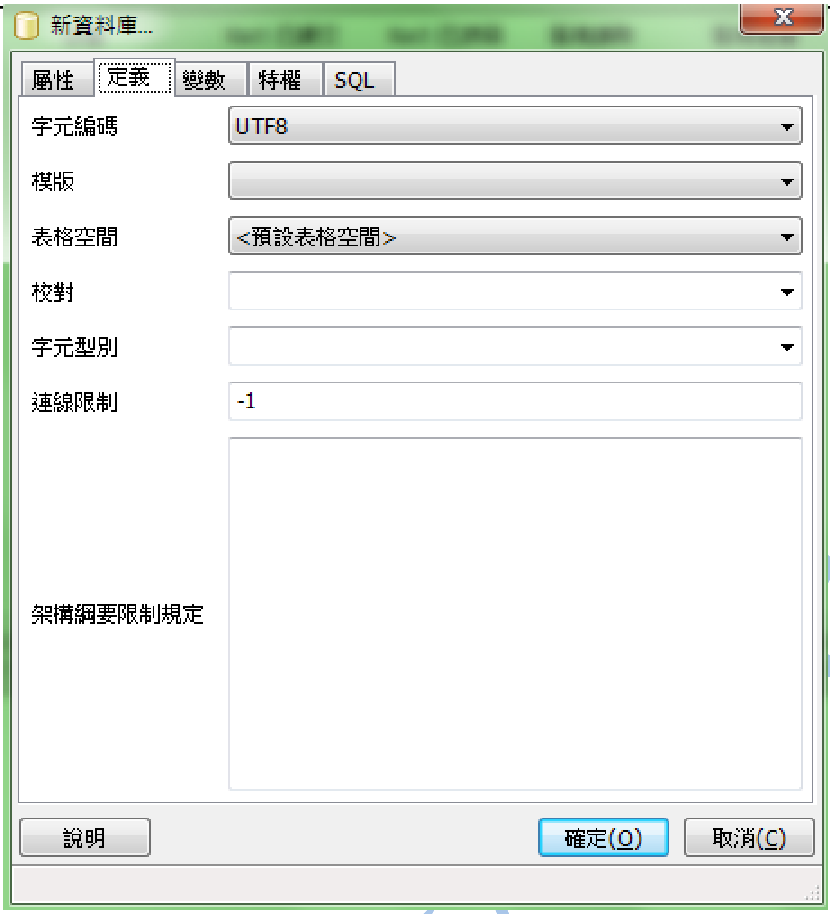


註冊伺服器完畢後，進入管理介面，便可針對可資料庫伺服器各元件進行管理如：資料庫、表格空間、角色…等等。

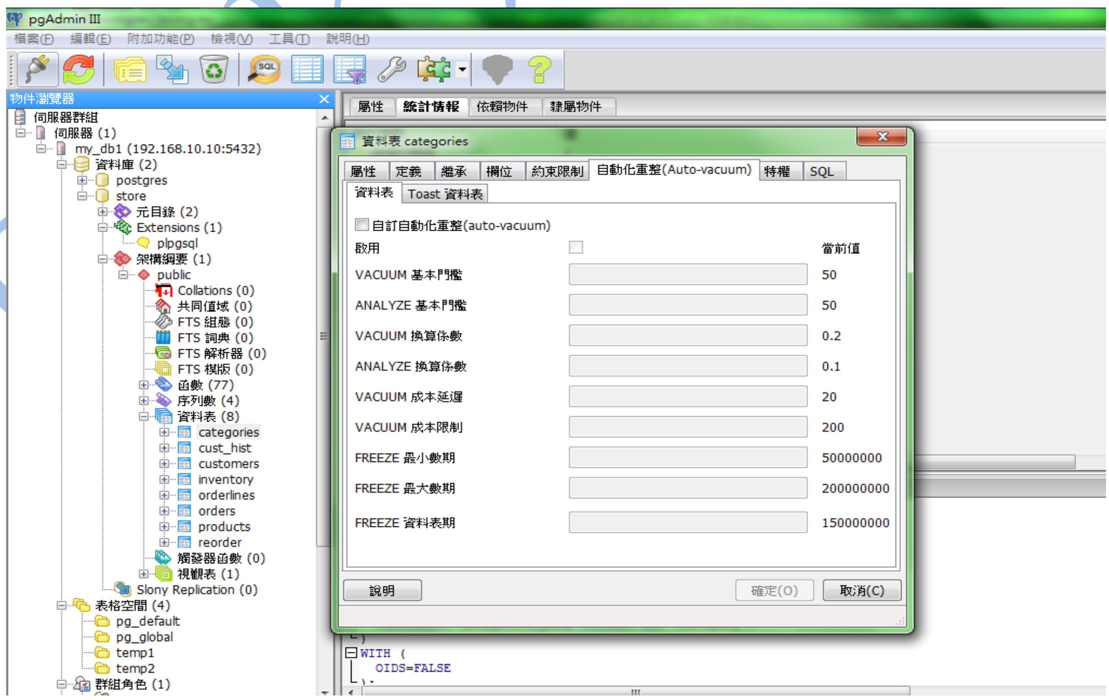


- 依照圖示建立新資料庫，須特別注意如欲使用中文字集但不使用 UTF-8，請選擇 ECU_TW，因 PostgreSQL 已停用 Big5 選項：

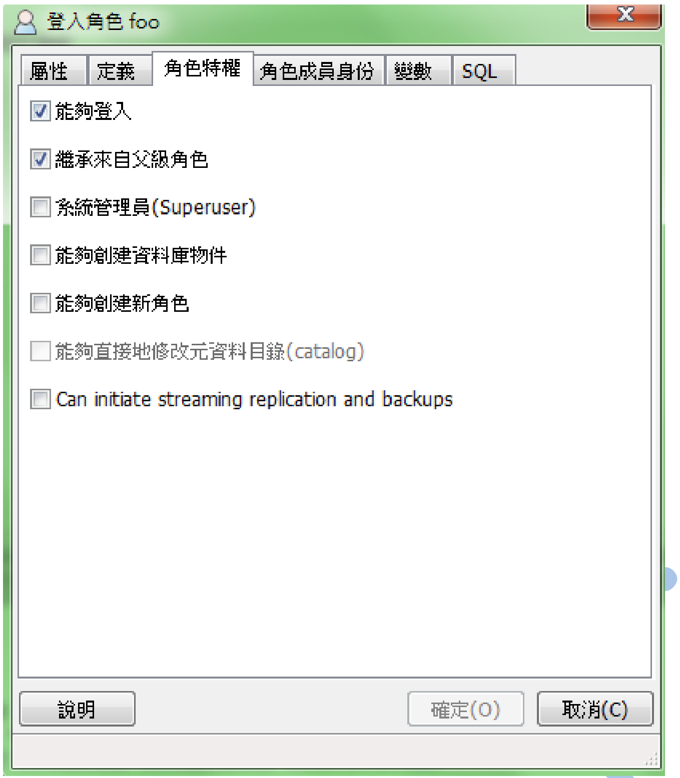


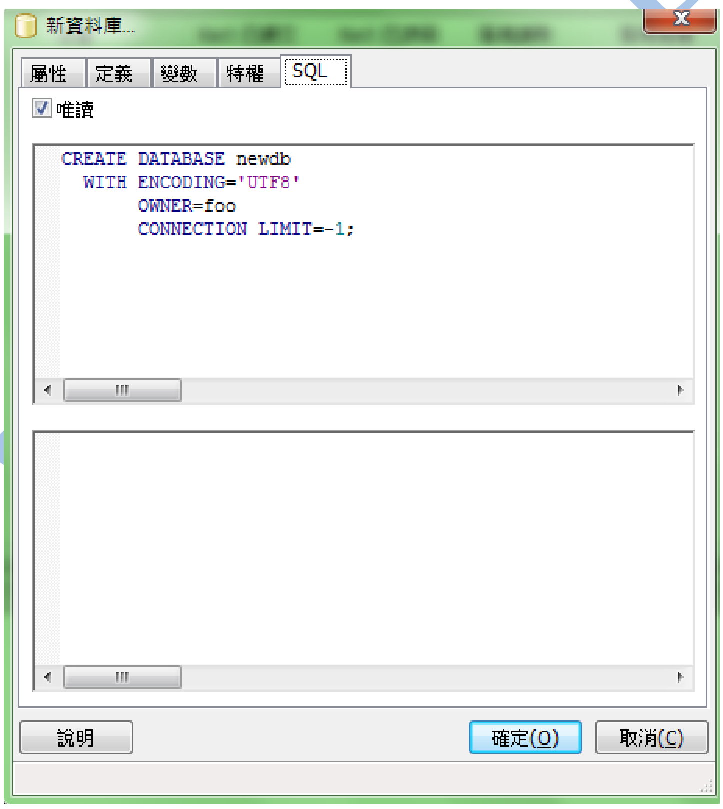


可透過工具重整資料表，將於後面章節詳細介紹重資資料庫：

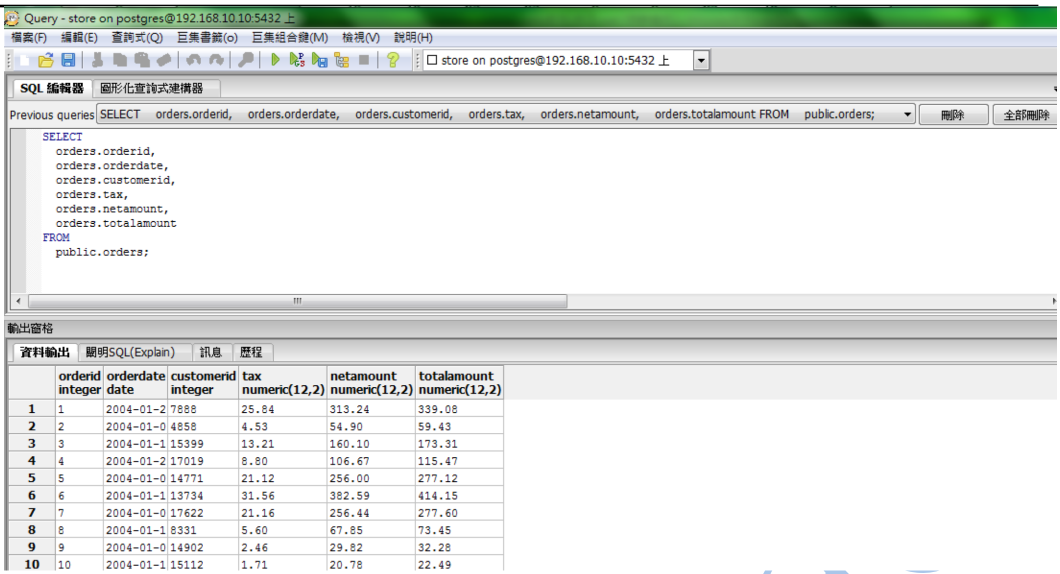


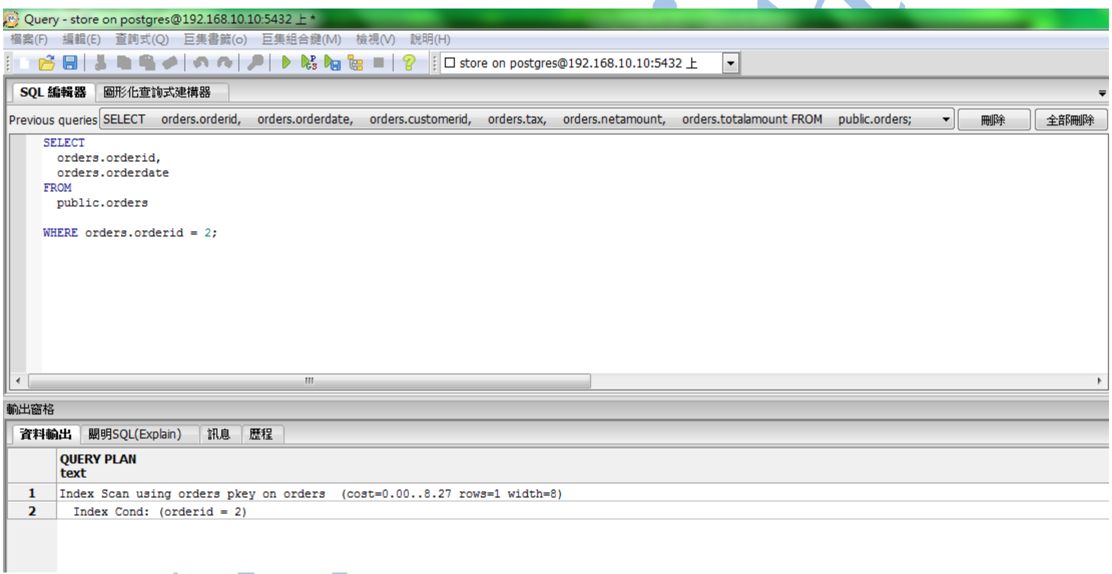
如圖修改使用者角色權限，並產生相對 SQL 語法：





圖型化查詢介面，並可提供執行計劃分析：





本書並非著重在如何操作 PGAdmin III 圖型化管理工具，基本上透過圖型工具進行管理也相當簡單直覺，就不多做介紹囉！

phpPgadmin：

首先，連接至以下網址下載 phpPgadmin 工具：

參考以下網址：http://phppgadmin.sourceforge.net/doku.php?id=start

- 安裝 phpPgAdmin

  phpPgAdmin 為 Web 圖型管理介面，故需安裝在網頁伺服器-Apache 且支援 php。

  下載 phpPgadmin 工具後， 解壓縮 phpPgAdmin 至網頁伺服器根目錄下 /var/www/html/。.

- phpPgAdmin 的設定

修改 phpPgAdmin 設定檔，此檔位於 phpPgAdmin 下子目錄 conf 中，名為 config.inc.php。

依提示修改設定變數如：

```
$conf['servers'][0]['desc'] = 'PostgreSQL';
$conf['servers'][0]['host'] = 'localhost';
$conf['servers'][0]['port'] = 5432;
```

如欲加入第二台資料庫伺服器管理，只需將陣列由 0 改為 1 即可，如欲再增加依此類推：

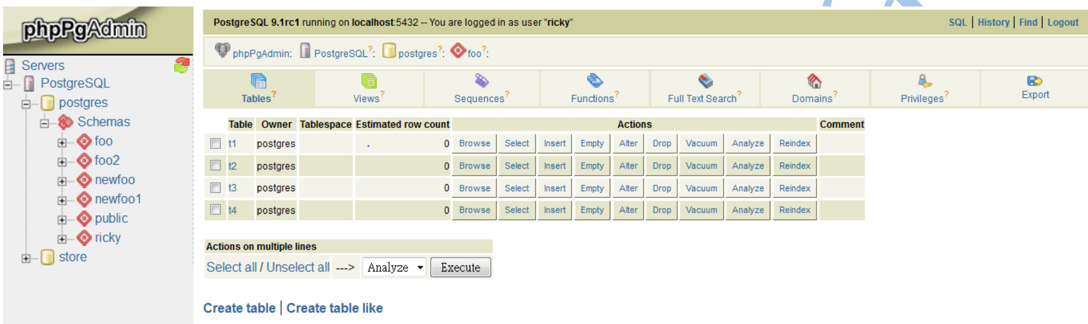
```
$conf['servers'][1]['desc'] = 'PostgreSQL';
$conf['servers'][1]['host'] = 'localhost';
$conf['servers'][1]['port'] = 5432;
```

設定檔修改完畢，如無誤，透過瀏覽器連接此網址：

http://IP 位址/phpPgadmin/

再選擇預管理的資料庫伺服器輸入帳號密碼即可進入：

phpPGAdmin 圖型化管理工具介面如下：



本書並非著重在如何操作 phpPgAdmin 圖型化管理工具，基本上透過圖型工具進行管理也相當簡單直覺，就不多做介紹囉！

更多…

除了上述介紹的開源圖型化管理工具外，市面上仍有許多商業版的開發管理工具，已分類列在 PostgreSQL 網站上可參閱以下網址：

http://www.postgresql.org/download/products/1

以下這幾個產品為較普遍上常聽到的管理且兼具開發的圖型化管理工具，有興趣的話可以參閱以下網址：

- ✧ Navicat
  http://www.navicat.com/
- ✧ EMS SQL Manager
  http://www.sqlmanager.net/en/products/manager
- ✧ Lightning Admin
  http://lightningadmin.com/

- PostgreSQL Maestro
  http://www.sqlmaestro.com/products/postgresql/maestro/

使用 psql 命令模式及如何載入腳本

psql 命令模式為 PostgreSQL 主要核心功能，所以身為 DBA 應該要非常熟悉此工具才是，透過 psql 命令模式可執行任何 SQL 語法並載入腳本。

psql 預設使用交談互動模式，但也可以搭配 Shell Script 批次執行。如 createdb 與 createuser 就擁有能夠呼叫 psql 的 Shell Script。

psql 指令相當簡易，應用範圍也很廣，熟悉 pqsl 就可以用來處理許多工作。

psql 基本使用方法如下：

$ psql 資料庫名稱 使用者名稱

執行後，提示輸入密碼，完成驗證後進入 psql 交談互動模式。

若省略資料庫名稱、使用者名稱則會讀取系統環境變數$PGDATABASE、$PGUSER

當 psql 命令列模式後，顯示如下圖便會進入 psql 提示符號，在此提示符號下，可輸入任何 SQL 敘述或者是輸入稱為「反斜線指令」(backslash commands)的 meta-command 指令。

```
[postgres@postgres postgres]$ psql
psql (9.1rc1)
Type "help" for help.

postgres=#
```

psql 說明功能：

我們可利用 psql 說明功能，了解 SQL 指令的簡單使用方法。譬如可利用說明功能，來取得下最 SQL 指令用法：

\? 列出所有 meta-command 指令及說明

\h 列出所有 SQL 指令

\h 指令名稱 --列出某個 SQL 指令說明

以下為列出 CREATE 語法說明：

```
\h CREATE
Command:     CREATE AGGREGATE
Description: define a new aggregate function
Syntax:
CREATE AGGREGATE name ( input_data_type [ , ... ] ) (
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , FINALFUNC = ffunc ]
    [ , INITCOND = initial_condition ]
    [ , SORTOP = sort_operator ]
)…(略)
```

如何註解：

為了在讓他人可更清楚 SQL 所要指行的內容，註解就成為一個非重要的事，註解主要可分為以下兩種方式：

```
-- 這是單行註解
```

```
/*
 * 這是多行註解
 */
```

更多…

在有些情況中可能不希望進入 psql 命令模式，希望以單行方式就能執行 SQL 語法，可在 psql 指令後指定參數「-c 」來達成：

```
$ psql -c "SELECT * FROM current_user;"
 current_user
--------------
 postgres
```

至於當欲執行多行以上 SQL 語法時呢？我們可將全部 SQL 語法寫入至單一檔案並命名再透過 psql 指令後指定參數「-f 」即可達成：

```
 $ psql -f scripte.sql
```

當欲在進入 psql 命令模式前，載入某些設定值或執行特定 SQL 語法，可建立檔案/etc/psqlrc 或於使用者家目錄~/.psqlrc 來設定。

變更使用者密碼

我想當安裝好資料庫伺服器並建立使用者帳號後，最常的管理動作應該是如何變更使用者密碼吧！本節就教導大家如何來變更使用者帳號密碼。

如何做…

變更使用者密碼可透過許多種工具來達成，當然最基本最直接的方式就是使用 psql 工具登入資料庫伺服器，再來就是變更使用者密碼，說到這變更使用者密碼也是有很多方式喔！如使用明碼或加密過的密碼儲存，但就不此節說明了，由於剛開始接觸而已就使用較簡單的方式：

進入 psql 命令模式後，執行 meta-command 如下，接著 PostgreSQL 會提示我們輸入欲變更的密碼：

```
\password
Enter new password:
Enter it again:
```

執行完動作後，psql 會將我們所輸入的內容轉化成 SQL 語法如下：

```
ALER USER foo PASSWORD 'md5fdba98970961edb29f88241b9d99d890';
```

當然也可透過 ALTER USER 執令來變更使用者密碼，將會於之後章節中介紹。

剛為大家介紹的是透過 psql 命令模式來變更密碼，其實也很多圖型工具可幫我們操作如：pgAdmin3 或 phpPgadmin，此兩種都是為開源軟體，另外也有商業軟體如：Navicat、EMS SQL Manager…等等。

PostgreSQL 本身並不會強制使用者定期變更密碼，如為加強安全性可搭配使用其它認證工具如 GSSAPI、SSPI、LDAP、RADIUS…等等。

資料庫伺服器密碼檔

一般而言使用者欲連接資料庫伺服器時，是得密碼驗証的是否很麻煩呢？其實有兩種方式可讓使用者不用輸入密碼：

- ✧ 資料庫伺服器密碼檔(.pgpass)
  將於介紹如何使用。
- ✧ 資料庫伺服器認証檔 pg_hba.conf
  此種方式可能有安全性疑慮故不建議。

準備開始

在有些情況中並非所有使用者連接資料庫伺服器皆需要密碼驗証，如共用使用者帳號，此時就可使用資料庫伺服器密碼檔-(.pgpass)達到此目的

建議不要使用環境變數 PGPASSWORD，此方式並不安全。

資料庫伺服器密碼檔包含五個欄位如下：

語法：

主機名稱：埠號：資料庫：使用者帳號：密碼

```
-- syntax host:port:dbname:user:password
db1:5432:store:postgres:P@ssw0rd
```

在語法中可使用星號(*)代表萬用字元，如在埠號或資料庫使用星號(*)表示，則表示可接受任何埠號及任何資料庫，範例如下：

```
db1:*:*:postgres:P@ssw0rd
```

如無名稱解悉者，請指定 IP

- ✧ Linux 或 unix 平台位於作業系統使用者家目錄中，~/.pgpass
- ✧ Windows 平台位於 PostgreSQL 軟體目錄中檔名為 pgpass.conf

資料庫伺服器密碼檔所在位置可定義於環境變數 PGPASSFILE 中，如未定義則以上面介紹位置存放。

在建立密碼檔後請不要忘記執行指令修改權限，否則 PostgreSQL 可是會忽略這個檔案喔而造成無法生效，至於 Windows 平台則不受此限制。

```
$ chmod 0600 ~/.pgpass
```

資料庫伺服器連接服務設定檔

身為 DBA 當管理多台資料庫伺服器可考慮使用資料庫伺服器連接服務設定檔來方便我們管理，透過資料庫伺服器連接服務設定檔 PostgreSQL 用戶端程式 libpq 連接時所需要的參數設定值，此檔可位於 PostgreSQL 設定檔目錄中或

$PGDATA，也可位於使用者家目錄~/.pg_service.conf，如欲自己定義路徑位置透過環境變數 PGSYSCONFDIR 定義資料庫伺服器連接服務設定檔位置，再由 PGSERVICE 指定預使用哪個 SERVICE。

建立一檔案名為 pg_service.conf 並編輯如內容：

```
pg_service.conf
[dbservice]
host=db1
port=5432
dbname=store
```

定義 pg_service.conf 檔所在目錄，執行 psql 時輸入參數 service 來指定預連的服務名稱：

```
$ export PGSYSCONFDIR=/postgres
$ psql service=dbservice
```

當然我們也可以預先指定欲連的服務名稱，就不用再執行 psql 時輸入參數：

```
export PGSERVICE=dbservice
$ psql
```

如何運作…

資料庫伺服器連接服務設定檔只對 PostgreSQL 用戶端程式 libpq 生效，是無法使用在 JDBC 上的喔！

資料庫伺服器連接服務設定檔可用於共用一特定使用者帳號，讓不同操作人員來使用此特定使用者帳號，透過設定檔後就無需指定使用者帳號及連接方式了，也方便了管理，另外，當使用 pg_service.conf 檔時可搭配.pgpass 檔，如此登入時就不需再輸入密碼，但得考慮安全性問題喔！

資料庫伺服器連線故障排除

此節將教導大家當遇到資料無法連接資料庫伺服器時，如何判斷並解決問題。

✧ 檢查使用者帳號密碼是否正確或資料庫名稱是否存在：

當無法連接資料庫伺服器時第一時間想到的就是使用者帳號密碼是否正確輸入無誤，因往往可能因為密碼記錯或是輸入錯誤而導致無法登入，當然也有另一個可能性就是資料庫名稱不存在或是輸入錯誤皆有可能，而如在登入時出現訊息「too many connections」則可能資料庫伺服器連線數量超過限制，只能增加連線數並找出連線數變高原因。

✧ 錯誤訊息如下：

```
pg_hba.conf rejects connection for host "192.168.10.253", user "postgres", database "postgres", SSL off
```

如出現以上訊息表示資料庫伺服器認証檔中有一規則將拒絕這台主機連接至資料庫伺服器，只能請 DBA 設定允許連線囉！

✧ 錯誤訊息如下：

```
no pg_hba.conf entry for host "192.168.10.253", user "postgres", database "postgres", SSL off
```

如出現以上訊息表示資料庫伺服器認証檔中缺少允許這台主機連接規則，只需增加此規則至檔 pg_hba.conf 並重新載入即可生效。

✧ 是否能進入 psql 命令模式

連接資料庫伺服器可能往往會使用其它方式如 JDBC 或其它開發管理工具，如發生異常時，最直接快速的方式就是使用 PostgreSQL 內建 psql 工具進入命令模式，代表本機存取是正常的，可能是遠端無法存取或其它原因導致。

✧ 檢查資料庫伺服器是否啟動

當資料庫伺服器關閉時當然無法接受任何連線，故當無法透過遠端及本機存取資料庫伺服器機，可執行以下指令判斷該程序是否存在：

```
$ ps -elf|grep postgres
```

✧ 資料庫伺服器是否允許新連線

當資料庫伺服器正常啟動，有可能資料庫伺服器正處於單一時用者模式(Single User Mode)或使用者連線數被限制。

✧ 檢查資料庫伺服器監聽埠是否正常

有可能資料庫伺服器監聽埠被變更等原因，可使用作業系統內建工具 telnet 判斷：

```
telnet IP 位址 埠號
```

✧ 檢查資料庫伺服器監聽埠是否允許我們這連接的網段

有些情況中資料庫伺服器監聽埠是正常的，但須注意 PostgreSQL 預設只允許本機存取，須修改資料庫伺服器設定檔 postgresql.conf 將參數 listen_addresses 註解移除，基於安全性理由可選擇只開放單一網段。

```
listen_addresses = '*'
listen_addresses = '192.168.10.10';
```

✧ 查看資料庫伺服器日誌檔

當對於錯誤訊息不是很明確時，可開啟稽核登出、登入使用者訊息寫至資料庫伺服器日誌檔，可透過修改資料庫伺服器設定檔開啟此功能：

```
log_connections = on
log_disconnections = on
```

CHAPTER 2

開始使用資料庫

本章內容大綱如下：

- 查詢資料庫版本
- 資料庫運行時間
- 資料庫檔案
- 資料庫伺服器日誌檔
- 列出資料庫
- 列出資料表
- 計算資料庫佔用空間
- 計算資料表佔用空間
- 找出最大資料表
- 資料表筆數
- 快速計算資料表筆數
- 了解物件相依性

前言

坐而言不如起而行，現在就開始著手操作資料庫吧！安裝完 PostgreSQL9 又不開始進行管理，怎麼會了解精隨呢？

現在開始我們假設您已經安裝好 PostgreSQL9 並建立資料庫，世面上已有許多關於如何設計資料庫的書籍，故在本書中將著墨在如何管理 PostgreSQL9 資料庫管理系統。

查詢資料庫版本

當資料庫遇到問題時，首先必需了解現行的資料庫版本。如此才會幫助我們進行問題排除並回報問題。

怎麼做

進入資料庫命令模式並執行以下命令：

```
postgres=# SELECT version();
```

我們將會得到以下回應如下：

PostgreSQL 9.0.4 on x86_64-unknown-linux-gnu, compiled by GCC gcc (GCC) 4.1.2 20080704 (Red Hat 4.1.2-46), 64-bit

資料庫版本講解

PostgreSQL9 版本編號，由「**Major.Minor.Maintenance**」組成。

Major 就是主要版本號碼，Minor 為次要版本編號，Maintenance 為維護版本編號。其實按照 PostgreSQL9 的慣例，只有 Major 是無意義的，PostgreSQL9 的版本都是由 Major.Minor 來區分版本版次的。每個版次功能上都會有所不同，資料檔的格式也都多少有變化。

關於每個版本支援的生命週期可參考以下網址：
http://wiki.PostgreSQL.org/wiki/PostgreSQL_Release_Support_Policy

每個資料庫版本基本上都會維持期間五年的支持維護。

所有 PostgreSQL9 舊版本包含 8.1 將名 2010 年 11 月停止支持。也就是說在此段時間之後只有 8.2 以上才得到支持。話雖如此，舊有版本仍有一定的穩定性，只是缺少了更多新特功能及效能強化。

以下列出 PostgreSQL9 各版本生命週期：

PostgreSQL9 7.4	2010/10	2003/11
PostgreSQL9 8.0	2010/10	2005/01
PostgreSQL9 8.1	2010/11	2005/11
PostgreSQL9 8.2	2011/12	2006/12
PostgreSQL9 8.3	2013/02	2008/02
PostgreSQL9 8.4	2014/07	2009/07
PostgreSQL9 9.0	2015/09	2010/09

更多

我們還可以透過以下方式在未進入資料庫前查詢資料庫版本：

```
bash# psql --versioin
psql (PostgreSQL9) 9.0.4
```

然而有時候我們可能查詢到的是資料庫客戶端軟體的版本，它有可能與資料庫伺服器版本是不同的哦。所以最好再透過以下方式確認：

```
bash#  cat $PGDATA/PG_VERSION
9.0
```

不過這邊是不會列出資料庫的維護版本的！

為什麼資料庫版本如此重要？
在每個不同的資料庫版本，資料庫都會有不同的資料檔案格式、系統資料表及媒體恢復方式。當資料庫軟體開始運行這些資料時都會一再確認。更何況在現實應用中每一個所開發出來的應用系統不一定可以穩定運作在這些不同版本的資料庫軟體，這也是值得我們注意的地方。

資料庫運行時間

怎麼做

進入資料庫命令模式並執行以下命令：

```
postgres=# SELECT date_trunc('second',current_timestamp -
pg_postmaster_start_time()) as uptime;
  uptime
----------
 00:40:28
```

資料庫運行時間講解

Postgres 啟動後會將儲存啟動時間：

```
postgres=# SELECT pg_postmaster_start_time();
pg_postmaster_start_time
-------------------------------
 2011-08-04 22:03:03.386874+08
```

所以我們可以將現在的時間減去資料庫啟動的時間即可得到資料庫共運行的時間：

```
postgres=# SELECT current_timestamp - pg_postmaster_start_time();
    ?column?
-----------------
 00:46:33.065991
```

再進行顯示格式調整就可以了：

```
postgres=# SELECT date_trunc('second',current_timestamp -
pg_postmaster_start_time()) as uptime;
  uptime
----------
 00:40:28
```

資料庫檔案架構

安裝完 PostgreSQL9 後，資料庫軟體目錄會列出主版本及次版本編號預設存放於以下目錄：

- ✧ /opt/PostgreSQL9/9.0/

資料庫建立後會將所有的資料存放在「資料目錄」(PGDATA)裏，當然我們也可以將這些資料存放在表格空間中(tablesapce)。

資料庫資料檔預設存放於以下目錄：

- ✧ /opt/PostgreSQL9/9.0/data

在此資料目錄下會包含所有關於資料庫所有設定檔(*.conf)，當然這些都是預設值，我們可以視需要進行調整。如透過 *initdb* 工具建立額外資料目錄。

資料庫檔案講解

各種不同的作業系統平台如：RedHat, Ubuntu 及 Debain，都會遵循 Linux 檔案系統階層標準(FHS, Filesystem Hierarchy Standard)，如不遵循此方式都是錯誤的。

當然資料庫管理系統也有自己對於檔案配置的標準，如最佳彈性結構(OFA, Optimal Flexible Architecture) 是由 Oracle 的一個概念，主要用途為了讓單一主機可安裝多種不同的資料庫版本軟體並建立多個資料庫，依照這個結構可以讓人猜出檔案的存放位置以及這檔案所代表的意義；透過 OFA 標準定義系統環境變數來代表各資料庫專有名稱所代表的路徑配置：

- ✧ export PGHOME=/opt/PostgreSQL9/9.0
- ✧ export PGPORT=5432
- ✧ export PGRELEASE=9.0
- ✧ export PGSERVERNAME=PGSERVER
- ✧ export PGDATA=$PGHOME/data

更多

當進入資料庫資料目錄(PGDATA)，將可看到有非常多的子目錄來組成一個資料庫。各目錄意義如下：

base	主要存放資料的目錄，此以下所建立的目錄代表各資料庫配置，來存放資料表及索引。
global	系統資料表(catalog)存放的地方，由各資料庫共同來使用。
pg_clog	交易狀態檔。
pg_multixact	Row-level lock 狀態檔
pg_subtrans	次交易狀態檔
pg_tblspc	連接至表格空間所建義的目錄
pg_twophase	"2-phase commit"或"Prepared transaction status"
pg_xlog	交易日誌檔(Write Ahead Log- WAL)

在以上這些目錄中強烈建議不要任意手動修改檔案或刪除檔案，僅管作業系統上可用空間不足夠，否則將造成不可回復的災難。至於如何備份拷貝這些目錄將在之後備份章節中來介紹。

唯一允許我們建立的是資料庫伺服器設定檔(*.conf)及資料庫伺服器日誌檔，但資料庫伺服器日誌檔可不一定要存放在資料目錄(PGDATA)中哦！也會在後面來介紹。

資料庫伺服器日誌檔

資料庫所有發生事件都會將記錄在資料庫伺服器日誌檔，當資料庫發生問題時，首先應該先檢查資料庫伺服器日誌檔以找出問題徵兆，當然在平時也最好有例行性檢查資料庫伺服器日誌檔的習慣。

以下為日誌檔的內容截取：

```
2011-08-05 10:54:48 CST LOG:  database system was shut down at 2011-08-05 10:54:35 CST
2011-08-05 10:54:48 CST LOG:  autovacuum launcher started
2011-08-05 10:54:48 CST LOG:  database system is ready to accept connections
2011-08-05 10:55:31 CST LOG:  unexpected EOF on client connection
2011-08-05 10:55:44 CST FATAL:  password authentication failed for user "postgres"
```

資料庫伺服器日誌檔解講

資料庫伺服器日誌檔可透過不同方式放置於不同路徑，所以我們可將資料庫伺服器日誌檔置於我們想要存放的路徑。

- 置於資料目錄(PGDATA)中。
- 置於其它系統檔案中，但該目錄需擁有權限存取。
- 透過 syslog 寫至系統日誌。

如果未透過 syslog 則資料庫伺服器日誌檔應該會有一個以上的日誌檔，我們可自依需求更改檔名及路徑，畢竟每台主機日誌檔配置不同。不同作業系統如：RHEL, CentOS, Debian 和 Ubuntu 所存放的路徑即不同，而 Windows 預設則會寫至事件檢事器。

資料庫伺服器日誌檔安全性等級

PostgreSQL 安全性等級	代表 意義	Syslog 安全性等級	Windows 事件檢視器
DEBUG 1~5	內部診斷	DEBUG	INFORMATION
INFO	執行命令後的輸出值	INFO	INFORMATION
NOTICE	一般提示資訊	NOTICE	INFORMATION
WARNING	警示訊息	NOTICE	WARRNING
ERROR	一般命令錯誤訊息	WARNING	ERROR
LOG	給系統管理者的訊息	INFO	INFORMATION
FATAL	Session 所記錄的訊息	ERR	ERROR
PANIC	發生 Crash 時	CRIT	ERROR

當資料庫伺服器日誌檔出現 FATAL 及 PANIC 時我們要特別的注意是不是資料庫發生了重大異常，當然某些情況或許可以排除如做資料庫複製或使用者進入資料庫時輸錯帳號密碼。

有些情況我們並不需要記錄全部等級的日誌則可以調整資料庫參數「log_min_messages」過濾哪些日誌安全性以上等級才進行記錄，預設值為 warning；而有些情況單純的記錄可能不符我們需求，如想知道使用者名稱甚至 PID 則可透過「log_line_prefix」來設定，預設值為%t；%t 代表 timestamp，另有%u 代表使用者、%p 代表 PID 及%d 代表資料庫名稱，故可用「%t[%p]%u@%d」來表示。

最後參數「log_error_verbosity」代表日誌檔裏面所要記錄的詳細程度，預設為 default，另有 terse 代表精簡則會忽略 Hint 及 Qeruy 等錯誤訊息。而 VERBOSE 則會連 SQLSTATE 的錯誤碼也寫記錄。

資料庫伺服器日誌檔所記錄的精細度往往取決與現有環境狀況及需求，當然有更詳細的資訊對於資料庫發生異常時有助於對問題的解決。

資料庫識別碼

資料庫識別碼((database's system identifier)，當一個資料庫伺服器建立完成後都會有一個唯一值，此值不可重複，可用於備份、資料庫複製(Clone)…等，甚至當資料庫產生 Bug 時也可以用此識別碼來辨別。

可透過以下命令來取得資料庫識別碼：

```
pg_controldata $PGDATA  |grep "system identifier"
Database system identifier:           5637199870721490939
```

不要使用參數-D 來表示資料庫的資料目錄，但可使用環境變數$PGDATA 或目錄絕對路徑來取得資料庫識別碼。

資料庫控制檔

透過 pg_controldata 此指令可列出 PostgreSQL 資料庫伺服器控制檔，此檔位置資料庫的資料目錄中(PGDATA)，於資料庫建立時一同建立此檔，記載了資料庫重要資訊並隨著資料庫異動而更新控制當。

以下列出資料庫控制檔的內容：

```
pg_control version number:            903
Catalog version number:               201008051
Database system identifier:           5637199870721490939
Database cluster state:               in production
pg_control last modified:             Fri 05 Aug 2011 10:59:49 AM CST
Latest checkpoint location:           0/15FCDE8
Prior checkpoint location:            0/15F74E8
Latest checkpoint's REDO location:    0/15FCDE8
Latest checkpoint's TimeLineID:       1
Latest checkpoint's NextXID:          0/671
Latest checkpoint's NextOID:          24576
Latest checkpoint's NextMultiXactId:  1
Latest checkpoint's NextMultiOffset:  0
Latest checkpoint's oldestXID:        655
Latest checkpoint's oldestXID's DB:   1
Latest checkpoint's oldestActiveXID:  0
Time of latest checkpoint:            Fri 05 Aug 2011 10:59:48 AM CST
Minimum recovery ending location:     0/0
Backup start location:                0/0
Current wal_level setting:            minimal
Current max_connections setting:      100
Current max_prepared_xacts setting:   0
Current max_locks_per_xact setting:   64
Maximum data alignment:               8
Database block size:                  8192
Blocks per segment of large relation: 131072
WAL block size:                       8192
Bytes per WAL segment:                16777216
Maximum length of identifiers:        64
Maximum columns in an index:          32
```

```
Maximum size of a TOAST chunk:        1996
Date/time type storage:              64-bit integers
Float4 argument passing:             by value
Float8 argument passing:             by value
```

列出資料庫

當我們進入 PostgreSQL 時，一次大部份只連接一個資料庫進行操作。假設在一台伺服器上不僅僅只有一個資料庫時對初學者而言如何分辨及連接正確的資料庫就很重要了。

於命令模式以下執行：

```
bash# psql -l
```

或進入資料庫後執行以下執令：

```
postgres-# \l
```

透過以上兩種任何方式都可以列出所有資料庫。

```
List of databases
   Name    |  Owner   | Encoding | Collation  |   Ctype    |   Access privileges
-----------+----------+----------+------------+------------+-----------------------
 a         | postgres | UTF8     | zh_TW.utf8 | zh_TW.utf8 |
postgres   | postgres | UTF8     | zh_TW.utf8 | zh_TW.utf8 |
template0  | postgres | UTF8     | zh_TW.utf8 | zh_TW.utf8 | =c/postgres          +
           |          |          |            |            | postgres=CTc/postgres
template1  | postgres | UTF8     | zh_TW.utf8 | zh_TW.utf8 | =c/postgres          +
           |          |          |            |            | postgres=CTc/postgres
(4 rows)
```

當然進入資料庫後命令模式後，透過 PortgreSQL 系統資料表(Catalog Table)，「pg_database」，可直接使用 SQL 語句查詢所有資料庫。

```
postgres=# SELECT datname FROM pg_database;
  datname
-----------
 template1
 template0
 postgres
(3 rows)
```

系統資料庫講解

當安裝完 PostgreSQL9 後會立即建立三個資料庫名為：「template0」，「template1」及「postgres」。「postgres 」為主要系統資料庫。

我們可透過以下此兩種方式命令來建立新資料庫：

```
bash# createdb my_dbname
```

或是

```
postgres=# CREATE DATABASE my_dbname;
```

當建立新資料庫時，事實上是由已存在的資料庫複製一份複本的，而該資料庫為「template1」;「template0」與「template1」均為範本資料庫，兩個最大差別在於我們可去修改範本資料庫:「template1」,來符合我們的需求，而「template0」主要用於當我們想還原「template1」此資料庫時的初始值時，透過「temploate0」即可。

關於更多資訊可透過 Postgres 系統資料表(Catalog Table)來獲得「pg_database」。比方資料庫預設的表格空間(Tablespace)，我們可將資料表或索引建立在表格空間，藉此分散硬碟 I/O 讀取效能，另外如資料庫的編碼(Encoding)及定序(Collation)，這些都與之後維護資料庫及執查詢語句時的排列順序悠關，會在後面章節有更詳細介紹。

列出資料表

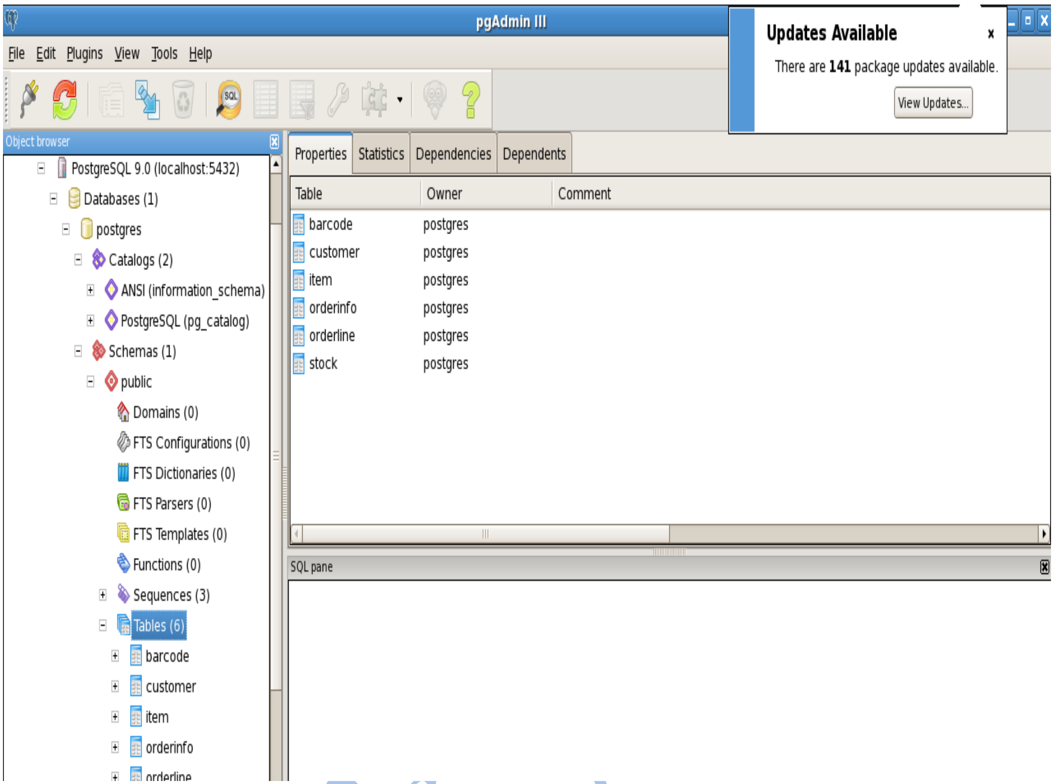
一個資料庫擁有多少資料表可做為一個資料庫是否複雜的標準，我們可透過以下 SQL 語句來列出所有資料表：

```
postgres=# SELECT count(*) FROM information_schema.tables
         WHERE table_schema NOT IN ('information_schema', 'pg_catalog');
```

當然也可以透過 psql 的 meta-command 列出所有資料表：
(meta-command 指的是進入 psql 後輸入為反斜線開頭的簡短指令)

```
postgres=# \dt
                   List of relations
 Schema |            Name            |   Type   |  Owner
--------+----------------------------+----------+----------
 public | barcode                    | table    | postgres
 public | customer                   | table    | postgres
 public | customer_customer_id_seq   | sequence | postgres
 public | item                       | table    | postgres
 public | item_item_id_seq           | sequence | postgres
 public | orderinfo                  | table    | postgres
 public | orderinfo_orderinfo_id_seq | sequence | postgres
 public | orderline                  | table    | postgres
 public | stock                      | table    | postgres
(9 rows)
```

如不熟悉指令，當然也可以透過圖型化工具如 pgAdmin3 或 phpPgadmin，如下圖：



PostgreSQL 將一些內部資訊儲存於一些系統資料表我們稱為「Catalog Table」，這些系統資料表主要位於資料綱要(Schema)「pg_catalog」裏，也位於另一資料綱要中「information_schema」，讓我們可以透過標準的 SQL 語法來查出我們所要的資訊。

計算資料庫佔用空間

我們常常會需要此資料庫的大小來監測或預估資料庫成長，可以透過此兩種方式來評估：

- ✧ 直接計算資料庫檔案所佔的空間。
- ✧ 使用 SQL 語法來查詢資料庫大小。

如果直接計算資料庫檔案所佔的空間，則必須包含資料目錄(PGDATA)及其子目錄，當然還要特別注意有無使用表格空間(Tablespace)，而表格空間(Tablespace)可能散落在各處而導致計算上的困難。

所以最簡單的方式就是透過以下 SQL 語法來查詢：

```
SELECT pg_database_size(current_database());
```

然而以上 SQL 語法只是查詢單一資料庫，如果想查所有資料庫則可以透過以下 SQL 語法查詢：

```
SELECT sum(pg_database_size(datname)) from pg_database;
```

資料庫伺服器理所當然會知道資料表位於哪裏？因而計算出每張資料表大小，故 pg_database_size()此函數可計算出資料庫所佔的空間。

計算資料表佔用空間

如何計算單一資料表大小？或是某些資料表所佔的空間大小呢？

我們可以透過以下 SQL 語法來查詢單一資料表所佔空間：

```
postgres=# SELECT pg_relation_size('accounts');
pg_relation_size
------------------
            8192
(1 row)
```

我們可以透過以下 SQL 語法來查詢除了單一資料表外包含與此資料表相關聯所佔空間如索引，全部所佔的空間大小：

```
postgres=#  SELECT pg_total_relation_size('dept');
 pg_total_relation_size
------------------------
                 73728
(1 row)
```

或者可以透過 psql 的 meta-command：

```
postgres=# \dt+ dept
                    List of relations
 Schema | Name | Type  |  Owner   |    Size    | Description
--------+------+-------+----------+------------+-------------
 public | dept | table | postgres | 8192 bytes |
(1 row)
```

PostgreSQL 中，資料表是由許多「relations」來構成的，主要的「relation」就是資料存放的資料表，也因此一張資料表會由許多資料檔來組成。而基於資料表所建立的每個索引也是稱為「relation」。Long data 的值則存放在第二張表格名為「TOAST」，因此每張表格就會有 TOAS 表格及 TOAST 索引。

每個 relation 是由多個資料檔來構成的，而資料檔最多至 1GB 則會再產生另一個檔。第一個檔案無字檔名結尾，第二個檔以上會有檔名結尾(如.2)，另外有結尾檔名如.vm 及.fsm 則分別代表為 Visibility Map 及 FreeSpaceMap；主要於維護資料時使用，此檔案所佔用空間很小，甚至該資料表很大時也是如此。

TOAST 代表為 Outsized Attribute Storage Technique。透過它可以讓資料庫來儲存長欄位的值，因此 PostgreSQL 允許許多資料類型最大可儲存至 1GB，PostgreSQL 透通的方式將大資料儲存於許多的小塊。因此同樣的資料類型儲存可以從 1Byte 可 1GB。

找出最大資料表

我們已在之前介紹過如何找出資料表所佔的空間大小，接著將延伸更廣泛的需求，如一連串相關資料表大小，而不是只是列出單一的資料表大小。

可以透過以下 SQL 語法查出前十大資料表(Top 10)：

```
postgres=# SELECT table_name,
pg_relation_size(table_schema||'.'||table_name) as size
FROM information_schema.tables
WHERE table_schema NOT IN('information_schema','pg_catalog')
ORDER BY size DESC
LIMIT 10;
```

由大至少列出前十大資料表，以上這個例出我們將會列出所有資料綱要中(Schema)的資料表，除了資料綱要 information_schema 及 pg_catalog 外，此兩資料綱要主要為 postgres 內部資料。

資料表筆數

許多人第一次體驗 PostgreSQL 查詢語句應該是執行 SQL 來計算資料筆數吧。

```
SELECT count(*) FROM table;
```

執行此 SQL 語法後將會回應一整數值如下：

```
postgres=# SELECT count(*) FROM orders;
count
-------
  345
(1 row)
```

資料表筆數講解

Count(*)函數時將會連續掃描(Sequential Scan)該張資料表每一列，我們將會接著一個的存取該張資料表每個區塊，讀取每一個區塊筆數，如果該資料表同時位於同一個硬碟上將有利於存取，而讓該語法執行速度相當快速。

有些人會認為測試資料庫效能最佳的方法就是執行「count」語法。因此有許多資料庫管理系統會針對「Count」來進行最佳化，但 PostgreSQL 並沒有針對它進行最佳化；許多人曾經對 PostgreSQL 專案團隊建議好幾次，但他們卻不認為應該為此進行最佳化。的確如此，Count 函數在應用上的確經常被執行，但使用 Count 語法後面卻不接 WHERE 語句是沒有意義的。因此 PostgreSQL 團隊選擇不進行最佳化。

主要原因是 PostgreSQL 提供一個非常好的特點 MVCC(Multi-Version Concurrency Control)，MVCC 允許執行 Insert, Update, Delete 並同時間執行 Count 語法讓我們可以掃該資料表中的每一個區塊。

假設當我們執行 SQL 查詢後才進行資料異動，MVCC 會記錄該資料表中每一列異動，以便查詢忽略掉這些異動；仔細來思考，當我們執行 SQL 查詢期間該資料表已經過多次異動。Count 語法只是這段時間異動的某一個時間點筆數而已。

快速計算資料表筆數

對一個非常大的資料表而言，我們不需要每次很精確的使用 Count 語法來計算總筆數，往往會花費很多時間。資料庫管理者經常會去評估這樣的動作會花費多少的時間，避免造成效能低落。

我們可透過 PostgreSQL 優化器(Optimizer)快速評估一張資料表總筆數，語法如下：

```
SELECT (CASE WHEN reltuples > 0 THEN
pg_relation_size('emp')/(8192*relpages/reltuples)
ELSE 0
END)::bigint AS estimated_row_count
FROM pg_class
WHERE oid = 'emp'::regclass;
```

透過這樣方式可以很快速到得資料表總筆數，也不用擔心因資料表太大而造成效能低落。

快速計算總筆數講解

在之前已介紹過 pg_relation_size()函數，它可以精確的計算出該資料表所佔空間。

當我們執行 **vacuum** 資料表時，會記錄二筆資訊至 pg_class 系統資料表內。此二筆分別為該資料表所有資料區塊(relpages)及筆數(reltuples)。許多人認為可以使用 pg_class 中的 relpages 值來評估總筆數，但如沒定期執行 vacuum 將造成系統資料表 pg_class 中的值過期而評估錯誤；pg_stat_user_tables 也同樣會有這樣情形。我們將會資料庫效能章節中來介紹。

PostgreSQL 優化器(Optimizer)使用資料區塊(relpages)及筆數(reltuples)來計算每一個區塊的所擁有平均筆數及筆數分佈密度。

假設每一個區塊的筆數分佈密度是最新的，則我們可透過以下公式來評估該資料表總筆數：

```
Row Estimate = Number of Data Blocks * Rows per Block
總筆數 = 總資料區塊數量 X 每一個區塊的筆數。
```

故利用 PotgreSQL 優化器(Optimizer)來計算 reltuples 及 relpages 機乎不會影響到 PostgreSQL 的效能，用此方式來計算資料表總筆數是一個很好的方法。

了解物件相依性

在資料庫中每個物件都會相依關係，有時候我們需了解這些相依關係來找出如何執行某些動作。以下將詳細介紹。

以下簡單例子會讓我們了解並觀察物件相依關係，有兩張資料表如下：

```
CREATE TABLE orders( orderid integer PRIMARY KEY);
```

```
CREATE TABLE orderlines(
orderid integer
, lineid   smallint
,PRIMARY KEY (orderid, lineid)
);
```

現在增加此兩張資料表的參考完整性關聯：

```
postgres=# ALTER TABLE orderlines ADD FOREIGN KEY (orderid)
REFERENCES orders(orderid);
```

假設我們試圖刪除該資料表會得到以下訊息：

```
postgres=# DROP TABLE orders;
錯誤:  無法刪除 資料表 orders，因為有其他物件依存於它
DETAIL:  限制 orderlines_orderid_fkey (在 資料表 orderlines) 依存於 資料表
orders
HINT:  使用 DROP ... CASCADE 刪除依存物件。
```

特別注意！假如我們按照提示(Hint)所做則會意外刪除所有與「orders」相關聯的資料表。你可能覺得「Drop...CASCADE」是不錯的想法如果懶得去找有關資料表「orders」相關聯的資料表，但得先確保所有與資料表「orders」相關聯的資料表是你所不需要的，但僅慎點的做法是手動一個一個刪除與資料表「orders」相關的資料表。

我們可以透過以下 psql meta-command 來顯示關於資料表所有限制約束(Constraints)以及與它相關聯：

```
postgres=# \d+ orders
               Table "postgres.orders"
 Column  |  Type   | Modifiers | Storage | Description
---------+---------+-----------+---------+-------------
 orderid | integer | not null  | plain   |
Indexes:
    "orders_pkey" PRIMARY KEY, btree (orderid)
Referenced by:
    TABLE "orderlines" CONSTRAINT "orderlines_orderid_fkey" FOREIGN KEY
(orderid) REFERENCES orders(orderid)
Has OIDs: no
```

我們也可以透過以下 SQL 語法來查詢：

```
SELECT * FROM pg_constraint
WHERE confrelid='orders'::regclass;
```

不幸的到此並還未結束喔！繼續以下的解說。

當我們建立的外鍵(Foreign Key)，則會在 pg_constraint 系統資料表中加入此限制約束，因此我們可以透過以下查詢找出關於此資料表所有的限制約束及相關聯的資料表：

```
WHERE confrelid = 'orders'::regclass;
```

介紹物件分辨的類基，了解此函數 relname2relid()的用法。

假設我們執行以下語法：

```
DROP TABLE orders;
```

CHAPTER 3

設定資料庫

本章內容大綱如下：

- 讀取 RTFM(Reading the Fine Manual)
- 規劃新資料庫
- 設定資料庫參數檔
- 列出線上資料庫參數
- 列出非預設資料庫參數
- 載入資料庫檔參數檔
- 設定資料庫安裝環境
- 增加 PostgreSQL 額外模組
- 資料庫省電模式

前言

我總是被問到如何設定 PostgreSQL 參數檔。

人平時總是忙錄的所以大多數的人希望只要花短短五分鐘的時間就可以完成這些工作，而這本書所要帶給大家的。

許多人相信有一些神奇的資料庫參數將會大大提昇效能，所以呢！花一些時間閱讀完本書將會為大家帶來更明確方向。而不是像有些人僅透過網路尋找答案就認為他們的資料庫設定妥當

在許多方面，有些設定值是容易被了解的；但要讓這些設定值達到最佳化卻是相當的困難，此章節將會介紹大家了解設定值且何時去使用它。

規劃新資料庫

規劃新資料庫是一個令人挑戰的任務，容易讓人失去信心；因此這裏會介紹大家一些規劃的想法，當得到這些觀念後，讓大家日後規劃資料庫時會思考週道。

此書將會含概這些項目：

- 資料庫設計：
  - 評估資料庫未來成長量及大小
- 交易分析：如何存取資料庫
  - 評估最常的交易行為
  - 需要多少交易回應時間
- 硬體設定：
  - 記憶體是否夠應付這些資料量
- 本土化規劃：
  - 資料庫編碼、語系及時區
- 存取和安全規劃：
  - 規劃客戶端及指定需要的驅動程式
  - 建立符合安全性的資料庫角色
  - 設定 pg_hba.conf 設定檔
- 維護規劃：什麼是持續要做的事以及如何做？
- 可用規劃：考量哪些可用性需求
  - checkpoint_timeout
  - 規劃備份及測試還原
- 高可用性規劃：
  - 選擇資料庫複製方式

其中最重要一個理由是規劃資料庫會花很多時間且困難。尤其是資料庫編碼及本土化設定，假如一開始沒規劃好，事後想更改這些設定，則資料庫必須重新建立而造成時間上的浪費及資料庫停擺(Downtime)。然而建立完資料庫系統後，安全性也是一個重大的課題。

在規劃資料庫當中很多人是需要幫助，你可能在此段時間知道要做哪些事情，但不見得別人也知道。在你要做這些事之前告訴其它人你要做什麼，避免溝通上問題而造成浪費時間？假設你不大確定做哪些事，就應該整理出一張列表哪些是你要做的，並且哪些事情不知道要做的也一併列出來，如此才會讓規劃資料庫順利些。

資料庫伺服器參數值套用層級

更改資料庫伺服器參數設定值時可分為會議(Session) 階段及交易(Transaction) 階段：

✧ 會議(Session) 階段：

以下為修改參數 work_mem 並套用於會議(Session) 階段：

```
SET work_mem = '10MB';
```

當套用參數值於會議(Session) 階段時，此參數值會持續生效直到執行以下 SQL 語法或該使用者登出：

```
RESET work_mem;
```

```
RESET ALL;
```

✧ 交易(Transaction) 階段：

以 SET LOCAL 執行，故名思意就是參數值僅在交易期間生效，當離開交易時則參數值會回到原本資料庫伺服器參數原本設定值：

```
BEGIN;
SET LOCAL work_mem = '10MB';
COMMIT;
```

如何運作…

以下為將介紹當套用參數值時會議(Session) 階段與交易(Transaction)階段的差別：

會議(Session)階段：

在套用參數 work_mem 設定值為 10MB 後，查詢系統資料表 pg_settings 如下：

```
SET work_mem = '10MB';
postgres=# SELECT name, setting, reset_val, source FROM pg_settings WHERE
source = 'session';
   name   | setting | reset_val | source
----------+---------+-----------+---------
 work_mem | 10240   | 1024      | session
```

透過查詢結果得知，於此會議(Session)階段參數 work_mem 為 10MB，且此參數預設為 1MB；接著重設參數 work_mem，將它還原為預設值：

```
postgres=# RESET work_mem;
```

再次查詢欄位 source 由值「session」更改為「default」:

```
postgres=# SELECT name, setting, reset_val, source FROM pg_settings where
name ='work_mem';
   name   | setting | reset_val | source
----------+---------+-----------+---------
 work_mem | 1024    | 1024      | default
```

✧ 交易(Transaction) 階段：

剛介紹過交易(Transaction) 階段僅在於交易期間生效，所以如未在交易(Transaction) 階段執行套用參數，則此參數並不會生效：

```
postgres=# SET LOCAL work_mem = '10MB';
postgres=# SELECT name, setting, reset_val, source
           FROM pg_settings
           WHERE name ='work_mem';
name | setting | reset_val | source
------+---------+-----------+--------
```

改以在交易(Tranctaion)階段套用參數，就可以得到參數 work_mem 已變更：

```
postgres=#
BEGIN;
   SET LOCAL work_mem = '10MB';
   SELECT name, setting, reset_val, source
   FROM pg_settings
   WHERE name ='work_mem';
COMMIT;
   name    | setting | reset_val | source
----------+---------+----------+---------
 work_mem | 10240   | 1024      | session
```

一般而言資料庫伺服器參數可分為好幾個層級，可透過系統資料表 pg_settings 中的欄位 source 查詢，包含：「environment variable」、「override」、「client」、「configuration file」、「default」及「session」，最常使用的情況就是剛所介紹的會議(Session)階段與交易(Tranctaion)階段，當然未執行任何修改參數動作則以資料庫伺服器參數檔 postgresql.conf 中的設定值為主，至於參數未列在資料庫伺服器參數檔中呢？就是以資料庫伺服器預設參數值囉！每個層級都代表不同意義使用方式也不同，可要分辨的很清楚喔！避免造成誤會。

找出資料庫伺服器參數設定值

DBA 進行一些例行性維設或管理時常常需要知道資料庫伺服器參數設定值，故本節教導大家以下幾種方式，讓大家可以快速查詢資料庫伺服器參數設定值。

第一個想到的方法就是透過資料庫伺服器設定檔 postgresql.conf 得知，然而如每次都要透過編輯器查看資料庫伺服器設定檔或許有些麻煩，且當有多個資料庫伺服器設定檔時更可能造成誤判，故最快最直接的方式就是直接透過資料庫伺服器查詢，以下列出這幾種方式介紹給大家：

執行以下 SQL 語法查詢資料庫伺服器參數設定值，由於輸出資料太多就不列給大家看囉！

```
postgres=# SHOW ALL;
```

也可指定已知的參數，只查詢該參數設定值：

```
postgres=# SHOW work_men;
work_mem
----------
 1MB
```

除以上方式外，PostgreSQL 提供系統資料表 pg_settings 方便 DBA 查詢，首先透過 meta-command 將輸出延展，接著再查詢參數 work_mem 設定值：

```
postgres=# \x
postgres=# SELECT * FROM pg_settings WHERE name = 'work_mem';
-[ RECORD  ]----------------------------------------------------------
name        | work_mem
setting     | 1024
unit        | kB
category    | Resource Usage / Memory
short_desc  | Sets the maximum memory to be used for query workspaces.
extra_desc  | This much memory can be used by each internal sort operation
and hash table before switching to temporary disk files.
context     | user
vartype     | integer
```

```
source      | default
min_val     | 64
max_val     | 2147483647
enumvals    |
boot_val    | 1024
reset_val   | 1024
sourcefile  |
sourceline  |
```

列出已更改過的資料庫伺服器參數值

當資料庫伺服器安裝完成後，基本上資料庫伺服器每個參數都擁有自己的預設值，而在日後可能因管理或效能上等其它因素故需修改資料庫伺服器參數預設值，然而要如何查詢哪些參數已被我們修改呢？執行以下 SQL 語法可以列出所有已被修改過後的資料庫伺服器參數值：

```
postgres=# SELECT name, source, setting
           FROM pg_settings
           WHERE source != 'default'
           AND source != 'override'
           ORDER by 2, 1;
           name             |        source        |     setting
-----------------------------+----------------------+------------------
--
 application_name            | client               | psql
 client_encoding             | client               | UTF8
 DateStyle                   | configuration file   | ISO, MDY
 default_text_search_config  | configuration file   | pg_catalog.english
 lc_messages                 | configuration file   | en_US.UTF-8
 lc_monetary                 | configuration file   | en_US.UTF-8
 lc_numeric                  | configuration file   | en_US.UTF-8
 lc_time                     | configuration file   | en_US.UTF-8
 log_autovacuum_min_duration | configuration file   | 0
 log_duration                | configuration file   | on
 log_line_prefix             | configuration file   | %t[%p]%u@%d
 log_statement               | configuration file   | ddl
 max_connections             | configuration file   | 100
 max_prepared_transactions   | configuration file   | 10
 shared_buffers              | configuration file   | 4096
 log_timezone                | environment variable | ROC
 max_stack_depth             | environment variable | 2048
 TimeZone                    | environment variable | ROC
```

載入資料庫檔參數檔

資料庫參數值主要存放在 PostgreSQL 的參數設定檔裏;所有參數可透過參數檔來設定，名為 postgresql.conf。

另有兩個設定檔分別為 pg_hba.conf 及 pg_ident.conf。主要針對連線及安全性，以下將為大家做些介紹。

透過 postgresql.conf 參數檔我們可以為資料庫設定所有參數值。有些參數值設定完後須將資料庫重啟後方能生效，如 shared_buffers，定義資料庫共享快取記憶體。

而另一些參數值則可以在資料庫運行同時動態生效,我們只要在編輯所要修改的資料庫參數後，執行「reload」指令讓 PostgreSQL 重新載入參數檔 postgresql.conf 即可。如以下指令：

```
pg_ctl -D data reload
```

postgresql.conf 是一個純文字檔可直接透過文件編輯器修改，大部份資料庫參數已定義在該檔中，故只要搜尋想修改的參數值並將註解拿掉即可。

資料庫參數檔講解

假如你不小心在資料庫參數檔中設定同一參數值二次,則只有最後一個設定值才會生效。這樣最大的好處，當你最後參數值設定感到混亂時，只要從結尾增加參數值即可。

最好的方法是建立一個新設定檔並只列出我們所想設的參數值,這會讓設定檔變的容易管理許多。

無論使用哪個方法,強烈建議在編輯設定檔前先行備份,你可以直接使用複製的方法或使用版本控制系統(Version control system)如 SVN。

更多…

資料庫參數檔 postgresql.conf 也支援呼叫外部檔，允許 postgresql.conf 參考至外部檔，這將有助於我們簡化及組織參數檔設定。

依群組設定資料庫參數

PostgreSQL 支援為不同的使用者群組設定資料庫參數值。

如何做…

針對資料庫「newdb」全部使用者：

```
postgres=# ALTER DATABASE newdb SET work_mem=1024;
ALTER DATABASE
```

針對特定特定使用者，全部資料庫接受影響：

```
ALTER ROLE foo SET work_mem = 1024;
```

針對特定資料庫中的特定使用者：

```
ALTER ROLE foo IN DATABASE newdb SET work_mem = 1024;
```

往往使用者並不知道自己套用的參數值為何？雖大部份為預設值，但也可以適自己需求去設定自己想用的資料庫設定值。

資料庫參數如何運做…

資料庫設定值可套用在以下幾個層級：

- ✧ 資料庫(Database)
- ✧ 使用者(PostgreSQL 稱為 Role)
- ✧ 資料庫結合使用者

資料庫設定值預設會依照以下順序被覆蓋，也就是最後面的設定值最後生效：

- ✧ 當使用者連接資料庫 newdb 時，則資料庫層級參數生效
- ✧ 當使用者「foo」連接資料庫時(newdb 除外)，使用者層級參數生效
- ✧ 當使用者「foo」連接資料庫「newdb」時，資料庫結合使用者層級生效。

但別忘記囉！當使用者執行 SET 語法設定資料庫參數，需要重新連接資料庫方能讓參數生效喔！

準備資料庫伺服器環境

由於 PostgreSQL 是一個專屬的資料庫管理系統用於服務，為了確保 PostgreSQL 可順利運行並發揮伺服器全部資源。身為系統管理員應該避免除資料庫管理系統外的操作以及調整系統參數最佳化以發揮最高效能。

在開始之前，我們需要知道以下二點：

- ✧ 伺服器所有記憶體應全讓 PostgreSQL 使用
- ✧ PostgreSQL 應用於何種系統類型

如何做

資料庫大於 32MB，就應該將增加資料庫共享記憶體(shared_buffers)提昇效能；固然共享記憶體愈大愈好但當值設定太大時可能造成作業系統虛擬記憶體切換(swap)，加上 Linux 系統上，此共享記憶體如未完全被使用作業系統會將未使用到的資料庫共享記憶體置換排出，反而造成硬碟 I/O 頻繁而造成效能低落，故設定資料庫共享記憶體時應保守點，最好方法是一步一步漸近增加，並確保該設定值可達到最佳效益。

假設我們安裝資料庫於非 Windows 作業系統上，首先第一步應調整系統核心參數「SHMMAX」，如此才能正常設定資料庫共享記憶體。

於 Linux/Mac OS/FreeBDS 作業系統上我們可直接編輯/etc/sysctl.conf 設定檔或使用命令 sysctl -w 來設定系統核心參數。

- ✧ Linux: kernel.shmmax=value
- ✧ Mac OS: kern.sysv.shmmax=value
- ✧ FreeBSD: kern.ipc.shmmax=value

比如在 Linux 系統上，在此檔案/etc/sysctl.conf 增加此行：

```
kernel.shmmax=value
```

shmmax 為定義允許容納資料庫共享記憶體區段的最大實體記憶體，shmmax 設定值應該大到足以同納一個資料庫共享記憶體區段，此設定會避免會有多個以上的資料庫共享記憶體而造成效能降低。

至於參數值 effective_cache_size 到不用太擔心，過於煩惱設定，這不是一個很重要的參數。

假設你正在管理一個交易行為較重的資料庫，則應將 wal_buffers 參數設定較高的值。

假設你正在管理一個交易行為較重且會載入大型資料量的資料庫，則應將 checkpoint_segments 參數設定較高的值。

假設你正的資料庫會有大資料庫量查詢量大，則應將 work_mem 參數設定較高的值。

切記 autovacuum 設定值將它開啟，除非你有很好的理由將它關閉，但我想大多數的人是沒有的，我將會後面章節有更多 autovacuum 的介紹。

如欲知更多關於效能調校資訊，可參考以下網址：

- http://pgfoundry.org/projects/pgtune/

尤其不要改變 fsync 參數預設值，它會悠關資料庫運行保持資料的完整性。

想獲得更多關於設定作業系統核心參數可參考以下網址：
http://www.postgresql.org/docs/current/static/kernel-resources.htm

增加 PostgreSQL 額外模組

PostgreSQL 最引以為傲的事就是它的功能擴充性吧！從 1980 開始功能擴充性就已成為 PostgreSQL 所努力的目標，直到現在 PostgreSQL9.0 已經有許多模組被置入 PostgreSQL 的核心中。

以下列出額外模組擴充：

- ✧ 額外功能(Functions)
- ✧ 額外資料類型(Datatypes)
- ✧ 額外運算子(Operators)
- ✧ 額外索引(Indexes)

準備開始安裝額外模組

首先，選擇一個適當的模組

注意喔！一般而言當我們用一般方式安裝完 PostgreSQL 後一些工具及客戶端程式都只是預設值安裝而已並無其它額外功能；而這邊所提到的是充擴這些原有功能如 SQL 語法、函數和功能，進而符合我們需求及彌補原由不足地方。

- ✧ Contrib－PostgreSQL 核心裏面包含許多功能程式，官方會將功能程式放置於此目錄，如有額外功能模組安裝只需將它複製於此目錄下即可
  - ➢ http://www.postgresql.org/docs/9.0/static/contrib.html
- ✧ pgFoundry－由自由組織針對 PostgreSQL 程式發展所成立的網站，提供一些額外的 PostgreSQL 模組與工具，就如同 SourceForget.net 一樣
  - ➢ http://pgFoundry.org/
- ✧ 獨立專案－大型專案如 PostGIS，提供強大複雜 PostgreSQL 模組
  - ➢ http://www.postgis.org/

當我們透過 OneClick 安裝 PostgreSQL 時，一般模組在安裝期間預設會被一併安裝，除了某此情況須要額外安裝套件，如 Oracle 相容模組 http://www.postgres.cz/index.php/Oracle_functionality

此套件可透過以下網址下載，接著以命令模式執行 rpm 安裝。

http://pgfoundry.org/frs/?group_id=1000113&release_id=1856

在某些情形下許多功能不錯的模組並非擁有完整的封裝，在以上案例中我們就需要手動去安裝它。雖然手動安裝不是很困難，不過此案例可以讓我們了解它的原理。

安裝模組主要透過以下兩種方式達成：

- ✧ 建立該模組的資料庫物件
- ✧ 建入該模組動態函式庫並載入

許多模組機乎都是透過以上方式來安裝，以下範例會經由安裝 AutoExplain 來教導大家如何建立資料庫物件及載入函式庫

- ✧ 建立函式庫

定義建立資料庫物件後所存放路徑並載入函式庫：

- ✧ shared_preload_libraries = '$libdir/modlib'
  建立資料庫物件
  psql -d dbname -f SHAREDIR/contrib/module.sql

如何運作

PostgreSQL 可以透過下方式載入動態函式庫：

- 連接資料庫後使用 load 指令載入函式庫：

```
postgres=# load '/opt/PostgreSQL/9.0/lib/utl_file.so';
LOAD
```

- 於資料庫參數檔定義參數「shared_preload_libraries」路徑讓資料庫開啟時載入函式庫。
- 連接資料庫後定義參數「local_preload_libraries」路徑讓指定使用者載入函式庫路徑。

```
ALTER ROLE rolename  SET local_preload_libraries = value
```

PostgreSQL 程式功能和物件會參考到函庫，而這些擴充模組與資料庫伺服器能夠緊密結合並順利運行將會讓資料庫提昇更好效能。

資料庫省電模式

節省能源消耗是近年來火紅的主題，當然 PostgreSQL 也有針對此議題設計。

假使 PostgreSQL 是運行在行動裝置如 NB 或是建立在虛擬伺服器中，而又很少運做，大部份處於閒置，則可以參考下以方式來減少資源消耗。

PostgreSQL 是一個伺服器架構的資料庫，如果沒有任何客戶端連接資料庫對 PostgreSQL 而言是機乎沒有動作的。為了讓伺服器有更少的運作，可以參考以下來設定資料庫參數檔 postgresql.conf：

- autovacuum = off
- wal_writer_delay = 10000
- bgwriter_delay = 10000

此設定並非對所有使用者而言是最佳化的，只適合用於為了降低 PostgreSQL 活動。假設該伺服器日後回到忙錄狀態則需重設該設定值。

如何做…

對 PostgreSQL 而言當資料庫開啟時，會有一些執行程序是會一直持續運作的，但除了以下執行序是由客戶端連接資料庫時才會運做：

- Writer (又稱為 Background writer)
- WAL writer(用於日誌檔寫入)
- Archiver(只有封存日誌模式啟用時才會產生)
- WAL receiver Process(使用資料庫複製 Streaming replication 才會產生)
- Autovacuum process

Backgruond writer 執行程序預設每 200 毫秒(ms)執行一次，最大值可設定 10 秒(s)，此設定無法設定太大，因為當 bgwriter_lru_maxpages = 0 時 background writer 執行程序是可以被停用。

WAL writer 執行程序預設每 200 毫秒(ms)執行一次，最大值一樣可設定 10 秒(s)，此執行程序是無法被停用的，如果資料庫沒有任何寫入動作，則此執行程

序會閒置，但有寫入動作時則會開始執行並檢查。

Archiver 執行程序每 15 秒(s)執行一次並檢查是否有無新封存日誌檔(WAL)被寫入，此執行程序會造成系統 I/O 活動，運作時間無法透過參數來設定。

WAL receiver 執行程度每 100 毫秒(ms)檢查遠端資料庫異動資料是否送達，如果沒有接收到任何新異動資料，則該執行程序會呈現閒置狀態，運作時間無法透過參數來設定。

Autovacuum 預設每 60 秒(s)執行一次，可透過參數 autovacuum_naptime 來修改間隔時間，藉由 autovacuum = off 可將 Autovacuum 停用。

總結以下，假設你使用資料庫複製 Streaming replication，則伺服器最久應為每 100 毫秒(ms)運作一次；而如未使用資料庫複製 Streaming replication 則可讓資料庫最久每 10 秒(s)運作一次，而不是預設值每 200 毫秒(ms)運作一次。

CHAPTER 4

伺服器控制

本章內容大綱如下：

- 手動啟動資料庫
- 安全的快速關閉資料庫
- 緊急關閉資料庫
- 重新載入資料庫設定
- 快速重啟資料庫
- 禁止新連線
- 限制只有一個使用者
- 踢除使用者
- 決定
- 使用多重資料綱要
- 給使用者私人資料庫
- 單一主機運行多重資料庫伺服器
- 設定連接池

前言

PostgreSQL 是眾多執行程序所組成，主要執行程序稱為「postmaster」，當啟動資料庫時會建立這些執行程序，關閉資料庫時則中止這些執行程序。

每個「postmaster」執行程序會產生一個監聽埠(port)並接受客戶端連線，而如單一主機運行多重資料庫伺服器時，是無法共用該監聽埠的，故每一「postmaster」會產生一個唯一的監聽埠。

當我們啟動資料庫時，會指向一個資料目錄，該目錄是資料庫主要運作的核心，除了子目錄「tablespace」外，該子目錄會連結至資料目錄以外的資料；每個資料庫伺服器只能擁有一個資料目錄，而資料目錄是由這伺服器運行的也稱為實例(Instance)。

為了於資料庫伺服器任何動作，我們必須知道何謂資料目錄，如此才能啟動或關閉資料庫，當然也可以重新啟動資料庫，但其實就是連續執行關閉再啟動的動作，除此之外也可以重新載入資料庫，也就是重新讀取資料庫伺服器設定檔。

PostgreSQL 預設監聽埠為 5432，該監聽埠已註冊於 Internet Assigned Numbers Authority(IANA)，所以作業系統應保留該監聽埠以提供 PostgreSQL 資料庫服務。假使不止一個資料庫伺服器也稱為實例(Instance)最簡易的方式就是使用下一個順序的監聽埠 5433。

在上面以提到資料庫伺服器也稱為資料庫實例，但在 PostgreSQL 中其實稱為資料庫叢集(Cluster)，但我不建議使用這個名詞，因一般人往往會被該名詞搞混，以為是多個結點服務一個資料庫的叢集伺服器。其實它只是單一主機運一個資料庫而已。

手動啟動資料庫

一般而言 PostgreSQL 會在作業系統開啟後自動啟動，當然也可以選擇手動關閉啟動資料庫，因為在現實情況中，可能因為某些原因必須手動關閉或啟動。

準備開始

首先必須了解「服務」及「伺服器」的差別，「伺服器」指的是資料庫伺服器及運行的執行程序，而「服務」是由作業系統包裹後的稱呼。資料庫伺服器本來就可以運行在不同的作業系統平台上，而不同的作業系統及 distribution 會由獨特的服務啟動設定。

如何…

於所有作業系統平台上，可以使用以下指令來啟動資料庫

```
pg_ctl -D $PGDATA start
```

但不同作業系統上針對於服務啟動卻有不同的方式：

RedHat / Fedora   service postgresql9 start
Windows   net start postgres

於 Ubuntu 或 Debian 中，則被包裹成 pg_ctlcluster 工具，透過該工具同時啟動多個資料庫伺服器就方便了許多。

安全快速關閉伺服器

我們可以透過執行以下指令快速關閉資料庫伺服器：

```
pg_ctl -D $PGDATA -m fast stop
```

假使我們想盡量快速又安全的關閉伺服器則須指定參數-m fast，正常關閉指的是「等待」所有使用者結束並離開，這會花費一段時間。

如何運作…

當我們快速關閉資料庫時，所有使用者交易將被停止(Transactions abort)並連線中斷，這是所謂的「clean shutdown」，對資料庫伺服器而言並非強制終止該主要執行程序，故較為安全。

PostgreSQL 就像其它資料庫系統一樣，在關閉資料庫伺服器前須做 checkpoint 並關閉資料庫，如此才能讓資料庫快速啟動。

有一點 PostgreSQL 和其它資料庫軟體如 Oracle, DB2 和 MSSQL Server 不同的地方是交易回滾(Transaction rollback)相當快速，然而其它資料庫軟體使用這個模式關閉資料庫伺服器，則會等待一段時間甚至更久方能關閉。這是唯一不同的一點，但並不代表 PostgreSQL 的作法是不安全的。

於 Debian 或 Ubuntu 有一個不同的指令參數-- force，首先嘗試使用 fast 模式關閉，如失敗使用 immediate 模式關閉，最後如再無法關閉則直接刪除「postmaster」執行程序。

-

緊急關閉資料庫伺服器

如果發生緊急狀況時則需要透過以下指令，使用 immediate 模式緊急關閉：

```
pg_ctl -D $PGDATA -m fast stop
```

如何做…

當我們使用 immediate 模式關閉資料庫伺服器，所有使用者的交易將被停止並中斷連線，但這不是「clean shutdown」；immediate shutdown 就像資料庫發生 crash，可能造成資料庫毀損，故在重啟後需做復原(recover)，所以資料庫在經歷過 crash 後需要進行 recovery 程序才能再次開啟。

注意！ 針對 Oracle DBA 而言，PostgreSQL 的 immediate 模式就如同 Oracle shutdown abort，這可是不同的喔！

重新載入資料庫伺服器設定檔

PostgreSQL 有些設定檔只能透過 reload 將資料庫伺服器設定檔重新載入。

我們可透過以下指令之一來將資料庫伺服器設定檔重新載入：

```
pg_ctl -D $PGDATA reload
service postgresql reload
```

另外也可以在仍連接資料庫的同時重新載入資料庫伺服器設定檔，需要有 superuser 的權限喔！

```
postgres=# SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 row)
```

透過執行 SQL 語法的方式重新載入資料庫伺服器設定檔，是否讓你聯想到今天增加一個新的權限限制，讓該使用者無法登入資料庫，現有的使用者是不會被強制登出的，但在該使用者斷線後將會無法重新連線資料庫。

如何運作

為了重新載入資料庫伺服器設定檔，作業系統會傳送 SIGHUP 信號給 postmaster，當 postmaster 接收到後便會重新載入資料庫伺服器設定檔。

查看系統資料表「pg_settings」，可看到一個名為「context」的欄位名稱，透過這個欄位可以查閱出哪些參數得在資料庫重啟後才生效而又有哪些參數可以透過「sighup」動態載入生效。

以下 SQL 語法可查出可動態載入的參數：

```
SELECT name, setting,unit, (source = 'default') as is_default
FROM pg_settings
WHERE context = 'sighup'
AND setting != '0';
```

快速重啟資料庫

上節我們有提到某些資料庫參數是一定要在資料庫重啟後方能生效，所以如何快速重啟資料庫就變成一個非常重要的事了，最好是能夠將時間縮得愈短愈好。

執行以下的指令將重啟資料庫：

```
pg_ctl -D $PGDATA restart -m fast
```

重啟資料庫對一般而言似乎聽來很簡單，就只是關閉後再開啟而已，事實上是嗎？在單純的環境上的確非常的簡單，但在資料庫非常繁忙的情況下可不見得是如此喔！這也是提出如何快速重啟資料庫的理由。

首先，需要快速的關閉資料庫，不要使用 smart 模式，否則伺服器會等待使用者完成交易並結束方能停止；也不要使用 immediate 模式，雖然會很快速的將資料庫伺服器關閉，但會造成資料庫 crash，而導致下次開啟時需要將資料庫復原，反而時間變長。

正在運行中的資料庫伺服器擁有多已被修改過的快取區塊(Dirty)，PostgreSQL 就如其它資料庫軟體一樣，在資料庫被關閉前需要執行 checkpoint，如此在下次才能快速開啟資料庫。

如果我們在資料庫執行 checkpoint，則在關閉資料庫所花費的時間將比只單純執行 shutdown 快的許多，因為所有 dirty 的 shared_buffers 已被寫入至硬碟，以下請使用資料庫的 superuser 帳號執行命令：

```
psql -c "checkpoint"
```

下一階段需要考量的是當我們關閉資料庫伺服器後再啟動，資料庫伺服器快取記憶體已被清空，此時資料庫效能將會變低，得運行一段時間後才能再把資料快取在快取記憶體內，才能恢復原本效能，如果快取記憶體設定很大的話，可能得花費半小時以上才能，在一個大型企業中是無法接受如此衝擊的。

psql -c "SELECT pg_cache_save('mycache');
pg_ctl -D $PGDATA -m immediate restart;
pg_ctl -c "SELECT pg_cache_warm('mycache');

禁止資料庫新連線

在特殊緊急情況下，可能需要避免特定使用者存取資料庫，我們很難預先可能會遭遇到哪些情況，所以以下列出幾種可能性：

- 暫停連接池(Connection pools)
- 關閉資料庫伺服器，不建議此方法。
- 藉由將資料庫連線限制設為 0，讓資料庫 newdb 無法接受新連線。

  指令如下：

```
ALTER DATABASE newdb CONNECTION LIMIT 0;
```

  此方式只限制一般使用者(Normal user)，超級使用者(Superuser)一樣可以連線。

- 藉由將特定使用者連線限制設為 0，讓特定使用者無法建立新連線。

```
ALTER USER foo CONNECTION LIMIT 0;
```

  此方式一樣只限制一般使用者(Normal user)，超級使用者(Superuser)一樣可以連線。

  修改 HBA(Host-based authentication)檔來禁止新連線，但需重新載入設定檔。

  修改設定檔 pg_hba.conf 如下：

```
# TYPE  DATABASE        USER            CIDR-ADDRESS            METHOD
local   all             postgres                                md5
local   all             all                                     reject
host    all             all             0.0.0.0/0               reject
```

  注意！為了避免禁止使用者新連線反而造成無法連入資料庫進行管理工作，務必在第一行加入超級使用者(Superuser)權限，設定完成後重新載入 HBA 即可生效。

限制單一使用者同時連線人數

為了資源或安全性考量，有些情況希望指定使用者，同時一次僅能有 n 個連線至資料庫。

執行以下指令即可達成訴求：

```
postgres=# ALTER ROLE foo CONNECTION LIMIT 1;
ALTER ROLE
```

如果再有此帳號嘗試連接資料庫則會出現以下錯誤訊息：

```
FATAL: too many connections for role "foo"
```

我們可以將值設定為-1 來排除使用者連線限制。

此設定值可以為 0(上節已有介紹)或任意數字，但千萬不要設定參數 max_connections 為 0，那是無意義的，會造成限制所有使用者無法連線，但僅管設定為 0，超級使用者(superuser)仍可以連線資料庫。

如何做…

使用者連線限制只影響尚未建立連線的使用者，假使該使用者在執行限制線制前早已登入則不受限，除非該帳號使用者登出再重新登入。

就以下例子，雖然已限制使用者「foo」同時只能有一個連線，但透過查詢系統資料表 pg_stat_activity，卻有二個使用者「foo」，就因為在執行限制前，該帳號已經登入。

```
postgres=# SELECT rolconnlimit FROM pg_roles WHERE rolname = 'foo';
rolconnlimit
--------------
     1
postgres=# SELECT COUNT(*) FROM pg_stat_activity WHERE usename = 'foo';
   count
--------------
     2
```

中斷使用者

在有些情況下為了某些理由須將使用者中斷，例如資料庫效能或該使用者閒置過久。

PostgreSQL 提供一個函數可將使用者中斷 pg_terminate_backend()，只要輸入使用者該 session 的 pid 或 process ID 即可。

那如何一個使用者所產生的 pid 呢？查詢系統資料表 pg_stat_activity 即可找出，再結合剛所介紹的函數產生 SQL 語法如下：

```
postgres=# SELECT pg_terminate_backend(procpid) FROM pg_stat_activity
WHERE     usename = 'foo';
```

在執行以上 SQL 語法須注意以下二件事：

- ✧ 假設 WHERE 條件式沒有符合任何使用者 session，則不會得到任何 pid。
- ✧ 假設不小心將自己的 session 包含在其中會導致中斷自己 session。

故以下 SQL 語法會顯得更安全且實用：

```
postgres=# SELECT count(pg_terminate_backend(procpid))
                FROM pg_stat_activity
                WHERE usename NOT IN
                (SELECT usename
                 FROM pg_user
                 WHERE usesuper);
count
-------
    1
```

除 WHERE 條件使用 usename 以外，也可使用這些條件搭配：

- WHERE application_name = 'appanem'
- WHERE waiting;
- WHERE current_query = '<IDLE> in transaction';
- WHERE current_query = '<IDLE>'

如何做…

當執行函數 pg_terminate_backend()時會直接傳送信號至該 seesion 產生出來的程序程序。

但有個值得注意的地方，pid 是由作業系統所分配出來的，當我們中斷該 session 的 pid 時又同時有新 session 連接，而新 session 的 pid 又與原有 session 的 pid 相同，造成新 session 中斷，所以要特別小心。

建立多重實例

一般而言為了安全、資源控制或方便管理，我們常將不同的應用程式分門別類成不同群組的資料庫表，我不使用「資料庫」字眼，避免大家有所困惑。

如何做

假使想在同一主機上運行多個實體資料庫，主要有四個方式：

- ✧ 1. 單一主機上運行一個資料庫實例(Instance)並建立不同的資料綱要。
- ✧ 2. 單一主機上運行一個資料庫實例(Instance)並建立多個資料庫。
- ✧ 3. 單一主機上運行多個資料庫實例(Instance)並各別於實例上建立資料庫。
- ✧ 4. 不同主機上各自運行一個資料庫實例(Instance)，並各別於實例上建立資料庫。

第一種方式會於「多重資料綱要」章節中詳細介紹。

第二種方式會於中「」詳細介紹

第三種方式會於「單一主機運行多重資料庫伺服器」中詳細介紹

往往有人疑惑採哪種方式最好呢？其實是要根據不同的環境來取決於不方方式。

為了安全性的理由，第二種方式將資料表存放在不同的資料庫中，如此可避免擁有這些資料表權限的使用者，將它拿來結合(Join)查詢；但在有些情況中可能得跨資料表結合查詢時，則可採用第一種方式。

第三種方式有小點複雜，預設安裝只有一個資料庫實例於一個主機上，故採此方式須建立第二個資料庫實例並要注意監聽埠不可相沖。

第四種方式透過不同的實體或虛擬伺服器，嚴格來講並沒有很複雜，但虛擬化並非本書重點，故不詳談。

附帶一提使用圖型工具管理資料庫時，是透過樹狀結構瀏覽物件，假使物件只有 100 個或 1000 個並沒有任何感覺，但當物件超過 2000 個以上時，則可能會造成一些負擔喔！

多重資料綱要

將眾多資料表組成一個相關的名稱空間，就稱為資料綱要。

建立資料綱要 SQL 語法如下：

```
postgres=# CREATE SCHEMA finance;
postgres=# CREATE SCHEMA sales;
```

我們可以建立資料表於資料綱要上，SQL 語法如下：

CREATE TABLE finance.month(...);

一般使用者登入可使用 SQL 語法查詢現在的資料綱要，照理說現行資料綱要應與使用者預設資料綱要相同：

```
postgres=# SELECT current_schema;
current_schema
----------------
  public
```

當我們存取資料庫物件時，透過資料庫參數 search_path 來定義使用者預設是由何資料綱要下的物件開始尋找；以下為 SQL 語法：

```
postgres=# ALTER ROLE foo SET search_path='sales';
```

另外注意一點，使用者更改預設資料綱要後，不僅只是列出該資料綱要的資料表而已，當使用者建立新資料表也會置放於該資料綱要喔！

一般而言當建立資料表後，我們需要再授權使用者是否夠能存取該資料表的權限，但當建立大量資料表時是不是覺得很麻煩呢？此時我們可使用以下 SQL 語法來設定新建立的資料表預設存取權限：

```
postgres=# ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT ALL ON TABLES
TO foo;
```

如何做…

PostgreSQL 中資料庫參數 search_path 就像作業系統中的 PATH 環境變數一樣，而「現在資料綱要(current schema)」就如同現在的工作目錄，只是在資料庫語法中不是使用「cd」，而是使用 SQL 語法。

此點與其它資料庫軟體如 Oracle 是有所不同的，大多數情況許多人建立使用者後會使用同一資料綱要，但在 PostgreSQL 中是可將使用者所建立的資料表置於不同的資料綱要中。

使用者獨立資料綱要

將各使用者的資料分別存是身為一個管理者的目標，達到此目的不僅加強安全性也方便日後管理。

透過以下 SQL 語法，可針對使用者建立獨自擁有的資料庫：

```
postgres=# CREATE USER foo;
CREATE ROLE
postgres=# CREATE DATABASE foodb OWNER = foo;
CREATE DATABASE
postgres=#
```

使用「CREATE USER」與「CREATE ROLE」最大的差別在於「CREATE ROLE」建立使用者時並無授予「CONNECT」權限，故建立使用者後需再手動下 GRANT CONNECT to username;

一個使用者擁有「CONNECT」權限時預設是可以連接至任何資料庫的，所以必須執行「CONNECT」徹消權限，讓其它使用者無法連接至此資料庫，SQL 語法如下：

```
postgres=# REVOKE CONNECT ON DATABASE foodb FROM public;
```

僅管徹消權限，但該資料庫所有人「foo」還是可以連接資料庫「foodb」喔！

我們就嘗示建立新使用者「foo1」並試著連接看看吧！即顯示錯誤訊息：

```
postgres=# CREATE USER foo1
postgres=# \c foo foo1 ;
Password for user foo1:
嚴重錯誤:  資料庫 "foo" 權限被拒
DETAIL:  使用者沒有 CONNECT 權限。
Previous connection kept
```

單一主機運行多重資料庫實例

在某此情況上是有可能在單一主機上運行超過二個資料庫實例的，先確認你已經讀過「設計多重租戶」章節，且運行多重資料庫實例是你最好的選擇。

PostgreSQL 是允許我們在單一主機建立二個以上資料庫實例；安裝完 PostgreSQL 後預設會產生一資料目錄「PGDATA」並運行資料庫實例，如欲再新增資料庫實例只要執行 initdb 並在參數 D 後指定資料目錄所在路徑即可建立新資料庫實例，切記該資料目錄需擁有 postgres 帳號權限方可正常建立，以下則是建立新資料庫實例語法：

```
# mkdir -p /opt/PostgreSQL/9.0/data1
# postgres chown postgres.postgres /opt/PostgresSQL/9.0/data1
# initdb -D /opt/PostgreSQL/9.0/data1
```

建立新資料庫實例後記得編輯資料庫參數檔 postgresql.conf 中參數「port」，將監聽埠更改，避免新資料庫實例與舊資料庫實例沖突而造成錯誤，再啟透過以下指令啟動新資料庫實例即可：

```
# pg_ctl -D /opt/PostgreSQL/9.0/data1 start
```

設定共連接池

連接池(Connection Pool)是用來收集預先連接資料庫 Session 的地方，避免使用者連接、斷線再重新連接資料庫所造成的負擔而影響效能。

根據不同應用程式及軟體會提供許多不同的連接池，較好方式是透過資料庫伺服端所提供的連接池，因為使用資料庫伺服端所提供的連接池適合運作在不同類型的應用程式，而使用應用程式端的連接池只適用在單一應用程式或軟體。

此書將探討「PgBouncer」，它是一個輕量級的資料庫伺服端的連接池，可同時管理超過一台資料庫伺服器，首先決定好存放 PgBouncer 的參數檔、日誌檔及 pid 檔。

1. 建立 PgBouncer 設定檔：
2. 建立 user.txt 檔案，此檔必須存在，最少包含
3. 啟動 pgbouncher

```
# pgbouncher -d PgBouncer.ini
```

4. 測式連線

   psql -p 6543 -U postgres PgBouncer -c "reload"

切記 PgBouncer 參數 max_client_conn 的數量不能超過資料庫參數 max_connections，避免資料庫發生錯誤。

如何做

pgbouner 是一個很棒的軟體，當初設計時開發人員非常嚴僅確保該軟體操作簡單、輕量、穩定且執行快速；pgbouner 並非多執行緒，執行時很有效率，並不支援 SSL 連接，主要原因加密、解密會造成運作時負擔，如有安全性考量可使用「stunnel」。

pgbouner 支援連接池，參數如下：

pood_mode = transaction

psql -p 6543 PgBouncer -c "shutdown"

指令	用途
show stats	網路流量
show servers	列出資料庫伺服器
show clients	列出客戶端
show pools	列出連接池
show lists	資源總結
show users	列出 user.txt 檔中使用者
show databases	列出 PgBouncer.ini 檔中資料庫
show config	列出設定檔
show fds	列出檔案描述
show sockets	列出 Socket
show version	PgBouncer 版本

CHAPTER 5

資料表 & 資料

本章內容大綱如下：

- 命名物件規範
- 處理物件
- 找出並刪除重複值
- 產生測試資料
- 產生亂數資料
- 避免重複資料
- 匯入 EXCEL 檔
- 匯入文字檔

前言

- 匯入 EXCEL 檔

物件命名規則

PostgreSQL 建立物件時是可包含空白跟區分大小寫的，但必須在建立物件時包覆在雙引號裏；一般其它資料庫管理系統中物件是無法區分大小寫的，如 MySQL。

以下為建立一有區分大小寫的資料表：

```
postgres=# CREATE TABLE "NEW_st1"
            AS
           SELECT * FROM st1;
```

當輸入資料表名稱時以小寫表示會出現錯誤：

```
postgres=# SELECT COUNT(*) FROM new_st1;
ERROR:  relation "new_st1" does not exist
LINE 1: SELECT COUNT(*) FROM new_st1;
                             ^
```

當輸入資料表名稱時有區分大小寫但由於未以雙引號包覆故一樣出現錯誤：

```
postgres=# SELECT COUNT(*) FROM NEW_st1;
ERROR:  relation "new_st1" does not exist
LINE 1: SELECT COUNT(*) FROM NEW_st1;
                             ^
```

須以雙引號包覆資料表名稱且區分大小寫才正確：

```
postgres=# SELECT COUNT(*) FROM "NEW_st1";
 count
-------
   486
```

一般而言 PostgresSQL 當執行 SQL 語法時如未特別使用雙引號，皆會被認定為小寫：

以下三個不同 SQL 語法，代表的是同一個資料表：

```
SELECT * FROM new_st1;
```

```
SELECT * FROM NEW_ST1;
```

```
SELECT * FROM NEW_st1;
```

更多…

PostgreSQL 有個函數名為 quote_ident()，可以用來判斷我們所輸入的物件名稱即資料表，是否在輸入 SQL 語法是需要使用雙引號包覆，範例如下：

```
postgres=# select quote_ident('new_st1');
 quote_ident
-------------
 new_st1
```

```
postgres=# select quote_ident('NEW_st1');
 quote_ident
-------------
 "NEW_st1"
```

由以上的例子得知資料表「NEW_st1」在執行 SQL 語法時是需要使用雙引號包覆否則會出現錯誤喔！

強制資料表及欄位命名

找出並移除重複資料列

在實際環境有時候可能因為資料處理的問題而導致資料列重複，此時就得找出來重複的資料列並將它刪除，但在刪除之前可得確認這些重複列是可以被刪除的喔！

當找出重複資料列後，我們可選擇以下兩種方式：

- ✧ 刪除重複資料列
- ✧ 更新重複資料列資料為其它值

  本節以刪除重複資料列為範例，至於更新重複資料列為其它值基本上是相同的原理。

執行以下 SQL 語法，找出欄位 id 有重複的資料列：

```
postgres=# SELECT * FROM book
            WHERE id IN
           (SELECT id
            FROM book
            GROUP BY id
            HAVING COUNT(*) > 1);
id | name | age
----+------+-----
 1  | foo  |  20
 1  | foo  |  20
 1  | foo  |  20
```

接著再透過隱藏欄位　找出欄位 id 有重複的資料列：

```
postges=# SELECT ctid, * FROM  book
          WHERE ctid NOT IN
         (SELECT MAX(ctid)
          FROM book GROUP BY id);
ctid  | id | name | age
-------+----+------+-----
 (0,1) | 1  | foo  |  20
 (0,2) | 1  | foo  |  20
```

執行以下語法刪除重複資料列：

```
postgres=# DELETE FROM book
           WHERE ctid NOT IN
           (SELECT MAX(ctid)
           FROM book GROUP BY id);
DELETE 2
```

再次查詢該資料表可發現重複資料列已行刪除：

```
postgres=# SELECT * FROM book;
id | name | age | rownum
----+------+-----+--------
 2  | foo2 |  30 |      3
 3  | foo3 |  40 |      4
 1  | foo  |  20 |      5
```

基本上資料表發生有重複資料列情形可能多半是當初資料表設計時並未週全而導致，以上範例當我們去除重複資料列後可採取以下方式避免再次發生重複資料列：

✧ 建立主鍵(Primary Key)：

```
ALTER TABLE book ADD PRIMARY KEY(id);
```

✧ 建立關聯鍵限制 UNIQUE：

ALTER TABLE book ADD UNIQUE(id);

✧ 建立 UNIQUE 索引：

```
CREATE UNIQUE INDEX ON book(id);
```

找出唯一值

接續上節我們介紹過如何透過 SQL 語法找出資料表中重複資料列，其實藉由其它方式也可以快速找到重複資料列，相反的也能找到該欄位的唯一值。

查詢資料表 book，我們可看到有重複資料列：

```
postgres=# select * from book;
 id | name | age
----+------+-----
 1  | foo  |  20
 1  | foo  |  20
 2  | foo2 |  30
 3  | foo3 |  40
```

使用 PostgreSQL 優化器更新該資料表統計值：

```
postgres=# ANALYZE book;
```

執行以下 SQL 語法得到 n_distinct 欄位值為「-0.75」:

```
postgres=# SELECT attname, n_distinct
           FROM pg_stats
           WHERE schemaname = 'public'
           AND tablename = 'book';
attname | n_distinct
---------+------------
 id      |        -0.75
 name    |        -0.75
 age     |        -0.75
```

接著去除掉資料表 book 重複資料列：

```
postgres=# select * from book;
 id | name | age
----+------+-----
 1  | foo  |  20
 2  | foo2 |  30
 3  | foo3 |  40
```

再次使用 PostgreSQL 優化器更新該資料表統計值：

```
postgres=# ANALYZE book;
```

執行以下 SQL 語法得到 n_distinct 欄位值由「-0.75」變為「-1」:

```
postgres=# SELECT attname, n_distinct
           FROM pg_stats
           WHERE schemaname = 'public'
           AND tablename = 'book';
attname | n_distinct
---------+------------
 id      |        -1
 name    |        -1
 age     |        -1
```

由以上範例得知，當資料表中欄位去除重複資料後，由系統資料表 pg_stats 查詢欄位 n_distinct 會得到值為「-1」，然而得記住喔！欲用此方法判斷得在執行 SQL 語法前更新該資料表統計值，避免資料誤判！

產生測試資料

在有些時候為了配合應用程式或進行效能分析，DBA 需要產生一些測試資料來符合需求。

以下將依序介紹三種方式來產生隨機資料：

- ✧ 產生大量資料列
- ✧ 隨機產生欄位值
- ✧ 結果以上兩種隨機產生大量資料列

產生大量資料列

為了產生大量測試資料，可透過執行 PostgreSQL 內建函數來產生。

執行以下 SQL 語法產生數值序列：

語法： `generate_series(start, stop, step)`

start：初始值，stop：結束值，step：間隔值

以下範例用來產生 1 至 5 的數值序列：

```
postgres=# SELECT * FROM generate_series(1,5);
generate_series
-----------------
              1
              2
              3
              4
              5
```

```
postgres=# SELECT * FROM generate_series(1,5,2);
 generate_series
-----------------
              1
              3
              5
```

也可執行以下 SQL 產生日期：

```
postgres=# SELECT date(generate_series(now(),now()+'1 week', '1 day'));
    date
------------
 2011-10-02
 2011-10-03
 2011-10-04
 2011-10-05
 2011-10-06
 2011-10-07
 2011-10-08
 2011-10-09
(8 rows)
```

隨機產生欄位值

隨機整數(interger)：

```
postgres=# SELECT (random()*(2*10^9))::integer;
   int4
-----------
 308870600
```

隨機 bigint：

```
postgres=# SELECT (random()*(9*10^18))::bigint;
        int8
---------------------
 4748645930550992896
```

隨機 numberic data：

```
postgres=# SELECT (random()*100.)::numeric(4,2);
 numeric
---------
   37.31
```

由指定字串中產生隨機字串：

```
postgres=# SELECT repeat('12',(random()*40)::integer);
 repeat
--------
 121212
```

由 26 個字母產生隨機字串：

```
postgres=# SELECT substr('abcdefghijklmnopqrstuvwxyz',1,
(random()*26)::integer);
         substr
---------------------------
 abcdefghijklmnopqrstuvwxy
```

由列表中產生隨機字串：

```
postgres=# SELECT (ARRAY['foo','foo1','foo2'])[1+random()*3];
 array
-------
 foo2
```

隨機產生大量資料列

結合以上兩種方式隨機產生大量資料：

```
postgres=# SELECT generate_series(1,5) as key
,(random()*100.)::numeric(5,2)
,repeat('1',(random()*20)::integer);
key | numeric |     repeat
-----+---------+----------------
   1 |   81.52 | 11111
   2 |    4.00 | 111111
   3 |   37.54 | 11111
   4 |   73.34 | 11111111
   5 |    2.57 | 11111111111111
(5 rows)
```

結合以上兩種方式隨機產生大量資料並隨機進行排序：

```
postgres=#  SELECT generate_series(1,5) as key
,(random()*100.)::numeric(5,2)
,repeat('1',(random()*20)::integer)
ORDER BY random();
 key | numeric |       repeat
-----+---------+--------------------
   4 |   53.71 | 1111111111111
   1 |   98.50 | 11
   5 |   82.67 | 1111111111111111
   2 |    7.94 | 1111111111111111111
   3 |   75.87 | 1111111
(5 rows)
```

透過以上方式可以隨機產生亂數資料，在我們需要進行應用程式或效能分析時幫助我們測試。

隨機產生樣本資料

有些時候為了測試或開發需求，須要將線上資料庫中部份資料轉移至測試環境或開發環境，然而測試環境或開發環境往往設備效能不如正式機來的好，實際上也不需要線上全部資料，故本章節將教導大家如何由線上資料庫隨機撈取母體資料產生樣本。

首先由上節介紹中，產生 1000 筆資料列於資料表 st1：

```
postgres=# CREATE TABLE st1 AS SELECT * FROM generate_series(1,1000);
```

確認資料表 st1 是否存在筆數為 1000：

```
postgres=# SELECT count(*) FROM st1;
 count
-------
  1000
```

接著透過函數 random()隨機撈取樣本資料，random() < 0.5 表示會撈取所有資料中的 5%來產生樣本：

```
postgres=# SELECT COUNT(*) FROM st1 WHERE random() < 0.5;
count
-------
   481
postgres=# SELECT * FROM st1 WHERE random() < 0.5;
generate_series
-----------------
               1
               3
               5
               8
              10
              11
.....(略)
```

剛剛介紹過如何撈取樣本資料了，然而 PostgreSQL 中備份工具 pg_dump 並不支援備份時指定 WHERE 條件示加入函數 random()呀！

將資料表 st1 隨機產生的樣本匯出至文字檔 st1.dat：

```
postgres=# \COPY  (SELECT * FROM st1 WHERE random() < 0.5) to st1.dat;
```

備份資料表 st1 的 schema

```
$ pg_dump -t st1 -s  > st1.schema
```

至測試或開發環境將資料表 st1 的 schema 匯入：

```
$ psql -f st1.schema
```

再匯入文字檔 st1.dat 即可：

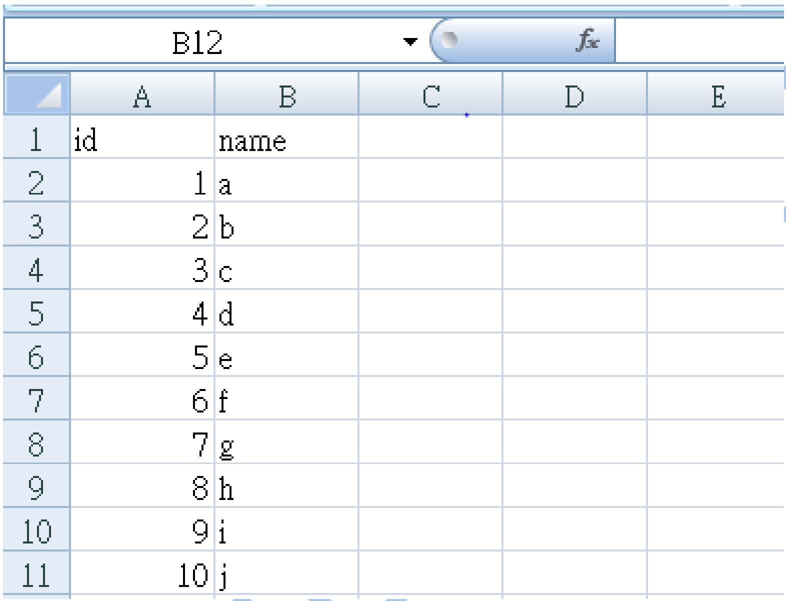
```
postgres=# \COPY st1 FROM st1.dat;
```

雖然備份工具 pg_dump 並不支援備份時指定 WHERE 條件示加入函數 random()，然而透過以上方式也可以成功產生樣本資料並匯入至試或開發環境中呢！

匯入電子試算表

電子試算表(Spreadsheets)可將資料儲存在每一個小格子中，然而除了存資料外也可撰寫公式，所以常用在商業用途上，最有名的電子試算表軟體無非就是我們耳熟能詳微軟所出的 EXECL，也因為這樣電子試算表常用來儲存各式各樣資料且用於報表，相對的 DBA 也經常會遇到需將這些資料匯入至資料庫中。

首先開啟任何一種電子試算表軟體，於工作表建立資料如下並以 CSV 格式儲存檔案，檔名為 book.csv。



	A	B	C	D	E
1	id	name			
2	1	a			
3	2	b			
4	3	c			
5	4	d			
6	5	e			
7	6	f			
8	7	g			
9	8	h			
10	9	i			
11	10	j			

接著，將檔案 book.csv 複製至資料庫伺服器中，進入資料庫建立一資料表名為 book，以 COPY 語法指定匯入格式為 CSV 將檔案中所有資料匯入至資料表中：

```
postgres=# CREATE TABLE  book (id varchar(10),name varchar(10));
CREATE TABLE
postgres=# \COPY book FROM book.csv CSV HEADER;
```

匯入完畢後，查詢資料表 book，的確檔案 book.csv 中的所有資料已被匯入至資料表中：

```
postgres=# SELECT * FROM book;
 id | name
----+------
 1  | a
 2  | b
 3  | c
 4  | d
 5  | e
 6  | f
 7  | g
 8  | h
 9  | i
 10 | j
(10 rows)
```

當執行 COPY 語法時並不會建立資料表，故在匯入之前須先建立資料表，語法最後如未 HEADER 則會將檔案 book.csv 中的標題「id, name」一併塞入至資料表中，故須注意，另外電子試算表的檔名也不一定要與資料表名稱相同喔！

CHAPTER 6

資料庫安全性

本章內容大綱如下：

- 命名物件規範
- 處理物件
- 找出並刪除重複值
- 產生測試資料
- 產生亂數資料
- 避免重複資料
- 從 Excel 匯入資料庫
- 匯入文字檔

前言

建立該使用者

本節將教導大家以兩種不同方式建立新使用者。

準備開始

為了可以建立新使用者，確認使用 Superuser 登入或該帳號具有 createrole 或 createuser 權限。

命令模式執行 createuser 指令，以互動模式建立新使用者：

```
$ createuser
Enter name of role to add: user1
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) y
Shall the new role be allowed to create more new roles? (y/n) n
```

指令 createuser 為 PostgreSQL 將建立新使用者 SQL 語法所撰寫出來的互動式程式，執行時會連接 postgres 資料庫並執行使用新使用者 SQL 語法。

執行 SQL 語法建立新使用者：

```
postgres=# CREATE ROLE user2 WITH NOSUPERUSER INHERIT NOCREATEROLE CREATEDB
LOGIN;
CREATE ROLE
postgres=# CREATE ROLE user3 WITH SUPERUSER;
CREATE ROLE
```

查看使用者

在建立完使用者後，執行 meta-command 查看使用者狀態：

```
postgres=# \du
                            List of roles
 Role name |                   Attributes                   | Member of
-----------+------------------------------------------------+-----------
 foo       |                                                | {}
 foo1      |                                                | {}
 postgres  | Superuser, Create role, Create DB, Replication | {}
 ricky     |                                                | {}
 user1     | Create DB                                      | {}
 user2     | Superuser, Cannot login, Replication           | {}
 user3     | Create DB                                      | {}
```

除了使用 CREATE ROLE 語法建立使用者，也可使用 CREATE USER 建立使用者，兩者最大差異當使用 CREATE ROLE 語法後面需接 LOGIN 否則所建立之使用者無法登入；而 CREATE USER 會隱含 LOGIN 故不需輸入。

授與使用者權限

當建立使用者後，首先第一次事就是授與該使用者存取資料表權限，儘管建立使用者後預設會授予綱要 public 權限，基於安全性理由，最好撤消此權限，並依照需求授予其它綱要或資料表存取權限。

準備開始

首先，確認使用 Superuser 登入或該帳號具有 GRANT 權限。

首先確認登入使用者擁有

假設已建立一使用者名為 user1，針對使用者 user1 授與綱要 foo 基本存取者權限：

```
GRANT USAGE ON SCHEMA foo TO user1;
```

授與使用者 user1 針對綱要 foo 下物件完全控制權限：

```
GRANT ALL ON SCHEMA foo TO user1;
```

注意！欲讓使用者 user1 擁有存取綱要 foo 下資料表權限，而並非完全控制權限，務必先受予 USAGE 權限，如此步驟省略，就算後面有授與使用者 user1 存取權限如 SELECT、INSERT、UPDATE、DELETE，該使用者 user1 仍無法存取該資料表。

授與使用者 user1 針對綱要 foo 底下資料表 t1 擁有 SELECT、IINSERT、UPDATE; DELETE 權限：

```
GRANT SELECT, INSERT, UPDATE, DELETE ON foo.t1 TO user1;
```

如欲針對綱要 foo 底下資料表分別授與不同存取權限，只能執行如上 SQL 語法，並無法一次指定全部資料表如 foo.*之語法，只能分別指定各別資料表直到 PostgreSQL 9.0 版本釋出。

PostgreSQL 9.0 版本後如欲一次指定綱要 foo 下所有資料表存取權限如以下 SQL 語法授與使用者 user1 針對綱要 foo 下所有資料表擁有 SELECT 權限：

```
GRANT SELECT ON ALL TABLES IN SCHEMA foo to user1;
```

撤消使用者權限

繼上節介紹過授與使用者權限後，接著當然就是如何撤消使用者權限。

準備開始

首先，確認使用 Superuser 登入或該帳號具有 GRANT 權限；另外我們是無法撤消 Supseruser 使用者任何權限喔！

如何做…

如欲撤消使用者 user1 存取資料表 t1 完全控制權限，執行以下 SQL 語法：

```
REVOKE ALL ON t1 FROM user1;
```

另外注意一點，當建立資料表時未指定綱要，預設建立在綱要 Public 底下，然而使用者預設擁有綱要 public 底下資料表存取權限如 SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES 及 TRIGGER，基於安全性理由，應該撤消使用者擁有存取綱要 Public 下權限，或撤消該資料表在綱要 public 下權限：

```
REVOKE ALL ON t1 FROM PUBLIC;
```

暫時禁止使用者登入

在某些情況中 DBA 並非要刪除某使用者而只是暫時性的該帳號禁止登入。

準備開始

在修改該使用者前，須確認已使用 Superuser 登入或該使用者具備 CREATEROLE 權限。

執行以下 SQL 語法禁止使用者 foo 登入，當禁止後如欲使用 foo 帳號登入會出現權限不足：

```
postgres=# ALTER USER foo nologin;
ALTER ROLE
postgres=# \c store foo
FATAL:  role "foo" is not permitted to log in
Previous connection kept
```

執行以下 SQL 語法允許使用者 foo 登入：

```
postgres=# ALTER USER foo login;
ALTER ROLE
```

限制使用者同時連接數量：

使用者擁有登入資料庫權限後，可搭配 Connection Limit 來限制該使用者同帳號時允許登入的數量。

限制同一使用者只允許 0 個至連接資料庫，也就是禁止登入意思：

```
postgres=# ALTER USER foo limit 0;
ALTER ROLE
```

限制同一使用者允許 10 個至連接資料庫：

```
postgres=# ALTER USER foo limit 10;
ALTER ROLE
```

限制同一使用者允許 -1 個至連接資料庫，代表不限制數量：

```
postgres=# ALTER USER foo limit -1;
ALTER ROLE
```

強制 NOLOGIN 使用者登出：

當我們修改使用者為 NOLOGIN 禁止登入後，然而此動作只生效於未登入的使用者，至於已登入的使用者呢？執行以下 SQL 語法將使用者狀態為 NOLOGIN 強制刪除：

```
postgres=# SELECT pg_terminate_backend(procpid)
FROM  pg_stat_activity a
JOIN pg_roles r ON a.usename = r.rolname AND not rolcanlogin;
```

在舊 PostgreSQL 版本中可能沒有函數 pg_terminate_backend，只能以 kill 指令強制刪除使用者：

```
$ psql -t -c "SELECT 'kill '  || procpid FROM pg_stat_activity a join pg_roles r on a.usename = r.rolname and not rolcanlogin;"
kill 4187
```

當執行 psql 時參數-t 表示只輸出資料列，參數-c 後面接欲執行的 SQL 語法，至於 SQL 語法內容代表使用者為 NOLOGIN 且處於連線狀態，再串接 kill 指令，所產生的腳本，最後執行輸出指行即可。

僅刪除使用者但保留其資料

在某些情況中欲刪除某一使用者帳號，而該使用者帳號擁有資料庫物件如資料表，刪在刪除時會出現以下錯誤：

```
store=# DROP USER foo;
ERROR:  role "foo" cannot be dropped because some objects depend on it
DETAIL:  1 object in database store
```

```
store=# ALTER USER foo nologin;
ALTER ROLE
```

建立新使用者帳號「foo1」

```
store=# CREATE USER foo1 password 'foo1';
CREATE ROLE
```

將原本舊使用者帳號 「foo」權限授與至「foo1」：

```
store=# GRANT foo to foo1;
GRANT ROLE
```

將原本舊使用者帳號「foo」所擁有之物件重新指派至「foo1」：

```
store=# REASSIGN OWNED BY foo to foo1;
REASSIGN OWNED
```

再執行一次刪除帳號指令，即可刪除：

```
store=# DROP USER foo;
DROP ROLE
```

其實在實際上有更快速且方便的做法，只要執行以下語法將使用者帳號重新命名為新名稱即可，由於本書主要教導大家運作原理，故一個一個步驟帶領大家實作：

```
store=# ALTER USER foo RENAME to foo1;
ALTER ROLE
```

密碼安全性強度

PostgreSQL 是允許使用者設定密碼時以明碼方式儲存，基於安全性理由應設定 pg_hba 設定檔以 MD5 取代明碼方式傳送密碼。

首先，建立使用者 user2 並故意以明碼方式儲存密碼，實際環境中不建議這樣做：

```
postgres=# CREATE USER user2 UNENCRYPTED password 'user2';
CREATE ROLE
```

接者，再查詢系統資料表 pg_shadow 得知剛所建立使用者 user2 以明碼方式儲存密碼：

```
postgres=# SELECt usename, passwd from pg_shadow;
 usename  |              passwd
----------+-------------------------------------
 postgres |
 foo      | md5fdba98970961edb29f88241b9d99d890
 user2    | user2
 (3 rows)
```

基於上述狀況，我們不難保證現實狀況中會有使用者以此方式儲存密碼，故執行以下 SQL 語法找出系統資料表 pg_shadow 中密碼欄位是否並非以 MD5 方式儲存，如未以 MD5 方式儲存則該字串不會以 md5 開頭且字串長度不等於 35：

```
postgres=# SELECT usename, passwd FROM pg_shadow WHERE passwd not like
'%md5%' or length(passwd) <> 35;
user2    | user2
 (1 rows)
```

稽核 DDL、DCL 動作

本節將教導大家如何稽核 DDL 動作，將任何關於 DDL 動作寫入至資料庫伺服器日誌檔。

準備開始

編輯資料庫伺服器設定檔 postgresql.conf 設定參數 log_statement：

log_statement= 'ddl'

設定完畢後無須重啟資料庫伺服器，重新載入即可生效：

```
$ pg_ctl -D $PGDATA reload
```

另外，如果只是針對使用者該次登入階段進行暫時稽核 DDL 動作，可改執行以下 SQL 語法：

```
postgres=# SET log_statement=none;
```

如何做…

開啟稽核 DDL 動作後，任何 SQL 語法有關 CREATE、ALTER、DROP 將寫入至資料庫伺服器日誌檔。

由於資料庫伺服器日誌檔中包含很多訊息，可以執行以下指令過濾出 DDL 動作：

```
$ egrep -i "create|alter|drop|CREATE|ALTER|DROP" postgresql.log > ddl.log
```

基本上資料庫伺服器日誌檔輪轉(Roataion)預設值如下，不見得適合每個環境，如日誌檔檔名、日誌檔輪轉保留日期天數、及日誌檔大小，因會與公司稽核策略有關，建議適需求更改。

```
log_filename = postgresql-%Y-%m-%d_%H%M%S.log
log_rotation_age = 1d
log_rotation_size = 10MB
```

當開始稽核 DDL 動作所有 DDL 動作皆會記錄至資料庫伺服器日誌檔中，然而我們並不知道該 DDL 動作是由何使用者、用戶端及何時動作，有些情況單純的記錄不符我們需求，可透過參數「log_line_prefix」來設定；%t 代表 timestamp，另有%u 代表使用者、%p 代表 PID 及%d 代表資料庫名稱，故可用「%t[%p]%u@%d」來表示，以下為資料庫伺服器日誌檔預設格式與修正後的格式，藉由參數修改後讓我們可以獲得更詳細資訊：

```
LOG:  statement: create table tb1 (a varchar(10));
```

```
2011-09-30 12:43:49 CST[29570]postgres@postgresDB:  statement: create table tb2 (a varchar(10));
```

另外值得注意一點，當開始稽核 DDL 動作後所有 DDL 動作皆會記錄至資料庫伺服器日誌檔中，然而 PostgreSQL 中 DDL 動作是可回滾(Rollback)，如下 SQL 語法，只會將 DDL 動作記錄至資料庫伺服器日誌檔中，ROLLBACK 指令並不會記錄，也就是說實際上該 DDL 動作可能沒執行喔！

```
BEGIN;
ALTER TABLE tb1 ADD COLUMN b varchar(10);
ROLLBACK;
```

假設如未開啟稽核 DDL 動作，也可以透過其它方式獲得資訊，於 Postgresql 中每個資料表都有個內建隱藏欄位 xmin，當資料表有任何 DML 或 xid(Transaction id)，也就是說只要資料列有任何異動，欄位 XMIN 的值就會加 1，如以下範例：

1. 查詢資料表 tb1 所有欄位及隱藏欄位 xmin：

```
postgres=# SELECT xmin, * FROM tb1;
xmin  |  a
------+------
 2543 | aaa
```

2. 當塞入一筆記錄後再次查詢資料表 tb1 所有欄位及隱藏欄位 xmin，我們會發現欄位 xmin 中的值會加 1：

```
postgres=# INSERT INTO tb1 VALUSE ('bbb');
postgres=# SELECT xmin, * FROM tb1;
xmin  |  a
------+-----
 2543 | aaa
 2544 | bbb
```

3. 此時資料表 tb1 增加一欄位 b，接著再次塞入第二筆記錄，再次查詢資料表 tb1 所有欄位及隱藏欄位 xmin，我們發現在第三筆記錄中，欄位 xmin 值為 2546 跳過 2545，主要原因是在塞入第二筆記錄之前，執行 DDL 動作，故 2545 為記錄 DDL 動作：

```
postgres=# ALTER TABLE tb1 ADD COLUMN b varchar(10);
postgres=# INSERT INTO tb1 VALUSE ('ccc');
postgres=# SELECT xmin, * FROM tb1;
xmin  |  a  | b
------+-----+---
 2543 | aaa |
 2544 | bbb |
 2546 | ccc |
```

剛介紹完如何稽核 DDL 動作，那當資料變動如何稽核呢？一樣設定 log_statement 即可：

```
log_statement= 'mod'
log_statement= 'all'
```

「all」表示「ddl」加上「mod」，當設定值為「mod」時，除上任何 DDL 動作會稽核外，另外 INSERT、UPDATE、DELETE、TRUNCATE、COPY FROM 及 EXPLAIN ANALYZE 也會稽核，基本上「all」、「mod」 設定與操作方式與「ddl」故就不多做介紹囉！

資料庫伺服器整合 LDAP

本節將教導大家如何讓 PostgreSQL 資料庫伺服器整合 LDAP 進行帳號證証。

準備開始

首先需確認資料庫伺服器中已建立帳號且存在於 LDAP 伺服器中。

如何做…

接著，編輯資料庫伺服器認證檔 pg_hba.conf，定義認證方式及認證伺服器位址範圍：

```
host all all 10.10.0.1/16 ldap ldapserver=ldap.example.net ldapprefix="cn="
ldapsuffix=", dc=example, dc=net"
```

如何運作…

當設定完成後，PostgreSQL 使用者進行認證時將改由確認 LDAP 伺服器中同一帳號使用者密碼；使用者權限方面仍由資料庫伺服器本身控制如 GRANT 及 REVOKE 指令。

更多…

設定用戶端使用 LDAP

編輯資料庫伺服器連接服務設定檔 pg_service.conf，定義連接存取參數如下：

```
ldap://ldap.mycompany.com/dc=mycompany,dc=com?uniqueMember?one?(cn=myda
tabase)
```

如欲更多資訊可參閱以下網址：

關於伺服器設定：

http://www.postgresql.org/docs/9.1/interactive/auth-methods.html#AUTH-LDAP

關於用戶端設定：

http://www.postgresql.org/docs/9.0/static/libpq-ldap.html

資料庫伺服器連接服務設定檔 pg_service.conf，定義 PostgreSQL 用戶端程式 libpq 連接時所需要的參數設定值，此檔可位於 PostgreSQL 設定檔目錄中或 $PGDATA，也可位於使用者家目錄 ~/.pg_service.conf，如欲自己定義路徑位置可透過環境變數 PGSYSCONFDIR 定義資料庫伺服器連接服務設定檔位置，再由 PGSERVICE 指定預使用哪個 SERVICE。

使用 SSL 機制連接

本節教導大家如何使用 SSL 機制連接資料庫伺服器，讓資料傳送時進行加密，如此網路封包監聽如 Sniffer 就算截取到網路封包關於使用者名稱、密碼或敏感性資料，也難以解讀，此應用大部份會用於遠端使用者透過 VPN 連接至公司網路中存取資料庫伺服器。

由於使用 SSL 機制會在連接資料庫伺服器進行資料加密，使傳送資料時較慢，且 SSL 機制會消耗 CPU 資源，故如有大量使用者連接至資料庫伺服器將導致效能大量衝擊，此時可考慮使用 PgBouncer 它是一個資料庫伺服端的連接池中介軟體，位於用戶端與資料庫伺服器間，來改善效能。

準備開始

取得或產生 SSL 金鑰及憑證申請檔：server.key 及 server.crt 並將此兩個檔案複製至$PGDATA 裏。

至於如何產生 SSL 金鑰及憑證申請檔可參考以下執指：

```
$ openssl genrsa 1024 > server.key
$ openssl req -new -x509 -key server.key -out server.crt
```

編輯資料庫伺服器設定檔，定義參數 ssl = on ，重新啟動資料庫伺服器使其生效。

如何運作…

PotgreSQL 開啟 SSL 機制後，SSL 監聽埠仍與原 PostgreSQL 服務監聽埠共用，預設埠號為 5432，至於是否 SSL 加密則取決於何種方式連接。

更多…

接著，編輯資料庫伺服器認証檔設定用戶端是否需使用加密才能連接至資料庫伺服器：

```
host database user IP-address IP-mask auth-method
host all all 192.168.10.1/24 md5
hostssl all all 0.0.0.0/0 md5
```

根據以下設定檔代表，192.168.10.x 網段將使用未加密方式連接，其它以外必須使用 SSL 才能連接資料庫伺服器。

設定用戶端使用 SSL 機制

在設定資料庫伺服器端後接著就是讓用戶端使用 SSL 連接，需藉由環境變數 PGSSLMODE，下表列出關於此 PGSSLMODE 可用模式及代表意義：

disable	NO	NO	不開啟加密，不願加密機制影響效能。
allow	Maybe	NO	不開啟加密，但伺服器堅持使用加密則啟用。
prefer	Mabye	NO	不開啟加密，但只要伺服器支援加密時則啟用。
require	YES	NO	開啟加密，且需位在信任的網段中。
verify-ca	YES	取決 CA 策略	開啟加密，且需確保為 CA 所信任的伺服器中，
verif-full	YES	YES	開啟加密，且需確保為指定的信任伺服器中。

P.S 中間人攻擊(Man-in-the-middle)指的是在網路傳輸過程中，協助傳送封包的網路元件。有心竊聽的駭客，透過網路協定容易相信他人的特點，將自己偽裝成協助傳訊的網路元件，實則把網路上的流量通通紀錄下來
欲了解更多資訊可參閱以下網址：
http://www.postgresql.org/docs/9.0/interactive/libpq-ssl.html

加密敏感資料

由於透過 Binary 安裝 PostgreSQL 並無提供只功能，故得下載原始碼安裝檔進行編譯才行，瀏覽以下網址進行下載 http://www.postgresql.org/ftp/source/v9.1.1/

- 編譯及安裝 pgcrypto
  使用 postgres 帳號編譯再搭配 su 指令使用 root 帳號進行安裝

```
$ tar zxvf postgresql-9.1.1.tar.gz
$ cd postgresql-9.1.1/contrib/pgcrypto/
$ make
$ gcc -O2 -Wall -Wmissing-prototypes -Wpointer-arith
-Wdeclaration-after-statement -Wendif-labels -Wformat-security
-fno-strict-aliasing -fwrapv -fpic -I. -I. -I../../src/include
-D_GNU_SOURCE   -c -o md5.o md5.c
gcc -O2 -Wall -Wmissing-prototypes -Wpointer-arith
-Wdeclaration-after-statement -Wendif-labels -Wformat-security
-fno-strict-aliasing -fwrapv -fpic -I. -I. -I../../src/include
-D_GNU_SOURCE   -c -o sha1.o sha1.c
gcc -O2 -Wall -Wmissing-prototypes -Wpointer-arith
-Wdeclaration-after-statement -Wendif-labels -Wformat-security
-fno-strict-aliasing -fwrapv -fpic -I. -I. -I../../src/include
-D_GNU_SOURCE   -c -o sha2.o sha2.c
gcc -O2 -Wall -Wmissing-prototypes -Wpointer-arith
-Wdeclaration-after-statement -Wendif-labels -Wformat-security
-fno-strict-aliasing -fwrapv -fpic -I. -I. -I../../src/include
-D_GNU_SOURCE   -c -o internal.o internal.c
gcc -O2 -Wall -Wmissing-prototypes -Wpointer-arith
-Wdeclaration-after-statement -Wendif-labels -Wformat-security
-fno-strict-aliasing -fwrapv -fpic -I. -I. -I../../src/include
-D_GNU_SOURCE   -c -o internal-sha2.o internal-sha2.c
gcc -O2 -Wall -Wmissing-prototypes -Wpointer-arith
-Wdeclaration-after-statement -Wendif-labels -Wformat-security
-fno-strict-aliasing -fwrapv -fpic -I. -I. -I../../src/include
-D_GNU_SOURCE   -c -o blf.o blf.c
gcc -O2 -Wall -Wmissing-prototypes -Wpointer-arith
-Wdeclaration-after-statement -Wendif-labels -Wformat-security
```

```
-fno-strict-aliasing -fwrapv -fpic -I. -I. -I../../src/include
-D_GNU_SOURCE   -c -o rijndael.o rijndael.c
gcc -O2 -Wall -Wmissing-prototypes -Wpointer-arith
-Wdeclaration-after-statement -Wendif-labels -Wformat-security
-fno-strict-aliasing -fwrapv -fpic -I. -I. -I../../src/include
-D_GNU_SOURCE   -c -o fortuna.o fortuna.c
gcc -O2 -Wall -Wmissing-prototypes -Wpointer-arith
-Wdeclaration-after-statement -Wendif-labels -Wformat-security
-fno-strict-aliasing -fwrapv -fpic -I. -I. -I../../src/include
-D_GNU_SOURCE   -c -o random.o random.c
gcc -O2 -Wall -Wmissing-prototypes -Wpointer-arith
-Wdeclaration-after-statement -Wendif-labels -Wformat-security
-fno-strict-aliasing -fwrapv -fpic -I. -I. -I../../src/include
-D_GNU_SOURCE   -c -o pgp-mpi-internal.o pgp-mpi-internal.c
gcc -O2 -Wall -Wmissing-prototypes -Wpointer-arith
-Wdeclaration-after-statement -Wendif-labels -Wformat-security
-fno-strict-aliasing -fwrapv -fpic -I. -I. -I../../src/include
-D_GNU_SOURCE   -c -o imath.o imath.c
gcc -O2 -Wall -Wmissing-prototypes -Wpointer-arith
-Wdeclaration-after-statement -Wendif-labels -Wformat-security
-fno-strict-aliasing -fwrapv -fpic -shared -o pgcrypto.so pgcrypto.o px.o
px-hmac.o px-crypt.o crypt-gensalt.o crypt-blowfish.o crypt-des.o
crypt-md5.o md5.o sha1.o sha2.o internal.o internal-sha2.o blf.o rijndael.o
fortuna.o random.o pgp-mpi-internal.o imath.o mbuf.o pgp.o pgp-armor.o
pgp-cfb.o pgp-compress.o pgp-decrypt.o pgp-encrypt.o pgp-info.o pgp-mpi.o
pgp-pubdec.o pgp-pubenc.o pgp-pubkey.o pgp-s2k.o pgp-pgsql.o
-L../../src/port
-Wl,-rpath,'/opt/PostgreSQL/9.1/lib',--enable-new-dtags   -lz
$ su -c "make install"
Password:
/bin/mkdir -p '/opt/PostgreSQL/9.1/lib/postgresql'
/bin/mkdir -p '/opt/PostgreSQL/9.1/share/postgresql/extension'
/bin/sh ../../config/install-sh -c -m 755  pgcrypto.so
'/opt/PostgreSQL/9.1/lib/postgresql/pgcrypto.so'
/bin/sh ../../config/install-sh -c -m 644 ./pgcrypto.control
'/opt/PostgreSQL/9.1/share/postgresql/extension/'
```

```
/bin/sh ../../config/install-sh -c -m
644 ./pgcrypto--1.0.sql ./pgcrypto--unpackaged--1.0.sql
'/opt/PostgreSQL/9.1/share/postgresql/extension/'
```

➢ 建立 pgcrypt 函數

當編譯完成後會產生檔案「pgcrypto--1.0.sql」，分別為 pgcrypto 目錄下或是安裝到/opt/PostgreSQL/9.1/bin/share/postgresql/extension/目錄下，此兩個檔案皆為同一檔案，只有權限不同而已，由於需將「MODULE_PATHNAME」更改，才能順利進行函數建立，故只要修改其權限即可；接著指定欲安裝函數的資料庫名稱進行建立，須注意必須一個一個資料庫進行安裝喔，也就是說有進行安裝的資料庫才擁有這個功能。

```
$ cd /opt/PostgreSQL/9.1/bin/share/postgresql/extension/
$ sed -i 's/MODULE_PATHNAME/\$libdir\/pgcrypto/g' pgcrypto--1.0.sql
$ psql -f pgcrypto--1.0.sql postgres
```

➢ 確認安裝

進行 postgresql 命令模式執行 Meta-Command 確認是否順利安裝：

```
postgres=#\df
public | armor          | text  | bytea                    | normal
public | crypt          | text  | text, text               | normal
public | dearmor        | bytea | text                     | normal
public | decrypt        | bytea | bytea, bytea, text       | normal
public | decrypt_iv     | bytea | bytea, bytea, bytea, text | normal
public | digest         | bytea | bytea, text              | normal
public | digest         | bytea | text, text               | normal
public | encrypt        | bytea | bytea, bytea, text       | normal
public | encrypt_iv     | bytea | bytea, bytea, bytea, text | normal
```

關於更多可參閱以下網址：

http://www.postgresql.org/docs/9.1/interactive/pgcrypto.html

CHAPTER 7

資料庫管理

本章內容大綱如下：

- 命名物件規範
- 處理物件
- 找出並刪除重複值
- 產生測試資料
- 產生亂數資料
- 避免重複資料
- 從 Excel 匯入資料庫
- 匯入文字檔

前言

交易控制階段

身為資料庫管理員(DBA)得掌握交易資料完整性，避免使用者更新資料時因其中某一階段中斷而造成資料不一致導致重大傷害，如金流系統；而以PostgreSQL 處理交易機制中，允許撰寫腳本來確保全部交易階段中處於全部執行成功或全部失敗來確保資料完整性。

如何做…

為了確保交易階段中 SQL 指令成功執行，可於該階段內加入交易控制指令：

```
BEGIN;
   COMMAND 1;
   COMMAND 2;
COMMIT;
```

我們也可以使用 psql 來達到此目的，只需參數以下參數即可：

```
# psql -1 -f scrip.sql
```

或

```
# psql --single-transaction -f script.sql
```

如何做…

交易控制指令無法以巢狀迴圈執行，如以下 SQL 語法將可能導致其中腳本發生錯誤。

```
postgres=# BEGIN;
BEGIN
postgres=# BEGIN;
WARNING:  there is already a transaction in progress
BEGIN
postgres=# COMMIT;
COMMIT
postgres=# COMMIT;
WARNING:  there is no transaction in progress
COMMIT
```

執行 SQL 腳本注意事項

當透過 psql 執行 SQL 腳本時遇到錯誤預設會繼續執行後續執令，似乎有點不恰當，但 PostgreSQL 採取此方式已經歷過好幾個版本，故本章將教導大家如何更改此預設動作。

準備開始

首先，撰寫一腳本並故意寫錯其中指令：

```
# vi script.sql
mistake1;
mistake2;
mistake3;
```

接著，執行剛所建立腳本，預設會全部執行腳本中所有指令，就算中間因錯誤指令遭遇到錯誤如下：

```
# psql -f script.sql
psql:script.sql:1: ERROR:  syntax error at or near "mistake1"
LINE 1: mistake1;
        ^
psql:script.sql:2: ERROR:  syntax error at or near "mistake2"
LINE 1: mistake2;
        ^
psql:script.sql:3: ERROR:  syntax error at or near "mistake3"
LINE 1: mistake3;
```

我們可透過增加 psql 參數，一旦指令出錯將馬上中斷並跳出：

```
# psql -f script.sql -v ON_ERROR_STOP=on
psql:script.sql:1: ERROR:  syntax error at or near "mistake1"
LINE 1: mistake1;
        ^
```

如不透過 psql 指令參數也可直接於腳本中增加此指令：

```
# vi script1.sql
\set ON_ERROR_STOP
mistake1;
mistake2;
mistake3;
```

如何運作…

ON_ERROR_STOP 為 psql 參數可控制當載入 SQL 腳本時遇錯將回傳給作業系統錯誤碼，故會立即中止執行並跳出。

另外，我們也可將 ON_ERROR_STOP 寫至 postgresql 個人 profile 中，如此就不用在執行 psql 時指定參數或於腳本中加入命令：

```
# vi ~/.psqlrc
\set ON_ERROR_STOP
```

在指定 postgresql 個人 profile 後，如欲暫時不套用 profile 中的設定值，可於 psql 指定-X 參數即可。

增加/刪除資料表欄位

於例行性維護中常會因需求增加或刪除資料表欄位，但執行此動作可得非常小心，避免因修改資料表欄位造成長期間全表鎖定而影響運作。

如何做…

可透過以下 SQL 語法增加資料表欄位：

```
ALTER TABLE table_name
ADD COLUMN last_update_timestamp TIMESTAMP WITHOUT TIME ZONE;
```

或透過以下 SQL 語法刪除資料表欄位：

```
ALTER TABLE table_name
DROP COLUMN last_update_timestamp;
```

我們也可使用 EXISTS 指令判斷當該欄位存在時，先執行刪除再重新增加該指定欄位：

```
ALTER TABLE table_name
DROP COLUMN IF EXISTS last_update_timestamp,
ADD COLUMN last_update_timestamp TIMESTAMP WITHOUT TIME ZONE;
```

如何運作…

使用 ALTER TABLE 指令增加或刪除欄位時，將造成全資料表鎖定(AccessExclusiveLock 層級)，而造存無法進行其它動作，故我們都希望增加或刪除欄位時所需時間愈短愈好。

使用 DROP CPLUMN 刪除欄位實際上並非將資料移除，只是標記該欄位已被刪除，這也是為什麼刪除欄位時速度即快的原因。

至於 ADD COLUMN 增加欄位，如增加的欄位值預設為空值(NULL)，速度也非常快速，但相反的如果該欄位並非空值(NOT NULL)，由於需重新寫一筆資料列故會相當緩慢。

剛我們有提到當刪除欄位時實際上並非將資料移除，直到後續有異動如 Insert 或 Update 時該資料列時才會更新為空值(NULL)至該欄位，也就是說雖然有執行刪除欄位動作，但資料庫本身空間並不會得到釋放，會於之後進行 DML 動作時才會釋放該資料列。

然而也可透過另一種方式釋放這些已刪除空間就是執行 VACUUM FULL 指令，但由於此動作將造成全資料表鎖定故使用時得非常小心，當然依筆者想法是能夠不刪除欄位就不要刪除，尤其當資料表非常巨大時，避免遇到一些不必的麻煩。

更多…

一般而言當刪除欄位時該欄位索引也一併會刪除，但如有其它物件參考到該欄位則在刪除該欄位時會發生錯誤，此時可使用 CACADE 指令強制刪除全部相關物件：

```
ALTER TABLE table_name
DROP COLUMN last_update_timestamp
CASCADE;
```

修改資料表欄位類型

修改欄位類型並非很常遇到，但一旦需要執行時則需要了解所有環節避免執行不當造成錯誤。

準備開始

```
postgres=# SELECT * FROM birthday;
```

```
CREATE TABLE birthday(name  text,dob int);
```

增加/刪除資料綱要

如何做…

執行以下 SQL 語法新增綱要：

```
postgres=# CREATE SCHEMA foo;
postgres=# \dn
  List of schemas
  Name  |  Owner
--------+----------
 foo    | postgres
 public | postgres
```

由上已建立資料綱要 foo，但擁有者預設為建立綱要該使用者，故如欲指定綱要擁有者，則改執行以下 SQL 語法：

```
postgres=# CREATE SCHEMA foo AUTHORIZATION foo;
CREATE SCHEMA
postgres=# \dn
  List of schemas
  Name  |  Owner
--------+----------
 foo    | foo
 public | postgres
```

另外，如欲由現有使用者，建立與該使用者同名稱綱要，且該使用者也是綱要擁有者可執行以下 SQL 語法：

```
postgres=# CREATE USER foo1 password 'foo1';
CREATE ROLE
postgres=# CREATE SCHEMA AUTHORIZATION foo1;
CREATE SCHEMA
```

一般而言，資料庫使用者都擁有自己所屬的綱要，如 Oracle 資料庫，而在 PostgreSQL 中是允許單一綱要被其它使用者所擁有或共享，可別搞混喔！

剛介紹完如何建立綱要，接著當然是如何刪除綱要，SQL 語法如下：

```
postgres=# DROP SCHEMA foo;
```

當欲刪除綱要下仍有物件存在時，執行以上指令會發生以下錯誤無法刪除：

```
ERROR:  cannot drop schema foo because other objects depend on it
DETAIL:  table foo.t1 depends on schema foo
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

可改執行以下 SQL 語法強制刪除相關聯物件：

```
postgres=# DROP SCHEMA IF EXISTS foo CASCADE;
```

更多…

綱要存取權限：

當欲授予使用者存取綱要下物件權限時，可執行以下 SQL 語法：

```
postgres=# GRANT SELECT ON ALL TABLES IN SCHEMA foo TO foo;
GRANT
```

根據以上所提供 SQL 語法，僅授予已存在綱要下的資料表存取權限，而當新建立資料表時又得重新授權一次，故可改採用以下 SQL 語法，讓綱要下資料表預設授予存取權限：

```
postgres=# ALTER DEFAULT PRIVILEGES IN SCHEMA foo
            GRANT SELECT ON TABLES TO foo;
ALTER DEFAULT PRIVILEGES
```

搬移物件至不同綱要

身為 DBA 有時候基於管理目的需要求某綱要下的物件如資料表移至另一綱要下。

如何做…

將指定 foo 資料表由現在 foo 綱要下搬移至 public 綱要：

```
postges=# ALTER TABLE foo SET SCHEMA public;
```

然而如這搬移 foo 綱要下所有物件，則不如將 foo 綱要重新命名，但新綱要名稱不能存在於現有綱要中：

```
postgres=# ALTER SCHEMA foo RENAME to newfoo;
```

除了資料表可搬移至不同綱要外，如視觀表、函數(Function)…等也可以透過以上方式搬移喔！

如何運作…

當我們將某一資料表搬移至另一綱要時，該資料表所屬索引、觸發及規則也會一併自動搬移至新綱要下，「綱要」主要為方便 DBA 管理，雖然將物件搬移至不同綱要，但事實上實體並不會改變。

增加/刪除表格空間

搬移物件至不同表格空間

某些情況中由於環境中不止一台資料庫伺服器，需讓 PostgreSQL 資料庫伺服器間能夠彼此存取，則須另外安裝額外模組，名為「dblink」，由於透過 Binary 安裝並無提供只功能，故得下載原始碼安裝檔進行編譯才行，瀏覽以下網址進行下載 http://www.postgresql.org/ftp/source/v9.1.1/

- 編譯及安裝 dblink

  使用 postgres 帳號編譯再搭配 su 指令使用 root 帳號進行安裝

```
$ tar zxvf postgresql-9.1.1.tar.gz
$ cd postgresql-9.1.1/contrib/dblink/
$ make
$ make -C ../../src/interfaces/libpq all
make[1]: Entering directory
`/home/postgres/postgresql-9.1.1/src/interfaces/libpq'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory
`/home/postgres/postgresql-9.1.1/src/interfaces/libpq'
$ su -c "make instsall"
Password:
make -C ../../src/interfaces/libpq all
make[1]: Entering directory
`/home/postgres/postgresql-9.1.1/src/interfaces/libpq'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory
`/home/postgres/postgresql-9.1.1/src/interfaces/libpq'
/bin/mkdir -p '/opt/PostgreSQL/9.1/lib/postgresql'
/bin/mkdir -p '/opt/PostgreSQL/9.1/share/postgresql/extension'
/bin/sh ../../config/install-sh -c -m 755  dblink.so
'/opt/PostgreSQL/9.1/lib/postgresql/dblink.so'
/bin/sh ../../config/install-sh -c -m 644 ./dblink.control
'/opt/PostgreSQL/9.1/share/postgresql/extension/'
/bin/sh ../../config/install-sh -c -m
644 ./dblink--1.0.sql ./dblink--unpackaged--1.0.sql
'/opt/PostgreSQL/9.1/share/postgresql/extension/'
```

- 建立 dblink 函數

  當編譯完成後會產生檔案「dblink--1.0.sql」，分別為 dblink 目錄下或是安裝到/opt/PostgreSQL/9.1/bin/share/postgresql/extension/目錄下，此兩個檔案皆為同一檔案，只有權限不同而已，由於需將「MODULE_PATHNAME」更改，才能順利進行函數建立；故只要修改其權限即改，接著指定欲安裝函數的資料庫名稱進行建立，須注意必須一個一個資料庫進行安裝喔，也就是說有進行安裝的資料庫才擁有這個功能。

```
$ sed -i 's/MODULE_PATHNAME/\$libdir\/dblink/g' dblink--1.0.sql
$ psql -f dblink--1.0.sql postgres
```

- 確認安裝

  進行 postgresql 命令模式執行 Meta-Command 確認是否順利安裝：

```
postgres=#\df
public | dblink | SETOF record | text                  | normal
public | dblink | SETOF record | text, boolean         | normal
public | dblink | SETOF record | text, text            | normal
public | dblink | SETOF record | text, text, boolean | normal
```

- 建立遠端資料庫伺服器存取

  指定遠端資料庫伺服器資料庫名稱、使用者帳號密碼及將此連線建立一名稱：

```
postgres# SELECT dblink_connect('mycon','host=localhost dbname=foo
user=foo password=foo);
```

- 查詢遠端資料庫資料表

  指定剛所建立連線名稱及所執行查詢的 SQL 語法，切記須指定該資料表的欄位類型喔！

```
postgres# SELECt  * FROM dblink('mycon','SELECT * FROM t1') as t1(name
          varchar(10));
```

- 更新遠端資料庫資料表資料

  如欲進行 DML 動作，執行以下 SQL 語法，須注意將單引號進行跳脫喔！

```
SELECT dblink_exec('mycon','INSERT INTO t1 VALUES(''123'');');
```

本書只是初步介紹大家如何使用 dblink 來存取遠端資料庫伺服器，如欲了解更多可參閱官方手冊，網址如下：

http://www.postgresql.org/docs/current/static/dblink.html

可修改內容的視觀表

PostgreSQL 在以前版本支援建立視觀表，但並無法修改視觀表中內容，但自 PostgreSQL9.0 版本後已突破此限制。

舉個例子來說，實際情況中並不見得需要查詢所有欄位資料：

```
store=# SELECT  * FROM  customers LIMIT 1;
```

建立視觀表來過濾我們欲呈現的資料內容，SQL 語法如下：

```
store=# CREATE VIEW  cust_view AS
        SELECT customerid, firstname,
         lastname, address1,
         city, country,
         region, creditcardtype,
         creditcard, creditcardexpiration,
         username, password
FROM  customers
```

接著再透過 cust_view 視觀表查詢資料，由於回傳欄位太多導致顯示不解讀，故改用延展模式顯示資料：

```
store=# \x
store=# SELECT * FROM cust_view limit 1;
-[ RECORD 1 ]--------+--------------------
customerid           | 19968
firstname            | WCXEDL
lastname             | KQUFVXLEBO
address1             | 5013247317 Dell Way
city                 | AJPXTTC
country              | Germany
region               | 2
creditcardtype       | 5
creditcard           | 1774747985415160
creditcardexpiration | 2011/07
username             | Rickynew
password             | password
```

在某此情況中，或許會希望直接透過視觀表，以下為 Insert 一筆資料列 SQL 語法，然而視觀表中內容預設是不允許修改的，故得到以下錯誤訊息：

```
store=# INSERT INTO customers
           (firstname,lastname,
           address1,city,
           country,region,
           creditcardtype,creditcard,
           creditcardexpiration, username,
           password)
         values
           ('Ricky','Liao',
            '4608499546 DellWay','QSDPAGD',
            'US',1,
             1,'1979279217775911',
             '2012/03','Ricky',
             'password');
ERROR:  cannot insert into view "cust_view"
HINT:  You need an unconditional ON INSERT DO INSTEAD rule or an INSTEAD OF
INSERT trigger.
store=# DELETE from cust_view where username='Ricky1';
ERROR:  cannot delete from view "cust_view"
HINT:  You need an unconditional ON DELETE DO INSTEAD rule or an INSTEAD OF
DELETE trigger.
```

透過 PostgreSQL 建立查詢重寫規則(Query Rewrite Rule)功能，如 Oracle 中 Instead-of Trigger 功能，可解決上述錯誤，也就是說呢！當我們欲針對視觀表中的內容執行 DML 動作如 Insert、Delete、Update，執行以下 SQL 語法建立規則即可：

```
store=# CREATE RULE cust_view_insert AS
        ON INSERT TO cust_view
        DO INSTEAD
        INSERT INTO customers (firstname,lastname,
        address1,city,
        country,region,
        creditcardtype,creditcard,
        creditcardexpiration,username,
        password)
        VALUES  (new.firstname,new.lastname,
        new.address1,new.city,
        new.country,new.region,
        new.creditcardtype,new.creditcard,
        new.creditcardexpiration,new.username,
        new.password);
store=# CREATE RULE cust_view_delete AS
        ON DELETE TO cust_view
        DO INSTEAD
        DELETE FROM customers
        WHERE username = old.username;
store=# CREATE RULE cust_view_update AS
        ON UPDATE TO cust_view
        DO INSTEAD
        UPDATE customers
        SET username = new.username
        WHERE customerid = old.customerid;
```

如建立重寫規則後需更改內容，只能透過重建方式重新建立規則，將 CREATE RULE 改為 CREATE OR REPLACE RULE。

在建立完以上規則後再次進行 Insert、Delete、Update 就不再產生之前所發生錯誤囉！如此便可更新視觀表中的內容。

```
store=# INSERT INTO cust_view
        VALUES ('Ricky1','Liao',
        '4608499546 Dell Way','QSDPAGD',
        'US',1,
        1,'1979279217775911',
        '2012/03','Ricky1',
        'password');
INSERT 0 1
store=# DELETE FROM cust_view where username='Ricky1';
DELETE 1
store=# UPDATE cust_view set username = 'Rickynew'
        WHERE customerid = 20001;
UPDATE 20001
```

更多…

剛介紹過如何建立查詢重寫規則，然而如何查詢已建立過的規則呢？透過系統資料表 pg_rules 即可查詢規則名稱及對應物件，SQL 語法如下：

```
store=# SELECT tablename, rulename FROM pg_rules;
  tablename  |     rulename
-------------+------------------
 pg_settings | pg_settings_n
 pg_settings | pg_settings_u
 cust_view   | cust_view_update
```

如在建立規則後不再使用，執行以下 SQL 語法即可刪除：

```
 store=# DROP RULE cust_view_update ON cust_view;
```

CHAPTER 8

監控與診斷

本章內容大綱如下：

- 找出已連接資料庫伺服器使用者
- Active 使用者
- Active 或 Block
- 找出鎖定原因
- 刪除指定 Session
- 解決有問題的 preppared transaction
- 哪些使用者存取資料表
- 暫存資料佔用空間
- 查詢效能差原因
- 找出 Bug
- 錯誤日報

前言

找出已連接資料庫伺服器使用者

本章節將教導大家如何查看有哪些使用者已連接至資料庫伺服器中。

準備開始

首先，以 Superuser 使用者登入資料庫伺服器。

執行以下 SQL 語法，找出已連接至資料庫伺服器所有使用者名稱：

```
postgres=# SELECT usename FROM pg_stat_activity;
```

也可以指定使用者過查看使用者是否已連接資料庫伺服器：

```
postgres=# SELECT usename, dtname
           FROM pg_stat_activity
           WHERE usename = 'foo';
```

如何運作…

PostgreSQL 系統資料表 pg_stat_activity 會持續記錄所有連接至資料庫伺服器的使用者行為，包括執行 SQL 語法內容、是否正在執行及開始執行時間…等資訊。

找出指定用戶端的使用者正在連接資料庫伺服器：

在一般情況中同一個使用者可能連接資料庫伺服器超過二個以上，此時就得去判斷該使用者是透過哪個用戶端連接，以便在問題發生時快速可快速解決。

```
postgres=# SELECT datname, usename, client_addr,
           client_port ,client_hostname
           FROM pg_stat_activity;
```

根據 client_addr、client_port、client_hostname 將幫住我們追查該使用者透過哪個用戶端連接。

查詢使用者動作

本章將引導大家如何查詢使用者正在執行何種動作？

準備開始

首先，以 Superuer 使用者登入並確認資料庫伺服器參數已設定為 track_activities = on ，可修改參數檔 postgresql.conf 或執行以下 SQL 執行：

```
postgres=# SET track_activities = on
```

如何做…

查詢哪些使用者已連接資料庫伺服器並找出其 SQL 語法：

```
postgres=# SELECT datname,usename,current_query
           FROM pg_stat_activity;
```

一般而言當大量使用者連接資料庫伺服器後，並非一直持續有執行動作，故會發現於 current_query 欄位中的值，有些使用者會呈現「<IDLE>」狀態，表示該使用者此時並未執行任何 SQL 語法，而 PostgreSQL 正在等待用戶端執行指令。

可改執行以下 SQL 語法，過濾出正在執行動作的使用者：

```
postgres=# SELECT datname,usename,current_query
           FROM pg_stat_activity
           WHERE current_query != '<IDLE>';
```

如何運作…

當參數設定 track_activities = on 時，PostgreSQL 會收集使用者執行動作，並將記錄寫入至系統資料表 pg_stat_activities。

另外可透過執行函式 pg_stat_get_activity()，藉由輸入用戶端所產生的執行程序(Process ID)，查詢該執行程式正在進行什麼動作：

找出用戶端所產生的程行程序：

```
#ps -elf|grep postgres
postgres   727 22712  0 09:50 ?  postgres: postgres postgres [local] idle in transaction
```

查詢該執行程序狀態：

```
postgres=# SELECT  pg_stat_get_activity(727);
```

而當輸入值指定為 NULL 時則會列出所有執行程序狀態：

```
postgres=# SELECT  pg_stat_get_activity(NULL);
```

更多…

在某些情況中我們想分析資料庫被執行了哪些 SQL 指令，而一般線上交易處理 (OLTP, Oline Transaction Processing) 系統，執行指令非常的短促，可能只要幾千毫秒即執行完畢，則無法透過系統資料表 pg_stat_activity 查詢，不如安裝 PostgreSQL 9.0 另一額外模組，名為 pg_stat_statements，可即時記錄資料庫統計值，便可進行分析，可參閱以下網址：
http://developer.postgresql.org/pgdocs/postgres/pgstatstatements.html

也可透過以下二種方式來抓取執行 SQL 語法，第一透過腳本方式使用 SELECT pg_sleep(5)，讓執行 SQL 指令後自動停止 5 秒鐘，方便我們由 pg_stat_activity 觀察；第二，設定參數 log_statement，開啟是否記錄所有被執行 SQL 語法，也可以修改資料庫參數檔 postgresql.conf 來完成設定：

```
postgres=# SET log_statement='all';    #開啟
postgres=# SET  log_statement=none;    #關閉
```

找出執行指令所需時間長度由長至短排列：

上面介紹如何找出用戶端所執行的 SQL 語法，但在某些時殊情況中，須找出執行較久的 SQL 語法及所執行的用戶端，進行分析解決，則執行以下 SQL 語法：

```
postges=# SELECT datname,usename,current_query,
          current_timestamp = query_start AS runtime
           FROM pg_stat_activity
           WHERE current_query != '<IDLE>'
           ORDER by 1 desc;
```

透過以上 SQL 語法，將找出執行時間較長之 SQL 語法，並根據時間長短排序。

使用 ps 指令查看 SQL 指令執行狀態：

雖然 PostgreSQL 預設已將此參數 update_process_title 開啟，但藉由此設定，可快速查看該用戶端連接資料庫伺服器且執行 SQL 指令時狀態，如下所示：

```
        postgres: postgres postgres [local] SELECT
        postgres: postgres postgres [local] idle
        postgres: postgres postgres [local] idle in transaction
```

假設您的資料庫伺服器未開啟此參數值則可執行以下 SQL 語法開啟：

```
postgres=# SET update_process_title = on;
```

使用者連線是否被鎖定

藉由查詢系統資料表 pg_stat_activity 的欄位 waiting 來判斷該使用者連線是否等待其它使用者執行 SQL 完畢，方能進行執行 SQL，也就是說此使用者處於被鎖定(Blocked)狀態：

```
postgres=# SELECT datname, usename, current_query
           FROM pg_stat_activity
           WHERE waiting = true;
```

另外，由於欄位 waiting 為佈林值，我們可省略字串「= true」，如以下 SQL 語法：

```
postgres=# SELECT datname, usename, current_query
           FROM pg_stat_activity
           WHERE waiting;
```

找出造成鎖定的使用者連線

當發現資料庫伺服器可能疑似資料鎖定造成執行異常，可執行以下 SQL 語法，使用者正在執行哪些語法造成哪些資料表鎖定：

```
SELECT
w.current_query as waiting_query,
w.procpid as w_pid,
w.usename as w_user,
l.current_query as locking_query,
l.procpid as l_pid,
l.usename as l_user,
t.schemaname || '.' || t.relname as tablename
from pg_stat_activity w
join pg_locks l1 on w.procpid = l1.pid and not l1.granted
join pg_locks l2 on l1.relation = l2.relation and l2.granted
join pg_stat_activity l on l2.pid = l.procpid
join pg_stat_user_tables t on l1.relation = t.relid
where w.waiting;
```

此 SQL 語法，首先查詢出被鎖定的 SQL 語句(pg_lock)上正在等待哪個 SQL 執行完畢(w.waiting)。

踢除使用者連線 Kill Session

本章帶領大家如何強制將使用者踢除。

首先，以 Superuser 使用者登入資料庫伺服器，接著之前介紹，找出欲踢除使用者所產生的執行程序，再執行函數 pg_terminate_backend(processid)將達到：

```
postgres=# SELECT datname, procpid FROM pg_stat_activity;
postgres=# SELECT pg_terminate_backend(18179);
```

如何運作…

當執行 pg_terminate_backend 函數時，PostgreSQL 會送信號 SIGQUIT 至該執行程序，當執行程序接受到此信號時便進行中止。

然而中止該使用者連線並非是最好方式，可先以較柔合方式進行，執行函數 pg_cancel_backend(processid)，pg_terminate_backend()與 pg_cancel_backend()最大的差異是 pg_cancel_backend()僅中止正在執行 SQL 語句，並未中斷該使用者連線即該執行程序。

```
postgres=# SELECT pg_cancel_backend(18179);
```

萬一 pg_terminate_backend()無法順利將該程行程序刪除，那就只好改由作業系統傳送信號 SIGKILL，只能由作業系統命令端執行該指令，且該使用者須為 root 或 postgres 帳號。

```
# kill -9 <processid>
```

注意，如要以 kill -9 方式刪除使用者所產生的執行程序，最好修改資料庫伺服器參數 synchronous_commit 值為 off，預設為 on，避免當強制中斷使用者時，因交易 commit 而造成資料遺失；執行 kill -9 是在最逼不得已狀況下才會這麼做，不要將此動作視為常態。

當執行 SQL 語法時間超過限定時間自動進行中斷

在某此情況中，如配合應用程式並不希望執行 SQL 語法所需時間太長，此時可藉由資料庫伺服器參數 statement_timeout 來限定時間，預設時間單位為毫秒 (Milliseconds)，可改以秒(s)為單位，可修改 postgresql.conf 檔或執行以為 SQL 語法：

```
postgres=#  SET statement_timeout='3 s';
SET
postgres=# SELECT pg_sleep(10);
ERROR:  canceling statement due to statement timeout
```

以上，將 timeout 時間設為 3 秒，接著藉由執行函數 pg_sleep()讓該使用者停置 10 秒，由於超過時間限制故出現錯誤訊息。

當使用者連線閒置超過指定時間時進行踢除

某此情況中，可能開發人員程式撰寫問題或用戶端操作問題，導致使用者登入執行所執行 SQL 語法後，未進行登出將連線中斷，而導致資料庫伺服器殘留大量使用者連線而佔用伺服器資源，進而影響效能。

故可執行以下 SQL 語法，在使用者連線閒置超過指定時間 20 分鐘進行踢除：

```
postgres=# SELECT  pg_terminate_backend(procpid)
           FROM  pg_stat_activity
           WHERE current_query LIKE '%IDLE%'
           AND current_timestamp - query_start > '20 min';
```

如欲達到自動化將使用者連線閒置超過指定時間 20 分鐘時進行踢除可以 psql 指令載入腳本方式來達到此動作。

解決有問題的 Prepared Transaction

當使用 2PC(Two phase Commit)時，可能遭遇鎖定而造成交易中斷，可透過以下 SQL 語法查詢哪些資料表及執行程序所造成：

```
SELECT t.schemaname || '.' || t.relname as tablename,
l.pid, l.granted
FROM pg_locks l join pg_stat_user_tables t
ON l.relation = t.relid;
```

找出未使用資料表

一些情況中部份資料表可能因某些情況而不再使用，但卻佔俱在資料庫中而浪費空間，故本節教導大家如何找出未使用資料表。

準備開始

確認已使用 Superuser 使用者登入或者該使用者對所有資料表擁有全部權限。

首先由系統 pg_stat_user_tables 資料表複製資料至暫存表 tmp_stat_user_tables：

```
CREATE temp table tmp_stat_user_tables
 AS SELECT * FROM pg_stat_user_tables;
```

建立資料表 tmp_stat_user_tables 一段時間後，執行以下 SQL 語法，將列出此段時間無任何存取動作之資料表，換句話說有被存取過的資料表由於會記錄至系統資料表 pg_stat_user_tables 內故不會出現在此查詢：

```
SELECT * FROM pg_stat_user_tables n
JOIN tmp_stat_user_tables t
ON n.relid=t.relid
AND (n.seq_scan,n.idx_scan,n.n_tup_ins,n.n_tup_upd,n.n_tup_del) =
(t.seq_scan,t.idx_scan,t.n_tup_ins,t.n_tup_upd,t.n_tup_del);
```

更多…

環境中如欲清除重設資料表統計值，可執行以下 SQL 語法使統計值歸零：

```
postgres=# SELECT pg_stat_reset();
```

待清除完畢後，接續有任何資料表存取動作也同樣會寫入至系統資料表 pg_stat_user_tables 中。

備份資料表統計值

剛介紹了當資料表有任何存取動作時將記錄至系統資料表 pg_stat_user_tables 中，故我們可藉此備份特定資料表或某一時間點的資料表統計值，以便日觀察比較資料表是否異動或效能是否有異，可參考以下作法：

第一，複製系統資料表 pg_stat_user_tables 中的資料至資料表 backup_stat_user_tables 中並於該資料表內增加一時間欄位：

```
CREATE table backup_stat_user_tables AS
SELECT current_timestamp AS snaptime, *
FROM  pg_stat_user_tables;
```

第二，定期將資料表統計值某一時間點資料塞入至資料表 backup_stat_user_tables。

```
INSERT INTO backup_stat_user_tables
SELECT current_timestamp AS snaptime, *
FROM pg_stat_user_tables;
```

欲達成定期塞入資料表統計值可透過作業系統排程執行 psql 載入 SQL 腳本或使用 PostgreSQL 額外模組 pg_agent。

計算暫存表所佔用空間

當查詢進行排序時如資料庫伺服器記憶體不夠時便會建立暫存表協助排序，如能計算暫存表所佔用空間，將助於分析資料庫使用狀況。

準備開始

首先，使用以下 SQL 語法查詢暫存表格空間，預設並未設定：

```
postgres=# SELECT current_setting('temp_tablespaces');
```

故以下二步驟，將分別建立表格空間 temp1 與 temp2，並將此二表格空間設定為暫存表格空間：

```
postgres=#  CREATE TABLESPACE temp1 LOCATION '/postgres/temp1';
CREATE TABLESPACE
postgres=#  CREATE TABLESPACE temp1 LOCATION '/postgres/temp2';
CREATE TABLESPACE
postgres=# SET temp_tablespaces=temp1,temp2;
```

再次執行以下 SQL 語法，查詢暫存表格空間設定是否生效：

```
postgres=# SELECT current_setting('temp_tablespaces');
current_setting
-----------------
 temp1, temp2
(1 row)
```

藉由系統資料表 pg_tablespace 查詢，暫存表格空間所在目錄：

```
postgres=# SELECT * from pg_tablespace;
  spcname   | spcowner |   spclocation    | spcacl | spcoptions
------------+----------+-----------------+--------+------------
 pg_default |       10 |                 |        |
 pg_global  |       10 |                 |        |
 temp1      |       10 | /postgres/temp1 |        |
 temp2      |       10 | /postgres/temp2 |        |
```

除了查詢會使用到暫存表格空間外，當建立暫存表時也會建立於暫存表格空間，以下將舉個例子帶領大家：

首先，先進入剛建立的暫存表格空間 temp1，並計算該目錄所佔空間：

```
# du -s
8
```

建立暫存表名稱為 temp_tb1，再次計算該目錄所佔空間：

```
postgres=#  CREATE TEMPORARY TABLE temp_tb1 (a varchar(10));
CREATE TABLE
```

```
# du -s
12
```

故可得知，當使用者建立暫存表時會於暫存表格空間所建立目錄上建立一資料檔，當然空間大小也會隨者使用量而增加。

除透過以上方式可計算暫存表空間外，也可透過以下 SQL 語法計算，由其當資料庫伺服器預設並未設定暫存表格空間時，如有建立暫存表時，該檔案將建立於$PGDATA/base 目錄下：

```
select sum(pg_total_relation_size(relid))
from pg_stat_all_tables
where schemaname like 'pg_%temp%';
```

設定暫存表格空間警戒值

資料庫伺服器參數 log_temp_files 預設值為 0，也就是說不管產生多大暫存空間也不會發生警示，但如須監控暫存表格使用空間時，則可設定 0 以上數字，此設是以 kilobytes 為單位，當超過此設定值時便將警示寫入至資料庫伺服器日誌檔。

照理說暫存表會在於使用者使用完成後自動清除空間，然而在某此情況中可能未自動清除，如資料庫伺服器發生異常而重啟，此時可能就需要手動清除這些沒必要的暫存表，如資料庫伺服器有設定暫存表格空間時，只要清除該暫存目錄即可；然而如未設定暫存表格空間，預設會寫入至$PGDATA/base 目錄中，檔案名稱為 tx_xxxx，由於混於資料庫伺服器資料目錄中，故不建議清除。

產生資料庫伺服器日誌檔分析報告

資料庫伺服器日誌檔會記錄資料庫伺服器所發生事件，然而在某些情況中可能一天就會產生好幾百 MB 甚至 GB 的資料量，就變的難以解讀分析，也就失去資料庫伺服器日誌檔的意義了。

準備開始

為避免資料庫伺服器日誌檔因太大而造成不易解讀，可於資料庫伺服器設定檔 postgresql.conf 設定此參數，以天為單位，針對資料庫伺服器日誌檔進行輪轉 (Log Rotation)

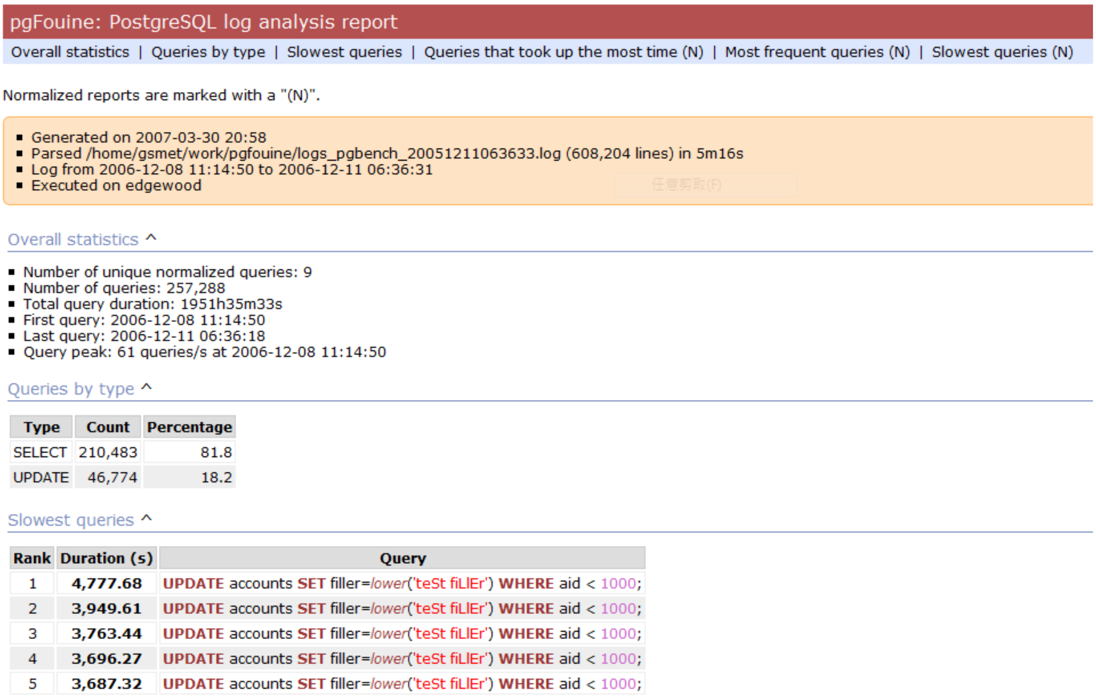
```
log_rotation_age = 1d
```

接著，安裝資料庫伺服器日誌檔分析程式 pgFouine，在安裝程式後，須特別注意將輸出格式設為 syslog，可更改參數 log_destination 為 syslog，如此 pgFouine 才能正確進行分析。

待設定完成後執行指令，產生分析報表：

```
# pgfouine.php -file /postgres/postgresql.log -reports
n-mostfrequenterrors > errors.html
```

以下為分析報表，為 HTML 格式，可配合作業系統排程定期產生報表，甚至在產生報表後以郵件方式傳送至 DBA。



pgFouine: PostgreSQL log analysis report

Overall statistics | Queries by type | Slowest queries | Queries that took up the most time (N) | Most frequent queries (N) | Slowest queries (N)

Normalized reports are marked with a "(N)".

- Generated on 2007-03-30 20:58
- Parsed /home/gsmet/work/pgfouine/logs_pgbench_20051211063633.log (608,204 lines) in 5m16s
- Log from 2006-12-08 11:14:50 to 2006-12-11 06:36:31
- Executed on edgewood

Overall statistics ^

- Number of unique normalized queries: 9
- Number of queries: 257,288
- Total query duration: 1951h35m33s
- First query: 2006-12-08 11:14:50
- Last query: 2006-12-11 06:36:18
- Query peak: 61 queries/s at 2006-12-08 11:14:50

Queries by type ^

Type	Count	Percentage
SELECT	210,483	81.8
UPDATE	46,774	18.2

Slowest queries ^

Rank	Duration (s)	Query
1	4,777.68	UPDATE accounts SET filler=lower('teSt fiLlEr') WHERE aid < 1000;
2	3,949.61	UPDATE accounts SET filler=lower('teSt fiLlEr') WHERE aid < 1000;
3	3,763.44	UPDATE accounts SET filler=lower('teSt fiLlEr') WHERE aid < 1000;
4	3,696.27	UPDATE accounts SET filler=lower('teSt fiLlEr') WHERE aid < 1000;
5	3,687.32	UPDATE accounts SET filler=lower('teSt fiLlEr') WHERE aid < 1000;

更多…

剛於更多有關於 pgFouine 軟體資訊可參閱網址：
http://pgfouine.projects.postgresql.org/

除了上述 pgFouine 資料庫伺服器日誌檔分析工具外，也有其它類似產品，PQA (Practical Query Analysis)，此工具由 Ruby 撰寫而成，可參閱網址：
http://pqa.projects.postgresql.org/

CHAPTER 9

資料庫維護機制

本章內容大綱如下：

- 資料庫自動維護機制
- 避免資料毀損
- 避免交易
- 移除舊的 Prepared Transactions
- 暫存表
- 辨別資料表及索引
- 管理索引
- 找出未使用索引
- 移除未使用索引
- 規劃如何進行資料庫維護

前言

PostgreSQL 內建擁有許多定期維護機制，本章節將以如何規劃維護資料庫為重點介紹。

PotgreSQL 提供一維護工具 VACUUM，可收集並清除一些資料庫垃圾資訊並進行修復，此工具與 ANALYZE 指令相似，可用於執行最佳化統計資訊，而此動作可透過 autovacuum 以背景程序執行達到自動化。，

PostgreSQL 9.0 版本預設啟用 Autovacuum，此動作機乎涵概所有資料庫例行性維護工作，然而執行 Autovacuum 時會影響資料庫效能，故如何訂定執行時間就成為一個重點課題。

準備開始

我們需要考量以下幾點：

- 一天中哪個時段系統資源較為充足，適何執行資料庫維護。
- 一個星期哪幾天較不忙碌，又哪幾天較忙碌。
- 資料庫中哪些資料表對於應用程式較為重要？

首先為確保 Autovacuum 已開啟，請確認以下兩個參數已於 postgresql.conf 檔中設定正確：

- autovacuum = on
- track_counts = on

在 PostgreSQL 中有 35 個參數針對 Autovacuum 工具進行調整，以符合不同環境。

- autovacuum
- autovacuum_analyze_scale_factor
- autovacuum_analyze_threshold
- autovacuum_freeze_max_age
- autovacuum_max_workers
- autovacuum_naptime
- autovacuum_vacuum_cost_delay
- autovacuum_vacuum_cost_limit
- autovacuum_vacuum_scale_factor
- autovacuum_vacuum_threshold
- log_autovacuum_min_duration

PostgreSQL 中可分別對資料表進行 Storage 參數設定，語法如下：

```
ALTER TABLE table_name SET (storage_parameter = value);
```

Storage 相關參數如下：

- autovacuum_enabled
- autovacuum_vacuum_cost_delay
- autovacuum_vacuum_cost_limit
- autovacuum_vacuum_scale_factor
- autovacuum_vacuum_threshold
- autovacuum_freeze_min_age
- autovacuum_freeze_max_age
- autovacuum_freeze_table_age
- autovacuum_analyze_scale_factor
- autovacuum_analyze_threshold

「TOAST」資料表(The Oversized-Attribute Storage Technique 特大儲存技術資料表)，可使用以下參數：

- toast.autovacuum_enabled
- toast.autovacuum_vacuum_cost_delay
- toast.autovacuum_vacuum_cost_limit
- toast.autovacuum_vacuum_scale_factor
- toast.autovacuum_vacuum_threshold
- toast.autovacuum_freeze_min_age
- toast.autovacuum_freeze_max_age
- toast.autovacuum_freeze_table_age
- toast.autovacuum_analyze_scale_factor
- toast.autovacuum_analyze_threshold

TOAST 資料表，一般而言 PostgreSQL 的頁面大小是固定 8Kb，並不允一個資料表內單一欄位中記錄超過一個頁面，但 PostgreSQL 7.1 後克服此限制，允許跨過多個頁面就稱為
TOAST(The Oversized-Attribute Storage Technique)，超大型字串儲存技術

如何運作…

當資料庫啟用 Autovacuum，會根據參數 autovacuum_naptime 所設定間隔時間，以秒為單位定期執行，選擇執行 VACUUM 或 ANALYZE。

一般情況中或許有人覺得同時啟用多個執行程序來加快 Vacuum 執行速度，但其實是不需要的喔！不用特地針對參數 autovacuum_max_workers 進行調整，保留預設值即可，因 Autovacuum slave 執行 I/O 動作時，會統計共消耗多少 I/O 資源，直到達到 autovacuum_vacuum_cost_limit 設定限制，便強制將 Autovacuum 停止，直到閒置時間所設定值 autovacuum_vacuum_cost_delay 到達後才繼續執行，主要避免執行 Autovacuum 佔據太多 I/O 資源，故也可藉由設定 autovacuum_vacuum_cost_delay 來降低執行太頻繁執行 Autovacuum 而造成系統資源衝擊，autovacuum_analyze_threshold 可用於設定當 autovacuum 啟動時執行 ANALYZE 一次 update 或 delete 多少記錄。

另外，當 Autovacuum 執行，如希望執行花費超過指定時間時，將記錄至資料庫日檔，可設定參數 log_autovacuum_min_duration 來達成。

關於 Autovacuum 相關參數並非只是侷限於 Instance 層級，如果只是想針對特定資料表設定也可於資料表層級執行：

```
postgres=# ALTER TABLe t1 SET (autovacuum_enabled = off);
ALTER TABLE
```

資料表層級也適用於 TOAST 資料表，

```
postgres=# ALTER TABLE t1 SET (toast.autovacuum_enabled = on,
        autovacuum_vacuum_cost_delay = 20);
ALTER TABLE
```

以上介紹我們可針對資料表層級設定 autovacuum，故提供以下 SQL 語法讓大家查詢特定資料表中各自 autovacuum 參數設定：

```
postgres=# SELECT n.nspname, c.relname,
         pg_catalog.array_to_string(c.reloptions || array(
         select 'toast.' ||
         x from pg_catalog.unnest(tc.reloptions) x),', ')
         as relopts
         FROM pg_catalog.pg_class c
         LEFT JOIN
         pg_catalog.pg_class tc ON (c.reltoastrelid = tc.oid) JOIN
         pg_namespace n ON c.relnamespace = n.oid
         WHERE c.relkind = 'r'
         AND nspname NOT IN ('pg_catalog', 'information_schema');
```

在進行 VACUUM 時是允許 Insert, Update, Delete 動作的，盡管在此期間是禁止 ALTER TABLE 或 CREATE INDEX。故 Autovacuum 會自動偵測在執行 VACUUM 期間是否有使用者因執行 ALTER TABLE 或 CREATE INDEX 而產生的鎖定，便自動停止 VACUUM 動作。

執行 VACUUM 後一般並不會壓縮(Shrink)資料表，除非資料表未端部份有大量未使用空間，且無任何使用者正在存取才會進行壓縮，為了正確壓縮資料表，可改執行 VACUUM FULL 執令，VACUUM FULL 將針對資料表內容重新排列並重建所有相關索引，在 PostgreSQL9.0 版本中此工具效能更優異，對於大型資料表重整更快速。

更多…

資料庫伺服器設定檔 postgresql.conf 是允許參照外部檔的，也就是說我們可將關於 Autovacuum 相關參數寫成一個檔案如 autovacuum.conf，再於 postgresql.conf 編輯如下參照至 autovacuum.conf。

```
include 'autovacuum.conf'
```

透過此方式，只需維護此三個參數值即可，就可應用在環境中，資料庫會依據日夜採取不同設定值符合不同需求，而不用一個一個修改相關 autovacuum 設定值

postgresql.conf 設定如下：

```
autovacuum = on
autovacuum_max_workers = 3
include 'autovacuum.conf.day'
#or
#include 'autovacuum.conf.night'
```

autovacuum.conf.day 設定值：

```
autovacuum_analyze_scale_factor = 0.1
autovacuum_analyze_threshold = 50
autovacuum_vacuum_cost_delay = 30
autovacuum_vacuum_cost_limit = -1
autovacuum_vacuum_scale_factor = 0.2
autovacuum_vacuum_threshold = 50
```

autovacuum.conf.night 設定值：

```
autovacuum_analyze_scale_factor = 0.05
autovacuum_analyze_threshold = 50
autovacuum_vacuum_cost_delay = 10
autovacuum_vacuum_cost_limit = -1
autovacuum_vacuum_scale_factor = 0.1
autovacuum_vacuum_threshold = 50
```

此案例中，我們將日夜分成兩個 profile 各自套用不同 Autovacuum 設定檔，以符合資料庫於不同期間所需的環境。

避免自動凍結及頁面毀損

當 PostgreSQL 定期清除舊交易記錄識別時造成交易記錄延遲產生的期間稱為凍結(Freezing)。

PostgreSQL 針對資料列(Row)有以下動作，當 Row Version 不再可用時使用 VACUUM 清除，或 ROW Version 太過老舊時將被 VACUUM 凍結。

舉個例子來說當某資料表擁有一百萬筆資料時，而因資料量太大造成無法凍結，故下一步 VACUUM 執行清理資料列並凍結交易識別；換言之，Autovacuum 將啟動並開始執行凍結而造成 I/O 大量衝擊。

也因為如此原因，Autovacuum 總是在不適合的時間自動執行造成效能上衝擊，一般人直覺反應下就將 Autovacuum 機制關閉，不信任 Autovacuum 會對資料庫效能帶來好處，我將在「控制自動化資料庫維護機制」章節詳細介紹。

一般而言 VACUUM 只會少量凍結資料，甚至感覺不到影響，只有在前面所介紹到大資料表可能才會造成效能問題。

當某張資料表屬性 pg_class.relfrozenxid 欄位到達參數 vacuum_freeze_table_age 設定值時，便會進行整張資料表掃描 ，而造成大量 I/O

如果想預防 VACUUM 無預警啟動，可透過設定參數 autovacuum_freeze_max_age 來控制何時 VACUUM 會強制掃描，也就是說我們無法關閉凍結 Row，但可以控制何時發生。故建議控制 Autovacuum 讓它在指定的時間內啟動。

VACUUM 除能最佳化資料庫外，也是用於檢查區塊是否毀損，值得定期執行 VACUUM 掃描資料庫，可執行以下 SQL 語法進行區塊檢查：

```
postgres=# set vacuum_freeze_table_age = 0;
SET
postgres=# vacuum;
VACUUM
postgres=#
```

當不幸遇到區塊毀損時，到目前為止並無法修復，只能盡量從已毀損區塊將資料救回存到不同位置。

避免 Transaction Wraparound

對許多人而言，太多「Transaction wraparound」對硬碟空間而言是個傷害，關於 Transaction wraparound

準備開始

首先，有無以下訊息？

```
WARNING: database "postgres" must be vacuumed with XXX transactions
HINT: To avoid a database shutdown, execute a database-wide VACUUM in that
database.
You might also need to commit or roll back old prepared transactions.
```

或更壞的訊息：

```
ERROR: database is not accepting commands to avoid wraparound data loss in
database "template0"
HINT: Stop the postmaster and use a standalone backend to vacuum that
database.
You might also need to commit or roll back old prepared transactions.
```

如果沒出現以上訊息，恭禧你，只要持續進行資料庫維護即可，以上訊息主要是為了警告使用者且會寫入資料庫伺服器日誌檔中。

如何做…

當資料庫伺服器出現以上訊息時，首先在提示出現問題的資料庫上執行 vacuum 指令：

```
# vacuumdb postgres
```

或者也可在 psql 下執行：

```
postgres=#  vacuum postgres;
```

如何運作

PostgreSQL 內部使用交易識別器(Transaction identifiers)，大小約 4 位元組，約有 2^31 次方(約二十億)個識別 id，當 PostgreSQL 運作後會重複循環配置新識別器，主要原因為是每次移動使用 8 位元組識別器會造成更大效能影響，故選擇使用 4 位元組。

PostgreSQL 在開始運作後會一直使用交易識別器，故定期使用 VACUUM 維護資料庫，將幫助資料庫正常運作也不再發生以上錯誤訊息。

當遭遇到前先所提到錯誤訊息時，資料庫便無法再接新的指令，此時只能讓資料庫進入單一使用者模式(Single user mode)，也就是「Stand-alone backend」。

```
# postgres --single -D /opt/Postgresql9.0/data postgres
PostgreSQL stand-alone backend 9.1rc1
backend>
```

接著執行 VACUMM 進行重整修復：

```
backend> VACUUM;
backend>
```

修復完畢後鍵盤輸入<CRTL> + D 跳出即可。

移除逾時 Prepared Transactions

Prepared Transaction 只有某些特殊用途才會產生，Prepared Transaction，簡稱 2PC 是 Two-phase commit 其中特色之一;Transactions Commit 會發生於兩種情況：第一，同時更新資料於二個以上資料庫時，為了保持全部資料庫一致性(不是全 Commit 就是全 Rollback)；第二，為了使用 XA 協定架構，同時使用於二個以上的 Transaction Manager(TM)，如使用 JAVA Transaction API(JTA)或其它中介軟體(Middleware)；如果以上情況皆未符合則不需使用 Prepared Transactions。

準備開始

首先，確認資料庫伺服器有無設定參數 max_prepared_transactions，假設參數為 0 時，則不須後續動作，或者的確很難了解 Prepared Transactions 運作，也可以將參數設為 0，避免產生其它問題。

執行以下 SQL 語法查詢 max_prepared_transactions：

```
postgres=# SHOW max_prepared_transactions;
```

執行以下 SQL 語法查詢資料庫伺服器中有哪些逾時 Prepare Transactions：

```
postgres=# \x
postgres=# SELECT * FROM pg_prepared_xacts;
-[ RECORD 1 ]-----------------------------
transaction | 121083
gid | prep1
prepared | 2010-03-28 15:47:57.637868+01
owner | postgres
database | postgres
```

如何做…

移除逾時 Prepared Transaction 可參考章節「解決未知 Transaction」。在 Two-phase commit 機制中資料庫伺服器當資料庫或 Transactions Manager 發生異常 Crash 時，會產生 Prepared Transaction。

假設連接池有 100 個連線正在交易，但不幸的，Transaction Manager 發生 Crash，可能會有 1 到 20 個以上的交易會處於 prepared 狀態。

為了要解決此交易狀態，此時我們得決定這些交易要進行 Commit 或 Rollback。

如要進行 Commit 執行以下 SQL 語法：

```
postgres=# COMMIT PREPARED 'prep1';
```

如要進行 Rollback 執行以下 SQL 語法：

```
postgres=# ROLLBACK PREPARED 'prep1';
```

如何運作…

Prepared Transactios 是為了確保交易的一致性，當然有 Prepared Transaction 發生時並不是重啟資料庫實例就可以解決。

假設我們不清楚發生 Prepared Transactions 要進行何種動作時，可執行以下 SQL 語法查詢哪些交易正鎖定資料表：

```
postgres=# SELECT l.locktype, x.database, l.relation, l.page,
l.tuple,l.classid, l.objid, l.objsubid,
l.mode, x.transaction, x.gid, x.prepared,
x.owner
FROM pg_locks l JOIN pg_prepared_xacts x
ON l.virtualtransaction = '-1/' ||
x.transaction::text
```

```
postgres=# SELECT DISTINCT x.database, l.relation
FROM pg_locks l JOIN pg_prepared_xacts x
ON l.virtualtransaction = '-1/' ||
x.transaction::text
WHERE l.locktype != 'transactionid';
```

藉由以上查詢結果，再決定該 Prepared Transactions 要進行 Commit 或 Rollback 即可。

管理暫存表

當使用者查詢資料需要大量排序時，需大量使用記憶體，然而實體記憶體是有限的，故資料庫便會建立暫存表來解決實體記憶體不足。

如何做…

- 確認系統資料表已執行 VACUUM，或有啟動 Autovacuum 功能。
- 觀察使用者查詢期間將使用暫存表空間。
- 調整記憶體參數。考量是否加大參數 temp_buffers；但須注意避免大於作業系統實體記憶體總和而造成記憶體切換(Swaping)。
- 將資料庫伺服器暫存表置於獨立掛載區以分散 I/O，在某些情況中暫存表所造成的 I/O 量可比其它資料表或索引所造成的 I/O 還高出許多！故可設定參數 temp_tablespace 將暫存表建立於指定表格空間內來分散 I/O。

如何運作…

PostgreSQL9.0 版本後當建立暫存表會同時於資料庫系統資料表 pg_class 與 pg_attribute 建立該筆記錄，故此系統資料表及索引將隨時間成長，為了讓系統資料表維持良好運作，可定期使用 VACUUM 或啟動 Autovacuum。

如欲使用手動執行 VACUUM 維護系統資料表，可透過以下 SQL 語法列出所有系統資料表：

```
postgres=# SELECT relname, pg_relation_size(oid)
FROM pg_class
WHERE relkind in ('i','r') and relnamespace = 11
ORDER BY 2 DESC;
               relname                | pg_relation_size
----------------------------------------+------------------
 pg_proc                              |         466944
 pg_depend                            |         360448
 pg_attribute                         |         327680
 pg_description                       |         237568
 pg_depend_depender_index             |         221184
```

```
pg_depend_reference_index               |           212992
pg_collation                            |          196608
pg_proc_proname_args_nsp_index          |          188416
pg_statistic                            |          139264
pg_attribute_relid_attnam_index         |          122880
pg_description_o_c_o_index              |          122880
pg_operator                             |          106496
pg_rewrite                              |           98304
pg_attribute_relid_attnum_index         |           90112
pg_proc_oid_index                       |           73728
pg_class                                |           65536
pg_type                                 |           65536
pg_collation_name_enc_nsp_index         |           49152
pg_statistic_relid_att_inh_index        |           40960
pg_operator_oprname_l_r_n_index         |           40960
pg_collation_oid_index                  |           40960
pg_type_typname_nsp_index               |           40960
.......(略)
```

註解： reliknd 欄位中 r 為資料表，i 為索引，s 為循序列(Sequence)，t 為 TOAST。

一般而言每個系統資料表 Size 應該不大，然而如未定期維護資料庫可能造成 Size 高達好幾 GB 以上，故定期執行 VACUUM 或 Autovacuum，不僅可避免空間異常增加，也有助於資料庫效能。

識別資料表與索引

PostgreSQL 支持 MVCC(Multi-Version Concurrency Control)，儘管資料庫有 DML 動作，仍允許二個以上使用者同時讀取資料，由於這樣的特色故有以下優點：

- 資料表鎖定較少發生，效能高。
- 減少資料庫死結發生。
- 讓管理及系統設計較為簡單容易。

MVCC 為 PostgreSQL 核心功能之一，無法將該功能關閉，身為 DBA 應該了解 MVCC 如何運作，也因以上的優點，當然也得付出待價，，因為須必保留 Dead Row Version，就算執行 Update 也會造成空間，Delete 或執行 Insert 到一半中止動作，仍佔用空間，須執行空間回收才能將這些資料清除，這也是為什麼要定期執行 VACUUM 回收空間的原因，回收空間有時可能會花費太多時間而影響效能，故須事前評估規劃，此動作也稱為 Defragmentation、Shrinking、Reorganization 或 Table Optimization。

```
postgres=# SELECT
nspname,relname,
round(100 * pg_relation_size(indexrelid) /
pg_relation_size(indrelid)) / 100
AS index_ratio,
pg_size_pretty(pg_relation_size(indexrelid))
AS index_size,
pg_size_pretty(pg_relation_size(indrelid))
AS table_size
FROM pg_index I
LEFT JOIN pg_class C ON (C.oid = I.indexrelid)
LEFT JOIN pg_namespace N ON (N.oid = C.relnamespace)
WHERE
nspname NOT IN ('pg_catalog', 'information_schema', 'pg_toast') AND
C.relkind='i' AND
pg_relation_size(indrelid) > 0;
```

管理索引

當使用者查詢時透過索引可縮短查詢時間進而提高效能，然而如果是偏重 DML 行為如 Insert、Update 及 Delete 的應用系統時可是會影響資料庫效能的喔！故過多索引未必會提高整體資料庫效能反而造成資料庫瓶頸；Autovacuum 並不會對索引進行維護或重建，故本章帶領大家如何管理索引。

準備開始

PostgreSQL 提供一工具名為「reindexdb」，透過作業系統執行此工具將重建指定資料庫所有索引：

```
# reindexdb dbname
```

如果想針對資料庫伺服器內所有資料庫索引進行重建則可執行以下命令：

```
# reindexdb -a
```

一般而言建立或重建索引時須非常小心，因執行此動作時會造成資料表鎖定(AccessExclusiveLock)，尤其當資料表很大時可能會持續好幾個小時，故非必要無須重建索引，如逼不得已重建，也須事前規劃時間。

以下範例，將引導大家藉由資料表 tb1 建立兩個索引，一個為 Primary Key，另一個 name 欄位索引。

建立資料表 tb1 及索引：

```
postgres=# CREATE TABLE tb1 (id      int,
                            name   varchar(10),
                            PRIMARY KEY  (id));
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "tb1_pkey"
for table "tb1"
CREATE TABLE
postgres=#  CREATE INDEX ON tb1(name);
CREATE INDEX
```

執行以下 SQL 語法查詢資料表 tb1 中所有索引：

```
postgres=# SELECT oid, relname, relfilenode
FROM pg_class
WHERE oid in (SELECT indexrelid
FROM pg_index
WHERE indrelid = 'tb1'::regclass);
  oid  |   relname    | relfilenode
-------+--------------+-------------
 17545 | tb1_name_idx |       17545
 17543 | tb1_pkey     |       17543
```

PostgreSQL 支援指令 CREATE INDEX CONCURRENTLY 讓在建立索引時不造成資料表的鎖定，也支援建立二個由同一個欄位產生的索引，之後可再重新命名索引名稱。

建立索引時，不造成資料表鎖定：

```
postgres=# CREATE INDEX CONCURRENTLY new_index ON tb1(name);
CREATE INDEX
```

重新命名索引名稱：

```
ALTER INDEX new_index RENAME TO tbs1_idx_name;
```

建立第二個由欄位 id 所產生的索引：

```
postgres=# CREATE UNIQUE INDEX new_pkey ON tb1(id);
CREATE INDEX
```

查詢資料表 tb1 現在所有索引，注意 tb1_pkey 及 new_pkey 皆由欄位 id 所產生的索引：

```
postgres=# SELECT oid, relname, relfilenode
FROM pg_class
WHERE oid in (SELECT indexrelid
FROM pg_index
WHERE indrelid = 'tb1'::regclass);
  oid  |    relname     | relfilenode
-------+---------------+-------------
 17543 | tb1_pkey      |       17543
 17547 | new_pkey      |       17547
 17545 | tb1_name_idx  |       17545
 17546 | tbs1_idx_name |       17546
(4 rows)
```

可執行以下 SQL 語法在透過更改 relfilenode 值在不影響關聯性限制下交換索引內容：

```
postgres=# BEGIN;
          UPDATE pg_class SET relfilenode=17543 WHERE oid = 17547;
          UPDATE pg_class SET relfilenode=17547 WHERE oid = 17543;
          COMMIT;
```

再次查詢資料表 tb1 現在所有索引，relfilenode 已更改：

```
postgres=# SELECT oid, relname, relfilenode
FROM pg_class
WHERE oid in (SELECT indexrelid
FROM pg_index
WHERE indrelid = 'tb1'::regclass);
  oid  |    relname     | relfilenode
-------+---------------+-------------
 17543 | tb1_pkey      |       17547
 17547 | new_pkey      |       17543
 17545 | tb1_name_idx  |       17545
 17546 | tbs1_idx_name |       17546
```

如何運作…

使用 CREATE INDEX CONCURRENTLY 在建立索引時，同時允許資料表進行 Insert、Update 及 Delete，但一次只允許 Transaction 期間只有一個 Session 可執行它；上述中有介紹透過更改 relfilenode 來達到置換索引，但盡量不要使用它，避免造成困惑而導致錯誤。

另外，PostgreSQL 在資料表刪除資料列時並不會同時刪除索引中該資料列，只是對該資料列進行標記，唯有資料區塊佔滿空間時，才會移除之前標記的資料列。

執行指令 CREATE INDEX CONCURRENTLY 建立索引期間，會有兩個步驟，首先，建立索引期間並將該索引狀態標記改為 Invalid，待該索引建立完成後再將索引狀態標記改為 Valid，查詢才會開始使用此索引；故在下面「刪除未使用索引」章節將介紹如何透過此方式快速更改索引狀態，避免因誤刪索引而造成資料庫不必要的傷害。

找出未使用索引

在查詢時選擇正確索引是一個重大議題，因為對的索引會加快查詢的速度，相反的錯的索引反而造成愈慢，故對一個 DBA 或開發人員而言，如何找出適當索引是非常重要。

如何做…

PostgreSQL 會定期追蹤資料庫中所有索引存取記錄，故可透過以下 SQL 語法查詢出哪些索引未使用過：

```
postgres=# SELECT schemaname, relname, indexrelname, idx_scan FROM
pg_stat_user_indexes ORDER BY idx_scan;
 schemaname | relname | indexrelname  | idx_scan
------------+---------+---------------+----------
 public     | tb1     | tb1_name_idx  |        0
 public     | tb1     | tbs1_idx_name |        0
 public     | tb1     | tb1_pkey      |    23140
 public     | tb1     | new_pkey      |    12341
(4 rows)
```

藉由以上結果，可找出 tb1_name_idx 與 tbs1_idx_name 索引從未被使用過，身為 DBA 就須考量是否該將未使用過的索引移除來增加資料庫效能。

刪除未使用索引

刪除索引時得非常的小心，因為一旦不小心刪錯索引，可能得花費很多時間重建，所以刪除前須先思考該索引是不是真得用不到，免得後悔莫極呢！

為了避免不小心刪除錯的索引，可藉由修改系統資料表 pg_index 中 indisvalid 欄位將索引停用，待確認正確無誤後再將索引刪除，由於 pg_index 並無提供索引名稱欄位，須結合系統資料表 pg_class，較為複雜，特地寫成 Function 讓大家方便執行：

建立停用索引 Function：

```
postgres=# CREATE OR REPLACE FUNCTION disable_index(iname TEXT)
RETURNS VOID
LANGUAGE SQL AS $$
UPDATE pg_index
SET indisvalid = false
WHERE indexrelid = $1::regclass;
$$;
CREATE FUNCTION
```

建立啟用索引 Function：

```
postgres=# CREATE OR REPLACE FUNCTION enable_index(iname TEXT)
RETURNS VOID
LANGUAGE SQL AS $$
UPDATE pg_index
SET indisvalid = true
WHERE indexrelid = $1::regclass;
$$;
CREATE FUNCTION
```

再執行剛建立的兩個 Function 來啟用或停用索引，以下為停用索引 tb1_name_idx 的 SQL 語法：

```
postgres=#  SELECT disable_index('tb1_name_idx');
```

接著，再查詢索引狀態已變成為 INVALID：

```
postgres=# \d+ tb1
                        Table "public.tb1"
 Column |         Type          | Modifiers | Storage  | Description
--------+-----------------------+-----------+----------+-------------
 id     | integer               | not null  | plain    |
 name   | character varying(10) |           | extended |
Indexes:
    "tb1_pkey" PRIMARY KEY, btree (id)
    "new_pkey" UNIQUE, btree (id)
    "tb1_name_idx" btree (name) INVALID
Has OIDs: no
```

當索引狀態改為 INVALID 後，並無影響到現有資料庫運作，待確認索引已不再須需要，再刪除該索引即可：

```
postgres=# DROP INDEX INDEX_NAME;
```

根據以上這種方式，可避免我們因誤刪索引導致在重新建立花費時間及效能影響，尤其在索引非常大時可能因重建索引而影響整體資料庫運作。

規劃資料庫維護

愈是愈重要的資料庫愈是愈繁忙，一般人總是有種心態，除非不得以，資料庫發生問題，否則不要去碰它、修改它，免得愈搞愈遭，然而這種心態是不好的，只是逃避無法改善資料庫運作，定期進行資料庫維護方能使整個運作更平順。

監控資料庫效能及健康狀態是個不錯的方式可即時接收最新資料庫狀況，但也只能在突發事件發生時可以快速解決問題，就其根本只是治標不治本，最好的方式應該是如何規劃資料庫維護。

如何做…

- 開始動手做：
  如何您還未備份公司中的資料庫，開始著手去規劃備份，並實作當災難發生時，如何快速還原。
- 善用您的時間：
  定期執行資料庫維護動作，如備份、重建索引 Autocuum…等，您可使用作業系統排程幫助達成自動化，避免增加工作負擔，但務必去執行。
- 對自己有信心：
  資料庫對公司而言是相當重要的資產，當然對於任何動作皆小心執行當然很重要，但可不要過於害怕，對自己沒信心，把所有事情想成太糟。

如何運作…

建立定期資料庫維護動作：

- 資料庫空間規劃：
  定期追蹤資料庫成長量，了解現有 Storage 空間依資料庫成長量預估何時空間滿載，避免空間不足時無法即時解決，畢竟現實環境中採購先硬體或增加硬碟並非那麼容易。
- 備份、還原測試及災難復原演練：
  檢查資料庫定期備份是否正常執行且完善，事先執行還原測試避免當資料庫毀損時，發生之前備份無法倒回，備而無用，最好也能規劃一套災難復原計劃與演練。

- Vacuum 及索引維護：
  執行 Vacuum 及 Analyze 定期更新資料庫統計值，
- 資料庫日誌分析：
  定期及持續分析資料庫所產生的日誌檔，以便即時發現問題立即處理。
- 資料庫安全及入侵偵測分析：
  確保資料庫未被入侵及資料庫是否處於安全環境。
- 了解您所負責的所有資料庫：
  唯有充份了解自己所負責的各資料庫，才能夠進行維護及調校。
- 資料庫效能分析：
  或許資料庫效能分析很漫長也無法立干見影，但唯有隨時進行監控效能，當發資料庫變緩慢時才能夠了解發生何種狀況進行分析改善。

或許定期執行這些資料庫例行性維護相當無聊且煩鎖，但唯有持續不間斷的維護，才能在資料庫發生異常時可以立即隨機應變，這也是身為一個 DBA 該有的責任，您說不是嗎？

CHAPTER 10

效能 & 並行處理

本章內容大綱如下：

- 找出效率差的 SQL 語法
- 定期收集資料庫統計值
- 找出 SQL 執行慢的原因
- 查詢時減少資料量傳回
- 將複雜 SQL 語法簡化
- 不使用 Rewriting 加速查詢
- 查詢不使用索引原因
- 強制查詢使用索引
- 最佳化鎖定
- 回報效能問題

前言

「並行處理」指的是資料庫伺服器可以同時服務幾個使用者查詢或 DML 動作，效能與資料庫並行處理緊緊相關聯，當並行處理量愈多時資料庫伺服器效能愈差，相反的解決了資料庫伺服器效能問題，並行處理量也就相對愈多。

本章節將教導大家如何找出查詢速度較慢的查詢，以及找出何種原因導致查詢速度慢。

定期收集資料庫統計值

http://pgstatspack.projects.postgresql.org/

```
# tar zxvf pgstatspack_version_2.2.tar.gz
pgstatspack/
pgstatspack/upgrade/
pgstatspack/remove_pgstats.sh
pgstatspack/sql/
pgstatspack/sql/pgstatspack_delete_old_stats.sql
pgstatspack/sql/pgstatspack_create_snap.sql
pgstatspack/sql/pgstatspack_remove_tables.sql
pgstatspack/sql/db_name.sql
pgstatspack/sql/pgstatspack_create_tables.sql
pgstatspack/sql/pgstats_exist.sql
pgstatspack/pgstatspack_sample_report.txt
pgstatspack/pgstatspack_README.txt
pgstatspack/install_pgstats.sh
pgstatspack/upgrade_pgstatspack.sh
pgstatspack/etc/
pgstatspack/bin/
pgstatspack/bin/pgstatspack_report.sh
pgstatspack/bin/delete_snapshot.sh
pgstatspack/bin/snapshot.sh
pgstatspack/pgstatspack_stat_explanation.txt
```

```
# cd pgstatspack
# ./install_pgstats.sh
```

```
# cd bin
# ./snapshot.sh
~/pgstatspack/bin ~/pgstatspack/bin
Password:
Results for database newdb
Password:
 pgstatspack_snap
------------------
               1
```

```
(1 row)
```

```
#  ./pgstatspack_report.sh
~/pgstatspack/bin ~/pgstatspack/bin
Please specify a username:
postgres
Password for user postgres:
List of available databases:

1 .  a
2 .  newdb
3 .  foo

Please select a number from the above list [ 1 -  3 ] : 2

Password for user postgres:
Password for user postgres:
 snapid |             ts             |    description
--------+----------------------------+---------------------
      2 | 2011-09-13 23:32:28.408399 | cron based snapshot
      1 | 2011-09-13 23:31:29.102267 | cron based snapshot
(2 rows)
Enter start snapshot id : 1
Enter stop snapshot id  : 2
....(略)
/tmp/pgstatreport_1_2.txt
#######################################################################
####################################
PGStatspack version 0.3 by frits.hoogland@interaccess.nl
#######################################################################
####################################

Snapshot information
Begin snapshot :
 snapid |             ts             |    description
--------+----------------------------+---------------------
      1 | 2011-09-13 23:31:29.102267 | cron based snapshot
(1 row)
```

```
End snapshot   :
 snapid |             ts             |    description
--------+----------------------------+---------------------
      2 | 2011-09-13 23:32:28.408399 | cron based snapshot
End snapshot   :
 snapid |             ts             |    description
--------+----------------------------+---------------------
      2 | 2011-09-13 23:32:28.408399 | cron based snapshot
(1 row)

Seconds in snapshot:  59.306132



Database version
                                                   version
-----------------------------------------------------------------------
-------------------------------------------------
 EnterpriseDB 9.0.4.14 on x86_64-unknown-linux-gnu, compiled by GCC gcc (GCC)
4.1.2 20080704 (Red Hat 4.1.2-50), 64-bit
(1 row)

 current_database | dbsize
------------------+---------
 newdb            | 7421 kB
(1 row)



Database statistics
 database  | tps  | hitrate | lio_ps | pio_ps | rollbk_ps
-----------+------+---------+--------+--------+-----------
 postgres  | 0.17 |   99.00 |   3.27 |   0.00 |      0.00
 a         | 0.10 |   99.00 |  48.04 |   0.10 |      0.00
 newdb     | 0.10 |   99.00 |  46.12 |   0.42 |      0.00
 foo       | 0.10 |   99.00 |  46.50 |   0.42 |      0.00
 template1 | 0.00 |    0.00 |   0.00 |   0.00 |      0.00
 template0 | 0.00 |    0.00 |   0.00 |   0.00 |      0.00
(6 rows)
Top 20 tables ordered by table size changes
```

```
...(略)
Top 20 tables ordered by high table to index read ratio
...(略)
Top 20 tables ordered by inserts
...(略)
Top 20 tables ordered by updates
...(略)
Top 20 tables ordered by deletes
...(略)
Tables ordered by percentage of tuples scanned
...(略)
Indexes ordered by scans
...(略)
Sequences ordered by blks_read
...(略)
Top 20 SQL statements ordered by total_time
...(略)
Parameters
...(略)
```

找出 SQL 執行慢的原因

造成 SQL 執行慢的原因有很多種情況，以下將會列出幾種情況說明：

母體資料太多：

於 SQL 語法前面加入 explain analyze，列出資料處理情形：

```
postgres=# explain analyze select count(*) from tb1;
                                              QUERY PLAN
------------------------------------------------------------------------------------
--------------------
 Aggregate  (cost=24.12..24.14 rows=1 width=0) (actual time=0.028..0.028 rows=1
loops=1)
   ->  Seq Scan on tb1  (cost=0.00..21.30 rows=1130 width=0) (actual time=0.001..0.001
rows=0 loops=1)
 Total runtime: 0.369 ms
(3 rows)
```

根據以上結果將顯示處理多少資料列、讀取多少資料區塊及執行花費多少時間，藉由比較這些差異來了解查詢對資料庫所需的負擔。

當然如欲了解更詳細資訊，可查詢系統資料表 pg_stat*，指定資料表來查看內容：

```
postgres=# SELECT s.relid, s.schemaname, s.relname,
              seq_scan,seq_tup_read,
              idx_scan,idx_tup_fetch,
              heap_blks_read,heap_blks_hit,
              idx_blks_read,idx_blks_hit,
              toast_blks_read,toast_blks_hit
          FROM pg_stat_user_tables s
          JOIN pg_statio_user_tables sio
          ON s.relid = sio.relid
          WHERE S.SCHEMANAME = 'public' AND s.relname = 'tb1' ;
```

以下範例將建立一資料表名為 events，並照順序塞入十萬筆資料列，分別比較建立索引前後，查詢該資料表前三筆資料列所花費的時間及資料庫影響效能。

建立資料為 events，查詢前三筆資料列的執行計劃：

```
postgres=# CREATE TABLE events(id serial);
postgres=# INSERT INTO events SELECT generate_series(1,100000);
postgres=# explain analyse SELECT  * FROM events ORDER BY ID DESC LIMIT 3;
                                                   QUERY PLAN
---------------------------------------------------------------------------------
---------------------------------------
 Limit  (cost=2735.48..2735.49 rows=3 width=4) (actual time=29.034..29.035 rows=3
loops=1)
   ->  Sort  (cost=2735.48..2985.48 rows=100000 width=4) (actual time=29.032..29.032
rows=3 loops=1)
        Sort Key: id
        Sort Method: top-N heapsort  Memory: 25kB
        ->  Seq Scan on events  (cost=0.00..1443.00 rows=100000 width=4) (actual
time=0.009..12.402 rows=100000 loops=1)
 Total runtime: 29.060 ms
(6 rows)
```

使用欄位 id 建立索引名稱為 events_id_idx，再次查詢前三筆資料列的執行計劃：

```
postgres=# CREATE INDEX events_id_idx ON events(id);
postgres=# explain analyse SELECT  * FROM events ORDER BY ID DESC LIMIT 3;
                                                           QUERY PLAN
---------------------------------------------------------------------------------
---------------------------------------------------------
 Limit  (cost=0.00..0.09 rows=3 width=4) (actual time=0.099..0.117 rows=3 loops=1)
   ->  Index Scan Backward using events_id_idx on events  (cost=0.00..3050.26
rows=100000 width=4) (actual time=0.098..0.100 rows=3 loops=1)
 Total runtime: 0.157 ms
(3 rows)
```

根據以上比較，可發現 Cost 值、資料列回傳值 Rows 及執行所需時間，差異極大，故得知當查詢時利用索引可大幅減少母體資料，效能當然較好。

共享記憶體命中率太低：

當共享記憶體設定值太低時，加上使用者所查詢資料重複性不高，導致資料庫伺服器須讀取實體硬碟資料並放置共享記憶體，此動作將大幅降低效能；我們可藉由系統資料表 pg_stat*，查詢 heap_blks_read、idx_blks_read、toast_blks_read 來判斷是否共享記憶體命中率太低。

另外或許會讓人有所困惑，當使用者第一次執行某 SQL 語法查詢時與第二次執行查詢，發現*_blks_read 欄位值差異甚大，此乃正常現象，因當使用者第一次執行該 SQL 語法後，已將所需資料載入至共享記憶體，當然第二次查詢時會由共享記憶體讀取，所以*_blks_read 欄位值當然會增加許多。

回傳資料列筆數太大：

在某些情況中程式設計師撰寫系統時可能偷懶或不小心，一次回傳全部資料列筆數，對於資料只有百筆，少量資料當然不受什麼影響，但當資料量幾千萬筆時，千萬不要一次查詢所有資料列，不僅前端系統無法一次處理那麼多回傳資料列，也造成資料庫伺服器及網路的負擔，應採分頁方式分批將資料列呈現，如此效能方能提昇。

鎖定問題：

僅管 MVCC 機制以大幅減少 PostgreSQL 遭遇鎖定問題，如允許當資料庫寫入時也能執行查詢，允許查詢時也能執行資料庫寫入；然而如兩個以上使用者同時執行 DML 寫入資料庫，鎖定也無法避免，唯有執行 DML 動作的使用者該交易結束後方能再執行下一個 DML 動作。

可執行以下 SQL 語法查詢哪個使用者或資料表正處於值鎖定：

```
postgres=# SELECT * FROM pg_locks WHERE not granted;
```

可進階查詢哪些 SQL 語法導致哪些資料表鎖定：

```
postges=# SELECT
  a1.current_query as blocking_query,
  a2.current_query as waiting_query,
  t.schemaname ||'.'||t.relname as locked_table
FROM pg_stat_activity a1
JOIN pg_locks p1
   ON a1.procpid = p1.pid and p1.granted
JOIN pg_locks p2
   ON p1.relation = p2.relation and not p2.granted
JOIN pg_stat_activity a2
   ON a2.procpid = p2.pid
JOIN pg_stat_all_tables t
   ON p1.relation = t.relid;
blocking_query | waiting_query | locked_table
-----------------------+------------------------+-------------
<IDLE> in transaction  | select * from tb1;       | public.tb1
<IDLE> in transaction  | select count(*) from event;  | public.event
(2 rows)
```

CPU 資源與硬碟 I/O 效能不足：

當然硬體規格絕對會影響資料庫伺服器效能，一般而言資料庫瓶頸不是 CPU 就是硬碟 I/O 了(CPU-Bound 與 I/O-Bound)，可使用 top 或 vmstat 指令監控系統效能，本書並未著重系統監控，故不詳細介紹囉！

減少資料列回傳筆數

有時候使用者查詢資料量太大，該資料列卻不需全部用到，造成資料庫伺服器與要傳輸回傳大量資料往客戶端，此方式除了造成伺服器效能差外也影響彼此網路頻寬。

以下介紹幾個方法來減少資料庫伺服器與客戶端網路間流量。

查詢回傳資料列 10,000 筆而真正需要的資料列只有前 20 筆：

正確 SQL 語法應如下：

```
postgres=# SELECT * FROM events LIMIT 20;
```

假設需要查詢下 20 筆資料列，不需使用 LIMIT 40 來限制回應 40 筆資料列，可改用「OFFSET 20 LIMIT 20」來呈現。

```
postgres=# SELECT * FROM events
postgres-# OFFSET 20 LIMIT 20;
```

如回傳資料需運算分析，應由資料庫伺服器運算後再傳回：

藉由此方式可減少資料庫伺服器回傳資料，

較差做法是，將所要資料全部列出傳回使用者端，再處理資料：

```
postgres=# SELECT * FROM events WHERE id = 10;
```

較好做法是，一次由資料庫處理完畢後再所需將資料傳回：

```
postgres=# SELECT COUNT(*), SUM(salary) FROM events WHERE id = 10;
```

如資料庫結構化語言(SQL)無法滿足處理資料需求時，則可採用資料庫程式語言如 PL/pgSQL 或其它嵌入式程序語言如 PL/Tcl、PL/Perl、PL/Python。

[redacted]

未使用索引原因

本章節解釋為何查詢未使用索引，主要有幾個原因，比較常發生優化器(Optimizer)誤判執行計劃不使用索引時查詢較快且花費成本較低。

如何做…

強制使用索引

資料表鎖定

SELECT 的讀取鎖定主要分為以下二大類：

- SELECT ... LOCK IN SHARE MODE

```
postgres=# BEGIN WORK;
LOCK TABLE products IN SHARE MODE;
UPDATE products SET quantity = 10 WHERE id=2 ;
COMMIT WORK;
```

- SELECT ... FOR UPDATE

```
postgres=# BEGIN WORK;
SELECT quantity FROM products WHERE id=3 FOR UPDATE;
UPDATE products SET quantity = 10 WHERE id=3 ;
COMMIT WORK;
```

此二種方式進行交易，當 SELECT 到同一個資料表時，必須等待它方交易資料被確認後(Commit)才能夠執行。主要的不同在於 LOCK IN SHARE MODE 很容易造成 Dead Lock 。

所以 SELECT 後面若要繼續 UPDATE 同一資料表，最好使用 SELECT ... FOR UPDATE，但為何要鎖定資料表呢？為什麼不直接使用以 SQL 下語法就好了：

```
postgres=# SELECT quantity FROM products WHERE id=3;
postgres=# UPDATE products SET quantity = 1 WHERE id=3;
```

基本上當少量交易時或許不會有什麼大問題，但如果同時有大量資料交易時絕對會出問題，這是為什麼呢？

舉以上的例子來說，products 為商品資料表，id 為商品編號而 quantity 為商品數量，而 quantity 商品數量在大於 0 也就是有庫存商品時才能進行扣除，然而現實情況中並非只有一個交易，同時還會有其它交易進行，第一次執行 SELECT 時或許 quantity 商品數量等於 2，而下達 UPDATE 時商品庫存數量已被扣為 0，此時 UPDATE 再進行扣除，就導致商品庫存數量為負數，就發生問題了。

最佳化鎖定

CHAPTER 11

備份 & 還原

本章內容大綱如下：

- 前言
- 了解資料庫復原
- 邏輯熱備份單一資料庫
- 邏輯熱備份全部資料庫
- 邏輯熱備份單一表格空間
- 單一資料庫熱備份
- 備份資料庫物件
- 還原全部資料庫
- 單一時間點還原

前言

備份對資料庫安全是必要的，因此需要花費一些時間規劃、執行備份是比維護資料庫更加的重要，畢竟沒有完整備份，當資料庫發生毀損時會造成不可恢復的災難。

本章節將會帶領大家當資料庫發生 Crash 時該如何處理，能夠找出問題所在，必要時將資料庫復原。

資料庫復原

資料庫自動復原(Recovery)機制，可在資料庫發生 Crash 或作業系統發生 Crash 時將資料庫回復避免資料庫造旁毀損而無法開啟。

當 PostgreSQL 發生 Crash 時除了會在日誌檔寫入 PANIC 等級錯誤訊息外，PostgreSQL 會自動重啟並使用交易記錄日誌檔或 WAL(Write Ahead Log)將資料庫復原。

WAL 全名(Write-Ahead Logging)是以一連續數字為檔名存放於資料目錄(PGDATA)的子目錄 pg_xlog 中，當資料庫因交易行為發生異動時，會將每筆記錄寫入至 WAL，當交易執行 Commit 時會將 WAL 裏的資料寫入硬碟。PostgreSQL 發生 Crash 時，資料庫會重新執行 WAL 裏的內容，讓資料庫回到 Crash 前最後一次 Commit 的時間點以確保資料完整性。

注意！當執行交易 Commit 時資料庫並未將異動資料寫入至硬碟喔！會後續由「Background Writer」執行程序透過演算機制定期寫入。

上面有提到當資料庫 Crash 時，資料庫會重新執行 WAL 裏的內容，但究竟從哪個時間點開始執行呢？復原是從 WAL 所知道的檢查點(Checkpoints)開始一直到發生 Crash 前最後一次檢查點(Checkpoints)。

預設時在交易日誌中每隔一段時間執行檢查點，而當資料庫本身運作已忙錄時，檢查點(Checkpoints)可能會造成資料庫的瓶頸，因為檢查點(Checkpoints)會將資料庫所有發生的異動同時寫入至硬碟，以下會介紹幾種調效的方式..

有二個資料庫參數可控制執行檢查點(Checkpoints)時多少量的 WAL 會被寫入硬碟；第一個參數為 checkpoint_setments，意指控制 n 個 16MB 的 WAL 產生時會觸發檢查點(Checkpoints)；第二個參數為 checkpoint_timeout，意指 n 秒後會觸發檢查點(Checkpoints)，以上二個參數不管哪一條件符合都會觸發。

舉個以下例子來講吧！將此二個參數設定為：

checkpoint_segments = 1000

checkpoint_timeout = 3600

那有沒有思考過？當資料庫發生 Crash 時要發多久時間復原呢？

另外必須確保 pg_xlog 目錄空間是足夠的，依照這個例子所需要空間是 3 x 16MB x checkpoint_segments，也就是說為這個參數 checkpoint_segments = 1000 至少需要 32GB 空間，而假設又設定參數 wal_keep_segments > 0，則空間將會達到 16MB X (wal_keep_segments + checkpoint_segments)。

如何運作…

資料庫復原動作會持續進行執行交易記錄重新執行完畢，此機制是一直連續性寫入直到最後一筆記錄故不須定義結束時間點。每個 WAL 記錄均透過循環冗餘檢查碼(CRC)檢查，每筆記錄又包含指標指到前筆記錄，便了解這一連串記錄是否可用故 PostgreSQL 知道 WAL 中哪些記錄是否可用於重新執行，而資料庫復原會持續至讀取到錯誤的 WAL 記錄。

資料庫復原所需的時間是與要執行哪些復原動作有關的，而測試資料庫復原的效能最好的方式就是建立 Standby 複寫資料庫，將會於資料庫複寫中介紹。

當資料庫進行復原時重新執行交易記錄是有可能發生問題而導致資料庫無法順利開啟，此時可以使用 pg_resetxlog 工具強迫將資料庫開啟，執行之後它會清除所有的資料庫日志檔也就是 pg_xlog 中的所有 WAL，並開始寫入新的 WAL。

pg_resetxlog 會導致資料庫不完全復原，也就是說資料遺失，非不得以需要執行此工具建議先將 pg_xlog 內所有 WAL 先行備份，而個人建議如遇資料庫復原時遇到問題而無法將資料庫開啟，最好尋求專業 DBA 幫忙，因為我們不知道或許你現在解決了這個問題，但可能會延伸出更多的問題來。

WAL 空間大小一定是 16MB，僅管使用 initdb 指令建立新資料庫實例也是無法改變此設定值；除非你是程式高手，可透過 PostgreSQL 安裝原始檔，修改其源始碼並重新編譯才能改變設定值。

規劃資料庫備份

本章節將告訴大家如何規劃備份資料庫，主要重點在於了解規劃資料庫如何復原而不是備份，備份類型將影響日後資料庫 Crash 時如何復原，所以得事先規劃使用何種方式；也就是說規劃資料庫復原計劃時，事先考量資料庫發生異常時可能遭遇哪幾種類型錯誤，而適情況執行何種方式復原？

我們需考量以下幾種方面：

- ✧ 完整或部份資料庫
- ✧ 所有物件內容或僅物件定義
- ✧ 單一時間點回復
- ✧ 資料庫復原效能

接著了解不同工具程式所提供的備份方式，找出當資料庫 Crash 時最適何最有效益的復原方式。

PostgreSQL 主要備份方式如下：

- ✧ 使用 pg_dump 工具執行邏輯備份。
- ✧ 使用檔案系統執行實體備份。

備份/復原 比較表表格：

	pg_dump -F c 匯出資料庫傾印檔	pg_dump -F p 或 pg_dumpall 匯出資料庫傾印檔	pg_start_backup 備份檔案系統
備份類型	邏輯	邏輯	實體
單一時間點回復	否	否	是
備份所有資料庫	一次只能單一資料庫	是(pg_dumpall)	是
同時間備份所有資料庫	否	否	是
備份特定物件	是	可	否(註 3)
遞增式備份	否	否	可(註 4)
回復特定物件	是	可(註 1)	否(註 5)
特定資料表回復	是	是	可(註 6)
特定表格空間回復	可(註 2)	可(註 6)	可(註 6)
壓縮備份	是	是	是
備份檔為多檔案	否	否	是
平行備份(Parallel)	否	否	是
平行回復(Parallel)	否	否	是
將備份回復於最新資料庫版本	是	是	否
Standalone 備份	是	是	是(註 7)
備份時允許 DDL	否	否	是

(資料庫傾印檔為 SQL DUMP)

註解：

1. 使用 pg_dump 或 pg_dumpall 產生資料庫傾印檔時，假設只需還原單一特定物件，可透過程式語言如 Perl 萃取部份傾印檔內容，雖然此方式會花費許多時間且麻煩，但比起將全部備份回復至另一資料庫伺服器再使用 pg_dump 傾印所需物件的方式所需時間來的少。
2. 可參閱復原表格空間章節。
3. 透過實體備份方式來備份特定物件是有可能的，雖然此方式在還原時可能會遇到一些狀況，可參閱復原表格空間章節。
4. 可參閱增遞式/差異備份。
5. PostgreSQL 中使用實體備份方式來回復特定物件是有困難的。
6. 可參閱復原表格空間章節。
7. 可參閱熱備份資料庫章節。

如何做…

備份資料庫伺服器中的所有資料庫，可能想到使用 pg_dumpall，但建議不要使用這工具，主要有以下四個原因：

- 使用 pg_dumpall 只能傾印單一備份檔，無法同時執行 pg_restore 來平行復原資料庫，故此復原方式較慢。
- pg_dumpall 只能一次傾印單一資料庫，且當一個資料庫備份完畢後才能進行下一個資料庫，也就是說：
  - pg_dumpall 所需時間會比同時執行多個 pg_dump 備份多個資料庫來的慢。
  - pg_dumpall 產生的備份檔，各別資料庫中的狀態，時間是不一致的；假設我們開始執行備份於 04:00，完成備份於 07:00，我們是無法確認各資料庫完成備份時間。
- 雖然 pg_dumpall 與 pg_dump 很相似，但某此動作是無法達到；總結來說：pg_dumpall 備份效能較慢，復原也慢，也比較難掌握。這是不要使用 pg_dumpall 的主要原因，建議如有多個資料庫，一個一個資料庫備份比較好。
- pg_dumpall 傾印 global Information 如角色或表格空間，接著使用 pg_dump 分別備份各資料庫，要特別小心檢查是否有錯誤訊息發生，使用此方式不如用實體備份方式來取代。

邏輯備份單一資料庫

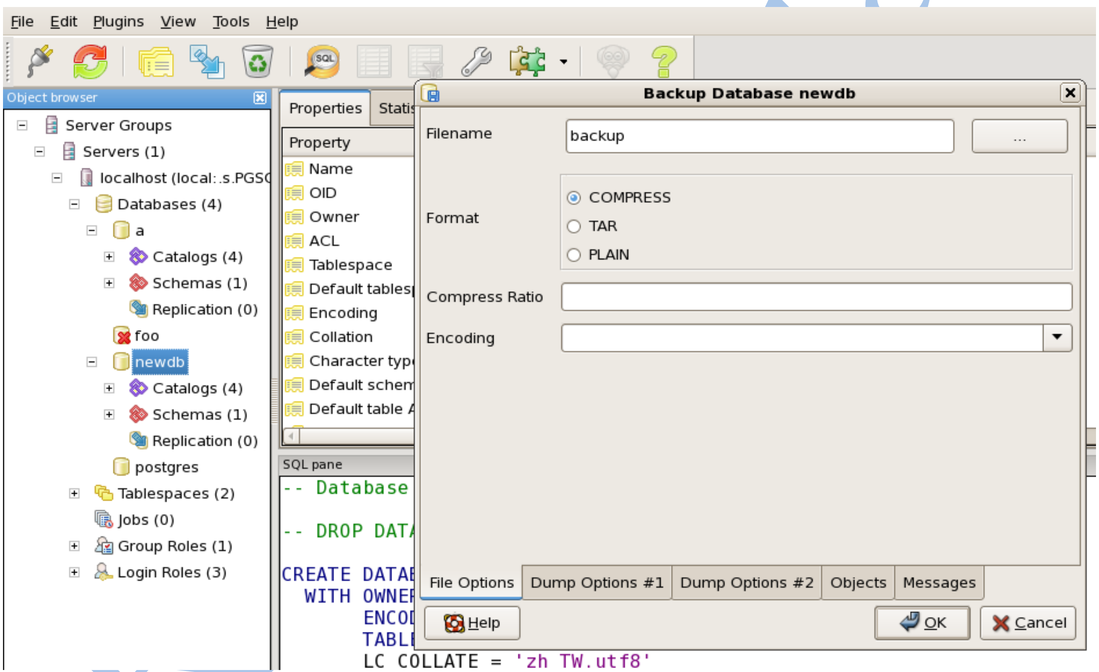
邏輯備份意指將資料庫中每張資料表傾印成備份檔。

如何做…

可執行指令如下：

```
# pg_dump -F c > dumpfile
# pg_dump -F -f > dumpfile
```

當然也可以使用圖型工具 pgadmin3 來執行：



pg_dump 傾印資料庫時會產生單一備份檔，當然備份檔可以適情況切割成多個備份集。

pg_dump 指定格式為 custom 時會將備份檔進行壓縮，可以依照自己需要將壓縮選項移除或提高壓縮率。

pg_dump 傾印檔內容為 SQL Script，當開始執行 pg_dump 時，備份內容為該時間點的資料庫快照，無法讓所有備份的資料庫快照保持在同一個時間點；備份快照也只能將資料庫復原至該時間點，無法復原資料庫備份至之前或之後的時間點；另外快照備份集是資料庫開始備份時間點並非備份結束的時間點喔！

當執行 pg_dump 時，會輕微鎖定資料表但只限於避免執行 DDL，使用參數-t，pg_dump 允許備份指定資料表、視觀表(view)或循序列號(sequence)，pg_dump 很早就在 PostgreSQL 釋出，是用來升級(migrate)資料庫不可缺少的工具。

pg_dump 是無法整合一些模組的如 Slony 或 PostGIS，因為它無法知道 PostgreSQL 已使用這些功能，只是單純傾印該物件而已，所以當復原備份檔至另一資料庫時仍須對該資料庫進行額外設定。

如何評估 pg_dump 傾印資料庫花費多少時間呢？可使用參數-v，該參數會記錄額外資訊至備份檔包含開始及結束時間。

> pg_dump 是無法傾印資料庫伺服器角色跟表格空間，須使用 pg_dumpall 才能進行備份，將會在後面章節詳細介紹喔！

邏輯備份全部資料庫

當 PostgreSQL 伺服器同時擁有一個以上資料庫時，你可能想一次備份所有資料庫，但我的建議是同時執行多個 pg_dump 分別備份資料庫，此方式可以加快備份速度，待備份完成後傾印資料庫伺服器 Global Information，指令如下：

```
# pg_dumpall -g
```

以上會備份資料庫伺服器角色(使用者)跟表格空間。

備份資料庫伺服器中的所有資料庫，你可能想到使用 pg_dumpall，但建議不要使用這工具，主要有以下四個原因：

- ✧ 使用 pg_dumpall 只能傾印單一備份檔，無法同時執行 pg_restore 來平行復原資料庫，故此復原方式較慢。
- ✧ pg_dumpall 只能一次傾印單一資料庫，且當一個資料庫備份完畢後才能進行下一個資料庫，也就是說：
  - ➢ pg_dumpall 所需時間會比同時執行多個 pg_dump 備份多個資料庫來的慢。
  - ➢ pg_dumpall 產生的備份檔，各別資料庫中的狀態，時間是不一致的；假設我們開始執行備份於 04:00，完成備份於 07:00，我們是無法確認各資料庫完成備份時間。
- ✧ 雖然 pg_dumpall 與 pg_dump 很相似，但某此動作是無法達到；總結來說：pg_dumpall 備份效能較慢，復原也慢，也比較難掌握。這是不要使用 pg_dumpall 的主要原因，建議如有多個資料庫，一個一個資料庫備份比較好。
- ✧ pg_dumpall 傾印 global Information 如角色或表格空間，接著使用 pg_dump 分別備份各資料庫，要特別小心檢查是否有錯誤訊息發生，使用此方式不如用實體備份方式來取代。

邏輯備份表格空間中所有資料表

在某此情況下我們可能需要備份特定表格空間中的資料表，很不幸性，並不是很簡單單一指令就能夠完成，故在此介紹 。

以下 SQL 語法，將會找出哪些資料庫中的資料表建立在特定表格空間：

```
postgres=# SELECT datname
          FROM pg_database
          WHERE oid IN(
                SELECT pg_tablespace_databases(ts.oid)
                FROM pg_tablespace ts
                WHERE spcname='tablespace_name');
```

此部驟列出特定表格空間中的所有資料表，建立一文字檔名為「onets.sql」SQL 語法如下：

```
SELECT 'pg_dump'
UNION ALL
SELECT '-t ' || spcname || '.' || relname
FROM pg_class t JOIN pg_tablespace ts
ON reltablespace = ts.oid AND spcname = :TSNAME
JOIN pg_namespace n ON n.oid = t.relnamespace
WHERE relkind = 'r'
UNION ALL
SELECT '-F c > dumpfile';
```

執行以下指令來產生 pg_dump 的 SQL Script：

```
# psql -t -v TSNAME="'tablespace_name'" -f onets.sql > ts_dump.sh
```

執行 ts_dump.sh 備份表格空間即可。

```
# chmod 755 ts_dump.sh
# ./ts_dump.sh
```

pg_dump 允許在執行指令時指定一個以上資料表，故可透過以上指令找出我們要備份的資料表並產生 SQL Script。

匯出資料庫物件結構

在某些情況中我們可能只想匯出資料庫所有結構定義檔，用於分析了解資料庫結構，也可用於將原資料庫的某些結構重新建立於新資料庫上，如使用者、資料表…等。

如何做…

如欲匯出所有物件包含使用者、表格空間、所有資料庫中資料表、索引、限制約束(Constraints)、觸發(Trigger)、視觀表(Views)、函數(Functions)及所有權限可使用以下指令：

```
pg_dumpall --schema-only > scriptdump.sql
```

如欲匯出使用者可使用以下指令：

```
pg_dumpall --role-only > role.sql
```

如欲匯出表格空間定義可使用以下指令：

```
pg_dumpall --tablespace-only > tsb.sql
```

如欲同時匯出使用者及表格空間可使用以下指令：

```
pt_dumpall --global-only > globals.sql
```

以上這些匯出的傾印備份檔，內容為純文字的 SQL 語法，是可解讀的，故可依需求更改內容或分析，且可重新執行於另一資料庫中。

「Schema」，一字於 PostgreSQL 可能與我們一般所認知想法有所不同，直覺上會覺得應該是資料表的結構，但於 pg_dumpall 中指的是資料庫實例中所有資料庫物件。

使用熱備份複製資料庫

對於資料庫軟體而言熱備份是一個很重要的功能。

熱備份讓我們可以同時獲得某一時間點所有資料庫中一致性資料，甚至此時正在執行 DDL 一樣可以進行熱備份不會造成負載過重或鎖定發生，實體檔案備份步驟會比邏輯備份複雜了一些，以下就進行資料庫熱備份來介紹吧。

接著帶領大家一個一個步驟使用熱備份複製資料庫：

1. 定義$PGDATA，資料目錄。
2. 定義$BACKUPNAME 備份檔名。
3. 所有於 PostgreSQL 存取相關資料庫參數。
4. 建立封存日誌目錄

```
# cd $PGDATA
# mkdir -p ../standalone
```

5. 設定參數 archive_command，

```
wal_level = archive
archive_mode = on
archive_command = 'test !
/opt/PostgresPlus/9.0AS/data/standalone/archiving_active || cp "%p"
/opt/PostgresPlus/9.0AS/data/archive/"%f"'
```

6. 開始產生封存日誌檔(Archiving)：

```
# mkdir -p $PGDATA/standalone/archive
# touch $PGDATA/standalone/archive/archiving_active
```

7. 進入熱備份模式：

```
# psql -c "SELECT pg_start_backup('standalone');"
```

8. 開始備份資料庫，但排除 pg_xlog 目錄

```
# tar -cv --exclude=$PGDATA/pg_xlog -f $BACKUPNAME $PGDATA
```

9. 結束備份：

```
# psql - c "SELECT pg_stop_backup(), current_timestamp"
```

10. 停止產生封存日誌檔(Archiving)：

```
# rm $PGDATA/standalone/archiving_active
```

11. 將剛熱備份模式期間所產生的封存日誌檔搬移至 archive 目錄：

```
# mv $PGDATA/standalone/archive $PGDATA/archive
```

12. 加入封存日誌檔至資料庫備份檔中：

```
# tar -rzf $BACKUPNAME $PGDATA/archive
```

13. 為了能進行復原資料庫故編輯 recovery.conf，如不存在需先建立該檔，指行以下指令：

```
#echo "restore_command = 'cp /opt/PostgresPlus/9.0AS/data/archive/%f %p'"
> recovery.conf
```

14. 再將 recovery.conf 檔加入至資料庫備份檔中：

```
# tar -rf $BACKUPNAME recovery.conf
# echo "recovery_end_command = 'rm -rf /opt/PostgresPlus/9.0AS/data/archive
'" >> recovery.conf
```

15. 將資料庫備份檔保存，如資料庫毀損時可進行復原。

如何運作…

當使用熱備份備份資庫時，日後可將資料庫復原至執行 pg_stop_backup 函數的時間點；如果僅執行作業系統方式備份資料庫如步驟 8，此方式是錯的喔！所備份的資料是不一致的，須結合步驟 7、9 才是正確的資料庫備份方式；在此熱備份模式下，交易記錄 WAL 會被放入至$PGDATA/standalone/archive/ 目錄下；步驟 6 建立封存日誌目錄，步驟 1、2、3、5 為設定參數包含啟用封存日誌模式及存放目錄；切記！啟用封存日誌模式得重啟資料庫實例喔！而 archiving_active 檔是用來判斷是否啟用封存日誌程序，該檔存在時啟用，不存在則停用如步驟 6 及 10，也就是說只有在資料庫進入熱備份啟用封存日誌模式。

為了當資料庫復原方便步驟 11 將 WAL 移至$PGDATA 下，步驟 12 則將 WAL 檔再打包入備份檔；步驟 13、14 是有關於資料庫復原時會用到的設定檔及參數，會於復原資料庫章節詳細說明。

此章節中主要重點是在執行資料庫熱備份時須啟用封存日誌模式，當還原(Restore)資料庫後仍需復原(Recovery)資料庫否則資料庫是無法正常開啟！一直持續啟用封存日誌模式，此方式允許資料庫回復於任何時間點，但本章節著重在使用熱備份複製資料庫，故並未一直啟用封存日誌，也可避免當使用熱備份複製資料時，哪些 WAL 才是資料庫復原(Recovery)時所需要的。

資料庫熱備份

接續上一章節「使用熱備份複製資料庫」後，本章節將教導大家還原(Restore)資料庫後，如何利用 WAL 將資料庫復原(Recovery)至某一時間點。

接續上章節，在此提出以下幾個問題讓大家考量：

- ✧ 封存日誌檔(WAL)存放位置？
- ✧ 如何複製封存日誌檔？
- ✧ 資料庫備份檔存放位置？
- ✧ 使用何種方式備份？
- ✧ 保存 n 份資料庫備份，封存日誌檔保存原則？

如何做…

假設以上問題答案如下：

- ✧ 封存日誌檔目錄位於遠端備份伺服器$OTHERNODE。
- ✧ 使用 SCP 複製封存日誌檔。
- ✧ 資料庫備份存放於遠端備份伺服器$OTHERNODE。
- ✧ 使用 rsync 備份資料庫。
- ✧ 只保存二份資料庫備份，當新備份產生時覆概原本備份檔且於每週星期日執行。

環境變數如下：

- ✧ $PGDATA，資料目錄。
- ✧ $OTHERNODE，遠端備份伺服器
- ✧ $BACKUPNAME，
- ✧ 任何有關 PostgreSQL 取存參數皆已設定。

接著帶領大家一個一個步驟熱備份資料庫：1.

✧ 於遠端備份伺服器建立封存日誌檔目錄及備份目錄。

```
# mkdir -p $PGDATA
# mkdir -p archive
```

✧ 設定參數 archive_command，設定完畢後需重啟資料庫：

```
archive_mode = on
archive_command = 'scp %p $OTHERNODE:/archive/%f'
```

✧ 進入熱備份模式：

```
# psql -c "SELECT pg_start_backup('backup')"
```

✧ 透過 rsync 複製資料庫資料，排除 pg_xlog 目錄：

```
rysnc -avc --inplace --exclude=*pg_xlog* $PGDATA/
$OTHERNODE:$BACKUPNAME/$PGDATA
```

✧ 結束熱備份模式：

```
# psql -c "SELECT pg_stop_backup(), current_timestamp"
```

如何運作…

進行資料庫備份時最重要的一點是除了基本資料庫備份外，封存日誌檔是不可豁缺的，如此方能完整復原資料庫並開始；當進入熱備份模式，可使用任何系統檔案備份方式如 rsync 或 cp…等。

許多管理工具都宣稱支援 PostgreSQL 備份，其實備份方式都屬邏輯備份，不僅速度慢又不安全；基本上最安全方式還是屬 PostgreSQL 熱備份搭配 rsync 莫屬，僅管備份時網路或備份伺服器發生異常造成 archive_command 失敗，備份會持續重試直到備份成功；PostgresSQL 並不會移除 pg_xlog 中的 WAL，直到 WAL 被複製成功，但卻可能造成 pg_xlog 目錄空間滿載，這可能會造成 PostgreSQL 因空間不足無法產生 WAL 而導致無法正常運作，所以監視系統可用空間大小，也是身為一個管理者須正視的課題。

啟用封存日誌模式時，會一直持續產生一連串的 WAL，假設將資料庫參數 archive_timeout 設定 30 秒(s)，則平均一天 WAL 最少產生 2 X 60 X 24 = 2880 個檔案，約為 46GB，但 WAL 可能不到 30 秒(s)就產生一個檔案，故 WAL 一天大概約佔 100GB，因此評估空間大小是很重要的喔！另外，進行資料庫備份時，判別哪些 WAL 是需要備份及哪些 WAL 是過期可以刪除也很重要！當我們執行 pg_start_backup('LABEL')時進入熱備份模式後，開始產生的 WAL 才是資料庫復原所需要的檔案，我們可使用額外模組稱為「pg_archivecleanup」，清除過期 WAL。

還原所有資料庫

如何還原資料庫伺服器中所有資料庫是一個很重要課題，本章節將教導大家如何輕易還原所有資料庫。

先前準備

首先，決定一台適何的伺服器提供資料庫還原。

在將備份還原至新資料庫伺服器前，應將原資料庫伺服器先行備份，避免在還原過程中發生任何問題而導致事情更糟。

如何做…

邏輯還原(dump -F c)：

✧ 匯入所有資料庫 Schema，須確認之前是否有正確備份檔。

```
# pg_restore --schmea-only -v dumpfile | head | grep Started
```

✧ 匯入 Global Informations，意指使用者及表格空間資訊。

```
# psql -f globals.sql
```

✧ 匯入所有資料庫資料，匯入工具 pg_restore 可指定同時 n 個執行程序平行執行，也支援遠端匯入；注意！匯入所有資料庫 Schema 後，得分別匯入各資料庫喔！

```
# pg_restore -d postgres -j 4 dumpfile
```

邏輯還原(dump -F p)：

✧ 還原(Restore)資料庫前，確認備份時間是否正確。

```
# head scriptdump.sql | grep Started
```

✧ 匯入 Global Information。

```
# psql -f global.sql
```

✧ 匯入資料庫資料。

```
# psql -f scriptdump.sql
```

邏輯還原(pg_dumpall)：

✧ 還原(Restore)資料庫前，確認備份時間是否正確。

```
# head scriptdump.sql | grep Started
```

✧ 選擇適合伺服器或建立虛擬伺服器，將資料庫還原於此。

✧ 匯入全部資料庫。

```
# psql -f scriptdump.sql
```

實體還原：

- ✧ 假設之前已使用上一章節使用熱備份複製資料庫，則還原資料庫是很簡單的。
- ✧ 解壓縮備份檔至新的$PGDATA。
- ✧ 在還原之前確認備份檔是否正確。

```
# cat backup_label
START WAL LOCATION: 0/3000020 (file 000000010000000000000003)
CHECKPOINT LOCATION: 0/3000020
START TIME: 2011-08-23 22:16:40 CST
LABEL: mybackup
```

- ✧ 確任所有檔案擁有者及權限是否正確，為避免出錯建議使用作業系統帳號 postgre 複製檔案。
- ✧ 啟動資料庫伺服器。

如果並非使用本書介紹備份及還原資料庫方法而使用其它方式，仍然會使用主要方式還原資料庫：

- ✧ 停止資料庫伺服器
- ✧ 還原備份檔至$PGDATA，可使用手動複製或 rsync 方式。
- ✧ 確任所有檔案擁有者及權限是否正確，為避免出錯建議使用作業系統帳號 postgre 複製檔案。
- ✧ 移除任何存在於 pg_xlog 目錄中檔案，如果使用我的方式備份資料庫則省略這一步驟，因並未備份 WAL。
- ✧ 複製最近 WAL 至新資料庫伺服器。
- ✧ 建立 recovery.conf 並設定正確內容權限。
- ✧ 啟動資料庫伺服器。

特別注意檢查 recovery.conf 裏參數設定，尤其 restore_command 可是非常重要的。

restore_command 定義如何透過 WAL 復原(Recovery)資料庫，參數指定如何將 WAL 複製還原回來。

舉個例子，假設遠端備份伺服器名為「backup1」，則將以下參數寫入 recovery.conf 檔中。

```
restore_command = 'scp backup1:/backups/pg/servername/archive/%f %p'
```

如何做…

邏輯還原跟實體還原主要分別如下：

邏輯還原：匯入之前建立的備份檔，也就是於新資料庫重新執行 SQL 語法來建立物件。

實體還原：直接複製實體檔案故速度快於邏輯還原很多，但此方式除了最基本資料庫備份外，仍需 WAL 方可讓資料庫正常開啟。

資料庫進入熱備份模式後，會於$PGDATA 產生 backup_label 檔案，該檔內容包含封存日誌起始點及檔案名稱，也就是說當日後資料庫還原時需要應用這些封存日誌復原資料庫方能正常開啟。

當復原(Recovery)資料庫時，資料庫伺服器日誌檔會記錄每一筆執行 WAL 結果，因此根據日誌檔我們可了解資料庫復原狀況，另一方面我們查詢最後一個 WAL 便可推算出剩下幾個 WAL 尚未執行及復原可能所需時間。

執行 restore_command 後所傳回值須為 0，0 代表執行正常，非 0 代表執行失敗，資料庫復原動作會一直持續到沒有 WAL 可用，故在資料庫伺服器日誌檔中最終一定會出現找不到 WAL 的錯誤訊息，屬於正常現象不用過於擔心。

然而假設中間因遺漏 WAL 或該檔毀損導致資料庫復原中斷，則資料庫復原將停止於該時間點，則會造成資料庫資料遺失。

一旦我們將資料庫從復原模式開啟後並無任何錯誤訊息，則表此資料庫已正常開啟，無法再進入復原模式執行 WAL 囉！當然除了另一情形外，熱備援資料庫(Hot Standby Database)，此模式可同時讓資料庫伺服器進行復原及查詢，將在後面章節介紹。

時間點復原

假設資料庫於 15:22 發生異常，而資料庫都於 04:00 備份，我們希望資料庫可還原至 04:00 至 15:22 中的某個時間點，此方式就稱為時間點復原 PITR(Point-in-time recovery)；然而時間點復原僅支援實體備份並不支援邏輯備份喔！

準備開始

首先，選擇要復原的時間點，如果希望復原資料庫至最近狀態，則不需使用時間點復原，只要復原至最後一個 WAL 即可。

如何做…

選擇要復原的時間點，這個時間即「資料庫復原目標」，所以我們得指定時間戳記(Timestamp)；在 recovery.conf 檔案中，增加下面一行內容：

```
recovery_target_time = '2011-08-24 09:32:46.793691+08'
```

注意！時間需符合資料庫所設定的時區，可透過資料庫伺服器日誌檔查看時區是否一致，之後檢查是否正常復原。

如何運作…

資料庫復原時會執行 WAL 中每筆記錄，每筆交易記錄會分別寫入 WAL 中的每個區塊，直到最後將交易 Commit 值寫入 WAL，所以每筆交易記錄記錄時間戳記，因此允許我們選擇欲復原時間點。

除了指定時間戳記外，也可指定 Transaction id(xid)，但找出 xid 可是比較困難的喔！

「資料庫復原目標」定義在 recovery.conf 檔案中，當資料庫伺服器已經啟動時，是無法更改此檔案的，如一定得修改檔案內容時，也只能先暫停將資料庫伺服器關閉後待修改完成後重新開啟。但須注意一點，當資料庫復原至一半因特殊情況更改資料庫復原目標，如果所指定的時間戳記已在上次資料庫復原時復原完成，會導致錯誤而立即停止資料庫復原；相反的如指定 xid 而非時間戳記，而該 xid 已在上次資料庫復原時復原完成，資料庫復原會持續直到執行完 xid 所在的日誌檔。

一但資料庫伺服器復原完成，則成為一個完整的資料庫，可在此資料庫伺服器執行任何修改。

pg_dump 無法支援時間點回復，主要原因資料庫復原時重新執行交易日誌檔讓實體檔案區塊改變，而並非資料庫邏輯改變；但如果想使用 WAL 來改變資料庫結構如透過匯出備份檔修改 SQL 可是沒辦法的喔！各有不同的用途呢！

預計 PostgreSQL 9.1 版本會讓資料庫復原功能暫停/恢復/停止功能，如此可讓資料庫進行復原時更有彈性，也會讓有熱備援資料庫(Ho t Standby)需求可更精確的指定復原時間點。

復原刪除/毀損資料表

某些情況可能意外刪除資料表或遭遇資料表毀損，資料表可分實體及邏輯毀損，前者可能因硬碟毀損導致，後者可能不甚使用 UPDATE 或 DELETE 導致。

如何做…

如何復原資料表取決於不同備份方式，以下列出幾種備份方式如何復原。

邏輯復原(dump -F c)，不需還原全部資料至新資料庫：

備份時，pg_dump 指定格式為 custom，根據以下指令將由備份檔匯出欲復原的資料表，再執行 psql 匯入。

```
# pg_restore -t droptable dumpfile | psql -d dbname
```

以上指令會將刪除資料表重建於指定資料庫並匯入資料，但須注意 pg_dump -t 並不包含該資料表的索引喔！

然而實際情況上，可能不只是將資料表中的資料重新匯入而已喔！可能遭遇資料表毀損或意外刪除，處理方式就都不同了。

特定資料表毀損時，由備份檔指定該資料表並重新匯入，依照以下步驟執行：

由備份檔匯出指定資料表至 droptable.sql 檔：

```
# pg_restore -t droptable dumpfile > droptable.sql
```

建立腳本 restore_droptable.sql，內容如下：

```
BEGIN;
TRUNCATE droptablename;
\i droptable.sql
COMMIT;
```

接著，執行改腳本來匯入資料表：

```
# psql -f restore_droptable.sql
```

然而如果是資料表意外刪除，則需要考量該資料表索引也被刪除，故須匯入整個資料庫結構，步驟就複雜了一些；當然我們也可選擇手動建立索引，但找出索引可麻煩了許多喔！

- ✧ 建立一個新資料庫，名為 newdb。

```
postgres=# CREATE DATABASE newdb;
```

- ✧ 找出備份檔，僅匯入資料結構至 newdb 資料庫：

```
# pg_restore --schema-only -d newdb dumpfile
```

- ✧ 指定參數--schema-only，匯出指定資料表 schema 包含索引、約束限制及權限：

```
# pg_dump -t droptable --schema-only newdb > droptable.schema
```

- ✧ 匯入該資料表 schema 至原資料庫「maindb」，如此方能將資料表 schema 包含索引、約束限制及權限一併匯入：

```
# psql -f droptable.schema
```

- ✧ 匯入資料表資料：

```
# pg_restore -t droptable --data-only -d maindb dumpfile
```

假設該資料表很大時，匯入資料表資料時可能會非常緩慢，原因是不僅只是匯入資料並包含索引資料，故可考慮先將資料匯入後，再手動建立索引或許也是個不錯的方法。

邏輯還原，需還原全部資料至新資料庫：

此方式較簡單，但因需建立另一主機並還原全部資料至新資料庫故發費較多時間。

- ✧ 建立新主機或虛擬主機。
- ✧ 於新主機匯入所有備份檔。

```
# psql -f dumpfile.sql
```

- ✧ 連接至新主機，匯入該資料庫 maindb 中資料表 schema 包含索引、約束限制及權限。

```
# psql -t droptable -F c maindb > dumpfile
```

- ✧ 再回至原主機，匯入由新主機所匯出備份檔即可。

```
# pg_restore -d maindb -j 2 dumpfile
```

實體還原：

- ✧ 建立新主機或虛擬主機。
- ✧ 於新主機還原所有資料庫。
- ✧ 連接至新主機，匯入該資料庫 maindb 中資料表 schema 包含索引、約束限制及權限。

```
# psql -t droptable -F c maindb > dumpfile
```

- ✧ 再回至原主機，匯入由新主機所匯出備份檔即可。

```
# pg_restore -d maindb -j 2 dumpfile
```

復原刪除/毀損表格空間

某些情況中復原表格空間是必要的，實際上它比復原資料表複雜語多；不同的備份方式會造就不同的復原方式，以下會一一列出不同方式：

邏輯備份，使用 Custom 格式(-F c)：

假設使用 pg_dump 備份表格空間，我們可執行以下指令將資料表匯入：

```
# pg_restore -t  table1
```

然而，我們是不會記得欲復原的表格空間中存在哪些資料表，可執行以下 SQL 語法找出特定表格空間中的資料表，

```
postgres=# SELECT  spcname ,relname
          FROM pg_class t JOIN pg_tablespace ts
          ON reltablespace = ts.oid AND spcname = 'pg_global'
          JOIN pg_namespace n ON n.oid = t.relnamespace
          WHERE relkind = 'r';
```

- ✧ 尋找適合主機或虛擬主機。
- ✧ 將資料庫備份復原至新主機。

```
# pg_restore -d dbname -j 4 dumpfile
```

✧ 於新主機上指定表格空間備份。

➢ 此部份已在先前章「邏輯備份表格空間」介紹過。此部驟列出特定表格空間中的所有資料表，建立一文字檔名為「onets.sql」SQL 語法如下：

```
SELECT 'pg_dump'
UNION ALL
SELECT '-t ' || spcname || '.' || relname
FROM pg_class t JOIN pg_tablespace ts
ON reltablespace = ts.oid AND spcname = :TSNAME
JOIN pg_namespace n ON n.oid = t.relnamespace
WHERE relkind = 'r'
UNION ALL
SELECT '-F c > dumpfile';
```

➢ 執行以下指令來產生 pg_dump 備份檔：

```
# psql -t -v TSNAME="'tablespace_name'" -f onets.sql > ts_dump.sh
```

➢ 執行 ts_dump.sh 備份表格空間即可。

```
# chmod 755 ts_dump.sh
# ./ts_dump.sh
```

✧ 將備份檔 dumpfile 複制至原有主機並執行以下指令復原表格空間。

```
# pg_restore -d  dbname -j 2 dumpfile
```

邏輯備份，使用純文字格式(-F p)：

✧ 尋找適合主機或虛擬主機。
✧ 將資料庫備份復原至新主機。

```
# psql -f dumpfile.sql
```

✧ 於新主機上指定表格空間備份，可參考上節，故在此略過。

✧ 將備份檔 dumpfile 複制至原有主機並執行以下指令復原表格空間，可參考上節，故在此略過。

實體備份：

✧ 尋找適合主機或虛擬主機。
✧ 將資料庫備份復原至新主機，可參考熱備份復原資料庫章節。
✧ 於新主機上指定表格空間備份，此部份已於上節介紹，故在此略過。
✧ 將備份檔 dumpfile 複制至原有主機並執行以下指令復原表格空間，此部份已於上節介紹，故在此略過。

pg_dump 使用 Custom 格式(-F c)所匯入的備份檔，可在執行 pg_restore 時指定-l 參數列出全部備份檔內容。

復原毀損資料庫

在某此情況中復原資料庫是需要的，如果有完整的備份，事實上，復原資料庫應該是蠻簡單的。

如何做…

不同備份方式會有不同的復原資料庫方法，以下列出幾種方式：

邏輯備份，使用 Custom 格式(-F c)：

於原資料庫伺服器執行以下指令匯入備份檔，而-j 4 代表同時產生四個執行程序：

```
# pg_restore -d dbname -j 4 dumpfile
```

邏輯備份，使用純文字格式(-F p)

於原資料庫伺服器執行以下指令匯入備份檔：

```
# psql -f dumpfile dbnam
```

邏輯備份，使用 pg_dumpall：

當備份時使用 pg_dumpall 備份資料庫伺服器時，會備份全部資料庫，無法指定復原單一資料庫時，故需依以下步驟：

- ✧ 決定適合主機或虛擬主機。
- ✧ 匯入備份至資料庫伺服器。

```
# psql -f dumpfile
```

- ✧ 待復原全部資料庫後，再參考「邏輯備份資料庫」章節，匯出特定資料庫。
- ✧ 將上一步驟產生備份重新匯入至資料庫伺服器。

實體備份：

當備份時使實體備份資料庫伺服器時，會備份全部資料庫(資料庫實例)，無法指定復原單一資料庫時，故需依以下步驟：

- ✧ 決定適合主機或虛擬主機。
- ✧ 復原資料庫伺服器，可參考「復原資料庫伺服器」章節。
- ✧ 待復原全部資料庫後，再根據「邏輯備份資料庫」章節，匯出特定資料庫。
- ✧ 將上一步驟產生備份重新匯入至資料庫伺服器。

調校備份/復原效能

當資料庫容量很大時就得考慮效能的問題了，備份效能是我們得思考的議題，當執行備份時會佔用所有作業系統資源進而影響資料庫效能，故資料庫大都在離峰時間執行備份；至於復原效能呢？則就比備份效能更為重要，它取決於資料庫復原所需時間也就是停止服務期間，所以這可是我們無法忽視的課題。

如何做…

實體備份：

實體備份效能首先想到的就是同時啟用多個備份程序來改善效能，但須考慮伺服器硬碟 I/O 吞吐量(throughput)，因此造成系統效能衝擊；另一方面可降低備份所需檔案，在一般情況中過多的小檔案反而比備份一個大檔案來的慢許多呢！以下列出可省略備份的檔案：

- ✧ 所有暫存檔，可能由 DBA 或其它原因而產生。
- ✧ 所有 pg_xlog 目錄中檔案
- ✧ 原伺服器上的資料庫伺服器日誌檔可能位於 pg_log 目錄中

切記！可不要忽略了真正該備份的檔案喔，否則得不償失反而造成資料移失；另外備份效能也可能與網路瓶頸有關，取決寬頻多寡，理論上資料庫伺服器應有專屬備份網路。

邏輯備份：

在先前章節已介紹，如果使用 pg_dump 指令備份資料庫伺服中全部資料庫，則應該同時執行多個 pg_dump 指令來平行備份；備份時的壓縮率也會影響備份效能。

實體復原：

與實體備份相同，同時啟用多個復原程序來改善效能。

邏輯復原：

當使用 psql 或 pg_dump 復原資料庫時，可設定參數 maintenance_work_mem = 128 MB 來改善復原效能，參數可透過以下方式設定：

- 使用作業系統環境變數： export PGOPTIONS=" -c work_men = 128000"。
- 於資料庫參數檔 postgres.sql 設定參數。

如果資料庫伺服器上啟用封存日誌模式或串流複寫(Streaming Repication)，則資料庫瓶頸可能發生在寫入交易檔行為，此時建議調大參數 wal_buffers 至 2,000 或 10,000 以上及參數 checkpoint_segments 至 1024 來改善效能。

在某些情況中，資料庫伺服器中並非服務重要系統，則可不啟用封存日誌模式或串流複寫來降低寫入交易檔行為所造成的硬碟 I/O 衝擊，如此方式也可以改善效能。

當執行 pg_dump 備份時使用 Custom 格式(-F c)，可在復原時指定多個執行程序：

```
# pg_restore -j NumJobs
```

但當同時指定多個程序進行資料庫備份或復原時，須注意主機 CPU 資源是否足夠？另外，需特別注意同時指定多個程序進行資料庫復原是否會造成主機實體記意體用盡而造成虛擬記體體切換(Swapping)，因每個復原執行程序進行資料庫復原時會使用 shared_buffers 及 mainenance_work_mem 這些記憶體來重建索引。

而當復原完成後也須注意是否有自動執行 autovacuum 或手動執行 Analyze，在資料庫很大時這可是會影響日後資料庫運行效能及應用系統順暢.。

更多…

壓縮備份是提昇效能一個不錯的選擇，壓縮不僅降低備份檔所佔空間，甚至在傳輸備份檔時大大加快及減少網路頻寬，唯一考量的是會提高 CPU 資源。

另外可透過 pg_lesslog 額外模組進行壓縮 WAL 來提昇效能可參閱此網站 http://pgfoundry.org/frs/?group_id=1000310。實體備份可透不同方式壓縮，如 pg_dump 使用 Custom 格式，預設會進行壓縮備份檔。

而如 pg_dump 使用文字格式則可搭配系統指令 zip 進行壓縮：

```
# pg_dump | gzip
```

當資料庫變的很大時，執行備份或復原可能變的愈來愈困難，不僅備份檔空間佔用大量空間，且復原速度也是重要關鍵，此時不彷可考慮啟用資料庫複寫來取代原本備份方式，可參閱後「資料庫複寫」章節。

遞增/差異備份及復原

如想使用遞增或異備份及復原，一般而言可使用 rsync 指令即可做到，然而此方式卻會重新覆蓋備份檔，在某些情況中是不符合我們備份需求，照理說應保留多天備份如以下表格：

日期	備份種類
星期日	完整備份
星期一	遞增備份
星期二	遞增備份
星期三	遞增備份
星期四	遞增備份
星期五	遞增備份
星期六	遞增備份

如何運作…

PostgreSQL 內部運作資料表或資料檔並不支援追蹤異動區塊，主要原因為 PostgreSQL 資料表存放於檔案系統中，當 mtime 機制被停用時，可能造成遞增備份錯誤，因追蹤異動區塊是透過 mtime 判斷，故不採取此方式。

一般情況下，也無法使用 pg_dump 達到遞增備份，故無法在備份時指定資料表條件範圍，所以就算在資料表增加欄位 last_changed_date 來辨別如何產生遞增備份，也只能手動執行條件產生遞增備份。

當思考遞增備份時，不彷考慮複寫技術取代，尤其複寫時在備援端是同時進行復原，所以不用擔心當資料庫 Crash 時須將備份檔復原至新資料庫伺服器所花費的時間；但切記！我們應該在備援伺服器端執行資料庫備份喔！

其實追蹤異動區塊產生遞增備份是可以達到的，當進行完整備份後，可透過讀取並檢查 Log Sqeuence Number(LSN)區塊來獲得資料庫異動資訊而 pg_rman 就是一個例子。

pg_rman 是一個開源專案，可參閱以下網址獲得更多資訊：
http://code.google.com/p/pg-rman/

pg_rman 透過追蹤資料區塊異動並壓縮異動資訊來達到資料庫遞增備份，是

一小型軟體，但無法同時產生多個執行程序，不過使用 pg_rman 比執行完整備份的速度可快了許多。

CHAPTER 12

複寫 & 升級

本章內容大綱如下：

- 前言
- 實作複寫
- 設定 file-based log shipping 複寫
- 設定串流複寫(Streaming Replication)
- 管理 log shipping 複寫
- 管理熱備援(Hot Standby)資料庫
- 使用 Londiste 3.0 實作複寫
- 使用 Slony 2.0 實作複寫
- 使用 pgpool II 3.0 達到負載平衡
- 資料庫小版本升級
- 使用 In-Place 實作資料庫大版本升級
- 透過複寫實作資料庫版本升級

前言

資料庫複寫架構比一般資料庫維護複雜了許多，有許多觀念非常重要，本章由簡入深從資料庫複寫觀念至實作複寫甚至透過複寫達到資料庫版本升級。

PostreSQL 從 8.2 版本就擁有複寫功能，儘管一開始版本並非相當的穩定，但對 PostgreSQL 發展史而言複寫功能是一個重要的里呈碑；或許有些人覺得 PostgreSQL 複寫功能有許多缺點功能，功能也不夠定整，對我而言，PostgreSQL 複寫功能已足夠應付現有狀況。

了解複寫概念

如何運作

資料庫複寫是一種用於將資料複製至遠端資料庫伺服器的技術，主要有兩個原因如下：

- 高可用性(High Availability)：
  透過安裝二個以上資料庫伺服器，彼此備援避免因單一伺服器發生故障造成服務停止。

- 資料搬移(Data Movement)：
  為了特殊目的而將資料搬移讓特定系統使用或減輕伺服器負擔，如商業智慧(Business Intelligence)或報表系統(Reporting System)。

資料搬移，如指定特定 PostgreSQL 資料庫伺服器複製至遠端資料庫伺服器，照理說該遠端資料庫伺服器不會有任何交易行為發生，只用於資料分析或報表。

一般而言，一個節點上會有一個資料庫伺服器，如又有另一結點透過複寫技術同步資料庫伺服器，則可稱為「叢集(Cluster)」架構；然而，PosgreSQL 對於資料庫實例(Instance)也稱為叢集(Cluster)，可不要搞混喔！

資料庫複寫技術中一般會將提供資料來源端稱為主(Mater)、首要(Primary)、提供者(Provider)、傳送者(Sender)或來源(Source)伺服器；而接收來源資料端稱為備援(Standby)、從屬(Slave)、訂閱者(Subscriber)或接收者(Receiver)伺服器。

此章節所介紹內容全為單一主伺服器、多台備援伺服器，至於多台主伺服器將於後面章節介紹。

在某些情況中需要將主伺服器切換至另一節點，此一動作稱為「Switchover」；而主伺服器發生異常並且服務無法復原，必須切換伺服器角色，此一動作稱為「Failover」，這兩個專業名詞常常被人們搞混，故舉例讓大家更易分辨。

一般特別為管理叢集所設計的軟體，自動執行 Failover，稱為叢集軟體(Clusterware)，此軟體也大部份提供負載平衡功能。

主伺服器中會截取資料異動並傳送至備援伺服器，但有些情況中某特定備援伺服器也有可能截取資料異動並傳送至其它備援伺服器，此伺服器稱為中繼(Relay)伺服器。

當主伺服器執行 Commit 後，主伺服器將異動資料傳送至備援伺服器所花費時間，一般稱為資料延遲(Latency)或複寫延遲(Replication Delay)，複寫延遲最好以秒(s)來評估；而當異動資料由主伺服器傳送至備援伺服器後，備援伺服器套用這些異動資料所花費時間稱為套用延遲(Apply Delay)；以上兩種延遲累加所花費的時間為主伺服器及備援伺服器同步差異，在實作資料庫複寫時 DBA 須評估主伺服器及備援伺服器之間資料落差時間及使用者可接受彼此的差異量，也就是說主伺服器及備援伺服器之間是否要求的那麼即時。

當主伺服器在交易尚未 Commit 之前就傳送資料異動通知至備援伺服器，此模式稱為同步(synchronous)複寫；相反的當主伺服器在交易 Commit 之後才傳送資料異動通知至備援伺服器，此模式稱為異步(asynchronous)複寫。在同步複寫模式中，複寫延遲可是會影響到主伺服器效能，異步複寫模式中主伺服器則不被受影響，但須考量主伺服器與備援伺服器間資料量是否落差太大故應做好監控才行。

建立單一資料庫複寫方式大多大同小異。首先，啟用追蹤異動資料功能，接著將異動資料傳送至備援伺服器，而備援伺服器套用這些異動資料；最佳狀態是複寫延遲時間應該與直接複寫 WAL 至備援伺服器所花費時間相同，而備援伺服器持續的套用這些異動資料，如此方能將同步差異降到最小，然而實際情況中可能因主伺服器過於忙碌導致同步差異時間加長，而造成使用者無法接受彼此資料差異太大而報怨或無法繼續作業，調效資料庫複寫就成為另一個課題囉！

資料庫複寫功能可複寫資料庫中所有資料表，也可過濾只針對某些資料表進行複寫，可將這些特定資料表群組成一複寫集，雖然此方式不必將資料庫中所有資料表複寫至備援資料庫，但相反的 DBA 管理工作相對增加，當欲做複寫資料表超過幾百張以上時，對於管理面也是一個負擔。

更多

多台主伺服器複寫架構在實作上會遭遇許多困難，主要為資料一致性問題，而產生複寫上困難；個人經驗單一主伺服器複寫架構加以應用規劃，就足以滿足現有狀況，無須使用更複雜的複寫架構而增加管理上負擔及資料一致性風險。故本章節主要著墨在單一主伺服器複寫架構上。

建立資料庫複寫時注意原則

以下列出當建立資料庫複寫時應有原則，可讓管理更明確簡單。

如何做…

- ✧ 使用同等級硬體及相同作業系統：
  資料庫複寫在節點間切換角色，不管是 Switchover 或 Failover，但當硬體備配相差甚遠時，將會造成效能議題，故須考量當節點切換時，備援伺服器是否能承載原有服務。

- ✧ 各節點間環境配置盡量使用：
  使用相同掛載點(Mount Point)、相同目錄名稱及相同使用者，在環境配置盡量，避免角色切換後產生不必要困擾。

- ✧ 良好主機命名原則：
  命名主機名稱時不要取 Mater 及 Slave 代表主伺服器及備援伺服器，因當切換角色後會造成困擾。

- ✧ 各節點時間同步：
  當查看資料庫伺服器日誌檔時，各節點保持相同時間將有助於分析問題，可使用 NTP 伺服器設定同步。

- ✧ 各節點相同時區：
  各節點皆設定協和標準時間(UTC)或相同時區，當各節點時區設定相同時也應注意是否有啟用日光節約時間*(*Daylight Saving Time)，避免造成管理不必要困擾。

- ✧ 監控各節點健康狀況：
  為了提高服務高可用性，應定期檢查各節點伺服器健康狀況，避免當災難發生時，備援伺服器無法接手服務，備而無用。

- ✧ 監控資料庫複寫同步狀況：
  建立資料庫複寫後應隨時監視主伺服器與備援伺服器同步狀況，避免當時間落差過大，造成資料誤差，故當同步差異過大時應對複寫進行調校，或找尋問題原因。

File-Based Log Shipping 複寫

Log Shipping 是資料庫複寫中的一種功能，由主伺服器記錄資料異動至交易記錄檔，再將記錄檔傳送至備援伺服器，最後備援伺服器套用。

File-based log shipping 功能出現於 PostgreSQL 8.2 版本，離 PostgreSQL 9.0 版本釋出約四年時間，已算是成熟功能，故此複寫功能所需資源少、穩定且簡單設定。

不過當 PostgreSQL 9.0 版本推出 Streaming 複寫功能時，File-Based log shipping 複寫機乎可被 Streaming 複寫取代，但在許多資料庫備份策略中仍非常有用，值得學習如何設定及原理，並可成為進入 Streaming 複寫的入門。

準備開始

在本章開始前請先詳讀「了解複寫概念」及「建立資料庫複寫時注意原則」章節，資料庫複寫是複雜困難的，儘管我們覺得設定應該沒問題，但實作上可能還會遭遇許多狀況。

如何做…

依照以下步驟設定 File-based log shipping 複寫：

✧ 確認封存日誌目錄並檢查空間是否足夠：

主伺服器及備援伺服器均須建立封存日誌目錄，主伺服器將在封存日誌目錄中寫入 WAL，而備援伺服器將在封存日誌目錄中讀取 WAL 復原資料庫。在此我們以系統環境變數$PGARCHIVE 代表封存日誌目錄路徑，$STANDBY 代表備援伺服器。

✧ 主伺服器及備援伺服器建立 SSH 信任授權機制，允許使用 rsync 指令同步資料。

✧ 設定主伺服器參數檔 postgresql.conf：

```
wal_level = 'archive'
archive_mode = on
archive_command = 'scp %p $STANDBY:$PGARCHIVE/%f'
archive_timeout = 30
```

✧ 設定備援伺服器參數檔。

```
wal_level = 'archive'
archive_mode = on
archive_command =
archive_timeout = 30
```

✧ 同步主伺服器資料，步驟與熱備份資料庫相似，可參閱「熱備份資料庫」章節。

➢ 進入熱備份模式：

```
# psql -c "SELECT pg_start_backup('backup for log shipping')"
```

➢ 複製主伺服器中所有資料檔至備援伺服器，須排除 pg_xlog 目錄：

```
# rsync -acv --inplace --exclude=*pg_xlog* $PGDATA $STANDBY:$PGDATA
```

➢ 離開熱備份模式：

```
# psql -c "SELECT pg_stop_backup(), current_timestamp"
```

✧ 於備援資料庫上建立 recovery.conf 檔並設定參數：

```
standby_mode= 'on'
restore_command = 'cp $PGARCHIVE/%f  %p'
archive_cleanup_command = 'pg_archivecleanup $PGARCHIVE %r'
trigger_file = '/tmp/postgresql.trigger.5432'
```

- ✧ 啟動備援伺服器。
- ✧ 確認資料庫複寫是否正常運作，並觀察同步差異是否過大。

如何運作…

主伺服器須確認是否啟用封存日誌模式且最佳化，藉由透過設定 archive_command 將 WAL 傳送至備援資料庫，而備援資料庫透過 restore_command 套用這些異動資料。

當 WAL 寫滿交易記錄或到達 archive_timeout 時間後，WAL 將傳送至備援資料庫，然而當主伺服器如無任何交易發生，則 WAL 會於到達 checkpoint_timeout 時間時再依序產生下一個 WAL 編號並不會造成複寫中斷。

備援伺服器使用 restore_command 參數藉由 cp 指令將 WAL 套用至備援資料庫來達到同步，而封存日誌目錄位於遠端伺服器時，則需掛載該遠端伺服器目錄，或使用網路複寫指令來達成，如 scp 指令。

自 PostgreSQL 9.0 版本後 archive_cleanup_command 指令設計於幫助備援伺服器定期清除不需要 WAL，當然也可使用在一般資料庫伺服器來清除過期 WAL，當下達 archive_cleanup_command 指令時須指定二個參數：封存日誌目錄名稱及 %r。

在欲開始備份主伺服器資料至備援伺服器之前，由於我們使用 rsync 同步彼此節點資料，故需建 SSH 信任授權機制，如此在執行 rsync 指令時才不會因需輸入密碼而造成麻煩，也增加自動化負擔。

更多…

監控 File-based log shipping 運作機制可藉由查看主伺服器及備援伺服器執行程序來判斷複寫機制是否運作。

```
master# ps -ef |grep archiver
postgres  4498  4474  0 01:09 ?        00:00:00 postgres: archiver

standby# ps -elf | grep startup
postgres: startup process   waiting for
```

在 PostgreSQL 9.0 版本以前或許很難評估複寫同步的差異，但現在已不是困難的事了，至於更多如何監控複寫機制會於「管理 log shipping」章節有更多介紹。

如使用 PostgreSQL 8.2 至 PostgreSQL 8.4 版本欲建立 File-based log shipping 機制，唯一不同的是 postgresql.conf 設定檔中的 wal_level 參數，且 PostgreSQL 9.0 版本可不使用 pg_standby 指令，當然該指令仍存在的！

不管我們使用 Log shipping 或 Streaming 複寫機制，基本上管理的方式都很類似。

如何做…

監控(Monitor)

監控複寫機制非常重要，藉由這些監控所獲得的資訊，讓我們可以掌握複寫情況及主伺服器與備援伺服器資料同步狀態。

如 Repmgr 與 pgpool 工具都提供監控複寫機制功能，可安裝外掛(Plugin)程式 Munin 套件讓複寫延遲透過圖型化呈現。

少了這些工具，我們也可手動執行指令的方式來獲得資訊，獲得主伺服器與備份伺服器同步狀態並進行分析。

在主伺服器執行以下 SQL 語法：

```
postgres=# SELECT pg_current_xlog_location();
```

在備援伺服器執行以下 SQL 語法：

```
postgres=# SELECT pg_last_xlog_receive_location();
```

跟據以上兩個查詢結果可獲得解複寫延遲狀況；而執行以下 SQL 語法在備援伺器可獲得套用延遲資訊：

```
postgres=# SELECT pg_last_xlog_apply_location()
```

Switchover & Failover

Switchover 意指切換備援伺服器成為主伺服器，假使切換動作正確及順利的話，照理說是不會有任何資料遺失；基於安全性理由，強烈建議關閉主伺服器時應使用 smart 或 fast 模式，不應該使用 immediate 模式，以免造成資料遺失。

Failover 意指可能因為主伺服器發生異常而強制切換備援伺服器成為主伺服器角色，然而如何將備援伺服器切換為主伺服器呢？藉由建立觸發(trigger)檔案讓備援伺服器切換為主伺服器即可。

```
# touch /tmp/postgresql.trigger.5432
```

備援伺服器成為主伺服器後，將開始正常運作提供服務並產生封存日誌檔，須得注意彼此設定值是否設定相同喔！避免切換後造成服務異常，另外一點，當備援伺服器成為主伺服器後，原本的主伺服器須確保停止服務，避免造成兩台「主」伺服器而造成腦分裂(Split-brain)。

在 PostgreSQL 中使用 failover 或許步驟較複雜麻煩，因未提供自動 Failover 功能，許多動作需手動去執行，並不怎麼方便管理。

Switchback

switchback 意指在切換備援伺服器成為主伺服器後，再彼此切換為原來角色；然後在 PostgreSQL 中當備援伺服器成為主伺服器後，無法再次切換回備援伺服器，許多人聽到後都非常驚訝，PostgreSQL 並無此功能，但是有其它方法變通的(Work around)。

- ✧ 重新設定原主伺服器，再依照之前所提方式重新複寫機制。
- ✧ Switchover 彼此角色。

須特別注意當我們重建原主伺服器時，並未刪除舊有資料，是透過 rsync 方式將新主伺服器(也就是原備援伺服器)資料同步至原主伺服器，由於差異資料未刪除，加上 rsync 只同步異動過的區塊，照理同步時間應該很快；待同步完成並重新建立複寫機制後再執行 Switchback 即可。

更多…

觸發檔案內容並無任何資料，透過建立該檔案讓備援伺服觸發成為主伺服器，當然觸發檔檔案名稱是可以自行定義的，而取 5432 主要原因是 PostgreSQL 預設監聽埠是 5432，並無在多意義存在；而 recovery_end_command 參數可以讓備援伺服器切換成為主伺服器後，觸發檔案會自動被清除，建議 PostgreSQL 之前版本須增加此設定。

雖然剛有說過 PostgreSQL 本身並無提供自動 Failover 機制，然而搭配第三方軟體是可以滿足的，如 Repmgr 工具。

Repmgr 為一開源軟體，特別為加強 PostgreSQL 複寫機制而開發，提供監控、自動 Failover 及讓管理複寫更方便簡易，可參閱以下網址：
http://projects.2ndQuadrant.com/repmgr/

另也有商業版本軟體名為 Continuent Tungsten，可支援 PostgreSQL 或 MySQL 資料庫，可參閱以下網址：
http://www.continuent.com/

Streaming Log 複寫

PostgreSQL 9.0 版本後推出 Streaming log 複寫功能，從設計、開發、測試到正式釋出歷經四年之久，主要特點為主伺服器直接傳送資料至備援伺服器，整合安全性也減少了複寫延遲時間。

使用 Streaming 複寫可使用兩種方式：傳送或不傳送封存日誌檔，本書介紹以不傳送封存日誌檔方式來建立複寫機制，此方式簡單又有效率。

如何做…

依照以下步驟：

- ✧ 主伺服器及備援伺服器均須建立封存日誌目錄，主伺服器將在封存日誌目錄中寫入 WAL，而備援伺服器將在封存日誌目錄中讀取 WAL 復原資料庫。在此我們以系統環境變數$PGARCHIVE 代表封存日誌目錄路徑，$STANDBY 代表備援伺服器。

- ✧ 主伺服器及備援伺服器建立 SSH 信任授權機制，允許使用 rsync 指令同步資料。

- ✧ 設定資料庫安全性，建立一使用者專門為複寫所使用。

```
postgres=# CREATE USER repuser
               SUPERUSER
                LOGIN
                CONNECTION LIMIT 1
                ENCRYPTED PASSWORD 'repuser';
```

- ✧ 允許「repuser」使用者連接資料庫伺服器，以下範例允許 repuser 使用者於任何主機使用加密過密碼(md5)登入。

```
host    replication    repuser    127.0.0.1/0      md5
host    replication    repuser    192.168.10.12/32        md5
```

- ✧ 在主伺服器及備援伺服器上稽核所有使用者登入記錄，將有幫助於建立複寫機制時發生問題的困難排除，於 postgresql.conf 檔中設定。

```
log_connections = on
```

- ✧ 建立封存日誌目錄

```
# mkdir -p /opt/PostgreSQL/9.1/archive
# chown -R postgres.postgres /opt/PostgreSQL/9.1/archive
```

- ✧ 在主伺服器中編輯 postgresql.conf 檔設定最多幾個 wal sender 執行程序。

```
max_wal_senders = 5
wal_level = archive
archive_mode = on
archive_command = 'cp %p /opt/PostgreSQL/9.1/archive/%f'
```

- ✧ 在主伺服器中編輯 postgresql.conf 檔，調整 wal_keep_segments 參數，此設定值所產生的空間不能比 pg_xlog 目錄所在的掛載點還要大，避免造成空間不足。
  16(MB) X wal_keep_segments 個數 = 所需空間

```
wal_keep_segments = 32   # 約 160GB
```

- ✧ 設定備援伺服器參數。
- ✧ 同步主伺服器資料，步驟與熱備份資料庫相似，可參閱「熱備份資料庫」章節。
  - ➢ 進入熱備份模式：

```
# psql -c "SELECT pg_start_backup('backup for streaming')"
```

  - ➢ 複製主伺服器中所有資料檔至備援伺服器，須排除 pg_xlog 目錄：

```
# export STANDBY=192.168.10.12
# export $PGDATA
# rsync -acv --inplace -e ssh --exclude=*pg_xlog* $PGDATA
$STANDBY:/opt/PostgreSQL/9.1/
```

rsync -acv --inplace --rsh 'ssh -p 5800' --exclude=*pg_xlog* $PGDATA
$STANDBY:/opt/PostgreSQL/9.1/

- 離開熱備份模式：

```
# psql -c "SELECT pg_stop_backup(), current_timestamp"
```

- 於備援資料庫上建立 recovery.conf 檔並設定參數，特別注意 primary_conninfo 使用 IP 位址取代主機名稱。

```
standby_mode = 'on'
primary_conninfo='host=192.168.10.11 port=5432 user=repuser'
trigger_file = '/tmp/postgresql.trigger.5432'
```

- 於家目錄建立資料庫伺服器密碼檔.pgpass
(當啟動備援資料庫伺服器時讓複寫帳號 repuser 可自動連接)

```
# vi ~/.pgpass
```

內容如下：

```
192.168.10.11:5432:*:repuser:repuser
```

```
# chmod 0600 .pgpass
```

- 啟動備援伺服器。

- 確認資料庫複寫是否正常運作，並觀察同步差異是否過大。

如何運作…

由於多台備援伺服器允許連接至單一主伺服器中，故如環境有此構架則須設定 max_wal_senders 參數，讓主伺服器分別建立專屬 WALSender 執行程序為每一台備援伺服器建立複寫機制，相反的各備援伺服器建立 WALReceiver 執行程序與主伺服器共同連接。資料以每一封包 8192 Bytes 有效率的傳輸。

WALSender 與 WALReceiver 將同時運作持續運作到無任何異動資料傳送，也就是說當一段時間無任何資料異動，則 WALReceiver 閒置 100 毫秒(ms)，而 WALSender 會根據 wal_sender_delay 參數設定值閒置。一般而言 wal_sender_delay 並不需要修改，預設為 1 秒(s)，故維持預設值即可。

而當主伺服器與備援伺服器之間頻寬受限，而主伺服器異動量又相當高時，則 wal_sender_delay 可能需要縮短至 20 毫秒(ms)、50 毫秒(ms)或更短時間，避免當主伺服器 Crash，切換至備援伺服器時，減少資料同步落差。

備援伺服器是透過 libpq 函式庫連接至主伺服器，也就是說此複寫方式是經過驗證及安全性保護，無須額外設定防火牆，複寫方式也支援 SSL 加密，但 SLL 會輕微增加 CPU 資源負擔及複寫延遲。

更多

主伺服器與備援伺服器網路是有可發生異常的，為了確保網路中斷時可以即時產生訊息，可透過 keepalive 參數調整。

假如欲在當主伺服器發生連接中斷時，備援伺服器產生訊息，則在 recovery.conf 檔加入 keepalives 參數：

```
primary_conninfo = 'host=192.168.10.11 port=5432 user=repuser
keepalives_idle = 60'
```

假如欲在備援伺服發生連接中斷時，主伺服器產生訊息，則在 postgresql.conf 檔加入 keepalive 參數：

```
tcp_keepalives_idle = 60   # time before we seend  keepalives
```

當主伺服器與多台備援伺服器建立複寫時，以上設定均套用全部複寫機制，而欲分別指定複寫機制則可在備援伺服器 recovery.conf 指定參數：

```
primary_conninfo = 'host=192.168.10.11 port=5432 user=repuser -c
keepalives_idle = 60'
```

以上範例皆設定連線中斷 60 秒(s)後產生訊息，因 PostgreSQL 預設為 2 小時偵測一次，不建議採用。

另外也可以增加 max_wal_senders 參數設定值，讓在發生連線中斷時，可以允許重新再建立連線並更容易快速連接；如欲增加此參數值時，記得加大 repuser 使用者連線限制喔！

在 Streaming 複寫中，主伺服器會保留最近一份 wal_keep_segments，但當備援伺服器停止運作並持續很長一段時間時，主伺服器會將最近一份 wal_keep_segments 移除並不再傳送，此時只能重建備援伺服器以建立複寫機制。

另外提醒一點，如在進行主伺服器與備援伺服器同步期間，花費時間太久，無法立即建立複寫機制，可能導致主伺服器中的 wal_keep_segments 移除而在建立複寫時發生錯誤：

```
FATAL:   requested WAL segment  000000010000000000000002 had already  been
moved
```

此時唯一能做的就是增加 wal_keep_segments，直到備援伺服器所需的 wal_keep_segments 不會被主伺服器移除，但相對這些封存日誌將佔很大的空間。

為了讓主伺服器與備援伺服器之間可同時扮演傳送端及接收端角色，讓發生 Crash 時可以彼此切換，可同時設定參數 archive_command 及 archive_mod3 = on 於 postgrel.conf 檔內，但相對的 WAL 可會被複製兩份喔，佔用空間更高。

異動資料只在主伺服器寫入異動至硬碟後才會傳送至備援伺服器，因此有些人主張設定參數 synchronous_commit = off 想要改善複寫延遲，但最多只會改善主伺服器效能無法縮短同步差異，當 WAL 從備援伺服器接收完成後，WAL 被 fsynced 寫入至硬碟，確保當備援伺服器因 Crash 重開時導致資料遺失。

監控 Streaming 複寫是非常重要的，然而 WANSender 執行程序無法在系統資料表 pg_stat_activity 查詢到，故提供以下 SQL 語法建立視觀表提供大家查詢：

```
postgres=# CREATE OR REPLACE VIEW pg_stat_replication AS
         SELECT
         S.procpid,
         S.usesysid,
         U.rolname AS usename,
         S.application_name,
         S.client_addr,
         S.client_port,
         S.backend_start
         FROM pg_stat_get_activity(NULL) AS S, pg_authid U
         WHERE S.usesysid = U.oid AND S.datid = 0;
```

管理 Hot Standby

Hot Standby 伺服器意指備援伺服器一邊進行同步資料一邊允許使用者連接進行資料查詢，Host Standby 可應用於 File-based 複寫或 Streaming 複寫上；由於 Hot Standby 可線上提供查詢，能將較複雜或執行較久查詢在 Hot Standby，藉此減輕主伺服器負擔，故 Hot Standby 除了當備援外，也常應用在報表系統上。

管理 Hot Standby 伺服器相當錯綜複雜，最好充份了解 Hot Standby 伺服器運作機制，尤其與主伺服器間的資料落差，可能導致使用者在 Hot Standby 伺服器與主伺服器所查詢到的資料產生衝突，進而引響作業。

Hot Standby 伺服器，同時扮演二種角色，第一，當主伺服器發生異常時，Hot Standby 伺服器進行 Failover 立即接手服務；第二，提供使用者查詢；在某些情況下此二種角色可能互相衝突，須考量環境中何種角色最符合需求或如何搭配應用。

如環境中主伺服器允許擁有兩台以上 Hot Standby 伺服器，不彷考慮一台 Hot Standby 伺服器扮演援備主伺服器角色，而另一台 Hot Standby 伺服器提供查詢用於減輕主伺服器負擔。

準備開始…

Hot Standby 伺服器可應用在以下方面：

- File-based 複寫
- Streaming 複寫
- 執行單一時間點回復

首先，須依照之前章節介紹建立複寫機制，再分別為主伺服器及備援伺服器中 postgresql.conf 設定檔增加參數：

主伺服器：

wal_level = 'hot_standby'

主備援伺服器：

hot_standby = on

在修改參數值後，同時重啟主伺服器及備援伺服器讓新參數生效，在重啟期間可能會造成彼此同步落差加大，一般落差時間應與 checkpoint_timeout 差不多了多少，但假設重啟備援伺服器時間太快，當備援伺服器開啟後仍會讀取舊交易記錄檔，導致無法正常開啟並出現如下錯誤訊息："you need enable Hot Standby"，只須再設定一次即可，此錯誤並不會很常發生。

以下列出 Hot Standby 額外參數：

- max_standby_archive_delay
- max_standby_streaming_delay
- vacuum_defer_cleanup_age
- trace_rcovery_messages

....

如何做…

當查詢在備援伺服器執行時，應會比在主伺服器執行查詢資料落差一些，但如何評估到底落差多少呢？我們可透過此兩個參數限制最高落差時間：max_standby_streaming_delay 及 max_standby_archive_delay。

但又如何設定這兩參數的設定值呢？則須了解主伺服器與備援伺服器間四種衝突：

- 資源-CPU、I/O…等硬體配備。
- 鎖定-AccessExclusiveLock。
- Cleanup records。
- 其它可能原因。

硬體設備資源衝突一般而言較容易理解，當備援伺服器正在同步由主伺服間所傳送資料差異時，此時相對的能提供查詢的資源就變少，相反的當主伺服器變更較少時，此時備援伺服器就有足夠的資源來提供查詢，所以如何分配備援伺服器資源就顯的特別重要。

鎖定衝突，假設主伺服器執行 ALTER TABLE table_name DROP COLUMN 後，首先會鎖定該資料表避免其它使用者存取，當然該鎖定需求也會傳送至備援伺服器，此時會造成備援伺服器無法提供查詢。

為了提供高可用性，在執行 DDL 指令時可能會造成資料表長時間鎖定的困擾，尤其在主伺服器上，有時候我們希望在主伺服器執行 DDL 而造成資料表鎖定這段時間，備援伺服器可暫時提供查詢直到主伺服器中的資料庫鎖定結束，為了達到以上目的；可藉由在備援伺服器設定參數 max_standby_streaming_delay = -1 與 max_standby_archive_delay = -1，執行 reload 後使參數生效；此時只要當備援伺服器接收到第一筆鎖定記錄時，便停止後面一連串由主伺服器所傳送異動的資訊套用，直到主伺服器鎖定解除後，再將備援伺服器剛所設定的參數還原回來即可。

Cleanup records 及其它特別原因也可能造成相衝，一些特別原因可能發生的機率很少，舉例來說當在主伺服器刪除資料庫時，將造成備援伺服器查詢不再繼續。

Cleanup records 會移除資料庫中舊版本 tuple，除了那些正在運作的 MVCC，欲瞭解此運作最直接方式就是在主伺服器上執行 VACUUM，此動作會將在現在查詢期間 MVCC 中不再使用的 Row 版本移除；然後在現實情況中 VACUUM 會定期在主伺服器執行，將 Cleanup records 寫入交易記錄檔中，而當主伺服器將該記錄檔傳送至備援伺服器卻造成備援伺服器所需要 MVCC 中 Row 版本清除，進而發生查詢異常，我們可藉由設定 vacuum_defer_cleanup_age 設定值大於 0 來避免發生此情況。

由於主伺服器會將異動資料傳送至備援伺服器，接著再由備援伺服器將這些異動資訊套用而達成同步，然而在備援伺服器查詢時，有時需要完整的 MVCC 資訊，為了盡量符合此情況，何種種類 SQL 查詢在備援伺服器上執行就非常重要，盡量選擇查詢所需時間短、快速、量大的 SQL 語法，相對主伺服器與備援伺服器間衝突減少，運作當然也正常，然而環境中當備援伺服器當報表系統使用時，就得盡量調整參數來符合需求囉。

```
# ./configure --with-pgsql=/opt/PostgreSQL/9.1
--with-pgsql-libdir=/opt/PostgreSQL/9.1/lib
# make
# make install
```

/usr/local/etc

```
cp pcp.conf.sample pcp.conf
```

vi /usr/local/etc/pcp.conf

```
[root@db1 etc]# pg_md5 postgres
e8a48653851e28c69d0506508fb27fc5
```

mkdir -p /var/run/pgpool
pgpool -f /usr/etc/pgpool.conf -d -n

pg_hba.conf 加入 192.168.10.1 /24 md5

使用 postgres 編譯

```
cd /home/postgres/pgpool-II-3.1/sql/pgpool-recovery
make
su -c "make instsall"
```

```
/home/postgres/pgpool-II-3.1/sql/pgpool-regclass
make
su -c "make instsall"
```

```
$ cd  /opt/PostgreSQL/9.1/share/postgresql/contrib
$ psql -f pgpool-recovery.sql template1
$ psql -f pgpool-recovery.sql postgres
$ psql -f pgpool-regclass.sql template1
$ psql -f pgpool-regclass.sql postgres
$ psql -f pgpool-walrecrunning.sql template1
$ psql -f pgpool-walrecrunning.sql postgres
```

資料庫伺服器版本升級

關於資料庫伺服器版本升級可分為兩種，次要版本升級如 9.0 升級至 9.0.1 或 9.0.2 而主要版本升級(也稱為 In-Place Upgrade) 如 8.4 升級至 9.0 或 9.0 升級至 9.1，尤其特別注意 PostgreSQL9.0 後提供一工具 pg_upgrade 協助我們升級主要版本。

準備開始

✧ 次要版本升級：

一般而言次要版本升級簡單許多，基本上只要將新版資料庫軟體安裝完成後，再透過新版資料庫軟體開啟資料庫實例即可，但仍需注意以下幾點：

1. 詳細閱讀版本釋出公告及注意事項，避免因升級而導致發生錯誤。
2. 根據指示開啟資料庫伺服器並執行相關升級後所需動作，完成指示動作後檢查是否運用正常。
3. 由於套用更新，如封存日誌格式異動，故在升級資料庫版本後須檢查資料庫複寫機制是否正常運作，甚至重新掃描資料庫或重建索引等動作，但多半升級並不需此動作，不過仍須特別注意。
4. 當使用資料庫複寫機制時，重啟備援資料庫伺服器並檢查複寫機制是否正常運作，最好方式是連備援資料庫伺服器版本一併升級
5. 當資料庫複寫機制一切正常，接著升級備援資料庫伺服器版本，升級方式與一般升級資料庫次要版本步驟相同。

✧ 主要版本升級(In-Place Upgrade)：

剛有提到在 PostgreSQL 9.0 版本後提供一工具 pg_upgrade 協助我們由 8.4 升級至 9.0 或 9.0 升級至 9.1。

In-Place Upgrade 意指升級資料庫伺服器版本時無須將資料庫使用邏輯方式如 pg_dump 搬移至異機來進行升級，待在本機上安裝新版資料庫軟體版本後直接執行 pg_upgrade 進行本機升級，執行前需注意以下幾點：

1. 執行 pg_upgrade 前須將資料庫伺服器關閉。

2. 須預留全部資料庫大小二倍空間，因進行 pg_upgrade 時會自動將資料庫伺服器中所有資料庫複製一份，一份為備份保留用途，另一份作為升級，如空間上的確不夠則可在執行 pg_upgrade 時可指定參數「link」，則進行 pg_upgrade 時不會自動備份資料庫伺服器中所有資料庫，但須特別注意當不幸升級失敗時並無法進行 Rollback 或 Downgrade，只能重新由資料庫備份檔倒回並再次執行 pg_upgrade。
3. pg_upgrade 只支援升級 PostgreSQL8.3 以上版本。

```
$pg_upgrade -d oldCluster/data -D newCluster/data -b oldCluster/bin -B
newCluster/bin
```

-d: 舊版資料庫資料目錄, -D: 新版資料庫資料目錄
-b: 舊版資料庫軟體版本, -B 新版資料庫軟體版本

如何運作…

在升級資料庫伺服器仍有以下幾項建議提供給大家參考：

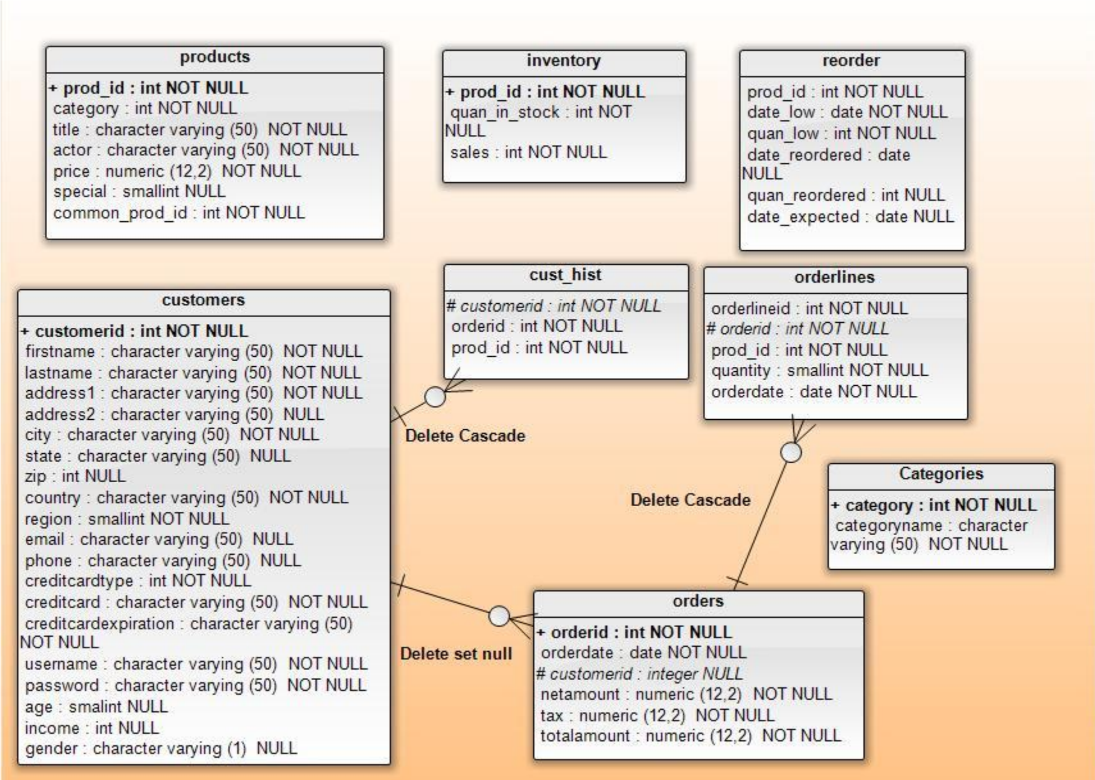
1. 詳細閱讀版本釋出公告及注意事項，尤其特別注意相容部份，新舊版本差異。
2. 在正式升級之前應先建立測試環境演練升級資料庫伺服器時所會遭遇到所有問題並解決在升級後是否有軟體相依性問題或資料毀損等相關問題。
3. 備份資料庫伺服器所有資料，以防升級失敗可隨時倒回資料。
4. 準備當升級失敗後所須還原的任何動作，確保可快速還原資料庫伺服器並進行服務。
5. 維持所有安全機制不應升級而取消，如 pg_hba 相關設定。
6. 升級完畢後應立即進行資料庫伺服器統計值收集及分析。
7. 當順利執行完 pg_upgrade 後，再次重新開啟資料庫伺服器並確認應用程式是否可順利連線。
8. 升級一切順利後，舊版本資料目錄應保留一段時間，避免日後發生問題時可將原有資料目錄倒回或檢查哪些升級步驟有異。

在升級資料庫伺服器主要版本時所需時間較常且步驟也較複雜，所以在升級之前應事先做好規劃或 SOP，以便在任何突然事件發生時可以隨時處理，唯有充份的準備才能順利進行升級並在意外發生時可將傷害降到最低，您說是嗎？

當資料庫空間非常巨大時，進行資料庫版本升級可能會花費相當多的時間，導致服務停擺，此時不如考慮使用資料庫複寫方式，於舊資料庫伺服器版本建立複寫至新版資料庫伺服器，藉由複寫機制逐步同步資料，待同步完成後，進行切換即可，可大大縮短升級時間並在同步資料時也不用停擺服務。

撈取資料表欄位

```
SELECT a.attname,b.relname,a.attnum
FROM PG_ATTRIBUTE a, PG_CLASS b
WHERE b.oid = a.attrelid
AND b.relname = 'pg_tables'
AND a.attname NOT IN ('tableoid', 'cmax', 'xmax', 'cmin','xmin', 'ctid')
order by attnum;
```


products
+ prod_id : int NOT NULL
category : int NOT NULL
title : character varying (50) NOT NULL
actor : character varying (50) NOT NULL
price : numeric (12,2) NOT NULL
special : smallint NULL
common_prod_id : int NOT NULL
inventory
+ prod_id : int NOT NULL
quan_in_stock : int NOT NULL
sales : int NOT NULL
reorder
prod_id : int NOT NULL
date_low : date NOT NULL
quan_low : int NOT NULL
date_reordered : date NULL
quan_reordered : int NULL
date_expected : date NULL
cust_hist
customerid : int NOT NULL
orderid : int NOT NULL
prod_id : int NOT NULL
orderlines
orderlineid : int NOT NULL
orderid : int NOT NULL
prod_id : int NOT NULL
quantity : smallint NOT NULL
orderdate : date NOT NULL
customers
+ customerid : int NOT NULL
firstname : character varying (50) NOT NULL
lastname : character varying (50) NOT NULL
address1 : character varying (50) NOT NULL
address2 : character varying (50) NULL
city : character varying (50) NOT NULL
state : character varying (50) NULL
zip : int NULL
country : character varying (50) NOT NULL
region : smallint NOT NULL
email : character varying (50) NULL
phone : character varying (50) NULL
creditcardtype : int NOT NULL
creditcard : character varying (50) NOT NULL
creditcardexpiration : character varying (50) NOT NULL
username : character varying (50) NOT NULL
password : character varying (50) NOT NULL
age : smalint NULL
income : int NULL
gender : character varying (1) NULL
Delete Cascade
Delete Cascade
Categories
+ category : int NOT NULL
categoryname : character varying (50) NOT NULL
Delete set null
orders
+ orderid : int NOT NULL
orderdate : date NOT NULL
customerid : integer NULL
netamount : numeric (12,2) NOT NULL
tax : numeric (12,2) NOT NULL
totalamount : numeric (12,2) NOT NULL