PostgreSQL: Introduction and Concepts

Bruce Momjian

19th February 2004

ADDISON–WESLEY

Boston • San Francisco • New York • Toronto • Montreal • London • Munich

Paris • Madrid • Cape Town • Sidney • Tokyo • Singapore • Mexico City

Pearson Education Corporate Sales Division
One Lake Street
Upper Saddle River, NJ 07458
(800) 382-3419
corpsales@pearsontechgroup.com
Visit AW on the Web: www.awl.com/cseng/

Library of Congress Cataloging-in-Publication Data

Momjian, Bruce.
PostgreSQL : introduction and concepts / Momjian, Bruce.
p. cm.
ISBN 0-201-70331-9
1. Database management. 2. PostgreSQL. I. Title.
QA76.9.D3 M647 2000

005.75'85–dc21 00-045367

CIP

This book was prepared with LyX and LaTeX and reproduced by Addison–Wesley from files supplied by the author.

Text printed on recycled and acid-free paper

2 3 4 5 6 7 8 9-MA-04030201

Second Printing, February 2001

Chapter 1

History of POSTGRESQL

Chapter 2

Issuing Database Commands

```
$ psql test
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

test=>
```

Figure 2.1: `psql` session start-up

```
test=> SELECT CURRENT_USER;
 getpgusername
---------------
 postgres
(1 row)

test=>
```

Figure 2.2: My first SQL query

```
test=> SELECT
test-> 1 + 3
test-> ;
 ?column?
----------
        4
(1 row)

test=>
```

Figure 2.3: Multiline query

```
test=> SELECT
test-> 2 * 10 + 1
test-> \p
SELECT
2 * 10 + 1
test-> \g
 ?column?
----------
       21
(1 row)

test=>
```

Figure 2.4: *Backslash-p* demo

Chapter 3

Basic SQL Commands

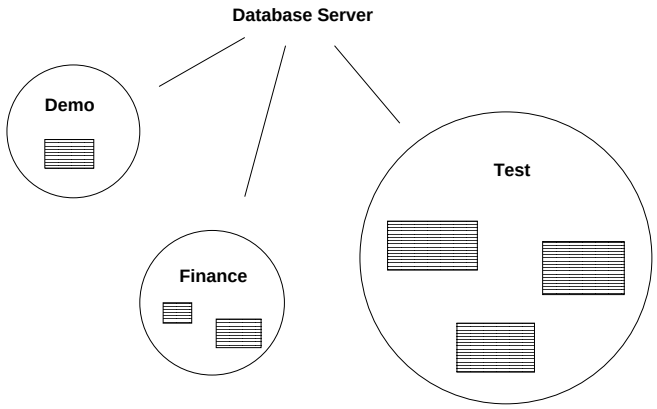

Figure 3.1: Databases

FirstName	LastName	City	State	Age
Mike	Nichols	Tampa	FL	19
Cindy	Anderson	Denver	CO	23
Sam	Jackson	Allentown	PA	22

Table 3.1: Table *friend*

```
test=> CREATE TABLE friend (
test(>              firstname CHAR(15),
test(>              lastname  CHAR(20),
test(>              city      CHAR(15),
test(>              state     CHAR(2),
test(>              age       INTEGER
test(> );
CREATE
```

Figure 3.2: Create table *friend*

```
test=> \d friend
         Table "friend"
 Attribute |   Type   | Modifier
-----------+----------+----------
 firstname | char(15) |
 lastname  | char(20) |
 city      | char(15) |
 state     | char(2)  |
 age       | integer  |
```

Figure 3.3: Example of *backslash-d*

```
test=> INSERT INTO friend VALUES (
test(>                            'Mike',
test(>                            'Nichols',
test(>                            'Tampa',
test(>                            'FL',
test(>                            19
test(> );
INSERT 19053 1
```

Figure 3.4: INSERT into *friend*

```
test=> INSERT INTO friend VALUES (
test(>                            'Cindy',
test(>                            'Anderson',
test(>                            'Denver',
test(>                            'CO',
test(>                            23
test(> );
INSERT 19054 1
test=> INSERT INTO friend VALUES (
test(>                            'Sam',
test(>                            'Jackson',
test(>                            'Allentown',
test(>                            'PA',
test(>                            22
test(> );
INSERT 19055 1
```

Figure 3.5: Additional *friend* INSERT commands

```
test=> SELECT * FROM friend;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Mike            | Nichols              | Tampa           | FL    |  19
 Cindy           | Anderson             | Denver          | CO    |  23
 Sam             | Jackson              | Allentown       | PA    |  22
(3 rows)
```

Figure 3.6: My first SELECT

```
test=> SELECT * FROM friend WHERE age = 23;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Cindy           | Anderson             | Denver          | CO    |  23
(1 row)
```

Figure 3.7: My first WHERE

```
test=> SELECT * FROM friend WHERE age <= 22;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Mike            | Nichols              | Tampa           | FL    |  19
 Sam             | Jackson              | Allentown       | PA    |  22
(2 rows)
```

Figure 3.8: More complex WHERE clause

```
test=> SELECT lastname FROM friend WHERE age = 22;
       lastname
----------------------
 Jackson
(1 row)
```

Figure 3.9: A single cell

```
test=> SELECT city, state FROM friend WHERE age >= 21;
      city       | state
-----------------+-------
 Denver          | CO
 Allentown       | PA
(2 rows)
```

Figure 3.10: A block of cells

```
test=> SELECT * FROM friend WHERE firstname = 'Sam';
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Sam             | Jackson              | Allentown       | PA    |  22
(1 row)
```

Figure 3.11: Comparing string fields

```
test=> SELECT * FROM friend;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Mike            | Nichols              | Tampa           | FL    |  19
 Cindy           | Anderson             | Denver          | CO    |  23
 Sam             | Jackson              | Allentown       | PA    |  22
(3 rows)

test=> INSERT INTO friend VALUES ('Jim', 'Barnes', 'Ocean City','NJ', 25);
INSERT 19056 1
test=> SELECT * FROM friend;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Mike            | Nichols              | Tampa           | FL    |  19
 Cindy           | Anderson             | Denver          | CO    |  23
 Sam             | Jackson              | Allentown       | PA    |  22
 Jim             | Barnes               | Ocean City      | NJ    |  25
(4 rows)

test=> DELETE FROM friend WHERE lastname = 'Barnes';
DELETE 1
test=> SELECT * FROM friend;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Mike            | Nichols              | Tampa           | FL    |  19
 Cindy           | Anderson             | Denver          | CO    |  23
 Sam             | Jackson              | Allentown       | PA    |  22
(3 rows)
```

Figure 3.12: DELETE example

```
test=> UPDATE friend SET age = 20 WHERE firstname = 'Mike';
UPDATE 1
test=> SELECT * FROM friend;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Cindy           | Anderson             | Denver          | CO    |  23
 Sam             | Jackson              | Allentown       | PA    |  22
 Mike            | Nichols              | Tampa           | FL    |  20
(3 rows)
```

Figure 3.13: My first UPDATE

```
test=> SELECT * FROM friend ORDER BY state;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Cindy           | Anderson             | Denver          | CO    |  23
 Mike            | Nichols              | Tampa           | FL    |  20
 Sam             | Jackson              | Allentown       | PA    |  22
(3 rows)
```

Figure 3.14: Use of ORDER BY

```
test=> SELECT * FROM friend ORDER BY age DESC;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Cindy           | Anderson             | Denver          | CO    |  23
 Sam             | Jackson              | Allentown       | PA    |  22
 Mike            | Nichols              | Tampa           | FL    |  20
(3 rows)
```

Figure 3.15: Reverse ORDER BY

```
test=> SELECT * FROM friend WHERE age >= 21 ORDER BY firstname;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Cindy           | Anderson             | Denver          | CO    |  23
 Sam             | Jackson              | Allentown       | PA    |  22
(2 rows)
```

Figure 3.16: Use of ORDER BY and WHERE

Chapter 4

Customizing Queries

Category	Type	Description
character string	CHAR(length)	blank-padded string, fixed storage length
	VARCHAR(length)	variable storage length
number	INTEGER	integer, +/–2 billion range
	FLOAT	floating point number, 15-digit precision
	NUMERIC(precision, decimal)	number with user-defined precision and decimal location
date/time	DATE	date
	TIME	time
	TIMESTAMP	date and time

Table 4.1: Common data types

```
test=> CREATE TABLE alltypes (
test(>                  state CHAR(2),
test(>                  name CHAR(30),
test(>                  children INTEGER,
test(>                  distance FLOAT,
test(>                  budget NUMERIC(16,2),
test(>                  born DATE,
test(>                  checkin TIME,
test(>                  started TIMESTAMP
test(> );
CREATE
test=> INSERT INTO alltypes
test-> VALUES (
test(>         'PA',
test(>         'Hilda Blairwood',
test(>          3,
test(>          10.7,
test(>          4308.20,
test(>          '9/8/1974',
test(>          '9:00',
test(>          '07/03/1996 10:30:00');
INSERT 19073 1
test=> SELECT state, name, children, distance, budget FROM alltypes;
 state |              name              | children | distance | budget
-------+--------------------------------+----------+----------+---------
 PA    | Hilda Blairwood                |        3 |     10.7 | 4308.20
(1 row)

test=> SELECT born, checkin, started FROM alltypes;
    born    | checkin  |        started
------------+----------+-----------------------
 1974-09-08 | 09:00:00 | 1996-07-03 10:30:00-04
(1 row)

test=> \x
Expanded display is on.
test=> SELECT * FROM alltypes;
-[ RECORD 1 ]--------------------------
state    | PA
name     | Hilda Blairwood
children | 3
distance | 10.7
budget   | 4308.20
born     | 1974-09-08
checkin  | 09:00:00
started  | 1996-07-03 10:30:00-04
```

Figure 4.1: Example of common data types

```
test=> INSERT INTO friend (firstname, lastname, city, state)
test-> VALUES ('Mark', 'Middleton', 'Indianapolis', 'IN');
INSERT 19074 1
```

Figure 4.2: Insertion of specific columns

```
test=> SELECT * FROM friend ORDER BY age DESC;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Cindy           | Anderson             | Denver          | CO    |  23
 Sam             | Jackson              | Allentown       | PA    |  22
 Mike            | Nichols              | Tampa           | FL    |  20
 Mark            | Middleton            | Indianapolis    | IN    |
(4 rows)

test=> SELECT * FROM friend WHERE age > 0 ORDER BY age DESC;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Cindy           | Anderson             | Denver          | CO    |  23
 Sam             | Jackson              | Allentown       | PA    |  22
 Mike            | Nichols              | Tampa           | FL    |  20
(3 rows)

test=> SELECT * FROM friend WHERE age <> 99 ORDER BY age DESC;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Cindy           | Anderson             | Denver          | CO    |  23
 Sam             | Jackson              | Allentown       | PA    |  22
 Mike            | Nichols              | Tampa           | FL    |  20
(3 rows)

test=> SELECT * FROM friend WHERE age IS NULL ORDER BY age DESC;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Mark            | Middleton            | Indianapolis    | IN    |
(1 row)
```

Figure 4.3: NULL handling

```
test=> INSERT INTO friend
test-> VALUES ('Jack', 'Burger', NULL, NULL, 27);
INSERT 19075 1
test=> SELECT * FROM friend WHERE city = state;
 firstname | lastname | city | state | age
-----------+----------+------+-------+-----
(0 rows)
```

Figure 4.4: Comparison of NULL fields

```
test=> CREATE TABLE nulltest (name CHAR(20), spouse CHAR(20));
CREATE
test=> INSERT INTO nulltest VALUES ('Andy', '');
INSERT 19086 1
test=> INSERT INTO nulltest VALUES ('Tom', NULL);
INSERT 19087 1
test=> SELECT * FROM nulltest ORDER BY name;
         name          |        spouse
-----------------------+----------------------
 Andy                  |
 Tom                   |
(2 rows)

test=> SELECT * FROM nulltest WHERE spouse = '';
         name          |        spouse
-----------------------+----------------------
 Andy                  |
(1 row)

test=> SELECT * FROM nulltest WHERE spouse IS NULL;
         name          | spouse
-----------------------+--------
 Tom                   |
(1 row)
```

Figure 4.5: NULL values and blank strings

```
test=> CREATE TABLE account (
test(>         name     CHAR(20),
test(>         balance  NUMERIC(16,2) DEFAULT 0,
test(>         active   CHAR(1) DEFAULT 'Y',
test(>         created  TIMESTAMP DEFAULT CURRENT_TIMESTAMP
test(> );
CREATE
test=> INSERT INTO account (name)
test-> VALUES ('Federated Builders');
INSERT 19103 1
test=> SELECT * FROM account;
         name         | balance | active |        created
----------------------+---------+--------+------------------------
 Federated Builders   |    0.00 | Y      | 1998-05-30 21:37:48-04
(1 row)
```

Figure 4.6: Using DEFAULT values

```
test=> SELECT firstname AS buddy FROM friend ORDER BY buddy;
      buddy
-----------------
 Cindy
 Jack
 Mark
 Mike
 Sam
(5 rows)
```

Figure 4.7: Controlling column labels

```
test=> SELECT 1 + 3 AS total;
 total
-------
     4
(1 row)
```

Figure 4.8: Computation using a column label

```
test=> -- a single line comment
test=> /* a multiline
test*>    comment */
```

Figure 4.9: Comment styles

```
test=> DELETE FROM friend;
DELETE 6
test=> INSERT INTO friend
test-> VALUES ('Dean', 'Yeager', 'Plymouth', 'MA', 24);
INSERT 19744 1
test=> INSERT INTO friend
test-> VALUES ('Dick', 'Gleason', 'Ocean City', 'NJ', 19);
INSERT 19745 1
test=> INSERT INTO friend
test-> VALUES ('Ned', 'Millstone', 'Cedar Creek', 'MD', 27);
INSERT 19746 1
test=> INSERT INTO friend
test-> VALUES ('Sandy', 'Gleason', 'Ocean City', 'NJ', 25);
INSERT 19747 1
test=> INSERT INTO friend
test-> VALUES ('Sandy', 'Weber', 'Boston', 'MA', 33);
INSERT 19748 1
test=> INSERT INTO friend
test-> VALUES ('Victor', 'Tabor', 'Williamsport', 'PA', 22);
INSERT 19749 1
test=> SELECT * FROM friend ORDER BY firstname;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Dean            | Yeager               | Plymouth        | MA    |  24
 Dick            | Gleason              | Ocean City      | NJ    |  19
 Ned             | Millstone            | Cedar Creek     | MD    |  27
 Sandy           | Gleason              | Ocean City      | NJ    |  25
 Sandy           | Weber                | Boston          | MA    |  33
 Victor          | Tabor                | Williamsport    | PA    |  22
(6 rows)
```

Figure 4.10: New friends

```
test=> SELECT * FROM friend
test-> WHERE firstname = 'Sandy' AND lastname = 'Gleason';
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Sandy           | Gleason              | Ocean City      | NJ    |  25
(1 row)
```

Figure 4.11: WHERE test for *Sandy Gleason*

```
test=> SELECT * FROM friend
test-> WHERE state = 'NJ' OR state = 'PA'
test-> ORDER BY firstname;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Dick            | Gleason              | Ocean City      | NJ    |  19
 Sandy           | Gleason              | Ocean City      | NJ    |  25
 Victor          | Tabor                | Williamsport    | PA    |  22
(3 rows)
```

Figure 4.12: Friends in New Jersey and Pennsylvania

Comparison	Operator
less than	<
less than or equal	<=
equal	=
greater than or equal	>=
greater than	>
not equal	<> or !=

Table 4.2: Comparison operators

```
test=> SELECT * FROM friend
test-> WHERE firstname = 'Victor' AND state = 'PA' OR state = 'NJ'
test-> ORDER BY firstname;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Dick            | Gleason              | Ocean City      | NJ    |  19
 Sandy           | Gleason              | Ocean City      | NJ    |  25
 Victor          | Tabor                | Williamsport    | PA    |  22
(3 rows)
```

Figure 4.13: Incorrectly mixing AND and OR clauses

```
test=> SELECT * FROM friend
test-> WHERE firstname = 'Victor' AND (state = 'PA' OR state = 'NJ')
test-> ORDER BY firstname;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Victor          | Tabor                | Williamsport    | PA    |  22
(1 row)
```

Figure 4.14: Correctly mixing AND and OR clauses

Comparison	Operation
begins with D	`LIKE 'D%'`
contains a D	`LIKE '%D%'`
has D in second position	`LIKE '_D%'`
begins with D and contains e	`LIKE 'D%e%'`
begins with D, contains e, then f	`LIKE 'D%e%f%'`
begins with non-D	`NOT LIKE 'D%'`

Table 4.3: LIKE comparisons

```
test=> SELECT *
test-> FROM friend
test-> WHERE age >= 22 AND age <= 25
test-> ORDER BY firstname;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Dean            | Yeager               | Plymouth        | MA    |  24
 Sandy           | Gleason              | Ocean City      | NJ    |  25
 Victor          | Tabor                | Williamsport    | PA    |  22
(3 rows)

test=> SELECT *
test-> FROM friend
test-> WHERE age BETWEEN 22 AND 25
test-> ORDER BY firstname;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Dean            | Yeager               | Plymouth        | MA    |  24
 Sandy           | Gleason              | Ocean City      | NJ    |  25
 Victor          | Tabor                | Williamsport    | PA    |  22
(3 rows)
```

Figure 4.15: Selecting a range of values

```
test=> SELECT * FROM friend
test-> WHERE firstname LIKE 'D%'
test-> ORDER BY firstname;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Dean            | Yeager               | Plymouth        | MA    |  24
 Dick            | Gleason              | Ocean City      | NJ    |  19
(2 rows)
```

Figure 4.16: *Firstname* begins with `D`

Comparison	Operator
regular expression	`~`
regular expression, case-insensitive	`~*`
not equal to regular expression	`!~`
not equal to regular expression, case-insensitive	`!~*`

Table 4.4: Regular expression operators

Test	Special Characters
start	`^`
end	`$`
any single character	`.`
set of characters	`[ccc]`
set of characters not equal	`[^ccc]`
range of characters	`[c-c]`
range of characters not equal	`[^c-c]`
zero or one of previous character	`?`
zero or multiple of previous characters	`*`
one or multiple of previous characters	`+`
OR operator	`|`

Table 4.5: Regular expression special characters

Test	Operation
begins with `D`	`~ '^D'`
contains `D`	`~ 'D'`
`D` in second position	`~ '^.D'`
begins with `D` and contains `e`	`~ '^D.*e'`
begins with `D`, contains `e`, and then `f`	`~ '^D.*e.*f'`
contains `A`, `B`, `C`, or `D`	`~ '[A-D]'` or `~ '[ABCD]'`
contains `A` or `a`	`~* 'a'` or `~ '[Aa]'`
does not contain `D`	`!~ 'D'`
does not begin with `D`	`!~ '^D'` or `~ '^[^D]'`
begins with `D`, with one optional leading space	`~ '^ ?D'`
begins with `D` , with optional leading spaces	`~ '^ *D'`
begins with `D`, with at least one leading space	`~ '^ +D'`
ends with `G`, with optional trailing spaces	`~ 'G *$'`

Table 4.6: Examples of regular expressions

```
test=> SELECT * FROM friend
test-> ORDER BY firstname;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Dean            | Yeager               | Plymouth        | MA    |  24
 Dick            | Gleason              | Ocean City      | NJ    |  19
 Ned             | Millstone            | Cedar Creek     | MD    |  27
 Sandy           | Gleason              | Ocean City      | NJ    |  25
 Sandy           | Weber                | Boston          | MA    |  33
 Victor          | Tabor                | Williamsport    | PA    |  22
(6 rows)

test=> -- firstname begins with 'S'
test=> SELECT * FROM friend
test-> WHERE firstname ~ '^S'
test-> ORDER BY firstname;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Sandy           | Gleason              | Ocean City      | NJ    |  25
 Sandy           | Weber                | Boston          | MA    |  33
(2 rows)

test=> -- firstname has an e in the second position
test=> SELECT * FROM friend
test-> WHERE firstname ~ '^.e'
test-> ORDER BY firstname;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Dean            | Yeager               | Plymouth        | MA    |  24
 Ned             | Millstone            | Cedar Creek     | MD    |  27
(2 rows)

test=> -- firstname contains b, B, c, or C
test=> SELECT * FROM friend
test-> WHERE firstname ~* '[bc]'
test-> ORDER BY firstname;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Dick            | Gleason              | Ocean City      | NJ    |  19
 Victor          | Tabor                | Williamsport    | PA    |  22
(2 rows)

test=> -- firstname does not contain s or S
test=> SELECT * FROM friend
test-> WHERE firstname !~* 's'
test-> ORDER BY firstname;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Dean            | Yeager               | Plymouth        | MA    |  24
 Dick            | Gleason              | Ocean City      | NJ    |  19
 Ned             | Millstone            | Cedar Creek     | MD    |  27
 Victor          | Tabor                | Williamsport    | PA    |  22
(4 rows)
```

Figure 4.17: Regular expression sample queries

```
test=> -- firstname ends with n
test=> SELECT * FROM friend
test-> WHERE firstname ~ 'n *$'
test-> ORDER BY firstname;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Dean            | Yeager               | Plymouth        | MA    |  24
(1 row)

test=> -- firstname contains a non-S character
test=> SELECT * FROM friend
test-> WHERE firstname ~ '[^S]'
test-> ORDER BY firstname;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Dean            | Yeager               | Plymouth        | MA    |  24
 Dick            | Gleason              | Ocean City      | NJ    |  19
 Ned             | Millstone            | Cedar Creek     | MD    |  27
 Sandy           | Gleason              | Ocean City      | NJ    |  25
 Sandy           | Weber                | Boston          | MA    |  33
 Victor          | Tabor                | Williamsport    | PA    |  22
(6 rows)
```

Figure 4.18: Complex regular expression queries

```
test=> SELECT firstname,
test->        age,
test->        CASE
test->            WHEN age >= 21 THEN 'adult'
test->            ELSE 'minor'
test->        END
test-> FROM friend
test-> ORDER BY firstname;
    firstname    | age | case
-----------------+-----+-------
 Dean            |  24 | adult
 Dick            |  19 | minor
 Ned             |  27 | adult
 Sandy           |  25 | adult
 Sandy           |  33 | adult
 Victor          |  22 | adult
(6 rows)
```

Figure 4.19: CASE example

```
test=> SELECT  firstname,
test->         state,
test->         CASE
test->                 WHEN state = 'PA' THEN 'close'
test->                 WHEN state = 'NJ' OR state = 'MD' THEN 'far'
test->                 ELSE 'very far'
test->         END AS distance
test-> FROM friend
test-> ORDER BY firstname;
    firstname    | state | distance
-----------------+-------+----------
 Dean            | MA    | very far
 Dick            | NJ    | far
 Ned             | MD    | far
 Sandy           | NJ    | far
 Sandy           | MA    | very far
 Victor          | PA    | close
(6 rows)
```

Figure 4.20: Complex CASE example

```
test=> SELECT state FROM friend ORDER BY state;
 state
-------
 MA
 MA
 MD
 NJ
 NJ
 PA
(6 rows)

test=> SELECT DISTINCT state FROM friend ORDER BY state;
 state
-------
 MA
 MD
 NJ
 PA
(4 rows)

test=> SELECT DISTINCT city, state FROM friend ORDER BY state, city;
      city       | state
-----------------+-------
 Boston          | MA
 Plymouth        | MA
 Cedar Creek     | MD
 Ocean City      | NJ
 Williamsport    | PA
(5 rows)
```

Figure 4.21: DISTINCT prevents duplicates

Function	SET option
DATESTYLE	DATESTYLE TO 'ISO' | 'POSTGRES' | 'SQL' | 'US' | 'NONEUROPEAN' | 'EUROPEAN' | 'GERMAN'
TIMEZONE	TIMEZONE TO '*value*'

Table 4.7: SET options

Style	Optional Ordering	Output for February 1, 1983
ISO		1983-02-01
POSTGRES	US or NONEUROPEAN	02-01-1983
POSTGRES	EUROPEAN	01-02-1983
SQL	US or NONEUROPEAN	02/01/1983
SQL	EUROPEAN	01/02/1983
German		01.02.1983

Table 4.8: DATESTYLE output

```
test=> \df
                              List of functions
  Result   |      Function       |                Arguments
-----------+---------------------+-----------------------------------------
 _bpchar   | _bpchar             | _bpchar int4
 _varchar  | _varchar            | _varchar int4
 float4    | abs                 | float4
 float8    | abs                 | float8


test=> \df int
                  List of functions
  Result  |     Function     |       Arguments
----------+------------------+------------------------
 int2     | int2             | float4
 int2     | int2             | float8
 int2     | int2             | int2
 int2     | int2             | int4


test=> \df upper
       List of functions
 Result | Function | Arguments
--------+----------+-----------
 text   | upper    | text
(1 row)

test=> \dd upper
      Object descriptions
 Name  |  Object  | Description
-------+----------+-------------
 upper | function | uppercase
(1 row)

test=> SELECT upper('jacket');
 upper
--------
 JACKET
(1 row)

test=> SELECT sqrt(2.0);  -- square root
      sqrt
-----------------
 1.4142135623731
(1 row)
```

Figure 4.22: Function examples

```
test=> \do
                                      List of operators
 Op  |  Left arg   |  Right arg  |  Result   |      Description
-----+-------------+-------------+-----------+---------------------------------
 !   | int2        |             | int4      |
 !   | int4        |             | int4      | factorial
 !   | int8        |             | int8      | factorial
 !!  |             | int2        | int4      |


test=> \do /
                           List of operators
 Op | Left arg | Right arg |  Result  |         Description
----+----------+-----------+----------+-----------------------------
 /  | box      | point     | box      | divide box by point (scale)
 /  | char     | char      | char     | divide
 /  | circle   | point     | circle   | divide
 /  | float4   | float4    | float4   | divide


test=> \do ^
                      List of operators
 Op | Left arg | Right arg | Result |     Description
----+----------+-----------+--------+---------------------
 ^  | float8   | float8    | float8 | exponentiation (x^y)
(1 row)

test=> \dd ^
          Object descriptions
 Name |  Object  |     Description
------+----------+---------------------
 ^    | operator | exponentiation (x^y)
(1 row)

test=> SELECT 2 + 3 ^ 4;
 ?column?
----------
       83
(1 row)
```

Figure 4.23: Operator examples

```
test=> SHOW DATESTYLE;
NOTICE: DateStyle is ISO with US (NonEuropean) conventions
SHOW VARIABLE
test=> SET DATESTYLE TO 'SQL, EUROPEAN';
SET VARIABLE
test=> SHOW DATESTYLE;
NOTICE:  DateStyle is SQL with European conventions
SHOW VARIABLE
test=> RESET DATESTYLE;
RESET VARIABLE
test=> SHOW DATESTYLE;
NOTICE:  DateStyle is ISO with US (NonEuropean) conventions
SHOW VARIABLE
```

Figure 4.24: SHOW and RESET examples

Chapter 5

SQL Aggregates

Aggregate	Function
COUNT(*)	count of rows
SUM(colname)	total
MAX(colname)	maximum
MIN(colname)	minimum
AVG(colname)	average

Table 5.1: Aggregates

```
test=> SELECT * FROM friend ORDER BY firstname;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Dean            | Yeager               | Plymouth        | MA    |  24
 Dick            | Gleason              | Ocean City      | NJ    |  19
 Ned             | Millstone            | Cedar Creek     | MD    |  27
 Sandy           | Gleason              | Ocean City      | NJ    |  25
 Sandy           | Weber                | Boston          | MA    |  33
 Victor          | Tabor                | Williamsport    | PA    |  22
(6 rows)

test=> SELECT COUNT(*) FROM friend;
 count
-------
     6
(1 row)

test=> SELECT SUM(age) FROM friend;
 sum
-----
 150
(1 row)

test=> SELECT MAX(age) FROM friend;
 max
-----
  33
(1 row)

test=> SELECT MIN(age) FROM friend;
 min
-----
  19
(1 row)

test=> SELECT AVG(age) FROM friend;
 avg
-----
  25
(1 row)
```

Figure 5.1: Examples of Aggregates

```
test=> CREATE TABLE aggtest (col INTEGER);
CREATE
test=> INSERT INTO aggtest VALUES (NULL);
INSERT 19759 1
test=> SELECT SUM(col) FROM aggtest;
 sum
-----

(1 row)

test=> SELECT MAX(col) FROM aggtest;
 max
-----

(1 row)

test=> SELECT COUNT(*) FROM aggtest;
 count
-------
     1
(1 row)

test=> SELECT COUNT(col) FROM aggtest;
 count
-------
     0
(1 row)

test=> INSERT INTO aggtest VALUES (3);
INSERT 19760 1
test=> SELECT AVG(col) FROM aggtest;
 avg
-----
   3
(1 row)

test=> SELECT COUNT(*) FROM aggtest;
 count
-------
     2
(1 row)

test=> SELECT COUNT(col) FROM aggtest;
 count
-------
     1
(1 row)
```

Figure 5.2: Aggregates and NULL values

```
test=> SELECT state, COUNT(*)
test-> FROM friend
test-> GROUP BY state;
 state | count
-------+-------
 MA    |     2
 MD    |     1
 NJ    |     2
 PA    |     1
(4 rows)

test=> SELECT state, MIN(age), MAX(age), AVG(age)
test-> FROM friend
test-> GROUP BY state
test-> ORDER BY 4 DESC;
 state | min | max | avg
-------+-----+-----+-----
 MA    |  24 |  33 |  28
 MD    |  27 |  27 |  27
 NJ    |  19 |  25 |  22
 PA    |  22 |  22 |  22
(4 rows)
```

Figure 5.3: Aggregate with GROUP BY

```
test=> SELECT city, state, COUNT(*)
test-> FROM friend
test-> GROUP BY state, city
test-> ORDER BY 1, 2;
      city       | state | count
-----------------+-------+-------
 Boston          | MA    |     1
 Cedar Creek     | MD    |     1
 Ocean City      | NJ    |     2
 Plymouth        | MA    |     1
 Williamsport    | PA    |     1
(5 rows)
```

Figure 5.4: GROUP BY with two columns

```
test=> SELECT state, COUNT(*)
test-> FROM friend
test-> GROUP BY state
test-> HAVING COUNT(*) > 1
test-> ORDER BY state;
 state | count
-------+-------
 MA    |     2
 NJ    |     2
(2 rows)
```

Figure 5.5: HAVING

Chapter 6

Joining Tables

```
test=> SELECT firstname FROM friend WHERE state = 'PA';
    firstname
-----------------
 Victor
(1 row)

test=> SELECT friend.firstname FROM friend WHERE friend.state = 'PA';
    firstname
-----------------
 Victor
(1 row)

test=> SELECT f.firstname FROM friend f WHERE f.state = 'PA';
    firstname
-----------------
 Victor
(1 row)
```

Figure 6.1: Qualified column names

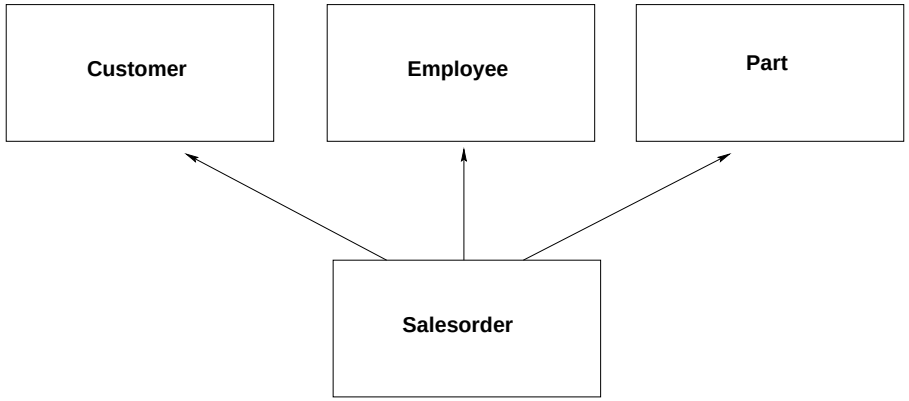

Figure 6.2: Joining tables

```
test=> CREATE TABLE customer (
test(>                        customer_id INTEGER,
test(>                        name        CHAR(30),
test(>                        telephone   CHAR(20),
test(>                        street      CHAR(40),
test(>                        city        CHAR(25),
test(>                        state       CHAR(2),
test(>                        zipcode     CHAR(10),
test(>                        country     CHAR(20)
test(> );
CREATE
test=> CREATE TABLE employee (
test(>                        employee_id INTEGER,
test(>                        name        CHAR(30),
test(>                        hire_date   DATE
test(> );
CREATE
test=> CREATE TABLE part (
test(>                    part_id     INTEGER,
test(>                    name        CHAR(30),
test(>                    cost        NUMERIC(8,2),
test(>                    weight      FLOAT
test(> );
CREATE
test=> CREATE TABLE salesorder (
test(>                 order_id      INTEGER,
test(>                 customer_id   INTEGER,  -- joins to customer.customer_id
test(>                 employee_id   INTEGER,  -- joins to employee.employee_id
test(>                 part_id       INTEGER,  -- joins to part.part_id
test(>                 order_date    DATE,
test(>                 ship_date     DATE,
test(>                 payment       NUMERIC(8,2)
test(> );
CREATE
```

Figure 6.3: Creation of company tables

```
test=> INSERT INTO customer VALUES (
test(>                               648,
test(>                               'Fleer Gearworks, Inc.',
test(>                               '1-610-555-7829',
test(>                               '830 Winding Way',
test(>                               'Millersville',
test(>                               'AL',
test(>                               '35041',
test(>                               'USA'
test(> );
INSERT 19815 1
test=> INSERT INTO employee VALUES (
test(>                               24,
test(>                               'Lee Meyers',
test(>                               '10/16/1989'
test(> );
INSERT 19816 1
test=> INSERT INTO part VALUES (
test(>                           153,
test(>                           'Garage Door Spring',
test(>                           6.20
test(> );
INSERT 19817 1
test=> INSERT INTO salesorder VALUES(
test(>                                14673,
test(>                                648,
test(>                                24,
test(>                                153,
test(>                                '7/19/1994',
test(>                                '7/28/1994',
test(>                                18.39
test(> );
INSERT 19818 1
```

Figure 6.4: Insertion into company tables

```
test=> SELECT customer_id FROM salesorder WHERE order_id = 14673;
 customer_id
-------------
         648
(1 row)

test=> SELECT name FROM customer WHERE customer_id = 648;
              name
--------------------------------
 Fleer Gearworks, Inc.
(1 row)
```

Figure 6.5: Finding a customer name using two queries

```
test=> SELECT customer.name                    -- query result
test-> FROM   customer, salesorder             -- query tables
test->                                         -- table join
test-> WHERE  customer.customer_id = salesorder.customer_id AND
test->        salesorder.order_id = 14673;     -- query restriction
              name
--------------------------------
 Fleer Gearworks, Inc.
(1 row)
```

Figure 6.6: Finding a customer name using one query

```
test=> SELECT salesorder.order_id
test-> FROM   salesorder, customer
test-> WHERE  customer.name = 'Fleer Gearworks, Inc.' AND
test->        salesorder.customer_id = customer.customer_id;
 order_id
----------
    14673
(1 row)
```

Figure 6.7: Finding an order number for a customer name

```
test=> SELECT customer.name, employee.name
test-> FROM   salesorder, customer, employee
test-> WHERE  salesorder.customer_id = customer.customer_id AND
test->        salesorder.employee_id = employee.employee_id AND
test->        salesorder.order_id = 14673;
              name              |              name
--------------------------------+--------------------------------
 Fleer Gearworks, Inc.          | Lee Meyers
(1 row)
```

Figure 6.8: Three-table join

```
test=> SELECT customer.name AS customer_name,
test->        employee.name AS employee_name,
test->        part.name AS part_name
test-> FROM   salesorder, customer, employee, part
test-> WHERE  salesorder.customer_id = customer.customer_id AND
test->        salesorder.employee_id = employee.employee_id AND
test->        salesorder.part_id = part.part_id AND
test->        salesorder.order_id = 14673;
        customer_name         |      employee_name       |      part_name
------------------------------+--------------------------+--------------------
 Fleer Gearworks, Inc.        | Lee Meyers               | Garage Door Spring
(1 row)
```

Figure 6.9: Four-table join

```
test=> SELECT DISTINCT customer.name, employee.name
test-> FROM   customer, employee, salesorder
test-> WHERE  customer.customer_id = salesorder.customer_id and
test->        salesorder.employee_id = employee.employee_id
test-> ORDER BY customer.name, employee.name;
              name              |              name
--------------------------------+--------------------------------
 Fleer Gearworks, Inc.          | Lee Meyers
(1 row)

test=> SELECT DISTINCT customer.name, employee.name, COUNT(*)
test-> FROM   customer, employee, salesorder
test-> WHERE  customer.customer_id = salesorder.customer_id and
test->        salesorder.employee_id = employee.employee_id
test-> GROUP BY customer.name, employee.name
test-> ORDER BY customer.name, employee.name;
              name              |              name              | count
--------------------------------+--------------------------------+-------
 Fleer Gearworks, Inc.          | Lee Meyers                     |     1
(1 row)
```

Figure 6.10: Employees who have taken orders for customers

```
SELECT employee.name
FROM   customer, employee
WHERE  customer.employee_id = employee.employee_id AND
       customer.customer_id = 648;

SELECT customer.name
FROM   customer, employee
WHERE  customer.employee_id = employee.employee_id AND
       employee.employee_id = 24
ORDER BY customer.name;
```

Figure 6.11: Joining *customer* and *employee*

```
-- find the employee assigned to part number 153
SELECT employee.name
FROM   part, employee
WHERE  part.employee_id = employee.employee_id AND
       part.part_id = 153;

-- find the parts assigned to employee 24
SELECT part.name
FROM   part, employee
WHERE  part.employee_id = employee.employee_id AND
       employee.employee_id = 24
ORDER BY name;
```

Figure 6.12: Joining *part* and *employee*

```
test=> CREATE TABLE statename (code CHAR(2),
test(>                          name  CHAR(30)
test(> );
CREATE
test=> INSERT INTO statename VALUES ('AL', 'Alabama');
INSERT 20629 1


test=> SELECT statename.name AS customer_statename
test-> FROM   customer, statename
test-> WHERE  customer.customer_id = 648 AND
test->        customer.state = statename.code;
```

Figure 6.13: The *statename* table

```
SELECT order_id
FROM   customer, salesorder
WHERE  customer.code = 'FLEOO1' AND
       customer.customer_id = salesorder.customer_id;
```

Figure 6.14: Using a customer code

```
test=> SELECT * FROM animal;
 animal_id |      name
-----------+-----------------
       507 | rabbit
       508 | cat
(2 rows)

test=> SELECT * FROM vegetable;
 animal_id |      name
-----------+-----------------
       507 | lettuce
       507 | carrot
       507 | nut
(3 rows)

test=> SELECT *
test-> FROM animal, vegetable
test-> WHERE animal.animal_id = vegetable.animal_id;
 animal_id |      name       | animal_id |      name
-----------+-----------------+-----------+-----------------
       507 | rabbit          |       507 | lettuce
       507 | rabbit          |       507 | carrot
       507 | rabbit          |       507 | nut
(3 rows)
```

Figure 6.15: A one-to-many join

```
test=> SELECT *
test-> FROM animal, vegetable;
 animal_id |      name       | animal_id |      name
-----------+-----------------+-----------+-----------------
       507 | rabbit          |       507 | lettuce
       508 | cat             |       507 | lettuce
       507 | rabbit          |       507 | carrot
       508 | cat             |       507 | carrot
       507 | rabbit          |       507 | nut
       508 | cat             |       507 | nut
(6 rows)
```

Figure 6.16: Unjoined tables

```
SELECT order_id
FROM   customer c, salesorder s
WHERE  c.code = 'FLEOO1' AND
       c.customer_id = s.customer_id;
```

Figure 6.17: Using table aliases

```
SELECT c2.name
FROM   customer c, customer c2
WHERE  c.customer_id = 648 AND
       c.zipcode = c2.zipcode;

SELECT c2.name, s.order_id
FROM customer c, customer c2, salesorder s
WHERE c.customer_id = 648 AND
      c.zipcode = c2.zipcode AND
      c2.customer_id = s.customer_id AND
      c2.customer_id <> 648;

SELECT c2.name, s.order_id, p.name
FROM   customer c, customer c2, salesorder s, part p
WHERE  c.customer_id = 648 AND
       c.zipcode = c2.zipcode AND
       c2.customer_id = s.customer_id AND
       s.part_id = p.part_id AND
       c2.customer_id <> 648;
```

Figure 6.18: Examples of self-joins using table aliases

```
SELECT c2.name
FROM   customer c, customer c2
WHERE  c.customer_id = 648 AND
       c.country <> c2.country
ORDER BY c2.name;

SELECT e2.name, e2.hire_date
FROM   employee e, employee e2
WHERE  e.employee_id = 24 AND
       e.hire_date < e2.hire_date
ORDER BY e2.hire_date, e2.name;

SELECT p2.name, p2.cost
FROM   part p, part p2
WHERE  p.part_id = 153 AND
       p.cost > p2.cost
ORDER BY p2.cost;
```

Figure 6.19: Non-equijoins

```
CREATE TABLE salesorder (
                     order_id      INTEGER,
                     customer_id   INTEGER,  -- joins to customer.customer_id
                     employee_id   INTEGER,  -- joins to employee.employee_id
                     order_date    DATE,
                     ship_date     DATE,
                     payment       NUMERIC(8,2)
);
```

Figure 6.20: New *salesorder* table for multiple parts per order

```
CREATE TABLE orderpart(
                        order_id INTEGER,
                        part_id  INTEGER,
                        quantity INTEGER DEFAULT 1
);
```

Figure 6.21: The *orderpart* table

```
-- first query
SELECT part.name
FROM   orderpart, part
WHERE  orderpart.part_id = part.part_id AND
       orderpart.order_id = 15398;

-- second query
SELECT part.name, orderpart.quantity
FROM salesorder, orderpart, part
WHERE salesorder.customer_id = 648 AND
      salesorder.order_date = '7/19/1994' AND
      salesorder.order_id = orderpart.order_id AND
      orderpart.part_id = part.part_id;

-- third query
SELECT part.name, part.cost, orderpart.quantity
FROM   customer, salesorder, orderpart, part
WHERE  customer.name = 'Fleer Gearworks, Inc.' AND
       salesorder.order_date = '7/19/1994' AND
       salesorder.customer_id = customer.customer_id AND
       salesorder.order_id = orderpart.order_id AND
       orderpart.part_id = part.part_id;

-- fourth query
SELECT SUM(part.cost * orderpart.quantity)
FROM   customer, salesorder, orderpart, part
WHERE  customer.name = 'Fleer Gearworks, Inc.' AND
       salesorder.order_date = '7/19/1994' AND
       salesorder.customer_id = customer.customer_id AND
       salesorder.order_id = orderpart.order_id AND
       orderpart.part_id = part.part_id;
```

Figure 6.22: Queries involving the *orderpart* table

Chapter 7

Numbering Rows

```
test=> CREATE TABLE oidtest(age INTEGER);
CREATE
test=> INSERT INTO oidtest VALUES (7);
INSERT 21515 1
test=> SELECT oid, age FROM oidtest;
  oid  | age
-------+-----
 21515 |   7
(1 row)
```

Figure 7.1: OID test

Function	Action
nextval('name')	Returns the next available sequence number, and updates the counter
currval('name')	Returns the sequence number from the previous *nextval()* call
setval('name', newval)	Sets the sequence number counter to the specified value

Table 7.1: Sequence number access functions

```
test=> CREATE TABLE salesorder (
test(>             order_id      INTEGER,
test(>             customer_oid   OID,  -- joins to customer.oid
test(>             employee_oid   OID,  -- joins to employee.oid
test(>             part_oid       OID,  -- joins to part.oid
```

Figure 7.2: Columns with OIDs

```
test=> CREATE SEQUENCE functest_seq;
CREATE
test=> SELECT nextval('functest_seq');
 nextval
---------
       1
(1 row)

test=> SELECT nextval('functest_seq');
 nextval
---------
       2
(1 row)

test=> SELECT currval('functest_seq');
 currval
---------
       2
(1 row)

test=> SELECT setval('functest_seq', 100);
 setval
--------
    100
(1 row)

test=> SELECT nextval('functest_seq');
 nextval
---------
     101
(1 row)
```

Figure 7.3: Examples of sequence function use

```
test=> CREATE SEQUENCE customer_seq;
CREATE
test=> CREATE TABLE customer (
test(>              customer_id INTEGER DEFAULT nextval('customer_seq'),
test(>              name CHAR(30)
test(> );
CREATE
test=> INSERT INTO customer VALUES (nextval('customer_seq'), 'Bread Makers');
INSERT 19004 1
test=> INSERT INTO customer (name) VALUES ('Wax Carvers');
INSERT 19005 1
test=> INSERT INTO customer (name) VALUES ('Pipe Fitters');
INSERT 19008 1
test=> SELECT * FROM customer;
 customer_id |              name
-------------+--------------------------------
           1 | Bread Makers
           2 | Wax Carvers
           3 | Pipe Fitters
(3 rows)
```

Figure 7.4: Numbering *customer* rows using a sequence

```
test=> CREATE TABLE customer (
test(>           customer_id SERIAL,
test(>           name CHAR(30)
test(> );
NOTICE:  CREATE TABLE will create implicit sequence 'customer_customer_id_seq' for SERIAL column 'customer.customer_id'
NOTICE:  CREATE TABLE/UNIQUE will create implicit index 'customer_customer_id_key' for table 'customer'
CREATE
test=> \d customer
                                Table "customer"
  Attribute  |   Type   |                           Extra
-------------+----------+-----------------------------------------------------------
 customer_id | int4     | not null default nextval('customer_customer_id_seq'::text)
 name        | char(30) |
Index: customer_customer_id_key
test=> INSERT INTO customer (name) VALUES ('Car Wash');
INSERT 19152 1
test=> SELECT * FROM customer;
 customer_id |              name
-------------+--------------------------------
           1 | Car Wash
(1 row)
```

Figure 7.5: The *customer* table using SERIAL

Chapter 8

Combining SELECTs

```
test=> SELECT firstname
test-> FROM friend
test-> UNION
test-> SELECT lastname
test-> FROM friend
test-> ORDER BY 1;
      firstname
----------------------
 Dean
 Dick
 Gleason
 Millstone
 Ned
 Sandy
 Tabor
 Victor
 Weber
 Yeager
(10 rows)
```

Figure 8.1: Combining two columns with UNION

```
test=> INSERT INTO terrestrial_animal (name) VALUES ('tiger');
INSERT 19122 1
test=> INSERT INTO aquatic_animal (name) VALUES ('swordfish');
INSERT 19123 1
test=> SELECT name
test-> FROM   aquatic_animal
test-> UNION
test-> SELECT name
test-> FROM   terrestrial_animal;
              name
--------------------------------
 swordfish
 tiger
(2 rows)
```

Figure 8.2: Combining two tables with UNION

```
test=> INSERT INTO aquatic_animal (name) VALUES ('penguin');
INSERT 19124 1
test=> INSERT INTO terrestrial_animal (name) VALUES ('penguin');
INSERT 19125 1
test=> SELECT name
test-> FROM   aquatic_animal
test-> UNION
test-> SELECT name
test-> FROM   terrestrial_animal;
              name
--------------------------------
 penguin
 swordfish
 tiger
(3 rows)
```

Figure 8.3: UNION with duplicates

```
test=> SELECT name
test-> FROM   aquatic_animal
test-> UNION ALL
test-> SELECT name
test-> FROM   terrestrial_animal;
              name
--------------------------------
 swordfish
 penguin
 tiger
 penguin
(4 rows)
```

Figure 8.4: UNION ALL with duplicates

```
test=> SELECT name
test-> FROM   aquatic_animal
test-> EXCEPT
test-> SELECT name
test-> FROM   terrestrial_animal;
              name
-------------------------------
 swordfish
(1 row)
```

Figure 8.5: EXCEPT restricts output from the first SELECT

```
test=> SELECT name
test-> FROM   aquatic_animal
test-> INTERSECT
test-> SELECT name
test-> FROM   terrestrial_animal;
              name
-------------------------------
 penguin
(1 row)
```

Figure 8.6: INTERSECT returns only duplicated rows

```
test=> SELECT * FROM friend ORDER BY firstname;
    firstname    |       lastname       |      city       | state | age
-----------------+----------------------+-----------------+-------+-----
 Dean            | Yeager               | Plymouth        | MA    |  24
 Dick            | Gleason              | Ocean City      | NJ    |  19
 Ned             | Millstone            | Cedar Creek     | MD    |  27
 Sandy           | Gleason              | Ocean City      | NJ    |  25
 Sandy           | Weber                | Boston          | MA    |  33
 Victor          | Tabor                | Williamsport    | PA    |  22
(6 rows)

test=> SELECT f1.firstname, f1.lastname, f1.state
test-> FROM   friend f1, friend f2
test-> WHERE  f1.state <> f2.state AND
test->        f2.firstname = 'Dick' AND
test->        f2.lastname = 'Gleason'
test-> ORDER BY firstname, lastname;
    firstname    |       lastname       | state
-----------------+----------------------+-------
 Dean            | Yeager               | MA
 Ned             | Millstone            | MD
 Sandy           | Weber                | MA
 Victor          | Tabor                | PA
(4 rows)

test=> SELECT f1.firstname, f1.lastname, f1.state
test-> FROM   friend f1
test-> WHERE  f1.state <> (
test(>                     SELECT f2.state
test(>                     FROM   friend f2
test(>                     WHERE  f2.firstname = 'Dick' AND
test(>                            f2.lastname = 'Gleason'
test(>                   )
test-> ORDER BY firstname, lastname;
    firstname    |       lastname       | state
-----------------+----------------------+-------
 Dean            | Yeager               | MA
 Ned             | Millstone            | MD
 Sandy           | Weber                | MA
 Victor          | Tabor                | PA
(4 rows)
```

Figure 8.7: Friends not in Dick Gleason's state

```
test=> SELECT name
test-> FROM   customer, salesorder
test-> WHERE  customer.customer_id = salesorder.customer_id AND
test->        salesorder.order_id = 14673;
              name
-------------------------------
 Fleer Gearworks, Inc.
(1 row)

test=> SELECT name
test-> FROM   customer
test-> WHERE  customer.customer_id = (
test(>                                SELECT salesorder.customer_id
test(>                                FROM salesorder
test(>                                WHERE order_id = 14673
test(>                               );
              name
-------------------------------
 Fleer Gearworks, Inc.
(1 row)
```

Figure 8.8: Subqueries can replace some joins

```
test=> SELECT f1.firstname, f1.lastname, f1.age
test-> FROM   friend f1, friend f2
test-> WHERE  f1.state = f2.state
test-> GROUP BY f2.state, f1.firstname, f1.lastname, f1.age
test-> HAVING f1.age = max(f2.age)
test-> ORDER BY firstname, lastname;
    firstname    |       lastname       | age
-----------------+----------------------+-----
 Ned             | Millstone            |  27
 Sandy           | Gleason              |  25
 Sandy           | Weber                |  33
 Victor          | Tabor                |  22
(4 rows)

test=> SELECT f1.firstname, f1.lastname, f1.age
test-> FROM   friend f1
test-> WHERE  age = (
test(>                SELECT MAX(f2.age)
test(>                FROM friend f2
test(>                WHERE f1.state = f2.state
test(>               )
test-> ORDER BY firstname, lastname;
    firstname    |       lastname       | age
-----------------+----------------------+-----
 Ned             | Millstone            |  27
 Sandy           | Gleason              |  25
 Sandy           | Weber                |  33
 Victor          | Tabor                |  22
(4 rows)
```

Figure 8.9: Correlated subquery

```
test=> SELECT DISTINCT employee.name
test-> FROM   employee, salesorder
test-> WHERE  employee.employee_id = salesorder.employee_id AND
test->        salesorder.order_date = '7/19/1994';
              name
-------------------------------
 Lee Meyers
(1 row)

test=> SELECT name
test-> FROM   employee
test-> WHERE  employee_id IN (
test(>                         SELECT employee_id
test(>                         FROM   salesorder
test(>                         WHERE  order_date = '7/19/1994'
test(>                        );
               name
-------------------------------
 Lee Meyers
(1 row)
```

Figure 8.10: Employees who took orders

```
test=> SELECT name
test-> FROM   customer
test-> WHERE  customer_id NOT IN (
test(>                            SELECT customer_id
test(>                            FROM salesorder
test(>                           );
 name
------
(0 rows)
```

Figure 8.11: Customers who have no orders

```
SELECT name
FROM   employee
WHERE  employee_id IN (
                       SELECT employee_id
                       FROM   salesorder
                       WHERE  order_date = '7/19/1994'
                      );

SELECT name
FROM   employee
WHERE  employee_id = ANY (
                       SELECT employee_id
                       FROM   salesorder
                       WHERE  order_date = '7/19/1994'
                      );

SELECT name
FROM   employee
WHERE  EXISTS (
                       SELECT employee_id
                       FROM   salesorder
                       WHERE  salesorder.employee_id = employee.employee_id AND
                              order_date = '7/19/1994'
                      );
```

Figure 8.12: IN query rewritten using ANY and EXISTS

```
SELECT name
FROM   customer
WHERE  customer_id NOT IN (
                           SELECT customer_id
                           FROM salesorder
                          );

SELECT name
FROM   customer
WHERE  customer_id <> ALL (
                           SELECT customer_id
                           FROM salesorder
                          );

SELECT name
FROM   customer
WHERE NOT EXISTS (
                           SELECT customer_id
                           FROM salesorder
                           WHERE salesorder.customer_id = customer.customer_id
                         );
```

Figure 8.13: NOT IN query rewritten using ALL and EXISTS

```
SELECT name, order_id
FROM   customer, salesorder
WHERE  customer.customer_id = salesorder.customer_id
UNION ALL
SELECT name, NULL
FROM   customer
WHERE  customer.customer_id NOT IN (SELECT customer_id FROM salesorder)
ORDER BY name;
```

Figure 8.14: Simulating outer joins

```
test=> DELETE FROM customer
test-> WHERE customer_id NOT IN (
test(>                           SELECT customer_id
test(>                           FROM salesorder
test(>                          );
DELETE 0
test=> UPDATE salesorder
test-> SET    ship_date = '11/16/96'
test-> WHERE  customer_id = (
test(>                       SELECT customer_id
test(>                       FROM   customer
test(>                       WHERE  name = 'Fleer Gearworks, Inc.'
test(>                      );
UPDATE 1
```

Figure 8.15: Subqueries with UPDATE and DELETE

```
UPDATE salesorder
SET    order_date = employee.hire_date
FROM   employee
WHERE  salesorder.employee_id = employee.employee_id AND
       salesorder.order_date < employee.hire_date;
```

Figure 8.16: UPDATE the *order_date*

```
test=> INSERT INTO customer (name, city, state, country)
test-> SELECT trim(firstname) || ' ' || lastname, city, state, 'USA'
test-> FROM friend;
INSERT 0 6
```

Figure 8.17: Using SELECT with INSERT

```
test=> SELECT firstname, lastname, city, state
test-> INTO   newfriend
test-> FROM   friend;
SELECT

test=> \d newfriend
      Table "newfriend"
 Attribute |   Type   | Extra
-----------+----------+-------
 firstname | char(15) |
 lastname  | char(20) |
 city      | char(15) |
 state     | char(2)  |

test=> SELECT * FROM newfriend ORDER BY firstname;
    firstname    |       lastname       |      city       | state
-----------------+----------------------+-----------------+-------
 Dean            | Yeager               | Plymouth        | MA
 Dick            | Gleason              | Ocean City      | NJ
 Ned             | Millstone            | Cedar Creek     | MD
 Sandy           | Gleason              | Ocean City      | NJ
 Sandy           | Weber                | Boston          | MA
 Victor          | Tabor                | Williamsport    | PA
(6 rows)
```

Figure 8.18: Table creation with SELECT

Chapter 9

Data Types

```
test=> SELECT * FROM functest;
 name
------
 Judy
(1 row)

test=> SELECT upper(name) FROM functest;
 upper
-------
 JUDY
(1 row)
```

Figure 9.1: Example of a function call

Category	Type	Description
Character string	TEXT	variable storage length
	VARCHAR(*length*)	variable storage length with maximum *length*
	CHAR(*length*)	fixed storage length, blank-padded to *length*, internally BPCHAR
Number	INTEGER	integer, ±2 billion range, internally INT4
	INT2	integer, ±32 thousand range
	INT8	integer, $\pm 4 \times 10^{18}$ range
	OID	object identifier
	NUMERIC(*precision*, *decimal*)	number, user-defined *precision* and *decimal* location
	FLOAT	floating-point number, 15-digit precision, internally FLOAT8
	FLOAT4	floating-point number, 6-digit precision
Temporal	DATE	date
	TIME	time
	TIMESTAMP	date and time
	INTERVAL	interval of time
Logical	BOOLEAN	boolean, *true* or *false*
Geometric	POINT	point
	LSEG	line segment
	PATH	list of points
	BOX	rectangle
	CIRCLE	circle
	POLYGON	polygon
Network	INET	IP address with optional netmask
	CIDR	IP network address
	MACADDR	Ethernet MAC address

Table 9.1: POSTGRESQL data types

Type	Example	Description
POINT	(2,7)	(*x*,*y*) coordinates
LSEG	[(0,0),(1,3)]	start and stop points of a line segment
PATH	((0,0),(3,0),(4,5),(1,6))	( ) is a closed path, [ ] is an open path
Box	(1,1),(3,3)	opposite corner points of a rectangle
CIRCLE	<(1,2),60>	center point and radius
POLYGON	((3,1),(3,3),(1,0))	points form closed polygon

Table 9.2: Geometric types

```
test=> SELECT date_part('year', '5/8/1971');
ERROR:  Function 'date_part(unknown, unknown)' does not exist
        Unable to identify a function that satisfies the given argument types
        You may need to add explicit typecasts
test=> SELECT date_part('year', CAST('5/8/1971' AS DATE));
 date_part
-----------
      1971
(1 row)
```

Figure 9.2: Error generated by undefined function/type combination.

Type	Function	Example	Returns
Character String	length()	length(*col*)	length of *col*
	character_length()	character_length(*col*)	length of *col*, same as length()
	octet_length()	octet_length(*col)*	length of *col*, including multibyte overhead
	trim()	trim(*col*)	*col* with leading and trailing spaces removed
	trim(BOTH…)	trim(BOTH *col*)	same as trim()
	trim(LEADING…)	trim(LEADING *col)*	*col* with leading spaces removed
	trim(TRAILING…)	trim(TRAILING *col)*	*col* with trailing spaces removed
	trim(…FROM…)	trim(*str* FROM *col)*	*col* with leading and trailing *str* removed
	rpad()	rpad(*col, len*)	*col* padded on the right to *len* characters
	rpad()	rpad(*col, len, str)*	*col* padded on the right using *str*
	lpad()	lpad(*col, len*)	*col* padded on the left to *len* characters
	lpad()	lpad(*col, len, str*)	*col* padded on the left using *str*
	upper()	upper(*col*)	*col* uppercased
	lower()	lower(*col*)	*col* lowercased
	initcap()	initcap(*col*)	*col* with the first letter capitalized
	strpos()	strpos(*col, str)*	position of *str* in *col*
	position()	position(*str* IN *col)*	same as strpos()
	substr()	substr(*col, pos)*	*col* starting at position *pos*
	substring(…FROM…)	substring(*col* FROM *pos)*	same as substr()
	substr()	substr(*col, pos*, *len)*	*col* starting at position *pos* for length *len*
	substring(…FROM… FOR…)	substring(*col* FROM pos FOR *len)*	same as substr()
	translate()	translate(*col, from, to*)	*col* with *from* characters mapped to *to*
	to_number()	to_number(*col, mask*)	convert *col* to NUMERIC() based on *mask*
	to_date()	to_date(*col, mask)*	convert *col* to DATE based on *mask*
	to_timestamp()	to_timestamp(*col, mask*)	convert *col* to TIMESTAMP based on *mask*
Number	round()	round(*col*)	round to an integer
	round()	round(*col, len)*	NUMERIC() *col* rounded to *len* decimal places
	trunc()	trunc(*col*)	truncate to an integer
	trunc()	trunc(*col, len)*	NUMERIC() *col* truncated to *len* decimal places
	abs()	abs(*col*)	absolute value
	factorial()	factorial(*col)*	factorial
	sqrt()	sqrt(*col*)	square root
	cbrt()	cbrt(*col*)	cube root
	exp()	exp(*col*)	exponential
	ln()	ln(*col*)	natural logarithm
	log()	log(*log*)	base-10 logarithm
	to_char()	to_char(*col, mask)*	convert *col* to a string based on *mask*
Temporal	date_part()	date_part(*units, col)*	*units* part of *col*
	extract(…FROM…)	extract(*units* FROM *col)*	same as date_part()
	date_trunc()	date_trunc(*units, col*)	*col* rounded to *units*
	isfinite()	isfinite(*col*)	BOOLEAN indicating whether *col* is a valid date
	now()	now()	TIMESTAMP representing current date and time
	timeofday()	timeofday()	string showing date/time in Unix format
	overlaps()	overlaps(*c1, c2, c3, c4*)	BOOLEAN indicating whether *col's* overlap in time
	to_char()	to_char(*col, mask*)	convert *col* to string based on *mask*
Geometric			see `psql`'s *\df* for a list of geometric functions
Network	broadcast()	broadcast(*col)*	broadcast address of *col*
	host()	host(*col*)	host address of *col*
	netmask()	netmask(*col*)	netmask of *col*
	masklen()	masklen(*col)*	mask length of *col*
	network()	network(*col)*	network address of *col*
NULL	nullif()	nullif(*col1, col2)*	return NULL if *col1* equals *col2,* else return *col1*
	coalesce()	coalesce(*col1, col2,…*)	return first non-NULL argument

Table 9.3: Common functions

Type	Function	Example	Returns
Character String	||	*col1* || *col2*	append *col2* on to the end of *col1*
	~	*col* ~ *pattern*	BOOLEAN, *col* matches regular expression *pattern*
	!~	*col* !~ *pattern*	BOOLEAN, *col* does not match regular expression *pattern*
	~*	*col* ~* *pattern*	same as ~, but case-insensitive
	!~*	*col* !~* *pattern*	same as !~, but case-insensitive
	~~	*col* ~~ *pattern*	BOOLEAN, *col* matches LIKE pattern
	LIKE	*col* LIKE *pattern*	same as ~~
	!~~	*col* !~~ *pattern*	BOOLEAN, *col* does not match LIKE pattern
	NOT LIKE	*col* NOT LIKE *pattern*	same as !~~
Number	!	!*col*	factorial
	+	*col1* + *col2*	addition
	–	*col1* – *col2*	subtraction
	*	*col1* * *col2*	multiplication
	/	*col1* / *col2*	division
	%	*col1* % *col2*	remainder/modulo
	^	*col1* ^ *col2*	*col1* raised to the power of *col2*
Temporal	+	*col1* + *col2*	addition of temporal values
	–	*col1* – *col2*	subtraction of temporal values
	(…) OVERLAPS (…)	*(c1, c2)* OVERLAPS *(c3, c4)*	BOOLEAN indicating *cols* overlap in time
Geometric			see `psql`'s *\do* for a list of geometric operators
Network	<<	*col1* << *col2*	BOOLEAN indicating if *col1* is a subnet of *col2*
	<<=	*col1* <<= *col2*	BOOLEAN indicating if *col1* is equal or a subnet of *col2*
	>>	*col1* >> *col2*	BOOLEAN indicating if *col1* is a supernet of *col2*
	>>=	*col1* >>= *col2*	BOOLEAN indicating if *col1* is equal or a supernet of *col2*

Table 9.4: Common operators

Variable	Meaning
CURRENT_DATE	current date
CURRENT_TIME	current time
CURRENT_TIMESTAMP	current date and time
CURRENT_USER	user connected to the database

Table 9.5: Common variables

```
test=> SELECT CAST('1/1/1992' AS DATE) + CAST('1/1/1993' AS DATE);
ERROR:  Unable to identify an operator '+' for types 'date' and 'date'
        You will have to retype this query using an explicit cast
test=> SELECT CAST('1/1/1992' AS DATE) + CAST('1 year' AS INTERVAL);
        ?column?
------------------------
 1993-01-01 00:00:00-05
(1 row)

test=> SELECT CAST('1/1/1992' AS TIMESTAMP) + '1 year';
        ?column?
------------------------
 1993-01-01 00:00:00-05
(1 row)
```

Figure 9.3: Error generated by undefined operator/type combination

```
test=> CREATE TABLE array_test (
test(>                          col1  INTEGER[5],
test(>                          col2  INTEGER[][],
test(>                          col3  INTEGER[2][2][]
test(> );
CREATE
```

Figure 9.4: Creation of array columns

```
test=> INSERT INTO array_test VALUES (
test(>                                '{1,2,3,4,5}',
test(>                                '{{1,2},{3,4}}',
test(>                                '{{{1,2},{3,4}},{{5,6}, {7,8}}}'
test(> );
INSERT 52694 1
test=> SELECT * FROM array_test;
    col1     |     col2      |             col3
-------------+---------------+-------------------------------
 {1,2,3,4,5} | {{1,2},{3,4}} | {{{1,2},{3,4}},{{5,6},{7,8}}}
(1 row)

test=> SELECT col1[4] FROM array_test;
 col1
------
    4
(1 row)

test=> SELECT col2[2][1] FROM array_test;
 col2
------
    3
(1 row)

test=> SELECT col3[1][2][2] FROM array_test;
 col3
------
    4
(1 row)
```

Figure 9.5: Using arrays

```
test=> CREATE TABLE fruit (name CHAR(30), image OID);
CREATE
test=> INSERT INTO fruit
test-> VALUES ('peach', lo_import('/usr/images/peach.jpg'));
INSERT 27111 1
test=> SELECT lo_export(fruit.image, '/tmp/outimage.jpg')
test-> FROM   fruit
test-> WHERE  name = 'peach';
 lo_export
-----------
         1
(1 row)

test=> SELECT lo_unlink(fruit.image) FROM fruit;
 lo_unlink
-----------
         1
(1 row)
```

Figure 9.6: Using large images

Chapter 10

Transactions and Locks

```
test=> INSERT INTO trans_test VALUES (1);
INSERT 130057 1
```

Figure 10.1: INSERT with no explicit transaction

```
test=> BEGIN WORK;
BEGIN
test=> INSERT INTO trans_test VALUES (1);
INSERT 130058 1
test=> COMMIT WORK;
COMMIT
```

Figure 10.2: INSERT using an explicit transaction

```
test=> BEGIN WORK;
BEGIN
test=> INSERT INTO trans_test VALUES (1);
INSERT 130059 1
test=> INSERT INTO trans_test VALUES (2);
INSERT 130060 1
test=> COMMIT WORK;
COMMIT
```

Figure 10.3: Two INSERTs in a single transaction

```
test=> BEGIN WORK;
BEGIN
test=> UPDATE bankacct SET balance = balance - 100 WHERE acctno = '82021';
UPDATE 1
test=> UPDATE bankacct SET balance = balance + 100 WHERE acctno = '96814';
UPDATE 1
test=> COMMIT WORK;
COMMIT
```

Figure 10.4: Multistatement transaction

User 1	User 2	Description
	SELECT (*) FROM *trans_test*	returns 0
INSERT INTO *trans_test* VALUES (1)		add row to *trans_test*
SELECT (*) FROM *trans_test*		returns 1
	SELECT (*) FROM *trans_test*	returns 1

Table 10.1: Visibility of single-query transactions

```
test=> INSERT INTO rollback_test VALUES (1);
INSERT 19369 1
test=> BEGIN WORK;
BEGIN
test=> DELETE FROM rollback_test;
DELETE 1
test=> ROLLBACK WORK;
ROLLBACK
test=> SELECT * FROM rollback_test;
 x
---
 1
(1 row)
```

Figure 10.5: Transaction rollback

User 1	User 2	Description
BEGIN WORK		User 1 starts a transaction
	SELECT (*) FROM *trans_test*	returns 0
INSERT INTO *trans_test* VALUES (1)		add row to *trans_test*
SELECT (*) FROM *trans_test*		returns 1
	SELECT (*) FROM *trans_test*	returns 0
COMMIT WORK		
	SELECT (*) FROM *trans_test*	returns 1

Table 10.2: Visibility of multiquery transactions

```
test=> BEGIN WORK;
BEGIN
test=> SELECT COUNT(*) FROM trans_test;
 count
-------
     5
(1 row)

test=> --
test=> -- someone commits INSERT INTO trans_test
test=> --
test=> SELECT COUNT(*) FROM trans_test;
 count
-------
     6
(1 row)

test=> COMMIT WORK;
COMMIT
```

Figure 10.6: Read-committed isolation level

Transaction 1	Transaction 2	Description
BEGIN WORK	BEGIN WORK	start both transactions
UPDATE row 64		transaction 1 exclusively locks row 64
	UPDATE row 64	transaction 2 must wait to see if transaction 1 commits
COMMIT WORK		transaction 1 commits; transaction 2 returns from UPDATE
	COMMIT WORK	transaction 2 commits

Table 10.3: Waiting for a lock

```
test=> BEGIN WORK;
BEGIN
test=> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET VARIABLE
test=> SELECT COUNT(*) FROM trans_test;
 count
-------
     5
(1 row)

test=> --
test=> -- someone commits INSERT INTO trans_test
test=> --
test=> SELECT COUNT(*) FROM trans_test;
 count
-------
     5
(1 row)

test=> COMMIT WORK;
COMMIT
```

Figure 10.7: Serializable isolation level

Transaction 1	Transaction 2	Description
BEGIN WORK	BEGIN WORK	start both transactions
UPDATE row 64	UPDATE row 83	independent rows write-locked
UPDATE row 83		holds waiting for transaction 2 to release write lock
	UPDATE row 64	attempt to get write lock held by transaction 1
	auto-ROLLBACK WORK	deadlock detected—transaction 2 is rolled back
COMMIT WORK		transaction 1 returns from UPDATE and commits

Table 10.4: Deadlock

```
test=> BEGIN WORK;
BEGIN
test=> SELECT *
test-> FROM lock_test
test-> WHERE name = 'James';
 id  |              name
-----+--------------------------------
 521 | James
(1 row)

test=> --
test=> -- the SELECTed row is not locked
test=> --
test=> UPDATE lock_test
test-> SET name = 'Jim'
test-> WHERE name = 'James';
UPDATE 1
test=> COMMIT WORK;
COMMIT
```

Figure 10.8: SELECT with no locking

```
test=> BEGIN WORK;
BEGIN
test=> SELECT *
test-> FROM lock_test
test-> WHERE name = 'James'
test-> FOR UPDATE;
 id  |              name
-----+--------------------------------
 521 | James
(1 row)

test=> --
test=> -- the SELECTed row is locked
test=> --
test=> UPDATE lock_test
test-> SET name = 'Jim'
test-> WHERE name = 'James';
UPDATE 1
test=> COMMIT WORK;
COMMIT
```

Figure 10.9: SELECT…FOR UPDATE

Chapter 11

Performance

```
test=> CREATE INDEX customer_custid_idx ON customer (customer_id);
CREATE
```

Figure 11.1: Example of CREATE INDEX

```
test=> CREATE TABLE duptest (channel INTEGER);
CREATE
test=> CREATE UNIQUE INDEX duptest_channel_idx ON duptest (channel);
CREATE
test=> INSERT INTO duptest VALUES (1);
INSERT 130220 1
test=> INSERT INTO duptest VALUES (1);
ERROR:  Cannot insert a duplicate key into unique index duptest_channel_idx
```

Figure 11.2: Example of a unique index

```
test=> EXPLAIN SELECT customer_id FROM customer;
NOTICE:  QUERY PLAN:

Seq Scan on customer  (cost=0.00..15.00 rows=1000 width=4)

EXPLAIN
```

Figure 11.3: Using EXPLAIN

```
test=> EXPLAIN SELECT customer_id FROM customer WHERE customer_id = 55;
NOTICE:  QUERY PLAN:

Seq Scan on customer  (cost=0.00..22.50 rows=10 width=4)

EXPLAIN
test=> VACUUM ANALYZE customer;
VACUUM
test=> EXPLAIN SELECT customer_id FROM customer WHERE customer_id = 55;
NOTICE:  QUERY PLAN:

Seq Scan on customer  (cost=0.00..17.50 rows=1 width=4)

EXPLAIN
test=> CREATE UNIQUE INDEX customer_custid_idx ON customer (customer_id);
CREATE
test=> EXPLAIN SELECT customer_id FROM customer WHERE customer_id = 55;
NOTICE:  QUERY PLAN:

Index Scan using customer_custid_idx on customer  (cost=0.00..2.01 rows=1 width=4)

EXPLAIN
test=> EXPLAIN SELECT customer_id FROM customer;
NOTICE:  QUERY PLAN:

Seq Scan on customer  (cost=0.00..15.00 rows=1000 width=4)

EXPLAIN
test=> EXPLAIN SELECT * FROM customer ORDER BY customer_id;
NOTICE:  QUERY PLAN:

Index Scan using customer_custid_idx on customer  (cost=0.00..42.00 rows=1000 width=4)

EXPLAIN
```

Figure 11.4: More complex EXPLAIN examples

```
test=> EXPLAIN SELECT * FROM tab1, tab2 WHERE col1 = col2;
NOTICE:  QUERY PLAN:

Merge Join  (cost=139.66..164.66 rows=10000 width=8)
  ->  Sort  (cost=69.83..69.83 rows=1000 width=4)
        ->  Seq Scan on tab2  (cost=0.00..20.00 rows=1000 width=4)
  ->  Sort  (cost=69.83..69.83 rows=1000 width=4)
        ->  Seq Scan on tab1  (cost=0.00..20.00 rows=1000 width=4)

EXPLAIN
```

Figure 11.5: EXPLAIN example using joins

Chapter 12

Controlling Results

```
test=> SELECT customer_id FROM customer ORDER BY customer_id LIMIT 3;
 customer_id
-------------
           1
           2
           3
(3 rows)

test=> SELECT customer_id FROM customer ORDER BY customer_id LIMIT 3 OFFSET 997;
 customer_id
-------------
         998
         999
        1000
(3 rows)
```

Figure 12.1: Examples of LIMIT and LIMIT/OFFSET

```
test=> BEGIN WORK;
BEGIN
test=> DECLARE customer_cursor CURSOR FOR
test-> SELECT customer_id FROM customer;
SELECT
test=> FETCH 1 FROM customer_cursor;
 customer_id
-------------
           1
(1 row)

test=> FETCH 1 FROM customer_cursor;
 customer_id
-------------
           2
(1 row)

test=> FETCH 2 FROM customer_cursor;
 customer_id
-------------
           3
           4
(2 rows)

test=> FETCH -1 FROM customer_cursor;
 customer_id
-------------
           3
(1 row)

test=> FETCH -1 FROM customer_cursor;
 customer_id
-------------
           2
(1 row)

test=> MOVE 10 FROM customer_cursor;
MOVE
test=> FETCH 1 FROM customer_cursor;
 customer_id
-------------
          13
(1 row)
test=> CLOSE customer_cursor;
CLOSE
test=> COMMIT WORK;
COMMIT
```

Figure 12.2: Cursor usage

Chapter 13

Table Management

User 1	User 2
CREATE TEMPORARY TABLE *temptest* (*col* INTEGER)	CREATE TEMPORARY TABLE *temptest* (*col* INTEGER)
INSERT INTO *temptes*t VALUES (1)	INSERT INTO *temptest* VALUES (2)
SELECT *col* FROM *temptest* returns 1	SELECT *col* FROM *temptest* returns 2

Table 13.1: Temporary table isolation

```
$ psql test
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

test=> CREATE TEMPORARY TABLE temptest(col INTEGER);
CREATE
test=> SELECT * FROM temptest;
 col
-----
(0 rows)

test=> \q
$ psql test
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

test=> SELECT * FROM temptest;
ERROR:  Relation 'temptest' does not exist
```

Figure 13.1: Temporary table auto-destruction

```
test=> SELECT *
test-> INTO TEMPORARY customer_pennsylvania
test-> FROM customer
test-> WHERE state = 'PA';
SELECT
test=> CREATE index customer_penna_custid_idx ON customer_pennsylvania (customer_id);
CREATE
```

Figure 13.2: Example of temporary table use

```
test=> CREATE TABLE altertest (col1 INTEGER);
CREATE
test=> ALTER TABLE altertest RENAME TO alterdemo;
ALTER
test=> ALTER TABLE alterdemo RENAME COLUMN col1 TO democol;
ALTER
test=> ALTER TABLE alterdemo ADD COLUMN col2 INTEGER;
ALTER
test=> -- show renamed table, renamed column, and new column
test=> \d alterdemo
       Table "alterdemo"
 Attribute |  Type   | Modifier
-----------+---------+----------
 democol   | integer |
 col2      | integer |

test=> ALTER TABLE alterdemo ALTER COLUMN col2 SET DEFAULT 0;
ALTER
test=> -- show new default value
test=> \d alterdemo
        Table "alterdemo"
 Attribute |  Type   | Modifier
-----------+---------+-----------
 democol   | integer |
 col2      | integer | default 0
test=> ALTER TABLE alterdemo ALTER COLUMN col2 DROP DEFAULT;
ALTER
```

Figure 13.3: ALTER TABLE examples

```
test=> CREATE TABLE permtest (col INTEGER);
CREATE
test=> -- now only the owner can use permtest
test->
test=> GRANT SELECT ON permtest TO meyers;
CHANGE
test=> -- now user 'meyers' can do SELECTs on permtest
test=>
test=> GRANT ALL ON permtest TO PUBLIC;
CHANGE
test=> -- now all users can perform all operations on permtest
test=>
```

Figure 13.4: Examples of the GRANT command

```
test=> CREATE TABLE parent_test (col1 INTEGER);
CREATE
test=> CREATE TABLE child_test (col2 INTEGER) INHERITS (parent_test);
CREATE
test=> \d parent_test
         Table "parent_test"
 Attribute |  Type   | Modifier
-----------+---------+----------
 col1      | integer |

test=> \d child_test
         Table "child_test"
 Attribute |  Type   | Modifier
-----------+---------+----------
 col1      | integer |
 col2      | integer |
```

Figure 13.5: Creation of inherited tables

```
test=> INSERT INTO parent_test VALUES (1);
INSERT 18837 1
test=> INSERT INTO child_test VALUES (2,3);
INSERT 18838 1
test=> SELECT * FROM parent_test;
 col1
------
    1
(1 row)

test=> SELECT * FROM child_test;
 col1 | col2
------+------
    2 |    3
(1 row)

test=> SELECT * FROM parent_test*;
 col1
------
    1
    2
(2 rows)
```

Figure 13.6: Accessing inherited tables

```
test=> CREATE TABLE grandchild_test (col3 INTEGER) INHERITS (child_test);
CREATE
test=> INSERT INTO grandchild_test VALUES (4, 5, 6);
INSERT 18853 1
test=> SELECT * FROM parent_test*;
 col1
------
    1
    2
    4
(3 rows)

test=> SELECT * FROM child_test*;
 col1 | col2
------+------
    2 |    3
    4 |    5
(2 rows)
```

Figure 13.7: Inheritance in layers

```
test=> CREATE VIEW customer_ohio AS
test-> SELECT *
test-> FROM customer
test-> WHERE state = 'OH';
CREATE 18908 1
test=>
test=> -- let sanders see only Ohio customers
test=> GRANT SELECT ON customer_ohio TO sanders;
CHANGE
test=>
test=> -- create view to show only certain columns
test=> CREATE VIEW customer_address AS
test-> SELECT customer_id, name, street, city, state, zipcode, country
test-> FROM customer;
CREATE 18909 1
test=>
test=> -- create view that combines fields from two tables
test=> CREATE VIEW customer_finance AS
test-> SELECT customer.customer_id, customer.name, finance.credit_limit
test-> FROM customer, finance
test-> WHERE customer.customer_id = finance.customer_id;
CREATE 18910 1
```

Figure 13.8: Examples of views

```
test=> CREATE TABLE ruletest (col INTEGER);
CREATE
test=> CREATE RULE ruletest_insert AS     -- rule name
test-> ON INSERT TO ruletest              -- INSERT rule
test-> DO INSTEAD                         -- DO INSTEAD-type rule
test->     NOTHING;                       -- ACTION is NOTHING
CREATE 18932 1
test=> INSERT INTO ruletest VALUES (1);
test=> SELECT * FROM ruletest;
 col
-----
(0 rows)
```

Figure 13.9: Rule to prevent an INSERT

```
test=> CREATE TABLE service_request (
test->                     customer_id INTEGER,
test->                     description text,
test->                     cre_user text DEFAULT CURRENT_USER,
test->                     cre_timestamp timestamp DEFAULT CURRENT_TIMESTAMP);
CREATE
test=> CREATE TABLE service_request_log (
test->                     customer_id INTEGER,
test->                     description text,
test->                     mod_type char(1),
test->                     mod_user text DEFAULT CURRENT_USER,
test->                     mod_timestamp timestamp DEFAULT CURRENT_TIMESTAMP);
CREATE
test=> CREATE RULE service_request_update AS     -- UPDATE rule
test-> ON UPDATE TO service_request
test-> DO
test->     INSERT INTO service_request_log (customer_id, description, mod_type)
test->     VALUES (old.customer_id, old.description, 'U');
CREATE 19670 1
test=> CREATE RULE service_request_delete AS     -- DELETE rule
test-> ON DELETE TO service_request
test-> DO
test->     INSERT INTO service_request_log (customer_id, description, mod_type)
test->     VALUES (old.customer_id, old.description, 'D');
CREATE 19671 1
```

Figure 13.10: Rules to log table changes

```
test=> INSERT INTO service_request (customer_id, description)
test-> VALUES (72321, 'Fix printing press');
INSERT 18808 1
test=> UPDATE service_request
test-> SET description = 'Fix large printing press'
test-> WHERE customer_id = 72321;
UPDATE 1
test=> DELETE FROM service_request
test-> WHERE customer_id = 72321;
DELETE 1
test=> SELECT *
test-> FROM service_request_log
test-> WHERE customer_id = 72321;
 customer_id |       description        | mod_type | mod_user |     mod_timestamp
-------------+--------------------------+----------+----------+------------------------
       72321 | Fix printing press       | U        | williams | 2000-04-09 07:13:07-04
       72321 | Fix large printing press | D        | matheson | 2000-04-10 12:47:20-04
(2 rows)
```

Figure 13.11: Use of rules to log table changes

```
test=> CREATE TABLE realtable (col INTEGER);
CREATE
test=> CREATE VIEW view_realtable AS SELECT * FROM realtable;
CREATE 407890 1
test=> INSERT INTO realtable VALUES (1);
INSERT 407891 1
test=> INSERT INTO view_realtable VALUES (2);
INSERT 407893 1
test=> SELECT * FROM realtable;
 col
-----
   1
(1 row)

test=> SELECT * FROM view_realtable;
 col
-----
   1
(1 row)
```

Figure 13.12: Views ignore table modifications

```
test=> CREATE RULE view_realtable_insert AS     -- INSERT rule
test-> ON INSERT TO view_realtable
test-> DO INSTEAD
test->     INSERT INTO realtable
test->     VALUES (new.col);
CREATE 407894 1
test=>
test=> CREATE RULE view_realtable_update AS     -- UPDATE rule
test-> ON UPDATE TO view_realtable
test-> DO INSTEAD
test->     UPDATE realtable
test->     SET col = new.col
test->     WHERE col = old.col;
CREATE 407901 1
test=>
test=> CREATE RULE view_realtable_delete AS     -- DELETE rule
test-> ON DELETE TO view_realtable
test-> DO INSTEAD
test->     DELETE FROM realtable
test->     WHERE col = old.col;
CREATE 407902 1
```

Figure 13.13: Rules to handle view modifications

```
test=> INSERT INTO view_realtable VALUES (3);
INSERT 407895 1
test=> SELECT * FROM view_realtable;
 col
-----
   1
   3
(2 rows)

test=> UPDATE view_realtable
test-> SET col = 4;
UPDATE 2
test=> SELECT * FROM view_realtable;
 col
-----
   4
   4
(2 rows)

test=> DELETE FROM view_realtable;
DELETE 2
test=> SELECT * FROM view_realtable;
 col
-----
(0 rows)
```

Figure 13.14: Example of rules that handle view modifications

Chapter 14

Constraints

```
test=> CREATE TABLE not_null_test (
test(>                              col1 INTEGER,
test(>                              col2 INTEGER NOT NULL
test(>                             );
CREATE
test=> INSERT INTO not_null_test
test-> VALUES (1, NULL);
ERROR:  ExecAppend: Fail to add null value in not null attribute col2
test=> INSERT INTO not_null_test (col1)
test-> VALUES (1);
ERROR:  ExecAppend: Fail to add null value in not null attribute col2
test=> INSERT INTO not_null_test VALUES (1, 1);
INSERT 174368 1
test=> UPDATE  not_null_test SET col2 = NULL;
ERROR:  ExecReplace: Fail to add null value in not null attribute col2
```

Figure 14.1: NOT NULL constraint

```
test=> CREATE TABLE not_null_with_default_test (
test(>                                          col1 INTEGER,
test(>                                          col2 INTEGER NOT NULL DEFAULT 5
test(>                                         );
CREATE
test=> INSERT INTO not_null_with_default_test (col1)
test-> VALUES (1);
INSERT 148520 1
test=> SELECT *
test-> FROM not_null_with_default_test;
 col1 | col2
------+------
    1 |    5
(1 row)
```

Figure 14.2: NOT NULL with DEFAULT constraint

```
test=> CREATE TABLE uniquetest (col1 INTEGER UNIQUE);
NOTICE:  CREATE TABLE/UNIQUE will create implicit index 'uniquetest_col1_key' for table 'uniquetest'
CREATE
test=> \d uniquetest
       Table "uniquetest"
 Attribute |  Type   | Modifier
-----------+---------+----------
 col1      | integer |
Index: uniquetest_col1_key

test=> INSERT INTO uniquetest VALUES (1);
INSERT 148620 1
test=> INSERT INTO uniquetest VALUES (1);
ERROR:  Cannot insert a duplicate key into unique index uniquetest_col1_key
test=> INSERT INTO uniquetest VALUES (NULL);
INSERT 148622 1
test=> INSERT INTO uniquetest VALUES (NULL);
INSERT
```

Figure 14.3: UNIQUE column constraint

```
test=> CREATE TABLE uniquetest2 (
test(>                           col1 INTEGER,
test(>                           col2 INTEGER,
test(>                           UNIQUE (col1, col2)
test(>                          );
NOTICE:  CREATE TABLE/UNIQUE will create implicit index 'uniquetest2_col1_key' for table 'uniquetest2'
```

Figure 14.4: Multicolumn UNIQUE constraint

```
test=> CREATE TABLE primarytest (col INTEGER PRIMARY KEY);
NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'primarytest_pkey' for table 'primarytest'
CREATE
test=> \d primarytest
      Table "primarytest"
 Attribute |  Type   | Modifier
-----------+---------+----------
 col       | integer | not null
Index: primarytest_pkey
```

Figure 14.5: Creation of a PRIMARY KEY column

```
test=> CREATE TABLE primarytest2 (
test(>                            col1 INTEGER,
test(>                            col2 INTEGER,
test(>                            PRIMARY KEY(col1, col2)
test(>                           );
NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'primarytest2_pkey' for table 'primarytest2'
CREATE
```

Figure 14.6: Example of a multicolumn PRIMARY KEY

```
test=>  CREATE TABLE statename (code CHAR(2) PRIMARY KEY,
test(>                           name  CHAR(30)
test(> );
CREATE
test=> INSERT INTO statename VALUES ('AL', 'Alabama');
INSERT 18934 1


test=> CREATE TABLE customer (
test(>                         customer_id INTEGER,
test(>                         name        CHAR(30),
test(>                         telephone   CHAR(20),
test(>                         street      CHAR(40),
test(>                         city        CHAR(25),
test(>                         state       CHAR(2) REFERENCES statename,
test(>                         zipcode     CHAR(10),
test(>                         country     CHAR(20)
test(> );
CREATE
```

Figure 14.7: Foreign key creation

```
test=> INSERT INTO customer (state)
test-> VALUES ('AL');
INSERT 148732 1
test=> INSERT INTO customer (state)
test-> VALUES ('XX');
ERROR:  <unnamed> referential integrity violation - key referenced from customer not found in statename
```

Figure 14.8: Foreign key constraints

```
test=> CREATE TABLE customer (
test(>                        customer_id INTEGER PRIMARY KEY,
test(>                        name        CHAR(30),
test(>                        telephone   CHAR(20),
test(>                        street      CHAR(40),
test(>                        city        CHAR(25),
test(>                        state       CHAR(2),
test(>                        zipcode     CHAR(10),
test(>                        country     CHAR(20)
test(> );
CREATE
test=> CREATE TABLE employee (
test(>                        employee_id INTEGER PRIMARY KEY,
test(>                        name        CHAR(30),
test(>                        hire_date   DATE
test(> );
CREATE
test=> CREATE TABLE part (
test(>                    part_id     INTEGER PRIMARY KEY,
test(>                    name        CHAR(30),
test(>                    cost        NUMERIC(8,2),
test(>                    weight      FLOAT
test(> );
CREATE
test=> CREATE TABLE salesorder (
test(>                          order_id      INTEGER,
test(>                          customer_id   INTEGER REFERENCES customer,
test(>                          employee_id   INTEGER REFERENCES employee,
test(>                          part_id       INTEGER REFERENCES part,
test(>                          order_date    DATE,
test(>                          ship_date     DATE,
test(>                          payment       NUMERIC(8,2)
test(> );
CREATE
```

Figure 14.9: Creation of company tables using primary and foreign keys

```
test=> CREATE TABLE customer (
test(>                         customer_id INTEGER,
test(>                         name        CHAR(30),
test(>                         telephone   CHAR(20),
test(>                         street      CHAR(40),
test(>                         city        CHAR(25),
test(>                         state       CHAR(2) REFERENCES statename
test(>                                             ON UPDATE CASCADE
test(>                                             ON DELETE SET NULL,
test(>                         zipcode     CHAR(10),
test(>                         country     CHAR(20)
test(> );
CREATE
```

Figure 14.10: *Customer* table with foreign key actions

```
test=> CREATE TABLE primarytest (col INTEGER PRIMARY KEY);
NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'primarytest_pkey' for table 'primarytest'
CREATE
test=> CREATE TABLE foreigntest (
test(>                           col2 INTEGER REFERENCES primarytest
test(>                           ON UPDATE CASCADE
test(>                           ON DELETE NO ACTION
test(>                          );
NOTICE:  CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
CREATE
test=> INSERT INTO primarytest values (1);
INSERT 148835 1
test=> INSERT INTO foreigntest values (1);
INSERT 148836 1
test=>
test=> -- CASCADE UPDATE is performed
test=>
test=> UPDATE primarytest SET col = 2;
UPDATE 1
test=> SELECT * FROM foreigntest;
 col2
------
    2
(1 row)

test=>
test=> -- NO ACTION prevents deletion
test=>
test=> DELETE FROM primarytest;
ERROR:  <unnamed> referential integrity violation - key in primarytest still referenced from foreigntest
test=>
test=> -- By deleting the foreign key first, the DELETE succeeds
test=>
test=> DELETE FROM foreigntest;
DELETE 1
test=> DELETE FROM primarytest;
DELETE 1
```

Figure 14.11: Foreign key actions

```
test=> CREATE TABLE primarytest2 (
test(>                            col1 INTEGER,
test(>                            col2 INTEGER,
test(>                            PRIMARY KEY(col1, col2)
test(>                           );
NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'primarytest2_pkey' for table 'primarytest2'
CREATE
test=> CREATE TABLE foreigntest2 (col3 INTEGER,
test(>                            col4 INTEGER,
test(>                            FOREIGN KEY (col3, col4) REFERENCES primarytest2
test->                           );
NOTICE:  CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
CREATE
```

Figure 14.12: Example of a multicolumn foreign key

```
test=> INSERT INTO primarytest2
test-> VALUES (1,2);
INSERT 148816 1
test=> INSERT INTO foreigntest2
test-> VALUES (1,2);
INSERT 148817 1
test=> UPDATE foreigntest2
test-> SET col4 = NULL;
UPDATE 1
test=> CREATE TABLE matchtest (
test(>                    col3 INTEGER,
test(>                    col4 INTEGER,
test(>                    FOREIGN KEY (col3, col4) REFERENCES primarytest2
test(>                                             MATCH FULL
test(>                        );
NOTICE:  CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
CREATE
test=> UPDATE matchtest
test-> SET col3 = NULL, col4 = NULL;
UPDATE 1
test=> UPDATE matchtest
test-> SET col4 = NULL;
ERROR:  <unnamed> referential integrity violation - MATCH FULL doesn't allow mixing of NULL and NON-NULL key values
```

Figure 14.13: MATCH FULL foreign key

```
test=> CREATE TABLE defertest(
test(>                          col2 INTEGER REFERENCES primarytest
test(>                                       DEFERRABLE
test(> );
NOTICE:  CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
CREATE
test=> BEGIN;
BEGIN
test=> -- INSERT is attempted in non-DEFERRABLE mode
test=>
test=> INSERT INTO defertest VALUES (5);
ERROR:  <unnamed> referential integrity violation - key referenced from defertest not found in primarytest
test=> COMMIT;
COMMIT
test=> BEGIN;
BEGIN
test=> -- all foreign key constraints are set to DEFERRED
test=>
test=> SET CONSTRAINTS ALL DEFERRED;
SET CONSTRAINTS
test=> INSERT INTO defertest VALUES (5);
INSERT 148946 1
test=> INSERT INTO primarytest VALUES (5);
INSERT 148947 1
test=> COMMIT;
COMMIT
```

Figure 14.14: DEFERRABLE foreign key constraint

```
test=> CREATE TABLE friend2 (
test(>              firstname CHAR(15),
test(>              lastname  CHAR(20),
test(>              city      CHAR(15),
test(>              state     CHAR(2)      CHECK (length(trim(state)) = 2),
test(>              age       INTEGER      CHECK (age >= 0),
test(>              gender    CHAR(1)      CHECK (gender IN ('M','F')),
test(>              last_met  DATE         CHECK (last_met BETWEEN '1950-01-01'
test(>                                             AND CURRENT_DATE),
test(>              CHECK (upper(trim(firstname)) != 'ED' OR
test(>                     upper(trim(lastname)) != 'RIVERS')
test(> );
CREATE
test=> INSERT INTO friend2
test-> VALUES ('Ed', 'Rivers', 'Wibbleville', 'J', -35, 'S', '1931-09-23');
ERROR:  ExecAppend: rejected due to CHECK constraint friend2_last_met
```

Figure 14.15: CHECK constraints

Chapter 15

Importing and Exporting Data

Backslash String	Description
\TAB	tab if using default delimiter tab
\\|	*pipe* if using *pipe* as the delimiter
\N	NULL if using the default NULL output
\b	backspace
\f	form feed
\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\###	character represented by octal number ###
\\	backslash

Table 15.1: Backslashes understood by COPY

```
test=> CREATE TABLE copytest (
test(>                          intcol  INTEGER,
test(>                          numcol  NUMERIC(16,2),
test(>                          textcol TEXT,
test(>                          boolcol BOOLEAN
test(> );
CREATE
test=> INSERT INTO copytest
test-> VALUES (1, 23.99, 'fresh spring water', 't');
INSERT 174656 1
test=> INSERT INTO copytest
test-> VALUES (2, 55.23, 'bottled soda', 't');
INSERT 174657 1
test=> SELECT * FROM copytest;
 intcol | numcol |      textcol       | boolcol
--------+--------+--------------------+---------
      1 |  23.99 | fresh spring water | t
      2 |  55.23 | bottled soda       | t
(2 rows)

test=> COPY copytest TO '/tmp/copytest.out';
COPY
test=> DELETE FROM copytest;
DELETE 2
test=> COPY copytest FROM '/tmp/copytest.out';
COPY
test=> SELECT * FROM copytest;
 intcol | numcol |      textcol       | boolcol
--------+--------+--------------------+---------
      1 |  23.99 | fresh spring water | t
      2 |  55.23 | bottled soda       | t
(2 rows)
```

Figure 15.1: Example of COPY…TO and COPY…FROM

```
test=> \q
$ cat /tmp/copytest.out
1       23.99   fresh spring water      t
2       55.23   bottled soda    t

$ sed 's/       /<TAB>/g' /tmp/copytest.out  # the gap between /  / is a TAB
1<TAB>23.99<TAB>fresh spring water<TAB>t
2<TAB>55.23<TAB>bottled soda<TAB>t
```

Figure 15.2: Example of COPY…FROM

```
test=> COPY copytest TO '/tmp/copytest.out' USING DELIMITERS '|';
COPY
test=> \q
$ cat /tmp/copytest.out
1|23.99|fresh spring water|t
2|55.23|bottled soda|t
```

Figure 15.3: Example of COPY…TO…USING DELIMITERS

```
test=> DELETE FROM copytest;
DELETE 2
test=>
test=> COPY copytest FROM '/tmp/copytest.out';
ERROR:  copy: line 1, pg_atoi: error in "1|23.99|fresh spring water|t": cannot parse "|23.99|fresh spring water|t"
test=>
test=> COPY copytest FROM '/tmp/copytest.out' USING DELIMITERS '|';
COPY
```

Figure 15.4: Example of COPY…FROM…USING DELIMITERS

```
test=> COPY copytest FROM stdin;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
test> 3 77.43   coffee  f
test> \.
test=> COPY copytest TO stdout;
1       23.99   fresh spring water      t
2       55.23   bottled soda    t
3       77.43   coffee  f
test=>
```

Figure 15.5: COPY using *stdin* and *stdout*

```
test=> DELETE FROM copytest;
DELETE 3
test=> INSERT INTO copytest
test-> VALUES (4, 837.20, 'abc|def', NULL);
INSERT 174786 1
test=> COPY copytest TO stdout USING DELIMITERS '|';
4|837.20|abc\|def|\N
```

Figure 15.6: COPY backslash handling

Chapter 16

Database Query Tools

Function	Command	Argument
Print	\p	
Execute	\g or ;	*file* or |*command*
Quit	\q	
Clear	\r	
Edit	\e	*file*
Backslash help	\?	
SQL help	\h	*topic*
Include file	\i	*file*
Output to file/command	\o	*file* or |*command*
Write buffer to file	\w	*file*
Show/save query history	\s	*file*
Run subshell	\!	*command*

Table 16.1: `psql`'s query buffer commands

Operation	Command
Connect to another database	\connect *dbname*
Copy table file to/from database	\copy *tablename* to|from *filename*
Set a variable	\set *variable* or \set *variable value*
Unset a variable	\unset *variable*
Set output format	\pset *option* or \pset *option value*
Echo	\echo *string* or \echo `*command*`
Echo to \o output	\qecho *string* or \qecho `*command*`
Copyright	\copyright
Change character encoding	\encoding *newencoding*

Table 16.2: `psql`'s general commands

Format	Parameter	Options
Field alignment	format	unaligned, aligned, html, or latex
Field separator	fieldsep	*separator*
One field per line	expanded	
Rows only	tuples_only	
Row separator	recordsep	*separator*
Table title	title	*title*
Table border	border	0, 1, or 2
Display NULL values	null	*null_string*
HTML table tags	tableattr	*tags*
Page output	pager	*command*

Table 16.3: `psql`'s `\pset` options

```
test=> SELECT NULL;
 ?column?
----------

(1 row)

test=> \pset tuples_only
Showing only tuples.
test=> SELECT NULL;


test=> \pset null '(null)'
Null display is '(null)'.
test=> SELECT NULL;
 (null)
```

Figure 16.1: Example of *\pset*

Modifies	Command	Argument
Field alignment	\a	
Field separator	\f	*separator*
One field per line	\x	
Rows only	\t	
Table title	\C	*title*
Enable HTML	\H	
HTML table tags	\T	*tags*

Table 16.4: `psql`'s output format shortcuts

```
test=> \set num_var 4
test=> SELECT :num_var;
 ?column?
----------
        4
(1 row)

test=> \set operation SELECT
test=> :operation :num_var;
 ?column?
----------
        4
(1 row)

test=> \set str_var '\'My long string\''
test=> \echo :str_var
'My long string'
test=> SELECT :str_var;
    ?column?
----------------
 My long string
(1 row)

test=> \set date_var `date`
test=> \echo :date_var
Thu Aug 11 20:54:21 EDT 1994

test=> \set date_var2 '\''`date`'\''
test=> \echo :date_var2
'Thu Aug 11 20:54:24 EDT 1994'
test=> SELECT :date_var2;
           ?column?
-----------------------------
 Thu Aug 11 20:54:24 EDT 1994
(1 row)
```

Figure 16.2: `psql` variables

Meaning	Variable Name	Argument
Database	DBNAME	
Multibyte encoding	ENCODING	
Host	HOST	
Previously assigned OID	LASTOID	
Port	PORT	
User	USER	
Echo queries	ECHO	all
Echo \d* queries	ECHO_HIDDEN	noexec
History control	HISTCONTROL	ignorespace, ignoredups, or ignoreboth
History size	HISTSIZE	*command_count*
Terminate on end of file	IGNOREEOF	*eof_count*
\lobject transactions	LO_TRANSACTION	rollback, commit, nothing
Stop on query errors	ON_ERROR_STOP	
Command prompt	PROMPT1, PROMPT2, PROMPT3	*string*
Suppress output	QUIET	
Single-line mode	SINGLELINE	
Single-step mode	SINGLESTEP	

Table 16.5: `psql`'s predefined variables

Listing	Command	Argument
Table, index, view, or sequence	\d	*name*
Tables	\dt	*name*
Indexes	\di	*name*
Sequences	\ds	*name*
Views	\dv	*name*
Permissions	\z or \dp	*name*
System tables	\dS	*name*
Large objects	\dl	*name*
Types	\dT	*name*
Functions	\df	*name*
Operators	\do	*name*
Aggregates	\da	*name*
Comments	\dd	*name*
Databases	\l	

Table 16.6: `psql`'s listing commands

Large Objects	Command	Argument
Import	\lo_import	*file*
Export	\lo_export	*oid file*
Unlink	\lo_unlink	*oid*
List	\lo_list	

Table 16.7: `psql`'s large object commands

Option	Capability	Argument	Additional Argument
Connection	Database (optional)	-d	*database*
	Host name	-h	*hostname*
	Port	-p	*port*
	User	-U	*user*
	Force password prompt	-W	
	Version	-V	
Controlling Output	Field alignment	-A	
	Field separator	-F	*separator*
	Record separator	-R	*separator*
	Rows only	-t	
	Extended output format	-x	
	Echo \d* queries	-E	
	Quiet mode	-q	
	HTML output	-H	
	HTML table tags	-T	*tags*
	Set \pset options	-P	*option* or *option=value*
	List databases	-l	
	Disable *readline*	-n	
Automation	Echo all queries from scripts	-a	
	Echo queries	-e	
	Execute query	-c	*query*
	Get queries from file	-f	*file*
	Output to file	-o	*file*
	Single-step mode	-s	
	Single-line mode	-S	
	Suppress reading `~/.psqlrc`	-X	
	Set variable	-v	*var* or *var=value*

Table 16.8: `psql`'s command-line arguments

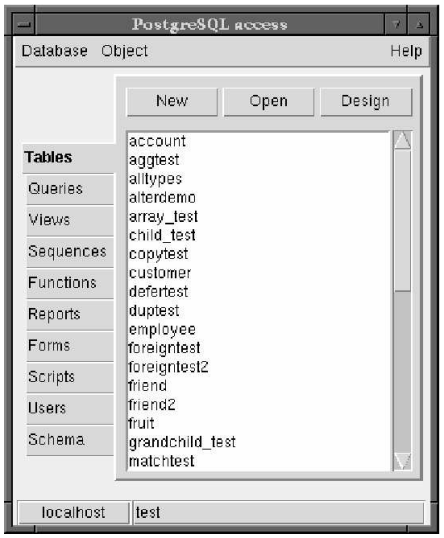

Figure 16.3: Pgaccess's opening window

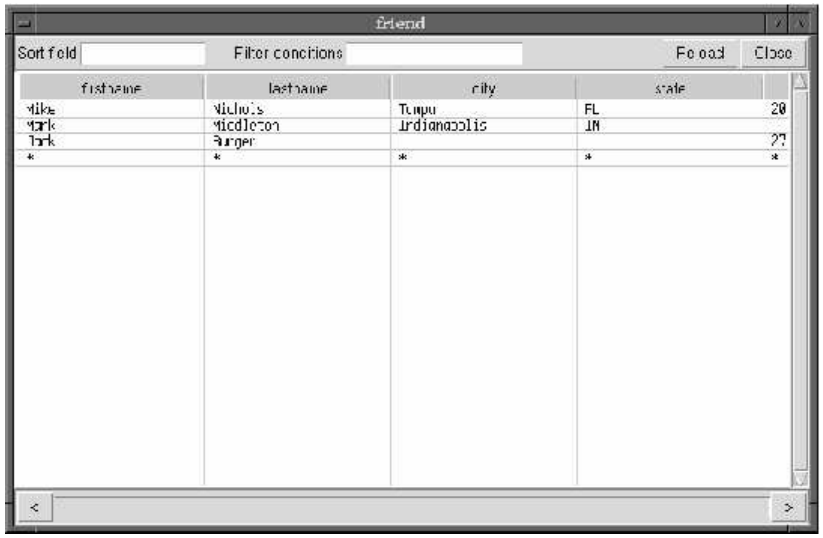

Figure 16.4: Pgaccess's table window

Chapter 17

Programming Interfaces

Interface	Language	Processing	Advantages
LIBPQ	C	compiled	native interface
LIBPGEASY	C	compiled	simplified C
ECPG	C	compiled	ANSI embedded SQL C
LIBPQ++	C++	compiled	object-oriented C
ODBC	ODBC	compiled	application connectivity
JDBC	Java	both	portability
PERL	Perl	interpreted	text processing
PGTCLSH	TCL/TK	interpreted	interfacing, windowing
PYTHON	Python	interpreted	object-oriented
PHP	HTML	interpreted	dynamic Web pages

Table 17.1: Interface summary

```
Enter a state code:  AL
Alabama
```

Figure 17.1: Sample application being run

```
test=>  CREATE TABLE statename (code CHAR(2) PRIMARY KEY,
test(>                          name  CHAR(30)
test(> );
CREATE
test=> INSERT INTO statename VALUES ('AL', 'Alabama');
INSERT 18934 1
test=> INSERT INTO statename VALUES ('AK', 'Alaska');
INSERT 18934 1
```

Figure 17.2: *Statename* table

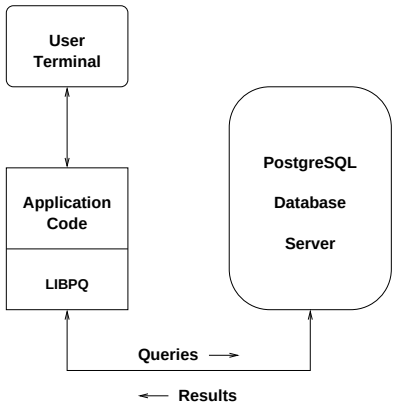

Figure 17.3: LIBPQ data flow

```
/*
 *  libpq sample program
 */

#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"                                   /* libpq header file */

int
main()
{
    char        state_code[3];                          /* holds state code entered by user */
    char        query_string[256];                      /* holds constructed SQL query */
    PGconn     *conn;                                   /* holds database connection */
    PGresult   *res;                                    /* holds query result */
    int         i;

    conn = PQconnectdb("dbname=test");                  /* connect to the database */

    if (PQstatus(conn) == CONNECTION_BAD)               /* did the database connection fail? */
    {
        fprintf(stderr, "Connection to database failed.\n");
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit(1);
    }

    printf("Enter a state code:  ");                    /* prompt user for a state code */
    scanf("%2s", state_code);

    sprintf(query_string,                               /* create an SQL query string */
            "SELECT name \
             FROM statename \
             WHERE code = '%s'", state_code);

    res = PQexec(conn, query_string);                   /* send the query */

    if (PQresultStatus(res) != PGRES_TUPLES_OK)         /* did the query fail? */
    {
        fprintf(stderr, "SELECT query failed.\n");
        PQclear(res);
        PQfinish(conn);
        exit(1);
    }

    for (i = 0; i < PQntuples(res); i++)                /* loop through all rows returned */
        printf("%s\n", PQgetvalue(res, i, 0));          /* print the value returned */

    PQclear(res);                                       /* free result */

    PQfinish(conn);                                     /* disconnect from the database */

    return 0;
}
```

Figure 17.4: LIBPQ sample program

```
/*
 *  libpgeasy sample program
 */

#include <stdio.h>
#include <libpq-fe.h>
#include <libpgeasy.h>                          /* libpgeasy header file */

int
main()
{
    char        state_code[3];                  /* holds state code entered by user */
    char        query_string[256];              /* holds constructed SQL query */
    char        state_name[31];                 /* holds returned state name */

    connectdb("dbname=test");                   /* connect to the database */

    printf("Enter a state code:  ");            /* prompt user for a state code */
    scanf("%2s", state_code);

    sprintf(query_string,                       /* create an SQL query string */
            "SELECT name \
             FROM statename \
             WHERE code = '%s'", state_code);

    doquery(query_string);                      /* send the query */

    while (fetch(state_name) != END_OF_TUPLES)  /* loop through all rows returned */
        printf("%s\n", state_name);             /* print the value returned */

    disconnectdb();                             /* disconnect from the database */

    return 0;
}
```

Figure 17.5: LIBPGEASY sample program

```
/*
 *  ecpg sample program
 */

#include <stdio.h>

EXEC SQL INCLUDE sqlca;                                 /* ecpg header file */

EXEC SQL WHENEVER SQLERROR sqlprint;

int
main()
{
EXEC SQL BEGIN DECLARE SECTION;
    char        state_code[3];                          /* holds state code entered by user */
    char       *state_name = NULL;                      /* holds value returned by query */
    char        query_string[256];                      /* holds constructed SQL query */
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO test;                           /* connect to the database */

    printf("Enter a state code:  ");                    /* prompt user for a state code */
    scanf("%2s", state_code);

    sprintf(query_string,                               /* create an SQL query string */
                "SELECT name \
                 FROM statename \
                 WHERE code = '%s'", state_code);

    EXEC SQL PREPARE s_statename FROM :query_string;
    EXEC SQL DECLARE c_statename CURSOR FOR s_statename;/* DECLARE a cursor */

    EXEC SQL OPEN c_statename;                          /* send the query */

    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)                                           /* loop through all rows returned */
    {
        EXEC SQL FETCH IN c_statename INTO :state_name;
        printf("%s\n", state_name);                     /* print the value returned */
        state_name = NULL;
    }

    free(state_name);                                   /* free result */

    EXEC SQL CLOSE c_statename;                         /* CLOSE the cursor */

    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT;                                /* disconnect from the database */

    return 0;
}
```

Figure 17.6: ECPG sample program

```
/*
 *  libpq++ sample program
 */

#include <iostream.h>
#include <libpq++.h>                                    // libpq++ header file

int main()
{
    char        state_code[3];                          // holds state code entered by user
    char        query_string[256];                      // holds constructed SQL query
    PgDatabase data("dbname=test");                     // connects to the database

    if ( data.ConnectionBad() )                         // did the database connection fail?
    {
        cerr << "Connection to database failed." << endl
             << "Error returned: " << data.ErrorMessage() << endl;
        exit(1);
    }

    cout << "Enter a state code:  ";                    // prompt user for a state code
    cin.get(state_code, 3, '\n');

    sprintf(query_string,                               // create an SQL query string
            "SELECT name \
             FROM statename \
             WHERE code = '%s'", state_code);

    if ( !data.ExecTuplesOk(query_string) )             // send the query
    {
        cerr << "SELECT query failed." << endl;
        exit(1);
    }

    for (int i=0; i < data.Tuples(); i++)               // loop through all rows returned
            cout << data.GetValue(i,0) << endl;         // print the value returned

    return 0;
}
```

Figure 17.7: LIBPQ++ sample program

```
/*
 *  Java sample program
 */

import java.io.*;
import java.sql.*;

public class sample
{
    Connection  conn;                                   // holds database connection
    Statement   stmt;                                   // holds SQL statement
    String      state_code;                             // holds state code entered by user

    public sample() throws ClassNotFoundException, FileNotFoundException, IOException, SQLException
    {
        Class.forName("org.postgresql.Driver");         // load database interface
                                                        // connect to the database
        conn = DriverManager.getConnection("jdbc:postgresql:test", "testuser", "");
        stmt = conn.createStatement();

        System.out.print("Enter a state code:  ");      // prompt user for a state code
        System.out.flush();
        BufferedReader r = new BufferedReader(new InputStreamReader(System.in));
        state_code = r.readLine();

        ResultSet res = stmt.executeQuery(              // send the query
            "SELECT name " +
            "FROM statename " +
            "WHERE code = '" + state_code + "'");

        if (res != null)
            while(res.next())
            {
                String state_name = res.getString(1);
                System.out.println(state_name);
            }

        res.close();
        stmt.close();
        conn.close();
    }

    public static void main(String args[])
    {
        try {
            sample test = new sample();
        } catch(Exception exc)
        {
            System.err.println("Exception caught.\n" + exc);
            exc.printStackTrace();
        }
    }
}
```

Figure 17.8: Java sample program

```
#!/usr/local/bin/perl -w
#
#   Perl sample program
#

use Pg;                                                 # load database routines

$conn = Pg::connectdb("dbname=test");                   # connect to the database
                                                        # did the database connection fail?
die $conn->errorMessage unless PGRES_CONNECTION_OK eq $conn->status;

print "Enter a state code: ";                           # prompt user for a state code
$state_code = <STDIN>;
chomp $state_code;

$result = $conn->exec(                                  # send the query
        "SELECT name \
         FROM statename \
         WHERE code = '$state_code'");
                                                        # did the query fail?
die $conn->errorMessage unless PGRES_TUPLES_OK eq $result->resultStatus;

while (@row = $result->fetchrow) {                      # loop through all rows returned
        print @row, "\n";                               # print the value returned
}
```

Figure 17.9: Perl sample program

```
#!/usr/local/pgsql/bin/pgtclsh
#
#   pgtclsh sample program
#

set conn [pg_connect -conninfo "dbname=test"]           ;# connect to the database

puts -nonewline "Enter a state code:  "                 ;# prompt user for a state code
flush stdout
gets stdin state_code
                                                        ;# send the query
set res [pg_exec $conn \
        "SELECT name \
         FROM statename \
         WHERE code = '$state_code'"]

set ntups [pg_result $res -numTuples]

for {set i 0} {$i < $ntups} {incr i} {                  ;# loop through all rows returned
        puts stdout [lindex [pg_result $res -getTuple $i] 0]    ;# print the value returned

}
pg_disconnect $conn                                     ;# disconnect from the database
```

Figure 17.10: TCL sample program

```
#! /usr/local/bin/python
#
#   Python sample program
#

import sys

from pg import DB                                       # load database routines

conn = DB('test')                                       # connect to the database

sys.stdout.write('Enter a state code:  ')               # prompt user for a state code
state_code = sys.stdin.readline()
state_code = state_code[:-1]

for name in conn.query(                                 # send the query
        "SELECT name \
         FROM statename \
         WHERE code = '"+state_code+"'").getresult():
        sys.stdout.write('%s\n' % name)                 # print the value returned
```

Figure 17.11: Python sample program

```
<!--
 -- PHP sample program -- input
 -->

<HTML>
<BODY>
<!-- prompt user for a state code -->
<FORM ACTION="<? echo $SCRIPT_NAME ?>/pg/sample2.phtml?state_code" method="POST">
Client Number:
<INPUT TYPE="text" name="state_code" value="<? echo $state_code ?>"
        maxlength=2 size=2>
<BR>
<INPUT TYPE="submit" value="Continue">
</FORM>
</BODY>
</HTML>
```

Figure 17.12: PHP sample program—input

```
<!--
 -- PHP sample program -- output
 -->

<HTML>
<BODY>
<?
        $database = pg_Connect("", "", "", "", "test"); # connect to the database

        if (!$database)                                 # did the database connection fail?
        {
                echo "Connection to database failed.";
                exit;
        }

        $result = pg_Exec($database,                    # send the query
                "SELECT name " .
                "FROM statename " .
                "WHERE code = '$state_code'");


        for ($i = 0; $i < pg_NumRows($result); $i++)    # loop through all rows returned
        {
                echo pg_Result($result,$i,0);           # print the value returned
                echo "<BR>";
        }
?>
</BODY>
</HTML>
```

Figure 17.13: PHP sample program—output

Chapter 18

Functions and Triggers

```
test=> CREATE FUNCTION ftoc(float)
test-> RETURNS float
test-> AS 'SELECT ($1 - 32.0) * 5.0 / 9.0;'
test-> LANGUAGE 'sql';
CREATE
test=> SELECT ftoc(68);
 ftoc
------
   20
(1 row)
```

Figure 18.1: SQL *ftoc* function

```
test=> CREATE FUNCTION tax(numeric)
test-> RETURNS numeric
test-> AS 'SELECT ($1 * 0.06::numeric(8,2))::numeric(8,2);'
test-> LANGUAGE 'sql';
CREATE
test=> SELECT tax(100);
 tax
------
 6.00
(1 row)
```

Figure 18.2: SQL *tax* function

```
test=> CREATE TABLE part (
test(>                    part_id     INTEGER,
test(>                    name        CHAR(30),
test(>                    cost        NUMERIC(8,2),
test(>                    weight      FLOAT
test(> );
CREATE
test=> INSERT INTO part VALUES (637, 'cable', 14.29, 5);
INSERT 20867 1
test=> INSERT INTO part VALUES (638, 'sticker', 0.84, 1);
INSERT 20868 1
test=> INSERT INTO part VALUES (639, 'bulb', 3.68, 3);
INSERT 20869 1
test=> SELECT part_id,
test->        name,
test->        cost,
test->        tax(cost),
test->        cost + tax(cost) AS total
test-> FROM part
test-> ORDER BY part_id;
 part_id |              name              | cost  | tax  | total
---------+--------------------------------+-------+------+-------
     637 | cable                          | 14.29 | 0.86 | 15.15
     638 | sticker                        |  0.84 | 0.05 |  0.89
     639 | bulb                           |  3.68 | 0.22 |  3.90
(3 rows)
```

Figure 18.3: Recreation of the *part* table

```
test=> CREATE FUNCTION shipping(numeric)
test-> RETURNS numeric
test-> AS 'SELECT CASE
test'>                 WHEN $1 < 2             THEN CAST(3.00 AS numeric(8,2))
test'>                 WHEN $1 >= 2 AND $1 < 4 THEN CAST(5.00 AS numeric(8,2))
test'>                 WHEN $1 >= 4            THEN CAST(6.00 AS numeric(8,2))
test'>            END;'
test-> LANGUAGE 'sql';
CREATE

test=> SELECT part_id,
test->        trim(name) AS name,
test->        cost,
test->        tax(cost),
test->        cost + tax(cost) AS subtotal,
test->        shipping(weight),
test->        cost + tax(cost) + shipping(weight) AS total
test-> FROM part
test-> ORDER BY part_id;
 part_id |  name   | cost  | tax  | subtotal | shipping | total
---------+---------+-------+------+----------+----------+-------
     637 | cable   | 14.29 | 0.86 |    15.15 |     6.00 | 21.15
     638 | sticker |  0.84 | 0.05 |     0.89 |     3.00 |  3.89
     639 | bulb    |  3.68 | 0.22 |     3.90 |     5.00 |  8.90
(3 rows)
```

Figure 18.4: SQL *shipping* function

```
test=> CREATE FUNCTION getstatename(text)
test-> RETURNS text
test-> AS 'SELECT CAST(name AS TEXT)
test->     FROM  statename
test->     WHERE code = $1;'
test-> LANGUAGE 'sql';
CREATE
test=> SELECT getstatename('AL');
          getstatename
-------------------------------
 Alabama
(1 row)
```

Figure 18.5: SQL *getstatename* function

```
test=> SELECT customer.name, statename.name
test-> FROM   customer, statename
test-> WHERE  customer.state = statename.code
test-> ORDER BY customer.name;
              name              |              name
--------------------------------+--------------------------------
 Fleer Gearworks, Inc.          | Alabama
 Mark Middleton                 | Indiana
 Mike Nichols                   | Florida
(3 rows)

test=> SELECT customer.name, getstatename(customer.state)
test-> FROM   customer
test-> ORDER BY customer.name;
              name              |          getstatename
--------------------------------+--------------------------------
 Fleer Gearworks, Inc.          | Alabama
 Mark Middleton                 | Indiana
 Mike Nichols                   | Florida
(3 rows)
```

Figure 18.6: Getting state name using a join and a function

```
test=> CREATE FUNCTION getstatename2(text)
test-> RETURNS text
test-> AS 'DECLARE ret TEXT;
test'>     BEGIN
test'>         SELECT INTO ret CAST(name AS TEXT)
test'>         FROM  statename
test'>         WHERE code = $1;
test'>         RETURN ret;
test'>     END;'
test'> LANGUAGE 'plpgsql';
CREATE
```

Figure 18.7: PL/PGSQL version of *getstatename*

```
test=> CREATE FUNCTION spread(text)
test-> RETURNS text
test-> AS 'DECLARE
test'>         str text;
test'>         ret text;
test'>         i   integer;
test'>         len integer;
test'>
test'>     BEGIN
test'>         str := upper($1);
test'>         ret := '''';           -- start with zero length
test'>         i   := 1;
test'>         len := length(str);
test'>         WHILE i <= len LOOP
test'>             ret := ret || substr(str, i, 1) || '' '';
test'>             i := i + 1;
test'>         END LOOP;
test'>         RETURN ret;
test'>     END;'
test-> LANGUAGE 'plpgsql';
CREATE
test=> SELECT spread('Major Financial Report');
                   spread
---------------------------------------------
 M A J O R   F I N A N C I A L   R E P O R T
(1 row)
```

Figure 18.8: PL/PGSQL *spread* function

```
test=> CREATE FUNCTION getstatecode(text)
test-> RETURNS text
test-> AS 'DECLARE
test'>         state_str  statename.name%TYPE;
test'>         statename_rec record;
test'>         i          integer;
test'>         len        integer;
test'>         matches    record;
test'>         search_str text;
test'>
test'>     BEGIN
test'>         state_str := initcap($1);              -- capitalization match column
test'>         len := length(trim($1));
test'>         i   := 2;
test'>
test'>         SELECT INTO statename_rec *             -- first try for an exact match
test'>         FROM   statename
test'>         WHERE  name = state_str;
test'>         IF FOUND
test'>         THEN    RETURN statename_rec.code;
test'>         END IF;
test'>
test'>         WHILE i <= len LOOP                    -- test 2,4,6,... chars for match
test'>             search_str = trim(substr(state_str, 1, i)) || ''%'';
test'>             SELECT INTO matches COUNT(*)
test'>             FROM   statename
test'>             WHERE  name LIKE search_str;
test'>
test'>             IF matches.count = 0               -- no matches, failure
test'>             THEN   RETURN NULL;
test'>             END IF;
test'>             IF matches.count = 1               -- exactly one match, return it
test'>             THEN
test'>                  SELECT INTO statename_rec *
test'>                  FROM   statename
test'>                  WHERE  name LIKE search_str;
test'>                  IF FOUND
test'>                  THEN    RETURN statename_rec.code;
test'>                  END IF;
test'>             END IF;
test'>             i := i + 2;                       -- >1 match, try 2 more chars
test'>         END LOOP;
test'>         RETURN '''' ;
test'>    END;'
test-> LANGUAGE 'plpgsql';
```

Figure 18.9: PL/PGSQL *getstatecode* function

```
test=> SELECT getstatecode('Alabama');
 getstatecode
--------------
 AL
(1 row)

test=> SELECT getstatecode('ALAB');
 getstatecode
--------------
 AL
(1 row)

test=> SELECT getstatecode('Al');
 getstatecode
--------------
 AL
(1 row)

test=> SELECT getstatecode('Ail');
 getstatecode
--------------

(1 row)
```

Figure 18.10: Calls to *getstatecode* function

```
test=> CREATE FUNCTION change_statename(char(2), char(30))
test-> RETURNS boolean
test-> AS 'DECLARE
test'>     state_code ALIAS FOR $1;
test'>     state_name ALIAS FOR $2;
test'>     statename_rec RECORD;
test'>
test'>     BEGIN
test'>         IF length(state_code) = 0                  -- no state code, failure
test'>         THEN    RETURN ''f'';
test'>         ELSE
test'>             IF length(state_name) != 0             -- is INSERT or UPDATE?
test'>             THEN
test'>                 SELECT INTO statename_rec *
test'>                 FROM   statename
test'>                 WHERE  code = state_code;
test'>                 IF NOT FOUND                       -- is state not in table?
test'>                 THEN    INSERT INTO statename
test'>                         VALUES (state_code, state_name);
test'>                 ELSE    UPDATE statename
test'>                         SET    name = state_name
test'>                         WHERE  code = state_code;
test'>                 END IF;
test'>                 RETURN ''t'';
test'>             ELSE                                   -- is DELETE
test'>                 SELECT INTO statename_rec *
test'>                 FROM   statename
test'>                 WHERE  code = state_code;
test'>                 IF FOUND
test'>                 THEN    DELETE FROM statename
test'>                         WHERE code = state_code;
test'>                         RETURN ''t'';
test'>                 ELSE    RETURN ''f'';
test'>                 END IF;
test'>             END IF;
test'>         END IF;
test'>    END;'
test-> LANGUAGE 'plpgsql';
```

Figure 18.11: PL/PGSQL *change_statename* function

```
test=> DELETE FROM statename;
DELETE 1
test=> SELECT change_statename('AL','Alabama');
 change_statename
------------------
 t
(1 row)

test=> SELECT * FROM statename;
 code |              name
------+--------------------------------
 AL   | Alabama
(1 row)

test=> SELECT change_statename('AL','Bermuda');
 change_statename
------------------
 t
(1 row)

test=> SELECT * FROM statename;
 code |              name
------+--------------------------------
 AL   | Bermuda
(1 row)

test=> SELECT change_statename('AL','');
 change_statename
------------------
 t
(1 row)

test=> SELECT change_statename('AL','');    -- row was already deleted
 change_statename
------------------
 f
(1 row)
```

Figure 18.12: Examples using *change_statename()*

```
test=> CREATE FUNCTION trigger_insert_update_statename()
test-> RETURNS opaque
test-> AS 'BEGIN
test'>         IF new.code !~ ''^[A-Za-z][A-Za-z]$''
test'>         THEN    RAISE EXCEPTION ''State code must be two alphabetic characters.'';
test'>         END IF;
test'>         IF new.name !~ ''^[A-Za-z ]*$''
test'>         THEN    RAISE EXCEPTION ''State name must be only alphabetic characters.'';
test'>         END IF;
test'>         IF length(trim(new.name)) < 3
test'>         THEN    RAISE EXCEPTION ''State name must longer than two characters.'';
test'>         END IF;
test'>         new.code = upper(new.code);               -- uppercase statename.code
test'>         new.name = initcap(new.name);             -- capitalize statename.name
test'>         RETURN new;
test'>     END;'
test-> LANGUAGE 'plpgsql';
CREATE

test=> CREATE TRIGGER trigger_statename
test-> BEFORE INSERT OR UPDATE
test-> ON statename
test-> FOR EACH ROW
test-> EXECUTE PROCEDURE trigger_insert_update_statename();
CREATE

test=> DELETE FROM statename;
DELETE 1
test=> INSERT INTO statename VALUES ('a', 'alabama');
ERROR:  State code must be two alphabetic characters.
test=> INSERT INTO statename VALUES ('al', 'alabama2');
ERROR:  State name must be only alphabetic characters.
test=> INSERT INTO statename VALUES ('al', 'al');
ERROR:  State name must longer than two characters.
test=> INSERT INTO statename VALUES ('al', 'alabama');
INSERT 292898 1
test=> SELECT * FROM statename;
 code |              name
------+--------------------------------
 AL   | Alabama
(1 row)
```

Figure 18.13: Trigger creation

Chapter 19

Extending POSTGRESQL Using C

```
#include "postgres.h"
double *ctof(double *deg)
{
    double *ret = palloc(sizeof(double));

    *ret = (*deg * 9.0 / 5.0) + 32.0;
    return ret;
}
```

Figure 19.1: C *ctof* function

```
test=> CREATE FUNCTION ctof(float)
test-> RETURNS float
test-> AS '/users/pgman/sample/ctof.so'
test-> LANGUAGE 'C';
CREATE
```

Figure 19.2: Create function *ctof*

```
test=> SELECT ctof(20);
 ctof
------
   68
(1 row)
```

Figure 19.3: Calling function *ctof*

Chapter 20

Administration

Table	Contents
pg_aggregate	aggregates
pg_attribute	columns
pg_class	tables
pg_database	databases
pg_description	comments
pg_group	groups
pg_index	indexes
pg_log	transaction status
pg_operator	operators
pg_proc	functions
pg_rewrite	rules and views
pg_shadow	users
pg_trigger	triggers
pg_type	types

Table 20.1: Commonly used system tables

```
$ createuser demouser1
Shall the new user be allowed to create databases? (y/n) n
Shall the new user be allowed to create more new users? (y/n) n
CREATE USER
$ psql test
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

test=> CREATE USER demouser2;
CREATE USER
test=> ALTER USER demouser2 CREATEDB;
ALTER USER
test=> CREATE GROUP demogroup WITH USER demouser1, demouser2;
CREATE GROUP
test=> CREATE TABLE grouptest (col INTEGER);
CREATE
test=> GRANT ALL on grouptest TO GROUP demogroup;
CHANGE
test=> \connect test demouser2
You are now connected to database test as user demouser2.
test=> \q
```

Figure 20.1: Examples of user administration

```
$ createdb demodb1
CREATE DATABASE
$ psql test
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

test=> CREATE DATABASE demodb2;
CREATE DATABASE
test=> DROP DATABASE demodb1;
DROP DATABASE
test=> \connect demodb2
You are now connected to database demodb2.
demodb2=> \q
```

Figure 20.2: Examples of database creation and removal

```
$ pg_dump test > /tmp/test.dump
$ createdb newtest
CREATE DATABASE
$ psql newtest < /tmp/test.dump
```

Figure 20.3: Making a new copy of database *test*

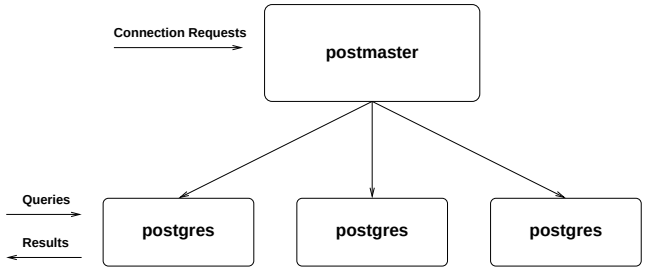

Figure 20.4: `Postmaster` and `postgres` processes