PostgreSQL: Introduction and Concepts

Bruce Momjian

19th February 2004
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;
getpusername
--------------
postgres
(1 row)

test=>

Figure 2.2: My first SQL query

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

test=>

Figure 2.3: Multiline query
Figure 2.4: Backslash-p demo
Chapter 3

Basic SQL Commands

Figure 3.1: Databases
<table>
<thead>
<tr>
<th>FirstName</th>
<th>LastName</th>
<th>City</th>
<th>State</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mike</td>
<td>Nichols</td>
<td>Tampa</td>
<td>FL</td>
<td>19</td>
</tr>
<tr>
<td>Cindy</td>
<td>Anderson</td>
<td>Denver</td>
<td>CO</td>
<td>23</td>
</tr>
<tr>
<td>Sam</td>
<td>Jackson</td>
<td>Allentown</td>
<td>PA</td>
<td>22</td>
</tr>
</tbody>
</table>

Table 3.1: Table friend

```sql
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
Figure 3.4: INSERT into friend

```
INSERT INTO friend VALUES ('Mike', 'Nichols', 'Tampa', 'FL', 19);
```

Figure 3.5: Additional friend INSERT commands

```
INSERT INTO friend VALUES ('Cindy', 'Anderson', 'Denver', 'CO', 23);
```

```
INSERT INTO friend VALUES ('Sam', 'Jackson', 'Allentown', 'PA', 22);
```
```
test=> SELECT * FROM friend;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mike</td>
<td>Nichols</td>
<td>Tampa</td>
<td>FL</td>
<td>19</td>
</tr>
<tr>
<td>Cindy</td>
<td>Anderson</td>
<td>Denver</td>
<td>CO</td>
<td>23</td>
</tr>
<tr>
<td>Sam</td>
<td>Jackson</td>
<td>Allentown</td>
<td>PA</td>
<td>22</td>
</tr>
</tbody>
</table>

(3 rows)
```

Figure 3.6: My first SELECT

```
test=> SELECT * FROM friend WHERE age = 23;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cindy</td>
<td>Anderson</td>
<td>Denver</td>
<td>CO</td>
<td>23</td>
</tr>
</tbody>
</table>

(1 row)
```

Figure 3.7: My first WHERE

```
test=> SELECT * FROM friend WHERE age <= 22;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mike</td>
<td>Nichols</td>
<td>Tampa</td>
<td>FL</td>
<td>19</td>
</tr>
<tr>
<td>Sam</td>
<td>Jackson</td>
<td>Allentown</td>
<td>PA</td>
<td>22</td>
</tr>
</tbody>
</table>

(2 rows)
```

Figure 3.8: More complex WHERE clause
```sql
SELECT lastname FROM friend WHERE age = 22;
```

<table>
<thead>
<tr>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jackson</td>
</tr>
</tbody>
</table>

(1 row)

Figure 3.9: A single cell

```sql
SELECT city, state FROM friend WHERE age >= 21;
```

<table>
<thead>
<tr>
<th>city</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Denver</td>
<td>CO</td>
</tr>
<tr>
<td>Allentown</td>
<td>PA</td>
</tr>
</tbody>
</table>

(2 rows)

Figure 3.10: A block of cells

```sql
SELECT * FROM friend WHERE firstname = 'Sam';
```

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sam</td>
<td>Jackson</td>
<td>Allentown</td>
<td>PA</td>
<td>22</td>
</tr>
</tbody>
</table>

(1 row)

Figure 3.11: Comparing string fields
```sql
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)
```

```sql
test=> INSERT INTO friend VALUES ('Jim', 'Barnes', 'Ocean City', 'NJ', 25);
  INSERT 19056 1
```

```sql
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)
```

```sql
test=> DELETE FROM friend WHERE lastname = 'Barnes';
  DELETE 1
```

```sql
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;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cindy</td>
<td>Anderson</td>
<td>Denver</td>
<td>CO</td>
<td>23</td>
</tr>
<tr>
<td>Sam</td>
<td>Jackson</td>
<td>Allentown</td>
<td>PA</td>
<td>22</td>
</tr>
<tr>
<td>Mike</td>
<td>Nichols</td>
<td>Tampa</td>
<td>FL</td>
<td>20</td>
</tr>
</tbody>
</table>

(3 rows)

Figure 3.13: My first UPDATE


test=> SELECT * FROM friend ORDER BY state;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cindy</td>
<td>Anderson</td>
<td>Denver</td>
<td>CO</td>
<td>23</td>
</tr>
<tr>
<td>Mike</td>
<td>Nichols</td>
<td>Tampa</td>
<td>FL</td>
<td>20</td>
</tr>
<tr>
<td>Sam</td>
<td>Jackson</td>
<td>Allentown</td>
<td>PA</td>
<td>22</td>
</tr>
</tbody>
</table>

(3 rows)

Figure 3.14: Use of ORDER BY


test=> SELECT * FROM friend ORDER BY age DESC;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cindy</td>
<td>Anderson</td>
<td>Denver</td>
<td>CO</td>
<td>23</td>
</tr>
<tr>
<td>Sam</td>
<td>Jackson</td>
<td>Allentown</td>
<td>PA</td>
<td>22</td>
</tr>
<tr>
<td>Mike</td>
<td>Nichols</td>
<td>Tampa</td>
<td>FL</td>
<td>20</td>
</tr>
</tbody>
</table>

(3 rows)

Figure 3.15: Reverse ORDER BY
```
SELECT * FROM friend WHERE age >= 21 ORDER BY firstname;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cindy</td>
<td>Anderson</td>
<td>Denver</td>
<td>CO</td>
<td>23</td>
</tr>
<tr>
<td>Sam</td>
<td>Jackson</td>
<td>Allentown</td>
<td>PA</td>
<td>22</td>
</tr>
</tbody>
</table>
```

(2 rows)

Figure 3.16: Use of ORDER BY and WHERE
Chapter 4

Customizing Queries

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

Table 4.1: Common data types
test=> CREATE TABLE alltypes (
  state CHAR(2),
  name CHAR(30),
  children INTEGER,
  distance FLOAT,
  budget NUMERIC(16,2),
  born DATE,
  checkin TIME,
  started TIMESTAMP
);
CREATE

test=> INSERT INTO alltypes
VALUES (
  'PA',
  'Hilda Blairwood',
  3,
  10.7,
  4308.20,
  '9/8/1974',
  '9:00',
  '07/03/1996 10:30:00');

INSERT 190731

test=> SELECT state, name, children, distance, budget FROM alltypes;

<table>
<thead>
<tr>
<th>state</th>
<th>name</th>
<th>children</th>
<th>distance</th>
<th>budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA</td>
<td>Hilda Blairwood</td>
<td>3</td>
<td>10.7</td>
<td>4308.20</td>
</tr>
</tbody>
</table>

(1 row)

test=> SELECT born, checkin, started FROM alltypes;

<table>
<thead>
<tr>
<th>born</th>
<th>checkin</th>
<th>started</th>
</tr>
</thead>
<tbody>
<tr>
<td>1974-09-08</td>
<td>09:00:00</td>
<td>1996-07-03 10:30:00-04</td>
</tr>
</tbody>
</table>

(1 row)

test=> \x
Expanded display is on.

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
### NULL handling

```sql
test=> SELECT * FROM friend ORDER BY age DESC;
f 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)
```

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

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

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

Figure 4.3: NULL handling
test=> INSERT INTO friend
  VALUES ('Jack', 'Burger', NULL, NULL, 27);
INSERT 19075 1

test=> SELECT * FROM friend WHERE city = state;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(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;

<table>
<thead>
<tr>
<th>name</th>
<th>spouse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andy</td>
<td></td>
</tr>
<tr>
<td>Tom</td>
<td></td>
</tr>
</tbody>
</table>

(2 rows)

test=> SELECT * FROM nulltest WHERE spouse = '';

<table>
<thead>
<tr>
<th>name</th>
<th>spouse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andy</td>
<td></td>
</tr>
</tbody>
</table>

(1 row)

test=> SELECT * FROM nulltest WHERE spouse IS NULL;

<table>
<thead>
<tr>
<th>name</th>
<th>spouse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tom</td>
<td></td>
</tr>
</tbody>
</table>

(1 row)

Figure 4.5: NULL values and blank strings
```
test=> CREATE TABLE account (  
  name CHAR(20),  
  balance NUMERIC(16,2) DEFAULT 0,  
  active CHAR(1) DEFAULT 'Y',  
  created TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)

CREATE

INSERT INTO account (name) 
VALUES ('Federated Builders');

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

```
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;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dean</td>
<td>Yeager</td>
<td>Plymouth</td>
<td>MA</td>
<td>24</td>
</tr>
<tr>
<td>Dick</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
<td>19</td>
</tr>
<tr>
<td>Ned</td>
<td>Millstone</td>
<td>Cedar Creek</td>
<td>MD</td>
<td>27</td>
</tr>
<tr>
<td>Sandy</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
<td>25</td>
</tr>
<tr>
<td>Sandy</td>
<td>Weber</td>
<td>Boston</td>
<td>MA</td>
<td>33</td>
</tr>
<tr>
<td>Victor</td>
<td>Tabor</td>
<td>Williamsport</td>
<td>PA</td>
<td>22</td>
</tr>
</tbody>
</table>

(6 rows)

Figure 4.10: New friends
test=> SELECT * FROM friend
test-> WHERE firstname = 'Sandy' AND lastname = 'Gleason';

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sandy</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
<td>25</td>
</tr>
</tbody>
</table>

(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;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dick</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
<td>19</td>
</tr>
<tr>
<td>Sandy</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
<td>25</td>
</tr>
<tr>
<td>Victor</td>
<td>Tabor</td>
<td>Williamsport</td>
<td>PA</td>
<td>22</td>
</tr>
</tbody>
</table>

(3 rows)

Figure 4.12: Friends in New Jersey and Pennsylvania

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>less than</td>
<td>&lt;</td>
</tr>
<tr>
<td>less than or equal</td>
<td>&lt;=</td>
</tr>
<tr>
<td>equal</td>
<td>=</td>
</tr>
<tr>
<td>greater than or equal</td>
<td>&gt;=</td>
</tr>
<tr>
<td>greater than</td>
<td>&gt;</td>
</tr>
<tr>
<td>not equal</td>
<td>&lt;&gt; or !=</td>
</tr>
</tbody>
</table>

Table 4.2: Comparison operators
test=> SELECT * FROM friend
  test-> WHERE firstname = 'Victor' AND state = 'PA' OR state = 'NJ'
  test-> ORDER BY firstname;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dick</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
<td>19</td>
</tr>
<tr>
<td>Sandy</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
<td>25</td>
</tr>
<tr>
<td>Victor</td>
<td>Tabor</td>
<td>Williamsport</td>
<td>PA</td>
<td>22</td>
</tr>
</tbody>
</table>

(3 rows)

Figure 4.13: Incorrectly mixing AND and OR clauses

Figure 4.14: Correctly mixing AND and OR clauses

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>begins with D</td>
<td>LIKE 'D%'</td>
</tr>
<tr>
<td>contains a D</td>
<td>LIKE '%D%'</td>
</tr>
<tr>
<td>has D in second position</td>
<td>LIKE '_D%'</td>
</tr>
<tr>
<td>begins with D and contains e</td>
<td>LIKE 'D*e%'</td>
</tr>
<tr>
<td>begins with D, contains e, then f</td>
<td>LIKE 'D*e%f'</td>
</tr>
<tr>
<td>begins with non-D</td>
<td>NOT LIKE 'D%'</td>
</tr>
</tbody>
</table>

Table 4.3: LIKE comparisons
```
test=> SELECT *
test-> FROM friend
test-> WHERE age >= 22 AND age <= 25
```
```
test-> ORDER BY firstname;
```
```
<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dean</td>
<td>Yeager</td>
<td>Plymouth</td>
<td>MA</td>
<td>24</td>
</tr>
<tr>
<td>Sandy</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
<td>25</td>
</tr>
<tr>
<td>Victor</td>
<td>Tabor</td>
<td>Williamsport</td>
<td>PA</td>
<td>22</td>
</tr>
</tbody>
</table>
```
(3 rows)

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

Figure 4.15: Selecting a range of values

```
test=> SELECT * FROM friend
```
```
test-> WHERE firstname LIKE 'D%'
```
```
test-> ORDER BY firstname;
```
```
<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dean</td>
<td>Yeager</td>
<td>Plymouth</td>
<td>MA</td>
<td>24</td>
</tr>
<tr>
<td>Dick</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
<td>19</td>
</tr>
</tbody>
</table>
```
(2 rows)

Figure 4.16: **Firstname** begins with D
<table>
<thead>
<tr>
<th>Comparison</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>regular expression</td>
<td>~</td>
</tr>
<tr>
<td>regular expression, case-insensitive</td>
<td>~*</td>
</tr>
<tr>
<td>not equal to regular expression</td>
<td>!~</td>
</tr>
<tr>
<td>not equal to regular expression, case-insensitive</td>
<td>!~*</td>
</tr>
</tbody>
</table>

Table 4.4: Regular expression operators

<table>
<thead>
<tr>
<th>Test</th>
<th>Special Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>^</td>
</tr>
<tr>
<td>end</td>
<td>$</td>
</tr>
<tr>
<td>any single character</td>
<td>.</td>
</tr>
<tr>
<td>set of characters</td>
<td>[ccc]</td>
</tr>
<tr>
<td>set of characters not equal</td>
<td>[^ccc]</td>
</tr>
<tr>
<td>range of characters</td>
<td>[c-c]</td>
</tr>
<tr>
<td>range of characters not equal</td>
<td>[^c-c]</td>
</tr>
<tr>
<td>zero or one of previous character</td>
<td>?</td>
</tr>
<tr>
<td>zero or multiple of previous characters</td>
<td>*</td>
</tr>
<tr>
<td>one or multiple of previous characters</td>
<td>+</td>
</tr>
<tr>
<td>OR operator</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Regular expression special characters
<table>
<thead>
<tr>
<th>Test</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>begins with D</td>
<td>~ '^[D]'</td>
</tr>
<tr>
<td>contains D</td>
<td>~ 'D'</td>
</tr>
<tr>
<td>D in second position</td>
<td>~ '^D'</td>
</tr>
<tr>
<td>begins with D and contains e</td>
<td>~ '^[D]*e'</td>
</tr>
<tr>
<td>begins with D, contains e, and then f</td>
<td>~ '^[D]*.e.*f'</td>
</tr>
<tr>
<td>contains A, B, C, or D</td>
<td>~ '^[A-D]' or ~ '^[ABCD]'</td>
</tr>
<tr>
<td>contains A or a</td>
<td>~* 'a' or ~ '^[Aa]'</td>
</tr>
<tr>
<td>does not contain D</td>
<td>~! 'D'</td>
</tr>
<tr>
<td>does not begin with D</td>
<td>~! '^[D]' or ~ '!^[D]'</td>
</tr>
<tr>
<td>begins with D, with one optional leading space</td>
<td>~ '^[?]D'</td>
</tr>
<tr>
<td>begins with D, with optional leading spaces</td>
<td>~ '^[*D]'</td>
</tr>
<tr>
<td>begins with D, with at least one leading space</td>
<td>~ '^[+D]'</td>
</tr>
<tr>
<td>ends with G, with optional trailing spaces</td>
<td>~ 'G *$'</td>
</tr>
</tbody>
</table>

Table 4.6: Examples of regular expressions
test=> SELECT * FROM friend
test=> ORDER BY firstname;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dean</td>
<td>Yeager</td>
<td>Plymouth</td>
<td>MA</td>
<td>24</td>
</tr>
<tr>
<td>Dick</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
<td>19</td>
</tr>
<tr>
<td>Ned</td>
<td>Millstone</td>
<td>Cedar Creek</td>
<td>MD</td>
<td>27</td>
</tr>
<tr>
<td>Sandy</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
<td>25</td>
</tr>
<tr>
<td>Sandy</td>
<td>Weber</td>
<td>Boston</td>
<td>MA</td>
<td>33</td>
</tr>
<tr>
<td>Victor</td>
<td>Tabor</td>
<td>Williamsport</td>
<td>PA</td>
<td>22</td>
</tr>
</tbody>
</table>

(6 rows)

test=> -- firstname begins with 'S'
test=> SELECT * FROM friend
test=> WHERE firstname = '^S'
test=> ORDER BY firstname;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sandy</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
<td>25</td>
</tr>
<tr>
<td>Sandy</td>
<td>Weber</td>
<td>Boston</td>
<td>MA</td>
<td>33</td>
</tr>
</tbody>
</table>

(2 rows)

test=> -- firstname has an e in the second position

test=> SELECT * FROM friend

test=> WHERE firstname = '^\..e'

test=> ORDER BY firstname;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dean</td>
<td>Yeager</td>
<td>Plymouth</td>
<td>MA</td>
<td>24</td>
</tr>
<tr>
<td>Ned</td>
<td>Millstone</td>
<td>Cedar Creek</td>
<td>MD</td>
<td>27</td>
</tr>
</tbody>
</table>

(2 rows)

test=> -- firstname contains b, B, c, or C

test=> SELECT * FROM friend

test=> WHERE firstname =~ '[bc]'

test=> ORDER BY firstname;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dick</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
<td>19</td>
</tr>
<tr>
<td>Victor</td>
<td>Tabor</td>
<td>Williamsport</td>
<td>PA</td>
<td>22</td>
</tr>
</tbody>
</table>

(2 rows)

test=> -- firstname does not contain s or S

test=> SELECT * FROM friend

test=> WHERE firstname !~ 's'

test=> ORDER BY firstname;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dean</td>
<td>Yeager</td>
<td>Plymouth</td>
<td>MA</td>
<td>24</td>
</tr>
<tr>
<td>Dick</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
<td>19</td>
</tr>
<tr>
<td>Ned</td>
<td>Millstone</td>
<td>Cedar Creek</td>
<td>MD</td>
<td>27</td>
</tr>
<tr>
<td>Victor</td>
<td>Tabor</td>
<td>Williamsport</td>
<td>PA</td>
<td>22</td>
</tr>
</tbody>
</table>

(4 rows)

Figure 4.17: Regular expression sample queries
Figure 4.18: Complex regular expression 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)
```
test=> SELECT firstname, age,
CASE WHEN age >= 21 THEN 'adult'
ELSE 'minor'
END FROM friend
ORDER BY firstname;

<table>
<thead>
<tr>
<th>firstname</th>
<th>age</th>
<th>case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dean</td>
<td>24</td>
<td>adult</td>
</tr>
<tr>
<td>Dick</td>
<td>19</td>
<td>minor</td>
</tr>
<tr>
<td>Ned</td>
<td>27</td>
<td>adult</td>
</tr>
<tr>
<td>Sandy</td>
<td>25</td>
<td>adult</td>
</tr>
<tr>
<td>Sandy</td>
<td>33</td>
<td>adult</td>
</tr>
<tr>
<td>Victor</td>
<td>22</td>
<td>adult</td>
</tr>
</tbody>
</table>

(6 rows)

Figure 4.19: CASE example
```
SELECT firstname,
       state,
       CASE
           WHEN state = 'PA' THEN 'close'
           WHEN state = 'NJ' OR state = 'MD' THEN 'far'
           ELSE 'very far'
        END AS distance
FROM friend
ORDER BY firstname;
```

<table>
<thead>
<tr>
<th>firstname</th>
<th>state</th>
<th>distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dean</td>
<td>MA</td>
<td>very far</td>
</tr>
<tr>
<td>Dick</td>
<td>NJ</td>
<td>far</td>
</tr>
<tr>
<td>Ned</td>
<td>MD</td>
<td>far</td>
</tr>
<tr>
<td>Sandy</td>
<td>NJ</td>
<td>far</td>
</tr>
<tr>
<td>Sandy</td>
<td>MA</td>
<td>very far</td>
</tr>
<tr>
<td>Victor</td>
<td>PA</td>
<td>close</td>
</tr>
</tbody>
</table>

(6 rows)

Figure 4.20: Complex CASE example
<table>
<thead>
<tr>
<th>city</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
<td>MA</td>
</tr>
<tr>
<td>Plymouth</td>
<td>MA</td>
</tr>
<tr>
<td>Cedar Creek</td>
<td>MD</td>
</tr>
<tr>
<td>Ocean City</td>
<td>NJ</td>
</tr>
<tr>
<td>Williamsport</td>
<td>PA</td>
</tr>
</tbody>
</table>

Figure 4.21: DISTINCT prevents duplicates
<table>
<thead>
<tr>
<th>Function</th>
<th>Set option</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATESTYLE</td>
<td>DATESTYLE TO 'ISO'</td>
</tr>
<tr>
<td>TIMEZONE</td>
<td>TIMEZONE TO 'value'</td>
</tr>
</tbody>
</table>

Table 4.7: Set options

<table>
<thead>
<tr>
<th>Style</th>
<th>Optional Ordering</th>
<th>Output for February 1, 1983</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO</td>
<td></td>
<td>1983-02-01</td>
</tr>
<tr>
<td>POSTGRES POSTGRES</td>
<td>US or NONEUROPEAN</td>
<td>02-01-1983</td>
</tr>
<tr>
<td>POSTGRES</td>
<td>EUROPEAN</td>
<td>01-02-1983</td>
</tr>
<tr>
<td>SQL</td>
<td>US or NONEUROPEAN</td>
<td>02/01/1983</td>
</tr>
<tr>
<td>SQL</td>
<td>EUROPEAN</td>
<td>01/02/1983</td>
</tr>
<tr>
<td>German</td>
<td></td>
<td>01.02.1983</td>
</tr>
</tbody>
</table>

Table 4.8: DATESTYLE output
### List of functions

<table>
<thead>
<tr>
<th>Result</th>
<th>Function</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>_bpchar</td>
<td>_bpchar</td>
<td>_bpchar int4</td>
</tr>
<tr>
<td>_varchar</td>
<td>_varchar</td>
<td>_varchar int4</td>
</tr>
<tr>
<td>float4</td>
<td>abs</td>
<td>float4</td>
</tr>
<tr>
<td>float8</td>
<td>abs</td>
<td>float8</td>
</tr>
</tbody>
</table>

### List of functions

<table>
<thead>
<tr>
<th>Result</th>
<th>Function</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>int2</td>
<td>int2</td>
<td>float4</td>
</tr>
<tr>
<td>int2</td>
<td>int2</td>
<td>float8</td>
</tr>
<tr>
<td>int2</td>
<td>int2</td>
<td>int2</td>
</tr>
<tr>
<td>int2</td>
<td>int2</td>
<td>int4</td>
</tr>
</tbody>
</table>

### List of functions

<table>
<thead>
<tr>
<th>Result</th>
<th>Function</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>upper</td>
<td>text</td>
</tr>
</tbody>
</table>

(1 row)

### Object descriptions

<table>
<thead>
<tr>
<th>Name</th>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>upper</td>
<td>function</td>
<td>uppercase</td>
</tr>
</tbody>
</table>

(1 row)

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

```sql
test=> SELECT sqrt(2.0); -- square root
  sqrt
     ---------
     1.4142135623731
(1 row)
```
### List of operators

<table>
<thead>
<tr>
<th>Op</th>
<th>Left arg</th>
<th>Right arg</th>
<th>Result</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>int2</td>
<td></td>
<td>int4</td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>int4</td>
<td></td>
<td>int4</td>
<td>factorial</td>
</tr>
<tr>
<td>!</td>
<td>int8</td>
<td></td>
<td>int8</td>
<td>factorial</td>
</tr>
<tr>
<td>!!</td>
<td></td>
<td>int2</td>
<td>int4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op</th>
<th>Left arg</th>
<th>Right arg</th>
<th>Result</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>box</td>
<td>point</td>
<td>box</td>
<td>divide box by point (scale)</td>
</tr>
<tr>
<td>/</td>
<td>char</td>
<td>char</td>
<td>char</td>
<td>divide</td>
</tr>
<tr>
<td>/</td>
<td>circle</td>
<td>point</td>
<td>circle</td>
<td>divide</td>
</tr>
<tr>
<td>/</td>
<td>float4</td>
<td>float4</td>
<td>float4</td>
<td>divide</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op</th>
<th>Left arg</th>
<th>Right arg</th>
<th>Result</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>float8</td>
<td>float8</td>
<td>float8</td>
<td>exponentiation (x^y)</td>
</tr>
</tbody>
</table>

(1 row)

### Object descriptions

<table>
<thead>
<tr>
<th>Name</th>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>operator</td>
<td>exponentiation (x^y)</td>
</tr>
</tbody>
</table>

(1 row)

```
test=> \do

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
# SQL Aggregates

<table>
<thead>
<tr>
<th>Aggregate</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT(*)</td>
<td>count of rows</td>
</tr>
<tr>
<td>SUM(colname)</td>
<td>total</td>
</tr>
<tr>
<td>MAX(colname)</td>
<td>maximum</td>
</tr>
<tr>
<td>MIN(colname)</td>
<td>minimum</td>
</tr>
<tr>
<td>AVG(colname)</td>
<td>average</td>
</tr>
</tbody>
</table>

Table 5.1: Aggregates
test=> SELECT * FROM friend ORDER BY firstname;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dean</td>
<td>Yeager</td>
<td>Plymouth</td>
<td>MA</td>
<td>24</td>
</tr>
<tr>
<td>Dick</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
<td>19</td>
</tr>
<tr>
<td>Ned</td>
<td>Millstone</td>
<td>Cedar Creek</td>
<td>MD</td>
<td>27</td>
</tr>
<tr>
<td>Sandy</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
<td>25</td>
</tr>
<tr>
<td>Sandy</td>
<td>Weber</td>
<td>Boston</td>
<td>MA</td>
<td>33</td>
</tr>
<tr>
<td>Victor</td>
<td>Tabor</td>
<td>Williamsport</td>
<td>PA</td>
<td>22</td>
</tr>
</tbody>
</table>

(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
Figure 5.2: Aggregates and NULL values
**Figure 5.3: Aggregate with GROUP BY**

```sql
test=> SELECT state, COUNT(*)
    FROM friend
    GROUP BY state;

<table>
<thead>
<tr>
<th>state</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>MA</td>
<td>2</td>
</tr>
<tr>
<td>MD</td>
<td>1</td>
</tr>
<tr>
<td>NJ</td>
<td>2</td>
</tr>
<tr>
<td>PA</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```sql
(4 rows)
```

```sql
test=> SELECT state, MIN(age), MAX(age), AVG(age)
    FROM friend
    GROUP BY state
    ORDER BY 4 DESC;

<table>
<thead>
<tr>
<th>state</th>
<th>min</th>
<th>max</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>MA</td>
<td>24</td>
<td>33</td>
<td>28</td>
</tr>
<tr>
<td>MD</td>
<td>27</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>NJ</td>
<td>19</td>
<td>25</td>
<td>22</td>
</tr>
<tr>
<td>PA</td>
<td>22</td>
<td>22</td>
<td>22</td>
</tr>
</tbody>
</table>
```

```sql
(4 rows)
```
test=> SELECT city, state, COUNT(*)
    FROM friend
    GROUP BY state, city
    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(*)
    FROM friend
    GROUP BY state
    HAVING COUNT(*) > 1
    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
CREATE TABLE customer (
  customer_id INTEGER,
  name CHAR(30),
  telephone CHAR(20),
  street CHAR(40),
  city CHAR(25),
  state CHAR(2),
  zipcode CHAR(10),
  country CHAR(20)
);

CREATE TABLE employee (
  employee_id INTEGER,
  name CHAR(30),
  hire_date DATE
);

CREATE TABLE part (
  part_id INTEGER,
  name CHAR(30),
  cost NUMERIC(8,2),
  weight FLOAT
);

CREATE TABLE salesorder (
  order_id INTEGER,
  customer_id INTEGER,  -- joins to customer.customer_id
  employee_id INTEGER,  -- joins to employee.employee_id
  part_id INTEGER,  -- joins to part.part_id
  order_date DATE,
  ship_date DATE,
  payment NUMERIC(8,2)
);
test=> INSERT INTO customer VALUES ( 
  648, 
  'Fleer Gearworks, Inc.', 
  '1-610-555-7829', 
  '830 Winding Way', 
  'Millersville', 
  'AL', 
  '35041', 
  'USA' 
); 

INSERT 19815 1 

test=> INSERT INTO employee VALUES ( 
  24, 
  'Lee Meyers', 
  '10/16/1989' 
); 

INSERT 19816 1 

test=> INSERT INTO part VALUES ( 
  153, 
  'Garage Door Spring', 
  6.20 
); 

INSERT 19817 1 

test=> INSERT INTO salesorder VALUES( 
  14673, 
  648, 
  24, 
  153, 
  '7/19/1994', 
  '7/28/1994', 
  18.39 
); 

INSERT 19818 1 

Figure 6.4: Insertion into company tables
Figure 6.5: Finding a customer name using two queries

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;

<table>
<thead>
<tr>
<th>order_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>14673</td>
</tr>
</tbody>
</table>

(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;

<table>
<thead>
<tr>
<th>name</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fleer Gearworks, Inc.</td>
<td>Lee Meyers</td>
</tr>
</tbody>
</table>

(1 row)

Figure 6.8: Three-table join
```sql
test=> SELECT customer.name AS customer_name,
      employee.name AS employee_name,
      part.name AS part_name
FROM salesorder, customer, employee, part
WHERE salesorder.customer_id = customer.customer_id AND
      salesorder.employee_id = employee.employee_id AND
      salesorder.part_id = part.part_id AND
      salesorder.order_id = 14673;
```

<table>
<thead>
<tr>
<th>customer_name</th>
<th>employee_name</th>
<th>part_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fleer Gearworks, Inc.</td>
<td>Lee Meyers</td>
<td>Garage Door Spring</td>
</tr>
</tbody>
</table>

(1 row)

Figure 6.9: Four-table join
```
test=> SELECT DISTINCT customer.name, employee.name
    FROM customer, employee, salesorder
    WHERE customer.customer_id = salesorder.customer_id and 
    salesorder.employee_id = employee.employee_id
ORDER BY customer.name, employee.name;

<table>
<thead>
<tr>
<th>name</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fleer Gearworks, Inc.</td>
<td>Lee Meyers</td>
</tr>
</tbody>
</table>
(1 row)

test=> SELECT DISTINCT customer.name, employee.name, COUNT(*)
    FROM customer, employee, salesorder
    WHERE customer.customer_id = salesorder.customer_id and 
    salesorder.employee_id = employee.employee_id
GROUP BY customer.name, employee.name
ORDER BY customer.name, employee.name;

<table>
<thead>
<tr>
<th>name</th>
<th>name</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fleer Gearworks, Inc.</td>
<td>Lee Meyers</td>
<td>1</td>
</tr>
</tbody>
</table>
(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
```sql
CREATE TABLE statename (code CHAR(2),
                        name CHAR(30));

CREATE

INSERT INTO statename VALUES ('AL', 'Alabama');

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

Figure 6.13: The `statename` table

```sql
SELECT order_id
FROM customer, salesorder
WHERE customer.code = 'FLED01' 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 *
  FROM animal, vegetable
  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 * 
    FROM animal, vegetable;  

<table>
<thead>
<tr>
<th>animal_id</th>
<th>name</th>
<th>animal_id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>507</td>
<td>rabbit</td>
<td>507</td>
<td>lettuce</td>
</tr>
<tr>
<td>508</td>
<td>cat</td>
<td>507</td>
<td>lettuce</td>
</tr>
<tr>
<td>507</td>
<td>rabbit</td>
<td>507</td>
<td>carrot</td>
</tr>
<tr>
<td>508</td>
<td>cat</td>
<td>507</td>
<td>carrot</td>
</tr>
<tr>
<td>507</td>
<td>rabbit</td>
<td>507</td>
<td>nut</td>
</tr>
<tr>
<td>508</td>
<td>cat</td>
<td>507</td>
<td>nut</td>
</tr>
</tbody>
</table>

(6 rows)

Figure 6.16: Unjoined tables

SELECT order_id 
FROM customer c, salesorder s 
WHERE c.code = 'FLE001' 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;
od | age
-------+-----
21515 | 7 (1 row)

Figure 7.1: OID test

<table>
<thead>
<tr>
<th>Function</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>nextval('name')</td>
<td>Returns the next available sequence number, and updates the counter</td>
</tr>
<tr>
<td>currval('name')</td>
<td>Returns the sequence number from the previous nextval() call</td>
</tr>
<tr>
<td>setval('name', newval)</td>
<td>Sets the sequence number counter to the specified value</td>
</tr>
</tbody>
</table>

Table 7.1: Sequence number access functions
```sql
CREATE TABLE salesorder (  
  order_id INTEGER,  
  customer_oid OID, -- joins to customer.oid  
  employee_oid OID, -- joins to employee.oid  
  part_oid OID, -- joins to part.oid
)
```

Figure 7.2: Columns with OIDs
Figure 7.3: Examples of sequence function use
test=> CREATE SEQUENCE customer_seq;
CREATE

CREATE

test=> CREATE TABLE customer ( 
  customer_id INTEGER DEFAULT nextval('customer_seq'),
  name CHAR(30)
);

CREATE

test=> INSERT INTO customer VALUES (nextval('customer_seq'), 'BreadMakers');
INSERT 19004 1

INSERT INTO customer (name) VALUES ('Wax Carvers');
INSERT 19005 1

INSERT INTO customer (name) VALUES ('Pipe Fitters');
INSERT 19008 1

SELECT * FROM customer;

<table>
<thead>
<tr>
<th>customer_id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bread Makers</td>
</tr>
<tr>
<td>2</td>
<td>Wax Carvers</td>
</tr>
<tr>
<td>3</td>
<td>Pipe Fitters</td>
</tr>
</tbody>
</table>

(3 rows)

Figure 7.4: Numbering customer rows using a sequence
test=> CREATE TABLE customer (  
customer_id SERIAL,  
name CHAR(30)  
);

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"

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>customer_id</td>
<td>int4</td>
<td>not null default nextval('customer_customer_id_seq'::text)</td>
</tr>
<tr>
<td>name</td>
<td>char(30)</td>
<td></td>
</tr>
</tbody>
</table>

Index: customer_customer_id_key
test=> INSERT INTO customer (name) VALUES ('Car Wash');
INSERT 19152 1
test=> SELECT * FROM customer;

<table>
<thead>
<tr>
<th>customer_id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Car Wash</td>
</tr>
</tbody>
</table>

(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;

<table>
<thead>
<tr>
<th>firstname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dean</td>
</tr>
<tr>
<td>Dick</td>
</tr>
<tr>
<td>Gleason</td>
</tr>
<tr>
<td>Millstone</td>
</tr>
<tr>
<td>Ned</td>
</tr>
<tr>
<td>Sandy</td>
</tr>
<tr>
<td>Tabor</td>
</tr>
<tr>
<td>Victor</td>
</tr>
<tr>
<td>Weber</td>
</tr>
<tr>
<td>Yeager</td>
</tr>
</tbody>
</table>

(10 rows)

Figure 8.1: Combining two columns with UNION
```sql
TEST=> INSERT INTO terrestrial_animal (name) VALUES ('tiger');
 INSERT 191221
TEST=> INSERT INTO aquatic_animal (name) VALUES ('swordfish');
 INSERT 191231
TEST=> SELECT name
FROM aquatic_animal
UNION
SELECT name
FROM terrestrial_animal;

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>swordfish</td>
</tr>
<tr>
<td>tiger</td>
</tr>
</tbody>
</table>

(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
FROM aquatic_animal
UNION
SELECT name
FROM terrestrial_animal;

--------------------------------
penguin  swordfish  tiger
(3 rows)

Figure 8.3: UNION with duplicates

test=> SELECT name
FROM aquatic_animal
UNION ALL
SELECT name
FROM terrestrial_animal;

--------------------------------
swordfish  penguin  tiger
(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;

-------------------------
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;

-------------------------
penguin
(1 row)

Figure 8.6: INTERSECT returns only duplicated rows
```
test=> SELECT * FROM friend ORDER BY firstname;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dean</td>
<td>Yeager</td>
<td>Plymouth</td>
<td>MA</td>
<td>24</td>
</tr>
<tr>
<td>Dick</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
<td>19</td>
</tr>
<tr>
<td>Ned</td>
<td>Millstone</td>
<td>Cedar Creek</td>
<td>MD</td>
<td>27</td>
</tr>
<tr>
<td>Sandy</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
<td>25</td>
</tr>
<tr>
<td>Sandy</td>
<td>Weber</td>
<td>Boston</td>
<td>MA</td>
<td>33</td>
</tr>
<tr>
<td>Victor</td>
<td>Tabor</td>
<td>Williamsport</td>
<td>PA</td>
<td>22</td>
</tr>
</tbody>
</table>

(6 rows)
```

```
test=> SELECT f1.firstname, f1.lastname, f1.state
FROM friend f1, friend f2
WHERE f1.state <> f2.state AND
  f2.firstname = 'Dick' AND
  f2.lastname = 'Gleason'
ORDER BY firstname, lastname;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dean</td>
<td>Yeager</td>
<td>MA</td>
</tr>
<tr>
<td>Ned</td>
<td>Millstone</td>
<td>MD</td>
</tr>
<tr>
<td>Sandy</td>
<td>Weber</td>
<td>MA</td>
</tr>
<tr>
<td>Victor</td>
<td>Tabor</td>
<td>PA</td>
</tr>
</tbody>
</table>

(4 rows)
```

```
test=> SELECT f1.firstname, f1.lastname, f1.state
FROM friend f1
WHERE f1.state <> (  
SELECT f2.state
FROM friend f2
WHERE f2.firstname = 'Dick' AND
  f2.lastname = 'Gleason'
)
ORDER BY firstname, lastname;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dean</td>
<td>Yeager</td>
<td>MA</td>
</tr>
<tr>
<td>Ned</td>
<td>Millstone</td>
<td>MD</td>
</tr>
<tr>
<td>Sandy</td>
<td>Weber</td>
<td>MA</td>
</tr>
<tr>
<td>Victor</td>
<td>Tabor</td>
<td>PA</td>
</tr>
</tbody>
</table>

(4 rows)
```

Figure 8.7: Friends not in Dick Gleason’s state
Figure 8.8: Subqueries can replace some joins
```
SELECT f1.firstname, f1.lastname, f1.age
FROM friend f1, friend f2
WHERE f1.state = f2.state
GROUP BY f2.state, f1.firstname, f1.lastname, f1.age
HAVING f1.age = MAX(f2.age)
ORDER BY firstname, lastname;
```

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ned</td>
<td>Millstone</td>
<td>27</td>
</tr>
<tr>
<td>Sandy</td>
<td>Gleason</td>
<td>25</td>
</tr>
<tr>
<td>Sandy</td>
<td>Weber</td>
<td>33</td>
</tr>
<tr>
<td>Victor</td>
<td>Tabor</td>
<td>22</td>
</tr>
</tbody>
</table>

(4 rows)

```test=> SELECT f1.firstname, f1.lastname, f1.age
FROM friend f1
WHERE age = (SELECT MAX(f2.age)
FROM friend f2
WHERE f1.state = f2.state)
ORDER BY firstname, lastname;
```

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ned</td>
<td>Millstone</td>
<td>27</td>
</tr>
<tr>
<td>Sandy</td>
<td>Gleason</td>
<td>25</td>
</tr>
<tr>
<td>Sandy</td>
<td>Weber</td>
<td>33</td>
</tr>
<tr>
<td>Victor</td>
<td>Tabor</td>
<td>22</td>
</tr>
</tbody>
</table>

(4 rows)

Figure 8.9: Correlated subquery
test=> SELECT DISTINCT employee.name
FROM employee, salesorder
WHERE employee.employee_id = salesorder.employee_id AND
salesorder.order_date = '7/19/1994';

Lee Meyers
(1 row)

Figure 8.10: Employees who took orders

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

Lee Meyers
(1 row)

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
DELETE FROM customer
WHERE customer_id NOT IN (
    SELECT customer_id
    FROM salesorder
);

UPDATE salesorder
SET ship_date = '11/16/96'
WHERE customer_id = (
    SELECT customer_id
    FROM customer
    WHERE name = 'Fleer Gearworks, Inc.'
);

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

INSERT INTO customer (name, city, state, country)
SELECT trim(firstname) || ' ' || lastname, city, state, 'USA'
FROM friend;

Figure 8.15: Subqueries with UPDATE and DELETE

Figure 8.16: UPDATE the order_date

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"

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>firstname</td>
<td>char(15)</td>
<td></td>
</tr>
<tr>
<td>lastname</td>
<td>char(20)</td>
<td></td>
</tr>
<tr>
<td>city</td>
<td>char(15)</td>
<td></td>
</tr>
<tr>
<td>state</td>
<td>char(2)</td>
<td></td>
</tr>
</tbody>
</table>

SELECT * FROM newfriend ORDER BY firstname;

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dean</td>
<td>Yeager</td>
<td>Plymouth</td>
<td>MA</td>
</tr>
<tr>
<td>Dick</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
</tr>
<tr>
<td>Ned</td>
<td>Millstone</td>
<td>Cedar Creek</td>
<td>MD</td>
</tr>
<tr>
<td>Sandy</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
</tr>
<tr>
<td>Sandy</td>
<td>Weber</td>
<td>Boston</td>
<td>MA</td>
</tr>
<tr>
<td>Victor</td>
<td>Tabor</td>
<td>Williamsport</td>
<td>PA</td>
</tr>
</tbody>
</table>

(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
Judy
(1 row)

Figure 9.1: Example of a function call
<table>
<thead>
<tr>
<th>Category</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character string</td>
<td>TEXT</td>
<td>variable storage length</td>
</tr>
<tr>
<td></td>
<td>VARCHAR(length)</td>
<td>variable storage length with maximum length</td>
</tr>
<tr>
<td></td>
<td>CHAR(length)</td>
<td>fixed storage length, blank-padded to length, internally BPCHAR</td>
</tr>
<tr>
<td>Number</td>
<td>INTEGER</td>
<td>integer, ±2 billion range, internally INT4</td>
</tr>
<tr>
<td></td>
<td>INT2</td>
<td>integer, ±32 thousand range</td>
</tr>
<tr>
<td></td>
<td>INT8</td>
<td>integer, ±4 × 10^18 range</td>
</tr>
<tr>
<td></td>
<td>OID</td>
<td>object identifier</td>
</tr>
<tr>
<td></td>
<td>NUMERIC(precision, decimal)</td>
<td>number, user-defined precision and decimal location</td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td>floating-point number, 15-digit precision, internally FLOAT8</td>
</tr>
<tr>
<td></td>
<td>FLOAT4</td>
<td>floating-point number, 6-digit precision</td>
</tr>
<tr>
<td>Temporal</td>
<td>DATE</td>
<td>date</td>
</tr>
<tr>
<td></td>
<td>TIME</td>
<td>time</td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP</td>
<td>date and time</td>
</tr>
<tr>
<td></td>
<td>INTERVAL</td>
<td>interval of time</td>
</tr>
<tr>
<td>Logical</td>
<td>BOOLEAN</td>
<td>boolean, true or false</td>
</tr>
<tr>
<td>Geometric</td>
<td>POINT</td>
<td>point</td>
</tr>
<tr>
<td></td>
<td>LSEG</td>
<td>line segment</td>
</tr>
<tr>
<td></td>
<td>PATH</td>
<td>list of points</td>
</tr>
<tr>
<td></td>
<td>BOX</td>
<td>rectangle</td>
</tr>
<tr>
<td></td>
<td>CIRCLE</td>
<td>circle</td>
</tr>
<tr>
<td></td>
<td>POLYGON</td>
<td>polygon</td>
</tr>
<tr>
<td>Network</td>
<td>INET</td>
<td>IP address with optional netmask</td>
</tr>
<tr>
<td></td>
<td>CIDR</td>
<td>IP network address</td>
</tr>
<tr>
<td></td>
<td>MACADDR</td>
<td>Ethernet MAC address</td>
</tr>
</tbody>
</table>

Table 9.1: PostgreSQL data types
### Table 9.2: Geometric types

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

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

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.
<table>
<thead>
<tr>
<th>Type</th>
<th>Function</th>
<th>Example</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>length()</td>
<td>length(col)</td>
<td>length of col</td>
</tr>
<tr>
<td>String</td>
<td>character_length()</td>
<td>character_length(col)</td>
<td>length of col, same as length()</td>
</tr>
<tr>
<td></td>
<td>octet_length()</td>
<td>octet_length(col)</td>
<td>length of col, including multibyte overhead</td>
</tr>
<tr>
<td></td>
<td>trim()</td>
<td>trim(col)</td>
<td>col with leading and trailing spaces removed</td>
</tr>
<tr>
<td></td>
<td>trim(BOTH...)</td>
<td>trim(BOTH col)</td>
<td>same as trim()</td>
</tr>
<tr>
<td></td>
<td>trim(LEADING...)</td>
<td>trim(LEADING col)</td>
<td>col with leading spaces removed</td>
</tr>
<tr>
<td></td>
<td>trim.TRAILING(...)</td>
<td>trim(TRAILING col)</td>
<td>col with trailing spaces removed</td>
</tr>
<tr>
<td></td>
<td>trim(...FROM...)</td>
<td>trim(str FROM col)</td>
<td>col with leading and trailing str removed</td>
</tr>
<tr>
<td></td>
<td>rpad()</td>
<td>rpad(col, len)</td>
<td>col padded on the right to len characters</td>
</tr>
<tr>
<td></td>
<td>lpad()</td>
<td>lpad(col, len)</td>
<td>col padded on the left to len characters</td>
</tr>
<tr>
<td></td>
<td>upper()</td>
<td>upper(col)</td>
<td>col uppercased</td>
</tr>
<tr>
<td></td>
<td>lower()</td>
<td>lower(col)</td>
<td>col lowercased</td>
</tr>
<tr>
<td></td>
<td>initcap()</td>
<td>initcap(col)</td>
<td>col with the first letter capitalized</td>
</tr>
<tr>
<td></td>
<td>strpos()</td>
<td>strpos(col, str)</td>
<td>position of str in col</td>
</tr>
<tr>
<td></td>
<td>position()</td>
<td>position(str IN col)</td>
<td>same as strpos()</td>
</tr>
<tr>
<td></td>
<td>substr()</td>
<td>substr(col, pos)</td>
<td>col starting at position pos</td>
</tr>
<tr>
<td></td>
<td>substring(...FROM...)</td>
<td>substring(col FROM pos)</td>
<td>same as substr()</td>
</tr>
<tr>
<td></td>
<td>sub_string()</td>
<td>sub_string(col FROM pos, len)</td>
<td>col starting at position pos for length len</td>
</tr>
<tr>
<td></td>
<td>translate()</td>
<td>translate(col, from, to)</td>
<td>col with from characters mapped to to</td>
</tr>
<tr>
<td></td>
<td>to_number()</td>
<td>to_number(col, mask)</td>
<td>convert col to NUMERIC() based on mask</td>
</tr>
<tr>
<td></td>
<td>to_date()</td>
<td>to_date(col, mask)</td>
<td>convert col to DATE based on mask</td>
</tr>
<tr>
<td></td>
<td>to_timestamp()</td>
<td>to_timestamp(col, mask)</td>
<td>convert col to TIMESTAMP based on mask</td>
</tr>
<tr>
<td>Number</td>
<td>round()</td>
<td>round(col)</td>
<td>round to an integer</td>
</tr>
<tr>
<td></td>
<td>round()</td>
<td>round(col, len)</td>
<td>NUMERIC() col rounded to len decimal places</td>
</tr>
<tr>
<td></td>
<td>trunc()</td>
<td>trunc(col)</td>
<td>truncate to an integer</td>
</tr>
<tr>
<td></td>
<td>trunc()</td>
<td>trunc(col, len)</td>
<td>NUMERIC() col truncated to len decimal places</td>
</tr>
<tr>
<td></td>
<td>abs()</td>
<td>abs(col)</td>
<td>absolute value</td>
</tr>
<tr>
<td></td>
<td>factorial()</td>
<td>factorial(col)</td>
<td>factorial</td>
</tr>
<tr>
<td></td>
<td>sqrt()</td>
<td>sqrt(col)</td>
<td>square root</td>
</tr>
<tr>
<td></td>
<td>cbrt()</td>
<td>cbrt(col)</td>
<td>cube root</td>
</tr>
<tr>
<td></td>
<td>exp()</td>
<td>exp(col)</td>
<td>exponential</td>
</tr>
<tr>
<td></td>
<td>ln()</td>
<td>ln(col)</td>
<td>natural logarithm</td>
</tr>
<tr>
<td></td>
<td>log()</td>
<td>log(col)</td>
<td>base-10 logarithm</td>
</tr>
<tr>
<td></td>
<td>to_char()</td>
<td>to_char(col, mask)</td>
<td>convert col to a string based on mask</td>
</tr>
<tr>
<td></td>
<td>date_part()</td>
<td>date_part(units, col)</td>
<td>units part of col</td>
</tr>
<tr>
<td></td>
<td>extract(...)</td>
<td>extract(units FROM col)</td>
<td>same as date_part()</td>
</tr>
<tr>
<td></td>
<td>date_trunc()</td>
<td>date_trunc(units, col)</td>
<td>col rounded to units</td>
</tr>
<tr>
<td></td>
<td>isfinite()</td>
<td>isfinite(col)</td>
<td>BOOLEAN indicating whether col is a valid date</td>
</tr>
<tr>
<td></td>
<td>now()</td>
<td>now()</td>
<td>TIMESTAMP representing current date and time</td>
</tr>
<tr>
<td></td>
<td>timeofday()</td>
<td>timeofday()</td>
<td>string showing date/time in Unix format</td>
</tr>
<tr>
<td></td>
<td>overlaps()</td>
<td>overlaps(c1, c2, c3, c4)</td>
<td>BOOLEAN indicating whether col's overlap in time</td>
</tr>
<tr>
<td></td>
<td>to_char()</td>
<td>to_char(col, mask)</td>
<td>convert col to string based on mask</td>
</tr>
<tr>
<td>Temporal</td>
<td>extract(...)</td>
<td>extract(units FROM col)</td>
<td>same as date_part()</td>
</tr>
<tr>
<td></td>
<td>date_trunc()</td>
<td>date_trunc(units, col)</td>
<td>col rounded to units</td>
</tr>
<tr>
<td></td>
<td>isfinite()</td>
<td>isfinite(col)</td>
<td>BOOLEAN indicating whether col is a valid date</td>
</tr>
<tr>
<td></td>
<td>now()</td>
<td>now()</td>
<td>TIMESTAMP representing current date and time</td>
</tr>
<tr>
<td></td>
<td>timeofday()</td>
<td>timeofday()</td>
<td>string showing date/time in Unix format</td>
</tr>
<tr>
<td></td>
<td>overlaps()</td>
<td>overlaps(c1, c2, c3, c4)</td>
<td>BOOLEAN indicating whether col's overlap in time</td>
</tr>
<tr>
<td></td>
<td>to_char()</td>
<td>to_char(col, mask)</td>
<td>convert col to string based on mask</td>
</tr>
<tr>
<td></td>
<td>date_part()</td>
<td>date_part(units, col)</td>
<td>units part of col</td>
</tr>
<tr>
<td></td>
<td>extract(...)</td>
<td>extract(units FROM col)</td>
<td>same as date_part()</td>
</tr>
<tr>
<td></td>
<td>date_trunc()</td>
<td>date_trunc(units, col)</td>
<td>col rounded to units</td>
</tr>
<tr>
<td></td>
<td>isfinite()</td>
<td>isfinite(col)</td>
<td>BOOLEAN indicating whether col is a valid date</td>
</tr>
<tr>
<td></td>
<td>now()</td>
<td>now()</td>
<td>TIMESTAMP representing current date and time</td>
</tr>
<tr>
<td></td>
<td>timeofday()</td>
<td>timeofday()</td>
<td>string showing date/time in Unix format</td>
</tr>
<tr>
<td></td>
<td>overlaps()</td>
<td>overlaps(c1, c2, c3, c4)</td>
<td>BOOLEAN indicating whether col's overlap in time</td>
</tr>
<tr>
<td></td>
<td>to_char()</td>
<td>to_char(col, mask)</td>
<td>convert col to string based on mask</td>
</tr>
<tr>
<td></td>
<td>date_part()</td>
<td>date_part(units, col)</td>
<td>units part of col</td>
</tr>
<tr>
<td></td>
<td>extract(...)</td>
<td>extract(units FROM col)</td>
<td>same as date_part()</td>
</tr>
<tr>
<td></td>
<td>date_trunc()</td>
<td>date_trunc(units, col)</td>
<td>col rounded to units</td>
</tr>
<tr>
<td></td>
<td>isfinite()</td>
<td>isfinite(col)</td>
<td>BOOLEAN indicating whether col is a valid date</td>
</tr>
<tr>
<td></td>
<td>now()</td>
<td>now()</td>
<td>TIMESTAMP representing current date and time</td>
</tr>
<tr>
<td></td>
<td>timeofday()</td>
<td>timeofday()</td>
<td>string showing date/time in Unix format</td>
</tr>
<tr>
<td></td>
<td>overlaps()</td>
<td>overlaps(c1, c2, c3, c4)</td>
<td>BOOLEAN indicating whether col's overlap in time</td>
</tr>
<tr>
<td></td>
<td>to_char()</td>
<td>to_char(col, mask)</td>
<td>convert col to string based on mask</td>
</tr>
<tr>
<td></td>
<td>date_part()</td>
<td>date_part(units, col)</td>
<td>units part of col</td>
</tr>
<tr>
<td></td>
<td>extract(...)</td>
<td>extract(units FROM col)</td>
<td>same as date_part()</td>
</tr>
<tr>
<td></td>
<td>date_trunc()</td>
<td>date_trunc(units, col)</td>
<td>col rounded to units</td>
</tr>
<tr>
<td></td>
<td>isfinite()</td>
<td>isfinite(col)</td>
<td>BOOLEAN indicating whether col is a valid date</td>
</tr>
<tr>
<td></td>
<td>now()</td>
<td>now()</td>
<td>TIMESTAMP representing current date and time</td>
</tr>
<tr>
<td></td>
<td>timeofday()</td>
<td>timeofday()</td>
<td>string showing date/time in Unix format</td>
</tr>
<tr>
<td></td>
<td>overlaps()</td>
<td>overlaps(c1, c2, c3, c4)</td>
<td>BOOLEAN indicating whether col's overlap in time</td>
</tr>
<tr>
<td></td>
<td>to_char()</td>
<td>to_char(col, mask)</td>
<td>convert col to string based on mask</td>
</tr>
</tbody>
</table>

**Table 9.3: Common functions**
<table>
<thead>
<tr>
<th>Type</th>
<th>Function</th>
<th>Example</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>--</td>
<td>col --</td>
<td>pattern</td>
</tr>
<tr>
<td></td>
<td>!-</td>
<td>col !-</td>
<td>pattern</td>
</tr>
<tr>
<td></td>
<td>-*</td>
<td>col -*</td>
<td>pattern</td>
</tr>
<tr>
<td></td>
<td>!-*</td>
<td>col !-*</td>
<td>pattern</td>
</tr>
<tr>
<td></td>
<td>--</td>
<td>col --</td>
<td>pattern</td>
</tr>
<tr>
<td>LIKE</td>
<td></td>
<td>col LIKE</td>
<td>pattern</td>
</tr>
<tr>
<td></td>
<td>!--</td>
<td>col !--</td>
<td>pattern</td>
</tr>
<tr>
<td>NOT LIKE</td>
<td></td>
<td>col NOT LIKE</td>
<td>pattern</td>
</tr>
<tr>
<td>Number</td>
<td>!</td>
<td>!col</td>
<td>factorial</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>col1 + col2</td>
<td>addition</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>col1 - col2</td>
<td>subtraction</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>col1 * col2</td>
<td>multiplication</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>col1 / col2</td>
<td>division</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>col1 % col2</td>
<td>remainder/modulo</td>
</tr>
<tr>
<td></td>
<td>^</td>
<td>col1 ^ col2</td>
<td>col1 raised to the power of col2</td>
</tr>
<tr>
<td>Temporal</td>
<td>+</td>
<td>col1 + col2</td>
<td>addition of temporal values</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>col1 - col2</td>
<td>subtraction of temporal values</td>
</tr>
<tr>
<td></td>
<td>(...) OVERLAPS</td>
<td>(c1, c2) OVERLAPS</td>
<td>BOOLEAN indicating cols overlap in time</td>
</tr>
<tr>
<td>Geometric</td>
<td></td>
<td></td>
<td>see psql’s doc for a list of geometric operators</td>
</tr>
<tr>
<td>Network</td>
<td>&lt;=</td>
<td>col1 &lt;= col2</td>
<td>BOOLEAN indicating if col1 is a subnet of col2</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
<td>col1 &lt;</td>
<td>col2</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>col1 &gt;=</td>
<td>col2</td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
<td>col1 &gt;</td>
<td>col2</td>
</tr>
</tbody>
</table>

Table 9.4: Common operators

<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT_DATE</td>
<td>current date</td>
</tr>
<tr>
<td>CURRENT_TIME</td>
<td>current time</td>
</tr>
<tr>
<td>CURRENT_TIMESTAMP</td>
<td>current date and time</td>
</tr>
<tr>
<td>CURRENT_USER</td>
<td>user connected to the database</td>
</tr>
</tbody>
</table>

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);

<table>
<thead>
<tr>
<th>?column?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1993-01-01 00:00:00-05</td>
</tr>
</tbody>
</table>
(1 row)

test=> SELECT CAST('1/1/1992' AS TIMESTAMP) + '1 year';

<table>
<thead>
<tr>
<th>?column?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1993-01-01 00:00:00-05</td>
</tr>
</tbody>
</table>
(1 row)

Figure 9.3: Error generated by undefined operator/type combination

test=> CREATE TABLE array_test (  
  col1 INTEGER[5],  
  col2 INTEGER[][],  
  col3 INTEGER[][2][])  
CREATE
test=> INSERT INTO array_test VALUES(
  '{1,2,3,4,5}',
  '{{1,2},{3,4}}',
  '{{{1,2},{3,4}},{{5,6},{7,8}}}'
);
INSERT 52694 1

test=> SELECT * FROM array_test;

<table>
<thead>
<tr>
<th>col1</th>
<th>col2</th>
<th>col3</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1,2,3,4,5}</td>
<td>{{1,2},{3,4}}</td>
<td>{{{1,2},{3,4}},{{5,6},{7,8}}}</td>
</tr>
</tbody>
</table>

(1 row)

Figure 9.5: Using arrays
CREATE TABLE fruit (name CHAR(30), image OID);

CREATE

INSERT INTO fruit VALUES ('peach', lo_import('/usr/images/peach.jpg'));

SELECT lo_export(fruit.image, '/tmp/outimage.jpg')
FROM fruit
WHERE name = 'peach';

----------------
  1
(1 row)

SELECT lo_unlink(fruit.image) FROM fruit;

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

---

<table>
<thead>
<tr>
<th>User 1</th>
<th>User 2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT INTO trans_test VALUES (1)</td>
<td>SELECT (*) FROM trans_test</td>
<td>returns 0</td>
</tr>
<tr>
<td>SELECT (*) FROM trans_test</td>
<td></td>
<td>add row to trans_test</td>
</tr>
<tr>
<td></td>
<td>SELECT (*) FROM trans_test</td>
<td>returns 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>returns 1</td>
</tr>
</tbody>
</table>

Table 10.1: Visibility of single-query transactions
User 1 | User 2 | Description
--- | --- | ---
BEGIN WORK | SELECT (*) FROM trans_test | User 1 starts a transaction
INSERT INTO trans_test VALUES (1) | add row to trans_test | returns 0
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
```sql
BEGIN WORK;
BEGIN
test=> SELECT COUNT(*) FROM trans_test;
  count
     -----
       5
(1 row)

--
--someone commits INSERT INTO trans_test
--

COUNT(*) FROM trans_test;
  count
     -----
       6
(1 row)

COMMIT WORK;
COMMIT
```

Figure 10.6: Read-committed isolation level

<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>Transaction 2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN WORK</td>
<td>BEGIN WORK</td>
<td>start both transactions</td>
</tr>
<tr>
<td>UPDATE row 64</td>
<td>UPDATE row 64</td>
<td>transaction 1 exclusively locks row 64</td>
</tr>
<tr>
<td>COMMIT WORK</td>
<td>COMMIT WORK</td>
<td>transaction 2 must wait to see if transaction 1 commits</td>
</tr>
</tbody>
</table>

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

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

Table 10.4: Deadlock
BEGIN WORK;
BEGIN

select *
from lock_test
where name = 'James';

id | name
--+-------------------------------
521 | James
(1 row)

the SELECTed row is not locked

UPDATE lock_test
SET name = 'Jim'
WHERE name = 'James';
UPDATE 1

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;

<table>
<thead>
<tr>
<th>customer_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

(3 rows)

test=> SELECT customer_id FROM customer ORDER BY customer_id LIMIT 3 OFFSET 997;

<table>
<thead>
<tr>
<th>customer_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>998</td>
</tr>
<tr>
<td>999</td>
</tr>
<tr>
<td>1000</td>
</tr>
</tbody>
</table>

(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

<table>
<thead>
<tr>
<th>User 1</th>
<th>User 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE TEMPORARY TABLE temptest (col INTEGER)</td>
<td>CREATE TEMPORARY TABLE temptest (col INTEGER)</td>
</tr>
<tr>
<td>INSERT INTO temptest VALUES (1)</td>
<td>INSERT INTO temptest VALUES (2)</td>
</tr>
<tr>
<td>SELECT col FROM temptest returns 1</td>
<td>SELECT col FROM temptest returns 2</td>
</tr>
</tbody>
</table>

Table 13.1: Temporary table isolation
Temporary table auto-destruction

```sql
$ 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
Figure 13.2: Example of temporary table use
```sql
CREATE TABLE altertest (col1 INTEGER);
ALTER TABLE altertest RENAME TO alterdemo;
ALTER TABLE alterdemo RENAME COLUMN col1 TO democol;
ALTER TABLE alterdemo ADD COLUMN col2 INTEGER;
-- show renamed table, renamed column, and new column
\d alterdemo

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Modifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>democol</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>col2</td>
<td>integer</td>
<td></td>
</tr>
</tbody>
</table>

ALTER TABLE alterdemo ALTER COLUMN col2 SET DEFAULT 0;
-- show new default value
\d alterdemo

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Modifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>democol</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>col2</td>
<td>integer</td>
<td>default 0</td>
</tr>
</tbody>
</table>

ALTER TABLE alterdemo ALTER COLUMN col2 DROP DEFAULT;

Figure 13.3: ALTER TABLE examples
test=> CREATE TABLE permtest (col INTEGER);
CREATE
-- now only the owner can use permtest
test=>--

-- now user 'meyers' can do SELECTs on permtest
CHANGE

-- now all users can perform all operations on permtest

Figure 13.4: Examples of the GRANT command

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
CREATE VIEW customer_ohio AS
SELECT *
FROM customer
WHERE state = 'OH';
CREATE 18908

-- let sanders see only Ohio customers
GRANT SELECT ON customer_ohio TO sanders;

CREATE VIEW customer_address AS
SELECT customer_id, name, street, city, state, zipcode, country
FROM customer;
CREATE 18909

-- create view that combines fields from two tables
CREATE VIEW customer_finance AS
SELECT customer.customer_id, customer.name, finance.credit_limit
FROM customer, finance
WHERE customer.customer_id = finance.customer_id;
CREATE 18910

Figure 13.8: Examples of views
test=> CREATE TABLE ruletest (col INTEGER);

CREATE

test=> CREATE RULE ruletest_insert AS -- rule name
    ON INSERT TO ruletest -- INSERT rule
    DO INSTEAD -- DO INSTEAD-type rule
    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
CREATE TABLE service_request (
  customer_id INTEGER,
  description text,
  cre_user text DEFAULT CURRENT_USER,
  cre_timestamp timestamp DEFAULT CURRENT_TIMESTAMP);

CREATE TABLE service_request_log (
  customer_id INTEGER,
  description text,
  mod_type char(1),
  mod_user text DEFAULT CURRENT_USER,
  mod_timestamp timestamp DEFAULT CURRENT_TIMESTAMP);

CREATE RULE service_request_update AS -- UPDATE rule
  ON UPDATE TO service_request
  DO
  INSERT INTO service_request_log (customer_id, description, mod_type)
  VALUES (old.customer_id, old.description, 'U');

CREATE RULE service_request_delete AS -- DELETE rule
  ON DELETE TO service_request
  DO
  INSERT INTO service_request_log (customer_id, description, mod_type)
  VALUES (old.customer_id, old.description, 'D');

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
-------------+--------------------------+----------+----------+------------------------

(2 rows)

Figure 13.11: Use of rules to log table changes
```sql
CREATE TABLE realtable (col INTEGER);
CREATE VIEW view_realtable AS SELECT * FROM realtable;
INSERT INTO realtable VALUES (1);
INSERT INTO view_realtable VALUES (2);
SELECT * FROM realtable;
  col
  ----
   1
   1
   (1 row)
SELECT * FROM view_realtable;
  col
  ----
   1
   (1 row)
```

Figure 13.12: Views ignore table modifications
CREATE RULE view_realtable_insert AS  -- INSERT rule
  ON INSERT TO view_realtable
  DO INSTEAD
  INSERT INTO realtable
  VALUES (new.col);

CREATE RULE view_realtable_update AS  -- UPDATE rule
  ON UPDATE TO view_realtable
  DO INSTEAD
  UPDATE realtable
  SET col = new.col
  WHERE col = old.col;

CREATE RULE view_realtable_delete AS  -- DELETE rule
  ON DELETE TO view_realtable
  DO INSTEAD
  DELETE FROM realtable
  WHERE col = old.col;

Figure 13.13: Rules to handle view modifications
Figure 13.14: Example of rules that handle view modifications
Chapter 14

Constraints

test=> CREATE TABLE not_null_test (  
  col1 INTEGER,  
  col2 INTEGER NOT NULL  
);  
CREATE  
test=> INSERT INTO not_null_test  
VALUES (1, NULL);  
ERROR: ExecAppend: Fail to add null value in not null attribute col2  
test=> INSERT INTO not_null_test (col1)  
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 (  
col1 INTEGER,  
col2 INTEGER NOT NULL DEFAULT 5
);
CREATE

test=> INSERT INTO not_null_with_default_test (col1)  
VALUES (1);
INSERT 148520 1

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
-----------+---------+----------
coll | integer |
Index: uniquetest_coll1_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 (  
coll1 INTEGER,  
coll2 INTEGER,  
UNIQUE (coll1, coll2)  
);
NOTICE: CREATE TABLE/UNIQUE will create implicit index 'uniquetest2_coll1_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 (
   col1 INTEGER,
   col2 INTEGER,
   PRIMARY KEY(col1, col2)
);
NOTICE: CREATE TABLE/PRIMARY KEY will create implicit index 'primarytest2_pkey' for table 'primarytest2'
CREATE

Figure 14.6: Example of a multicolumn PRIMARY KEY
```sql
CREATE TABLE statename (code CHAR(2) PRIMARY KEY,
                        name CHAR(30))
CREATE
INSERT INTO statename VALUES ('AL', 'Alabama');

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

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

Figure 14.7: Foreign key creation

Figure 14.8: Foreign key constraints
```
CREATE TABLE customer (
    customer_id INTEGER PRIMARY KEY,
    name CHAR(30),
    telephone CHAR(20),
    street CHAR(40),
    city CHAR(25),
    state CHAR(2),
    zipcode CHAR(10),
    country CHAR(20)
);

CREATE TABLE employee (
    employee_id INTEGER PRIMARY KEY,
    name CHAR(30),
    hire_date DATE
);

CREATE TABLE part (
    part_id INTEGER PRIMARY KEY,
    name CHAR(30),
    cost NUMERIC(8,2),
    weight FLOAT
);

CREATE TABLE salesorder (
    order_id INTEGER,
    customer_id INTEGER REFERENCES customer,
    employee_id INTEGER REFERENCES employee,
    part_id INTEGER REFERENCES part,
    order_date DATE,
    ship_date DATE,
    payment NUMERIC(8,2)
);
```

Figure 14.9: Creation of company tables using primary and foreign keys
CREATE TABLE customer (
  customer_id INTEGER,
  name CHAR(30),
  telephone CHAR(20),
  street CHAR(40),
  city CHAR(25),
  state CHAR(2) REFERENCES statename ON UPDATE CASCADE ON DELETE SET NULL,
  zipcode CHAR(10),
  country CHAR(20)
);

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 (col2 INTEGER REFERENCES primarytest ON UPDATE CASCADE ON DELETE NO ACTION);
NOTICE: CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
CREATE
test=> INSERT INTO primarytest values (1);
INSERT 1
INSERT 1
INSERT 1
(1 row)
test=> -- CASCADE UPDATE is performed

-- CASCADE UPDATE is performed

-- CASCADE UPDATE is performed

test=> UPDATE primarytest SET col = 2;
UPDATE 1

test=> SELECT * FROM foreigntest;
    col2
------
      2
(1 row)

test=> -- NO ACTION prevents deletion

test=> DELETE FROM primarytest;
ERROR: <unnamed> referential integrity violation - key in primarytest still referenced from foreigntest

test=> -- By deleting the foreign key first, the DELETE succeeds

-- By deleting the foreign key first, the DELETE succeeds

-- By deleting the foreign key first, the DELETE succeeds

test=> DELETE FROM foreigntest;
DELETE 1

-- By deleting the foreign key first, the DELETE succeeds

-- By deleting the foreign key first, the DELETE succeeds

test=> DELETE FROM primarytest;
DELETE 1

-- By deleting the foreign key first, the DELETE succeeds

-- By deleting the foreign key first, the DELETE succeeds

--- By deleting the foreign key first, the DELETE succeeds

Figure 14.11: Foreign key actions
Figure 14.12: Example of a multicolurnm 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 (col3 INTEGER, col4 INTEGER,
          FOREIGN KEY (col3, col4) REFERENCES primarytest2 MATCH FULL);

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

Figure 14.14: DEFERRABLE foreign key constraint
```sql
CREATE TABLE friend2 (  
    firstname CHAR(15),  
    lastname CHAR(20),  
    city CHAR(15),  
    state CHAR(2) CHECK (length(trim(state)) = 2),  
    age INTEGER CHECK (age >= 0),  
    gender CHAR(1) CHECK (gender IN ('M', 'F')),  
    last_met DATE CHECK (last_met BETWEEN '1950-01-01' AND CURRENT_DATE),  
    CHECK (upper(trim(firstname)) != 'ED' OR  
    upper(trim(lastname)) != 'RIVERS')
);
CREATE
```

```sql
INSERT INTO friend2  
```

ERROR: ExecAppend: rejected due to CHECK constraint friend2_last_met

Figure 14.15: CHECK constraints
Chapter 15

Importing and Exporting Data

<table>
<thead>
<tr>
<th>Backslash String</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\TAB</td>
<td>tab if using default delimiter tab</td>
</tr>
<tr>
<td>|</td>
<td>pipe if using pipe as the delimiter</td>
</tr>
<tr>
<td>\N</td>
<td>NULL if using the default NULL output</td>
</tr>
<tr>
<td>\b</td>
<td>backspace</td>
</tr>
<tr>
<td>\f</td>
<td>form feed</td>
</tr>
<tr>
<td>\n</td>
<td>newline</td>
</tr>
<tr>
<td>\r</td>
<td>carriage return</td>
</tr>
<tr>
<td>\t</td>
<td>tab</td>
</tr>
<tr>
<td>\v</td>
<td>vertical tab</td>
</tr>
<tr>
<td>###</td>
<td>character represented by octal number ###</td>
</tr>
<tr>
<td>\</td>
<td>backslash</td>
</tr>
</tbody>
</table>

Table 15.1: Backslashes understood by COPY
CREATE TABLE copytest (  intcol INTEGER,  numcol NUMERIC(16,2),  textcol TEXT,  boolcol BOOLEAN);

CREATE

INSERT INTO copytest  VALUES (1, 23.99, 'fresh spring water', 't');
INSERT 1746561

INSERT INTO copytest  VALUES (2, 55.23, 'bottled soda', 't');
INSERT 1746571

SELECT * FROM copytest;

<table>
<thead>
<tr>
<th>intcol</th>
<th>numcol</th>
<th>textcol</th>
<th>boolcol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23.99</td>
<td>fresh spring water</td>
<td>t</td>
</tr>
<tr>
<td>2</td>
<td>55.23</td>
<td>bottled soda</td>
<td>t</td>
</tr>
</tbody>
</table>

COPY copytest TO '/tmp/copytest.out';
COPY

DELETE FROM copytest;
DELETE 2

COPY copytest FROM '/tmp/copytest.out';
COPY

SELECT * FROM copytest;

<table>
<thead>
<tr>
<th>intcol</th>
<th>numcol</th>
<th>textcol</th>
<th>boolcol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23.99</td>
<td>fresh spring water</td>
<td>t</td>
</tr>
<tr>
<td>2</td>
<td>55.23</td>
<td>bottled soda</td>
<td>t</td>
</tr>
</tbody>
</table>

(2 rows)

Figure 15.1: Example of COPY...TO and COPY...FROM
Figure 15.2: Example of COPY...FROM

Figure 15.3: Example of COPY...TO...USING DELIMITERS

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=> **COPY** copytest **TO** stdout;
```
```
1  23.99  fresh spring water  t
2  55.23  bottled soda        t
3  77.43  coffee f
```

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

```
test=> DELETE FROM copytest;
DELETE 3
```
```
test=> INSERT INTO copytest
  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

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

Table 16.1: psql’s query buffer commands
<table>
<thead>
<tr>
<th>Operation</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect to another database</td>
<td><code>\connect dbname</code></td>
</tr>
<tr>
<td>Copy table file to/from database</td>
<td><code>\copy tablename to/from filename</code></td>
</tr>
<tr>
<td>Set a variable</td>
<td><code>\set variable</code> or <code>\set variable value</code></td>
</tr>
<tr>
<td>Unset a variable</td>
<td><code>\unset variable</code></td>
</tr>
<tr>
<td>Set output format</td>
<td><code>\pset option</code> or <code>\pset option value</code></td>
</tr>
<tr>
<td>Echo</td>
<td><code>\echo string</code> or <code>\echo </code>command``</td>
</tr>
<tr>
<td>Echo to \o output</td>
<td><code>\qecho string</code> or <code>\qecho </code>command``</td>
</tr>
<tr>
<td>Copyright</td>
<td><code>\copyright</code></td>
</tr>
<tr>
<td>Change character encoding</td>
<td><code>\encoding newencoding</code></td>
</tr>
</tbody>
</table>

Table 16.2: psql’s general commands

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

Table 16.3: psql’s `\pset` options
test=> SELECT NULL;
?column?
----------
(1 row)

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

null

Figure 16.1: Example of \pset

<table>
<thead>
<tr>
<th>Modifies</th>
<th>Command</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field alignment</td>
<td>\a</td>
<td></td>
</tr>
<tr>
<td>Field separator</td>
<td>\f</td>
<td>separator</td>
</tr>
<tr>
<td>One field per line</td>
<td>\x</td>
<td></td>
</tr>
<tr>
<td>Rows only</td>
<td>\t</td>
<td></td>
</tr>
<tr>
<td>Table title</td>
<td>\C</td>
<td>title</td>
</tr>
<tr>
<td>Enable HTML</td>
<td>\H</td>
<td></td>
</tr>
<tr>
<td>HTML table tags</td>
<td>\T</td>
<td>tags</td>
</tr>
</tbody>
</table>

Table 16.4: psql’s output format shortcuts
Figure 16.2: psql variables
<table>
<thead>
<tr>
<th>Meaning</th>
<th>Variable Name</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>DBNAME</td>
<td></td>
</tr>
<tr>
<td>Multibyte encoding</td>
<td>ENCODING</td>
<td></td>
</tr>
<tr>
<td>Host</td>
<td>HOST</td>
<td></td>
</tr>
<tr>
<td>Previously assigned OID</td>
<td>LASTOID</td>
<td></td>
</tr>
<tr>
<td>Port</td>
<td>PORT</td>
<td></td>
</tr>
<tr>
<td>User</td>
<td>USER</td>
<td></td>
</tr>
<tr>
<td>Echo queries</td>
<td>ECHO</td>
<td>all</td>
</tr>
<tr>
<td>Echo \d* queries</td>
<td>ECHO_HIDDEN</td>
<td>noexec</td>
</tr>
<tr>
<td>History control</td>
<td>HISTCONTROL</td>
<td>ignorespace, ignoredups, or ignoreboth</td>
</tr>
<tr>
<td>History size</td>
<td>HISTSIZE</td>
<td>command_count</td>
</tr>
<tr>
<td>Terminate on end of file</td>
<td>IGNOREEOF</td>
<td>eof_count</td>
</tr>
<tr>
<td>\object transactions</td>
<td>LO_TRANSACTION</td>
<td>rollback, commit, nothing</td>
</tr>
<tr>
<td>Stop on query errors</td>
<td>ON_ERROR_STOP</td>
<td></td>
</tr>
<tr>
<td>Command prompt</td>
<td>PROMPT1, PROMPT2, PROMPT3</td>
<td>string</td>
</tr>
<tr>
<td>Suppress output</td>
<td>QUIET</td>
<td></td>
</tr>
<tr>
<td>Single-line mode</td>
<td>SINGLELINE</td>
<td></td>
</tr>
<tr>
<td>Single-step mode</td>
<td>SINGLESTEP</td>
<td></td>
</tr>
</tbody>
</table>

Table 16.5: psql’s predefined variables
<table>
<thead>
<tr>
<th>Listing</th>
<th>Command</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table, index, view, or sequence</td>
<td>\d</td>
<td>name</td>
</tr>
<tr>
<td>Tables</td>
<td>\dt</td>
<td>name</td>
</tr>
<tr>
<td>Indexes</td>
<td>\di</td>
<td>name</td>
</tr>
<tr>
<td>Sequences</td>
<td>\ds</td>
<td>name</td>
</tr>
<tr>
<td>Views</td>
<td>\dv</td>
<td>name</td>
</tr>
<tr>
<td>Permissions</td>
<td>\z or \dp</td>
<td>name</td>
</tr>
<tr>
<td>System tables</td>
<td>\dS</td>
<td>name</td>
</tr>
<tr>
<td>Large objects</td>
<td>\dl</td>
<td>name</td>
</tr>
<tr>
<td>Types</td>
<td>\dT</td>
<td>name</td>
</tr>
<tr>
<td>Functions</td>
<td>\df</td>
<td>name</td>
</tr>
<tr>
<td>Operators</td>
<td>\do</td>
<td>name</td>
</tr>
<tr>
<td>Aggregates</td>
<td>\da</td>
<td>name</td>
</tr>
<tr>
<td>Comments</td>
<td>\dd</td>
<td>name</td>
</tr>
<tr>
<td>Databases</td>
<td>\l</td>
<td></td>
</tr>
</tbody>
</table>

Table 16.6: psql’s listing commands

<table>
<thead>
<tr>
<th>Large Objects</th>
<th>Command</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Import</td>
<td>\lo_import</td>
<td>file</td>
</tr>
<tr>
<td>Export</td>
<td>\lo_export</td>
<td>oid file</td>
</tr>
<tr>
<td>Unlink</td>
<td>\lo_unlink</td>
<td>oid</td>
</tr>
<tr>
<td>List</td>
<td>\lo_list</td>
<td></td>
</tr>
</tbody>
</table>

Table 16.7: psql’s large object commands
<table>
<thead>
<tr>
<th>Option</th>
<th>Capability</th>
<th>Argument</th>
<th>Additional Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Connection</strong></td>
<td>Database (optional)</td>
<td>-d</td>
<td>database</td>
</tr>
<tr>
<td></td>
<td>Host name</td>
<td>-h</td>
<td>hostname</td>
</tr>
<tr>
<td></td>
<td>Port</td>
<td>-p</td>
<td>port</td>
</tr>
<tr>
<td></td>
<td>User</td>
<td>-U</td>
<td>user</td>
</tr>
<tr>
<td></td>
<td>Force password prompt</td>
<td>-W</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Version</td>
<td>-V</td>
<td></td>
</tr>
<tr>
<td><strong>Controlling Output</strong></td>
<td>Field alignment</td>
<td>-A</td>
<td>separator</td>
</tr>
<tr>
<td></td>
<td>Field separator</td>
<td>-F</td>
<td>separator</td>
</tr>
<tr>
<td></td>
<td>Record separator</td>
<td>-R</td>
<td>separator</td>
</tr>
<tr>
<td></td>
<td>Rows only</td>
<td>-t</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Extended output format</td>
<td>-x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Echo \d* queries</td>
<td>-E</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Quiet mode</td>
<td>-q</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HTML output</td>
<td>-H</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HTML table tags</td>
<td>-T</td>
<td>tags</td>
</tr>
<tr>
<td></td>
<td>Set \pset options</td>
<td>-P</td>
<td>option or option=value</td>
</tr>
<tr>
<td></td>
<td>List databases</td>
<td>-l</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disable readline</td>
<td>-n</td>
<td></td>
</tr>
<tr>
<td><strong>Automation</strong></td>
<td>Echo all queries from scripts</td>
<td>-a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Echo queries</td>
<td>-e</td>
<td>query</td>
</tr>
<tr>
<td></td>
<td>Execute query</td>
<td>-c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Get queries from file</td>
<td>-f</td>
<td>file</td>
</tr>
<tr>
<td></td>
<td>Output to file</td>
<td>-o</td>
<td>file</td>
</tr>
<tr>
<td></td>
<td>Single-step mode</td>
<td>-s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Single-line mode</td>
<td>-S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Suppress reading ~/.psqlrc</td>
<td>-X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set variable</td>
<td>-v</td>
<td>var or var=value</td>
</tr>
</tbody>
</table>

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

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

Table 17.1: Interface summary

Enter a state code: AL
Alabama

Figure 17.1: Sample application being run
**Figure 17.2: Statename table**

```
CREATE TABLE statename (code CHAR(2) PRIMARY KEY,
                        name CHAR(30)
);

INSERT INTO statename VALUES ('AL', 'Alabama');
INSERT 18934 1
INSERT INTO statename VALUES ('AK', 'Alaska');
INSERT 18934 1
```

**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 \n        FROM statename \n        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,
            "SELECT name \n FROM statename \n WHERE code = '%s', state_code;", state_code);

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

    while (fetch(state_name) != END_OF_TUPLES)
    {
        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() {
  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 */
  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 \n            FROM statename \n            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 \n    FROM statename \n   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
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("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.
* + 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 = $conn->exec( # send the query
"SELECT name \n FROM statename \n WHERE code = '$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

set res [pg_exec $conn \\
  "SELECT name \\
    FROM statename \\
    WHERE code = '${state_code}'"] ;# send the query

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("SELECT name \nFROM statename \nWHERE code = '%s'" % state_code).getresult():
    sys.stdout.write('%s
' % name)  # print the value returned

Figure 17.11: Python sample program
Figure 17.12: PHP sample program—input
$database = pg_Connect("", ",", ",", "test"); # connect to the database
if (!$database) # did the database connection fail?
{
    echo "Connection to database failed.
    exit;
}
$result = pg_Exec($database, # send the query
    "SELECT name \n    FROM statename \n    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>
}

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)
    RETURNS numeric
    AS 'SELECT ($1 * 0.06::numeric(8,2))::numeric(8,2);'
    LANGUAGE 'sql';

CREATE

SELECT tax(100);

<table>
<thead>
<tr>
<th>tax</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.00</td>
</tr>
</tbody>
</table>

(1 row)

Figure 18.2: SQL tax function
```sql
CREATE TABLE part(
    part_id INTEGER,
    name CHAR(30),
    cost NUMERIC(8,2),
    weight FLOAT
);

INSERT INTO part VALUES (637, 'cable', 14.29, 5);
INSERT INTO part VALUES (638, 'sticker', 0.84, 1);
INSERT INTO part VALUES (639, 'bulb', 3.68, 3);

SELECT part_id, name, cost, tax(cost),
       cost + tax(cost) AS total
FROM part
ORDER BY part_id;
```

<table>
<thead>
<tr>
<th>part_id</th>
<th>name</th>
<th>cost</th>
<th>tax</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>637</td>
<td>cable</td>
<td>14.29</td>
<td>0.86</td>
<td>15.15</td>
</tr>
<tr>
<td>638</td>
<td>sticker</td>
<td>0.84</td>
<td>0.05</td>
<td>0.89</td>
</tr>
<tr>
<td>639</td>
<td>bulb</td>
<td>3.68</td>
<td>0.22</td>
<td>3.90</td>
</tr>
</tbody>
</table>

(3 rows)

Figure 18.3: Recreation of the `part` table
CREATE FUNCTION shipping(numeric)
RETURNS numeric
AS 'SELECT CASE
WHEN $1 < 2 THEN CAST(3.00 AS numeric(8,2))
WHEN $1 >= 2 AND $1 < 4 THEN CAST(5.00 AS numeric(8,2))
WHEN $1 >= 4 THEN CAST(6.00 AS numeric(8,2))
END;'
LANGUAGE 'sql';

SELECT part_id,
     trim(name) AS name,
     cost,
     tax(cost),
     cost + tax(cost) AS subtotal,
     shipping(weight),
     cost + tax(cost) + shipping(weight) AS total
FROM part
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
CREATE FUNCTION getstatename(text)
RETURNS text
AS 'SELECT CAST(name AS TEXT)
FROM statename
WHERE code = $1,'
LANGUAGE 'sql';
CREATE
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;

<table>
<thead>
<tr>
<th>name</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fleer Gearworks, Inc.</td>
<td>Alabama</td>
</tr>
<tr>
<td>Mark Middleton</td>
<td>Indiana</td>
</tr>
<tr>
<td>Mike Nichols</td>
<td>Florida</td>
</tr>
</tbody>
</table>
(3 rows)

Figure 18.6: Getting state name using a join and a function
CREATE FUNCTION getstatename2(text)
  RETURNS text
  AS 'DECLARE ret TEXT;
  BEGIN
    SELECT INTO ret CAST(name AS TEXT)
    FROM statename
    WHERE code = $1;
    RETURN ret;
  END;'
  LANGUAGE 'plpgsql';

Figure 18.7: PL/pgSQL version of getstatename
test=> CREATE FUNCTION spread(text)
  RETURNS text
  AS 'DECLARE
    str text;
    ret text;
    i integer;
    len integer;
    BEGIN
      str := upper($1);
      ret := ' '; -- start with zero length
      i := 1;
      len := length(str);
      WHILE i <= len LOOP
        ret := ret || substr(str, i, 1) || ' '; 
        i := i + 1;
      END LOOP;
      RETURN ret;
    END;'
  LANGUAGE 'plpgsql';
CREATE
   SELECT spread('Major Financial Report');

----------------------------------------------
MAJOR FINANCIAL REPORT
(1 row)

Figure 18.8: PL/PgSQL spread function
CREATE FUNCTION getstatecode(text)
RETURNS text AS 
'DECLARE
state_str state.name%TYPE;
statename_rec record;
i integer;
len integer;
matches record;
search_str text;
BEGIN
state_str := initcap($1); -- capitalization match column
len := length(trim($1));
i := 2;
SELECT INTO statename_rec * -- first try for an exact match
FROM statename
WHERE name = state_str;
IF FOUND
THEN RETURN statename_rec.code;
END IF;
WHILE i <= len LOOP -- test 2,4,6,... chars for match
search_str = trim(substr(state_str,1,i)) || '%%%';
SELECT INTO matches COUNT(*)
FROM statename
WHERE name LIKE search_str;
IF matches.count = 0 -- no matches, failure
THEN RETURN NULL;
END IF;
IF matches.count = 1 -- exactly one match, return it
THEN
SELECT INTO statename_rec *
FROM statename
WHERE name LIKE search_str;
IF FOUND
THEN RETURN statename_rec.code;
END IF;
END IF;
i := i + 2; -- >1 match, try 2 more chars
END LOOP;
RETURN ''; 
END;
' LANGUAGE 'plpgsql';
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
CREATE FUNCTION change_statename(char(2), char(30))
RETURNS boolean
AS 'DECLARE
    state_code ALIAS FOR $1;
    state_name ALIAS FOR $2;
    statename_rec RECORD;
    BEGIN
        IF length(state_code) = 0 -- no state code, failure
        THEN RETURN ''f'';
        ELSE
            IF length(state_name) != 0 -- is INSERT or UPDATE?
            THEN
                SELECT INTO statename_rec *
                FROM statename
                WHERE code = state_code;
                IF NOT FOUND -- is state not in table?
                THEN INSERT INTO statename
                VALUES (state_code, state_name);
                ELSE UPDATE statename
                SET name = state_name
                WHERE code = state_code;
                END IF;
                RETURN ''t'';
            ELSE -- is DELETE
                SELECT INTO statename_rec *
                FROM statename
                WHERE code = state_code;
                IF FOUND
                THEN DELETE FROM statename
                WHERE code = state_code;
                RETURN ''t'';
                ELSE RETURN ''f'';
                END IF;
            END IF;
        END IF;
    END;' 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()
CREATE FUNCTION trigger_insert_update_statename()
RETURNS opaque
AS 'BEGIN
    IF new.code !~ '[A-Za-z][A-Za-z]$
    THEN RAISE EXCEPTION ''State code must be two alphabetic characters.''';
    END IF;
    IF new.name !~ '[A-Za-z]*$
    THEN RAISE EXCEPTION ''State name must be only alphabetic characters.''';
    END IF;
    IF length(trim(new.name)) < 3
    THEN RAISE EXCEPTION ''State name must longer than two characters.''';
    END IF;
    new.code = upper(new.code); -- uppercase statename.code
    new.name = initcap(new.name); -- capitalize statename.name
    RETURN new;
END;' LANGUAGE 'plpgsql';

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

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 ctos(20);
ctos
------
  68
(1 row)

Figure 19.3: Calling function ctos
# Chapter 20

## Administration

<table>
<thead>
<tr>
<th>Table</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_aggregate</td>
<td>aggregates</td>
</tr>
<tr>
<td>pg_attribute</td>
<td>columns</td>
</tr>
<tr>
<td>pg_class</td>
<td>tables</td>
</tr>
<tr>
<td>pg_database</td>
<td>databases</td>
</tr>
<tr>
<td>pg_description</td>
<td>comments</td>
</tr>
<tr>
<td>pg_group</td>
<td>groups</td>
</tr>
<tr>
<td>pg_index</td>
<td>indexes</td>
</tr>
<tr>
<td>pg_log</td>
<td>transaction status</td>
</tr>
<tr>
<td>pg_operator</td>
<td>operators</td>
</tr>
<tr>
<td>pg_proc</td>
<td>functions</td>
</tr>
<tr>
<td>pg_rewrite</td>
<td>rules and views</td>
</tr>
<tr>
<td>pg_shadow</td>
<td>users</td>
</tr>
<tr>
<td>pg_trigger</td>
<td>triggers</td>
</tr>
<tr>
<td>pg_type</td>
<td>types</td>
</tr>
</tbody>
</table>

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