PostgreSQL: Introduction and Concepts

Bruce Momjian

April 3, 2000
Note to Reviewers

The material on these pages is a work in progress, tentatively titled, *PostgreSQL: Introduction and Concepts*, to be published in 2000, ©Addison–Wesley. Posted with permission of the publisher. All rights reserved.

I have completed my first draft through chapter 10. Later chapters may not be spell checked, proofread, or critiqued. You are seeing it as it is being written.

I am interested in any comments you may have, including typographic errors, places with not enough detail or too much detail, missing topics, extraneous topics, confusing sentences, poor word choice, etc. The PDF version has numbers appearing in the margins to allow you to easily refer to specific lines in the book. People reading the web version may refer to specific URLs. Please mention the date of April 3, 2000 when referring to this document. You may contact me at mailto:pgman@candle.pha.pa.us.

A current copy may be retrieved from http://www.postgresql.org/docs/awbook.html. Also, it is available from the PostgreSQL FAQ’s and Documentation page, http://www.postgresql.org/docs. It is updated automatically every night. This book is set in Bitstream Century Old Style, 11 point.

Keep in mind that this is to be printed as a book. In the PDF version, diagrams may not appear on the same pages that refer to them. They will appear on the facing page when printed in book format.
Foreword

Most research projects never leave the academic environment. Occasionally, exceptional ones survive the transition from the university to the real world and go on to become a phenomenon. POSTGRESQL is one of those projects. Its popularity and success is a testament to the dedication and hard work of the POSTGRESQL global development team. Developing an advanced database system is no small feat. Maintaining and enhancing an inherited code base is even more challenging. The POSTGRESQL team has not only managed to improve the quality and usability of the system, but to spread its use among the Internet user community. This book is a major milestone in the history of the project.

POSTGRES95, later renamed POSTGRESQL, started out as a pet project to overhaul POSTGRES. POSTGRES is a novel and feature-rich database system created by many students and staff at the UNIVERSITY OF CALIFORNIA AT BERKELEY. Our goal was to keep the powerful and useful features while trimming down the bloat caused by much experimentation and research. We had a lot of fun reworking the internals. At the time, we had no idea where we were going with the project. The POSTGRES95 exercise was not research, but simply a bit of engineering housecleaning. By the spring of 1995, it occurred to us that there was a need for an open-source SQL-based multi-user database in the Internet user community. Our first release was met with great enthusiasm. We are very pleased to see the project continuing.

Obtaining information about a complex system like POSTGRESQL is a great barrier to its adoption. This book fills a critical gap in the documentation of the project. This book provides an excellent overview of the system. It covers a wide range of topics from the basics to the more advanced and unique features of POSTGRESQL.

In writing this book, Bruce Momjian has drawn on his experience in helping beginners with POSTGRESQL. The text is easy to understand and full of practical tips. Momjian captures database concepts using simple and easy to understand language. He also presents numerous real life examples throughout the book. He does an outstanding job and covers many advanced POSTGRESQL topics. Enjoy reading the book and have fun exploring POSTGRESQL! It is our hope this book will not only teach you about using PostgreSQL but also inspire you to delve into its innards and contribute to the ongoing POSTGRESQL development effort.

JOLLY CHEN and ANDREW YU, co-authors of POSTGRES95
Preface

This book is about POSTGRESQL, the most advanced open source database. From its origins in academia, POSTGRESQL has moved to the Internet with explosive growth. It is hard to believe the advances during the past three years under the guidance of a team of world-wide Internet developers. This book is a testament to their vision, and to the success POSTGRESQL has become.

The book is designed to lead the reader from their first database query through the complex queries needed to solve real-world problems. No knowledge of database theory or practice is required. Basic knowledge of operating system capabilities is expected, like the ability to type at an operating system prompt.

The book starts with a short history of POSTGRESQL. It leads the reader through their first query, and teaches the most important database commands. Common problems are covered early, like placing quotes inside quoted strings. This should prevent users from getting stuck with queries that fail. I have seen many bug reports in the past few years, and try to cover the common pitfalls.

With a firm foundation established, additional commands are introduced. Finally, specialty chapters outline complex topics like multi-user control and performance. While coverage of these complex topics is not exhaustive, I try to show common real-world problems and their solutions.

I do not shy away from complex topics, like index types and table join methods. If these topics are confusing, it is sufficient to skim these chapters and continue. Start using the database, and return to these topics later.

At each step, the purpose of the commands is clearly illustrated. I want readers to understand more than query syntax. I want them to know why each command is valuable, so they will use the proper commands in their real-world database applications.

A novice should read the entire book, while skimming over the later chapters. The complex nature of database systems should not prevent readers from getting started. Test databases are safe way to try queries. As readers gain more experience, the later chapters will start to make sense. Experienced database users can skip the chapters on basic SQL functionality. The cross-referencing of sections should allow you to quickly move from general to more specific information.

Appendix D contains a copy of the POSTGRESQL reference manual which should be consulted anytime you are having trouble with query syntax. Also, I should mention the excellent documentation that is part of POSTGRESQL. The documentation covers many complex topics. It includes much POSTGRESQL-specific functionality that cannot be covered in a book of this length. I refer to sections of the documentation in this text where appropriate.

The website for this book is located at http://www.postgresql.org/docs/awbook.html.
Acknowledgements

Update this page with current information before publication.

POSTGRESQL and this book would not be possible without the talented and hard-working members of the POSTGRESQL Global Development Team. They took source code that could have become just another abandoned project, and turned it into the open source alternative to commercial database systems. POSTGRESQL is a shining example of Internet community development.

Steering

• Fournier, Marc G. in Wolfville, Nova Scotia, Canada (scrappy@hub.org) coordinates the whole effort and provides the server and administers our primary web site, mailing lists, ftp site, and source code repository.

• Lane, Tom in Pittsburgh, PA, USA (tgl@sss.pgh.pa.us) has performed many PostgreSQL improvements. He has worked on the optimizer and a variety of complex issues.

• Lockhart, Thomas G. in Pasadena, California, USA (lockhart@alumni.caltech.edu) works on documentation, data types, particularly date/time and geometric objects, and on SQL standards compatibility.

• Mikheev, Vadim B. in Krasnoyarsk, Russia (vadim@krs.ru) does large projects, like vacuum, subselects, triggers, and multi-version concurrency control (MVCC).

• Momjian, Bruce in Philadelphia, Pennsylvania, USA (maillist@candle.pha.pa.us) maintains FAQ and TODO lists, code cleanup, some patch application, makes training materials, and some coding.

• Wieck, Jan in Hamburg, Germany (wieck@debis.com) overhauled the query rewrite rule system, wrote our procedural languages PL/pgSQL and PL/Tcl and added the NUMERIC/DECIMAL type.

Major Developers

• Cain, D'Arcy J.M. in Toronto, Ontario, Canada (darcy@druid.net) worked on the Tcl interface, PyGreSQL, and the INET type.

• Dal Zotto, Massimo near Trento, Italy (dz@cs.unitn.it) has done locking code and other improvements.

• Elphick, Oliver in Newport, Isle of Wight, UK (olly@lfix.co.uk) maintains the PostgreSQL package for Debian GNU/Linux.

• Horak, Daniel near Pilzen, Czech Republic (dan.horak@email.cz) did the WinNT port of PostgreSQL (using the Cygwin environment).

• Inoue, Hiroshi in Fukui, Japan (Inoue@tpf.co.jp) improved btree index access.
ACKNOWLEDGEMENTS

• Ishii, Tatsuo in Zushi, Kanagawa, Japan (t-ishii@sra.co.jp) handles multi-byte foreign language support and porting issues.

• Martin, Dr. Andrew C.R. in London, England (martin@biochem.ucl.ac.uk) helped in the Linux and Irix FAQ’s including some patches to the PostgreSQL code.

• Mergl, Edmund in Stuttgart, Germany (E.Mergl@bawue.de) created and maintains pgsql_perl5. He also created DBD-Pg which is available via CPAN.

• Meskes, Michael in Dusseldorf, Germany (meskes@postgresql.org) handles multi-byte foreign language support, and maintains ecpg.

• Mount, Peter in Maidstone, Kent, United Kingdom (peter@retep.org.uk) has done the Java JDBC Interface.

• Nikolaidis, Byron in Baltimore, Maryland (byron.nikolaidis@home.com) rewrote and maintains the ODBC interface for Windows.

• Owen, Lamar in Pisgah Forest, North Carolina, USA (lamar.owen@wgcr.org) RPM package maintainer.

• Teodorescu, Constantin in Braila, Romania (teo@flex.ro) has done the PgAccess DB Interface.

• Thyni, Göran in Kiruna, Sweden (goran@kyla.kiruna.se) has worked on the UNIX socket code.

Non-code contributors

• Bartunov, Oleg in Moscow, Russia (oleg@sai.msu.su) introduced the locale support.

• Vielhaber, Vince near Detroit, Michigan, USA (vev@michvhf.com) maintains our website.

All developers listed in alphabetical order.
# Contents

Note to Reviewers ................................................................. iii
Foreword .............................................................................. v
Preface ................................................................................ vii
Acknowledgements ................................................................ ix

## 1 History of PostgreSQL
1.1 Introduction ........................................................................ 1
1.2 UNIVERSITY OF CALIFORNIA AT BERKELEY ...................... 1
1.3 Development Leaves BERKELEY ........................................ 1
1.4 PostgreSQL Global Development Team .............................. 2
1.5 Open Source Software ....................................................... 3

## 2 Issuing Database Commands
2.1 Starting a Database Session ................................................ 5
2.2 Controlling a Session ........................................................ 6
2.3 Getting Help .................................................................... 7
2.4 Exiting a Session ............................................................... 8

## 3 Basic SQL Commands
3.1 Relational Databases .......................................................... 9
3.2 Creating Tables ................................................................. 10
3.3 Adding Data with INSERT ................................................ 11
3.4 Viewing Data with SELECT .............................................. 12
3.5 Selecting Specific Rows with WHERE ............................... 13
3.6 Removing Data with DELETE ........................................... 13
3.7 Modifying Data with UPDATE ......................................... 14
3.8 Sorting Data with ORDER BY ........................................ 14
3.9 Destroying Tables ............................................................. 17
3.10 Summary ....................................................................... 17

## 4 Customizing Queries
4.1 Data types ..................................................................... 19
4.2 Quotes Inside Text ........................................................... 19
4.3 Using NULL Values .......................................................... 21
4.4 Controlling DEFAULT Values ......................................... 23
4.5 Column Labels ................................................................. 24
## CONTENTS

4.6 Comments ................................................. 24 0727
4.7 AND/OR Usage ............................................ 25 0728
4.8 Range of Values ......................................... 27 0729
4.9 LIKE Comparison ......................................... 27 0730
4.10 Regular Expressions ...................................... 29 0731
4.11 CASE Clause ............................................... 32 0732
4.12 Distinct Rows ............................................ 32 0733
4.13 Functions and Operators ................................. 33 0734
4.14 SET, SHOW, and RESET ................................. 33 0735

5 SQL Aggregates .............................................. 39 0736
5.1 Aggregates ................................................. 39 0737
5.2 Using GROUP BY ........................................... 42 0738
5.3 Using HAVING ............................................ 42 0739
5.4 Query Tips ............................................... 43 0740

6 Joining Tables .............................................. 45 0741
6.1 Table and Column References ............................ 45 0742
6.2 Joined Tables ............................................. 45 0743
6.3 Creating Joined Tables .................................... 47 0744
6.4 Performing Joins ......................................... 50 0745
6.5 Three and Four Table Joins ............................... 51 0746
6.6 Additional Join Possibilities ............................. 52 0747
6.7 Choosing a Join Key ....................................... 55 0748
6.8 One-to-Many Joins ....................................... 56 0749
6.9 Unjoined tables ............................................ 56 0750
6.10 Table Aliases and Self-Joins ......................... 56 0751
6.11 Non-Equijoins ........................................... 58 0752
6.12 Ordering Multiple Parts ................................ 58 0753
6.13 Primary and Foreign Keys ............................... 60 0754
6.14 Summary ............................................... 60 0755

7 Numbering Rows ........................................... 63 0756
7.1 Object Identification Numbers (OIDs) ................ 63 0757
7.2 Object Identification Number Limitations .......... 64 0758
7.3 Sequences ............................................... 65 0759
7.4 Creating Sequences ..................................... 65 0760
7.5 Using Sequences to Number Rows .................... 67 0761
7.6 Serial Column Type ...................................... 67 0762
7.7 Manually Numbering Rows ............................... 69 0763
7.8 Summary ............................................... 69 0764

8 Combining SELECTS ....................................... 71 0765
8.1 UNION, EXCEPT, INTERSECT Clauses ................. 71 0766
8.2 Subqueries ............................................... 74 0767
8.3 Outer Joins ............................................... 81 0768
8.4 Subqueries in Non-SELECT Queries .................. 81 0769
8.5 UPDATE with FROM ..................................... 83 0770
8.6 Inserting Data Using SELECT .......................... 83 0771
CONTENTS

15 Extending POSTGRESQL 115
   15.1 User-Defined Functions ........................................ 115
   15.2 User-Defined Operators ........................................ 115
   15.3 User-Defined Types ............................................ 115

16 Programming Interfaces 117
   16.1 Need for a Programming Language ............................ 118
   16.2 C Language API (LIBPQ) ..................................... 118
   16.3 Embedded C (ECPG) ............................................. 118
   16.4 C++ (LIBPQ++) ................................................ 118
   16.5 JAVA (JDBC) .................................................. 118
   16.6 ODBC .......................................................... 118
   16.7 PERL (PGSQL_PERL5) ....................................... 118
   16.8 TCL/TK (LIBPGTCL) ......................................... 118
   16.9 PYTHON (PYGREEN) .......................................... 118
   16.10Web Access .................................................... 118
   16.11Server-side Programming ..................................... 118

17 User Applications 119
   17.1 PSQL .......................................................... 119
   17.2 PGACCESS ..................................................... 122

18 POSTGRESQL Administration 123
   18.1 Creating Users and Databases ................................. 123
   18.2 Backup and Restore .......................................... 123
   18.3 Performance .................................................. 123
   18.4 Troubleshooting ............................................... 123
   18.5 Customization ................................................ 123
   18.6 System Tables ................................................ 123
   18.7 Access Configuration ........................................ 123
   18.8 Internationalization ......................................... 123

A Additional Resources 125
   A.1 Frequently Asked Questions (FAQ'S) ......................... 125
   A.2 Mailing List Support ......................................... 125
   A.3 Supplied Documentation ..................................... 125
   A.4 Commercial Support ......................................... 125
   A.5 Modifying the Source Code .................................. 125

B Installation 127
   B.1 Getting POSTGRESQL ........................................ 127
   B.2 Compiling ...................................................... 127
   B.3 Initialization ................................................ 127
   B.4 Starting the Server .......................................... 127
   B.5 Creating a Database ......................................... 127

C PostgreSQL Non-Standard Features by Chapter 129

D Reference Manual 131
CONTENTS

Bibliography 133

Index 133
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>psql session startup</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>My first SQL query</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>Multi-line query</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>Backslash-p demo</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Databases</td>
<td>9</td>
</tr>
<tr>
<td>3.2</td>
<td>Create table</td>
<td>10</td>
</tr>
<tr>
<td>3.3</td>
<td>Example of backslash-d</td>
<td>11</td>
</tr>
<tr>
<td>3.4</td>
<td>Insert</td>
<td>12</td>
</tr>
<tr>
<td>3.5</td>
<td>My first SELECT</td>
<td>12</td>
</tr>
<tr>
<td>3.6</td>
<td>My first WHERE</td>
<td>13</td>
</tr>
<tr>
<td>3.7</td>
<td>More complex WHERE clause</td>
<td>13</td>
</tr>
<tr>
<td>3.8</td>
<td>A single cell</td>
<td>13</td>
</tr>
<tr>
<td>3.9</td>
<td>A block of cells</td>
<td>14</td>
</tr>
<tr>
<td>3.10</td>
<td>Comparing string fields</td>
<td>14</td>
</tr>
<tr>
<td>3.11</td>
<td>DELETE example</td>
<td>15</td>
</tr>
<tr>
<td>3.12</td>
<td>My first UPDATE</td>
<td>15</td>
</tr>
<tr>
<td>3.13</td>
<td>Use of ORDER BY</td>
<td>16</td>
</tr>
<tr>
<td>3.14</td>
<td>Reverse ORDER BY</td>
<td>16</td>
</tr>
<tr>
<td>3.15</td>
<td>Use of ORDER BY and WHERE</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of common data types</td>
<td>20</td>
</tr>
<tr>
<td>4.2</td>
<td>Insertion of specific columns</td>
<td>21</td>
</tr>
<tr>
<td>4.3</td>
<td>NULL handling</td>
<td>22</td>
</tr>
<tr>
<td>4.4</td>
<td>Comparison of NULL fields</td>
<td>22</td>
</tr>
<tr>
<td>4.5</td>
<td>NULLS and blank strings</td>
<td>23</td>
</tr>
<tr>
<td>4.6</td>
<td>Using DEFAULTs</td>
<td>24</td>
</tr>
<tr>
<td>4.7</td>
<td>Controlling column labels</td>
<td>24</td>
</tr>
<tr>
<td>4.8</td>
<td>Computation using a column label</td>
<td>25</td>
</tr>
<tr>
<td>4.9</td>
<td>Comment styles</td>
<td>25</td>
</tr>
<tr>
<td>4.10</td>
<td>New friends</td>
<td>25</td>
</tr>
<tr>
<td>4.11</td>
<td>WHERE test for Sandy Gleason</td>
<td>26</td>
</tr>
<tr>
<td>4.12</td>
<td>Friends in New Jersey and Pennsylvania</td>
<td>26</td>
</tr>
<tr>
<td>4.13</td>
<td>Mixing ANDs and ORs</td>
<td>26</td>
</tr>
<tr>
<td>4.14</td>
<td>Properly mixing ANDs and ORs</td>
<td>27</td>
</tr>
<tr>
<td>4.15</td>
<td>Selecting a range of values</td>
<td>28</td>
</tr>
<tr>
<td>4.16</td>
<td>Firstname begins with D.</td>
<td>28</td>
</tr>
<tr>
<td>4.17</td>
<td>Regular expression sample queries</td>
<td>30</td>
</tr>
<tr>
<td>4.18</td>
<td>Complex regular expression queries</td>
<td>31</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

4.19 CASE example .............................................. 32
4.20 Complex CASE example .................................. 33
4.21 DISTINCT prevents duplicates .......................... 34
4.22 Function examples ........................................ 35
4.23 Operator examples ........................................ 36
4.24 SHOW and RESET examples .............................. 37

5.1 Aggregate examples ....................................... 40
5.2 Aggregates and NULLS ..................................... 41
5.3 Aggregate with GROUP BY ................................. 42
5.4 GROUP BY on two columns ............................... 43
5.5 HAVING usage .............................................. 43

6.1 Qualified column names .................................. 46
6.2 Joining tables .............................................. 46
6.3 Creation of company tables .............................. 48
6.4 Insertion into company tables ........................... 49
6.5 Finding customer name using two queries .......... 50
6.6 Finding customer name using one query ................ 50
6.7 Finding order number for customer name ............. 51
6.8 Three-table join .......................................... 51
6.9 Four-table join ........................................... 52
6.10 Employees who have taken orders for customers. .... 53
6.11 Joining customer and employee ........................ 53
6.12 Joining part and employee ............................... 54
6.13 Statename table .......................................... 54
6.14 Using a customer code ................................... 55
6.15 One-to-many join .......................................... 57
6.16 Unjoined tables .......................................... 57
6.17 Using table aliases ....................................... 57
6.18 Examples of self-joins using table aliases .......... 58
6.19 Non-equijoins ............................................ 59
6.20 New salesorder table for multiple parts per order ... 59
6.21 Orderpart table .......................................... 60
6.22 Queries involving orderpart table .................... 61

7.1 OID test .................................................... 64
7.2 Columns with OIDs ........................................ 64
7.3 Examples of sequence function use ................... 66
7.4 Numbering customer rows using a sequence ......... 67
7.5 Customer table using SERIAL ........................... 68

8.1 Combining two columns with UNION ..................... 71
8.2 Combining two tables with UNION ...................... 72
8.3 UNION with duplicates .................................. 73
8.4 UNION ALL with duplicates ............................. 73
8.5 EXCEPT restricts output from the first SELECT ....... 73
8.6 INTERSECT returns only duplicated rows ............ 74
8.7 Friends not in Dick Gleason’s state .................... 75
LIST OF FIGURES

8.8 Subqueries can replace some joins ........................................ 76
8.9 Correlated subquery .......................................................... 77
8.10 Employees who took orders .................................................. 78
8.11 Customers who have no orders ............................................. 79
8.12 IN query rewritten using ANY and EXISTS .............................. 80
8.13 NOT IN query rewritten using ALL and EXISTS ....................... 81
8.14 Simulating outer joins ....................................................... 82
8.15 Subqueries with UPDATE and DELETE .................................... 82
8.16 UPDATE the order_date ...................................................... 83
8.17 Using SELECT with INSERT ................................................ 83
8.18 Table creation with SELECT ............................................... 84
9.1 Error generated by undefined function/type combination. ............ 91
9.2 Error generated by undefined operator/type combination ............ 92
9.3 Creation of array columns .................................................. 92
9.4 Using arrays ................................................................. 93
9.5 Using large images .......................................................... 94
10.1 INSERT with no explicit transaction ..................................... 95
10.2 INSERT with explicit transaction ........................................ 96
10.3 Two INSERTs in a single transaction .................................... 96
10.4 Multi-statement transaction ............................................... 96
10.5 Transaction rollback ....................................................... 97
10.6 Read-committed isolation level .......................................... 98
10.7 Serializable isolation level ............................................... 99
10.8 SELECT with no locking .................................................. 100
10.9 SELECT...FOR UPDATE ................................................... 101
11.1 Example of CREATE INDEX ................................................ 103
11.2 Example of a unique index ................................................. 104
11.3 Using EXPLAIN ............................................................ 105
11.4 More complex EXPLAIN examples ....................................... 106
11.5 EXPLAIN example using joins ........................................... 107
LIST OF FIGURES

1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
List of Tables

3.1 Table friend ................................................................. 10
4.1 Common data types ...................................................... 19
4.2 Comparisons ............................................................... 27
4.3 LIKE comparison ......................................................... 28
4.4 Regular expression operators .......................................... 29
4.5 Regular expression special characters ................................ 29
4.6 Regular expression examples .......................................... 31
4.7 SET options ............................................................... 34
4.8 DATESTYLE output ...................................................... 37
5.1 Aggregates ................................................................. 39
7.1 Sequence number access functions ................................... 65
9.1 POSTGRESQL data types ............................................... 86
9.2 Geometric types .......................................................... 88
9.3 Common functions ..................................................... 90
9.4 Common operators ...................................................... 91
9.5 Common variables ...................................................... 92
10.1 Visibility of single-query transactions ............................. 97
10.2 Visibility using multi-query transactions ......................... 98
10.3 Waiting for a lock ...................................................... 100
10.4 Deadlock ................................................................. 101
16.1 Interface summary ...................................................... 117
17.1 psql query buffer commands ........................................ 120
17.2 psql listing commands ................................................. 120
17.3 psql output commands ............................................... 120
17.4 psql special capabilities .............................................. 121
17.5 psql command-line arguments ..................................... 121
LIST OF TABLES

1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
Chapter 1

History of POSTGRESQL

1.1 Introduction

POSTGRESQL is the most advanced open source database server. In this chapter, you will learn about databases, open source software, and the history of POSTGRESQL.

There are three basic office productivity applications: word processors, spreadsheets, and databases. Word processors produce text documents critical to any business. Spreadsheets are used for financial calculations and analysis. Databases are used primarily for data storage and retrieval. You can use a word processor or a spreadsheet to store small amounts of data. However, with large volumes of data or data that must be retrieved and updated frequently, databases are the best choice. Databases allow orderly data storage, rapid data retrieval, and complex data analysis, as you will see in the coming chapters.

1.2 UNIVERSITY OF CALIFORNIA AT BERKELEY

POSTGRESQL’s ancestor was INGRES, developed at the UNIVERSITY OF CALIFORNIA AT BERKELEY (1977–1985). The INGRES code was taken and enhanced by RELATIONAL TECHNOLOGIES/INGRES CORPORATION\(^1\), which produced one of the first commercially successful relational database servers. Also at Berkeley, MICHAEL STONEBRAKER led a team to develop an object-relational database server called POSTGRES (1986–1994). The POSTGRES code was taken by ILLUSTR\(^2\) and developed into a commercial product. Two Berkeley graduate students, JOLLY CHEN and ANDREW YU, added SQL capabilities to POSTGRES, and called it POSTGRES95 (1994–1995). They left Berkeley, but Chen continued maintaining POSTGRES95, which had an active mailing list.

1.3 Development Leaves BERKELEY

In the summer of 1996, it became clear that the demand for an open source SQL database server was great, and a team was formed to continue development. MARC G. Fournier, Toronto, Canada, offered to host the mailing list, and provide a server to host the source tree. One thousand mailing list subscribers were moved to the new list. A server was configured, giving a few people login accounts to apply patches to the source code using cvs\(^3\).

---

\(^1\) Ingres Corp. was later purchased by Computer Associates.

\(^2\) Illustra was later purchased by Informix and integrated into Informix’s Universal Server.

\(^3\) cvs synchronizes access by developers to shared program files.
JOLLY CHEN had stated, "This project needs a few people with lots of time, not many people with a little time." With 250,000 lines of C code, we understood what he meant. In the early days, there were four major people involved, MARC FOUNNIER, THOMAS LOCKHART in Pasadena, California, VADIM MIKHEEV in Krasnoyarsk, Russia, and myself in Philadelphia, Pennsylvania. We all had full-time jobs, so we were doing this in our spare time. It certainly was a challenge.

Our first goal was to scour the old mailing list, evaluating patches that had been posted to fix various problems. The system was quite fragile then, and not easily understood. During the first six months of development, there was fear that a patch would break the system, and we would be unable to correct the problem. Many bug reports had us scratching our heads, trying to figure out not only what was wrong, but how the system even performed many functions.

We inherited a huge installed base. A typical bug report was, "When I do this, it crashes the database." We had a whole list of them. It became clear that some organization was needed. Most bug reports required significant research to fix, and many were duplicates, so our TODO list reported every buggy SQL query. It helped us identify our bugs, and made users aware of them too, cutting down on duplicate bug reports.

We had many eager developers, but the learning curve in understanding how the back-end worked was significant. Many developers got involved in the edges of the source code, like language interfaces or database tools, where things were easier to understand. Other developers focused on specific problem queries, trying to locate the source of the bug. It was amazing to see that many bugs were fixed with just one line of C code. POSTGRES had evolved in an academic environment, and had not been exposed to the full spectrum of real-world queries. During that period, there was talk of adding features, but the instability of the system made bug fixing our major focus.

1.4 POSTGRESQL Global Development Team

In late 1996, we changed the name from POSTGRES95 to POSTGRESQL. It is a mouthful, but honors the Berkeley name and SQL capabilities. We started distributing the source code using remote cvs, which allowed people to keep up-to-date copies of the development tree without downloading an entire set of files every day.

Releases were every 3–5 months. This consisted of 2–3 months of development, one month of beta testing, a major release, and a few weeks to issue sub-releases to correct serious bugs. We were never tempted to follow a more aggressive schedule with more releases. A database server is not like a word processor or a game, where you can easily restart it if there is a problem. Databases are multi-user, and lock user data inside the database, so we must make our software as reliable as possible.

Development of source code of this scale and complexity is not for the novice. We had trouble getting developers interested in a project with such a steep learning curve. However, our civilized atmosphere, and our improved reliability and performance, finally helped attract the experienced talent we needed.

Getting our developers the knowledge they needed to assist with POSTGRESQL was clearly a priority. We had a TODO list that outlined what needed to be done, but with 250,000 lines of code, taking on any TODO item was a major project. We realized developer education would pay major benefits in helping people get started. We wrote a detailed flowchart of the back-end modules. We wrote a developers' FAQ, to describe some of the common questions of POSTGRESQL developers. With this, developers became more productive at fixing bugs and adding features.

The source code we inherited from Berkeley was very modular. However, most Berkeley coders used POSTGRESQL as a test bed for research projects. Improving existing code was not a priority. Their coding

---

4C is a popular computer language first developed in the 1970's.
5All the files mentioned in this chapter are available as part of the POSTGRESQL distribution, or at http://www.postgresql.org/docs.
6Frequently Asked Questions
styles were also quite varied.

We wrote a tool to reformat the entire source tree in a consistent manner. We wrote a script to find functions that could be marked as static\(^7\), or never-called functions that could be removed completely. These are run just before each release. A release checklist reminds us of the items to be changed for each release.

As we gained knowledge of the code, we were able to perform more complicated fixes and feature additions. We redesigned poorly structured code. We moved into a mode where each release had major new features, instead of just bug fixes. We improved SQL conformance, added sub-selects, improved locking, and added missing SQL functionality. We added commercial-style telephone support.

The Usenet discussion group archives started touting us. In the previous year, we searched for POSTGRESQL, and found many people were recommending other databases, even though we were addressing user concerns as rapidly as possible. One year later, many people were recommending us to users who needed transaction support, complex queries, commercial-grade SQL support, complex data types, and reliability. This more clearly portrayed our strengths. Other databases were recommended when speed was the overriding concern. REDHAT’s shipment of POSTGRESQL as part of their LINUX\(^8\) distribution quickly multiplied our user base.

Every release is now a major improvement over the last. Our global development team now has mastery of the source code we inherited from Berkeley. Finally, every module is understood by at least one development team member. We are now easily adding major features, thanks to the increasing size and experience of our world-wide development team.

1.5 Open Source Software

POSTGRESQL is open source software. The term open source software often confuses people. With commercial software, a company hires programmers, develops a product, and sells it to users. With Internet communication, there are new possibilities. In open source software, there is no company. Capable programmers with interest and some free time get together via the Internet and exchange ideas. Someone writes a program and puts it in a place everyone can access. Other programmers join and make changes. When the program is sufficiently functional, they advertise the program’s availability to other Internet users. Users find bugs or missing features and report them back to the developers, who enhance the program.

It sounds like an unworkable cycle, but in fact it has several advantages:

• A company structure is not required, so there is no overhead and no economic restrictions.

• Program development is not limited to a hired programming staff, but taps the capabilities and experience of a large pool of Internet programmers.

• User feedback is facilitated, allowing program testing by a large number of users in a short period of time.

• Program enhancements can be rapidly distributed to users.

\(^7\)A static function is a function that is used by only one program file.

\(^8\)Linux is a popular UNIX-like, open source operating system.
CHAPTER 1. HISTORY OF POSTGRESQL
Chapter 2

Issuing Database Commands

At this point, the book assumes you have:

- POSTGRESQL installed
- POSTGRESQL server running
- You are a configured POSTGRESQL user
- You have created a database called test.

If not, please see appendix B.

In this chapter, you will learn how to connect to the database server, and issue simple commands to the POSTGRESQL server.

2.1 Starting a Database Session

POSTGRESQL uses a client/server model of communication. That means that a POSTGRESQL server continually runs, waiting for client requests. The server processes the request and returns the result to the client.

Choosing an Interface

Because the POSTGRESQL server runs as an independent process on the computer, there is no way for a user to interact with it directly. Instead, there are client applications designed specifically for user interaction. This chapter shows you how to interact with POSTGRESQL using the psql1 interface. Additional interfaces are covered in Chapter 16.

Choosing a Database

Each POSTGRESQL server controls access to a number of databases. Databases are storage areas used by the server to partition information. For example, a typical installation may have a production database, used to keep all information about a company. They may also have a training database, used for training and testing purposes. They may have private databases, used by individuals to store personal information. For this exercise, we will assume you have created an empty database called test. If this is not the case, see section B.5.
CHAPTER 2. ISSUING DATABASE COMMANDS

Starting a Session

To start a psql session and connect to the test database, type psql test at the command prompt. Your output should look similar to figure 2.1. Remember, the operating system command prompt is case-sensitive, so you must type this in all lowercase.\(^1\)

```
$ 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 startup

2.2 Controlling a Session

Congratulations. You have successfully connected to the PostgreSQL server. You can now issue commands, and receive replies from the server. Let’s try one. Type `SELECT CURRENT_USER;` and press Enter (see figure 2.2). If you make a mistake, just press `backspace` and retype. This should show your login name underneath the dashed line. In the example, the login name `postgres` is shown. The word `getpgusername` is a column label.

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

```
test=>
```

Figure 2.2: My first SQL query

dashed line. In the example, the login name `postgres` is shown. The word `getpgusername` is a column label. The server is also reporting that it has returned one row of data. The line `test=>` tells you that the server is done and is waiting for your next database query.

Let’s try another one. At the `test=>` prompt, type `SELECT CURRENT_TIMESTAMP;` and press Enter. It should show the current date and time. Each time you execute the query, the server will report the current time to you.

Typing in the Query Buffer

Typing in the query buffer is similar to typing at an operating system command prompt. However, at an operating system command prompt, `Enter` completes each command. In psql, commands are completed only

\(^1\)A few operating systems are case-insensitive.
when you enter a semicolon (;) or backslash-g (\g). Here’s a good example. Let’s do \texttt{SELECT 1 + 3}; but in a different way. See figure 2.3. Notice the query is spread over three lines. Notice the prompt changed

\begin{verbatim}
test=> SELECT
test-> 1 + 3
test-> ;
?column?
--------
  4
(1 row)
test=>
\end{verbatim}

Figure 2.3: Multi-line query

from => on the first line to -> on the second line to indicate the query was being continued. The semicolon told \texttt{psql} to send the query to the server. We could easily have replaced the semicolon with backslash-g. I don’t recommend you type queries as ugly as this one, but longer queries will benefit from the ability to spread them over multiple lines. You might notice the query is in uppercase. Unless you are typing a string in quotes, the POSTGRESQL server doesn’t care whether words are uppercase or lowercase. For stylistic reasons, I recommend you enter words special to POSTGRESQL in uppercase.

Try some queries on your own involving arithmetic. Each computation must start with the word \texttt{SELECT}, then your computation, and finally a semicolon or backslash-g to finish. For example, \texttt{SELECT 4 * 10;} would return 40. Addition is performed using plus (+), subtraction using minus (-), multiplication using asterisk (*), and division using forward slash (/).

If you have \texttt{readline} installed, \texttt{psql} will even allow you to use your arrow keys. Your \texttt{left} and \texttt{right} arrow keys allow you to move around, and the \texttt{up} and \texttt{down} arrows retrieve previously typed queries.

### Displaying the Query Buffer

You can continue typing indefinitely, until you use a semicolon or backslash-g. Everything you type will be buffered by \texttt{psql} until you are ready to send the query. If you use backslash-p (\p), you see everything accumulated in the query buffer. In figure 2.4, three lines of text are accumulated and displayed by the user using backslash-p. After display, we use backslash-g to execute the query which returns the value 21. This comes in handy with long queries.

### Erasing the Query Buffer

If you don’t like what you have typed, use backslash-r (\r) to reset or erase the buffer.

### 2.3 Getting Help

You might ask, “Are these backslash commands documented anywhere?” If you look at figure 2.1, you will see the answer is printed every time \texttt{psql} starts. Backslash-? (\?) prints all valid backslash commands. Backslash-h displays help for SQL commands. SQL commands are covered in the next chapter.

---

\footnote{Don't be concerned about \texttt{?column?}. We will cover that in section 4.7.}

\footnote{\texttt{Readline} is an open-source library that allows powerful command-line editing.}
2.4 Exiting a Session

This chapter would not be complete without showing you how to exit psql. Use `backslash-q (\q)` to `quit` the session. Backslash-q exits psql. Backslash g (go), p (print), r (reset), and q (quit) should be all you need for a while. Section 17.1 has more information about psql.
Chapter 3

Basic SQL Commands

SQL stands for *Structured Query Language*. It is the most common way of communicating with database servers, and is supported by almost all database systems. In this chapter, you will learn about relational database systems and how to issue the most important SQL commands.

3.1 Relational Databases

As I mentioned in section 1.1, the purpose of a database is rapid data storage and retrieval. Today, most database systems are *relational databases*. While the term *relational database* has a mathematical foundation, in practice it means that all data stored in the database is arranged in a uniform structure.

In figure 3.1, you see the database server with access to three databases, *test*, *demo*, and *finance*. You could issue the command `psql finance` and be connected to the *finance* database. You have already dealt with this in chapter 2. Using `psql`, you chose to connect to database *test* with the command `psql test`. To see a list of databases available at your site, type `psql -l`. The first column lists the database names. However, you may not have permission to connect to them.

You might ask, “What are those black rectangles in the databases?” Those are *tables*. Tables are the foundation of a *relational database management system (RDBMS)*. As I mentioned earlier, databases store...
data. Those tables are where data is stored in a database. Each table has a name defined by the person who created it.

Let’s look at a single table called friend in table 3.1. You can easily see how tables are used to store data.

<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

Each friend is listed as a separate row in the table. The table records five pieces of information about each friend, firstname, lastname, city, state, and age.¹

Each friend is on a separate row. Each column contains the same type of information. This is the type of structure that makes relational databases successful. Relational databases allow you to select certain rows of data, certain columns of data, or certain cells. You could select the entire row for Mike, the entire column for City, or a specific cell like Denver. There are synonyms for the terms table, row, and column. Table is more formally referred to as a relation or class, row as record or tuple, and column as field or attribute.

### 3.2 Creating Tables

Let’s create our own table and call it friend. The psql statement to create the table is shown in figure 3.2.

```sql
CREATE TABLE friend (
    firstname CHAR(15),
    lastname CHAR(20),
    city CHAR(15),
    state CHAR(2),
    age INTEGER
)
```

Figure 3.2: Create table

Let’s look at it from the top down. The words CREATE TABLE have special meaning to the database server. They indicate that the next request from the user is to create a table. You will find most SQL requests can be quickly identified by the first few words. The rest of the request has a specific format that is understood by the database server. While capitalization and spacing are optional, the format for a query must be followed. Otherwise, the database server will issue an error such as parse error at or near "pencil", meaning the database server got confused near the word pencil. In such a case, the manual page for the command should be consulted and the query reissued in the proper format. A copy of the PostgreSQL manual pages appear in appendix D.

The CREATE TABLE command follows a specific format. First, the two words CREATE TABLE, then the table name, then an open parenthesis, then a list of column names and their types, followed by a close parenthesis.

¹In a real-world database, the person’s birth date would be stored and not the person’s age. Age has to be updated every time the person has a birthday. A person’s age can be computed when needed from a birth date field.
The important part of this query is between the parentheses. You will notice there are five lines there. The first line, `firstname CHAR(15)`, represents the first column of the table to create. The word `firstname` is the name of the first column, and the text `CHAR(15)` indicates the column type and length. The `CHAR(15)` means the first column of every row holds up to 15 characters. The second column is called `lastname` and holds up to 20 characters. Columns of type `char` hold characters of a specified length. User-supplied character strings\(^2\) that do not fill the entire length of the field are right-padded with blanks. Columns `city` and `state` are similar. The final column, `age`, is different. It is not a `CHAR()` column. It holds whole numbers, not characters. Even if there were 5,000 friends in the table, you can be certain that there are no names appearing in the `age` column, only whole numbers. It is this structure that helps databases to be fast and reliable.

PostgreSQL supports more column types than just `char()` and `integer`. However, in this chapter we will use only these two. Sections 4.1 and 9.2 cover column types in more detail.

Create some tables yourself now. Only use letters for your table and column names. Don’t use any numbers, punctuation, or spaces at this time.

The `\d` command allows you to see information about a specific table, or a list of all table names in the current database. To see information about a specific table, type `\d` followed by the name of the table. For example, to see the column names and types of your new `friend` table in `psql`, type `\d friend`. Figure 3.3 shows this. If you use `\d` with no table name after it, you will see a list of all table names in the database.

```

Figure 3.3: Example of backslash-d
```

### 3.3 Adding Data with INSERT

Let’s continue toward the goal of making a table exactly like the `friend` table in table 3.1. We have the table created, but there is no data/friends in it. You add data into a table with the `INSERT` command. Just as `CREATE TABLE` has a specific format that must be followed, `INSERT` has a specific format too. You can see the format in figure 3.4. First, you must use single quotes around the character strings. Double quotes will not work. Spacing and capitalization are optional, except inside the single quotes. Inside them, the text is taken as literal, so any capitalization will be stored in the database exactly as you specify. If you type too many quotes, you might get to a point where your backslash commands don’t work anymore, and your prompt will appear as `test'>`. Notice the single-quote before the greater-than sign. Just type another single quote to get out of this mode, use `\r` to clear the query buffer and start again. Notice that the 19 doesn’t have quotes. It doesn’t need them because the column is a numeric column, not a character column. When you do your inserts, be sure to match each piece of data to the receiving column. Use the `INSERT` query in figure 3.4 as a sample and complete the insertion of the three friends shown in table 3.1.

\(^2\)A character string is a group of characters **strung** together.
CHAPTER 3. BASIC SQL COMMANDS

3.4 Viewing Data with SELECT

You have seen how to store data in the database. Let’s show you how to retrieve data. Surprisingly, there is only one command to get data out of the database, and that command is SELECT. You have already used SELECT in your first database query in figure 2.2 on page 6. SELECT has many variations. We are going to use it to show the rows in the table friend. The query is shown in figure 3.5. In this case, I put the entire query on one line. That’s fine. As queries get longer, breaking them into multiple lines helps make things clearer.

Let’s look at this in detail. First, we have the word SELECT, followed by an asterisk (*), then the word FROM, and our table name friend, and a semicolon to execute the query. The SELECT starts our command, and tells the database server what is coming next. The * tells the server we want all the columns from the table. The FROM friend tells which table we want to see. So, we have said we want all (*) columns from our table friend, and indeed, that is what is displayed. It should have the same data as table 3.1 on page 10.

As I mentioned, SELECT has a large number of variations, and we will look at a few of them now. First, suppose you want to retrieve only one of the columns from the friend table. You might already suspect that the asterisk (*) has to be changed in the query. If you replace the asterisk (*) with one of the column names, you will see only that column. Try SELECT city FROM friend. You can choose any of the columns. You can even choose multiple columns, by separating the names with a comma. For example, to see first and last names only, use SELECT firstname, lastname FROM friend. Try a few more SELECT commands until you get comfortable. If you specify a name that is not a valid column name, you will get an error message, ERROR: attribute 'mycolname' not found. If you try selecting from a table that does not exist, you will get an error message like ERROR: Relation 'mytablename' does not exist. POSTGRESQL is using the formal relational database terms relation and attribute in these error messages.
### 3.5 Selecting Specific Rows with WHERE

Let's take the next step in controlling the output of SELECT. In the previous section, we showed how to select only certain columns from the table. Now, we will show how to select only certain rows. The additional thing needed to do this is the WHERE clause. Without a WHERE clause, every row is returned.

The WHERE clause goes right after the FROM clause. In the WHERE clause, you specify the rows you want returned, as shown in figure 3.6. The query returns the rows that have an `age` column equal to 23. Figure 3.7 shows a more complex example that returns two rows. You can combine the column restrictions and the row restrictions in a single query, allowing you to select any single cell, or a block of cells. See figures 3.8 and 3.9.

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

Figure 3.6: My first WHERE

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

Figure 3.7: More complex WHERE clause

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

Figure 3.8: A single cell

Try using one of the other columns in the WHERE clause. Up to this point, we have made only comparisons on the `age` column. The `age` column is `integer`. The only tricky part about the other columns is that they are `char()` columns, so you have to put the comparison value in single quotes. You also have to match the capitalization exactly. See figure 3.10. If you had compared the `firstname` column to 'SAM' or 'sam', it would have returned no rows.

Try a few more until you are comfortable.

### 3.6 Removing Data with DELETE

We know how to add data to the database. Now we learn how to remove it. Removal is quite simple. The DELETE command can quickly remove any or all rows from a table. The command `DELETE FROM friend` will
test=> SELECT city, state FROM friend WHERE age >= 21;

+-------------+-----+
| city        | state|
|-------------+-----+
| Denver      | CO   |
| Allentown   | PA   |
+(2 rows)    

Figure 3.9: A block of cells

test=> SELECT * FROM friend WHERE firstname = 'Sam';

+---------------+--------------------+---------------+-----+----+
| firstname     | lastname            | city          | state| age|
|---------------+--------------------+---------------+-----+----+
| Sam           | Jackson             | Allentown     | PA   | 22 |
+(1 row)       

Figure 3.10: Comparing string fields

delete all rows from the table friend. The query DELETE FROM friend WHERE age = 19 will remove only those rows that have an age column equal to 19.

Here is a good exercise. INSERT a row into the friend table, use SELECT to verify the row has been properly added, then use DELETE to remove the row. This combines the things you learned in the previous sections. Figure 3.11 shows an example.

### 3.7 Modifying Data with UPDATE

How do you modify data already in the database? You could use DELETE to remove the row, then INSERT to insert a new row, but that is quite inefficient. The UPDATE command allows you to update data already in the database. It follows a format similar to the previous commands.

Mike had a birthday, so we want to update his age in this table. Figure 3.12 shows an example. The example shows the word UPDATE, the table name friend, followed by SET, then the column name, the equals sign (=), and the new value. The WHERE clause restricts the number of rows affected by the update, as in DELETE. Without a WHERE clause, all rows are updated.

Notice that the Mike row has moved to the end of the list. The next section will show you how to control the order of the row display.

### 3.8 Sorting Data with ORDER BY

In a SELECT query, rows are displayed in an undetermined order. If you want to guarantee the rows are returned from SELECT in a specific order, you need to add the ORDER BY clause to the end of the SELECT. Figure 3.13 shows the use of ORDER BY. You can reverse the order by adding DESC, as seen in figure 3.14. If the query were to use a WHERE clause too, the ORDER BY would appear after the WHERE clause, as in figure 3.15.

You can ORDER BY more than one column by specifying multiple column names or labels, separated by commas. It would sort by the first column specified. For rows with equal values in the first column, it would sort based on the second column specified. Of course, this does not make sense in the friend example because all column values are unique.
3.8. SORTING DATA WITH ORDER BY

```sql
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)

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

INSERT 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>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>
<tr>
<td>Jim</td>
<td>Barnes</td>
<td>Ocean City</td>
<td>NJ</td>
<td>25</td>
</tr>
</tbody>
</table>

(4 rows)

Figure 3.11: DELETE example

```sql

```sql
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 3.12: My first UPDATE
```
### Figure 3.13: Use of ORDER BY

```sql
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: Reverse ORDER BY

```sql
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: Use of ORDER BY and WHERE

```sql
test=> 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)
```
3.9 Destroying Tables

This chapter would not be complete without showing how to destroy tables. It is accomplished using the DROP TABLE command. The command DROP TABLE friend will remove the friend table. Both the table structure and the data contained in the table will be erased. We will be using the friend table in the next chapter, so I don’t recommend you remove the table at this time. Remember, to remove only the data in the table, without removing the table structure itself, use DELETE.

3.10 Summary

In this chapter, we have shown you the basic operations of any database:

- Table creation (CREATE TABLE)
- Table destruction (DROP TABLE)
- Adding (INSERT)
- Displaying (SELECT)
- Removing (DELETE)
- Replacing (UPDATE)
Chapter 4

Customizing Queries

This chapter will illustrate additional capabilities of the basic SQL commands.

4.1 Data types

Table 4.1 shows the most common column data types. Figure 4.1 shows queries using these types. There

<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

is table creation, INSERT, and SELECT. There are a few things of interest in this example. First, notice how
the numbers do not require quotes, while character strings, dates, and times require them. Also note the
timestamp column displays its value in the standard UNIX date\footnote{This is the format generated by typing the command date at the UNIX command prompt.} format. It also displays the time zone.

The final SELECT uses psql’s \x display mode.\footnote{See section 17.1 for a full list of the psql backslash commands.} Without the \x, the SELECT would have displayed too much information to fit on one line. The fields would have wrapped around the edge of the display, making it hard to read. The columns would still line up, but there would be other data in the way. Of course, another solution to field wrapping is to select fewer columns. Remember, you can select any columns from the table in any order.

Section 9.2 covers column types in more detail.

4.2 Quotes Inside Text

Suppose you want to insert the name O’Donnell. You might be tempted to enter this in psql as ’O’Donnell’,
but this will not work. The presence of a single quote inside a single-quoted string generates a parse error.
CREATE TABLE alltypes (  
  state CHAR(2),  
  name CHAR(30),  
  children INTEGER,  
  distance FLOAT,  
  budget NUMERIC(16,2),  
  born DATE,  
  checkin TIME,  
  started TIMESTAMP  
);  
CREATE  
INSERT INTO alltypes  
VALUES (  
  'PA',  
  'Hilda Blairwood',  
  3,  
  10.7,  
  4308.20,  
  '9/21/74',  
  '9:00',  
  '07/03/1996 10:30:00');  
SELECT state, name, children, distance, budget FROM alltypes;  
state|name |children|distance| budget  
-----+------------------------------+--------+--------+-------  
PA |Hilda Blairwood | 3| 10.7|4308.20  
(1 row)  
SELECT born, checkin, started FROM alltypes;  
born|checkin |started  
----------+--------+----------------------------  
09-21-1974|09:00:00|Wed Jul 03 10:30:00 1996 EDT  
(1 row)  
SELECT * FROM alltypes;  
- [ RECORD 1 ] -----------------------------  
state | PA  
name | Hilda Blairwood  
children | 3  
distance | 10.7  
budget | 4308.20  
born | 09-21-1974  
checkin | 09:00:00  
started | Wed Jul 03 10:30:00 1996 EDT

Figure 4.1: Example of common data types
4.3 Using NULL Values

Let’s return to the INSERT statement described in section 3.3 on page 11. We will continue to use the friend table from the previous chapter. In figure 3.4, we specified a value for friend column. Suppose we wanted to insert a new row, but did not want to supply data for all the columns, i.e. we want to insert information about Mark, but we don’t know Mark’s age.

Figure 4.2 shows this. After the table name, we have column names in parentheses. These columns will be assigned, in order, to the supplied data values. If we were supplying data for all columns, we wouldn’t need to name them. In this example, we must name the columns. The table has five columns, but we are only supplying four data values.

The column we did not assign was age. The interesting question is, “What is in the age cell for Mark?”.

The answer is that the age cell contains a NULL value.

NULL is a special value that is valid in any column. It is used when a valid entry for a field is not known or not applicable. In the previous example, we wanted to add Mark to the database but we didn’t know his age. It is hard to imagine what numeric value could be used for Mark’s age column. Zero or minus-one would be strange age values. NULL is the perfect value for his age.

Suppose we had a spouse column. What value should be used if someone is not married? A NULL value would be the proper value for that field. If there were a wedding_anniversary column, unmarried people would have a NULL value in that field. NULL values are very useful. Before databases supported NULL values, users would put special values in columns, like -1 for unknown numbers and 1/1/1900 for unknown dates. NULLs are much clearer.

NULLs have a special behavior in comparisons. Look at figure 4.3. First, notice the age column for Mark is empty. It is really a NULL. In the next query, because NULL values are unknown, the NULL row does not appear in the output. The third query really confuses people. Why doesn’t the Mark row appear? The age is NULL or unknown, meaning the database doesn’t know if it equals 99 or not, so it doesn’t guess. It refuses to print it. In fact, there is no comparison that will produce the NULL row, except the last query shown. The tests is NULL and IS NOT NULL are designed specifically to test for the existence of NULL values. NULLs often confuse new users. Remember, if you are making comparisons on columns that could contain NULL values, you must test for them specifically.

Figure 4.4 shows an example. We have inserted Jack, but the city and state were not known, so they are set to NULL. The next query’s WHERE comparison is contrived, but illustrative. Because city and state are both NULL, you might suspect that the Jack row would be returned. However, because NULL means unknown, there is no way to know if the two NULL values are equal. Again, POSTGRESQL doesn’t guess, and refuses to print it.

---

3 That is not a double quote between the O and D. Those are two single quotes.

4 The <> means not equal.
CHAPTER 4. CUSTOMIZING QUERIES

2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2900
2901
2902
2903
2904

---

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

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

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

```
test=> SELECT * FROM friend WHERE age IS NULL;
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 19053 1
```

```
test=> SELECT * FROM friend WHERE city = state;
firstname|lastname|city|state|age
---------+--------+----+-----+---
(0 rows)
```

---

**Figure 4.4:** Comparison of NULL fields
4.4. CONTROLLING DEFAULT VALUES

There is one more issue with NULLs that needs clarification. In character columns, a NULL is not the same as a zero length field. That means that the string '' and NULL are different. Figure 4.5 shows an example of this. There are no valid numeric and date blank values, but a character string can be blank. When viewed in psql, any numeric field that is blank has to contain a NULL because there is no blank number. However, there are blank strings, so blank strings and NULLs are displayed the same in psql. However, they are not the same, so be careful not to confuse the meaning of NULLs in character fields.

```
CREATE TABLE nulltest (name CHAR(20), spouse CHAR(20));
CREATE
INSERT INTO nulltest VALUES ('Andy', '');
INSERT 18986 1
INSERT INTO nulltest VALUES ('Tom', NULL);
INSERT 18987 1
SELECT * FROM nulltest;
name | spouse
------------------------+------------------------
Andy |                     
Tom  |                     
(2 rows)
SELECT * FROM nulltest WHERE spouse = '';
name | spouse
------------------------+------------------------
Andy |                     
(1 row)
SELECT * FROM nulltest WHERE spouse IS NULL;
name | spouse
------------------------+------------------------
Tom  |                     
(1 row)
```

Figure 4.5: NULLs and blank strings

4.4 Controlling DEFAULT Values

As we learned in the previous section, columns not specified in an INSERT statement are given NULL values. This can be changed using the DEFAULT keyword. When creating a table, next to each column type, you can use the keyword DEFAULT and then a value. The value will be used anytime the column value is not supplied in an INSERT. If no DEFAULT is defined, a NULL is used for the column. Figure 4.6 shows a typical use of default values. The default for the timestamp column is actually a call to an internal POSTGRESQL variable that returns the current date and time. If any value is supplied for a field with a default, that value is used instead.
```
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 | Sat Nov 13 13:50:15 1994 EST
```

(1 row)

**Figure 4.6: Using DEFAULTS**

### 4.5 Column Labels

You might have noticed the text that appears at the top of each column in the `SELECT` output. That is called the *column label*. Usually, the label is the name of the column being selected. However, you can control what text appears at the top of each column by using the `AS` keyword. For example, figure 4.7 replaces the default column label `firstname` with the column label `buddy`. You might have noticed that the query in figure 2.3 on page 7 has the column label `?column?`. The database server returns this label when there is no suitable label. In that case, the result of an addition doesn’t have an appropriate label. Figure 4.8 shows the same query with an appropriate label added using `AS`.

```
SELECT firstname AS buddy FROM friend;
```

```
buddy
---------------
Cindy
Sam
Mike
Mark
```

(4 rows)

**Figure 4.7: Controlling column labels**

### 4.6 Comments

`POSTGRESQL` allows you to place any text into `psql` for use as comments. There are two comment styles. The presence of two dashes (`--`) marks all text to the end of the line as a comment. `POSTGRESQL` also understand C-style comments, where the comment begins with `/*` and ends with `*/`. Figure 4.9 shows these comment styles. Notice how the multi-line comment is marked by a `psql`
4.7 AND/OR Usage

Up to this point, there have been only simple WHERE clause tests. In the next few sections, we will show how to do more complex WHERE clause testing.

Complex WHERE clause tests are done by connecting simple tests using the words AND and OR. For illustration, I have loaded the friend table with new people in figure 4.10. Selecting certain rows from the table will require more complex WHERE conditions. For example, if we wanted to select Sandy Gleason by name, it would be difficult with only one comparison in the WHERE clause. If we tested for firstname = 'Sandy', we would select both Sandy Gleason and Sandy Weber. If we tested for lastname = 'Gleason', we would get both Sandy Gleason and her brother Dick Gleason. The proper way is to use AND to join tests of both firstname and lastname. The proper query is shown in figure 4.11. The AND joins the two comparisons we need.

A similar comparison could be done to select friends living in Cedar Creek, Maryland. There could be other friends living in Cedar Creek, Ohio, so the comparison city = 'Cedar Creek' is not enough. The proper test is city = 'Cedar Creek' AND state = 'MD'.

---

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

Figure 4.8: Computation using a column label

command prompt of *>. It is a reminder you are in a multi-line comment, just as -> is a reminder you are in a multi-line statement, and '>' is a reminder you are in a multi-line quoted string.

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

Figure 4.9: Comment styles
CHAPTER 4. CUSTOMIZING QUERIES

```
test=> SELECT * FROM friend

firstname |lastname |city |state|age
---------------+--------------------+---------------+-----+---
Sandy |Gleason |Ocean City |NJ | 25
(1 row)
```

Figure 4.11: WHERE test for Sandy Gleason

Another complex test would be to select people who are in the state of New Jersey (NJ) or Pennsylvania (PA). Such a comparison requires the use of OR. The test state = 'NJ' OR state = 'PA' would return the desired rows, as shown in figure 4.12.

```
test=> SELECT * FROM friend

firstname |lastname |city |state|age
---------------+--------------------+---------------+-----+---
Dick |Gleason |Ocean City |NJ | 19
Sandy |Gleason |Ocean City |NJ | 25
Victor |Tabor |Williamsport |PA | 22
(3 rows)
```

Figure 4.12: Friends in New Jersey and Pennsylvania

An unlimited number of ANDs and ORs can be linked together to perform complex comparison tests. When ANDs are linked with other ANDs, there is no possibility for confusion. The same is true of ORs. However, when ANDs and ORs are both used in the same query, the results can be confusing. Figure 4.13 shows such a case. You might suspect that it would return rows with firstname equal to Victor and state equals PA or NJ. In fact, the query returns rows with firstname equal to Victor and state equals PA or NJ. In this case, AND is evaluated first, then OR. When mixing ANDs and ORs, it is best to collect the ANDs and ORs into common groups using parentheses. Figure 4.14 shows the proper way to enter this query. Without parentheses, it is very difficult to understand a query with mixed ANDs and ORs.

```
test=> SELECT * FROM friend

firstname |lastname |city |state|age
---------------+--------------------+---------------+-----+---
Dick |Gleason |Ocean City |NJ | 19
Sandy |Gleason |Ocean City |NJ | 25
Victor |Tabor |Williamsport |PA | 22
(3 rows)
```

Figure 4.13: Mixing ANDs and ORs
4.8. RANGE OF VALUES

Suppose we wanted to see all friends who had ages between 22 and 25. Figure 4.15 shows two queries that produce this result. The first query uses AND to perform two comparisons that both must be true. We used <= and >= so the age comparisons included the limiting ages of 22 and 25. If we used < and > the ages 22 and 25 would not have been included in the output. The second query uses BETWEEN to generate the same comparison. BETWEEN comparisons include the limiting values in the result.

4.9 LIKE Comparison

Greater-than and less-than comparisons are possible, as shown in table 4.2. Even more complex comparisons are available. Users often need to compare character strings to see if they match a certain pattern. For example, sometimes they only want fields that begin with a certain letter, or contain a certain word. The LIKE keyword allows such comparisons. The query in figure 4.16 returns rows where the firstname begins with D. The percent (%) is interpreted to mean any characters can follow the D. The query performs the test firstname LIKE 'D%'.

The test firstname LIKE '%D%' returns rows where firstname contains a D anywhere in the field, not just at the beginning. The effect of the having a % before and after a character allows the character to appear anywhere in the string.

More complex tests can be performed with LIKE, as shown in table 4.3. While percent (%) matches an unlimited number of characters, the underscore (_) matches only a single character. The underscore allows any single character to appear in its position. To test if a field does not match a pattern, use NOT LIKE. To test for an actual percent sign (%), use %%. An actual underscore (_) is tested with two underscores.

Attempting to find all character fields that end with a certain character can be difficult. For char() columns, like firstname, there are trailing spaces that make such trailing comparisons difficult with LIKE. Other character column types don’t use trailing spaces. Those can use the test colname LIKE '%g' to find all rows that end with g. See section 9.2 for complete coverage on character data types.
CHAPTER 4. CUSTOMIZING QUERIES

```
test=> SELECT * 
  test-> FROM friend 
  test-> WHERE age >= 22 AND age <= 25; 

<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; 

<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 WHERE firstname LIKE 'D%'; 

<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.

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

Table 4.3: LIKE comparison
4.10 Regular Expressions

Regular expressions allow more powerful comparisons than the more standard LIKE and NOT LIKE. Regular expression comparisons are a unique feature of PostgreSQL. They are very common in UNIX, such as in the UNIX grep command.\(^5\)

Table 4.4 shows the regular expression operators and table 4.5 shows the regular expression special characters. Note that the caret (`^`) has a different meaning outside and inside square brackets (`[]`). While regular expressions are powerful, they are complex to create. Table 4.6 shows some examples. Figure 4.17 shows examples of queries using regular expressions. For a description, see the comment above each query.

\(^5\) Actually, PostgreSQL regular expressions are like egrep extended regular expressions.

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>regular expression</td>
<td><code>~</code></td>
</tr>
<tr>
<td>regular expression, case insensitive</td>
<td><code>~*</code></td>
</tr>
<tr>
<td>not equal to regular expression</td>
<td><code>!~</code></td>
</tr>
<tr>
<td>not equal to regular expression, case insensitive</td>
<td><code>!*</code></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><code>~</code></td>
</tr>
<tr>
<td>end</td>
<td><code>$</code></td>
</tr>
<tr>
<td>any single character</td>
<td>.</td>
</tr>
<tr>
<td>set of characters</td>
<td><code>[ccc]</code></td>
</tr>
<tr>
<td>set of characters not equal</td>
<td><code>[^ccc]</code></td>
</tr>
<tr>
<td>range of characters</td>
<td>[c-c]</td>
</tr>
<tr>
<td>range of characters not equal</td>
<td><code>[^c-c]</code></td>
</tr>
<tr>
<td>zero or one of previous character</td>
<td><code>?</code></td>
</tr>
<tr>
<td>zero or multiple of previous characters</td>
<td><code>*</code></td>
</tr>
<tr>
<td>one or multiple of previous characters</td>
<td><code>+</code></td>
</tr>
<tr>
<td>OR operator</td>
<td>`</td>
</tr>
</tbody>
</table>

Table 4.5: Regular expression special characters

Figure 4.18 shows two more complex regular expressions. The first query shows the way to properly test for a trailing n. Because `char()` columns have trailing space to fill the column, you need to test for possible trailing spaces. See section 9.2 for complete coverage on character data types. The second query might be surprising. Some think it returns rows that do not contain an S. Instead, the query returns all rows that have any character that is not an S. Sandy contains characters that are not S, such as a, n, d, and y, so that row is returned. The test would only prevent rows containing only S’s from being printed.

You can test for the literal characters listed in table 4.5. For example, to test for a dollar sign, use `\$`. To test for an asterisk, use `\*`. The backslash removes any special meaning from the character that follows it. To test for a literal backslash, use two backslashes, like `\\`. This is different from LIKE special character literal handling, where `%%` was used to test for a literal percent sign.

Because regular expressions have a powerful special character command set, creating them can be difficult. Try some queries on the `friend` table until you are comfortable with regular expression comparisons.
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)

(6 rows)

*test=>* -- firstname begins with 'S'

*test=>* SELECT * FROM friend WHERE firstname \^ 'S' 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)

(2 rows)

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

*test=>* SELECT * FROM friend WHERE firstname \^ 'e' 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)

(2 rows)

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

*test=>* SELECT * FROM friend WHERE firstname \^ '* [bc]' 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)

(2 rows)

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

*test=>* SELECT * FROM friend WHERE firstname !\^ 's' 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)

(4 rows)

Figure 4.17: Regular expression sample queries
Test | Operation
--- | ---
begins with D | \`'D'`
begins with D and contains e | \`'D.*e'`
begins with D, contains e, and then f | \`'D.*e.*f'`
contains A, B, C, or D | \`(A-D)' or \`(ABCD)'`
contains A or a | \`'*a' or \'[Aa]'`
does not contain D | \`'D'`
does not begin with D | \`'D'` or \`'[A-D]'`
contains D, with one optional leading space | \`'^?D'`
contains D, with optional leading spaces | \`'^D'`
contains D, with at least one leading space | \`'^+D'`
ends with G, with optional trailing spaces | \`'G *$'`

Table 4.6: Regular expression examples

test=> -- firstname ends with n

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dean</td>
<td>Yeager</td>
</tr>
<tr>
<td></td>
<td>Plymouth</td>
</tr>
<tr>
<td></td>
<td>MA</td>
</tr>
<tr>
<td></td>
<td>24</td>
</tr>
</tbody>
</table>

(1 row)

test=> -- firstname contains a non-S character

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

(6 rows)

Figure 4.18: Complex regular expression queries
4.11 Case Clause

Many programming languages have conditional statements, stating if condition is true then do-something, else do-something-else. This allows execution of statements based on some condition. While SQL is not a procedural programming language, it does allow conditional control over what data is returned from a query. The WHERE clause uses comparisons to control row selection. The CASE statement allows comparisons in column output. Figure 4.19 shows a query using CASE to create a new output column showing adult or minor as appropriate, based on the age field. Of course, the values adult and minor do not appear in the table friend.

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

Figure 4.19: CASE example

The CASE clause allows the creation of those conditional strings.

A more complex example is shown in figure 4.20. In this example, there are multiple WHEN clauses. The AS clause is used to label the column with the word distance. Though I have shown only SELECT examples, CASE can be used in UPDATE and other complex situations. CASE allows the creation of conditional values, which can be used for output or for further processing in the same query. CASE values only exist inside a single query, so they can’t be used outside the query that defines them.

4.12 Distinct Rows

It is often desirable to return the results of a query with no duplicates. The keyword DISTINCT prevents duplicates from being returned. Figure 4.21 shows the use of the DISTINCT keyword to prevent duplicate states and duplicate city and state combinations. Notice DISTINCT operates only on the columns selected in the query. It does not compare non-selected columns when determining uniqueness. Section 5.2 shows how counts can be generated for each of the distinct values.
4.13 Functions and Operators

There are a large number of functions and operators available in PostgreSQL. Function calls take zero, one, or more arguments and return a single value. You can list all functions and their arguments using psql’s \df command. You can use psql’s \dd command to display comments about any specific function or group of functions, as shown in figure 4.22.

Operators differ from functions in the following ways:

- Operators are symbols, not names
- Always take two arguments
- Arguments appear to the left and right of the operator symbol

For example, + is an operator that takes one argument on the left and one on the right, and returns their sum. Psql’s \do command lists all PostgreSQL operators and their arguments. Figure 4.23 shows operator listings and their use. The standard arithmetic operators: addition (+), subtraction (-), multiplication (*), division (/), modulo/remainder (%), and exponentiation (^) honor the standard precedence rules. Exponentiation is done first, multiplication, division, and modulo are second, and addition and subtraction are performed last. Parentheses can be used to alter this precedence. Other operators are evaluated left-to-right, unless parentheses are present.

4.14 SET, SHOW, and RESET

The set command allows the changing of various PostgreSQL parameters. The changes remain in effect for the duration of the database connection. Table 4.7 shows various parameters that can be controlled with set.

Datestyle controls the appearance of dates when printed in psql as seen in table 4.8. It controls the
test=> SELECT state FROM friend ORDER BY state;
    state 
-------
     MA  
     MA  
     MD  
     NJ  
     NJ  
     PA  
(6 rows)

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

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

Figure 4.21: DISTINCT prevents duplicates

<table>
<thead>
<tr>
<th>Function</th>
<th>SET option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datestyle</td>
<td>DATESTYLE TO 'POSTGRES'/'SQL'/'ISO'/'GERMAN'/'US'/'NFEUROPEAN'/'EUROPEAN'</td>
</tr>
<tr>
<td>Timezone</td>
<td>TIMEZONE TO 'value'</td>
</tr>
<tr>
<td>Client encodings</td>
<td>CLIENT_ENCODING NAMES TO encoding</td>
</tr>
<tr>
<td>Server encodings</td>
<td>SERVER_ENCODING TO encoding</td>
</tr>
<tr>
<td>Transaction isolation level</td>
<td>TRANSACTION ISOLATION LEVEL 'SERIALIZABLE'/'READ COMMITTED'</td>
</tr>
<tr>
<td>Optimizer heap cost</td>
<td>COST_HEAP TO #</td>
</tr>
<tr>
<td>Optimizer index cost</td>
<td>COST_INDEX TO #</td>
</tr>
<tr>
<td>Genetic Query Optimizer (GEQO)</td>
<td>GEQO TO 'ON[= #]'/'OFF'</td>
</tr>
<tr>
<td>Key Set Query Optimizer (KSQO)</td>
<td>KSQO TO 'ON'/'OFF'</td>
</tr>
<tr>
<td>Enable trace flags</td>
<td>PG_OPTIONS TO 'value'</td>
</tr>
</tbody>
</table>

Table 4.7: SET options
4.14. SET, SHOW, AND RESET

test=> \df

<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>numeric</td>
<td>abs</td>
<td>numeric</td>
</tr>
</tbody>
</table>

- 

test=> \df int

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

- 

test=> \df upper

<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)

test=> \dd upper

Object descriptions

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

(1 row)

test=> SELECT upper('jacket');

upper

--------

JACKET

test=> SELECT sqrt(2.0); -- square root

sqrt

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

1.4142135624

(1 row)

Figure 4.22: Function examples
### List of operators

<table>
<thead>
<tr>
<th>Op</th>
<th>Left arg</th>
<th>Right arg</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>int4</td>
<td></td>
<td>int4</td>
</tr>
<tr>
<td>!!</td>
<td></td>
<td>int4</td>
<td>int4</td>
</tr>
<tr>
<td>!=</td>
<td>int4</td>
<td>name</td>
<td>bool</td>
</tr>
</tbody>
</table>

### List of operators

<table>
<thead>
<tr>
<th>Op</th>
<th>Left arg</th>
<th>Right arg</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>box</td>
<td>point</td>
<td>box</td>
</tr>
<tr>
<td>/</td>
<td>char</td>
<td>char</td>
<td>char</td>
</tr>
</tbody>
</table>

### List of operators

<table>
<thead>
<tr>
<th>Op</th>
<th>Left arg</th>
<th>Right arg</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>float8</td>
<td>float8</td>
<td>float8</td>
</tr>
</tbody>
</table>

### Object descriptions

<table>
<thead>
<tr>
<th>Name</th>
<th>What</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>operator</td>
<td>exponentiation</td>
</tr>
</tbody>
</table>

```sql
SELECT 2 + 3 ^ 4;
```

<table>
<thead>
<tr>
<th>?column?</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>83</td>
</tr>
</tbody>
</table>

Figure 4.23: Operator examples
4.14. SET, SHOW, AND RESET

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

Table 4.8: DATESTYLE output

(format (slashes, dashes, or year first), and the display of the month first (US) or day first (European). The command SET DATESTYLE TO 'SQL,US' would most likely be used by users in the USA, while Europeans might prefer SET DATESTYLE TO 'POSTGRES,EUROPEAN'. The ISO DATESTYLE and GERMAN DATESTYLE are not affected by any of the other options.

TIMEZONE defaults to the timezone of the server or the PGTZ environment variable. The psql client might be in a different timezone, and SET TIMEZONE allows this to be changed inside psql. TRANSACTION ISOLATION will be covered in section 10.3.

The other options are for developer use during debugging sessions. The SET manual page covers them in more detail.

The SHOW command is used to display current database session parameters. RESET allows session parameters to be reset to their default values. Figure 4.24 shows an example of this.  

```
test=> SHOW DATESTYLE; 
NOTICE: DateStyle is Postgres 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 Postgres with US (NonEuropean) conventions
SHOW VARIABLE
```

Figure 4.24: SHOW and RESET examples

6Your site defaults may be different.
Chapter 5

SQL Aggregates

Users often require the ability to summarize database information. Instead of seeing all rows, they want just a count or total. This is called aggregation or gathering together. This chapter deals with PostgreSQL's ability to generate summarized database information using aggregates.

5.1 Aggregates

There are five aggregates outlined in table 5.1. COUNT operates on entire rows. The others operate on specific columns. Figure 5.1 shows examples of aggregate queries.

<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>column total</td>
</tr>
<tr>
<td>MAX(colname)</td>
<td>column maximum</td>
</tr>
<tr>
<td>MIN(colname)</td>
<td>column minimum</td>
</tr>
<tr>
<td>AVG(colname)</td>
<td>column average</td>
</tr>
</tbody>
</table>

Table 5.1: Aggregates

Aggregates can be combined with the WHERE clause to produce more complex results. The query SELECT AVG(age) FROM friend WHERE age >= 21 computes the average age of people age 21 or older. This prevents Dick Gleason from being included in the average computation because he is younger than 21. The column label defaults to the name of the aggregate. You can use AS to change it, as shown in section 4.5.

NULLs are not processed by most aggregates, like MAX(), SUM(), and AVG(). If a column is NULL, it is skipped and the result is not affected by any NULL values. However, if a column contains only NULL values, the result is NULL, not zero. COUNT(*) is different. It does count NULLs because it is looking at entire rows by using the asterisk(*). It is not looking at individual columns like the other aggregates. To find the COUNT of all non-NULL values in a certain column, use COUNT(columnname).

Figure 5.2 illustrates aggregate handling of NULLs. First, a single row containing a NULL column is used to show aggregates returning NULL results. Two versions of COUNT on a NULL column are shown. Notice COUNT never returns a NULL value. Then, a single non-NULL row is inserted, and the results shown. Notice the AVG() of 3 and NULL is 3, not 1.5, illustrating the NULL is not involved in the average computation.
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 5.1: Aggregate examples
5.1. AGGREGATES

```
test=> CREATE TABLE aggtest (col INTEGER);
CREATE

test=> INSERT INTO aggtest VALUES (NULL);
INSERT 18826 1

test=> SELECT SUM(col) FROM aggtest;
    sum
    -----

(1 row)

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

(1 row)

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

(1 row)

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

(1 row)

test=> INSERT INTO aggtest VALUES (3);
INSERT 18827 1

test=> SELECT AVG(col) FROM aggtest;
       avg
       -----
          3

(1 row)

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

(1 row)

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

(1 row)
```

Figure 5.2: Aggregates and NULLs
5.2 Using GROUP BY

Simple aggregates return one row as a result. It is often desirable to apply an aggregate to groups of rows. Queries using aggregates with GROUP BY have the aggregate applied to rows grouped by another column in the table. For example, SELECT COUNT(*) FROM friend returns the total number of rows in the table. The query in figure 5.3 shows the use of GROUP BY to generate a count of the number of people in each state. COUNT(*) is not applied to the entire table at once. With GROUP BY, the table is split up into groups by state, and COUNT(*) is applied to each group.

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

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

Figure 5.3: Aggregate with GROUP BY

The second query shows the minimum, maximum, and average age of the people in each state. It also shows an ORDER BY on the aggregate column. Because the column is the fourth column in the result, you can identify the column by the number 4. Doing ORDER BY avg would have worked too. You can GROUP BY more than one column, as shown in figure 5.4.

GROUP BY collects all NULL values into a single group. Psql’s \da command lists all the aggregates supported by POSTGRESQL

5.3 Using HAVING

There is one more aggregate capability that is often overlooked. It is the HAVING clause. HAVING allows you to perform conditional tests on aggregate values. It is often used with GROUP BY. With HAVING, you can include or exclude groups based on the aggregate value for that group. For example, suppose you want to know all the states where there is more than one friend. Looking at the first query in figure 5.3, you can see exactly which states have more than one friend. HAVING allows you to programmatically test on the count
5.4. QUERY TIPS

```sql
test=> SELECT city, state, COUNT(*)
test-> FROM friend
test-> GROUP BY state, city;
test-> ORDER BY 1, 2

<table>
<thead>
<tr>
<th>city</th>
<th>state</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
<td>MA</td>
<td>1</td>
</tr>
<tr>
<td>Cedar Creek</td>
<td>MD</td>
<td>1</td>
</tr>
<tr>
<td>Ocean City</td>
<td>NJ</td>
<td>2</td>
</tr>
<tr>
<td>Plymouth</td>
<td>MA</td>
<td>1</td>
</tr>
<tr>
<td>Williamsport</td>
<td>PA</td>
<td>1</td>
</tr>
</tbody>
</table>

(5 rows)
```

Figure 5.4: GROUP BY on two columns

column, as shown in figure 5.5. Aggregates can’t be used in a WHERE clause. They are valid only inside

```sql
test=> SELECT state, COUNT(*)
test-> FROM friend
test-> GROUP BY state
test-> HAVING COUNT(*) > 1;
```

<table>
<thead>
<tr>
<th>state</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>MA</td>
<td>2</td>
</tr>
<tr>
<td>NJ</td>
<td>2</td>
</tr>
</tbody>
</table>

(2 rows)

Figure 5.5: HAVING usage

HAVING.

5.4 Query Tips

In figures 5.3 and 5.5, the queries are spread over several lines. When a query has several clauses, like FROM, WHERE, and GROUP BY, it is best to place each clause on a separate line. It makes queries easier to understand. Clear queries also use appropriate capitalization.

In a test database, it isn’t a problem if you make a mistake. In a live, production database, one incorrect query can cause great difficulties. It takes five seconds to issue an erroneous query, and sometimes five days to recover from it. Double-check your queries before executing them. This is especially important for UPDATE, DELETE, and INSERT queries because they modify the database. Also, before performing UPDATE or DELETE, do a SELECT or SELECT COUNT(*) with the same WHERE clause. Make sure the SELECT result is reasonable before doing the UPDATE or DELETE.
Chapter 6

Joining Tables

This chapter shows how to store data using multiple tables. Multi-table storage and multi-table queries are fundamental to relational databases.

We start this chapter with table and column references. These are important in multi-table queries. Then, we cover the advantages of splitting data across multiple tables. Next, we introduce an example based on a mail order company, showing table creation, insertion, and queries using joins. Finally, we explore various join types.

6.1 Table and Column References

Before dealing with joins, there is one important feature that must be mentioned. Up to this point, all queries have involved a single table. With multiple tables in a query, column names get confusing. Unless you are familiar with each table, it is difficult to know which column names belong to which tables. Sometimes two tables have the same column name. For these reasons, SQL allows you to fully qualify column names by preceding the column name with the table name. An example of table name prefixing is shown in figure 6.1. The first query has unqualified column names. The second is the same query, but with fully qualified column names. A period separates the table name from the column name.

The final query shows another feature. Instead of specifying the table name, you can create a table alias to take the place of the table name in the query. The alias name follows the table name in the FROM clause. In this example, f is used as an alias for the friend table. While these features are not important in single table queries, they are useful in multi-table queries.

6.2 Joined Tables

In our friend example, splitting data into multiple tables makes little sense. However, in cases where we must record information about a variety of things, multiple tables have benefits. Consider a company that sells parts to customers through the mail. The database has to record information about many things: customers, employees, sales orders, and parts. It is obvious a single table cannot hold the different types of information in an organized manner. Therefore, we create four tables: customer, employee, salesorder, and part. However, putting information in different tables causes problems. How do we record which sales orders belong to which customers? How do we record the parts for the sales orders? How do we record which employee received the sales order? The answer is to assign unique numbers to every customer, employee, and part. When we want to record the customer in the salesorder table, we put the customer’s number in the salesorder table. When we want to record which employee took the order, we put the employee’s number in the salesorder table. When we want to record which part has been ordered, we put the part number in the salesorder table.
Breaking up the information into separate tables allows us to keep detailed information about customers, employees, and parts. It also allows us to refer to those specific entries as many times as needed by using a unique number. This is illustrated in figure 6.2.

Figure 6.2: Joining tables

People might question whether it is necessary to use separate tables. While not necessary, it is often a good idea. Without having a separate customer table, every piece of information about a customer would have to be stored in the salesorder table every time a salesorder row was added. The customer’s name, telephone number, address, and other information would have to be repeated. Any change in customer information, like a change in telephone number, would have to be performed in all places that information is stored. With a customer table, the information is stored in one place, and each salesorder points to the customer table. This is more efficient, and allows easier administration and data maintenance. The advantages of using multiple tables are:
6.3. CREATING JOINED TABLES

- Easier data modification
- Easier data lookup
- Data stored in only one place
- Less storage space required

The only time duplicate data need not be moved to a separate table is when all of these are true:

- Time required to perform a join is prohibitive.
- Data lookup is unnecessary.
- Duplicate data requires little storage space.
- Data is very unlikely to change.

The customer, employee, part, and salesorder example clearly benefits from multiple tables.¹

6.3 Creating Joined Tables

Figure 6.3 shows the SQL statements needed to create those tables.² The customer, employee, and part tables each have a column to hold their unique identification numbers. The salesorder³ table has columns to hold the customer, employee, and part numbers associated with the sales order. For the sake of simplicity, we will assume that each salesorder contains only one part number.

We have used underscores (_) to allow multiple words in column names, i.e. customer_id. This is common. You could enter the column as CustomerId, but POSTGRESQL converts all identifiers, like column and table names, to lowercase, so the actual column name becomes customerid, which is not very clear. You can’t put spaces in table or column names either unless you put double quotes (") around the name like "customer id".

Double quotes also preserve any capitalization you supply. If you decide to use this feature, you have to put double quotes around the table or column name every time you reference it. This can be cumbersome.

Keep in mind that all table and column names not protected by double quotes should be made up of only letters, numbers, and the underscore character. Each name must start with a letter, not a number. Don’t use punctuation, except underscore, in your names either. For example, address, office, and zipcode⁹ are valid names, while 2pair and my# are not.

The example also shows the existence of a column named customer_id in two tables. This is done because the two columns contain the same type of number, a customer identification number. Naming them the same clearly shows which columns join the tables together. If you wanted to use unique names, you could name the column salesorder_customer_id or sales_cust_id. This makes the column names unique, but still documents the columns to be joined.

Figure 6.4 shows the insertion of a row into the customer, employee, and part tables. It also shows the insertion of a row into the salesorder table, using the same customer, employee, and part numbers to link the salesorder row to the other rows we inserted.⁴ For simplicity, we will use only a single row per table.

¹The process of distributing data across multiple tables to prevent redundancy is called data normalization.
²In the real-world, the name columns would be much longer, perhaps char(60) or char(180). You should base the length on the longest name you may ever wish to store. I am using short names so they display properly in the examples.
³A table can not be called order. Order is a reserved keyword, for use in the ORDER BY clause. Reserved keywords are not available as table or column names.
⁴Technically, the column customer.customer_id is a primary key because it is the unique key for each customer row. The column salesorder.customer_id is a foreign key because it points to another table’s primary key. This is covered in more detail in section 6.13.
test=> 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  

CREATE  
test=> CREATE TABLE employee (  
employee_id INTEGER,  
name CHAR(30),  
hire_date DATE  
)  
CREATE  

CREATE  
test=> CREATE TABLE part (  
part_id INTEGER,  
name CHAR(30),  
cost NUMERIC(8,2)  
)  
CREATE  

CREATE  
test=> 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)  
)  
CREATE  

Figure 6.3: Creation of company tables
6.3. CREATING JOINED TABLES

```sql
test=> INSERT INTO customer VALUES (648,
  test(> 'Fleer Gearworks, Inc.',
  test(> '1-610-555-782',
  test(> '830 Winding Way',
  test(> 'Millersville',
  test(> 'AL',
  test(> '35041',
  test(> 'USA')
INSERT 18838 1

test=> INSERT INTO employee VALUES (24,
  test(> 'Lee Meyers',
  test(> '10/16/1989')
INSERT 18839 1

test=> INSERT INTO part VALUES (153,
  test(> 'Garage Door Spring',
  test(> 18.39)
INSERT 18840 1

test=> INSERT INTO salesorder VALUES(14673,
  test(> 648,
  test(> 24,
  test(> 153,
  test(> '7/19/1994',
  test(> '7/28/1994',
  test(> 18.39)
INSERT 18841 1
```

Figure 6.4: Insertion into company tables
6.4 Performing Joins

With data spread across multiple tables, an important issue is how to retrieve the data. Figure 6.5 shows how to find the customer name for a given order number. It uses two queries. The first gets the `customer_id` for order number 14673. The user then uses the returned customer identification number of 648 in the WHERE clause of the next query. That query finds the customer name record where the `customer_id` equals 648. We can call this two query approach a manual join, because the user manually took the result from the first query and placed that number into the WHERE clause of the second query.

Fortunately, relational databases can perform this join automatically. Figure 6.6 shows the same join as figure 6.5 but in a single query. This query shows all the elements necessary to perform the join of two tables:

- The two tables involved in the join are specified in the `FROM` clause.
- The two columns needed to perform the join are specified as equal in the `WHERE` clause.
- The `salesorder` table’s order number is tested in the `WHERE` clause.
- The `customer` table’s customer name is returned from the `SELECT`.

Internally, the database performs the join by:

- `salesorder.order_id = 14673`: Find that row in the `salesorder` table.
• salesorder.customer_id = customer.customer_id: From the row just found, get the \textit{customer\_id}. Find the equal \textit{customer\_id} in the \textit{customer} table.

• customer.name: Return \textit{name} from the \textit{customer} table.

You can see the database is performing the same steps as our \textit{manual join}, but much faster.

Notice that figure 6.6 qualifies each column name by prefixing it with the table name, as discussed in section 6.1. While such prefixing is optional in many cases, in this example it is required because the column \textit{customer\_id} exists in both tables mentioned in the \textit{FROM} clause, \textit{customer} and \textit{salesorder}. If this were not done, the query would generate an error: \texttt{ERROR: Column 'customer\_id' is ambiguous.}

You can perform the join in the opposite direction too. In the previous query, the order number is supplied, and the customer name is returned. In figure 6.7, the customer name is supplied, and the order number returned. I have switched the order of items in the \textit{FROM} clause and in the \textit{WHERE} clause. The ordering of items is not important in those clauses.

\section*{6.5 Three and Four Table Joins}

You can perform a three-table join as shown in figure 6.8. The first printed column is the customer name.

\begin{verbatim}
test=> SELECT salesorder.order_id
    test-> FROM salesorder, customer
    test-> WHERE customer.name = 'Fleer Gearworks, Inc.' AND
    test-> salesorder.customer_id = customer.customer_id
    test-> \g
test-> \\
order_id
    -------
        14673
(1 row)
\end{verbatim}

Figure 6.7: Finding order number for customer name

The second column is the employee name. Both columns are labeled \textit{name}. You could use \texttt{AS} to give the columns unique labels. Figure 6.9 shows a four-table join, using \texttt{AS} to make each column label unique. The four-table join matches the arrows in figure 6.2, with the arrows of the \textit{salesorder} table pointing to the other three tables.
CHAPTER 6. JOINING TABLES

```
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

Joins can be performed among tables that are only indirectly related. Suppose you wish to find employees who have taken orders for each customer. Figure 6.10 shows such a query. Notice that the query displays just the `customer` and `employee` tables. The `salesorder` table is used to join the two tables but is not displayed. The `DISTINCT` keyword is used because multiple orders taken by the same employee for the same customer would make that employee appear more than once, which was not desired. The second query uses an aggregate to return a count for each unique customer, employee pair.

Up to this point, we have had only a single row in each table. As an exercise, add additional `customer`, `employee`, and `part` rows, and add `salesorder` rows that join to these new entries. You can use figure 6.4 as an example. You can use any identification numbers you wish. Try the queries already shown in this chapter with your new data.

6.6 Additional Join Possibilities

At this point, all joins have involved the `salesorder` table in some form. Suppose we wanted to assign an employee to manage each customer account. If we add an `employee_id` column to the `customer` table, the column could store the identification number of the employee assigned to manage the customer’s account. Figure 6.11 shows how to perform the join between `customer` and `employee` tables. The first query finds the employee name assigned to manage customer number 648. The second query shows the customer names managed by employee 24. Notice the `salesorder` table is not involved in this query.

Suppose you wanted to assign an employee to be responsible for answering detailed questions about parts. Add an `employee_id` column to the `part` table, place valid employee identifiers in the column, and perform similar queries as shown in figure 6.12. Adding columns to existing tables is covered in section 13.2.

There are cases where a join could be performed with the `state` column. For example, to check state codes for validity\(^5\), a `statecode` table could be created with all valid state codes. An application could check the state code entered by the user, and report an error if the state code is not in the `statecode` table. Another example would be the need to print the full state name in queries. State names could be stored in a separate table and joined when the full state name is desired. Figure 6.13 shows an example of a `statename` table. This shows two more uses for additional tables:

- Check codes against a list of valid values, i.e. only allow valid state codes

\(^5\)The United States Postal Service has assigned a unique two-letter code to each U.S. state.
6.6. ADDITIONAL JOIN POSSIBILITIES

```
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

g
<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

(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

g
<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.

```

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

```

```
test=> 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 14673
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

CREATE TABLE statename (state CHAR(2),
    name CHAR(30))
CREATE
INSERT INTO statename VALUES ('AL', 'Alabama');
SELECT statename.name AS customer_statename
FROM customer, statename
WHERE customer.customer_id = 648 AND
    customer.state = statename.state

Figure 6.13: Statename table
6.7 Choosing a Join Key

The join key is the value used to link entries between tables. For example, in figure 6.4, 648 is the customer key, appearing in the customer table to uniquely identify the row, and in the salesorder table to refer to that specific customer row.

Some people might question whether an identification number is needed. Should the customer name be used as a join key? Using the customer name as the join key is not good because:

- Numbers are less likely to be entered incorrectly.
- Two customers with the same name would be impossible to distinguish in a join.
- If the customer name changes, all references to that name would have to change.
- Numeric joins are more efficient than long character string joins.
- Numbers require less storage than characters strings.

In the statename table, the two-letter state code is probably a good join key because:

- Two letter codes are easy for users to remember and enter.
- State codes are always unique.
- State codes do not change.
- Short two-letter codes are not significantly slower than integers in joins.
- Two-letter codes do not require significantly more storage than integers.

There are basically two choices for join keys, identification numbers and short character codes. If an item is referred to repeatedly, it is best to use a short character code as a join key. You can display this key to users and allow them to refer to customers and employees using codes. Users prefer to identify items by short, fixed-length character codes containing numbers and letters. For example, customers can be identified by six-character codes, FLE001, employees by their initials, BAW, and parts by five-character codes, E7245. Codes are easy to use and remember. In many cases, users can choose the codes, as long as they are unique.

It is possible to allow users to enter short character codes and still use identification numbers as join keys. This is done by adding a code column to the table. For the customer table, a new column called code can be added to hold the customer code. When the user enters a customer code, the query can find the customer id assigned to the customer code, and use that customer id in joins with other tables. Figure 6.14 shows a query using a customer code to find all order numbers for that customer.

```
test=> SELECT order_id
    test-> FROM customer, salesorder
    test-> WHERE customer.code = 'FLE001' AND
    test-> customer.customer_id = salesorder.customer_id
```

Figure 6.14: Using a customer code

In some cases, identification numbers are fine and codes unnecessary:
• Items with short lifespans, i.e. order numbers
• Items without appropriate codes, i.e. payroll batch numbers
• Items used internally and not referenced by users

Defining codes for such values would be useless. It is better to allow the database to assign a unique number to each item. The next chapter covers database support for assigning unique identifiers.

There is no universal rule about when to choose codes or identification numbers. U.S. states are clearly better keyed on codes, because there are only 50 U.S. states, the codes are short, unique, and well known by most users. At the other extreme, order numbers are best used without codes because there are too many of them and codes would be of little use.

6.8 One-to-Many Joins

Up to this point, when two tables were joined, one row in the first table matched exactly one row in the second table, making the joins one-to-one joins. Imagine if there were more than one salesorder row for a customer id. Multiple order numbers would be printed. That would be a one-to-many join, where one customer row joins to more than one salesorder row. Suppose there were no orders made by a customer. Even though there was a valid customer row, if there were no salesorder row for that customer identification number, no rows would be returned. We could call that a one-to-none join.\textsuperscript{6}

Figure 6.15 shows an example. Because the animal table’s 507 rabbit row join to three rows in the vegetable table, the rabbit row is duplicated three times in the output. This is a one-to-many join. There is no join for the 508 cat row in vegetable table, so the 508 cat row does not appear in the output. This is an example of a one-to-none join.

6.9 Unjoined tables

When joining tables, it is necessary to join each table mentioned in the FROM clause by specifying joins in the WHERE clause. If you list a table name in the FROM clause, but fail to join it in the WHERE clause, the effect is to mark that table as unjoined. This causes it to be paired with every row in the query result. Figure 6.16 illustrates this effect using tables from figure 6.15. The SELECT does not join any column from animal to any column in vegetable, causing every value in animal to be paired with every value in vegetable. This effect is called a Cartesian product and is usually not intended. When a query returns many more rows than expected, look for an unjoined table in the query.

6.10 Table Aliases and Self-Joins

In section 6.1, you saw how to refer to specific tables in the FROM clause using a shorter name. Figure 6.17 shows a rewrite of the query in figure 6.14 using aliases. A c is used as an alias for the customer table, and s is used as an alias for the salesorder table. Table aliases are handy in these cases.

However, with table aliases, you can even join a table to itself. Such joins are called self-joins. The same table is given two different alias names. Each alias then represents a different instance of the table. This might seem like a concept of questionable utility, but it can prove useful. Figure 6.18 shows practical examples.

\textsuperscript{6} Many database servers support a special type of join called an outer join that allows non-joined data to appear in the query. Unfortunately, PostgreSQL does not support outer joins at this time.
### 6.10. Table Aliases and Self-Joins

The code examples illustrate the use of table aliases and self-joins in a database query language.

#### Figure 6.15: One-to-many join

```sql
test=> SELECT * FROM animal;
animal_id | name
-----------+-----------------
      507 | rabbit
      508 | cat

(2 rows)
```

```sql
test=> SELECT * FROM vegetable;
animal_id | name
-----------+-----------------
      507 | lettuce
      507 | carrot
      507 | nuts

(3 rows)
```

```sql
test=> SELECT *
    test-> FROM animal, vegetable
    test-> WHERE animal.animal_id = vegetable.animal_id;
animal_id | name | animal_id | name
-----------+-----------------+-----------+-----------------
      507 | rabbit | 507 | lettuce
      507 | rabbit | 507 | carrot
      507 | rabbit | 507 | nuts

(3 rows)
```

#### Figure 6.16: Unjoined tables

```sql
test=> SELECT *
    test-> FROM animal, vegetable;
animal_id | name | animal_id | name
-----------+-----------------+-----------+-----------------
      507 | rabbit | 507 | lettuce
      507 | rabbit | 507 | carrot
      507 | rabbit | 507 | nuts
      508 | cat | 507 | lettuce
      508 | cat | 507 | carrot
      508 | cat | 507 | nuts

(6 rows)
```

#### Figure 6.17: Using table aliases

```sql
test=> SELECT order_id
    test-> FROM customer c, salesorder s
    test-> WHERE c.code = 'FLE001' AND
    test-> c.customer_id = s.customer_id
```
test=> SELECT c2.name
    test-> FROM customer c, customer c2
    test-> WHERE c.customer_id = 648 AND
    test-> c.zipcode = c2.zipcode

test=> SELECT c2.name, s.order_id
    test-> FROM customer c, customer c2, salesorder s
    test-> WHERE c.customer_id = 648 AND
    test-> c.zipcode = c2.zipcode AND
    test-> c2.customer_id = s.customer_id AND
    test-> c2.customer_id <> 648
    test-> \g

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

Figure 6.18: Examples of self-joins using table aliases

The first query finds all customers in the same zipcode as customer number 648. The second finds all customers in the same zipcode as customer number 648. It then finds the order numbers placed by those customers. We have restricted the c2 table’s customer identification number to not equal 648 because we don’t want customer 648 to appear in the result. The third query goes farther by retrieving the part numbers associated with those orders.

6.11 Non-Equijoins

Equijoins are the most common type of join. They use equality comparisons (=) to join tables. Figure 6.19 shows our first non-equijoin. The first query is a non-equijoin because it uses a not-equal (<> or !=) comparison to perform the join. It returns all customers not in the same country as customer number 648. The second query uses less-than (<) to perform the join. Instead of finding equal values to join, all rows greater-than the column’s value are joined. The query returns the all employees hired after employee number 24. The third query uses greater-than (>) in a similar way. The query returns all parts that cost less than part number 153. Non-equijoins are not used very often, but there are certain queries that can only be performed using non-equijoins.

6.12 Ordering Multiple Parts

Our customer, employee, part, and salesorder example has a serious limitation. It allows only one part per order. In the real world, this would never be acceptable. Having covered many complex join topics in this chapter, a more complete database layout can be created to allow multiple parts per order.
6.12. ORDERING MULTIPLE PARTS

The following SQL queries demonstrate ordering multiple parts:

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

```sql
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
```

```sql
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

Figure 6.20 shows a new version of the salesorder table. Notice that the part_id column has been removed. The customer, employee, and part tables remain unchanged.

```sql
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

Figure 6.21 shows a new table, orderpart. This table is needed because the original salesorder table could hold only one part number per order. Instead of putting the part_id in the salesorder table, the orderpart table will hold one row for each part number ordered. If five part numbers are in order number 15398, there will be five rows in the orderpart table with order_id equal to 15398.

We have also added a quantity column. If someone orders seven of the same part number, we put only one row in the orderpart table, but set the quantity field equal to 7. We have used DEFAULT to set the quantity to one if no quantity is supplied.

Notice there is no price field in the orderpart table. This is because the price is stored in the part table. Anytime the price is needed, a join is performed to get the price. This allows a part’s price to be changed in
one place, and all references to it automatically updated.\footnote{In our example, changing \texttt{part.price} would change the price on previous orders of the part. This would be inaccurate. In the real-world, there would have to be a \texttt{partprice} table to store the part number, price, and effective date for the price.}

\begin{verbatim}
  test=> CREATE TABLE orderpart(
  test(> order_id INTEGER,
  test(> part_id INTEGER,
  test(> quantity INTEGER DEFAULT 1
  test(> )

CREATE
\end{verbatim}

This new table layout illustrates the \emph{master / detail} use of tables. The \texttt{salesorder} table is the \emph{master} table because it holds information common to each order, such as customer and employee identifiers, and order date. The \texttt{orderpart} table is the \emph{detail} table because it contains the specific parts making up the order. Master/detail tables are a common use of multiple tables.

Figure 6.22 shows a variety of queries using the new \texttt{orderpart} table. The queries are of increasing complexity. The first query already contains the order number of interest, so there is no reason to use the \texttt{salesorder} table. It goes directly to the \texttt{orderpart} table to find the parts making up the order, and joins to the \texttt{part} table for part descriptions. The second query does not have the order number. It only has the customer id and order date. It must use the \texttt{salesorder} table to find the order number, and then join to the \texttt{orderpart} and \texttt{part} tables to get order quantities and part information. The third query does not have the customer id, but instead must join to the customer table to get the \texttt{customer_id} for use with the other tables. Notice each query displays more columns to the user. The final query computes the total cost of the order. It uses an aggregate to \texttt{SUM} cost times (*) quantity for each part in the order.

\subsection{6.13 Primary and Foreign Keys}

A join is performed by comparing two columns, like \texttt{customer.customer_id} and \texttt{salesorder.customer_id}. \texttt{Customer.customer_id} is called a \emph{primary key} because it is the unique (primary) identifier for the \texttt{customer} table. \texttt{Salesorder.customer_id} is called a \emph{foreign key} because it holds a key to another (foreign) table.

\subsection{6.14 Summary}

Previous chapters covered query tasks. This chapter dealt with technique — the technique of creating an orderly data layout using multiple tables. Acquiring this skill takes practice. Expect to redesign your first table layout many times as you improve it.

Good data layout can make your job easier. Bad data layout can make queries a nightmare. As you create your first real-world tables, you will soon learn to identify good and bad data designs. Continually review your table structures. Refer to this chapter again for ideas. Don’t be afraid to redesign everything. Redesign is hard, but when it is done properly, queries are much easier to craft.

Relational databases excel in their ability to relate and compare data. Tables can be joined and analyzed in ways never anticipated. With good data layout and the power of SQL, you can retrieve an unlimited amount of information from your database.
6.14. SUMMARY

```sql
-- 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 `orderpart` table
Chapter 7

Numbering Rows

Unique identification numbers and short character codes allow reference to specific rows in a table. They were used extensively in the previous chapter. The customer table had a customer_id column that held a unique identification number for each customer. The employee and part tables had similar uniquely numbered columns. Those columns were important for joins to those tables.

While unique character codes must be supplied by users, unique row numbers can be generated automatically using two methods. This chapter shows how to uniquely number rows in PostgreSQL.

7.1 Object Identification Numbers (OIDs)

Every row in PostgreSQL is assigned a unique, normally invisible number called an object identification number or OID. When the software is initialized with initdb, 1 a counter is created and set to approximately seventeen-thousand.2 The counter is used to uniquely number every row. Databases can be created and destroyed, but the counter continues to increase. The counter is used by all databases, so object identification numbers are always unique. No two rows in any table or in any database have the same object id.3

You have seen object identification numbers already. Object identification numbers are displayed after every INSERT statement. If you look back at figure 3.4 on page 12, you will see the line INSERT 18720 1. INSERT is the command that was executed, 18720 is the object identification number assigned to the inserted row, and 1 is the number of rows inserted. A similar line appears after every INSERT statement. Figure 6.4 on page 49 shows sequential object identification numbers assigned by consecutive INSERT statements.

Normally, a row’s object identification number is displayed only by INSERT queries. However, if the OID is specified by a non-INSERT query, it will be displayed, as shown in figure 7.1. The SELECT has accessed the normally invisible OID column. The OID displayed by the INSERT and the OID displayed by the SELECT are the same.

Even though no OID column is mentioned in CREATE TABLE statements, every PostgreSQL table has an invisible column called OID. The column only appears if you specifically access it.4 The query SELECT * FROM table_name does not display the OID column. SELECT OID, * FROM table_name will display it.

Object identification numbers can be used as primary and foreign key values in joins. Since every row has a unique object id, there is no need for a separate column to hold the row’s unique number.

For example, in the previous chapter there was a column called customer.customer_id. This column held the customer number. It uniquely identified each row. However, we could have used the row’s object

---

1See section B.3 for a description of initdb.
2Values less than this are reserved for internal use.
3Technically, OID’s are unique among all databases sharing a common data directory tree.
4There are several other invisible columns. The PostgreSQL manuals cover their meaning and use.
test=> CREATE TABLE oidtest(age INTEGER);
CREATE

test=> INSERT INTO oidtest VALUES (7);
INSERT 18697 1

test=> SELECT oid, age from oidtest;

oid | age
-------+-----
18697 | 7

Figure 7.1: OID test

identification number as the unique number for each row. Then, there would be no need to create the column customer.customer_id. Customer.oid would be the unique customer number.

With this change, a similar change would be needed in the salesorder table. We would rename salesorder.customer_id to salesorder.customer_oid because the column now refers to an OID. The column type should be changed also. Salesorder.customer_id was defined as type INTEGER. The new salesorder.customer_oid column would hold the OID of the customer who made the order. For this reason, we would change the column type from INTEGER to OID. Figure 7.2 shows a new version of the salesorder table using each row's OID as a join key.

```
test=> 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

A column of type OID is similar to an INTEGER column, but defining it as type OID documents that the column holds OID values. Don’t confuse a column of type OID with a the column named OID. Every row has a normally invisible column named OID. A row can have zero, one, or more user-defined columns of type OID.

A column of type OID is not automatically assigned any special value from the database. Only the column named OID is specially assigned during INSERT.

Also, the order_id column in the salesorder table could be eliminated. The salesorder.oid column could represent the unique order number.

### 7.2 Object Identification Number Limitations

Object identification numbers have some limitations:

- Object identification numbers are normally not sequential in a table.
- Once assigned during INSERT, object identification numbers cannot be modified.
- By default, backups do not record the normally invisible column named OID.
The global nature of object identification assignment means most OIDs in a table are not sequential. For example, if you insert a customer today, and another one tomorrow, the two customers will not get sequential OIDs. The two customer OIDs could differ by thousands. This is because inserts into other tables between the two customer inserts increment the object counter. If the OID is not visible to users, this is not a problem. Non-sequential numbering does not affect query processing. However, if users see and enter these numbers, it might seem strange customer identification numbers are not sequential and have large gaps in numbering.

An OID is assigned to every row during INSERT. UPDATE cannot modify the system-generated OID of a row.

When performing database backups, the system-generated OID of each row is normally not backed up. A flag must be added to enable the backup of OIDs. See section 18.2 for details.

7.3 Sequences

POSTGRESQL has another way of uniquely numbering rows. They are called sequences. Sequences are named counters created by users. After creation, the sequence can be assigned to a table as a column default. Using sequences, unique numbers can be automatically assigned during INSERT.

The advantage of sequences is that there are no gaps in numeric assignment, as happens with OIDs. Sequences are ideal as user-visible identification numbers. If a customer is created today, and another tomorrow, the two customers will have sequential numbers. This is because no other table shares the sequence counter.

Sequence numbers are unique only within a single table. For example, if a table has a unique row numbered 937, another table might have a row numbered 937 also, assigned by a different sequence counter.

7.4 Creating Sequences

Sequences are not created automatically like OIDs. You must create sequences using the CREATE SEQUENCE command. Three functions control the sequence counter. They are listed in table 7.1.

<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>

Figure 7.3 shows an example of sequence creation and sequence function usage. The first command creates the sequence. Then, various sequence functions are called. Note the SELECTs do not have a FROM clause. Sequence function calls are not directly tied to any table. This figure shows that:

- `nextval()` returns ever increasing values
- `currval()` returns the previous sequence value without incrementing
- `setval()` sets the sequence counter to a new value

`currval()` returns the sequence number assigned by a prior `nextval()` call in the current session. It is not affected by `nextval()` calls of other users. This allows reliable retrieval of `nextval()`-assigned values in latter queries.

---

5 This is not completely true. Gaps can occur if a query is assigned a sequence number as part of an aborted transaction. See section 10.2 for a description of aborted transactions.
test=> CREATE SEQUENCE functest_seq;
CREATE

CREATE

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


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


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


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


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

Figure 7.3: Examples of sequence function use
7.5 Using Sequences to Number Rows

Configuring a sequence to uniquely number rows involves several steps:

- Create the sequence.
- Create the table, defining \texttt{nextval()} as the column default.
- During \texttt{INSERT}, do not supply a value for the sequenced column, or use \texttt{nextval()}.

Figure 7.4 shows the use of a sequence for unique row numbering in the customer table. The first statement creates a sequence counter named \texttt{customer_seq}. The second command creates the \texttt{customer} table, and defines \texttt{nextval('customer_seq') as the default for the customer_id column}. The first \texttt{INSERT} manually supplies the sequence value for the column. The \texttt{nextval('customer_seq')} function call will return the next available sequence number, and increment the sequence counter. The second and third \texttt{INSERTs} allow the \texttt{nextval('customer_seq')} DEFAULT be used for the \texttt{customer_id} column. Remember, a column’s DEFAULT value is used only when a value is not supplied by an \texttt{INSERT} statement. This is covered in section 4.4. The \texttt{SELECT} shows the sequence has sequentially numbered the customer rows.

7.6 Serial Column Type

There is an easier way to use sequences. If you define a column of type SERIAL, a sequence will be automatically created, and a proper DEFAULT assigned to the column. Figure 7.5 shows an example of this. Do not be concerned about the \texttt{index} lines in the figure. Indexing is covered in section 11.1.
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 1 row

test=> SELECT * FROM customer;
customer_id | name
-------------|-----------------------------------------------
1 | Car Wash

(1 row)

Figure 7.5: Customer table using SERIAL
7.7 Manually Numbering Rows

Some people wonder why OIDs and sequences are needed. Why can’t a database user just find the highest number in use, add one, and use that as the new unique row number? There are several reasons why OIDs and sequences are preferred:

- Performance
- Concurrency
- Standardization

First, it is usually slow to scan all numbers currently in use to find the next available number. Using a counter in a separate location is faster. Second, there is the problem of concurrency. If one user gets the highest number, and another user is looking for the highest number at the same time, the two users might choose the same next available highest number. Of course, if this happens, the number would not be unique. Such concurrency problems do not occur when using OIDs or sequences. Third, it is more reliable to use database-supplied unique number generation than to generate unique numbers manually.

7.8 Summary

Both OIDs and sequences allow the automatic unique numbering of rows. OIDs are always created and numbered, while sequences require more work to configure. Both are valuable tools for uniquely numbering rows.
Chapter 8

Combining SELECTs

This book has covered various topics like regular expressions, aggregates, and joins. These are powerful SQL features that allow the construction of complex queries. However, in some cases, even these tools are not enough. This chapter shows how SELECTs can be combined to create even more powerful queries.

8.1 UNION, EXCEPT, INTERSECT Clauses

Sometimes a single SELECT statement can not produce the desired result. UNION, EXCEPT, and INTERSECT allow SELECT statements to be chained together, allowing more complex queries to be constructed.

For example, suppose we want to output the friend table’s firstname and lastname in the same column. Normally two queries would be required, one for each column. However, with UNION, the output of two SELECTs can be combined in a single query, as shown in figure 8.1. The query combines two columns into a single output column.

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

Figure 8.1: Combining two columns with UNION
UNION allows an unlimited number of SELECT statements to be combined to produce a single result. Each SELECT must return the same number of columns. If the first SELECT returns two columns, the other SELECTs must return two columns. The column types must be similar also. If the first SELECT returns an INTEGER value in the first column, the other SELECTs must return an INTEGER in their first columns.

With UNION, an ORDER BY clause can be used only at the end of the last SELECT. The ordering applies to the output of the entire query. In the previous figure 8.1, the ORDER BY clause specifies the ordering column by number. Instead of a number, we could use ORDER BY firstname because UNION's output labels are the same as the column labels of the first SELECT.

As another example, suppose we have two tables that hold information about various animals. One table holds information about aquatic animals, and another contains information about terrestrial animals. Two separate tables are used because each table records information specific to a class of animal. The aquatic_animal table holds information meaningful only for aquatic animals, like preferred water temperature. The terrestrial_animal table holds information meaningful only for terrestrial animals, like running speed. We could have put the animals in the same table, but it was clearer to keep them separate. In most cases, we deal with the animal types separately.

However, suppose we need to list all the animals, both aquatic and terrestrial. There is no single SELECT that will show animals from both tables. We can not join the tables because there is no join key. Joining is not desired. We want rows from the terrestrial_animal table and the aquatic_animal table output together in a single column. Figure 8.2 shows how these two tables can be combined with UNION.

```
test=> INSERT INTO terrestrial_animal (name) VALUES ('tiger');
  INSERT 19122 1

test=> INSERT INTO aquatic_animal (name) VALUES ('swordfish');
  INSERT 19123 1

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

  name
  ---------
  swordfish
  tiger

(2 rows)
```

Figure 8.2: Combining two tables with UNION

By default, UNION prevents duplicate rows from being displayed. For example, figure 8.3 inserts penguin into both tables. However, penguin is not duplicated in the output. To preserve duplicates, you must use UNION ALL, as shown in figure 8.4.

You can do more complex things when chaining SELECTs. EXCEPT allows all rows to be returned from the first SELECT except rows that also appear in the second SELECT. Figure 8.5 shows an EXCEPT query. While the aquatic_animal table contains swordfish and penguin, the query returns only swordfish. Penguin is excluded from the output because it is returned by the second query. While UNION adds rows to the first SELECT, EXCEPT subtracts rows from the first SELECT.

INTERSECT returns only rows generated by all SELECTs. Figure 8.6 uses INTERSECT and displays only penguin. While several animals are returned by the two SELECTs, only penguin is returned by both SELECTs.

Any number of SELECTs can be linked using these methods. The previous examples allowed multiple
8.1. UNION, EXCEPT, INTERSECT CLAUSES

```
INSERT INTO aquatic_animal (name) VALUES ('penguin');
INSERT 19124 1
INSERT INTO terrestrial_animal (name) VALUES ('penguin');
INSERT 19125 1

SELECT name
FROM aquatic_animal
UNION
SELECT name
FROM terrestrial_animal;
```

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

Figure 8.3: UNION with duplicates

```
SELECT name
FROM aquatic_animal
UNION ALL
SELECT name
FROM terrestrial_animal;
```

```
swordfish
penguin
tiger
(4 rows)
```

Figure 8.4: UNION ALL with duplicates

```
SELECT name
FROM aquatic_animal
EXCEPT
SELECT name
FROM terrestrial_animal;
```

```
swordfish
(1 row)
```

Figure 8.5: EXCEPT restricts output from the first SELECT
test=> SELECT name
      FROM aquatic_animal
      INTERSECT
      SELECT name
      FROM terrestrial_animal;
      name
      --------------------------------
penguin
      (1 row)

Figure 8.6: INTERSECT returns only duplicated rows

columns to populate a single result column. Without the ability to chain SELECTs using UNION, EXCEPT, and INTERSECT, it would be impossible to generate the desired results. SELECT chaining can do other sophisticated things, like joining a column to one table in the first SELECT, and joining the same column to another table in the second SELECT.

8.2 Subqueries

Subqueries are similar to SELECT chaining. While SELECT chaining combines SELECTs on the same level in a query, subqueries allow SELECTs to be embedded inside other queries. Subqueries can:

- Take the place of a constant in a comparison
- Take the place of a constant yet vary based on the row being processed
- Return a list of values for use in a comparison

Subqueries as Constants

A subquery, also called a subselect, can take the place of a constant in a query. Though a constant never changes, a subquery’s value is recomputed every time the query is executed.

As an example, we will use the friend table shown in figure 8.7. Suppose we want to find friends who are not in the same state as Dick Gleason. We could place his state in the query using the constant string ’NJ’, but if he moves to another state, the query would have to be changed. Using his state column is more reliable.

The figure shows two ways to generate the correct result. One query uses a self-join to do the comparison to Dick Gleason’s state. The last query uses a subquery which returns his state as ’NJ’. This value is used by the upper query. The subquery has taken the place of a constant. Unlike a constant, the value is recomputed every time the query is executed.

Though we have used table aliases in the subquery for clarity, they are not required. A column name with no table specification is automatically paired with a table in the current subquery. If no matching table is found in the current subquery, higher parts of the query are searched for a match. State, firstname, and lastname in the subquery refer to the instance of the friend table in the subquery. The same column names in the upper query automatically refer to the friend instance in the upper query. If a column name matches two tables in the same subquery, an error is returned indicating the column is ambiguous.

Subqueries can eliminate table joins also. For example, consider the parts order company in figures 6.3 and 6.4 on page 48. To find the customer name for order number 14673, we join the salesorder and customer
8.2. SUBQUERIES

```sql
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>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)

```sql
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)

```sql
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
CHAPTER 8. COMBINING SELECTS

tables. This is shown as the first query in figure 8.8. The second query does not have a join, but instead gets

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

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

Figure 8.8: Subqueries can replace some joins

of the customer_id from a subquery. In general, if a table is involved in only one join, and no columns from the table appear in the query result, the join can be eliminated and the table moved to a subquery.

In this example, we have specified salesorder.customer_id and customer.customer_id to clearly indicate the tables being referenced. However, this is not required. We could have used only customer_id in both places. PostgreSQL finds the first table in the same subquery or higher that contains a matching column name.

Subqueries can be used anywhere a computed value is needed. A subquery has its own FROM and WHERE clauses. It can have its own aggregates, GROUP BY, and HAVING. Its only interaction with the upper query is the value it returns. This allows sophisticated comparisons that would be difficult if the subquery’s clauses had to be combined with those of the upper query.

Subqueries as Correlated Values

While subqueries can act as constants in queries, subqueries can also act as correlated values. Correlated values vary based on the row being processed. A normal subquery is evaluated once and its value used by the upper query. In a correlated subquery, the subquery is evaluated repeatedly for every row processed.

For example, suppose you want to know the name of your oldest friend in each state. You can do this with HAVING and table aliases, as shown in the first query of figure 8.9. Another way is to execute a subquery for each row which finds the maximum age for that state. If the maximum age equals the age of the current row, the row is output, as shown in the second query. The query references the friend table two times, using aliases f1 and f2. The upper query uses f1. The subquery uses f2. The correlating specification is WHERE f1.state = f2.state. This makes it a correlated subquery because the subquery references a column from the upper query. Such a subquery cannot be evaluated once and the same result used for all rows. It must be evaluated for every row because the upper column value can change.
8.2. SUBQUERIES

<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
Subqueries as List of Values

The previous subqueries returned one row of data to the upper query. If any of the previous subqueries returned more than one row, an error would be generated: ERROR: More than one tuple returned by a subselect used as an expression. However, it is possible to use subqueries returning multiple rows.

Normal comparison operators like equal and less-than expect a single value on the left and on the right. Two special comparisons, IN and NOT IN, allow multiple values to appear on the right-hand side. For example, the test col IN (1,2,3,4) compares col against four values. If col equals any of the four values, the comparison will return true. The test col NOT IN (1,2,3,4) will return true if col does not equal any of the four values.

An unlimited number of values can be specified on the right-hand side of an IN or NOT IN comparison. In addition, instead of constants, a subquery can be placed on the right-hand side. The subquery can return multiple rows. The subquery is evaluated, and its output used like a list of constant values.

Suppose we want all employees who took sales orders on a certain date. We could perform the query two ways. We could join the employee and salesorder tables, as shown in the first query of figure 8.10. The second

```sql
SELECT DISTINCT employee.name
FROM employee, salesorder
WHERE employee.employee_id = salesorder.employee_id AND
 salesorder.order_date = '7/19/1994';
```

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lee Meyers</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lee Meyers</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

Figure 8.10: Employees who took orders

query uses a subquery. The subquery is evaluated, and generates a list of values used by IN to perform the comparison. The subquery is possible because the salesorder table is involved in a single join, and no columns from the salesorder table are returned by the query.

A NOT IN comparison returns true if a column’s value is not found. For example, suppose we want to see all customers who have never ordered a product. We need to find the customers who have no sales orders. This cannot be done with a join. We need an anti-join, because we want to find all customer rows that do not join to any salesorder row. Figure 8.11 shows the query. The subquery returns a list of customer_ids representing all customers who have placed orders. The upper query returns all customer names where the customer_id does not appear in the subquery output.
8.2. SUBQUERIES

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

name
------
(0 rows)
```

Figure 8.11: Customers who have no orders

**NOT IN and Subqueries with NULLs**

If a NOT IN subquery returns a NULL row, the NOT IN comparison always returns *false*. This is because NOT IN requires the upper column to be not equal to *every* value returned by the subquery. Every inequality comparison must return true. However, all comparisons with NULL return false, even inequality comparisons, so NOT IN returns false. NULL comparisons are covered in section 4.3.

We can prevent NULLs from reaching the upper query by adding `IS NOT NULL` to the subquery. As an example, in figure 8.11, if there were any NULL `customer_id` values, the query would return no rows. We can prevent this by adding `WHERE customer_id IS NOT NULL` to the subquery.

An IN subquery does not have this problem with NULLs because IN will return true if it finds any true equality comparison. NOT IN must find *all* inequality comparison to be true.

There is another way to analyze subqueries returning NULLs. Suppose a subquery returns three rows, 1, 2, and NULL. The test `uppercol1 NOT IN (subquery)` expands to `uppercol1 NOT IN (1,2,NULL)`. This further expands to `uppercol1 <> 1 AND uppercol1 <> 2 AND uppercol1 <> NULL`. The last comparison with NULL is false because all comparisons with NULL are false, even *not equal* comparisons. AND returns false if any of its comparisons return false. Therefore, the NOT IN comparison returns false.

If the test used IN, the comparison would be `uppercol1 = 1 OR uppercol1 = 2 OR uppercol1 = NULL`. While the last comparison is false, OR will return true if *any* of the comparisons is true. It does not require them *all* to be true like AND.

**Subqueries Returning Multiple Columns**

Most subqueries return a single column to the upper query. However, it is possible to handle subqueries returning more than one column. For example, the test `WHERE (7, 3) IN (SELECT col1, col2 FROM subtable)` returns true if the subquery returns a row with 7 in the first column, and 3 in the second column. The test `WHERE (uppercol1, uppercol2) IN (SELECT col1, col2 FROM subtable)` performs equality comparisons between the upper two columns and the subquery’s two columns. This allows multiple columns in the upper query to be compared with multiple columns in the subquery. Of course, the number of values specified on the left of IN or NOT IN must be the same as the number of columns returned by the subquery.

**ANY, ALL, and EXISTS Clauses**

IN and NOT IN are special cases of the more generic subquery clauses ANY, ALL, and EXISTS. ANY will return true if the comparison operator is true for any value in the subquery. The test `col1 < ANY(5,7,9)` returns true if `col1` is less than *any* of the three values. ALL requires *all* subquery values to compare as true, so `col1 <
ALL(5,7,9) returns true if col is less than all three values. IN is the same as = ANY, and NOT IN is the same as <> ALL.

Normally, you can use operators like equal and greater-than only with subqueries returning one row. With ANY and ALL, comparisons can be made with subqueries returning multiple rows. They allow you to specify whether any or all of the subquery values must compare as true.

EXISTS returns true if the subquery returns any rows, and NOT EXISTS returns true if the subquery returns no rows. By using a correlated subquery, EXISTS allows complex comparisons of upper query values inside the subquery. For example, two upper query variables can be compared in the subquery’s WHERE clause. EXISTS and NOT EXISTS do not allow values on their left-hand sides, so it does not matter which columns are returned by the subquery.

For example, figure 8.12 shows the IN subquery from figure 8.10 and the query rewritten using ANY and EXISTS. Notice the EXISTS subquery uses a correlated subquery to join the employee_id columns of the two tables. Figure 8.13 shows the NOT IN query from figure 8.11 and the query rewritten using ALL and NOT EXISTS.

**Summary**

A subquery can represent a fixed value, a correlated value, or a list of values. An unlimited number of subqueries can be used. Subqueries can be nested inside other subqueries.
8.3 OUTER JOINS

An outer join is like a normal join, except special handling is performed to prevent unjoined rows from being suppressed in the result. For example, in the join `customer.customer_id = salesorder.customer_id`, only customers that have sales orders appear in the result. If a customer has no sales orders, he is suppressed from the output. However, if the `salesorder` table is used in an outer join, the result will include all customers. The `customer` and `salesorder` tables are joined and output, plus one row for every unjoined `customer` is output. In the query, any reference to `salesorders` columns for these unjoined `customers` returns `NULL`.

As of PostgreSQL 7.0, outer joins are not supported. They can be simulated using subqueries and `UNION ALL`, as shown in figure 8.14. The first `SELECT` performs a normal join of the `customer` and `salesorder` tables. The second `SELECT` displays all customer who have no orders, and displays `NULL` as their order number.

8.4 Subqueries in Non-SELECT Queries

Subqueries can be used in `UPDATE` and `DELETE` statements also. Figure 8.15 shows two examples. The first query deletes all customers who have no sales orders. The second query sets the `ship_date` equal to 

'11/16/96' for all orders made by customer Fleer Gearworks, Inc. The numbers after `DELETE` and `UPDATE` indicate the number of rows affected by the queries.


Figure 8.14: Simulating outer joins

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

Figure 8.15: Subqueries with UPDATE and DELETE
8.5 UPDATE with FROM

UPDATE can have an optional FROM clause, which allows joins to other tables. The FROM clause also allows the use of columns from other tables in the SET clause. With this capability, columns can be updated with data from other tables.

Suppose we want to update the salesorder table’s order_date column. For some reason, some orders exist in the system that have order_dates earlier than the hire_date of the employee who recorded the sale. For these rows, we wish to set the order_date equal to the employee’s hire_date. Figure 8.16 shows this query.

```sql
test=> UPDATE salesorder
  test-> SET order_date = employee.hire_date
  test-> FROM employee
  test-> WHERE salesorder.employee_id = employee.employee_id AND
            salesorder.order_date < employee.hire_date;
UPDATE 0
```

Figure 8.16: UPDATE the order_date

The FROM clause allows the use of the employee table in the WHERE and SET clauses. While UPDATE can use subqueries to control which data rows are updated, only the FROM clause allows columns from other tables to be used in the SET clause.

8.6 Inserting Data Using SELECT

Up to this point, every INSERT statement has inserted a single row. Each INSERT had a VALUES clause listing the constants to be inserted. However, there is a second form of the INSERT statement. It allows the output of a SELECT to be used to insert values into a table.

Suppose we wish to add all our friends from the friend table to the customer table. Figure 8.17 shows that instead of a VALUES clause, INSERT can use the output of SELECT to insert data into the table. Each column

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

Figure 8.17: Using SELECT with INSERT

of the SELECT matches a receiving column in the INSERT. Column names and character string constants can be used in the SELECT output. The line INSERT 0 6 shows six rows were inserted into the customer table. A zero object identifier is returned because more than one row was inserted.

Inserting into the customer name column presents an interesting challenge. The friend table stores first and last names in separate columns. The customer table has a single name column. The only solution is to combine the firstname and lastname columns, with a space between them. For example, a firstname of 'Dean' and lastname of 'Yeager' must be inserted into customer.name as 'Dean Yeager'. This is possible using the || operator. It allows character strings to be joined together to form a single string, a process called concatenation. In this example, firstname, space (' '), and lastname are joined using ||.
8.7 Creating Tables Using SELECT

In addition to inserting into existing tables, SELECT has an INTO clause that can create a table and place all its output into the new table. For example, suppose we want to create a new table called newfriend just like our friend table, but without an age column. This is easily done with the query in figure 8.18. The SELECT…INTO

```
SELECT firstname, lastname, city, state
INTO newfriend
FROM friend;
```

The other queries in the figure show the new table’s structure and contents.

Figure 8.18: Table creation with SELECT

- Creates a table called newfriend
- Uses SELECT’s column labels to name the columns of the new table
- Uses SELECT’s column types as the column types of the new table

SELECT…INTO is CREATE TABLE and SELECT combined in a single statement. The AS clause can be used to change the column labels and thus control the column names in the new table. The other queries in the figure show the new table’s structure and contents.

8.8 Summary

This section introduced a variety of powerful SQL features. It takes experience to know when and how to use them.

With so many capabilities, it can be difficult to construct queries. Your goal should be to make queries as clear as possible.
Chapter 9

Data Types

Data types have been used in previous chapters. This chapter covers them in detail.

9.1 Purpose of Data Types

It is tempting to think databases would be easier to use if there was only one data type – a type that could hold any type of information: numbers, character strings, or dates. While a single data type would certainly make table creation simpler, there are definite advantages to having different data types:

**Consistent Results** Columns of a uniform type produce consistent results. Displaying, ordering, aggregates, and joins deliver consistent results. There is no conflict about how different types are compared or displayed. Selecting from an `INTEGER` column always yields `INTEGER` values.

**Data Validation** Columns of a uniform type accept only properly formatted data. Invalid data is rejected. A column of type `INTEGER` will reject a `DATE` value.

**Compact Storage** Columns of a uniform type are stored more compactly.

**Performance** Columns of a uniform type are processed more quickly.

For these reasons, each column in a relational database can hold only one type of data. Data types can not be mixed within a column.

This limitation can cause some difficulties. For example, in our `friend` table, there is an `age` column of type `INTEGER`. Only whole numbers can be placed in that column. The values "I will ask for his age soon" or "She will not tell me her age" can not be placed in that column. NULL can represent "I don't know her age." The solution is to create an `age_comments` column of type `CHAR()` to hold comments which can not be placed in the `age` field.

9.2 Installed Types

`POSTGRESQL` supports a large number of data types, as shown in table 9.1. Except for the numeric types, all entered values must be surrounded by single quotes.
<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>Numeric</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>BOOL</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
### 9.2. INSTALLED TYPES

**Character String**

Character string types are the most commonly used data types. They can hold any sequence of letters, digits, punctuation, and other valid ASCII characters. Typical character strings are names, descriptions, and mailing addresses. Any value can be stored in a character string. However, character strings should be used only when other data types are inappropriate. The other types provide data validation, more compact storage, and better performance.

There are three character string data types: TEXT, VARCHAR(length), and CHAR(length). TEXT allows an unlimited number of characters to be stored. VARCHAR(length) limits the length of the field to length characters. Both TEXT and VARCHAR() store only the number of characters in the string. CHAR(length) is similar to VARCHAR(), except it always stores exactly length characters. It pads the value with trailing spaces to the specified length. It provides slightly faster access than TEXT or VARCHAR().

Understanding why character string types are different from other data types can be difficult. For example, you can store 763 as a character string. In this case, you are storing the symbols 7, 6, and 3, not the numeric value 763. You can’t add a number to the character string 763 because it doesn’t make sense to add a number to three symbols. Similarly, the character string 3/8/1992 is eight symbols starting with 3 and ending with 2. If you store it in a character string data type, it is not a date. You can not sort it with other values and expect them to be in chronological order. The string 1/4/1998 is less than 3/6/1992 when these are sorted as character strings because 1 is less than 3.

This illustrates why the other data types are valuable. The other types have a predefined format for their data, and can do more appropriate operations on the stored information.

Still, there is nothing wrong with storing numbers or dates in character strings when appropriate. The street address 100 Maple Avenue is best stored in a character string type, even though a number is part of the street address. It makes no sense to store the street number in a separate INTEGER field. Also, part numbers like G8223-9 must be stored in character strings because of the G and dash. In fact, part numbers that are always five digits, like 32911 or 00413 should be stored in character strings too. They are not real numbers, but symbols. Leading zeros can not be displayed by INTEGER fields, but are easily displayed in character strings.

**Numeric**

Numeric types allow the storage of numbers. The numeric types are: INTEGER, INT2, INT8, OID, NUMERIC(), FLOAT, and FLOAT4.

INTEGER, INT2, and INT8 store whole numbers of various ranges. Larger ranges require more storage, i.e. INT8 requires twice the storage of INTEGER, and is slower.

OID is used to store PostgreSQL object identifiers. While INTEGER could be used for this purpose, OID helps document the meaning of the value stored in the column.

NUMERIC(precision, decimal) allows user-defined digits of precision, rounded to decimal places. This type is slower than the other numeric types.

FLOAT and FLOAT4 allow storage of floating-point values. Numbers are stored using fifteen (FLOAT) or six (FLOAT4) digits of precision. The location of the decimal point is stored separately, so large values like 4.78145e+32 can be represented. FLOAT and FLOAT4 are fast and have compact storage, but can produce imprecise rounding during computations. When complete accuracy of floating point values is required, NUMERIC() should be used.

---

1. ASCII is the standard encoding used to map symbols to values. For example, uppercase A maps to the internal value 65. Lowercase a maps to the value 97. Period (.) maps to 46. Space maps to 32.
CHAPTER 9. DATA TYPES

Temporal

Temporal types allow storage of date, time, and time interval information. While these can be stored in character strings, it is better to use temporal types, for reasons outlined earlier in this chapter.

The four temporal types are: DATE, TIME, TIMESTAMP, and INTERVAL. DATE allows storage of a single date consisting of year, month, and day. The format used to input and display dates is controlled by the DATESTYLE setting covered in section 4.14 on page 33. TIME allows storage of hour, minute, and second, separated by colons. TIMESTAMP represents storage of both date and time, i.e. 2000-12-17 17:34:29. INTERVAL represents an interval of time, like 5 hours or 7 days. INTERVAL values are often generated by subtracting two TIMESTAMP values to find the elapsed time. For example, 1996-12-15 19:00:40 minus 1996-12-8 14:00:10 results in an INTERVAL value of 7 05:00:30, which is seven days, five hours, and thirty seconds. Temporal types can also handle timezone designations.

Logical

The only logical type is BOOLEAN. A BOOLEAN field can store only true or false, and of course NULL too. You can input true as true, t, yes, y, or 1. False can be input as false, f, no, n, or 0. While true and false can be input in a variety of ways, true is always output as t and false as f.

Geometric

The geometric types allow storage of geometric primitives. The geometric types are: POINT, LSEG, PATH, BOX, CIRCLE, and POLYGON. Table 9.2 shows the geometric types and typical values.

<table>
<thead>
<tr>
<th>Types</th>
<th>Example</th>
<th>Notes</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 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>

Table 9.2: Geometric types

Network

The network types are: INET, CIDR, and MACADDR. INET allows storage of an IP address, with or without a netmask. A typical INET value with netmask is 172.20.90.150 255.255.255.0. CIDR stores IP network addresses. It allows a subnet mask to specify the size of the network segment. A typical CIDR value is 172.20.90.150/24. MACADDR stores MAC (Media Access Control) addresses. These are assigned to Ethernet network cards at the time of manufacture. A typical MACADDR value is 0:50:4:1d:f6:db.

Internal

There are a variety of types used internally. Psql’s ‘dT’ command shows all data types.
9.3 Type Conversion and Casting

In most cases, values of one type are converted to another type automatically. In rare circumstances where you need to explicitly convert one type to another, you can use a function to perform the conversion. Conversion functions have the same names as data types. To convert a value to type INTEGER, use the function called `integer()`. To convert a column `date_col` of type `DATE` to type `TEXT`, use `text(date_col)`. You can also perform type casting using double-colons or the `CAST` operator, i.e. `date_col::text` or `CAST(datecol AS TEXT)`.

9.4 Support Functions

Functions allows access to specialized routines from SQL. Functions take one or more arguments, and return a result.

Suppose you want to uppercase a value or column. There is no command for uppercase, but there is a function that will do it. POSTGRESQL has a function called `upper`. `Upper` takes a single string argument, and returns the argument in uppercase. The function call `upper(col)` calls the function `upper` with `col` as its argument, and returns `col` in uppercase. Figure 9.4 shows an example of the use of the `upper` function.

```
test=> SELECT * FROM functest;
    name
-------
   Judy
    (1 row)
test=> SELECT upper(name) FROM functest;
    upper
-------
    JUDY
    (1 row)
```
Example of a function call

There are many functions available. Table 9.3 shows the most common ones, organized by the data types they support. Psql's `\df` shows all defined functions and their arguments. Section 17.1 has information about all `psql` commands.

If you call a function with a type for which it is not defined, you will get an error, as shown in the first query of figure 9.1. In the first query, the 732 is a number, not a character string. The second query converts 732 to a character string so the length can be computed.

9.5 Support Operators

Operators are similar to functions, and are covered in section 4.13 on page 33. Table 9.4 shows the most common operators. Psql's `\do` shows all defined operators and their arguments.

All data types have the standard comparison operators `<`, `<=`, `=`, `>=`, `>`, and `<>`. Not all operator/type combinations are defined. For example, if you try to add two `DATE` values, you will get an error, as shown in the first query of figure 9.2.
<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 multi-byte 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>rpad()</td>
<td>rpad(col, len, str)</td>
<td>col padded on the right using str</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>lpad()</td>
<td>lpad(col, len, str)</td>
<td>col padded on the left using str</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() above</td>
</tr>
<tr>
<td></td>
<td>substr()</td>
<td>substr(col, pos, len)</td>
<td>col starting at position pos for length len</td>
</tr>
<tr>
<td></td>
<td>substring(...FROM...FOR...)</td>
<td>substring(col FROM pos FOR len)</td>
<td>same as substr() above</td>
</tr>
<tr>
<td></td>
<td>translate()</td>
<td>translate(col, from, to)</td>
<td>col with from changed 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>Numeric</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(log)</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>Temporal</td>
<td>date_part()</td>
<td>date_part(units, col)</td>
<td>units part of col</td>
</tr>
<tr>
<td></td>
<td>extract(...FROM...)</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>isnfinite()</td>
<td>isnfinite(col)</td>
<td>BOOLEAN indicating if 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 if 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>Geometric</td>
<td>broadcast()</td>
<td>broadcast(col)</td>
<td>broadcast address of col</td>
</tr>
<tr>
<td>Network</td>
<td>host()</td>
<td>host(col)</td>
<td>host address of col</td>
</tr>
<tr>
<td></td>
<td>netmask()</td>
<td>netmask(col)</td>
<td>netmask of col</td>
</tr>
<tr>
<td></td>
<td>masklen()</td>
<td>masklen(col)</td>
<td>mask length of col</td>
</tr>
<tr>
<td></td>
<td>network()</td>
<td>network(col)</td>
<td>network address of col</td>
</tr>
<tr>
<td>NULL</td>
<td>nullif()</td>
<td>nullif(col1, col2)</td>
<td>return NULL if col1 equals col2, else return col2</td>
</tr>
<tr>
<td></td>
<td>coalesce()</td>
<td>coalesce(col1, col2, …)</td>
<td>return first non-NULL argument</td>
</tr>
</tbody>
</table>

Table 9.3: Common functions
9.5. SUPPORT OPERATORS

```
test=> SELECT length(755);
ERROR: Function 'length(int4)' does not exist
    Unable to identify a function which satisfies the given argument types
    You will have to retype your query using explicit typecasts
```
```
test=> SELECT length((text(755)));
  length
--------
          3
(1 row)
```

Figure 9.1: 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>[ ]</td>
<td>`col1</td>
<td>col2`</td>
</tr>
<tr>
<td>String</td>
<td><code>-</code></td>
<td><code>col</code> <code>pattern</code></td>
<td>BOOLEAN, <code>col</code> matches regular expression <code>pattern</code></td>
</tr>
<tr>
<td></td>
<td><code>!</code></td>
<td><code>col</code> <code>!</code> <code>pattern</code></td>
<td>BOOLEAN, <code>col</code> does not match regular expression <code>pattern</code></td>
</tr>
<tr>
<td></td>
<td><code>!*</code></td>
<td><code>col</code> <code>!*</code> <code>pattern</code></td>
<td>same as <code>!</code>, but case-insensitive</td>
</tr>
<tr>
<td></td>
<td><code>!*</code></td>
<td><code>col</code> <code>!*</code> <code>pattern</code></td>
<td>same as <code>!</code> , but case-insensitive</td>
</tr>
<tr>
<td></td>
<td><code>-</code></td>
<td><code>col</code> <code>match</code></td>
<td>BOOLEAN, <code>col</code> matches LIKE pattern</td>
</tr>
<tr>
<td></td>
<td>LIKE</td>
<td><code>col</code> LIKE <code>pattern</code></td>
<td>same as <code>*</code></td>
</tr>
<tr>
<td></td>
<td><code>!LIKE</code></td>
<td><code>col</code> <code>!LIKE</code> <code>pattern</code></td>
<td>BOOLEAN, <code>col</code> does not match LIKE <code>pattern</code></td>
</tr>
<tr>
<td></td>
<td>NOT LIKE</td>
<td><code>col</code> NOT LIKE <code>pattern</code></td>
<td>same as <code>!LIKE</code></td>
</tr>
<tr>
<td>Numeric</td>
<td><code>!</code></td>
<td><code>!col</code></td>
<td>factorial</td>
</tr>
<tr>
<td></td>
<td><code>+</code></td>
<td><code>col1 + col2</code></td>
<td>addition</td>
</tr>
<tr>
<td></td>
<td><code>-</code></td>
<td><code>col1 - col2</code></td>
<td>subtraction</td>
</tr>
<tr>
<td></td>
<td><code>*</code></td>
<td><code>col1 * col2</code></td>
<td>multiplication</td>
</tr>
<tr>
<td></td>
<td><code>/</code></td>
<td><code>col1 / col2</code></td>
<td>division</td>
</tr>
<tr>
<td></td>
<td><code>%</code></td>
<td><code>col1 % col2</code></td>
<td>remainder/modulo</td>
</tr>
<tr>
<td></td>
<td><code>-</code></td>
<td><code>col1</code> <code>pow</code> <code>col2</code></td>
<td><code>col1</code> raised to the power of <code>col2</code></td>
</tr>
<tr>
<td>Temporal</td>
<td><code>+</code></td>
<td><code>col1 + col2</code></td>
<td>addition of temporal values</td>
</tr>
<tr>
<td></td>
<td><code>-</code></td>
<td><code>col1</code> <code>-</code> <code>col2</code></td>
<td>subtraction of temporal values</td>
</tr>
<tr>
<td></td>
<td><code>(c1, c2)</code> OVERLAPS <code>(c3, c4)</code></td>
<td>BOOLEAN indicating <code>col</code>'s overlap in time</td>
<td></td>
</tr>
<tr>
<td>Geometric</td>
<td></td>
<td><code>psql</code> <code>do</code> for a list of geometric operators</td>
<td></td>
</tr>
<tr>
<td>Network</td>
<td><code>&lt;&lt;</code></td>
<td><code>col1</code> <code>&lt;&lt;</code> <code>col2</code></td>
<td>BOOLEAN indicating if <code>col1</code> is a subnet of <code>col2</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;=</code></td>
<td><code>col1</code> <code>&lt;=</code> <code>col2</code></td>
<td>BOOLEAN indicating if <code>col1</code> is equal or a subnet of <code>col2</code></td>
</tr>
<tr>
<td></td>
<td><code>&gt;&gt;</code></td>
<td><code>col1</code> <code>&gt;&gt;</code> <code>col2</code></td>
<td>BOOLEAN indicating if <code>col1</code> is a supernet of <code>col2</code></td>
</tr>
<tr>
<td></td>
<td><code>&gt;&gt;=</code></td>
<td><code>col1</code> <code>&gt;&gt;=</code> <code>col2</code></td>
<td>BOOLEAN indicating if <code>col1</code> is equal or a supernet of <code>col2</code></td>
</tr>
</tbody>
</table>

Table 9.4: Common operators
test=> SELECT date('1/1/1992') + date('1/1/1993');
ERROR: Unable to identify an operator '+' for types 'date' and 'date'
You will have to retype this query using an explicit cast

Figure 9.2: Error generated by undefined operator/type combination

### 9.6 Support Variables

There are several defined variables. These are shown in table 9.5.

<table>
<thead>
<tr>
<th>Meaning</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

### 9.7 Arrays

Arrays allow a column to store several simple data values. You can store one-dimensional arrays, two-dimensional arrays, or arrays with any number of dimensions.

An array column is created like an ordinary column, except brackets are used to specify the dimensions of the array. The number of dimensions and size of each dimension are for documentation purposes only. Values that do not match the dimensions specified at column creation are not rejected. Figure 9.3 creates a table with one-, two-, and three-dimensional INTEGER columns. The first and last columns have sizes specified.

Figure 9.3: Creation of array columns
The first column is a one-dimensional array, also called a list or vector. Values inserted into that column look like \{3,10,9,32,24\} or \{20,8,9,1,4\}. Each value is a list of integers, surrounded by curly braces. The second column, \textit{col2}, is a two-dimensional array. Typical values for this column are \{\{2,9,3\},\{4,3,5\}\} or \{\{18,6\},\{32,5\}\}. Notice double braces are used. The outer brace surrounds two one-dimensional arrays. You can think of it as a matrix, with the first one-dimensional array representing the first row of the array, and the second representing the second row of the array. Commas separate the individual elements, and each pair of braces. The third column of the \textit{array_test} table is a three-dimensional array, holding values like \{\{\{3,1\},\{1,9\}\},\{\{4,5\},\{8,2\}\}\}. This is a three-dimensional matrix made up of two 2×2 matrices. Arrays of any size can be constructed.

Figure 9.4 shows a query inserting values into \textit{array_test}, and several queries selecting data from the table. Brackets are used to access individual array elements.

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

Figure 9.4: Using arrays

Any data type can be used as an array. If individual elements of the array are accessed frequently or updated, it is better to use separate columns or tables rather than arrays.
CHAPTER 9. DATA TYPES

9.8 Large Objects (BLOBS)

POSTGRESQL can not store values of more than several thousand bytes using the above data types, nor can binary data be easily entered within single quotes. Large objects, also called Binary Large Objects or BLOBS, are used to store very large values and binary data.

Large objects allow storage of any operating system file, like images or large text files, directly into the database. You load the file into the database using \texttt{lo\_import()}, and retrieve the file from the database using \texttt{lo\_export()}. Figure 9.5 shows an example that stores a fruit name and image. \texttt{Lo\_import()} stores

\begin{verbatim}
  test=> CREATE TABLE fruit (name CHAR(30), image OID);
  CREATE
  test=> INSERT INTO fruit
  test-> VALUES ('peach', lo\_import('/usr/images/peach.jpg'));
  INSERT 27111 1
  test=> SELECT lo\_export(fruit.image, '/tmp/outimage.jpg')
  test-> FROM fruit
  test-> WHERE name = 'peach';
  lo\_export
  -----------
  1
  (1 row)
\end{verbatim}

Figure 9.5: Using large images


/usr/images/peach.jpg into the database and returns an OID value that is inserted into \texttt{fruit.image}. \texttt{Lo\_export()} uses the OID value stored in \texttt{fruit.image} to find the large object stored in the database, and places the image into the new file \texttt{/tmp/outimage.jpg}. The 1 returned by \texttt{lo\_export()} indicates a successful export.

Full pathnames must be used with large objects because the database server is running in a different directory than the \texttt{psql} client. Files are imported and exported by the \texttt{postgres} user, so \texttt{postgres} must have permission to read the file for \texttt{lo\_import()}, and directory write permission for \texttt{lo\_export()}.

9.9 Summary

Care should be used when choosing data types. The many data types give users great flexibility. Wise decisions about column names and types give the database structure and consistency. It also improves performance and allows efficient data storage. Don’t choose types hastily — you will regret it later.
Chapter 10
Transactions and Locks

Up to this point, we have used POSTGRESQL as a sophisticated filing cabinet. However, a database is much more. It allows users to view and modify information simultaneously. It helps ensure data integrity. This chapter explores these database capabilities.

10.1 Transactions

Though you may not have heard the term transaction before, you have already used them. Every SQL query is executed in a transaction. Transactions give databases an all-or-nothing capability when making modifications.

For example, suppose the query UPDATE trans_test SET col = 3 is in the process of modifying 700 rows. And suppose, after it has modified 200 rows, the user types control-C, or the computer reset button is pressed. When the user looks at trans_test, he will see that none of the rows have been updated.

This might surprise you. Because 200 of the 700 rows had already updated, you might suspect 200 rows had been modified. However, POSTGRESQL uses transactions to guarantee queries are either completed, or have no effect.

This feature is valuable. Suppose you were executing a query to add $500 to everyone’s salary. And suppose you kicked the power cord out of the wall while the update was happening. Without transactions, the query may have updated half the salaries, but not the rest. It would be difficult to know where the UPDATE stopped. You would wonder, “Which rows were updated, and which ones were not?” You can’t just re-execute the query, because some people have already received their $500 increase. With transactions, you can check to see if any of the rows were updated. If one was updated, they all were updated. If not, simply re-execute the query.

10.2 Multi-Statement Transactions

By default, each SQL query runs in its own transaction. Figures 10.1 and 10.2 show two identical queries.

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

Figure 10.1: INSERT with no explicit transaction

Figure 10.1 shows a typical INSERT query. Before POSTGRESQL starts the INSERT, it begins a transaction. It performs the INSERT, then commits the transaction. This is done automatically for any query with no explicit
test=> BEGIN WORK;
BEGIN
test=> INSERT INTO trans_test VALUES (1);
INSERT 130058 1
test=> COMMIT WORK;
COMMIT

Figure 10.2: INSERT with explicit transaction

transaction. Figure 10.2 shows an INSERT using an explicit transaction. BEGIN WORK starts the transaction, and COMMIT WORK commits the transaction. The only difference between the two queries is that there is an implied BEGIN WORK…COMMIT WORK surrounding first INSERT.

Even more valuable is the ability to bind multiple queries into a single transaction. When this is done, either all the queries execute to completion, or none of them have any effect. For example, figure 10.3 shows two INSERTs in a transaction. PostgreSQL guarantees either both INSERTs succeed, or none of them.

    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

For a more complicated example, suppose you have a table of bank account balances, and suppose you wish to transfer $100 from one account to another account. This is performed using two queries — an UPDATE to subtract $100 from one account, and an UPDATE to add $100 to another account. The UPDATES should either both complete, or none of them. If the first UPDATE completes but not the second, the $100 would disappear from the bank records. It would have been subtracted from one account, but never added. Such errors are very hard to find. Multi-statement transactions prevent them from happening. Figure 10.4 shows the two queries bound into a single transaction. The transaction forces POSTGRESQL to perform the

    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: Multi-statement transaction

queries as a single operation.
When you begin a transaction with `BEGIN WORK`, you don't have to commit it using `COMMIT WORK`. You can close the transaction with `ROLLBACK WORK` and the transaction will be discarded. The database is left as though the transaction had never been executed. In figure 10.5, the current transaction is rolled back, causing the `DELETE` have no effect. Also, if any query inside a multi-statement transaction can not be executed due to an error, the entire transaction is automatically rolled back.

### 10.3 Visibility of Committed Transactions

Though we have focused on the *all-or-nothing* nature of transactions, they have other important benefits. Only committed transactions are visible to users. Though the current users sees his changes, other users do not see them until the transaction is committed.

For example, figure 10.1 shows two users issuing queries using the default mode in which every statement is in its own transaction. Figure 10.2 shows the same query with *user 1* using a multi-query transaction. *User 1* sees the changes made by his transaction. However, *user 2* does not see the changes until *user 1* commits the transaction.

This is another advantage of transactions. They insulate users from seeing uncommitted transactions. Users never see a partially committed view of the database.

As another example, consider the bank account query where we transfered $100 from one bank account to another. Suppose we were calculating the total amount of money in all bank accounts at the same time the $100 was being transfered. If we did not see a consistent view of the database, we could have seen the $100 removed from the account, but not see the $100 added. Our bank account total would be wrong. A consistent database view means we either see the $100 in its original account, or we see it in its new account.

```sql
test=> INSERT INTO rollback_test VALUES (1);
    INSERT 1

test=> BEGIN WORK;
     BEGIN

test=> DELETE FROM rollback_test;
     DELETE 1

test=> ROLLBACK WORK;
     ROLLBACK

test=> SELECT * FROM rollback_test;

    x
    ---
    1

(1 row)
```

Figure 10.5: Transaction rollback
### Table 10.2: Visibility using multi-query transactions

<table>
<thead>
<tr>
<th>User 1</th>
<th>User 2</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN WORK</td>
<td>SELECT (*) FROM trans_test</td>
<td>User 1 starts a transaction</td>
</tr>
<tr>
<td>INSERT INTO trans_test VALUES (1)</td>
<td>returns 0</td>
<td>add row to trans_test</td>
</tr>
<tr>
<td>SELECT (*) FROM trans_test</td>
<td>returns 1</td>
<td></td>
</tr>
<tr>
<td>COMMIT WORK</td>
<td>SELECT (*) FROM trans_test</td>
<td>returns 0</td>
</tr>
<tr>
<td></td>
<td>SELECT (*) FROM trans_test</td>
<td>returns 1</td>
</tr>
</tbody>
</table>

Without this feature, we would have to make sure no one was making bank account transfers while we were calculating the amount of money in all accounts.

While this is a contrived example, real-world database users INSERT, UPDATE, and DELETE data all at the same time, while others SELECT data. All this activity is orchestrated by the database so each user can operate in a secure manner, knowing other users will not affect their results in an unpredictable way.

### 10.4 Read Committed and Serializable Isolation Levels

The previous section illustrated that users only see committed transactions. This does not address what happens if someone commits a transaction while you are in your own transaction. There are cases where you need to control if other transaction commits are seen by your transaction.

POSTGRESQL’s default isolation level, READ COMMITTED, allows you to see other transaction commits while your transaction is open. Figure 10.6 illustrates this effect. First, the transaction does a `SELECT COUNT(*)`. Then, while sitting at a `psql` prompt, someone inserts into the table. The next `SELECT COUNT(*)` returns 6.

```sql
BEGIN;
SELECT COUNT(*) FROM trans_test;
COUNT
-------
  5
(1 row)

-- someone commits INSERT INTO trans_test

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

COMMIT;
```

Figure 10.6: Read-committed isolation level

COUNT(*).
shows the newly INSERTED row. When another user commits a transaction, it is seen by the current transaction, even if it is committed after the current transaction started.

You can prevent your transaction from seeing changes made to the database. SET TRANSACTION ISOLATION LEVEL SERIALizable changes the isolation level of the current transaction. SERIALizable isolation prevents the current transaction from seeing commits made by other transactions. Any commit made after the start of the first query of the transaction is not visible. Figure 10.7 shows an example of a SERIALizable transaction.

```
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

SERIALizable isolation provides a stable view of the database for SELECT transactions. For transactions containing UPDATE and DELETE queries, SERIALizable mode is more complicated. SERIALizable isolation forces the database to execute all transactions as though they were run serially, one after another, even if they are run concurrently. If two concurrent transactions attempt to update the same row, serializability is impossible. When this happens, POSTGRESQL forces one transaction to roll back.

For SELECT-only transactions, SERIALizable isolation level should be used when you don’t want to see other transaction commits during your transaction. For UPDATE and DELETE transactions, SERIALizable isolation prevents concurrent modification of the same data row, and should be used with caution.

10.5 Locking

Exclusive locks, also called write locks, prevent other users from modifying a row or an entire table. Rows modified by UPDATE and DELETE are exclusively locked automatically for the duration of the transaction. This prevents other users from making changes to the row until the transaction is either committed or rolled back.

For example, table 10.3 shows two simultaneous UPDATE transactions affecting the same row. One trans-
### Table 10.3: Waiting for a lock

<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>Transaction 2</th>
<th>Notes</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 1 commits. Transaction 2 returns from UPDATE.</td>
</tr>
</tbody>
</table>

action must wait to see if the other transaction commits or rolls back. If these had been using SERIALIZABLE isolation level, transaction 2 would have been rolled back automatically if transaction 1 committed.

The only time users must wait for other users is when they are trying to modify the same row. If they modify different rows, there is no waiting. SELECT queries never have to wait.

Locking is done automatically by the database. However, there are cases when locking must be controlled manually. For example, figure 10.8 shows a query that first SELECTs a row, then performs an UPDATE. The

```sql
BEGIN WORK;
BEGIN
SELECT *
FROM lock_test
WHERE name = 'James';

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

TEST=>
--
-- the SELECTed row is not locked
--
UPDATE lock_test
SET name = 'Jim'
WHERE name = 'James';
UPDATE 1

TEST=> COMMIT WORK;
COMMIT
```

Figure 10.8: SELECT with no locking

problem is another user can modify the *James* row between the SELECT and UPDATE. To prevent this, you can use SERIALIZABLE isolation. In this mode, one of the UPDATES would fail. A better solution is to use SELECT...FOR UPDATE to lock the selected rows. Figure 10.9 shows the same query using SELECT...FOR UPDATE. Another user can not modify the *James* row between the SELECT...FOR UPDATE and UPDATE.

You can also manually control locking using the LOCK command. It allows specification of a transaction's lock type and scope. See the LOCK manual page for more information.
10.6 Deadlocks

It is possible to create an unrecoverable lock condition, called a deadlock. Figure 10.4 illustrates how two transactions become deadlocked. In this example, each transaction holds a lock and is waiting for the other

<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>Transaction2</th>
<th>Notes</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>UPDATE row 64</td>
<td>Holds waiting for transaction 2 to release write lock</td>
</tr>
<tr>
<td>COMMIT WORK</td>
<td>auto-ROLLBACK</td>
<td>Attempt to get write lock held by transaction 1</td>
</tr>
<tr>
<td></td>
<td>WORK</td>
<td>Deadlock detected — transaction 2 automatically rolled back</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>Transaction2</th>
<th>Notes</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>UPDATE row 64</td>
<td>Holds waiting for transaction 2 to release write lock</td>
</tr>
<tr>
<td>COMMIT WORK</td>
<td>auto-ROLLBACK</td>
<td>Attempt to get write lock held by transaction 1</td>
</tr>
<tr>
<td></td>
<td>WORK</td>
<td>Deadlock detected — transaction 2 automatically rolled back</td>
</tr>
</tbody>
</table>

Transaction 1 returns from UPDATE and commits

Table 10.4: Deadlock

transaction’s lock to be released. One transaction must be rolled back by POSTGRESQL because the two transactions will wait forever. Obviously, if they had acquired locks in the same order no deadlock would occur.

10.7 Conclusion

Single-user database queries are concerned with getting the job done. Multi-user queries must be designed to gracefully handle multiple users accessing the data.

Multi-user interaction can be very confusing. The database is constantly changing. In a multi-user environment, improperly constructed queries can randomly fail when users perform simultaneously operations.
Queries can not assume that rows from previous transactions still exist.

By understanding PostgreSQL's multi-user behavior, you are now prepared to create robust queries. Overlapping transactions and locking must always be considered. PostgreSQL has a powerful set of features to allow the construction of reliable multi-user queries.
Chapter 11

Performance

In an ideal world, users would never have to be concerned about performance. The system would tune itself. However, databases don’t live in an ideal world. An untuned database can be thousands of times slower than a tuned one, so it pays to take steps to get optimal performance. This chapter shows how to get optimal performance from your database.

11.1 Indexes

Indexing allows fast retrieval of specific rows from a table. For example, consider the query SELECT * FROM bigtable WHERE col = 43. Without an index, POSTGRESQL must scan through the entire table looking for rows where col equals 43. With an index on col, POSTGRESQL can go directly to the rows where col equals 43, bypassing all other rows.

Tables can be accessed two ways. Without an index, POSTGRESQL starts at the beginning of the table file and reads to the end, looking for relevant rows. With an index, POSTGRESQL looks up specific values in the index, and goes directly to the matching rows. For large tables, it can take minutes to check every row. Using an index, finding a specific row takes fractions of a second.

Internally, POSTGRESQL stores user data in operating system files. Each table is in its own file, and rows are stored one after another in the file. An index is a separate file that is sorted by one or more columns. It contains pointers into the table file, allowing rapid access to specific values in the table.

Unfortunately, POSTGRESQL does not create indexes automatically. Users must create them for frequently accessed columns. Indexes are created using the CREATE INDEX command, as shown in figure 11.1. In this example, bigtable_custid_idx is the name of the index, bigtable is the table being indexed, and customer_id is the column being indexed. You can use any name for the index, but it is good to use the table and column names as part of the index name, i.e. bigtable_customer_id_idx or i_bigtable_custid. This index is only useful for finding rows in bigtable for specific customer_ids. It can not help when accessing other columns because indexes are sorted by a specific column.

You can create as many indexes as you wish. Of course, an index on a seldom used column is a waste of disk space, and every table modification requires an update to each index, so performance can suffer with too many indexes.
It is possible to create an index spanning multiple columns. Multi-column indexes are sorted by the first indexed column, then in cases where the first column has several equal values, sorting continues using the second indexed column. The command `CREATE INDEX bigtable_age_gender_idx ON bigtable (age, gender)` creates an index on the column `age`, and where several `age` rows have the same value, then sorts on `gender`. Multi-column indexes are only useful with columns that have many duplicate values.

The query `SELECT * FROM bigtable WHERE age = 36 AND gender = 'F'` can use the index, as can `SELECT * FROM bigtable WHERE age = 36.` However, index `bigtable_age_gender_idx` is useless if you wish to look up rows only based on `gender`. The `gender` component of the index is used only after the `age` value has been specified. The query `SELECT * FROM bigtable WHERE gender = 'F'` can not use the index because there is no restriction on the `age` column, which is the first part of the index.

`CREATE INDEX` has several other options. See the manual page for more information.

### 11.2 Unique Indexes

Unique indexes are like ordinary indexes, except they prevent duplicate values from occurring in the table. For example, figure 11.2 shows the creation of a table and unique index, and then the attempted insertion of duplicate values. In `CREATE INDEX`, the word `UNIQUE` prevents the index from accepting duplicate values.

```
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

Sometimes unique indexes are created only to prevent duplicate values from appearing in the table. Multi-column unique indexes ensure the combination of all indexed columns remain unique. Unique indexes speed data access and prevent duplicates.

### 11.3 Cluster

The `CLUSTER` command reorders the table file to match the ordering of an index. This is a specialized command that is valuable only in cases where performance is critical, and the indexed column has many equal values.

For example, suppose `bigtable.age` has many duplicate values, and the query `SELECT * FROM bigtable WHERE age = 25` is executed. An index on `age` allows rapid retrieval of the row locations from the index, but if there are thousands of matching rows, they may be scattered in the table file, requiring many disk accesses to retrieve them. `CLUSTER` reorders the table, placing duplicate values next to each other. This speeds access for large queries accessing many duplicate values. See the `CLUSTER` manual page for more information.
11.4 Vacuum

When PostgreSQL updates a row, it keeps the old copy of the row in the table file and writes a new one. The old row is marked as expired, and is used by other transactions still viewing the database in its prior state. Deletions are similarly marked as expired, but not removed from the file.

The VACUUM command removes expired rows from the table file. While it removes them, it moves rows from the end of the table into the expired spots, thereby compacting the table. While VACUUM is doing this, no one is allowed to access the table.

There are two ways to run VACUUM. VACUUM alone vacuums all tables in the database. VACUUM tablename vacuums a single table.

The VACUUM command should be run periodically to clean out expired rows. For tables that are heavily modified, it may be useful to run VACUUM every night in an automated manner. For tables with few modifications, VACUUM should be run as needed.

11.5 Vacuum Analyze

VACUUM ANALYZE is an enhanced version of VACUUM. While VACUUM ANALYZE is vacuuming a table, it collects statistics about the each column’s proportion of duplicate values and the maximum and minimum values. This information is used by PostgreSQL when deciding how to efficiently execute complex queries. VACUUM ANALYZE should be run when the data stored in a table dramatically changes.

11.6 EXPLAIN

EXPLAIN reports how PostgreSQL executes queries. For example, figure 11.3 shows a SELECT query preceeded with the word EXPLAIN. EXPLAIN causes PostgreSQL to display how the query will be executed,

test=> EXPLAIN SELECT customer_id FROM bigtable;
NOTICE: QUERY PLAN:
Seq Scan on bigtable (cost=0.00..15.00 rows=1000 width=4)

EXPLAIN

Figure 11.3: Using EXPLAIN

rather than executing it. In the figure, PostgreSQL reports a sequential scan will be used on bigtable, meaning it will scan the entire table. Cost is an estimate of the effort required to perform the query. The numbers are only meaningful for comparison. Rows indicates the number of rows it expects to return. Width is the number of bytes per row.

Figure 11.4 shows more interesting examples of EXPLAIN. The first EXPLAIN shows a SELECT with the restriction customer_id = 55. This is again a sequential scan, but the restriction makes PostgreSQL think it is returning ten rows. A VACUUM ANALYZE is run, causing the next query to properly show a rows value of one instead of ten. An index is created, and the query rerun. This time, an index scan is used, allowing PostgreSQL to go right to the rows where customer_id equals 55. The next query illustrates that if no restriction is supplied, PostgreSQL properly decides that the index is of no use, and a sequential scan is performed. The last query uses an ORDER BY that matches the index, so PostgreSQL uses an index scan.

Much more complex queries can be viewed using EXPLAIN, as shown in figure 11.5. In this example, tab1
Figure 11.4: More complex EXPLAIN examples
test=> EXPLAIN SELECT * FROM tab1, tab2 WHERE col1 = col2;
NOTICE: QUERY PLAN:

<table>
<thead>
<tr>
<th>Plan Type</th>
<th>Cost</th>
<th>Rows</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Join</td>
<td>139.66</td>
<td>10000</td>
<td>8</td>
</tr>
<tr>
<td>-&gt; Sort</td>
<td>69.83</td>
<td>1000</td>
<td>4</td>
</tr>
<tr>
<td>-&gt; Seq Scan</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tab2</td>
<td>0.00</td>
<td>1000</td>
<td>4</td>
</tr>
<tr>
<td>-&gt; Sort</td>
<td>69.83</td>
<td>1000</td>
<td>4</td>
</tr>
<tr>
<td>-&gt; Seq Scan</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tab1</td>
<td>0.00</td>
<td>1000</td>
<td>4</td>
</tr>
</tbody>
</table>

EXPLAIN

Figure 11.5: EXPLAIN example using joins

and tab2 are joined on col1 and col2. In this case, each table is sequential scanned, and the result sorted. The two results are then merge joined to produce output. POSTGRESQL also supports hash join and nested loop join methods. POSTGRESQL chooses the join method it believes to be the fastest.

11.7 Summary

There are a variety of tools available to speed up POSTGRESQL queries. While they are not required to get results, performance tuning can produce huge improvements in query speed.
Chapter 12

Controlling Results

12.1 LIMIT

12.2 Cursors

DECLARE

FETCH

CLOSE
CHAPTER 12. CONTROLLING RESULTS
Chapter 13

Table Management

13.1 Temporary Tables

Purpose

Creation

13.2 ALTER TABLE

13.3 Inheritance

Purpose

Creation

Examples

13.4 Views

Creation

Limitations

13.5 Rules

Creation

Limitations

13.6 Triggers

13.7 Primary/Foreign Keys

13.8 NOTIFY and LISTEN

POSTGRESQL allows users to send signals to each other using LISTEN and NOTIFY. For example, suppose a user wants to receive notification when a table is updated. He can register the table name using the LISTEN
command. If someone updates the table and issues a NOTIFY command, all registered listeners will receive notification. For more information, see the NOTIFY and LISTEN manual pages.
Chapter 14
Importing and Exporting Data

14.1 COPY
Import
Export
DELIMITERS
BINARY
Frontend COPY
CHAPTER 14. IMPORTING AND EXPORTING DATA
Chapter 15

Extending POSTGRESQL

15.1 User-Defined Functions

Purpose

Creation

Examples

15.2 User-Defined Operators

Arithmetic Processing

Creation

15.3 User-Defined Types

Purpose

Creation

Indexing
Chapter 16

Programming Interfaces

There has been a lot of interest in this section. I plan to cover the strengths of each interface, and show an example of each. I do not plan to teach how to program in each interface nor provide a list of function calls for each, though this is possible depending on the publisher.

Add diagram here.

<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>speed, portability</td>
</tr>
<tr>
<td>libpq++</td>
<td>C++</td>
<td>compiled</td>
<td>speed, object-oriented</td>
</tr>
<tr>
<td>ecpg</td>
<td>C</td>
<td>compiled</td>
<td>SQL-embedded C</td>
</tr>
<tr>
<td>ODBC</td>
<td>C</td>
<td>compiled</td>
<td>non-UNIX portability</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java</td>
<td>interpreted</td>
<td>portability</td>
</tr>
<tr>
<td>libpgtcl</td>
<td>Tcl/Tk</td>
<td>interpreted</td>
<td>simplicity, windowing</td>
</tr>
<tr>
<td>perl</td>
<td>perl</td>
<td>interpreted</td>
<td>text processing</td>
</tr>
<tr>
<td>python</td>
<td>python</td>
<td>interpreted</td>
<td>object oriented</td>
</tr>
</tbody>
</table>

Table 16.1: Interface summary
16.1 Need for a Programming Language
16.2 C Language API (LIBPQ)
16.3 Embedded C (ECPG)
16.4 C++ (LIBPQ++)
16.5 JAVA (JDBC)
16.6 ODBC
16.7 PERL (PGSQL_PERL5)
16.8 TCL/TK (LIBPGTCL)
16.9 PYTHON (PYGRESQ5L)
16.10 Web Access
   PHP
   CGI
16.11 Server-side Programming
   PLPGSQL
   PL/TCL
   SPI
Chapter 17

User Applications

17.1 PSQL

The following sections describe advanced psql features. See chapter 2 for an introduction to psql.

Controlling Query Buffer

You can easily control the psql query buffer. Table 17.1 summarizes the commands. See the psql manual page for a full description of each item. There is one item of particular interest, edit (\e). This allows you to use your default editor to edit the query buffer. Type \e, and the current contents of the query buffer are loaded into your editor. Exit the editor, and the editor contents are reloaded into your query buffer, ready for execution.1

Additional PSQL Commands

You can display a variety of information about the current database from psql’s listing commands. Table 17.2 summarizes them. See the psql manual page for a full description of each item. You can control the way psql displays output. Table 17.3 summarizes the output format commands. See the psql manual page for a full description of each item. You can access special psql capabilities. Table 17.4 summarizes them. See the psql manual page for a full description of each item.

PSQL command-line arguments

You can change the behavior of psql in other ways. You normally start psql from the command line as psql followed by the database name. However, you can add extra arguments between psql and the database name to modify psql behavior. For example, psql -f file test will read commands from file, rather than from your keyboard. There is a summary of the options in table 17.5. Consult the psql manual page for more detailed information.

1Here is how to set your default editor. At the operating system prompt, try typing:

```bash
EDITOR=vi
export EDITOR
```

This sets the default editor to vi. If that doesn’t work, try:

```bash
setenv EDITOR vi
```

Do this at the operating system command prompt, not inside psql.
### Table 17.1: psql query buffer commands

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

### Table 17.2: psql listing commands

<table>
<thead>
<tr>
<th>Listing Type</th>
<th>psql Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>All tables</td>
<td>\dt</td>
</tr>
<tr>
<td>A single table</td>
<td>\d tablename</td>
</tr>
<tr>
<td>A single index</td>
<td>\d indexname</td>
</tr>
<tr>
<td>Indexes</td>
<td>\di</td>
</tr>
<tr>
<td>Tables and indexes</td>
<td>\d</td>
</tr>
<tr>
<td>Functions</td>
<td>\df</td>
</tr>
<tr>
<td>Operators</td>
<td>\do</td>
</tr>
<tr>
<td>Aggregates</td>
<td>\da</td>
</tr>
<tr>
<td>Sequences</td>
<td>\ds</td>
</tr>
<tr>
<td>System tables</td>
<td>\dS</td>
</tr>
<tr>
<td>Comments</td>
<td>\dd object</td>
</tr>
<tr>
<td>Databases</td>
<td>\l</td>
</tr>
<tr>
<td>Grant permissions</td>
<td>\z</td>
</tr>
</tbody>
</table>

### Table 17.3: psql output commands

<table>
<thead>
<tr>
<th>Modifies</th>
<th>psql Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field alignment</td>
<td>\a</td>
</tr>
<tr>
<td>Field separator</td>
<td>\f separator</td>
</tr>
<tr>
<td>Expanded output</td>
<td>\x</td>
</tr>
<tr>
<td>No table headings or row count</td>
<td>\t</td>
</tr>
<tr>
<td>Compatibility display</td>
<td>\m</td>
</tr>
<tr>
<td>Enable HTML</td>
<td>\H</td>
</tr>
<tr>
<td>HTML3 options</td>
<td>\T html</td>
</tr>
<tr>
<td>HTML caption</td>
<td>\C caption</td>
</tr>
</tbody>
</table>
### Table 17.4: psql special capabilities

<table>
<thead>
<tr>
<th>Capability</th>
<th>psql Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect to another database</td>
<td><code>\connect dbname</code></td>
</tr>
<tr>
<td>Copy datafile into database</td>
<td>`\copy tablename to</td>
</tr>
</tbody>
</table>

### Table 17.5: psql command-line arguments

<table>
<thead>
<tr>
<th>Option</th>
<th>Capability</th>
<th>psql Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controlling Output</td>
<td>Field justification</td>
<td><code>-A</code></td>
</tr>
<tr>
<td></td>
<td>Field separator</td>
<td><code>-F separator</code></td>
</tr>
<tr>
<td></td>
<td>No column titles and row counts</td>
<td><code>-t</code></td>
</tr>
<tr>
<td></td>
<td>Extended output format</td>
<td><code>-x</code></td>
</tr>
<tr>
<td></td>
<td>Echo query</td>
<td><code>-e</code></td>
</tr>
<tr>
<td></td>
<td>Echo <code>\d</code> query</td>
<td><code>-E</code></td>
</tr>
<tr>
<td></td>
<td>Quiet mode</td>
<td><code>-q</code></td>
</tr>
<tr>
<td></td>
<td>HTML output</td>
<td><code>-H</code></td>
</tr>
<tr>
<td></td>
<td>HTML table options</td>
<td><code>-T options</code></td>
</tr>
<tr>
<td></td>
<td>List databases</td>
<td><code>-l</code></td>
</tr>
<tr>
<td></td>
<td>Disable readme</td>
<td><code>-n</code></td>
</tr>
<tr>
<td>Automation</td>
<td>Execute query</td>
<td><code>-c query</code></td>
</tr>
<tr>
<td></td>
<td>Get queries from file</td>
<td><code>-f file</code></td>
</tr>
<tr>
<td></td>
<td>Output to file</td>
<td><code>-o file</code></td>
</tr>
<tr>
<td></td>
<td>Single-step mode</td>
<td><code>-s</code></td>
</tr>
<tr>
<td></td>
<td>Single-step with newline termination</td>
<td><code>-S</code></td>
</tr>
<tr>
<td>Connection</td>
<td>Hostname</td>
<td><code>-h hostname</code></td>
</tr>
<tr>
<td></td>
<td>Port</td>
<td><code>-p port</code></td>
</tr>
<tr>
<td></td>
<td>Password</td>
<td><code>-u</code></td>
</tr>
</tbody>
</table>
17.2  PGACCESS
Chapter 18

POSTGRESQL Administration

18.1 Creating Users and Databases
18.2 Backup and Restore
18.3 Performance
18.4 Troubleshooting
18.5 Customization
18.6 System Tables
18.7 Access Configuration
   Server Access – flags, pg_hba.conf
   Database Access create user
   Table Access – GRANT
18.8 Internationalization
   National Character Encodings
   Date Formats
Appendix A

Additional Resources

A.1 Frequently Asked Questions (FAQ’S)
A.2 Mailing List Support
A.3 Supplied Documentation
A.4 Commercial Support
A.5 Modifying the Source Code
Appendix B

Installation

B.1 Getting POSTGRESQL
B.2 Compiling
B.3 Initialization
B.4 Starting the Server
B.5 Creating a Database
Appendix C

PostgreSQL Non-Standard Features by Chapter
APPENDIX C. POSTGRESQL NON-STANDARD FEATURES BY CHAPTER
Appendix D

Reference Manual

The following is a copy of the reference manual pages (man pages) as they appeared in a pre-release version of PostgreSQL 7.0.

The manual pages go here. They are in SGML/Docbook format. Approximately 100 pages.
Bibliography


[Date, Standard] Date, C.J. *A Guide to The SQL Standard*, Addison–Wesley

[Date, Introduction] Date. C.J. *An Introduction to Database Systems*, Addison–Wesley

[Celko] Celko, Joe *SQL For Smarties*, Morgan, Kaufmann


