PostgreSQL: Introduction and Concepts

Bruce Momjian

June 17, 2000
<table>
<thead>
<tr>
<th>WHERE</th>
<th>NULL</th>
<th>CREATE</th>
<th>UNION</th>
<th>AS</th>
<th>DISTINCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX</td>
<td>TRIGGER</td>
<td>GRANT</td>
<td>ROLLBACK</td>
<td>DEFAULT</td>
<td>SUM</td>
</tr>
<tr>
<td>INTO</td>
<td>ALTER</td>
<td>COMMIT</td>
<td>SELECT</td>
<td>REVOKE</td>
<td>CASE</td>
</tr>
<tr>
<td>TABLE</td>
<td>FROM</td>
<td>INSERT</td>
<td>OPERATOR</td>
<td>SET</td>
<td>UPDATE</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>EXCEPT</td>
<td>DELETE</td>
<td>VALUES</td>
<td>ORDER BY</td>
<td>COUNT</td>
</tr>
<tr>
<td>BEGIN WORK</td>
<td>LIKE</td>
<td>IN</td>
<td>VIEW</td>
<td>HAVING</td>
<td>EXISTS</td>
</tr>
</tbody>
</table>
Note to Reviewers

The material on these pages is a work in progress, 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. The appendix needs a little more work.

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 June 17, 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.
NOTE TO REVIEWERS
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 are 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 also expand 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 small project to overhaul POSTGRES. POSTGRES is a novel and feature-rich database system created by the 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 and 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
FOREWORD
Preface

This book is about PostgreSQL, the most advanced open source database. From its origin in academia, PostgreSQL has moved to the Internet with explosive growth. It is hard to believe the advances during the past four 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, such as the ability to type at an operating system prompt.

The book begins 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.

At each step, the purpose of each command 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 database 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 a safe way to try queries. As readers gain experience, later chapters will begin 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.

Much information has been moved out of the main body of the book into appendices. Appendix A shows how to find additional information about PostgreSQL. Appendix B has information about installing PostgreSQL. Appendix C lists the features of PostgreSQL not found in other database systems. 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 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, Pennsylvania, USA Often seen working on planner/optimizer, but has left fingerprints in many places. Generally more interested in bugfixes and performance improvements than adding features.
- **LOCKHART, THOMAS G.** in Pasadena, California, USA works on documentation, data types, particularly date/time and geometric objects, and on SQL standards compatibility.
- **MIKHEEV, VADIM B.** in San Francisco, California, USA does large projects, like vacuum, subselects, triggers, and multi-version concurrency control(MVCC).
- **MOMJIAN, BRUCE** in Philadelphia, Pennsylvania, USA maintains FAQ and TODO lists, code cleanup, patch application, training materials, and some coding.
- **WIECK, JAN** near Hamburg, Germany overhauled the query rewrite rule system, wrote our procedural languages PL/PGSQL and PL/TCL and added the NUMERIC type.

Major Developers

- **CAIN, D’ARCY J.M.** in Toronto, Ontario, Canada worked on the TCL interface, PyGreSQL, and the INET type.
- **DAL ZOTTO, MASSIMO** near Trento, Italy has done locking code and other improvements.
- **ELPHICK, OLIVER** in Newport, Isle of Wight, UK maintains the POSTGRESQL package for Debian GNU/Linux.
- **HORAK, DANIEL** near Pilzen, Czech Republic did the WinNT port of PostgreSQL (using the Cygwin environment).
- **INOUE, HIROSHI** in Fukui, Japan improved btree index access.
• Ishii, Tatsuo in Zushi, Kanagawa, Japan handles multi-byte foreign language support and porting issues.

• Martin, Dr. Andrew C.R. in London, England has done the ECPG interface and helped in the Linux and Irix FAQs including some patches to the PostgreSQL code.

• Mergl, Edmund in Stuttgart, Germany created and maintains pgsql_perl5. He also created DBD-Pg which is available via CPAN.

• Meskes, Michael in Dusseldorf, Germany handles multi-byte foreign language support, and maintains ecpg.

• Mount, Peter in Maidstone, Kent, United Kingdom has done the Java JDBC Interface.

• Nikolaidis, Byron in Baltimore, Maryland, USA rewrote and maintains the ODBC interface for Windows.

• Owen, Lamar in Pisgah Forest, North Carolina, USA RPM package maintainer.

• Teodorescu, Constantin in Braila, Romania has done the PgAccess DB Interface.

• Thyni, Göran in Kiruna, Sweden has worked on the unix socket code.

Non-code contributors

• Bartunov, Oleg in Moscow, Russia introduced the locale support.

• Vielhaber, Vince near Detroit, Michigan, USA 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
   1.1 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
   1.6 Summary .................................................................. 3

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

3 Basic SQL Commands ....................................................... 9
   3.1 Relational Databases .................................................... 9
   3.2 Creating Tables .......................................................... 10
   3.3 Adding Data with INSERT ........................................... 11
   3.4 Viewing Data with SELECT ......................................... 13
   3.5 Selecting Specific Rows with WHERE .............................. 13
   3.6 Removing Data with DELETE ....................................... 15
   3.7 Modifying Data with UPDATE ...................................... 15
   3.8 Sorting Data with ORDER BY ........................................ 15
   3.9 Destroying Tables ........................................................ 18
   3.10 Summary .................................................................. 18

4 Customizing Queries ......................................................... 19
   4.1 Data types ................................................................. 19
   4.2 Quotes Inside Text ....................................................... 19
   4.3 Using NULL Values ....................................................... 21
## CONTENTS

4.4 Controlling DEFAULT Values .................................................. 23 0727
4.5 Column Labels ........................................................................ 24 0728
4.6 Comments .............................................................................. 24 0729
4.7 AND/OR Usage ....................................................................... 25 0730
4.8 Range of Values ..................................................................... 28 0732
4.9 LIKE Comparison ..................................................................... 28 0733
4.10 Regular Expressions ............................................................... 29 0735
4.11 CASE Clause ......................................................................... 33 0736
4.12 Distinct Rows ........................................................................ 34 0739
4.13 Functions and Operators ......................................................... 34 0740
4.14 SET, SHOW, and RESET .......................................................... 38 0741
4.15 Summary .............................................................................. 38 0742

5 SQL Aggregates ......................................................................... 41 0743
5.1 Aggregates .............................................................................. 41 0744
5.2 Using GROUP BY ..................................................................... 44 0745
5.3 Using HAVING ......................................................................... 44 0750
5.4 Query Tips .............................................................................. 45 0751
5.5 Summary .............................................................................. 46 0752

6 Joining Tables ............................................................................ 47 0753
6.1 Table and Column References .................................................. 47 0754
6.2 Joined Tables .......................................................................... 47 0755
6.3 Creating Joined Tables ............................................................. 49 0756
6.4 Performing Joins ..................................................................... 51 0757
6.5 Three and Four Table Joins ...................................................... 53 0758
6.6 Additional Join Possibilities ..................................................... 55 0760
6.7 Choosing a Join Key ................................................................. 56 0761
6.8 One-to-Many Joins ................................................................. 57 0762
6.9 Unjoined Tables ...................................................................... 59 0763
6.10 Table Aliases and Self-Joins ..................................................... 59 0764
6.11 Non-EquiJoins ...................................................................... 60 0765
6.12 Ordering Multiple Parts ........................................................... 60 0766
6.13 Primary and Foreign Keys ....................................................... 62 0767
6.14 Summary .............................................................................. 62 0768

7 Numbering Rows ....................................................................... 65 0769
7.1 Object Identification Numbers (OIDs) ....................................... 65 0770
7.2 Object Identification Number Limitations .................................. 66 0771
7.3 Sequences .............................................................................. 67 0772
7.4 Creating Sequences ................................................................. 67 0773
7.5 Using Sequences to Number Rows .......................................... 69 0774
7.6 Serial Column Type ................................................................. 70 0775
7.7 Manually Numbering Rows ...................................................... 70 0776
7.8 Summary .............................................................................. 71 0777
## CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Combining SELECTs</td>
<td>73</td>
</tr>
<tr>
<td>8.1</td>
<td>UNION, EXCEPT, INTERSECT Clauses</td>
<td>73</td>
</tr>
<tr>
<td>8.2</td>
<td>Subqueries</td>
<td>76</td>
</tr>
<tr>
<td>8.3</td>
<td>Outer Joins</td>
<td>83</td>
</tr>
<tr>
<td>8.4</td>
<td>Subqueries in Non-SELECT Queries</td>
<td>83</td>
</tr>
<tr>
<td>8.5</td>
<td>UPDATE with FROM</td>
<td>85</td>
</tr>
<tr>
<td>8.6</td>
<td>Inserting Data Using SELECT</td>
<td>85</td>
</tr>
<tr>
<td>8.7</td>
<td>Creating Tables Using SELECT</td>
<td>86</td>
</tr>
<tr>
<td>8.8</td>
<td>Summary</td>
<td>87</td>
</tr>
<tr>
<td>9</td>
<td>Data Types</td>
<td>89</td>
</tr>
<tr>
<td>9.1</td>
<td>Purpose of Data Types</td>
<td>89</td>
</tr>
<tr>
<td>9.2</td>
<td>Installed Types</td>
<td>89</td>
</tr>
<tr>
<td>9.3</td>
<td>Type Conversion using CAST</td>
<td>93</td>
</tr>
<tr>
<td>9.4</td>
<td>Support Functions</td>
<td>93</td>
</tr>
<tr>
<td>9.5</td>
<td>Support Operators</td>
<td>93</td>
</tr>
<tr>
<td>9.6</td>
<td>Support Variables</td>
<td>96</td>
</tr>
<tr>
<td>9.7</td>
<td>Arrays</td>
<td>96</td>
</tr>
<tr>
<td>9.8</td>
<td>Large Objects (BLOBs)</td>
<td>98</td>
</tr>
<tr>
<td>9.9</td>
<td>Summary</td>
<td>98</td>
</tr>
<tr>
<td>10</td>
<td>Transactions and Locks</td>
<td>99</td>
</tr>
<tr>
<td>10.1</td>
<td>Transactions</td>
<td>99</td>
</tr>
<tr>
<td>10.2</td>
<td>Multi-Statement Transactions</td>
<td>99</td>
</tr>
<tr>
<td>10.3</td>
<td>Visibility of Committed Transactions</td>
<td>101</td>
</tr>
<tr>
<td>10.4</td>
<td>Read Committed and Serializable Isolation Levels</td>
<td>102</td>
</tr>
<tr>
<td>10.5</td>
<td>Locking</td>
<td>103</td>
</tr>
<tr>
<td>10.6</td>
<td>Deadlocks</td>
<td>105</td>
</tr>
<tr>
<td>10.7</td>
<td>Summary</td>
<td>105</td>
</tr>
<tr>
<td>11</td>
<td>Performance</td>
<td>107</td>
</tr>
<tr>
<td>11.1</td>
<td>Indexes</td>
<td>107</td>
</tr>
<tr>
<td>11.2</td>
<td>Unique Indexes</td>
<td>108</td>
</tr>
<tr>
<td>11.3</td>
<td>Cluster</td>
<td>108</td>
</tr>
<tr>
<td>11.4</td>
<td>Vacuum</td>
<td>109</td>
</tr>
<tr>
<td>11.5</td>
<td>Vacuum Analyze</td>
<td>109</td>
</tr>
<tr>
<td>11.6</td>
<td>EXPLAIN</td>
<td>109</td>
</tr>
<tr>
<td>11.7</td>
<td>Summary</td>
<td>111</td>
</tr>
<tr>
<td>12</td>
<td>Controlling Results</td>
<td>113</td>
</tr>
<tr>
<td>12.1</td>
<td>LIMIT</td>
<td>113</td>
</tr>
<tr>
<td>12.2</td>
<td>Cursors</td>
<td>114</td>
</tr>
<tr>
<td>12.3</td>
<td>Summary</td>
<td>114</td>
</tr>
<tr>
<td>13</td>
<td>Table Management</td>
<td>117</td>
</tr>
<tr>
<td>13.1</td>
<td>Temporary Tables</td>
<td>117</td>
</tr>
<tr>
<td>13.2</td>
<td>ALTER TABLE</td>
<td>117</td>
</tr>
<tr>
<td>13.3</td>
<td>GRANT and REVOKE</td>
<td>119</td>
</tr>
<tr>
<td>13.4</td>
<td>Inheritance</td>
<td>120</td>
</tr>
</tbody>
</table>
## CONTENTS

13.5 Views ............................................................... 122
13.6 Rules ................................................................... 123
13.7 LISTEN and NOTIFY .......................................... 126
13.8 Summary ............................................................. 126

14 Constraints .......................................................... 129
  14.1 NOT NULL ........................................................ 129
  14.2 UNIQUE .......................................................... 130
  14.3 PRIMARY KEY .................................................. 131
  14.4 FOREIGN KEY/REFERENCES ............................... 131
  14.5 CHECK ........................................................... 139
  14.6 Summary .......................................................... 139

15 Importing and Exporting Data .................................. 141
  15.1 Using COPY ...................................................... 141
  15.2 COPY File Format .............................................. 141
  15.3 DELIMITERS .................................................... 143
  15.4 COPY without files ............................................ 143
  15.5 Backslashes and NULLS ....................................... 144
  15.6 COPY Tips ........................................................ 145
  15.7 Summary .......................................................... 145

16 Database Query Tools .............................................. 147
  16.1 Psql ............................................................... 147
  16.2 PGACCESS ...................................................... 152
  16.3 Summary .......................................................... 154

17 Programming Interfaces ........................................... 155
  17.1 C Language Interface (LIBPQ) .............................. 156
  17.2 Pgeasy(LIBPGEASY) ......................................... 158
  17.3 Embedded C (ECPG) ......................................... 158
  17.4 C++ (LIBPQ++) ................................................. 158
  17.5 Compiling Programs .......................................... 158
  17.6 Assignment to Program Variables ........................ 162
  17.7 ODBC ............................................................ 162
  17.8 JAVA (JDBC) ...................................................... 163
  17.9 Scripting Languages .......................................... 163
  17.10 PERL ........................................................... 163
  17.11 TCL/TK (PGTCLSH/PGTKSH) ............................. 163
  17.12 PYTHON (PYGRESQL) ..................................... 163
  17.13 PHP ............................................................. 166
  17.14 Installing Scripting Languages ............................ 168
  17.15 Summary ........................................................ 168

18 Functions and Triggers ............................................. 169
  18.1 Functions ......................................................... 169
  18.2 SQL Functions .................................................. 169
  18.3 PL/PGSQL Functions .......................................... 174
  18.4 Triggers ........................................................... 180
## CONTENTS

18.5 Summary ...................................................... 180

### 19 Extending PostgreSQL Using C

19.1 Writing C code .............................................. 183
19.2 Compile the C code ........................................... 184
19.3 Register the New Functions ................................. 184
19.4 Optionally Create Operators, Types, and Aggregates .... 185
19.5 Summary ...................................................... 185

### 20 Administration

20.1 Files ......................................................... 187
20.2 Creating Users .............................................. 187
20.3 Creating Databases ......................................... 189
20.4 Access Configuration ....................................... 189
20.5 Backup and Restore ......................................... 191
20.6 Server Startup and Shutdown ............................... 192
20.7 Monitoring ................................................... 192
20.8 Performance ................................................ 193
20.9 System Tables ............................................. 194
20.10 Internationalization ........................................ 194
20.11 Upgrading .................................................. 194
20.12 Summary ................................................... 194

### A Additional Resources

A.1 Frequently Asked Questions (FAQ's) ...................... 197
A.2 Mailing List Support ....................................... 197
A.3 Supplied Documentation .................................... 197
A.4 Commercial Support ....................................... 197
A.5 Modifying the Source Code ................................. 197

### B Installation

199

### C PostgreSQL Non-Standard Features by Chapter

201

### D Reference Manual

203

### Bibliography

205

### Index

205
List of Figures

2.1 psql session startup .................................................. 6
2.2 My first SQL query ................................................. 6
2.3 Multi-line query ....................................................... 7
2.4 Backslash-p demo ....................................................... 8
3.1 Databases ............................................................. 9
3.2 Create table friend .................................................. 10
3.3 Example of backslash-d .............................................. 11
3.4 INSERT into friend .................................................. 12
3.5 Additional friend INSERTs ......................................... 12
3.6 My first SELECT ....................................................... 13
3.7 My first WHERE ....................................................... 14
3.8 More complex WHERE clause ...................................... 14
3.9 A single cell .......................................................... 14
3.10 A block of cells ...................................................... 14
3.11 Comparing string fields .......................................... 15
3.12 DELETE example .................................................... 16
3.13 My first UPDATE ..................................................... 16
3.14 Use of ORDER BY ................................................... 17
3.15 Reverse ORDER BY ................................................ 17
3.16 Use of ORDER BY and WHERE ................................... 17
4.1 Example of common data types .................................... 20
4.2 Insertion of specific columns ..................................... 21
4.3 NULL handling ........................................................ 22
4.4 Comparison of NULL fields ......................................... 22
4.5 NULLs and blank strings .......................................... 23
4.6 Using DEFAULTs ...................................................... 24
4.7 Controlling column labels ......................................... 24
4.8 Computation using a column label ............................... 25
4.9 Comment styles ........................................................ 25
4.10 New friends .......................................................... 26
4.11 WHERE test for Sandy Gleason ................................... 26
4.12 Friends in New Jersey and Pennsylvania ....................... 27
4.13 Mixing ANDs and ORs .............................................. 27
4.14 Properly mixing ANDs and ORs ................................... 27
4.15 Selecting a range of values ....................................... 28
4.16 Firstname begins with D .......................................... 29
4.17 Regular expression sample queries .............................. 31
# LIST OF FIGURES

4.18 Complex regular expression queries ................................................. 32
4.19 CASE example ............................................................................. 33
4.20 Complex CASE example .................................................................. 34
4.21 DISTINCT prevents duplicates ......................................................... 35
4.22 Function examples ......................................................................... 36
4.23 Operator examples .......................................................................... 37
4.24 SHOW and RESET examples ............................................................. 39

5.1 Aggregate examples .......................................................................... 42
5.2 Aggregates and NULLS ...................................................................... 43
5.3 Aggregate with GROUP BY ................................................................. 44
5.4 GROUP BY on two columns ................................................................. 45
5.5 HAVING usage ................................................................................. 45

6.1 Qualified column names .................................................................... 48
6.2 Joining tables .................................................................................. 48
6.3 Creation of company tables ............................................................... 50
6.4 Insertion into company tables .......................................................... 51
6.5 Finding customer name using two queries ........................................ 52
6.6 Finding customer name using one query .......................................... 52
6.7 Finding order number for customer name ........................................ 53
6.8 Three-table join ............................................................................... 53
6.9 Four-table join ................................................................................ 54
6.10 Employees who have taken orders for customers. ............................ 54
6.11 Joining customer and employee ....................................................... 55
6.12 Joining part and employee ............................................................... 55
6.13 Statename table ............................................................................... 56
6.14 Using a customer code .................................................................... 57
6.15 One-to-many join ........................................................................... 58
6.16 Unjoined tables ............................................................................. 59
6.17 Using table aliases .......................................................................... 59
6.18 Examples of self-joins using table aliases ....................................... 60
6.19 Non-equi-joins ............................................................................... 61
6.20 New salesorder table for multiple parts per order ......................... 61
6.21 Orderpart table ............................................................................. 61
6.22 Queries involving orderpart table ................................................... 63

7.1 OID test .......................................................................................... 66
7.2 Columns with OIDs .......................................................................... 66
7.3 Examples of sequence function use ................................................. 68
7.4 Numbering customer rows using a sequence .................................. 69
7.5 Customer table using SERIAL ......................................................... 70

8.1 Combining two columns with UNION .............................................. 73
8.2 Combining two tables with UNION .................................................. 74
8.3 UNION with duplicates .................................................................... 75
8.4 UNION ALL with duplicates .............................................................. 75
8.5 EXCEPT restricts output from the first SELECT ............................... 76
8.6 INTERSECT returns only duplicated rows ....................................... 76
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.7</td>
<td>Friends not in <em>Dick Gleason’s state</em></td>
<td>77</td>
</tr>
<tr>
<td>8.8</td>
<td>Subqueries can replace some joins</td>
<td>78</td>
</tr>
<tr>
<td>8.9</td>
<td>Correlated subquery</td>
<td>79</td>
</tr>
<tr>
<td>8.10</td>
<td>Employees who took orders</td>
<td>80</td>
</tr>
<tr>
<td>8.11</td>
<td>Customers who have no orders</td>
<td>81</td>
</tr>
<tr>
<td>8.12</td>
<td>Inquery rewritten using ANY and EXISTS</td>
<td>82</td>
</tr>
<tr>
<td>8.13</td>
<td>NOT IN query rewritten using ALL and EXISTS</td>
<td>83</td>
</tr>
<tr>
<td>8.14</td>
<td>Simulating outer joins</td>
<td>84</td>
</tr>
<tr>
<td>8.15</td>
<td>Subqueries with UPDATE and DELETE</td>
<td>84</td>
</tr>
<tr>
<td>8.16</td>
<td>UPDATE the <code>order_date</code></td>
<td>85</td>
</tr>
<tr>
<td>8.17</td>
<td>Using SELECT with INSERT</td>
<td>85</td>
</tr>
<tr>
<td>8.18</td>
<td>Table creation with SELECT</td>
<td>86</td>
</tr>
<tr>
<td>9.1</td>
<td>Example of a function call</td>
<td>93</td>
</tr>
<tr>
<td>9.2</td>
<td>Error generated by undefined function/type combination</td>
<td>95</td>
</tr>
<tr>
<td>9.3</td>
<td>Error generated by undefined operator/type combination</td>
<td>96</td>
</tr>
<tr>
<td>9.4</td>
<td>Creation of array columns</td>
<td>96</td>
</tr>
<tr>
<td>9.5</td>
<td>Using arrays</td>
<td>97</td>
</tr>
<tr>
<td>9.6</td>
<td>Using large images</td>
<td>98</td>
</tr>
<tr>
<td>10.1</td>
<td>INSERT with no explicit transaction</td>
<td>99</td>
</tr>
<tr>
<td>10.2</td>
<td>INSERT with explicit transaction</td>
<td>100</td>
</tr>
<tr>
<td>10.3</td>
<td>Two INSERTs in a single transaction</td>
<td>100</td>
</tr>
<tr>
<td>10.4</td>
<td>Multi-statement transaction</td>
<td>100</td>
</tr>
<tr>
<td>10.5</td>
<td>Transaction rollback</td>
<td>101</td>
</tr>
<tr>
<td>10.6</td>
<td>Read-committed isolation level</td>
<td>102</td>
</tr>
<tr>
<td>10.7</td>
<td>Serializable isolation level</td>
<td>103</td>
</tr>
<tr>
<td>10.8</td>
<td>SELECT with no locking</td>
<td>104</td>
</tr>
<tr>
<td>10.9</td>
<td>SELECT...FOR UPDATE</td>
<td>105</td>
</tr>
<tr>
<td>11.1</td>
<td>Example of CREATE INDEX</td>
<td>107</td>
</tr>
<tr>
<td>11.2</td>
<td>Example of a unique index</td>
<td>108</td>
</tr>
<tr>
<td>11.3</td>
<td>Using EXPLAIN</td>
<td>109</td>
</tr>
<tr>
<td>11.4</td>
<td>More complex EXPLAIN examples</td>
<td>110</td>
</tr>
<tr>
<td>11.5</td>
<td>EXPLAIN example using joins</td>
<td>111</td>
</tr>
<tr>
<td>12.1</td>
<td>Examples of LIMIT and LIMIT/OFFSET</td>
<td>113</td>
</tr>
<tr>
<td>12.2</td>
<td>Cursor usage</td>
<td>115</td>
</tr>
<tr>
<td>13.1</td>
<td>Temporary table auto-destruction</td>
<td>118</td>
</tr>
<tr>
<td>13.2</td>
<td>Example of temporary table use</td>
<td>118</td>
</tr>
<tr>
<td>13.3</td>
<td>ALTER TABLE examples</td>
<td>119</td>
</tr>
<tr>
<td>13.4</td>
<td>Examples of the GRANT command</td>
<td>120</td>
</tr>
<tr>
<td>13.5</td>
<td>Creation of inherited tables</td>
<td>120</td>
</tr>
<tr>
<td>13.6</td>
<td>Accessing inherited tables</td>
<td>121</td>
</tr>
<tr>
<td>13.7</td>
<td>Inheritance in layers</td>
<td>121</td>
</tr>
<tr>
<td>13.8</td>
<td>Examples of views</td>
<td>122</td>
</tr>
<tr>
<td>13.9</td>
<td>Rule that prevents INSERT</td>
<td>123</td>
</tr>
<tr>
<td>13.10</td>
<td>Rules to log table changes</td>
<td>124</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

13.11 Use of rule to log table changes .................................................. 125  
13.12 Views ignore table modifications ................................................. 125  
13.13 Rules to handle view modifications ............................................. 126  
13.14 Rules handle view modifications ................................................... 127  

14.1 NOT NULL constraint .................................................................. 129  
14.2 NOT NULL with DEFAULT constraint ............................................. 130  
14.3 Unique column constraint ............................................................. 130  
14.4 Multi-column unique constraint ................................................. 131  
14.5 Creation of PRIMARY KEY column .............................................. 131  
14.6 Example of a multi-column primary key ..................................... 132  
14.7 Foreign key creation ................................................................. 132  
14.8 Foreign key constraints ............................................................... 133  
14.9 Creation of company tables using primary and foreign keys ....... 133  
14.10 Customer table with foreign key actions .................................... 134  
14.11 Foreign key actions ................................................................. 135  
14.12 Example of a multi-column foreign key .................................... 136  
14.13 MATCH FULL foreign key ........................................................... 137  
14.14 DEFERRABLE foreign key constraint ....................................... 138  
14.15 CHECK constraints ................................................................. 139  

15.1 Example of COPY...TO and COPY...FROM .................................. 142  
15.2 Example of COPY...FROM ............................................................ 142  
15.3 Example of COPY...TO...USING DELIMITERS ......................... 143  
15.4 Example of COPY...FROM...USING DELIMITERS ................. 143  
15.5 COPY using stdin and stdout ...................................................... 144  
15.6 COPY backslash handling ............................................................ 144  

16.1 Example of \pset .......................................................................... 149  
16.2 \psql variables ........................................................................ 150  
16.3 \pgaccess opening window .......................................................... 153  
16.4 \pgaccess table window ............................................................... 154  

17.1 Sample application being run ...................................................... 155  
17.2 Statename table ........................................................................ 156  
17.3 Libpq data flow .......................................................................... 156  
17.4 Libpq sample program ............................................................... 157  
17.5 libpgeasy sample program ......................................................... 159  
17.6 Ecpg sample program ................................................................. 160  
17.7 Libpq++ sample program ......................................................... 161  
17.8 JAVA sample program ............................................................... 164  
17.9 PERL sample program ............................................................... 165  
17.10 TCL sample program ............................................................... 165  
17.11 PYTHON sample program ....................................................... 166  
17.12 PHP sample program — Input ............................................... 166  
17.13 PHP sample program — Output ............................................... 167  

18.1 SQL flocc function ...................................................................... 170  
18.2 SQL tax function ........................................................................ 171
LIST OF FIGURES

18.3 Recreation of the part table ........................................ 171
18.4 SQL shipping function .................................................. 172
18.5 SQL function getstename ................................................. 173
18.6 Getting state name using join and function ...................... 173
18.7 PL/PGSQL version of getstename ...................................... 174
18.8 PL/PGSQL spread function ............................................. 175
18.9 PL/PGSQL getstatecode function .................................... 176
18.10 Calls to getstatecode function ...................................... 177
18.11 PL/PGSQL change_statename function ......................... 178
18.12 Example of change_statename() .................................... 179
18.13 Trigger creation ....................................................... 181
19.1 C ctof function ........................................................ 184
19.2 Create function ctof ..................................................... 184
19.3 Calling function ctof .................................................... 185
20.1 Examples of user administration ..................................... 188
20.2 Examples of database creation and removal ...................... 189
20.3 Making a new copy of database test ............................... 191
20.4 Postmaster and postgres processes ................................. 192
LIST OF FIGURES

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
List of Tables

3.1 Table friend ................................................................. 10

4.1 Common data types .......................................................... 19
4.2 Comparisons ................................................................. 27
4.3 LIKE comparison ............................................................. 29
4.4 Regular expression operators ............................................. 29
4.5 Regular expression special characters .................................. 30
4.6 Regular expression examples ............................................. 30
4.7 SET options ................................................................. 38
4.8 DATESTYLE output ......................................................... 38

5.1 Aggregates ................................................................. 41

7.1 Sequence number access functions ..................................... 67

9.1 POSTGRESQL data types .................................................. 90
9.2 Geometric types ........................................................... 92
9.3 Common functions ......................................................... 94
9.4 Common operators ........................................................ 95
9.5 Common variables ........................................................ 96

10.1 Visibility of single-query transactions ................................. 101
10.2 Visibility using multi-query transactions .............................. 102
10.3 Waiting for a lock .......................................................... 104
10.4 Deadlock ................................................................. 105

13.1 Temporary table isolation ............................................... 117

15.1 Backslashes understood by COPY ..................................... 145

16.1 psql query buffer commands ............................................ 147
16.2 psql general commands .................................................. 148
16.3 psql \pset options ......................................................... 148
16.4 psql output format shortcuts ............................................ 149
16.5 psql predefined variables .............................................. 151
16.6 psql listing commands .................................................... 152
16.7 psql large object commands ............................................ 152
16.8 psql command-line arguments ........................................ 153

17.1 Interface summary ....................................................... 155
LIST OF TABLES

20.1 Commonly used system tables ........................................ 194
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 ILLUSTRA\(^2\) 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 people heavily involved, MARC FOURNIER in Canada, THOMAS LOCKHART in Pasadena, California, VADIM MIKHEEV in Krasnoyarsk, Russia, and me in Philadelphia, Pennsylvania. We all had full-time jobs, so we did 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 single 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 POSTGRES 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 occurred 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 initially 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

---

*C 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 unused 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. A company formed to offer 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 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 expanded 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.

1.6 Summary

This chapter has explored the long history of POSTGRESQL, starting with its roots in university research. POSTGRESQL would not have grown to the success it is today without the Internet. The ability to communicate with people around the world has allowed a community of unpaid developers to enhance and support

\(^{7}\) 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.
software that rivals commercial database offerings. By allowing everyone to see the source code and con-
tribute, POSTGRESQL continues to improve every day. The remainder of this book shows how to use this
amazing piece of software.
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 psql interface. Additional interfaces are covered in Chapter 17.

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.
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. 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.

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

test=>
```

Figure 2.2: My first SQL query

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

¹A few operating systems are case-insensitive.
2.3. GETTING HELP

when you enter a semicolon (;) or backslash-g (\g). Here’s a good example. Let’s do SELECT 1 + 3; but in a different way. See figure 2.3. Notice the query is spread over three lines. Notice the prompt changed

```
test=> SELECT
1 test-> 1 + 3
2 test-> ;
3 ?column?
4 ----------------
5 (1 row)
6 test=>
```

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 psql to send the query to the server. We could easily have replaced the semicolon with backslash-g. I do not 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 does not 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 SELECT, then your computation, and finally a semicolon or backslash-g to finish. For example, 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 readline installed, psql will even allow you to use your arrow keys. Your left and right arrow keys allow you to move around, and the up and 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 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 do not 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 psql starts. Backslash-? (\?) prints all valid backslash commands. Backslash-h displays help for SQL commands. SQL commands are covered in the next chapter.

---

2 Don’t be concerned about ?column?. We will cover that in section 4.7.
3 Readline is an open-source library that allows powerful command-line editing.
test=> SELECT
2 * 10 + 1
SELECT
2 * 10 + 1
test-> \p
?column?
--------
  21
(1 row)

test=>

Figure 2.4: Backslash-p demo

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.

2.5 Summary

This chapter has shown how to use the most important features of psql. This knowledge will allow you to try all the examples in this book. However, psql has many features that can assist you. Section 16.1 covers psql in detail. You may want to review that chapter while reading through the book.
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.
CHAPTER 3. BASIC SQL COMMANDS

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.

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. You do not have to type it exactly like that. You could have used all lowercase, or you could have written it in one long line, and it would have worked just the same.

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

Figure 3.2: Create table friend

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 exactly. Otherwise, the database server will issue an error such as parser: 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.

\footnote{In 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 is an `INTEGER` 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.

`PGSQL` 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. Do not 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.

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

Figure 3.3: Example of backslash-d

### 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 do not 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 does not have quotes. It does not 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. Figure 3.5 shows the additional `INSERT`s needed to make the `friend` table match the three friends shown in table 3.1.

\(^2\)A character string is a group of characters strung together.
test=> INSERT INTO friend VALUES ('Mike', 'Nichols', 'Tampa', 'FL', 19);
INSERT 19053 1

Figure 3.4: INSERT into friend

insert friend

Figure 3.5: Additional friend INSERTs
3.4 Viewing Data with SELECT

You have just seen how to store data in the database. Now, let's show you how to retrieve that 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. We are going to use it to show the rows in the table friend. The query is shown in figure 3.6. In this case, I put the entire query on one line.

\[
\text{test=> SELECT * FROM friend;}
\]

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

(3 rows)

Figure 3.6: My first SELECT

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 indicates 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. This requires a 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 3.7. The query returns the rows that have an age column equal to 23. Figure 3.8 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 3.9 and 3.10. 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 3.11. If you had compared the firstname column to 'SAM' or 'sam', it would have returned no rows.
CHAPTER 3. BASIC SQL COMMANDS

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

Figure 3.7: My first WHERE

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

Figure 3.8: More complex WHERE clause

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

Figure 3.9: A single cell

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

Figure 3.10: A block of cells
3.6. REMOVING DATA WITH DELETE

Try a few more until you are comfortable.

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

Figure 3.11: Comparing string fields

3.6 Removing Data with DELETE

We now 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 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 3.12 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.

Continuing with our friend table, suppose Mike had a birthday, so we want to update his age in the table. Figure 3.13 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 3.14 shows the use of ORDER BY. You can reverse the order by adding DESC, as seen in figure 3.15. If the query were to use a WHERE clause too, the ORDER BY would appear after the WHERE clause, as in figure 3.16.

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.
CHAPTER 3. BASIC SQL COMMANDS

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

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

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

test=> DELETE FROM friend WHERE lastname = 'Barnes';
  DELETE FROM friend WHERE lastname = 'Barnes';

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

Figure 3.12: DELETE example

```
test=> UPDATE friend SET age = 20 WHERE firstname = 'Mike';
  UPDATE friend SET age = 20 WHERE firstname = 'Mike';

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

Figure 3.13: My first UPDATE
3.8. **SORTING DATA WITH 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: Use of 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: Reverse ORDER BY**

```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)

**Figure 3.16: Use of ORDER BY and WHERE**
3.9 Destroying Tables

This chapter would not be complete without showing how to delete tables. It is accomplished using the DROP TABLE command. The command \texttt{DROP TABLE friend} will remove the \texttt{friend} table. Both the table structure and the data contained in the table will be erased. We will be using the \texttt{friend} table in the next chapter, so I do not 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

This chapter has shown the basic operations of any database:

- Table creation (\texttt{CREATE TABLE})
- Table destruction (\texttt{DROP TABLE})
- Displaying (\texttt{SELECT})
- Adding (\texttt{INSERT})
- Replacing (\texttt{UPDATE})
- Removing (\texttt{DELETE})

This chapter has shown these commands in their simplest forms. Real-world queries are much more complex. The next chapters will show how these simple commands can be used to handle some very complicated tasks.
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 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\[1\] format. It also displays the time zone.

The final SELECT uses psql's \x display mode.\[2\] 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.

---

1. This is the format generated by typing the command date at the UNIX command prompt.
2. See section 16.1 for a full list of the psql backslash commands.
CHAPTER 4. CUSTOMIZING QUERIES

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

INSERT INTO alltypes
VALUES('PA',
    'Hilda Blairwood',
    3,
    10.7,
    4308.20,
    '9/8/1974',
    '9:00',
    '07/03/1996 10:30:00');
INSERT 19073 1

SELECT state, name, children, distance, budget FROM alltypes;
state | name            | children | distance | budget
------|-----------------|----------|----------|--------
PA    | Hilda Blairwood| 3        | 10.7     | 4308.20

SELECT born, checkin, started FROM alltypes;
born  | checkin        | started
-------|----------------|--------
1974-09-08 | 09:00:00 | 1996-07-03 10:30:00-04

Expanded display is on.

SELECT * FROM alltypes;
- [ RECORD 1 ]-----------------------------
state  | PA
name   | Hilda Blairwood
children | 3
distance | 10.7
budget  | 4308.20
born   | 1974-09-08
checkin | 09:00:00
started | 1996-07-03 10:30:00-04
```

Figure 4.1: Example of common data types
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 do not 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 appropriate 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 often confuses people. Why doesn’t the Mark row appear? The age is NULL or unknown, meaning the database does not 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 does not 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.
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>
<tr>
<td>Mark</td>
<td>Middleton</td>
<td>Indianapolis</td>
<td>IN</td>
<td></td>
</tr>
</tbody>
</table>

(4 rows)

test=> SELECT * FROM friend WHERE age > 0 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)

test=> SELECT * FROM friend WHERE age <= 99 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)

test=> SELECT * FROM friend WHERE age IS NULL 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>Mark</td>
<td>Middleton</td>
<td>Indianapolis</td>
<td>IN</td>
<td></td>
</tr>
</tbody>
</table>

(1 row)

Figure 4.3: NULL handling

test=> INSERT INTO friend
    test-> VALUES ('Jack', 'Burger', NULL, NULL, 27);

INSERT 19075 1

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

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

(0 rows)

Figure 4.4: Comparison of NULL fields
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

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

```

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

```

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

Figure 4.5: NULLs and blank strings

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.

### 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.
test=> CREATE TABLE account (  
  name CHAR(20),  
  balance NUMERIC(16,2) DEFAULT 0,  
  active CHAR(1) DEFAULT 'Y',  
  created TIMESTAMP DEFAULT CURRENT_TIMESTAMP  
);  
CREATE  

test=> INSERT INTO account (name)  
VALUES ('Federated Builders');  
INSERT 19103 1  
test=> SELECT * FROM account;  
  name   | balance | active | created  
----------------------+---------+--------+------------------------  
Federated Builders | 0.00 | Y | 2000-05-30 21:37:48-04  
(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 selected column. 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 does not have an appropriate label. Figure 4.8 shows the same query with an appropriate label added using AS.

```

test=> SELECT firstname AS buddy FROM friend ORDER BY buddy;  
  buddy  
-----------------  
   Cindy  
    Jack  
   Mark  
   Mike  
   Sam  
(5 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 slash-asterisk (/\*) and ends with asterisk-slash (*). 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, we have used only simple WHERE clause tests. In the following 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 inserted new people into the friend table, as shown 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'.

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.

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 state equals 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.
CHAPTER 4. CUSTOMIZING QUERIES

test=> DELETE FROM friend;
DELETE 6

DETECT INTO friend

VALUES ('Dean', 'Yeager', 'Plymouth', 'MA', 24);
INSERT 19744 1

VALUES ('Dick', 'Gleason', 'Ocean City', 'NJ', 19);
INSERT 19745 1

VALUES ('Ned', 'Millstone', 'Cedar Creek', 'MD', 27);
INSERT 19746 1

VALUES ('Sandy', 'Gleason', 'Ocean City', 'NJ', 25);
INSERT 19747 1

VALUES ('Sandy', 'Weber', 'Boston', 'MA', 33);
INSERT 19748 1

VALUES ('Victor', 'Tabor', 'Williamsport', 'PA', 22);
INSERT 19749 1

SELECT * FROM friend ORDER BY firstname;

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

Figure 4.10: New friends

test=> SELECT * FROM friend

WHERE firstname = 'Sandy' AND lastname = 'Gleason';

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

Figure 4.11: WHERE test for Sandy Gleason
test=> SELECT * FROM friend
  WHERE state = 'NJ' OR state = 'PA'
  ORDER BY firstname;

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

(3 rows)

Figure 4.12: Friends in New Jersey and Pennsylvania

```sql
test=> SELECT * FROM friend
  WHERE firstname = 'Victor' AND state = 'PA' OR state = 'NJ'
  ORDER BY firstname;

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

(3 rows)
```

Figure 4.13: Mixing ANDs and ORs

```sql
test=> SELECT * FROM friend
  WHERE firstname = 'Victor' AND (state = 'PA' OR state = 'NJ')
  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>Victor</td>
<td>Tabor</td>
<td>Williamsport</td>
<td>PA</td>
<td>22</td>
</tr>
</tbody>
</table>

(1 row)
```

Figure 4.14: Properly mixing ANDs and ORs

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

Table 4.2: Comparisons
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

\[
\text{test=> SELECT * FROM friend WHERE age >= 22 AND age <= 25 ORDER BY firstname;}
\]

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dean</td>
<td>Yeager</td>
<td>Plymouth</td>
<td>MA</td>
<td>24</td>
</tr>
<tr>
<td>Sandy</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
<td>25</td>
</tr>
<tr>
<td>Victor</td>
<td>Tabor</td>
<td>Williamsport</td>
<td>PA</td>
<td>22</td>
</tr>
</tbody>
</table>

(3 rows)

\[
\text{test=> SELECT * FROM friend WHERE age BETWEEN 22 AND 25 ORDER BY firstname;}
\]

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dean</td>
<td>Yeager</td>
<td>Plymouth</td>
<td>MA</td>
<td>24</td>
</tr>
<tr>
<td>Sandy</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
<td>25</td>
</tr>
<tr>
<td>Victor</td>
<td>Tabor</td>
<td>Williamsport</td>
<td>PA</td>
<td>22</td>
</tr>
</tbody>
</table>

(3 rows)

Figure 4.15: Selecting a range of values

<= 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 possible. 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 sign (%) 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 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 (___).
4.10. REGULAR EXPRESSIONS

```sql
  test-> SELECT * FROM friend
  test-> WHERE firstname LIKE 'D%'
  test-> ORDER BY firstname;
```

<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dean</td>
<td>Yeager</td>
<td>Plymouth</td>
<td>MA</td>
<td>24</td>
</tr>
<tr>
<td>Dick</td>
<td>Gleason</td>
<td>Ocean City</td>
<td>NJ</td>
<td>19</td>
</tr>
</tbody>
</table>

(2 rows)

Figure 4.16: Firstname begins with D.

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

Table 4.3: LIKE comparison

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 do not 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.

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 illustrates examples of queries using regular expressions. For a description, see the comment above each query.

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

\(^5\)Actually, POSTGRESQL regular expressions are like egrep extended regular expressions.
### Table 4.5: Regular expression special characters

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

### Table 4.6: Regular expression examples

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

```
4.10. REGULAR EXPRESSIONS

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 4.17: Regular expression sample queries
test=> -- firstname ends with n
test=> SELECT * FROM friend
    test-> WHERE firstname ˜ 'n *$'
    test-> ORDER BY firstname;
    
<table>
<thead>
<tr>
<th>firstname</th>
<th>lastname</th>
<th>city</th>
<th>state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dean</td>
<td>Yeager</td>
<td>Plymouth</td>
<td>MA</td>
<td>24</td>
</tr>
</tbody>
</table>

(1 row)

Figure 4.18: Complex regular expression queries
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 (\\). 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.

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 THEN 'adult'
        test-> ELSE 'minor'
        test-> END
        test-> FROM friend
        test-> ORDER BY firstname;

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

Figure 4.19: CASE example

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 exist only inside a single query, so they cannot be used outside the query that defines them.
CHAPTER 4. CUSTOMIZING QUERIES

```sql
select
    firstname,
    state,
    case
        when state = 'PA' then 'close'
        when state = 'NJ' or state = 'MD' then 'far'
        else 'very far'
    end as distance
from
    friend
order by
    firstname;
```

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

Figure 4.20: Complex CASE example

### 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
- Operators usually 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 standard precedence rules. Exponentiation is performed first, multiplication, division, and modulo second, and addition and subtraction are performed
4.13. FUNCTIONS AND OPERATORS

```
---
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
CHAPTER 4. CUSTOMIZING QUERIES

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

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

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

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

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

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

Figure 4.22: Function examples
Table 4.13: Operator examples

| Op | Left arg | Right arg | Result | Description |
|----+----------+-----------+--------+----------------------|
| !  | int2     |           | int4   |                       |
| !  | int4     |           | int4   | factorial             |
| !  | int8     |           | int8   | factorial             |
| !! |          | int2      | int4   |                       |

List of operators

| Op | Left arg | Right arg | Result | Description |
|----+----------+-----------+--------+----------------------|
| /  | box      | point     | box    | divide box by point (scale) |
| /  | char     | char      | char   | divide |
| /  | circle   | point     | circle | divide |
| /  | float4   | float4    | float4 | divide |

List of operators

| Op | Left arg | Right arg | Result | Description |
|----+----------+-----------+--------+----------------------|
| ^  | float8   | float8    | float8 | exponentiation (x^y) |

Object descriptions

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

SELECT 2 + 3 ^ 4;

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

Figure 4.23: Operator examples
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**.

<table>
<thead>
<tr>
<th>Function</th>
<th>SET option</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATESTYLE</td>
<td>DATESTYLE TO 'Postgres'</td>
</tr>
<tr>
<td>TIMEZONE</td>
<td>TIMEZONE TO 'value'</td>
</tr>
</tbody>
</table>

Table 4.7: **SET** options

**DATESTYLE** controls the appearance of dates when printed in **psql** as seen in table 4.8. It controls the **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 selected 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**.

See the **SET** manual page for a full list of **SET** options.

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.

4.15 Summary

This chapter has shown how simple commands can be enhanced using features like **DISTINCT**, **NULL**, and complex **WHERE** clauses. These features give users great control over how queries are executed. They were chosen by committees to be important features that should be in all **SQL** databases. While you may never use all the features listed in this chapter, many of them will be valuable when solving real-world problems.

---

6Your site defaults may be different.
4.15. SUMMARY

Figure 4.24: SHOW and RESET examples
Chapter 5

SQL Aggregates

Users often need 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.

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)

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

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

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

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

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

Figure 5.1: Aggregate examples
5.1. AGGREGATES

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.

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

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

Figure 5.3: Aggregate with GROUP BY

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 a user 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

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, mistakes are not a problem. 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.
5.5 Summary

Sometimes users want less output rather than more. They want a total, count, average, maximum, or minimum value for a column. Aggregates make this possible. They collect or aggregate data into fewer rows and send the result to the user.
Chapter 6
Joining Tables

This chapter will show 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 can be 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, \textit{f} is used as an alias for the \textit{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 \textit{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: \textit{customer}, \textit{employee}, \textit{salesorder}, and \textit{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 \textit{salesorder} table, we put the customer’s number in the \textit{salesorder} table. When we want to record which employee took the order, we put the employee’s number in the \textit{salesorder} table. When we want to record which part has been ordered, we put the part number in the \textit{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.

People might question the necessity of using 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:

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

The only time duplicate data should 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, e.g. 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 customer\_id, which is not very clear. The only way to define non-lowercase column and table names it to use double quotes. Double quotes preserve any capitalization you supply. You can even have spaces in table and column names if you surround the name with double quotes ("), e.g. "customer id". If you decide to use this feature, you must put double quotes around the table or column name every time it is referenced. 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. Do not 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.

---

1 The process of distributing data across multiple tables to prevent redundancy is called data normalization.
2 In 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.
3 A 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.
CHAPTER 6. JOINING TABLES

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

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

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
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.

```sql
test=> INSERT INTO customer VALUES (648,
     'Fleer Gearworks, Inc.',
     '1-610-555-782',
     '830 Winding Way',
     'Millersville',
     'AL',
     '35041',
     'USA');
INSERT 19815 1

test=> INSERT INTO employee VALUES (24,
     'Lee Meyers',
     '10/16/1989');
INSERT 19816 1

test=> INSERT INTO part VALUES (153,
     'Garage Door Spring',
     18.39);
INSERT 19817 1

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

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

\(^4\)Technically, 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.
Figure 6.5: Finding customer name using two queries

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 customer_id. Find the equal customer_id in the customer table.
- customer.name: Return name from the customer table.
6.5. THREE AND FOUR TABLE JOINS

You can see the database is performing the same steps as our 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 \texttt{customer\_id} exists in both tables mentioned in the \texttt{FROM} clause, \texttt{customer} and \texttt{salesorder}. If this were not done, the query would generate an error: \texttt{ERROR: Column 'customer\_id' is ambiguous}.

You can also 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 \texttt{FROM} clause and in the \texttt{WHERE} clause. The ordering of items is not important in these clauses.

\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;
   order_id
   ----------
       14673
   (1 row)
\end{verbatim}

Figure 6.7: Finding order number for customer name

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

Figure 6.8: Three-table join

The second column is the employee name. Both columns are labeled \texttt{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 \texttt{salesorder} table pointing to the other three tables.

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 \texttt{customer} and \texttt{employee} tables. The \texttt{salesorder} table is used to join the two tables but is not displayed. The \texttt{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.
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

\[
\begin{align*}  
\text{test}=> & \text{SELECT DISTINCT customer.name, employee.name} \\
& \text{FROM customer, employee, salesorder} \\
& \text{WHERE customer.customer_id = salesorder.customer_id and} \\
& \text{salesorder.employee_id = employee.employee_id} \\
& \text{ORDER BY customer.name, employee.name};  
\end{align*}
\]

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

Figure 6.10: Employees who have taken orders for customers.
6.6. ADDITIONAL JOIN POSSIBILITIES

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 unique 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

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

```
-- 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;
```

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 17.2 shows an example of a statename table. This

```sql
CREATE TABLE statename (code CHAR(2),
                         name CHAR(30))
CREATE TABLE statename VALUES ('AL', 'Alabama');
```

Figure 6.13: Statename table

shows two more uses for additional tables:

- Check codes against a list of valid values, i.e. only allow valid state codes
- Store code descriptions, i.e. state code and state name

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.

\(^5\)The United States Postal Service has assigned a unique two-letter code to each U.S. state.
• 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 referenced 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, FLE0001, 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, e.g. order numbers
• Items without appropriate codes, e.g. 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.}

5
test=> SELECT * FROM animal;
animal_id | name
-----------+-----------------
507 | rabbit
508 | cat
(2 rows)

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

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

Figure 6.15: One-to-many join
Figure 6.15 shows an example. Because the animal table’s 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 cat row in vegetable table, so the 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.

\(^6\)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.
This might seem like a concept of questionable utility, but it can prove useful. Figure 6.18 shows practical examples. For simplicity, results are not shown for these queries.

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

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

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

Figure 6.18: Examples of self-joins using table aliases

The first figure uses c as an alias for the customer table, and c2 as a secondary alias for customer. It finds all customers in the same zipcode as customer number 648. The second query 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 do not want customer 648 to appear in the result. The third query goes further 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 (<>) 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 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 often, but certain queries can only be performed using them.

6.12 Ordering Multiple Parts

Our customer, employee, part, and salesorder example has a serious limitation. It allows only one part_id per salesorder. 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

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.

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 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 a customer 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 specified.

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.\(^7\)

This new table layout illustrates the master/detail use of tables. The salesorder table is the master table because it holds information common to each order, such as customer and employee identifiers, and order date. The orderpart table is the 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 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 salesorder table. It goes directly to the orderpart table to find the parts making up the order, and joins to the 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 salesorder table to find the order number, and then join to the orderpart and 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 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 \textit{SUM} cost times \((\ast)\) quantity for each part in the order.

6.13 Primary and Foreign Keys

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

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 layouts many times as you improve them.

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 and refer to this chapter again for ideas. Do not be afraid to redesign everything. Redesign is hard, but when done properly, queries become 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.

\(^7\)In our example, changing 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 partprice table to store the part number, price, and effective date for the price.
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, a counter is created and set to approximately seventeen-thousand. 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.

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 19053 1. INSERT is the command that was executed, 19053 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 51 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. 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 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 21515 1
test=> SELECT oid, age FROM oidtest;
<table>
<thead>
<tr>
<th>oid</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>21515</td>
<td>7</td>
</tr>
</tbody>
</table>
(1 row)

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. Do not confuse a column of type OID with a 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

This section covers three limitations of object identification numbers.

Non-Sequential Numbering

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.

**Non-Modifiable**

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

**Not backed up by default**

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 20.5 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.\(^5\) 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 usually 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><code>nextval('name')</code></td>
<td>Returns the next available sequence number, and updates the counter</td>
</tr>
<tr>
<td><code>currval('name')</code></td>
<td>Returns the sequence number from the previous <code>nextval()</code> call</td>
</tr>
<tr>
<td><code>setval('name',newval)</code></td>
<td>Sets the sequence number counter to the specified value</td>
</tr>
</tbody>
</table>

Table 7.1: Sequence number access functions

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

---

\(^5\)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)

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 \texttt{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.

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

(3 rows)
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. The first NOTICE line indicates a sequence was created for the SERIAL column. Do not be concerned about

```sql
CREATE TABLE customer (  
customer_id SERIAL,  
name CHAR(30)  
);  
```

The first NOTICE line indicates a sequence was created for the SERIAL column. Do not be concerned about

```sql
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'
```

The second NOTICE line in the figure. Indexing is covered in section 11.1.

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 cannot 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 UNIION, 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 cannot 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

INSERT 19123 1

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

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

(2 rows)
```

Figure 8.2: Combining two tables with UNION

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

```sql
INSERT INTO aquatic_animal (name) VALUES ('penguin');
INSERT INTO terrestrial_animal (name) VALUES ('penguin');
```

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

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

(3 rows)

Figure 8.3: UNION with duplicates

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

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

(4 rows)

Figure 8.4: UNION ALL with duplicates

```sql
test=> SELECT name FROM aquatic_animal EXCEPT SELECT name FROM terrestrial_animal;
```

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

(1 row)

Figure 8.5: EXCEPT restricts output from the first SELECT
test-> SELECT name
  test-> FROM    aquatic_animal
  test-> INTERSECT
  test-> SELECT name
  test-> FROM    terrestrial_animal;

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

Figure 8.6: INTERSECT returns only duplicated rows

columns to occupy 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. While 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 from the previous chapters. 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.

Figure 8.7 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 mail order parts company in figures 6.3 and 6.4 on page 50. To find the customer name for order number 14673, we join the salesorder
8.2. SUBQUERIES

<table>
<thead>
<tr>
<th>first name</th>
<th>last name</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 8.7: Friends not in Dick Gleason’s state
and customer tables. This is shown as the first query in figure 8.8. The second query does not have a join,

```
test=> SELECT name
    test-> FROM   customer, salesorder
    test-> WHERE customer.customer_id = salesorder.customer_id AND
    test-> salesorder.order_id = 14673;
```

```
---------------
Fleer Gearworks, Inc.
(1 row)
```

```
name
----------
Fleer Gearworks, Inc.
(1 row)
```

Figure 8.8: Subqueries can replace some joins

but instead gets 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. A subquery’s 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
8.2. SUBQUERIES

\[
\begin{align*}
\text{SELECT} & \ f1.\text{firstname}, f1.\text{lastname}, f1.\text{age} \\
\text{FROM} & \ f1, f2 \\
\text{WHERE} & \ f1.\text{state} = f2.\text{state} \\
\text{GROUP BY} & \ f2.\text{state}, f1.\text{firstname}, f1.\text{lastname}, f1.\text{age} \\
\text{HAVING} & \ f1.\text{age} = \text{max}(f2.\text{age}) \\
\text{ORDER BY} & \ f1.\text{firstname}, f1.\text{lastname};
\end{align*}
\]

<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)

\[
\begin{align*}
\text{SELECT} & \ f1.\text{firstname}, f1.\text{lastname}, f1.\text{age} \\
\text{FROM} & \ f1 \\
\text{WHERE} & \ \text{age} = \left( \\
\text{SELECT} & \ \text{MAX}(f2.\text{age}) \\
\text{FROM} & \ f2 \\
\text{WHERE} & \ f1.\text{state} = f2.\text{state} \\
\right)
\text{ORDER BY} & \ f1.\text{firstname}, f1.\text{lastname};
\end{align*}
\]

<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
be evaluated for every row because the upper column value can change.

**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. For example, equality expects one value on the left of the = and one on the right, i.e. \( \text{col} = 3 \). Two special comparisons, \( \text{IN} \) and \( \text{NOT IN} \), allow multiple values to appear on the right-hand side. For example, the test \( \text{col} \ \text{IN} \ (1,2,3,4) \) compares \( \text{col} \) against four values. If \( \text{col} \) equals any of the four values, the comparison will return **true** and output the row. The test \( \text{col} \ \text{NOT IN} \ (1,2,3,4) \) will return true if \( \text{col} \) does **not** equal any of the four values.

An unlimited number of values can be specified on the right-hand side of an \( \text{IN} \) or \( \text{NOT IN} \) 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';
```

```
name-------------------
Lee Meyers
(1 row)
```

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

```
name-------------------
Lee Meyers
(1 row)
```

**Figure 8.10:** Employees who took orders

query uses a subquery. The subquery is evaluated, and generates a list of values used by \( \text{IN} \) 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 \( \text{NOT IN} \) 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

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

Figure 8.11: Customers who have no orders

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.

**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, 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 uppercol NOT IN (subquery) expands to uppercol NOT IN (1, 2, NULL). This further expands to uppercol <> 1 AND uppercol <> 2 AND uppercol <> 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 uppercol = 1 OR uppercol = 2 OR uppercol = 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 col < ANY(5,7,9) returns true if col is less than any of the three values. ALL requires all subquery values to compare as true, so col < 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 compare anything in the upper query, 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.
8.3. OUTER JOINS

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

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

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

Figure 8.13: NOT IN query rewritten using ALL and EXISTS

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.

In some cases, subqueries simply allow an additional way to phrase a query. In others, a subquery is the only way to produce the desired result.

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 customers 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 with no sales orders. The second query sets the ship_date equal to '11/16/96'
SELECT name, order_id
FROM customer, salesorder
WHERE customer.customer_id = salesorder.customer_id
UNION ALL
SELECT name, NULL
FROM customer
WHERE customer.customer_id NOT IN (SELECT customer_id FROM salesorder)
ORDER BY name;

Figure 8.14: Simulating outer joins

test=> DELETE FROM customer
  test-> WHERE customer_id NOT IN (test>
  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>
  test(>     SELECT customer_id
  test(>       FROM  customer
  test(>     WHERE  name = 'Fleer Gearworks, Inc.'
  test(>   );
UPDATE 1

Figure 8.15: Subqueries with UPDATE and DELETE
for all orders made by customer Fleer Gearworks, Inc. The numbers after DELETE and UPDATE indicate the number of rows affected by the queries.

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

Figure 8.16: UPDATE the order_date

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

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 customername as ‘Dean Yeager’. This is possible using trim()
and the || operator. Trim() removes trailing spaces. Two pipe symbols, ||, allow character strings to be joined together to form a single string, a process called concatenation. In this example, trim(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 query:

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

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

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 table names and thus control the column names in the new table. The other commands in the figure show the new table’s structure and contents.

SELECT...INTO tablename can also be written as CREATE TABLE tablename AS SELECT.... The above query can be rewritten as CREATE TABLE newfriend AS SELECT firstname, lastname, city, state FROM friend.
8.8 Summary

This chapter has shown how to combine queries in ways you probably never anticipated. It showed how queries could be chained, and placed inside other queries. It showed how FROM can be used by UPDATE, and how SELECT can create its own tables.

While these features are confusing, they are also very powerful. In most cases, you will need only the simplest features from this chapter. However, you may get that one-in-a-thousand request that requires one of the more complicated queries covered in this chapter. Hopefully this chapter was clear enough so you will recognize that query, and return to this chapter to refresh your memory.
CHAPTER 8. COMBINING SELECTS
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 were 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, sorting, 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 cannot 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" cannot be placed in that column. NULL can represent "I do not know her age."
The solution is to create an age_comments column of type CHAR() to hold comments which cannot 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 number types, all entered values must be surrounded by single quotes.
### Table 9.1: PostgreSQL data types

<table>
<thead>
<tr>
<th>Category</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character string</td>
<td>TEXT</td>
<td>variable storage length</td>
</tr>
<tr>
<td></td>
<td>VARCHAR(length)</td>
<td>variable storage length with maximum length</td>
</tr>
<tr>
<td></td>
<td>CHAR(length)</td>
<td>fixed storage length, blank-padded to length, internally BPCHAR</td>
</tr>
<tr>
<td>Number</td>
<td>INTEGER</td>
<td>integer, ±2 billion range, internally INT4</td>
</tr>
<tr>
<td></td>
<td>INT2</td>
<td>integer, ±32 thousand range</td>
</tr>
<tr>
<td></td>
<td>INT8</td>
<td>integer, ±4 × 10^{18} range</td>
</tr>
<tr>
<td></td>
<td>OID</td>
<td>object identifier</td>
</tr>
<tr>
<td></td>
<td>NUMERIC(precision, decimal)</td>
<td>number, user-defined precision and decimal location</td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td>floating-point number, 15-digit precision, internally FLOAT8</td>
</tr>
<tr>
<td></td>
<td>FLOAT4</td>
<td>floating-point number, 6-digit precision</td>
</tr>
<tr>
<td>Temporal</td>
<td>DATE</td>
<td>date</td>
</tr>
<tr>
<td></td>
<td>TIME</td>
<td>time</td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP</td>
<td>date and time</td>
</tr>
<tr>
<td></td>
<td>INTERVAL</td>
<td>interval of time</td>
</tr>
<tr>
<td>Logical</td>
<td>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>

7459
7460
7461
7462
7463
7464
7465
7466
7467
7468
7469
7470
7471
7472
7473
7474
7475
7476
7477
7478
7479
7480
7481
7482
7483
7484
7485
7486
7487
7488
7489
7490
7491
7492
7493
7494
7495
7496
7497
7498
7499
7500
7501
7502
7503
7504
7505
7506
7507
7508
7509
7510
7511
7512
7513
7514
7515
7516
7517
7518
7519
7520
7521
7522
7523
7524
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 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, since they provide better data validation, more compact storage, and better performance.

There are three character string data types: TEXT, VARCHAR(length), and CHAR(length). TEXT does not limit the number of characters 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 cannot add a number to the character string 763 because it does not 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 cannot sort it with other values and expect them to be in chronological order. The string 1/4/1998 is less than 3/8/1992 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 cannot be displayed by INTEGER fields, but are easily displayed in character strings.

Number

Number types allow the storage of numbers. The number 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, e.g. 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 number 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.

1ASCII 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 38. TIME allows storage of hour, minute, and second, separated by colons. TIMESTAMP represents storage of both date and time, e.g. 2000-7-12 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 using CAST

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 CAST to perform the conversion. To convert val to an INTEGER, use CAST(val AS INTEGER). To convert a column date_col of type DATE to type TEXT, use CAST(date_col AS TEXT). You can also perform type casting using double-colons, i.e. date_col::text or num_val::numeric(10,2).

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 9.1 shows an example of the use of the upper function.

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

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

Figure 9.1: Example of a function call

There are many functions available. Table 9.3 shows the most common ones, organized by the data types they support. Psq's \df shows all defined functions and their arguments. Section 16.1 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 9.2. In the first query, 5/8/1971 is a character string, not a date. The second query converts 5/8/1971 to a date so date_part() can be used.

9.5 Support Operators

Operators are similar to functions, and are covered in section 4.13 on page 34. Table 9.4 shows the most common operators. Psq'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 9.3.
<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>substring(...)</td>
<td>substring(col FROM pos)</td>
<td>col starting at position pos for length</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>Number</td>
<td>round()</td>
<td>round(col)</td>
<td>round to an integer</td>
</tr>
<tr>
<td></td>
<td>round()</td>
<td>round(col, len)</td>
<td>NUMERIC() col rounded to len decimal places</td>
</tr>
<tr>
<td></td>
<td>trunc()</td>
<td>trunc(col)</td>
<td>truncate to an integer</td>
</tr>
<tr>
<td></td>
<td>trunc()</td>
<td>trunc(col, len)</td>
<td>NUMERIC() col truncated to len decimal places</td>
</tr>
<tr>
<td></td>
<td>abs()</td>
<td>abs(col)</td>
<td>absolute value</td>
</tr>
<tr>
<td></td>
<td>factorial()</td>
<td>factorial(col)</td>
<td>factorial</td>
</tr>
<tr>
<td></td>
<td>sqrt()</td>
<td>sqrt(col)</td>
<td>square root</td>
</tr>
<tr>
<td></td>
<td>cbrt()</td>
<td>cbrt(col)</td>
<td>cube root</td>
</tr>
<tr>
<td></td>
<td>exp()</td>
<td>exp(col)</td>
<td>exponential</td>
</tr>
<tr>
<td></td>
<td>ln()</td>
<td>ln(col)</td>
<td>natural logarithm</td>
</tr>
<tr>
<td></td>
<td>log()</td>
<td>log(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>isfinite()</td>
<td>isfinite(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>see psql’s df for a list of geometric functions</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>host address of col</td>
</tr>
<tr>
<td></td>
<td>masklen()</td>
<td>masklen(col)</td>
<td>netmask of col</td>
</tr>
<tr>
<td></td>
<td>network()</td>
<td>network(col)</td>
<td>mask length of col</td>
</tr>
<tr>
<td></td>
<td></td>
<td></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 date_part('year', '5/8/1971');
  ERROR: Function 'date_part(unknown, unknown)' does not exist
          Unable to identify a function that satisfies the given argument types
  You may need to add explicit typecasts
  test=> SELECT date_part('year', CAST('5/8/1971' AS DATE));

                date_part
  -----------
           1971

(1 row)
```

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

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

Table 9.4: Common operators
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 9.4 creates a table with one-, two-, and three-dimensional INTEGER columns. The first and last columns have sizes specified.

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

Figure 9.4: 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, \(\text{col2}\), 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 \(\text{array_test}\) 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 \(2 \times 2\) matrices. Arrays of any size can be constructed.

Figure 9.5 shows a query inserting values into \(\text{array_test}\), and several queries selecting data from the table. Brackets are used to access individual array elements.

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

Figure 9.5: Using arrays

Any data type can be used as an array. If individual elements of the array are accessed or updated frequently, it is better to use separate columns or tables rather than arrays.
9.8 Large Objects (BLOBS)

`POSTGRESQL` cannot 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 `lo_import()`, and retrieve the file from the database using `lo_export()`. Figure 9.6 shows an example that stores a fruit name and image. `Lo_import()` stores

```
test=> CREATE TABLE fruit (name CHAR(30), image OID);
CREATE
```

```
test=> INSERT INTO fruit
  VALUES ('peach', lo_import('/usr/images/peach.jpg'));
INSERT 27111 1
```

```
test=> SELECT lo_export(fruit.image, '/tmp/outimage.jpg')
  FROM fruit
  WHERE name = 'peach';
lo_export
-----------
1
(1 row)
```

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

Figure 9.6: Using large images

`/usr/images/peach.jpg` into the database. The function call returns an OID which is used to refer to the imported large object. The OID value is stored in fruit.image. `Lo_export()` uses the OID value to find the large object stored in the database, and places the image into the new file `/tmp/outimage.jpg`. The 1 returned by `lo_export()` indicates a successful export. `Lo_unlink()` removes large objects.

Full pathnames must be used with large objects because the database server is running in a different directory than the `psql` client. Files are imported and exported by the `postgres` user, so `postgres` must have permission to read the file for `lo_import()`, and directory write permission for `lo_export()`. Because large objects use the local filesystem, users connecting over a network cannot use `lo_import` and `lo_export()`. They can use `psql`'s `\lo_import` and `\lo_export` commands.

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. Do not 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 cannot 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

**Figure 10.2: INSERT with explicit transaction**

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

 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 the 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.

**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 to any account. 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 queries as a single operation.

**Figure 10.4: Multi-statement transaction**

test=> BEGIN WORK;
BEGIN

```sql
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
```
10.3. VISIBILITY OF COMMITTED TRANSACTIONS

When you begin a transaction with `BEGIN WORK`, you do not 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 cannot be executed due to an error, the entire transaction is automatically rolled back.

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

Figure 10.5: Transaction rollback

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 user 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 transferred $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 transferred. 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.

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

Table 10.1: Visibility of single-query transactions
### 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 returns 0</td>
</tr>
<tr>
<td>INSERT INTO trans_test VALUES (1)</td>
<td>SELECT (*) FROM trans_test</td>
<td>add row to trans_test returns 1</td>
</tr>
<tr>
<td>SELECT (*) FROM trans_test</td>
<td>SELECT (*) FROM trans_test</td>
<td>returns 0</td>
</tr>
<tr>
<td>COMMIT WORK</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(*) FROM trans_test;`

```
test=> BEGIN WORK;
BEGIN
  test=> SELECT COUNT(*) FROM trans_test;
  count
  -------
     5
     (1 row)
  test=> --
  test=> -- someone commits INSERT INTO trans_test
  test=> --
  test=> SELECT COUNT(*) FROM trans_test;
  count
  -------
     6
     (1 row)

  test=> COMMIT WORK;
COMMIT
```

Figure 10.6: Read-committed isolation level

COUNT(*). Then, while sitting at a psql prompt, someone INSERTs into the table. The next SELECT 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.

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

-- someone commits INSERT INTO trans_test

--

SELECT COUNT(*) FROM trans_test;
  count
  -------
    5
  (1 row)

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 do not 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-
**CHAPTER 10. TRANSACTIONS AND LOCKS**

<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>

Table 10.3: Waiting for a lock

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

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

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

Figure 10.8: SELECT with no locking

problem is another user can modify the James row between the SELECT and UPDATE. To prevent this, you can use SERIALIZABLE isolation. However, 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 cannot modify the James row between the SELECT...FOR UPDATE and UPDATE. In fact, the row remains locked until the transaction ends.

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 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.

<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 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></td>
<td>auto-ROLLBACK WORK</td>
<td>Attempt to get write lock held by transaction 1</td>
</tr>
<tr>
<td>COMMIT WORK</td>
<td></td>
<td>Deadlock detected — transaction 2 automatically rolled back</td>
</tr>
</tbody>
</table>

Transaction 1 returns from UPDATE and commits

Table 10.4: Deadlock

10.7 Summary

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 simultaneous operations.
Queries cannot 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 need to be concerned about performance. The system would tune itself. However, databases do not 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 improve performance. This chapter shows how to get optimal performance from your database.

11.1 Indexes

When accessing a table, POSTGRESQL normally reads from the beginning of the table to the end, looking for relevant rows. With an index, POSTGRESQL can quickly find specific values in the index, and go directly to matching rows. Indexes allow fast retrieval of specific rows from a table.

For example, consider the query 

```sql
SELECT * FROM customer WHERE col = 43.
```

Without an index, POSTGRESQL must scan the entire table looking for rows where `col` equals 43. With an index on `col`, POSTGRESQL can go directly to rows where `col` equals 43, bypassing all other rows.

For a large table, it can take minutes to check every row. Using an index, finding a specific row takes fractions of a second.

Internally, POSTGRESQL stores data in operating system files. Each table has its own file. Data 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.

However, POSTGRESQL does not create indexes automatically. Users should create them for columns frequently used in `WHERE` clauses.

Indexes are created using the `CREATE INDEX` command, as shown in figure 11.1. In this example,

```sql
CREATE INDEX customer_custid_idx ON customer (customer_id);
```

**Figure 11.1: Example of CREATE INDEX**

customer_custid_idx is the name of the index, customer 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. `customer_customer_id_idx` or `i_customer_custid`. This index is only useful for finding rows in customer for specific `customer_ids`. It cannot 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. Also, performance can suffer with too many indexes because row changes require an update to each index.
It is possible to create an index spanning multiple columns. Multi-column indexes are sorted by the first indexed column. When the first column has several equal values, sorting continues using the second indexed column. Multi-column indexes are only useful on columns with many duplicate values.

The command CREATE INDEX customer_age_gender_idx ON customer (age, gender) creates an index which is sorted by age, and when several age rows have the same value, then sorted on gender. This index can be used by the query SELECT * FROM customer WHERE age = 36 AND gender = 'F' and the query SELECT * FROM customer WHERE age = 36.

However, index customer_age_gender_idx is useless if you wish to find rows based only on gender. The gender component of the index can be used only after the age value has been specified. The query SELECT * FROM customer WHERE gender = 'F' cannot use the index because there is no restriction on age, which is the first part of the index.

Indexes can be useful for columns involved in joins too. An index can even be used to speed up some ORDER BY clauses.

Indexes are removed using the DROP INDEX command. See the CREATE_INDEX and DROP_INDEX manual pages 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 a unique index. The index is unique because the keyword UNIQUE was used. The remaining queries try to insert a duplicate value. The unique index prevents this and displays an appropriate error message.

Sometimes unique indexes are created only to prevent duplicate values, and not for performance reasons. Multi-column unique indexes ensure the combination of indexed columns remains unique. Unique indexes allow multiple NULL values. 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 when performance is critical, and the indexed column has many duplicate values.

For example, suppose column customer.age has many duplicate values, and the query SELECT * FROM customer WHERE age = 98 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.
CLUSTER even helps with range queries like \( \text{col} \geq 3 \) AND \( \text{col} \leq 5 \). CLUSTER places these rows next to each other on disk, speeding indexed lookups.

CLUSTER can also speed \textit{ORDER BY} processing. See the CLUSTER manual page for more information.

### 11.4 Vacuum

When \textsc{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 used by other transactions still viewing the database in its prior state. Deletions are similarly marked as expired, but not removed from the table file.

The \texttt{VACUUM} command removes expired rows from the file. While it removes them, it moves rows from the end of the table into the expired spots, thereby compacting the table file.

The \texttt{VACUUM} command should be run periodically to clean out expired rows. For tables that are heavily modified, it is useful to run \texttt{VACUUM} every night in an automated manner. For tables with few modifications, \texttt{VACUUM} should be run only periodically. \texttt{VACUUM} exclusively locks the table while processing.

There are two ways to run \texttt{VACUUM}. \texttt{VACUUM} alone vacuums all tables in the database. \texttt{VACUUM \textit{tablename}} vacuums a single table.

### 11.5 Vacuum Analyze

The \texttt{VACUUM ANALYZE} command is like \texttt{VACUUM}, except it also collects statistics about each column’s proportion of duplicate values and the maximum and minimum values. This information is used by \textsc{postgresql} when deciding how to efficiently execute complex queries. \texttt{VACUUM ANALYZE} should be run when a table is initially loaded, and when the table data dramatically changes.

The \texttt{VACUUM} manual page shows all of the \texttt{VACUUM} options.

### 11.6 EXPLAIN

\texttt{EXPLAIN} causes \textsc{postgresql} to display how a query will be executed, rather than executing it. For example, figure 11.3 shows a \texttt{SELECT} query preceded by the word \texttt{EXPLAIN}. In the figure, \textsc{postgresql} reports a

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

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

\texttt{EXPLAIN}

Figure 11.3: Using \texttt{EXPLAIN}

\textit{sequential scan} will be used on \textit{customer}, meaning it will scan the entire table. \textit{Cost} is an estimate of the work required to execute the query. The numbers are only meaningful for comparison. \textit{Rows} indicates the number of rows it expects to return. \textit{Width} is the number of bytes per row.

Figure 11.4 shows more interesting examples of \texttt{EXPLAIN}. The first \texttt{EXPLAIN} shows a \texttt{SELECT} with the restriction \texttt{customer\_id = 55}. This is again a \textit{sequential scan}, but the restriction causes \textsc{postgresql} to estimate ten rows will be returned. A \texttt{VACUUM ANALYZE} is run, causing the next query to properly estimate one row will be returned instead of ten. An index is created, and the query rerun. This time, an \textit{index scan}
test=> EXPLAIN SELECT customer_id FROM customer WHERE customer_id = 55;
NOTICE: QUERY PLAN:

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

EXPLAIN

test=> VACUUM ANALYZE customer;
VACUUM

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

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

EXPLAIN

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

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

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

EXPLAIN

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

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

EXPLAIN

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

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

EXPLAIN

Figure 11.4: More complex EXPLAIN examples
is used, allowing PostgreSQL to go directly to the rows where customer_id equals 55. The next one shows a query with no WHERE restriction. PostgreSQL realizes the index is of no use and performs a sequential scan. The last query has an ORDER BY that matches an index, so PostgreSQL uses an index scan.

Even more complex queries can be studied using EXPLAIN, as shown in figure 11.5. In this example,

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

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

EXPLAIN

Figure 11.5: EXPLAIN example using joins

$tab1$ and $tab2$ are joined on $col1$ and $col2$. Each table is sequentially 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 their use is not required, they can produce huge improvements in query speed. Section 20.8 outlines more steps database administrators can take to improve performance.
CHAPTER 11. PERFORMANCE
Chapter 12

Controlling Results

When a SELECT query is issued from \texttt{psql}, it travels to the POSTGRESQL server, is executed, and the result sent back to \texttt{psql} to be displayed. POSTGRESQL allows fine-grained control over which rows are returned. This chapter explores the methods available.

12.1 LIMIT

The LIMIT and OFFSET clauses of SELECT allow the user to specify which rows should be returned. For example, suppose \texttt{customer} has 1000 rows with \texttt{customer\_id} values from 1 to 1000. Figure 12.1 shows queries using LIMIT and LIMIT...OFFSET. The first query sorts the table by \texttt{customer\_id} and uses LIMIT to

\begin{verbatim}
test=> SELECT customer\_id FROM customer ORDER BY customer\_id LIMIT 3;
customer\_id
-----------
     1
     2
     3
(3 rows)

test=> SELECT customer\_id FROM customer ORDER BY customer\_id LIMIT 3 OFFSET 997;
customer\_id
-----------
     998
     999
    1000
(3 rows)
\end{verbatim}

Figure 12.1: Examples of LIMIT and LIMIT/OFFSET

return the first three rows. The second query is similar, except it skips to the 997th row before returning three rows.

Notice each query uses ORDER BY. While this is not required, LIMIT without ORDER BY returns random rows from the query, which is useless.

LIMIT improves performance because it reduces the number of rows returned to the client. If an index matches the ORDER BY, sometimes LIMIT can even produce correct results without executing the entire query.
12.2 Cursors

Ordinarily, all rows generated by a SELECT are returned to the client. Cursors allow a SELECT query to be named, and individual result rows fetched as needed by the client.

Figure 12.2 shows an example of cursor usage. Notice cursor activity must take place inside a transaction. Cursors are declared using DECLARE…CURSOR FOR SELECT…. Result rows are retrieved using FETCH. MOVE allows the user to move the cursor position. CLOSE releases all rows stored in the cursor. See the DECLARE, FETCH, MOVE, and CLOSE manual pages for more information.

12.3 Summary

LIMIT specifies which rows to return. Cursors allow dynamic row retrieval. The difference between LIMIT and cursors is that LIMIT specifies the rows as part of the SELECT, while cursors allow dynamic fetching of rows. LIMIT and cursors offer new ways to tailor your queries so you get exactly the results you desire.
12.3. SUMMARY

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

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

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

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

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

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

 test=> CLOSE customer_cursor;
CLOSE
 test=> COMMIT WORK;
COMMIT
```

Figure 12.2: Cursor usage
Chapter 13

Table Management

This chapter covers a variety of topics involved in managing SQL tables.

13.1 Temporary Tables

Temporary tables are short-lived tables. They exist only for the duration of a database session. When a database session terminates, its temporary tables are automatically destroyed. Figure 13.1 illustrates this.

In the figure, CREATE TEMPORARY TABLE creates a temporary table. On psql exit, the temporary table is destroyed. Restarting psql shows the temporary table no longer exists.

Temporary tables are visible only to the session that creates them. They are invisible to other users. In fact, several users can create temporary tables with the same name, and each user sees only their version of the table. Table 13.1 shows an example of this. Temporary tables will even mask ordinary tables with the same name.

Table 13.1: Temporary table isolation

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

Temporary tables are ideal for holding intermediate data used by the current SQL session. For example, suppose you need to do many SELECTs on the result of a complex query. An efficient way to do this is to execute the complex query once, and store the result in a temporary table.

Figure 13.2 shows an example of this. It uses SELECT … INTO TEMPORARY TABLE to collect all Pennsylvania customers into a temporary table. It also creates a temporary index on the temporary table. Customer_pennsylvania can then be used in subsequent SELECT queries. Multiple users can do this at the same time with the same temporary names without fear of collision.

13.2 ALTER TABLE

ALTER TABLE allows the following operations:

- rename tables
- rename columns
$ psql test
Welcome to psql, the PostgreSQL interactive terminal.

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

test=> CREATE TEMPORARY TABLE temptest(col INTEGER);
CREATE
test=> SELECT * FROM temptest;
  col
-----
(0 rows)
test=> \q
$s psql test
Welcome to psql, the PostgreSQL interactive terminal.

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

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

Figure 13.1: Temporary table auto-destruction

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

Figure 13.2: Example of temporary table use
13.3. GRANT AND REVOKE

When a table is created, only the owner can access it. If the owner wants others to have access, the table’s permissions must be changed using the GRANT command. Figure 13.4 shows some examples of GRANT. Available privileges are SELECT, UPDATE, DELETE, RULE, and ALL. Rules are covered later in section 13.6.

REVOKE removes permissions from a table. See the GRANT and REVOKE manual pages for more information.
CHAPTER 13. TABLE MANAGEMENT

13.4 Inheritance

Inheritance allows the creation of a new table related to an existing table. Figure 13.5 shows the creation of an inherited table. Using inheritance, the child table gets all the columns of the parent, plus the additional columns it defines. In the example, child_test gets col1 from parent_test, plus the column col2.

Inheritance also links rows in parent and child tables. If the parent table is referenced with an asterisk suffix, rows from the parent and all children are accessed. Figure 13.6 shows insertion into two tables related by inheritance. The figure then shows that while parent_test access only the parent_test rows, parent_test* accesses both parent_test and child_test rows. Parent_test* accesses only columns common to all tables. Child_test.col2 is not in the parent table so it is not displayed. Figure 13.7 shows inherited tables can be layered on top of each other.
13.4. INHERITANCE

```sql
test=> INSERT INTO parent_test VALUES (1);
INSERT 18837 1

test=> INSERT INTO child_test VALUES (2,3);
INSERT 18838 1

test=> SELECT * FROM parent_test;
col1
------
  1
(1 row)

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

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

Figure 13.6: Accessing inherited tables

```sql
test=> CREATE TABLE grandchild_test (col3 INTEGER) INHERITS (child_test);
CREATE

test=> INSERT INTO grandchild_test VALUES (4, 5, 6);
INSERT 18853 1

test=> SELECT * FROM parent_test*;
col1
------
  1
  2
  4
(3 rows)

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

Figure 13.7: Inheritance in layers
Consider a practical example that records information about employees and managers. Table `employee` can hold information about non-managerial employees. `Manager` can hold information about managers. `Manager` can inherit all the columns from `employee`, and have additional columns. You can then access non-managerial employees using `employee`, managers using `manager`, and all employees including managers using `employee*`.

### 13.5 Views

Views are pseudo-tables. They are not real tables, but appear as ordinary tables to `SELECT`. Views can represent a subset of a real table. A view can select certain columns or certain rows from an ordinary table. Views can even represent joined tables. Because views have separate permissions, they can be used to restrict table access so users see only specific rows or columns of a table.

Views are created using the CREATE VIEW command. Figure 13.8 shows the creation of several views. The view `customer_ohio` selects only customers from Ohio. SELECTS on it will show only Ohio customers.

```
test=> CREATE VIEW customer_ohio AS
  test-> SELECT *
  test-> FROM customer
  test-> WHERE state = 'OH';
CREATE 18908 1
```

User `sanders` is then given SELECT access to the view. `Customer_address` will show only address information. `Customer_finance` is a join of `customer` and `finance`, showing columns from both tables.

DROP VIEW removes a view. Because views are not ordinary tables, INSERTS, UPDATES, and DELETES on views have no effect. The next section shows how rules can correct this.
13.6 Rules

Rules allow actions to take place when a table is accessed. Rules can modify the effect of SELECT, INSERT, UPDATE, and DELETE.

Figure 13.9 shows a rule that prevents INSERTs into a table. The INSERT rule is named `ruletest_insert` and

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

the action is NOTHING. NOTHING is a special rule keyword that does nothing.

There are two types of rules. DO rules perform SQL commands in addition to the submitted query. DO INSTEAD rules replace the user query with the rule action.

Figure 13.10 shows how rules can track table changes. The figure creates `service_request` to hold current service requests, and `service_request_log` to record changes in the `service_request` table. The figure also creates two DO rules on `service_request`. Rule `service_request_update` causes an INSERT into `service_request_log` each time `service_request` is updated. The special keyword `old` is used to insert the pre-UPDATE column values into `service_request_log`. The keyword `new` would refer to the new query values. The second rule tracks deletions to `service_request` by inserting into `service_request_log`. To distinguish updates from deletes in `service_request_log`, updates are inserted with a `mod_type` of 'U', and deletes with a `mod_type` of 'D'.

`DEFAULT` was used for the username and timestamp fields. A column's default value is used when an INSERT does not supply a value for the column. In this example, defaults allow auto-assignment of these values on INSERT to `service_request`, and on rule INSERTS to `service_request_log`.

Figure 13.11 shows these rules in use. A row is inserted, updated, and deleted from `service_request`. A SELECT on `service_request_log` shows the UPDATE rule recorded the pre-UPDATE values, a 'U' in `mod_type`, and the user, date and time of the UPDATE. The DELETE appears similarly.

While views ignore INSERT, UPDATE and DELETE, rules can be used to properly handle them. Figure 13.12 shows the creation of a table and view on the table. The figure also illustrates views ignore INSERTS. UPDATES and DELETES are similarly ignored.

Figure 13.13 shows the creation of DO INSTEAD rules to properly handle INSERT, UPDATE, and DELETE. This is done by changing INSERT, UPDATE, and DELETE queries on the view to queries on `realtable`. Notice `new` is used by the INSERT rule to reference the new value to be inserted. In UPDATE and DELETE, `old` is used to reference old values. Figure 13.14 shows the view now properly handles modifications. It would be wise to add an index on `col` because the rules do lookups on that column.

SELECT rules can also be created. Views are implemented internally as SELECT rules. Rules can even be applied to only certain rows. Rules are removed with the DROP RULE command. See the CREATE RULE and DROP RULE manual pages for more information.
test=> CREATE TABLE service_request (customer_id INTEGER,
  description text,
  cre_user text DEFAULT CURRENT_USER,
  cre_timestamp timestamp DEFAULT CURRENT_TIMESTAMP);

CREATE

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

CREATE

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

CREATE 19670 1

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

CREATE 19671 1

Figure 13.10: Rules to log table changes
13.6. RULES

```sql
test=> INSERT INTO service_request (customer_id, description)
VALUES (72321, 'Fix printing press');
INSERT 18808 1

test=> UPDATE service_request
SET description = 'Fix large printing press'
WHERE customer_id = 72321;
UPDATE 1

test=> DELETE FROM service_request
WHERE customer_id = 72321;
DELETE 1

test=> SELECT *
FROM service_request_log
WHERE customer_id = 72321;

<table>
<thead>
<tr>
<th>customer_id</th>
<th>description</th>
<th>mod_type</th>
<th>mod_user</th>
<th>mod_timestamp</th>
</tr>
</thead>
</table>

(2 rows)
```

Figure 13.11: Use of rule to log table changes

```sql
CREATE TABLE realtable (col INTEGER);
CREATE TABLE realtable (col INTEGER);
CREATE VIEW view_realtable AS SELECT * FROM realtable;
CREATE VIEW view_realtable AS SELECT * FROM realtable;

INSERT INTO realtable VALUES (1);
INSERT INTO realtable VALUES (1);
INSERT INTO view_realtable VALUES (2);
INSERT INTO view_realtable VALUES (2);
SELECT * FROM realtable;
SELECT * FROM realtable;
SELECT * FROM realtable;
SELECT * FROM realtable;

<table>
<thead>
<tr>
<th>col</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

(1 row)

```sql
SELECT * FROM view_realtable;
SELECT * FROM view_realtable;
SELECT * FROM view_realtable;
SELECT * FROM view_realtable;

<table>
<thead>
<tr>
<th>col</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

(1 row)
```

Figure 13.12: Views ignore table modifications
test=> CREATE RULE view_realtable_insert AS -- INSERT rule
  test=> ON INSERT TO view_realtable
  test=> DO INSTEAD
  test=> INSERT INTO realtable
  test=> VALUES (new.col);
CREATE 407894 1

CREATE 407901 1

Figure 13.13: Rules to handle view modifications

Creating a rule whose action performs the same command on the same table causes an infinite loop. POSTGRESQL will call the rule again and again from the rule action. For example, if an UPDATE rule on ruletest has a rule action that also performs an UPDATE on ruletest, an infinite loop is created. POSTGRESQL will detect the infinite loop and return an error.

Fortunately, POSTGRESQL also supports triggers. Triggers allow actions to be performed when a table is modified. They can perform actions that cannot be implemented using rules. See section 18.4 for information about using triggers.

13.7 LISTEN and NOTIFY

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 then issues a NOTIFY command, all registered listeners will be notified. For more information, see the LISTEN and NOTIFY manual pages.

13.8 Summary

This chapter has covered features that give administrators and users new capabilities in managing database tables. The next chapter covers restrictions that can be placed on table columns to improve data management.
13.8. SUMMARY

```
  test=> INSERT INTO view_realtable VALUES (3);
  INSERT 407895 1
  test=> SELECT * FROM view_realtable;
  col
     ----
     1
     3
  (2 rows)
  test=> UPDATE view_realtable
  test=> SET col = 4;
  UPDATE 2
  test=> SELECT * FROM view_realtable;
  col
     ----
     4
     4
  (2 rows)
  test=> DELETE FROM view_realtable;
  DELETE 2
  test=> SELECT * FROM view_realtable;
  col
     ----
  (0 rows)
```

Figure 13.14: Rules handle view modifications
Chapter 14

Constraints

Constraints keep user data *constrained*. They help prevent invalid data from being entered into the database. Defining a data type for a column is a constraint itself. A column of type DATE constrains the column to valid dates.

This chapter covers a variety of constraints. We have already shown DEFAULT can be specified at table creation. Constraints are defined at table creation in a similar way.

### 14.1 NOT NULL

The constraint NOT NULL prevents NULLs from appearing in a column. Figure 14.1 shows the creation of a table with a NOT NULL constraint. Insertion of a NULL value, or an INSERT that would set col2 to NULL, will cause the INSERT to fail. The figure shows UPDATE of a NULL value also fails.

Figure 14.1 adds a DEFAULT value for col2. This allows INSERTs that do not specify a value for col2, as illustrated in the figure.
14.2 UNIQUE

The UNIQUE constraint prevents duplicate values from appearing in the column. UNIQUE columns can contain multiple NULL values however. UNIQUE is implemented by creating a unique index on the column. Figure 14.3 shows that UNIQUE prevents duplicates. CREATE TABLE displays the name of the unique index it creates. The

```sql
CREATE TABLE uniquetest (col1 INTEGER UNIQUE);
```

```sql
NOTICE: CREATE TABLE/UNIQUE will create implicit index 'uniquetest_col1_key' for table 'uniquetest'
```

```sql
CREATE TABLE uniquetest (col1 INTEGER UNIQUE);
```

```sql
NOTICE: CREATE TABLE/UNIQUE will create implicit index 'uniquetest_col1_key' for table 'uniquetest'
```

```sql
CREATE TABLE uniquetest (col1 INTEGER UNIQUE);
```

```sql
NOTICE: CREATE TABLE/UNIQUE will create implicit index 'uniquetest_col1_key' for table 'uniquetest'
```

```sql
CREATE TABLE uniquetest (col1 INTEGER UNIQUE);
```

```sql
NOTICE: CREATE TABLE/UNIQUE will create implicit index 'uniquetest_col1_key' for table 'uniquetest'
```
Instead, a separate UNIQUE line is required to specify the columns that make up the constraint. This is called a UNIQUE table constraint. Figure 14.4 shows a multi-column UNIQUE constraint. While col1 or col2 themselves may not be unique, the constraint requires the combination of col1 and col2 to be unique. For example, in a table that contains the driver's license numbers of people in various states, two people in different states may have the same license number, but the combination of their state and license number should always be unique.

### 14.3 PRIMARY KEY

The PRIMARY KEY constraint marks the column that uniquely identifies each row. It is a combination of UNIQUE and NOT NULL constraints. UNIQUE prevents duplicates, and NOT NULL prevents NULL values in the column. Figure 14.5 shows the creation of a PRIMARY KEY column. Notice an index is created automatically,

```
test=> CREATE TABLE primarytest (col INTEGER PRIMARY KEY);
NOTICE: CREATE TABLE/PRIMARY KEY will create implicit index 'primarytest_pkey' for table 'primarytest'
```

and the column defined as NOT NULL.

Just as with UNIQUE, a multi-column PRIMARY KEY constraint must be specified on a separate line. Figure 14.6 shows an example of this. It shows col1 and col2 are combined to form a primary key.

There cannot be more than one PRIMARY KEY specification per table. PRIMARY KEYS have special meaning when using foreign keys, which are covered in the next section.

### 14.4 FOREIGN KEY/REFERENCES

Foreign keys are more complex than primary keys. Primary keys make a column UNIQUE and NOT NULL. Foreign keys constrain based on columns in other tables. They are called foreign keys because the constraints
test=> CREATE TABLE primarytest2 (  
col1 INTEGER,  
col2 INTEGER,  
PRIMARY KEY(col1, col2)  
);  
NOTICE: CREATE TABLE/PRIMARY KEY will create implicit index 'primarytest2_pkey' for table 'primarytest2'

Figure 14.6: Example of a multi-column primary key

are foreign or outside the table.

For example, suppose a table contains customer addresses, and part of that address is the United States two-character state code. If a table existed with all valid state codes, a foreign key constraint could be created to prevent invalid state codes from being entered.

Figure 14.7 shows the creation of a primary key/foreign key relationship. Foreign key constraints are created by using REFERENCES to refer to the PRIMARY KEY of another table. Foreign keys link the tables together and prevent invalid data from being inserted or updated.

Figure 14.8 shows how foreign keys constrain column values. AL is a primary key value in statename, so the INSERT is accepted. XX is not a primary key value in statename, so the INSERT is rejected by the foreign key constraint.

Figure 14.9 shows the creation of the company tables from figure 6.3, page 50, using primary and foreign keys.

There are a variety of foreign key options listed below that make foreign keys even more powerful.
test=> INSERT INTO customer (state)
test-> VALUES ('AL');

INSERT 148732 1

ERROR: <unnamed> referential integrity violation -
key referenced from customer not found in statename

Figure 14.8: Foreign key constraints

create TABLE customer (customer_id INTEGER PRIMARY KEY,
name CHAR(30),
telephone CHAR(20),
street CHAR(40),
city CHAR(25),
state CHAR(2),
zipcode CHAR(10),
country CHAR(20));

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

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

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

Figure 14.9: Creation of company tables using primary and foreign keys
CHAPTER 14. CONSTRAINTS

Modification of Primary Key Row

If a foreign key constraint references a row as its primary key, and the primary key row is updated or deleted, the default foreign key action is to prevent the operation. Foreign key options ON UPDATE and ON DELETE allow a different action to be taken. Figure 14.10 shows the use of these options. The new customer table’s

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

Figure 14.10: Customer table with foreign key actions

ON UPDATE CASCADE specifies that if statename’s PRIMARY KEY is updated, customer.state should be updated with the new value too. The foreign key ON DELETE SET NULL option specifies that if someone tries to delete a statename row that is referenced by another table, the delete should set the foreign key to NULL.

The possible ON UPDATE and ON DELETE actions are:

- **NO ACTION** Updates and deletes to the PRIMARY KEY are prohibited if referenced by a foreign key row. This is the default.

- **CASCADE** Updates to the PRIMARY KEY cause updates to all foreign key columns that reference it. Deletes on the PRIMARY KEY cause deletes of all foreign key rows that reference it.

- **SET NULL** Updates and deletes to the PRIMARY KEY row cause the foreign key to be set to NULL.

- **SET DEFAULT** Updates and deletes to the PRIMARY KEY row cause the foreign key to be set to its DEFAULT.

Figure 14.11 illustrates the use of CASCADE and NO ACTION rules. The figure first shows the creation of primarytest which was used in figure 14.5. It then creates a foreigntest table with ON UPDATE CASCADE and ON DELETE NO ACTION. NO ACTION is the default, so ON DELETE NO ACTION was not required. The figure inserts a single row into each table, then shows an UPDATE on primarytest cascades to UPDATE foreigntest. The figure also shows that the primarytest row cannot be deleted unless the foreign key row is deleted first. Foreign key actions offer great flexibility in controlling how primary key changes affect foreign key rows.

Multi-Column Primary Keys

In order to specify a multi-column primary key, it was necessary to use PRIMARY KEY on a separate line in the CREATE TABLE statement. Multi-column foreign keys have the same requirement. Using primarytest2 from figure 14.6, figure 14.12 shows how to create a multi-column foreign key. FOREIGN KEY (col, …) must be used to label multi-column foreign key table constraints.
test=> CREATE TABLE primarytest (col INTEGER PRIMARY KEY);
NOTICE: CREATE TABLE/PRIMARY KEY will create implicit index 'primarytest_pkey' for table 'primarytest'
CREATE
test=> CREATE TABLE foreigntest (col2 INTEGER REFERENCES primarytest
ON UPDATE CASCADE
ON DELETE NO ACTION
);
NOTICE: CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
CREATE
test=> INSERT INTO primarytest values (1);
INSERT 148835 1
test=> INSERT INTO foreigntest values (1);
INSERT 148836 1
test=>
test=> -- CASCADE UPDATE is performed
test=>
test=> UPDATE primarytest SET col = 2;
UPDATE 1
test=> SELECT * FROM foreigntest;
col2
-----
 2
(1 row)

Figure 14.11: Foreign key actions
CHAPTER 14. CONSTRAINTS

```sql
CREATE TABLE primarytest2 (col1 INTEGER, col2 INTEGER, PRIMARY KEY(col1, col2));
```

**CREATE TABLE/PRIMARY KEY will create implicit index 'primarytest2_pkey' for table 'primarytest2'**

```sql
CREATE TABLE foreigntest2 (col3 INTEGER, col4 INTEGER, FOREIGN KEY (col3, col4) REFERENCES primarytest2);
```

**NOTICE: CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)**

Figure 14.12: Example of a multi-column foreign key

### Handling of NULL Values in the Foreign Key

A NULL value cannot reference a primary key. A single-column foreign key is either NULL or matches a primary key. In a multi-column foreign key, there are cases where only part of a foreign key can be NULL. The default behavior allows some columns in a multi-column foreign key to be NULL and some not NULL.

Using MATCH FULL in a multi-column foreign key constraint requires all columns in the key to be NULL or all columns to be not NULL. Figure 14.13 illustrates this. First, the tables from previous figure 14.12 are used to show that the default allows one column of a foreign key to be set to NULL. Table matchtest is created with the MATCH FULL foreign key constraint option. MATCH FULL allows all key columns to be set to NULL, but rejects the setting of only some multi-column key values to NULL.

### Frequency of Foreign Key Checking

By default, foreign key constraints are checked at the end of each INSERT, UPDATE, and DELETE query. This means if you perform a set of complex table modifications, foreign key constraints must remain valid at all times. For example, using the tables in figure 14.7, if there is a new state, and a new customer in the new state, the new state must be added to statename before the customer is added to customer.

In some cases, it is not possible to keep foreign key constraints valid between queries. For example, if two tables are foreign keys for each other, it may not be possible to INSERT into one table without having the other table row already present. A solution is to use the DEFERRABLE foreign key option and SET CONSTRAINTS so foreign key constraints are checked only at transaction commit. Using these, a multi-query transaction can make table modifications that violate foreign key constraints inside the transaction as long as the foreign key constraints are met at transactions commit. Figure 14.14 illustrates this. This is a contrived example because the proper way to perform this query is to INSERT into primarytest first, then INSERT into defertest. However, in complex situations, this reordering might not be possible, and DEFERRABLE and SET CONSTRAINTS should be used to defer foreign key constraints. A foreign key may also be configured as INITIALLY DEFERRED causing the constraint to be checked only at transaction commit by default.

Constraints can even be named. Constraint names appear in constraint violation messages, and can be used by SET CONSTRAINTS. See the CREATE TABLE and SET manual pages for more information.
test=> INSERT INTO primarytest2
  VALUES (1,2);
INSERT 148816 1

test=> INSERT INTO foreigntest2
  VALUES (1,2);
INSERT 148817 1

test=> UPDATE foreigntest2
  SET col4 = NULL;
UPDATE 1

test=> CREATE TABLE matchtest (  
  col3 INTEGER,
  col4 INTEGER,
  FOREIGN KEY (col3, col4) REFERENCES primarytest2
MATCH FULL );

NOTICE: CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
CREATE

test=> UPDATE matchtest
  SET col3 = NULL, col4 = NULL;
UPDATE 1

ERROR: <unnamed> referential integrity violation -
MATCH FULL doesn't allow mixing of NULL and NON-NULL key values

Figure 14.13: MATCH FULL foreign key
test=> CREATE TABLE defertest(
    col2 INTEGER REFERENCES primarytest DEFERRABLE);

NOTICE: CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
CREATE

BEGIN;

-- INSERT is attempted in non-DEFERRABLE mode

INSERT INTO defertest VALUES (5);
ERROR: <unnamed> referential integrity violation -
key referenced from defertest not found in primarytest

COMMIT;

BEGIN;

-- all foreign key constraints are set to DEFERRED

INSERT INTO defertest VALUES (5);
INSERT INTO primarytest VALUES (5);
COMMIT

Figure 14.14: DEFERRABLE foreign key constraint
14.5 CHECK

The CHECK constraint enforces column value restrictions. CHECK constraints can restrict a column to a set of values, only positive numbers, or reasonable dates. Figure 14.15 shows an example of CHECK constraints. This is a modified version of the friend table from figure 3.2, page 10. This figure has many CHECK clauses:

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

```sql
INSERT INTO friend2 VALUES ('Al', 'Rivers', 'Wibbleville', 'J', 35, 'S', '1951-09-23');
```

ERROR: ExecAppend: rejected due to CHECK constraint friend2_last_met

Figure 14.15: CHECK constraints

- **state** Forces the column to be two characters long. CHAR() pads the field with spaces, so state must be trim()-ed of trailing spaces before the length() is computed.

- **age** Forces the column to hold only positive values.

- **gender** Forces the column to hold either M or F.

- **last_met** Forces the column to be between January 1, 1950 and the current date.

- **table** Forces the table to only accept rows where firstname is not AL or lastname is not RIVERS. The effect of this rule is to prevent Al Rivers from being entered into the table. His name will be rejected if it is in uppercase, lowercase, or mixed case. This must be done as a table-level CHECK constraint. Comparing firstname to AL at the column level would have prevented all ALs from being entered, which was not desired. The desired restriction is a combination of firstname and lastname.

The figure then tries to INSERT a row that violates all CHECK constraints. Though the CHECK failed on the friend2_last_met constraint, if that were corrected, the other constraints would prevent the insertion. By default, CHECK allows NULL values.

14.6 Summary

This chapter covered a variety of constraints that help keep user data constrained within specified limits. With small databases, constraints are of marginal benefit. With databases holding millions of rows, constraints help keep database information organized and complete.
CHAPTER 14. CONSTRAINTS
Chapter 15

Importing and Exporting Data

COPY allows rapid loading and unloading of user tables. COPY can write the contents of a table to an ASCII file, and it can load a table from an ASCII file. These files can be used for backup or to transfer data between PostgreSQL and other applications.

The first section of this chapter shows how COPY can be used to unload and load database tables. The remainder of the chapter covers topics of interest to those using COPY to share data with other applications. The last section contains tips for using COPY.

15.1 Using COPY

COPY...TO allows the contents of a table to be copied out to a file. The file can later be read in using COPY...FROM.

Figure 15.1 illustrates this. It shows the creation of a table with columns of various types. Two rows are then inserted into copytest. SELECT shows the contents of the table, and COPY...TO writes the table to file /tmp/copytest.out. The rows are then deleted, and COPY...FROM reloads the table, as shown by the last SELECT.

COPY provides a quick way to load and unload tables. It is used for database backup, as covered in section 20.5. The following sections cover various COPY features that are important when reading or writing COPY files in other applications.

15.2 COPY File Format

COPY...TO can export data to be loaded into other applications, and COPY...FROM can import data from other applications. If you are constructing a file to be used by COPY, or you are reading a COPY file in another application, it is important to understand COPY’s file format.

Figure 15.2 shows the contents of the COPY file from figure 15.1. First, \q exits psq1 to an operating system prompt. Then, the UNIX cat\(^1\) command displays the file /tmp/copytest.out. The file contains one line for every row in the table. Columns in the file are separated by TABs. These TABs are called delimiters because they delimit or separate columns.

However, TABs are hard to see. They look like multiple spaces. The next command processes the file using sed\(^2\) to display TABs as <TAB>. This clearly shows the TABs in the file. Notice TABs are different from spaces.

The columns do not line up as they do in psq1. This is because the columns are of different lengths. The value of textcol in the first line is longer than value in the the second line. The lack of alignment is expected

---

\(^1\)Non-UNIX operating system users would use the type command.

\(^2\)Sed is an operating system command that replaces one string with another. See the sed(1) manual page for more information.
CHAPTER 15. IMPORTING AND EXPORTING DATA

test=> CREATE TABLE copytest (
  intcol INTEGER,
  numcol NUMERIC(16,2),
  textcol TEXT,
  boolcol BOOLEAN
);
CREATE

test=> INSERT INTO copytest
VALUES (1, 23.99, 'fresh spring water', 't');
INSERT 174656 1

test=> INSERT INTO copytest
VALUES (2, 55.23, 'bottled soda', 't');
INSERT 174657 1

test=> SELECT * FROM copytest;

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

test=> COPY copytest TO '/tmp/copytest.out';
COPY

test=> DELETE FROM copytest;
DELETE 2

test=> COPY copytest FROM '/tmp/copytest.out';
COPY

test=> SELECT * FROM copytest;

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

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

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

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

Figure 15.2: Example of COPY...FROM
because the COPY file is designed for easy processing, with one TAB between each column. It is not designed
for display purposes.

15.3  DELIMITERS

The default TAB column delimiter can be changed. COPY has a USING DELIMITERS option that sets the column
delimiter. Figure 15.3 shows that setting the delimiter to a pipe symbol (|) causes the output file to use pipes
to separate columns.

```
test=> COPY copytest TO '/tmp/copytest.out' USING DELIMITERS '|';
COPY
```

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

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

If a COPY file does not use the default TAB column delimiter, COPY...FROM must use the proper USING
DELIMITERS option. Figure 15.3 shows that if a file uses pipes rather than TABs as column delimiters,
COPY...FROM must specify pipes as delimiters. The first COPY...FROM fails because it cannot find a TAB to
separate the columns. The second COPY...FROM succeeds because the proper delimiter for the file was used.

```
test=> DELETE FROM copytest;
DELETE 2
```

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

```
test=> COPY copytest FROM '/tmp/copytest.out' USING DELIMITERS '|';
COPY
```

Figure 15.4: Example of COPY...FROM...USING DELIMITERS

15.4  COPY without files

COPY can be used without files. COPY can use the same input and output locations used by psql. The special
name stdin represents the psql input, and stdout represents the psql output. Figure 15.5 shows how stdin
can be used to supply COPY input directly from your keyboard. For clarity, text typed by the user is in bold.
The gaps in second line typed by the user were generated by pressing the TAB key. The user types \.. to exit
COPY...FROM. COPY to stdout displays the COPY output on your screen. This can be useful when using psql
in automated scripts.
15.5 Backslashes and NULLS

There is potential confusion if the character used as a column delimiter also exists in user data. If they appeared the same in the file, \COPY\ FROM would be unable to determine if the character was a delimiter or user data.

\COPY\ avoids any confusion by specially marking delimiters appearing in user data. It prececedes them with a backslash (\). If pipe is the delimiter, \COPY\ TO uses pipes (|) for delimiters, and backslash-pipes (\ |) for pipes in user data. Figure 15.6 shows an example of this. Each column is separated by a pipe, but the pipe that appears in user data is output as abc \ | def.

Backslash causes any character that follows it to be treated specially. Because of this, a backslash in user data must be output as two backslashes, \ \.

Another special backslash in this figure the use of \N to represent NULL. This prevents NULLs from being confused with user values.

The default NULL representation can be changed using WITH NULL AS. The command \COPY\ copytest TO '/tmp/copytest.out' WITH NULL AS '?' will output NULLs as a question marks. However, this will make a user column containing a single question mark indistinguishable from a NULL in the file. To output NULLs as blank columns, use the command \COPY\ copytest TO '/tmp/copytest.out' WITH NULL AS ' '. To treat empty columns as NULLS on input, use \COPY\ copytest FROM '/tmp/copytest.out' WITH NULL AS ' '.

Table 15.1 summarizes the delimiter, NULL, and backslash handling of \COPY\. The first two lines in the table show that preceeding a character with a backslash prevents the character from being interpreted as a delimiter. The next line shows that \N means NULL when using the default NULL representation.

The other backslash entries show simple representations for common characters. The last line shows double-backslash is required to represent a literal backslash.
15.6 COPY Tips

Full pathnames must be used with the COPY command because the database server is running in a different directory than the psql client. Files are read and written by the postgres user, so postgres must have permission to read the file for COPY...FROM, and directory write permission for COPY...TO. Because COPY uses the local file system, users connecting over a network cannot use filenames. They can use stdin and stdout, or psql’s \copy command.

By default, the system-generated OID column is not written out, and loaded rows are given new OID’s. COPY...WITH OIDS allows OID’s to be written and read.

COPY writes only entire tables. To COPY only part of a table, use SELECT...INTO TEMPORARY TABLE with an appropriate WHERE clause and then COPY the temporary table to a file.

See the COPY manual page for more detailed information.

15.7 Summary

COPY can be thought of as a crude INSERT and SELECT. It imports and exports data in a very generic format. This makes it ideal for use by other applications and for backup purposes.
CHAPTER 15. IMPORTING AND EXPORTING DATA
Chapter 16

Database Query Tools

This chapter covers two popular POSTGRESQL database query tools, psql and pgaccess.

16.1 PSQL

The following sections summarize the capabilities of psql. The psql manual has detailed information about each item. See chapter 2 for an introduction to psql.

Query Buffer Commands

Table 16.1 shows the commands used to control the psql query buffer. There is one item of particular interest, edit (\e). This allows editing of the query buffer. The \e command loads the contents of the query buffer into the default editor. When the user exits the editor, the editor contents are reloaded into the query buffer, ready for execution. The environment variable EDITOR specifies the default editor.

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

Table 16.1: psql query buffer commands

General Commands

A list of general psql commands is shown in table 16.2. Psql has a local copy interface that allows copy operations using files local to the computer running psql, rather than local to the computer running the database server. Later sections cover the use of \set, \unset, and \pset.
CHAPTER 16. DATABASE QUERY TOOLS

<table>
<thead>
<tr>
<th>Operation</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect to another database</td>
<td><code>\connect dbname</code></td>
</tr>
<tr>
<td>Copy tablefile to/from database</td>
<td>`\copy tablename &lt;</td>
</tr>
<tr>
<td>Set a variable</td>
<td><code>\set variable or \set variable value</code></td>
</tr>
<tr>
<td>Unset a variable</td>
<td><code>\unset variable</code></td>
</tr>
<tr>
<td>Set output format</td>
<td><code>\pset option or \pset option value</code></td>
</tr>
<tr>
<td>Echo</td>
<td><code>\echo string or \echo </code>command`</td>
</tr>
<tr>
<td>Echo to \o output</td>
<td><code>\qecho </code>command`</td>
</tr>
<tr>
<td>Copyright</td>
<td><code>\copyright</code></td>
</tr>
<tr>
<td>Change character encoding</td>
<td><code>\encoding newencoding</code></td>
</tr>
</tbody>
</table>

Table 16.2: psql general commands

Output Format Options

The `\pset` command controls the output format used by psql. Table 16.3 shows all the formatting commands and figure 16.1 shows examples of their use. In the figure, `\pset tuples_only` causes psql to show only data rows, suppressing table headings and row counts. `Tuples only` does not take a second argument. It is an on/off parameter. The first `\pset tuples_only` turns it on, and another one turns it off. The second `\pset` in the figure causes psql to display NULL as `(null).

Output Format Shortcuts

In addition to using `\pset`, some output format options have shortcuts as shown in table 16.4.

Variables

The `\set` command sets a variable, and `\unset` removes a variable. Variables are accessed by preceeding the variable name with a colon. The `\set` command used alone lists all defined variables.

Figure 16.2 shows the use of psql variables. The first variable assigned is `num_var`. It is accessed in the SELECT query by preceeding the variable name with a colon. The second `\set` command places the word SELECT into a variable, and uses that variable to perform a SELECT query. The next example uses `backslash-quotes` (`\`) to create a string that contains single-quotes. This variable can then be used in place of a quoted string in queries. `Date_var` shows that grave accents (``) allow a command to be run and the result
test=> SELECT NULL;
  ?column?
  ********
  (1 row)

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

  (null)

Figure 16.1: Example of \pset

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

Table 16.4: psql output format shortcuts
test=> \set num_var 4
test=> SELECT :num_var;
?column?
--------
  4
(1 row)

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

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

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

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

Figure 16.2: psql variables
placed into a variable. In this case, the output of the UNIX date command is captured and placed into the variable. The assignment to date var2 combines the use of backslash-quotes and grave accents to run the date command and surround it with single quotes. The final SELECT shows that date var2 holds a quoted date string that can be used in queries.

psql predefines a number of variables. They are listed in table 16.5. The variables in the first group contain useful information. The rest affect the behavior of psql. Some of the predefined variables do not take an argument. They are activated using \set, and deactivated using \unset.

### Listing Commands

You can find a great deal of information about the current database using psql’s listing commands, as shown in table 16.6. They show information about tables, indexes, functions, and other objects defined in the database.

Most listing commands take an optional name parameter. This parameter can be specified as a regular expression. For example, \dt sec displays all table names beginning with sec, and \dt .*x.* shows all table names containing an x. Regular expressions are covered in section 4.10.

When using listing commands, the descriptions of data types and functions are called comments. POSTGRESQL predefines many comments, and the COMMENT command allows users to define their own. The \dd command and others display these comments. See the COMMENT manual page for more information.

Many of the commands allow an optional plus sign, which shows additional information. For example, \dT lists all data types, while \dT+ includes the size of each type. \df+ shows addition information about functions. When using the other commands, a plus sign causes the comments for the object to be displayed.

### Large Object Commands

psql has a local large object interface that allows large object operations using files local to the computer running psql, rather than local to the computer running the database server. Table 16.4 shows the local large object commands supported by psql.
CHAPTER 16. DATABASE QUERY TOOLS

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

Table 16.6: psql listing commands

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

Table 16.7: psql large object commands

Psql command-line arguments and startup file

You can change the behavior of psql when starting the psql session. Psql is normally started from the command line with psql followed by the database name. However, psql accepts extra arguments between psql and the database name which modify psql's behavior. For example, psql -f file test will read commands from file, rather than from the keyboard. Table 16.8 summarizes psql's command-line options. Consult the psql manual page for more detailed information.

Another way to change the behavior of psql on startup is to create a file called .psqlrc in your home directory. Each time psql starts, it executes any backslash or SQL commands in that file.

16.2 PGACCESS

Pgaccess is a graphical database tool. It is used for accessing tables, queries, views, sequences, functions, reports, forms, scripts, users, and schemas. PGACCESS is written using the POSTGRESQL TCL/TK interface. The PGACCESS source code is in pgsql/src/bin/pgaccess.

Figure 16.3 shows the opening pgaccess window. The tabs on the left show the items that can be accessed. The menu at the top allows database actions, table import/export, and object creation, deletion, and renaming.

Figure 16.4 shows the table window. This window allows table rows to be viewed and modified.

Pgaccess has many help screens which cover its capabilities in more detail.
<table>
<thead>
<tr>
<th>Option</th>
<th>Capability</th>
<th>Argument</th>
<th>Additional argument</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Connection</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Database (optional)</td>
<td>-d</td>
<td></td>
<td>database</td>
</tr>
<tr>
<td>Hostname</td>
<td>-h</td>
<td></td>
<td>hostname</td>
</tr>
<tr>
<td>Port</td>
<td>-p</td>
<td></td>
<td>port</td>
</tr>
<tr>
<td>User</td>
<td>-U</td>
<td></td>
<td>user</td>
</tr>
<tr>
<td>Force password prompt</td>
<td>-W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Version</td>
<td>-V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Field alignment</td>
<td>-A</td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>Field separator</td>
<td>-F</td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>Record separator</td>
<td>-R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rows only</td>
<td>-t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extended output format</td>
<td>-x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Echo \d* queries</td>
<td>-E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quiet mode</td>
<td>-q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTML output</td>
<td>-H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTML table tags</td>
<td>-T</td>
<td></td>
<td>tags</td>
</tr>
<tr>
<td>Set \psql options</td>
<td>-P</td>
<td></td>
<td>option or option=value</td>
</tr>
<tr>
<td>List databases</td>
<td>-l</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disable readline</td>
<td>-n</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Controlling Output</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Echo all queries from scripts</td>
<td>-a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Echo queries</td>
<td>-e</td>
<td></td>
<td>query</td>
</tr>
<tr>
<td>Execute query</td>
<td>-c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Get queries from file</td>
<td>-f</td>
<td></td>
<td>file</td>
</tr>
<tr>
<td>Output to file</td>
<td>-o</td>
<td></td>
<td>file</td>
</tr>
<tr>
<td>Single-step mode</td>
<td>-s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single-line mode</td>
<td>-S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Suppress reading */.psqlrc</td>
<td>-X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set variable</td>
<td>-v</td>
<td></td>
<td>var or var=value</td>
</tr>
</tbody>
</table>

Table 16.8: psql command-line arguments

Figure 16.3: Pgaccess opening window
16.3  Summary

This chapter covered psql and pgaccess. These are the most popular POSTGRESQL query tools.
Chapter 17

Programming Interfaces

psql is ideal for interactively entering SQL commands, and for running automated scripts. However, psql is not ideal for writing applications. Fortunately, POSTGRESQL has interfaces for many programming languages. Programming languages have variables, functions, conditional evaluation, looping, and complex input/output routines. These are required for writing good applications.

Table 17.1 shows the programming interfaces supported by POSTGRESQL. These language interfaces allow applications to pass queries to POSTGRESQL and receive results. The compiled languages execute faster, but are harder to program than the interpreted ones.

This chapter will show the same application using each interface. The application is a very simple one that prompts the user for a United States state code, and outputs the state name that goes with the code. Figure 17.1 shows the sample application being run. For clarity, the text typed by the user is in bold. The

Enter a state code: AL

Alabama

Figure 17.1: Sample application being run

program displays a prompt, the user types AL, and the program displays Alabama. Though state codes are unique, the application is written to allow multiple query return values. The application uses the statename table, which is recreated in figure 17.2.

Additional information about POSTGRESQL interfaces is available in the Programmer's Manual mentioned in section A.3.
test=> CREATE TABLE statename (code CHAR(2) PRIMARY KEY,
          name CHAR(30))
CREATE

test=> INSERT INTO statename VALUES ('AL', 'Alabama');
INSERT 18934 1

test=> INSERT INTO statename VALUES ('AK', 'Alaska');
INSERT 18934 1

Figure 17.2: Statename table

17.1 C Language Interface (LIBPQ)

Libpq is the native C interface to PostgreSQL. Psql and most other interfaces use libpq internally for database access.

Figure 17.3 shows how libpq is used. The application code communicates with the user’s terminal and uses libpq for database access. Libpq sends queries to the database server and retrieves results.

Figure 17.4 shows the sample program using libpq to access PostgreSQL. These are the tasks performed by the sample program:

- Establish database connection
- Prompt for and read the state code
- Form an appropriate SQL query
- Pass the SQL query to libpq
- POSTGRESQL executes the query
- Retrieve the query results from libpq
/*
   * libpq sample program
   */

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

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

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

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

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

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

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

    PQclear(res); /* free result */
    PQfinish(conn); /* disconnect from the database */

    return 0;
}

Figure 17.4: Libpq sample program
• Display results to the user
• Terminate database connection

All interaction with the database is done using libpq functions. The libpq functions called by the sample program are:

- `PQconnectdb()` Connects to the database
- `PQexec()` Sends the query to the database
- `PQntuples()` Returns number of rows (tuples) in the result
- `PQgetvalue()` Returns a specific row and column of the result
- `PQclear()` Frees resources used by the result
- `PQfinish()` Closes database connection

These are the most common libpq functions. The Programmer’s Manual covers all libpq functions and shows additional examples.

### 17.2 Pgeasy(LIBPGEASY)

Libpgeasy is a simplified C interface. It hides some of the complexity of libpq. Figure 17.5 shows a libpgeasy version of the same application. No error checking is required because libpgeasy automatically terminates the program if an error occurs. This can be changed using `on_error_continue()`.

### 17.3 Embedded C (ECPG)

Rather than using function calls to perform SQL queries, ecpg allows SQL commands to be embedded in a C program. The `ecpg` preprocessor converts lines marked by `EXEC SQL` to native SQL calls. The resulting file is then compiled as a C program.

Figure 17.6 shows an ecpg version of the same application. Ecpg implements the ANSI embedded SQL C standard, which is supported by many database systems.

### 17.4 C++ (LIBPQ++)

Libpq++ is POSTGRESQL’s C++ interface. Figure 17.7 shows the same application using libpq++. Libpq++ allows database access using object methods rather than function calls.

### 17.5 Compiling Programs

The above interfaces are based on C or C++. Each interface requires certain `include` and `library` files to generate an executable version of the program.

Interface `include` files are usually installed in `/usr/local/pgsql/include`. The compiler flag `-I` is needed so the compiler searches that directory for include files, i.e. `-I/usr/local/pgsql/include`.

Interface `libraries` are usually installed in `/usr/local/pgsql/lib`. The compiler flag `-L` is needed so the compiler searches that directory for library files, i.e. `-L/usr/local/pgsql/lib`.
/*
 * libpgeasy sample program
 */

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

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

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

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

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

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

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

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

    return 0;
}

Figure 17.5: libpgeasy sample program
/*
 * ecpg sample program
 */

#include <stdio.h>

EXEC SQL INCLUDE sqlca; /* ecpg header file */
EXEC SQL WHENEVER SQLERROR sqlprint;

int main()
{

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

EXEC SQL CONNECT TO test; /* connect to the database */
printf("Enter a state code: "); /* prompt user for a state code */
scanf("%2s", state_code);

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

EXEC SQL PREPARE s_statename FROM :query_string;
EXEC SQL DECLARE c_statename CURSOR FOR s_statename;/* DECLARE a cursor */
EXEC SQL OPEN c_statename; /* send the query */
EXEC SQL WHENEVER NOT FOUND DO BREAK;

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

free(state_name); /* free result */
EXEC SQL CLOSE c_statename; /* CLOSE the cursor */
EXEC SQL COMMIT;
EXEC SQL DISCONNECT; /* disconnect from the database */

return 0;
}

Figure 17.6: Ecpg sample program
/*  
 * libpq++ sample program  
 */

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

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

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

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

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

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

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

    return 0;
}

Figure 17.7: Libpq++ sample program
The compiler flag `-l` is needed so the compiler links to a specific library file. To link to `libpq.a` or `libpq.so`, the flag `-lpq` is needed. The `-l` flag knows the file begins with `lib`, so `-llibpq` is not required, just `-lpq`.

The commands to compile `myapp` for various interfaces are listed below:

```plaintext
libpq  cc -I/usr/local/pgsql/include -o myapp myapp.c -L/usr/local/pgsql/lib -lpq
libpqeasy cc -I/usr/local/pgsql/include -o myapp myapp.c -L/usr/local/pgsql/lib -lpqeasy
cpg    ecpg myapp.pgc
       cc -I/usr/local/pgsql/include -o myapp myapp.c -L/usr/local/pgsql/lib -lecpg
libpq++ cc++ -I/usr/local/pgsql/include -o myapp myapp.cpp -L/usr/local/pgsql/lib -lpq++
```

Notice each interface has its own library. *Ecpg* requires the *ecpg* preprocessor to be run before compilation. *Libpq++* requires a different compiler to be used.

### 17.6 Assignment to Program Variables

*PostgreSQL* is a network-capable database. This means the database server and user application can be run on different computers. Because character strings have the same representation on all computers, they are used for communication between the user program and database server. Queries are submitted as character strings, and results are passed back as character strings. This allows reliable communication even if the two computers are quite different.

The sample programs perform SELECTs on a `CHAR(30)` column. Because query results are returned as character strings, returned values can be assigned directly to program variables. However, non-character string columns, like `INTEGER` and `FLOAT`, cannot be assigned directly to integer or floating-point variables. A conversion might be required.

For example, using `libpq` or `libpq++`, a SELECT on an `INTEGER` column does not return an integer from the database, but a character string that must be converted to an integer by the application. An `INTEGER` is returned as the string '983' rather than the integer value 983. To assign this to an integer variable, the C library function `atoi()` must be used, i.e. `var = atoi(colval)`.

One exception to this is `BINARY` cursors, which return binary representations of column values. Results from `BINARY` cursors can be assigned directly to program variables. However, because they return column values in binary format, the application and database server must be running on the same computer, or computers with the same CPU architecture. See the DECLARE manual page for more information on `BINARY` cursors.

*Libpqeasy* uses `fetch()` to return values directly into program variables. `Fetch()` should place results into character string variables, or use `BINARY` cursors if possible.

*Ecpg* automatically converts data returned by *PostgreSQL* to the proper format before assignment to program variables.

The interpreted languages covered later have *type*-less variables, so they do not have this problem.

### 17.7 ODBC

*ODBC* (Open Database Connectivity) is an interface used by some applications and application-building tools to access SQL databases. *ODBC* is a middle-ware layer that is not meant for programming directly, but for communicating with other applications.

The *ODBC* source code is located in `pgsql/src/interfaces/odbc`. It can be compiled on UNIX and non-UNIX operating systems.
17.8 JAVA (JDBC)

Figure 17.8 shows a JAVA version of the same application.

The JAVA interface source code is located in `pgsql/src/interfaces/jdbc`. Once the interface is compiled, the file `postgresql.jar` should be copied to the directory containing the other jar files. The full path name of `postgresql.jar` must then be added to the CLASSPATH environment variable.

JAVA programs are compiled using `javac` and run using `java`. JAVA is both a compiled and interpreted language. It is compiled for speed, but interpreted when executed so any computer can run the compiled program.

17.9 Scripting Languages

The previous interfaces used compiled languages. Compiled languages require user programs to be compiled into CPU instructions.

The remaining interfaces are scripting languages. Scripting languages execute slower than compiled languages, but have several advantages:

- No compile required
- More powerful commands
- Automatic creation of variables
- Variables can hold any type of data

17.10 PERL

Figure 17.9 shows the same application in PERL. PERL is good for writing scripts and small applications. It is popular for processing text files and generating dynamic web pages using CGI (Common Gateway Interface). A PERL/DBI interface is also available.

17.11 TCL/Tk (PGTCLSH/PGTKSH)

Figure 17.10 shows a TCL version of the same application. TCL’s specialty is accessing other toolkits and applications.

The TK graphical interface toolkit is one example. It is used by TCL when writing graphical applications. The TK toolkit has become so popular that other scripting languages use it as their graphical interface library.

17.12 PYTHON (PYGRESQL)

PYTHON is an object-oriented scripting language. It is considered to be a well-designed language, with code that is easy to read and maintain. Figure 17.11 shows the same application written in PYTHON. The PYTHON interface source code is located in `pgsql/src/interfaces/python`. 
import java.io.*;
import java.sql.*;

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

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

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

        ResultSet res = stmt.executeQuery("SELECT name " +
                "FROM statename " +
                "WHERE code = " + state_code + ");

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

    public static void main(String args[]) {
        try {
            sample test = new sample();
        } catch(Exception exc) {
            System.err.println("Exception caught.\n" + exc);
            exc.printStackTrace();
        }
    }
}
#!/usr/local/bin/perl
#
# perl sample program
#
use Pg; # load database routines

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

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

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

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

Figure 17.9: PERL sample program

#!/usr/local/pgsql/bin/pgtclsh
#
# pgtclsh sample program
#
set conn [pg_connect test] ;# connect to the database
puts -nonewline "Enter a state code: " ;# prompt user for a state code
flush stdout
gets stdin state_code ;# send the query
set res [pg_exec $conn "SELECT name \n FROM statename \n WHERE code = '$state_code'"]
set ntups [pg_result $res -numTuples]

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

pg_disconnect $conn ;# disconnect from the database

Figure 17.10: TCL sample program
CHAPTER 17. PROGRAMMING INTERFACES

#! /usr/local/bin/python
#
# python sample program
#
import sys

from pg import DB # load database routines
conn = DB('test') # connect to the database

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

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

Figure 17.11: PYTHON sample program

17.13 PHP

PHP allows web browser access to POSTGRESQL. Using PHP, database commands can be embedded in web pages.

Two web pages are required for the sample application — one for data entry and another for display. Figure 17.12 shows a web page that allows entry of a state code. Figure 17.13 shows a second web page that

<!-- prompt user for a state code -->

Client Number:

perform a SELECT and displays the results. Normal web page commands (HTML tags) begin with < and end with >. PHP code begins with <? and ends with ?>.

The PHP interface is not shipped with POSTGRESQL. It can be downloaded from http://www.php.net.
<HTML>
<BODY>
<?
$database = pg_Connect("", ",", ",", "test"); # connect to the database
if (!$database) # did the database connection fail?

    echo "Connection to database failed."
    exit;

$result = pg_Exec($database,
"SELECT name 
FROM statename 
WHERE code = 'state_code'");

for ($i = 0; $i < pg_NumRows($result); $i++) # loop through all rows returned

    echo pg_Result($result,$i,0); # print the value returned
    echo "<BR>

?>
</BODY>
</HTML>

Figure 17.13: PHP sample program – Output
17.14 Installing Scripting Languages

The interpreted languages above require a database interface to be installed into the language. This is done by either recompiling the language, or dynamically loading the interface into the language. The following gives details about each interface:

**PERL** Use loads the POSTGRESQL interface into the PERL interpreter.

**TCL/TK** TCL/TK offers three interface options:

- Pre-built TCL interpreter called `pgtcsh`
- Pre-built TCL/TK interpreter called `pgtksh`, like TCL/TK’s `wish`
- Loadable library called `libpgtcl`

**PYTHON** Import loads the POSTGRESQL interface into the PYTHON interpreter.

**PHP** PHP must be recompiled to access POSTGRESQL.

17.15 Summary

All interface source code is located in `pgsql/src/interfaces`. Each interface includes sample source code for use in writing your own programs.

These interfaces allow the creation of professional database applications. Each interface has advantages. Some are easier, some faster, some more popular, and some work better in certain environments. The choice of an interface is often difficult. Hopefully this chapter will make that choice easier.
Chapter 18

Functions and Triggers

The previous chapter focused on client-side programming — programs that run on the user’s computer and interact with the POSTGRESQL database. Server-side functions, sometimes called stored procedures, run inside the database server rather than in the client application.

There are some good uses for server-side functions. For example, if a function is used by many applications, it can be embedded into the database server. Each application, then, no longer needs a copy of the function. Whenever it is needed, it can be called by the client. Unlike client-side functions, server-side functions can be called from inside SQL queries. Also, functions centrally installed in the server are easily modified. When a function is changed, client applications immediately start using the new version.

Figure 9.3 on page 94 lists many pre-installed server-side functions, like upper() and date_part(). This chapter shows how to create your own. This chapter also covers special server-side functions called triggers which are called automatically when a table is modified.

18.1 Functions

Server-side functions can be written in several languages:

- SQL
- PL/PGSQL
- PL/TCL
- PL/PERL
- C

SQL and PL/PGSQL functions will be covered in this chapter. C functions are more complex and will be covered in chapter 19.

18.2 SQL Functions

SQL functions allow queries to be named and stored in the database for later access. This section shows a variety of SQL functions of increasing complexity.

Functions are created using the CREATE FUNCTION command and removed with DROP FUNCTION. CREATE FUNCTION requires the following information:
- Function name
- Number of function arguments
- Data type of each argument
- Function return type
- Function action
- Language used by function action

Figure 18.1 shows the creation of a simple SQL function to convert from Fahrenheit to centigrade. It supplies the following information to CREATE FUNCTION:

- Function name is `ftoc`
- Function takes one argument of type `float`
- Function returns a `float`
- Function action is `SELECT ($1 - 32.0) * 5.0 / 9.0;`
- Function language is SQL

Most functions only return one value. SQL functions can return multiple values using SETOR. Function actions can contain INSERTS, UPDATES, and DELETES too. Function actions can also contain multiple queries separated by semicolons.

The function action in `ftoc()` uses SELECT to perform a computation. It does not access any tables. The $1 in the SELECT is automatically replaced by the first argument of the function call. If there were a second argument, it would be represented as $2.

Constants in the function contain decimal points so floating-point computations are performed. Without them, division would be performed using integers. For example, the query `SELECT 1/4` returns 0, while `SELECT 1.0/4.0` returns 0.25.

When the query `SELECT ftoc(68)` is executed, it calls `ftoc()`. `Ftoc()` replaces $1 with 68, and the computation in `ftoc()` is executed. In a sense, this is a SELECT inside a SELECT. The outer SELECT calls `ftoc()`, and `ftoc()` uses its own SELECT to perform the computation.

Figure 18.2 shows an SQL server-side function to compute tax. The casts to NUMERIC(8,2) are required...
18.2. SQL FUNCTIONS

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

```sql
CREATE
SELECT tax(100);
```

```sql
-----
6.00
(1 row)
```

Figure 18.2: SQL tax function

because the result of the computation must be rounded to two decimal places. This function uses the more
compact double-colon form of type-casting, rather than CAST. See section 9.3 for more information about
type casting. `SELECT tax(100)` performs a simple computation, similar to `ftoc()`.

One powerful use of server-side functions is their use in SQL queries. Figure 18.3 shows the use of
tax() with the part table from figure 6.3. In this figure, three rows are inserted into the table, then a SELECT

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

```sql
CREATE
INSERT INTO part VALUES (637, 'cable', 14.29, 5);
INSERT 20867 1
INSERT INTO part VALUES (638, 'sticker', 0.84, 1);
INSERT 20868 1
INSERT INTO part VALUES (639, 'bulb', 3.68, 3);
INSERT 20869 1
```

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

```sql
part_id | name | cost | tax | total
--------+-------------------+-------+-------+-------
637     | cable            | 14.29 | 0.86 | 15.15
638     | sticker          | 0.84  | 0.05  | 0.89  
639     | bulb             | 3.68  | 0.22  | 3.90  
```

(3 rows)

Figure 18.3: Recreation of the part table
displays columns from the part table with additional computed columns showing tax and cost plus tax.

Figure 18.4 shows a more complex function that computes shipping charges. The function uses `CASE` to compute shipping charges based on weight. The figure calls `shipping()` to generate a detailed analysis of the tax and shipping charges associated with each part. It prints the part number, name, cost, tax, subtotal of cost plus tax, shipping charge, and total of cost, tax, and shipping charge. The `SELECT` uses `trim()` to remove trailing spaces and narrow the displayed result.

If tax rate or shipping charges change, it is easy to change the function to reflect the new rates. Simply use `DROP FUNCTION` to remove the function, and recreate it with new values. All user applications will automatically start using the new version because the computations are embedded in the database, not in user applications.

Server-side functions can also access database tables. Figure 18.5 shows an SQL function that internally accesses the `statename` table. It looks up the proper state name for the state code supplied to the function.

Figure 18.6 shows two queries which yield identical results. The first query joins the `customer` and `statename` tables. The second query does a `SELECT` on `customer` and for each row, `getstatename()` is called to find the customer’s state name. These two queries yield the same result only if each customer row joins to exactly one `statename` row. If there were `customer` rows that did not join to any `statename` row, or joined to many `statename` rows, the results would be different. Also, because the second query executes the SQL function for every row in `customer`, it is slower.
18.2. SQL FUNCTIONS

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

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

Figure 18.5: SQL function getstatename
```

```
test=> SELECT customer.name, statename.name 
test-> FROM customer, statename 
test-> WHERE customer.state = statename.code 
test-> ORDER BY customer.name; 

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

test=> SELECT customer.name, getstatename(customer.state) 
test=> FROM customer 
test=> ORDER BY customer.name; 

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

Figure 18.6: Getting state name using join and function
18.3 PL/PGSQL Functions

PL/PGSQL is another language for server-side functions. It is a true programming language. While SQL functions only allow argument substitution, PL/PGSQL has features like variables, conditional evaluation, and looping.

PL/PGSQL is not installed in each database by default. To use it in database test, it must be installed by running `createlang plpgsql test` from the operating system prompt.

Figure 18.7 shows a PL/PGSQL version of the SQL function `getstatename` from figure 18.5. The only differences are the addition of `BEGIN...END` and the language definition as `PL/PGSQL`.

```sql
CREATE FUNCTION getstatename2(text)
    RETURNS text
    AS 'BEGIN
    SELECT CAST(name AS TEXT)
    FROM statename
    WHERE code = $1;
    END;' LANGUAGE 'plpgsql';
```

Figure 18.7: PL/PGSQL version of `getstatename`

Figure 18.8 shows a more complicated PL/PGSQL function. It accepts a `text` argument, and returns the argument in uppercase, with a space between each character. This is used in the next `SELECT` to display a report heading. This function illustrates the use of variables and `WHILE` loops in PL/PGSQL.

Figure 18.9 shows a much more complicated PL/PGSQL function. This function takes a state name as a parameter and finds the proper state code. Because state names are longer than state codes, they are often misspelled. This function deals with misspellings by performing lookups in several ways. First, it attempts to find an exact match. If that fails, it searches for a unique state name that matches the first 2, 4, or 6 characters, up to the length of the supplied string. If a unique state is not found, an empty string (`''`) is returned. Figure 18.10 shows several `getstatecode()` function calls.

Getstatecode() illustrates several unique PL/PGSQL features:

- **%TYPE** Data type that matches a database column.
- **RECORD** Data type that stores the result of a `SELECT`.
- **SELECT INTO** A special form of `SELECT` that allows query results to be placed into variables. It should not be confused with `SELECT * INTO`.
- **FOUND** Predefined BOOLEAN variable that represents the status of the previous `SELECT INTO`.
- **RETURN** Exits and returns a value from the function.

Many other PL/PGSQL features are covered in the User’s Manual mentioned in section A.3.

Figure 18.11 shows a PL/PGSQL function that provides a server-side interface for maintaining the `statename` table. Function `change_statecode` performs `INSERT`, `UPDATE`, and `DELETE` operations on the `statename` table. `Change_statecode()` is called with a state code and state name. If the state code is not in the table, it is inserted. If it already exists, the state name is updated. If the function is called with an empty state name (''), the state is deleted from the table. The function returns true ('t') if `statename` was changed, and false ('f') if the `statename` table was unmodified. Figure 18.12 shows examples of its use.
18.3. PL/PGSQL FUNCTIONS

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

    Figure 18.8: PL/PGSQL spread function
test=> CREATE FUNCTION getstatecode(text)
  test=> RETURNS text
  test=> AS 'DECLARE
      state_str statename.name%TYPE;
      statename_rec record;
      i integer;
      len integer;
      matches record;
      search_str text;
      BEGIN
        state_str := initcap($1); -- capitalization match column
        len := length(trim($1));
        i := 2;
        SELECT INTO statename_rec * -- first try for an exact match
          FROM statename
          WHERE name = state_str;
        IF FOUND
          THEN RETURN statename_rec.code;
        END IF;
        WHILE i <= len LOOP -- test 2,4,6,... chars for match
          search_str = trim(substr(state_str, 1, i)) || ''%'';
          SELECT INTO matches COUNT(*)
            FROM statename
            WHERE name LIKE search_str;
          IF matches.count = 0 -- no matches, failure
            THEN RETURN NULL;
          END IF;
          IF matches.count = 1 -- exactly one match, return it
            THEN
              SELECT INTO statename_rec *
                FROM statename
                WHERE name LIKE search_str;
              IF FOUND
                THEN RETURN statename_rec.code;
              END IF;
            END IF;
          i := i + 2; -- >1 match, try 2 more chars
        END LOOP;
        RETURN '''' ;
      END;' LANGUAGE 'plpgsql';

Figure 18.9: PL/PGSQL getstatecode function
test=> SELECT getstatecode('Alabama');
getstatecode
------------
   AL
(1 row)

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

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

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

(1 row)

Figure 18.10: Calls to getstatecode function
CREATE FUNCTION change_statename(char(2), char(30)) RETURNS boolean AS 'DECLARE
    state_code ALIAS FOR $1;
    state_name ALIAS FOR $2;
    statename_rec RECORD;
BEGIN
    IF length(state_code) = 0 -- no state code, failure
    THEN RETURN 'f';
    ELSE
        IF length(state_name) != 0 -- is INSERT or UPDATE?
        THEN
            SELECT INTO statename_rec *
            FROM statename
            WHERE code = state_code;
            IF NOT FOUND -- is state not in table?
                INSERT INTO statename
                VALUES (state_code, state_name);
            ELSE
                UPDATE statename
                SET name = state_name
                WHERE code = state_code;
            END IF;
            RETURN 't';
        ELSE
            SELECT INTO statename_rec *
            FROM statename
            WHERE code = state_code;
            IF FOUND
                DELETE FROM statename
                WHERE code = state_code;
            ELSE
                RETURN 'f';
            END IF;
        END IF;
    END IF;
END;' LANGUAGE 'plpgsql';
18.3. PL/PGSQL FUNCTIONS

```sql
test=> DELETE FROM statename;
DELETE 1

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

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

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

Figure 18.12: Example of change_statename()
18.4 Triggers

Rules allow SQL queries to be executed when a table is accessed. They are covered in section 13.6. Triggers offer another way to perform actions on INSERT, UPDATE, or DELETE. Triggers are ideal for checking or modifying a column value before it is added to the database.

Triggers and rules are implemented differently. Triggers call server-side functions for each modified row while rules rewrite user queries or add additional queries. Triggers are ideal for checking or modifying a row before it is added to the database. Rules are ideal when the action affects other tables.

Triggers allow special server-side functions to be called every time a row is modified. These special functions can be written in any server-side language except SQL. These functions control the action taken by the query. They can reject certain values, or modify them before they are added to the database. Triggers that return NULL cause the operation that caused the trigger to be ignored.

Server-side trigger functions are special because they have predefined variables to access the row that caused the trigger. For INSERT triggers, the variable new represents the row being inserted. For DELETE, the variable old represents the row being deleted. For UPDATE, triggers can access the pre-UPDATE row using old and the post-UPDATE row using new. These are the same as the old and new variables in rules.

Figure 18.13 shows the creation of a special server-side trigger function called trigger_insert_update_statename. This function uses the new RECORD variable to:

- Reject a state code that is not exactly two alphabetic characters
- Reject a state name that contains non-alphabetic characters
- Reject a state name less than three characters in length
- Uppercase the state code
- Capitalize the state name

When invalid data is entered, RAISE EXCEPTION aborts the current query and displays an appropriate error message. Validity checks can also be performed using CHECK constraints covered in section 14.5.

Uppercase and capitalization occur by simply assigning values to the new variable. The function return type is opaque because new is returned by the function.

CREATE TRIGGER causes trigger_insert_update_statename() to be called every time a row is inserted or updated in statename. The remaining queries in the figure show three rejected INSERTs, and a successful INSERT that is properly upperscased and capitalized by the function.

Trigger functions can be quite complicated. They can perform loops, SQL queries, and any operation supported in server-side functions. See the CREATE TRIGGER and DROP TRIGGER manual pages for additional information.

18.5 Summary

Server-side functions allow programs to be embedded into the database. These programs can be accessed from client applications, and used in database queries. Moving code into the server allows for increased efficiency, maintainability, and consistency. Triggers are special server-side functions called when a table is modified.
test=> CREATE FUNCTION trigger_insert_update_statename()
  test-> RETURNS opaque
  test-> AS 'BEGIN
  test'> IF new.code !~ '^([A-Za-z][A-Za-z])$' THEN RAISE EXCEPTION ''Code must be two alphabetic characters.''';
  test'> END IF;
  test'> IF new.name !~ '^([A-Za-z ]+)$' THEN RAISE EXCEPTION ''Name must be only alphabetic characters.''';
  test'> END IF;
  test'> IF length(trim(new.name)) < 3 THEN RAISE EXCEPTION ''Name must be longer than two characters.''';
  test'> END IF;
  test'> new.code = upper(new.code); -- uppercase statename.code
  test'> new.name = initcap(new.name); -- capitalize statename.name
  test'> RETURN new;
  test'> END;
  test-> LANGUAGE 'plpgsql';
CREATE

Figure 18.13: Trigger creation
CHAPTER 18. FUNCTIONS AND TRIGGERS
Chapter 19

Extending POSTGRESQL Using C

While POSTGRESQL has a large number of functions, operators, data types, and aggregates, there are cases when users need to create their own. The previous chapter already showed how to create functions in languages other than C. This chapter covers C functions and the creation of custom operators, data types, and aggregates that behave just like the ones already in POSTGRESQL.

Extending POSTGRESQL in this way involves several steps:

• Write C code to implement the new functionality

• Compile the C code into an object file that contains CPU instructions

• Issue CREATE FUNCTION commands to register the new functions

• Issue the proper commands if creating operators, data types, or aggregates:
  – CREATE OPERATOR
  – CREATE TYPE
  – CREATE AGGREGATE

The full details of extending POSTGRESQL are beyond the scope of this book. This chapter is just an overview. The Programmer’s Manual mentioned in section A.3 has more detailed information.

19.1 Writing C code

The best way to add a new function, operator, data type, or aggregate is to start with a copy of a file from the POSTGRESQL source directory pgsql/src/backend/utils/adt. Start with a file that has functions similar to the ones you need. Make sure your new function names are unique.

For example, the previous chapter had a floc() SQL function that converted Fahrenheit to centigrade. Figure 19.1 shows a C function that converts centigrade to Fahrenheit.

While writing C functions, you may find it necessary to execute SQL queries from inside the function. The Server Programming Interface (SPI) allows C functions to execute SQL queries and process results from within C functions.
19.2 Compile the C code

The next step is to compile your C file into an object file that contains CPU instructions.

In fact, a special object file must be created that can be dynamically linked into the POSTGRESQL server. Many operating systems require special flags to create an object file that can be dynamically linked. The best way to find the required flags is to go to `pgsql/src/test/regress` and type `make clean` and then `make regress.so`. This will display the compile commands used to generate the dynamically linkable object file `regress.so`. The `-I` compile flags allow searching for include files. Some of the other flags are used for generating dynamic object files. Use those flags to compile your C code into a dynamically linkable object file. You may need to consult your operating system documentation for assistance in locating the proper flags.

19.3 Register the New Functions

Now that a dynamically linkable object file has been created, its functions must be registered with POSTGRESQL. The `CREATE FUNCTION` command registers a new function by storing information in the database. Figure 19.2 shows the `CREATE FUNCTION` command for `ctof`. `Ctof` takes a `float` argument and returns a `float`.

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

The SQL data type `float` is the same as the C type `double` used in `ctof()` above. The dynamically linkable object files is `/users/pgman/sample/ctof.so` and it is written in the C language.

A single object file can contain many functions. You must use `CREATE FUNCTION` to register each function you want to access from POSTGRESQL. `CREATE FUNCTION` also allows non-object files to be used as functions. This is covered in chapter 18.

With the functions registered, they can be called just like POSTGRESQL internal functions. Figure 19.3 shows the `ctof()` function used in a `SELECT` statement. See `CREATE_FUNCTION` for more information.

---

1 Some operating systems may need to use `gmake` rather than `make`. Also, some operating systems will use `regress.o` rather than `regress.so`.

19.4 Optionally Create Operators, Types, and Aggregates

Operators, types, and aggregates are built using functions. `CREATE OPERATOR`, `CREATE TYPE`, and `CREATE AGGREGATE` register that a set of functions should behave as an operator, type, or aggregate. They name the new operator, type, or aggregate, and call the supplied functions whenever that name is accessed. See `CREATE_OPERATOR`, `CREATE_TYPE`, and `CREATE_AGGREGATE` for more information.

19.5 Summary

Extending POSTGRESQL is a complicated process. This chapter has covered only the basic concepts. As mentioned earlier, the *Programmer’s Manual* mentioned in section A.3 has more detailed information.
CHAPTER 19. EXTENDING POSTGRESQL USING C
Chapter 20

Administration

This chapter covers a variety of administrative tasks. The chapter assumes PostgreSQL is installed and running. If it is not, see appendix B.

20.1 Files

When PostgreSQL is installed, it creates files in its home directory, typically /usr/local/pgsql. This directory contains all the files needed by PostgreSQL. It contains various subdirectories:

/bin This contains PostgreSQL command-line programs, like psql.
/data This contains configuration files and tables shared by all databases. For example, pg_shadow is a table shared by all databases.
/data/base This contains a subdirectory for each database. Using the du and ls commands, administrators can display the amount of disk space used by each database, table, or index.
/doc This contains PostgreSQL documentation and manual pages.
/include This contains include files used by various programming languages.
/lib This contains libraries used by various programming languages. It also contains files used during initialization and sample configuration files that can be copied to /data and modified.

20.2 Creating Users

New users are created by running createuser from an operating system prompt. Initially, only the PostgreSQL super-user, typically postgres, can create new users. Other users can be given permission to create new users and databases.

PostgreSQL usernames do not have to exist as operating system users. For installations using database password authentication, a createuser flag is available so passwords can be assigned.

Users are removed with dropuser. CREATE USER, ALTER USER, and DROP USER commands are available in SQL.

PostgreSQL also allows the creation of groups using CREATE GROUP in SQL. GRANT permissions can be specified using these groups.

Figure 20.1 shows examples of user administration commands. It creates one user from the command line, a second user in psql, and alters a user. It then creates a group, and gives table permissions to the
$ createuser demouser1
Shall the new user be allowed to create databases? (y/n) n
Shall the new user be allowed to create more new users? (y/n) n
CREATE USER
$ psql test
Welcome to psql, the PostgreSQL interactive terminal.

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

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

Figure 20.1: Examples of user administration
group. Finally it reconnections to the database as a different user. This was possible because the site has local
users configured with trust access. This is covered in section 20.4.

These commands can only be performed by a user with create user privileges. More information about
each command can be found in the manual pages.

### 20.3 Creating Databases

New databases are created by running `createdb` from an operating system prompt. Initially, only the POST-
GRESQL super-user can create new databases. Other users can be given permission to create new databases.

`createdb` creates a new database by making a copy of the `template1` database. `Template1` is made when
POSTGRESQL is first initialized. Any modifications to `template1` will appear in newly created databases.

Databases are removed with `dropdb`. `CREATE DATABASE` and `DROP DATABASE` commands are also available
in SQL.

Figure 20.2 shows a database created from the command line and another one created in `psql`. A database

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

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

```

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

Figure 20.2: Examples of database creation and removal

is then destroyed, and a connection made to a new database. Additional information about each command
can be found in the manual pages.

### 20.4 Access Configuration

POSTGRESQL allows administrators to control database access. Access can be granted by database, user,
or TCP/IP network address. By default, POSTGRESQL allows database access only to users logged into the
computer running the database server. To enable network access, the `postmaster` must be started with the
-i flag.

Database access is controlled by the `data/pg_hba.conf` file, which is located in the POSTGRESQL home
directory. It contains several types of configuration entries:
local

Local entries control access by users logged into the same computer as the database server. Local connections use unix domain sockets. These are the per-database authentication options:

- **trust** — Trust users connecting to this database.
- **password** — Require a password of users connecting to this database.
- **crypt** — Like password, except send the password in an encrypted manner. This method is more secure than password.
- **reject** — Reject all connection requests for this database.

host and hostssl

Host and hostssl entries control TCP/IP network access. They include host and netmask fields. They support all the local options, plus:

- **ident** — Use a remote ident server for authentication.
- **krb4** — Use Kerberos IV authentication.
- **krb5** — Use Kerberos V authentication.

These entries are only effective if the postmaster is using the -i option. Hostssl controls access via the Secure Socket Layer (SSL) if enabled in the server.

User Mappings

By default, passwords used by password and crypt are contained in the pg_shadow table. This table is managed by createuser and ALTER USER.

However, password takes an optional argument that specifies a secondary password file which overrides pg_shadow. This file contains usernames and passwords of people allowed to connect. Using this method, a set of users can be given access to certain databases. See the pg_passwd manual page for more information on creating secondary password files. Currently, crypt does not support secondary password files.

The ident entry also takes an optional argument that specifies a special map name to map ident usernames to database usernames. The file data/pg_ident.conf is used to record these mappings.

Examples

Local entries are configured per database. A database entry of all applies to all databases. In data/pg_hba.conf, the lines:

```
local    all    trust
host     all    127.0.0.1  255.255.255.255  trust
```

cause all local users to be trusted. The first line affects users connecting via unix domain sockets, while the second line controls local users connecting to the same machine by TCP/IP. The local machine is accessed as TCP/IP address 127.0.0.1 (localhost).

Host and hostssl entries require the additional specification of host addresses and network masks. The lines:

```
host     all    127.0.0.1  255.255.255.255  trust
hostssl  all    127.0.0.1  255.255.255.255  trust
```
Database backups allow databases to be restored if a disk drive fails, a table is accidentally dropped, or a database file is accidentally deleted. If the databases are idle, a standard file system backup is sufficient as a PostgreSQL backup. If the databases are active, the pg_dumpall utility must be used for reliable backup. pg_dumpall outputs a consistent snapshot of all databases into a file that can be included in a file system backup. In fact, once a pg_dumpall file has been created, there is no need to backup the /data/base database files. There are a few configuration files in /data, like data/pg_hba.conf, which should be included in a file system backup because they are not in the pg_dumpall file. pg_dump can dump a single PostgreSQL database.

To restore using a pg_dumpall file, PostgreSQL must be initialized, any manually edited configuration files restored to /data, and the database dump file run by psql. This will recreate and reload all databases.

Individual databases can be reloaded from pg_dump files by creating a new database and loading it using psql. For example, figure 20.3 creates an exact copy of the test database. It dumps the contents of the database into a file. A new database called newtest is created, then the dump file is loaded into the new database.

Dump files contain ordinary SQL queries and COPY commands. Because the files contain database information, they should be created so only authorized users have permission to read them. See pg_dump and pg_dumpall manual pages for more information about these commands.
20.6 Server Startup and Shutdown

The PostgreSQL server uses two distinct programs — postmaster and postgres. Postmaster accepts all requests for database access. It does authentication and starts a postgres process to handle the connection. The postgres process executes user queries and returns results. Figure 20.4 illustrates this relationship.

![Figure 20.4: Postmaster and postgres processes](image)

POSTGRESQL sites normally have only one postmaster process, but many postgres processes. There is one postgres process for every open database session.

Administrators only need to start the postmaster, and the postmaster will start postgres backends as connection requests arrive. The postmaster can be started from the command line, or from a script.

Another way to start the postmaster is using pg_ctl. The pg_ctl utility allows easy starting and stopping of the postmaster. See the pg_ctl manual page for more information. The operating system startup scripts can even be modified to start the postmaster automatically.

The postmaster can be stopped by sending the process a signal using kill, or by using pg_ctl.

20.7 Monitoring

Postmaster and postgres produce useful information for administrators. They have many flags to control the information they output. They can show user connection information, SQL queries, and detailed performance statistics.

When the postmaster is started, its output should be sent to a file in the PostgreSQL home directory. That file can then be used to monitor database activity. See the postmaster and postgres manual pages for a complete list of output options. To specify flags to be passed to each postgres process, use the postmaster -o flag.

Another way to monitor the database is by using ps. The ps operating system command displays information about system processes, including information about the postmaster and postgres processes. It is a good tool for analyzing PostgreSQL activity, particularly for diagnosing problems. The ps command can display information about a process's:

- Current CPU usage
- Total CPU usage
- Start time
- Memory usage
20.8. PERFORMANCE

• Disk operations (on some operating systems)

Each operating system uses different `ps` flags to output these values. A typical display is:

```
USER      PID  %CPU  TIME   STARTED  VSZ  INBLK  OUBLK COMMAND
---       --   ---   ----   --------  --   -----  ------  ---------
postgres  18923  45.4  0:27.79 1:15PM   2140  34    1 /usr/local/postgres/ ...
```

In this case, process 18923 is using 45.4% of the CPU, has used 27.79 seconds of CPU time, was started at 1:15PM, has read 34 blocks, and has written 1 block.

To identify who is using each `postgres` process, most operating systems allow `ps` to display connection information:

• Username
• User’s network address
• Database
• SQL command keyword (SELECT, INSERT, UPDATE, DELETE, CREATE, idle, …)

`Ps` displays this information next to the name of each `postgres` process. A typical display is:

```
PID TT STAT   TIME   COMMAND
--- -- ---   ----   --------
---    
18923 ?? S 0:27.79 /usr/local/postgres/bin/postgres demouser localhost test SELECT
```

In this example, `demouser`, using process id 18923, is connecting from the local machine to database `test`, and is executing a `SELECT`. Administrators can use `ps` to analyze who is connected to each database, the query command they are running, and the system resources used.

20.8 Performance

Chapter 11 covers the performance of SQL queries. This chapter covers more general performance considerations.

One of the most important administrative tasks is the scheduling of the `vacuumdb -a` command. This vacuums all databases. It should be run when the databases are least busy. Section 11.4 describes the purpose of vacuuming. Vacuum analyze should also be performed periodically. This is covered in section 11.5. `Vacuumdb` can perform analyzing as well. See the `vacuumdb` manual page for more information.

Postmaster and `postgres` have several flags that can improve performance. The `postmaster -B` flag controls the amount of shared buffer memory allocated. The `postgres -S` flag controls the amount sort memory allocated. While these consume system resources, they also improve performance by reducing disk access.

Database performance can also be improved by moving databases to different disk drives. This allows disk access to be spread among multiple drives. The `initlocation` utility allows new database locations to be created on different drives. `Createdb` can use these locations for new databases.

`POSTGRESQL` stores tables and indexes in operating system files. Using operating system symbolic links, databases, tables, and indexes can be moved to different disk drives, often improving performance.
20.9 System Tables

There is a great deal of information stored in PostgreSQL system tables. These tables begin with `pg_`. They contain information about data types, functions, operators, databases, users, and groups. Table 20.1 shows the most commonly used tables.

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

Table 20.1: Commonly used system tables

`Pg_log` is an binary file and not a real table. `Pg_shadow` contains user passwords and is not visible to ordinary users. `Pg_user` (not shown) is a view of `pg_shadow` that does not display the password field. There are several other system views available. Most system tables are joined using OID’s, which are covered in section 7.1. The `psql \dS` command lists all system tables and views.

20.10 Internationalization

PostgreSQL supports several features important for international use. Multi-byte encoding allows non-ASCII character sets to be accurately stored in the database. It can be specified during PostgreSQL initialization, at database creation, or inside `psql`. PostgreSQL can also be installed to support locales.

PostgreSQL can read and display dates in a variety of formats. The default date format can be specified as a `postgres` flag, using `SET DATESTYLE` from inside `psql`, or using the `PGDATESTYLE` environment variable.

20.11 Upgrading

The process of upgrading from previous PostgreSQL releases is covered in the documentation distributed with each version. Sometimes, the `pg_upgrade` utility can be used. In other cases, a `pg_dumpall` and reload are required.

20.12 Summary

This chapter is only a summary of basic administrative tasks. Each utility has many options not covered in this chapter.
20.12. SUMMARY

Administration can be quite challenging. It takes skill and experience. Hopefully this chapter has supplied enough information for you to start exploring topics of interest. The manual pages and Administrators Guide mentioned in appendix A.3 contain more valuable information.
Appendix A

Additional Resources

A.1 Frequently Asked Questions (FAQ’S)

This information comes from http://www.postgresql.org/docs/faq-english.html.

A.2 Mailing List Support

This information comes from http://www.postgresql.org/lists/mailing-list.html.

A.3 Supplied Documentation

This information comes from http://www.postgresql.org/docs/index.html.

A.4 Commercial Support


A.5 Modifying the Source Code

POSTGRESQL allows access to all its source code. The web page http://www.postgresql.org/docs/index.html has a Developers section.
Appendix B

Installation

Getting POSTGRESQL

The POSTGRESQL software is distributed in several formats:

- Tar-gzipped file with file extension .tar.gz
- Prepackaged file with file extension .rpm
- Another prepackaged format
- CD-ROM

Because there are so many formats, this appendix will only cover the general steps need to install POSTGRESQL. Each distribution comes with a INSTALL or README file with more specific instructions.

Create the POSTGRESQL User

It is best to create a separate user to own the POSTGRESQL files and processes that are about to be installed. The user name is typically postgres.

Configure

Many distributions use a configure command which allows users to choose various options before compiling and installing the software.

Compiling

POSTGRESQL is usually distributed in source code format. This means that the C source code must be compiled into a format that is understood by the CPU inside the computer. This process is usually performed by a compiler often called cc or gcc. Several distribution formats automatically perform these steps for the user.

Installation

This process involves copying all compiled programs into a directory that will serve as the home of all POSTGRESQL activity. It will also contain all POSTGRESQL programs, databases, and log files. The directory is typically /usr/local/pgsql.
APPENDIX B. INSTALLATION

Initialization

Initialization creates a database called template1 in the PGHOME home directory. This database is used to create all other databases. initdb performs this initialization step.

Starting the Server

Once template1 is created, the database server can be started. This is typically done by running the program called postmaster.

Creating a Database

Once the database server is running, databases can be created by running createdb from the operating system prompt. Chapter 20 covers PostgreSQL administration in detail.
Appendix C

PostgreSQL Non-Standard Features by Chapter

This section outlines the non-standard features covered in this book:

Chapter 1 None.

Chapter 2 pgsql is a unique feature of PostgreSQL.

Chapter 3 None.

Chapter 4 Use of regular expressions, SET, SHOW, and RESET are features unique to PostgreSQL.

Chapter 5 None.

Chapter 6 None.

Chapter 7 OID’s, sequences, and SERIAL are unique features of PostgreSQL.

Chapter 8 FROM in UPDATE is a unique features of PostgreSQL. Some databases support the creation of tables by SELECT.

Chapter 9 Most databases support only a few of the many datatypes, functions, and operators included in PostgreSQL. Arrays are a unique features of PostgreSQL. Large objects are implemented differently by other database systems.

Chapter 10 None.

Chapter 11 CLUSTER, VACUUM, and EXPLAIN are features unique to PostgreSQL.

Chapter 12 LIMIT is implemented by a few other database systems.

Chapter 13 Inheritance, RULES, LISTEN, and NOTIFY are features unique to PostgreSQL.

Chapter 14 None.

Chapter 15 COPY is a unique feature of PostgreSQL.

Chapter 16 pgsql and pgaccess are unique features of PostgreSQL.

Chapter 17 All the programming interfaces except libecpg and JAVA are implemented differently in other database systems.
Chapter 18  Server-side functions and triggers are implemented differently in other database systems.

Chapter 19  Using C to enhance the database is a unique POSTGRESQL feature.

Chapter 20  The administrative utilities are unique to POSTGRESQL.
Appendix D

Reference Manual

The following is a copy of the reference manual pages (man pages) as they appeared in POSTGRESQL 7.0. These come from http://www.postgresql.org/docs/user/sql-commands.htm and http://www.postgresql.org/docs/user/applications.htm. They are in sgml/Docbook format. Approximately 200 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


[Hilton] Hilton, Craig and Jeff Willis *Building Database Applications on the Web Using PHP3*, Addison–Wesley


