

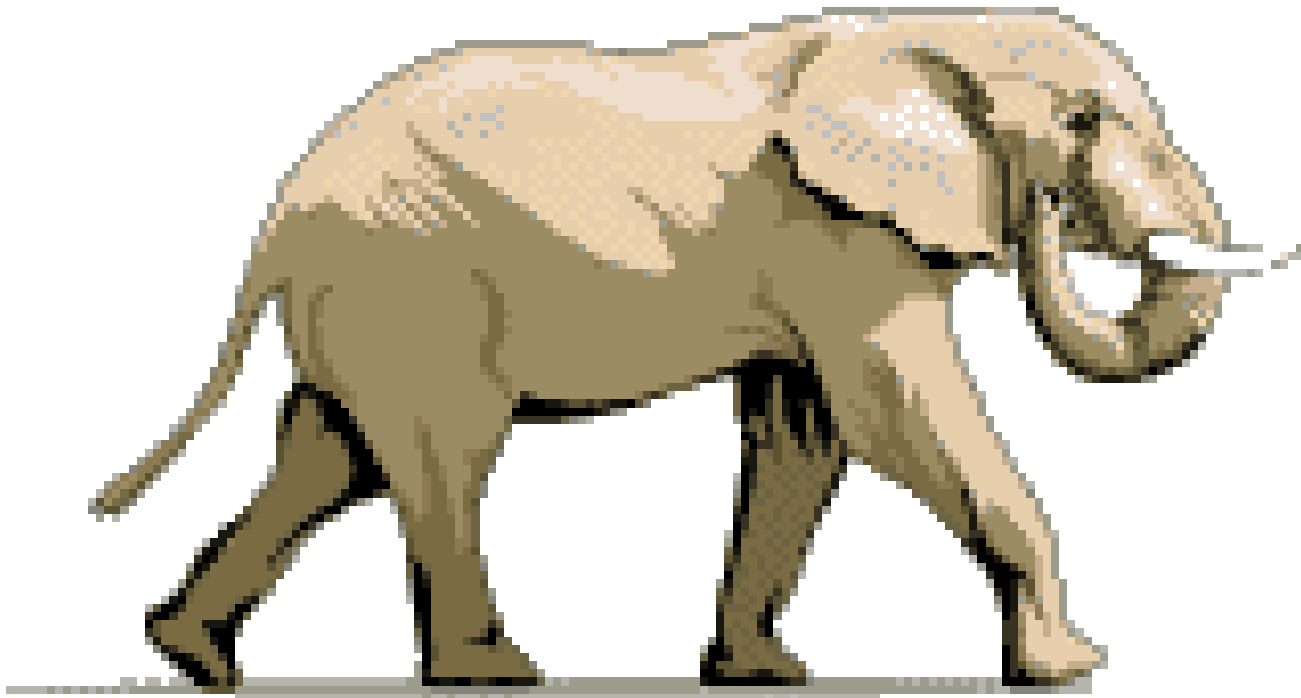
0001
0002
0003
0004
0005
0006
0007
0008
0009
0010
0011
0012
0013
0014
0015
0016
0017
0018
0019
0020
0021
0022
0023
0024
0025
0026
0027
0028
0029
0030
0031
0032
0033
0034
0035
0036
0037
0038
0039
0040
0041
0042
0043
0044
0045
0046
0047
0048
0049
0050
0051
0052
0053
0054
0055
0056
0057
0058
0059
0060
0061
0062
0063
0064
0065
0066

PostgreSQL: Introduction and Concepts

Bruce Momjian

June 17, 2000

WHERE	NULL	CREATE	UNION	AS	DISTINCT
INDEX	TRIGGER	GRANT	ROLLBACK	DEFAULT	SUM
INTO	ALTER	COMMIT	SELECT	REVOKE	CASE
TABLE	FROM	INSERT	OPERATOR	SET	UPDATE
FUNCTION	EXCEPT	DELETE	VALUES	ORDER BY	COUNT
BEGIN WORK	LIKE	IN	VIEW	HAVING	EXISTS



0067
0068
0069
0070
0071
0072
0073
0074
0075
0076
0077
0078
0079
0080
0081
0082
0083
0084
0085
0086
0087
0088
0089
0090
0091
0092
0093
0094
0095
0096
0097
0098
0099
0100
0101
0102
0103
0104
0105
0106
0107
0108
0109
0110
0111
0112
0113
0114
0115
0116
0117
0118
0119
0120
0121
0122
0123
0124
0125
0126
0127
0128
0129
0130
0131
0132

0133
0134
0135
0136
0137
0138
0139
0140
0141
0142
0143
0144
0145
0146
0147
0148
0149
0150
0151
0152
0153
0154
0155
0156
0157
0158
0159
0160
0161
0162
0163
0164
0165
0166
0167
0168
0169
0170
0171
0172
0173
0174
0175
0176
0177
0178
0179
0180
0181
0182
0183
0184
0185
0186
0187
0188
0189
0190
0191
0192
0193
0194
0195
0196
0197
0198

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 URL'S. 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.



0199
0200
0201
0202
0203
0204
0205
0206
0207
0208
0209
0210
0211
0212
0213
0214
0215
0216
0217
0218
0219
0220
0221
0222
0223
0224
0225
0226
0227
0228
0229
0230
0231
0232
0233
0234
0235
0236
0237
0238
0239
0240
0241
0242
0243
0244
0245
0246
0247
0248
0249
0250
0251
0252
0253
0254
0255
0256
0257
0258
0259
0260
0261
0262
0263
0264

0265
0266
0267
0268
0269
0270
0271
0272
0273
0274
0275
0276
0277
0278
0279
0280
0281
0282
0283
0284
0285
0286
0287
0288
0289
0290
0291
0292
0293
0294
0295
0296
0297
0298
0299
0300
0301
0302
0303
0304
0305
0306
0307
0308
0309
0310
0311
0312
0313
0314
0315
0316
0317
0318
0319
0320
0321
0322
0323
0324
0325
0326
0327
0328
0329
0330

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

0331
0332
0333
0334
0335
0336
0337
0338
0339
0340
0341
0342
0343
0344
0345
0346
0347
0348
0349
0350
0351
0352
0353
0354
0355
0356
0357
0358
0359
0360
0361
0362
0363
0364
0365
0366
0367
0368
0369
0370
0371
0372
0373
0374
0375
0376
0377
0378
0379
0380
0381
0382
0383
0384
0385
0386
0387
0388
0389
0390
0391
0392
0393
0394
0395
0396

0397
0398
0399
0400
0401
0402
0403
0404
0405
0406
0407
0408
0409
0410
0411
0412
0413
0414
0415
0416
0417
0418
0419
0420
0421
0422
0423
0424
0425
0426
0427
0428
0429
0430
0431
0432
0433
0434
0435
0436
0437
0438
0439
0440
0441
0442
0443
0444
0445
0446
0447
0448
0449
0450
0451
0452
0453
0454
0455
0456
0457
0458
0459
0460
0461
0462

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>.

0463
0464
0465
0466
0467
0468
0469
0470
0471
0472
0473
0474
0475
0476
0477
0478
0479
0480
0481
0482
0483
0484
0485
0486
0487
0488
0489
0490
0491
0492
0493
0494
0495
0496
0497
0498
0499
0500
0501
0502
0503
0504
0505
0506
0507
0508
0509
0510
0511
0512
0513
0514
0515
0516
0517
0518
0519
0520
0521
0522
0523
0524
0525
0526
0527
0528

0529
0530
0531
0532
0533
0534
0535
0536
0537
0538
0539
0540
0541
0542
0543
0544
0545
0546
0547
0548
0549
0550
0551
0552
0553
0554
0555
0556
0557
0558
0559
0560
0561
0562
0563
0564
0565
0566
0567
0568
0569
0570
0571
0572
0573
0574
0575
0576
0577
0578
0579
0580
0581
0582
0583
0584
0585
0586
0587
0588
0589
0590
0591
0592
0593
0594

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. 0595
0596
0597
- 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. 0598
0599
0600
0601
- MERGL, EDMUND in Stuttgart, Germany created and maintains pgsq1_perl5. He also created DBD-Pg which is available via CPAN. 0602
0603
0604
- MESKES, MICHAEL in Dusseldorf, Germany handles multi-byte foreign language support, and maintains ecpg. 0605
0606
0607
0608
- MOUNT, PETER in Maidstone, Kent, United Kingdom has done the Java JDBC Interface. 0609
0610
- NIKOLAIDIS, BYRON in Baltimore, Maryland, USA rewrote and maintains the ODBC interface for Windows. 0611
0612
- OWEN, LAMAR in Pisgah Forest, North Carolina, USA RPM package maintainer. 0613
0614
- TEODORESCU, CONSTANTIN in Braila, Romania has done the PgAccess DB Interface. 0615
0616
0617
- THYNI, GÖRAN in Kiruna, Sweden has worked on the unix socket code. 0618
0619

Non-code contributors

- BARTUNOV, OLEG in Moscow, Russia introduced the locale support. 0620
0621
0622
0623
0624
0625
- VIELHABER, VINCE near Detroit, Michigan, USA maintains our website. 0626
0627
0628

All developers listed in alphabetical order.

0629
0630
0631
0632
0633
0634
0635
0636
0637
0638
0639
0640
0641
0642
0643
0644
0645
0646
0647
0648
0649
0650
0651
0652
0653
0654
0655
0656
0657
0658
0659
0660

0661
0662
0663
0664
0665
0666
0667
0668
0669
0670
0671
0672
0673
0674
0675
0676
0677
0678
0679
0680
0681
0682
0683
0684
0685
0686
0687
0688
0689
0690
0691
0692
0693
0694
0695
0696
0697
0698
0699
0700
0701
0702
0703
0704
0705
0706
0707
0708
0709
0710
0711
0712
0713
0714
0715
0716
0717
0718
0719
0720
0721
0722
0723
0724
0725
0726

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

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	0731
4.9	LIKE Comparison	28	0732
4.10	Regular Expressions	29	0733
4.11	CASE Clause	33	0734
4.12	Distinct Rows	34	0735
4.13	Functions and Operators	34	0736
4.14	SET, SHOW, and RESET	38	0737
4.15	Summary	38	0738
5	SQL Aggregates	41	0739
5.1	Aggregates	41	0740
5.2	Using GROUP BY	44	0741
5.3	Using HAVING	44	0742
5.4	Query Tips	45	0743
5.5	Summary	46	0744
6	Joining Tables	47	0745
6.1	Table and Column References	47	0746
6.2	Joined Tables	47	0747
6.3	Creating Joined Tables	49	0748
6.4	Performing Joins	51	0749
6.5	Three and Four Table Joins	53	0750
6.6	Additional Join Possibilities	55	0751
6.7	Choosing a Join Key	56	0752
6.8	One-to-Many Joins	57	0753
6.9	Unjoined Tables	59	0754
6.10	Table Aliases and Self-Joins	59	0755
6.11	Non-Equijoins	60	0756
6.12	Ordering Multiple Parts	60	0757
6.13	Primary and Foreign Keys	62	0758
6.14	Summary	62	0759
7	Numbering Rows	65	0760
7.1	Object Identification Numbers (OIDs)	65	0761
7.2	Object Identification Number Limitations	66	0762
7.3	Sequences	67	0763
7.4	Creating Sequences	67	0764
7.5	Using Sequences to Number Rows	69	0765
7.6	Serial Column Type	70	0766
7.7	Manually Numbering Rows	70	0767
7.8	Summary	71	0768
			0769
			0770
			0771
			0772
			0773
			0774
			0775
			0776
			0777
			0778
			0779
			0780
			0781
			0782
			0783
			0784
			0785
			0786
			0787
			0788
			0789
			0790
			0791
			0792

0793	8 Combining SELECTs	73
0794	8.1 UNION, EXCEPT, INTERSECT Clauses	73
0795	8.2 Subqueries	76
0796	8.3 Outer Joins	83
0797	8.4 Subqueries in Non-SELECT Queries	83
0798	8.5 UPDATE with FROM	85
0799	8.6 Inserting Data Using SELECT	85
0800	8.7 Creating Tables Using SELECT	86
0801	8.8 Summary	87
0802		
0803	9 Data Types	89
0804	9.1 Purpose of Data Types	89
0805	9.2 Installed Types	89
0806	9.3 Type Conversion using CAST	93
0807	9.4 Support Functions	93
0808	9.5 Support Operators	93
0809	9.6 Support Variables	96
0810	9.7 Arrays	96
0811	9.8 Large Objects(BLOBS)	98
0812	9.9 Summary	98
0813		
0814	10 Transactions and Locks	99
0815	10.1 Transactions	99
0816	10.2 Multi-Statement Transactions	99
0817	10.3 Visibility of Committed Transactions	101
0818	10.4 Read Committed and Serializable Isolation Levels	102
0819	10.5 Locking	103
0820	10.6 Deadlocks	105
0821	10.7 Summary	105
0822		
0823	11 Performance	107
0824	11.1 Indexes	107
0825	11.2 Unique Indexes	108
0826	11.3 Cluster	108
0827	11.4 Vacuum	109
0828	11.5 Vacuum Analyze	109
0829	11.6 EXPLAIN	109
0830	11.7 Summary	111
0831		
0832	12 Controlling Results	113
0833	12.1 LIMIT	113
0834	12.2 Cursors	114
0835	12.3 Summary	114
0836		
0837	13 Table Management	117
0838	13.1 Temporary Tables	117
0839	13.2 ALTER TABLE	117
0840	13.3 GRANT and REVOKE	119
0841	13.4 Inheritance	120
0842		
0843		
0844		
0845		
0846		
0847		
0848		
0849		
0850		
0851		
0852		
0853		
0854		
0855		
0856		
0857		
0858		

13.5 Views	122	0859
13.6 Rules	123	0860
13.7 LISTEN and NOTIFY	126	0861
13.8 Summary	126	0862
14 Constraints	129	0863
14.1 NOT NULL	129	0864
14.2 UNIQUE	130	0865
14.3 PRIMARY KEY	131	0866
14.4 FOREIGN KEY/REFERENCES	131	0867
14.5 CHECK	139	0868
14.6 Summary	139	0869
15 Importing and Exporting Data	141	0870
15.1 Using COPY	141	0871
15.2 COPY File Format	141	0872
15.3 DELIMITERS	143	0873
15.4 COPY without files	143	0874
15.5 Backslashes and NULLS	144	0875
15.6 COPY Tips	145	0876
15.7 Summary	145	0877
16 Database Query Tools	147	0878
16.1 PSQL	147	0879
16.2 PGACCESS	152	0880
16.3 Summary	154	0881
17 Programming Interfaces	155	0882
17.1 C Language Interface (LIBPQ)	156	0883
17.2 Pgeasy(LIBPGEASY)	158	0884
17.3 Embedded C (ECPG)	158	0885
17.4 C++ (LIBPQ++)	158	0886
17.5 Compiling Programs	158	0887
17.6 Assignment to Program Variables	162	0888
17.7 ODBC	162	0889
17.8 JAVA (JDBC)	163	0890
17.9 Scripting Languages	163	0891
17.10 PERL	163	0892
17.11 TCL/Tk (PGTCLSH/PGTKSH)	163	0893
17.12 PYTHON (PYGRESQL)	163	0894
17.13 PHP	166	0895
17.14 Installing Scripting Languages	168	0896
17.15 Summary	168	0897
18 Functions and Triggers	169	0898
18.1 Functions	169	0899
18.2 SQL Functions	169	0900
18.3 PL/PGSQL Functions	174	0901
18.4 Triggers	180	0902
		0903
		0904
		0905
		0906
		0907
		0908
		0909
		0910
		0911
		0912
		0913
		0914
		0915
		0916
		0917
		0918
		0919
		0920
		0921
		0922
		0923
		0924

0925	18.5 Summary	180
0926		
0927	19 Extending PostgreSQL Using C	183
0928	19.1 Writing C code	183
0929	19.2 Compile the C code	184
0930	19.3 Register the New Functions	184
0931	19.4 Optionally Create Operators, Types, and Aggregates	185
0932	19.5 Summary	185
0933		
0934		
0935		
0936	20 Administration	187
0937	20.1 Files	187
0938	20.2 Creating Users	187
0939	20.3 Creating Databases	189
0940	20.4 Access Configuration	189
0941	20.5 Backup and Restore	191
0942	20.6 Server Startup and Shutdown	192
0943	20.7 Monitoring	192
0944	20.8 Performance	193
0945	20.9 System Tables	194
0946	20.10 Internationalization	194
0947	20.11 Upgrading	194
0948	20.12 Summary	194
0949		
0950		
0951	A Additional Resources	197
0952	A.1 Frequently Asked Questions (FAQ'S)	197
0953	A.2 Mailing List Support	197
0954	A.3 Supplied Documentation	197
0955	A.4 Commercial Support	197
0956	A.5 Modifying the Source Code	197
0957		
0958		
0959		
0960		
0961		
0962		
0963		
0964	B Installation	199
0965		
0966		
0967	C PostgreSQL Non-Standard Features by Chapter	201
0968		
0969	D Reference Manual	203
0970		
0971	Bibliography	205
0972		
0973		
0974	Index	205
0975		
0976		
0977		
0978		
0979		
0980		
0981		
0982		
0983		
0984		
0985		
0986		
0987		
0988		
0989		
0990		

0991
0992
0993
0994
0995
0996
0997
0998
0999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056

List of Figures

1057		
1058		
1059		
1060		
1061		
1062		
1063		
1064		
1065		
1066		
1067		
1068		
1069		
1070	2.1	psql session startup 6
1071	2.2	My first SQL query 6
1072	2.3	Multi-line query 7
1073	2.4	Backslash-p demo 8
1074		
1075		
1076	3.1	Databases 9
1077	3.2	Create table <i>friend</i> 10
1078	3.3	Example of backslash-d 11
1079	3.4	INSERT into <i>friend</i> 12
1080	3.5	Additional <i>friend</i> INSERTS 12
1081	3.6	My first SELECT 13
1082	3.7	My first WHERE 14
1083	3.8	More complex WHERE clause 14
1084	3.9	A single cell 14
1085	3.10	A block of cells 14
1086	3.11	Comparing string fields 15
1087	3.12	DELETE example 16
1088	3.13	My first UPDATE 16
1089	3.14	Use of ORDER BY 17
1090	3.15	Reverse ORDER BY 17
1091	3.16	Use of ORDER BY and WHERE 17
1092		
1093		
1094	4.1	Example of common data types 20
1095	4.2	Insertion of specific columns 21
1096	4.3	NULL handling 22
1097	4.4	Comparison of NULL fields 22
1098	4.5	NULLs and blank strings 23
1099	4.6	Using DEFAULTS 24
1100	4.7	Controlling column labels 24
1101	4.8	Computation using a column label 25
1102	4.9	Comment styles 25
1103	4.10	New friends 26
1104	4.11	WHERE test for <i>Sandy Gleason</i> 26
1105	4.12	Friends in New Jersey and Pennsylvania 27
1106	4.13	Mixing ANDs and ORs 27
1107	4.14	Properly mixing ANDs and ORs 27
1108	4.15	Selecting a range of values 28
1109	4.16	<i>Firstname</i> begins with D. 29
1110	4.17	Regular expression sample queries 31
1111		
1112		
1113		
1114		
1115		
1116		
1117		
1118		
1119		
1120		
1121		
1122		

4.18	Complex regular expression queries	32	1123
4.19	CASE example	33	1124
4.20	Complex CASE example	34	1125
4.21	DISTINCT prevents duplicates	35	1126
4.22	Function examples	36	1127
4.23	Operator examples	37	1128
4.24	SHOW and RESET examples	39	1129
			1130
			1131
			1132
5.1	Aggregate examples	42	1133
5.2	Aggregates and NULLs	43	1134
5.3	Aggregate with GROUP BY	44	1135
5.4	GROUP BY on two columns	45	1136
5.5	HAVING usage	45	1137
			1138
			1139
			1140
6.1	Qualified column names	48	1141
6.2	Joining tables	48	1142
6.3	Creation of company tables	50	1143
6.4	Insertion into company tables	51	1144
6.5	Finding customer name using two queries	51	1145
6.6	Finding customer name using one query	52	1146
6.7	Finding order number for customer name	52	1147
6.8	Three-table join	53	1148
6.9	Four-table join	53	1149
6.10	Employees who have taken orders for customers.	54	1150
6.11	Joining customer and employee	54	1151
6.12	Joining part and employee	55	1152
6.13	Statename table	55	1153
6.14	Using a customer code	56	1154
6.15	One-to-many join	57	1155
6.16	Unjoined tables	58	1156
6.17	Using table aliases	59	1157
6.18	Examples of self-joins using table aliases	59	1158
6.19	Non-equijoins	60	1159
6.20	New salesorder table for multiple parts per order	61	1160
6.21	Orderpart table	61	1161
6.22	Queries involving <i>orderpart</i> table	61	1162
			1163
			1164
			1165
			1166
			1167
			1168
			1169
			1170
			1171
			1172
7.1	OID test	66	1173
7.2	Columns with OIDs	66	1174
7.3	Examples of sequence function use	68	1175
7.4	Numbering <i>customer</i> rows using a sequence	69	1176
7.5	<i>Customer</i> table using SERIAL	70	1177
			1178
			1179
8.1	Combining two columns with UNION	73	1180
8.2	Combining two tables with UNION	74	1181
8.3	UNION with duplicates	74	1182
8.4	UNION ALL with duplicates	75	1183
8.5	EXCEPT restricts output from the first SELECT	75	1184
8.6	INTERSECT returns only duplicated rows	75	1185
			1186
			1187
			1188

1189	8.7 Friends not in <i>Dick Gleason's</i> state	77
1190	8.8 Subqueries can replace some joins	78
1191	8.9 Correlated subquery	79
1192		
1193	8.10 Employees who took orders	80
1194	8.11 Customers who have no orders	81
1195		
1196	8.12 IN query rewritten using ANY and EXISTS	82
1197	8.13 NOT IN query rewritten using ALL and EXISTS	83
1198	8.14 Simulating outer joins	84
1199		
1200	8.15 Subqueries with UPDATE and DELETE	84
1201	8.16 UPDATE the <i>order_date</i>	85
1202	8.17 Using SELECT with INSERT	85
1203		
1204	8.18 Table creation with SELECT	86
1205		
1206	9.1 Example of a function call	93
1207		
1208	9.2 Error generated by undefined function/type combination.	95
1209	9.3 Error generated by undefined operator/type combination	96
1210	9.4 Creation of array columns	96
1211		
1212	9.5 Using arrays	97
1213	9.6 Using large images	98
1214		
1215		
1216	10.1 INSERT with no explicit transaction	99
1217	10.2 INSERT with explicit transaction	100
1218	10.3 Two INSERTs in a single transaction	100
1219		
1220	10.4 Multi-statement transaction	100
1221	10.5 Transaction rollback	101
1222	10.6 Read-committed isolation level	102
1223		
1224	10.7 Serializable isolation level	103
1225	10.8 SELECT with no locking	104
1226		
1227	10.9 SELECT...FOR UPDATE	105
1228		
1229	11.1 Example of CREATE INDEX	107
1230	11.2 Example of a unique index	108
1231	11.3 Using EXPLAIN	109
1232		
1233	11.4 More complex EXPLAIN examples	110
1234	11.5 EXPLAIN example using joins	111
1235		
1236		
1237	12.1 Examples of LIMIT and LIMIT/OFFSET	113
1238	12.2 Cursor usage	115
1239		
1240		
1241	13.1 Temporary table auto-destruction	118
1242	13.2 Example of temporary table use	118
1243	13.3 ALTER TABLE examples	119
1244		
1245	13.4 Examples of the GRANT command	120
1246	13.5 Creation of inherited tables	120
1247	13.6 Accessing inherited tables	121
1248		
1249	13.7 Inheritance in layers	121
1250	13.8 Examples of views	122
1251		
1252	13.9 Rule that prevents INSERT	123
1253	13.10 Rules to log table changes	124
1254		

13.11	Use of rule to log table changes	125	1255
13.12	Views ignore table modifications	125	1256
13.13	Rules to handle view modifications	126	1257
13.14	Rules handle view modifications	127	1258
			1259
			1260
14.1	NOT NULL constraint	129	1261
14.2	NOT NULL with DEFAULT constraint	130	1262
14.3	Unique column constraint	130	1263
14.4	Multi-column unique constraint	131	1264
14.5	Creation of PRIMARY KEY column	131	1265
14.6	Example of a multi-column primary key	132	1266
14.7	Foreign key creation	132	1267
14.8	Foreign key constraints	133	1268
14.9	Creation of company tables using primary and foreign keys	133	1269
14.10	Customer table with foreign key actions	134	1270
14.11	Foreign key actions	135	1271
14.12	Example of a multi-column foreign key	136	1272
14.13	MATCH FULL foreign key	137	1273
14.14	DEFERRABLE foreign key constraint	138	1274
14.15	CHECK constraints	139	1275
			1276
			1277
15.1	Example of COPY...TO and COPY...FROM	142	1278
15.2	Example of COPY...FROM	142	1279
15.3	Example of COPY...TO...USING DELIMITERS	143	1280
15.4	Example of COPY...FROM...USING DELIMITERS	143	1281
15.5	COPY using <i>stdin</i> and <i>stdout</i>	144	1282
15.6	COPY backslash handling	144	1283
			1284
			1285
16.1	Example of <i>\pset</i>	149	1286
16.2	psql variables	150	1287
16.3	Pgaccess opening window	153	1288
16.4	Pgaccess table window	154	1289
			1290
			1291
17.1	Sample application being run	155	1292
17.2	<i>Statename</i> table	156	1293
17.3	Libpq data flow	156	1294
17.4	<i>Libpq</i> sample program	157	1295
17.5	<i>libpqeasy</i> sample program	159	1296
17.6	<i>Ecpq</i> sample program	160	1297
17.7	<i>Libpq++</i> sample program	161	1298
17.8	JAVA sample program	164	1299
17.9	PERL sample program	165	1300
17.10	TCL sample program	165	1301
17.11	PYTHON sample program	166	1302
17.12	PHP sample program — Input	166	1303
17.13	PHP sample program – Output	167	1304
			1305
			1306
			1307
			1308
			1309
			1310
			1311
			1312
			1313
			1314
			1315
			1316
			1317
18.1	SQL <i>floc</i> function	170	1318
18.2	SQL <i>tax</i> function	171	1319
			1320

1321	18.3 Recreation of the <i>part</i> table	171
1322	18.4 SQL <i>shipping</i> function	172
1323	18.5 SQL function <i>getstatename</i>	173
1324	18.6 Getting state name using join and function	173
1326	18.7 PL/PGSQL version of <i>getstatename</i>	174
1327	18.8 PL/PGSQL <i>spread</i> function	175
1328	18.9 PL/PGSQL <i>getstatecode</i> function	176
1329	18.10 Calls to <i>getstatecode</i> function	177
1330	18.11 PL/PGSQL <i>change_statename</i> function	178
1331	18.12 Example of <i>change_statename()</i>	179
1332	18.13 Trigger creation	181
1333		
1334	19.1 C <i>ctof</i> function	184
1335	19.2 Create function <i>ctof</i>	184
1336	19.3 Calling function <i>ctof</i>	185
1337		
1338	20.1 Examples of user administration	188
1339	20.2 Examples of database creation and removal	189
1340	20.3 Making a new copy of database test	191
1341	20.4 Postmaster and postgres processes	192
1342		
1343		
1344		
1345		
1346		
1347		
1348		
1349		
1350		
1351		
1352		
1353		
1354		
1355		
1356		
1357		
1358		
1359		
1360		
1361		
1362		
1363		
1364		
1365		
1366		
1367		
1368		
1369		
1370		
1371		
1372		
1373		
1374		
1375		
1376		
1377		
1378		
1379		
1380		
1381		
1382		
1383		
1384		
1385		
1386		

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

1453		
1454		
1455		
1456		
1457		
1458		
1459		
1460		
1461		
1462		
1463		
1464		
1465		
1466	3.1 Table <i>friend</i>	10
1467		
1468	4.1 Common data types	19
1469	4.2 Comparisons	27
1470	4.3 LIKE comparison	29
1471	4.4 Regular expression operators	29
1472	4.5 Regular expression special characters	30
1473	4.6 Regular expression examples	30
1474	4.7 SET options	38
1475	4.8 DATESTYLE output	38
1476		
1477		
1478		
1479		
1480	5.1 Aggregates	41
1481		
1482		
1483	7.1 Sequence number access functions	67
1484		
1485	9.1 POSTGRESQL data types	90
1486	9.2 Geometric types	92
1487	9.3 Common functions	94
1488	9.4 Common operators	95
1489	9.5 Common variables	96
1490		
1491		
1492		
1493	10.1 Visibility of single-query transactions	101
1494	10.2 Visibility using multi-query transactions	102
1495	10.3 Waiting for a lock	104
1496	10.4 Deadlock	105
1497		
1498		
1499		
1500	13.1 Temporary table isolation	117
1501		
1502		
1503	15.1 Backslashes understood by COPY	145
1504		
1505	16.1 psql query buffer commands	147
1506	16.2 psql general commands	148
1507	16.3 psql \pset options	148
1508	16.4 psql output format shortcuts	149
1509	16.5 psql predefined variables	151
1510	16.6 psql listing commands	152
1511	16.7 psql large object commands	152
1512	16.8 psql command-line arguments	153
1513		
1514		
1515		
1516		
1517	17.1 Interface summary	155
1518		

20.1 Commonly used system tables	194	1519
		1520
		1521
		1522
		1523
		1524
		1525
		1526
		1527
		1528
		1529
		1530
		1531
		1532
		1533
		1534
		1535
		1536
		1537
		1538
		1539
		1540
		1541
		1542
		1543
		1544
		1545
		1546
		1547
		1548
		1549
		1550
		1551
		1552
		1553
		1554
		1555
		1556
		1557
		1558
		1559
		1560
		1561
		1562
		1563
		1564
		1565
		1566
		1567
		1568
		1569
		1570
		1571
		1572
		1573
		1574
		1575
		1576
		1577
		1578
		1579
		1580
		1581
		1582
		1583
		1584

1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650

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 CORPORATION¹, 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 ILLUSTRATE² 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

¹Ingres Corp. was later purchased by Computer Associates.

²Illustrate was later purchased by Informix and integrated into Informix's Universal Server.

³cvs 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 C⁴ 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 POSTGRESQL Global Development Team

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

Releases 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.⁵ We wrote a developers' FAQ⁶, to describe some of the common questions of POSTGRESQL developers. With this, developers became more productive at fixing bugs and adding features.

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

⁴C is a popular computer language first developed in the 1970's.

⁵All the files mentioned in this chapter are available as part of the POSTGRESQL distribution, or at <http://www.postgresql.org/docs>.

⁶Frequently Asked Questions

1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716

1717 styles were also quite varied.

1718 We wrote a tool to reformat the entire source tree in a consistent manner. We wrote a script to find
1719 functions that could be marked as *static*⁷, or unused functions that could be removed completely. These are
1720 run just before each release. A release checklist reminds us of the items to be changed for each release.

1721 As we gained knowledge of the code, we were able to perform more complicated fixes and feature
1722 additions. We redesigned poorly structured code. We moved into a mode where each release had major new
1723 features, instead of just bug fixes. We improved SQL conformance, added sub-selects, improved locking, and
1724 added missing SQL functionality. A company formed to offer telephone support.

1725 The Usenet discussion group archives started touting us. In the previous year, we searched for POST-
1726 GRESQL, and found many people were recommending other databases, even though we were addressing user
1727 concerns as rapidly as possible. One year later, many people were recommending us to users who needed
1728 transaction support, complex queries, commercial-grade SQL support, complex data types, and reliability.
1729 This clearly portrayed our strengths. Other databases were recommended when speed was the overriding
1730 concern. REDHAT's shipment of POSTGRESQL as part of their LINUX⁸ distribution quickly expanded our user
1731 base.

1732 Every release is now a major improvement over the last. Our global development team now has mastery of
1733 the source code we inherited from Berkeley. Finally, every module is understood by at least one development
1734 team member. We are now easily adding major features, thanks to the increasing size and experience of our
1735 world-wide development team.

1743 1.5 Open Source Software

1744 POSTGRESQL is *open source software*. The term *open source software* often confuses people. With commercial
1745 software, a company hires programmers, develops a product, and sells it to users. With Internet communi-
1746 cation, there are new possibilities. In *open source software*, there is no company. Capable programmers with
1747 interest and some free time get together via the Internet and exchange ideas. Someone writes a program
1748 and puts it in a place everyone can access. Other programmers join and make changes. When the program
1749 is sufficiently functional, they advertise the program's availability to other Internet users. Users find bugs
1750 or missing features and report them back to the developers, who enhance the program.

1751 It sounds like an unworkable cycle, but in fact it has several advantages:

- 1752 • A company structure is not required, so there is no overhead and no economic restrictions.
- 1753 • Program development is not limited to a hired programming staff, but taps the capabilities and experi-
1754 ence of a large pool of Internet programmers.
- 1755 • User feedback is facilitated, allowing program testing by a large number of users in a short period of
1756 time.
- 1757 • Program enhancements can be rapidly distributed to users.

1770 1.6 Summary

1771 This chapter has explored the long history of POSTGRESQL, starting with its roots in university research.
1772 POSTGRESQL would not have grown to the success it is today without the Internet. The ability to commu-
1773 nicate with people around the world has allowed a community of unpaid developers to enhance and support
1774

1780 ⁷A *static* function is a function that is used by only one program file.

1781 ⁸Linux is a popular UNIX-like, open source operating system.

software that rivals commercial database offerings. By allowing everyone to see the source code and contribute, POSTGRESQL continues to improve every day. The remainder of this book shows how to use this amazing piece of software.

1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848

1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914

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

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

test=>
```

Figure 2.2: My first SQL query

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

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

Typing in the Query Buffer

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

¹A few operating systems are case-insensitive.

1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980

```

1981 when you enter a semicolon (;) or backslash-g (\g). Here's a good example. Let's do SELECT 1 + 3; but in
1982 a different way. See figure 2.3.2 Notice the query is spread over three lines. Notice the prompt changed
1983
1984     test=> SELECT
1985     test-> 1 + 3
1986     test-> ;
1987         ?column?
1988     -----
1989         4
1990
1991     (1 row)
1992
1993
1994
1995     test=>
1996
1997
1998
1999
2000

```

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*³ 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.

²Don'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
test-> 2 * 10 + 1
test-> \p
SELECT
2 * 10 + 1
test-> \g
?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.

2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112

2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178

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

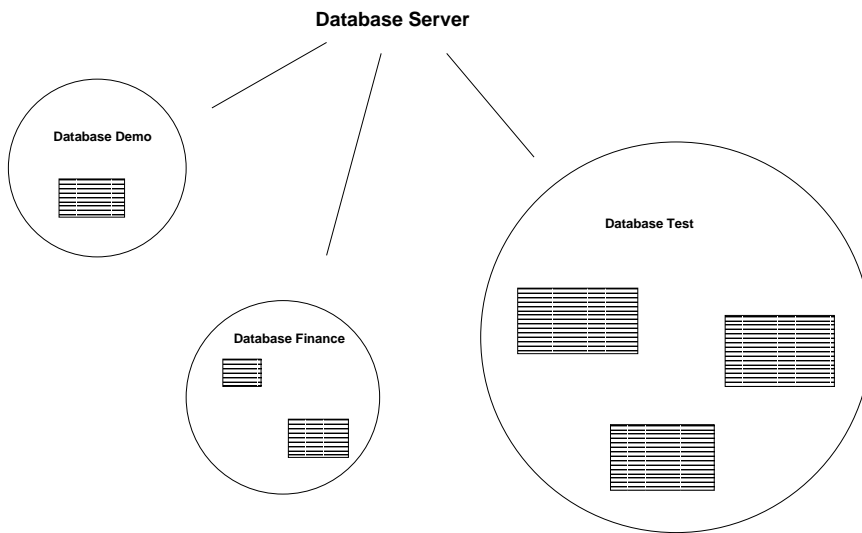


Figure 3.1: Databases

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

You might ask, “What are those black rectangles in the databases?” Those are *tables*. Tables are the foundation of a *relational database management system* (RDBMS). As I mentioned earlier, databases store data.

Those tables are where data is stored in a database. Each table has a name defined by the person who created it.

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

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

Table 3.1: Table *friend*

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

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

3.2 Creating Tables

Let's create our own table and call it *friend*. The `psql` statement to create the table is shown in figure 3.2. You do not have to type it exactly like that. You could have used all lowercase, or you could have written it

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

Figure 3.2: Create table *friend*

in one long line, and it would have worked just the same.

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.

¹In a real-world database, the person's birth date would be stored and not the person's age. Age has to be updated every time the person has a birthday. A person's age can be computed when needed from a birth date field.

2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244

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 strings² 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.

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

Create some tables yourself now. Only use letters for your table and column names. 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.

²A character string is a group of characters *strung* together.

```

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

```

Figure 3.4: INSERT into *friend*

```

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

```

Figure 3.5: Additional *friend* INSERTS

2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376

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.

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

Figure 3.6: My first SELECT

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.

```

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

2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508

Try a few more until you are comfortable.

```

2509
2510
2511     test=> SELECT * FROM friend WHERE firstname = 'Sam';
2512           firstname |      lastname      |      city      | state | age
2513           -----+-----+-----+-----+-----
2514           Sam       | Jackson           | Allentown     | PA   | 22
2515           (1 row)
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525

```

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.

2574

```

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

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

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

```

Figure 3.12: DELETE example

```

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

```

Figure 3.13: My first UPDATE

2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640


```

2641
2642
2643
2644
2645 test=> SELECT * FROM friend ORDER BY state;
2646      firstname |      lastname      |      city      | state | age
2647 -----+-----+-----+-----+-----
2648      Cindy     | Anderson          | Denver         | CO    | 23
2649      Mike      | Nichols           | Tampa          | FL    | 20
2650      Sam       | Jackson           | Allentown      | PA    | 22
2651
2652 (3 rows)
2653
2654
2655

```

Figure 3.14: Use of ORDER BY

```

2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667 test=> SELECT * FROM friend ORDER BY age DESC;
2668      firstname |      lastname      |      city      | state | age
2669 -----+-----+-----+-----+-----
2670      Cindy     | Anderson          | Denver         | CO    | 23
2671      Sam       | Jackson           | Allentown      | PA    | 22
2672      Mike      | Nichols           | Tampa          | FL    | 20
2673
2674 (3 rows)
2675
2676
2677

```

Figure 3.15: Reverse ORDER BY

```

2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690 test=> SELECT * FROM friend WHERE age >= 21 ORDER BY firstname;
2691      firstname |      lastname      |      city      | state | age
2692 -----+-----+-----+-----+-----
2693      Cindy     | Anderson          | Denver         | CO    | 23
2694      Sam       | Jackson           | Allentown      | PA    | 22
2695
2696 (2 rows)
2697
2698
2699

```

Figure 3.16: Use of ORDER BY and WHERE

```

2700
2701
2702
2703
2704
2705
2706

```

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 `DROP TABLE friend` will remove the *friend* table. Both the table structure and the data contained in the table will be erased. We will be using the *friend* table in the next chapter, so I 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 have shown the basic operations of any database:

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

2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772

2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838

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

Category	Type	Description
character string	char(length)	blank-padded string, fixed storage length
	varchar(length)	variable storage length
number	integer	integer, +/-2 billion range
	float	floating point number, 15-digit precision
	numeric(precision, decimal)	number with user-defined precision and decimal location
date/time	date	date
	time	time
	timestamp	date and time

Table 4.1: Common data types

is table creation, INSERT, and SELECT. There are a few things of interest in this example. First, notice how the numbers do not require quotes, while character strings, dates, and times require them. Also note the *timestamp* column displays its value in the standard UNIX date¹ 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.

¹This is the format generated by typing the command date at the UNIX command prompt.

²See section 16.1 for a full list of the psql backslash commands.

```

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

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

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

```

Figure 4.1: Example of common data types

One way to place a single quote inside a single-quoted string is to use two quotes together like this, 'O' 'Donne11'.³ Two single quotes inside a single-quoted string cause one single quote to be generated. Another way is to use a backslash like this, 'O\'Donne11'. The backslash *escapes* the single quote character.

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

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

Figure 4.2: Insertion of specific columns

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.⁴ 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.

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

⁴The <> means *not equal*.

```

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

```

```

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

```

```

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

```

```

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

```

Figure 4.3: NULL handling

```

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

```

Figure 4.4: Comparison of NULL fields

2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3030
3031
3032
3033
3034
3035
3036

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

```

3037
3038
3039
3040
3041
3042     test=> CREATE TABLE nulltest (name CHAR(20), spouse CHAR(20));
3043     CREATE
3044
3045     test=> INSERT INTO nulltest VALUES ('Andy', '');
3046     INSERT 19086 1
3047
3048     test=> INSERT INTO nulltest VALUES ('Tom', NULL);
3049     INSERT 19087 1
3050
3051     test=> SELECT * FROM nulltest ORDER BY name;
3052           name          |          spouse
3053     -----+-----
3054     Andy                |
3055     Tom                 |
3056     (2 rows)
3057
3058
3059
3060     test=> SELECT * FROM nulltest WHERE spouse = '';
3061           name          |          spouse
3062     -----+-----
3063     Andy                |
3064     (1 row)
3065
3066
3067
3068     test=> SELECT * FROM nulltest WHERE spouse IS NULL;
3069           name          |          spouse
3070     -----+-----
3071     Tom                 |
3072     (1 row)
3073
3074
3075
3076
3077
3078
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
3100
3101
3102

```

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 (
test(>     name      CHAR(20),
test(>     balance  NUMERIC(16,2) DEFAULT 0,
test(>     active   CHAR(1) DEFAULT 'Y',
test(>     created  TIMESTAMP DEFAULT CURRENT_TIMESTAMP
test(> );
CREATE
test=> INSERT INTO account (name)
test-> VALUES ('Federated Builders');
INSERT 19103 1
test=> SELECT * FROM account;
      name      | balance | active |      created
-----+-----+-----+-----
Federated Builders |    0.00 | Y      | 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

```

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

```

Figure 4.7: Controlling column labels

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.

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`

3103
3104
3105
3106
3107
3108
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
3150
3151
3152
3153
3154
3155
3156
3157
3158
3159
3160
3161
3162
3163
3164
3165
3166
3167
3168


```

3169         test=> SELECT 1 + 3 AS total;
3170         total
3171         -----
3172             4
3173         (1 row)
3174
3175
3176
3177
3178
3179
3180

```

Figure 4.8: Computation using a column label

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

```

3185         test=> -- a single line comment
3186         test=> /* a multi-line
3187         test*>  comment */
3188
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3199
3200

```

Figure 4.9: Comment styles

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.

```

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

```

Figure 4.10: New friends

```

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

```

Figure 4.11: WHERE test for *Sandy Gleason*

3235
3236
3237
3238
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
3250
3251
3252
3253
3254
3255
3256
3257
3258
3259
3260
3261
3262
3263
3264
3265
3266
3267
3268
3269
3270
3271
3272
3273
3274
3275
3276
3277
3278
3279
3280
3281
3282
3283
3284
3285
3286
3287
3288
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299
3300

```

3301 test=> SELECT * FROM friend
3302 test-> WHERE state = 'NJ' OR state = 'PA'
3303 test-> ORDER BY firstname;
3304
3305     firstname |      lastname      |      city      | state | age
3306 -----+-----+-----+-----+-----
3307 Dick         | Gleason            | Ocean City     | NJ    | 19
3308 Sandy        | Gleason            | Ocean City     | NJ    | 25
3309 Victor       | Tabor              | Williamsport   | PA    | 22
3310
3311 (3 rows)
3312
3313
3314
3315
3316
3317
3318
3319
3320
3321
3322
3323
3324
3325
3326
3327
3328
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349
3350
3351
3352
3353
3354
3355
3356
3357
3358
3359
3360
3361
3362
3363
3364
3365
3366

```

Figure 4.12: Friends in New Jersey and Pennsylvania

```

3319 test=> SELECT * FROM friend
3320 test-> WHERE firstname = 'Victor' AND state = 'PA' OR state = 'NJ'
3321 test-> ORDER BY firstname;
3322
3323     firstname |      lastname      |      city      | state | age
3324 -----+-----+-----+-----+-----
3325 Dick         | Gleason            | Ocean City     | NJ    | 19
3326 Sandy        | Gleason            | Ocean City     | NJ    | 25
3327 Victor       | Tabor              | Williamsport   | PA    | 22
3328
3329 (3 rows)
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349
3350
3351
3352
3353
3354
3355
3356
3357
3358
3359
3360
3361
3362
3363
3364
3365
3366

```

Figure 4.13: Mixing ANDs and ORs

```

3337 test=> SELECT * FROM friend
3338 test-> WHERE firstname = 'Victor' AND (state = 'PA' OR state = 'NJ')
3339 test-> ORDER BY firstname;
3340
3341     firstname |      lastname      |      city      | state | age
3342 -----+-----+-----+-----+-----
3343 Victor       | Tabor              | Williamsport   | PA    | 22
3344
3345 (1 row)
3346
3347
3348
3349
3350
3351
3352
3353
3354
3355
3356
3357
3358
3359
3360
3361
3362
3363
3364
3365
3366

```

Figure 4.14: Properly mixing ANDs and ORs

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

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

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

firstname	lastname	city	state	age
Dean	Yeager	Plymouth	MA	24
Sandy	Gleason	Ocean City	NJ	25
Victor	Tabor	Williamsport	PA	22

(3 rows)

```
test=> SELECT *
test-> FROM friend
test-> WHERE age BETWEEN 22 AND 25
test-> ORDER BY firstname;
```

firstname	lastname	city	state	age
Dean	Yeager	Plymouth	MA	24
Sandy	Gleason	Ocean City	NJ	25
Victor	Tabor	Williamsport	PA	22

(3 rows)

Figure 4.15: Selecting a range of values

`<=` 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 ().

3367
3368
3369
3370
3371
3372
3373
3374
3375
3376
3377
3378
3379
3380
3381
3382
3383
3384
3385
3386
3387
3388
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3399
3400
3401
3402
3403
3404
3405
3406
3407
3408
3409
3410
3411
3412
3413
3414
3415
3416
3417
3418
3419
3420
3421
3422
3423
3424
3425
3426
3427
3428
3429
3430
3431
3432

```

3433 test=> SELECT * FROM friend
3434 test-> WHERE firstname LIKE 'D%'
3435 test-> ORDER BY firstname;
3436      firstname |      lastname      |      city      | state | age
3437 -----+-----+-----+-----+-----
3438      Dean      |      Yeager        |      Plymouth  | MA    | 24
3439      Dick      |      Gleason       |      Ocean City | NJ    | 19
3440
3441 (2 rows)
3442
3443
3444
3445
3446
3447

```

Figure 4.16: *Firstname* begins with D.

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

Table 4.3: LIKE comparison

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

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

Table 4.4: Regular expression operators

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

⁵Actually, PostgreSQL regular expressions are like `egrep` extended regular expressions.

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

Table 4.5: Regular expression special characters

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

Table 4.6: Regular expression examples

3499
3500
3501
3502
3503
3504
3505
3506
3507
3508
3509
3510
3511
3512
3513
3514
3515
3516
3517
3518
3519
3520
3521
3522
3523
3524
3525
3526
3527
3528
3529
3530
3531
3532
3533
3534
3535
3536
3537
3538
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549
3550
3551
3552
3553
3554
3555
3556
3557
3558
3559
3560
3561
3562
3563
3564

```

3565 test=> SELECT * FROM friend
3566 test-> ORDER BY firstname;
3567      firstname |      lastname |      city | state | age
3568 -----+-----+-----+-----+-----
3569      Dean      |      Yeager   | Plymouth | MA    | 24
3570      Dick      |      Gleason  | Ocean City | NJ    | 19
3571      Ned       |      Millstone | Cedar Creek | MD    | 27
3572      Sandy     |      Gleason  | Ocean City | NJ    | 25
3573      Sandy     |      Weber    | Boston    | MA    | 33
3574      Victor   |      Tabor    | Williamsport | PA    | 22
3575      (6 rows)
3576
3577
3578 test=> -- firstname begins with 'S'
3579 test=> SELECT * FROM friend
3580 test-> WHERE firstname ~ '^S'
3581 test-> ORDER BY firstname;
3582      firstname |      lastname |      city | state | age
3583 -----+-----+-----+-----+-----
3584      Sandy     |      Gleason  | Ocean City | NJ    | 25
3585      Sandy     |      Weber    | Boston    | MA    | 33
3586      (2 rows)
3587
3588
3589
3590 test=> -- firstname has an e in the second position
3591 test=> SELECT * FROM friend
3592 test-> WHERE firstname ~ '^e'
3593 test-> ORDER BY firstname;
3594      firstname |      lastname |      city | state | age
3595 -----+-----+-----+-----+-----
3596      Dean      |      Yeager   | Plymouth | MA    | 24
3597      Ned       |      Millstone | Cedar Creek | MD    | 27
3598      (2 rows)
3599
3600
3601
3602 test=> -- firstname contains b, B, c or C
3603 test=> SELECT * FROM friend
3604 test-> WHERE firstname ~* '[bc]'
3605 test-> ORDER BY firstname;
3606      firstname |      lastname |      city | state | age
3607 -----+-----+-----+-----+-----
3608      Dick      |      Gleason  | Ocean City | NJ    | 19
3609      Victor   |      Tabor    | Williamsport | PA    | 22
3610      (2 rows)
3611
3612
3613
3614 test=> -- firstname does not contain s or S
3615 test=> SELECT * FROM friend
3616 test-> WHERE firstname !~* 's'
3617 test-> ORDER BY firstname;
3618      firstname |      lastname |      city | state | age
3619 -----+-----+-----+-----+-----
3620      Dean      |      Yeager   | Plymouth | MA    | 24
3621      Dick      |      Gleason  | Ocean City | NJ    | 19
3622      Ned       |      Millstone | Cedar Creek | MD    | 27
3623      Victor   |      Tabor    | Williamsport | PA    | 22
3624      (4 rows)
3625
3626
3627
3628
3629
3630

```

Figure 4.17: Regular expression sample queries

```

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

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

```

Figure 4.18: Complex regular expression queries

3631
3632
3633
3634
3635
3636
3637
3638
3639
3640
3641
3642
3643
3644
3645
3646
3647
3648
3649
3650
3651
3652
3653
3654
3655
3656
3657
3658
3659
3660
3661
3662
3663
3664
3665
3666
3667
3668
3669
3670
3671
3672
3673
3674
3675
3676
3677
3678
3679
3680
3681
3682
3683
3684
3685
3686
3687
3688
3689
3690
3691
3692
3693
3694
3695
3696

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*.

```

3726 test-> SELECT  firstname,
3727             age,
3728             CASE
3729             WHEN age >= 21 THEN 'adult'
3730             ELSE 'minor'
3731             END
3732             FROM friend
3733             ORDER BY  firstname;
3734
3735      firstname | age | case
3736 -----+-----+-----
3737      Dean      | 24 | adult
3738      Dick      | 19 | minor
3739      Ned       | 27 | adult
3740      Sandy     | 25 | adult
3741      Sandy     | 33 | adult
3742      Victor   | 22 | adult
3743
3744 (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.

```

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

```

Figure 4.20: Complex CASE example

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

```
3829
3830
3831
3832
3833
3834
3835
3836
3837
3838
3839
3840 test=> SELECT state FROM friend ORDER BY state;
3841     state
3842     -----
3843     MA
3844     MA
3845     MD
3846     NJ
3847     NJ
3848     PA
3849     (6 rows)
3850
3851
3852
3853
3854 test=> SELECT DISTINCT state FROM friend ORDER BY state;
3855     state
3856     -----
3857     MA
3858     MD
3859     NJ
3860     PA
3861     (4 rows)
3862
3863
3864
3865
3866 test=> SELECT DISTINCT city, state FROM friend ORDER BY state, city;
3867     city      | state
3868     -----+-----
3869     Boston    | MA
3870     Plymouth  | MA
3871     Cedar Creek | MD
3872     Ocean City | NJ
3873     Williamsport | PA
3874     (5 rows)
3875
3876
3877
3878
3879
3880
3881
3882
3883
3884
3885
3886
3887
3888
3889
3890
3891
3892
3893
3894
```

Figure 4.21: DISTINCT prevents duplicates

```

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

3895
3896
3897
3898
3899
3900
3901
3902
3903
3904
3905
3906
3907
3908
3909
3910
3911
3912
3913
3914
3915
3916
3917
3918
3919
3920
3921
3922
3923
3924
3925
3926
3927
3928
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
3950
3951
3952
3953
3954
3955
3956
3957
3958
3959
3960

```

3961
3962
3963 test=> \do
3964
3965                               List of operators
3966 Op | Left arg | Right arg | Result | Description
3967
3968 -----+-----+-----+-----+-----
3969
3970 -----
3971 ! | int2      |          | int4    |
3972 ! | int4      |          | int4    | factorial
3973 ! | int8      |          | int8    | factorial
3974 !! |          | int2    | int4    |
3975 ...
3976
3977 test=> \do /
3978
3979                               List of operators
3980 Op | Left arg | Right arg | Result | Description
3981
3982 -----+-----+-----+-----+-----
3983 / | box      | point    | box     | divide box by point (scale)
3984 / | char     | char     | char    | divide
3985 / | circle   | point    | circle  | divide
3986 / | float4   | float4   | float4  | divide
3987 ...
3988
3989 test=> \do ^
3990
3991                               List of operators
3992 Op | Left arg | Right arg | Result | Description
3993
3994 -----+-----+-----+-----+-----
3995 ^ | float8   | float8   | float8  | exponentiation (x^y)
3996 (1 row)
3997
3998 test=> \dd ^
3999
4000                               Object descriptions
4001 Name | Object | Description
4002
4003 -----+-----+-----
4004 ^ | operator | exponentiation (x^y)
4005 (1 row)
4006
4007 test=> SELECT 2 + 3 ^ 4;
4008 ?column?
4009 -----
4010 83
4011 (1 row)
4012
4013
4014
4015
4016
4017
4018
4019
4020
4021
4022
4023
4024
4025
4026

```

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.

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

Table 4.7: SET options

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

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

Table 4.8: DATESTYLE output

format (slashes, dashes, or year first), and the display of the month first (US) or day first (European). The command SET DATESTYLE TO 'SQL,US' would most likely be 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.⁶

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.

```
4093
4094
4095
4096
4097
4098
4099
4100
4101
4102
4103
4104
4105
4106
4107
4108
4109
4110
4111
4112
4113
4114
4115     test=> SHOW DATESTYLE;
4116     NOTICE: DateStyle is ISO with US (NonEuropean) conventions
4117     SHOW VARIABLE
4118     test=> SET DATESTYLE TO 'SQL, EUROPEAN';
4119     SET VARIABLE
4120     test=> SHOW DATESTYLE;
4121     NOTICE: DateStyle is SQL with European conventions
4122     SHOW VARIABLE
4123     test=> RESET DATESTYLE;
4124     RESET VARIABLE
4125     test=> SHOW DATESTYLE;
4126     NOTICE: DateStyle is ISO with US (NonEuropean) conventions
4127     SHOW VARIABLE
4128
4129
4130
4131
4132
4133
4134
4135
4136
4137
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
4150
4151
4152
4153
4154
4155
4156
4157
4158
```

Figure 4.24: SHOW and RESET examples

4159
4160
4161
4162
4163
4164
4165
4166
4167
4168
4169
4170
4171
4172
4173
4174
4175
4176
4177
4178
4179
4180
4181
4182
4183
4184
4185
4186
4187
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
4200
4201
4202
4203
4204
4205
4206
4207
4208
4209
4210
4211
4212
4213
4214
4215
4216
4217
4218
4219
4220
4221
4222
4223
4224

4225
4226
4227
4228
4229
4230
4231
4232
4233
4234
4235
4236
4237
4238
4239
4240
4241
4242
4243
4244
4245
4246
4247
4248
4249
4250
4251
4252
4253
4254
4255
4256
4257
4258
4259
4260
4261
4262
4263
4264
4265
4266
4267
4268
4269
4270
4271
4272
4273
4274
4275
4276
4277
4278
4279
4280
4281
4282
4283
4284
4285
4286
4287
4288
4289
4290

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

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

Table 5.1: Aggregates

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

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

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

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

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

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

```

Figure 5.1: Aggregate examples

4291
4292
4293
4294
4295
4296
4297
4298
4299
4300
4301
4302
4303
4304
4305
4306
4307
4308
4309
4310
4311
4312
4313
4314
4315
4316
4317
4318
4319
4320
4321
4322
4323
4324
4325
4326
4327
4328
4329
4330
4331
4332
4333
4334
4335
4336
4337
4338
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
4350
4351
4352
4353
4354
4355
4356

```
4357      test=> CREATE TABLE aggtest (col INTEGER);
4358      CREATE
4359      test=> INSERT INTO aggtest VALUES (NULL);
4360      INSERT 19759 1
4361      test=> SELECT SUM(col) FROM aggtest;
4362      sum
4363      -----
4364
4365      (1 row)
4366
4367      test=> SELECT MAX(col) FROM aggtest;
4368      max
4369      -----
4370
4371      (1 row)
4372
4373      test=> SELECT COUNT(*) FROM aggtest;
4374      count
4375      -----
4376      1
4377      (1 row)
4378
4379      test=> SELECT COUNT(col) FROM aggtest;
4380      count
4381      -----
4382      0
4383      (1 row)
4384
4385      test=> INSERT INTO aggtest VALUES (3);
4386      INSERT 19760 1
4387      test=> SELECT AVG(col) FROM aggtest;
4388      avg
4389      -----
4390      3
4391      (1 row)
4392
4393      test=> SELECT COUNT(*) FROM aggtest;
4394      count
4395      -----
4396      2
4397      (1 row)
4398
4399      test=> SELECT COUNT(col) FROM aggtest;
4400      count
4401      -----
4402      1
4403      (1 row)
4404
4405
4406
4407
4408
4409
4410
4411
4412
4413
4414
4415
4416
4417
4418
4419
4420
4421
4422
```

Figure 5.2: Aggregates and NULLs

5.2 Using GROUP BY

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

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

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

Figure 5.3: Aggregate with GROUP BY

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

4423
4424
4425
4426
4427
4428
4429
4430
4431
4432
4433
4434
4435
4436
4437
4438
4439
4440
4441
4442
4443
4444
4445
4446
4447
4448
4449
4450
4451
4452
4453
4454
4455
4456
4457
4458
4459
4460
4461
4462
4463
4464
4465
4466
4467
4468
4469
4470
4471
4472
4473
4474
4475
4476
4477
4478
4479
4480
4481
4482
4483
4484
4485
4486
4487
4488

```

4489     test=> SELECT city, state, COUNT(*)
4490     test-> FROM friend
4491     test-> GROUP BY state, city
4492     test-> ORDER BY 1, 2;
4493
4494         city      | state | count
4495     -----+-----+-----
4496     Boston        | MA    |     1
4497     Cedar Creek   | MD    |     1
4498     Ocean City    | NJ    |     2
4499     Plymouth      | MA    |     1
4500     Williamsport  | PA    |     1
4501
4502     (5 rows)
4503
4504
4505
4506
4507
4508
4509
4510

```

Figure 5.4: GROUP BY on two columns

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

```

4511
4512
4513
4514     test=> SELECT state, COUNT(*)
4515     test-> FROM friend
4516     test-> GROUP BY state
4517     test-> HAVING COUNT(*) > 1
4518     test-> ORDER BY state;
4519
4520         state | count
4521     -----+-----
4522     MA      |     2
4523     NJ      |     2
4524
4525     (2 rows)
4526
4527
4528
4529
4530
4531
4532
4533
4534
4535
4536
4537
4538
4539
4540
4541
4542
4543
4544
4545
4546
4547
4548
4549
4550
4551
4552
4553
4554

```

Figure 5.5: HAVING usage

HAVING.

5.4 Query Tips

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

In a test database, 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.

4555
4556
4557
4558
4559
4560
4561
4562
4563
4564
4565
4566
4567
4568
4569
4570
4571
4572
4573
4574
4575
4576
4577
4578
4579
4580
4581
4582
4583
4584
4585
4586
4587
4588
4589
4590
4591
4592
4593
4594
4595
4596
4597
4598
4599
4600
4601
4602
4603
4604
4605
4606
4607
4608
4609
4610
4611
4612
4613
4614
4615
4616
4617
4618
4619
4620

4621
4622
4623
4624
4625
4626
4627
4628
4629
4630
4631
4632
4633
4634
4635
4636
4637
4638
4639
4640
4641
4642
4643
4644
4645
4646
4647
4648
4649
4650
4651
4652
4653
4654
4655
4656
4657
4658
4659
4660
4661
4662
4663
4664
4665
4666
4667
4668
4669
4670
4671
4672
4673
4674
4675
4676
4677
4678
4679
4680
4681
4682
4683
4684
4685
4686

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, *f* is used as an alias for the *friend* table. While these features are not important in single table queries, they are useful in multi-table queries.

6.2 Joined Tables

In our *friend* example, splitting data into multiple tables makes little sense. However, in cases where we must record information about a variety of things, multiple tables have benefits. Consider a company that sells parts to customers through the mail. The database has to record information about many things: customers, employees, sales orders, and parts. It is obvious a single table cannot hold the different types of information in an organized manner. Therefore, we create four tables: *customer*, *employee*, *salesorder*, and *part*. However, putting information in different tables causes problems. How do we record which sales orders belong to which customers? How do we record the parts for the sales orders? How do we record which employee received the sales order? The answer is to assign unique numbers to every customer, employee, and part. When we want to record the customer in the *salesorder* table, we put the customer's number in the *salesorder* table. When we want to record which employee took the order, we put the employee's number in the *salesorder* table. When we want to record which part has been ordered, we put the part number in the *salesorder* table.

```

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

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

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

```

4687
4688
4689
4690
4691
4692
4693
4694
4695
4696
4697
4698
4699
4700
4701
4702
4703
4704
4705
4706
4707
4708
4709
4710
4711
4712
4713
4714
4715
4716
4717
4718
4719
4720
4721
4722
4723
4724
4725
4726
4727
4728
4729
4730
4731
4732
4733
4734
4735
4736
4737
4738
4739
4740
4741
4742
4743
4744
4745
4746
4747
4748
4749
4750
4751
4752

Figure 6.1: Qualified column names

Breaking up the information into separate tables allows us to keep detailed information about customers, employees, and parts. It also allows us to refer to those specific entries as many times as needed by using a unique number. This is illustrated in figure 6.2.

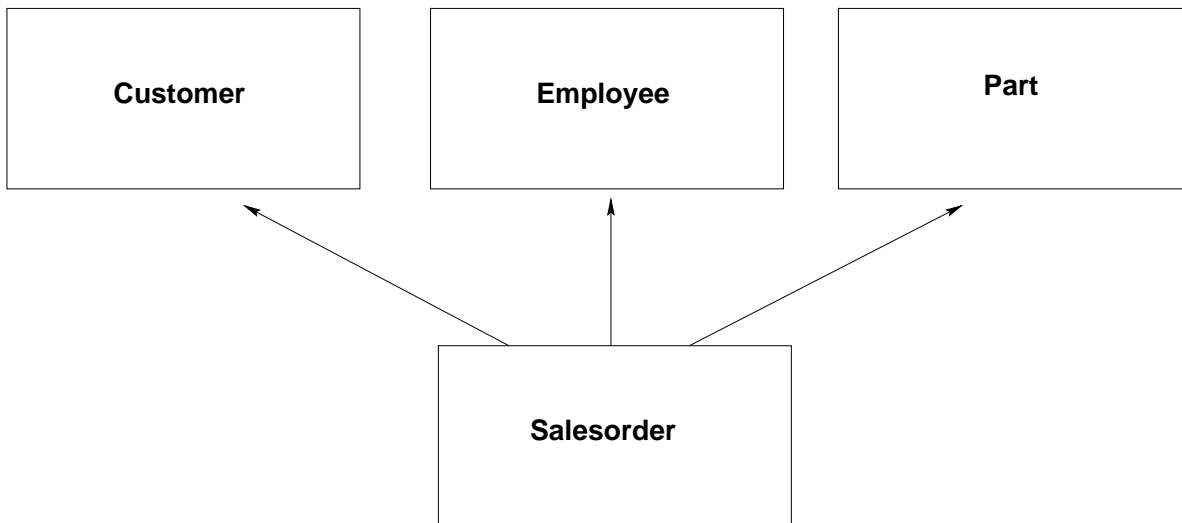


Figure 6.2: Joining tables

People might question 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 *salesorder*³ 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 `customerid`, which is not very clear. The only way to define non-lowercase column and table names is 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.

¹The process of distributing data across multiple tables to prevent redundancy is called *data normalization*.

²In the real-world, the *name* columns would be much longer, perhaps *char(60)* or *char(180)*. You should base the length on the longest name you may ever wish to store. I am using short *names* so they display properly in the examples.

³A table can not be called *order*. *Order* is a reserved keyword, for use in the ORDER BY clause. Reserved keywords are not available as table or column names.

```

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

```

Figure 6.3: Creation of company tables

4819
4820
4821
4822
4823
4824
4825
4826
4827
4828
4829
4830
4831
4832
4833
4834
4835
4836
4837
4838
4839
4840
4841
4842
4843
4844
4845
4846
4847
4848
4849
4850
4851
4852
4853
4854
4855
4856
4857
4858
4859
4860
4861
4862
4863
4864
4865
4866
4867
4868
4869
4870
4871
4872
4873
4874
4875
4876
4877
4878
4879
4880
4881
4882
4883
4884

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.

```

4885
4886
4887
4888
4889
4890     test=> INSERT INTO customer VALUES (
4891     test(>             648,
4892     test(>             'Fleer Gearworks, Inc.',
4893     test(>             '1-610-555-782',
4894     test(>             '830 Winding Way',
4895     test(>             'Millersville',
4896     test(>             'AL',
4897     test(>             '35041',
4898     test(>             'USA'
4899
4900
4901     test(> );
4902
4903     INSERT 19815 1
4904
4905     test=> INSERT INTO employee VALUES (
4906     test(>             24,
4907     test(>             'Lee Meyers',
4908     test(>             '10/16/1989'
4909
4910     test(> );
4911
4912     INSERT 19816 1
4913
4914     test=> INSERT INTO part VALUES (
4915     test(>             153,
4916     test(>             'Garage Door Spring',
4917     test(>             18.39
4918
4919     test(> );
4920
4921     INSERT 19817 1
4922
4923     test=> INSERT INTO salesorder VALUES(
4924     test(>             14673,
4925     test(>             648,
4926     test(>             24,
4927     test(>             153,
4928     test(>             '7/19/1994',
4929     test(>             '7/28/1994',
4930     test(>             18.39
4931
4932     test(> );
4933
4934     INSERT 18841 1
4935
4936
4937
4938
4939
4940
4941
4942
4943
4944
4945
4946
4947
4948
4949
4950

```

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

⁴Technically, the column *customer.customer_id* is a *primary key* because it is the unique key for each customer row. The column *salesorder.customer_id* is a *foreign key* because it points to another table's primary key. This is covered in more detail in section 6.13.

```

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

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

```

Figure 6.5: Finding 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

```

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

```

Figure 6.6: Finding customer name using one query

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.

4951
4952
4953
4954
4955
4956
4957
4958
4959
4960
4961
4962
4963
4964
4965
4966
4967
4968
4969
4970
4971
4972
4973
4974
4975
4976
4977
4978
4979
4980
4981
4982
4983
4984
4985
4986
4987
4988
4989
4990
4991
4992
4993
4994
4995
4996
4997
4998
4999
5000
5001
5002
5003
5004
5005
5006
5007
5008
5009
5010
5011
5012
5013
5014
5015
5016

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 *customer_id* exists in both tables mentioned in the FROM clause, *customer* and *salesorder*. If this were not done, the query would generate an error: 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 FROM clause and in the WHERE clause. The

```

5017
5018 test=> SELECT salesorder.order_id
5019 test-> FROM   salesorder, customer
5020
5021 test-> WHERE  customer.name = 'Fleer Gearworks, Inc.' AND
5022 test->        salesorder.customer_id = customer.customer_id;
5023
5024
5025
5026
5027
5028
5029
5030
5031
5032
5033
5034
5035
5036
5037
5038
5039
5040
5041
5042
5043
5044
5045
5046
5047
5048
5049
5050
5051
5052
5053
5054
5055
5056
5057
5058
5059
5060
5061
5062
5063
5064
5065
5066
5067
5068
5069
5070
5071
5072
5073
5074
5075
5076
5077
5078
5079
5080
5081
5082

```

order_id
14673

(1 row)

Figure 6.7: Finding order number for customer name

ordering of items is not important in these clauses.

6.5 Three and Four Table Joins

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

```

5052 test=> SELECT customer.name, employee.name
5053 test-> FROM   salesorder, customer, employee
5054 test-> WHERE  salesorder.customer_id = customer.customer_id AND
5055 test->        salesorder.employee_id = employee.employee_id AND
5056 test->        salesorder.order_id = 14673;
5057
5058
5059
5060
5061
5062
5063
5064
5065
5066
5067
5068
5069
5070
5071
5072
5073
5074
5075
5076
5077
5078
5079
5080
5081
5082

```

name	name
Fleer Gearworks, Inc.	Lee Meyers

(1 row)

Figure 6.8: Three-table join

The second column is the employee name. Both columns are labeled *name*. You could use AS to give the columns unique labels. Figure 6.9 shows a four-table join, using AS to make each column label unique. The four-table join matches the arrows in figure 6.2, with the arrows of the *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 *customer* and *employee* tables. The *salesorder* table is used to join the two tables but is not displayed. The DISTINCT keyword is used because multiple orders taken by the same employee for the same customer would make that employee appear more than once, which was not desired. The second query uses an aggregate to return a count for each unique customer, employee pair.

```

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

```

Figure 6.9: Four-table join

```

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

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

```

Figure 6.10: Employees who have taken orders for customers.

5083
5084
5085
5086
5087
5088
5089
5090
5091
5092
5093
5094
5095
5096
5097
5098
5099
5100
5101
5102
5103
5104
5105
5106
5107
5108
5109
5110
5111
5112
5113
5114
5115
5116
5117
5118
5119
5120
5121
5122
5123
5124
5125
5126
5127
5128
5129
5130
5131
5132
5133
5134
5135
5136
5137
5138
5139
5140
5141
5142
5143
5144
5145
5146
5147
5148

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

```

5149
5150
5151
5152
5153
5154
5155
5156
5157
5158
5159
5160
5161
5162
5163
5164
5165
5166
5167
5168
5169
5170
5171
5172
5173
5174
5175
5176
5177
5178
5179
5180
5181
5182
5183
5184
5185
5186
5187
5188
5189
5190
5191
5192
5193
5194
5195
5196
5197
5198
5199
5200
5201
5202
5203
5204
5205
5206
5207
5208
5209
5210
5211
5212
5213
5214

```

```

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

test=> SELECT customer.name
test-> FROM   customer, employee
test-> WHERE  customer.employee_id = employee.employee_id AND
test->        employee.employee_id = 24
test-> ORDER BY customer.name;

```

Figure 6.11: Joining customer and employee

employee name assigned to manage customer number 648. The second query shows the customer names managed by employee 24. Notice the *salesorder* table is not involved in this query.

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

```

5192
5193
5194
5195
5196
5197
5198
5199
5200
5201
5202
5203
5204
5205
5206
5207
5208
5209
5210
5211
5212
5213
5214

```

```

test=> -- find the employee assigned to part number 14673
test=> SELECT employee.name
test-> FROM   part, employee
test-> WHERE  part.employee_id = employee.employee_id AND
test->        part.part_id = 153;

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

```

Figure 6.12: Joining part and employee

There are cases where a join could be performed with the *state* column. For example, to check state

codes for validity⁵, 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

```

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

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

```

Figure 6.13: 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.

⁵The United States Postal Service has assigned a unique two-letter code to each U.S. state.

5215
5216
5217
5218
5219
5220
5221
5222
5223
5224
5225
5226
5227
5228
5229
5230
5231
5232
5233
5234
5235
5236
5237
5238
5239
5240
5241
5242
5243
5244
5245
5246
5247
5248
5249
5250
5251
5252
5253
5254
5255
5256
5257
5258
5259
5260
5261
5262
5263
5264
5265
5266
5267
5268
5269
5270
5271
5272
5273
5274
5275
5276
5277
5278
5279
5280

- 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, FLE001, employees by their initials, BAW, and parts by five-character codes, E7245. Codes are easy to use and remember. In many cases, users can choose the codes, as long as they are unique.

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

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

Figure 6.14: Using a customer code

In some cases, identification numbers are fine and codes unnecessary:

- Items with short lifespans, 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.⁶

```

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

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

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

```

Figure 6.15: One-to-many join

5347
5348
5349
5350
5351
5352
5353
5354
5355
5356
5357
5358
5359
5360
5361
5362
5363
5364
5365
5366
5367
5368
5369
5370
5371
5372
5373
5374
5375
5376
5377
5378
5379
5380
5381
5382
5383
5384
5385
5386
5387
5388
5389
5390
5391
5392
5393
5394
5395
5396
5397
5398
5399
5400
5401
5402
5403
5404
5405
5406
5407
5408
5409
5410
5411
5412

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

6.9 Unjoined Tables

When joining tables, it is necessary to join each table mentioned in the FROM clause by specifying joins in the WHERE clause. If you list a table name in the FROM clause, but fail to join it in the WHERE clause, the effect is to mark that table as unjoined. This causes it to be paired with every row in the query result. Figure 6.16 illustrates this effect using tables from figure 6.15. The SELECT does not join any column from *animal* to any

```

5429 test=> SELECT *
5430 test-> FROM animal, vegetable;
5431
5432 animal_id | name | animal_id | name
5433 -----+-----+-----+-----
5434          507 | rabbit |          507 | lettuce
5435          508 | cat   |          507 | lettuce
5436          507 | rabbit |          507 | carrot
5437          508 | cat   |          507 | carrot
5438          507 | rabbit |          507 | nut
5439          508 | cat   |          507 | nut
5440
5441 (6 rows)

```

Figure 6.16: Unjoined tables

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*

```

5461 test=> SELECT order_id
5462 test-> FROM customer c, salesorder s
5463 test-> WHERE c.code = 'FLE001' AND
5464 test-> c.customer_id = s.customer_id;

```

Figure 6.17: Using table aliases

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.

⁶Many 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.

```

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

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

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

```

Figure 6.18: Examples of self-joins using table aliases

The first 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.

5479
5480
5481
5482
5483
5484
5485
5486
5487
5488
5489
5490
5491
5492
5493
5494
5495
5496
5497
5498
5499
5500
5501
5502
5503
5504
5505
5506
5507
5508
5509
5510
5511
5512
5513
5514
5515
5516
5517
5518
5519
5520
5521
5522
5523
5524
5525
5526
5527
5528
5529
5530
5531
5532
5533
5534
5535
5536
5537
5538
5539
5540
5541
5542
5543
5544

```

5545      test=> SELECT c2.name
5546      test-> FROM   customer c, customer c2
5547      test-> WHERE  c.customer_id = 648 AND
5548      test->         c.country <> c2.country
5549      test->
5550      test-> ORDER BY c2.name;
5551
5552      test=> SELECT e2.name, e2.hire_date
5553      test-> FROM   employee e, employee e2
5554      test-> WHERE  e.employee_id = 24 AND
5555      test->         e.hire_date < e2.hire_date
5556      test->
5557      test-> ORDER BY e2.hire_date, e2.name;
5558
5559
5560      test=> SELECT p2.name, p2.cost
5561      test-> FROM   part p, part p2
5562      test-> WHERE  p.part_id = 153 AND
5563      test->         p.cost > p2.cost
5564      test->
5565      test-> ORDER BY p2.cost;
5566
5567
5568
5569
5570
5571
5572

```

Figure 6.19: Non-equi joins

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.

```

5573
5574
5575
5576      test=> CREATE TABLE salesorder (
5577
5578      test(>         order_id      INTEGER,
5579      test(>         customer_id   INTEGER, -- joins to customer.customer_id
5580      test(>         employee_id    INTEGER, -- joins to employee.employee_id
5581      test(>         order_date     DATE,
5582      test(>         ship_date      DATE,
5583      test(>         payment        NUMERIC(8,2)
5584      test(> );
5585
5586      CREATE
5587
5588
5589
5590

```

Figure 6.20: New salesorder table for multiple parts per order

Figure 6.21 shows a new table, *orderpart*. This table is needed because the original *salesorder* table could

```

5591
5592
5593
5594      test=> CREATE TABLE orderpart(
5595
5596      test(>         order_id INTEGER,
5597      test(>         part_id  INTEGER,
5598      test(>         quantity INTEGER DEFAULT 1
5599      test(> );
5600
5601      CREATE
5602
5603
5604
5605
5606
5607
5608
5609
5610

```

Figure 6.21: Orderpart table

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 `SUM cost times (*) 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.

⁷In 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.

5611
5612
5613
5614
5615
5616
5617
5618
5619
5620
5621
5622
5623
5624
5625
5626
5627
5628
5629
5630
5631
5632
5633
5634
5635
5636
5637
5638
5639
5640
5641
5642
5643
5644
5645
5646
5647
5648
5649
5650
5651
5652
5653
5654
5655
5656
5657
5658
5659
5660
5661
5662
5663
5664
5665
5666
5667
5668
5669
5670
5671
5672
5673
5674
5675
5676

```
5677
5678
5679
5680
5681
5682
5683
5684
5685
5686
5687      test=> -- first query
5688      test=> SELECT part.name
5689      test-> FROM   orderpart, part
5690      test-> WHERE  orderpart.part_id = part.part_id AND
5691      test->         orderpart.order_id = 15398;
5692
5693
5694      test=> -- second query
5695      test=> SELECT part.name, orderpart.quantity
5696      test-> FROM salesorder, orderpart, part
5697      test-> WHERE salesorder.customer_id = 648 AND
5698      test->         salesorder.order_date = '7/19/1994' AND
5699      test->         salesorder.order_id = orderpart.order_id AND
5700      test->         orderpart.part_id = part.part_id;
5701
5702
5703
5704
5705      test=> -- third query
5706      test=> SELECT part.name, part.cost, orderpart.quantity
5707      test-> FROM   customer, salesorder, orderpart, part
5708      test-> WHERE  customer.name = 'Fleer Gearworks, Inc.' AND
5709      test->         salesorder.order_date = '7/19/1994' AND
5710      test->         salesorder.customer_id = customer.customer_id AND
5711      test->         salesorder.order_id = orderpart.order_id AND
5712      test->         orderpart.part_id = part.part_id;
5713
5714
5715
5716
5717      test=> -- fourth query
5718      test=> SELECT SUM(part.cost * orderpart.quantity)
5719      test-> FROM   customer, salesorder, orderpart, part
5720      test-> WHERE  customer.name = 'Fleer Gearworks, Inc.' AND
5721      test->         salesorder.order_date = '7/19/1994' AND
5722      test->         salesorder.customer_id = customer.customer_id AND
5723      test->         salesorder.order_id = orderpart.order_id AND
5724      test->         orderpart.part_id = part.part_id;
5725
5726
5727
5728
5729
5730
5731
5732
5733
5734
5735
5736
5737
5738
5739
5740
5741
5742
```

Figure 6.22: Queries involving *orderpart* table

5743
5744
5745
5746
5747
5748
5749
5750
5751
5752
5753
5754
5755
5756
5757
5758
5759
5760
5761
5762
5763
5764
5764
5765
5766
5767
5768
5769
5770
5771
5772
5773
5774
5775
5776
5777
5778
5779
5780
5781
5782
5783
5784
5785
5786
5787
5788
5789
5790
5791
5792
5793
5794
5795
5796
5797
5798
5799
5800
5801
5802
5803
5804
5805
5806
5807
5808

5809
5810
5811
5812
5813
5814
5815
5816
5817
5818
5819
5820
5821
5822
5823
5824
5825
5826
5827
5828
5829
5830
5831
5832
5833
5834
5835
5836
5837
5838
5839
5840
5841
5842
5843
5844
5845
5846
5847
5848
5849
5850
5851
5852
5853
5854
5855
5856
5857
5858
5859
5860
5861
5862
5863
5864
5865
5866
5867
5868
5869
5870
5871
5872
5873
5874

Chapter 7

Numbering Rows

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

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

7.1 Object Identification Numbers (OIDs)

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

You have seen object identification numbers already. Object identification numbers are displayed after every INSERT statement. If you look back at figure 3.4 on page 12, you will see the line `INSERT 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.⁴ The query `SELECT * FROM table_name` does not display the OID column. `SELECT OID, * FROM table_name` will display it.

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

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

¹See section B for a description of `initdb`.
²Values less than this are reserved for internal use.
³Technically, OIDs 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;
  oid | age
-----+-----
 21515 |   7
(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 (
test(>           order_id      INTEGER,
test(>           customer_oid  OID, -- joins to customer.oid
test(>           employee_oid  OID, -- joins to employee.oid
test(>           part_oid      OID, -- joins to part.oid
...

```

Figure 7.2: Columns with OIDs

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

5875
5876
5877
5878
5879
5880
5881
5882
5883
5884
5885
5886
5887
5888
5889
5890
5891
5892
5893
5894
5895
5896
5897
5898
5899
5900
5901
5902
5903
5904
5905
5906
5907
5908
5909
5910
5911
5912
5913
5914
5915
5916
5917
5918
5919
5920
5921
5922
5923
5924
5925
5926
5927
5928
5929
5930
5931
5932
5933
5934
5935
5936
5937
5938
5939
5940

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.

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

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

⁵This 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
test=> SELECT nextval('functest_seq');
nextval
-----
         1
(1 row)

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

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

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

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

```

Figure 7.3: Examples of sequence function use

6007
6008
6009
6010
6011
6012
6013
6014
6015
6016
6017
6018
6019
6020
6021
6022
6023
6024
6025
6026
6027
6028
6029
6030
6031
6032
6033
6034
6035
6036
6037
6038
6039
6040
6041
6042
6043
6044
6045
6046
6047
6048
6049
6050
6051
6052
6053
6054
6055
6056
6057
6058
6059
6060
6061
6062
6063
6064
6065
6066
6067
6068
6069
6070
6071
6072

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

7.5 Using Sequences to Number Rows

Configuring a sequence to uniquely number rows involves several steps:

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

Figure 7.4 shows the use of a sequence for unique row numbering in the customer table. The first state-

```

6097 test=> CREATE SEQUENCE customer_seq;
6098 CREATE
6099 test=> CREATE TABLE customer (
6100 test(>         customer_id INTEGER DEFAULT nextval('customer_seq'),
6101 test(>         name CHAR(30)
6102 test(> );
6103 CREATE
6104 test=> INSERT INTO customer VALUES (nextval('customer_seq'), 'Bread Makers');
6105 INSERT 19004 1
6106 test=> INSERT INTO customer (name) VALUES ('Wax Carvers');
6107 INSERT 19005 1
6108 test=> INSERT INTO customer (name) VALUES ('Pipe Fitters');
6109 INSERT 19008 1
6110 test=> SELECT * FROM customer;
6111 customer_id | name
6112 -----+-----
6113          1 | Bread Makers
6114          2 | Wax Carvers
6115          3 | Pipe Fitters
6116 (3 rows)

```

Figure 7.4: Numbering *customer* rows using a sequence

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

7.6 Serial Column Type

There is an easier way to use sequences. If you define a column of type SERIAL, a sequence will be automatically created, and a proper DEFAULT assigned to the column. Figure 7.5 shows an example of this. The first NOTICE line indicates a sequence was created for the SERIAL column. Do not be concerned about

```

test=> CREATE TABLE customer (
test(>         customer_id SERIAL,
test(>         name CHAR(30)
test(> );
NOTICE: CREATE TABLE will create implicit sequence 'customer_customer_id_
seq' for SERIAL column 'customer.customer_id'
NOTICE: CREATE TABLE/UNIQUE will create implicit index 'customer_customer_id_
key' for table 'customer'
CREATE
test=> \d customer

```

Table "customer"		
Attribute	Type	Extra
customer_id	int4	not null default nextval('customer_customer_id_seq'::text)
name	char(30)	

```

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

```

Figure 7.5: *Customer* table using SERIAL

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

6139
6140
6141
6142
6143
6144
6145
6146
6147
6148
6149
6150
6151
6152
6153
6154
6155
6156
6157
6158
6159
6160
6161
6162
6163
6164
6165
6166
6167
6168
6169
6170
6171
6172
6173
6174
6175
6176
6177
6178
6179
6180
6181
6182
6183
6184
6185
6186
6187
6188
6189
6190
6191
6192
6193
6194
6195
6196
6197
6198
6199
6200
6201
6202
6203
6204

6205
6206
6207
6208
6209
6210
6211
6212
6213
6214
6215
6216
6217
6218
6219
6220
6221
6222
6223
6224
6225
6226
6227
6228
6229
6230
6231
6232
6233
6234
6235
6236
6237
6238
6239
6240
6241
6242
6243
6244
6245
6246
6247
6248
6249
6250
6251
6252
6253
6254
6255
6256
6257
6258
6259
6260
6261
6262
6263
6264
6265
6266
6267
6268
6269
6270

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.

6271
6272
6273
6274
6275
6276
6277
6278
6279
6280
6281
6282
6283
6284
6285
6286
6287
6288
6289
6290
6291
6292
6293
6294
6295
6296
6297
6298
6299
6300
6301
6302
6303
6304
6305
6306
6307
6308
6309
6310
6311
6312
6313
6314
6315
6316
6317
6318
6319
6320
6321
6322
6323
6324
6325
6326
6327
6328
6329
6330
6331
6332
6333
6334
6335
6336

6337
6338
6339
6340
6341
6342
6343
6344
6345
6346
6347
6348
6349
6350
6351
6352
6353
6354
6355
6356
6357
6358
6359
6360
6361
6362
6363
6364
6365
6366
6367
6368
6369
6370
6371
6372
6373
6374
6375
6376
6377
6378
6379
6380
6381
6382
6383
6384
6385
6386
6387
6388
6389
6390
6391
6392
6393
6394
6395
6396
6397
6398
6399
6400
6401
6402

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

```
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

single output column.

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

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

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

However, suppose we need to list all the animals, both *aquatic* and *terrestrial*. There is no single SELECT that will show animals from both tables. We 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
test=> INSERT INTO aquatic_animal (name) VALUES ('swordfish');
INSERT 19123 1
test=> SELECT name
test-> FROM   aquatic_animal
test-> UNION
test-> SELECT name
test-> FROM   terrestrial_animal;
           name
-----
swordfish
tiger
(2 rows)

```

Figure 8.2: Combining two tables with UNION

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

6403
6404
6405
6406
6407
6408
6409
6410
6411
6412
6413
6414
6415
6416
6417
6418
6419
6420
6421
6422
6423
6424
6425
6426
6427
6428
6429
6430
6431
6432
6433
6434
6435
6436
6437
6438
6439
6440
6441
6442
6443
6444
6445
6446
6447
6448
6449
6450
6451
6452
6453
6454
6455
6456
6457
6458
6459
6460
6461
6462
6463
6464
6465
6466
6467
6468

```

6469      test=> INSERT INTO aquatic_animal (name) VALUES ('penguin');
6470      INSERT 19124 1
6471      test=> INSERT INTO terrestrial_animal (name) VALUES ('penguin');
6472      INSERT 19125 1
6473      test=> SELECT name
6474      test-> FROM   aquatic_animal
6475      test-> UNION
6476      test-> SELECT name
6477      test-> FROM   terrestrial_animal;
6478      name
6479      -----
6480      penguin
6481      swordfish
6482      tiger
6483      (3 rows)

```

Figure 8.3: UNION with duplicates

```

6495      test=> SELECT name
6496      test-> FROM   aquatic_animal
6497      test-> UNION ALL
6498      test-> SELECT name
6499      test-> FROM   terrestrial_animal;
6500      name
6501      -----
6502      swordfish
6503      penguin
6504      tiger
6505      penguin
6506      (4 rows)

```

Figure 8.4: UNION ALL with duplicates

```

6517      test=> SELECT name
6518      test-> FROM   aquatic_animal
6519      test-> EXCEPT
6520      test-> SELECT name
6521      test-> FROM   terrestrial_animal;
6522      name
6523      -----
6524      swordfish
6525      (1 row)

```

Figure 8.5: EXCEPT restricts output from the first SELECT

6534

```

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

```

Figure 8.6: INTERSECT returns only duplicated rows

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*

6535
6536
6537
6538
6539
6540
6541
6542
6543
6544
6545
6546
6547
6548
6549
6550
6551
6552
6553
6554
6555
6556
6557
6558
6559
6560
6561
6562
6563
6564
6565
6566
6567
6568
6569
6570
6571
6572
6573
6574
6575
6576
6577
6578
6579
6580
6581
6582
6583
6584
6585
6586
6587
6588
6589
6590
6591
6592
6593
6594
6595
6596
6597
6598
6599
6600

```

6601
6602
6603 test=> SELECT * FROM friend ORDER BY firstname;
6604
6605     firstname |      lastname      |      city      | state | age
6606 -----+-----+-----+-----+-----
6607     Dean      |      Yeager       |    Plymouth   |    MA |   24
6608     Dick      |      Gleason      |    Ocean City |    NJ |   19
6609     Ned       |    Millstone     |    Cedar Creek |    MD |   27
6610     Sandy     |      Gleason      |    Ocean City |    NJ |   25
6611     Sandy     |      Weber        |      Boston   |    MA |   33
6612     Victor   |      Tabor        | Williamsport  |    PA |   22
6613
6614 (6 rows)
6615
6616
6617
6618 test=> SELECT f1.firstname, f1.lastname, f1.state
6619 test-> FROM   friend f1, friend f2
6620 test-> WHERE  f1.state <> f2.state AND
6621 test->        f2.firstname = 'Dick' AND
6622 test->        f2.lastname  = 'Gleason'
6623 test-> ORDER BY  firstname, lastname;
6624
6625     firstname |      lastname      | state
6626 -----+-----+-----
6627     Dean      |      Yeager       |    MA
6628     Ned       |    Millstone     |    MD
6629     Sandy     |      Weber        |    MA
6630     Victor   |      Tabor        |    PA
6631
6632 (4 rows)
6633
6634
6635
6636
6637
6638 test=> SELECT f1.firstname, f1.lastname, f1.state
6639 test-> FROM   friend f1
6640 test-> WHERE  f1.state <> (
6641 test(>      SELECT f2.state
6642 test(>      FROM   friend f2
6643 test(>      WHERE  f2.firstname = 'Dick' AND
6644 test(>      WHERE  f2.lastname  = 'Gleason'
6645 test(>      )
6646 test(>      )
6647 test-> ORDER BY  firstname, lastname;
6648
6649     firstname |      lastname      | state
6650 -----+-----+-----
6651     Dean      |      Yeager       |    MA
6652     Ned       |    Millstone     |    MD
6653     Sandy     |      Weber        |    MA
6654     Victor   |      Tabor        |    PA
6655
6656 (4 rows)
6657
6658
6659
6660
6661
6662
6663
6664
6665
6666

```

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

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

```

Figure 8.8: Subqueries can replace some joins

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

6667
6668
6669
6670
6671
6672
6673
6674
6675
6676
6677
6678
6679
6680
6681
6682
6683
6684
6685
6686
6687
6688
6689
6690
6691
6692
6693
6694
6695
6696
6697
6698
6699
6700
6701
6702
6703
6704
6705
6706
6707
6708
6709
6710
6711
6712
6713
6714
6715
6716
6717
6718
6719
6720
6721
6722
6723
6724
6725
6726
6727
6728
6729
6730
6731
6732

```

6733
6734
6735
6736
6737
6738
6739
6740
6741
6742
6743
6744 test=> SELECT f1.firstname, f1.lastname, f1.age
6745 test-> FROM   friend f1, friend f2
6746 test-> WHERE  f1.state = f2.state
6747
6748 test-> GROUP BY f2.state, f1.firstname, f1.lastname, f1.age
6749 test-> HAVING f1.age = max(f2.age)
6750 test-> ORDER BY firstname, lastname;
6751
6752      firstname |      lastname      | age
6753 -----+-----+-----
6754 Ned           | Millstone          | 27
6755 Sandy        | Gleason            | 25
6756 Sandy        | Weber              | 33
6757 Victor       | Tabor              | 22
6758 (4 rows)
6759
6760
6761
6762 test=> SELECT f1.firstname, f1.lastname, f1.age
6763 test-> FROM   friend f1
6764 test-> WHERE  age = (
6765 test(>      SELECT MAX(f2.age)
6766 test(>      FROM friend f2
6767 test(>      WHERE f1.state = f2.state
6768 test(>      )
6769 test-> ORDER BY firstname, lastname;
6770
6771      firstname |      lastname      | age
6772 -----+-----+-----
6773 Ned           | Millstone          | 27
6774 Sandy        | Gleason            | 25
6775 Sandy        | Weber              | 33
6776 Victor       | Tabor              | 22
6777 (4 rows)
6778
6779
6780
6781
6782
6783
6784
6785
6786
6787
6788
6789
6790
6791
6792
6793
6794
6795
6796
6797
6798

```

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

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

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

```

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

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

```

Figure 8.10: Employees who took orders

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

A `NOT IN` comparison returns true if a column's value is not found. For example, suppose we want to see all customers who have never ordered a product. We need to find the *customers* who have no sales orders. This cannot be done with a join. We need an *anti-join*, because we want to find all *customer* rows that do

6799
6800
6801
6802
6803
6804
6805
6806
6807
6808
6809
6810
6811
6812
6813
6814
6815
6816
6817
6818
6819
6820
6821
6822
6823
6824
6825
6826
6827
6828
6829
6830
6831
6832
6833
6834
6835
6836
6837
6838
6839
6840
6841
6842
6843
6844
6845
6846
6847
6848
6849
6850
6851
6852
6853
6854
6855
6856
6857
6858
6859
6860
6861
6862
6863
6864

not join to any *salesorder* row. Figure 8.11 shows the query. The subquery returns a list of *customer_ids*

6865
6866
6867
6868
6869
6870
6871
6872
6873
6874
6875
6876
6877
6878
6879
6880
6881
6882
6883
6884
6885
6886
6887
6888
6889
6890
6891
6892
6893
6894
6895
6896
6897
6898
6899
6900
6901
6902
6903
6904
6905
6906
6907
6908
6909
6910
6911
6912
6913
6914
6915
6916
6917
6918
6919
6920
6921
6922
6923
6924
6925
6926
6927
6928
6929
6930

```
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, even inequality comparisons, so NOT IN returns false. NULL comparisons are covered in section 4.3.

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

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

There is another way to analyze subqueries returning NULLs. Suppose a subquery returns three rows, 1, 2, and NULL. The test `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

```

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

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

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

```

Figure 8.12: IN query rewritten using ANY and EXISTS

tables. Figure 8.13 shows the NOT IN query from figure 8.11 and the query rewritten using ALL and NOT EXISTS.

6931
6932
6933
6934
6935
6936
6937
6938
6939
6940
6941
6942
6943
6944
6945
6946
6947
6948
6949
6950
6951
6952
6953
6954
6955
6956
6957
6958
6959
6960
6961
6962
6963
6964
6965
6966
6967
6968
6969
6970
6971
6972
6973
6974
6975
6976
6977
6978
6979
6980
6981
6982
6983
6984
6985
6986
6987
6988
6989
6990
6991
6992
6993
6994
6995
6996

```

6997     SELECT name
6998     FROM   customer
6999     WHERE  customer_id NOT IN (
7000         SELECT customer_id
7001         FROM   salesorder
7002         );
7003
7004
7005     SELECT name
7006     FROM   customer
7007     WHERE  customer_id <> ALL (
7008         SELECT customer_id
7009         FROM   salesorder
7010         );
7011
7012
7013
7014
7015     SELECT name
7016     FROM   customer
7017     WHERE NOT EXISTS (
7018         SELECT customer_id
7019         FROM   salesorder
7020         WHERE  salesorder.customer_id = customer.customer_id
7021         );
7022
7023
7024
7025
7026
7027
7028
7029
7030
7031
7032
7033
7034
7035
7036
7037
7038
7039
7040
7041
7042
7043
7044
7045
7046
7047
7048
7049
7050
7051
7052
7053
7054
7055
7056
7057
7058
7059
7060
7061
7062

```

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(>             SELECT customer_id
test(>             FROM salesorder
test(>             );
DELETE 0
test=> UPDATE salesorder
test-> SET   ship_date = '11/16/96'
test-> WHERE customer_id = (
test(>             SELECT customer_id
test(>             FROM   customer
test(>             WHERE  name = 'Fleer Gearworks, Inc.'
test(>             );
UPDATE 1

```

Figure 8.15: Subqueries with UPDATE and DELETE

7063
7064
7065
7066
7067
7068
7069
7070
7071
7072
7073
7074
7075
7076
7077
7078
7079
7080
7081
7082
7083
7084
7085
7086
7087
7088
7089
7090
7091
7092
7093
7094
7095
7096
7097
7098
7099
7100
7101
7102
7103
7104
7105
7106
7107
7108
7109
7110
7111
7112
7113
7114
7115
7116
7117
7118
7119
7120
7121
7122
7123
7124
7125
7126
7127
7128

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.

```

7145      test=> UPDATE salesorder
7146      test-> SET   order_date = employee.hire_date
7147
7148      test-> FROM   employee
7149
7150      test-> WHERE  salesorder.employee_id = employee.employee_id AND
7151      test->        salesorder.order_date < employee.hire_date;
7152      UPDATE 0

```

Figure 8.16: UPDATE the *order_date*

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

8.6 Inserting Data Using SELECT

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

Suppose we wish to add all 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

```

7175      test=> INSERT INTO customer (name, city, state, country)
7176      test-> SELECT trim(firstname) || ' ' || lastname, city, state, 'USA'
7177      test-> FROM friend;
7178
7179      INSERT 0 6

```

Figure 8.17: Using SELECT with INSERT

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

Inserting into the customer name column presents an interesting challenge. The *friend* table stores first and last names in separate columns. The *customer* table has a single *name* column. The only solution is to combine the *firstname* and *lastname* columns, with a space between them. For example, a *firstname* of 'Dean' and *lastname* of 'Yeager' must be inserted into *customer.name* as 'Dean Yeager'. This is possible using *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`

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

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

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

Figure 8.18: Table creation with SELECT

query:

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

`SELECT...INTO` is `CREATE TABLE` and `SELECT` combined in a single statement. The `AS` clause can be used to change the column labels and thus control the column names in the new table. The other 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.`

7195
7196
7197
7198
7199
7200
7201
7202
7203
7204
7205
7206
7207
7208
7209
7210
7211
7212
7213
7214
7215
7216
7217
7218
7219
7220
7221
7222
7223
7224
7225
7226
7227
7228
7229
7230
7231
7232
7233
7234
7235
7236
7237
7238
7239
7240
7241
7242
7243
7244
7245
7246
7247
7248
7249
7250
7251
7252
7253
7254
7255
7256
7257
7258
7259
7260

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.

7261
7262
7263
7264
7265
7266
7267
7268
7269
7270
7271
7272
7273
7274
7275
7276
7277
7278
7279
7280
7281
7282
7283
7284
7285
7286
7287
7288
7289
7290
7291
7292
7293
7294
7295
7296
7297
7298
7299
7300
7301
7302
7303
7304
7305
7306
7307
7308
7309
7310
7311
7312
7313
7314
7315
7316
7317
7318
7319
7320
7321
7322
7323
7324
7325
7326

7327
7328
7329
7330
7331
7332
7333
7334
7335
7336
7337
7338
7339
7340
7341
7342
7343
7344
7345
7346
7347
7348
7349
7350
7351
7352
7353
7354
7355
7356
7357
7358
7359
7360
7361
7362
7363
7364
7365
7366
7367
7368
7369
7370
7371
7372
7373
7374
7375
7376
7377
7378
7379
7380
7381
7382
7383
7384
7385
7386
7387
7388
7389
7390
7391
7392

7393
7394
7395
7396
7397
7398
7399
7400
7401
7402
7403
7404
7405
7406
7407
7408
7409
7410
7411
7412
7413
7414
7415
7416
7417
7418
7419
7420
7421
7422
7423
7424
7425
7426
7427
7428
7429
7430
7431
7432
7433
7434
7435
7436
7437
7438
7439
7440
7441
7442
7443
7444
7445
7446
7447
7448
7449
7450
7451
7452
7453
7454
7455
7456
7457
7458

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.

Category	Type	Description	
Character string	TEXT	variable storage length	7459
	VARCHAR(<i>length</i>)	variable storage length with maximum <i>length</i>	7460
	CHAR(<i>length</i>)	fixed storage length, blank-padded to <i>length</i> , internally BPCHAR	7461
Number	INTEGER	integer, ± 2 billion range, internally INT4	7462
	INT2	integer, ± 32 thousand range	7463
	INT8	integer, $\pm 4 \times 10^{18}$ range	7464
	OID	object identifier	7465
	NUMERIC(<i>precision</i> , <i>decimal</i>)	number, user-defined <i>precision</i> and <i>decimal</i> location	7466
	FLOAT	floating-point number, 15-digit precision, internally FLOAT8	7467
	FLOAT4	floating-point number, 6-digit precision	7468
Temporal	DATE	date	7469
	TIME	time	7470
	TIMESTAMP	date and time	7471
	INTERVAL	interval of time	7472
Logical	BOOL	boolean, <i>true</i> or <i>false</i>	7473
Geometric	POINT	point	7474
	LSEG	line segment	7475
	PATH	list of points	7476
	BOX	rectangle	7477
	CIRCLE	circle	7478
	POLYGON	polygon	7479
Network	INET	IP address with optional netmask	7480
	CIDR	IP network address	7481
	MACADDR	Ethernet MAC address	7482

Table 9.1: PostgreSQL data 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.¹ 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.

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.

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

Table 9.2: Geometric types

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.

```

7681      test=> SELECT * FROM functest;
7682      name
7683      -----
7684      Judy
7685      (1 row)
7686
7687      test=> SELECT upper(name) FROM functest;
7688      upper
7689      -----
7690      JUDY
7691      (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. *Psql'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. *Psql's \do* shows all defined operators and their arguments.

All data types have the standard comparison operators `<`, `<=`, `=`, `>=`, `>`, and `<>`. Not all operator/type combinations are defined. For example, if you try to add two `DATE` values, you will get an error, as shown in the first query of figure 9.3.

Type	Function	Example	Returns	
Character String	length()	length(<i>col</i>)	length of <i>col</i>	7723
	character_length()	character_length(<i>col</i>)	length of <i>col</i> , same as length()	7724
	octet_length()	octet_length(<i>col</i>)	length of <i>col</i> , including multi-byte overhead	7725
	trim()	trim(<i>col</i>)	<i>col</i> with leading and trailing spaces removed	7726
	trim(BOTH...)	trim(BOTH, <i>col</i>)	same as trim()	7727
	trim(LEADING...)	trim(LEADING <i>col</i>)	<i>col</i> with leading spaces removed	7728
	trim(TRAILING...)	trim(TRAILING <i>col</i>)	<i>col</i> with trailing spaces removed	7729
	trim(...FROM...)	trim(<i>str</i> FROM <i>col</i>)	<i>col</i> with leading and trailing <i>str</i> removed	7730
	rpad()	rpad(<i>col</i> , <i>len</i>)	<i>col</i> padded on the right to <i>len</i> characters	7731
	rpad()	rpad(<i>col</i> , <i>len</i> , <i>str</i>)	<i>col</i> padded on the right using <i>str</i>	7732
	lpad()	lpad(<i>col</i> , <i>len</i>)	<i>col</i> padded on the left to <i>len</i> characters	7733
	lpad()	lpad(<i>col</i> , <i>len</i> , <i>str</i>)	<i>col</i> padded on the left using <i>str</i>	7734
	upper()	upper(<i>col</i>)	<i>col</i> uppercased	7735
	lower()	lower(<i>col</i>)	<i>col</i> lowercased	7736
	initcap()	initcap(<i>col</i>)	<i>col</i> with the first letter capitalized	7737
	strpos()	strpos(<i>col</i> , <i>str</i>)	position of <i>str</i> in <i>col</i>	7738
	position()	position(<i>str</i> IN <i>col</i>)	same as strpos()	7739
	substr()	substr(<i>col</i> , <i>pos</i>)	<i>col</i> starting at position <i>pos</i>	7740
	substring(...FROM...)	substring(<i>col</i> FROM <i>pos</i>)	same as substr() above	7741
	substr()	substr(<i>col</i> , <i>pos</i> , <i>len</i>)	<i>col</i> starting at position <i>pos</i> for length <i>len</i>	7742
	substring(...FROM...FOR...)	substring(<i>col</i> FROM <i>pos</i> FOR <i>len</i>)	same as substr() above	7743
	translate()	translate(<i>col</i> , <i>from</i> , <i>to</i>)	<i>col</i> with <i>from</i> changed to <i>to</i>	7744
	to_number()	to_number(<i>col</i> , <i>mask</i>)	convert <i>col</i> to NUMERIC() based on <i>mask</i>	7745
to_date	to_date(<i>col</i> , <i>mask</i>)	convert <i>col</i> to DATE based on <i>mask</i>	7746	
to_timestamp	to_timestamp(<i>col</i> , <i>mask</i>)	convert <i>col</i> to TIMESTAMP based on <i>mask</i>	7747	
Number	round()	round(<i>col</i>)	round to an integer	7753
	round()	round(<i>col</i> , <i>len</i>)	NUMERIC() <i>col</i> rounded to <i>len</i> decimal places	7754
	trunc()	trunc(<i>col</i>)	truncate to an integer	7755
	trunc()	trunc(<i>col</i> , <i>len</i>)	NUMERIC() <i>col</i> truncated to <i>len</i> decimal places	7756
	abs()	abs(<i>col</i>)	absolute value	7757
	factorial()	factorial(<i>col</i>)	factorial	7758
	sqrt()	sqrt(<i>col</i>)	square root	7759
	cbrt()	cbrt(<i>col</i>)	cube root	7760
	exp()	exp(<i>col</i>)	exponential	7761
	ln()	ln(<i>col</i>)	natural logarithm	7762
	log()	log(<i>log</i>)	base-10 logarithm	7763
	to_char()	to_char(<i>col</i> , <i>mask</i>)	convert <i>col</i> to a string based on <i>mask</i>	7764
	Temporal	date_part()	date_part(<i>units</i> , <i>col</i>)	<i>units</i> part of <i>col</i>
extract(...FROM...)		extract(<i>units</i> FROM <i>col</i>)	same as date_part()	7766
date_trunc()		date_trunc(<i>units</i> , <i>col</i>)	<i>col</i> rounded to <i>units</i>	7767
isfinite()		isfinite(<i>col</i>)	BOOLEAN indicating if <i>col</i> is a valid date	7768
now()		now()	TIMESTAMP representing current date and time	7769
timeofday()		timeofday()	string showing date/time in UNIX format	7770
overlaps()		overlaps(<i>c1</i> , <i>c2</i> , <i>c3</i> , <i>c4</i>)	BOOLEAN indicating if <i>col</i> 's overlap in time	7771
to_char()		to_char(<i>col</i> , <i>mask</i>)	convert <i>col</i> to string based on <i>mask</i>	7772
Geometric			see <i>psql</i> 's <i>\df</i> for a list of geometric functions	7773
Network	broadcast()	broadcast(<i>col</i>)	broadcast address of <i>col</i>	7774
	host()	host(<i>col</i>)	host address of <i>col</i>	7775
	netmask()	netmask(<i>col</i>)	netmask of <i>col</i>	7776
	masklen()	masklen(<i>col</i>)	mask length of <i>col</i>	7777
	network()	network(<i>col</i>)	network address of <i>col</i>	7778
NULL	nullif()	nullif(<i>col1</i> , <i>col2</i>)	return NULL if <i>col1</i> equals <i>col2</i> , else return <i>col1</i>	7779
	coalesce()	coalesce(<i>col1</i> , <i>col2</i> , ...)	return first non-NULL argument	7780

Table 9.3: Common functions

```

7789
7790
7791
7792
7793 test=> SELECT date_part('year', '5/8/1971');
7794 ERROR:  Function 'date_part(unknown, unknown)' does not exist
7795         Unable to identify a function that satisfies the given argument types
7796         You may need to add explicit typecasts
7797
7798 test=> SELECT date_part('year', CAST('5/8/1971' AS DATE));
7799         date_part
7800 -----
7801
7802         1971
7803
7804 (1 row)
7805
7806
7807
7808
7809
7810
7811
7812
7813
7814
7815
7816
7817
7818

```

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

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

Table 9.4: Common operators

7848
7849
7850
7851
7852
7853
7854

```

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

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

```

Figure 9.3: Error generated by undefined operator/type combination

9.6 Support Variables

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

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

Table 9.5: Common variables

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 (
test(>           col1  INTEGER[5],
test(>           col2  INTEGER[] [],
test(>           col3  INTEGER[2] [2] []
test(> );
CREATE

```

Figure 9.4: Creation of array columns

7855
7856
7857
7858
7859
7860
7861
7862
7863
7864
7865
7866
7867
7868
7869
7870
7871
7872
7873
7874
7875
7876
7877
7878
7879
7880
7881
7882
7883
7884
7885
7886
7887
7888
7889
7890
7891
7892
7893
7894
7895
7896
7897
7898
7899
7900
7901
7902
7903
7904
7905
7906
7907
7908
7909
7910
7911
7912
7913
7914
7915
7916
7917
7918
7919
7920

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, `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 `array_test` table is a three-dimensional array, holding values like `{{{3,1},{1,9}},{4,5},{8,2}}`. This is a three-dimensional matrix made up of two 2×2 matrices. Arrays of any size can be constructed.

Figure 9.5 shows a query inserting values into `array_test`, and several queries selecting data from the table. Brackets are used to access individual array elements.

```

7938 test=> INSERT INTO array_test VALUES (
7939 test(>                                '{1,2,3,4,5}',
7940 test(>                                '{{1,2},{3,4}}',
7942 test(>                                '{{{1,2},{3,4}},{5,6},{7,8}}}'
7943 test(> );
7944 INSERT 52694 1
7946 test=> SELECT * FROM array_test;
7947      col1      |      col2      |      col3
7948 -----+-----+-----
7949 {1,2,3,4,5} | {{1,2},{3,4}} | {{{1,2},{3,4}},{5,6},{7,8}}
7950 (1 row)
7954 test=> SELECT col1[4] FROM array_test;
7955      col1
7956 -----
7957          4
7959 (1 row)
7962 test=> SELECT col2[2][1] FROM array_test;
7963      col2
7964 -----
7965          3
7967 (1 row)
7970 test=> SELECT col3[1][2][2] FROM array_test;
7971      col3
7972 -----
7973          4
7975 (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
test-> VALUES ('peach', lo_import('/usr/images/peach.jpg'));
INSERT 27111 1
test=> SELECT lo_export(fruit.image, '/tmp/outimage.jpg')
test-> FROM   fruit
test-> WHERE  name = 'peach';
 lo_export
-----
          1
(1 row)

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

Figure 9.6: Using large images

/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.

7987
7988
7989
7990
7991
7992
7993
7994
7995
7996
7997
7998
7999
8000
8001
8002
8003
8004
8005
8006
8007
8008
8009
8010
8011
8012
8013
8014
8015
8016
8017
8018
8019
8020
8021
8022
8023
8024
8025
8026
8027
8028
8029
8030
8031
8032
8033
8034
8035
8036
8037
8038
8039
8040
8041
8042
8043
8044
8045
8046
8047
8048
8049
8050
8051
8052

8053
8054
8055
8056
8057
8058
8059
8060
8061
8062
8063
8064
8065
8066
8067
8068
8069
8070
8071
8072
8073
8074
8075
8076
8077
8078
8079
8080
8081
8082
8083
8084
8085
8086
8087
8088
8089
8090
8091
8092
8093
8094
8095
8096
8097
8098
8099
8100
8101
8102
8103
8104
8105
8106
8107
8108
8109
8110
8111
8112
8113
8114
8115
8116
8117
8118

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

```

8119
8120
8121
8122
8123
8124
8125
8126
8127
8128
8129

Figure 10.2: INSERT with explicit transaction

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

```

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

```

8130
8131
8132
8133
8134
8135
8136
8137
8138
8139
8140

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

```

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

```

8141
8142
8143
8144
8145
8146
8147
8148
8149
8150
8151
8152
8153
8154
8155
8156

Figure 10.4: Multi-statement transaction

perform the queries as a single operation.

8157
8158
8159
8160
8161
8162
8163
8164
8165
8166
8167
8168
8169
8170
8171
8172
8173
8174
8175
8176
8177
8178
8179
8180
8181
8182
8183
8184

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

8185
8186
8187
8188
8189
8190
8191
8192
8193
8194
8195
8196
8197
8198
8199
8200
8201
8202
8203
8204
8205
8206
8207
8208
8209
8210
8211
8212
8213
8214
8215
8216
8217
8218
8219
8220
8221
8222
8223
8224
8225
8226
8227
8228
8229
8230
8231
8232
8233
8234
8235
8236
8237
8238
8239
8240
8241
8242
8243
8244
8245
8246
8247
8248
8249
8250

```
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*

User 1	User 2	Notes
	<code>SELECT (*) FROM trans_test</code>	returns 0
<code>INSERT INTO trans_test VALUES (1)</code>		add row to <code>trans_test</code>
<code>SELECT (*) FROM trans_test</code>		returns 1
	<code>SELECT (*) FROM trans_test</code>	returns 1

Table 10.1: Visibility of single-query transactions

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.

User 1	User 2	Notes
BEGIN WORK		User 1 starts a transaction
	SELECT (*) FROM <i>trans_test</i>	returns 0
INSERT INTO <i>trans_test</i> VALUES (1)		add row to <i>trans_test</i>
SELECT (*) FROM <i>trans_test</i>		returns 1
	SELECT (*) FROM <i>trans_test</i>	returns 0
COMMIT WORK		
	SELECT (*) FROM <i>trans_test</i>	returns 1

Table 10.2: Visibility using multi-query transactions

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

```

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(*)

8251
8252
8253
8254
8255
8256
8257
8258
8259
8260
8261
8262
8263
8264
8265
8266
8267
8268
8269
8270
8271
8272
8273
8274
8275
8276
8277
8278
8279
8280
8281
8282
8283
8284
8285
8286
8287
8288
8289
8290
8291
8292
8293
8294
8295
8296
8297
8298
8299
8300
8301
8302
8303
8304
8305
8306
8307
8308
8309
8310
8311
8312
8313
8314
8315
8316

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.

```

8317
8318 shows the newly INSERTED row. When another user commits a transaction, it is seen by the current
8319 transaction, even if it is committed after the current transaction started.
8320
8321 You can prevent your transaction from seeing changes made to the database. SET TRANSACTION ISOLATION
8322 LEVEL SERIALIZABLE changes the isolation level of the current transaction. SERIALIZABLE isolation prevents
8323 the current transaction from seeing commits made by other transactions. Any commit made after the start of
8324 the first query of the transaction is not visible. Figure 10.7 shows an example of a SERIALIZABLE transaction.
8325
8326     test=> BEGIN WORK;
8327     BEGIN
8328     test=> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
8329     SET VARIABLE
8330     test=> SELECT COUNT(*) FROM trans_test;
8331     count
8332     -----
8333         5
8334     (1 row)
8335
8336
8337
8338
8339     test=> --
8340     test=> -- someone commits INSERT INTO trans_test
8341     test=> --
8342     test=> SELECT COUNT(*) FROM trans_test;
8343     count
8344     -----
8345         5
8346     (1 row)
8347
8348
8349
8350
8351     test=> COMMIT WORK;
8352     COMMIT
8353
8354
8355
8356
8357
8358
8359

```

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-

8381
8382

Transaction 1	Transaction 2	Notes
BEGIN WORK	BEGIN WORK	Start both transactions
UPDATE row 64		Transaction 1 exclusively locks row 64
	UPDATE row 64	Transaction 2 must wait to see if first transaction commits
COMMIT WORK		Transaction 1 commits. Transaction 2 returns from UPDATE.
	COMMIT WORK	Transaction 2 commits

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

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

Figure 10.8: SELECT with no locking

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.

8383
8384
8385
8386
8387
8388
8389
8390
8391
8392
8393
8394
8395
8396
8397
8398
8399
8400
8401
8402
8403
8404
8405
8406
8407
8408
8409
8410
8411
8412
8413
8414
8415
8416
8417
8418
8419
8420
8421
8422
8423
8424
8425
8426
8427
8428
8429
8430
8431
8432
8433
8434
8435
8436
8437
8438
8439
8440
8441
8442
8443
8444
8445
8446
8447
8448


```

8449      test=> BEGIN WORK;
8450      BEGIN
8451      test=> SELECT *
8452      test-> FROM lock_test
8453      test-> WHERE name = 'James'
8454      test-> FOR UPDATE;
8455      id |          name
8456      ----+-----
8457      521 | James
8458      (1 row)
8459
8460      test=> --
8461      test=> -- the SELECTed row is locked
8462      test=> --
8463      test=> UPDATE lock_test
8464      test-> SET name = 'Jim'
8465      test-> WHERE name = 'James';
8466      UPDATE 1
8467      test=> COMMIT WORK;
8468      COMMIT
8469
8470
8471
8472
8473
8474
8475
8476
8477
8478
8479
8480
8481
8482
8483
8484
8485
8486
8487
8488
8489
8490
8491
8492
8493
8494
8495
8496
8497
8498
8499
8500
8501
8502
8503
8504
8505
8506
8507
8508
8509
8510
8511
8512
8513
8514

```

Figure 10.9: SELECT...FOR UPDATE

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 1	Transaction2	Notes
BEGIN WORK	BEGIN WORK	Start both transactions
UPDATE row 64	UPDATE row 83	Independent rows write locked
UPDATE row 83		Holds waiting for transaction 2 to release write lock
	UPDATE row 64	Attempt to get write lock held by transaction 1
	auto-ROLLBACK WORK	Deadlock detected — transaction 2 automatically rolled back
COMMIT WORK		Transaction 1 returns from UPDATE and commits

Table 10.4: Deadlock

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

10.7 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.

8515
8516
8517
8518
8519
8520
8521
8522
8523
8524
8525
8526
8527
8528
8529
8530
8531
8532
8533
8534
8535
8536
8537
8538
8539
8540
8541
8542
8543
8544
8545
8546
8547
8548
8549
8550
8551
8552
8553
8554
8555
8556
8557
8558
8559
8560
8561
8562
8563
8564
8565
8566
8567
8568
8569
8570
8571
8572
8573
8574
8575
8576
8577
8578
8579
8580

8581
8582
8583
8584
8585
8586
8587
8588
8589
8590
8591
8592
8593
8594
8595
8596
8597
8598
8599
8600
8601
8602
8603
8604
8605
8606
8607
8608
8609
8610
8611
8612
8613
8614
8615
8616
8617
8618
8619
8620
8621
8622
8623
8624
8625
8626
8627
8628
8629
8630
8631
8632
8633
8634
8635
8636
8637
8638
8639
8640
8641
8642
8643
8644
8645
8646

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 `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,

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

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

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

Figure 11.2: Example of a unique index

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 do 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.

8647
8648
8649
8650
8651
8652
8653
8654
8655
8656
8657
8658
8659
8660
8661
8662
8663
8664
8665
8666
8667
8668
8669
8670
8671
8672
8673
8674
8675
8676
8677
8678
8679
8680
8681
8682
8683
8684
8685
8686
8687
8688
8689
8690
8691
8692
8693
8694
8695
8696
8697
8698
8699
8700
8701
8702
8703
8704
8705
8706
8707
8708
8709
8710
8711
8712

CLUSTER even helps with range queries like `col >= 3 AND col <= 5`. CLUSTER places these rows next to each other on disk, speeding indexed lookups.

CLUSTER can also speed ORDER BY processing. See the CLUSTER manual page for more information.

11.4 Vacuum

When POSTGRESQL updates a row, it keeps the old copy of the row in the table file and writes a new one. The old row is marked as expired, and 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 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 VACUUM command should be run periodically to clean out expired rows. For tables that are heavily modified, it is useful to run VACUUM every night in an automated manner. For tables with few modifications, VACUUM should be run only periodically. VACUUM exclusively locks the table while processing.

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

11.5 Vacuum Analyze

The VACUUM ANALYZE command is like 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 POSTGRESQL when deciding how to efficiently execute complex queries. VACUUM ANALYZE should be run when a table is initially loaded, and when the table data dramatically changes.

The VACUUM manual page shows all of the VACUUM options.

11.6 EXPLAIN

EXPLAIN causes POSTGRESQL to display how a query will be executed, rather than executing it. For example, figure 11.3 shows a SELECT query preceded by the word EXPLAIN. In the figure, 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)

EXPLAIN
```

Figure 11.3: Using EXPLAIN

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

Figure 11.4 shows more interesting examples of EXPLAIN. The first EXPLAIN shows a SELECT with the restriction *customer_id = 55*. This is again a *sequential scan*, but the restriction causes POSTGRESQL to estimate ten rows will be returned. A 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 *index scan*

```

8779
8780
8781
8782
8783
8784
8785
8786
8787
8788
8789
8790
8791
8792
8793
8794
8795
8796
8797
8798
8799
8800
8801
8802
8803
8804
8805
8806
8807
8808
8809
8810
8811
8812
8813
8814
8815
8816
8817
8818
8819
8820
8821
8822
8823
8824
8825
8826
8827
8828
8829
8830
8831
8832
8833
8834
8835
8836
8837
8838
8839
8840
8841
8842
8843
8844

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,

```

8845
8846
8847
8848
8849
8850
8851      test=> EXPLAIN SELECT * FROM tab1, tab2 WHERE col1 = col2;
8852      NOTICE: QUERY PLAN:
8853
8854
8855      Merge Join  (cost=139.66..164.66 rows=10000 width=8)
8856      ->  Sort    (cost=69.83..69.83 rows=1000 width=4)
8857          ->  Seq Scan on tab2  (cost=0.00..20.00 rows=1000 width=4)
8858      ->  Sort    (cost=69.83..69.83 rows=1000 width=4)
8859          ->  Seq Scan on tab1  (cost=0.00..20.00 rows=1000 width=4)
8860
8861
8862
8863      EXPLAIN
8864
8865
8866
8867
8868
8869
8870
8871
8872
8873
8874
8875
8876
8877
8878
8879
8880
8881
8882
8883
8884
8885
8886
8887
8888
8889
8890
8891
8892
8893
8894
8895
8896
8897
8898
8899
8900
8901
8902
8903
8904
8905
8906
8907
8908
8909
8910

```

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.

8911
8912
8913
8914
8915
8916
8917
8918
8919
8920
8921
8922
8923
8924
8925
8926
8927
8928
8929
8930
8931
8932
8933
8934
8935
8936
8937
8938
8939
8940
8941
8942
8943
8944
8945
8946
8947
8948
8949
8950
8951
8952
8953
8954
8955
8956
8957
8958
8959
8960
8961
8962
8963
8964
8965
8966
8967
8968
8969
8970
8971
8972
8973
8974
8975
8976

8977
8978
8979
8980
8981
8982
8983
8984
8985
8986
8987
8988
8989
8990
8991
8992
8993
8994
8995
8996
8997
8998
8999
9000
9001
9002
9003
9004
9005
9006
9007
9008
9009
9010
9011
9012
9013
9014
9015
9016
9017
9018
9019
9020
9021
9022
9023
9024
9025
9026
9027
9028
9029
9030
9031
9032
9033
9034
9035
9036
9037
9038
9039
9040
9041
9042

Chapter 12

Controlling Results

When a `SELECT` query is issued from `psql`, it travels to the `POSTGRESQL` server, is executed, and the result sent back to `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 `customer` has 1000 rows with `customer_id` values from 1 to 1000. Figure 12.1 shows queries using `LIMIT` and `LIMIT...OFFSET`. The first query sorts the table by `customer_id` and uses `LIMIT` to

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

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

Figure 12.1: Examples of `LIMIT` and `LIMIT/OFFSET`

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.

9043
9044
9045
9046
9047
9048
9049
9050
9051
9052
9053
9054
9055
9056
9057
9058
9059
9060
9061
9062
9063
9064
9065
9066
9067
9068
9069
9070
9071
9072
9073
9074
9075
9076
9077
9078
9079
9080
9081
9082
9083
9084
9085
9086
9087
9088
9089
9090
9091
9092
9093
9094
9095
9096
9097
9098
9099
9100
9101
9102
9103
9104
9105
9106
9107
9108

```
9109      test=> BEGIN WORK;
9110      BEGIN
9111      test=> DECLARE customer_cursor CURSOR FOR
9112      test-> SELECT customer_id FROM customer;
9113      SELECT
9114      test=> FETCH 1 FROM customer_cursor;
9115      customer_id
9116      -----
9117      1
9118      (1 row)
9119
9120      test=> FETCH 1 FROM customer_cursor;
9121      customer_id
9122      -----
9123      2
9124      (1 row)
9125
9126      test=> FETCH 2 FROM customer_cursor;
9127      customer_id
9128      -----
9129      3
9130      4
9131      (2 rows)
9132
9133      test=> FETCH -1 FROM customer_cursor;
9134      customer_id
9135      -----
9136      3
9137      (1 row)
9138
9139      test=> FETCH -1 FROM customer_cursor;
9140      customer_id
9141      -----
9142      2
9143      (1 row)
9144
9145      test=> MOVE 10 FROM customer_cursor;
9146      MOVE
9147      test=> FETCH 1 FROM customer_cursor;
9148      customer_id
9149      -----
9150      13
9151      (1 row)
9152      test=> CLOSE customer_cursor;
9153      CLOSE
9154      test=> COMMIT WORK;
9155      COMMIT
9156
9157
9158
9159
9160
9161
9162
9163
9164
9165
9166
9167
9168
9169
9170
9171
9172
9173
9174
```

Figure 12.2: Cursor usage

9175
9176
9177
9178
9179
9180
9181
9182
9183
9184
9185
9186
9187
9188
9189
9190
9191
9192
9193
9194
9195
9196
9197
9198
9199
9200
9201
9202
9203
9204
9205
9206
9207
9208
9209
9210
9211
9212
9213
9214
9215
9216
9217
9218
9219
9220
9221
9222
9223
9224
9225
9226
9227
9228
9229
9230
9231
9232
9233
9234
9235
9236
9237
9238
9239
9240

9241
9242
9243
9244
9245
9246
9247
9248
9249
9250
9251
9252
9253
9254
9255
9256
9257
9258
9259
9260
9261
9262
9263
9264
9265
9266
9267
9268
9269
9270
9271
9272
9273
9274
9275
9276
9277
9278
9279
9280
9281
9282
9283
9284
9285
9286
9287
9288
9289
9290
9291
9292
9293
9294
9295
9296
9297
9298
9299
9300
9301
9302
9303
9304
9305
9306

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

User 1	User 2
CREATE TEMPORARY TABLE <i>temptest</i> (<i>col</i> INTEGER)	CREATE TEMPORARY TABLE <i>temptest</i> (<i>col</i> INTEGER)
INSERT INTO <i>temptest</i> VALUES (1)	INSERT INTO <i>temptest</i> VALUES (2)
SELECT <i>col</i> FROM <i>temptest</i> returns 1	SELECT <i>col</i> FROM <i>temptest</i> returns 2

Table 13.1: Temporary table isolation

same name.

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
$ 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

9307
9308
9309
9310
9311
9312
9313
9314
9315
9316
9317
9318
9319
9320
9321
9322
9323
9324
9325
9326
9327
9328
9329
9330
9331
9332
9333
9334
9335
9336
9337
9338
9339
9340
9341
9342
9343
9344
9345
9346
9347
9348
9349
9350
9351
9352
9353
9354
9355
9356
9357
9358
9359
9360
9361
9362
9363
9364
9365
9366
9367
9368
9369
9370
9371
9372

- add columns
- add column defaults
- remove column defaults

Figure 13.3 shows examples of all of these.

```

9373
9374
9375
9376
9377
9378
9379
9380
9381
9382
9383
9384 test=> CREATE TABLE altertest (col1 INTEGER);
9385 CREATE
9386 test=> ALTER TABLE altertest RENAME TO alterdemo;
9387 ALTER
9388 test=> ALTER TABLE alterdemo RENAME COLUMN col1 TO democol;
9389 ALTER
9390 test=> ALTER TABLE alterdemo ADD COLUMN col2 INTEGER;
9391 ALTER
9392 test=> -- show renamed table, renamed column, and new column
9393 test=> \d alterdemo
9394
9395      Table "alterdemo"
9396      Attribute | Type   | Modifier
9397      -----+-----+-----
9400      democol  | integer |
9401      col2     | integer |
9402
9403
9404
9405 test=> ALTER TABLE alterdemo ALTER COLUMN col2 SET DEFAULT 0;
9406 ALTER
9407 test=> -- show new default value
9408 test=> \d alterdemo
9409
9410      Table "alterdemo"
9411      Attribute | Type   | Modifier
9412      -----+-----+-----
9415      democol  | integer |
9416      col2     | integer | default 0
9417
9418 test=> ALTER TABLE alterdemo ALTER COLUMN col2 DROP DEFAULT;
9419 ALTER
9420
9421
9422
9423
9424
9425
9426
9427
9428
9429
9430
9431
9432
9433
9434
9435
9436
9437
9438

```

Figure 13.3: ALTER TABLE examples

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.

```

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

```

Figure 13.4: Examples of the GRANT command

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

```

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

```

```

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

```

Figure 13.5: Creation of inherited tables

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.

9439
9440
9441
9442
9443
9444
9445
9446
9447
9448
9449
9450
9451
9452
9453
9454
9455
9456
9457
9458
9459
9460
9461
9462
9463
9464
9465
9466
9467
9468
9469
9470
9471
9472
9473
9474
9475
9476
9477
9478
9479
9480
9481
9482
9483
9484
9485
9486
9487
9488
9489
9490
9491
9492
9493
9494
9495
9496
9497
9498
9499
9500
9501
9502
9503
9504


```

9505      test=> INSERT INTO parent_test VALUES (1);
9506      INSERT 18837 1
9507
9508      test=> INSERT INTO child_test VALUES (2,3);
9509      INSERT 18838 1
9510
9511      test=> SELECT * FROM parent_test;
9512      col1
9513      -----
9514      1
9515      (1 row)
9516
9517
9518
9519      test=> SELECT * FROM child_test;
9520      col1 | col2
9521      -----+-----
9522      2 | 3
9523      (1 row)
9524
9525
9526
9527      test=> SELECT * FROM parent_test*;
9528      col1
9529      -----
9530      1
9531      2
9532      (2 rows)
9533
9534
9535
9536
9537
9538
9539
9540

```

Figure 13.6: Accessing inherited tables

```

9541      test=> CREATE TABLE grandchild_test (col3 INTEGER) INHERITS (child_test);
9542      CREATE
9543
9544      test=> INSERT INTO grandchild_test VALUES (4, 5, 6);
9545      INSERT 18853 1
9546
9547      test=> SELECT * FROM parent_test*;
9548      col1
9549      -----
9550      1
9551      2
9552      4
9553      (3 rows)
9554
9555
9556
9557      test=> SELECT * FROM child_test*;
9558      col1 | col2
9559      -----+-----
9560      2 | 3
9561      4 | 5
9562      (2 rows)
9563
9564
9565
9566
9567
9568
9569
9570

```

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

```

Figure 13.8: Examples of views

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.

9571
9572
9573
9574
9575
9576
9577
9578
9579
9580
9581
9582
9583
9584
9585
9586
9587
9588
9589
9590
9591
9592
9593
9594
9595
9596
9597
9598
9599
9600
9601
9602
9603
9604
9605
9606
9607
9608
9609
9610
9611
9612
9613
9614
9615
9616
9617
9618
9619
9620
9621
9622
9623
9624
9625
9626
9627
9628
9629
9630
9631
9632
9633
9634
9635
9636

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

```

9637
9638
9639
9640
9641
9642
9643
9644     test=> CREATE TABLE ruletest (col INTEGER);
9645     CREATE
9646
9647     test=> CREATE RULE ruletest_insert AS      -- rule name
9648     test-> ON INSERT TO ruletest             -- INSERT rule
9649
9650     test-> DO INSTEAD                          -- DO INSTEAD-type rule
9651     test->     NOTHING;                        -- ACTION is NOTHING
9652
9653     CREATE 18932 1
9654     test=> INSERT INTO ruletest VALUES (1);
9655     test=> SELECT * FROM ruletest;
9656         col
9657         ----
9658         (0 rows)

```

Figure 13.9: Rule that prevents INSERT

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.

9637
9638
9639
9640
9641
9642
9643
9644
9645
9646
9647
9648
9649
9650
9651
9652
9653
9654
9655
9656
9657
9658
9659
9660
9661
9662
9663
9664
9665
9666
9667
9668
9669
9670
9671
9672
9673
9674
9675
9676
9677
9678
9679
9680
9681
9682
9683
9684
9685
9686
9687
9688
9689
9690
9691
9692
9693
9694
9695
9696
9697
9698
9699
9700
9701
9702

```

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

```

Figure 13.10: Rules to log table changes

9703
9704
9705
9706
9707
9708
9709
9710
9711
9712
9713
9714
9715
9716
9717
9718
9719
9720
9721
9722
9723
9724
9725
9726
9727
9728
9729
9730
9731
9732
9733
9734
9735
9736
9737
9738
9739
9740
9741
9742
9743
9744
9745
9746
9747
9748
9749
9750
9751
9752
9753
9754
9755
9756
9757
9758
9759
9760
9761
9762
9763
9764
9765
9766
9767
9768

```

9769
9770      test=> INSERT INTO service_request (customer_id, description)
9771      test-> VALUES (72321, 'Fix printing press');
9772      INSERT 18808 1
9773      test=> UPDATE service_request
9774      test-> SET description = 'Fix large printing press'
9775      test-> WHERE customer_id = 72321;
9776      UPDATE 1
9777      test=> DELETE FROM service_request
9778      test-> WHERE customer_id = 72321;
9779      DELETE 1
9780      test=> SELECT *
9781      test-> FROM service_request_log
9782      test-> WHERE customer_id = 72321;
9783      customer_id |          description          | mod_type | mod_user |      mod_timestamp
9784      -----+-----+-----+-----+-----
9785      72321 | Fix printing press      | U        | williams | 2000-04-09 07:13:07-04
9786      72321 | Fix large printing press | D        | matheson | 2000-04-10 12:47:20-04
9787      (2 rows)

```

Figure 13.11: Use of rule to log table changes

```

9800
9801
9802
9803      test=> CREATE TABLE realtable (col INTEGER);
9804      CREATE
9805      test=> CREATE VIEW view_realtable AS SELECT * FROM realtable;
9806      CREATE 407890 1
9807      test=> INSERT INTO realtable VALUES (1);
9808      INSERT 407891 1
9809      test=> INSERT INTO view_realtable VALUES (2);
9810      INSERT 407893 1
9811      test=> SELECT * FROM realtable;
9812      col
9813      ----
9814      1
9815      (1 row)
9816
9817      test=> SELECT * FROM view_realtable;
9818      col
9819      ----
9820      1
9821      (1 row)

```

Figure 13.12: Views ignore table modifications

9830
9831
9832
9833
9834

```

test=> CREATE RULE view_realtable_insert AS      -- INSERT rule      9835
test-> ON INSERT TO view_realtable              9836
test-> DO INSTEAD                               9837
test->     INSERT INTO realtable                9838
test->     VALUES (new.col);                   9839
CREATE 407894 1                                 9840
test=>                                          9841
test=>                                          9842
test=> CREATE RULE view_realtable_update AS     -- UPDATE rule      9843
test-> ON UPDATE TO view_realtable            9844
test-> DO INSTEAD                              9845
test->     UPDATE realtable                    9846
test->     SET col = new.col                   9847
test->     WHERE col = old.col;                9848
CREATE 407901 1                                 9849
test=>                                          9850
test=>                                          9851
test=> CREATE RULE view_realtable_delete AS    -- DELETE rule     9852
test-> ON DELETE TO view_realtable           9853
test-> DO INSTEAD                              9854
test->     DELETE FROM realtable              9855
test->     WHERE col = old.col;               9856
CREATE 407902 1                                 9857
test=>                                          9858
test=>                                          9859
test=>                                          9860
test=>                                          9861
test=>                                          9862
test=>                                          9863
test=>                                          9864
test=>                                          9865
test=>                                          9866
test=>                                          9867
test=>                                          9868
test=>                                          9869
test=>                                          9870
test=>                                          9871
test=>                                          9872
test=>                                          9873
test=>                                          9874
test=>                                          9875
test=>                                          9876
test=>                                          9877
test=>                                          9878
test=>                                          9879
test=>                                          9880
test=>                                          9881
test=>                                          9882
test=>                                          9883
test=>                                          9884
test=>                                          9885
test=>                                          9886
test=>                                          9887
test=>                                          9888
test=>                                          9889
test=>                                          9890
test=>                                          9891
test=>                                          9892
test=>                                          9893
test=>                                          9894
test=>                                          9895
test=>                                          9896
test=>                                          9897
test=>                                          9898
test=>                                          9899
test=>                                          9900

```

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.

```
9901
9902
9903
9904
9905
9906
9907
9908
9909
9910
9911
9912
9913
9914      test=> INSERT INTO view_realtable VALUES (3);
9915      INSERT 407895 1
9916      test=> SELECT * FROM view_realtable;
9917      col
9918      -----
9919      1
9920      3
9921      (2 rows)
9922
9923      test=> UPDATE view_realtable
9924      test-> SET col = 4;
9925      UPDATE 2
9926      test=> SELECT * FROM view_realtable;
9927      col
9928      -----
9929      4
9930      4
9931      (2 rows)
9932
9933      test=> DELETE FROM view_realtable;
9934      DELETE 2
9935      test=> SELECT * FROM view_realtable;
9936      col
9937      -----
9938      (0 rows)
9939
9940
9941
9942
9943
9944
9945
9946
9947
9948
9949
9950
9951
9952
9953
9954
9955
9956
9957
9958
9959
9960
9961
9962
9963
9964
9965
9966
```

Figure 13.14: Rules handle view modifications

9967
9968
9969
9970
9971
9972
9973
9974
9975
9976
9977
9978
9979
9980
9981
9982
9983
9984
9985
9986
9987
9988
9989
9990
9991
9992
9993
9994
9995
9996
9997
9998
9999
10000
10001
10002
10003
10004
10005
10006
10007
10008
10009
10010
10011
10012
10013
10014
10015
10016
10017
10018
10019
10020
10021
10022
10023
10024
10025
10026
10027
10028
10029
10030
10031
10032

10033
10034
10035
10036
10037
10038
10039
10040
10041
10042
10043
10044
10045
10046
10047
10048
10049
10050
10051
10052
10053
10054
10055
10056
10057
10058
10059
10060
10061
10062
10063
10064
10065
10066
10067
10068
10069
10070
10071
10072
10073
10074
10075
10076
10077
10078
10079
10080
10081
10082
10083
10084
10085
10086
10087
10088
10089
10090
10091
10092
10093
10094
10095
10096
10097
10098

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

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

Figure 14.1: NOT NULL constraint

cause the INSERT to fail. The figure shows UPDATE of a NULL value also fails.

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

```

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

```

Figure 14.2: NOT NULL with DEFAULT constraint

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

```

test=> CREATE TABLE uniquetest (col1 INTEGER UNIQUE);
NOTICE: CREATE TABLE/UNIQUE will create implicit index 'uniquetest_col1_
key' for table 'uniquetest'
CREATE
test=> \d uniquetest
      Table "uniquetest"
  Attribute | Type      | Modifier
-----+-----+-----
   col1    | integer  |
Index: uniquetest_col1_key

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

```

Figure 14.3: Unique column constraint

figure also shows multiple NULL values can be inserted into a UNIQUE column.

If a UNIQUE constraint is made up of more than one column, UNIQUE cannot be used as a column constraint.

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

```

10165
10166
10167
10168     test=> CREATE TABLE uniuqetest2 (
10169     test(>             col1 INTEGER,
10170     test(>             col2 INTEGER,
10171     test(>             UNIQUE (col1, col2)
10172     test(>             );
10173
10174     NOTICE: CREATE TABLE/UNIQUE will create implicit index 'uniuqetest2_col1_
10175     key' for table 'uniuqetest2'
10176
10177
10178
10179
10180
10181
10182
10183
10184
10185
10186
10187
10188
10189
10190
10191
10192
10193
10194
10195
10196
10197
10198
10199
10200
10201
10202
10203
10204
10205
10206
10207
10208
10209
10210
10211
10212
10213
10214
10215
10216
10217
10218
10219
10220
10221
10222
10223
10224
10225
10226
10227
10228
10229
10230

```

Figure 14.4: Multi-column unique constraint

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,

```

10198     test=> CREATE TABLE primarytest (col INTEGER PRIMARY KEY);
10199     NOTICE: CREATE TABLE/PRIMARY KEY will create implicit index 'primarytest_
10200     pkey' for table 'primarytest'
10201
10202     CREATE
10203     test=> \d primarytest
10204           Table "primarytest"
10205           Attribute | Type      | Modifier
10206           -----+-----+-----
10207           col       | integer  | not null
10208
10209           Index: primarytest_pkey
10210
10211
10212
10213
10214
10215
10216
10217
10218
10219
10220
10221
10222
10223
10224
10225
10226
10227
10228
10229
10230

```

Figure 14.5: Creation of PRIMARY KEY column

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 (
test(>             col1 INTEGER,
test(>             col2 INTEGER,
test(>             PRIMARY KEY(col1, col2)
test(>             );
NOTICE: CREATE TABLE/PRIMARY KEY will create implicit index 'primarytest2_
pkey' for table 'primarytest2'
CREATE

```

Figure 14.6: Example of a 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

```

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

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

```

Figure 14.7: Foreign key creation

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.

10231
10232
10233
10234
10235
10236
10237
10238
10239
10240
10241
10242
10243
10244
10245
10246
10247
10248
10249
10250
10251
10252
10253
10254
10255
10256
10257
10258
10259
10260
10261
10262
10263
10264
10265
10266
10267
10268
10269
10270
10271
10272
10273
10274
10275
10276
10277
10278
10279
10280
10281
10282
10283
10284
10285
10286
10287
10288
10289
10290
10291
10292
10293
10294
10295
10296

```

10297 test=> INSERT INTO customer (state)
10298 test-> VALUES ('AL');
10299 INSERT 148732 1
10300
10301 test=> INSERT INTO customer (state)
10302 test-> VALUES ('XX');
10303 ERROR: <unnamed> referential integrity violation -
10304 key referenced from customer not found in statename
10305
10306
10307
10308
10309
10310
10311

```

Figure 14.8: Foreign key constraints

```

10312 test=> CREATE TABLE customer (
10313 test(>         customer_id INTEGER PRIMARY KEY,
10314 test(>         name          CHAR(30),
10315 test(>         telephone    CHAR(20),
10316 test(>         street        CHAR(40),
10317 test(>         city          CHAR(25),
10318 test(>         state         CHAR(2),
10319 test(>         zipcode       CHAR(10),
10320 test(>         country      CHAR(20)
10321 test(> );
10322 CREATE
10323 test=> CREATE TABLE employee (
10324 test(>         employee_id INTEGER PRIMARY KEY,
10325 test(>         name          CHAR(30),
10326 test(>         hire_date    DATE
10327 test(> );
10328 CREATE
10329 test=> CREATE TABLE part (
10330 test(>         part_id     INTEGER PRIMARY KEY,
10331 test(>         name        CHAR(30),
10332 test(>         cost        NUMERIC(8,2),
10333 test(>         weight     FLOAT
10334 test(> );
10335 CREATE
10336 test=> CREATE TABLE salesorder (
10337 test(>         order_id    INTEGER,
10338 test(>         customer_id INTEGER REFERENCES customer,
10339 test(>         employee_id INTEGER REFERENCES employee,
10340 test(>         part_id     INTEGER REFERENCES part,
10341 test(>         order_date  DATE,
10342 test(>         ship_date   DATE,
10343 test(>         payment    NUMERIC(8,2)
10344 test(> );
10345 CREATE
10346
10347
10348
10349
10350
10351
10352
10353
10354
10355
10356
10357
10358
10359
10360
10361
10362

```

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

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

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

Figure 14.10: Customer table with foreign key actions

`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.

```
10429
10430
10431
10432
10433 test=> CREATE TABLE primarytest (col INTEGER PRIMARY KEY);
10434 NOTICE: CREATE TABLE/PRIMARY KEY will create implicit index 'primarytest_
10435 pkey' for table 'primarytest'
10436
10437 CREATE
10438 test=> CREATE TABLE foreigntest (
10439 test(> col2 INTEGER REFERENCES primarytest
10440 test(> ON UPDATE CASCADE
10441 test(> ON DELETE NO ACTION
10442 test(> );
10443
10444 NOTICE: CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
10445 CREATE
10446 test=> INSERT INTO primarytest values (1);
10447 INSERT 148835 1
10448 test=> INSERT INTO foreigntest values (1);
10449 INSERT 148836 1
10450 test=>
10451 test=> -- CASCADE UPDATE is performed
10452 test=>
10453 test=> UPDATE primarytest SET col = 2;
10454 UPDATE 1
10455 test=> SELECT * FROM foreigntest;
10456 col2
10457 -----
10458 2
10459 (1 row)
10460
10461 test=>
10462 test=> -- NO ACTION prevents deletion
10463 test=>
10464 test=> DELETE FROM primarytest;
10465 ERROR: <unnamed> referential integrity violation -
10466 key in primarytest still referenced from foreigntest
10467 test=>
10468 test=> -- By deleting the foreign key first, the DELETE succeeds
10469 test=>
10470 test=> DELETE FROM foreigntest;
10471 DELETE 1
10472 test=> DELETE FROM primarytest;
10473 DELETE 1
```

Figure 14.11: Foreign key actions

```
10487
10488
10489
10490
10491
10492
10493
10494
```

```

test=> CREATE TABLE primarytest2 (
test(>             col1 INTEGER,
test(>             col2 INTEGER,
test(>             PRIMARY KEY(col1, col2)
test(>             );
NOTICE: CREATE TABLE/PRIMARY KEY will create implicit index 'primarytest2_
pkey' for table 'primarytest2'
CREATE
test=> CREATE TABLE foreigntest2 (col3 INTEGER,
test(>             col4 INTEGER,
test(>             FOREIGN KEY (col3, col4) REFERENCES primarytest2
test->             );
NOTICE: CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
CREATE

```

Figure 14.12: Example of a 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.


```

10561
10562
10563
10564
10565
10566
10567
10568
10569
10570
10571
10572
10573
10574
10575      test=> INSERT INTO primarytest2
10576      test-> VALUES (1,2);
10577      INSERT 148816 1
10579      test=> INSERT INTO foreigntest2
10580      test-> VALUES (1,2);
10581      INSERT 148817 1
10583      test=> UPDATE foreigntest2
10584      test-> SET col4 = NULL;
10585      UPDATE 1
10587      test=> CREATE TABLE matchtest (
10588      test(>                col3 INTEGER,
10590      test(>                col4 INTEGER,
10591      test(>                FOREIGN KEY (col3, col4) REFERENCES primarytest2
10592                        MATCH FULL
10593      test(>
10594      test(>                );
10595      NOTICE: CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
10596      CREATE
10598      test=> UPDATE matchtest
10599      test-> SET col3 = NULL, col4 = NULL;
10600      UPDATE 1
10602      test=> UPDATE matchtest
10603      test-> SET col4 = NULL;
10605      ERROR: <unnamed> referential integrity violation -
10606      MATCH FULL doesn't allow mixing of NULL and NON-NULL key values
10607
10608
10609
10610
10611
10612
10613
10614
10615
10616
10617
10618
10619
10620
10621
10622
10623
10624
10625
10626

```

Figure 14.13: MATCH FULL foreign key

```

10627
10628
10629
10630
10631
10632
10633
10634
10635
10636
10637
10638
10639
test=> CREATE TABLE defertest(
10640
test(>          col2 INTEGER REFERENCES primary-
10641
test test(>          DEFERRABLE
10642
test(> );
10643
NOTICE: CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
10644
CREATE
10645
test=> BEGIN;
10646
BEGIN
10647
test=> -- INSERT is attempted in non-DEFERRABLE mode
10648
test=>
10649
test=> INSERT INTO defertest VALUES (5);
10650
ERROR: <unnamed> referential integrity violation -
10651
key referenced from defertest not found in primarytest
10652
test=> COMMIT;
10653
COMMIT
10654
test=> BEGIN;
10655
BEGIN
10656
test=> -- all foreign key constraints are set to DEFERRED
10657
test=>
10658
test=> SET CONSTRAINTS ALL DEFERRED;
10659
SET CONSTRAINTS
10660
test=> INSERT INTO defertest VALUES (5);
10661
INSERT 148946 1
10662
test=> INSERT INTO primarytest VALUES (5);
10663
INSERT 148947 1
10664
test=> COMMIT;
10665
COMMIT
10666
10667
10668
10669
10670
10671
10672
10673
10674
10675
10676
10677
10678
10679
10680
10681
10682
10683
10684
10685
10686
10687
10688
10689
10690
10691
10692

```

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:

```

10693
10694
10695
10696
10697
10698
10699
10700     test=> CREATE TABLE friend2 (
10701     test(>         firstname CHAR(15),
10702     test(>         lastname CHAR(20),
10703     test(>         city      CHAR(15),
10704     test(>         state     CHAR(2)      CHECK (length(trim(state)) = 2),
10705     test(>         age       INTEGER     CHECK (age >= 0),
10706     test(>         gender    CHAR(1)     CHECK (gender IN ('M','F')),
10707     test(>         last_met  DATE       CHECK (last_met BETWEEN '1950-01-01'
10708     test(>                                     AND CURRENT_DATE),
10709     test(>         CHECK (upper(trim(firstname)) != 'AL' OR
10710     test(>                                     upper(trim(lastname)) != 'RIVERS')
10711     test(> );
10712
10713     CREATE
10714     test=> INSERT INTO friend2
10715     test-> VALUES ('Al', 'Rivers', 'Wibbleville', 'J', -35, 'S', '1931-09-23');
10716
10717     ERROR: ExecAppend: rejected due to CHECK constraint friend2_last_met
10718
10719
10720
10721
10722
10723
10724
10725
10726
10727
10728
10729
10730
10731
10732
10733
10734
10735
10736
10737
10738
10739
10740
10741
10742
10743
10744
10745
10746
10747
10748
10749
10750
10751
10752
10753
10754
10755
10756
10757
10758

```

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 *AL*'s 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.

10759
10760
10761
10762
10763
10764
10765
10766
10767
10768
10769
10770
10771
10772
10773
10774
10775
10776
10777
10778
10779
10780
10781
10782
10783
10784
10785
10786
10787
10788
10789
10790
10791
10792
10793
10794
10795
10796
10797
10798
10799
10800
10801
10802
10803
10804
10805
10806
10807
10808
10809
10810
10811
10812
10813
10814
10815
10816
10817
10818
10819
10820
10821
10822
10823
10824

10825
10826
10827
10828
10829
10830
10831
10832
10833
10834
10835
10836
10837
10838
10839
10840
10841
10842
10843
10844
10845
10846
10847
10848
10849
10850
10851
10852
10853
10854
10855
10856
10857
10858
10859
10860
10861
10862
10863
10864
10865
10866
10867
10868
10869
10870
10871
10872
10873
10874
10875
10876
10877
10878
10879
10880
10881
10882
10883
10884
10885
10886
10887
10888
10889

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 psql 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 psql. 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

¹Non-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.

```

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

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

```

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

```

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

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

```

Figure 15.2: Example of COPY...FROM

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.

```

10971 test=> COPY copytest TO '/tmp/copytest.out' USING DELIMITERS '|';
10972 COPY
10973 test=> \q
10974 $ cat /tmp/copytest.out
10975 1|23.99|fresh spring water|t
10976 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

```

10989 test=> DELETE FROM copytest;
10990 DELETE 2
10991 test=>
10992 test=> COPY copytest FROM '/tmp/copytest.out';
10993 ERROR: copy: line 1, pg_atoi: error in "1|23.99|fresh spring water|t": can-
10994 not parse "|23.99|fresh spring water|t"
10995 test=>
10996 test=> COPY copytest FROM '/tmp/copytest.out' USING DELIMITERS '|';
11000 COPY

```

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

separate the columns. The second COPY...FROM succeeds because the proper delimiter for the file was used.

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.

```

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

```

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

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 precedes 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

```

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

```

Figure 15.6: COPY backslash handling

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 preceding 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.

Backslash string	Meaning
\TAB	TAB if using default delimiter TAB
\	<i>pipe</i> if using <i>pipe</i> as the delimiter
\N	NULL if using the default NULL output
\b	backspace
\f	form feed
\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\###	character represented by octal number ###
\\	backslash

Table 15.1: Backslashes understood by COPY

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.

11155
11156
11157
11158
11159
11160
11161
11162
11163
11164
11165
11166
11167
11168
11169
11170
11171
11172
11173
11174
11175
11176
11177
11178
11179
11180
11181
11182
11183
11184
11185
11186
11187
11188
11189
11190
11191
11192
11193
11194
11195
11196
11197
11198
11199
11200
11201
11202
11203
11204
11205
11206
11207
11208
11209
11210
11211
11212
11213
11214
11215
11216
11217
11218
11219
11220

11221
11222
11223
11224
11225
11226
11227
11228
11229
11230
11231
11232
11233
11234
11235
11236
11237
11238
11239
11240
11241
11242
11243
11244
11245
11246
11247
11248
11249
11250
11251
11252
11253
11254
11255
11256
11257
11258
11259
11260
11261
11262
11263
11264
11265
11266
11267
11268
11269
11270
11271
11272
11273
11274
11275
11276
11277
11278
11279
11280
11281
11282
11283
11284
11285
11286

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

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

Table 16.1: `psql` query buffer commands

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.

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`.

Operation	Command
Connect to another database	<code>\connect <i>dbname</i></code>
Copy tablefile to/from database	<code>\copy <i>tablename</i> to from <i>filename</i></code>
Set a variable	<code>\set <i>variable</i> or \set <i>variable value</i></code>
Unset a variable	<code>\unset <i>variable</i></code>
Set output format	<code>\pset <i>option</i> or \pset <i>option value</i></code>
Echo	<code>\echo <i>string</i> or \echo <i>command</i></code>
Echo to \o output	<code>\qecho <i>string</i> or \qecho <i>command</i></code>
Copyright	<code>\copyright</code>
Change character encoding	<code>\encoding <i>newencoding</i></code>

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

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

Table 16.3: psql \pset options

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 preceding 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 preceding 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

```

11353
11354
11355
11356
11357
11358
11359      test=> SELECT NULL;
11360      ?column?
11361      -----
11362
11363
11364      (1 row)
11365
11366
11367      test=> \set tuples_only
11368      Showing only tuples.
11369      test=> SELECT NULL;
11370
11371
11372
11373
11374      test=> \set null '(null)'
11375      Null display is "(null)".
11376      test=> SELECT NULL;
11377
11378      (null)
11379
11380
11381

```

Figure 16.1: Example of `\set`

```

11382
11383
11384
11385
11386
11387
11388
11389
11390
11391
11392
11393
11394
11395
11396
11397
11398
11399
11400
11401
11402
11403
11404
11405
11406
11407
11408
11409
11410
11411
11412
11413
11414
11415
11416
11417
11418

```

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

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

11419
11420
11421
11422
11423
11424
11425
11426
11427
11428
11429
11430
11431
11432
11433
11434
11435
11436
11437
11438
11439
11440
11441
11442
11443
11444
11445
11446
11447
11448
11449
11450
11451
11452
11453
11454
11455
11456
11457
11458
11459
11460
11461
11462
11463
11464
11465
11466
11467
11468
11469
11470
11471
11472
11473
11474
11475
11476
11477
11478
11479
11480
11481
11482
11483
11484

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

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

Table 16.5: psql predefined variables

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.

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

Table 16.6: psql listing commands

Large Objects	Command	Argument
Import	\lo_import	<i>file</i>
Export	\lo_export	<i>oid file</i>
Unlink	\lo_unlink	<i>oid</i>
List	\lo_list	

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.

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

Table 16.8: psql command-line arguments

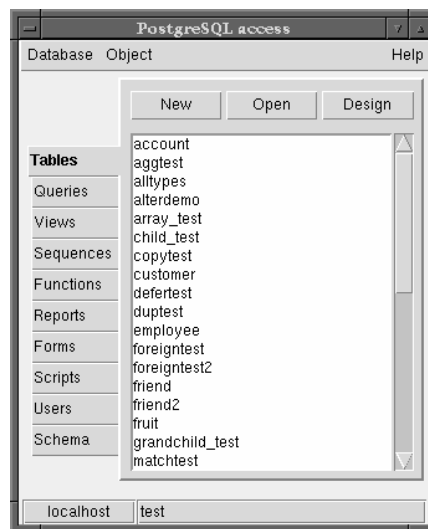


Figure 16.3: Pgaccess opening window

Sort field	Filter conditions	Reload	Close	
firstname	lastname	city	state	
Mike	Nichols	Tampa	FL	20
Mark	Middleton	Indianapolis	IN	
Jack	Burger			27
*	*	*	*	*

Figure 16.4: Pgaccess table window

16.3 Summary

This chapter covered `psql` and `pgaccess`. These are the most popular PostgreSQL query tools.

11683
11684
11685
11686
11687
11688
11689
11690
11691
11692
11693
11694
11695
11696
11697
11698
11699
11700
11701
11702
11703
11704
11705
11706
11707
11708
11709
11710
11711
11712
11713
11714
11715
11716
11717
11718
11719
11720
11721
11722
11723
11724
11725
11726
11727
11728
11729
11730
11731
11732
11733
11734
11735
11736
11737
11738
11739
11740
11741
11742
11743
11744
11745
11746
11747
11748

11749
11750
11751
11752
11753
11754
11755
11756
11757
11758
11759
11760
11761
11762
11763
11764
11765
11766
11767
11768
11769
11770
11771
11772
11773
11774
11775
11776
11777
11778
11779
11780
11781
11782
11783
11784
11785
11786
11787
11788
11789
11790
11791
11792
11793
11794
11795
11796
11797
11798
11799
11800
11801
11802
11803
11804
11805
11806
11807
11808
11809
11810
11811
11812
11813
11814

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

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

Table 17.1: Interface summary

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,
test(>
          name CHAR(30)
test(> );
CREATE
test=> INSERT INTO statename VALUES ('AL', 'Alabama');
INSERT 18934 1
test=> INSERT INTO statename VALUES ('AK', 'Alaska');
INSERT 18934 1
...

```

11815
11816
11817
11818
11819
11820
11821
11822
11823
11824
11825
11826
11827
11828
11829
11830
11831
11832
11833
11834
11835
11836
11837
11838
11839
11840
11841
11842
11843
11844
11845
11846
11847
11848
11849
11850
11851
11852
11853
11854
11855
11856
11857
11858
11859
11860
11861
11862
11863
11864
11865
11866
11867
11868
11869
11870
11871
11872
11873
11874
11875
11876
11877
11878
11879
11880

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

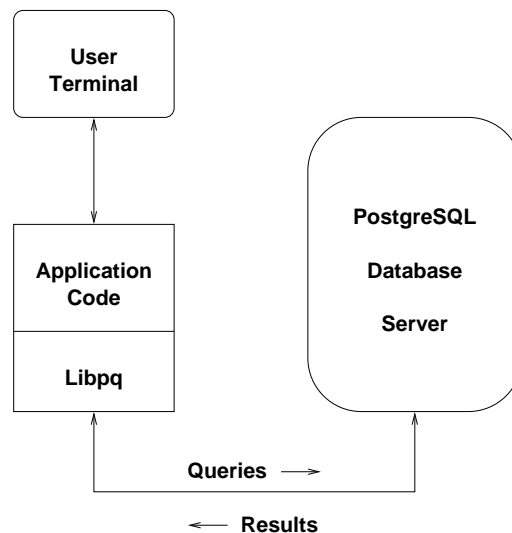


Figure 17.3: Libpq data flow

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*

11830
11831
11832
11833
11834
11835
11836
11837
11838
11839
11840
11841
11842
11843
11844
11845
11846
11847
11848
11849
11850
11851
11852
11853
11854
11855
11856
11857
11858
11859
11860
11861
11862
11863
11864
11865
11866
11867
11868
11869
11870
11871
11872
11873
11874
11875
11876
11877
11878
11879
11880

```

11881     /*
11882     * libpq sample program
11883     */
11884
11885     #include <stdio.h>
11886     #include <stdlib.h>
11887     #include "libpq-fe.h"           /* libpq header file */
11888
11889     int
11890     main()
11891     {
11892         char        state_code[3];           /* holds state code entered by user */
11893         char        query_string[256];      /* holds constructed SQL query */
11894         PGconn      *conn;                  /* holds database connection */
11895         PGresult    *res;                  /* holds query result */
11896         int         i;
11897
11898         conn = PQconnectdb("dbname=test");  /* connect to the database */
11899
11900         if (PQstatus(conn) == CONNECTION_BAD) /* did the database connection fail? */
11901         {
11902             fprintf(stderr, "Connection to database failed.\n");
11903             fprintf(stderr, "%s", PQerrorMessage(conn));
11904             exit(1);
11905         }
11906
11907         printf("Enter a state code: ");      /* prompt user for a state code */
11908         scanf("%2s", state_code);
11909
11910         sprintf(query_string,                /* create an SQL query string */
11911                 "SELECT name \
11912                  FROM statename \
11913                  WHERE code = '%s'", state_code);
11914
11915         res = PQexec(conn, query_string);    /* send the query */
11916
11917         if (PQresultStatus(res) != PGRES_TUPLES_OK) /* did the query fail? */
11918         {
11919             fprintf(stderr, "SELECT query failed.\n");
11920             PQclear(res);
11921             PQfinish(conn);
11922             exit(1);
11923         }
11924
11925         for (i = 0; i < PQntuples(res); i++) /* loop through all rows returned */
11926             printf("%s\n", PQgetvalue(res, i, 0)); /* print the value returned */
11927
11928         PQclear(res);                        /* free result */
11929
11930         PQfinish(conn);                      /* disconnect from the database */
11931
11932         return 0;
11933     }
11934
11935
11936
11937
11938
11939
11940
11941
11942
11943
11944
11945
11946

```

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*.

11947
11948
11949
11950
11951
11952
11953
11954
11955
11956
11957
11958
11959
11960
11961
11962
11963
11964
11965
11966
11967
11968
11969
11970
11971
11972
11973
11974
11975
11976
11977
11978
11979
11980
11981
11982
11983
11984
11985
11986
11987
11988
11989
11990
11991
11992
11993
11994
11995
11996
11997
11998
11999
12000
12001
12002
12003
12004
12005
12006
12007
12008
12009
12010
12011
12012

```

12013
12014
12015
12016
12017
12018
12019
12020
12021
12022
12023      /*
12024      *  libpgeasy sample program
12025      */
12026
12027
12028      #include <stdio.h>
12029      #include <libpq-fe.h>
12030      #include <libpgeasy.h>                                /* libpgeasy header file */
12031
12032      int
12033      main()
12034      {
12035
12036          char      state_code[3];                          /* holds state code entered by user */
12037          char      query_string[256];                      /* holds constructed SQL query */
12038          char      state_name[31];                        /* holds returned state name */
12039
12040          connectdb("dbname=test");                          /* connect to the database */
12041
12042          printf("Enter a state code: ");                    /* prompt user for a state code */
12043          scanf("%2s", state_code);
12044
12045          sprintf(query_string,                              /* create an SQL query string */
12046                  "SELECT name \
12047                   FROM statename \
12048                   WHERE code = '%s'", state_code);
12049
12050          doquery(query_string);                             /* send the query */
12051
12052          while (fetch(state_name) != END_OF_TUPLES)        /* loop through all rows returned */
12053              printf("%s\n", state_name);                  /* print the value returned */
12054
12055          disconnectdb();                                    /* disconnect from the database */
12056
12057          return 0;
12058      }
12059
12060
12061
12062
12063
12064
12065
12066
12067
12068
12069
12070
12071
12072
12073
12074
12075
12076
12077
12078

```

Figure 17.5: *libpgeasy* sample program

```

/*
 * ecpg sample program
 */
12079
12080
12081
12082
#include <stdio.h>
12083
12084
EXEC SQL INCLUDE sqlca;
12085
12086
12087
12088
EXEC SQL WHENEVER SQLERROR sqlprint;
12089
12090
int
12091
main()
12092
{
12093
EXEC SQL BEGIN DECLARE SECTION;
12094
char state_code[3];
12095
12096
char *state_name = NULL;
12097
12098
char query_string[256];
12099
EXEC SQL END DECLARE SECTION;
12100
12101
EXEC SQL CONNECT TO test;
12102
12103
12104
12105
12106
12107
12108
12109
12110
12111
12112
12113
12114
12115
12116
12117
12118
12119
12120
12121
12122
12123
12124
12125
12126
12127
12128
12129
12130
12131
12132
12133
12134
12135
12136
12137
12138
12139
12140
12141
12142
12143
12144
/* holds state code entered by user */
/* holds value returned by query */
/* holds constructed SQL query */
/* connect to the database */
/* prompt user for a state code */
scanf("%2s", state_code);
/* 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


```

12145
12146
12147
12148
12149
12150
12151
12152      /*
12153      *   libpq++ sample program
12154      */
12155
12156      #include <iostream.h>
12157      #include <libpq++.h>                                // libpq++ header file
12158
12159      int main()
12160      {
12161          char      state_code[3];                        // holds state code entered by user
12162          char      query_string[256];                   // holds constructed SQL query
12163          PgDatabase data("dbname=test");                // connects to the database
12164
12165          if ( data.ConnectionBad() )                    // did the database connection fail?
12166          {
12167              cerr << "Connection to database failed." << endl
12168                  << "Error returned: " << data.ErrorMessage() << endl;
12169              exit(1);
12170          }
12171
12172          cout << "Enter a state code: ";                // prompt user for a state code
12173          cin.get(state_code, 3, '\n');
12174
12175          sprintf(query_string,                           // create an SQL query string
12176                 "SELECT name \
12177                 FROM statename \
12178                 WHERE code = '%s'", state_code);
12179
12180          if ( !data.ExecTuplesOk(query_string) )        // send the query
12181          {
12182              cerr << "SELECT query failed." << endl;
12183              exit(1);
12184          }
12185
12186          for (int i=0; i < data.Tuples(); i++)          // loop through all rows returned
12187              cout << data.GetValue(i,0) << endl;        // print the value returned
12188
12189          return 0;
12190      }
12191
12192
12193
12194
12195
12196
12197
12198
12199
12200
12201
12202
12203
12204
12205
12206
12207
12208
12209
12210

```

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:

```
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 -lpgeasy
```

```
ecpg 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. `Ecpq` 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 `SELECT`s 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.

`Ecpq` 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.

12211
12212
12213
12214
12215
12216
12217
12218
12219
12220
12221
12222
12223
12224
12225
12226
12227
12228
12229
12230
12231
12232
12233
12234
12235
12236
12237
12238
12239
12240
12241
12242
12243
12244
12245
12246
12247
12248
12249
12250
12251
12252
12253
12254
12255
12256
12257
12258
12259
12260
12261
12262
12263
12264
12265
12266
12267
12268
12269
12270
12271
12272
12273
12274
12275
12276

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*.

12277
12278
12279
12280
12281
12282
12283
12284
12285
12286
12287
12288
12289
12290
12291
12292
12293
12294
12295
12296
12297
12298
12299
12300
12301
12302
12303
12304
12305
12306
12307
12308
12309
12310
12311
12312
12313
12314
12315
12316
12317
12318
12319
12320
12321
12322
12323
12324
12325
12326
12327
12328
12329
12330
12331
12332
12333
12334
12335
12336
12337
12338
12339
12340
12341
12342

```

/*
 * Java sample program
 */
import java.io.*;
import java.sql.*;

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

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

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

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

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

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

```

Figure 17.8: JAVA sample program

```

12409     #!/usr/local/bin/perl
12410     #
12411     # perl sample program
12412     #
12413
12414     use Pg;                               # load database routines
12415
12416     $conn = Pg::connectdb("dbname=test");   # connect to the database
12417                                           # did the database connection fail?
12418
12419     die $conn->errorMessage unless PGRES_CONNECTION_OK eq $conn->status;
12420
12421
12422     print "Enter a state code: ";          # prompt user for a state code
12423     $state_code = <STDIN>;
12424     chomp $state_code;
12425
12426     $result = $conn->exec(                  # send the query
12427         "SELECT name \
12428         FROM statename \
12429         WHERE code = '$state_code'");
12430                                           # did the query fail?
12431
12432     die $conn->errorMessage unless PGRES_TUPLES_OK eq $result->resultStatus;
12433
12434     while (@row = $result->fetchrow) {      # loop through all rows returned
12435         print @row, "\n";                 # print the value returned
12436     }
12437
12438
12439
12440
12441
12442

```

Figure 17.9: PERL sample program

```

12443     #!/usr/local/pgsql/bin/pgtclsh
12444     #
12445     # pgtclsh sample program
12446     #
12447
12448
12449     set conn [pg_connect test]              ;# connect to the database
12450
12451     puts -nonewline "Enter a state code: "  ;# prompt user for a state code
12452     flush stdout
12453     gets stdin state_code
12454                                           ;# send the query
12455
12456     set res [pg_exec $conn \
12457         "SELECT name \
12458         FROM statename \
12459         WHERE code = '$state_code'"]
12460
12461
12462     set ntups [pg_result $res -numTuples]
12463
12464     for {set i 0} {$i < $ntups} {incr i} {  ;# loop through all rows returned
12465         puts stdout [lindex [pg_result $res -getTuple $i] 0] ;# print the value returned
12466     }
12467
12468
12469     pg_disconnect $conn                    ;# disconnect from the database
12470
12471
12472
12473
12474

```

Figure 17.10: TCL sample program

```

12475  #! /usr/local/bin/python
12476  #
12477  #  python sample program
12478  #
12479
12480  import sys
12481
12482  from pg import DB                    # load database routines
12483
12484  conn = DB('test')                  # connect to the database
12485
12486  sys.stdout.write('Enter a state code: ') # prompt user for a state code
12487  state_code = sys.stdin.readline()
12488  state_code = state_code[:-1]
12489
12490
12491  for name in conn.query(              # send the query
12492      "SELECT name \
12493      FROM statename \
12494      WHERE code = '"+state_code+"'").getresult():
12495
12496      sys.stdout.write('%s\n' % name)   # print the value returned
12497
12498
12499
12500
12501
12502
12503
12504
12505
12506
12507
12508
12509
12510
12511
12512
12513
12514
12515
12516
12517
12518
12519
12520
12521
12522
12523
12524
12525
12526
12527
12528
12529
12530
12531
12532
12533
12534
12535
12536
12537
12538
12539
12540

```

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

```

12511  <!--
12512  -- PHP sample program -- Input
12513  -->
12514
12515  <HTML>
12516  <BODY>
12517
12518  <!-- prompt user for a state code -->
12519
12520  <FORM ACTION="<? echo $SCRIPT_NAME ?>/pg/sample2.phtml?state_code" method="POST">
12521  Client Number:
12522  <INPUT TYPE="text" name="state_code" value="<? echo $state_code ?>"
12523  maxlength=2 size=2>
12524
12525  <BR>
12526  <INPUT TYPE="submit" value="Continue">
12527
12528  </FORM>
12529  </BODY>
12530  </HTML>
12531
12532
12533
12534
12535
12536
12537
12538
12539
12540

```

Figure 17.12: PHP sample program — Input

performs 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>.

```
12541
12542
12543
12544
12545
12546
12547
12548
12549
12550
12551
12552
12553
12554      <!--
12555      -- PHP sample program -- Output
12556      -->
12557
12558      <HTML>
12559      <BODY>
12560      <?
12561          $database = pg_Connect("", "", "", "", "test"); # connect to the database
12562
12563          if (!$database)                                # did the database connection fail?
12564          {
12565              echo "Connection to database failed.";
12566              exit;
12567          }
12568
12569          $result = pg_Exec($database,                    # send the query
12570              "SELECT name " .
12571              "FROM statename " .
12572              "WHERE code = '$state_code'");
12573
12574          for ($i = 0; $i < pg_NumRows($result); $i++)  # loop through all rows returned
12575          {
12576              echo pg_Result($result,$i,0);            # print the value returned
12577              echo "<BR>";
12578          }
12579      ?>
12580      </BODY>
12581      </HTML>
12582
12583
12584
12585
12586
12587
12588
12589
12590
12591
12592
12593
12594
12595
12596
12597
12598
12599
12600
12601
12602
12603
12604
12605
12606
```

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 *pgtclsh*
- 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.

12607
12608
12609
12610
12611
12612
12613
12614
12615
12616
12617
12618
12619
12620
12621
12622
12623
12624
12625
12626
12627
12628
12629
12630
12631
12632
12633
12634
12635
12636
12637
12638
12639
12640
12641
12642
12643
12644
12645
12646
12647
12648
12649
12650
12651
12652
12653
12654
12655
12656
12657
12658
12659
12660
12661
12662
12663
12664
12665
12666
12667
12668
12669
12670
12671
12672

12673
12674
12675
12676
12677
12678
12679
12680
12681
12682
12683
12684
12685
12686
12687
12688
12689
12690
12691
12692
12693
12694
12695
12696
12697
12698
12699
12700
12701
12702
12703
12704
12705
12706
12707
12708
12709
12710
12711
12712
12713
12714
12715
12716
12717
12718
12719
12720
12721
12722
12723
12724
12725
12726
12727
12728
12729
12730
12731
12732
12733
12734
12735
12736
12737
12738

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

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

Figure 18.1: SQL *ftoc* function

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 SETOF. 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

12739
12740
12741
12742
12743
12744
12745
12746
12747
12748
12749
12750
12751
12752
12753
12754
12755
12756
12757
12758
12759
12760
12761
12762
12763
12764
12765
12766
12767
12768
12769
12770
12771
12772
12773
12774
12775
12776
12777
12778
12779
12780
12781
12782
12783
12784
12785
12786
12787
12788
12789
12790
12791
12792
12793
12794
12795
12796
12797
12798
12799
12800
12801
12802
12803
12804

```

12805 test=> CREATE FUNCTION tax(numeric)
12806 test-> RETURNS numeric
12807 test-> AS 'SELECT ($1 * 0.06::numeric(8,2))::numeric(8,2);'
12808 test-> LANGUAGE 'sql';
12809 CREATE
12810 test=> SELECT tax(100);
12811 tax
12812 -----
12813 6.00
12814 (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 `floc()`.

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`

```

12831 test=> CREATE TABLE part (
12832 test(>          part_id    INTEGER,
12833 test(>          name       CHAR(30),
12834 test(>          cost       NUMERIC(8,2),
12835 test(>          weight     FLOAT
12836 test(> );
12837 CREATE
12838 test=> INSERT INTO part VALUES (637, 'cable', 14.29, 5);
12839 INSERT 20867 1
12840 test=> INSERT INTO part VALUES (638, 'sticker', 0.84, 1);
12841 INSERT 20868 1
12842 test=> INSERT INTO part VALUES (639, 'bulb', 3.68, 3);
12843 INSERT 20869 1
12844 test=> SELECT part_id,
12845 test->          name,
12846 test->          cost,
12847 test->          tax(cost),
12848 test->          cost + tax(cost) AS total
12849 test-> FROM part
12850 test-> ORDER BY part_id;
12851 part_id |          name          | cost | tax | total
12852 -----+-----+-----+-----+-----
12853 637 | cable                  | 14.29 | 0.86 | 15.15
12854 638 | sticker                | 0.84 | 0.05 | 0.89
12855 639 | bulb                   | 3.68 | 0.22 | 3.90
12856 (3 rows)

```

Figure 18.3: Recreation of the `part` table

12870

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

```

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

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

```

Figure 18.4: SQL *shipping* function

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.

12871
12872
12873
12874
12875
12876
12877
12878
12879
12880
12881
12882
12883
12884
12885
12886
12887
12888
12889
12890
12891
12892
12893
12894
12895
12896
12897
12898
12899
12900
12901
12902
12903
12904
12905
12906
12907
12908
12909
12910
12911
12912
12913
12914
12915
12916
12917
12918
12919
12920
12921
12922
12923
12924
12925
12926
12927
12928
12929
12930
12931
12932
12933
12934
12935
12936

```

12937
12938
12939
12940 test=> CREATE FUNCTION getstatename(text)
12941 test-> RETURNS text
12942 test-> AS 'SELECT CAST(name AS TEXT)
12943 test->     FROM statename
12944 test->     WHERE code = $1;'
12945 test-> LANGUAGE 'sql';
12946
12947 CREATE
12948
12949 test=> SELECT getstatename('AL');
12950
12951     getstatename
12952 -----
12953 Alabama
12954 (1 row)

```

Figure 18.5: SQL function *getstatename*

```

12960
12961
12962
12963
12964
12965
12966
12967
12968 test=> SELECT customer.name, statename.name
12969 test-> FROM   customer, statename
12970 test-> WHERE  customer.state = statename.code
12971 test-> ORDER BY customer.name;
12972
12973     name          |          name
12974 -----+-----
12975 Fleeer Gearworks, Inc. | Alabama
12976 Mark Middleton         | Indiana
12977 Mike Nichols           | Florida
12978 (3 rows)

```

```

12982 test=> SELECT customer.name, getstatename(customer.state)
12983 test-> FROM   customer
12984 test-> ORDER BY customer.name;
12985
12986     name          |          getstatename
12987 -----+-----
12988 Fleeer Gearworks, Inc. | Alabama
12989 Mark Middleton         | Indiana
12990 Mike Nichols           | Florida
12991 (3 rows)

```

Figure 18.6: Getting state name using join and function

```

13000
13001
13002

```

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 *getstatername* from figure 18.5. The only differences are the addition of `BEGIN...END` and the language definition as PL/PGSQL.

```
test=> CREATE FUNCTION getstatername2(text)
test-> RETURNS text
test-> AS 'BEGIN
test'>     SELECT CAST(name AS TEXT)
test'>     FROM statername
test'>     WHERE code = $1;
test'>     END;'
test-> LANGUAGE 'plpgsql';
CREATE
```

Figure 18.7: PL/PGSQL version of *getstatername*

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 *statername* table. Function *change_statername* performs `INSERT`, `UPDATE`, and `DELETE` operations on the *statername* table. *Change_statername()* 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 *statername* was changed, and false (*f*) if the *statername* table was unmodified. Figure 18.12 shows examples of its use.

13003
13004
13005
13006
13007
13008
13009
13010
13011
13012
13013
13014
13015
13016
13017
13018
13019
13020
13021
13022
13023
13024
13025
13026
13027
13028
13029
13030
13031
13032
13033
13034
13035
13036
13037
13038
13039
13040
13041
13042
13043
13044
13045
13046
13047
13048
13049
13050
13051
13052
13053
13054
13055
13056
13057
13058
13059
13060
13061
13062
13063
13064
13065
13066
13067
13068

```

13069
13070
13071
13072
13073
13074
13075
13076
13077
13078
13079
13080
13081
13082      test=> CREATE FUNCTION spread(text)
13083      test-> RETURNS text
13084      test-> AS 'DECLARE
13085      test'>      str text;
13086      test'>      ret text;
13087      test'>      i integer;
13088      test'>      len integer;
13089      test'>
13090      test'>      BEGIN
13091      test'>          str := upper($1);
13092      test'>          ret := '';          -- start with zero length
13093      test'>          i := 1;
13094      test'>          len := length(str);
13095      test'>          WHILE i <= len LOOP
13096      test'>              ret := ret || substr(str, i, 1) || ' ';
13097      test'>              i := i + 1;
13098      test'>          END LOOP;
13099      test'>          RETURN ret;
13100      test'>      END;'
13101      test-> LANGUAGE 'plpgsql';
13102      CREATE
13103      test=> SELECT spread('Major Financial Report');
13104      spread
13105      -----
13106      M A J O R   F I N A N C I A L   R E P O R T
13107      (1 row)
13108
13109
13110
13111
13112
13113
13114
13115
13116
13117
13118
13119
13120
13121
13122
13123
13124
13125
13126
13127
13128
13129
13130
13131
13132
13133
13134

```

Figure 18.8: PL/PGSQL *spread* function

```

test=> CREATE FUNCTION getstatecode(text)                                13135
test-> RETURNS text                                                    13136
test-> AS 'DECLARE                                                       13137
test'>     state_str statename.name%TYPE;                               13138
test'>     statename_rec record;                                         13139
test'>     i          integer;                                           13140
test'>     len        integer;                                           13141
test'>     matches    record;                                             13142
test'>     search_str text;                                              13143
test'>     search_str text;                                              13144
test'>     search_str text;                                              13145
test'>     search_str text;                                              13146
test'>     search_str text;                                              13147
test'> BEGIN                                                              13148
test'>     state_str := initcap($1); -- capitalization match column      13149
test'>     len := length(trim($1));                                       13150
test'>     i := 2;                                                         13151
test'>     i := 2;                                                         13152
test'>     i := 2;                                                         13153
test'>     i := 2;                                                         13154
test'>     SELECT INTO statename_rec * -- first try for an exact match    13155
test'>     FROM   statename                                               13156
test'>     WHERE  name = state_str;                                         13157
test'>     IF FOUND                                                         13158
test'>     THEN   RETURN statename_rec.code;                               13159
test'>     END IF;                                                         13160
test'>     THEN   RETURN statename_rec.code;                               13161
test'>     END IF;                                                         13162
test'>     THEN   RETURN statename_rec.code;                               13163
test'>     THEN   RETURN statename_rec.code;                               13164
test'>     WHILE i <= len LOOP -- test 2,4,6,... chars for match          13165
test'>         search_str = trim(substr(state_str, 1, i)) || '%';         13166
test'>         SELECT INTO matches COUNT(*)                                13167
test'>         FROM   statename                                             13168
test'>         WHERE  name LIKE search_str;                                  13169
test'>         WHERE  name LIKE search_str;                                  13170
test'>         WHERE  name LIKE search_str;                                  13171
test'>         WHERE  name LIKE search_str;                                  13172
test'>         IF matches.count = 0 -- no matches, failure                 13173
test'>         THEN   RETURN NULL;                                         13174
test'>         END IF;                                                     13175
test'>         IF matches.count = 1 -- exactly one match, return it       13176
test'>         THEN   RETURN statename_rec.code;                               13177
test'>         THEN   RETURN statename_rec.code;                               13178
test'>         THEN   RETURN statename_rec.code;                               13179
test'>         SELECT INTO statename_rec *                                13180
test'>         FROM   statename                                             13181
test'>         WHERE  name LIKE search_str;                                  13182
test'>         IF FOUND                                                         13183
test'>         THEN   RETURN statename_rec.code;                               13184
test'>         THEN   RETURN statename_rec.code;                               13185
test'>         THEN   RETURN statename_rec.code;                               13186
test'>         THEN   RETURN statename_rec.code;                               13187
test'>         END IF;                                                     13188
test'>         i := i + 2; -- >1 match, try 2 more chars                 13189
test'>         i := i + 2; -- >1 match, try 2 more chars                 13190
test'>     END LOOP;                                                       13191
test'>     RETURN '' ;                                                    13192
test'> END;'                                                                13193
test-> LANGUAGE 'plpgsql';                                              13194
test-> LANGUAGE 'plpgsql';                                              13195
test-> LANGUAGE 'plpgsql';                                              13196
test-> LANGUAGE 'plpgsql';                                              13197
test-> LANGUAGE 'plpgsql';                                              13198
test-> LANGUAGE 'plpgsql';                                              13199
test-> LANGUAGE 'plpgsql';                                              13200

```

Figure 18.9: PL/PGSQL *getstatecode* function


```
13201
13202
13203
13204
13205
13206
13207
13208
13209
13210
13211
13212
13213
13214
13215
13216      test=> SELECT getstatecode('Alabama');
13217      getstatecode
13218      -----
13219      AL
13220      (1 row)
13221
13222
13223
13224      test=> SELECT getstatecode('ALAB');
13225      getstatecode
13226      -----
13227      AL
13228      (1 row)
13229
13230
13231
13232      test=> SELECT getstatecode('Al ');
13233      getstatecode
13234      -----
13235      AL
13236      (1 row)
13237
13238
13239
13240      test=> SELECT getstatecode('Al ');
13241      getstatecode
13242      -----
13243      AL
13244      (1 row)
13245
13246
13247
13248
13249
13250
13251
13252
13253
13254
13255
13256
13257
13258
13259
13260
13261
13262
13263
13264
13265
13266
```

Figure 18.10: Calls to *getstatecode* function

```

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

```

Figure 18.11: PL/PGSQL *change_statename* function

13267
13268
13269
13270
13271
13272
13273
13274
13275
13276
13277
13278
13279
13280
13281
13282
13283
13284
13285
13286
13287
13288
13289
13290
13291
13292
13293
13294
13295
13296
13297
13298
13299
13300
13301
13302
13303
13304
13305
13306
13307
13308
13309
13310
13311
13312
13313
13314
13315
13316
13317
13318
13319
13320
13321
13322
13323
13324
13325
13326
13327
13328
13329
13330
13331
13332

```

13333
13334
13335
13336
13337
13338      test=> DELETE FROM statename;
13339      DELETE 1
13340
13341      test=> SELECT change_statename('AL','Alabama');
13342      change_statename
13343      -----
13344      t
13345      (1 row)
13346
13347
13348
13349      test=> SELECT * FROM statename;
13350      code |          name
13351      -----+-----
13352      AL  | Alabama
13353      (1 row)
13354
13355
13356
13357      test=> SELECT change_statename('AL','Bermuda');
13358      change_statename
13359      -----
13360      t
13361      (1 row)
13362
13363
13364
13365      test=> SELECT * FROM statename;
13366      code |          name
13367      -----+-----
13368      AL  | Bermuda
13369      (1 row)
13370
13371
13372
13373      test=> SELECT change_statename('AL','');
13374      change_statename
13375      -----
13376      t
13377      (1 row)
13378
13379
13380
13381      test=> SELECT change_statename('AL','');          -- row was already deleted
13382      change_statename
13383      -----
13384      f
13385      (1 row)
13386
13387
13388
13389
13390
13391
13392
13393
13394
13395
13396
13397
13398

```

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 uppercased 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.

13399
13400
13401
13402
13403
13404
13405
13406
13407
13408
13409
13410
13411
13412
13413
13414
13415
13416
13417
13418
13419
13420
13421
13422
13423
13424
13425
13426
13427
13428
13429
13430
13431
13432
13433
13434
13435
13436
13437
13438
13439
13440
13441
13442
13443
13444
13445
13446
13447
13448
13449
13450
13451
13452
13453
13454
13455
13456
13457
13458
13459
13460
13461
13462
13463
13464

```

13465
13466
13467
13468 test=> CREATE FUNCTION trigger_insert_update_statename()
13469 test-> RETURNS opaque
13470 test-> AS 'BEGIN
13471 test'>
13472     IF new.code !~ '^[A-Za-z][A-Za-z]$'
13473 test'>     THEN RAISE EXCEPTION 'Code must be two alphabetic characters.';
13474 test'>     END IF;
13475 test'>
13476     IF new.name !~ '^[A-Za-z ]*$'
13477 test'>     THEN RAISE EXCEPTION 'Name must be only alphabetic characters.';
13478 test'>     END IF;
13479 test'>
13480     IF length(trim(new.name)) < 3
13481 test'>     THEN RAISE EXCEPTION 'Name must be longer than two characters.';
13482 test'>     END IF;
13483 test'>
13484     new.code = upper(new.code);           -- uppercase statename.code
13485 test'>     new.name = initcap(new.name);      -- capitalize statename.name
13486 test'>
13487     RETURN new;
13488 test'> END;'
13489 test-> LANGUAGE 'plpgsql';
13490 CREATE
13491
13492
13493 test=> CREATE TRIGGER trigger_statename
13494 test-> BEFORE INSERT OR UPDATE
13495 test-> ON statename
13496 test-> ON statename
13497 test-> FOR EACH ROW
13498 test-> EXECUTE PROCEDURE trigger_insert_update_statename();
13499 CREATE
13500
13501
13502 test=> DELETE FROM statename;
13503 DELETE 1
13504 test=> INSERT INTO statename VALUES ('a', 'alabama');
13505 ERROR: State code must be two alphabetic characters.
13506 test=> INSERT INTO statename VALUES ('al', 'alabama2');
13507 ERROR: State name must be only alphabetic characters.
13508 test=> INSERT INTO statename VALUES ('al', 'al');
13509 ERROR: State name must longer than two characters.
13510 test=> INSERT INTO statename VALUES ('al', 'alabama');
13511 INSERT 292898 1
13512 test=> SELECT * FROM statename;
13513
13514 code | name
13515 -----+-----
13516 AL   | Alabama
13517 (1 row)
13518
13519
13520
13521
13522
13523
13524
13525
13526
13527
13528
13529
13530

```

Figure 18.13: Trigger creation

13531
13532
13533
13534
13535
13536
13537
13538
13539
13540
13541
13542
13543
13544
13545
13546
13547
13548
13549
13550
13551
13552
13553
13554
13555
13556
13557
13558
13559
13560
13561
13562
13563
13564
13565
13566
13567
13568
13569
13570
13571
13572
13573
13574
13575
13576
13577
13578
13579
13580
13581
13582
13583
13584
13585
13586
13587
13588
13589
13590
13591
13592
13593
13594
13595
13596

13597
13598
13599
13600
13601
13602
13603
13604
13605
13606
13607
13608
13609
13610
13611
13612
13613
13614
13615
13616
13617
13618
13619
13620
13621
13622
13623
13624
13625
13626
13627
13628
13629
13630
13631
13632
13633
13634
13635
13636
13637
13638
13639
13640
13641
13642
13643
13644
13645
13646
13647
13648
13649
13650
13651
13652
13653
13654
13655
13656
13657
13658
13659
13660
13661
13662

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 `ftoc()` 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.

```

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

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

```

Figure 19.1: C *ctof* function

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*.¹ 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*.

```

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

```

Figure 19.2: Create function *ctof*

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.

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

13663
13664
13665
13666
13667
13668
13669
13670
13671
13672
13673
13674
13675
13676
13677
13678
13679
13680
13681
13682
13683
13684
13685
13686
13687
13688
13689
13690
13691
13692
13693
13694
13695
13696
13697
13698
13699
13700
13701
13702
13703
13704
13705
13706
13707
13708
13709
13710
13711
13712
13713
13714
13715
13716
13717
13718
13719
13720
13721
13722
13723
13724
13725
13726
13727
13728


```
13729      test=> SELECT ctof(20);
13730          ctof
13731      -----
13732             68
13733      (1 row)
```

Figure 19.3: Calling function *ctof*

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.

13729
13730
13731
13732
13733
13734
13735
13736
13737
13738
13739
13740
13741
13742
13743
13744
13745
13746
13747
13748
13749
13750
13751
13752
13753
13754
13755
13756
13757
13758
13759
13760
13761
13762
13763
13764
13765
13766
13767
13768
13769
13770
13771
13772
13773
13774
13775
13776
13777
13778
13779
13780
13781
13782
13783
13784
13785
13786
13787
13788
13789
13790
13791
13792
13793
13794

13795
13796
13797
13798
13799
13800
13801
13802
13803
13804
13805
13806
13807
13808
13809
13810
13811
13812
13813
13814
13815
13816
13817
13818
13819
13820
13821
13822
13823
13824
13825
13826
13827
13828
13829
13830
13831
13832
13833
13834
13835
13836
13837
13838
13839
13840
13841
13842
13843
13844
13845
13846
13847
13848
13849
13850
13851
13852
13853
13854
13855
13856
13857
13858
13859
13860

13861
13862
13863
13864
13865
13866
13867
13868
13869
13870
13871
13872
13873
13874
13875
13876
13877
13878
13879
13880
13881
13882
13883
13884
13885
13886
13887
13888
13889
13890
13891
13892
13893
13894
13895
13896
13897
13898
13899
13900
13901
13902
13903
13904
13905
13906
13907
13908
13909
13910
13911
13912
13913
13914
13915
13916
13917
13918
13919
13920
13921
13922
13923
13924
13925
13926

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

```

13927
13928
13929
13930
13931
13932
13933
13934
13935
13936
13937
13938
13939
13940
13941
13942
13943
13944
13945
13946
13947
13948
13949
13950
13951
13952
13953
13954
13955
13956
13957
13958
13959
13960
13961
13962
13963
13964
13965
13966
13967
13968
13969
13970
13971
13972
13973
13974
13975
13976
13977
13978
13979
13980
13981
13982
13983
13984
13985
13986
13987
13988
13989
13990
13991
13992

$ 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 reconnects 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 PostgreSQL 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

```

14013     $ createdb demodb1
14014     CREATE DATABASE
14015     $ psql test
14016     Welcome to psql, the PostgreSQL interactive terminal.
14017
14018
14019
14020     Type: \copyright for distribution terms
14021           \h for help with SQL commands
14022           \? for help on internal slash commands
14023           \g or terminate with semicolon to execute query
14024           \q to quit
14025
14026
14027
14028     test=> CREATE DATABASE demodb2;
14029     CREATE DATABASE
14030
14031     test=> DROP DATABASE demodb1;
14032     DROP DATABASE
14033
14034     test=> \connect demodb2
14035     You are now connected to database demodb2.
14036     demodb2=> \q
14037

```

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:

14059
14060
14061
14062
14063
14064
14065
14066
14067
14068
14069
14070
14071
14072
14073
14074
14075
14076
14077
14078
14079
14080
14081
14082
14083
14084
14085
14086
14087
14088
14089
14090
14091
14092
14093
14094
14095
14096
14097
14098
14099
14100
14101
14102
14103
14104
14105
14106
14107
14108
14109
14110
14111
14112
14113
14114
14115
14116
14117
14118
14119
14120
14121
14122
14123
14124

```

14125      host      all      192.168.34.0  255.255.255.255  crypt
14126      host      all      192.168.90.0   255.255.255.0   password
14127

```

force passwords of all users from host *192.168.34.0* and network *192.168.90.0*. *Crypt* encrypts passwords when sent, while *password* sends passwords over the network without encryption. The line:

```

14132      host      all      192.168.98.0  255.255.255.255  password finance
14133

```

is similar to the previous entries, except it uses the usernames/passwords stored in *finance* to authenticate users.

The lines:

```

14139      host      sales    192.168.7.12  255.255.255.255  ident
14140      host      sales    192.168.7.64  255.255.255.255  ident support
14141

```

uses *ident* on the remote machine to verify the users connecting to database *sales* from host *192.168.7.12* and *192.168.7.64*. The second entry uses the *support* username mapping in *data/pg_ident.conf*.

Connections are rejected from hosts and networks not appearing in the file. For more information, see the file *data/pg_hba.conf* and the *Administrator's Guide* mentioned in appendix [A.3](#).

For database client applications, the environment variables `PGHOST`, `PGPORT`, `PGUSER`, `PGPASSWORD`, `PGDATESTYLE`, `PGTZ`, `PGCLIENTENCODING`, and `PGDATABASE` are helpful for setting default connection parameters and options. The `POSTGRESQL` documentation has more information about these.

20.5 Backup and Restore

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

```

14173      $ pg_dump test > /tmp/test.dump
14174      $ createdb newtest
14175      CREATE DATABASE
14176      $ psql newtest < /tmp/test.dump
14177

```

Figure 20.3: Making a new copy of database test

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.

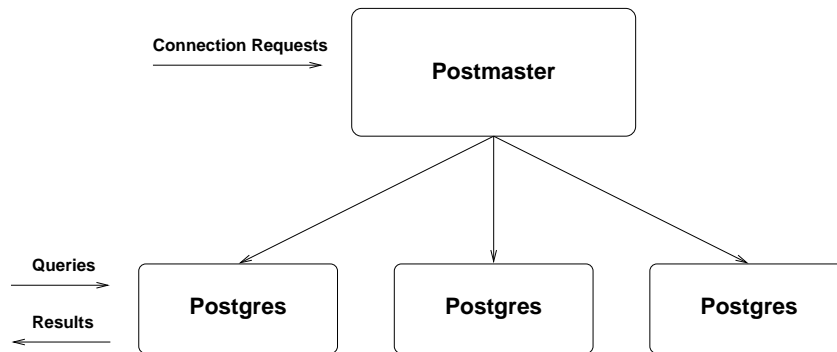


Figure 20.4: Postmaster and postgres processes

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

14191
14192
14193
14194
14195
14196
14197
14198
14199
14200
14201
14202
14203
14204
14205
14206
14207
14208
14209
14210
14211
14212
14213
14214
14215
14216
14217
14218
14219
14220
14221
14222
14223
14224
14225
14226
14227
14228
14229
14230
14231
14232
14233
14234
14235
14236
14237
14238
14239
14240
14241
14242
14243
14244
14245
14246
14247
14248
14249
14250
14251
14252
14253
14254
14255
14256

- 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.

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

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.

14323
14324
14325
14326
14327
14328
14329
14330
14331
14332
14333
14334
14335
14336
14337
14338
14339
14340
14341
14342
14343
14344
14345
14346
14347
14348
14349
14350
14351
14352
14353
14354
14355
14356
14357
14358
14359
14360
14361
14362
14363
14364
14365
14366
14367
14368
14369
14370
14371
14372
14373
14374
14375
14376
14377
14378
14379
14380
14381
14382
14383
14384
14385
14386
14387
14388

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.

14389
14390
14391
14392
14393
14394
14395
14396
14397
14398
14399
14400
14401
14402
14403
14404
14405
14406
14407
14408
14409
14410
14411
14412
14413
14414
14415
14416
14417
14418
14419
14420
14421
14422
14423
14424
14425
14426
14427
14428
14429
14430
14431
14432
14433
14434
14435
14436
14437
14438
14439
14440
14441
14442
14443
14444
14445
14446
14447
14448
14449
14450
14451
14452
14453
14454

14455
14456
14457
14458
14459
14460
14461
14462
14463
14464
14465
14466
14467
14468
14469
14470
14471
14472
14473
14474
14475
14476
14477
14478
14479
14480
14481
14482
14483
14484
14485
14486
14487
14488
14489
14490
14491
14492
14493
14494
14495
14496
14497
14498
14499
14500
14501
14502
14503
14504
14505
14506
14507
14508
14509
14510
14511
14512
14513
14514
14515
14516
14517
14518
14519
14520

14521
14522
14523
14524
14525
14526
14527
14528
14529
14530
14531
14532
14533
14534
14535
14536
14537
14538
14539
14540
14541
14542
14543
14544
14545
14546
14547
14548
14549
14550
14551
14552
14553
14554
14555
14556
14557
14558
14559
14560
14561
14562
14563
14564
14565
14566
14567
14568
14569
14570
14571
14572
14573
14574
14575
14576
14577
14578
14579
14580
14581
14582
14583
14584
14585
14586

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/mailling-list.html>.

A.3 Supplied Documentation

This information comes from <http://www.postgresql.org/docs/index.html>.

A.4 Commercial Support

Information from <http://www.pgsq1.com/> and <http://www.greatbridge.com/>.

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

14587
14588
14589
14590
14591
14592
14593
14594
14595
14596
14597
14598
14599
14600
14601
14602
14603
14604
14605
14606
14607
14608
14609
14610
14611
14612
14613
14614
14615
14616
14617
14618
14619
14620
14621
14622
14623
14624
14625
14626
14627
14628
14629
14630
14631
14632
14633
14634
14635
14636
14637
14638
14639
14640
14641
14642
14643
14644
14645
14646
14647
14648
14649
14650
14651
14652

14653
14654
14655
14656
14657
14658
14659
14660
14661
14662
14663
14664
14665
14666
14667
14668
14669
14670
14671
14672
14673
14674
14675
14676
14677
14678
14679
14680
14681
14682
14683
14684
14685
14686
14687
14688
14689
14690
14691
14692
14693
14694
14695
14696
14697
14698
14699
14700
14701
14702
14703
14704
14705
14706
14707
14708
14709
14710
14711
14712
14713
14714
14715
14716
14717
14718

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*.

Initialization

Initialization creates a database called *template1* in the PostgreSQL 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.

14719
14720
14721
14722
14723
14724
14725
14726
14727
14728
14729
14730
14731
14732
14733
14734
14735
14736
14737
14738
14739
14740
14741
14742
14743
14744
14745
14746
14747
14748
14749
14750
14751
14752
14753
14754
14755
14756
14757
14758
14759
14760
14761
14762
14763
14764
14765
14766
14767
14768
14769
14770
14771
14772
14773
14774
14775
14776
14777
14778
14779
14780
14781
14782
14783
14784

14785
14786
14787
14788
14789
14790
14791
14792
14793
14794
14795
14796
14797
14798
14799
14800
14801
14802
14803
14804
14805
14806
14807
14808
14809
14810
14811
14812
14813
14814
14815
14816
14817
14818
14819
14820
14821
14822
14823
14824
14825
14826
14827
14828
14829
14830
14831
14832
14833
14834
14835
14836
14837
14838
14839
14840
14841
14842
14843
14844
14845
14846
14847
14848
14849
14850

Appendix C

PostgreSQL Non-Standard Features by Chapter

This section outlines the non-standard features covered in this book:

Chapter 1 None.

Chapter 2 `psql` 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` s a unique feature of PostgreSQL.

Chapter 16 `psql` 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 implented differently in other database systems.	14851
	14852
Chapter 19 Using C to enhance the database is a unique POSTGRESQL feature.	14853
	14854
Chapter 20 The administrative utilities are unique to POSTGRESQL.	14855
	14856
	14857
	14858
	14859
	14860
	14861
	14862
	14863
	14864
	14865
	14866
	14867
	14868
	14869
	14870
	14871
	14872
	14873
	14874
	14875
	14876
	14877
	14878
	14879
	14880
	14881
	14882
	14883
	14884
	14885
	14886
	14887
	14888
	14889
	14890
	14891
	14892
	14893
	14894
	14895
	14896
	14897
	14898
	14899
	14900
	14901
	14902
	14903
	14904
	14905
	14906
	14907
	14908
	14909
	14910
	14911
	14912
	14913
	14914
	14915
	14916

14917
14918
14919
14920
14921
14922
14923
14924
14925
14926
14927
14928
14929
14930
14931
14932
14933
14934
14935
14936
14937
14938
14939
14940
14941
14942
14943
14944
14945
14946
14947
14948
14949
14950
14951
14952
14953
14954
14955
14956
14957
14958
14959
14960
14961
14962
14963
14964
14965
14966
14967
14968
14969
14970
14971
14972
14973
14974
14975
14976
14977
14978
14979
14980
14981
14982

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/>. They are in sgml/Docbook format. Approximately 200 pages.

14983
14984
14985
14986
14987
14988
14989
14990
14991
14992
14993
14994
14995
14996
14997
14998
14999
15000
15001
15002
15003
15004
15005
15006
15007
15008
15009
15010
15011
15012
15013
15014
15015
15016
15017
15018
15019
15020
15021
15022
15023
15024
15025
15026
15027
15028
15029
15030
15031
15032
15033
15034
15035
15036
15037
15038
15039
15040
15041
15042
15043
15044
15045
15046
15047
15048

15049
15050
15051
15052
15053
15054
15055
15056
15057
15058
15059
15060
15061
15062
15063
15064
15065
15066
15067
15068
15069
15070
15071
15072
15073
15074
15075
15076
15077
15078
15079
15080
15081
15082
15083
15084
15085
15086
15087
15088
15089
15090
15091
15092
15093
15094
15095
15096
15097
15098
15099
15100
15101
15102
15103
15104
15105
15106
15107
15108
15109
15110
15111
15112
15113
15114

Bibliography

[Bowman] Bowman et al., *The Practical SQL Handbook*, Addison–Wesley

[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

[Groff] Groff, James R. and Paul N. Weinberg *The Complete Reference SQL*, McGraw–Hill

[Hilton] Hilton, Craig and Jeff Willis *Building Database Applications on the Web Using PHP3*, Addison–Wesley

[User’s Guide] PostgreSQL User’s Guide, <http://www.postgresql.org/docs/user>

[Tutorial] PostgreSQL Tutorial, <http://www.postgresql.org/docs/tutorial>

[Administrator’s Guide] PostgreSQL Administrators Guide, <http://www.postgresql.org/docs/admin>

[Programmer’s Guide] PostgreSQL Programmer’s Guide, <http://www.postgresql.org/docs/programmer>

[Appendices] PostgreSQL Appendices, <http://www.postgresql.org/docs/postgres/part-appendix.htm>