PostgreSQL Tutorial

The PostgreSQL Development Team

Edited by

Thomas Lockhart

PostgreSQL Tutorial

by The PostgreSQL Development Team

Edited by Thomas Lockhart

Table of Contents

List of Figures

List of Examples

Summary

Postgres, developed originally in the UC Berkeley Computer Science Department, pioneered many of the object-relational concepts now becoming available in some commercial databases. It provides SQL92/SQL3 language support, transaction integrity, and type extensibility. PostgreSQL is a public-domain, open source descendant of this original Berkeley code.

Chapter 1. Introduction

This document is the user manual for the PostgreSQL (http://postgresql.org/) database management system, originally developed at the University of California at Berkeley. PostgreSQL is based on Postgres release 4.2 (http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/postgres.html). The Postgres project, led by Professor Michael Stonebraker, was sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), and ESL, Inc.

What is Postgres?

Traditional relational database management systems (DBMSs) support a data model consisting of a collection of named relations, containing attributes of a specific type. In current commercial systems, possible types include floating point numbers, integers, character strings, money, and dates. It is commonly recognized that this model is inadequate for future data processing applications. The relational model successfully replaced previous models in part because of its "Spartan simplicity". However, as mentioned, this simplicity often makes the implementation of certain applications very difficult. Postgres offers substantial additional power by incorporating the following four additional basic concepts in such a way that users can easily extend the system:

classes
inheritance
types
functions

Other features provide additional power and flexibility:

constraints
triggers
rules
transaction integrity

These features put Postgres into the category of databases referred to as object-relational. Note that this is distinct from those referred to as object-oriented, which in general are not as well suited to supporting the traditional relational database languages. So, although Postgres has some object-oriented features, it is firmly in the relational database world. In fact, some commercial databases have recently incorporated features pioneered by Postgres.

A Short History of Postgres

The Berkeley Postgres Project

Implementation of the Postgres DBMS began in 1986. The initial concepts for the system were presented in *The Design of Postgres* and the definition of the initial data model appeared in *The Postgres Data Model*. The design of the rule system at that time was described in *The Design of the Postgres Rules System*. The rationale and architecture of the storage manager were detailed in *The Postgres Storage System*.

Postgres has undergone several major releases since then. The first "demoware" system became operational in 1987 and was shown at the 1988 ACM-SIGMOD Conference. We released Version 1, described in *The Implementation of Postgres*, to a few external users in June 1989. In response to a critique of the first rule system (*A Commentary on the Postgres Rules System*), the rule system was redesigned (*On Rules, Procedures, Caching and Views in Database Systems*) and Version 2 was released in June 1990 with the new rule system. Version 3 appeared in 1991 and added support for multiple storage managers, an improved query executor, and a rewritten rewrite rule system. For the most part, releases since then have focused on portability and reliability.

Postgres has been used to implement many different research and production applications. These include: a financial data analysis system, a jet engine performance monitoring package, an asteroid tracking database, a medical information database, and several geographic information systems. Postgres has also been used as an educational tool at several universities. Finally, Illustra Information Technologies (http://www.illustra.com/) (since merged into Informix (http://www.informix.com/)) picked up the code and commercialized it. Postgres became the primary data manager for the Sequoia 2000 (http://www.sdsc.edu/0/Parts_Collabs/S2K/s2k_home.html) scientific computing project in late 1992. Furthermore, the size of the external user community nearly doubled during 1993. It became increasingly obvious that maintenance of the prototype code and support was taking up large amounts of time that should have been devoted to database research. In an effort to reduce this support burden, the project officially ended with Version 4.2.

Postgres95

In 1994, Andrew Yu (mailto:ayu@informix.com) and Jolly Chen (http://http.cs.berkeley.edu/~jolly/) added a SQL language interpreter to Postgres, and the code was subsequently released to the Web to find its own way in the world. Postgres95 was a public-domain, open source descendant of this original Berkeley code.

Postgres95 is a derivative of the last official release of Postgres (version 4.2). The code is now completely ANSI C and the code size has been trimmed by 25%. There are a lot of internal changes that improve performance and code maintainability. Postgres95 v1.0.x runs about 30-50% faster on the Wisconsin Benchmark compared to v4.2. Apart from bug fixes, these are the major enhancements:

The query language Postquel has been replaced with SQL (implemented in the server). We do not yet support subqueries (which can be imitated with user defined SQL functions). Aggregates have been re-implemented. We also added support for ‘‘GROUP BY’’. The libpq interface is still available for C programs.

In addition to the monitor program, we provide a new program (psql) which supports GNU readline.

We added a new front-end library, libpgtcl, that supports Tcl-based clients. A sample shell, pgtclsh, provides new Tcl commands to interface tcl programs with the Postgres95 backend.

The large object interface has been overhauled. We kept Inversion large objects as the only mechanism for storing large objects. (This is not to be confused with the Inversion file system which has been removed.)

The instance-level rule system has been removed. Rules are still available as rewrite rules.

A short tutorial introducing regular SQL features as well as those of ours is distributed with the source code.

GNU make (instead of BSD make) is used for the build. Also, Postgres95 can be compiled with an unpatched gcc (data alignment of doubles has been fixed).

PostgreSQL

By 1996, it became clear that the name Postgres95 would not stand the test of time. A new name, PostgreSQL, was chosen to reflect the relationship between original Postgres and the more recent versions with SQL capability. At the same time, the version numbering was reset to start at 6.0, putting the numbers back into the sequence originally begun by the Postgres Project.

The emphasis on development for the v1.0.x releases of Postgres95 was on stabilizing the backend code. With the v6.x series of PostgreSQL, the emphasis has shifted from identifying and understanding existing problems in the backend to augmenting features and capabilities, although work continues in all areas.

Major enhancements include:

Important backend features, including subselects, defaults, constraints, and triggers, have been implemented.

Additional SQL92-compliant language features have been added, including primary keys, quoted identifiers, literal string type coersion, type casting, and binary and hexadecimal integer input.

Built-in types have been improved, including new wide-range date/time types and additional geometric type support.

Overall backend code speed has been increased by approximately 20-40%, and backend startup time has decreased 80% since v6.0 was released.

About This Release

PostgreSQL is available without cost. This manual describes version 6.5 of PostgreSQL.

We will use Postgres to mean the version distributed as PostgreSQL.

Check the Administrator's Guide for a list of currently supported machines. In general, Postgres is portable to any Unix/Posix-compatible system with full libc library support.

Resources

This manual set is organized into several parts:

Tutorial

An introduction for new users. Does not cover advanced features.

User's Guide

General information for users, including available commands and data types.

Programmer's Guide

Advanced information for application programmers. Topics include type and function extensibility, library interfaces, and application design issues.

Administrator's Guide

Installation and management information. List of supported machines.

Developer's Guide

Information for Postgres developers. This is intended for those who are contributing to the Postgres project; application development information should appear in the Programmer's Guide. Currently included in the Programmer's Guide.

Reference Manual

Detailed reference information on command syntax. Currently included in the User's Guide.

In addition to this manual set, there are other resources to help you with Postgres installation and use:

man pages

The man pages have general information on command syntax.

FAQs

The Frequently Asked Questions (FAQ) documents address both general issues and some platform-specific issues.

READMEs

README files are available for some contributed packages.

Web Site

The Postgres (postgresql.org) web site has some information not appearing in the distribution. There is a mhonarc catalog of mailing list traffic which is a rich resource for many topics.

Mailing Lists

The Postgres Questions (mailto:questions@postgresql.org) mailing list is a good place to have user questions answered. Other mailing lists are available; consult the web page for details.

Yourself!

Postgres is an open source product. As such, it depends on the user community for ongoing support. As you begin to use Postgres, you will rely on others for help, either through the documentation or through the mailing lists. Consider contributing your knowledge back. If you learn something which is not in the documentation, write it up and contribute it. If you add features to the code, contribute it. Even those without a lot of experience can provide corrections and minor changes in the documentation, and that is a good way to start. The Postgres Documentation (mailto:docs@postgresql.org) mailing list is the place to get going.

Terminology

In the following documentation, site may be interpreted as the host machine on which Postgres is installed. Since it is possible to install more than one set of Postgres databases on a single host, this term more precisely denotes any particular set of installed Postgres binaries and databases.

The Postgres superuser is the user named postgres who owns the Postgres binaries and database files. As the database superuser, all protection mechanisms may be bypassed and any data accessed arbitrarily. In addition, the Postgres superuser is allowed to execute some support programs which are generally not available to all users. Note that the Postgres superuser is not the same as the Unix superuser (which will be referred to as root). The superuser should have a non-zero user identifier (UID) for security reasons.

The database administrator or DBA, is the person who is responsible for installing Postgres with mechanisms to enforce a security policy for a site. The DBA can add new users by the method described below and maintain a set of template databases for use by createdb.

The postmaster is the process that acts as a clearing-house for requests to the Postgres system. Frontend applications connect to the postmaster, which keeps tracks of any system errors and communication between the backend processes. The postmaster can take several command-line arguments to tune its behavior. However, supplying arguments is necessary only if you intend to run multiple sites or a non-default site.

The Postgres backend (the actual executable program postgres) may be executed directly from the user shell by the Postgres super-user (with the database name as an argument). However, doing this bypasses the shared buffer pool and lock table associated with a postmaster/site, therefore this is not recommended in a multiuser site.

Notation

... or /usr/local/pgsql/ at the front of a file name is used to represent the path to the Postgres superuser's home directory.

In a command synopsis, brackets ([and]) indicate an optional phrase or keyword. Anything in braces ({ and }) and containing vertical bars (|) indicates that you must choose one.

In examples, parentheses ((and)) are used to group boolean expressions. | is the boolean operator OR.

Examples will show commands executed from various accounts and programs. Commands executed from the root account will be preceeded with > . Commands executed from the Postgres superuser account will be preceeded with % , while commands executed from an unprivileged user's account will be preceeded with $. SQL commands will be preceeded with => or will have no leading prompt, depending on the context.

> Note: At the time of writing (Postgres v6.5) the notation for flagging commands is not universally consistant throughout the documentation set. Please report problems to the Documentation Mailing List (mailto:docs@postgresql.org).

Y2K Statement

> Author: Written by Thomas Lockhart (mailto:lockhart@alumni.caltech.edu) on 1998-10-22.

The PostgreSQL Global Development Team provides the Postgres software code tree as a public service, without warranty and without liability for it's behavior or performance. However, at the time of writing:

- The author of this statement, a volunteer on the Postgres support team since November, 1996, is not aware of any problems in the Postgres code base related to time transitions around Jan 1, 2000 (Y2K).
- The author of this statement is not aware of any reports of Y2K problems uncovered in regression testing or in other field use of recent or current versions of Postgres. We might have expected to hear about problems if they existed, given the installed base and the active participation of users on the support mailing lists.
- To the best of the author's knowledge, the assumptions Postgres makes about dates specified with a two-digit year are documented in the current User's Guide (http://www.postgresql.org/docs/user/datatype.htm) in the chapter on data types. For two-digit years, the significant transition year is 1970, not 2000; e.g. 70-01-01 is interpreted as 1970-01-01 , whereas 69-01-01 is interpreted as 2069-01-01 .
- Any Y2K problems in the underlying OS related to obtaining "the current time" may propagate into apparent Y2K problems in Postgres.

Refer to The Gnu Project (http://www.gnu.org/software/year2000.html) and The Perl Institute (http://language.perl.com/news/y2k.html) for further discussion of Y2K issues, particularly as it relates to open source, no fee software.

Copyrights and Trademarks

Chapter 2. SQL

This chapter originally appeared as a part of Stefan Simkovics' Master's Thesis (*Simkovics, 1998*).

SQL has become the most popular relational query language. The name SQL is an abbreviation for Structured Query Language. In 1974 Donald Chamberlin and others defined the language SEQUEL (Structured English Query Language) at IBM Research. This language was first implemented in an IBM prototype called SEQUEL-XRM in 1974-75. In 1976-77 a revised version of SEQUEL called SEQUEL/2 was defined and the name was changed to SQL subsequently.

A new prototype called System R was developed by IBM in 1977. System R implemented a large subset of SEQUEL/2 (now SQL) and a number of changes were made to SQL during the project. System R was installed in a number of user sites, both internal IBM sites and also some selected customer sites. Thanks to the success and acceptance of System R at those user sites IBM started to develop commercial products that implemented the SQL language based on the System R technology.

Over the next years IBM and also a number of other vendors announced SQL products such as SQL/DS (IBM), DB2 (IBM), ORACLE (Oracle Corp.), DG/SQL (Data General Corp.), and SYBASE (Sybase Inc.).

SQL is also an official standard now. In 1982 the American National Standards Institute (ANSI) chartered its Database Committee X3H2 to develop a proposal for a standard relational language. This proposal was ratified in 1986 and consisted essentially of the IBM dialect of SQL. In 1987 this ANSI standard was also accepted as an international standard by the International Organization for Standardization (ISO). This original standard version of SQL is often referred to, informally, as "SQL/86". In 1989 the original standard was extended and this new standard is often, again informally, referred to as "SQL/89". Also in 1989, a related standard called Database Language Embedded SQL (ESQL) was developed.

The ISO and ANSI committees have been working for many years on the definition of a greatly expanded version of the original standard, referred to informally as SQL2 or SQL/92. This version became a ratified standard - "International Standard ISO/IEC 9075:1992, Database Language SQL" - in late 1992. SQL/92 is the version normally meant when people refer to "the SQL standard". A detailed description of SQL/92 is given in *Date and Darwen, 1997*. At the time of writing this document a new standard informally referred to as SQL3 is under development. It is planned to make SQL a Turing-complete language, i.e. all computable queries (e.g. recursive queries) will be possible. This is a very complex task and therefore the completion of the new standard can not be expected before 1999.

The Relational Data Model

As mentioned before, SQL is a relational language. That means it is based on the relational data model first published by E.F. Codd in 1970. We will give a formal description of the relational model later (in *Relational Data Model Formalities*) but first we want to have a look at it from a more intuitive point of view.

A relational database is a database that is perceived by its users as a collection of tables (and nothing else but tables). A table consists of rows and columns where each row represents a record and each column represents an attribute of the records contained in the table. *The Suppliers and Parts Database* shows an example of a database consisting of three tables:

- SUPPLIER is a table storing the number (SNO), the name (SNAME) and the city (CITY) of a supplier.
- PART is a table storing the number (PNO) the name (PNAME) and the price (PRICE) of a part.
- SELLS stores information about which part (PNO) is sold by which supplier (SNO). It serves in a sense to connect the other two tables together.

Example 2-1. The Suppliers and Parts Database

```
SUPPLIER   SNO |  SNAME  |  CITY      SELLS   SNO | PNO
          -----+---------+--------           -----+-----
            1  |  Smith  | London              1  |  1
            2  |  Jones  | Paris               1  |  2
            3  |  Adams  | Vienna              2  |  4
            4  |  Blake  | Rome                3  |  1
                                               3  |  3
                                               4  |  2
PART       PNO |  PNAME  |  PRICE              4  |  3
          -----+---------+---------            4  |  4
            1  |  Screw  |   10
            2  |  Nut    |    8
            3  |  Bolt   |   15
            4  |  Cam    |   25
```

The tables PART and SUPPLIER may be regarded as entities and SELLS may be regarded as a relationship between a particular part and a particular supplier.

As we will see later, SQL operates on tables like the ones just defined but before that we will study the theory of the relational model.

Relational Data Model Formalities

The mathematical concept underlying the relational model is the set-theoretic relation which is a subset of the Cartesian product of a list of domains. This set-theoretic relation gives the model its name (do not confuse it with the relationship from the Entity-Relationship model). Formally a domain is simply a set of values. For example the set of integers is a domain. Also the set of character strings of length 20 and the real numbers are examples of domains.

The Cartesian product of domains D1, D2, ... Dk, written D1 × D2 × ... × Dk is the set of all k-tuples v1, v2, ... vk, such that v1 $\in$ D1, v1 $\in$ D1, ... vk $\in$ Dk.

For example, when we have k=2, D1={0,1} and D2={a,b,c} then D1 × D2 is {(0,a),(0,b),(0,c),(1,a),(1,b),(1,c)}.

A Relation is any subset of the Cartesian product of one or more domains: R $\subseteq$ D1 × D2 × ... × Dk.

For example {(0,a),(0,b),(1,a)} is a relation; it is in fact a subset of D1 × D2 mentioned above.

The members of a relation are called tuples. Each relation of some Cartesian product D1 × D2 × ... × Dk is said to have arity k and is therefore a set of k-tuples.

A relation can be viewed as a table (as we already did, remember *The Suppliers and Parts Database* where every tuple is represented by a row and every column corresponds to one component of a tuple. Giving names (called attributes) to the columns leads to the definition of a relation scheme.

A relation scheme R is a finite set of attributes A1, A2, ... Ak. There is a domain Di, for each attribute Ai, 1 <= i <= k, where the values of the attributes are taken from. We often write a relation scheme as R(A1, A2, ... Ak).

> Note: A relation scheme is just a kind of template whereas a relation is an instance of a relation scheme. The relation consists of tuples (and can therefore be viewed as a table); not so the relation scheme.

Domains vs. Data Types

We often talked about domains in the last section. Recall that a domain is, formally, just a set of values (e.g., the set of integers or the real numbers). In terms of database systems we often talk of data types instead of domains. When we define a table we have to make a decision about which attributes to include. Additionally we have to decide which kind of data is going to be stored as attribute values. For example the values of SNAME from the table SUPPLIER will be character strings, whereas SNO will store integers. We define this by assigning a data type to each attribute. The type of SNAME will be VARCHAR(20) (this is the SQL type for character strings of length <= 20), the type of SNO will be INTEGER. With the assignment of a data type we also have selected a domain for an attribute. The domain of SNAME is the set of all character strings of length <= 20, the domain of SNO is the set of all integer numbers.

Operations in the Relational Data Model

In the previous section (*Relational Data Model Formalities*) we defined the mathematical notion of the relational model. Now we know how the data can be stored using a relational data model but we do not know what to do with all these tables to retrieve something from the database yet. For example somebody could ask for the names of all suppliers that sell the part 'Screw'. Therefore two rather different kinds of notations for expressing operations on relations have been defined:

- The Relational Algebra which is an algebraic notation, where queries are expressed by applying specialized operators to the relations.
- The Relational Calculus which is a logical notation, where queries are expressed by formulating some logical restrictions that the tuples in the answer must satisfy.

Relational Algebra

The Relational Algebra was introduced by E. F. Codd in 1972. It consists of a set of operations on relations:

SELECT (ó): extracts tuples from a relation that satisfy a given restriction. Let R be a table that contains an attribute A. óA=a(R) = {t ∈ R ? t(A) = a} where t denotes a tuple of R and t(A) denotes the value of attribute A of tuple t.

PROJECT (ð): extracts specified attributes (columns) from a relation. Let R be a relation that contains an attribute X. ðX(R) = {t(X) ? t ∈ R}, where t(X) denotes the value of attribute X of tuple t.

PRODUCT (×): builds the Cartesian product of two relations. Let R be a table with arity k1 and let S be a table with arity k2. R × S is the set of all k1 + k2-tuples whose first k1 components form a tuple in R and whose last k2 components form a tuple in S.

UNION (∪): builds the set-theoretic union of two tables. Given the tables R and S (both must have the same arity), the union R ∪ S is the set of tuples that are in R or S or both.

INTERSECT (∩): builds the set-theoretic intersection of two tables. Given the tables R and S, R ∪ S is the set of tuples that are in R and in S. We again require that R and S have the same arity.

DIFFERENCE (− or ?): builds the set difference of two tables. Let R and S again be two tables with the same arity. R - S is the set of tuples in R but not in S.

JOIN (∏): connects two tables by their common attributes. Let R be a table with the attributes A,B and C and let S be a table with the attributes C,D and E. There is one attribute common to both relations, the attribute C. R ∏ S = ðR.A,R.B,R.C,S.D,S.E(óR.C=S.C(R × S)). What are we doing here? We first calculate the Cartesian product R × S. Then we select those tuples whose values for the common attribute C are equal (óR.C = S.C). Now we have a table that contains the attribute C two times and we correct this by projecting out the duplicate column.

Example 2-2. An Inner Join

Let's have a look at the tables that are produced by evaluating the steps necessary for a join. Let the following two tables be given:

```
R  A | B | C     S  C | D | E
  ---+---+---      ---+---+---
   1 | 2 | 3        3 | a | b
   4 | 5 | 6        6 | c | d
   7 | 8 | 9
```

First we calculate the Cartesian product R × S and get:

```
R x S   A | B | R.C | S.C | D | E
       ---+---+-----+-----+---+---
        1 | 2 |  3  |  3  | a | b
        1 | 2 |  3  |  6  | c | d
        4 | 5 |  6  |  3  | a | b
        4 | 5 |  6  |  6  | c | d
        7 | 8 |  9  |  3  | a | b
        7 | 8 |  9  |  6  | c | d
```

After the selection óR.C=S.C(R × S) we get:

```
        A | B | R.C | S.C | D | E
       ---+---+-----+-----+---+---
```

1	2	3	3	a	b
4	5	6	6	c	d

To remove the duplicate column S.C we project it out by the following operation: ðR.A,R.B,R.C,S.D,S.E(óR.C=S.C(R × S)) and get:

A	B	C	D	E
1	2	3	a	b
4	5	6	c	d

DIVIDE (÷): Let R be a table with the attributes A, B, C, and D and let S be a table with the attributes C and D. Then we define the division as: R ÷ S = {t ? ∀ ts ∈ S ∃ tr ∈ R such that tr(A,B)=t∧tr(C,D)=ts} where tr(x,y) denotes a tuple of table R that consists only of the components x and y. Note that the tuple t only consists of the components A and B of relation R.

Given the following tables

R

A	B	C	D
a	b	c	d
a	b	e	f
b	c	e	f
e	d	c	d
e	d	e	f
a	b	d	e

S

C	D
c	d
e	f

R ÷ S is derived as

A	B
a	b
e	d

For a more detailed description and definition of the relational algebra refer to [*Ullman, 1988*] or [*Date, 1994*].

Example 2-3. A Query Using Relational Algebra

Recall that we formulated all those relational operators to be able to retrieve data from the database. Let's return to our example from the previous section (*Operations in the Relational Data Model*) where someone wanted to know the names of all suppliers that sell the part Screw. This question can be answered using relational algebra by the following operation: ðSUPPLIER.SNAME(óPART.PNAME='Screw'(SUPPLIER ∏ SELLS ∏ PART))

We call such an operation a query. If we evaluate the above query against the our example tables (*The Suppliers and Parts Database*) we will obtain the following result:

```
 SNAME
-------
 Smith
 Adams
```

Relational Calculus

The relational calculus is based on the first order logic. There are two variants of the relational calculus:

- The Domain Relational Calculus (DRC), where variables stand for components (attributes) of the tuples.
- The Tuple Relational Calculus (TRC), where variables stand for tuples.

We want to discuss the tuple relational calculus only because it is the one underlying the most relational languages. For a detailed discussion on DRC (and also TRC) see [*Date, 1994*] or [*Ullman, 1988*].

Tuple Relational Calculus

The queries used in TRC are of the following form: x(A) ? F(x) where x is a tuple variable A is a set of attributes and F is a formula. The resulting relation consists of all tuples t(A) that satisfy F(t).

If we want to answer the question from example *A Query Using Relational Algebra* using TRC we formulate the following query:

$$
\begin{array}{l}
\{x(\mathrm{SNAME})\ ?\ x \in \mathrm{SUPPLIER} \wedge \backslash\mathrm{nonumber} \\
\qquad \exists\, y \in \mathrm{SELLS}\ \exists\, z \in \mathrm{PART}\ (y(\mathrm{SNO})=x(\mathrm{SNO}) \wedge \backslash\mathrm{nonumber} \\
\qquad z(\mathrm{PNO})=y(\mathrm{PNO}) \wedge \backslash\mathrm{nonumber} \\
\qquad z(\mathrm{PNAME})=\text{'Screw'})\} \ \backslash\mathrm{nonumber}
\end{array}
$$

Evaluating the query against the tables from *The Suppliers and Parts Database* again leads to the same result as in *A Query Using Relational Algebra*.

Relational Algebra vs. Relational Calculus

The relational algebra and the relational calculus have the same expressive power; i.e. all queries that can be formulated using relational algebra can also be formulated using the relational calculus and vice versa. This was first proved by E. F. Codd in 1972. This proof is based on an algorithm (Codd's reduction algorithm) by which an arbitrary expression of the relational calculus can be reduced to a semantically equivalent expression of relational algebra. For a more detailed discussion on that refer to [*Date, 1994*] and [*Ullman, 1988*].

It is sometimes said that languages based on the relational calculus are "higher level" or "more declarative" than languages based on relational algebra because the algebra (partially) specifies the order of operations while the calculus leaves it to a compiler or interpreter to determine the most efficient order of evaluation.

The SQL Language

As is the case with most modern relational languages, SQL is based on the tuple relational calculus. As a result every query that can be formulated using the tuple relational calculus (or equivalently, relational algebra) can also be formulated using SQL. There are, however, capabilities beyond the scope of relational algebra or calculus. Here is a list of some additional features provided by SQL that are not part of relational algebra or calculus:

- Commands for insertion, deletion or modification of data.
- Arithmetic capability: In SQL it is possible to involve arithmetic operations as well as comparisons, e.g. A < B + 3. Note that + or other arithmetic operators appear neither in relational algebra nor in relational calculus.
- Assignment and Print Commands: It is possible to print a relation constructed by a query and to assign a computed relation to a relation name.
- Aggregate Functions: Operations such as average, sum, max, etc. can be applied to columns of a relation to obtain a single quantity.

Select

The most often used command in SQL is the SELECT statement, used to retrieve data. The syntax is:

```
SELECT [ALL|DISTINCT]
       { * | expr_1 [AS c_alias_1] [, ...
           [, expr_k [AS c_alias_k]]]}
FROM table_name_1 [t_alias_1]
    [, ... [, table_name_n [t_alias_n]]]
[WHERE condition]
[GROUP BY name_of_attr_i
         [,... [, name_of_attr_j]] [HAVING condition]]
[{UNION [ALL] | INTERSECT | EXCEPT} SELECT ...]
[ORDER BY name_of_attr_i [ASC|DESC]
         [, ... [, name_of_attr_j [ASC|DESC]]]];
```

Now we will illustrate the complex syntax of the SELECT statement with various examples. The tables used for the examples are defined in *The Suppliers and Parts Database*.

Simple Selects

Here are some simple examples using a SELECT statement:

Example 2-4. Simple Query with Qualification

To retrieve all tuples from table PART where the attribute PRICE is greater than 10 we formulate the following query:

```
SELECT * FROM PART
  WHERE PRICE > 10;
```

and get the table:

```
PNO | PNAME | PRICE
-----+---------+--------
 3 | Bolt  |  15
 4 | Cam   |  25
```

Using "*" in the SELECT statement will deliver all attributes from the table. If we want to retrieve only the attributes PNAME and PRICE from table PART we use the statement:

```
SELECT PNAME, PRICE
FROM PART
WHERE PRICE > 10;
```

In this case the result is:

```
PNAME | PRICE
--------+--------
Bolt  |  15
Cam   |  25
```

Note that the SQL SELECT corresponds to the "projection" in relational algebra not to the "selection" (see *Relational Algebra* for more details).

The qualifications in the WHERE clause can also be logically connected using the keywords OR, AND, and NOT:

```
SELECT PNAME, PRICE
FROM PART
WHERE PNAME = 'Bolt' AND
    (PRICE = 0 OR PRICE < 15);
```

will lead to the result:

```
PNAME | PRICE
--------+--------
Bolt  |  15
```

Arithmetic operations may be used in the target list and in the WHERE clause. For example if we want to know how much it would cost if we take two pieces of a part we could use the following query:

```
SELECT PNAME, PRICE * 2 AS DOUBLE
FROM PART
WHERE PRICE * 2 < 50;
```

and we get:

```
PNAME | DOUBLE
--------+---------
Screw |   20
Nut   |   16
Bolt  |   30
```

Note that the word DOUBLE after the keyword AS is the new title of the second column. This technique can be used for every element of the target list to assign a new title to the resulting column. This new title is often referred to as alias. The alias cannot be used throughout the rest of the query.

Joins

The following example shows how joins are realized in SQL.

To join the three tables SUPPLIER, PART and SELLS over their common attributes we formulate the following statement:

```
SELECT S.SNAME, P.PNAME
FROM SUPPLIER S, PART P, SELLS SE
WHERE S.SNO = SE.SNO AND
    P.PNO = SE.PNO;
```

and get the following table as a result:

```
 SNAME | PNAME
-------+-------
 Smith | Screw
 Smith | Nut
 Jones | Cam
 Adams | Screw
 Adams | Bolt
 Blake | Nut
 Blake | Bolt
 Blake | Cam
```

In the FROM clause we introduced an alias name for every relation because there are common named attributes (SNO and PNO) among the relations. Now we can distinguish between the common named attributes by simply prefixing the attribute name with the alias name followed by a dot. The join is calculated in the same way as shown in *An Inner Join*. First the Cartesian product SUPPLIER × PART × SELLS is derived. Now only those tuples satisfying the conditions given in the WHERE clause are selected (i.e. the common named attributes have to be equal). Finally we project out all columns but S.SNAME and P.PNAME.

Aggregate Operators

SQL provides aggregate operators (e.g. AVG, COUNT, SUM, MIN, MAX) that take the name of an attribute as an argument. The value of the aggregate operator is calculated over all values of the specified attribute (column) of the whole table. If groups are specified in the query the calculation is done only over the values of a group (see next section).

Example 2-5. Aggregates

If we want to know the average cost of all parts in table PART we use the following query:

```
SELECT AVG(PRICE) AS AVG_PRICE
FROM PART;
```

The result is:

```
 AVG_PRICE
-----------
   14.5
```

If we want to know how many parts are stored in table PART we use the statement:

```
SELECT COUNT(PNO)
FROM PART;
```

and get:

```
 COUNT
-------
   4
```

Aggregation by Groups

SQL allows one to partition the tuples of a table into groups. Then the aggregate operators described above can be applied to the groups (i.e. the value of the aggregate operator is no longer calculated over all the values of the specified column but over all values of a group. Thus the aggregate operator is evaluated individually for every group.)

The partitioning of the tuples into groups is done by using the keywords GROUP BY followed by a list of attributes that define the groups. If we have GROUP BY A1, ?, Ak we partition the relation into groups, such that two tuples are in the same group if and only if they agree on all the attributes A1, ?, Ak.

Example 2-6. Aggregates

If we want to know how many parts are sold by every supplier we formulate the query:

```
SELECT S.SNO, S.SNAME, COUNT(SE.PNO)
FROM SUPPLIER S, SELLS SE
WHERE S.SNO = SE.SNO
GROUP BY S.SNO, S.SNAME;
```

and get:

```
 SNO | SNAME | COUNT
-----+-------+-------
  1  | Smith |   2
  2  | Jones |   1
  3  | Adams |   2
  4  | Blake |   3
```

Now let’s have a look of what is happening here. First the join of the tables SUPPLIER and SELLS is derived:

```
 S.SNO | S.SNAME | SE.PNO
-------+---------+--------
   1   |  Smith  |   1
   1   |  Smith  |   2
   2   |  Jones  |   4
   3   |  Adams  |   1
   3   |  Adams  |   3
   4   |  Blake  |   2
   4   |  Blake  |   3
   4   |  Blake  |   4
```

Next we partition the tuples into groups by putting all tuples together that agree on both attributes S.SNO and S.SNAME:

```
 S.SNO | S.SNAME | SE.PNO
-------+---------+--------
   1   |  Smith  |   1
```

```
                 |   2
--------------------------
   2   |  Jones  |   4
--------------------------
   3   |  Adams  |   1
                 |   3
--------------------------
   4   |  Blake  |   2
                 |   3
                 |   4
```

In our example we got four groups and now we can apply the aggregate operator COUNT to every group leading to the total result of the query given above.

Note that for the result of a query using GROUP BY and aggregate operators to make sense the attributes grouped by must also appear in the target list. All further attributes not appearing in the GROUP BY clause can only be selected by using an aggregate function. On the other hand you can not use aggregate functions on attributes appearing in the GROUP BY clause.

Having

The HAVING clause works much like the WHERE clause and is used to consider only those groups satisfying the qualification given in the HAVING clause. The expressions allowed in the HAVING clause must involve aggregate functions. Every expression using only plain attributes belongs to the WHERE clause. On the other hand every expression involving an aggregate function must be put to the HAVING clause.

Example 2-7. Having

If we want only those suppliers selling more than one part we use the query:

```
SELECT S.SNO, S.SNAME, COUNT(SE.PNO)
FROM SUPPLIER S, SELLS SE
WHERE S.SNO = SE.SNO
GROUP BY S.SNO, S.SNAME
HAVING COUNT(SE.PNO) > 1;
```

and get:

```
 SNO | SNAME | COUNT
-----+-------+-------
  1  | Smith |   2
  3  | Adams |   2
  4  | Blake |   3
```

Subqueries

In the WHERE and HAVING clauses the use of subqueries (subselects) is allowed in every place where a value is expected. In this case the value must be derived by evaluating the subquery first. The usage of subqueries extends the expressive power of SQL.

Example 2-8. Subselect

If we want to know all parts having a greater price than the part named 'Screw' we use the query:

```
SELECT *
FROM PART
WHERE PRICE > (SELECT PRICE FROM PART
               WHERE PNAME='Screw');
```

The result is:

```
 PNO |  PNAME  |  PRICE
-----+---------+--------
  3  |  Bolt   |   15
  4  |  Cam    |   25
```

When we look at the above query we can see the keyword SELECT two times. The first one at the beginning of the query - we will refer to it as outer SELECT - and the one in the WHERE clause which begins a nested query - we will refer to it as inner SELECT. For every tuple of the outer SELECT the inner SELECT has to be evaluated. After every evaluation we know the price of the tuple named 'Screw' and we can check if the price of the actual tuple is greater.

If we want to know all suppliers that do not sell any part (e.g. to be able to remove these suppliers from the database) we use:

```
SELECT *
FROM SUPPLIER S
WHERE NOT EXISTS
          (SELECT * FROM SELLS SE
           WHERE SE.SNO = S.SNO);
```

In our example the result will be empty because every supplier sells at least one part. Note that we use S.SNO from the outer SELECT within the WHERE clause of the inner SELECT. As described above the subquery is evaluated for every tuple from the outer query i.e. the value for S.SNO is always taken from the actual tuple of the outer SELECT.

Union, Intersect, Except

These operations calculate the union, intersect and set theoretic difference of the tuples derived by two subqueries.

Example 2-9. Union, Intersect, Except

The following query is an example for UNION:

```
SELECT S.SNO, S.SNAME, S.CITY
FROM SUPPLIER S
WHERE S.SNAME = 'Jones'
UNION
SELECT S.SNO, S.SNAME, S.CITY
FROM SUPPLIER S
WHERE S.SNAME = 'Adams';
```

gives the result:

```
 SNO | SNAME |  CITY
-----+-------+--------
  2  | Jones | Paris
  3  | Adams | Vienna
```

Here an example for INTERSECT:

```
SELECT S.SNO, S.SNAME, S.CITY
```

```
FROM SUPPLIER S
WHERE S.SNO > 1
INTERSECT
SELECT S.SNO, S.SNAME, S.CITY
FROM SUPPLIER S
WHERE S.SNO > 2;
```

gives the result:

```
 SNO | SNAME |  CITY
-----+-------+--------
  2  | Jones | Paris
```

```
The only tuple returned by both parts of the query is the one having
$SNO=2$.
```

Finally an example for EXCEPT:

```
SELECT S.SNO, S.SNAME, S.CITY
FROM SUPPLIER S
WHERE S.SNO > 1
EXCEPT
SELECT S.SNO, S.SNAME, S.CITY
FROM SUPPLIER S
WHERE S.SNO > 3;
```

gives the result:

```
 SNO | SNAME |  CITY
-----+-------+--------
  2  | Jones | Paris
  3  | Adams | Vienna
```

Data Definition

There is a set of commands used for data definition included in the SQL language.

Create Table

The most fundamental command for data definition is the one that creates a new relation (a new table). The syntax of the CREATE TABLE command is:

```
CREATE TABLE table_name
             (name_of_attr_1 type_of_attr_1
              [, name_of_attr_2 type_of_attr_2
              [, ...]]);
```

Example 2-10. Table Creation

To create the tables defined in *The Suppliers and Parts Database* the following SQL statements are used:

```
CREATE TABLE SUPPLIER
             (SNO   INTEGER,
              SNAME VARCHAR(20),
              CITY  VARCHAR(20));


CREATE TABLE PART
             (PNO   INTEGER,
```

```
               PNAME VARCHAR(20),
               PRICE DECIMAL(4 , 2));


CREATE TABLE SELLS
             (SNO INTEGER,
              PNO INTEGER);
```

Data Types in SQL

The following is a list of some data types that are supported by SQL:

INTEGER: signed fullword binary integer (31 bits precision).

SMALLINT: signed halfword binary integer (15 bits precision).

DECIMAL (p[,q]): signed packed decimal number of p digits precision with assumed q of them right to the decimal point. ($15 \geq p \geq qq \geq 0$). If q is omitted it is assumed to be 0.

FLOAT: signed doubleword floating point number.

CHAR(n): fixed length character string of length n.

VARCHAR(n): varying length character string of maximum length n.

Create Index

Indices are used to speed up access to a relation. If a relation R has an index on attribute A then we can retrieve all tuples t having t(A) = a in time roughly proportional to the number of such tuples t rather than in time proportional to the size of R.

To create an index in SQL the CREATE INDEX command is used. The syntax is:

```
CREATE INDEX index_name
ON table_name ( name_of_attribute );
```

Example 2-11. Create Index

To create an index named I on attribute SNAME of relation SUPPLIER we use the following statement:

```
CREATE INDEX I
ON SUPPLIER (SNAME);
```

The created index is maintained automatically, i.e. whenever a new tuple is inserted into the relation SUPPLIER the index I is adapted. Note that the only changes a user can percept when an index is present are an increased speed.

Create View

A view may be regarded as a virtual table, i.e. a table that does not physically exist in the database but looks to the user as if it does. By contrast, when we talk of a base table there is really a physically stored counterpart of each row of the table somewhere in the physical storage.

Views do not have their own, physically separate, distinguishable stored data. Instead, the system stores the definition of the view (i.e. the rules about how to access physically stored base tables in order to materialize the view) somewhere in the system catalogs (see *System Catalogs*). For a discussion on different techniques to implement views refer to SIM98.

In SQL the CREATE VIEW command is used to define a view. The syntax is:

```
CREATE VIEW view_name
AS select_stmt
```

where select_stmt is a valid select statement as defined in *Select*. Note that select_stmt is not executed when the view is created. It is just stored in the system catalogs and is executed whenever a query against the view is made.

Let the following view definition be given (we use the tables from *The Suppliers and Parts Database* again):

```
CREATE VIEW London_Suppliers
   AS SELECT S.SNAME, P.PNAME
      FROM SUPPLIER S, PART P, SELLS SE
      WHERE S.SNO = SE.SNO AND
            P.PNO = SE.PNO AND
            S.CITY = 'London';
```

Now we can use this virtual relation London_Suppliers as if it were another base table:

```
SELECT *
FROM London_Suppliers
WHERE P.PNAME = 'Screw';
```

which will return the following table:

```
 SNAME | PNAME
-------+-------
 Smith | Screw
```

To calculate this result the database system has to do a hidden access to the base tables SUPPLIER, SELLS and PART first. It does so by executing the query given in the view definition against those base tables. After that the additional qualifications (given in the query against the view) can be applied to obtain the resulting table.

Drop Table, Drop Index, Drop View

To destroy a table (including all tuples stored in that table) the DROP TABLE command is used:

```
DROP TABLE table_name;
```

To destroy the SUPPLIER table use the following statement:

```
DROP TABLE SUPPLIER;
```

The DROP INDEX command is used to destroy an index:

```
DROP INDEX index_name;
```

Finally to destroy a given view use the command DROP VIEW:

```
DROP VIEW view_name;
```

Data Manipulation

Insert Into

Once a table is created (see *Create Table*), it can be filled with tuples using the command INSERT INTO. The syntax is:

```
INSERT INTO table_name (name_of_attr_1
                          [, name_of_attr_2 [,...]])
VALUES (val_attr_1
        [, val_attr_2 [, ...]]);
```

To insert the first tuple into the relation SUPPLIER (from *The Suppliers and Parts Database*) we use the following statement:

```
INSERT INTO SUPPLIER (SNO, SNAME, CITY)
VALUES (1, 'Smith', 'London');
```

To insert the first tuple into the relation SELLS we use:

```
INSERT INTO SELLS (SNO, PNO)
VALUES (1, 1);
```

Update

To change one or more attribute values of tuples in a relation the UPDATE command is used. The syntax is:

```
UPDATE table_name
SET name_of_attr_1 = value_1
    [, ... [, name_of_attr_k = value_k]]
WHERE condition;
```

To change the value of attribute PRICE of the part 'Screw' in the relation PART we use:

```
UPDATE PART
SET PRICE = 15
WHERE PNAME = 'Screw';
```

The new value of attribute PRICE of the tuple whose name is 'Screw' is now 15.

Delete

To delete a tuple from a particular table use the command DELETE FROM. The syntax is:

```
DELETE FROM table_name
WHERE condition;
```

To delete the supplier called 'Smith' of the table SUPPLIER the following statement is used:

```
DELETE FROM SUPPLIER
WHERE SNAME = 'Smith';
```

System Catalogs

In every SQL database system system catalogs are used to keep track of which tables, views indexes etc. are defined in the database. These system catalogs can be queried as if they were normal relations. For example there is one catalog used for the definition of views. This catalog stores the query from the view definition. Whenever a query against a view is made, the system first gets the view definition query out of the catalog and materializes the view before proceeding with the user query (see SIM98 for a more detailed description). For more information about system catalogs refer to DATE.

Embedded SQL

In this section we will sketch how SQL can be embedded into a host language (e.g. C). There are two main reasons why we want to use SQL from a host language:

- There are queries that cannot be formulated using pure SQL (i.e. recursive queries). To be able to perform such queries we need a host language with a greater expressive power than SQL.
- We simply want to access a database from some application that is written in the host language (e.g. a ticket reservation system with a graphical user interface is written in C and the information about which tickets are still left is stored in a database that can be accessed using embedded SQL).

A program using embedded SQL in a host language consists of statements of the host language and of embedded SQL (ESQL) statements. Every ESQL statement begins with the keywords EXEC SQL. The ESQL statements are transformed to statements of the host language by a precompiler (which usually inserts calls to library routines that perform the various SQL commands).

When we look at the examples throughout *Select* we realize that the result of the queries is very often a set of tuples. Most host languages are not designed to operate on sets so we need a mechanism to access every single tuple of the set of tuples returned by a SELECT statement. This mechanism can be provided by declaring a cursor. After that we can use the FETCH command to retrieve a tuple and set the cursor to the next tuple.

For a detailed discussion on embedded SQL refer to [*Date and Darwen, 1997*], [*Date, 1994*], or [*Ullman, 1988*].

Chapter 3. Architecture

Postgres Architectural Concepts

Before we begin, you should understand the basic Postgres system architecture. Understanding how the parts of Postgres interact will make the next chapter somewhat clearer. In database jargon, Postgres uses a simple "process per-user" client/server model. A Postgres session consists of the following cooperating UNIX processes (programs):

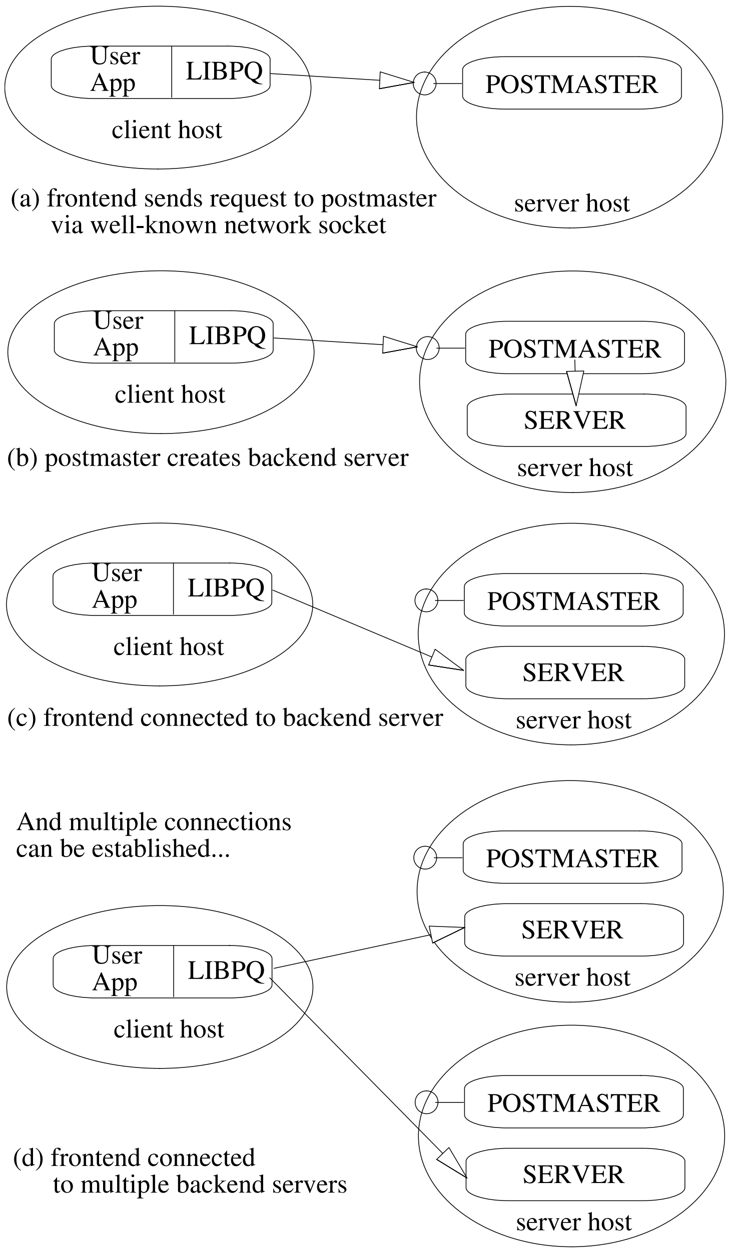
- A supervisory daemon process (postmaster),
- the user’s frontend application (e.g., the psql program), and
- the one or more backend database servers (the postgres process itself).

A single postmaster manages a given collection of databases on a single host. Such a collection of databases is called an installation or site. Frontend applications that wish to access a given database within an installation make calls to the library. The library sends user requests over the network to the postmaster (*How a connection is established*), which in turn starts a new backend server process and connects the frontend process to the new server. From that point on, the frontend process and the backend server communicate without intervention by the postmaster. Hence, the postmaster is always running, waiting for requests, whereas frontend and backend processes come and go.

The libpq library allows a single frontend to make multiple connections to backend processes. However, the frontend application is still a single-threaded process. Multithreaded frontend/backend connections are not currently supported in libpq. One implication of this architecture is that the postmaster and the backend always run on the same machine (the database server), while the frontend application may run anywhere. You should keep this in mind, because the files that can be accessed on a client machine may not be accessible (or may only be accessed using a different filename) on the database server machine.

You should also be aware that the postmaster and postgres servers run with the user-id of the Postgres "superuser." Note that the Postgres superuser does not have to be a special user (e.g., a user named "postgres"). Furthermore, the Postgres superuser should definitely not be the UNIX superuser ("root")! In any case, all files relating to a database should belong to this Postgres superuser.

Figure 3-1. How a connection is established



Chapter 4. Getting Started

How to begin work with Postgres for a new user.

Some of the steps required to use Postgres can be performed by any Postgres user, and some must be done by the site database administrator. This site administrator is the person who installed the software, created the database directories and started the postmaster process. This person does not have to be the UNIX superuser (root) or the computer system administrator; a person can install and use Postgres without any special accounts or privileges.

If you are installing Postgres yourself, then refer to the Administrator's Guide for instructions on installation, and return to this guide when the installation is complete.

Throughout this manual, any examples that begin with the character % are commands that should be typed at the UNIX shell prompt. Examples that begin with the character * are commands in the Postgres query language, Postgres SQL.

Setting Up Your Environment

This section discusses how to set up your own environment so that you can use frontend applications. We assume Postgres has already been successfully installed and started; refer to the Administrator's Guide and the installation notes for how to install Postgres.

Postgres is a client/server application. As a user, you only need access to the client portions of the installation (an example of a client application is the interactive monitor psql). For simplicity, we will assume that Postgres has been installed in the directory /usr/local/pgsql. Therefore, wherever you see the directory /usr/local/pgsql you should substitute the name of the directory where Postgres is actually installed. All Postgres commands are installed in the directory /usr/local/pgsql/bin. Therefore, you should add this directory to your shell command path. If you use a variant of the Berkeley C shell, such as csh or tcsh, you would add

```
% set path = ( /usr/local/pgsql/bin path )
```

in the .login file in your home directory. If you use a variant of the Bourne shell, such as sh, ksh, or bash, then you would add

```
% PATH=/usr/local/pgsql/bin:$PATH
% export PATH
```

to the .profile file in your home directory. From now on, we will assume that you have added the Postgres bin directory to your path. In addition, we will make frequent reference to setting a shell variable or setting an environment variable throughout this document. If you did not fully understand the last paragraph on modifying your search path, you should consult the UNIX manual pages that describe your shell before going any further.

If your site administrator has not set things up in the default way, you may have some more work to do. For example, if the database server machine is a remote machine, you will need to set the PGHOST environment variable to the name of the database server machine. The environment variable PGPORT may also have to be set. The bottom line is this: if you try to start an application program and it complains that it cannot connect to the postmaster, you should immediately consult your site administrator to make sure that your environment is properly set up.

Starting the Interactive Monitor (psql)

Assuming that your site administrator has properly started the postmaster process and authorized you to use the database, you (as a user) may begin to start up applications. As previously mentioned, you should add /usr/local/pgsql/bin to your shell search path. In most cases, this is all you should have to do in terms of preparation.

As of Postgres v6.3, two different styles of connections are supported. The site administrator will have chosen to allow TCP/IP network connections or will have restricted database access to local (same-machine) socket connections only. These choices become significant if you encounter problems in connecting to a database.

If you get the following error message from a Postgres command (such as psql or createdb):

```
% psql template1
Connection to database 'postgres' failed.
connectDB() failed: Is the postmaster running and accepting connections
    at 'UNIX Socket' on port '5432'?
```

or

```
% psql -h localhost template1
Connection to database 'postgres' failed.
connectDB() failed: Is the postmaster running and accepting TCP/IP
    (with -i) connections at 'localhost' on port '5432'?
```

it is usually because (1) the postmaster is not running, or (2) you are attempting to connect to the wrong server host. If you get the following error message:

```
FATAL 1:Feb 17 23:19:55:process userid (2360) != database owner (268)
```

it means that the site administrator started the postmaster as the wrong user. Tell him to restart it as the Postgres superuser.

Managing a Database

Now that Postgres is up and running we can create some databases to experiment with. Here, we describe the basic commands for managing a database.

Most Postgres applications assume that the database name, if not specified, is the same as the name on your computer account.

If your database administrator has set up your account without database creation privileges, then she should have told you what the name of your database is. If this is the case, then you can skip the sections on creating and destroying databases.

Creating a Database

Let's say you want to create a database named mydb. You can do this with the following command:

```
% createdb mydb
```

If you do not have the privileges required to create a database, you will see the following:

```
% createdb mydb
WARN:user "your username" is not allowed to create/destroy databases
createdb: database creation failed on mydb.
```

Postgres allows you to create any number of databases at a given site and you automatically become the database administrator of the database you just created. Database names must have an alphabetic first character and are limited to 32 characters in length. Not every user has authorization to become a database administrator. If Postgres refuses to create databases for you, then the site administrator needs to grant you permission to create databases. Consult your site administrator if this occurs.

Accessing a Database

Once you have constructed a database, you can access it by:

- running the Postgres terminal monitor programs (e.g. psql) which allows you to interactively enter, edit, and execute SQL commands.
- writing a C program using the LIBPQ subroutine library. This allows you to submit SQL commands from C and get answers and status messages back to your program. This interface is discussed further in The PostgreSQL Programmer's Guide.

You might want to start up psql, to try out the examples in this manual. It can be activated for the mydb database by typing the command:

```
% psql mydb
```

You will be greeted with the following message:

```
Welcome to the POSTGRESQL interactive sql monitor:
  Please read the file COPYRIGHT for copyright terms of POSTGRESQL

   type \? for help on slash commands
   type \q to quit
   type \g or terminate with semicolon to execute query
 You are currently connected to the database: template1

mydb=>
```

This prompt indicates that the terminal monitor is listening to you and that you can type SQL queries into a workspace maintained by the terminal monitor. The psql program responds to escape codes that begin with the backslash character, \ For example, you can get help on the syntax of various Postgres SQL commands by typing:

```
mydb=> \h
```

Once you have finished entering your queries into the workspace, you can pass the contents of the workspace to the Postgres server by typing:

```
mydb=> \g
```

This tells the server to process the query. If you terminate your query with a semicolon, the \g is not necessary. psql will automatically process semicolon terminated queries. To read queries from a file, say myFile, instead of entering them interactively, type:

```
mydb=> \i fileName
```

To get out of psql and return to UNIX, type

```
mydb=> \q
```

and psql will quit and return you to your command shell. (For more escape codes, type \h at the monitor prompt.) White space (i.e., spaces, tabs and newlines) may be used freely in SQL queries. Single-line comments are denoted by `--`. Everything after the dashes up to the end of the line is ignored. Multiple-line comments, and comments within a line, are denoted by `/* ... */`

Destroying a Database

If you are the database administrator for the database mydb, you can destroy it using the following UNIX command:

```
% destroydb mydb
```

This action physically removes all of the UNIX files associated with the database and cannot be undone, so this should only be done with a great deal of forethought.

Chapter 5. The Query Language

The Postgres query language is a variant of the SQL3 draft next-generation standard. It has many extensions such as an extensible type system, inheritance, functions and production rules. These are features carried over from the original Postgres query language, PostQuel. This section provides an overview of how to use Postgres SQL to perform simple operations. This manual is only intended to give you an idea of our flavor of SQL and is in no way a complete tutorial on SQL. Numerous books have been written on SQL, including [MELT93] and [DATE97]. You should be aware that some language features are extensions to the ANSI standard.

Interactive Monitor

In the examples that follow, we assume that you have created the mydb database as described in the previous subsection and have started psql. Examples in this manual can also be found in /usr/local/pgsql/src/tutorial/. Refer to the README file in that directory for how to use them. To start the tutorial, do the following:

```
% cd /usr/local/pgsql/src/tutorial
% psql -s mydb
Welcome to the POSTGRESQL interactive sql monitor:
  Please read the file COPYRIGHT for copyright terms of POSTGRESQL

   type \? for help on slash commands
   type \q to quit
   type \g or terminate with semicolon to execute query
 You are currently connected to the database: postgres

mydb=> \i basics.sql
```

The \i command read in queries from the specified files. The -s option puts you in single step mode which pauses before sending a query to the backend. Queries in this section are in the file basics.sql.

psql has a variety of \d commands for showing system information. Consult these commands for more details; for a listing, type \? at the psql prompt.

Concepts

The fundamental notion in Postgres is that of a class, which is a named collection of object instances. Each instance has the same collection of named attributes, and each attribute is of a specific type. Furthermore, each instance has a permanent object identifier (OID) that is unique throughout the installation. Because SQL syntax refers to tables, we will use the terms table and class interchangeably. Likewise, an SQL row is an instance and SQL columns are attributes. As previously discussed, classes are grouped into databases, and a collection of databases managed by a single postmaster process constitutes an installation or site.

Creating a New Class

You can create a new class by specifying the class name, along with all attribute names and their types:

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,           -- low temperature
    temp_hi         int,           -- high temperature
    prcp            real,          -- precipitation
    date            date
);
```

Note that both keywords and identifiers are case-insensitive; identifiers can become case-sensitive by surrounding them with double-quotes as allowed by SQL92. Postgres SQL supports the usual SQL types int, float, real, smallint, char(N), varchar(N), date, time, and timestamp, as well as other types of general utility and a rich set of geometric types. As we will see later, Postgres can be customized with an arbitrary number of user-defined data types. Consequently, type names are not syntactical keywords, except where required to support special cases in the SQL92 standard. So far, the Postgres create command looks exactly like the command used to create a table in a traditional relational system. However, we will presently see that classes have properties that are extensions of the relational model.

Populating a Class with Instances

The insert statement is used to populate a class with instances:

```
INSERT INTO weather
    VALUES ('San Francisco', 46, 50, 0.25, '11/27/1994')
```

You can also use the copy command to perform load large amounts of data from flat (ASCII) files. This is usually faster because the data is read (or written) as a single atomic transaction directly to or from the target table. An example would be:

```
COPY INTO weather FROM '/home/user/weather.txt'
    USING DELIMITERS '|';
```

where the path name for the source file must be available to the backend server machine, not the client, since the backend server reads the file directly.

Querying a Class

The weather class can be queried with normal relational selection and projection queries. A SQL select statement is used to do this. The statement is divided into a target list (the part that lists the attributes to be returned) and a qualification (the part that specifies any restrictions). For example, to retrieve all the rows of weather, type:

```
SELECT * FROM WEATHER;
```

and the output should be:

```
+--------------+---------+---------+------+------------+
|city          | temp_lo | temp_hi | prcp | date       |
+--------------+---------+---------+------+------------+
|San Francisco | 46      | 50      | 0.25 | 11-27-1994 |
+--------------+---------+---------+------+------------+
|San Francisco | 43      | 57      | 0    | 11-29-1994 |
+--------------+---------+---------+------+------------+
|Hayward       | 37      | 54      |      | 11-29-1994 |
+--------------+---------+---------+------+------------+
```

You may specify any arbitrary expressions in the target list. For example, you can do:

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date FROM weather;
```

Arbitrary Boolean operators (and, or and not) are allowed in the qualification of any query. For example,

```
SELECT * FROM weather
    WHERE city = ’San Francisco’
    AND prcp > 0.0;
```

results in:

```
+--------------+---------+---------+------+------------+
|city          | temp_lo | temp_hi | prcp | date       |
+--------------+---------+---------+------+------------+
|San Francisco | 46      | 50      | 0.25 | 11-27-1994 |
+--------------+---------+---------+------+------------+
```

As a final note, you can specify that the results of a select can be returned in a sorted order or with duplicate instances removed.

```
SELECT DISTINCT city
    FROM weather
    ORDER BY city;
```

Redirecting SELECT Queries

Any select query can be redirected to a new class

```
SELECT * INTO TABLE temp FROM weather;
```

This forms an implicit create command, creating a new class temp with the attribute names and types specified in the target list of the select into command. We can then, of course, perform any operations on the resulting class that we can perform on other classes.

Joins Between Classes

Thus far, our queries have only accessed one class at a time. Queries can access multiple classes at once, or access the same class in such a way that multiple instances of the class are being processed at the same time. A query that accesses multiple instances of the same or different classes at one time is called a join query. As an example, say we wish to find all the records that are in the temperature range of other records. In effect, we need to compare the temp_lo and temp_hi attributes of each EMP instance to the temp_lo and temp_hi attributes of all other EMP instances.

> Note: This is only a conceptual model. The actual join may be performed in a more efficient manner, but this is invisible to the user.

We can do this with the following query:

```
SELECT W1.city, W1.temp_lo AS low, W1.temp_hi AS high,
    W2.city, W2.temp_lo AS low, W2.temp_hi AS high
    FROM weather W1, weather W2
    WHERE W1.temp_lo < W2.temp_lo
    AND W1.temp_hi > W2.temp_hi;

+--------------+-----+------+---------------+-----+------+
|city          | low | high | city          | low | high |
+--------------+-----+------+---------------+-----+------+
|San Francisco | 43  | 57   | San Francisco | 46  | 50   |
+--------------+-----+------+---------------+-----+------+
|San Francisco | 37  | 54   | San Francisco | 46  | 50   |
+--------------+-----+------+---------------+-----+------+
```

> Note: The semantics of such a join are that the qualification is a truth expression defined for the Cartesian product of the classes indicated in the query. For those instances in the Cartesian product for which the qualification is true, Postgres computes and returns the values specified in the target list. Postgres SQL does not assign any meaning to duplicate values in such expressions. This means that Postgres sometimes recomputes the same target list several times; this frequently happens when Boolean expressions are connected with an "or". To remove such duplicates, you must use the select distinct statement.

In this case, both W1 and W2 are surrogates for an instance of the class weather, and both range over all instances of the class. (In the terminology of most database systems, W1 and W2 are known as range variables.) A query can contain an arbitrary number of class names and surrogates.

Updates

You can update existing instances using the update command. Suppose you discover the temperature readings are all off by 2 degrees as of Nov 28, you may update the data as follow:

```
UPDATE weather
    SET temp_hi = temp_hi - 2,  temp_lo = temp_lo - 2
    WHERE date > '11/28/1994';
```

Deletions

Deletions are performed using the delete command:

```
DELETE FROM weather WHERE city = 'Hayward';
```

All weather recording belongs to Hayward is removed. One should be wary of queries of the form

```
DELETE FROM classname;
```

Without a qualification, delete will simply remove all instances of the given class, leaving it empty. The system will not request confirmation before doing this.

Using Aggregate Functions

Like most other query languages, PostgreSQL supports aggregate functions. The current implementation of Postgres aggregate functions have some limitations. Specifically, while there are aggregates to compute such functions as the count, sum, avg (average), max (maximum) and min (minimum) over a set of instances, aggregates can only appear in the target list of a query and not directly in the qualification (the where clause). As an example,

```
SELECT max(temp_lo) FROM weather;
```

is allowed, while

```
SELECT city FROM weather WHERE temp_lo = max(temp_lo);
```

is not. However, as is often the case the query can be restated to accomplish the intended result; here by using a subselect:

```
SELECT city FROM weather WHERE temp_lo = (SELECT max(temp_lo) FROM
weather);
```

Aggregates may also have group by clauses:

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city;
```

Chapter 6. Advanced Postgres SQL Features

Having covered the basics of using Postgres SQL to access your data, we will now discuss those features of Postgres that distinguish it from conventional data managers. These features include inheritance, time travel and non-atomic data values (array- and set-valued attributes). Examples in this section can also be found in advance.sql in the tutorial directory. (Refer to Chapter 5 for how to use it.)

Inheritance

Let's create two classes. The capitals class contains state capitals which are also cities. Naturally, the capitals class should inherit from cities.

```
CREATE TABLE cities (
    name            text,
    population      float,
    altitude        int             -- (in ft)
);

CREATE TABLE capitals (
    state           char2
) INHERITS (cities);
```

In this case, an instance of capitals inherits all attributes (name, population, and altitude) from its parent, cities. The type of the attribute name is text, a native Postgres type for variable length ASCII strings. The type of the attribute population is float, a native Postgres type for double precision floating point numbers. State capitals have an extra attribute, state, that shows their state. In Postgres, a class can inherit from zero or more other classes, and a query can reference either all instances of a class or all instances of a class plus all of its descendants.

> Note: The inheritance hierarchy is a directed acyclic graph.

For example, the following query finds all the cities that are situated at an attitude of 500ft or higher:

```
SELECT name, altitude
    FROM cities
    WHERE altitude > 500;

+----------+----------+
|name      | altitude |
+----------+----------+
|Las Vegas | 2174     |
+----------+----------+
|Mariposa  | 1953     |
+----------+----------+
```

On the other hand, to find the names of all cities, including state capitals, that are located at an altitude over 500ft, the query is:

```
SELECT c.name, c.altitude
    FROM cities* c
    WHERE c.altitude > 500;
```

which returns:

```
+----------+----------+
|name      | altitude |
+----------+----------+
|Las Vegas | 2174     |
+----------+----------+
|Mariposa  | 1953     |
+----------+----------+
|Madison   | 845      |
+----------+----------+
```

Here the * after cities indicates that the query should be run over cities and all classes below cities in the inheritance hierarchy. Many of the commands that we have already discussed (select, update and delete) support this * notation, as do others, like alter.

Non-Atomic Values

One of the tenets of the relational model is that the attributes of a relation are atomic. Postgres does not have this restriction; attributes can themselves contain sub-values that can be accessed from the query language. For example, you can create attributes that are arrays of base types.

Arrays

Postgres allows attributes of an instance to be defined as fixed-length or variable-length multi-dimensional arrays. Arrays of any base type or user-defined type can be created. To illustrate their use, we first create a class with arrays of base types.

```
CREATE TABLE SAL_EMP (
    name            text,
    pay_by_quarter  int4[],
    schedule        text[][]
);
```

The above query will create a class named SAL_EMP with a text string (name), a one-dimensional array of int4 (pay_by_quarter), which represents the employee's salary by quarter and a two-dimensional array of text (schedule), which represents the employee's weekly schedule. Now we do some INSERTSs; note that when appending to an array, we enclose the values within braces and separate them by commas. If you know C, this is not unlike the syntax for initializing structures.

```
INSERT INTO SAL_EMP
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {}}');

INSERT INTO SAL_EMP
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"talk", "consult"}, {"meeting"}}');
```

By default, Postgres uses the "one-based" numbering convention for arrays -- that is, an array of n elements starts with array[1] and ends with array[n]. Now, we can run some queries on SAL_EMP. First, we show how to access a single element of an array at a time. This query retrieves the names of the employees whose pay changed in the second quarter:

```
SELECT name
    FROM SAL_EMP
    WHERE SAL_EMP.pay_by_quarter[1] <>
    SAL_EMP.pay_by_quarter[2];

+------+
|name  |
+------+
|Carol |
+------+
```

This query retrieves the third quarter pay of all employees:

```
SELECT SAL_EMP.pay_by_quarter[3] FROM SAL_EMP;


+---------------+
|pay_by_quarter |
+---------------+
|10000          |
+---------------+
|25000          |
+---------------+
```

We can also access arbitrary slices of an array, or subarrays. This query retrieves the first item on Bill's schedule for the first two days of the week.

```
SELECT SAL_EMP.schedule[1:2][1:1]
    FROM SAL_EMP
    WHERE SAL_EMP.name = 'Bill';

+-------------------+
|schedule           |
+-------------------+
|{{"meeting"},{""}} |
+-------------------+
```

Time Travel

As of Postgres v6.2, time travel is no longer supported. There are several reasons for this: performance impact, storage size, and a pg_time file which grows toward infinite size in a short period of time.

New features such as triggers allow one to mimic the behavior of time travel when desired, without incurring the overhead when it is not needed (for most users, this is most of the time). See examples in the contrib directory for more information.

> Time travel is deprecated: The remaining text in this section is retained only until it can be rewritten in the context of new techniques to accomplish the same purpose. Volunteers? - thomas 1998-01-12

Postgres supports the notion of time travel. This feature allows a user to run historical queries. For example, to find the current population of Mariposa city, one would query:

```
SELECT * FROM cities WHERE name = 'Mariposa';

+---------+------------+----------+
|name     | population | altitude |
+---------+------------+----------+
|Mariposa | 1320       | 1953     |
+---------+------------+----------+
```

Postgres will automatically find the version of Mariposa's record valid at the current time. One can also give a time range. For example to see the past and present populations of Mariposa, one would query:

```
SELECT name, population
    FROM cities['epoch', 'now']
    WHERE name = 'Mariposa';
```

where "epoch" indicates the beginning of the system clock.

> Note: On UNIX systems, this is always midnight, January 1, 1970 GMT.

If you have executed all of the examples so far, then the above query returns:

```
+---------+------------+
|name     | population |
+---------+------------+
|Mariposa | 1200       |
+---------+------------+
|Mariposa | 1320       |
+---------+------------+
```

The default beginning of a time range is the earliest time representable by the system and the default end is the current time; thus, the above time range can be abbreviated as ‘‘[,].’’

More Advanced Features

Postgres has many features not touched upon in this tutorial introduction, which has been oriented toward newer users of SQL. These are discussed in more detail in both the User's and Programmer's Guides.

Bibliography

Selected references and readings for SQL and Postgres.

SQL Reference Books

The Practical SQL Handbook, Using Structured Query Language , 3, Judity Bowman, Sandra Emerson, and Marcy Damovsky, 0-201-44787-8, 1996, Addison-Wesley, 1997.

A Guide to the SQL Standard, A user's guide to the standard database language SQL , 4, C. J. Date and Hugh Darwen, 0-201-96426-0, 1997, Addison-Wesley, 1997.

An Introduction to Database Systems, 6, C. J. Date, 1, 1994, Addison-Wesley, 1994.

Understanding the New SQL, A complete guide, Jim Melton and Alan R. Simon, 1-55860-245-3, 1993, Morgan Kaufmann, 1993.

Abstract

Accessible reference for SQL features.

Principles of Database and Knowledge : Base Systems, Jeffrey D. Ullman, 1, Computer Science Press , 1988 .

PostgreSQL-Specific Documentation

The PostgreSQL Administrator's Guide , The Administrator's Guide , Edited by Thomas Lockhart, 1998-10-01, The PostgreSQL Global Development Group.

The PostgreSQL Developer's Guide , The Developer's Guide , Edited by Thomas Lockhart, 1998-10-01, The PostgreSQL Global Development Group.

The PostgreSQL Programmer's Guide , The Programmer's Guide , Edited by Thomas Lockhart, 1998-10-01, The PostgreSQL Global Development Group.

The PostgreSQL Tutorial Introduction , The Tutorial , Edited by Thomas Lockhart, 1998-10-01, The PostgreSQL Global Development Group.

The PostgreSQL User's Guide , The User's Guide , Edited by Thomas Lockhart, 1998-10-01, The PostgreSQL Global Development Group.

Enhancement of the ANSI SQL Implementation of PostgreSQL , Simkovics, 1998 , Stefan Simkovics, O.Univ.Prof.Dr.. Georg Gottlob, November 29, 1998, Department of Information Systems, Vienna University of Technology .

Discusses SQL history and syntax, and describes the addition of INTERSECT and EXCEPT constructs into Postgres. Prepared as a Master's Thesis with the support of O.Univ.Prof.Dr. Georg Gottlob and Univ.Ass. Mag. Katrin Seyr at Vienna University of Technology.

The Postgres95 User Manual , Yu and Chen, 1995 , A. Yu and J. Chen, The POSTGRES Group , Sept. 5, 1995, University of California, Berkeley CA.

Proceedings and Articles

Partial indexing in POSTGRES: research project , Olson, 1993 , Nels Olson, 1993, UCB Engin T7.49.1993 O676, University of California, Berkeley CA.

A Unified Framework for Version Modeling Using Production Rules in a Database System , Ong and Goh, 1990 , L. Ong and J. Goh, April, 1990, ERL Technical Memorandum M90/33, University of California, Berkeley CA.

The Postgres Data Model , Rowe and Stonebraker, 1987 , L. Rowe and M. Stonebraker, Sept. 1987, VLDB Conference, Brighton, England, 1987.

Generalized partial indexes (http://simon.cs.cornell.edu/home/praveen/papers/partindex.de95.ps.Z) , , P. Seshadri and A. Swami, March 1995, Eleventh International Conference on Data Engineering, 1995, Cat. No.95CH35724, IEEE Computer Society Press.

The Design of Postgres , Stonebraker and Rowe, 1986 , M. Stonebraker and L. Rowe, May 1986, Conference on Management of Data, Washington DC, ACM-SIGMOD, 1986.

The Design of the Postgres Rules System, Stonebraker, Hanson, Hong, 1987 , M. Stonebraker, E. Hanson, and C. H. Hong, Feb. 1987, Conference on Data Engineering, Los Angeles, CA, IEEE, 1987.

The Postgres Storage System , Stonebraker, 1987 , M. Stonebraker, Sept. 1987, VLDB Conference, Brighton, England, 1987.

A Commentary on the Postgres Rules System , Stonebraker et al, 1989, M. Stonebraker, M. Hearst, and S. Potamianos, Sept. 1989, Record 18(3), SIGMOD, 1989.

The case for partial indexes (DBMS) (http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M89-17.pdf) , Stonebraker, M, 1989b, M. Stonebraker, Dec. 1989, Record 18(no.4):4-11, SIGMOD, 1989.

The Implementation of Postgres , Stonebraker, Rowe, Hirohama, 1990 , M. Stonebraker, L. A. Rowe, and M. Hirohama, March 1990, Transactions on Knowledge and Data Engineering 2(1), IEEE.

On Rules, Procedures, Caching and Views in Database Systems , Stonebraker et al, ACM, 1990 , M. Stonebraker and et al, June 1990, Conference on Management of Data, ACM-SIGMOD.