APPENDIX A

PostgreSQL Database Limits

When we use a database to store information, we are tempted to ignore the fact that on no platform do we have the luxury of infinite storage. All database systems are limited in some way, and PostgreSQL is no exception. The amount of data that can be stored in a single column, the maximum number of columns allowed in a table, and the total size of any table all have limits, albeit quite large ones.

As a limit is approached, the performance of the database will degrade. If we are, for example, manipulating very large fields consuming a large fraction of available (virtual) memory, it is likely that performance will begin to be unacceptable. Finally, PostgreSQL will be physically unable to perform an update.

Recent releases of PostgreSQL have seen most database limits relaxed, and in many cases, effectively removed. In this appendix, we will mention some of the restrictions that remain as of PostgreSQL version 8.0. For updates on limits for later versions, check out the PostgreSQL web site at `http://www.postgresql.org`.

Note The information here is derived from the PostgreSQL FAQ and mailing list contributions made by PostgreSQL developers.

Where a size is given as "No Limit," this means that PostgreSQL alone imposes no limit. The maximum size will be determined by other factors, such as operating system limits and the amount of available disk space or virtual memory. The network transport may also impose limits. For example, there are typically limits on the size of a query that can be made via ODBC, depending on the driver. Memory limits may prevent very large columns, rows, or result sets from being created, transferred across a network (which in itself will be slow), or received by the client.

Database Size: No Limit

PostgreSQL does not impose a limit on the total size of a database. Databases of 4 terabytes (TB) are reported to exist. A database of this size is more than sufficient for all but the most demanding applications.

Due to the way that PostgreSQL arranges its data storage, you may see some performance degradation associated with databases containing many tables. PostgreSQL may use a large number of files for storing the table data, and performance may suffer if the operating system does not cope well with many files in a single directory. The introduction of tablespaces in PostgreSQL 8.0 helps the database administrator to minimize these effects. Tablespaces are covered in Chapter 11.

Table Size: 16TB–64TB

PostgreSQL normally stores its table data in chunks of 8KB. The number of these blocks is limited to a 32-bit signed integer (just over two billion), giving a maximum table size of 16TB. The basic block size can be increased when PostgreSQL is built, up to a maximum of 32KB, thereby giving a theoretical table size limit of 64TB.

Some operating systems impose a file size limit that prevent files of this size from being created, so PostgreSQL stores table data in multiple files, each 1GB in size. For large tables, this will result in many files and potential operating system performance degradation, as noted earlier.

Rows in a Table: No Limit

PostgreSQL does not impose a limit on the number of rows in any table.

Table Indexes: No Limit

There is no PostgreSQL-imposed limit on the number of indexes you can create on a table. Of course, performance may degrade if you choose to create more and more indexes on a table with more and more columns.

Field Size: 1GB

PostgreSQL has a limit of 1GB for the size of any one field in a table. In practice, the limit comes from the amount of memory available for the server to manipulate the data and transfer it to the client.

Columns in a Table: 250+

The maximum number of columns that can be accommodated in a PostgreSQL table depends on the configured block size and the type of the column. For the default block size of 8KB, at least 250 columns can be stored. This can rise to 1,600 columns if all of the columns are very simple fields, such as integer values. Increasing the block size increases these limits accordingly.

Row Size: No Limit

There is no explicit maximum size of a row. But, of course, the size of columns and their number are limited as described in the preceding text.

APPENDIX B

PostgreSQL Data Types

PostgreSQL has a particularly rich set of data types, which are described in Chapter 11 of this book, as well as in Chapter 8 of the PostgreSQL documentation.

In this appendix, we list the more useful types, ignoring some of the very specialized types and those types used only internally by PostgreSQL. Use `\dT` (or `\dT+` for even more detail) from `psql` for a definitive list of types.

In the tables in this appendix, the standard SQL name appears first, which PostgreSQL generally accepts, followed by any PostgreSQL-specific alternative names. Some types are specific to PostgreSQL and in such cases, no SQL standard name is given in the tables.

As long as it's practical, we suggest that you stick to the standard SQL types and names. Some of the official SQL names are almost invariably shortened in real use; for example `int` for `integer`, `bool` for `boolean`, and `varchar` for `character varying`. We have adhered to the common convention of using the shorter name in this book.

Logical Types

Table B-1 shows the PostgreSQL logical data type: `boolean`. Note that `boolean` was not officially added to the SQL language until the SQL99 standard, although it was in common use long before that.

Table B-1. *PostgreSQL Logical Data Type*

SQL Name	PostgreSQL Alternative Name	Notes
`boolean`	`bool`	Holds a truth value. Will accept values such as `TRUE`, `'t'`, `'true'`, `'y'`, `'yes'`, and `'1'` as `true`. Uses 1 byte of storage, and can store `NULL`, unlike a few proprietary databases.

Exact Number Types

Table B-2 shows the PostgreSQL exact number data types.

Table B-2. *postgresql Exact Number Types*

SQL Name	PostgreSQL Alternative Name	Notes
`smallint`	`int2`	A signed 2-byte integer that can store –32768 to +32767.
`integer`, `int`	`int4`	A signed 4-byte integer that can store –2147483648 to +2147483647.
`bigint`	`int8`	A signed 8-byte integer, giving approximately 18 digits of precision.
`bit`	`bit`	Stores a single bit, 0 or 1. To insert into a table, use syntax such as `INSERT INTO ... VALUES(B'1');`.
`bit varying`	`varbit(n)`	Stores a string of bits. To insert into a table, use syntax such as `INSERT INTO ... VALUES(B'011101');` .

Approximate Number Types

Table B-3 shows the PostgreSQL approximate number data types. Note that the `decimal` type is just an alias for `numeric`, which is the term used by the SQL standard and generally preferred. Similarly, rather than `float`, use `real` or `double precision`.

Table B-3. *PostgreSQL Approximate Number Types*

SQL Name	PostgreSQL Alternative Name	Notes
`numeric (precision, scale)`		Stores an exact number to the precision specified. The user guide states there is no limit to the precision that may be specified.
`real`	`float4`	A 4-byte, single-precision, floating-point number.
`double precision`	`float8`	An 8-byte, double-precision, floating-point number.
`money`		Equivalent to `numeric(9,2)`, storing 4 bytes of data. Its use is discouraged, as it is deprecated and support may be dropped in the future.

Temporal Types

Table B-4 shows the PostgreSQL data types for date and time.

Table B-4. *PostgreSQL Types for Date and Time*

SQL Name	PostgreSQL Alternative Name	Notes
`timestamp`	`datetime`	Stores dates and times from 4713 BC to 1465001 AD, with a resolution of 1 microsecond. You may also see `timestamptz` used sometimes in PostgreSQL, which is a shorthand for timestamp with time zone.
`interval`	`interval`	Stores an interval of approximately +/– 178,000,000 years, with a resolution of 1 microsecond.
`date`	`date`	Stores dates from 4713 BC to 32767 AD, with a resolution of 1 day.
`time`	`time`	Stores a time of day, from 0 to 23:59:59.99, with a resolution of 1 microsecond.

Character Types

Table B-5 shows the PostgreSQL character data types.

Table B-5. *PostgreSQL Character Types*

SQL Name	PostgreSQL Alternative Name	Notes
`char, character`	`bpchar`	Stores a single character.
`char(n)`	`bpchar(n)`	Stores exactly *n* characters, which will be padded with blanks if fewer characters are actually stored.
`character varying(n)`	`varchar(n)`	Stores a variable number of characters, up to a maximum of *n* characters, which are not padded with blanks. This is the standard choice for character strings.
	`text`	A PostgreSQL-specific variant of `varchar`, which does not require you to specify an upper limit on the number of characters.

Geometric Types

Table B-6 shows the PostgreSQL geometric data types. These are specific to PostgreSQL, so there are no SQL names listed.

Table B-6. *PostgreSQL Geometric Types*

PostgreSQL Name	Notes
`point`	An x,y value
`line`	A line (pt1, pt2)
`lseg`	A line segment (pt1, pt2)
`box`	A box specified by a pair of points
`path`	A sequence of points, which may be closed or open
`polygon`	A sequence of points, effectively a closed path
`circle`	A point and a length, which specify a circle

Miscellaneous PostgreSQL Types

As shown in Table B-7, PostgreSQL has some other data types, which do not fit into the previous categories. SQL names are not applicable to these types.

Note that PostgreSQL does not implement the `serial` type as a separate type, although it accepts the conventional SQL syntax. Internally PostgreSQL uses an `integer` to store the value and a sequence to manage the automatic incrementing of the value. When a table is created with a `serial` type, an implicit sequence (named using an underscore separated combination of the table name, the column name, and `seq`) is created to manage the `serial` data column. This implicit sequence will be dropped automatically if the table is dropped.

The `cidr` type refers to Classless Inter-Domain Routing (CIDR). This is a newer standard for IP addressing. This is in contrast to the original form of IP address assignment, which uses three classes—A, B, and C—that have a network part of 8, 16, and 24 bits, respectively, allowing 16.7 million, 65 thousand, and 254 hosts per network, respectively. CIDR allows network masks of any size, so you can better allocate IP addresses and route between them in a hierarchical fashion.

Table B-7. *Other PostgreSQL Types*

PostgreSQL Name	Notes
`serial`	In conventional SQL usage, a `serial` (or auto-incrementing integer) is a numeric column in a table that increases each time a row is added.
`oid`	An object identifier. Internally, PostgreSQL adds, by default, a hidden `oid` to each row, and stores a 4-byte integer, giving a maximum value of approximately 4 billion. This type is also used as a reference when storing binary large objects. We recommend you do not use this type or rely on its existence.
`cidr`	Stores a network address of the form *x.x.x.x*/*y* where *y* is the netmask.
`inet`	Similar to `cidr`, except the host part can be 0.
`macaddr`	A MAC address of the form *XX*:*XX*:*XX*:*XX*:*XX*:*XX*.

APPENDIX C

PostgreSQL SQL Syntax Reference

This appendix presents a list of the PostgreSQL commands, followed by the syntax for each of these commands. This set of commands is taken from the `psql` command-line tool. Using `psql`, you can generate the complete list of commands by using the `\help` command. For the syntax of a specific command, use `\help <command>`.

More detailed explanations are available in Part VI (Reference), Section I (SQL Commands) of the PostgreSQL manual.

PostgreSQL SQL Commands

```
ABORT                       CREATE INDEX              DROP TYPE
ALTER AGGREGATE             CREATE LANGUAGE           DROP USER
ALTER CONVERSION            CREATE OPERATOR CLASS     DROP VIEW
ALTER DATABASE              CREATE OPERATOR           END
ALTER DOMAIN                CREATE RULE               EXECUTE
ALTER FUNCTION              CREATE SCHEMA             EXPLAIN
ALTER GROUP                 CREATE SEQUENCE           FETCH
ALTER INDEX                 CREATE TABLE              GRANT
ALTER LANGUAGE              CREATE TABLE AS           INSERT
ALTER OPERATOR CLASS        CREATE TABLESPACE         LISTEN
ALTER OPERATOR              CREATE TRIGGER            LOAD
ALTER SCHEMA                CREATE TYPE               LOCK
ALTER SEQUENCE              CREATE USER               MOVE
ALTER TABLE                 CREATE VIEW               NOTIFY
ALTER TABLESPACE            DEALLOCATE                PREPARE
ALTER TRIGGER               DECLARE                   REINDEX
```

ALTER TYPE
ALTER USER
ANALYZE
BEGIN
CHECKPOINT
CLOSE
CLUSTER
COMMENT
COMMIT
COPY
CREATE AGGREGATE
CREATE CAST
CREATE CONSTRAINT TRIGGER
CREATE CONVERSION
CREATE DATABASE
CREATE DOMAIN
CREATE FUNCTION
CREATE GROUP
DELETE
DROP AGGREGATE
DROP CAST
DROP CONVERSION
DROP DATABASE
DROP DOMAIN
DROP FUNCTION
DROP GROUP
DROP INDEX
DROP LANGUAGE
DROP OPERATOR
DROP OPERATOR CLASS
DROP RULE
DROP SCHEMA
DROP SEQUENCE
DROP TABLE
DROP TABLESPACE
DROP TRIGGER
RELEASE SAVEPOINT
RESET
REVOKE
ROLLBACK
ROLLBACK TO SAVEPOINT
SAVEPOINT
SELECT
SELECT INTO
SET
SET CONSTRAINTS
SET SESSION AUTHORIZATION
SET TRANSACTION
SHOW
START TRANSACTION
TRUNCATE
UNLISTEN
UPDATE
VACUUM

PostgreSQL SQL Syntax

ABORT

Abort the current transaction.

```
ABORT [ WORK | TRANSACTION ]
```

ALTER AGGREGATE

Change the definition of an aggregate function.

```
ALTER AGGREGATE name ( type ) RENAME TO new_name
ALTER AGGREGATE name ( type ) OWNER TO new_owner
```

ALTER CONVERSION

Change the definition of a conversion.

```
ALTER CONVERSION name RENAME TO new_name
ALTER CONVERSION name OWNER TO new_owner
```

ALTER DATABASE

Change a database.

```
ALTER DATABASE name SET parameter { TO | = } { value | DEFAULT }
ALTER DATABASE name RESET parameter
ALTER DATABASE name RENAME TO new_name
ALTER DATABASE name OWNER TO new_owner
```

ALTER DOMAIN

Change the definition of a domain.

```
ALTER DOMAIN name
    { SET DEFAULT expression | DROP DEFAULT }
ALTER DOMAIN name
    { SET | DROP } NOT NULL
ALTER DOMAIN name
    ADD domain_constraint
ALTER DOMAIN name
    DROP CONSTRAINT constraint_name [ RESTRICT | CASCADE ]
ALTER DOMAIN name
    OWNER TO new_owner
```

ALTER FUNCTION

Change the definition of a function.

```
ALTER FUNCTION name ( [ type [, ...] ] ) RENAME TO new_name
ALTER FUNCTION name ( [ type [, ...] ] ) OWNER TO new_owner
```

ALTER GROUP

Change a user group.

```
ALTER GROUP groupname ADD USER username [, ... ]
ALTER GROUP groupname DROP USER username [, ... ]
ALTER GROUP groupname RENAME TO new_name
```

ALTER INDEX

Change the definition of an index.

```
ALTER INDEX name
    action [, ... ]
ALTER INDEX name
    RENAME TO new_name
```

Where `action` is one of:

```
    OWNER TO new_owner
    SET TABLESPACE indexspace_name
```

ALTER LANGUAGE

Change the definition of a procedural language.

```
ALTER LANGUAGE name RENAME TO new_name
```

ALTER OPERATOR

Change the definition of an operator.

```
ALTER OPERATOR name ( { lefttype | NONE } , { righttype | NONE } )
    OWNER TO new_owner
```

ALTER OPERATOR CLASS

Change the definition of an operator class.

```
ALTER OPERATOR CLASS name USING index_method RENAME TO new_name
ALTER OPERATOR CLASS name USING index_method OWNER TO new_owner
```

ALTER SCHEMA

Change the definition of a schema.

```
ALTER SCHEMA name RENAME TO new_name
ALTER SCHEMA name OWNER TO new_owner
```

ALTER SEQUENCE

Change the definition of a sequence generator.

```
ALTER SEQUENCE name [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ]
    [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ RESTART [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
```

ALTER TABLE

Change the definition of a table.

```
ALTER TABLE [ ONLY ] name [ * ]
    action [, ... ]
ALTER TABLE [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column TO new_column
ALTER TABLE name
    RENAME TO new_name
```

Where action is one of:

```
ADD [ COLUMN ] column_type [ column_constraint [ ... ] ]
DROP [ COLUMN ] column [ RESTRICT | CASCADE ]
ALTER [ COLUMN ] column TYPE type [ USING expression ]
ALTER [ COLUMN ] column SET DEFAULT expression
ALTER [ COLUMN ] column DROP DEFAULT
ALTER [ COLUMN ] column { SET | DROP } NOT NULL
ALTER [ COLUMN ] column SET STATISTICS integer
ALTER [ COLUMN ] column SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
ADD table_constraint
DROP CONSTRAINT constraint_name [ RESTRICT | CASCADE ]
CLUSTER ON index_name
SET WITHOUT CLUSTER
SET WITHOUT OIDS
OWNER TO new_owner
SET TABLESPACE tablespace_name
```

ALTER TABLESPACE

Change the definition of a tablespace.

```
ALTER TABLESPACE name RENAME TO new_name
ALTER TABLESPACE name OWNER TO new_owner
```

ALTER TRIGGER

Change the definition of a trigger.

```
ALTER TRIGGER name ON table RENAME TO new_name
```

ALTER TYPE

Change the definition of a type.

```
ALTER TYPE name OWNER TO new_owner
```

ALTER USER

Change a database user account.

```
ALTER USER name [ [ WITH ] option [ ... ] ]
ALTER USER name RENAME TO new_name
ALTER USER name SET parameter { TO | = } { value | DEFAULT }
ALTER USER name RESET parameter
```

Where option can be:

```
[ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
| CREATEDB | NOCREATEDB
| CREATEUSER | NOCREATEUSER
| VALID UNTIL 'abstime'
```

ANALYZE

Collect statistics about a database.

```
ANALYZE [ VERBOSE ] [ table [ (column [, ...] ) ] ]
```

BEGIN

Start a transaction block.

```
BEGIN [ WORK | TRANSACTION ] [ transaction_mode [, ...] ]
```

Where `transaction_mode` is one of:

```
    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED
                    | READ UNCOMMITTED }
    READ WRITE | READ ONLY
```

CHECKPOINT

Force a transaction log checkpoint.

```
CHECKPOINT
```

CLOSE

Close a cursor.

```
CLOSE name
```

CLUSTER

Cluster a table according to an index.

```
CLUSTER index_name ON table_name
CLUSTER table_name
CLUSTER
```

COMMENT

Define or change the comment of an object.

```
COMMENT ON
{
  TABLE object_name |
  COLUMN table_name.column_name |
  AGGREGATE agg_name (agg_type) |
  CAST (source_type AS target_type) |
  CONSTRAINT constraint_name ON table_name |
  CONVERSION object_name |
  DATABASE object_name |
  DOMAIN object_name |
  FUNCTION func_name (arg1_type, arg2_type, ...) |
```

```
  INDEX object_name |
  LARGE OBJECT large_object_oid |
  OPERATOR op (left_operand_type, right_operand_type) |
  OPERATOR CLASS object_name USING index_method |
  [ PROCEDURAL ] LANGUAGE object_name |
  RULE rule_name ON table_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  TRIGGER trigger_name ON table_name |
  TYPE object_name |
  VIEW object_name
} IS 'text'
```

COMMIT

Commit the current transaction.

```
COMMIT [ WORK | TRANSACTION ]
```

COPY

Copy data between a file and a table.

```
COPY table_name [ ( column [, ...] ) ]
    FROM { 'filename' | STDIN }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'delimiter' ]
          [ NULL [ AS ] 'null string' ]
          [ CSV [ QUOTE [ AS ] 'quote' ]
                [ ESCAPE [ AS ] 'escape' ]
                [ FORCE NOT NULL column [, ...] ]

COPY table_name [ ( column [, ...] ) ]
    TO { 'filename' | STDOUT }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'delimiter' ]
          [ NULL [ AS ] 'null string' ]
          [ CSV [ QUOTE [ AS ] 'quote' ]
                [ ESCAPE [ AS ] 'escape' ]
                [ FORCE QUOTE column [, ...] ]
```

CREATE AGGREGATE

Define a new aggregate function.

```
CREATE AGGREGATE name (
    BASETYPE = input_data_type,
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , FINALFUNC = ffunc ]
    [ , INITCOND = initial_condition ]
)
```

CREATE CAST

Define a new cast.

```
CREATE CAST (source_type AS target_type)
    WITH FUNCTION func_name (arg_types)
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (source_type AS target_type)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

CREATE CONSTRAINT TRIGGER

Define a new constraint trigger.

```
CREATE CONSTRAINT TRIGGER name
    AFTER events ON
    table_name constraint attributes
    FOR EACH ROW EXECUTE PROCEDURE func_name ( args )
```

CREATE CONVERSION

Define a new conversion.

```
CREATE [DEFAULT] CONVERSION name
    FOR source_encoding TO dest_encoding FROM func_name
```

CREATE DATABASE

Create a new database.

```
CREATE DATABASE name
    [ [ WITH ] [ OWNER [=] db_owner ]
           [ TEMPLATE [=] template ]
           [ ENCODING [=] encoding ]
           [ TABLESPACE [=] tablespace ] ]
```

CREATE DOMAIN

Define a new domain.

```
CREATE DOMAIN name [AS] data_type
    [ DEFAULT expression ]
    [ constraint [ ... ] ]
```

Where `constraint` is:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | CHECK (expression) }
```

CREATE FUNCTION

Define a new function.

```
CREATE [ OR REPLACE ] FUNCTION name ( [ [ arg_name ] arg_type [, ...] ] )
    RETURNS ret_type
  { LANGUAGE lang_name
    | IMMUTABLE | STABLE | VOLATILE
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
    [ WITH ( attribute [, ...] ) ]
```

CREATE GROUP

Define a new user group.

```
CREATE GROUP name [ [ WITH ] option [ ... ] ]
```

Where `option` can be:

```
     SYSID gid
   | USER  username [, ...]
```

CREATE INDEX

Define a new index.

```
CREATE [ UNIQUE ] INDEX name ON table [ USING method ]
    ( { column | ( expression ) } [ opclass ] [, ...] )
    [ TABLESPACE tablespace ]
    [ WHERE predicate ]
```

CREATE LANGUAGE

Define a new procedural language.

```
CREATE [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
    HANDLER call_handler [ VALIDATOR val_function ]
```

CREATE OPERATOR

Define a new operator.

```
CREATE OPERATOR name (
    PROCEDURE = func_name
    [, LEFTARG = left_type ] [, RIGHTARG = right_type ]
    [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
    [, RESTRICT = res_proc ] [, JOIN = join_proc ]
    [, HASHES ] [, MERGES ]
    [, SORT1 = left_sort_op ] [, SORT2 = right_sort_op ]
    [, LTCMP = less_than_op ] [, GTCMP = greater_than_op ]
)
```

CREATE OPERATOR CLASS

Define a new operator class.

```
CREATE OPERATOR CLASS name [ DEFAULT ] FOR TYPE data_type
  USING index_method AS
  {  OPERATOR strategy_number operator_name [ ( op_type, op_type ) ] [ RECHECK ]
   | FUNCTION support_number func_name ( argument_type [, ...] )
   | STORAGE storage_type
  } [, ... ]
```

CREATE RULE

Define a new rewrite rule.

```
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

CREATE SCHEMA

Define a new schema.

```
CREATE SCHEMA schema_name
  [ AUTHORIZATION username ] [ schema_element [ ... ] ]
CREATE SCHEMA AUTHORIZATION username
  [ schema_element [ ... ] ]
```

CREATE SEQUENCE

Define a new sequence generator.

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE name
    [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ]
    [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
```

CREATE TABLE

Define a new table.

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name (
  { column_name data_type [ DEFAULT default_expr ] [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE parent_table [ { INCLUDING | EXCLUDING } DEFAULTS ] }  [, ... ]
)
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace ]
```

Where `column_constraint` is:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  UNIQUE [ USING INDEX TABLESPACE tablespace ] |
  PRIMARY KEY [ USING INDEX TABLESPACE tablespace ] |
  CHECK (expression) |
  REFERENCES ref_table [ ( ref_column ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

And `table_constraint` is:

```
[ CONSTRAINT constraint_name ]
{ UNIQUE ( column_name [, ... ] ) [ USING INDEX TABLESPACE tablespace ] |
  PRIMARY KEY ( column_name [, ... ] ) [ USING INDEX TABLESPACE tablespace ] |
  CHECK ( expression ) |
  FOREIGN KEY ( column_name [, ... ] )
    REFERENCES ref_table [ ( ref_column [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

CREATE TABLE AS

Define a new table from the results of a query.

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name
    [ (column_name [, ...] ) ] [ [ WITH | WITHOUT ] OIDS ]
    AS query
```

CREATE TABLESPACE

Define a new tablespace.

```
CREATE TABLESPACE tablespace_name [ OWNER username ] LOCATION 'directory'
```

CREATE TRIGGER

Define a new trigger.

```
CREATE TRIGGER name { BEFORE | AFTER } { event [ OR ... ] }
    ON table [ FOR [ EACH ] { ROW | STATEMENT } ]
    EXECUTE PROCEDURE func_name ( arguments )
```

CREATE TYPE

Define a new data type.

```
CREATE TYPE name AS
    ( attribute_name data_type [, ... ] )

CREATE TYPE name (
    INPUT = input_function,
    OUTPUT = output_function
    [ , RECEIVE = receive_function ]
    [ , SEND = send_function ]
    [ , ANALYZE = analyze_function ]
    [ , INTERNALLENGTH = { internal_length | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignment ]
    [ , STORAGE = storage ]
    [ , DEFAULT = default ]
    [ , ELEMENT = element ]
    [ , DELIMITER = delimiter ]
)
```

CREATE USER

Define a new database user account.

```
CREATE USER name [ [ WITH ] option [ ... ] ]
```

Where option can be:

```
      SYSID uid
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | CREATEDB | NOCREATEDB
    | CREATEUSER | NOCREATEUSER
    | IN GROUP group_name [, ...]
    | VALID UNTIL 'abs_time'
```

CREATE VIEW

Define a new view.

```
CREATE [ OR REPLACE ] VIEW name [ ( column_name [, ...] ) ] AS query
```

DEALLOCATE

Deallocate a prepared statement.

```
DEALLOCATE [ PREPARE ] plan_name
```

DECLARE

Define a cursor.

```
DECLARE name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]
    CURSOR [ { WITH | WITHOUT } HOLD ] FOR query
    [ FOR { READ ONLY | UPDATE [ OF column [, ...] ] } ]
```

DELETE

Delete rows of a table.

```
DELETE FROM [ ONLY ] table [ WHERE condition ]
```

DROP AGGREGATE

Remove an aggregate function.

```
DROP AGGREGATE name ( type ) [ CASCADE | RESTRICT ]
```

DROP CAST

Remove a cast.

```
DROP CAST (source_type AS target_type) [ CASCADE | RESTRICT ]
```

DROP CONVERSION

Remove a conversion.

```
DROP CONVERSION name [ CASCADE | RESTRICT ]
```

DROP DATABASE

Remove a database.

```
DROP DATABASE name
```

DROP DOMAIN

Remove a domain.

```
DROP DOMAIN name [, ...]  [ CASCADE | RESTRICT ]
```

DROP FUNCTION

Remove a function.

```
DROP FUNCTION name ( [ type [, ...] ] ) [ CASCADE | RESTRICT ]
```

DROP GROUP

Remove a user group.

```
DROP GROUP name
```

DROP INDEX

Remove an index.

```
DROP INDEX name [, ...] [ CASCADE | RESTRICT ]
```

DROP LANGUAGE

Remove a procedural language.

```
DROP [ PROCEDURAL ] LANGUAGE name [ CASCADE | RESTRICT ]
```

DROP OPERATOR

Remove an operator.

```
DROP OPERATOR name ( { left_type | NONE } , { right_type | NONE } )
  [ CASCADE | RESTRICT ]
```

DROP OPERATOR CLASS

Remove an operator class.

```
DROP OPERATOR CLASS name USING index_method [ CASCADE | RESTRICT ]
```

DROP RULE

Remove a rewrite rule.

```
DROP RULE name ON relation [ CASCADE | RESTRICT ]
```

DROP SCHEMA

Remove a schema.

```
DROP SCHEMA name [, ...] [ CASCADE | RESTRICT ]
```

DROP SEQUENCE

Remove a sequence.

```
DROP SEQUENCE name [, ...] [ CASCADE | RESTRICT ]
```

DROP TABLE

Remove a table.

```
DROP TABLE name [, ...] [ CASCADE | RESTRICT ]
```

DROP TABLESPACE

Remove a tablespace.

```
DROP TABLESPACE tablespace_name
```

DROP TRIGGER

Remove a trigger.

```
DROP TRIGGER name ON table [ CASCADE | RESTRICT ]
```

DROP TYPE

Remove a data type.

```
DROP TYPE name [, ...] [ CASCADE | RESTRICT ]
```

DROP USER

Remove a database user account.

```
DROP USER name
```

DROP VIEW

Remove a view.

```
DROP VIEW name [, ...] [ CASCADE | RESTRICT ]
```

END

Commit the current transaction.

```
END [ WORK | TRANSACTION ]
```

EXECUTE

Execute a prepared statement.

```
EXECUTE plan_name [ (parameter [, ...] ) ]
```

EXPLAIN

Show the execution plan of a statement.

```
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
```

FETCH

Retrieve rows from a query using a cursor.

```
FETCH [ direction { FROM | IN } ] cursor_name
```

Where `direction` can be empty or one of:

```
    NEXT
    PRIOR
    FIRST
    LAST
    ABSOLUTE count
    RELATIVE count
    count
    ALL
    FORWARD
    FORWARD count
    FORWARD ALL
    BACKWARD
    BACKWARD count
    BACKWARD ALL
```

GRANT

Define access privileges.

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES | TRIGGER }
    [,...] | ALL [ PRIVILEGES ] }
    ON [ TABLE ] table_name [, ...]
    TO { username | GROUP group_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | TEMPORARY | TEMP } [,...] | ALL [ PRIVILEGES ] }
    ON DATABASE db_name [, ...]
    TO { username | GROUP group_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespace_name [, ...]
    TO { username | GROUP group_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTION func_name ([type, ...]) [, ...]
    TO { username | GROUP group_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE lang_name [, ...]
    TO { username | GROUP group_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | USAGE } [,...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schema_name [, ...]
    TO { username | GROUP group_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

INSERT

Create new rows in a table.

```
INSERT INTO table [ ( column [, ...] ) ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) | query }
```

LISTEN

Listen for a notification.

```
LISTEN name
```

LOAD

Load or reload a shared library file.

```
LOAD 'filename'
```

LOCK

Lock a table.

```
LOCK [ TABLE ] name [, ...] [ IN lock_mode MODE ] [ NOWAIT ]
```

Where `lock_mode` is one of:

```
    ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
    | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

MOVE

Position a cursor.

```
MOVE [ direction { FROM | IN } ] cursor_name
```

NOTIFY

Generate a notification.

```
NOTIFY name
```

PREPARE

Prepare a statement for execution.

```
PREPARE plan_name [ (data_type [, ...] ) ] AS statement
```

REINDEX

Rebuild indexes.

```
REINDEX { DATABASE | TABLE | INDEX } name [ FORCE ]
```

RELEASE SAVEPOINT

Destroy a previously defined savepoint.

```
RELEASE [ SAVEPOINT ] savepoint_name
```

RESET

Restore the value of a runtime parameter to the default value.

```
RESET name
RESET ALL
```

REVOKE

Remove access privileges.

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES | TRIGGER }
    [,...] | ALL [ PRIVILEGES ] }
    ON [ TABLE ] table_name [, ...]
    FROM { username | GROUP group_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | TEMPORARY | TEMP } [,...] | ALL [ PRIVILEGES ] }
    ON DATABASE db_name [, ...]
    FROM { username | GROUP group_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespace_name [, ...]
    FROM { username | GROUP group_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTION func_name ([type, ...]) [, ...]
    FROM { username | GROUP group_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE lang_name [, ...]
    FROM { username | GROUP group_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | USAGE } [,...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schema_name [, ...]
    FROM { username | GROUP group_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

ROLLBACK

Abort the current transaction.

```
ROLLBACK [ WORK | TRANSACTION ]
```

ROLLBACK TO SAVEPOINT

Roll back to a savepoint.

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] savepoint_name
```

SAVEPOINT

Define a new savepoint within the current transaction.

```
SAVEPOINT savepoint_name
```

SELECT

Retrieve rows from a table or view.

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ AS output_name ] [, ...]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
    [ FOR UPDATE [ OF table_name [, ...] ] ]
```

Where `from_item` can be one of:

```
    [ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    ( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
    function_name ( [ argument [, ...] ] )
        [ AS ] alias [ ( column_alias [, ...] | column_definition [, ...] ) ]
    function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
    from_item [ NATURAL ] join_type from_item
        [ ON join_condition | USING ( join_column [, ...] ) ]
```

SELECT INTO

Define a new table from the results of a query.

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ AS output_name ] [, ...]
    INTO [ TEMPORARY | TEMP ] [ TABLE ] new_table
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]
    [ FOR UPDATE [ OF table_name [, ...] ] ]
```

SET

Change a runtime parameter.

```
SET [ SESSION | LOCAL ] name { TO | = } { value | 'value' | DEFAULT }
SET [ SESSION | LOCAL ] TIME ZONE { time_zone | LOCAL | DEFAULT }
```

SET CONSTRAINTS

Set constraint checking modes for the current transaction.

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

SET SESSION AUTHORIZATION

Set the session user identifier and the current user identifier of the current session.

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION username
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

SET TRANSACTION

Set the characteristics of the current transaction.

```
SET TRANSACTION transaction_mode [, ...]
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]
```

Where `transaction_mode` is one of:

```
    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED
                    | READ UNCOMMITTED }
    READ WRITE | READ ONLY
```

SHOW

Show the value of a runtime parameter.

```
SHOW name
SHOW ALL
```

START TRANSACTION

Start a transaction block.

```
START TRANSACTION [ transaction_mode [, ...] ]
```

Where `transaction_mode` is one of:

```
    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED
                    | READ UNCOMMITTED }
    READ WRITE | READ ONLY
```

TRUNCATE

Empty a table.

```
TRUNCATE [ TABLE ] name
```

UNLISTEN

Stop listening for a notification.

```
UNLISTEN { name | * }
```

UPDATE

Update rows of a table.

```
UPDATE [ ONLY ] table SET column = { expression | DEFAULT } [, ...]
    [ FROM from_list ]
    [ WHERE condition ]
```

VACUUM

Garbage-collect and optionally analyze a database.

```
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ table ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ table [ (column [, ...] ) ] ]
```

APPENDIX D

psql Reference

This appendix defines the `psql` command-line options and internal commands. This information is taken from the `psql` command-line tool's internal help.

Command-Line Options

`psql` has the following command-line usage:

```
psql [options] [dbname [username]]
```

Table D-1 shows the command-line options.

Table D-1. psql Command-Line Options

Option	Meaning
`-?, --help`	Show help, then exit
`-a, --echo_all`	Echo all input from script
`-A, --no-align`	Unaligned table output mode (`-P format=unaligned`)
`-c, --command <command>`	Run only single command (SQL or internal) and exit
`-d, --dbname <dbname>`	Specify database name to connect to (default: current username)
`-e, --echo-queries`	Echo commands sent to server
`-E, --echo-hidden`	Display queries that internal commands generate
`-f, --file <filename>`	Execute commands from file, then exit
`-F, --field-separator <string>`	Set field separator (default: "|") (`-P fieldsep=`)
`-h, --host <hostname>`	Database server host or socket directory (default: "`local socket`")
`-H, --html`	HTML table output mode (`-P format=html`)
`-l, --list`	List available databases, then exit
`-n`	Disable enhanced command-line editing (readline)
`-o, --output <filename>`	Send query results to file (use `-o |program` for a pipe)
`-p, --port <port>`	Database server port (default: `5432`)

Table D-1. psql Command-Line Options (Continued)

Option	Meaning
`-P, --pset var[=arg]`	Set printing option `var` to `arg` (see `\pset` command)
`-q, --quiet`	Run quietly (no messages, only query output)
`-R, --record-separator <string>`	Set record separator (default: newline) (`-P recordsep=`)
`-s, --single-step`	Single-step mode (confirm each query)
`-S, --single-line`	Single-line mode (end of line terminates SQL command)
`-t, --tuples-only`	Print rows only (`-P tuples_only`)
`-T, --table-attr <text>`	Set HTML table tag attributes (width, border) (`-P tableattr=`)
`-U, --username <name>`	Database username (default: current username)
`-v, --set, --variable name=value`	Set `psql` variable `name` to `value`
`-V, --version`	Output version information, then exit
`-W, --password`	Prompt for password (should happen automatically)
`-x, --expanded`	Turn on expanded table output (`-P expanded`)
`-X, --no-psqlrc`	Do not read startup file (`~/.psqlrc`)

Internal Commands

Table D-2 lists the `psql` internal commands.

Table D-2. psql Internal Commands

Command	Meaning
`\! <command>`	Execute command in shell or start interactive shell
`\a`	Toggle between unaligned and aligned output mode
`\c[onnect] [<dbname>|- [<user>]]`	Connect to new database
`\C <string>`	Set table title, or unset if none
`\cd <dir>`	Change the current working directory
`\copy`	Perform SQL `COPY` with data stream to the client host
`\copyright`	Show PostgreSQL usage and distribution terms
`\d, \d+ <name>`	Describe table, index, sequence, or view (with + gives expanded output)
`\d{t|i|s|v|S} <pattern>`	List tables/indexes/sequences/views/system tables
`\da <pattern>`	List aggregate functions
`\db, \db+ <pattern>`	List tablespaces (with + gives expanded output)

Table D-2. psql Internal Commands (Continued)

Command	Meaning
\dc <pattern>	List conversions
\dC	List casts
\dd <pattern>	Show comment for object
\dD <pattern>	List domains
\df, \df+ <pattern>	List functions (with + gives expanded output)
\dg <pattern>	List groups
\dn, \dn+ <pattern>	List schemas (with + gives expanded output)
\do <name>	List operators
\dl	List large objects, same as \lo_list
\dp <pattern>	List table, view, and sequence access privileges
\dT, \dT+ <pattern>	List data types (with + gives expanded output)
\du <pattern>	List users
\e[dit] [<file>]	Edit the query buffer (or file) with external editor
\echo <string>	Write string to standard output
\encoding [<encoding>]	Show or set client encoding
\f [<string>]	Show or set field separator for unaligned query output
\g <file>	Send query buffer to server (and results to file or |program for a pipe)
\h[elp] [<name>]	Help on syntax of SQL commands; * for all commands
\H	Toggle HTML output mode
\i <file>	Execute commands from file
\l[ist], \l[ist]+	List all databases (with + gives expanded output)
\lo_export <loboid> <file>	Export large objects
\lo_import <file> [<comment>]	Import large objects
\lo_list	List large objects
\lo_unlink <loboid>	Delete large objects
\o <file>	Send all query results to file or |program for a pipe
\p	Show the contents of the query buffer
\pset name [value]	Set table output option (name := {format|border|expanded|fieldsep|footer|null|recordsep|tuples_only|title|tableattr|pager})
\q	Quit psql
\qecho <string>	Write string to query output stream (see \o)

Table D-2. psql Internal Commands (Continued)

Command	Meaning
\r	Reset (clear) the query buffer
\s [<file>]	Display history or save it to file
\set [<name> [<value>]]	Set internal variable, or list all if no parameters
\t	Show only rows
\T [<string>]	Set HTML <table> tag attributes, or unset if none
\timing	Toggle timing of commands
\unset <name>	Unset (delete) internal variable
\w <file>	Write query buffer to file
\x	Toggle expanded output
\z <pattern>	List table, view, and sequence access privileges (same as \dp)

APPENDIX E

Database Schema and Tables

The database schema used in the examples in this book is a simplified customer/orders/items database, as shown in Figure E-1.

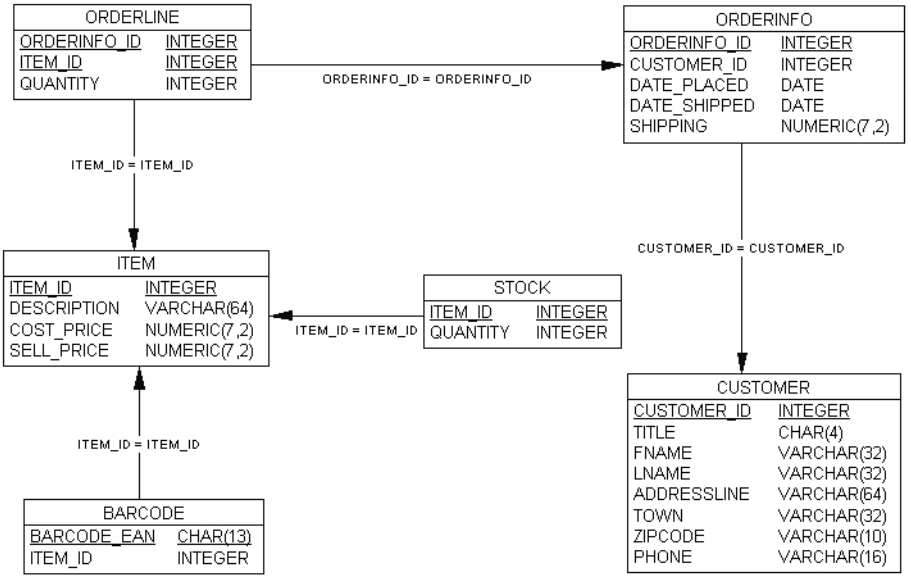

Figure E-1. *Final schema design*

The tables need to be created in an appropriate order so that dependent tables are created first, because of the foreign key constraints (see Chapter 8). This is the same order to be followed as data is inserted into the tables. The appropriate order is as follows:

- `customer`
- `orderinfo`
- `item`
- `orderline`
- `stock`
- `barcode`

The SQL to create the final version of this sample database, `bpfinal`, including the foreign key constraints follows. This code can be found in the download bundle available from the Downloads section of the Apress web site (`http://www.apress.com`) as `create_tables-bpfinal.sql`. The simplified version, `bpsimple` (see Chapter 3), excluding the foreign key constraints, can be found in `create_tables-bpsimple.sql`.

The download code bundle has the table-population commands, `pop-all-tables.sql`, in an appropriate order ready for populating either schema.

Customer table

```
create table customer
(
    customer_id                 serial,
    title                       char(4),
    fname                       varchar(32),
    lname                       varchar(32) not null,
    addressline                 varchar(64),
    town                        varchar(32),
    zipcode                     char(10) not null,
    phone                       varchar(16),
    CONSTRAINT                  customer_pk PRIMARY KEY(customer_id)
);
```

Orderinfo table

```
create table orderinfo
(
    orderinfo_id                serial,
    customer_id                 integer not null,
    date_placed                 date not null,
    date_shipped                date,
    shipping                    numeric(7,2) ,
    CONSTRAINT                  orderinfo_pk PRIMARY KEY(orderinfo_id),
    CONSTRAINT orderinfo_customer_id_fk FOREIGN KEY(customer_id) REFERENCES
customer(customer_id)
);
```

Item table

```
create table item
(
    item_id                     serial,
    description                 varchar(64) not null,
    cost_price                  numeric(7,2),
    sell_price                  numeric(7,2),
    CONSTRAINT                  item_pk PRIMARY KEY(item_id)
);
```

Orderline table

```
create table orderline
(
    orderinfo_id                    integer not null,
    item_id                         integer not null,
    quantity                        integer not null,
    CONSTRAINT                      orderline_pk PRIMARY KEY(orderinfo_id,
item_id),
    CONSTRAINT orderline_orderinfo_id_fk FOREIGN KEY(orderinfo_id) REFERENCES
orderinfo(orderinfo_id),
    CONSTRAINT orderline_item_id_fk FOREIGN KEY(item_id) REFERENCES item(item_id)
);
```

Stock table

```
create table stock
(
    item_id                         integer not null,
    quantity                        integer  not null,
    CONSTRAINT                      stock_pk PRIMARY KEY(item_id),
    CONSTRAINT stock_item_id_fk FOREIGN KEY(item_id) REFERENCES item(item_id)
);
```

Barcode table

```
create table barcode
(
    barcode_ean                     char(13) not null,
    item_id                         integer not null,
    CONSTRAINT                      barcode_pk PRIMARY KEY(barcode_ean),
    CONSTRAINT barcode_item_id_fk FOREIGN KEY(item_id) REFERENCES item(item_id)
);
```

APPENDIX F

Large Objects Support in PostgreSQL

Traditionally, databases have been able to store data in a limited number of forms, usually as numeric values (integers, floating point, and fixed point) and text strings. Often, the size of the text data is limited. In the past, even PostgreSQL enforced a limit of a few thousand characters as the maximum length of a field.

It might be useful to be able to create a database application that can handle *arbitrary* unstructured data formats, such as images. As an example of how to handle large data items, we will consider how we might add photographs to a database. There are several ways we could do this:

- Use links into a file system or the Web
- Use encoded long text strings
- Use BLOBs

We will look at each of these approaches in this appendix, using the sample database we built in this book, `bpfinal` (see Chapter 8 and Appendix E for details on the `bpfinal` schema). We will see how we might add images of products, so that our web-based interface could provide an online catalog.

Using Links

Our first option is to avoid storing images in the physical database at all. The idea is to place all of the images in the normal filing system of a server, which may be the database server, a file sharing server, or a web server. The database itself then only needs to contain a text link to the file. Any client application would follow the link to retrieve the image.

We need to create an additional table in the database to store the image links. It is very similar to the `stock` table in that we are providing additional information for each item we carry:

```
CREATE TABLE image
(
    item_id     integer NOT NULL,
    picture     varchar(512),
    CONSTRAINT  image_pk PRIMARY KEY(item_id),
    CONSTRAINT  image_item_id_fk FOREIGN KEY(item_id)
                REFERENCES item(item_id)
);
```

Here, we have added constraints to ensure that we can add images only for items that exist.

Now we can update the `image` table with links to product photographs:

```
INSERT INTO image VALUES (3, 'http://server/images/rubik.jpg');
INSERT INTO image VALUES (9, '//server/images/coin.bmp');
INSERT INTO image VALUES (5, '/mnt/server/images/frame.png');
```

This solution has both advantages and disadvantages. Storing links rather than pictures means that the database size is kept to a minimum and applications will be portable to other database systems, as we have not used any esoteric features to handle images. Furthermore, retrieving the actual images will also be very fast, as reading a file from the file system will typically be many times faster than querying a database. Also we have a wide choice of locations for storing the images. In this example, we have used the following:

- A URL to provide a link into a web server
- A UNC file reference for a Windows file server
- A reference to an NFS-mounted UNIX server

However, by using links we are unable to enforce referential integrity in the database. If the images that are stored elsewhere are changed or deleted, the database is not automatically updated. To back up our system, we will need to attend to image files on the file system (and elsewhere), as well as the database itself. We must also ensure that the links we use are in a form that all possible clients can use. For example, the NFS form requires that all clients access the shared files in the same way and have access rights on the server.

Using Encoded Text Strings

In PostgreSQL 7.1, the limit on the size of a field was raised to 1GB. For all practical purposes, this is effectively unlimited. We could consider using the `text` type to store the images in the database directly. This is possible, if a little tricky.

Images are in general binary data, not well suited to a `character` type. So, we need to encode the image in some way, perhaps by using hexadecimal or MIME encoding. Handling the very long strings that result may also cause a problem with limits imposed by client applications, network transport mechanisms, or ODBC drivers. The storage space needed for encoded strings will also be up to double the size of the binary image file.

Using BLOBs

One of the wide variety of data types PostgreSQL currently supports is the binary large object, or BLOB, which is suitable for storing large data items. This allows us to create a database application that can handle *arbitrary* unstructured data formats, such as images. Thus, we can store our image data, or any large or binary data object, in a PostgreSQL database by using BLOBs.

PostgreSQL supports a column type of `oid`, which is an object identifier, a reference to arbitrary data. These are used to manage the BLOBs and can be used to transfer the contents of any file into the database, and to extract an object from the database into a file. They can therefore be used to handle our product images, or any other data that we might wish to store.

We can modify the sample `image` table definition from earlier in this appendix to use BLOBs by specifying `oid` as the image data type:

```
CREATE TABLE image
(
    item_id     integer NOT NULL,
    picture     oid,
    CONSTRAINT  image_pk PRIMARY KEY(item_id),
    CONSTRAINT  image_item_id_fk FOREIGN KEY(item_id) REFERENCES item(item_id)
);
```

Importing and Exporting Images

PostgreSQL provides a number of functions that can be used in SQL queries for inserting into, retrieving from, and deleting BLOB data in a table.

To add an image to the table, we can use the SQL function `lo_import`, like this:

```
INSERT INTO image VALUES (3, lo_import('/tmp/image.jpg'));
```

The contents of the specified file are read into a BLOB object and stored in the database. The `image` table will now have a non-`NULL` `oid` that references the BLOB:

```
bpfinal=# SELECT * FROM image;
 item_id | picture
---------+---------
       3 |  163055
(1 row)

bpfinal=#
```

We can see all of the large objects stored in the database by using a `psql` internal command, `\lo_list`, or `\dl`, to list them:

```
bpfinal=# \dl
    Large objects
   ID   | Description
--------+-------------
 163055 |
(1 row)

bpfinal=#
```

Large objects are retrieved from the database using `lo_export`, which writes a file containing the contents of the BLOB:

```
bpfinal=# SELECT lo_export(picture, '/tmp/image2.jpg')
bpfinal-# FROM image WHERE item_id = 3;
 lo_export
-----------
         1
(1 row)

bpfinal=#
```

We can delete a large object from the database with `lo_unlink`:

```
bpfinal=# SELECT lo_unlink(picture) FROM image WHERE item_id = 3;
 lo_unlink
-----------
         1
(1 row)

bpfinal=# \dl
  Large objects
 ID | Description
----+-------------
(0 rows)

bpfinal=#
```

We must be careful when we delete large objects, as any references to the BLOB will remain:

```
bpfinal=# SELECT * FROM image;
 item_id | picture
---------+---------
       3 |  163055
(1 row)

bpfinal=#
```

As operations on a large object and its object identifier are essentially decoupled, we must make sure to take care of both when manipulating BLOBs. So, when deleting a BLOB, we should set the object reference to `NULL` to prevent errors in our client applications:

```
bpfinal=# UPDATE image SET picture=NULL WHERE item_id = 3;
```

In the examples here, we have used `psql` to manipulate binary objects. It is important to understand that the import and export functions `lo_import` and `lo_export` are executed by the back-end database server, not by `psql`. Any client application using SQL statements can use these SQL functions to add and update BLOBs.

There are three caveats when importing and exporting BLOBs:

- As the import is performed by the server, the files read and written by the `import` and `export` must be specified using a path and filename that are accessible to the server, rather than the client. If in `psql` we had simply said this:

  ```
  INSERT INTO image VALUES (3, lo_import('image.jpg'));
  ```

  and expected PostgreSQL to insert the contents of a file in the current directory, we would have received an error message. This is because the import fails to find the file. We need to arrange that files for import are (temporarily) placed in a location that the server can access. Similarly, we need to use full filenames for exporting binary objects.

- Exported files must be placed in a directory that the server user can write; that is, a location where the operating system user `postgres` has permission to create files.

- All large object manipulation must take place within a SQL transaction—between `BEGIN` and `COMMIT` or `END` statements. By default, `psql` executes each SQL statement in its own transaction, so this is not a problem, but client applications that perform imports and exports must be written with transactions.

Remote Importing and Exporting

Because the SQL functions `lo_import` and `lo_export` use the server file system, using them can be inconvenient when creating BLOBs. However, `psql` contains internal commands that can be used to import and export binary objects from a remote client machine.

We can add a BLOB to the database by using `\lo_import` and passing it a local filename. In this case, files in the current directory will work just fine, and we can see the object listed with `\lo_list`:

```
bpfinal=# \lo_import image.jpg
lo_import 163059
bpfinal=# \lo_list
    Large objects
   ID   | Description
--------+-------------
 163059 |
(1 row)

bpfinal=#
```

Now we need to associate the BLOB with the `image` table by updating the appropriate row:

```
bpfinal=# UPDATE image SET picture=163059 WHERE item_id = 3;
UPDATE 1
bpfinal=# SELECT * FROM image;
 item_id | picture
---------+---------
       3 |  163059
(1 row)

bpfinal=#
```

We can extract a BLOB with \lo_export, specifying the required object identifier and a file to write. Again, a local filename is fine:

```
bpfinal=# \lo_export 163059 image2.jpg
lo_export

bpfinal=#
```

Finally, we can delete a large object with \lo_unlink:

```
bpfinal=# \lo_unlink 163059
lo_unlink 163059

bpfinal=#
```

Programming BLOBs

As you might expect, it is possible to use BLOB import and export functions from the programming languages supported by PostgreSQL.

From C, using the libpq library (see Chapter 13), we can use the functions lo_import, lo_export, and lo_unlink in much the same way as described in the preceding sections:

```
Oid lo_import(PGconn *conn, const char *filename);
int lo_export(PGconn *conn, Oid lobjId, const char *filename);
int lo_unlink(PGconn *conn, Oid lobjId);
```

Here is a sample program that imports an image file into the database. Note that the large object functions must be called within a transaction:

```
#include <stdlib.h>
#include <libpq-fe.h>

int main()
{
  PGconn *myconnection = PQconnectdb("");
  PGresult *res;
  Oid blob;

  if(PQstatus(myconnection) == CONNECTION_OK)
    printf("connection made\n");
  else
    printf("connection failed\n");

  res = PQexec(myconnection, "begin");
  PQclear(res);

  blob = lo_import(myconnection, "image.jpg");
  printf("import returned oid %d\n", blob);
```

```
  res = PQexec(myconnection, "end");
  PQclear(res);

  PQfinish(myconnection);
  return EXIT_SUCCESS;
}
```

When we compile and run the program, we can see the new binary object identifier reported. (This program is included in the sample programs for Chapter 13.)

```
$ make import
cc -I/usr/local/pgsql/include  -L/usr/local/pgsql/lib -lpq  import.c   -o import
$ PGDATABASE=bpfinal ./import
connection made
import returned oid 163066
$
```

BLOBs can be imported and exported from other languages in similar ways.

For finer control over large object access, PostgreSQL provides a suite of low-level functions akin to open, read, write, and others for ordinary files:

```
int lo_open(PGconn *conn, Oid lobjId, int mode);
int lo_close(PGconn *conn, int fd);
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
int lo_write(PGconn *conn, int fd, char *buf, size_t len);
int lo_lseek(PGconn *conn, int fd, int offset, int whence);
Oid lo_creat(PGconn *conn, int mode);
int lo_tell(PGconn *conn, int fd);
```

Refer to the online documentation for more details on these functions.

INDEX

B

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z