PostgreSQL 7.1 Developer's Guide

The PostgreSQL Global Development Group

PostgreSQL 7.1 Developer’s Guide

by The PostgreSQL Global Development Group

This document contains assorted information that can be of use to PostgreSQL developers.

Legal Notice

Table of Contents

Table of Contents ... i
List of Tables ... iii
Chapter 1. Postgres Source Code ... 1
- 1.1. Formatting ... 1

Chapter 2. Overview of PostgreSQL Internals ... 2
- 2.1. The Path of a Query ... 2
- 2.2. How Connections are Established ... 2
- 2.3. The Parser Stage ... 3
  - 2.3.1. Parser ... 3
  - 2.3.2. Transformation Process ... 4
- 2.4. The Postgres Rule System ... 5
  - 2.4.1. The Rewrite System ... 5
- 2.5. Planner/Optimizer ... 6
  - 2.5.1. Generating Possible Plans ... 6
  - 2.5.2. Data Structure of the Plan ... 7
- 2.6. Executor ... 7

Chapter 3. System Catalogs ... 9
- 3.1. Overview ... 9
- 3.2. pg_aggregate ... 10
- 3.3. pg_attrdef ... 10
- 3.4. pg_attribute ... 11
- 3.5. pg_class ... 12
- 3.6. pg_database ... 13
- 3.7. pg_description ... 14
- 3.8. pg_group ... 14
- 3.9. pg_index ... 15
- 3.10. pg_inherits ... 15
- 3.11. pg_language ... 16
- 3.12. pg_operator ... 16
- 3.13. pg_proc ... 17
- 3.14. pg_relcheck ... 18
- 3.15. pg_shadow ... 18
- 3.16. pg_type ... 19

Chapter 4. Frontend/Backend Protocol ... 22
- 4.1. Overview ... 22
- 4.2. Protocol ... 22
  - 4.2.1. Start-up ... 22
  - 4.2.2. Query ... 24
  - 4.2.3. Function Call ... 25
  - 4.2.4. Notification Responses ... 26
  - 4.2.5. Cancelling Requests in Progress ... 26
  - 4.2.6. Termination ... 27
- 4.3. Message Data Types ... 27
- 4.4. Message Formats ... 27

Chapter 5. gcc Default Optimizations ... 35
Chapter 6. BKI Backend Interface ... 36
- 6.1. BKI File Format ... 36
- 6.2. BKI Commands ... 36
- 6.3. Example ... 37

Chapter 7. Page Files ... 38
Chapter 8. Genetic Query Optimization ... 40
- 8.1. Query Handling as a Complex Optimization Problem ... 40

8.2. Genetic Algorithms (GA) . . . 40
8.3. Genetic Query Optimization (GEQO) in Postgres. . . 41
8.3.1. Future Implementation Tasks for PostgreSQL GEQO . . . 42
References . . . 42
Bibliography . . . 43
SQL Reference Books . . . 43
PostgreSQL-Specific Documentation. . . 43
Proceedings and Articles. . . 44
Appendix DG1. The CVS Repository . . . 45
DG1.1. Getting The Source Via Anonymous CVS . . . 45
DG1.2. CVS Tree Organization . . . 46
DG1.3. Getting The Source Via CVSup. . . 48
DG1.3.1. Preparing A CVSup Client System. . . 48
DG1.3.2. Running a CVSup Client . . . 48
DG1.3.3. Installing CVSup . . . 50
DG1.3.4. Installation from Sources. . . 51
Appendix DG2. Documentation. . . 53
DG2.1. DocBook. . . 53
DG2.2. Toolsets . . . 53
DG2.2.1. Linux RPM Installation . . . 54
DG2.2.2. FreeBSD Installation . . . 54
DG2.2.3. Debian Packages. . . 55
DG2.2.4. Manual Installation from Source . . . 55
DG2.3. Building The Documentation. . . 56
DG2.3.1. HTML . . . 57
DG2.3.2. Manpages . . . 58
DG2.3.3. Hardcopy Generation. . . 58
DG2.3.4. Plain Text Files . . . 60
DG2.4. Documentation Authoring . . . 60
DG2.4.1. Emacs/PSGML . . . 60
DG2.4.2. Other Emacs modes . . . 61

List of Tables

3-1. System Catalogs 9
3-2. pg_aggregate Columns 10
3-3. pg_attrdef Columns 11
3-4. pg_attribute Columns 11
3-5. pg_class Columns 12
3-6. pg_database Columns 13
3-7. pg_description Columns 14
3-8. pg_group Columns 14
3-9. pg_index Columns 15
3-10. pg_inherits Columns 15
3-11. pg_language Columns 16
3-12. pg_operator Columns 16
3-13. pg_proc Columns 17
3-14. pg_relcheck Columns 18
3-15. pg_shadow Columns 18
3-16. pg_type Columns 19
7-1. Sample Page Layout 38

Chapter 1. Postgres Source Code

1.1. Formatting

Source code formatting uses a 4 column tab spacing, currently with tabs preserved (i.e. tabs are not expanded to spaces).

For emacs, add the following (or something similar) to your `~/.emacs` initialization file:

```
;; check for files with a path containing "postgres" or "pgsql"
(setq auto-mode-alist (cons '("\\(postgres\\|pgsql\\).*\\.[ch]\\'" .
pgsql-c-mode) auto-mode-alist))
(setq auto-mode-alist (cons '("\\(postgres\\|pgsql\\).*\\.cc\\'" .
pgsql-c-mode) auto-mode-alist))

(defun pgsql-c-mode ()
  ;; sets up formatting for Postgres C code
  (interactive)
  (c-mode)
  (setq-default tab-width 4)
  (c-set-style "bsd")             ; set c-basic-offset to 4, plus other
stuff
  (c-set-offset 'case-label '+)   ; tweak case indent to match PG custom
  (setq indent-tabs-mode t))      ; make sure we keep tabs when indenting
```

For vi, your `~/.vimrc` or equivalent file should contain the following:

```
set tabstop=4
```

or equivalently from within vi, try

```
:set ts=4
```

The text browsing tools more and less can be invoked as

```
more -x4
less -x4
```

Chapter 2. Overview of PostgreSQL Internals

Author: This chapter originally appeared as a part of *Simkovics, 1998*, Stefan Simkovics' Master's Thesis prepared at Vienna University of Technology under the direction of O.Univ.Prof.Dr. Georg Gottlob and Univ.Ass. Mag. Katrin Seyr.

This chapter gives an overview of the internal structure of the backend of Postgres. After having read the following sections you should have an idea of how a query is processed. Don't expect a detailed description here (I think such a description dealing with all data structures and functions used within Postgres would exceed 1000 pages!). This chapter is intended to help understanding the general control and data flow within the backend from receiving a query to sending the results.

2.1. The Path of a Query

Here we give a short overview of the stages a query has to pass in order to obtain a result.

1. A connection from an application program to the Postgres server has to be established. The application program transmits a query to the server and receives the results sent back by the server.
2. The *parser stage* checks the query transmitted by the application program (client) for correct syntax and creates a *query tree*.
3. The *rewrite system* takes the query tree created by the parser stage and looks for any *rules* (stored in the *system catalogs*) to apply to the *querytree* and performs the transformations given in the *rule bodies*. One application of the rewrite system is given in the realization of *views*.

   Whenever a query against a view (i.e. a *virtual table*) is made, the rewrite system rewrites the user's query to a query that accesses the *base tables* given in the *view definition* instead.
4. The *planner/optimizer* takes the (rewritten) querytree and creates a *queryplan* that will be the input to the *executor*.

   It does so by first creating all possible *paths* leading to the same result. For example if there is an index on a relation to be scanned, there are two paths for the scan. One possibility is a simple sequential scan and the other possibility is to use the index. Next the cost for the execution of each plan is estimated and the cheapest plan is chosen and handed back.
5. The executor recursively steps through the *plan tree* and retrieves tuples in the way represented by the plan. The executor makes use of the *storage system* while scanning relations, performs *sorts* and *joins*, evaluates *qualifications* and finally hands back the tuples derived.

In the following sections we will cover every of the above listed items in more detail to give a better understanding on Postgres's internal control and data structures.

2.2. How Connections are Established

Postgres is implemented using a simple "process per-user" client/server model. In this model there is one *client process* connected to exactly one *server process*. As we don't know *per se* how many connections will be made, we have to use a *master process* that spawns a new server process every time a connection is requested. This master process is called `postmaster` and listens at a specified TCP/IP port for incoming connections. Whenever a request for a connection is detected the `postmaster` process spawns a new server process called `postgres`. The server tasks (`postgres` processes) communicate with each other using *semaphores* and *shared memory* to ensure data integrity throughout concurrent data access. Figure \ref{connection} illustrates the interaction of the master process `postmaster` the server process `postgres` and a client application.

The client process can either be the psql frontend (for interactive SQL queries) or any user application implemented using the `libpg` library. Note that applications implemented using ecpg (the Postgres embedded SQL preprocessor for C) also use this library.

Once a connection is established the client process can send a query to the *backend* (server). The query is transmitted using plain text, i.e. there is no parsing done in the *frontend* (client). The server parses the query, creates an *execution plan*, executes the plan and returns the retrieved tuples to the client by transmitting them over the established connection.

2.3. The Parser Stage

The *parser stage* consists of two parts:

The *parser* defined in `gram.y` and `scan.l` is built using the Unix tools yacc and lex.

The *transformation process* does modifications and augmentations to the data structures returned by the parser.

2.3.1. Parser

The parser has to check the query string (which arrives as plain ASCII text) for valid syntax. If the syntax is correct a *parse tree* is built up and handed back otherwise an error is returned. For the implementation the well known Unix tools lex and yacc are used.

The *lexer* is defined in the file `scan.l` and is responsible for recognizing *identifiers*, the *SQL keywords* etc. For every keyword or identifier that is found, a *token* is generated and handed to the parser.

The parser is defined in the file `gram.y` and consists of a set of *grammar rules* and *actions* that are executed whenever a rule is fired. The code of the actions (which is actually C-code) is used to build up the parse tree.

The file `scan.l` is transformed to the C-source file `scan.c` using the program lex and `gram.y` is transformed to `gram.c` using yacc. After these transformations have taken place a normal C-compiler can be used to create the parser. Never make any changes to the generated C-files as they will be overwritten the next time lex or yacc is called.

> **Note:** The mentioned transformations and compilations are normally done automatically using the *makefiles* shipped with the Postgres source distribution.

A detailed description of yacc or the grammar rules given in `gram.y` would be beyond the scope of this paper. There are many books and documents dealing with lex and yacc. You should be familiar with yacc before you start to study the grammar given in `gram.y` otherwise you won't understand what happens there.

For a better understanding of the data structures used in Postgres for the processing of a query we use an example to illustrate the changes made to these data structures in every stage. This example contains the following simple query that will be used in various descriptions and figures throughout the following sections. The query assumes that the tables given in *The Supplier Database* have already been defined.

Example 2-1. A Simple Select

```
select s.sname, se.pno
    from supplier s, sells se
    where s.sno > 2 and s.sno = se.sno;
```

Figure \ref{parsetree} shows the *parse tree* built by the grammar rules and actions given in `gram.y` for the query given in Example 2-1 (without the *operator tree* for the *where clause* which is shown in figure \ref{where_clause} because there was not enough space to show both data structures in one figure).

The top node of the tree is a `SelectStmt` node. For every entry appearing in the *from clause* of the SQL query a `RangeVar` node is created holding the name of the *alias* and a pointer to a `RelExpr` node holding the name of the *relation*. All `RangeVar` nodes are collected in a list which is attached to the field `fromClause` of the `SelectStmt` node.

For every entry appearing in the *select list* of the SQL query a `ResTarget` node is created holding a pointer to an `Attr` node. The `Attr` node holds the *relation name* of the entry and a pointer to a `Value` node holding the name of the *attribute*. All `ResTarget` nodes are collected to a list which is connected to the field `targetList` of the `SelectStmt` node.

Figure \ref{where_clause} shows the operator tree built for the where clause of the SQL query given in Example 2-1 which is attached to the field `qual` of the `SelectStmt` node. The top node of the operator tree is an `A_Expr` node representing an `AND` operation. This node has two successors called `lexpr` and `rexpr` pointing to two *subtrees*. The subtree attached to `lexpr` represents the qualification `s.sno > 2` and the one attached to `rexpr` represents `s.sno = se.sno`. For every attribute an `Attr` node is created holding the name of the relation and a pointer to a `Value` node holding the name of the attribute. For the constant term appearing in the query a `Const` node is created holding the value.

2.3.2. Transformation Process

The *transformation process* takes the tree handed back by the parser as input and steps recursively through it. If a `SelectStmt` node is found, it is transformed to a `Query` node that will be the top most node of the new data structure. Figure \ref{transformed} shows the transformed data structure (the part for the transformed *where clause* is given in figure \ref{transformed_where} because there was not enough space to show all parts in one figure).

Now a check is made, if the *relation names* in the *FROM clause* are known to the system. For every relation name that is present in the *system catalogs* a RTE node is created containing the relation name, the *alias name* and the *relation id*. From now on the relation ids are used to refer to the *relations* given in the query. All RTE nodes are collected in the *range table entry list* that is connected to the field `rtable` of the `Query` node. If a name of a relation that is not known to the system is detected in the query an error will be returned and the query processing will be aborted.

Next it is checked if the *attribute names* used are contained in the relations given in the query. For every attribute} that is found a TLE node is created holding a pointer to a `Resdom` node (which holds the name of the column) and a pointer to a `VAR` node. There are two important numbers in the `VAR` node. The field `varno` gives the position of the relation containing the current attribute} in the range table entry list created above. The field `varattno` gives the position of the attribute within the relation. If the name of an attribute cannot be found an error will be returned and the query processing will be aborted.

2.4. The Postgres Rule System

Postgres supports a powerful *rule system* for the specification of *views* and ambiguous *view updates*. Originally the Postgres rule system consisted of two implementations:

> The first one worked using *tuple level* processing and was implemented deep in the *executor*. The rule system was called whenever an individual tuple had been accessed. This implementation was removed in 1995 when the last official release of the Postgres project was transformed into Postgres95.
>
> The second implementation of the rule system is a technique called *query rewriting*. The *rewrite system*} is a module that exists between the *parser stage* and the *planner/optimizer*. This technique is still implemented.

For information on the syntax and creation of rules in the Postgres system refer to *The PostgreSQL User's Guide*.

2.4.1. The Rewrite System

The *query rewrite system* is a module between the parser stage and the planner/optimizer. It processes the tree handed back by the parser stage (which represents a user query) and if there is a rule present that has to be applied to the query it rewrites the tree to an alternate form.

2.4.1.1. Techniques To Implement Views

Now we will sketch the algorithm of the query rewrite system. For better illustration we show how to implement views using rules as an example.

Let the following rule be given:

```
create rule view_rule
as on select
to test_view
do instead
   select s.sname, p.pname
   from supplier s, sells se, part p
   where s.sno = se.sno and
         p.pno = se.pno;
```

The given rule will be *fired* whenever a select against the relation `test_view` is detected. Instead of selecting the tuples from `test_view` the select statement given in the *action part* of the rule is executed.

Let the following user-query against `test_view` be given:

```
select sname
from test_view
where sname <> 'Smith';
```

Here is a list of the steps performed by the query rewrite system whenever a user-query against `test_view` appears. (The following listing is a very informal description of the algorithm just intended for basic understanding. For a detailed description refer to *Stonebraker et al, 1989*).

`test_view` Rewrite

1. Take the query given in the action part of the rule.
2. Adapt the targetlist to meet the number and order of attributes given in the user-query.
3. Add the qualification given in the where clause of the user-query to the qualification of the query given in the action part of the rule.

Given the rule definition above, the user-query will be rewritten to the following form (Note that the rewriting is done on the internal representation of the user-query handed back by the parser stage but the derived new data structure will represent the following query):

```
select s.sname
from supplier s, sells se, part p
where s.sno = se.sno and
      p.pno = se.pno and
      s.sname <> 'Smith';
```

2.5. Planner/Optimizer

The task of the *planner/optimizer* is to create an optimal execution plan. It first combines all possible ways of *scanning* and *joining* the relations that appear in a query. All the created paths lead to the same result and it's the task of the optimizer to estimate the cost of executing each path and find out which one is the cheapest.

2.5.1. Generating Possible Plans

The planner/optimizer decides which plans should be generated based upon the types of indices defined on the relations appearing in a query. There is always the possibility of performing a sequential scan on a relation, so a plan using only sequential scans is always created. Assume an index is defined on a relation (for example a B-tree index) and a query contains the restriction `relation.attribute OPR constant`. If `relation.attribute` happens to match the key of the B-tree index and `OPR` is anything but '<>' another plan is created using the B-tree index to scan the relation. If there are further indices present and the restrictions in the query happen to match a key of an index further plans will be considered.

After all feasible plans have been found for scanning single relations, plans for joining relations are created. The planner/optimizer considers only joins between every two relations for which there exists a corresponding join clause (i.e. for which a restriction like `where rel1.attr1=rel2.attr2` exists) in the where qualification. All possible plans are generated for every join pair considered by the planner/optimizer. The three possible join strategies are:

nested iteration join: The right relation is scanned once for every tuple found in the left relation. This strategy is easy to implement but can be very time consuming.

merge sort join: Each relation is sorted on the join attributes before the join starts. Then the two relations are merged together taking into account that both relations are ordered on the join attributes. This kind of join is more attractive because every relation has to be scanned only once.

hash join: the right relation is first hashed on its join attributes. Next the left relation is scanned and the appropriate values of every tuple found are used as hash keys to locate the tuples in the right relation.

2.5.2. Data Structure of the Plan

Here we will give a little description of the nodes appearing in the plan. Figure \ref{plan} shows the plan produced for the query in example \ref{simple_select}.

The top node of the plan is a `MergeJoin` node that has two successors, one attached to the field `lefttree` and the second attached to the field `righttree`. Each of the subnodes represents one relation of the join. As mentioned above a merge sort join requires each relation to be sorted. That's why we find a `Sort` node in each subplan. The additional qualification given in the query (`s.sno > 2`) is pushed down as far as possible and is attached to the `qpqual` field of the leaf `SeqScan` node of the corresponding subplan.

The list attached to the field `mergeclauses` of the `MergeJoin` node contains information about the join attributes. The values `65000` and `65001` for the `varno` fields in the `VAR` nodes appearing in the `mergeclauses` list (and also in the `targetlist`) mean that not the tuples of the current node should be considered but the tuples of the next "deeper" nodes (i.e. the top nodes of the subplans) should be used instead.

Note that every `Sort` and `SeqScan` node appearing in figure \ref{plan} has got a `targetlist` but because there was not enough space only the one for the `MergeJoin` node could be drawn.

Another task performed by the planner/optimizer is fixing the *operator ids* in the `Expr` and `Oper` nodes. As mentioned earlier, Postgres supports a variety of different data types and even user defined types can be used. To be able to maintain the huge amount of functions and operators it is necessary to store them in a system table. Each function and operator gets a unique operator id. According to the types of the attributes used within the qualifications etc., the appropriate operator ids have to be used.

2.6. Executor

The *executor* takes the plan handed back by the planner/optimizer and starts processing the top node. In the case of our example (the query given in example \ref{simple_select}) the top node is a `MergeJoin` node.

Before any merge can be done two tuples have to be fetched (one from each subplan). So the executor recursively calls itself to process the subplans (it starts with the subplan attached to `lefttree`). The new top node (the top node of the left subplan) is a `SeqScan` node and again a tuple has to be fetched before the node itself can be processed. The executor calls itself recursively another time for the subplan attached to `lefttree` of the `SeqScan` node.

Now the new top node is a `Sort` node. As a sort has to be done on the whole relation, the executor starts fetching tuples from the `Sort` node's subplan and sorts them into a temporary relation (in memory or a file) when the `Sort` node is visited for the first time. (Further examinations of the `Sort` node will always return just one tuple from the sorted temporary relation.)

Every time the processing of the `Sort` node needs a new tuple the executor is recursively called for the `SeqScan` node attached as subplan. The relation (internally referenced by the value given in the `scanrelid` field) is scanned for the next tuple. If the tuple satisfies the qualification given by the tree attached to `qpqual` it is handed back, otherwise the next tuple is fetched until the qualification is satisfied. If the last tuple of the relation has been processed a `NULL` pointer is returned.

After a tuple has been handed back by the `lefttree` of the `MergeJoin` the `righttree` is processed in the same way. If both tuples are present the executor processes the `MergeJoin` node. Whenever a new tuple from one of the subplans is needed a recursive call to the executor is performed to obtain it. If a joined tuple could be created it is handed back and one complete processing of the plan tree has finished.

Now the described steps are performed once for every tuple, until a `NULL` pointer is returned for the processing of the `MergeJoin` node, indicating that we are finished.

Chapter 3. System Catalogs

3.1. Overview

The system catalogs are the place where a relational database management system stores schema metadata, such as information about tables and columns, and internal bookkeeping information. PostgreSQL's system catalogs are regular tables. You can drop and recreate the tables, add columns, insert and update values, and severely mess up your system that way. Normally one never has to change the system catalogs by hand, there are always SQL commands to do that. (For example, **CREATE DATABASE** inserts a row into the `pg_database` catalog -- and actually creates the database on disk.) There are some exceptions for esoteric operations, such as adding index access methods.

Table 3-1. System Catalogs

Catalog Name	Purpose
pg_aggregate	aggregate functions
pg_am	index access methods
pg_amop	access method operators
pg_amproc	access method support procedures
pg_attrdef	column default values
pg_attribute	table columns (attributes, fields)
pg_class	tables, indexes, sequences (relations)
pg_database	databases
pg_description	descriptions or comments on database objects
pg_group	user groups
pg_index	additional index information
pg_inheritproc	(not used)
pg_inherits	table inheritance hierarchy
pg_ipl	(not used)
pg_language	languages for writing functions
pg_largeobject	large objects
pg_listener	asynchronous notification
pg_opclass	index access method operator classes
pg_operator	operators
pg_proc	functions and procedures
pg_relcheck	check constraints
pg_rewrite	query rewriter rules

Catalog Name	Purpose
pg_shadow	database users
pg_statistic	optimizer statistics
pg_trigger	triggers
pg_type	data types

More detailed documentation of most catalogs follow below. The catalogs that relate to index access methods are explained in the *Programmer's Guide*. Some catalogs don't have any documentation, yet.

3.2. pg_aggregate

`pg_aggregate` stores information about aggregate functions. An aggregate function is a function that operates on a set of values (typically one column from each the row that matches a query condition) and returns a single value computed from all these values. Typical aggregate functions are `sum`, `count`, and `max`.

Table 3-2. pg_aggregate Columns

Name	Type	References	Description
aggname	`name`		Name of the aggregate function
aggowner	`int4`	pg_shadow.usesysid	Owner (creator) of the aggregate function
aggtransfn	`regproc`		Transition function
aggfinalfn	`regproc`		Final function
aggbasetype	`oid`	pg_type.oid	The type on which this function operates when invoked from SQL
aggtranstype	`oid`	pg_type.oid	The type of the aggregate function's internal transition (state) data
aggfinaltype	`oid`	pg_type.oid	The type of the result
agginitval	`text`		The initial value of the transition state. This is a text field which will be cast to the type of aggtranstype.

New aggregate functions are registered with the **CREATE AGGREGATE** command. See the *Programmer's Guide* for more information about writing aggregate functions and the meaning of the transition functions, etc.

An aggregate function is identified through name *and* argument type. Hence aggname and aggname are the composite primary key.

3.3. pg_attrdef

This catalog stores column default values. The main information about columns is stored in `pg_attribute` (see below). Only columns that explicitly specify a default value (when the table is created or the column is added) will have an entry here.

Table 3-3. pg_attrdef Columns

Name	Type	References	Description
adrelid	`oid`	pg_class.oid	The table this column belongs to
adnum	`int2`		The number of the column; see `pg_attribute.pg_attnum`
adbin	`text`		An internal representation of the column default value
adsrc	`text`		A human-readable representation of the default value

3.4. pg_attribute

`pg_attribute` stores information about table columns. There will be exactly one `pg_attribute` row for every column in every table in the database. (There will also be attribute entries for indexes and other objects. See `pg_class`.)

The term attribute is equivalent to column and is used for historical reasons.

Table 3-4. pg_attribute Columns

Name	Type	References	Description
attrelid	`oid`	pg_class.oid	The table this column belongs to
attname	`name`		Column name
atttypid	`oid`	pg_type.oid	The data type of this column
attdispersion	`float4`		`attdispersion` is the dispersion statistic of the column (0.0 to 1.0), or zero if the statistic has not been calculated, or -1.0 if **VACUUM** found that the column contains no duplicate entries (in which case the dispersion should be taken as 1.0/numberOfRows for the current table size). The -1.0 hack is useful because the number of rows may be updated more often than `attdispersion` is. We assume that the column will retain its no-duplicate-entry property.
attlen	`int2`		This is a copy of the `pg_type.typlen` for this column's type.
attnum	`int2`		The number of the column. Ordinary columns are numbered from 1 up. System columns, such as `oid`, have (arbitrary) negative numbers.
attnelems	`int4`		Number of dimensions, if the column is an array
attcacheoff	`int4`		Always -1 in storage, but when loaded into a tuple descriptor in memory this may be updated cache the offset of the attribute within the tuple.
atttypmod	`int4`		`atttypmod` records type-specific data supplied at table creation time (for example, the maximum length of a `varchar` column). It is passed to type-specific input and output functions as the third argument. The value will generally be -1 for types that do not need typmod.
attbyval	`bool`		A copy of `pg_type.typbyval` of this column's type

Name	Type	References	Description
attstorage	`char`		A copy of `pg_type.typstorage` of this column's type
attisset	`bool`		If true, this attribute is a set. In that case, what is really stored in the attribute is the OID of a tuple in the `pg_proc` catalog. The `pg_proc` tuple contains the query string that defines this set - i.e., the query to run to get the set. So the `atttypid` (see above) refers to the type returned by this query, but the actual length of this attribute is the length (size) of an `oid`. --- At least this is the theory. All this is probably quite broken these days.
attalign	`char`		A copy of `pg_type.typalign` of this column's type
attnotnull	`bool`		This represents a NOT NULL constraint. It is possible to change this field to enable or disable the constraint.
atthasdef	`bool`		This column has a default value, in which case there will be a corresponding entry in the `pg_attrdef` catalog that actually defines the value.

3.5. pg_class

`pg_class` catalogues tables and mostly everything else that has columns or is otherwise similar to a table. This includes indexes (but see `pg_index`), sequences, views, and some kinds of special relation kinds. Below, when we mean all of these kinds of objects we speak of relations. Not all fields are meaningful for all relation types.

Table 3-5. pg_class Columns

Name	Type	References	Description
relname	`name`		Name of the table, index, view, etc.
reltype	`oid`	pg_type.oid	The data type that corresponds to this table (not functional, only set for system tables)
relowner	`int4`	pg_shadow.usesysid	Owner of the relation
relam	`oid`	pg_am.oid	If this is an index, the access method used (btree, hash, etc.)
relfilenode	`oid`		Name of the on-disk file of this relation
relpages	`int4`		Size of the on-disk representation of this table in pages (size BLCKSZ). This is only an approximate value which is calculated during vacuum.
reltuples	`int4`		Number of tuples in the table. This is only an estimate used by the planner, updated by **VACUUM**.
reltoastrelid	`oid`	pg_class.oid	Oid of the TOAST table associated with this table, 0 if none. The TOAST table stores large attributes out of line in a secondary table.
reltoastidxid	`oid`	pg_class.oid	Oid of the index on the TOAST table for this table, 0 if none

Name	Type	References	Description
relhasindex	`bool`		True if this is a table and it has at least one index
relisshared	`bool`		XXX (This is not what it seems to be.)
relkind	`char`		’r’ = ordinary table, ’i’ = index, ’S’ = sequence, ’v’ = view, ’s’ = special, ’t’ = secondary TOAST table
relnatts	`int2`		Number of columns in the relation, besides system columns. There must be this many corresponding entries in `pg_attribute`. See also `pg_attribute.attnum`.
relchecks	`int2`		Number of check constraints on the table; see `pg_relcheck` catalog
reltriggers	`int2`		Number of triggers on the table; see `pg_trigger` catalog
relukeys	`int2`		unused (*Not* the number of unique keys or something.)
relfkeys	`int2`		Number foreign keys on the table
relhaspkey	`bool`		unused (No, this does not say whether the table has a primary key. It’s really unused.)
relhasrules	`bool`		Table has rules
relhas-subclass	`bool`		At least one table inherits this one
relacl	`aclitem[]`		Access permissions. See the descriptions of **GRANT** and **REVOKE** for details.

3.6. pg_database

The `pg_database` catalog stores information about the available databases. The `pg_database` table is shared between all databases of a cluster. Databases are created with the **CREATE DATABASE**. Consult the *Administrator’s Guide* for details about the meaning of some of the parameters.

Table 3-6. pg_database Columns

Name	Type	References	Description
datname	`name`		Database name
datdba	`int4`	pg_shadow.usesysid	Owner of the database, initially who created it
encoding	`int4`		Character/multibyte encoding for this database
datistemplate	`bool`		If true then this database can be used in the TEMPLATE clause of **CREATE DATABASE** to create the new database as a clone of this one.

Name	Type	References	Description
datallowconn	`bool`		If false then no one can connect to this database. This is used to protect the template0 database from being altered.
datlastsysoid	`oid`		Last oid in existence after the database was created; useful particularly to pg_dump
datpath	`text`		If the database is stored at an alternative location then this records the location. It's either an environment variable name or an absolute path, depending how it was entered.

3.7. pg_description

The pg_description table can store an optional description or comment for each database object. Descriptions can be manipulated with the **COMMENT** command. Client applications can view the descriptions by joining with this table. Many builtin system objects have comments associated with them that are shown by psql's `\d` commands.

Table 3-7. pg_description Columns

Name	Type	References	Description
objoid	`oid`	any oid attribute	The oid of the object this description pertains to
description	`text`		Arbitrary text that serves as the description of this object.

3.8. pg_group

This catalog defines groups and stores what users belong to what groups. Groups are created with the **CREATE GROUP** command. Consult the *Administrator's Guide* for information about user permission management.

Table 3-8. pg_group Columns

Name	Type	References	Description
groname	`name`		Name of the group
grosysid	`int4`		An arbitrary number to identify this group
grolist	`int4[]`	pg_shadow.usesysid	An array containing the ids of the users in this group

3.9. pg_index

`pg_index` contains part of the information about indexes. The rest is mostly in `pg_class`.

Table 3-9. pg_index Columns

Name	Type	References	Description
indexrelid	`oid`	pg_class.oid	The oid of the pg_class entry for this index
indrelid	`oid`	pg_class.oid	The oid of the pg_class entry for the table this index is for
indproc	`oid`	pg_proc.oid	The registered procedure if this is a functional index
indkey	`int2vector`	pg_attribute.attnum	This is an vector (array) of up to INDEX_MAX_KEYS values that indicate which table columns this index pertains to. For example a value of `1 3` would mean that the first and the third column make up the index key.
indclass	`oidvector`	pg_opclass.oid	For each column in the index key this contains a reference to the operator class to use. See `pg_opclass` for details.
indisclustered	`bool`		unused
indislossy	`bool`		???
indisunique	`bool`		If true, this is a unique index.
indisprimary	`bool`		If true, this index is a unique index that represents the primary key of the table.
indreference	`oid`		unused
indpred	`text`		Query plan for partial index predicate (not functional)

3.10. pg_inherits

This catalog records information about table inheritance hierarchies.

Table 3-10. pg_inherits Columns

Name	Type	References	Description
inhrelid	`oid`	pg_class.oid	This is the reference to the subtable, that is, it records the fact that the identified table is inherited from some other table.
inhparent	`oid`	pg_class.oid	This is the reference to the parent table, from which the table referenced by `inhrelid` inherited from.
inhseqno	`int4`		If there is more than one subtable/parent pair (multiple inheritance), this number tells the order in which the inherited columns are to be arranged. The count starts at 1.

3.11. pg_language

`pg_language` registers call interfaces or languages in which you can write functions or stored procedures. See under **CREATE LANGUAGE** and in the *Programmer's Guide* for more information about language handlers.

Table 3-11. pg_language Columns

Name	Type	References	Description
lanname	`name`		Name of the language (to be specified when creating a function)
lanispl	`bool`		This is false for internal languages (such as SQL) and true for dynamically loaded language handler modules. It essentially means that, if it is true, the language may be dropped.
lanpltrusted	`bool`		This is a trusted language. See under **CREATE LANGUAGE** what this means. If this is an internal language (`lanispl` is false) then this field is meaningless.
lanplcallfoid	`oid`	pg_proc.oid	For non-internal languages this references the language handler, which is a special function that is responsible for executing all functions that are written in the particular language.
lancompiler	`text`		not used

3.12. pg_operator

See **CREATE OPERATOR** and the *Programmer's Guide* for details on these operator parameters.

Table 3-12. pg_operator Columns

Name	Type	References	Description
oprname	`name`		Name of the operator
oprowner	`int4`	pg_shadow.usesysid	Owner (creator) of the operator
oprprec	`int2`		unused
oprkind	`char`		'b' = infix (both), 'l' = prefix (left), 'r' = postfix (right)
oprisleft	`bool`		unused
oprcanhash	`bool`		This operator supports hash joins.
oprleft	`oid`	pg_type.oid	Type of the left operand
oprright	`oid`	pg_type.oid	Type of the right operand
oprresult	`oid`	pg_type.oid	Type of the result
oprcom	`oid`	pg_operator.oid	Commutator of this operator, if any

Name	Type	References	Description
oprnegate	`oid`	pg_operator.oid	Negator of this operator, if any
oprlsortop	`oid`	pg_operator.oid	If this operator supports merge joins, the operator that sorts the type of the left-hand operand
oprrsortop	`oid`	pg_operator.oid	If this operator supports merge joins, the operator that sorts the type of the right-hand operand
oprcode	`regproc`		Function that implements this operator
oprrest	`regproc`		Restriction selectivity estimation function for this operator
oprjoin	`regproc`		Join selectivity estimation function for this operator

3.13. pg_proc

This catalog stores information about functions (or procedures). The description of **CREATE FUNCTION** and the *Programmer's Guide* contain more information about the meaning of some fields.

Table 3-13. pg_proc Columns

Name	Type	References	Description
proname	`name`		Name of the function
proowner	`int4`	pg_shadow.usesysid	Owner (creator) of the function
prolang	`oid`	pg_language.oid	Implementation language or call interface of this function
proisinh	`bool`		unused
proistrusted	`bool`		not functional
proiscachable	`bool`		Function returns same result for same input values
proisstrict	`bool`		Function returns null if any call argument is null. In that case the function won't actually be called at all. Functions that are not strict must be prepared to handle null inputs.
pronargs	`int2`		Number of arguments
proretset	`bool`		Function returns a set (probably not functional)
prorettype	`oid`	pg_type.oid	Data type of the return value (0 if the function does not return a value)
proargtypes	`oidvector`	pg_type.oid	A vector with the data types of the function arguments
probyte_pct	`int4`		dead code
properbyte_pct	`int4`		dead code

Name	Type	References	Description
propercall_pct	`int4`		dead code
prooutin_ratio	`int4`		dead code
prosrc	`text`		This tells the function handler how to invoke the function. It might be the actual source code of the function for interpreted languages, a link symbol, a file name, or just about anything else, depending the implementation language/call convention.
probin	`bytea`		?

3.14. pg_relcheck

This system catalog stores CHECK constraints on tables. (Column constraints are not treated specially. Every column constraint is equivalent to some table constraint.) See under **CREATE TABLE** for more information.

Table 3-14. pg_relcheck Columns

Name	Type	References	Description
rcrelid	`oid`	pg_class.oid	The table this check constraint is on
rcname	`name`		Constraint name
rcbin	`text`		An internal representation of the constraint expression
rcsrc	`text`		A human-readable representation of the consraint expression

> **Note:** `pg_class.relchecks` needs to match up with the entries in this table.

3.15. pg_shadow

`pg_shadow` contains information about database users. The name stems from the fact that this table should not be readable by the public since it contains passwords. `pg_user` is a view on `pg_shadow` that blanks out the password field.

The *Administrator's Guide* contains detailed information about user and permission management.

Table 3-15. pg_shadow Columns

Name	Type	References	Description
usename	`name`		User name
usesysid	`int4`		User id (arbitrary number used to reference this user)
usecreatedb	`bool`		User may create databases
usetrace	`bool`		not used
usesuper	`bool`		User is a superuser
usecatupd	`bool`		User may update system catalogs. (Even a superuser may not do this unless this attribute is true.)

Name	Type	References	Description
passwd	`text`		Password
valuntil	`abstime`		Account expiry time (only used for password authentication)

3.16. pg_type

Table 3-16. pg_type Columns

Name	Type	References	Description
typname	`name`		Data type name
typowner	`int4`	pg_shadow.usesysid	Owner (creator) of the type
typlen	`int2`		Length of the storage representation of the type, -1 if variable length
typprtlen	`int2`		unused
typbyval	`bool`		`typbyval` determines whether internal routines pass a value of this type by value or by reference. Only `char`, `short`, and `int` equivalent items can be passed by value, so if the type is not 1, 2, or 4 bytes long, Postgres does not have the option of passing by value and so `typbyval` had better be false. Variable-length types are always passed by reference. Note that `typbyval` can be false even if the length would allow pass-by-value; this is currently true for type `float4`, for example.
typtype	`char`		`typtype` is `b` for a basic type and `c` for a catalog type (i.e., a table). If `typtype` is `c`, `typrelid` is the OID of the type's entry in `pg_class`.
typisdefined	`bool`		???
typdelim	`char`		Character that separates two values of this type when parsing array input
typrelid	`oid`	pg_class.oid	If this is a catalog type (see `typtype`), then this field points to the `pg_class` entry that defines the corresponding table. A table could theoretically be used as a composite data type, but this is not fully functional.
typelem	`oid`	pg_type.oid	If `typelem` is not 0 then it identifies another row in `pg_type`. The current type can then be subscripted like an array yielding values of type `typelem`. A non-zero `typelem` does not guarantee this type to be a real array type; some ordinary fixed-length types can also be subscripted (e.g., `oidvector`). Variable-length types can *not* be turned into pseudo-arrays like that. Hence, the way to determine whether a type is a true array type is typelem != 0 and typlen < 0.

Name	Type	References	Description
typinput	`regproc`		Input function
typoutput	`regproc`		Output function
typreceive	`regproc`		unused
typsend	`regproc`		unused
typalign	`char`		`typalign` is the alignment required when storing a value of this type. It applies to storage on disk as well as most representations of the value inside Postgres. When multiple values are stored consecutively, such as in the representation of a complete row on disk, padding is inserted before a datum of this type so that it begins on the specified boundary. The alignment reference is the beginning of the first datum in the sequence. Possible values are: 'c' = CHAR alignment, i.e., no alignment needed. 's' = SHORT alignment (2 bytes on most machines). 'i' = INT alignment (4 bytes on most machines). 'd' = DOUBLE alignment (8 bytes on many machines, but by no means all). **Note:** For types used in system tables, it is critical that the size and alignment defined in `pg_type` agree with the way that the compiler will lay out the field in a struct representing a table row.

Name	Type	References	Description
typstorage	char		typstorage tells for variable-length types (those with typlen = -1) if the type is prepared for toasting and what the default strategy for attributes of this type should be. Possible values are 'p': Value must always be stored plain. 'e': Value can be stored in a secondary relation (if relation has one, see pg_class.reltoastrelid). 'm': Value can be stored compressed inline. 'x': Value can be stored compressed inline or in secondary . Note that 'm' fields can also be moved out to secondary storage, but only as a last resort ('e' and 'x' fields are moved first).
typdefault	text		???

Chapter 4. Frontend/Backend Protocol

> **Note:** Written by Phil Thompson (<phil@river-bank.demon.co.uk>). Updates for protocol 2.0 by Tom Lane (<tgl@sss.pgh.pa.us>).

Postgres uses a message-based protocol for communication between frontends and backends. The protocol is implemented over TCP/IP and also on Unix sockets. Postgres 6.3 introduced version numbers into the protocol. This was done in such a way as to still allow connections from earlier versions of frontends, but this document does not cover the protocol used by those earlier versions.

This document describes version 2.0 of the protocol, implemented in Postgres 6.4 and later.

Higher level features built on this protocol (for example, how `libpq` passes certain environment variables after the connection is established) are covered elsewhere.

4.1. Overview

The three major components are the frontend (running on the client) and the postmaster and backend (running on the server). The postmaster and backend have different roles but may be implemented by the same executable.

A frontend sends a start-up packet to the postmaster. This includes the names of the user and the database the user wants to connect to. The postmaster then uses this, and the information in the `pg_hba.conf` file to determine what further authentication information it requires the frontend to send (if any) and responds to the frontend accordingly.

The frontend then sends any required authentication information. Once the postmaster validates this it responds to the frontend that it is authenticated and hands over the connection to a backend. The backend then sends a message indicating successful start-up (normal case) or failure (for example, an invalid database name).

Subsequent communications are query and result packets exchanged between the frontend and the backend. The postmaster takes no further part in ordinary query/result communication. (However, the postmaster is involved when the frontend wishes to cancel a query currently being executed by its backend. Further details about that appear below.)

When the frontend wishes to disconnect it sends an appropriate packet and closes the connection without waiting for a response for the backend.

Packets are sent as a data stream. The first byte determines what should be expected in the rest of the packet. The exception is packets sent from a frontend to the postmaster, which comprise a packet length then the packet itself. The difference is historical.

4.2. Protocol

This section describes the message flow. There are four different types of flows depending on the state of the connection: start-up, query, function call, and termination. There are also special provisions for notification responses and command cancellation, which can occur at any time after the start-up phase.

4.2.1. Start-up

Start-up is divided into an authentication phase and a backend start-up phase.

Initially, the frontend sends a StartupPacket. The postmaster uses this info and the contents of the pg_hba.conf file to determine what authentication method the frontend must use. The postmaster then responds with one of the following messages:

ErrorResponse

The postmaster then immediately closes the connection.

AuthenticationOk

The postmaster then hands over to the backend. The postmaster takes no further part in the communication.

AuthenticationKerberosV4

The frontend must then take part in a Kerberos V4 authentication dialog (not described here) with the postmaster. If this is successful, the postmaster responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

AuthenticationKerberosV5

The frontend must then take part in a Kerberos V5 authentication dialog (not described here) with the postmaster. If this is successful, the postmaster responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

AuthenticationUnencryptedPassword

The frontend must then send an UnencryptedPasswordPacket. If this is the correct password, the postmaster responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

AuthenticationEncryptedPassword

The frontend must then send an EncryptedPasswordPacket. If this is the correct password, the postmaster responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

If the frontend does not support the authentication method requested by the postmaster, then it should immediately close the connection.

After sending AuthenticationOk, the postmaster attempts to launch a backend process. Since this might fail, or the backend might encounter a failure during start-up, the frontend must wait for the backend to acknowledge successful start-up. The frontend should send no messages at this point. The possible messages from the backend during this phase are:

BackendKeyData

This message is issued after successful backend start-up. It provides secret-key data that the frontend must save if it wants to be able to issue cancel requests later. The frontend should not respond to this message, but should continue listening for a ReadyForQuery message.

ReadyForQuery

Backend start-up is successful. The frontend may now issue query or function call messages.

ErrorResponse

Backend start-up failed. The connection is closed after sending this message.

NoticeResponse

A warning message has been issued. The frontend should display the message but continue listening for ReadyForQuery or ErrorResponse.

The ReadyForQuery message is the same one that the backend will issue after each query cycle. Depending on the coding needs of the frontend, it is reasonable to consider ReadyForQuery as starting a query cycle (and then BackendKeyData indicates successful conclusion of the start-up phase), or to consider ReadyForQuery as ending the start-up phase and each subsequent query cycle.

4.2.2. Query

A Query cycle is initiated by the frontend sending a Query message to the backend. The backend then sends one or more response messages depending on the contents of the query command string, and finally a ReadyForQuery response message. ReadyForQuery informs the frontend that it may safely send a new query or function call.

The possible response messages from the backend are:

CompletedResponse

An SQL command completed normally.

CopyInResponse

The backend is ready to copy data from the frontend to a relation. The frontend should then send a CopyDataRows message. The backend will then respond with a CompletedResponse message with a tag of "COPY".

CopyOutResponse

The backend is ready to copy data from a relation to the frontend. It then sends a CopyDataRows message, and then a CompletedResponse message with a tag of "COPY".

CursorResponse

The query was either an insert(l), delete(l), update(l), fetch(l) or a select(l) command. If the transaction has been aborted then the backend sends a CompletedResponse message with a tag of "*ABORT STATE*". Otherwise the following responses are sent.

For an insert(l) command, the backend then sends a CompletedResponse message with a tag of "INSERT `oid` `rows`" where `rows` is the number of rows inserted, and `oid` is the object ID of the inserted row if `rows` is 1, otherwise `oid` is 0.

For a delete(l) command, the backend then sends a CompletedResponse message with a tag of "DELETE `rows`" where `rows` is the number of rows deleted.

For an update(l) command, the backend then sends a CompletedResponse message with a tag of "UPDATE `rows`" where `rows` is the number of rows deleted.

For a fetch(l) or select(l) command, the backend sends a RowDescription message. This is then followed by an AsciiRow or BinaryRow message (depending on whether a binary cursor was specified) for each row being returned to the frontend. Finally, the backend sends a CompletedResponse message with a tag of "SELECT".

EmptyQueryResponse

An empty query string was recognized. (The need to specially distinguish this case is historical.)

ErrorResponse

An error has occurred.

ReadyForQuery

Processing of the query string is complete. A separate message is sent to indicate this because the query string may contain multiple SQL commands. (CompletedResponse marks the end of processing one SQL command, not the whole string.) ReadyForQuery will always be sent, whether processing terminates successfully or with an error.

NoticeResponse

A warning message has been issued in relation to the query. Notices are in addition to other responses, i.e., the backend will continue processing the command.

A frontend must be prepared to accept ErrorResponse and NoticeResponse messages whenever it is expecting any other type of message.

Actually, it is possible for NoticeResponse to arrive even when the frontend is not expecting any kind of message, that is, the backend is nominally idle. (In particular, the backend can be commanded to terminate by its postmaster. In that case it will send a NoticeResponse before closing the connection.) It is recommended that the frontend check for such asynchronous notices just before issuing any new command.

Also, if the frontend issues any listen(l) commands then it must be prepared to accept NotificationResponse messages at any time; see below.

4.2.3. Function Call

A Function Call cycle is initiated by the frontend sending a FunctionCall message to the backend. The backend then sends one or more response messages depending on the results of the function call, and finally a ReadyForQuery response message. ReadyForQuery informs the frontend that it may safely send a new query or function call.

The possible response messages from the backend are:

ErrorResponse

An error has occurred.

FunctionResultResponse

The function call was executed and returned a result.

FunctionVoidResponse

The function call was executed and returned no result.

ReadyForQuery

Processing of the function call is complete. ReadyForQuery will always be sent, whether processing terminates successfully or with an error.

NoticeResponse

A warning message has been issued in relation to the function call. Notices are in addition to other responses, i.e., the backend will continue processing the command.

A frontend must be prepared to accept ErrorResponse and NoticeResponse messages whenever it is expecting any other type of message. Also, if it issues any listen(l) commands then it must be prepared to accept NotificationResponse messages at any time; see below.

4.2.4. Notification Responses

If a frontend issues a listen(l) command, then the backend will send a NotificationResponse message (not to be confused with NoticeResponse!) whenever a notify(l) command is executed for the same notification name.

Notification responses are permitted at any point in the protocol (after start-up), except within another backend message. Thus, the frontend must be prepared to recognize a NotificationResponse message whenever it is expecting any message. Indeed, it should be able to handle NotificationResponse messages even when it is not engaged in a query.

NotificationResponse

> A notify(l) command has been executed for a name for which a previous listen(l) command was executed. Notifications may be sent at any time.

It may be worth pointing out that the names used in listen and notify commands need not have anything to do with names of relations (tables) in the SQL database. Notification names are simply arbitrarily chosen condition names.

4.2.5. Cancelling Requests in Progress

During the processing of a query, the frontend may request cancellation of the query by sending an appropriate request to the postmaster. The cancel request is not sent directly to the backend for reasons of implementation efficiency: we don't want to have the backend constantly checking for new input from the frontend during query processing. Cancel requests should be relatively infrequent, so we make them slightly cumbersome in order to avoid a penalty in the normal case.

To issue a cancel request, the frontend opens a new connection to the postmaster and sends a CancelRequest message, rather than the StartupPacket message that would ordinarily be sent across a new connection. The postmaster will process this request and then close the connection. For security reasons, no direct reply is made to the cancel request message.

A CancelRequest message will be ignored unless it contains the same key data (PID and secret key) passed to the frontend during connection start-up. If the request matches the PID and secret key for a currently executing backend, the postmaster signals the backend to abort processing of the current query.

The cancellation signal may or may not have any effect --- for example, if it arrives after the backend has finished processing the query, then it will have no effect. If the cancellation is effective, it results in the current command being terminated early with an error message.

The upshot of all this is that for reasons of both security and efficiency, the frontend has no direct way to tell whether a cancel request has succeeded. It must continue to wait for the backend to respond to the query. Issuing a cancel simply improves the odds that the current query will finish soon, and improves the odds that it will fail with an error message instead of succeeding.

Since the cancel request is sent to the postmaster and not across the regular frontend/backend communication link, it is possible for the cancel request to be issued by any process, not just the frontend whose query is to be canceled. This may have some benefits of flexibility in building multiple-process applications. It also introduces a security risk, in that unauthorized persons might try to cancel queries. The security risk is addressed by requiring a dynamically generated secret key to be supplied in cancel requests.

4.2.6. Termination

The normal, graceful termination procedure is that the frontend sends a Terminate message and immediately closes the connection. On receipt of the message, the backend immediately closes the connection and terminates.

An ungraceful termination may occur due to software failure (i.e., core dump) at either end. If either frontend or backend sees an unexpected closure of the connection, it should clean up and terminate. The frontend has the option of launching a new backend by recontacting the postmaster, if it doesn't want to terminate itself.

4.3. Message Data Types

This section describes the base data types used in messages.

Int*n*(*i*)

An *n* bit integer in network byte order. If *i* is specified it is the literal value. Eg. Int16, Int32(42).

LimString*n*(*s*)

A character array of exactly *n* bytes interpreted as a '\0' terminated string. The '\0' is omitted if there is insufficient room. If *s* is specified it is the literal value. Eg. LimString32, LimString64("user").

String(*s*)

A conventional C '\0' terminated string with no length limitation. If *s* is specified it is the literal value. Eg. String, String("user").

> **Note:** *There is no predefined limit* on the length of a string that can be returned by the backend. Good coding strategy for a frontend is to use an expandable buffer so that anything that fits in memory can be accepted. If that's not feasible, read the full string and discard trailing characters that don't fit into your fixed-size buffer.

Byte*n*(*c*)

Exactly *n* bytes. If *c* is specified it is the literal value. Eg. Byte, Byte1('\n').

4.4. Message Formats

This section describes the detailed format of each message. Each can be sent by either a frontend (F), a postmaster/backend (B), or both (F & B).

AsciiRow (B)

Byte1('D')

Identifies the message as an ASCII data row. (A prior RowDescription message defines the number of fields in the row and their data types.)

Byte*n*

A bit map with one bit for each field in the row. The 1st field corresponds to bit 7 (MSB) of the 1st byte, the 2nd field corresponds to bit 6 of the 1st byte, the 8th field corresponds to bit 0 (LSB) of the 1st byte, the 9th field corresponds to bit 7 of the 2nd byte, and so on. Each bit is set if the value of the corresponding field is not NULL. If the number of fields is not a multiple of 8, the remainder of the last byte in the bit map is wasted.

Then, for each field with a non-NULL value, there is the following:

Int32

Specifies the size of the value of the field, including this size.

Byte*n*

Specifies the value of the field itself in ASCII characters. *n* is the above size minus 4. There is no trailing '\0' in the field data; the front end must add one if it wants one.

AuthenticationOk (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(0)

Specifies that the authentication was successful.

AuthenticationKerberosV4 (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(1)

Specifies that Kerberos V4 authentication is required.

AuthenticationKerberosV5 (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(2)

Specifies that Kerberos V5 authentication is required.

AuthenticationUnencryptedPassword (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(3)

Specifies that an unencrypted password is required.

AuthenticationEncryptedPassword (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(4)

Specifies that an encrypted password is required.

Byte2

The salt to use when encrypting the password.

BackendKeyData (B)

Byte1('K')

Identifies the message as cancellation key data. The frontend must save these values if it wishes to be able to issue CancelRequest messages later.

Int32

The process ID of this backend.

Int32

The secret key of this backend.

BinaryRow (B)

Byte1('B')

Identifies the message as a binary data row. (A prior RowDescription message defines the number of fields in the row and their data types.)

Byte*n*

A bit map with one bit for each field in the row. The 1st field corresponds to bit 7 (MSB) of the 1st byte, the 2nd field corresponds to bit 6 of the 1st byte, the 8th field corresponds to bit 0 (LSB) of the 1st byte, the 9th field corresponds to bit 7 of the 2nd byte, and so on. Each bit is set if the value of the corresponding field is not NULL. If the number of fields is not a multiple of 8, the remainder of the last byte in the bit map is wasted.

Then, for each field with a non-NULL value, there is the following:

Int32

Specifies the size of the value of the field, excluding this size.

Byte*n*

Specifies the value of the field itself in binary format. *n* is the above size.

CancelRequest (F)

Int32(16)

The size of the packet in bytes.

Int32(80877102)

The cancel request code. The value is chosen to contain "1234" in the most significant 16 bits, and "5678" in the least 16 significant bits. (To avoid confusion, this code must not be the same as any protocol version number.)

Int32

The process ID of the target backend.

Int32

The secret key for the target backend.

CompletedResponse (B)

Byte1('C')

Identifies the message as a completed response.

String

The command tag. This is usually (but not always) a single word that identifies which SQL command was completed.

CopyDataRows (B & F)

This is a stream of rows where each row is terminated by a Byte1('\n'). This is then followed by the sequence Byte1('\\'), Byte1('.'), Byte1('\n').

CopyInResponse (B)

Byte1('G')

Identifies the message as a Start Copy In response. The frontend must now send a CopyDataRows message.

CopyOutResponse (B)

Byte1('H')

Identifies the message as a Start Copy Out response. This message will be followed by a CopyDataRows message.

CursorResponse (B)

Byte1('P')

Identifies the message as a cursor response.

String

The name of the cursor. This will be "blank" if the cursor is implicit.

EmptyQueryResponse (B)

Byte1('I')

Identifies the message as a response to an empty query string.

String("")

Unused.

EncryptedPasswordPacket (F)

Int32

The size of the packet in bytes.

String

The encrypted (using crypt()) password.

ErrorResponse (B)

Byte1('E')

Identifies the message as an error.

String

The error message itself.

FunctionCall (F)

Byte1('F')

Identifies the message as a function call.

String("")

Unused.

Int32

Specifies the object ID of the function to call.

Int32

Specifies the number of arguments being supplied to the function.

Then, for each argument, there is the following:

Int32

Specifies the size of the value of the argument, excluding this size.

Byte*n*

Specifies the value of the field itself in binary format. *n* is the above size.

FunctionResultResponse (B)

Byte1('V')

Identifies the message as a function call result.

Byte1('G')

Specifies that a nonempty result was returned.

Int32

Specifies the size of the value of the result, excluding this size.

Byte*n*

Specifies the value of the result itself in binary format. *n* is the above size.

Byte1('0')

Unused. (Strictly speaking, FunctionResultResponse and FunctionVoidResponse are the same thing but with some optional parts to the message.)

FunctionVoidResponse (B)

Byte1('V')

Identifies the message as a function call result.

Byte1('0')

Specifies that an empty result was returned.

NoticeResponse (B)

Byte1('N')

Identifies the message as a notice.

String

The notice message itself.

NotificationResponse (B)

Byte1('A')

Identifies the message as a notification response.

Int32

The process ID of the notifying backend process.

String

The name of the condition that the notify has been raised on.

Query (F)

Byte1('Q')

Identifies the message as a query.

String

The query string itself.

ReadyForQuery (B)

Byte1('Z')

Identifies the message type. ReadyForQuery is sent whenever the backend is ready for a new query cycle.

RowDescription (B)

Byte1('T')

Identifies the message as a row description.

Int16

Specifies the number of fields in a row (may be zero).

Then, for each field, there is the following:

String

Specifies the field name.

Int32

Specifies the object ID of the field type.

Int16

Specifies the type size.

Int32

Specifies the type modifier.

StartupPacket (F)

Int32(296)

The size of the packet in bytes.

Int32

The protocol version number. The most significant 16 bits are the major version number. The least 16 significant bits are the minor version number.

LimString64

The database name, defaults to the user name if empty.

LimString32

The user name.

LimString64

Any additional command line arguments to be passed to the backend by the postmaster.

LimString64

Unused.

LimString64

The optional tty the backend should use for debugging messages.

Terminate (F)

Byte1('X')

Identifies the message as a termination.

UnencryptedPasswordPacket (F)

Int32

The size of the packet in bytes.

String

The unencrypted password.

Chapter 5. gcc Default Optimizations

Note: Contributed by Brian Gallew (<`geek+@cmu.edu`>)

Configuring gcc to use certain flags by default is a simple matter of editing the `/usr/local/lib/gcc-lib/`*`platform`*`/`*`version`*`/specs` file. The format of this file pretty simple. The file is broken into sections, each of which is three lines long. The first line is "*`section_name`*:" (e.g. "*asm:"). The second line is a list of flags, and the third line is blank.

The easiest change to make is to append the desired default flags to the list in the appropriate section. As an example, let's suppose that I have linux running on a '486 with gcc 2.7.2 installed in the default location. In the file /usr/local/lib/gcc-lib/i486-linux/2.7.2/specs, 13 lines down I find the following section:

```
- ----------SECTION----------
*cc1:


- ----------SECTION----------
```

As you can see, there aren't any default flags. If I always wanted compiles of C code to use "-m486 -fomit-frame-pointer", I would change it to look like:

```
- ----------SECTION----------
*cc1:
- -m486 -fomit-frame-pointer

- ----------SECTION----------
```

If I wanted to be able to generate 386 code for another, older linux box lying around, I'd have to make it look like this:

```
- ----------SECTION----------
*cc1:
%{!m386:-m486} -fomit-frame-pointer

- ----------SECTION----------
```

This will always omit frame pointers, any will build 486-optimized code unless -m386 is specified on the command line.

You can actually do quite a lot of customization with the specs file. Always remember, however, that these changes are global, and affect all users of the system.

Chapter 6. BKI Backend Interface

Backend Interface (BKI) files are scripts in a special language that are input to the Postgres backend running in the special bootstrap mode that allows it to perform database functions without a database system already existing. BKI files can therefore be used to create the database system in the first place. (And they are probably not useful for anything else.)

initdb uses BKI files to do part of its job when creating a new database cluster. The input files used by initbd are created as part of building and installing Postgres by a program named `genbki.sh` from some specially formatted C header files in the source tree. The created BKI files are called `global.bki` (for global catalogs) and `template1.bki` (for the catalogs initially stored in the template1 database and then duplicated in every created database) and are normally installed in the `share` subdirectory of the installation tree.

Related information may be found in the documentation for initdb.

6.1. BKI File Format

This section describes how the Postgres backend interprets BKI files. This description will be easier to understand if the `global.bki` file is at hand as an example. You should also study the source code of initdb to get an idea of how the backend is invoked.

BKI input consists of a sequence of commands. Commands are made up of a number of tokens, depending on the syntax of the command. Tokens are usually separated by whitespace, but need not be if there is no ambiguity. There is not special command separator; the next token that syntactically cannot belong to the preceeding command starts a new one. (Usually you would put a new command on a new line, for clarity.) Tokens can be certain key words, special characters (parentheses, commas, etc.), numbers, or double-quoted strings. Everything is case sensitive.

Lines starting with a # are ignored.

6.2. BKI Commands

open *tablename*

Open the table called *tablename* for further manipulation.

close [*tablename*]

Close the open table called *tablename*. It is an error if *tablename* is not already opened. If no *tablename* is given, then the currently open table is closed.

create *tablename* (*name1* = *type1* [, *name2* = *type2*, ...])

Create a table named *tablename* with the columns given in parentheses.

The *type* is not necessarily the data type that the column will have in the SQL environment; that is determined by the `pg_attribute` system catalog. The type here is essentially only used to allocate storage. The following types are allowed: `bool`, `bytea`, `char` (1 byte), `name`, `int2`, `int2vector`, `int4`, `regproc`, `text`, `oid`, `tid`, `xid`, `cid`, `oidvector`, `smgr`, `_int4` (array), `_aclitem` (array). Array types can also be indicated by writing `[]` after the name of the element type.

> **Note:** The table will only be created on disk, it will not automatically be registered in the system catalogs and will therefore not be accessible unless appropriate rows are inserted in `pg_class`, `pg_attribute`, etc.

insert [OID = *`oid_value`*] (*`value1`* *`value2`* ...)

Insert a new row into the open table using *`value1`*, *`value2`*, etc., for its column values and *`oid_value`* for its OID. If *`oid_value`* is zero (0) or the clause is ommitted, then the next available OID is used.

NULL values can be specified using the special key word `_null_`. Values containing spaces should be double quoted.

declare [unique] index *`indexname`* on *`tablename`* using *`amname`* (*`opclass1`* *`name1`* [, ...])

Create an index named *`indexname`* on the table named *`tablename`* using the *`amname`* access method. The fields to index are called *`name1`*, *`name2`* etc., and the operator classes to use are *`opclass1`*, *`opclass2`* etc., respectively.

build indices

Build the indices that have previously been declared.

6.3. Example

The following sequence of commands will create the `test_table` table with the two columns `cola` and `colb` of type `int4` and `text`, respectively, and insert two rows into the table.

```
create test_table (cola = int4, colb = text)
open test_table
insert OID=421 ( 1 "value1" )
insert OID=422 ( 2 _null_ )
close test_table
```

Chapter 7. Page Files

A description of the database file default page format.

This section provides an overview of the page format used by Postgres tables. User-defined access methods need not use this page format.

In the following explanation, a *byte* is assumed to contain 8 bits. In addition, the term *item* refers to data that is stored in Postgres tables.

The following table shows how pages in both normal Postgres tables and Postgres indices (e.g., a B-tree index) are structured.

Table 7-1. Sample Page Layout

Item	Description
itemPointerData	
filler	
itemData...	
Unallocated Space	
ItemContinuationData	
Special Space	
‘‘ItemData 2’’	
‘‘ItemData 1’’	
ItemIdData	
PageHeaderData	

The first 8 bytes of each page consists of a page header (PageHeaderData). Within the header, the first three 2-byte integer fields (*lower*, *upper*, and *special*) represent byte offsets to the start of unallocated space, to the end of unallocated space, and to the start of *special space*. Special space is a region at the end of the page that is allocated at page initialization time and contains information specific to an access method. The last 2 bytes of the page header, *opaque*, encode the page size and information on the internal fragmentation of the page. Page size is stored in each page because frames in the buffer pool may be subdivided into equal sized pages on a frame by frame basis within a table. The internal fragmentation information is used to aid in determining when page reorganization should occur.

Following the page header are item identifiers (*ItemIdData*). New item identifiers are allocated from the first four bytes of unallocated space. Because an item identifier is never moved until it is freed, its index may be used to indicate the location of an item on a page. In fact, every pointer to an item (*ItemPointer*) created by Postgres consists of a frame number and an index of an item identifier. An item identifier contains a byte-offset to the start of an item, its length in bytes, and a set of attribute bits which affect its interpretation.

The items themselves are stored in space allocated backwards from the end of unallocated space. Usually, the items are not interpreted. However when the item is too long to be placed on a single page or when fragmentation of the item is desired, the item is divided and each piece is handled as distinct items in the following manner. The first through the next to last piece are placed in an item continuation structure (*ItemContinuationData*). This structure contains itemPointerData which points to the next piece and the piece itself. The last piece is handled normally.

Chapter 8. Genetic Query Optimization

Author: Written by Martin Utesch (<`utesch@aut.tu-freiberg.de`>) for the Institute of Automatic Control at the University of Mining and Technology in Freiberg, Germany.

8.1. Query Handling as a Complex Optimization Problem

Among all relational operators the most difficult one to process and optimize is the *join*. The number of alternative plans to answer a query grows exponentially with the number of **join**s included in it. Further optimization effort is caused by the support of a variety of *join methods* (e.g., nested loop, hash join, merge join in Postgres) to process individual **join**s and a diversity of *indices* (e.g., r-tree, b-tree, hash in Postgres) as access paths for relations.

The current Postgres optimizer implementation performs a *near-exhaustive search* over the space of alternative strategies. This query optimization technique is inadequate to support database application domains that involve the need for extensive queries, such as artificial intelligence.

The Institute of Automatic Control at the University of Mining and Technology, in Freiberg, Germany, encountered the described problems as its folks wanted to take the Postgres DBMS as the backend for a decision support knowledge based system for the maintenance of an electrical power grid. The DBMS needed to handle large **join** queries for the inference machine of the knowledge based system.

Performance difficulties in exploring the space of possible query plans created the demand for a new optimization technique being developed.

In the following we propose the implementation of a *Genetic Algorithm* as an option for the database query optimization problem.

8.2. Genetic Algorithms (GA)

The GA is a heuristic optimization method which operates through determined, randomized search. The set of possible solutions for the optimization problem is considered as a *population* of *individuals*. The degree of adaption of an individual to its environment is specified by its *fitness*.

The coordinates of an individual in the search space are represented by *chromosomes*, in essence a set of character strings. A *gene* is a subsection of a chromosome which encodes the value of a single parameter being optimized. Typical encodings for a gene could be *binary* or *integer*.

Through simulation of the evolutionary operations *recombination*, *mutation*, and *selection* new generations of search points are found that show a higher average fitness than their ancestors.

According to the "comp.ai.genetic" FAQ it cannot be stressed too strongly that a GA is not a pure random search for a solution to a problem. A GA uses stochastic processes, but the result is distinctly non-random (better than random).

```
Structured Diagram of a GA:
---------------------------

P(t)    generation of ancestors at a time t
P''(t)  generation of descendants at a time t

+=========================================+
|>>>>>>>>>>>  Algorithm GA  <<<<<<<<<<<<<<|
+=========================================+
| INITIALIZE t := 0                       |
+=========================================+
| INITIALIZE P(t)                         |
+=========================================+
| evalute FITNESS of P(t)                 |
+=========================================+
| while not STOPPING CRITERION do         |
|   +-------------------------------------+
|   | P'(t)  := RECOMBINATION{P(t)}       |
|   +-------------------------------------+
|   | P''(t) := MUTATION{P'(t)}           |
|   +-------------------------------------+
|   | P(t+1) := SELECTION{P''(t) + P(t)}  |
|   +-------------------------------------+
|   | evalute FITNESS of P''(t)           |
|   +-------------------------------------+
|   | t := t + 1                          |
+===+=====================================+
```

8.3. Genetic Query Optimization (GEQO) in Postgres

The GEQO module is intended for the solution of the query optimization problem similar to a traveling salesman problem (TSP). Possible query plans are encoded as integer strings. Each string represents the **join** order from one relation of the query to the next. E. g., the query tree

```
   /\
  /\ 2
 /\ 3
4  1
```

is encoded by the integer string '4-1-3-2', which means, first join relation '4' and '1', then '3', and then '2', where 1, 2, 3, 4 are relids within the Postgres optimizer.

Parts of the GEQO module are adapted from D. Whitley's Genitor algorithm.

Specific characteristics of the GEQO implementation in Postgres are:

- Usage of a *steady state* GA (replacement of the least fit individuals in a population, not whole-generational replacement) allows fast convergence towards improved query plans. This is essential for query handling with reasonable time;
- Usage of *edge recombination crossover* which is especially suited to keep edge losses low for the solution of the TSP by means of a GA;

Mutation as genetic operator is deprecated so that no repair mechanisms are needed to generate legal TSP tours.

The GEQO module allows the Postgres query optimizer to support large **join** queries effectively through non-exhaustive search.

8.3.1. Future Implementation Tasks for PostgreSQL GEQO

Work is still needed to improve the genetic algorithm parameter settings. In file `backend/optimizer/geqo/geqo_params.c`, routines `gimme_pool_size` and `gimme_number_generations`, we have to find a compromise for the parameter settings to satisfy two competing demands:

Optimality of the query plan

Computing time

References

Reference information for GEQ algorithms.

The Hitch-Hiker's Guide to Evolutionary Computation , Jörg Heitkötter and David Beasley, InterNet resource , *The Design and Implementation of the Postgres Query Optimizer* , Z. Fong, University of California, Berkeley Computer Science Department , *Fundamentals of Database Systems* , R. Elmasri and S. Navathe, The Benjamin/Cummings Pub., Inc. .

FAQ in comp.ai.genetic (news://comp.ai.genetic) is available at Encore (ftp://ftp.Germany.EU.net/pub/research/softcomp/EC/Welcome.html).

File `planner/Report.ps` in the 'postgres-papers' distribution.

Bibliography

Selected references and readings for SQL and Postgres.

Some white papers and technical reports from the original Postgres development team are available at the University of California, Berkeley, Computer Science Department web site (http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/)

SQL Reference Books

The Practical SQL Handbook , Bowman et al, 1996 , *Using Structured Query Language* , 3, Judith Bowman, Sandra Emerson, and Marcy Darnovsky, 0-201-44787-8, 1996, Addison-Wesley, 1996.

A Guide to the SQL Standard , Date and Darwen, 1997 , *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 , Date, 1994 , 6, C. J. Date, 1, 1994, Addison-Wesley, 1994.

Understanding the New SQL , Melton and Simon, 1993 , *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 , Ullman, 1988 , Jeffrey D. Ullman, 1, Computer Science Press , 1988 .

PostgreSQL-Specific Documentation

The PostgreSQL Administrator's Guide , The Administrator's Guide , Edited by Thomas Lockhart, 2001-04-13, The PostgreSQL Global Development Group.

The PostgreSQL Developer's Guide , The Developer's Guide , Edited by Thomas Lockhart, 2001-04-13, The PostgreSQL Global Development Group.

The PostgreSQL Programmer's Guide , The Programmer's Guide , Edited by Thomas Lockhart, 2001-04-13, The PostgreSQL Global Development Group.

The PostgreSQL Tutorial Introduction , The Tutorial , Edited by Thomas Lockhart, 2001-04-13, The PostgreSQL Global Development Group.

The PostgreSQL User's Guide , The User's Guide , Edited by Thomas Lockhart, 2001-04-13, 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) , Seshardri, 1995 , 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.

Appendix DG1. The CVS Repository

The Postgres source code is stored and managed using the CVS code management system.

At least two methods, anonymous CVS and CVSup, are available to pull the CVS code tree from the Postgres server to your local machine.

DG1.1. Getting The Source Via Anonymous CVS

If you would like to keep up with the current sources on a regular basis, you can fetch them from our CVS server and then use CVS to retrieve updates from time to time.

Anonymous CVS

1. You will need a local copy of CVS (Concurrent Version Control System), which you can get from http://www.cyclic.com/ or any GNU software archive site. We currently recommend version 1.10 (the most recent at the time of writing). Many systems have a recent version of cvs installed by default.
2. Do an initial login to the CVS server:

   ```
   $ cvs -d :pserver:anoncvs@postgresql.org:/home/projects/pgsql/cvsroot
   login
   ```

   You will be prompted for a password; enter 'postgresql'. You should only need to do this once, since the password will be saved in `.cvspass` in your home directory.
3. Fetch the Postgres sources:

   ```
   cvs -z3 -d :pserver:anoncvs@postgresql.org:/home/projects/pgsql/cvsroot
   co -P pgsql
   ```

   which installs the Postgres sources into a subdirectory `pgsql` of the directory you are currently in.

   Note: If you have a fast link to the Internet, you may not need `-z3`, which instructs CVS to use gzip compression for transferred data. But on a modem-speed link, it's a very substantial win.

   This initial checkout is a little slower than simply downloading a `tar.gz` file; expect it to take 40 minutes or so if you have a 28.8K modem. The advantage of CVS doesn't show up until you want to update the file set later on.
4. Whenever you want to update to the latest CVS sources, **cd** into the `pgsql` subdirectory, and issue

   ```
   $ cvs -z3 update -d -P
   ```

   This will fetch only the changes since the last time you updated. You can update in just a couple of minutes, typically, even over a modem-speed line.
5. You can save yourself some typing by making a file `.cvsrc` in your home directory that

contains

```
cvs -z3
update -d -P
```

This supplies the `-z3` option to all cvs commands, and the `-d` and `-P` options to cvs update. Then you just have to say

```
$ cvs update
```

to update your files.

Caution

Some older versions of CVS have a bug that causes all checked-out files to be stored world-writable in your directory. If you see that this has happened, you can do something like

```
$ chmod -R go-w pgsql
```

to set the permissions properly. This bug is fixed as of CVS version 1.9.28.

CVS can do a lot of other things, such as fetching prior revisions of the Postgres sources rather than the latest development version. For more info consult the manual that comes with CVS, or see the online documentation at http://www.cyclic.com/.

DG1.2. CVS Tree Organization

Author: Written by Marc G. Fournier (<`scrappy@hub.org`>) on 1998-11-05

The command **cvs checkout** has a flag, `-r`, that lets you check out a certain revision of a module. This flag makes it easy to, for example, retrieve the sources that make up release 6_4 of the module 'tc' at any time in the future:

```
$ cvs checkout -r REL6_4 tc
```

This is useful, for instance, if someone claims that there is a bug in that release, but you cannot find the bug in the current working copy.

Tip: You can also check out a module as it was at any given date using the `-D` option.

When you tag more than one file with the same tag you can think about the tag as "a curve drawn through a matrix of filename vs. revision number". Say we have 5 files with the following revisions:

```
file1   file2   file3   file4   file5

1.1     1.1     1.1     1.1  /--1.1*      <-*-  TAG
1.2*-   1.2     1.2    -1.2*-
1.3  \- 1.3*-   1.3   / 1.3
1.4          \  1.4  /  1.4
              \-1.5*-   1.5
                1.6
```

then the tag "`TAG`" will reference file1-1.2, file2-1.3, etc.

> **Note:** For creating a release branch, other then a -b option added to the command, it's the same thing.

So, to create the 6.4 release I did the following:

```
$ cd pgsql
$ cvs tag -b REL6_4
```

which will create the tag and the branch for the RELEASE tree.

For those with CVS access, it's simple to create directories for different versions. First, create two subdirectories, RELEASE and CURRENT, so that you don't mix up the two. Then do:

```
cd RELEASE
cvs checkout -P -r REL6_4 pgsql
cd ../CURRENT
cvs checkout -P pgsql
```

which results in two directory trees, `RELEASE/pgsql` and `CURRENT/pgsql`. From that point on, CVS will keep track of which repository branch is in which directory tree, and will allow independent updates of either tree.

If you are *only* working on the `CURRENT` source tree, you just do everything as before we started tagging release branches.

After you've done the initial checkout on a branch

```
$ cvs checkout -r REL6_4
```

anything you do within that directory structure is restricted to that branch. If you apply a patch to that directory structure and do a

```
cvs commit
```

while inside of it, the patch is applied to the branch and *only* the branch.

DG1.3. Getting The Source Via CVSup

An alternative to using anonymous CVS for retrieving the Postgres source tree is CVSup. CVSup was developed by John Polstra (`<jdp@polstra.com>`) to distribute CVS repositories and other file trees for the FreeBSD project (http://www.freebsd.org).

A major advantage to using CVSup is that it can reliably replicate the *entire* CVS repository on your local system, allowing fast local access to cvs operations such as `log` and `diff`. Other advantages include fast synchronization to the Postgres server due to an efficient streaming transfer protocol which only sends the changes since the last update.

DG1.3.1. Preparing A CVSup Client System

Two directory areas are required for CVSup to do it's job: a local CVS repository (or simply a directory area if you are fetching a snapshot rather than a repository; see below) and a local CVSup bookkeeping area. These can coexist in the same directory tree.

Decide where you want to keep your local copy of the CVS repository. On one of our systems we recently set up a repository in `/home/cvs/`, but had formerly kept it under a Postgres development tree in `/opt/postgres/cvs/`. If you intend to keep your repository in `/home/cvs/`, then put

```
setenv CVSROOT /home/cvs
```

in your `.cshrc` file, or a similar line in your `.bashrc` or `.profile` file, depending on your shell.

The cvs repository area must be initialized. Once CVSROOT is set, then this can be done with a single command:

```
$ cvs init
```

after which you should see at least a directory named `CVSROOT` when listing the CVSROOT directory:

```
$ ls $CVSROOT
CVSROOT/
```

DG1.3.2. Running a CVSup Client

Verify that cvsup is in your path; on most systems you can do this by typing

```
which cvsup
```

Then, simply run cvsup using:

```
$ cvsup -L 2 postgres.cvsup
```

where `-L 2` enables some status messages so you can monitor the progress of the update, and *`postgres.cvsup`* is the path and name you have given to your CVSup configuration file.

Here is a CVSup configuration file modified for a specific installation, and which maintains a full local CVS repository:

```
# This file represents the standard CVSup distribution file
# for the PostgreSQL ORDBMS project
# Modified by lockhart@alumni.caltech.edu 1997-08-28
# - Point to my local snapshot source tree
# - Pull the full CVS repository, not just the latest snapshot
#
# Defaults that apply to all the collections
*default host=postgresql.org
*default compress
*default release=cvs
*default delete use-rel-suffix
# enable the following line to get the latest snapshot
#*default tag=.
# enable the following line to get whatever was specified above or by
default
# at the date specified below
#*default date=97.08.29.00.00.00

# base directory points to where CVSup will store its 'bookmarks' file(s)
# will create subdirectory sup/
#*default base=/opt/postgres # /usr/local/pgsql
*default base=/home/cvs

# prefix directory points to where CVSup will store the actual
distribution(s)
*default prefix=/home/cvs

# complete distribution, including all below
pgsql

# individual distributions vs 'the whole thing'
# pgsql-doc
# pgsql-perl5
# pgsql-src
```

The following is a suggested CVSup config file from the Postgres ftp site (ftp://ftp.postgresql.org/pub/CVSup/README.cvsup) which will fetch the current snapshot only:

```
# This file represents the standard CVSup distribution file
# for the PostgreSQL ORDBMS project
#
# Defaults that apply to all the collections
*default host=postgresql.org
*default compress
*default release=cvs
*default delete use-rel-suffix
*default tag=.

# base directory points to where CVSup will store its 'bookmarks' file(s)
*default base=/usr/local/pgsql
```

```
# prefix directory points to where CVSup will store the actual distribution(s)
*default prefix=/usr/local/pgsql

# complete distribution, including all below
pgsql

# individual distributions vs 'the whole thing'
# pgsql-doc
# pgsql-perl5
# pgsql-src
```

DG1.3.3. Installing CVSup

CVSup is available as source, pre-built binaries, or Linux RPMs. It is far easier to use a binary than to build from source, primarily because the very capable, but voluminous, Modula-3 compiler is required for the build.

CVSup Installation from Binaries

You can use pre-built binaries if you have a platform for which binaries are posted on the Postgres ftp site (ftp://postgresql.org/pub), or if you are running FreeBSD, for which CVSup is available as a port.

> **Note:** CVSup was originally developed as a tool for distributing the FreeBSD source tree. It is available as a "port", and for those running FreeBSD, if this is not sufficient to tell how to obtain and install it then please contribute a procedure here.

At the time of writing, binaries are available for Alpha/Tru64, ix86/xBSD, HPPA/HPUX-10.20, MIPS/irix, ix86/linux-libc5, ix86/linux-glibc, Sparc/Solaris, and Sparc/SunOS.

1. Retrieve the binary tar file for cvsup (cvsupd is not required to be a client) appropriate for your platform.
   a. If you are running FreeBSD, install the CVSup port.
   b. If you have another platform, check for and download the appropriate binary from the Postgres ftp site (ftp://postgresql.org/pub).
2. Check the tar file to verify the contents and directory structure, if any. For the linux tar file at least, the static binary and man page is included without any directory packaging.
   a. If the binary is in the top level of the tar file, then simply unpack the tar file into your target directory:

      ```
      $ cd /usr/local/bin
      $ tar zxvf /usr/local/src/cvsup-16.0-linux-i386.tar.gz
      $ mv cvsup.1 ../doc/man/man1/
      ```

   b. If there is a directory structure in the tar file, then unpack the tar file within /usr/local/src and move the binaries into the appropriate location as above.

3. Ensure that the new binaries are in your path.

```
$ rehash
$ which cvsup
$ set path=(path to cvsup $path)
$ which cvsup
/usr/local/bin/cvsup
```

DG1.3.4. Installation from Sources

Installing CVSup from sources is not entirely trivial, primarily because most systems will need to install a Modula-3 compiler first. This compiler is available as Linux RPM, FreeBSD package, or source code.

> **Note:** A clean-source installation of Modula-3 takes roughly 200MB of disk space, which shrinks to roughly 50MB of space when the sources are removed.

Linux installation

1. Install Modula-3.
   a. Pick up the Modula-3 distribution from Polytechnique Montréal (http://m3.polymtl.ca/m3), who are actively maintaining the code base originally developed by the DEC Systems Research Center (http://www.research.digital.com/SRC/modula-3/html/home.html). The PM3 RPM distribution is roughly 30MB compressed. At the time of writing, the 1.1.10-1 release installed cleanly on RH-5.2, whereas the 1.1.11-1 release is apparently built for another release (RH-6.0?) and does not run on RH-5.2.

      > **Tip:** This particular rpm packaging has *many* RPM files, so you will likely want to place them into a separate directory.

   b. Install the Modula-3 rpms:

      ```
      # rpm -Uvh pm3*.rpm
      ```

2. Unpack the cvsup distribution:

   ```
   # cd /usr/local/src
   # tar zxf cvsup-16.0.tar.gz
   ```

3. Build the cvsup distribution, suppressing the GUI interface feature to avoid requiring X11 libraries:

   ```
   # make M3FLAGS="-DNOGUI"
   ```

   and if you want to build a static binary to move to systems that may not have Modula-3 installed, try:

```
# make M3FLAGS="-DNOGUI -DSTATIC"
```

4. Install the built binary:

```
# make M3FLAGS="-DNOGUI -DSTATIC" install
```

Appendix DG2. Documentation

PostgreSQL has four primary documentation formats:

- Plain text, for pre-installation information
- HTML, for on-line browsing and reference
- Postscript, for printing
- man pages, for quick reference.

Additionally, a number of plain-text README-type files can be found throughout the PostgreSQL source tree, documenting various implementation issues.

The documentation is organized into several books:

- *Tutorial*: introduction for new users
- *User's Guide*: documents the query language environment
- *Reference Manual*: documents the query language
- *Administrator's Guide*: installation and server maintenance
- *Programmer's Guide*: programming client applications and server extensions
- *Developer's Guide*: assorted information for developers of PostgreSQL proper

All books are available as HTML and Postscript. The *Reference Manual* contains reference entries which are also shipped as man pages.

HTML documentation and man pages are part of a standard distribution and are installed by default. Postscript format documentation is available separately for download.

DG2.1. DocBook

The documentation sources are written in *DocBook*, which is a markup language superficially similar to HTML. Both of these languages are applications of the *Standard Generalized Markup Language*, SGML, which is essentially a language for describing other languages. In what follows, the terms DocBook and SGML are both used, but technically they are not interchangeable.

DocBook allows an author to specify the structure and content of a technical document without worrying about presentation details. A document style defines how that content is rendered into one of several final forms. DocBook is maintained by the OASIS (http://www.oasis-open.org) group. The official DocBook site (http://www.oasis-open.org/docbook) has good introductory and reference documentation and a complete O'Reilly book for your online reading pleasure. The FreeBSD Documentation Project (http://www.freebsd.org/docproj/docproj.html) also uses DocBook and has some good information, including a number of style guidelines that might be worth considering.

DG2.2. Toolsets

The following tools are used to process the documentation. Some may be optional, as noted.

DocBook DTD (http://www.oasis-open.org/docbook/sgml/)

> This is the definition of DocBook itself. We currently use version 3.1; you cannot use later or earlier versions. Note that there is also an XML version of DocBook -- do not use that.

ISO 8879 character entities (http://www.oasis-open.org/cover/ISOEnts.zip)

These are required by DocBook but are distributed separately because they are maintained by ISO.

Jade (http://openjade.sourceforge.net)

This is the base package of SGML processing. It contains an SGML parser, a DSSSL processor (that is, a program to convert SGML to other formats using DSSSL stylesheets), as well as a number of related tools. Jade is now being maintained by the OpenJade group, no longer by James Clark.

Norm Walsh's Modular DocBook Stylesheets (http://nwalsh.com/docbook/dsssl/index.html)

These contain the processing instructions for converting the DocBook sources to other formats, such as HTML.

DocBook2X tools (http://docbook2x.sourceforge.net)

This optional package is used to create man pages. It has a number of prerequisite packages of its own. Check the web site.

JadeTeX

If you want to, you can also install JadeTeX to use TeX as a formatting backend for Jade. This will generate printed output that is inferior to what you get from the RTF backend. Tables are a particular problem area. Also, there is no opportunity to manually polish the results. Still, it works all right, especially for simpler documents that don't use tables, and as both JadeTeX and the style sheets are under continuous improvement, it will certainly get better over time.

We have documented experience with several installation methods for the various tools that are needed to process the documentation. These will be described below. There may be some other packaged distributions for these tools. Please report package status to the docs mailing list and we will include that information here.

DG2.2.1. Linux RPM Installation

Many vendors provide a complete RPM set for DocBook processing in their distribution, which is usually based on the docbook-tools (http://sources.redhat.com/docbook-tools/) effort at Red Hat Software. Look for an SGML option while installing, or the following packages: `sgml-common`, `docbook`, `stylesheets`, `openjade` (or `jade`). Possibly `sgml-tools` will be needed as well. If your distributor does not provide these then you should be able to make use of the packages from some large, reasonably compatible vendor.

DG2.2.2. FreeBSD Installation

The FreeBSD Documentation Project is itself a heavy user of DocBook, so it comes as no surprise that there is a full set of ports of the documentation tools available on FreeBSD. The following ports need to be installed to build the documentation on FreeBSD.

`textproc/sp`

`textproc/openjade`

`textproc/docbook-310`

`textproc/iso8879`

```
textproc/dsssl-docbook-modular
```

A number of things from `/usr/ports/print` (`tex`, `jadetex`) might also be of interest.

It's possible that the ports do not update the main catalog file in `/usr/local/share/sgml/catalog`. Be sure to have the following line in there:

```
CATALOG "/usr/local/share/sgml/docbook/3.1/catalog"
```

If you do not want to edit the file you can also set the environment variable SGML_CATALOG_FILES to a colon-separated list of catalog files (such as the one above).

More information about the FreeBSD documentation tools can be found in the FreeBSD Documentation Project's instructions (http://www.freebsd.org/tutorials/docproj-primer/tools.html).

DG2.2.3. Debian Packages

There is a full set of packages of the documentation tools available for Debian GNU/Linux. To install, simply use:

```
apt-get install jade
apt-get install docbook
apt-get install docbook-stylesheets
```

DG2.2.4. Manual Installation from Source

The manual installation process of the DocBook tools is somewhat complex, so if you have pre-built packages available, use them. We describe here only a standard setup, with reasonabley standard installation paths, and no fancy features. For details, you should study the documentation of the respective package, and read SGML introductory material.

DG2.2.4.1. Installing Jade

The installation of OpenJade offers a GNU-style `./configure; make; make install` build process. Details can be found in the OpenJade source distribution. In a nutshell:

```
./configure --enable-default-catalog=/usr/local/share/sgml/catalog
make
make install
```

Be sure to remember where you put the default catalog ; you will need it below. You can also leave it off, but then you will have to set the environment variable SGML_CATALOG_FILES to point to the file whenever you use jade later on.

Additionally, you should install the files `dsssl.dtd`, `fot.dtd`, `style-sheet.dtd`, and `catalog` from the `dsssl` directory somewhere, perhaps into `/usr/local/share/sgml/dsssl`. (Or just copy the entire directory.)

DG2.2.4.2. Installing the DocBook DTD Kit

1. Obtain the DocBook V3.1 (http://www.oasis-open.org/docbook/sgml/3.1/docbk31.zip) distribution.
2. Unpack the archive.

   $ **unzip -a docbk31.zip**

3. Place the files into the directory `/usr/local/share/sgml/docbook31`. (The exact location is irrelevant, but this one is fairly standard.)
4. Create a file `/usr/local/share/sgml/catalog` (or whatever you told jade during installation) and put a line like this into it:

   ```
   CATALOG "docbook31/docbook.cat"
   ```

   Optionally, you can edit the file `docbook.cat` and comment out or remove the line containing `DTDDECL`. If you do not then you will get warnings from jade, but there is no further harm.
5. Download the ISO 8879 character entities (http://www.oasis-open.org/cover/ISOEnts.zip) archive, unpack it, and put the files in the same directory you put the DocBook files in.

DG2.2.4.3. Installing Norman Walsh's DSSSL Style Sheets

To install the style sheets, simply unzip the distribution kit in a suitable place, for example `/usr/local/share/sgml/stylesheets`. (The archive will automatically create a `docbook` subdirectory.)

DG2.2.4.4. Installing JadeTeX

To install and use JadeTeX, you will need a working installation of TeX and LaTeX2e, including the supported tools and graphics packages, Babel, AMS fonts and AMS-LaTeX, the PSNFSS extension and companion kit of the 35 fonts, the dvips program for generating PostScript, the macro packages fancyhdr, hyperref, minitoc, url and ot2enc, and of course JadeTeX itself. All of these can be found on your friendly neighborhood CTAN (http://www.ctan.org) site.

JadeTeX does not at the time of writing come with much of an installation guide, but there is a `makefile` that shows what is needed. It also includes a directory `cooked`, wherein you'll find some of the macro packages it needs, but not all, and not complete -- at least last we looked.

Before building the `jadetex.fmt` format file, you'll probably want to edit the `jadetex.ltx` file, to change the configuration of Babel to suit your locality. The line to change looks something like

```
\RequirePackage[german,french,english]{babel}[1997/01/23]
```

and you should obviously list only the languages you actually need, and have configured Babel for.

It is quite likely that when you use JadeTeX with PostgreSQL documentation sources, that TeX will stop during the second run, and tell you that its capacity has been exceeded. This is, as far as we can tell, because of the way JadeTeX generates cross referencing information. TeX can, of course, be compiled with larger data structure sizes. The details of this will vary according to your installation.

DG2.3. Building The Documentation

Before you can build the documentation you need to run the `configure` script as you would when building the programs themselves. Check the output near the end of the run, it should look something like this:

```
checking for onsgmls... onsgmls
checking for openjade... openjade
checking for DocBook V3.1... yes
```

```
checking for DocBook stylesheets... /usr/lib/sgml/stylesheets/nwalsh-modular
checking for sgmlspl... sgmlspl
```

If neither `onsgmls` nor `nsgmls` were found then you will not see the remaining 4 lines. `nsgmls` is part of the Jade package. If DocBook V3.1 was not found then you did not install the DocBook DTD kit in a place where jade can find it, or you have not set up the catalog files correctly. See the installation hints above. The DocBook stylesheets are looked for in a number of relatively standard places, but if you have them some other place then you should set the environment variable DOCBOOKSTYLE to the location and rerun `configure` afterwards.

Once you have everything set up, change to the directory `doc/src/sgml` and run one of the following commands: (Remember to use GNU make.)

- To build the HTML version of the *Administrator's Guide*:

  ```
  doc/src/sgml$ gmake admin.html
  ```

- For the RTF version of the same:

  ```
  doc/src/sgml$ gmake admin.rtf
  ```

- To get a DVI version via JadeTeX:

  ```
  doc/src/sgml$ gmake admin.dvi
  ```

- And Postscript from the DVI:

  ```
  doc/src/sgml$ gmake admin.ps
  ```

> **Note:** The official Postscript format documentation is generated differently. See Section DG2.3.3 below.

The other books can be built with analogous commands by replacing `admin` with one of `developer`, `programmer`, `tutorial`, or `user`. Using `postgres` builds an integrated version of all 5 books, which is practical since the browser interface makes it easy to move around all of the documentation by just clicking.

DG2.3.1. HTML

When building HTML documentation in `doc/src/sgml`, some of the resulting files will possibly (or quite certainly) have conflicting names between books. Therefore the files are not in that directory in the regular distribution. Instead, the files belonging to each book are stored in a tar archive that is unpacked at installation time. To create a set of HTML documentation packages use the commands

```
cd doc/src
gmake tutorial.tar.gz
gmake user.tar.gz
gmake admin.tar.gz
gmake programmer.tar.gz
gmake postgres.tar.gz
gmake install
```

In the distribution, these archives live in the `doc` directory and are installed by default with **gmake install**.

DG2.3.2. Manpages

We use the docbook2man utility to convert DocBook REFENTRY pages to *roff output suitable for man pages. The man pages are also distributed as a tar archive, similar to the HTML version. To create the man page package, use the commands

```
cd doc/src
gmake man
```

which will result in a tar file being generated in the `doc/src` directory.

The man build leaves a lot of confusing output, and special care must be taken to produce quality results. There is still room for improvement in this area.

DG2.3.3. Hardcopy Generation

The hardcopy Postscript documentation is generated by converting the SGML source code to RTF, then importing into ApplixWare-4.4.1. After a little cleanup (see the following section) the output is "printed" to a postscript file.

Several areas are addressed while generating Postscript hardcopy, including RTF repair, ToC generation, and page break adjustments.

Applixware RTF Cleanup

jade, an integral part of the hardcopy procedure, omits specifying a default style for body text. In the past, this undiagnosed problem led to a long process of Table of Contents (ToC) generation. However, with great help from the ApplixWare folks the symptom was diagnosed and a workaround is available.

1. Generate the RTF input by typing (for example):

   ```
   % cd doc/src/sgml
   % make tutorial.rtf
   ```

2. Repair the RTF file to correctly specify all styles, in particular the default style. The field can be added using vi or the following small sed procedure:

   ```
   #!/bin/sh
   # fixrtf.sh
   # Utility to repair slight damage in RTF files generated by jade
   # Thomas Lockhart <lockhart@alumni.caltech.edu>
   #
   for i in $* ; do
     mv $i $i.orig
     cat $i.orig | sed 's#\\stylesheet#\\stylesheet{\\s0 Normal;}#' > $i
   done

   exit
   ```

where the script is adding `{\s0 Normal;}` as the zero-th style in the document. According to ApplixWare, the RTF standard would prohibit adding an implicit zero-th style, though M$Word happens to handle this case.

3. Open a new document in Applix Words and then import the RTF file.
4. Generate a new ToC using ApplixWare.
   a. Select the existing ToC lines, from the beginning of the first character on the first line to the last character of the last line.
   b. Build a new ToC using `Tools.BookBuilding.CreateToC`. Select the first three levels of headers for inclusion in the ToC. This will replace the existing lines imported in the RTF with a native ApplixWare ToC.
   c. Adjust the ToC formatting by using `Format.Style`, selecting each of the three ToC styles, and adjusting the indents for `First` and `Left`. Use the following values:

Table DG2-1. Indent Formatting for Table of Contents

Style	First Indent (inches)	Left Indent (inches)
`TOC-Heading 1`	`0.6`	`0.6`
`TOC-Heading 2`	`1.0`	`1.0`
`TOC-Heading 3`	`1.4`	`1.4`

5. Work through the document to:

   Adjust page breaks.

   Adjust table column widths.

   Insert figures into the document. Center each figure on the page using the centering margins button on the ApplixWare toolbar.

   Note: Not all documents have figures. You can grep the SGML source files for the string "`graphic`" to identify those parts of the documentation that may have figures. A few figures are replicated in various parts of the documentation.

6. Replace the right-justified page numbers in the Examples and Figures portions of the ToC with correct values. This only takes a few minutes per document.
7. If a bibliography is present, remove the *short form* reference title from each entry. The DocBook stylesheets from Norm Walsh seem to print these out, even though this is a subset of the information immediately following.
8. Save the document as native Applix Words format to allow easier last minute editing later.
9. "Print" the document to a file in Postscript format.
10. Compress the Postscript file using gzip. Place the compressed file into the `doc` directory.

DG2.3.4. Plain Text Files

Several files are distributed as plain text, for reading during the installation process. The `INSTALL` file corresponds to the chapter in the *Administrator's Guide*, with some minor changes to account for the different context. To recreate the file, change to the directory `doc/src/sgml` and enter **`gmake INSTALL`**. This will create a file `INSTALL.html` that can be saved as text with Netscape Navigator and put into the place of the existing file. Netscape seems to offer the best quality for HTML to text conversions (over lynx and w3m).

The file `HISTORY` can be created similarly, using the command **`gmake HISTORY`**. The table of contents should be removed manually from the resulting text file.

Since it does not change very often, the generation of the file `src/test/regress/README` is not fully automated. After building the HTML version of the *Administrator's Guide*, convert the resulting files `regress.html` and `regress-platform.html` to text, using Netscape. Then paste the text files together and edit them to taste (e.g., remove the navigation bars, remove the references to other chapters).

DG2.4. Documentation Authoring

SGML and DocBook do not suffer from an oversupply of open-source authoring tools. The most common toolset is the Emacs/XEmacs editor with appropriate editing mode. On some systems (e.g., RedHat Linux) these tools are provided in a typical full installation.

DG2.4.1. Emacs/PSGML

PSGML is the most common and most powerful mode for editing SGML documents. When properly configured, it will allow you to use Emacs to insert tags and check markup consistancy. You could use it for HTML as well. Check the PSGML web site (http://www.lysator.liu.se/projects/about_psgml.html) for downloads, installation instructions, and detailed documentation.

There is one important thing to note with PSGML: its author assumed that your main SGML DTD directory would be `/usr/local/lib/sgml`. If, as in the examples in this chapter, you use `/usr/local/share/sgml`, you have to compensate for this, either by setting SGML_CATALOG_FILES environment variable, or you can customize your PSGML installation (its manual tells you how).

Put the following in your `~/.emacs` environment file (adjusting the path names to be appropriate for your system):

```
; ********** for SGML mode (psgml)

(setq sgml-omittag t)
(setq sgml-shorttag t)
(setq sgml-minimize-attributes nil)
(setq sgml-always-quote-attributes t)
(setq sgml-indent-step 1)
(setq sgml-indent-data t)
(setq sgml-parent-document nil)
(setq sgml-default-dtd-file "./reference.ced")
(setq sgml-exposed-tags nil)
(setq sgml-catalog-files '("/usr/local/share/sgml/catalog"))
(setq sgml-ecat-files nil)
```

```
(autoload 'sgml-mode "psgml" "Major mode to edit SGML files." t )
```

and in the same file add an entry for SGML into the (existing) definition for `auto-mode-alist`:

```
(setq
  auto-mode-alist
  '(("\\.sgml$" . sgml-mode)
   ))
```

Currently, each SGML source file has the following block at the end of the file:

```
<!-- Keep this comment at the end of the file
Local variables:
mode: sgml
sgml-omittag:t
sgml-shorttag:t
sgml-minimize-attributes:nil
sgml-always-quote-attributes:t
sgml-indent-step:1
sgml-indent-data:t
sgml-parent-document:nil
sgml-default-dtd-file:"./reference.ced"
sgml-exposed-tags:nil
sgml-local-catalogs:("/usr/lib/sgml/catalog")
sgml-local-ecat-files:nil
End:
-->
```

This will set up a number of editing mode parameters even if you do not set up your `~/.emacs` file, but it is a bit unfortunate, since if you followed the installation instructions above, then the catalog path will not match your location. Hence you might need to turn off local variables:

```
(setq inhibit-local-variables t)
```

The PostgreSQL distribution includes a parsed DTD definitions file `reference.ced`. You may find that when using PSGML, a comfortable way of working with these separate files of book parts is to insert a proper `DOCTYPE` declaration while you're editing them. If you are working on this source, for instance, it is an appendix chapter, so you would specify the document as an appendix instance of a DocBook document by making the first line look like this:

```
<!doctype appendix PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
```

This means that anything and everything that reads SGML will get it right, and I can verify the document with **nsgmls -s docguide.sgml**. (But you need to take out that line before building the entire documentation set.)

DG2.4.2. Other Emacs modes

GNU Emacs ships with a different SGML mode, which is not quite as powerful as PSGML, but it's less confusing and lighter weight. Also, it offers syntax highlighting (font lock), which can be very helpful.

Norm Walsh offers a major mode specifically for DocBook (http://nwalsh.com/emacs/docbookide/index.html) which also has font-lock and a number of features to reduce typing.