Diplomarbeit

Enhancement of the ANSI SQL Implementation of PostgreSQL

ausgeführt am Institut für Informationssysteme
der Technischen Universität Wien
unter der Anleitung von

O.Univ.Prof.Dr. Georg Gottlob
und
Univ.Ass. Mag. Katrin Seyr
als verantwortlicher Universitätsassistentin

durch

Stefan Simkovics
Paul Petersgasse 36
A - 2384 Breitenfurt

November 29, 1998
Datum

Unterschrift

Abstract

PostgreSQL is an *object-relational* database management system that runs on almost any UNIX based operating system and is distributed as C-source code. It is neither *freeware* nor *public domain* software. It is copyrighted by the University of California but may be used, modified and distributed as long as the licensing terms of the copyright are accepted.

As the name already suggests, PostgreSQL uses an extended subset of the SQL92 standard as the query language. At the time of writing this document the actual version of PostgreSQL was v6.3.2. In this version the implemented part of SQL did not support some important features included in the SQL92 standard. Two of the not supported features were:

- the *having clause*
- the support of the set theoretic operations *intersect* and *except*

It was the author's task to add the support for the two missing features to the existing source code. Before the implementation could be started an intensive study of the *relevant parts of the SQL92 standard* and the implementation of the *existing features* of PostgreSQL had been necessary. This document will not present only the results of the implementation but also the knowledge collected while studying the SQL language and the source code of the already existing features.

Chapter 1 presents an overview on the SQL92 standard. It gives a description of the *relational data model* and the theoretical (mathematical) background of SQL. Next the SQL language itself is described. The most important SQL statements are presented and a lot of examples are included for better understanding. The information given in this chapter has mainly been taken form the books [DATE96], [DATE94] and [ULL88].

Chapter 2 gives a description on how to use PostgreSQL. First it is shown how the *backend* (server) can be started and how a connection from a client to the server can be established. Next some basic database management tasks like creating a database, creating a table etc. are described. Finally some of PostgreSQL's special features like *user defined functions*, *user defined types*, the *rule system* etc. are presented and illustrated using a lot of examples. The information given in chapter 2 has mainly been taken from the PostgreSQL documentation (see [LOCK98]), the PostgreSQL manual pages and was verified by the author throughout various examples which have also been included.

Chapter 3 concentrates on the internal structure of the PostgreSQL *backend*. First the stages that a query has to pass in order to retrieve a result are described using a lot of figures to illustrate the involved data structures. The information given in that part of chapter 3 has been collected while intensively studying the source code of the relevant parts of PostgreSQL. This intensive and detailed examination of the source code had been necessary to be able to add the missing functionality. The knowledge gathered during that period of time has been summarized here in order to make it easier for programmers who are new to PostgreSQL to find their way in.

The following sections cover the author's ideas for the implementation of the two missing SQL features mentioned above and a description of the implementation itself.

Section 3.7 deals with the implementation of the *having logic*. As mentioned earlier the *having logic* is one of the two missing SQL92 features that the author had to implement.

The first parts of the chapter describe how *aggregate functions* are realized in PostgreSQL and after that a description of the enhancements applied to the code of the *planner/optimizer* and the *executor* in order to realize the new functionality is given. The functions and data structures used and added to the source code are also handled here.

Section 3.8 deals with the implementation of the *intersect* and *except* functionality which was the second missing SQL92 feature that had to be added by the author. First a theoretical description of the basic idea is given. The *intersect* and *except* logic is implemented using a *query rewrite* technique (i.e. a query involving an *intersect* and/or *except* operation is *rewritten* to a semantically equivalent form that does not use these *set operations* any more). After presenting the basic idea the changes made to the *parser* and the *rewrite system* are described and the added functions and data structures are presented.

Contents

List of Figures

Chapter 1

SQL

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

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

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

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

The ISO and ANSI committees have been working for many years on the definition of a greatly expanded version of the original standard, referred to informally as "SQL2" or "SQL/92". This version became a ratified standard - "International Standard ISO/IEC 9075:1992, *Database Language SQL*" - in late 1992. "SQL/92" is the version normally meant when people refer to "the SQL standard". A detailed description of "SQL/92" is given in [DATE96]. At the time of writing this document a new standard informally referred to as "SQL3" is under development. It is planned to make SQL a turing-complete language, i.e. all computable queries (e.g. recursive queries) will be possible. This is a very complex task and therefore the completion of the new standard can not be expected before 1999.

1.1 The Relational Data Model

As mentioned before, SQL is a relational language. That means it is based on the "relational data model" first published by E.F. Codd in 1970. We will give a formal description of the relational model in section 1.1.1 *Formal Notion of the Relational Data Model* but first we want to have a look at it from a more intuitive point of view.

A *relational database* is a database that is perceived by its users as a *collection of tables* (and nothing else but tables). A table consists of rows and columns where each row represents a record and each column represents an attribute of the records contained in the table. Figure 1.1 shows an example of a database consisting of three tables:

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

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

Figure 1.1: The suppliers and parts database

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

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

1.1.1 Formal Notion of the Relational Data Model

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

Definition 1.1 The *Cartesian* product of domains $D_1, D_2, \ldots, D_k$ written $D_1 \times D_2 \times \ldots \times D_k$ is the set of all k-tuples $(v_1, v_2, \ldots, v_k)$ such that $v_1 \in D_1, v_2 \in D_2, \ldots, v_k \in D_k$.

For example, when we have $k = 2$, $D_1 = \{0, 1\}$ and $D_2 = \{a, b, c\}$, then $D_1 \times D_2$ is $\{(0, a), (0, b), (0, c), (1, a), (1, b), (1, c)\}$.

Definition 1.2 A Relation is any subset of the Cartesian product of one or more domains: $R \subseteq D_1 \times D_2 \times \ldots \times D_k$

For example $\{(0, a), (0, b), (1, a)\}$ is a relation, it is in fact a subset of $D_1 \times D_2$ mentioned above. The members of a relation are called tuples. Each relation of some Cartesian product $D_1 \times D_2 \times \ldots \times D_k$ is said to have arity k and is therefore a set of k-tuples.

A relation can be viewed as a table (as we already did, remember figure 1.1 *The suppliers and parts database*) where every tuple is represented by a row and every column corresponds to one component of a tuple. Giving names (called attributes) to the columns leads to the definition of a *relation scheme*.

Definition 1.3 A *relation scheme* R is a finite set of attributes $\{A_1, A_2, \ldots, A_k\}$. There is a domain D_i for each attribute $A_i, 1 \leq i \leq k$ where the values of the attributes are taken from. We often write a relation scheme as $R(A_1, A_2, \ldots, A_k)$.

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

Domains vs. Data Types

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

1.2 Operations in the Relational Data Model

In section 1.1.1 we defined the mathematical notion of the relational model. Now we know how the data can be stored using a relational data model but we do not know what to do with all these tables to retrieve something from the database yet. For example somebody could ask for the names of all suppliers that sell the part 'Screw'. Therefore two rather different kinds of notations for expressing operations on relations have been defined:

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

1.2.1 Relational Algebra

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

- SELECT (σ): extracts *tuples* from a relation that satisfy a given restriction. Let R be a table that contains an attribute A. $\sigma_{A=a}(R) = \{t \in R \mid t(A) = a\}$ where t denotes a tuple of R and $t(A)$ denotes the value of attribute A of tuple t.

- PROJECT (π): extracts specified *attributes* (columns) from a relation. Let R be a relation that contains an attribute X. $\pi_X(R) = \{t(X) \mid t \in R\}$, where $t(X)$ denotes the value of attribute X of tuple t.

- PRODUCT ($\times$): builds the Cartesian product of two relations. Let R be a table with arity k_1 and let S be a table with arity k_2. $R \times S$ is the set of all $(k_1 + k_2)$-tuples whose first k_1 components form a tuple in R and whose last k_2 components form a tuple in S.

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

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

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

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

Example 1.1 Let's have a look at the tables that are produced by evaluating the steps necessary for a join.
Let the following two tables be given:

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

First we calculate the Cartesian product $R \times S$ and get:

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

After the selection $\sigma_{R.C=S.C}(R \times S)$ we get:

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

To remove the duplicate column $S.C$ we project it out by the following operation: $\pi_{R.A,R.B,R.C,S.D,S.E}(\sigma_{R.C=S.C}(R \times S))$ and get:

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

- DIVIDE ($\div$): Let R be a table with the attributes A, B, C and D and let S be a table with the attributes C and D. Then we define the division as: $R \div S = \{t \mid \forall t_s \in S\ \exists t_r \in R$ such that $t_r(A,B) = t \ \wedge \ t_r(C,D) = t_s\}$ where $t_r(x,y)$ denotes a tuple of table R that consists only of the components x and y. Note that the tuple t only consists of the components A and B of relation R.

  Example 1.2 Given the following tables

  R

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

  S

C	D
c	d
e	f

  $R \div S$ is derived as

A	B
a	b
e	d

For a more detailed description and definition of the relational algebra refer to [ULL88] or [DATE94].

Example 1.3 Recall that we formulated all those relational operators to be able to retrieve data from the database. Let's return to our example of section 1.2 where someone wanted to know the names of all suppliers that sell the part 'Screw'. This question can be answered using relational algebra by the following operation:

$$\pi_{SUPPLIER.SNAME}(\sigma_{PART.PNAME='Screw'}(SUPPLIER \bowtie SELLS \bowtie PART))$$

We call such an operation a query. If we evaluate the above query against the tables form figure 1.1 *The suppliers and parts database* we will obtain the following result:

SNAME
Smith
Adams

1.2.2 Relational Calculus

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

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

We want to discuss the tuple relational calculus only because it is the one underlying the most relational languages. For a detailed discussion on DRC (and also TRC) see [DATE94] or [ULL88].

Tuple Relational Calculus

The queries used in TRC are of the following form:

$$\{x(A) \mid F(x)\}$$

where x is a tuple variable A is a set of attributes and F is a formula. The resulting relation consists of all tuples $t(A)$ that satisfy $F(t)$.

Example 1.4 If we want to answer the question from example 1.3 using TRC we formulate the following query:

$$\begin{array}{lll} \{x(SNAME) & \mid & x \in SUPPLIER \wedge \\ & & \exists y \in SELLS\ \exists z \in PART \quad (y(SNO) = x(SNO) \wedge \\ & & \qquad\qquad\qquad\qquad\qquad z(PNO) = y(PNO) \wedge \\ & & \qquad\qquad\qquad\qquad\qquad z(PNAME) = 'Screw')\} \end{array}$$

Evaluating the query against the tables from figure 1.1 *The suppliers and parts database* again leads to the same result as in example 1.3.

1.2.3 Relational Algebra vs. Relational Calculus

The relational algebra and the relational calculus have the same *expressive power* i.e. all queries that can be formulated using relational algebra can also be formulated using the relational calculus and vice versa. This was first proved by E. F. Codd in 1972. This proof is based on an algorithm -"Codd's reduction algorithm"- by which an arbitrary expression of the relational calculus can be reduced to a semantically equivalent expression of relational algebra. For a more detailed discussion on that refer to [DATE94] and [ULL88].

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

1.3 The SQL Language

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

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

1.3.1 Select

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

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

Now we will illustrate the complex syntax of the SELECT statement with various examples. The tables used for the examples are defined in figure 1.1 *The suppliers and parts database*.

Simple Selects

Example 1.5 Here are some simple examples using a SELECT statement:

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

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

and get the table:

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

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

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

In this case the result is:

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

Note that the SQL SELECT corresponds to the ”projection” in relational algebra not to the ”selection” (see section 1.2.1 *Relational Algebra*).

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

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

will lead to the result:

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

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

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

and we get:

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

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

Joins

Example 1.6 The following example shows how *joins* are realized in SQL:

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

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

and get the following table as a result:

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

In the FROM clause we introduced an alias name for every relation because there are common named attributes (SNO and PNO) among the relations. Now we can distinguish between the common named attributes by simply prefixing the attribute name with the alias name followed by a dot. The join is calculated in the same way as shown in example 1.1. First the Cartesian product $SUPPLIER \times PART \times SELLS$ is derived. Now only those tuples satisfying the conditions given in the WHERE clause are selected (i.e. the common named attributes have to be equal). Finally we project out all columns but S.SNAME and P.PNAME.

Aggregate Operators

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

Example 1.7 If we want to know the average cost of all parts in table PART we use the following query:

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

The result is:

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

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

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

and get:

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

Aggregation by Groups

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

The partitioning of the tuples into groups is done by using the keywords GROUP BY followed by a list of attributes that define the groups. If we have GROUP BY $A_1, \ldots, A_k$ we partition the relation into groups, such that two tuples are in the same group if and only if they agree on all the attributes $A_1, \ldots, A_k$.

Example 1.8 If we want to know how many parts are sold by every supplier we formulate the query:

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

and get:

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

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

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

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

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

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

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

Having

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

Example 1.9 If we want only those suppliers selling more than one part we use the query:

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

and get:

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

Subqueries

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

Example 1.10 If we want to know all parts having a greater price than the part named 'Screw' we use the query:

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

The result is:

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

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

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

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

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

Union, Intersect, Except

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

Example 1.11 The following query is an example for UNION:

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

gives the result:

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

Here an example for INTERSECT:

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

gives the result:

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

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

Finally an example for EXCEPT:

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

gives the result:

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

1.3.2 Data Definition

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

Create Table

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

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

Example 1.12 To create the tables defined in figure 1.1 the following SQL statements are used:

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

CREATE TABLE PART
             (PNO   INTEGER,
              PNAME VARCHAR(20),
              PRICE DECIMAL(4 , 2));

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

Data Types in SQL

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

- INTEGER: signed fullword binary integer (31 bits precision).
- SMALLINT: signed halfword binary integer (15 bits precision).

- DECIMAL ($p[,q]$): signed packed decimal number of p digits precision with assumed q of them right to the decimal point. $(15 \geq p \geq q \geq 0)$. If q is omitted it is assumed to be 0.
- FLOAT: signed doubleword floating point number.
- CHAR(n): fixed length character string of length n.
- VARCHAR(n): varying length character string of maximum length n.

Create Index

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

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

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

Example 1.13 To create an index named I on attribute SNAME of relation SUPPLIER we use the following statement:

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

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

Create View

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

Views do not have their own, physically separate, distinguishable stored data. Instead, the system stores the *definition* of the view (i.e. the rules about how to access physically stored *base tables* in order to materialize the view) somewhere in the *system catalogs* (see section 1.3.4 *System Catalogs*). For a discussion on different techniques to implement views refer to section 3.4.1 *Techniques To Implement Views*.

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

```
CREATE VIEW <view_name>
AS <select_stmt>
```

where <select_stmt> is a valid select statement as defined in section 1.3.1. Note that the <select_stmt> is not executed when the view is created. It is just stored in the *system catalogs* and is executed whenever a query against the view is made.

Example 1.14 Let the following view definition be given (we use the tables from figure 1.1 *The suppliers and parts database* again):

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

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

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

will return the following table:

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

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

Drop Table, Drop Index, Drop View

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

```
DROP TABLE <table_name>;
```

Example 1.15 To destroy the SUPPLIER table use the following statement:

```
DROP TABLE SUPPLIER;
```

The DROP INDEX command is used to destroy an index:

```
DROP INDEX <index_name>;
```

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

```
DROP VIEW <view_name>;
```

1.3.3 Data Manipulation

Insert Into

Once a table is created (see section 1.3.2), it can be filled with tuples using the command INSERT INTO. The syntax is:

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

Example 1.16 To insert the first tuple into the relation SUPPLIER of figure 1.1 *The suppliers and parts database* we use the following statement:

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

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

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

Update

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

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

Example 1.17 To change the value of attribute PRICE of the part 'Screw' in the relation PART we use:

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

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

Delete

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

```
DELETE FROM <table_name>
WHERE <condition>;
```

Example 1.18 To delete the supplier called 'Smith' of the table SUPPLIER the following statement is used:

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

1.3.4 System Catalogs

In every SQL database system *system catalogs* are used to keep track of which tables, views indexes etc. are defined in the database. These system catalogs can be queried as if they were normal relations. For example there is one catalog used for the definition of views. This catalog stores the query from the view definition. Whenever a query against a view is made, the system first gets the *view-definition-query* out of the catalog and materializes the view before proceeding with the user query (see section 3.4.1 *Techniques To Implement Views* for a more detailed description). For more information about *system catalogs* refer to [DATE96].

1.3.5 Embedded SQL

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

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

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

When we look at the examples throughout section 1.3.1 we realize that the result of the queries is very often a set of tuples. Most host languages are not designed to operate on sets so we need a mechanism to access every single tuple of the set of tuples returned by a SELECT statement. This mechanism can be provided by declaring a *cursor*. After that we can use the FETCH command to retrieve a tuple and set the cursor to the next tuple.

For a detailed discussion on embedded SQL refer to [DATE96], [DATE94] or [ULL88].

Chapter 2

PostgreSQL from the User's Point of View

This chapter contains information that will be useful for people that only want to use PostgreSQL. It gives a listing and description of the available features including a lot of examples. The users interested in the internals of PostgreSQL should read chapter 3 *PostgreSQL from the Programmer's Point of View*.

2.1 A Short History of PostgreSQL

PostgreSQL is an enhancement of the POSTGRES database management system, a next-generation relational DBMS research prototype running on almost any UNIX based operating system. The original POSTGRES code, from which PostgreSQL is derived, was the effort of many graduate students, undergraduate students, and staff programmers working under the direction of Professor Michael Stonebraker at the University of California, Berkeley. Originally POSTGRES implemented its own query language called POSTQUEL.

In 1995 Andrew Yu and Jolly Chen adapted the last official release of POSTGRES (version 4.2) to meet their own requirements and made some major changes to the code. The most important change is the replacement of POSTQUEL by an extended subset of SQL92. The name was changed to Postgres95 and since that time many other people have contributed to the porting, testing, debugging and enhancement of the code. In late 1996 the name was changed again to the new official name PostgreSQL.

2.2 An Overview on the Features of PostgreSQL

As mentioned earlier PostgreSQL is a relational database management system (RDBMS) but in contrast to the most traditional RDBMSs it is designed to provide more flexibility to the user. One example for the improved flexibility is the support for *user defined* or *abstract data types* (ADTs). Another example is the support of user defined SQL functions. (We will discuss these features later in section 2.5 *Some of PostgreSQL's Special Features in Detail*)

Here is a list of the features PostgreSQL provides:

- An extended subset of SQL92 as query language.
- A commandline interface called `psql` using GNU readline.
- A client/server architecture allowing concurrent access to the databases.

- Support for btree, hash or rtree indexes.
- A transaction mechanism based on the two phase commit protocol is used to ensure data integrity throughout concurrent data access.
- Host based, password, crypt, ident (RFC 1413) or Kerberos V4/V5 authentication can be used to ensure authorized data access.
- A huge amount of predefined data types.
- Support for user defined data types.
- Support for user defined SQL functions.
- Support for recovery after a crash.
- A precompiler for embedded SQL in C.
- An ODBC interface.
- A JDBC interface.
- A Tcl/Tk interface.
- A Perl interface.

2.3 Where to Get PostgreSQL

PostgreSQL is available as source distribution (v6.3.2 at the time of writing) from `ftp://ftp.postgresql.org/pub/`. There is also an official homepage for PostgreSQL at `http://www.postgresgl.org/`. There are a lot of hosts mirroring the contents of the above mentioned ones all over the world.

Copyright of PostgreSQL

PostgreSQL is not public domain software. It is copyrighted by the University of California but may be used according to the licensing terms of the the copyright included in every distribution (refer to the file `COPYRIGHT` included in every distribution for more information).

Support for PostgreSQL

There is no official support for PostgreSQL. That means there is no obligation for anybody to provide maintenance, support, updates, enhancements or modifications to the code. The whole PostgreSQL project is maintained through volunteer effort only. However there are many mailing lists which can be subscribed to in case of problems:

Support Mailing Lists:
`announce@postgreSQL.org` for announcements.
`ports@postgreSQL.org` for OS-specific bugs.
`bugs@postgreSQL.org` for other unsolved bugs.
`questions@postgreSQL.org` for general discussion.

Mailing Lists for Developers:
`hackers@postgreSQL.org` for server internals discussion.
`docs@postgreSQL.org` for the documentation project.
`patches@postgreSQL.org` for patches and discussion.
`mirrors@postgreSQL.org` for mirror site announcements.

To subscribe to the mailing list `questions@postgreSQL.org` for example just send an email to `questions-request@postgreSQL.org` with the lines

```
subscribe
end
```

in the body (not the subject line).

2.4 How to use PostgreSQL

Before we can use PostgreSQL we have to get and install it. We won't talk about installing PostgreSQL here because the installation procedure is straight forward and described very detailed in the file `INSTALL` contained in the distribution. We want to concentrate on the basic usage of PostgreSQL after a successful setup has taken place.

2.4.1 Starting The Postmaster

As mentioned earlier PostgreSQL uses a traditional client/server architecture to provide multi user access. The server is represented by a program called `postmaster` which is started only once at each host. This master server process listens at a specified TCP/IP port for incoming connections by a client. For every incoming connection the `postmaster` spawns a new server process (`postgres`) and continues listening for further connections. Every server process spawned in this way handles exactly one connection to one client. The `postgres` server processes communicate with each other using UNIX semaphores and shared memory to ensure data integrity throughout concurrent data access. (For a more detailed description on these architectural concepts see chapter 3 *PostgreSQL from the Programmer's Point of View*.)

To start the master server process use the following command:

```
$ nohup postmaster > server.log 2>&1 &
```

which will start `postmaster` in the background and even if you log out of the system the process remains active. All errors and messages will be logged to the file `server.log`.

Note: The `postmaster` process is usually started by a special *database superuser* called `postgres` which is a normal UNIX user but has more rights concerning PostgreSQL. For security reasons it is strongly recommended not to run the `postmaster` process as the *system super user* `root`.

2.4.2 Creating a New Database

Once the `postmaster` daemon is running we can create a new database using the following command:

```
$ createdb testdb
```

which will create a database called `testdb`. The user executing the command will become the database administrator and will therefore be the only user (except the *database superuser* `postgres`) who can destroy the database again.

Note: To create the database you don't need to know anything about the tables (relations) that will be used within the database. The tables will be defined later using the SQL statements shown in section 2.4.4 *Defining and Populating Tables*.

2.4.3 Connecting To a Database

After having created at least one database we can make our first client connection to the database system to be able to define tables, populate them, retrieve data, update data etc. Note that most database manipulation is done this way (just creating and destroying a database is done by separate commands which are in fact just shell scripts also using `psql`)

The connection to the DBMS is established by the following command:

```
$ psql testdb
```

which will make a connection to a database called `testdb`. `psql` is a command line interface using GNU readline. It can handle a connection to only one database at a time. When the connection is established `psql` presents itself as follows:

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

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

testdb=>
```

Now you can either enter any valid SQL statement terminated by a ';' or use one of the *slash commands*. A list of all available *slash commands* can be obtained by typing `'\?'`.

Here is a list of the most important *slash commands*:

- `\?` lists all available *slash commands* and gives a short description.
- `\q` quits `psql`.
- `\d` lists all tables, views and indexes existing in the current database.
- `\dt` lists only tables.
- `\dT` lists all available data types.
- `\i <filename>` reads and executes the queries contained in `filename`.
- `\l` lists all available databases known to the system.
- `\connect <database>` terminates the current connection and opens a new connection to `database`.
- `\o [<filename>]` sends all query output to `file`.

2.4.4 Defining and Populating Tables

Defining tables and inserting tuples is done by the SQL statements `CREATE TABLE` and `INSERT INTO`. For a detailed description on the syntax of these commands refer to section 1.3.2 *Data Definition*.

Example 2.1 To create and populate the table SUPPLIER used in figure 1.1 *The suppliers and parts database* we could use the following session:

```
$ psql testdb

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

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

testdb=> create table supplier (sno int4,
testdb-> sname varchar(20),
testdb-> city varchar(20));
CREATE
testdb=> insert into supplier (sno, sname, city)
testdb-> values (1,'Smith','London');
INSERT 26187 1
testdb=> insert into supplier (sno, sname, city)
testdb-> values (2,'Jones','Paris');
INSERT 26188 1
testdb=> insert into supplier (sno, sname, city)
testdb-> values (3,'Adams','Vienna');
INSERT 26189 1
testdb=> insert into supplier (sno, sname, city)
testdb-> values (4,'Blake','Rome');
INSERT 26190 1
testdb=>
```

If you first put all the above commands into a file you can easily execute the statements by the *slash command* `\i <file>`.

Note: The data type `int4` is not part of the SQL92 standard. It is a built in PostgreSQL type denoting a four byte signed integer. For information on which data types are available you can use the `\dT` command which will give a list and short description of all datatypes currently known to PostgreSQL.

2.4.5 Retrieving Data From The Database

After having defined and populated the tables in the database `testdb` we are able to retrieve data by formulating queries using `psql`. Every query has to be terminated by a ';'.

Example 2.2 We assume that all the tables form figure 1.1 *The suppliers and parts database* exist in the database `testdb`. If we want to know all parts that are sold in London we use the following session:

```
testdb=> select p.pname
testdb-> from supplier s, sells se, part p
testdb-> where s.sno=se.sno and
testdb->       p.pno=se.pno and
testdb->       s.city='London';
```

```
pname
-----
Screw
Nut
(2 rows)

testdb=>
```

Example 2.3 We use again the tables given in figure 1.1. Now we want to retrieve all suppliers selling no parts at all (to remove them from the suppliers table for example):

```
testdb=> select * from supplier s
testdb-> where not exists
testdb->                 (select sno from sells se
testdb->                  where se.sno = s.sno);
sno|sname|city
---+-----+----
(0 rows)

testdb=>
```

The result relation is empty in our example telling us that every supplier contained in the database sells at least one part. Note that we used a nested subselect to formulate the query.

2.5 Some of PostgreSQL's Special Features in Detail

Traditional relational database management systems (RDMSs) provide only very few datatypes including floating point numbers, integers, character strings, money, and dates. This makes the implementation of many applications very difficult and that's why PostgreSQL offers substantial additional power by incorporating the following additional basic concepts in such a way that users can easily extend the system:

- inheritance
- user defined functions
- user defined types
- rules

Some other features, implemented in most modern RDBMSs provide additional power and flexibility:

- constraints (given in the `create table` command)
- triggers
- transaction integrity

2.5.1 Inheritance

Inheritance is a feature well known from object oriented programming languages such as Smalltalk or C++. PostgreSQL refers to tables as *classes* and the definition of a *class* may inherit the contents of another *class*:

Example 2.4 First we define a table (class) `city`:

```
testdb=> create table city (
testdb-> name varchar(20),
testdb-> population int4,
testdb-> altitude int4);
CREATE
testdb=>
```

Now we define a new table (class) `capital` that inherits all attributes from `city` and adds a new attribute `country` storing the country which it is the capital of.

```
testdb=> create table capital (
testdb-> country varchar(20)
testdb-> ) inherits (city);
CREATE
testdb=>
```

Note: The class `capital` inherits only the attributes of `city` (not the tuples stored in `city`). The new table can be used as if it were defined without using inheritance:

```
testdb=> insert into capital (name, population,
testdb->                      altitude, state)
testdb-> values ('Vienna', 1500000, 200, 'Austria');
INSERT 26191 1
testdb=>
```

Let's assume that the tables `city` and `capital` have been populated in the following way:

```
city      name     | population | altitude
          ---------+------------+---------
          Linz     |     200000 |      270
          Graz     |     250000 |      350
          Villach  |      50000 |      500
          Salzburg |     150000 |      420


capital   name     | population | altitude |  country
          ---------+------------+----------+---------
          Vienna   |    1500000 |      200 |  Austria
```

Standard SQL92 queries against the above tables behave exactly as expected:

```
testdb=> select * from city
testdb-> where altitude > 400;
name     | population | altitude
---------+------------+---------
Villach  |      50000 |      500
Salzburg |     150000 |      420
(2 rows)

testdb=> select * from capital;
name     | population | altitude |  country
---------+------------+----------+---------
Vienna   |    1500000 |      200 |  Austria
(1 row)

testdb=>
```

If we want to know the names of all cities (including capitals) that are located at an altitude over 100 meters the query is:

```
testdb=> select * from city*
testdb-> where altitude > 100;
name     | population | altitude
---------+------------+---------
Linz     |     200000 |      270
Graz     |     250000 |      350
Villach  |      50000 |      500
Salzburg |     150000 |      420
Vienna   |    1500000 |      200
(5 rows)

testdb=>
```

Here the '*' after `city` indicates that the query should be run over `city` and all classes below `city` in the inheritance hierarchy. Many of the commands that we have already discussed (SELECT, UPDATE, DELETE, etc) support this '*' notation.

2.5.2 User Defined Functions

PostgreSQL allows the definition and usage of *user defined functions*. The new defined functions can be used within every query. PostgreSQL provides two types of functions:

- Query Language (SQL) Functions: functions written in SQL.
- Programming Language Functions: functions written in a *compiled* language such as C.

Query Language (SQL) Functions

These functions are defined using SQL. Note that *query language functions* do not extend the *expressive power* of the SQL92 standard. Every *query language function* can be replaced by an appropriate nested query (*subselect*) without changing the semantical meaning of the whole query. However, since PostgreSQL does not allow *subselects* in the *selectlist* at the moment but does allow the usage of *query language functions*, the *expressive power* of PostgreSQL's current SQL implementation is extended.

The definition of *query language functions* is done using the command `create function <function_name>`. Every function can take zero or more arguments. The type of every argument is specified in the list of arguments in the function definition. The type of the function's result is given after the keyword `returns` in the function definition. The types used for the arguments and the return value of the function can either be *base types* (e.g. int4, varchar, …) or *composite types*. (For each class (table) that is created, a corresponding *composite type* is defined. `supplier` and `part` are examples for *composite types* after the tables `supplier` and `part` have been created.)

Example 2.5 This is an example using only *base types*.

Before PostgreSQL was extended to support nested subqueries user defined query language (SQL) functions could be used to simulate them. Consider example 2.3 where we have wanted to know the names of all suppliers that do not sell any part at all. Normally we would formulate the query as we did in example 2.3. Here we want to show a possible way of formulating the query without using a subquery. This is done in two steps. In the first step we define the function `my_exists`. In the second step we formulate a query using the new function.

In the first step we define the new function `my_exists(int4)` which takes an integer as argument:

```
testdb=> create function my_exists(int4) returns int4
testdb-> as 'select count(pno) from sells
testdb->     where sno = $1;' language 'sql';
CREATE
testdb=>
```

Here is the second step which performs the intended retrieve:

```
testdb=> select s.sname from supplier s
testdb-> where my_exists(s.sno) = 0;
sname
-----
(0 rows)

testdb=>
```

Now let's have a look at what is happening here. The function `my_exists(int4)` takes one argument which must be of type integer. Within the function definition this argument can be refered to using the $1 notation (if there were furhter arguments they could be referred to by $2, $3, …). `my_exists(int4)` returns the number of tuples from table `sells` where the attribute `sno` is equal to the given argument $1 (`sno = $1`). The keyword `language 'sql'` tells PostgreSQL that the new function is a query language function.

The query in the second step examines every tuple from table `supplier` and checks if it satifies the given qualification. It does so by taking the supplier id `sno` of every tuple and giving it as an argument to the function `my_exists(int4)`. In other words the function `my_exists(int4)` is called once for every tuple of table `supplier`. The function returns the number of tuples having the given supplier id `sno` contained in table `sells`. A result of zero means that no such tuple is available meaning that the corresponding supplier does not sell a single part. We can see that this query is semantically equivalent to the one given in example 2.3.

Example 2.6 This example shows how to use a *composite type* in a function definition.

Imagine that the price of every part was doubled over night. If you want to look at the part table with the new values you could use the following function which uses the *composite type* `part` for its argument:

```
testdb=> create function new_price(part) returns float
testdb-> as 'select $1.price * 2;' language 'sql';
CREATE
testdb=> select pno, pname, new_price(price) as new_price
testdb-> from part;
pno |  pname   |  new_price
----+---------+-----------
 1  |  Screw   |         20
 2  |  Nut     |         16
 3  |  Bolt    |         30
 4  |  Cam     |         50
(4 rows)

testdb=>
```

Note that this could have been done by a normal query (without using a user defined function) as well but it's an easy to understand example for the usage of functions.

Programming Language Functions

PostgreSQL also supports *user defined functions* written in C. This is a very powerful feature because you can add any function that can be formulated in C. For example PostgreSQL lacks the function `sqrt()` but it can be easily added using a programming language function.

Example 2.7 We show how to realize the user defined function `pg_my_sqrt(int4)`. The implementation can be divided into three steps:

- formulating the new function in C
- compiling and linking it to a shared library
- making the new function known to PostgreSQL

Formulating the New Function in C: We create a new file called `sqrt.c` and add the following lines:

```
#include <postgres.h>
#include <utils/palloc.h>
#include <math.h>

int4 pg_my_sqrt(int4 arg1)
{
  return (ceil(sqrt(arg1)));
}
```

The function `pg_my_sqrt()` takes one argument of type `int4` which is a PostgreSQL type defined in `postgres.h` and returns the integer value next to the square root of the argument. As with *query language functions* (see previous section) the arguments can be of *base* or of *composite type*. Special care must be taken when using *base types* that are larger than four bytes in length. PostgreSQL supports three types of passing a value to the user defined function:

- pass by value, fixed length
- pass by reference, fixed length
- pass by reference, variable length

Only data types that are 1, 2 or 4 bytes in length can be *passed by value*. We just give an example for the usage of *base types* that can be used for *pass by value* here. For information on how to use types that require *pass by reference* or how to use *composite types* refer to [LOCK98].

Compiling and Linking It to a Shared Library: PostgreSQL binds the new function to the runtime system by using a shared library containing the function. Therefore we have to create a shared library out of the objectfile(s) containing the function(s). It depends on the system and the compiler how this can be done. On a Linux ELF system using `gcc` it can be done by using the following commands:

```
$ gcc -I$PGROOT/include -fpic -c  sqrt.c -o sqrt.o
$ gcc -shared sqrt.o -o sqrt.so
```

where `$PGROOT` is the path PostgreSQL was installed to. The important options given to `gcc` here are `-fpic` in the first line which tells `gcc` to produce *position independent code* that can be loaded to any address of the process image and `-shared` in the second line telling `gcc` to produce a shared library. If you got another system where the above described steps do not work you will have to refer to the manual pages of your c-compiler (often `man cc`) and your linker (`man ld`) to see how shared libraries can be built.

Making the New Function Known to PostgreSQL: Now we have to tell PostgreSQL about the new function. We do so by using the `create function` command within `psql` as we did for *query language functions*:

```
testdb=> create function pg_my_sqrt(int4) returns int4
testdb-> as '/<where_ever_you_put_it>/sqrt.so'
testdb-> language 'c';
CREATE
testdb=>
```

From now on the function `pg_my_sqrt(int4)` can be used in every query. Here is a query against table `part` using the new function:

```
testdb=> select pname, price, pg_my_sqrt(price)
testdb-> from part
testdb-> where pg_my_sqrt(price) < 10;
pname     |price|pg_my_sqrt
----------+-----+----------
Screw     | 10  |         4
Nut       |  8  |         3
Bolt      | 15  |         4
Cam       | 25  |         5
(4 rows)

testdb=>
```

2.5.3 User Defined Types

Adding a new data type to PostgreSQL also requires the definition of an *input* and an *output function*. These functions are implemented using the techniques presented in the previous section *Programming Language Functions*. The functions determine how the type appears in strings (for input by the user and output to the user) and how the type is organized in memory. The input function takes a null-delimited character string as its input and returns the internal (in memory) representation of the type. The output function takes the internal representation of the type and returns a null delimited character string. Besides the definition of *input* and *output functions* it is often necessary to enhance operators (e.g. `'+'`) and aggregate functions for the new data type. How this is done is described in section 2.5.4 *Extending Operators* and section 2.5.5 *Extending Aggregates*.

Example 2.8 Suppose we want to define a complex type which represents complex numbers. Therefore we create a new file called `complex.c` with the following contents:

```
#include <postgres.h>
#include <utils/palloc.h>
#include <math.h>

/* Type definition of Complex */
typedef struct Complex {
    double      x;
    double      y;
} Complex;

/* Input function: takes a char string of the from
 * 'x,y' as argument where x and y must be string
 * representations of double numbers. It returns a
 * pointer to an instance of structure Complex that
 * is setup with the given x and y values. */
Complex *
complex_in(char *str)
{
  double x, y;
  Complex *result;

  /* scan the input string and set x and y to the
   * corresponding double numbers */
  if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2) {
    elog(ERROR, "complex_in: error in parsing");
    return NULL;
  }
  /* reserve memory for the Complex data structure
   * Note: we use palloc here because the memory
   * allocated using palloc is freed automatically
   * by PostgreSQL when it is not used any more */
  result = (Complex *)palloc(sizeof(Complex));
  result->x = x;
  result->y = y;
  return (result);
}

/* Output Function */
/* Takes a pointer to an instance of structure Complex
 * as argument and returns a character pointer to a
 * string representation of the given argument */
char *
complex_out(Complex *complex)
{
  char *result;

  if (complex == NULL) return(NULL);
  result = (char *) palloc(60);
  sprintf(result, "(%g,%g)", complex->x, complex->y);
  return(result);
}
```

Note that the functions defined above operate on types that require *pass by reference*. The functions take a pointer to the data as argument and return a pointer to the derived data instead of passing and returning the data itself. That's why we have to reserve memory using `palloc` within the functions. (If we would just define a local variable and return the addresses of these variables the system would fail, because the memory used by local variables is freed when the function defining these variables completes.)

The next step is to compile the C-functions and create the shared library `complex.so`. This is done in the way described in the previous section *Programming Language Functions* and depends on the system you are using. On a Linux ELF system using `gcc` it would look like this:

```
$ gcc -I$PGROOT/include -fpic -c  complex.c -o complex.o
$ gcc -shared -o complex.so complex.o
```

Now we are ready to define the new datatype but before that we have to make the *input* and *output function* known to PostgreSQL:

```
testdb=> create function complex_in(opaque)
testdb-> returns complex
testdb-> as '/<where_ever_you_put_it>/complex.so'
testdb-> language 'c';
NOTICE:  ProcedureCreate: type 'complex' is not
         yet defined
CREATE
testdb=> create function complex_out(opaque)
testdb-> returns opaque
testdb-> as '/<where_ever_you_put_it>/complex.so'
testdb-> language 'c';
CREATE
testdb=> create type complex (
testdb-> internallength = 16,
testdb-> input = complex_in,
testdb-> output = complex_out
testdb-> );
CREATE
testdb=>
```

Note that the argument type given in the definition of `complex_out()` and `complex_in()` - `opaque` - is needed by PostgreSQL to be able to provide an uniform mechanism for the definition of the *input* and *output functions* needed by a new data type. It is not necessary to specify the exact type of the arguments given to the functions. The *input function* is never called explicitly and when it is called implicitly (e.g. by a statement like `insert into`) it is clear that a character string (i.e. a part of the *insert* query) will be passed to it. The *output function* is only called (by an internal mechanism of PostgreSQL) when data of the corresponding user defined type has to be displayed. In this case it is also clear that the input is of the type used for the internal representation (e.g. `complex`). The output is of type character string.

The new type can now be used as if it were another base type:

```
testdb=> create table complex_test
testdb-> (val complex);
CREATE
testdb=> insert into complex_test
testdb-> (val) values ('(1,2)');
INSERT  155872 1
```

```
testdb=> insert into complex_test
testdb-> (val) values ('(3,4)');
INSERT  155873 1
testdb=> insert into complex_test
testdb-> (val) values ('(5,6)');
INSERT  155874 1

testdb=> select * from complex_test;
  val
-----
(1,2)
(3,4)
(5,6)
(3 rows)

testdb=>
```

2.5.4 Extending Operators

So far we are able to define a new type, create tables that use the new type for one (or more) attribute(s) and populate the new tables with data. We are also able to retrieve data from those tables as long as we do not use the new data types within the qualification of the query. If we want to use the new data types in the `where` clause we have to adapt some (or all) of the operators.

Example 2.9 We show how the operator '=' can be adapted for the usage on the `complex` data type defined in section 2.5.3 *User Defined Types*. We need a user defined function `complex_cmp(complex,complex)` that returns `true` if the complex numbers given as arguments are equal and `false` otherwise. This function is defined as described in section 2.5.2 *User Defined Functions*. In our case there are already two functions present for the usage of type `complex` -the *input* and *output function* defined in example 2.8. So we can add the new function `complex_cmp(complex,complex)` by simply appending the following lines to the file `complex.c` given in example 2.8:

```
/* Comparison Function */
/* returns true if arg1 and arg2 are equal */
bool complex_cmp(Complex *arg1, Complex *arg2)
{
  if((arg1->x == arg2->x) &&
     (arg1->y == arg2->y))
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

Now we create the shared library again:

```
$ gcc -I$PGROOT/include -fpic -c  complex.c -o complex.o
$ gcc -shared -o complex.so complex.o
```

Note that all the functions defined in `complex.c(complex_in(), complex_out()` and `complex_cmp()`) are now contained in the shared library `complex.so`.

Now we make the new function known to PostgreSQL and after that we define the new operator '=' for the `complex` type:

```
testdb=> create function complex_cmp(complex,complex)
testdb-> returns complex
testdb-> as '/<where_ever_you_put_it>/complex.so'
testdb-> language 'c';
CREATE
testdb=> create operator = (
testdb-> leftarg = complex,
testdb-> rightarg = complex,
testdb-> procedure = complex_cmp,
testdb-> commutator = =
testdb-> );
CREATE
testdb=>
```

From now on we are able to perform comparisons between complex numbers in a query's qualification (We use the table `complex_test` as defined in example 2.8):

```
testdb=> select * from complex_test
testdb-> where val = '(1,2)';
  val
-----
(1,2)
(1 row)

testdb=> select * from complex_test
testdb-> where val = '(7,8)';
  val
-----
(0 rows)

testdb=>
```

2.5.5 Extending Aggregates

If we want to use aggregate functions on attributes of a user defined type, we have to add aggregate functions designed to work on the new type. Aggregates in PostgreSQL are realized using three functions:

- `sfunc1` (state function one): is called for every tuple of the current group and the appropriate attribute's value of the current tuple is passed to the function. The given argument is used to change the internal state of the function in the way given by the body of the function. For example `sfunc1` of the aggregate function `sum` is called for every tuple of the current group. The value of the attribute the sum is built on is taken from the current tuple and added to the internal sum state of `sfunc1`.

- `sfunc2` is also called for every tuple of the group but it does not use any argument from outside to manipulate its state. It just keeps track of the own internal state. A typical application for `sfunc2` is a counter that is incremented for every tuple of the group that has been processed.

- `finalfunc` is called after all tuples of the current group have been processed. It takes the internal state of `sfunc1` and the state of `sfunc2` as arguments and derives the result of the aggregate function from the two given arguments. For example with the aggregate function `average`, `sfunc1` sums up the attribute values of each tuple in the group, `sfunc2` counts the tuples in the group. `finalfunc` divides the *sum* by the *count* to derive the average.

If we define an aggregate using only `sfunc1` we get an aggregate that computes a running function of the attribute values from each tuple. `sum` is an example for this kind of aggregate. On the other hand, if we create an aggregate function using only `sfunc2` we get an aggregate that is independent of the attribute values from each tuple. `count` is a typical example of this kind of aggregate.

Example 2.10 Here we want to realize the aggregate functions `complex_sum` and `complex_avg` for the user defined type `complex` (see example 2.8).

First we have to create the user defined functions `complex_add` and `complex_scalar_div`. We can append these two functions to the file `complex.c` from example 2.8 again (as we did with `complex_cmp`):

```
/* Add Complex numbers */
Complex *
complex_add(Complex *arg1, Complex *arg2)
{
  Complex *result;

  result = (Complex *)palloc(sizeof(Complex));
  result->x = arg1->x + arg2->x;
  result->y = arg1->y + arg2->y;
  return(result);
}

/* Final function for complex average */
/* Transform arg1 to polar coordinate form
 * R * e^(j*phi) and divide R by arg2.
 * Transform the new result back to cartesian
 * coordinates */
Complex *
complex_scalar_div(Complex *sum, int count)
{
  Complex *result;
  double R, phi;

  result = (Complex *)palloc(sizeof(Complex));

  /* transform to polar coordinates */
  R = hypot(sum->x,sum->y);
  phi = atan(sum->y / sum->x);

  /* divide by the scalar count */
  R = R / count;

  /* transform back to cartesian coordinates */
  result->x = R * cos(phi);
  result->y = R * sin(phi);
  return(result);
}
```

Next we create the shared library `complex.so` again, which will contain all functions defined in the previous examples as well as the new functions `complex_add` and `complex_scalar_div`:

```
$ gcc -I$PGROOT/include -fpic -c  complex.c -o complex.o
$ gcc -shared -o complex.so complex.o
```

Now we have to make the functions needed by the new aggregates known to PostgreSQL. After that we define the two new aggregate functions `complex_sum` and `complex_avg` that make use of the functions `complex_add` and `complex_scalar_div`:

```
testdb=> create function complex_add(complex,complex)
testdb-> returns complex
testdb-> as '/<where_ever_you_put_it>/complex.so'
testdb-> language 'c';
CREATE
testdb=> create function complex_scalar_div(complex,int)
testdb-> returns complex
testdb-> as '/<where_ever_you_put_it>/complex.so'
testdb-> language 'c';
CREATE
testdb=> create aggregate complex_sum (
testdb-> sfunc1 = complex_add,
testdb-> basetype = complex,
testdb-> stype1 = complex,
testdb-> initcond1 = '(0,0)'
testdb-> );
CREATE
testdb=> create aggregate complex_avg (
testdb-> sfunc1 = complex_add,
testdb-> basetype = complex,
testdb-> stype1 = complex,
testdb-> sfunc2 = int4inc,
testdb-> stype2 = int4,
testdb-> finalfunc = complex_scalar_div,
testdb-> initcond1 = '(0,0)',
testdb-> initcond2 = '0'
testdb-> );
CREATE
```

The aggregate function `complex_sum` is defined using only `sfunc1`. `basetype` is the type of the result of the aggregate function. The function `complex_add` is used as `sfunc1` and `stype1` defines the type `sfunc1` will operate on. `initcond1` gives the initial value of the internal state of `sfunc1`.

If we look at the definition of the aggregate function `complex_avg` we can see that the part concerning `sfunc1` is identical to the corresponding part of the definition of `complex_sum`. The only difference is the additional definition of `sfunc2` and `finalfunc`. The built in function `int4inc` is used as `sfunc2` and increments the internal state of `sfunc2` for every tuple processed. After all tuples have been processed `complex_scalar_div` is used as `finalfunc` to create the average.

From now on we can use the new aggregate functions:

```
testdb=> select * from complex_test;
  val
-----
(1,2)
(3,4)
(5,6)
(3 rows)

testdb=> select complex_sum(val) from complex_test;
   val
------
(9,12)
(1 row)

testdb=> select complex_avg(val) from complex_test;
  val
-----
(3,4)
(1 row)

testdb=>
```

2.5.6 Triggers

PostgreSQL supports the calling of C functions as trigger actions. Triggers are not a feature that is only present in PostgreSQL. In fact most modern RDMSs support triggers. We describe them here because the previous sections are necessary to understand the implementation chosen.

At the moment it is possible to define trigger actions that are executed *before* and *after* the SQL commands `insert`, `update` or `delete` for a tuple. Triggers can be used to ensure data integrity. For example we can define a trigger action that returns an error whenever somebody wants to insert (or update) a tuple with a negative supplier id `sno` into table `supplier` defined in figure 1.1.

The system stores information about when a trigger action has to be performed. Whenever a command triggering an action is detected, the `trigger manager` is called within PostgreSQL, which initializes a global data structure `TriggerData *CurrentTriggerData` and calls the appropriate trigger function.

A central role in the definition of trigger functions plays the global data structure `TriggerData *CurrentTriggerData` (The global pointer `CurrentTriggerData` can be accessed from within every trigger function):

```
typedef struct TriggerData
{
  TriggerEvent    tg_event;
  Relation        tg_relation;
  HeapTuple       tg_trigtuple;
  HeapTuple       tg_newtuple;
  Trigger         *tg_trigger;
} TriggerData;
```

Now we give a short description of the structure's contents relevant for the example below. For a detailed description of this and other structures and functions refer to [LOCK98]:

- `tg_event`: Describes the event the function is called for. Contains information about when the function was called (*before* or *after* the command execution) and for which command it was called (`insert, update` or `delete`).
- `tg_relation`: Is a pointer to a structure describing the relation. `tg_relation->rd_att` is of special interest for us because it can be given as an argument to the function `SPI_getbinval()` described later.
- `tg_trigtuple`: Is a pointer to the tuple for which the trigger is fired. If the command is `insert` or `delete` this is the tuple to be returned by the trigger function.
- `tg_newtuple`: If the command is `update` this is a pointer to the new version of tuple and NULL otherwise. This is what has to be returned by the trigger function if the command is `update`.

Example 2.11 We define a trigger function called `trigf()` that is designed to prevent inserting (updating) tuples with a negative supplier id `sno` into table `supplier`.

First we have to define the function `trigf()` using C and therefore we create a new file `trigger.c`. The function definition is done in exactly the same way as for the definition of user defined functions (see section 2.5.2).

Here are the contents of `trigger.c`:

```
#include <executor/spi.h>
#include <commands/trigger.h>

HeapTuple
trigf()
{ TupleDesc       tupdesc;
  HeapTuple       rettuple;
  bool            isnull;
  int             val;

  if (!CurrentTriggerData)
    elog(ERROR, "trigf: triggers are not initialized");

  /* tuple to return to Executor */
  if (TRIGGER_FIRED_BY_UPDATE(CurrentTriggerData->tg_event))
    rettuple = CurrentTriggerData->tg_newtuple;
  else
    rettuple = CurrentTriggerData->tg_trigtuple;

  tupdesc = CurrentTriggerData->tg_relation->rd_att;
  CurrentTriggerData = NULL;

  /* get the value of attribute 1 of the current tuple */
  val = SPI_getbinval(rettuple, tupdesc, 1, &isnull);

  /* if the value is NULL or < 0 return an error */
  if (isnull || val < 0) {
     elog(ERROR,"insert/update: sno must be a value > 0");
  }
  return (rettuple);
}
```

The function `SPI_getbinval` is part of the *Server Programming Interface (SPI)* described in section 2.5.7. It takes the current tuple, the description of the relation, the number of the attribute and the address of a variable `isnull` as arguments and returns the binary value of the given attribute from the current tuple. If the attribute's value is NULL `isnull` is set to `true` otherwise to `false`.

`trigf()` first checks whether it was called by an `update` command or by an `insert` and sets `rettuple` accordingly. Then it gets the value of the first attribute of the current tuple (remember `sno` is the first attribute in the relation `supplier`). If the value is greater than zero the current tuple is returned (and inserted or updated by the executor of PostgreSQL) otherwise an error is returned (`elog(ERROR,...)` logs an error and aborts processing) and the table `supplier` won't be affected.

To create a shared library out of `trigger.c` we use the commands:

```
$ gcc -I$PGROOT/include -I$PGSRC/include/ -fpic \
> -c trigger.c -o trigger.o
$ gcc -shared -o trigger.so trigger.o
```

Note that for the compilation of trigger functions the source code of PostgreSQL is necessary. `$PGSRC` points to the place where the sources are installed.

Next we have to make the function `trigf()` known to PostgreSQL by the `create function` command in the same way we did for the functions in the previous examples (e.g. example 2.8). After that we can define the trigger `tbefore`. The definition ensures that function `trigf()` given as the trigger action will be executed `before` the execution of the commands `insert` and `update` affecting the table `supplier`. Note that `insert` and `update` commands executed against tables other than `supplier` won't cause the execution of function `trigf()`.

```
testdb=> create function trigf() returns opaque
testdb-> as '/<where_ever_you_put_it/trigger.so'
testdb-> language 'c';
CREATE
testdb=> create trigger tbefore
testdb-> before insert or update on supplier
testdb-> for each row execute procedure trigf();
CREATE
testdb=>
```

Now we can check if the trigger is working correctly. If the value for `sno` is greater than zero the `insert` and `update` commands against table `supplier` are not affected.

```
testdb=> insert into supplier
testdb-> (sno, sname, city)
testdb-> values(5, 'Miles', 'Berlin');
INSERT 156064 1

testdb=> insert into supplier
testdb-> (sno, sname, city)
testdb-> values(-2, 'Huber', 'Munich');
ERROR:  insert/update: sno must be a value > 0

testdb=> update supplier
testdb-> set sno = -2
testdb-> where sname = 'Adams';
ERROR:  insert/update: sno must be a value > 0
testdb=>
```

2.5.7 Server Programming Interface (SPI)

The *Server Programming Interface (SPI)* gives the user the ability to run SQL queries from inside user defined functions. SPI is just a set of native interface functions to simplify the access to the `query parser, planner, optimizer` and `executor` of PostgreSQL (refer to chapter 3 *PostgreSQL from the Programmer's Point of View*).

The set of functions of SPI can be divided into the following parts:

- Interface Functions: These functions are used to establish a connection to a running backend. Whenever you want to execute a SQL query within a user defined function you will have to connect to a backend by `SPI_connect()`.
  - `SPI_connect()` opens a connection to the PostgreSQL backend.
  - `SPI_finish()` closes a connection to the PostgreSQL backend.
  - `SPI_exec()` takes a character string containing a SQL query and a number `tcount` as arguments and executes the query. The result can be obtained from the global data structure `SPI_tuptable` which is set by `SPI_exec()`. If `tcount` is zero then the query is executed for all tuples returned by the query scan. Using `tcount` > 0 restricts the number of tuples for which the query will be executed. This function should only be called after `SPI_connect()` has been processed and a connection has been established.
  - `SPI_prepare()` creates and returns an execution plan (parser+planner+optimizer) but doesn't execute the query. (The function performs the same steps as `SPI_exec()` except that it does not execute the plan. Should only be called after a connection has been established.
  - `SPI_saveplan()` stores a plan prepared by `SPI_prepare()` in safe memory protected from freeing by `SPI_finish()`.
  - `SPI_execp()` executes a plan prepared by `SPI_prepare()` or returned by `SPI_saveplan()`.
- Interface Support Functions: These functions can be used from within a connected or within an unconnected user defined function. An example for the use from within an unconnected function was given in example 2.11 where `SPI_getbinval()` was used to deliver the value of attribute `sno` from the new (to be inserted) tuple. An example for the use from within a connected function will be given in example 2.12.
  - `SPI_copytuple()` makes a copy of the tuple given as argument.
  - `SPI_modifytuple()` modifies one or more attributes of a given tuple. The new values for the attributes to be changed are passed to `SPI_modifytuple()` as arguments.
  - `SPI_fnumber()` takes the description of a tuple and the *name* of an attribute as arguments and returns the *number* of the attribute.
  - `SPI_fname()` takes the description of a tuple and the *number* of an attribute as arguments and returns the *name* of the attribute.
  - `SPI_getvalue()` takes the description of a tuple, the tuple and the number of an attribute as arguments and returns a *string representation* of the given attribute's value.
  - `SPI_getbinval()` takes the description of a tuple, the tuple and the number of an attribute as arguments and returns the binary value of the given attribute.
  - `SPI_gettype()` returns a copy of the type name for the specified attribute.
  - `SPI_gettypeid()` returns the type OID for the specified attribute.

  - `SPI_getrelname()` returns the name of the specified relation.
  - `SPI_palloc()` allocates memory. In contrast to `malloc()` (normally used) it allocates memory in such a way that it can be freed automatically by `SPI_finish()`.
  - `SPI_repalloc()` reallocates memory that has originally been allocated using `SPI_palloc()`.
  - `SPI_pfree()` frees memory allocated by `SPI_palloc()`.

Example 2.12 We want PostgreSQL to automatically generate a value for the supplier id `sno` whenever a new tuple is inserted into table `supplier`. Therefore we define a *trigger function* `trigf_sno()` that is called before the execution of every `insert` statement. `trigf_sno()` has to perform the following steps:

- establish a connection to the PostgreSQL backend using `SPI_connect()`
- get the greatest supplier id `sno` contained in table `supplier` using `SPI_exec()`
- modify attribute `sno` of the tuple to be inserted to contain the next greater number using `SPI_modifytuple()`
- disconnect from the backend using `SPI_finish()`
- return the modified tuple

Here are the contents of function `trigf_sno()` that can be appended to the file `trigger.c` from example 2.11:

```
HeapTuple
trigf_sno()
{ HeapTuple       rettuple;
  bool            isnull;
  int             ret, max;
  int             atts_to_be_changed[1];
  Datum           new_value[1];
  char            nulls;

  if (!CurrentTriggerData)
    elog(ERROR,
         "trigf: triggers are not initialized");

  /* This is the tuple to be inserted */
  rettuple = CurrentTriggerData->tg_trigtuple;

  /* Connect to backend */
  if ((ret = SPI_connect()) < 0)
    elog(ERROR,
         "trigf_sno: SPI_connect returned %d",ret);

  /* Get greatest sno in relation supplier */
  /* Execute the query */
  ret = SPI_exec("select max(sno) from supplier",0);
  if (ret < 0)
     elog(ERROR,
          "trigf_sno: SPI_exec returned %d",ret);
```

```
    /* extract the number from the result
     * returned by the query. SPI_exec() puts
     * the result into the global structure
     * SPI_tuptable */
    max = SPI_getbinval(SPI_tuptable->vals[0],
                        SPI_tuptable->tupdesc, 1,
                        &isnull);

    /* disconnect from backend */
    SPI_finish();

    /* array containing the numbers of attributes
     * to be changed: sno has attno 1 */
    atts_to_be_changed[0] = 1;

    /* new values for attributes to be changed:
     * the next number for sno is max+1 */
    new_value[0] = (max+1);

    /* modify the tuple to be inserted to contain
     * max+1 as sno */
    rettuple =
      SPI_modifytuple(CurrentTriggerData->tg_relation,
                      rettuple,
                      1,
                      atts_to_be_changed,
                      new_value,
                      &nulls);

    if (rettuple == NULL)
       elog(ERROR,
            "trigf_sno: SPI_modifytuple failed");

    CurrentTriggerData = NULL;
    return (rettuple);
  }
```

We again create the shared library `trigger.so` out of `trigger.c`:

```
  $ gcc -I$PGROOT/include -I$PGSRC/include/ -fpic \
  > -c trigger.c -o trigger.o
  $ gcc -shared -o trigger.so trigger.o
```

Next we have to make the function `trigf_sno()` known to PostgreSQL and after that we can define the trigger `sno_before` which is called before an `insert` to the table `supplier`. The trigger function ensures that the next greater number for `sno` will be used.

```
  testdb=> create function trigf_sno() returns opaque
  testdb-> as '/<where_ever_you_put_it>/trigger.so'
  testdb-> language 'c';
  CREATE
```

```
testdb=> create trigger sno_before
testdb-> before insert on supplier
testdb-> for each row execute procedure trigf_sno();
CREATE
testdb=>
```

Every time a new tuple is inserted to the table `supplier` a new `sno` is assigned automatically regardless of the value given in the `insert` statement:

```
testdb=> select * from supplier;
sno|sname |city
---+------+------
  1|Smith |London
  2|Jones |Paris
  3|Adams |Vienna
  4|Blake |Rome
(4 rows)

testdb=> insert into supplier
testdb-> (sno, sname, city)
testdb-> values (200,'Cook', 'Boston');
INSERT 15606 1

testdb=> select * from supplier;
sno|sname |city
---+------+------
  1|Smith |London
  2|Jones |Paris
  3|Adams |Vienna
  4|Blake |Rome
  5|Cook  |Boston
(5 rows)

testdb=>
```

2.5.8 Rules in PostgreSQL

PostgreSQL supports a powerful *rule system*. The user can define a rule and connect it to an event. Whenever the event occurs the rule body is executed in addition to or instead of the commands of the event.

A rule is created using the following SQL statement:

```
create rule rule_name
as on event
to object [where clause]
do [instead]
[action | nothing | [actions...]]
```

where `event` is one of `select, update, delete` or `insert. object` is the name of a *table* or *table.column*. The `where` clause, and the `action` are a normal SQL `where` clause and collection of SQL commands.

One application of rules is the implementation of *views* in PostgreSQL. A *view* is a *virtual* table that does not physically exist within the database but looks to the user as if it did. To realize a *view* we can create an empty table with the name of the *view*. Then we define a rule that is fired every time the new table is accessed. Instead of retrieving the data from the *virtual* table the rule body is executed retrieving the data from one or more physically stored tables.

Example 2.13 We want to show how a view could be realized in PostgreSQL. Note that there is of course an own command to create *views* `create view` in PostgreSQL which performs the steps of our example internally.

First we create a new, empty table `my_view`:

```
testdb=> create table my_view (
testdb-> sname varchar(20),
testdb-> pname varchar(20)
testdb-> );
```

Next we create the rule that will be fired whenever a `select` against the table `my_view` shows up:

```
testdb=> create rule my_view_rule
testdb-> as on select to my_view
testdb-> do instead select s.sname, p.pname
testdb-> from supplier s, part p, sells se
testdb-> where s.sno=se.sno and p.pno=se.pno;
CREATE
testdb=>
```

Now we can use the table `my_view` as if it were populated with tuples:

```
testdb=> select * from my_view;
sname |pname
------+-----
Smith |Screw
Smith |Nut
Jones |Cam
Adams |Screw
Adams |Bolt
Blake |Nut
Blake |Bolt
Blake |Cam
(8 rows)

testdb=> select * from my_view
testdb-> where sname = 'Blake';
sname |pname
------+-----
Blake |Nut
Blake |Bolt
Blake |Cam
(3 rows)

testdb=>
```

Chapter 3

PostgreSQL from the Programmer's Point of View

This chapter gives an overview of the internal structure of the backend of PostgreSQL. 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 PostgreSQL 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.

3.1 The Way of a Query

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

- First a connection from an application program to the PostgreSQL server has to be established. The application program transmits a query to the server and receives the results sent back by the server.
- The *parser stage* checks the query transmitted by the application program (client) for correct syntax and creates a *query tree*.
- The *rewrite system* takes the *query tree* created by the *parser stage* 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.
- 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.
- 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 PostgreSQL's internal control and data structures.

3.2 How Connections are Established

PostgreSQL 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 3.1 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 `ecpq` (i.e. the postgres embedded C compiler) also use this library.

Once a connection is established the client process can send a query to the *backend*. 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.

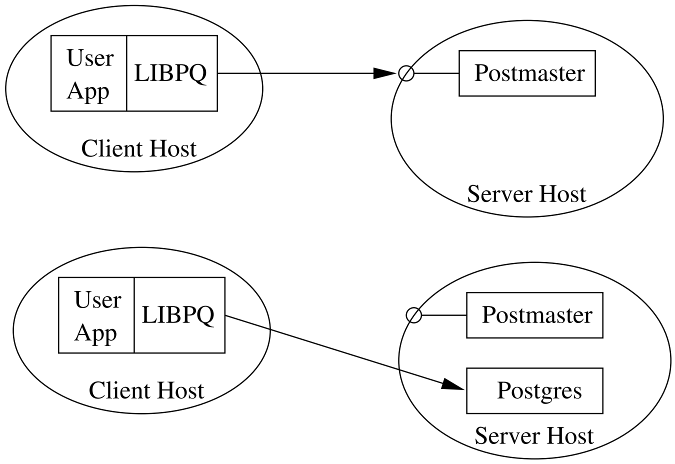

Figure 3.1: How a connection is established

3.3 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 some necessary transformation to the data structure returned by the *parser*.

3.3.1 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 that the mentioned transformations and compilations are normally done automatically using the `makefiles` shipped with the PostgreSQL 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 PostgreSQL for the processing of a query we use an example to illustrate the changes made to these data structures in every stage.

Example 3.1 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 example 1.1 have already been defined.

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

Figure 3.2 shows the *parse tree* built by the *grammar rules* and *actions* given in `gram.y` for the query given in example 3.1 (without the *operator tree* for the *where clause* which is shown in figure 3.3 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 3.3 shows the *operator tree* built for the *where clause* of the SQL query given in example 3.1 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* appearing in the query a `Const` node is created holding the value.

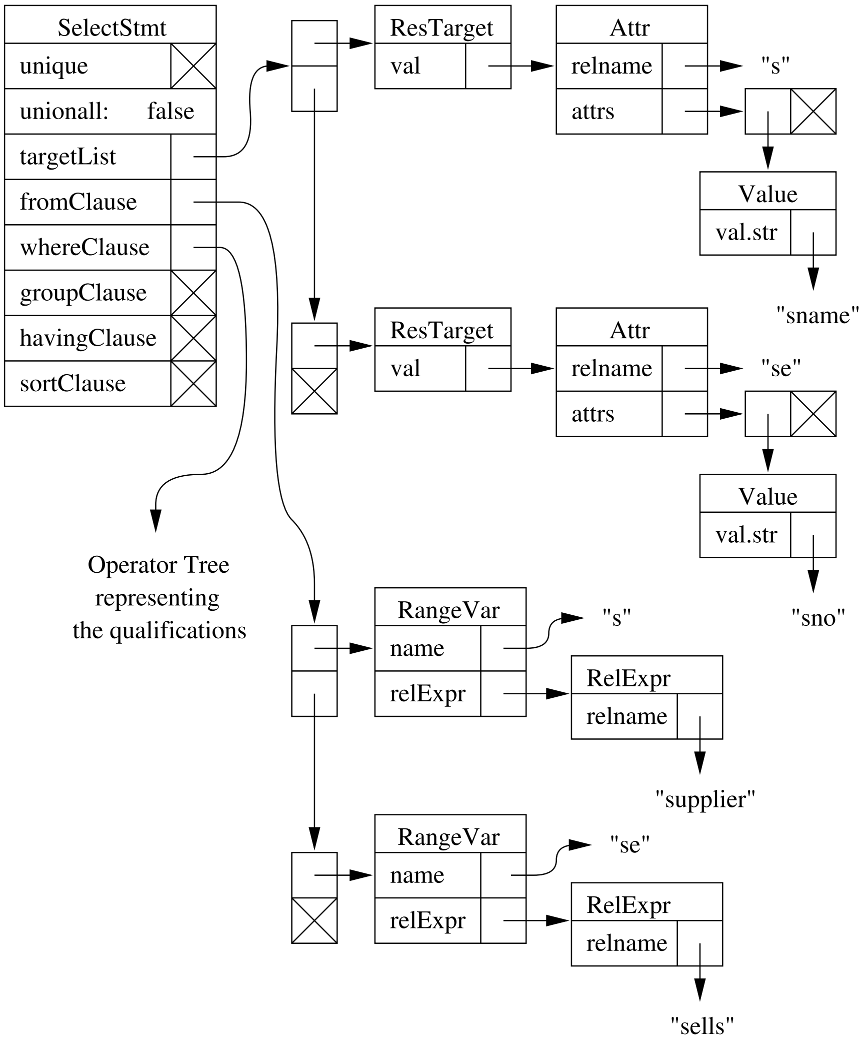

Figure 3.2: *TargetList* and *FromList* for query of example 3.1

3.3.2 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 which will be the top most node of the new data structure. Figure 3.4 shows the transformed data structure (the part for the transformed *where clause* is given in figure 3.5 because there was not enough space to show all parts in one figure).

Now a check is made, if the *relation names* in the *fromClause* 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* which 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.

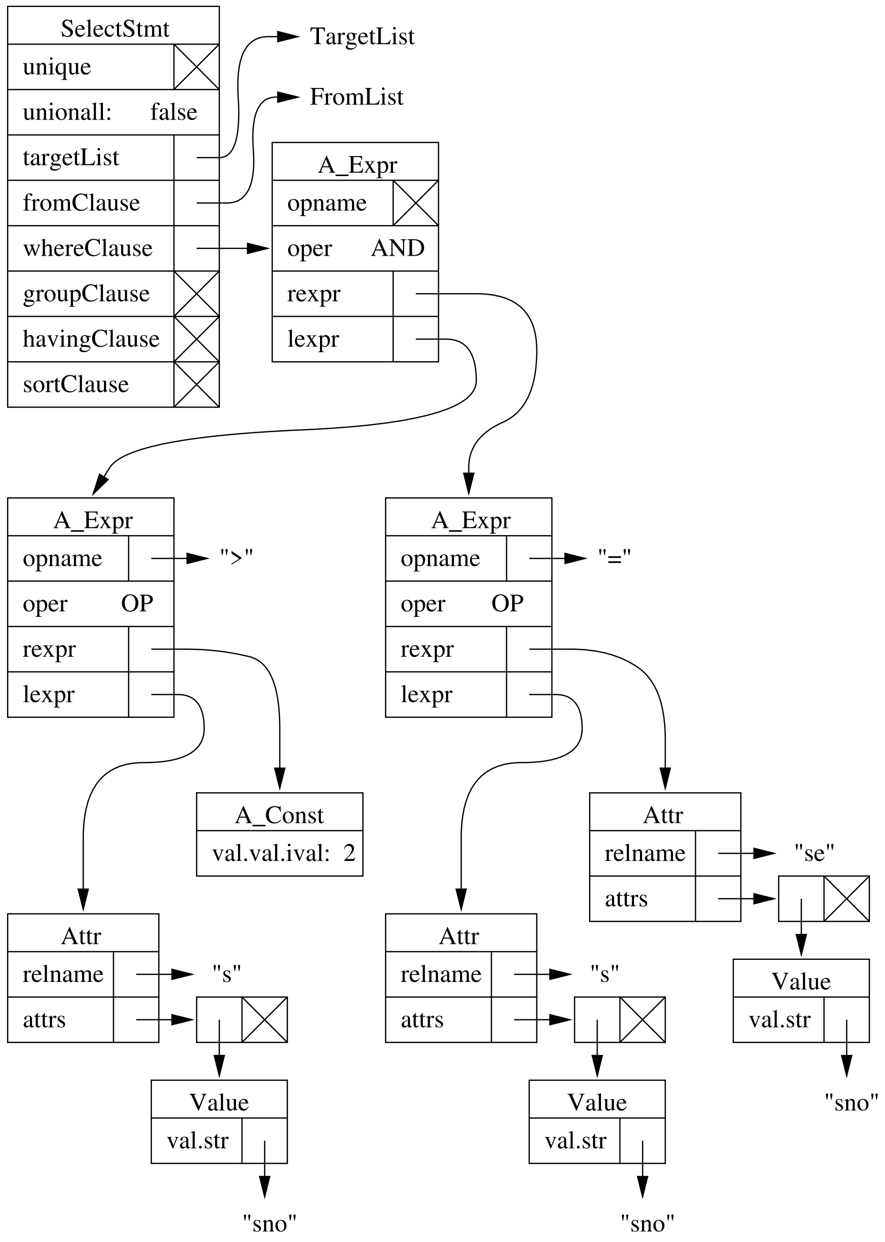

Figure 3.3: *WhereClause* for query of example 3.1

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.

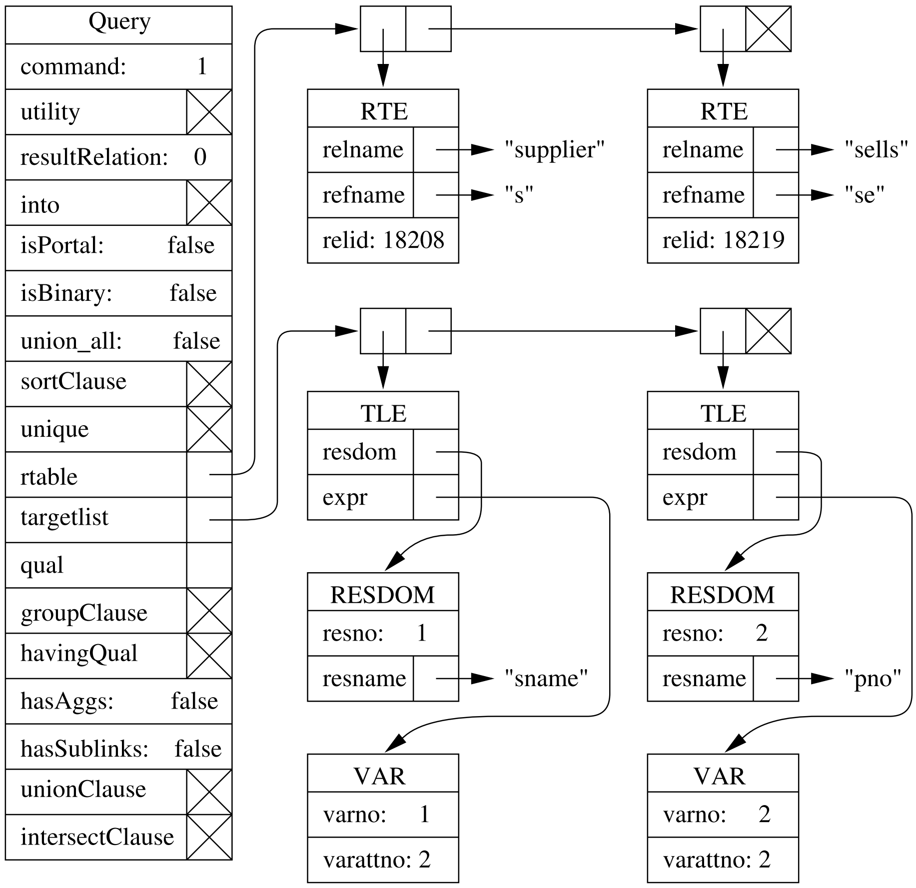

Figure 3.4: Transformed *TargetList* and *FromList* for query of example 3.1

Figure 3.5 shows the transformed *where clause*. Every `A_Expr` node is transformed to an `Expr` node. The `Attr` nodes representing the attributes are replaced by `VAR` nodes as it has been done for the *targetlist* above. Checks if the appearing *attributes* are valid and known to the system are made. If there is an *attribute* used which is not known an error will be returned and the *query processing* will be aborted.

The whole *transformation process* performs a transformation of the data structure handed back by the *parser* to a more comfortable form. The character strings representing the *relations* and *attributes* in the original tree are replaced by *relation ids* and `VAR` nodes whose fields are referring to the entries of the *range table entry list*. In addition to the transformation, also various checks if the used *relation* and *attribute* names (appearing in the query) are valid in the current context are performed.

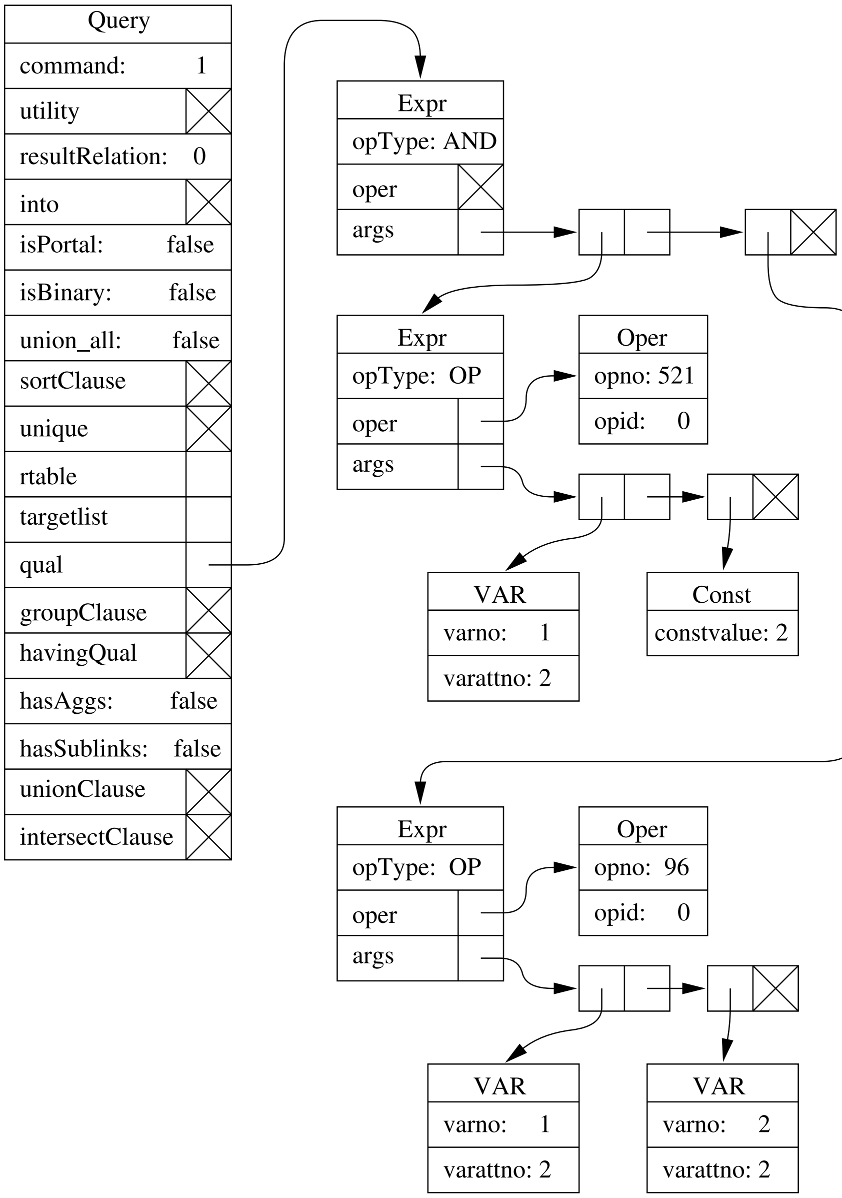

Figure 3.5: Transformed *where clause* for query of example 3.1

3.4 The PostgreSQL Rule System

PostgreSQL supports a powerful *rule system* for the specification of *views* and ambiguous *view updates*. Originally the PostgreSQL *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 has been removed in 1995 when the last official release of the Postgres project was further enhanced to 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 PostgreSQL system refer to section 2.5.8 *Rules in PostgreSQL*.

3.4.1 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.

Techniques To Implement Views

Now 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 [STON89]).

- Take the query given in the *action part* of the *rule*.
- Adapt the *targetlist* to meet the number and order of attributes given in the user-query.

- 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;
```

3.5 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 *optimizer's* task to estimate the cost of executing each path and find out which one is the cheapest.

3.5.1 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 '$\neq$' 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*.

3.5.2 Data Structure of the Plan

Here we will give a little description of the nodes appearing in the *plan*. Figure 3.6 shows the *plan* produced for the query in example 3.1.

The top node of the *plan* is a `MergeJoin` node which 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 3.6 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, PostgreSQL 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.

3.6 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 3.1) 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.

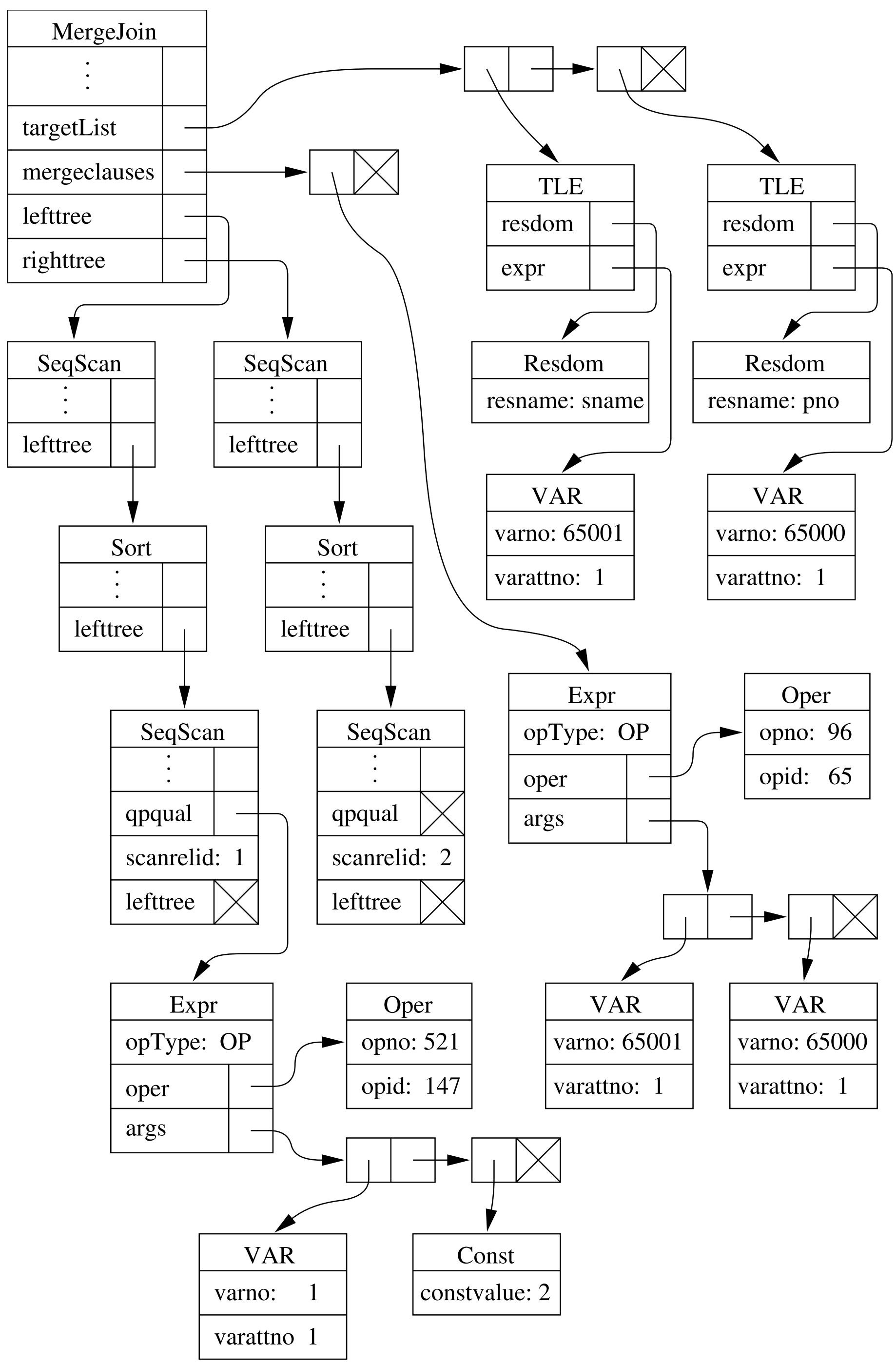

Figure 3.6: *Plan* for query of example 3.1

3.7 The Realization of the Having Clause

The *having clause* has been designed in SQL to be able to use the results of *aggregate functions* within a query qualification. The handling of the *having clause* is very similar to the handling of the *where clause*. Both are formulas in first order logic (FOL) that have to evaluate to true for a certain object to be handed back:

- The formula given in the *where clause* is evaluated for every tuple. If the evaluation returns `true` the tuple is returned, every tuple not satisfying the qualification is ignored.

- In the case of *groups* the *having clause* is evaluated for every group. If the evaluation returns `true` the group is taken into account otherwise it is ignored.

3.7.1 How Aggregate Functions are Implemented

Before we can describe how the *having clause* is implemented we will have a look at the implementation of *aggregate functions* as long as they just appear in the *targetlist*. Note that *aggregate functions* are applied to groups so the query must contain a *group clause*.

Example 3.2 Here is an example of the usage of the *aggregate function* `count` which counts the number of part numbers (`pno`) of every group. (The table `sells` is defined in example 1.1.)

```
select sno, count(pno)
from sells
group by sno;
```

A query like the one in example 3.2 is processed by the usual stages:

- the parser stage

- the rewrite system

- the planner/optimizer

- the executor

and in the following sections we will describe what every stage does to the query in order to obtain the appropriate result.

The Parser Stage

The parser stage builds up a *querytree* containing the *where* qualification and information about the *grouping* that has to be done (i.e. a list of all attributes to group for is attached to the field `groupClause`). The main difference to *querytrees* built up for queries without *aggregate functions* is given in the field `hasAggs` which is set to `true` and in the *targetlist*. The `expr` field of the second `TLE` node of the *targetlist* shown in figure 3.7 does not point directly to a `VAR` node but to an `Aggreg` node representing the *aggregate function* used in the query.

A check is made that every attribute grouped for appears only without an *aggregate function* in the *targetlist* and that every attribute which appears without an *aggregate function* in the *targetlist* is grouped for.

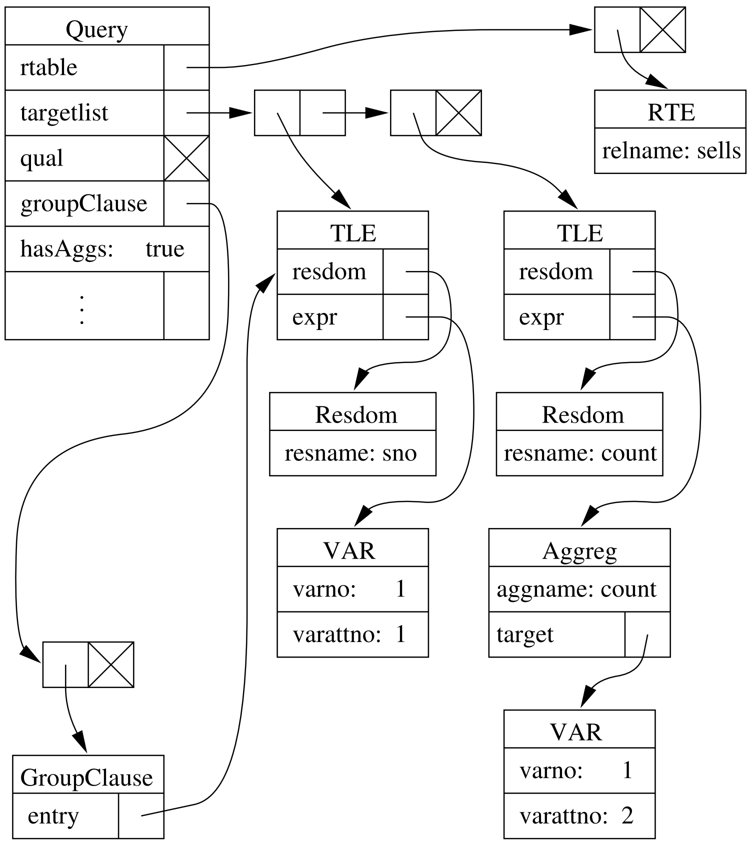

Figure 3.7: *Querytree* built up for the query of example 3.2

The Rewrite System

The rewriting system does not make any changes to the *querytree* as long as the query involves just *base tables*. If any *views* are present the query is rewritten to access the tables specified in the *view definition*.

Planner/Optimizer

Whenever an *aggregate function* is involved in a query (which is indicated by the `hasAggs` flag set to `true`) the planner creates a *plantree* whose top node is an `AGG` node. The *targetlist* is searched for *aggregate functions* and for every function that is found, a pointer to the corresponding `Aggreg` node is added to a list which is finally attached to the field `aggs` of the `AGG` node. This list is needed by the *executor* to know which *aggregate functions* are present and have to be handled.

The `AGG` node is followed by a `GRP` node. The implementation of the *grouping* logic needs a sorted table for its operation so the `GRP` node is followed by a `SORT` node. The `SORT` operation gets its tuples from a kind of `Scan` node (if no indices are present this will be a simple `SeqScan` node). Any qualifications present are attached to the `Scan` node. Figure 3.8 shows the *plan* created for the query given in example 3.2.

Note that every node has its own *targetlist* which may differ from the one of the node above or below. The field `varattno` of every `VAR` node included in a *targetlist* contains a number representing the position of the attribute's value in the tuple of the current node.

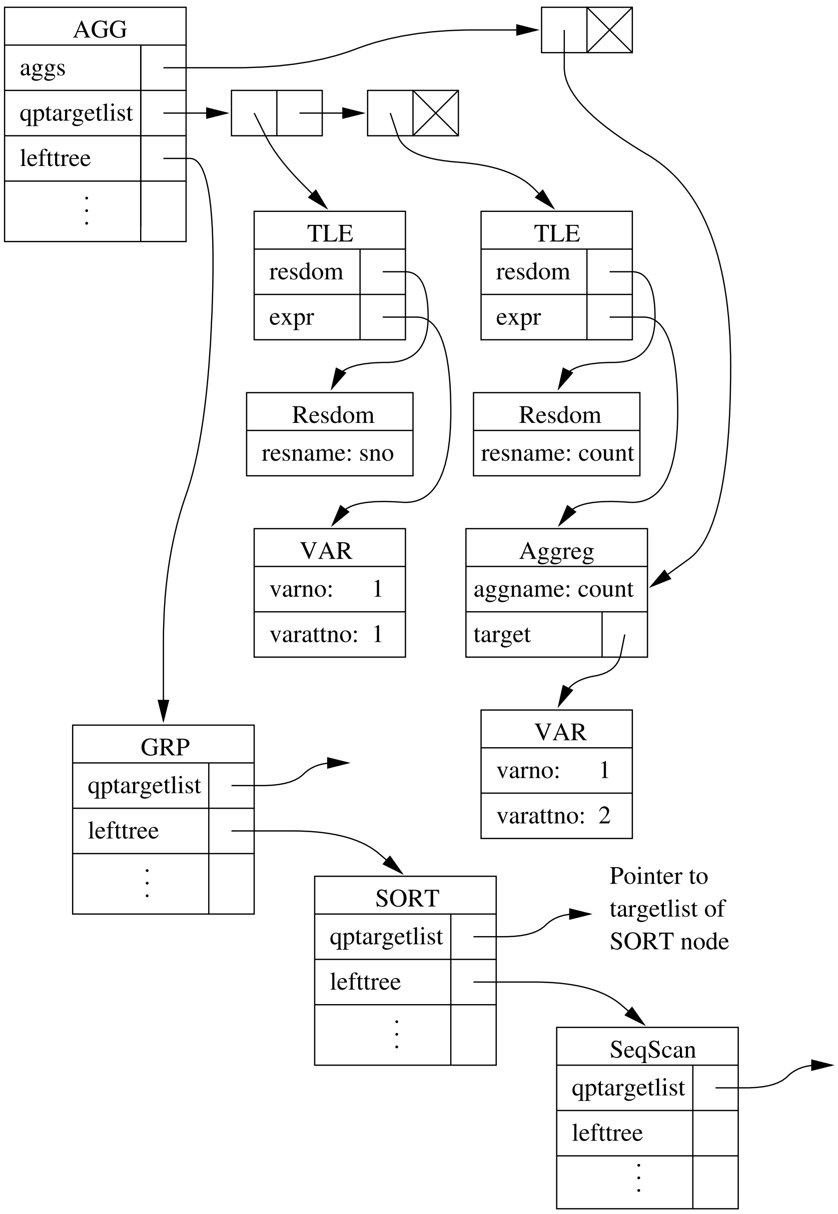

Figure 3.8: *Plantree* for the query of example 3.2

Executor

The *executor* uses the function `execAgg()` to execute `AGG` nodes. As described earlier it uses one main function `ExecProcNode` which is called recursively to execute subtrees. The following steps are performed by `execAgg()`:

- The list attached to the field `aggs` of the `AGG` node is examined and for every *aggregate function* included the *transition functions* are fetched from a *function table*. Calculating the value of an *aggregate function* is done using three functions:
  - The *first transition function* `xfn1` is called with the current value of the attribute the *aggregate function* is applied to and changes its internal state using the attribute's value given as an argument.
  - The *second transition function* `xfn2` is called without any arguments and changes its internal state only according to internal rules.
  - The *final function* `finalfn` takes the final states of `xfn1` and `xfn2` as arguments and finishes the *aggregation*.

  Example 3.3 Recall the functions necessary to implement the *aggregate function* `avg` building the average over all values of an attribute in a group (see section 2.5.5 *Extending Aggregates*):

  - The first transition function `xfn1` has to be a function that takes the actual value of the attribute `avg` is applied to as an argument and adds it to the internally stored sum of previous calls.
  - The second transition function `xfn2` only increases an internal counter every time it is called.
  - The final function `finalfn` divides the result of `xfn1` by the counter of `xfn2` to calculate the average.

  Note that `xfn2` and `finalfn` may be absent (e.g. for the *aggregate function* `sum` which simply sums up all values of the given attribute within a group.

  `execAgg()` creates an array containing one entry for every *aggregate function* found in the list attached to the field `aggs`. The array will hold information needed for the execution of every *aggregate function* (including the *transition functions* described above).

- The following steps are executed in a loop as long as there are still tuples returned by the subplan (i.e. as long as there are still tuples left in the current group). When there are no tuples left in the group a `NULL` pointer is returned indicating the end of the group.
  - A new tuple from the subplan (i.e. the *plan* attached to the field `lefttree`) is fetched by recursively calling `ExecProcNode()` with the subplan as argument.
  - For every *aggregate function* (contained in the array created before) apply the transition functions `xfn1` and `xfn2` to the values of the appropriate attributes of the current tuple.

- When we get here, all tuples of the current group have been processed and the *transition functions* of all *aggregate functions* have been applied to the values of the attributes. We are now ready to complete the *aggregation* by applying the *final function* (`finalfn`) for every *aggregate function*.

- Store the tuple containing the new values (the results of the *aggregate functions*) and hand it back.

Note that the procedure described above only returns one tuple (i.e. it processes just one group and when the end of the group is detected it processes the *aggregate functions* and hands back one tuple). To retrieve all tuples (i.e. to process all groups) the function `execAgg()` has to be called (returning a new tuple every time) until it returns a `NULL` pointer indicating that there are no groups left to process.

3.7.2 How the Having Clause is Implemented

The basic idea of the implementation is to attach the *operator tree* built for the *having clause* to the field `qpqual` of node `AGG` (which is the top node of the query tree). Now the executor has to evaluate the new *operator tree* attached to `qpqual` for every group processed. If the evaluation returns `true` the group is taken into account otherwise it is ignored and the next group will be examined.

In order to implement the *having clause* a variety of changes have been made to the following stages:

- The *parser stage* has been modified slightly to build up and transform an *operator tree* for the *having clause*.
- The *rewrite system* has been adapted to be able to use *views* with the *having logic*.
- The *planner/optimizer* now takes the *operator tree* of the *having clause* and attaches it to the `AGG` node (which is the top node of the *queryplan*).
- The *executor* has been modified to evaluate the *operator tree* (i.e. the internal representation of the *having qualification*) attached to the `AGG` node and the results of the *aggregation* are only considered if the evaluation returns `true`.

In the following sections we will describe the changes made to every single stage in detail.

The Parser Stage

The grammar rules of the *parser* defined in `gram.y` did not require any changes (i.e. the rules had already been prepared for the *having clause*). The *operator tree* built up for the *having clause* is attached to the field `havingClause` of the `SelectStmt` node handed back by the *parser*.

The *transformation* procedures applied to the tree handed back by the *parser* transform the *operator tree* attached to the field `havingClause` using exactly the same functions used for the transformation of the *operator tree* for the *where clause*. This is possible because both trees are built up by the same grammar rules of the *parser* and are therefore compatible. Additional checks which make sure that the *having clause* involves at least one *aggregate function* etc. are performed at a later point in time in the *planner/optimizer* stage.

The necessary changes have been applied to the following functions included in the file `.../src/backend/parser/analyze.c`. Note, that only the relevant parts of the affected code are presented instead of the whole functions. Every added source line will be marked by a `'+'` at the beginning of the line and every changed source line will be marked by a `'!'` throughout the following code listings. Whenever a part of the code which is not relevant at the moment is skipped, three vertical dots are inserted instead.

- `transformInsertStmt()`
  This function becomes is invoked every time a SQL `insert` statement involving a `select` is used like the following example illustrates:

```
insert into t2
select x, y
from t1;
```

  Two statements have been added to this function. The first one performs the transformation of the *operator tree* attached to the field `havingClause` using the function `transformWhereClause()` as done for the *where clause*. It is possible to use the same function for both clauses, because they are both built up by the same *grammar rules* given in `gram.y` and are therefore compatible.

  The second statement makes sure, that *aggregate functions* are involved in the query whenever a *having clause* is used, otherwise the query could have been formulated using only a *where clause*.

```
  static Query *
  transformInsertStmt(ParseState *pstate,
                      InsertStmt *stmt)
  {
    /* make a new query tree */
    Query *qry = makeNode(Query);
                            .
                            .
                            .
    /* fix where clause */
    qry->qual = transformWhereClause(pstate,
                                     stmt->whereClause);

+   /* The havingQual has a similar meaning as "qual" in
+    * the where statement. So we can easily use the
+    * code from the "where clause" with some additional
+    * traversals done in .../optimizer/plan/planner.c
+    */
+   qry->havingQual = transformWhereClause(pstate,
+                                    stmt->havingClause);
                            .
                            .
                            .
+   /* If there is a havingQual but there are no
+    * aggregates, then there is something wrong with
+    * the query because having must contain aggregates
+    * in its expressions! Otherwise the query could
+    * have been formulated using the where clause.
+    */
+   if((qry->hasAggs == false) &&
+      (qry->havingQual != NULL))
+   {
+     elog(ERROR,"This is not a valid having query!");
+     return (Query *)NIL;
+   }
    return (Query *) qry;
  }
```

- `transformSelectStmt()`
  Exactly the same statements added to the function `transformInsertStmt()` above have been added here as well.

```
  static Query *
  transformSelectStmt(ParseState *pstate,
                      SelectStmt *stmt)
  {
    Query  *qry = makeNode(Query);

    qry->commandType = CMD_SELECT;
                              .
                              .
                              .
    qry->qual = transformWhereClause(pstate,
                                     stmt->whereClause);

+   /* The havingQual has a similar meaning as "qual" in
+    * the where statement. So we can easily use the
+    * code from the "where clause" with some additional
+    * traversals done in .../optimizer/plan/planner.c
+    */
+   qry->havingQual = transformWhereClause(pstate,
+                                    stmt->havingClause);
                              .
                              .
                              .
+   /* If there is a havingQual but there are no
+    * aggregates, then there is something wrong with
+    * the query because having must contain aggregates
+    * in its expressions! Otherwise the query could
+    * have been formulated using the where clause.
+    */
+   if((qry->hasAggs == false) &&
+      (qry->havingQual != NULL))
+   {
+     elog(ERROR,"This is not a valid having query!");
+     return (Query *)NIL;
+   }
    return (Query *) qry;
  }
```

The Rewrite System

This section describes the changes to the *rewrite system* of PostgreSQL that have been necessary to support the use of *views* within queries using a *having clause* and to support the definition of *views* by queries using a *having clause*.

As described in section 3.4.1 *Techniques To Implement Views* a query rewrite technique is used to implement *views*. There are two cases to be handled within the *rewrite system* as far as the *having clause* is concerned:

- The *view definition* does not contain a *having clause* but the queries evaluated against this view may contain *having clauses*.

- The *view definition* contains a *having clause*. In this case queries evaluated against this view must meet some restrictions as we will describe later.

No having clause in the view definition: First we will look at the changes necessary to support queries using a *having clause* against a *view* defined without a *having clause*.

Let the following view definition be given:

```
create view test_view
as select sno, pno
   from sells
   where sno > 2;
```

and the following query made against `test_view`:

```
select *
from testview
where sno <> 5;
```

The query will be rewritten to:

```
select sno, pno
from sells
where sno > 2 and
      sno <> 5;
```

The query given in the definition of the *view* `test_view` is the *backbone* of the rewritten query. The *targetlist* is taken from the user's query and also the *where qualification* of the user's query is added to the *where qualification* of the new query by using an `AND` operation.

Now consider the following query:

```
select sno, count(pno)
from testview
where sno <> 5
group by sno
having count(pno) > 1;
```

From now on it is no longer sufficient to add just the *where clause* and the *targetlist* of the user's query to the new query. The *group clause* and the *having qualification* also have to be added to the rewritten query:

```
select sno, count(pno)
from sells
where sno > 2 and
      sno <> 5
group by sno
having count(pno) > 1;
```

Several changes that have already been applied to the *targetlist* and the *where clause* also have to be applied to the *having clause*. Here is a collection of all *additional* steps that have to be performed in order to rewrite a query using a *having clause* against a simple *view* (i.e. a *view* whose definition does not use any *group* and *having clauses*):

- Rewrite the subselects contained in the *having clause* if any are present.
- Adapt the `varno` and `varattno` fields of all `VAR` nodes contained in the *operator tree* representing the *having clause* in the same way as it has been done for the tree representing the *where clause*. The `varno` fields are changed to use the *base tables* given in the *view definition* (which have been inserted into the *range table entry list* in the meantime) instead of the *virtual tables*. The positions of the attributes used in the *view* may differ from the positions of the corresponding attributes in the *base tables*. That's why the `varattno` fields also have to be adapted.
- Adapt the `varno` and `varattno` fields of all `VAR` nodes contained in the `groupClause` of the user's query in the way and for the reasons described above.
- Attach the tree representing the *having qualification* (which is currently attached to the field `havingClause` of the `Query` node for the user's query) to the field `havingClause` of the `Query` node for the new (rewritten) query.
- Attach the list representing the *group clause* (currently attached to the field `groupClause` of the `Query` node for the user's query) to the field *group clause* of the node for the new (rewritten) query.

The view definition contains a having clause: Now we will look at the problems that can arise using *views* that are defined using a query involving a *having clause*.

Let the following *view definition* be given:

```
create view test_view
as select sno, count(pno) as number
   from sells
   where sno > 2
   group by sno
   having count(pno) > 1;
```

Simple queries against this *view* will not cause any troubles:

```
select *
from test_view
where sno <> 5;
```

This query can easily be rewritten by adding the *where qualification* of the user's query (`sno <> 5`) to the *where qualification* of the *view definition's* query.

The next query is also simple but it will cause troubles when it is evaluated against the above given *view definition*:

```
select *
from test_view
where number > 1; /* count(pno) in the view def.
                   * is called  number here */
```

The currently implemented techniques for query rewriting will rewrite the query to:

```
select *
from sells
where sno > 2 and
      count(pno) > 1
group by sno
having count(pno) > 1;
```

which is an invalid query because an *aggregate function* appears in the *where clause*.

Also the next query will cause troubles:

```
select pno, count(sno)
from test_view
group by pno;
```

As you can see this query does neither involve a *where clause* nor a *having clause* but it contains a *group clause* which groups by the attribute `pno`. The query in the definition of the *view* also contains a *group clause* that groups by the attribute `sno`. The two *group clauses* are in conflict with each other and therefore the query cannot be rewritten to a form that would make sense.

Note: There is no solution to the above mentioned problems at the moment and it does not make sense to put much effort into that because the implementation of the support for queries like:

```
select pno_count, count(sno)
from ( select sno, count(pno) as pno_count
       from sells
       where sno > 2
       group by sno
       having count(pno) > 1)
group by pno_count;
```

(which is part of the SQL92 standard) will automatically also solve these problems.

In the next part of the current section we will present the changes applied to the source code in order to realize the above described items. Note that it is not necessary to understand the meaning of every single source line here and therefore we will not discuss detailed questions like "Why has the variable `varno` to be increased by 3?". Questions like that belong to a chapter dealing with the implementation of *views* in PostgreSQL and to be able to answer them it would be necessary to know all the functions and not only those described here. The fact important for us is to make sure, that whatever is applied to the *targetlist* and the data structures representing the *where clause* is also applied to the data structures for the *having clause*. There are three files affected:

```
.../src/backend/rewrite/rewriteHandler.c
.../src/backend/rewrite/rewriteManip.c
.../src/backend/commands/view.c
```

Here is a description of the changes made to the functions contained in the file `.../src/backend/rewrite/rewriteHandler.c`:

- `ApplyRetrieveRule()`
  This function becomes invoked whenever a `select` statement against a *view* is recognized and applies the *rewrite rule* stored in the *system catalogs*. The additional source lines given in the listing below make sure that the functions `OffsetVarNodes()` and `ChangeVarNodes()` that are invoked for the *where clause* and the *targetlist* of the query given in the *view definition* are also called for the *having clause* and the *group clause* of the query in the *view definition*. These functions adapt the `varno` and `varattno` fields of the `VAR` nodes involved.

  The additional source lines at the end of `ApplyRetrieveRule()` attach the data structures representing the *having clause* and the *group clause* of the query in the *view definition* to the rewritten *parsetree*. As mentioned earlier, a *view definition* involving a *group clause* will cause troubles whenever a query using a different *group clause* against this *view* is executed. There is no mechanism preventing these troubles included at the moment.

  Note that the functions `OffsetVarNodes()` , `ChangeVarNodes()` and `AddHavingQual()` appearing in `ApplyRetrieveRule()` are described at a later point in time.

```
    static void
    ApplyRetrieveRule(Query *parsetree, RewriteRule *rule,
                      int rt_index, int relation_level,
                      Relation relation, int *modified)
    {
      Query  *rule_action = NULL;
      Node   *rule_qual;
      List   *rtable,
                              .
                              .
                              .
      OffsetVarNodes((Node *) rule_action->targetList,
                     rt_length);
      OffsetVarNodes(rule_qual, rt_length);

+     OffsetVarNodes((Node *) rule_action->groupClause,
+                    rt_length);
+     OffsetVarNodes((Node *) rule_action->havingQual,
+                    rt_length);
                              .
                              .
                              .
      ChangeVarNodes(rule_qual,
                     PRS2_CURRENT_VARNO + rt_length,
                     rt_index, 0);

+     ChangeVarNodes((Node *) rule_action->groupClause,
+                    PRS2_CURRENT_VARNO + rt_length,
+                    rt_index, 0);
+     ChangeVarNodes((Node *) rule_action->havingQual,
+                    PRS2_CURRENT_VARNO + rt_length,
+                    rt_index, 0);
                              .
                              .
                              .
```

```
      if (*modified && !badsql)
      {
        AddQual(parsetree, rule_action->qual);
+       /* This will only work if the query made to the
+        * view defined by the following groupClause
+        * groups by the same attributes or does not use
+        * groups at all!
+        */
+        if (parsetree->groupClause == NULL)
+           parsetree->groupClause =
+                     rule_action->groupClause;
+        AddHavingQual(parsetree,
+                     rule_action->havingQual);
+        parsetree->hasAggs =
+           (rule_action->hasAggs || parsetree->hasAggs);
+        parsetree->hasSubLinks =
+           (rule_action->hasSubLinks ||
+            parsetree->hasSubLinks);
      }
    }
```

- `QueryRewriteSubLink()`
  This function is called by `QueryRewrite()` to process possibly contained subqueries first. It searches for nested queries by recursively tracing through the *parsetree* given as argument. The additional statement makes sure that the *having clause* is also examined.

```
    static void
    QueryRewriteSubLink(Node *node)
    {
      if (node == NULL)
         return;
      switch (nodeTag(node))
      {
        case T_SubLink:
        {
                              .
                              .
                              .
          QueryRewriteSubLink((Node *) query->qual);
+         QueryRewriteSubLink((Node *)
+                              query->havingQual);
                              .
                              .
                              .
        }
                              .
                              .
                              .
      }
      return;
    }
```

- `QueryRewrite()`
  This function takes the *parsetree* of a query and rewrites it using PostgreSQL's *rewrite system*. Before the query itself can be rewritten, subqueries that are possibly part of the query have to be processed. Therefore the function `QueryRewriteSubLink()` is called for the *where clause* and for the *having clause*.

```
  List *
  QueryRewrite(Query *parsetree)
  {
    QueryRewriteSubLink(parsetree->qual);
+   QueryRewriteSubLink(parsetree->havingQual);
    return QueryRewriteOne(parsetree);
  }
```

Here we present the changes applied to the functions that are contained in the file `.../src/backend/rewrite/rewriteManip.c`:

- `OffsetVarNodes()`
  Recursively steps through the *parsetree* given as the first argument and increments the `varno` and `varnoold` fields of every `VAR` node found by the *offset* given as the second argument. The additional statements are necessary to be able to handle `GroupClause` nodes and `Sublink` nodes that may appear in the *parsetree* from now on.

```
  void
  OffsetVarNodes(Node *node, int offset)
  {
     if (node == NULL)
        return;
     switch (nodeTag(node))
     {
                              .
                              .
                              .
+       /* This has to be done to make queries using
+        * groupclauses work on views
+        */
+        case T_GroupClause:
+        {
+          GroupClause *group = (GroupClause *) node;
+
+          OffsetVarNodes((Node *)(group->entry),
+                         offset);
+        }
+        break;
                              .
                              .
                              .
+        case T_SubLink:
+        {
+          SubLink *sublink = (SubLink *) node;
+          List *tmp_oper, *tmp_lefthand;
+
```

```
+          /* We also have to adapt the variables used
+           * in sublink->lefthand and sublink->oper
+           */
+          OffsetVarNodes((Node *)(sublink->lefthand),
+                         offset);
+
+          /* Make sure the first argument of
+           * sublink->oper points to the same var as
+           * sublink->lefthand does otherwise we will
+           * run into troubles using aggregates (aggno
+           * will not be set correctly)
+           */
+          tmp_lefthand = sublink->lefthand;
+          foreach(tmp_oper, sublink->oper)
+          {
+            lfirst(((Expr *)lfirst(tmp_oper))->args) =
+                                   lfirst(tmp_lefthand);
+            tmp_lefthand = lnext(tmp_lefthand);
+          }
+        }
+        break;
                              .
                              .
                              .
     }
  }
```

- `ChangeVarNodes()`
  This function is similar to the above described function `OffsetVarNodes()` but instead of incrementing the fields `varno` and `varnoold` of *all* `VAR` nodes found, it processes only those `VAR` nodes whose `varno` value matches the parameter `old_varno` given as argument and whose `varlevelsup` value matches the parameter `sublevels_up`. Whenever such a node is found, the `varno` and `varnoold` fields are set to the value given in the parameter `new_varno`. The additional statements are necessary to be able to handle `GroupClause` and `Sublink` nodes.

```
  void
  ChangeVarNodes(Node *node, int old_varno,
                 int new_varno, int sublevels_up)
  {
    if (node == NULL)
       return;
    switch (nodeTag(node))
    {
                              .
                              .
                              .
+     /* This has to be done to make queries using
+      * groupclauses work on views */
+     case T_GroupClause:
+     {
+       GroupClause  *group = (GroupClause *) node;
+
```

```
+       ChangeVarNodes((Node *)(group->entry),
+                      old_varno, new_varno,
+                      sublevels_up);
+     }
+     break;
                              .
                              .
                              .
      case T_Var:
      {
                              .
                              .
                              .
        /* This is a hack: Whenever an attribute
         * from the "outside" query is used within
         * a nested subquery, the varlevelsup will
         * be >0. Nodes having varlevelsup > 0 are
         * forgotten to be processed. The call to
         * OffsetVarNodes() should really be done at
         * another place but this hack makes sure
         * that also those VAR nodes are processed.
         */
+       if (var->varlevelsup > 0)
+          OffsetVarNodes((Node *)var,3);
      }
      break;
                              .
                              .
                              .
      case T_SubLink:
      {
                              .
                              .
                              .
+       ChangeVarNodes((Node *) query->havingQual,
+                      old_varno, new_varno,
+                      sublevels_up);
+       ChangeVarNodes((Node *) query->targetList,
+                      old_varno, new_varno,
+                      sublevels_up);
+
+       /* We also have to adapt the variables used in
+        * sublink->lefthand and sublink->oper
+        */
+       ChangeVarNodes((Node *) (sublink->lefthand),
+                      old_varno, new_varno,
+                      sublevels_up);
      }
      break;
                              .
                              .
                              .
    }
  }
```

- `AddHavingQual()`
  This function adds the *operator tree* given by the parameter `havingQual` to the one attached to the field `havingQual` of the *parsetree* given by the parameter `parsetree`. This is done by adding a new `AND` node and attaching the old and the new *operator tree* as arguments to it. `AddHavingQual()` has not been existing until v6.3.2. It has been created for the *having logic*.

```
void
AddHavingQual(Query *parsetree, Node *havingQual)
{
  Node  *copy, *old;

  if (havingQual == NULL)
     return;

  copy = havingQual;

  old = parsetree->havingQual;
  if (old == NULL)
      parsetree->havingQual = copy;
  else
      parsetree->havingQual =
          (Node *) make_andclause(
                     makeList(parsetree->havingQual,
                              copy, -1));
}
```

- `AddNotHavingQual()`
  This function is similar to the above described function `AddHavingQual()`. It also adds the *operator tree* given by the parameter `havingQual` but prefixes it by a `NOT` node. `AddNotHavingQual()` has also not been existing until v6.3.2 and has been created for the *having logic*.

```
void
AddNotHavingQual(Query *parsetree,
                 Node *havingQual)
{
  Node *copy;

  if (havingQual == NULL)
     return;

  copy = (Node *) make_notclause((Expr *)havingQual);
  AddHavingQual(parsetree, copy);
}
```

- `nodeHandleViewRule()`
  This function is called by `HandleViewRule()`. It replaces all `VAR` nodes of the *user query* evaluated against the *view* (the fields of these `VAR` nodes represent the positions of the attributes in the *virtual* table) by `VAR` nodes that have already been prepared to represent the positions of the corresponding attributes in the *physical* tables (given in the *view definition*). The additional statements make sure that `GroupClause` nodes and `Sublink` nodes are handled correctly.

```
  static void
  nodeHandleViewRule(Node **nodePtr, List *rtable,
                     List *targetlist, int rt_index,
                     int *modified, int sublevels_up)
  {
    Node *node = *nodePtr;
    if (node == NULL)
       return;
    switch (nodeTag(node))
    {
                            .
                            .
                            .
+     /* This has to be done to make queries using
+      * groupclauses work on views
+      */
+     case T_GroupClause:
+     {
+       GroupClause  *group = (GroupClause *) node;
+       nodeHandleViewRule((Node **) (&(group->entry)),
+                          rtable, targetlist, rt_index,
+                          modified, sublevels_up);
+     }
+     break;
                            .
                            .
                            .
      case T_Var:
      {
                            .
                            .
                            .
        if (n == NULL)
        {
          *nodePtr = make_null(((Var *)node)->vartype);
        }
        else
+       /* This is a hack: The varlevelsup of the
+        * original variable and the new one should
+        * be the same. Normally we adapt the node
+        * by changing a pointer to point to a var
+        * contained in 'targetlist'. In the
+        * targetlist all varlevelsups are 0 so if
+        * we want to change it to the original
+        * value we have to copy the node before!
+        * (Maybe this will cause troubles with some
+        * sophisticated queries on views?)
+        */
+       {
+         if(this_varlevelsup>0)
+         {
+            *nodePtr = copyObject(n);
+         }
+         else
```

```
+         {
+           *nodePtr = n;
+         }
+         ((Var *)*nodePtr)->varlevelsup =
+                             this_varlevelsup;
+       }
        *modified = TRUE;
      }
      break;
                             .
                             .
                             .
      case T_SubLink:
      {
                             .
                             .
                             .
+       nodeHandleViewRule(
+                  (Node **) &(query->havingQual),
+                  rtable, targetlist, rt_index,
+                  modified, sublevels_up);
+       nodeHandleViewRule(
+                  (Node **) &(query->targetList),
+                  rtable, targetlist, rt_index,
+                  modified, sublevels_up);
+       /* We also have to adapt the variables used
+        * in sublink->lefthand and sublink->oper
+        */
+       nodeHandleViewRule(
+                  (Node **) &(sublink->lefthand),
+                  rtable, targetlist, rt_index,
+                  modified, sublevels_up);
+       /* Make sure the first argument of
+        * sublink->oper points to the same var as
+        * sublink->lefthand does otherwise we will
+        * run into troubles using aggregates
+        * (aggno will not be set correctly!)
+        */
+       pfree(lfirst(((Expr *)
+                     lfirst(sublink->oper))->args));
+       tmp_lefthand = sublink->lefthand;
+       foreach(tmp_oper, sublink->oper)
+       {
+          lfirst(((Expr *) lfirst(tmp_oper))->args) =
+                                   lfirst(tmp_lefthand);
+          tmp_lefthand = lnext(tmp_lefthand);
+       }
      }
      break;
                             .
                             .
                             .
    }
  }
```

- `HandleViewRule()`
  This function calls `nodeHandleViewRule()` for the *where clause*, the *targetlist*, the *group clause* and the *having clause* of the *user query* evaluated against the given *view*.

```
  void
  HandleViewRule(Query *parsetree, List *rtable,
                 List *targetlist, int rt_index,
                 int *modified)
  {
                             .
                             .
                             .
+   /* The variables in the havingQual and
+    * groupClause also have to be adapted
+    */
+   nodeHandleViewRule(&parsetree->havingQual, rtable,
+                      targetlist, rt_index,
+                      modified, 0);
+   nodeHandleViewRule(
+           (Node **)(&(parsetree->groupClause)),
+           rtable, targetlist, rt_index, modified, 0);
  }
```

The following function is contained in `.../src/backend/commands/view.c`:

- `UpdateRangeTableOfViewParse()`
  This function updates the *range table* of the *parsetree* given by the parameter `viewParse`. The additional statement makes sure that the `VAR` nodes of the *having clause* are modified in the same way as the `VAR` nodes of the *where clause* are.

```
  static void
  UpdateRangeTableOfViewParse(char *viewName,
                              Query *viewParse)
  {
                             .
                             .
                             .
    OffsetVarNodes(viewParse->qual, 2);

+   OffsetVarNodes(viewParse->havingQual, 2);
                             .
                             .
                             .
  }
```

Planner/Optimizer

The *planner* builds a *queryplan* like the one shown in figure 3.8 and in addition to that it takes the *operator tree* attached to the field `havingClause` of the `Query` node and attaches is to the `qpqual` field of the `AGG` node.

Unfortunately this is not the only thing to do. Remember from section 3.7.1 *How Aggregate Functions are Implemented* that the *targetlist* is searched for *aggregate functions* which are appended to a list that will be attached to the field `aggs` of the `AGG` node. This was sufficient as long as *aggregate functions* have only been allowed to appear within the *targetlist*. Now the *having clause* is another source of *aggregate functions*. Consider the following example:

```
select sno, max(pno)
from sells
group by sno
having count(pno) > 1;
```

Here the *aggregate functions* `max` and `count` are in use. If only the *targetlist* is scanned (as it was the case before the *having clause* had been implemented) we will only find and process the *aggregate function* `max`. The second function `count` is not processed and therefore any reference to the result of `count` from within the *having clause* will fail. The solution to this problem is to scan the whole *operator tree* representing the *having clause* for *aggregate functions* not contained in the *targetlist* yet and add them to the list of *aggregate functions* attached to the field `aggs` of the `AGG` node. The scanning is done by the function `check_having_qual_for_aggs()` which steps recursively through the tree.

While scanning the *having clause* for *aggregate functions* not contained in the *targetlist* yet, an additional check is made to make sure that *aggregate functions* are used within the *having clause* (otherwise the query could have been formulated using the *where clause*). Consider the following query which is not a valid SQL92 query:

```
testdb=> select sno, max(pno)
testdb-> from sells
testdb-> group by sno
testdb-> having sno > 1;
ERROR:  This could have been done in a where clause!!
testdb=>
```

There is no need to express this query using a *having clause*, this kind of qualification belongs to the *where clause*:

```
select sno, max(pno)
from sells
where sno > 1
group by sno;
```

There is still an unsolved problem left. Consider the following query where we want to know just the supplier numbers (`sno`) of all suppliers selling more than one part:

```
select sno
from sells
group by sno
having count(pno) > 1;
```

The *planner* creates a *queryplan* (like the one shown in figure 3.8) where the *targetlists* of all nodes involved contain only entries of those attributes listed after the `select` keyword of the query. Looking at the example above this means that the *targetlists* of the `AGG` node, the `GRP` node the `SORT` node and the `SeqScan` node contain only the entry for the attribute `sno`. As described earlier the *aggregation logic* operates on attributes of the tuples returned by the subplan of the `AGG` node (i.e. the result of the `GRP` node). Which attributes are contained in the tuples handed back by a subplan is determined by the *targetlist*. In the case of our example the attribute `pno` needed for the *aggregate function* `count` is not included and therefore the *aggregation* will fail.

The solution to this problem is given in the following steps:

- Make a copy of the actual *targetlist* of the AGG node.
- Search the *operator tree* representing the *having clause* for attributes that are not contained in the *targetlist* of the AGG node yet and add them to the previously made copy.
- The extended *targetlist* is used to create the subplan attached to the `lefttree` field of the AGG node. That means that the *targetlists* of the GRP node, of the SORT node and of the SeqScan node will now contain an entry for the attribute `pno`. The *targetlist* of the AGG node itself will not be changed because we do not want to include the attribute `pno` into the result returned by the whole query.

Care has to be taken that the `varattno` fields of the VAR nodes used in the *targetlists* contain the position of the corresponding attribute in the *targetlist* of the subplan (i.e the subplan delivering the tuples for further processing by the actual node).

The following part deals with the source code of the new and changed functions involved in the planner/optimizer stage. The files affected are:

```
.../src/backend/optimizer/plan/setrefs.c
.../src/backend/optimizer/plan/planner.c
```

Since all of the functions presented here are very long and would need very much space if presented as a whole, we just list the most important parts.

The following two functions are new and have been introduced for the *having logic*. They are contained in the file `.../src/backend/optimizer/plan/setrefs.c`:

- `check_having_qual_for_aggs()`
  This function takes the representation of a *having clause* given by the parameter `clause`, a *targetlist* given by the parameter `subplanTargetList` and a *group clause* given by the parameter `groupClause` as arguments and scans the representation of the *having clause* recursively for *aggregate functions*. If an *aggregate function* is found it is attached to a list (internally called `agg_list`) and finally returned by the function.

  Additionally the `varno` field of every VAR node found is set to the position of the corresponding attribute in the *targetlist* given by `subplanTargetList`.

  If the *having clause* contains a subquery the function also makes sure, that every attribute from the *main query* that is used within the subquery also appears in the *group clause* given by `groupClause`. If the attribute cannot be found in the *group clause* an error message is printed to the screen and the query processing is aborted.

```
List *
check_having_qual_for_aggs(Node *clause,
                           List *subplanTargetList,
                           List *groupClause)
{
  List *t, *l1;
  List *agg_list = NIL;
  int  contained_in_group_clause = 0;

  if (IsA(clause, Var))
  {
```

```
    TargetEntry *subplanVar;

    subplanVar = match_varid((Var *) clause,
                             subplanTargetList);
    /* Change the varattno fields of the
     * var node to point to the resdom->resnofields
     * of the subplan (lefttree)
     */
    ((Var *) clause)->varattno =
      subplanVar->resdom->resno;
    return NIL;
  }
  else
    if (is_funcclause(clause) || not_clause(clause)
        || or_clause(clause) || and_clause(clause))
    {
      int new_length=0, old_length=0;

      /* This is a function. Recursively call this
       * routine for its arguments... (i.e. for AND,
       * OR, ... clauses!)
       */
      foreach(t, ((Expr *) clause)->args)
      {
        old_length=length((List *)agg_list);
        agg_list = nconc(agg_list,
            check_having_qual_for_aggs(lfirst(t),
                                 subplanTargetList,
                                 groupClause));
        if(((new_length=length((List *)agg_list)) ==
            old_length) || (new_length == 0))
        {
          elog(ERROR,"This could have been done
                                 in a where clause!!");
          return NIL;
        }
      }
      return agg_list;
    }
    else
      if (IsA(clause, Aggreg))
      {
        return lcons(clause,
                     check_having_qual_for_aggs(
                         ((Aggreg *)clause)->target,
                         subplanTargetList,
                         groupClause));
      }
      else
                         .
                         .
                         .
}
```

- `check_having_qual_for_vars()`
  This function takes the representation of a *having clause* given by the parameter `clause` and the actual *targetlist* given by the parameter `targetlist_so_far` as arguments and recursively scans the representation of the *having clause* for attributes that are not included in the actual *targetlist* yet. Whenever such an attribute is found it is added to the actual *targetlist* which is finally returned by the function.

  Attributes contained in the *having clause* but not in the *targetlist* show up with queries like:

  ```
  select sid
  from  part
  group by sid
  having min(pid) > 1;
  ```

  In the above query the attribute `pid` is used in the *having clause* but it does not appear in the *targetlist* (i.e. the list of attributes after the keyword `select`). Unfortunately only those attributes are delivered by the subplan and can therefore be used within the *having clause*. To become able to handle queries like that correctly, we have to extend the actual *targetlist* by those attributes used in the `having clause` but not already appearing in the *targetlist*.

  ```
  List *
  check_having_qual_for_vars(Node *clause,
                             List *targetlist_so_far)
  {
    List     *t;

    if (IsA(clause, Var))
    {
      Rel         tmp_rel;

      tmp_rel.targetlist = targetlist_so_far;
      /* Check if the VAR is already contained in the
       * targetlist
       */
      if (tlist_member((Var *)clause,
                 (List *)targetlist_so_far) == NULL)
      {
        add_tl_element(&tmp_rel, (Var *)clause);
      }
      return tmp_rel.targetlist;
    }
    else
      if (is_funcclause(clause) || not_clause(clause)
          || or_clause(clause) || and_clause(clause))
      {
        /* This is a function. Recursively call this
         * routine for its arguments...
         */
        foreach(t, ((Expr *) clause)->args)
        {
          targetlist_so_far =
            check_having_qual_for_vars(lfirst(t),
                                targetlist_so_far);
  ```

```
        }
        return targetlist_so_far;
      }
      else
        if (IsA(clause, Aggreg))
        {
          targetlist_so_far =
            check_having_qual_for_vars(
                            ((Aggreg *)clause)->target,
                            targetlist_so_far);
          return targetlist_so_far;
        }
                            .
                            .
                            .
}
```

The next function is found in `.../src/backend/optimizer/plan/planner.c`:

- `union_planner()`
  This function creates a *plan* from the *parsetree* given to it by the parameter `parse` that can be executed by the *executor*.

  If *aggregate functions* are present (indicated by `parse->hasAggs` set to true) the first step is to extend the *targetlist* by those attributes that are used within the *having clause* (if any is present) but do not appear in the *select list* (Refer to the description of `check_having_qual_for_vars()` above).

  The next step is to call the function `query_planner()` creating a *plan* without taking the *group clause*, the *aggregate functions* and the *having clause* into account for the moment.

  Next insert a `GRP` node at the top of the *plan* according to the *group clause* of the *parsetree* if any is present.

  Add an `AGG` node to the top of the current *plan* if *aggregate functions* are present and if a *having clause* is present additionally perform the following steps:

  – Perform various transformations to the representation of the *having clause* (e.g. transform it to CNF, . . .).
  – Attach the transformed representation of the *having clause* to the field `plan.qual` of the just created `AGG` node.
  – Examine the whole *having clause* and search for *aggregate functions*. This is done using the function `check_having_qual_for_aggs()` which appends every *aggregate function* found to a list that is finally returned.
  – Append the list just created to the list already attached to the field `aggs` of the `AGG` node (this list contains the *aggregate functions* found in the *targetlist*).
  – Make sure that *aggregate functions* do appear in the *having clause*. This is done by comparing the length of the list attached to `aggs` before and after the call to `check_having_qual_for_aggs()`. If the length has not changed, we know that no *aggregate function* has been detected and that this query could have been formulated using only a *where clause*. In this case an error message is printed to the screen and the processing is aborted.

```
  Plan *
  union_planner(Query *parse)
  {
     List       *tlist = parse->targetList;

+    /* copy the original tlist, we will need the
+     * original one for the AGG node later on */
+     List *new_tlist = new_unsorted_tlist(tlist);
                          .
                          .
                          .
+         if (parse->hasAggs)
+         {
+           /* extend targetlist by variables not
+            * contained already but used in the
+            * havingQual.
+            */
+           if (parse->havingQual != NULL)
+             {
+               new_tlist =
+                 check_having_qual_for_vars(
+                              parse->havingQual,
+                              new_tlist);
+             }
+         }
                          .
                          .
                          .
          /* Call the planner for everything
           * but groupclauses and aggregate funcs.
           */
          result_plan = query_planner(parse,
                                 parse->commandType,
                                 new_tlist,
                                 (List *) parse->qual);
                          .
                          .
                          .
        /* If aggregate is present, insert the AGG node
         */
        if (parse->hasAggs)
        {
          int old_length=0, new_length=0;
          /* Create the AGG node but use 'tlist' not
           * 'new_tlist' as target list because we
           * don't want the additional attributes
           * (only used for the havingQual, see
           * above) to show up in the result.
           */
          result_plan = (Plan *) make_agg(tlist,
                                      result_plan);
                          .
                          .
                          .
```

```
+         /* Check every clause of the havingQual for
+          * aggregates used and append them to
+          * the list in result_plan->aggs
+          */
+         foreach(clause,
+                 ((Agg *) result_plan)->plan.qual)
+         {
+           /* Make sure there are aggregates in the
+            * havingQual if so, the list must be
+            * longer after check_having_qual_for_aggs
+            */
+           old_length =
+             length(((Agg *) result_plan)->aggs);
+
+           ((Agg *) result_plan)->aggs =
+              nconc(((Agg *) result_plan)->aggs,
+                    check_having_qual_for_aggs(
+                      (Node *) lfirst(clause),
+                      ((Agg *)result_plan)->
+                          plan.lefttree->targetlist,
+                      ((List *) parse->groupClause)));
+           /* Have a look at the length of the returned
+            * list. If there is no difference, no
+            * aggregates have been found and that means
+            * that the Qual belongs to the where clause
+            */
+           if (((new_length =
+                 length(((Agg *) result_plan)->aggs))==
+                 old_length) || (new_length == 0))
+           {
+             elog(ERROR,"This could have been done in a
+                                    where clause!!");
+             return (Plan *)NIL;
+           }
+         }
                              .
                              .
                              .
    }
```

Executor

The *executor* takes the *queryplan* produced by the *planner/optimizer* in the way just described and processes all *aggregate functions* in the way described in section 3.7.1 *The Implementation of Aggregate Functions* but before the tuple derived is handed back the *operator tree* attached to the field `qpqual` is evaluated by calling the function `ExecQual()`. This function recursively steps through the *operator tree* (i.e. the *having clause*) and evaluates the predicates appearing there. Thanks to our changes that have been made to the *planner* the values of all operands needed to evaluate the predicates (e.g. the values of all *aggregate functions*) are already present and can be accessed throughout the evaluation without any problems.

If the evaluation of the *having qualification* returns `true` the tuple is returned by the function `execAgg()` otherwise it is ignored and the next group is processed.

The necessary changes and enhancements have been applied to the following function in the file `.../src/backend/executor/nodeAgg.c`:

- `execAgg()` Whenever the *executor* gets to an `AGG` node this function is called. Before the *having logic* had been implemented, all the *tuples* of the current group were fetched from the *subplan* and all *aggregate functions* were applied to these tuples. After that, the results were handed back to the calling function.

  Since the *having logic* has been implemented there is one additional step executed. Before the results of applying the *aggregate functions* are handed back, the function `ExecQual()` is called with the representation of the *having clause* as an argument. If `true` is returned, the results are handed back, otherwise they are ignored and we start from the beginning for the next group until a group meeting the restrictions given in the *having clause* is found.

```
  TupleTableSlot *
  ExecAgg(Agg *node)
  {
                              .
                              .
                              .
        /* We loop retrieving groups until we find one
         * matching node->plan.qual
         */
+       do
+       {
                              .
                              .
                              .
          /* Apply *all* aggregate function to the
           * tuples of the *current* group
           */
                              .
                              .
                              .
          econtext->ecxt_scantuple =
                   aggstate->csstate.css_ScanTupleSlot;
          resultSlot = ExecProject(projInfo, &isDone);

+         /* As long as the retrieved group does not
+          * match the qualifications it is ignored and
+          * the next group is fetched
+          */
+         if(node->plan.qual != NULL)
+         {
+           qual_result =
+               ExecQual(fix_opids(node->plan.qual),
+                        econtext);
+         }
+         if (oneTuple) pfree(oneTuple);
+       }
+       while((node->plan.qual!=NULL) &&
+                               (qual_result!=true));
        return resultSlot;
  }
```

3.8 The Realization of Union, Intersect and Except

SQL92 supports the well known set theoretic operations *union*, *intersect* and *set difference* (the *set difference* is called *except* in SQL92). The operators are used to connect two or more `select` statements. Every `select` statement returns a set of tuples and the operators between the `select` statements tell how to merge the returned sets of tuples into one result relation.

Example 3.4 Let the following tables be given:

```
A   C1|C2|C3          B   C1|C2|C3
    --+--+--              --+--+--
     1| a| b               1| a| b
     2| a| b               5| a| b
     3| c| d               3| c| d
     4| e| f               8| e| f

           C   C1|C2|C3
               --+--+--
                4| e| f
                8| e| f
```

Now let's have a look at the results of the following queries:

```
select * from A
union
select * from B;
```

derives the set theoretic *union* of the two tables:

```
C1|C2|C3
--+--+--
 1| a| b
 2| a| b
 3| c| d
 4| e| f
 5| a| b
 8| e| f
```

The `select` statements used may be more complex:

```
select C1, C3 from A
  where C2 = 'a'
union
select C1, C2 from B
  where C3 = 'b';
```

will return the following table:

```
C1|C3
--+--
 1| b
 2| b
 1| a
 5| a
```

Note that the selected columns do not need to have identical names, they only have to be of the same type. In the previous example we selected for `C1` and `C3` in the first `select` statement and for `C1` and `C2` in the second one. The names of the resulting columns are taken from the first `select` statement.

Let's have a look at a query using `intersect`:

```
select * from A
intersect
select * from B;
```

will return:

```
C1|C2|C3
--+--+--
 1| a| b
 3| c| d
```

Here is an example using `except`:

```
select * from A
except
select * from B;
```

will return:

```
C1|C2|C3
--+--+--
 2| a| b
 4| e| f
```

The last examples were rather simple because they only used one set operator at a time with only two operands. Now we look at some more complex queries involving more *operators*:

```
select * from A
union
select * from B
intersect
select * from C;
```

will return:

```
C1|C2|C3
--+--+--
 4| e| f
 8| e| f
```

The above query performs the set theoretic computation $(A \cup B) \cap C$. When no parentheses are used, the operations are considered to be left associative, i.e. $A \cup B \cup C \cup D$ will be treated as $((A \cup B) \cup C) \cup D$.

The same query using parenthesis can lead to a completely different result:

```
select * from A
union
(select * from B
 intersect
 select * from C);
```

performs $A \cup (B \cap C)$ and will return:

```
C1|C2|C3
--+--+--
 1| a| b
 2| a| b
 3| c| d
 4| e| f
 8| e| f
```

3.8.1 How Unions have been Realized Until Version 6.3.2

First we give a description of the implementation of *union* and *union all* until version 6.3.2 because we need it to understand the implementation of *intersect* and *except* described later.

A *union* query is passed through the usual stages:

- parser
- rewrite system
- planner/optimizer
- executor

and we will now describe what every single stage does to the query. For our explanation we assume to process a simple query (i.e. a query without *subselects*, *aggregates* and without involving *views*)

The Parser Stage

As described earlier the *parser stage* can be divided into two parts:

- the *parser* built up by the grammar rules given in `gram.y` and
- the *transformation routines* performing a lot of changes and analysis to the tree built up by the parser. Most of these routines reside in `analyze.c`.

A *union* statement consists of two or more *select* statements connected by the keyword `union` as the following example shows:

```
select * from A
  where C1=1
union
select * from B
  where C2 = 'a'
union
select * from C
  where C3 = 'f'
```

The above *union* statement consists of three *select* statements connected by the keyword `union`. We will refer to the first *select* statement by A, to the second one by B and to the third one by C for our further explanation (in the new notation our query looks like this: `A union B union C`).

The *parser* (given by `gram.y`) processes all three `select` statements, creates a `SelectStmt` node for every `select` and attaches the `where` qualifications, *targetlists* etc. to the corresponding nodes. Then it creates a list of the second and the third

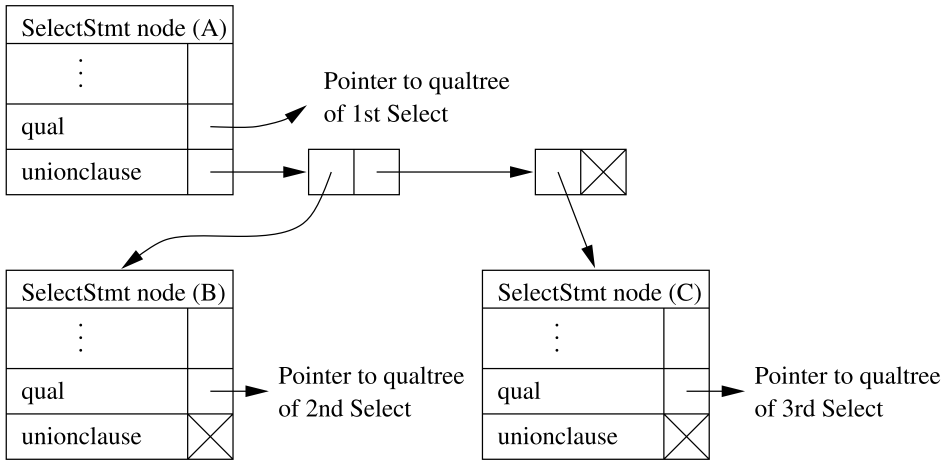

Figure 3.9: Data structure handed back by the *parser*

`SelectStmt` node (of B and C) and attaches it to the field `unionClause` of the first node (of A). Finally it hands back the first node (node A) with the list of the remaining nodes attached as shown in figure 3.9.

The following *transformation routines* process the data structure handed back by the *parser*. First the top node (node A) is transformed from a `SelectStmt` node to a `Query` node. The *targetlist*, the `where` qualification etc. attached to it are transformed as well. Next the list of the remaining nodes (attached to `unionClause` of node A) is transformed and in this step also a check is made if the types and lengths of the *targetlists* of the involved nodes are equal. The new `Query` nodes are now handed back in the same way as the `SelectStmt` nodes were before (i.e. the `Query` nodes B and C are collected in a list which is attached to `unionClause` of `Query` node A).

The Rewrite System

If any *rewrite rules* are present for the `Query` nodes (i.e. one of the *select* statements uses a *view*) the necessary changes to the `Query` nodes are performed (see section 3.4.1 *Techniques To Implement Views*). Otherwise no changes are made to the nodes in this stage.

Planner/Optimizer

This stage has to create a *plan* out of the *querytree* produced by the *parser stage* that can be executed by the *executor*. In most cases there are several ways (paths) with different cost to get to the same result. It's the *planner/optimizer's* task to find out which path is the cheapest and to create a *plan* using this path. The implementation of *unions* in PostgreSQL is based on the following idea:

The set derived by evaluating $A \cup B$ must contain every member of A **and** every member of B. So if we append the members of B to the members of A we are almost done. If there exist members common to A and B these members are now contained twice in our new set, so the only thing left to do is to remove these duplicates.

In the case of our example the *planner* would build up the *tree* shown in figure 3.10. Every `Query` node is planned separately and results in a `SeqScan` node in our example. The three `SeqScan` nodes are put together into a list which is attached to `unionplans` of an `Append` node.

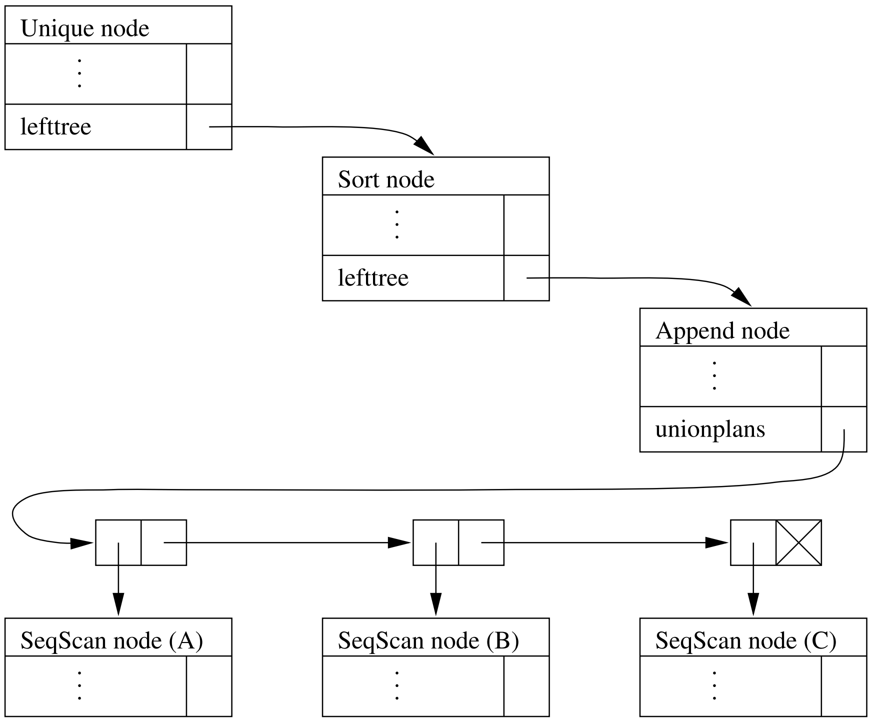

Figure 3.10: *Plan* for a union query

Executor

The *executor* will process all the `SeqScan` nodes and append all the delivered tuples to a single *result relation*. Now it is possible that duplicate tuples are contained in the *result relation* which have to be removed. The removal is done by the `Unique` node and the sort is just performed to make its work easier.

3.8.2 How Intersect, Except and Union Work Together

The last section showed that every stage (*parser stage*, *planner/optimizer*, *executor*) of PostgreSQL has to provide features in order to support *union* statements. For the implementation of *intersect* and *except* statements (and statements involving all *set operators*) we choose a different approach based on *query rewriting*.

The idea is based on the fact that *intersect* and *except* statements are redundant in SQL, i.e. for every *intersect* or *except* statement it is possible to formulate a semantically equivalent statement without using *intersect* or *except*.

Example 3.5 This example shows how a query using *intersect* can be transformed to a semantically equivalent query without an *intersect*:

```
select C1, C3 from A
where C1 = 1
intersect
select C1, C2 from B
where C2 = 'c';
```

is equivalent to:

```
select C1, C3
from A
where C1 = 1 and
      (C1, C3) in (select C1, C2
                   from B
                   where C2 = 'c');
```

This example shows how an *except* query can be transformed to an *except*-less form:

```
select C1, C2 from A
where C2 = 'c'
except
select C1, C3 from B
where C3 = 'f';
```

is equivalent to:

```
select C1, C2
from A
where C2 = 'c' and
      (C1, C2) not in (select C1, C3
                       from B
                       where C3 = 'f');
```

The transformations used in example 3.5 are always valid because they just implement the set theoretic definition of *intersect* and *except*:

Definition 3.1
The *intersection* of two sets A and B is defined as:

$$(A \cap B) := \{x \mid x \in A \wedge x \in B\}$$

The *intersection* of n sets $A_1, \ldots, A_n$ is defined as:

$$\bigcap_{i=1}^{n} A_i := \{x \mid \bigwedge_{i=1}^{n} x \in A_i\}$$

Definition 3.2
The *difference* of two sets A and B is defined as:

$$(A \backslash B) := \{x \mid x \in A \wedge x \notin B\}$$

Definition 3.3
The *union* of two sets A and B is defined as:

$$(A \cup B) := \{x \mid x \in A \vee x \in B\}$$

The *union* of n sets $A_1, \ldots, A_n$ is defined as:

$$\bigcup_{i=1}^{n} A_i := \{x \mid \bigvee_{i=1}^{n} x \in A_i\}$$

Definition 3.4 Disjunctive Normal Form (DNF)
Let $F = C_1 \vee \ldots \vee C_n$ be given where every C_i is of the form $(L_i^1 \wedge \ldots \wedge L_i^{k_i})$ and L_i^j is a propositional variable or the negation of a propositional variable. Now we say F is in DNF.

Example 3.6 In the following example the L_i^j are of the form $x \in X$ or $\neg(x \in X)$:
$((x \in A \wedge \neg(x \in B) \wedge x \in C) \vee (x \in D \wedge x \in E))$ is a formula in DNF
$((x \in A \vee x \in B) \wedge (x \in C \vee \neg(x \in D)))$ is not in DNF.

The transformation of any formula in propositional logic into DNF is done by successively applying the following rules:

$(R1)$ $\neg(F_1 \vee F_2) \Rightarrow (\neg F_1 \wedge \neg F_2)$
$(R2)$ $\neg(F_1 \wedge F_2) \Rightarrow (\neg F_1 \vee \neg F_2)$
$(R3)$ $F_1 \wedge (F_2 \vee F_3) \Rightarrow (F_1 \wedge F_2) \vee (F_1 \wedge F_3)$
$(R4)$ $(F_1 \vee F_2) \wedge F_3 \Rightarrow (F_1 \wedge F_3) \vee (F_2 \wedge F_3)$

It can be shown that the transformation using the rules (R1) to (R4) always terminates after a finite number of steps.

Set Operations as Propositional Logic Formulas

Using the definitions from above we can treat formulas involving set theoretic operations as formulas of *propositional logic*. As we will see later these formulas can easily be used in the `where-` and `having` qualifications of the `select` statements involved in the query.

Example 3.7 Here are some examples:

$((A \cup B) \cap C) := \{x \mid (x \in A \vee x \in B) \wedge x \in C\}$
$((A \cup B) \cap (C \cup D)) := \{x \mid (x \in A \vee x \in B) \wedge (x \in C \vee x \in D)\}$
$((A \cap B) \backslash C) := \{x \mid (x \in A \wedge x \in B) \wedge x \notin C\}$
$(A \backslash (B \cup C)) := \{x \mid x \in A \wedge \neg(x \in B \vee x \in C)\}$
$(((A \cap B) \cup (C \backslash D)) \cap E) := \{((x \in A \wedge x \in B) \vee (x \in C \wedge x \notin D)) \wedge x \in E\}$

3.8.3 Implementing Intersect and Except Using the Union Capabilities

We want to be able to use queries involving more than just one type of set operation (e.g. only *union* or only *intersect*) at a time, so we have to look for a solution that supports correct handling of queries like that. As described above there is a solution for pure *union* statements implemented already, so we have to develop an approach that makes use of these *union* capabilities.

As figure 3.9 illustrates, the operands of a *union* operation are just `Query` nodes (the first operand is the top node and all further operands form a list which is attached to the field `unionClause` of the top node). So our goal will be to transform every query involving set operations into this form. (Note that the operands to the *union* operation may be complex, i.e. *subselects*, *grouping*, *aggregates* etc. are allowed.)

The transformation of a query involving set operations in any order into a query that can be accepted by the *union* logic is equivalent to transforming the *membership formula* (see definitions 3.1, 3.2 and 3.3) in propositional logic into *disjunctive normal form* (DNF). The transformation of any formula in propositional logic into DNF is always possible in a finite number of steps.

The advantage of this *transformation technique* is the little impact on the whole system and the implicit invocation of the *planner/optimizer*. The only changes necessary are made to the *parser stage* and the *rewrite system*.

Here are some changes that had to be applied to the source code before the *parser stage* and the *rewrite system* could be adapted:

- Add the additional field `intersectClause` to the data structures `Query` and `InsertStmt` defined in the file `.../src/include/nodes/parsenodes.h`:

```
  typedef struct Query
  {
    NodeTag     type;
    CmdType     commandType;
              .
              .
              .
    Node        *havingQual;
+   List        *intersectClause;
    List        *unionClause;
    List        *base_relation_list_;
    List        *join_relation_list_;
  } Query;

  typedef struct InsertStmt
  {
    NodeTag     type;
              .
              .
              .
    bool        unionall;
+   List        *intersectClause;
  } InsertStmt;
```

- Add the new keywords `EXCEPT` and `INTERSECT` to the file `.../src/backend/parser/keywords.c`:

```
  static ScanKeyword ScanKeywords[] = {
    {"abort", ABORT_TRANS},
    {"action", ACTION},
                .
                .
                .
    {"end", END_TRANS},
+   {"except", EXCEPT},
                .
                .
                .
    {"instead", INSTEAD},
+   {"intersect", INTERSECT},
                .
                .
                .
  };
```

- PostgreSQL contains functions to convert the internal representation of a *parsetree* or *plantree* into an ASCII representation (that can easily be printed to the screen (for debugging purposes) or be stored in a file) and vice versa. These functions have to be adapted to be able to deal with *intersects* and *excepts*. These functions can be found in the files `.../src/backend/nodes/outfuncs.c` and `.../src/backend/nodes/readfuncs.c`:

```
  static void
  _outQuery(StringInfo str, Query *node)
  {
                              .
                              .
                              .
    appendStringInfo(str, " :unionClause ");
    _outNode(str, node->unionClause);
+   appendStringInfo(str, " :intersectClause ");
+   _outNode(str, node->intersectClause);
  }

  static Query *
  _readQuery()
  {
                              .
                              .
                              .
    token = lsptok(NULL, &length);
    local_node->unionClause = nodeRead(true);
+   token = lsptok(NULL, &length);
+   local_node->intersectClause = nodeRead(true);

    return (local_node);
  }
```

- The function `ExecReScan()` is called whenever a new execution of a given *plan* has to be started (i.e. whenever we have to start from the beginning with the first tuple again). The call to this function happens implicitly. For the special kind of subqueries we are using for the rewritten queries (see example 3.5) we have to take that also `Group` nodes are processed. The function can be found in the file $ldots$`/backend/executor/execAmi.c`.

```
  void
  ExecReScan(Plan *node, ExprContext *exprCtxt,
             Plan *parent)
  {
                              .
                              .
                              .
     switch (nodeTag(node))
     {
                              .
                              .
                              .
```

```
+       case T_Group:
+               ExecReScanGroup((Group *) node,
+                                exprCtxt, parent);
+               break;
                            .
                            .
                            .
    }
  }
```

- The function `ExecReScanGroup()` is called by `ExecReScan()` described above whenever a `Group` node is detected and can be found in the file `.../src/backend/executor/nodeGroup.c` . It has been created for the *intersect* and *except* logic although it is actually needed by the special kind of subselect (see above).

```
  void
  ExecReScanGroup(Group *node, ExprContext *exprCtxt,
                  Plan *parent)
  {
    GroupState *grpstate = node->grpstate;

    grpstate->grp_useFirstTuple = FALSE;
    grpstate->grp_done = FALSE;
    grpstate->grp_firstTuple = (HeapTupleData *)NIL;

    /*
     * if chgParam of subnode is not null then plan
     * will be re-scanned by first ExecProcNode.
     */
    if (((Plan *) node)->lefttree->chgParam == NULL)
      ExecReScan(((Plan *) node)->lefttree,
                 exprCtxt, (Plan *) node);
  }
```

Parser

The *parser* defined in the file `.../src/backend/parser/gram.y` had to be modified in two ways:

- The grammar had to be adapted to support the usage of parenthesis (to be able to specify the order of execution of the set operators).

- The code building up the data structures handed back by the *parser* had to be inserted.

Here is a part of the grammar which is responsible for `select` statements having the code building up the data structures inserted:

```
SelectStmt :  select_w_o_sort sort_clause
    {
                         .
                         .
                         .
       /* $1 holds the tree built up by the
        * rule 'select_w_o_sort'
        */
       Node *op = (Node *) $1;

       if IsA($1, SelectStmt)
       {
         SelectStmt *n = (SelectStmt *)$1;
         n->sortClause = $2;
         $$ = (Node *)n;
       }
       else
       {
         /* Create a "flat list" of the operator
          * tree built up by 'select_w_o_sort' and
          * let select_list point to it
          */
         create_select_list((Node *)op,
                            &select_list,
                            &unionall_present);
         /* Replace all the A_Expr nodes in the
          * operator tree by Expr nodes.
          */
          op = A_Expr_to_Expr(op, &intersect_present);
                         .
                         .
                         .
          /* Get the leftmost SelectStmt node (which
           * automatically represents the first Select
           * Statement of the query!) */
          first_select =
                 (SelectStmt *)lfirst(select_list);
          /* Attach the list of all SelectStmt nodes
           * to unionClause
           */
          first_select->unionClause = select_list;

          /* Attach the whole operator tree to
           * intersectClause */
          first_select->intersectClause =
                                        (List *) op;
          /* finally attach the sort clause */
          first_select->sortClause = $2;

          /* Now hand it back! */
          $$ = (Node *)first_select;
        }
    }
;
```

```
select_w_o_sort :  '(' select_w_o_sort ')'
      {
        $$ = $2;
      }
  |   SubSelect
      {
        $$ = $1;
      }
  |   select_w_o_sort EXCEPT select_w_o_sort
      {
        $$ = (Node *)makeA_Expr(AND,NULL,$1,
                         makeA_Expr(NOT,NULL,NULL,$3));
      }
  |   select_w_o_sort UNION opt_union select_w_o_sort
      {
        if (IsA($4, SelectStmt))
        {
          SelectStmt *n = (SelectStmt *)$4;
          n->unionall = $3;
        }
        $$ = (Node *)makeA_Expr(OR,NULL,$1,$4);
      }
  |   select_w_o_sort INTERSECT select_w_o_sort
      {
        $$ = (Node *)makeA_Expr(AND,NULL,$1,$3);
      }
;

SubSelect :  SELECT opt_unique res_target_list2
             result from_clause where_clause
             group_clause having_clause
    {
      SelectStmt *n = makeNode(SelectStmt);
      n->unique = $2;
              .
              .
              .
      n->havingClause = $8;
      $$ = (Node *)n;
    }
;
```

The keywords SELECT, EXCEPT, UNION, INTERSECT, '(' and ')' are *terminal symbols* and SelectStmt, select_w_o_sort, sort_clause, opt_union, SubSelect, opt_unique, res_target_list2, result, from_clause, where_clause, group_clause, having_clause are *nonterminal symbols*. The symbols EXCEPT, UNION and INTERSECT are *left associative* meaning that a statement like:

```
select * from A
union
select * from B
union
select * from C;
```

will be treated as:

```
((select * from A
  union
  select * from B)
  union
  select * from C)
```

The `select_w_o_sort` rule builds up an *operator tree* using nodes of type `A_Expr`. For every *union* an `OR` node is created, for every *intersect* an `AND` node and for every *except* and `AND NOT` node building up a representation of a formula in propositional logic. If the query parsed did not contain any set operations the rule hands back a `SelectStmt` node representing the query otherwise the top node of the *operator tree* is returned. Figure 3.11 shows a typical *operator tree* returned by the `select_w_o_sort` rule.

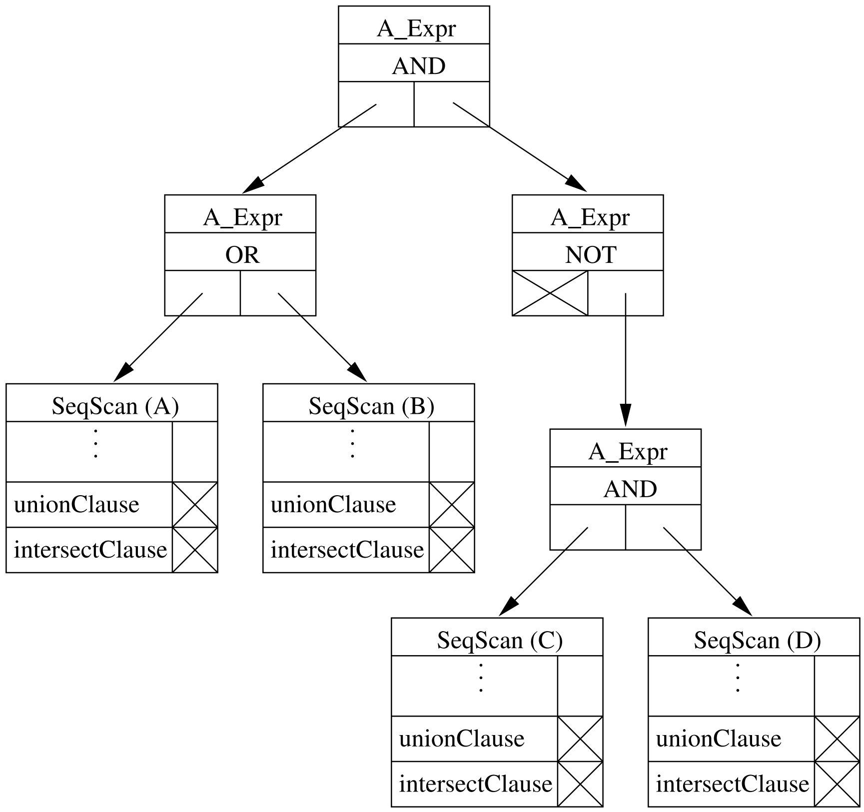

Figure 3.11: *Operator tree* for $(A \cup B) \backslash (C \cap D)$

The rule `SelectStmt` transforms the *operator tree* built of `A_Expr` nodes into an *operator tree* using `Expr` nodes by a call to the function `A_Expr_to_Expr()` which additionally replaces every `OR` node by an `AND` node and vice versa. This is performed in order to be able to use the function `cnfify()` later on.

The *transformations* following the *parser* expect a `SelectStmt` node to be returned by the rule `SelectStmt` and not an *operator tree*. So if the rule `select_w_o_sort` hands back such a node (meaning that the query did not contain any set operations) we just have to attach the data structure built up by the `sort_clause` rule and are finished, but when we get an *operator tree* we have to perform the following steps:

- Create a flat list of all `SelectStmt` nodes of the *operator tree* (by a call to the function `create_select_list()`) and attach the list to the field `unionClause` of the leftmost `SelectStmt` (see next step).
- Find the leftmost leaf (`SelectStmt` node) of the *operator tree* (this is automatically the first member of the above created list because of the technique `create_select_list()` uses to create the list).
- Attach the whole *operator tree* (including the leftmost node itself) to the field `intersectClause` of the leftmost `SelectStmt` node.
- Attach the data structure built up by the `sort_clause` rule to the field `sortClause` of the leftmost `SelectStmt` node.
- Hand back the leftmost `SelectStmt` node (with the *operator tree*, the list of all `SelectStmt` nodes and the `sortClause` attached to it).

Figure 3.12 shows the final data structure derived from the *operator tree* shown in figure 3.11 handed back by the `SelectStmt` rule:

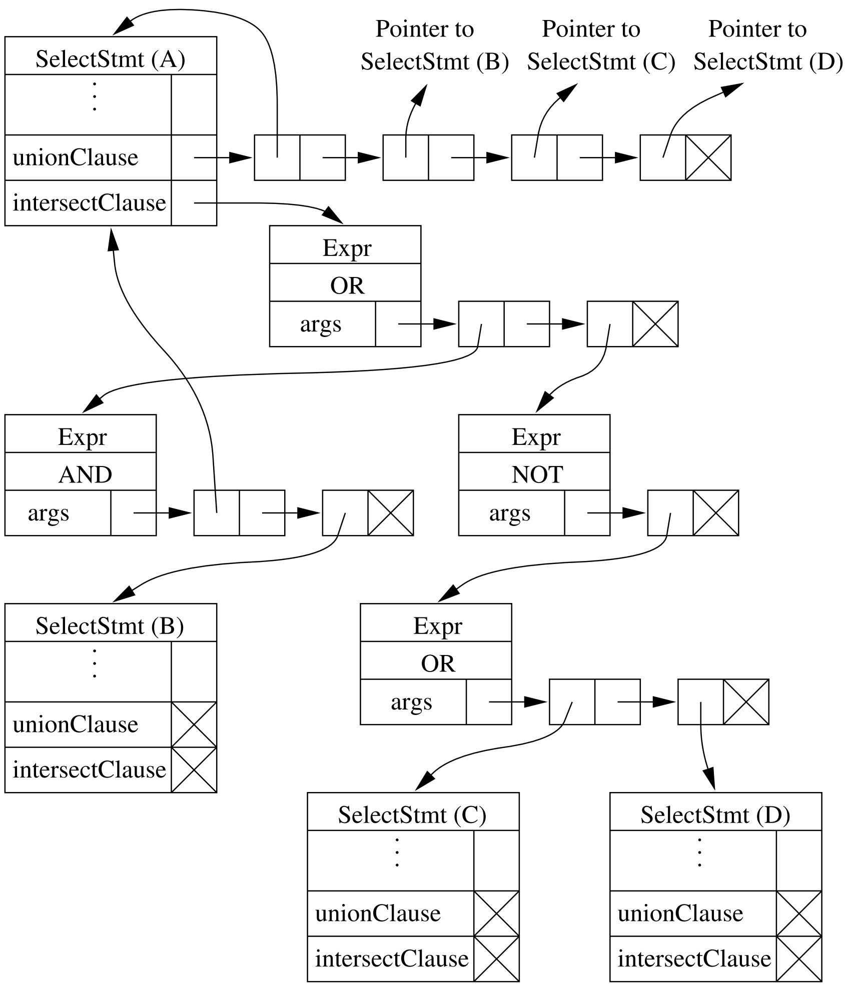

Figure 3.12: Data structure handed back by `SelectStmt` rule

Here is a description of the above used functions. They can be found in the file `.../src/backend/parser/anlayze.c`.

- `create_select_list()`:
  This function steps through the *tree* handed to it by the parameter `ptr` and creates a list of all `SelectStmt` nodes found. The list is handed back by the parameter `select_list`. The function uses a recursive *depth first search* algorithm to examine the *tree* leading to the fact that the leftmost `SelectStmt` node will appear first in the created list.

```
  void
  create_select_list(Node *ptr, List **select_list,
                     bool *unionall_present)
  {
    if(IsA(ptr, SelectStmt))
    {
      *select_list = lappend(*select_list, ptr);
      if(((SelectStmt *)ptr)->unionall == TRUE)
        *unionall_present = TRUE;
      return;
    }

    /* Recursively call for all arguments.
     * A NOT expr has no lexpr!
     */
    if (((A_Expr *)ptr)->lexpr != NULL)
      create_select_list(((A_Expr *)ptr)->lexpr,
                         select_list, unionall_present);
    create_select_list(((A_Expr *)ptr)->rexpr,
                       select_list, unionall_present);
}
```

- `A_Expr_to_Expr()`:
  This function recursively steps through the *operator tree* handed to it by the parameter `ptr` and replaces `A_Expr` nodes by `Expr` nodes. Additionally it exchanges `AND` nodes with `OR` nodes and vice versa. The reason for this exchange is easy to understand. We implement *intersect* and *except* clauses by rewriting these queries to semantically equivalent queries that use `IN` and `NOT IN` subselects. To be able to use all three operations (*unions*, *intersects* and *excepts*) in one complex query, we have to translate the queries into *Disjunctive Normal Form* (DNF). Unfortunately there is no function `dnfify()`, but there is a function `cnfify()` which produces DNF when we exchange `AND` nodes with `OR` nodes and vice versa before calling `cnfify()` and exchange them again in the result.

```
  Node *
  A_Expr_to_Expr(Node *ptr,
                 bool *intersect_present)
  {
    Node *result;

    switch(nodeTag(ptr))
    {
      case T_A_Expr:
      {
```

```
    A_Expr *a = (A_Expr *)ptr;

    switch (a->oper)
    {
      case AND:
      {
        Expr *expr = makeNode(Expr);
        Node *lexpr =
           A_Expr_to_Expr(((A_Expr *)ptr)->lexpr,
                          intersect_present);
        Node *rexpr =
           A_Expr_to_Expr(((A_Expr *)ptr)->rexpr,
                          intersect_present);

        *intersect_present = TRUE;

        expr->typeOid = BOOLOID;
        expr->opType = OR_EXPR;
        expr->args = makeList(lexpr, rexpr, -1);
        result = (Node *) expr;
        break;
      }
      case OR:
      {
        Expr *expr = makeNode(Expr);
        Node *lexpr =
           A_Expr_to_Expr(((A_Expr *)ptr)->lexpr,
                          intersect_present);
        Node *rexpr =
           A_Expr_to_Expr(((A_Expr *)ptr)->rexpr,
                          intersect_present);

        expr->typeOid = BOOLOID;
        expr->opType = AND_EXPR;
        expr->args = makeList(lexpr, rexpr, -1);
        result = (Node *) expr;
        break;
      }
      case NOT:
      {
        Expr *expr = makeNode(Expr);
        Node *rexpr =
           A_Expr_to_Expr(((A_Expr *)ptr)->rexpr,
                          intersect_present);

        expr->typeOid = BOOLOID;
        expr->opType = NOT_EXPR;
        expr->args = makeList(rexpr, -1);
        result = (Node *) expr;
        break;
      }
    }
    break;
  }
```

```
      default:
      {
        result = ptr;
      }
    }
  }
  return result;
}
```

Note that the `stmtmulti` and `OptStmtMulti` rules had to be changed in order to avoid *shift/reduce* conflicts. The old rules allowed an invalid syntax (e.g. the concatenation of two statements without a ';' inbetween) which will be prevented now. The new rules have the second line commented out as shown below:

```
stmtmulti          :  stmtmulti stmt ';'
                /* |  stmtmulti stmt */
                   |  stmt ';'
                   ;

OptStmtMulti       :  OptStmtMulti OptimizableStmt ';'
                /* |  OptStmtMulti OptimizableStmt */
                   |  OptimizableStmt ';'
                   ;
```

Transformations

This step normally transforms every `SelectStmt` node found into a `Query` node and does a lot of transformations to the *targetlist*, the `where` qualification etc. As mentioned above this stage expects a `SelectStmt` node and cannot handle an `A_Expr` node. That's why we did the changes to the *operator tree* shown in figure 3.12.

In this stage only very few changes have been necessary:

- The transformation of the list attached to `unionClause` is prevented. The raw list is now passed through instead and the necessary transformations are performed at a later point in time.
- The additionally introduced field `intersectClause` is also passed untouched through this stage.

The changes described in the above paragraph have been applied to the functions `transformInsertStmt()` and `transformSelectStmt()` which are contained in the file `.../src/backend/parser/analyze.c`:

- `transformInsertStmt()`:

```
static Query *
transformInsertStmt(ParseState *pstate,
                    InsertStmt *stmt)
{
                            .
                            .
                            .
```

```
    /* Just pass through the unionClause and
     * intersectClause. We will process it in
     * the function Except_Intersect_Rewrite()
     */
    qry->unionClause = stmt->unionClause;
    qry->intersectClause = stmt->intersectClause;
                           .
                           .
                           .

    return (Query *) qry;
  }
```

- `transformSelectStmt()`:

```
  static Query *
  transformSelectStmt(ParseState *pstate,
                      SelectStmt *stmt)
  {
                           .
                           .
                           .
    /* Just pass through the unionClause and
     * intersectClause. We will process it in
     * the function Except_Intersect_Rewrite()
     */
    qry->unionClause = stmt->unionClause;
    qry->intersectClause = stmt->intersectClause;
                           .
                           .
                           .
    return (Query *) qry;
  }
```

The Rewrite System

In this stage the information contained in the *operator tree* attached to the topmost `SelectStmt` node is used to form a tree of `Query` nodes representing the rewritten query (i.e. the semantically equivalent query that contains only *union* but no *intersect* or *except* operations).

The following steps have to be performed:

- Save the `sortClause`, `uniqueFlag`, `targetList` fields etc. of the topmost `Query` node because the topmost node may change during the rewrite process (remember (only) the topmost `SelectStmt` node has already been transformed to a `Query` node).

- Recursively step through the *operator tree* and transform every `SelectStmt` node to a `Query` node using the function `intersect_tree_analyze()` described below. The one node already transformed (the topmost node) is still contained in the *operator tree* and must not be transformed again because this would cause troubles in the *transforming logic*.

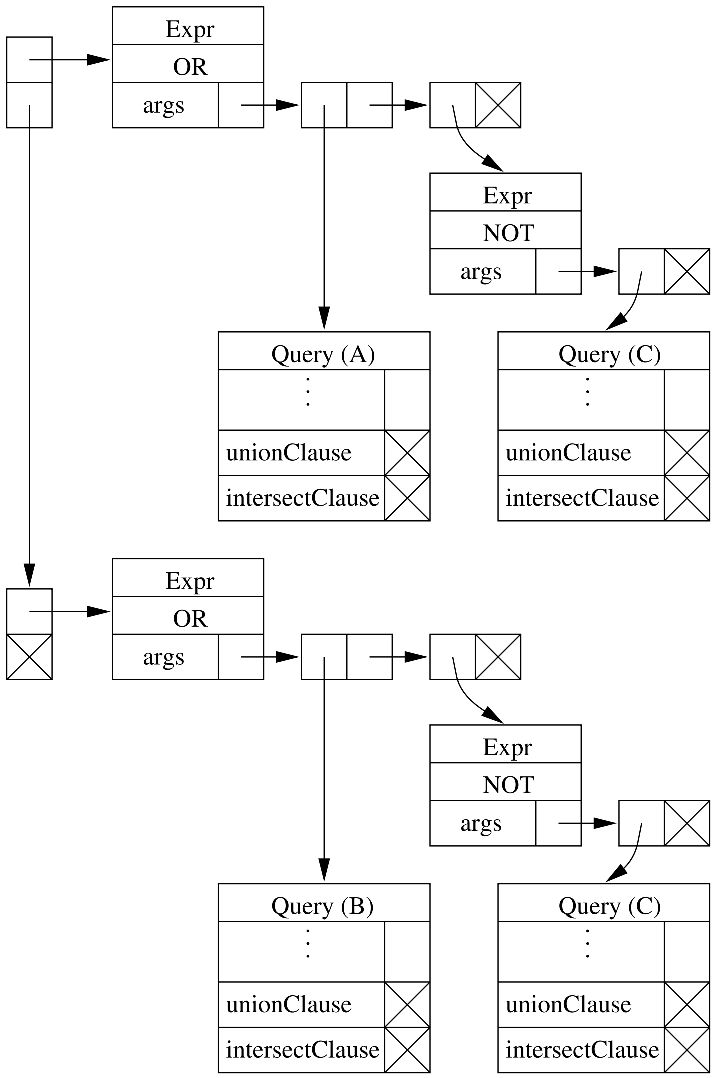

Figure 3.13: Data structure of $(A \cup B) \backslash C$ after transformation to DNF

- Transform the new *operator tree* into DNF (disjunctive normal form). PostgreSQL does not provide any function for the transformation into DNF but it provides a function `cnfify()` that performs a transformation into CNF (conjunctive normal form). So we can easily make use of this function when we exchange every `OR` with an `AND` and vice versa before calling `cnfify()` as we did already in the *parser* (compare figure 3.11 to figure 3.12). When `cnfify()` is called with a special flag, the `removeAndFlag` set to `true` it returns a list where the entries can be thought of being connected together by `AND`s, so the explicit `AND` nodes are removed.

  After `cnfify()` has been called we normally would have to exchange `OR` and `AND` nodes again. We skip this step by simply treating every `OR` node as an `AND` node throughout the following steps (remember, that there are no `AND` nodes left that have to be treated as `OR` nodes because of the `removeAndFlag`).

  Figure 3.13 shows what the data structure looks like after the transformation to DNF has taken place for the following query:

```
(select * from A
 union
 select * from B)
 except
 select * from C;
```

- For every entry of the list returned by `cnfify()` (i.e. for every *operator tree* which may only contain `OR` and `NOT` operator nodes and `Query` nodes as leaves) contained in the `union_list` perform the following steps:
  - Check if the *targetlists* of all `Query` nodes appearing are compatible (i.e. all *targetlists* have the same number of attributes and the corresponding attributes are of the same type)
  - There must be at least one positive `OR` node (i.e. at least one `OR` node which is not preceded by a `NOT` node). Create a list of all `Query` nodes (or of `Query` nodes preceded by `NOT` nodes if `OR NOT` is found) with the non negated node first using the function `create_list()` described below.
  - The first (non negated) node of the list will be the topmost `Query` node of the current *union* operand. For all other nodes found in the list add an `IN` subselect (`NOT IN` subselect if the `Query` node is preceded by a `NOT`) to the `where` qualification of the topmost node. Adding a subselect to the `where` qualification is done by logically `AND`ing it to the original qualification.
  - Append the `Query` node setup in the last steps to a list which is hold by the pointer `union_list`.
- Take the first node of `union_list` as the new topmost node of the whole query and attach the rest of the list to the field `unionClause` of this topmost node. Since the new topmost node might differ from the original one (i.e. from the node which was topmost when we entered the *rewrite stage*) we have to attach the fields saved in the first step to the new topmost node (i.e. the `sortClause`, `targetList`, `unionFlag`, etc.).
- Hand the new topmost `Query` node back. Now the normal *query rewriting* takes place (in order to handle views if present) and then the *planner/optimizer* and *executor* functions are called to get a result. There have no changes been made to the code of these stages.

Figure 3.14 shows the rewritten data structure of the query:

```
select C1, C2 from A
intersect
select C1, C3 from C;
```

against the tables defined in example 3.4. The rewritten data structure represents the query:

```
select C1, C2 form A
where (C1, C2) in
      (select C1,C3 from C);
```

The field `lefttree` of the `Sublink` node points to a list where every entry points to a `VAR` node of the *targetlist* of the topmost node (node A). The field `oper` of the `Sublink` node points to a list holding a pointer to an `Expr` node for every attribute of the topmost *targetlist*. Every `Expr` node is used to compare a `VAR` node of the topmost *targetlist* with the corresponding `VAR` node of the subselect's *targetlist*. So the first argument of every `Expr` node points to a `VAR` node of the topmost *targetlist* and the second argument points to the corresponding `VAR` node of the subselect's *targetlist*.

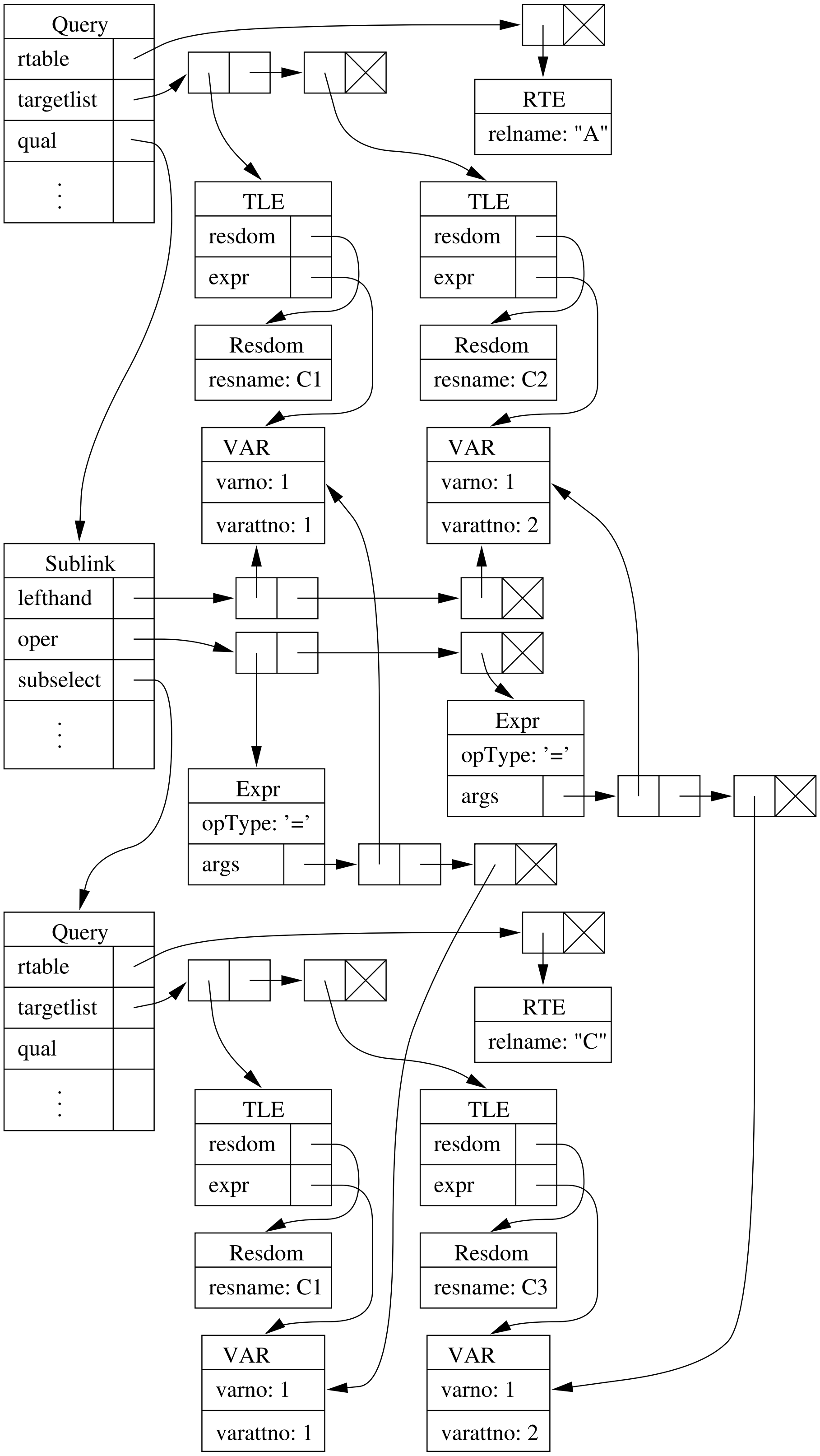

Figure 3.14: Data structure of $A \cap C$ after query rewriting

If the user's query involves *union* as well as *intersect* or *except* there will be more `Query` nodes of the form shown in figure 3.14. One will be the topmost node (as described above) and the others will be collected in a list which is attached to the field `unionClause` of the topmost node. (The *intersectClause* fields of all `Query` nodes will be set to `NULL` because they are no longer needed.)

The function `pg_parse_and_plan()` is responsible for invoking the rewrite procedure. It can be found in the file `.../src/backend/tcop/postgres.c`.

```
List *
pg_parse_and_plan(char *query_string, Oid *typev,
                  int nargs,
                  QueryTreeList **queryListP,
                  CommandDest dest)
{
                          .
                          .
                          .
  /* Rewrite Union, Intersect and Except Queries
   * to normal Union Queries using IN and NOT
   * IN subselects */
  if(querytree->intersectClause != NIL)
  {
    querytree = Except_Intersect_Rewrite(querytree);
  }
                          .
                          .
                          .
}
```

Here are the functions that have been added to perform the functionality described above. They can be found in the file `.../src/backend/rewrite/rewriteHandler.c`.

- `Except_Intersect_Rewrite()`
  Rewrites queries involving *union clauses*, *intersect clauses* and *except clauses* to semantiacally equivalent queries that use `IN` and `NOT IN` subselects instead.

  The *operator tree* is attached to `intersectClause` (see rule `SelectStmt` above) of the *parsetree* given as an argument. First we save some clauses (the `sortClause`, the `unique flag` etc.). Then we translate the *operator tree* to DNF (*Disjunctive Normal Form*) by `cnfify()`. Note that `cnfify()` produces CNF but as we exchanged `AND` nodes with `OR` nodes within function `A_Expr_to_Expr()` earlier we get DNF when we exchange `AND` nodes and `OR` nodes again in the result. Now we create a new (rewritten) query by examining the new *operator tree* which is in DNF now. For every `AND` node we create an entry in the *union list* and for every `OR` node we create an `IN` subselect. (`NOT IN` subselects are created for `OR NOT` nodes). The first entry of the *union list* is handed back but before that the saved clauses (`sortClause` etc.) are restored to the new top node. Note that the new top node can differ from the one of the *parsetree* given as argument because of the translation into DNF. That's why we had to save the `sortClause` etc.

```
Query *
Except_Intersect_Rewrite (Query *parsetree)
{
                            .
                            .
                            .
  /* Save some fields, to be able to restore them
   * to the resulting top node at the end of the
   * function
   */
  sortClause = parsetree->sortClause;
  uniqueFlag = parsetree->uniqueFlag;
  into = parsetree->into;
  isBinary = parsetree->isBinary;
  isPortal = parsetree->isPortal;

  /* Transform the SelectStmt nodes into Query nodes
   * as usually done by transformSelectStmt() earlier.
   * /
  intersectClause =
    (List *)intersect_tree_analyze(
            (Node *)parsetree->intersectClause,
            (Node *)lfirst(parsetree->unionClause),
            (Node *)parsetree);
                            .
                            .
                            .
  /* Transform the operator tree to DNF */
  intersectClause =
             cnfify((Expr *)intersectClause, true);
  /* For every entry of the intersectClause list we
   * generate one entry in the union_list
   */
  foreach(intersect, intersectClause)
  {
    /* For every OR we create an IN subselect and
     * for every OR NOT we create a NOT IN subselect,
     */
    intersect_list = NIL;
    create_list((Node *)lfirst(intersect),
                &intersect_list);
    /* The first node will become the Select Query
     * node, all other nodes are transformed into
     * subselects under this node!
     */
    intersect_node = (Query *)lfirst(intersect_list);
    intersect_list = lnext(intersect_list);
                            .
                            .
                            .
    /* Transform all remaining nodes into subselects
     * and add them to the qualifications of the
     * Select Query node
     */
```

```
while(intersect_list != NIL)
{
  n = makeNode(SubLink);

  /* Here we got an OR so transform it to an
   * IN subselect
   */
  if(IsA(lfirst(intersect_list), Query))
  {
                      .
                      .
                      .
    n->subselect = lfirst(intersect_list);
    op = "=";
    n->subLinkType = ANY_SUBLINK;
    n->useor = false;
  }

  /* Here we got an OR NOT node so transform
   * it to a NOT IN  subselect
   */
  else
  {
                      .
                      .
                      .
    n->subselect =
        (Node *)lfirst(((Expr *)
              lfirst(intersect_list))->args);
    op = "<>";
    n->subLinkType = ALL_SUBLINK;
    n->useor = true;
  }

  /* Prepare the lefthand side of the Sublinks:
   * All the entries of the targetlist must be
   * (IN) or must not be (NOT IN) the subselect
   */
  foreach(elist, intersect_node->targetList)
  {
    Node        *expr = lfirst(elist);
    TargetEntry *tent = (TargetEntry *)expr;

    n->lefthand =
            lappend(n->lefthand, tent->expr);
  }

  /* The first arguments of oper also have to be
   * created for the sublink (they are the same
   * as the lefthand side!)
   */
  left_expr = n->lefthand;
  right_expr =
       ((Query *)(n->subselect))->targetList;
```

```
        foreach(elist, left_expr)
        {
          Node         *lexpr = lfirst(elist);
          Node         *rexpr = lfirst(right_expr);
          TargetEntry *tent = (TargetEntry *) rexpr;
          Expr         *op_expr;

          op_expr = make_op(op, lexpr, tent->expr);
          n->oper = lappend(n->oper, op_expr);
          right_expr = lnext(right_expr);
        }

        /* If the Select Query node has aggregates
         * in use add all the subselects to the
         * HAVING qual else to the WHERE qual
         */
        if(intersect_node->hasAggs == false)
        {
          AddQual(intersect_node, (Node *)n);
        }
        else
        {
          AddHavingQual(intersect_node, (Node *)n);
        }

        /* Now we got sublinks */
        intersect_node->hasSubLinks = true;
        intersect_list = lnext(intersect_list);
      }
      intersect_node->intersectClause = NIL;
      union_list = lappend(union_list, intersect_node);
    }

    /* The first entry to union_list is our
     * new top node
     */
    result = (Query *)lfirst(union_list);

    /* attach the rest to unionClause */
    result->unionClause = lnext(union_list);

    /* Attach all the items saved in the
     * beginning of the function */
    result->sortClause = sortClause;
    result->uniqueFlag = uniqueFlag;
    result->into = into;
    result->isPortal = isPortal;
    result->isBinary = isBinary;
                               .
                               .
                               .
    return  result;
  }
```

- `create_list()`
  Create a list of nodes that are either `Query` nodes or `NOT` nodes followed by a `Query` node. The *tree* given in `ptr` contains at least one *non negated* `Query` node. This node is put to the beginning of the list.

```
void create_list(Node *ptr,
                 List **intersect_list)
{
  List *arg;

  if(IsA(ptr,Query))
  {
    /* The non negated node is attached at the
     * beginning (lcons) */
    *intersect_list = lcons(ptr, *intersect_list);
    return;
  }
  if(IsA(ptr,Expr))
  {
    if(((Expr *)ptr)->opType == NOT_EXPR)
    {
      /* negated nodes are appended to the
       * end (lappend)
       */
      *intersect_list =
             lappend(*intersect_list, ptr);
      return;
    }
    else
    {
      foreach(arg, ((Expr *)ptr)->args)
      {
        create_list(lfirst(arg), intersect_list);
      }
      return;
    }
    return;
  }
}
```

- `intersect_tree_analyze()`
  The nodes given in `tree` are not transformed yet so process them using `parse_analyze()`. The node given in `first_select` has already been processed, so don't transform it again but return a pointer to the already processed version given in `parsetree` instead.

```
Node *intersect_tree_analyze(Node *tree,
              Node *first_select, Node *parsetree)
{
  Node *result;
  List *arg;

  if(IsA(tree, SelectStmt))
  {
```

```
    QueryTreeList *qtree;

    /* If we get to the tree given in first_select
     * return parsetree instead of performing
     * parse_analyze() */
    if(tree == first_select)
    {
      result = parsetree;
    }
    else
    {
      /* transform the unprocessed Query nodes */
      qtree =
           parse_analyze(lcons(tree, NIL), NULL);
      result = (Node *)qtree->qtrees[0];
    }
  }
  if(IsA(tree,Expr))
  {
    /* Call recursively for every argument */
     foreach(arg, ((Expr *)tree)->args)
     {
       lfirst(arg) =
           intersect_tree_analyze(lfirst(arg),
                        first_select, parsetree);
     }
     result = tree;
  }
  return result;
}
```

Summary

PostgreSQL has become one of the most popular non commercial RDBMSs in the UNIX world. It provides an extended subset of the SQL92 standard as query language, allows concurrent database access, provides a huge amount of datatypes, etc. At the the time of writing this document the implemented part of SQL92 in PostgreSQL lacked two important features:

- The *having clause* was not implemented.
- The use of *except* and *intersect* statements was not possible.

The implementation of these two items was the motivation for the whole work. When I started to implement the above mentioned features, I noticed that there was almost no documentation on the internal structure of PostgreSQL available that could help a programmer to find his way in. So I decided to include all the knowledge I collected while working with the source code into this document, hoping that it will be a useful for any newcomer who wants to enhance PostgreSQL etc. Additionally I included a short discussion on SQL and a description of PostgreSQL's features (like *triggers* etc.) and how they can be used.

Chapter 1 discusses the theoretical (mathematical) background of relational database management systems (RDBMSs) ending in a short description of SQL.

Chapter 2 first gives an overview of how to setup and administrate PostgreSQL. Next some of PostgreSQL's special features are presented (i.e. *multiple inheritance*, *user defined data types*, *rules*, *triggers*, etc.) using a lot of examples.

Chapter 3 first gives an overview on the internal structure of PostgreSQL and presents the stages and data structures that are involved whenever a SQL query arrives. The *parser stage*, the *rule system* (which is mainly used for the implementation of *views*), the *planner/optimizer* and the *executor* are described and illustrated by a lot of figures. After that the changes necessary for the implementation of the missing features (*having clause* and *except/intersect*) are presented including parts of the added source code.

Bibliography

[DATE96] C. J. Date with H. Darwen: *A Guide to the SQL Standard*, Fourth Edition, Addison-Wesley 1996

[ULL88] Jeffrey D. Ullman: *Principles of Database and Knowledge - Base Systems*, Volume 1, Computer Science Press 1988

[DATE94] C. J. Date: *An Introduction to Database Systems*, Vol. 1, 6th Edition, Addison-Wesley 1994

[LOCK98] Thomas Lockhart: *PostgreSQL Programmer's Guide*, part of the PostgreSQL documentation.

[STON89] Michael Stonebraker et. al.: *On Rules, Procedures, Caching and Views in Data Base Systems*.