QUERY TREE SPECIFICATION
(Printed: May 6, 1994)

This document describes the query tree as output by the parser. The tree is represented internally as a LISP object, which can be displayed by calling the procedure **lispDisplay**(lispObject). In addition, there are three backend flags that can be set to display the query tree at different stages of processing :

DebugPrintParse, **DebugPrintRewrittenParsetree** and
DebugPrintPlan

All three are set when POSTGRES is run in debug mode.

For **retrieve**, **replace**, **append** and **delete** commands, the parser will produce a query tree as described below. Other commands produce a list in which the first element is the command name. The format of the rest of the tree is command-specific and will not be described here.

For notational purposes, anything appearing in italics is defined elsewhere in this document. Anything in bold face surrounded by quotes appears explicitly within a runtime query tree/plan as a LISP string. Anything appearing as a list is represented literally as a LISP list. [] indicates something that is optional; {} indicates 0 or more.

1. Query Tree Representation

The parser will produce the following query tree for each query that requires optimization:

(*Root TargetList Qualification*)

Each of the above three components is a list containing further information.

1.1. Root

Root looks like:

(*NumLevels CommandType ResultRelation RangeTable Priority RuleInfo UniqueFlag SortClause*)

This list contains information required by the planner as well as the executor. The planner itself does not modify any of these elements, although some of its preprocessing routines may change portions of root before it is seen by the executor.

1.1.1. NumLevels

NumLevels indicates the maximum attribute nesting in a query. That is, it indicates the maximum number of "nested dot" fields found in any variable in that query. For example, a query without range variables such as

retrieve (x = 1 + 2)

will have **NumLevels** = 0, a query such as

retrieve (pinhead.name) where pinhead.saying = "Yow!!!"

that contains normal range variables will have **NumLevels** = 1, a query such as

retrieve (sorority.all) where sorority.members.name = "Marti"

will have **NumLevels** = 2, and so on.

1.1.2. CommandType

CommandType indicates what kind of query is being executed. **CommandType** will be a single symbol from:

retrieve append replace delete

(other POSTQUEL commands are not optimized, so the planner should never see them).

1.1.3. ResultRelation

ResultRelation specifies the target relation of a ‘‘retrieve’’ or the update relation for ‘‘append’’, ‘‘replace’’, and ‘‘delete’’. It may take one of three forms:

(1) If the query is a ‘‘retrieve’’ and no result relation is specified, then **ResultRelation** is nil.

(2) If the query is a ‘‘retrieve into’’, then **ResultRelation** is a list containing the symbol **into** and the result relation's name.

(3) If the query is an update, **ResultRelation** will always be the index into the *rangetable* corresponding to the relation to be updated.

1.1.4. RangeTable

RangeTable is a list containing a *rangetable entry* for each instance of a relation (i.e., range variable) in a query. Every *relid* and *varno* within the query tree, and almost every one within the resulting query plan, is an integer index within the query's rangetable. This index is 1-based (e.g., var nodes that refer to the first relation in the rangetable are given varno ‘‘1’’).

A rangetable entry looks like:

(*InstanceVariableName RelationName RelationOID TimeRange Flags RuleLocks*)

InstanceVariableName is the name used to identify the relation. It may be a class name, a surrogate for a class name, or the symbol new or current. **RelationName** is a string corresponding to the relation's catalog name, **RelationOID** is an integer corresponding to the OID of the RELATION relation tuple that describes the given relation, **TimeRange** is an internal structure that describes the historical time span over which this rangetable entry is valid, **Flags** is a list of zero or more symbols from:

archive inheritance union version

that specifies special treatment from the planner, and **RuleLocks** holds the rule lock information for that relation (since this is generated by a planner preprocessor, the parser simply puts nil here). In the case of archival, inheritance, union and version queries, the planner generates an **APPEND** query plan node (see plan tree documentation) that contains a set of rangetable entries which must be substituted in turn for this rangetable entry, as well as separate query plans to be executed.

1.1.5. Priority

Priority is simply an integer from 0..7 that indicates the query priority. Only rule plans can have

priorities greater than 0.

1.1.6. RuleInfo

If the query is part of a ''define rule'' query, **RuleInfo** is a containing a *rule identifier* and exactly one symbol from:

always once never

The rule identifier is an object identifier from the RULE relation; the symbol is the tag associated with the rule definition. For example, **RuleInfo** might look like:

(387457893 always)

The rule identifier will be filled in by a traffic cop or planner preprocessor routine. Coming out of the parser, it is just a list containing the integer 0. For all other queries, **RuleInfo** is nil.

1.1.7. UniqueFlag

If the query is a ''retrieve'' and the unique tag is present, this will be the symbol **unique**, otherwise it will be nil.

1.1.8. SortClause

If the query is a ''retrieve'' and the sort clause or the unique tag is present, this is a list of three elements: the symbol **sort**, a list of **resdom** nodes (see below) indicating which attributes are to be used to perform the sort, and a list of operator OIDs, one for each attribute used. If the unique tag is present, all attributes in the target list will be used in the sort.

1.2. TargetList

A targetlist describes the structure of a tuple and consists of a list of (*resdom expr*) pairs, one for each attribute in the tuple to be constructed. A **resdom** (result domain) node contains information about the attribute, and an **expr** (expression) describes how to set the value of the attribute.

The initial targetlist from the query tree describes what the final result tuples should look like (targetlists are also used in query plans to describe intermediate results such as scan projections, join results, and temporaries). In the case of the delete command, the initial targetlist is always nil. For the other update queries, the planner's preprocessor fills in any missing attributes that the user has not specified, either with constants corresponding to the default values of the missing attributes.nr $d 1 1 (for ''append'') or with variables corresponding to the unchanged attributes of a tuple (for ''replace'').

The structure of the **resdom** node is described below; an **expr** can consist of a combination of nodes:

expr = *var* | *const* | *param* | (*oper expr*
[*expr*]) | (*func* {*expr*} | *arrayref*)

The structures of the different nodes in an expr are described below, but the function of the nodes may be

---

will have a null value.

briefly summarized as:

var	Refers to some relation attribute entry which corresponds to the target list entry (i.e., the contents of a relation's tuple).
const	Constant values.
param	Used as parameters — placeholders for constants — within a query. The planner treats them as if they are constants of unknown value (i.e., null constants) for optimization purposes.
oper	Describes the system- or user-defined unary or binary operator that will be applied to the argument exprs to produce a new value.
func	Correspond to calls to system- or user-defined functions, which may have zero or more arguments.
arrayref	Refers to an array reference within some array. It contains a slot describing how to produce the array.

Example: ((*resdom var*) (*resdom* (*oper var var*)))
is a targetlist with two attributes, where the second element involves a computation between two tuple attributes.

1.3. Qualification

A qualification consists of restriction and join clauses which are evaluated before the final result is returned.

The qualifications produced by the parser are not normalized in any way. That is, the parsed qualification will be an arbitrary boolean expression:

qualification = ({*qual*})
qual = *expr* | (“**NOT**” *qual*) | (“**OR**” *qual* {*qual*}) | (“**AND**” *qual* {*qual*})

The qualification supplied to the planner must be normalized in certain ways.. rr * A qualification must be specified in conjunctive normal form. Furthermore, if it is at all possible, all ‘‘not’’s are removed and all variables are placed on the left hand side of the clause operator. If it is not possible to remove a ‘‘not’’ (perhaps because a negator is unavailable), then the ‘‘not’’ must be pushed to the innermost comparison, e.g., ‘‘(not ((a = b) and (c = d)))’’ might become ‘‘(not (a = b) or not (c = d))’’ if the operator ‘‘!=’’ does not exist for the appropriate types.

A qualification as seen by the planner is:

qualification = ({*qual* | *orqual*})
qual = *expr* | (“**NOT**” *expr*)
orqual = (“**OR**” *qual qual* {*qual*})

Example: If a user specifies the following clause:

---

not (r.f = 1) or (r.f2 > 1 and 2 > r.f2),

then in conjunctive normal form with ‘‘not’’s removed and variables on the left hand side, we have the following equivalent clause:

(r.f != 1 or r.f2 > 1) and (r.f != 1 or r.f2 < 2)

In LISP form, it would look like:

((or (!= r.f 1) (> r.f2 1)) (or (!= r.f 1) (< r.f2 2)))

2. Primitive Nodes

The following is a description of the primitive nodes, which are used in both the input query tree and the final query plan.

Each type of node is implemented as a defstruct vector. To access slots within a node, use:

(get_*slotname node*)

where *slotname* is the desired slot name as described in this section and *node* is an appropriate defstruct node.

To set a slot field, use:

(set_*slotname node new-slot-value*)

To create a node, use:

(Make*Nodename* {*slot-value-i*})

where *Nodename* is the name of the node with the first letter capitalized, and *slot-value-i* is the value for the i-th slot in the node.

Some of the slot fields in these nodes are not set by the parser. In addition, **fjoin** nodes are not created by the parser at all, while **array** nodes do not appear in query-trees, but they are included here for completeness because they are also primitive nodes.

2.1. RESDOM

1) **resno** - result domain number. This is equivalent to the attribute number for a (temporary or result) tuple. Just as attribute numbers being at 1, a **resno** of 1 denotes the first attribute in a targetlist.
2) **restype** - either:
   (1) the OID corresponding to the type of the result, or
   (2) *UNION-TYPE-ID*, an identifier for a type internal to the planner that describes multi-dot attributes (since it cannot determine the final attribute type until the nesting is eliminated by the executor).
3) **rescomplex** - t if the result type is a complex (i.e. tuple) type
4) **reslen** - length of a domain element in bytes (-1 if it is variable length).
5) **resname** - result domain name, either user-specified or the corresponding name of an attribute (if applicable).
6) **reskey** - ordinality of the result domain as a sort or hash key, e.g., 0 if the domain won't be included as a sort or hash key; 1 if the domain is the first key, 2 if it is the second, and so on.
7) **reskeyop** - OID of the operator on which **reskey** will be sorted or hashed.
8) **resjunk** - t if the attribute is a junk attribute internal to the executor

2.2. VAR

1) **varno** - From the parser's standpoint, **varno** is simply the index into the rangetable corresponding to the relation an attribute belongs to.

   For the query processor, **varno** may take on one of three values depending on the attribute entry under consideration.
   (1) If **VAR** belongs to the relation currently being scanned (either a cataloged or temporary relation, but not a materialized one), **varno** is still the *index into the rangetable* corresponding to the relation being scanned.
   (2) If **VAR** is associated with some join node, varno is the *name of the temporary join relation* accessed. If **VAR** originates from the outer join relation, **varno = "OUTER"**. If it originated from the inner join relation, **varno = "INNER"**.
   (3) For **VAR**s corresponding to entries within relations that are materialized from a POSTQUEL field or attributes from some previous nesting level, **varno** is a *reference pair*, i.e., a list (*levelnum resno*), that corresponds to the attribute entry where either the query field or desired value can be found. *levelnum* refers to the processing level number (where *0* is the topmost), and *resno* refers to the result domain number within the tuple where the entry is stored.

   Therefore, whether a relation should be materialized is implicit in the **varno** value and information in the attribute entry corresponding to the query field.

2) **varattno** - as with **varno**, the meaning of this field depends on the node's position in the plan tree.
   (1) For attributes at the topmost nesting level, **varattno** corresponds to the *attribute number* within a relation (or domain number within a tuple if this is a join tuple — that is, if **varno = "OUTER"** or **"INNER"**).
   (2) If the attribute corresponds to some attribute from a previous nesting level, **varattno** = *nil*.
   (3) For nested attributes, the value of **varattno** is a *string* identical to the attribute name since the attribute number will not be known until the POSTQUEL field is materialized. Note that at the lowest nesting level, **varattno** may be "**ALL**" (again, since neither the parser nor the planner know what the attributes of the field's parent are — it must be determined at run-time by the query processor when the appropriate POSTQUEL field is retrieved).

3) **vartype** - either:
   (1) the OID of the type of the attribute referred to by **varattno**, or
   (2) *UNION-TYPE-ID*

4) **varid** - a list containing **varno** and **varattno**.
   (*This field is internal to the planner*)

5) **varslot** - cached pointer to address of expression context slot.
   (*This field is internal to the executor*)

2.3. OPER

1) **opno** - PG_OPERATOR OID of the operator to which this node corresponds.

2) **opid** - PG_PROC OID of the function to which this operator is attached. This will be determined in the planner.

3) **oprelationlevel** - t if an operator is a relation level operator.

4) **opresulttype** - OID of the result type of this particular operator.

5) **opsize** - size of the return result

6) **op_fcache** - pointer to cached information on the function to which the operator is attached. (*This field is internal to the planner and executor*)

2.4. CONST

1) **consttype** - OID of the constant's type.

2) **constlen** - length in bytes (-1 if it is a variable-length type).

3) **constvalue** - actual value of the constant (i.e., the internal representation). **constvalue** is nil if the constant is null. Objects represented by pointers must appear as palloc'ed LISP vectori-bytes.

4) **constisnull** - t if the constant has the null value. This is set by the planner when filling in ''replace'' and ''append'' targetlist entries which are unspecified by the user. If the typdefault field of the **TYPE** relation is not specified, **constisnull** is set to t and **constvalue** is set to nil.

2.5. PARAM

1) **paramkind** - specifies the kind of parameter. The possible values for this field are specified in ''rules/params.h''. They are: **PARAM_NAMED**: The parameter has a name, i.e. something like '$.salary' or '$.foobar'. In this case field 'paramname' must be a valid Name. **PARAM_NUM**: The parameter has only a numeric identifier, i.e. something like '$1', '$2' etc. The number is contained in the 'paramid' field. **PARAM_NEW**: Used in an instance level rule, similar to PARAM_NAMED. The 'paramname' and 'paramid' refer to the ''NEW'' tuple. The 'paramname' is the attribute name and 'paramid' is the attribute number. PARAM_OLD: Same as PARAM_NEW, but in this case we refer to the ''OLD'' tuple.

2) **paramid** - numeric identifier for literal constant parameters (''$1'').

3) **paramname** - attribute name for tuple substitution parameters (''$.foo'').

4) **paramtype** - OID of the type of the parameter

5) **param_tlist** - a *targetlist* describing the structure of the parameter as if it were the target of a query. This allows for projection in a param node.

2.6. FUNC

1) **funcid** - OID of the function to which this node corresponds.

2) **functype** - OID of the type of the function's return value.

3) **funcisindex** - can an index be used to evaluate this function? (*In the present system only operators can be evaluated using indices*)

4) **funcsize** - size of the function's return result

5) **func_fcache** - pointer to cached information on the function

6) **func_tlist** - a *targetlist* describing the function's return value

7) **func_planlist** - list of plan trees generated by planning the function, if it's a POSTQUEL function. (*This field is internal to the planner and executor*)

2.7. ARRAYREF

1) **refattrlength** - length of array being referred to, or -1 if it's a variable length array

2) **refelemlength** - length in bytes of array element

3 **refelemtype** - OID of type of array element

4 **refelembyval** - true if the element type can be passed by value

5 **upperindexpr** - expression evaluating to upper array index

6 **lowerindexpr** - expression evaluating to lower array index

7 **refexpr** - expression evaluating to the array being referred to

8 **refassgnexpr** - expression evaluating to the new value to be assigned to the array in a **REPLACE** query

2.8. ARRAY

An **ARRAY** node describes the structure of an array type. It appears in the tree for **CREATE** and **ADD_ATTR** commands.

1) **arrayelemtype** - OID of type of array element

2) **arrayelemlength** - length in bytes of array element

3) **arrayelembyval** - true if the array element can be passed by value

4) **arrayndim** - number of dimensions of the array

5) **arraylow** - base for array indexing

6) **arrayhigh** - limit for array indexing

7) **arraylen** - length of entire array (*not used*)

To create an **ARRAY** node:

(MakeArray *arrayelemtype arrayelemlength arrayelembyval arrayndim arraylow arrayhigh arraylen*)

2.9. FJOIN

An **FJOIN** node describes how to flatten a target list after executing a join. This node is internal to the planner and executor. A list will be produced containing an **FJOIN** node for each node included in the join.

1) **fj_initialized** - true if the node has been initialized for the current target list evaluation

2) **fj_nnodes** - number of nodes to flatten
3) **fj_innerNode** - one node to flatten
4) **fj_results** - the complete flattened join result
5) **fl_alwaysDone** - true if the node never produces any results, and therefore never needs to be reprocessed