QUERY TREE / QUERY PLAN SPECIFICATION
(Printed: July 30, 1990)

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*)

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 a string representing the result relation’s name and a string from:

‘‘HEAVY’’ ‘‘LIGHT’’ ‘‘NONE’’

which represents the level of archiving which has been specified for the result relation.

(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:

(*RelationName RelationOID TimeRange Flags RuleLocks*)

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 below) 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. For all other queries, **RuleInfo** is nil.

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[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*})

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

var — Refer 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.

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.[2] 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,

[1] If no default value exists for an attribute type, the attribute will have a null value.

[2] This normalization occurs in yet another preprocessing module.

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)))

A set of macros for manipulating clause entries is contained in:

> ˜postgres/src/planner/clauses.l

Additional parse tree manipulation routines are in:

> ˜postgres/src/planner/nodeDefs.l

2. Query Plan Representation

A query plan is represented as a node. Every plan node contains the following slots:

> **state**
> **qptargetlist**
> **lefttree**
> **righttree**

Thus, while a query plan is externally a single object, it may in fact constitute a tree of plan nodes because of the child tree fields.

A *plan* consists of *subplans* interconnected together with **RESULT**, **EXISTENTIAL**, and **APPEND** nodes. These nodes may also have each other as children.

RESULT nodes interconnect plan levels. The planner generates subqueries for all variables which operate at a given level of nesting (that is, all non-nested variables are handled in one plan level, all variables of the form ‘‘foo.bar.baz’’ are handled in another, and so on). Thus, **RESULT** nodes indicate which attributes should be selected from lower levels in the plan tree. If a **RESULT** node has subtrees, then the left tree always is a *subplan*, and the right tree is a *plan* for processing the (more deeply nested) remainder of the query. It is also possible for a **RESULT** node to have only a single left subtree. This occurs when the **RESULT** node is needed solely for storing relation-level clauses. A query plan can be a single **RESULT** node with no subtrees if the query contains no variables.

EXISTENTIAL nodes also connect subplans. The left subtree of an **EXISTENTIAL** node is a special subplan which determines whether the existential members of the subplan qualification are all true or not. The right subtree is the actual query plan for the subquery (minus the existential qualification clauses). **EXISTENTIAL** nodes may also have one (left) or zero subtrees.

APPEND nodes indicate that the executor should use the subplans it contains sequentially, returning the appropriate tuples in a seamless manner (at least as far as the external interface is concerned). **APPEND** nodes may be thought of as taking the place of N scan nodes, each of which are substituted for the **APPEND** node in turn.

A *subplan* is an access plan for processing the (current) topmost level of attributes in a query and consists of join and scan nodes.

Joins are represented such that the left tree is the outer join relation and the right tree is the inner join relation. The three types of join nodes are:

> **NESTLOOP**
> **MERGESORT**
> **HASHJOIN**

The two types of scan nodes are:

> **SEQSCAN**
> **INDEXSCAN**

Lastly, there are two types of nodes which involve the creation of temporary relations:

> **SORT**
> **HASH**

If a stream of tuples must be sorted into a temporary prior to join processing, its parent is a **SORT** node and the parent of the **SORT** node is a **SEQSCAN** node:

<some-join> **<—— SEQSCAN <—— SORT <——** <some-scan>

Similarly, if the relation is to be hashed, its corresponding scan node's parent is a **HASH** node.

<some-join> **<—— SEQSCAN <—— HASH <——** <some-scan>

Example: A ‘‘retrieve’’ involving a single mergesort join that uses an index as an ordered inner path and a sorted outer path might look like:

MERGE <—— SEQSCAN <—— SORT <—— SEQSCAN
SORT <—— INDEXSCAN

The same idea applies to sorts in general. The node is simply preceded by a **SORT** and then a **SEQSCAN** node in the plan tree. However, if a retrieve result is to be sorted into a relation, the **SEQSCAN** node is omitted.

Example: The top level nodes for a plan that returns a set of tuples that must be sorted into a particular order might look like:

RESULT <—— SORT <—— ...rest of plan...

3. 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*)

For details see

˜postgres/src/planner/nodeDefs.l

3.1. 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) **reslen -** length of a domain element in bytes (-1 if it is variable length).

4) **resname -** result domain name, either user-specified or the corresponding name of an attribute (if applicable).

5) **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.

6) **reskeyop -** OID of the operator on which **reskey** will be sorted or hashed.

To create a **RESDOM** node:

(make_resdom *resno restype reslen resname reskey reskeyop*)

3.2. 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) **vardotfields -** list of attribute names beyond the first nesting level.
(*This field is internal to the parser and planner*)

5) **vararrayindex -** array index of a variable, nil if this is not an array attribute.
Since arrays can only contain primitive types, only the bottommost attribute in a nesting will be array indexed. The parser will set the **vararrayindex** field if a variable contains an array index. The planner will only set this field if the query processor is to retrieve an array element within some attribute entry.

6) **varid -** a list containing **varno** and **varattno**, as well as **vardotfields** and **vararrayindex** (if the latter two exist).
(*This field is internal to the planner*)

Example: w.x.y.z[a]

The parser will set the following fields:

varno = identifier of w, **varattno** = attribute number of x,
vardotfields = (y z), **vararrayindex** = a,
varid = (**varno varattno** y z (a)),
vartype = type of attribute x

To create a **VAR** node:

(make_var *varno varattno vartype vardotfields vararrayindex varid*)

3.3. OPER

1) **opno -** for the parser, **opno** is the OID of the operator to which this node corresponds. For the executor, **opno** is the OID of the operator *procedure*. The conversion occurs in the planner (which needs the operator OID to determine selectivities).
2) **oprelationlevel -** t if an operator is a relation level operator.
3) **opresulttype -** OID of the result type of this particular operator.

To create a **OPER** node:

(make_oper *opno oprelationlevel opresulttype*)

3.4. 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.

To create a **CONST** node:

(make_const *consttype constlen constvalue constisnull*)

3.5. PARAM

1) **paramid -** either:
   (1) a fixnum corresponding to the parameter name for constant value substitution, e.g., ''1'' in the case of ''$1''
   (2) a string corresponding to the attribute name for relation name substitution, e.g, ''attname'' in the case of ''$.attname''

To create a **PARAM** node:

(make_param *paramid*)

3.6. FUNC

1) **funcid -** OID of the function to which this node corresponds.
2) **functype -** OID of the type of the function's return value.

To create a **FUNC** node:

(make_func *funcid functype*)

4. Plan Nodes

The following is a description of each type of plan node. 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*)

For details see

˜postgres/src/planner/nodeDefs.l

4.1. RESULT

1) **state -** used by the query executor to store runtime-specific information.
2) **qptargetlist -** the target list for a sublevel of the query; the attributes in node **RESULT***(i)* are derived from values returned by *subplan(i)*, and either **RESULT***(i+1)* or *subplan(i+1)*, depending on whether or not there is an i+1'th nesting level.
3) **lefttree -** *subplan(i)*.
4) **righttree -** either **RESULT***(i+1)* or *subplan(i+1)*, depending on whether or not there are deeper nesting levels.
5) **resrellevelqual -** list of any relation level clauses that must be satisfied (e.g., ‘‘R1.f1 = R2.f2’’ or ‘‘foo.bar = 2’’)
6) **resconstantqual -** a list containing all qualifications which contain no var nodes.
   This field will only be set at the topmost **RESULT** node. These qualifications should always be tested first at runtime; if they are false, then the rest of the query plan need not be processed because no tuples will be returned.

   To create a **RESULT** node:

   (make_result *qptargetlist resrellevelqual lefttree righttree*)

4.2. EXISTENTIAL

1) **state**
2) **qptargetlist -** always nil.
3) **lefttree** - subplan for the existential query.
   This is planned as a ‘‘retrieve’’ with a null target list.
4) **righttree -** subplan for the non-existential query.

To create a **EXISTENTIAL** node:

(make_existential *lefttree righttree*)

4.3. APPEND

1) **state**

2) - 4) **qptargetlist**, **lefttree**, **righttree** - always nil

5) **plans** - a list of plans which are to be executed sequentially.

6) **relid** - the rangetable index of the variable which caused this node to be formed. For example, if this is a query that uses the **from** clause ‘‘from p in person*’’, then this would contain the rangetable index of the entry for ‘‘person’’.

7) **rtentries** - a list of rangetable entries which must be substituted into the executor’s rangetable in conjunction with execution of the plans stored in the **plans** field.

8) **flags** - a symbol from **archive**, **inheritance**, **union**, or **version** that specifies how the executor should execute each entry in the **plans**.

To create a **APPEND** node:

(make_append *plans relid rtentries flags*)

4.4. NESTLOOP

1) **state**

2) **qptargetlist** - attributes to be retrieved into the join result from attributes in the inner and outer join relations.

3) **lefttree** - outer join path.

4) **righttree** - inner join path.

5) **qpqual** - a list of join clauses to be satisfied.

To create a **NESTLOOP** node:

(make_nestloop *qptargetlist qpqual lefttree righttree*)

4.5. HASHJOIN

1) - 5) from above

6) **mergehashclauses** - join clauses to be used in performing the hash join; if there are multiple clauses, the clauses are arranged in the order of the hash keys, from primary downward.
Note: These clauses are not contained in **qpqual** above to avoid redundant checking.

To create a **HASHJOIN** node:

(make_hashjoin *qptargetlist qpqual mergehashclauses lefttree righttree*)

4.6. MERGESORT

1) - 5) from above

6) **mergehashclauses** - join clauses to be used in performing the merge join; if there are multiple clauses, the clauses are arranged in the order of the merge keys, from the primary ordering key

downward.
(Again, these clauses are not contained in **qpqual**)

7) **mergesortop -** OID of the operator procedure used to merge the tuples.

To create a **MERGESORT** node:

(make_mergesort *qptargetlist qpqual mergehashclauses mergesortop lefttree righttree*)

4.7. SEQSCAN

1) **state**
2) **qptargetlist -** attributes to be retrieved into scan result.
3) **lefttree -** either nil, a **SORT** node, or a **HASH** node.
4) **righttree -** always nil.
5) **qpqual -** list of restriction clauses on the relation.
6) **scanrelid** - either:
(1) an index into the rangetable corresponding to the relation to be scanned, or
(2) a reference to an attribute from a previous level that corresponds to a relation to be materialized from some POSTQUEL field.

To create a **SEQSCAN** node:

(make_seqscan *qptargetlist qpqual scanrelid lefttree*)

4.8. INDEXSCAN

1) - 2) from above

3) - 4) **lefttree**, **righttree -** always nil.

5) - 6) from above

7) **indxid -** list of the OIDs of all indices to be used to scan a relation: (OID1 OID2 ...). The idea here is that more than one index may be used to scan a relation. For example, if we have the clause ‘‘R.foo = foo or R.foo = foobar’’, the first index might use ‘‘foo’’ as a key and the second could use ‘‘foobar’’. Currently, this tactic is only used as just described (in conjunction with ‘‘or’’ clauses).

8) **indxqual -** list of qualifications to be used in conjunction with each index (e.g., ((qual1) (qual2)), where qual1 corresponds to ID1 and qual2 corresponds to ID2). For multi-key indices, subclauses are arranged in the order of the index keys.
Note, again, that the clauses here are not contained in the **qpqual** field above to avoid redundant checking.

   If **indxqual** is nil, but **indxid** isn’t, then the entire relation is to be scanned starting at the first key of the index. This is used when the ordering of an indexscan is needed for a mergesort.

   If an index is defined on an inner join relation, **indxqual** may be a join clause. If this is the case, the query processor must substitute values for the outer relation at runtime. The outer join attribute is identified by its relative position of within the resulting tuple of the outer join relation. This type of qualification is also omitted from **qpqual** above to avoid redundant checking.

To create a **INDEXSCAN** node:

(make_indexscan *qptargetlist qpqual scanrelid indxid indxqual*)

4.9. SORT

1) **state**
2) **qptargetlist -** targetlist for the tuples which are to be placed into the temporary relation.
3) **lefttree -** plan node whose result is to be sorted.
4) **righttree -** always nil.
5) **tempid -** either:
   (1) a string corresponding to a user-specified result relation, or
   (2) *TEMP-RELATION-ID* (a special identifier indicating a temporary relation).
6) **keycount -** the number of sort keys.

To create a **SORT** node:

(make_sort *qptargetlist tempid lefttree*)

4.10. HASH

1) - 6) from above

To create a **HASH** node:

(make_hash *qptargetlist tempid lefttree*)