***Run Time Executor - Implementation Notes***

by
Chin-Heng Hong
University of California, Berkeley

(Printed October 29, 1990)

## 1. Introduction

This document gives an overview of the *Run Time Executor (RTE)* and its implementation details. It shows where and how the RTE fits into the POSTGRES data base system, and the data structures used by it. The algorithms used to process the nodes in the query plan are also presented. It is hoped that this document will allow someone to understand the implementation details of the RTE without having to go through the painful process of deciphering the actual codes. This document describes the first prototype of the RTE running in August 1986. It should be updated to reflect any changes made to the RTE since then so that it will remain helpful and accurate.

## 2. An Overview

The Run Time Executor (RTE) is the module in the POSTGRES that executes the query plan produced by *Query Optimizer*. It accepts the query plan from the *Traffic Cop*, initializes it, retrieves and processed tuples according to the query plan, and then returns a stream of tuples to the Traffic Cop. During the execution of the query plan, the RTE makes calls to the *Access Method* to retrieve tuples from relations, to the *Tuple Rule Manager* to process record level rules, and to the *Function Manager* to evaluate expressions. It may also call some of the utility routines to create or sort a relation. Figure 1 shows the interactions of the RTE with other modules in the POSTGRES.

### 2.1. Call Interface

The RTE accepts the following list from the Traffic Cop[1]:

(*commandType queryFeatures inputTree queryPlan queryState*)
where

| | |
|---|---|
| *commandType* | = (retrieve/append/update resultRelation pipe) |
| *queryFeature* | = (tupleCount owner time direction) |
| *inputTree* | = not used by the RTE |
| *queryPlan* | = the query plan |
| *queryState* | = (initFlag xactID subPlanTuples) |

The first element of the *commandType* is a string of "RETRIEVE", or "APPEND" or "UPDATE" to indicate the type of command. *ResultRelation* is the relation ID of a relation where the tuples retrieved will go to. If the tuples are to be passed back to a front-end, *pipe* will contain a socket ID.

The *tupleCountR in queryFeature* is the number of tuples that is needed by the front-end. In the case that all the tuples are to be processed, it is set to 0. The other fields are parameters needed by the AMI.

[1] Not well defined yet.

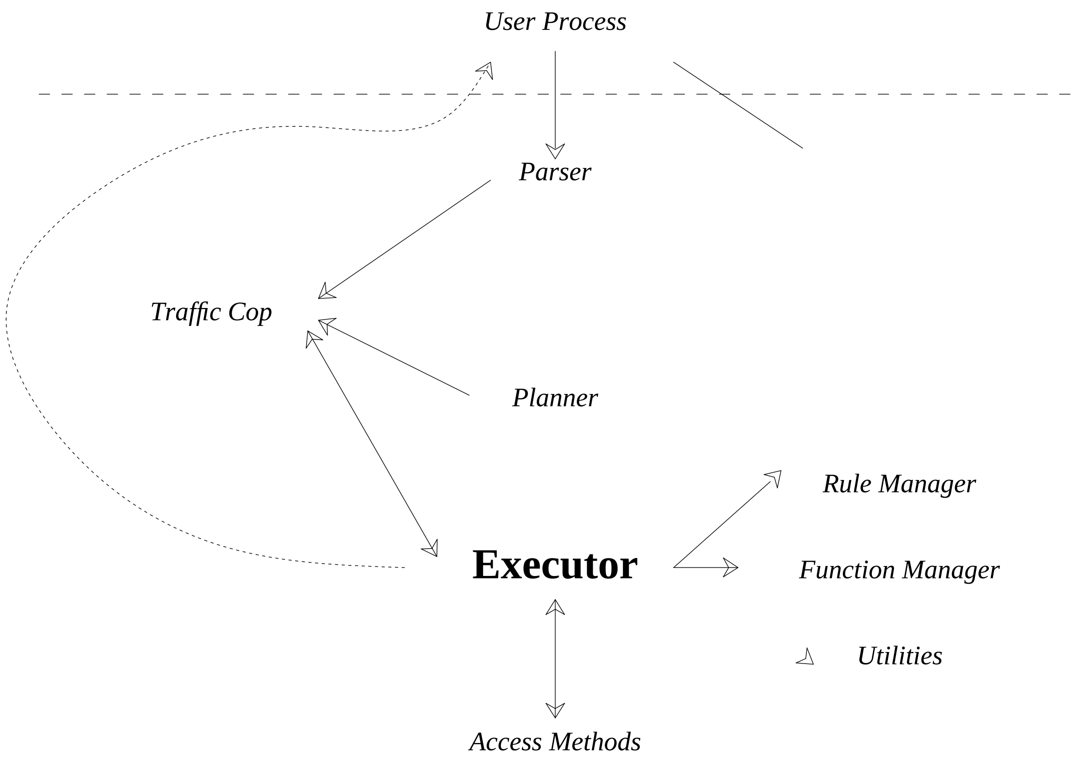


Figure 1. The Run Time Executor and Other POSTGRES modules

The *queryState* contains some global state information used by the RTE. Other modules should not try to manipulate it. It should be set to a empty list [(nil)] when the query plan is first constructed, and RTE will initialize it with necessary information.

**2.2. Control Flow**

The main routine of the Run Time Executor (RTE) accepts the parameter list from the Traffic Cop and processes the query plan to retrieve the required number of tuples. The tuples returned from the processing of the plan are treated differently for different commands:

(1) retrieve — The tuples are either passed to the front-end through a pipe or inserted to the result relation indicated. Tuples are passed to the front-end as a formated stream of bytes[2].

(2) append — The tuples are simply inserted to the result relation.

(3) update — The tuples are inserted to the result relation, and the corresponding old tuples are deleted.

The main routine of the RTE is ExecMain():

[2]Refer to the document on the front-end library -- pqlib.

```
ExecMain((commandType queryFeature inputTree queryPlan queryState))
{
   initialize query plan if called first time;
   while (moreTuples) {
           /* process query plan */
           tuple = ExecProcNode(queryPlan);
           if (tuple = null) {
              do necessary cleaning up;
              return;
           } else {
              if (retrieveCommand and returnToFrontEnd) {
                      format and pass tuple down the pipe;
              } else {
                      update result relation;
              }
           }
           numberTuples++;
           if (numberTuples == tupleCount) {
              do necessary cleaning up;
              return;
           }
   }
}
```

## 3. Module Organization

The source files for the Run Time Executor (RTE) are organized as follow:

| | |
|---|---|
| execInt.h | main internal header file, contains global definitions. |
| execMain.l | main routine. |
| exec*node*.l | routines to process the *node*. |
| exec*node*.h | header file for the *node*. |
| execProcNode.l | routine to call the right *node* routine. |
| execInit.l | routine to initialize query plan. |
| execQual.l | routines to evaluate the qualification. |
| execTargetList.l | routines to evaluate the target list. |
| execPipe.l,c | routines to communicate with the front-end. |
| execCStruct.c | C routines handling C structures. |
| execCStruct.l | lisp code to load in C routines in execCStruct.c. |

The hierarchical relationship of the routines in RTE module is shown in Figure 2.

## 4. Data Structures

The Run Time Executor (RTE) keeps some state information in the query plan so that it can pick up the query plan at any point and continue from where it has left off. This allows the the *front-end* to retrieve some number of tuples into a *portal*, process them, and then ask for more tuples using to the same query plan.

The RTE needs to *keep track of the tuples formed at different subplans*. This is because the query plan is composed of one or more subplans as shown in Figure 3. Nodes in lower subplans may reference

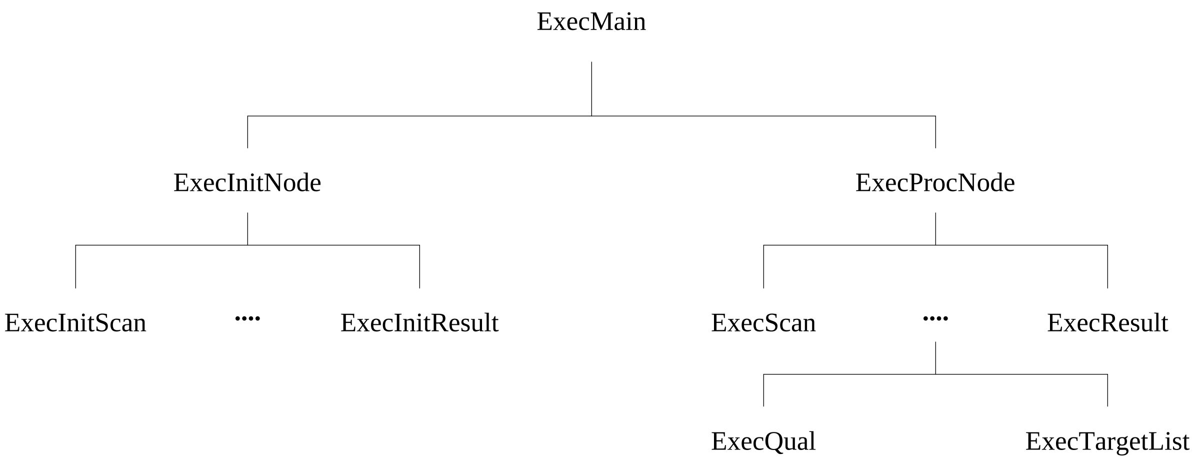


Figure 2. Hierarchical Relaionship of RTE Routines

the tuples formed at higher level of subplans. Therefore, an array of containing tuples formed at each subplan is maintained. This allows fast access to these tuples without the need of traversing the query plan to find the particular subplan interested. This structure is stored in the state variable *subPlanTuples*.

Also, some of the *global variables* are defined so that other routines can readily accessed some of the fields from the parameter list passed to the RTE from the Traffic Cop. These global variables are *execDirection*, *execTime*, *execOwner* and *execSPlanInfo*. The suffixes of these variables indicates the fileds they represent. The global variable, *execErrorMssg*, is set by the error handling routine to indicate an error condition. Another global variable (*execReplTID*) is used only for *replace command*. execReplTID is set by a SCAN node to keep track of the *TID* (tuple ID) of the tuple passed to higher nodes for modifications. An update is done at the highest level by inserting the modified tuple and then deleting the old tuple. The highest level routine finds the TID of the old tuple from execReplTID.

Besides the global structures and variables, the Query Optimizer also reserves a slot in every nodes in the query plan for the RTE to store state information local to each node. The RTE then initializes the slot reserved to contain the necessary state variables during the initialization phase[3]. These state variables are updated during the execution of the query plan. For example, the SCAN node contains a flag to indicate whether the Tuple Rule Manager has been activated. The flag is set and cleared appropriately to reflect the state of the SCAN node.

## 5. Node Processing

The execution of the query plan is very simple. The RTE starts at the root of the query plan and calls the right and left subtrees for tuples. The subtrees recursively calls their subtrees for tuples and process the tuples according to the type of the node. Therefore, there is one principal routine per node type, and the corresponding routine for a *node* is Exec*node*(). The general format of such a principal routine is a follow:

[3]The RTE initializes the query plan once prior to actual processing. Each node has its own initialization routine because different node needs different state information.

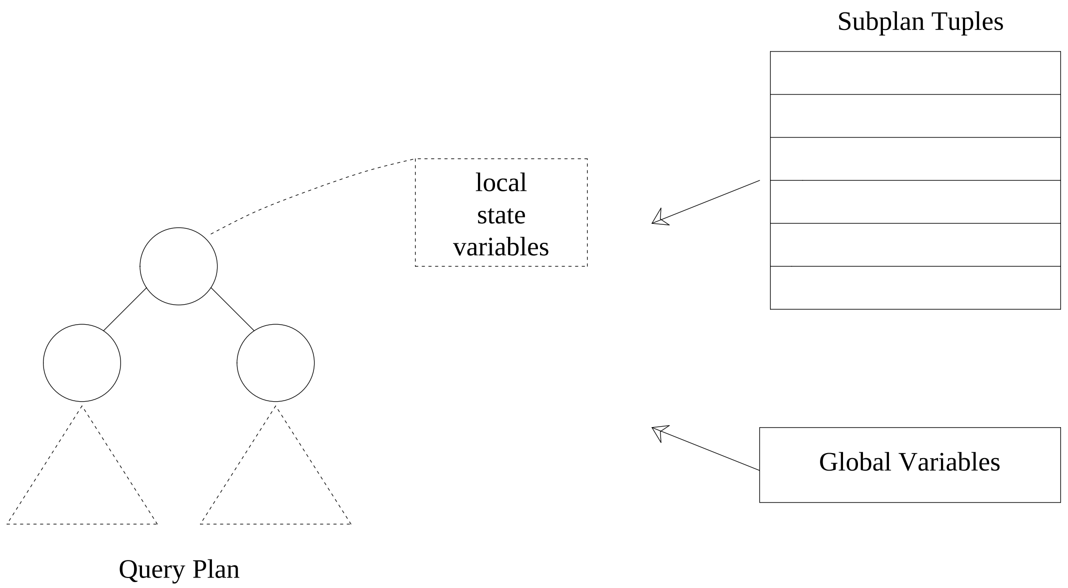


Figure 3. Data Structures Maintained by the RTE

```
Execnode(node)
{
   leftTuple = ExecProcNode⁴(GetLeftTree(node));
   rightTuple = ExecProcNode(GetRightTree(node));
   do processing specific to the node;
   if (sucessful) {
           return(tuple formed according to target list);
   } else {
           return(Execnode(node));
   }
}
```

### 5.1. Sequential and Index Scans

Sequntial and index scans are very similar. The main difference is that they use different access methods; the access method for the index scan retrieve tuples using some indices while sequntial scan does so sequentially calling the AMI's getnext(). Once a tuple is retrieved, the processing is the same for both type of scans. Therefore, a general scan routine is written:

ExecScan(node, accessMethod, direction, lock)

[4] ExecProcNode() in the above code segment is a routine that checks the type of a node, and calls the corresponding principal routine to process it.

This general routine call the *accessMethod()* to retrieve tuples and then process the tuples retrieved independent of the type of scan. We will first examine the ExecScan(), and then the accessMethod() for both the sequential and index scans.

The main task of ExecScan() is calling the Tuple Rule Manager when necessary. Whenever a tuple is retrieved, it is checked for locks set by tuple level rules (*ExecRuleLocks()*). if such lock is found, the tuple is passed to the Tuple Rule Manager and the *rule descriptor* returned is kept. The rule descriptor is used to retrieve tuples resulted from the set of rules activated for the original tuple. The set of rules activated may produce 0 or more tuples. A flag is set to indicate that some rules have been activated and that the Tuple Rule Manager should be called to retrieve such tuples. This flag is kept as part of the state information associated with the node so that it can be examined during subsequent calls.

Of course, the tuple retrieved from a relation or from the Tuple Rule Manager is always checked against the *qualification* associated with the node. If the tuple satisfies the qualification, then a new tuple formed from it according to the *target list* is returned. Otherwise, the whole process is repeated to find a qualifying tuple. In fact, checking a tuple against the qualification (*ExecQual()*) and forming a new tuple according to the target list (*ExecTargetList()*) is universal to all nodes. These 2 operations will be examined in more details in the next section.

If the query is an update operation and the currently scanned tuple is to be eventually modified, the *updateFlag* in the SCAN node is set by the Query Optimizer. In this case, the TID of the tuple is stored in the global variable *execReplTID* so that higher level nodes can access it when doing the update.

Another interesting point is that tuples can be retrieved in either forward or backward directions. The direction of scanning can be changed at any point and is indicated by the global variable *execDirection*. **The AMI supports scanning in both directions, and it is assumed that the Tuple Rule Manager provides similar support**. This means that the Tuple Rule Manager must maintain the order of tuple produced from a set of rules.

```
ExecScan(node, accessMethod)
{
   if (rulesActivated) {
          /* get tuple from Tuple Rule Manager */
          tuple = RuleMgrGetTuple(ruleDescriptor);
          if (tuple = nil) {
             ruleActivated = false
             return(ExecScan(node, accessMethod);
          }
   } else {
          /* use the access method to get tuple from relation */
          tuple = accessMethod(node);
          if (tuple = nil) return(nil);  /* no more tuple */
          if (ExecRuleLocks(tuple)) {
             /* process tuple level rules */
             rulesActivated = true;
             ruleDescriptor = RuleManagerGetDesc(tuple);
             return(ExecScan(node, accessMethod));
          }
   }
   /* check tuple against qualification */
   if (ExecQual(qual, tuple)) {
          /* keep track of TID if the tuple is to be updated */
          if (updateFlag = true) {
             execReplTID = getTID(tuple);
          }
          return(ExecTargetList(targetList tuple);
   } else {
          return(ExecScan(node, accessMethod));
   }
}
```

#### 5.1.1. Sequential Scan

The ExecSeqScan() scans a relation sequntially calling ExecScan() and passing the AMI's getnext() as the access method. One subtle point about ExecSeqScan() is that it may need to sort or hash a relation before scanning. This is indicated by the presence of a left subtree (HASH or SORT node). If the left subtree is presence, it is processed first and the resulting relation is then scanned by the ExecSeqScan().

#### 5.1.2. Index Scan

An access method (*IndexNext()*) is written to use a list of indices to retrieve tuples from a relation. The list of indices is constructed by the Query Optimizer, and it may contain both *primary* and *secondary indices*. The indices is used one by one according to their order in the index list. Secondary indices are different from primary indices in that one more level of indirection is involved. Secondary indices provide the *TID* of a tuple, and the tuple is then retrieved directly from a relation using the AMI's getunique().

When an INDEX node is initialized, the index relations are opened for scanning, and the index qualifications associated with each index are used to restrict to scanning[5]. It is possible that index qualifications contain join clauses. This happens when an index is defined on an inner join relation. In this case, the values from the outer tuple has to be substituted into the join clauses for each outer tuple. **Currently, this feature is not supported by the RTE**[6].

```
IndexNext(node)
{
   currentIndex = GetIndexId(indices);
   case (currentIndex) {
           nil:              return(nil);        /* no index to use */
           PRIMARY:          tuple = getnext(currentIndex);
                             if (tuple <> nil) return(tuple);
           SECONDARY:        Tid = getnext(indexRelation);
                             if (Tid <> nil)
                                return(getunique(Tid);
   }
   /* no tuple from current index, use next index */
   advance index pointer to point to next index;
   return(IndexNext(node));
}
```

### 5.2. Sort

As mentioned earlier, a relation may need to be sorted prior to scanning. The relation to be sorted is scanned calling the left subtree of the SORT node. The keys used to sort the relation is found in the target list. More than one keys may be used to sort the relation and their relative order of significance is found in the *RESDOM* primitive node prepared by the Query Optimizer.

The *ExecSort()* calls the left subtree for tuples and put them in a temporary relation. It then calls the sort utility routine to sort the temporary relation. The ID of the sorted relation is returned to upper level node. The parameter to the sort utility routine is a vector containing entries of (key, sort operator). This vector parameter is prepared by *ExecInitSort()* by scanning the target list. The most significant key appears first in the vector parameter.

[5] The SKEY structures are formed from the qualifications and is used by the AMI’s getnext to restrict scanning. See OpenScanIndices() and CreateSkeys() in execIndexScan.l.

Need to change the constants in the skey structures and do a "beginscan" with the modified skeys.

```
ExecSort(node)
{
   create temporary relation;
   While ((tuple = ExecProcNode(GetLeftTree(node))) <> nil)
         insert(tempRelation, tuple);
   }
   sortUtility(tempRelation, keys);
   return(sorted relation's ID);
}
```

### 5.3. Nested-Loop Join

Nested-loop join is done by iteratively retrieving a tuple from the outer relation (left subtree), joining it with all the tuples from the inner relation (right subtree). The inner relation is scanned once for each outer tuple. A new outer tuple is retrieved if there is no inner tuple satisfying the join clauses with the current tuple or a complete scan of the inner relation is done. The join can be done in either forward or backward direction at any point. No special code is needed to support this feature because the subtrees return the outer and inner tuples in the direction specified by the global variable *execDirection*. As a result, the join will invariably be done in the direction desired.

```
ExecNestLoop(node)
{
   innerTuple = ExecProcNode(GetRightTree(node));
   if (innerTuple = nil) {
         /* complete scan of inner relation, get a new outer tuple */
         outerTuple = ExecProcNode(GetLeftTree(node));
         if (outerTuple = nil) return(nil);
         ExecBeginScan(GetRightTree(node));
     innerTuple = ExecProcNode(GetRightTree(node));
   }
   /* join the inner and outer tuples */
   if (ExecQual(qual, outerTuple, innerTuple)) {
         return(ExecTargetList(targetList, outerTuple, innerTuple));
   } else {
         return(ExecNestLoop(node));
   }
}
```

### 5.4. Merge-Join

Merge-join is used if the outer and inner relations are sorted in ascending or descending order using the merge-sort operator in the OPERATOR relation. **The user must specify "<" or ">" as merge-sort operator for ascending and descending orders respectively**[7]. Merge-join is done by joining the inner

[7]For performance reason, "<=" and ">=" are used as merge-sort operators because a "<" comparison would then have to be turned into "<=" and "not =" comparisons.

and outer tuples satisfying the join clauses of the form ((outerKey = innerKey) ...). The join clauses is provided by the query planner and may contain more than one (outerKey = innerKey) clauses.

However, the query executor needs to know whether an outer tuple is "greater/smaller" than an inner tuple so that it can "synchronize" the two relations. For example, consider the following relations:

```
outer: (0 ^1 1 2 5 5 5 6 6 7)      current tuple: 1
inner: (1 ^3 5 5 5 5 6)            current tuple: 3
```

To continue the merge-join, the executor needs to scan both the inner and outer relations until the matching tuples "5". It needs to know that currently inner tuple "3" is "greater" than outer tuple "1" and therefore it should scan the outer relation first to find a matching tuple and so on.

Therefore, when initializing the merge-join node, the executor creates the "smaller/greater" clause by substituting the "=" operator in the join clauses with the merge-sort operator to form (outerKey sortOp innerKey) clauses. The sort operator is "<" if the relations are in ascending order; otherwise, it is ">" if the relations are in descending order. The opposite "greater/smaller" clause is formed by reversing the outer and inner keys to form (innerKey sortOp outerKey) clauses.

It is sometimes necessary to reposition the "cursor" of inner relation to do merge-join. Take the above relations for example, after joining the inner "5's" with the first outer "5", we need to reposition the inner "cursor" at the first inner "5" again to join with the second outer "5". **Currently, this is done by scanning the inner relation in the reverse direction**. A more efficient method is to make use of the AMI routine to mark and reposition the "cursor" at the appropriate point in the inner relation.

The main routine for merge-join is *ExecMergeJoin()*. It first tries to join the next inner tuple with the current outer tuple, if that fails, it then call *MergeSync()* to synchronize the relations to next matching tuples. The matching outer and inner tuples are then checked against the other qualification[8].

```
ExecMergeJoin(node)
{
   innerTuple = ExecProcNode(GetRightTree(node));
   if (innerTuple <> nil and ExecQual(joinQual, outerTuple, innerTuple)) {
           /* resynchronizes relations. Returns inner tuple.
              Matching outer tuple is returned as one of the state
              variable of the node.
           */
           innerTuple = MergeSync(node);
   }
   if (innerTuple = nil) then return(nil)
   else if (ExecQual(qual, outerTuple, innerTuple))
       /* matching tuples satisfy qualification */
       return(ExecTargetList(targetList, outerTuple, innerTuple);
   else
           return(ExecMergeJoin(node));
}
```

[8]The qualification for the node is separated into one used for doing the merge-join and another that contains the rest of the clauses. The reason is that only certain attributes can be used to do merge-join.

### 5.5. Subplans Interconnection

Subplans are interconnected by RESULT nodes. The subplan on the left of the RESULT node is of higher level than those on the right. An implicit nested-loop join is done at this node: for every tuple retrieved from the left subtree, it is joined with tuples retrieved from the right subtree. The reason is that the subplans of the right subtree may reference and materialize fields from the tuple produced by the higher subplan of the left subtree; therefore, for a tuple from the left subtree, there may be zero or more tuples resulted from the right subtree. If there is no right subtree, only tuples from the right subtree is scanned and returned.

The tuple produced by the left subtree is kept in a global structure so that it can be referenced easily by lower subplans. The routine *ExecSetSPlanTuple(levelNo, tuple)* is called to set the tuple produced by subplan of level "levelNo" in the global structure. Tuple produced by subplans can then be referenced by calling *ExecGetSPlanTuple(levelNo)*. To get and set the corresponding type information on the tuple, call *ExecGetSPlanType(levelNo)* and *ExecSPlanType(levelNo, type)* respectively.

The tuples from left and right subtree are checked against the qualifications associated with the node. If the tuples satisfy the qualifications, a tuple formed according to the target list is returned.

```
    ExecResult(node)
    {
       /* get next tuple from left subtree if necessary */
       if ((leftTuple = nil) or (no right subtree) or
               ((rightTuple = ExecProcNode(GetRightTree(node))) = nil))
       {
               leftTuple = ExecProcNode(GetLeftTree(node));
               if (leftTuple = nil) return(nil);
               else {
                  ExecSetSPlanTuple(levelNo, leftTuple);
                  rightTuple = ExecProcNode(GetRightTree(node));
                  if (rightTuple = nil) return(ExecResult(node));
               }
       }
       /* check qualifications */
       if (ExecQual(qual, leftTuple, rightTuple)
               return(ExecTargetList(targetList leftTuple rightTuple));
       else
               return(ExecResult(node));
}
```

## 6. Qualification and Target List

The routines *ExecQual()* and *ExecTargetList()* are very important modules of the Run Time Executor (RTE). ExecQual() checks tuples against a qualification clause, and it returns a boolean value as the result of the test. ExecTargetList(), on the other hand, constructs a new tuple from some tuples according to a target list. Both routines have to call the Function Manager to evaluate expressions and certain C routines to manipulate tuple fields.

### 6.1. ExecQual

*ExecQual()* checks the currently scanned tuple (Scan nodes) or the inner/outer tuples (Join nodes) against the qualification associated with a node. A qualification is expressed in conjunctive normal form. It consists of zero or more or-clauses "anded" together. The syntax of the qualification is as follow[9]:

```
qualification   :=  {expr}
expr            :=  ("OR" expr expr {expr}) | ("NOT" expr)
                    | (op expr expr) | (func {expr})
                    | var | const | param
orExpr          :=  ("OR" expr expr {expr}) | expr
```

The module for evaluating the qualification falls nicely into 3 main routines: ExecQual(), ExecEvalOr, and *ExecExpr*:

```
ExecQual(qualification scanTuple innerTuple outerTuple)
{
   while (more expr) {
           expr = next expr in qualification;
           if (ExecEvalOr(expr) == nil) return(nil);
   }
   return(true);
}

ExecExpr(expr)
{
   case (type of expr) {
           orExpr: ExecOrExpr(orExpr);
           oper:      call function manager;
           var:        evaluate the variables;
           ....
           ....
   }
}

ExecOrExpr(orExpr)
{
   if (single expression) return(ExecEvalExpr(orExpr));
   else {
           while (more expr) {
              expr = next expr in orExpr;
              if (ExecEvalExpr(expr) == true) return(true);
           }
           return(false);
   }
}
```

[9]Literals are represented in bolds. {} means 0 or more. "Var", "const", "param", "oper" and "func" are primitive nodes defined in the Planner Specifications.

### 6.2. ExecTargetList

ExecTargetList() constructs a new tuple either from the currently scanned tuple or the inner/outer tuples according to the target list associated with a node. A target list is a list of one or more (*resdom expr*). The return value of *expr* is stored in the attribute specified by the *resdom* structure. ExecTargetList() calls ExecEvalExpr() to evaluate expressions in all the (resdom expr) pairs and stored the addresses of the values in an list according to their attribute number. This list of attribute values and other information are then passed to a C routine that constructs a new tuple containing the values.

```
ExecTargetList(targetlist, attributeType)
{
   /* evaluate the exp in all (resdom expr) pairs in the
      target list.
   */
   for all (resdom expr) in target list {
           value = ExecEvalExpr(expr);
           insert value in attList;
   }
   /* construct the tuple */
   return(formTuple(numberAttr, attributeInfo, attList));
}
```

## 7. Error Handling

When an error occurs during the execution of the query plan, the error handling routine, IExecError(errorMssg), is called. The convention adopted for the error message is as follow:

"*RoutineName*: the cause of error."

The error handling routine set the global variable, *execErrorMssg*, to indicate the type of error and then forces control to the main routine using *(err)R call. The err* routine causes control to be passed to the previous call of *errset* with the return value *nil*. The error message is intercepted in the main routine which call *errset* before executing the query plan:

```
if (errset(execute query plan) == nil) {
   handle error condition;
} else {
   return successfully;
}
```