Take a walk on the query side

- Georgios

What’s this about?

- A very thin skeleton of query processing backend
- Basic entry points in the code
- Basic data structures

The data

```
CREATE TABLE hobbies (
    id integer NOT NULL,
    name text NOT NULL,
    hobby text NOT NULL
);


-- Add some values
INSERT INTO hobbies VALUES
    (1, 'daniel', 'photography'),
    (2, 'georgios', ‘motorcycling');
```

The query

```
SELECT ctid, * FROM hobbies WHERE id > 0;

-- Which you have guessed correctly will hit the disk
        QUERY PLAN
-------------------------------
 Seq Scan on public.hobbies
   Output: ctid, id, name, hobby
   Filter: (hobbies.id > 0)
(3 rows)
```

NODE and LIST

```
typedef struct Node
{
    NodeTag     type;
} Node;
```

And Lists….

* Once upon a time, parts of Postgres were written in Lisp and used real
* cons-cell lists for major data structures. When that code was rewritten
* in C, we initially had a faithful emulation of cons-cell lists, which

* unsurprisingly was a performance bottleneck. A couple of major rewrites
* later, these data structures are actually simple expansible arrays;
* but the "List" name and a lot of the notation survives.

Let’s dive in

Traffic cop, dispatches requests to the proper module

exec_simple_query(const char *query_string)

What was that? - Entry

(List *)parsetree_list = pg_parse_query(query_string);

Do basic parsing of the query. Absolutely no catalog lookups. Simple tokenisation via flex and grammar matching via bison.

What was that? - Lexer

```
/*
 * OK, here is a short description of lex/flex rules behavior.
 * The longest pattern which matches an input string is always chosen.
 * For equal-length patterns, the first occurring in the rules list is chosen.
 * INITIAL is the starting state, to which all non-conditional rules apply.
 * Exclusive states change parsing rules while the state is active.  When in
 * an exclusive state, only those rules defined for that state apply.
 *
 * We use exclusive states for quoted strings, extended comments,
 * and to eliminate parsing troubles for numeric strings.
* Exclusive states:
*  <xb> bit string literal
*  <xc> extended C-style comments
*  <xd> delimited identifiers (double-quoted identifiers)
*  <xh> hexadecimal numeric string
*  <xq> standard quoted strings
*  <xe> extended quoted strings (support backslash escape sequences)
*  <xdolq> $foo$ quoted strings
*  <xui> quoted identifier with Unicode escapes
*  <xuiend> end of a quoted identifier with Unicode escapes, UESCAPE can follow
*  <xus> quoted string with Unicode escapes
*  <xusend> end of a quoted string with Unicode escapes, UESCAPE can follow
*  <xeu> Unicode surrogate pair in extended quoted string
*
* Remember to add an <<EOF>> case whenever you add a new exclusive state!
* The default one is probably not the right thing.
*/
```

What was that? - Grammar

```
/* ----------------------
 *      Select Statement
 *
 * A "simple" SELECT is represented in the output of gram.y by a single
* SelectStmt node; so is a VALUES construct.  A query containing set
* operators (UNION, INTERSECT, EXCEPT) is represented by a tree of SelectStmt
* nodes, in which the leaf nodes are component SELECTs and the internal nodes
* represent UNION, INTERSECT, or EXCEPT operators.  Using the same node
* type for both leaf and internal nodes allows gram.y to stick ORDER BY,
* LIMIT, etc, clause values into a SELECT statement without worrying
* whether it is a simple or compound SELECT.
* ----------------------
*/
```

What was that? - Grammar

```
simple_select:
    SELECT opt_all_clause opt_target_list
    into_clause from_clause where_clause
    group_clause having_clause window_clause
        {
        SelectStmt *n = makeNode(SelectStmt);
        n->targetList = $3;
        n->intoClause = $4;
        n->fromClause = $5;
        n->whereClause = $6;
        n->groupClause = $7;
        n->havingClause = $8;
        n->windowClause = $9;
        $$ = (Node *)n;
}
```

What was that? - Grammar

```
--  Let's take a better look at the targetList and basically at '*'

    It will match:
        | '*'
            {
                ColumnRef *n = makeNode(ColumnRef);
                n->fields = list_make1(makeNode(A_Star));
                n->location = @1;

                $$ = makeNode(ResTarget);
                $$->name = NULL;
                $$->indirection = NIL;
                $$->val = (Node *)n;
                $$->location = @1;
            }
  ;
```

What was that? - Grammar

```
/*
 * ColumnRef - specifies a reference to a column, or possibly a whole tuple
 *
 * The "fields" list must be nonempty.  It can contain string Value nodes
 * (representing names) and A_Star nodes (representing occurrence of a '*').
 */
typedef struct ColumnRef
{
    NodeTag     type;
    List       *fields;         /* field names (Value strings) or A_Star */
    int         location;       /* token location, or -1 if unknown */
} ColumnRef;

/*
 * ResTarget -
 *    result target (used in target list of pre-transformed parse trees)
 *
 * In a SELECT target list, 'name' is the column label from an
 * 'AS ColumnLabel' clause, or NULL if there was none, and 'val' is the
 * value expression itself.  The 'indirection' field is not used.
 */
typedef struct ResTarget
{
    NodeTag     type;
    char       *name;           /* column name or NULL */
    List       *indirection;    /* subscripts, field names, and '*', or NIL */
    Node       *val;            /* the value expression to compute or assign */
    int         location;       /* token location, or -1 if unknown */
} ResTarget;
```

Ok, but what was it really? - Analysis

```
/*
 *      RawStmt --- container for any one statement's raw parse tree
 *
 * Parse analysis converts a raw parse tree headed by a RawStmt node into
 * an analyzed statement headed by a Query node.  For optimizable statements,
 * the conversion is complex.  For utility statements, the parser usually just
 * transfers the raw parse tree (sans RawStmt) into the utilityStmt field of
 * the Query node, and all the useful work happens at execution time.
 *
 * stmt_location/stmt_len identify the portion of the source text string
 * containing this raw statement (useful for multi-statement strings).
 */
typedef struct RawStmt
{
    NodeTag     type;
    Node       *stmt;           /* raw parse tree */
    int         stmt_location;  /* start location, or -1 if unknown */
    int         stmt_len;       /* length in bytes; 0 means "rest of string" */
} RawStmt;
```

Ok, but what was it really? - Analysis

```
/*
 * Given a raw parsetree (gram.y output), and optionally information about
 * types of parameter symbols ($n), perform parse analysis and rule rewriting.
 *
 * A list of Query nodes is returned, since either the analyzer or the
 * rewriter might expand one query to several.
 */
List *
pg_analyze_and_rewrite(RawStmt *parsetree, const char *query_string,
                       Oid *paramTypes, int numParams,
                       QueryEnvironment *queryEnv)
```

Ok, but what was it really? - Analysis

The example of targetList. We need to change what the user wrote, with what we can actually work with.

Enter TargetEntry.

Ok, but what was it really? - Analysis

```
typedef struct TargetEntry
{
    Expr        xpr;
    Expr       *expr;           /* expression to evaluate */
    AttrNumber  resno;          /* attribute number (see notes above) */
    char       *resname;        /* name of the column (could be NULL) */
    Index       ressortgroupref;    /* nonzero if referenced by a sort/group
                                     * clause */
    Oid         resorigtbl;     /* OID of column's source table */
    AttrNumber  resorigcol;     /* column's number in source table */
    bool        resjunk;        /* set to true to eliminate the attribute from
                                 * final target list */
} TargetEntry;
```

```
List *
transformTargetList(ParseState *pstate, List *targetlist,
                    ParseExprKind exprKind)
```

Ok, but what was it really? - Analysis

```
-> transformColumnRef
     -> colNameToVar

     /*
      * colNameToVar
      *    Search for an unqualified column name.
      *    If found, return the appropriate Var node (or expression).
      *    If not found, return NULL.  If the name proves ambiguous, raise error.
      *    If localonly is true, only names in the innermost query are considered.
      */
     Node *
     colNameToVar(ParseState *pstate, const char *colname, bool localonly,
                  int location)
```

Ok, but what was it really? - Analysis

```
SELECT * FROM pg_attribute WHERE attrelid = 'hobbies'::regclass AND attname = 'ctid';
```

Ok, but what was it really? - Analysis

```
typedef struct Var
{
    Expr        xpr;
    Index       varno;          /* index of this var's relation in the range
                                 * table, or INNER_VAR/OUTER_VAR/INDEX_VAR */
    AttrNumber  varattno;       /* attribute number of this var, or zero for
                                 * all attrs ("whole-row Var") */
    Oid         vartype;        /* pg_type OID for the type of this var */
    int32       vartypmod;      /* pg_attribute typmod value */
    Oid         varcollid;      /* OID of collation, or InvalidOid if none */
    Index       varlevelsup;    /* for subquery variables referencing outer
                                 * relations; 0 in a normal var, >0 means N
                                 * levels up */
    Index       varnoold;       /* original value of varno, for debugging */
    AttrNumber  varoattno;      /* original value of varattno */
    int         location;       /* token location, or -1 if unknown */
} Var;
```

Ok, but what was it really? - Analysis

```
/*
 * ExpandColumnRefStar()
 *      Transforms foo.* into a list of expressions or targetlist entries.
 *
 * This handles the case where '*' appears as the last or only item in a
 * ColumnRef.  The code is shared between the case of foo.* at the top level
 * in a SELECT target list (where we want TargetEntry nodes in the result)
 * and foo.* in a ROW() or VALUES() construct (where we want just bare
 * expressions).
 *
 * The referenced columns are marked as requiring SELECT access.
 */
static List *
ExpandColumnRefStar(ParseState *pstate, ColumnRef *cref,
                    bool make_target_entry)
```

Ok, but what was it really? - Rewrite

```
/*
 * Perform rewriting of a query produced by parse analysis.
 */
Static List *
Pg_rewrite_query(Query *query)
```

Query?

```
/*
 * Query -
 *    Parse analysis turns all statements into a Query tree
 *    for further processing by the rewriter and planner.
 *
 *    Utility statements (i.e. non-optimizable statements) have the
 *    utilityStmt field set, and the rest of the Query is mostly dummy.
 *
 *    Planning converts a Query tree into a Plan tree headed by a PlannedStmt
 *    node --- the Query structure is not used by the executor.
 */
typedef struct Query
{
    NodeTag     type;

    CmdType     commandType;    /* select|insert|update|delete|utility */

    QuerySource querySource;    /* where did I come from? */

    uint64      queryId;        /* query identifier (can be set by plugins) */

    bool        canSetTag;      /* do I set the command result tag? */

    Node       *utilityStmt;    /* non-null if commandType == CMD_UTILITY */

    int         resultRelation; /* rtable index of target relation for
                                 * INSERT/UPDATE/DELETE; 0 for SELECT */

    bool        hasAggs;        /* has aggregates in tlist or havingQual */
```

How can we do that?

Planning!

But let’s not spend the rest of next year here…

Just the basics!

How can we do that?

```
querytree_list = pg_analyze_and_rewrite(parsetree, query_string,
                                        NULL, 0, NULL);

plantree_list = pg_plan_queries(querytree_list,
                                CURSOR_OPT_PARALLEL_OK, NULL);
```

How can we do that?

```
typedef struct Path
{
    NodeTag     type;

    NodeTag     pathtype;       /* tag identifying scan/join method */

    RelOptInfo *parent;         /* the relation this path can build */
    PathTarget *pathtarget;     /* list of Vars/Exprs, cost, width */

    ParamPathInfo *param_info;  /* parameterization info, or NULL if none */

    bool        parallel_aware; /* engage parallel-aware logic? */
    bool        parallel_safe;  /* OK to use as part of parallel plan? */
    int         parallel_workers;   /* desired # of workers; 0 = not parallel */

    /* estimated size/costs for path (see costsize.c for more info) */
    double      rows;           /* estimated number of result tuples */
    Cost        startup_cost;   /* cost expended before fetching any tuples */
    Cost        total_cost;     /* total cost (assuming all tuples fetched) */

    List       *pathkeys;       /* sort ordering of path's output */
    /* pathkeys is a List of PathKey nodes; see above */
} Path;
```

How can we do that?

```
/*
 * query_planner
 *    Generate a path (that is, a simplified plan) for a basic query,
 *    which may involve joins but not any fancier features.
 *
 * Since query_planner does not handle the toplevel processing (grouping,
 * sorting, etc) it cannot select the best path by itself.  Instead, it
 * returns the RelOptInfo for the top level of joining, and the caller
 * (grouping_planner) can choose among the surviving paths for the rel.
 *
 * root describes the query to plan
 * qp_callback is a function to compute query_pathkeys once it's safe to do so
 * qp_extra is optional extra data to pass to qp_callback
 *
 * Note: the PlannerInfo node also includes a query_pathkeys field, which
 * tells query_planner the sort order that is desired in the final output
 * plan.  This value is *not* available at call time, but is computed by
 * qp_callback once we have completed merging the query's equivalence classes.
 * (We cannot construct canonical pathkeys until that's done.)
 */
RelOptInfo *
query_planner(PlannerInfo *root,
          query_pathkeys_callback qp_callback, void *qp_extra)
```

How can we do that?

RelOptInfo

Per relation information for planning / optimization

It has all the information that is needed about the relation
Which can be a base or output of SubSelect or functions
That appear in the range table.

Parts of this data structure are specific to various scan
and join mechanisms.

* src/include/nodes/pathnodes.h

How can we do that?

```
/*
 * get_relation_info -
 *    Retrieves catalog information for a given relation.
 *
 * Given the Oid of the relation, return the following info into fields
 * of the RelOptInfo struct:
 *
 *  min_attr    lowest valid AttrNumber
 *  max_attr    highest valid AttrNumber
 *  indexlist   list of IndexOptInfos for relation's indexes
 *  statlist    list of StatisticExtInfo for relation's statistic objects
 *  serverid    if it's a foreign table, the server OID
 *  fdwroutine  if it's a foreign table, the FDW function pointers
 *  pages       number of pages
 *  tuples      number of tuples
 *  rel_parallel_workers user-defined number of parallel workers
 *
 * Also, add information about the relation's foreign keys to root->fkey_list.
 *
 * Also, initialize the attr_needed[] and attr_widths[] arrays.  In most
 * cases these are left as zeroes, but sometimes we need to compute attr
 * widths here, and we may as well cache the results for costsize.c.
 *
 * If inhparent is true, all we need to do is set up the attr arrays:
 * the RelOptInfo actually represents the appendrel formed by an inheritance
 * tree, and so the parent rel's physical size and index information isn't
 * important for it.
 */
void
get_relation_info(PlannerInfo *root, Oid relationObjectId, bool inhparent,
                  RelOptInfo *rel)
```

How can we do that?

```
(gdb) p *path
$7 = {
  type = T_Path,
  pathtype = T_SeqScan,
  parent = 0x55764fd3ac20,
  pathtarget = 0x55764fdf4318,
  param_info = 0x0,
  parallel_aware = false,
  parallel_safe = true,
  parallel_workers = 0,
  rows = 283,
  startup_cost = 0,
  total_cost = 20.625,
  pathkeys = 0x0
}
```

```
(gdb) p *(Var *)path->pathtarget->exprs->elements[0]
$12 = {
  xpr = {
    type = T_Var
  },
  varno = 1,
  varattno = -1,
  vartype = 27,
  vartypmod = -1,
  varcollid = 0,
  varlevelsup = 0,
  varnoold = 1,
  varoattno = -1,
  location = 7
}
```

How can we do that?

```
 *
 * The output of the planner is a Plan tree headed by a PlannedStmt node.
 * PlannedStmt holds the "one time" information needed by the executor.
 *
 * For simplicity in APIs, we also wrap utility statements in PlannedStmt
 * nodes; in such cases, commandType == CMD_UTILITY, the statement itself
 * is in the utilityStmt field, and the rest of the struct is mostly dummy.
 * (We do use canSetTag, stmt_location, stmt_len, and possibly queryId.)
 * ----------------
 */
typedef struct PlannedStmt
{
    NodeTag     type;

    CmdType     commandType;    /* select|insert|update|delete|utility */

    uint64      queryId;        /* query identifier (copied from Query) */

    bool        hasReturning;   /* is it insert|update|delete RETURNING? */

    bool        hasModifyingCTE;    /* has insert|update|delete in WITH? */

    bool        canSetTag;      /* do I set the command result tag? */

    bool        transientPlan;  /* redo plan when TransactionXmin changes? */

    bool        dependsOnRole;  /* is plan specific to current role? */

    bool        parallelModeNeeded; /* parallel mode required to execute? */

    int         jitFlags;       /* which forms of JIT should be performed */

    struct Plan *planTree;      /* tree of Plan nodes */
```

Let’s do it then!

```
* A portal is an abstraction which represents the execution state of
* a running or runnable query.  Portals support both SQL-level CURSORs
* and protocol-level portals.
*
* Scrolling (nonsequential access) and suspension of execution are allowed
* only for portals that contain a single SELECT-type query.  We do not want
* to let the client suspend an update-type query partway through!  Because
* the query rewriter does not allow arbitrary ON SELECT rewrite rules,
* only queries that were originally update-type could produce multiple
* plan trees; so the restriction to a single query is not a problem
* in practice.
```

Let’s do it then!

```
typedef struct PortalData
{
    /* Bookkeeping data */
    const char *name;           /* portal's name */
    const char *prepStmtName;   /* source prepared statement (NULL if none) */
    MemoryContext portalContext;    /* subsidiary memory for portal */
    ResourceOwner resowner;     /* resources owned by portal */
    void        (*cleanup) (Portal portal); /* cleanup hook */

    SubTransactionId createSubid;   /* the creating subxact */
    SubTransactionId activeSubid;   /* the last subxact with activity */

    /* The query or queries the portal will execute */
    const char *sourceText;     /* text of query (as of 8.4, never NULL) */
    const char *commandTag;     /* command tag for original query */
    List       *stmts;          /* list of PlannedStmts */
    CachedPlan *cplan;          /* CachedPlan, if stmts are from one */

    ParamListInfo portalParams; /* params to pass to query */
    QueryEnvironment *queryEnv; /* environment for query */

    /* Features/options */
    PortalStrategy strategy;    /* see above */
    int         cursorOptions;  /* DECLARE CURSOR option bits */
    bool        run_once;       /* portal will only be run once */
```

Let’s do it then!

```
/*
 * Run the portal to completion, and then drop it (and the receiver).
 */
(void) PortalRun(portal,
                 FETCH_ALL,
                 true,  /* always top level */
                 true,
                 receiver,
                 receiver,
                 completionTag);
```

Let’s do it then!

```
* The executor stores tuples in a "tuple table" which is a List of
* independent TupleTableSlots.
*
* There's various different types of tuple table slots, each being able to
* store different types of tuples. Additional types of slots can be added
* without modifying core code. The type of a slot is determined by the
* TupleTableSlotOps* passed to the slot creation routine. The builtin types
* of slots are
*
* 1. physical tuple in a disk buffer page (TTSOpsBufferHeapTuple)
* 2. physical tuple constructed in palloc'ed memory (TTSOpsHeapTuple)
* 3. "minimal" physical tuple constructed in palloc'ed memory
*    (TTSOpsMinimalTuple)
* 4. "virtual" tuple consisting of Datum/isnull arrays (TTSOpsVirtual)
```

Let’s do it then!

```
/* base tuple table slot type */
typedef struct TupleTableSlot
{
    NodeTag     type;
#define FIELDNO_TUPLETABLESLOT_FLAGS 1
    uint16      tts_flags;      /* Boolean states */
#define FIELDNO_TUPLETABLESLOT_NVALID 2
    AttrNumber  tts_nvalid;     /* # of valid values in tts_values */
    const TupleTableSlotOps *const tts_ops; /* implementation of slot */
#define FIELDNO_TUPLETABLESLOT_TUPLEDESCRIPTOR 4
    TupleDesc   tts_tupleDescriptor;    /* slot's tuple descriptor */
#define FIELDNO_TUPLETABLESLOT_VALUES 5
    Datum      *tts_values;     /* current per-attribute values */
#define FIELDNO_TUPLETABLESLOT_ISNULL 6
    bool       *tts_isnull;     /* current per-attribute isnull flags */
    MemoryContext tts_mcxt;     /* slot itself is in this context */
    ItemPointerData tts_tid;    /* stored tuple's tid */
    Oid         tts_tableOid;   /* table oid of tuple */
} TupleTableSlot;
```

Let’s do it then!

```
*      At ExecutorStart()
*      ----------------
*
*      - ExecInitSeqScan() calls ExecInitScanTupleSlot() to construct a
*        TupleTableSlots for the tuples returned by the access method, and
*        ExecInitResultTypeTL() to define the node's return
*        type. ExecAssignScanProjectionInfo() will, if necessary, create
*        another TupleTableSlot for the tuples resulting from performing
*        target list projections.
*
*      During ExecutorRun()
*      ----------------
*      - SeqNext() calls ExecStoreBufferHeapTuple() to place the tuple
*        returned by the access method into the scan tuple slot.
*
*      - ExecSeqScan() (via ExecScan), if necessary, calls ExecProject(),
*        putting the result of the projection in the result tuple slot. If
*        not necessary, it directly returns the slot returned by SeqNext().
*
*      - ExecutePlan() calls the output function.
*
```

Let’s do it then!

```
/*
 * Return next tuple from `scan`, store in slot.
 */
static inline bool
table_scan_getnextslot(TableScanDesc sscan, ScanDirection direction, TupleTableSlot *slot)
{
    slot->tts_tableOid = RelationGetRelid(sscan->rs_rd);
    return sscan->rs_rd->rd_tableam->scan_getnextslot(sscan, direction, slot);
}
```

Let’s do it then!

```
(gdb) p *slot
$13 = {
  type = T_TupleTableSlot,
  tts_flags = 16,
  tts_nvalid = 0,
  tts_ops = 0x55764f4911c0 <TTSOpsBufferHeapTuple>,
  tts_tupleDescriptor = 0x7f74cbe72488,
  tts_values = 0x55764fdf9510,
  tts_isnull = 0x55764fdf9528,
  tts_mcxt = 0x55764fdf8f40,
  tts_tid = {
    ip_blkid = {
      bi_hi = 0,
      bi_lo = 0
    },
    ip_posid = 1
  },
  tts_tableOid = 16384
}
(gdb)
```

Let’s do it then!

```
/*
 * check that the current tuple satisfies the qual-clause
 *
 * check for non-null qual here to avoid a function call to ExecQual()
 * when the qual is null ... saves only a few cycles, but they add up
 * ...
 */
if (qual == NULL || ExecQual(qual, econtext))
{
    /*
     * Found a satisfactory scan tuple.
     */
    if (projInfo)
    {
        /*
         * Form a projection tuple, store it in the result tuple slot
         * and return it.
         */
        return ExecProject(projInfo);
```

Let’s do it then!

```
/* ----------------
 *      printtup --- print a tuple in protocol 3.0
 * ----------------
 */
static bool
printtup(TupleTableSlot *slot, DestReceiver *self)
```

Let’s do it then!

```
/*
 * send the attributes of this tuple
 */
for (i = 0; i < natts; ++i)
{
    PrinttupAttrInfo *thisState = myState->myinfo + i;
    Datum       attr = slot->tts_values[i];

    if (slot->tts_isnull[i])
    {
        pq_sendint32(buf, -1);
        continue;
    }

    /*
     * Here we catch undefined bytes in datums that are returned to the
     * client without hitting disk; see comments at the related check in
     * PageAddItem().  This test is most useful for uncompressed,
     * non-external datums, but we're quite likely to see such here when
     * testing new C functions.
     */
    if (thisState->typisvarlena)
        VALGRIND_CHECK_MEM_IS_DEFINED(DatumGetPointer(attr),
                                      VARSIZE_ANY(attr));

    if (thisState->format == 0)
    {
        /* Text output */
        char       *outputstr;

        outputstr = OutputFunctionCall(&thisState->finfo, attr);
        pq_sendcountedtext(buf, outputstr, strlen(outputstr), false);
    }
```

Let’s do it then!

```
postgres=# SELECT ctid, * FROM hobbies WHERE id > 0;
 ctid  | id |   name    |     hobby
-------+----+-----------+---------------
 (0,1) |  1 | daniel    | photography
 (0,4) |  2 | georgios | motorcycles
(2 rows)
```