Handout: Introduction to Hacking PostgreSQL

Neil Conway and Gavin Sherry

July 11, 2006

Parser Changes

`src/backend/parser/gram.y`, circa line 2521:

```
CreateTrigStmt:
            CREATE TRIGGER name
            TriggerActionTime TriggerEvents ON
            qualified_name TriggerForSpec TriggerWhen EXECUTE PROCEDURE
            func_name '(' TriggerFuncArgs ')'
                {
                    CreateTrigStmt *n = makeNode(CreateTrigStmt);
                    n->trigname = $3;
                    n->relation = $8;
                    n->funcname = $13;
                    n->args = $15;
                    n->before = $5;
                    n->row = $9;
                    memcpy(n->actions, $6, 4);
                    n->isconstraint  = FALSE;
                    n->deferrable    = FALSE;
                    n->initdeferred  = FALSE;
                    n->constrrel = NULL;
                    n->when = $10;
                    n->rtable = NIL;
                    $$ = (Node *)n;
                }
        ;


TriggerWhen:
            WHEN '(' a_expr ')'                         { $$ = $3; }
            | /*EMPTY*/                                 { $$ = NULL; }
        ;
```

Node changes

`src/include/nodes/parsenodes.h` circa line 1148:

```
/* ----------------------
 *      Create/Drop TRIGGER Statements
 * ----------------------
 */

typedef struct CreateTrigStmt
{
    NodeTag     type;
    char       *trigname;       /* TRIGGER's name */
    RangeVar   *relation;       /* relation trigger is on */
    List       *funcname;       /* qual. name of function to call */
    List       *args;           /* list of (T_String) Values or NIL */
    bool        before;         /* BEFORE/AFTER */
    bool        row;            /* ROW/STATEMENT */
    char        actions[4];     /* 1 to 3 of 'i', 'u', 'd', + trailing \0 */

    /* The following are used for referential */
    /* integrity constraint triggers */
    bool        isconstraint;   /* This is an RI trigger */
    bool        deferrable;     /* [NOT] DEFERRABLE */
    bool        initdeferred;   /* INITIALLY {DEFERRED|IMMEDIATE} */
    RangeVar   *constrrel;      /* opposite relation */
    Node       *when;           /* WHEN clause qual */
    List       *rtable;         /* range table for interpreting WHEN expr */
} CreateTrigStmt;
```

`src/backend/nodes/copyfuncs.c` circa line 2428:

```
static CreateTrigStmt *
_copyCreateTrigStmt(CreateTrigStmt *from)
{
    CreateTrigStmt *newnode = makeNode(CreateTrigStmt);

    COPY_STRING_FIELD(trigname);
    COPY_NODE_FIELD(relation);
    COPY_NODE_FIELD(funcname);
    COPY_NODE_FIELD(args);
    COPY_SCALAR_FIELD(before);
    COPY_SCALAR_FIELD(row);
    strcpy(newnode->actions, from->actions);    /* in-line string field */
    COPY_SCALAR_FIELD(isconstraint);
    COPY_SCALAR_FIELD(deferrable);
    COPY_SCALAR_FIELD(initdeferred);
    COPY_NODE_FIELD(constrrel);
    COPY_NODE_FIELD(when);
    COPY_NODE_FIELD(rtable);

    return newnode;
}
```

`src/backend/nodes/equalfuncs.c` circa line 1295:

```
static bool
_equalCreateTrigStmt(CreateTrigStmt *a, CreateTrigStmt *b)
{
    COMPARE_STRING_FIELD(trigname);
    COMPARE_NODE_FIELD(relation);
    COMPARE_NODE_FIELD(funcname);
    COMPARE_NODE_FIELD(args);
    COMPARE_SCALAR_FIELD(before);
    COMPARE_SCALAR_FIELD(row);
    if (strcmp(a->actions, b->actions) != 0)    /* in-line string field */
        return false;
    COMPARE_SCALAR_FIELD(isconstraint);
    COMPARE_SCALAR_FIELD(deferrable);
    COMPARE_SCALAR_FIELD(initdeferred);
    COMPARE_NODE_FIELD(constrrel);
    COMPARE_NODE_FIELD(when);
    COMPARE_NODE_FIELD(rtable);

    return true;
}
```

Trigger descriptor changes

`src/include/utils/rel.h`, circa line 68:

```
typedef struct TriggerDesc
{
    /*
     * Index data to identify which triggers are which.  Since each trigger
     * can appear in more than one class, for each class we provide a list of
     * integer indexes into the triggers array.
     */
#define TRIGGER_NUM_EVENT_CLASSES  3

    uint16      n_before_statement[TRIGGER_NUM_EVENT_CLASSES];
    uint16      n_before_row[TRIGGER_NUM_EVENT_CLASSES];
    uint16      n_after_row[TRIGGER_NUM_EVENT_CLASSES];
    uint16      n_after_statement[TRIGGER_NUM_EVENT_CLASSES];
    int        *tg_before_statement[TRIGGER_NUM_EVENT_CLASSES];
    int        *tg_before_row[TRIGGER_NUM_EVENT_CLASSES];
    int        *tg_after_row[TRIGGER_NUM_EVENT_CLASSES];
    int        *tg_after_statement[TRIGGER_NUM_EVENT_CLASSES];

    /* The actual array of triggers is here */
    Trigger    *triggers;
    int         numtriggers;
} TriggerDesc;
```

`src/include/utils/rel.h`, circa line 45:

```
/*
 * Likewise, this struct really belongs to trigger.h, but for convenience
 * we put it here.
 */
typedef struct Trigger
{
    Oid         tgoid;          /* OID of trigger (pg_trigger row) */
    /* Remaining fields are copied from pg_trigger, see pg_trigger.h */
    char       *tgname;
    Oid         tgfoid;
    int16       tgtype;
    bool        tgenabled;
    bool        tgisconstraint;
    Oid         tgconstrrelid;
    bool        tgdeferrable;
    bool        tginitdeferred;
    int16       tgnargs;
    int16       tgnattr;
    int16      *tgattr;
    char      **tgargs;
    Node       *when;
} Trigger;
```

Analysis Phase Changes

`src/backend/parser/analyze.c`, circa line 1861:

```
/*
 * transformCreateTrigStmt -
 *    transform a CREATE TRIGGER statement. Most of the work we do is
 *    transforming the statement's WHEN clause, if any.
 */
static Query *
transformCreateTrigStmt(ParseState *pstate, CreateTrigStmt *stmt)
{
    Query       *qry;
    Relation     rel;
    RangeTblEntry *oldrte;
    RangeTblEntry *newrte;
    int          i;

    qry = makeNode(Query);
    qry->commandType = CMD_UTILITY;
    qry->utilityStmt = (Node *) stmt;

    /* If there's no WHEN clause, we're done. */
    if (!stmt->when)
        return qry;

    /*
```

```
     * Note that we acquire and keep an exclusive lock on the target table
     * only if there's a WHEN clause; if there's no WHEN, we acquire the same
     * lock in CreateTrigger(), so the effect should be the same.
     */
    rel = heap_openrv(stmt->relation, AccessExclusiveLock);

    /*
     * Setup RTEs for the NEW and OLD relations in the main pstate, for use
     * in parsing the trigger qualification. We initially add "OLD" with RT
     * index 1, and "NEW" with RT index 2, and then change them to use the
     * correct varnos below.
     */
    Assert(pstate->p_rtable == NIL);
    oldrte = addRangeTableEntryForRelation(pstate, rel,
                                           makeAlias("*OLD*", NIL),
                                           false, false);
    newrte = addRangeTableEntryForRelation(pstate, rel,
                                           makeAlias("*NEW*", NIL),
                                           false, false);

    for (i = 0; stmt->actions[i] != '\0'; i++)
    {
        if (stmt->actions[i] == 'd' || stmt->actions[i] == 'u')
        {
            addRTEtoQuery(pstate, oldrte, false, true, true);
            break;
        }
    }

    for (i = 0; stmt->actions[i] != '\0'; i++)
    {
        if (stmt->actions[i] == 'i' || stmt->actions[i] == 'u')
        {
            addRTEtoQuery(pstate, newrte, false, true, true);
            break;
        }
    }

    /* process WHEN clause as though it was a WHERE clause */
    stmt->when = transformWhereClause(pstate, stmt->when, "WHEN");

    if (list_length(pstate->p_rtable) != 2)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_OBJECT_DEFINITION),
                 errmsg("trigger WHEN condition may not contain "
                        "references to other relations")));

    stmt->rtable = list_copy(pstate->p_rtable);

    /* aggregates not allowed */
    if (pstate->p_hasAggs)
```

```
        ereport(ERROR,
                (errcode(ERRCODE_GROUPING_ERROR),
        errmsg("trigger WHEN condition may not contain aggregate functions"))
        );

    /* subselects are not allowed either, at least for now */
    if (pstate->p_hasSubLinks)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_OBJECT_DEFINITION),
                 errmsg("trigger WHEN condition may not contain subqueries"))
                 );

    /*
     * Rewrite the WHEN expression to give the right varno to the NEW and OLD
     * relations, so that these relations can be treated specially by the
     * executor.
     */
    ChangeVarNodes(stmt->when, 1, TRIG_OLD_VARNO, 0);
    ChangeVarNodes(stmt->when, 2, TRIG_NEW_VARNO, 0);

    /* Close relation, but keep the exclusive lock */
    heap_close(rel, NoLock);

    return qry;
}
```

System Catalog Changes

`src/include/catalog/pg_trigger.h`, circa line 51:

```
CATALOG(pg_trigger,2620)
{
    Oid         tgrelid;         /* triggered relation */
    NameData    tgname;          /* trigger' name */
    Oid         tgfoid;          /* OID of function to be called */
    int2        tgtype;          /* BEFORE/AFTER UPDATE/DELETE/INSERT
                                  * ROW/STATEMENT */
    bool        tgenabled;       /* trigger is enabled/disabled */
    bool        tgisconstraint;  /* trigger is a RI constraint */
    NameData    tgconstrname;    /* RI constraint name */
    Oid         tgconstrrelid;   /* RI table of foreign key definition */
    bool        tgdeferrable;    /* RI trigger is deferrable */
    bool        tginitdeferred;  /* RI trigger is deferred initially */
    int2        tgnargs;         /* # of extra arguments in tgargs */

    /* VARIABLE LENGTH FIELDS: */
    int2vector  tgattr;          /* reserved for column-specific triggers */
    bytea       tgargs;          /* first\000second\000tgnargs\000 */
    text        tgqual;          /* string form of qualification clause */
} FormData_pg_trigger;

```

```
/* ... */

/* ----------------
 *        compiler constants for pg_trigger
 * ----------------
 */
#define Natts_pg_trigger                14
#define Anum_pg_trigger_tgrelid         1
#define Anum_pg_trigger_tgname          2
#define Anum_pg_trigger_tgfoid          3
#define Anum_pg_trigger_tgtype          4
#define Anum_pg_trigger_tgenabled       5
#define Anum_pg_trigger_tgisconstraint  6
#define Anum_pg_trigger_tgconstrname    7
#define Anum_pg_trigger_tgconstrrelid   8
#define Anum_pg_trigger_tgdeferrable    9
#define Anum_pg_trigger_tginitdeferred  10
#define Anum_pg_trigger_tgnargs         11
#define Anum_pg_trigger_tgattr          12
#define Anum_pg_trigger_tgargs          13
#define Anum_pg_trigger_tgqual          14
```

CREATE TRIGGER Changes

`src/backend/commands/trigger.c`, circa line 334, in `CreateTrigger()`:

```
    values[Anum_pg_trigger_tgqual - 1] = DirectFunctionCall1(textin,
                                  CStringGetDatum(nodeToString(stmt->when)));
```

Relation initialisation

`src/backend/commands/trigger.c` circa line 959 in `RelationBuildTriggers()`

```
        /* get the trigger's WHEN clause, if any */
         tmp = heap_getattr(htup, Anum_pg_trigger_tgqual,
                            RelationGetDescr(tgrel), &isnull);
         Assert(!isnull);

         when_str = DatumGetCString(DirectFunctionCall1(textout,
                                                    PointerGetDatum(tmp)))
                                                    ;

         /*
          * XXX: we leak the node here because FreeTriggerDesc() has no
          * ability to do a deep free of a Node.
          *
          * Ideally, the Node would be created in its own context which
          * we could just reset. Since we create the node in the
          * CacheMemoryContext the effect of this leak will be long lived
          */

        build->when = (Node *) stringToNode(when_str);
```

Executor Changes

`src/backend/commands/trigger.c`, circa line 3367:

```
/*
 * There is some inefficiency here. Subsequent calls to setup_trigger_quals()
 * during the same command will end up iterating through the array of
 * triggers. Ideally, this redundant work should be avoided.
 */
static void
setup_trigger_quals(ResultRelInfo *ri, EState *estate,
                    bool before, int event)
{
    TriggerDesc        *trigdesc = ri->ri_TrigDesc;
    MemoryContext       old_cxt;
    TrigQualState      *qual_state;
    int                 i;
    int                 ntrigs;
    int                *tgindx;

    old_cxt = MemoryContextSwitchTo(estate->es_query_cxt);
    if (ri->ri_TrigQuals == NULL)
    {
        ri->ri_TrigQuals = palloc(sizeof(TrigQualState));
        ri->ri_TrigQuals->quals = palloc0(trigdesc->numtriggers * sizeof(List
        *));
    }
    qual_state = ri->ri_TrigQuals;

    if (before)
    {
        ntrigs = trigdesc->n_before_row[event];
        tgindx = trigdesc->tg_before_row[event];
    }
    else
    {
        ntrigs = trigdesc->n_after_row[event];
        tgindx = trigdesc->tg_after_row[event];
    }

    for (i = 0; i < ntrigs; i++)
    {
        Trigger *trigger;
        int      trig_idx;
        List    *qual;

        trig_idx = tgindx[i];
        trigger = &trigdesc->triggers[trig_idx];

        if (!trigger->when)
            continue;

        if (qual_state->quals[trig_idx])
            continue;

        qual = make_ands_implicit((Expr *) trigger->when);
        qual_state->quals[trig_idx] = (List *) ExecPrepareExpr((Expr *) qual,
                                                                estate);
    }

    MemoryContextSwitchTo(old_cxt);
}
```

`src/backend/commands/trigger.c`, circa line 3426:

```
/*
 * Test if a trigger qualification evaluates true for the
 * input tuple(s)
 */

static bool
test_trig_qual(EState *estate, Relation rel, HeapTuple oldtuple,
               HeapTuple newtuple, List *qual, int event)
{
    ExprContext *econtext = GetPerTupleExprContext(estate);
    TupleDesc tupdesc = RelationGetDescr(rel);

    if (event == TRIGGER_EVENT_INSERT ||
        event == TRIGGER_EVENT_UPDATE)
    {
        if (econtext->ecxt_newtuple == NULL)
            econtext->ecxt_newtuple = MakeSingleTupleTableSlot(tupdesc);
        ExecClearTuple(econtext->ecxt_newtuple);
        ExecStoreTuple(newtuple, econtext->ecxt_newtuple,
                  InvalidBuffer, false);
    }
    if (event == TRIGGER_EVENT_UPDATE ||
        event == TRIGGER_EVENT_DELETE)
    {
        if (econtext->ecxt_oldtuple == NULL)
            econtext->ecxt_oldtuple = MakeSingleTupleTableSlot(tupdesc);
        ExecClearTuple(econtext->ecxt_oldtuple);
        ExecStoreTuple(oldtuple, econtext->ecxt_oldtuple,
                     InvalidBuffer, false);
    }

    return ExecQual(qual, econtext, false);
}
```

`src/backend/commands/trigger.c`, circa line 1449 in ExecBRInsertTriggers()

```
    setup_trigger_quals(relinfo, estate, true, TRIGGER_EVENT_INSERT);
    qual_state = relinfo->ri_TrigQuals;

    LocTriggerData.type = T_TriggerData;
    LocTriggerData.tg_event = TRIGGER_EVENT_INSERT |
        TRIGGER_EVENT_ROW |
        TRIGGER_EVENT_BEFORE;
    LocTriggerData.tg_relation = relinfo->ri_RelationDesc;
    LocTriggerData.tg_newtuple = NULL;
    LocTriggerData.tg_newtuplebuf = InvalidBuffer;

    for (i = 0; i < ntrigs; i++)
    {
        Trigger     *trigger = &trigdesc->triggers[tgindx[i]];

        if (!trigger->tgenabled)
            continue;

        /* Check the trigger's WHEN clause, if any */
        if (trigger->when)
        {
            bool res;

            res = test_trig_qual(estate, relinfo->ri_RelationDesc,
                                 NULL, trigtuple,
                                 qual_state->quals[tgindx[i]],
                                 TRIGGER_EVENT_INSERT);
            if (!res)
                continue;
        }

        LocTriggerData.tg_trigtuple = oldtuple = newtuple;
        LocTriggerData.tg_trigtuplebuf = InvalidBuffer;
        LocTriggerData.tg_trigger = trigger;
        newtuple = ExecCallTriggerFunc(&LocTriggerData,
                                       tgindx[i],
                                       relinfo->ri_TrigFunctions,
                                       relinfo->ri_TrigInstrument,
                                       GetPerTupleMemoryContext(estate));
        if (oldtuple != newtuple && oldtuple != trigtuple)
            heap_freetuple(oldtuple);
        if (newtuple == NULL)
            break;
    }
```

Note: also see identical modifications made to `ExecBRDeleteTriggers()` and `ExecBRUpdateTriggers()`.

Low level executor hackery

`src/backend/commands/trigger.c` circa line 463 in `CreateTrigger()`

```
    if (stmt->when)
    {
        ChangeVarNodes(stmt->when, TRIG_OLD_VARNO, 1, 0);
        ChangeVarNodes(stmt->when, TRIG_NEW_VARNO, 2, 0);
        recordDependencyOnExpr(&myself, stmt->when, stmt->rtable,
                               DEPENDENCY_NORMAL);
    }
```

`src/backend/executor/execQual.c` circa line 477 in `ExecEvalVar()`

```
        case TRIG_OLD_VARNO:    /* old tuple in trigger context */
            slot = econtext->ecxt_oldtuple;
            break;

        case TRIG_NEW_VARNO:    /* new tuple in trigger context */
            slot = econtext->ecxt_newtuple;
            break;
```

pg_dump and psql

`src/backend/utils/adt/ruleutils.c` circa line 522 in `pg_get_triggerdef()`

```
    /* handle WHEN clause */
    when_text = heap_getattr(ht_trig, Anum_pg_trigger_tgqual,
                             RelationGetDescr(tgrel), &isnull);
    Assert(!isnull);

    when_str = DatumGetCString(DirectFunctionCall1(textout, when_text));
    node = (Node *) stringToNode(when_str);

    if (node)
    {
        Relation rel;
        RangeTblEntry *oldrte;
        RangeTblEntry *newrte;
        deparse_context context;
        deparse_namespace dpns;

        rel = heap_open(trigrec->tgrelid, AccessShareLock);

        appendStringInfo(&buf, "WHEN ");

        oldrte = addRangeTableEntryForRelation(NULL, rel,
                                               makeAlias("*OLD*", NIL),
                                               false, false);

        newrte = addRangeTableEntryForRelation(NULL, rel,
                                               makeAlias("*NEW*", NIL),
                                               false, false);

        ChangeVarNodes(node, TRIG_OLD_VARNO, 1, 0);
        ChangeVarNodes(node, TRIG_NEW_VARNO, 2, 0);

        context.buf = &buf;
        context.namespaces = list_make1(&dpns);
        context.varprefix = true;
        context.prettyFlags = 0;
        context.indentLevel = PRETTYINDENT_STD;

        dpns.rtable = list_make2(oldrte, newrte);
        dpns.outer_varno = dpns.inner_varno = 0;
        dpns.outer_rte = dpns.inner_rte = NULL;

        get_rule_expr(node, &context, false);
        appendStringInfo(&buf, " ");
        heap_close(rel, AccessShareLock);
    }
```

Regression test system

`src/test/regress/sql/triggers.sql` circa line 419:

```
-- test the WHEN clause for trigger definitions
CREATE OR REPLACE FUNCTION notify_trig() RETURNS trigger LANGUAGE plpgsql AS $$
begin
    raise notice '%s invoked: new.a = %, new.b = %', tg_name, new.a, new.b;
    return new;
end$$;

CREATE TABLE when_test (a int, b int);

CREATE TRIGGER t1 BEFORE INSERT ON when_test FOR EACH ROW
WHEN (new.a > 5)
EXECUTE PROCEDURE notify_trig();

INSERT INTO when_test VALUES (NULL, 0);     -- shouldn't fire
INSERT INTO when_test VALUES (10, 100);     -- should fire

CREATE TRIGGER t2 BEFORE UPDATE ON when_test FOR EACH ROW
WHEN (new.b > 50)
EXECUTE PROCEDURE notify_trig();

UPDATE when_test SET b = b + 50;       -- should fire once
UPDATE when_test SET b = b + 1;        -- should fire twice

DROP TABLE when_test;
DROP FUNCTION notify_trig();
```

Documentation

`doc/src/sgml/ref/create_trigger.sgml` circa line 29:

```
<synopsis>
CREATE TRIGGER <replaceable class="PARAMETER">name</replaceable> { BEFORE | ↘
↪AFTER } { <replaceable class="PARAMETER">event</replaceable> [ OR ... ] }
    ON <replaceable class="PARAMETER">table</replaceable> [ FOR [ EACH ] { ↘
    ↪ROW | STATEMENT } ]
    [ WHEN ( <replaceable class="PARAMETER">expr</> ) ]
    EXECUTE PROCEDURE <replaceable class="PARAMETER">funcname</replaceable> (↘
    ↪ <replaceable class="PARAMETER">arguments</replaceable> )
</synopsis>

<!-- ... -->

   <varlistentry>
    <term><replaceable class="parameter">expr</replaceable></term>
    <listitem>
     <para>
      An SQL expression which returns <type>boolean</type> result.
     </para>

     <para>
      <literal>INSERT</literal> triggers may refer only to the
      <literal>NEW</literal> table. <literal>DELETE</literal> triggers
      may only refer to the <literal>OLD</literal> table.
      <literal>UPDATE</literal> triggers may refer to both. The
      expression may not refer to any other tables.
     </para>

     <para>
      This feature is not supported on <literal>FOR STATEMENT</> triggers.
     <para>
    </listitem>
   </varlistentry>
```