Writing A Foreign Data Wrapper

Bernd Helmle, bernd.helmle@credativ.de

24. Oktober 2012
Why FDWs?

- ...it is in the SQL Standard (SQL/MED)
- ...migration
- ...heterogeneous infrastructure
- ...integration of remote non-relational datasources
- ...fun

CREATE EXTENSION IF NOT EXISTS informix_fdw;

CREATE SERVER sles11_tcp FOREIGN DATA WRAPPER informix_fdw OPTIONS ( 
  informixdir '/Applications/IBM/informix',  
  informixserver 'ol_informix1170' 
);

CREATE USER MAPPING FOR bernd SERVER sles11_tcp OPTIONS ( 
  password 'informix',  
  username 'informix' 
);

CREATE FOREIGN TABLE bar ( 
  id integer,  
  value text 
) SERVER sles11_tcp OPTIONS ( 
  client_locale 'en_US.utf8', database 'test',  
  db_locale 'en_US.819', query 'SELECT * FROM bar' 
);

SELECT * FROM bar;
What we need...

- ...a C-interface to our remote datasource
- ...knowledge about PostgreSQL’s FDW API
- ...an idea how we deal with errors
- ...how remote data can be mapped to PostgreSQL datatypes
- ...time and steadiness

Python-Gurus also could use http://multicorn.org/.
Before you start your own...

Have a look at
http://wiki.postgresql.org/wiki/Foreign_data_wrappers
extern Datum ifx_fdw_handler(PG_FUNCTION_ARGS);
extern Datum ifx_fdw_validator(PG_FUNCTION_ARGS);

CREATE FUNCTION ifx_fdw_handler() RETURNS fdw_handler
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;

CREATE FUNCTION ifx_fdw_validator(text[], oid) RETURNS void
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;

CREATE FOREIGN DATA WRAPPER informix_fdw
  HANDLER ifx_fdw_handler
  VALIDATOR ifx_fdw_validator;
FDW handler

Creates and initializes a FdwRoutine structure:

Datum
ifx_fdw_handler(PG_FUNCTION_ARGS)
{
    FdwRoutine *fdwRoutine = makeNode(FdwRoutine);
    fdwRoutine->ExplainForeignScan = ifxExplainForeignScan;
    fdwRoutine->BeginForeignScan = ifxBeginForeignScan;
    fdwRoutine->IterateForeignScan = ifxIterateForeignScan;
    fdwRoutine->EndForeignScan = ifxEndForeignScan;
    fdwRoutine->ReScanForeignScan = ifxReScanForeignScan;

#if PG_VERSION_NUM < 90200
    fdwRoutine->PlanForeignScan = ifxPlanForeignScan;
#endif

#else

    fdwRoutine->GetForeignRelSize = ifxGetForeignRelSize;
    fdwRoutine->GetForeignPaths = ifxGetForeignPaths;
    fdwRoutine->GetForeignPlan = ifxGetForeignPlan;
#endif

    PG_RETURN_POINTER(fdwRoutine);
FDW validator callback

- Called via `CREATE FOREIGN TABLE` or `ALTER FOREIGN TABLE`
- Validates a List of FDW options.
- Use `untransformRelOptions()` to get a list of FDW options
- Don’t forget to test for duplicated options!
- Up to you which options you want to support
Helper functions

**Functions to ease access to FDW options**

foreign/foreign.h

extern ForeignServer *GetForeignServerByName(const char *name,
                                            bool missing_ok);

extern UserMapping *GetUserMapping(Oid userid, Oid serverid);

extern ForeignDataWrapper
  *GetForeignDataWrapperByName(const char *name,
                               bool missing_ok);

extern ForeignTable *GetForeignTable(Oid relid);

extern Oid get_foreign_data_wrapper_oid(const char *fdwname,
                                         bool missing_ok);

extern Oid get_foreign_server_oid(const char *servername,
                                   bool missing_ok);
#ifdef PG_VERSION_NUM < 90200

static FdwPlan *PlanForeignScan(Oid foreignTableOid,
                               PlannerInfo *planInfo,
                               RelOptInfo *baserel);
#else

static void GetForeignRelSize(PlannerInfo *root,
                               RelOptInfo *baserel,
                               Oid foreignTableId);
static void GetForeignPaths(PlannerInfo *root,
                             RelOptInfo *baserel,
                             Oid foreignTableId);
static ForeignScan *GetForeignPlan(PlannerInfo *root,
                                   RelOptInfo *baserel,
                                   Oid foreignTableId,
                                   ForeignPath *best_path,
                                   List *tlist,
                                   List *scan_clauses);
#endif
FDW API callback routines (2)

```
static void ExplainForeignScan(ForeignScanState *node, ExplainState *es);

static void BeginForeignScan(ForeignScanState *node, int eflags);

static TupleTableSlot *IterateForeignScan(ForeignScanState *node);

static void EndForeignScan(ForeignScanState *node);
```
FDW API callback routines (3)

9.2 has callbacks for ANALYZE, too:

```c
bool AnalyzeForeignTable (Relation relation,
                         AcquireSampleRowsFunc *func,
                         BlockNumber *totalpages);

int AcquireSampleRowsFunc (Relation relation, int elevel,
                           HeapTuple *rows, int targrows,
                           double *totalrows,
                           double *totaldeadrows);
```
FDW Flow

GetForeignRelSize() → Query Cost Estimates

GetForeignPaths() → Query Planning

GetForeignPlan() → Query Planning - Finalize

BeginForeignScan() → Prepare / Metadata

IterateForeignScan() → Materialize Tuple

EndForeignScan() → Free resources
FDW Query Planning

- Setup and Planning a scan on a foreign datasource
- E.g. establish and cache remote connection
- Initialize required supporting structures for remote access
- Planner info and cost estimates via baserel and root parameters.
- Big differences between 9.1 and 9.2 API
GetForeignRelSize() (1)

- Size restimates for remote datasource (table size, ...)

- root: Query Information Structure

- baserel: Table Information Structure, carry your FDW private information in baserel->fdw_private.

- Sets cost values
Save plan costs and estimates in baserel structure.

```c
baserel->rows = ifxGetEstimatedNRows(&coninfo);
baserel->width = ifxGetEstimatedRowSize(&coninfo);
planState->coninfo = coninfo;
planState->state = state;
baserel->fdw_private = (void *) planState;
```
GetForeignPaths() (1)

- Create access path for foreign datasource.
- ForeignPath access path required at least.
- Multiple paths possible (e.g. presorted results, ...)
- Arbitrarily complex
planState = (IfxFdwPlanState *) baserel->fdw_private;

/*
 * Create a generic foreign path for now. We need to consider any
 * restriction quals later, to get a smarter path generation here.
 *
 * For example, it is quite interesting to consider any index scans
 * or sorted output on the remote side and reflect it in the
 * chosen paths (helps nested loops et al.).
 */
add_path(baserel, (Path *)
        create_foreignscan_path(root, baserel,
                                baserel->rows,
                                planState->coninfo->planData.costs,
                                planState->coninfo->planData.costs,
                                NIL,
                                NULL,
                                NIL));
GetForeignPlan() (1)

- Creates a final ForeignScan plan node based on paths created by GetForeignPaths()

- Additional parameters

  - ForeignPath *best_path: Chosen foreign access path (best)

  - List *tlist: Target list

  - List *scan_clauses: Restriction clauses enforced by the plan
ForeignScan plan node should be created by `make_foreignscan()`:

```c
ForeignScan *
make_foreignscan(List *qptlist,
    List *qpqual,
    Index scanrelid,
    List *fdw_exprs,
    List *fdw_private)
```

- `fdw_exprs`: Expressions to be evaluated by the planner
- `fdw_private`: Private FDW data
Passing FDW planning info to execution state

- Save parameters in the foreignScan->fdw_private pointer.
- Must be copiable with copyObject.
- Use a List * with either bytea or/and constant values (via makeConst).

```c
List *plan_values;

plan_values = NIL;
plan_values = lappend(plan_values,
    makeConst(BYTEAOID, -1, InvalidOid, -1,
        PointerGetDatum(ifxFdwPlanDataAsBytea(coninfo)),
        false, false));

plan->fdw_private = plan_values;
```
Challenge: Filter the data on the remote dataset before transferring them, e.g.

```sql
SELECT COUNT(*) FROM sles11.inttest;
   count
-------
   10001
(1 row)
```

```sql
EXPLAIN SELECT * FROM foo
   JOIN (SELECT f1 FROM sles11.inttest
          WHERE f1 = 104 AND f2 = 120) AS t(id)
   ON (t.id = foo.id);
```

---

**QUERY PLAN**

```
Nested Loop  (cost=1.00..9.28 rows=1 width=12)
   ->  Index Only Scan using foo_id_idx on foo  (cost=0.00..8.27 rows=1 width=4)
       Index Cond: (id = 104)
   ->  Foreign Scan on inttest  (cost=1.00..1.00 rows=1 width=8)
       Filter: ((f1 = 104) AND (f2 = 120))
       Informix query: SELECT * FROM inttest WHERE  (f1 = 104) AND (f2 = 120)
(6 rows)
```
Nobody wants to filter thousands of rows to just get one
Idea: push filter conditions down to the foreign datasource (if possible)
Done during planning phase (GetForeignRelSize(), GetForeignPaths())
baserel->baserestrictinfo
Hard to get it right
Predicate Pushdown

- baserel->baserestrictinfo: List of predicates belonging to the foreign table (logically AND'ed)
- baserel->reltargetlist: List of columns belonging to the foreign table
- Have a look at expression_tree_walker() and ruleutils API (include/nodes/nodeFuncs.h, include/utils/ruleutils.h)

```c
ListCell *cell;

foreach(cell, baserel->baserestrictinfo)
{
    RestrictInfo *info;
    info = (RestrictInfo *) lfirst(cell);

    if (IsA(info->clause, OpExpr))
    {
        /* examine right and left operand */
    }
}
void
BeginForeignScan (ForeignScanState *node,
int eflags);

- Execute startup callback for the FDW.
- Basically prepares the FDW for executing a scan.
- ForeignScanState saves function state values.
- Use node->fdw_state to assign your own FDW state structure.
- Must handle EXPLAIN and EXPLAIN ANALYZE by checking eflags & EXEC_FLAG_EXPLAIN_ONLY
void
ExplainForeignScan (ForeignScanState *node,
                     ExplainState *es);

- Only ran when EXPLAIN is used.
- “Injects” EXPLAIN information.
- If there’s no additional information, just return
- E.g. calculated connection costs, timings etc.
IterateForeignScan() (1)

TupleTableSlot *
IterateForeignScan (ForeignScanState *node);

- Fetches data from the remote source.
- Data conversion
- Materializes a physical or virtual tuple to be returned.
- Needs to return an empty tuple when done.
Returning a virtual tuple

TupleTableSlot *slot = node->ss.ss_ScanTupleSlot;

slot->tts_isempty = false;
slot->tts_nvalid = number_cols;;
slot->tts_values = (Datum *)palloc(sizeof(Datum) * slot->tts_nvalid);
slot->tts_isnull = (bool *)palloc(sizeof(bool) * slot->tts_nvalid);

for (i = 0; j < attrCount - 1; i)
{
    tupleSlot->tts_isnull[i] = false;
    tupleSlot->tts_values[i] = PointerGetDatum(val);
}
void ReScanForeignScan (ForeignScanState *node);

- Prepares the FDW to handle a rescan
- Begins the scan from the beginning
- Must take care for changed query parameters!
- Better to just “instruct” IterateForeignScan() to do the right thing (tm)
EndForeignScan()

```c
void
EndForeignScan (ForeignScanState *node);
```

- Run when IterateForeignScan returns no more rows
- Finalizes the remote scan
- Close result sets, handles, connection, free memory, etc...
Memory Management

- PostgreSQL uses `palloc()`
- Memory is allocated in `CurrentMemoryContext`
- Use your own `MemoryContext` where necessary (e.g., `IterateForeignScan()`)
- Memory allocated in external libraries need special care
Data conversion

- Easy, if the remote datasource delivers a well formatted value string (e.g. date strings formatted as yyyy-mm-dd).
- Use type input function directly
- Binary compatible types (e.g integer)
- Binary data should always be bytea
- String data must have a valid encoding!
Within a FDW, a backend acts like any other client: ensure encoding compatibility or encode your string data properly.

A look at mb/pg_wchar.h might be of interest.

GetDatabaseEncoding()

pg_do_encoding_conversion()
regproc result;
HeapTuple type_tuple;

type_tuple = SearchSysCache1(TYPEOID, inputOid);
if (!HeapTupleIsValid(type_tuple))
{
    /*
     * Oops, this is not expected...
     */
    ifxRewindCallstack(&(state->stmt_info));
    elog(ERROR,
        "cache lookup failed for input function for type %u", inputOid);
}

ReleaseSysCache(type_tuple);
result = ((Form_pg_type) GETSTRUCT(type_tuple))->typinput;
Data conversion - Calling type input functions

Once having its OID, any type input function can be called like this:

```c
/* errors out */
typinputfunc = getTypeInputFunction(state, PG_ATTRTYPE_P(state, attnum));
result = OidFunctionCall2(typinputfunc,
    CStringGetDatum(buf),
    ObjectIdGetDatum(InvalidOid));
```
Error Handling (1)

- Set FDW SQLSTATE according to your error condition `class HV`, see `http://www.postgresql.org/docs/9.1/static/errcodes-appendix.html`
- Alternative: map remote error conditions to PostgreSQL errors
- Be careful with `elog(ERROR, ...)`. 
Example (there is no FDW_WARNING SQLSTATE):

```c
if (err == IFX_CONNECTION_WARN) {
    IfxSqlStateMessage message;
    ifxGetSqlStateMessage(1, &message);

    ereport(WARNING, (errcode(WARNING),
        errmsg("opened informix connection with warnings"),
        errdetail("informix SQLSTATE %s: "%s",
            message.sqlstate, message.text))));
}
```
Sometimes necessary to catch backend errors

Synchronize error conditions between PostgreSQL and remote datasource

Possibility: use a PG_TRY...PG_CATCH block.
Catching Errors (2)

```c
PG_TRY();
{
    ...
    typinputfunc = getTypeInputFunction(state, PG_ATTRTYPE_P(state, attnum));
    result = OidFunctionCall2(typinputfunc,
                             CStringGetDatum(pstrdup(buf)),
                             ObjectIdGetDatum(InvalidOid));
}
PG_CATCH();
{
    ifxRewindCallstack(&((state->stmt_info));
    PG_RE_THROW();
}
PG_END_TRY();
```
Thank You!