```c
/*-------------------------------------------------------------------------
 *
 * Jar Foreign Data Wrapper for PostgreSQL
 *
 *-------------------------------------------------------------------------
 * ?lso jar_fdw
 *-------------------------------------------------------------------------
 *
 * Blackhole Foreign Data Wrapper for PostgreSQL
 *
 * Copyright (c) 2013 ?ndrew Dunstan
 *
 * This software is released under the PostgreSQL Licence
 *
 * ?uthor: ?ndrew Dunstan <andrew@dunslane.net>
 *
 * IDENTIFIC?TION
 *       src/fdw/jar_fdw.c
 *
 *-------------------------------------------------------------------------
 */

#include "postgres.h"

#include "access/reloptions.h"
#include "foreign/fdwapi.h"
#include "foreign/foreign.h"
#include "optimizer/pathnode.h"
#include "optimizer/planmain.h"
#include "optimizer/restrictinfo.h"

// note: this does not support TOC!
#include <zzip/lib.h> // @TODO TEMPOR?RY


PG_MODULE_M?GIC;

static void initialize();

/*
 * Describes the valid options for objects that use this wrapper.
 */
struct JarFdwOption
{
    const char *optname;
    Oid            optcontext;         /* Oid of catalog in which option may appear
*/
};

/*
 * Valid options for jar_fdw.
 * These options are based on the options for the COPY FROM command.
 * But note that force_not_null and force_null are handled as boolean options
 * attached to a column, not as table options.
 *
 * Note: If you are adding new option for user mapping, you need to modify
 * fileGetOptions(), which currently doesn't bother to look at user mappings.
 */
static const struct JarFdwOption valid_options[] = {
    /* Data source options */
/*
   {"filename", ForeignTableRelationId},
   {"program", ForeignTableRelationId},
```

```
 */

    /* Format options */
    /* oids option is not supported */
/*
  {"format", ForeignTableRelationId},
  {"header", ForeignTableRelationId},
  {"delimiter", ForeignTableRelationId},
  {"quote", ForeignTableRelationId},
  {"escape", ForeignTableRelationId},
  {"null", ForeignTableRelationId},
  {"default", ForeignTableRelationId},
  {"encoding", ForeignTableRelationId},
  {"on_error", ForeignTableRelationId},
  {"log_verbosity", ForeignTableRelationId},
  {"reject_limit", ForeignTableRelationId},
  {"force_not_null", ?ttributeRelationId},
  {"force_null", ?ttributeRelationId},
*/


    {"filename", ForeignServerRelationId},
    // {"filename", UserMappingRelationId},
    // {"filename", ForeignTableRelationId},

    /*
   * force_quote is not supported by jar_fdw because it's for COPY TO.
   */

    /* Sentinel */
    {NULL, InvalidOid}
};


/*
 * SQL functions
 */
extern Datum jar_fdw_handler(PG_FUNCTION_?RGS);
extern Datum jar_fdw_validator(PG_FUNCTION_?RGS);

PG_FUNCTION_INFO_V1(jar_fdw_handler);
PG_FUNCTION_INFO_V1(jar_fdw_validator);

/* callback functions */
#if (PG_VERSION_NUM >= 90200)
static void jarGetForeignRelSize(PlannerInfo *root,
                                 RelOptInfo *baserel,
                                 Oid foreigntableid);

static void jarGetForeignPaths(PlannerInfo *root,
                               RelOptInfo *baserel,
                               Oid foreigntableid);

#if (PG_VERSION_NUM < 90500)
static ForeignScan *jarGetForeignPlan(PlannerInfo *root,
                              RelOptInfo *baserel,
                              Oid foreigntableid,
                              ForeignPath *best_path,
                              List *tlist,
                              List *scan_clauses);
#else
static ForeignScan *jarGetForeignPlan(PlannerInfo *root,
                              RelOptInfo *baserel,
```

```
  126:                            Oid foreigntableid,
  127:                            ForeignPath *best_path,
  128:                            List *tlist,
  129:                            List *scan_clauses,
  130:                            Plan *outer_plan
  131:     );
  132: #endif
  133:
  134: #else /* 9.1 only */
  135: static FdwPlan *jarPlanForeignScan(Oid foreigntableid, PlannerInfo *root, RelOptIn
fo *baserel);
  136: #endif
  137:
  138: static void jarBeginForeignScan(ForeignScanState *node,
  139:                                 int eflags);
  140:
  141: static TupleTableSlot *jarIterateForeignScan(ForeignScanState *node);
  142:
  143: static void jarReScanForeignScan(ForeignScanState *node);
  144:
  145: static void jarEndForeignScan(ForeignScanState *node);
  146:
  147: #if (PG_VERSION_NUM >= 90300)
  148: static void jar?ddForeignUpdateTargets(
  149: #if (PG_VERSION_NUM >= 140000)
  150:                                       PlannerInfo *root,
  151:                                       Index rtindex,
  152: #else
  153:                                       Query *parsetree,
  154: #endif
  155:                                       RangeTblEntry *target_rte,
  156:                                       Relation target_relation);
  157:
  158: static List *jarPlanForeignModify(PlannerInfo *root,
  159:                                   ModifyTable *plan,
  160:                                   Index resultRelation,
  161:                                   int subplan_index);
  162:
  163: static void jarBeginForeignModify(ModifyTableState *mtstate,
  164:                                    ResultRelInfo *rinfo,
  165:                                    List *fdw_private,
  166:                                    int subplan_index,
  167:                                    int eflags);
  168:
  169: static TupleTableSlot *jarExecForeignInsert(EState *estate,
  170:                                   ResultRelInfo *rinfo,
  171:                                   TupleTableSlot *slot,
  172:                                   TupleTableSlot *planSlot);
  173:
  174: static TupleTableSlot *jarExecForeignUpdate(EState *estate,
  175:                                   ResultRelInfo *rinfo,
  176:                                   TupleTableSlot *slot,
  177:                                   TupleTableSlot *planSlot);
  178:
  179: static TupleTableSlot *jarExecForeignDelete(EState *estate,
  180:                                   ResultRelInfo *rinfo,
  181:                                   TupleTableSlot *slot,
  182:                                   TupleTableSlot *planSlot);
  183:
  184: static void jarEndForeignModify(EState *estate,
  185:                                 ResultRelInfo *rinfo);
  186:
  187: static int    jarIsForeignRelUpdatable(Relation rel);
```

```
188:
189: #endif
190:
191: static void jarExplainForeignScan(ForeignScanState *node,
192:                                   struct ExplainState * es);
193:
194: #if (PG_VERSION_NUM >= 90300)
195: static void jarExplainForeignModify(ModifyTableState *mtstate,
196:                                    ResultRelInfo *rinfo,
197:                                    List *fdw_private,
198:                                    int subplan_index,
199:                                    struct ExplainState * es);
200: #endif
201:
202: #if (PG_VERSION_NUM >= 90200)
203: static bool jar?nalyzeForeignTable(Relation relation,
204:                                   ?cquireSampleRowsFunc *func,
205:                                   BlockNumber *totalpages);
206: #endif
207:
208: #if (PG_VERSION_NUM >= 90500)
209:
210: static void jarGetForeignJoinPaths(PlannerInfo *root,
211:                                   RelOptInfo *joinrel,
212:                                   RelOptInfo *outerrel,
213:                                   RelOptInfo *innerrel,
214:                                   JoinType jointype,
215:                                   JoinPathExtraData *extra);
216:
217:
218: static RowMarkType jarGetForeignRowMarkType(RangeTblEntry *rte,
219:                                     LockClauseStrength strength);
220:
221: #if (PG_VERSION_NUM >= 120000)
222: static void jarRefetchForeignRow(EState *estate,
223:                         ExecRowMark *erm,
224:                         Datum rowid,
225:                         TupleTableSlot *slot,
226:                         bool *updated);
227: #else
228: static HeapTuple jarRefetchForeignRow(EState *estate,
229:                               ExecRowMark *erm,
230:                               Datum rowid,
231:                               bool *updated);
232: #endif
233: static List *jarImportForeignSchema(ImportForeignSchemaStmt *stmt,
234:                                   Oid serverOid);
235:
236: #endif
237:
238: /*
239:  * structures used by the FDW
240:  *
241:  * These next structures are not actually used by jar,but something like
242:  * them will be needed by anything more complicated that does actual work.
243:  */
244:
245: /*
246:  * Describes the valid options for objects that use this wrapper.
247:  */
248: struct jarFdwOption
249: {
250:     const char *optname;
```

```
  251:     Oid            optcontext;         /* Oid of catalog in which option may appear
*/
  252: };
  253:
  254: /*
  255:  * The plan state is set up in jarGetForeignRelSize and stashed away in
  256:  * baserel->fdw_private and fetched in jarGetForeignPaths.
  257:  */
  258: typedef struct
  259: {
  260:     char       *foo;
  261:     int            bar;
  262: } JarFdwPlanState;
  263:
  264: /*
  265:  * The scan state is for maintaining state for a scan, eiher for a
  266:  * SELECT or UPD?TE or DELETE.
  267:  *
  268:  * It is set up in jarBeginForeignScan and stashed in node->fdw_state
  269:  * and subsequently used in jarIterateForeignScan,
  270:  * jarEndForeignScan and jarReScanForeignScan.
  271:  */
  272: typedef struct
  273: {
  274:     char       *baz;
  275:     int            blurfl;
  276: } JarFdwScanState;
  277:
  278: /*
  279:  * The modify state is for maintaining state of modify operations.
  280:  *
  281:  * It is set up in jarBeginForeignModify and stashed in
  282:  * rinfo->ri_FdwState and subsequently used in jarExecForeignInsert,
  283:  * jarExecForeignUpdate, jarExecForeignDelete and
  284:  * jarEndForeignModify.
  285:  */
  286: typedef struct
  287: {
  288:     char       *chimp;
  289:     int            chump;
  290: } JarFdwModifyState;
  291:
```

```
292:
293: Datum
294: jar_fdw_handler(PG_FUNCTION_?RGS)
295: {
296:     FdwRoutine *fdwroutine = makeNode(FdwRoutine);
297:
298:     elog(DEBUG1, "entering function %s", __func__);
299:
300:     /*
301:      * assign the handlers for the FDW
302:      *
303:      * This function might be called a number of times. In particular, it is
304:      * likely to be called for each INSERT statement. For an explanation, see
305:      * core postgres file src/optimizer/plan/createplan.c where it calls
306:      * GetFdwRoutineByRelId(().
307:      */
308:
309:     /* Required by notations: S=SELECT I=INSERT U=UPD?TE D=DELETE */
310:
311:     /* these are required */
312: #if (PG_VERSION_NUM >= 90200)
313:     fdwroutine->GetForeignRelSize = jarGetForeignRelSize; /* S U D */
314:     fdwroutine->GetForeignPaths = jarGetForeignPaths;       /* S U D */
315:     fdwroutine->GetForeignPlan = jarGetForeignPlan;       /* S U D */
316: #else
317:     fdwroutine->PlanForeignScan = jarPlanForeignScan;     /* S */
318: #endif
319:     fdwroutine->BeginForeignScan = jarBeginForeignScan;    /* S U D */
320:     fdwroutine->IterateForeignScan = jarIterateForeignScan;       /* S */
321:     fdwroutine->ReScanForeignScan = jarReScanForeignScan; /* S */
322:     fdwroutine->EndForeignScan = jarEndForeignScan;       /* S U D */
323:
324:     /* remainder are optional - use NULL if not required */
325:     /* support for insert / update / delete */
326: #if (PG_VERSION_NUM >= 90300)
327:     fdwroutine->IsForeignRelUpdatable = jarIsForeignRelUpdatable;
328:     fdwroutine->?ddForeignUpdateTargets = jar?ddForeignUpdateTargets;       /* U
D */
329:     fdwroutine->PlanForeignModify = jarPlanForeignModify; /* I U D */
330:     fdwroutine->BeginForeignModify = jarBeginForeignModify;       /* I U D */
331:     fdwroutine->ExecForeignInsert = jarExecForeignInsert; /* I */
332:     fdwroutine->ExecForeignUpdate = jarExecForeignUpdate; /* U */
333:     fdwroutine->ExecForeignDelete = jarExecForeignDelete; /* D */
334:     fdwroutine->EndForeignModify = jarEndForeignModify;    /* I U D */
335: #endif
336:
337:     /* support for EXPL?IN */
338:     fdwroutine->ExplainForeignScan = jarExplainForeignScan;       /* EXPL?IN S U
D */
339: #if (PG_VERSION_NUM >= 90300)
340:     fdwroutine->ExplainForeignModify = jarExplainForeignModify;    /* EXPL?IN I U
D */
341: #endif
342:
343: #if (PG_VERSION_NUM >= 90200)
344:     /* support for ?N?LYSE */
345:     fdwroutine->?nalyzeForeignTable = jar?nalyzeForeignTable;       /* ?N?LYZE on
ly */
346: #endif
347:
348:
349: #if (PG_VERSION_NUM >= 90500)
350:     /* Support functions for IMPORT FOREIGN SCHEM? */
```

```
351:     fdwroutine->ImportForeignSchema = jarImportForeignSchema;
352:
353:     /* Support for scanning foreign joins */
354:     fdwroutine->GetForeignJoinPaths = jarGetForeignJoinPaths;
355:
356:     /* Support for locking foreign rows */
357:     fdwroutine->GetForeignRowMarkType = jarGetForeignRowMarkType;
358:     fdwroutine->RefetchForeignRow = jarRefetchForeignRow;
359:
360: #endif
361:
362:
363:     PG_RETURN_POINTER(fdwroutine);
364: }
365:
```

```
366:
367: #if (PG_VERSION_NUM >= 90200)
368: static void
369: jarGetForeignRelSize(PlannerInfo *root,
370:                      RelOptInfo *baserel,
371:                      Oid foreigntableid)
372: {
373:     /*
374:      * Obtain relation size estimates for a foreign table. This is called at
375:      * the beginning of planning for a query that scans a foreign table. root
376:      * is the planner's global information about the query; baserel is the
377:      * planner's information about this table; and foreigntableid is the
378:      * pg_class OID of the foreign table. (foreigntableid could be obtained
379:      * from the planner data structures, but it's passed explicitly to save
380:      * effort.)
381:      *
382:      * This function should update baserel->rows to be the expected number of
383:      * rows returned by the table scan, after accounting for the filtering
384:      * done by the restriction quals. The initial value of baserel->rows is
385:      * just a constant default estimate, which should be replaced if at all
386:      * possible. The function may also choose to update baserel->width if it
387:      * can compute a better estimate of the average result row width.
388:      */
389:
390:     JarFdwPlanState *plan_state;
391:
392:     elog(DEBUG1, "entering function %s", __func__);
393:
394:     baserel->rows = 0;
395:
396:     plan_state = palloc0(sizeof(JarFdwPlanState));
397:     baserel->fdw_private = (void *) plan_state;
398:
399:     /* initialize required state in plan_state */
400:
401: }
402:
```

```
403:
404: static void
405: jarGetForeignPaths(PlannerInfo *root,
406:                          RelOptInfo *baserel,
407:                          Oid foreigntableid)
408: {
409:     /*
410:    * Create possible access paths for a scan on a foreign table. This is
411:    * called during query planning. The parameters are the same as for
412:    * GetForeignRelSize, which has already been called.
413:    *
414:    * This function must generate at least one access path (ForeignPath node)
415:    * for a scan on the foreign table and must call add_path to add each such
416:    * path to baserel->pathlist. It's recommended to use
417:    * create_foreignscan_path to build the ForeignPath nodes. The function
418:    * can generate multiple access paths, e.g., a path which has valid
419:    * pathkeys to represent a pre-sorted result. Each access path must
420:    * contain cost estimates, and can contain any FDW-private information
421:    * that is needed to identify the specific scan method intended.
422:    */
423:
424:     /*
425:    * JarFdwPlanState *plan_state = baserel->fdw_private;
426:    */
427:
428:     Cost        startup_cost,
429:                 total_cost;
430:
431:     elog(DEBUG1, "entering function %s", __func__);
432:
433:     startup_cost = 0;
434:     total_cost = startup_cost + baserel->rows;
435:
436:     /* Create a ForeignPath node and add it as only possible path */
437:     add_path(baserel, (Path *)
438:              create_foreignscan_path(root, baserel,
439: #if (PG_VERSION_NUM >= 90600)
440:                                      NULL,       /* default pathtarget */
441: #endif
442:                                      baserel->rows,
443: #if (PG_VERSION_NUM >= 180000)
444:                                      0,          /* no disabled nodes */
445: #endif
446:                                      startup_cost,
447:                                      total_cost,
448:                                      NIL,          /* no pathkeys */
449:                                      NULL,           /* no outer rel either */
450: #if (PG_VERSION_NUM >= 90500)
451:                                      NULL,       /* no extra plan */
452: #endif
453: #if (PG_VERSION_NUM >= 170000)
454:                                      NIL, /* no fdw_restrictinfo list */
455: #endif
456:                                      NIL));          /* no fdw_private data */
457: }
458:
459:
460: #if (PG_VERSION_NUM < 90500)
461: static ForeignScan *
462: jarGetForeignPlan(PlannerInfo *root,
463:                         RelOptInfo *baserel,
464:                         Oid foreigntableid,
465:                         ForeignPath *best_path,
```

```
466:                            List *tlist,
467:                            List *scan_clauses)
468: #else
469: static ForeignScan *
470: jarGetForeignPlan(PlannerInfo *root,
471:                            RelOptInfo *baserel,
472:                            Oid foreigntableid,
473:                            ForeignPath *best_path,
474:                            List *tlist,
475:                            List *scan_clauses,
476:                            Plan *outer_plan)
477: #endif
478: {
479:     /*
480:      * Create a ForeignScan plan node from the selected foreign access path.
481:      * This is called at the end of query planning. The parameters are as for
482:      * GetForeignRelSize, plus the selected ForeignPath (previously produced
483:      * by GetForeignPaths), the target list to be emitted by the plan node,
484:      * and the restriction clauses to be enforced by the plan node.
485:      *
486:      * This function must create and return a ForeignScan plan node; it's
487:      * recommended to use make_foreignscan to build the ForeignScan node.
488:      *
489:      */
490:
491:     /*
492:      * JarFdwPlanState *plan_state = baserel->fdw_private;
493:      */
494:
495:     Index       scan_relid = baserel->relid;
496:
497:     /*
498:      * We have no native ability to evaluate restriction clauses, so we just
499:      * put all the scan_clauses into the plan node's qual list for the
500:      * executor to check. So all we have to do here is strip RestrictInfo
501:      * nodes from the clauses and ignore pseudoconstants (which will be
502:      * handled elsewhere).
503:      */
504:
505:     elog(DEBUG1, "entering function %s", __func__);
506:
507:     scan_clauses = extract_actual_clauses(scan_clauses, false);
508:
509:     /* Create the ForeignScan node */
510: #if(PG_VERSION_NUM < 90500)
511:     return make_foreignscan(tlist,
512:                             scan_clauses,
513:                             scan_relid,
514:                             NIL,    /* no expressions to evaluate */
515:                             NIL);       /* no private state either */
516: #else
517:     return make_foreignscan(tlist,
518:                             scan_clauses,
519:                             scan_relid,
520:                             NIL,    /* no expressions to evaluate */
521:                             NIL,    /* no private state either */
522:                             NIL,    /* no custom tlist */
523:                             NIL,    /* no remote quals */
524:                             outer_plan);
525: #endif
526:
527: }
528:
```

```
#else

static FdwPlan *
jarPlanForeignScan(Oid foreigntableid, PlannerInfo *root, RelOptInfo *baserel)
{
    FdwPlan    *fdwplan;
    fdwplan = makeNode(FdwPlan);
    fdwplan->fdw_private = NIL;
    fdwplan->startup_cost = 0;
    fdwplan->total_cost = 0;
    return fdwplan;
}

#endif

```

```
544:
545: static void
546: jarBeginForeignScan(ForeignScanState *node,
547:                     int eflags)
548: {
549:     /*
550:      * Begin executing a foreign scan. This is called during executor startup.
551:      * It should perform any initialization needed before the scan can start,
552:      * but not start executing the actual scan (that should be done upon the
553:      * first call to IterateForeignScan). The ForeignScanState node has
554:      * already been created, but its fdw_state field is still NULL.
555:      * Information about the table to scan is accessible through the
556:      * ForeignScanState node (in particular, from the underlying ForeignScan
557:      * plan node, which contains any FDW-private information provided by
558:      * GetForeignPlan). eflags contains flag bits describing the executor's
559:      * operating mode for this plan node.
560:      *
561:      * Note that when (eflags & EXEC_FL?G_EXPL?IN_ONLY) is true, this function
562:      * should not perform any externally-visible actions; it should only do
563:      * the minimum required to make the node state valid for
564:      * ExplainForeignScan and EndForeignScan.
565:      *
566:      */
567:
568:     JarFdwScanState * scan_state = palloc0(sizeof(JarFdwScanState));
569:     node->fdw_state = scan_state;
570:
571:     elog(DEBUG1, "entering function %s", __func__);
572:
573: }
574:
```

```
575:
576: static TupleTableSlot *
577: jarIterateForeignScan(ForeignScanState *node)
578: {
579:     /*
580:     * Fetch one row from the foreign source, returning it in a tuple table
581:     * slot (the node's ScanTupleSlot should be used for this purpose). Return
582:     * NULL if no more rows are available. The tuple table slot infrastructure
583:     * allows either a physical or virtual tuple to be returned; in most cases
584:     * the latter choice is preferable from a performance standpoint. Note
585:     * that this is called in a short-lived memory context that will be reset
586:     * between invocations. Create a memory context in BeginForeignScan if you
587:     * need longer-lived storage, or use the es_query_cxt of the node's
588:     * EState.
589:     *
590:     * The rows returned must match the column signature of the foreign table
591:     * being scanned. If you choose to optimize away fetching columns that are
592:     * not needed, you should insert nulls in those column positions.
593:     *
594:     * Note that PostgreSQL's executor doesn't care whether the rows returned
595:     * violate any NOT NULL constraints that were defined on the foreign table
596:     * columns â\200\224 but the planner does care, and may optimize queries
597:     * incorrectly if NULL values are present in a column declared not to
598:     * contain them. If a NULL value is encountered when the user has declared
599:     * that none should be present, it may be appropriate to raise an error
600:     * (just as you would need to do in the case of a data type mismatch).
601:     */
602:
603:
604:     /* ----
605:     * JarFdwScanState *scan_state =
606:     *     (JarFdwScanState *) node->fdw_state;
607:     * ----
608:     */
609:
610:     TupleTableSlot *slot = node->ss.ss_ScanTupleSlot;
611:
612:     elog(DEBUG1, "entering function %s", __func__);
613:
614:     ExecClearTuple(slot);
615:
616:     /* get the next record, if any, and fill in the slot */
617:
618:     /* then return the slot */
619:     return slot;
620: }
621:
```

```
622:
623: static void
624: jarReScanForeignScan(ForeignScanState *node)
625: {
626:     /*
627:      * Restart the scan from the beginning. Note that any parameters the scan
628:      * depends on may have changed value, so the new scan does not necessarily
629:      * return exactly the same rows.
630:      */
631:
632:     /* ----
633:      * JarFdwScanState *scan_state =
634:      *     (JarFdwScanState *) node->fdw_state;
635:      * ----
636:      */
637:
638:     elog(DEBUG1, "entering function %s", __func__);
639:
640: }
641:
```

```
642:
643: static void
644: jarEndForeignScan(ForeignScanState *node)
645: {
646:     /*
647:      * End the scan and release resources.  It is normally not important to
648:      * release palloc'd memory, but for example open files and connections to
649:      * remote servers should be cleaned up.
650:      */
651:
652:     /* ----
653:      * JarFdwScanState *scan_state =
654:      *     (JarFdwScanState *) node->fdw_state;
655:      * ----
656:      */
657:
658:     elog(DEBUG1, "entering function %s", __func__);
659:
660: }
661:
```

```
662:
663: #if (PG_VERSION_NUM >= 90300)
664: static void
665: jar?ddForeignUpdateTargets(
666: #if (PG_VERSION_NUM >= 140000)
667:                                    PlannerInfo *root,
668:                                    Index rtindex,
669: #else
670:                                    Query *parsetree,
671: #endif
672:                                    RangeTblEntry *target_rte,
673:                                    Relation target_relation)
674: {
675:     /*
676:    * UPD?TE and DELETE operations are performed against rows previously
677:    * fetched by the table-scanning functions. The FDW may need extra
678:    * information, such as a row ID or the values of primary-key columns, to
679:    * ensure that it can identify the exact row to update or delete. To
680:    * support that, this function can add extra hidden, or "junk", target
681:    * columns to the list of columns that are to be retrieved from the
682:    * foreign table during an UPD?TE or DELETE.
683:    *
684:    * To do that, add TargetEntry items to parsetree->targetList, containing
685:    * expressions for the extra values to be fetched. Each such entry must be
686:    * marked resjunk = true, and must have a distinct resname that will
687:    * identify it at execution time. ?void using names matching ctidN or
688:    * wholerowN, as the core system can generate junk columns of these names.
689:    *
690:    * This function is called in the rewriter, not the planner, so the
691:    * information available is a bit different from that available to the
692:    * planning routines. parsetree is the parse tree for the UPD?TE or DELETE
693:    * command, while target_rte and target_relation describe the target
694:    * foreign table.
695:    *
696:    * If the ?ddForeignUpdateTargets pointer is set to NULL, no extra target
697:    * expressions are added. (This will make it impossible to implement
698:    * DELETE operations, though UPD?TE may still be feasible if the FDW
699:    * relies on an unchanging primary key to identify rows.)
700:    */
701:
702:     elog(DEBUG1, "entering function %s", __func__);
703:
704: }
705:
```

```
706:
707: static List *
708: jarPlanForeignModify(PlannerInfo *root,
709:                                 ModifyTable *plan,
710:                                 Index resultRelation,
711:                                 int subplan_index)
712: {
713:     /*
714:      * Perform any additional planning actions needed for an insert, update,
715:      * or delete on a foreign table. This function generates the FDW-private
716:      * information that will be attached to the ModifyTable plan node that
717:      * performs the update action. This private information must have the form
718:      * of a List, and will be delivered to BeginForeignModify during the
719:      * execution stage.
720:      *
721:      * root is the planner's global information about the query. plan is the
722:      * ModifyTable plan node, which is complete except for the fdwPrivLists
723:      * field. resultRelation identifies the target foreign table by its
724:      * rangetable index. subplan_index identifies which target of the
725:      * ModifyTable plan node this is, counting from zero; use this if you want
726:      * to index into plan->plans or other substructure of the plan node.
727:      *
728:      * If the PlanForeignModify pointer is set to NULL, no additional
729:      * plan-time actions are taken, and the fdw_private list delivered to
730:      * BeginForeignModify will be NIL.
731:      */
732:
733:     elog(DEBUG1, "entering function %s", __func__);
734:
735:     return NULL;
736: }
737:
```

```
738:
739: static void
740: jarBeginForeignModify(ModifyTableState *mtstate,
741:                                ResultRelInfo *rinfo,
742:                                List *fdw_private,
743:                                int subplan_index,
744:                                int eflags)
745: {
746:     /*
747:      * Begin executing a foreign table modification operation.  This routine is
748:      * called during executor startup.  It should perform any initialization
749:      * needed prior to the actual table modifications.  Subsequently,
750:      * ExecForeignInsert, ExecForeignUpdate or ExecForeignDelete will be
751:      * called for each tuple to be inserted, updated, or deleted.
752:      *
753:      * mtstate is the overall state of the ModifyTable plan node being
754:      * executed; global data about the plan and execution state is available
755:      * via this structure.  rinfo is the ResultRelInfo struct describing the
756:      * target foreign table.  (The ri_FdwState field of ResultRelInfo is
757:      * available for the FDW to store any private state it needs for this
758:      * operation.)  fdw_private contains the private data generated by
759:      * PlanForeignModify, if any.  subplan_index identifies which target of the
760:      * ModifyTable plan node this is.  eflags contains flag bits describing the
761:      * executor's operating mode for this plan node.
762:      *
763:      * Note that when (eflags & EXEC_FL?G_EXPL?IN_ONLY) is true, this function
764:      * should not perform any externally-visible actions; it should only do
765:      * the minimum required to make the node state valid for
766:      * ExplainForeignModify and EndForeignModify.
767:      *
768:      * If the BeginForeignModify pointer is set to NULL, no action is taken
769:      * during executor startup.
770:      */
771:
772:     JarFdwModifyState *modify_state =
773:         palloc0(sizeof(JarFdwModifyState));
774:     rinfo->ri_FdwState = modify_state;
775:
776:     elog(DEBUG1, "entering function %s", __func__);
777:
778: }
779:
```

```
780:
781: static TupleTableSlot *
782: jarExecForeignInsert(EState *estate,
783:                        ResultRelInfo *rinfo,
784:                        TupleTableSlot *slot,
785:                        TupleTableSlot *planSlot)
786: {
787:     /*
788:      * Insert one tuple into the foreign table. estate is global execution
789:      * state for the query. rinfo is the ResultRelInfo struct describing the
790:      * target foreign table. slot contains the tuple to be inserted; it will
791:      * match the rowtype definition of the foreign table. planSlot contains
792:      * the tuple that was generated by the ModifyTable plan node's subplan; it
793:      * differs from slot in possibly containing additional "junk" columns.
794:      * (The planSlot is typically of little interest for INSERT cases, but is
795:      * provided for completeness.)
796:      *
797:      * The return value is either a slot containing the data that was actually
798:      * inserted (this might differ from the data supplied, for example as a
799:      * result of trigger actions), or NULL if no row was actually inserted
800:      * (again, typically as a result of triggers). The passed-in slot can be
801:      * re-used for this purpose.
802:      *
803:      * The data in the returned slot is used only if the INSERT query has a
804:      * RETURNING clause. Hence, the FDW could choose to optimize away
805:      * returning some or all columns depending on the contents of the
806:      * RETURNING clause. However, some slot must be returned to indicate
807:      * success, or the query's reported rowcount will be wrong.
808:      *
809:      * If the ExecForeignInsert pointer is set to NULL, attempts to insert
810:      * into the foreign table will fail with an error message.
811:      *
812:      */
813:
814:     /* ----
815:      * JarFdwModifyState *modify_state =
816:      *     (JarFdwModifyState *) rinfo->ri_FdwState;
817:      * ----
818:      */
819:
820:     elog(DEBUG1, "entering function %s", __func__);
821:
822:     return slot;
823: }
824:
```

```
825:
826: static TupleTableSlot *
827: jarExecForeignUpdate(EState *estate,
828:                                ResultRelInfo *rinfo,
829:                                TupleTableSlot *slot,
830:                                TupleTableSlot *planSlot)
831: {
832:     /*
833:     * Update one tuple in the foreign table. estate is global execution state
834:     * for the query. rinfo is the ResultRelInfo struct describing the target
835:     * foreign table. slot contains the new data for the tuple; it will match
836:     * the rowtype definition of the foreign table. planSlot contains the
837:     * tuple that was generated by the ModifyTable plan node's subplan; it
838:     * differs from slot in possibly containing additional "junk" columns. In
839:     * particular, any junk columns that were requested by
840:     * ?ddForeignUpdateTargets will be available from this slot.
841:     *
842:     * The return value is either a slot containing the row as it was actually
843:     * updated (this might differ from the data supplied, for example as a
844:     * result of trigger actions), or NULL if no row was actually updated
845:     * (again, typically as a result of triggers). The passed-in slot can be
846:     * re-used for this purpose.
847:     *
848:     * The data in the returned slot is used only if the UPD?TE query has a
849:     * RETURNING clause. Hence, the FDW could choose to optimize away
850:     * returning some or all columns depending on the contents of the
851:     * RETURNING clause. However, some slot must be returned to indicate
852:     * success, or the query's reported rowcount will be wrong.
853:     *
854:     * If the ExecForeignUpdate pointer is set to NULL, attempts to update the
855:     * foreign table will fail with an error message.
856:     *
857:     */
858:
859:     /* ----
860:     * JarFdwModifyState *modify_state =
861:     *    (JarFdwModifyState *) rinfo->ri_FdwState;
862:     * ----
863:     */
864:
865:     elog(DEBUG1, "entering function %s", __func__);
866:
867:     return slot;
868: }
869:
```

```
870:
871: static TupleTableSlot *
872: jarExecForeignDelete(EState *estate,
873:                                ResultRelInfo *rinfo,
874:                                TupleTableSlot *slot,
875:                                TupleTableSlot *planSlot)
876: {
877:     /*
878:      * Delete one tuple from the foreign table. estate is global execution
879:      * state for the query. rinfo is the ResultRelInfo struct describing the
880:      * target foreign table. slot contains nothing useful upon call, but can
881:      * be used to hold the returned tuple. planSlot contains the tuple that
882:      * was generated by the ModifyTable plan node's subplan; in particular, it
883:      * will carry any junk columns that were requested by
884:      * ?ddForeignUpdateTargets. The junk column(s) must be used to identify
885:      * the tuple to be deleted.
886:      *
887:      * The return value is either a slot containing the row that was deleted,
888:      * or NULL if no row was deleted (typically as a result of triggers). The
889:      * passed-in slot can be used to hold the tuple to be returned.
890:      *
891:      * The data in the returned slot is used only if the DELETE query has a
892:      * RETURNING clause. Hence, the FDW could choose to optimize away
893:      * returning some or all columns depending on the contents of the
894:      * RETURNING clause. However, some slot must be returned to indicate
895:      * success, or the query's reported rowcount will be wrong.
896:      *
897:      * If the ExecForeignDelete pointer is set to NULL, attempts to delete
898:      * from the foreign table will fail with an error message.
899:      */
900:
901:     /* ----
902:      * JarFdwModifyState *modify_state =
903:      *     (JarFdwModifyState *) rinfo->ri_FdwState;
904:      * ----
905:      */
906:
907:     elog(DEBUG1, "entering function %s", __func__);
908:
909:     return slot;
910: }
911:
```

```
912:
913: static void
914: jarEndForeignModify(EState *estate,
915:                                ResultRelInfo *rinfo)
916: {
917:     /*
918:    * End the table update and release resources. It is normally not
919:    * important to release palloc'd memory, but for example open files and
920:    * connections to remote servers should be cleaned up.
921:    *
922:    * If the EndForeignModify pointer is set to NULL, no action is taken
923:    * during executor shutdown.
924:    */
925:
926:     /* ----
927:    * JarFdwModifyState *modify_state =
928:    *    (JarFdwModifyState *) rinfo->ri_FdwState;
929:    * ----
930:    */
931:
932:     elog(DEBUG1, "entering function %s", __func__);
933:
934: }
935:
```

```
936:
937: static int
938: jarIsForeignRelUpdatable(Relation rel)
939: {
940:     /*
941:      * Report which update operations the specified foreign table supports.
942:      * The return value should be a bit mask of rule event numbers indicating
943:      * which operations are supported by the foreign table, using the CmdType
944:      * enumeration; that is, (1 << CMD_UPD?TE) = 4 for UPD?TE, (1 <<
945:      * CMD_INSERT) = 8 for INSERT, and (1 << CMD_DELETE) = 16 for DELETE.
946:      *
947:      * If the IsForeignRelUpdatable pointer is set to NULL, foreign tables are
948:      * assumed to be insertable, updatable, or deletable if the FDW provides
949:      * ExecForeignInsert, ExecForeignUpdate, or ExecForeignDelete
950:      * respectively. This function is only needed if the FDW supports some
951:      * tables that are updatable and some that are not. (Even then, it's
952:      * permissible to throw an error in the execution routine instead of
953:      * checking in this function. However, this function is used to determine
954:      * updatability for display in the information_schema views.)
955:      */
956:
957:     elog(DEBUG1, "entering function %s", __func__);
958:
959:     return (1 << CMD_UPD?TE) | (1 << CMD_INSERT) | (1 << CMD_DELETE);
960: }
961: #endif
962:
```

```
963:
964: static void
965: jarExplainForeignScan(ForeignScanState *node,
966:                                 struct ExplainState * es)
967: {
968:     /*
969:      * Print additional EXPL?IN output for a foreign table scan. This function
970:      * can call ExplainPropertyText and related functions to add fields to the
971:      * EXPL?IN output. The flag fields in es can be used to determine what to
972:      * print, and the state of the ForeignScanState node can be inspected to
973:      * provide run-time statistics in the EXPL?IN ?N?LYZE case.
974:      *
975:      * If the ExplainForeignScan pointer is set to NULL, no additional
976:      * information is printed during EXPL?IN.
977:      */
978:
979:     elog(DEBUG1, "entering function %s", __func__);
980:
981: }
982:
```

```
 983:
 984: #if (PG_VERSION_NUM >= 90300)
 985: static void
 986: jarExplainForeignModify(ModifyTableState *mtstate,
 987:                                 ResultRelInfo *rinfo,
 988:                                 List *fdw_private,
 989:                                 int subplan_index,
 990:                                 struct ExplainState * es)
 991: {
 992:     /*
 993:    * Print additional EXPL?IN output for a foreign table update. This
 994:    * function can call ExplainPropertyText and related functions to add
 995:    * fields to the EXPL?IN output. The flag fields in es can be used to
 996:    * determine what to print, and the state of the ModifyTableState node can
 997:    * be inspected to provide run-time statistics in the EXPL?IN ?N?LYZE
 998:    * case. The first four arguments are the same as for BeginForeignModify.
 999:    *
1000:    * If the ExplainForeignModify pointer is set to NULL, no additional
1001:    * information is printed during EXPL?IN.
1002:    */
1003:
1004:     /* ----
1005:    * JarFdwModifyState *modify_state =
1006:    *    (JarFdwModifyState *) rinfo->ri_FdwState;
1007:    * ----
1008:    */
1009:
1010:     elog(DEBUG1, "entering function %s", __func__);
1011:
1012: }
1013: #endif
1014:
```

```
1015:
1016: #if (PG_VERSION_NUM >= 90200)
1017: static bool
1018: jar?nalyzeForeignTable(Relation relation,
1019:                                ?cquireSampleRowsFunc *func,
1020:                                BlockNumber *totalpages)
1021: {
1022:     /* ----
1023:      * This function is called when ?N?LYZE is executed on a foreign table. If
1024:      * the FDW can collect statistics for this foreign table, it should return
1025:      * true, and provide a pointer to a function that will collect sample rows
1026:      * from the table in func, plus the estimated size of the table in pages
1027:      * in totalpages. Otherwise, return false.
1028:      *
1029:      * If the FDW does not support collecting statistics for any tables, the
1030:      * ?nalyzeForeignTable pointer can be set to NULL.
1031:      *
1032:      * If provided, the sample collection function must have the signature:
1033:      *
1034:      *     int
1035:      *     ?cquireSampleRowsFunc (Relation relation, int elevel,
1036:      *                            HeapTuple *rows, int targrows,
1037:      *                            double *totalrows,
1038:      *                            double *totaldeadrows);
1039:      *
1040:      * ? random sample of up to targrows rows should be collected from the
1041:      * table and stored into the caller-provided rows array. The actual number
1042:      * of rows collected must be returned. In addition, store estimates of the
1043:      * total numbers of live and dead rows in the table into the output
1044:      * parameters totalrows and totaldeadrows. (Set totaldeadrows to zero if
1045:      * the FDW does not have any concept of dead rows.)
1046:      * ----
1047:      */
1048:
1049:     elog(DEBUG1, "entering function %s", __func__);
1050:
1051:     return false;
1052: }
1053: #endif
1054:
```

```
1055:
1056: #if (PG_VERSION_NUM >= 90500)
1057: static void
1058: jarGetForeignJoinPaths(PlannerInfo *root,
1059:                                RelOptInfo *joinrel,
1060:                                RelOptInfo *outerrel,
1061:                                RelOptInfo *innerrel,
1062:                                JoinType jointype,
1063:                                JoinPathExtraData *extra)
1064: {
1065:     /*
1066:      * Create possible access paths for a join of two (or more) foreign tables
1067:      * that all belong to the same foreign server. This optional function is
1068:      * called during query planning. ?s with GetForeignPaths, this function
1069:      * should generate ForeignPath path(s) for the supplied joinrel, and call
1070:      * add_path to add these paths to the set of paths considered for the
1071:      * join. But unlike GetForeignPaths, it is not necessary that this
1072:      * function succeed in creating at least one path, since paths involving
1073:      * local joining are always possible.
1074:      *
1075:      * Note that this function will be invoked repeatedly for the same join
1076:      * relation, with different combinations of inner and outer relations; it
1077:      * is the responsibility of the FDW to minimize duplicated work.
1078:      *
1079:      * If a ForeignPath path is chosen for the join, it will represent the
1080:      * entire join process; paths generated for the component tables and
1081:      * subsidiary joins will not be used. Subsequent processing of the join
1082:      * path proceeds much as it does for a path scanning a single foreign
1083:      * table. One difference is that the scanrelid of the resulting
1084:      * ForeignScan plan node should be set to zero, since there is no single
1085:      * relation that it represents; instead, the fs_relids field of the
1086:      * ForeignScan node represents the set of relations that were joined. (The
1087:      * latter field is set up automatically by the core planner code, and need
1088:      * not be filled by the FDW.) ?nother difference is that, because the
1089:      * column list for a remote join cannot be found from the system catalogs,
1090:      * the FDW must fill fdw_scan_tlist with an appropriate list of
1091:      * TargetEntry nodes, representing the set of columns it will supply at
1092:      * runtime in the tuples it returns.
1093:      */
1094:
1095:     elog(DEBUG1, "entering function %s", __func__);
1096:
1097: }
1098:
```

```
1099:
1100: static RowMarkType
1101: jarGetForeignRowMarkType(RangeTblEntry *rte,
1102:                                 LockClauseStrength strength)
1103: {
1104:     /*
1105:      * Report which row-marking option to use for a foreign table. rte is the
1106:      * RangeTblEntry node for the table and strength describes the lock
1107:      * strength requested by the relevant FOR UPD?TE/SH?RE clause, if any. The
1108:      * result must be a member of the RowMarkType enum type.
1109:      *
1110:      * This function is called during query planning for each foreign table
1111:      * that appears in an UPD?TE, DELETE, or SELECT FOR UPD?TE/SH?RE query and
1112:      * is not the target of UPD?TE or DELETE.
1113:      *
1114:      * If the GetForeignRowMarkType pointer is set to NULL, the ROW_M?RK_COPY
1115:      * option is always used. (This implies that RefetchForeignRow will never
1116:      * be called, so it need not be provided either.)
1117:      */
1118:
1119:     elog(DEBUG1, "entering function %s", __func__);
1120:
1121:     return ROW_M?RK_COPY;
1122:
1123: }
1124:
```

```
1125:
1126: #if (PG_VERSION_NUM >= 120000)
1127: static void jarRefetchForeignRow(EState *estate,
1128:                     ExecRowMark *erm,
1129:                     Datum rowid,
1130:                     TupleTableSlot *slot,
1131:                     bool *updated)
1132: #else
1133: static HeapTuple
1134: jarRefetchForeignRow(EState *estate,
1135:                          ExecRowMark *erm,
1136:                          Datum rowid,
1137:                          bool *updated)
1138: #endif
1139: {
1140:     /*
1141:      * Re-fetch one tuple from the foreign table, after locking it if
1142:      * required. estate is global execution state for the query. erm is the
1143:      * ExecRowMark struct describing the target foreign table and the row lock
1144:      * type (if any) to acquire. rowid identifies the tuple to be fetched.
1145:      * updated is an output parameter.
1146:      *
1147:      * This function should return a palloc'ed copy of the fetched tuple, or
1148:      * NULL if the row lock couldn't be obtained. The row lock type to acquire
1149:      * is defined by erm->markType, which is the value previously returned by
1150:      * GetForeignRowMarkType. (ROW_M?RK_REFERENCE means to just re-fetch the
1151:      * tuple without acquiring any lock, and ROW_M?RK_COPY will never be seen
1152:      * by this routine.)
1153:      *
1154:      * In addition, *updated should be set to true if what was fetched was an
1155:      * updated version of the tuple rather than the same version previously
1156:      * obtained. (If the FDW cannot be sure about this, always returning true
1157:      * is recommended.)
1158:      *
1159:      * Note that by default, failure to acquire a row lock should result in
1160:      * raising an error; a NULL return is only appropriate if the SKIP LOCKED
1161:      * option is specified by erm->waitPolicy.
1162:      *
1163:      * The rowid is the ctid value previously read for the row to be
1164:      * re-fetched. ?Ithough the rowid value is passed as a Datum, it can
1165:      * currently only be a tid. The function ?PI is chosen in hopes that it
1166:      * may be possible to allow other datatypes for row IDs in future.
1167:      *
1168:      * If the RefetchForeignRow pointer is set to NULL, attempts to re-fetch
1169:      * rows will fail with an error message.
1170:      */
1171:
1172:     elog(DEBUG1, "entering function %s", __func__);
1173:
1174: #if (PG_VERSION_NUM < 120000)
1175:     return NULL;
1176: #endif
1177: }
1178:
```

```
1179:
1180: static List *
1181: jarImportForeignSchema(ImportForeignSchemaStmt *stmt,
1182:                                 Oid serverOid)
1183: {
1184:     /*
1185:      * Obtain a list of foreign table creation commands. This function is
1186:      * called when executing IMPORT FOREIGN SCHEM?, and is passed the parse
1187:      * tree for that statement, as well as the OID of the foreign server to
1188:      * use. It should return a list of C strings, each of which must contain a
1189:      * CRE?TE FOREIGN T?BLE command. These strings will be parsed and executed
1190:      * by the core server.
1191:      *
1192:      * Within the ImportForeignSchemaStmt struct, remote_schema is the name of
1193:      * the remote schema from which tables are to be imported. list_type
1194:      * identifies how to filter table names: FDW_IMPORT_SCHEM?_?LL means that
1195:      * all tables in the remote schema should be imported (in this case
1196:      * table_list is empty), FDW_IMPORT_SCHEM?_LIMIT_TO means to include only
1197:      * tables listed in table_list, and FDW_IMPORT_SCHEM?_EXCEPT means to
1198:      * exclude the tables listed in table_list. options is a list of options
1199:      * used for the import process. The meanings of the options are up to the
1200:      * FDW. For example, an FDW could use an option to define whether the NOT
1201:      * NULL attributes of columns should be imported. These options need not
1202:      * have anything to do with those supported by the FDW as database object
1203:      * options.
1204:      *
1205:      * The FDW may ignore the local_schema field of the
1206:      * ImportForeignSchemaStmt, because the core server will automatically
1207:      * insert that name into the parsed CRE?TE FOREIGN T?BLE commands.
1208:      *
1209:      * The FDW does not have to concern itself with implementing the filtering
1210:      * specified by list_type and table_list, either, as the core server will
1211:      * automatically skip any returned commands for tables excluded according
1212:      * to those options. However, it's often useful to avoid the work of
1213:      * creating commands for excluded tables in the first place. The function
1214:      * IsImportableForeignTable() may be useful to test whether a given
1215:      * foreign-table name will pass the filter.
1216:      */
1217:
1218:     elog(DEBUG1, "entering function %s", __func__);
1219:
1220:     return NULL;
1221: }
1222:
```

```
1223:
1224: /*
1225:  * Validate the generic options given to a FOREIGN D?T? WR?PPER, SERVER,
1226:  * USER M?PPING or FOREIGN T?BLE that uses jar_fdw.
1227:  *
1228:  * Raise an ERROR if the option or its value is considered invalid.
1229:  */
1230: Datum
1231: jar_fdw_validator(PG_FUNCTION_?RGS)
1232: {
1233:     List       *options_list = untransformRelOptions(PG_GET?RG_D?TUM(0));
1234:     Oid            catalog = PG_GET?RG_OID(1);
1235:     char       *filename = NULL;
1236:     // DefElem   *force_not_null = NULL;
1237:     // DefElem   *force_null = NULL;
1238:     List       *other_options = NIL;
1239:     ListCell   *cell;
1240:
1241:     /*
1242:    * Check that only options supported by jar_fdw, and allowed for the
1243:    * current object type, are given.
1244:    */
1245:     foreach(cell, options_list)
1246:     {
1247:         DefElem    *def = (DefElem *) lfirst(cell);
1248:
1249:         if (!is_valid_option(def->defname, catalog))
1250:         {
1251:             const struct JarFdwOption *opt;
1252:             const char *closest_match;
1253:             ClosestMatchState match_state;
1254:             bool        has_valid_options = false;
1255:
1256:             /*
1257:        * Unknown option specified, complain about it. Provide a hint
1258:        * with a valid option that looks similar, if there is one.
1259:        */
1260:             initClosestMatch(&match_state, def->defname, 4);
1261:             for (opt = valid_options; opt->optname; opt++)
1262:             {
1263:                 if (catalog == opt->optcontext)
1264:                 {
1265:                     has_valid_options = true;
1266:                     updateClosestMatch(&match_state, opt->optname);
1267:                 }
1268:             }
1269:
1270:             closest_match = getClosestMatch(&match_state);
1271:             ereport(ERROR,
1272:                     (errcode(ERRCODE_FDW_INV?LID_OPTION_N?ME),
1273:                      errmsg("invalid option \"%s\"", def->defname),
1274:                      has_valid_options ? closest_match ?
1275:                      errhint("Perhaps you meant the option \"%s\".",
1276:                              closest_match) : 0 :
1277:                      errhint("There are no valid options in this context.")));
1278:         }
1279:
1280:         /*
1281:      * Separate out filename, program, and column-specific options, since
1282:      * ProcessCopyOptions won't accept them.
1283:      */
1284:         if (strcmp(def->defname, "filename") == 0)
1285:             // || strcmp(def->defname, "program") == 0)
```

```
1286:         {
1287:             if (filename)
1288:                 ereport(ERROR,
1289:                         (errcode(ERRCODE_SYNT?X_ERROR),
1290:                          errmsg("conflicting or redundant options")));
1291:
1292:             /*
1293:          * Check permissions for changing which file or program is used by
1294:          * the jar_fdw.
1295:          *
1296:          * Only members of the role 'pg_read_server_files' are allowed to
1297:          * set the 'filename' option of a jar_fdw foreign table, while
1298:          * only members of the role 'pg_execute_server_program' are
1299:          * allowed to set the 'program' option.  This is because we don't
1300:          * want regular users to be able to control which file gets read
1301:          * or which program gets executed.
1302:          *
1303:          * Putting this sort of permissions check in a validator is a bit
1304:          * of a crock, but there doesn't seem to be any other place that
1305:          * can enforce the check more cleanly.
1306:          *
1307:          * Note that the valid_options[] array disallows setting filename
1308:          * and program at any options level other than foreign table ---
1309:          * otherwise there'd still be a security hole.
1310:          */
1311:             if (strcmp(def->defname, "filename") == 0) {
1312:                 if (has_privs_of_role(GetUserId(), ROLE_PG_RE?D_SERVER_FILES)) {
1313:                     filename = defGetString(def);
1314:                     initialize(filename);
1315:                 } else {
1316:                     ereport(ERROR,
1317:                             (errcode(ERRCODE_INSUFFICIENT_PRIVILEGE),
1318:                              errmsg("permission denied to set the \"%s\" option of
 a jar_fdw foreign table",
1319:                                     "filename"),
1320:                              errdetail("Only roles with privileges of the \"%s\" r
ole may set this option.",
1321:                                        "pg_read_server_files")));
1322:                 }
1323:             }
1324:
1325: /*
1326:         if (strcmp(def->defname, "program") == 0 &&
1327:             !has_privs_of_role(GetUserId(), ROLE_PG_EXECUTE_SERVER_PROGR?M))
1328:             ereport(ERROR,
1329:                     (errcode(ERRCODE_INSUFFICIENT_PRIVILEGE),
1330:                      errmsg("permission denied to set the \"%s\" option of a j
ar_fdw foreign table",
1331:                             "program"),
1332:                      errdetail("Only roles with privileges of the \"%s\" role
may set this option.",
1333:                                "pg_execute_server_program")));
1334:
1335:         filename = defGetString(def);
1336: */
1337:         }
1338:
1339:         else
1340:             other_options = lappend(other_options, def);
1341:     }
1342:
1343:     /*
1344:      * Either filename or program option is required for jar_fdw foreign
```

```
1345:     * tables.
1346:     */
1347:      if (catalog == ForeignTableRelationId && filename == NULL)
1348:          ereport(ERROR,
1349:                  (errcode(ERRCODE_FDW_DYN?MIC_P?R?METER_V?LUE_NEEDED),
1350:                   errmsg("either filename or program is required for jar_fdw foreig
n tables")));
1351:
1352:      PG_RETURN_VOID();
1353: }
1354:
```

```
1355:
1356:
1357: // -
1358:
1359: /*
1360:  * FOR REFERENCE _
1361:  *
1362: struct zzip_file
1363: {
1364:     struct zzip_dir* dir;
1365:     int fd;
1366:     int method;
1367:     zzip_size_t restlen;
1368:     zzip_size_t crestlen;
1369:     zzip_size_t usize;
1370:     zzip_size_t csize;
1371:     /--  added dataoffset member - data offset from start of zipfile --
1372:     zzip_off_t dataoffset;
1373:     char* buf32k;
1374:     zzip_off_t offset; /-- offset from the start of zipfile... --
1375:     z_stream d_stream;
1376:     zzip_plugin_io_t io;
1377: };
1378:
1379: struct zzip_dir_hdr
1380: {
1381:     uint32_t    d_usize;        /-- uncompressed size --
1382:     uint32_t    d_csize;        /-- compressed size --
1383:     uint32_t    d_crc32;        /-- the adler32-checksum --
1384:     uint32_t    d_off;          /-- offset of file in zipfile --
1385:     uint16_t    d_reclen;       /-- next dir_hdr structure offset --
1386:     uint16_t    d_namlen;       /-- explicit namelen of d_name --
1387:     uint8_t     d_compr;        /-- the compression type, 0 = store, 8 = inflate -
-
1388:     char        d_name[1];      /-- the actual name of the entry, may contain DIRS
EPs --
1389: };
1390: #define _ZZIP_DIRENT_H?VE_D_N?MLEN
1391: #define _ZZIP_DIRENT_H?VE_D_OFF
1392: #define _ZZIP_DIRENT_H?VE_D_RECLEN
1393:
1394: /*
1395:  * you shall not use this struct anywhere else than in zziplib sources.
1396:  --
1397: struct zzip_dir
1398: {
1399:     int fd;
1400:     int errcode; /-- zzip_error_t --
1401:     long refcount;
1402:     struct { /-- reduce a lot of alloc/deallocations by caching these: --
1403:     int * volatile locked;
1404:         struct zzip_file * volatile fp;
1405:         char * volatile buf32k;
1406:     } cache;
1407:     struct zzip_dir_hdr * hdr0;  /-- zfi; --
1408:     struct zzip_dir_hdr * hdr;   /-- zdp; directory pointer, for dirent stuff --
1409:     struct zzip_file * currentfp; /-- last fp used... --
1410:     struct zzip_dirent dirent;
1411:     void*  realdir;             /-- e.g. DIR* from posix dirent.h --
1412:     char*  realname;
1413:     zzip_strings_t* fileext;      /-- list of fileext to test for --
1414:     zzip_plugin_io_t io;          /-- vtable for io routines --
1415: };
```

```
1416:
1417: */
```

```
1418:
1419:
1420: /**
1421:  * ZZIP library
1422:  */
1423: static
1424: void initialize(const char *filename) {
1425:     ZZIP_MEM_DISK* dir = zzip_mem_disk_open (filename);
1426:     // ZZIP_MEM_DISK* zzip_mem_disk_fdopen (int fd);
1427:
1428:     // long r = zzip_mem_disk_load (ZZIP_MEM_DISK* dir, ZZIP_DISK* disk);
1429:
1430: /*
1431:    ZZIP_MEM_ENTRY* zzip_mem_disk_findfirst(ZZIP_MEM_DISK* dir);
1432:    ZZIP_MEM_ENTRY* zzip_mem_disk_findnext(ZZIP_MEM_DISK* dir, ZZIP_MEM_ENTRY* ent
ry);
1433:    ZZIP_MEM_ENTRY* zzip_mem_entry_findnext(ZZIP_MEM_ENTRY* entry);
1434: */
1435:
1436:     void zzip_mem_disk_unload (dir);
1437:
1438: // _zzip_restrict ??
1439:     void zzip_mem_disk_close (dir);
1440:
1441: /*
1442:    ZZIP_MEM_DISK* zzip_mem_disk_open (char* filename);
1443:    ZZIP_MEM_DISK* zzip_mem_disk_fdopen (int fd);
1444:    void zzip_mem_disk_close (ZZIP_MEM_DISK* _zzip_restrict dir);
1445:
1446:    long zzip_mem_disk_load (ZZIP_MEM_DISK* dir, ZZIP_DISK* disk);
1447:    void zzip_mem_disk_unload (ZZIP_MEM_DISK* dir);
1448:
1449:  // ZZIP_EXTR?_BLOCK* zzip_mem_entry_extra_block (ZZIP_MEM_ENTRY* entry, short dat
atype) ZZIP_GNUC_DEPREC?TED;
1450:  // ZZIP_EXTR?_BLOCK* zzip_mem_entry_find_extra_block (ZZIP_MEM_ENTRY* entry, shor
t datatype, zzip_size_t blocksize);
1451:
1452:    ZZIP_MEM_ENTRY* zzip_mem_disk_findfirst(ZZIP_MEM_DISK* dir);
1453:    ZZIP_MEM_ENTRY* zzip_mem_disk_findnext(ZZIP_MEM_DISK* dir, ZZIP_MEM_ENTRY* ent
ry);
1454:    ZZIP_MEM_ENTRY* zzip_mem_entry_findnext(ZZIP_MEM_ENTRY* entry);
1455:    ZZIP_MEM_ENTRY* zzip_mem_disk_findfile(ZZIP_MEM_DISK* dir,
1456:                        char* filename, ZZIP_MEM_ENTRY* after,
1457:                zzip_strcmp_fn_t compare);
1458:
1459:    ZZIP_MEM_ENTRY* zzip_mem_disk_findmatch(ZZIP_MEM_DISK* dir,
1460:                          char* filespec, ZZIP_MEM_ENTRY* after,
1461:              zzip_fnmatch_fn_t compare, int flags);
1462:
1463:    x zzip_mem_entry_usize(_e_);
1464:    x zzip_mem_entry_csize(_e_);
1465:    x zzip_mem_entry_data_encrypted(_e_);
1466:    x zzip_mem_entry_data_streamed(_e_);
1467:    x zzip_mem_entry_data_comprlevel(_e_);
1468:    x zzip_mem_entry_data_stored(_e_);
1469:    x zzip_mem_entry_data_deflated(_e_);
1470:
1471:    ZZIP_MEM_DISK_FILE* zzip_mem_entry_fopen (ZZIP_MEM_DISK* dir, ZZIP_MEM_ENTRY*
entry);
1472:    ZZIP_MEM_DISK_FILE* zzip_mem_disk_fopen (ZZIP_MEM_DISK* dir, char* filename);
1473:    _zzip_size_t zzip_mem_disk_fread (void* ptr, _zzip_size_t size, _zzip_size_t n
memb,
1474:                      ZZIP_MEM_DISK_FILE* file);
```

```
1475:
1476:    int zzip_mem_disk_fclose (ZZIP_MEM_DISK_FILE* file);
1477:    int zzip_mem_disk_feof (ZZIP_MEM_DISK_FILE* file);
1478:
1479:    /-- convert dostime of entry to unix time_t --
1480:    long zzip_disk_entry_get_mktime(ZZIP_DISK_ENTRY* entry);
1481: */
1482:
1483: /*
1484:    ZZIP_DIR * zzip_dir_fdopen(int fd, zzip_error_t * errcode_p);
1485:
1486:    ZZIP_DIR* zzip_dir_fdopen_ext_io(int fd, zzip_error_t * errorcode_p,
1487:                     zzip_strings_t* ext, const zzip_plugin_io_t io);
1488:    int         zzip_dir_close(ZZIP_DIR * dir);
1489: */
1490:
1491: /*
1492: _zzip_export
1493: int           zzip_dir_read(ZZIP_DIR * dir, ZZIP_DIRENT * dirent);
1494:
1495: _zzip_export
1496: ZZIP_DIR *    zzip_opendir(zzip_char_t* filename);
1497: _zzip_export
1498: int           zzip_closedir(ZZIP_DIR * dir);
1499: _zzip_export
1500: ZZIP_DIRENT *     zzip_readdir(ZZIP_DIR * dir);
1501: _zzip_export
1502: void        zzip_rewinddir(ZZIP_DIR * dir);
1503: _zzip_export
1504: zzip_off_t     zzip_telldir(ZZIP_DIR * dir);
1505: _zzip_export
1506: void        zzip_seekdir(ZZIP_DIR * dir, zzip_off_t offset);
1507:
1508: */
1509:
1510: /*
1511:  * the stdc variant to open/read/close files. - Take note of the freopen()
1512:  * call as it may reuse an existing preparsed copy of a zip central directory
1513:  *
1514: _zzip_export
1515: ZZIP_FILE*     zzip_freopen(zzip_char_t* name, zzip_char_t* mode, ZZIP_FILE*);
1516: _zzip_export
1517: ZZIP_FILE*     zzip_fopen(zzip_char_t* name, zzip_char_t* mode);
1518: _zzip_export
1519: zzip_size_t    zzip_fread(void *ptr, zzip_size_t size, zzip_size_t nmemb,
1520:                ZZIP_FILE * file);
1521: _zzip_export
1522: int        zzip_fclose(ZZIP_FILE * fp);
1523: */
1524:
1525:
1526: /*
1527:  * reading info of a single file
1528:  * zzip/stat.c
1529:  *
1530: _zzip_export
1531: int       zzip_dir_stat(ZZIP_DIR * dir, zzip_char_t* name,
1532:                 ZZIP_ST?T * zs, int flags);
1533: _zzip_export
1534: int       zzip_file_stat(ZZIP_FILE * fp, ZZIP_ST?T * zs);
1535: _zzip_export
1536: int       zzip_fstat(ZZIP_FILE * fp, ZZIP_ST?T * zs);
1537: */
```

```
1538: }
1539:
```