pg_stat_statements插件分析

函数`_PG_init`-负责插件的初始化

函数原型

- 源文件

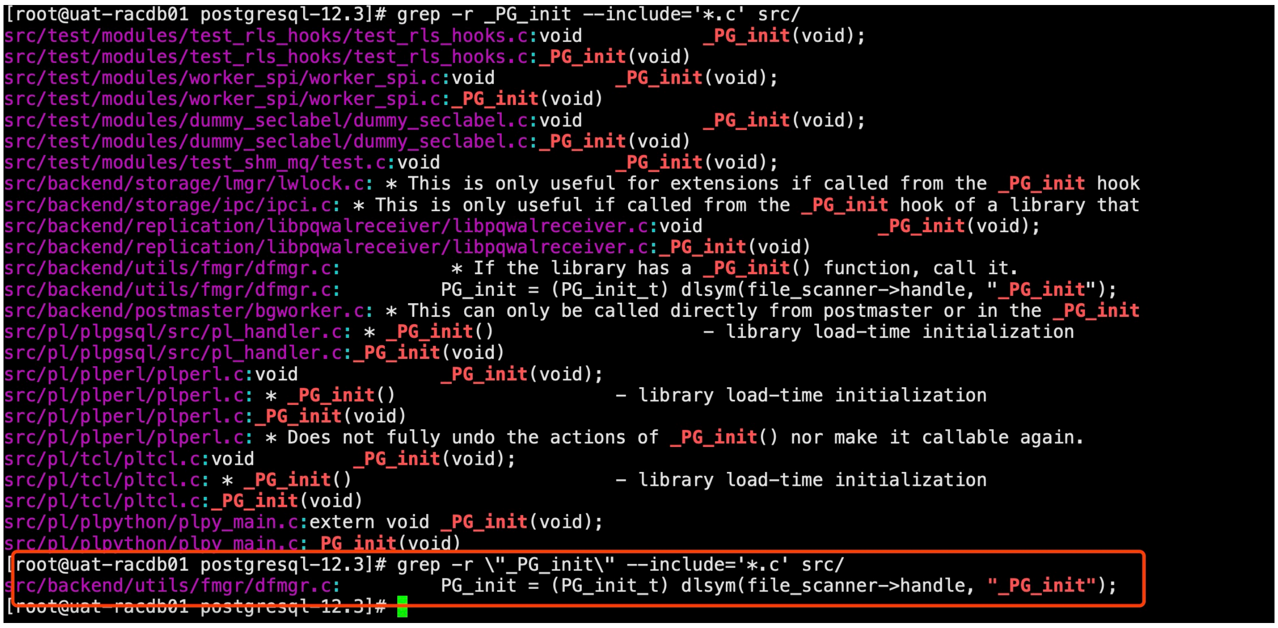
  ```
  src/backend/utils/fmgr/dfmgr.c
  ```

- 原型定义

```
/* signatures for PostgreSQL-specific library init/fini functions */
typedef void (*PG_init_t) (void);
typedef void (*PG_fini_t) (void);
```

函数功能

- 插件都必须定义函数`_PG_init`，PostgreSQL在加载插件对应的动态库文件时，会查找初始化函数`_PG_init`，该函数负责创建的初始化。以下是PostgreSQL源码中与函数`_PG_init`相关的文件。



```
[root@uat-racdb01 postgresql-12.3]# grep -r _PG_init --include='*.c' src/
src/test/modules/test_rls_hooks/test_rls_hooks.c:void           _PG_init(void);
src/test/modules/test_rls_hooks/test_rls_hooks.c:_PG_init(void)
src/test/modules/worker_spi/worker_spi.c:void           _PG_init(void);
src/test/modules/worker_spi/worker_spi.c:_PG_init(void)
src/test/modules/dummy_seclabel/dummy_seclabel.c:void           _PG_init(void);
src/test/modules/dummy_seclabel/dummy_seclabel.c:_PG_init(void)
src/test/modules/test_shm_mq/test.c:void                _PG_init(void);
src/backend/storage/lmgr/lwlock.c: * This is only useful for extensions if called from the _PG_init hook
src/backend/storage/ipc/ipci.c: * This is only useful if called from the _PG_init hook of a library that
src/backend/replication/libpqwalreceiver/libpqwalreceiver.c:void            _PG_init(void);
src/backend/replication/libpqwalreceiver/libpqwalreceiver.c:_PG_init(void)
src/backend/utils/fmgr/dfmgr.c:          * If the library has a _PG_init() function, call it.
src/backend/utils/fmgr/dfmgr.c:         PG_init = (PG_init_t) dlsym(file_scanner->handle, "_PG_init");
src/backend/postmaster/bgworker.c: * This can only be called directly from postmaster or in the _PG_init
src/pl/plpgsql/src/pl_handler.c: * _PG_init()                    - library load-time initialization
src/pl/plpgsql/src/pl_handler.c:_PG_init(void)
src/pl/plperl/plperl.c:void             _PG_init(void);
src/pl/plperl/plperl.c: * _PG_init()                    - library load-time initialization
src/pl/plperl/plperl.c:_PG_init(void)
src/pl/plperl/plperl.c: * Does not fully undo the actions of _PG_init() nor make it callable again.
src/pl/tcl/pltcl.c:void         _PG_init(void);
src/pl/tcl/pltcl.c: * _PG_init()                        - library load-time initialization
src/pl/tcl/pltcl.c:_PG_init(void)
src/pl/plpython/plpy_main.c:extern void _PG_init(void);
src/pl/plpython/plpy_main.c:_PG_init(void)
[root@uat-racdb01 postgresql-12.3]# grep -r \"_PG_init\" --include='*.c' src/
src/backend/utils/fmgr/dfmgr.c:         PG_init = (PG_init_t) dlsym(file_scanner->handle, "_PG_init");
[root@uat-racdb01 postgresql-12.3]# 
```

- PostgreSQL在加载插件对应的库文件时，会调用函数`_PG_init`完成创建的初始化。
    - 函数`internal_load_library`
    - 源文件`src/backend/utils/fmgr/dfmgr.c`
    - 初始化代码

    ```
    /*
     * If the library has a _PG_init() function, call it.
     */
    PG_init = (PG_init_t) dlsym(file_scanner->handle, "_PG_init");
    if (PG_init)
      (*PG_init) ();
    ```

- 插件`pg_stat_statements`在该函数中通过调用以下几个函数定义一些GUC参数(整型、枚举型和布尔型)，并初始化以下几个pg钩子。
    - 创建GUC变量（postgresql参数）

```
DefineCustomIntVariable
DefineCustomEnumVariable
DefineCustomBoolVariable
```

-
    - 安装钩子`hook`

    ```
      /*
       * Install hooks.
       */
      prev_shmem_startup_hook = shmem_startup_hook;  /* 保存原来创建共享内存的hook，比如其他创建创建的. */
      shmem_startup_hook = pgss_shmem_startup;    /* 创建创建的共享内存 */
      prev_post_parse_analyze_hook = post_parse_analyze_hook;
      post_parse_analyze_hook = pgss_post_parse_analyze;
      prev_ExecutorStart = ExecutorStart_hook;    /* 生成执行计划的hook */
      ExecutorStart_hook = pgss_ExecutorStart;
      prev_ExecutorRun = ExecutorRun_hook;    /* 执行sql语句. */
      ExecutorRun_hook = pgss_ExecutorRun;
      prev_ExecutorFinish = ExecutorFinish_hook;
      ExecutorFinish_hook = pgss_ExecutorFinish;
      prev_ExecutorEnd = ExecutorEnd_hook;
      ExecutorEnd_hook = pgss_ExecutorEnd;
      prev_ProcessUtility = ProcessUtility_hook;   /* 非DML语句的执行钩子，比如create table、alter table语句. */
      ProcessUtility_hook = pgss_ProcessUtility;
    ```

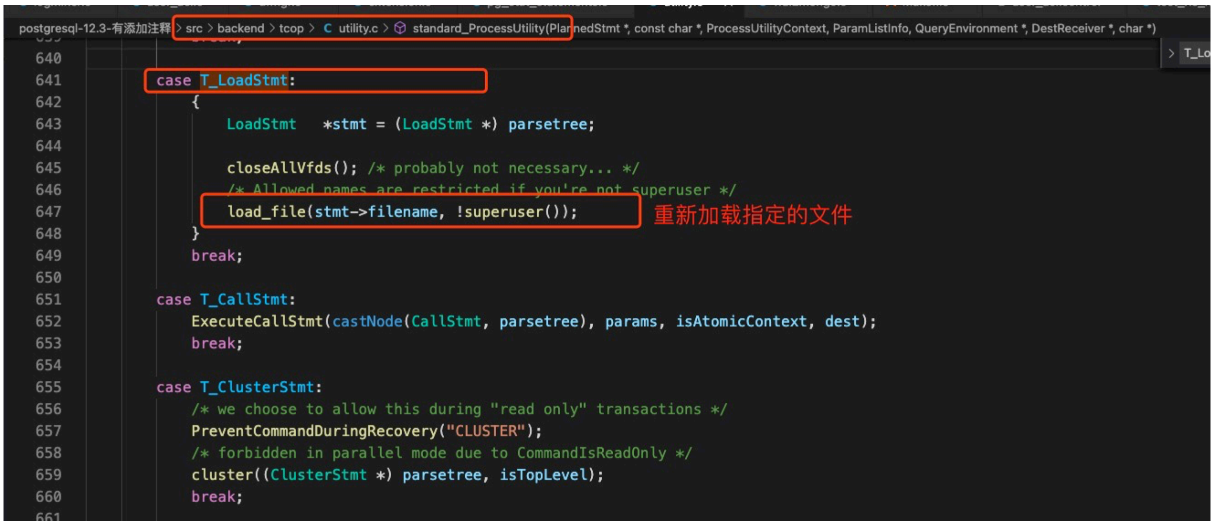
函数代码

```
/*
 * Module load callback
 */
void
_PG_init(void)
{
  /*
   * In order to create our shared memory area, we have to be loaded via
   * shared_preload_libraries.  If not, fall out without hooking into any of
   * the main system.  (We don't throw error here because it seems useful to
   * allow the pg_stat_statements functions to be created even when the
   * module isn't active.  The functions must protect themselves against
   * being called then, however.)
   */
  if (!process_shared_preload_libraries_in_progress)
    return;

  /*
   * Define (or redefine) custom GUC variables.
   */
  DefineCustomIntVariable("pg_stat_statements.max",
              "Sets the maximum number of statements tracked by
pg_stat_statements.",
              NULL,
              &pgss_max,
              5000,
              100,
              INT_MAX,
              PGC_POSTMASTER,
              0,
              NULL,
              NULL,
              NULL);

  DefineCustomEnumVariable("pg_stat_statements.track",
               "Selects which statements are tracked by pg_stat_statements.",
               NULL,
               &pgss_track,
               PGSS_TRACK_TOP,
               track_options,
               PGC_SUSET,
               0,
               NULL,
               NULL,
               NULL);

  DefineCustomBoolVariable("pg_stat_statements.track_utility",
               "Selects whether utility commands are tracked by
pg_stat_statements.",
               NULL,
               &pgss_track_utility,
               true,
               PGC_SUSET,
               0,
               NULL,
               NULL,
               NULL);

  DefineCustomBoolVariable("pg_stat_statements.save",
               "Save pg_stat_statements statistics across server shutdowns.",
               NULL,
               &pgss_save,
               true,
               PGC_SIGHUP,
               0,
               NULL,
               NULL,
               NULL);

  EmitWarningsOnPlaceholders("pg_stat_statements");

  /*
   * Request additional shared resources.  (These are no-ops if we're not in
   * the postmaster process.)  We'll allocate or attach to the shared
   * resources in pgss_shmem_startup().
   */
  RequestAddinShmemSpace(pgss_memsize());
  RequestNamedLWLockTranche("pg_stat_statements", 1);

  /*
   * Install hooks.
   */
  prev_shmem_startup_hook = shmem_startup_hook;
  shmem_startup_hook = pgss_shmem_startup;
  prev_post_parse_analyze_hook = post_parse_analyze_hook;
  post_parse_analyze_hook = pgss_post_parse_analyze;
  prev_ExecutorStart = ExecutorStart_hook;
  ExecutorStart_hook = pgss_ExecutorStart;
  prev_ExecutorRun = ExecutorRun_hook;
  ExecutorRun_hook = pgss_ExecutorRun;
  prev_ExecutorFinish = ExecutorFinish_hook;
  ExecutorFinish_hook = pgss_ExecutorFinish;
  prev_ExecutorEnd = ExecutorEnd_hook;
  ExecutorEnd_hook = pgss_ExecutorEnd;
  prev_ProcessUtility = ProcessUtility_hook;
  ProcessUtility_hook = pgss_ProcessUtility;
}
```

函数`_PG_fini`-负责插件的清理

注意事项

- 虽然函数`_PG_fini`负责插件的清理工作，但是在使用sql命令`drop extension`删除创建时，PG并不会调用该函数。
- 程序执行到以下代码才会调用该函数。但是调试`pg_ctl reload`命令，发现并不会执行以下代码。只有sql命令`load 'user_acl'`加载库文件时才会触发。



```
postgresql-12.3-有添加注释 > src > backend > tcop > C utility.c > standard_ProcessUtility(PlannedStmt *, const char *, ProcessUtilityContext, ParamListInfo, QueryEnvironment *, DestReceiver *, char *)

        case T_LoadStmt:
            {
                LoadStmt   *stmt = (LoadStmt *) parsetree;

                closeAllVfds(); /* probably not necessary... */
                /* Allowed names are restricted if you're not superuser */
                load_file(stmt->filename, !superuser());
            }
            break;

        case T_CallStmt:
            ExecuteCallStmt(castNode(CallStmt, parsetree), params, isAtomicContext, dest);
            break;

        case T_ClusterStmt:
            /* we choose to allow this during "read only" transactions */
            PreventCommandDuringRecovery("CLUSTER");
            /* forbidden in parallel mode due to CommandIsReadOnly */
            cluster((ClusterStmt *) parsetree, isTopLevel);
            break;
```

- load命令加载库文件
    - 会话1.

    ```
    postgres=# select pg_backend_pid();
     pg_backend_pid
    ----------------
              44478
    (1 row)

    postgres=# load 'user_acl';
    ```

    - gdb调试

```
(gdb) bt
#0  standard_ProcessUtility (pstmt=0x2ec0dd8, queryString=0x2ec0068
"load 'user_acl';", context=PROCESS_UTILITY_TOPLEVEL, params=0x0,
queryEnv=0x0, dest=0x2ec0ed0,
    completionTag=0x7ffde1121260 "") at utility.c:643
#1  0x00007fe1c3ac8bd0 in uacl_ProcessUtility (pstmt=0x2ec0dd8,
queryString=0x2ec0068 "load 'user_acl';",
context=PROCESS_UTILITY_TOPLEVEL, params=0x0, queryEnv=0x0,
    dest=0x2ec0ed0, completionTag=0x7ffde1121260 "") at user_acl.c:110
#2  0x00007fe1c38c0c64 in pgss_ProcessUtility (pstmt=0x2ec0dd8,
queryString=0x2ec0068 "load 'user_acl';",
context=PROCESS_UTILITY_TOPLEVEL, params=0x0, queryEnv=0x0,
    dest=0x2ec0ed0, completionTag=0x7ffde1121260 "") at
pg_stat_statements.c:1002
#3  0x00000000008e5866 in ProcessUtility (pstmt=0x2ec0dd8,
queryString=0x2ec0068 "load 'user_acl';",
context=PROCESS_UTILITY_TOPLEVEL, params=0x0, queryEnv=0x0,
    dest=0x2ec0ed0, completionTag=0x7ffde1121260 "") at utility.c:356
#4  0x00000000008e48a2 in PortalRunUtility (portal=0x2f5a238,
pstmt=0x2ec0dd8, isTopLevel=true, setHoldSnapshot=false,
dest=0x2ec0ed0, completionTag=0x7ffde1121260 "")
    at pquery.c:1175
#5  0x00000000008e4aba in PortalRunMulti (portal=0x2f5a238,
isTopLevel=true, setHoldSnapshot=false, dest=0x2ec0ed0,
altdest=0x2ec0ed0, completionTag=0x7ffde1121260 "")
    at pquery.c:1321
#6  0x00000000008e3fff in PortalRun (portal=0x2f5a238,
count=9223372036854775807, isTopLevel=true, run_once=true,
dest=0x2ec0ed0, altdest=0x2ec0ed0,
    completionTag=0x7ffde1121260 "") at pquery.c:796
#7  0x00000000008de0d5 in exec_simple_query (query_string=0x2ec0068
"load 'user_acl';") at postgres.c:1215
#8  0x00000000008e2256 in PostgresMain (argc=1, argv=0x2ef4710,
dbname=0x2ebcd08 "postgres", username=0x2ef4558 "admin") at
postgres.c:4247
#9  0x0000000000839272 in BackendRun (port=0x2ee7430) at
postmaster.c:4448
#10 0x0000000000838a38 in BackendStartup (port=0x2ee7430) at
postmaster.c:4139
#11 0x0000000000834df1 in ServerLoop () at postmaster.c:1704
#12 0x00000000008346c0 in PostmasterMain (argc=3, argv=0x2ebac80) at
postmaster.c:1377
#13 0x0000000000755bd8 in main (argc=3, argv=0x2ebac80) at main.c:228
(gdb)
```

功能描述

- 卸载插件时，负责插件的清理工作。

- 源文件`src/backend/utils/fmgr/dfmgr.c`中的函数`internal_unload_library`通过以下代码调用插件的清理函数`_PG_fini`。

```
/*
 * If the library has a _PG_fini() function, call it.
 */
PG_fini = (PG_fini_t) dlsym(file_scanner->handle, "_PG_fini");
if (PG_fini)
```

- 插件`pg_stat_statements`公共该函数恢复在初始化函数中修改过的钩子。

函数代码

```
/*
 * Module unload callback
 */
void
_PG_fini(void)
{
  /* Uninstall hooks. */
  shmem_startup_hook = prev_shmem_startup_hook;
  post_parse_analyze_hook = prev_post_parse_analyze_hook;
  ExecutorStart_hook = prev_ExecutorStart;
  ExecutorRun_hook = prev_ExecutorRun;
  ExecutorFinish_hook = prev_ExecutorFinish;
  ExecutorEnd_hook = prev_ExecutorEnd;
  ProcessUtility_hook = prev_ProcessUtility;
}
```

函数`pgss_shmem_startup`-创建共享内存

函数代码

```
/*
 * shmem_startup hook: allocate or attach to shared memory,
 * then load any pre-existing statistics from file.
 * Also create and load the query-texts file, which is expected to exist
 * (even if empty) while the module is enabled.
 */
static void
pgss_shmem_startup(void)
{
  bool    found;
  HASHCTL   info;
  FILE     *file = NULL;
  FILE     *qfile = NULL;
  uint32    header;
  int32   num;
  int32   pgver;
  int32   i;
  int     buffer_size;
  char     *buffer = NULL;

  if (prev_shmem_startup_hook)
    prev_shmem_startup_hook();

  /* reset in case this is a restart within the postmaster */
  pgss = NULL;
  pgss_hash = NULL;

  /*
   * Create or attach to the shared memory state, including hash table
   */
  LWLockAcquire(AddinShmemInitLock, LW_EXCLUSIVE);

  pgss = ShmemInitStruct("pg_stat_statements",
             sizeof(pgssSharedState),
             &found);

  if (!found)
  {
    /* First time through ... */
    pgss->lock = &(GetNamedLWLockTranche("pg_stat_statements"))->lock;
    pgss->cur_median_usage = ASSUMED_MEDIAN_INIT;
    pgss->mean_query_len = ASSUMED_LENGTH_INIT;
    SpinLockInit(&pgss->mutex);
    pgss->extent = 0;
    pgss->n_writers = 0;
    pgss->gc_count = 0;
  }
```

```
memset(&info, 0, sizeof(info));
info.keysize = sizeof(pgssHashKey);
info.entrysize = sizeof(pgssEntry);
pgss_hash = ShmemInitHash("pg_stat_statements hash",
              pgss_max, pgss_max,
              &info,
              HASH_ELEM | HASH_BLOBS);

LWLockRelease(AddinShmemInitLock);

/*
 * If we're in the postmaster (or a standalone backend...), set up a shmem
 * exit hook to dump the statistics to disk.
 */
if (!IsUnderPostmaster)
  on_shmem_exit(pgss_shmem_shutdown, (Datum) 0);

/*
 * Done if some other process already completed our initialization.
 */
if (found)
  return;

/*
 * Note: we don't bother with locks here, because there should be no other
 * processes running when this code is reached.
 */

/* Unlink query text file possibly left over from crash */
unlink(PGSS_TEXT_FILE);

/* Allocate new query text temp file */
qfile = AllocateFile(PGSS_TEXT_FILE, PG_BINARY_W);
if (qfile == NULL)
  goto write_error;

/*
 * If we were told not to load old statistics, we're done.  (Note we do
 * not try to unlink any old dump file in this case.  This seems a bit
 * questionable but it's the historical behavior.)
 */
if (!pgss_save)
{
  FreeFile(qfile);
  return;
}

/*
 * Attempt to load old statistics from the dump file.
```

```
   */
  file = AllocateFile(PGSS_DUMP_FILE, PG_BINARY_R);
  if (file == NULL)
  {
    if (errno != ENOENT)
      goto read_error;
    /* No existing persisted stats file, so we're done */
    FreeFile(qfile);
    return;
  }

  buffer_size = 2048;
  buffer = (char *) palloc(buffer_size);

  if (fread(&header, sizeof(uint32), 1, file) != 1 ||
    fread(&pgver, sizeof(uint32), 1, file) != 1 ||
    fread(&num, sizeof(int32), 1, file) != 1)
    goto read_error;

  if (header != PGSS_FILE_HEADER ||
    pgver != PGSS_PG_MAJOR_VERSION)
    goto data_error;

  for (i = 0; i < num; i++)
  {
    pgssEntry temp;
    pgssEntry  *entry;
    Size    query_offset;

    if (fread(&temp, sizeof(pgssEntry), 1, file) != 1)
      goto read_error;

    /* Encoding is the only field we can easily sanity-check */
    if (!PG_VALID_BE_ENCODING(temp.encoding))
      goto data_error;

    /* Resize buffer as needed */
    if (temp.query_len >= buffer_size)
    {
      buffer_size = Max(buffer_size * 2, temp.query_len + 1);
      buffer = repalloc(buffer, buffer_size);
    }

    if (fread(buffer, 1, temp.query_len + 1, file) != temp.query_len + 1)
      goto read_error;

    /* Should have a trailing null, but let's make sure */
    buffer[temp.query_len] = '\0';
    /* Skip loading "sticky" entries */
    if (temp.counters.calls == 0)
      continue;

    /* Store the query text */
    query_offset = pgss->extent;
    if (fwrite(buffer, 1, temp.query_len + 1, qfile) != temp.query_len + 1)
      goto write_error;
    pgss->extent += temp.query_len + 1;

    /* make the hashtable entry (discards old entries if too many) */
    entry = entry_alloc(&temp.key, query_offset, temp.query_len,
              temp.encoding,
              false);

    /* copy in the actual stats */
    entry->counters = temp.counters;
  }

  pfree(buffer);
  FreeFile(file);
  FreeFile(qfile);

  /*
   * Remove the persisted stats file so it's not included in
   * backups/replication standbys, etc.  A new file will be written on next
   * shutdown.
   *
   * Note: it's okay if the PGSS_TEXT_FILE is included in a basebackup,
   * because we remove that file on startup; it acts inversely to
   * PGSS_DUMP_FILE, in that it is only supposed to be around when the
   * server is running, whereas PGSS_DUMP_FILE is only supposed to be around
   * when the server is not running.  Leaving the file creates no danger of
   * a newly restored database having a spurious record of execution costs,
   * which is what we're really concerned about here.
   */
  unlink(PGSS_DUMP_FILE);

  return;

read_error:
  ereport(LOG,
      (errcode_for_file_access(),
       errmsg("could not read file \"%s\": %m",
          PGSS_DUMP_FILE)));
  goto fail;
data_error:
  ereport(LOG,
      (errcode(ERRCODE_INVALID_PARAMETER_VALUE),
        errmsg("ignoring invalid data in file \"%s\"",
            PGSS_DUMP_FILE)));
  goto fail;
write_error:
  ereport(LOG,
      (errcode_for_file_access(),
       errmsg("could not write file \"%s\": %m",
            PGSS_TEXT_FILE)));
fail:
  if (buffer)
    pfree(buffer);
  if (file)
    FreeFile(file);
  if (qfile)
    FreeFile(qfile);
  /* If possible, throw away the bogus file; ignore any error */
  unlink(PGSS_DUMP_FILE);

  /*
   * Don't unlink PGSS_TEXT_FILE here; it should always be around while the
   * server is running with pg_stat_statements enabled
   */
}
```

函数`pgss_shmem_shutdown`-将统计信息dump到文件

函数代码

```
/*
 * shmem_shutdown hook: Dump statistics into file.
 *
 * Note: we don't bother with acquiring lock, because there should be no
 * other processes running when this is called.
 */
static void
pgss_shmem_shutdown(int code, Datum arg)
{
  FILE     *file;
  char     *qbuffer = NULL;
  Size     qbuffer_size = 0;
  HASH_SEQ_STATUS hash_seq;
	int32		num_entries;
	pgssEntry  *entry;

	/* Don't try to dump during a crash. */
	if (code)
		return;

	/* Safety check ... shouldn't get here unless shmem is set up. */
	if (!pgss || !pgss_hash)
		return;

	/* Don't dump if told not to. */
	if (!pgss_save)
		return;

	file = AllocateFile(PGSS_DUMP_FILE ".tmp", PG_BINARY_W);
	if (file == NULL)
		goto error;

	if (fwrite(&PGSS_FILE_HEADER, sizeof(uint32), 1, file) != 1)
		goto error;
	if (fwrite(&PGSS_PG_MAJOR_VERSION, sizeof(uint32), 1, file) != 1)
		goto error;
	num_entries = hash_get_num_entries(pgss_hash);
	if (fwrite(&num_entries, sizeof(int32), 1, file) != 1)
		goto error;

	qbuffer = qtext_load_file(&qbuffer_size);
	if (qbuffer == NULL)
		goto error;

	/*
	 * When serializing to disk, we store query texts immediately after their
	 * entry data.  Any orphaned query texts are thereby excluded.
	 */
	hash_seq_init(&hash_seq, pgss_hash);
	while ((entry = hash_seq_search(&hash_seq)) != NULL)
	{
		int			len = entry->query_len;
		char	   *qstr = qtext_fetch(entry->query_offset, len,
									   qbuffer, qbuffer_size);

		if (qstr == NULL)
			continue;		/* Ignore any entries with bogus texts */

		if (fwrite(entry, sizeof(pgssEntry), 1, file) != 1 ||
			fwrite(qstr, 1, len + 1, file) != len + 1)
		{
			/* note: we assume hash_seq_term won't change errno */
```

```
        hash_seq_term(&hash_seq);
        goto error;
      }
    }

    free(qbuffer);
    qbuffer = NULL;

    if (FreeFile(file))
    {
      file = NULL;
      goto error;
    }

    /*
     * Rename file into place, so we atomically replace any old one.
     */
    (void) durable_rename(PGSS_DUMP_FILE ".tmp", PGSS_DUMP_FILE, LOG);

    /* Unlink query-texts file; it's not needed while shutdown */
    unlink(PGSS_TEXT_FILE);

    return;

  error:
    ereport(LOG,
        (errcode_for_file_access(),
         errmsg("could not write file \"%s\": %m",
            PGSS_DUMP_FILE ".tmp")));
    if (qbuffer)
      free(qbuffer);
    if (file)
      FreeFile(file);
    unlink(PGSS_DUMP_FILE ".tmp");
    unlink(PGSS_TEXT_FILE);
  }
```

函数`pgss_post_parse_analyze`

函数代码

```
/*
 * Post-parse-analysis hook: mark query with a queryId
```

```c
 */
static void
pgss_post_parse_analyze(ParseState *pstate, Query *query)
{
  pgssJumbleState jstate;

  if (prev_post_parse_analyze_hook)
    prev_post_parse_analyze_hook(pstate, query);

  /* Assert we didn't do this already */
  Assert(query->queryId == UINT64CONST(0));

  /* Safety check... */
  if (!pgss || !pgss_hash)
    return;

  /*
   * Utility statements get queryId zero.  We do this even in cases where
   * the statement contains an optimizable statement for which a queryId
   * could be derived (such as EXPLAIN or DECLARE CURSOR).  For such cases,
   * runtime control will first go through ProcessUtility and then the
   * executor, and we don't want the executor hooks to do anything, since we
   * are already measuring the statement's costs at the utility level.
   */
  if (query->utilityStmt)
  {
    query->queryId = UINT64CONST(0);
    return;
  }

  /* Set up workspace for query jumbling */
  jstate.jumble = (unsigned char *) palloc(JUMBLE_SIZE);
  jstate.jumble_len = 0;
  jstate.clocations_buf_size = 32;
  jstate.clocations = (pgssLocationLen *)
    palloc(jstate.clocations_buf_size * sizeof(pgssLocationLen));
  jstate.clocations_count = 0;
  jstate.highest_extern_param_id = 0;

  /* Compute query ID and mark the Query node with it */
  JumbleQuery(&jstate, query);
  query->queryId =
    DatumGetUInt64(hash_any_extended(jstate.jumble, jstate.jumble_len, 0));

  /*
   * If we are unlucky enough to get a hash of zero, use 1 instead, to
   * prevent confusion with the utility-statement case.
   */
  if (query->queryId == UINT64CONST(0))
		query->queryId = UINT64CONST(1);

	/*
	 * If we were able to identify any ignorable constants, we immediately
	 * create a hash table entry for the query, so that we can record the
	 * normalized form of the query string.  If there were no such constants,
	 * the normalized string would be the same as the query text anyway, so
	 * there's no need for an early entry.
	 */
	if (jstate.clocations_count > 0)
		pgss_store(pstate->p_sourcetext,
				   query->queryId,
				   query->stmt_location,
				   query->stmt_len,
				   0,
				   0,
				   NULL,
				   &jstate);
}
```

函数`pgss_ExecutorStart`-开启sql语句统计跟踪

功能描述

- 在执行完函数`standard_ExecutorStart`后，初始化与当前sql语句相关的统计信息，如下：

```
	/*
	 * Set up to track total elapsed time in ExecutorRun.  Make sure the
	 * space is allocated in the per-query context so it will go away at
	 * ExecutorEnd.
	 */
	if (queryDesc->totaltime == NULL)
	{
		MemoryContext oldcxt;

		oldcxt = MemoryContextSwitchTo(queryDesc->estate->es_query_cxt);
		queryDesc->totaltime = InstrAlloc(1, INSTRUMENT_ALL);
		MemoryContextSwitchTo(oldcxt);
	}
```

函数代码

```c
/*
 * ExecutorStart hook: start up tracking if needed
 */
static void
pgss_ExecutorStart(QueryDesc *queryDesc, int eflags)
{
  if (prev_ExecutorStart)
    prev_ExecutorStart(queryDesc, eflags);
  else
    standard_ExecutorStart(queryDesc, eflags);

  /*
   * If query has queryId zero, don't track it.  This prevents double
   * counting of optimizable statements that are directly contained in
   * utility statements.
   */
  if (pgss_enabled() && queryDesc->plannedstmt->queryId != UINT64CONST(0))
  {
    /*
     * Set up to track total elapsed time in ExecutorRun.  Make sure the
     * space is allocated in the per-query context so it will go away at
     * ExecutorEnd.
     */
    if (queryDesc->totaltime == NULL)
    {
      MemoryContext oldcxt;

      oldcxt = MemoryContextSwitchTo(queryDesc->estate->es_query_cxt);
      queryDesc->totaltime = InstrAlloc(1, INSTRUMENT_ALL);
      MemoryContextSwitchTo(oldcxt);
    }
  }
}
```

函数`pgss_ProcessUtility`

功能描述

- 工具命令是除了`SELECT`、`INSERT`、 `UPDATE`和`DELETE`之外所有的其他命令。
- 该函数负责处理工具命令的统计信息。

钩子的定义

- 源文件

```
src/backend/tcop/utility.c
```

- 钩子的定义

```
/* Hook for plugins to get control in ProcessUtility() */
ProcessUtility_hook_type ProcessUtility_hook = NULL;
```

- 钩子的函数原型

```
src/include/tcop/utility.h
```

```
/* Hook for plugins to get control in ProcessUtility() */
typedef void (*ProcessUtility_hook_type) (PlannedStmt *pstmt,
                                          const char *queryString, ProcessUtilityContext
context,
                                          ParamListInfo params,
                                          QueryEnvironment *queryEnv,
                                          DestReceiver *dest, char *completionTag);
```

函数代码

```
/*
 * ProcessUtility hook
 */
static void
pgss_ProcessUtility(PlannedStmt *pstmt, const char *queryString,
                    ProcessUtilityContext context,
                    ParamListInfo params, QueryEnvironment *queryEnv,
                    DestReceiver *dest, char *completionTag)
{
    Node       *parsetree = pstmt->utilityStmt;

    /*
     * If it's an EXECUTE statement, we don't track it and don't increment the
     * nesting level.  This allows the cycles to be charged to the underlying
     * PREPARE instead (by the Executor hooks), which is much more useful.
     *
     * We also don't track execution of PREPARE.  If we did, we would get one
     * hash table entry for the PREPARE (with hash calculated from the query
     * string), and then a different one with the same query string (but hash
     * calculated from the query tree) would be used to accumulate costs of
     * ensuing EXECUTEs.  This would be confusing, and inconsistent with other
     * cases where planning time is not included at all.
     *
     * Likewise, we don't track execution of DEALLOCATE.
     */
    if (pgss_track_utility && pgss_enabled() &&
      !IsA(parsetree, ExecuteStmt) &&
      !IsA(parsetree, PrepareStmt) &&
      !IsA(parsetree, DeallocateStmt))
    {
      instr_time  start;
      instr_time  duration;
      uint64    rows;
      BufferUsage bufusage_start,
                    bufusage;

      bufusage_start = pgBufferUsage;
      INSTR_TIME_SET_CURRENT(start);

      nested_level++;
      PG_TRY();
      {
        if (prev_ProcessUtility)
          prev_ProcessUtility(pstmt, queryString,
                    context, params, queryEnv,
                    dest, completionTag);
        else
          standard_ProcessUtility(pstmt, queryString,
                      context, params, queryEnv,
                      dest, completionTag);
        nested_level--;
      }
      PG_CATCH();
      {
        nested_level--;
        PG_RE_THROW();
      }
      PG_END_TRY();

      INSTR_TIME_SET_CURRENT(duration);
      INSTR_TIME_SUBTRACT(duration, start);

      /* parse command tag to retrieve the number of affected rows. */
      if (completionTag &&
        strncmp(completionTag, "COPY ", 5) == 0)
        rows = pg_strtouint64(completionTag + 5, NULL, 10);
      else
```

```
      rows = 0;

    /* calc differences of buffer counters. */
    bufusage.shared_blks_hit =
      pgBufferUsage.shared_blks_hit - bufusage_start.shared_blks_hit;
    bufusage.shared_blks_read =
      pgBufferUsage.shared_blks_read - bufusage_start.shared_blks_read;
    bufusage.shared_blks_dirtied =
      pgBufferUsage.shared_blks_dirtied - bufusage_start.shared_blks_dirtied;
    bufusage.shared_blks_written =
      pgBufferUsage.shared_blks_written - bufusage_start.shared_blks_written;
    bufusage.local_blks_hit =
      pgBufferUsage.local_blks_hit - bufusage_start.local_blks_hit;
    bufusage.local_blks_read =
      pgBufferUsage.local_blks_read - bufusage_start.local_blks_read;
    bufusage.local_blks_dirtied =
      pgBufferUsage.local_blks_dirtied - bufusage_start.local_blks_dirtied;
    bufusage.local_blks_written =
      pgBufferUsage.local_blks_written - bufusage_start.local_blks_written;
    bufusage.temp_blks_read =
      pgBufferUsage.temp_blks_read - bufusage_start.temp_blks_read;
    bufusage.temp_blks_written =
      pgBufferUsage.temp_blks_written - bufusage_start.temp_blks_written;
    bufusage.blk_read_time = pgBufferUsage.blk_read_time;
    INSTR_TIME_SUBTRACT(bufusage.blk_read_time, bufusage_start.blk_read_time);
    bufusage.blk_write_time = pgBufferUsage.blk_write_time;
    INSTR_TIME_SUBTRACT(bufusage.blk_write_time,
bufusage_start.blk_write_time);

    pgss_store(queryString,
           0,     /* signal that it's a utility stmt */
           pstmt->stmt_location,
           pstmt->stmt_len,
           INSTR_TIME_GET_MILLISEC(duration),
           rows,
           &bufusage,
           NULL);
  }
  else
  {
    if (prev_ProcessUtility)
      prev_ProcessUtility(pstmt, queryString,
                context, params, queryEnv,
                dest, completionTag);
    else
      standard_ProcessUtility(pstmt, queryString,
                  context, params, queryEnv,
                  dest, completionTag);
  }
```

```
}
```

与工具命令相关的其他函数

函数`ProcessUtility`-工具命令处理函数

- 源文件

  ```
  src/backend/tcop/utility.c
  ```

- 函数工具
  - 工具命令的总入口，会调用函数`standard_ProcessUtility`执行具体的操作
  - 调用钩子函数`ProcessUtility_hook`

- 函数代码

```
/*
 * ProcessUtility
 *    general utility function invoker
 *
 *  pstmt: PlannedStmt wrapper for the utility statement
 *  queryString: original source text of command
 *  context: identifies source of statement (toplevel client command,
 *    non-toplevel client command, subcommand of a larger utility command)
 *  params: parameters to use during execution
 *  queryEnv: environment for parse through execution (e.g., ephemeral named
 *    tables like trigger transition tables).  May be NULL.
 *  dest: where to send results
 *  completionTag: points to a buffer of size COMPLETION_TAG_BUFSIZE
 *    in which to store a command completion status string.
 *
 * Caller MUST supply a queryString; it is not allowed (anymore) to pass NULL.
 * If you really don't have source text, you can pass a constant string,
 * perhaps "(query not available)".
 *
 * completionTag is only set nonempty if we want to return a nondefault status.
 *
 * completionTag may be NULL if caller doesn't want a status string.
 *
 * Note for users of ProcessUtility_hook: the same queryString may be passed
 * to multiple invocations of ProcessUtility when processing a query string
 * containing multiple semicolon-separated statements.  One should use
 * pstmt->stmt_location and pstmt->stmt_len to identify the substring
 * containing the current statement.  Keep in mind also that some utility
```

```
 * statements (e.g., CREATE SCHEMA) will recurse to ProcessUtility to process
 * sub-statements, often passing down the same queryString, stmt_location,
 * and stmt_len that were given for the whole statement.
 */
void
ProcessUtility(PlannedStmt *pstmt,
         const char *queryString,
         ProcessUtilityContext context,
         ParamListInfo params,
         QueryEnvironment *queryEnv,
         DestReceiver *dest,
         char *completionTag)
{
  Assert(IsA(pstmt, PlannedStmt));
  Assert(pstmt->commandType == CMD_UTILITY);
  Assert(queryString != NULL);  /* required as of 8.4 */

  /*
   * We provide a function hook variable that lets loadable plugins get
   * control when ProcessUtility is called.  Such a plugin would normally
   * call standard_ProcessUtility().
   */
  if (ProcessUtility_hook)
    (*ProcessUtility_hook) (pstmt, queryString,
                context, params, queryEnv,
                dest, completionTag);
  else
    standard_ProcessUtility(pstmt, queryString,
                context, params, queryEnv,
                dest, completionTag);
}
```

函数`standard_ProcessUtility`-工具命令处理函数

源文件

```
src/backend/tcop/utility.c
```

函数代码

```c
/*
 * standard_ProcessUtility itself deals only with utility commands for
 * which we do not provide event trigger support.  Commands that do have
 * such support are passed down to ProcessUtilitySlow, which contains the
 * necessary infrastructure for such triggers.
 *
 * This division is not just for performance: it's critical that the
 * event trigger code not be invoked when doing START TRANSACTION for
 * example, because we might need to refresh the event trigger cache,
 * which requires being in a valid transaction.
 */
void
standard_ProcessUtility(PlannedStmt *pstmt,
            const char *queryString,
            ProcessUtilityContext context,
            ParamListInfo params,
            QueryEnvironment *queryEnv,
            DestReceiver *dest,
            char *completionTag)
{
  Node     *parsetree = pstmt->utilityStmt;
  bool    isTopLevel = (context == PROCESS_UTILITY_TOPLEVEL);
  bool    isAtomicContext = (!(context == PROCESS_UTILITY_TOPLEVEL || context
== PROCESS_UTILITY_QUERY_NONATOMIC) || IsTransactionBlock());
  ParseState *pstate;

  /* This can recurse, so check for excessive recursion */
  check_stack_depth();

  check_xact_readonly(parsetree);

  if (completionTag)
    completionTag[0] = '\0';

  pstate = make_parsestate(NULL);
  pstate->p_sourcetext = queryString;

  switch (nodeTag(parsetree))
  {
      /*
       * ******************** transactions ********************
       */
    case T_TransactionStmt:
      {
        TransactionStmt *stmt = (TransactionStmt *) parsetree;

        switch (stmt->kind)
        {
```

```
  /*
   * START TRANSACTION, as defined by SQL99: Identical
   * to BEGIN.  Same code for both.
   */
case TRANS_STMT_BEGIN:
case TRANS_STMT_START:
  {
    ListCell   *lc;

    BeginTransactionBlock();
    foreach(lc, stmt->options)
    {
      DefElem    *item = (DefElem *) lfirst(lc);

      if (strcmp(item->defname, "transaction_isolation") == 0)
        SetPGVariable("transaction_isolation",
                list_make1(item->arg),
                true);
      else if (strcmp(item->defname, "transaction_read_only") == 0)
        SetPGVariable("transaction_read_only",
                list_make1(item->arg),
                true);
      else if (strcmp(item->defname, "transaction_deferrable") == 0)
        SetPGVariable("transaction_deferrable",
                list_make1(item->arg),
                true);
    }
  }
  break;

case TRANS_STMT_COMMIT:
  if (!EndTransactionBlock(stmt->chain))
  {
    /* report unsuccessful commit in completionTag */
    if (completionTag)
      strcpy(completionTag, "ROLLBACK");
  }
  break;

case TRANS_STMT_PREPARE:
  PreventCommandDuringRecovery("PREPARE TRANSACTION");
  if (!PrepareTransactionBlock(stmt->gid))
  {
    /* report unsuccessful commit in completionTag */
    if (completionTag)
      strcpy(completionTag, "ROLLBACK");
  }
  break;
```

```
                case TRANS_STMT_COMMIT_PREPARED:
                  PreventInTransactionBlock(isTopLevel, "COMMIT PREPARED");
                  PreventCommandDuringRecovery("COMMIT PREPARED");
                  FinishPreparedTransaction(stmt->gid, true);
                  break;

                case TRANS_STMT_ROLLBACK_PREPARED:
                  PreventInTransactionBlock(isTopLevel, "ROLLBACK PREPARED");
                  PreventCommandDuringRecovery("ROLLBACK PREPARED");
                  FinishPreparedTransaction(stmt->gid, false);
                  break;

                case TRANS_STMT_ROLLBACK:
                  UserAbortTransactionBlock(stmt->chain);
                  break;

                case TRANS_STMT_SAVEPOINT:
                  RequireTransactionBlock(isTopLevel, "SAVEPOINT");
                  DefineSavepoint(stmt->savepoint_name);
                  break;

                case TRANS_STMT_RELEASE:
                  RequireTransactionBlock(isTopLevel, "RELEASE SAVEPOINT");
                  ReleaseSavepoint(stmt->savepoint_name);
                  break;

                case TRANS_STMT_ROLLBACK_TO:
                  RequireTransactionBlock(isTopLevel, "ROLLBACK TO SAVEPOINT");
                  RollbackToSavepoint(stmt->savepoint_name);

                  /*
                   * CommitTransactionCommand is in charge of
                   * re-defining the savepoint again
                   */
                  break;
              }
          }
          break;

          /*
           * Portal (cursor) manipulation
           */
        case T_DeclareCursorStmt:
          PerformCursorOpen((DeclareCursorStmt *) parsetree, params,
                    queryString, isTopLevel);
          break;

        case T_ClosePortalStmt:
          {
```

```
        ClosePortalStmt *stmt = (ClosePortalStmt *) parsetree;

        CheckRestrictedOperation("CLOSE");
        PerformPortalClose(stmt->portalname);
      }
      break;

    case T_FetchStmt:
      PerformPortalFetch((FetchStmt *) parsetree, dest,
                 completionTag);
      break;

    case T_DoStmt:
      ExecuteDoStmt((DoStmt *) parsetree, isAtomicContext);
      break;

    case T_CreateTableSpaceStmt:
      /* no event triggers for global objects */
      PreventInTransactionBlock(isTopLevel, "CREATE TABLESPACE");
      CreateTableSpace((CreateTableSpaceStmt *) parsetree);
      break;

    case T_DropTableSpaceStmt:
      /* no event triggers for global objects */
      PreventInTransactionBlock(isTopLevel, "DROP TABLESPACE");
      DropTableSpace((DropTableSpaceStmt *) parsetree);
      break;

    case T_AlterTableSpaceOptionsStmt:
      /* no event triggers for global objects */
      AlterTableSpaceOptions((AlterTableSpaceOptionsStmt *) parsetree);
      break;

    case T_TruncateStmt:
      ExecuteTruncate((TruncateStmt *) parsetree);
      break;

    case T_CopyStmt:
      {
        uint64    processed;

        DoCopy(pstate, (CopyStmt *) parsetree,
             pstmt->stmt_location, pstmt->stmt_len,
             &processed);
        if (completionTag)
          snprintf(completionTag, COMPLETION_TAG_BUFSIZE,
               "COPY " UINT64_FORMAT, processed);
      }
      break;
    case T_PrepareStmt:
      CheckRestrictedOperation("PREPARE");
      PrepareQuery((PrepareStmt *) parsetree, queryString,
             pstmt->stmt_location, pstmt->stmt_len);
      break;

    case T_ExecuteStmt:
      ExecuteQuery((ExecuteStmt *) parsetree, NULL,
             queryString, params,
             dest, completionTag);
      break;

    case T_DeallocateStmt:
      CheckRestrictedOperation("DEALLOCATE");
      DeallocateQuery((DeallocateStmt *) parsetree);
      break;

    case T_GrantRoleStmt:
      /* no event triggers for global objects */
      GrantRole((GrantRoleStmt *) parsetree);
      break;

    case T_CreatedbStmt:
      /* no event triggers for global objects */
      PreventInTransactionBlock(isTopLevel, "CREATE DATABASE");
      createdb(pstate, (CreatedbStmt *) parsetree);
      break;

    case T_AlterDatabaseStmt:
      /* no event triggers for global objects */
      AlterDatabase(pstate, (AlterDatabaseStmt *) parsetree, isTopLevel);
      break;

    case T_AlterDatabaseSetStmt:
      /* no event triggers for global objects */
      AlterDatabaseSet((AlterDatabaseSetStmt *) parsetree);
      break;

    case T_DropdbStmt:
      {
        DropdbStmt *stmt = (DropdbStmt *) parsetree;

        /* no event triggers for global objects */
        PreventInTransactionBlock(isTopLevel, "DROP DATABASE");
        dropdb(stmt->dbname, stmt->missing_ok);
      }
      break;
      /* Query-level asynchronous notification */
    case T_NotifyStmt:
      {
        NotifyStmt *stmt = (NotifyStmt *) parsetree;

        PreventCommandDuringRecovery("NOTIFY");
        Async_Notify(stmt->conditionname, stmt->payload);
      }
      break;

    case T_ListenStmt:
      {
        ListenStmt *stmt = (ListenStmt *) parsetree;

        PreventCommandDuringRecovery("LISTEN");
        CheckRestrictedOperation("LISTEN");
        Async_Listen(stmt->conditionname);
      }
      break;

    case T_UnlistenStmt:
      {
        UnlistenStmt *stmt = (UnlistenStmt *) parsetree;

        /* we allow UNLISTEN during recovery, as it's a noop */
        CheckRestrictedOperation("UNLISTEN");
        if (stmt->conditionname)
          Async_Unlisten(stmt->conditionname);
        else
          Async_UnlistenAll();
      }
      break;

    case T_LoadStmt:
      {
        LoadStmt   *stmt = (LoadStmt *) parsetree;

        closeAllVfds(); /* probably not necessary... */
        /* Allowed names are restricted if you're not superuser */
        load_file(stmt->filename, !superuser());
      }
      break;

    case T_CallStmt:
      ExecuteCallStmt(castNode(CallStmt, parsetree), params, isAtomicContext,
dest);
      break;

    case T_ClusterStmt:
      /* we choose to allow this during "read only" transactions */
      PreventCommandDuringRecovery("CLUSTER");
      /* forbidden in parallel mode due to CommandIsReadOnly */
      cluster((ClusterStmt *) parsetree, isTopLevel);
      break;

    case T_VacuumStmt:
      {
        VacuumStmt *stmt = (VacuumStmt *) parsetree;

        /* we choose to allow this during "read only" transactions */
        PreventCommandDuringRecovery(stmt->is_vacuumcmd ?
                       "VACUUM" : "ANALYZE");
        /* forbidden in parallel mode due to CommandIsReadOnly */
        ExecVacuum(pstate, stmt, isTopLevel);
      }
      break;

    case T_ExplainStmt:
      ExplainQuery(pstate, (ExplainStmt *) parsetree, queryString, params,
             queryEnv, dest);
      break;

    case T_AlterSystemStmt:
      PreventInTransactionBlock(isTopLevel, "ALTER SYSTEM");
      AlterSystemSetConfigFile((AlterSystemStmt *) parsetree);
      break;

    case T_VariableSetStmt:
      ExecSetVariableStmt((VariableSetStmt *) parsetree, isTopLevel);
      break;

    case T_VariableShowStmt:
      {
        VariableShowStmt *n = (VariableShowStmt *) parsetree;

        GetPGVariable(n->name, dest);
      }
      break;

    case T_DiscardStmt:
      /* should we allow DISCARD PLANS? */
      CheckRestrictedOperation("DISCARD");
      DiscardCommand((DiscardStmt *) parsetree, isTopLevel);
      break;

    case T_CreateEventTrigStmt:
      /* no event triggers on event triggers */
      CreateEventTrigger((CreateEventTrigStmt *) parsetree);
      break;

    case T_AlterEventTrigStmt:
      /* no event triggers on event triggers */
      AlterEventTrigger((AlterEventTrigStmt *) parsetree);
      break;

      /*
       * ******************************** ROLE statements ****
       */
    case T_CreateRoleStmt:
      /* no event triggers for global objects */
      CreateRole(pstate, (CreateRoleStmt *) parsetree);
      break;

    case T_AlterRoleStmt:
      /* no event triggers for global objects */
      AlterRole((AlterRoleStmt *) parsetree);
      break;

    case T_AlterRoleSetStmt:
      /* no event triggers for global objects */
      AlterRoleSet((AlterRoleSetStmt *) parsetree);
      break;

    case T_DropRoleStmt:
      /* no event triggers for global objects */
      DropRole((DropRoleStmt *) parsetree);
      break;

    case T_ReassignOwnedStmt:
      /* no event triggers for global objects */
      ReassignOwnedObjects((ReassignOwnedStmt *) parsetree);
      break;

    case T_LockStmt:

      /*
       * Since the lock would just get dropped immediately, LOCK TABLE
       * outside a transaction block is presumed to be user error.
       */
      RequireTransactionBlock(isTopLevel, "LOCK TABLE");
      /* forbidden in parallel mode due to CommandIsReadOnly */
      LockTableCommand((LockStmt *) parsetree);
      break;

    case T_ConstraintsSetStmt:
      WarnNoTransactionBlock(isTopLevel, "SET CONSTRAINTS");
      AfterTriggerSetState((ConstraintsSetStmt *) parsetree);
      break;

    case T_CheckPointStmt:
      if (!superuser())
        ereport(ERROR,
            (errcode(ERRCODE_INSUFFICIENT_PRIVILEGE),
             errmsg("must be superuser to do CHECKPOINT")));

      /*
       * You might think we should have a PreventCommandDuringRecovery()
       * here, but we interpret a CHECKPOINT command during recovery as
       * a request for a restartpoint instead. We allow this since it
       * can be a useful way of reducing switchover time when using
       * various forms of replication.
       */
      RequestCheckpoint(CHECKPOINT_IMMEDIATE | CHECKPOINT_WAIT |
                (RecoveryInProgress() ? 0 : CHECKPOINT_FORCE));
      break;

    case T_ReindexStmt:
      {
        ReindexStmt *stmt = (ReindexStmt *) parsetree;

        if (stmt->concurrent)
          PreventInTransactionBlock(isTopLevel,
                        "REINDEX CONCURRENTLY");

        /* we choose to allow this during "read only" transactions */
        PreventCommandDuringRecovery("REINDEX");
        /* forbidden in parallel mode due to CommandIsReadOnly */
        switch (stmt->kind)
        {
          case REINDEX_OBJECT_INDEX:
            ReindexIndex(stmt->relation, stmt->options, stmt->concurrent);
            break;
          case REINDEX_OBJECT_TABLE:
            ReindexTable(stmt->relation, stmt->options, stmt->concurrent);
            break;
          case REINDEX_OBJECT_SCHEMA:
          case REINDEX_OBJECT_SYSTEM:
          case REINDEX_OBJECT_DATABASE:

            /*
             * This cannot run inside a user transaction block; if
             * we were inside a transaction, then its commit- and
             * start-transaction-command calls would not have the
             * intended effect!
             */
            PreventInTransactionBlock(isTopLevel,
                                (stmt->kind == REINDEX_OBJECT_SCHEMA) ? "REINDEX
SCHEMA" :
                                (stmt->kind == REINDEX_OBJECT_SYSTEM) ? "REINDEX
SYSTEM" :
                                "REINDEX DATABASE");
            ReindexMultipleTables(stmt->name, stmt->kind, stmt->options, stmt-
>concurrent);
            break;
          default:
            elog(ERROR, "unrecognized object type: %d",
               (int) stmt->kind);
            break;
        }
      }
      break;

      /*
       * The following statements are supported by Event Triggers only
       * in some cases, so we "fast path" them in the other cases.
       */

    case T_GrantStmt:
      {
        GrantStmt  *stmt = (GrantStmt *) parsetree;

        if (EventTriggerSupportsObjectType(stmt->objtype))
          ProcessUtilitySlow(pstate, pstmt, queryString,
                      context, params, queryEnv,
                      dest, completionTag);
        else
          ExecuteGrantStmt(stmt);
      }
      break;

    case T_DropStmt:
      {
        DropStmt   *stmt = (DropStmt *) parsetree;

        if (EventTriggerSupportsObjectType(stmt->removeType))
          ProcessUtilitySlow(pstate, pstmt, queryString,
                      context, params, queryEnv,
                      dest, completionTag);
        else
          ExecDropStmt(stmt, isTopLevel);
      }
      break;

    case T_RenameStmt:
      {
        RenameStmt *stmt = (RenameStmt *) parsetree;

        if (EventTriggerSupportsObjectType(stmt->renameType))
          ProcessUtilitySlow(pstate, pstmt, queryString,
                             context, params, queryEnv,
                             dest, completionTag);
        else
          ExecRenameStmt(stmt);
      }
      break;

    case T_AlterObjectDependsStmt:
      {
        AlterObjectDependsStmt *stmt = (AlterObjectDependsStmt *) parsetree;

        if (EventTriggerSupportsObjectType(stmt->objectType))
          ProcessUtilitySlow(pstate, pstmt, queryString,
                             context, params, queryEnv,
                             dest, completionTag);
        else
          ExecAlterObjectDependsStmt(stmt, NULL);
      }
      break;

    case T_AlterObjectSchemaStmt:
      {
        AlterObjectSchemaStmt *stmt = (AlterObjectSchemaStmt *) parsetree;

        if (EventTriggerSupportsObjectType(stmt->objectType))
          ProcessUtilitySlow(pstate, pstmt, queryString,
                             context, params, queryEnv,
                             dest, completionTag);
        else
          ExecAlterObjectSchemaStmt(stmt, NULL);
      }
      break;

    case T_AlterOwnerStmt:
      {
        AlterOwnerStmt *stmt = (AlterOwnerStmt *) parsetree;

        if (EventTriggerSupportsObjectType(stmt->objectType))
          ProcessUtilitySlow(pstate, pstmt, queryString,
                             context, params, queryEnv,
                             dest, completionTag);
        else
          ExecAlterOwnerStmt(stmt);
      }
      break;
    case T_CommentStmt:
      {
        CommentStmt *stmt = (CommentStmt *) parsetree;

        if (EventTriggerSupportsObjectType(stmt->objtype))
          ProcessUtilitySlow(pstate, pstmt, queryString,
                     context, params, queryEnv,
                     dest, completionTag);
        else
          CommentObject(stmt);
        break;
      }

    case T_SecLabelStmt:
      {
        SecLabelStmt *stmt = (SecLabelStmt *) parsetree;

        if (EventTriggerSupportsObjectType(stmt->objtype))
          ProcessUtilitySlow(pstate, pstmt, queryString,
                     context, params, queryEnv,
                     dest, completionTag);
        else
          ExecSecLabelStmt(stmt);
        break;
      }

    default:
      /* All other statement types have event trigger support */
      ProcessUtilitySlow(pstate, pstmt, queryString,
                 context, params, queryEnv,
                 dest, completionTag);
      break;
  }

  free_parsestate(pstate);

  /*
   * Make effects of commands visible, for instance so that
   * PreCommit_on_commit_actions() can see them (see for example bug
   * #15631).
   */
  CommandCounterIncrement();
}
```

函数`ProcessUtilitySlow`

源文件

```
src/backend/tcop/utility.c
```

函数代码

```
/*
 * The "Slow" variant of ProcessUtility should only receive statements
 * supported by the event triggers facility.  Therefore, we always
 * perform the trigger support calls if the context allows it.
 */
static void
ProcessUtilitySlow(ParseState *pstate,
           PlannedStmt *pstmt,
           const char *queryString,
           ProcessUtilityContext context,
           ParamListInfo params,
           QueryEnvironment *queryEnv,
           DestReceiver *dest,
           char *completionTag)
{
  Node      *parsetree = pstmt->utilityStmt;
  bool     isTopLevel = (context == PROCESS_UTILITY_TOPLEVEL);
  bool     isCompleteQuery = (context != PROCESS_UTILITY_SUBCOMMAND);
  bool     needCleanup;
  bool     commandCollected = false;
  ObjectAddress address;
  ObjectAddress secondaryObject = InvalidObjectAddress;

  /* All event trigger calls are done only when isCompleteQuery is true */
  needCleanup = isCompleteQuery && EventTriggerBeginCompleteQuery();

  /* PG_TRY block is to ensure we call EventTriggerEndCompleteQuery */
  PG_TRY();
  {
    if (isCompleteQuery)
      EventTriggerDDLCommandStart(parsetree);

    switch (nodeTag(parsetree))
    {
        /*
         * relation and attribute manipulation
         */
      case T_CreateSchemaStmt:
        CreateSchemaCommand((CreateSchemaStmt *) parsetree,
                queryString,
                pstmt->stmt_location,
                pstmt->stmt_len);

        /*
         * EventTriggerCollectSimpleCommand called by
         * CreateSchemaCommand
         */
        commandCollected = true;
        break;

      case T_CreateStmt:
      case T_CreateForeignTableStmt:
        {
          List      *stmts;
          ListCell    *l;

          /* Run parse analysis ... */
          stmts = transformCreateStmt((CreateStmt *) parsetree,
                        queryString);

          /* ... and do it */
          foreach(l, stmts)
          {
            Node      *stmt = (Node *) lfirst(l);

            if (IsA(stmt, CreateStmt))
            {
              Datum    toast_options;
              static char *validnsps[] = HEAP_RELOPT_NAMESPACES;

              /* Create the table itself */
              address = DefineRelation((CreateStmt *) stmt,
                           RELKIND_RELATION,
                           InvalidOid, NULL,
                           queryString);
              EventTriggerCollectSimpleCommand(address,
                               secondaryObject,
                               stmt);

              /*
               * Let NewRelationCreateToastTable decide if this
               * one needs a secondary relation too.
               */
              CommandCounterIncrement();

              /*
               * parse and validate reloptions for the toast
               * table
```

```
                 */
                toast_options = transformRelOptions((Datum) 0,
                                  ((CreateStmt *) stmt)->options,
                                  "toast",
                                  validnsps,
                                  true,
                                  false);
                (void) heap_reloptions(RELKIND_TOASTVALUE,
                             toast_options,
                             true);

                NewRelationCreateToastTable(address.objectId,
                              toast_options);
            }
            else if (IsA(stmt, CreateForeignTableStmt))
            {
                /* Create the table itself */
                address = DefineRelation((CreateStmt *) stmt,
                             RELKIND_FOREIGN_TABLE,
                             InvalidOid, NULL,
                             queryString);
                CreateForeignTable((CreateForeignTableStmt *) stmt,
                           address.objectId);
                EventTriggerCollectSimpleCommand(address,
                                 secondaryObject,
                                 stmt);
            }
            else
            {
                /*
                 * Recurse for anything else.  Note the recursive
                 * call will stash the objects so created into our
                 * event trigger context.
                 */
                PlannedStmt *wrapper;

                wrapper = makeNode(PlannedStmt);
                wrapper->commandType = CMD_UTILITY;
                wrapper->canSetTag = false;
                wrapper->utilityStmt = stmt;
                wrapper->stmt_location = pstmt->stmt_location;
                wrapper->stmt_len = pstmt->stmt_len;

                ProcessUtility(wrapper,
                         queryString,
                         PROCESS_UTILITY_SUBCOMMAND,
                         params,
                         NULL,
                         None_Receiver,
                         NULL);
              }

              /* Need CCI between commands */
              if (lnext(l) != NULL)
                CommandCounterIncrement();
            }

            /*
             * The multiple commands generated here are stashed
             * individually, so disable collection below.
             */
            commandCollected = true;
          }
          break;

        case T_AlterTableStmt:
          {
            AlterTableStmt *atstmt = (AlterTableStmt *) parsetree;
            Oid     relid;
            List     *stmts;
            ListCell   *l;
            LOCKMODE  lockmode;

            /*
             * Figure out lock mode, and acquire lock.  This also does
             * basic permissions checks, so that we won't wait for a
             * lock on (for example) a relation on which we have no
             * permissions.
             */
            lockmode = AlterTableGetLockLevel(atstmt->cmds);
            relid = AlterTableLookupRelation(atstmt, lockmode);

            if (OidIsValid(relid))
            {
              /* Run parse analysis ... */
              stmts = transformAlterTableStmt(relid, atstmt,
                              queryString);

              /* ... ensure we have an event trigger context ... */
              EventTriggerAlterTableStart(parsetree);
              EventTriggerAlterTableRelid(relid);

              /* ... and do it */
              foreach(l, stmts)
              {
                Node     *stmt = (Node *) lfirst(l);

                if (IsA(stmt, AlterTableStmt))
```

```
                {
                  /* Do the table alteration proper */
                  AlterTable(relid, lockmode,
                         (AlterTableStmt *) stmt);
                }
                else
                {
                  /*
                   * Recurse for anything else.  If we need to
                   * do so, "close" the current complex-command
                   * set, and start a new one at the bottom;
                   * this is needed to ensure the ordering of
                   * queued commands is consistent with the way
                   * they are executed here.
                   */
                  PlannedStmt *wrapper;

                  EventTriggerAlterTableEnd();
                  wrapper = makeNode(PlannedStmt);
                  wrapper->commandType = CMD_UTILITY;
                  wrapper->canSetTag = false;
                  wrapper->utilityStmt = stmt;
                  wrapper->stmt_location = pstmt->stmt_location;
                  wrapper->stmt_len = pstmt->stmt_len;
                  ProcessUtility(wrapper,
                           queryString,
                           PROCESS_UTILITY_SUBCOMMAND,
                           params,
                           NULL,
                           None_Receiver,
                           NULL);
                  EventTriggerAlterTableStart(parsetree);
                  EventTriggerAlterTableRelid(relid);
                }

                /* Need CCI between commands */
                if (lnext(l) != NULL)
                  CommandCounterIncrement();
              }

              /* done */
              EventTriggerAlterTableEnd();
            }
            else
              ereport(NOTICE,
                  (errmsg("relation \"%s\" does not exist, skipping",
                      atstmt->relation->relname)));
          }
          /* ALTER TABLE stashes commands internally */
          commandCollected = true;
          break;

        case T_AlterDomainStmt:
          {
            AlterDomainStmt *stmt = (AlterDomainStmt *) parsetree;

            /*
             * Some or all of these functions are recursive to cover
             * inherited things, so permission checks are done there.
             */
            switch (stmt->subtype)
            {
              case 'T': /* ALTER DOMAIN DEFAULT */

                /*
                 * Recursively alter column default for table and,
                 * if requested, for descendants
                 */
                address =
                  AlterDomainDefault(stmt->typeName,
                             stmt->def);
                break;
              case 'N': /* ALTER DOMAIN DROP NOT NULL */
                address =
                  AlterDomainNotNull(stmt->typeName,
                             false);
                break;
              case 'O': /* ALTER DOMAIN SET NOT NULL */
                address =
                  AlterDomainNotNull(stmt->typeName,
                             true);
                break;
              case 'C': /* ADD CONSTRAINT */
                address =
                  AlterDomainAddConstraint(stmt->typeName,
                               stmt->def,
                               &secondaryObject);
                break;
              case 'X': /* DROP CONSTRAINT */
                address =
                  AlterDomainDropConstraint(stmt->typeName,
                                stmt->name,
                                stmt->behavior,
                                stmt->missing_ok);
                break;
              case 'V': /* VALIDATE CONSTRAINT */
                address =
                  AlterDomainValidateConstraint(stmt->typeName,
                                                stmt->name);
                break;
              default:  /* oops */
                elog(ERROR, "unrecognized alter domain type: %d",
                   (int) stmt->subtype);
                break;
            }
          }
          break;

          /*
           * ************ object creation / destruction *************
           */
        case T_DefineStmt:
          {
            DefineStmt *stmt = (DefineStmt *) parsetree;

            switch (stmt->kind)
            {
              case OBJECT_AGGREGATE:
                address =
                  DefineAggregate(pstate, stmt->defnames, stmt->args,
                          stmt->oldstyle,
                          stmt->definition,
                          stmt->replace);
                break;
              case OBJECT_OPERATOR:
                Assert(stmt->args == NIL);
                address = DefineOperator(stmt->defnames,
                             stmt->definition);
                break;
              case OBJECT_TYPE:
                Assert(stmt->args == NIL);
                address = DefineType(pstate,
                           stmt->defnames,
                           stmt->definition);
                break;
              case OBJECT_TSPARSER:
                Assert(stmt->args == NIL);
                address = DefineTSParser(stmt->defnames,
                             stmt->definition);
                break;
              case OBJECT_TSDICTIONARY:
                Assert(stmt->args == NIL);
                address = DefineTSDictionary(stmt->defnames,
                               stmt->definition);
                break;
              case OBJECT_TSTEMPLATE:
```

```
          Assert(stmt->args == NIL);
          address = DefineTSTemplate(stmt->defnames,
                                     stmt->definition);
          break;
        case OBJECT_TSCONFIGURATION:
          Assert(stmt->args == NIL);
          address = DefineTSConfiguration(stmt->defnames,
                                          stmt->definition,
                                          &secondaryObject);
          break;
        case OBJECT_COLLATION:
          Assert(stmt->args == NIL);
          address = DefineCollation(pstate,
                                    stmt->defnames,
                                    stmt->definition,
                                    stmt->if_not_exists);
          break;
        default:
          elog(ERROR, "unrecognized define stmt type: %d",
             (int) stmt->kind);
          break;
      }
    }
    break;

  case T_IndexStmt: /* CREATE INDEX */
    {
      IndexStmt  *stmt = (IndexStmt *) parsetree;
      Oid     relid;
      LOCKMODE  lockmode;

      if (stmt->concurrent)
        PreventInTransactionBlock(isTopLevel,
                                  "CREATE INDEX CONCURRENTLY");

      /*
       * Look up the relation OID just once, right here at the
       * beginning, so that we don't end up repeating the name
       * lookup later and latching onto a different relation
       * partway through.  To avoid lock upgrade hazards, it's
       * important that we take the strongest lock that will
       * eventually be needed here, so the lockmode calculation
       * needs to match what DefineIndex() does.
       */
      lockmode = stmt->concurrent ? ShareUpdateExclusiveLock
        : ShareLock;
      relid =
        RangeVarGetRelidExtended(stmt->relation, lockmode,
                                 0,
                              RangeVarCallbackOwnsRelation,
                              NULL);

          /*
           * CREATE INDEX on partitioned tables (but not regular
           * inherited tables) recurses to partitions, so we must
           * acquire locks early to avoid deadlocks.
           *
           * We also take the opportunity to verify that all
           * partitions are something we can put an index on, to
           * avoid building some indexes only to fail later.
           */
          if (stmt->relation->inh &&
            get_rel_relkind(relid) == RELKIND_PARTITIONED_TABLE)
          {
            ListCell   *lc;
            List     *inheritors = NIL;

            inheritors = find_all_inheritors(relid, lockmode, NULL);
            foreach(lc, inheritors)
            {
              char    relkind = get_rel_relkind(lfirst_oid(lc));

              if (relkind != RELKIND_RELATION &&
                relkind != RELKIND_MATVIEW &&
                relkind != RELKIND_PARTITIONED_TABLE &&
                relkind != RELKIND_FOREIGN_TABLE)
                elog(ERROR, "unexpected relkind \"%c\" on partition \"%s\"",
                   relkind, stmt->relation->relname);

              if (relkind == RELKIND_FOREIGN_TABLE &&
                (stmt->unique || stmt->primary))
                ereport(ERROR,
                    (errcode(ERRCODE_WRONG_OBJECT_TYPE),
                     errmsg("cannot create unique index on partitioned table
\"%s\"",
                        stmt->relation->relname),
                     errdetail("Table \"%s\" contains partitions that are
foreign tables.",
                           stmt->relation->relname)));
            }
            list_free(inheritors);
          }

          /* Run parse analysis ... */
          stmt = transformIndexStmt(relid, stmt, queryString);

          /* ... and do it */
          EventTriggerAlterTableStart(parsetree);
```

```
            address =
              DefineIndex(relid,  /* OID of heap relation */
                    stmt,
                    InvalidOid, /* no predefined OID */
                    InvalidOid, /* no parent index */
                    InvalidOid, /* no parent constraint */
                    false,  /* is_alter_table */
                    true, /* check_rights */
                    true, /* check_not_in_use */
                    false,  /* skip_build */
                    false); /* quiet */

            /*
             * Add the CREATE INDEX node itself to stash right away;
             * if there were any commands stashed in the ALTER TABLE
             * code, we need them to appear after this one.
             */
            EventTriggerCollectSimpleCommand(address, secondaryObject,
                                 parsetree);
            commandCollected = true;
            EventTriggerAlterTableEnd();
          }
          break;

        case T_CreateExtensionStmt:
          address = CreateExtension(pstate, (CreateExtensionStmt *) parsetree);
          break;

        case T_AlterExtensionStmt:
          address = ExecAlterExtensionStmt(pstate, (AlterExtensionStmt *)
parsetree);
          break;

        case T_AlterExtensionContentsStmt:
          address = ExecAlterExtensionContentsStmt((AlterExtensionContentsStmt *)
parsetree,
                                   &secondaryObject);
          break;

        case T_CreateFdwStmt:
          address = CreateForeignDataWrapper((CreateFdwStmt *) parsetree);
          break;

        case T_AlterFdwStmt:
          address = AlterForeignDataWrapper((AlterFdwStmt *) parsetree);
          break;

        case T_CreateForeignServerStmt:
          address = CreateForeignServer((CreateForeignServerStmt *) parsetree);
          break;

        case T_AlterForeignServerStmt:
          address = AlterForeignServer((AlterForeignServerStmt *) parsetree);
          break;

        case T_CreateUserMappingStmt:
          address = CreateUserMapping((CreateUserMappingStmt *) parsetree);
          break;

        case T_AlterUserMappingStmt:
          address = AlterUserMapping((AlterUserMappingStmt *) parsetree);
          break;

        case T_DropUserMappingStmt:
          RemoveUserMapping((DropUserMappingStmt *) parsetree);
          /* no commands stashed for DROP */
          commandCollected = true;
          break;

        case T_ImportForeignSchemaStmt:
          ImportForeignSchema((ImportForeignSchemaStmt *) parsetree);
          /* commands are stashed inside ImportForeignSchema */
          commandCollected = true;
          break;

        case T_CompositeTypeStmt: /* CREATE TYPE (composite) */
          {
            CompositeTypeStmt *stmt = (CompositeTypeStmt *) parsetree;

            address = DefineCompositeType(stmt->typevar,
                            stmt->coldeflist);
          }
          break;

        case T_CreateEnumStmt:  /* CREATE TYPE AS ENUM */
          address = DefineEnum((CreateEnumStmt *) parsetree);
          break;

        case T_CreateRangeStmt: /* CREATE TYPE AS RANGE */
          address = DefineRange((CreateRangeStmt *) parsetree);
          break;

        case T_AlterEnumStmt: /* ALTER TYPE (enum) */
          address = AlterEnum((AlterEnumStmt *) parsetree);
          break;

        case T_ViewStmt:  /* CREATE VIEW */
          EventTriggerAlterTableStart(parsetree);
          address = DefineView((ViewStmt *) parsetree, queryString,
                     pstmt->stmt_location, pstmt->stmt_len);
          EventTriggerCollectSimpleCommand(address, secondaryObject,
                           parsetree);
          /* stashed internally */
          commandCollected = true;
          EventTriggerAlterTableEnd();
          break;

        case T_CreateFunctionStmt:  /* CREATE FUNCTION */
          address = CreateFunction(pstate, (CreateFunctionStmt *) parsetree);
          break;

        case T_AlterFunctionStmt: /* ALTER FUNCTION */
          address = AlterFunction(pstate, (AlterFunctionStmt *) parsetree);
          break;

        case T_RuleStmt:  /* CREATE RULE */
          address = DefineRule((RuleStmt *) parsetree, queryString);
          break;

        case T_CreateSeqStmt:
          address = DefineSequence(pstate, (CreateSeqStmt *) parsetree);
          break;

        case T_AlterSeqStmt:
          address = AlterSequence(pstate, (AlterSeqStmt *) parsetree);
          break;

        case T_CreateTableAsStmt:
          address = ExecCreateTableAs((CreateTableAsStmt *) parsetree,
                        queryString, params, queryEnv,
                        completionTag);
          break;

        case T_RefreshMatViewStmt:

          /*
           * REFRESH CONCURRENTLY executes some DDL commands internally.
           * Inhibit DDL command collection here to avoid those commands
           * from showing up in the deparsed command queue.  The refresh
           * command itself is queued, which is enough.
           */
          EventTriggerInhibitCommandCollection();
          PG_TRY();
          {
            address = ExecRefreshMatView((RefreshMatViewStmt *) parsetree,
                           queryString, params, completionTag);
          }
```

```
          PG_CATCH();
          {
            EventTriggerUndoInhibitCommandCollection();
            PG_RE_THROW();
          }
          PG_END_TRY();
          EventTriggerUndoInhibitCommandCollection();
          break;

        case T_CreateTrigStmt:
          address = CreateTrigger((CreateTrigStmt *) parsetree,
                        queryString, InvalidOid, InvalidOid,
                        InvalidOid, InvalidOid, InvalidOid,
                        InvalidOid, NULL, false, false);
          break;

        case T_CreatePLangStmt:
          address = CreateProceduralLanguage((CreatePLangStmt *) parsetree);
          break;

        case T_CreateDomainStmt:
          address = DefineDomain((CreateDomainStmt *) parsetree);
          break;

        case T_CreateConversionStmt:
          address = CreateConversionCommand((CreateConversionStmt *) parsetree);
          break;

        case T_CreateCastStmt:
          address = CreateCast((CreateCastStmt *) parsetree);
          break;

        case T_CreateOpClassStmt:
          DefineOpClass((CreateOpClassStmt *) parsetree);
          /* command is stashed in DefineOpClass */
          commandCollected = true;
          break;

        case T_CreateOpFamilyStmt:
          address = DefineOpFamily((CreateOpFamilyStmt *) parsetree);
          break;

        case T_CreateTransformStmt:
          address = CreateTransform((CreateTransformStmt *) parsetree);
          break;

        case T_AlterOpFamilyStmt:
          AlterOpFamily((AlterOpFamilyStmt *) parsetree);
          /* commands are stashed in AlterOpFamily */
          commandCollected = true;
          break;

        case T_AlterTSDictionaryStmt:
          address = AlterTSDictionary((AlterTSDictionaryStmt *) parsetree);
          break;

        case T_AlterTSConfigurationStmt:
          AlterTSConfiguration((AlterTSConfigurationStmt *) parsetree);

          /*
           * Commands are stashed in MakeConfigurationMapping and
           * DropConfigurationMapping, which are called from
           * AlterTSConfiguration
           */
          commandCollected = true;
          break;

        case T_AlterTableMoveAllStmt:
          AlterTableMoveAll((AlterTableMoveAllStmt *) parsetree);
          /* commands are stashed in AlterTableMoveAll */
          commandCollected = true;
          break;

        case T_DropStmt:
          ExecDropStmt((DropStmt *) parsetree, isTopLevel);
          /* no commands stashed for DROP */
          commandCollected = true;
          break;

        case T_RenameStmt:
          address = ExecRenameStmt((RenameStmt *) parsetree);
          break;

        case T_AlterObjectDependsStmt:
          address =
            ExecAlterObjectDependsStmt((AlterObjectDependsStmt *) parsetree,
                           &secondaryObject);
          break;

        case T_AlterObjectSchemaStmt:
          address =
            ExecAlterObjectSchemaStmt((AlterObjectSchemaStmt *) parsetree,
                          &secondaryObject);
          break;

        case T_AlterOwnerStmt:
          address = ExecAlterOwnerStmt((AlterOwnerStmt *) parsetree);
          break;
```

```
      case T_AlterOperatorStmt:
        address = AlterOperator((AlterOperatorStmt *) parsetree);
        break;

      case T_CommentStmt:
        address = CommentObject((CommentStmt *) parsetree);
        break;

      case T_GrantStmt:
        ExecuteGrantStmt((GrantStmt *) parsetree);
        /* commands are stashed in ExecGrantStmt_oids */
        commandCollected = true;
        break;

      case T_DropOwnedStmt:
        DropOwnedObjects((DropOwnedStmt *) parsetree);
        /* no commands stashed for DROP */
        commandCollected = true;
        break;

      case T_AlterDefaultPrivilegesStmt:
        ExecAlterDefaultPrivilegesStmt(pstate, (AlterDefaultPrivilegesStmt *)
parsetree);
        EventTriggerCollectAlterDefPrivs((AlterDefaultPrivilegesStmt *)
parsetree);
        commandCollected = true;
        break;

      case T_CreatePolicyStmt:  /* CREATE POLICY */
        address = CreatePolicy((CreatePolicyStmt *) parsetree);
        break;

      case T_AlterPolicyStmt: /* ALTER POLICY */
        address = AlterPolicy((AlterPolicyStmt *) parsetree);
        break;

      case T_SecLabelStmt:
        address = ExecSecLabelStmt((SecLabelStmt *) parsetree);
        break;

      case T_CreateAmStmt:
        address = CreateAccessMethod((CreateAmStmt *) parsetree);
        break;

      case T_CreatePublicationStmt:
        address = CreatePublication((CreatePublicationStmt *) parsetree);
        break;
      case T_AlterPublicationStmt:
        AlterPublication((AlterPublicationStmt *) parsetree);

        /*
         * AlterPublication calls EventTriggerCollectSimpleCommand
         * directly
         */
        commandCollected = true;
        break;

      case T_CreateSubscriptionStmt:
        address = CreateSubscription((CreateSubscriptionStmt *) parsetree,
                       isTopLevel);
        break;

      case T_AlterSubscriptionStmt:
        address = AlterSubscription((AlterSubscriptionStmt *) parsetree);
        break;

      case T_DropSubscriptionStmt:
        DropSubscription((DropSubscriptionStmt *) parsetree, isTopLevel);
        /* no commands stashed for DROP */
        commandCollected = true;
        break;

      case T_CreateStatsStmt:
        address = CreateStatistics((CreateStatsStmt *) parsetree);
        break;

      case T_AlterCollationStmt:
        address = AlterCollation((AlterCollationStmt *) parsetree);
        break;

      default:
        elog(ERROR, "unrecognized node type: %d",
           (int) nodeTag(parsetree));
        break;
    }

    /*
     * Remember the object so that ddl_command_end event triggers have
     * access to it.
     */
    if (!commandCollected)
      EventTriggerCollectSimpleCommand(address, secondaryObject,
                       parsetree);

    if (isCompleteQuery)
    {
          EventTriggerSQLDrop(parsetree);
          EventTriggerDDLCommandEnd(parsetree);
      }
  }
  PG_CATCH();
  {
      if (needCleanup)
          EventTriggerEndCompleteQuery();
      PG_RE_THROW();
  }
  PG_END_TRY();

  if (needCleanup)
      EventTriggerEndCompleteQuery();
}
```