NAME

SPI_execute - execute a command

SYNOPSIS

int SPI_execute(const char * *command*, bool *read_only*, long *count*)

DESCRIPTION

SPI_execute executes the specified SQL command for *count* rows. If *read_only* is true, the command must be read-only, and execution overhead is somewhat reduced.

This function can only be called from a connected C function.

If *count* is zero then the command is executed for all rows that it applies to. If *count* is greater than zero, then no more than *count* rows will be retrieved; execution stops when the count is reached, much like adding a LIMIT clause to the query. For example,

```
SPI_execute("SELECT * FROM foo", true, 5);
```

will retrieve at most 5 rows from the table. Note that such a limit is only effective when the command actually returns rows. For example,

```
SPI_execute("INSERT INTO foo SELECT * FROM bar", false, 5);
```

inserts all rows from bar, ignoring the *count* parameter. However, with

```
SPI_execute("INSERT INTO foo SELECT * FROM bar RETURNING *", false, 5);
```

at most 5 rows would be inserted, since execution would stop after the fifth RETURNING result row is retrieved.

You can pass multiple commands in one string; **SPI_execute** returns the result for the command executed last. The *count* limit applies to each command separately (even though only the last result will actually be returned). The limit is not applied to any hidden commands generated by rules.

When *read_only* is false, **SPI_execute** increments the command counter and computes a new snapshot before executing each command in the string. The snapshot does not actually change if the current transaction isolation level is SERIALIZABLE or REPEATABLE READ, but in READ COMMITTED mode the snapshot update allows each command to see the results of newly committed transactions from other sessions. This is essential for consistent behavior when the commands are modifying the database.

When *read_only* is true, **SPI_execute** does not update either the snapshot or the command counter, and it allows only plain **SELECT** commands to appear in the command string. The commands are executed using the snapshot previously established for the surrounding query. This execution mode is somewhat faster than the read/write mode due to eliminating per-command overhead. It also allows genuinely stable functions to be built: since successive executions will all use the same snapshot, there will be no change in the results.

It is generally unwise to mix read-only and read-write commands within a single function using SPI; that could result in very confusing behavior, since the read-only queries would not see the results of any database updates done by the read-write queries.

The actual number of rows for which the (last) command was executed is returned in the global variable *SPI_processed*. If the return value of the function is SPI_OK_SELECT, SPI_OK_INSERT_RETURNING, SPI_OK_DELETE_RETURNING, or SPI_OK_UPDATE_RETURNING, then you can use the global pointer SPITupleTable *SPI_tuptable to access the result rows. Some utility commands (such as **EXPLAIN**) also return row sets, and SPI_tuptable will contain the result in these cases too. Some utility commands (**COPY**, **CREATE TABLE AS**) don’t return a row set, so SPI_tuptable is NULL, but they still return the number of rows processed in *SPI_processed*.

The structure SPITupleTable is defined thus:

```
typedef struct SPITupleTable
{
   /* Public members */
   TupleDesc   tupdesc;       /* tuple descriptor */
   HeapTuple  *vals;          /* array of tuples */
   uint64      numvals;       /* number of valid tuples */

   /* Private members, not intended for external callers */
   uint64      alloced;       /* allocated length of vals array */
   MemoryContext tuptabcxt;   /* memory context of result table */
   slist_node  next;          /* link for internal bookkeeping */
   SubTransactionId subid;    /* subxact in which tuptable was created */
} SPITupleTable;
```

The fields tupdesc, vals, and numvals can be used by SPI callers; the remaining fields are internal. vals is an array of pointers to rows. The number of rows is given by numvals (for somewhat historical reasons, this count is also returned in *SPI_processed*). tupdesc is a row descriptor which you can pass to SPI functions dealing with rows.

SPI_finish frees all SPITupleTables allocated during the current C function. You can free a particular result table earlier, if you are done with it, by calling **SPI_freetuptable**.

ARGUMENTS

const char * *command*
    string containing command to execute

bool *read_only*
    true for read-only execution

long *count*
    maximum number of rows to return, or 0 for no limit

RETURN VALUE

If the execution of the command was successful then one of the following (nonnegative) values will be returned:

SPI_OK_SELECT
    if a **SELECT** (but not **SELECT INTO**) was executed

SPI_OK_SELINTO
    if a **SELECT INTO** was executed

SPI_OK_INSERT
    if an **INSERT** was executed

SPI_OK_DELETE
    if a **DELETE** was executed

SPI_OK_UPDATE
    if an **UPDATE** was executed

SPI_OK_MERGE
    if a **MERGE** was executed

SPI_OK_INSERT_RETURNING
    if an **INSERT RETURNING** was executed

SPI_OK_DELETE_RETURNING
    if a **DELETE RETURNING** was executed

SPI_OK_UPDATE_RETURNING
: if an **UPDATE RETURNING** was executed

SPI_OK_UTILITY
: if a utility command (e.g., **CREATE TABLE**) was executed

SPI_OK_REWRITTEN
: if the command was rewritten into another kind of command (e.g., **UPDATE** became an **INSERT**) by a rule.

On error, one of the following negative values is returned:

SPI_ERROR_ARGUMENT
: if *command* is NULL or *count* is less than 0

SPI_ERROR_COPY
: if **COPY TO stdout** or **COPY FROM stdin** was attempted

SPI_ERROR_TRANSACTION
: if a transaction manipulation command was attempted (**BEGIN**, **COMMIT**, **ROLLBACK**, **SAVEPOINT**, **PREPARE TRANSACTION**, **COMMIT PREPARED**, **ROLLBACK PREPARED**, or any variant thereof)

SPI_ERROR_OPUNKNOWN
: if the command type is unknown (shouldn’t happen)

SPI_ERROR_UNCONNECTED
: if called from an unconnected C function

NOTES

All SPI query-execution functions set both *SPI_processed* and *SPI_tuptable* (just the pointer, not the contents of the structure). Save these two global variables into local C function variables if you need to access the result table of **SPI_execute** or another query-execution function across later calls.