The Function Manager

1. General

The abstraction of functions is provided by the Function Manager layer. A function in POSTGRES falls into one of the following categories: built-in functions, C functions or POSTQUEL functions. Each function is represented by its unique Object ID (also known as a Procedure ID).

The built-in functions are a set of selected C procedures compiled into the backend. They are used by built-in operators, access methods, etc. User-defined functions yield either C functions (when declared with `language="C"`) or POSTQUEL functions (when declared with `language="postquel"`).

This layer is intended to provide uniform access to a function regardless of its type and bindings (ie. static or dynamic). Unfortunately, invoking POSTQUEL functions requires more sophistication. (Note that function and procedure is used interchangeably to mean the same thing in this document.)

2. The Internals

2.1. The System Catalog pg_proc

Each function has an entry in the system catalog, `pg_proc`. The structure and declaration of an entry is as follows:

```
CATALOG(pg_proc) BOOTSTRAP {
    char16      proname;      /* procedure name */
    oid         proowner;     /* procedure owner */
    oid         prolang;      /* procedure language */
    bool        proisinh;     /* is procedure inheritable? */
    bool        proistrusted; /* is procedure spawned or called? */
    bool        proiscachable;/* is value precomputable? */
    int2        pronargs;     /* number of arguments */
    bool        proretset;    /* does procedure return set of tuples */
    oid         prorettype;   /* procedure return type */
    oid8        proargtypes;  /* argument types */
      int4        probyte_pct;      /* percentage of size of input */
      int4        properbyte_cpu;   /* cost per byte of operand */
      int4        propercall_cpu;   /* CPU time per call */
      int4        prooutin_ratio; /* ratio of output to input sizes */
    text        prosrc;     /* query string for postquel functions */
    bytea       probin;     /* object file name for C procedures */
} FormData_pg_proc;

typedef FormData_pg_proc      *Form_pg_proc;
```

The `prolang` field specifies the category of the function. It contains the Object ID of the ‘‘language’’ the function is written in. Valid values are listed in the table below.

Language	*description*
`INTERNALlanguageId`	a built-in function
`ClanguageId`	a C function
`POSTQUELlanguageId`	a POSTQUEL function

The fields `probyte_pct`, `properbyte_cpu`, `propercall_cpu`, and `prooutin_ratio` are used by the optimizer for estimating the cost of a procedure invocation. They are described in greater detail in the POSTGRES reference manual. The field `proistrusted` indicates if the procedure is trusted or not. If the procedure is trusted it is invoked in the same address space as the POSTGRES backend, otherwise, it is run as a seperate process.

2.2. The Function Cache

For faster access, information about the functions is cached in a function cache. `lib/fcache.c` contains code which handles the function cache.

2.3. Built-in Functions

Built-in functions are collected at compile time from the source code to produce a table of entry points. The table is generated by the shell script `utils/Gen_fmgrtab.sh`. It generates `fmgrtab.c` and `fmgr.h` from `catalog/pg_proc.h`.

Each entry in the table contains the following structure (defined in `utils/fmgrtab.h`).

```
typedef struct {
        ObjectId        proid;      /* procedure Id */
        uint16          nargs;      /* number of arguments */
        func_ptr        func;       /* entry point */
} FmgrCall;
```

The table is sorted by proid. The following is an excerpt of the table. For instance, the two entries specify the Procedure Id's and number of arguments for the `textin` and `textout` procedures.

```
static FmgrCall fmgr_builtins[] = {
      ...
      { 46, 1, textin },
      { 47, 1, textout },
      ...
}
```

The directory `utils/adt`contains

2.4. Functions with Dynamic Bindings

User-defined functions are dynamically loaded into the backend on demand. The POSTQUEL `define function` command registers the procedure in the system catalog, `pg_proc`. The function is loaded into the running backend when either the POSTQUEL `load` command is encountered or the first time the function is invoked.

The code which handles dynamic loading is in `util/fmgr/dfmgr.c`. Details of dynamic loading is beyond the scope of this section.

3. Interface Routines

Routines for invoking C functions are defined in `utils/fmgr/fmgr.c`. The following interface routines can be used:

```
char *
fmgr(procedureId [, ... ] )
      ObjectId procedureId;
```

`fmgr()` takes the object ID of a procedure and its arguments. Note that you can only have eight arguments for user-defined C functions while at most nine for internal functions. `fmgr()` invokes the function and returns the result if successful and 0 if unsuccessful. For example, the following code segment taken from `catalog/index.c` illustrates how the text-in procedure, `textin()`, is invoked through `fmgr()`.

```
if (predicate != LispNil) {
      predString = lispOut(predicate);
      predText = (text *)fmgr(F_TEXTIN, predString);
      pfree(predString);
} else {
      predText = (text *)fmgr(F_TEXTIN, "");
}
```

```
char *
fmgr_c(user_fn, func_id, n_arguments, values, isNull)
        func_ptr        user_fn;     /* function entry point */
        ObjectId        func_id;     /* OID of function      */
        int             n_arguments; /* number of arguments to the function */
        FmgrValues      *values;     /* array of function arguments */
        Boolean         *isNull;
```

This routine is called by `fmgr()` for invoking a C function. It is also called directly by the executor for invoking C functions. For untrusted C functions the function pointer `user_fn` is set to null. Functions defined in the same address space (builtin functions and trusted user-defined functions) are called directly from this routine whereas functions in a different address space (untrusted functions) are handled by the routine `fmgr_ufp()` defined in `utils/fmgr/ufp.c`. If the function needs to be dynamically loaded into the backend, it will be done automatically.

POSTQUEL functions are invoked with `postquel_function` (defined in `executor/functions.c`).

```
Datum
postquel_function(funcNode, args, isNull, isDone)
     Func funcNode;
     char *args[];
     bool *isNull;
     bool *isDone;
```

This routine is used only by the executor.

To appreciate how the function manager works, one might look at the executor. In the executor `(executor/ex_qual.c)`, `ExecEvalExpr` evaluates operators and function clauses with `ExecEvalOper` and `ExecEvalFunction`. Both of them invokes `ExecMakeFunctionResult` which in turn invokes `fmgr_c()` and `postquel_function()` accordingly to evaluate the results.

Another useful interface routine is `fmgr_info()` (defined in `fmgr/fmgr.c`). It can be used to obtain information about any built-in or user-defined function and is declared as follows:

```
void
fmgr_info(procedureId, function, nargs)
        ObjectId        procedureId;
        func_ptr        *function;
        int             *nargs;
```

It takes the procedure ID of a function and returns its entry point in `function` and the number of arguments it takes in `nargs`.

For example, the following code segment also taken from `catalog/index.c` illustrates how to obtain the entry point (ie. the function pointer) of the procedure testing equality of ObjectId's, `oideq()`, through `fmgr_info()`. It stores the entry point in the index key for later use.

```
fmgr_info(ObjectIdEqualRegProcedure,
      &indexKey[0].func, &indexKey[0].nargs);
```

To use the above interfaces, you may need the following include file:

```
#include "fmgr.h"
```