Fast Path

1. General

The term *Fast Path* is used throughout the POSTGRES documentation to denote two different ideas. The first is the notion of being able to call a POSTGRES backend function directly from a frontend application. The second is the notion of being able to directly access a major subsystem of the postgres backend from within a user–defined function which is itself loaded into the backend. These are really two very different ideas and unfortunately the same term is sometimes used to describe both. This document describes the latter: accessing the major subsystems of postgres from within a backend resident user–defined function. Details of the former can be found in the LIBPQ documentation describing the `PQfn()` function, but it should be noted that this is by far less frequently used and is certainly of limited capability (constrained mainly by the fact that the frontend and backend applications are running in completely different address spaces).

A user–defined function (that is not defined as "untrusted") is loaded into the POSTGRES backend and shares the address space with it. Because of this it is possible for a user–defined function to access any and every routine resident in the backend. So at a low level, strictly speaking, the user–defined function can do anything including step on random memory and call functions that should not be called. At this low level there is no "interface" as the user–defined function just calls other functions that happen to reside in the POSTGRES backend. The purpose of the *Fast Path* interface, then, is conceptual. It is to document at a higher level just what the major subsystems in the POSTGRES backend are and what routines a user–defined function can call and get predictable and usable results. There are six functional levels that can be accessed. They are

```
1)    Postquel interface (traffic cop level)
      A function can pass a string with postquel commands.

2)    Parser
      A function can access the POSTGRES parser, getting a parse tree
      in return.

3)    Query Optimizer
      A function can call the query optimizer, passing it a parse
      tree and obtaining a query plan in return.

4)    Executor
      A function can call the executor and pass it a query plan to
      be executed.

5)    Access methods
      A function can directly call the access methods.

6)    Function manager
      A function can call other registered POSTGRES functions using this
      level.
```

The most common levels for a function to access are 1 and 5. Level 1 allows the function to perform postquel queries almost as if it were a frontend application and level 5 allows a function to access and manipulate tuples in a relation directly.

2. Level 1) Postquel interface

The internal backend function called is

```
char *
PQexec(query)
      char *query;
```

and is similar to the `PQexec()` function that a frontend application using LIBPQ would call except for some important restrictions on what you can do and how data is retrieved. In addition, the following LIBPQ functions are not available from a user–defined function in the backend.

```
PQsetdb()
PQdb()
PQreset()
PQfinish()
PQNotifies()
PQRemoveNotify()
PQgetline()
PQputline()
PQendcopy()
```

The environment variables discussed in the LIBPQ document (PGHOST, PGDATABASE, PGPORT, etc) are also not used in the backend.

2.1. Retrieving data

In a frontend application using LIBPQ a user can do a retrieve into a named portal using the "retrieve portal ..." syntax. This does not work in the backend because the backend `PQexec()` interface manages portals in a different way and also because you cannot issue LIBPQ "begin" or "end" statements. When issuing a POSTQUEL query that returns tuples, the `PQexec()` function will return a string that starts with "P" and the remainder of which is the name of the portal where the tuples are. This is not different from the frontend LIBPQ `PQexec()` function *except* that in the backend there is no default portal named "blank" and so the portal name must be looked at and passed to the `PQparray()` function to get the portal buffer. Also, each `PQexec()` that returns data in a portal allocates a new unique portal rather than overwriting data in the same one (as with the blank portal in the frontend `PQexec()`). There is also a very low limit (around 10 currently) on the number of unique portals the backend may have in use, so it is wise for the backend function to call the `PQclear()` function to deallocate the portal after you are finished accessing the data in it.

While this inconsistency between the frontend and backend handling of portals may seem like a bug at first glance, it is necessary because it is possible in the backend to have one user–defined function that uses `PQexec()` call another user–defined function that also uses `PQexec()`. If PQexec() overwrites the same buffer, the second call to PQexec() might overwrite the data from the previous PQexec() before the first function got a chance to access it. We are looking at other ways we may implement this to avoid the problem, but for now it is wise to just call PQclear() on the portal after you are finished with the data.

2.2. Other Restrictions

The backend PQexec() interface is meant to be used by a user−defined function. In general, a user−defined function might be called during the course of executing a single POSTQUEL command, and so certain semantics concerning atomicity and visibility of updates may make the POSTQUEL command that is executed by the backend function behave differently than if it were executed from the frontend LIBPQ interface. Specifically, it is not supposed to be possible for a single command to see its own updates. For example, consider the following POSTQUEL fragment:

```
The function "sal" returns the salary of the employee
whose name matches the supplied argument.

(before)
NAME  SALARY
joe   100
bob   100
jim   100

replace emp(sal = 2 * sal("bob"))

(after)
NAME  SALARY
joe   200
bob   200
jim   200
<discuss visiblity issues, commnd_counter_increment, and flag>
```

Consider if the previous command could see its own updates. The replace command would makes its way through all the employees giving them 2 times what bob makes. But as soon as "bob" himself get two times his salary, then the remaining employees would get two times his *new* salary, which would be 400 instead of 200. If the order that the replace command happened to access the tuples in was the order listed above, then the results would be:

```
(after)
NAME  SALARY
joe   200
bob   200
jim   400
```

and this would be wrong. So the normal semantics are that a single command cannot see its own updates. The reason we metion this here is that in a front−end LIBPQ application, each POSTQUEL command is indeed a separate command (and each can see the results of the previous command, but not its own), however, in the backend the POSTQUEL commands called through PQexec() from a user−defined function are all in the *same* command, and so cannot see the updates they make. If they could, then the instantiation of the user−defined function that was

running this POSTQUEL command would itself span more than one command and would break the visibility semantics.
<WHAT ABOUT USER DEFINED POSTQUEL FUNCTIONS>
This can be very confusing since the function could retrieve tuples into another relation, and then try to retrieve them out of the new relation (in the same instantiation of the function) and not find anything. Because there may be a very real and practical need for a user–defined function to see the results of updates made during the course of its execution, we have provided a new function that temporarily disables the visibility semantics that were just discussed. It is important to understand that when using these functions you are temporarily breaking the visibility semantics in POSTGRES and you must be aware of any side effects, such as in the example cited above, that this may have. The side effects could be more serious than just wrong answers.
The following function turns off the visibililty rules:

blah()   # name to be decided

The following function turns back on the visibililty rules:

blahblah()   # name to be decided

Another restriction is that since the command that calls the user defined function is already inside a transaction, and POSTGRES doesn't support nested transactions, it would be wrong to call either "begin" or "end" in the postquel that is passed to the backend PQexec().

Probably the biggest problem with using PQexec() in the backend is that currently if any error occurs in processing the POSTQUEL commands, or if the executed commands happen to call the backend `warn()` routine, then the call to PQexec() never returns. This is a restriction that we consider a bug and hope we can fix for the next release (5.0).

2.3. Examples

The following code fragment shows a user–defined function called xconnect that issues a POSTQUEL command to retrieve all the tuples from a relation.

```
int4 *xconnect (arg)
    int4 *arg;
{

    PortalBuffer *buffer;
    int n_groups;
    int n_tuples;
    int n_fields;
    int i, j, k;
    int t;
    char *retval;
    int n;
    char *portal;

    /*
     * Here we issue a POSTQUEL command to retrieve
```

```
 * all the instances from relation I
 */
retval = (char *) PQexec ("retrieve (I.all)");

/*
 * Let’s print the return code and portal name.  Note that the
 * portal name is not "blank", as it would be if this code were
 * run in the frontend.
 */
printf("PQexec returned %s\n", retval);

if (*retval == ’P’) {
  /*
   * The retrieve returnd our data in a portal.  The portal
   * name starts right after the ’P’, so point "portal"
   * one character past the ’P’
   */
  portal= retval+1;
} else {
     /*
   * Otherwise something else happened.  For brevity we’ll
   * leave out the error handling code
   */
  ...
}
/*
 * Convert the portal name to a portal buffer
 */
buffer = PQparray (portal);

/*
 * Print the tuples.
 * The rest looks just as it would in a frontend LIBPQ
 * application.
 */
n_groups = PQngroups (buffer);
t = 0;
for (k=0; k < n_groups; k++) {
  printf ("new instance group number=(%d)\n", k);
  n_tuples = PQntuplesGroup (buffer, k);
  n_fields = PQnfieldsGroup (buffer, k);

  /* Print Att-Names */
  for (i=0; i < n_fields; i++) {
      printf ("%-15s", PQfnameGroup (buffer, k, i) );
  }
  printf ("\n");
```

```
        /* Print values */
        for (i=0; i< n_tuples; i++) {
            for (j=0; j < n_fields; j++) {
             printf ("%-15s", PQgetvalue (buffer, t+i, j));
            }
            printf ("\n");
        }
        t += n_tuples;
       }

       /*
        * Now that we’re done, remember to clear the portal otherwise
        * we’ll soon run out of them.
        */
       PQclear(portal);

       return (arg);

   }
```

3. Level 2 Parser

The parser can be used to convert a POSTQUEL query string into a parse tree. The parse tree will be needed for both the planner and executor. The following routine is used to get a parse tree (from .../src/backend/parser/ylib.c ... made by gram.y):

```
parser(str, l, typev, nargs)
     char *str;
     LispValue l;
     ObjectId *typev;
     int nargs;
```

"str" is the query string, ie. "retrieve (x.all) where....". The argument "l" is a pointer to the list of parse trees where the result will be stored. <typev and nargs apply to the arguments (????) usually typecast NULLS ???> Note that the function has no return type, the result is the second argument.

Once the parse tree is obtained, it can be filtered through the rewrite rules and some other time routines. There is no single routine to call to do this. Please refer to the following section entitled "Parser and Planner Combined Call". It has all the code to do this.

Some of the functions needed to make the parse tree complete are :

```
rewritten_tree = QueryRewrite (parse_tree)
```

This takes the initial parse-tree and applies the current rules to re-write the query. It is important to note that utilities should not be rewritten so they must be moved to a seperate parse-tree list, and then re-combined after the re-write. This function is called for every non-utility parse tree.

4. Level 3) Query Optimizer (Planner)

The Planner is used to optimize the parse-tree. It can be used on every parse-tree in the list except for utilities. The planner must first be initialized for each call, using the following (code is in .../src/backend/planner/util/internal.c):

```
init_planner()
```

Then it is ready to receive a parse-tree with the following command (code in .../src/backend/planner/plan/planner.c):

```
Plan planner(parse-tree)
        LispValue parse;
```

This returns a plan of type "Plan". By looping through those two commands for every non-utility parse-tree, a plan list can be made to match the parse-tree list. You must link each Plan together to form a plan-list of type "List".

5. Parser and Planner Combined Call

The following function is available to combine the parser and planner steps described above. It is a valueable function, as it does all the details necessary to tidy up the parse-tree-list after calling parser() (code is in .../src/backend/tcop/postgres.c):

```
List pg_plan(query_string, typev, nargs, parsetreeP, dest)
    String      query_string;   /* string to execute */
    ObjectId    *typev;         /* argument types */
    int         nargs;          /* number of arguments */
    LispValue   *parsetreeP;    /* pointer to the parse trees */
    CommandDest dest;           /* where results should go */

This returns a list of plans.
query_string is a postquel query.
typev, nargs are arguments  (???)
parsetreeP is a pointer to the parse tree which will
    be created in the process
dest is the location for the output
  (used for ending transactions)

---------------------------------------------------------------

MORE ABOUT THE ARGUMENTS:

the "dest" argument which is passed to many of these functions is:

 *    - stdout is the desination only when we are running a
 *      backend without a postmaster and are returning results
 *      back to the user.
 *
```

```
 *    - a local portal buffer is the destination when a backend
 *      executes a user-defined function which calls PQexec() or
 *      PQfn().  In this case, the results are collected into a
 *      PortalBuffer which the user's function may diddle with.
 *
 *    - a remote portal buffer is the destination when we are
 *      running a backend with a frontend and the frontend executes
 *      PQexec() or PQfn().  In this case, the results are sent
 *      to the frontend via the pq_ functions (in tcop/dest.c).
 *
 *    - None is the destination when the system executes
 *      a query internally.  This is not used now but it may be
 *      useful for the parallel optimiser/executor.
 *
```

```
so dest can be

    None,               /* results are discarded */
    Debug,              /* results go to debugging output */
    Local,              /* results go in local portal buffer */
    Remote,             /* results sent to frontend process */
        CopyBegin,      /* results sent to frontend process
                  but are strings */
        CopyEnd,         /* results sent to frontend process
                  but are strings */
   RemoteInternal      /* results sent to frontend process in internal
                           (binary) form */
```

6. Level 4) Executor

The executor is called once the parse-tree-list and plan-list have been obtained. It works off both lists to produce the desired action. The user must separate the two different kinds of actions, utilities and regular commands. The executor must be called for each parse-tree in the parse-tree-list. Note that there is no plan for a utility parse-tree. Also, no matter which type you execute, you must call CommandCounterIncrement() if you have more parse-trees to execute. This makes the results visible to future queries. To check whether a parse-tree represents a utility, use:

```
if (atom(CAR(parse_tree))) (then utility) (else normal)
```

UTILITY EXECUTOR:

```
which_util = LISPVALUE_INTEGER(CAR(parsetree));
ProcessUtility(which_util, CDR(parsetree), query_string, dest);
```

(code in src/backend/tcop/utility.c)

```
void
```

```
ProcessUtility(command, args, commandString, dest)
    int         command;        /* "tag" */
    LispValue   args;
    char        *commandString;
    CommandDest dest;


Command is a integer value identifing which utility to execute
args is ????
CommandString is the actually postqeul query string
Dest is the location to send the result
NOTE: the first three arguments can be taken from the parse-tree
```

NORMAL EXECUTOR: (for non-utility queries)

(code in src/backend/tcop/pquery.c)

```
void
ProcessQuery(parsertree, plan, argv, typev, nargs, dest)
    List        parsertree;
    Plan        plan;
    char        *argv;
    ObjectId    *typev;
    int         nargs;
    CommandDest dest;


parsetree is just a single parse tree from the parser
plan is a single plan from the planner
argv, typev, nargs  is just argument descriptions
dest is the location for output to go.
```

7. Level 5) Access Methods

7.1. Pointer to Access Method documentation Calling the access method routines gives the caller the ability to open relations directly and manipulate, as well as retrieve, the tuples within. The actual interface to the access methods is explained in great detail in the document entitled "The Postgres Access Methods". All of the relevent functions are described there. Rather than duplicate that information here, it is recommended that you use that document as your reference, and instead we will provide a number of examples as a small tutorial here. From here on it is assumed that you are familar with the access method routines as described in that document.

7.2. Examples

<get exmaples from ~/mao.examples> <and from the

8. Level 6) Function Manager

<Point to function manager document>