Defininga

This tutorial will show you how to define a new operator class in POSTGRES for use with an existing access method.

There are several classes that are important in understanding how the access methods work. These will be discussed, and then a sample procedure for adding a new set of operators to an existing access method will be shown as an example. The description is interspersed with actual monitor commands and output;

PG_AM contains one instnace for every user-defined access method. Support for the HEAP access method is built into Postgres (since you have to start somewhere), but every other access method is described here. The schema is

amname: name of the access method

amowner: oid of the owner's instance in pg_user

amkind: not used at present, but set to 'o' as a place holder

amstrategies: number of strategies for this access method (see below)

am*: procedure identifiers for interface routines to the access method.
for example, regproc ids for opening, closing, and getting instances
from the access method appear here.

The object ID of the instance in PG_AM is used as a foreign key in lots of other classes. For BTREES, this object ID is 400. You don't need to add a new instance to this class; all you're interested in is the object ID of the BTREE instance.

```
*retrieve (pg_am.oid) where pg_am.amname = "btree"
---------------
| oid         |
---------------
| 400         |
---------------
```

The AMSTRATEGIES attribute exists to standardize comparisons across data types. For example, BTREES impose a strict ordering on keys, less to greater. Since POSTGRES allows the user to define operators, POSTGRES cannot in general look at the name of an operator (eg, ">", "<") and tell what sort of comparison it is. In fact, some access methods (like rtrees) don't impose a less to greater ordering, but some other ordering, like containment. POSTGRES needs some consistent way of taking a scan qualification, looking at the operator, deciding if a usable index exists, and rewriting the query qualification in order to improve access speeds. This implies that POSTGRES needs to know, for example, that "<=" and ">" partition a BTREE. Strategies is the way that we do this.

Defining a new set of strategies is beyond the scope of this discussion, but how the BTREE strategies work will be explained, since you'll need to know that to add a new operator class. In the PG_AM class, the AMSTRATEGIES attribute is the number of strategies defined for this access method. For BTREES, this number is 5. These strategies correspond to

less than	1
less than or equal	2
equal	3
greater than or equal	4
greater than	5

The idea is that you'll add procedures corresponding to the comparisons above to the PG_AMOP relation (see below). The access method code can use these numbers, regardless of data type, to figure out how to partition the BTREE, compute selectivity, and so on. Don't worry about the details of adding procedures yet; just understand that there's a set of these for int2's, int4's, oid's, and every other data type on which a BTREE can operate.

The AMSTRATEGIES entry in PG_AM is just the *number* of strategies defined for the access method in question. The procedures for less than, less equal, and so on don't appear in PG_AM.

The next class of interest is PG_OPCLASS. This class exists only to associate a name with an oid. In PG_AMOP, every operator class has a set of procedures, one through five, above. Some existing opclasses are int2_ops, int4_ops, and oid_ops. You need to add an instance with your opclass name (for example, "int4_abs_ops") to pg_opclass. The oid of this instance is a foreign key in other classes.

```
*append pg_opclass (opcname = "int4_abs_ops")
*retrieve (cl.oid, cl.opcname) from cl in pg_opclass
   where cl.opcname = "int4_abs_ops"
-----------------------------
| oid        | opcname     |
-----------------------------
| 17314      | int4_abs_ops|
-----------------------------
```

NOTE: The oid for your PG_OPCLASS instnace MAY BE DIFFERENT! You should substitute your value for 17314 wherever it appears in this discussion.

So now you have an access method and an operator class. You need some operators. You've added these before, and examples appear in the demo scripts and in other tutorials. First define the functions, then define operators to get POSTGRES to use the functions in the access methods. The c code that implements the functions defined is stored in the file

/users/mao/POSTGRES/demo/int4_abs.c

which is compiled to a .o file for dynamic loading by POSTGRES. The code is

```
/* int4_abs.c -- absolute value comparison functions for int4 data */
#include "tmp/c.h"

#define ABS(a) a = ((a < 0) ? -a : a)

bool int4_abs_lt(a, b) int32 a, b; { ABS(a); ABS(b); return (a < b); }
bool int4_abs_le(a, b) int32 a, b; { ABS(a); ABS(b); return (a <= b); }
bool int4_abs_eq(a, b) int32 a, b; { ABS(a); ABS(b); return (a == b); }
bool int4_abs_ge(a, b) int32 a, b; { ABS(a); ABS(b); return (a >= b); }
bool int4_abs_gt(a, b) int32 a, b; { ABS(a); ABS(b); return (a > b); }
```

There are a couple of important things that are happening below. First, note that operators for less, less equal, equal, greater equal, and greater for int4 are being defined. All of these operators are already defined for int4 under the names "<", "<=", "=", ">=", and ">". The new operators behave differently, of course. In order to guarantee that POSTGRES uses these new operators rather than the old ones, they need to be named differently from the old ones. This is a key point: you can overload operators in POSTGRES, but only if the operator isn't already defined for the argument types.

That is, if you have "<" defined for (int4, int4), you can't define it again. POSTGRES doesn't check this when you define your operator, so you can screw yourself up if you're not careful. To avoid this problem, odd names will be used for the operators. If you *do* get this wrong, the access methods are likely to crash when you try to do scans.

The other important point is that all the functions return BOOLEAN values; the access methods rely on this fact.

so here goes -- let's define some functions.

```
*define c function int4_abs_lt (
```

```
        file = "/users/mao/POSTGRES/demo/int4_abs.o",
        returntype = bool
    ) arg is (int4, int4)

*define c function int4_abs_le (
        file = "/users/mao/POSTGRES/demo/int4_abs.o",
        returntype = bool
    ) arg is (int4, int4)

*define c function int4_abs_eq (
        file = "/users/mao/POSTGRES/demo/int4_abs.o",
        returntype = bool
    ) arg is (int4, int4)

*define c function int4_abs_ge (
        file = "/users/mao/POSTGRES/demo/int4_abs.o",
        returntype = bool
    ) arg is (int4, int4)

*define c function int4_abs_gt (
        file = "/users/mao/POSTGRES/demo/int4_abs.o",
        returntype = bool
    ) arg is (int4, int4)
```

Now define the operators that use them. As noted, the operator names must be unique for two int4 operands. You can do a query on pg_operator to see if your name is taken for the types you want. The important things here are the procedure (which is the C function defined above) and the restriction and join selectivity functions. You should just use the ones used below -- note that there are different such functions for the less, equal, and greater cases. These MUST be supplied or the access method will die when it tries to use the operator. You should copy the names for restrict and join, but use the procedure names you defined in the last step.

```
*define operator <<& (arg1 = int4, arg2 = int4, procedure=int4_abs_lt,
            associativity = left, restrict = intltsel,
      join = intltjoinsel)

*define operator <=& (arg1 = int4, arg2 = int4, procedure=int4_abs_le,
            associativity = left, restrict = intltsel,
      join = intltjoinsel)

*define operator ==& (arg1 = int4, arg2 = int4, procedure=int4_abs_eq,
            associativity = left, restrict = eqsel,
      join = eqjoinsel)

*define operator >=& (arg1 = int4, arg2 = int4, procedure=int4_abs_ge,
            associativity = left, restrict = intgtsel,
      join = intgtjoinsel)

*define operator >>& (arg1 = int4, arg2 = int4, procedure=int4_abs_gt,
            associativity = left, restrict = intgtsel,
      join = intgtjoinsel)
```

Notice that five operators corresponding to less, less equal, equal, greater, and greater equal are defined. We're just about finished. the last thing we need to do is to update the PG_AMOP relation. To do this, we need the following attributes:

amopid: the oid of the PG_AM instance for BTREE (== 400, see above);

amopclaid: the oid of the pg_opclass instance for int4_abs_ops
(== whatever you got instead of 17314, see above);

amopopr: the oids of the operators for the opclass (which we'll
get in just a minute); and

amopselect,
amopnpages: cost functions.

The cost functions are used by the query optimizer to decide whether or not to use a given index in a scan. Fortunately for you, these already exist. The two functions we'll use are btreesel(), which estimates the selectivity of the btree, and btreenpage(), which estimates the number of pages a search will touch in the tree.

So we need the oids of the operators we just defined. We'll look up the names of all the operators that take two int4's, and pick ours out:

```
*retrieve (o.oid, o.oprname) from o in pg_operator, t in pg_type
   where o.oprleft = t.oid and o.oprright = t.oid
   and t.typname = "int4"

-----------------------------
| oid      | oprname   |
-----------------------------
| 96       | =         |
-----------------------------
| 97       | <         |
-----------------------------
| 514      | *         |
-----------------------------
| 518      | !=        |
-----------------------------
| 521      | >         |
-----------------------------
| 523      | <=        |
-----------------------------
| 525      | >=        |
-----------------------------
| 528      | /         |
-----------------------------
| 530      | %         |
-----------------------------
| 551      | +         |
-----------------------------
| 555      | -         |
-----------------------------  _
| 17321    | <<&       |     -----------------------------     | 17322    | <=&       |     -----------------------------     | 17
-----------------------------    /  -- the values you get for oids
| 17324    | >=&       |      /   will probably be different, and
-----------------------------   /    you should substitute them for
| 17325    | >>&       |    /     the values i use below.
-----------------------------  -
```

We can look at the operator names and pick out the ones we just added. (Of course, there are lots of other queries we could used to get the oids we wanted.)

Now we’re ready to update PG_AMOP with our new operator class. The most important thing in this entire discussion is that the operators are ordered, from less equal through greater equal, in PG_AMOP. Recall that the BTREE instance’s oid is 400 and int4_abs_ops is oid 17314. Then we add the instances we need:

```
*append pg_amop (amopid = "400"::oid,        /* btree oid */
        amopclaid = "17314"::oid,    /* pg_opclass tup */
        amopopr = "17321"::oid,    /* <<& tup oid */
        amopstrategy = "1"::int2,      /* 1 is <<& */
        amopselect = "btreesel"::regproc,
        amopnpages = "btreenpage"::regproc)

*append pg_amop (amopid = "400"::oid,        /* btree oid */
        amopclaid = "17314"::oid,    /* pg_opclass tup */
        amopopr = "17322"::oid,    /* <=& tup oid */
        amopstrategy = "2"::int2,      /* 2 is <=& */
        amopselect = "btreesel"::regproc,
        amopnpages = "btreenpage"::regproc)

*append pg_amop (amopid = "400"::oid,        /* btree oid */
        amopclaid = "17314"::oid,    /* pg_opclass tup */
        amopopr = "17323"::oid,    /* ==& tup oid */
        amopstrategy = "3"::int2,      /* 3 is ==& */
        amopselect = "btreesel"::regproc,
        amopnpages = "btreenpage"::regproc)

*append pg_amop (amopid = "400"::oid,        /* btree oid */
        amopclaid = "17314"::oid,    /* pg_opclass tup */
        amopopr = "17324"::oid,    /* >=& tup oid */
        amopstrategy = "4"::int2,      /* 4 is >=& */
        amopselect = "btreesel"::regproc,
        amopnpages = "btreenpage"::regproc)

*append pg_amop (amopid = "400"::oid,        /* btree oid */
        amopclaid = "17314"::oid,    /* pg_opclass tup */
        amopopr = "17325"::oid,    /* >>& tup oid */
        amopstrategy = "5"::int2,      /* 5 is >>& */
        amopselect = "btreesel"::regproc,
        amopnpages = "btreenpage"::regproc)
```

NOTE the order: "less" is 1, "less equal" is 2, "equal" is 3, "greater equal" is 4, and "greater" is 5.

Okay, now it’s time to test the new opclass. First we’ll create and populate a class:

```
*create pairs (name = char16, number = int4)

*append pairs (name = "mike", number = -10000)
*append pairs (name = "greg", number = 3000)
*append pairs (name = "lay peng", number = 5000)
*append pairs (name = "jeff", number = -2000)
*append pairs (name = "mao", number = 7000)
*append pairs (name = "cimarron", number = -3000)

*retrieve (pairs.all)

-----------------------------
| name        | number      |
-----------------------------
```

mike	-10000
greg	3000
lay peng	5000
jeff	-2000
mao	7000
cimarron	-3000

okay, looks pretty random. Define an index using the new opclass:

*define index pairsind on pairs using btree (number int4_abs_ops)

Now run a query that doesn't use one of our new operators. What we're trying to do here is to run a query that *won't* use our index, so that we can tell the difference when we see a query that *does* use the index. This query won't use the index because the operator we use in the qualification isn't one that appears in the list of strategies for our index.

*retrieve (pairs.all) where pairs.number < 9000

name	number
mike	-10000
greg	3000
lay peng	5000
jeff	-2000
mao	7000
cimarron	-3000

Yup, just as random; that didn't use the index. Okay, let's run a query that *does* use the index:

*retrieve (pairs.all) where pairs.number <<& 9000

name	number
jeff	-2000
cimarron	-3000
greg	3000
lay peng	5000
mao	7000

```
-----------------------------
```

Note that the "number" values are in order of increasing absolute value (as they should be, since the index was used for this scan) and that we got the right answer -- the instance for "mike" doesn’t appear, because -10000 >=& 9000.