Custom indexing with GiST and PostgreSQL

Dimitri Fontaine

October 18, 2008
Table of contents

1 Introduction: problem and existing solutions

2 Developing a GiST module
   - PostgreSQL module development
   - GiST specifics
   - GiST challenges
   - Testing, debugging, tools

3 Current status and roadmap
The **prefix** project is about solving prefix queries where a literal is compared to potential prefixes in a column data.

**Example**

```sql
SELECT ... FROM prefixes WHERE prefix @> 'abcdef';
```

You want to find rows where prefix is 'a', 'abc', 'abcd', etc.
The plain SQL way

depesz has a blog entry about it: http://www.depesz.com/index.php/2008/03/04/searching-for-longest-prefix/

Example

```sql
create table prefixes (  
id serial primary key,  
prefix text not null unique,  
operator text,  
something1 text,  
something2 text  
);
```
This works well when you know about the prefix length in your queries:

```
Example

CREATE INDEX pa1 on prefixes (prefix) 
    WHERE length(prefix) = 1;

CREATE INDEX pa2 on prefixes (prefix) 
    WHERE length(prefix) = 2;

CREATE INDEX pa3 on prefixes (substring(prefix for 3))
    WHERE length(prefix) >= 3;
```
This works well when you know about the prefix length in your queries:

Example

```sql
select * from prefixes
where ( length(prefix) = 1 and prefix = ? )
or ( length(prefix) = 2 and prefix = ? )
or ( length(prefix) >= 3
  and substring(prefix for 3) = ? )
order by length(prefix) desc
limit 1;
```
depesz thought of simply using a list of generated prefixes of phone number. For example for phone number 0123456789, we would have: `prefix in ('0', '01', '012', '0123', ...').`
The generic solution here is the specialized **GiST** index.

**Example**

```
CREATE INDEX idx_prefix ON prefixes
    USING GIST(prefix gist_prefix_ops);

SELECT ... FROM prefixes WHERE prefix @> 'abcdef';
```

So let’s talk about developing this solution!
What’s GiST?

A kind of index for PostgreSQL: Generalized Search Tree.
What’s GiST?

A kind of index for PostgreSQL: Generalized Search Tree.

PostgreSQL supports several kinds of indexes:
What’s GiST?

A kind of index for PostgreSQL: Generalized Search Tree.

PostgreSQL supports several kinds of indexes:

- BTree
What’s GiST?

A kind of index for PostgreSQL: Generalized Search Tree.

PostgreSQL supports several kinds of indexes:

- BTree
- Hash
What’s GiST?

A kind of index for PostgreSQL: Generalized Search Tree.

PostgreSQL supports several kinds of indexes:

- BTree
- Hash
- GiST
What’s GiST?

A kind of index for PostgreSQL: Generalized Search Tree.

PostgreSQL supports several kinds of indexes:

- BTree
- Hash
- GiST
- GIN
What’s GiST?

A kind of index for PostgreSQL: Generalized Search Tree.

PostgreSQL supports several kinds of indexes:

- BTree
- Hash
- GiST
- GIN
What’s GiST?

A kind of index for PostgreSQL: Generalized Search Tree.

PostgreSQL supports several kinds of indexes:

- BTree
- Hash
- GiST
- GIN

What’s special about GiST?

- balanced index
A kind of index for PostgreSQL: Generalized Search Tree.

PostgreSQL supports several kinds of indexes:

- BTree
- Hash
- GiST
- GIN

What’s special about GiST?

- balanced index
- tree-structured access method
What’s GiST?

What’s GiST?

A kind of index for PostgreSQL: Generalized Search Tree.

PostgreSQL supports several kinds of indexes:

- BTree
- Hash
- GiST
- GIN

What’s special about GiST?

- balanced index
- tree-structured access method
- acts as a base template
What’s GiST?

A kind of index for PostgreSQL: Generalized Search Tree.

PostgreSQL supports several kinds of indexes:

- BTree
- Hash
- GiST
- GIN

What’s special about GiST?

- balanced index
- tree-structured access method
- acts as a base template

It’s a kind of a plug-in index system, easy enough to work with to plug your own datatype smartness into PostgreSQL index searches.
Developing a GiST indexing module

Big picture steps:

- internal representation of data
Developing a GiST indexing module

Big picture steps:

- internal representation of data
- a *standard* PostgreSQL extension module
Developing a GiST indexing module

Big picture steps:

- internal representation of data
- a *standard* PostgreSQL extension module
- exporting C functions in SQL
Developing a GiST indexing module

Big picture steps:

- Internal representation of data
- A *standard* PostgreSQL extension module
- Exporting C functions in SQL
- Using *pgxs*
prefix_range datatype

Internal representation of data is the following:

Example

```c
typedef struct {
    char first;
    char last;
    char prefix[1]; /* varlena struct, data follows */
} prefix_range;
```

It came from internal representation to full new SQL visible datatype, prefix_range.
This part of the development is the same whether you’re targeting index code or general purpose code. It’s rather a steep learning curve... You’ll have to read the source.

Helpers: http://doxygen.postgresql.org/ and #postgresql

Example

DatumGetCString(
    DirectFunctionCall1(
        prefix_range_out,
        PrefixRangeGetDatum(orig)
    )
)
If you want to support multiple major versions of PostgreSQL, check PG_VERSION_NUM and... read the source to find out about discrepancies.

**Example**

```c
#if PG_VERSION_NUM / 100 == 802
#define PREFIX_VARSIZE(x) (VARSIZE(x) - VARHDRSZ)
#define PREFIX_VARDATA(x) (VARDATA(x))

#if PG_VERSION_NUM / 100 == 803
#define PREFIX_VARSIZE(x) (VARSIZE_ANY_EXHDR(x))
#define PREFIX_VARDATA(x) (VARDATA_ANY(x))
```
PostgreSQL code style uses macros to simplify raw C-structure accesses, the extension modules writers had better use the same technique.

**Example**

```c
#define DatumGetPrefixRange(X) ((prefix_range *) PREFIX_VARDATA(DatumGetPointer(X))
#define PrefixRangeGetDatum(X) PointerGetDatum(make_varlena(X))
#define PG_GETARG_PREFIX_RANGE_P(n) DatumGetPrefixRange(PG_DETOAST_DATUM(PG_GETARG_DATUM(n))
#define PG_RETURN_PREFIX_RANGE_P(x) return PrefixRangeGetDatum(x)
```
PostgreSQL has support for polymorphic and overloading functions, even at its innermost foundation: C-level code.

**Example**

```c
PG_FUNCTION_INFO_V1(prefix_range_cast_from_text);
Datum prefix_range_cast_from_text(PG_FUNCTION_ARGS)
{
    text *txt = PG_GETARG_TEXT_P(0);
    Datum cstring = DirectFunctionCall1(textout,
        PointerGetDatum(txt))
    return DirectFunctionCall1(prefix_range_in, cstring);
}
```
Here’s how to declare previous function in SQL:

```sql
CREATE OR REPLACE FUNCTION prefix_range(text)
RETURNS prefix_range
AS 'MODULE_PATHNAME', 'prefix_range_cast_from_text'
LANGUAGE 'C' IMMUTABLE STRICT;

CREATE CAST (text as prefix_range)
    WITH FUNCTION prefix_range(text) AS IMPLICIT;
```
Use `palloc` unless told not to, or when the code you’re getting inspiration from avoids `palloc` for `malloc`.

Use `palloc` unless told not to, or when the code you’re getting inspiration from avoids `palloc` for `malloc`.

`palloc` memory lives in a `Context` which is freed in one sweep at its death (end of query execution, end of transaction, etc).
PostgreSQL module development: allocating memory

- Use palloc unless told not to, or when the code you’re getting inspiration from avoids palloc for malloc.
- palloc memory lives in a Context which is freed in one sweep at its death (end of query execution, end of transaction, etc).
- PostgreSQL has support for polymorphic and overloading functions, even at the C-level.
PostgreSQL provides the tool suite for easy building and integration of your module: put the following into a Makefile

Example

```makefile
MODULES = prefix
DATA_built = prefix.sql

PGXS = $(shell pg_config --pgxs)
include $(PGXS)
```
When developing a PostgreSQL extension, you’ll find it convenient for your installation to exports DEBUG symbols and check for C-level Asserts.

Example

```
./configure --prefix=/home/dim/pgsql \  
  --enable-debug \  
  --enable-cassert
```
New datatype magic

We choose to export the internal data structure as a full type:

Example

CREATE TYPE prefix_range (  
    INPUT   = prefix_range_in,  
    OUTPUT  = prefix_range_out,  
    RECEIVE = prefix_range_recv,  
    SEND    = prefix_range_send
);
New datatype magic

We choose to export the internal data structure as a full type:

```sql
Example

dim=# select '0123'::prefix_range | '0137' as union;
union
---------
01[2-3]
(1 row)
```
We choose to export the internal data structure as a full type:

```
CREATE TABLE prefixes (  
    prefix    prefix_range primary key,  
    name      text not null,  
    shortname text,  
    state     char default 'S',  
);  
```
We choose to export the internal data structure as a full type:

```
CREATE TABLE prefixes (
    prefix prefix_range primary key,
    name text not null,
    shortname text,
    state char default 'S',
);```

SQL integration means column storage too! wow.
The GiST interface API

To code a new GiST index, one only has to code 7 functions in a dynamic module for PostgreSQL:

- consistent()
- union()
- compress()
- decompress()
- penalty()
- picksplit()
- same()
The GiST interface API

To code a new GiST index, one only has to code 7 functions in a dynamic module for PostgreSQL:

- consistent()
- union()
- compress()
- decompress()
- penalty()
- picksplit()
- same()

All entries in a subtree will share any property you implement. StrategyNumber is the operator used into the query. You get to implement your equality operator (=, pr_eq) for the internal datatype in the index.
The GiST interface API

To code a new GiST index, one only has to code 7 functions in a dynamic module for PostgreSQL:

- **union()**

  Input: a set of entries.

  Output: a new data which is consistent with all of them.

  This will form the index tree non-leaf elements, any element in a subtree has to be consistent with all the nodes atop.
The GiST interface API

To code a new GiST index, one only has to code 7 functions in a dynamic module for PostgreSQL:

- compress()
- decompress()

Index internal leaf data.

Example

```c
PG_FUNCTION_INFO_V1(gpr_compress);
Datum gpr_compress
(PG_FUNCTION_ARGS)
{
    PG_RETURN_POINTER(
        PG_GETARG_POINTER(0));
}
```
The GiST interface API

To code a new GiST index, one only has to code 7 functions in a dynamic module for PostgreSQL:

- penalty()
- picksplit()

In order for your GiST index to show up good performance characteristics, you’ll have to take extra care in implementing good versions of those two.

see next slides
The GiST interface API

To code a new GiST index, one only has to code 7 functions in a dynamic module for PostgreSQL:

- consistent()
- union()
- compress()
- decompress()
- penalty()
- picksplit()
- same()

Those functions expect *internal* datatypes as argument and return values, and store *exactly* this.

It’s easy to mess it up and have CREATE INDEX segfault. Assert() your code.
You declare *OPERATOR CLASSes* over the datatype to tell PostgreSQL how to index your data. It’s all dynamic down to the datatypes, operator and indexing support. Another *wow*.
You declare OPERATOR CLASSes over the datatype to tell PostgreSQL how to index your data. It's all dynamic down to the datatypes, operator and indexing support. Another wow.

**Example**

```sql
CREATE OPERATOR CLASS gist_prefix_range_ops
FOR TYPE prefix_range USING gist
AS
  OPERATOR 1 @>,
  FUNCTION 1 gpr_consistent (internal, prefix_range, prefix_range)
...
```
GiST penalty

Is this user data more like this one or that one?

Example

```sql
select a, b, pr_penalty(a::prefix_range, b::prefix_range)
from
order by 3 asc;
```
GiST penalty

Is this user data more like this one or that one?

Example

```sql
select a, b, pr_penalty(a::prefix_range, b::prefix_range)
from (values('095[4-5]', '0[8-9]'),
          ('095[4-5]', '0[0-9]'),
          ('095[4-5]', '[0-3]'),
          ('095[4-5]', '0'),
          ('095[4-5]', '[0-9]'),
          ('095[4-5]', '0[1-5]'),
          ('095[4-5]', '32'),
          ('095[4-5]', '[1-3]')) as t(a, b)
order by 3 asc;
```
GiST penalty

Is this user data more like this one or that one?

**Example**

```sql
select a, b, pr_penalty(a::prefix_range, b::prefix_range)
from (values
     ('095[4-5]', '32'),
     ('095[4-5]', '[1-3]')) as t(a, b)
order by 3 asc;
```
GiST penalty

Is this user data more like this one or that one?

Example

```sql
select a, b, pr_penalty(a::prefix_range, b::prefix_range)
from (values('095[4-5]', '0[8-9]'),
          ('095[4-5]', '0[0-9]'),
          ) as t(a, b)
order by 3 asc;
```
GiST penalty

Is this user data more like this one or that one?

Example

```sql
select a, b, pr_penalty(a::prefix_range, b::prefix_range)
from (values
    ('095[4-5]', '0-3'),
    ('095[4-5]', '0'),
    ('095[4-5]', '0-9'),
    ('095[4-5]', '0[1-5]'),
) as t(a, b)
order by 3 asc;
```
GiST penalty

Is this user data more like this one or that one?

Example

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>gpr_penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>095[4-5]</td>
<td>0[8-9]</td>
<td>1.52588e-05</td>
</tr>
<tr>
<td>095[4-5]</td>
<td>0[0-9]</td>
<td>1.52588e-05</td>
</tr>
<tr>
<td>095[4-5]</td>
<td>[0-3]</td>
<td>0.00390625</td>
</tr>
<tr>
<td>095[4-5]</td>
<td>0</td>
<td>0.00390625</td>
</tr>
<tr>
<td>095[4-5]</td>
<td>[0-9]</td>
<td>0.00390625</td>
</tr>
<tr>
<td>095[4-5]</td>
<td>0[1-5]</td>
<td>0.0078125</td>
</tr>
<tr>
<td>095[4-5]</td>
<td>32</td>
<td>1</td>
</tr>
<tr>
<td>095[4-5]</td>
<td>[1-3]</td>
<td>1</td>
</tr>
</tbody>
</table>
GiST picksplit

The index grows as you insert data, remember?
GiST picksplit

The index grows as you insert data, remember?

prefix picksplit first pass step: presort the GistEntryVector vector by positioning the elements sharing the non-empty prefix which is the most frequent in the distribution at the beginning of the vector.
GiST picksplit

The index grows as you insert data, remember?

prefix picksplit first pass step: presort the GistEntryVector vector by positioning the elements sharing the non-empty prefix which is the most frequent in the distribution at the beginning of the vector.

Then consume the vector by both ends, compare them and choose to move them in the left or the right side of the new subtree.
The index grows as you insert data, remember?

Example

```c
Datum pr_picksplit(GistEntryVector *entryvec,
               GIST_SPLITVEC *v,
               bool presort)
{
    OffsetNumber maxoff = entryvec->n - 1;
    GISTENTRY *ent = entryvec->vector;

    nbytes = (maxoff + 1) * sizeof(OffsetNumber);
```
The index grows as you insert data, remember?

Example

```c
listL = (OffsetNumber *) palloc(nbytes);
listR = (OffsetNumber *) palloc(nbytes);

unionL = DatumGetPrefixRange(ent[offl].key);
unionR = DatumGetPrefixRange(ent[offr].key);
```
GiST picksplit

The index grows as you insert data, remember?

Example

```c
pll = __pr_penalty(unionL, curl);
plr = __pr_penalty(unionR, curl);
prl = __pr_penalty(unionL, curr);
prr = __pr_penalty(unionR, curr);
```
GiST picksplit

The index grows as you insert data, remember?

Example

```c
if( pll <= plr && prl >= prr )    { l, r }
else if( pll > plr && prl >= prr ) { , r }
else if( pll <= plr && prl < prr ) { l, }
else if( (pll - plr) < (prr - prl) ) { all to l }
else { /* all to listR */ }
```
ART is the French Telecom Regulation Authority. It provides a list of all prefixes for local operators. Let’s load some 11966 prefixes from http://www.art-telecom.fr/fileadmin/wopnum.rtf.
ART is the French Telecom Regulation Authority. It provides a list of all prefixes for local operators. Let’s load some 11966 prefixes from http://www.art-telecom.fr/fileadmin/wopnum.rtf.

Example

```
dim=# select prefix, shortname from prefixes limit 5;
   prefix  |    shortname
----------+--------
010001[]  | COLT
010002[]  | EQFR
010003[]  | NURC
010004[]  | PROS
010005[]  | ITNF
(5 rows)
```
The gevel module allows to SQL query any GiST index!

Example
The `gevel` module allows to SQL query any GiST index!

**Example**

```
dim=# select gist_stat('idx_prefix');
Number of levels: 2
Number of pages: 63
Number of leaf pages: 62
Number of tuples: 10031
Number of invalid tuples: 0
Number of leaf tuples: 9969
Total size of tuples: 279904 bytes
Total size of leaf tuples: 278424 bytes
Total size of index: 516096 bytes
```
The gevel module allows to SQL query any GiST index!

**Example**

```sql
select *
  from gist_print('idx_prefix')
  as t(level int, valid bool, a prefix_range)
where level =1;

select *
  from gist_print('idx_prefix')
  as t(level int, valid bool, a prefix_range)
order by level;
```
Correctness testing

Even when your index builds without a segfault you have to test. It can happen at query time.
Correctness testing

Even when your index builds without a segfault you have to test. It can happen at query time, or worse:

**Example**

```sql
set enable_seqscan to on;
select * from prefixes where prefix @@ '0146640123';
select * from prefixes where prefix @@ '0100091234';
set enable_seqscan to off;
select * from prefixes where prefix @@ '0146640123';
select * from prefixes where prefix @@ '0100091234';
```
### Example

```sql
create table numbers(number text primary key);
insert into numbers
    select '01' || to_char((random()*100)::int, 'FM09')
    || to_char((random()*100)::int, 'FM09')
    || to_char((random()*100)::int, 'FM09')
    || to_char((random()*100)::int, 'FM09')
    from generate_series(1, 5000);
INSERT 0 5000
```
Performance testing

Example

dim=# explain analyze
    SELECT *
        FROM numbers n
        JOIN prefixes r
            ON r.prefix @> n.number;
Performance testing

Example

Nested Loop
(cost=0.00..4868614.00 rows=149575000 width=45)  
(actual time=0.345..4994.296 rows=10213 loops=1)

→ Seq Scan on numbers n
(cost=0.00..375.00 rows=25000 width=11)  
(actual time=0.015..12.917 rows=25000 loops=1)

→ Index Scan using idx_prefix on ranges r
(cost=0.00..104.98 rows=5983 width=34)  
(actual time=0.182..0.197 rows=0 loops=25000)

Index Cond: (r.prefix @> (n.number)::prefix_range)

Total runtime: 4998.936 ms
(5 rows)
Current release is 0.3-1 and CVS version is live! and has been involved in more than 7 million calls, 2 lookups per call
Current release is 0.3-1 and CVS version is live! 
and has been involved in more than 7 million calls, 2 lookups per call

Open item #1: add support for indexing text data directly, using prefix_range internally without the user noticing.
Current release is 0.3-1 and CVS version is live!

*and has been involved in more than 7 million calls, 2 lookups per call*

- Open item #1: add support for indexing text data directly, using prefix range internally without the user noticing.
- Open item #2: implement a simple optimisation idea (see next slide).
Current release is 0.3-1 and CVS version is live! 
_and has been involved in more than 7 million calls, 2 lookups per call_

Open item #1: add support for indexing text data directly, using _prefix_range_ internally without the user noticing.

Open item #2: implement a simple optimisation idea (see next slide).

Release Version 1.0, go into maintenance mode!
Some more optimisation

`prefix` next version will provide some more optimisation by having its internal data structure accept wider ranges of prefixes. The user visible part of this will the the input format of the `prefix_range` datatype:
Some more optimisation

prefix next version will provide some more optimisation by having its internal data structure accept wider ranges of prefixes. The user visible part of this will the the input format of the prefix_range datatype:

Example

```
SELECT 'abc[def-xyz]':prefix_range;
```
prefix project is using http://pgfoundry.org hosting facilities, has no mailing-list and currently one maintainer. Contributions and usage feedbacks are more than welcome.
prefix project is using http://pgfoundry.org hosting facilities, has no mailing-list and currently one maintainer. Contributions and usage feedbacks are more than welcome.

While developing the solution, the IRC channel #postgresql was a great resource, especially thanks to the invaluable help from RhodiumToad, formerly known as AndrewSN, Andrew Gierth.