GiN in practice

- What is GIN?
- GIN interface
- HStore – useful module
- Developing GIN extension for HStore
GIN in practice: GIN

Generalized Inverted Index

Termin 'inverted' comes from the theory of fulltext search. Usual (direct) index stores pairs id of document and some representation of text of document. Inverted index stores pairs of word from document and it's id. So, in direct index there is only one and only one entry for each document. In inverted index – as much as words in document.
GIN in practice: GIN

For optimization reasons, inverted index stores key (in fulltext – word) and list of document's ids. Usually, lists are stored as ordered, which allows their fast merge in case of search of several words.

PgSQL implementation details:
- document's id is a pointer to table's row.
- list of document's ids is stored in B-Tree to simplify its update
GIN in practice: GIN

Generalized Inverted iNdex

Although inverted indexes was invented for fulltext search, in practice they are applicable for other purposes. Documents contain variable number of data of the same nature – it is a usecase for inverted index. For example – arrays. So, inverted index may be generalized to support not only fulltext search.
GIN Structure

Entry page, level 0 (leaf)

aaa
- Pointer to posting tree: B-Tree over ItemPointer to heap

abc
- Posting list: sorted array of ItemPointer to heap

Entry page, level 0

baa  bar

Posting page, level N: ItemPointer

14:17  218:1  1021:6

Posting page, level 0 (leaf)

1:33  2:7  14:17
Right bound 14:17

Posting page, level 0 (leaf)

123:1  158:18
Right bound 218:1
GIN in practice: GIN

GIN takes care of:
• Concurrency
  • Lehman and Yao's high-concurrency B-tree management algorithm
• WAL
• Recovery

GIN is utilized like other indexes in PgSQL with a help of user-defined opclasses organized similar to GiST
GIN in practice: interface

Four interface functions (pseudocode):
- `Datum* extractValue(Datum inputValue, uint32* nentries)`
- `int compareEntry(Datum a, Datum b)`
- `Datum* extractQuery(Datum query, uint32* nentries, StrategyNumber n)`
- `bool consistent(bool check[], StrategyNumber n, Datum query)`
GIN in practice: interface

Datum* extractValue(Datum inputValue, uint32* nentries)
Returns an array of Datum of entries of the value to be indexed. nentries should contain the number of returned entries.
Tsearch2 example: inputValue is tsvector, output is array of text type, containing lexemes.
int compareEntry(Datum a, Datum b) 
Compares two entries (not the indexing values), returns <0, 0, >0 
Tsearch2 example: built-in bttextcmp(), used for built-in B-Tree index over texts.
GIN in practice: GIN

StrategyNumber – one of argument of extractQuery & consistent, pointed in CREATE OPERATOR CLASS per each operation.

Operators: int4 < int4, Overloaded \[\text{int4} = \{\text{int2} \mid \text{int4}\}\]

```
CREATE OPERATOR CLASS ... FOR TYPE int4
USING {GIN|GiST}
OPERATOR 1 = (int4,int2), -- int4 = int2
OPERATOR 2 = (int4,int8), -- int4 = int8
OPERATOR 3 < (int4,int4), -- int4 < int4
...;
```

StrategyNumber
GIN in practice: interface

Datum* extractQuery(Datum query, 
uint32* nentries, StrategyNumber n)

Returns an array of Datum of entries of 
the query to be executed. n is the 
strategy number of the operation. 
Depending on n, query can be different 
type.

Tsearch2 example: query is tsquery, 
output is array of text type, containing 
lexemes.
bool consistent(bool check[], StrategyNumber n, Datum query)  
Each element of the check array is true if the indexed value has a corresponding entry in the query: if (check[i] = TRUE) then the i-th entry of the query is present in the indexed value. The function should return true if the indexed value matches by StrategyNumber and the query.
GIN in practice: interface

- `extractValue` and consistent methods are called in short-lived memory context, which resets after every call.
- `extractQuery` and `compareEntry` called in current memory context
GIN in practice: HStore

HStore - contrib module for storing multiple (key,value) pairs in one field.

HStore in PgSQL has direct analogy in Perl language: hash. Syntax is very similar to syntax of creating hashes in Perl. Some operations:
• select 'a=>q, b=>g::hstore->'a';
  q
• select akeys('a=>1,b=>2::hstore);
  {a,b}
GIN in practice: HStore

- `akeys(hstore), avals(hstore)` – returns keys/values as an array
- `skeys(hstore), skeys(hstore)` – returns keys/values as a set of strings
- `each(hstore)` – returns set of (key,value)
- `exist(hstore, text), defined(hstore, text)` – similarly to Perl (undef is NULL)
- `hstore ? text` – equivalent for `exist()`
- `hstore @> hstore`, `hstore <@ hstore` – contains/contained operation
- `hstore || hstore` – concatenate
GIN in practice: HStore

Use cases for Hstore:
- Semi-structured data
- Variable and not-predefined number of fields
- Several slightly different objects (usual way is to store in several inherited tables or in single table with all possible fields)
Storing user settings for WEB-application:

```sql
CREATE TABLE Users (  
id int4 primary key,
login text not null unique,
firstname text,
birthdate date, --timestamp?  
...
settings hstore  
);  

UPDATE Users SET settings = settings ||  
'ShowNewsBlock=>1, NewsBlockAlign=>left' WHERE login='teodor';
```
GIN in practice: HStore

Retrieve settings:
SELECT settings->'ShowNewsBlock',
   settings->'NewsBlockAlign' FROM Users WHERE
login = 'teodor';

Perl tip:
sub NULL { return undef; }
$settings = $dbi->....
$settings =~ s/([^@%])/\$1/g; # escape Perl's
# special chars
my $hsettings = eval( "\$settings" );
if ( $hsettings->{ShowNewsBlock} ) {
  ...
}
GIN in practice: HStore

Track changes in DB:
CREATE TABLE history (  
id int4 primary key,  
user_id int4 references..., --who change?  
object_name text not null, --table name  
object_id int4 not null,  
change_date timestamp not null,  
changes hstore not null  
);

INSERT INTO history VALUES (123, 1, -- it's me  
'document', 12, -- object  
'2007-05-23 15:00',  
'title=>'GIN in pratice'' --old title  
);
GIN in practice: HStore

Internet shop:
CREATE TABLE goods (  
    id int4 primary key,  
    goods_type_id int4 not null references...,  
    name text not null,  
    info hstore not null  
);

INSERT INTO goods VALUES ( 123, 1, --mobile phone  
    'Phone name',  
    'Standard=>GSM, SubStandard=>1900MHz,  
    Localization=>RU, Color=>magenta');
Most popular queries (column 'info' is of hstore type):

- SELECT ... WHERE info->'Standard' = 'GSM';
- SELECT ... WHERE exist(info, 'Localization');
- SELECT ... WHERE exist(info,'Localization') AND info->'Standart' = 'GSM';
- SELECT ... WHERE exist(info,'Localization') AND exist(info,'Color') AND info->'Standard' = 'GSM';
GIN in practice: HStore

How to speed up?

• CREATE INDEX ... USING BTREE ((info->'Standard'));
• CREATE INDEX ... USING BTREE
  ((info ? 'Localization' ));
• To utilize index replace exist(info, 'key')
  to info ? 'key':
  SELECT ... WHERE (info ? 'Localization') = 't'
  AND (info->'Standard') = 'GSM';

• CREATE INDEX ... USING GIST (info);
• To utilize GiST index use @> operation:
  SELECT ... WHERE info ? 'Localization' AND
  info @> 'Standard=>GSM';
GIN in practice: HStore

+ B-Tree:
  - Nice looking query
  - Index may be used for ordering
  - Index allows to search value by 
    \(<,>,\leq,\geq\) operations

- B-Tree:
  - It requires two indexes per key
  - What's about other key(s)?
  - Combination of several search keys – using several indexes in one query.
GIN in practice: HStore

+GiST:
  • One index is enough for one column
  • Several clauses can be used in one index scan
- GiST:
  • Using RD-Tree over signatures: false drop problem and recheck is needed
  • Fast signature's saturation for large sets
  • Only equality operation
Possible index structure

Keys in HStore
- Standard
- Localization
- ...

Values of key 'Standart'
- GSM
- CDMA
- ...

List of pointers to rows with HStore containing key 'Standart' with value 'GSM'

List of pointers to rows with HStore containing key 'Standart' with value 'CDMA'

List of pointers to rows with HTStore containing key 'Standart'

Looks very fast for searching... But GIN can't support this scheme
GIN in practice: HStore

GIN's keys

- KEY 'Standard'
- KEY 'Localizationon'
- VALUE 'GSM'
- ....
- NULL

Posting lists (ItemPointer's sorted lists)

- List of pointer to rows with HStore contains 'Standard' key
- List of pointer to rows with HStore contains 'Locale' key
- List of pointer to rows with HStore contains 'GSM' value
- List of pointer to rows with HStore contains NULL value
To store hstore in GIN index:
• Each key, each value will be stored separately. But it's needed to distinguish key and value in index.
• NULL is stored as special value.
• Simple storage: as string. First character is reserved to flag pointing to type: key, value or NULL
GIN in practice: HStore

HStore value:
"Standard"=>"GSM",
"Localization"=>"RU",
"Color"=>NULL'

GIN's keys:
KStandard, KLocale, KColor,
VGSM, VRU, N
GIN in practice: HStore

GIN logical structure for HStore

**GIN's keys**
- KStandard
- KLocalization
- VGSM
- ...
- N

**Posting lists (ItemPointer's sorted lists)**
- List of pointer to rows with HStore contains 'Standart' key
- List of pointer to rows with HStore contains 'Locale' key
- List of pointer to rows with HStore contains 'GSM' value
- List of pointer to rows with HStore contains NULL value
GIN in practice: HStore

typedef struct {
    int32 vl_len_; // varlena header
    int4 size;    // number of pairs
        // (key,value)
    char data[1]; //storage of pairs:
        // -array of Hentry
        // -keys/values as
        //  string
} Hstore;

HStore *hstore = PG_GETARG_HS(0);
GIN in practice: HStore

typedef struct {
    uint16   keylen;  // key's length
    uint16   vallen; // value's length
    uint32   valisnull:1, // true if value is
             pos:31;  // position of key. Value is
             // placed after key
} HEntry;

Access to key's and value's value:
HEntry *ith_entry = ARRPTR(hstore) + i;
char *ith_key = STRPTR(hstore) + ith_entry->pos;
char *ith_value = ith_key + ith_entry->keylen;
GIN in practice: HStore

**Varlena header**

Number of pairs (HStore->size)

Sorted array of HEntry

HStore

<table>
<thead>
<tr>
<th>int32</th>
<th>int32</th>
<th>sizeof(HEntry) * size</th>
<th>Keys and values</th>
</tr>
</thead>
</table>

Offset to i-th key is a corresponding HEntry->pos value. Value of pair is placed after its key.
GIN in practice: HStore & GIN

extractValue method

i = 0

i < HStore->size

return entries

entries[2*i] = key => 'Kkey'

HEntry[i]->valisnull

entries[2*i+1] = 'N'

entries[2*i+1] = value => 'Vvalue'

i = i + 1
Datum gin_extract_hstore(PG_FUNCTION_ARGS) {
  Datum *entries; // return value
  for(i=0; i<hstore->size; i++) {
    HEntry ptr = ARRPTR(hstore) + i;
    // makeitem returns text* with len+1 size.
    // First character is set to first argument
    item = makeitem('K', // mark key
                    STRPTR(hstore) + ptr->pos, ptr->keylen);
    entries[ 2*i ] = PointerGetDatum(item);
    if ( ptr->valisnull ) {
      item = makeitem('N', NULL, 0 ); //mark NULL
    } else {
      item = makeitem('N', // mark value
                      STRPTR(hstore) + ptr->pos + ptr->keylen,
                      ptr->vallen );
    }
    entries[ 2*i+1 ] = PointerGetDatum(item);
  }
  PG_RETURN_POINTER(entries);
}
GIN in practice: HStore & GIN

ExtractQuery method

Strategy == HstoreContainsStrategyNumber

N

entries[0] = query -> 'Kquery'

Y

entries = extractValue(query)

return entries
Datum gin_extract_hstore_query(PG_FUNCTION_ARGS) {
    StrategyNumber strategy = PG_GETARG_UINT16(2);

    if (strategy == HstoreContainsStrategyNumber) {
        // argument is hstore, operation hstore @> hstore
        entries = gin_extract_hstore( // pseudocode!
            PG_GETARG_HS(0)
        );
    } else { // strategy == HstoreExistsStrategyNumber
        // argument is text, operation hstore ? text
        text *q = PG_GETARG_TEXT_P(0);
        nentries = 1;
        *entries = PointerGetDatum(makeitem('K', // key's mark
                                               VARDATA(q), VARSIZE(q) – VARHDRSZ));
    }

    return entries;
}
compareEntry method is built-in
bttextcmp() used for B-Tree over text
GIN in practice: HStore & GIN

Consistent method

Strategy == HstoreContainsStrategyNumber

return true

N

i = 0

Y

i < 2 * HStore_query->size

N

return false

Y

check[i] == false

Y

i = i + 1

N

i = 0

N

return true

Y

i < 2 * HStore_query->size

Y

check[i] == false

N

i = i + 1
Datum gin_consistent_hstore(PG_FUNCTION_ARGS) {
    StrategyNumber strategy = PG_GETARG_UINT16(1);
    bool res;
    if (strategy == HstoreExistsStrategyNumber) {
        // hstore ? text operation
        res = true; // exact match
    } else if (strategy == HStoreContainsStrategyNumber) {
        // hstore @> hstore operation
        bool *check = (bool *) PG_GETARG_POINTER(0);
        HStore *query = PG_GETARG_HS(2);

        res = true;
        for (i = 0; res && i < 2 * query->size; i++)
            if (check[i] == false)
                res = false;
    }

    PG_RETURN_BOOL(res);
}
GIN in practice: HStore & GIN

SELECT ... WHERE info @> 'Standard=>GSM, Locale=>RU';

extractQuery: KStandard, VGSM, KLocale, VRU

consistent returns true for:
• 'Standard=>GSM, Localization=>RU, ...'
• 'Standard=>RU, Localization=>GSM, ...'
• 'Standard=>CDMA, Localization=>US,' Something=>GSM, Code=>RU, ...'

Operation hstore @> hstore should be marked by RECHECK flag in OPERATOR CLASS.
--Create operator class
... --create methods

CREATE OPERATOR CLASS gin_hstore_ops
DEFAULT FOR TYPE hstore USING gin
AS

  OPERATOR  7       @> RECHECK,
  OPERATOR  9       ?(hstore,text),
  FUNCTION  1       bttextcmp,
  FUNCTION  2       gin_extract_hstore,
  FUNCTION  3       gin_extract_hstore_query,
  FUNCTION  4       gin_consistent_hstore,

STORAGE     text;

hstore.h:
#define HstoreContainsStrategyNumber  7
#define HstoreExistsStrategyNumber    9
GIN in practice: HStore & GIN

+ GIN
  - One index is enough for one column
  - Several clauses can be used in one index scan
  - Exist operation is fast as possible
  - Good scalability – much better than GiST

- GIN
  - Recheck is needed for contains operation (although false drops are rare than in GiST)
  - Only equality operation
  - Rather slow update/insert
  - Only one column
GIN in practice

GIN's TODO:
- Allow not only equality match
- Prefix search
- Increase the number of possible operations on GIN's key
- Full index scan
- Allow to store some additional info in index per ItemPointer
- Cost function to optimizer