# 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.

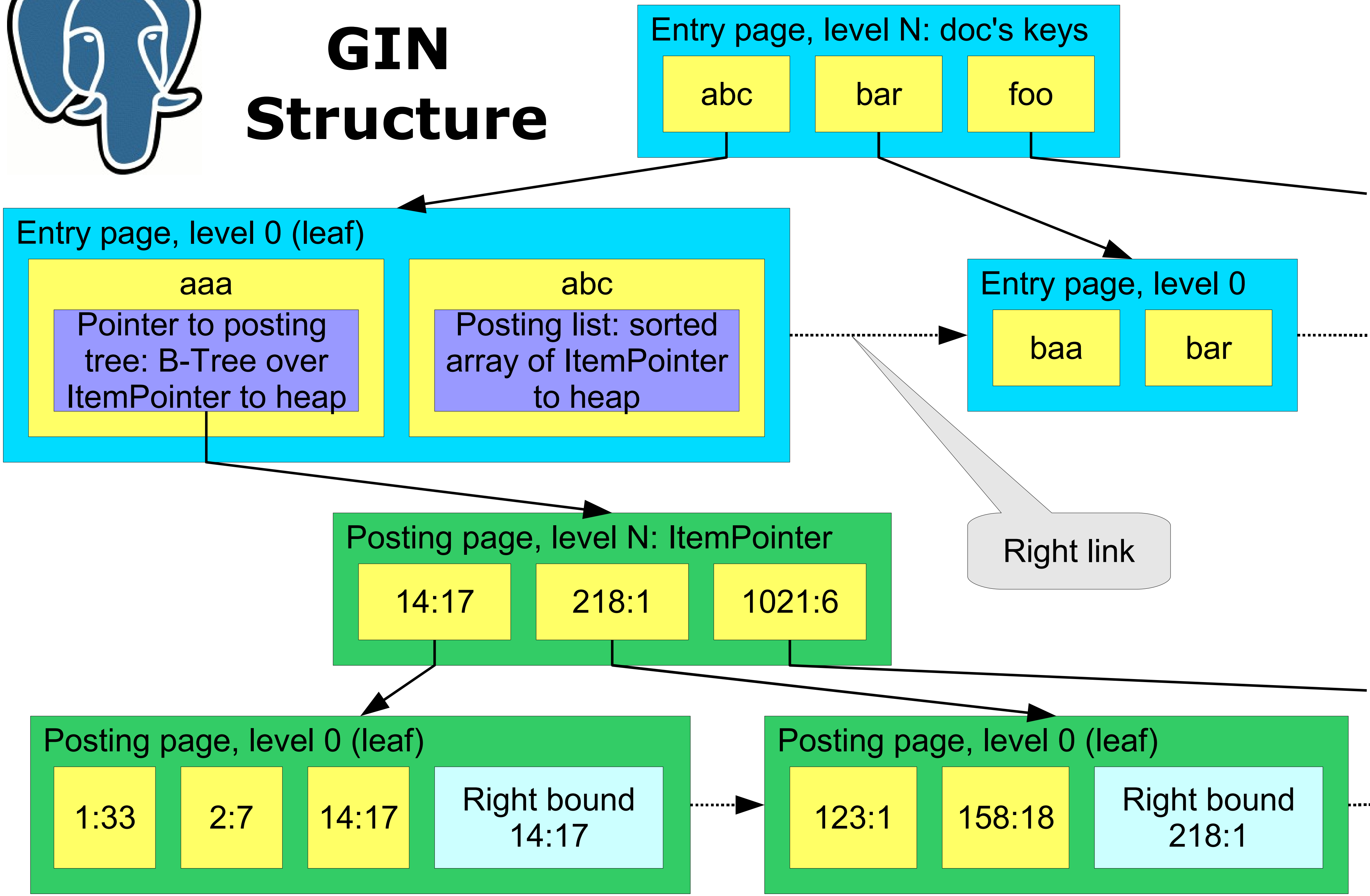
PgSQL implementation details:
- document's id is a pointer to table's row.
- list of document's ids is stored in B-Tree to simplify its update

# 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



# 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.

# GIN in practice: interface

```
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   int4 = {int2 | 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.

# GIN in practice: interface

```
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)` – retunrs 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` – containes/contained operation
- `hstore || hstore` - concatenate

# GIN in practice: HStore

Usecases 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)

# GIN in practice: HStore

Storing user settings for WEB-application:

```
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');
```

# GIN in practice: HStore

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 <,>,<=,>= 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:

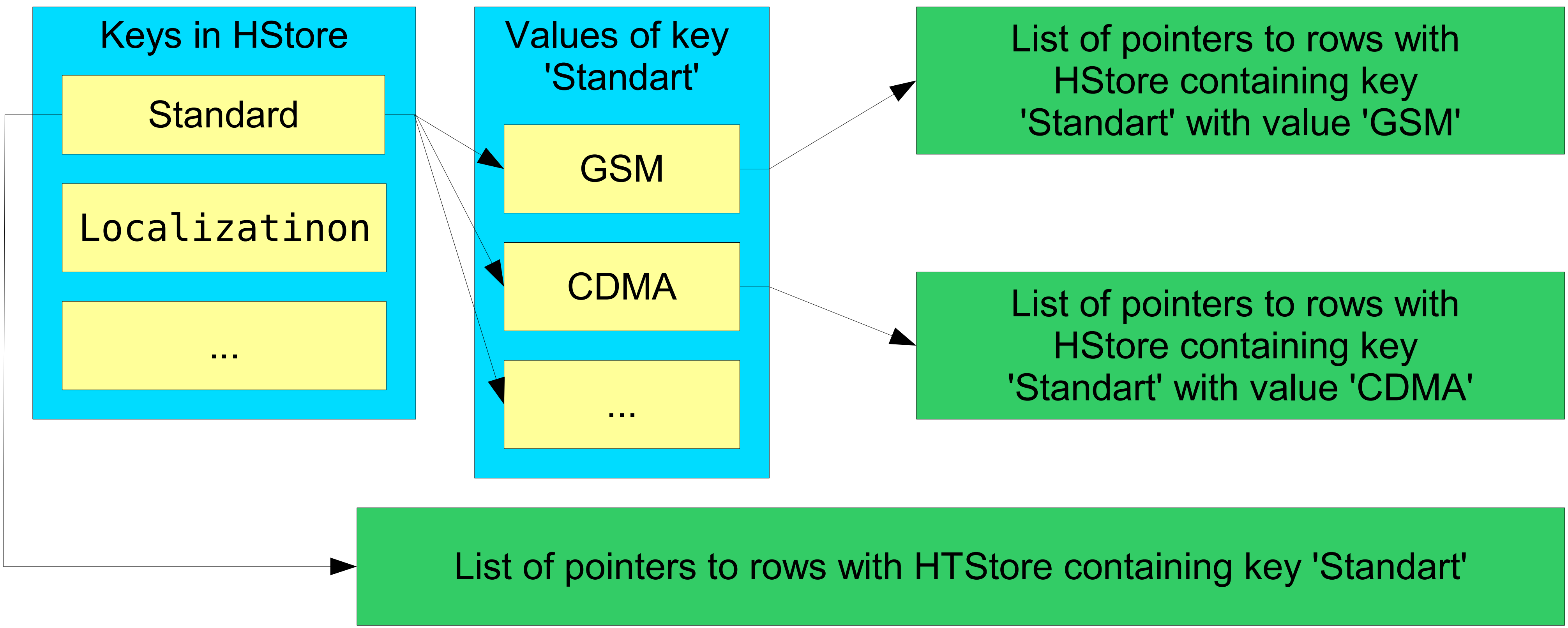
- One index  is enough for one column
- Several clauses can be used in one index scan

- GiST:

- Using RD-Tree over signatures: false drop problem and recheck is needed
- Fast signature's saturation for large sets
- Only equality operation

# GIN in practice: HStore

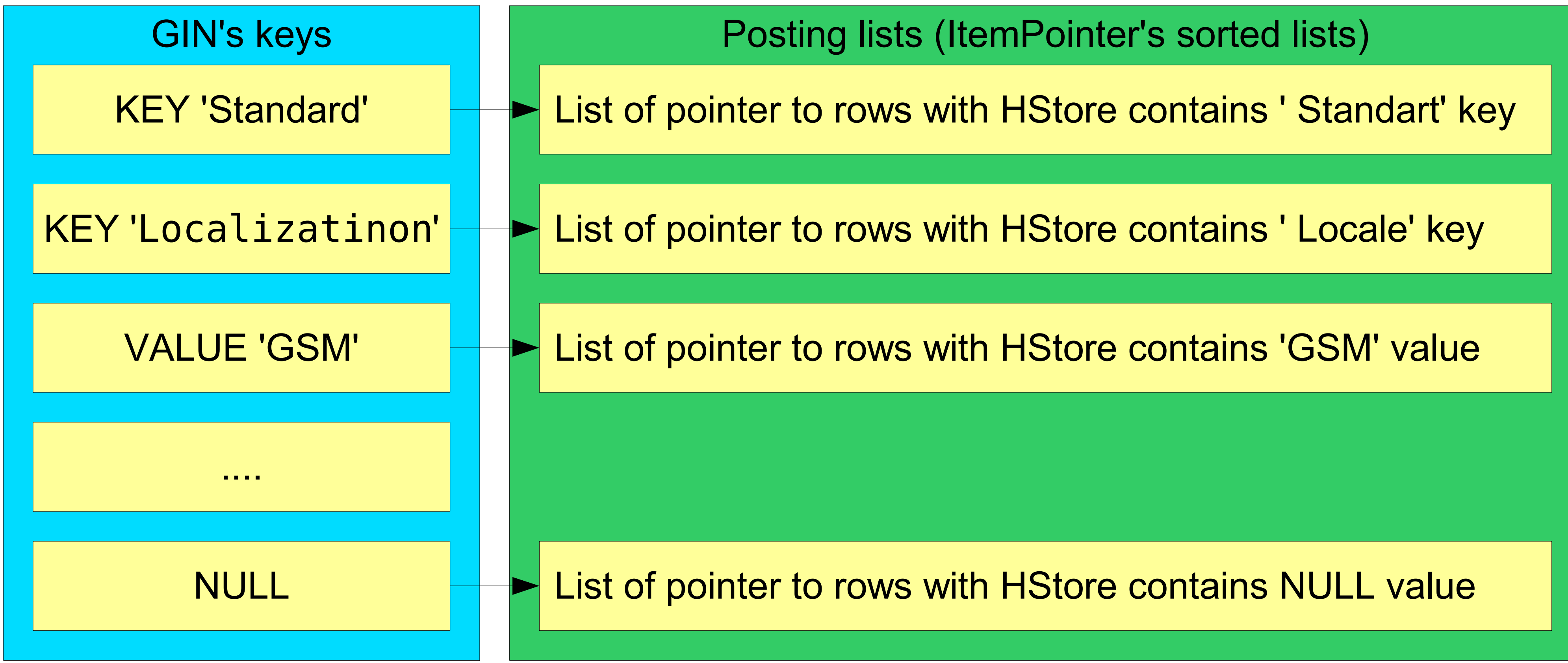
## Possible index structure




Looks very fast for searching... But GIN can't support this scheme

# GIN in practice: HStore

## GIN logical structure for HStore



# GIN in practice: HStore

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:

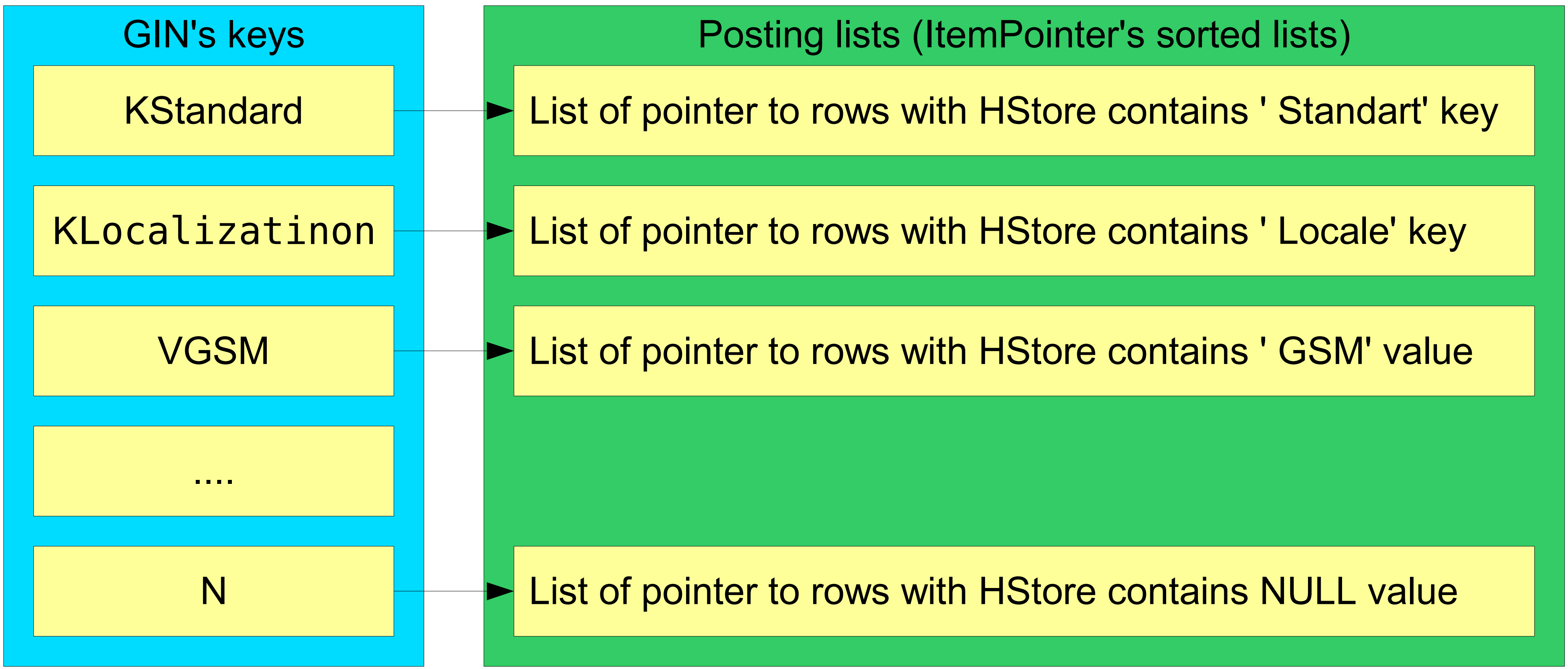
```
'"Standard"=>"GSM",
 "Localization"=>"RU",
 "Color"=>NULL'
```

GIN's keys:

```
KStandard, KLocale, KColor,
VGSM, VRU, N
```

# GIN in practice: HStore

## GIN logical structure for HStore



# GIN 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);
```

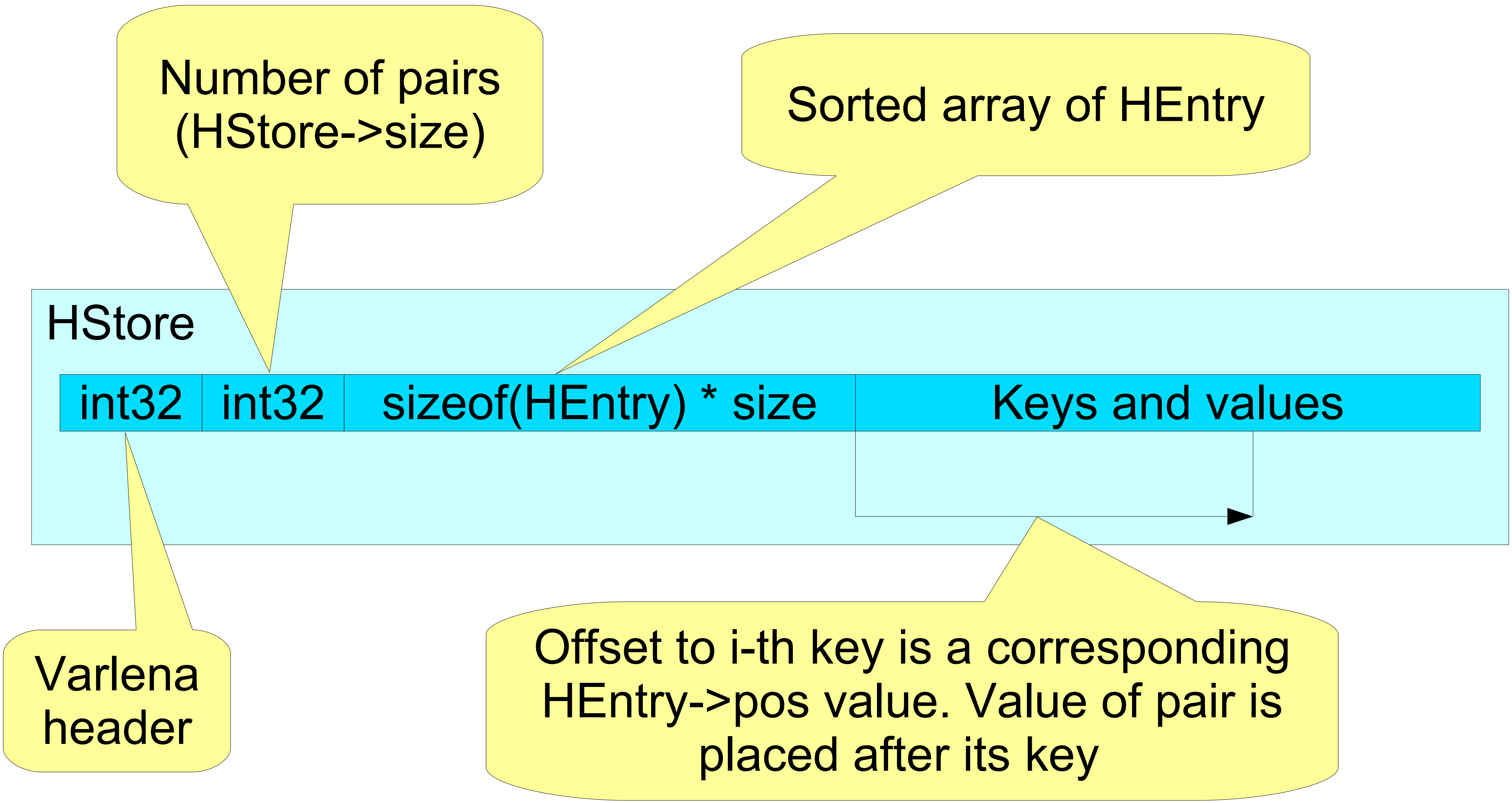
# GIN in practice: HStore

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

Access to key's and value's value:

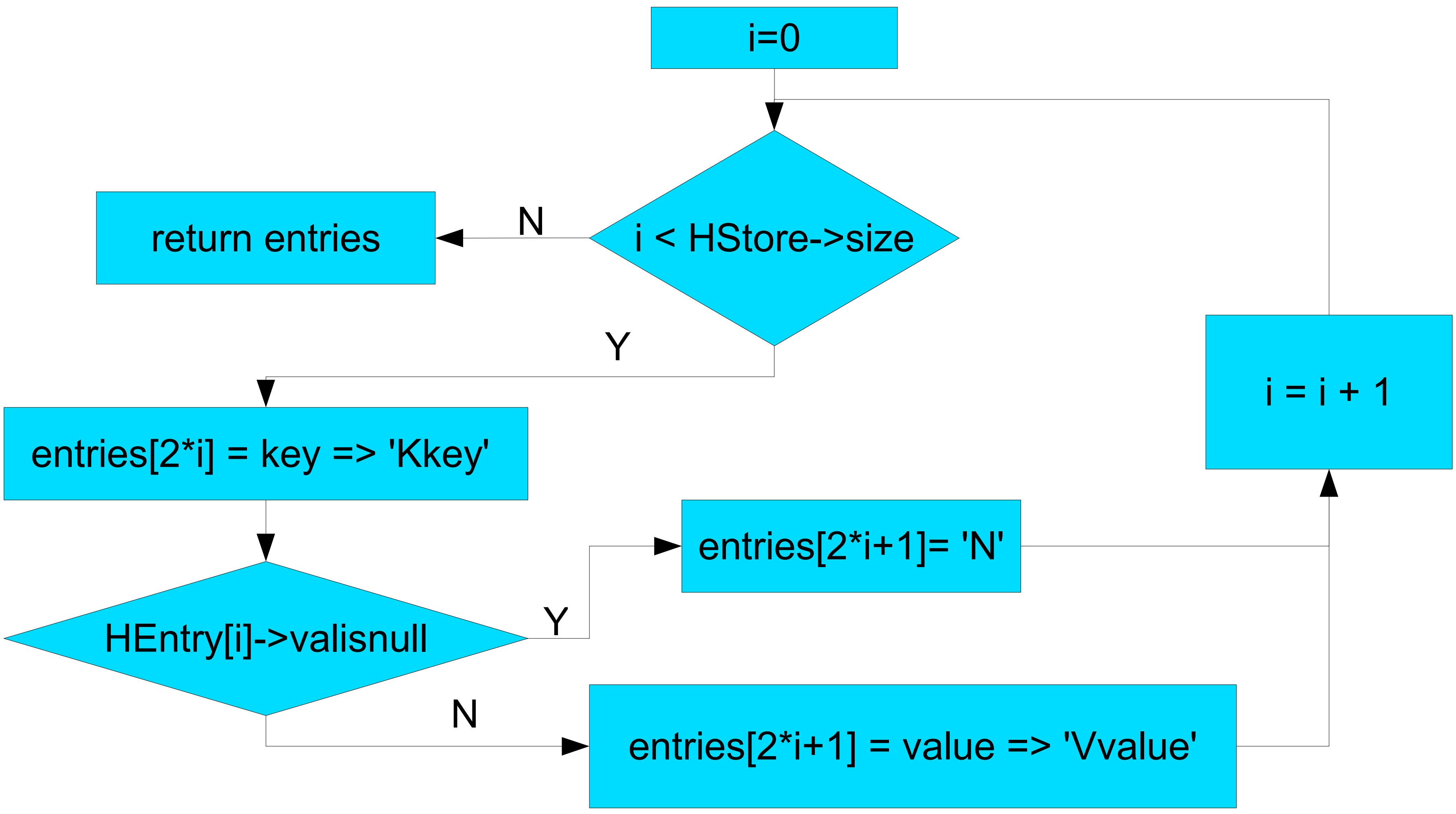
```
HEntry *ith_entry = ARRPTR(hstore) + i;
char *ith_key = STRPTR(hstore) + ith_entry->pos;
char *ith_value = ith_key + ith_entry->keylen;
```

# GIN in practice: HStore



# GIN in practice: HStore & GIN

## extractValue method

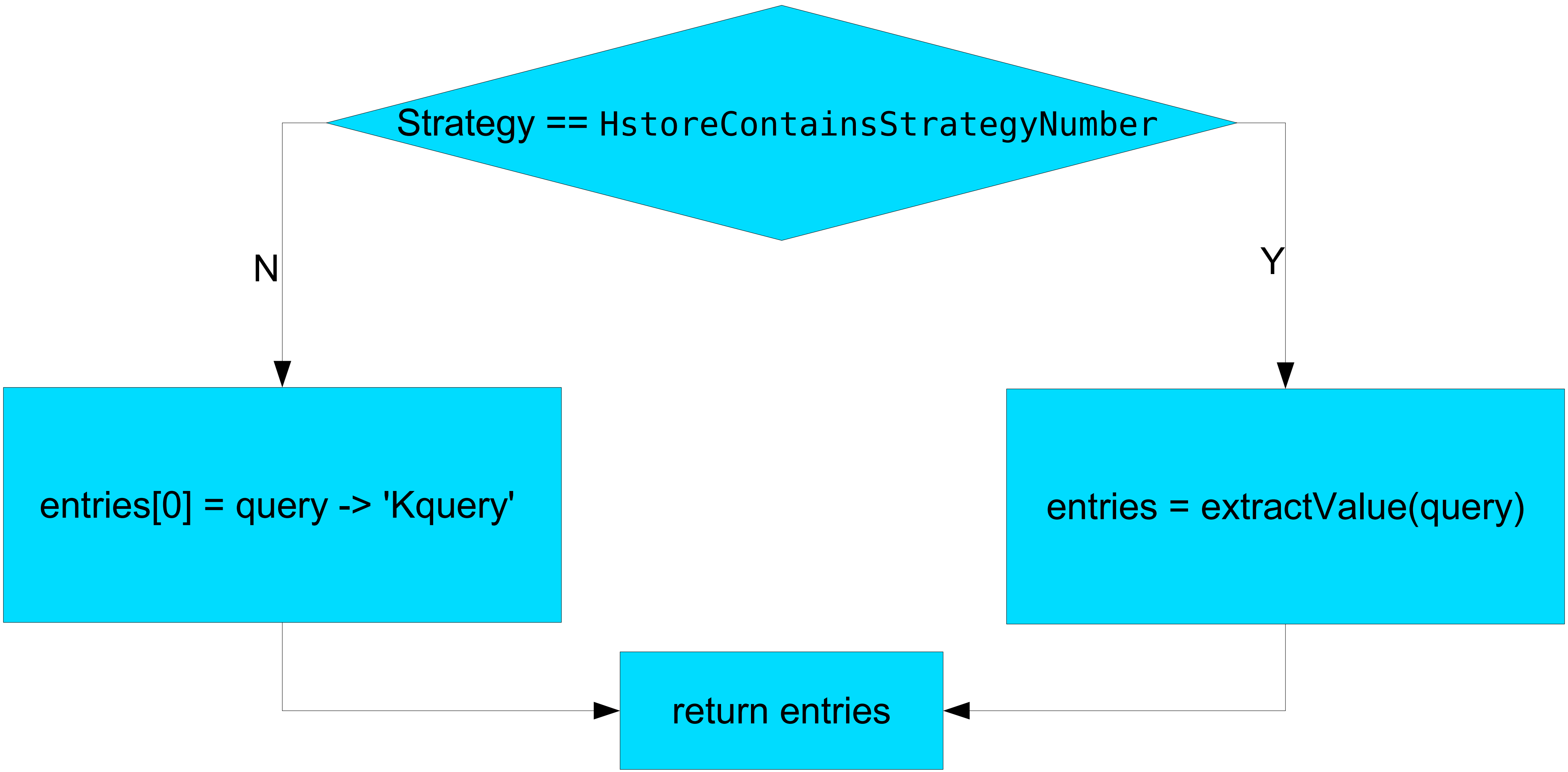


# GIN in practice: HStore & GIN

```
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



# GIN in practice: HStore & GIN

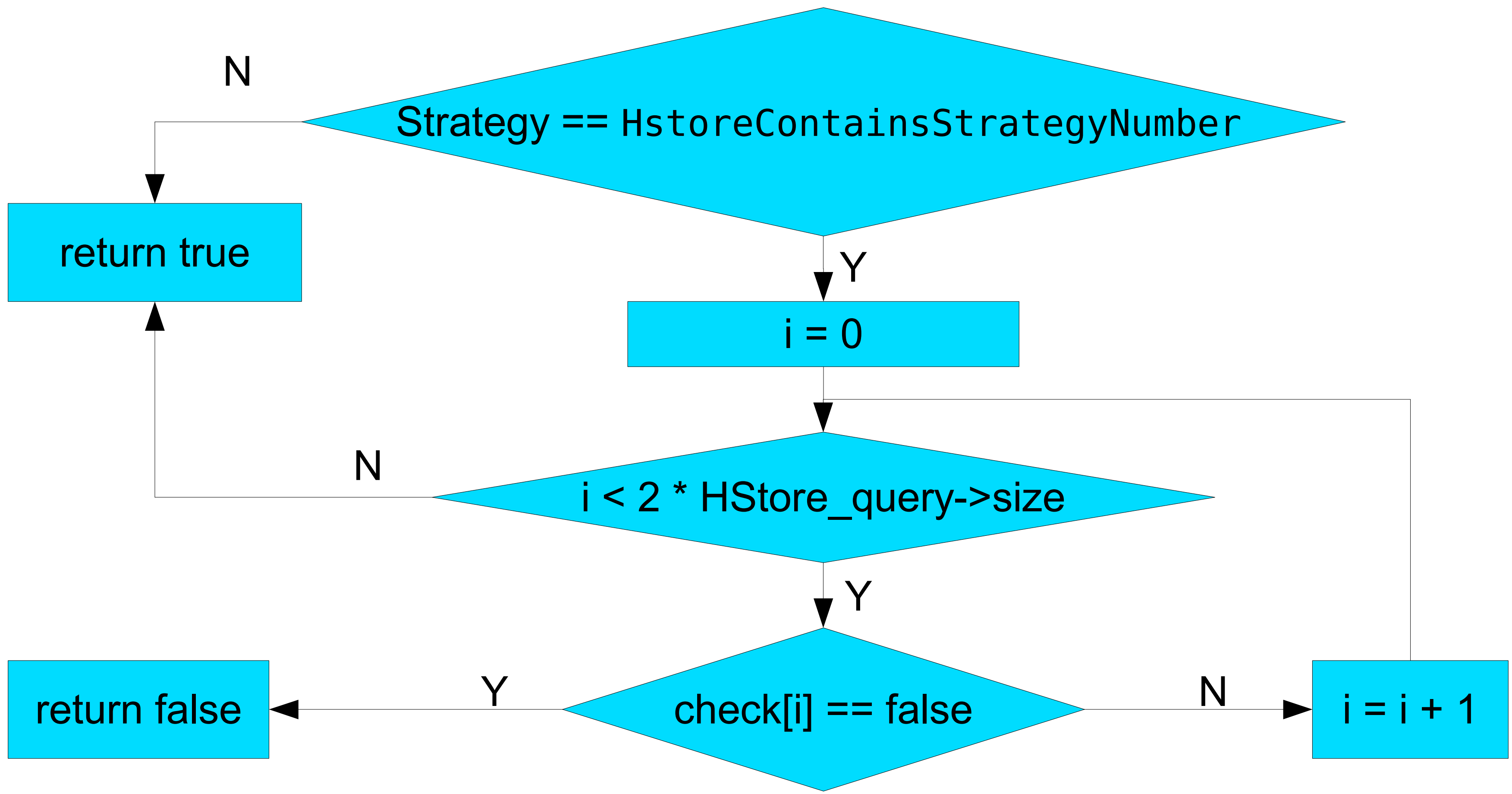
```
Datum gin_extract_hstore_query(PG_FUNCTION_ARGS) {
   StrategyNumber strategy = PG_GETARG_UINT16(2);

   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;
}
```

# GIN in practice: HStore & GIN

compareEntry method is built-in
bttextcmp() used for B-Tree over text

# GIN in practice: HStore & GIN

## Consistent method



# GIN in practice: HStore & GIN

```
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.

# GIN in practice: HStore & GIN

```
--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