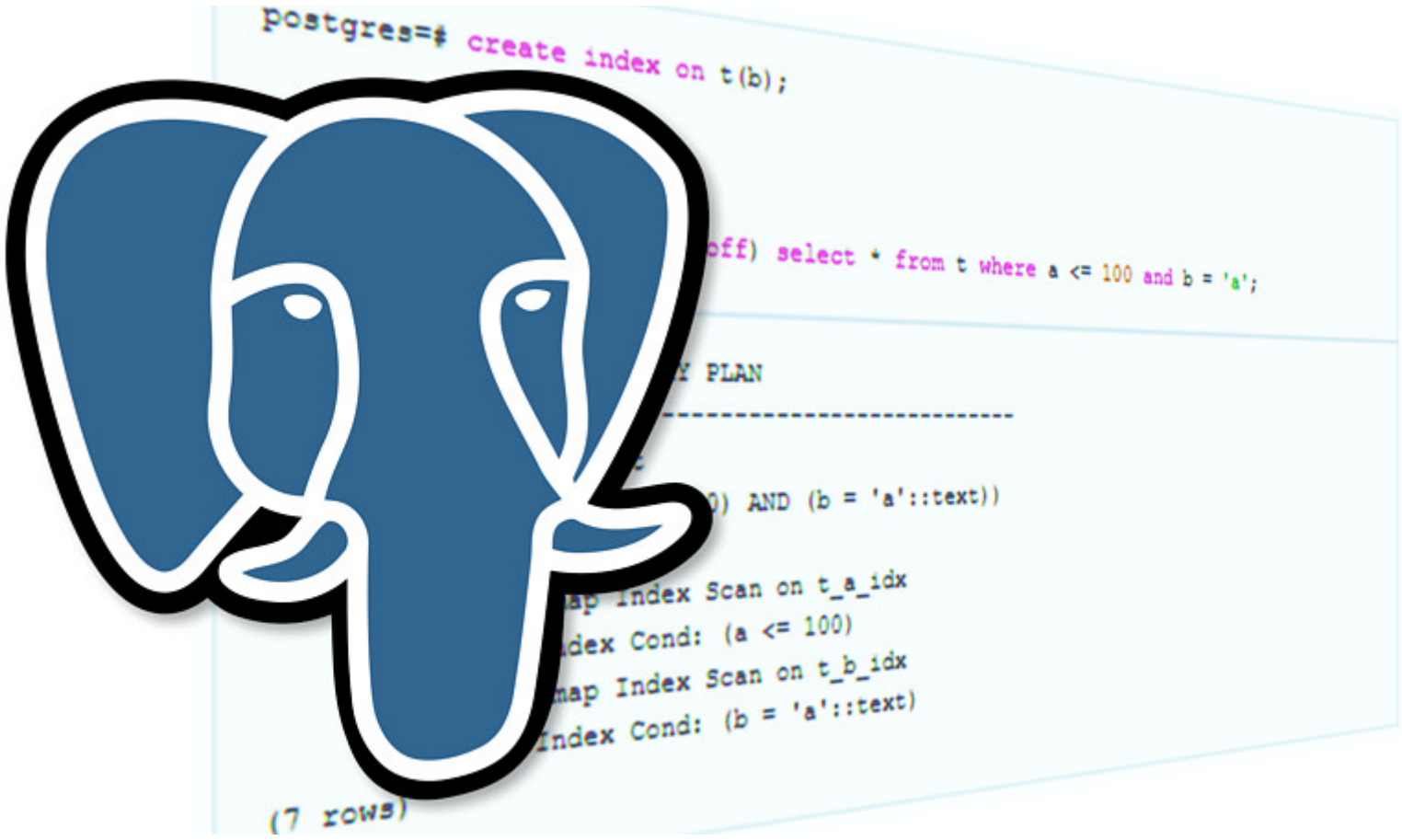
August 6, 2020 · PostgreSQL · By Egor Rogov

Indexes in PostgreSQL – 1



Habr Article

Introduction

This series of articles is largely concerned with indexes in PostgreSQL.

Any subject can be considered from different perspectives. We will discuss matters that should interest an application developer who uses DBMS: what indexes are available, why there are so many different types of them, and how to use them to speed up queries. The topic can probably be covered in fewer words, but in secrecy we hope for a curious developer, who is also interested in details of the internals, especially since understanding of such details allows you to not only defer to other's judgement, but also make conclusions of your own.

Development of new types of indexes is outside the scope. This requires knowledge of the C programming language and pertains to the expertise of a system programmer rather than an application developer. For the same reason we almost won't discuss programming interfaces, but will focus only on what matters for working with ready-to-use indexes.

In this article we will discuss the distribution of responsibilities between the **general indexing engine** related to the DBMS core and individual index access methods, which PostgreSQL enables us to add as extensions. In the next article we will discuss the interface of the access method and critical concepts such as classes and operator families. After that long but necessary introduction we will consider details of the structure and application of different types of indexes: Hash, B-tree, GiST, SP-GiST, GIN and RUM, BRIN, and Bloom.

> Before we start, I would like to thank Elena Indrupskaya for translating the articles to English. Things have changed a bit since the original publication in 2017 on habr.com. My comments on the current state of affairs are indicated like this.

Indexes

In PostgreSQL, indexes are special database objects mainly designed to speed up data access. They are auxiliary structures: each index can be deleted and recreated back from the information in the table. You may sometimes happen to hear that a DBMS can work without indexes although slowly. However, this is not the case, since indexes also serve to enforce some integrity constraints.

At present, six different kinds of indexes are built into PostgreSQL 9.6, and one more index is available as an extension – thanks to significant changes in version 9.6. So expect new types of indexes in a near future.

Despite all differences between types of indexes (also called access methods), each of them eventually associates a key (for example, the value of the indexed column) with table rows that contain this key. Each row is identified by TID (tuple id), which consists of the number of block in the file and the position of the row inside the block. That said, with the known key or some information about it we can quickly read those rows that may contain the information of our interest without scanning the entire table.

It is important to understand that an index speeds up data access at a certain maintenance cost. For each operation on indexed data, whether it be insertion, deletion, or update of table rows, indexes for that table need to be updated too, and in the same transaction. Note that update of table fields for which indexes haven't been built does not result in index update; this technique is called HOT (Heap-Only Tuples).

Extensibility entails some implications. To enable easy addition of a new access method to the system, an interface of the general indexing engine has been implemented. Its main task is to get TIDs from the access method and to work with them:

- Read data from corresponding versions of table rows.
- Fetch row versions TID by TID or in a batch using a prebuilt bitmap.
- Check visibility of row versions for the current transaction taking into account its isolation level.

Indexing engine is involved in performing queries. It is called according to a plan created at the optimization stage. The optimizer, sorting out and evaluating different ways to perform the query, should understand capabilities of all access methods that are potentially applicable. Will the method be able to return data in the needed order or should we anticipate sorting? Can we use this method to search for NULL? These are problems that the optimizer is regularly solving.

Not only the optimizer needs information about the access method. When building an index, the system must decide whether the index can be built on several columns and whether this index ensures uniqueness.

So, each access method should provide all the necessary information about itself. Versions lower than 9.6 used the "pg_am" table for this, while starting with version 9.6 the data moved to deeper levels, inside special functions. We will get acquainted with this interface a bit further.

All the rest is the task of the access method:

- Implement an algorithm for building the index and map the data into pages (for the buffer cache manager to uniformly process each index).
- Search information in the index by a predicate in the form "*indexed-field operator expression*".
- Evaluate the index usage cost.
- Manipulate the locks required for correct parallel processing.
- Generate write-ahead log (WAL) records.

We will first consider capabilities of the general indexing engine and then proceed to considering different access methods.

Indexing engine

Indexing engine enables PostgreSQL to work with various access methods uniformly, but taking into account their features.

Main scanning techniques

Index scan

We can work differently with TIDs provided by an index. Let's consider an example:

```
postgres=# create table t(a integer, b text, c boolean);

postgres=# insert into t(a,b,c)
  select s.id, chr((32+random()*94)::integer), random() < 0.01
  from generate_series(1,100000) as s(id)
  order by random();

postgres=# create index on t(a);

postgres=# analyze t;
```

We created a three-field table. The first field contains numbers from 1 to 100.000, and an index (no matter what type) is created on this field. The second field contains various ASCII characters except non-printable ones. Finally, the third field contains a logical value that is true for about 1% of rows and false for the rest. Rows are inserted into the table in a random order.

Let's try to select a value by the condition "a = 1". Note that the condition looks like "*indexed-field operator expression*", where *operator* is "equal" and *expression* (search key) is "1". In most cases, the condition must look like this for the index to be used.

```
postgres=# explain (costs off) select * from t where a = 1;
```

```
          QUERY PLAN
------------------------------
 Index Scan using t_a_idx on t
   Index Cond: (a = 1)
(2 rows)
```

In this case, the optimizer decided to use *index scan*. With index scanning, the access method returns TID values one by one until the last matching row is reached. The indexing engine accesses the table rows indicated by TIDs in turn, gets the row version, checks its visibility against multiversion concurrency rules, and returns the data obtained.

Bitmap scan

Index scan works fine when we deal with only a few values. However, as the number of retrieved rows increases, it is more likely to get back to the same table page several times. Therefore, the optimizer switches to the *bitmap scan*.

```
postgres=# explain (costs off) select * from t where a <= 100;
```

```
             QUERY PLAN
-----------------------------------
 Bitmap Heap Scan on t
   Recheck Cond: (a <= 100)
   ->  Bitmap Index Scan on t_a_idx
         Index Cond: (a <= 100)
(4 rows)
```

The access method first returns all TIDs that match the condition (Bitmap Index Scan node), and the bitmap of row versions is built from these TIDs. Row versions are then read from the table (Bitmap Heap Scan), each page being read only once.

Note that in the second step, the condition may be rechecked (Recheck Cond). The number of retrieved rows can be too large for the bitmap of row versions to fully fit the RAM (limited by the "work_mem" parameter). In this case, the bitmap is only built for pages that contain at least one matching row version. This "lossy" bitmap requires less space, but when reading a page, we need to recheck the conditions for each row contained there. Note that even for a small number of retrieved rows and therefore "exact" bitmap (such as in our example), the "Recheck Cond" step is represented in the plan anyway, although not actually performed.

If conditions are imposed on several table fields and these fields are indexed, the bitmap scan enables use of several indexes simultaneously (if the optimizer considers this efficient). For each index, bitmaps of row versions are built, for which bitwise boolean multiplication (if the expressions are joined by AND) or boolean addition (if the expressions are joined by OR) is then performed. For example:

```
postgres=# create index on t(b);

postgres=# analyze t;

postgres=# explain (costs off) select * from t where a <= 100 and b = 'a';
```

```
                     QUERY PLAN
-----------------------------------------------------
 Bitmap Heap Scan on t
   Recheck Cond: ((a <= 100) AND (b = 'a'::text))
   ->  BitmapAnd
         ->  Bitmap Index Scan on t_a_idx
               Index Cond: (a <= 100)
         ->  Bitmap Index Scan on t_b_idx
               Index Cond: (b = 'a'::text)
(7 rows)
```

Here the BitmapAnd node joins two bitmaps by the bitwise "and" operation.

Bitmap scan enables us to avoid repeated accesses to the same data page. But what if the data in table pages is physically ordered exactly the same way as index records? It's beyond doubt that we cannot fully rely on the physical order of data in pages. If sorted data is needed, we must explicitly specify the ORDER BY clause in the query. But situations are likely where actually "almost all" data is ordered: for example, if rows are added in the needed order and do not change after that or after performing the CLUSTER command. In cases like this, building a bitmap is an excessive step, and a regular index scan will be just as good (unless we take into account a possibility to join several indexes). Therefore, when choosing an access method, the planner looks into a special statistic that shows the correlation between physical row ordering and logical ordering of column values:

```
postgres=# select attname, correlation from pg_stats where tablename = 't';
```

```
 attname | correlation
---------+-------------
 b       |    0.533512
 c       |    0.942365
 a       | -0.00768816
(3 rows)
```

Absolute values close to one indicate a high correlation (as for column "c"), while values close to zero, on the contrary, indicate a chaotic distribution (column "a").

Sequential scan

To complete the picture, we should note that with a non-selective condition, the optimizer will be right to prefer sequential scan of the entire table to the use of index:

```
postgres=# explain (costs off) select * from t where a <= 40000;
```

```
       QUERY PLAN
------------------------
 Seq Scan on t
   Filter: (a <= 40000)
(2 rows)
```

The thing is that indexes work the better the higher the condition selectivity, that is, the fewer rows match it. Growth of the number of retrieved rows increases overhead costs of reading index pages.

Sequential scans being faster that random scans compounds the situation. This especially holds for hard disks, where the mechanical operation of bringing a magnetic head to a track takes significantly more time than data reading itself. This effect is less noticeable for SSD. Two parameters are available to take account of the differences in the access costs, "seq_page_cost" and "random_page_cost", which we can set not only globally, but at the level of tablespaces, this way adjusting to the characteristics of different disk subsystems.

Covering indexes

As a rule, the main task of an access method is to return the identifiers of matching table rows for the indexing engine to read necessary data from these rows. But what if the index already contains all the data needed for the query? Such an index is called *covering*, and in this case, the optimizer can apply the *index-only scan*:

```
postgres=# vacuum t;

postgres=# explain (costs off) select a from t where a < 100;
```

```
             QUERY PLAN
-----------------------------------
 Index Only Scan using t_a_idx on t
   Index Cond: (a < 100)
(2 rows)
```

This name may give an idea that the indexing engine does not access the table at all and gets all the necessary information from the access method alone. But this is not exactly the case since indexes in PostgreSQL do not store information that enables us to judge the row visibility. Therefore, an access method returns versions of rows that match the search condition regardless of their visibility in the current transaction.

However, if the indexing engine needed to look into the table for visibility every time, this scanning method would not have been any different from a regular index scan.

To solve the problem, for tables PostgreSQL maintains a so-called *visibility map* in which vacuuming marks the pages where data was not changed long enough for this data to be visible by all transactions regardless of the start time and isolation level. If the identifier of a row returned by the index relates to such a page, visibility check can be avoided.

Therefore, regular vacuuming increases efficiency of covering indexes. Moreover, the optimizer takes into account the number of dead tuples and can decide not to use the index-only scan if it predicts high overhead costs for the visibility check.

We can learn the number of forced accesses to a table using the EXPLAIN ANALYZE command:

```
postgres=# explain (analyze, costs off) select a from t where a < 100;
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan using t_a_idx on t (actual time=0.025..0.036 rows=99 loops=1)
   Index Cond: (a < 100)
   Heap Fetches: 0
 Planning time: 0.092 ms
 Execution time: 0.059 ms
(5 rows)
```

In this case, it was not needed to access the table (Heap Fetches: 0), since vacuuming has just been done. In general, the closer this number to zero the better.

Not all indexes store indexed values along with row identifiers. If the access method cannot return the data, it cannot be used for index-only scans.

PostgreSQL 11 has introduced a new feature: INCLUDE-indexes. What if there is a unique index which lacks some columns to be used as a covering index for some query? You cannot simply add the columns to the index as it will break its uniqueness. The feature allows to include *non-key* columns which don't affect uniqueness and cannot be used in search predicates, but still can serve index-only scans. The patch was developed by my colleague Anastasia Lubennikova.

NULL

NULLs play an important role in relational databases as a convenient way to represent a nonexistent or unknown value.

But a special value is special to deal with. A regular boolean algebra becomes ternary; it's unclear whether NULL should be smaller or larger than regular values (this requires special constructs for sorting, NULLS FIRST and NULLS LAST); it's not evident whether aggregate functions should consider NULLs or not; a special statistic is needed for the planner...

From the perspective of the index support, it's also unclear whether we need to index these values or not. If NULLs are not indexed, the index may be more compact. But if NULLs are indexed, we will be able to use the index for conditions like "*indexed-field* IS [NOT] NULL" and also as a covering index when no conditions at all are specified for the table (since in this case, the index must return the data of all table rows, including those with NULLs).

For each access method, the developers make an individual decision whether to index NULLs or not. But as a rule, they do get indexed.

Indexes on several fields

To support conditions for several fields, *multicolumn indexes* can be used. For example, we might build an index on two fields of our table:

```
postgres=# create index on t(a,b);

postgres=# analyze t;
```

The optimizer will most likely prefer this index to joining bitmaps since here we readily get the needed TIDs without any auxiliary operations:

```
postgres=# explain (costs off) select * from t where a <= 100 and b = 'a';
```

```
                    QUERY PLAN
-----------------------------------------------
 Index Scan using t_a_b_idx on t
   Index Cond: ((a <= 100) AND (b = 'a'::text))
(2 rows)
```

A multicolumn index can also be used to speed up data retrieval by a condition for some of the fields, starting with the first one:

```
postgres=# explain (costs off) select * from t where a <= 100;
```

```
              QUERY PLAN
--------------------------------------
 Bitmap Heap Scan on t
   Recheck Cond: (a <= 100)
   ->  Bitmap Index Scan on t_a_b_idx
         Index Cond: (a <= 100)
(4 rows)
```

In general, if the condition is not imposed on the first field, the index will not be used. But sometimes the optimizer may regard use of the index as more efficient than the sequential scan. We will expand on this topic when considering "btree" indexes.

Not all access methods support building indexes on several columns.

Indexes on expressions

We've already mentioned that the search condition must look like "*indexed-field operator expression*". In the example below, index will not be used since an expression containing the field name is used instead of the field name itself:

```
postgres=# explain (costs off) select * from t where lower(b) = 'a';
```

```
                QUERY PLAN
-----------------------------------------
 Seq Scan on t
   Filter: (lower((b)::text) = 'a'::text)
(2 rows)
```

It does not take much to rewrite this specific query so that only the field name is written to the left of the operator. But if this is not possible, indexes on expressions (functional indexes) will help:

```
postgres=# create index on t(lower(b));

postgres=# analyze t;

postgres=# explain (costs off) select * from t where lower(b) = 'a';
```

```
                     QUERY PLAN
-----------------------------------------------------
 Bitmap Heap Scan on t
   Recheck Cond: (lower((b)::text) = 'a'::text)
   ->  Bitmap Index Scan on t_lower_idx
         Index Cond: (lower((b)::text) = 'a'::text)
(4 rows)
```

The functional index is built not on a table field, but on an arbitrary expression. The optimizer will consider this index for conditions like "*indexed-expression operator expression*". If computation of the expression to be indexed is a costly operation, update of the index will also require considerable computation resources.

Please also keep in mind that an individual statistic is gathered for the indexed expression. We can get to know this statistic in the "pg_stats" view by the index name:

```
postgres=# \d t
```

```
       Table "public.t"
 Column |  Type   | Modifiers
--------+---------+-----------
 a      | integer |
 b      | text    |
 c      | boolean |
Indexes:
    "t_a_b_idx" btree (a, b)
    "t_a_idx" btree (a)
    "t_b_idx" btree (b)
    "t_lower_idx" btree (lower(b))
```

```
postgres=# select * from pg_stats where tablename = 't_lower_idx';
```

It is possible, if necessary, to control the number of histogram baskets the same way as for regular data fields (noting that the column name can differ depending on the indexed expression):

```
postgres=# \d t_lower_idx
```

```
 Index "public.t_lower_idx"
 Column | Type | Definition
--------+------+------------
 lower  | text | lower(b)
btree, for table "public.t"
```

```
postgres=# alter index t_lower_idx alter column "lower" set statistics 69;
```

PostgreSQL 11 has introduces a cleaner way to control statistics target for indexes by specifying the column *number* in ALTER INDEX ... SET STATISTICS command. The patch was developed by my colleague Alexander Korotkov, and Adrien Nayrat.

Partial indexes

Sometimes a need arises to index only part of table rows. This is usually related to a highly nonuniform distribution: it makes sense to search for an infrequent value by an index, but it is easier to find a frequent value by full scan of the table.

We can certainly build a regular index on the "c" column, which will work the way we expect:

```
postgres=# create index on t(c);

postgres=# analyze t;

postgres=# explain (costs off) select * from t where c;
```

```
          QUERY PLAN
------------------------------
 Index Scan using t_c_idx on t
   Index Cond: (c = true)
   Filter: c
(3 rows)
```

```
postgres=# explain (costs off) select * from t where not c;
```

```
    QUERY PLAN
------------------
 Seq Scan on t
   Filter: (NOT c)
(2 rows)
```

And the index size is 276 pages:

```
postgres=# select relpages from pg_class where relname='t_c_idx';
```

```
 relpages
----------
      276
(1 row)
```

But since the "c" column has the value of true only for 1% of rows, 99% of the index is actually never used. In this case, we can build a partial index:

```
postgres=# create index on t(c) where c;

postgres=# analyze t;
```

The size of the index is reduced to 5 pages:

```
postgres=# select relpages from pg_class where relname='t_c_idx1';
```

```
 relpages
----------
        5
(1 row)
```

Sometimes the difference in the size and performance may be pretty significant.

Sorting

If an access method returns row identifiers in some particular order, this provides the optimizer with additional options to perform the query.

We can scan the table and then sort the data:

```
postgres=# set enable_indexscan=off;

postgres=# explain (costs off) select * from t order by a;
```

```
     QUERY PLAN
---------------------
 Sort
   Sort Key: a
   ->  Seq Scan on t
(3 rows)
```

But we can read the data using the index readily in a desired order:

```
postgres=# set enable_indexscan=on;

postgres=# explain (costs off) select * from t order by a;
```

```
          QUERY PLAN
-----------------------------
 Index Scan using t_a_idx on t
(1 row)
```

Only "btree" out of all access methods can return sorted data, so let's put off a more detailed discussion until considering this type of index.

Concurrent building

Usually building of an index acquires a SHARE lock for the table. This lock permits reading data from the table, but forbids any changes while index is being built.

We can make sure of this if, say, during building of an index on the table "t", we perform the query below in another session:

```
postgres=# select mode, granted from pg_locks where relation = 't'::regclass;
```

```
   mode    | granted
-----------+---------
 ShareLock | t
(1 row)
```

If the table is large enough and extensively used for insertion, update, or deletion, this may appear to be inadmissible since modifying processes will wait for a lock release for a long time.

In this case, we can use concurrent building of an index.

```
postgres=# create index concurrently on t(a);
```

This command locks the table in the SHARE UPDATE EXCLUSIVE mode, which permits both reading and update (only changing the table structure is forbidden, as well as concurrent vacuuming, analysis, or building another index on this table).

However, there is also a flip side. First, the index will be built more slowly than usual since two passes across the table are done instead of one, and it is also necessary to be waiting for completion of parallel transactions that modify the data.

Second, with concurrent building of the index, a deadlock can occur or unique constraints can be violated. However, the index will be built, although nonoperating. Such an index must be deleted and rebuilt. Nonoperating indexes are marked with the INVALID word in the output of the psql \d command, and the query below returns a complete list of those:

```
postgres=# select indexrelid::regclass index_name, indrelid::regclass table_name
from pg_index where not indisvalid;
```

```
 index_name | table_name
------------+------------
 t_a_idx    | t
(1 row)
```

Next article

Egor Rogov

← Back to all articles

Egor Rogov

Products

Postgres Pro Enterprise
Postgres Pro Standard
Cloud Solutions
Postgres Extensions

Services

24×7×365 Technical Support
Migration to Postgres
High Availability Deployment
Database Audit
Remote DBA for PostgreSQL

About

Leadership team
Partners
Customers
In the News
Press Releases
Press Info

Resources

Blog
Documentation
Webinars
Videos
Presentations

Community

Events
Training Courses
Intro Book
Demo Database
Mailing List Archives

Contacts

Neptune House, Marina Bay, office 207, Gibraltar, GX11 1AA
info@postgrespro.com

Get in touch!

Your First and Last Name

Company

E-mail

Message

I confirm that I have read and accepted PostgresPro's Privacy Policy.

I agree to get Postgres Pro discount offers and other marketing communications.

Send a message

EULA
EULA for Cloud Environments
Privacy Policy