pganalyze

LUKAS FITTL
FOUNDER AT PGANALYZE

Effective Indexing in Postgres

Creating the Best Index for your Queries

Effective Indexing in Postgres: Creating the Best Index for your Queries

Creating indexes effectively requires reviewing your query workload in detail, understanding the involved data types and operators, and then creating one or multiple indexes of the right index type, on the right index columns.

Whilst this eBook is primarily focused on Postgres, you will find that some of the concepts are universal and can be found in other relational databases as well.

In this ebook, we'll discuss how to create the best indexes for your queries. Indexing can make a significant difference for the performance of your application's query workload. Too many indexes will slow down your write performance. But, creating the right indexes can often improve your query performance by 10x or even 100x.

ABOUT

How Postgres uses indexes

When thinking through indexing, it helps to remember that indexes are redundant data structures. That is, in the most simplest definition, they are an alternate way of finding specific rows of data stored in a Postgres table by use of a specialized, redundant data structure. Besides helping find data, there are some other use cases for indexes, such as enforcing uniqueness in a table - for the context of this eBook we are solely focused on the performance gains through adding the right index(es).

Creating an index in Postgres is as simple as running the `CREATE INDEX` command for a particular table and a set of columns:

SQL

```
1   CREATE INDEX ON mytable (column1, column2);
```

You will see that we do not specify an index type here, which, by default, makes Postgres choose the B-tree index data structure. There are alternate index types besides B-tree indexes that we'll get to in a bit. Some indexes are for improving performance of looking up all rows matching a specific value for

a single column, others can be more complicated, containing multiple columns, making use of expressions, or restricting the indexed rows to a subset of the table data (called a "**partial index**").

Before we dive into all of that, it's important to note the downside of adding a lot of indexes on a specific table. Even if we would never actually use a particular index, the fact that it exists means it has to be kept up to date with each write to a table. This means any inserts, updates and deletes will also update the index, and cause a record to be written to the Postgres WAL. In short: **Adding too many indexes will slow down your write performance**.

Now, to talk more about the primary motivation for indexes: **Improving query performance**.

Let's take a look at a simple example of a query for the first column we've created the index for:

SQL

```sql
SELECT * FROM mytable WHERE column1 = 123;
```

We can refer to "column1 = 123" as a **predicate**. You can have multiple predicates as part of your WHERE clause, combined using AND or

OR keywords. Based on these predicates, Postgres will determine how to scan the referenced table ("mytable").

During the planning process, Postgres will go through the applicable indexes for the particular query, and based on statistics about the table, as well as information such as the size of the indexes, determine the cost of executing a scan using each usable index.

When you review the resulting query plan using EXPLAIN, you will typically see one of four main types of scanning a table in Postgres:

- **Sequential Scan:** Uses the source table to find data, by going through it row-by-row without using any index for reducing which rows are scanned (this type of scan is always available for a table)
- **Index Scan:** Determines which table rows match based on an index, and then retrieves the rows from the table, potentially filtering some of the data if not all queried columns are included in the index
- **Index Only Scan:** Retrieves the actual query result directly from the index, and avoids accessing the table itself for the requested data (except when verifying visibility information for recently changed rows)
- **Bitmap Index Scan + Bitmap Heap Scan:** Uses an index to generate a bitmap of which parts

of the table likely contain the matching rows, and then access the actual table to get these rows using the bitmap - this is particularly useful to combine different indexes (each resulting in a bitmap) to implement AND and OR conditions in a query

Commonly, when optimizing query performance using indexes, you are either trying to change a Sequential Scan to one of the index-based scan types, or you are optimizing how indexes get used. For example, whilst you can use multiple indexes to satisfy a query filtering multiple columns (using a Bitmap Index/Heap Scan), it is often more performant to implement a multi-column index. You may also try to consolidate indexes and check the scans they are used in to reduce write overhead and improve the chance an index is in the cache instead of on disk.

To get a good understanding of how to create indexes, we need to go one step further, and review a critical concept of how Postgres itself works: **data types and operators.**

Data types, operators and index types

Going back to our earlier example of a very simple query:

SQL

```sql
SELECT * FROM mytable WHERE column1 = 123;
```

We'll also need the schema definition to understand how Postgres works with the query:

SQL

```sql
CREATE TABLE mytable (int column1, jsonb column2);
```

When Postgres analyzes this query, it will determine the involved data types and operators. In this query, the data type for column1 is a 4-byte integer, commonly referred to by "int" or "int4". The operator is the equality operator ("=") that finds rows matching the exact value that was requested ("123").

The combination of the data type and the operator determine which index types are most effective. In this example, for most situations, a B-tree index

on column1 is the best fit. B-tree indexes support equality and range comparisons, and work with all commonly used data types. You can find out more on how B-tree indexes work later in this eBook.

Let's go back to our earlier index, which actually indexed both "column1" and "column2":

SQL

```
1   CREATE INDEX ON mytable (column1, column2);
```

And look at a more relevant query that also involves "column2":

SQL

```
1   SELECT * FROM mytable WHERE column1 = 123 AND column2 ? 'mykey';
```

This query retrieves all rows that have column1 set to "123" and where the JSON data in "column2" has a key named "mykey".

If you were to run EXPLAIN on this query, you would notice that it would include “column1” in the index condition, but not “column2”. That is because **the “?” operator for JSONB is not supported by B-tree indexes**. The root cause for this is that the B-tree data structure does not support indexing lookups on arbitrary keys for a JSONB value.

If instead we were to create a GIN index like this:

SQL

```
1   CREATE INDEX ON mytable USING gin (column2);
```

You would notice that an EXPLAIN plan now shows the index being used if you query for “column2” with the “?” operator.

Behind the scenes, Postgres uses a concept called “operator classes” to map a specific operator on a data type to the actual implementation. Operator classes are then grouped into “operator families" for similar data types / operators. Typically one does not interact with this logic, but it’s easy to see all the details in the Postgres catalog tables:

SQL

```sql
SELECT am.amname AS index_method,
       opf.opfname AS opfamily_name,
       amop.amopopr::regoperator AS opfamily_operator
  FROM pg_am am,
       pg_opfamily opf,
       pg_amop amop
 WHERE opf.opfmethod = am.oid AND amop.amopfamily = opf.oid
       AND amop.amopopr = '=(text,text)'::regoperator;
```

This will return all operator classes/families defined for the equality operator (“=”) involving the comparison of two text columns:

SQL

```
 index_method |  opfamily_name   | opfamily_operator
--------------+------------------+-------------------
 btree        | text_ops         | =(text,text)
 btree        | text_pattern_ops | =(text,text)
 hash         | text_ops         | =(text,text)
 hash         | text_pattern_ops | =(text,text)
 spgist       | text_ops         | =(text,text)
 brin         | text_minmax_ops  | =(text,text)
 gist         | gist_text_ops    | =(text,text)
(7 rows)
```

You can see that equality comparisons on text data types are not supported on the GIN index type by default. You can also see that for B-tree and Hash

indexes there are actually two different operator classes - "text_ops" (the default), and "text_pattern_ops". That second operator class can be useful when your database is using a non-C locale (e.g. "en_US.UTF-8"), and you are trying to index the "~~" operator (commonly known by its alias, "LIKE").

You can then specify the special operator classes during index creation, like this:

SQL

```sql
CREATE INDEX ON mytable (column3 text_pattern_ops);
```

For a full overview of which built-in data types work together with which index type and which operators, check the **Appendix** later in this eBook.

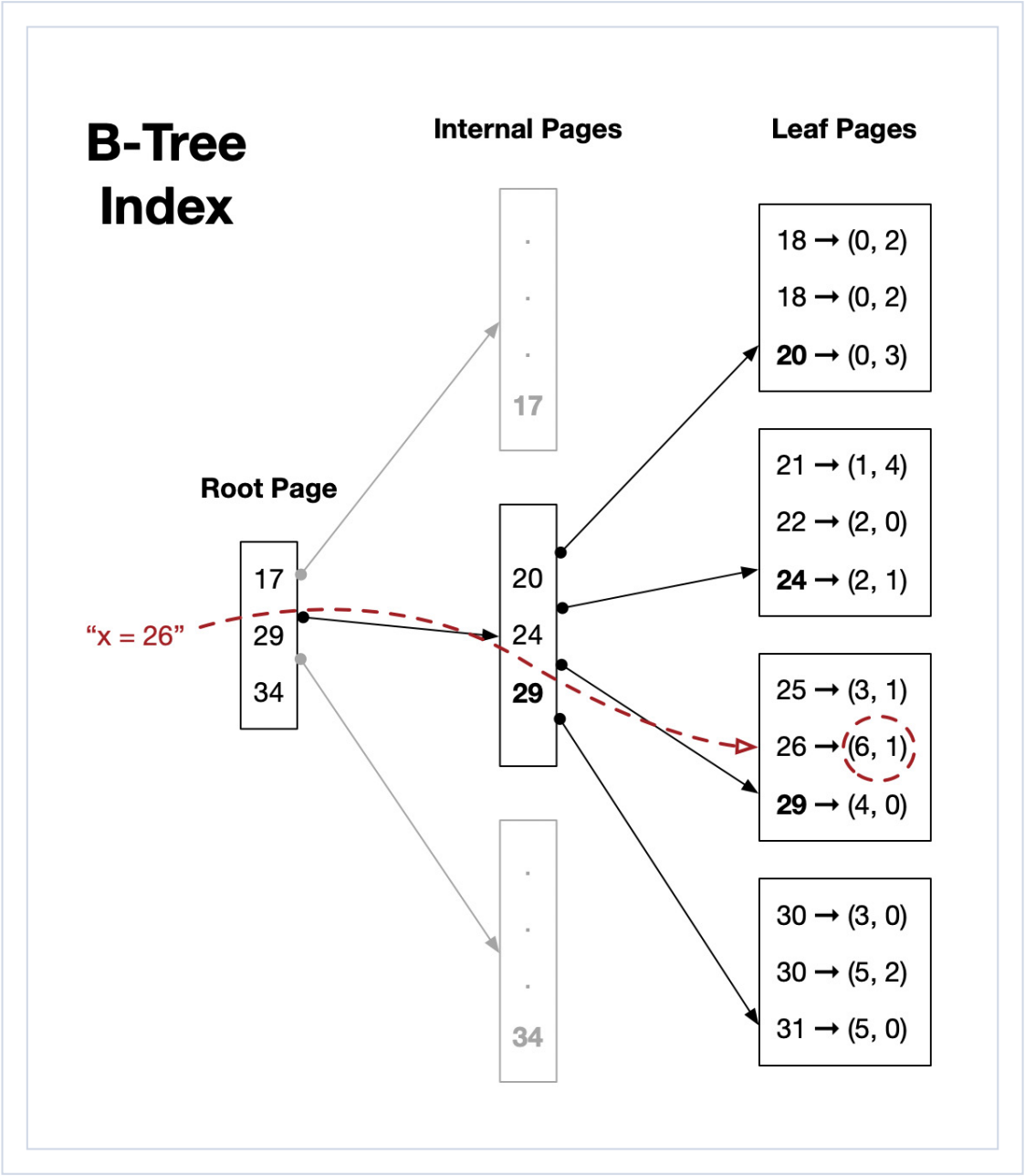
Now, let's get a better sense for which index types exist in Postgres, and how they are actually structured internally.

B-tree indexes

We'll start with the most commonly used Postgres index type: The B-tree index. You may remember B-tree indexes from a Computer Science class, or running across them with most database systems that

exist today. They are very effective for a majority of use cases, and you'll often find that your query workload only requires B-tree indexes for optimal performance.

The high-level structure of the B-tree Index index looks like this:



You can see that at its simplest, **a B-tree index is a balanced tree structure**, with each leaf page (shown on the right) having the exact same distance to the root page, typically going through one or more levels of so-called "internal" pages. The majority of the pages in an index are leaf pages - and they contain the actual pointers to where the associated rows for a particular key exist on the table. The **role of internal pages is to allow efficient traversal of the tree**, without having to scan a large amount of unnecessary leaf pages.

In the example of querying for all rows matching "x = 26", we first go through the root page to locate the page(s) possibly containing the value 26. You can see that there is an internal page for all values up to 29, and a different internal page for all values up to 17. We choose the internal page with values up to 29. That page then again references multiple leaf pages, but only one of the leaf pages (for values up to 29) contains values possibly being 26. We visit this page, and determine that there is one index entry for the value 26, that points to the table row location "(6, 1)". With that, we have found the location of the table row in a fixed number of steps, instead of having to scan over a whole table.

It's also important to note that Postgres B-tree indexes are more advanced than what you would

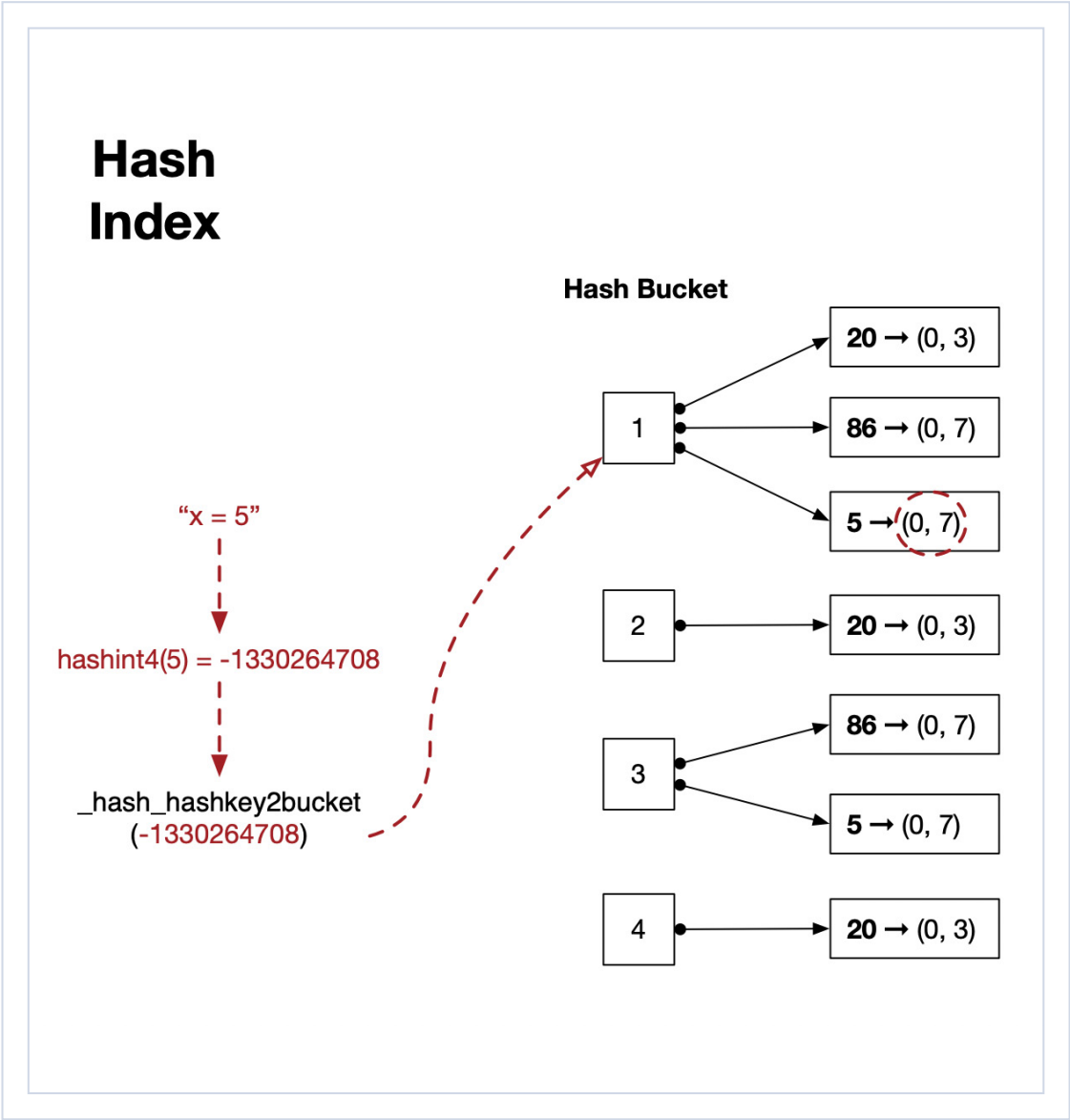
typically call a "B-tree" in computing literature. It would take multiple chapters to discuss all the details, but suffice to say that Postgres visibility rules complicate how indexes are actually structured bit-for-bit, and responding to table modifications requires particular behaviours. If you are interested in all the details, reading the B-tree README file in the Postgres source code is a good start.

From a query perspective, B-tree indexes implement the most commonly used operators: equality ("=") and range comparison ("<"', "<=", ">=", ">"), for all of the built-in Postgres data types. If you are using these operators, 99% of the time, B-tree indexes are the right choice.

B-tree indexes also have a few other useful attributes, for example they can produce sorted output for your queries (as the leaf pages are pre-sorted in the index), avoiding the need of doing an additional Sort step. Sorted output is especially useful when using LIMIT, since only the data up to the limit needs to be retrieved - if the output was unsorted all data would need to be retrieved and then passed through a Sort step before applying a Limit. This property of B-tree indexes can be very useful when implementing keyset-based pagination for an application.

Hash indexes

Hash indexes fit very similar use cases compared to B-tree indexes. They are solely focused on equality operators ("=") and do not support any other operators. Their structure looks like this:



Hash indexes build on the built-in Postgres hash functions, that can hash almost all Postgres data types to a 4-byte integer value. In the example shown, a 4-byte integer value itself is mapped to the 4-byte hash value - as you can see the hash value is clearly distinct from the input value. The hash index maps a set of hash values to a fixed number of hash buckets. Each hash bucket contains one or more index entries.

To create a hash index, we can specify the type explicitly in the `CREATE INDEX` command, like this:

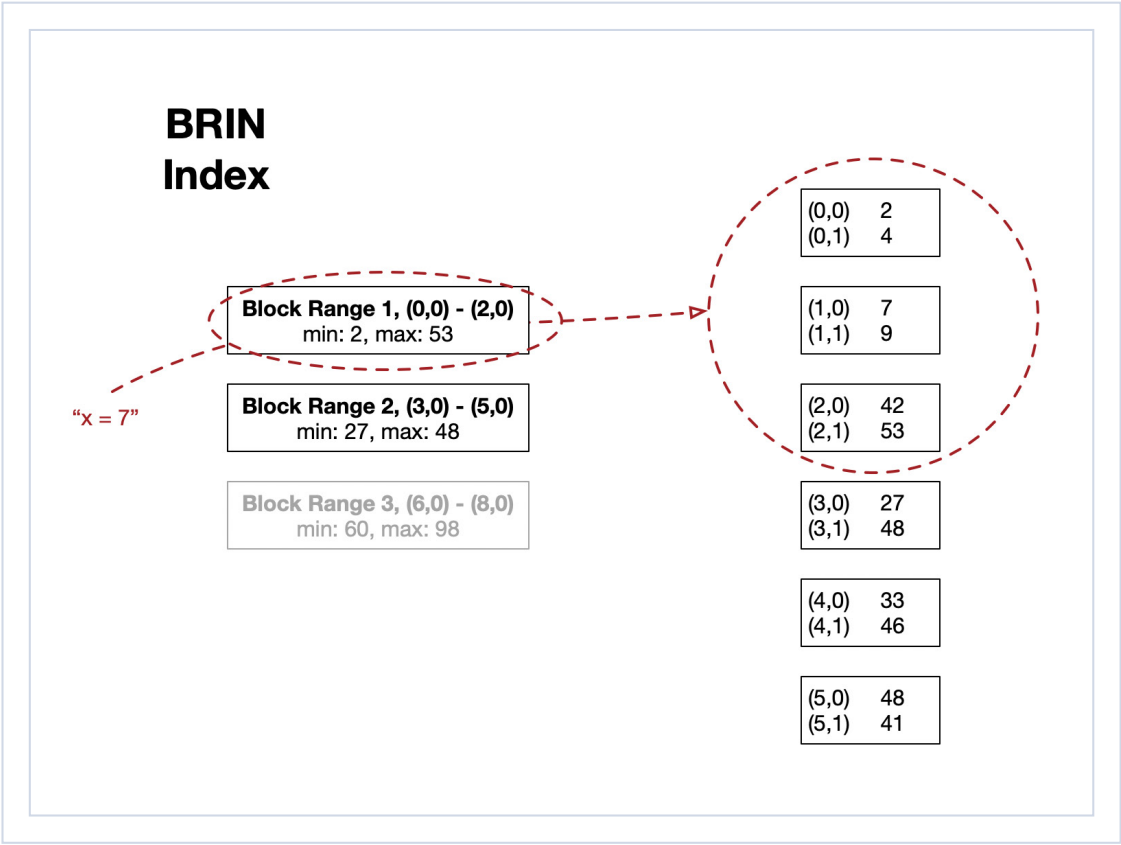
SQL

```
1   CREATE INDEX ON mytable USING hash (column1);
```

For **workloads that only require the equality operator** ("=") and don't need any other advanced B-tree features, using **a hash index can yield a small but visible performance benefit**. When using hash indexes make sure to **only do so with Postgres 10 or newer**, as hash indexes were not crash-safe before Postgres 10.

BRIN indexes

BRIN is short for Block Range Index, and as the name suggests, they exist to index a particular range of blocks. A "Block" in Postgres terminology refers to a section of a table, typically 8 kb in size. This special index is usually very small and imprecise, as it points to a section of the table for a given value, instead of to a specific row directly:



In this example, when searching for the value "7" we can see that the first block range contains

values from 2 to 53, so 7 might be located there. To actually find the matching row we will need to utilize a Bitmap Heap Scan and filter out all rows that don't match. In this case only a single block range matches, but it could very well be that a majority of block ranges have a min/max that includes the searched value. This type of situation would cause a very slow Bitmap Heap Scan, since Postgres would effectively have to look at the whole table.

BRIN indexes are best used with append-only tables, where the indexed values linearly increase and the rows are added at the end of the table, and you don't delete/update values. If there were a lot of updates and deletions, it may cause the performance of the BRIN index to degrade significantly, due to the physical location of the data being non-linear in relation to the indexed value.

Put differently, **BRIN indexes help in very specific situations where the physical structure of the table correlates with the range of a value**, i.e. similar values are in physical proximity in the table on disk. In those situations they are very efficient due to their small size.

Generic Index Types (GIN, GIST, SP-GIST)

Last but not least, Postgres features infrastructure for generic index types that enable easier implementation of custom indexes that are based on a data structure that can be reused for different domain specific applications.

There are three generic index types in Postgres:

- **GIN:** Inverted data structures - commonly the index entries are composite values and the kind of query the index supports is a search for one of the elements of the composite value
- **GIST:** Search trees - implements a balanced, tree-structured index that is versatile. B-trees, R-trees and other data structures are a good fit for GIST.
- **SP-GIST:** Spatial search trees - implements a non-balanced data structure that repeatedly divides the search space into partitions of different sizes, usable for structures such as quad-trees, k-d trees and radix trees.

In practice, there are a few specific cases you will commonly find these index types used for:

- GIN: JSONB (e.g. finding key/value pairs, checking whether keys exist)
- GIN / GIST: Full text search - see our eBook about full text search here
- GIST: Geospatial data
- GIST: ltree
- GIST: Range types (e.g. checking whether values are contained in the range)

In addition, Postgres also includes the optional "btree_gin" and "btree_gist" modules, which can be used to have B-tree like behaviour in a GIN/GIST index. This is useful for combining a typical B-tree operator/data type with one that's better suited for GIN/GIST. An additional use case for "btree_gist" are exclusion constraints, as "btree_gist" supports the non-equal operator ("<>").

Creating The Best Index For Your Queries

Now that we've looked at all the built-in Postgres index types, we can now choose the correct index type for our index. This choice is important, because the index type defines what operator and data type can be supported, and what the actual underlying data structure of the index is.

When we create an index, besides choosing the index type, we have a few other choices to make. One very important choice is whether to index a single column, or whether **to index multiple columns with the same index**. Indexing multiple columns is supported for B-tree, GIST, GIN and BRIN index types. What exactly a multi-column index does is dependent on the index type.

For the most common example of a B-tree index, there are a few things to watch out for in **multi-column indexes**:

- Ordering matters: If you have different queries using the index, make sure that the columns in the beginning are used by all the queries, and only the columns later in the index are used by a subset of queries. Putting a rarely queried column at the beginning of the index definition will cause those

index values to be read every time the deeper nested values need to be searched - causing a lot of wasted I/O.

- In Postgres versions prior to Postgres 13, you can benefit from putting high-cardinality columns in front of the index - that is, if a column's value is very specific, it's better if its early in the query, to avoid having a lot of duplicated entries to scan through in a low-cardinality column. This mostly no longer holds true with Postgres 13, due to B-tree deduplication automatically avoiding duplicate index entries.
- The more columns you index, the bigger the index gets - making it unlikely to be picked by the planner, since scanning it is expensive. As a ballpark rule, it rarely makes sense to create multi-column indexes that have more than 3 indexed columns.
- If you want to include additional columns to achieve Index Only Scans, that is, to have the index contain all the data that you need returned (not just the one you are querying by), you can use the `INCLUDE` keyword to specify columns whose data is included in the index, but which are not used for filtering purposes. This is also called a covering index. This improves overall performance of the B-tree index, as compared to just putting those columns directly into the index definition.

When you compare having one index on multiple columns vs multiple indexes on one column each, **it often is preferable to rely on a single index**. It's also important to note that combining multiple indexes using a Bitmap Index/Heap Scan is only supported when they are of the same index type.

Now, let's take a look at another common situation you will run into with complex queries. Oftentimes our queries can contain more complex expressions than just a simple comparison with a column value. For example, you might have a query like this:

SQL

```
1   SELECT * FROM mytable WHERE lower(column3) = 'foobar';
```

If you simply create an index on "column3", the index could not be used, as the query would first have to get all the column values, and then apply the function.

But you're in luck: Not only can you create an index on one or multiple columns - the specified columns don't have to be the actual columns after all!

Instead, you can use functions in your index definition, creating what's called an **expression index**. You can create an index on the function, or

more generally speaking, the expression, like this:

SQL

```
1   CREATE INDEX ON mytable (lower(column3));
```

This indexes the result of "lower(column3)" instead of "column3" itself, which can then be used to satisfy queries that are searching in that particular way. Note that if you also query for "column3" without calling the lower function, you will need a second index that can support these queries.

Last but not least, let's look at **partial indexes**. Simply put, these are indexes with a `WHERE` clause:

SQL

```
1   CREATE INDEX ON mytable (column1) WHERE column4 IS NULL;
```

This Postgres index feature can be helpful if you are querying your data with a particular predicate a lot, with the same constant values. For example, you might always check that a "deleted_at" column is NULL, so you are only looking at data that is not yet deleted. **Partial indexes only contain index entries for data that matches the specified WHERE condition**, making their index size smaller

as well. They can be very effective when your query workload is consistent enough to make use of them.

You might be tempted to simply create a partial index for each possible query that your application generates, ensuring that there is always an index that is small and fast. The main problem with a setup like this, that could see hundreds or thousands of partial indexes, is that it takes Postgres longer to determine which index to use during the planning process.

These are the most important index features to know about, which will help you create the best index for your query. And remember: The table/index scan method with the lowest cost wins. Oftentimes a simpler index is better than one that tries to do everything. If you make index changes, always verify whether the new indexes actually get used, for example by running `EXPLAIN` on your queries.

Conclusion

The goal of this eBook is to give you an in-depth understanding of what it takes to create the best index for your situation. It's important to start by evaluating your data types and the operators required by the application workload. This will then lead to picking the right index type, as well as determining whether a more advanced index such as one using multiple columns, or a partial index could be a good fit.

When optimizing indexes, it is important to stay close to the truth of what Postgres actually does. You can check on Postgres by using the `EXPLAIN` command, or by utilizing the auto_explain extension to see which indexes actually get picked by Postgres when a particular slow invocation of a query occurs.

About pganalyze.

DBAs and developers use pganalyze to identify the root cause of performance issues, optimize queries and to get alerts about critical issues.

Our rich feature set lets you optimize your database performance, discover root causes for critical issues, get alerted about problems before they become big, gives you answers and lets you plan ahead.

Hundreds of companies monitor their production PostgreSQL databases with pganalyze.

Be one of them.
Get started easily with a free 14-day trial, or learn more about our Enterprise product.

If you want, you can also request a personal demo.

"Our overall usage of Postgres is growing, as is the amount of data we're storing and the number of users that interact with our products. pganalyze is essential to making our Postgres databases run faster, and makes sure end-users have the best experience possible."

Robin Fernandes, Software Development Manager
Atlassian

Appendix: Built-In Index Types & Supported Data Types and Operators

Index Type	Data Types	Operators
B-tree	Most built-in types	< <= = >= > ~ (LIKE - in some cases) ~* (ILIKE - in some cases)
Hash	Most built-in types	=
BRIN - Min/Max	Most built-in types	< <= = >= >
BRIN - Box Inclusion	box	<< &< && &> >> ~= @> <@ &<| <<| |>> |&>
BRIN - Range Inclusion	any range type	<< &< && &> >> @> <@ -|- = < <= = > >=
BRIN - Network Inclusion	inet	&& >>= <<= = >> <<
GIN - Array	any array type	&& <@ = @>
GIN - JSONB	jsonb	? ?& ?| @> @? @@
GIN - JSONB Path	jsonb	@> @? @@
GIN - Text Search	tsvector	@@ @@@

Appendix: Built-In Index Types & Supported Data Types and Operators

Index Type	Data Types	Operators
GIST - Text Search	tsquery	<@ @>
	tsvector	@@
GIST - Geometry	point	>> >^ << <@ <@ <^ ~= <-> (Order)
GIST - Geometry	box circle polygon	&& &> &< &<| >> << <<| <@ @> @ |&> |>> ~ ~= <-> (Order)
GIST - Inet	inet cidr	&& >> >>= > >= <> << <<= < <= =
SP-GIST - Geometry	point	<< <@ <^ >> >^ ~= <-> (nearest-neighbor search)
SP-GIST - Geometry	box polygon	<< &< && &> >> ~= @> <@ &<| <<| |>> |&> <-> (nearest-neighbor search)
SP-GIST - Range	any range type	&& &< &> -|- << <@ = >> @>
SP-GIST - Text	text	< <= = > >= ~<=~ ~<~ ~>=~ ~>~ ^@
SP-GIST - Inet	inet cidr	&& >> >>= > >= <> << <<= < <= =