Disk Management and Indexes

Contents

Gittu George

Todays Agenda

- HOW do humans store and search data?
- HOW does Postgres store data?
- HOW does Postgres search data?
- WHY indexes?
- WHAT are indexes?
- Different kinds of indexes
- Do I need an index?

Learning objectives

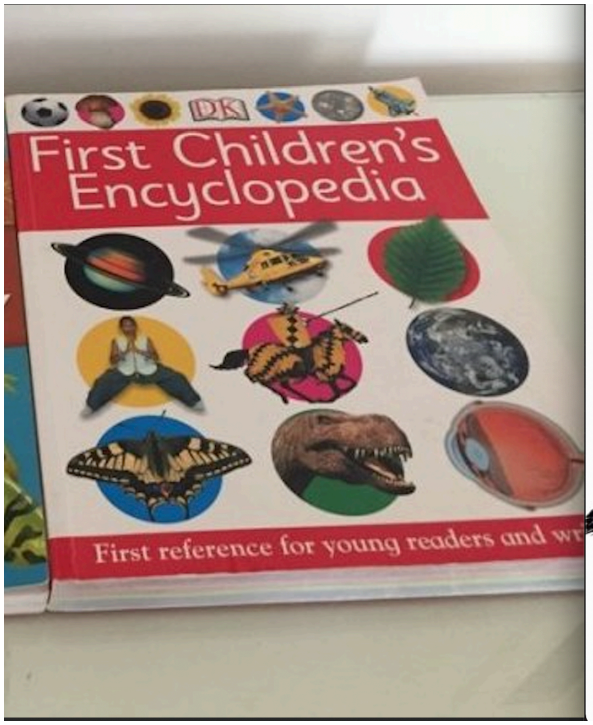
- You will apply knowledge of disk management and query structure to optimize complex queries.
- You will understand the different types of INDEXes available in Postgres and the best-suited applications
- You will be able to profile and optimize various queries to provide the fastest response.

You already have the data we will use in this lecture from worksheet 2, but if not, please load it so that you can follow with me in lecture (You can load this data to your database using this dump. It's a zip file; do make sure you extract it.). I will use the `testindex` table from `fakedata` schema for demonstration.

Skip to main content

How do humans store and search data?

Think about how we used to store data before the era of computers? We used to store the data in books. Let's take this children's encyclopedia as an example.



Index

.

There are around 300 `pages` in this book, and if a kid wants to know about `China`, they will go from the first `page` until they get to the 48th page`. There might be other places too that talk about china. So the kid wants to go over all the pages to gather all the information about China.

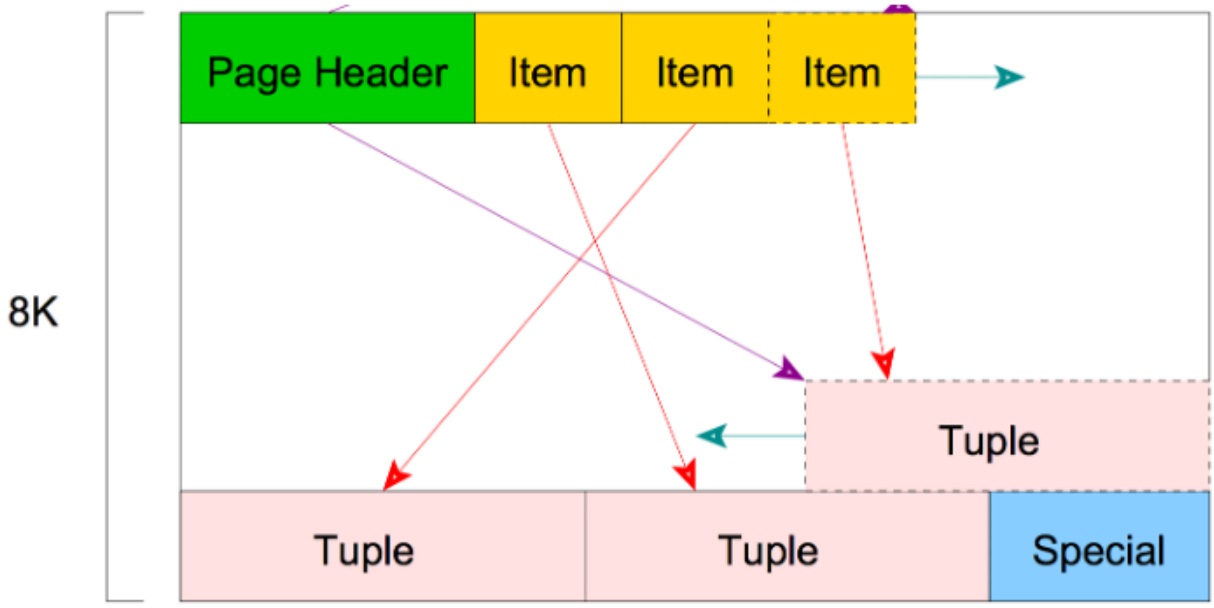
This is how a kid would `search`, but we are grown up and know about indexes in a book. So if you want to read about china, you would go to the `index` of this book, look for `China`, and go to the 48th page straight away (and to other pages listed in the index). If you want to read some history about indexes, check out this article.

OKAY, now you all know how humans search for data from a book; Postgres also store and search data similarly.

How does Postgres store data?

Information within a database is also stored in pages. A database page is not structured like in a book, but it's structured like below.

Skip to main content



.

So these pages are of 8 kB blocks; there are some meta-data as headers (green) at the front of it and item ID (yellow) that points to the tuples(rows). This ends with a special section (blue) left empty in ordinary tables, but if we have some indexes set on the tables, it will have some information to facilitate the index access methods.

So a table within a database will have a list of pages where the data is stored, and these individual pages store multiple tuples within that table.

Read more about it here.

How does Postgres search data?

Let's consider a complicated table.

Name	Department	Mailing Location
Jessica Wong	Computer Science	201-2366 Main Mall
Rachel Pottinger	Computer Science	201-2366 Main Mall
Joel Friedman	Computer Science	201-2366 Main Mall
Joel Friedman	Math	121-1984 Mathematics Rd
Mark MacLean	Math	121-1984 Mathematics Rd

For explanation, let's assume that the contents in this table are stored in 3 page files. Here we are interested in finding the rows with the unique name that starts with `y`. A database, just like a kid, needs to search through the 3 pages to find the rows that contain a name that begins with `y`. This is a sequential search, and this is how a database search by default. This process could be slow if we got millions of rows spread across multiple pages, as the computer needs to go through the entire pages to find the names that start with y.

Also, here in this example, let's assume that Yokamino is on page 2, the database will find this on page 2, but it will still go to page 3 in search of other occurrences of names that start with y. WHY? Because it doesn't know that there is only a single occurrence of a name that starts with y.

Let's look at how Postgres search for data using `EXPLAIN`. We will do this on table `fakedata.testindex`,

Skip to main content

```
import pandas as pd
import matplotlib.pyplot as plt
import psycopg2
import json
import urllib.parse

%load_ext sql
%config SqlMagic.displaylimit = 20
%config SqlMagic.autolimit = 30

with open('credentials.json') as f:
    login = json.load(f)

username = login['user']
password = urllib.parse.quote(login['password'])
host = login['host']
port = login['port']
```

```
%sql postgresql://{username}:{password}@{host}:{port}/postgres
```

```
'Connected: postgres@postgres'
```

```
%%time
%%sql

DROP INDEX if exists fakedata.hash_testindex_index;
DROP INDEX if exists fakedata.pgweb_idx;
```

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
Done.
Done.
CPU times: user 2.73 ms, sys: 1.44 ms, total: 4.17 ms
Wall time: 202 ms
```

```
[]
```

Here, we do a normal search for a product name within the column `productname`.

```
%%time
%%sql
explain analyze select count(*) from fakedata.testindex where productname = 'flavor halibut';
```

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
10 rows affected.
CPU times: user 3.76 ms, sys: 1.53 ms, total: 5.29 ms
Wall time: 4.78 s
```

Skip to main content

QUERY PLAN

Finalize Aggregate (cost=144310.50..144310.51 rows=1 width=8) (actual time=4293.470..4293.553 rows=1 loops=1)

-> Gather (cost=144310.29..144310.50 rows=2 width=8) (actual time=4293.461..4293.547 rows=3 loops=1)

Workers Planned: 2

Workers Launched: 2

-> Partial Aggregate (cost=143310.29..143310.30 rows=1 width=8) (actual time=4286.234..4286.235 rows=1 loops=3)

-> Parallel Seq Scan on testindex (cost=0.00..143310.22 rows=25 width=0) (actual time=4286.231..4286.231 rows=0 loops=3)

Filter: (productname = 'flavor halibut'::text)

Rows Removed by Filter: 3724110

Planning Time: 0.464 ms

Execution Time: 4293.645 ms

We are doing a pattern matching search to return all the `productname` that start with `fla`.

```
%%time
%%sql
EXPLAIN ANALYZE SELECT COUNT(*) FROM fakedata.testindex WHERE productname LIKE 'fla%';
```

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
10 rows affected.
CPU times: user 3.57 ms, sys: 1.76 ms, total: 5.34 ms
Wall time: 1.91 s
```

QUERY PLAN

Finalize Aggregate (cost=144311.60..144311.61 rows=1 width=8) (actual time=1246.016..1246.112 rows=1 loops=1)

-> Gather (cost=144311.39..144311.60 rows=2 width=8) (actual time=1244.293..1246.103 rows=3 loops=1)

Workers Planned: 2

Workers Launched: 2

-> Partial Aggregate (cost=143311.39..143311.40 rows=1 width=8) (actual time=1238.505..1238.506 rows=1 loops=3)

-> Parallel Seq Scan on testindex (cost=0.00..143310.22 rows=465 width=0) (actual time=0.091..1236.500 rows=9271 loops=3)

Filter: (productname ~~ 'fla%'::text)

Rows Removed by Filter: 3714839

Planning Time: 0.064 ms

Execution Time: 1246.140 ms

Note

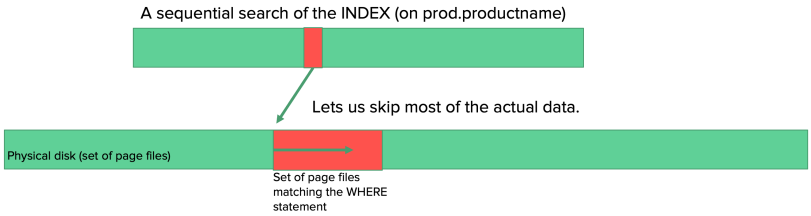
A query is turned into an internal execution plan breaking the query into specific elements that are re-ordered and optimized. Read more about EXPLAIN here. This blog also explains it well.

Skip to main content

These are computers, so it's faster than humans to go through all the pages, but it's inefficient. ***Won't it be cool to tell the database that the data you are looking for is only on certain pages, so database don't want to go through all these pages !!!*** This is WHY we want indexes. We will see if indexing can speed up the previous query.

WHAT are indexes?

Indexes make a map of each of these rows effectively as where they are put into the page files so we can do a sequential scan of an index to find the pointer to page/pages this information is stored. So the database can go straight to those pages and skip the rest of the pages.



Index gets applied to a column in a table. As we were querying on the column `productname`, let's now apply an index to the column `productname` in the database and see how the query execution plan changes. Will it improve the speed? Let's see..

```
%%time
%%sql
CREATE INDEX if not exists default_testindex_index ON fakedata.testindex (productname varchar_pattern_ops
```

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
Done.
CPU times: user 3.47 ms, sys: 1.79 ms, total: 5.26 ms
Wall time: 10.5 s
```

```
[]
```

```
%%time
%%sql
explain analyze select count(*) from fakedata.testindex where productname = 'flavor halibut';
```

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
6 rows affected.
CPU times: user 3.81 ms, sys: 1.46 ms, total: 5.27 ms
Wall time: 457 ms
```

QUERY PLAN

Aggregate (cost=5.63..5.64 rows=1 width=8) (actual time=0.035..0.035 rows=1 loops=1)

-> Index Only Scan using default_testindex_index on testindex (cost=0.43..5.48 rows=60 width=0) (actual time=0.032..0.032 rows=0 loops=1)

Index Cond: (productname = 'flavor halibut'::text)

Heap Fetches: 0

Planning Time: 0.194 ms

Skip to main content

```
%%time
%%sql
EXPLAIN ANALYZE
SELECT COUNT(*) FROM fakedata.testindex
WHERE productname LIKE 'fla%';
```

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
7 rows affected.
CPU times: user 2.06 ms, sys: 1.39 ms, total: 3.45 ms
Wall time: 42 ms
```

QUERY PLAN

Aggregate (cost=7.25..7.26 rows=1 width=8) (actual time=4.377..4.378 rows=1 loops=1)

-> Index Only Scan using default_testindex_index on testindex (cost=0.43..4.46 rows=1117 width=0) (actual time=0.046..3.112 rows=27814 loops=1)

Index Cond: ((productname ~>=~ 'fla'::text) AND (productname ~<~ 'flb'::text))

Filter: (productname ~~ 'fla%'::text)

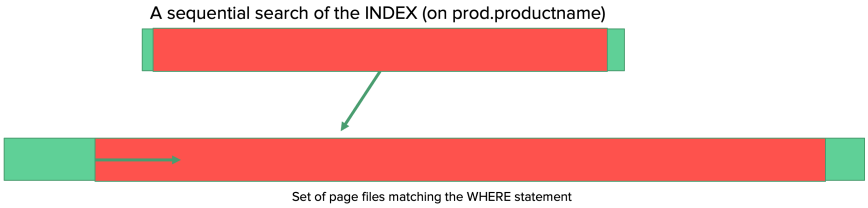
Heap Fetches: 0

Planning Time: 0.192 ms

Execution Time: 4.413 ms

Hurrayyy!!!! It increased the speed, and you see that your query planner used the index to speed up queries.

But keep in mind if the selection criteria are too broad, or the INDEX is too imprecise, the query planner will skip it.



Let's try that out.

```
%%time
%%sql

EXPLAIN ANALYZE
SELECT COUNT(*) FROM fakedata.testindex
WHERE productname LIKE '%';
```

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
9 rows affected.
CPU times: user 3.24 ms, sys: 1.05 ms, total: 4.29 ms
Wall time: 1.97 s
```

Skip to main content

QUERY PLAN
Finalize Aggregate (cost=155947.12..155947.13 rows=1 width=8) (actual time=1863.853..1863.921 rows=1 loops=1)
-> Gather (cost=155946.90..155947.11 rows=2 width=8) (actual time=1863.843..1863.914 rows=3 loops=1)
Workers Planned: 2
Workers Launched: 2
-> Partial Aggregate (cost=154946.90..154946.91 rows=1 width=8) (actual time=1858.746..1858.747 rows=1 loops=3)
-> Parallel Seq Scan on testindex (cost=0.00..143310.22 rows=4654672 width=0) (actual time=0.035..1380.389 rows=3724110 loops=3)
Filter: (productname ~~ '%'::text)
Planning Time: 0.067 ms
Execution Time: 1863.953 ms

It's good that database engines are smart enough to identify if it's better to look at an index or perform a sequential scan. Here in this example, the database engine understands the query is too broad. But, ultimately, it's going to search through entire rows, so it's better to perform a sequential scan rather than look up at the index.

As the index is making a map in the disk, it takes up disk storage. Let's see how much space this index is taking up

```
%%time
%%sql
SELECT pg_size_pretty (pg_indexes_size('fakedata.testindex'));
```

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
1 rows affected.
CPU times: user 1.72 ms, sys: 1.17 ms, total: 2.89 ms
Wall time: 37.7 ms
```

pg_size_pretty
83 MB

Now we realize indexes are great! So let's try some different queries.

```
%%time
%%sql

EXPLAIN ANALYZE
SELECT COUNT(*) FROM fakedata.testindex
WHERE productname LIKE '%fla%';
```

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
10 rows affected.
CPU times: user 2.91 ms, sys: 1.3 ms, total: 4.21 ms
Wall time: 1.46 s
```

Skip to main content

QUERY PLAN

Finalize Aggregate (cost=144311.60..144311.61 rows=1 width=8) (actual time=1410.933..1412.149 rows=1 loops=1)

-> Gather (cost=144311.39..144311.60 rows=2 width=8) (actual time=1410.924..1412.142 rows=3 loops=1)

Workers Planned: 2

Workers Launched: 2

-> Partial Aggregate (cost=143311.39..143311.40 rows=1 width=8) (actual time=1404.357..1404.358 rows=1 loops=3)

-> Parallel Seq Scan on testindex (cost=0.00..143310.22 rows=465 width=0) (actual time=0.051..1401.221 rows=22257 loops=3)

Filter: (productname ~~ '%fla%'::text)

Rows Removed by Filter: 3701853

Planning Time: 0.073 ms

Execution Time: 1412.184 ms

This was not too broad search; we were trying to return all the elements that contains `fla`. ***But why didn't it speed things up? So we can't always go with default indexing, and it might not be helpful in all cases.*** That's why it's important to know about different indexes and a general understanding of how it works. This will help you to choose indexes properly in various situations.

Different kinds of indexes

In this section, we will go through various kinds of indexes and syntax for creating them;

- B-Tree (binary search tree - btree)
- Hash
- GIST (Generalized Search Tree)
- GIN (Generalized Inverted Tree)
- BRIN (Block Range Index)

Each has its own set of operations, tied to Postgres functions/operators. For example, you can read about indexes here.

The general syntax for making an index:

```
CREATE INDEX indexname ON schema.tablename USING method (columnname opclass);
```

B-Tree

B-Tree is the default index, and you can see we used this previously for column `productname`.

General Syntax: Here the `Operator_Classes` is optional. Read more on it here

```
CREATE INDEX if not exists index_name ON schema.tablename (columnname Operator_Classes);
```

Example:

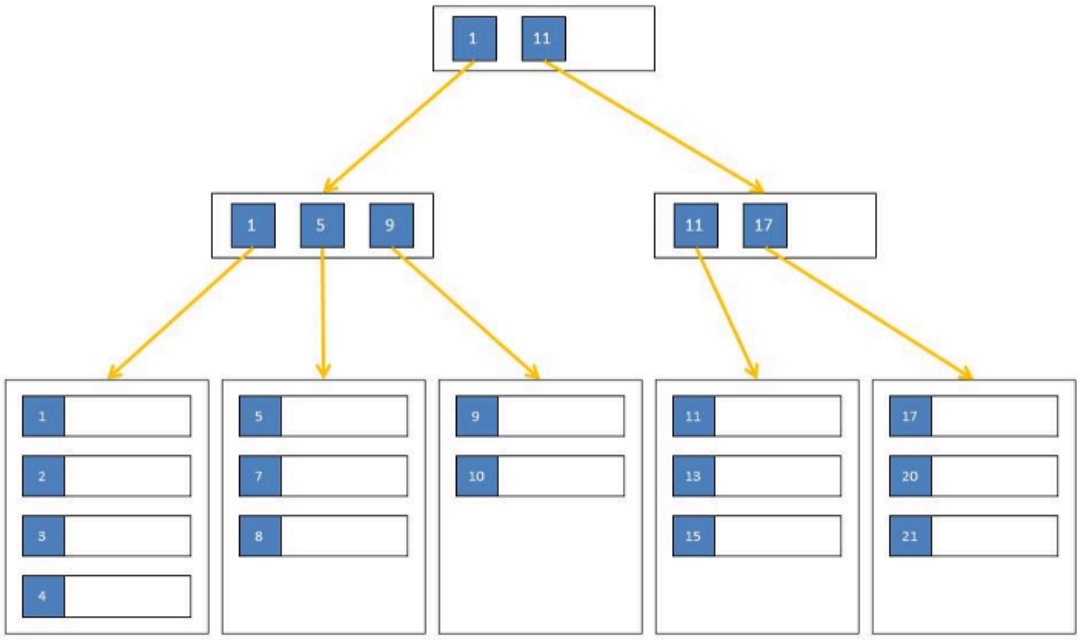
Skip to main content

```
CREATE INDEX if not exists default_testindex_index ON fakedata.testindex (productname varchar_pattern_ops)
```

> **Note**
>
> If you are using B-Tree indexing on a numeric column that you don't want to specify 'Operator_Classes'. Operator class `varchar_pattern_ops` is used for character column.

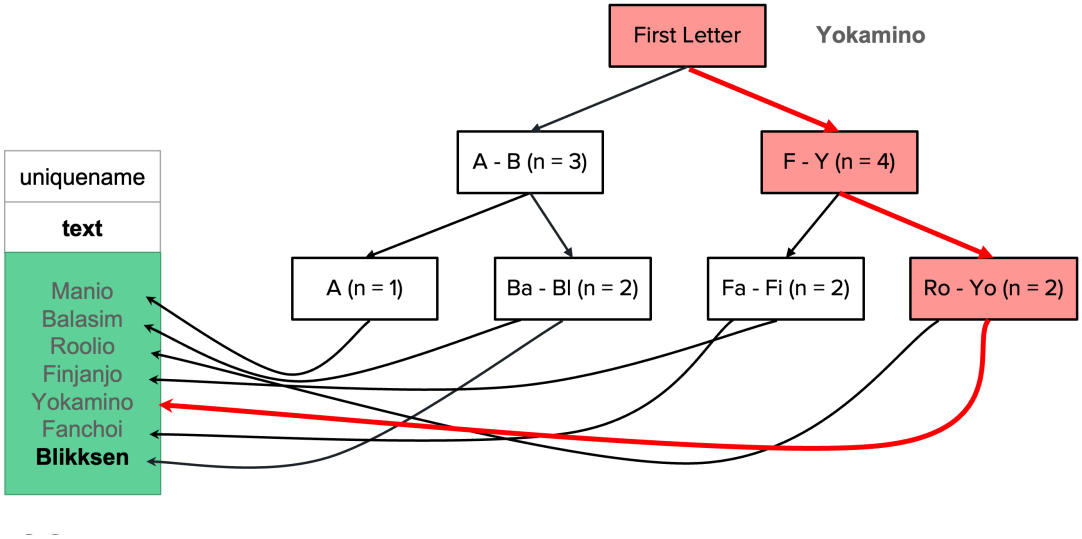
You will do it on a numeric column in worksheet 3.

- Values are balanced across nodes to produce a minimum-spanning tree.
- Each node splits values, each "leaf" points to addresses.
- All nodes are equidistant from the root.
- High Key



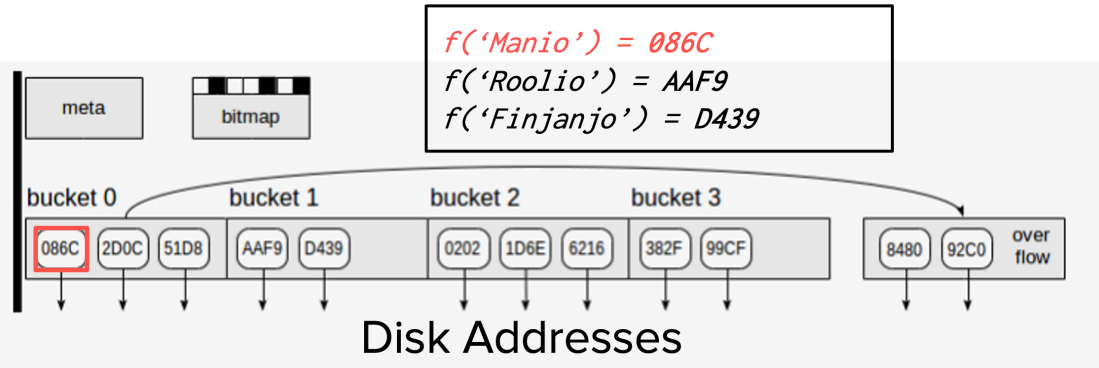
Btree is not just about dealing with numbers but also handling text columns; that's the reason why I gave `varchar_pattern_ops` as `opclass`. So in this example, finds the field that we are looking for in just 3 steps, rather than scanning 7 rows.

Skip to main content



Hash

- Cell values are hashed (encoded) form and mapped to address “buckets”
- Hash -> bucket mappings -> disk address
- The hash function tries to balance the number of buckets & number of addresses within a bucket
- Hash only supports EQUALITY



Let’s create a hash index on column `productname` and execute queries that we ran before

```
%%time
%%sql

DROP INDEX fakedata.default_testindex_index;
CREATE INDEX if not exists hash_testindex_index ON fakedata.testindex USING hash(productname);
```

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
Done.
Done.
CPU times: user 9 ms, sys: 3.08 ms, total: 12.1 ms
Wall time: 16.3 s
```

```
[]
```

Skip to main content

```
%%time
%%sql
explain analyze select count(*) from fakedata.testindex where productname = 'flavor halibut';
```

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
7 rows affected.
CPU times: user 2.96 ms, sys: 1.55 ms, total: 4.5 ms
Wall time: 413 ms
```

QUERY PLAN

Aggregate (cost=240.59..240.60 rows=1 width=8) (actual time=0.012..0.012 rows=1 loops=1)

-> Bitmap Heap Scan on testindex (cost=4.46..240.44 rows=60 width=0) (actual time=0.009..0.009 rows=0 loops=1)

Recheck Cond: (productname = 'flavor halibut'::text)

-> Bitmap Index Scan on hash_testindex_index (cost=0.00..4.45 rows=60 width=0) (actual time=0.008..0.008 rows=0 loops=1)

Index Cond: (productname = 'flavor halibut'::text)

Planning Time: 0.297 ms

Execution Time: 0.038 ms

```
%%time
%%sql
EXPLAIN ANALYZE
SELECT COUNT(*) FROM fakedata.testindex
WHERE productname LIKE 'fla%';
```

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
10 rows affected.
CPU times: user 3.58 ms, sys: 1.58 ms, total: 5.16 ms
Wall time: 1.54 s
```

QUERY PLAN

Finalize Aggregate (cost=144311.60..144311.61 rows=1 width=8) (actual time=1088.010..1088.146 rows=1 loops=1)

-> Gather (cost=144311.39..144311.60 rows=2 width=8) (actual time=1086.327..1088.135 rows=3 loops=1)

Workers Planned: 2

Workers Launched: 2

-> Partial Aggregate (cost=143311.39..143311.40 rows=1 width=8) (actual time=1082.838..1082.840 rows=1 loops=3)

-> Parallel Seq Scan on testindex (cost=0.00..143310.22 rows=465 width=0) (actual time=0.153..1077.249 rows=9271 loops=3)

Filter: (productname ~~ 'fla%'::text)

Rows Removed by Filter: 3714839

Planning Time: 0.081 ms

Execution Time: 1088.174 ms

```
%%time
%%sql
SELECT pg_size_pretty (pg_indexes_size('fakedata.testindex'));
```

Skip to main content

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
1 rows affected.
CPU times: user 1.47 ms, sys: 1.14 ms, total: 2.61 ms
Wall time: 37.4 ms
```

pg_size_pretty
381 MB

- Why do you think hash takes up more space here?
- When do you think it's best to you hash indexing?

GIN

Before talking about GIN indexes, let me give you a little bit of background on a full-text search. Full-text search is usually used if you have a column with sentences and you need to query rows based on the match for a particular string in that sentence. For example, say we want to get rows with `new` in the `sentence` column.

row number	sentence
1	this column has to do some thing with new year, also has to do something with row
2	this column has to do some thing with colors
3	new year celebrated on 1st Jan
4	new year celebration was very great this year
5	This is about cars released this year

We usually query it this way, this definitely will return our result, but it takes time.

```
%%time
%%sql

SELECT * FROM fakedata.testindex
WHERE productname LIKE '%new%';
```

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
27510 rows affected.
CPU times: user 19.6 ms, sys: 15 ms, total: 34.6 ms
Wall time: 2.4 s
```

Skip to main content

nameid	productid	productname	price
60	325578	replace news	$41.82
171	34359	newsstand bird	$26.18
183	24364	newsstand Charles	$76.91
201	142804	Venezuela news	$8.37
285	90194	news cub	$31.87
392	185778	ounce newsstand	$92.40
496	158475	newsprint softball	$81.24
582	83529	hub newsstand	$87.06
644	130920	news carbon	$44.99
691	158475	newsprint softball	$81.24
760	221233	news frog	$64.35
783	199957	cathedral newsstand	$23.38
798	92081	cod newsstand	$87.92
858	250217	result newsstand	$50.36
1002	204775	sing newsstand	$57.54
1025	83529	hub newsstand	$87.06
1262	166133	chauffeur newsprint	$77.66
1324	24364	newsstand Charles	$76.91
1410	51369	lynx newsstand	$0.85
1490	158475	newsprint softball	$81.24

30 rows, truncated to displaylimit of 20

That's why we go for full-text search; I found this blog to help understand this.

```
SELECT * FROM fakedata.testindex
WHERE to_tsvector('english', sentence) @@ to_tsquery('english','new');'
```

Here we are converting the type of column `sentence` to `tsvector` Here the first row in sentence column `this column has to do something with the new year, also has to do something with row` will be represented like this internally

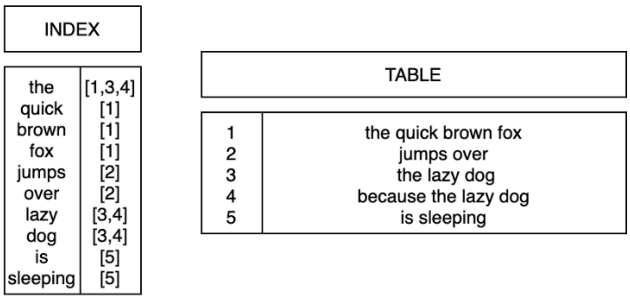
```
'also':11 'column':2 'do':5,14 'has':3,12 'new':9 'row':17 'some':6 'something':15 'thing':7 'this':1 'to
```

`to_tsquery` is how you query, and here we query for `new` using `to_tsquery('english','new')`.

Postgres does a pretty good job with the full-text search, but if we want to speed up the search, we go for GIN indexes. The column needs to be of `tsvector` type for a full-text search.

- Indexing many values to the same row
- Inverse of B-tree - one value to many rows e.g., "quick", or "brown" or "the" all point to row 1
- Most commonly used for `full-text searching.`

Skip to main content



Let's try these on our tables; Following is how much time it will take without indexing.

```
%%time
%%sql

EXPLAIN ANALYZE SELECT count(*) FROM fakedata.testindex WHERE to_tsvector('english', productname) @@ to_t
```

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
10 rows affected.
CPU times: user 4.39 ms, sys: 1.71 ms, total: 6.1 ms
Wall time: 25.6 s
```

QUERY PLAN

Finalize Aggregate (cost=1308153.11..1308153.12 rows=1 width=8) (actual time=25421.466..25421.542 rows=1 loops=1)

-> Gather (cost=1308152.89..1308153.10 rows=2 width=8) (actual time=25419.402..25421.533 rows=3 loops=1)

Workers Planned: 2

Workers Launched: 2

-> Partial Aggregate (cost=1307152.89..1307152.90 rows=1 width=8) (actual time=25394.768..25394.769 rows=1 loops=3)

-> Parallel Seq Scan on testindex (cost=0.00..1307094.70 rows=23276 width=0) (actual time=35.912..25391.699 rows=3158 loops=3)

Filter: (to_tsvector('english'::regconfig, productname) @@ '''flavor'''::tsquery)

Rows Removed by Filter: 3720952

Planning Time: 0.826 ms

Execution Time: 25421.609 ms

Let's see how much time it is going to take with indexing. Before that we want to create index for the column.

```
%%time
%%sql

DROP INDEX if exists fakedata.hash_testindex_index;
DROP INDEX if exists fakedata.pgweb_idx;
CREATE INDEX if not exists pgweb_idx ON fakedata.testindex USING GIN (to_tsvector('english', productname)
```

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
Done.
Done.
Done.
CPU times: user 7.82 ms, sys: 2.77 ms, total: 10.6 ms
Wall time: 40.7 s
```

Skip to main content

```
[]
```

Here is how much time it is taking with indexing.

```
%%time
%%sql

EXPLAIN ANALYZE SELECT count(*) FROM fakedata.testindex WHERE to_tsvector('english', productname) @@ to_t
```

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
12 rows affected.
CPU times: user 1.9 ms, sys: 1.16 ms, total: 3.07 ms
Wall time: 71.1 ms
```

QUERY PLAN

Finalize Aggregate (cost=87233.51..87233.52 rows=1 width=8) (actual time=25.011..26.488 rows=1 loops=1)

-> Gather (cost=87233.30..87233.51 rows=2 width=8) (actual time=24.412..26.478 rows=3 loops=1)

Workers Planned: 2

Workers Launched: 2

-> Partial Aggregate (cost=86233.30..86233.31 rows=1 width=8) (actual time=10.295..10.296 rows=1 loops=3)

-> Parallel Bitmap Heap Scan on testindex (cost=520.93..86175.11 rows=23276 width=0) (actual time=2.513..8.635 rows=3158 loops=3)

Recheck Cond: (to_tsvector('english'::regconfig, productname) @@ '''flavor'''::tsquery)

Heap Blocks: exact=6104

-> Bitmap Index Scan on pgweb_idx (cost=0.00..506.96 rows=55862 width=0) (actual time=5.155..5.156 rows=9474 loops=1)

Index Cond: (to_tsvector('english'::regconfig, productname) @@ '''flavor'''::tsquery)

Planning Time: 0.914 ms

Execution Time: 26.536 ms

Wooohoooo!!! A great improvement!! By creating a materialized view (We will see more about views in next class) with a computed `tsvector` column, we can make searches even faster, since it will not be necessary to redo the `to_tsvector` calls to verify index matches.

```
%%time
%%sql
CREATE MATERIALIZED VIEW if not exists fakedata.testindex_materialized as select *,to_tsvector('english',
FROM fakedata.testindex;
CREATE INDEX if not exists pgweb_idx_mat ON fakedata.testindex_materialized USING GIN (productname_ts);
```

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
Done.
Done.
CPU times: user 2.54 ms, sys: 1.12 ms, total: 3.66 ms
Wall time: 133 ms
```

Skip to main content

```
%%time
%%sql

EXPLAIN ANALYZE SELECT count(*) FROM fakedata.testindex_materialized WHERE productname_ts @@ to_tsquery('
```

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
8 rows affected.
CPU times: user 2.09 ms, sys: 1.06 ms, total: 3.16 ms
Wall time: 49.6 ms
```

QUERY PLAN

Aggregate (cost=36143.83..36143.84 rows=1 width=8) (actual time=5.859..5.861 rows=1 loops=1)

-> Bitmap Heap Scan on testindex_materialized (cost=116.36..36114.04 rows=11917 width=0) (actual time=2.774..4.968 rows=9474 loops=1)

Recheck Cond: (productname_ts @@ '''flavor'''::tsquery)

Heap Blocks: exact=9117

-> Bitmap Index Scan on pgweb_idx_mat (cost=0.00..113.38 rows=11917 width=0) (actual time=1.396..1.397 rows=9474 loops=1)

Index Cond: (productname_ts @@ '''flavor'''::tsquery)

Planning Time: 0.253 ms

Execution Time: 5.890 ms

This indexing does speed up things if we want to search for a particular word from a column, like `flavor`. But if we want to search for some pattern within a sentence, then this index won't help. EG the query what we trying from beginning

```
%%time
%%sql

EXPLAIN ANALYZE
SELECT COUNT(*) FROM fakedata.testindex
WHERE productname LIKE '%fla%';
```

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
10 rows affected.
CPU times: user 4.38 ms, sys: 1.52 ms, total: 5.9 ms
Wall time: 1.65 s
```

Skip to main content

QUERY PLAN

Finalize Aggregate (cost=144311.60..144311.61 rows=1 width=8) (actual time=1582.412..1582.515 rows=1 loops=1)

-> Gather (cost=144311.39..144311.60 rows=2 width=8) (actual time=1580.652..1582.504 rows=3 loops=1)

Workers Planned: 2

Workers Launched: 2

-> Partial Aggregate (cost=143311.39..143311.40 rows=1 width=8) (actual time=1570.566..1570.567 rows=1 loops=3)

-> Parallel Seq Scan on testindex (cost=0.00..143310.22 rows=465 width=0) (actual time=0.112..1564.991 rows=22257 loops=3)

Filter: (productname ~~ '%fla%'::text)

Rows Removed by Filter: 3701853

Planning Time: 0.064 ms

Execution Time: 1582.544 ms

Hence, we want a different flavor of gin indexing, trigram index. I found this blog to help understand this.

`Trigrams` are a special case of N-grams. The concept relies on dividing the sentence into a sequence of three consecutive letters, and this result is considered as a `set`. Before performing this process

- Two blank spaces are added at the beginning.
- One at the end.
- Double ones replace single spaces.

The trigram set corresponding to “flavor halibut” looks like this:

{” f”,” h”,” fl”,” ha”,ali,avo,but,fla,hal,ibu,lav,lib,”or “,”ut “,vor}

These are considered words, and the rest remains the same as what we discussed before. To use this index add `gin_trgm_ops` as `operator class`. Let’s do it.

```
%%time
%%sql
CREATE EXTENSION pg_trgm;
DROP INDEX if exists fakedata.pgweb_idx;
CREATE INDEX pgweb_idx ON fakedata.testindex USING GIN (productname gin_trgm_ops);
```

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
(psycopg2.errors.DuplicateObject) extension "pg_trgm" already exists

[SQL: CREATE EXTENSION pg_trgm;]
(Background on this error at: https://sqlalche.me/e/14/f405)
CPU times: user 1.54 ms, sys: 1.02 ms, total: 2.56 ms
Wall time: 41.6 ms
```

Let’s see if it speeds up the query that we tried in many cases before.

```
%%time
%%sql

EXPLAIN ANALYZE
SELECT COUNT(*) FROM fakedata.testindex
```

Skip to main content

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
10 rows affected.
CPU times: user 2.51 ms, sys: 1.25 ms, total: 3.76 ms
Wall time: 1.45 s
```

QUERY PLAN

Finalize Aggregate (cost=144311.60..144311.61 rows=1 width=8) (actual time=1374.986..1375.040 rows=1 loops=1)

-> Gather (cost=144311.39..144311.60 rows=2 width=8) (actual time=1374.976..1375.032 rows=3 loops=1)

Workers Planned: 2

Workers Launched: 2

-> Partial Aggregate (cost=143311.39..143311.40 rows=1 width=8) (actual time=1366.057..1366.058 rows=1 loops=3)

-> Parallel Seq Scan on testindex (cost=0.00..143310.22 rows=465 width=0) (actual time=0.069..1360.534 rows=22257 loops=3)

Filter: (productname ~~ '%fla%'::text)

Rows Removed by Filter: 3701853

Planning Time: 0.071 ms

Execution Time: 1375.069 ms

```
%%time
%%sql

EXPLAIN ANALYZE
SELECT COUNT(*) FROM fakedata.testindex
WHERE productname LIKE '%flavor%';
```

```
 * postgresql://postgres:***@mbandtweet.cumg1xvg68gc.us-west-2.rds.amazonaws.com:5432/postgres
10 rows affected.
CPU times: user 3.16 ms, sys: 1.51 ms, total: 4.67 ms
Wall time: 1.66 s
```

QUERY PLAN

Finalize Aggregate (cost=144311.60..144311.61 rows=1 width=8) (actual time=1614.378..1615.948 rows=1 loops=1)

-> Gather (cost=144311.39..144311.60 rows=2 width=8) (actual time=1614.368..1615.941 rows=3 loops=1)

Workers Planned: 2

Workers Launched: 2

-> Partial Aggregate (cost=143311.39..143311.40 rows=1 width=8) (actual time=1571.745..1571.746 rows=1 loops=3)

-> Parallel Seq Scan on testindex (cost=0.00..143310.22 rows=465 width=0) (actual time=1.401..1570.977 rows=3158 loops=3)

Filter: (productname ~~ '%flavor%'::text)

Rows Removed by Filter: 3720952

Planning Time: 1.687 ms

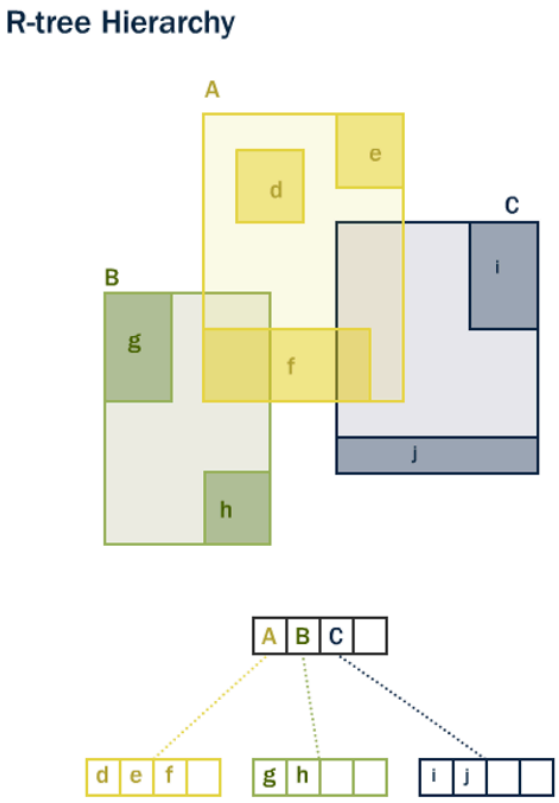
Execution Time: 1615.994 ms

Finally, we see this query is using indexes, and it did speed up the query.

Skip to main content

GIST

- Supports many search types, including spatial and full text.
- Can be extended to custom data types
- Balanced tree-based method (nodes & leaves)



BRIN

- Indexes block statistics for ordered data
- Block min & max mapped to index
- Search finds block first, then finds values
- Best for larger volumes of data
- This is essentially how we read a book's index.

TABLE Block 0	TABLE Block 1	TABLE Block 2	TABLE Block 3
ts_created	ts_created	ts_created	ts_created
2010-01-01 00:00:00	2010-01-01 00:00:09	2010-01-01 00:00:18	2010-01-01 00:00:27
2010-01-01 00:00:01	2010-01-01 00:00:10	2010-01-01 00:00:19	2010-01-01 00:00:28
2010-01-01 00:00:02	2010-01-01 00:00:11	2010-01-01 00:00:20	2010-01-01 00:00:29
2010-01-01 00:00:03	2010-01-01 00:00:12	2010-01-01 00:00:21	2010-01-01 00:00:30
2010-01-01 00:00:04	2010-01-01 00:00:13	2010-01-01 00:00:22	2010-01-01 00:00:31
2010-01-01 00:00:05	2010-01-01 00:00:14	2010-01-01 00:00:23	2010-01-01 00:00:32
2010-01-01 00:00:06	2010-01-01 00:00:15	2010-01-01 00:00:24	2010-01-01 00:00:33
2010-01-01 00:00:07	2010-01-01 00:00:16	2010-01-01 00:00:25	2010-01-01 00:00:34
2010-01-01 00:00:08	2010-01-01 00:00:17	2010-01-01 00:00:26	2010-01-01 00:00:35

Usage

Skip to main content

```
CREATE INDEX index_name ON schema.table/view USING BRIN(columnname);
```

We will use this later when we go through our Twitter example (tomorrow's lecture).

Now we have learned about indexes, can you answer these questions?

- What are indexes?
- Different types of indexes?
- When to use indexes?
- How to use an index?
- Why do I need an index?

Compared to Other Services (optional)

- BigQuery, AWS RedShift
  - Don't use Indexes, infrastructure searches whole "columns"
- MongoDB
  - Single & multiparameter indexes (similar to b-tree)
  - spatial (similar to GIST)
  - text indexes (similar to GIN)
  - hash index
- Neo4j (Graph Database)
  - b-tree and full text

Summary

- Indexes are important to speed up operations
- Indexes are optimized to certain kinds of data
- Index performance can be assessed using `EXPLAIN.`
- Indexes can come at a cost and should be careful in selecting it.

Class activity

> **Note**
>
> Check worksheet 3 in canvas for detailed instructions on activity and submission instructions.

- Setting up index
- Experience the performance with/without and with different types of indexes.