Indexing Essentials in SQL | Atlassian

atlassian.com/data/sql/how-indexing-works

What is Indexing?

Indexing makes columns faster to query by creating pointers to where data is stored within a database.

Imagine you want to find a piece of information that is within a large database. To get this information out of the database the computer will look through every row until it finds it. If the data you are looking for is towards the very end, this query would take a long time to run.

Visualization for finding the last entry:

SELECT * FROM friends WHERE name = 'Zack';

friends		
id	**name**	**city**
1	Matt	San Francisco
2	Dave	Oakland
3	Andrew	Blacksburg
4	Todd	Chicago
5	Blake	Atlanta
6	Evan	Detroit
7	Nick	New York City
8	Zack	Seattle

If the table was ordered alphabetically, searching for a name could happen a lot faster because we could skip looking for the data in certain rows. If we wanted to search for "Zack" and we know the data is in alphabetical order we could jump down to halfway through the data to see if Zack comes before or after that row. We could then half the remaining rows and make the same comparison.

SELECT * FROM friends WHERE name = 'Zack';

friends_name_asc	
Name	**Index**
Andrew	3
Blake	5
Dave	2
Evan	6
Matt	1
Nick	7
Todd	4
Zack	8

This took 3 comparisons to find the right answer instead of 8 in the unindexed data.

Indexes allow us to create sorted lists without having to create all new sorted tables, which would take up a lot of storage space.

What exactly is an Index?

An index is a structure that holds the field the index is sorting and a pointer from each record to their corresponding record in the original table where the data is actually stored. Indexes are used in things like a contact list where the data may be physically stored in the order you add people's contact information but it is easier to find people when listed out in alphabetical order.

Let's look at the index from the previous example and see how it maps back to the original Friends table:

Index

friends_name_asc	
name	**id**
Andrew	3
Blake	5
Dave	2
Evan	6
Matt	1
Nick	7
Todd	4
Zack	8

points to

Table

Friends		
id	**name**	**city**
1	Matt	San Francisco
2	Dave	Oakland
3	Andrew	Blacksburg
4	Todd	Chicago
5	Blake	Atlanta
6	Evan	Detroit
7	Nick	New York City
8	Zack	Seattle

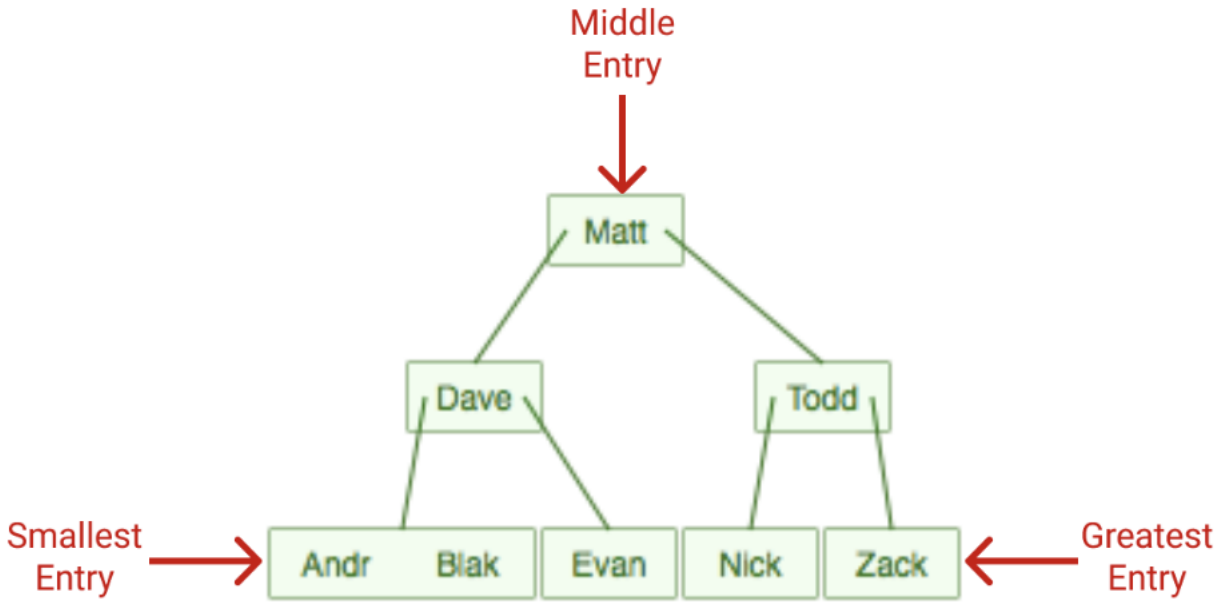
We can see here that the table has the data stored ordered by an incrementing id based on the order in which the data was added. And the Index has the names stored in alphabetical order.

Types of Indexing

There are two types of databases indexes:

1. **Clustered**
2. **Non-clustered**

Both clustered and non-clustered indexes are stored and searched as B-trees, a data structure similar to a binary tree. A B-tree is a "self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time." Basically it creates a tree-like structure that sorts data for quick searching.



Here is a B-tree of the index we created. Our smallest entry is the leftmost entry and our largest is the rightmost entry. All queries would start at the top node and work their way down the tree, if the target entry is less than the current node the left path is followed, if greater the right path is followed. In our case it checked against Matt, then Todd, and then Zack.

To increase efficiency, many B-trees will limit the number of characters you can enter into an entry. The B-tree will do this on it's own and does not require column data to be restricted. In the example above the B-tree below limits entries to 4 characters.

Clustered Indexes

Clustered indexes are the unique index per table that uses the primary key to organize the data that is within the table. The clustered index ensures that the primary key is stored in increasing order, which is also the order the table holds in memory.

- Clustered indexes do not have to be explicitly declared.
- Created when the table is created.
- Use the primary key sorted in ascending order.

Creating clustered Indexes

The clustered index will be automatically created when the primary key is defined:

```
CREATE TABLE friends (id INT PRIMARY KEY, name VARCHAR, city VARCHAR);
```

Once filled in, that table would look something like this:

Friends		
id	**name**	**city**
1	Matt	San Francisco
2	Dave	Oakland
3	Andrew	Blacksburg
4	Todd	Chicago
5	Blake	Atlanta
6	Evan	Detroit
7	Nick	New York City
8	Zack	Seattle

The created table, “friends”, will have a clustered index automatically created, organized around the Primary Key “id” called “friends_pkey”:

friends_pkey
id
1
2
3
4
5
6
7
8

Friends		
id	name	city
1	Matt	San Francisco
2	Dave	Oakland
3	Andrew	Blacksburg
4	Todd	Chicago
5	Blake	Atlanta
6	Evan	Detroit
7	Nick	New York City
8	Zack	Seattle

When searching the table by “id”, the ascending order of the column allows for optimal searches to be performed. Since the numbers are ordered, the search can navigate the B-tree allowing searches to happen in logarithmic time.

However, in order to search for the “name” or “city” in the table, we would have to look at every entry because these columns do not have an index. This is where non-clustered indexes become very useful.

Non-clustered Indexes

Non-clustered indexes are sorted references for a specific field, from the main table, that hold pointers back to the original entries of the table. The first example we showed is an example of a non-clustered table:

Index points to **Table**

friends_name_asc	
name	id
Andrew	3
Blake	5
Dave	2
Evan	6
Matt	1
Nick	7
Todd	4
Zack	8

Friends		
id	name	city
1	Matt	San Francisco
2	Dave	Oakland
3	Andrew	Blacksburg
4	Todd	Chicago
5	Blake	Atlanta
6	Evan	Detroit
7	Nick	New York City
8	Zack	Seattle

They are used to increase the speed of queries on the table by creating columns that are more easily searchable. Non-clustered indexes can be created by data analysts/ developers after a table has been created and filled.

Note: Non-clustered indexes are **not** new tables. Non-clustered indexes hold the field that they are responsible for sorting and a pointer from each of those entries back to the full entry in the table.

You can think of these just like indexes in a book. The index points to the location in the book where you can find the data you are looking for.

Index

A
Additive color model, 3

B
Binding, 20
Bitmap image
  defined, 9
  resolution of, 11
  tonal range in, 11
Bleed, checking, 46
Blueline, 42

C
Chroma, 2
CMS. *See* Color management system
CMY color model, 4
Color
  characteristics of, 2
  checking definitions, 46
  displaying on monitors, 2
  high-fidelity, 15
  in imported illustrations, 45
  matching, 6
  perception of, 2
  proofing, 37, 38–43
  properties of, 1
  screening, 6
  systems for managing, 15
  tints of, 6
Color bars, 50
Color gamut, 4
Color management system, 15
Color model, 3
Color proof
  checking, 50
  contract, 40, 50
  separation-based, 40
Color separations. *See* Separations
Color space, 4
Color value, 2
Commercial printing
  inking, 18
  offsetting, 19
  platemaking, 17
  press check, 40–43, 51
  terminology, 5–8
  types of, 16–??
  wetting, 18
Continuous-tone art
  defined, 5
Contract proof, 40, 50
Creep, 21

D
Device profile, 15
Direct-digital printing, 16
Dot gain, 10

F
File formats, for hand-off, 45
Film separations, 6
  See also Separations
Flat, 20
Flexography, 22
Form, 20
Frequency modulation (FM) screening, 14

G
Gamut, color, 4
GCR (gray-component replacement), 7
Gravure, 22
Gray, shades of, 13

H
Halftone cell, 13
Halftone dot, 5, 13
Halftone frequency, 12
Halftone screen
  defined, 5
  moiré patterns, 9, 15
  process colors, 6
Hand-off, 37
  creating report for, 43
  organizing files for, 46
High-fidelity color, 15
Hue, 2

I
Illustrations
  preparing for imaging, 45
  proofing, 37, 38–43
Imaging, preparing files for, 45
Imposition, 20
Ink
  gray-component replacement, 7
  oversaturating, 6
  undercolor removal, 7
Inking, 18

Non-clustered indexes point to memory addresses instead of storing data themselves. This makes them slower to query than clustered indexes but typically much faster than a non-indexed column.

You can create many non-clustered indexes. As of 2008, you can have up to 999 non-clustered indexes in SQL Server and there is no limit in PostgreSQL.

Creating non-clustered databases(PostgreSQL)

To create an index to sort our friends' names alphabetically:

```
CREATE INDEX friends_name_asc ON friends(name ASC);
```

This would create an index called "friends_name_asc", indicating that this index is storing the **names** from "friends" stored alphabetically in **ascending** order.

friends_name_asc	
name	**id**
Andrew	3
Blake	5
Dave	2
Evan	6
Matt	1
Nick	7
Todd	4
Zack	8

Note that the "city" column is not present in this index. That is because indexes do not store all of the information from the original table. The "id" column would be a pointer back to the original table. The pointer logic would look like this:

friends_name_asc	
name	**id**
Andrew	3
Blake	5
Dave	2
Evan	6
Matt	1
Nick	7
Todd	4
Zack	8

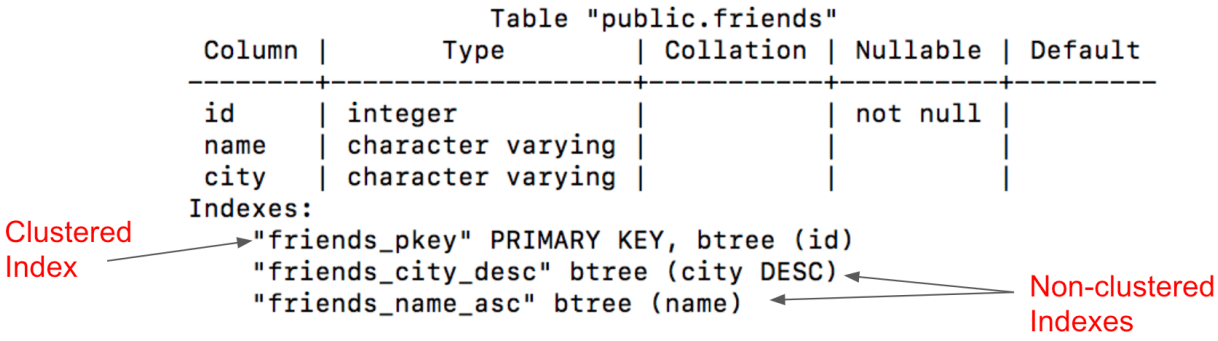
Friends		
id	**name**	**city**
1	Matt	San Francisco
2	Dave	Oakland
3	Andrew	Blacksburg
4	Todd	Chicago
5	Blake	Atlanta
6	Evan	Detroit
7	Nick	New York City
8	Zack	Seattle

Creating Indexes

In PostgreSQL, the “\d” command is used to list details on a table, including table name, the table columns and their data types, indexes, and constraints.

The details of our friends table now look like this:

Query providing details on the friends table: \d friends;



Looking at the above image, the “friends_name_asc” is now an associated index of the “friends” table. That means the query plan, the plan that SQL creates when determining the best way to perform a query, will begin to use the index when queries are being made. Notice that “friends_pkey” is listed as an index even though we never declared that as an index. That is the **clustered index** that was referenced earlier in the article that is automatically created based off of the primary key.

We can also see there is a “friends_city_desc” index. That index was created similarly to the names index:

```
CREATE INDEX friends_city_desc ON friends(city DESC);
```

This new index will be used to sort the cities and will be stored in reverse alphabetical order because the keyword “DESC” was passed, short for “descending”. This provides a way for our database to swiftly query city names.

Searching Indexes

After your non-clustered indexes are created you can begin querying with them. Indexes use an optimal search method known as binary search. Binary searches work by constantly cutting the data in half and checking if the entry you are searching for comes before or after the entry in the middle of the current portion of data. This works well with B-trees because they are designed to start at the middle entry; to search for the entries within the tree you know the entries down the left path will be smaller or before the current entry and the entries to the right will be larger or after the current entry. In a table this would look like:

SELECT * FROM friends WHERE name = ‘Zack’;

friends_name_asc	
Name	**Index**
Andrew	3
Blake	5
Dave	2
Evan	6
Matt	1
Nick	7
Todd	4
Zack	8

Comparing this method to the query of the non-indexed table at the beginning of the article, we are able to reduce the total number of searches from eight to three. Using this method, a search of 1,000,000 entries can be reduced down to just 20 jumps in a binary search.

Binary Search Complexity	
Number of Entries	**Greatest number of searches to find target**
8	3
100	7
1,000	10
10,000	14
100,000	17
1,000,000	20

When to use Indexes

Indexes are meant to speed up the performance of a database, so use indexing whenever it significantly improves the performance of your database. As your database becomes larger and larger, the more likely you are to see benefits from indexing.

When not to use Indexes

When data is written to the database, the original table (the clustered index) is updated first and then all of the indexes off of that table are updated. Every time a write is made to the database, the indexes are unusable until they have updated. If the database is constantly receiving writes then the indexes will never be usable. This is why indexes are typically applied to databases in data warehouses that get new data updated on a scheduled basis(off-peak hours) and not production databases which might be receiving new writes all the time.

NOTE: The newest version of Postgres (that is currently in beta) will allow you to query the database while the indexes are being updated.

Testing Index performance

To test if indexes will begin to decrease query times, you can run a set of queries on your database, record the time it takes those queries to finish, and then begin creating indexes and rerunning your tests.

To do this, try using the EXPLAIN ANALYZE clause in PostgreSQL.:

```
EXPLAIN ANALYZE SELECT * FROM friends WHERE name = 'Blake';
```

Which on my small database yielded:

```
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on friends  (cost=0.00..1.06 rows=1 width=68) (actual time=1.320..1.320 rows=0 loops=1)
   Filter: ((name)::text = 'Blake'::text)
   Rows Removed by Filter: 5
 Planning Time: 4.225 ms
 Execution Time: 1.834 ms
(5 rows)

(END)
```

This output will tell you which method of search from the query plan was chosen and how long the planning and execution of the query took.

Only create one index at a time because not all indexes will decrease query time.

- PostgreSQL's query planning is pretty efficient, so adding a new index may not affect how fast queries are performed.
- Adding an index will always mean storing more data
- Adding an index will increase how long it takes your database to fully update after a write operation.

If adding an index does not decrease query time, you can simply remove it from the database.

To remove an index use the DROP INDEX command:

```
DROP INDEX friends_name_asc;
```

The outline of the database now looks like:

```
                  Table "public.friends"
 Column |       Type        | Collation | Nullable | Default
--------+-------------------+-----------+----------+---------
 id     | integer           |           | not null |
 name   | character varying |           |          |
 city   | character varying |           |          |
Indexes:
    "friends_pkey" PRIMARY KEY, btree (id)
    "friends_city_desc" btree (city DESC)
```

Which shows the successful removal of the index for searching names.

Summary

- Indexing can vastly reduce the time of queries
- Every table with a primary key has one clustered index
- Every table can have many non-clustered indexes to aid in querying
- Non-clustered indexes hold pointers back to the main table
- Not every database will benefit from indexing
- Not every index will increase the query speed for the database

References:

https://www.geeksforgeeks.org/indexing-in-databases-set-1/
https://www.c-sharpcorner.com/blogs/differences-between-clustered-index-and-nonclustered-index1
https://en.wikipedia.org/wiki/B-tree
https://www.tutorialspoint.com/postgresql/postgresql_indexes.htm
https://www.cybertec-postgresql.com/en/postgresql-indexing-index-scan-vs-bitmap-scan-vs-sequential-scan-basics/#

Written by: Blake Barnhill
Reviewed by: Matt David , Matthew Layne

Next Topic

Single quote, double quote, and backticks in MySQL queries→