What Does It Mean for a Column to Be Indexed

11-14 minutes : 5/2/2020

May 2, 2020 · 10 min read

When optimizing queries on a database table, most developers tend to just create an index on the field to be queried. They have questions like

> I don't really understand what it means for a column to be "indexed"
>
> in addition to simply boosting the efficiency of a query, are there any other uses of indexing or things to be aware of?

In this post

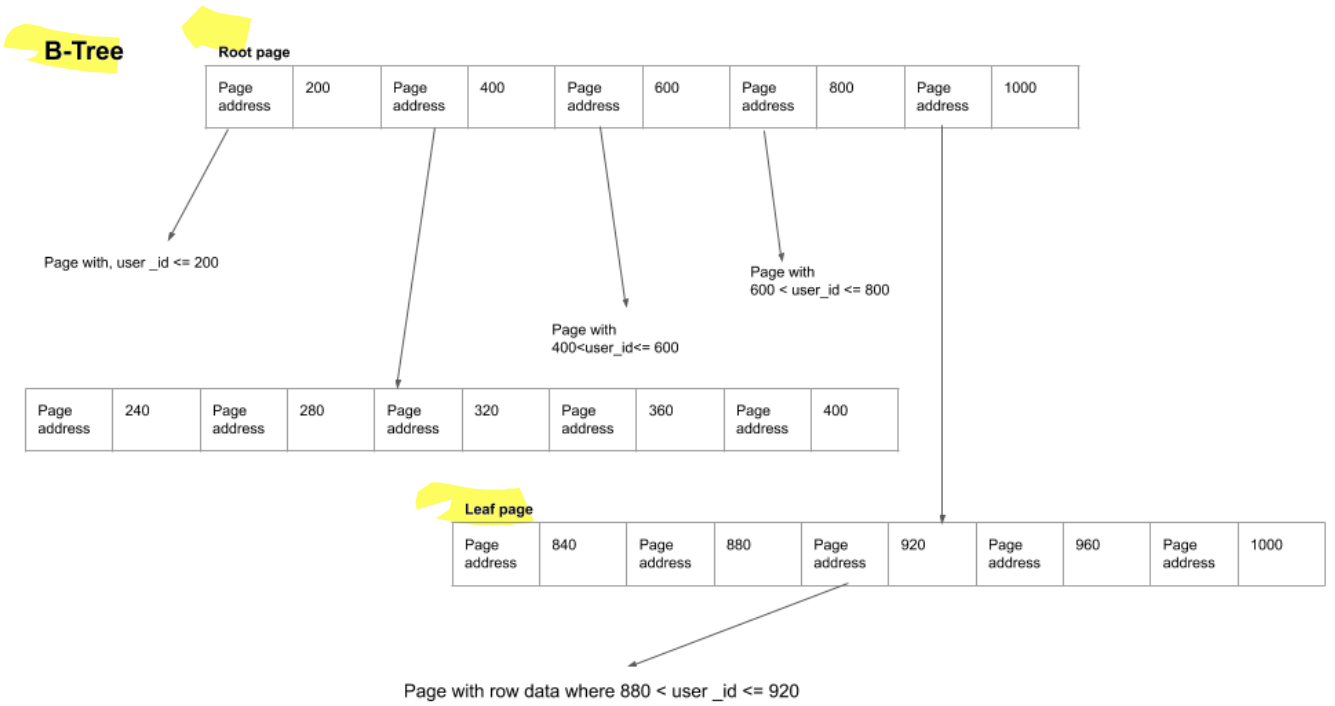
1. We cover the most widely used index structure in databases, the B-tree
2. We go over Scan Types, which is how the database engine reads data from disk
3. Run experiments to understand how index affects CRUD operations for different types of fields and queries
4. Go over the trade-offs that are made when we use an index

B-tree

Invented in the 1970's B-tree's are the most popular data structure used as default indexing mechanism in most databases. Although there are many types of indexing data structures, we are going to only look at b-tree as they are the most popular and widely used. In order to understand B-tree let's consider a simple dataset, users with the following fields

1. `user_id`: a unique identifier for the user (assume user ids start from 1 - 1000)
2. `user_name`: the name of the user
3. `user_account_type`: the account type of the user (assume it can only be one of C or S or I)

The database stores data in small chunks on the disk called `blocks` or `pages`, traditionally 4KB in size. This is hardware implementation based optimization. When we create an index on `user_id` column, a separate B-tree is created with the following approximate shape. The B-tree starts at the root page and has references to other pages. The pages which points to the data are called the `leaf pages`.



As you can see from the diagram B-Tree has many pages , each leading to a leaf page from which we can get a pointer to the data location(implementation depends on the database being used). The number of references to child pages from a parent page is called branching factor in our example it is 5.

The advantage of B-tree is that data access has a Big-O runtime of `O(log(n))` which is very efficient. (ref Big-O)

Then why not create indexes for all columns in the database? Well as with most things in software engineering there is a tradeoff. In case of indexes it is the extra work that the database engine has to do when we insert/update/delete records to keep the B-Tree balanced. We will see the pros and cons of indexing in the last section.

Scan types

All database operations are one of Create Read Update Delete(CRUD) operations. These operations require the database engine to read/write data from the disk, accessing one or many blocks/pages of the data from the disk requires scanning of the data to get the exact block(s). A scan is the operation we aim to optimize with an index. We will see 4 main types of scan

1. `Seq Scan`: The database engine scans through all the data sequentially
2. `Parllel Seq Scan`: The database scans through all the data in parallel with n number of workers (can be gotten using explain). This is not used always because the overhead of starting, managing and reading data from multiple workers is a lot, but used to improve the performance of queries that require scanning a large amount of data but only few of them satisfy the selection criteria.

3. `Index Scan`: If your query has a filter on an indexed column, the database engine uses the B-tree index to get the data location and just reads that page (this is the fastest). Depending on the database, if the database engine determines your query will return > 5-10% of the entire data Index Scan will be skipped for a Seq Scan, this is because the database engine decides that overhead of getting the page location from B-tree index, then reading the page for multiple values of the indexed column is not worth it compared to a simple sequential scan of the entire data.

4. `Bitmap Index Scan`: This is a combination of Index Scan and Seq Scan, this scan is used when the number of rows to be selected are too large for Index Scan and too low for Seq Scan.

Experiments

In this section we will go over some common scenarios, see how the queries perform and cases when a specific type of scan will be preferred by the database engine. The idea is to get a sense of why a certain query may be faster. In the next section we will see the pros and cons of indexing

Set up

In order to run the experiments you will need

1. docker (preferred) to run `postgres`
2. pgcli to connect to our postgres instance
3. Data Download here

Download the data to a folder in your local machine. Docker set up with your data folder as a volume mount

```
docker run --name pg_local -p 5432:5432 -e POSTGRES_USER=start_data_engineer
\
-e POSTGRES_PASSWORD=password -e POSTGRES_DB=tutorial \
-v <your-data-folder-location>:/data -d postgres:12.2
```

Start pgcli

```
pgcli -h localhost -p 5432 -U start_data_engineer tutorial
```

type in password when pgcli prompts you for password, and create a `user` table as shown below

```
CREATE SCHEMA bank;
SET search_path TO bank,public;
CREATE TABLE bank.user (
    user_id int,
    user_name varchar(50),
    score int,
    user_account_type char(1)
```

```
);
COPY bank.user FROM '/data/user.csv' DELIMITER ',' CSV HEADER;
```

Explain

We can use the explain command to explain how the database engine is going to execute our query. It is meant to be read bottom up. (ref: EXPLAIN) This command gives you a cost metric, which is what the database engine aims to minimize.

Experiment 1 (E1)

A select query with equality filter on score, without an index on score

```
explain select * from bank.user where score = 900001;
```

Experiment 1

1. `Scan Type`: `Parallel seq scan` because a scan needs to be done on the entire dataset (since we have no index) and the result will be very few(if any) rows from the table.
2. Execution taken: 0.055s

Experiment 2 (E2)

A select query with range filter on score, without an index on score

```
explain select * from bank.user where score > 900000;
```

Experiment 2

1. `Scan Type`: Seq Scan because a scan needs to be done on the entire dataset (since we have no index).
2. Execution taken: 0.354s

Experiment 3 (E3)

An update query on score with filter on score, without an index on score

```
explain update bank.user set score = 1000000 where score > 1000;
```

Experiment 3

1. `Scan Type`: Seq Scan because a scan needs to be done on the entire dataset (since we have no index) filter for data which has score>1000 and then update the records that satisfy the filter.
2. Execution taken: 2.447s

Experiment 4 (E4)

A select query with equality filter on user_account_type, without an index on user_account_type

```
explain select * from bank.user where user_account_type = 'S';
```

Experiment 4

1. `Scan Type`: Seq Scan because a scan needs to be done on the entire dataset (since we have no index) filter for data which has user_account_type='S' and then update the records that satisfy the filter.
2. Execution taken: 0.830s

Experiment 5 (E5)

A select query with filter on score, with an index on score

```
CREATE INDEX score_idx ON bank.user (score);
-- Time 0.709s
explain select * from bank.user where score = 900001;
```

Experiment 5

1. `Scan Type`: Index Scan which means it uses the b-tree index to get the reference to the page the required data lives in.
2. Execution taken: 0.019s

Experiment 6 (E6)

A select query with range filter on score, with an index on score

```
explain select * from bank.user where score > 900000;
```

Experiment 6

1. `Scan Type`: Bitmap Index Scan which means the number of records that satisfy the condition is estimated to be too large to benefit from Index Scan and is too small to warrant a full Seq Scan
2. Execution taken: 0.411s

```
explain select * from bank.user where score > 600000;
```

Experiment 6.2

1. `Scan Type`: Seq Scan is used even though we have an index on the score column. This is because the database engine estimates that the cost of checking the index, then grabbing the data page and repeating this for each score > 600000 is more than a simple Seq Scan
2. Execution taken: 0.585s

Experiment 7 (E7)

An update query on score with filter on score, with an index on score

```
explain update bank.user set score = 1000000 where score > 1000;
```

Experiment 7

1. `Scan Type`: Bitmap Index Scan which means the number of records that satisfy the condition is estimated to be too large to benefit from Index Scan and is too small to warrant a full Seq Scan
2. Execution taken: 4.239s

Experiment 8 (E8)

A select query with equality filter on user_account_type, with an index on user_account_type

```
CREATE INDEX acct_type_idx ON bank.user (user_account_type);
-- Time: 0.846s
explain select * from bank.user where user_account_type = 'S';
```

Experiment 8

1. `Scan Type`: Bitmap Index Scan which means the number of records that satisfy the condition is estimated to be too large to benefit from Index Scan and is too small to warrant a full Seq Scan
2. Execution taken: 0.660s

```
CREATE SCHEMA bank;
SET search_path TO bank,public;
CREATE TABLE bank.user (
    user_id int,
    user_name varchar(50),
    score int,
    user_account_type char(1)
);
COPY bank.user FROM '/data/user.csv' DELIMITER ',' CSV HEADER;
-- Time: 1.122s

-- E1
explain select * from bank.user where score = 900001;
-- Time: 0.070s

-- E2
explain select * from bank.user where score > 900000;
-- Time: 0.422s

-- E3
```

```
explain update bank.user set score = 1000000 where score > 1000;
-- Time: 2.580s (2 seconds)

-- E4
explain select * from bank.user where user_account_type = 'S';
-- Time: 0.747s

-- Adding Index
CREATE INDEX score_idx ON bank.user (score);
CREATE INDEX acct_type_idx ON bank.user (user_account_type);

-- E5
explain select * from bank.user where score = 900001;
-- Time: 0.017s

-- E6
explain select * from bank.user where score > 900000;
-- Time: 1.351s (a second)
explain select * from bank.user where score > 600000;
-- Time: 1.675s (a second)

-- E7
explain update bank.user set score = 1000000 where score > 1000;
-- Time: 8.627s (8 seconds)

-- E8
explain select * from bank.user where user_account_type = 'S';
-- Time: 0.883s (a moment)

-- Note: Execution time may vary slightly
```

Cardinality

Cardinality is defined as the uniqueness of the data field. For e.g. if we have a field called gender with M or F as acceptable values, then we say that field has low cardinality. Basically you can think of

1. `high cardinality` field has many unique values (e.g. score in our dataset)
2. `low cardinality` field has few unique values (e.g. user_account_type in our dataset)

Conclusion

Hope the above experiments give you a good understanding of when and how to use indexes, how to use explain to check your query performance and what the different types of scans mean.

TL;DR

Pros of using an index:

1. Query with filter on an indexed column is much faster than one without. Check E1 v E5

Cons of using an index:

1. Performance drop for insert/update/delete operations, since the B-tree also has to be managed. Compare E7 and E3, Update time 8.627s with index v 2.447s without index.
2. Depending on the flavor of SQL you are using, Create, Update, Delete operations can be locked while the column is being indexed.
3. An index will not be used always, depending on the estimated result size of your query (use Explain to check your query plan)
4. The increase in performance of index for low cardinality column will be lower than that of a high cardinality one. In our example reduction in latency by adding index on a low cardinality field user_account_type was 20%, whereas the reduction in latency by adding index on a high cardinality field score was 65%

Hope this gives you a good idea of how to effectively use indexes. If you have any questions or comments please reach out or leave a comment below.

Previous Chapter Next Chapter