Performance Optimization: How to Get the Best Out of Your Indexes on Postgres and MySQL

Charly Batista

PERCONA

A brief introduction...

- Postgres Tech Lead at Percona
- PostgreSQL and MySQL expert
- Working with development and databases for over 20 years
- Presenter at conferences and meetups about Database and Development

Agenda

Overview

PERCONA

What is this webinar about?

Have you ever wondered why a PK index is much faster than any secondary index in MySQL but not necessarily faster than secondary indexes in Postgres?

Maybe you found that the optimizer did a full table scan even having a nice index that was able to solve the query?

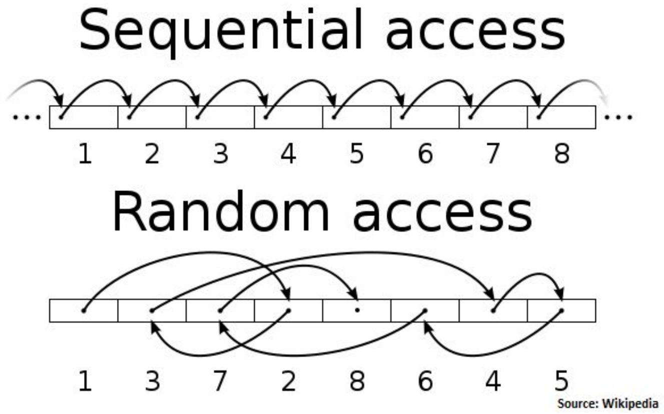
This is what we'll talk about today, INDEX, but not to say that we need to use index but how we can use them.

We will see how the data is stored and organized internally in both Postgres and MySQL. This will help us to understand how the data is accessed and how the indexes are used, consequently, helping us to understand how we can use them to improve performance!

Data Locality

Data Access Methods

- **Data in disk can only be read/written in blocks**
- **Most of the systems use blocks of 4kB size**
- **We are only interested in 2 access methods:**
  - Sequential access
  - Random access
- **Random access is still slower than sequential access**
- **Enforcing sequential IO improves performance**



Cache and Data Locality

- **We are only interested in temporal and spatial locality here**
- **Spatial locality**
  - If a particular storage location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future;
- **Temporal locality:**
  - If at one point a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future;
- **Data locality is particularly relevant when designing storage caches;**
- **DBMS usually make use of data locality when designing their buffers and caches:**
  - Write merges make usage of temporal locality;
  - Read-ahead algorithms make use of spatial locality;
  - Query caches make usage of both temporal and spatial locality;

Heap or Clustered?

PERCONA

Heap File

One of the simplest form of file organization

Unordered set of records stored on pages

Insert efficient

- New records are inserted at the end of the file

No sorting or ordering of the records can be expected

Once the page is full, next record is stored in a new page

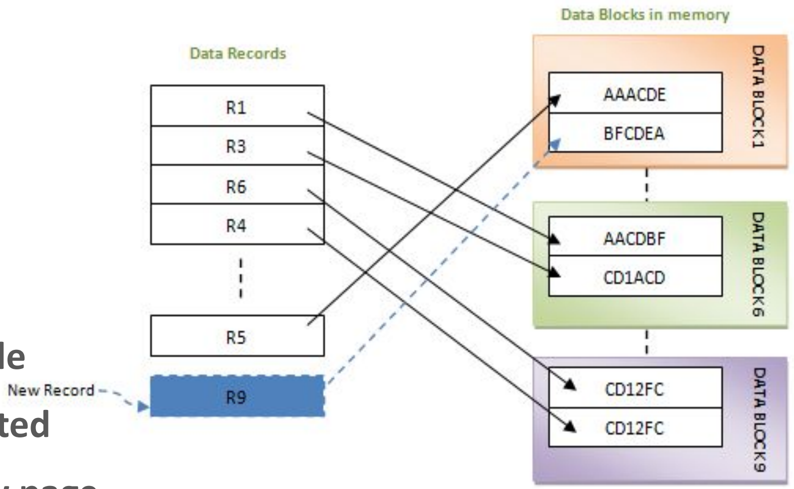
The new page is logically the next closer page

The new page can be physically located anywhere in the disk

Deletion is accomplished by marking records as "deleted"

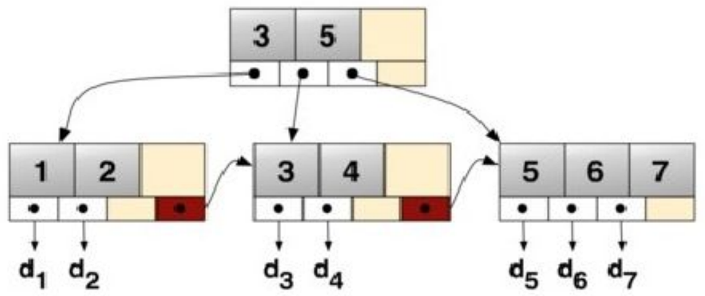
Update is done by: “delete” the old record and insert the new one

Indexes are stored separately and reference the heap file records



https://www.tutorialcup.com/dbms/heap-file-organization.htm

Clustered table



https://www.percona.com/blog/2013/07/26/checking-btree-leaf-nodes-list-consistency-in-innodb/

The table is the clustered index

The row order of the table is physically enforced

Each table has only one clustered index

InnoDB requires one clustered index:

- **If there is a PRIMARY KEY on a table, it will be used as the clustered index**
- **If NO PK, InnoDB uses the first NON-NULL UNIQUE index in the table**
- **If NO PK or UK, InnoDB generates a hidden clustered index named GEN_CLUST_INDEX**

InnoDB uses B+ tree data structure for the clustered index

ALL secondary indexes contains the PK columns for the row

Small PKs are more performant that larger PKs in InnoDB

https://memegenerator.net/instance/54486906/jack-skellington-nightmare-before-christmas-making-indexes-making-indexes-its-sql-time

Primary Key

PKs are by definition:

- **Unique**
- **Not null**
- **The identifier of the row (data)**

InnoDB

- **Clustered Index**
- **The table data and the the PK index are stored in the same data structure**
- **The PK gives the table order**

Postgres

- **Non-Clustered Index**
- **There is no really PK index in Postgres, all index are basically the same**
- **The table data and index data are stored in separate storage structures**
- **Maintains a pointer to corresponding heap data**

Secondary index

Can hold an unique restriction or not (having duplicates)

Can be used for Index Only Scan (covering index)

InnoDB

- Does not impact physical storage location
- Only stores the key of the secondary index and the clustered key in the leaf node
- Has to traverse the index to find the PK, then traverse the PK to retrieve the table info

Postgres

- All indexes are basically the same (all indexes in Postgres are secondary indexes)
- Maintains a pointer to corresponding heap data

Multi-column (composite) index

An index can be defined on more than one column of a table

Column order matters

Can use a partial index

MySQL

- **Can use up to 32 columns**
- **Usually better than multiple single column indexes (index_merge is inefficient)**

Postgres

- **Can use up to 32 columns**
- **Cannot use parallel scan in a single index**
- **Multiple single indexes with parallel scan with a bitmap index may be faster**

Index type

- **We will not cover all data structures here and some mentions are for reference only;**
- **There are other index types than the one mentioned here, it's not an inclusive list;**
- **Index types and implementations varies from database to database, for example:**
- **MySQL and Postgres:**
  - B-tree index
  - Hash index
- **Postgres:**
  - Inverted index (GIN)
  - Block Range index (BRIN)
  - Generalized Search Tree index (GiST)
  - Space partitioned GiST (SP-GiST)

Access methods

Table scan

InnoDB

- **Needs to traverse the PRIMARY index (key order, physical order is not supported)**
- **Index pages are not necessarily ordered physically and logically the same way**
- **It is random, rather than sequential, page reads**
- **It’s usually expensive**

Postgres

- **Simple scan the whole heap table, row by row, in its physical sequential order**
- **It is sequential, rather than random, page reads**
- **Sequential scans can be cheaper than index access, especially on small tables**

Index scan

Can only be traversed in key order, not in physical order

Can be an unique or a range scan

MySQL:

- **Traverse the index to find the PRIMARY value**
- **Traverse the PRIMARY index to retrieve the table info**

Postgres:

- **It's a tree traversal followed by processing pages indicated in the tuple ID**
- **Points at the tuple ID which stores the page number and row number within a page**

Index scan

There are different optimizations

MySQL

- **Range Optimization**
- **Index Merge Optimization**
- **ORDER BY Optimization**
- **Index Condition Pushdown Optimization**

Postgres

- **ORDER BY Optimization**
- **Parallel Index Scan**
- **Bitmap Index Scan**

Index only scan

It is a secondary (non-clustered) index

Has all columns needed from a table in the query

All data is retrieved from the index avoiding table access

Available in MySQL since early versions (tested in 5.5)

Later adopted in Postgres (since version 9.6)

Helping the database

PERCONA

Techniques to reduce table access

Good statistics

Covering index

Partial index

Expression indexes

Late row lookup

… … …

Summary

PERCONA

Summary

- InnoDB uses a clustered index data structure to store its data
- Postgres stores its data in heap files
- Table scan is usually more expensive in InnoDB due to random access
- Postgres can physically traverse the table reducing random access
- All indexes in Postgres are secondary, PK is more a formal distinction
- InnoDB needs to traverse the secondary to find the PK and then retrieve the row values
- Indexes speed up performance reducing table access
- We can help the database to reduce table access with some SQL tricks

PERCONA

Thanks!

Any questions?

linkedin.com/in/charlybatista

charly.batista@percona.com

https://github.com/elchinoo/webinar