Postgres Indexes

Zach Croft

Postgres Internals Resource: https://www.interdb.jp/pg/index.html

Index: What, Why?

```
postgres=# SELECT relname, oid, relfilenode FROM pg_class WHERE relname = 'idx_grades_id_covering';
        relname         |  oid  | relfilenode
------------------------+-------+-------------
 idx_grades_id_covering | 24594 |       24594
(1 row)
```

```
root@9c50520968ae:/# ls -la $PGDATA/base/* | grep 24594
-rw------- 1 postgres postgres 112336896 Jun 12 01:32 24594
```

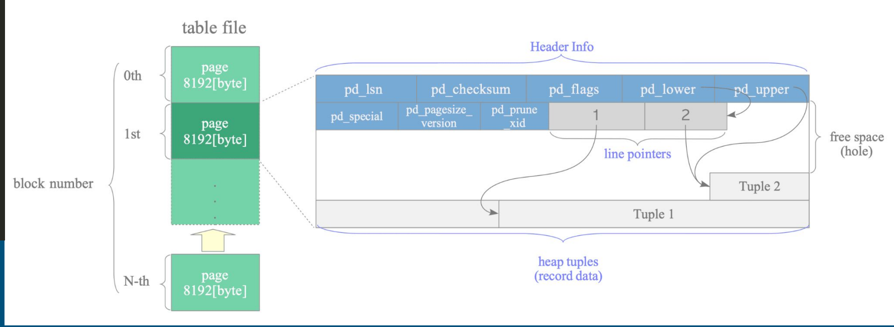
- Data structure to enhance database performance.
    - analogous to using binary search instead of linear search on a sorted list, or categorizing library books by sections but not one-by-one.
- Supported index types: B-tree, Hash, GiST, SP-GiST, GIN, BRIN, and bloom.
    - PG primary and secondary keys use indexes and default to B-tree.

Heap and Pages

```c
typedef struct PageHeaderData @src/include/storage/bufpage.h
{
  /* XXX LSN is member of *any* block, not only page-organized ones */
  PageXLogRecPtr       pd_lsn;      /* LSN: next byte after last byte of xlog
                                     * record for last change to this page */
  uint16          pd_checksum; /* checksum */
  uint16          pd_flags;    /* flag bits, see below */
  LocationIndex   pd_lower;    /* offset to start of free space */
  LocationIndex   pd_upper;    /* offset to end of free space */
  LocationIndex   pd_special;  /* offset to start of special space */
  uint16          pd_pagesize_version;
  TransactionId   pd_prune_xid;/* oldest prunable XID, or zero if none */
  ItemIdData      pd_linp[1];  /* beginning of line pointer array */
} PageHeaderData;

typedef PageHeaderData *PageHeader;

typedef uint64 XLogRecPtr;
```

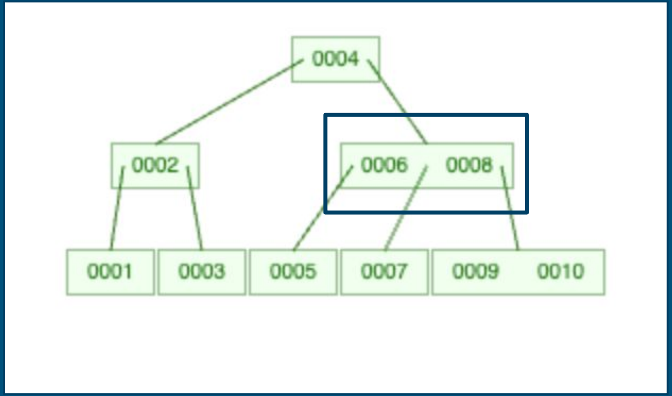


- Data files (table, index, etc.) are divided into pages (or blocks) of a fixed default length of 8KB, each identified with a block number.
- Tables are organized in a ‘heap’ structure (this is distinct from the heap tree structure AND the heap related to memory allocation).
- Heap tables contain ‘pages’ (this is also distinct from the page related to memory allocation, those are typically 4KB) with header metadata, line pointer(s), free space, and heap tuple(s). A DB page could span 3 OS pages.
  - Line pointers grow ‘up’ and heap tuples grow ‘down’ similar to heap and stack in memory.
- Heap tuples store the actual records and are identified by a **tuple identifier (TID)** built from the block number and the offset number (line pointer ID).
  - Line pointers are essentially mini-indexes within a page. Don’t confuse this with a PostgreSQL index.

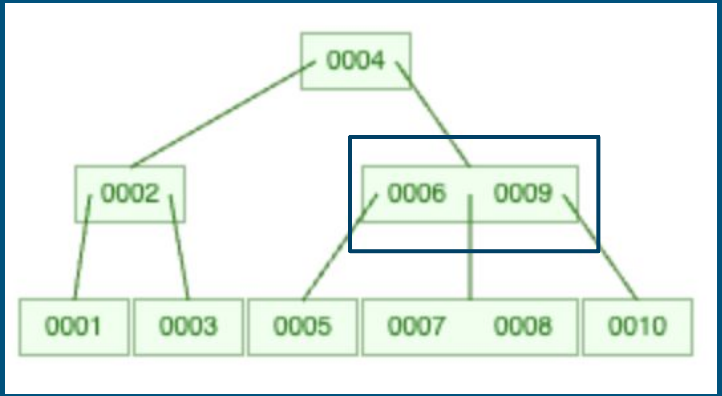
B-tree (Original)

Note: these images only show the **key.**

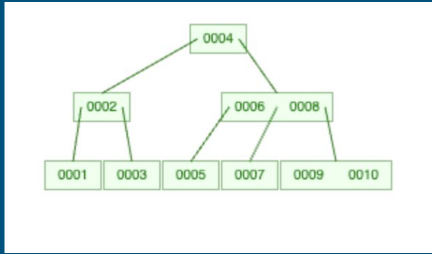
- B-trees (bee trees) are balanced trees with pages as nodes.
  - Each node contains elements, which are key-value pairs from key to TID.
    - Postgres does not actually map to the primary key, which is not universal between different DB engines.
  - If a node has m child nodes, then it can have at most m-1 elements.



- (left) I added sequentially, (right) added in a different order.
  - Ordering of the tree is dependent on input order.



B-tree Limitations



Storage

- All nodes (root, internal, leaf) have key-value pair elements (we will see, not necessary), taking more storage, therefore taller trees, and therefore more IO to disk (fetches more OS pages).
  - Indexes may not entirely fit in memory if they are large–the database might have to use the swap which is <u>slow</u>.

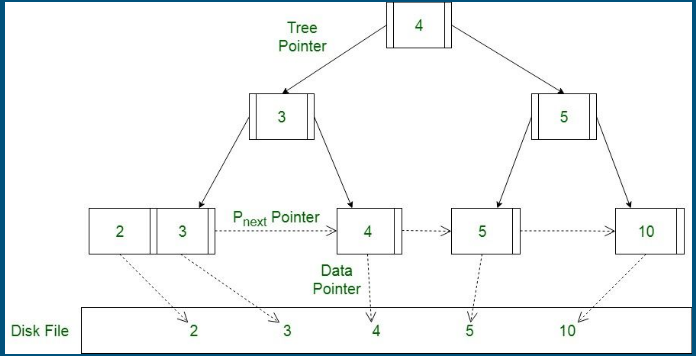
Performance: Range Query

- Consider the above b-tree on id:
  - SELECT id FROM grades WHERE id >= 1 AND id < 8;
  - To get to id=1, fetches 3 pages.
    - id=2 will probably be cached unless moved into swap.
  - id=4 is probably cached.
  - Id=5, fetches 2 pages.
  - id=6 is probably cached.
    - We got id=8 whether we wanted it or not–page IO fetches everything in it.

B+tree (Fancy)

Consequent nodes are doubly-linked which makes range queries much easier.

- B+trees are balanced trees with pages as nodes, but:
  - Internal nodes (everything except leaf nodes), are just keys.
    - Nodes are pages; height is reduced and therefore traversal times are reduced.
- Some keys are duplicated, and searches always go to the leaf nodes.
- Postgres often operates by keeping all internal nodes in memory and then leaving the leaf nodes in the heap for quicker traversal.



Postgres really uses B+trees (modified) in all instances that there is a B-tree. B+trees are not always better as there are trade-offs, and also other indexes might be a better fit for situations.

How Do Indexes Perform?

```
Indexes:
    "idx_grades_id_covering" btree (id) INCLUDE (grade)
```

```
postgres=# \d grades
                              Table "public.grades"
 Column |          Type          | Collation | Nullable |
--------+------------------------+-----------+----------+
 id     | integer                |           | not null |
 grade  | integer                |           |          |
 name   | character varying(255) |           | not null |
Indexes:
    "grades_pkey" PRIMARY KEY, btree (id)

postgres=# SELECT COUNT(*) from grades;
  count
---------
 5000000
(1 row)
```

```
 Seq Scan on grades  (cost=0.00..102028.00 rows=2970027
   Filter: ((grade >= 0) AND (grade < 60))
   Rows Removed by Filter: 2026244
 Planning Time: 0.315 ms
 JIT:
   Functions: 4
   Options: Inlining false, Optimization false, Expressi
   Timing: Generation 0.504 ms (Deform 0.137 ms), Inlini
 Execution Time: 334.864 ms
(9 rows)

Time: 337.576 ms
```

```
postgres=# SELECT id
postgres=# ,grade
postgres=# FROM grades
postgres=# WHERE id < 1500 AND grade > 75;

---------------------------------------------
 Index Only Scan using idx_grades_id_coverin
   Index Cond: (id < 1500)
   Filter: (grade > 75)
   Rows Removed by Filter: 1144
```

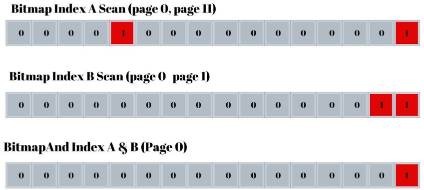


Table Scan

- Generally, the slowest way queries can be made.
  - DB engines often use multiple workers for speed.
  - Buffer system and cache layers help.

Index Scan

- A typical index scan where a key-value pair is found and then the corresponding heap table page is fetched.
- SELECT * and SELECT name are same (row-based).

Index-Only Scan

- Never jump to the heap!
- All requested values are **in** the index file.
- Composite indexes (including other values in your index) work well here.

Bitmap Index Scan

- Works with bitmap heap scans to create a bitmap for heap pages to include.
- BitmapAnd takes the intersection of multiple bitmaps.

Shopify

"In one high-throughput system at Shopify we've seen a 50 percent decrease in INSERT statement duration by switching from UUIDv4 to ULID for idempotency keys."

Line Pointers (mini-indexes) And Clustering

If I don't really mind how long insertion takes, can I just sort the heap table pages and heap tuples on insertion and skip the IO's from the index? Index-organized tables… are not in Postgres. But you can cluster periodically.

Demo!

Final Thoughts And Discussion

- Indexes are generally useful but can slow down smaller queries, the planner takes note of this and may skip indexes.
- Indexes help most on massive tables, but horizontal partitioning and/or sharding can help by splitting massive tables into smaller tables.
- How do indexes work with locking and transactions?