PostgreSQL-Consulting

data egret

nbtree-indexes in PostgreSQL. Useful novelties.

2019

Victor Yegorov
vyegorov@dataegret.com

Introduction

- With PostrgeSQL since 1998 / 6.5
- PHP and C developer, SQL
- ORACLE and PostgreSQL administration, Linux and HP-UX
- Telecommuncation billing, card processing systems, web projects
- Currently PostrgeSQL DBA

What we will speak about?

1. Reasons for changing nbtree indexes
2. nbtree internals
3. New changes and how they perform
4. Other novelties

1. Reasons for changing nbtree indexes

- 2016, July
- Uber unveils its switch to MySQL

- A lot of discussions in the Community:
- Why we lost Uber as a user
- On Uber’s Choice of Databases

1. Why Uber switched away?

1. UPDATE-s on central table with lot’s of indexes
2. Absence of secondary indexes
3. Index re-balancing
4. Binary replication

1. UPDATE-s on a table

- Any UPDATE modifies all table indexes
- IO increases with the number of table’s indexes
- Heap-Only Tuples (HOT) optimization possible:
  - Enough space for new tuple in the heap page, and
  - No indexed columns are updated
- Sometimes it’s better to use explicit Sort, but allow HOT optimization

1. Suggested improvements

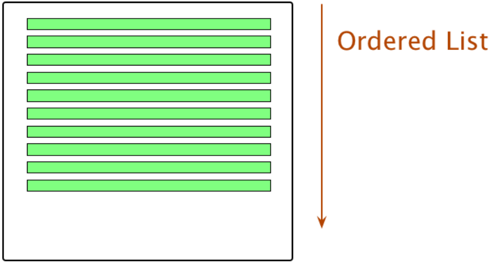
- Write Amplification Reduction Method (WARM)
- Table AM
  - zheap
  - zedstore
- Retail Index Deletion

2. nbtree internals

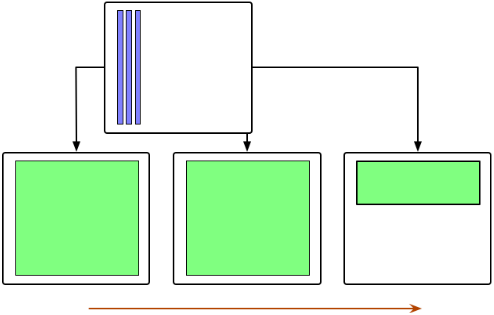
- src/backend/access/nbtree
- README contains lots of useful details
  github shows it straight away ;)

2. nbtree internals

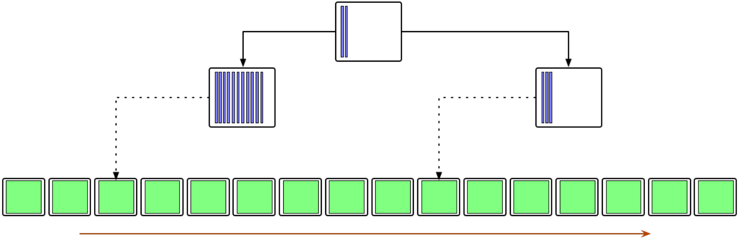
- src/backend/access/nbtree
- README contains lots of useful details
- Index is filled via Leaf pages

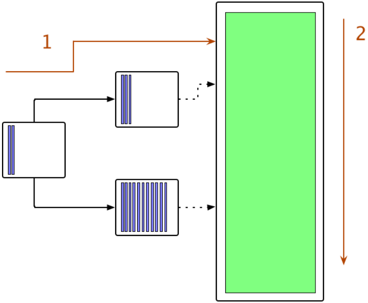


- Index is filled via Leaf pages
- Leaf pages are covered with a Tree
  (as soon as there're 2 or more Leafs)



- Tree starts with a Root page
- It grows via Intermediate pages
- All Tree entries point either to Intermediate, or to Leaf pages





1. Tree traversal
   Unique Scan

2. Walk along the Ordered List
   Range Scan

2. Working with index

```
SELECT * FROM order WHERE customer_id=1471
   AND order_dt>='2019-01-15';
```

(customer_id, order_dt)

```
1014;  2019-01-14
1014;  2019-01-28
1014;  2019-02-02
1201;  2019-01-19
1201;  2019-01-29
1201;  2019-03-06
1201;  2019-05-06
1471;  2019-03-05
1471;  2019-03-09
1471;  2019-04-04
1471;  2019-05-12
2012;  2019-01-15
2012;  2019-02-15
2012;  2019-03-15
2012;  2019-04-15
```

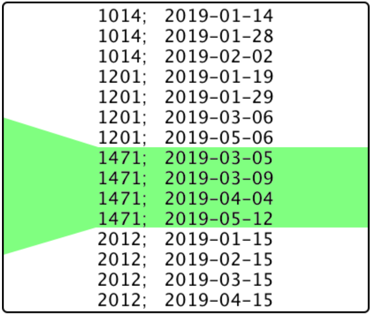
(order_dt, customer_id)

```
2019-01-14;  1014
2019-01-15;  2012
2019-01-19;  1201
2019-01-28;  1014
2019-01-29;  1201
2019-02-02;  1014
2019-02-15;  2012
2019-03-05;  1471
2019-03-06;  1201
2019-03-09;  1471
2019-03-15;  2012
2019-04-04;  1471
2019-04-15;  2012
2019-05-06;  1201
2019-05-12;  1471
```

2. Working with index

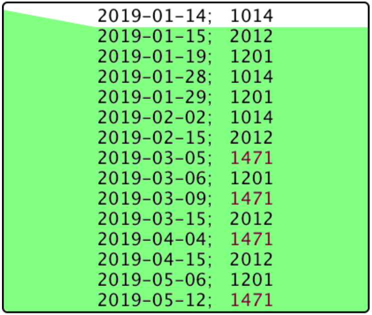
```
SELECT * FROM order WHERE customer_id=1471
   AND order_dt>='2019-01-15';
```

(customer_id, order_dt)



```
1014;  2019-01-14
1014;  2019-01-28
1014;  2019-02-02
1201;  2019-01-19
1201;  2019-01-29
1201;  2019-03-06
1201;  2019-05-06
1471;  2019-03-05
1471;  2019-03-09
1471;  2019-04-04
1471;  2019-05-12
2012;  2019-01-15
2012;  2019-02-15
2012;  2019-03-15
2012;  2019-04-15
```

(order_dt, customer_id)



```
2019-01-14;  1014
2019-01-15;  2012
2019-01-19;  1201
2019-01-28;  1014
2019-01-29;  1201
2019-02-02;  1014
2019-02-15;  2012
2019-03-05;  1471
2019-03-06;  1201
2019-03-09;  1471
2019-03-15;  2012
2019-04-04;  1471
2019-04-15;  2012
2019-05-06;  1201
2019-05-12;  1471
```

2. Analyzing index

```
CREATE EXTENSION pageinspect;

CREATE TABLE tb (id int, val bool);
CREATE INDEX i_tb_val ON tb(val);
INSERT INTO tb SELECT gs,true
  FROM generate_series(1,100000) gs;

SELECT * FROM bt_metap('i_tb_val');
```

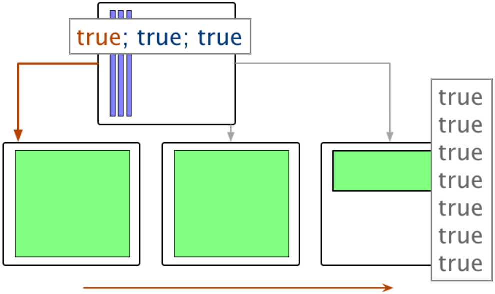
2. Analyzing index

```
SELECT * FROM bt_metap('i_tb_val');
magic  version root level fastroot fastlevel oldest_xact last_cleanup_num_tuples
------ ------- ---- ----- -------- --------- ----------- -----------------------
340322       3    3     1        3         1           0                      -1
```

```
WITH p AS (
    SELECT blkno FROM pg_class ic, generate_series(1,relpages-1) s(blkno)
     WHERE oid='i_tb_val'::regclass
)
SELECT sum(s.page_size) ttl, sum(s.free_size) free, count(*),
  round(sum(s.free_size)*100.0/sum(s.page_size),2) free_pct
  FROM p, bt_page_stats('i_tb_val',blkno) s
 WHERE s.type = 'l';

  ttl    free  count free_pct
------- ------ ----- --------
2940928 877972   359    29.85
```

- Leaf pages contain duplicated entries
- Tree is not working:
  - Traversal get's one into the beginning of Ordered List
  - Linear ordered scan follows



3. Index structure

Lehman and Yao algorithm defines, that:

- for the set of keys of any subtree S
- the following holds true:

$$K_i < v <= K_{i+1}$$

- where K_i and K_{i+1} are adjacent keys on the upper level
- and all v are unique

3. Index structure

- In PostgreSQL this used to be different
- Insertions where done at the end of list of duplicates in order to avoid splits: $O(N^2)$
- “Getting tired” optimization: in 1% of insertions avoid search for the end of list, insert at the current page, splitting if necessary

Downsides of the old approach:

- Suboptimal space usgae on Leaf pages
- Impossible to find index key for the heap tuple
- Index entries’ cleanup is delayed till VACUUM
- Accounts for index bloat
- Reindexing required to maintain index health

3. Index structure changes

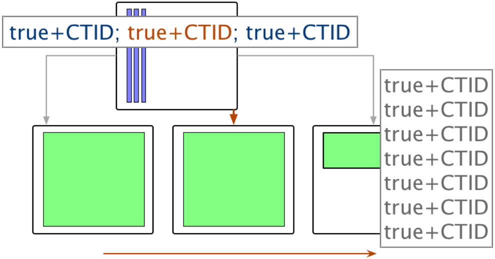
- CTID as a tie-breaker

  Use CTID as part of the index key
- Suffix truncation

  Remove attributes at the end of the key, if uniqueness holds true without
- Page split heuristics

  Pick split points to maximize suffix truncation

- All keys are unique due to CTID
- Tree is working properly now



true+CTID; true+CTID; true+CTID

true+CTID
true+CTID
true+CTID
true+CTID
true+CTID
true+CTID
true+CTID

3. Analyzing index

```
WITH p AS (
    SELECT blkno FROM pg_class ic, generate_series(1,relpages-1) s(blkno)
     WHERE oid='i_tb_val'::regclass
)
SELECT sum(s.page_size) ttl, sum(s.free_size) free, count(*),
  round(sum(s.free_size)*100.0/sum(s.page_size),2) free_pct
  FROM p, bt_page_stats('i_tb_val',blkno) s
 WHERE s.type = 'l';
```

```
  ttl    free  count free_pct
------- ----- ----- --------
2105344 86868   257     4.13
```

3. Index structure changes

- Leaf contents remains the same, but all keys are unique now
- Tree keys are bigger now (due to CTID), Tree “should” grow faster
- Tree grows is balanced with higher density due to Suffix truncation:
  - old index occupied 359 pages
  - new index uses only 257 pages
- Index state after population is almost identical to the one after REINDEX

3. Multicolumn index

```
CREATE TABLE tab (
    id int        NOT NULL,
    dt timestamp  NOT NULL,
    a  char(10)   NOT NULL DEFAULT ’AAAAAAAAAA’,
    b  char(10)   NOT NULL DEFAULT ’BBBBBBBBBB’,
    c  char(10)   NOT NULL DEFAULT ’CCCCCCCCCC’,
    d  char(10)   NOT NULL DEFAULT ’DDDDDDDDDD’,
    e  char(10)   NOT NULL DEFAULT ’EEEEEEEEEE’
);
CREATE INDEX i_tab_multi ON tab(a,b,c,d,e);
INSERT INTO tab
SELECT gs,
       DATE ’2019-10-01’ + (INTERVAL ’1sec’ *
      ((random()*31*24*3600)::int)) AS dt
  FROM generate_series(1,1000000) gs;
```

3. Multicolumn index

```
WITH p AS (
  SELECT blkno FROM pg_class ic, generate_series(1,relpages-1) s(blkno)
   WHERE oid='i_tab_multi'::regclass)
SELECT s.type, sum(s.page_size) ttl, sum(s.free_size) free, count(*) cnt,
       round(sum(s.free_size)*100.0/sum(s.page_size),2) free_pct
  FROM p, bt_page_stats('i_tab_multi',blkno) s GROUP BY s.type;

type   ttl      free    cnt  free_pct
---- -------- -------- ----- --------
r        8192     8000     1    97.66
i     1409024   662816   172    47.04
l    87367680 18173268 10665    20.80

REINDEX INDEX i_tab_multi;

type   ttl     free   cnt  free_pct
---- -------- ------- ---- --------
r        8192    8068    1    98.49
i      933888  284496  114    30.46
l    76562432 7515748 9346     9.82
```

3. Multicolumn index

```
            INITIAL                REINDEX                  v12

type     cnt  free_pct          cnt  free_pct          cnt  free_pct
----    ----- --------          ---- --------          ---- --------
r           1    97.66             1    98.49             1    98.39
i         172    47.04           114    30.46           119    30.03
l       10665    20.80          9346     9.82          8772     3.91
```

4. Further improvements

- Currently suffixes are truncated on the attribute boundaries, it is possible to use substrings
- “Classic” nbtree suffix truncation prototype
- Remove index keys on DELETE and during microvacuum
- Use new infrastructure for the global indexes on partitioned tables *(speculation)*

4. Further improvements

- Currently suffixes are truncated on the attribute boundaries, it is possible to use substrings
- “Classic” nbtree suffix truncation prototype
- Remove index keys on DELETE and during microvacuum
- Use new infrastructure for the global indexes on partitioned tables *(speculation)*

- Effective storage of duplicates in B-tree index
  not quite related, but very promising!

4. Availability

- New index structure comes as Version 4
- All new indexes are built with Version 4
- Old indexes (pg_upgraded ones) must be rebuilt

- REINDEX CONCURRENTLY !!!
- Report progress of CREATE INDEX and REINDEX operations
- Support for INCLUDE attributes in GiST indexes
- Less WAL during GiST, GIN and SP-GiST index build.
- Allow VACUUM to be run with index cleanup disabled.

Thank you!

Victor Yegorov
vyegorov@dataegret.com