B-tree - explore the heart of PostgreSQL

Anastasia Lubennikova

www.postgrespro.ru

Objectives

- Inspect internals of B-tree index
- Present new features
- Understand difficulties of development
- Clarify our roadmap

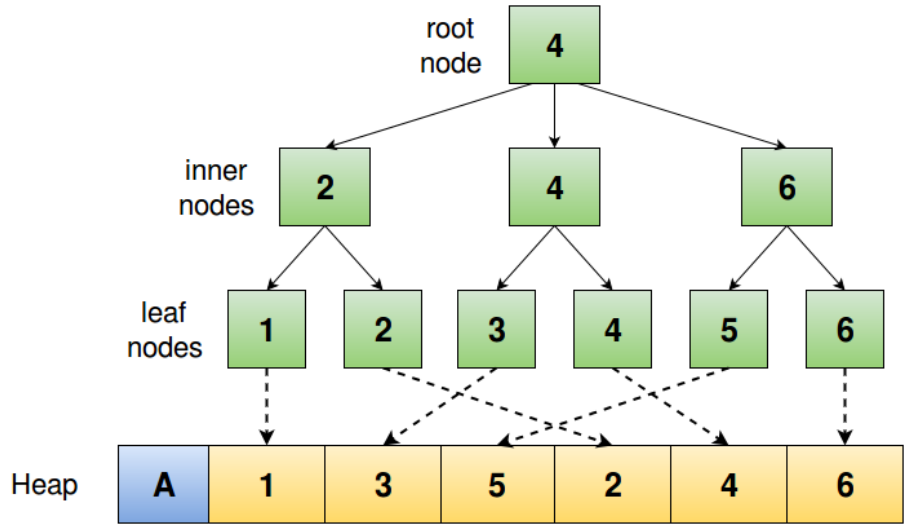
http://goo.gl/TkrHJa

Important notes about indexes

- All indexes are secondary
- Index files are divided into standard-size pages
  - Page layout is defined by Access Method
- Indexes store TIDs of heap tuples
- There is no visibility information in indexes
  - But we have visibility map and LP_DEAD flag
- VACUUM removes dead tuples
- Update = insert
  - except HOT-updates

B+ tree

- Tree is balanced
- Root node and inner nodes contain keys and pointers to lower level nodes
- Leaf nodes contain keys and pointers to the heap
- All keys are sorted inside the node



Lehman & Yao Algorithm

- ✔ Right-link pointer to the page's right sibling
- ✔ "High key" - upper bound on the keys that are allowed on that page
- ✔ Assume that we can fit at least three items per page

- ✗ Assume that all btree keys are unique
- ✗ Assume fixed size keys
- ✗ Assume that in-memory copies of tree pages are unshared

PostgreSQL B-tree implementation

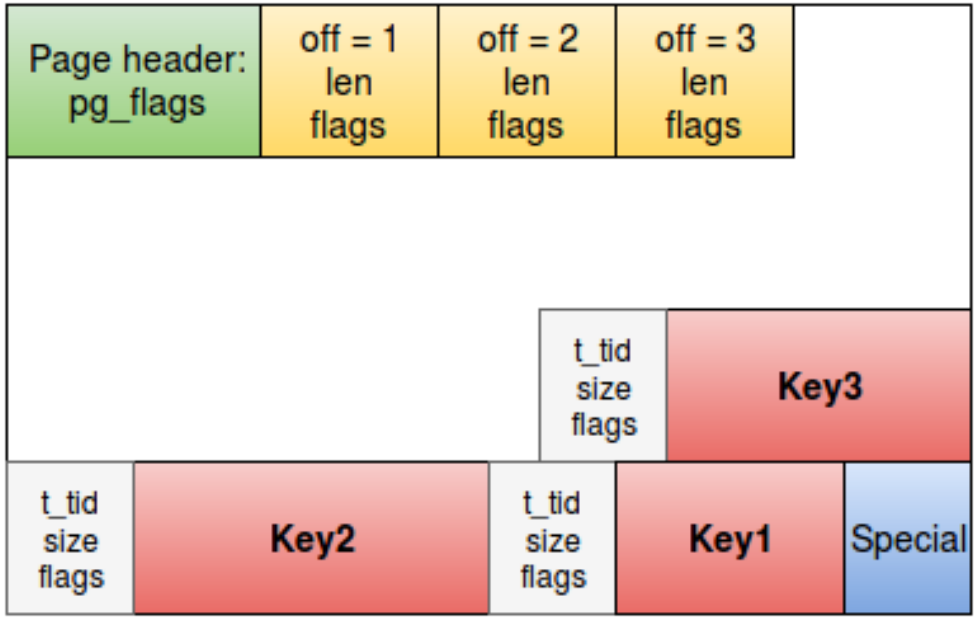
- Left-link pointer to the page's left sibling
  - For backward scan purposes
- Nonunique keys
  - Use link field to check tuples equality
  - Search must descend to the left subtree
  - Tuples in b-tree are not ordered by TID
- Variable-size keys
  - ```
    #define MaxIndexTuplesPerPage \
    ((int) ((BLCKSZ - SizeOfPageHeaderData) / \
    (MAXALIGN(sizeof(IndexTupleData) + 1) + sizeof(ItemIdData))))
    ```
- Pages are shared
  - Page-level read locking is required
    ```

Meta page

- Page zero of every btree is a meta-data page

```
typedef struct BTMetaPageData
{
    uint32      btm_magic;      /* should contain BTREE_MAGIC */
    uint32      btm_version;    /* should contain BTREE_VERSION */
    BlockNumber btm_root;       /* current root location */
    uint32      btm_level;      /* tree level of the root page */
    BlockNumber btm_fastroot; /* current "fast" root location */
    uint32      btm_fastlevel; /* tree level of the "fast" root page */
} BTMetaPageData;
```

Page layout




```
typedef struct BTPageOpaqueData
{
    BlockNumber btpo_prev;
    BlockNumber btpo_next;
    union
    {
        uint32      level;
        TransactionId xact;
    }           btpo;
    uint16      btpo_flags;
    BTCycleId   btpo_cycleid;
} BTPageOpaqueData;
```

Unique and Primary Key B-tree index

- 6.0 Add UNIQUE index capability (Dan McGuirk)

```
CREATE UNIQUE INDEX ON tbl (a);
```

- 6.3 Implement SQL92 PRIMARY KEY and UNIQUE clauses using indexes (Thomas Lockhart)

```
CREATE TABLE tbl (int a, PRIMARY KEY(a));
```

System and TOAST indexes

- System and TOAST indexes

```
postgres=# select count(*) from pg_indexes where
           schemaname='pg_catalog';
 count
-------
   103
```

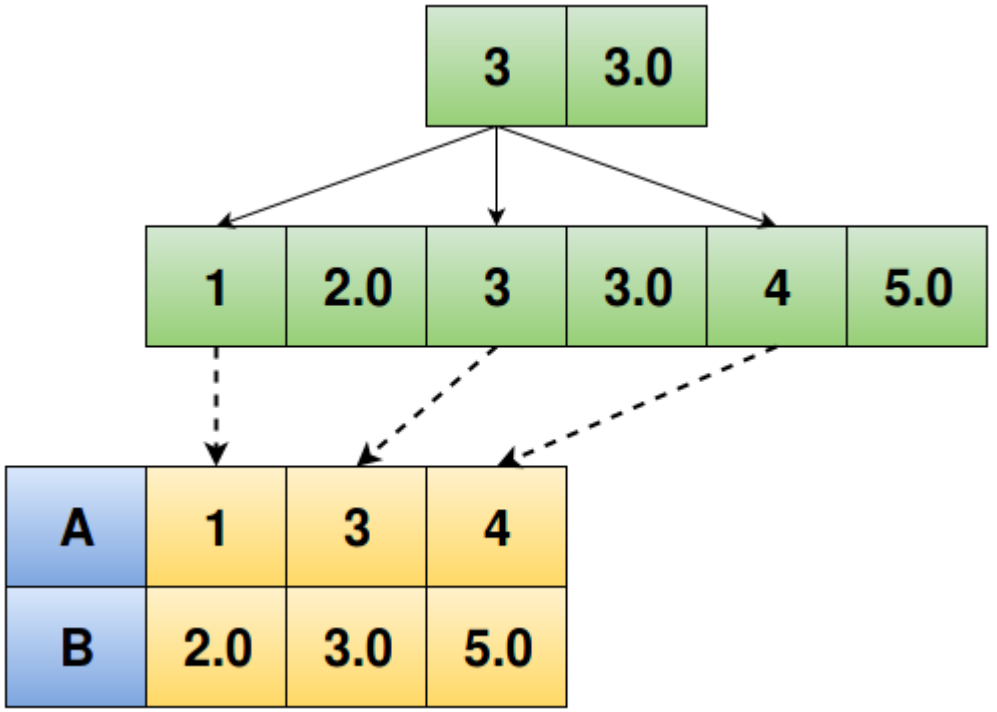
Fast index build

- Uses tuplesort.c to sort given tuples
- Loads tuples into leaf-level pages
- Builds index from the bottom up
- There is a trick for support of high-key

Multicolumn B-tree index

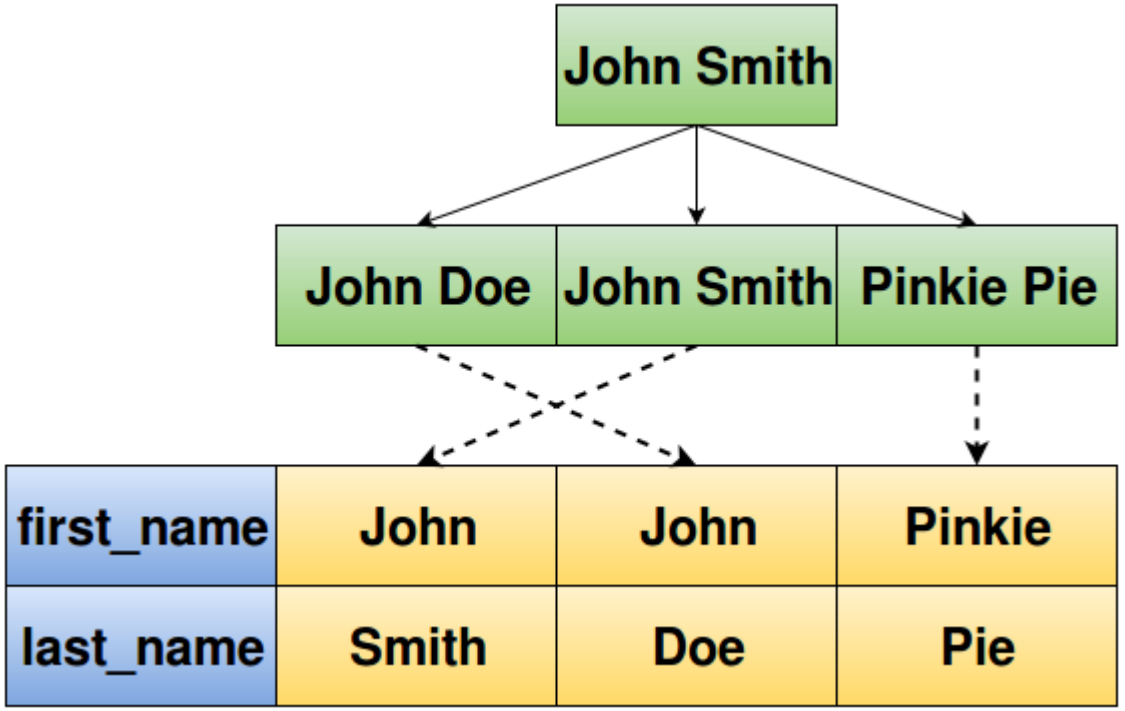
- 6.1 multicolumn btree indexes (Vadim Mikheev)

```
CREATE INDEX ON tbl (a, b);
```



Expressional B-tree index

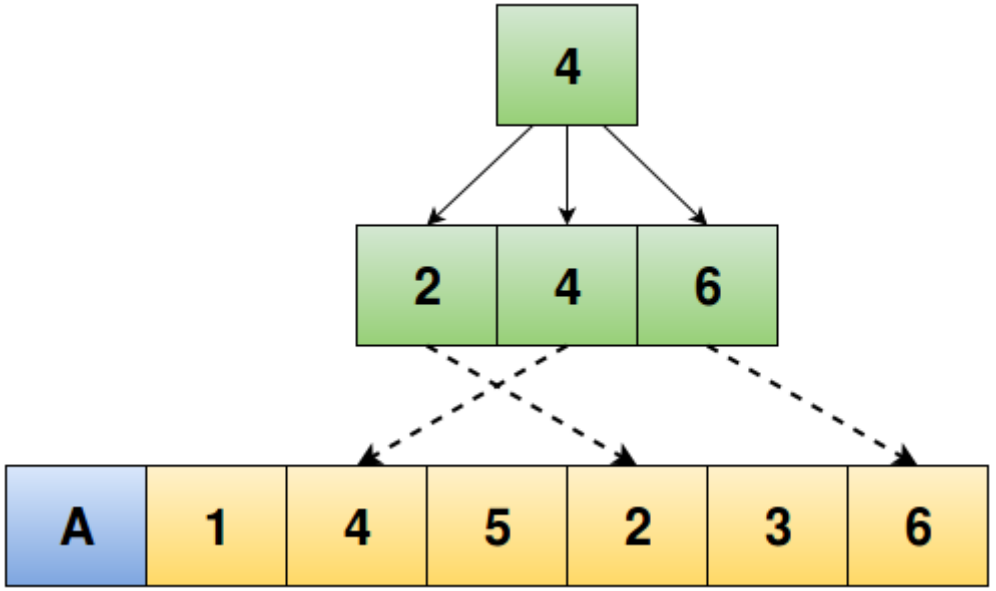
```
CREATE INDEX people_names
ON people ((first_name || ' ' || last_name));
```



Partial B-tree index

- 7.2 Enable partial indexes (Martijn van Oosterhout)

```
CREATE INDEX ON tbl (a) WHERE a%2 = 0;
```

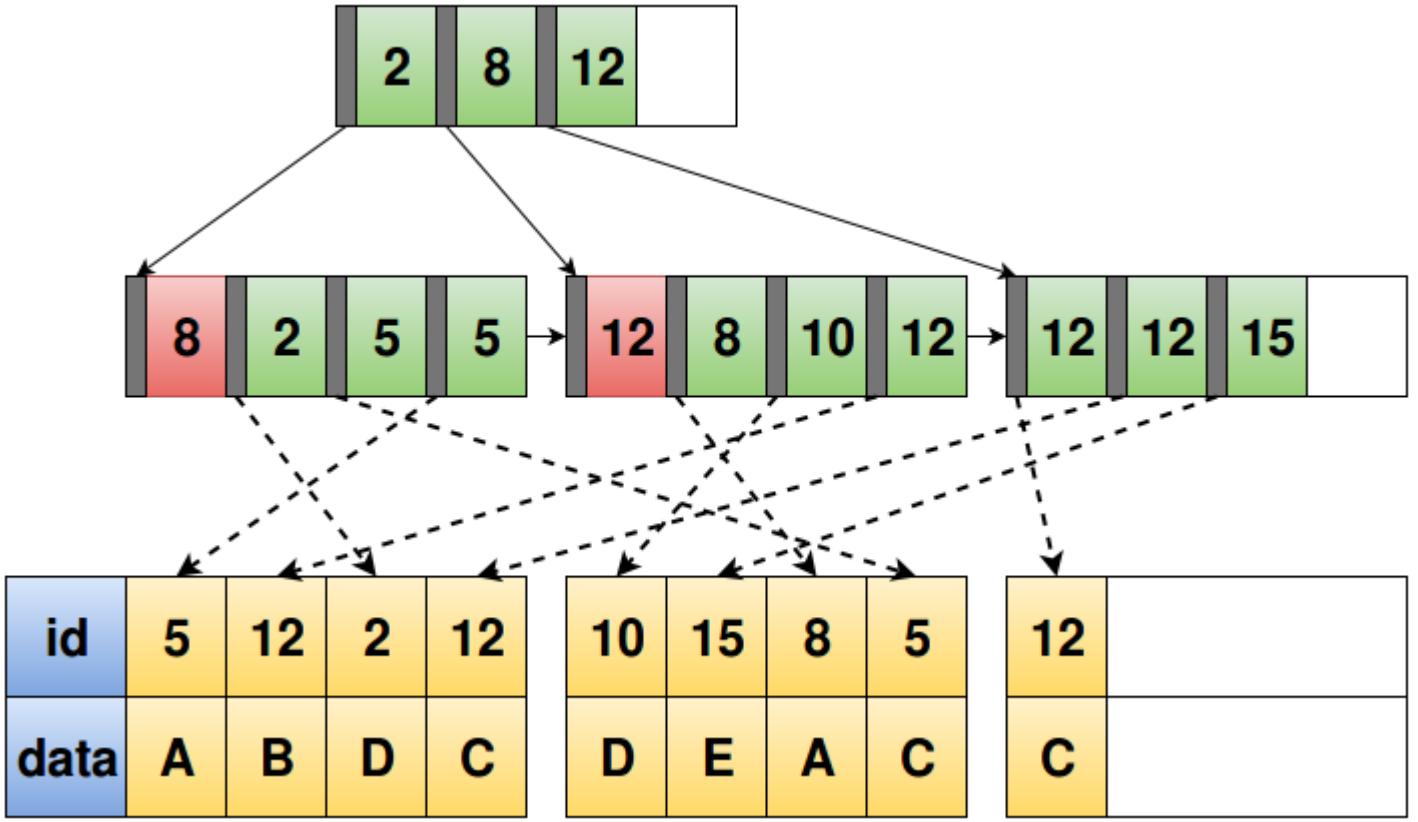


On-the-fly deletion (microvacuum)

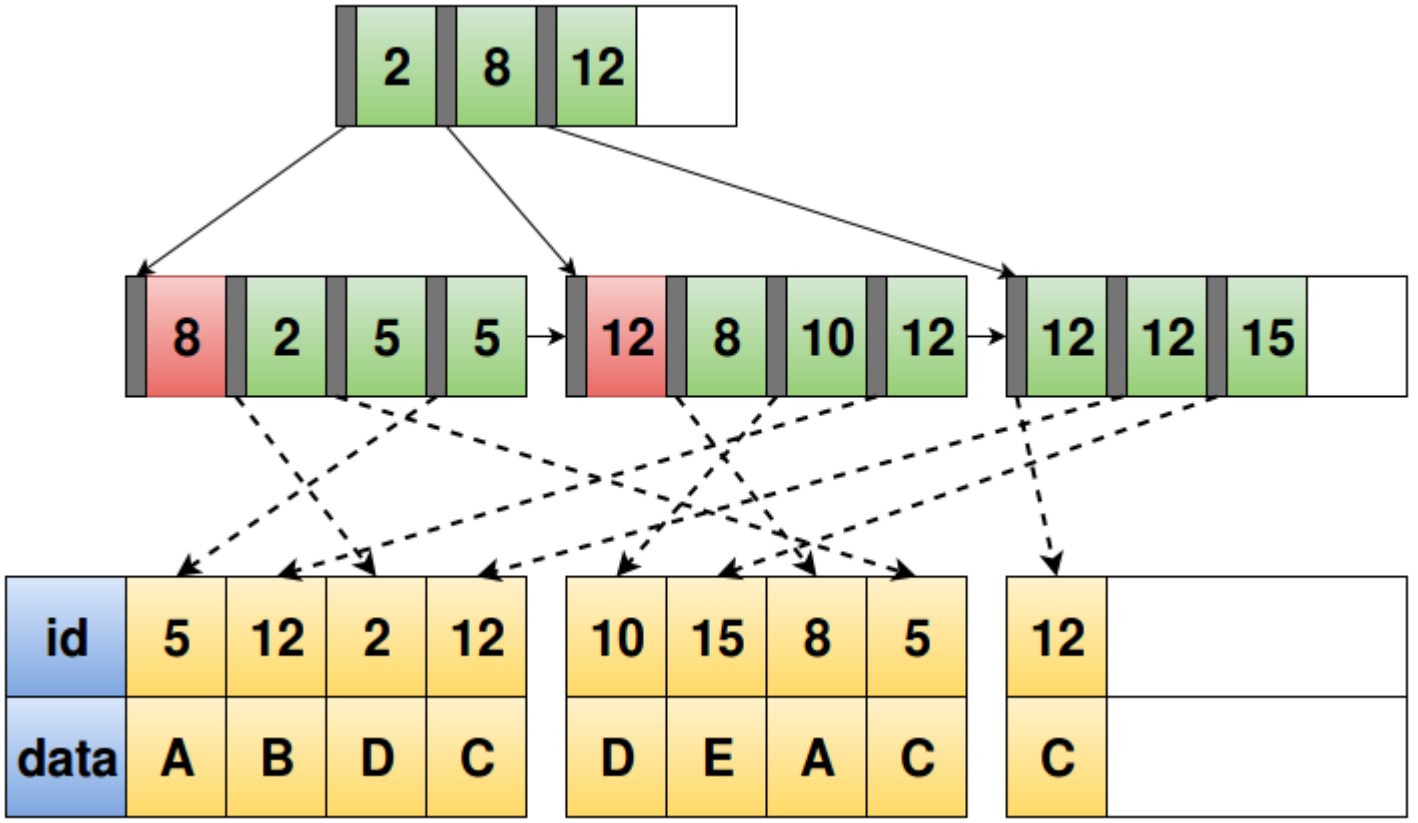
- 8.2 Remove dead index entries before B-Tree page split (Junji Teramoto)
- Also known as **microvacuum**
- Mark tuples as dead when reading table.
- Then remove them while performing insertion before the page split.

HOT-updates

- 8.3 Heap-Only Tuples (HOT) accelerate space reuse for most UPDATEs and DELETEs (Pavan Deolasee, with ideas from many others)

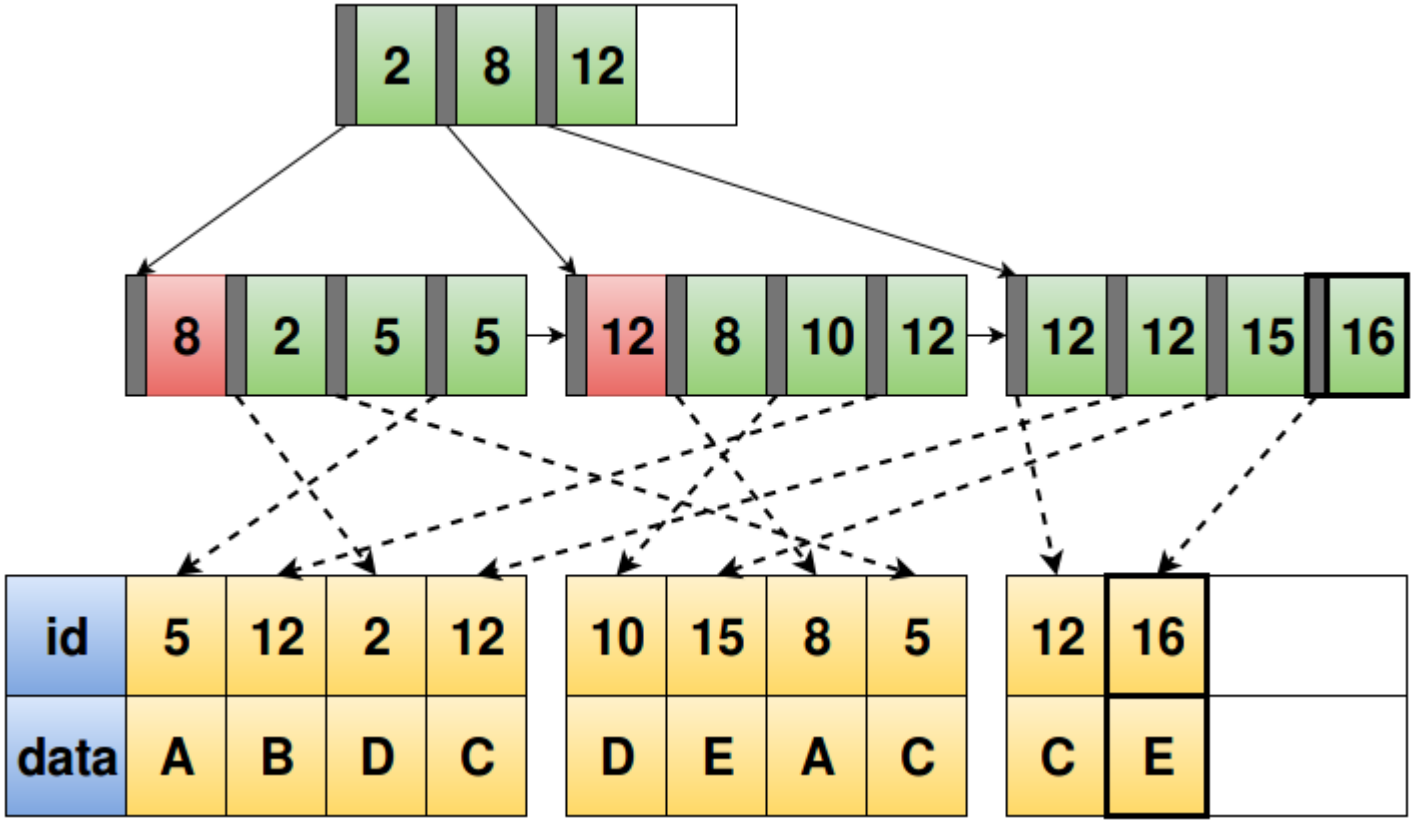


HOT-updates



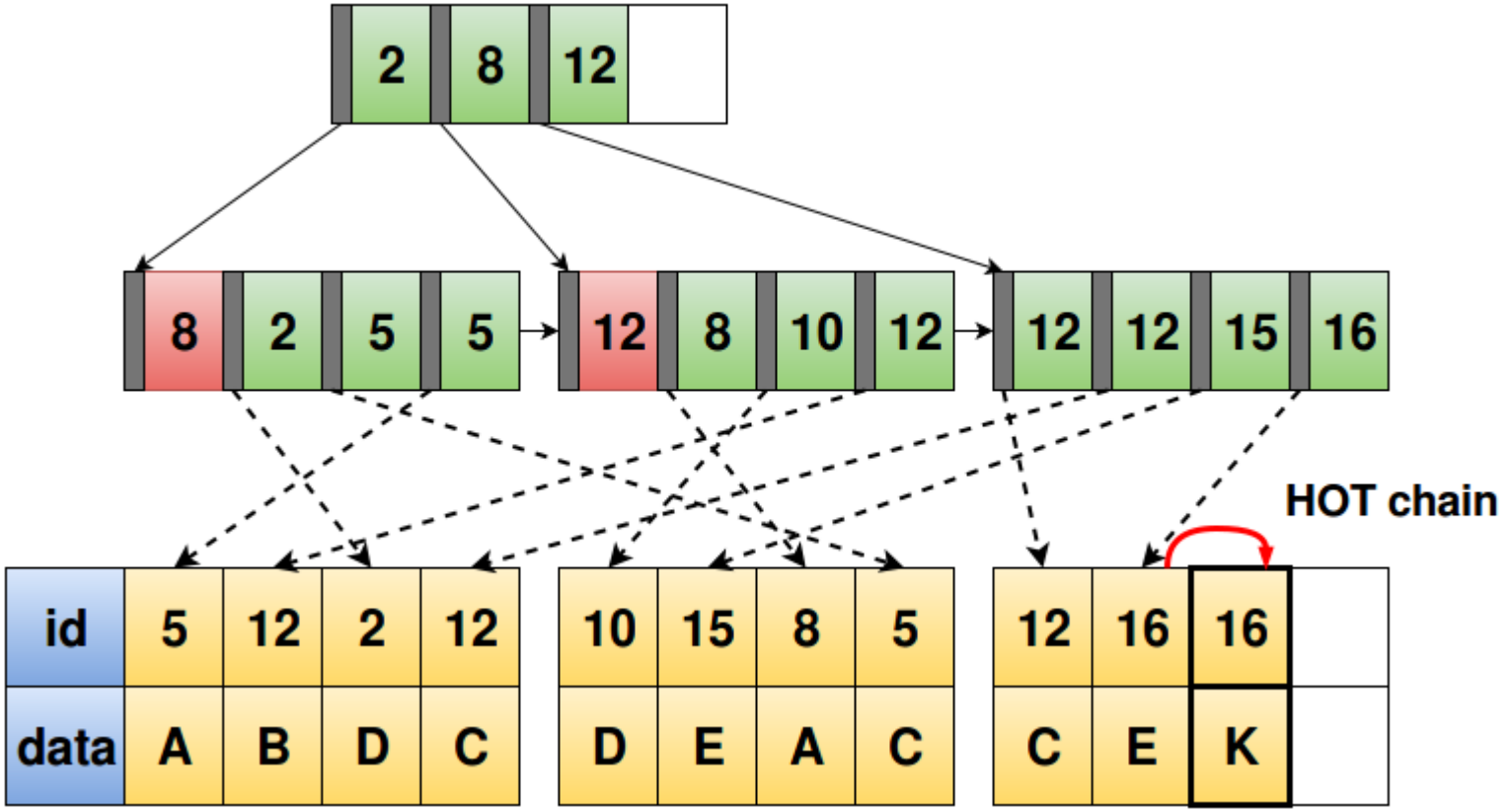

```
UPDATE tbl SET id = 16 WHERE id = 15;
```

HOT-updates




```
UPDATE tbl SET data = K WHERE id = 16;
```

HOT-updates

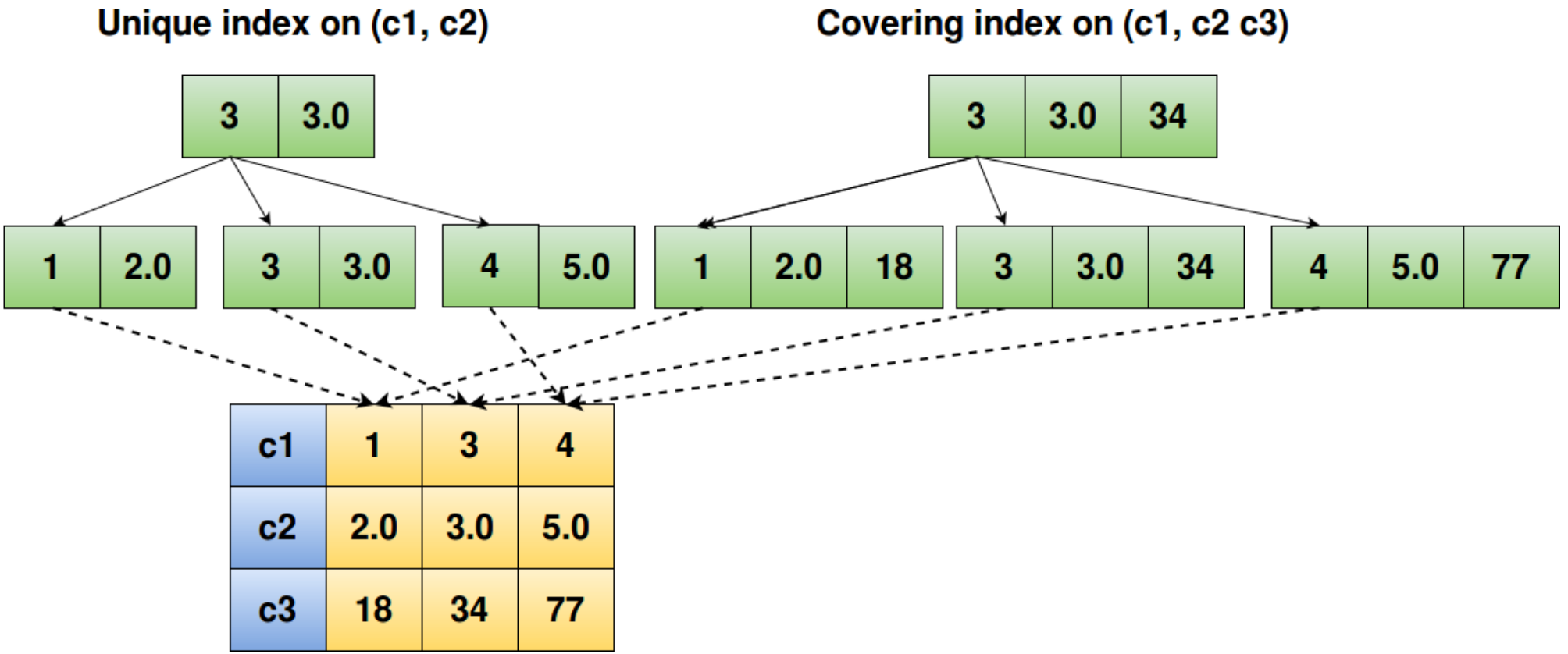



```
UPDATE tbl SET data = K WHERE id = 16;
```

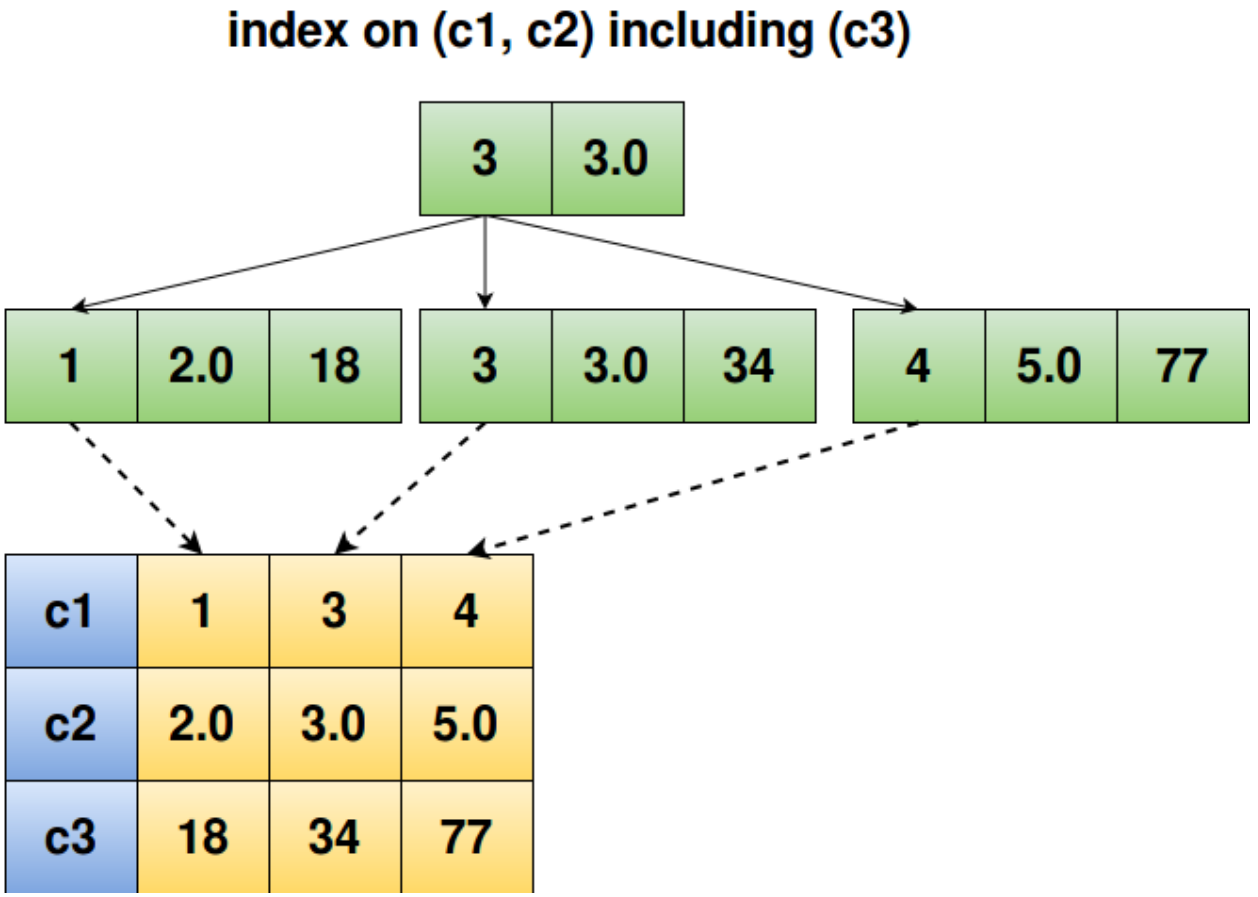
Index-only scans

- 9.2 index-only scans Allow queries to retrieve data only from indexes, avoiding heap access (Robert Haas, Ibrar Ahmed, Heikki Linnakangas, Tom Lane)

Covering indexes



Index with included columns



Index with included columns

```
CREATE UNIQUE INDEX ON tbl (a) INCLUDING (b);
CREATE INDEX ON tbl (a) INCLUDING (b);
```

```
CREATE TABLE tbl (c1 int, c2 int, c3 int, c4 box);
CREATE UNIQUE INDEX tbl_idx_unique ON tbl
using btree(c1, c2) INCLUDING(c3,c4);
ALTER TABLE tbl add UNIQUE USING INDEX
tbl_idx_unique;
```

```
CREATE TABLE tbl (c1 int, c2 int, c3 int, c4 box);
ALTER TABLE tbl add UNIQUE(c1,c2) INCLUDING(c3,c4);
```

```
CREATE TABLE tbl(c1 int,c2 int, c3 int, c4 box,
UNIQUE(c1,c2)INCLUDING(c3,c4));
```

Catalog changes

```
CREATE TABLE tbl
(c1 int,c2 int, c3 int, c4 box,
PRIMARY KEY(c1,c2)INCLUDING(c3,c4));
```

pg_class

```
 indexrelid | indnatts | indnkeyatts | indkey   | indclass
------------+----------+-------------+----------+----------
 tbl_pkey   |        4 |           2 | 1 2 3 4 | 1978 1978
```

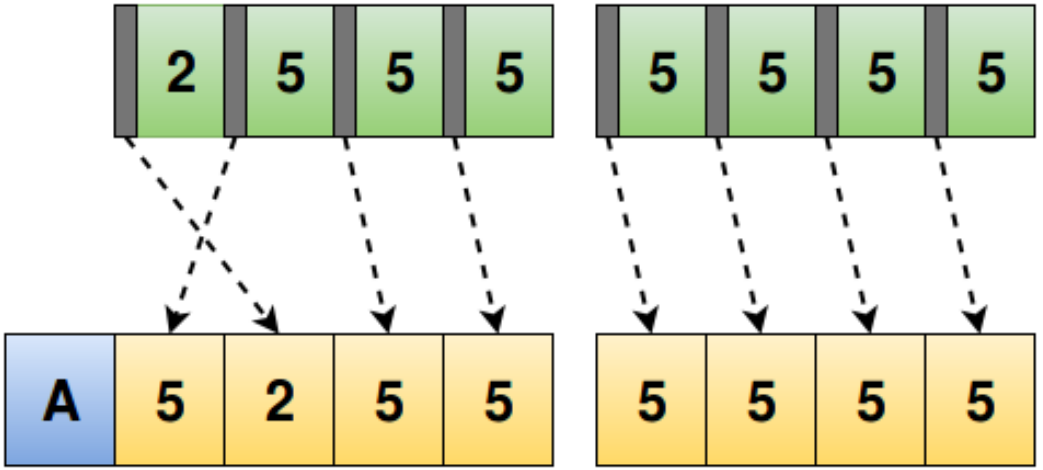
pg_constraint

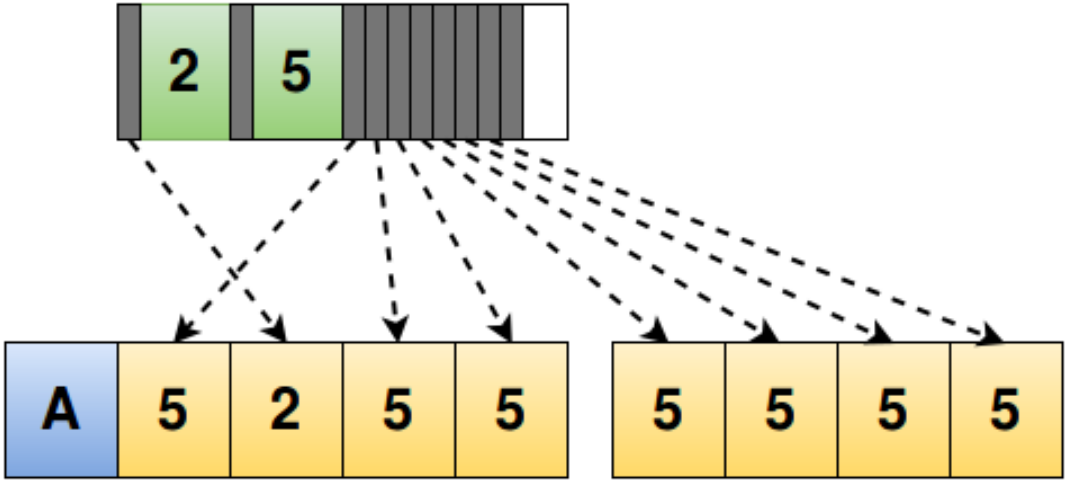
```
 conname   | conkey | coninclduing
-----------+--------+-------------
 tbl_pkey | {1,2}   | {3,4}
```

Index with included columns

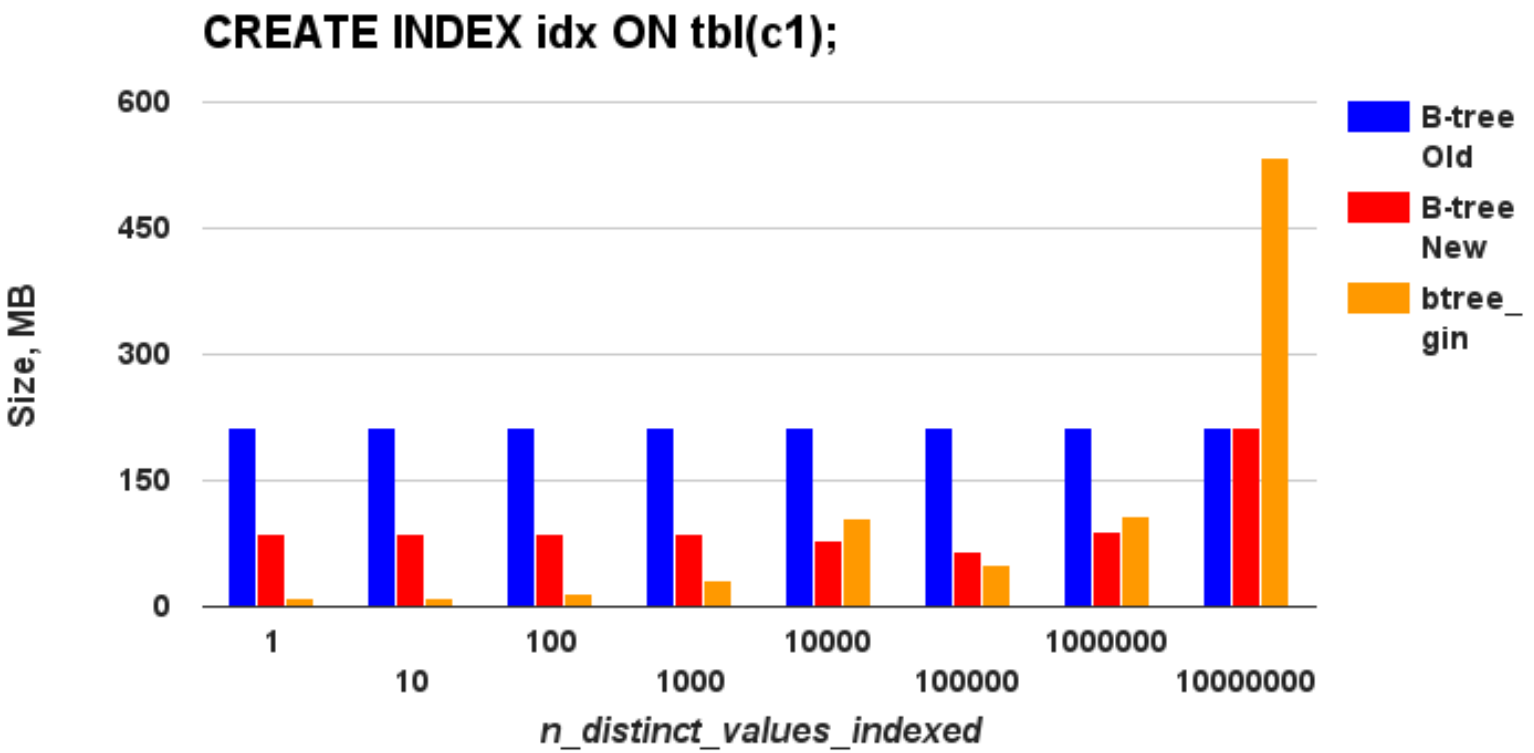
- Indexes maintenace overhead decreased
  - Size is smaller
  - Inserts are faster
- Index can contain data with no suitable opclass
- Any column deletion leads to index deletion
- HOT-updates do not work for indexed data

Effective storage of duplicates

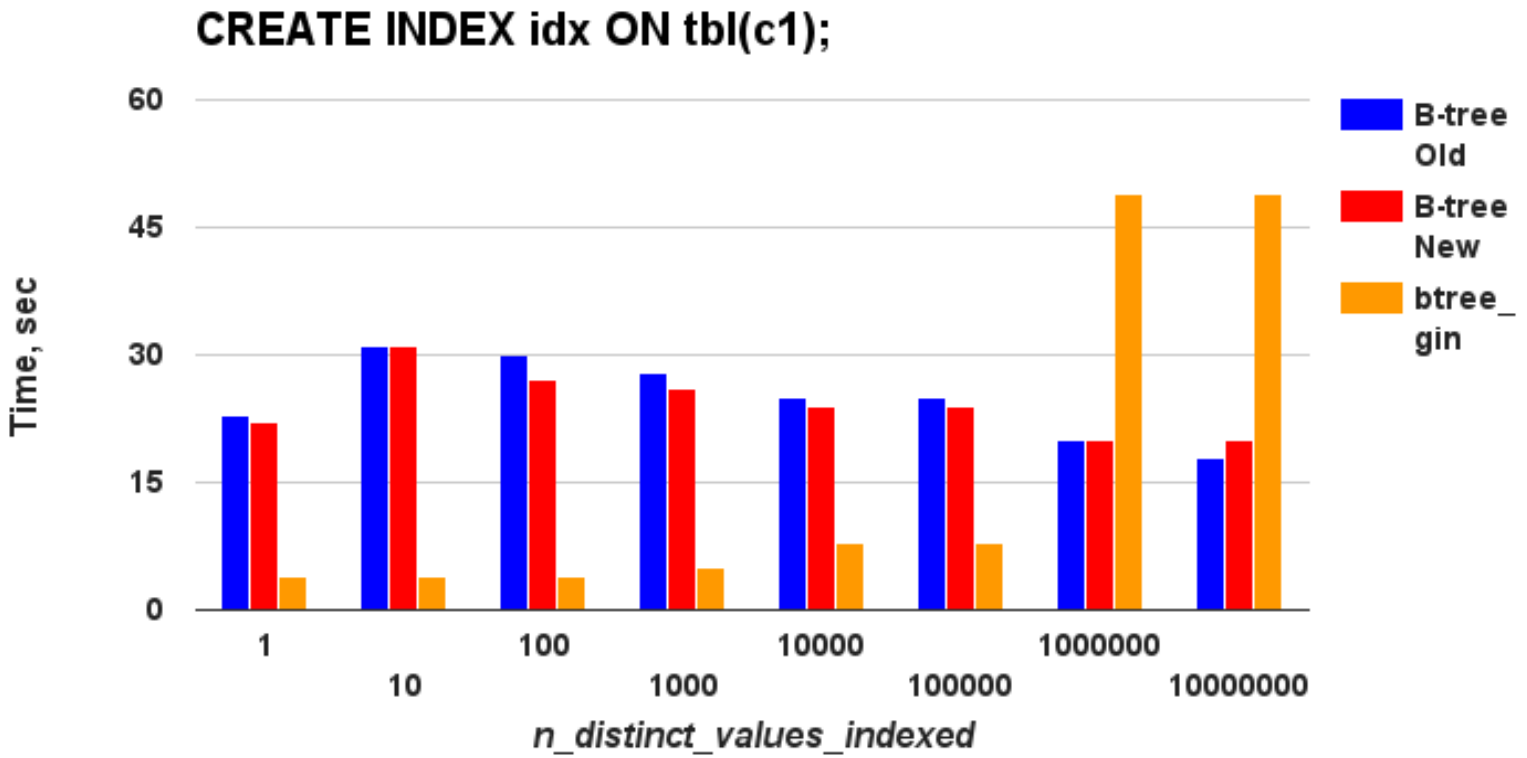




Effective storage of duplicates



Effective storage of duplicates



Difficulties of development

- Complicated code
- It's a crucial subsystem that can badly break everything including system catalog
- Few active developers and experts in this area
- Few instruments for developers

pageinspect

```
postgres=# select bt.blkno, bt.type, bt.live_items,
bt.free_size, bt.btpo_prev, bt.btpo_next
from generate_series(1,pg_relation_size('idx')/8192 - 1) as n,
lateral bt_page_stats('idx', n::int) as bt;

 blkno | type | live_items | free_size | btpo_prev | btpo_next
-------+------+------------+-----------+-----------+-----------
     1 | l    |        367 |       808 |         0 |         2
     2 | l    |        235 |      3448 |         1 |         0
     3 | r    |          2 |      8116 |         0 |         0
```

pageinspect

```
postgres=# select * from bt_page_items('idx', 1);
```

itemoffset	ctid	itemlen	nulls	vars	data
1	(0,1)	16	f	f	01 00 00 00 00 00 00 00
2	(0,2)	16	f	f	01 00 00 00 01 00 00 00
3	(0,3)	16	f	f	01 00 00 00 02 00 00 00
4	(0,4)	16	f	f	01 00 00 00 03 00 00 00
5	(0,5)	16	f	f	01 00 00 00 04 00 00 00
6	(0,6)	16	f	f	01 00 00 00 05 00 00 00
7	(0,7)	16	f	f	01 00 00 00 06 00 00 00
8	(0,8)	16	f	f	01 00 00 00 07 00 00 00
9	(0,9)	16	f	f	01 00 00 00 08 00 00 00
10	(0,10)	16	f	f	01 00 00 00 09 00 00 00
11	(0,11)	16	f	f	01 00 00 00 0a 00 00 00

pageinspect

```
postgres=# select blkno, bt.itemoffset, bt.ctid, bt.itemlen, bt.data
from generate_series(1,pg_relation_size('idx')/8192 - 1) as blkno,
lateral bt_page_items('idx', blkno::int) as bt where itemoffset <3;

 blkno | itemoffset |  ctid   | itemlen |          data
-------+------------+---------+---------+------------------------
     1 |          1 | (1,141) |      16 | 01 00 00 00 6e 01 00 00
     1 |          2 | (0,1)   |      16 | 01 00 00 00 00 00 00 00
     2 |          1 | (1,141) |      16 | 01 00 00 00 6e 01 00 00
     2 |          2 | (1,142) |      16 | 01 00 00 00 6f 01 00 00
     3 |          1 | (1,1)   |       8 |
     3 |          2 | (2,1)   |      16 | 01 00 00 00 6e 01 00 00
```

Postgres PROFESSIONAL

amcheck

- `bt_index_check(index regclass);`
- `bt_index_parent_check(index regclass);`

Postgres PROFESSIONAL

What do we need?

- Index compression
  - Compression of leading columns
  - Page compression
- Index-Organized-Tables (primary indexes)
  - Users don't want to store data twice
- KNN for B-tree
  - Nice task for beginners
- Batch update of indexes
- Indexes on partitioned tables
  - pg_pathman
  - Global indexes
  - Global Partitioned Indexes

Postgres PROFESSIONAL

Thanks for attention!
Any questions?

a.lubennikova@postgrespro.ru