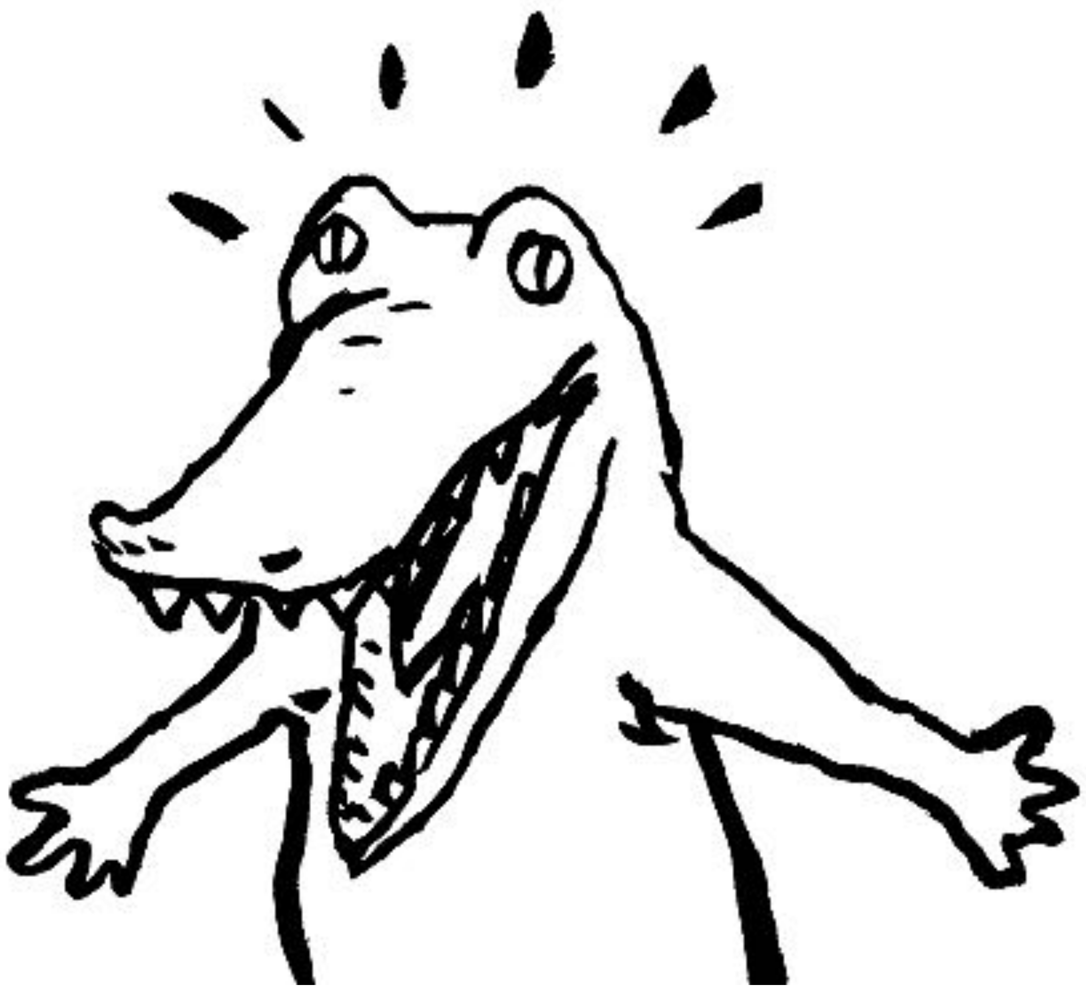
Indexes in Postgres

(the long story or crocodiles going to the dentist)

Louise Grandjonc

About me



Solutions Engineer at Citus Data

Previously lead python developer

Postgres enthusiast

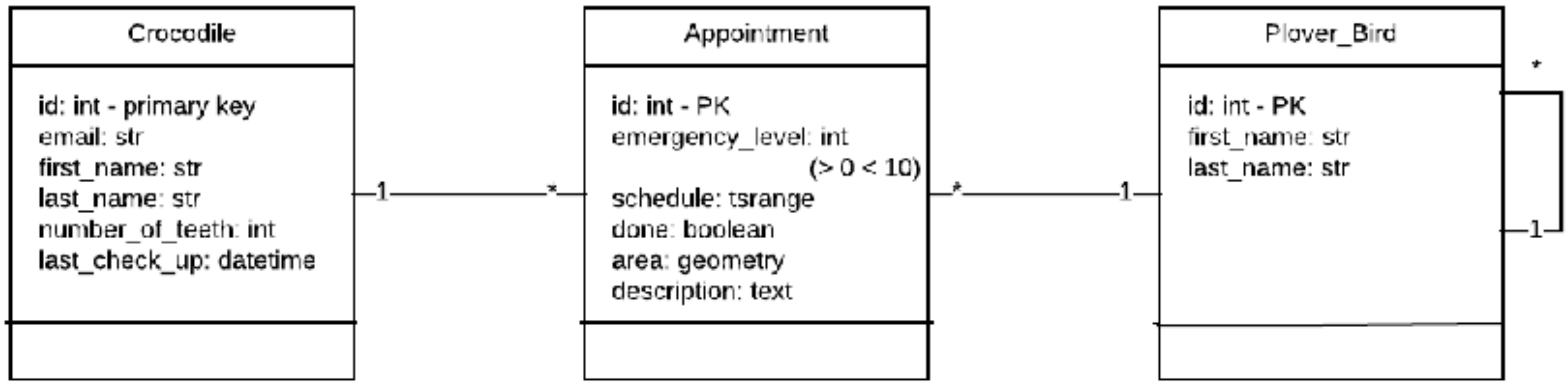
@louisemeta on twitter

www.louisemeta.com

What we're going to talk about

1. What are indexes for?
2. Pages and CTIDs
3. B-Tree
4. GIN
5. GiST
6. SP-GiST
7. Brin
8. Hash

First things first: the crocodiles



- 250k crocodiles
- 100k birds
- 2M appointments

What are indexes for?

Constraints

Some constraints transform into indexes.

- PRIMARY KEY
- UNIQUE
- EXCLUDE USING

In the crocodile table

```
"crocodile_pkey" PRIMARY KEY, btree (id)
"crocodile_email_uq" UNIQUE CONSTRAINT, btree (email)
```

In the appointment table

```
Indexes:
    "appointment_pkey" PRIMARY KEY, btree (id)
    "appointment_crocodile_id_schedule_excl" EXCLUDE USING gist
    (crocodile_id WITH =, schedule WITH &&)
```

Query optimization

Often the main reason why we create indexes

Why do indexes make queries faster

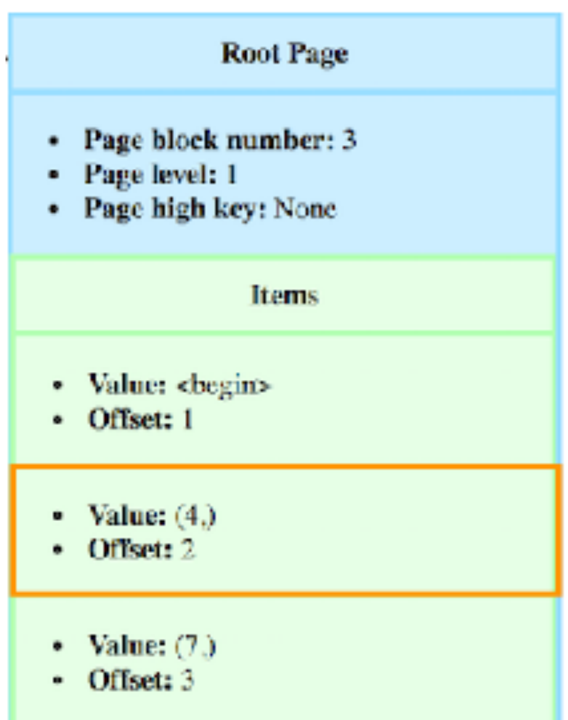
In an index, tuples (value, pointer) are stored.
Instead of reading the entire table for a value, you just go to the index (kind of like in an encyclopedia)

Email	Pointer to
li.miller220003@croco.com	Pointer to crocodile pk: 220003
agustin.lambert220004@croco.com	Pointer to crocodile pk: 220004
rebecca.ahn220005@croco.com	Pointer to crocodile pk: 220005
guy.morelli220006@croco.com	Pointer to crocodile pk: 220006
deborah.martini220007@croco.com	Pointer to crocodile pk: 220007
pedro.ito220008@croco.com	Pointer to crocodile pk: 220008
mary.patel220009@croco.com	Pointer to crocodile pk: 220009
pedro.richardson220010@croco.com	Pointer to crocodile pk: 220010
deborah.colombo220011@croco.com	Pointer to crocodile pk: 220011
will.becker220012@croco.com	Pointer to crocodile pk: 220012

Pages, heaps and their pointers

Pages

- PostgreSQL uses **pages to store data** from indexes or tables
- A page has a **fixed size of 8kB**
- A page has a header and items
- **In an index**, each item is a **tuple (value, pointer)**
- Each item in a page is referenced to with a pointer called **ctid**
- The ctid consist of **two numbers**, the number of the page (**the block number**) and the **offset of the item.**



The ctid of the item with value 4 would be (3, 2).

pageinspect and gevel

Extensions to look into your index pages

pageinspect, gevel and a bit of python

Page inspect is an extension that allows you to explore a bit what's inside the pages

Functions for **BTree, GIN, BRIN and Hash** indexes

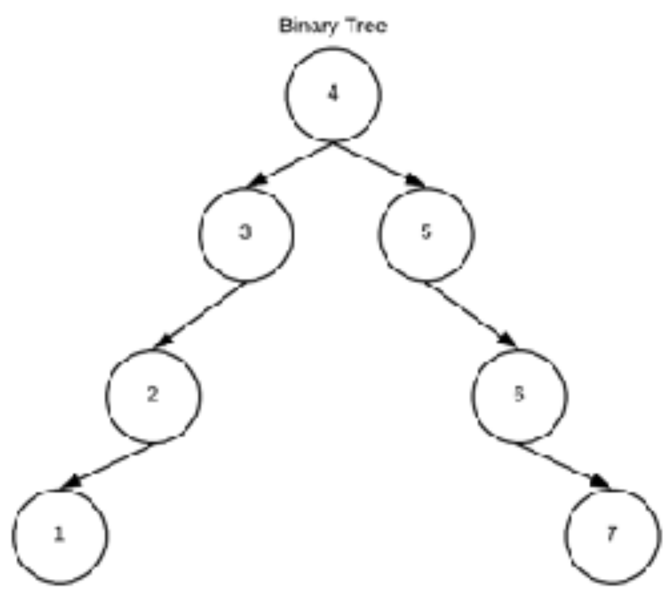
Gevel adds functions to **GiST, SP-Gist and GIN.**

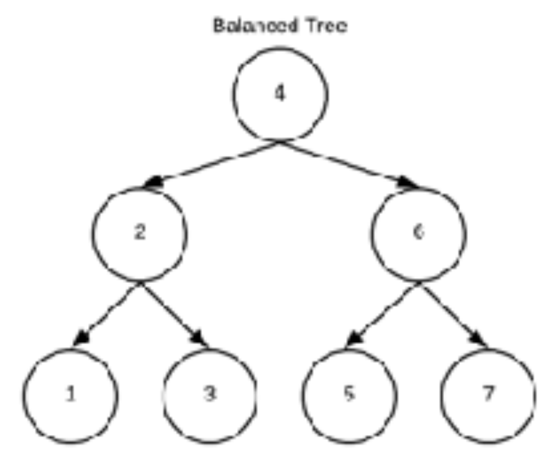
Used them to generate pictures for BTree and GiST
https://github.com/louiseGrandjonc/pageinspect_inspector

B-Trees

B-Trees internal data structure - 1

- A BTree in a **balanced tree**
- All the leaves are at **equal distance from the root.**
- A parent node can have **multiple children** minimizing the tree's depth
- Postgres implements the **Lehman & Yao Btree**





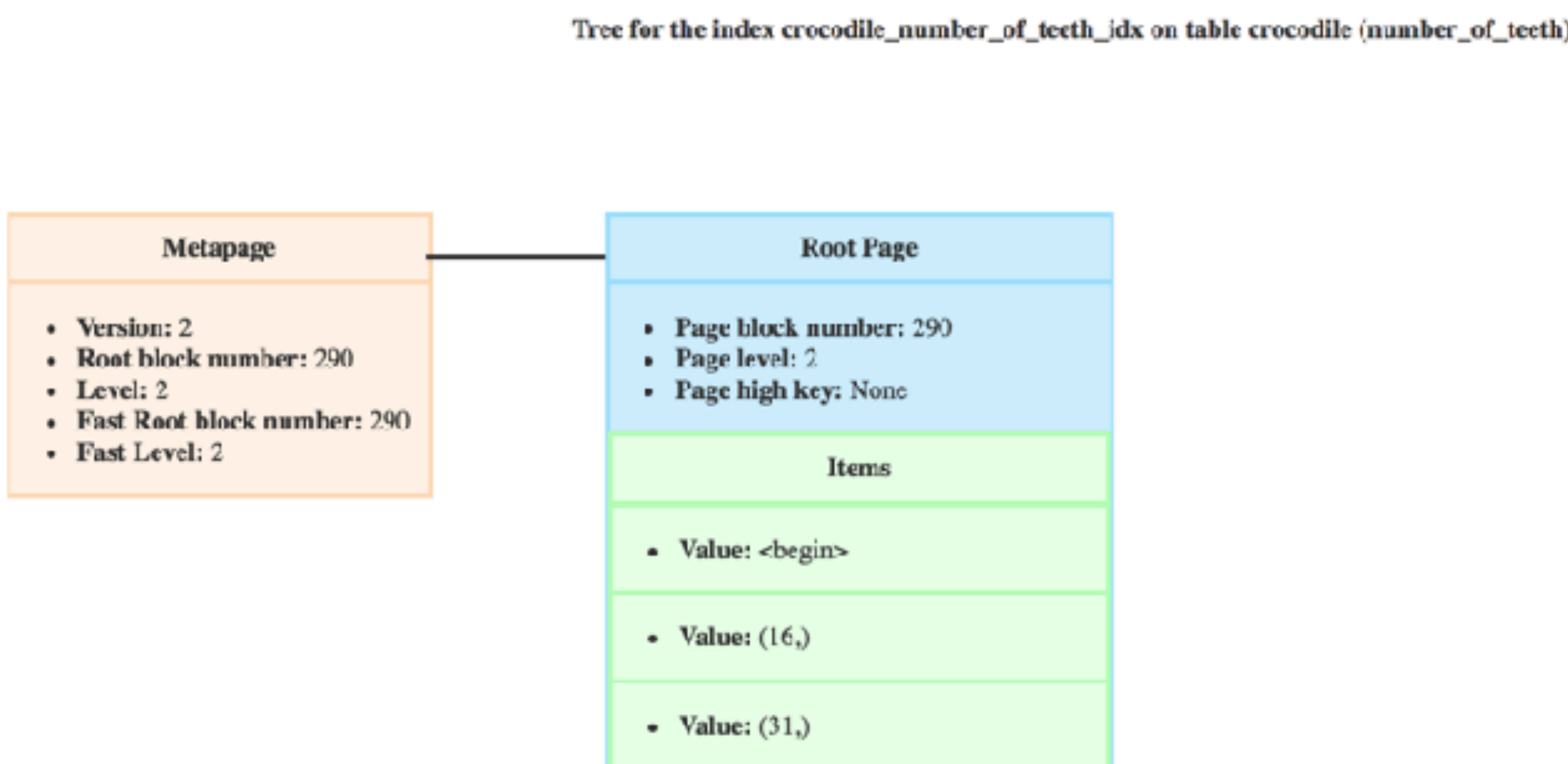
Let's say we would like to filter or order on the crocodile's number of teeth.

```
CREATE INDEX ON crocodile (number_of_teeth);
```

B-Trees internal data structure - 2
Metapage

The **metapage** is always the **first page** of a BTree index. It contains:

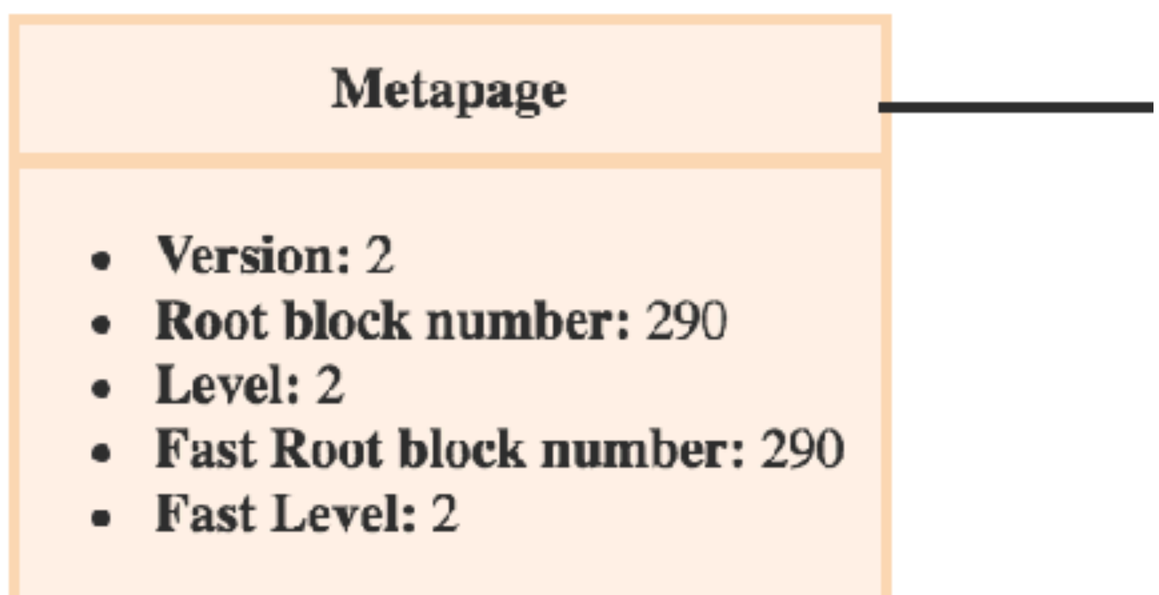
- The **block number** of the **root page**
- The **level** of the **root**
- A **block number** for the **fast root**
- The **level** of the **fast root**



B-Trees internal data structure - 2
Metapage

Using page inspect, you can get the information on the metapage

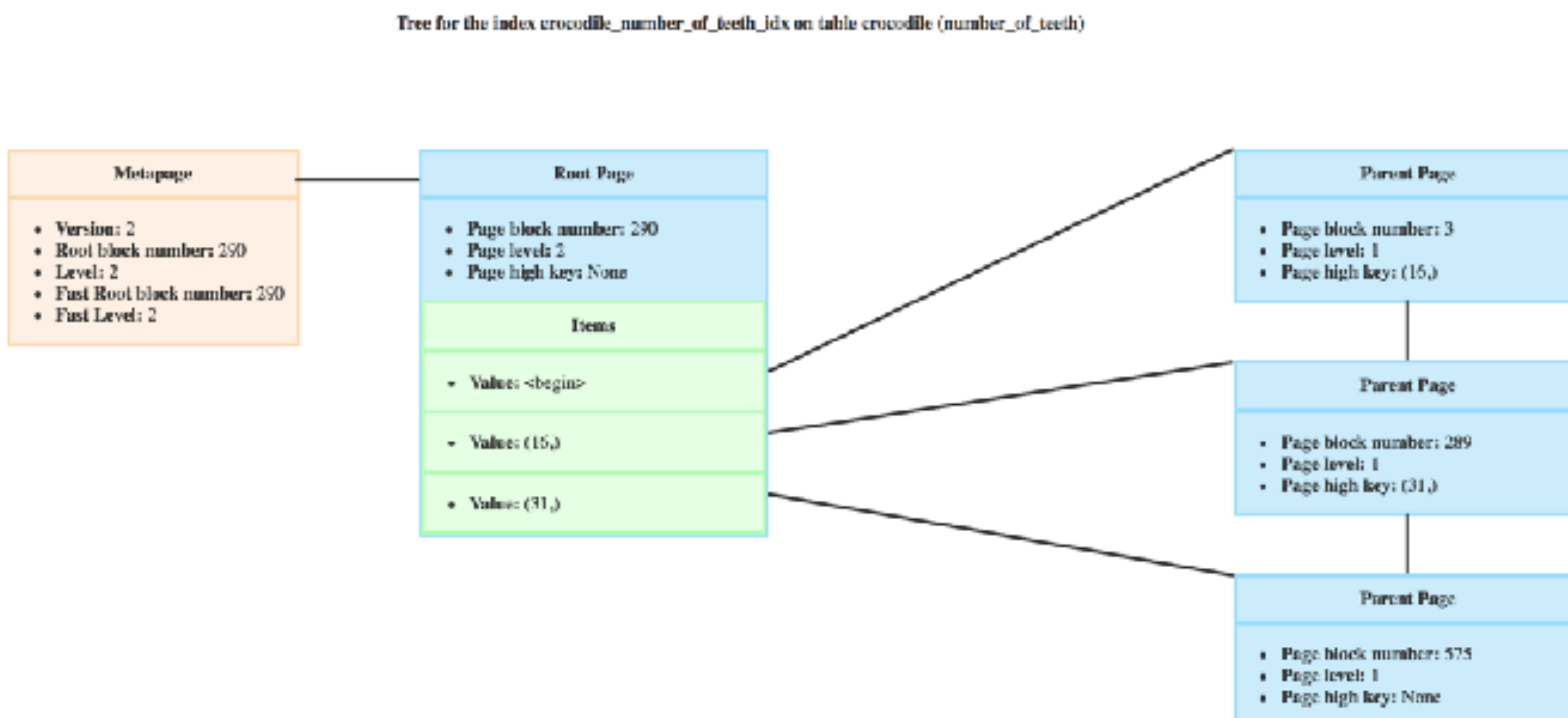
```
SELECT * FROM bt_metap('crocodile_number_of_teeth_idx');
 magic  | version | root | level | fastroot | fastlevel
--------+---------+------+-------+----------+-----------
 340322 |       2 |  290 |     2 |      290 |         2
(1 row)
```



B-Trees internal data structure - 3
Pages

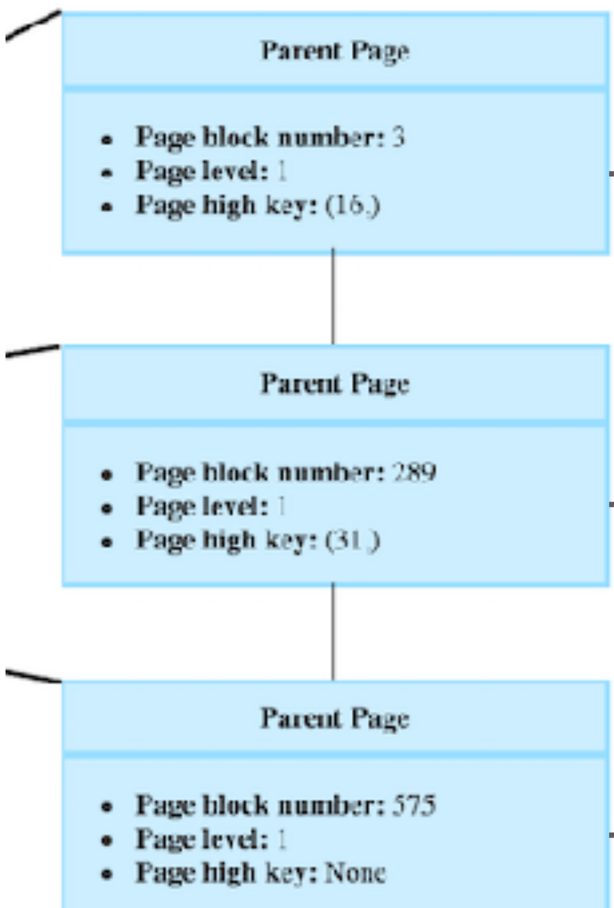
The root, the parents, and the leaves are all pages with the same structure.
Pages have:

- **A block number**, here the root block number is 290
- A **high key**
- A **pointer** to the **next (right) and previous pages**
- **Items**



B-Trees internal data structure - 4
Pages high key

- High key is specific to Lehman & Yao BTrees
- **Any item** in the page will have a **value lower or equal to the high key**
- The **root doesn't have a high key**
- The **right-most page** of a level **doesn't have a high key**



In page 3, I will find crocodiles with 3 or less teeth

In page 289, with 31 and less

And in page 575, there is no high key as it's the rightmost page.

B-Trees internal data structure - 5
Next and previous pages pointers

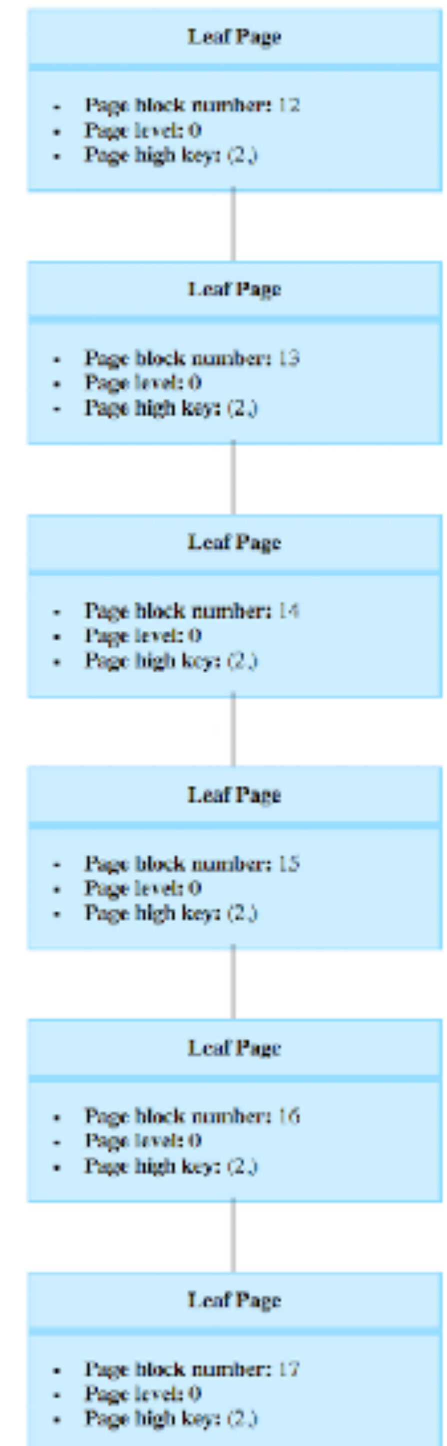
- Specificity of the Yao and Lehmann BTree
- Pages in the **same level** are in a **linked list**

Very useful for ORDER BY

For example:

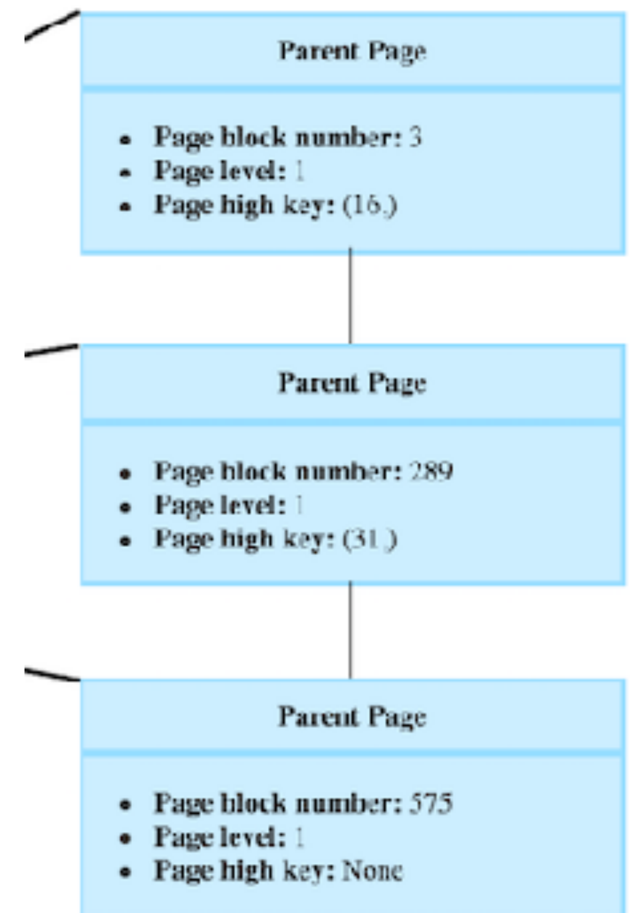
```
SELECT number_of_teeth
FROM crocodile ORDER BY number_of_teeth ASC
```

Postgres would start at the first leaf page and thanks to the next page pointer, has directly all rows in the right order.



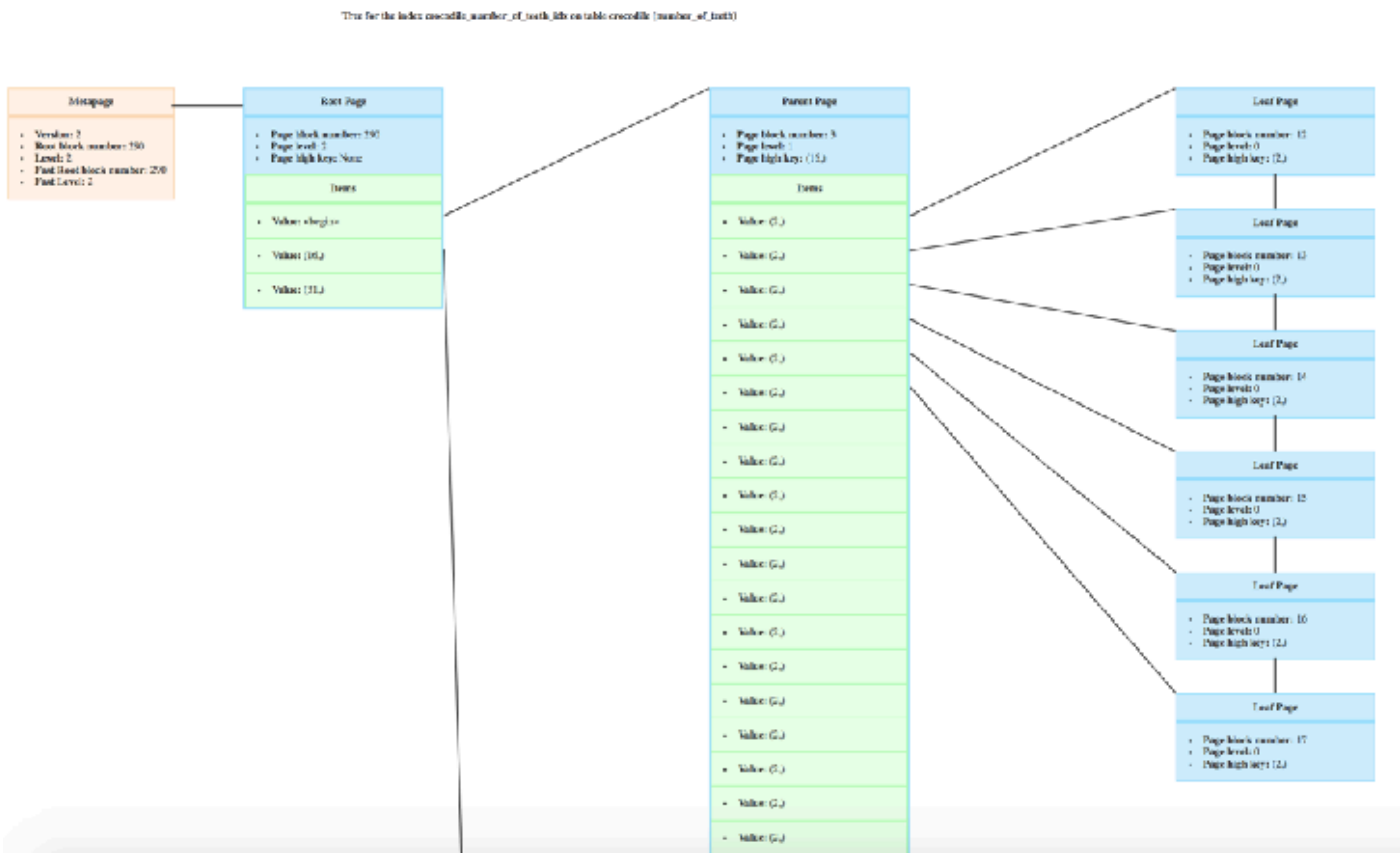
B-Trees internal data structure - 6
Page inspect for BTree pages

```
SELECT * FROM bt_page_stats('crocodile_number_of_teeth_idx',
                            289);
-[ RECORD 1 ]-+-----
blkno         | 289
type          | i
live_items    | 285
dead_items    | 0
avg_item_size | 15
page_size     | 8192
free_size     | 2456
btpo_prev     | 3
btpo_next     | 575
btpo          | 1
btpo_flags    | 0
```



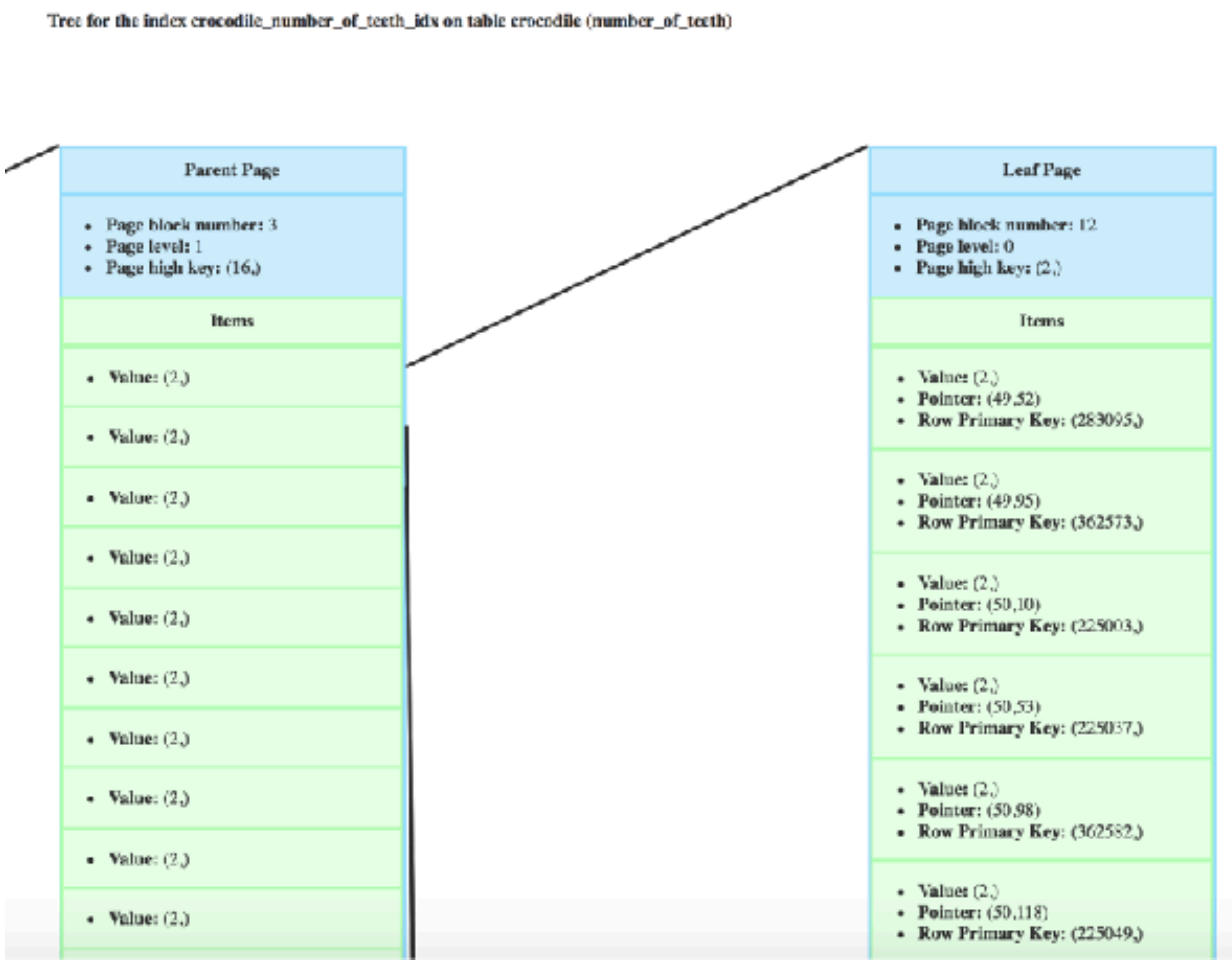
B-Trees internal data structure - 7
Items

- Items have **a value and a pointer**
- **In the parents**, the **ctid** points to the **child page**
- In the **parents**, the value is the **value of the first item** in the **child** page



B-Trees internal data structure - 8
Items

- In the **leaves**, the **ctid** is to the **heap tuple in the table**
- In **the leaves** it's the **value of the column(s)** of the row



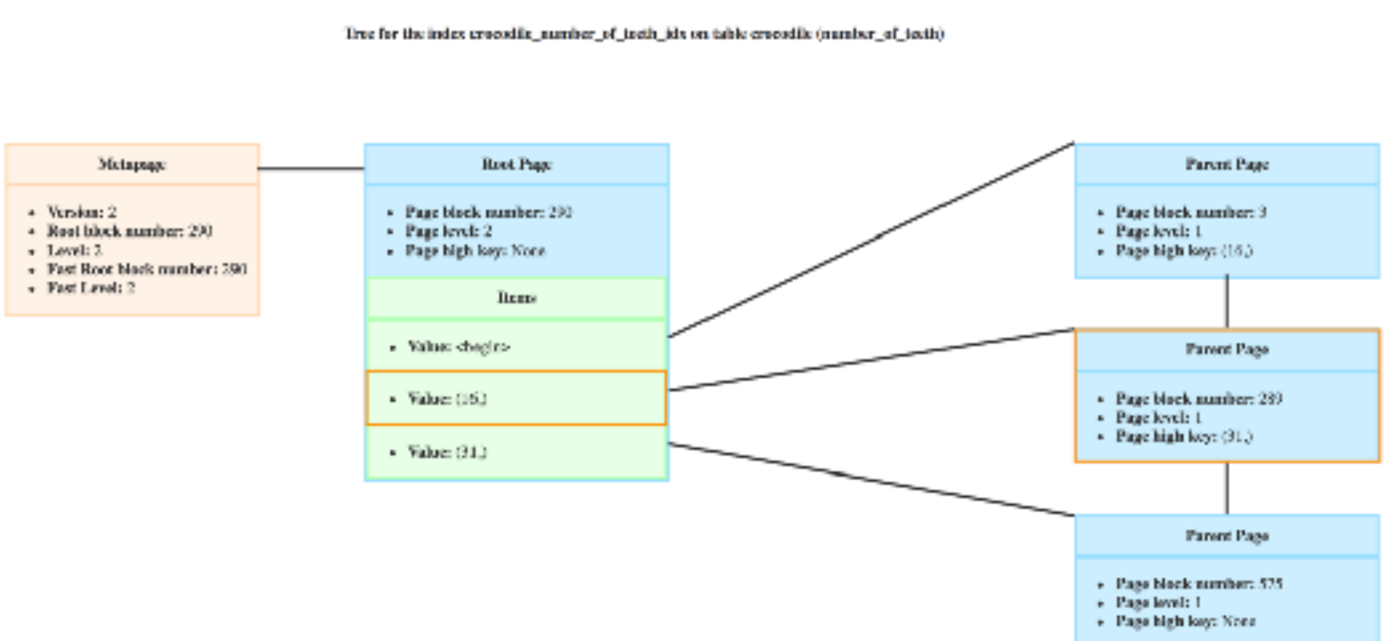
B-Trees internal data structure
To sum it up

- A **Btree** is a **balanced tree**. PostgreSQL implements the Lehmann & Yao algorithm
- **Metapage** contains information on the root and fast root
- **Root, parent, and leaves are pages**.
- **Each level is a linked list** making it easier to move from one page to an other within the same level.
- **Pages have a high key** defining the biggest value in the page
- **Pages have items pointing to an other page or the row**.

B-Trees - Searching in a BTree

1. **Scan keys** are created

2. Starting from the root until a leaf page
   - Is **moving to the right page necessary?**
   - If the **page is a leaf, return the first item** with a value **higher or equal to the scan key**
   - **Binary search** to find the right path to follow
   - **Descend** to the **child page and lock it**



```
SELECT email FROM crocodile WHERE number_of_teeth >= 20;
```

B-Trees - Scan keys

Postgres uses the query scan to **define scankeys**.

If possible, **redundant keys** in your query **are eliminated** to keep only the **tightest bounds.**

```
SELECT email, number_of teeth FROM crocodile
WHERE number_of_teeth > 4 AND number_of_teeth > 5
ORDER BY number_of_teeth ASC;
```

The tightest bound is number_of_teeth > 5

```
                 email                   | number_of_teeth
-----------------------------------------+-----------------
 anne.chow222131@croco.com               |               6
 valentin.williams222154@croco.com       |               6
 pauline.lal222156@croco.com             |               6
 han.yadav232276@croco.com               |               6
```

B-Trees - About read locks

We put a **read lock** on the **currently examined page**.

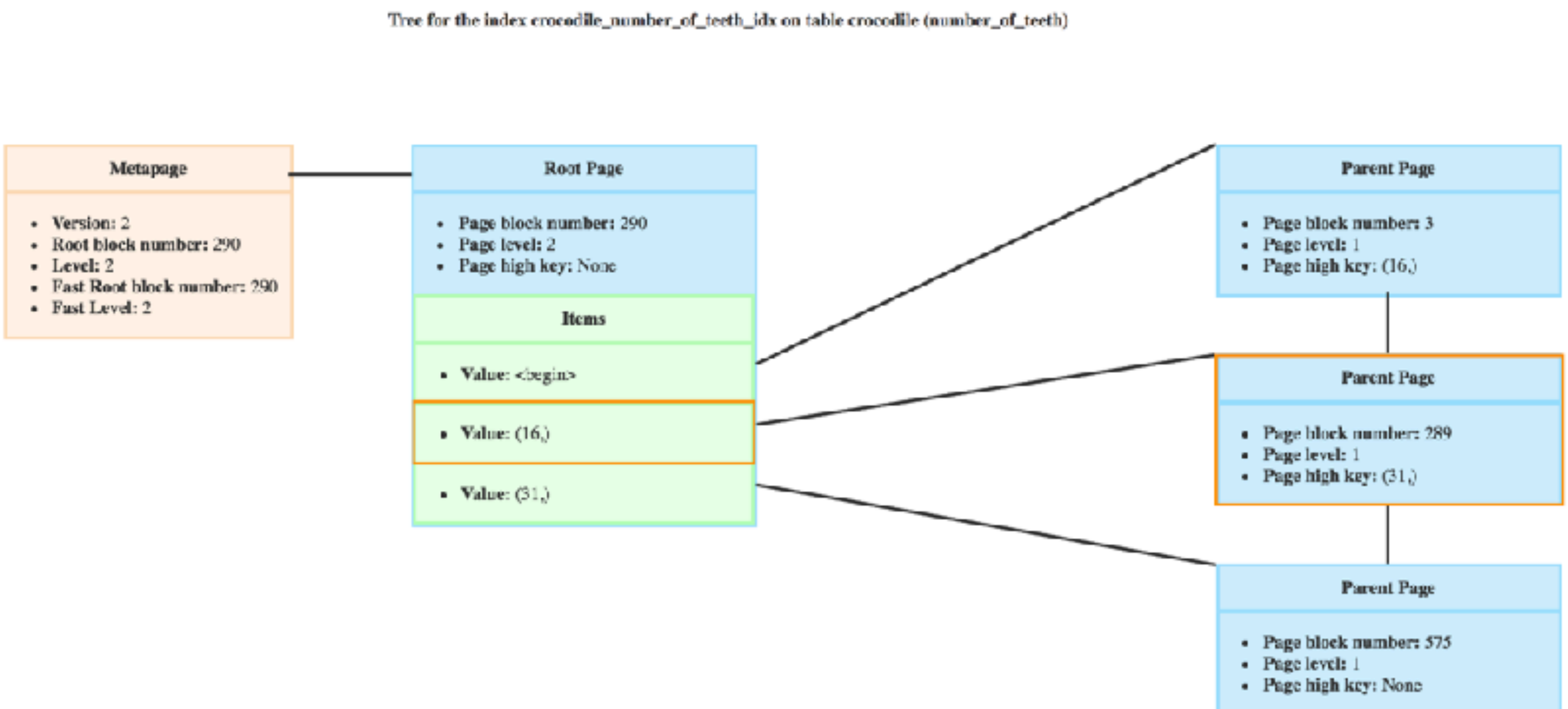
Read locks **ensure that the records on that page are not modified while reading it.**

There could still be a **concurrent insert on a child page causing a page split.**

BTrees - Is moving right necessary?

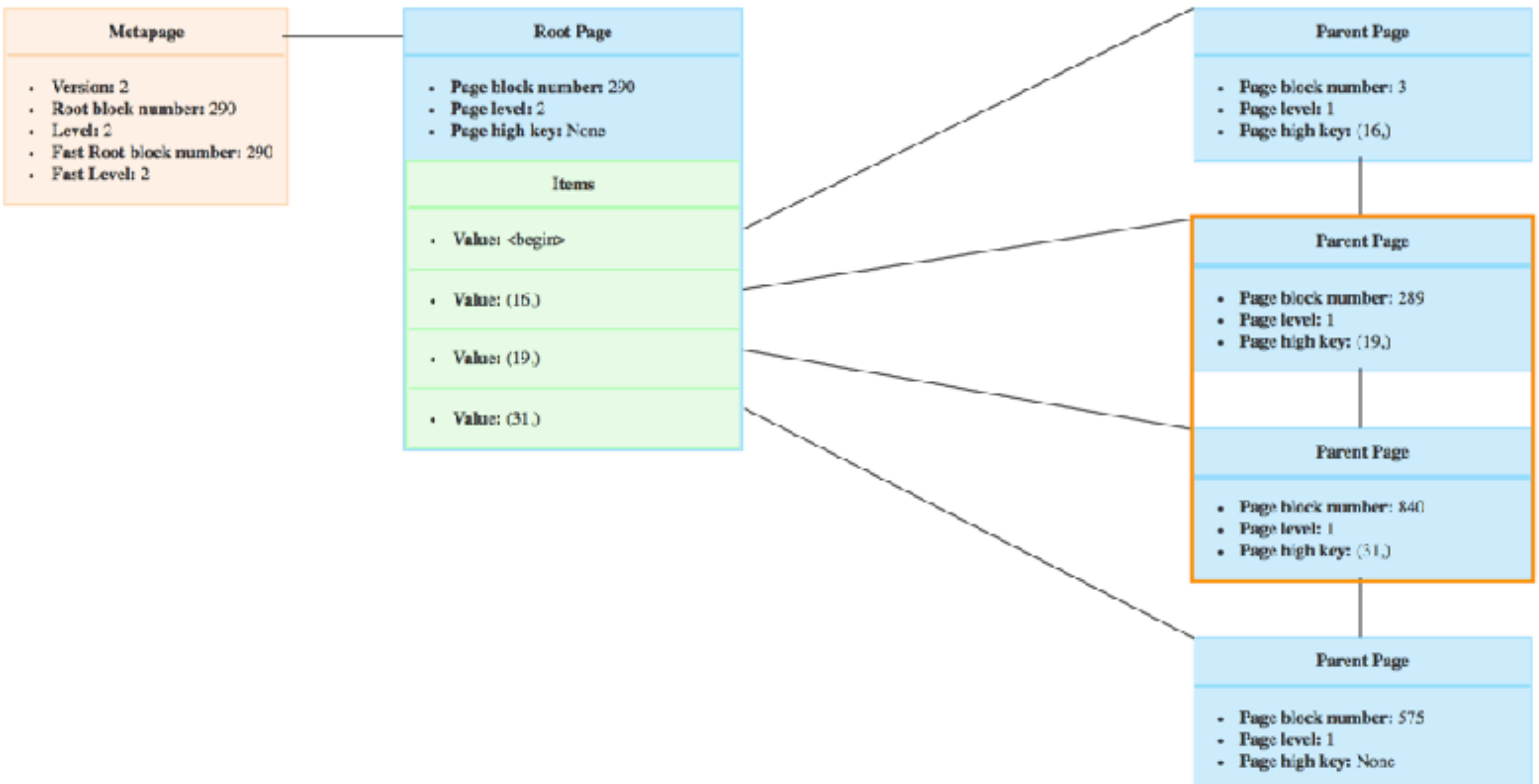
```
SELECT email FROM crocodile WHERE number_of_teeth >= 20;
```

Concurrent insert while visiting the root:



BTrees - Is moving right necessary?

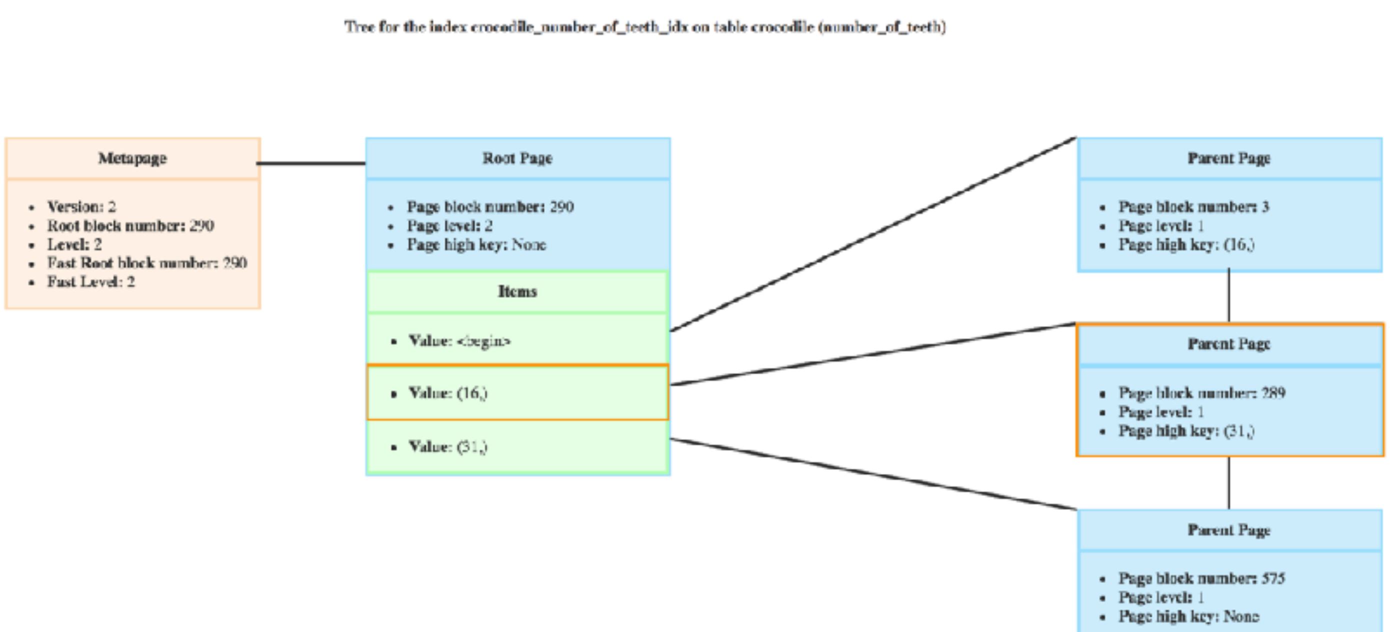
The **new high key of child page** is **19**
So we need to move right to the page 840



B-Trees - Searching in a BTree

1. **Scan keys** are created

2. Starting from the root until a leaf page
   - Is **moving to the right page necessary?**
   - If the **page is a leaf, return the first item** with a value **higher or equal to the scan key**
   - **Binary search** to find the right path to follow
   - **Descend** to the **child page and lock it**



```
SELECT email FROM crocodile WHERE number_of_teeth >= 20;
```

BTrees - Inserting

1. Find the **right insert page**
2. **Lock** the page
3. Check **constraint**
4. **Split page** if necessary and **insert row**
5. **In case of page split**, recursively insert a **new item in the parent level**

BTrees -Inserting
Finding the right page

Auto-incremented values:

Primary keys with a sequence for example, like the index crocodile_pkey.

New values will **always be inserted in the right-most leaf page**.
To avoid using the search algorithm**, Postgres caches this page.**

Non auto-incremented values:

The search algorithm is used to find the right leaf page.

BTrees -Inserting
Page split

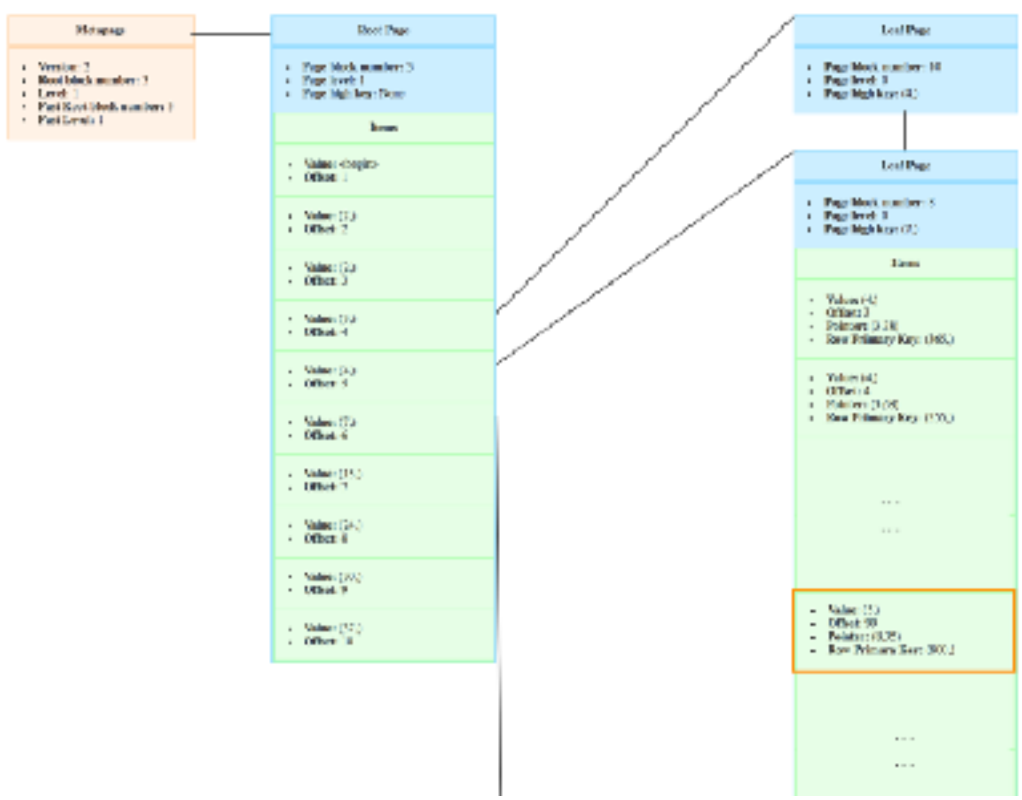
1. Is a split necessary?

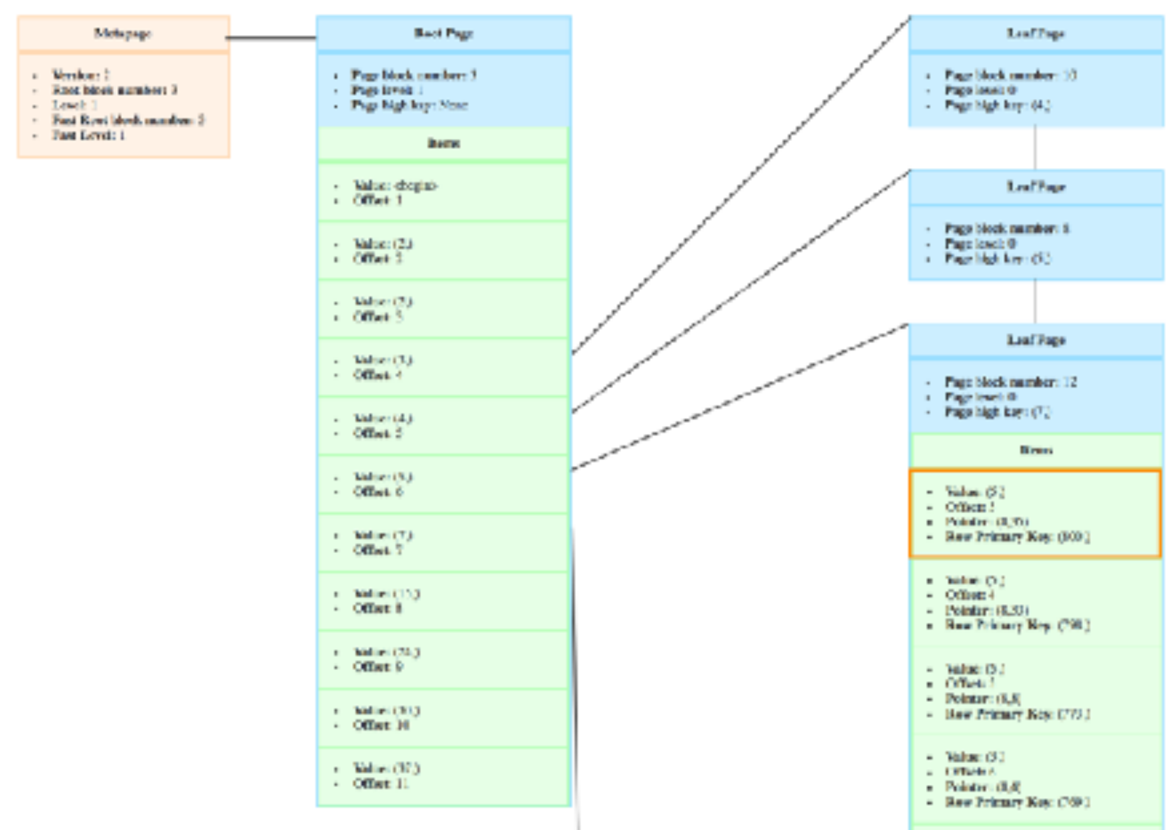
If the **free space on the target page is lower than the item's size**, then a split is necessary.

2. Finding the split point

Postgres wants to **equalize the free space on each page** to limit page splits in future inserts.

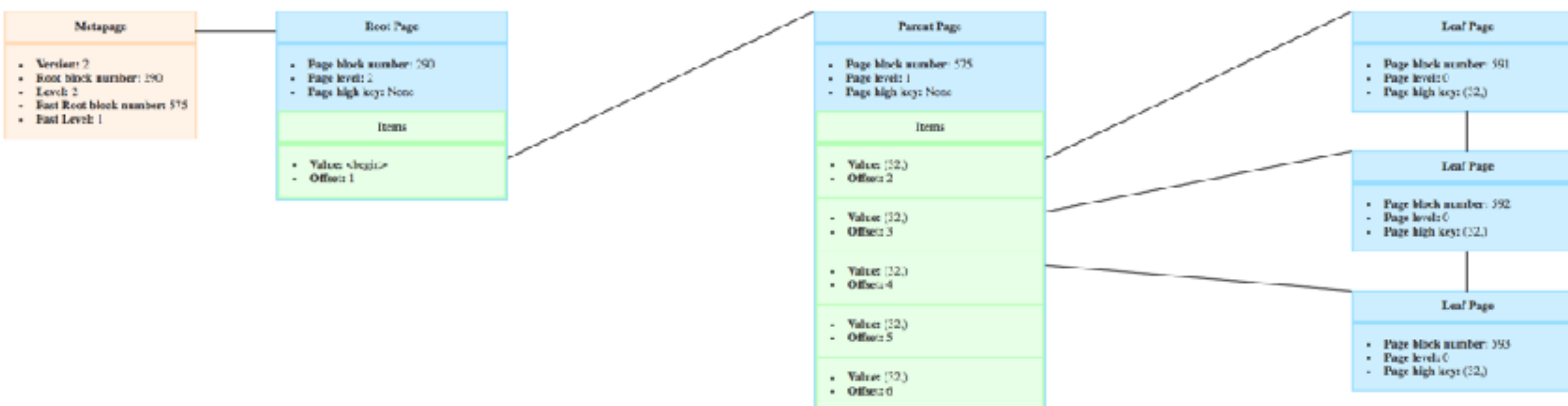
3. Splitting





BTrees - Deleting

- Items are **marked as deleted** and will be **ignored in future index scans until VACUUM**
- A **page is deleted** only if **all its items** have been **deleted**.
- It is possible to end up with a tree with several levels with only one page.
- The fast root is used to **optimize the search**.



GIN

GIN

- **GIN (Generalized Inverted Index)**
- Used to index **arrays, jsonb, and tsvector** (for fulltext search) columns.
- Efficient for <@, &&, @@@ operators

New column healed_teeth (integer[])

```
croco=# SELECT email, number_of_teeth, healed_teeth FROM crocodile WHERE id =1;
-[ RECORD 1 ]---+-----------------------------------------------------
email           | louise.grandjonc1@croco.com
number_of_teeth | 58
healed_teeth    | {16,11,55,27,22,41,38,2,5,40,52,57,28,50,10,15,1,12,46}
```

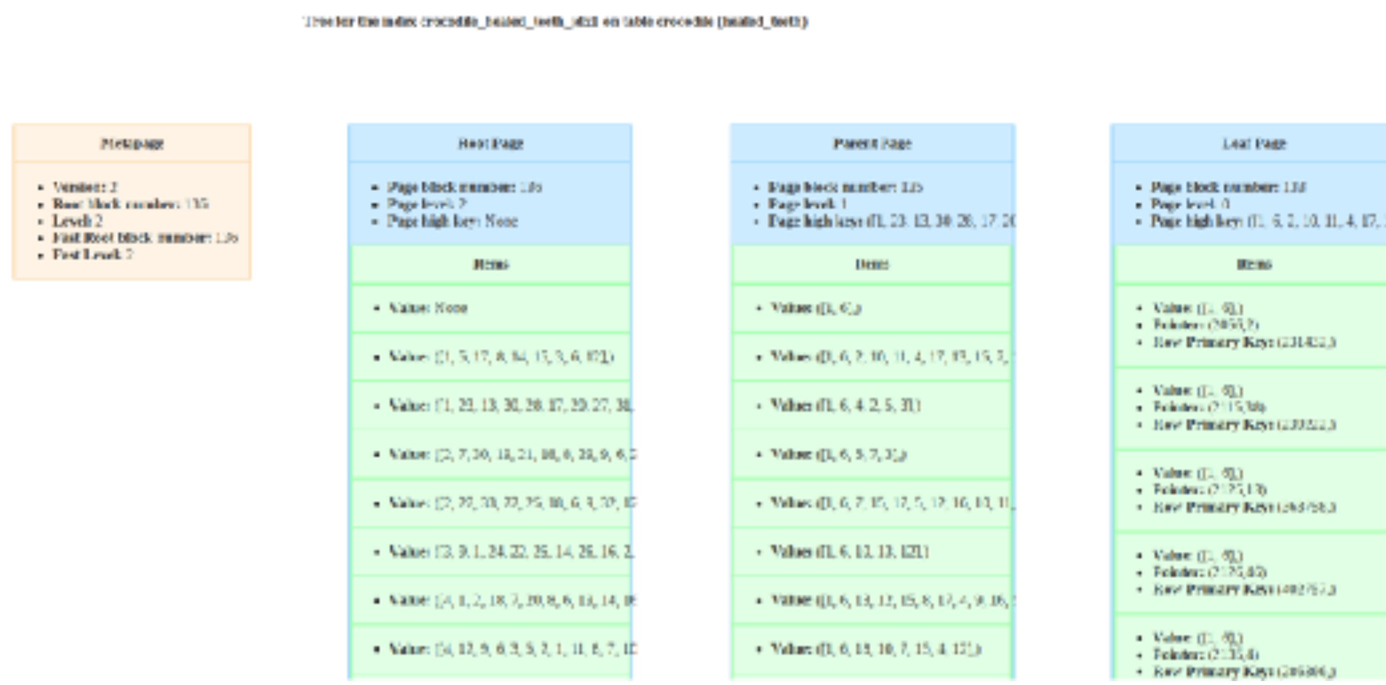
Here is how to create the GIN index for this column

```
CREATE INDEX ON crocodile USING GIN(healed_teeth);
```

GIN
How is it different from a BTree? - Keys

- GIN indexes are **binary trees**
- Just like BTree, their **first page** is a **metapage**

First difference: the keys



BTree index on healed_teeth

The indexed values are arrays

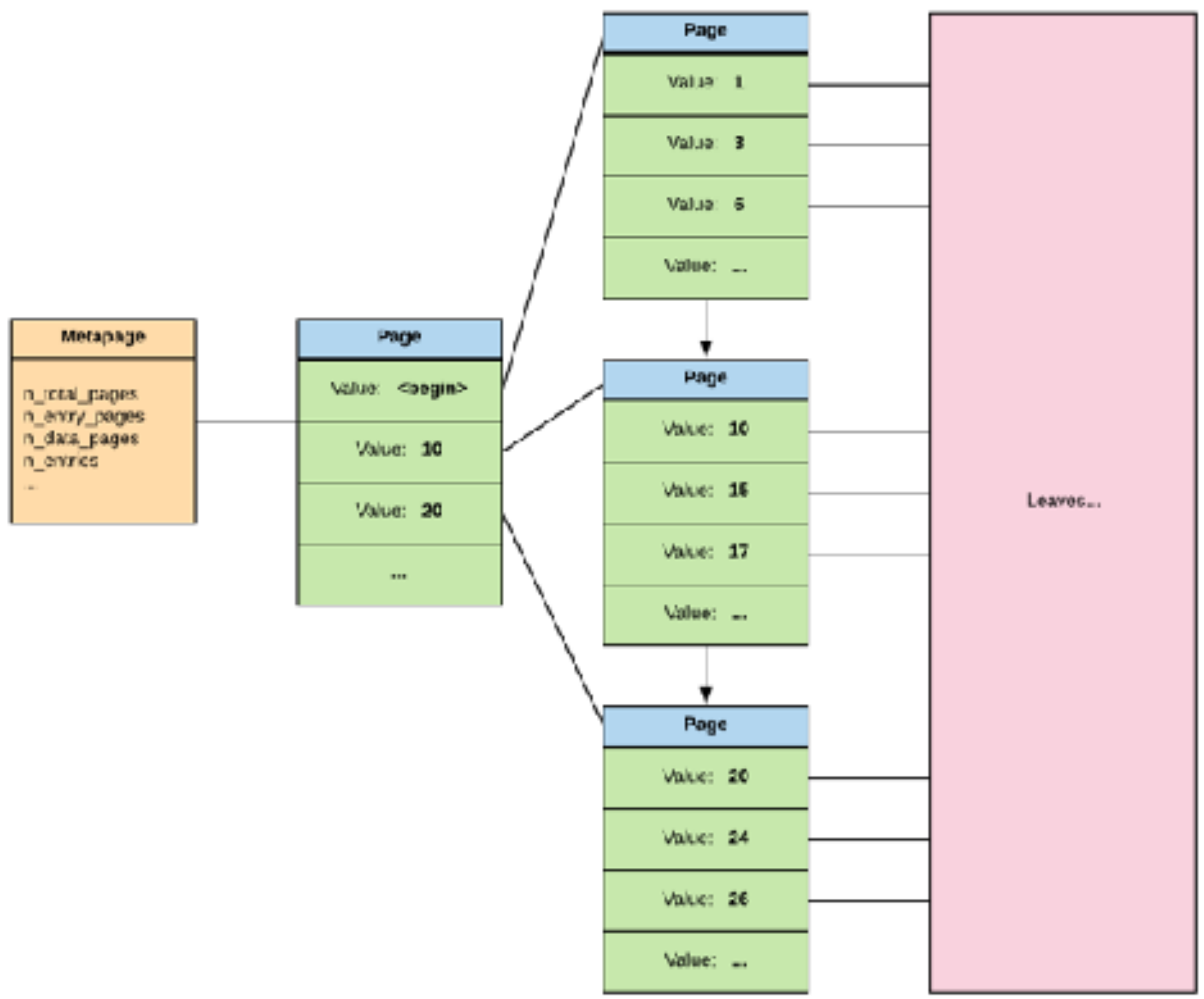
```
SELECT email FROM crocodile
WHERE ARRAY[1, 2] <@ healed_teeth;
```

```
Seq Scan on crocodile  (cost=…)
   Filter: ('{1,2}'::integer[] <@ healed_teeth)
   Rows Removed by Filter: 250728
 Planning time: 0.157 ms
 Execution time: 161.716 ms
(5 rows)
```

GIN

How is it different from a BTree? - Keys

- In a GIN index, the **array is split** and **each value is an entry**
- The **values are unique**



GIN

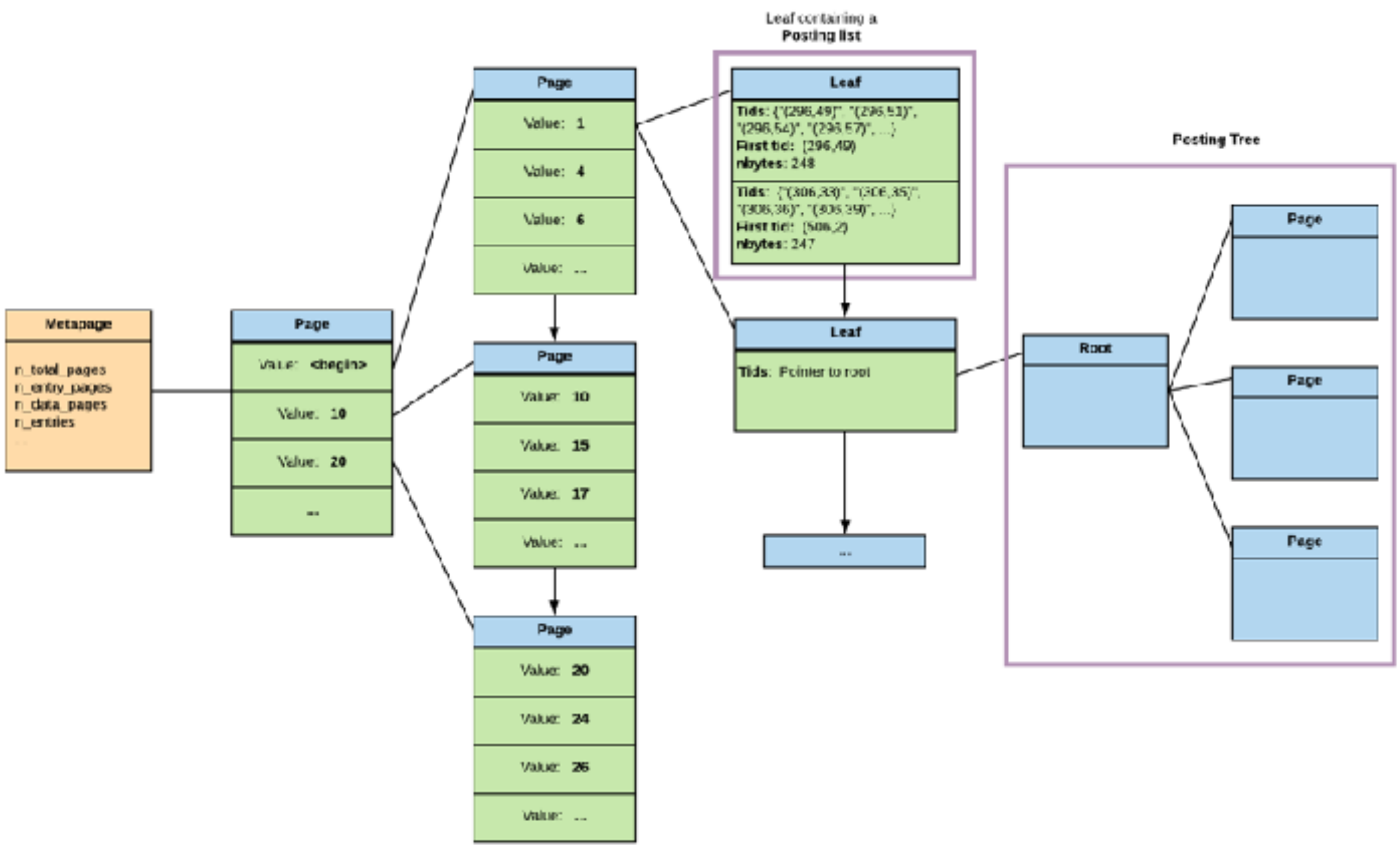
How is it different from a BTree? - Keys

```
Seq Scan on crocodile  (cost=…)
   Filter: ('{1,2}'::integer[] <@ healed_teeth)
   Rows Removed by Filter: 250728
 Planning time: 0.157 ms
 Execution time: 161.716 ms
(5 rows)
```

```
Bitmap Heap Scan on crocodile
(cost=516.59..6613.42 rows=54786 width=29)
(actual time=15.960..38.197 rows=73275 loops=1)
   Recheck Cond: ('{1,2}'::integer[] <@ healed_teeth)
   Heap Blocks: exact=4218
   ->  Bitmap Index Scan on crocodile_healed_teeth_idx
       (cost=0.00..502.90 rows=54786 width=0)
       (actual time=15.302..15.302 rows=73275 loops=1)
         Index Cond: ('{1,2}'::integer[] <@ healed_teeth)
 Planning time: 0.124 ms
 Execution time: 41.018 ms
(7 rows)
```

GIN
How is it different from a BTree? Leaves

- In a leaf page, the items contain a **posting list of pointers to the rows** in the table
- If **the list can't fit in the page**, it becomes a **posting tree**
- In the **leaf item** remains a **pointer to the posting tree**



GIN

How is it different from a BTree? Pending list

- To **optimise inserts**, we store the **new entries** in a **pending list** (linear list of pages)
- Entries are **moved to the main tree** on VACUUM or when the list is full
- You can disable the pending list by **setting `fastupdate` to false** (on CREATE or ALTER INDEX)

```
SELECT * FROM gin_metapage_info(get_raw_page('crocodile_healed_teeth_idx', 0));
-[ RECORD 1 ]----+-----------
pending_head     | 4294967295
pending_tail     | 4294967295
tail_free_size   | 0
n_pending_pages  | 0
n_pending_tuples | 0
n_total_pages    | 358
n_entry_pages    | 1
n_data_pages     | 356
n_entries        | 47
version          | 2
```

GIN
To sum it up

To sum up, a GIN index has:

- A **metapage**
- A **BTree of key entries**
- The **values are unique** in the main binary tree
- The **leaves** either contain a pointer to a **posting tree, or a posting list** of heap pointers
- New rows go into a **pending list** until it's full or VACUUM, that list needs to be scanned while searching the index

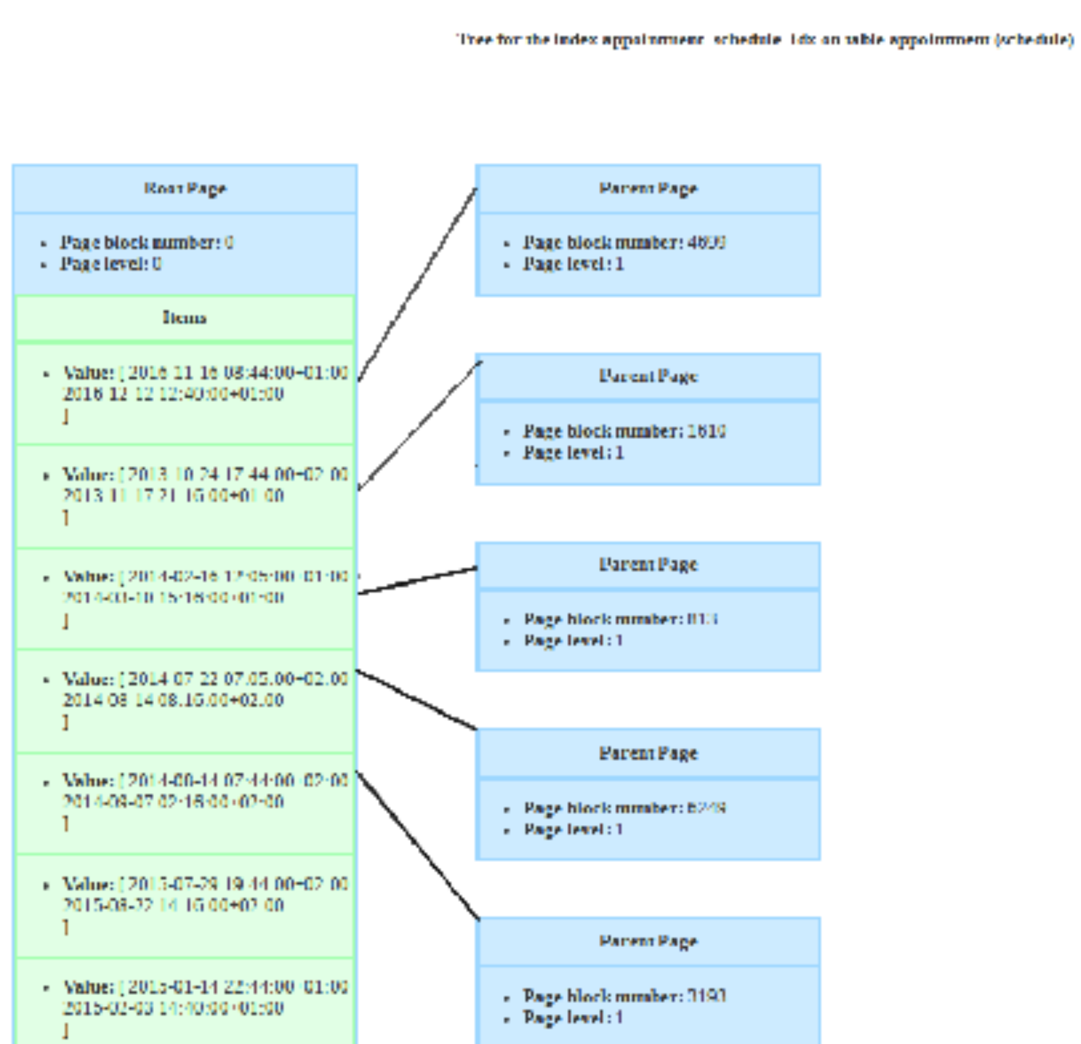
GIST

GiST - keys

Differences with a BTree index

- **Data isn't ordered**
- The **key ranges** can **overlap**

Which means that a same value **can be inserted in different pages**



GiST - keys

Differences with a BTree index

- **Data isn't ordered**
- The **key ranges** can **overlap**

Which means that a same value **can be inserted in different pages**

Items
• **Value:** [2016-11-16 08:44:00+01:00 2016-12-12 12:40:00+01:00]
• **Value:** [2013-10-24 17:44:00+02:00 2013-11-17 21:16:00+01:00]
• **Value:** [2014-02-16 12:05:00+01:00 2014-03-10 15:16:00+01:00]

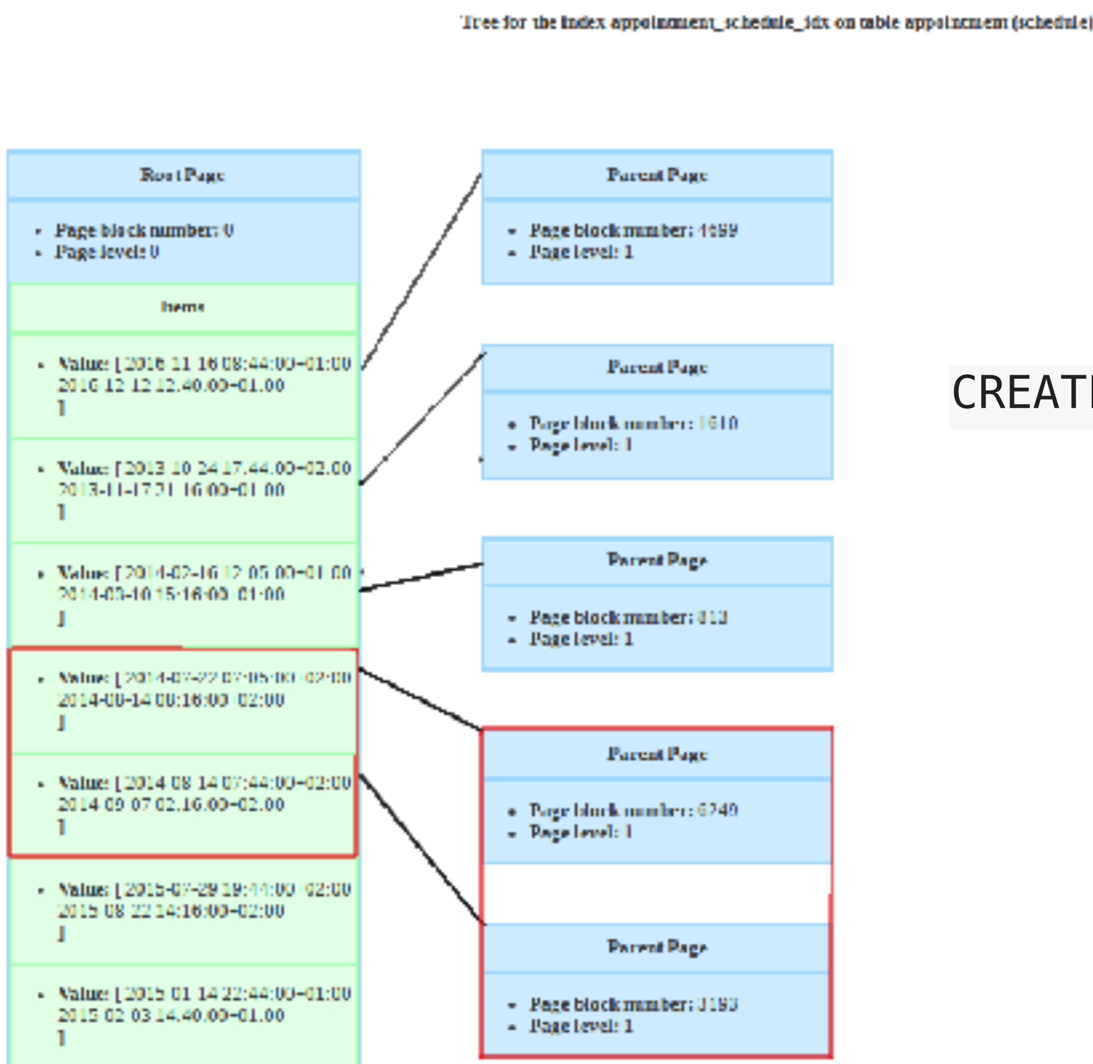
Data isn't ordered

GiST - keys

Differences with a BTree index

- **Data isn't ordered**
- The **key ranges** can **overlap**

Which means that a same value **can be inserted in different pages**




```
CREATE INDEX ON appointment USING GIST(schedule)
```

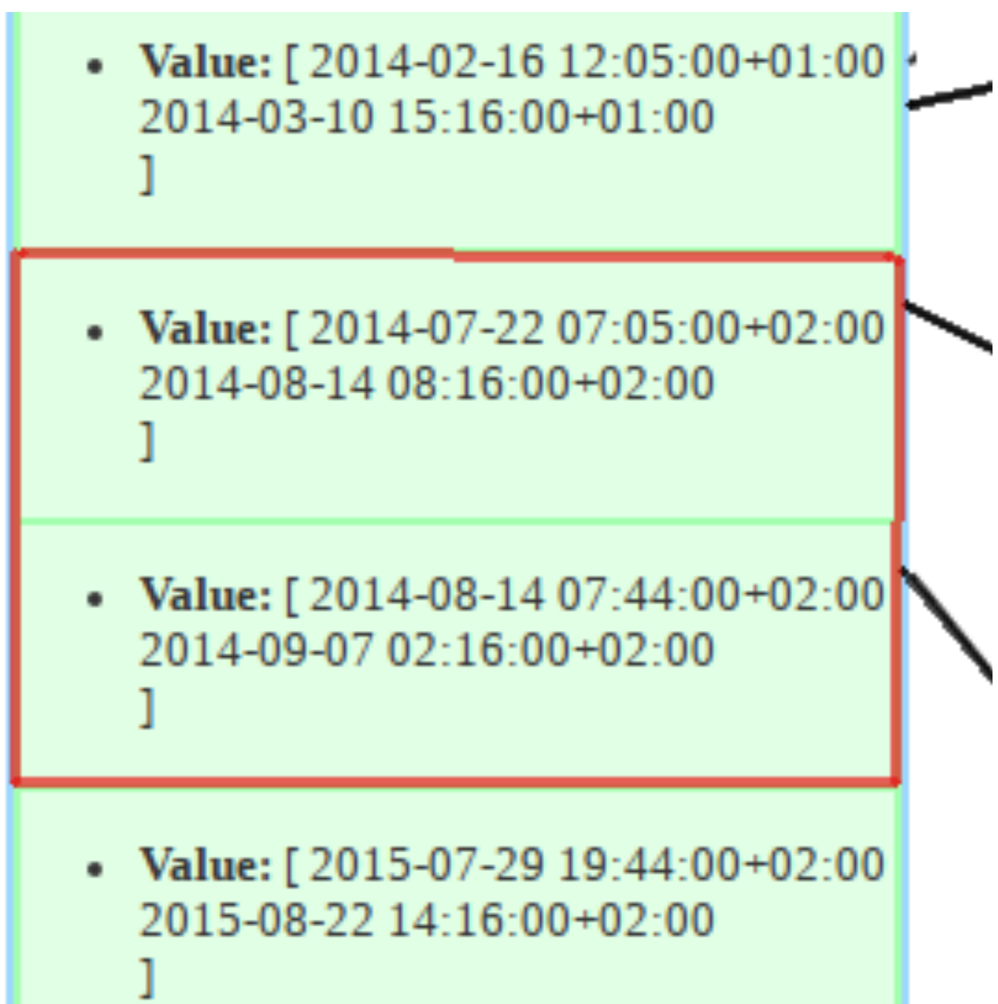
A new appointment scheduled from August 14th 2014 7:30am to 8:30am can be inserted in both pages.

GiST - keys

Differences with a BTree index

- **Data isn't ordered**
- The **key ranges** can **overlap**

Which means that a same value **can be inserted in different pages**




```
CREATE INDEX ON appointment USING GIST(schedule)
```

A new appointment scheduled from August 14th 2014 7:30am to 8:30am can be inserted in both pages.

GiST
key class functions

GiST allows the **development of custom data types** with the appropriate access methods.

These functions are **key class functions:**

Union: used while inserting, if the range changed

- **Value:** [2013-06-20 11:34:00+02:00
  2013-07-06 00:16:00+02:00
  ]

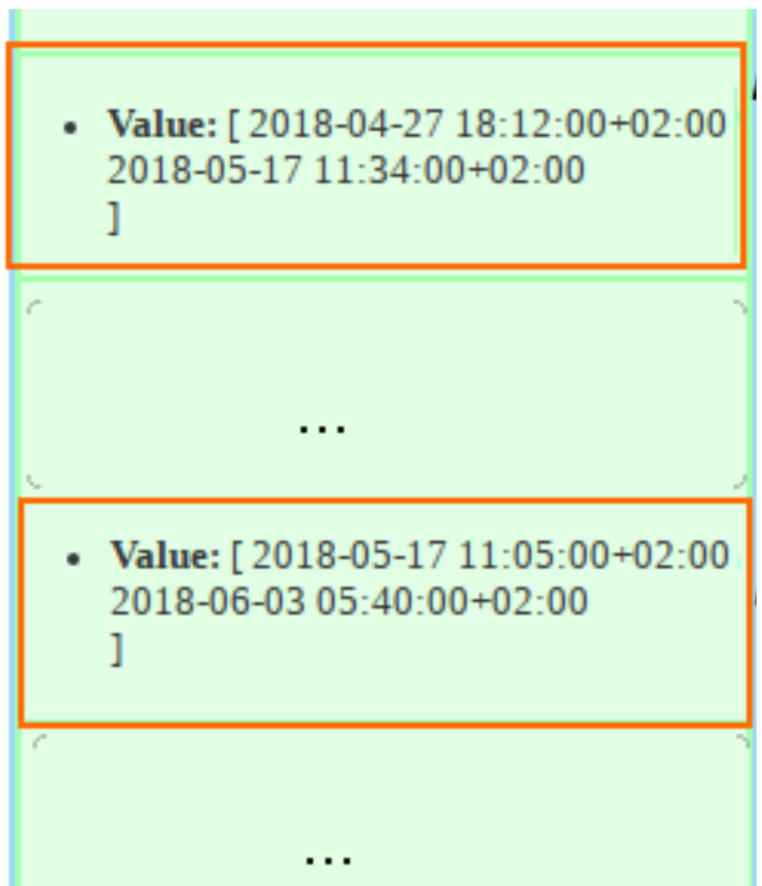
→

- **Value:** [2013-06-20 05:30:00+02:00
  2013-07-06 00:16:00+02:00
  ]

Distance: used for ORDER BY and nearest neighbor, calculates the **distance to the scan key**

GiST
key class functions - 2

Consistent: returns **MAYBE** if the **range contains the searched value**, meaning that rows could be in the page

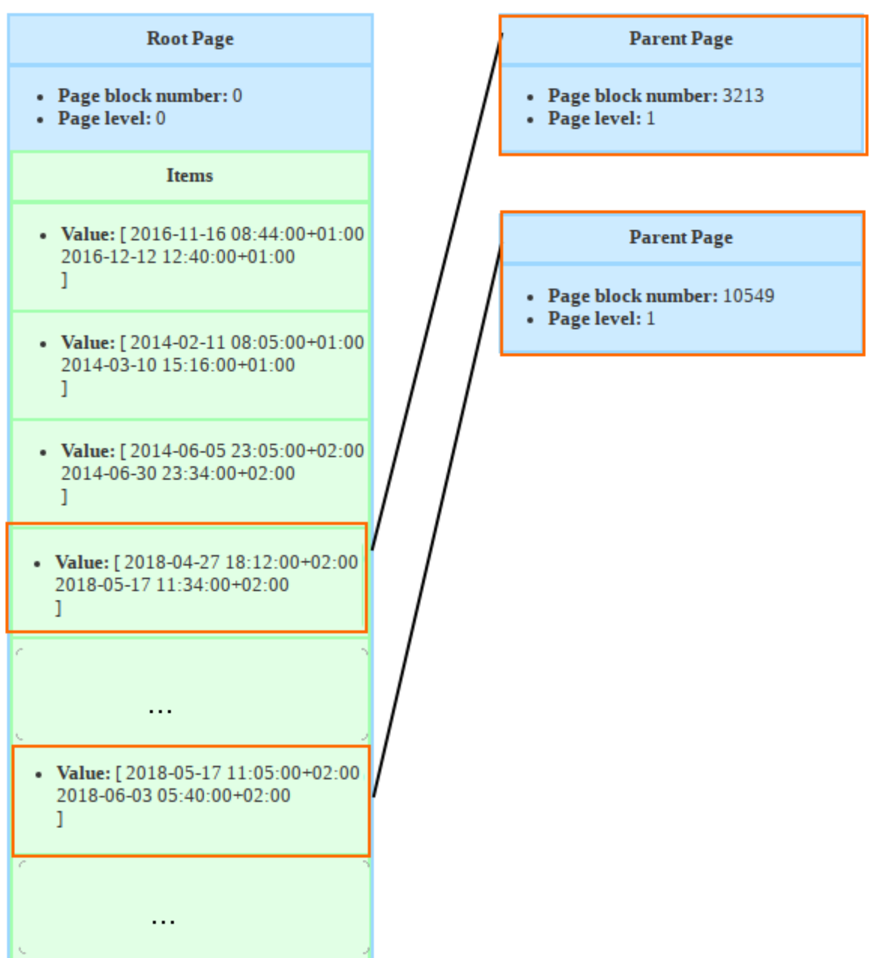


Child pages could contain the appointments overlapping [2018-05-17 08:00:00, 2018-05-17 13:00:00]

Consistent returns MAYBE

GiST - Searching

1. Create a **search queue** of **pages to explore** with the root in it
2. While the search queue isn't empty, pops a page
    1. If the **page is a leaf: update the bitmap** with CTIDs of rows
    2. Else**, adds to the search queue** the items where Consistent returned MAYBE




```
SELECT c.email, schedule, done, emergency_level
FROM appointment
INNER JOIN crocodile c ON (c.id=crocodile_id)
WHERE schedule && '[2018-05-17 08:00:00,
                    2018-05-17 13:00:00]'::tstzrange
      AND done IS FALSE
ORDER BY schedule DESC LIMIT 3;
```

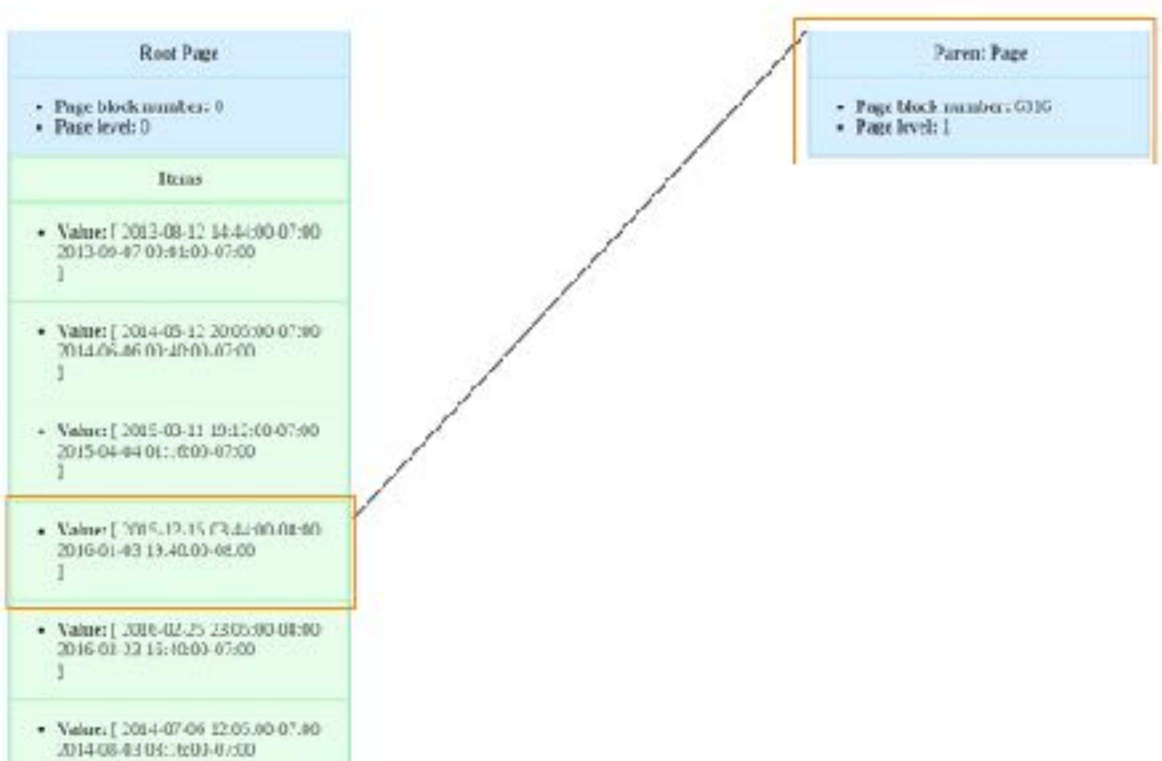
GiST - Inserting

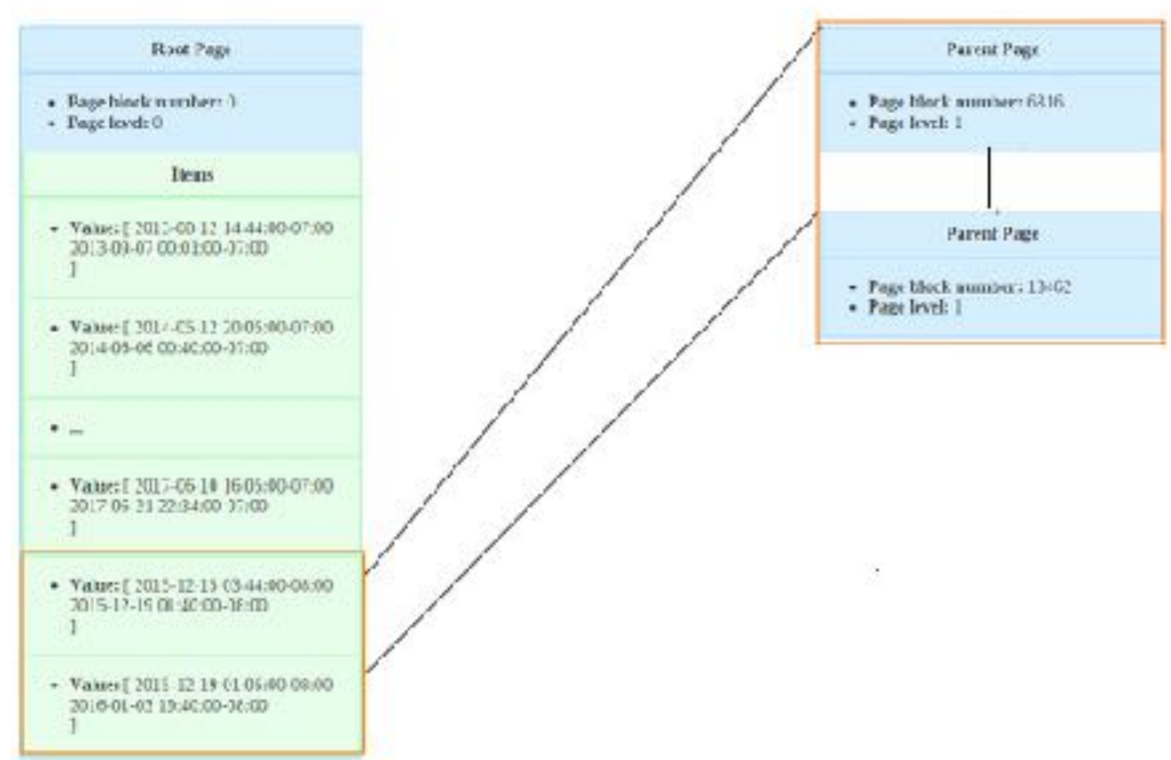
A **new item can be inserted in any page**.
Penalty: key class function (defined by user) gives a number representing **how bad it would be to insert the value in the child page**.

About page split:

Picksplit: makes groups with little distance

Performance of search will depend a lot of Picksplit





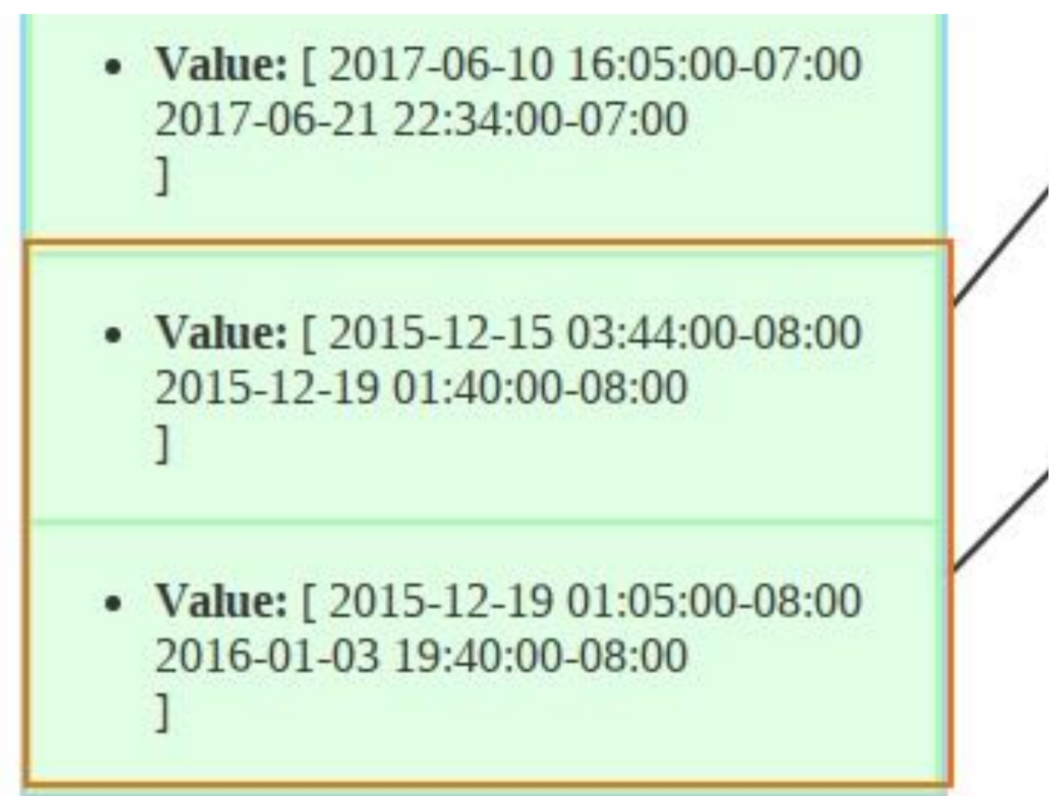
GiST - Inserting

A **new item can be inserted in any page**.
Penalty: key class function (defined by user) gives a number representing **how bad it would be to insert the value in the child page**.

About page split:

Picksplit: makes groups with little distance

Performance of search will depend a lot of Picksplit



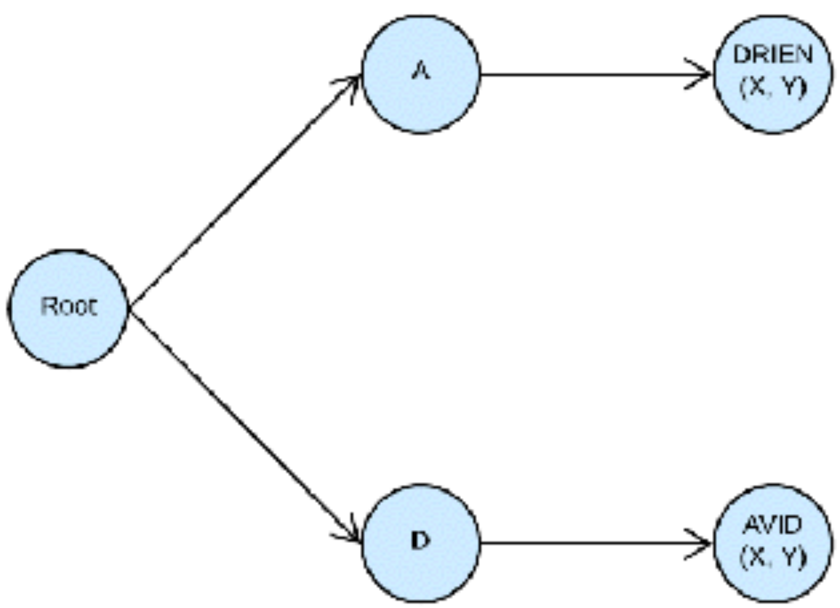
To sum up

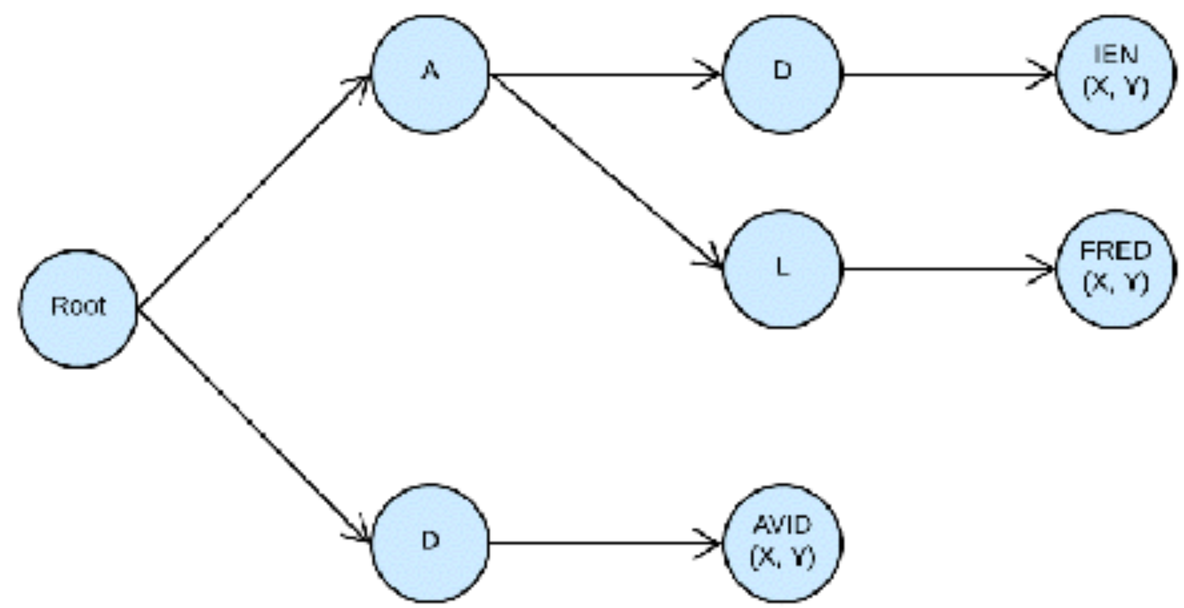
- Useful for **overlapping** (geometries, array etc.)
- **Nearest neighbor**
- Can be used for full text search (tsvector, tsquery)
- Any **data type can implement GiST** as long as a few methods are available

SP-GiST

SP-GiST
Internal data structure

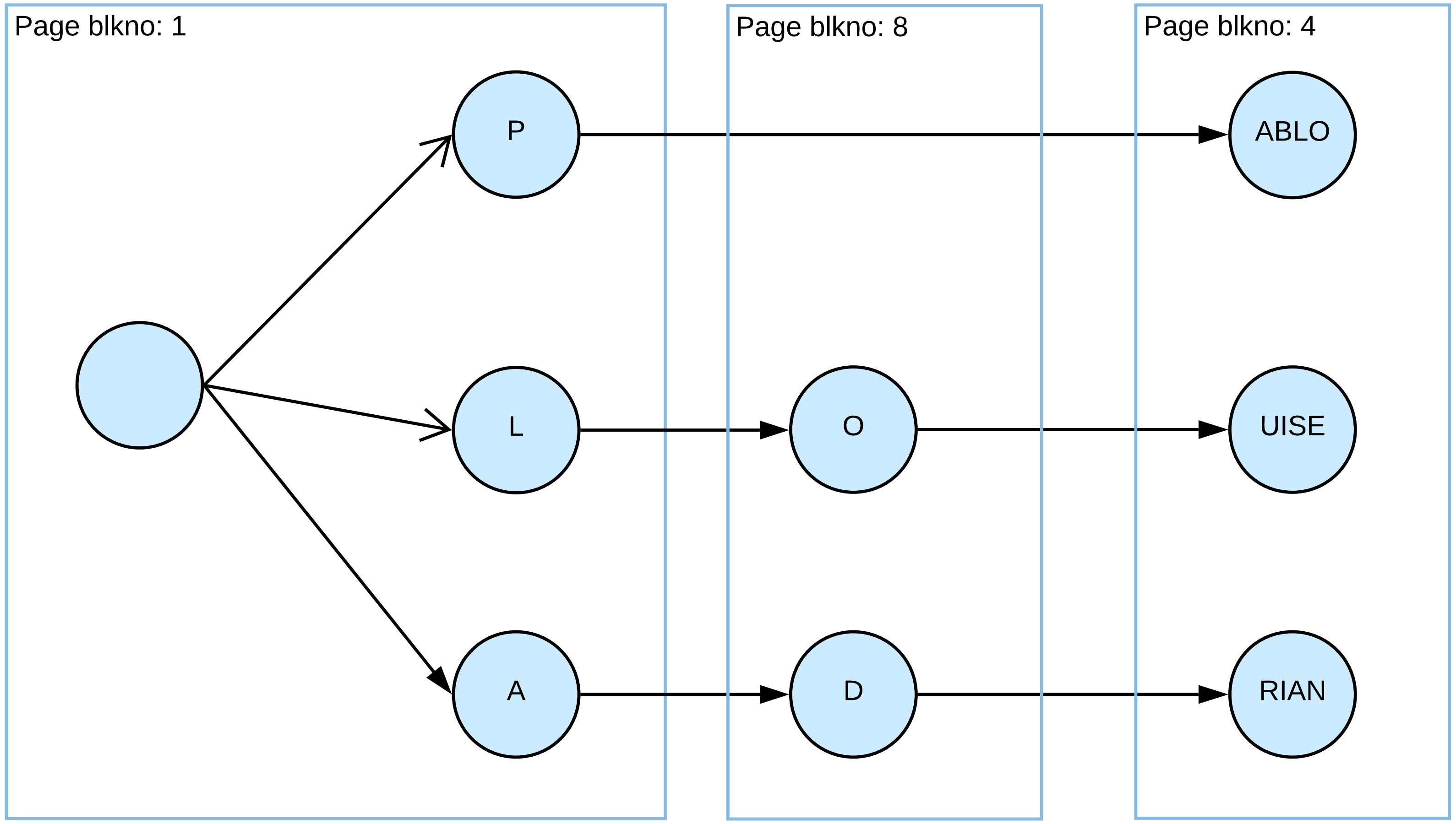
- **Not** a **balanced tree**
- A **same page can't have inner tuples and leaf tuples**
- Keys are decomposed
  - In an **inner tuple,** the **value** is the **prefix**
  - In a **leaf tuple**, the value is the rest (**postfix**)





SP-GiST
Pages

```
SELECT tid, level, leaf_value FROM spgist_print('crocodile_first_name_idx3') as t
       (tid tid, a bool, n int, level int, p tid, pr text, l smallint, leaf_value text) ;
 tid    | level | leaf_value
----------+-------+------------
 …
(4,36)   |     2 | ablo
(4,57)   |     2 | ustafa
(4,84)   |     3 | rian
(4,153)  |     3 | uise
…
```



Here are how the pages are organized if we look into gevel's sp-gist functions for this index

SP-GiST

- Can be used for **points**
- For text to search for prefix
- For non balanced data structures (k-d trees)
- **Like GiST:** allows the **development of custom data types**

BRIN

BRIN
Internal data structure

- Block Range Index
- Not a **binary tree**
- **Not even a tree**
- Block range: group of **pages physically adjacent**
- For each block range: **the range of values** is stored
- BRIN indexes are **very small**
- Fast scanning on large tables

BRIN
Internal data structure

```
SELECT * FROM brin_page_items(get_raw_page('appointment_created_at_idx', 2), 'appointment_created_at_idx');
 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |                       value
------------+--------+--------+----------+----------+-------------+--------------------------------------------------
          1 |      0 |      1 | f        | f        | f           | {2008-03-01 00:00:00-08 .. 2009-07-07 07:30:00-07}
          2 |    128 |      1 | f        | f        | f           | {2009-07-07 08:00:00-07 .. 2010-11-12 15:30:00-08}
          3 |    256 |      1 | f        | f        | f           | {2010-11-12 16:00:00-08 .. 2012-03-19 23:30:00-07}
          4 |    384 |      1 | f        | f        | f           | {2012-03-20 00:00:00-07 .. 2013-07-26 07:30:00-07}
          5 |    512 |      1 | f        | f        | f           | {2013-07-26 08:00:00-07 .. 2014-12-01 15:30:00-08}
```

```
SELECT id, created_at FROM appointment WHERE ctid='(0, 1)'::tid;
   id   |       created_at
--------+------------------------
 101375 | 2008-03-01 00:00:00-08
(1 row)
```

BRIN
Internal data structure

```
SELECT * FROM brin_page_items(get_raw_page('crocodile_birthday_idx', 2),
                              'crocodile_birthday_idx');
 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |          value
------------+--------+--------+----------+----------+-------------+--------------------------
          1 |      0 |      1 | f        | f        | f           | {1948-09-05 .. 2018-09-04}
          2 |    128 |      1 | f        | f        | f           | {1948-09-07 .. 2018-09-03}
          3 |    256 |      1 | f        | f        | f           | {1948-09-05 .. 2018-09-03}
          4 |    384 |      1 | f        | f        | f           | {1948-09-05 .. 2018-09-04}
          5 |    512 |      1 | f        | f        | f           | {1948-09-05 .. 2018-09-02}
          6 |    640 |      1 | f        | f        | f           | {1948-09-09 .. 2018-09-04}
          …
(14 rows)
```

In this case, the values in birthday has **no correlation** with the physical location, the index would **not speed up the search** as all pages would have to be visited.

BRIN is interesting for data where the **value is correlated with the physical location.**

BRIN
Warning on DELETE and INSERT

Deleted and then vacuum on the appointment table

```
DELETE FROM appointment  WHERE created_at >= '2009-07-07' AND created_at < ‘2009-07-08';

DELETE FROM appointment  WHERE created_at >= '2012-03-20' AND created_at < ‘2012-03-25';
```

New rows are inserted in the free space after VACUUM
BRIN index has some ranges with big data ranges.
Search will visit a lot of pages.

```
SELECT * FROM brin_page_items(get_raw_page('appointment_created_at_idx', 2), 'appointment_created_at_idx');
 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |                       value
------------+--------+--------+----------+----------+-------------+--------------------------------------------------
          1 |      0 |      1 | f        | f        | f           | {2008-03-01 00:00:00-08 .. 2018-07-01 07:30:00-07}
          2 |    128 |      1 | f        | f        | f           | {2009-07-07 08:00:00-07 .. 2018-07-01 23:30:00-07}
          3 |    256 |      1 | f        | f        | f           | {2010-11-12 16:00:00-08 .. 2012-03-19 23:30:00-07}
          4 |    384 |      1 | f        | f        | f           | {2012-03-20 00:00:00-07 .. 2018-07-06 23:30:00-07}
```

HASH

Hash
Internal data structure

- Only useful if you have a data not fitting into a page (8kb)
- Only operator is =
- If you use a PG version < 10, it's just awful

Conclusion

- B-Tree
  - Great for <, >, =, >=, <=
- GIN
  - Fulltext search, jsonb, arrays
  - ADD OPERATORS
  - Inserts can be slow because of unicity of the keys
- BRIN
  - Great for huge table with correlation between value and physical location
  - <, >, =, >=, <=
- GiST
  - Great for overlapping
  - Using key class functions
  - Can be implemented for any data type
- SP-Gist
  - Also using key class function
  - Decomposed keys
  - Can be used for non balanced data structures (k-d trees)
- Hash
  - If you have a value > 8kB
  - Only for =

Questions

Thanks for your attention

Go read the articles www.louisemeta.com

Now only the ones on BTrees are published,
but I'll announce the rest on twitter
@louisemeta

Crocodiles by https://www.instagram.com/zimmoriarty/?hl=en