BRIN improvements

Tomas Vondra, EDB, @fuzzycz
tomas.vondra@enterprisedb.com
pgcon 2023, May 30 - June 2

https://blog.pgaddict.com/

Why this talk?

- BRIN indexes are …
  - not sufficiently known / appreciated
  - tool many people don't know about
- interesting area for research / development
- pretty good place for new contributors
  - somewhat isolated part of code
  - plenty of space for heresy / experiments

Agenda

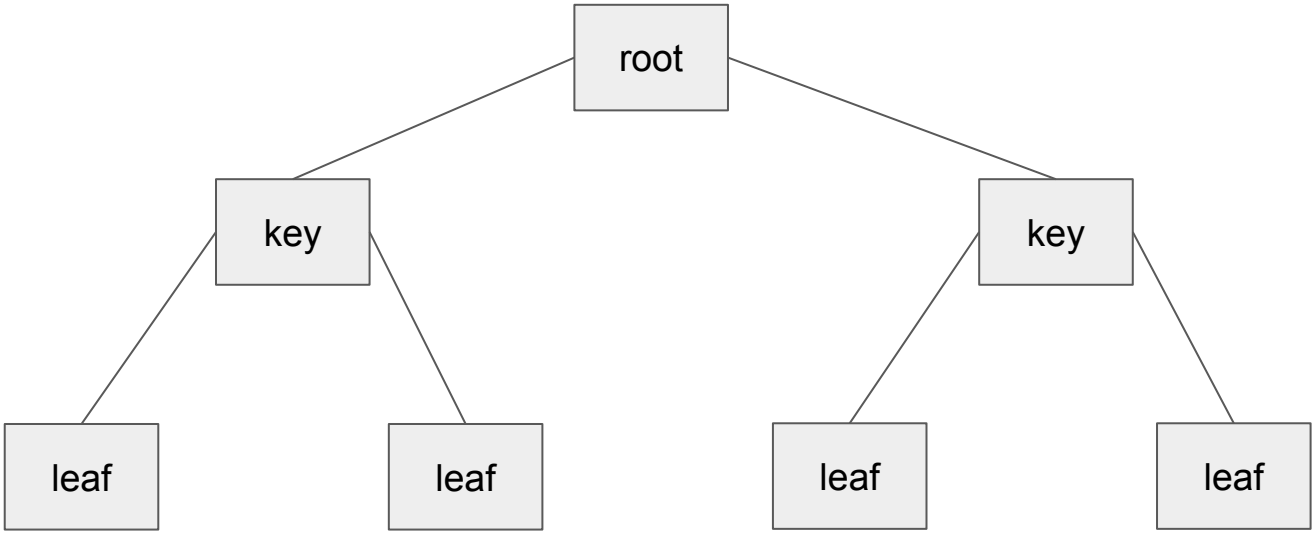
- What are BRIN indexes?
- Advantages and disadvantages
- PG14 & PG15 improvements
- Future improvements (ideas)

EDB

BTREE - traditional tree-like index

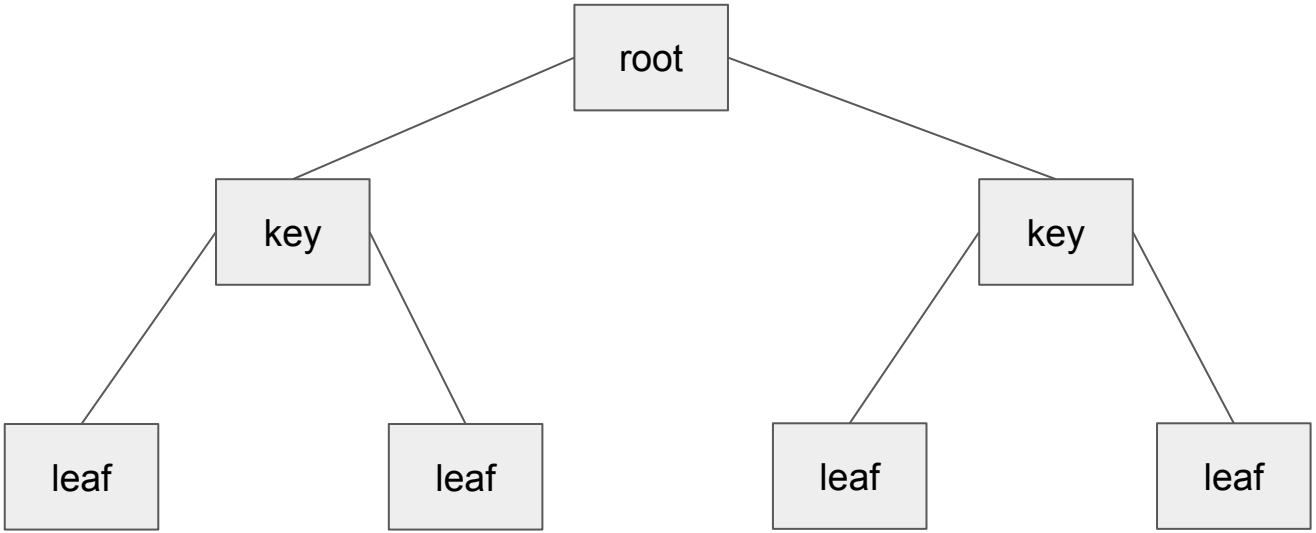
- 1:1 between rows and index entries
- organized in a tree
- great for "point queries", can do range queries
- allows ordering, uniqueness, covering indexes (INCLUDE)
- index scans, index only scans, bitmap index scans
- may get quite large

BTREE - classical tree-like index



EDB

BTREE - classical tree-like index



key => ctid (block, offset)
~ROWID

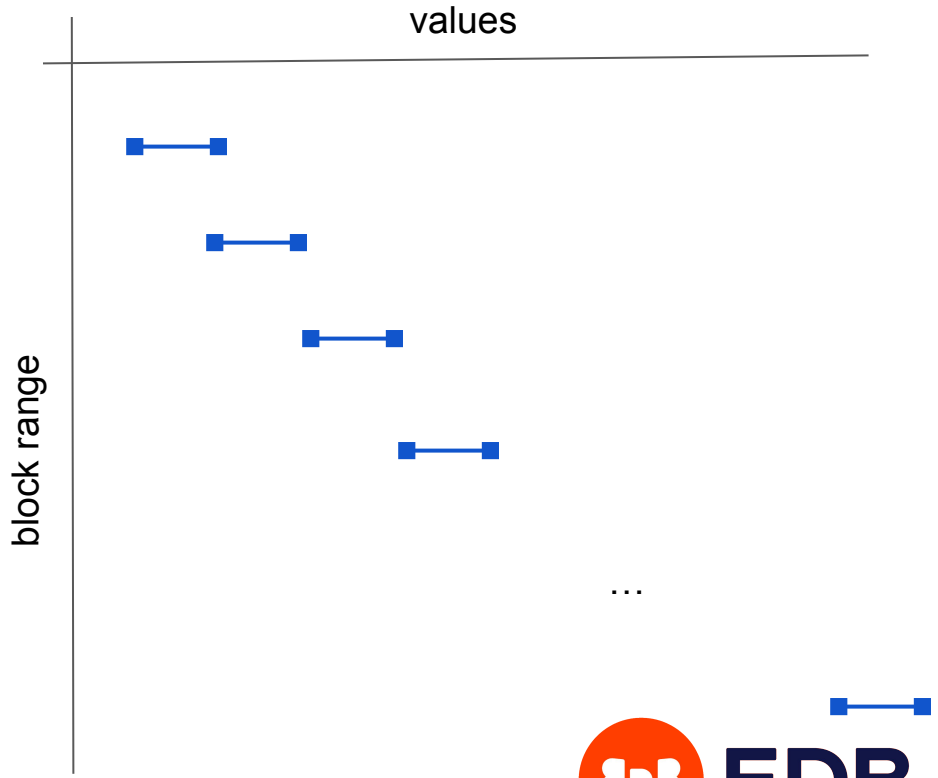
EDB

BRIN - block range index

- splits table into chunks (1MB default)
- stores small "summary" for each range (not per row)
  - min/max
  - inclusion (box, ipv4, range, …)
  - …
- bitmap index scans only
  - not great for point queries (more expensive than btree)
  - cache-friendly, access is more sequential

BRIN - block range index

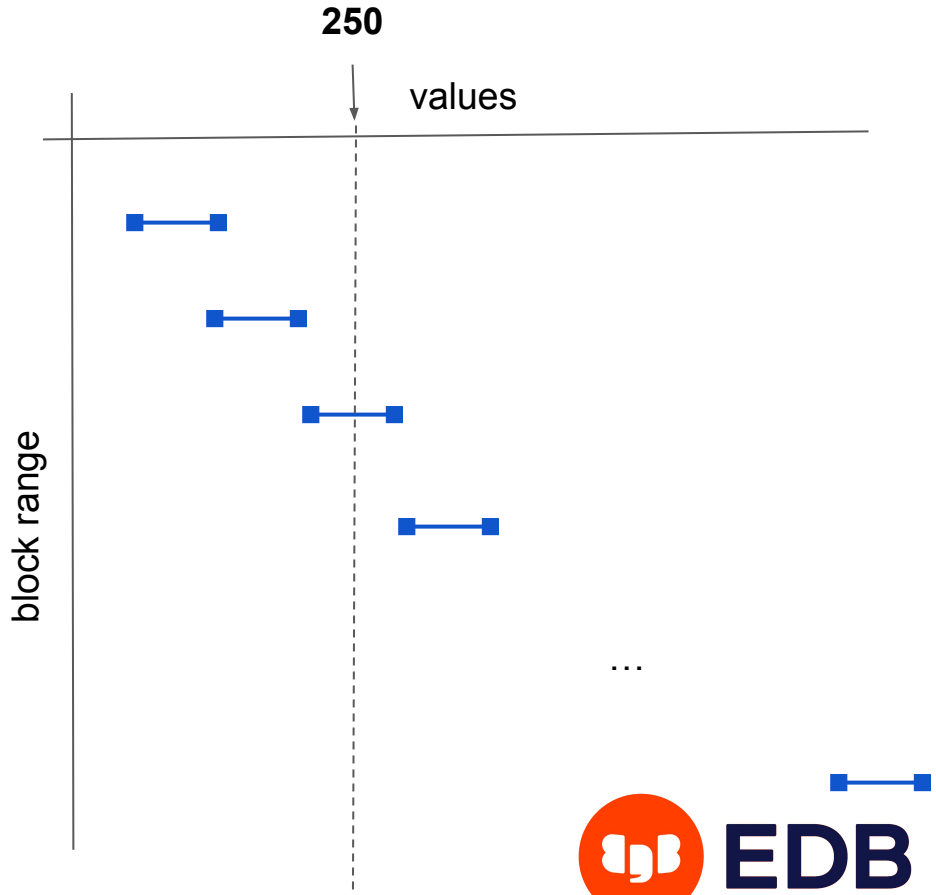
table	min/max
1MB	(1, 100)
1MB	(101, 200)
1MB	(201, 300)
1MB	(301, 400)
…	…
1MB	(901,1000)



EDB

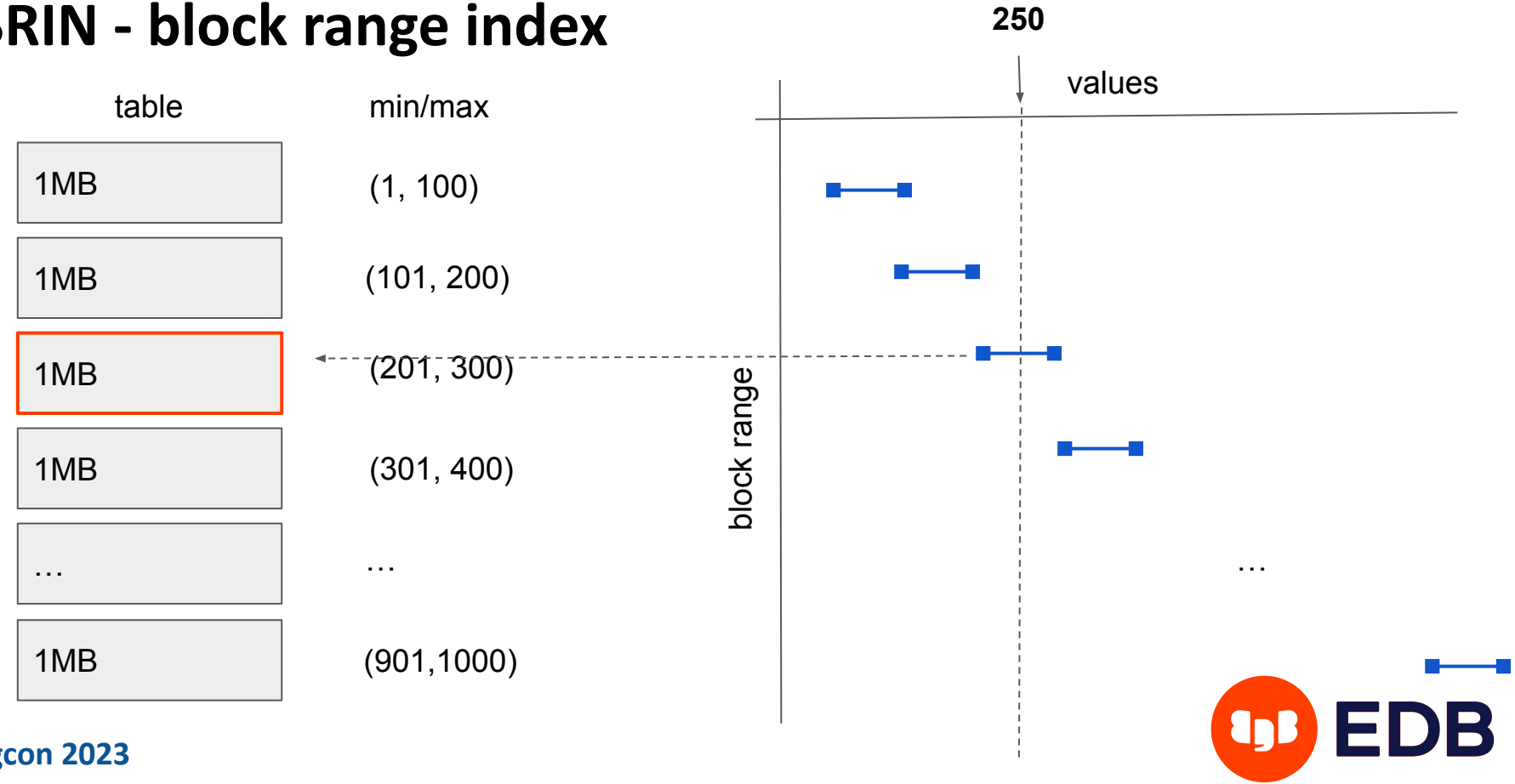
BRIN - block range index

table	min/max
1MB	(1, 100)
1MB	(101, 200)
1MB	(201, 300)
1MB	(301, 400)
…	…
1MB	(901,1000)



EDB

BRIN - block range index



EDB

Example

```
CREATE TABLE t (a BIGINT);

ALTER TABLE t SET (fillfactor = 10);

INSERT INTO t SELECT mod(i, 100000)
  FROM generate_series(1,10000000) s(i);

CREATE INDEX ON t USING BRIN (a);
```

Example

```
SELECT
  regexp_replace(ctid::text, '\((.*),.*\)', '\1')::int/128 AS page_range,
  min(a),
  max(a)
FROM t GROUP BY 1 ORDER BY 1;

 page_range |  min   |  max
------------+--------+-------
          0 |      1 |  2816
          1 |   2817 |  5632
          2 |   5633 |  8448
          3 |   8449 | 11264
          4 | 11265 | 14080
          5 | 14081 | 16896
          6 | 16897 | 19712
...
```


EDB

Example

```
CREATE EXTENSION pageinspect;

SELECT * FROM
  brin_page_items(get_raw_page('t_a_idx', 6), 't_a_idx') ORDER BY blknum;

 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |      value
------------+--------+--------+----------+----------+-------------+-----------------
        197 |      0 |      1 | f        | f        | f           | {1 .. 2816}
        198 |    128 |      1 | f        | f        | f           | {2817 .. 5632}
        199 |    256 |      1 | f        | f        | f           | {5633 .. 8448}
        200 |    384 |      1 | f        | f        | f           | {8449 .. 11264}
        201 |    512 |      1 | f        | f        | f           | {11265 .. 14080}
        202 |    640 |      1 | f        | f        | f           | {14081 .. 16896}
        203 |    768 |      1 | f        | f        | f           | {16897 .. 19712}
        204 |    896 |      1 | f        | f        | f           | {19713 .. 22528}
...
```

Example

```
test=# \d+
                                    List of relations
 Schema | Name | Type  | Owner | Persistence | Access method |  Size
--------+------+-------+-------+-------------+---------------+---------
 public | t    | table | user  | permanent   | heap          | 3552 MB
(1 row)                                                       ~450k pages (8K)

test=# \di+
                                        List of relations
 Schema |   Name    | Type  | Owner | Table | Persistence | Access method |  Size
--------+-----------+-------+-------+-------+-------------+---------------+--------
 public | t_a_idx   | index | user  | t     | permanent   |brin           | 160 kB
 public | t_a_idx1  | index | user  | t     | permanent   | btree         | 65 MB
(2 rows)
```


EDB

Example

```
SET max_parallel_workers_per_gather = 0;

EXPLAIN ANALYZE SELECT COUNT(*) FROM t WHERE a = 4000;

                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=571712.27..571712.28 rows=1 width=8)
            (actual time=2731.015..2731.020 rows=1 loops=1)
   ->  Bitmap Heap Scan on t  (cost=75.91..571712.03 rows=99 width=0)
                              (actual time=10.697..2730.815 rows=100 loops=1)
      Recheck Cond: (a = 4000)
      Rows Removed by Index Recheck: 560668
      Heap Blocks: lossy=25490   <- 5% of 450k pages
      ->  Bitmap Index Scan on t_a_idx  (cost=0.00..75.89 rows=1221990 width=0)
                                        (actual time=4.306..4.307 rows=254900 loops=1)
                Index Cond: (a = 4000)
 Planning Time: 0.119 ms
 Execution Time: 2731.060 ms
(9 rows)
```


EDB

Problems (minmax)

- requires correlation to efficient "elimination" of ranges
- great for timestamps / sequential IDs in append-only tables
- but may degrade over time (UPDATE / INSERT / DELETE)
- some data is naturally random (IP addresses, UUIDs, …)
- no correlation to even start with

Example

```
UPDATE t SET a = 0 WHERE random() < 0.01;
UPDATE t SET a = 99999 WHERE random() < 0.01;

EXPLAIN ANALYZE SELECT COUNT(*) FROM t WHERE a = 4000;

                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------
-
 Aggregate  (cost=314711.92..314711.93 rows=1 width=8)
            (actual time=27214.468..27214.472 rows=1 loops=1)
   ->  Bitmap Heap Scan on t  (cost=63.13..314711.66 rows=103 width=0)
                              (actual time=16.102..27214.261 rows=96 loops=1)
      Recheck Cond: (a = 4000)
      Rows Removed by Index Recheck: 9999904
      Heap Blocks: lossy=454546  <- 100% pages
      ->  Bitmap Index Scan on t_a_idx  (cost=0.00..63.11 rows=97383 width=0)
                                        (actual time=15.089..15.090 rows=4545460 loops=1)
                Index Cond: (a = 4000)
 Planning Time: 7.714 ms
 Execution Time: 27214.514 ms    <- seqscan would be ~5000 ms
(9 rows)                            (related to prefetching, increasing
                                     effective_io_concurrency would help)
```


EDB

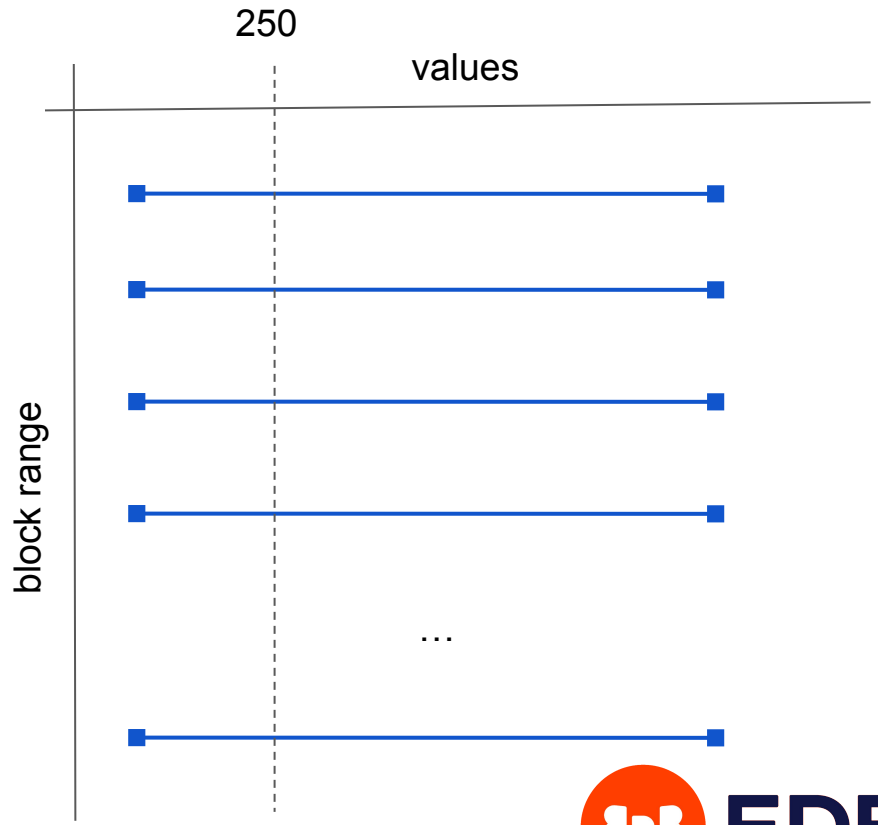
BRIN - example

```
SELECT * FROM
  brin_page_items(get_raw_page('t_a_idx', 6), 't_a_idx') ORDER BY blknum;

 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |    value
------------+--------+--------+----------+----------+-------------+--------------
        197 |      0 |      1 | f        | f        | f           | {0 .. 99999}
        198 |    128 |      1 | f        | f        | f           | {0 .. 99999}
        199 |    256 |      1 | f        | f        | f           | {0 .. 99999}
        200 |    384 |      1 | f        | f        | f           | {0 .. 99999}
        201 |    512 |      1 | f        | f        | f           | {0 .. 99999}
        202 |    640 |      1 | f        | f        | f           | {0 .. 99999}
        203 |    768 |      1 | f        | f        | f           | {0 .. 99999}
        204 |    896 |      1 | f        | f        | f           | {0 .. 99999}
        205 |   1024 |      1 | f        | f        | f           | {0 .. 99999}
        206 |   1152 |      1 | f        | f        | f           | {0 .. 99999}
...
```

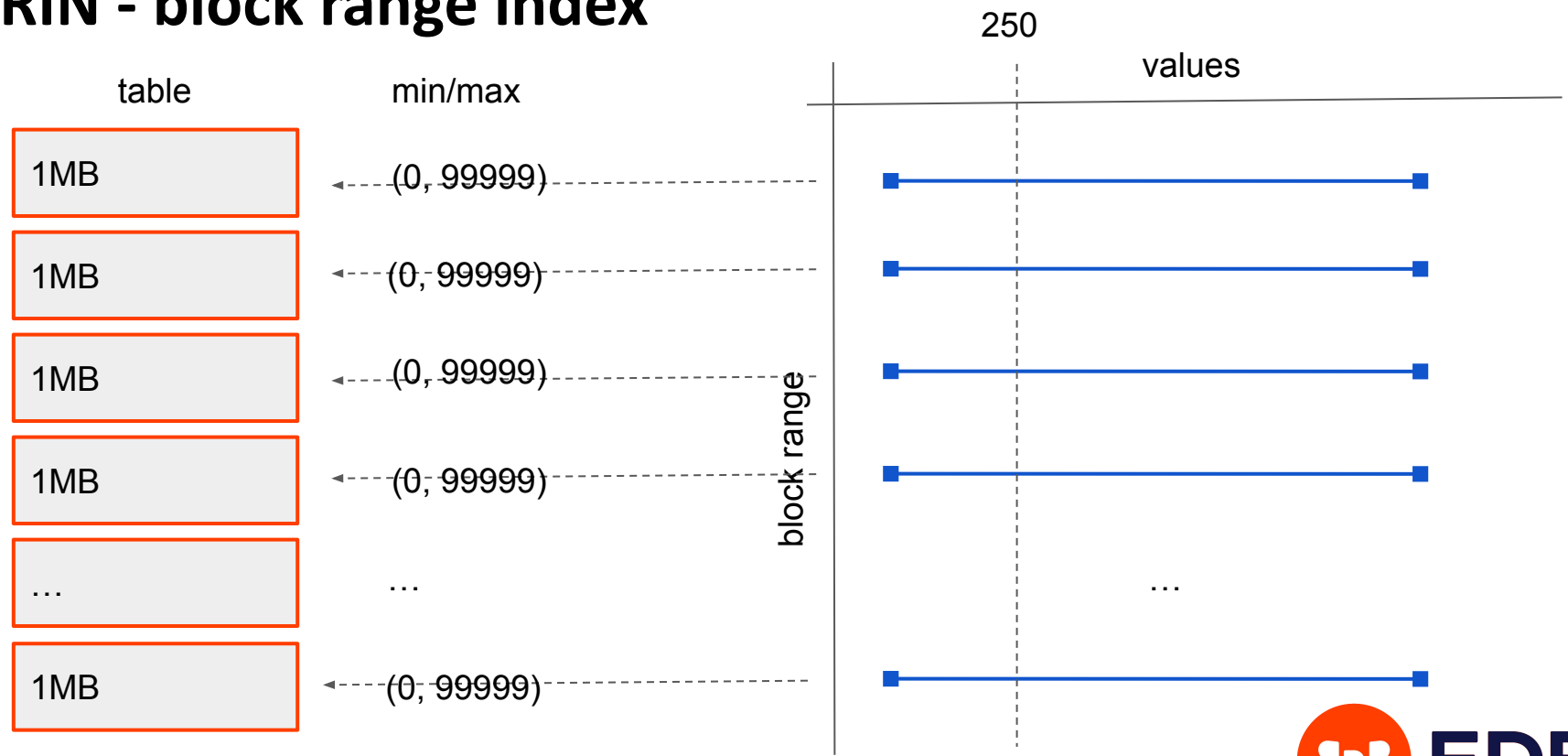
BRIN - block range index

table	min/max
1MB	(0, 99999)
1MB	(0, 99999)
1MB	(0, 99999)
1MB	(0, 99999)
…	…
1MB	(0, 99999)



EDB

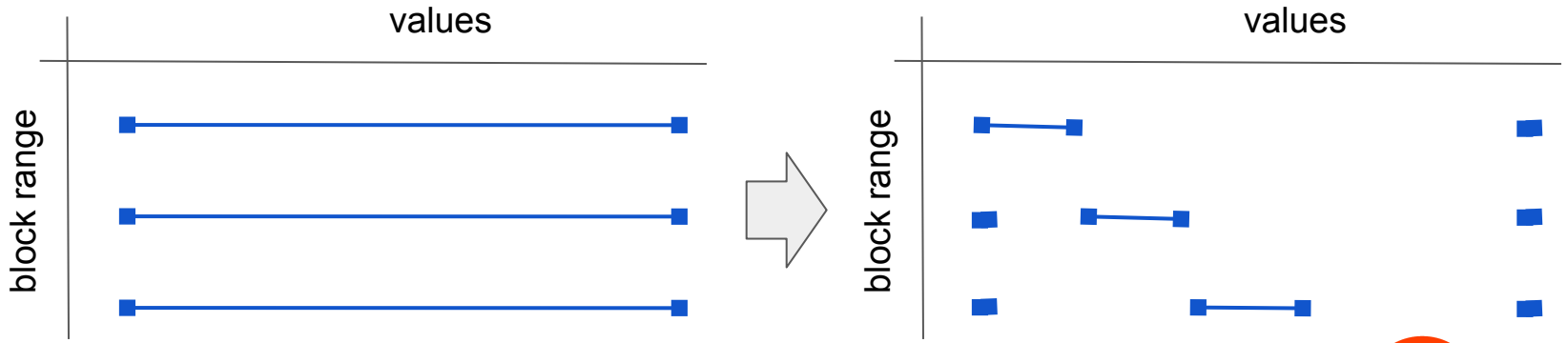
BRIN - block range index



EDB

PG14 improvements / minmax-multi

- keep multiple min/max ranges, not just a single one
- better in handling outliers / imperfectly correlated data



EDB

minmax-multi opclass

```
CREATE INDEX ON t USING BRIN (a  int8_minmax_multi_ops);

EXPLAIN ANALYZE SELECT COUNT(*) FROM t WHERE a = 4000;

                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------
 Aggregate  (cost=413832.55..413832.56 rows=1 width=8)
            (actual time=1560.740..1560.745 rows=1 loops=1)
   ->  Bitmap Heap Scan on t  (cost=503.82..413832.29 rows=104 width=0)
                              (actual time=14.972..1560.565 rows=96 loops=1)
      Recheck Cond: (a = 4000)
      Rows Removed by Index Recheck: 275872
      Heap Blocks: lossy=12544   <- 2.5%
      ->  Bitmap Index Scan on t_a_idx  (cost=0.00..503.79 rows=157693 width=0)
                                        (actual time=7.554..7.555 rows=125440 loops=1)
                Index Cond: (a = 4000)
 Planning Time: 6.368 ms
 Execution Time:  1562.068 ms
```


EDB

bloom opclass

- summarizes data into a bloom filter
- more suitable for naturally random data (ipv4, uuid)
- supports only equality searches

```
CREATE TABLE t (a UUID) WITH (fillfactor = 10);

INSERT INTO t SELECT md5(mod(i, 100000)::text)::uuid
  FROM generate_series(1,10000000) s(i);

CREATE INDEX ON t USING BRIN (a uuid_bloom_ops);
```


EDB

bloom opclass

```
EXPLAIN ANALYZE SELECT * FROM t WHERE a = 'f80fab2d-6a2f-65c2-1817-31623ee0993b';


                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on t  (cost=17382.86..707531.22 rows=99 width=16)
                        (actual time=49.958..905.123 rows=100 loops=1)
   Recheck Cond: (a = 'f80fab2d-6a2f-65c2-1817-31623ee0993b'::uuid)
   Rows Removed by Index Recheck: 230300
   Heap Blocks: lossy=12800
   ->  Bitmap Index Scan on t_a_idx  (cost=0.00..17382.84 rows=564230 width=0)
                                     (actual time=42.274..42.274 rows=128000 loops=1)
         Index Cond: (a = 'f80fab2d-6a2f-65c2-1817-31623ee0993b'::uuid)
 Planning Time: 0.074 ms
 Execution Time: 905.582 ms
(8 rows)
```


EDB

bloom opclass parameters

```
test=# \di+

                                    List of relations
 Schema |   Name    | Type  | Owner | Table | Persistence | Access method |  Size
--------+-----------+-------+-------+-------+-------------+---------------+-------
 public | t_a_idx   | index | user  | t     | permanent   | brin          | 34 MB
 public | t_a_idx1  | index | user  | t     | permanent   | btree         | 71 MB
(2 rows)

CREATE INDEX ON t USING BRIN (a uuid_bloom_ops (n_distinct_per_range=2500,
                                                false_positive_rate=0.05));

 Schema |   Name    | Type  | Owner | Table | Persistence | Access method |  Size
--------+-----------+-------+-------+-------+-------------+---------------+---------
 public | t_a_idx   | index | user  |     t | permanent   | brin          | 34 MB
 public | t_a_idx1  | index | user  |     t | permanent   | btree         | 71 MB
 public | t_a_idx2  | index | user  |     t | permanent   | brin          | 8752 kB
(3 rows)
```


EDB

Future improvements

- using BRIN (minmax) for sorting
  - should be pretty efficient for top-N sorts
  - might be better even for full sorts (lower memory requirement, no I/O)
  - works only for minmax (or ordering-based summaries)
- SK_SEARCHARRAY support
  - faster array queries - WHERE c IN (1, 2, 3, 4, …)
- CREATE INDEX parallelism

HOT updates (PG15)

- naive: update of column - modify all indexes on the table
  - bad: index updates = random I/O
- HOT = Heap-Only Tuples
  - optimization: update indexes only when updating indexed column
- PG15
  - we can go a bit further for BRIN indexes
  - updating column with a BRIN index -> update just BRIN indexes

Future improvements (?)

- retry insert (for large summaries)
  - index tuples have to be smaller than 8kB (no TOAST)
  - summaries can get too large (esp. for multi-column indexes)
  - inserts may fail unpredictably (pretty confusing for users)
  - maybe retry the insert automatically (or even discard the summary)?
- use BRIN to route inserts (maintain correlation)
  - maybe we could route new inserts to consistent ranges
  - what if there are multiple indexes? combine / pick one?

Future improvements (??)

- other types of summaries
  - false positives are OK (to some extent - size/efficiency trade-off)
- could we use BRIN to speed-up COUNT(*) on all-visible page ranges?
  - maybe, but what about grouping / WHERE conditions?

Stress testing

- regression tests
  - predefined order of steps
  - limited concurrency
- alternative: randomized stress testing
  - high-concurrency
  - randomized workload
  - check consistency (compare to seqscan)

Q & A