BRIN improvements

Tomas Vondra, EDB
tomas.vondra@enterprisedb.com
P2D2 2022, June 2

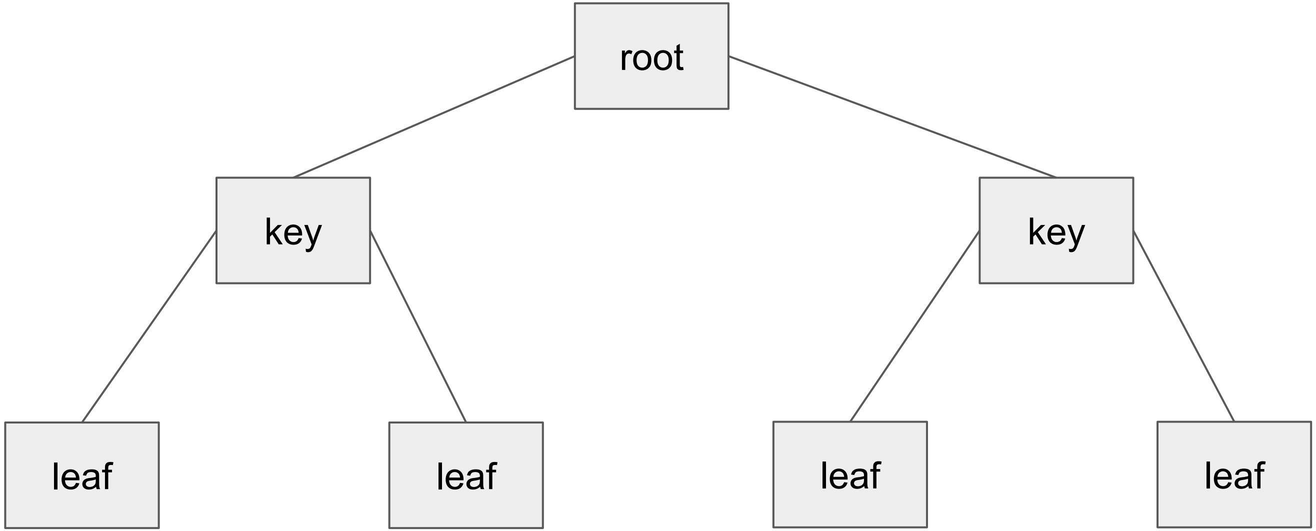
Agenda

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

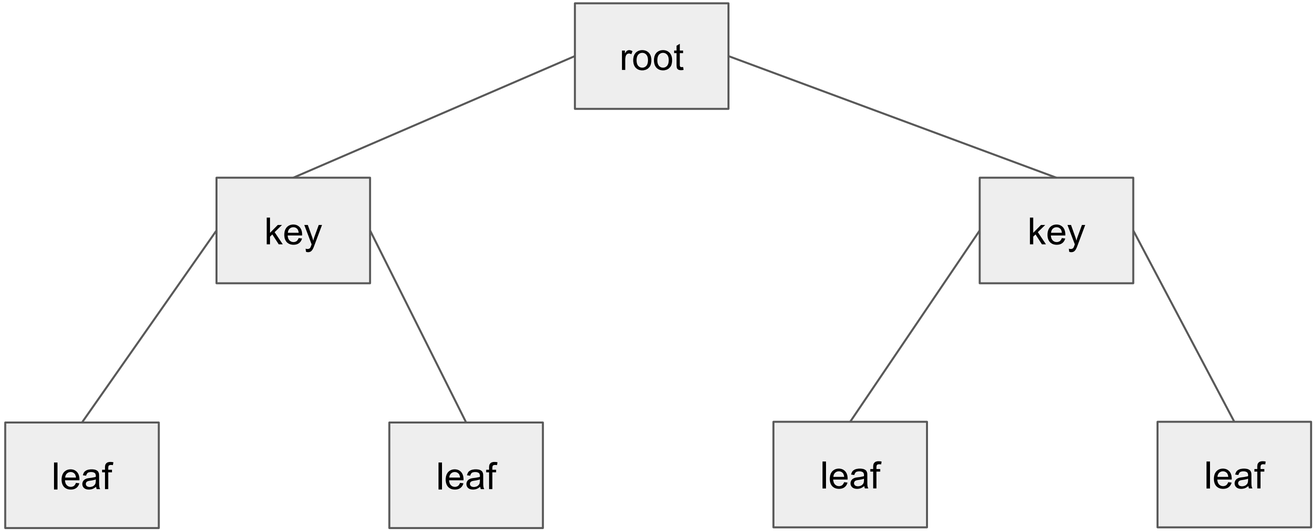
BTREE - traditional tree-like index

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

BTREE - classical tree-like index



BTREE - classical tree-like index



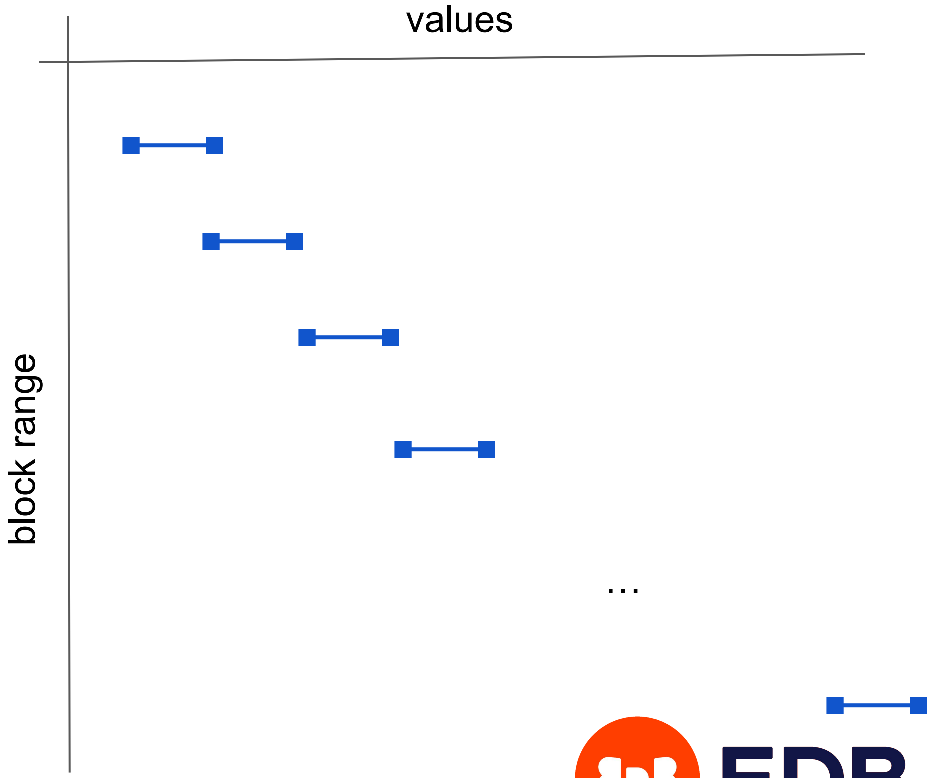
(key1, key2, …) => ctid
(block, offset)

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

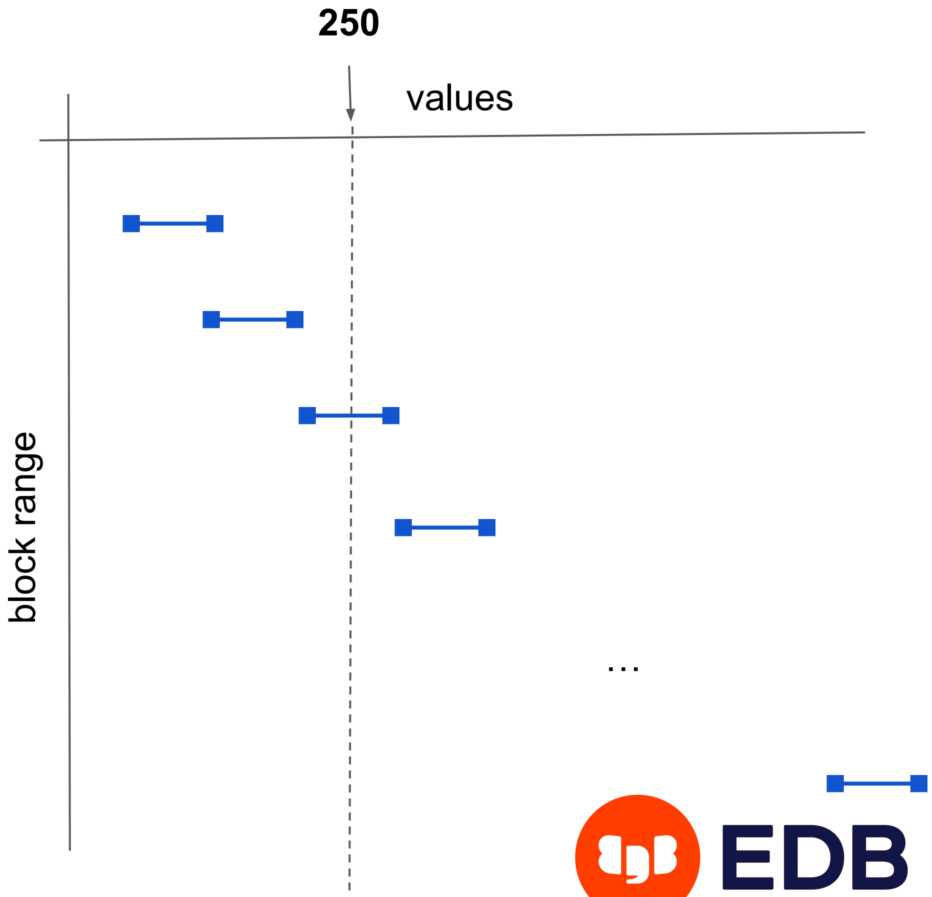
BRIN - block range index

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



BRIN - block range index

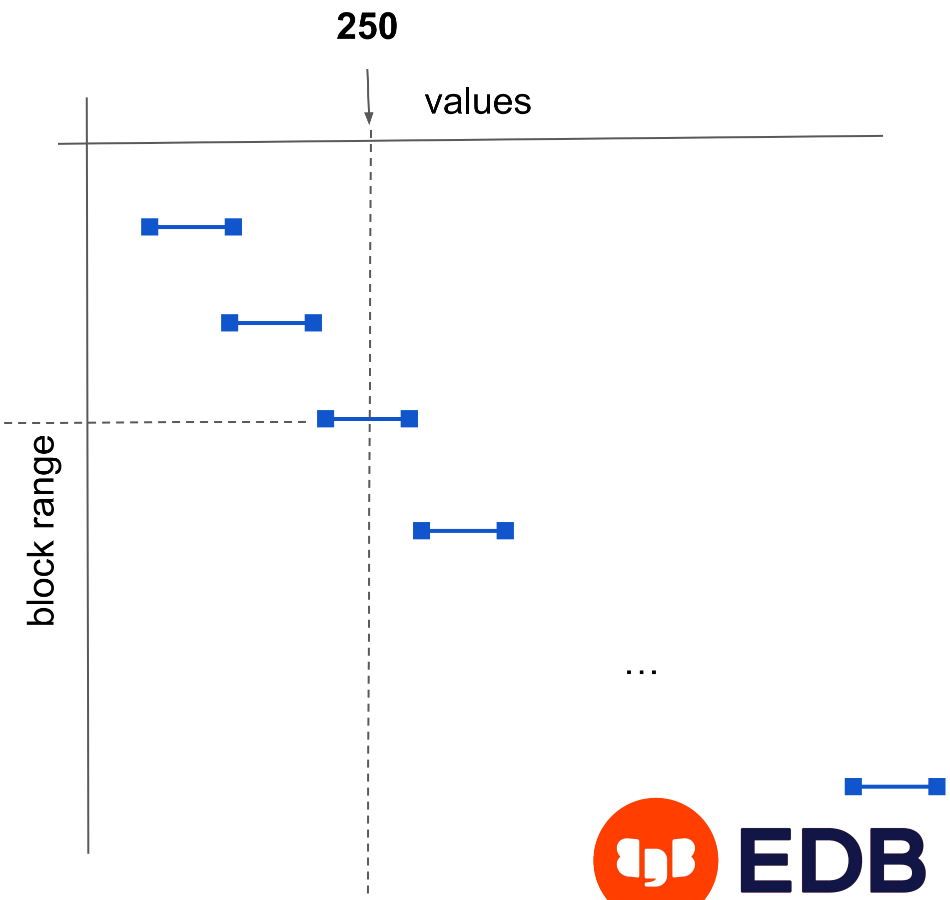
table	min/max
1MB	(1, 100)
1MB	(101, 200)
1MB	(201, 300)
1MB	(301, 400)
…	…
1MB	(901,1000)



EDB

BRIN - block range index

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



BRIN - 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);
```

BRIN - example

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

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)
```

BRIN - example

```
SET max_parallel_workers_per_gather = 0;

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

                                           QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate  (cost=468197.27..468197.28 rows=1 width=8)
            (actual time=107.061..107.065 rows=1 loops=1)
   ->  Bitmap Heap Scan on t  (cost=64.39..468197.02 rows=103 width=0)
                              (actual time=1.165..106.947 rows=100 loops=1)
     Recheck Cond: (a = 4000)
     Rows Removed by Index Recheck: 560668
     Heap Blocks: lossy=25490   <- 5%
     ->  Bitmap Index Scan on t_a_idx  (cost=0.00..64.36 rows=207603 width=0)
                                        (actual time=0.904..0.905 rows=254900 loops=1)
               Index Cond: (a = 4000)
 Planning Time: 0.052 ms
 Execution Time: 107.094 ms
(9 rows)
```


EDB

BRIN - problems

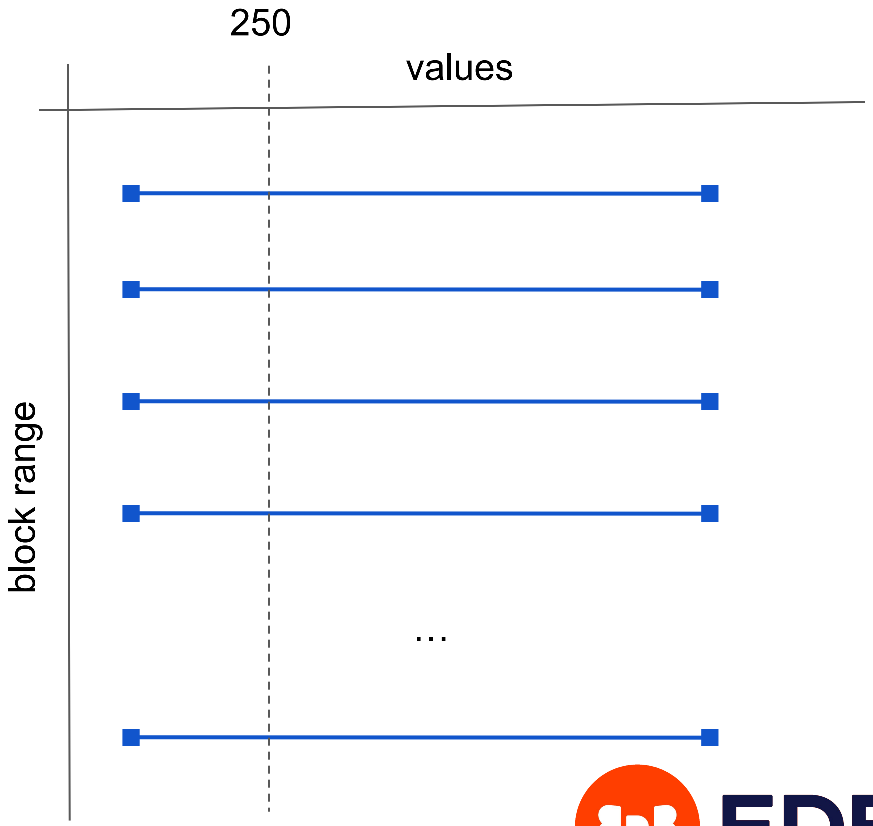
- requires correlation to efficient "elimination" of ranges
- gread for timestamps / sequential IDs in appen-only tables
- correlation may degrade over time (UPDATE / INSERT / DELETE)
- some data is naturally random (IP addresses, UUIDs, ...)

BRIN - example

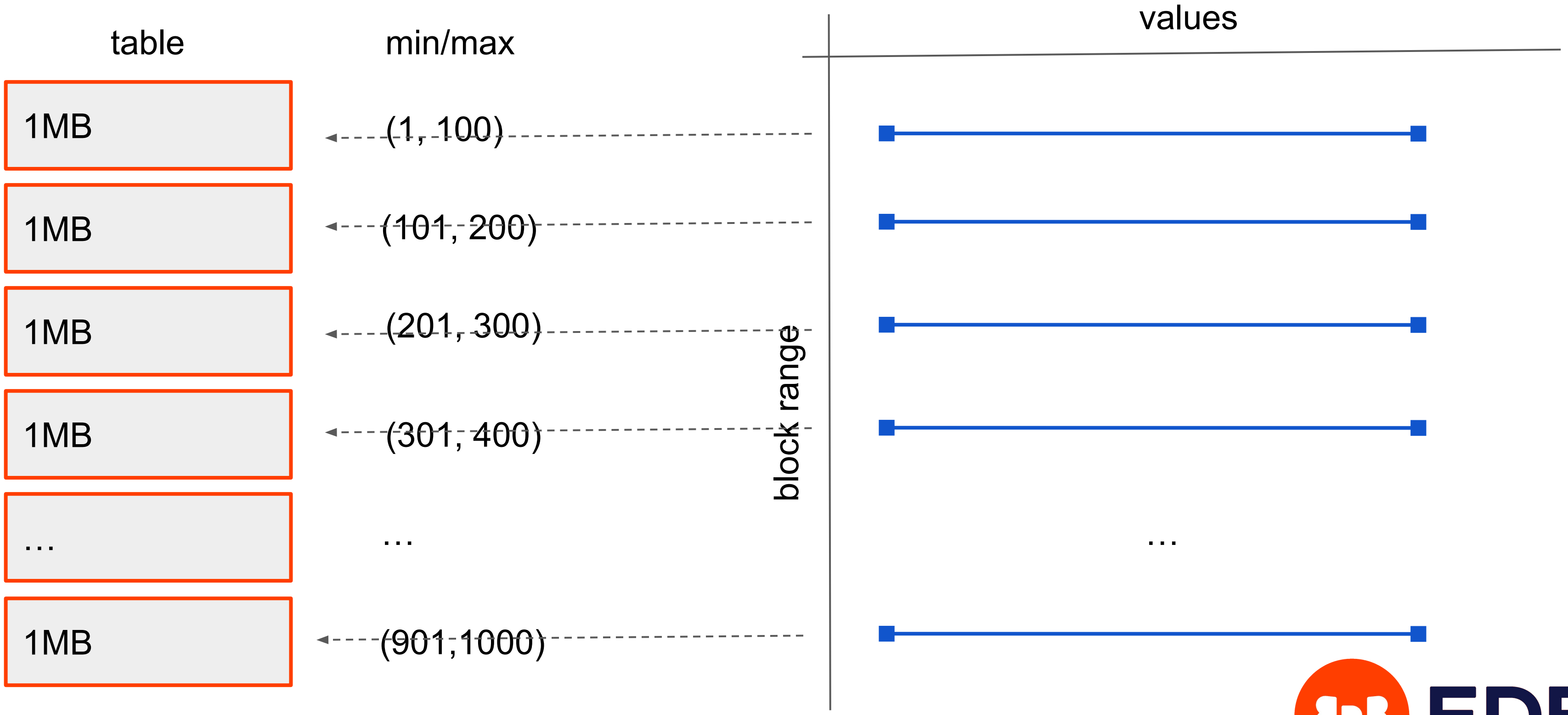
```
UPDATE t SET a = 0 WHERE random() < 0.01;
UPDATE t SET a = 99999 WHERE random() < 0.01;

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%
      ->  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 ~5000 ms
(9 rows)
```

BRIN - block range index

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

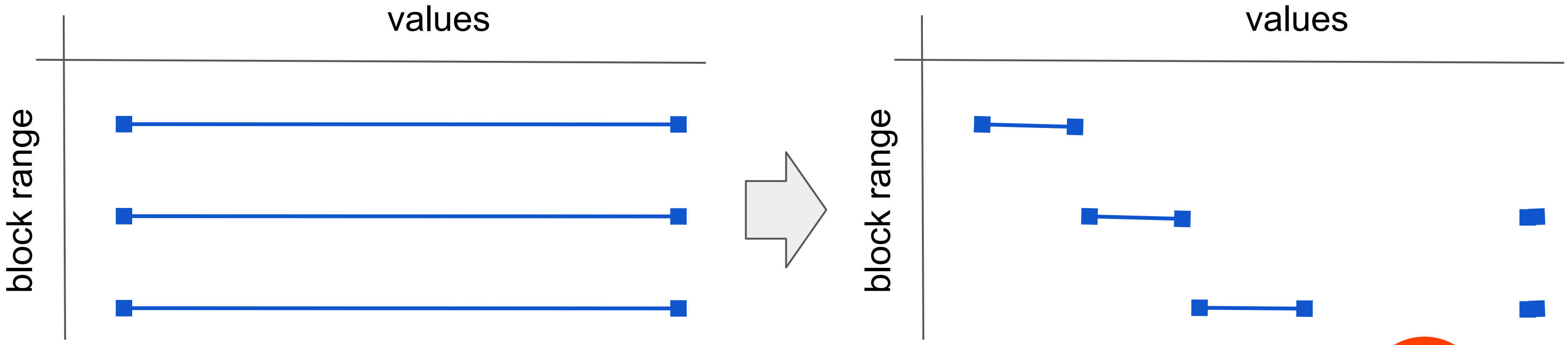


BRIN - block range index



PG14 improvements / minmax-multi

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



EDB

BRIN - minmax-multi

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

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

                                              QUERY PLAN
------------------------------------------------------------------------------------
 Aggregate  (cost=10000576545.84..10000576545.85 rows=1 width=8)
            (actual time=564.053..564.057 rows=1 loops=1)
   ->  Bitmap Heap Scan on t  (cost=10000000506.84..10000576545.58 rows=103 width=0)
                              (actual time=4.922..563.916 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..506.81 rows=423370 width=0)
                                       (actual time=3.384..3.385 rows=125440 loops=1)
                Index Cond: (a = 4000)
 Planning Time: 0.074 ms
 Execution Time: 564.109 ms
(9 rows)
```

PG14 improvements / bloom

- 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);
```

BRIN - bloom

```
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)
```

BRIN - bloom

```
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

- 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?
- 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)?

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)
- speed-up COUNT(*) - could it work for all-visible pages?
  - problem: grouping / WHERE conditions
- other types of summaries
  - false positives are OK

Q & A