BRIN improvements

Tomas Vondra, EDB, @fuzzycz
tomas.vondra@enterprisedb.com
pgday UK 2023, September 12

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

EDB

Agenda

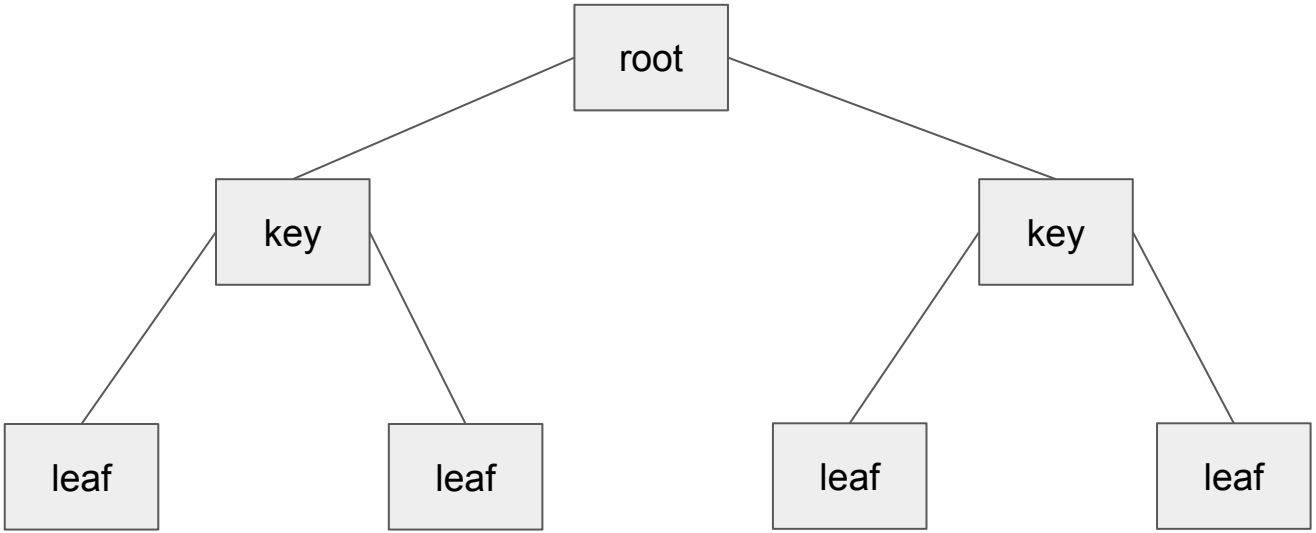
- What are BRIN indexes?
- Advantages and disadvantages
- PG14-16 improvements
- Future improvements (PG17?)
- Maybe some ideas for hacking

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



key => ctid (block, offset)
~ROWID

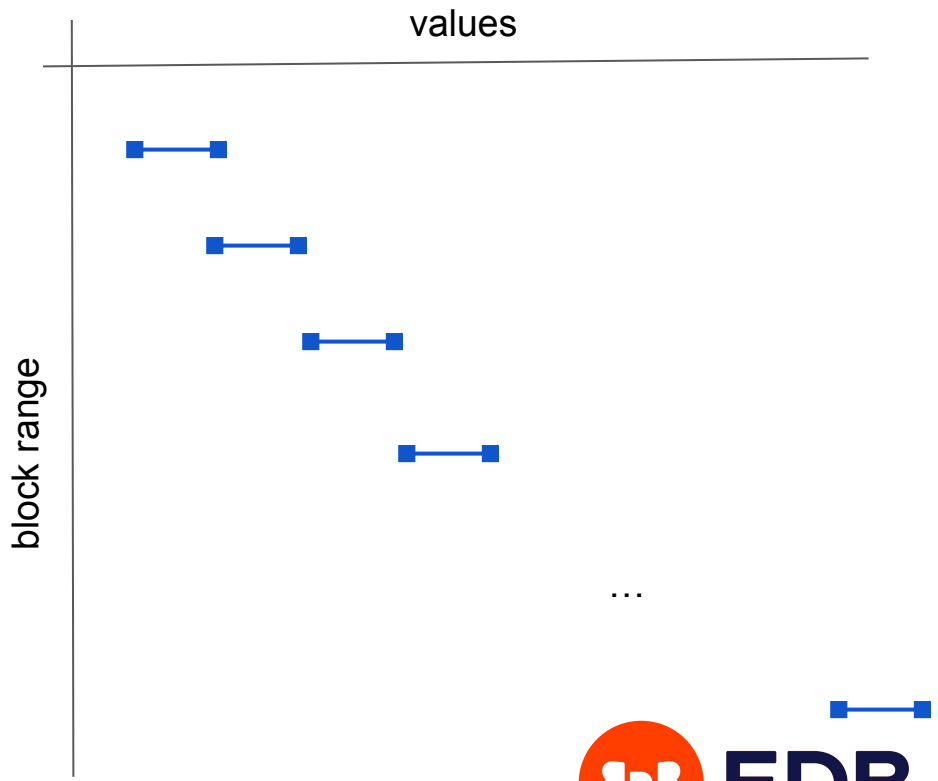
EDB

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

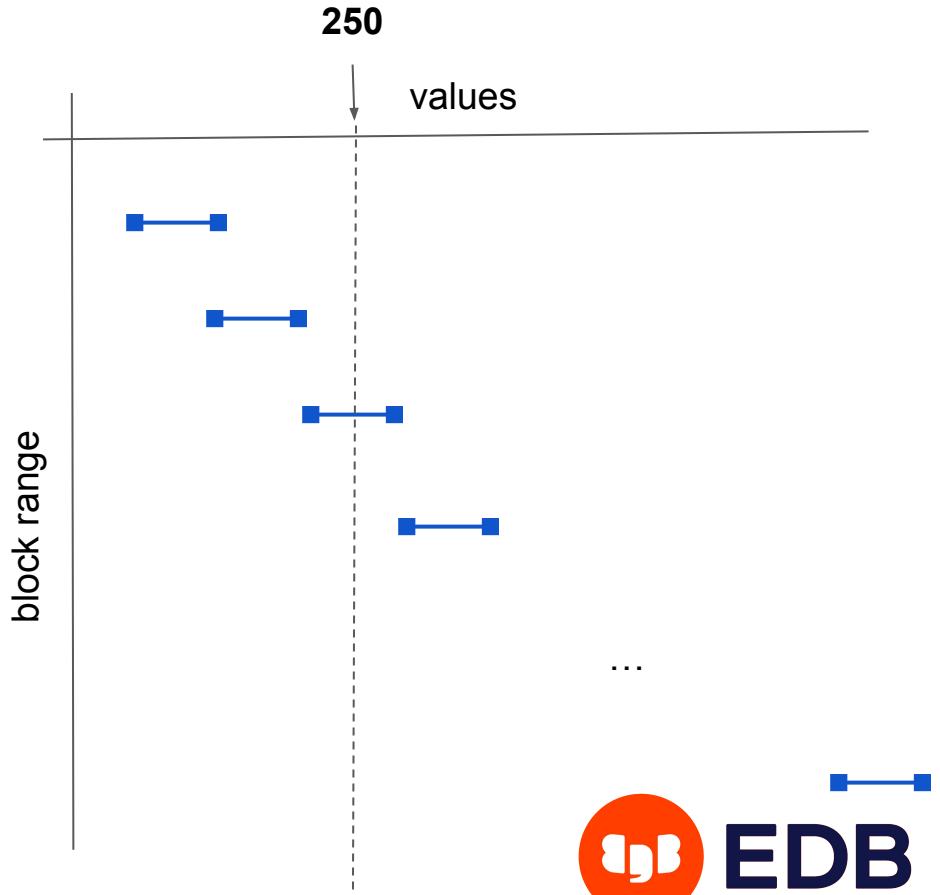
table	min/max
1MB	(1, 100)
1MB	(101, 200)
1MB	(201, 300)
1MB	(301, 400)
…	…
1MB	(901,1000)



EDB

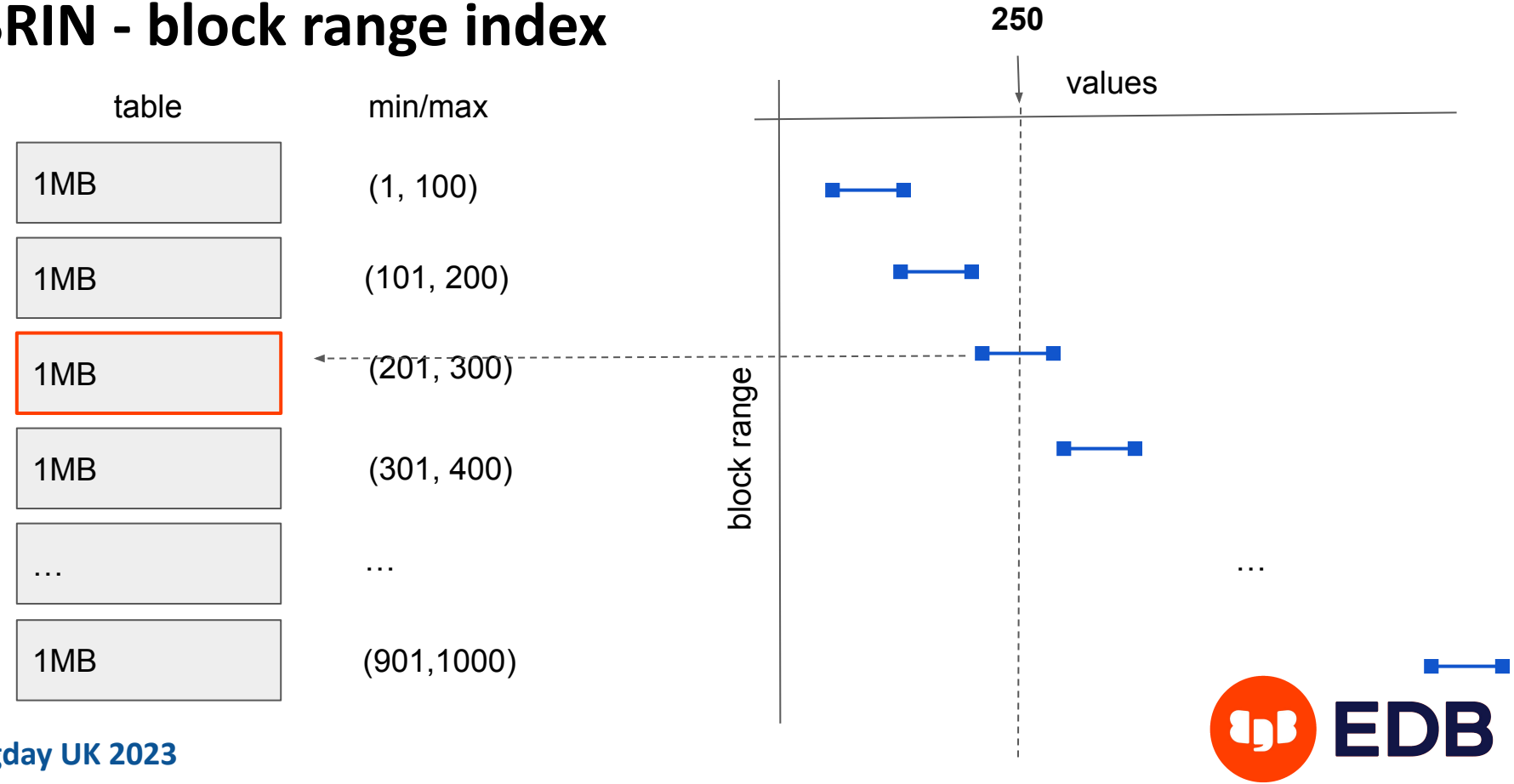
BRIN - block range index

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(1,25000000) s(i);

CREATE INDEX ON t USING BRIN (a);

ANALYZE t;
```

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

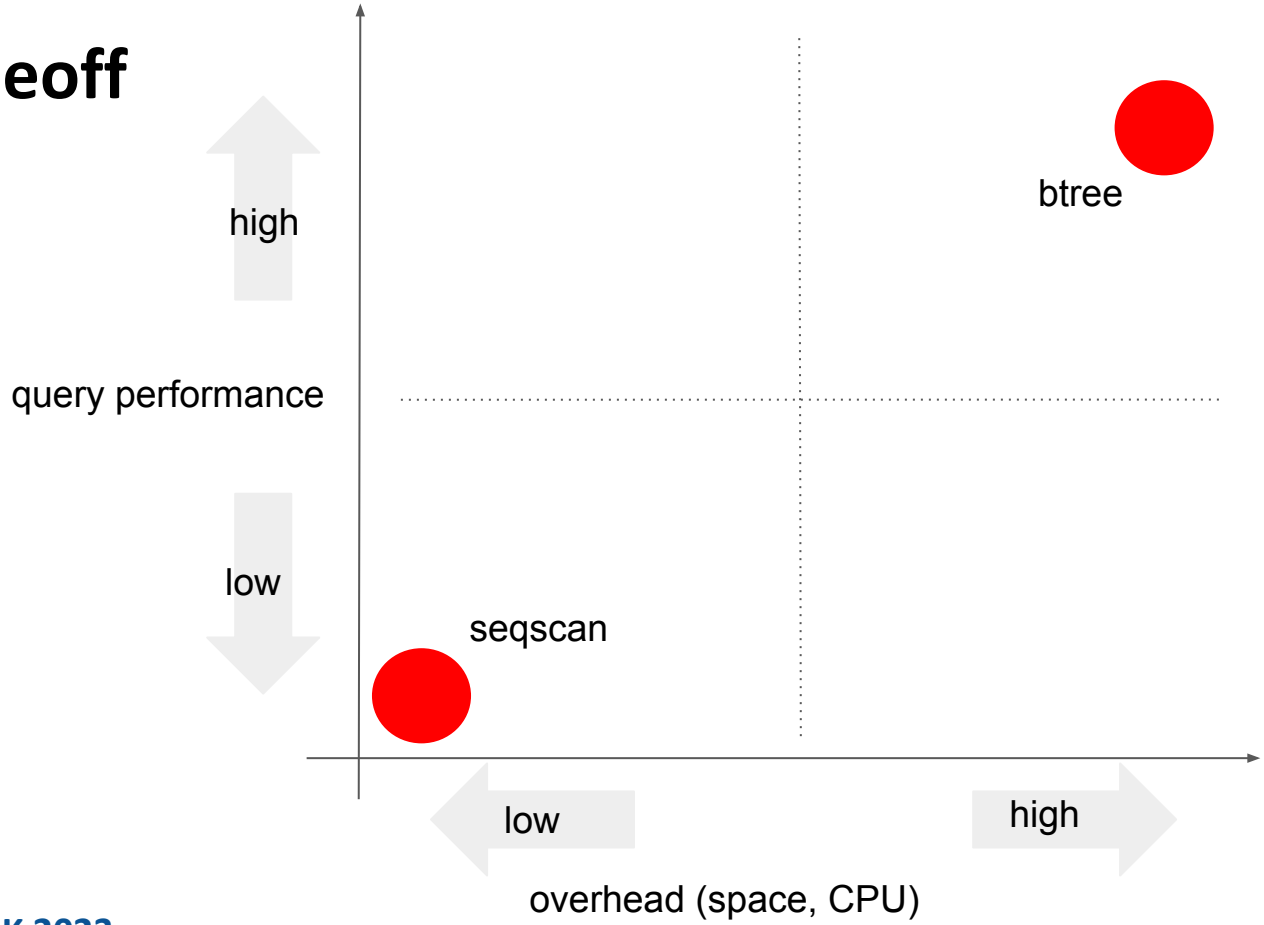
```
test=# \d+
                                    List of relations
 Schema | Name | Type  | Owner | Persistence | Access method |  Size
--------+------+-------+-------+-------------+---------------+---------
 public | t    | table | user  | permanent   | heap          | 8880 MB
(1 row)                                                       ~1.14M 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           | 336 kB
 public | t_a_idx1 | index | user  | t     | permanent   | btree         |174 MB
(2 rows)
```


EDB

Example

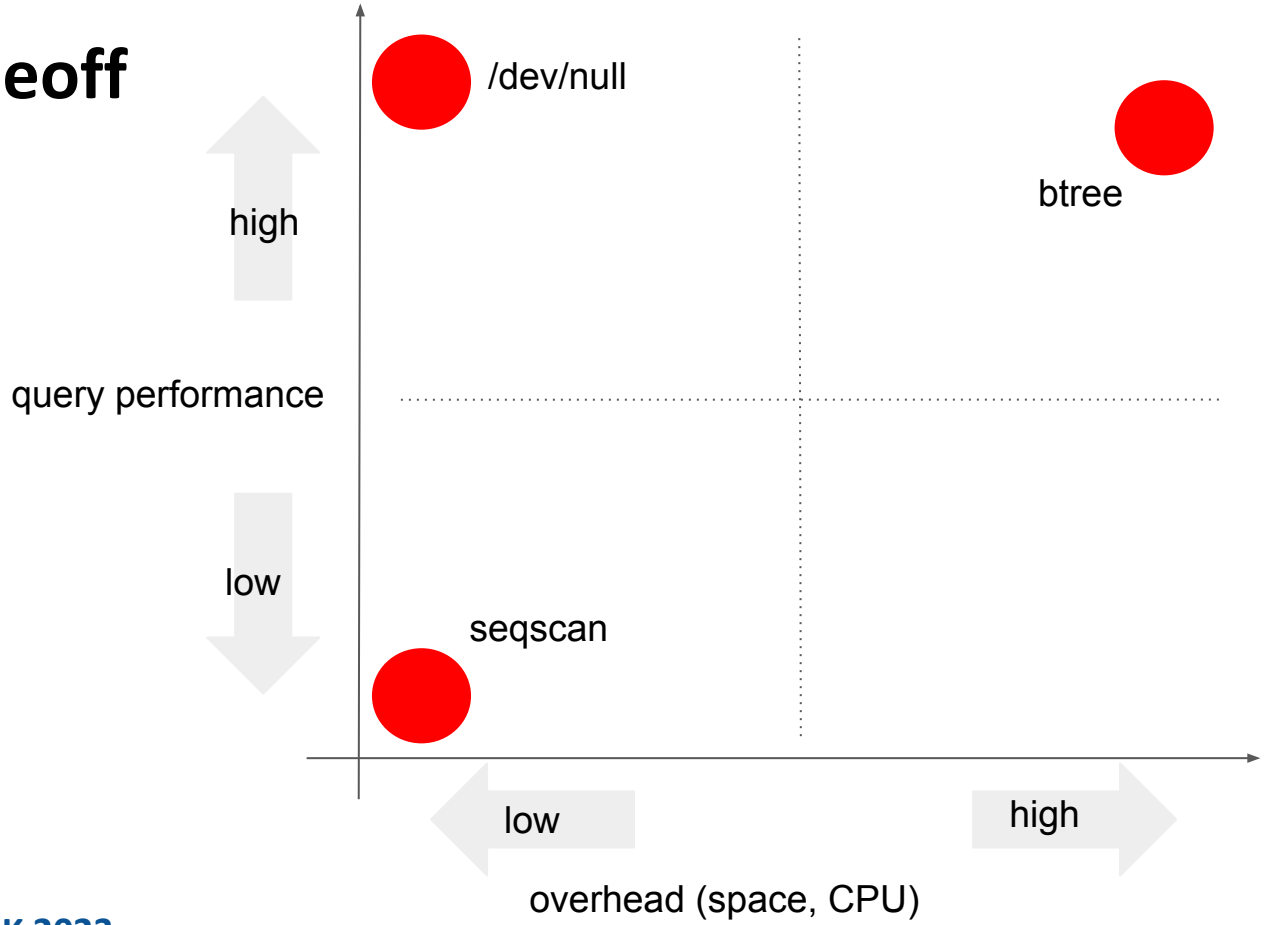
```
SET max_parallel_workers_per_gather = 0;

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

                                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------------
-
 Aggregate  (cost=950092.71..950092.72 rows=1 width=8)
            (actual time=8976.068..8976.075 rows=1 loops=1)
   ->  Bitmap Heap Scan on t  (cost=141.47..950092.09 rows=251 width=0)
                              (actual time=19.483..8975.470 rows=250 loops=1)
      Recheck Cond: (a = 4000)
      Rows Removed by Index Recheck: 1407302
      Heap Blocks: lossy= 63980    ~5% of 1.14M pages
      ->  Bitmap Index Scan on t_a_idx  (cost=0.00..141.41 rows=299678 width=0)
                                        (actual time=8.511..8.512 rows=639800 loops=1)
                  Index Cond: (a = 4000)
 Planning Time: 6.195 ms
 Execution Time: 1136.553 ms          seqscan: 17050.101 ms                  btree: 2.789 ms
(9 rows)
```

Tradeoff



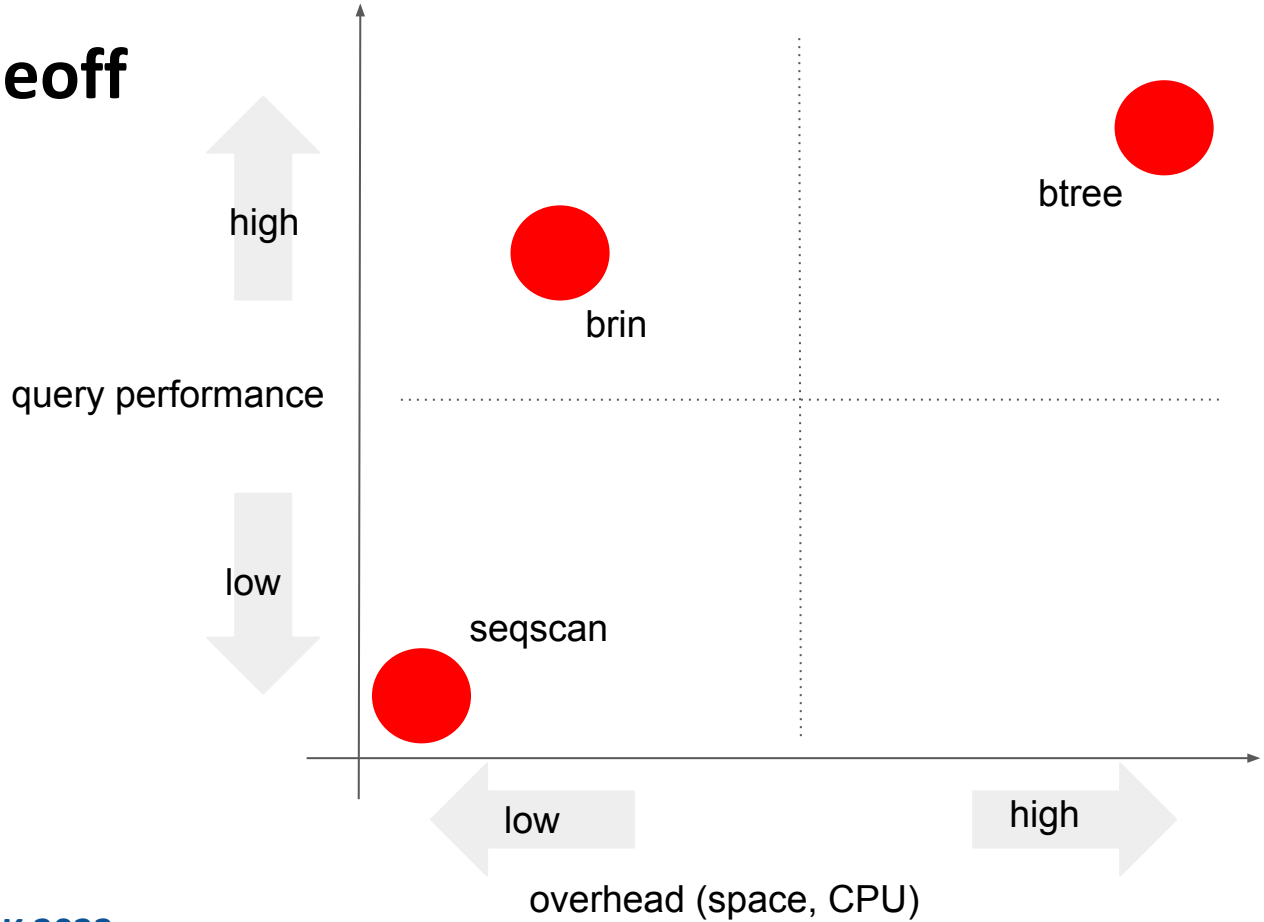
EDB

Tradeoff



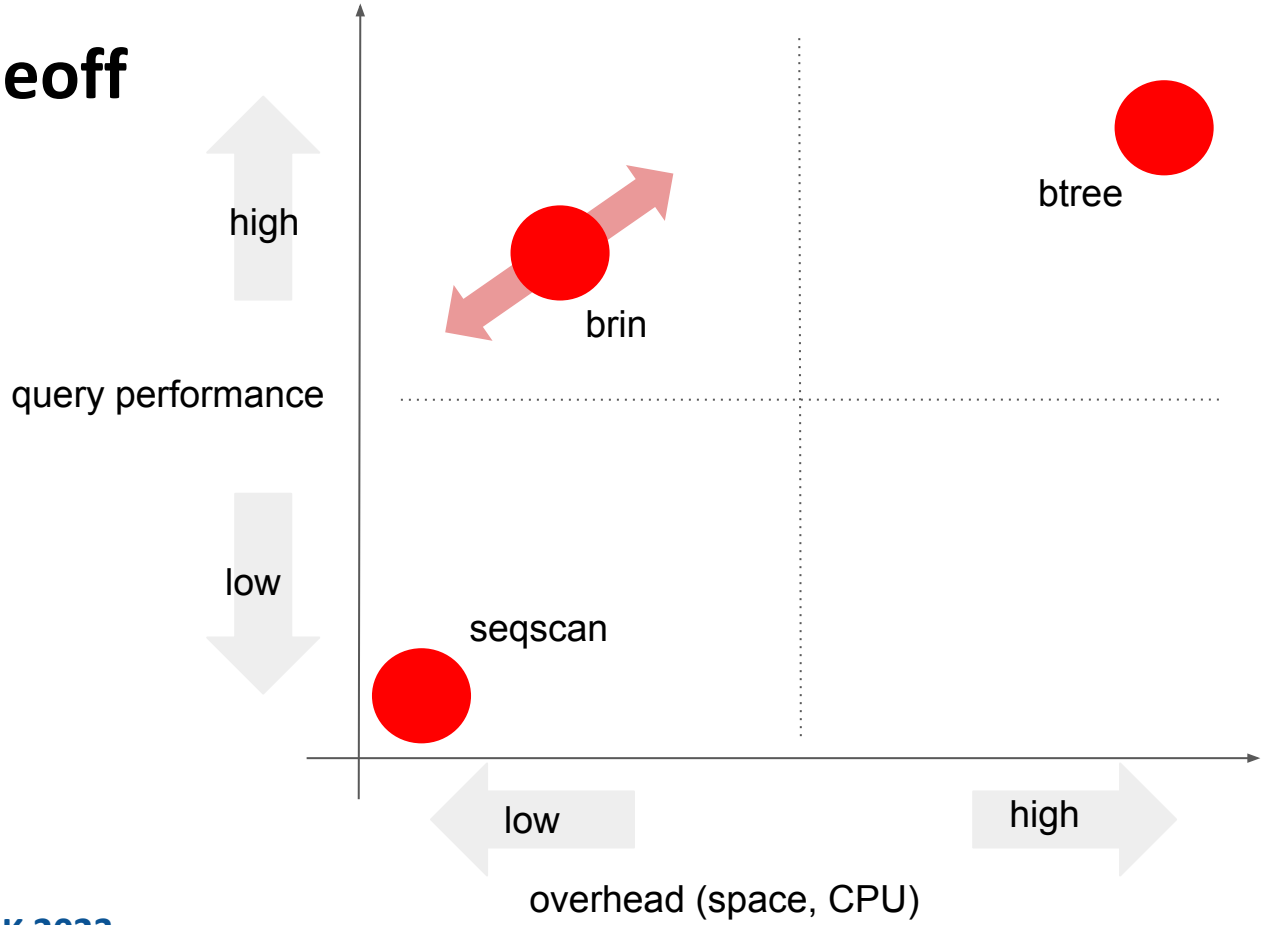
EDB

Tradeoff



EDB

Tradeoff



EDB

Problems (minmax, inclusion, ...)

- efficient "elimination" of ranges requires correlation
- 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=1465702.76..1465702.77 rows=1 width=8)
            (actual time=15992.319..15992.326 rows=1 loops=1)
   ->  Bitmap Heap Scan on t  (cost=150.39..1465702.10 rows=263 width=0)
                              (actual time=65.586..15991.439 rows=246 loops=1)
      Recheck Cond: (a = 4000)
      Rows Removed by Index Recheck: 24999754
      Heap Blocks: lossy=1136364      ~100% of pages
      ->  Bitmap Index Scan on t_a_idx      (cost=0.00..150.33 rows=1084812 width=0)
                                            (actual time=60.003..60.004 rows=11363640 loops=1)
                Index Cond: (a = 4000)
 Planning Time: 7.839 ms
 Execution Time: 15992.499 ms    <- seqscan: ~17000 ms
```


EDB

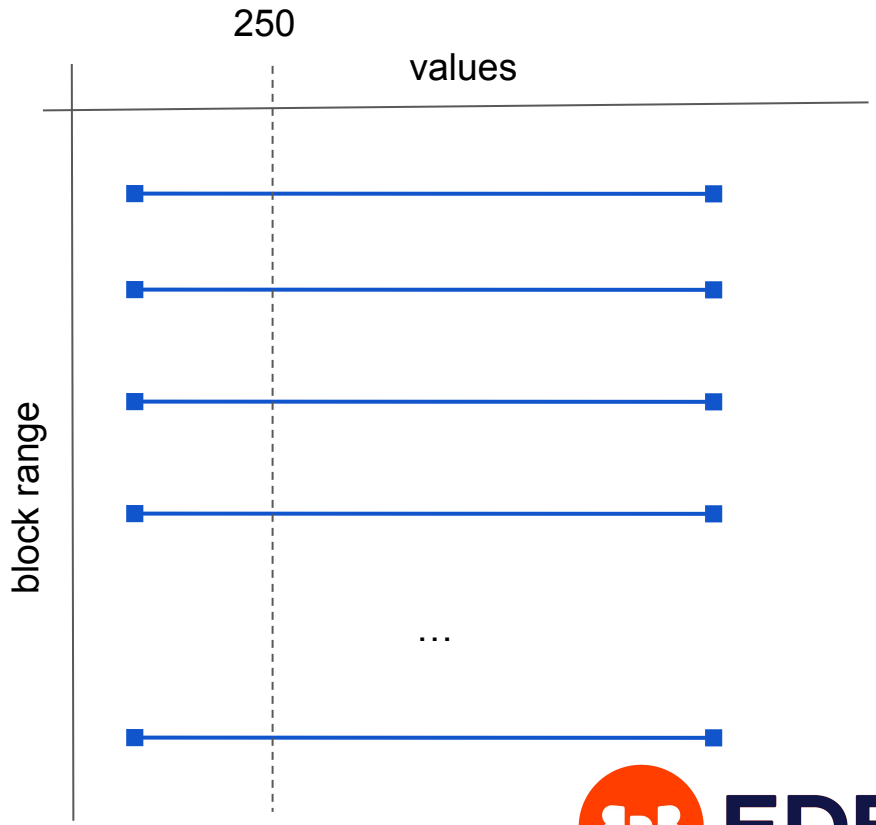
BRIN - example

```
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}
...
```


EDB

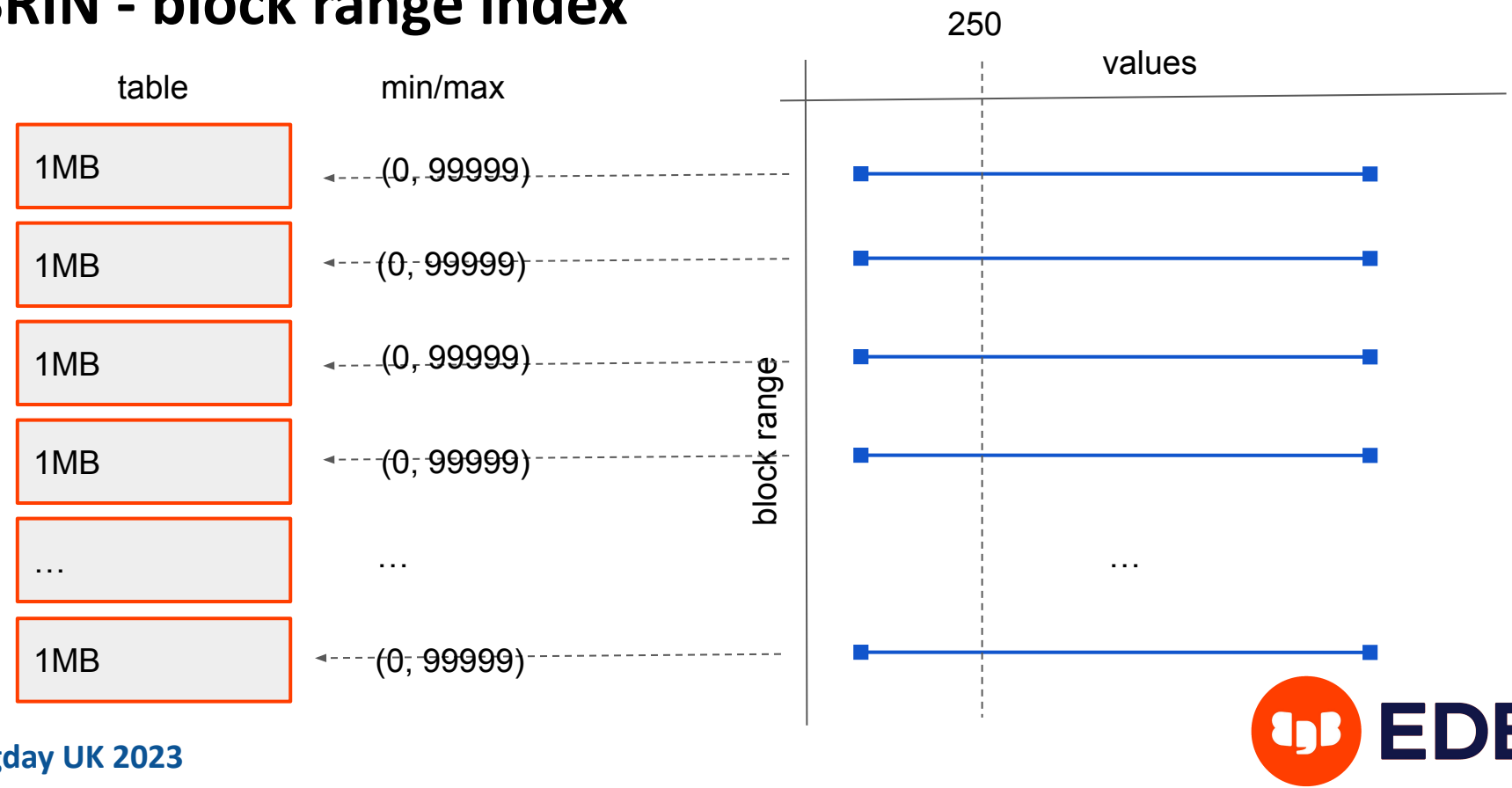
BRIN - block range index

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



EDB

BRIN - block range index

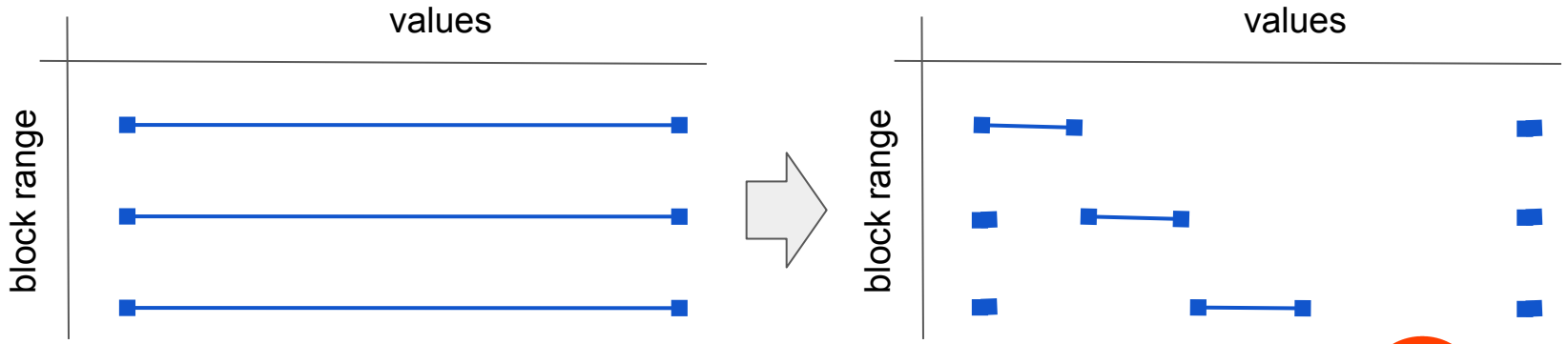


EDB

PG 14

minmax-multi opclass

- 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=889411.03..889411.04 rows=1 width=8)
            (actual time=662.853..662.861 rows=1 loops=1)
   ->  Bitmap Heap Scan on t  (cost=1245.04..889410.39 rows=257 width=0)
                              (actual time=22.494..662.388 rows=246 loops=1)
      Recheck Cond: (a = 4000)
      Rows Removed by Index Recheck: 703754
      Heap Blocks: lossy=32000
      ->  Bitmap Index Scan on t_a_idx  (cost=0.00..1244.97 rows=261742 width=0)
                                        (actual time=19.870..19.872 rows=320000 loops=1)
                Index Cond: (a = 4000)
 Planning Time: 6.810 ms
 Execution Time:  664.333 ms
(9 rows)
```


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

PG 16 (*)

Fix NULL-handling

- actually a bug
- occasionally managed to "forget" a range contains NULL
  - incorrect results for "IS NULL" queries
- fixed (and backpatched)
- now clear distinction between "empty" and "NULL-only" ranges
  - matters for tables with bulk DELETE

BRIN now allows HOT

- HOT - optimization allowing not updating indexes
- two conditions
  - new tuple version fits on the same page (fillfactor)
  - does not update any indexed columns
- used to be: updates any indexed column => has to update all indexes
- new: updates only BRIN-indexed columns => updates only BRIN indexes
- no indexes -> only BRIN indexes => all indexes
- much cheaper (BTREE updates mean a lot of random I/O)

PG 17 (?)
(no promises)

Parallel CREATE INDEX

- we only support that for BTREE
- actually fairly simple to do for BRIN
- naturally parallelizable
- you don't create indexes often
- but it usually happens on large data

SK_SEARCHARRAY

```
CREATE INDEX ON t USING BRIN (a) with (pages_per_range=1);

EXPLAIN ANALYZE SELECT * FROM t WHERE a IN (1000000);

                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Bitmap Heap Scan on t  (cost=16468.00..455522.00 rows=251 width=8)
                        (actual time=217.306..217.310 rows=0 loops=1)
   Recheck Cond: (a = 1000000)
   ->  Bitmap Index Scan on t_a_idx  (cost=0.00..16467.94 rows=82386 width=0)
                                     (actual time=217.300..217.301 rows=0 loops=1)
         Index Cond: (a = 1000000)
 Planning Time: 0.181 ms
 Execution Time: 217.337 ms
(6 rows)

... a IN (1000000, 1000001) => 445.038 ms
... a IN (1000000, 1000001, 1000002) => 660.994 ms    :-(
```


EDB

SK_SEARCHARRAY / patched

```
CREATE INDEX ON t USING BRIN (a) with (pages_per_range=1);

EXPLAIN ANALYZE SELECT * FROM t WHERE a IN (1000000);

                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Bitmap Heap Scan on t  (cost=16468.00..455522.00 rows=251 width=8)
                        (actual time=217.306..217.310 rows=0 loops=1)
   Recheck Cond: (a = 1000000)
   ->  Bitmap Index Scan on t_a_idx  (cost=0.00..16467.94 rows=82386 width=0)
                                     (actual time=217.300..217.301 rows=0 loops=1)
         Index Cond: (a = 1000000)
 Planning Time: 0.181 ms
 Execution Time: 217.337 ms
(6 rows)

... a IN (1000000, 1000001) => 445.038 ms
... a IN (1000000, 1000001, 1000002) => 660.994 ms    :-(
```


EDB

SK_SEARCHARRAY

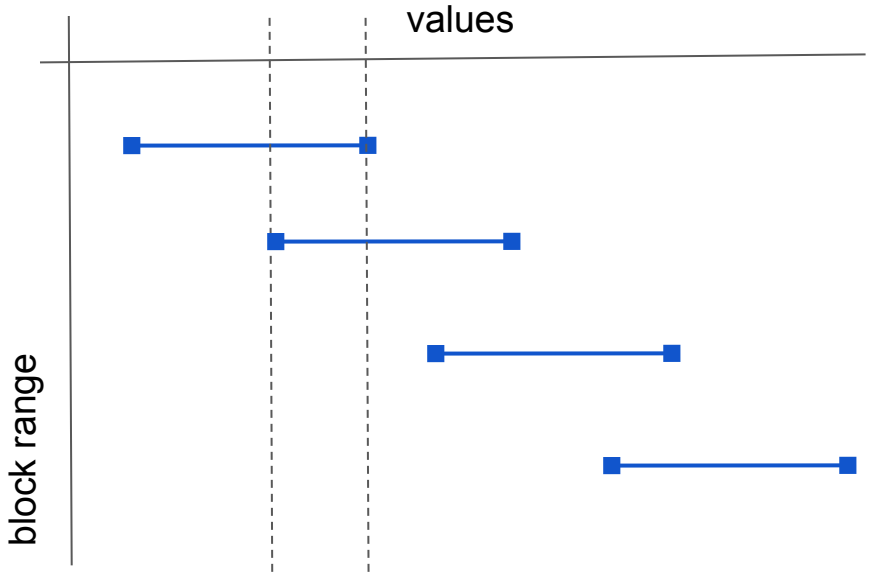
```
                                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on t  (cost=16468.67..647508.70 rows=1009 width=8)
                        (actual time=334.169..334.175 rows=0 loops=1)
   Recheck Cond: (a = ANY ('{1000000,1000001,1000002,1000003}'::bigint[]))
   ->  Bitmap Index Scan on t_a_idx  (cost=0.00..16468.42 rows=125024 width=0)
                                     (actual time=334.160..334.162 rows=0 loops=1)
         Index Cond: (a = ANY ('{1000000,1000001,1000002,1000003}'::bigint[]))
 Planning Time: 0.187 ms
 Execution Time: 334.217 ms     (~1000ms now)
(6 rows)
```

- real (generated) queries often have hundreds of values in the list
- matters even with the default pages_per_range value

EDB

Sorting using BRIN

- BTREE is sorted, can be used for sorting easily
- BRIN seemingly incompatible, but ...
  - minmax can still help
  - incremental sorting
- greate for ORDER BY + LIMIT queries
- without LIMIT may eliminate spilling



EDB

Sorting using BRIN

```
SELECT * FROM t ORDER BY a LIMIT 1000;

                                        QUERY PLAN
-------------------------------------------------------------------------------------
 Limit  (cost=2757087.55..2757090.05 rows=1000 width=8)
        (actual time=57639.680..57642.581 rows=1000 loops=1)
   ->  Sort  (cost=2757087.55..2819587.57 rows=25000008 width=8)
             (actual time=57639.676..57640.657 rows=1000 loops=1)
         Sort Key: a
         Sort Method: top-N heapsort  Memory: 49kB
         ->  Seq Scan on t  (cost=0.00..1386364.08 rows=25000008 width=8)
                            (actual time=0.705..32530.479 rows=25000000 loops=1)
 Planning Time: 5.128 ms
 Execution Time: 57644.530 ms
(7 rows)
```


EDB

Sorting using BRIN

```
SET enable_brinsort = on;

SELECT * FROM t ORDER BY a LIMIT 1000;

                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Limit  (cost=6.00..2195.74 rows=1000 width=8)
        (actual time=959.081..1075.650 rows=1000 loops=1)
   ->  BRIN Sort using t_a_idx on t  (cost=6.00..54743398.10 rows=25000000 width=8)
                                     (actual time=959.077..1073.741 rows=1000 loops=1)
         Sort Key: a
         Ranges: 8878  Build time: 12  Method: quicksort  Space: 870 kB (Memory)
         ...
         Tuples Sorted: 8497  Per-sort: 4248  Direct: 596  Spilled: 711404  ...
         Sorts (in-memory)  Count: 2  Space Total: 458 kB  Maximum: 418 kB  ...
 Planning Time: 6.142 ms
 Execution Time: 1083.079 ms              (btree: 5ms)
(12 rows)
```


EDB

patch ideas

Patch ideas

- 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?

Patch ideas

- 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?

Q & A

EDB

Enter for a chance to win a

LEGO® Millennium Falcon™