CYBERTEC
POSTGRESQL SERVICES & SUPPORT

Counting things at the speed of light with roaring bitmaps

Ants Aasma

pgconf.eu 2023

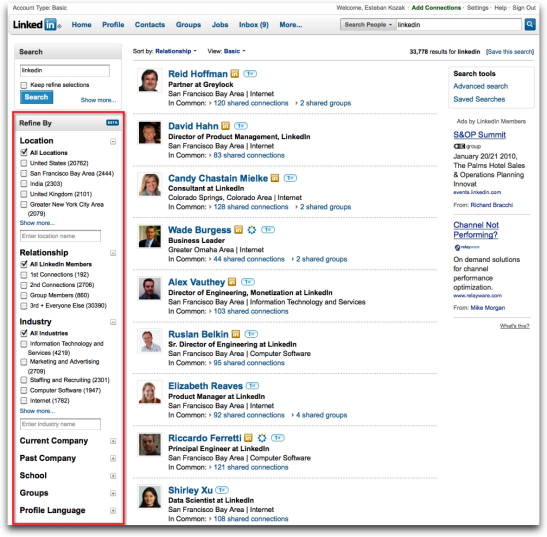
Hello

About me

- Ants Aasma
- Senior Database Consultant
- 12 years of helping people make PostgreSQL run fast

What is faceting

Counting on steroids



The trouble with faceting

- Core task is simple:

```
SELECT attribute, COUNT(*) FROM sometable WHERE .. GROUP BY attribute
```

- Attribute can be anything:
  - Category
  - Status
  - Date
  - Tags
  - ...
- There are too many filter combinations to precompute the counts.
- Some filters are not very selective, need to tally up a large fraction of all objects.
- Is used as a navigational aid, so needs to be interactive fast.

Story time

- We need to provide faceting on upwards of 100M documents.
- Want to have accurate counts, at worst slightly stale ones.
  - No data leaking!
- Response time: < 2s
- Want to do this in PostgreSQL, because it is cool.
  (also because maintaining an external ElasticSearch cluster is a pain)

Example schema

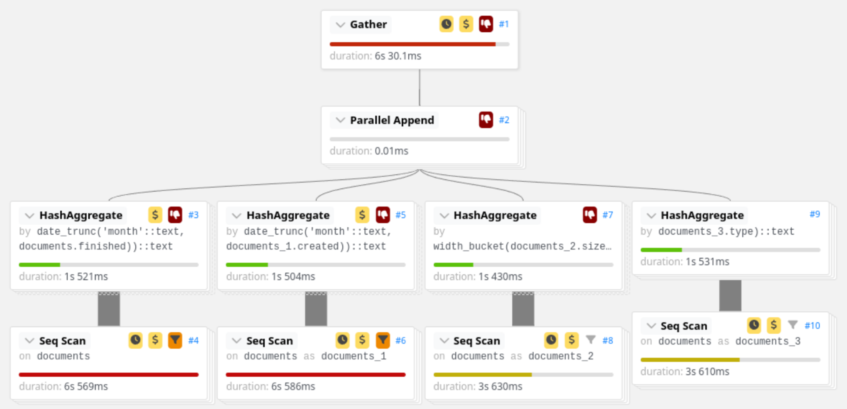
```
faceting=# \d documents
                    Table "public.documents"
   Column    |           Type           | Collation | Nullable | Default
-------------+--------------------------+-----------+----------+---------
 id          | integer                  |           | not null |
 created     | timestamp with time zone |           | not null |
 finished    | timestamp with time zone |           |          |
 category_id | bigint                   |           | not null |
 tags        | text[]                   |           |          |
 type        | mimetype                 |           |          |
 size        | bigint                   |           |          |
 title       | text                     |           |          |
Indexes:
    "documents_pkey" PRIMARY KEY, btree (id)

faceting=# SELECT COUNT(*), COUNT(*) FILTER (WHERE category_id = 24) in_cat24 FROM documents;
   count   | in_cat24
-----------+----------
 100000000 | 60819016
```

Naive implementation

```
(SELECT 'created' AS facet_name, date_trunc('month', created)::text AS facet_value,
        COUNT(*) AS cardinality
 FROM documents WHERE category_id = 24 GROUP BY 1, 2)
    UNION ALL
(SELECT 'finished', date_trunc('month', finished)::text, COUNT(*)
 FROM documents WHERE category_id = 24 GROUP BY 1, 2)
    UNION ALL
(SELECT 'type', type::text, COUNT(*)
 FROM documents WHERE category_id = 24 GROUP BY 1, 2)
    UNION ALL
(SELECT 'size', width_bucket(size, array[0,1000,5000,10000,50000,100000,500000])::text,
        COUNT(*)
 FROM documents WHERE category_id = 24 GROUP BY 1, 2);
```

Naive result

- 32s

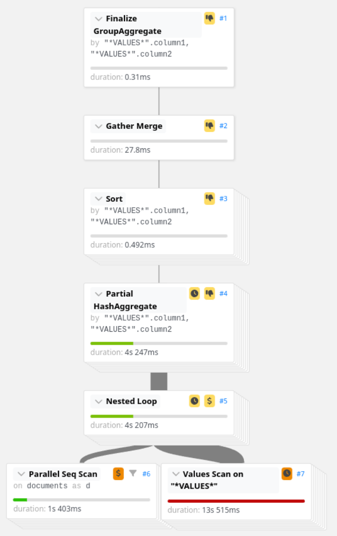


Getting rid of multiple scans

```
SELECT facet_name, facet_value, COUNT(*) cardinality
FROM documents d, LATERAL (VALUES
    ('created', date_trunc('month', created)::text),
    ('finished', date_trunc('month', finished)::text),
    ('type', type::text),
    ('size', width_bucket(size, array[0,1000,5000,10000,50000,100000,500000])::text)
) t(facet_name, facet_value)
WHERE category_id = 24 GROUP BY 1, 2;
```

Slightly better. . .

- 23.4s



More parallelism?

- This was with 8 parallel workers.
- With 24 (number of logical threads) I got 16.8s.

```
ALTER TABLE documents SET (parallel_workers = 23)
```

- Only looking at 4 facets here.
- This does not scale well...

Back of the envelope calculations

- For each row:
  - For each facet:
    - Find the counter for (facet, value) and add 1 to it.
- For 100M rows and 10 facets need to do the inner loop 1 billion times.
- Even averaging a single memory access per row gets us 100ns*1B = 100s of CPU time.
  - Each memory access is 30 lightmeters

Back of the envelope calculations

- For each row:
  - For each facet:
    - Find the counter for (facet, value) and add 1 to it.
- For 100M rows and 10 facets need to do the inner loop 1 billion times.
- Even averaging a single memory access per row gets us 100ns*1B = 100s of CPU time.
  - Each memory access is 30 lightmeters
- Randomized access is slooooow. . .

What CPUs do fast

- CPUs are really good at bit arithmetic and vectorized execution.

What CPUs do fast

- CPUs are really good at bit arithmetic and vectorized execution.
- Useful instructions:
  - `VANDPD zmm, zmm, m512`
    - Read 512 bits from memory and do a logical AND with a value in register
    - Intel Xeon 4th gen and AMD EPYC 4th gen can do 2 per cycle
  - `VPOPCNTQ zmm, zmm`
    - Count number of set bits in 8 64bit words.
    - 1 per cycle
  - `VPADDQ zmm, zmm, zmm`
    - Add together elements of two 8x64bit vectors.
    - 2 per cycle.

What CPUs do fast

- CPUs are really good at bit arithmetic and vectorized execution.
- Useful instructions:
  - `VANDPD zmm, zmm, m512`
    - Read 512 bits from memory and do a logical AND with a value in register
    - Intel Xeon 4th gen and AMD EPYC 4th gen can do 2 per cycle
  - `VPOPCNTQ zmm, zmm`
    - Count number of set bits in 8 64bit words.
    - 1 per cycle
  - `VPADDQ zmm, zmm, zmm`
    - Add together elements of two 8x64bit vectors.
    - 2 per cycle.
- Need 1 - 1.5 cycles per iteration.
- ~3 GHz / 1.5 cycles * 512 bits ~= 1’000’000’000’000 bit intersections/s/core
  - That’s 1T with a T
  - 0.3 lightmm per bit

Converting our problem to bitmaps

- Lets assume we have an integer id field
- Lets precalculate a bitmap for each attribute-value combination
- For every document where `attribute=value` set bit at position `id` to 1
- Store bitmaps in a `(attr, value, bitmap)` table.

Converting our problem to bitmaps

- Lets assume we have an integer id field
- Lets precalculate a bitmap for each attribute-value combination
- For every document where `attribute=value` set bit at position `id` to 1
- Store bitmaps in a `(attr, value, bitmap)` table.
- Not actually a new idea - usually called an inverted index.
  - (like GIN)

Calculating facet counts

1. Build bitmap corresponding to where clause.
   - If can be expressed in terms of facets can combine existing bitmaps.
2. For each facet and value, calculate:

```
POPCNT(AND(lookup_bitmap, facet_value_bitmap))
```

Some numbers

- Assuming 10 facets, avg 1’000 values each, 10k * 100M = 1T bits
- In theory can calculate it in 1 CPU second

Some numbers

- Assuming 10 facets, avg 1’000 values each, 10k * 100M = 1T bits
- In theory can calculate it in 1 CPU second

- 1T bits = 125GB (1.25KB per document)
  - Memory usage
  - Memory bandwidth (typical ~10GB/s per physical core)

Can we do better

- 99.9% of those 1T bits are 0
- Some things are very popular, some things less so.
- Can we use some hybrid storage approach?

Roaring Bitmaps

What are Roaring Bitmaps

- Fast implementation of compressed integer sets.
  - Daniel Lemire, et al. 2017
- Adaptive datastructure.
- SIMD accelerated.

How do Roaring Bitmaps work

- 2 level tree.
- 32bit integers are split into low and high words.
- First level is sorted list of high words that have a container.
  - For each one store the high word, pointer and container type.

Roaring bitmap container types

- Array
  - Sorted list of 16bit low words
  - Up to 4096 entries.
- Bitmap
  - 2^{16} entry bitmap (8KB)
- (optional) Run length encoded
  - 4 byte pairs of `(starting_value, run_length)`

Roaring bitmap operations

- Pattern: pair up containers of two bitmaps, run operation on pair

Roaring bitmap operations

- Pattern: pair up containers of two bitmaps, run operation on pair
- Example: a *intersect* b
  - null & any $\Rightarrow$ null
  - array & array $\Rightarrow$ SIMD accelerated intersection, with special cases for skew
  - bitmap & bitmap $\Rightarrow$ SIMD intersection, convert to array if small
  - array & bitmap $\Rightarrow$ branchless loop to filter array with lookups to bitmap
  - array & run $\Rightarrow$ merge join
  - ...

Other users

Used all over the place:

- ClickHouse
- Apache Lucene (Elasticsearch, Solr)
- Apache Hive
- Pinot

pg_roaringbitmap

What is pg_roaringbitmap

- PostgreSQL extension that wraps C Roaring Bitmap library.
- Introduces a `roaringbitmap` datatype and associated operations.
- Available from github.com/ChenHuajun/pg_roaringbitmap
  - Not available on AWS RDS, Google Cloud SQL or Azure (yet)

Using pg_roaringbitmap

Regular PostgreSQL datatype:

```
CREATE EXTENSION roaringbitmap;

CREATE TABLE document_facets (
    facet_id int4,
    facet_value text,
    postinglist roaringbitmap NOT NULL,
    PRIMARY KEY (facet_id, facet_value)
);
```

Building bitmaps

- Convert an array to a roaring bitmap

```
INSERT INTO document_facets
VALUES (1, 'helloworld', rb_build(array[1,2,3]));
```

Building bitmaps

- Aggregate a set of integers to a roaring bitmap

```
SELECT rb_build_agg(i) FROM generate_series(1, 100) i;

INSERT INTO document_facets
SELECT 1, category_id::text, rb_build_agg(id)
    FROM documents GROUP BY 1, 2;
```

Useful operations

- Combine two bitmaps:
  - `roaringbitmap & roaringbitmap` ⇒ `roaringbitmap` - AND
  - `roaringbitmap | roaringbitmap` ⇒ `roaringbitmap` - OR
  - `roaringbitmap # roaringbitmap` ⇒ `roaringbitmap` - XOR
  - `roaringbitmap - roaringbitmap` ⇒ `roaringbitmap` - AND NOT

Element wise operations

- Add element:

  `roaringbitmap | integer` ⇒ `roaringbitmap`

- Remove element:

  `roaringbitmap - integer` ⇒ `roaringbitmap`

- Check if member

  `roaringbitmap @> integer` ⇒ `boolean`

- Get members

  `rb_iterate(roaringbitmap)` ⇒ `SETOF integer`

  `rb_to_array(roaringbitmap)` ⇒ `integer[]`

Counting things

- Number of elements in set:

  `rb_cardinality(roaringbitmap)` ⇒ `bigint`

  + combined op & count functions:

  ```
  rb_and_cardinality(roaringbitmap, roaringbitmap) ⇒ bigint
  rb_or_cardinality(roaringbitmap, roaringbitmap) ⇒ bigint
  rb_xor_cardinality(roaringbitmap, roaringbitmap) ⇒ bigint
  rb_andnot_cardinality(roaringbitmap, roaringbitmap) ⇒ bigint
  ```

- Empty:

  `rb_is_empty(roaringbitmap)` ⇒ `boolean`

Aggregating things

- Aggregate functions to aggregate across rows:

  rb_or_agg(roaringbitmap) ⇒ roaringbitmap
  rb_and_agg(roaringbitmap) ⇒ roaringbitmap
  rb_xor_agg(roaringbitmap) ⇒ roaringbitmap

- When we only care about the count

  rb_or_cardinality_agg(roaringbitmap) ⇒ roaringbitmap
  rb_and_cardinality_agg(roaringbitmap) ⇒ roaringbitmap
  rb_xor_cardinality_agg(roaringbitmap) ⇒ roaringbitmap

Limitations

- Currently only 32 bit integers are supported

Building the faceting

Storing facets as roaring bitmaps

```
CREATE TABLE documents_facets AS
SELECT facet_name COLLATE "C", facet_value COLLATE "C", rb_build_agg(id) postinglist
FROM documents d, LATERAL (VALUES
    ('category_id', category_id::text),
    ('created', date_trunc('month', created)::text),
    ('finished', date_trunc('month', finished)::text),
    ('type', type::text),
    ('size', width_bucket(size, array[0,1000,5000,10000,50000,100000,500000])::text)
) t(facet_name, facet_value)
GROUP BY 1, 2;
ALTER TABLE documents_facets ADD PRIMARY KEY (facet_name, facet_value);
```

Execution time: 34s

Resulting table

```
faceting=# SELECT pg_size_pretty(pg_total_relation_size('documents_facets'));
 pg_size_pretty
----------------
 214 MB
(1 row)

faceting=# SELECT facet_name, COUNT(*), MIN(LENGTH(postinglist::bytea)),
faceting-#     MAX(LENGTH(postinglist::bytea)), SUM(LENGTH(postinglist::bytea))
faceting-# FROM documents_facets GROUP BY 1;
 facet_name  | count |   min   |   max    |   sum
-------------+-------+---------+----------+----------
 type        |     8 | 3980664 | 12513208 | 84585018
 size        |     7 |  244942 | 12513208 | 54145664
 created     |   168 |      65 |      300 |    26735
 category_id |   100 |   31272 | 12513208 | 73180638
 finished    |   168 |    9211 |    33108 |  3739175
(5 rows)
```

Getting our facets

```sql
WITH lookup AS  (
   SELECT postinglist FROM documents_facets
   WHERE facet_name = 'category_id' AND facet_value = '24'
)
SELECT facet_name, facet_value,
       rb_and_cardinality(facet.postinglist, lookup.postinglist)
FROM lookup, documents_facets facet
WHERE facet.facet_name != 'category_id';
```

```
Execution Time: 1057.078 ms
```

Explain plan

```
Nested Loop  (cost=0.27..26.32 rows=351 width=35) (actual time=3.213..1056.931 rows=351 loops=1)
  Buffers: shared hit=575611
  ->  Index Scan using documents_facets_pkey on documents_facets  (cost=0.27..8.29 rows=1 width=72)
      (actual time=0.009..0.011 rows=1 loops=1)
        Index Cond: ((facet_name = 'category_id'::text) AND (facet_value = '24'::text))
        Buffers: shared hit=3
 -> Seq Scan on documents_facets facet  (cost=0.00..13.64 rows=351 width=99) (actual time=0.012..0.630 rows=3
        Filter: (facet_name <> 'category_id'::text)
        Rows Removed by Filter: 100
        Buffers: shared hit=8
Planning:
  Buffers: shared hit=2
Planning Time: 0.102 ms
Execution Time: 1057.078 ms
```

TOAST!!!!!!

- Large values are stored out of line in a secondary table.
  - `(chunk_id oid, chunk_seq int, chunk_data bytea)`
- Every time a toasted value is accessed this table is queried.
- PostgreSQL is not very smart about when to detoast.
- PostgreSQL offers very little control over when to detoast.

Some tricks to force the planner

```
WITH lookup AS  (
   SELECT postinglist << 0 postinglist FROM documents_facets
   WHERE facet_name = 'category_id' AND facet_value = '24'
   OFFSET 0
)
SELECT facet_name, facet_value,
       rb_and_cardinality(facet.postinglist, lookup.postinglist)
FROM lookup, documents_facets facet
WHERE facet.facet_name != 'category_id';
```

Execution Time: 80.100 ms

Other TOAST tricks

If possible store inline.

```
ALTER TABLE documents_facets SET (toast_tuple_target = 8160);
```

Use faster compression.

```
ALTER TABLE documents_facets ALTER postinglist SET COMPRESSION "lz4";
-- or even better
SET default_toast_compression = 'lz4';
```

Or no compression at all.

```
ALTER TABLE documents_facets ALTER postinglist SET STORAGE EXTERNAL;
```

Surprisingly no major impact.

Dealing with write amplification

- Keeping facets up to date on every modification creates insane amounts of write amplification
- For each insert need to update 12.5MB * N_facets of data.
- Updates are up to double that.

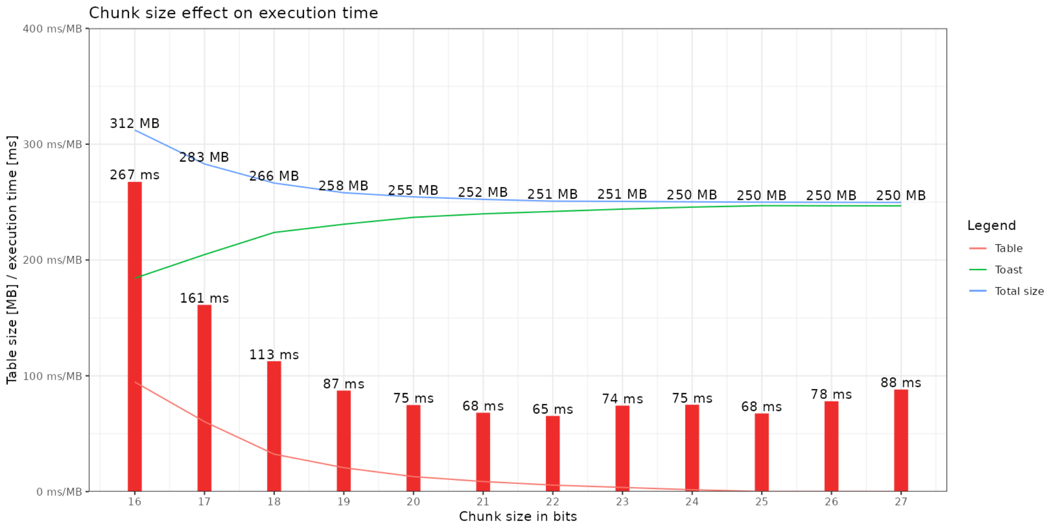
Chunking

- Lets partition the id range into smaller chunks.
- Chunks need to be big enough to hide overheads, but small enough to not be too costly to update.
- Example chunking method: `id >> 20 AS chunk_id`
  - 1M rows per chunk means up to 128KB bitmaps
- Query needs to join facets using chunk_id and tally up results at the end:

```
SELECT facet_name, facet_value,
       sum(rb_and_cardinality(facet.postinglist, lookup.postinglist))
FROM lookup JOIN documents_facets facet USING (chunk_id)
GROUP BY 1,2;
```

- ~10% performance improvement.

Chunking effect



Delta tables

- Further reduce overhead by storing updates in a delta table.

```
CREATE TABLE documents_facets_deltas (
    facet_id int4 not null,
    facet_value text collate "C" null,
    posting integer not null,
    delta int2,
    primary key (facet_id, facet_value, posting)
);
```

- On insert run:

```
INSERT INTO documents_facets_deltas VALUES (..., ..., id, +1)
ON CONFLICT (facet_id, facet_value, posting) DO UPDATE
    SET delta = EXCLUDED.delta + documents_facets_deltas.delta;
```

- Delete is same with -1

Delta tables

- Triggers can automatically maintain delta tables.
- Periodically aggregate together deltas and merge them in:

```
WITH tbd AS (DELETE FROM documents_facets_deltas RETURNING *),
deltas AS (SELECT facet_name, facet_value,
            rb_build_agg(posting) FILTER (WHERE delta > 0) AS added,
            rb_build_agg(posting) FILTER (WHERE delta < 0) AS removed
        FROM tbd GROUP BY 1, 2)
MERGE INTO documents_facets df
USING deltas d
ON df.facet_name = d.facet_name AND df.facet_value = d.facet_value
WHEN MATCHED THEN
    UPDATE SET postinglist = postinglist | added - removed
WHEN NOT MATCHED THEN
    INSERT VALUES (d.facet_name, d.facet_value, added);
```

Multi-valued facets

- Sometimes a row can have more than one value for an attribute.
  - Tags
  - Keywords
  - Joined attributes
- Easy to handle - just generate `(facet_name, facet_value)` pair for each value.

Packaging it all up

- This is all available as ready to use code. Work sponsored by XeniT.
- github.com/cybertec-postgresql/pgfaceting
- Usage:

```
CREATE EXTENSION pgfaceting;
SELECT faceting.add_faceting_to_table(
    'documents', key => 'id', keep_deltas => true,
    facets => array[
        faceting.datetrunc_facet('created', 'month'),
        faceting.datetrunc_facet('finished', 'month'),
        faceting.plain_facet('category_id'),
        faceting.plain_facet('type'),
        faceting.bucket_facet('size', buckets =>
          array[0,1000,5000,10000,50000,100000,500000])
    ]
);
```

Generating search queries

```
SELECT * FROM faceting.count_results('documents',
    filters => array[row('category_id', '24'),
                     row('type', 'image/jpeg')
                ]::faceting.facet_filter[]);
```

Add this as a cron job to merge in deltas periodically.

```
CALL faceting.run_maintenance();
```

Future work

- Maintenance tools (add facet/remove facet)
- Better interface for generating search queries
- Automatically join in deltas when searching
- Option to only keep top facets

What doesn’t work well

- Sparse values will be worse than just storing an integer array.

Other uses for Roaring Bitmaps

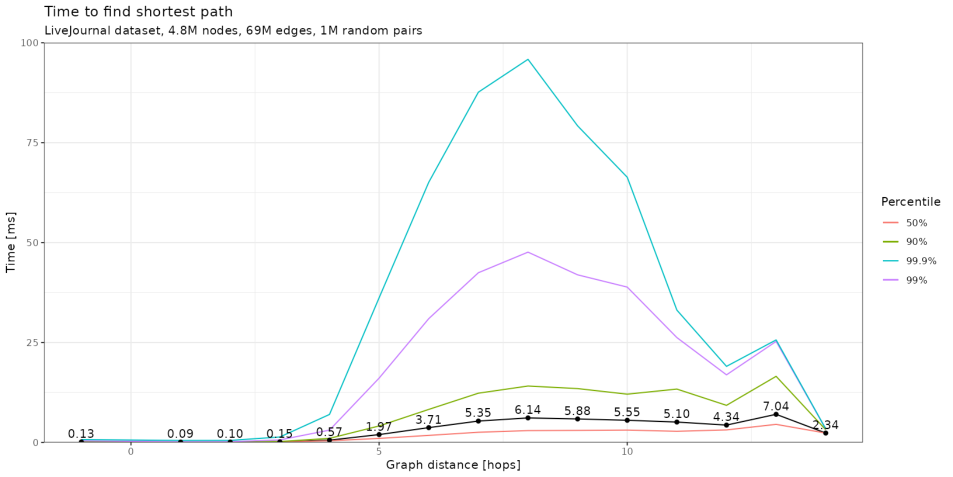
Graphs

- Fast graph algorithms
  - Number nodes with integer id’s
  - Store for each node incoming and/or outgoing edges as roaring bitmap

Finding shortest path

```
CREATE OR REPLACE FUNCTION shortest_path(src int, dest int) RETURNS int AS $$
DECLARE
  o_depth int := 0; o_new roaringbitmap := rb_build(array[src]); o_tc roaringbitmap := o_new;
  i_depth int := 0; i_new roaringbitmap := rb_build(array[dest]); i_tc roaringbitmap := i_new;
BEGIN
  WHILE NOT i_new && o_new LOOP
    IF rb_cardinality(i_new) < rb_cardinality(o_new) THEN
      SELECT rb_or_agg(edges) - i_tc INTO i_new FROM lj_i WHERE node = ANY (rb_to_array(i_new));
      IF rb_is_empty(i_new) THEN RETURN null; END IF;
      i_depth := i_depth + 1; i_tc := i_tc | i_new;
    ELSE
      SELECT rb_or_agg(edges) - o_tc INTO o_new FROM lj_o WHERE node = ANY (rb_to_array(o_new));
      IF rb_is_empty(o_new) THEN RETURN null; END IF;
      o_depth := o_depth + 1; o_tc := o_tc | o_new;
    END IF;
  END LOOP;
  RETURN i_depth + o_depth;
END;$$ LANGUAGE plpgsql STABLE STRICT;
```

Graph benchmark



That's all folks!

Questions

Questions

You can leave feedback at

2023.pgconf.eu/f

Bonus

Why/why not use this in GIN

- Conceptually very similar to what GIN postinglist is.
- GIN needs to run visibility checks
- CTID is `[      32bit block      ][ 16bit lp ]`
  - The linepointer values are < 300
- Page headers make it so a bitmap container doesn’t fit in a page.