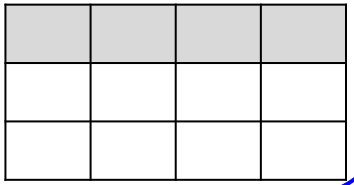
Multi-Entry Generalized Search Trees

PGConf.dev 2024

Maxime Schoemans
Université Libre de Bruxelles

	PostgreSQL	PostGIS	MobilityDB
SQL Optimization	Scalar statistics & selectivity estimation	Grid-based statistics	Spatial grid + period bound histograms
Operations	Comparison, transformation, casting, ...etc	topological, CRS, properties, overlay, ...etc.	trajectory, temporal properties, lifted predicates, aggregations
Indexes	B-tree, hash, GiST, SP-GiST, GIN, BRIN	GiST, SP-GiST, BRIN	GiST, SP-GiST
Type System	numeric, character, date/time, bool, xml, json	Geometry, geography	tgeompoint, tgeogpoint, tint, tfloat, ttext, tbool



Indexing PostgreSQL data types

- B-Tree
  - Numeric types (integer, real)
  - Character types (char, varchar, text)
  - Date / Time types (date, time, timestamp)
- GiST / SP-GiST
  - Geometric types (point, line, box, polygon, path, circle)
  - Range and Multirange types
- GIN
  - Text search types (tsvector)
  - JSON types (jsonb)
  - Array types
- Hash / BRIN

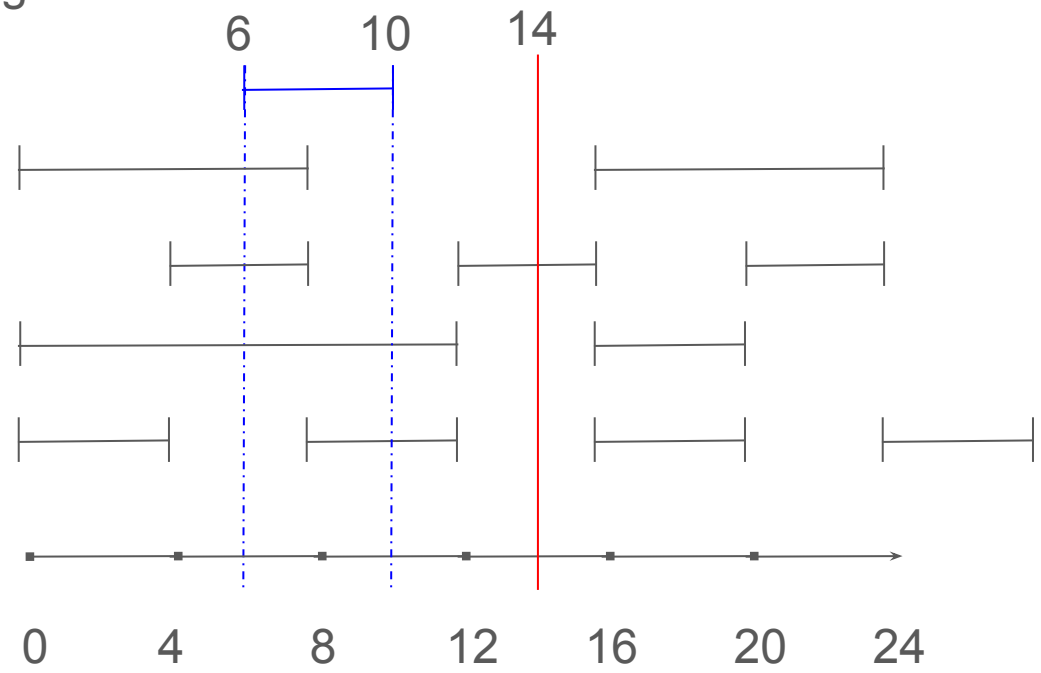
In some cases indexing these data types does not improve query performance enough

Custom array operators cannot always be answered using GIN

Multirange type

- Represents a disjoint set of ranges

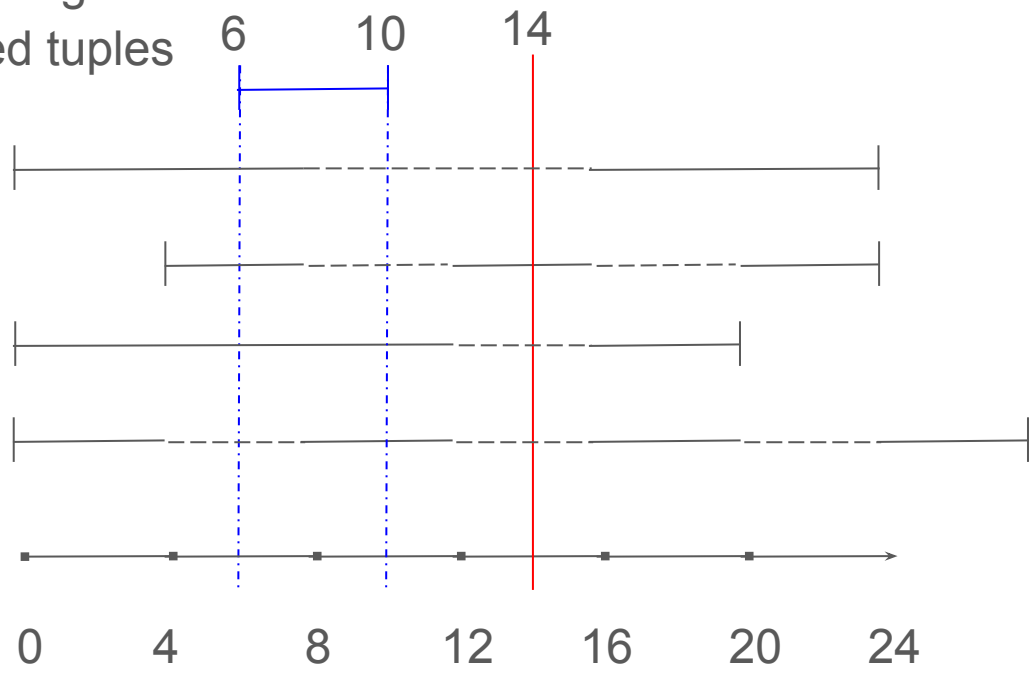
1	{[0, 8], [16, 24]}
2	{[4, 8], [12, 16], [20, 24]}
3	{[0, 12], [16, 20]}
4	{[0, 4], [8, 12], [16, 20], [24, 28]}



Multirange type - GiST Indexing

- Stored in the index as a single range
- We need to recheck the returned tuples

1	{[0, 8], [16, 24]}
2	{[4, 8], [12, 16], [20, 24]}
3	{[0, 12], [16, 20]}
4	{[0, 4], [8, 12], [16, 20], [24, 28]}



Problem: Too many false positives

▲ phamilton on Oct 27, 2022 | parent | next [–]

However, multirange GiST is implemented by merging all ranges into a single range (ignoring any gaps), which makes it less useful in many cases.

src: https://github.com/postgres/postgres/blob/f14aad5169baa5e2ac...

> ▲ Tostino on Oct 27, 2022 | root | parent | next [–]
>
> Right, but it just means the index is not used as the final source of truth and the DB has to run an exact check on the smaller subset of rows returned. You won't get incorrect queries, but depending on the workload it may not be a useful index for improving query speed.
>
> > ▲ phamilton on Oct 27, 2022 | root | parent | next [–]
> >
> > Yep. In our case we were using it for scheduling purposes and "Who is free on thursday at 2pm?" and we got very little narrowing because most people are available at least some time in the morning and some time in the evening.
> >
> > We've loosely landed at just materializing the dataset into another table by calling `unnest` in a trigger and then using an index on that table.

Subject: Re: range_agg
Date: 2020-12-27 19:38:38
Message-ID: CAPpHfdva3ZZwg-WO66O+F08TZZfz_ED=mbYbeG5-DsPohrzhaQ@mail.gmail.com
Views: Raw Message | Whole Thread | Download mbox | Resend email
Lists: pgsql-hackers

On Sun, Dec 27, 2020 at 9:07 PM David Fetter <david(at)fetter(dot)org> wrote:
> On Sun, Dec 27, 2020 at 09:53:13AM -0800, Zhihong Yu wrote:
> > This is not an ideal way to index multirages, but something we can
> > easily have.
>
> What sort of indexing improvements do you have in mind?

Approximation of multirange as a range can cause false positives.
It's good if gaps are small, but what if they aren't.

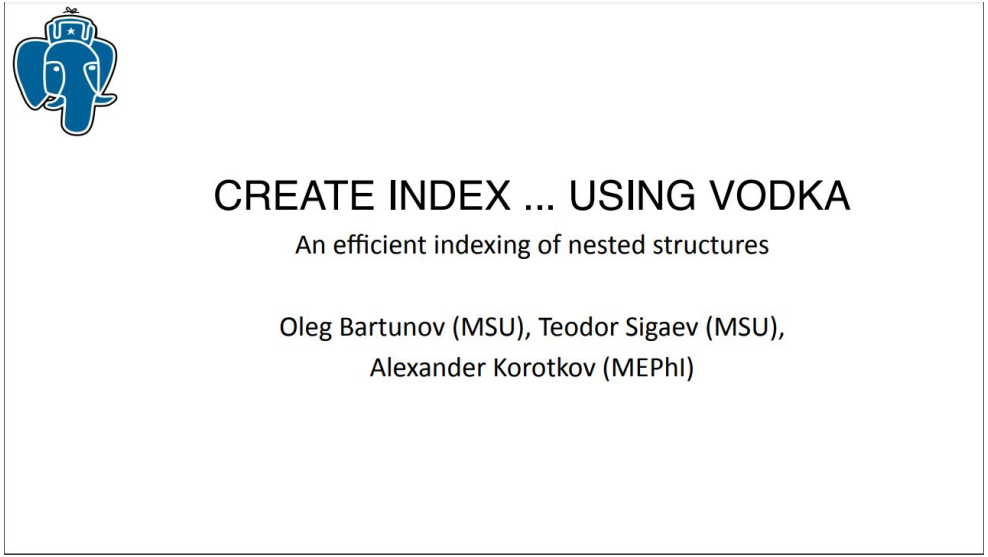
Ideally, we should split multirange to the ranges and index them
separately. So, we would need a GIN-like index. The problem is that
the GIN entry tree is a B-tree, which is not very useful for searching
for ranges. If we could replace the GIN entry tree with GiST or
SP-GiST, that should be good. We could index multirage parts
separately and big gaps wouldn't be a problem. Similar work was
already prototyped (it was prototyped under the name "vodka", but I'm
not a big fan of this name). FWIW, such a new access method would
need a lot of work to bring it to commit. I don't think it would be
reasonable, before multiranges get popular.

Regarding the GiST opclass, it seems the best we can do in GiST.

Regards,
Alexander Korotkov

CREATE INDEX … USING VODKA - PGCon 2014

“*We present a prototype of new access method, heavily based on GIN and optimized for efficient indexing of nested structures like hstore and json(b)*.”



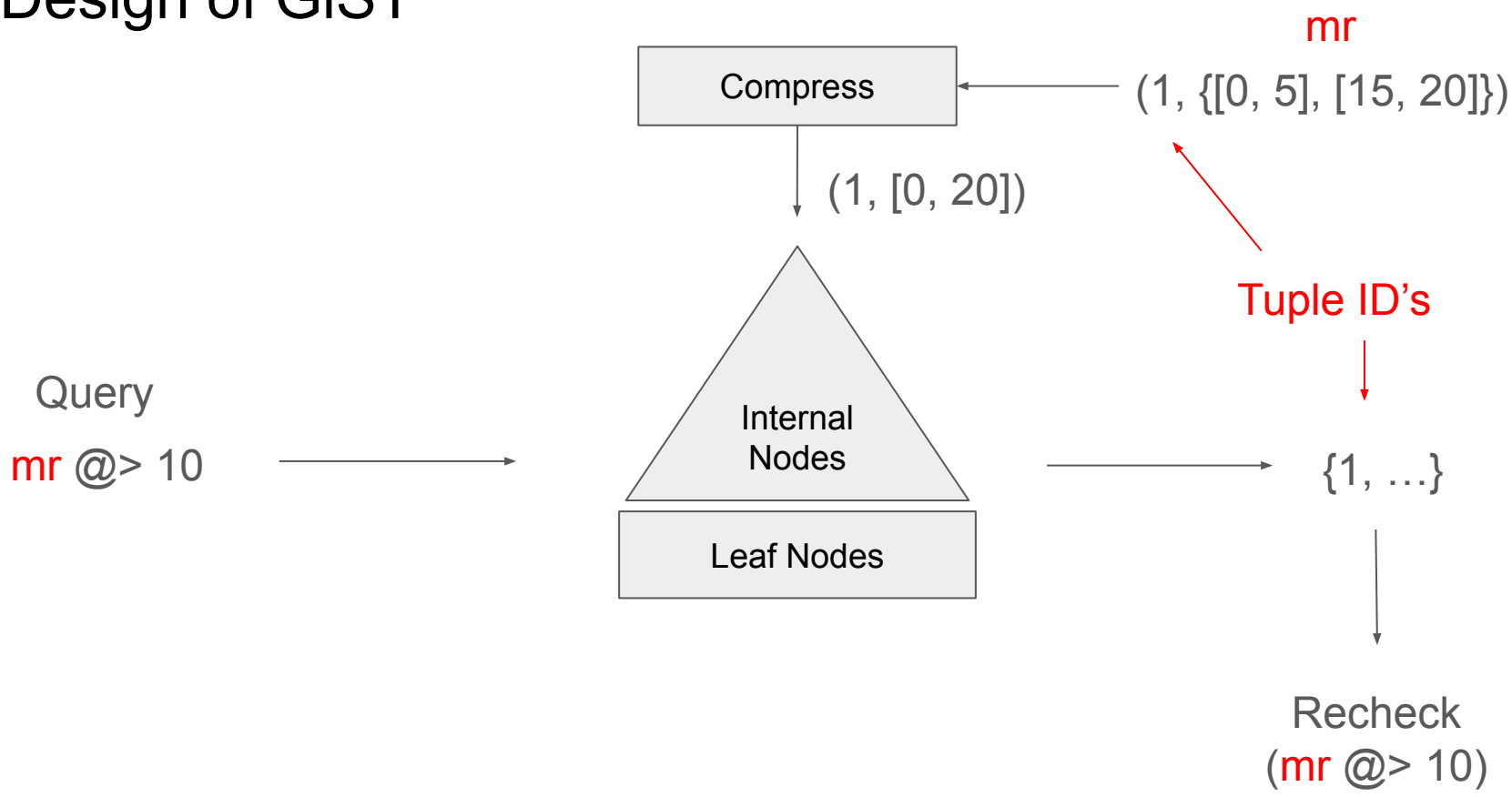
Multi-entry generalized search trees - MGiST

	Single-Entry	Multi-Entry
<, <=, =, >, >=	B-Tree	GIN
Custom Predicates	(SP-)GiST	**M(SP-)GiST**

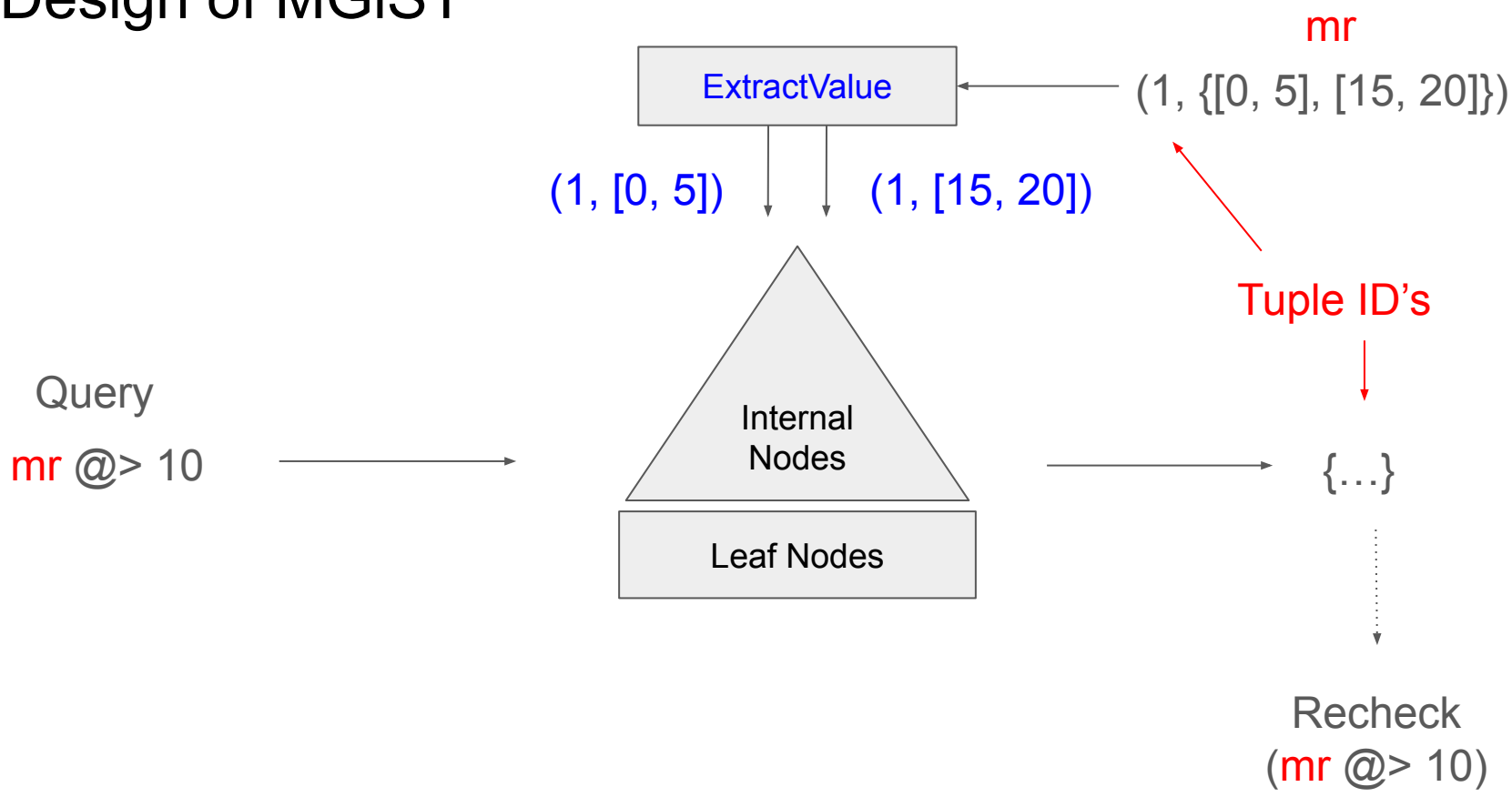
- New index access method

```
CREATE INDEX … USING M(SP)GIST
```

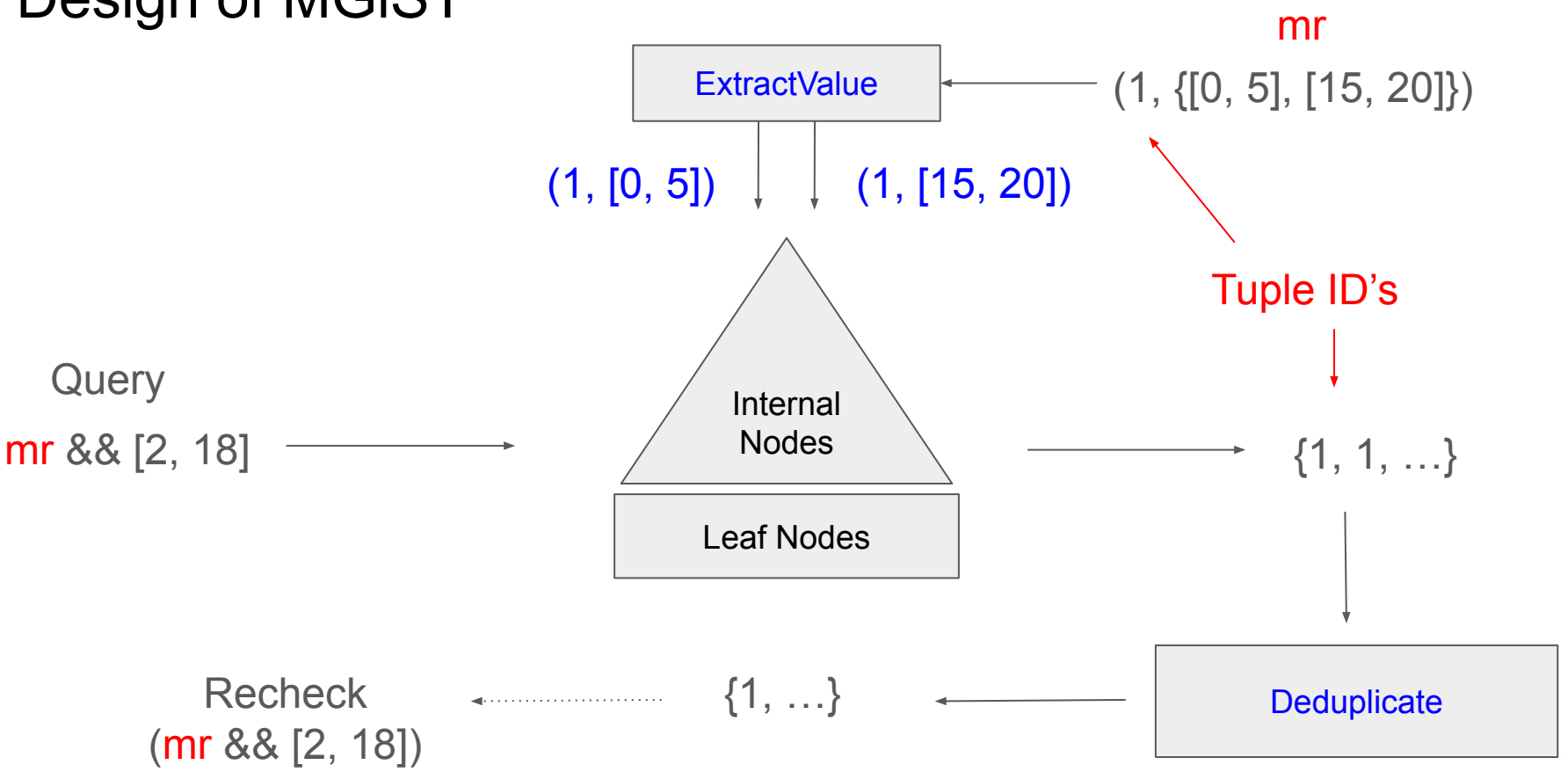
Design of GiST



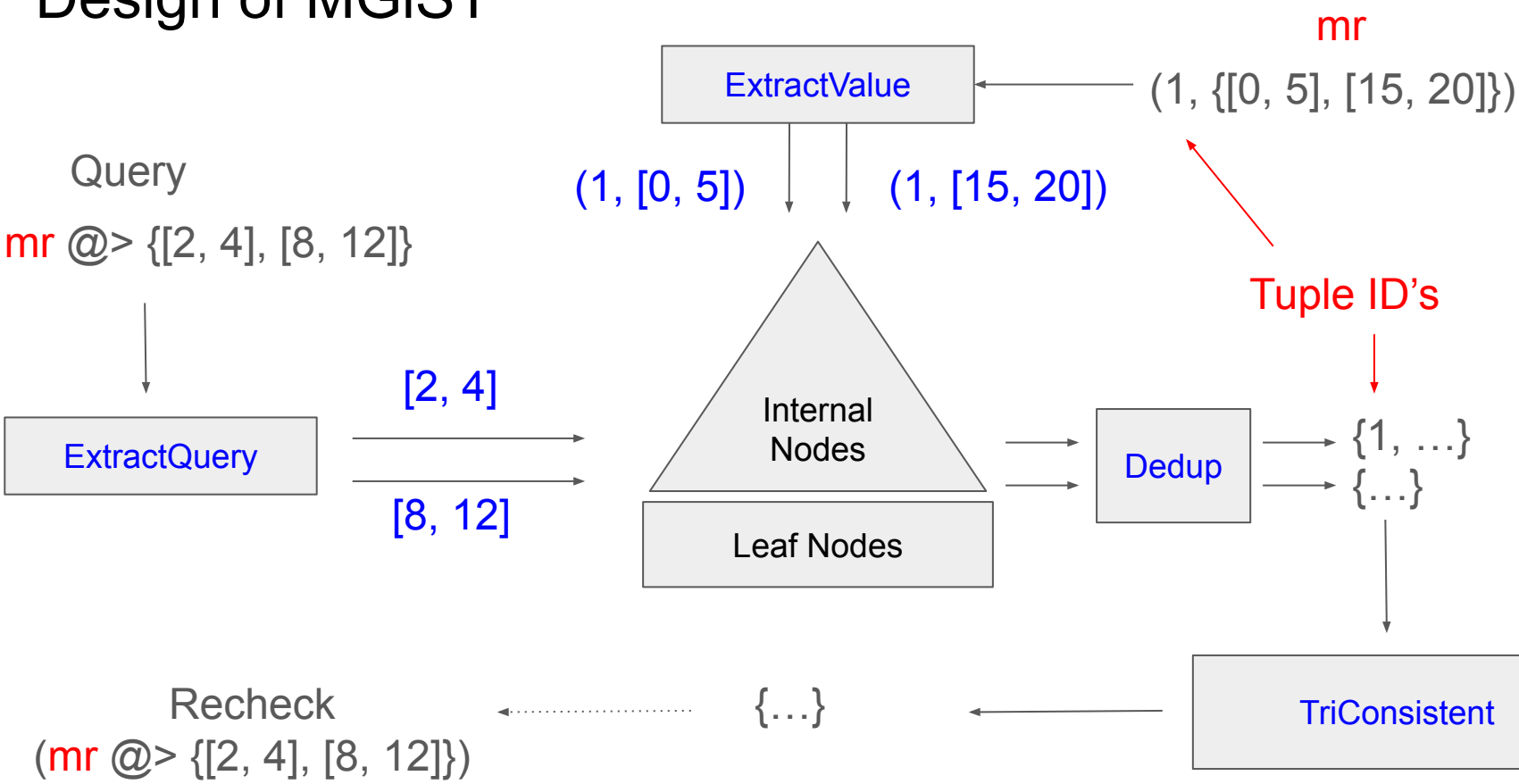
Design of MGiST



Design of MGiST



Design of MGiST



Multirange experiments

```
    Table "public.tmr"
 Column |      Type
--------+----------------
 mr     | int4multirange
```

```
    Table "public.tmr_g"
 Column |      Type
--------+----------------
 mr     | int4multirange
Indexes:
    "tmr_g_idx" gist (mr)
```

```
    Table "public.tmr_m"
 Column |      Type
--------+----------------
 mr     | int4multirange
Indexes:
    "tmr_m_idx" mgist (mr)
```

```
                List of relations
   Name    | Table | Access method |  Size
-----------+-------+---------------+---------
 tmr_g_idx | tmr_g | gist          | 600 kB
 tmr_m_idx | tmr_m | mgist         | 1688 kB
(2 rows)
```

```
                                mr
-------------------------------------------------------------------
 {[-8298,-8293),[5147,5156),[5425,5427)}
 {[1154,1164),[1214,1224),[2765,2767),[2974,2976),[5469,5476)}
 {[4474,4482),[7924,7929)}
 {[-7292,-7288),[9208,9218),[9918,9920),[9949,9951),[9980,9988)}
 {[7187,7189),[7650,7651)}
 {[-6393,-6390),[269,275),[9857,9866),[9884,9894),[9978,9986)}
 {[-4189,-4187),[9291,9293),[9469,9477)}
 {[-61,-56),[1195,1196)}
 {[-8769,-8761),[1952,1954)}
 {[-377,-370),[8023,8025),[9869,9879)}
 ...
```

Multirange experiments - sequential scan

```
explain (analyze, costs off) select count(*) from tmr where mr @> 0;
                           QUERY PLAN
-----------------------------------------------------------------
 Aggregate (actual time=1.707..1.708 rows=1 loops=1)
   ->  Seq Scan on tmr (actual time=0.053..1.705 rows=7 loops=1)
         Filter: (mr @> 0)
         Rows Removed by Filter: 9993
 Planning Time: 0.039 ms
 Execution Time: 1.725 ms
(6 rows)
```

- Query returns 7 rows out of 10 000
- We would expect indexing to speed up this query

Multirange experiments - GiST index

```
explain (analyze, costs off) select count(*) from tmr_g where mr @> 0;
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Aggregate (actual time=2.069..2.070 rows=1 loops=1)
   ->  Index Scan using tmr_g_idx on tmr_g (actual time=0.650..2.066 rows=7 loops=1)
         Index Cond: (mr @> 0)
         Rows Removed by Index Recheck: 4403
 Planning Time: 0.052 ms
 Execution Time: 2.088 ms
(6 rows)
```

- Too many false positives
- GiST index does not provide any speedup

Multirange experiments - MGiST index

```
explain (analyze, costs off) select count(*) from tmr_m where mr @> 0;
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Aggregate (actual time=0.065..0.065 rows=1 loops=1)
   ->  Index Scan using tmr_m_idx on tmr_m (actual time=0.056..0.062 rows=7 loops=1)
         Index Cond: (mr @> 0)
 Planning Time: 0.050 ms
 Execution Time: 0.084 ms
(5 rows)
```

- MGiST Index returns only 7 rows
- No recheck needed
- Speedup: **21x**

Multirange comparison operators

- Can be answered efficiently using GiST:
  - Strictly left / right of: <<, >>
  - Does not extend to the left / right of: &>, &<
  - Adjacent: -|-
  - Contained: <@ (anyrange)
- Can be answered efficiently using MGiST:
  - Overlaps: &&
  - Contains: @>
  - Contained by: <@ (anymultirange)

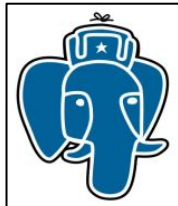
Indexing PostgreSQL data types

- B-Tree
  - Numeric types (integer, real)
  - Character types (char, varchar, text)
  - Date / Time types (date, time, timestamp)
- GiST / SP-GiST
  - Geometric types (point, line, box, polygon, path, circle)
  - Range and Multirange types
- GIN
  - Text search types (tsvector)
  - JSON types (jsonb)
  - Array types
- Hash / BRIN

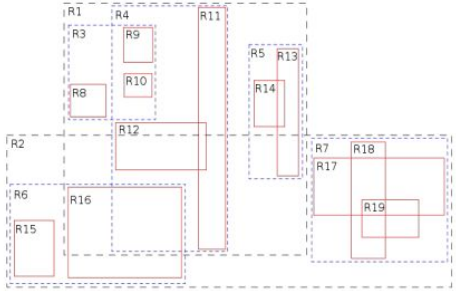
In some cases indexing these data types does not improve query performance enough

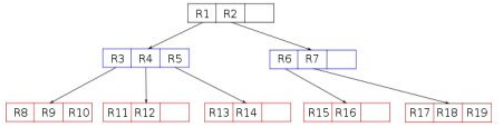
Custom array operators cannot always be answered using GIN

Other use-cases of MGiST - VODKA



Spaghetti indexing ...

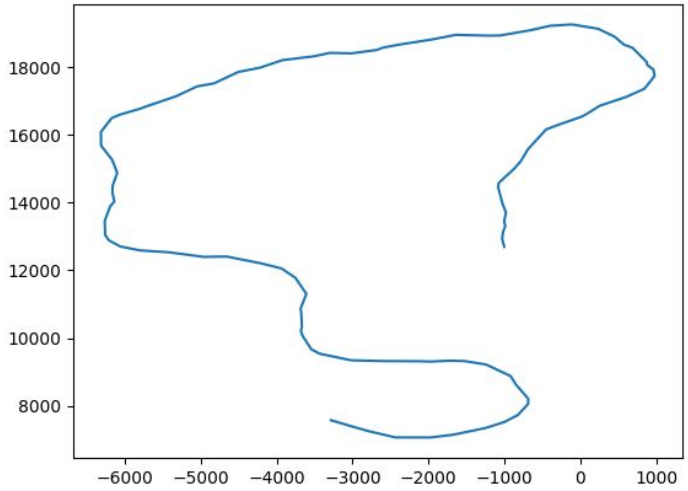


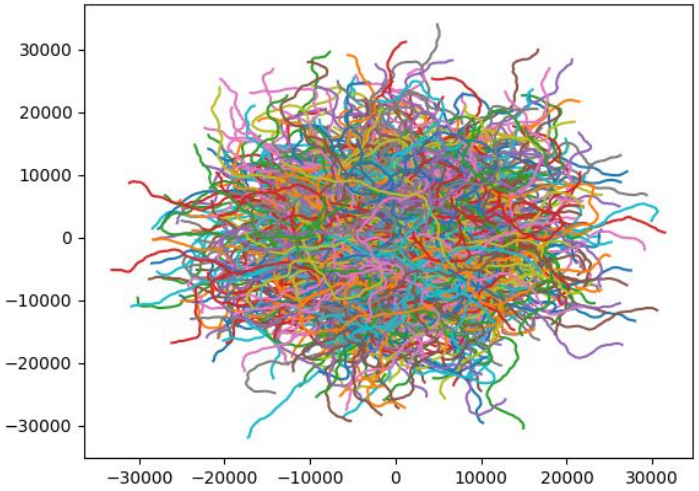


R-tree fails here — bounding box of each separate spaghetti is the same

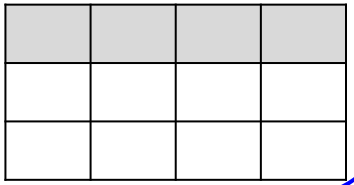
Other use-cases of MGiST - Path type

- Value table: **tp(id integer, p path)**
- Query: SELECT id FROM tp WHERE **p && box '...'**

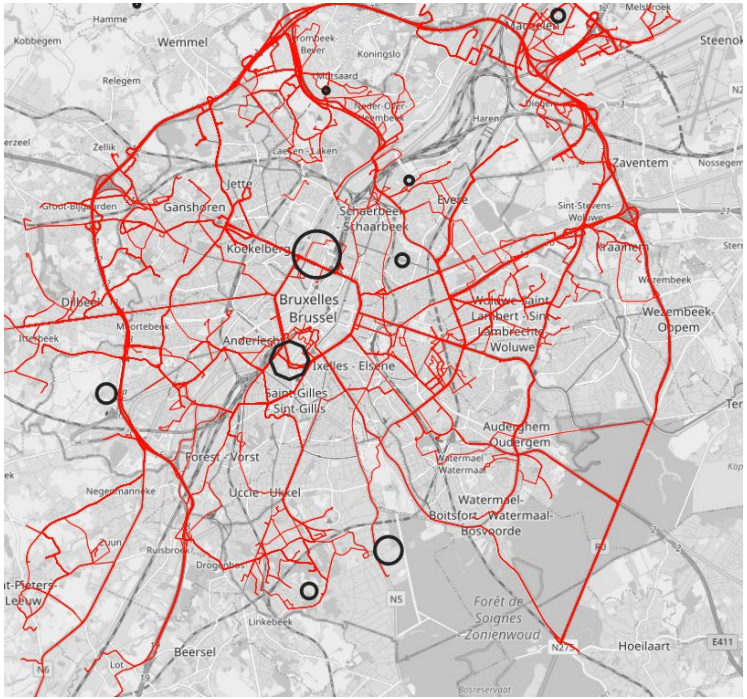


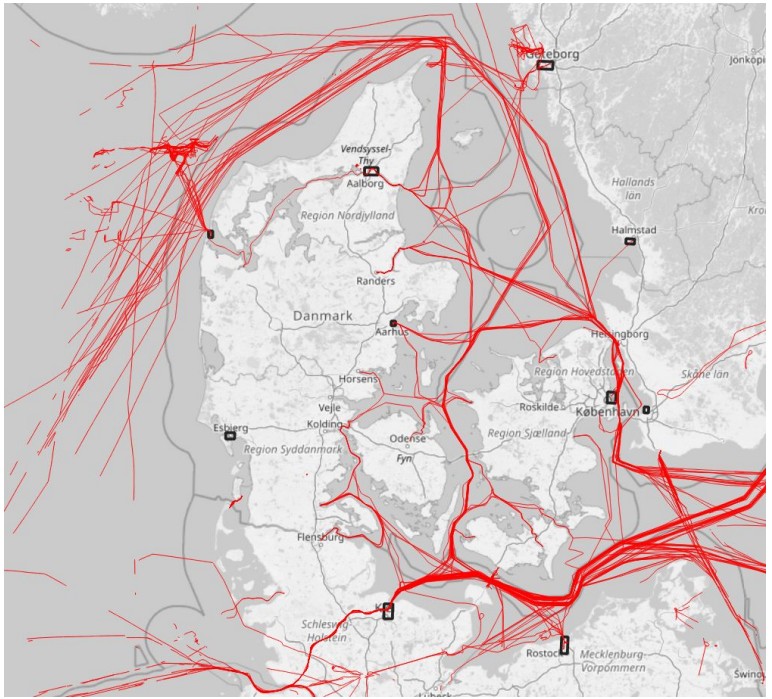


	PostgreSQL	PostGIS	MobilityDB
SQL Optimization	Scalar statistics & selectivity estimation	Grid-based statistics	Spatial grid + period bound histograms
Operations	Comparison, transformation, casting, ...etc	topological, CRS, properties, overlay, ...etc.	trajectory, temporal properties, lifted predicates, aggregations
Indexes	B-tree, hash, GiST, SP-GiST, GIN, BRIN	GiST, SP-GiST, BRIN	GiST, SP-GiST
Type System	numeric, character, date/time, bool, xml, json	Geometry, geography	tgeompoint, tgeogpoint, tint, tfloat, ttext, tbool



Trip data - examples





Trip data - PostGIS

mmsi	t	latitude	longitude	geom
2190045	2023-01-01 00:00:07+01	55.471807	8.423375	POINT(463548.4504248183 6147446.793007763)
2190045	2023-01-01 00:05:26+01	55.471787	8.423357	POINT(463547.2941072371 6147444.576729493)
2190045	2023-01-01 00:17:36+01	55.471802	8.423342	POINT(463546.3597296113 6147446.253876722)
2190045	2023-01-01 00:50:35+01	55.471818	8.423327	POINT(463545.4262755093 6147448.042309809)
2190045	2023-01-01 01:09:36+01	55.471835	8.423358	POINT(463547.40161387844 6147449.917916518)
...	...	...	...	...
2190064	2023-01-01 00:00:00+01	56.716572	11.519052	POINT(654168.7538896014 6288670.642298187)
2190064	2023-01-01 00:05:40+01	56.716572	11.519052	POINT(654168.7538896014 6288670.642298187)
2190064	2023-01-01 00:09:30+01	56.716572	11.519052	POINT(654168.7538896014 6288670.642298187)
2190064	2023-01-01 00:46:30+01	56.716572	11.519052	POINT(654168.7538896014 6288670.642298187)
2190064	2023-01-01 01:14:50+01	56.716572	11.519052	POINT(654168.7538896014 6288670.642298187)
...	...	...	...	...
2190072	2023-01-01 00:00:06+01	57.523658	9.963755	POINT(557720.1402536251 6376089.077306443)
2190072	2023-01-01 00:07:56+01	57.523658	9.963755	POINT(557720.1402536251 6376089.077306443)
2190072	2023-01-01 00:21:16+01	57.523658	9.963755	POINT(557720.1402536251 6376089.077306443)
2190072	2023-01-01 00:27:16+01	57.523658	9.963755	POINT(557720.1402536251 6376089.077306443)
2190072	2023-01-01 00:38:36+01	57.523658	9.963755	POINT(557720.1402536251 6376089.077306443)
...	...	...	...	...

~ 9K rows

Total rows: 7.5 M

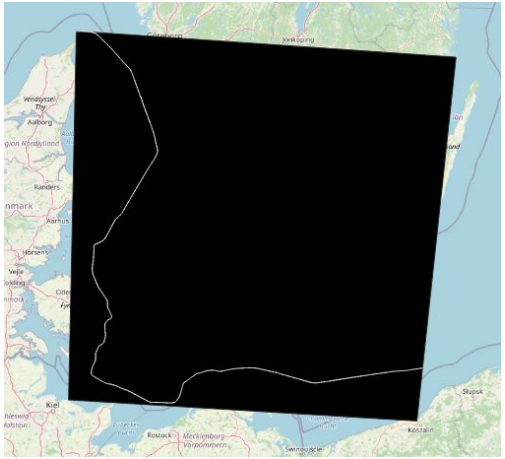
Trip data - MobilityDB

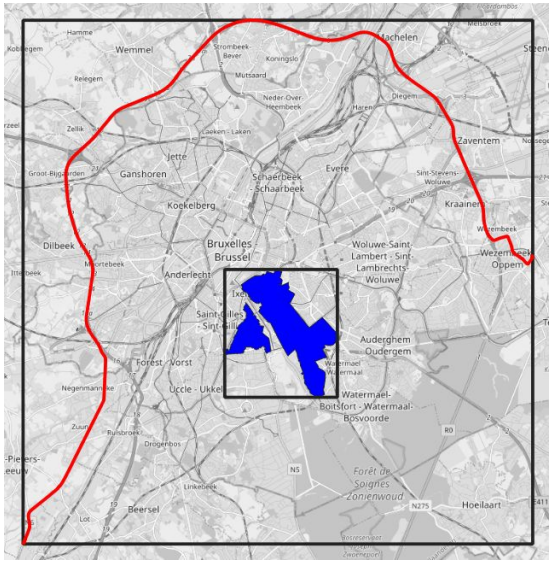
mmsi	tgeompoint
2190045	[POINT(463548.45 6147446.79)@2023-01-01 00:00:07+01, POINT(463548.45 6147446.79)@2023-01-0...
2190064	[POINT(654168.75 6288670.64)@2023-01-01 00:00:00+01, POINT(654168.75 6288670.64)@2023-01-0...
2190072	[POINT(557720.14 6376089.08)@2023-01-01 00:00:06+01, POINT(557720.14 6376089.08)@2023-01-0...
2190075	[POINT(603207.22 6098997.13)@2023-01-01 00:00:05+01, POINT(603207.22 6098997.13)@2023-01-0...
2190076	[POINT(531574.59 6169984.45)@2023-01-01 00:00:01+01, POINT(531574.59 6169984.45)@2023-01-0...
2194016	[POINT(874846.71 6128425.66)@2023-01-01 00:00:00+01, POINT(874846.83 6128425.67)@2023-01-0...
2573115	[POINT(539895.09 6566456.55)@2023-01-01 00:00:10+01, POINT(539894.81 6566456.55)@2023-01-0...
2655150	[POINT(682297.19 6398756.12)@2023-01-01 00:00:02+01, POINT(682297.19 6398756.12)@2023-01-0...
2655185	[POINT(857848.63 6246350.24)@2023-01-01 00:00:01+01, POINT(857848.34 6246349.99)@2023-01-0...
205011000	[POINT(417651.91 6264403.21)@2023-01-01 00:00:03+01, POINT(417626.26 6264366.19)@2023-01-0...
205198000	[POINT(452236.66 6284525.47)@2023-01-01 00:00:10+01, POINT(452236.84 6284525.25)@2023-01-0...
205465000	[POINT(484222.93 6344899.05)@2023-01-01 00:00:01+01, POINT(484239.63 6344909.11)@2023-01-0...
205482000	[POINT(652642.46 6370157.81)@2023-01-01 00:00:11+01, POINT(652606.61 6370195.8)@2023-01-01...
205538000	[POINT(729805.79 6216721.48)@2023-01-01 00:00:00+01, POINT(729805.79 6216721.48)@2023-01-0...
205688000	[POINT(518367.27 5969009.34)@2023-01-01 16:32:24+01, POINT(496005.38 5966799.8)@2023-01-01...
207124000	[POINT(602694.52 6386110)@2023-01-01 00:00:09+01, POINT(602692.43 6386114.4)@2023-01-01 00...
...	...

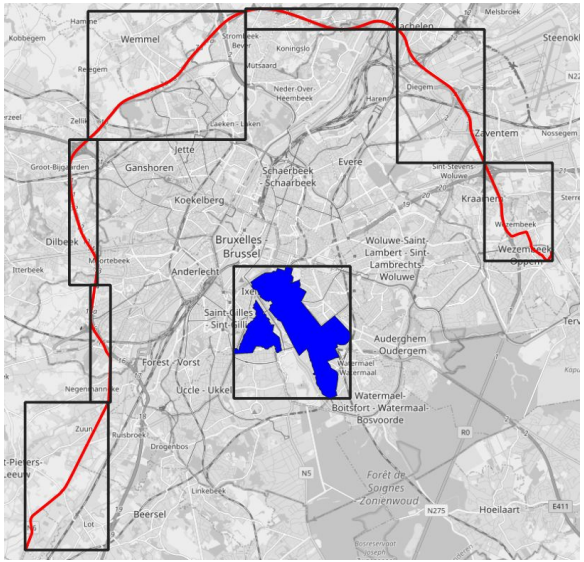
Total rows: 2.2 K

Trip data - indexing

- Problem:
  - Poor performance of (SP)GiST due to large bounding boxes
- Solution:
  - Split long trajectories into multiple smaller boxes (MGiST)







Trip data - GiST vs MGiST

```
explain (analyze, costs off)
     select distinct vehid, tripid from trips, points where edwithin(trip, geom, 10);
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Unique (actual time=30983.483..30983.749 rows=1630 loops=1)
   ->  Sort (actual time=30983.482..30983.547 rows=1791 loops=1)
         Sort Key: trips.vehid, trips.tripid
         Sort Method: quicksort  Memory: 132kB
         ->  Nested Loop (actual time=446.656..30982.354 rows=1791 loops=1)
               ->  Seq Scan on points (actual time=0.019..0.111 rows=100 loops=1)
               ->  Bitmap Heap Scan on trips (actual time=179.113..309.809 rows=18 loops=100)
                     Filter: edwithin(trip, points.geom, '10'::double precision)
                     Rows Removed by Filter: 942
                     Heap Blocks: exact=35443
                     ->  Bitmap Index Scan on trip_gist_idx
                         (actual time=0.800..0.800 rows=959 loops=100)
                           Index Cond: (trip && expandspace(points.geom, '10'::double precision))
 Planning Time: 0.334 ms
 Execution Time: 30983.831 ms (~31s)
(14 rows)
```

Trip data - GiST vs MGiST

```
explain (analyze, costs off)
     select distinct vehid, tripid from trips, points where edwithin(trip, geom, 10);
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Unique (actual time=2819.958..2820.219 rows=1630 loops=1)
   ->  Sort (actual time=2819.957..2820.023 rows=1791 loops=1)
         Sort Key: trips.vehid, trips.tripid
         Sort Method: quicksort  Memory: 132kB
         ->  Nested Loop (actual time=43.581..2819.304 rows=1791 loops=1)
               ->  Seq Scan on points (actual time=0.016..0.068 rows=100 loops=1)
               ->  Bitmap Heap Scan on trips (actual time=19.537..28.186 rows=18 loops=100)
                     Filter: edwithin(trip, points.geom, '10'::double precision)
                     Rows Removed by Filter: 41
                     Heap Blocks: exact=3219
                     ->  Bitmap Index Scan on trip_mgist_idx
                         (actual time=15.135..15.135 rows=61 loops=100)
                           Index Cond: (trip && expandspace(points.geom, '10'::double precision))
 Planning Time: 0.319 ms
 Execution Time: 2820.317 ms (~3s -> 10x speedup)
(14 rows)
```

Indexing PostgreSQL data types

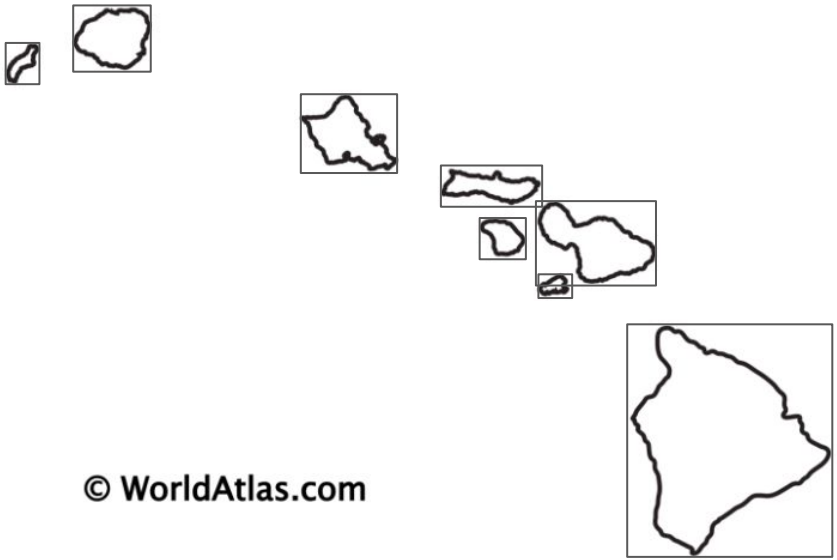
- B-Tree
  - Numeric types (integer, real)
  - Character types (char, varchar, text)
  - Date / Time types (date, time, timestamp)
- GiST / SP-GiST
  - Geometric types (point, line, box, polygon, path, circle)
  - Range and Multirange types
- GIN
  - Text search types (tsvector)
  - JSON types (jsonb)
  - Array types
- Hash / BRIN

In some cases indexing these data types does not improve query performance enough

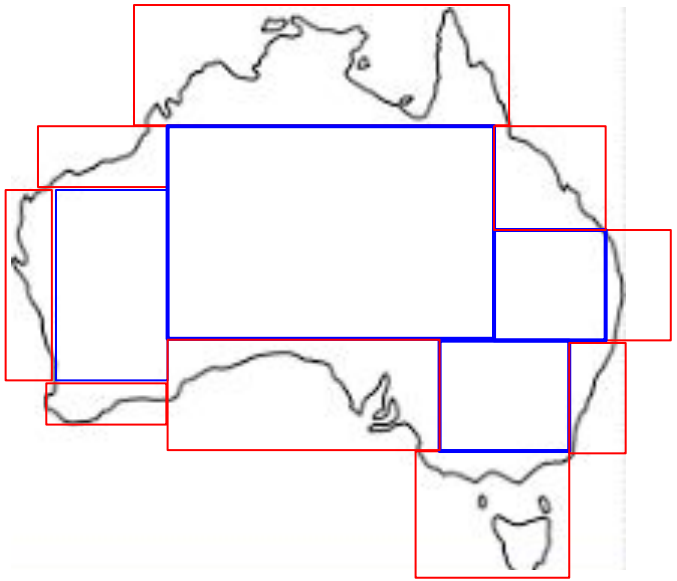
Custom array operators cannot always be answered using GIN

MGiST - More use-cases

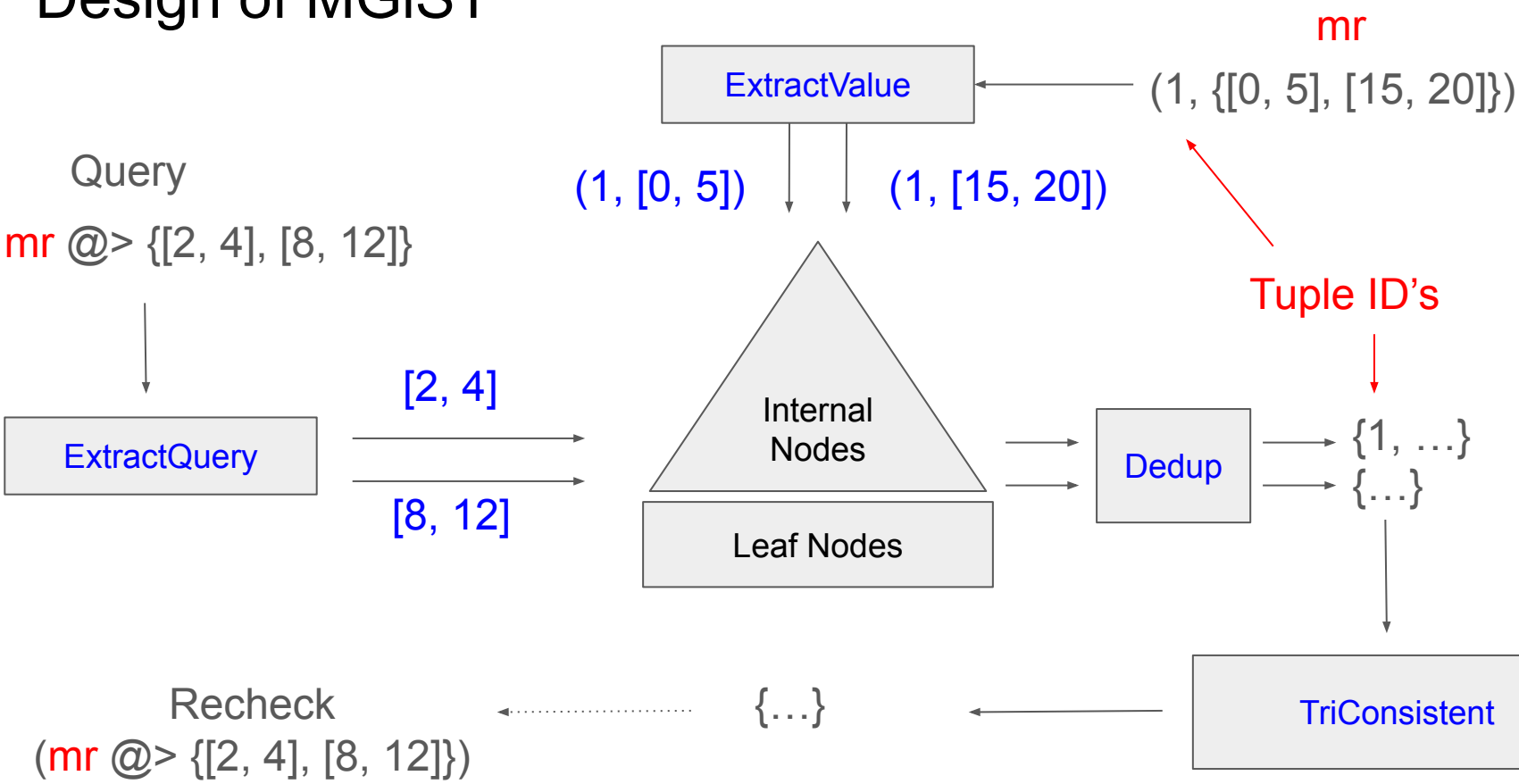
- Geometry Collections



- Point in polygon

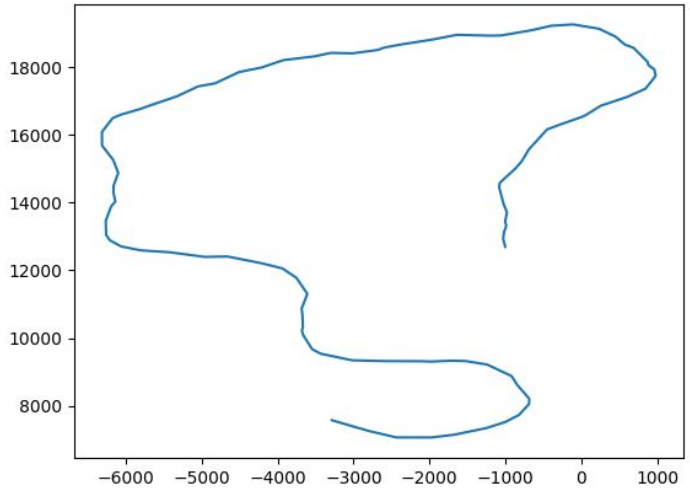


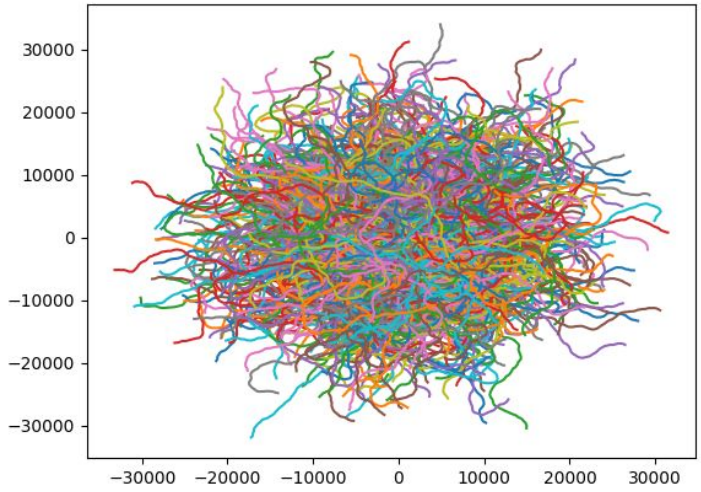
Design of MGiST



ExtractQuery - dataset

- Value table: **t(p path)** 5k trajs (10MB)
- Query table: **q(i integer, p path)** 100 trajs
- Query: SELECT i, count(*) FROM t, q WHERE **t.p && q.p** GROUP BY i





ExtractQuery - experiments

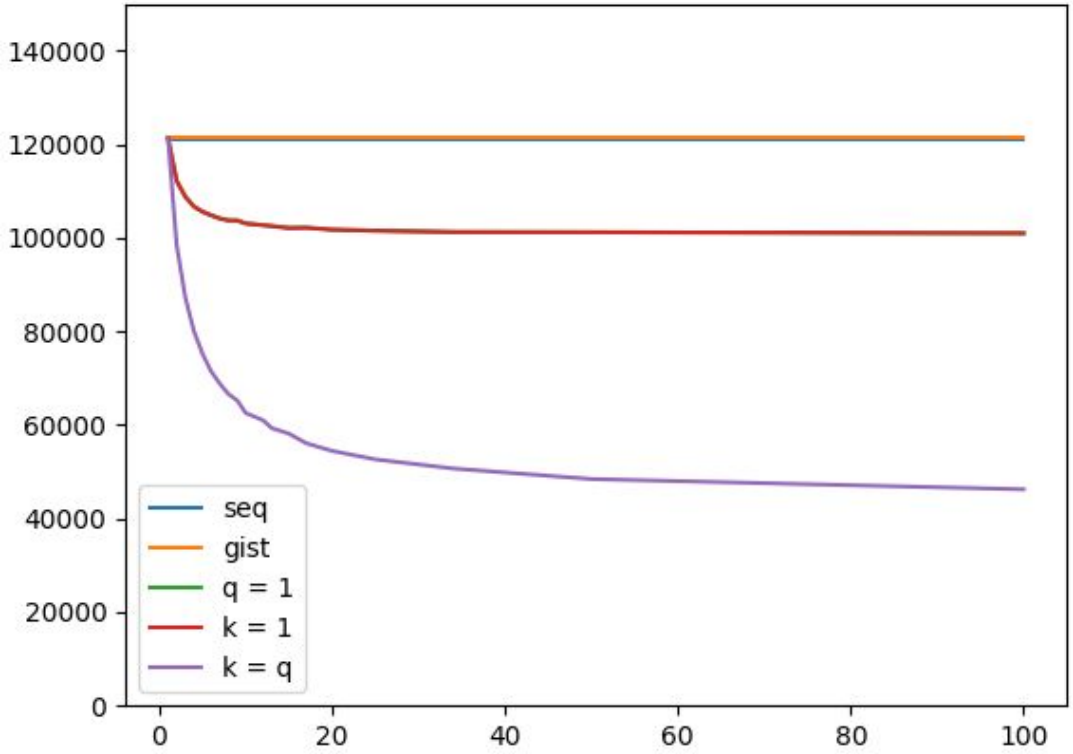
Possible n° boxes for extractValues and extractQuery is independent.

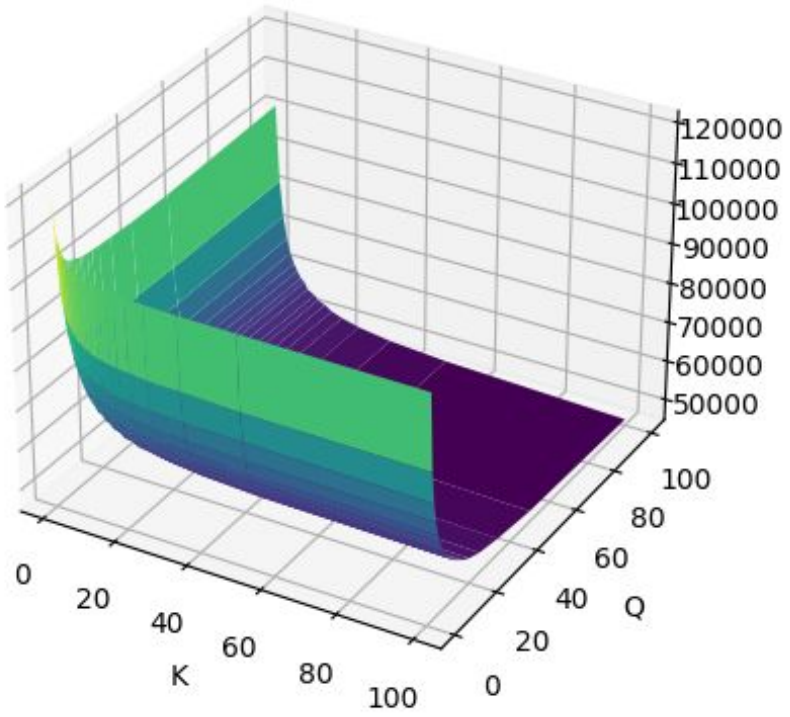
- K in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 15, 17, 20, 25, 34, 50, 100]
- Q in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 15, 17, 20, 25, 34, 50, 100]

Results:

- Sequential Scan: 121.2s
- GiST: 121.5s
- MGiST (K = 100, Q = 1) 101.0s
- MGiST (K = Q = 100) 46.2s

ExtractQuery experiments - results





https://github.com/MobilityDB/mest

README

Multi-Entry Search Trees for PostgreSQL

This directory contains implementations for the Multi-Entry GiST and SP-GiST access methods. These access methods are variations of the GiST and SP-GiST indices, allowing for more efficient indexing of complex and composite data types.

Contents

The repository contains 4 PostgreSQL extensions split into 4 separate folders:

- mgist:
  - contains the Multi-Entry GiST access method and an implementation of a multi-entry R-tree for the PostgreSQL *path* type.
- mgist-mobilitydb:
  - contains the implementation of a multi-entry R-tree for the MobilityDB *tgeompoint* type.
- mspgist:
  - contains the Multi-Entry SP-GiST access method.
- mspgist-mobilitydb:
  - contains the implementations of a multi-entry Quadtree and Kd-tree for the MobilityDB *tgeompoint* type.

For more information about each extension, please refer to their associated README file.