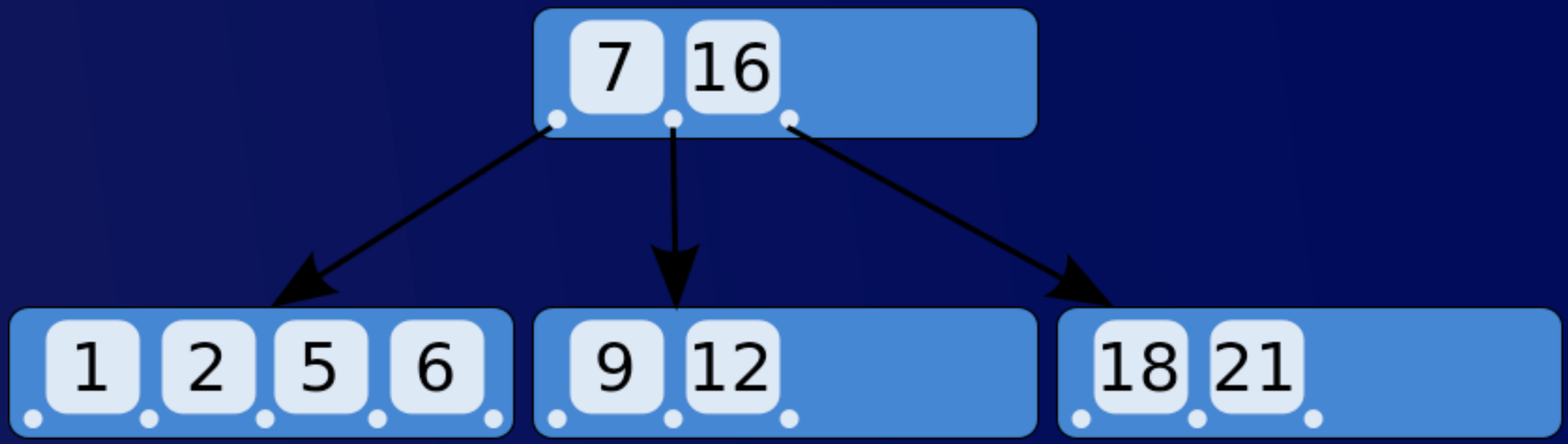
Beyond the B-Tree

Christophe Pettus
@xof

thebuild.com
pgexperts.com

Let us now praise famous data structures.

PGX
POSTGRESQL
EXPERTS, INC.



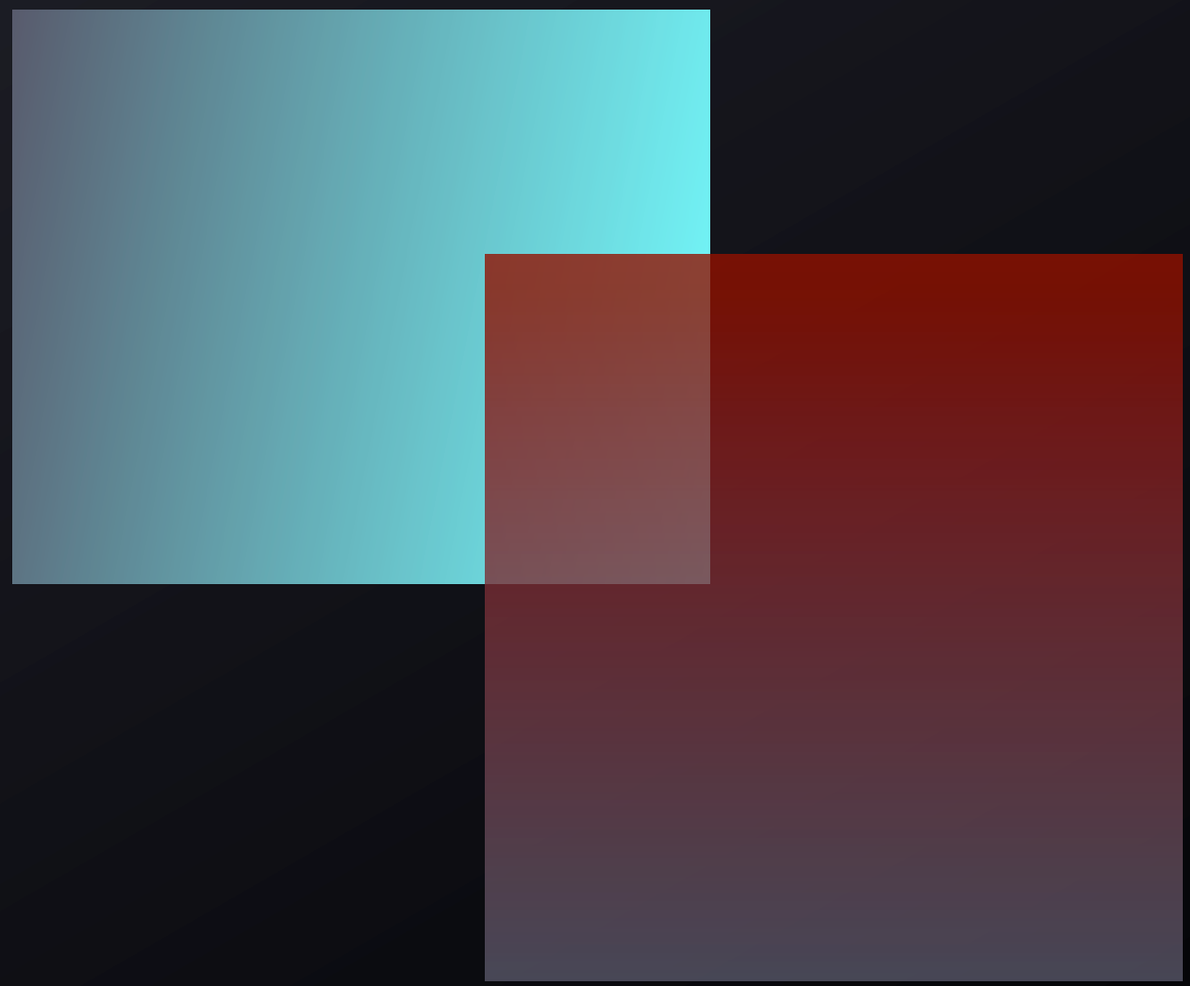
Thanks, wikipedia.

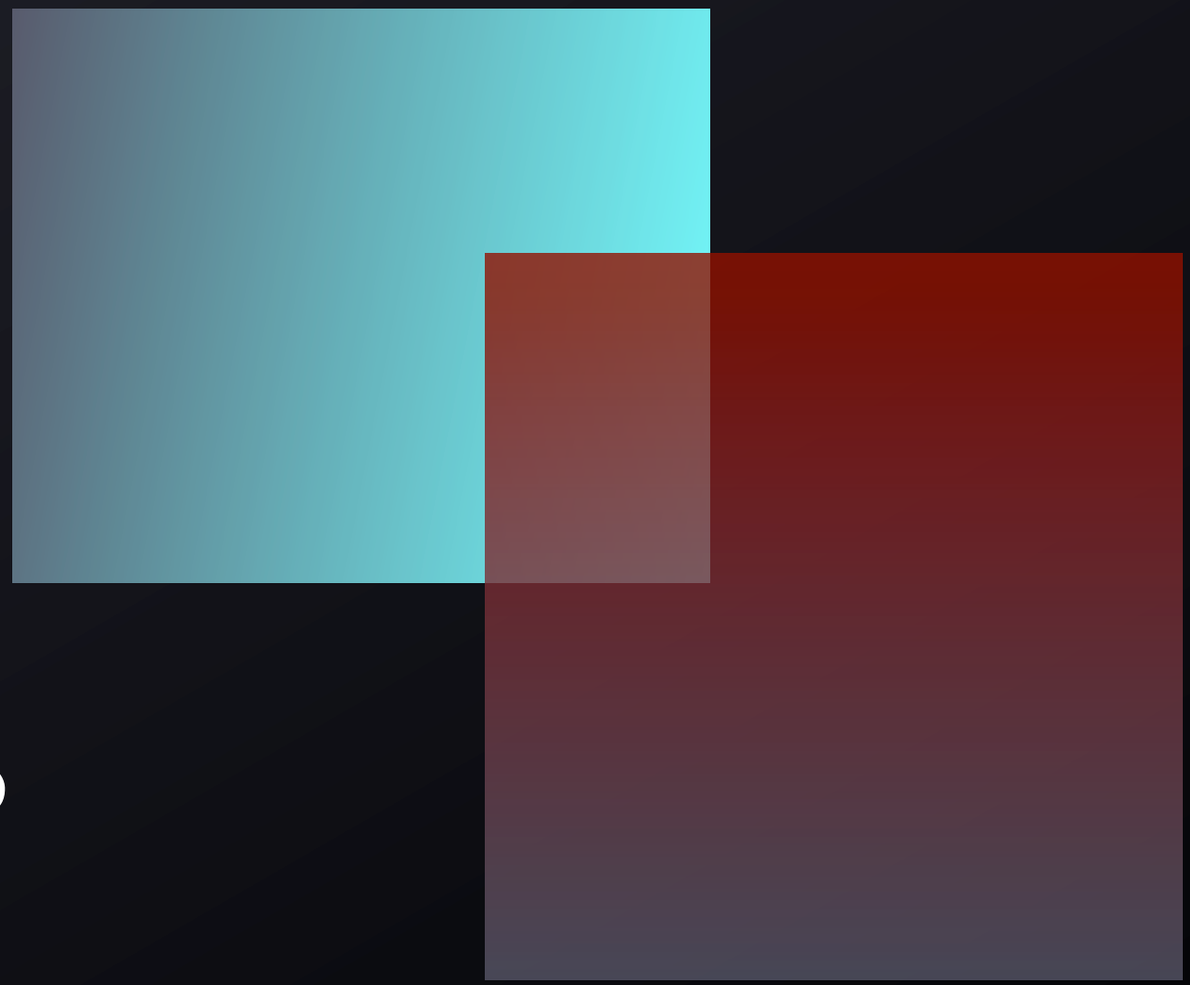
The B-Tree!

- Invented at Boeing Research Labs in 1971.
- Provides O(log *n*) access to leaf nodes on equality searches.
- Compact, well-understood, efficient.
- If only we knew what the "B" stands for!

Slices, dices, makes julienne fries.

- Can be used to traverse the index in forward- or reverse-sorted order.
- Also helps with comparison searches: <, >.
- Can work on any type that is totally ordered.
- So, problem solved!





> ?

Problem not quite solved.

- Not all types are totally ordered.
- Or, if you can define a total ordering, the total ordering is arbitrary or not naturally useful.
- Or, the type in question uses operators that don't map easily into direct comparison.

GIST and GIN!

- **G**eneralized **I**ndex **S**torage **T**echnique.
- **G**eneralized **I**nverted i**N**dex.
- Index frameworks, not single index types.
- Can be used for pretty much any type to support pretty much any operator it chooses to.

What's GIST good for?

- GIST indexes are generally used for types that partition a mathematical space.
  - Geometries, ranges, etc.
  - "Inside of," "Close to."
- Generally supports containment, distance, and similar operations.

What's GIN good for?

- Stores tokens, and pointers to the rows that contain those tokens.
- Good for inverted indexes such as full-text searches ("these are the rows with 'cat'").
- A "token" can be an array entry, JSON key/value, etc.: Any scalar value that PostgreSQL supports.

Just what it says on the tin.

- Each GIST / GIN index implementation specifies which operators it can support.
- Read the documentation!
- You can define indexes for your own types, too.
  - Some C language programming required.

PGX
POSTGRESQL
EXPERTS, INC.

Creating a GIST index.

- `CREATE INDEX ON t USING GIST(f);`
- You need to specify the USING GIST clause even if there's no other way to index the type.
- Selects the default operator class for that particular type.

Wait, what?

- It's possible to have more than one "operator class" for a particular type.
- Creates different kinds of indexes optimized for different kinds of queries.
- Read the documentation! Make sure you know what kinds of queries you want to do.

Why would we use this?

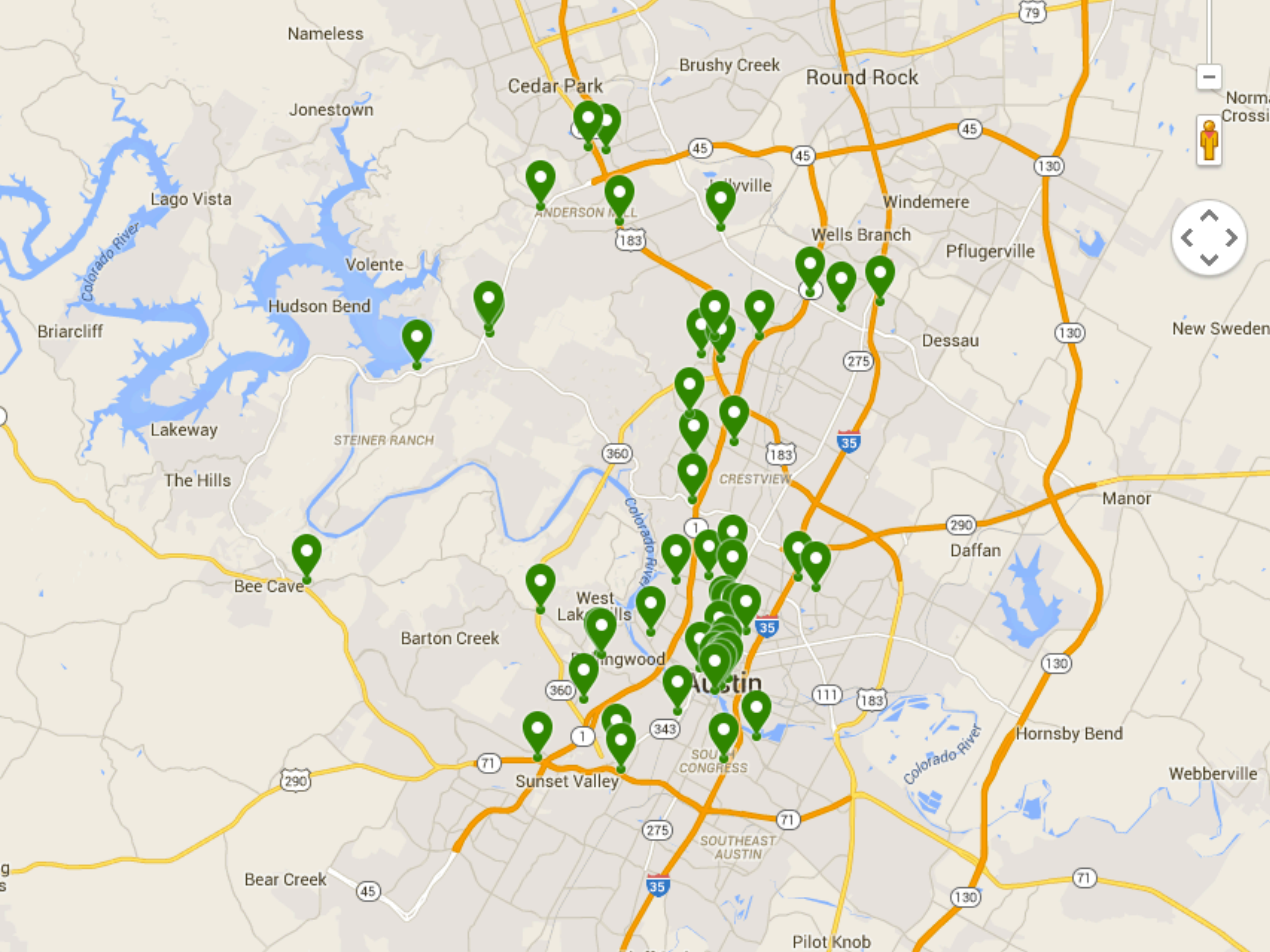
- Geographic data. (“What polygons are this point in?”)
- Range data. (“What date ranges overlap with this one?”)
- Similarity data. (“What phrases, in trigrams, are most similar to this one?”)

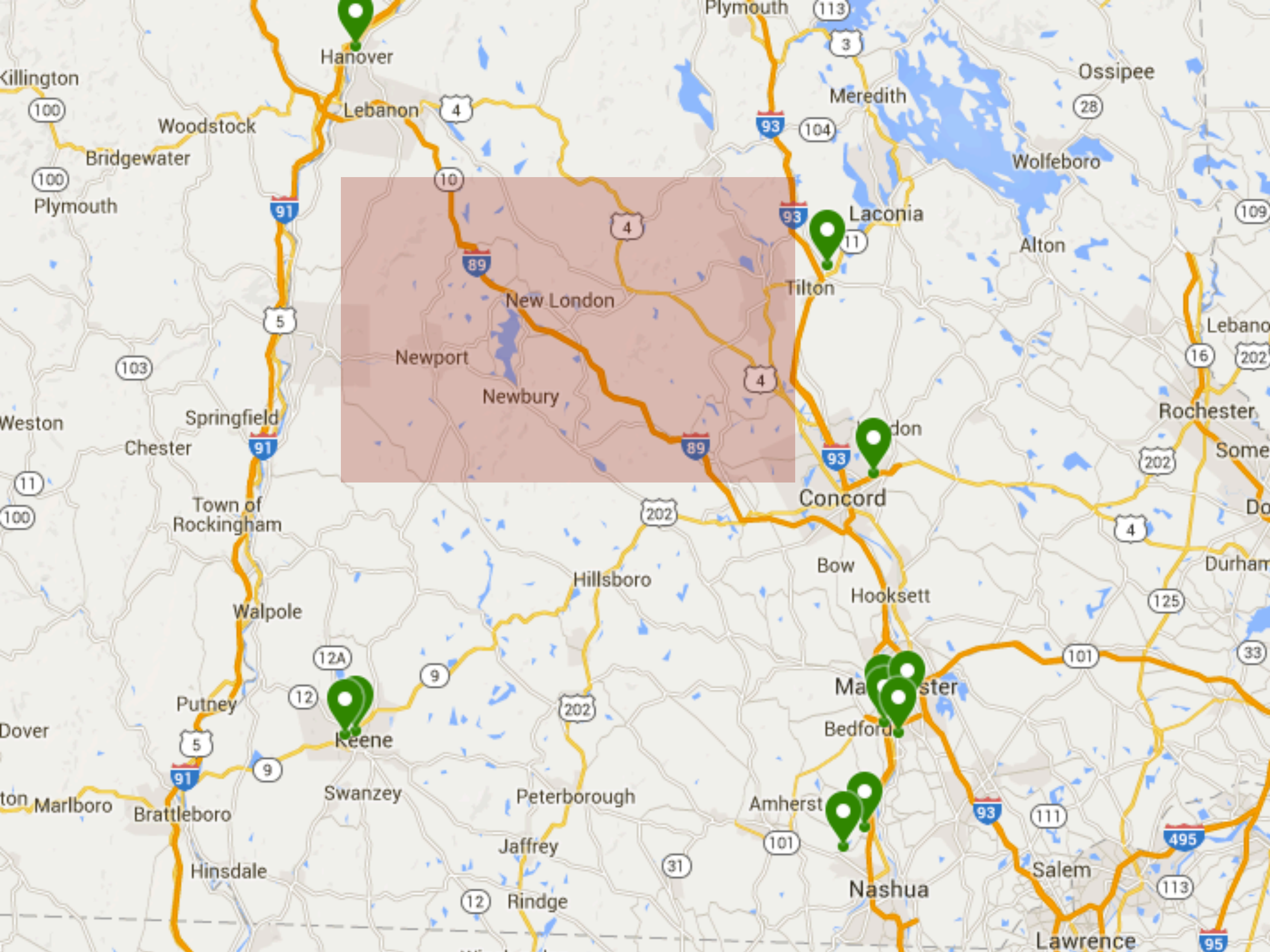
The Starbucks Problem.

- "Where are the nearest *n* Starbucks to this point?"
- This has to be an easy problem, right?
- I mean, what else are geo databases *for*?

Except it's not.

- "Find all points in the database within this bounding rectangle, then sort by distance."
- OK, great! We'll just… um… we'll… er…
- Um, how big should that rectangle be?


Nameless
Cedar Park
Brushy Creek
Round Rock
Jonestown
Lago Vista
Jollyville
Windemere
Wells Branch
Pflugerville
Colorado River
Volente
Hudson Bend
Briarcliff
Dessau
New Sweden
Lakeway
STEINER RANCH
The Hills
CRESTVIEW
Manor
Daffan
Bee Cave
West Lake Hills
Barton Creek
Austin
Hornsby Bend
Sunset Valley
SOUTH CONGRESS
Webberville
SOUTHEAST AUSTIN
Bear Creek
Pilot Knob
Colorado River


Hanover
Lebanon
Killington
Woodstock
Bridgewater
Plymouth
New London
Newport
Newbury
Springfield
Chester
Weston
Town of Rockingham
Walpole
Putney
Dover
Marlboro
Brattleboro
Hinsdale
Keene
Swanzey
Hillsboro
Peterborough
Jaffrey
Rindge
Plymouth
Meredith
Laconia
Tilton
Ossipee
Wolfeboro
Alton
Concord
Bow
Hooksett
Bedford
Amherst
Nashua
Salem
Lawrence
Rochester
Durham
Lebano
Some
Do

Problematic!

- Traditionally, had to do something like a binary search, store heuristics, or some other hack.
- But now, **KNN Indexes** to the rescue!

<-> operator

- If the GIST index for the type provides the <-> operator, you can use it for nearest neighbors:
  - SELECT id, store_loc<->point(43.45, -71.91)
    FROM stores
    ORDER BY store_loc<->point(43.45, -71.91)
    LIMIT 10;

Not just geometries!

- Any data that defines a <-> operator.
  - The type turns "how close?" into a scalar value for the indexing system.
- pg_tgrm does this for text similarity matching.
- Your own types! (Some C programming required.)

GIST on scalar types.

- You can (with an extension in contrib/) create GIST indexes on scalar types!
- You get <, ≤, =, ≥, > and <->.
- Why would you want to do that?

A Hard Problem.

- "Don't allow two bookings to be inserted into the database for the same room where the dates overlap."
- There's no way to express this using traditional UNIQUE constraints.
- Constraint Exclusion to the rescue!

One Catch.

- It has to be a single index.
- Since RANGE types require a GIST index…
- The index has to be a GIST index.
- By default, simple scalar values don't have GIST indexing.

Let's say we have...

```
xof=# \d reservations_booking
                                    Table
"public.reservations_booking"
 Column |          Type          |
Modifiers
--------+------------------------
+-----------------------------------------
----------
 id     | integer                | not null default
nextval('reservations_booking_id_seq'::regclass)
 room   | character varying(4) | not null
 dates  | daterange              | not null
Indexes:
    "reservations_booking_pkey" PRIMARY KEY, btree (id)
```

And add constraint index...

```
xof=# CREATE EXTENSION btree_gist;
CREATE EXTENSION
xof=# ALTER TABLE reservations_booking ADD EXCLUDE USING
GIST ( room WITH =, dates WITH && );
ALTER TABLE
```

And profit!

```
>>> Booking(room='123', dates=DateRange(date(2015,9,1),
date(2015,9,2))).save()
>>> Booking(room='123', dates=DateRange(date(2015,9,2),
date(2015,9,7))).save()
>>> Booking(room='127', dates=DateRange(date(2015,9,2),
date(2015,9,7))).save()
>>> Booking(room='123', dates=DateRange(date(2015,9,5),
date(2015,9,9))).save()
(blah blah blah)
IntegrityError: conflicting key value violates exclusion
constraint "reservations_booking_room_dates_excl"
DETAIL:  Key (room, dates)=(123, [2015-09-05,2015-09-09))
conflicts with existing key (room, dates)=(123,
[2015-09-02,2015-09-07)).
```

PGX
POSTGRESQL
EXPERTS, INC.

GIN Indexes. The La Brea Tar Pits.

- Maps "tokens" (arbitrary scalar values) to the rows that contain them.
- Most familiar use is in full-text search, mapping lexemes to the rows that contain them.
- Allow annotations (such as frequency) on index entries.

But the coolest use: JSON!

- PostgreSQL has two JSON types: json and jsonb.
- json stores the raw text of the json blob, whitespace and all.
- jsonb is a compact, indexable representation.

Why use json instead of jsonb?

- json (vs jsonb) is faster to insert, since it doesn't have to process the data.
- json allows for two highly dubious "features" (duplicate object keys, stable object key order).
- OK if you are just logging json that you don't plan to extensively query.

Why use jsonb instead of json?

- All other applications want jsonb.
- jsonb can be indexed in useful ways, unlike json.

jsonb indexing.

- jsonb has GIN indexing.
- Default operator class supports queries with the @>, ?, ?& and ?| operators.
- The query must be against the top-level object for the index to be useful.
- Can query nested objects, but only in paths rooted at the top level.

jsonb_path_ops

- Optional GIN index type for jsonb.
- Only supports @>.
- Hashes paths for each item, rather than just storing the key itself.
- Faster for @> operations with nesting.

jdoc @> '{"tags": ["qui"]}'

- Both index types support this.
- jsonb_ops (the default) will seach for everything that has “tags”, has “qui”, AND them, and then do a recheck for the path structure.
- jsonb_path_ops will go directly to entries for that path.

Which to use?

- If you just need @>, jsonb_path_ops will probably be faster.
- If you need the other supported operators, you need jsonb_ops.

Some caveats.

- GIST and GIN indexes can (traditionally) be big.
  - GIN posting list compression in 9.4 can make them super-small.
- They are not free to create and maintain.
- Like any index, only create them if you need them.

Questions?

Christophe Pettus
@xof

thebuild.com
pgexperts.com

PGX

POSTGRESQL

EXPERTS, INC.

Thank you!

Christophe Pettus
@xof

thebuild.com
pgexperts.com