

Non-Relational Postgres

BRUCE MOMJIAN



This talk explores the advantages of non-relational storage, and the Postgres support for such storage.

Creative Commons Attribution License

<http://momjian.us/presentations>

Last updated: November, 2018

Relational Storage

- ▶ Relational storage was proposed by E. F. Codd in 1970
- ▶ Very flexible, 40+ years still in use
- ▶ Not always ideal

What Is Relational Storage?

- ▶ Row, column, table (tuple, attribute, relation)
- ▶ Constraints
- ▶ **Normalization, joins**

https://en.wikipedia.org/wiki/Relational_database

What Is Data Normalization?

First Normal Form

- ▶ Each column/attribute contains only atomic indivisible values
- ▶ Eliminate repeating groups in individual tables
- ▶ Create a separate table for each set of related data
- ▶ Identify each set of related data with a primary key

https://en.wikipedia.org/wiki/First_normal_form

<http://www.anaesthetist.com/mnm/sql/normal.htm>

Downsides of First Normal Form

- ▶ Query performance
- ▶ Query complexity
- ▶ Storage inflexibility
- ▶ Storage overhead
- ▶ Indexing limitations

Postgres Non-Relational Storage Options

1. Arrays
2. Range types
3. Geometry
4. XML
5. JSON
6. JSONB
7. Row types
8. Character strings

1. Arrays

```
CREATE TABLE employee
(name TEXT PRIMARY KEY, certifications TEXT[]);
```

```
INSERT INTO employee
VALUES ('Bill', '{"CCNA", "ACSP", "CISSP}');
```

```
SELECT * FROM employee;
 name | certifications
-----+-----
 Bill | {CCNA,ACSP,CISSP}
```

```
SELECT name
FROM employee
WHERE certifications @> '{ACSP}';
 name
-----
 Bill
```

All queries used in this presentation are available at <http://momjian.us/main/writings/pgsql/non-relational.sql>.

Array Access

```
SELECT certifications[1]  
FROM employee;  
certifications
```

```
-----  
CCNA
```

```
SELECT unnest(certifications)  
FROM employee;  
unnest
```

```
-----  
CCNA  
ACSP  
CISSP
```


Array Unrolling

```
SELECT name, unnest(certifications)
FROM employee;
```

name	unnest
Bill	CCNA
Bill	ACSP
Bill	CISSP

Array Creation

```
SELECT DISTINCT relkind  
FROM pg_class  
ORDER BY 1;  
relkind
```

```
i  
r  
t  
v
```

```
SELECT array_agg(DISTINCT relkind)  
FROM pg_class;  
array_agg
```

```
{i,r,t,v}
```

2. Range Types

```
CREATE TABLE car_rental
(id SERIAL PRIMARY KEY, time_span TSTZRANGE);
```

```
INSERT INTO car_rental
VALUES (DEFAULT, '[2016-05-03 09:00:00, 2016-05-11 12:00:00)');
```

```
SELECT *
FROM car_rental
WHERE time_span @> '2016-05-09 00:00:00'::timestampz;
```

```
id | time_span
-----+-----
 1 | ["2016-05-03 09:00:00-04", "2016-05-11 12:00:00-04")
```

```
SELECT *
FROM car_rental
WHERE time_span @> '2018-06-09 00:00:00'::timestampz;
```

```
id | time_span
----+-----
```

Range Type Indexing

```
INSERT INTO car_rental (time_span)
SELECT tstzrange(y, y + '1 day')
FROM generate_series('2001-09-01 00:00:00'::timestampz,
    '2010-09-01 00:00:00'::timestampz, '1 day') AS x(y);
```

```
SELECT *
FROM car_rental
WHERE time_span @> '2007-08-01 00:00:00'::timestampz;
```

id	time_span
2162	["2007-08-01 00:00:00-04","2007-08-02 00:00:00-04")

```
EXPLAIN SELECT *
FROM car_rental
WHERE time_span @> '2007-08-01 00:00:00'::timestampz;
```

QUERY PLAN

```
-----...
Seq Scan on car_rental (cost=0.00..64.69 rows=16 width=36)
  Filter: (time_span @> '2007-08-01 00:00:00-04'::timestampz)
```

Range Type Indexing

```
CREATE INDEX car_rental_idx ON car_rental
USING GIST (time_span);
```

```
EXPLAIN SELECT *
FROM car_rental
WHERE time_span @> '2007-08-01 00:00:00'::timestampz;
                                QUERY PLAN
```

```
-----
Index Scan using car_rental_idx on car_rental (cost=0.15..8.17...
  Index Cond: (time_span @> '2007-08-01 00:00:00-04'::timestamp...
```

Exclusion Constraints

```
ALTER TABLE car_rental ADD EXCLUDE USING GIST (time_span WITH &&);
```

```
INSERT INTO car_rental
```

```
VALUES (DEFAULT, '[2003-04-01 00:00:00, 2003-04-01 00:00:01)');
```

```
ERROR: conflicting key value violates exclusion constraint "car...
```

```
DETAIL: Key (time_span)=(["2003-04-01 00:00:00-05", "2003-04-01 ...  
with existing key (time_span)=(["2003-04-01 00:00:00-05", "2003-...
```

3. Geometry

```
CREATE TABLE dart (dartno SERIAL, location POINT);
```

```
INSERT INTO dart (location)
SELECT CAST('(' || random() * 100 || ',' ||
          random() * 100 || ')') AS point)
FROM generate_series(1, 1000);
```

```
SELECT *
FROM dart
LIMIT 5;
```

dartno	location
1	(60.1593657396734,64.1712633892894)
2	(22.9252253193408,38.7973457109183)
3	(54.7123382799327,16.1387695930898)
4	(60.5669556651264,53.1596980988979)
5	(22.7800350170583,90.8143546432257)

Geometry Restriction

```
-- find all darts within four units of point (50, 50)
```

```
SELECT *
```

```
FROM dart
```

```
WHERE location <@ '<(50, 50), 4>'::circle;
```

dartno	location
308	(52.3920683190227,49.3803130928427)
369	(52.1113255061209,52.9995835851878)
466	(47.5943599361926,49.0266934968531)
589	(46.3589935097843,50.3238912206143)
793	(47.3468563519418,50.0582652166486)

```
EXPLAIN SELECT *
```

```
FROM dart
```

```
WHERE location <@ '<(50, 50), 4>'::circle;
```

```
QUERY PLAN
```

```
-----  
Seq Scan on dart (cost=0.00..19.50 rows=1 width=20)  
  Filter: (location <@ '<(50,50),4>'::circle)
```


Indexed Geometry Restriction

```
CREATE INDEX dart_idx ON dart
USING GIST (location);
```

```
EXPLAIN SELECT *
FROM dart
WHERE location <@ '<(50, 50), 4>'::circle;
                                QUERY PLAN
```

```
-----...
Index Scan using dart_idx on dart (cost=0.14..8.16 rows=1 ...
  Index Cond: (location <@ '<(50,50),4>'::circle)
```

Geometry Indexes with LIMIT

```
-- find the two closest darts to (50, 50)
```

```
SELECT *  
FROM dart  
ORDER BY location <-> '(50, 50)::point'  
LIMIT 2;
```

dartno	location
308	(52.3920683190227,49.3803130928427)
466	(47.5943599361926,49.0266934968531)

```
EXPLAIN SELECT *  
FROM dart  
ORDER BY location <-> '(50, 50)::point'  
LIMIT 2;
```

QUERY PLAN

```
-----...  
Limit (cost=0.14..0.33 rows=2 width=20)  
-> Index Scan using dart_idx on dart (cost=0.14..92.14...  
Order By: (location <-> '(50,50)::point))
```

4. XML

```
$ # Run with foomatic installed, or download:
$ # https://www.openprinting.org/download/foomatic/foomatic-db-4.0-current.tar.gz
$ cd /usr/share/foomatic/db/source/opt
$ for FILE in *.xml
do    tr -d '\n' < "$FILE"
      echo
done > /tmp/foomatic.xml
```

```
$ psql
CREATE TABLE printer (doc XML);

COPY printer from '/tmp/foomatic.xml';
```

Xpath Query

```
SELECT (xpath('/option/arg_shortcode/en', doc))  
FROM printer  
LIMIT 5;
```

xpath

{<en>Dithering</en>}
{<en>BottomMargin</en>}
{<en>Uniform</en>}
{<en>Cur1CorrectionAlways</en>}
{<en>Encoding</en>}

Remove XML Array

```
SELECT (xpath('/option/arg_shortname/en', doc))[1]  
FROM printer  
LIMIT 5;
```

xpath

<en>Dithering</en>
<en>BottomMargin</en>
<en>Uniform</en>
<en>CurlCorrectionAlways</en>
<en>Encoding</en>

Xpath to XML Text

```
-- convert to XML text
SELECT (xpath('/option/arg_shortcode/en/text()', doc))[1]
FROM printer
LIMIT 5;
      xpath
```

```
-----
Dithering
BottomMargin
Uniform
CurlCorrectionAlways
Encoding
```

Xpath to SQL Text

```
-- convert to SQL text so we can do DISTINCT and ORDER BY
SELECT DISTINCT text((xpath('/option/arg_shortname/en/text()', doc))[1])
FROM printer
ORDER BY 1
LIMIT 5;
    text
-----
AlignA
AlignB
AlignC
AlignD
AllowReprint
```

XML Non-Root Query

```
SELECT xpath('//driver/text()', doc)
FROM printer
ORDER BY random()
LIMIT 5;
      xpath
```

```
-----
{bjc600,bjc800,hpdj}
{hl1250}
{oki182}
{bjc600,bjc800}
{Postscript1}
```


Unnest XML Arrays

```
SELECT DISTINCT unnest((xpath('//driver/text()', doc))::text[])
FROM printer
ORDER BY 1
LIMIT 5;
  unnest
-----
ap3250
appledmp
bj10
bj10e
bj10v
```

Search XML Text

```
WITH driver (name) AS (  
    SELECT DISTINCT unnest(xpath('//driver/text()', doc))::text  
    FROM printer  
)  
SELECT name  
FROM driver  
WHERE name LIKE 'hp%'  
ORDER BY 1;  
    name  
-----  
    hpdj  
    hpijs  
    hpijs-pcl3  
    hpijs-pcl5c  
    hpijs-pcl5e
```

5. JSON Data Type

- ▶ JSON data type, not to be confused with JSONB
- ▶ Similar to XML in that the JSON is stored as text and validated
- ▶ ~100 JSON functions

Load JSON Data

```
-- download sample data from https://www.mockaroo.com/  
-- remove 'id' column, output as JSON, uncheck 'array'  
CREATE TABLE friend (id SERIAL, data JSON);
```

```
COPY friend (data) FROM '/tmp/MOCK_DATA.json';
```

```
SELECT *  
FROM friend  
ORDER BY 1  
LIMIT 2;
```

id	data
1	{"gender": "Male", "first_name": "Eugene", "last_name": "Reed", ...
2	{"gender": "Female", "first_name": "Amanda", "last_name": "Morr...

Pretty Print JSON

```
SELECT id, jsonb_pretty(data::jsonb)
FROM friend
ORDER BY 1
LIMIT 1;
```

id	jsonb_pretty
1	{ "email": "ereed0@businesswire.com", "gender": "Male", "last_name": "Reed", "first_name": "Eugene", "ip_address": "46.168.181.79" }

Access JSON Values

```
SELECT data->>'email'  
FROM friend  
ORDER BY 1  
LIMIT 5;
```

?column?

```
aaalexandere0@europa.eu  
aalvarezdk@miitbeian.gov.cn  
aandrewsd9@usda.gov  
aarmstrong61@samsung.com  
abarnes55@de.vu
```

Concatenate JSON Values

```
SELECT data->>'first_name' || ' ' ||  
       (data->>'last_name')  
FROM friend  
ORDER BY 1  
LIMIT 5;  
    ?column?
```

```
-----  
Aaron Alvarez  
Aaron Murphy  
Aaron Rivera  
Aaron Scott  
Adam Armstrong
```

JSON Value Restrictions

```
SELECT data->>'first_name'  
FROM friend  
WHERE data->>'last_name' = 'Banks'  
ORDER BY 1;  
?column?
```

```
-----  
Bruce  
Fred
```

```
-- the JSON way  
SELECT data->>'first_name'  
FROM friend  
WHERE data::jsonb @> '{"last_name" : "Banks"}'  
ORDER BY 1;  
?column?
```

```
-----  
Bruce  
Fred
```


Single-Key JSON Index

```
-- need double parentheses for the expression index
CREATE INDEX friend_idx ON friend ((data->>'last_name'));
```

```
EXPLAIN SELECT data->>'first_name'
FROM friend
WHERE data->>'last_name' = 'Banks'
ORDER BY 1;
```

QUERY PLAN

```
-----...
Sort (cost=12.89..12.90 rows=3 width=123)
  Sort Key: ((data ->> 'first_name'::text))
  -> Bitmap Heap Scan on friend (cost=4.30..12.87 rows=3 width=123)
    Recheck Cond: ((data ->> 'last_name'::text) = 'Banks'::text)
    -> Bitmap Index Scan on friend_idx (cost=0.00..4.30 rows=3 ...)
      Index Cond: ((data ->> 'last_name'::text) = 'Banks'::t...
```

JSON Calculations

```
SELECT data->>'first_name' || ' ' || (data->>'last_name'),
       data->>'ip_address'
FROM friend
WHERE (data->>'ip_address')::inet <= '172.0.0.0/8'::cidr
ORDER BY 1;
```

?column?		?column?
-----+-----		
Lisa Holmes		172.65.223.150
Walter Miller		172.254.148.168

```
SELECT data->>'gender', COUNT(data->>'gender')
FROM friend
GROUP BY 1
ORDER BY 2 DESC;
```

?column?		count
-----+-----		
Male		507
Female		493

6. JSONB

Like the JSON data type, except:

- ▶ Values are native JavaScript data types: text, number, boolean, null, subobject
- ▶ Indexing of all keys and values
- ▶ Stored in compressed format
- ▶ Sorts keys to allow binary-search key look up
- ▶ Does not preserve key order
- ▶ Does not preserve whitespace syntax
- ▶ Retains only the last duplicate key

hstore is similar non-hierarchical key/value implementation.

JSON vs. JSONB Data Types

```
SELECT '{"name" : "Jim", "name" : "Andy", "age" : 12}'::json;  
      json
```

{**"name"** : "Jim", **"name"** : "Andy", "age" : 12}

```
SELECT '{"name" : "Jim", "name" : "Andy", "age" : 12}'::jsonb;  
      jsonb
```

{**"age"**: 12, "name": **"Andy"**}

JSONB Index

```
CREATE TABLE friend2 (id SERIAL, data JSONB);
```

```
INSERT INTO friend2  
SELECT * FROM friend;
```

```
-- jsonb_path_ops indexes are smaller and faster,  
-- but do not support key-existence lookups.  
CREATE INDEX friend2_idx ON friend2  
USING GIN (data);
```

JSONB Index Queries

```
SELECT data->>'first_name'  
FROM friend2  
WHERE data @> '{"last_name" : "Banks"}'  
ORDER BY 1;  
?column?
```

```
-----  
Bruce  
Fred
```

```
EXPLAIN SELECT data->>'first_name'  
FROM friend2  
WHERE data @> '{"last_name" : "Banks"}'  
ORDER BY 1;          QUERY PLAN
```

```
-----...  
Sort (cost=24.03..24.04 rows=1 width=139)  
  Sort Key: ((data ->> 'first_name'::text))  
    -> Bitmap Heap Scan on friend2 (cost=20.01..24.02 rows=1 ...  
      Recheck Cond: (data @> '{"last_name": "Banks"}'::jsonb)  
        -> Bitmap Index Scan on friend2_idx (cost=0.00..20.01 .....  
          Index Cond: (data @> '{"last_name": "Banks"}'::js...
```

JSONB Index Queries

```
SELECT data->>'last_name'  
FROM friend2  
WHERE data @> '{"first_name" : "Jane"}'  
ORDER BY 1;  
?column?
```

```
-----  
Tucker  
Williams
```

```
EXPLAIN SELECT data->>'last_name'  
FROM friend2  
WHERE data::jsonb @> '{"first_name" : "Jane"}'  
ORDER BY 1;          QUERY PLAN
```

```
-----...  
Sort (cost=24.03..24.04 rows=1 width=139)  
  Sort Key: ((data ->> 'last_name')::text)  
    -> Bitmap Heap Scan on friend2 (cost=20.01..24.02 rows=1 ...  
      Recheck Cond: (data @> '{"first_name": "Jane"}')::jsonb)  
        -> Bitmap Index Scan on friend2_idx (cost=0.00..20.01 .....  
          Index Cond: (data @> '{"first_name": "Jane"}')::js...
```

JSONB Index Queries

```
SELECT data->>'first_name' || ' ' || (data->>'last_name')
FROM friend2
WHERE data @> '{"ip_address" : "62.212.235.80"}'
ORDER BY 1;
      ?column?
```

Theresa Schmidt

```
EXPLAIN SELECT data->>'first_name' || ' ' || (data->>'last_name')
FROM friend2
WHERE data @> '{"ip_address" : "62.212.235.80"}'
ORDER BY 1;
                                QUERY PLAN
```

-----...
Sort (cost=24.04..24.05 rows=1 width=139)
Sort Key: (((data ->> 'first_name'::text) || ' '::text) || ...
-> Bitmap Heap Scan on friend2 (cost=20.01..24.03 rows=1 ...
Recheck Cond: (data @> '{"ip_address": "62.212.235.80"}' ...
-> **Bitmap Index Scan** on friend2_idx (cost=0.00..20.01 ...
Index Cond: (data @> '{"ip_address": "62.212.235...

7. Row Types

```
CREATE TYPE drivers_license AS  
(state CHAR(2), id INTEGER, valid_until DATE);
```

```
CREATE TABLE truck_driver  
(id SERIAL, name TEXT, license DRIVERS_LICENSE);
```

```
INSERT INTO truck_driver  
VALUES (DEFAULT, 'Jimbo Biggins', ('PA', 175319, '2017-03-12'));
```

Row Types

```
SELECT *
FROM truck_driver;
  id |      name      |      license
-----+-----+-----
   1 | Jimbo Biggins | (PA,175319,2017-03-12)
```

```
SELECT license
FROM truck_driver;
      license
-----
(PA,175319,2017-03-12)
```

-- parentheses are necessary

```
SELECT (license).state
FROM truck_driver;
state
-----
PA
```

8. Character Strings

```
$ cd /tmp  
$ wget http://web.mit.edu/freebsd/head/games/fortune/datfiles/fortunes  
$ psql postgres
```

```
CREATE TABLE fortune (line TEXT);
```

```
COPY fortune FROM '/tmp/fortunes' WITH (DELIMITER E'\x1F');
```

8.1 Case Folding and Prefix

```
SELECT * FROM fortune WHERE line = 'underdog';  
line  
-----
```

```
SELECT * FROM fortune WHERE line = 'Underdog';  
line  
-----  
Underdog
```

```
SELECT * FROM fortune WHERE lower(line) = 'underdog';  
line  
-----  
Underdog
```

Case Folding

```
CREATE INDEX fortune_idx_text ON fortune (line);
```

```
EXPLAIN SELECT * FROM fortune WHERE lower(line) = 'underdog';  
          QUERY PLAN
```

```
Seq Scan on fortune (cost=0.00..1384.63 rows=295 width=36)  
  Filter: (lower(line) = 'underdog'::text)
```

Indexed Case Folding

```
CREATE INDEX fortune_idx_lower ON fortune (lower(line));
```

```
EXPLAIN SELECT * FROM fortune WHERE lower(line) = 'underdog';  
QUERY PLAN
```

```
-----...  
Bitmap Heap Scan on fortune (cost=14.70..468.77 rows=295 ...  
  Recheck Cond: (lower(line) = 'underdog'::text)  
-> Bitmap Index Scan on fortune_idx_lower (cost=0.00....  
   Index Cond: (lower(line) = 'underdog'::text)
```

String Prefix

```
SELECT line
FROM fortune
WHERE line LIKE 'Mop%'
ORDER BY 1;
         line
```

Mophobia, n.:
Moping, melancholy mad:

String Prefix

```
EXPLAIN SELECT line
FROM fortune
WHERE line LIKE 'Mop%'
ORDER BY 1;
```

QUERY PLAN

```
Sort (cost=1237.07..1237.08 rows=4 width=36)
  Sort Key: line
  -> Seq Scan on fortune (cost=0.00..1237.03 rows=4 width=36)
       Filter: (line ~ 'Mop% '::text)
```


Indexed String Prefix

```
-- The default op class does string ordering of non-ASCII
-- collations, but not partial matching. text_pattern_ops
-- handles prefix matching, but not ordering.
CREATE INDEX fortune_idx_ops ON fortune (line text_pattern_ops);
```

```
EXPLAIN SELECT line
FROM fortune
WHERE line LIKE 'Mop%'
ORDER BY 1;
```

QUERY PLAN

```
-----...
Sort (cost=8.48..8.49 rows=4 width=36)
  Sort Key: line
  -> Index Only Scan using fortune_idx_ops on fortune (cost=0.41 ...
    Index Cond: ((line >= 'Mop'::text) AND (line < 'Moq'::...
    Filter: (line ~ 'Mop%'::text)
```

Case Folded String Prefix

```
EXPLAIN SELECT line
FROM fortune
WHERE lower(line) LIKE 'mop%'
ORDER BY 1;
```

QUERY PLAN

```
Sort (cost=1396.73..1397.47 rows=295 width=36)
  Sort Key: line
  -> Seq Scan on fortune (cost=0.00..1384.63 rows=295 width=36)
      Filter: (lower(line) ~ 'mop% '::text)
```

Indexed Case Folded String Prefix

```
CREATE INDEX fortune_idx_ops_lower ON fortune  
(lower(line) text_pattern_ops);
```

```
EXPLAIN SELECT line  
FROM fortune  
WHERE lower(line) LIKE 'mop%'  
ORDER BY 1;
```

QUERY PLAN

```
-----  
Sort (cost=481.61..482.35 rows=295 width=36)  
  Sort Key: line  
    -> Bitmap Heap Scan on fortune (cost=15.44..469.51 rows=295 ...  
        Filter: (lower(line) ~ 'mop% '::text)  
          -> Bitmap Index Scan on fortune_idx_ops_lower (cost=0...  
              Index Cond: ((lower(line) ~>= 'mop' '::text) AND (...
```

8.2. Full Text Search

- ▶ Allows whole-word or word prefix searches
- ▶ Supports *and*, *or*, *not*
- ▶ Converts words to lexemes
 - ▶ stemming
 - ▶ 21 languages supported
 - ▶ 'Simple' search config bypasses stemming
- ▶ Removes stop words
- ▶ Supports synonyms and phrase transformations (thesaurus)

Tsvector and Tsquery

```
SHOW default_text_search_config;  
default_text_search_config
```

```
-----  
pg_catalog.english
```

```
SELECT to_tsvector('I can hardly wait.');
```

```
to_tsvector
```

```
-----  
'hard':3 'wait':4
```

```
SELECT to_tsquery('hardly & wait');
```

```
to_tsquery
```

```
-----  
'hard' & 'wait'
```

Tsvector and Tsquery

```
SELECT to_tsvector('I can hardly wait.') @@  
       to_tsquery('hardly & wait');  
?column?  
-----  
t
```

```
SELECT to_tsvector('I can hardly wait.') @@  
       to_tsquery('softly & wait');  
?column?  
-----  
f
```

Indexing Full Text Search

```
CREATE INDEX fortune_idx_ts ON fortune  
USING GIN (to_tsvector('english', line));
```

Full Text Search Queries

```
SELECT line
FROM fortune
WHERE to_tsvector('english', line) @@ to_tsquery('pandas');
```

A giant **panda** bear is really a member of the raccoon family.

```
EXPLAIN SELECT line
FROM fortune
WHERE to_tsvector('english', line) @@ to_tsquery('pandas');
```

QUERY PLAN

```
Bitmap Heap Scan on fortune (cost=12.41..94.25 rows=21 width=36)
  Recheck Cond: (to_tsvector('english':regconfig, line) @@ to_ts...
-> Bitmap Index Scan on fortune_idx_ts (cost=0.00..12.40 rows...
   Index Cond: (to_tsvector('english':regconfig, line) @@ t...
```


Complex Full Text Search Queries

```
SELECT line
FROM fortune
WHERE to_tsvector('english', line) @@ to_tsquery('cat & sleep');
      line
```

People who take **cat** naps don't usually **sleep** in a **cat**'s cradle.

```
SELECT line
FROM fortune
WHERE to_tsvector('english', line) @@ to_tsquery('cat & (sleep | nap)');
      line
```

People who take **cat naps** don't usually **sleep** in a **cat**'s cradle.

Q: What is the sound of one **cat napping**?

Word Prefix Search

```
SELECT line
FROM fortune
WHERE to_tsvector('english', line) @@
      to_tsquery('english', 'zip:*')
ORDER BY 1;
```

line

Bozo is the Brotherhood of Zips and Others. Bozos are people who band ... -- he's the one who's in trouble. One round from an Uzi can zip far I've got two Bics, four Zippos and eighteen books of matches." Postmen never die, they just lose their zip.

Word Prefix Search

```
EXPLAIN SELECT line
FROM fortune
WHERE to_tsvector('english', line) @@
      to_tsquery('english', 'zip:*')
ORDER BY 1;
```

QUERY PLAN

```
-----
Sort (cost=101.21..101.26 rows=21 width=36)
  Sort Key: line
  -> Bitmap Heap Scan on fortune (cost=24.16..100.75 rows=21 ...
        Recheck Cond: (to_tsvector('english'::regconfig, line) ...
        -> Bitmap Index Scan on fortune_idx_ts (cost=0.00..24 ...
              Index Cond: (to_tsvector('english'::regconfig, li...
```

8.3. Adjacent Letter Search

```
-- ILIKE is case-insensitive LIKE
SELECT line
FROM fortune
WHERE line ILIKE '%verit%'
ORDER BY 1;
```

line

body. There hangs from his belt a veritable arsenal of deadly weapons:
In wine there is truth (In vino veritas).

Passes wind, water, or out depending upon the severity of the

Adjacent Letter Search

```
EXPLAIN SELECT line
FROM fortune
WHERE line ILIKE '%verit%'
ORDER BY 1;
```

QUERY PLAN

```
Sort (cost=1237.07..1237.08 rows=4 width=36)
  Sort Key: line
  -> Seq Scan on fortune (cost=0.00..1237.03 rows=4 width=36)
       Filter: (line ~* '%verit% '::text)
```

Indexed Adjacent Letters

```
CREATE EXTENSION pg_trgm;
```

```
CREATE INDEX fortune_idx_trgm ON fortune  
USING GIN (line gin_trgm_ops);
```

Indexed Adjacent Letters

```
SELECT line
FROM fortune
WHERE line ILIKE '%verit%'
ORDER BY 1;
```

line

body. There hangs from his belt a veritable arsenal of deadly weapons:
In wine there is truth (In vino veritas).

Passes wind, water, or out depending upon the severity of the

Indexed Adjacent Letters

```
EXPLAIN SELECT line
FROM fortune
WHERE line ILIKE '%verit%'
ORDER BY 1;
```

QUERY PLAN

```
-----...
Sort  (cost=43.05..43.06 rows=4 width=36)
  Sort Key: line
    -> Bitmap Heap Scan on fortune  (cost=28.03..43.01 rows=4 ...
      Recheck Cond: (line ~* '%verit% '::text)
        -> Bitmap Index Scan on fortune_idx_trgm  (cost=0.00....
          Index Cond: (line ~* '%verit% '::text)
```


Word Prefix Search

```
-- ~* is case-insensitive regular expression
SELECT line
FROM fortune
WHERE line ~* '([^a-z])zip'
ORDER BY 1;
```

line

Bozo is the Brotherhood of Zips and Others. Bozos are people who band
... -- he's the one who's in trouble. One round from an Uzi can zip
far I've got two Bics, four Zippos and eighteen books of matches."
Postmen never die, they just lose their zip.

Word Prefix Search

```
EXPLAIN SELECT line
FROM fortune
WHERE line ~* '(^|[^a-z])zip'
ORDER BY 1;
```

QUERY PLAN

```
-----...
Sort (cost=27.05..27.06 rows=4 width=36)
  Sort Key: line
  -> Bitmap Heap Scan on fortune (cost=12.03..27.01 rows=4 ...
    Recheck Cond: (line ~* '(^|[^a-z])zip'::text)
    -> Bitmap Index Scan on fortune_idx_trgm (cost=0.00....
      Index Cond: (line ~* '(^|[^a-z])zip'::text)
```

Similarity

```
SELECT show_limit();  
show_limit  
-----  
0.3
```

```
SELECT line, similarity(line, 'So much for the plan')  
FROM fortune  
WHERE line % 'So much for the plan'  
ORDER BY 1;
```

line	similarity
Oh, it's so much fun,	0.325
So much	0.380952
There's so much plastic in this culture that	0.304348

Similarity

```
EXPLAIN SELECT line, similarity(line, 'So much for the plan')
FROM fortune
WHERE line % 'So much for the plan'
ORDER BY 1;
```

QUERY PLAN

```
-----...
Sort (cost=342.80..342.95 rows=59 width=36)
  Sort Key: line
    -> Bitmap Heap Scan on fortune (cost=172.46..341.06 rows=59 ...
      Recheck Cond: (line % 'So much for the plan'::text)
        -> Bitmap Index Scan on fortune_idx_trgm (cost=0.00....
          Index Cond: (line % 'So much for the plan'::text)
```

Soundex, metaphone, and levenshtein word similarity comparisons are also available.

Indexes Created in this Section

```
\dt+ fortune
```

List of relations

Schema	Name	Type	Owner	Size	Description
public	fortune	table	postgres	4024 kB	

```
\d fortune and \di+
```

fortune_idx_text	btree	line		3480 kB
fortune_idx_lower	btree	lower(line)		3480 kB
fortune_idx_ops	btree	line text_pattern_ops		3480 kB
fortune_idx_ops_lower	btree	lower(line) text_pattern_ops		3480 kB
fortune_idx_ts	gin	to_tsvector(...)		2056 kB
fortune_idx_trgm	gin	line gin_trgm_ops		4856 kB

Use of the Contains Operator @> in this Presentation

\do @>

List of operators

Schema	Name	Left arg type	Right arg type	Result type	Description
pg_catalog	@>	aclitem[]	aclitem	boolean	contains
pg_catalog	@>	anyarray	anyarray	boolean	contains
pg_catalog	@>	anyrange	anyelement	boolean	contains
pg_catalog	@>	anyrange	anyrange	boolean	contains
pg_catalog	@>	box	box	boolean	contains
pg_catalog	@>	box	point	boolean	contains
pg_catalog	@>	circle	circle	boolean	contains
pg_catalog	@>	circle	point	boolean	contains
pg_catalog	@>	jsonb	jsonb	boolean	contains
pg_catalog	@>	path	point	boolean	contains
pg_catalog	@>	polygon	point	boolean	contains
pg_catalog	@>	polygon	polygon	boolean	contains
pg_catalog	@>	tsquery	tsquery	boolean	contains

Conclusion



<http://momjian.us/presentations>

<https://www.flickr.com/photos/patgaines/>