Mining Text Data

DATA 351: Data Management with SQL

Lucas P. Cordova, Ph.D.

2026-04-06

This lecture covers string functions, POSIX regular expressions in PostgreSQL, extracting structured fields from messy narrative text, and full-text search with tsvector, tsquery, GIN indexes, ts_headline, and ts_rank. We use crime report narratives and State of the Union speeches. Based on Chapter 14 of Practical SQL, 2nd Edition.

Table of contents

1 What You'll Learn

1.1 Learning Objectives

By the end of this session, you will be able to:

1. Apply common string functions for trimming, case conversion, measuring length, and slicing substrings
2. Read basic POSIX regex notation (`\d`, `{n,m}`, groups, alternation) and use `~`, `~*`, `!~`, and `substring(... from ...)`
3. Use `regexp_match`, `regexp_matches`, `regexp_replace`, and `regexp_split_to_table` to transform text
4. Extract structured columns from semi-structured blobs using capture groups and array indexing
5. Build full-text search queries with `to_tsvector`, `to_tsquery`, the `@@` match operator, and GIN indexes
6. Present and rank search hits with `ts_headline` and `ts_rank` (including length normalization)

. . .

Text is not a luxury column type. It is where half your "CSV from a government website" problems live. Today we give SQL something sharper than hope.

2 Part 1: Environment

2.1 Create the Database and Load Data

```
createdb mining_text;
```

2.2 Create the Database and Load Data · Steps

Create a dedicated database for this chapter using whatever method you prefer or the command above in a terminal.

Download and run the setup script **mining_text.sql ↓** while connected to `mining_text`. The file embeds all rows with `INSERT` statements (dollar-quoted text), so you can run it from pgAdmin or Beekeeper without configuring CSV paths.

Optional: **mining_text_csvs.zip ↓** still has the raw extracts if you want to practice `\COPY` separately.

2.3 Tables You Will See Today

Table	Source	Role
`county_regex_demo`	bundled in `mining_text.sql`	Regex filtering practice (`~`, `~*`)
`crime_reports`	embedded in `mining_text.sql`	Parse semi-structured police narratives
`president_speeches`	embedded in `mining_text.sql`	Full-text search on State of the Union addresses

2.4 Sanity Checks

```
SELECT count(*) FROM crime_reports;
SELECT count(*) FROM president_speeches;
SELECT count(*) FROM county_regex_demo;
```

2.5 Sanity Checks · Output

Run these after the setup script finishes to verify the counts of the tables.

```
SELECT count(*) FROM crime_reports;
-- 5
SELECT count(*) FROM president_speeches;
-- 79
SELECT count(*) FROM county_regex_demo;
-- 8
```

3 Part 2: String Functions

3.1 Why Strings Still Matter

JSON gets the hype. Arrays get the conference talks. Strings pay the rent: addresses, notes, log lines, pasted emails, and whatever your stakeholder calls "structured" because it has a colon somewhere.

PostgreSQL's string toolkit is documented in the manual under String Functions. We start with the ones you will actually type.

3.2 Casing and Cleanup

```
SELECT upper('Neal7');
SELECT lower('Randy');
SELECT initcap('at the end of the day');
SELECT initcap('Practical SQL');  -- acronyms: not perfect
```

3.3 Casing and Cleanup · Output

Run these in order and watch how forgiving real-world text is not.

```
SELECT upper('Neal7');
-- NEAL7
SELECT lower('Randy');
-- randy
SELECT initcap('at the end of the day');
-- At The End Of The Day
SELECT initcap('Practical SQL');  -- acronyms: not perfect
-- Practical Sql
```

`initcap` capitalizes word starts. It does not read minds. Your acronym table lives in application code, not here.

3.4 Length, Position, Trim

```
SELECT char_length(' Pat ');
SELECT length(' Pat ');          -- same for varchar; byte length differs on some types
SELECT position(', ' in 'Tan, Bella');
SELECT trim('s' from 'socks');
SELECT trim(trailing 's' from 'socks');
SELECT trim(' Pat ');
SELECT char_length(trim(' Pat '));
SELECT ltrim('socks', 's');
SELECT rtrim('socks', 's');
```

3.5 Length, Position, Trim · Output

Run these in order.

```sql
SELECT char_length(' Pat ');
-- 5
SELECT length(' Pat ');          -- same for varchar; byte length differs on some types
-- 5
SELECT position(', ' in 'Tan, Bella');
-- 4
SELECT trim('s' from 'socks');
-- ock
SELECT trim(trailing 's' from 'socks');
-- sock
SELECT trim(' Pat ');
-- Pat
SELECT char_length(trim(' Pat '));
-- 3
SELECT ltrim('socks', 's');
-- ocks
SELECT rtrim('socks', 's');
-- sock
```

If you are counting characters for validation rules, `char_length` after `trim` is the boring, correct path.

3.6 Slicing and Replacing

```sql
SELECT left('703-555-1212', 3);
SELECT right('703-555-1212', 8);
SELECT replace('bat', 'b', 'c');
SELECT replace('aaa', 'aa', 'b');
```

3.7 Slicing and Replacing · Output

Run these (including the `replace('aaa', ...)` check).

```sql
SELECT left('703-555-1212', 3);
-- 703
SELECT right('703-555-1212', 8);
-- 555-1212
SELECT replace('bat', 'b', 'c');
-- cat
SELECT replace('aaa', 'aa', 'b');
-- ba
```

Quick check: `replace('aaa', 'aa', 'b')` returns `ba` (left-to-right, non-overlapping substitution). String replacement is greedy in a way that will eventually humble everyone.

4 Part 3: Regular Expressions

4.1 What is a regular expression?

A **regular expression** (regex) is a small pattern language for describing text shape. You hand PostgreSQL a pattern string; the engine can test a match, return the matched substring, split on delimiters, or pull out captured pieces.

PostgreSQL uses **POSIX** regular expressions (documented under Pattern Matching). In SQL the pattern usually lives in a single-quoted string. Backslashes are easy to get wrong: the string parser and the regex engine both care about `\`, so you often write `'\\d'` (two characters in source, one backslash in the pattern) or use dollar-quoting when patterns get busy.

4.2 Notation

Piece	Meaning
`.`	one character (with caveats: not always newline)
`\d`	a digit `0`–`9`
`\w`	a word character (letters, digits, underscore in the POSIX class)
`[0-9]`	any single digit from the list
`{n}`	repeat the previous "atom" exactly `n` times
`{n,m}`	repeat at least `n` and at most `m` times, so `{1,2}` means one or two
`+`	one or more; `*` zero or more; `?` optional (zero or one)
`^ $`	start and end of the string (for these lectures, think whole-field matching unless noted)
`|`	alternation: `A|B` is A or B
`()`	grouping; also **capture** for `regexp_match`
`(?: … )`	group **without** capturing (still useful for precedence)

4.3 Example

`\d{1,2}/\d{1,2}/\d{2}` describes a date fragment like `4/16/17`: one or two digits, slash, one or two digits, slash, two digits. Slashes are literal here; we sometimes escape them as `\/` in patterns for clarity or habit from other tools.

4.4 Operators You Must Know

Operator	Meaning
`~`	matches, case sensitive
`~*`	matches, case insensitive
`!~`	does not match, case sensitive
`!~*`	does not match, case insensitive

In `WHERE` clauses, regex turns a pile of `LIKE` patterns into one expressive filter. It also turns a simple bug into a performance mystery. Use indexes thoughtfully (later: GIN for FTS; for regex alone, sometimes a functional index, sometimes a redesign).

4.5 Substring with a Pattern

```
SELECT substring('The game starts at 7 p.m. on May 2, 2024.' from '.+');
SELECT substring('The game starts at 7 p.m. on May 2, 2024.' from '\d{1,2} (?:a.m.|p.m.)');
SELECT substring('The game starts at 7 p.m. on May 2, 2024.' from '^\w+');
SELECT substring('The game starts at 7 p.m. on May 2, 2024.' from '\w+.$');
SELECT substring('The game starts at 7 p.m. on May 2, 2024.' from 'May|June');
SELECT substring('The game starts at 7 p.m. on May 2, 2024.' from '\d{4}');
SELECT substring('The game starts at 7 p.m. on May 2, 2024.' from 'May \d, \d{4}');
```

4.6 Substring with a Pattern · Output

`substring(string from pattern)` returns the first match. **Run them** one at a time.

```
SELECT substring('The game starts at 7 p.m. on May 2, 2024.' from '.+');
-- The game starts at 7 p.m. on May 2, 2024.
SELECT substring('The game starts at 7 p.m. on May 2, 2024.' from '\d{1,2} (?:a.m.|p.m.)');
-- 7 p.m.
SELECT substring('The game starts at 7 p.m. on May 2, 2024.' from '^\w+');
-- The
SELECT substring('The game starts at 7 p.m. on May 2, 2024.' from '\w+.$');
```

```
-- 2024.
SELECT substring('The game starts at 7 p.m. on May 2, 2024.' from 'May|June');
-- May
SELECT substring('The game starts at 7 p.m. on May 2, 2024.' from '\d{4}');
-- 2024
SELECT substring('The game starts at 7 p.m. on May 2, 2024.' from 'May \d, \d{4}');
-- May 2, 2024
```

Notice how greedy `.+` is on the first pattern. Regex is a contract between you and the parser. Read the contract.

4.7 Filtering Rows

```
SELECT county_name
FROM county_regex_demo
WHERE county_name ~* '(lade|lare)'
ORDER BY county_name;

SELECT county_name
FROM county_regex_demo
WHERE county_name ~* 'ash' AND county_name !~ 'Wash'
ORDER BY county_name;
```

4.8 Filtering Rows · Output

We use `county_regex_demo` instead of the full Census extract:

```
SELECT county_name
FROM county_regex_demo
WHERE county_name ~* '(lade|lare)'
ORDER BY county_name;
-- Clare County
SELECT county_name
FROM county_regex_demo
WHERE county_name ~* 'ash' AND county_name !~ 'Wash'
ORDER BY county_name;
-- Ashland County
-- Nash County
```

Run both queries. Advance once per query, then per result line.

The second query is the classic "I want Ashland but not Washington" maneuver. If your data contains `NULL` county names, remember that `WHERE col ~ 'pattern'` does not match `NULL`. Neither does your excuse.

4.9 Replacing and Splitting

```
SELECT regexp_replace('05/12/2024', '\d{4}', '2023');
SELECT regexp_split_to_table('Four,score,and,seven,years,ago', ',');
SELECT regexp_split_to_array('Phil Mike Tony Steve', ' ');
SELECT array_length(regexp_split_to_array('Phil Mike Tony Steve', ' '), 1);
```

4.10 Replacing and Splitting · Output

Run these.

```
SELECT regexp_replace('05/12/2024', '\d{4}', '2023');
-- 05/12/2023
SELECT regexp_split_to_table('Four,score,and,seven,years,ago', ',');
-- Four
-- score
-- and
-- seven
-- years
-- ago
SELECT regexp_split_to_array('Phil Mike Tony Steve', ' ');
-- {Phil,Mike,Tony,Steve}
SELECT array_length(regexp_split_to_array('Phil Mike Tony Steve', ' '), 1);
-- 4
```

`regexp_split_to_table` is a set-returning function: it produces one row per piece. That makes it handy for normalizing multi-valued cells someone swore were "atomic."

5 Part 4: From Blob to Columns

5.1 The Crime Narratives

```
SELECT crime_id, original_text
FROM crime_reports
ORDER BY crime_id
LIMIT 1;
```

5.2 The Crime Narratives · Output

Our `crime_reports` table loads only `original_text`. Everything else (dates, address, offense, case number) is hiding inside the blob.

Run this to open one row:

```
SELECT original_text
FROM crime_reports
ORDER BY crime_id
LIMIT 1;
-- 4/16/17-4/17/17
-- 2100-0900 hrs.
-- 46000 Block Ashmere Sq.
-- Sterling
-- Larceny: The victim reported that a
-- bicycle was stolen from their opened
-- garage door during the overnight hours.
-- C0170006614
```

Beautiful. Readable. Terrible for `GROUP BY`. We will fix that with patterns, not with interns.

5.3 First Date Match

Run this.

```
SELECT crime_id,
       regexp_match(original_text, '\d{1,2}\/\d{1,2}\/\d{2}')
FROM crime_reports
ORDER BY crime_id;
```

5.4 First Date Match · Output

`regexp_match` returns the first match as a text array (or `NULL`).

```
SELECT crime_id,
       regexp_match(original_text, '\d{1,2}\/\d{1,2}\/\d{2}')
FROM crime_reports
ORDER BY crime_id;
-- 1 | {4/16/17}
-- 2 | {4/8/17}
-- 3 | {4/4/17}
```

```
-- 4 | {04/10/17}
-- 5 | {04/09/17}
```

5.5 All Date Matches

Row 1 has two dates in the narrative.

```
4/16/17-4/17/17
2100-0900 hrs.
46000 Block Ashmere Sq.
Sterling
```

. . .

Run this and compare to the previous slide.

```
SELECT crime_id,
       regexp_matches(original_text, '\d{1,2}\/\d{1,2}\/\d{2}', 'g')
FROM crime_reports
ORDER BY crime_id;
```

5.6 All Date Matches · Output

`regexp_matches` with the `g` flag finds every non-overlapping match.

```
SELECT crime_id,
       regexp_matches(original_text, '\d{1,2}\/\d{1,2}\/\d{2}', 'g')
FROM crime_reports
ORDER BY crime_id;
-- 1 | {4/16/17}
-- 1 | {4/17/17}
-- 2 | {4/8/17}
-- 3 | {4/4/17}
-- 4 | {04/10/17}
-- 5 | {04/09/17}
```

Compare the two outputs for rows that mention more than one calendar date.

5.7 Capture Groups Clean Up Noise

Run these:

```
SELECT crime_id,
       regexp_match(original_text, '-\d{1,2}\/\d{1,2}\/\d{2}')
FROM crime_reports
ORDER BY crime_id;
SELECT crime_id,
       regexp_match(original_text, '-(\d{1,2}\/\d{1,2}\/\d{2})')
FROM crime_reports
ORDER BY crime_id;
```

5.8 Capture Groups · Output

Parentheses in the pattern define capture groups. `regexp_match` still returns the whole match as element `[1]` when you wrap only part of the pattern. **Run both.**

```
SELECT crime_id,
       regexp_match(original_text, '-\d{1,2}\/\d{1,2}\/\d{2}')
FROM crime_reports
ORDER BY crime_id;
-- 1 | {-4/17/17}
-- 2 |
-- 3 |
-- 4 |
-- 5 |
SELECT crime_id,
       regexp_match(original_text, '-(\d{1,2}\/\d{1,2}\/\d{2})')
FROM crime_reports
ORDER BY crime_id;
-- 1 | {4/17/17}
-- 2 |
-- 3 |
-- 4 |
-- 5 |
```

The second version drops the leading hyphen by capturing only the date digits.

5.9 Pull Several Fields at Once

Run this:

```sql
SELECT
    crime_id,
    regexp_match(original_text, '(?:C0|SO)[0-9]+') AS case_number,
    regexp_match(original_text, '\d{1,2}\/\d{1,2}\/\d{2}') AS date_1,
    regexp_match(original_text, '\n(?:\w+ \w+|\w+)\n(.*):') AS crime_type,
    regexp_match(original_text, '(?:Sq.|Plz.|Dr.|Ter.|Rd.)\n(\w+ \w+|\w+)\n')
        AS city
FROM crime_reports
ORDER BY crime_id;
```

5.10 Pull Several Fields · Output

This is not pretty. It is honest.

```sql
SELECT
    crime_id,
    regexp_match(original_text, '(?:C0|SO)[0-9]+') AS case_number,
    regexp_match(original_text, '\d{1,2}\/\d{1,2}\/\d{2}') AS date_1,
    regexp_match(original_text, '\n(?:\w+ \w+|\w+)\n(.*):') AS crime_type,
    regexp_match(original_text, '(?:Sq.|Plz.|Dr.|Ter.|Rd.)\n(\w+ \w+|\w+)\n')
        AS city
FROM crime_reports
ORDER BY crime_id;
-- 1 | {C0170006614} | {4/16/17} | {Larceny} | {Sterling}
-- 2 | {C0170006162} | {4/8/17} | {Destruction of Property} | {Sterling}
-- 3 | {C0170006079} | {4/4/17} | {Larceny} | {Sterling}
-- 4 | {SO170006250} | {04/10/17} | {Larceny} | {Middleburg}
-- 5 | {SO170006211} | {04/09/17} | {Destruction of Property} | {Sterling}
```

Patterns like this are why we version-control SQL. The day the sheriff's office changes a label from `hrs.` to `hours`, you will discover the true meaning of "breaking change."

5.11 Arrays in PostgreSQL: `[1]`

Run this:

```sql
SELECT
    crime_id,
    (regexp_match(original_text, '(?:C0|SO)[0-9]+'))[1] AS case_number
FROM crime_reports
ORDER BY crime_id;
```

5.12 Arrays in PostgreSQL · Output

`regexp_match` returns `text[]`. Grab the first element explicitly.

```
SELECT
    crime_id,
    (regexp_match(original_text, '(?:CO|SO)[0-9]+'))[1] AS case_number
FROM crime_reports
ORDER BY crime_id;
-- 1 | CO170006614
-- 2 | CO170006162
-- 3 | CO170006079
-- 4 | SO170006250
-- 5 | SO170006211
```

If the match fails, `regexp_match` is `NULL` and `[1]` is `NULL`. Plan for that when you cast to timestamps.

5.13 Optional: Populate Columns

```
-- Pattern sketch (abbreviated): build a string, cast to timestamptz
-- ((regexp_match(...))[1] || ' ' || (regexp_match(...))[1])::timestamptz
```

5.14 Optional: Populate Columns · Notes

The textbook walks through `UPDATE` statements that cast extracted dates to `timestamptz` and fill `street`, `city`, `crime_type`, and `description`. The expressions are long but repetitive: once you trust one `regexp_match`, the rest are the same idea with different patterns.

Question for you: Why might `EXISTS (SELECT regexp_matches(...))` appear in a `CASE` branch?

5.15 Answer

Because sometimes you need to know whether a second date exists before you try to parse it.

5.16 Walkthrough: Build `date_1` Once

Don't run this (unless you are using a scratch database or copy of the crime_reports table):

```
CREATE temp TABLE crime_reports_copy AS SELECT * FROM crime_reports;

UPDATE crime_reports
SET date_1 =
(
    (regexp_match(original_text, '\d{1,2}\/\d{1,2}\/\d{2}'))[1]
        || ' ' ||
    (regexp_match(original_text, '\/\d{2}\n(\d{4})'))[1]
        || ' US/Eastern'
)::timestamptz
RETURNING crime_id, date_1, original_text;
```

5.17 Walkthrough: Build `date_1` · Output

The book example concatenates the first date in `MM/DD/YY` form with the first time field (four digits) and a time zone. **Run once** (it updates every row). Use a scratch database or `ROLLBACK` if you are experimenting.

```
UPDATE crime_reports_copy
SET date_1 =
(
    (regexp_match(original_text, '\d{1,2}\/\d{1,2}\/\d{2}'))[1]
        || ' ' ||
    (regexp_match(original_text, '\/\d{2}\n(\d{4})'))[1]
        || ' US/Eastern'
)::timestamptz
RETURNING crime_id, date_1, original_text;
-- 1 | 2017-04-17 01:00:00+00 | ...
-- 2 | 2017-04-08 20:00:00+00 | ...
-- 3 | 2017-04-04 18:00:00+00 | ...
-- 4 | 2017-04-10 20:05:00+00 | ...
-- 5 | 2017-04-09 16:00:00+00 | ...
```

5.18 Why `RETURNING`?

`RETURNING` shows you what you broke before you commit. This is not Tinder. Do not swipe right on untested regex.

If a row lacks one of the pieces, the corresponding `regexp_match` is `NULL` and the concatenation collapses in ways PostgreSQL will explain loudly. That is good. Silent failure is how you become a historian instead of a data engineer.

5.19 View the Cleaned Rows

Run this:

```
SELECT date_1,
       street,
       city,
       crime_type
FROM crime_reports_copy
ORDER BY crime_id;
```

5.20 View the Cleaned Rows · Output

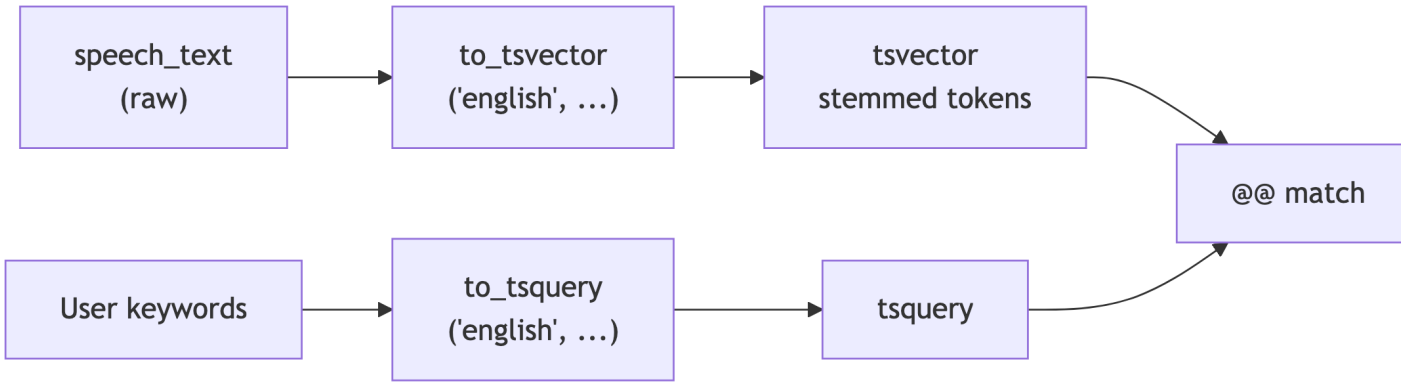
After a full `UPDATE` of all parsed columns (as in the book), a simple report query becomes possible. The `UPDATE` sets `date_1` only (other parsed columns stay `NULL` until you run the full `UPDATE` block that sets all parsed columns).

```
SELECT date_1,
       street,
       city,
       crime_type
FROM crime_reports_copy
ORDER BY crime_id;
-- 2017-04-17 01:00:00+00 |  |  |
-- 2017-04-08 20:00:00+00 |  |  |
-- 2017-04-04 18:00:00+00 |  |  |
-- 2017-04-10 20:05:00+00 |  |  |
-- 2017-04-09 16:00:00+00 |  |  |
```

Compare mentally to the blob you started with. That difference is the whole lecture in one `SELECT`.

6 Part 5: Full-Text Search

6.1 Pipeline: From Speech to Search



Raw text enters, tokens come out, queries meet documents at `@@`. The GIN index sits on the `tsvector` column so the middle of this diagram does not require a sequential read of every presidency.

6.2 `pg_ts_config`

Run this:

```
SELECT cfgname FROM pg_ts_config;
```

6.3 `pg_ts_config` · Output

Full-text search is not `LIKE '%tax%'`. It tokenizes, stems, drops noise words, and gives you operators that behave well at scale.

Check which language configurations exist.

```
SELECT cfgname FROM pg_ts_config;
-- arabic
-- armenian
-- basque
-- catalan
-- danish
-- dutch
-- english
```

```
-- finnish
-- ... (more rows on your server)
```

6.4 `to_tsvector` and `@@`

```
SELECT to_tsvector('english', 'I am walking across the sitting room to sit with you.');

SELECT to_tsquery('english', 'walking & sitting');

SELECT to_tsvector('english', 'I am walking across the sitting room')
       @@ to_tsquery('english', 'walking & sitting');

SELECT to_tsvector('english', 'I am walking across the sitting room')
       @@ to_tsquery('english', 'walking & running');
```

6.5 `to_tsvector` and `@@` · Output

Convert document and query:

Run these.

```
SELECT to_tsvector('english', 'I am walking across the sitting room to sit with you.');
-- 'across':4 'room':7 'sit':6,9 'walk':3

SELECT to_tsquery('english', 'walking & sitting');
-- 'walk' & 'sit'

SELECT to_tsvector('english', 'I am walking across the sitting room')
       @@ to_tsquery('english', 'walking & sitting');
-- t

SELECT to_tsvector('english', 'I am walking across the sitting room')
       @@ to_tsquery('english', 'walking & running');
-- f
```

`@@` answers: does this document match this query?

6.6 Operators Inside `to_tsquery`

- `&` AND
- `|` OR
- `!` NOT

- `<->` adjacent terms (phrase)
- `<N>` distance (terms within N positions)

We will lean on the State of the Union table. Our setup script already adds `search_speech_text` and a GIN index.

6.7 Index for Search

```
-- Already in mining_text.sql after INSERT + UPDATE:
-- CREATE INDEX search_idx ON president_speeches USING gin (search_speech_text);
```

6.8 Index for Search · Notes

GIN is the usual choice for `tsvector` columns. Without it, you are scanning speeches the way I scroll my inbox: slowly and with regret.

6.9 Find Speeches

Run this:

```
SELECT president, speech_date
FROM president_speeches
WHERE search_speech_text @@ to_tsquery('english', 'Vietnam')
ORDER BY speech_date;
```

6.10 Find Speeches · Output

```
SELECT president, speech_date
FROM president_speeches
WHERE search_speech_text @@ to_tsquery('english', 'Vietnam')
ORDER BY speech_date;
-- John F. Kennedy | 1961-05-25
-- Lyndon B. Johnson | 1966-01-12
-- Lyndon B. Johnson | 1967-01-10
-- ... (16 more rows; 19 total in this bundle)
```

6.11 Snippets with ts_headline

Run this:

```
SELECT president,
       speech_date,
       ts_headline(
           speech_text,
           to_tsquery('english', 'tax'),
           'StartSel = <,
            StopSel = >,
            MinWords=5,
            MaxWords=7,
            MaxFragments=1'
       )
FROM president_speeches
WHERE search_speech_text @@ to_tsquery('english', 'tax')
ORDER BY speech_date;
```

6.12 Snippets with ts_headline · Output

```
SELECT president,
       speech_date,
       ts_headline(
           speech_text,
           to_tsquery('english', 'tax'),
           'StartSel = <,
            StopSel = >,
            MinWords=5,
            MaxWords=7,
            MaxFragments=1'
       )
FROM president_speeches
WHERE search_speech_text @@ to_tsquery('english', 'tax')
ORDER BY speech_date;
-- Harry S. Truman | 1946-01-21 | ... <tax> POLICY ...
-- Harry S. Truman | 1947-01-06 | excise <tax> rates which, under the present
-- Harry S. Truman | 1948-01-07 | point in our <tax> structure. ...
-- ... (many more rows mention tax)
```

That angle-bracket markup is for HTML consumers. In slides, it still reads clearly as “here is the hit in context.”

6.13 AND / NOT and Phrase Proximity

Run each query:

```
SELECT president,
       speech_date,
       ts_headline(
           speech_text,
           to_tsquery('english', 'transportation & !roads'),
           'StartSel = <, StopSel = >, MinWords=5, MaxWords=7, MaxFragments=1'
       )
FROM president_speeches
WHERE search_speech_text @@ to_tsquery('english', 'transportation & !roads')
ORDER BY speech_date;
SELECT president,
       speech_date,
       ts_headline(
           speech_text,
           to_tsquery('english', 'military <-> defense'),
           'StartSel = <, StopSel = >, MinWords=5, MaxWords=7, MaxFragments=1'
       )
FROM president_speeches
WHERE search_speech_text @@ to_tsquery('english', 'military <-> defense')
ORDER BY speech_date;
```

6.14 AND / NOT and Phrase Proximity · Output

```
SELECT president,
       speech_date,
       ts_headline(
           speech_text,
           to_tsquery('english', 'transportation & !roads'),
           'StartSel = <, StopSel = >, MinWords=5, MaxWords=7, MaxFragments=1'
       )
FROM president_speeches
WHERE search_speech_text @@ to_tsquery('english', 'transportation & !roads')
ORDER BY speech_date;
-- Harry S. Truman | 1947-01-06 | Mr. President, Mr. Speaker, Members
-- Harry S. Truman | 1949-01-05 | Mr. President, Mr. Speaker, Members
SELECT president,
       speech_date,
       ts_headline(
```

```
            speech_text,
            to_tsquery('english', 'military <-> defense'),
            'StartSel = <, StopSel = >, MinWords=5, MaxWords=7, MaxFragments=1'
        )
FROM president_speeches
WHERE search_speech_text @@ to_tsquery('english', 'military <-> defense')
ORDER BY speech_date;
-- Dwight D. Eisenhower | 1956-01-05 | To the Congress of the
-- Dwight D. Eisenhower | 1958-01-09 | Mr. President, Mr. Speaker, Members
-- Dwight D. Eisenhower | 1959-01-09 | Mr. President, Mr. Speaker, Members
```

Try `<2>` instead of `<->` in a copy of the query to allow a small gap between words. Language is messy. PostgreSQL knows.

6.15 Ranking

Run both and compare ordering.

```
SELECT president,
       speech_date,
       ts_rank(
           search_speech_text,
           to_tsquery('english', 'war & security & threat & enemy')
       ) AS score
FROM president_speeches
WHERE search_speech_text @@
      to_tsquery('english', 'war & security & threat & enemy')
ORDER BY score DESC
LIMIT 5;
SELECT president,
       speech_date,
       ts_rank(
           search_speech_text,
           to_tsquery('english', 'war & security & threat & enemy'),
           2
       )::numeric AS score
FROM president_speeches
WHERE search_speech_text @@
      to_tsquery('english', 'war & security & threat & enemy')
ORDER BY score DESC
LIMIT 5;
```

6.16 Ranking · Output

```
SELECT president,
       speech_date,
       ts_rank(
           search_speech_text,
           to_tsquery('english', 'war & security & threat & enemy')
       ) AS score
FROM president_speeches
WHERE search_speech_text @@
      to_tsquery('english', 'war & security & threat & enemy')
ORDER BY score DESC
LIMIT 5;
-- William J. Clinton | 1997-02-04 | 0.35810584
-- George W. Bush | 2004-01-20 | 0.29587495
-- George W. Bush | 2003-01-28 | 0.28381455
-- Harry S. Truman | 1946-01-21 | 0.25752166
-- William J. Clinton | 2000-01-27 | 0.22214262
SELECT president,
       speech_date,
       ts_rank(
           search_speech_text,
           to_tsquery('english', 'war & security & threat & enemy'),
           2
       )::numeric AS score
FROM president_speeches
WHERE search_speech_text @@
      to_tsquery('english', 'war & security & threat & enemy')
ORDER BY score DESC
LIMIT 5;
-- George W. Bush | 2004-01-20 | 0.000103
-- William J. Clinton | 1997-02-04 | 0.000098
-- George W. Bush | 2003-01-28 | 0.000096
-- Jimmy Carter | 1979-01-23 | 0.000090
-- Lyndon B. Johnson | 1968-01-17 | 0.000073
```

Long speeches accumulate token hits. Normalization option `2` divides rank by document length.

Compare the ordering before and after normalization. This is the difference between "mentions the keywords a lot" and "mentions them a lot relative to length."

The book points to a live demo at anthonydebarros.com/sotu. Worth clicking after class.

7 Part 6: Practice and Pitfalls

7.1 Try It: Crime Case Numbers

Write a query that returns `crime_id` and the case number extracted from `original_text` using the same `(?:CO|SO)[0-9]+` pattern, but filter to rows where the case starts with `SO`.

Stretch: Add a column that counts how many distinct dates appear in the narrative using `regexp_matches` with the `g` flag and a subquery or lateral trick. (If you get stuck, move on. The point is to think in matches, not to win a Nobel prize in SQL golf.)

7.2 Try It: Speeches

Find speeches where the query `economy & jobs` matches, show `ts_headline`, and sort by normalized rank (`ts_rank` with the length divisor). Pick a president you have opinions about. Keep the opinions out of the query.

7.3 Common Footguns (Collected)

- Forgetting that `regexp_match` returns `NULL` when nothing matches, then indexing `[1]` in a cast chain
- Using `LIKE` patterns with regex operators by mistake (they are not interchangeable)
- Building huge `OR` chains when a single well-factored regex would do
- Skipping the GIN index on `tsvector` columns and wondering why search "works on my laptop"

8 Summary

8.1 What We Covered

Topic	Core functions and ideas
Strings	`upper`, `lower`, `initcap`, `trim`, `left`/`right`, `replace`, `char_length`
Regex filters	`~`, `~*`, `!~`, `substring`, `regexp_replace`, splits
Structured extraction	`regexp_match`, `regexp_matches`, capture groups, `[1]` indexing
Full-text search	`to_tsvector`, `to_tsquery`, `@@`, `ts_headline`, `ts_rank`
Performance	GIN index on `tsvector`

8.2 The One-Sentence Version

Treat text like data: measure it, constrain it, extract what you need, and index what you search. Everything else is a spreadsheet someone emailed you as prose.

8.3 References

1. DeBarros, A. (2022). *Practical SQL* (2nd ed.). No Starch Press. Chapter 14.
2. PostgreSQL: Pattern Matching
3. PostgreSQL: String Functions
4. PostgreSQL: Regular Expressions
5. PostgreSQL: Full Text Search
6. PostgreSQL: ts_rank