# Data Mining with Text

**DATA 503: Fundamentals of Data Engineering**

Lucas P. Cordova, Ph.D.

2026-04-08

A concise version of text-oriented data work in PostgreSQL: string cleanup, POSIX regex for filters and extraction, parsing semi-structured crime narratives into columns, and full-text search with `tsvector` / `tsquery`, `@@`, GIN indexes, plus `ts_headline` and `ts_rank`. Checkpoints with hints, then solutions on the following slide. Based on Lecture 13-1 and Chapter 14 of Practical SQL, 2nd Edition.

## Table of contents



## 1 Learning objectives

### 1.1 What you should be able to do

1. Use core string functions for cleanup and slicing before heavier parsing
2. Apply `~` / `~*` and `regexp_match` / `regexp_matches` with capture groups to pull fields from text

3. Explain `tsvector` vs `tsquery`, filter with `@@`, and why GIN indexes matter for search at scale
4. Produce snippets with `ts_headline` and sort by relevance with `ts_rank` (including length normalization)

. . .

**Course connection:** Logs, tickets, and exports often arrive as prose. Regex and full-text search let you normalize and search near the database when you do not yet have a dedicated search service.

# 2 Load the practice data

## 2.1 Tables and sanity checks

| Table | Role |
|---|---|
| `county_regex_demo` | Regex operator practice |
| `crime_reports` | Semi-structured police narratives |
| `president_speeches` | Full-text search (State of the Union) |

**Check this:**

```
SELECT count(*) AS crime_reports FROM crime_reports;
SELECT count(*) AS speeches FROM president_speeches;
SELECT count(*) AS counties FROM county_regex_demo;
```

Expect `5`, `79`, and `8` respectively.

# 3 String functions

## 3.1 Why this layer exists

JSON and arrays get conference talks. Strings pay the rent: addresses, log lines, pasted emails, and CSV columns that are secretly paragraphs. PostgreSQL documents these under String Functions.

## 3.2 Toolkit (high level)

| Need | Examples |
|---|---|
| Case | `upper`, `lower`, `initcap` |
| Trim / measure | `trim`, `char_length`, `position` |
| Slice / replace | `left`, `right`, `replace` |

### 3.3 Demo: cleanup

**Run this.**

```
SELECT trim('  Pat  ') AS trimmed,
       char_length(trim('  Pat  ')) AS len,
       left('703-555-1212', 3) AS area;
```

`replace` is greedy and left-to-right: `replace('aaa', 'aa', 'b')` yields `ba`, which surprises everyone exactly once.

### 3.4 Checkpoint: initcap and replace

**Task:** From the literal `'  data engineering  '`, produce a single row with:

- `title_phrase`: `initcap` after `trim` (one column)
- `demo_replace`: `replace('batch-etl-batch', 'batch', 'stream')` (shows non-overlapping replacement)

**Hint:** Nest `initcap(trim(...))`.

### 3.5 Solution: initcap and replace

```
SELECT initcap(trim('  data engineering  ')) AS title_phrase,
       replace('batch-etl-batch', 'batch', 'stream') AS demo_replace;
```

## 4 Regular expressions

### 4.1 What regex is for

A **regular expression** describes text shape. PostgreSQL uses **POSIX** patterns (Pattern Matching). Mind backslashes: you often write `'\\d'` in SQL so the pattern sees one `\`.

## 4.2 Notation (compressed)

| Piece | Meaning |
|---|---|
| `\d` | Digit |
| `{n,m}` | Repeat between n and m |
| `()` | Group; also **capture** for `regexp_match` |
| `\|` | Alternation |
| `^ $` | Start and end of string (for whole-field tests) |

## 4.3 Operators

| Operator | Meaning |
|---|---|
| `~` | Matches (case sensitive) |
| `~*` | Matches (case insensitive) |
| `!~ / !~*` | Does not match |

## 4.4 Demo: filter counties

**Run this.**

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

`WHERE col ~ 'pattern'` does not match `NULL` rows.

## 4.5 Checkpoint: alternation

**Task:** Return rows from `county_regex_demo` where `county_name` matches `(lade|lare)` case-insensitively. Select `county_name` only, ordered by name.

**Expected:** At least **Clare County**.

**Hint:** One `WHERE` with `~*` and parentheses.

### 4.6 Solution: alternation

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

## 5 From blob to columns

### 5.1 The crime narrative shape

`crime_reports` stores `original_text` blobs. Dates, streets, offense labels, and case numbers sit inside the text, not in typed columns yet.

**Run this.**

```
SELECT original_text
FROM crime_reports
ORDER BY crime_id;
```

### 5.2 `regexp_match` vs `regexp_matches`

- **`regexp_match`:** first match as `text[]`, or `NULL`
- **`regexp_matches(..., 'g')`:** one row per non-overlapping match (set-returning)

**Run both** on `crime_id` order and compare row `1` (two dates in the narrative).

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

### 5.3 Capture groups and `[1]`

Parentheses **capture** substrings. Index with `[1]`, `[2]`, and so on when you want scalars.

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

`(?: … )` is a **non-capturing** group: it groups for precedence and repetition but does **not** add a numbered capture. That is why the case-number pattern uses `(?:CO|SO)` so the whole token still lands in `[1]`.

If the match fails, the array is `NULL` and so is `[1]`.

### 5.4 Capturing vs non-capturing (same URL)

**Run these** and compare the returned arrays.

```
-- Capturing: (s) is a numbered capture
SELECT regexp_match('https://site.com', 'http(s)?://');

-- Non-capturing: (?:s) groups the optional s without a capture
SELECT regexp_match('https://site.com', 'http(?:s)?://');
```

With **capturing** `(s)?`, PostgreSQL puts **only the captured substrings** in the result array (not the full match first). Here `[1]` is `'s'` when the URL uses `https`, because that is the first (and only) capture.

With **non-capturing** `(?:s)?`, there are no numbered captures left, so `regexp_match` returns a **one-element** array: the whole match, `'https://'`.

### 5.5 How about this?

**Run this** for plain `http`:

```
SELECT regexp_match('http://site.com', 'http(s)?://');    -- first capture empty: NULL
SELECT regexp_match('http://site.com', 'http(?:s)?://');   -- whole match: http://
```

### 5.6 Checkpoint: second date

**Task:** For each `crime_id`, return the **second** date in the narrative when it exists. Use `regexp_matches` with the `g` flag and pick the second element of the returned array, or use a lateral pattern you trust.

**Stretch (optional):** Only show rows where two dates exist.

**Hint:** `regexp_matches` with `g` returns one row per match; aggregate or filter on `ordinality` if you use `WITH ORDINALITY` in a lateral join, or take the second row in a subquery.

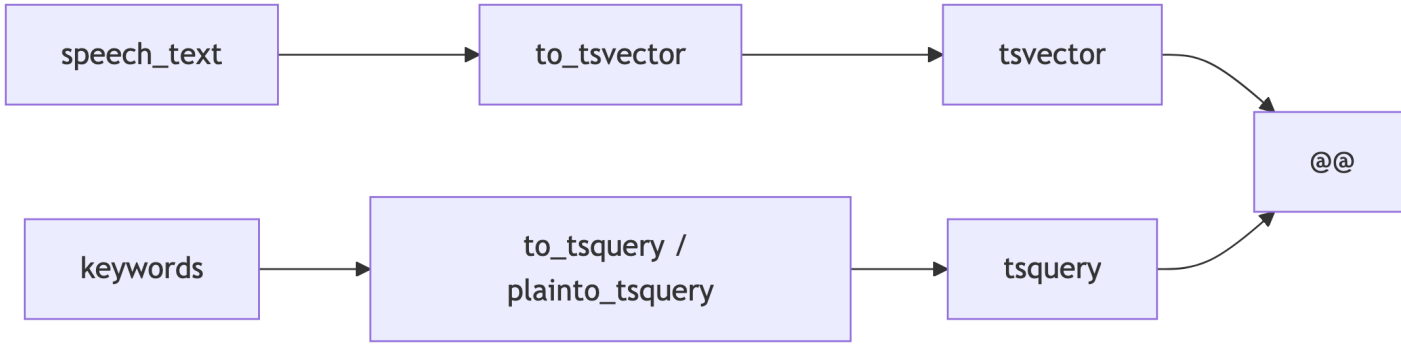
### 5.7 Solution: second date

```
SELECT c.crime_id,
       m.match[1] AS second_date
FROM crime_reports AS c
JOIN LATERAL (
    SELECT match
    FROM regexp_matches(
              c.original_text,
              '\d{1,2}/\d{1,2}/\d{2}',
              'g'
          ) WITH ORDINALITY AS t(match, ord)
    WHERE ord = 2
) AS m ON true
ORDER BY c.crime_id;
```

Rows with only one date in the narrative do not appear in this inner join result. `LEFT JOIN LATERAL` plus `NULL` would keep those rows if you need them.

# 6 Full-text search

## 6.1 Pipeline




Store or compute a **`tsvector`** for documents; build a **`tsquery`** for the question; **`@@`** is the match predicate. **`ts_headline`** formats snippets; **`ts_rank`** orders by relevance. A **GIN** index on the `tsvector` column keeps **`@@`** from scanning every row.

## 6.2 Types and functions (short)

| Piece | Role |
|---|---|
| `to_tsvector('english', text)` | Tokenize, drop stop words, stem |
| `to_tsquery` / `plainto_tsquery` | Build a query (`&`, `\|`, `!`, `<->`, `<N>` in `to_tsquery`) |
| `@@` | `tsvector @@ tsquery` |

`plainto_tsquery` is handy for simple keyword boxes; `to_tsquery` shows operators explicitly.

## 6.3 GIN in one minute

**GIN** is an inverted index: lexeme to row lists. It fits **`@@`** on **`tsvector`**. B-trees do not. Writes cost more than raw text; bulk load then create index is a common pattern.

The setup script already adds `search_speech_text` and a GIN index on `president_speeches`.

### 6.4 Demo: match and snippet

**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
LIMIT 5;
```

Filter with `@@` first; `ts_headline` is presentation on matching rows.

### 6.5 Demo: rank with normalization

**Run both** and compare top five ordering.

```
SELECT president, speech_date,
       ts_rank(search_speech_text, to_tsquery('english', 'war & security')) AS r0
FROM president_speeches
WHERE search_speech_text @@ to_tsquery('english', 'war & security')
ORDER BY r0 DESC
LIMIT 5;

SELECT president, speech_date,
       ts_rank(search_speech_text, to_tsquery('english', 'war & security'), 2)::numeric AS r
FROM president_speeches
WHERE search_speech_text @@ to_tsquery('english', 'war & security')
ORDER BY r2 DESC
LIMIT 5;
```

Flag 2 divides by document length so long speeches do not always win.

### 6.6 Checkpoint: economy and jobs

**Task:** Find speeches where **`economy and jobs`** both match (use `to_tsquery` with `&`). Return `president`, `speech_date`, and a **`ts_headline`** on `speech_text` for the query. Sort by **`ts_rank`** with normalization flag 2 descending; limit **10** rows.

**Hint:** Single `to_tsquery('english', 'economy & jobs')` for filter, headline, and rank.

### 6.7 Solution: economy and jobs

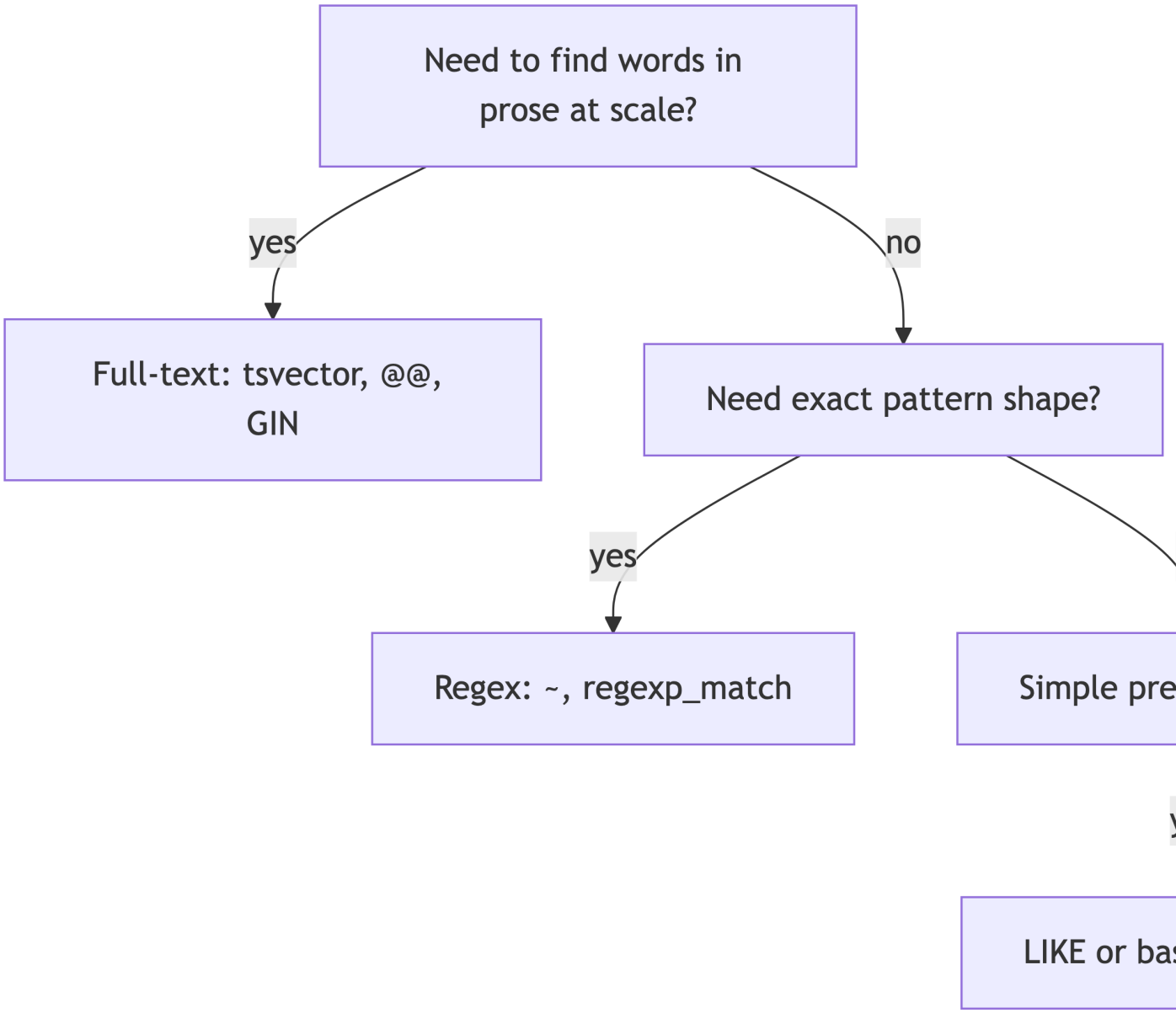
```sql
SELECT president,
       speech_date,
       ts_headline(
           speech_text,
           to_tsquery('english', 'economy & jobs'),
           'StartSel = <, StopSel = >, MinWords=5, MaxWords=10, MaxFragments=1'
       ),
       ts_rank(
           search_speech_text,
           to_tsquery('english', 'economy & jobs'),
           2
       )::numeric AS relevance
FROM president_speeches
WHERE search_speech_text @@ to_tsquery('english', 'economy & jobs')
ORDER BY relevance DESC
LIMIT 10;
```

## 7 Summary

### 7.1 Techniques in one place

| Technique | Use it when |
|---|---|
| `trim`, `left`, `replace` | Normalize display and prep for parsing |
| `~` / `~*`, `regexp_match` | Filters and first structured pull from text |
| `regexp_matches(…, 'g')` | Every occurrence; drive lateral or counting |
| `to_tsvector` + GIN | Searchable document column |
| `@@`, `ts_headline`, `ts_rank` | Filter, snippet, sort |

### 7.2 Choosing a tool




### 7.3 References


1. DeBarros, A. (2022). *Practical SQL* (2nd ed.). No Starch Press. Chapter 14.
2. PostgreSQL: Pattern Matching
3. PostgreSQL: String Functions

4. PostgreSQL: Full Text Search
5. PostgreSQL: ts_rank