

# Mozilla tokenizer setup



what do you know about an fts3 tokenizer in SQLite?

SQLite's **FTS3 tokenizer** is a pluggable component responsible for breaking text into searchable terms for **Full-Text Search (FTS3/FTS4)** virtual tables. Here's a concise overview:

## What an FTS3 Tokenizer Does

When you insert text into an FTS3/FTS4 table, SQLite doesn't index the raw string. Instead, it passes the text through a **tokenizer**, which:

1. **Splits text into tokens** (words/terms)
2. **Normalizes** them (lowercasing, removing punctuation, etc.)
3. Optionally filters stop-words
4. Returns tokens with positional information for indexing

## Built-in Tokenizers

SQLite ships with a few tokenizers:

### 1. **simple**

- Splits on non-alphanumeric characters.
- Lowercases ASCII only.
- No Unicode awareness.
- No stemming.

### 2. **porter**

- Same as **simple** but adds **Porter stemming** (English).

### 3. **unicode61 (default for FTS4)**

- Unicode-aware (uses Unicode 6.1 word-boundary rules)
- Case folding for many languages
- Configurable:
  - **tokenchars=** to include additional characters
  - **separators=** to treat characters as delimiters

- `remove_diacritics=0/1/2`
- `stemmer=...` (SQLite 3.33+)

Example:

```
sql
CREATE VIRTUAL TABLE docs USING fts4(
  content,
  tokenize = unicode61 "remove_diacritics=2"
);
```

## Custom Tokenizers

You can create your own tokenizers in C (or via extensions like `fts3tokenize` or `sqlite-fts5-tokenizer`) using the **SQLite tokenizer API**, which requires implementing:

- `xCreate`
- `xDestroy`
- `xOpen`
- `xClose`
- `xNext` (main tokenizing function)

You register it with:

```
c
sqlite3Fts3InitTokenizer(db, "mytokenizer", &MyTokenizerModule);
```

This allows:

- Language-specific processing
- Custom stemming
- Token filtering (e.g., stop-words)
- Splitting on domain-specific syntax (like code, hashtags, etc.)

## Bonus: The FTS3 Tokenizer Extension

SQLite includes a special virtual table module for debugging tokenizers:

```
sql
CREATE VIRTUAL TABLE t USING fts3tokenize(unicode61);
SELECT * FROM t WHERE input='This is a test.';
```

Returns columns:

- **token**
- **start**
- **end**
- **position**

Useful for inspecting how the tokenizer works.

## ⚠️ **FTS3 vs FTS4 vs FTS5**

- **FTS3/FTS4** share the tokenizer code and accept the same tokenizers.
- **FTS5 has its own tokenizer API** (simpler but not directly compatible).

If you want, I can also:

- ✓ Write an example custom tokenizer
- ✓ Explain FTS5's tokenizer
- ✓ Show how to debug a tokenizer or anything else.



Well, I am looking at some Mozilla code which uses its own tokenizer. To simplify things, let's say the table is created with the special tokenizer and two columns, body and subject, for indexing e-mail. Makes sense so far?

Yes — that makes perfect sense.

If Mozilla is using its own tokenizer, then they've implemented (or wrapped) an **FTS3/FTS4 custom tokenizer module**, and the FTS virtual table is declared something like:

```
sql

CREATE VIRTUAL TABLE messages USING fts4(
  body,
  subject,
  tokenize = mozTokenizer
);
```

(or equivalent; Mozilla historically used **FTS3** with a custom tokenizer implemented in C++.)

In this setup:

- **The tokenizer is invoked separately for each column (body and subject).**
- The tokenizer determines exactly how the text is split into indexed terms.
- FTS3 stores the resulting (token, rowid, column, position) tuples in its inverted index.

This is a common pattern for email indexing:

- **body** gets full-text tokenization

- **subject** may get slightly different tokenization (e.g., different weighting or handling punctuation)

But in FTS3/FTS4, both are handled through the **same tokenizer**, though the tokenizer can use the *column number* if it wants to treat them differently.

So yes — a custom tokenizer + two-column FTS virtual table for email indexing is a totally expected architecture and consistent with what Mozilla implemented.

If you want, I can walk you through:

- how FTS3 calls the tokenizer internally
- how column boundaries are handled
- how terms for different columns are stored in the index
- or how Mozilla's specific tokenizer behaves (if you want to paste a snippet)

What would you like to examine next?



let's say the virtual table is called `messagesText`. The query seems to be, roughly:

```
select * from messagesText WHERE messagesText MATCH "test"
```

Does that make sense?

Yes — that query is exactly what you would expect with an FTS3/FTS4 virtual table using a custom tokenizer.

### ✓ What the query means

```
sql
```

```
SELECT * FROM messagesText
WHERE messagesText MATCH "test";
```

This tells SQLite:

“Search all indexed columns of `messagesText` for the token **test**, using the tokenizer associated with this FTS table.”

### How SQLite interprets this under the hood

1. The right-hand side ("**test**") is passed into the **same tokenizer** used when indexing.
2. The tokenizer produces one or more tokens.
  - If the tokenizer lowercases, stems, splits, etc., the query terms go through all that.
3. SQLite searches the FTS index for rows where *either* of the indexed columns (**body** or **subject**) contains those tokens.
4. Matching rows are returned — by default with all columns.

## Equivalent / explicit form

If the table has two indexed columns (**body**, **subject**), then:

```
sql
messagesText MATCH "test"
```

is the shorthand for:

```
sql
messagesText.body MATCH "test" OR messagesText.subject MATCH "test"
```

## Important note

With FTS3/4, the left side of the MATCH operator **must be the table name**, not the column name (column MATCH is allowed only in FTS5). So the syntax you showed is fully correct for FTS3/4.

## What this means for the custom (Mozilla) tokenizer

Your custom tokenizer determines:

- What counts as a token
- Whether "test" becomes **test**, or **Test**, or **tes**, etc.
- Whether punctuation affects matching
- Unicode/language behavior
- Whether "test" matches "Test" or "testing" or "test-case"

So yes — the query is correct, and exactly what one would expect in a Mozilla-style email indexer using FTS3 with a custom tokenizer.

---

If you want, I can also explain:

- ♦ how MATCH transforms multi-word queries
- ♦ how boolean operators (AND/OR/NEAR) work
- ♦ how prefix searches (**test\***) are handled
- ♦ how the tokenizer affects ranking / snippet extraction

Just tell me which direction you'd like to go.



So you said: Search all indexed columns of messagesText for the token test, using the tokenizer associated with this FTS table.

That means, all columns are searched. So if you get a row returned, you don't know from which column the hit came.

Correct — **if you use MATCH in that form, SQLite will return rows without telling you which column(s) matched.**

This is a known behavior of **FTS3/FTS4**:

✓ `messagesText MATCH "test"`

Searches **all indexed columns**, and the result set does **not indicate which column matched**.

The query returns only rows.

FTS3/4 does *store* column/position info internally (so it can do snippet generation and ranking), but the **SELECT** result does not expose which column contained the match.

## If you *need* to know which column matched

### Option 1 — Use the `offsets()` function

SQLite provides:

```
sql
SELECT offsets(messagesText) FROM messagesText
WHERE messagesText MATCH "test";
```

`offsets()` returns a text blob describing where matches occurred:

```
sql
column position offset length column position offset length ...
```

Example return:

```
arduino
"0 3 10 4 1 0 0 4"
```

Meaning:

- column 0 (body), position 3, byte offset 10, length 4
- column 1 (subject), position 0, byte offset 0, length 4

So this tells you **exactly which column(s)** matched.

### Option 2 — Restrict **MATCH** to a specific column (FTS3/4 special syntax)

In FTS3/4 you can prefix the query with a column name **inside the MATCH string**:

```
sql
SELECT * FROM messagesText
WHERE messagesText MATCH 'subject:test';
```

or:

```
sql
... MATCH 'body:test';
```

This forces matching only in that column.

### Option 3 — “OR” the columns explicitly

FTS3/4 doesn't support `body MATCH "test"` directly, but this workaround exists:

```
sql
SELECT *,
  messagesText MATCH 'body:test' AS bodyHit,
  messagesText MATCH 'subject:test' AS subjectHit
FROM messagesText
WHERE messagesText MATCH 'test';
```

Now you get boolean indicators for each column.

## ✓ So your statement is true

If you get a row returned, you don't know from which column the hit came.

Yes — at least not from the row data itself.

To know the columns, you must use `offsets()`, constrain the query, or perform column-prefixed `MATCH` queries.

If you want, I can show you how Mozilla's tokenizer interacts with `offsets()` or how to decode `snippet()` results too.



Oh, so the code I'm looking at, also selects on offsets. So looking at those offsets, you can perfectly determine which column or columns caused the match of the row.

Exactly — **that's precisely what `offsets()` is for.**

## ✓ `offsets()` lets you *deterministically* identify which column(s) matched

When FTS3/FTS4 returns a row, you only know:

“This row matched the query somewhere.”

But if your query also selects:

```
sql
SELECT offsets(messagesText) ...
```

you get back a string encoding every match location:

```
sql
column position offset length column position offset length ...
```

From this, you can determine:

- which **column** matched (0 = first column, 1 = second column, etc.)
- which **token position** within the column
- where in the source text the match occurred
- how long the matched token is

So yes — with `offsets()`, you can *perfectly and unambiguously* know:

- whether the match came from the **subject**
- from the **body**
- from **both**
- and even *where in the body/subject* the token came from.

This is exactly how snippet generation works internally.

## ✓ Example

Given:

```
sql
CREATE VIRTUAL TABLE messagesText USING fts4(body, subject);
```

Query:

```
sql
SELECT rowid, offsets(messagesText)
FROM messagesText
WHERE messagesText MATCH 'test';
```

Suppose the result is:

```
arduino
"1 0 15 4 0 5 88 4"
```

Interpretation:

- **1 0 15 4**
  - column 1 → **subject**
  - token position 0
  - at byte offset 15
  - length 4
- **0 5 88 4**
  - column 0 → **body**
  - token position 5
  - at byte offset 88
  - length 4

This row matched **both** the subject and body.

---

## ✓ Why Mozilla code would use this

Email apps often need to:

- Highlight search terms
- Rank results differently depending on whether they match in the **subject** vs **body**
- Display partial snippets
- Automatically scroll to the match location

All of these require knowing match locations, which `offsets()` provides.

---

If you want, I can also explain:

- How to parse the offsets string programmatically
- Why offsets appear in pairs of (column, position, offset, length)
- How `snippet()` uses the same internal data
- How a custom tokenizer affects offsets

Just tell me!