SQLite FTS5 Extension

▶ Table Of Contents

1. Overview of FTS5

FTS5 is an SQLite virtual table module that provides full-text search functionality to database applications. In their most elementary form, full-text search engines allow the user to efficiently search a large collection of documents for the subset that contain one or more instances of a search term. The search functionality provided to world wide web users by Google is, among other things, a full-text search engine, as it allows users to search for all documents on the web that contain, for example, the term "fts5".

To use FTS5, the user creates an FTS5 virtual table with one or more columns. For example:

```
CREATE VIRTUAL TABLE email USING fts5(sender, title, body);
```

It is an error to add types, constraints or PRIMARY KEY declarations to a CREATE VIRTUAL TABLE statement used to create an FTS5 table. Once created, an FTS5 table may be populated using INSERT, UPDATE or DELETE statements like any other table. Like any other table with no PRIMARY KEY declaration, an FTS5 table has an implicit INTEGER PRIMARY KEY field named rowid.

Not shown in the example above is that there are also various options that may be provided to FTS5 as part of the CREATE VIRTUAL TABLE statement to configure various aspects of the new table. These may be used to modify the way in which the FTS5 table extracts terms from documents and queries, to create extra indexes on disk to speed up prefix queries, or to create an FTS5 table that acts as an index on content stored elsewhere.

Once populated, there are three ways to execute a full-text query against the contents of an FTS5 table:

- Using a MATCH operator in the WHERE clause of a SELECT statement, or
- Using an equals ("=") operator in the WHERE clause of a SELECT statement, or
- using the table-valued function syntax.

If using the MATCH or = operators, the expression to the left of the MATCH operator is usually the name of the FTS5 table (the exception is when specifying a column-filter). The expression on the right must be a text value specifying the term to search for. For the table-valued function syntax, the term to search for is specified as the first table argument. For example:

```
-- Query for all rows that contain at least once instance of the term
-- "fts5" (in any column). The following three queries are equivalent.
SELECT * FROM email WHERE email MATCH 'fts5';
SELECT * FROM email WHERE email = 'fts5';
SELECT * FROM email('fts5');
```

By default, FTS5 full-text searches are case-independent. Like any other SQL query that does not contain an ORDER BY clause, the example above returns results in an arbitrary order. To sort results by relevance (most to least relevant), an ORDER BY may be added to a full-text query as follows:

```
-- Query for all rows that contain at least once instance of the term
-- "fts5" (in any column). Return results in order from best to worst
-- match.
SELECT * FROM email WHERE email MATCH 'fts5' ORDER BY rank;
```

As well as the column values and rowid of a matching row, an application may use FTS5 auxiliary functions to retrieve extra information regarding the matched row. For example, an auxiliary function may be used to retrieve a copy of a column value for a matched row with all instances of the matched term surrounded by html tags. Auxiliary functions are invoked in the same way as SQLite scalar functions, except that the name of the FTS5 table is specified as the first argument. For example:

```
-- Query for rows that match "fts5". Return a copy of the "body" column
-- of each row with the matches surrounded by <b></b> tags.
SELECT highlight(email, 2, '<b>', '</b>') FROM email('fts5');
```

A description of the available auxiliary functions, and more details regarding configuration of the special "rank" column, are available below. Custom auxiliary functions may also be implemented in C and registered with FTS5, just as custom SQL functions may be registered with the SQLite core.

As well as searching for all rows that contain a term, FTS5 allows the user to search for rows that contain:

- any terms that begin with a specified prefix,
- "phrases" - sequences of terms or prefix terms that must feature in a document for it to match the query,
- sets of terms, prefix terms or phrases that appear within a specified proximity of each other (these are called "NEAR queries"), or
- boolean combinations of any of the above.

Such advanced searches are requested by providing a more complicated FTS5 query string as the text to the right of the MATCH operator (or = operator, or as the first argument to a table-valued function syntax). The full query syntax is described here.

2. Compiling and Using FTS5

2.1. Building FTS5 as part of SQLite

As of version 3.9.0 (2015-10-14), FTS5 is included as part of the SQLite amalgamation. If using one of the two autoconf build system, FTS5 is enabled by specifying the "--enable-fts5" option when running the configure script. (FTS5 is currently disabled by default for the source-tree configure script and enabled by default for the amalgamation configure script, but these defaults might change in the future.)

Or, if sqlite3.c is compiled using some other build system, by arranging for the SQLITE_ENABLE_FTS5 pre-processor symbol to be defined.

2.2. Building a Loadable Extension

Alternatively, FTS5 may be built as a loadable extension.

The canonical FTS5 source code consists of a series of *.c and other files in the "ext/fts5" directory of the SQLite source tree. A build process reduces this to just two files - "fts5.c" and "fts5.h" - which may

be used to build an SQLite loadable extension.

1. Obtain the latest SQLite code from fossil.
2. Create a Makefile as described in How To Compile SQLite.
3. Build the "fts5.c" target. Which also creates fts5.h.

```
$ wget -c https://www.sqlite.org/src/tarball/SQLite-trunk.tgz?uuid=trunk -O SQLite-trunk.tgz
.... output ...
$ tar -xzf SQLite-trunk.tgz
$ cd SQLite-trunk
$ ./configure && make fts5.c
... lots of output ...
$ ls fts5.[ch]
fts5.c        fts5.h
```

The code in "fts5.c" may then be compiled into a loadable extension or statically linked into an application as described in Compiling Loadable Extensions. There are two entry points defined, both of which do the same thing:

- sqlite3_fts_init
- sqlite3_fts5_init

The other file, "fts5.h", is not required to compile the FTS5 extension. It is used by applications that implement custom FTS5 tokenizers or auxiliary functions.

3. Full-text Query Syntax

The following block contains a summary of the FTS query syntax in BNF form. A detailed explanation follows.

```
<phrase>    := string [*]
<phrase>    := <phrase> + <phrase>
<neargroup> := NEAR ( <phrase> <phrase> ... [, N] )
<query>     := [ [-] <colspec> :] [^] <phrase>
<query>     := [ [-] <colspec> :] <neargroup>
<query>     := [ [-] <colspec> :] ( <query> )
<query>     := <query> AND <query>
<query>     := <query> OR <query>
<query>     := <query> NOT <query>
<colspec>   := colname
<colspec>   := { colname1 colname2 ... }
```

3.1. FTS5 Strings

Within an FTS expression a **string** may be specified in one of two ways:

- By enclosing it in double quotes ("). Within a string, any embedded double quote characters may be escaped SQL-style - by adding a second double-quote character.
- As an FTS5 bareword that is not "AND", "OR" or "NOT" (case sensitive). An FTS5 bareword is a string of one or more consecutive characters that are all either:
  - Non-ASCII range characters (i.e. unicode codepoints greater than 127), or
  - One of the 52 upper and lower case ASCII characters, or
  - One of the 10 decimal digit ASCII characters, or
  - The underscore character (unicode codepoint 96).
  - The substitute character (unicode codepoint 26).

  Strings that include any other characters must be quoted. Characters that are not currently allowed in barewords, are not quote characters and do not currently serve any special purpose in FTS5 query expressions may at some point in the future be allowed in barewords or used to

implement new query functionality. This means that queries that are currently syntax errors because they include such a character outside of a quoted string may be interpreted differently by some future version of FTS5.

3.2. FTS5 Phrases

Each string in an fts5 query is parsed ("tokenized") by the tokenizer and a list of zero or more **tokens**, or terms, extracted. For example, the default tokenizer tokenizes the string "alpha beta gamma" to three separate tokens - "alpha", "beta" and "gamma" - in that order.

FTS queries are made up of **phrases**. A phrase is an ordered list of one or more tokens. The tokens from each string in the query each make up a single phrase. Two phrases can be concatenated into a single large phrase using the "+" operator. For example, assuming the tokenizer module being used tokenizes the input "one.two.three" to three separate tokens, the following four queries all specify the same phrase:

```
... MATCH '"one two three"'
... MATCH 'one + two + three'
... MATCH '"one two" + three'
... MATCH 'one.two.three'
```

A phrase matches a document if the document contains at least one sub-sequence of tokens that matches the sequence of tokens that make up the phrase.

3.3. FTS5 Prefix Queries

If a "*" character follows a string within an FTS expression, then the final token extracted from the string is marked as a **prefix token**. As you might expect, a prefix token matches any document token of which it is a prefix. For example, the first two queries in the following block will match any document that contains the token "one" immediately followed by the token "two" and then any token that begins with "thr".

```
... MATCH '"one two thr" * '
... MATCH 'one + two + thr*'
... MATCH '"one two thr*"'      -- May not work as expected!
```

The final query in the block above may not work as expected. Because the "*" character is inside the double-quotes, it will be passed to the tokenizer, which will likely discard it (or perhaps, depending on the specific tokenizer in use, include it as part of the final token) instead of recognizing it as a special FTS character.

3.4. FTS5 Initial Token Queries

If a "^" character appears immediately before a phrase that is not part of a NEAR query, then that phrase only matches a document only if it starts at the first token in a column. The "^" syntax may be combined with a column filter, but may not be inserted into the middle of a phrase.

```
... MATCH '^one'              -- first token in any column must be "one"
... MATCH '^ one + two'       -- phrase "one two" must appear at start of a column
... MATCH '^ "one two"'       -- same as previous
... MATCH 'a : ^two'          -- first token of column "a" must be "two"
... MATCH 'NEAR(^one, two)'   -- syntax error!
... MATCH 'one + ^two'        -- syntax error!
... MATCH '"^one two"'        -- May not work as expected!
```

3.5. FTS5 NEAR Queries

Two or more phrases may be grouped into a **NEAR group**. A NEAR group is specified by the token "NEAR" (case sensitive) followed by an open parenthesis character, followed by two or more whitespace separated phrases, optionally followed by a comma and the numeric parameter *N*, followed by a close parenthesis. For example:

```
... MATCH 'NEAR("one two" "three four", 10)'
... MATCH 'NEAR("one two" thr* + four)'
```

If no *N* parameter is supplied, it defaults to 10. A NEAR group matches a document if the document contains at least one clump of tokens that:

1. contains at least one instance of each phrase, and
2. for which the number of tokens between the end of the first phrase and the beginning of the last phrase in the clump is less than or equal to *N*.

For example:

```
CREATE VIRTUAL TABLE ft USING fts5(x);
INSERT INTO ft(rowid, x) VALUES(1, 'A B C D x x x E F x');

... MATCH 'NEAR(e d, 4)';                      -- Matches!
... MATCH 'NEAR(e d, 3)';                      -- Matches!
... MATCH 'NEAR(e d, 2)';                      -- Does not match!

... MATCH 'NEAR("c d" "e f", 3)';              -- Matches!
... MATCH 'NEAR("c"   "e f", 3)';              -- Does not match!

... MATCH 'NEAR(a d e, 6)';                    -- Matches!
... MATCH 'NEAR(a d e, 5)';                    -- Does not match!

... MATCH 'NEAR("a b c d" "b c" "e f", 4)';    -- Matches!
... MATCH 'NEAR("a b c d" "b c" "e f", 3)';    -- Does not match!
```

3.6. FTS5 Column Filters

A single phrase or NEAR group may be restricted to matching text within a specified column of the FTS table by prefixing it with the column name followed by a colon character. Or to a set of columns by prefixing it with a whitespace separated list of column names enclosed in parenthesis ("curly brackets") followed by a colon character. Column names may be specified using either of the two forms described for strings above. Unlike strings that are part of phrases, column names are not passed to the tokenizer module. Column names are case-insensitive in the usual way for SQLite column names - upper/lower case equivalence is understood for ASCII-range characters only.

```
... MATCH 'colname : NEAR("one two" "three four", 10)'
... MATCH '"colname" : one + two + three'

... MATCH '{col1 col2} : NEAR("one two" "three four", 10)'
... MATCH '{col2 col1 col3} : one + two + three'
```

If a column filter specification is preceded by a "-" character, then it is interpreted as a list of column not to match against. For example:

```
-- Search for matches in all columns except "colname"
... MATCH '- colname : NEAR("one two" "three four", 10)'

-- Search for matches in all columns except "col1", "col2" and "col3"
... MATCH '- {col2 col1 col3} : one + two + three'
```

Column filter specifications may also be applied to arbitrary expressions enclosed in parenthesis. In this case the column filter applies to all phrases within the expression. Nested column filter operations

may only further restrict the subset of columns matched, they can not be used to re-enable filtered columns. For example:

```
-- The following are equivalent:
... MATCH '{a b} : ( {b c} : "hello" AND "world" )'
... MATCH '(b : "hello") AND ({a b} : "world")'
```

Finally, a column filter for a single column may be specified by using the column name as the LHS of a MATCH operator (instead of the usual table name). For example:

```
-- Given the following table
CREATE VIRTUAL TABLE ft USING fts5(a, b, c);

-- The following are equivalent
SELECT * FROM ft WHERE b MATCH 'uvw AND xyz';
SELECT * FROM ft WHERE ft MATCH 'b : (uvw AND xyz)';

-- This query cannot match any rows (since all columns are filtered out):
SELECT * FROM ft WHERE b MATCH 'a : xyz';
```

3.7. FTS5 Boolean Operators

Phrases and NEAR groups may be arranged into expressions using **boolean operators**. In order of precedence, from highest (tightest grouping) to lowest (loosest grouping), the operators are:

Operator	Function
`<query1> NOT <query2>`	Matches if query1 matches and query2 does not match.
`<query1> AND <query2>`	Matches if both query1 and query2 match.
`<query1> OR <query2>`	Matches if either query1 or query2 match.

Parenthesis may be used to group expressions in order to modify operator precedence in the usual ways. For example:

```
-- Because NOT groups more tightly than OR, either of the following may
-- be used to match all documents that contain the token "two" but not
-- "three", or contain the token "one".
... MATCH 'one OR two NOT three'
... MATCH 'one OR (two NOT three)'

-- Matches documents that contain at least one instance of either "one"
-- or "two", but do not contain any instances of token "three".
... MATCH '(one OR two) NOT three'
```

Phrases and NEAR groups may also be connected by **implicit AND operators**. For simplicity, these are not shown in the BNF grammar above. Essentially, any sequence of phrases or NEAR groups (including those restricted to matching specified columns) separated only by whitespace are handled as if there were an implicit AND operator between each pair of phrases or NEAR groups. Implicit AND operators are never inserted after or before an expression enclosed in parenthesis. Implicit AND operators group more tightly than all other operators, including NOT. For example:

```
... MATCH 'one two three'         -- 'one AND two AND three'
... MATCH 'three "one two"'       -- 'three AND "one two"'
... MATCH 'NEAR(one two) three'   -- 'NEAR(one two) AND three'
... MATCH 'one OR two three'      -- 'one OR two AND three'
... MATCH 'one NOT two three'     -- 'one NOT (two AND three)'

... MATCH '(one OR two) three'    -- Syntax error!
... MATCH 'func(one two)'         -- Syntax error!
```

4. FTS5 Table Creation and Initialization

Each argument specified as part of a "CREATE VIRTUAL TABLE ... USING fts5 ..." statement is either a column declaration or a configuration option. A **column declaration** consists of one or more whitespace separated FTS5 barewords or string literals quoted in any manner acceptable to SQLite.

The first string or bareword in a column declaration is the column name. It is an error to attempt to name an fts5 table column "rowid" or "rank", or to assign the same name to a column as is used by the table itself. This is not supported.

Each subsequent string or bareword in a column declaration is a column option that modifies the behaviour of that column. Column options are case-independent. Unlike the SQLite core, FTS5 considers unrecognized column options to be errors. Currently, the only option recognized is "UNINDEXED" (see below).

A **configuration option** consists of an FTS5 bareword - the option name - followed by an "=" character, followed by the option value. The option value is specified using either a single FTS5 bareword or a string literal, again quoted in any manner acceptable to the SQLite core. For example:

```
CREATE VIRTUAL TABLE mail USING fts5(sender, title, body, tokenize = 'porter ascii');
```

There are currently the following configuration options:

- The "tokenize" option, used to configure a custom tokenizer.
- The "prefix" option, used to add prefix indexes to an FTS5 table.
- The "content" option, used to make the FTS5 table an external content or contentless table.
- The "content_rowid" option, used to set the rowid field of an external content table.
- The "columnsize" option, used to configure whether or not the size in tokens of each value in the FTS5 table is stored separately within the database.
- The "detail" option. This option may be used to reduce the size of the FTS index on disk by omitting some information from it.

4.1. The UNINDEXED column option

The contents of columns qualified with the UNINDEXED column option are not added to the FTS index. This means that for the purposes of MATCH queries and FTS5 auxiliary functions, the column contains no matchable tokens.

For example, to avoid adding the contents of the "uuid" field to the FTS index:

```
CREATE VIRTUAL TABLE customers USING fts5(name, addr, uuid UNINDEXED);
```

4.2. Prefix Indexes

By default, FTS5 maintains a single index recording the location of each token instance within the document set. This means that querying for complete tokens is fast, as it requires a single lookup, but

querying for a prefix token can be slow, as it requires a range scan. For example, to query for the prefix token "abc*" requires a range scan of all tokens greater than or equal to "abc" and less than "abd".

A prefix index is a separate index that records the location of all instances of prefix tokens of a certain length in characters used to speed up queries for prefix tokens. For example, optimizing a query for prefix token "abc*" requires a prefix index of three-character prefixes.

To add prefix indexes to an FTS5 table, the "prefix" option is set to either a single positive integer or a text value containing a white-space separated list of one or more positive integer values. A prefix index is created for each integer specified. If more than one "prefix" option is specified as part of a single CREATE VIRTUAL TABLE statement, all apply.

```
-- Two ways to create an FTS5 table that maintains prefix indexes for
-- two and three character prefix tokens.
CREATE VIRTUAL TABLE ft USING fts5(a, b, prefix='2 3');
CREATE VIRTUAL TABLE ft USING fts5(a, b, prefix=2, prefix=3);
```

4.3. Tokenizers

The CREATE VIRTUAL TABLE "tokenize" option is used to configure the specific tokenizer used by the FTS5 table. The option argument must be either an FTS5 bareword, or an SQL text literal. The text of the argument is itself treated as a white-space series of one or more FTS5 barewords or SQL text literals. The first of these is the name of the tokenizer to use. The second and subsequent list elements, if they exist, are arguments passed to the tokenizer implementation.

Unlike option values and column names, SQL text literals intended as tokenizers must be quoted using single quote characters. For example:

```
-- The following are all equivalent
CREATE VIRTUAL TABLE ft USING fts5(x, tokenize = 'porter ascii');
CREATE VIRTUAL TABLE ft USING fts5(x, tokenize = "porter ascii");
CREATE VIRTUAL TABLE ft USING fts5(x, tokenize = "'porter' 'ascii'");
CREATE VIRTUAL TABLE ft USING fts5(x, tokenize = '''porter'' ''ascii''');

-- But this will fail:
CREATE VIRTUAL TABLE ft USING fts5(x, tokenize = '"porter" "ascii"');

-- This will fail too:
CREATE VIRTUAL TABLE ft USING fts5(x, tokenize = 'porter' 'ascii');
```

FTS5 features four built-in tokenizer modules, described in subsequent sections:

- The **unicode61** tokenizer, based on the Unicode 6.1 standard. This is the default.
- The **ascii** tokenizer, which assumes all characters outside of the ASCII codepoint range (0-127) are to be treated as token characters.
- The **porter** tokenizer, which implements the porter stemming algorithm.
- The **trigram** tokenizer, which treats each contiguous sequence of three characters as a token, allowing FTS5 to support more general substring matching.

It is also possible to create custom tokenizers for FTS5. The API for doing so is described here.

4.3.1. Unicode61 Tokenizer

The unicode tokenizer classifies all unicode characters as either "separator" or "token" characters. By default all space and punctuation characters, as defined by Unicode 6.1, are considered separators, and all other characters as token characters. More specifically, all unicode characters assigned to a general category beginning with "L" or "N" (letters and numbers, specifically) or to category "Co" ("other, private use") are considered tokens. All other characters are separators.

Each contiguous run of one or more token characters is considered to be a token. The tokenizer is case-insensitive according to the rules defined by Unicode 6.1.

By default, diacritics are removed from all Latin script characters. This means, for example, that "A", "a", "À", "à", "Â" and "â" are all considered to be equivalent.

Any arguments following "unicode61" in the token specification are treated as a list of alternating option names and values. Unicode61 supports the following options:

Option	Usage
remove_diacritics	This option should be set to "0", "1" or "2". The default value is "1". If it is set to "1" or "2", then diacritics are removed from Latin script characters as described above. However, if it is set to "1", then diacritics are not removed in the fairly uncommon case where a single unicode codepoint is used to represent a character with more that one diacritic. For example, diacritics are not removed from codepoint 0x1ED9 ("LATIN SMALL LETTER O WITH CIRCUMFLEX AND DOT BELOW"). This is technically a bug, but cannot be fixed without creating backwards compatibility problems. If this option is set to "2", then diacritics are correctly removed from all Latin characters.
categories	This option may be used to modify the set of Unicode general categories that are considered to correspond to token characters. The argument must consist of a space separated list of two-character general category abbreviations (e.g. "Lu" or "Nd"), or of the same with the second character replaced with an asterisk ("*"), interpreted as a glob pattern. The default value is "L* N* Co".
tokenchars	This option is used to specify additional unicode characters that should be considered token characters, even if they are white-space or punctuation characters according to Unicode 6.1. All characters in the string that this option is set to are considered token characters.
separators	This option is used to specify additional unicode characters that should be considered as separator characters, even if they are token characters according to Unicode 6.1. All characters in the string that this option is set to are considered separators.

For example:

```
-- Create an FTS5 table that does not remove diacritics from Latin
-- script characters, and that considers hyphens and underscore characters
-- to be part of tokens.
CREATE VIRTUAL TABLE ft USING fts5(a, b,
    tokenize = "unicode61 remove_diacritics 0 tokenchars '-_'"
);
```

or:

```
-- Create an FTS5 table that, as well as the default token character classes,
-- considers characters in class "Mn" to be token characters.
CREATE VIRTUAL TABLE ft USING fts5(a, b,
    tokenize = "unicode61 categories 'L* N* Co Mn'"
);
```

The fts5 unicode61 tokenizer is byte-for-byte compatible with the fts3/4 unicode61 tokenizer.

4.3.2. Ascii Tokenizer

The Ascii tokenizer is similar to the Unicode61 tokenizer, except that:

- All non-ASCII characters (those with codepoints greater than 127) are always considered token characters. If any non-ASCII characters are specified as part of the separators option, they are ignored.
- Case-folding is only performed for ASCII characters. So while "A" and "a" are considered to be equivalent, "Ã" and "ã" are distinct.
- The remove_diacritics option is not supported.

For example:

```
-- Create an FTS5 table that uses the ascii tokenizer, but does not
-- consider numeric characters to be part of tokens.
CREATE VIRTUAL TABLE ft USING fts5(a, b,
    tokenize = "ascii separators '0123456789'"
);
```

4.3.3. Porter Tokenizer

The porter tokenizer is a wrapper tokenizer. It takes the output of some other tokenizer and applies the porter stemming algorithm to each token before it returns it to FTS5. This allows search terms like "correction" to match similar words such as "corrected" or "correcting". The porter stemmer algorithm is designed for use with English language terms only - using it with other languages may or may not improve search utility.

By default, the porter tokenizer operates as a wrapper around the default tokenizer (unicode61). Or, if one or more extra arguments are added to the "tokenize" option following "porter", they are treated as a specification for the underlying tokenizer that the porter stemmer uses. For example:

```
-- Two ways to create an FTS5 table that uses the porter tokenizer to
-- stem the output of the default tokenizer (unicode61).
CREATE VIRTUAL TABLE ft USING fts5(x, tokenize = porter);
CREATE VIRTUAL TABLE ft USING fts5(x, tokenize = 'porter unicode61');

-- A porter tokenizer used to stem the output of the unicode61 tokenizer,
-- with diacritics removed before stemming.
CREATE VIRTUAL TABLE ft USING fts5(x, tokenize = 'porter unicode61 remove_diacritics 1');
```

4.3.4. The Trigram Tokenizer

The trigram tokenizer extends FTS5 to support substring matching in general, instead of the usual token matching. When using the trigram tokenizer, a query or phrase token may match any sequence of characters within a row, not just a complete token. For example:

```
CREATE VIRTUAL TABLE tri USING fts5(a, tokenize="trigram");
INSERT INTO tri VALUES('abcdefghij KLMNOPQRST uvwxyz');

-- The following queries all match the single row in the table
SELECT * FROM tri('cdefg');
SELECT * FROM tri('cdefg AND pqr');
SELECT * FROM tri('"hij klm" NOT stuv');
```

The trigram tokenizer supports the following options:

Option	Usage

case_sensitive	This value may be set to 1 or 0 (the default). If it is set to 1, then matching is case sensitive. Otherwise, if this option is set to 0, matching is case insensitive.
remove_diacritics	This value may also be set to 1 or 0 (the default). It may only be set to 1 if the case_sensitive options is set to 0 - setting both options to 1 is an error. If this option is set, then diacritics are removed from the text before matching (e.g. so that "á" matches "a").

```
-- A case-sensitive trigram index
CREATE VIRTUAL TABLE tri USING fts5(a, tokenize="trigram case_sensitive 1");
```

Unless the remove_diacritics option is set, FTS5 tables that use the trigram tokenizer also support indexed GLOB and LIKE pattern matching. For example:

```
SELECT * FROM tri WHERE a LIKE '%cdefg%';
SELECT * FROM tri WHERE a GLOB '*ij klm*xyz';
```

If an FTS5 trigram tokenizer is created with the case_sensitive option set to 1, it may only index GLOB queries, not LIKE.

Notes:

- Substrings consisting of fewer than 3 unicode characters do not match any rows when used with a full-text query. If a LIKE or GLOB pattern does not contain at least one sequence of non-wildcard unicode characters, FTS5 falls back to a linear scan of the entire table.
- If the FTS5 table is created with the detail=none or detail=column option specified, full-text queries may not contain any tokens longer than 3 unicode characters. LIKE and GLOB pattern matching may be slightly slower, but still works. If the index is to be used only for LIKE and/or GLOB pattern matching, these options are worth experimenting with to reduce the index size.
- The index cannot be used to optimize LIKE patterns if the LIKE operator has an ESCAPE clause.

4.4. External Content and Contentless Tables

Normally, when a row is inserted into an FTS5 table, in addition to building the index, FTS5 makes a copy of the original row content. When column values are requested from the FTS5 table by the user or by an auxiliary function implementation, those values are read from that private copy of the content. The "content" option may be used to create an FTS5 table that stores only FTS full-text index entries. Because the column values themselves are usually much larger than the associated full-text index entries, this can save significant database space.

There are two ways to use the "content" option:

- By setting it to an empty string to create a contentless FTS5 table. In this case FTS5 assumes that the original column values are unavailable to it when processing queries. Full-text queries and some auxiliary functions can still be used, but no column values apart from the rowid may be read from the table.
- By setting it to the name of a database object (table, virtual table or view) that may be queried by FTS5 at any time to retrieve the column values. This is known as an "external content" table. In this case all FTS5 functionality may be used, but it is the responsibility of the user to ensure that the contents of the full-text index are consistent with the named database object. If they are not, query results may be unpredictable.

4.4.1. Contentless Tables

A contentless FTS5 table is created by setting the "content" option to an empty string. For example:

```
CREATE VIRTUAL TABLE ft USING fts5(a, b, c, content='');
```

Contentless FTS5 tables do not support UPDATE or DELETE statements, or INSERT statements that do not supply a non-NULL value for the rowid field. Contentless tables do not support REPLACE conflict handling. REPLACE and INSERT OR REPLACE statements are treated as regular INSERT statements. Rows may be deleted from a contentless table using an FTS5 delete command.

Attempting to read any column value except the rowid from a contentless FTS5 table returns an SQL NULL value.

4.4.2. Contentless-Delete Tables

As of version 3.43.0, also available are contentless-delete tables. A contentless-delete table is created by setting the content option to an empty string and also setting the contentless_delete option to 1. For example:

```
CREATE VIRTUAL TABLE ft USING fts5(a, b, c, content='', contentless_delete=1);
```

A contentless-delete table differs from a contentless table in that:

- Contentless-delete tables support both DELETE and "INSERT OR REPLACE INTO" statements.
- Contentless-delete tables support UPDATE statements, but only if new values are supplied for all user-defined columns of the fts5 table.
- Contentless-delete tables do **not** support the FTS5 delete command.

```
-- Supported UPDATE statement:
UPDATE ft SET a=?, b=?, c=? WHERE rowid=?;

-- This UPDATE is not supported, as it does not supply a new value
-- for column "c".
UPDATE ft SET a=?, b=? WHERE rowid=?;
```

Unless backwards compatibility is required, new code should prefer contentless-delete tables to contentless tables.

4.4.3. External Content Tables

An external content FTS5 table is created by setting the content option to the name of a table, virtual table or view (hereafter the "content table") within the same database. Whenever column values are required by FTS5, it queries the content table as follows, with the rowid of the row for which values are required bound to the SQL variable:

```
SELECT <content_rowid>, <cols> FROM <content> WHERE <content_rowid> = ?;
```

In the above, <content> is replaced by the name of the content table. By default, <content_rowid> is replaced by the literal text "rowid". Or, if the "content_rowid" option is set within the CREATE VIRTUAL TABLE statement, by the value of that option. <cols> is replaced by a comma-separated list of the FTS5 table column names. For example:

```
-- If the database schema is:
CREATE TABLE t1 (a, b, c, d INTEGER PRIMARY KEY);
CREATE VIRTUAL TABLE ft USING fts5(a, c, content=t1, content_rowid=d);

-- Fts5 may issue queries such as:
SELECT d, a, c FROM t1 WHERE d = ?;
```

The content table may also be queried as follows:

```
SELECT <content_rowid>, <cols> FROM <content> ORDER BY <content_rowid> ASC;
SELECT <content_rowid>, <cols> FROM <content> ORDER BY <content_rowid> DESC;
```

It is still the responsibility of the user to ensure that the contents of an external content FTS5 table are kept up to date with the content table. One way to do this is with triggers. For example:

```
-- Create a table. And an external content fts5 table to index it.
CREATE TABLE t1(a INTEGER PRIMARY KEY, b, c);
CREATE VIRTUAL TABLE fts_idx USING fts5(b, c, content='t1', content_rowid='a');

-- Triggers to keep the FTS index up to date.
CREATE TRIGGER t1_ai AFTER INSERT ON t1 BEGIN
  INSERT INTO fts_idx(rowid, b, c) VALUES (new.a, new.b, new.c);
END;
CREATE TRIGGER t1_ad AFTER DELETE ON t1 BEGIN
  INSERT INTO fts_idx(fts_idx, rowid, b, c) VALUES('delete', old.a, old.b, old.c);
END;
CREATE TRIGGER t1_au AFTER UPDATE ON t1 BEGIN
  INSERT INTO fts_idx(fts_idx, rowid, b, c) VALUES('delete', old.a, old.b, old.c);
  INSERT INTO fts_idx(rowid, b, c) VALUES (new.a, new.b, new.c);
END;
```

Like contentless tables, external content tables do not support REPLACE conflict handling. Any operations that specify REPLACE conflict handling are handled using ABORT.

4.4.4. External Content Table Pitfalls

It is the responsibility of the user to ensure that an FTS5 external content table (one with a non-empty content= option) is kept consistent with the content table itself (the table named by the content= option). If these are allowed to become inconsistent, then the results of queries against the FTS5 table may become unintuitive and appear inconsistent.

In these situations, the apparently inconsistent results produced by queries against the FTS5 external content table may be understood as follows:

- If the query does not use the full-text index - does not contain a MATCH operator or equivalent table-valued function syntax - then the query is effectively passed through to the external content table. In this case the contents of the FTS index have no effect on the results of the query.
- If the query does use the full text index, then the FTS5 module queries it for the set of rowid values corresponding to documents that match the query. For each such rowid, it then runs a query similar to the following to retrieve any required column values, where '?' is replaced by the rowid value, and <content> and <content_rowid> by the values specified for the content= and content_rowid= options:

```
SELECT <content_rowid>, <cols> FROM <content> WHERE <content_rowid> = ?;
```

For example, if a database is created using the following script:

```
-- Create and populate a table.
CREATE TABLE t1(a INTEGER PRIMARY KEY, t TEXT);
INSERT INTO t1 VALUES(1, 'all that glitters');
INSERT INTO t1 VALUES(2, 'is not gold');

-- Create an external content FTS5 table
CREATE VIRTUAL TABLE ft USING fts5(t, content='t1', content_rowid='a');
```

then the content table contains two rows, but the FTS index contains no entries corresponding to them. In this case the following queries will return inconsistent results as follows:

```
-- Returns 2 rows.  Because the query does not use the FTS index, it is
-- effectively executed against table 't1' directly, and so returns
-- both rows.
SELECT * FROM ft;

-- Returns 0 rows.  This query does use the FTS index, which currently
-- contains no entries. So it returns 0 rows.
SELECT rowid, t FROM ft('gold')
```

Alternatively, if the database were created and populated as follows:

```
-- Create and populate a table.
CREATE TABLE t1(a INTEGER PRIMARY KEY, t TEXT);

-- Create an external content FTS5 table
CREATE VIRTUAL TABLE ft USING fts5(t, content='t1', content_rowid='a');
INSERT INTO ft(rowid, t) VALUES(1, 'all that glitters');
INSERT INTO ft(rowid, t) VALUES(2, 'is not gold');
```

then the content table is empty, but the FTS index contains entries for 6 different tokens. In this case the following queries will return inconsistent results as follows:

```
-- Returns 0 rows.  Since it does not use the FTS index, the query is
-- passed directly through to table 't1', which contains no data.
SELECT * FROM ft;

-- Returns 1 row. The "rowid" field of the returned row is 2, and
-- the "t" field set to NULL. "t" is set to NULL because when the external
-- content table "t1" was queried for the data associated with the row
-- with a=2 ("a" is the content_rowid column), none could be found.
SELECT rowid, t FROM ft('gold')
```

As described in the previous section, triggers on the content table are a good way to ensure that an FTS5 external content table is kept consistent. However, triggers are only fired when rows are inserted, updated or deleted in the content table. This means that if, for example, a database is created as follows:

```
-- Create and populate a table.
CREATE TABLE t1(a INTEGER PRIMARY KEY, t TEXT);
INSERT INTO t1 VALUES(1, 'all that glitters');
INSERT INTO t1 VALUES(2, 'is not gold');

-- Create an external content FTS5 table
CREATE VIRTUAL TABLE ft USING fts5(t, content='t1', content_rowid='a');

-- Create triggers to keep the FTS5 table up to date
CREATE TRIGGER t1_ai AFTER INSERT ON t1 BEGIN
  INSERT INTO ft(rowid, t) VALUES (new.a, new.t);
END;
<similar triggers for update + delete>
```

then the content table and external content FTS5 table are inconsistent, as creating the triggers does not copy existing rows from the content table into the FTS index. The triggers are only able to ensure that updates made to the content table after they are created are reflected in the FTS index.

In this, and any other situation where the FTS index and its content table have become inconsistent, the 'rebuild' command may be used to completely discard the contents of the FTS index and rebuild it based on the current contents of the content table.

4.5. The Columnsize Option

Normally, FTS5 maintains a special backing table within the database that stores the size of each column value in tokens inserted into the main FTS5 table in a separate table. This backing table is used by the xColumnSize API function, which is in turn used by the built-in bm25 ranking function (and is likely to be useful to other ranking functions as well).

In order to save space, this backing table may be omitted by setting the columnsize option to zero. For example:

```
-- A table without the xColumnSize() values stored on disk:
CREATE VIRTUAL TABLE ft USING fts5(a, b, c, columnsize=0);

-- Three equivalent ways of creating a table that does store the
-- xColumnSize() values on disk:
CREATE VIRTUAL TABLE ft USING fts5(a, b, c);
CREATE VIRTUAL TABLE ft USING fts5(a, b, c, columnsize=1);
CREATE VIRTUAL TABLE ft USING fts5(a, b, columnsize='1', c);
```

It is an error to set the columnsize option to any value other than 0 or 1.

If an FTS5 table is configured with columnsize=0 but is not a contentless table, the xColumnSize API function still works, but runs much more slowly. In this case, instead of reading the value to return directly from the database, it reads the text value itself and count the tokens within it on demand.

Or, if the table is also a contentless table, then the following apply:

- The xColumnSize API always returns -1. There is no way to determine the number of tokens in a value stored within a contentless FTS5 table configured with columnsize=0.
- Each inserted row must be accompanied by an explicitly specified rowid value. If a contentless table is configured with columnsize=0, attempting to insert a NULL value into the rowid is an SQLITE_MISMATCH error.
- All queries on the table must be full-text queries. In other words, they must use the MATCH or = operator with the table-name column as the left-hand operand, or else use the table-valued function syntax. Any query that is not a full-text query results in an error.

The name of the table in which the xColumnSize values are stored (unless columnsize=0 is specified) is "<name>_docsize", where <name> is the name of the FTS5 table itself. The sqlite3_analyzer tool may be used on an existing database in order to determine how much space might be saved by recreating an FTS5 table using columnsize=0.

4.6. The Detail Option

For each term in a document, the FTS index maintained by FTS5 stores the rowid of the document, the column number of the column that contains the term and the offset of the term within the column value. The "detail" option may be used to omit some of this information. This reduces the space that the index consumes within the database file, but also reduces the capability and efficiency of the system.

The detail option may be set to "full" (the default value), "column" or "none". For example:

```
-- The following two lines are equivalent (because the default value
-- of "detail" is "full".
CREATE VIRTUAL TABLE ft USING fts5(a, b, c);
CREATE VIRTUAL TABLE ft USING fts5(a, b, c, detail=full);

CREATE VIRTUAL TABLE ft USING fts5(a, b, c, detail=column);
CREATE VIRTUAL TABLE ft USING fts5(a, b, c, detail=none);
```

If the detail option is set to **column**, then for each term the FTS index records the rowid and column number only, omitting the term offset information. This results in the following restrictions:

- NEAR queries are not available.
- Phrase queries are not available.
- Assuming the table is not also a contentless table, the xInstCount, xInst, xPhraseFirst and xPhraseNext are slower than usual. This is because instead of reading the required data directly from the FTS index they have to load and tokenize the document text on demand.
- If the table is also a contentless table, the xInstCount, xInst, xPhraseFirst and xPhraseNext APIs behave as if the current row contains no phrase matches at all (i.e. xInstCount() returns 0).

If the detail option is set to **none**, then for each term the FTS index records just the rowid is stored. Both column and offset information are omitted. As well as the restrictions itemized above for detail=column mode, this imposes the following extra limitations:

- Column filter queries are not available.
- Assuming the table is not also a contentless table, the xPhraseFirstColumn and xPhraseNextColumn are slower than usual.
- If the table is also a contentless table, the xPhraseFirstColumn and xPhraseNextColumn APIs behave as if the current row contains no phrase matches at all (i.e. xPhraseFirstColumn() sets the iterator to EOF).

In one test that indexed a large set of emails (1636 MiB on disk), the FTS index was 743 MiB on disk with detail=full, 340 MiB with detail=column and 134 MiB with detail=none.

4.7. The Tokendata Option

This option is only useful to applications that implement custom tokenizers. Usually, tokenizers may return tokens that consist of any sequence of bytes, including 0x00 bytes. However, if the table specifies the tokendata=1 option, then fts5 ignores the first 0x00 byte and any trailing data in the token for the purposes of matching. It still stores the entire token as returned by the tokenizer, but it is ignored by the fts5 core.

The full version of the token, including any 0x00 byte and trailing data, is available to custom auxiliary functions via the xQueryToken and xInstToken APIs.

This may be useful for ranking functions. A custom tokenizer may add extra data to some document tokens allowing a ranking function to give more weight to hits of some tokens (e.g. those in document headings).

The combination of a custom tokenizer and a custom auxiliary function may be used to implement asymmetric search. The tokenizer could (say) for each document token return the case-normalized and unmarked version of the token, followed by an 0x00 byte, followed by the full text of the token from the document. When queried, fts5 would provide results as if all characters in the query were case-normalized and unmarked. The custom auxiliary function could then be used in the WHERE clause of the query to filter out any rows that do not match based on secondary or tertiary markings in the document or query terms.

4.8. The Locale Option

This option is only useful to applications that implement custom tokenizers. If an fts5 table is created with the "locale=1" option specified, then the fts5_locale() SQL function may be used to associate a locale value (e.g. "th_TH" or "en_US") with strings passed to FTS5. FTS5 itself does not use locale values, but makes them available to the tokenizer implementation whenever the string is tokenized. The tokenizer may then adjust its behaviour based on the locale.

```
-- The following statement creates an fts5 table with locale support.
-- The "tokenizer=..." option below must be replaced with a real tokenizer
-- specification for a tokenizer that supports locales.
CREATE VIRTUAL TABLE ft USING fts5(a, b, locale=1, tokenizer=...);

-- This statement inserts a row into the table. The value inserted into
-- column "a" uses locale "th_TH", the value written to column "b" uses the
```

```
-- tokenizer's default locale
INSERT INTO ft(a, b) VALUES(
     fts5_locale('th_TH', 'Tokenize this in Thai locale'),
     'Tokenize this in the default locale.'
);

-- The "en_US" locale is used to tokenize the query terms in the
--following query.
SELECT * FROM ft( fts5_locale('en_US', 'query terms') );
```

Attempting to pass an fts5_locale() string to an fts5 table that was not created with the locale=1 option is an error.

When an fts5_locale() string is stored in a normal content table (i.e. not a contentless or external content table), the attached locale is stored along with it. If the string is tokenized by FTS5 again, for example because its row is being deleted or as part of an auxiliary function evaluation, the attached locale is again passed to the tokenizer implementation.

In order to support locales, an FTS5 external-content table may use an SQL view that returns fts5_locale() values as the content table. For example:

```
-- Each row of this table contains a string and its locale.
CREATE TABLE t1(val, locale);
INSERT INTO t1 VALUES('a text value', 'en_US');

-- A view to combine the string and locale from table t1.
CREATE VIEW v1 AS SELECT rowid, fts5_locale(val, locale) AS val FROM t1;

-- An FTS5 table to read locale-enabled strings from view v1.
CREATE VIRTUAL TABLE ft USING fts5(val, locale=1, content=v1, tokenize=...);
```

If an fts5_locale() value is written to an UNINDEXED column of an fts5 table, the locale value is discarded and the string stored by itself.

The fts5_get_locale() function may be used to retreive the locale of a value stored in an FTS5 table.

4.9. The Contentless-Unindexed Option

Usually, UNINDEXED columns belonging to contentless tables are not very useful. Values written to them are not indexed or stored, and reading from such an UNINDEXED column always returns NULL. However, if the "contentless_unindexed=1" option is specified on a contentless table, then the values of UNINDEXED columns are stored persistently, even though values written to other columns are not.

```
-- Create a contentless table with the contentless_unindexed=1 option.
-- Of the row written to it, the value 'one' will be indexed and then
-- discarded, and the value "1" will be stored but not indexed.
CREATE VIRTUAL TABLE ft USING fts5(a, b UNINDEXED, content='', contentless_unindexed=1);
INSERT INTO ft(a, b) VALUES('one', 1);

-- This query returns 1 row with 2 columns - (NULL, 1). Reading from
-- column "a" is always NULL, as the table is contentless. But reading from
-- "b" returns the value, as the table uses contentless_unindexed=1.
SELECT a, b FROM ft('one');
```

It is an error to specify contentless_unindexed=1 for an fts5 table that is not a contentless or contentless-delete table.

5. Auxiliary Functions

Auxiliary functions are similar to SQL scalar functions, except that they may only be used within full-text queries (those that use the MATCH operator, or LIKE/GLOB with the trigram tokenizer) on an FTS5

table. Their results are calculated based not only on the arguments passed to them, but also on the current match and matched row. For example, an auxiliary function may return a numeric value indicating the accuracy of the match (see the bm25() function), or a fragment of text from the matched row that contains one or more instances of the search terms (see the snippet() function).

To invoke an auxiliary function, the name of the FTS5 table should be specified as the first argument. Other arguments may follow the first, depending on the specific auxiliary function being invoked. For example, to invoke the "highlight" function:

```
-- Assuming fts5 table:
CREATE VIRTUAL TABLE ft USING fts5(a, b, c);

-- Invoke the highlight() function:
SELECT highlight(ft, 2, '<b>', '</b>') FROM ft WHERE ft MATCH 'fts5'
```

The built-in auxiliary functions provided as part of FTS5 are described in the following section. Applications may also implement custom auxiliary functions in C.

5.1. Built-in Auxiliary Functions

FTS5 provides three built-in auxiliary functions:

- The bm25() auxiliary function returns a real value reflecting the accuracy of the current match. Better matches are assigned numerically lower values.
- The highlight() auxiliary function returns a copy of the text from one of the columns of the current match with each instance of a queried term within the result surrounded by specified markup (for example "" and "").
- The snippet() auxiliary function selects a short fragment of text from one of the columns of the matched row and returns it with each instance of a queried term surrounded by markup in the same manner as the highlight() function. The fragment of text is selected so as to maximize the number of distinct queried terms it contains. Higher weight is given to snippets that occur at the start of a column value, or that immediately follow "." or ":" characters in the text.
- The fts5_get_locale() auxiliary function is used to retrieve the locale, if any, associated with a value stored in an FTS5 table.

5.1.1. The bm25() function

The built-in auxiliary function bm25() returns a real value indicating how well the current row matches the full-text query. The better the match, the numerically smaller the value returned. A query such as the following may be used to return matches in order from best to worst match:

```
SELECT * FROM ft WHERE ft MATCH ? ORDER BY bm25(ft)
```

In order to calculate a documents score, the full-text query is separated into its component phrases. The bm25 score for document *D* and query *Q* is then calculated as follows:

$$bm25(D,Q) \;=\; -1 \sum_{i=1}^{nPhrase} IDF(q_i) \frac{f(q_i,D)\cdot(k_1+1)}{f(q_i,D)+k_1\cdot\left(1-b+b\dfrac{|D|}{avgdl}\right)}$$

In the above, *nPhrase* is the number of phrases in the query. |*D*| is the number of tokens in the current document, and *avgdl* is the average number of tokens in all documents within the FTS5 table. k_1 and *b* are both constants, hard-coded at 1.2 and 0.75 respectively.

The "-1" term at the start of the formula is not found in most implementations of the BM25 algorithm. Without it, a better match is assigned a numerically higher BM25 score. Since the default sorting order is "ascending", this means that appending "ORDER BY bm25(ft)" to a query would cause results to be returned in order from worst to best. The "DESC" keyword would be required in order to return the best matches first. In order to avoid this pitfall, the FTS5 implementation of BM25 multiplies the result by -1 before returning it, ensuring that better matches are assigned numerically lower scores.

IDF(q_i) is the inverse-document-frequency of query phrase *i*. It is calculated as follows, where *N* is the total number of rows in the FTS5 table and *n(q_i)* is the total number of rows that contain at least one instance of phrase *i*:

$$IDF(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5}$$

Finally, *f(q_i,D)* is the phrase frequency of phrase *i*. By default, this is simply the number of occurrences of the phrase within the current row. However, by passing extra real value arguments to the bm25() SQL function, each column of the table may be assigned a different weight and the phrase frequency calculated as follows:

$$f(q_i, D) = \sum_{c=1}^{nCol} w_c \cdot n(q_i, c)$$

where w_c is the weight assigned to column *c* and *n(q_i,c)* is the number of occurrences of phrase *i* in column *c* of the current row. The first argument passed to bm25() following the table name is the weight assigned to the leftmost column of the FTS5 table. The second is the weight assigned to the second leftmost column, and so on. If there are not enough arguments for all table columns, remaining columns are assigned a weight of 1.0. If there are too many trailing arguments, the extras are ignored. For example:

```
-- Assuming the following schema:
CREATE VIRTUAL TABLE email USING fts5(sender, title, body);

-- Return results in bm25 order, with each phrase hit in the "sender"
-- column considered the equal of 10 hits in the "body" column, and
-- each hit in the "title" column considered as valuable as 5 hits in
-- the "body" column.
SELECT * FROM email WHERE email MATCH ? ORDER BY bm25(email, 10.0, 5.0);
```

Refer to wikipedia for more information regarding BM25 and its variants.

5.1.2. The highlight() function

The highlight() function returns a copy of the text from a specified column of the current row with extra markup text inserted to mark the start and end of phrase matches.

The highlight() must be invoked with exactly three arguments following the table name. To be interpreted as follows:

1. An integer indicating the index of the FTS table column to read the text from. Columns are numbered from left to right starting at zero.
2. The text to insert before each phrase match.
3. The text to insert after each phrase match.

For example:

```
-- Return a copy of the text from the leftmost column of the current
-- row, with phrase matches marked using html "b" tags.
SELECT highlight(ft, 0, '<b>', '</b>') FROM ft WHERE ft MATCH ?
```

In cases where two or more phrase instances overlap (share one or more tokens in common), a single open and close marker is inserted for each set of overlapping phrases. For example:

```
-- Assuming this:
CREATE VIRTUAL TABLE ft USING fts5(a);
INSERT INTO ft VALUES('a b c x c d e');
INSERT INTO ft VALUES('a b c c d e');
INSERT INTO ft VALUES('a b c d e');

-- The following SELECT statement returns these three rows:
--   '[a b c] x [c d e]'
--   '[a b c] [c d e]'
--   '[a b c d e]'
SELECT highlight(ft, 0, '[', ']') FROM ft WHERE ft MATCH 'a+b+c AND c+d+e';
```

5.1.3. The snippet() function

The snippet() function is similar to highlight(), except that instead of returning entire column values, it automatically selects and extracts a short fragment of document text to process and return. The snippet() function must be passed five parameters following the table name argument:

1. An integer indicating the index of the FTS table column to select the returned text from. Columns are numbered from left to right starting at zero. A negative value indicates that the column should be automatically selected.
2. The text to insert before each phrase match within the returned text.
3. The text to insert after each phrase match within the returned text.
4. The text to add to the start or end of the selected text to indicate that the returned text does not occur at the start or end of its column, respectively.
5. The maximum number of tokens in the returned text. This must be greater than zero and equal to or less than 64.

5.1.4. The fts5_get_locale() function

The fts5_get_locale() function is used to retrieve the locale, if any, associated with a value stored in an FTS5 table. It accepts a single argument following the table name, the index of the column of the current row to query. Columns are numbered in the order they appeared in the CREATE VIRTUAL TABLE statement starting from 0.

If the FTS5 table does not support locales (i.e. was not created with the locale=1 option), or if there is no locale associated with the nominated value, then this function returns NULL. Otherwise it returns a text value, the name of the locale that the value in question is associated with.

```
CREATE VIRTUAL TABLE ft USING fts5(a, b, c, locale=1);
INSERT INTO ft VALUES(
    'no locale',
    fts5_locale('th_TH', 'Thai locale'),
    fts5_locale('en_US', 'US locale')
);

-- The following statement returns a single row containing three values:
-- NULL, text value 'th_TH', and text value 'en_US'.
SELECT
    fts5_get_locale(ft, 0),
    fts5_get_locale(ft, 1),
    fts5_get_locale(ft, 2)
FROM ft;
```

5.2. Sorting by Auxiliary Function Results

All FTS5 tables feature a special hidden column named "rank". If the current query is not a full-text query (i.e. if it does not include a MATCH operator), the value of the "rank" column is always NULL. Otherwise, in a full-text query, column rank contains by default the same value as would be returned by executing the bm25() auxiliary function with no trailing arguments.

The difference between reading from the rank column and using the bm25() function directly within the query is only significant when sorting by the returned value. In this case, using "rank" is faster than using bm25().

```
-- The following queries are logically equivalent. But the second may
-- be faster, particularly if the caller abandons the query before
-- all rows have been returned (or if the queries were modified to
-- include LIMIT clauses).
SELECT * FROM ft WHERE ft MATCH ? ORDER BY bm25(ft);
SELECT * FROM ft WHERE ft MATCH ? ORDER BY rank;
```

Instead of using bm25() with no trailing arguments, the specific auxiliary function mapped to the rank column may be configured either on a per-query basis, or by setting a different persistent default for the FTS table.

In order to change the mapping of the rank column for a single query, a term similar to either of the following is added to the WHERE clause of a query:

```
rank MATCH 'auxiliary-function-name(arg1, arg2, ...)'
rank = 'auxiliary-function-name(arg1, arg2, ...)'
```

The right-hand-side of the MATCH or = operator must be a constant expression that evaluates to a string consisting of the auxiliary function to invoke, followed by zero or more comma separated arguments within parenthesis. Arguments must be SQL literals. For example:

```
-- The following queries are logically equivalent. But the second may
-- be faster. See above.
SELECT * FROM ft WHERE ft MATCH ? ORDER BY bm25(ft, 10.0, 5.0);
SELECT * FROM ft WHERE ft MATCH ? AND rank MATCH 'bm25(10.0, 5.0)' ORDER BY rank;
```

The table-valued function syntax may also be used to specify an alternative ranking function. In this case the text describing the ranking function should be specified as the second table-valued function argument. The following three queries are equivalent:

```
SELECT * FROM ft WHERE ft MATCH ? AND rank MATCH 'bm25(10.0, 5.0)' ORDER BY rank;
SELECT * FROM ft WHERE ft = ? AND rank = 'bm25(10.0, 5.0)' ORDER BY rank;
SELECT * FROM ft WHERE ft(?, 'bm25(10.0, 5.0)') ORDER BY rank;
```

The default mapping of the rank column for a table may be modified using the FTS5 rank configuration option.

6. Special INSERT Commands

6.1. The 'automerge' Configuration Option

Instead of using a single data structure on disk to store the full-text index, FTS5 uses a series of b-trees. Each time a new transaction is committed, a new b-tree containing the contents of the committed transaction is written into the database file. When the full-text index is queried, each b-tree must be queried individually and the results merged before being returned to the user.

In order to prevent the number of b-trees in the database from becoming too large (slowing down queries), smaller b-trees are periodically merged into single larger b-trees containing the same data. By default, this happens automatically within INSERT, UPDATE or DELETE statements that modify the full-text index. The 'automerge' parameter determines how many smaller b-trees are merged together at a time. Setting it to a small value can speed up queries (as they have to query and merge the results from fewer b-trees), but can also slow down writing to the database (as each INSERT, UPDATE or DELETE statement has to do more work as part of the automatic merging process).

Each of the b-trees that make up the full-text index is assigned to a "level" based on its size. Level-0 b-trees are the smallest, as they contain the contents of a single transaction. Higher level b-trees are the result of merging two or more level-0 b-trees together and so they are larger. FTS5 begins to merge b-trees together once there exist *M* or more b-trees with the same level, where *M* is the value of the 'automerge' parameter.

The maximum allowed value for the 'automerge' parameter is 16. The default value is 4. Setting the 'automerge' parameter to 0 disables the automatic incremental merging of b-trees altogether.

```
INSERT INTO ft(ft, rank) VALUES('automerge', 8);
```

6.2. The 'crisismerge' Configuration Option

The 'crisismerge' option is similar to 'automerge', in that it determines how and how often the component b-trees that make up the full-text index are merged together. Once there exist *C* or more b-trees on a single level within the full-text index, where *C* is the value of the 'crisismerge' option, all b-trees on the level are immediately merged into a single b-tree.

The difference between this option and the 'automerge' option is that when the 'automerge' limit is reached FTS5 only begins to merge the b-trees together. Most of the work is performed as part of subsequent INSERT, UPDATE or DELETE operations. Whereas when the 'crisismerge' limit is reached, the offending b-trees are all merged immediately. This means that an INSERT, UPDATE or DELETE that triggers a crisis-merge may take a long time to complete.

The default 'crisismerge' value is 16. There is no maximum limit. Attempting to set the 'crisismerge' parameter to a value of 0 or 1 is equivalent to setting it to the default value (16). It is an error to attempt to set the 'crisismerge' option to a negative value.

```
INSERT INTO ft(ft, rank) VALUES('crisismerge', 16);
```

6.3. The 'delete' Command

This command is only available with external content and contentless tables. It is used to delete the index entries associated with a single row from the full-text index. This command and the delete-all command are the only ways to remove entries from the full-text index of a contentless table.

In order to use this command to delete a row, the text value 'delete' must be inserted into the special column with the same name as the table. The rowid of the row to delete is inserted into the rowid column. The values inserted into the other columns must match the values currently stored in the table. For example:

```
-- Insert a row with rowid=14 into the fts5 table.
INSERT INTO ft(rowid, a, b, c) VALUES(14, $a, $b, $c);

-- Remove the same row from the fts5 table.
INSERT INTO ft(ft, rowid, a, b, c) VALUES('delete', 14, $a, $b, $c);
```

If the values "inserted" into the text columns as part of a 'delete' command are not the same as those currently stored within the table, the results may be unpredictable.

The reason for this is easy to understand: When a document is inserted into the FTS5 table, an entry is added to the full-text index to record the position of each token within the new document. When a document is removed, the original data is required in order to determine the set of entries that need to be removed from the full-text index. So if the data supplied to FTS5 when a row is deleted using this command is different from that used to determine the set of token instances when it was inserted, some full-text index entries may not be correctly deleted, or FTS5 may try to remove index entries that do not exist. This can leave the full-text index in an unpredictable state, making future query results unreliable.

6.4. The 'delete-all' Command

This command is only available with external content and contentless tables (including contentless-delete tables). It deletes all entries from the full-text index.

```
INSERT INTO ft(ft) VALUES('delete-all');
```

6.5. The 'deletemerge' Configuration Option

The 'deletemerge' option is only used by contentless-delete tables.

When a row is deleted from a contentless-delete table, the entries associated with its tokens are not immediately removed from the FTS index. Instead, a "tombstone" marker containing the rowid of the deleted row is attached to the b-tree that contains the row's FTS index entries. When the b-tree is queried, any query result rows for which there exist tombstone markers are omitted from the results. When the b-tree is merged with other b-trees, both the deleted rows and their tombstone markers are discarded.

This option specifies a minimum percentage of rows in a b-tree that must have tombstone markers before the b-tree is made eligible for merging - either by automatic merges or explicit user 'merge' commands - even if it does not meet the usual criteria as determined by the 'automerge' and 'usermerge' options.

For example, to specify that FTS5 should consider merging a component b-tree after 15% of its rows have associated tombstone markers:

```
INSERT INTO ft(ft, rank) VALUES('deletemerge', 15);
```

The default value of this option is 10. Attempting to set it to less than zero restores the default value. Setting this option to 0 or to greater than 100 ensures that b-trees are never made eligible for merging due to tombstone markers.

6.6. The 'integrity-check' Command

This command is used to verify that the full-text index is internally consistent, and, optionally, that it is consistent with any external content table.

The integrity-check command is invoked by inserting the text value 'integrity-check' into the special column with the same name as the FTS5 table. If a value is supplied for the "rank" column, it must be either 0 or 1. For example:

```
INSERT INTO ft(ft) VALUES('integrity-check');
INSERT INTO ft(ft, rank) VALUES('integrity-check', 0);
INSERT INTO ft(ft, rank) VALUES('integrity-check', 1);
```

The three forms above are equivalent for all FTS tables that are not external content tables. They check that the index data structures are not corrupt, and, if the FTS table is not contentless, that the

contents of the index match the contents of the table itself.

For an external content table, the contents of the index are only compared to the contents of the external content table if the value specified for the rank column is 1.

In all cases, if any discrepancies are found, the command fails with an SQLITE_CORRUPT_VTAB error.

6.7. The 'merge' Command

```
INSERT INTO ft(ft, rank) VALUES('merge', 500);
```

This command merges b-tree structures together until roughly N pages of merged data have been written to the database, where N is the absolute value of the parameter specified as part of the 'merge' command. The size of each page is as configured by the FTS5 pgsz option.

If the parameter is a positive value, B-tree structures are only eligible for merging if one of the following is true:

- There are U or more such b-trees on a single level (see the documentation for the FTS5 automerge option for an explanation of b-tree levels), where U is the value assigned to the FTS5 usermerge option option.
- A merge has already been started (perhaps by a 'merge' command that specified a negative parameter).

It is possible to tell whether or not the 'merge' command found any b-trees to merge together by checking the value returned by the sqlite3_total_changes() API before and after the command is executed. If the difference between the two values is 2 or greater, then work was performed. If the difference is less than 2, then the 'merge' command was a no-op. In this case there is no reason to execute the same 'merge' command again, at least until after the FTS table is next updated.

If the parameter is negative, and there are B-tree structures on more than one level within the FTS index, all B-tree structures are assigned to the same level before the merge operation is commenced. Additionally, if the parameter is negative, the value of the usermerge configuration option is not respected - as few as two b-trees from the same level may be merged together.

The above means that executing the 'merge' command with a negative parameter until the before and after difference in the return value of sqlite3_total_changes() is less than two optimizes the FTS index in the same way as the FTS5 optimize command. However, if a new b-tree is added to the FTS index while this process is ongoing, FTS5 will move the new b-tree to the same level as the existing b-trees and restart the merge. To avoid this, only the first call to 'merge' should specify a negative parameter. Each subsequent call to 'merge' should specify a positive value so that the merge started by the first call is run to completion even if new b-trees are added to the FTS index.

6.8. The 'optimize' Command

This command merges all individual b-trees that currently make up the full-text index into a single large b-tree structure. This ensures that the full-text index consumes the minimum space within the database and is in the fastest form to query.

Refer to the documentation for the FTS5 automerge option for more details regarding the relationship between the full-text index and its component b-trees.

```
INSERT INTO ft(ft) VALUES('optimize');
```

Because it reorganizes the entire FTS index, the optimize command can take a long time to run. The FTS5 merge command can be used to divide the work of optimizing the FTS index into multiple steps. To do this:

- Invoke the 'merge' command once with the parameter set to -N, then
- Invoke the 'merge' command zero or more times with the parameter set to N.

where N is the number of pages of data to merge within each invocation of the merge command. The application should stop invoking merge when the difference in the value returned by the sqlite3_total_changes() function before and after the merge command drops to below two. The merge commands may be issued as part of the same or separate transactions, and by the same or different database clients. Refer to the documentation for the merge command for further details.

6.9. The 'pgsz' Configuration Option

This command is used to set the persistent "pgsz" option.

The full-text index maintained by FTS5 is stored as a series of fixed-size blobs in a database table. It is not strictly necessary for all blobs that make up a full-text index to be the same size. The pgsz option determines the size of all blobs created by subsequent index writers. The default value is 4050.

```
INSERT INTO ft(ft, rank) VALUES('pgsz', 4072);
```

6.10. The 'rank' Configuration Option

This command is used to set the persistent "rank" option.

The rank option is used to change the default auxiliary function mapping for the rank column. The option should be set to a text value in the same format as described for "rank MATCH ?" terms above. For example:

```
INSERT INTO ft(ft, rank) VALUES('rank', 'bm25(10.0, 5.0)');
```

6.11. The 'rebuild' Command

This command first deletes the entire full-text index, then rebuilds it based on the contents of the table or content table. It is not available with contentless tables.

```
INSERT INTO ft(ft) VALUES('rebuild');
```

6.12. The 'secure-delete' Configuration Option

This command is used to set the persistent boolean "secure-delete" option. For example:

```
INSERT INTO ft(ft, rank) VALUES('secure-delete', 1);
```

Normally, when an entry in an fts5 table is updated or deleted, instead of removing entries from the full-text index, delete-keys are added to the new b-tree created by the transaction. This is efficient, but it means that the old full-text index entries remain in the database file until they are eventually removed by merge operations on the full-text index. Anyone with access to the database can use these entries to trivially reconstruct the contents of deleted FTS5 table rows. However, if the 'secure-delete' option is set to 1, then full-text entries are actually removed from the database when existing FTS5 table rows are updated or deleted. This is slower, but it prevents old full-text entries from being used to reconstruct deleted table rows.

This option ensures that old full-text entries are not available to attackers with SQL access to the database. To also ensure that they may not be recovered by attackers with access to the SQLite

database file itself, the application must also enable the SQLite core secure-delete option with a command like "PRAGMA secure_delete = 1".

Warning: Once one or more table rows have been updated or deleted with this option set, the FTS5 table may no longer be read or written by any version of FTS5 earlier than 3.42.0 (the first version in which this option was available). Attempting to do so results in an error, with an error message like "invalid fts5 file format (found 5, expected 4) - run 'rebuild'". The FTS5 file format may be reverted, so that it may be read by earlier versions of FTS5, by running the 'rebuild' command on the table using version 3.42.0 or later.

The default value of the secure-delete option is 0.

6.13. The 'usermerge' Configuration Option

This command is used to set the persistent "usermerge" option.

The usermerge option is similar to the automerge and crisismerge options. It is the minimum number of b-tree segments that will be merged together by a 'merge' command with a positive parameter. For example:

```
INSERT INTO ft(ft, rank) VALUES('usermerge', 4);
```

The default value of the usermerge option is 4. The minimum allowed value is 2, and the maximum 16.

7. Extending FTS5

FTS5 features APIs allowing it to be extended by:

- Adding new auxiliary functions implemented in C, and
- Adding new tokenizers, also implemented in C.

The built-in tokenizers and auxiliary functions described in this document are all implemented using the publicly available API described below.

Before a new auxiliary function or tokenizer implementation may be registered with FTS5, an application must obtain a pointer to the "fts5_api" structure. There is one fts5_api structure for each database connection with which the FTS5 extension is registered. To obtain the pointer, the application invokes the SQL user-defined function fts5() with a single argument. That argument must be set to a pointer to a pointer to an fts5_api object using the sqlite3_bind_pointer() interface. The following example code demonstrates the technique:

```
/*
** Return a pointer to the fts5_api pointer for database connection db.
** If an error occurs, return NULL and leave an error in the database
** handle (accessible using sqlite3_errcode()/errmsg()).
*/
fts5_api *fts5_api_from_db(sqlite3 *db){
  fts5_api *pRet = 0;
  sqlite3_stmt *pStmt = 0;

  if( SQLITE_OK==sqlite3_prepare(db, "SELECT fts5(?1)", -1, &pStmt, 0) ){
    sqlite3_bind_pointer(pStmt, 1, (void*)&pRet, "fts5_api_ptr", NULL);
    sqlite3_step(pStmt);
  }
  sqlite3_finalize(pStmt);
  return pRet;
}
```

Backwards Compatibility Warning: Prior to SQLite version 3.20.0 (2017-08-01), the fts5() worked slightly differently. Older applications that extend FTS5 must be revised to use the new technique

shown above.

The fts5_api structure is defined as follows. It exposes five methods:

- xCreateTokenizer() and xCreateTokenizer_v2(), for registering new custom tokenizer implementations.
- xFindTokenizer() and xFindTokenizer_v2(), for retrieving existing tokenizer implementations. This can be useful for implementing "tokenizer wrappers", similar to the built-in porter tokenizer.
- xCreateFunction(), for registering new auxiliary function implementations.

The two "v2" methods above are only available if the fts5_api.iVersion field is set to 3 or greater. Attempting to access the "v2" APIs via an fts5_api object with a lower value for iVersion results in undefined behaviour.

```
typedef struct fts5_api fts5_api;
struct fts5_api {
  int iVersion;                   /* Currently always set to 3 */

  /* Create a new tokenizer */
  int (*xCreateTokenizer)(
    fts5_api *pApi,
    const char *zName,
    void *pUserData,
    fts5_tokenizer *pTokenizer,
    void (*xDestroy)(void*)
  );

  /* Find an existing tokenizer */
  int (*xFindTokenizer)(
    fts5_api *pApi,
    const char *zName,
    void **ppUserData,
    fts5_tokenizer *pTokenizer
  );

  /* Create a new auxiliary function */
  int (*xCreateFunction)(
    fts5_api *pApi,
    const char *zName,
    void *pUserData,
    fts5_extension_function xFunction,
    void (*xDestroy)(void*)
  );

  /* APIs below this point are only available if iVersion>=3 */

  /* Create a new tokenizer */
  int (*xCreateTokenizer_v2)(
    fts5_api *pApi,
    const char *zName,
    void *pUserData,
    fts5_tokenizer_v2 *pTokenizer,
    void (*xDestroy)(void*)
  );

  /* Find an existing tokenizer */
  int (*xFindTokenizer_v2)(
    fts5_api *pApi,
    const char *zName,
    void **ppUserData,
    fts5_tokenizer_v2 **ppTokenizer
  );
};
```

To invoke a method of the fts5_api object, the fts5_api pointer itself should be passed as the methods first argument followed by the other, method specific, arguments. For example:

```
rc = pFts5Api->xCreateTokenizer(pFts5Api, ... other args ...);
```

The fts5_api structure methods are described individually in the following sections.

7.1. Custom Tokenizers

To create a custom tokenizer, an application must implement three functions: a tokenizer constructor (xCreate), a destructor (xDelete) and a function to do the actual tokenization (xTokenize). The type of each function is as for the member variables of the fts5_tokenizer_v2 struct:

```
typedef struct Fts5Tokenizer Fts5Tokenizer;
typedef struct fts5_tokenizer_v2 fts5_tokenizer_v2;
struct fts5_tokenizer_v2 {
  int iVersion;             /* Currently always 2 */

  int (*xCreate)(void*, const char **azArg, int nArg, Fts5Tokenizer **ppOut);
  void (*xDelete)(Fts5Tokenizer*);
  int (*xTokenize)(Fts5Tokenizer*,
      void *pCtx,
      int flags,            /* Mask of FTS5_TOKENIZE_* flags */
      const char *pText, int nText,
      const char *pLocale, int nLocale,
      int (*xToken)(
        void *pCtx,         /* Copy of 2nd argument to xTokenize() */
        int tflags,         /* Mask of FTS5_TOKEN_* flags */
        const char *pToken, /* Pointer to buffer containing token */
        int nToken,         /* Size of token in bytes */
        int iStart,         /* Byte offset of token within input text */
        int iEnd            /* Byte offset of end of token within input text */
      )
  );
};

/* Flags that may be passed as the third argument to xTokenize() */
#define FTS5_TOKENIZE_QUERY     0x0001
#define FTS5_TOKENIZE_PREFIX    0x0002
#define FTS5_TOKENIZE_DOCUMENT  0x0004
#define FTS5_TOKENIZE_AUX       0x0008

/* Flags that may be passed by the tokenizer implementation back to FTS5
** as the third argument to the supplied xToken callback. */
#define FTS5_TOKEN_COLOCATED    0x0001      /* Same position as prev. token */
```

The implementation is registered with the FTS5 module by populating an instance of the fts5_tokenizer_v2 struct and passing a pointer to it to the xCreateTokenizer_v2() method of the fts5_api object. If there is already a tokenizer with the same name, it is replaced. If a non-NULL xDestroy parameter is passed to xCreateTokenizer(), it is invoked with a copy of the pUserData pointer passed as the only argument when the database handle is closed or when the tokenizer is replaced.

If successful, xCreateTokenizer() returns SQLITE_OK. Otherwise, it returns an SQLite error code. In this case the xDestroy function is **not** invoked.

When an FTS5 table uses the custom tokenizer, the FTS5 core calls xCreate() once to create a tokenizer, then xTokenize() zero or more times to tokenize strings, then xDelete() to free any resources allocated by xCreate(). More specifically:

xCreate:

> This function is used to allocate and initialize a tokenizer instance. A tokenizer instance is required to actually tokenize text.
>
> The first argument passed to this function is a copy of the (void*) pointer provided by the application when the fts5_tokenizer_v2 object was registered with FTS5 (the third argument to xCreateTokenizer()). The second and third arguments are an array of nul-terminated strings

containing the tokenizer arguments, if any, specified following the tokenizer name as part of the CREATE VIRTUAL TABLE statement used to create the FTS5 table.

The final argument is an output variable. If successful, (*ppOut) should be set to point to the new tokenizer handle and SQLITE_OK returned. If an error occurs, some value other than SQLITE_OK should be returned. In this case, fts5 assumes that the final value of *ppOut is undefined.

xDelete:

This function is invoked to delete a tokenizer handle previously allocated using xCreate(). Fts5 guarantees that this function will be invoked exactly once for each successful call to xCreate().

xTokenize:

This function is expected to tokenize the nText byte string indicated by argument pText. pText may or may not be nul-terminated. The first argument passed to this function is a pointer to an Fts5Tokenizer object returned by an earlier call to xCreate().

The third argument indicates the reason that FTS5 is requesting tokenization of the supplied text. This is always one of the following four values:

- **FTS5_TOKENIZE_DOCUMENT** - A document is being inserted into or removed from the FTS table. The tokenizer is being invoked to determine the set of tokens to add to (or delete from) the FTS index.
- **FTS5_TOKENIZE_QUERY** - A MATCH query is being executed against the FTS index. The tokenizer is being called to tokenize a bareword or quoted string specified as part of the query.
- **(FTS5_TOKENIZE_QUERY | FTS5_TOKENIZE_PREFIX)** - Same as FTS5_TOKENIZE_QUERY, except that the bareword or quoted string is followed by a "*" character, indicating that the last token returned by the tokenizer will be treated as a token prefix.
- **FTS5_TOKENIZE_AUX** - The tokenizer is being invoked to satisfy an fts5_api.xTokenize() request made by an auxiliary function. Or an fts5_api.xColumnSize() request made by the same on a columnsize=0 database.

The sixth and seventh arguments passed to xTokenize() - pLocale and nLocale - are a pointer to a buffer containing the locale to use for tokenization (e.g. "en_US") and its size in bytes, respectively. The pLocale buffer is not nul-terminated. pLocale may be passed NULL (in which case nLocale is always 0) to indicate that the tokenizer should use its default locale.

For each token in the input string, the supplied callback xToken() must be invoked. The first argument to it should be a copy of the pointer passed as the second argument to xTokenize(). The third and fourth arguments are a pointer to a buffer containing the token text, and the size of the token in bytes. The 4th and 5th arguments are the byte offsets of the first byte of and first byte immediately following the text from which the token is derived within the input.

The second argument passed to the xToken() callback ("tflags") should normally be set to 0. The exception is if the tokenizer supports synonyms. In this case see the discussion below for details.

FTS5 assumes the xToken() callback is invoked for each token in the order that they occur within the input text.

If an xToken() callback returns any value other than SQLITE_OK, then the tokenization should be abandoned and the xTokenize() method should immediately return a copy of the xToken() return value. Or, if the input buffer is exhausted, xTokenize() should return SQLITE_OK. Finally, if an error occurs with the xTokenize() implementation itself, it may abandon the tokenization and return any error code other than SQLITE_OK or SQLITE_DONE.

If the tokenizer is registered using an fts5_tokenizer_v2 object, then the xTokenize() method has two additional arguments - pLocale and nLocale. These specify the locale that the tokenizer

should use for the current request. If pLocale and nLocale are both 0, then the tokenizer should use its default locale. Otherwise, pLocale points to an nLocale byte buffer containing the name of the locale to use as utf-8 text. pLocale is not nul-terminated.

There is also an fts5_tokenizer object. This is an older, deprecated, version of fts5_tokenizer_v2. It is similar except that:

- There is no "iVersion" field, and
- The xTokenize() method does not take a locale argument.

Legacy fts5_tokenizer tokenizers must be registered using the legacy xCreateTokenizer() function, instead of xCreateTokenizer_v2().

Tokenizer implementations registered using either API may be retrieved using both xFindTokenizer() and xFindTokenizer_v2().

7.1.1. Synonym Support

Custom tokenizers may also support synonyms. Consider a case in which a user wishes to query for a phrase such as "first place". Using the built-in tokenizers, the FTS5 query 'first + place' will match instances of "first place" within the document set, but not alternative forms such as "1st place". In some applications, it would be better to match all instances of "first place" or "1st place" regardless of which form the user specified in the MATCH query text.

There are several ways to approach this in FTS5:

1. By mapping all synonyms to a single token. In this case, using the above example, this means that the tokenizer returns the same token for inputs "first" and "1st". Say that token is in fact "first", so that when the user inserts the document "I won 1st place" entries are added to the index for tokens "i", "won", "first" and "place". If the user then queries for '1st + place', the tokenizer substitutes "first" for "1st" and the query works as expected.

2. By querying the index for all synonyms of each query term separately. In this case, when tokenizing query text, the tokenizer may provide multiple synonyms for a single term within the document. FTS5 then queries the index for each synonym individually. For example, faced with the query:

   ```
   ... MATCH 'first place'
   ```

   the tokenizer offers both "1st" and "first" as synonyms for the first token in the MATCH query and FTS5 effectively runs a query similar to:

   ```
   ... MATCH '(first OR 1st) place'
   ```

   except that, for the purposes of auxiliary functions, the query still appears to contain just two phrases - "(first OR 1st)" being treated as a single phrase.

3. By adding multiple synonyms for a single term to the FTS index. Using this method, when tokenizing document text, the tokenizer provides multiple synonyms for each token. So that when a document such as "I won first place" is tokenized, entries are added to the FTS index for "i", "won", "first", "1st" and "place".

   This way, even if the tokenizer does not provide synonyms when tokenizing query text (it should not - to do so would be inefficient), it doesn't matter if the user queries for 'first + place' or '1st + place', as there are entries in the FTS index corresponding to both forms of the first token.

Whether it is parsing document or query text, any call to xToken that specifies a *tflags* argument with the FTS5_TOKEN_COLOCATED bit is considered to supply a synonym for the previous token. For

example, when parsing the document "I won first place", a tokenizer that supports synonyms would call xToken() 5 times, as follows:

```
xToken(pCtx, 0, "i",                      1,  0,  1);
xToken(pCtx, 0, "won",                    3,  2,  5);
xToken(pCtx, 0, "first",                  5,  6, 11);
xToken(pCtx, FTS5_TOKEN_COLOCATED, "1st", 3,  6, 11);
xToken(pCtx, 0, "place",                  5, 12, 17);
```

It is an error to specify the FTS5_TOKEN_COLOCATED flag the first time xToken() is called. Multiple synonyms may be specified for a single token by making multiple calls to xToken(FTS5_TOKEN_COLOCATED) in sequence. There is no limit to the number of synonyms that may be provided for a single token.

In many cases, method (1) above is the best approach. It does not add extra data to the FTS index or require FTS5 to query for multiple terms, so it is efficient in terms of disk space and query speed. However, it does not support prefix queries very well. If, as suggested above, the token "first" is substituted for "1st" by the tokenizer, then the query:

```
... MATCH '1s*'
```

will not match documents that contain the token "1st" (as the tokenizer will probably not map "1s" to any prefix of "first").

For full prefix support, method (3) may be preferred. In this case, because the index contains entries for both "first" and "1st", prefix queries such as 'fi*' or '1s*' will match correctly. However, because extra entries are added to the FTS index, this method uses more space within the database.

Method (2) offers a midpoint between (1) and (3). Using this method, a query such as '1s*' will match documents that contain the literal token "1st", but not "first" (assuming the tokenizer is not able to provide synonyms for prefixes). However, a non-prefix query like '1st' will match against "1st" and "first". This method does not require extra disk space, as no extra entries are added to the FTS index. On the other hand, it may require more CPU cycles to run MATCH queries, as separate queries of the FTS index are required for each synonym.

When using methods (2) or (3), it is important that the tokenizer only provide synonyms when tokenizing document text (method (3)) or query text (method (2)), not both. Doing so will not cause any errors, but is inefficient.

7.2. Custom Auxiliary Functions

Implementing a custom auxiliary function is similar to implementing a scalar SQL function. The implementation should be a C function of type fts5_extension_function, defined as follows:

```
typedef struct Fts5ExtensionApi Fts5ExtensionApi;
typedef struct Fts5Context Fts5Context;
typedef struct Fts5PhraseIter Fts5PhraseIter;

typedef void (*fts5_extension_function)(
  const Fts5ExtensionApi *pApi,   /* API offered by current FTS version */
  Fts5Context *pFts,              /* First arg to pass to pApi functions */
  sqlite3_context *pCtx,          /* Context for returning result/error */
  int nVal,                       /* Number of values in apVal[] array */
  sqlite3_value **apVal           /* Array of trailing arguments */
);
```

The implementation is registered with the FTS5 module by calling the xCreateFunction() method of the fts5_api object. If there is already an auxiliary function with the same name, it is replaced by the new function. If a non-NULL xDestroy parameter is passed to xCreateFunction(), it is invoked with a copy of

the pUserData pointer passed as the only argument when the database handle is closed or when the registered auxiliary function is replaced.

If successful, xCreateFunction() returns SQLITE_OK. Otherwise, it returns an SQLite error code. In this case the xDestroy function is **not** invoked.

The final three arguments passed to the auxiliary function callback (pCtx, nVal and apVal above) are similar to the three arguments passed to the implementation of a scalar SQL function. The apVal[] array contains all SQL arguments except the first passed to the auxiliary function. The implementation should return a result or error via the content handle pCtx.

The first argument passed to an auxiliary function callback is a pointer to a structure (pApi above) containing methods that may be invoked in order to obtain information regarding the current query or row. The second argument is an opaque handle (pFts above) that should be passed as the first argument to any such method invocation. For example, the following auxiliary function returns the total number of tokens in all columns of the current row:

```
/*
** Implementation of an auxiliary function that returns the number
** of tokens in the current row (including all columns).
*/
static void column_size_imp(
  const Fts5ExtensionApi *pApi,
  Fts5Context *pFts,
  sqlite3_context *pCtx,
  int nVal,
  sqlite3_value **apVal
){
  int rc;
  int nToken;
  rc = pApi->xColumnSize(pFts, -1, &nToken);
  if( rc==SQLITE_OK ){
    sqlite3_result_int(pCtx, nToken);
  }else{
    sqlite3_result_error_code(pCtx, rc);
  }
}
```

The following section describes the API offered to auxiliary function implementations in detail. Further examples may be found in the "fts5_aux.c" file of the source code.

7.2.1. Custom Auxiliary Functions API Overview

This section provides an overview of the capabilities of the auxiliary function API. It does not describe every function. Refer to the reference text below for a complete description.

When invoked, an auxiliary function implementation has access to APIs that allow it to query FTS5 for various information. Some of these APIs return information relating to the current row of the FTS5 table being visited, some relating to the entire set of rows that will be visited by the FTS5 query, and some relating to the FTS5 table. Given an FTS5 table populated as follows:

```
CREATE VIRTUAL TABLE ft USING fts5(a, b);
INSERT INTO ft(rowid, a, b) VALUES
        (1, 'ab cd', 'cd de one'),
        (2, 'de fg', 'fg gh'),
        (3, 'gh ij', 'ij ab three four');
```

and the query:

```
SELECT my_aux_function(ft) FROM ft('ab')
```

then the custom auxiliary function will be invoked for rows 1 and 3 (all rows that contain the token "ab" and therefore match the query).

Number of rows/columns in table: xRowCount, xColumnCount

The system may be queried for the total number of rows in the FTS5 table using the xRowCount API. This provides the total number of rows in the table, not the number that match the current query.

Table columns are numbered from left to right starting from 0. The "rowid" column does not count - only user declared columns - so in the example above column "a" is column 0 and column "b" is column 1. From within an auxiliary function implementation, the xColumnCount API may be used to determine how many columns the table being queried has. If the xColumnCount() API is invoked from within the implementation of the auxiliary function my_aux_function in the example above, it returns 2.

Data From Current Row: xColumnText, xRowid

The xRowid API may be used to find the rowid value for the current row. The xColumnText may be used to obtain the text stored in a specified column of the current row.

Token Counts: xColumnSize, xColumnTotalSize

FTS5 divides documents inserted into an fts5 table into tokens. These are usually just words, perhaps folded to either upper or lower case and with any punctuation removed. For example, the default unicode61 tokenizer tokenizes the text "The tokenizer is case-insensitive" to a list of 5 tokens - "the", "tokenizer", is", "case" and "insensitive". Exactly how tokens are extracted from text is determined by the tokenizer.

The auxiliary functions API provides functions to query for both the number of tokens in a specified column of the current row (the xColumnSize API), or for the number of tokens in a specified column of all rows of the table (the xColumnTotalSize API). For the example at the top of this section, when visiting row 1, xColumnSize returns 2 for column 0 and 3 for column 1. xColumnTotalSize returns 6 for column 0 and 9 for column 1 regardless of the current row.

The Current Full-Text Query: xPhraseCount, xPhraseSize, xQueryToken

An FTS5 query contains one or more phrases. The xPhraseCount, xPhraseSize and xQueryToken APIs allow an auxiliary function implementation to query the system for details of the current query. The xPhraseCount API returns the number of phrases in the current query. For example, if an FTS5 table is queried as follows:

```
SELECT my_aux_function(ft) FROM ft('ab AND "cd ef gh" OR ij + kl')
```

and the xPhraseCount() API invoked from within the implementation of the auxiliary function, it returns 3 (the three phrases being "ab", "ce ef gh" and "ij kl").

Phrases are numbered in order of appearance within a query starting from 0. The xPhraseSize() API may be used to query for the number of tokens in a specified phrase of the query. In the example above, phrase 0 contains 1 token, phrase 1 contains 3 tokens, and phrase 2 contains 2.

The xQueryToken API may be used to access the text of a specified token within a specified phrase of the query. Tokens are numbered within their phrases from left to right starting from 0. For example, if the xQueryToken API is used to request token 1 of phrase 2 in the example above, it returns the text "kl". Token 0 of phrase 0 is "ab".

Phrase Hits in the Current Row: xPhraseFirst, xPhraseNext

These two API functions may be used to iterate through the matches for a specified phrase of the query within the current row. Phrase matches are identified by the column and token offset within the current row. For example, say the following example table:

```
CREATE VIRTUAL TABLE ft2 USING fts5(x, y);
INSERT INTO ft2(rowid, x, y) VALUES
        (1, 'xxx one two xxx five xxx six', 'seven four'),
        (2, 'five four four xxx six', 'three four five six four five six');
```

is queried with:

```
SELECT my_aux_function(ft2) FROM ft2(
    '("one two" OR "three") AND y:four NEAR(five six, 2)'
);
```

The query above contains 5 phrases - "one two", "three", "four", "five" and "six". It matches all rows of the table, so the auxiliary function is invoked for each row.

In row 1, for phrase 0, "one two", there is exactly one match to iterate through - at column 0 token offset 1. The column number is 0 because the match appears in the left most column. The token offset is 1 because there is exactly one token ("xxx") before the phrase match in the column value. For phrase 1, "three", there are no matches. Phrase 2, "four", has one match, at column 1, token offset 0. Phrase 3, "five", has one match at column 0, token offset 4, and phrase 4, "six", has one match at column 0 token offset 6.

The set of matches for each phrase in each row of the example is presented in the table below. Each match is notated as (column-number, token-offset):

Row	Phrase 0	Phrase 1	Phrase 2	Phrase 3	Phrase 4
1	(0, 1)		(1, 1)	(0, 4)	(0, 6)
2		(1,0)	(1, 1), (1,4)	(1, 2), (1, 5)	(1, 3), (1, 6)

The second row is slightly more complicated. There were no occurrences of phrase 0. Phrase 1 ("three") appears once, at column 1 token offset 0. Although there are instances of phrase 2 ("four") in column 0, none of them are reported by the API, as phrase 4 has a column filter - "y:". Matches that are filtered out by column filters do not count. Similarly, although phrases 3 and 4 do occur in column "x" of row 2, they are filtered out by the NEAR filter. Matches that are filtered out by NEAR filters do not count either.

Phrase Hits in the Current Row (2): xInstCount, xInst

The xInstCount and xInst APIs provide access to the same information as the xPhraseFirst and xPhraseNext described above. The difference is that instead of iterating through the matches for a single, specified phrase, the xInstCount/xInst APIs collate all matches into a single flat array, sorted in order of occurrence within the current row. Elements of this array may then be accessed randomly.

Each array element consists of three values:

- A phrase number,
- A column number, and
- A token offset

Using the same example data and query as for xPhraseFirst/xPhraseNext above, the array accessible via xInstCount/xInst consists of the following entries for each row:

Row	xInstCount/xInst array
1	(0, 0, 1), (3, 0, 4), (4, 0, 6), (2, 1, 1)
2	(1, 1, 0), (2, 1, 1), (3, 1, 2), (4, 1, 3), (2, 1, 4), (3, 1, 5), (4, 1, 6)

Each entry of the array is called a phrase match. Phrase matches are numbered in order, starting from 0. So, in the example above, in row 2, phrase match 3 is (4, 1, 3) - phrase 4 of the query matches at column 1, token offset 3.

7.2.2. Custom Auxiliary Functions API Reference

```
struct Fts5ExtensionApi {
  int iVersion;                   /* Currently always set to 4 */

  void *(*xUserData)(Fts5Context*);

  int (*xColumnCount)(Fts5Context*);
  int (*xRowCount)(Fts5Context*, sqlite3_int64 *pnRow);
  int (*xColumnTotalSize)(Fts5Context*, int iCol, sqlite3_int64 *pnToken);

  int (*xTokenize)(Fts5Context*,
    const char *pText, int nText, /* Text to tokenize */
    void *pCtx,                   /* Context passed to xToken() */
    int (*xToken)(void*, int, const char*, int, int, int)       /* Callback */
  );

  int (*xPhraseCount)(Fts5Context*);
  int (*xPhraseSize)(Fts5Context*, int iPhrase);

  int (*xInstCount)(Fts5Context*, int *pnInst);
  int (*xInst)(Fts5Context*, int iIdx, int *piPhrase, int *piCol, int *piOff);

  sqlite3_int64 (*xRowid)(Fts5Context*);
  int (*xColumnText)(Fts5Context*, int iCol, const char **pz, int *pn);
  int (*xColumnSize)(Fts5Context*, int iCol, int *pnToken);

  int (*xQueryPhrase)(Fts5Context*, int iPhrase, void *pUserData,
    int(*)(const Fts5ExtensionApi*,Fts5Context*,void*)
  );
  int (*xSetAuxdata)(Fts5Context*, void *pAux, void(*xDelete)(void*));
  void *(*xGetAuxdata)(Fts5Context*, int bClear);

  int (*xPhraseFirst)(Fts5Context*, int iPhrase, Fts5PhraseIter*, int*, int*);
  void (*xPhraseNext)(Fts5Context*, Fts5PhraseIter*, int *piCol, int *piOff);

  int (*xPhraseFirstColumn)(Fts5Context*, int iPhrase, Fts5PhraseIter*, int*);
  void (*xPhraseNextColumn)(Fts5Context*, Fts5PhraseIter*, int *piCol);

  /* Below this point are iVersion>=3 only */
  int (*xQueryToken)(Fts5Context*,
      int iPhrase, int iToken,
      const char **ppToken, int *pnToken
  );
  int (*xInstToken)(Fts5Context*, int iIdx, int iToken, const char**, int*);

  /* Below this point are iVersion>=4 only */
  int (*xColumnLocale)(Fts5Context*, int iCol, const char **pz, int *pn);
  int (*xTokenize_v2)(Fts5Context*,
    const char *pText, int nText,      /* Text to tokenize */
    const char *pLocale, int nLocale,  /* Locale to pass to tokenizer */
    void *pCtx,                        /* Context passed to xToken() */
    int (*xToken)(void*, int, const char*, int, int, int)       /* Callback */
  );
};
```

void *(*xUserData)(Fts5Context*)

Return a copy of the pUserData pointer passed to the xCreateFunction() API when the extension function was registered.

int (*xColumnTotalSize)(Fts5Context*, int iCol, sqlite3_int64 *pnToken)

If parameter iCol is less than zero, set output variable *pnToken to the total number of tokens in the FTS5 table. Or, if iCol is non-negative but less than the number of columns in the table, return the total number of tokens in column iCol, considering all rows in the FTS5 table.

If parameter iCol is greater than or equal to the number of columns in the table, SQLITE_RANGE is returned. Or, if an error occurs (e.g. an OOM condition or IO error), an appropriate SQLite error code is returned.

int (*xColumnCount)(Fts5Context*)
Return the number of columns in the table.

int (*xColumnSize)(Fts5Context*, int iCol, int *pnToken)
If parameter iCol is less than zero, set output variable *pnToken to the total number of tokens in the current row. Or, if iCol is non-negative but less than the number of columns in the table, set *pnToken to the number of tokens in column iCol of the current row.

If parameter iCol is greater than or equal to the number of columns in the table, SQLITE_RANGE is returned. Or, if an error occurs (e.g. an OOM condition or IO error), an appropriate SQLite error code is returned.

This function may be quite inefficient if used with an FTS5 table created with the "columnsize=0" option.

int (*xColumnText)(Fts5Context*, int iCol, const char **pz, int *pn)
If parameter iCol is less than zero, or greater than or equal to the number of columns in the table, SQLITE_RANGE is returned.

Otherwise, this function attempts to retrieve the text of column iCol of the current document. If successful, (*pz) is set to point to a buffer containing the text in utf-8 encoding, (*pn) is set to the size in bytes (not characters) of the buffer and SQLITE_OK is returned. Otherwise, if an error occurs, an SQLite error code is returned and the final values of (*pz) and (*pn) are undefined.

int (*xPhraseCount)(Fts5Context*)
Returns the number of phrases in the current query expression.

int (*xPhraseSize)(Fts5Context*, int iPhrase)
If parameter iCol is less than zero, or greater than or equal to the number of phrases in the current query, as returned by xPhraseCount, 0 is returned. Otherwise, this function returns the number of tokens in phrase iPhrase of the query. Phrases are numbered starting from zero.

int (*xInstCount)(Fts5Context*, int *pnInst)
Set *pnInst to the total number of occurrences of all phrases within the query within the current row. Return SQLITE_OK if successful, or an error code (i.e. SQLITE_NOMEM) if an error occurs.

This API can be quite slow if used with an FTS5 table created with the "detail=none" or "detail=column" option. If the FTS5 table is created with either "detail=none" or "detail=column" and "content=" option (i.e. if it is a contentless table), then this API always returns 0.

int (*xInst)(Fts5Context*, int iIdx, int *piPhrase, int *piCol, int *piOff)
Query for the details of phrase match iIdx within the current row. Phrase matches are numbered starting from zero, so the iIdx argument should be greater than or equal to zero and smaller than the value output by xInstCount(). If iIdx is less than zero or greater than or equal to the value returned by xInstCount(), SQLITE_RANGE is returned.

Otherwise, output parameter *piPhrase is set to the phrase number, *piCol to the column in which it occurs and *piOff the token offset of the first token of the phrase. SQLITE_OK is returned if successful, or an error code (i.e. SQLITE_NOMEM) if an error occurs.

This API can be quite slow if used with an FTS5 table created with the "detail=none" or "detail=column" option.

sqlite3_int64 (*xRowid)(Fts5Context*)

Returns the rowid of the current row.

```
int (*xTokenize)(Fts5Context*,
    const char *pText, int nText,
    void *pCtx,
    int (*xToken)(void*, int, const char*, int, int, int)
)
```

Tokenize text using the tokenizer belonging to the FTS5 table.

```
int (*xQueryPhrase)(Fts5Context*, int iPhrase, void *pUserData,
    int(*)(const Fts5ExtensionApi*,Fts5Context*,void*)
)
```

This API function is used to query the FTS table for phrase iPhrase of the current query. Specifically, a query equivalent to:

```
... FROM ftstable WHERE ftstable MATCH $p ORDER BY rowid
```

with $p set to a phrase equivalent to the phrase iPhrase of the current query is executed. Any column filter that applies to phrase iPhrase of the current query is included in $p. For each row visited, the callback function passed as the fourth argument is invoked. The context and API objects passed to the callback function may be used to access the properties of each matched row. Invoking Api.xUserData() returns a copy of the pointer passed as the third argument to pUserData.

If parameter iPhrase is less than zero, or greater than or equal to the number of phrases in the query, as returned by xPhraseCount(), this function returns SQLITE_RANGE.

If the callback function returns any value other than SQLITE_OK, the query is abandoned and the xQueryPhrase function returns immediately. If the returned value is SQLITE_DONE, xQueryPhrase returns SQLITE_OK. Otherwise, the error code is propagated upwards.

If the query runs to completion without incident, SQLITE_OK is returned. Or, if some error occurs before the query completes or is aborted by the callback, an SQLite error code is returned.

int (*xSetAuxdata)(Fts5Context*, void *pAux, void(*xDelete)(void*))

Save the pointer passed as the second argument as the extension function's "auxiliary data". The pointer may then be retrieved by the current or any future invocation of the same fts5 extension function made as part of the same MATCH query using the xGetAuxdata() API.

Each extension function is allocated a single auxiliary data slot for each FTS query (MATCH expression). If the extension function is invoked more than once for a single FTS query, then all invocations share a single auxiliary data context.

If there is already an auxiliary data pointer when this function is invoked, then it is replaced by the new pointer. If an xDelete callback was specified along with the original pointer, it is invoked at this point.

The xDelete callback, if one is specified, is also invoked on the auxiliary data pointer after the FTS5 query has finished.

If an error (e.g. an OOM condition) occurs within this function, the auxiliary data is set to NULL and an error code returned. If the xDelete parameter was not NULL, it is invoked on the auxiliary data pointer before returning.

void *(*xGetAuxdata)(Fts5Context*, int bClear)

Returns the current auxiliary data pointer for the fts5 extension function. See the xSetAuxdata() method for details.

If the bClear argument is non-zero, then the auxiliary data is cleared (set to NULL) before this function returns. In this case the xDelete, if any, is not invoked.

int (*xRowCount)(Fts5Context*, sqlite3_int64 *pnRow)

This function is used to retrieve the total number of rows in the table. In other words, the same value that would be returned by:

```
SELECT count(*) FROM ftstable;
```

int (*xPhraseFirst)(Fts5Context*, int iPhrase, Fts5PhraseIter*, int*, int*)

This function is used, along with type Fts5PhraseIter and the xPhraseNext method, to iterate through all instances of a single query phrase within the current row. This is the same information as is accessible via the xInstCount/xInst APIs. While the xInstCount/xInst APIs are more convenient to use, this API may be faster under some circumstances. To iterate through instances of phrase iPhrase, use the following code:

```
Fts5PhraseIter iter;
int iCol, iOff;
for(pApi->xPhraseFirst(pFts, iPhrase, &iter, &iCol, &iOff);
    iCol>=0;
    pApi->xPhraseNext(pFts, &iter, &iCol, &iOff)
){
  // An instance of phrase iPhrase at offset iOff of column iCol
}
```

The Fts5PhraseIter structure is defined above. Applications should not modify this structure directly - it should only be used as shown above with the xPhraseFirst() and xPhraseNext() API methods (and by xPhraseFirstColumn() and xPhraseNextColumn() as illustrated below).

This API can be quite slow if used with an FTS5 table created with the "detail=none" or "detail=column" option. If the FTS5 table is created with either "detail=none" or "detail=column" and "content=" option (i.e. if it is a contentless table), then this API always iterates through an empty set (all calls to xPhraseFirst() set iCol to -1).

In all cases, matches are visited in (column ASC, offset ASC) order. i.e. all those in column 0, sorted by offset, followed by those in column 1, etc.

void (*xPhraseNext)(Fts5Context*, Fts5PhraseIter*, int *piCol, int *piOff)

See xPhraseFirst above.

int (*xPhraseFirstColumn)(Fts5Context*, int iPhrase, Fts5PhraseIter*, int*)

This function and xPhraseNextColumn() are similar to the xPhraseFirst() and xPhraseNext() APIs described above. The difference is that instead of iterating through all instances of a phrase in the current row, these APIs are used to iterate through the set of columns in the current row that contain one or more instances of a specified phrase. For example:

```
Fts5PhraseIter iter;
int iCol;
for(pApi->xPhraseFirstColumn(pFts, iPhrase, &iter, &iCol);
    iCol>=0;
    pApi->xPhraseNextColumn(pFts, &iter, &iCol)
){
```

```
  // Column iCol contains at least one instance of phrase iPhrase
}
```

This API can be quite slow if used with an FTS5 table created with the "detail=none" option. If the FTS5 table is created with either "detail=none" "content=" option (i.e. if it is a contentless table), then this API always iterates through an empty set (all calls to xPhraseFirstColumn() set iCol to -1).

The information accessed using this API and its companion xPhraseFirstColumn() may also be obtained using xPhraseFirst/xPhraseNext (or xInst/xInstCount). The chief advantage of this API is that it is significantly more efficient than those alternatives when used with "detail=column" tables.

void (*xPhraseNextColumn)(Fts5Context*, Fts5PhraseIter*, int *piCol)

See xPhraseFirstColumn above.

int (*xQueryToken)(Fts5Context*,
** int iPhrase, int iToken,**
** const char **ppToken, int *pnToken**
)

This is used to access token iToken of phrase iPhrase of the current query. Before returning, output parameter *ppToken is set to point to a buffer containing the requested token, and *pnToken to the size of this buffer in bytes.

If iPhrase or iToken are less than zero, or if iPhrase is greater than or equal to the number of phrases in the query as reported by xPhraseCount(), or if iToken is equal to or greater than the number of tokens in the phrase, SQLITE_RANGE is returned and *ppToken and *pnToken are both zeroed.

The output text is not a copy of the query text that specified the token. It is the output of the tokenizer module. For tokendata=1 tables, this includes any embedded 0x00 and trailing data.

int (*xInstToken)(Fts5Context*, int iIdx, int iToken, const char, int*)**

This is used to access token iToken of phrase hit iIdx within the current row. If iIdx is less than zero or greater than or equal to the value returned by xInstCount(), SQLITE_RANGE is returned. Otherwise, output variable (*ppToken) is set to point to a buffer containing the matching document token, and (*pnToken) to the size of that buffer in bytes. This API is not available if the specified token matches a prefix query term. In that case both output variables are always set to 0.

The output text is not a copy of the document text that was tokenized. It is the output of the tokenizer module. For tokendata=1 tables, this includes any embedded 0x00 and trailing data.

This API can be quite slow if used with an FTS5 table created with the "detail=none" or "detail=column" option.

int (*xColumnLocale)(Fts5Context*, int iCol, const char **pz, int *pn)

If parameter iCol is less than zero, or greater than or equal to the number of columns in the table, SQLITE_RANGE is returned.

Otherwise, this function attempts to retrieve the locale associated with column iCol of the current row. Usually, there is no associated locale, and output parameters (*pzLocale) and (*pnLocale) are set to NULL and 0, respectively. However, if the fts5_locale() function was used to associate a locale with the value when it was inserted into the fts5 table, then (*pzLocale) is set to point to a nul-terminated buffer containing the name of the locale in utf-8 encoding. (*pnLocale) is set to the size in bytes of the buffer, not including the nul-terminator.

If successful, SQLITE_OK is returned. Or, if an error occurs, an SQLite error code is returned. The final value of the output parameters is undefined in this case.

```
int (*xTokenize_v2)(Fts5Context*,
    const char *pText, int nText,
    const char *pLocale, int nLocale,
    void *pCtx,
    int (*xToken)(void*, int, const char*, int, int, int)
)
```

Tokenize text using the tokenizer belonging to the FTS5 table. This API is the same as the xTokenize() API, except that it allows a tokenizer locale to be specified.

8. The fts5vocab Virtual Table Module

The fts5vocab virtual table module allows users to extract information from an FTS5 full-text index directly. The fts5vocab module is a part of FTS5 - it is available whenever FTS5 is.

Each fts5vocab table is associated with a single FTS5 table. An fts5vocab table is usually created by specifying two arguments in place of column names in the CREATE VIRTUAL TABLE statement - the name of the associated FTS5 table and the type of fts5vocab table. Currently there are three types of fts5vocab table; "row", "col" and "instance". Unless the fts5vocab table is created within the "temp" database, it must be part of the same database as the associated FTS5 table.

```
-- Create an fts5vocab "row" table to query the full-text index belonging
-- to FTS5 table "ft1".
CREATE VIRTUAL TABLE ft1_v USING fts5vocab('ft1', 'row');

-- Create an fts5vocab "col" table to query the full-text index belonging
-- to FTS5 table "ft2".
CREATE VIRTUAL TABLE ft2_v USING fts5vocab(ft2, col);

-- Create an fts5vocab "instance" table to query the full-text index
-- belonging to FTS5 table "ft3".
CREATE VIRTUAL TABLE ft3_v USING fts5vocab(ft3, instance);
```

If an fts5vocab table is created in the temp database, it may be associated with an FTS5 table in any attached database. In order to attach the fts5vocab table to an FTS5 table located in a database other than "temp", the name of the database is inserted before the FTS5 table name in the CREATE VIRTUAL TABLE arguments. For example:

```
-- Create an fts5vocab "row" table to query the full-text index belonging
-- to FTS5 table "ft1" in database "main".
CREATE VIRTUAL TABLE temp.ft1_v USING fts5vocab(main, 'ft1', 'row');

-- Create an fts5vocab "col" table to query the full-text index belonging
-- to FTS5 table "ft2" in attached database "aux".
CREATE VIRTUAL TABLE temp.ft2_v USING fts5vocab('aux', ft2, col);

-- Create an fts5vocab "instance" table to query the full-text index
-- belonging to FTS5 table "ft3" in attached database "other".
CREATE VIRTUAL TABLE temp.ft2_v USING fts5vocab('aux', ft3, 'instance');
```

Specifying three arguments when creating an fts5vocab table in any database other than "temp" results in an error.

An fts5vocab table of type "row" contains one row for each distinct term in the associated FTS5 table. The table columns are as follows:

Column	Contents
term	The term, as stored in the FTS5 index.
doc	The number of rows that contain at least one instance of the term.

cnt	The total number of instances of the term in the entire FTS5 table.

An fts5vocab table of type "col" contains one row for each distinct term/column combination in the associated FTS5 table. Table columns are as follows:

Column	Contents
term	The term, as stored in the FTS5 index.
col	The name of the FTS5 table column that contains the term.
doc	The number of rows in the FTS5 table for which column $col contains at least one instance of the term.
cnt	The total number of instances of the term that appear in column $col of the FTS5 table (considering all rows).

An fts5vocab table of type "instance" contains one row for each term instance stored in the associated FTS index. Assuming the FTS5 table is created with the 'detail' option set to 'full', table columns are as follows:

Column	Contents
term	The term, as stored in the FTS5 index.
doc	The rowid of the document that contains the term instance.
col	The name of the column that contains the term instance.
offset	The index of the term instance within its column. Terms are numbered in order of occurrence starting from 0.

If the FTS5 table is created with the 'detail' option set to 'col', then the *offset* column of an instance virtual table always contains NULL. In this case there is one row in the table for each unique term/doc/col combination. Or, if the FTS5 table is created with 'detail' set to 'none', then both *offset* and *col* always contain NULL values. For detail=none FTS5 tables, there is one row in the fts5vocab table for each unique term/doc combination.

Example:

```
-- Assuming a database created using:
CREATE VIRTUAL TABLE ft USING fts5(c1, c2);
INSERT INTO ft VALUES('apple banana cherry', 'banana banana cherry');
INSERT INTO ft VALUES('cherry cherry cherry', 'date date date');

-- Then querying the following fts5vocab table (type "col") returns:
--
--    apple  | c1 | 1 | 1
--    banana | c1 | 1 | 1
--    banana | c2 | 1 | 2
--    cherry | c1 | 2 | 4
--    cherry | c2 | 1 | 1
--    date   | c3 | 1 | 3
--
CREATE VIRTUAL TABLE ft_v_col USING fts5vocab(ft, col);

-- Querying an fts5vocab table of type "row" returns:
--
--    apple  | 1 | 1
--    banana | 1 | 3
--    cherry | 2 | 5
--    date   | 1 | 3
--
CREATE VIRTUAL TABLE ft_v_row USING fts5vocab(ft, row);

-- And, for type "instance"
INSERT INTO ft VALUES('apple banana cherry', 'banana banana cherry');
INSERT INTO ft VALUES('cherry cherry cherry', 'date date date');
--
```

```
--    apple  | 1 | c1 | 0
--    banana | 1 | c1 | 1
--    banana | 1 | c2 | 0
--    banana | 1 | c2 | 1
--    cherry | 1 | c1 | 2
--    cherry | 1 | c2 | 2
--    cherry | 2 | c1 | 0
--    cherry | 2 | c1 | 1
--    cherry | 2 | c1 | 2
--    date   | 2 | c2 | 0
--    date   | 2 | c2 | 1
--    date   | 2 | c2 | 2
--
CREATE VIRTUAL TABLE ft_v_instance USING fts5vocab(ft, instance);
```

9. FTS5 Data Structures

This section describes at a high-level the way the FTS module stores its index and content in the database. It is not necessary to read or understand the material in this section in order to use FTS in an application. However, it may be useful to application developers attempting to analyze and understand FTS performance characteristics, or to developers contemplating enhancements to the existing FTS feature set.

When an FTS5 virtual table is created in a database, between 3 and 5 real tables are created in the database. These are known as "shadow tables", and are used by the virtual table module to store persistent data. They should not be accessed directly by the user. Many other virtual table modules, including FTS3 and rtree, also create and use shadow tables.

FTS5 creates the following shadow tables. In each case the actual table name is based on the name of the FTS5 virtual table (in the following, replace % with the name of the virtual table to find the actual shadow table name).

```
-- This table contains most of the full-text index data.
CREATE TABLE %_data(id INTEGER PRIMARY KEY, block BLOB);

-- This table contains the remainder of the full-text index data.
-- It is almost always much smaller than the %_data table.
CREATE TABLE %_idx(segid, term, pgno, PRIMARY KEY(segid, term)) WITHOUT ROWID;

-- Contains the values of persistent configuration parameters.
CREATE TABLE %_config(k PRIMARY KEY, v) WITHOUT ROWID;

-- Contains the size of each column of each row in the virtual table
-- in tokens. This shadow table is not present if the "columnsize"
-- option is set to 0.
CREATE TABLE %_docsize(id INTEGER PRIMARY KEY, sz BLOB);

-- Contains the actual data inserted into the FTS5 table. There
-- is one "cN" column for each indexed column in the FTS5 table.
-- This shadow table is not present for contentless or external
-- content FTS5 tables.
CREATE TABLE %_content(id INTEGER PRIMARY KEY, c0, c1...);
```

The following sections describe in more detail how these five tables are used to store FTS5 data.

9.1. Varint Format

The sections below refer to 64-bit signed integers stored in "varint" form. FTS5 uses the same varint format as used in various places by the SQLite core.

A varint is between 1 and 9 bytes in length. The varint consists of either zero or more bytes which have the high-order bit set followed by a single byte with the high-order bit clear, or nine bytes, whichever is

shorter. The lower seven bits of each of the first eight bytes and all 8 bits of the ninth byte are used to reconstruct the 64-bit twos-complement integer. Varints are big-endian: bits taken from the earlier byte of the varint are more significant than bits taken from the later bytes.

9.2. The FTS Index (%_idx and %_data tables)

The FTS index is an ordered key-value store where the keys are document terms or term prefixes and the associated values are "doclists". A doclist is a packed array of varints that encodes the position of each instance of the term within the FTS5 table. The position of a single term instance is defined as the combination of:

- The rowid of the FTS5 table row it appears in,
- The index of the column the term instance appears in (columns are numbered from left to right starting from zero), and
- The offset of the term within the column value (i.e. the number of tokens that appear within the column value before this one).

The FTS index contains up to (nPrefix+1) entries for each token in the data set, where nPrefix is the number of defined prefix indexes.

Keys associated with the main FTS index (the one that is not a prefix index) are prefixed with the character "0". Keys for the first prefix index are prefixed with "1". Keys for the second prefix index are prefixed with "2", and so on. For example, if the token "document" is inserted into an FTS5 table with prefix indexes specified by prefix="2 4", then the keys added to the FTS index would be "0document", "1do" and "2docu".

The FTS index entries are not stored in a single tree or hash table structure. Instead, they are stored in a series of immutable b-tree like structures referred to as "segment b-trees". Each time a write to the FTS5 table is committed, one or more (but usually just one) new segment b-trees are added containing both the new entries and tombstones for any deleted entries. When the FTS index is queried, the reader queries each segment b-tree in turn and merges the results, giving priority to newer data.

Each segment b-tree is assigned a numerical level. When a new segment b-tree is written to the database as part of committing a transaction, it is assigned to level 0. Segment b-trees belonging to a single level are periodically merged together to create a single, larger segment b-tree that is assigned to the next level (i.e. level 0 segment b-trees are merged to become a single level 1 segment b-tree). Thus the numerically larger levels contain older data in (usually) larger segment b-trees. Refer to the 'automerge', 'crisismerge' and 'usermerge' options, along with the 'merge' and 'optimize' commands for details on how to control the merging.

In cases where the doclist associated with a term or term prefix is very large, there may be an associated doclist index. A doclist index is similar to the set of internal nodes of a b-tree. It allows a large doclist to be efficiently queried for rowids or ranges of rowids. For example, when processing a query like:

```
SELECT ... FROM ft('term') WHERE rowid BETWEEN ? AND ?
```

FTS5 uses the segment b-tree index to locate the doclist for term "term", then uses its doclist index (assuming it is present) to efficiently identify the subset of matches with rowids in the required range.

9.2.1. The %_data Table Rowid Space

```
CREATE TABLE %_data(
  id INTEGER PRIMARY KEY,
  block BLOB
);
```

The %_data table is used to store three types of records:

- The special structure record, stored with id=10.
- The special averages record, stored with id=1.
- A record to store each segment b-tree leaf and doclist index leaf and internal node. See below for how id values are calculated for these records.

Each segment b-tree in the system is assigned a unique 16-bit segment id. Segment ids may only be reused after the original owner segment b-tree is completely merged into a higher level segment b-tree. Within a segment b-tree, each leaf page is assigned a unique page number - 1 for the first leaf page, 2 for the second, and so on.

Each doclist index leaf page is also assigned a page number. The first (leftmost) leaf page in a doclist index is assigned the same page number as the segment b-tree leaf page on which its term appears (because doclist indexes are only created for terms with very long doclists, at most one term per segment b-tree leaf has an associated doclist index). Call this page number P. If the doclist is so large that it requires a second leaf, the second leaf is assigned page number P+1. The third leaf P+2. Each tier of a doclist index b-tree (leaves, parents of leaves, grandparents etc.) is assigned page numbers in this fashion, starting with page number P.

The "id" value used in the %_data table to store any given segment b-tree leaf or doclist index leaf or node is composed as follows:

Rowid Bits	Contents
38..43	(16 bit) Segment b-tree id value.
37	(1 bit) Doclist index flag. Set for doclist index pages, clear for segment b-tree leaves.
32..36	(5 bits) Height in tree. This is set to 0 for segment b-tree and doclist index leaves, to 1 for the parents of doclist index leaves, 2 for the grandparents, etc.
0..31	(32 bits) Page number

9.2.2. Structure Record Format

The structure record identifies the set of segment b-trees that make up the current FTS index, along with details of any ongoing incremental merge operations. It is stored in the %_data table with id=10. A structure record begins with a single 32-bit unsigned value - the cookie value. This value is incremented each time the structure is modified. Following the cookie value are three varint values, as follows:

- The number of levels in the index (i.e. the maximum level associated with any segment b-tree plus one).
- The total number of segment b-trees in the index.
- The total number of segment b-tree leaves written to level 0 trees since the FTS5 table was created.

Then, for each level from 0 to nLevel:

- The number of input segments from the previous level being used as inputs for the current incremental merge, or zero if there is no ongoing incremental merge to create a new segment b-tree for this level.
- The total number of segment b-trees on the level.
- Then, for each segment b-tree, from oldest to newest:
  - The segment id.
  - Page number of first leaf (often 1, always >0).
  - Page number of last leaf (always >0).

9.2.3. Averages Record Format

The averages record, which is always stored with id=1 in the %_data table, does not store the average of anything. Instead, it contains a vector of (nCol+1) packed varint values, where nCol is the number of columns in the FTS5 table, including unindexed columns. The first varint contains the total number of rows in the FTS5 table. The second contains the total number of tokens in all values stored in the leftmost FTS5 table column. The third the number of tokens in all values for the next leftmost, and so on. The value for unindexed columns is always zero.

9.2.4. Segment B-Tree Format

9.2.4.1. The Key/Doclist Format

The key/doclist format is a format used to store a series of keys (document terms or term prefixes prefixed by a single character to indentify the specific index to which they belong) in sorted order, each with their associated doclist. The format consists of alternating keys and doclists packed together.

The first key is stored as:

- A varint indicating the number of bytes in the key (N), followed by
- The key data itself (N bytes).

Each subsequent key is stored as:

- A varint indicating the size of the prefix that the key has in common with the previous key in bytes,
- A varint indicating the number of bytes in the key following the common prefix (N), followed by
- The key suffix data itself (N bytes).

For example, if the first two keys in an FTS5 key/doclist record are "0challenger" and "0chandelier", then the first key is stored as varint 11 followed by the 11 bytes "0challenger", and the second key is stored as varints 4 and 7, followed by the 7 bytes "ndelier".

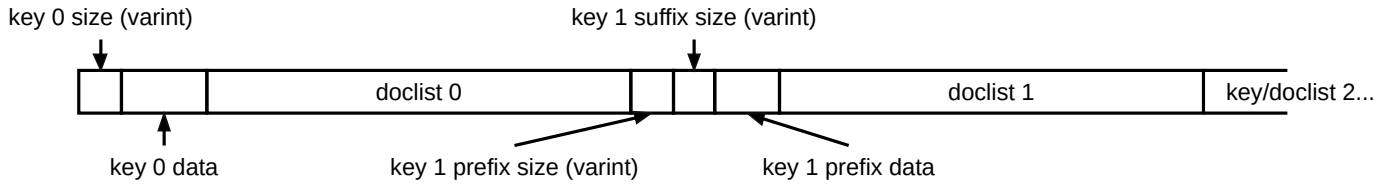

Figure 1 - Term/Doclist Format

Each doclist identifies the rows (by their rowid values) that contain at least one instance of the term or term prefix and an associated position list, or "poslist" enumerating the position of each term instance within the row. In this sense a "position" is defined as a column number and term offset within the column value.

Within a doclist, documents are always stored in order sorted by rowid. The first rowid in a doclist is stored as is, as a varint. It is immediately followed by its associated position list. Following this, the difference between the first rowid and the second, as a varint, followed by the doclist associated with the second rowid in the doclist. And so on.

There is no way to determine the size of a doclist by parsing it. This must be stored externally. See the section below for details of how this is accomplished in FTS5.

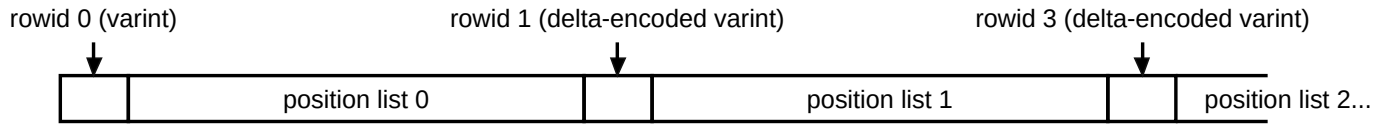

Figure 2 - Doclist Format

A position list - often shortened to "poslist" - identifies the column and token offset within the row of each instance of the token in question. The format of a poslist is:

- Varint set to twice the size of the poslist, not including this field, plus one if the "delete" flag is set on the entry.
- A (possibly empty) list of offsets for column 0 (the leftmost column) of the row. Each offset is stored as a varint. The first varint contains the value of the first offset, plus 2. The second variant contains the difference between the second and first offsets, plus 2. etc. For example, if the offset list is to contain offsets 0, 10, 15 and 16, it is encoded by packing the following values, encoded as varints, end to end:

```
2, 12, 7, 3
```

- For each column other than column 0 that contains one of more instances of the token:
  - Byte value 0x01.
  - The column number, as a varint.
  - An offset list, in the same format as the offset list for column 0.

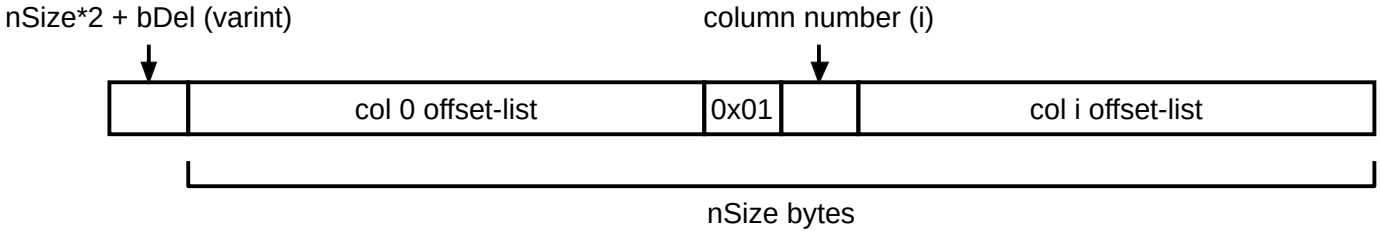

Figure 3 - Position List (poslist) With Offsets in Columns 0 and i

9.2.4.2. Pagination

If it is small enough (by default this means smaller than 4000 bytes), the entire contents of a segment b-tree may be stored in the key/doclist format described in the previous section as a single blob within the %_data table. Otherwise, the key/doclist is split into pages (by default, of approximately 4000 bytes each) and stored in a contiguous set of entries in the %_data table (see above for details).

When a key/doclist is divided into pages, the following modifications are made to the format:

- A single varint or key data field never spans two pages.
- The first key on each page is not prefix-compressed. It is stored in the format described above for the first key of a doclist - its size as a varint followed by the key data.
- If there are one or more rowids on a page before the first key, then the first of them is not delta compressed. It is stored as is, just as if it were the first rowid of its doclist (which it may or may not be).

Each page also has fixed-size 4-byte header and a variably-sized footer. The header is divided into 2 16-bit big-endian integer fields. They contain:

- The byte offset of the first rowid value on the page, if it occurs before the first key, or 0 otherwise.
- The byte offset of the page footer.

The page footer consists of a series of varints containing the byte offset of each key that appears on the page. The page footer is zero bytes in size if there are no keys on the page.

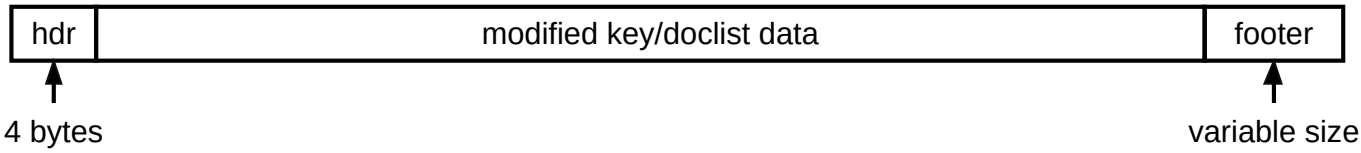

Figure 4 - Page Format

9.2.4.3. Segment Index Format

The result of formatting the contents of the segment b-tree in the key/doclist format and then splitting it into pages is something very similar to the leaves of a b+tree. Instead of creating a format for the internal nodes of this b+tree and storing them in the %_data table alongside the leaves, the keys that would have been stored on such nodes are added to the %_idx table, defined as:

```
CREATE TABLE %_idx(
  segid INTEGER,                -- segment id
  term TEXT,                    -- prefix of first key on page
  pgno INTEGER,                 -- (2*pgno + bDoclistIndex)
  PRIMARY KEY(segid, term)
);
```

For each "leaf" page that contains at least one key, an entry is added to the %_idx table. Fields are set as follows:

Column	Contents
segid	The integer segment id.
term	The smallest prefix of the first key on the page that is larger than all keys on the previous page. For the first page in a segment, this prefix is zero bytes in size.
pgno	This field encodes both the page number (within the segment - starting from 1) and the doclist index flag. The doclist index flag is set if the final key on the page has an associated doclist index. The value of this field is: `(pgno*2 + bDoclistIndexFlag)`

Then, to find the leaf for segment i that may contain term t, instead of searching through internal nodes, FTS5 runs the query:

```
SELECT pgno FROM %_idx WHERE segid=$i AND term>=$t ORDER BY term LIMIT 1
```

9.2.4.4. Doclist Index Format

The segment index described in the previous section allows a segment b-tree to be efficiently queried by term or, assuming there is a prefix index of the required size, a term prefix. The data structure described in this section, doclist indexes, allows FTS5 to efficiently search for a rowid or range or rowids within the doclist associated with a single term or term prefix.

Not all keys have associated doclists indexes. By default, a doclist index is only added for a key if its doclist spans more than 4 segment b-tree leaf pages. Doclist indexes are themselves b-trees, with both leaves and internal nodes stored as entries in the %_data table, but in practice most doclists are small enough to fit on a single leaf. FTS5 uses the same rough size for doclist index node and leaves as it does for segment b-tree leaves (by default 4000 bytes).

Doclist index leaves and internal nodes use the same page format. The first byte is a "flags" byte. This is set to 0x00 for the root page of the doclist index b-tree, and 0x01 for all other pages. The remainder of the page is a series of tightly packed varints, as follows:

- page number of leftmost child page, followed by
- the smallest rowid value on the left most child page, followed by
- one varint for each subsequent child page, containing the value:
  - 0x00 if there are no rowids on the child page (this can only happen when the "child" page is actually a segment b-tree leaf), or
  - the difference between the smallest rowid on the child page and the previous rowid value stored on the doclist index page.

For the leftmost doclist index leaf in a doclist index, the leftmost child page is the first segment b-tree leaf after the one that contains the key itself.

9.3. Document Sizes Table (%_docsize table)

```
CREATE TABLE %_docsize(
    id INTEGER PRIMARY KEY,    -- id of FTS5 row this record pertains to
    sz BLOB                    -- blob containing nCol packed varints
);
```

Many common search result ranking functions require as an input the size in tokens of the result document (as a search term hit in a short document is considered more significant than one in a long document). To provide fast access to this information, for each row in the FTS5 table there exists a corresponding record (with the same rowid) in the %_docsize shadow table that contains the size of each column value in the row, in tokens.

The column value sizes are stored in a blob containing one packed varint for each column of the FTS5 table, from left to right. The varint contains, of course, the total number of tokens in the corresponding column value. Unindexed columns are included in this vector of varints; for them the value is always set to zero.

This table is used by the xColumnSize API. It can be omitted altogether by specifying the columnsize=0 option. In that case the xColumnSize API is still available to auxiliary functions, but runs much more slowly.

9.4. The Table Contents (%_content table)

```
-- locale=0 (the default) table
CREATE TABLE %_content(id INTEGER PRIMARY KEY, c0, c1...);

-- locale=1 table
CREATE TABLE %_content(id INTEGER PRIMARY KEY, c0, c1..., l0, l1...);
```

The actual table content - the values inserted into the FTS5 table, is stored in the %_content table. This table is created with one "c*" column for each column of the FTS5 table, including any unindexed columns. The values for the leftmost FTS5 table column are stored in column "c0" of the %_content table, the values from the next FTS5 table column in column "c1", and so on.

For an FTS5 table with the locale option set to 1, the %_content table also contains one "l*" column for each indexed (i.e. not UNINDEXED) column of the table. For values that were written to the fts5 table using the default locale, this column contains a NULL. Or, for values that were written with an associated locale (fts5_locale() values), this column contains the name of the locale, as text.

Each "l*" column name has the same integer component as its associated "c*" column. This means that if the fts5 table has one or more UNINDEXED columns, the set of "l*" column names may not contain a contiguous set of integer components. For example:

```
-- This fts5 table:
CREATE VIRTUAL TABLE ft USING fts5(a, b UNINDEXED, c, locale=1);

-- uses a %_content table with no "l1" column:
CREATE TABLE ft_content(id INTEGER PRIMARY KEY, c0, c1, c2, l0, l2);
```

Unless the contentless_unindexed=1 option is specified, this table is omitted completely for external content or contentless FTS5 tables. For contentless tables that do specify the contentless_unindexed=1 option, the %_content table is created, but contains only those "c*" columns that correspond to UNINDEXED columns of the fts5 table. For example:

```
-- This fts5 table:
CREATE VIRTUAL TABLE ft USING fts5(a, b UNINDEXED, c, contentless_unindexed=1);

-- uses a %_content table with only the "c1" (b) column
CREATE TABLE ft_content(id INTEGER PRIMARY KEY, c1);
```

9.5. Configuration Options (%_config table)

```
CREATE TABLE %_config(k PRIMARY KEY, v) WITHOUT ROWID;
```

This table stores the values of any persistent configuration options. Column "k" stores the name of the option (text) and column "v" the value. Example contents:

```
sqlite> SELECT * FROM ft_config;
```

k	v
crisismerge	8
pgsz	8000
usermerge	4
version	4

Appendix A: Comparison with FTS3/4

Also available is the similar but more mature FTS3/4 module. FTS5 is a new version of FTS4 that includes various fixes and solutions for problems that could not be fixed in FTS4 without sacrificing backwards compatibility. Some of these problems are described below.

Application Porting Guide

In order to use FTS5 instead of FTS3 or FTS4, applications usually require minimal modifications. Most of these fall into three categories - changes required to the CREATE VIRTUAL TABLE statement used to create the FTS table, changes required to SELECT queries used to execute queries against the table, and changes required to applications that use FTS auxiliary functions.

Changes to CREATE VIRTUAL TABLE statements

1. The module name must be changed from "fts3" or "fts4" to "fts5".
2. All type information or constraint specifications must be removed from column definitions. FTS3/4 ignores everything following the column name in a column definition, FTS5 attempts to parse it (and will report an error if it fails to).
3. The "matchinfo=fts3" option is not available. The "columnsize=0" option is equivalent.
4. The notindexed= option is not available. Adding UNINDEXED to the column definition is equivalent.
5. The ICU tokenizer is not available.
6. The compress=, uncompress= and languageid= options are not available. There is as of yet no equivalent for their functionality.

```
 -- FTS3/4 statement
CREATE VIRTUAL TABLE ft USING fts4(
  linkid INTEGER,
```

```
  header CHAR(20),
  text VARCHAR,
  notindexed=linkid,
  matchinfo=fts3,
  tokenizer=unicode61
);

 -- FTS5 equivalent (note - the "tokenizer=unicode61" option is not
 -- required as this is the default for FTS5 anyway)
CREATE VIRTUAL TABLE ft USING fts5(
  linkid UNINDEXED,
  header,
  text,
  columnsize=0
);
```

Changes to SELECT statements

1. The "docid" alias does not exist. Applications must use "rowid" instead.

2. The behaviour of queries when a column-filter is specified both as part of the FTS query and by using a column as the LHS of a MATCH operator is slightly different. For a table with columns "a" and "b" and a query similar to:

   ```
   ... a MATCH 'b: string'
   ```

   FTS3/4 searches for matches in column "b". However, FTS5 always returns zero rows, as results are first filtered for column "b", then for column "a", leaving no results. In other words, in FTS3/4 the inner filter overrides the outer, in FTS5 both filters are applied.

3. The FTS query syntax (right hand side of the MATCH operator) has changed in some ways. The FTS5 syntax is quite close to the FTS4 "enhanced syntax". The main difference is that FTS5 is fussier about unrecognized punctuation characters and similar within query strings. Most queries that work with FTS3/4 should also work with FTS5, and those that do not should return parse errors.

Auxiliary Function Changes

FTS5 has no matchinfo() or offsets() function, and the snippet() function is not as fully-featured as in FTS3/4. However, since FTS5 does provide an API allowing applications to create custom auxiliary functions, any required functionality may be implemented within the application code.

The set of built-in auxiliary functions provided by FTS5 may be improved upon in the future.

Other Issues

1. The functionality provided by the fts4aux module is now provided by fts5vocab. The schema of these two tables is slightly different.

2. The FTS3/4 "merge=X,Y" command has been replaced by the FTS5 merge command.

3. The FTS3/4 "automerge=X" command has been replaced by the FTS5 automerge option.

Summary of Technical Differences

FTS5 is similar to FTS3/4 in that the primary task of each is to maintain an index mapping from each unique token to a list of instances of that token within a set of documents, where each instance is identified by the document in which it appears and its position within that document. For example:

```
-- Given the following SQL:
CREATE VIRTUAL TABLE ft USING fts5(a, b);
INSERT INTO ft(rowid, a, b) VALUES(1, 'X Y', 'Y Z');
INSERT INTO ft(rowid, a, b) VALUES(2, 'A Z', 'Y Y');

-- The FTS5 module creates the following mapping on disk:
A --> (2, 0, 0)
X --> (1, 0, 0)
Y --> (1, 0, 1) (1, 1, 0) (2, 1, 0) (2, 1, 1)
Z --> (1, 1, 1) (2, 0, 1)
```

In the example above, each triple identifies the location of a token instance by rowid, column number (columns are numbered sequentially starting at 0 from left to right) and position within the column value (the first token in a column value is 0, the second is 1, and so on). Using this index, FTS5 is able to provide timely answers to queries such as "the set of all documents that contain the token 'A'", or "the set of all documents that contain the sequence 'Y Z'". The list of instances associated with a single token is called an "instance-list".

The principle difference between FTS3/4 and FTS5 is that in FTS3/4, each instance-list is stored as a single large database record, whereas in FTS5 large instance-lists are divided between multiple database records. This has the following implications for dealing with large databases that contain large lists:

- FTS5 is able to load instance-lists into memory incrementally in order to reduce memory usage and peak allocation size. FTS3/4 very often loads entire instance-lists into memory.
- When processing queries that feature more than one token, FTS5 is sometimes able to determine that the query can be answered by inspecting a subset of a large instance-list. FTS3/4 almost always has to traverse entire instance-lists.
- If an instance-list grows so large that it exceeds the SQLITE_MAX_LENGTH limit, FTS3/4 is unable to handle it. FTS5 does not have this problem.

For these reasons, many complex queries may use less memory and run faster using FTS5.

Some other ways in which FTS5 differs from FTS3/4 are:

- FTS5 supports "ORDER BY rank" for returning results in order of decreasing relevancy.
- FTS5 features an API allowing users to create custom auxiliary functions for advanced ranking and text processing applications. The special "rank" column may be mapped to a custom auxiliary function so that adding "ORDER BY rank" to a query works as expected.
- FTS5 recognizes unicode separator characters and case equivalence by default. This is also possible using FTS3/4, but must be explicitly enabled.
- The query syntax has been revised where necessary to remove ambiguities and to make it possible to escape special characters in query terms.
- By default, FTS3/4 occasionally merges together two or more of the b-trees that make up its full-text index within an INSERT, UPDATE or DELETE statement executed by the user. This means that any operation on an FTS3/4 table may turn out to be surprisingly slow, as FTS3/4 may unpredictably choose to merge together two or more large b-trees within it. FTS5 uses incremental merging by default, which limits the amount of processing that may take place within any given INSERT, UPDATE or DELETE operation.