SQLite FTS3 and FTS4 Extensions

► Table Of Contents

Overview

FTS3 and FTS4 are SQLite virtual table modules that allows users to perform full-text searches on a set of documents. The most common (and effective) way to describe full-text searches is "what Google, Yahoo, and Bing do with documents placed on the World Wide Web". Users input a term, or series of terms, perhaps connected by a binary operator or grouped together into a phrase, and the full-text query system finds the set of documents that best matches those terms considering the operators and groupings the user has specified. This article describes the deployment and usage of FTS3 and FTS4.

FTS1 and FTS2 are obsolete full-text search modules for SQLite. There are known issues with these older modules and their use should be avoided. Portions of the original FTS3 code were contributed to the SQLite project by Scott Hess of Google. It is now developed and maintained as part of SQLite.

1. Introduction to FTS3 and FTS4

The FTS3 and FTS4 extension modules allows users to create special tables with a built-in full-text index (hereafter "FTS tables"). The full-text index allows the user to efficiently query the database for all rows that contain one or more words (hereafter "tokens"), even if the table contains many large documents.

For example, if each of the 517430 documents in the "Enron E-Mail Dataset" is inserted into both an FTS table and an ordinary SQLite table created using the following SQL script:

```
CREATE VIRTUAL TABLE enrondata1 USING fts3(content TEXT);     /* FTS3 table */
CREATE TABLE enrondata2(content TEXT);                        /* Ordinary table */
```

Then either of the two queries below may be executed to find the number of documents in the database that contain the word "linux" (351). Using one desktop PC hardware configuration, the query on the FTS3 table returns in approximately 0.03 seconds, versus 22.5 for querying the ordinary table.

```
SELECT count(*) FROM enrondata1 WHERE content MATCH 'linux';  /* 0.03 seconds */
SELECT count(*) FROM enrondata2 WHERE content LIKE '%linux%'; /* 22.5 seconds */
```

Of course, the two queries above are not entirely equivalent. For example the LIKE query matches rows that contain terms such as "linuxophobe" or "EnterpriseLinux" (as it happens, the Enron E-Mail Dataset does not actually contain any such terms), whereas the MATCH query on the FTS3 table selects only those rows that contain "linux" as a discrete token. Both searches are case-insensitive. The FTS3 table consumes around 2006 MB on disk compared to just 1453 MB for the ordinary table. Using the same hardware configuration used to perform the SELECT queries above, the FTS3 table took just under 31 minutes to populate, versus 25 for the ordinary table.

1.1. Differences between FTS3 and FTS4

FTS3 and FTS4 are nearly identical. They share most of their code in common, and their interfaces are the same. The differences are:

- FTS4 contains query performance optimizations that may significantly improve the performance of full-text queries that contain terms that are very common (present in a large percentage of table rows).
- FTS4 supports some additional options that may used with the matchinfo() function.
- Because it stores extra information on disk in two new shadow tables in order to support the performance optimizations and extra matchinfo() options, FTS4 tables may consume more disk space than the equivalent table created using FTS3. Usually the overhead is 1-2% or less, but may be as high as 10% if the documents stored in the FTS table are very small. The overhead may be reduced by specifying the directive "matchinfo=fts3" as part of the FTS4 table declaration, but this comes at the expense of sacrificing some of the extra supported matchinfo() options.
- FTS4 provides hooks (the compress and uncompress options) allowing data to be stored in a compressed form, reducing disk usage and IO.

FTS4 is an enhancement to FTS3. FTS3 has been available since SQLite version 3.5.0 (2007-09-04) The enhancements for FTS4 were added with SQLite version 3.7.4 (2010-12-07).

Which module, FTS3 or FTS4, should you use in your application? FTS4 is sometimes significantly faster than FTS3, even orders of magnitude faster depending on the query, though in the common case the performance of the two modules is similar. FTS4 also offers the enhanced matchinfo() outputs which can be useful in ranking the results of a MATCH operation. On the other hand, in the absence of a matchinfo=fts3 directive FTS4 requires a little more disk space than FTS3, though only a percent of two in most cases.

For newer applications, FTS4 is recommended; though if compatibility with older versions of SQLite is important, then FTS3 will usually serve just as well.

1.2. Creating and Destroying FTS Tables

Like other virtual table types, new FTS tables are created using a CREATE VIRTUAL TABLE statement. The module name, which follows the USING keyword, is either "fts3" or "fts4". The virtual table module arguments may be left empty, in which case an FTS table with a single user-defined column named "content" is created. Alternatively, the module arguments may be passed a list of comma separated column names.

If column names are explicitly provided for the FTS table as part of the CREATE VIRTUAL TABLE statement, then a datatype name may be optionally specified for each column. This is pure syntactic sugar, the supplied typenames are not used by FTS or the SQLite core for any purpose. The same applies to any constraints specified along with an FTS column name - they are parsed but not used or recorded by the system in any way.

```
-- Create an FTS table named "data" with one column - "content":
CREATE VIRTUAL TABLE data USING fts3();

-- Create an FTS table named "pages" with three columns:
CREATE VIRTUAL TABLE pages USING fts4(title, keywords, body);

-- Create an FTS table named "mail" with two columns. Datatypes
-- and column constraints are specified along with each column. These
-- are completely ignored by FTS and SQLite.
CREATE VIRTUAL TABLE mail USING fts3(
  subject VARCHAR(256) NOT NULL,
  body TEXT CHECK(length(body)<10240)
);
```

As well as a list of columns, the module arguments passed to a CREATE VIRTUAL TABLE statement used to create an FTS table may be used to specify a tokenizer. This is done by specifying a string of the form "tokenize= <tokenizer name> <tokenizer args>" in place of a column name, where <tokenizer name> is the name of the tokenizer to use and <tokenizer args> is an optional list of whitespace separated qualifiers to pass to the tokenizer implementation. A tokenizer specification may be placed anywhere in the column list, but at most one tokenizer

declaration is allowed for each CREATE VIRTUAL TABLE statement. See below for a detailed description of using (and, if necessary, implementing) a tokenizer.

```
-- Create an FTS table named "papers" with two columns that uses
-- the tokenizer "porter".
CREATE VIRTUAL TABLE papers USING fts3(author, document, tokenize=porter);

-- Create an FTS table with a single column - "content" - that uses
-- the "simple" tokenizer.
CREATE VIRTUAL TABLE data USING fts4(tokenize=simple);

-- Create an FTS table with two columns that uses the "icu" tokenizer.
-- The qualifier "en_AU" is passed to the tokenizer implementation
CREATE VIRTUAL TABLE names USING fts3(a, b, tokenize=icu en_AU);
```

FTS tables may be dropped from the database using an ordinary DROP TABLE statement. For example:

```
-- Create, then immediately drop, an FTS4 table.
CREATE VIRTUAL TABLE data USING fts4();
DROP TABLE data;
```

1.3. Populating FTS Tables

FTS tables are populated using INSERT, UPDATE and DELETE statements in the same way as ordinary SQLite tables are.

As well as the columns named by the user (or the "content" column if no module arguments were specified as part of the CREATE VIRTUAL TABLE statement), each FTS table has a "rowid" column. The rowid of an FTS table behaves in the same way as the rowid column of an ordinary SQLite table, except that the values stored in the rowid column of an FTS table remain unchanged if the database is rebuilt using the VACUUM command. For FTS tables, "docid" is allowed as an alias along with the usual "rowid", "oid" and "_oid_" identifiers. Attempting to insert or update a row with a docid value that already exists in the table is an error, just as it would be with an ordinary SQLite table.

There is one other subtle difference between "docid" and the normal SQLite aliases for the rowid column. Normally, if an INSERT or UPDATE statement assigns discrete values to two or more aliases of the rowid column, SQLite writes the rightmost of such values specified in the INSERT or UPDATE statement to the database. However, assigning a non-NULL value to both the "docid" and one or more of the SQLite rowid aliases when inserting or updating an FTS table is considered an error. See below for an example.

```
-- Create an FTS table
CREATE VIRTUAL TABLE pages USING fts4(title, body);

-- Insert a row with a specific docid value.
INSERT INTO pages(docid, title, body) VALUES(53, 'Home Page', 'SQLite is a software...');

-- Insert a row and allow FTS to assign a docid value using the same algorithm as
-- SQLite uses for ordinary tables. In this case the new docid will be 54,
-- one greater than the largest docid currently present in the table.
INSERT INTO pages(title, body) VALUES('Download', 'All SQLite source code...');

-- Change the title of the row just inserted.
UPDATE pages SET title = 'Download SQLite' WHERE rowid = 54;

-- Delete the entire table contents.
DELETE FROM pages;

-- The following is an error. It is not possible to assign non-NULL values to both
-- the rowid and docid columns of an FTS table.
INSERT INTO pages(rowid, docid, title, body) VALUES(1, 2, 'A title', 'A document body');
```

To support full-text queries, FTS maintains an inverted index that maps from each unique term or word that appears in the dataset to the locations in which it appears within the table contents. For the curious, a complete

description of the data structure used to store this index within the database file appears below. A feature of this data structure is that at any time the database may contain not one index b-tree, but several different b-trees that are incrementally merged as rows are inserted, updated and deleted. This technique improves performance when writing to an FTS table, but causes some overhead for full-text queries that use the index. Evaluating the special "optimize" command, an SQL statement of the form "INSERT INTO <fts-table>(<fts-table>) VALUES('optimize')", causes FTS to merge all existing index b-trees into a single large b-tree containing the entire index. This can be an expensive operation, but may speed up future queries.

For example, to optimize the full-text index for an FTS table named "docs":

```
-- Optimize the internal structure of FTS table "docs".
INSERT INTO docs(docs) VALUES('optimize');
```

The statement above may appear syntactically incorrect to some. Refer to the section describing the simple fts queries for an explanation.

There is another, deprecated, method for invoking the optimize operation using a SELECT statement. New code should use statements similar to the INSERT above to optimize FTS structures.

1.4. Simple FTS Queries

As for all other SQLite tables, virtual or otherwise, data is retrieved from FTS tables using a SELECT statement.

FTS tables can be queried efficiently using SELECT statements of two different forms:

- **Query by rowid**. If the WHERE clause of the SELECT statement contains a sub-clause of the form "rowid = ?", where ? is an SQL expression, FTS is able to retrieve the requested row directly using the equivalent of an SQLite INTEGER PRIMARY KEY index.
- **Full-text query**. If the WHERE clause of the SELECT statement contains a sub-clause of the form "<column> MATCH ?", FTS is able to use the built-in full-text index to restrict the search to those documents that match the full-text query string specified as the right-hand operand of the MATCH clause.

If neither of these two query strategies can be used, all queries on FTS tables are implemented using a linear scan of the entire table. If the table contains large amounts of data, this may be an impractical approach (the first example on this page shows that a linear scan of 1.5 GB of data takes around 30 seconds using a modern PC).

```
-- The examples in this block assume the following FTS table:
CREATE VIRTUAL TABLE mail USING fts3(subject, body);

SELECT * FROM mail WHERE rowid = 15;                -- Fast. Rowid lookup.
SELECT * FROM mail WHERE body MATCH 'sqlite';       -- Fast. Full-text query.
SELECT * FROM mail WHERE mail MATCH 'search';       -- Fast. Full-text query.
SELECT * FROM mail WHERE rowid BETWEEN 15 AND 20;   -- Fast. Rowid lookup.
SELECT * FROM mail WHERE subject = 'database';      -- Slow. Linear scan.
SELECT * FROM mail WHERE subject MATCH 'database';  -- Fast. Full-text query.
```

In all of the full-text queries above, the right-hand operand of the MATCH operator is a string consisting of a single term. In this case, the MATCH expression evaluates to true for all documents that contain one or more instances of the specified word ("sqlite", "search" or "database", depending on which example you look at). Specifying a single term as the right-hand operand of the MATCH operator results in the simplest and most common type of full-text query possible. However more complicated queries are possible, including phrase searches, term-prefix searches and searches for documents containing combinations of terms occurring within a defined proximity of each other. The various ways in which the full-text index may be queried are described below.

Normally, full-text queries are case-insensitive. However, this is dependent on the specific tokenizer used by the FTS table being queried. Refer to the section on tokenizers for details.

The paragraph above notes that a MATCH operator with a simple term as the right-hand operand evaluates to true for all documents that contain the specified term. In this context, the "document" may refer to either the data stored in a single column of a row of an FTS table, or to the contents of all columns in a single row, depending on

the identifier used as the left-hand operand to the MATCH operator. If the identifier specified as the left-hand operand of the MATCH operator is an FTS table column name, then the document that the search term must be contained in is the value stored in the specified column. However, if the identifier is the name of the FTS *table* itself, then the MATCH operator evaluates to true for each row of the FTS table for which any column contains the search term. The following example demonstrates this:

```
-- Example schema
CREATE VIRTUAL TABLE mail USING fts3(subject, body);

-- Example table population
INSERT INTO mail(docid, subject, body) VALUES(1, 'software feedback', 'found it too slow');
INSERT INTO mail(docid, subject, body) VALUES(2, 'software feedback', 'no feedback');
INSERT INTO mail(docid, subject, body) VALUES(3, 'slow lunch order',  'was a software problem');

-- Example queries
SELECT * FROM mail WHERE subject MATCH 'software';    -- Selects rows 1 and 2
SELECT * FROM mail WHERE body    MATCH 'feedback';    -- Selects row 2
SELECT * FROM mail WHERE mail    MATCH 'software';    -- Selects rows 1, 2 and 3
SELECT * FROM mail WHERE mail    MATCH 'slow';        -- Selects rows 1 and 3
```

At first glance, the final two full-text queries in the example above seem to be syntactically incorrect, as there is a table name ("mail") used as an SQL expression. The reason this is acceptable is that each FTS table actually has a HIDDEN column with the same name as the table itself (in this case, "mail"). The value stored in this column is not meaningful to the application, but can be used as the left-hand operand to a MATCH operator. This special column may also be passed as an argument to the FTS auxiliary functions.

The following example illustrates the above. The expressions "docs", "docs.docs" and "main.docs.docs" all refer to column "docs". However, the expression "main.docs" does not refer to any column. It could be used to refer to a table, but a table name is not allowed in the context in which it is used below.

```
-- Example schema
CREATE VIRTUAL TABLE docs USING fts4(content);

-- Example queries
SELECT * FROM docs WHERE docs MATCH 'sqlite';              -- OK.
SELECT * FROM docs WHERE docs.docs MATCH 'sqlite';         -- OK.
SELECT * FROM docs WHERE main.docs.docs MATCH 'sqlite';    -- OK.
SELECT * FROM docs WHERE main.docs MATCH 'sqlite';         -- Error.
```

1.5. Summary

From the users point of view, FTS tables are similar to ordinary SQLite tables in many ways. Data may be added to, modified within and removed from FTS tables using the INSERT, UPDATE and DELETE commands just as it may be with ordinary tables. Similarly, the SELECT command may be used to query data. The following list summarizes the differences between FTS and ordinary tables:

1. As with all virtual table types, it is not possible to create indices or triggers attached to FTS tables. Nor is it possible to use the ALTER TABLE command to add extra columns to FTS tables (although it is possible to use ALTER TABLE to rename an FTS table).

2. Data-types specified as part of the "CREATE VIRTUAL TABLE" statement used to create an FTS table are ignored completely. Instead of the normal rules for applying type affinity to inserted values, all values inserted into FTS table columns (except the special rowid column) are converted to type TEXT before being stored.

3. FTS tables permit the special alias "docid" to be used to refer to the rowid column supported by all virtual tables.

4. The FTS MATCH operator is supported for queries based on the built-in full-text index.

5. The FTS auxiliary functions, snippet(), offsets(), and matchinfo() are available to support full-text queries.

6. Every FTS table has a hidden column with the same name as the table itself. The value contained in each row for the hidden column is a blob that is only useful as the left operand of a MATCH operator, or as the left-most argument to one of the FTS auxiliary functions.

2. Compiling and Enabling FTS3 and FTS4

Although FTS3 and FTS4 are included with the SQLite core source code, they are not enabled by default. To build SQLite with FTS functionality enabled, define the preprocessor macro SQLITE_ENABLE_FTS3 when compiling. New applications should also define the SQLITE_ENABLE_FTS3_PARENTHESIS macro to enable the enhanced query syntax (see below). Usually, this is done by adding the following two switches to the compiler command line:

```
-DSQLITE_ENABLE_FTS3
-DSQLITE_ENABLE_FTS3_PARENTHESIS
```

Note that enabling FTS3 also makes FTS4 available. There is not a separate SQLITE_ENABLE_FTS4 compile-time option. A build of SQLite either supports both FTS3 and FTS4 or it supports neither.

If using the amalgamation autoconf based build system, setting the CPPFLAGS environment variable while running the 'configure' script is an easy way to set these macros. For example, the following command:

```
CPPFLAGS="-DSQLITE_ENABLE_FTS3 -DSQLITE_ENABLE_FTS3_PARENTHESIS" ./configure <configure options>
```

where *<configure options>* are those options normally passed to the configure script, if any.

Because FTS3 and FTS4 are virtual tables, The SQLITE_ENABLE_FTS3 compile-time option is incompatible with the SQLITE_OMIT_VIRTUALTABLE option.

If a build of SQLite does not include the FTS modules, then any attempt to prepare an SQL statement to create an FTS3 or FTS4 table or to drop or access an existing FTS table in any way will fail. The error message returned will be similar to "no such module: ftsN" (where N is either 3 or 4).

If the C version of the ICU library is available, then FTS may also be compiled with the SQLITE_ENABLE_ICU pre-processor macro defined. Compiling with this macro enables an FTS tokenizer that uses the ICU library to split a document into terms (words) using the conventions for a specified language and locale.

```
-DSQLITE_ENABLE_ICU
```

3. Full-text Index Queries

The most useful thing about FTS tables is the queries that may be performed using the built-in full-text index. Full-text queries are performed by specifying a clause of the form "<column> MATCH <full-text query expression>" as part of the WHERE clause of a SELECT statement that reads data from an FTS table. Simple FTS queries that return all documents that contain a given term are described above. In that discussion the right-hand operand of the MATCH operator was assumed to be a string consisting of a single term. This section describes the more complex query types supported by FTS tables, and how they may be utilized by specifying a more complex query expression as the right-hand operand of a MATCH operator.

FTS tables support three basic query types:

- **Token or token prefix queries**. An FTS table may be queried for all documents that contain a specified term (the simple case described above), or for all documents that contain a term with a specified prefix. As we have seen, the query expression for a specific term is simply the term itself. The query expression used to search for a term prefix is the prefix itself with a '*' character appended to it. For example:

```
-- Virtual table declaration
CREATE VIRTUAL TABLE docs USING fts3(title, body);
```

```
-- Query for all documents containing the term "linux":
SELECT * FROM docs WHERE docs MATCH 'linux';

-- Query for all documents containing a term with the prefix "lin". This will match
-- all documents that contain "linux", but also those that contain terms "linear",
--"linker", "linguistic" and so on.
SELECT * FROM docs WHERE docs MATCH 'lin*';
```

Normally, a token or token prefix query is matched against the FTS table column specified as the left-hand side of the MATCH operator. Or, if the special column with the same name as the FTS table itself is specified, against all columns. This may be overridden by specifying a column-name followed by a ":" character before a basic term query. There may be space between the ":" and the term to query for, but not between the column-name and the ":" character. For example:

```
-- Query the database for documents for which the term "linux" appears in
-- the document title, and the term "problems" appears in either the title
-- or body of the document.
SELECT * FROM docs WHERE docs MATCH 'title:linux problems';

-- Query the database for documents for which the term "linux" appears in
-- the document title, and the term "driver" appears in the body of the document
-- ("driver" may also appear in the title, but this alone will not satisfy the
-- query criteria).
SELECT * FROM docs WHERE body MATCH 'title:linux driver';
```

If the FTS table is an FTS4 table (not FTS3), a token may also be prefixed with a "^" character. In this case, in order to match the token must appear as the very first token in any column of the matching row. Examples:

```
-- All documents for which "linux" is the first token of at least one
-- column.
SELECT * FROM docs WHERE docs MATCH '^linux';

-- All documents for which the first token in column "title" begins with "lin".
SELECT * FROM docs WHERE body MATCH 'title: ^lin*';
```

- **Phrase queries**. A phrase query is a query that retrieves all documents that contain a nominated set of terms or term prefixes in a specified order with no intervening tokens. Phrase queries are specified by enclosing a space separated sequence of terms or term prefixes in double quotes ("). For example:

```
-- Query for all documents that contain the phrase "linux applications".
SELECT * FROM docs WHERE docs MATCH '"linux applications"';

-- Query for all documents that contain a phrase that matches "lin* app*". As well as
-- "linux applications", this will match common phrases such as "linoleum appliances"
-- or "link apprentice".
SELECT * FROM docs WHERE docs MATCH '"lin* app*"';
```

- **NEAR queries**. A NEAR query is a query that returns documents that contain a two or more nominated terms or phrases within a specified proximity of each other (by default with 10 or less intervening terms). A NEAR query is specified by putting the keyword "NEAR" between two phrase, token or token prefix queries. To specify a proximity other than the default, an operator of the form "NEAR/<*N*>" may be used, where <*N*> is the maximum number of intervening terms allowed. For example:

```
-- Virtual table declaration.
CREATE VIRTUAL TABLE docs USING fts4();

-- Virtual table data.
INSERT INTO docs VALUES('SQLite is an ACID compliant embedded relational database management system');

-- Search for a document that contains the terms "sqlite" and "database" with
-- not more than 10 intervening terms. This matches the only document in
-- table docs (since there are only six terms between "SQLite" and "database"
```

```
-- in the document).
SELECT * FROM docs WHERE docs MATCH 'sqlite NEAR database';

-- Search for a document that contains the terms "sqlite" and "database" with
-- not more than 6 intervening terms. This also matches the only document in
-- table docs. Note that the order in which the terms appear in the document
-- does not have to be the same as the order in which they appear in the query.
SELECT * FROM docs WHERE docs MATCH 'database NEAR/6 sqlite';

-- Search for a document that contains the terms "sqlite" and "database" with
-- not more than 5 intervening terms. This query matches no documents.
SELECT * FROM docs WHERE docs MATCH 'database NEAR/5 sqlite';

-- Search for a document that contains the phrase "ACID compliant" and the term
-- "database" with not more than 2 terms separating the two. This matches the
-- document stored in table docs.
SELECT * FROM docs WHERE docs MATCH 'database NEAR/2 "ACID compliant"';

-- Search for a document that contains the phrase "ACID compliant" and the term
-- "sqlite" with not more than 2 terms separating the two. This also matches
-- the only document stored in table docs.
SELECT * FROM docs WHERE docs MATCH '"ACID compliant" NEAR/2 sqlite';
```

More than one NEAR operator may appear in a single query. In this case each pair of terms or phrases separated by a NEAR operator must appear within the specified proximity of each other in the document. Using the same table and data as in the block of examples above:

```
-- The following query selects documents that contains an instance of the term
-- "sqlite" separated by two or fewer terms from an instance of the term "acid",
-- which is in turn separated by two or fewer terms from an instance of the term
-- "relational".
SELECT * FROM docs WHERE docs MATCH 'sqlite NEAR/2 acid NEAR/2 relational';

-- This query matches no documents. There is an instance of the term "sqlite" with
-- sufficient proximity to an instance of "acid" but it is not sufficiently close
-- to an instance of the term "relational".
SELECT * FROM docs WHERE docs MATCH 'acid NEAR/2 sqlite NEAR/2 relational';
```

Phrase and NEAR queries may not span multiple columns within a row.

The three basic query types described above may be used to query the full-text index for the set of documents that match the specified criteria. Using the FTS query expression language it is possible to perform various set operations on the results of basic queries. There are currently three supported operations:

- The AND operator determines the **intersection** of two sets of documents.
- The OR operator calculates the **union** of two sets of documents.
- The NOT operator (or, if using the standard syntax, a unary "-" operator) may be used to compute the **relative complement** of one set of documents with respect to another.

The FTS modules may be compiled to use one of two slightly different versions of the full-text query syntax, the "standard" query syntax and the "enhanced" query syntax. The basic term, term-prefix, phrase and NEAR queries described above are the same in both versions of the syntax. The way in which set operations are specified is slightly different. The following two sub-sections describe the part of the two query syntaxes that pertains to set operations. Refer to the description of how to compile fts for compilation notes.

3.1. Set Operations Using The Enhanced Query Syntax

The enhanced query syntax supports the AND, OR and NOT binary set operators. Each of the two operands to an operator may be a basic FTS query, or the result of another AND, OR or NOT set operation. Operators must be entered using capital letters. Otherwise, they are interpreted as basic term queries instead of set operators.

The AND operator may be implicitly specified. If two basic queries appear with no operator separating them in an FTS query string, the results are the same as if the two basic queries were separated by an AND operator. For example, the query expression "implicit operator" is a more succinct version of "implicit AND operator".

```
-- Virtual table declaration
CREATE VIRTUAL TABLE docs USING fts3();

-- Virtual table data
INSERT INTO docs(docid, content) VALUES(1, 'a database is a software system');
INSERT INTO docs(docid, content) VALUES(2, 'sqlite is a software system');
INSERT INTO docs(docid, content) VALUES(3, 'sqlite is a database');

-- Return the set of documents that contain the term "sqlite", and the
-- term "database". This query will return the document with docid 3 only.
SELECT * FROM docs WHERE docs MATCH 'sqlite AND database';

-- Again, return the set of documents that contain both "sqlite" and
-- "database". This time, use an implicit AND operator. Again, document
-- 3 is the only document matched by this query.
SELECT * FROM docs WHERE docs MATCH 'database sqlite';

-- Query for the set of documents that contains either "sqlite" or "database".
-- All three documents in the database are matched by this query.
SELECT * FROM docs WHERE docs MATCH 'sqlite OR database';

-- Query for all documents that contain the term "database", but do not contain
-- the term "sqlite". Document 1 is the only document that matches this criteria.
SELECT * FROM docs WHERE docs MATCH 'database NOT sqlite';

-- The following query matches no documents. Because "and" is in lowercase letters,
-- it is interpreted as a basic term query instead of an operator. Operators must
-- be specified using capital letters. In practice, this query will match any documents
-- that contain each of the three terms "database", "and" and "sqlite" at least once.
-- No documents in the example data above match this criteria.
SELECT * FROM docs WHERE docs MATCH 'database and sqlite';
```

The examples above all use basic full-text term queries as both operands of the set operations demonstrated. Phrase and NEAR queries may also be used, as may the results of other set operations. When more than one set operation is present in an FTS query, the precedence of operators is as follows:

Operator	Enhanced Query Syntax Precedence
NOT	Highest precedence (tightest grouping).
AND	
OR	Lowest precedence (loosest grouping).

When using the enhanced query syntax, parenthesis may be used to override the default precedence of the various operators. For example:

```
-- Return the docid values associated with all documents that contain the
-- two terms "sqlite" and "database", and/or contain the term "library".
SELECT docid FROM docs WHERE docs MATCH 'sqlite AND database OR library';

-- This query is equivalent to the above.
SELECT docid FROM docs WHERE docs MATCH 'sqlite AND database'
  UNION
SELECT docid FROM docs WHERE docs MATCH 'library';

-- Query for the set of documents that contains the term "linux", and at least
-- one of the phrases "sqlite database" and "sqlite library".
SELECT docid FROM docs WHERE docs MATCH '("sqlite database" OR "sqlite library") AND linux';

-- This query is equivalent to the above.
SELECT docid FROM docs WHERE docs MATCH 'linux'
  INTERSECT
SELECT docid FROM (
  SELECT docid FROM docs WHERE docs MATCH '"sqlite library"'
    UNION
  SELECT docid FROM docs WHERE docs MATCH '"sqlite database"'
);
```

3.2. Set Operations Using The Standard Query Syntax

FTS query set operations using the standard query syntax are similar, but not identical, to set operations with the enhanced query syntax. There are four differences, as follows:

1. Only the implicit version of the AND operator is supported. Specifying the string "AND" as part of a standard query syntax query is interpreted as a term query for the set of documents containing the term "and".

2. Parenthesis are not supported.

3. The NOT operator is not supported. Instead of the NOT operator, the standard query syntax supports a unary "-" operator that may be applied to basic term and term-prefix queries (but not to phrase or NEAR queries). A term or term-prefix that has a unary "-" operator attached to it may not appear as an operand to an OR operator. An FTS query may not consist entirely of terms or term-prefix queries with unary "-" operators attached to them.

```
-- Search for the set of documents that contain the term "sqlite" but do
-- not contain the term "database".
SELECT * FROM docs WHERE docs MATCH 'sqlite -database';
```

4. The relative precedence of the set operations is different. In particular, using the standard query syntax the "OR" operator has a higher precedence than "AND". The precedence of operators when using the standard query syntax is:

Operator	Standard Query Syntax Precedence
Unary "-"	Highest precedence (tightest grouping).
OR	
AND	Lowest precedence (loosest grouping).

   The following example illustrates precedence of operators using the standard query syntax:

```
-- Search for documents that contain at least one of the terms "database"
-- and "sqlite", and also contain the term "library". Because of the differences
-- in operator precedences, this query would have a different interpretation using
-- the enhanced query syntax.
SELECT * FROM docs WHERE docs MATCH 'sqlite OR database library';
```

4. Auxiliary Functions - Snippet, Offsets and Matchinfo

The FTS3 and FTS4 modules provide three special SQL scalar functions that may be useful to the developers of full-text query systems: "snippet", "offsets" and "matchinfo". The purpose of the "snippet" and "offsets" functions is to allow the user to identify the location of queried terms in the returned documents. The "matchinfo" function provides the user with metrics that may be useful for filtering or sorting query results according to relevance.

The first argument to all three special SQL scalar functions must be the FTS hidden column of the FTS table that the function is applied to. The FTS hidden column is an automatically-generated column found on all FTS tables that has the same name as the FTS table itself. For example, given an FTS table named "mail":

```
SELECT offsets(mail) FROM mail WHERE mail MATCH <full-text query expression>;
SELECT snippet(mail) FROM mail WHERE mail MATCH <full-text query expression>;
SELECT matchinfo(mail) FROM mail WHERE mail MATCH <full-text query expression>;
```

The three auxiliary functions are only useful within a SELECT statement that uses the FTS table's full-text index. If used within a SELECT that uses the "query by rowid" or "linear scan" strategies, then the snippet and offsets both return an empty string, and the matchinfo function returns a blob value zero bytes in size.

All three auxiliary functions extract a set of "matchable phrases" from the FTS query expression to work with. The set of matchable phrases for a given query consists of all phrases (including unquoted tokens and token prefixes) in the expression except those that are prefixed with a unary "-" operator (standard syntax) or are part of a sub-expression that is used as the right-hand operand of a NOT operator.

With the following provisos, each series of tokens in the FTS table that matches one of the matchable phrases in the query expression is known as a "phrase match":

1. If a matchable phrase is part of a series of phrases connected by NEAR operators in the FTS query expression, then each phrase match must be sufficiently close to other phrase matches of the relevant types to satisfy the NEAR condition.
2. If the matchable phrase in the FTS query is restricted to matching data in a specified FTS table column, then only phrase matches that occur within that column are considered.

4.1. The Offsets Function

For a SELECT query that uses the full-text index, the offsets() function returns a text value containing a series of space-separated integers. For each term in each phrase match of the current row, there are four integers in the returned list. Each set of four integers is interpreted as follows:

Integer	Interpretation
0	The column number that the term instance occurs in (0 for the leftmost column of the FTS table, 1 for the next leftmost, etc.).
1	The term number of the matching term within the full-text query expression. Terms within a query expression are numbered starting from 0 in the order that they occur.
2	The byte offset of the matching term within the column.
3	The size of the matching term in bytes.

The following block contains examples that use the offsets function.

```
CREATE VIRTUAL TABLE mail USING fts3(subject, body);
INSERT INTO mail VALUES('hello world', 'This message is a hello world message.');
INSERT INTO mail VALUES('urgent: serious', 'This mail is seen as a more serious mail');

-- The following query returns a single row (as it matches only the first
-- entry in table "mail". The text returned by the offsets function is
-- "0 0 6 5 1 0 24 5".
--
-- The first set of four integers in the result indicate that column 0
-- contains an instance of term 0 ("world") at byte offset 6. The term instance
-- is 5 bytes in size. The second set of four integers shows that column 1
-- of the matched row contains an instance of term 0 ("world") at byte offset
-- 24. Again, the term instance is 5 bytes in size.
SELECT offsets(mail) FROM mail WHERE mail MATCH 'world';

-- The following query returns also matches only the first row in table "mail".
-- In this case the returned text is "1 0 5 7 1 0 30 7".
SELECT offsets(mail) FROM mail WHERE mail MATCH 'message';

-- The following query matches the second row in table "mail". It returns the
-- text "1 0 28 7 1 1 36 4". Only those occurrences of terms "serious" and "mail"
-- that are part of an instance of the phrase "serious mail" are identified; the
-- other occurrences of "serious" and "mail" are ignored.
SELECT offsets(mail) FROM mail WHERE mail MATCH '"serious mail"';
```

4.2. The Snippet Function

The snippet function is used to create formatted fragments of document text for display as part of a full-text query results report. The snippet function may be passed between one and six arguments, as follows:

Argument	Default Value	Description
0	N/A	The first argument to the snippet function must always be the FTS hidden column of the FTS table being queried and from which the snippet is to be taken. The FTS hidden column is an automatically generated column with the same name as the FTS table itself.
1	""	The "start match" text.
2	""	The "end match" text.
3	"..."	The "ellipses" text.
4	-1	The FTS table column number to extract the returned fragments of text from. Columns are numbered from left to right starting with zero. A negative value indicates that the text may be extracted from any column.
5	-15	The absolute value of this integer argument is used as the (approximate) number of tokens to include in the returned text value. The maximum allowable absolute value is 64. The value of this argument is referred to as *N* in the discussion below.

The snippet function first attempts to find a fragment of text consisting of |*N*| tokens within the current row that contains at least one phrase match for each matchable phrase matched somewhere in the current row, where |*N*| is the absolute value of the sixth argument passed to the snippet function. If the text stored in a single column contains less than |*N*| tokens, then the entire column value is considered. Text fragments may not span multiple columns.

If such a text fragment can be found, it is returned with the following modifications:

- If the text fragment does not begin at the start of a column value, the "ellipses" text is prepended to it.
- If the text fragment does not finish at the end of a column value, the "ellipses" text is appended to it.
- For each token in the text fragment that is part of a phrase match, the "start match" text is inserted into the fragment before the token, and the "end match" text is inserted immediately after it.

If more than one such fragment can be found, then fragments that contain a larger number of "extra" phrase matches are favored. The start of the selected text fragment may be moved a few tokens forward or backward to attempt to concentrate the phrase matches toward the center of the fragment.

Assuming *N* is a positive value, if no fragments can be found that contain a phrase match corresponding to each matchable phrase, the snippet function attempts to find two fragments of approximately *N*/2 tokens that between them contain at least one phrase match for each matchable phrase matched by the current row. If this fails, attempts are made to find three fragments of *N*/3 tokens each and finally four *N*/4 token fragments. If a set of four fragments cannot be found that encompasses the required phrase matches, the four fragments of *N*/4 tokens that provide the best coverage are selected.

If *N* is a negative value, and no single fragment can be found containing the required phrase matches, the snippet function searches for two fragments of |*N*| tokens each, then three, then four. In other words, if the specified value of *N* is negative, the sizes of the fragments is not decreased if more than one fragment is required to provide the desired phrase match coverage.

After the *M* fragments have been located, where *M* is between two and four as described in the paragraphs above, they are joined together in sorted order with the "ellipses" text separating them. The three modifications enumerated earlier are performed on the text before it is returned.

```
Note: In this block of examples, newlines and whitespace characters have
been inserted into the document inserted into the FTS table, and the expected
results described in SQL comments. This is done to enhance readability only,
they would not be present in actual SQLite commands or output.

-- Create and populate an FTS table.
```

```
CREATE VIRTUAL TABLE text USING fts4();
INSERT INTO text VALUES('
  During 30 Nov-1 Dec, 2-3oC drops. Cool in the upper portion, minimum temperature 14-16oC
  and cool elsewhere, minimum temperature 17-20oC. Cold to very cold on mountaintops,
  minimum temperature 6-12oC. Northeasterly winds 15-30 km/hr. After that, temperature
  increases. Northeasterly winds 15-30 km/hr.
');

-- The following query returns the text value:
--
--   "<b>...</b>cool elsewhere, minimum temperature 17-20oC. <b>Cold</b> to very
--    <b>cold</b> on mountaintops, minimum temperature 6<b>...</b>".
--
SELECT snippet(text) FROM text WHERE text MATCH 'cold';

-- The following query returns the text value:
--
--   "...the upper portion, [minimum] [temperature] 14-16oC and cool elsewhere,
--    [minimum] [temperature] 17-20oC. Cold..."
--
SELECT snippet(text, '[', ']', '...') FROM text WHERE text MATCH '"min* tem*"'
```

4.3. The Matchinfo Function

The matchinfo function returns a blob value. If it is used within a query that does not use the full-text index (a "query by rowid" or "linear scan"), then the blob is zero bytes in size. Otherwise, the blob consists of zero or more 32-bit unsigned integers in machine byte-order. The exact number of integers in the returned array depends on both the query and the value of the second argument (if any) passed to the matchinfo function.

The matchinfo function is called with either one or two arguments. As for all auxiliary functions, the first argument must be the special FTS hidden column. The second argument, if it is specified, must be a text value comprised only of the characters 'p', 'c', 'n', 'a', 'l', 's', 'x', 'y' and 'b'. If no second argument is explicitly supplied, it defaults to "pcx". The second argument is referred to as the "format string" below.

Characters in the matchinfo format string are processed from left to right. Each character in the format string causes one or more 32-bit unsigned integer values to be added to the returned array. The "values" column in the following table contains the number of integer values appended to the output buffer for each supported format string character. In the formula given, *cols* is the number of columns in the FTS table, and *phrases* is the number of matchable phrases in the query.

Character	Values	Description
p	1	The number of matchable phrases in the query.
c	1	The number of user defined columns in the FTS table (i.e. not including the docid or the FTS hidden column).
x	3 * *cols* * *phrases*	For each distinct combination of a phrase and table column, the following three values: • In the current row, the number of times the phrase appears in the column. • The total number of times the phrase appears in the column in all rows in the FTS table. • The total number of rows in the FTS table for which the column contains at least one instance of the phrase. The first set of three values corresponds to the left-most column of the table (column 0) and the left-most matchable phrase in the query (phrase 0). If the table has more than one column, the second set of three values in the output array correspond to phrase 0 and column 1. Followed by phrase 0, column 2 and so on for all columns of the table. And so on for phrase 1, column 0, then phrase 1, column 1 etc. In other words, the data for

occurrences of phrase *p* in column *c* may be found using the following formula:

```
hits_this_row  = array[3 * (c + p*cols) + 0]
hits_all_rows  = array[3 * (c + p*cols) + 1]
docs_with_hits = array[3 * (c + p*cols) + 2]
```

y	*cols * phrases*	For each distinct combination of a phrase and table column, the number of usable phrase matches that appear in the column. This is usually identical to the first value in each set of three returned by the matchinfo 'x' flag. However, the number of hits reported by the 'y' flag is zero for any phrase that is part of a sub-expression that does not match the current row. This makes a difference for expressions that contain AND operators that are descendants of OR operators. For example, consider the expression:
		
		`a OR (b AND c)`
		
		and the document:
		
		`"a c d"`
		
		The matchinfo 'x' flag would report a single hit for the phrases "a" and "c". However, the 'y' directive reports the number of hits for "c" as zero, as it is part of a sub-expression that does not match the document - (b AND c). For queries that do not contain AND operators descended from OR operators, the result values returned by 'y' are always the same as those returned by 'x'.
		
		The first value in the array of integer values corresponds to the leftmost column of the table (column 0) and the first phrase in the query (phrase 0). The values corresponding to other column/phrase combinations may be located using the following formula:
		
		`hits_for_phrase_p_column_c  = array[c + p*cols]`
		
		For queries that use OR expressions, or those that use LIMIT or return many rows, the 'y' matchinfo option may be faster than 'x'.

b	*((cols+31)/32) * phrases*	The matchinfo 'b' flag provides similar information to the matchinfo 'y' flag, but in a more compact form. Instead of the precise number of hits, 'b' provides a single boolean flag for each phrase/column combination. If the phrase is present in the column at least once (i.e. if the corresponding integer output of 'y' would be non-zero), the corresponding flag is set. Otherwise cleared. If the table has 32 or fewer columns, a single unsigned integer is output for each phrase in the query. The least significant bit of the integer is set if the phrase appears at least once in column 0. The second least significant bit is set if the phrase appears once or more in column 1. And so on. If the table has more than 32 columns, an extra integer is added to the output of each phrase for each extra 32 columns or part thereof. Integers corresponding to the same phrase are clumped together. For example, if a table with 45 columns is queried for two phrases, 4 integers are output. The first corresponds to phrase 0 and columns 0-31 of the table. The second integer contains data for phrase 0 and columns 32-44, and so on.

		For example, if nCol is the number of columns in the table, to determine if phrase p is present in column c: `p_is_in_c = array[p * ((nCol+31)/32)] & (1 << (c % 32))`
n	1	The number of rows in the FTS4 table. This value is only available when querying FTS4 tables, not FTS3.
a	*cols*	For each column, the average number of tokens in the text values stored in the column (considering all rows in the FTS4 table). This value is only available when querying FTS4 tables, not FTS3.
l	*cols*	For each column, the length of the value stored in the current row of the FTS4 table, in tokens. This value is only available when querying FTS4 tables, not FTS3. And only if the "matchinfo=fts3" directive was not specified as part of the "CREATE VIRTUAL TABLE" statement used to create the FTS4 table.
s	*cols*	For each column, the length of the longest subsequence of phrase matches that the column value has in common with the query text. For example, if a table column contains the text 'a b c d e' and the query is 'a c "d e"', then the length of the longest common subsequence is 2 (phrase "c" followed by phrase "d e").

For example:

```
-- Create and populate an FTS4 table with two columns:
CREATE VIRTUAL TABLE t1 USING fts4(a, b);
INSERT INTO t1 VALUES('transaction default models default', 'Non transaction reads');
INSERT INTO t1 VALUES('the default transaction', 'these semantics present');
INSERT INTO t1 VALUES('single request', 'default data');

-- In the following query, no format string is specified and so it defaults
-- to "pcx". It therefore returns a single row consisting of a single blob
-- value 80 bytes in size (20 32-bit integers - 1 for "p", 1 for "c" and
-- 3*2*3 for "x"). If each block of 4 bytes in the blob is interpreted
-- as an unsigned integer in machine byte-order, the values will be:
--
--     3 2  1 3 2  0 1 1  1 2 2  0 1 1  0 0 0  1 1 1
--
-- The row returned corresponds to the second entry inserted into table t1.
-- The first two integers in the blob show that the query contained three
-- phrases and the table being queried has two columns. The next block of
-- three integers describes column 0 (in this case column "a") and phrase
-- 0 (in this case "default"). The current row contains 1 hit for "default"
-- in column 0, of a total of 3 hits for "default" that occur in column
-- 0 of any table row. The 3 hits are spread across 2 different rows.
--
-- The next set of three integers (0 1 1) pertain to the hits for "default"
-- in column 1 of the table (0 in this row, 1 in all rows, spread across
-- 1 rows).
--
SELECT matchinfo(t1) FROM t1 WHERE t1 MATCH 'default transaction "these semantics"';

-- The format string for this query is "ns". The output array will therefore
-- contain 3 integer values - 1 for "n" and 2 for "s". The query returns
-- two rows (the first two rows in the table match). The values returned are:
--
--     3  1 1
--     3  2 0
--
-- The first value in the matchinfo array returned for both rows is 3 (the
-- number of rows in the table). The following two values are the lengths
-- of the longest common subsequence of phrase matches in each column.
SELECT matchinfo(t1, 'ns') FROM t1 WHERE t1 MATCH 'default transaction';
```

The matchinfo function is much faster than either the snippet or offsets functions. This is because the implementation of both snippet and offsets is required to retrieve the documents being analyzed from disk, whereas all data required by matchinfo is available as part of the same portions of the full-text index that are required to implement the full-text query itself. This means that of the following two queries, the first may be an order of magnitude faster than the second:

```
SELECT docid, matchinfo(tbl) FROM tbl WHERE tbl MATCH <query expression>;
SELECT docid, offsets(tbl) FROM tbl WHERE tbl MATCH <query expression>;
```

The matchinfo function provides all the information required to calculate probabilistic "bag-of-words" relevancy scores such as Okapi BM25/BM25F that may be used to order results in a full-text search application. Appendix A of this document, "search application tips", contains an example of using the matchinfo() function efficiently.

5. Fts4aux - Direct Access to the Full-Text Index

As of version 3.7.6 (2011-04-12), SQLite includes a new virtual table module called "fts4aux", which can be used to inspect the full-text index of an existing FTS table directly. Despite its name, fts4aux works just as well with FTS3 tables as it does with FTS4 tables. Fts4aux tables are read-only. The only way to modify the contents of an fts4aux table is by modifying the contents of the associated FTS table. The fts4aux module is automatically included in all builds that include FTS.

An fts4aux virtual table is constructed with one or two arguments. When used with a single argument, that argument is the unqualified name of the FTS table that it will be used to access. To access a table in a different database (for example, to create a TEMP fts4aux table that will access an FTS3 table in the MAIN database) use the two-argument form and give the name of the target database (ex: "main") in the first argument and the name of the FTS3/4 table as the second argument. (The two-argument form of fts4aux was added for SQLite version 3.7.17 (2013-05-20) and will throw an error in prior releases.) For example:

```
-- Create an FTS4 table
CREATE VIRTUAL TABLE ft USING fts4(x, y);

-- Create an fts4aux table to access the full-text index for table "ft"
CREATE VIRTUAL TABLE ft_terms USING fts4aux(ft);

-- Create a TEMP fts4aux table accessing the "ft" table in "main"
CREATE VIRTUAL TABLE temp.ft_terms_2 USING fts4aux(main,ft);
```

For each term present in the FTS table, there are between 2 and N+1 rows in the fts4aux table, where N is the number of user-defined columns in the associated FTS table. An fts4aux table always has the same four columns, as follows, from left to right:

Column Name	Column Contents
term	Contains the text of the term for this row.
col	This column may contain either the text value '*' (i.e. a single character, U+002a) or an integer between 0 and N-1, where N is again the number of user-defined columns in the corresponding FTS table.
documents	This column always contains an integer value greater than zero. If the "col" column contains the value '*', then this column contains the number of rows of the FTS table that contain at least one instance of the term (in any column). If col contains an integer value, then this column contains the number of rows of the FTS table that contain at least one instance of the term in the column identified by the col value. As usual, the columns of the FTS table are numbered from left to right, starting with zero.
occurrences	This column also always contains an integer value greater than zero.

	If the "col" column contains the value '*', then this column contains the total number of instances of the term in all rows of the FTS table (in any column). Otherwise, if col contains an integer value, then this column contains the total number of instances of the term that appear in the FTS table column identified by the col value.
languageid *(hidden)*	This column determines which languageid is used to extract vocabulary from the FTS3/4 table. The default value for languageid is 0. If an alternative language is specified in WHERE clause constraints, then that alternative is used instead of 0. There can only be a single languageid per query. In other words, the WHERE clause cannot contain a range constraint or IN operator on the languageid.

For example, using the tables created above:

```
INSERT INTO ft(x, y) VALUES('Apple banana', 'Cherry');
INSERT INTO ft(x, y) VALUES('Banana Date Date', 'cherry');
INSERT INTO ft(x, y) VALUES('Cherry Elderberry', 'Elderberry');

-- The following query returns this data:
--
--     apple       |  *  |  1  |  1
--     apple       |  0  |  1  |  1
--     banana      |  *  |  2  |  2
--     banana      |  0  |  2  |  2
--     cherry      |  *  |  3  |  3
--     cherry      |  0  |  1  |  1
--     cherry      |  1  |  2  |  2
--     date        |  *  |  1  |  2
--     date        |  0  |  1  |  2
--     elderberry  |  *  |  1  |  2
--     elderberry  |  0  |  1  |  1
--     elderberry  |  1  |  1  |  1
--
SELECT term, col, documents, occurrences FROM ft_terms;
```

In the example, the values in the "term" column are all lower case, even though they were inserted into table "ft" in mixed case. This is because an fts4aux table contains the terms as extracted from the document text by the tokenizer. In this case, since table "ft" uses the simple tokenizer, this means all terms have been folded to lower case. Also, there is (for example) no row with column "term" set to "apple" and column "col" set to 1. Since there are no instances of the term "apple" in column 1, no row is present in the fts4aux table.

During a transaction, some of the data written to an FTS table may be cached in memory and written to the database only when the transaction is committed. However the implementation of the fts4aux module is only able to read data from the database. In practice this means that if an fts4aux table is queried from within a transaction in which the associated FTS table has been modified, the results of the query are likely to reflect only a (possibly empty) subset of the changes made.

6. FTS4 Options

If the "CREATE VIRTUAL TABLE" statement specifies module FTS4 (not FTS3), then special directives - FTS4 options - similar to the "tokenize=*" option may also appear in place of column names. An FTS4 option consists of the option name, followed by an "=" character, followed by the option value. The option value may optionally be enclosed in single or double quotes, with embedded quote characters escaped in the same way as for SQL literals. There may not be whitespace on either side of the "=" character. For example, to create an FTS4 table with the value of option "matchinfo" set to "fts3":

```
-- Create a reduced-footprint FTS4 table.
CREATE VIRTUAL TABLE papers USING fts4(author, document, matchinfo=fts3);
```

FTS4 currently supports the following options:

Option	Interpretation
compress	The compress option is used to specify the compress function. It is an error to specify a compress function without also specifying an uncompress function. See below for details.
content	The content allows the text being indexed to be stored in a separate table distinct from the FTS4 table, or even outside of SQLite.
languageid	The languageid option causes the FTS4 table to have an additional hidden integer column that identifies the language of the text contained in each row. The use of the languageid option allows the same FTS4 table to hold text in multiple languages or scripts, each with different tokenizer rules, and to query each language independently of the others.
matchinfo	When set to the value "fts3", the matchinfo option reduces the amount of information stored by FTS4 with the consequence that the "l" option of matchinfo() is no longer available.
notindexed	This option is used to specify the name of a column for which data is not indexed. Values stored in columns that are not indexed are not matched by MATCH queries. Nor are they recognized by auxiliary functions. A single CREATE VIRTUAL TABLE statement may have any number of notindexed options.
order	The "order" option may be set to either "DESC" or "ASC" (in upper or lower case). If it is set to "DESC", then FTS4 stores its data in such a way as to optimize returning results in descending order by docid. If it is set to "ASC" (the default), then the data structures are optimized for returning results in ascending order by docid. In other words, if many of the queries run against the FTS4 table use "ORDER BY docid DESC", then it may improve performance to add the "order=desc" option to the CREATE VIRTUAL TABLE statement.
prefix	This option may be set to a comma-separated list of positive non-zero integers. For each integer N in the list, a separate index is created in the database file to optimize prefix queries where the query term is N bytes in length, not including the '*' character, when encoded using UTF-8. See below for details.
uncompress	This option is used to specify the uncompress function. It is an error to specify an uncompress function without also specifying a compress function. See below for details.

When using FTS4, specifying a column name that contains an "=" character and is not either a "tokenize=*" specification or a recognized FTS4 option is an error. With FTS3, the first token in the unrecognized directive is interpreted as a column name. Similarly, specifying multiple "tokenize=*" directives in a single table declaration is an error when using FTS4, whereas the second and subsequent "tokenize=*" directives are interpreted as column names by FTS3. For example:

```
-- An error. FTS4 does not recognize the directive "xyz=abc".
CREATE VIRTUAL TABLE papers USING fts4(author, document, xyz=abc);

-- Create an FTS3 table with three columns - "author", "document"
-- and "xyz".
CREATE VIRTUAL TABLE papers USING fts3(author, document, xyz=abc);

-- An error. FTS4 does not allow multiple tokenize=* directives
CREATE VIRTUAL TABLE papers USING fts4(tokenize=porter, tokenize=simple);

-- Create an FTS3 table with a single column named "tokenize". The
-- table uses the "porter" tokenizer.
CREATE VIRTUAL TABLE papers USING fts3(tokenize=porter, tokenize=simple);

-- An error. Cannot create a table with two columns named "tokenize".
CREATE VIRTUAL TABLE papers USING fts3(tokenize=porter, tokenize=simple, tokenize=icu);
```

6.1. The compress= and uncompress= options

The compress and uncompress options allow FTS4 content to be stored in the database in a compressed form. Both options should be set to the name of an SQL scalar function registered using sqlite3_create_function() that accepts a single argument.

The compress function should return a compressed version of the value passed to it as an argument. Each time data is written to the FTS4 table, each column value is passed to the compress function and the result value stored in the database. The compress function may return any type of SQLite value (blob, text, real, integer or null).

The uncompress function should uncompress data previously compressed by the compress function. In other words, for all SQLite values X, it should be true that uncompress(compress(X)) equals X. When data that has been compressed by the compress function is read from the database by FTS4, it is passed to the uncompress function before it is used.

If the specified compress or uncompress functions do not exist, the table may still be created. An error is not returned until the FTS4 table is read (if the uncompress function does not exist) or written (if it is the compress function that does not exist).

```
-- Create an FTS4 table that stores data in compressed form. This
-- assumes that the scalar functions zip() and unzip() have been (or
-- will be) added to the database handle.
CREATE VIRTUAL TABLE papers USING fts4(author, document, compress=zip, uncompress=unzip);
```

When implementing the compress and uncompress functions it is important to pay attention to data types. Specifically, when a user reads a value from a compressed FTS table, the value returned by FTS is exactly the same as the value returned by the uncompress function, including the data type. If that data type is not the same as the data type of the original value as passed to the compress function (for example if the uncompress function is returning BLOB when compress was originally passed TEXT), then the users query may not function as expected.

6.2. The content= option

The content option allows FTS4 to forego storing the text being indexed. The content option can be used in two ways:

- The indexed documents are not stored within the SQLite database at all (a "contentless" FTS4 table), or
- The indexed documents are stored in a database table created and managed by the user (an "external content" FTS4 table).

Because the indexed documents themselves are usually much larger than the full-text index, the content option can be used to achieve significant space savings.

6.2.1. Contentless FTS4 Tables

In order to create an FTS4 table that does not store a copy of the indexed documents at all, the content option should be set to an empty string. For example, the following SQL creates such an FTS4 table with three columns - "a", "b", and "c":

```
CREATE VIRTUAL TABLE t1 USING fts4(content="", a, b, c);
```

Data can be inserted into such an FTS4 table using an INSERT statements. However, unlike ordinary FTS4 tables, the user must supply an explicit integer docid value. For example:

```
-- This statement is Ok:
```

```
INSERT INTO t1(docid, a, b, c) VALUES(1, 'a b c', 'd e f', 'g h i');

-- This statement causes an error, as no docid value has been provided:
INSERT INTO t1(a, b, c) VALUES('j k l', 'm n o', 'p q r');
```

It is not possible to UPDATE or DELETE a row stored in a contentless FTS4 table. Attempting to do so is an error.

Contentless FTS4 tables also support SELECT statements. However, it is an error to attempt to retrieve the value of any table column other than the docid column. The auxiliary function matchinfo() may be used, but snippet() and offsets() may not. For example:

```
-- The following statements are Ok:
SELECT docid FROM t1 WHERE t1 MATCH 'xxx';
SELECT docid FROM t1 WHERE a MATCH 'xxx';
SELECT matchinfo(t1) FROM t1 WHERE t1 MATCH 'xxx';

-- The following statements all cause errors, as the value of columns
-- other than docid are required to evaluate them.
SELECT * FROM t1;
SELECT a, b FROM t1 WHERE t1 MATCH 'xxx';
SELECT docid FROM t1 WHERE a LIKE 'xxx%';
SELECT snippet(t1) FROM t1 WHERE t1 MATCH 'xxx';
```

Errors related to attempting to retrieve column values other than docid are runtime errors that occur within sqlite3_step(). In some cases, for example if the MATCH expression in a SELECT query matches zero rows, there may be no error at all even if a statement does refer to column values other than docid.

6.2.2. External Content FTS4 Tables

An "external content" FTS4 table is similar to a contentless table, except that if evaluation of a query requires the value of a column other than docid, FTS4 attempts to retrieve that value from a table (or view, or virtual table) nominated by the user (hereafter referred to as the "content table"). The FTS4 module never writes to the content table, and writing to the content table does not affect the full-text index. It is the responsibility of the user to ensure that the content table and the full-text index are consistent.

An external content FTS4 table is created by setting the content option to the name of a table (or view, or virtual table) that may be queried by FTS4 to retrieve column values when required. If the nominated table does not exist, then an external content table behaves in the same way as a contentless table. For example:

```
CREATE TABLE t2(id INTEGER PRIMARY KEY, a, b, c);
CREATE VIRTUAL TABLE t3 USING fts4(content="t2", a, c);
```

Assuming the nominated table does exist, then its columns must be the same as or a superset of those defined for the FTS table. The external table must also be in the same database file as the FTS table. In other words, The external table cannot be in a different database file connected using ATTACH nor may one of the FTS table and the external content be in the TEMP database when the other is in a persistent database file such as MAIN.

When a users query on the FTS table requires a column value other than docid, FTS attempts to read the requested value from the corresponding column of the row in the content table with a rowid value equal to the current FTS docid. Only the subset of content-table columns duplicated in the FTS/34 table declaration can be queried for - to retrieve values from any other columns the content table must be queried directly. Or, if such a row cannot be found in the content table, a NULL value is used instead. For example:

```
CREATE TABLE t2(id INTEGER PRIMARY KEY, a, b, c);
CREATE VIRTUAL TABLE t3 USING fts4(content="t2", b, c);

INSERT INTO t2 VALUES(2, 'a b', 'c d', 'e f');
INSERT INTO t2 VALUES(3, 'g h', 'i j', 'k l');
INSERT INTO t3(docid, b, c) SELECT id, b, c FROM t2;
-- The following query returns a single row with two columns containing
-- the text values "i j" and "k l".
--
```

```
-- The query uses the full-text index to discover that the MATCH
-- term matches the row with docid=3. It then retrieves the values
-- of columns b and c from the row with rowid=3 in the content table
-- to return.
--
SELECT * FROM t3 WHERE t3 MATCH 'k';

-- Following the UPDATE, the query still returns a single row, this
-- time containing the text values "xxx" and "yyy". This is because the
-- full-text index still indicates that the row with docid=3 matches
-- the FTS4 query 'k', even though the documents stored in the content
-- table have been modified.
--
UPDATE t2 SET b = 'xxx', c = 'yyy' WHERE rowid = 3;
SELECT * FROM t3 WHERE t3 MATCH 'k';

-- Following the DELETE below, the query returns one row containing two
-- NULL values. NULL values are returned because FTS is unable to find
-- a row with rowid=3 within the content table.
--
DELETE FROM t2;
SELECT * FROM t3 WHERE t3 MATCH 'k';
```

When a row is deleted from an external content FTS4 table, FTS4 needs to retrieve the column values of the row being deleted from the content table. This is so that FTS4 can update the full-text index entries for each token that occurs within the deleted row to indicate that row has been deleted. If the content table row cannot be found, or if it contains values inconsistent with the contents of the FTS index, the results can be difficult to predict. The FTS index may be left containing entries corresponding to the deleted row, which can lead to seemingly nonsensical results being returned by subsequent SELECT queries. The same applies when a row is updated, as internally an UPDATE is the same as a DELETE followed by an INSERT.

This means that in order to keep an FTS in sync with an external content table, any UPDATE or DELETE operations must be applied first to the FTS table, and then to the external content table. For example:

```
CREATE TABLE t1_real(id INTEGER PRIMARY KEY, a, b, c, d);
CREATE VIRTUAL TABLE t1_fts USING fts4(content="t1_real", b, c);

-- This works. When the row is removed from the FTS table, FTS retrieves
-- the row with rowid=123 and tokenizes it in order to determine the entries
-- that must be removed from the full-text index.
--
DELETE FROM t1_fts WHERE rowid = 123;
DELETE FROM t1_real WHERE rowid = 123;

-- This does not work. By the time the FTS table is updated, the row
-- has already been deleted from the underlying content table. As a result
-- FTS is unable to determine the entries to remove from the FTS index and
-- so the index and content table are left out of sync.
--
DELETE FROM t1_real WHERE rowid = 123;
DELETE FROM t1_fts WHERE rowid = 123;
```

Instead of writing separately to the full-text index and the content table, some users may wish to use database triggers to keep the full-text index up to date with respect to the set of documents stored in the content table. For example, using the tables from earlier examples:

```
CREATE TRIGGER t2_bu BEFORE UPDATE ON t2 BEGIN
  DELETE FROM t3 WHERE docid=old.rowid;
END;
CREATE TRIGGER t2_bd BEFORE DELETE ON t2 BEGIN
  DELETE FROM t3 WHERE docid=old.rowid;
END;

CREATE TRIGGER t2_au AFTER UPDATE ON t2 BEGIN
  INSERT INTO t3(docid, b, c) VALUES(new.rowid, new.b, new.c);
END;
CREATE TRIGGER t2_ai AFTER INSERT ON t2 BEGIN
```

```
  INSERT INTO t3(docid, b, c) VALUES(new.rowid, new.b, new.c);
END;
```

The DELETE trigger must be fired before the actual delete takes place on the content table. This is so that FTS4 can still retrieve the original values in order to update the full-text index. And the INSERT trigger must be fired after the new row is inserted, so as to handle the case where the rowid is assigned automatically within the system. The UPDATE trigger must be split into two parts, one fired before and one after the update of the content table, for the same reasons.

The FTS4 "rebuild" command deletes the entire full-text index and rebuilds it based on the current set of documents in the content table. Assuming again that "t3" is the name of the external content FTS4 table, the rebuild command looks like this:

```
INSERT INTO t3(t3) VALUES('rebuild');
```

This command may also be used with ordinary FTS4 tables, for example if the implementation of the tokenizer changes. It is an error to attempt to rebuild the full-text index maintained by a contentless FTS4 table, since no content will be available to do the rebuilding.

6.3. The languageid= option

When the languageid option is present, it specifies the name of another hidden column that is added to the FTS4 table and which is used to specify the language stored in each row of the FTS4 table. The name of the languageid hidden column must be distinct from all other column names in the FTS4 table. Example:

```
CREATE VIRTUAL TABLE t1 USING fts4(x, y, languageid="lid")
```

The default value of a languageid column is 0. Any value inserted into a languageid column is converted to a 32-bit (not 64) signed integer.

By default, FTS queries (those that use the MATCH operator) consider only those rows with the languageid column set to 0. To query for rows with other languageid values, a constraint of the form "

= " must be added to the queries WHERE clause. For example:

```
SELECT * FROM t1 WHERE t1 MATCH 'abc' AND lid=5;
```

It is not possible for a single FTS query to return rows with different languageid values. The results of adding WHERE clauses that use other operators (e.g. lid!=5, or lid<=5) are undefined.

If the content option is used along with the languageid option, then the named languageid column must exist in the content= table (subject to the usual rules - if a query never needs to read the content table then this restriction does not apply).

When the languageid option is used, SQLite invokes the xLanguageid() on the sqlite3_tokenizer_module object immediately after the object is created in order to pass in the language id that the tokenizer should use. The xLanguageid() method will never be called more than once for any single tokenizer object. The fact that different languages might be tokenized differently is one reason why no single FTS query can return rows with different languageid values.

6.4. The matchinfo= option

The matchinfo option may only be set to the value "fts3". Attempting to set matchinfo to anything other than "fts3" is an error. If this option is specified, then some of the extra information stored by FTS4 is omitted. This reduces the amount of disk space consumed by an FTS4 table until it is almost the same as the amount that would be used by the equivalent FTS3 table, but also means that the data accessed by passing the 'l' flag to the matchinfo() function is not available.

6.5. The notindexed= option

Normally, the FTS module maintains an inverted index of all terms in all columns of the table. This option is used to specify the name of a column for which entries should not be added to the index. Multiple "notindexed" options may be used to specify that multiple columns should be omitted from the index. For example:

```
-- Create an FTS4 table for which only the contents of columns c2 and c4
-- are tokenized and added to the inverted index.
CREATE VIRTUAL TABLE t1 USING fts4(c1, c2, c3, c4, notindexed=c1, notindexed=c3);
```

Values stored in unindexed columns are not eligible to match MATCH operators. They do not influence the results of the offsets() or matchinfo() auxiliary functions. Nor will the snippet() function ever return a snippet based on a value stored in an unindexed column.

6.6. The prefix= option

The FTS4 prefix option causes FTS to index term prefixes of specified lengths in the same way that it always indexes complete terms. The prefix option must be set to a comma separated list of positive non-zero integers. For each value N in the list, prefixes of length N bytes (when encoded using UTF-8) are indexed. FTS4 uses term prefix indexes to speed up prefix queries. The cost, of course, is that indexing term prefixes as well as complete terms increases the database size and slows down write operations on the FTS4 table.

Prefix indexes may be used to optimize prefix queries in two cases. If the query is for a prefix of N bytes, then a prefix index created with "prefix=N" provides the best optimization. Or, if no "prefix=N" index is available, a "prefix=N+1" index may be used instead. Using a "prefix=N+1" index is less efficient than a "prefix=N" index, but is better than no prefix index at all.

```
-- Create an FTS4 table with indexes to optimize 2 and 4 byte prefix queries.
CREATE VIRTUAL TABLE t1 USING fts4(c1, c2, prefix="2,4");

-- The following two queries are both optimized using the prefix indexes.
SELECT * FROM t1 WHERE t1 MATCH 'ab*';
SELECT * FROM t1 WHERE t1 MATCH 'abcd*';

-- The following two queries are both partially optimized using the prefix
-- indexes. The optimization is not as pronounced as it is for the queries
-- above, but still an improvement over no prefix indexes at all.
SELECT * FROM t1 WHERE t1 MATCH 'a*';
SELECT * FROM t1 WHERE t1 MATCH 'abc*';
```

7. Special Commands For FTS3 and FTS4

Special INSERT operates can be used to issue commands to FTS3 and FTS4 tables. Every FTS3 and FTS4 has a hidden, read-only column which is the same name as the table itself. INSERTs into this hidden column are interpreted as commands to the FTS3/4 table. For a table with the name "xyz" the following commands are supported:

- INSERT INTO xyz(xyz) VALUES('optimize');
- INSERT INTO xyz(xyz) VALUES('rebuild');
- INSERT INTO xyz(xyz) VALUES('integrity-check');
- INSERT INTO xyz(xyz) VALUES('merge=X,Y');
- INSERT INTO xyz(xyz) VALUES('automerge=N');

7.1. The "optimize" command

The "optimize" command causes FTS3/4 to merge together all of its inverted index b-trees into one large and complete b-tree. Doing an optimize will make subsequent queries run faster since there are fewer b-trees to search, and it may reduce disk usage by coalescing redundant entries. However, for a large FTS table, running optimize can be as expensive as running VACUUM. The optimize command essentially has to read and write the entire FTS table, resulting in a large transaction.

In batch-mode operation, where an FTS table is initially built up using a large number of INSERT operations, then queried repeatedly without further changes, it is often a good idea to run "optimize" after the last INSERT and before the first query.

7.2. The "rebuild" command

The "rebuild" command causes SQLite to discard the entire FTS3/4 table and then rebuild it again from original text. The concept is similar to REINDEX, only that it applies to an FTS3/4 table instead of an ordinary index.

The "rebuild" command should be run whenever the implementation of a custom tokenizer changes, so that all content can be retokenized. The "rebuild" command is also useful when using the FTS4 content option after changes have been made to the original content table.

7.3. The "integrity-check" command

The "integrity-check" command causes SQLite to read and verify the accuracy of all inverted indices in an FTS3/4 table by comparing those inverted indices against the original content. The "integrity-check" command silently succeeds if the inverted indices are all ok, but will fail with an SQLITE_CORRUPT error if any problems are found.

The "integrity-check" command is similar in concept to PRAGMA integrity_check. In a working system, the "integrity-command" should always be successful. Possible causes of integrity-check failures include:

- The application has made changes to the FTS shadow tables directly, without using the FTS3/4 virtual table, causing the shadow tables to become out of sync with each other.
- Using the FTS4 content option and failing to manually keep the content in sync with the FTS4 inverted indices.
- Bugs in the FTS3/4 virtual table. (The "integrity-check" command was original conceived as part of the test suite for FTS3/4.)
- Corruption to the underlying SQLite database file. (See documentation on how to corrupt and SQLite database for additional information.)

7.4. The "merge=X,Y" command

The "merge=X,Y" command (where X and Y are integers) causes SQLite to do a limited amount of work toward merging the various inverted index b-trees of an FTS3/4 table together into one large b-tree. The X value is the target number of "blocks" to be merged, and Y is the minimum number of b-tree segments on a level required before merging will be applied to that level. The value of Y should be between 2 and 16 with a recommended value of 8. The value of X can be any positive integer but values on the order of 100 to 300 are recommended.

When an FTS table accumulates 16 b-tree segments at the same level, the next INSERT into that table will cause all 16 segments to be merged into a single b-tree segment at the next higher level. The effect of these level merges is that most INSERTs into an FTS table are very fast and take minimal memory, but an occasional INSERT is slow and generates a large transaction because of the need to do merging. This results in "spiky" performance of INSERTs.

To avoid spiky INSERT performance, an application can run the "merge=X,Y" command periodically, possibly in an idle thread or idle process, to ensure that the FTS table never accumulates too many b-tree segments at the same level. INSERT performance spikes can generally be avoided, and performance of FTS3/4 can be maximized, by running "merge=X,Y" after every few thousand document inserts. Each "merge=X,Y" command will run in a separate transaction (unless they are grouped together using BEGIN...COMMIT, of course). The transactions can be kept small by choosing a value for X in the range of 100 to 300. The idle thread that is running the merge commands can know when it is done by checking the difference in sqlite3_total_changes() before and after each "merge=X,Y" command and stopping the loop when the difference drops below two.

7.5. The "automerge=N" command

The "automerge=N" command (where N is an integer between 0 and 15, inclusive) is used to configure an FTS3/4 tables "automerge" parameter, which controls automatic incremental inverted index merging. The default automerge value for new tables is 0, meaning that automatic incremental merging is completely disabled. If the value of the automerge parameter is modified using the "automerge=N" command, the new parameter value is stored persistently in the database and is used by all subsequently established database connections.

Setting the automerge parameter to a non-zero value enables automatic incremental merging. This causes SQLite to do a small amount of inverted index merging after every INSERT operation. The amount of merging performed is designed so that the FTS3/4 table never reaches a point where it has 16 segments at the same level and hence has to do a large merge in order to complete an insert. In other words, automatic incremental merging is designed to prevent spiky INSERT performance.

The downside of automatic incremental merging is that it makes every INSERT, UPDATE, and DELETE operation on an FTS3/4 table run a little slower, since extra time must be used to do the incremental merge. For maximum performance, it is recommended that applications disable automatic incremental merge and instead use the "merge" command in an idle process to keep the inverted indices well merged. But if the structure of an application does not easily allow for idle processes, the use of automatic incremental merge is a very reasonable fallback solution.

The actual value of the automerge parameter determines the number of index segments merged simultaneously by an automatic inverted index merge. If the value is set to N, the system waits until there are at least N segments on a single level before beginning to incrementally merge them. Setting a lower value of N causes segments to be merged more quickly, which may speed up full-text queries and, if the workload contains UPDATE or DELETE operations as well as INSERTs, reduce the space on disk consumed by the full-text index. However, it also increases the amount of data written to disk.

For general use in cases where the workload contains few UPDATE or DELETE operations, a good choice for automerge is 8. If the workload contains many UPDATE or DELETE commands, or if query speed is a concern, it may be advantageous to reduce automerge to 2.

For reasons of backwards compatibility, the "automerge=1" command sets the automerge parameter to 8, not 1 (a value of 1 would make no sense anyway, as merging data from a single segment is a no-op).

8. Tokenizers

An FTS tokenizer is a set of rules for extracting terms from a document or basic FTS full-text query.

Unless a specific tokenizer is specified as part of the CREATE VIRTUAL TABLE statement used to create the FTS table, the default tokenizer, "simple", is used. The simple tokenizer extracts tokens from a document or basic FTS full-text query according to the following rules:

- A term is a contiguous sequence of eligible characters, where eligible characters are all alphanumeric characters and all characters with Unicode codepoint values greater than or equal to 128. All other characters are discarded when splitting a document into terms. Their only contribution is to separate adjacent terms.
- All uppercase characters within the ASCII range (Unicode codepoints less than 128), are transformed to their lowercase equivalents as part of the tokenization process. Thus, full-text queries are case-insensitive when using the simple tokenizer.

For example, when a document containing the text "Right now, they're very frustrated.", the terms extracted from the document and added to the full-text index are, in order, "right now they re very frustrated". Such a document would match a full-text query such as "MATCH 'Frustrated'", as the simple tokenizer transforms the term in the query to lowercase before searching the full-text index.

As well as the "simple" tokenizer, the FTS source code features a tokenizer that uses the Porter Stemming algorithm. This tokenizer uses the same rules to separate the input document into terms including folding all terms

into lower case, but also uses the Porter Stemming algorithm to reduce related English language words to a common root. For example, using the same input document as in the paragraph above, the porter tokenizer extracts the following tokens: "right now thei veri frustrat". Even though some of these terms are not even English words, in some cases using them to build the full-text index is more useful than the more intelligible output produced by the simple tokenizer. Using the porter tokenizer, the document not only matches full-text queries such as "MATCH 'Frustrated'", but also queries such as "MATCH 'Frustration'", as the term "Frustration" is reduced by the Porter stemmer algorithm to "frustrat" - just as "Frustrated" is. So, when using the porter tokenizer, FTS is able to find not just exact matches for queried terms, but matches against similar English language terms. For more information on the Porter Stemmer algorithm, please refer to the page linked above.

Example illustrating the difference between the "simple" and "porter" tokenizers:

```
-- Create a table using the simple tokenizer. Insert a document into it.
CREATE VIRTUAL TABLE simple USING fts3(tokenize=simple);
INSERT INTO simple VALUES('Right now they''re very frustrated');

-- The first of the following two queries matches the document stored in
-- table "simple". The second does not.
SELECT * FROM simple WHERE simple MATCH 'Frustrated';
SELECT * FROM simple WHERE simple MATCH 'Frustration';

-- Create a table using the porter tokenizer. Insert the same document into it
CREATE VIRTUAL TABLE porter USING fts3(tokenize=porter);
INSERT INTO porter VALUES('Right now they''re very frustrated');

-- Both of the following queries match the document stored in table "porter".
SELECT * FROM porter WHERE porter MATCH 'Frustrated';
SELECT * FROM porter WHERE porter MATCH 'Frustration';
```

If this extension is compiled with the SQLITE_ENABLE_ICU pre-processor symbol defined, then there exists a built-in tokenizer named "icu" implemented using the ICU library. The first argument passed to the xCreate() method (see fts3_tokenizer.h) of this tokenizer may be an ICU locale identifier. For example "tr_TR" for Turkish as used in Turkey, or "en_AU" for English as used in Australia. For example:

```
CREATE VIRTUAL TABLE thai_text USING fts3(text, tokenize=icu th_TH)
```

The ICU tokenizer implementation is very simple. It splits the input text according to the ICU rules for finding word boundaries and discards any tokens that consist entirely of white-space. This may be suitable for some applications in some locales, but not all. If more complex processing is required, for example to implement stemming or discard punctuation, this can be done by creating a tokenizer implementation that uses the ICU tokenizer as part of its implementation.

The "unicode61" tokenizer is available beginning with SQLite version 3.7.13 (2012-06-11). Unicode61 works very much like "simple" except that it does simple unicode case folding according to rules in Unicode Version 6.1 and it recognizes unicode space and punctuation characters and uses those to separate tokens. The simple tokenizer only does case folding of ASCII characters and only recognizes ASCII space and punctuation characters as token separators.

By default, "unicode61" attempts to remove diacritics from Latin script characters. This behaviour can be overridden by adding the tokenizer argument "remove_diacritics=0". For example:

```
-- Create tables that remove alldiacritics from Latin script characters
-- as part of tokenization.
CREATE VIRTUAL TABLE txt1 USING fts4(tokenize=unicode61);
CREATE VIRTUAL TABLE txt2 USING fts4(tokenize=unicode61 "remove_diacritics=2");

-- Create a table that does not remove diacritics from Latin script
-- characters as part of tokenization.
CREATE VIRTUAL TABLE txt3 USING fts4(tokenize=unicode61 "remove_diacritics=0");
```

The remove_diacritics option may 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.

It is also possible to customize the set of codepoints that unicode61 treats as separator characters. The "separators=" option may be used to specify one or more extra characters that should be treated as separator characters, and the "tokenchars=" option may be used to specify one or more extra characters that should be treated as part of tokens instead of as separator characters. For example:

```
-- Create a table that uses the unicode61 tokenizer, but considers "."
-- and "=" characters to be part of tokens, and capital "X" characters to
-- function as separators.
CREATE VIRTUAL TABLE txt3 USING fts4(tokenize=unicode61 "tokenchars=.=" "separators=X");

-- Create a table that considers space characters (codepoint 32) to be
-- a token character
CREATE VIRTUAL TABLE txt4 USING fts4(tokenize=unicode61 "tokenchars= ");
```

If a character specified as part of the argument to "tokenchars=" is considered to be a token character by default, it is ignored. This is true even if it has been marked as a separator by an earlier "separators=" option. Similarly, if a character specified as part of a "separators=" option is treated as a separator character by default, it is ignored. If multiple "tokenchars=" or "separators=" options are specified, all are processed. For example:

```
-- Create a table that uses the unicode61 tokenizer, but considers "."
-- and "=" characters to be part of tokens, and capital "X" characters to
-- function as separators. Both of the "tokenchars=" options are processed
-- The "separators=" option ignores the "." passed to it, as "." is by
-- default a separator character, even though it has been marked as a token
-- character by an earlier "tokenchars=" option.
CREATE VIRTUAL TABLE txt5 USING fts4(
    tokenize=unicode61 "tokenchars=." "separators=X." "tokenchars=="
);
```

The arguments passed to the "tokenchars=" or "separators=" options are case-sensitive. In the example above, specifying that "X" is a separator character does not affect the way "x" is handled.

8.1. Custom (Application Defined) Tokenizers

In addition to providing built-in "simple", "porter" and (possibly) "icu" and "unicode61" tokenizers, FTS provides an interface for applications to implement and register custom tokenizers written in C. The interface used to create a new tokenizer is defined and described in the fts3_tokenizer.h source file.

Registering a new FTS tokenizer is similar to registering a new virtual table module with SQLite. The user passes a pointer to a structure containing pointers to various callback functions that make up the implementation of the new tokenizer type. For tokenizers, the structure (defined in fts3_tokenizer.h) is called "sqlite3_tokenizer_module".

FTS does not expose a C-function that users call to register new tokenizer types with a database handle. Instead, the pointer must be encoded as an SQL blob value and passed to FTS through the SQL engine by evaluating a special scalar function, "fts3_tokenizer()". The fts3_tokenizer() function may be called with one or two arguments, as follows:

```
SELECT fts3_tokenizer(<tokenizer-name>);
SELECT fts3_tokenizer(<tokenizer-name>, <sqlite3_tokenizer_module ptr>);
```

Where <tokenizer-name> is parameter to which a string is bound using sqlite3_bind_text() where the string identifies the tokenizer and <sqlite3_tokenizer_module ptr> is a parameter to which a BLOB is bound using sqlite3_bind_blob() where the value of the BLOB is a pointer to an sqlite3_tokenizer_module structure. If the second argument is present, it is registered as tokenizer <tokenizer-name> and a copy of it returned. If only one

argument is passed, a pointer to the tokenizer implementation currently registered as <tokenizer-name> is returned, encoded as a blob. Or, if no such tokenizer exists, an SQL exception (error) is raised.

Prior to SQLite version 3.11.0 (2016-02-15), the arguments to fts3_tokenzer() could be literal strings or BLOBs. They did not have to be bound parameters. But that could lead to security problems in the event of an SQL injection. Hence, the legacy behavior is now disabled by default. But the old legacy behavior can be enabled, for backwards compatibility in applications that really need it, by calling sqlite3_db_config(db,SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER,1,0).

The following block contains an example of calling the fts3_tokenizer() function from C code:

```
/*
** Register a tokenizer implementation with FTS3 or FTS4.
*/
int registerTokenizer(
  sqlite3 *db,
  char *zName,
  const sqlite3_tokenizer_module *p
){
  int rc;
  sqlite3_stmt *pStmt;
  const char *zSql = "SELECT fts3_tokenizer(?1, ?2)";

  rc = sqlite3_prepare_v2(db, zSql, -1, &pStmt, 0);
  if( rc!=SQLITE_OK ){
    return rc;
  }

  sqlite3_bind_text(pStmt, 1, zName, -1, SQLITE_STATIC);
  sqlite3_bind_blob(pStmt, 2, &p, sizeof(p), SQLITE_STATIC);
  sqlite3_step(pStmt);

  return sqlite3_finalize(pStmt);
}

/*
** Query FTS for the tokenizer implementation named zName.
*/
int queryTokenizer(
  sqlite3 *db,
  char *zName,
  const sqlite3_tokenizer_module **pp
){
  int rc;
  sqlite3_stmt *pStmt;
  const char *zSql = "SELECT fts3_tokenizer(?)";

  *pp = 0;
  rc = sqlite3_prepare_v2(db, zSql, -1, &pStmt, 0);
  if( rc!=SQLITE_OK ){
    return rc;
  }

  sqlite3_bind_text(pStmt, 1, zName, -1, SQLITE_STATIC);
  if( SQLITE_ROW==sqlite3_step(pStmt) ){
    if( sqlite3_column_type(pStmt, 0)==SQLITE_BLOB ){
      memcpy(pp, sqlite3_column_blob(pStmt, 0), sizeof(*pp));
    }
  }

  return sqlite3_finalize(pStmt);
}
```

8.2. Querying Tokenizers

The "fts3tokenize" virtual table can be used to directly access any tokenizer. The following SQL demonstrates how to create an instance of the fts3tokenize virtual table:

```
CREATE VIRTUAL TABLE tok1 USING fts3tokenize('porter');
```

The name of the desired tokenizer should be substituted in place of 'porter' in the example, of course. If the tokenizer requires one or more arguments, they should be separated by commas in the fts3tokenize declaration (even though they are separated by spaces in declarations of regular fts4 tables). The following creates fts4 and fts3tokenize tables that use the same tokenizer:

```
CREATE VIRTUAL TABLE text1 USING fts4(tokenize=icu en_AU);
CREATE VIRTUAL TABLE tokens1 USING fts3tokenize(icu, en_AU);

CREATE VIRTUAL TABLE text2 USING fts4(tokenize=unicode61 "tokenchars=@." "separators=123");
CREATE VIRTUAL TABLE tokens2 USING fts3tokenize(unicode61, "tokenchars=@.", "separators=123");
```

Once the virtual table is created, it can be queried as follows:

```
SELECT token, start, end, position
  FROM tok1
 WHERE input='This is a test sentence.';
```

The virtual table will return one row of output for each token in the input string. The "token" column is the text of the token. The "start" and "end" columns are the byte offset to the beginning and end of the token in the original input string. The "position" column is the sequence number of the token in the original input string. There is also an "input" column which is simply a copy of the input string that is specified in the WHERE clause. Note that a constraint of the form "input=?" must appear in the WHERE clause or else the virtual table will have no input to tokenize and will return no rows. The example above generates the following output:

```
thi|0|4|0
is|5|7|1
a|8|9|2
test|10|14|3
sentenc|15|23|4
```

Notice that the tokens in the result set from the fts3tokenize virtual table have been transformed according to the rules of the tokenizer. Since this example used the "porter" tokenizer, the "This" token was converted into "thi". If the original text of the token is desired, it can be retrieved using the "start" and "end" columns with the substr() function. For example:

```
SELECT substr(input, start+1, end-start), token, position
  FROM tok1
 WHERE input='This is a test sentence.';
```

The fts3tokenize virtual table can be used on any tokenizer, regardless of whether or not there exists an FTS3 or FTS4 table that actually uses that tokenizer.

9. 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.

9.1. Shadow Tables

For each FTS virtual table in a database, three to five real (non-virtual) tables are created to store the underlying data. These real tables are called "shadow tables". The real tables are named "%_content", "%_segdir", "%_segments", "%_stat", and "%_docsize", where "%" is replaced by the name of the FTS virtual table.

The leftmost column of the "%_content" table is an INTEGER PRIMARY KEY field named "docid". Following this is one column for each column of the FTS virtual table as declared by the user, named by prepending the column name supplied by the user with "c*N*", where *N* is the index of the column within the table, numbered from left to right starting with 0. Data types supplied as part of the virtual table declaration are not used as part of the %_content table declaration. For example:

```
-- Virtual table declaration
CREATE VIRTUAL TABLE abc USING fts4(a NUMBER, b TEXT, c);

-- Corresponding %_content table declaration
CREATE TABLE abc_content(docid INTEGER PRIMARY KEY, c0a, c1b, c2c);
```

The %_content table contains the unadulterated data inserted by the user into the FTS virtual table by the user. If the user does not explicitly supply a "docid" value when inserting records, one is selected automatically by the system.

The %_stat and %_docsize tables are only created if the FTS table uses the FTS4 module, not FTS3. Furthermore, the %_docsize table is omitted if the FTS4 table is created with the "matchinfo=fts3" directive specified as part of the CREATE VIRTUAL TABLE statement. If they are created, the schema of the two tables is as follows:

```
CREATE TABLE %_stat(
  id INTEGER PRIMARY KEY,
  value BLOB
);

CREATE TABLE %_docsize(
  docid INTEGER PRIMARY KEY,
  size BLOB
);
```

For each row in the FTS table, the %_docsize table contains a corresponding row with the same "docid" value. The "size" field contains a blob consisting of *N* FTS varints, where *N* is the number of user-defined columns in the table. Each varint in the "size" blob is the number of tokens in the corresponding column of the associated row in the FTS table. The %_stat table always contains a single row with the "id" column set to 0. The "value" column contains a blob consisting of *N+1* FTS varints, where *N* is again the number of user-defined columns in the FTS table. The first varint in the blob is set to the total number of rows in the FTS table. The second and subsequent varints contain the total number of tokens stored in the corresponding column for all rows of the FTS table.

The two remaining tables, %_segments and %_segdir, are used to store the full-text index. Conceptually, this index is a lookup table that maps each term (word) to the set of docid values corresponding to records in the %_content table that contain one or more occurrences of the term. To retrieve all documents that contain a specified term, the FTS module queries this index to determine the set of docid values for records that contain the term, then retrieves the required documents from the %_content table. Regardless of the schema of the FTS virtual table, the %_segments and %_segdir tables are always created as follows:

```
CREATE TABLE %_segments(
  blockid INTEGER PRIMARY KEY,       -- B-tree node id
  block blob                         -- B-tree node data
);

CREATE TABLE %_segdir(
  level INTEGER,
  idx INTEGER,
  start_block INTEGER,               -- Blockid of first node in %_segments
  leaves_end_block INTEGER,          -- Blockid of last leaf node in %_segments
  end_block INTEGER,                 -- Blockid of last node in %_segments
  root BLOB,                         -- B-tree root node
  PRIMARY KEY(level, idx)
);
```

The schema depicted above is not designed to store the full-text index directly. Instead, it is used to store one or more b-tree structures. There is one b-tree for each row in the %_segdir table. The %_segdir table row contains

the root node and various meta-data associated with the b-tree structure, and the %_segments table contains all other (non-root) b-tree nodes. Each b-tree is referred to as a "segment". Once it has been created, a segment b-tree is never updated (although it may be deleted altogether).

The keys used by each segment b-tree are terms (words). As well as the key, each segment b-tree entry has an associated "doclist" (document list). A doclist consists of zero or more entries, where each entry consists of:

- A docid (document id), and
- A list of term offsets, one for each occurrence of the term within the document. A term offset indicates the number of tokens (words) that occur before the term in question, not the number of characters or bytes. For example, the term offset of the term "war" in the phrase "Ancestral voices prophesying war!" is 3.

Entries within a doclist are sorted by docid. Positions within a doclist entry are stored in ascending order.

The contents of the logical full-text index is found by merging the contents of all segment b-trees. If a term is present in more than one segment b-tree, then it maps to the union of each individual doclist. If, for a single term, the same docid occurs in more than one doclist, then only the doclist that is part of the most recently created segment b-tree is considered valid.

Multiple b-tree structures are used instead of a single b-tree to reduce the cost of inserting records into FTS tables. When a new record is inserted into an FTS table that already contains a lot of data, it is likely that many of the terms in the new record are already present in a large number of existing records. If a single b-tree were used, then large doclist structures would have to be loaded from the database, amended to include the new docid and term-offset list, then written back to the database. Using multiple b-tree tables allows this to be avoided by creating a new b-tree which can be merged with the existing b-tree (or b-trees) later on. Merging of b-tree structures can be performed as a background task, or once a certain number of separate b-tree structures have been accumulated. Of course, this scheme makes queries more expensive (as the FTS code may have to look up individual terms in more than one b-tree and merge the results), but it has been found that in practice this overhead is often negligible.

9.2. Variable Length Integer (varint) Format

Integer values stored as part of segment b-tree nodes are encoded using the FTS varint format. This encoding is similar, but **not identical**, to the SQLite varint format.

An encoded FTS varint consumes between one and ten bytes of space. The number of bytes required is determined by the sign and magnitude of the integer value encoded. More accurately, the number of bytes used to store the encoded integer depends on the position of the most significant set bit in the 64-bit twos-complement representation of the integer value. Negative values always have the most significant bit set (the sign bit), and so are always stored using the full ten bytes. Positive integer values may be stored using less space.

The final byte of an encoded FTS varint has its most significant bit cleared. All preceding bytes have the most significant bit set. Data is stored in the remaining seven least significant bits of each byte. The first byte of the encoded representation contains the least significant seven bits of the encoded integer value. The second byte of the encoded representation, if it is present, contains the seven next least significant bits of the integer value, and so on. The following table contains examples of encoded integer values:

Decimal	Hexadecimal	Encoded Representation
43	0x000000000000002B	0x2B
200815	0x000000000003106F	0x9C 0xA0 0x0C
-1	0xFFFFFFFFFFFFFFFF	0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0x01

9.3. Segment B-Tree Format

Segment b-trees are prefix-compressed b+-trees. There is one segment b-tree for each row in the %_segdir table (see above). The root node of the segment b-tree is stored as a blob in the "root" field of the corresponding row of the %_segdir table. All other nodes (if any exist) are stored in the "blob" column of the %_segments table. Nodes

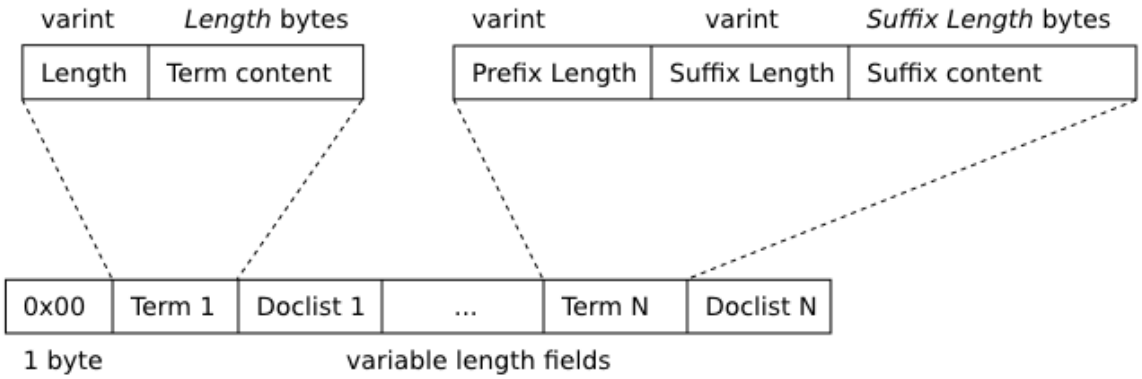
within the %_segments table are identified by the integer value in the blockid field of the corresponding row. The following table describes the fields of the %_segdir table:

Column	Interpretation
level	Between them, the contents of the "level" and "idx" fields define the relative age of the segment b-tree. The smaller the value stored in the "level" field, the more recently the segment b-tree was created. If two segment b-trees are of the same "level", the segment with the larger value stored in the "idx" column is more recent. The PRIMARY KEY constraint on the %_segdir table prevents any two segments from having the same value for both the "level" and "idx" fields.
idx	See above.
start_block	The blockid that corresponds to the node with the smallest blockid that belongs to this segment b-tree. Or zero if the entire segment b-tree fits on the root node. If it exists, this node is always a leaf node.
leaves_end_block	The blockid that corresponds to the leaf node with the largest blockid that belongs to this segment b-tree. Or zero if the entire segment b-tree fits on the root node.
end_block	This field may contain either an integer or a text field consisting of two integers separated by a space character (unicode codepoint 0x20). The first, or only, integer is the blockid that corresponds to the interior node with the largest blockid that belongs to this segment b-tree. Or zero if the entire segment b-tree fits on the root node. If it exists, this node is always an interior node. The second integer, if it is present, is the aggregate size of all data stored on leaf pages in bytes. If the value is negative, then the segment is the output of an unfinished incremental-merge operation, and the absolute value is current size in bytes.
root	Blob containing the root node of the segment b-tree.

Apart from the root node, the nodes that make up a single segment b-tree are always stored using a contiguous sequence of blockids. Furthermore, the nodes that make up a single level of the b-tree are themselves stored as a contiguous block, in b-tree order. The contiguous sequence of blockids used to store the b-tree leaves are allocated starting with the blockid value stored in the "start_block" column of the corresponding %_segdir row, and finishing at the blockid value stored in the "leaves_end_block" field of the same row. It is therefore possible to iterate through all the leaves of a segment b-tree, in key order, by traversing the %_segments table in blockid order from "start_block" to "leaves_end_block".

9.3.1. Segment B-Tree Leaf Nodes

The following diagram depicts the format of a segment b-tree leaf node.

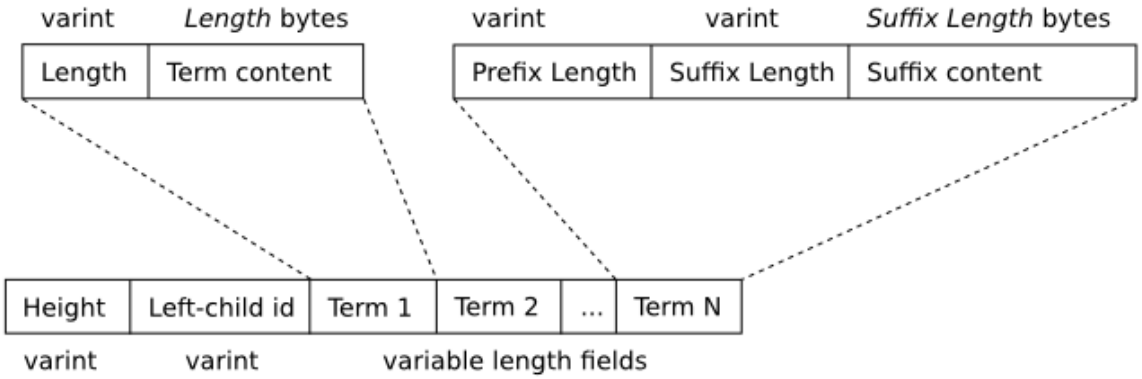


Segment B-Tree Leaf Node Format

The first term stored on each node ("Term 1" in the figure above) is stored verbatim. Each subsequent term is prefix-compressed with respect to its predecessor. Terms are stored within a page in sorted (memcmp) order.

9.3.2. Segment B-Tree Interior Nodes

The following diagram depicts the format of a segment b-tree interior (non-leaf) node.

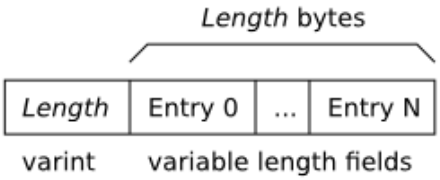


Segment B-Tree Interior Node Format

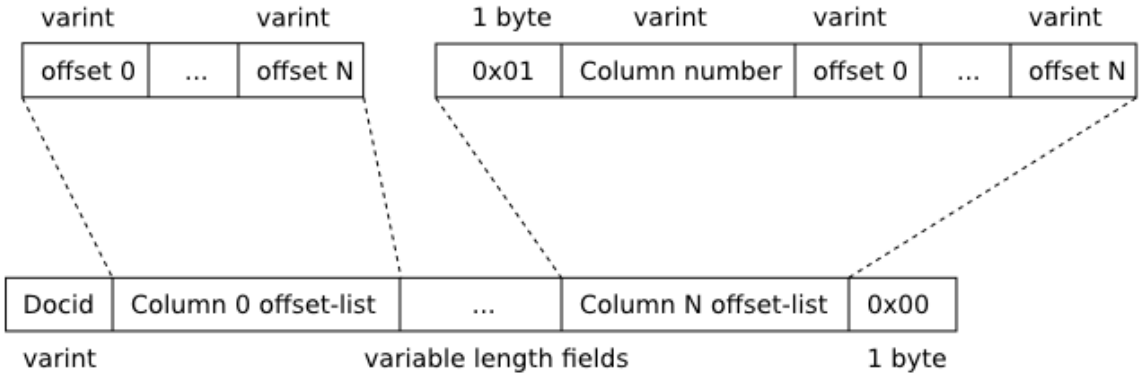
9.4. Doclist Format

A doclist consists of an array of 64-bit signed integers, serialized using the FTS varint format. Each doclist entry is made up of a series of two or more integers, as follows:

1. The docid value. The first entry in a doclist contains the literal docid value. The first field of each subsequent doclist entry contains the difference between the new docid and the previous one (always a positive number).
2. Zero or more term-offset lists. A term-offset list is present for each column of the FTS virtual table that contains the term. A term-offset list consists of the following:
    1. Constant value 1. This field is omitted for any term-offset list associated with column 0.
    2. The column number (1 for the second leftmost column, etc.). This field is omitted for any term-offset list associated with column 0.
    3. A list of term-offsets, sorted from smallest to largest. Instead of storing the term-offset value literally, each integer stored is the difference between the current term-offset and the previous one (or zero if the current term-offset is the first), plus 2.
3. Constant value 0.



FTS3 Doclist Format



FTS Doclist Entry Format

For doclists for which the term appears in more than one column of the FTS virtual table, term-offset lists within the doclist are stored in column number order. This ensures that the term-offset list associated with column 0 (if any) is always first, allowing the first two fields of the term-offset list to be omitted in this case.

10. Limitations

10.1. UTF-16 byte-order-mark problem

For UTF-16 databases, when using the "simple" tokenizer, it is possible to use malformed unicode strings to cause the integrity-check special command to falsely report corruption, or for auxiliary functions to return incorrect results. More specifically, the bug can be triggered by any of the following:

- A UTF-16 byte-order-mark (BOM) is embedded at the beginning of an SQL string literal value inserted into an FTS3 table. For example:

  ```
  INSERT INTO fts_table(col) VALUES(char(0xfeff)||'text...');
  ```

- Malformed UTF-8 that SQLite converts to a UTF-16 byte-order-mark is embedded at the beginning of an SQL string literal value inserted into an FTS3 table.

- A text value created by casting a blob that begins with the two bytes 0xFF and 0xFE, in either possible order, is inserted into an FTS3 table. For example:

  ```
  INSERT INTO fts_table(col) VALUES(CAST(X'FEFF' AS TEXT));
  ```

Everything works correctly if any of the following are true:

- The database encoding is UTF-8.
- All text strings are insert using one of the sqlite3_bind_text() family of functions.
- Literal strings contain no byte-order-marks.

- A tokenizer is used that recognizes byte-order-marks as whitespace. (The default "simple" tokenizer for FTS3/4 does not think that BOMs are whitespace, but the unicode tokenizer does.)

All of the above conditions must be false in order for problems to occur. And even if all of the conditiona above are false, most things will still operator correctly. Only the integrity-check command and the auxiliary functions might given unexpected results.

Appendix A: Search Application Tips

FTS is primarily designed to support Boolean full-text queries - queries to find the set of documents that match a specified criteria. However, many (most?) search applications require that results are somehow ranked in order of "relevance", where "relevance" is defined as the likelihood that the user who performed the search is interested in a specific element of the returned set of documents. When using a search engine to find documents on the world wide web, the user expects that the most useful, or "relevant", documents will be returned as the first page of results, and that each subsequent page contains progressively less relevant results. Exactly how a machine can determine document relevance based on a users query is a complicated problem and the subject of much ongoing research.

One very simple scheme might be to count the number of instances of the users search terms in each result document. Those documents that contain many instances of the terms are considered more relevant than those with a small number of instances of each term. In an FTS application, the number of term instances in each result could be determined by counting the number of integers in the return value of the offsets function. The following example shows a query that could be used to obtain the ten most relevant results for a query entered by the user:

```
-- This example (and all others in this section) assumes the following schema
CREATE VIRTUAL TABLE documents USING fts3(title, content);

-- Assuming the application has supplied an SQLite user function named "countintegers"
-- that returns the number of space-separated integers contained in its only argument,
-- the following query could be used to return the titles of the 10 documents that contain
-- the greatest number of instances of the users query terms. Hopefully, these 10
-- documents will be those that the users considers more or less the most "relevant".
SELECT title FROM documents
  WHERE documents MATCH <query>
  ORDER BY countintegers(offsets(documents)) DESC
  LIMIT 10 OFFSET 0
```

The query above could be made to run faster by using the FTS matchinfo function to determine the number of query term instances that appear in each result. The matchinfo function is much more efficient than the offsets function. Furthermore, the matchinfo function provides extra information regarding the overall number of occurrences of each query term in the entire document set (not just the current row) and the number of documents in which each query term appears. This may be used (for example) to attach a higher weight to less common terms which may increase the overall computed relevancy of those results the user considers more interesting.

```
-- If the application supplies an SQLite user function called "rank" that
-- interprets the blob of data returned by matchinfo and returns a numeric
-- relevancy based on it, then the following SQL may be used to return the
-- titles of the 10 most relevant documents in the dataset for a users query.
SELECT title FROM documents
  WHERE documents MATCH <query>
  ORDER BY rank(matchinfo(documents)) DESC
  LIMIT 10 OFFSET 0
```

The SQL query in the example above uses less CPU than the first example in this section, but still has a non-obvious performance problem. SQLite satisfies this query by retrieving the value of the "title" column and matchinfo data from the FTS module for every row matched by the users query before it sorts and limits the results. Because of the way SQLite's virtual table interface works, retrieving the value of the "title" column requires loading the entire row from disk (including the "content" field, which may be quite large). This means that if the users query matches several thousand documents, many megabytes of "title" and "content" data may be loaded from disk into memory even though they will never be used for any purpose.

The SQL query in the following example block is one solution to this problem. In SQLite, when a sub-query used in a join contains a LIMIT clause, the results of the sub-query are calculated and stored in temporary table before the main query is executed. This means that SQLite will load only the docid and matchinfo data for each row matching the users query into memory, determine the docid values corresponding to the ten most relevant documents, then load only the title and content information for those 10 documents only. Because both the matchinfo and docid values are gleaned entirely from the full-text index, this results in dramatically less data being loaded from the database into memory.

```
SELECT title FROM documents JOIN (
    SELECT docid, rank(matchinfo(documents)) AS rank
    FROM documents
    WHERE documents MATCH <query>
    ORDER BY rank DESC
    LIMIT 10 OFFSET 0
) AS ranktable USING(docid)
ORDER BY ranktable.rank DESC
```

The next block of SQL enhances the query with solutions to two other problems that may arise in developing search applications using FTS:

1. The snippet function cannot be used with the above query. Because the outer query does not include a "WHERE ... MATCH" clause, the snippet function may not be used with it. One solution is to duplicate the WHERE clause used by the sub-query in the outer query. The overhead associated with this is usually negligible.

2. The relevancy of a document may depend on something other than just the data available in the return value of matchinfo. For example each document in the database may be assigned a static weight based on factors unrelated to its content (origin, author, age, number of references etc.). These values can be stored by the application in a separate table that can be joined against the documents table in the sub-query so that the rank function may access them.

This version of the query is very similar to that used by the sqlite.org documentation search application.

```
-- This table stores the static weight assigned to each document in FTS table
-- "documents". For each row in the documents table there is a corresponding row
-- with the same docid value in this table.
CREATE TABLE documents_data(docid INTEGER PRIMARY KEY, weight);

-- This query is similar to the one in the block above, except that:
--
--   1. It returns a "snippet" of text along with the document title for display. So
--      that the snippet function may be used, the "WHERE ... MATCH ..." clause from
--      the sub-query is duplicated in the outer query.
--
--   2. The sub-query joins the documents table with the document_data table, so that
--      implementation of the rank function has access to the static weight assigned
--      to each document.
SELECT title, snippet(documents) FROM documents JOIN (
    SELECT docid, rank(matchinfo(documents), documents_data.weight) AS rank
    FROM documents JOIN documents_data USING(docid)
    WHERE documents MATCH <query>
    ORDER BY rank DESC
    LIMIT 10 OFFSET 0
) AS ranktable USING(docid)
WHERE documents MATCH <query>
ORDER BY ranktable.rank DESC
```

All the example queries above return the ten most relevant query results. By modifying the values used with the OFFSET and LIMIT clauses, a query to return (say) the next ten most relevant results is easy to construct. This may be used to obtain the data required for a search applications second and subsequent pages of results.

The next block contains an example rank function that uses matchinfo data implemented in C. Instead of a single weight, it allows a weight to be externally assigned to each column of each document. It may be registered with SQLite like any other user function using sqlite3_create_function.

Security Warning: Because it is just an ordinary SQL function, rank() may be invoked as part of any SQL query in any context. This means that the first argument passed may not be a valid matchinfo blob. Implementors should take care to handle this case without causing buffer overruns or other potential security problems.

```
/*
** SQLite user defined function to use with matchinfo() to calculate the
** relevancy of an FTS match. The value returned is the relevancy score
** (a real value greater than or equal to zero). A larger value indicates
** a more relevant document.
**
** The overall relevancy returned is the sum of the relevancies of each
** column value in the FTS table. The relevancy of a column value is the
** sum of the following for each reportable phrase in the FTS query:
**
**   (<hit count> / <global hit count>) * <column weight>
**
** where <hit count> is the number of instances of the phrase in the
** column value of the current row and <global hit count> is the number
** of instances of the phrase in the same column of all rows in the FTS
** table. The <column weight> is a weighting factor assigned to each
** column by the caller (see below).
**
** The first argument to this function must be the return value of the FTS
** matchinfo() function. Following this must be one argument for each column
** of the FTS table containing a numeric weight factor for the corresponding
** column. Example:
**
**     CREATE VIRTUAL TABLE documents USING fts3(title, content)
**
** The following query returns the docids of documents that match the full-text
** query <query> sorted from most to least relevant. When calculating
** relevance, query term instances in the 'title' column are given twice the
** weighting of those in the 'content' column.
**
**     SELECT docid FROM documents
**     WHERE documents MATCH <query>
**     ORDER BY rank(matchinfo(documents), 1.0, 0.5) DESC
*/
static void rankfunc(sqlite3_context *pCtx, int nVal, sqlite3_value **apVal){
  int *aMatchinfo;                /* Return value of matchinfo() */
  int nMatchinfo;                 /* Number of elements in aMatchinfo[] */
  int nCol = 0;                   /* Number of columns in the table */
  int nPhrase = 0;                /* Number of phrases in the query */
  int iPhrase;                    /* Current phrase */
  double score = 0.0;             /* Value to return */

  assert( sizeof(int)==4 );

  /* Check that the number of arguments passed to this function is correct.
  ** If not, jump to wrong_number_args. Set aMatchinfo to point to the array
  ** of unsigned integer values returned by FTS function matchinfo. Set
  ** nPhrase to contain the number of reportable phrases in the users full-text
  ** query, and nCol to the number of columns in the table. Then check that the
  ** size of the matchinfo blob is as expected. Return an error if it is not.
  */
  if( nVal<1 ) goto wrong_number_args;
  aMatchinfo = (unsigned int *)sqlite3_value_blob(apVal[0]);
  nMatchinfo = sqlite3_value_bytes(apVal[0]) / sizeof(int);
  if( nMatchinfo>=2 ){
    nPhrase = aMatchinfo[0];
    nCol = aMatchinfo[1];
  }
  if( nMatchinfo!=(2+3*nCol*nPhrase) ){
    sqlite3_result_error(pCtx,
      "invalid matchinfo blob passed to function rank()", -1);
    return;
  }
  if( nVal!=(1+nCol) ) goto wrong_number_args;

  /* Iterate through each phrase in the users query. */
  for(iPhrase=0; iPhrase<nPhrase; iPhrase++){
```

```
    int iCol;                       /* Current column */

    /* Now iterate through each column in the users query. For each column,
    ** increment the relevancy score by:
    **
    **   (<hit count> / <global hit count>) * <column weight>
    **
    ** aPhraseinfo[] points to the start of the data for phrase iPhrase. So
    ** the hit count and global hit counts for each column are found in
    ** aPhraseinfo[iCol*3] and aPhraseinfo[iCol*3+1], respectively.
    */
    int *aPhraseinfo = &aMatchinfo[2 + iPhrase*nCol*3];
    for(iCol=0; iCol<nCol; iCol++){
      int nHitCount = aPhraseinfo[3*iCol];
      int nGlobalHitCount = aPhraseinfo[3*iCol+1];
      double weight = sqlite3_value_double(apVal[iCol+1]);
      if( nHitCount>0 ){
        score += ((double)nHitCount / (double)nGlobalHitCount) * weight;
      }
    }
  }

  sqlite3_result_double(pCtx, score);
  return;

  /* Jump here if the wrong number of arguments are passed to this function */
wrong_number_args:
  sqlite3_result_error(pCtx, "wrong number of arguments to function rank()", -1);
}
```