**8. Tokenizers**

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

FTS tokenizer是一个规则集合：这些规则用于从一篇文档中提取词语或者是基本全文检索查询。

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:

在使用CREATE VIRTUAL TABLE命令创建FTS表时，默认的tokenizer是”simple”。 simple tokenizer的规则如下：

* 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](http://tartarus.org/%7Emartin/PorterStemmer/). 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](http://www.sqlite.org/releaselog/3_7_13.html). Unicode61 works very much like "simple" except that it does full 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" also removes all diacritics from Latin script characters. This behaviour can be overridden by adding the tokenizer argument "remove\_diacritics=0". For example:

|  |
| --- |
| *-- Create tables that remove diacritics 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=1");  *-- 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"); |

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 tables 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.