

組み込み型全文検索エンジン Senna
(初出:オープンソースマガジン 2006年4月号)

(有) 未来検索ブラジル

2006/06/21



Senna は LGPL で配布される転置インデックスタイプの高速・高精度な全文検索ソフトウェアです。この記事では Senna の設計方針や技術的な特徴を解説します。また、具体的な使用方法として、MySQL と Perl のそれぞれに組み込んだ場合について説明します。

1 開発の狙いと設計方針

今日、Yahoo!、Google、楽天、mixi、GREE、はてななどの多くのネットワークサービスが、LAMP (Linux、Apache、MySQL、PHP/Perl) に代表される優れたオープンソースソフトウェアの組み合わせによって構築されています。アイデアと行動力があれば、誰でも (少なくともソフトウェア面については) 非常に安いコストで、こうしたサービスの提供者に参入できるのです。

しかし、提供するサービスの種類によっては、既存のオープンソースソフトではまかないきれないソフトウェア部品が存在するのも事実です。たとえば Yahoo! や Google の提供する Web 検索エンジンは、LAMP だけでは構築できません。

これらの構築に不可欠な部品の 1 つが、高精度な全文検索機能です。Senna はその補完を目的として開発されました。LAMP で構築したシステムに強力な全文検索機能を付加し、本格的な検索サービスを容易に構築できる環境を整えることが Senna 開発の狙いです。これを実現するために、2 つの方針に従って設計しています。

1.1 検索精度重視

Senna は、検索精度 (適合率と再現率) を最も重視しており、第一に適合率を、次いで可能な限り再現率を、そして第三には精度を疎外しない範囲で処理速度を追究しています。このようにしているのは、処理速度はサーバーを増やすことによってスケールさせられますが、精度はエンジンの基本設計によってほぼ決定してしまうからです。

1.2 組み込み型に特化

Senna はまた、DBMS、言語処理系、ファイルシステムなど、何らかのストレージ系を持つシステムに組み込んで使用することに特化して設計されています。組み込み先のシステムの性能を阻害せず、最小限のオーバーヘッドで動作することに最大限配慮しています。

2 Senna の技術的特徴

上記の設計方針に基づいて実装された Senna ですが、より具体的な技術的特徴を説明しましょう。



2.1 高精度な検索

適合率に優れた単語インデックスか、再現率に優れた N-gram インデックスのいずれかを、インデックス作成時に選択できますが、お勧めは単語インデックスのほうです。通常、単語インデックスタイプのエンジンには、分かち書きの結果によって検索漏れが発生するという欠点があるのですが、この欠点を解消する工夫がなされているからです。

たとえば「自由民主党」や「社会民主党」が形態素解析辞書に登録されていた場合、これらは分かち書きされることなく、一単語として扱われます。その結果、「民主党」で検索したときにこれらはヒットしませんが、用途によってはこれらの文字列もヒットさせたい場合があります。

こういった場合にそなえて、Senna ではクエリ文字列が単語文字列に部分一致する場合にヒットさせるかどうかを、検索時に指定するオプションで制御できます。このオプションの組み合わせによって、

- 「民主党」に完全一致する文書がない場合のみ部分一致を行って、「自由民主党」や「社会民主党」を検索する
- 「民主党」に完全一致する文書には高いスコアを与え、「自由民主党」や「社会民主党」でヒットする文書には低いスコアを与える

などのようなきめ細かい操作が行えます。このほかにも次のような高度な検索機能を備えています。

- 近傍検索：指定された複数の単語が文書中の近傍に現れるものを検索する
- 段落検索：文書を複数の段落として処理し、複数の単語が同一の段落に現れる文書を検索する
- 類似検索：クエリに指定された文書と内容が類似する文書を検索する

2.2 高速な検索

完全転置インデックス方式を採用しており、高精度な検索結果を高速に返します。方式の特性上インデックスサイズは大きくなりますが、断片化を抑える機構が組み込まれているため、検索速度が犠牲になりません。



2.3 高速なインデックス更新

インデックス作成時に指定サイズの更新バッファを確保することで、高速な更新処理を実現しています。また、マルチスレッド/マルチプロセスで動かすときには、参照処理と更新処理を排他制御なしで同時実行可能です。そのため、組み込み先システムの処理性能を阻害しません。

2.4 自動再配置

更新/削除操作によって発生するゴミの回収と、検索時の読み出し単位に合わせたインデックスの再配置処理を自動的に実行します。これによって、追加/更新/削除操作を繰り返し実行しても検索性能が劣化しません。

2.5 ユーザーレベルキャッシュを使用可

高効率なユーザーレベルキャッシュを使用できます。Linux の AIO/DIO 機能¹を使用して、低メモリ消費で（カーネルのバッファキャッシュを消費せずに）、高い検索スループットを実現します（Linux 2.6 のみ対応）。

3 Senna の使い方

Senna は組み込み型に特化した検索エンジンなので、単体では使用できません。そのため、言語処理系や DBMS へのバインディングを通して使用することになります。本稿執筆時点の 2006 年 2 月 11 日現在では、次のバインディングが存在します。

- 言語処理系 : Perl、PHP、Ruby
- DBMS : PostgreSQL、MySQL

DBMS の場合は、テーブルやカラムに全文検索機能を付加する形になります。言語処理系の場合は、永続的なハッシュ（連想配列）などのストレージに全文検索機能を付加する形で使用するのが自然です。たとえば Perl であれば tie、Ruby であれば Mix-in によって、既存のデータ構造に全文検索機能を付加するような使い方が想定されます。

これ以降では、MySQL バインディングと Perl バインディングについて詳細に説明します。

¹Asynchronous IO と Direct IO。AIO は、アプリケーションと非同期に I/O 処理が行える機能。DIO は、ユーザーバッファとディスクの間で直接（ファイルキャッシュを利用せずに）データをやり取りできる機能。ともに Linux カーネル 2.6 で実装された。



4 MySQLで検索エンジンを作る！

オープンソースのDBMS、MySQLの利用が増えています。MySQLに格納された文字列を高速に検索したい、と思ったことはないでしょうか。

MySQLでは、格納されているデータの件数が少ないうちは、SQLでも実用的な速度で条件を絞り込めます。たとえばリスト1のクエリを実行すると、bodyフィールドに「単語」が含まれているレコードを取得できます。しかし、このようなクエリではインデックスを利用できないため、レコード数が多くなると非常に速度が遅くなってしまいます。

4.1 MySQLビルトインのFULLTEXTインデックスの問題点

MySQLにはもともとFULLTEXTインデックスと呼ばれるインデックスが実装されており、上記のようなクエリを高速に行えます（以降、「ビルトインのFULLTEXTインデックス」と呼びます）。たとえば、リスト2のクエリを実行すれば、前述のクエリと同様、bodyフィールドに「単語」が含まれているレコードを取得できます。

しかし、ビルトインのFULLTEXTインデックスは、実用的な全文検索アプリケーションを作るうえで次のような問題点があります。

・日本語の分かち書きの問題・フレーズ検索の速度の問題・インデックス更新の速度の問題・FULLTEXTインデックスとほかのインデックスを組み合わせられない問題これらの問題を解決するのが、SennaのMySQLバインディングです。

4.1.1 日本語の分かち書きの問題

ビルトインのFULLTEXTインデックスの設計は、日本語の文章を扱うことを想定していません。よって、日本語の文章に対してFULLTEXTインデックスを利用するためには、ひと工夫が必要です。たとえば、データベースにレコードを追加する際、単語と単語の間に区切りのスペースを入れると、日本語でも英語と同様にFULLTEXTインデックスを利用できます。これは、ChaSenやMeCabといった形態素解析ソフトウェアを使うことによって実現できます。

Sennaに付属のMySQL bindingというパッチを適用すると、ビルトインのFULLTEXTインデックスの代わりにSennaのインデックスを利用できるようになります。この場合は、単語をスペースで区切る必要がありません。



4.1.2 フレーズ検索の速度の問題

上記のパッチによる Senna のインデックスと、MySQL ビルトインの FULLTEXT インデックスの性能を比較した結果を表 1 に示します（検索対象として Wikipedia 英語版²を用い、「united states」を検索語としました。英語版を用いたのは、前述した日本語特有の問題の影響を除くためです）。

表 1: ビルトイン FULLTEXT インデックスと Senna インデックスとの比較（Wikipedia 45 万 8713 ページ 1088MB の文書データを対象にして計測）

A	BB	CCC
	ビルトイン	Senna
インデックスサイズ	109MB	1028MB
フレーズ検索にかかる時間	44.91 秒 ¹	0.40 秒
すでにデータが存在するテーブルにインデックスを付与するためにかかる時間	1,474 秒（24 分） ²	1,808 秒（30 分）
FULLTEXT インデックスを付与した空のテーブルにデータを追加するためにかかる時間	28,182 秒（469 分） ³	1,839 秒（30 分）
order by 他条件	20.33 秒 ⁴	0.89 秒
where 全文検索 and 他条件	6.55 秒 ⁵	0.32 秒

¹ where match (cur_text) against(' "united states "' in boolean mode)

² load してから add fulltext index

³ add fulltext index してから load

⁴ where match (cur_text) against(' united ') order by cur_id

⁵ where match (cur_text) against(' united ') and cur_id > 20000

まず、表 1 の 3 行目にある「フレーズ検索の速度」を見てください。「united states」という単語を検索した結果に、およそ 100 倍ほどの性能の差があるのが分かります。しかし、なぜこれほどの違いがあるのでしょうか。この違いを理解するために、ビルトインの FULLTEXT インデックスの仕組みを説明しましょう。

ビルトインの FULLTEXT インデックスは「ある単語がどのレコードに含まれているか」という情報のみを格納しています。「united states」を検索する場合、まず、「united」という単語を含んだレコードの候補と、「states」という単語を含んだレコードの候補を検索します。

² インターネット上の百科事典編集プロジェクト。
http://en.wikipedia.org/wiki/Main_Page



しかし、その両者に共通して現れるレコードをそのまま結果とすることはできません。なぜならば、「united」と「states」という単語を含んでいても、「united states」という単語を含まない文章があるからです。たとえば「the states of united kingdom」という文節は検索結果として不適切です。このような文章を検索結果から除外するためには、「united」と「states」の両方を含んだレコードのデータを読み込み、「united」と「states」が連続して出現していることを確認する必要があります。このレコードのデータを読み込むのに時間がかかるのです。

一方で Senna のインデックスは、「ある単語がどのレコードのどの場所にあるか」という情報を持っています。そのため、表 1 の 2 行目にあるとおりインデックスファイルのサイズは大きくなる半面、インデックスファイルだけで、「united」、「states」という 2 つの単語が連続して出現していることが分かるのです。Senna では、レコードのデータをまったく読むことなく高速な検索を行えるため、表 1 のような性能の差が現れます。

実はこのフレーズ検索の性能が、日本語の検索を行う場合には重要なのです。たとえば、「日本大学」という単語を検索する場合を考えてみましょう。

一見、「日本大学」は 1 つの単語のように見えます。しかし、一般的に ChaSen や MeCab を利用した場合、「日本大学」という単語は「日本/大学」というように、さらに細かい単語に分割されます。「united states」の検索と同様に、たとえば「日本の大学」という文字列が含まれているレコードが検索結果に出ないようにするため、「日本」、「大学」という単語をそれぞれ含んだレコードのデータをすべて読んで、検索を行わなければいけません。

このように日本語の検索では、一見フレーズ検索を利用しないようなクエリでも、内部的にはフレーズ検索を行っている場合が多く、フレーズ検索の性能が重要となります。

4.1.3 インデックス更新の速度の問題

さて、では次に表 1 の 4 行目と 5 行目について見てみましょう。4 行目ではすでにデータが存在するテーブルにインデックスを付与するためにかかる時間を、5 行目では FULLTEXT インデックスを付与した空のテーブルにデータを追加するためにかかる時間が書かれています。

ビルトインの FULLTEXT インデックスでは、表 1 のとおり前者と後者の速度差が顕著ですが、Senna を用いたインデックスでは、前者と後者の速度差が非常に小さくなっています。これは、Senna が既存のインデックスに対する更新操作のパフォーマンスを考慮して設計されているためです。つまり、インデックスに対する更新をバッファしたり、インデックスを自動再配置したりする機構を備えているため、高速な逐次更新を実現できます。



FULLTEXT インデックスを付与したテーブルに対して、レコードが頻繁に追加されたり更新されたりするケースも多いと思います。そのような場合には、すでにインデックスが存在する場合の更新速度が速いことは重要です。

4.1.4 FULLTEXT インデックスとほかのインデックスを組み合わせられない問題

最後に表1の6行目と7行目です。6行目（order by 他条件）は、「united」という単語の検索結果を、文書IDでソートした場合にかかる時間です。また、7行目（where 全文検索 and 他条件）は、「united」という単語の検索結果の中で、文書IDが20,000以上であるレコードの検索にかかる時間です。

表1のパフォーマンスを測定したテーブルには文書IDにPRIMARY インデックスが付与されていますが、通常FULLTEXT インデックスを利用する場合には、PRIMARY インデックスが利用されません。MySQLは、1つのクエリに対して原則1つのインデックスしか使用できないからです。そのため、MySQLでは全文検索で検索したレコードに対して文書IDでソートや絞込みを行うため、時間がかかってしまいます。

この問題を解決するのが、SennaのMySQL バインディングに含まれる2ind-patchです。このパッチを適用すれば、FULLTEXT インデックスと別のインデックスと組み合わせて利用できるようになります³。FULLTEXT インデックスとPRIMARY インデックスを両方利用することで、このようなクエリに対する性能が大きく向上します。

全文検索機能を用いた実際のアプリケーションでは、たとえば特定のドメインで絞り込んで検索したり、検索結果を適合度だけでなく日付順などでソートしたい場合がよくあります。そのような場合には2ind-patchが効力を発揮します。

4.2 高速で実用的なMySQL + Senna

このように、SennaのMySQL バインディングを用いることで、MySQLにビルトインされたFULLTEXT インデックスが持つ4つの問題点を解決し、MySQLを実用的な日本語対応の検索エンジンに変身させることができます。

また、Sennaのインデックスはビルトインのインデックスを置き換える形で組み込まれるため、ビルトインのFULLTEXT機能とまったく同じ使い勝手で自然に使用できます。導入方法などは、Sennaの公式ページ（<http://qwik.jp/senna/>）に書いてあるので、MySQLを用いたアプリケーションを稼働させている方は、ぜひSennaのMySQL バインディングを導入して、全文検索の便利さをお試しく下さい。

³クエリの種類によっては2つのインデックスが使用できない場合もある。



5 Perl バインディングを使おう！

Senna をプログラミング言語から使う例として、Senna の Perl バインディングを紹介します。これを用いることで、hash (連想配列) や dbm などのようなキーと値が対になったさまざまなデータ集合に、全文検索機能を付け加えることができます。ここでは、Senna の Perl バインディングの使い方について、具体的な例を交えながら説明しましょう。実際に作る各プログラムは以下の URL からダウンロードできます。

- Senna Perl バインディング : <http://search.cpan.org/dist/Senna>
- 本章で説明するプログラム例 : <http://dev.razil.jp/archive/osm/podfts.tar.gz>

5.1 基本的な使用方法

Senna の Perl バインディングは、Senna.pm というモジュールの形で提供されています。2006 年 2 月の時点で、Senna.pm は Senna ライブラリ (libsenna) の一部の API のみ利用できます (データの挿入/変更/削除と、それらに対するシンプルな検索が可能)。たとえば新規にインデックスを作成したときの簡単な使用例は、次のようになります。

```
use Senna::Index qw(:all);

my $index = Senna::Index->create('index');
$index->put("hoge1", "ほげ");
$index->put("hoge2", "ひげ");
$index->put("hoge3", "ほげほげ");
```

簡単なコードですが、これだけでもうデータの登録が終了しており、後は search を呼ぶだけで、「ほげ」にマッチするキーを検索できます。

```
my $cursor = $index->search("ほげ");
while (my $result = $cursor->next) {
    print "hit: ", $result->key, "\n";
}
# "hoge1", "hoge2"
```

ただし、これだけだと検索にマッチするキーの値しか取り出せないで、どのような内容の文章がマッチしたのか判断できません。キーに対応するデータを何らかの形で保持/更新していないと、有用な検索とはいえないでしょう。

Perl でこのようなデータの保持/更新を行う場合、データベースやファイルに保存するなどの方法もありますが、最も単純なのは次のようにして連想配列にデータを入れることです。



```
my %hash;  
$hash{hoge1} = "ほげ";  
$index->put("hoge1", "ほげ");
```

しかし、これを毎回行うのは何とも面倒なので、今回は Perl が提供する tie メカニズムを使ってみましょう（リスト 3）。tie は通常、BerkeleyDBなどを連想配列から透過的に使用するために使われますが、ここでは連想配列のデータに更新があったとき、強制的に senna インデックスへの更新をトリガーするために使っています。

ただ、このままではデータ本体がメモリ上に存在する%hashの中に格納されているので、プログラム終了時にインデックスしか残らず、あまり有用ではありません。そこでTie::Sennaで実装されているデータ格納領域を別途指定するオプションを使います（リスト 4）。

これで、%hashに格納されるデータが%storageに格納され、なおかつ\$indexで指定されたSennaインデックスを更新することになります。こうしておけば、\$fileで指定されたファイルにデータ本体が、\$indexで指定されたファイルにインデックスが保存されるので、次回検索をするときにすぐ使えます。

5.2 実用アプリケーションへの第一歩

以上を踏まえたうえで、いよいよアプリケーションを作成してみましょう。ここでは、Perl プログラマに最も身近な存在である POD ドキュメントのインデックスを作って、POD をすべて検索できるようにしてみます。これ以降、コードが長いので主要な部分を抜粋して解説しますが⁴、主な構成は次のようになります。

- podfts/bin/podfts

```
sub main メイン関数
```

- podfts/lib/Pod/FTS.pm（文書の登録/検索部）

```
sub build_index : インデックス作成  
sub make_wanted : ファイルが見つかったときの処理  
sub search : 検索部  
sub wprint : 検索結果の表示
```

⁴コード全体は、前述のとおり次の URL からダウンロードできる。
<http://dev.razil.jp/archive/osm/podfts.tar.gz>



5.2.1 インデックス作成 (build_index 関数)

まず、検索のためのインデックスを作成する必要があります。前述したとおり、取得した元データを Senna に登録する作業です。ここでは検索対象ファイルが保存されているディレクトリと、インデックスファイルなどを格納しているディレクトリをユーザーに問い合わせ、そのディレクトリ内にインデックスを作ります。

検索対象ファイルを探すために、File::Find というモジュールを使います。このモジュールは、ファイルが見つかったときにどういった動作をするのか指定するコールバック関数を渡して利用できます。まずこのコールバック関数を、make_wanted 関数内で作成します (リスト 5)。

5.2.2 ファイルが見つかったときの処理 (make_wanted 関数)

リスト 5 では、どのインデックスファイルを使用するのか調べています。インデックスファイルがすでに存在する場合はそのインデックスを open 関数で開き、存在しない場合は create 関数で新たにデータを作成します。

次に、このインデックスを使用して、Tie::Senna で tie された連想配列を作成、これをリスト 6 のクロージャで使用しています。実際に File::Find に使用されるコールバック関数はこのクロージャなので、ここが一番のキモとなります。

リスト 6 では、まず .pm と .pod のファイルを、Pod::Parser を継承するモジュールでパースし (それ以外のファイルは無視します) 実際にはドキュメントとして扱われるコンテンツのみを抜き出します。ここでようやく senna の登場で、%hash に抜き出したコンテンツとファイルパスを登録します。たとえば、Senna.pm が /Library/Perl/5.8.3/darwin-2level/Senna.pm にあったら、POD の内容がそのファイル名にひも付けられるイメージです。

このクロージャを File::Find に渡すと、インデックスの完成です。後は検索対象の文字列をこれに渡して、search 関数を呼ぶだけで検索ができます。実際の検索コードはインデックス作成時のコードと似ています (リスト 7)。

5.2.3 検索 (sub search 関数)

検索時には、インデックスとデータファイルを指定して \$index を作成し、検索対象文字列を search 関数に渡すだけで検索できます。searchn 関数から戻ってくる \$cursor はイテレータで、ヒット件数を調べるなどの動作を行えます。次はこの個々の検索結果をユーザーが使いやすい形で表示しましょう。



基本動作としては、検索にヒットしたファイル名に対し、そのファイル内で検索文字列にあたる前後の文字列を表示するか、そのファイルの perldoc を直接表示するか選べる形にします。細かい動作はちょっと込み入っているので、全体像だけを抜き出して説明しますが（リスト 8）、次のようになります。

- (1) \$cursor->next 関数で検索にヒットしたキー（この場合ファイル名）を表示し、ユーザーに判断を求める
- (2) ユーザーが「Y」を入力した場合、perldoc 全体を表示するので、perldoc の位置を予測して exec 関数で perldoc を起動する（リスト 9）
- (3) 「s」を指定した場合は、検索対象文字列の周辺を表示するので、%storage 内からデータを持ってきて表示する。実際のコードではテキストをラッピングなどの処理を行っているのでリスト 9 のコードとは少し違うものが必要だが、基本的には%storage に格納されているデータを print する
- (4) 「q」を指定された場合はそこで検索を終了する

ここでは Tie::Senna を使用しませんでした。あえてこだわるのならリスト 10 のようなコードで同じことが実現できます。このコードのポイントは、tied 関数でインデックスの実体を参照するところと、いったん Tie::Senna で tie を実行した連想配列なら%storage を直接参照する必要がない、というところです。

6 最後に

以上、駆け足で Senna の特徴や使用方法について説明してきました。Senna はコンパクトながら高性能であり、さまざまなアプリケーション要件に応えるために高度な機能をたくさん備えています。本稿では説明しきれなかった Senna の機能については、Web ページを参照してください。

<http://qwik.jp/senna/>

7 リスト

リスト 1: body フィールドに「単語」が含まれているレコードを取得するクエリ

```
SELECT * FROM articles WHERE body LIKE '%単語%';
```

リスト 2: body フィールドに「単語」が含まれているレコードを取得するクエリ（ビルトインの FULLTEXT インデックスを使用）



```
SELECT * FROM articles WHERE MATCH(body) AGAINST ('単語')
```

リスト 3: tie を使って、senna インデックスへの更新をトリガー

```
my %hash
tie %hash, 'Tie::Senna', index => $index;

$hash{hoge1} = "ほげ";
my $cursor = tied(%hash)->search("ほげ");
while (my $result = $cursor->next) {
    print $hash{$result->key}, "\n";
}
```

リスト 4: Tie::Senna のデータ格納領域を指定するオプションを使用

```
my %storage;
tie %storage, 'DB_File', $file or die $!;

my %hash;
tie %hash, 'Tie::Senna', index => $index, storage => \%storage;
```

リスト 5: ファイルが見つかったときに動作するコールバック関数

```
my $file = File::Spec->catfile($args{data_dir}, 'podfts.db');
my $index =
    (-f "$file.SEN.i") ?
        Senna::Index->open($file) :
        Senna::Index->create($file, SEN_VARCHAR_KEY, undef, SEN_ENC_UTF8);
```

リスト 6: データを Senna インデックスに登録するクロージャ

```
return sub {
    return if (! -f $File::Find::name ||
        $File::Find::name !~ /\.(?:pm|pod)$/);

    my $indexer = Pod::FTS::Indexer->new;
    $indexer->parse_from_file($File::Find::name);
    if (length($indexer->{text}) <= 0) {
        return;
    }
}
```



```
}  
  
$hash{$File::Find::name} = $indexer->{text};  
}
```

リスト7: 検索のためのコード

```
my $file = File::Spec->catfile($datadir, 'podfts.db');  
my %storage;  
tie %storage, 'DB_File', $file or die $!;  
  
my $index = Senna::Index->open(  
    File::Spec->catfile($datadir, 'podfts.db'));  
my $cursor = $index->search($query);  
  
print $cursor->hits(), " documents found\n";
```

リスト8: 検索にヒットした場合の処理 (概要)

```
while (my $result = $cursor->next) {  
    print " + ", $result->key, "\n    Open this document? [y/N/s/q] ";  
  
    my $ack = <STDIN>;
```

リスト9: 検索にヒットした場合の表示方法選択

```
my $ack = <STDIN>; (1)  
if ($ack =~ /^y(?:es)?$/i) { (2)  
    require File::Basename;  
    my $dirname = File::Basename::dirname($^X);  
    my $perldoc = $dirname !~ /\^\.$/ ?  
::Spec->catfile($dirname, 'perldoc') : 'perldoc';  
    exec($perldoc, '-F', $result->key) or die $!;  
} elsif ($ack =~ /^s(?:nippet)?$/i) { (3)  
    print $storage{$result->key()}, "\n";  
} elsif ($ack =~ /^q(?:uit)?$/i) { (4)  
    last;  
}
```

リスト10: Tie::Senna を使用した場合のコード



```
my %storage;
tie %storage, 'DB_File', $file or die $!;

my $index = Senna::Index->open(
    File::Spec->catfile($datadir, 'podfts.db'));
my %hash;
tie %hash, 'Tie::Senna', index => $index, storage => \%storage;

tied(%hash)->search($query);
    :
    :
print $hash{$result->key()}, "\n";
```