Mettez un Sphinx dans votre moteur de recherche !

A l'heure où les bases de données sont de plus en plus volumineuses et les interfaces de recherche de plus en plus riches et pertinentes, les index FULLTEXT des SGBD ne suffisent plus. Lenteur et pauvreté fonctionnelle sont leurs points faibles. Alors, pourquoi ne pas utiliser un outil spécialisé dans l'indexation et l'interrogation de données ? Cet article vous propose de découvrir la richesse et la simplicité d'utilisation d'un tel outil : Sphinx.

1.Comment Sphinx s'imposa à nous

Que tous ceux qui exploitent des bases de données (très) volumineuses et qui aspirent à optimiser les temps de réponse des recherches lisent ces lignes !

Voilà comment tout a commencé :

Etape 1: "où les temps de réponse se dégradent..."

Depuis 2 ans, nous exploitons une base sous MySQL à l'aide de scripts PHP. Jusque là, rien d'original. Lors de sa création, cette base contenait environ 30 millions de lignes. Mise à jour quotidiennement, elle en contient plus de 50 millions aujourd'hui !

Les scripts PHP permettent aux utilisateurs d'effectuer des recherches dans cette base. Ces recherches peuvent être exactes, partielles ou même phonétiques (voir **[LMAG1]**). Toujours est-il que les temps de réponse se dégradent mois après mois, devenant insupportables pour les utilisateurs.

Etape 2 : "où il faut développer une v2..."

La décision logique s'impose alors : il faut penser une nouvelle architecture qui améliore drastiquement les temps de réponse. Les idées qui viennent immédiatement à l'esprit sont :

- l'amélioration du schéma de la base
- l'utilisation de machines plus puissantes voire beaucoup plus puissantes, avec énormément de RAM !
- l'utilisation d'un cluster MySQL
- évaluer l'impact d'un partitionnement des tables

Etape 3 : "où on élimine la solution standard et complexe..."

Nous prévoyons d'acheter de nouvelles machines et, en parallèle, nous commençons par refondre le schéma. Le gain de temps est de l'ordre de 60%. Insuffisant : passer de 1 minute 30s à 30s reste toujours insupportable...

Allons-nous nous orienter vers le cluster de type MySQL-NDB ? De par sa complexité et notre manque de recul, nous hésitons.

Etape 4 : "où on nous propose la solution miracle..."

Intervient alors Damien.M (je cite son prénom car il le mérite amplement !). Il nous propose une solution originale et, selon ses dires, "hyper-performante". Pour étayer son affirmation, il indique que le site *DailyMotion* utilise cet outil pour rechercher les vidéos en fonctions des mots saisis par les internautes. Une première maquette, assez vite réalisée, prouve qu'il a raison. Ce fameux outil, **Sphinx**, est tout simplement impressionnant, décoiffant, bluffant de performances !

2.Principe de fonctionnement

Sphinx est l'acronyme de *SQL Phrase INdeX*. C'est un logiciel OpenSource disponible sur le site **www.sphinxsearch.com**. Ce logiciel est écrit en C++, l'accent a été clairement mis sur les performances.

Sphinx est un moteur de recherche FULLTEXT qui indexe des données issues de sources diverses (MySQL, PostgreSQL et flux XML) et offre ensuite des API pour les consulter. Simple, non ?

Prenons l'exemple de notre base de 50 millions de lignes : vus les volumes de données, une recherche avec un **`SELECT`**, des jointures et des clauses **`LIKE`** est très consommateur de ressources (CPU et mémoire). C'est exactement là que Sphinx peut apporter son savoir-faire.

Dans un premier temps, Sphinx doit indexer les données de la base pour créer ses propres index. C'est le processus **`indexer`** qui se charge de cette "compilation" (voir figure 1, étape 1).

Pour rechercher des données dans l'index, les programmes disposent d'une API qui interroge le processus **`searchd`** (étape 2). Ce processus retourne des identifiants qui correspondent à des items de la base (étape 4).

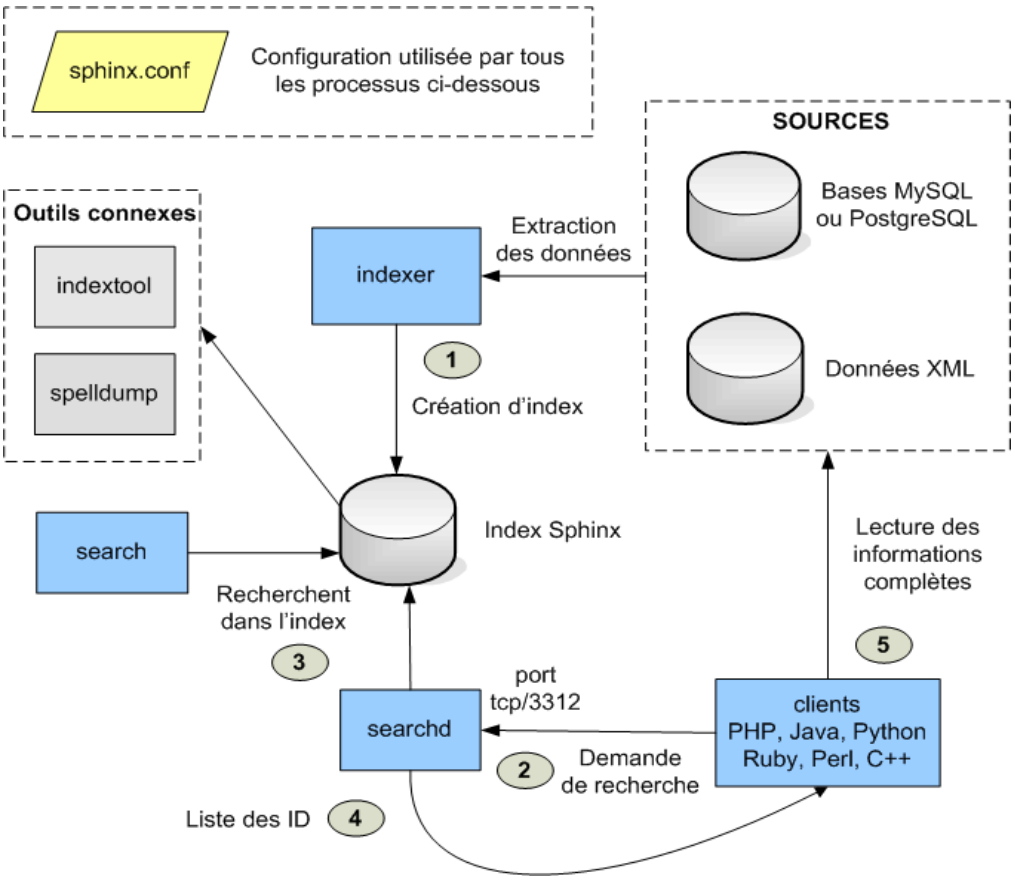
Pour le processus client, une recherche se déroule donc en deux étapes :

– rechercher dans l'index de Sphinx et obtenir des ID (étapes 2,3,4)

– rechercher les informations dans la véritable base à l'aide des ID obtenus (étape 5)

Certes, ce processus ajoute une étape par rapport à une recherche directe en base, mais au vu des performances obtenues, cela vaut vraiment la peine de l'étudier et de le déployer. Dans une de ses présentations, l'auteur, Andrew Aksyonoff, précise que le temps d'indexation est 50 à 100 fois plus rapide que MySQL lorsqu'il crée un index FULLTEXT et jusqu'à 1000 fois plus rapide pour des recherches retournant de très nombreuses lignes (il annonce un temps de réponse < 0.1s pour des index Sphinx de plusieurs Go)... et c'est vrai ! Epoustouflés vous allez être !

figure 1 :



3.Installation et premiers tests

Soyons brefs : vous téléchargez la version 0.9.8.1 ou 0.9.9-rc2 à partir du site **www.sphinxsearch.com**. Vous extrayez les sources dans un répertoire (**`/opt`** au hasard), puis vous lancez les commandes **`./configure`**, **`make`**, **`make install`**.

Par défaut le support de MySQL est activé. Pour le support de PostgreSQL, il faudra indiquer **`--with-pgsql`** sur la ligne de commande de **`./configure`**.

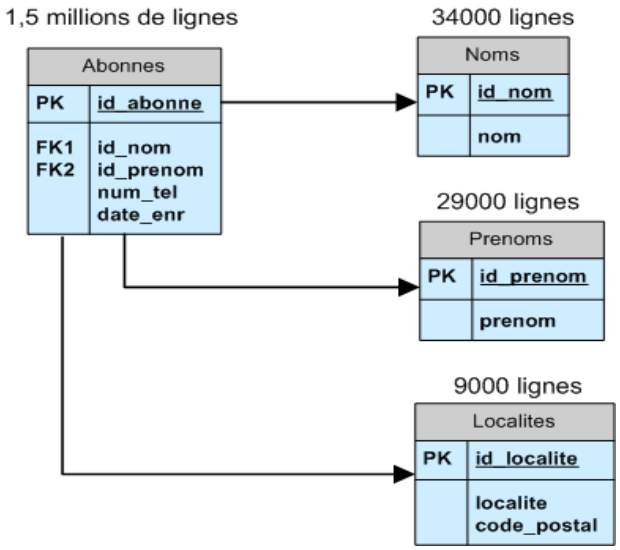
La page **http://www.sphinxsearch.com/docs/manual-0.9.9.html#quick-tour** vous donne une procédure pour tester votre installation avec une mini-table à injecter dans la base test de MySQL. Nous allons plutôt travailler d'emblée sur un cas plus réaliste.

4.Un exemple concret, vite !

4.1.Schéma de la base de démonstration

Le scénario est le suivant : nous stockons en base les informations communiquées par les internautes qui ouvrent un compte pour accéder à nos services. Cette base, nommée **`Abonnés`**, contient environ 1,5 millions de lignes réparties en plusieurs tables (voir figure 2), dont le nom des colonnes est assez explicite :

figure 2 :



Munis de toutes ces données, notre objectif est de permettre des recherches FULLTEXT sur les noms, les prénoms et les noms des localités.

Par exemple, si la valeur de recherche est **`JEAN`**, nous devons théoriquement retourner la liste des internautes dont le nom propre est **`Jean`** ou bien le prénom est **`Jean`** ou bien qui habite **`Saint-Jean-de-quelque-chose`** !

4.2.Création de l'index de Sphinx

Toute la configuration est contenue dans le fichier **`sphinx.conf`**. Lors de l'installation de Sphinx, un fichier **`/usr/local/etc/sphinx.conf.dist`** a été créé. Recopiez-le sous le nom **`/usr/local/etc/sphinx.conf`**. Pour des raisons de simplicité, nous appellerons **`sphinx.conf`**, le fichier de configuration, sans nommer son chemin d'accès.

Ce fichier de configuration contient au moins 4 blocs :

- un bloc qui définit une source de données: ici, ce sera notre base **`Abonnés`**
- un bloc qui définit un index à créer à partir de la source de données
- la configuration de la commande d'indexation : **`indexer`**
- la configuration du démon d'interrogation de l'index : **`searchd`**

4.2.1.Définition de la source de données

Nous devons naturellement indiquer le nom du serveur de base de données, le nom de la base, quel nom d'utilisateur et quel mot de passe utiliser, mais le paramètre le plus important concerne la requête SQL que Sphinx va jouer pour retirer les informations à indexer.

Dans notre exemple, nous voulons effectuer des recherches sur toutes les noms, prénoms et localités. La requête est donc une jointure entre les 4 tables :

```
SELECT Abonnes.id_abonne,Noms.nom,Prenoms.prenom,Localites.localite
    FROM Abonnes
        INNER JOIN Noms ON Noms.id_nom = Abonnes.id_nom
        INNER JOIN Prenoms ON Prenoms.id_prenom = Abonnes.id_prenom
        INNER JOIN Localites ON Localites.id_localite = Abonnes.id_localite
```

Sphinx impose toutefois une contrainte tout à fait légitime : la première colonne du **SELECT** doit être la clé sur les objets indexés. Dans notre cas, cette clé est l'attribut **id_abonne** de la table **Abonnes**. La valeur de celle clé doit être un entier unique supérieur à 0 (voir **[SPH9112]** pour les contraintes sur la valeur de **sql_query**).

Appelons **source1** notre source de données, le fichier **sphinx.conf** doit maintenant contenir la description de notre source :

```
source source1
{
 sql_host = localhost
 sql_user = sphinx
 sql_pass = sphinx
 sql_db = Abonnes

 # Et voilà notre SELECT :
 sql_query = SELECT Abonnes.id_abonne, Noms.nom AS nom, \
                    Prenoms.prenom AS prenom, Localites.localite AS localite \
             FROM Abonnes \
                 INNER JOIN Noms ON Noms.id_nom = Abonnes.id_nom \
                 INNER JOIN Prenoms ON Prenoms.id_prenom = Abonnes.id_prenom \
                 INNER JOIN Localites ON Localites.id_localite = Abonnes.id_localite
}
```

4.2.2.Définition de l'index

Un index est caractérisé par 2 paramètres obligatoires : le nom de la source d'où lire les données et le répertoire avec le préfixe des noms de fichiers qui contiendront les informations sur l'index. Appelons **index1** notre index, sa définition dans le fichier **sphinx.conf** est alors :

```
source index1
{
 source = source1

 # 1) le répertoire /var/sphinx/data/ doit exister
 # 2) /var/sphinx/data/index1 n'est pas un répertoire, c'est la racine des noms des fichiers
 #    qui composent l'index créé par Sphinx
 path = /var/sphinx/data/index1
}
```

4.2.3.Configuration des processsus indexer et searchd

Le fichier **sphinx.conf** contient aussi 2 rubriques pour fixer les paramètres des processus **indexer** et **searchd**. Nous laissons les valeurs par défaut qui y sont consignées :

```
indexer
{
 # valeurs par défaut
 ...
}

searchd
{
 # valeurs par défaut
 ...
}
```

4.2.4.Premiers tests et aide à la mise au point

La première étape consiste à compiler l'index :

```
# indexer -c sphinx.conf index1
Sphinx 0.9.8.1-release (r1533)
Copyright (c) 2001-2008, Andrew Aksyonoff

using config file 'sphinx.conf'...
indexing index 'index1'...
WARNING: zero/NULL document_id, skipping
collected 1529908 docs, 38.8 MB
sorted 6.2 Mhits, 100.0% done
total 1529908 docs, 38813635 bytes
total 349.732 sec, 110981.07 bytes/sec, 4374.52 docs/sec
```

Le temps nécessaire à cette opération dépend naturellement du volume de données à indexer et de la complexité de la requête SQL (le **`SELECT`** défini au §4.2.1). Ici, 1529908 documents, c'est à dire Abonnés, ont été indexés.

L'outil **`search`** (et non pas **`searchd`**) permet d'interroger directement les index. Par exemple, pour connaitre les Abonnés dont le nom, le prénom ou la localité contient le mot "Jean", il suffit de lancer la commande suivante :

```
$ search -c sphinx.conf -i index1 Jean
...
using config file 'sphinx.conf'...
index 'index1': query 'Jean ': returned 1000 matches of 108376 total in 0.016 sec

displaying matches:
1. document=41651, weight=2
2. document=67612, weight=2
...
19. document=297863, weight=2
20. document=307259, weight=2

words:
1. 'jean': 108376 documents, 109022 hits
```

L'affichage précédent indique que **`search`** a trouvé 108376 Abonnés correspondant au critère de recherche mais que seulement les 1000 premiers sont retournés (nous verrons plus loin comment classer les résultats selon nos propres critères). L'option **`-i`** permet de nommer l'index à interroger. L'option **`-l`** (absente ici) permet d'indiquer le nombre de lignes à retourner (20 par défaut).

Les valeurs des documents retournées sont en réalité les valeurs des attributs **`Abonnes.id_abonnes`** puisque cet attribut est le premier nommé dans la requête **`sql_query`** (voir §4.2.1). Cette liste de valeurs n'est pas propice à un diagnostic rapide sur la pertinence des résultats retournés. Nous allons améliorer ce compte-

rendu de recherche en faisant en sorte que la commande **search** se charge d'aller chercher elle-même dans la base **Abonnes** les informations associés à l'attribut **id_abonne**.

Pour cela, nous donnons une valeur au paramètre **sql_query_info** dans la définition de la source **source1** :

```
source source1
{
 ...
 sql_query_info = SELECT Abonnes.id_abonne,Noms.nom,Prenoms.prenom,Localites.localite \
                    FROM Abonnes \
                      INNER JOIN Noms ON Noms.id_nom = Abonnes.id_nom \
                      INNER JOIN Prenoms ON Prenoms.id_prenom = Abonnes.id_prenom \
                      INNER JOIN Localites ON Localites.id_localite = Abonnes.id_localite \
                      WHERE Abonnes.id_abonne = $id
}
```

Le comportement de **search** va changer : pour chaque valeur de **id_abonne**, **search** va effectuer la requête SQL indiquée par **sql_query_info** en ayant pris soin de substituer le paramètre **$id** par la valeur de **id_abonne** ! Le résultat d'un test de recherche est maintenant plus explicite :

```
$ search -c sphinx.conf -i index1 Jean
...
using config file 'sphinx.conf'...
index 'index1': query 'Jean ': returned 1000 matches of 108376 total in 0.015 sec

displaying matches:
1. document=41651, weight=2
        id_abonne=41651
        nom=Saint-Jean
        prenom=Jean
        localite=...valeur masquée ici...
2. document=67612, weight=2
        id_abonne=67612
        nom=Jean
        prenom=Jean-Yves
        localite=... valeur masquée ici...
3. document=144390, weight=2
        id_abonne=144390
        nom=...valeur masquée ici...
        prenom=Jean-Luc
        localite=Mont Saint Jean
...
```

Mais ce n'est pas tout, Sphinx offre plusieurs modes de recherche ! Nous pouvons par exemple, rechercher les abonnés qui habitent à **Foix** (ville suffisamment petite pour ne pas nous noyer sous des tonnes de résultats) et dont le prénom est **Jean** :

```
$ search -c sphinx.conf -i index1 -e "@localite Foix @prenom Jean"
```

```
...
using config file 'sphinx.conf'...
index 'index1': query '@localite Foix @prenom Jean ': returned 310 matches of 310 total in 0.007 sec

displaying matches:
1. document=1238554, weight=2566
        id_abonne=1238554
        nom=...nom masqué..
        prenom=Jean
        localite=Foix
2. document=1238650, weight=2566
        id_abonne=1238650
        nom=...nom masqué...
        prenom=Jean-Noël
        localite=Foix
...
```

Le paragraphe §7 vous en dira beaucoup plus sur les modes de recherche proposés par Sphinx.

4.3.Script de recherche en PHP

L'ultime étape de notre prise en main consiste à montrer comment utiliser les index de Sphinx à partir d'une application tierce. Avec les sources de Sphinx, vous trouverez un sous-répertoire **`./api/`** qui contient les API pour les langages Java, C, Ruby, PHP et Python. Nous nous intéresserons ici à l'API PHP.

Les API implémentent en réalité un protocole réseau qui permet de se connecter à un service distant qui va consulter les index de Sphinx. Ce service réseau est assuré par le processus **`searchd`** qu'on lancera ainsi :

```
# searchd -c sphinx.conf
Sphinx 0.9.8.1-release (r1533)
Copyright (c) 2001-2008, Andrew Aksyonoff

using config file 'sphinx.conf'...
creating server socket on 0.0.0.0:3312
```

Par défaut, il se met en mode "démon" et est à l'écoute du port 3312/tcp (voir figure 1).

L'algorithme d'interrogation de l'index suit, en PHP, la structure suivante :

- inclusion du fichier **`sphinxapi.php`** fourni avec la distribution de Sphinx
- créer un objet **`SphinxClient`**
- positionner les options de requêtage
- se connecter au serveur (démon **`searchd`** distant)
- envoyer la requête

– lire les réponses (ici: des valeurs de **`id_abonne`**)
– fermer la connexion
– utiliser les valeurs obtenues pour consulter la base de données

Traduit en PHP et adapté à notre exemple de recherche, l'algorithme précédent devient :

```
<?php
// 1-Invocation de Sphinx
// ----------------------------
include "sphinxapi.php";

$sp = new SphinxClient();
$sp->SetServer("localhost",3312); // valeurs par défaut
$sp->SetMatchMode(SPH_MATCH_ALL);  // voir §7

$res = $sp->Query("Jean","index1"); //on cherche Jean dans l'index index1

if ($res === false) {
  die("Echec: ".$sp->GetLastError()."\n");
}

// $res est un tableau associatif :
print "La recherche a retourné ${res['total']} items sur ${res['total_found']} en ${res['time']}
secondes\n";

// 2-Récupération des données dans la base
// ----------------------------------------------------------
if (isset($res['matches'])) {
  // Connexion à la base (ici: MySQL)
  if (($dbh = mysql_connect("localhost")) === false) {
    die("Erreur de cnx MySQL : ".mysql_error());
  }
  mysql_select_db("Abonnes") or die("Impossible de sélectionner la base: ".mysql_error());

  printf("%7s %5s %20s %20s %30s %10s %10s\n",
          "ID","Poids","Nom","Prenom","Localite","NumTel","Date");

  // Boucle optimisable !
  foreach ($res['matches'] as $docid => $docinfo) {
     $query = "SELECT abo.id_abonne AS id_abonne, Noms.nom AS nom, ".
                              "Prenoms.prenom AS prenom,".
                              "Localites.localite AS localite, abo.num_tel As num_tel ".
                              "abo.date_enr AS date_enr ".
                    "FROM Abonnes AS abo ".
                       "INNER JOIN Noms on Noms.id_nom = abo.id_nom ".
                       "INNER JOIN Prenoms on Prenoms.id_prenom = abo.id_prenom ".
                       "INNER JOIN Localites on Localites.id_localite = abo.id_localite ".
                     "WHERE abo.id_abonne = $docid";

    if (($result = mysql_query($query)) !== false) {
      if (($row = mysql_fetch_array($result,MYSQL_ASSOC)) === false) {
        print "Erreur avec $query: ".mysql_error()."\n";
      }
      else {
        printf("%7d %5d %20s %20s %30s %10s %10s\n",
                $row['id_abonne'],$docinfo['weight'],$row['nom'],$row['prenom'],
                $row['localite'],$row['num_tel'],$row['date_enr']);
      }
      mysql_free_result($result);
    }
  }

   mysql_close($dbh);
}
?>
```

Ce script pourra être déclenché à partir d'un formulaire de saisie qui permettra de choisir les valeurs des critères de recherche.

5.Attributs, tri et pertinence des résultats

Nous sommes donc en mesure d'effectuer des recherches FULLTEXT, mais comment exprimer au mieux la requête suivante "je cherche toutes les valeurs de 'Jean', seulement pour les abonnés qui habitent le département de l'Ariège (09) !"

Avec notre exemple de code PHP précédent, il faut rechercher toutes les valeurs de "Jean", puis ce sera lors de l'interrogation à la base de données qu'on pourra examiner la valeur de l'attribut **`Localites.code_postal`**. Ce fonctionnement n'est pas idéal car on va lire dans la base des informations qui seront finalement éliminées ! Pour optimiser ce cas de figure, mieux vaut laisser à Sphinx la direction des opérations. En effet, Sphinx peut filtrer, regrouper et trier les résultats, un peu comme on écrirait une requête **`SELECT...FROM....WHERE....GROUP BY...ORDER BY...`** en SQL. Accepter que Sphinx fasse le travail évitera de surcharger en requêtes inutiles la base de données.

5.1.Prise en compte des attributs

Lors de la définition de la requête de recherche (avec le paramètre **`sql_query`**), il est possible de lister des colonnes qui ne seront pas indexées en FULLTEXT, mais qui seront utilisées pour filtrer, regrouper ou trier les résultats. Ces colonnes ne peuvent pas (encore) être des chaines de caractères (c'est une des futures fonctionnalités de la version 0.9.10), mais elles peuvent être des entiers, des flottants, des estampilles temporelles...

Si nous voulons donc définir la colonne **`Localites.code_postal`** comme attribut pour Sphinx, nous réécrirons la requête du §4.2.1 ainsi :

```
source source1
{
 ...
 sql_query = SELECT Abonnes.id_abonne,Noms.nom, \
                        Prenoms.prenom,Localites.localite, \
                       Localites.code_postal AS code_postal \
                  FROM Abonnes \
                    INNER JOIN Noms ON Noms.id_nom = Abonnes.id_nom \
                    INNER JOIN Prenoms ON Prenoms.id_prenom = Abonnes.id_prenom \
                    INNER JOIN Localites ON Localites.id_localite = Abonnes.id_localite
 sql_attr_int = code_postal
 ...
}
```

Reconstruisons l'index :

```
# indexer -c sphinx.conf index1
```

et maintenant nous pouvons effectuer des recherches avec des filtres. Soit à l'aide de la commande **`search`** (mais c'est assez limité car on ne peut préciser qu'une valeur exacte de l'attribut filtrant), soit en PHP, à l'aide des méthodes **`SetFilter*()`**.

Avec **`search`**, la commande pourrait être :

```
$ search -c sphinx.conf -i index1 -e "@prenom Jean" -f code_postal 9000
index 'index1': query '@prenom jean ': returned 665 matches of 665 total in 0.046 sec

displaying matches:
1. document=1250085, weight=1556, code_postal=9000
        id_abonne=1250085
        nom=...nom masqué...
        prenom=Jean-Paul
        localite=Saint Jean de Verges
2. document=1254258, weight=1556, code_postal=9000
        id_abonne=1254258
        nom=...nom masqué...
        prenom=Jean
        localite=Saint Jean de Verges
...
```

Ce qui revient à chercher tous les abonnés dont le prénom contient "Jean" et qui sont domiciliés dans la ville dont le code postal est 9000 (Foix et les petites communes avoisinantes !).

En PHP, on peut invoquer les méthodes **`SetFilter()`** et **`SetFilterRange()`** par exemple. La 1ère permet de lister les valeurs exactes qu'on souhaite inclure ou exclure, la 2ème permet de définir des intervalles à inclure ou exclure. Toutes les valeurs données sont des entiers (il existe aussi **`SetFilterFloatRange()`** et **`SetGeoAnchor()`** pour filtrer les résultats).

Exemple 1 :

```
// Inclus les abonnés dont le code postal est 9000 ou 10000
$sp->SetFilter("code_postal",(9000,10000));
```

Exemple 2 :

```
// Exclus les abonnés dont le code postal n'est pas compris entre 1000 et 1000
$sp->SetFilterRange("code_postal",1000,10000,true);
```

Il est possible de placer plusieurs filtres sur un même requête.

5.2.Hiérarchisation des résultats

Ce procédé est dénommé *weighting* dans la documentation de Sphinx. Il s'agit, une fois les documents trouvés, de retourner une liste où les documents les plus pertinents apparaissent en premier.

Sphinx utilise deux algorithmes et permet - dans une certaine limite - au développeur de proposer sa propre pondération. Pour corser le tout, ces algorithmes sont appliqués différemment selon les modes de recherche (voir §7 et **`[SPH44]`**) !

Le premier algorithme utilisé par Sphinx s'appelle "*phrase ranking*". Le poids (*weight*) d'un document est directement fonction du nombre de mots recherchés qu'il contient. Par exemple, si on recherche "Jean OU Dupond", un document contenant "Jean" sera pondéré avec la valeur 1, un document contenant "Dupont Jean" sera pondéré avec la valeur 2.

Le deuxième algorithme s'appelle "*statistical ranking*", il tient compte de la fréquence des mots dans tout l'index. Le poids est une valeur flottante comprise entre 0 et 1, le poids affecté à un document est d'autant plus élevé qu'il contiendra soit de nombreuses occurrences des termes recherchés ou bien que ceux-ci seront rares dans tout l'index.

Enfin, le développeur peut associer à chaque champ de l'index un poids spécifique sous forme de valeur entière (la valeur par défaut des champs est 1). Ce poids sera toujours multiplié à la valeur trouvée par le "*phrase ranking*".

Mettons en pratique cette pondération personnalisable sur la recherche du mot "Jean" dans notre index :

```
<?php
...
// "jean" dans le champ "localite" vaut plus que "jean" dans le
// champ "nom" qui vaut lui-même plus que "jean" dans le champ "prenom"
$sp->SetFieldWeights(array('localite' => 30,  'nom' => 10, 'prenom' => 5));
...
?>
```

Le résultat ressemble bien à nos espérances :

ID	Poids	Nom	Prenom	Localite	CP	Date
270033	5556	T******	Jean	Saint Jean de Bonneval	10320	2008-02-01 00:00:00
270324	5556	B******	Jean-Pierre	Saint Jean de Bonneval	10320	2000-03-14 00:00:00
...						
238183	5541	F******	Jean-Jacques	Villenauxe la Grande	10370	1997-01-29 00:00:00
238187	5541	L******	Jean-Pierre	Villenauxe la Grande	10370	1997-01-29 00:00:00
...						

Les "Jean" qui habitent des localités dont le nom contient "Jean" apparaissent en tête de liste.

5.3.Tri des résultats

Par défaut, les résultats sont naturellement triés par pertinence décroissante (voir §5.2). Ce mode de tri est appelé **`SPH_SORT_RELEVANCE`**. Mais on peut spécifier d'autres attributs à utiliser pour effectuer un tri croissant ou décroissant.

Les attributs possibles sont ceux définis dans le fichier **`sphinx.conf`**, à l'aide des paramètres **`sql_attr_xxxx`** (dans notre cas, nous avons défini **`code_postal`**), ou bien ce peut être des noms d'attributs internes, gérés par Sphinx. Leur nom débute par un **`@`**. Les principaux attributs internes sont :

`@id` : valeur de la première colonne du **`SELECT`** indiqué par **`sql_query`**

`@weight` : poids du document trouvé (alias de **`@rank`** et **`@relevance`**)

Les modes **`SPH_SORT_ATTR_DESC`** et **`SPH_SORT_ATTR_ASC`** permettent de spécifier un attribut de tri.

Le mode **`SPH_SORT_EXTENDED`** permet de spécifier une expression "qui ressemble à du SQL". Par exemple, si nous voulons trier les résultats par ordre de pertinence décroissant et code postal croissant, le critère de tri s'écrira :

```
@weight DESC, code_postal ASC
```

c'est à dire, traduit en PHP:

```
<?php
...
$sp->SetSortMode(SPH_SORT_EXTENDED,"@weight DESC, code_postal ASC");
...
?>
```

Le mode **SPH_SORT_TIME_SEGMENTS** est intéresant quand on souhaite classer les résultats par date d'ancienneté. Sphinx utilise un algorithme original : on spécifie le nom d'une colonne de type TIMESTAMP et il répartit les résultats dans des "segments temporels" qui représentent la dernière heure, le dernier jour, la dernière semaine, etc... Dans un même segment, les lignes sont triées par degré de pertinence décroissante.

Et enfin, si vous n'avez pas trouvé votre bonheur, il existe le mode **SPH_SORT_EXPR** qui permet au développeur de préciser une expression de tri. Cette expression peut référencer les attributs, utiliser des opérateurs arithmétiques et booléens ainsi que quelques fonctions mathématiques. L'expression suivante - sans intérêt autre que pédagogique ! - permet de classer les résultats en fonction du poids mais aussi de l'âge et de la valeur du numéro de téléphone :

```
age + num_tel / 100000 + @weight * 10
```

5.4.Regroupement des résultats

Après avoir filtré et trié les résultats, Sphinx nous permet encore de soulager les accès à la base de données en effectuant lui-même l'agrégation des résultats, à la manière d'une clause **GROUP BY** en SQL.

Lorsqu'une agrégation est requise, Sphinx utilise une fonction de regroupement qui permet de différencier les différents agrégats. Il retourne alors uniquement les éléments les plus pertinents de chaque agrégat mais ajoute des attributs internes qui permettent de connaitre le nombre d'éléments rassemblés dans chacun de ces agrégats.

Le critère de regroupement peut être la valeur d'un attribut ou bien le jour, la semaine, le mois ou l'année d'un attribut de type **TIMESTAMP**. Par exemple, nous pouvons écrire en PHP, le code suivant qui va regrouper tous les abonnés dont le prénom est "Jean" et le code postal 9000, par valeur identique de leur date d'enregistrement (attribut **date_enr**).

```
<?php
...
$sp->SetMatchMode(SPH_MATCH_EXTENDED2);
$sp->SetFilter("code_postal",array(9000));
$sp->SetGroupBy("date_enr",SPH_GROUPBY_ATTR);
$res = $sp->Query("@prenom Jean","index1");
...
// Affichage des résultats:
foreach ($res['matches'] as $docid => $docinfo) {
  // faire une requête à la base avec $docid
  // $docinfo['attrs']['@count'] donne le nombre de lignes regroupées pour la journée
  ...
}
?>
```

6.Indexation, alphabet et grammaire

Dans ce paragraphe nous allons explorer le fonctionnement interne de Sphinx quand il indexe des documents. Quelle est sa définition d'un mot ? Qu'indexe-t-il ? Quels sont les paramètres des index qui nous permettent de modifier son comportement ?

Notez bien que les paramètres décrits dans les paragraphes suivants s'appliquant à des définitions d'index, toute modification desdits paramètres nécessitera la regénération des index impactés avec la commande **`indexer`**.

6.1.Découpage du document en mots

L'algorithme de base utilisé par Sphinx consiste à lire le document, le découper en mots, éliminer les majuscules au profit des minuscules (*case folding*). Mais pour réaliser ces opérations apparemment simples, Sphinx a besoin de connaitre l'alphabet utilisé, la liste des caractères qui sont considérés comme des délimiteurs, les correspondances entre caractères.

Le paramètre **`charset_type`** indique si l'encodage des caractères est sur un ou plusieurs octets. Les valeurs possibles sont **`sbcs`** (valeur par défaut) ou **`utf-8`**.

Vient ensuite le paramètre **`charset_table`** qui donne la liste des caractères acceptés avec des règles de correspondance, par exemple :

`A -> a` : indique que la lettre 'A' équivaut à un 'a'

`A..Z -> a..z` : traduit toutes les majuscules en minuscules

`U+A8` : indique le caractère de code A8 en hexadécimal

La valeur par défaut pour le type **`sbcs`** est :

```
charset_table = 0..9, A..Z->a..z, _, a..z, \
                U+A8->U+B8, U+B8, U+C0..U+DF->U+E0..U+FF, U+E0..U+FF
```

qui signifie :

- les chiffres sont laissés tels quels
- les majuscules de A à Z sont traduites en minuscules
- le "souligné" et les minuscules sont inchangés
- les majuscules accentuées sont traduites en minuscules accentuées
- plus un cas particulier pour la langue russe !

Le paramètre **`min_word_len`** (valeur 1 par défaut) indique la taille minimale d'un mot pour qu'il soit indexé.

Le paramètre **`ignore_chars`** permet de lister les caractères ignorés qui ne seront pas considérés comme des délimiteurs. Par exemple, les caractères '**`-`**', apostophes et guillemets ne sont pas considérés par défaut comme des caractères faisant partie de mots, car ils ne sont pas listés par le paramètre **`charset_table`**. Ainsi, si on cherche quelqu'un dont le nom est Jean-Jacques, on aura la liste des "Jean" et la liste des "Jacques" (à moins d'être dans le mode de recherche SPH_MATCH_PHRASE).

En définissant :

```
ignore_chars = -,',"
```

Les caractères indiqués sont simplement ignorés et ne sont donc plus considérés comme des délimiteurs de mots. De manière interne, le prénom "Jean-Jacques" est donc indexé sous la forme "jeanjacques".

Le paramètre **`stopwords`** donne le nom d'un fichier texte qui liste les mots à ne pas indexer. Cela permet d'éliminer de l'index les articles et tous les petits mots non-significatifs (le, la, les, du, de, d, à, etc...).

Notons que Sphinx est aussi capable d'indexer non seulement les mots entiers mais aussi les préfixes et les suffixes (voir **`min_infix_len`**, **`min_prefix_len`** et les paramètres s'y afférents dans **[SPH9215]**).

6.2.Reconnaissance des radicaux

A ce point de notre exposé, Sphinx sait découper un texte en mots, mais si on cherche le mot "boire" alors que le document contient "buvons", la recherche risque d'être infructueuse ! Pour éviter cela, Sphinx peut être configuré pour indexer des radicaux ou des formes normalisées.

Par exemple, le paramètre **`wordforms`** indique le nom d'un fichier texte qui permet d'associer une forme normalisée à un ensemble de mots. Ce fichier contient des lignes de la forme :

```
mot > mot_de_référence
```

Ces définitions peuvent être utilisées pour apporter des règles grammaticales ou bien pour normaliser ou associer des mots. Par exemple :

```
buvons > boire
boivent > boire
windaube > windows
st > saint
ste > sainte
```

Sans fichier **`wordforms`**, si on recherche la liste des abonnés qui habitent une localité dont le nom contient "ste", la réponse est sans appel :

```
$ search -c sphinx.conf -i index1 -e "@localite ste"
...
index 'index1': query '@localite ste ': returned 0 matches of 0 total in 0.000 sec

words:
1. 'ste': 147 documents, 147 hits
```

Aucune réponse ! Regénérons un nouvel index avec la prise en compte du petit fichier **`wordforms`** donné en exemple plus haut. C'est à dire qu'on crée un fichier texte, nommé par exemple, **`radicaux.txt`**, dans lequel on indique au minimum la ligne suivante :

```
ste > sainte
```

Puis, il faut modifier le fichier **sphinx.conf** pour y ajouter le paramètre **wordforms**. Mais, plutôt que modifier la définition de l'index **index1**, nous pouvons définir un nouvel index, qu'on nommera **index2** (!), qui sera identique à **index1** au paramètre **wordforms** près. Les habitués des langages orientés-objets sentent-ils l'astuce ? Nous ajoutons dans le fichier **sphinx.conf**, la définition suivante :

```
index index2 : index1
{
  wordforms = radicaux.txt
}
```

Nous pouvons générer notre nouvel index et reposer la question précédente :

```
# indexer -c sphinx.conf index2
...
$ search -c sphinx.conf -i index2 -e "@localite ste"
...
index 'index2': query '@localite ste ': returned 1000 matches of 11282 total in 0.004 sec
...
1. document=104683, weight=2606, code_postal=02470
        id_abonne=104683
        nom=...nom masqué...
        prenom=
        localite=Marizy Sainte Geneviève
2. document=238919, weight=2606, code_postal=10170
        id_abonne=238919
        nom=...nom masqué...
        prenom=
        localite=Droupt Sainte Marie
...
```

On peut constater que les localités contenant "Sainte" dans leur nom sont bien sélectionnées.

L'élaboration d'un fichier **wordforms** est une tâche qui peut être incommensurable, aussi, Sphinx propose-t-il la commande **spelldump** pour créer un fichier de base à partir des dictionnaires au format **ispell ouMySpell**. Par exemple, pour convertir le dictionnaire fourni avec OpenOffice, on peut exécuter la commande :

```
$ spelldump /usr/lib/openoffice.org/share/dict/ooo/fr_FR.dic \
                /usr/lib/openoffice.org/share/dict/ooo/fr_FR.aff
```

Le fichier résultat est nommé **result.txt** et contient la liste de tous les mots du dictionnaire sous la forme :

```
mot > mot
```

Il ne reste plus qu'à personnaliser la partie "gauche" des règles !!!

Dans le même esprit, le paramètre **exceptions** permet d'associer une suite de mots à un seul mot. Par exemples :

```
Open Source => opensource
Electricité de France => edf
```

6.3.Reconnaissance automatique des radicaux

L'utilisation des dictionnaires de traduction définis par le paramètre **wordforms** doit, à notre avis, rester marginale. Mieux vaut utiliser des algorithmes de radicalisation automatiques (*stemming* en anglais). Ces algorithmes existent "clé-en-main" (voir site **[STEM]**).

La procédure d'installation et d'intégration de **libstemmer_c** est la suivante :

- télécharger le fichier **libstemmer_c.tgz** à partir de **[STEM]**
- extrayez son contenu sous le répertoire des sources de Sphinx. Il va écraser le contenu du sous-répertoire **./libstemmer_c**
- recompilez Sphinx en ajoutant l'option **--with-libstemmer** au script **./configure**

Pour invoquez ce processeur lors de l'indexation, il faut le déclarer à l'aide du paramètre **morphology** :

```
morphology = libstemmer_french
```

Note : vous pouvez tester le fonctionnement de **libstemmer_c** en compilant le programme **stemwords.c** fourni dans le répertoire **./examples** :

```
$ cd examples
$ gcc -o stemwords stemwords.c -I../include -L.. -lstemmer
$ ./stemwords -l french
continuait
continu
maladive
malad
buvons
buvon
```

Ce processeur a le mérite d'exister mais il n'est pas parfait (regardez comment il a radicalisé le terme "buvons" !).

Le paramètre **min_stemming_len** (valeur 1 par défaut) fixe la taille minimale d'un mot pour qu'il soit radicaliser.

6.4.Prise en compte des aspects phonétiques

Reprenons notre exemple type fétiche, mais simulons une faute de frappe et écrivons "Jan" au lieu de "Jean". Que trouve la commande **`search`** ?

```
$ search -c sphinx.conf -i index1 -e "@localite Foix @prenom Jan"
...
index 'index1': query '@localite Foix @prenom Jan ': returned 0 matches of 0 total in 0.000 sec
```

Rien ? Eh oui, notre index ne supporte pas les fautes d'orthographe ! Il est très vraisemblable que vous aurez besoin de modifier le comportement de Sphinx.

Pour cela, le paramètre **`morphology`** va venir à notre secours car il propose une valeur **`soundex`** qui semble correspondre à une notion de phonétique. Toutefois, le code **`src/sphinxsoundex.cpp`** montre que l'algorithme appliqué n'est pas celui qui convient au français...

Il nous faut donc trouver le code d'une fonction de traduction phonétique, puis l'intégrer au code source de Sphinx. En s'inspirant du code exposé dans **[LMAG94]** (téléchargeable à l'adresse **[MAJE]**) et en y apportant une petite modification, on obtient le source d'une fonction de phonétisation utilisable par Sphinx.

Il y a ensuite deux méthodes pour intégrer ce nouvel algorithme :

- méthode simple et suffisante : on écrase le fichier **`src/sphinxsoundex.cpp`** avec notre propre fichier. Dans ce cas, pour invoquer cette fonction lors de la création de l'index, il faudra spécifier dans **`sphinx.conf`** :

```
morphology = soundex
```

- méthode plus longue et moins destructrice : on crée une fonction **`stem_monsoundex()`** et on ajoute le fichier **`sphinxmonsoundex.cpp`** dans le sous-répertoire **`./src`**. Il faut ensuite patcher le fichier **`src/sphinx.cpp`** (c'est facile), enfin, il faudra spécifier dans **`sphinx.conf`** :

```
index index2 : index1
{
 ...
 morphology = monsoundex
 ...
}
```

Regénérons lindex index2, et relançons la recherche précédente :

```
# indexer -c sphinx.conf index2
...
$ search -c sphinx.conf -i index2 -e "@localite Foix @prenom Jan"
...
index 'index2': query '@localite Foix @prenom Jan ': returned 1 matches of 1 total in 0.000 sec

displaying matches:
1. document=1489112, weight=3617, code_postal=9000
```

```
        id_abonne=1489112
        nom=...nom masqué...
        prenom=Jen-Claude
        localite=Foix
        code_postal=09000

words:
1. 'foix': 10023 documents, 10086 hits
2. 'jan': 643 documents, 644 hits
```

Ah ah, on a trouvé un abonné ! Et oui, il y avait une faute d'orthographe sur son prénom, mais en tout cas, la recherche phonétique a fonctionné !

7.Modes de recherche

Sphinx propose plusieurs modes pour exprimer une recherche dans l'index. Chaque mode dispose de sa propre syntaxe.

7.1.Le mode SPH_MATCH_ALL

C'est le mode par défaut : tous les mots donnés comme critères doivent être trouvés, comme s'il s'agissait d'un ET implicite. Par exemple, la commande suivante cherche les abonnés pour lesquels on trouve à la fois "Foix" et "Jean" dans les colonnes indexées :

```
$ search -c sphinx.conf -i index1 "Foix Jean"
...
index 'index1': query 'Foix Jean ': returned 319 matches of 319 total in 0.005 sec
...
words:
1. 'foix': 4944 documents, 4980 hits
2. 'jean': 108376 documents, 109022 hits
```

7.2.Le mode SPH_MATCH_ANY

De manière triviale, lorsqu'on précise plusieurs valeurs, Sphinx agit comme s'il y avait un OU implicite. L'option **`-a`** de la commande **`search`** active ce mode. La requête suivante permet donc d'obtenir la liste des abonnés ayant "Foix" ou "Jean" dans les colonnes indexées :

```
$ search -c sphinx.conf -i index1 -a "Foix Jean"
...
index 'index1': query 'Foix Jean ': returned 1000 matches of 113001 total in 0.028 sec
...
words:
1. 'foix': 4944 documents, 4980 hits
```

```
2. 'jean': 108376 documents, 109022 hits
```

On constate bien une différence de la quantité de résultats retournés dans un cas comme dans l'autre.

7.3.Le mode SPH_MATCH_PHRASE

Les termes recherchés forment une phrase que Sphinx doit retrouver à l'identique dans l'index. L'option **`-p`** de la commande **`search`** active ce mode.

7.4.Le mode SPH_MATCH_BOOLEAN

Ce mode plus puissant offre une syntaxe simple permettant de mieux affiner les recherches. La syntaxe supporte les opérateurs ET, OU, NON ainsi que les parenthèses pour grouper des termes.

Exemples :

`"Jean & Jacques"` : cherche Jean ET Jacques

`"Jean | Jacques"` : cherche Jean OU Jacques

`"Jean -Jacques"` : cherche Jean SANS Jacques

`"Jean !Jacques"` : idem

`"(Jean -Jacques) | (Jean -Luc)"` : cherche Jean mais ni Jean-Jacques, ni Jean-Luc ! (attention, il est interdit de mettre une négation seule)

Ce mode peut être testé avec l'option **`-b`** de la commande **`search`**.

```
$ search -c sphinx.conf -i index1 -b "Dupont & Jean -Foix"
...
index 'index1': query 'Dupont & Jean -Foix ': returned 100 matches of 100 total in 0.004 sec

displaying matches:
1. document=6899, weight=1, a1=2
        id_numero_abonne=6899
        nom=Dupont
        prenom=Jean-Francois
        localite=...ville masquée qui n'est pas Foix...
...
words:
1. 'dupont': 1343 documents, 1343 hits
2. 'jean': 108376 documents, 109022 hits
3. 'foix': 4944 documents, 4980 hits
```

7.5.Le mode SPH_MATCH_EXTENDED2

De loin le mode le plus puissant. Voici quelques exemples de la notation proposée (voir **[SPH41]** pour la documentation complète) :

- le "ET" est implicite, le "OU" est noté par un "**|**" et le NON par un "**-**" ou un "**!**" (comme pour le mode BOOLEAN)
- l'opérateur "**@**" permet de nommer des colonnes et donc de définir des portées de recherches plus précises
  - exemple : **`@nom Jean`** : recherche les abonnés dont le nom ressemble à "Jean"
- les guillemets permettent de rechercher une phrase
- l'opérateur "**~**" (tilde) permet de définir une valeur de proximité
  - exemple : **`"Place Foch"~5`** : indique que les mots "Place" et "Foch" peuvent être séparés d'au plus 2 mots (la chaine "Place xxx yyy Foch" fait 4 mots ce qui est bien inférieur à 5)
- l'opérateur "**/n**" permet de définit un quorum de mots pris dans un ensemble
  - exemple : **`"Jean Dupont Place Foix"/2`** : retourne les entrées pour lesquels au moins 2 des mots listés sont trouvés
- on dispose des ancres chères aux expressions régulières : **^** et **$** qui indiquent respectivement que le terme doit coïncider avec le début ou la fin de la colonne indexée

Gageons que vous arriverez à réaliser une interface utilisateur compréhensible pour utiliser les arcanes de ce mode ! Ce mode peut être testé avec l'option **`-e`** de la commande **`search`** :

```
$ search -c sphinx.conf -i index1 -e "@nom Jean @localite Foix"
...
index 'index1': query '@nom Jean @localite Foix ': returned 6 matches of 6 total in 0.004 sec
...
1. document=1247776, weight=2566, a1=6
        id_numero_abonne=1247776
        nom=...nom masqué...
        prenom= Annie
        localite=Foix
...
words:
1. 'jean': 108376 documents, 109022 hits
2. 'foix': 4944 documents, 4980 hits
```

7.6.Le mode SPH_MATCH_FULLSCAN

Ce mode examine toutes les entrées de l'index, comme si la requête était vide. Seuls les filtres, le tri et le regroupement sont activés (voir **[SPH41]**).

8.Conclusion

J'espère vous avoir communiqué un peu de mon enthousiasme à propos de ce remarquable outil d'indexation. Certes il n'est pas le seul dans sa catégorie (Lucene en est un autre très connu), mais il est probablement le plus rapide et le plus simple à déployer et à utiliser.

Je ne résiste pas à l'envie de vous parler de quelques fonctionnalités complémentaires et dignes d'intérêt :

1) **La recherche distribuée** : si une instance de Sphinx n'améliore pas suffisamment les performances de

votre application, pensez à le configurer en mode distribué ! Dans ce mode, un processus **`searchd`** peut relayer les requêtes à d'autres processus **`searchd`** distants et profiter ainsi d'un cluster pour-pas-cher !

2) **La fusion d'index** : même si Sphinx s'avère très rapide pour créer des index, il ne peut être utilisé pour avoir une vision temps-réel du contenu véritable des bases. Selon votre architecture, lorsque vous recréez l'index, votre application devient indisponible.

Parmi les solutions à ce problème, sachez que Sphinx peut fusionner des index. Le principe est alors de créer un index à partir des données actuelles, puis de calculer un index sur les nouvelles données. Il ne reste plus qu'à fusionner les deux index sachant que ce temps de traitement sera très inférieur au temps de réindexation du contenu total.

Auteur : Jérôme Delamarche - Consultant indépendant en technologies Open Source

Références

[LMAG94] Ajout d'une fonction de reconnaissance phonétique pour MySQL (Linux Magazine n° 94 - Mai 2007)

[MAJE] http://www.maje.biz/downloads/phonex2.c

[SPH] http://www.sphinxsearch.com

[SPH41] http://www.sphinxsearch.com/docs/manual-0.9.9.html#matching-modes

[SPH44] http://www.sphinxsearch.com/docs/manual-0.9.9.html#weighting

[SPH9112] http://www.sphinxsearch.com/docs/manual-0.9.9.html#conf-sql-query

[SPH9215] http://www.sphinxsearch.com/docs/manual-0.9.9.html#conf-min-prefix-len

[STEM] http://snowball.tartarus.org/algorithms/french/stemmer.html