对于结构化数据的搜索：如数据库，用SQL语句；

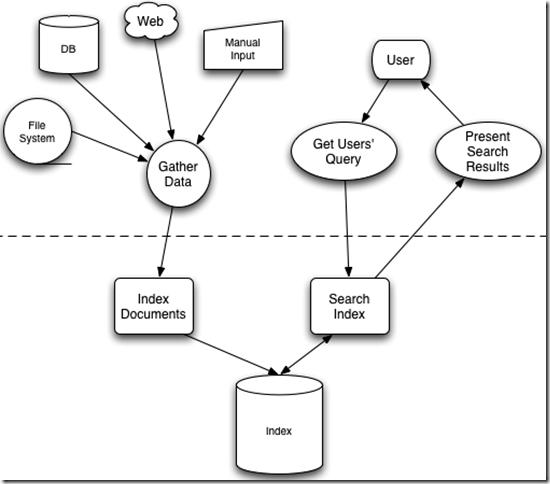
对非结构化数据的搜索：如利用windows的搜索来搜索文件内容，用google和百度也可以搜索大量数据。

非结构化的数据也即对全文数据的搜索主要有两种方法：

一种是顺序扫描法：所谓顺序扫描，即要找内容包含某一字符串的文件，就是一个文档一个文档的看，对于每一个文档，从头看到尾，如果此文档包含此文档，则此文档为我们要找的文件，接着看下一个文档，知道扫描完所有的文件。

一种从非结构化数据中抽取出的然后重新组织的信息，我们称之为索引。

Lucene的检索过程，也是全文检索的一般过程：



全文检索大体分两个过程，索引创建和搜索索引：

索引创建：将现实世界中所有的结构化和非结构化数据提取信息，创建索引的过程。

搜索索引：就是得到用户的查询请求，搜索创建的索引，然后返回结果的过程。

全文搜索大致存在三个问题：

索引里到底存些什么？如何创建索引？如何对索引进行搜索？

**索引到底存了什么：**

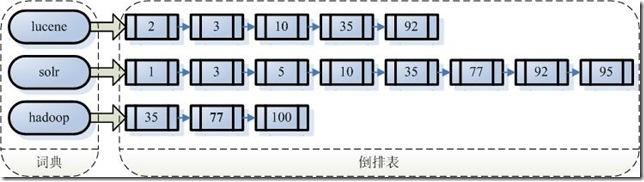
顺序扫描为什么速度慢：是由于我们想要搜索的信息和非结构化数据中所存储的信息不一样造成的。

非结构化数据中所存储的信息是每个文件包含哪些字符串，也即已知文件，欲求字符串相对容易，也即是从文件到字符串的映射。而我们想搜索的信息是哪些文件包含此字符串，也即已知字符串，欲求文件，也即从字符串到文件的映射。两者恰恰相反。于是如果索引总能够保存从字符串到文件的映射，则会大大提高搜索速度。

由于从字符串到文件的映射是文件到字符串映射的反向过程，于是保存这种信息的索引称为反向索引。

反向索引所保存的信息一般如下：

假设我的文档集合里面有100篇文档，为了方便表示，我们为文档编号1到100，得到下面的结构：



左边保存的是一系列字符串，称为词典。

每个字符串都指向包含此字符串的文档(Document)链表，此文档链表称为倒排表(Posting List)。

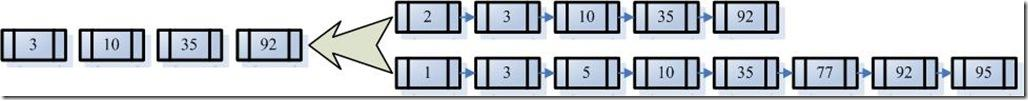
有了索引，便使保存的信息和要搜索的信息一致，可以大大加快搜索的速度。

比如说，我们要寻找既包含字符串“lucene”又包含字符串“solr”的文档，我们只需要以下几步：

1. 取出包含字符串“lucene”的文档链表。

2. 取出包含字符串“solr”的文档链表。

3. 通过合并链表，找出既包含“lucene”又包含“solr”的文件。



**如何创建索引：**

全文检索的索引创建过程一般有以下几步：

**第一步：一些要索引的原文档(Document)。**

为了方便说明索引创建过程，这里特意用两个文件为例：

文件一：Students should be allowed to go out with their friends, but not allowed to drink beer.

文件二：My friend Jerry went to school to see his students but found them drunk which is not allowed.

**第二步：将原文档传给分次组件(Tokenizer)。**

**分词组件(Tokenizer)会做以下几件事情(此过程称为Tokenize)：**

**1. 将文档分成一个一个单独的单词。**

**2. 去除标点符号。**

**3. 去除停词(Stop word)。（为标准分析器的设置，但是默认情况下没有启动）**

所谓停词(Stop word)就是一种语言中最普通的一些单词，由于没有特别的意义，因而大多数情况下不能成为搜索的关键词，因而创建索引时，这种词会被去掉而减少索引的大小。

英语中挺词(Stop word)如：“the”,“a”，“this”等。

对于每一种语言的分词组件(Tokenizer)，都有一个停词(stop word)集合。

**经过分词(Tokenizer)后得到的结果称为词元(Token)。**

在我们的例子中，便得到以下词元(Token)：

“Students”，“allowed”，“go”，“their”，“friends”，“allowed”，“drink”，“beer”，“My”，“friend”，“Jerry”，“went”，“school”，“see”，“his”，“students”，“found”，“them”，“drunk”，“allowed”。

**第三步：将得到的词元(Token)传给语言处理组件(Linguistic Processor)。**

语言处理组件(linguistic processor)主要是对得到的词元(Token)做一些同语言相关的处理。

**对于英语，语言处理组件(Linguistic Processor)一般做以下几点：**

**1. 变为小写(Lowercase)。**

**2. 将单词缩减为词根形式，如“cars”到“car”等。这种操作称为：stemming。**

**3. 将单词转变为词根形式，如“drove”到“drive”等。这种操作称为：lemmatization。**

**Stemming 和 lemmatization的异同：**

* 相同之处：Stemming和lemmatization都要使词汇成为词根形式。
* 两者的方式不同：
  + Stemming采用的是“缩减”的方式：“cars”到“car”，“driving”到“drive”。
  + Lemmatization采用的是“转变”的方式：“drove”到“drove”，“driving”到“drive”。
* 两者的算法不同：
  + Stemming主要是采取某种固定的算法来做这种缩减，如去除“s”，去除“ing”加“e”，将“ational”变为“ate”，将“tional”变为“tion”。
  + Lemmatization主要是采用保存某种字典的方式做这种转变。比如字典中有“driving”到“drive”，“drove”到“drive”，“am, is, are”到“be”的映射，做转变时，只要查字典就可以了。
* Stemming和lemmatization不是互斥关系，是有交集的，有的词利用这两种方式都能达到相同的转换。

**语言处理组件(linguistic processor)的结果称为词(Term)。**

在我们的例子中，经过语言处理，得到的词(Term)如下：

“student”，“allow”，“go”，“their”，“friend”，“allow”，“drink”，“beer”，“my”，“friend”，“jerry”，“go”，“school”，“see”，“his”，“student”，“find”，“them”，“drink”，“allow”。

也正是因为有语言处理的步骤，才能使搜索drove，而drive也能被搜索出来。

**第四步：将得到的词(Term)传给索引组件(Indexer)。**

**索引组件(Indexer)主要做以下几件事情：**

**1. 利用得到的词(Term)创建一个字典。**

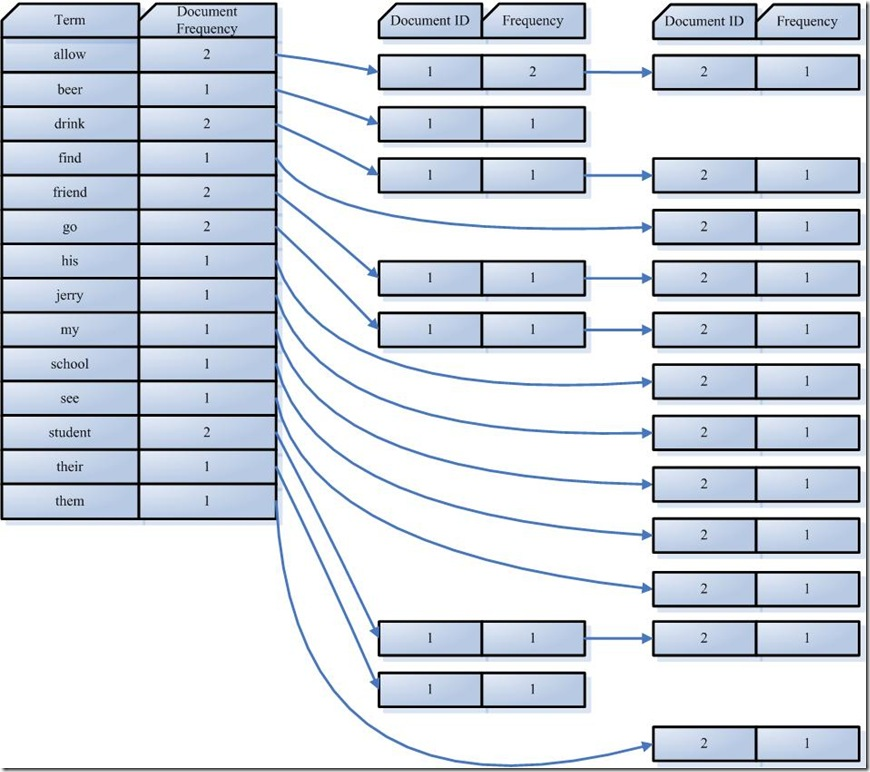
在我们的例子中字典如下：

|  |  |
| --- | --- |
| Term | Document ID |
| student | 1 |
| allow | 1 |
| go | 1 |
| their | 1 |
| friend | 1 |
| allow | 1 |
| drink | 1 |
| beer | 1 |
| my | 2 |
| friend | 2 |
| jerry | 2 |
| go | 2 |
| school | 2 |
| see | 2 |
| his | 2 |
| student | 2 |
| find | 2 |
| them | 2 |
| drink | 2 |
| allow | 2 |

**2. 对字典按字母顺序进行排序。**

|  |  |
| --- | --- |
| Term | Document ID |
| allow | 1 |
| allow | 1 |
| allow | 2 |
| beer | 1 |
| drink | 1 |
| drink | 2 |
| find | 2 |
| friend | 1 |
| friend | 2 |
| go | 1 |
| go | 2 |
| his | 2 |
| jerry | 2 |
| my | 2 |
| school | 2 |
| see | 2 |
| student | 1 |
| student | 2 |
| their | 1 |
| them | 2 |

**3. 合并相同的词(Term)成为文档倒排(Posting List)链表。**



在此表中，有几个定义：

* Document Frequency 即文档频次，表示总共有多少文件包含此词(Term)。
* Frequency 即词频率，表示此文件中包含了几个此词(Term)。

所以对词(Term) “allow”来讲，总共有两篇文档包含此词(Term)，从而词(Term)后面的文档链表总共有两项，第一项表示包含“allow”的第一篇文档，即1号文档，此文档中，“allow”出现了2次，第二项表示包含“allow”的第二个文档，是2号文档，此文档中，“allow”出现了1次。

到此为止，索引已经创建好了，我们可以通过它很快的找到我们想要的文档。

而且在此过程中，我们惊喜地发现，搜索“drive”，“driving”，“drove”，“driven”也能够被搜到。因为在我们的索引中，“driving”，“drove”，“driven”都会经过语言处理而变成“drive”，在搜索时，如果您输入“driving”，输入的查询语句同样经过我们这里的一到三步，从而变为查询“drive”，从而可以搜索到想要的文档。

## 三、如何对索引进行搜索？

到这里似乎我们可以宣布“我们找到想要的文档了”。

然而事情并没有结束，找到了仅仅是全文检索的一个方面。不是吗？如果仅仅只有一个或十个文档包含我们查询的字符串，我们的确找到了。然而如果结果有一千个，甚至成千上万个呢？那个又是您最想要的文件呢？

打开Google吧，比如说您想在微软找份工作，于是您输入“Microsoft job”，您却发现总共有22600000个结果返回。好大的数字呀，突然发现找不到是一个问题，找到的太多也是一个问题。在如此多的结果中，如何将最相关的放在最前面呢？



当然Google做的很不错，您一下就找到了jobs at Microsoft。想象一下，如果前几个全部是“Microsoft does a good job at software industry…”将是多么可怕的事情呀。

如何像Google一样，在成千上万的搜索结果中，找到和查询语句最相关的呢？

如何判断搜索出的文档和查询语句的相关性呢？

这要回到我们第三个问题：如何对索引进行搜索？

搜索主要分为以下几步：

### ****第一步：用户输入查询语句。****

查询语句同我们普通的语言一样，也是有一定语法的。

不同的查询语句有不同的语法，如SQL语句就有一定的语法。

查询语句的语法根据全文检索系统的实现而不同。最基本的有比如：AND, OR, NOT等。

举个例子，用户输入语句：lucene AND learned NOT hadoop。

说明用户想找一个包含lucene和learned然而不包括hadoop的文档。

### ****第二步：对查询语句进行词法分析，语法分析，及语言处理。****

由于查询语句有语法，因而也要进行语法分析，语法分析及语言处理。

**1. 词法分析主要用来识别单词和关键字。**

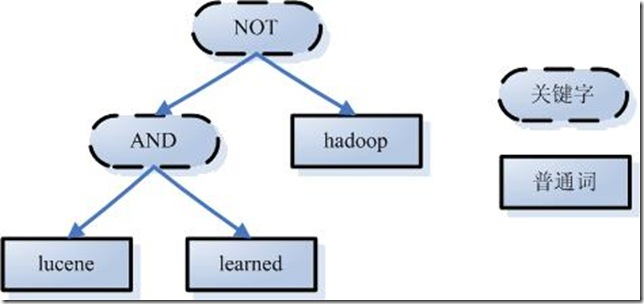
如上述例子中，经过词法分析，得到单词有lucene，learned，hadoop, 关键字有AND, NOT。

如果在词法分析中发现不合法的关键字，则会出现错误。如lucene AMD learned，其中由于AND拼错，导致AMD作为一个普通的单词参与查询。

**2. 语法分析主要是根据查询语句的语法规则来形成一棵语法树。**

如果发现查询语句不满足语法规则，则会报错。如lucene NOT AND learned，则会出错。

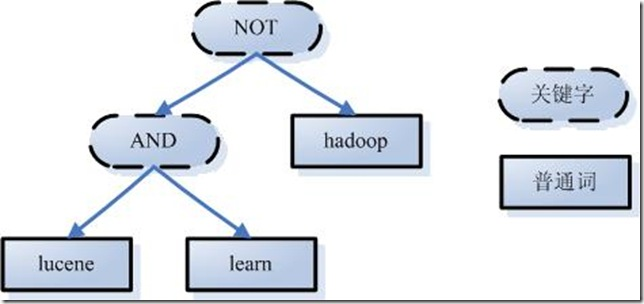
如上述例子，lucene AND learned NOT hadoop形成的语法树如下：



**3. 语言处理同索引过程中的语言处理几乎相同。**

如learned变成learn等。

经过第二步，我们得到一棵经过语言处理的语法树。



### ****第三步：搜索索引，得到符合语法树的文档。****

此步骤有分几小步：

1. 首先，在反向索引表中，分别找出包含lucene，learn，hadoop的文档链表。
2. 其次，对包含lucene，learn的链表进行合并操作，得到既包含lucene又包含learn的文档链表。
3. 然后，将此链表与hadoop的文档链表进行差操作，去除包含hadoop的文档，从而得到既包含lucene又包含learn而且不包含hadoop的文档链表。
4. 此文档链表就是我们要找的文档。

### ****第四步：根据得到的文档和查询语句的相关性，对结果进行排序。****

虽然在上一步，我们得到了想要的文档，然而对于查询结果应该按照与查询语句的相关性进行排序，越相关者越靠前。

如何计算文档和查询语句的相关性呢？

不如我们把查询语句看作一片短小的文档，对文档与文档之间的相关性(relevance)进行打分(scoring)，分数高的相关性好，就应该排在前面。

**如何判断文档之间的关系：**

**首先，一个文档有很多词(Term)组成**，如search, lucene, full-text, this, a, what等。

**其次对于文档之间的关系，不同的Term重要性不同**，比如对于本篇文档，search, Lucene, full-text就相对重要一些，this, a , what可能相对不重要一些。所以如果两篇文档都包含search, Lucene，fulltext，这两篇文档的相关性好一些，然而就算一篇文档包含this, a, what，另一篇文档不包含this, a, what，也不能影响两篇文档的相关性。

因而判断文档之间的关系，首先找出哪些词(Term)对文档之间的关系最重要，如search, Lucene, fulltext。然后判断这些词(Term)之间的关系。

**找出词(Term)对文档的重要性的过程称为计算词的权重(Term weight)的过程。**

计算词的权重(term weight)有两个参数，第一个是词(Term)，第二个是文档(Document)。

词的权重(Term weight)表示此词(Term)在此文档中的重要程度，越重要的词(Term)有越大的权重(Term weight)，因而在计算文档之间的相关性中将发挥更大的作用。

**判断词(Term)之间的关系从而得到文档相关性的过程应用一种叫做向量空间模型的算法(Vector Space Model)。**

下面仔细分析一下这两个过程：

#### ****1. 计算权重(Term weight)的过程。****

影响一个词(Term)在一篇文档中的重要性主要有两个因素：

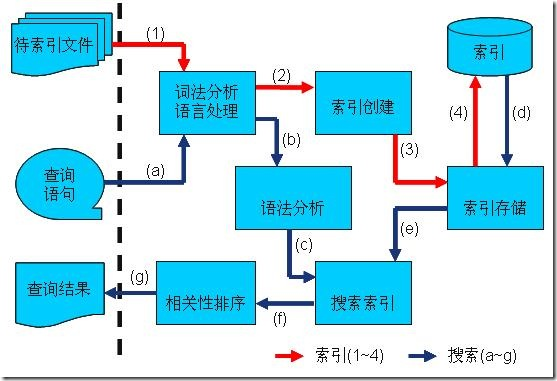
* Term Frequency (tf)：即此Term在此文档中出现了多少次。tf 越大说明越重要。
* Document Frequency (df)：即有多少文档包含次Term。df 越大说明越不重要。

容易理解吗？词(Term)在文档中出现的次数越多，说明此词(Term)对该文档越重要，如“搜索”这个词，在本文档中出现的次数很多，说明本文档主要就是讲这方面的事的。然而在一篇英语文档中，this出现的次数更多，就说明越重要吗？不是的，这是由第二个因素进行调整，第二个因素说明，有越多的文档包含此词(Term), 说明此词(Term)太普通，不足以区分这些文档，因而重要性越低。

这也如我们程序员所学的技术，对于程序员本身来说，这项技术掌握越深越好（掌握越深说明花时间看的越多，tf越大），找工作时越有竞争力。然而对于所有程序员来说，这项技术懂得的人越少越好（懂得的人少df小），找工作越有竞争力。人的价值在于不可替代性就是这个道理。

具体公式等内容见：<http://forfuture1978.iteye.com/blog/546771>

**对上述内容做个总结，如下：**



**1. 索引过程：**

**1) 有一系列被索引文件**

**2) 被索引文件经过语法分析和语言处理形成一系列词(Term)。**

**3) 经过索引创建形成词典和反向索引表。**

**4) 通过索引存储将索引写入硬盘。**

**2. 搜索过程：**

**a) 用户输入查询语句。**

**b) 对查询语句经过语法分析和语言分析得到一系列词(Term)。**

**c) 通过语法分析得到一个查询树。**

**d) 通过索引存储将索引读入到内存。**

**e) 利用查询树搜索索引，从而得到每个词(Term)的文档链表，对文档链表进行交，差，并得到结果文档。**

**f) 将搜索到的结果文档对查询的相关性进行排序。**

**g) 返回查询结果给用户。**