Analizadores morfológicos aplicados al lenguaje natural, aplicaciones para búsqueda de información

Gunnar Eyal Wolf Iszaevich

Instituto de Investigaciones Económicas, UNAM

Resumen

Casi todos nos hemos topado con la necesidad de almacenar una gran cantidad de información generada por humanos (me refiero principalmente a textos extensos), teniendo que facilitar posteriormente la búsqueda sobre de ella. Hay muchas técnicas a las que podemos recurrir — Pero no siempre acudimos al lugar correcto para buscarla.

El lenguaje hablado o escrito por un ser humano se conoce como lenguaje natural. Inicio este trabajo presentando los principales conceptos utilizados en el campo de análisis de lenguaje natural, para presentar algunas técnicas que permiten procesarlo para hacer más simple y efectivo su análisis y más eficaces las búsquedas sobre de él, específicamente basándome en el proyecto Snowball y su aplicación a las bases de datos a través del módulo TSearch2 de PostgreSQL.

1. ¿Qué es un lenguaje y cómo tratarlo?

Un lenguaje es un conjunto de símbolos que se articulan de determinada manera y encierran un determinado significado. Y en el mundo del desarrollo de sistemas, todos nos vemos empujados a escribir una y otra vez sistemas que sepan entender todo tipo de lenguajes.

No, con esto no estoy implicando que el desarrollo de sistemas consista de escribir compiladores una y otra vez — A lo que me refiero es que, viendo esta definición tan amplia como la estoy planteando, prácticamente todo sistema de cómputo puede ser definido como un analizador de un lenguaje determinado. A fin de cuentas, todo sistema

- Toma datos de entrada que están en una representación determinada, que cumple con ciertas reglas
- Procesa internamente estos datos
- Los transforma para convertirlos en una salida coherente
- Posiblemente, los puede volver a colocar en la entrada y volver a trabajar sobre de ellos

No es casualidad que esta definición nos recuerde a la de una máquina de Turing — Una máquina de Turing es, también, un simple *aparato* de comprensión de un lenguaje muy limitado.

Podemos hablar de cuatro tipos básicos de lenguajes, según su nivel de complejidad:

1.1. Lenguajes descriptivos

Son utilizados para describir situaciones, objetos, estados, configuraciones o conjuntos de datos de una manera estática. Son los lenguajes más sencillos de manejar, dado que sólo tenemos que preocuparnos de leer la representación y transformarla para su uso en memoria — Incluso a veces hasta la transformación es innecesaria, podemos utilizarlos directamente.

Si bien su manejo es el más sencillo de todos los lenguajes, hay importantes esfuerzos basándose en los principales estándares (tanto para crear implementaciones compatibles como para aprender de los

errores de malas implementaciones), buscando una mayor simplicidad y legibilidad tanto ante el humano como ante la computadora [SML, Grant 2002, Ben-Kiki 2004, Langdale 2002, Dumbill 2002].

Las herramientas más comunmente empleadas para su análisis son desde subrutinas hechas a la medida y sin ningún método hasta expresiones regulares, analizadores léxicos simples, y en los casos más complejos incluso analizadores gramaticales simples[1] [Hopcroft 2002].

Veremos a los lenguajes descriptivos ser utilizados principalmente para el manejo de archivos de configuración (de texto plano, de texto con jerarquía, de texto con formato estructurado, XML, YAML), así como en datos generados (de forma manual o automática) expresamente para su posterior interpretación (XML, HTML, CSV, DBF, archivos de datos generados por programas de aplicación).

1.2. Protocolos

Un protocolo es un lenguaje que define cómo se llevará a cabo la comunicación entre diferentes entidades (procesos, sistemas, computadoras, etc.), teniendo un universo de posibilidades de interacción claramente acotado. Su principal diferencia con los lenguajes descriptivos es la intervención del *tiempo* y el cambio del estado interno de cada uno de los participantes. En otras palabras, no nos basta ya analizar un pedazo estático de información, ahora tenemos que analizar una *charla* entre dos partes.

La estructura más adecuada para representar esta conversación es un autómata de estado finito, en el que tenemos una serie de *estados* (incluídos algunos inciales y algunos terminales), relacionados por *eventos* que ocasionan transiciones entre ellos.

Al diseñar un protocolo, un principio casi siempre tomado en cuenta es que el protocolo debe ser únicamente un *canal de comunicación*, y debe consumir la menor cantidad posible de recursos, por lo que normalmente analizar a cada uno de los elementos enviados es computacionalmente muy simple — Una violación al protocolo establecido normalmente causa la interrupción completa de la comunicación.

Ahora, no sólamente encontraremos el uso de protocolos para la comunicación entre computadoras remotas, como aquellos detallados por el proceso RFC de Internet [RFC, Bradner 1996], sino también en la comunicación entre procesos (IPC, RPC) dentro de un mismo sistema.

1.3. Lenguajes formales

Aquellos lenguajes creados para permitir la descripción de un *proceso* más de una *situación*, y hechos para permitir flexibilidad en *qué* es lo que van a representar, son conocidos como lenguajes formales.

En un lenguaje formal tenemos una estructura suprema, la *gramática*, a la que toda expresión en este lenguaje debe ceñirse — La representación más común del análisis de estos lenguajes es, por medio de la gramática, convertir una instancia de este lenguaje en un árbol, donde cada nodo representa un concepto y cada rama una relación.

En los lenguajes formales, dada la flexibilidad que deben ofrecer, comenzamos a lidiar con la existencia de ambigüedad, sin embargo, la manera en que podemos resolverla es por medio de reglas claras de precedencia o significado.

Las aplicaciones más comunes que existen de lenguajes formales son, claro está, los compiladores — Sin que esto deje de lado a muchos dominios como los lenguajes tipográficos o de presentación especializados, e incluso ciertos aspectos de los procesadores de textos, que, si bien deberían caber en los lenguajes descriptivos, la flexibilidad que deben exponer es tal que en la práctica son tratados como lenguajes formales.

1.4. Lenguajes naturales

Por último, los lenguajes naturales son aquellos que utilizamos los seres humanos para nuestra comunicación. Todos los lenguajes naturales tienen gramáticas básicas que nos indican cómo deben organizarse los

[1] Con simplicidad no me refiero a que estos analizadores no sean muy complejos — Los lenguajes pueden tener cientos de símbolos, y representar estructuras muy complejas. La complejidad del analizador, sin embargo, no es demasiado alta, y típicamente su rol dentro de un programa está limitado a la importación y exportación de datos

conceptos, pero en todos hay una cantidad espeluznante de excepciones[2].

Los lenguajes naturales están plagados de ambigüedades. Nuestro cerebro, desde que aprendemos a lidiar con el mundo y con la representación que podemos darle, se especializa resolverlas — Pero una y otra vez nos topamos con la realidad que ni siquiera siendo seres humanos somos capaces de resolverlas correctamente de forma consistente.

Parte de esta ambigüedad reside en nuestra gran capacidad para razonar con ausencia de información — Casi toda la comunicación entre personas no explicita todos los puntos que toca, gracias a que los seres humanos estamos entrenados para encontrar el *contexto* al que pertenecen las ideas, englobar las estructuras que nos son presentadas en superestructuras creadas a partir de la información con la que ya contamos, y manejar razonablemente bien los huecos en nuestro conocimiento, aquellos aspectos que no conocemos y que son relevantes para la comprensión de nuevos conceptos.

2. Encontrando la necesidad de un analizador morfológico: Ejemplo de aplicación

El trabajo que yo he realizado va mucho más enfocado a la resolución de problemas específicos que hacia un enfoque teórico. La razón por la que entré en contacto con este tema fue por haber diseñado el sistema Web del Cuerpo Académico Historia del Presente [UPN], de la Universidad Pedagógica Nacional — Un sistema de catalogación y búsqueda de artículos periodísticos relacionados con la educación, que a septiembre del 2005 cuenta con más de 73,000 artículos registrados, y crece a ritmo promedio de 3000 artículos al mes, con un tamaño promedio de 3Kb por artículo, y ocurrencias de artículos de hasta 70Kb. Si bien este sistema fue creado para implementar una búsqueda acotada por categorías expuesta hacia los integrantes del Cuerpo Académico, la interfaz disponible al público en general permite una mucho menor expresividad — y como resultado, la mayor parte de las búsquedas son sobre el texto completo, cuando mucho acotadas por fecha o por región geográfica de la publicación. Estas búsquedas pretenden encontrar las cadenas solicitadas en un universo de más de 200 MB de texto.

2.1. Velocidad

El motor de base de datos que elegimos para guardar la información del sistema es PostgreSQL [PostgreSQL], un motor robusto, eficiente y muy extensible. Si bien este motor está perfectamente a la altura de cualquier otro motor libre o propietario, el uso de nuestro sistema dió con un limitante en su modelo datos: No incluye internamente un sistema de FTI[3]. Esto nos causa el que la primer vez que realizamos una búsqueda de este tipo nos encontremos con tiempos de búsqueda de entre 15 y 30 segundos (ver fig. 1 en la página 6). Para búsquedas subsecuentes, teniendo suficiente memoria, la situación mejora drásticamente, con tiempos de ejecución de hasta el 5 % de la primer solicitud — Pero, además de no ser una solución ni elegante ni suficiente (no podemos esperar una carga constante de solicitudes para garantizar tiempos de respuesta decentes), no es una solución escalable: esta tabla tarde o temprano crecerá más allá de los límites de la memoria física del sistema, con lo cual el efecto del cache dejará de ser tan claro.

2.2. Exceso de exactitud

Una segunda problemática que se nos presenta es la de la de la inexactitud (o más bien, la excesiva exactitud) de los resultados utilizando los operadores comunes de comparación. Por ejemplo: Una palabra muy recurrente en esta base de datos es *educación*. Si solicitamos a la base que nos dé el número de artículos con esta palabra (utilizando el operador *LIKE*), recibimos 5624 registros. Sin embargo, para *Educación* nos da otros 6189, y para *EDUCACIÓN* otros 9. Y si

[2] A excepción de las decenas de *lenguajes naturales artificiales* [IAL], creados con el noble fin de auxiliar a la comunicación entre personas de diferentes culturas... Pero con una cantidad minúscula de usuarios.

[3] Full Text Indexing – indexado de texto completo

agregamos la falta de ortografía más obvia, tenemos otros 221 para *educacion*, 115 para *EDUCACION* y 56 para *Educacion*. El total de registros que cumple con cualquiera de estos criterios es de 8713, lo cual indica que hay empalmes y no podemos simplemente concatenar los resultados. Podemos, sí, utilizar operadores no sensibles a mayúsculas y minúsculas (por ejemplo, búsqueda de expresiones regulares), con lo cual el problema específico aparenta quedar resuelto

Sin embargo, siendo un poco más ambiciosos, este problema puede fácilmente crecer: La palabra *educación* es sólamente una forma de conjugación de muchas que nos presenta un sólo concepto. ¿Qué hay acerca de *educativo* (18759), *educado* (1727), *educador* (1528), *educar* (1125), *educando* (1008), *eduque* (111), y tantas más?

2.3. Primer posible solución: FTI

La primer solución propuesta fue utilizar el módulo FTI [FTI], del *contrib* de PostgreSQL. Este módulo resolvería únicamente la primer problemática. FTI fue el primer acercamiento para crear índices de texto completo en PostgreSQL — Es muy eficiente, pero bastante intrusivo en el diseño de la base de datos (requiere crear tablas adicionales, haciendo referencias débiles por medio de los identificadores de objeto — a fin de cuentas, apuntadores a una dirección en memoria, con todas las vulnerabilidades que eso puede introducir), y únicamente resuelve el primero de nuestros problemas.

FTI es un sistema maduro y utilizable — El equipo de desarrollo de PostgreSQL, sin embargo, advierte desde el 2004 que FTI no debe ser ya utilizado, dado que la infraestructura TSearch2/GiST es ya suficientemente madura para reemplazarlo.

2.4. Índices genéricos: GiST

El primer paso para implementar una mejor búsqueda en texto completo que FTI fue crear un índice genérico para tipos de datos complejos que no formen parte de los provistos por los autores de PostgreSQL. GiST [Helllerstein 1999, Sigaev 2002] fue creado para permitir, exigiendo únicamente la implementación de una interfaz estándar, que el indexado de cada tipo de datos sea especificado por los expertos en cierto campo específico del conocimiento, ya no necesariamente por expertos en la estructura interna de PostgreSQL.

GiST ha sido empleado para el desarrollo de varios módulos contribuídos a PostgreSQL, entre los cuales destacan TSearch2 [Kopciuch 2004, Rhodes 2003-1, Rhodes 2003-2], del cual hablaremos con detenimiento a continuación, y PostGIS [Santili 2005], herramienta que dota a PostgreSQL de tipos de datos aptos para representar todo tipo de operaciones espaciales, geométricas y geográficas.

Una de las principales desventajas del uso de GiST es que, al no soportar el índice accesos concurrentes ni el aprovechamiento del WAL[4], su rendimiento era notablemente inferior al logrado por indexadores internos — Para la versión 8.1 de PostgreSQL, la cual al momento de escribir este artículo está ya en etapa beta, ambos problemas deben haber sido ya resueltos.

2.5. Búsqueda sobre texto completo con TSearch2

TSearch2 implementa un esquema de búsqueda en texto completo empleando un índice GiST, a través de una colección de tipos de datos especializados para este tipo de búsqueda y funciones para su manejo. Sin embargo, más que los detalles técnicos, lo importante de TSearch2 es la riqueza que brinda al implementador de una búsqueda sobre texto en lenguaje natural: Desde el mismo analizador léxico de las consultas nos ofrece construcciones booleanas, ordenado por relevancia, y reducción a lexemas.

No entro en detalles en este texto acerca de la activación y configuración de TSearch2 en una base de datos, es una tarea relativamente simple (siete simples comandos y básicamente una tarea de cortar y pegar), y es tratada con detalle en la guía y referencia de TSearch2 [Rhodes 2003-1, Rhodes 2003-2] y en la presentación correspondiente a este artículo [Wolf 2005] — Me limito a explicar los tipos de datos utilizados. De siete los tipos de datos que son definidos para el uso de TSearch2, los más importantes para su uso son *tsvector* y *tsquery*.

Para que un documento sea indexado por

[4] Write-Ahead Log, bitácora de pre-escritura

TSearch2[5] es necesario convertirlo en un tsvector — Una lista de las palabras (o lexemas, como veremos más adelante) de los que consta, ordenados de manera óptima para búsquedas, incluyendo su posición en el documento. Es importante recordar que el tsvector es guardado *además* del documento mismo — No significa, sin embargo, un incremento importante en el espacio que ocupan las tablas, dado que, por la tendencia de repetición de la gran mayoría de las palabras en los diferentes documentos, su crecimiento es aproximadamente logarítmico.

Un tsquery es una lista de lexemas relacionados por medio de operadores booleanos, utilizado para realizar búsquedas en el campo en que almacenamos los tsvectors. Podemos ver un ejemplo de estos dos tipos de datos en la figura 2 en la página siguiente.

Como parte de la creación del subssitema de TSearch2, veremos también la creación de cuatro nuevas tablas en nuestra base de datos [6]: *pg_ts_dict*, *pg_ts_parser*, *pg_ts_cfg* y *pg_ts_cfgmap*. La primera guarda la lista de diccionarios que tenemos disponibles para utilizar (ver figura 3 en la página siguiente), la segunda la lista de referencias (por OID) a los analizadores disponibles, la tercera la configuración de qué analizador utilizar con qué configuración de *locales*[7], y por último la cuarta indica qué diccionarios aplicar a cada tipo de símbolo entregado por el analizador dependiendo de la configuración de analizador activa o especificada.

[5] Lo cual puede ocurrir de manera manual o a través de un *trigger* (procedimiento disparado por un evento — en este caso, inserción o actualización de un registro). Es preferible hacerlo a través de un trigger, dado que esto nos permite que el índice sea actualizado de inmediato cuando se registre un nuevo artículo, y es transparente a la aplicación, a cambio de un par de milisegundos extra al insertar.

[6] Que, al ser su nombre *pg_ ** como los demás catálogos del sistema de Postgres, no deben hacernos mayor ruido

[7] Esto incluye a diferentes puntos relativos al lenguaje y presentación del sistema — especialmente nos referimos al juego de caracteres e idioma declarado de la sesión

3. TSearch2 aplicando análisis morfológico en español a PostgreSQL a través de Snowball

Tras estas consideraciones, y después de que nuestra base de datos indexó todo el contenido de nuestra tabla, esperaríamos que la tecnología trabaje para nosotros — Sorprendentemente, lo que encontramos fue justamente lo contrario: Si bien las consultas complejas (requiriendo varias palabras) registran una ligera mejoría, las búsquedas simples a través de una instalación estándar de TSearch2 resultaron ser considerablemente más lentas que las que originalmente teníamos. ¿Por qué?

Si hablamos de una aplicación que trata directamente con volúmenes importantes de palabras, no podemos dejar de lado las peculiaridades de cada idioma. Tras hacer un análisis de la frecuencia y de la dispersión de términos, encontramos que hay ciertas palabras demasiado frecuentes, como las preposiciones (la palabra *de* es la más frecuente, aparece en 73846 de los 73869 artículos), la lista total de palabras es ridículamente larga, con 215967 términos únicos, el 46 % de las palabras aparece en un sólo artículo, y el 42 % de los términos aparece únicamente una vez en toda la base de datos (esto es, es empleado sólo una vez y no se repite en el artículo). Al hablar de optimización, [Essig 2004] nos recomienda fuertemente no tener tablas con más de 100,000 términos — y estamos excediendo este límite en más de un 100 %.

3.1. Diferentes diccionarios en TSearch2

En las consultas de la figura 2 en la página siguiente llama la atención el uso de la cadena `'simple'` en ambas funciones. Esta cadena indica qué diccionario utilizará sobre los símbolos generados por el analizador para crear el índice. En la instalación default de TSearch2 (ver figura 3 en la página siguiente) encontramos cinco diccionarios: *simple* (únicamente remueve símbolos de puntuación y espacios, iguala

Figura 1: Análisis de una búsqueda simple sin acotar

```
hist_pres=> EXPLAIN ANALYZE SELECT id FROM articulo WHERE nota LIKE '%analisis%';
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Seq Scan on articulo  (cost=0.00..19121.67 rows=1 width=4) (actual time=212.388..17494.361 rows=2 loops=1)
   Filter: (nota ~~ '%analisis%'::text)
 Total runtime: 17494.490 ms
(3 rows)
```

Figura 2: Uso de los tipos de datos tsvector y tsquery

```
hist_pres=> select to_tsvector('simple','Esta es una historia acerca de un pez. El pez es historia.');
                                      to_tsvector
---------------------------------------------------------------------------------
 'de':6 'el':9 'es':2,11 'un':7 'pez':8,10 'una':3 'esta':1 'acerca':5 'historia':4,12
(1 row)

hist_pres=> select to_tsquery('simple','(una&historia)|(un&pez)&!leyenda');
                   to_tsquery
-------------------------------------------------
 'una' & 'historia' | 'un' & 'pez' & !'leyenda'
(1 row)
```

Figura 3: Lista de diccionarios en la instalación default de TSearch2

```
hist_pres=> SELECT dict_name,dict_comment from pg_ts_dict ;
    dict_name    |                   dict_comment
-----------------+-------------------------------------------------
 simple          | Simple example of dictionary.
 en_stem         | English Stemmer. Snowball.
 ru_stem         | Russian Stemmer. Snowball.
 ispell_template | ISpell interface. Must have .dict and .aff files
 synonym         | Example of synonym dictionary
(5 rows)
```

mayúsculas y minúsculas, e indexa sobre el resultante), *ispell_template* (a través de una interfaz al filtro de verificación de ortografía *ispell* ayuda a converger a la grafía correcta de cada palabra; hay diccionarios de ispell en una gran cantidad de lenguajes, y si bien implementa un análisis morfológico básico, está basado más en que una palabra concuerde con una ocurrencia existente en su diccionario que en el análisis propiamente), *synonym* (un diccionario mínimo de sinónimos en inglés, que ayuda a encontrar conceptos relacionados al solicitado) — Pero especialmente de nuestro interés son *en_stem* y *ru_stem*. Estos dos diccionarios proveen una interfaz al lenguaje Snowball [Porter 2001], que parte de la misma filosofía que el mismo GiST: Reconocer que los expertos en diferentes dominios no requieren ser expertos en cualquier otro dominio, y por tanto, reconocer la necesidad de proveerles una herramienta para crear una biblioteca con la información específica al dominio que ellos manejan, que sea aplicable en cualquier otro terreno. En el caso de Snowball, lo que permite es la implementación de las reglas de *des-conjugación*[8] de diversos lenguajes occidentales[9]. Podemos ver en la figura 4 en la página 9 un ejemplo de la ejecución de consultas similares a las mencionadas en la figura 2 en la página anterior, utilizando el diccionario *en_stem* — Podemos apreciar no sólo que la cantidad de palabras a indexar es la mitad de lo que nos arroja *simple*, y que las cadenas indexadas son las raíces, no casos específicos de las palabras. Además, vemos que el marcado de los resultados pasa también a través de estas mismas funciones.

Martin Porter, autor de Snowball, publicó un algoritmo [Porter 1980] que implementa las reglas de des-conjugación del inglés. Su motivación para escribir Snowball como un lenguaje implementado a través de una biblioteca libre fue la experiencia de ver una gran cantidad de reimplementaciones incorrectas de su algoritmo (publicado en BPCL) en otros lenguajes, y anticipar que esto mismo se presentaría con algoritmos sugeridos para otros algoritmos que en esta materia iban siendo publiados [Kraaij 1995]. El nombre Snowball fue elegido como tributo al lenguaje SNOBOL `http://en.wikipedia.org/wiki/SNOBOL`, especializado en manejo de cadenas y reconocimiento de patrones de la década de los 60.

Dentro del *contrib* de PostgreSQL encontraremos también a `gendict`, herramienta que nos sirve para integrar a nuestra base de datos conjuntos adicionales de reglas de Snowball. Al igual que con TSearch2, no entro en detalles respecto a su instalación y configuración [Chobot 2004, Bartunov 2003, Wolf 2005], dado que hay varios documentos que lo tratan ya directamente. Cabe mencionar que [Chobot 2004, Bartunov 2003] cuentan con ejemplos en que se implementan diccionarios especializados no específicos a un lenguaje, sino que a una aplicación.

Una vez indexada la tabla entera utilizando las reglas en español, observamos una disminución notable del tsvector resultante de cada uno de los artículos — En casos extremos (de notas muy cortas, mayormente de hasta 50 palabras, aunque con algunos casos de notas de tamaño promedio), la estructura resultante es del 30 % de la obtenida a través de las reglas *simple*. El promedio de las notas ronda el 75 %. El 23 % de los tsvectors resultantes son menores al 70 %, 13 % son del 80 % o mayores.

Comparando los datos con los que presentamos en la sección 3 en la página 5, encontramos que el número total de términos ahora se redujo a 99,724 (46 % del tamaño original, y justo debajo del límite sugerido [Essig 2004]). El término que en más artículos aparece ahora aparece sólo en 36,810 artículos, poco menos del 50 %, y el porcentaje de artículos repetidos decrece rápidamente. El porcentaje de términos que aparece en sólamente un artículo se elevó ligeramente, quedando en el 49 %, y el de términos que ocurren una sóla vez en toda la base aumentó también -aunque casi de manera imperceptible- llegando al 43 %. El 14 % de este último porcentaje, además, consiste de números, lo que nos apunta a mayores optimizaciones (aunque con menor impacto) que po-

[8] Utilizo este término a falta de una mejor traducción de *stemming*, el proceso por medio del cual podemos encontrar la raíz o morfema de una palabra, removiéndole los elementos, tanto los que indican conjugación (persona, tiempo, etc.) como aquellos que indican composición (prefijos/sufijos que refieren relaciones con otros conceptos).

[9] Snowball no es aplicable a todos los idiomas — Principalmente los idiomas del sur y del oriente de Asia no son *conjugados* propiamente, sino que las palabras —inmutables— asumen diferentes roles dependiendo del contexto en que son presentadas o de la presencia de ciertas otras palabras en la oración.

demos encontrar.

3.2. Afinando el rendimiento

En este momento, el sistema está en este punto, y lo que aquí detallaremos son refinamientos pendientes que debemos tener concluídos para cuando esta plática sea presentada.

Sólo con utilizar el conjunto de reglas adecuadas para el español logramos una notable mejoría. Sin embargo, el rendimiento no es aún tan espectacular como podría. Quedan muchos puntos, sin embargo, por afinar. Los principales pendientes en este momento son:

Tipos de símbolo El analizador léxico de TSearch2 clasifica cada una de las palabras según lo que parecen (ver figura 5 en la página siguiente) antes de entregarlas al analizador morfológico que le corresponda. Según la configuración especificada en la tabla pg_ts_cfgmap[10], TSearch2 pasa por el analizador construído por Snowball para el español los símbolos reconocidos como palabras con alfabeto latino, palabras con alfabeto latino truncas, palabras con alfabeto latino separadas en sílabas y palabras con alfabeto latino truncas separadas por sílabas. Los URLs, nombres de host y números (enteros, enteros sin signo y flotantes) son pasados por el analizador simple, que únicamente les remueve los signos de puntuación. Ahora, dependiendo del uso que se dé a la aplicación, es posible que varios de estos tipos de símbolo se vuelvan innecesarios, reduciendo más aún el índice generado

Limpiar datos de entrada La base de datos fue llenada filtrando todos los artículos a través del módulo `HTML::Entities` de Perl, para evitar problemas relativos a las diferentes codificaciones en que recibimos los datos (principalmente UTF-8, ISO-8859-1). Esto significa que las palabras acentuadas tienen entidades HTML mezcladas (por ejemplo, *educación* es guardado como *educación*, que TSearch2 expande a las tres entidades *educaci*, *oacute* y *n*). Esto, sobra decirlo, agrega una gran cantidad de imprecisión, y (dado que las reglas de Snowball requieren reconocer palabras completas) hacen crecer la lista de palabras. Esta situación probablemente pueda ser corregida llamando a una función en pgperl (Perl embebido en Postgres), que será llamada al analizar una nueva nota, y reemplazará estas entidades por las letras que representan antes de hacer el análisis. Estimamos que esto reducirá hasta en un 15 % adicional el universo de palabras.

Corregir lógica del sistema A lo largo del desarrollo de este trabajo, se nos han hecho evidentes muchos importantes puntos respecto al correcto empleo de búsquedas en una base de datos. Un importante punto a desarrollar para mejorar el rendimiento de toda aplicación es revisar la manera en que todas las consultas son llevadas a cabo, evitando a la base de datos hacer trabajo innecesario, y reduciendo la cantidad de datos que son enviados entre la base de datos y la aplicación.

4. Conclusiones

Queda mucho aún por avanzar para dar este trabajo por concluído. Tenemos resultados mixtos en lo que respecta a la motivación original, la velocidad de búsqueda: Las búsquedas complejas (conjunción de varios términos) y las búsquedas que no requieren ordenado (que implica encontrar todos los resultados antes de hacer el ordenamiento) son mucho más rápidas que las realizadas por el método tradicional (usando LIKE o expresiones regulares). Las búsquedas que requieren ordenamiento son más lentas. La ventaja que nos da el análisis morfológico por sí sóla vale la pena — aún si la velocidad no resulta tan

[10] La tabla tiene tres columnas: `ts_name`, `tok_alias` y `dict_name`. La primer columna apunta al conjunto de reglas a emplear, sea especificado por el usuario o reconocido por los locales. La segunda indica a qué tipo de símbolos nos estamos refiriendo. La tercera indica qué analizador morfológico recibirá el símbolo en cuestión. Los símbolos que no estén definidos para un juego de reglas son simplemente ignorados.

Figura 4: Ejemplo de consultas a TSearch2 con el diccionario *en_stem*

```
hist_pres=# SELECT to_tsvector('simple','We are stemming and testing for occurrence of common words');
                                                to_tsvector
-----------------------------------------------------------------------------------------------------------
 'of':8 'we':1 'and':4 'are':2 'for':6 'words':10 'common':9 'testing':5 'stemming':3 'occurrence':7
(1 row)

hist_pres=# SELECT to_tsvector('default','We are stemming and testing for occurrence of common words');
                    to_tsvector
--------------------------------------------------
 'stem':3 'test':5 'word':10 'common':9 'occurr':7
(1 row)

hist_pres=# SELECT headline('default','I am not testing this. You are welcome to run any tests
hist_pres'# it defines',to_tsquery('default','test'));
                                    headline
-------------------------------------------------------------------------------
 I am not <b>testing</b> this. You are welcome to run any <b>tests</b> it defines
(1 row)
```

Figura 5: Tipos de símbolo reconocidos por el analizador léxico *default*

```
ts2test=# SELECT * from token_type();
 tokid |    alias     |               descr
-------+--------------+-----------------------------------
     1 | lword        | Latin word
     2 | nlword       | Non-latin word
     3 | word         | Word
     4 | email        | Email
     5 | url          | URL
     6 | host         | Host
     7 | sfloat       | Scientific notation
     8 | version      | VERSION
     9 | part_hword   | Part of hyphenated word
    10 | nlpart_hword | Non-latin part of hyphenated word
    11 | lpart_hword  | Latin part of hyphenated word
    12 | blank        | Space symbols
    13 | tag          | HTML Tag
    14 | http         | HTTP head
    15 | hword        | Hyphenated word
    16 | lhword       | Latin hyphenated word
    17 | nlhword      | Non-latin hyphenated word
    18 | uri          | URI
    19 | file         | File or path name
    20 | float        | Decimal notation
    21 | int          | Signed integer
    22 | uint         | Unsigned integer
    23 | entity       | HTML Entity
```

atractiva.

No podemos dejar de considerar la interfaz al usuario: A lo largo de los años, la gente se ha acostumbrado a hacer búsquedas simples, y a recibir los resultados que encuentren textualmente lo que solicitaron. La mayor parte de los usuarios no sabe manejar una búsqueda con expresiones booleanas — Posiblemente, más por este punto que por cualquier otro, para la aplicación específica que nos concierne, lo más indicado sea mantener dos interfaces, cada una conectada a otro tipo de búsqueda.

Con este trabajo no hemos hecho más que arañar el análisis de lenguaje natural — Este es un campo apasionante en el que ha habido importantes avances — y cada vez más, las aplicaciones prácticas se hacen obvias, y parecen estar más a nuestro alcance.

Referencias

[SML] SML: Simplifying XML `http://www.xml.com/lpt/a/1999/11/sml/index.html`, Robert E. LaQuey, XML.com, noviembre de 1999

[Grant 2002] Look Ma, No Tags `http://www.xml.com/lpt/a/2002/07/24/yaml.html`, Kendall Grant Clark, julio del 2002

[Ben-Kiki 2004] YAML Working Draft `http://yaml.org/spec/current.html`, Oren Ben-Kiki, Clark Evans, Brian Ingerson, diciembre del 2004

[Langdale 2002] Simple Outline XML: SOX `www.langdale.com.au/SOX/`, Langdale Consultants, 2001-2002

[Dumbill 2002] Exploring alternative syntaxes for XML `http://www-128.ibm.com/developerworks/xml/library/x-syntax.html`, Edd Dumbill, IBM DeveloperWorks, octubre del 2002

[Hopcroft 2002] Introducción a la Teoría de Autómatas, Lenguajes y Computación; J. Hopcroft, R. Motwani, J. Ullman, Pearson Educación 2002

[RFC] RFC Editor `http://www.rfc-editor.org/`

[Bradner 1996] RFC 2026, The Internet Standards Process – Revision 3 `ftp://ftp.isi.edu/in-notes/rfc2026.txt`, S. Bradner, Harvard University, 1996

[IAL] International auxiliary language `http://en.wikipedia.org/wiki/International_auxiliary_language`, Wikipedia

[UPN] Cuerpo Académico Historia del Presente `http://anuario.upn.mx/site/`, Universidad Pedagógica Nacional

[PostgreSQL] PostgreSQL `http://www.postgresql.org`

[FTI] PostgreSQL Full Text Indexing `http://techdocs.postgresql.org/techdocs/fulltextindexing.php`

[Helllerstein 1999] GiST: A Generalized Search Tree for Secondary Storage `http://gist.cs.berkeley.edu/gist1.html`, Joe Hellerstein, University of California at Berkeley 1999

[Sigaev 2002] Introduction to GiST `http://www.sai.msu.su/~megera/postgres/gist/doc/intro.shtml`, Teodor Sigaev, Oleg Bartunov, Russian Foundation for Basic Research 2002

[Kopciuch 2004] TSearch2 introduction `http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/`

docs/tsearch-V2-intro.html, Andrew J. Kopciuch, mayo del 2004

[Rhodes 2003-1] TSearch2 guide http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/docs/tsearch2-guide.html, Brandon Craig Rhodes, junio del 2003

[Rhodes 2003-2] The tsearch2 Reference http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/docs/tsearch2-ref.html, Brandon Craig Rhodes, junio del 2003

[Santili 2005] PostGIS Manual http://postgis.refractions.net/docs/postgis.pdf, Sandro Santili 2005

[Wolf 2005] Analizadores léxicos aplicados al lenguaje natural, aplicaciones para búsqueda de información [láminas de la presentación] http://www.gwolf.org/soft/an_lex_leng_nat/, Gunnar Wolf 2005

[Essig 2004] Tsearch2: PostgreSQL Full Text Search Extension http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/docs/oscon_tsearch2/, George Essig, O'Reilly Open Source Convention, julio del 2004

[Porter 2001] Snowball: A language for stemming algorithms http://snowball.tartarus.org/texts/introduction.html, Martin Porter, octubre del 2001

[Porter 1980] An algorithm for suffix stripping http://www.tartarus.org/~martin/PorterStemmer/def.txt, Martin Porter, Program, julio de 1980

[Kraaij 1995] Evaluation of a Dutch Stemming Algorithm http://dis.tpd.tno.nl/mmts/pubs/Keyword/DUTCH-STEMMER.html, Wessel Kraaij, Renée Pohlmann, The New Review of Document and Text Management, 1995

[Bartunov 2003] Gendict - generate dictionary templates for tsearch2 module http://www.sai.msu.su/~megera/oddmuse/index.cgi/Gendict, Oleg Bartunov et. al., julio del 2003

[Chobot 2004] A Walkthrough for Making A Custom Dictionaries for tsearch2 http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/docs/custom-dict.html, Ben Chobot, marzo del 2004