# Desarrollo de Mt77

Vladimir Támara Patiño
vtamara@pasosdeJesus.org

6 de enero de 2010

**Resumen**

Describimos detalles del desarrollo de la versión 1.0 del búscador Mt77. La estructura del documento toma ideas de [7]. Esta información se libera al dominio público y se dedica a Dios por su Enseñanza.

## 1. Proceso de Ingeniería de Software

Se ha planteado en el marco del proyecto SINCODH financiado por la agencia Diakonia, teniendo en cuenta lineamientos de programación extrema (pequeñas entregas, diseño simple, probar, refactorizar, ver [13] y [18]). La implementación de la versión 1.0 se ha desarrollo iterativamente con 5 prototipos (cada iteración con refinamiento de requerimientos, de diseño, de implementación y pruebas).

El quinto prototipo corresponde a la versión 1.0a1 (alfa 1), el cual invitamos a auditar publicamente.

En `https://Mt77.pasosdeJesus.org` se mantiene el repositorio CVS y se realizan pruebas.

## 2. Diseño

### 2.1. Ambiente

Debe funcionar en un computador con características modestas al momento de este escrito. Se sugiere uno con procesador AMD de 64 bits, 1GB en RAM y una partición de al menos 20GB para los documentos e índices. Como sistema operativo se sugiere la distribución adJ de OpenBSD ([20]).

### 2.2. Metodología de Desarrollo

Ver sección 1.

#### 2.2.1. Estudio de Caso: Amberfish

Amberfish permite indexar colecciones de documentos y hacer búsquedas sobre los índices. Su autor previamente había desarrollado `isearch`.

Desventajas: no soporta ISO-8859-1, al parecer sólo ASCII de 7 bits. (ver [17]).

Ventajas: Permite búsquedas booleanas, opera sobre archivos XML de forma genérica.

### 2.2.2. Estudio de Caso: Lucene

Es un buscador muy popular con años de desarrollo en el marco del proyecto Apache. Hay bastantes programas que lo incluyen y otros que son compatibles con sus índices binarios. En cuanto a motores de búsqueda, Lucene es posiblemente el parámetro de comparación estándar.

Ventajas: modularidad, diseño maduro, tamaño de índices y velocidad de indexado.

Desventajas: escrito en Java que no ha sido tan portable en la práctica.

Las fuentes de Lucene constan de varios paquetes y clases bien divididos e ideados. En [16] se describe una extensión a Lucene, y se describen brevemente las 2 operaciones que Lucene realiza: indexar y buscar. El proceso de *indexado* se hace sobre documentos puestos a disposición por el *recolector de datos,* que el *reconocedor* convierte a textos planos. En la fase de *análisis* los datos se dividen de acuerdo a delimitadores predefinidos y se realizan algunas operaciones sobre estos –por ejemplo convertir a minúsculas, retirar algunas palabras, convertir a forma raíz. El proceso de *búsqueda* primero *reconoce* la consulta del usuario para extraer sus unidades y operandos, estos pasan al mismo *analizador* del indexado, después el índice se recorre en busca de coincidencias y se retorna una lista de hallazgos ordenados. El *procesador de consultas difuso* define los criterios de coincidencia y el puntaje de cada hallazgo.

## 2.3. Infraestructura Tecnológica

### 2.3.1. Herramientas de Desarrollo y Estándares

El código fuente seguirá estándares de indentación (ver [4]) que se aseguran con `make indenta`, e incluirá comentarios con estructura para generar documentación del mismo (ver [22]) con `make doctec`. Además de estar en un lenguaje multiplataforma, se empleará bibliotecas de funciones estándar y el código buscará ser portables. Los programas o scripts que dependan del sistema operativo en sus fuentes tendrán la sección dependiente aislada en funciones que podrán remplazarse fácilmente de una plataforma a otra.

Las librerías desarrolladas deben contar con pruebas de unidad (ver [8]) que se ejecutan con `make unidad`. Las pruebas de unidad deben cubrir por completo las librerías (ver [11]), como debe verificar `make cobertura`. Los programas deben contar con pruebas de regresión, que se ejecutan con `make regr`. Para optimizar recursos se realizan análisis de desempeño (profiling) con `make desempeno`.

No nos acogeremos a un formato de índices, pues aún cuando diversas herramientas emplean el formato en disco de Lucene (e.g. Zend Framework, ver [23]), por lo menos Ferret –un descendiente de ese buscador, ver [5]– prefirió cambiar el formato para mejorar la velocidad, y buscadores como Google en lugar de un formato emplea todo un sistema de archivos nuevo (ver [9]).

Se ha elegido C++ por eficiencia y posibilidad de organizar diseño y fuentes, por ser multiplataforma, de amplia difusión y contar con implementaciones de fuentes abiertas desde hace tiempo. Además cuenta con librerías de estructuras de datos bien diseñadas (STL y Boost).

Se descartó Java, porque sólo desde 2007 su licencia ha permitido portabilidad y aunque el popular buscador Lucene está implementando en ese lenguaje, hemos comprobado que aún con compiladores JIT, pierde eficiencia al ejecutarse en una máquina virtual —la cual debe implementarse sobre la máquina real.

### 2.3.2. Estructuras de datos

El buscador requiere un índice invertido que a cada palabra de los textos indexados le asocie las posiciones precisas donde aparece, es decir a cada palabra le asocie una lista

$$\{(d_1, \{p_{1,1}, \ldots, p_{1,n_1}\}), (d_2, \{p_{2,1}, \ldots p_{2,n_2}\}) \ldots (d_t, \{p_{t,1}, \ldots, p_{t,n_t}\})\}$$

donde cada una de las $t$ parejas representa ocurrencias de la palabra. $d_i$ es índice del i-ésimo documento donde aparece y la lista de posiciones donde aparece en ese documento es $(\{p_{i,1}, \ldots, p_{i,n_i}\})$. Entre las estructuras de datos para implementarlo que [12] recomienda están tablas de hashing y un tipo no convencional de tries. De una experiencia anterior con tries convencionales (ver [19]) encontramos que desperdician RAM en palabras únicas (pues cada letra ocupa un nodo) y que termina siendo más crítico el formato en disco.

El formato en disco es clave para lograr balance entre uso de memoria RAM y uso de disco tanto en indexado como en búsquedas. Una búsqueda en RAM es muy veloz, pero cargar una estructura de datos grande de disco a RAM es demasiado costoso en tiempo. La estrategia de Lucene (ver [10]) es mantener varios índices en disco con facilidad para mezclarlos también en disco, las búsquedas las realiza en disco para lo cual emplea listas ordenadas, resultando una complejidad temporal que está en $O(\log n)$ donde $n$ es cantidad de elementos en el índice.

Dado que la complejidad temporal de la búsqueda en tries está en $O(L)$ donde $L$ es longitud de la palabra buscada, preferimos esa estructura. Para disminuir la cantidad de nodos experimentamos con tries especiales (los llamamos TrieS) que en cada nodo mantienen una cadena (en lugar de un caracter). Así un TrieS que únicamente representa la palabra `AMOR` como primera palabra de un primer documento, sólo requerirá un nodo con una sóla posición (1,{1}). Como ejemplo el TrieS del texto `LA VERDAD SI NOS LIBERARÁ SI` cuyas posiciones se presentan en el cuadro 1 se presenta como árbol en las figuras 1 y 2.

| **Posición.** | **Palabra** |
|---|---|
| 1 | LA |
| 3 | VERDAD |
| 10 | SI |
| 13 | NOS |
| 17 | LIBERARÁ, |
| 26 | SI |

Cuadro 1: Posiciones de un texto de ejemplo

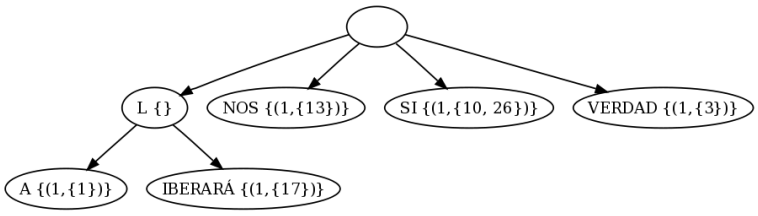



Figura 1: TrieS

### 2.3.3. Formato

Experimentamos un formato eficiente para almacenar el trie especial en disco de forma que la búsqueda aún en disco se mantenga en $O(L)$ y que la mezcla de varios TrieS pueda hacerse eficientemente. Así podríamos adoptar la estrategia de Lucene pero mejorando su velocidad de búsqueda.

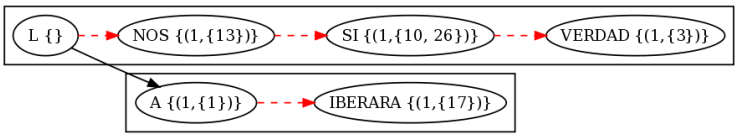



Figura 2: Árbol n-ario con apuntador a hijo menor y a hermano mayor

Dado que el diseño planteado no requiere modificación de TrieS una vez en disco, la propuesta inicial es imitar la organización de un árbol n-ario en RAM cuando se dota a cada nodo de apuntador a hermano mayor e hijo menor (ver figura 2 donde los apuntadores a hermanos son líneas punteadas), pero cambiando los apuntadores por desplazamientos en el archivo. Los nodos se organizarán como en un recorrido en preórden.

La estrategia para mezclas en disco es recorrer simultáneamente los dos TrieS como por grupo de hermanos. Se analiza cada grupo de hermanos y se escribe el grupo de hermanos que resulte posible dejando una cola de pendientes que se retoma tras completar el grupo de hermanos (primero hermanos después hijos con sobrinos).

**Rationale**

Un índice en el formato propuesto no es fácil de modificar después de escrito. Justamente la innovación de los TrieS de nodos con cadenas en lugar de caracteres es la fuente del problema. Empleando la analogía de apuntadores como desplazamientos en el archivo sería más sencillo modificar tries convencionales en disco.

Otra opción para facilitar actualización manteniendo los TrieS es permitir nodos con cadenas vacías que signifiquen continuación del nodo papá, así como cadenas de nodo terminadas en '\0' o algún caracter que indique su fin pudiendo dejar espacio localizado sobrante. Con esto podrían eliminarse y agregarse palabras, pero se pierde la unicidad de la representación en disco y a menos que se optimizara/reorganizara con relativa frecuencia, podría perderse la eficiencia en las búsquedas.

### 2.3.4. Estrategia de Desarrollo

Al tener un mezclador de índices en disco, el proceso de indexado de documentos locales se facilita indexando en RAM algunos documentos y empleando un mezclador que opere en disco para mezclar el nuevo índice con uno completo.

Para disminuir problemas de bloqueos entre búsquedas que se realicen simultáneamente con indexados, los índices no se sobreescriben (excepto cuando esté terminado un procesamiento).

Con respecto a la información por mantener junto con cada palabra consideramos mejor mantener posición dentro del documento donde aparece ('índice posicional' según [14]), para facilitar la búsqueda de varias palabras consecutivas —en la figura 1 son los conjuntos de parejas junto a cada palabra.

Para hacer búsquedas sobre campos se puede indexar el nombre del campo seguido de su valor e.g `DEPARTAMENTO:ANTIOQUIA`. Análogamente puede indexarse el tipo (`TIPO:VIDEO`), el sitio de origen (`SITIO:WWW.TANGLE.COM`) y otros metadatos de cada documento.

El diseño propuesto se ha implementado progresivamente con cada prototipo como se presenta en el cuadro 2.

| Pr. | F. Indice | Indexado en | F. Doc. | Normalización | Consulta |
|---|---|---|---|---|---|
| 1 | Texto | RAM | Texto | Mayúsculas<br>Sólo letras | Una palabra |
| 2 | Texto | RAM y Disco | Relato | | Varias palabras |
| 3 | Binario | RAM y Disco<br>en RAM 1 doc | ODF | Sin conectores español | Metainformación |
| 4 | Comprimido | HTML | | | |
| 5 | Más Comprimido | RAM y Disco<br>lotes de 50MB | PDF | Algunos dígitos y puntos | Cadenas |

Cuadro 2: Progreso en prototipos

# 3. Implementación

Junto con el diseño se ha implementado en lenguaje C++ con STL: indexador, de motor de búsqueda y mezclador; emplean TrieS como estructura de datos almacenada en disco en un formato comprimido.

## 3.1. Formato

Cada nodo del TrieS se almacena como: cadena , "apuntador" a hermano mayor, "apuntador" a hijo menor y listado de posiciones de la forma

$$\{(d_1, \{p_{1,1}, \Delta_{1,2} \ldots \Delta_{1,n_1}\}), (d_2, \{p_{2,1} \ldots \Delta_{2,n_2}\}) \ldots (d_k, \{p_{t,1} \ldots \Delta_{t,n_t}\})\}$$

Note que se ordenan ascendentemente por número de documento (i.e $1 \leq i < j \leq t \rightarrow d_i < d_j$) y de posición (i.e $1 \leq i \leq t$ y $1 \leq j < k \leq n_i$ implican $p_{i,j} < p_{i,k}$), para almacenar diferencias entre una posición y otra (i.e si $1 \leq i \leq n$ y $1 < j \leq n_i$ entonces $\Delta_{i,j} = p_{i,j} - p_{i,j-1}$) y así obtener una secuencia de números más pequeños y facilitar diversos esquemas de compresión.

Suponiendo que los números se representarán en notación decimal, y que se usarán bastantes caracteres para separar, el TrieS en disco correspondiente al de la figura 1 se presenta en el cuadro 3.

| Posición | 1 2 3 4 5 6<br>12345678901234567890123456789012345678901234567890123456 |
|---|---|
| 1<br>67 | `L{9;67;}NOS{26;0;(1,{13})SI{45;0;(1,{10,27})}VERDAD{65;0;(1,{3})}`<br>`A{82;0;(1,{1})}IBERARA{105;0;(1,{17})}` |

Cuadro 3: Aproximación a la representación en disco

En disco, no se emplean tantos caracteres para separar, sólo se separa la cadena de los apuntadores con '`{`', una pareja de otra con '`(`', el final de un listado de posiciones con '`}`' y el fin de una secuencia de hermanos se denota con un cambio de línea.

Por sugerencia de [12] se comprimen tanto los números de documentos, como las posiciones y deltas con códigos Γ de Elías (ver [15]), de forma que los números menores a 65535 ocupan entre uno y cuatro bytes[1]. Se podrían comprimir más las secuencias de deltas si no se alinea a bytes, aunque eventualmente se afectaría más la velocidad de indexado.

Cada "apuntador" es un desplazamiento dentro del archivo representado como un número en base 128 de 5 "dígitos" o bytes (un poco más legible que la representación binaria de 4 bytes).

A continuación se presenta el índice que correspondería al TrieS de la figura 1 como lo genera `hexdump -C` (cada fila presenta 16 bytes, la columna de la izquierda es posición en hexadecimal, la siguiente es columna son los 8 primeros bytes en hexadecimal, la siguiente son los 8 siguientes bytes en hexadecimal y la cuarta columna es la representación en ASCII):

```
00000000  4c 7b 30 30 30 30 3d 30  30 30 30 75 7d 4e 4f 53  |L{0000=0000u}NOS|
00000010  7b 30 30 30 30 4e 30 30  30 30 30 00 ea 7d 53 49  |{0000N00000.ê}SI|
00000020  7b 30 30 30 30 60 30 30  30 30 30 00 e4 f0 00 7d  |{0000‘00000.äð.}|
00000030  56 45 52 44 41 44 7b 30  30 30 30 74 30 30 30 30  |VERDAD{0000t0000|
00000040  30 00 a0 7d 0a 41 7b 30  30 30 30 84 30 30 30 30  |0.ă}.A{0000.0000|
00000050  30 00 00 7d 49 42 45 52  41 52 c1 7b 30 30 30 30  |0..}IBERARÁ{0000|
00000060  9a 30 30 30 30 30 00 f0  80 7d 0a                 |.00000.ð.}.|
0000006b
```

La relación de documentos del índice estaría en otro archivo y suponiendo que el archivo indexado se llame `verdad.txt` y que estuviera en el nodo de desarrollo sería algo como:

```
https://pasosdeJesus/verdad.txt 77e3dc9e5afc3ccf1deef0d4730eae56ff57c1320c6887432a3da1cc92795e9f 2009-09-30
```

Note en el índice:

- Después de cada grupo de nodos hermanos se agrega un cambio de línea.
- El número 0 en la codificación en base 128 empleada es 00000.
- Se espera que comience con un identificador de la versión del formato y que esté almacenado en un archivo con extensión `.indice`.

Con respecto a la relación de documentos:

- A continuación de cada URL se pone un condensado (del inglés *digest*), empleando el algoritmo seguro SHA256 y a continuación la fecha de modificación del archivo.
- Se espera que comience con un identificador de la versión del formato y que esté almacenado en un archivo con extensión `.relacion`.

La actual implementación emplea un normalizador que excluye palabra comunes del español (como artículos y preposiciones), que preserva palabras de máximo 32 caracteres y puede reconocer textos planos, documentos XML, HTML, documentos ODF y PDF Las etiquetas y datos de documentos XML las almacena como campos con valores y de todo documento (incluso vacíos) almacena como campos la siguiente metainformación:

- `TITULO` con las palabras del nombre del archivo.
- `SITIO` con el nodo de procedencia.
- `TIPO` con el tipo del archivo, que en este momento puede ser `TEXTO`, `XML`, `XRLAT` o `ODT` para documentos en OpenOffice Document Format.

[1]Si $k$ es entero no negativo, el entero $2^{4k}$ requiere $k+1$ bytes en codificacisn Γ de Elías.

Por esto mismo, el ejemplo anterior en realidad genera el siguiente índice (`verdad.indice`):

```
00000000  4d 74 37 37 3a ed 6e 64  69 63 65 50 35 0a 4c 49  |Mt77:índiceP5.LI|
00000010  42 45 52 41 52 41 7b 30  30 30 30 55 30 30 30 30  |BERARA{0000U0000|
00000020  30 00 f0 80 7d 4e 4f 53  7b 30 30 30 30 66 30 30  |0.ð.}NOS{0000f00|
00000030  30 30 30 00 ea 7d 53 49  7b 30 30 30 30 78 30 30  |000.ê}SI{0000x00|
00000040  30 30 9b 00 e4 f0 80 7d  54 49 7b 30 30 30 30 86  |00..äð.}TI{0000.|
00000050  30 30 30 31 3a 7d 56 45  52 44 41 44 7b 30 30 30  |0001:}VERDAD{000|
00000060  30 9a 30 30 30 31 9f 00  a0 7d 0a 54 49 4f 3a 50  |0.0001..ă}.TIO:P|
00000070  41 53 4f 53 44 45 4a 45  53 55 53 7b 30 30 30 31  |ASOSDEJESUS{0001|
00000080  39 30 30 30 30 30 00 ea  7d 0a 50 4f 3a 44 4f 43  |900000.ê}.PO:DOC|
00000090  55 4d 45 4e 54 4f 7b 30  30 30 31 54 30 30 30 30  |UMENTO{0001T0000|
000000a0  30 00 e4 7d 54 55 4c 4f  3a 7b 30 30 30 31 65 30  |0.ä}TULO:{0001e0|
000000b0  30 30 31 66 7d 0a 54 58  54 7b 30 30 30 31 77 30  |001f}.TXT{0001w0|
000000c0  30 30 30 30 00 c0 7d 56  45 52 44 41 44 7b 30 30  |0000.À}VERDAD{00|
000000d0  30 31 8b 30 30 30 31 8c  00 d8 7d 0a 2e 54 58 54  |01.0001..Ø}..TXT|
000000e0  7b 30 30 30 31 9e 30 30  30 30 30 00 e6 7d 0a 2e  |{0001.00000.æ}..|
000000f0  54 58 54 7b 30 30 30 32  31 30 30 30 30 30 00 00  |TXT{0002100000..|
00000100  7d 0a                                             |}.|
00000102
```

El cual excluye el artículo LA y agrega la metainformación: `SITIO:PASOSDEJESUS`, `TITULO:VERDAD`, `TITULO:TXT` y `TIPO:TEXTO`.

La relación de documentos (`verdad.relacion`) sería:

## 3.2. Otras características del código fuente

Los errores de ejecución fatales generan excepciones con cadenas explicativas. Las advertencias durante ejecución se envían al flujo `cerr`. Para depurar se usa en las fuentes el flujo `clog`.

## 3.3. Indexador

El indexador se ha refinado para que opere en RAM en grupos de documentos cuyos tamaños no pasen los 50MB. Los índices de cada grupo se van mezclando con un índice en disco, hasta completar el procesamiento de todos los documentos. Par leer los diversos formatos emplea varias estrategias:

- ODT: Descomprime y extrae el contenido XML que transforma a texto plano con una hoja de estilo XSLT y la librería libxslt.
- XML y relatos: lo lee con la librería expat. Todas las entidades que encuentra las agrega como etiquetas cuyo valor es el dato que tengan.
- HTML: Un reconocedor que ignora el formato.
- PDF: Emplea la herramienta pdftotxt incluida en el paquete xpdf-utils para convertir a texto plano.

## 3.4. Mezclador

Entre diversas funcionalidades permite mezclar dos índices que están en disco para producir un tercer índice en disco. Por ejemplo en sistemas distribuidos esto es útil para producir un índice general a partir de los provenientes de diversos nodos.

## 3.5. Buscador

Ordena resultados por fecha y frecuencia de ocurrencia y retorna una parte de los mismos en JSON para que sean interpretados con fácilidad. La parte que retorna depende de parámetros que se especifican en la línea de comandos para facilitar presentación por páginas. La búsqueda de cadenas opera buscando las palabras de la cadena que resulten tras normalizar y que estén consecutivas (máximo a 5 espacios de difencia). La frecuencia retornada corresponde a la suma de frecuencias de las palabras o cadenas individuales.

## 3.6. Comparación

Para comparar tiempos de respuesta de acuerdo al estado del arte en buscadores, hemos preparado 3 corpus de prueba y comparado con Amberfish y Lucene.

### 3.6.1. Ambiente de pruebas

El ambiente de pruebas en un computador con AMD Athlon de 1.8GHz, con 1GB en RAM, 120GB en disco duro y OpenBSD-current. Realizamos pruebas con 3 corpus de pruebas que contienen sólo documentos de dominio público.

| Criterio | Corpus 1 | Corpus 2 | Corpus 3 |
|---|---|---|---|
| Número de documentos | 7 | 40 | 96 |
| Espacio en disco | 1.5MB | 22.6MB | 160MB |

Cuadro 4: Proporciones de los corpus de prueba

En el caso de Amberfish se compiló la versión 1.6.4 con las opciones por defecto y después de cada indexación se linealizó el índice (el tiempo presentado suma indexado más linealizado y el tamaño es el tamaño después de linealizar).

Con respecto a Lucene se compiló la versión 2.4.0 con un JDK-1.7, y se utilizó sin cambios el programa de demostración incluido con las fuentes.

| Lucene 2.4 | Amberfish | Prot. 1 | Prot. 2 | Prot. 3 | Prot. 4 | Prot. 5 |
|---|---|---|---|---|---|---|
| 30s | 7.4s | - | 35s | 24s | 25s | |

Cuadro 5: Tiempo de compilación

En las siguiente subsecciones se comparan los resultados empleando como parámetros de comparación: Tiempo para indexar corpus, Espacio en disco del índice producido, Tiempo para buscar una palabra y Precisión de los resultados de la búsqueda.

### 3.6.2. Corpus de prueba 1

Se trata de Evangelios en español ([2]), en inglés y en griego. En total 7 documentos de 1.5MB.

| Criterio | Lucene 2.4 | Amberfish | P1 | P2 | P3 | P4 | P5 |
|---|---|---|---|---|---|---|---|
| Indexado | 2.2s | 2.4s | 12s | 13.5s | 22.09s | 15.42s | 5.69s |
| Tamaño del índice | 192K | 2.3MB | 5.5MB | 4.4MB | 3.3MB | 1.7MB | 1.2MB |
| Buscar "Jesus" | 0.5s | 0.2s | <0.1s | <0.1s | <0.1s | <0.1s | <0.1s |
| Resultados precisos | SI | SI | SI | SI | SI | SI | SI |

Cuadro 6: Corpus 1

### 3.6.3. Corpus de prueba 2

Incluye el primero además Biblias en español y literatura en español del proyecto Gutenberg ([1]). Son 40 documentos de 22.6MB.

| Criterio | Lucene 2.4 | Amberfish | P2 | P3 | P4 | P5 |
|---|---|---|---|---|---|---|
| Indexado | 7.3s | 20.2s | 171s | 8m29s | 6m54s | 1m26 |
| Tamaño del índice | 983K | 12.1M | 47.9MB | 34.3MB | 16.2MB | 11.6M |
| Buscar "Jesus" | 0.5s | 0.2s | 0.1s | 0.1s | 0.1s | |
| Resultados precisos | NO* | NO** | SI | SI | SI | SI |

Cuadro 7: Corpus 2

El primer prototipo (que operaba enteramente en RAM) no logró indexar la colección completa por lo que no se incluyó en esta comparación.

Empleando otra herramienta (`grep`) comprobamos que los 26 resultados reportados por los prototipos de Mt77 son justamente los correctos al considerar palabras completas, ignorar diferencias entre mayúsculas y minúsculas y entre vocales con tilde y sin tilde (en 18 documentos aparece sólo con tilde, en 3 aparece sólo sin tilde y en 5 aparece tanto con tilde como sin tilde).

* Los 10 resultados de Amberfish no fueron precisos porque si bien incluyeron los 8 textos donde estaba la palabra sin tilde, agregó otros 2 donde no estaba. En esos 2 textos aparecía la palabra con tilde pero Amberfish no reportó los otros 16 donde también aparecía sólo con tilde. Con Amberfish no nos fue posible realizar búsqueda de palabras con tilde.

** Por su parte Lucene de los 5 resultados que reportó, incluye 1 donde no está la palabra buscada. Es decir reportó sólo 4 de los 8 textos donde estaba. 3 de los 4 documentos que Lucene no reportó tienen la palabra seguida de un signo de puntuación, lo cual nos hace pensar que el analizador por defecto de Lucene (usado por el buscador de demostración) no soporta bien signos de puntuación y hace diferencia entre vocales tildadas y sin tilde (que consideramos deberían ser intercambiables para buscadores sobre textos en español). A diferencia de Amberfish, Lucene si permite buscar palabras con tilde, pero al buscar "Jesús" tampoco dio los 23 documentos esperados (dio 16).

### 3.6.4. Corpus de prueba 3

Incluye el segundo corpus junto con libros históricos de los Padres de la Iglesia de los siglos I a IV traducidos al inglés ([21]). Son 98 documentos de 160MB.

El segundo prototipo no logró indexar la colección completa por agotamiento de memoria RAM como se predecía por lo que no se incluyó en esta comparación.

| Criterio | Lucene 2.4 | Amberfish | P3 | P4 | P5 |
|---|---|---|---|---|---|
| Indexado | 14.6s | 3m48s | 169m | 100m | 13m58 |
| Tamaño del índice | 2.4M | 12.1M | 301M | 117M | 77.7M |
| Buscar "Jesus" | 0.5s | 0.2s | 0.1s | 0.1s | <0.1s |
| Resultados precisos | NO | NO | SI | SI | SI |

Cuadro 8: Corpus 3

La palabra completa Jesus (sin tilde e ignorando mayúsculas y minúsculas) se encontró con otras herramientas en 63 de los 96 documentos, la palabra completa Jesús (con tilde e ignorando mayúsculas y minúsculas) se encontró en 24 documentos. La palabra aparece tanto con tilde como sin tilde en 6 documentos y por tanto está con tilde o sin tilde en 81 documentos que son justamente los reportados por Mt77.

## 3.7. Límites

Los estimamos a partir del ambiente sugerido (ver 2.1). Y tras las comparaciones, suponiendo índices comprimidos al 60 % respecto a los documentos i.e $P = 0{,}6T$ .

- Sistema operativo: Con FFS2 sistemas de archivos y particiones máximo de 2TB i.e $2^{41}$ bytes.
- Formato: Máxima cantidad de documentos que puede indexar: $2^{31}$ Máximo tamaño de un documento: $2^{31}$ = 2GB. Máximo tamaño de un índice: $2^{31}$ = 2GB, lo cual a su vez limita el máximo tamaño de un documento indexable a $\frac{2^{31}}{2}$ = 1GB. El método de compresión empleado mejora al indexar documentos de menos de 64K.
- Indexador de un documento: Opera en RAM, en experimentos ha logrado indexar documentos de hasta 60MB. Aunque limitado, lo consideramos suficiente porque no es típico tener documentos de tales tamaños y el mezclador permite manejar colecciones mucho mayores.
- Mezclador: En RAM sólo requiere conjunto de posiciones de un nodo, las cuales según experimentos con 1GB en RAM alcanzan para indexar una misma palabra aproximadamente en 10 millones de posiciones. La limitación en su operación la impone el espacio disponible en disco. Para mezclar 2 índices de tamaños $P$ y $Q$ requiere espacio $P + Q$ temporales, es decir si todas las mezclas se hacen en la misma partición $2P + 2Q$. O si la colección de documentos de una organización es de tamaño $T$, el índice será de tamaño $0{,}6T$ y la construcción del mismo requerirá un espacio similar es decir $2{,}2T$. i.e. máximo 9GB en una partición de 20GB.

En resumen y de forma más práctica en el ambiente propuesto con una partición de 20GB dedicada a documentos e índices, estimamos que pueden indexarse máximo 600.000 documentos, cada uno de los cuales tenga menos de 60MB, con un tamaño total de documentos máximo de 9GB (pueden publicarse vídeos y audios de tamaños arbitrarios pues su contenido no será indexado –aunque restarán espacio en disco que es requerido en los procesos de indexación).

# 4. Control de versiones

Ver cuadro 9.

| | |
|---|---|
| P5 | Formato de índices comprimido empleando códigos Γ de Elías. Nueva mezcla de un índice en RAM con uno en disco, que resulta más rápida que mezclar dos índices en disco y agiliza operación de agregar un documento a un índice existente (opción `agregadoc` de `operaindice`), así como el indexado de colecciones. Análisis de desempeño (profiling) de indexado y mezcla de indice en disco con índice en RAM. El indexador puede leer documentos PDF (ISO 32000-1, ver [3]) y ahora opera sobre colecciones, indexando por lotes en RAM de hasta 50MB y mezclando con índice temporal en disco hasta completar todos los lotes. El buscador puede buscar cadenas, encontrando las palabras normalizadas que aparezcan consecutivas (de 1 a 5 espacios de distancia). El normalizador ahora incluye números y puntos en algunos casos. Se indexa el título como palabra del documento. |
| P4 | Indexa documentos HTML. Formato de índices cambiado para disminuir tamaño e incluir fecha en relación de documentos. Buscador ordena resultados según fecha (primero los más recientes) y retorna resultados en JSON posibilitando paginación. |
| P3 | División de índices en 2 archivos para agilizar actualización distribuida. Esquema de indexación local cambiado para indexar de a un documento en RAM y mezclar en disco –que aumenta límites de indexación para que dependan principalmente del espacio en disco y no en RAM. Formato cambiado para lograr más velocidad en búsquedas y disminuir tamaño de índices. Permite nombres con espacios y caracteres de español. Permite búsquedas con metainformación sitio, tipo y titulo. Indexa documentos XML y ODF (.odt). Comparaciones con Lucene y Amberfish. |
| P2 | Mezclador opera enteramente en disco, aunque el conjunto de posiciones lo mantiene en RAM (i.e la cantidad de documentos retornados está limitada por la cantidad de RAM –se han probado máximo 10'000.000 con 1GB en RAM). El indexador incluye URL origen del documento. Permite búsqueda de varias palabras. Normalizador elimina algunas palabras de español tan comunes que no añaden información a una consulta (artículos, preposiciones, conjunciones). |
| P1 | Indexador y mezclador operan enteramente en RAM (i.e a lo sumo indexan y mezclan en proporción a la RAM que tenga el servidor[2]), en cuanto a formato de entrada sólo soportan textos planos, normalizan convirtiendo a mayúsculas y eliminando números, tildes y signos de puntuación. |

Cuadro 9: Control de versiones

# 5. Conclusiones

- Los tamaños de índices de Lucene y de Amberfish son mejores que los de Mt77, aunque con cada prototipo de Mt77 el tamaño promedio de los índices se ha reducido en las proporciones aproximadas presentadas en la tabla 10.

  Consideramos que es buena la tasa de compresión alcanzada que estimamos mejorará entre más

| | **P1** | **P2** | **P3** | **P4** | **P5** |
|---|---|---|---|---|---|
| Tamaño del índice / Tamaño del documento | 4 | 3 | 2 | 1 | 0.6 |

Cuadro 10: Proporción tamaño del índice sobre tamaño del documento

pequeños sean los documentos por indexar (se sugiere menos de 64KB cada uno). Preveemos más posibilidades de comprimir aplicando más intensamente las estrategias hasta ahora empleadas (por ejemplo empleando otra representación para desplazamientos y comprimiendo la secuencia completa de posiciones y deltas de un documento y no una a una) o investigando otros esquemas de compresión.

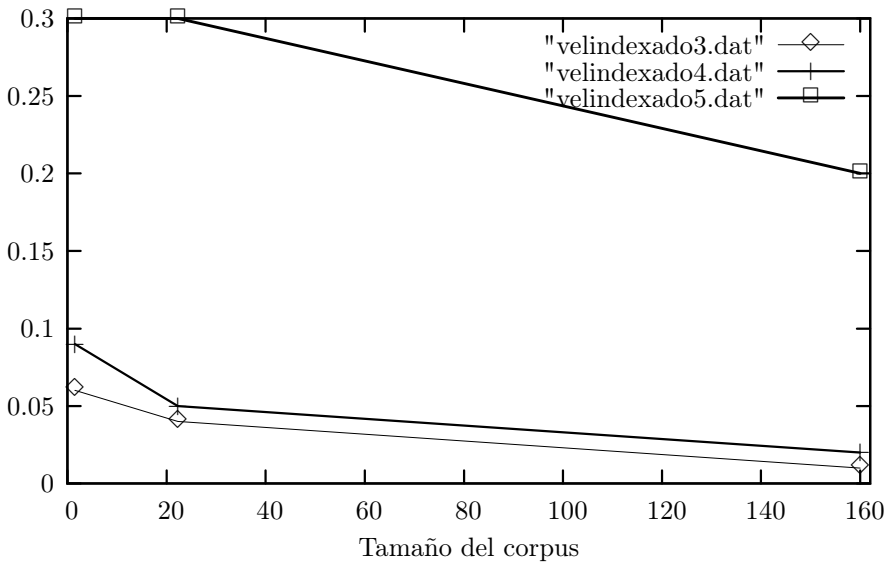
- Los tiempos de indexado de Lucene y Amberfish son mucho mejores que los de Mt77, aunque la poca precisión en la búsqueda de esos programas no permite compararlos justamente con Mt77. El prototipo cinco ha logrado el mejor tiempo de indexado de los cinco desarrollados aumentando más de 280 % la velocidad de indexado con respecto al prototipo 4, sin embargo esta velocidad decrece para colecciones de más de 50MB y a medida que aumenta el tamaño de la colección como muestra la tabla 11 y la gráfica 5. Aunque la velocidad del indexador no era un requerimiento crítico, y permite cumplir los demás requerimientos esperados sin afectar directamente la interacción del usuario final (i.e la del buscador), es importante continuar el análisis de desempeño e investigar optimizaciones más profundas para el indexado y la mezcla de índices.

| Corpus | Tamaño | **Lucene** | **Amberfish** | **P3** | **P4** | **P5** |
|---|---|---|---|---|---|---|
| 1 | 1.5MB | 0.68 MB/s | 0.62 MB/s | 0.06 MB/s | 0.09 MB/s | 0.26 MB/s |
| 2 | 22.3MB | 3.09 MB/s | 1.11 MB/s | 0.04 MB/s | 0.05 MB/s | 0.25 MB/s |
| 3 | 160MB | 10.95 MB/s | 0.70 MB/s | 0.01 MB/s | 0.02 MB/s | 0.19MB/s |

Cuadro 11: Velocidad de indexado. MB por segundo

- Mt77 tuvo los mejores tiempos de búsqueda con respecto a Lucene y Amberfish, dio los resultados más precisos y por defecto opera especialmente bien con documentos y búsquedas en español.
- Deben emplearse otros buscadores que sean precisos con los corpus de pruebas, de forma que resulte justa la comparación de tamaño de índice y tiempo de indexado. También debe ampliarse el ambiente de pruebas empleando otro corpus por lo menos de 1GB.
- Encontramos una balanza o canje entre el tiempo de búsqueda y el tiempo de indexado[3] y hemos ubicado Mt77 en favor del tiempo de búsqueda por haber sido este uno de los requerimientos explícitos del proyecto —requerimiento que consideramos ampliamente satisfecho.

[3]*'La eficiencia de un algoritmo es típicamente medida de acuerdo al uso de algún recurso computacional relevante. El recurso más estudiado es tiempo computacional, pero otro recurso importante es espacio computacional. Tipicamente, los problemas de diseño algoritmico se enfocan en la meta de minimizar uno de estos recursos. ... De hecho para muchos problemas computacionales fundamentales como ordenamiento, multiplicación matricial y conectividad de grafos dirigidos, las metas de minimizar tiempo y espacio parecen estar en conflicito; los algoritmos más eficientes en tiempo conocidos requieren pesados recursos de memoria y a medida que decrece la cantidad de memoria usada, la cantidad de tiempo requerida para resolver el problema aparentemente aumenta significativamente. Este canje aparente entre tiempo y espacio ha motivado un gran cuerpo de investigación en teoría de la complejidad' [6]*




# Anexo A. Extender soporte a otros Formatos

Para soportar otros formatos de documentos, en el momento de este escrito debe:

1. Incluir un caso de detección del formato en la función `determinaFormato` del archivo `Operaciones.cpp`.
2. Incluir en el proceso de compilación un nuevo archivo donde esté la función de conversión (i.e en el archivo `Makefile` ver ejemplo `leeXML.cpp`), esta función típicamente recibirá el nombre del archivo, el número de documento al que corresponde y el árbol `NodoTrieS` donde deben añadirse las palabras. Lo que debe hacer es abrir el archivo, leer las palabras indexables y su posición y agregarlas al árbol que recibe. Para añadir las palabras una a una puede emplear la función `inserta` (ver ejemplo en función `leeXML`) o bien puede convertir todo el documento a texto plano y emplear la función `leeTexto` (ver ejemplo en función `leeODT`).
3. Incluir un caso de conversión en la función `indexa` de `Operaciones.cpp` que llame la función del punto anterior cada vez que se lea un archivo en el nuevo formato.

# Referencias

[1] Project gutenberg. http://www.gutenberg.org/browse/languages/es.

[2] Traducción de los evangelios de dominio público y abierta a mejoras. http://de.geocities.com/nuestroamigojesus/bdp, 2003.

[3] Pdf format becomes iso standard. http://www.iso.org/iso/pressrelease.htm?refid=Ref1141, 2008.

[4] Artistic style: A free, fast and small automatic formatter for c, c++, c#, and java source code. http://astyle.sourceforge.net/http://astyle.sourceforge.net/, 2009.

[5] David Balmain. Ferret. http://ferret.davebalmain.com/api/classes/Ferret.html, 2005.

[6] Paul Beame, Michael Saks, Xiadong Sun, and Erik Vee. Time-space tradeoff lower bounds for randomized computation of decision problems. *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science*, 2002.

[7] Industry Canada: Application Services Branch. High level design and architecture document. http://strategis.ic.gc.ca/epic/site/infodev.nsf/en/dv00236e.html.

[8] Eric Sommerlade et al. Cppunit documentation. http://cppunit.sourceforge.net/doc/lastest/index.htmlhttp: 2008.

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[10] Brian Goetz. The lucene search engine: Powerful, flexible, and free. http://www.javaworld.com/javaworld/jw-09-2000/jw-0915-lucene.html?page=2, 2000.

[11] Brian J. GoughBrian J. Gough. *An Introduction to GCC - for the GNU compilers gcc and g++*. http://www.network-theory.co.uk/docs/gccintro/gccintro_81.html, 2005.

[12] S. Heinz and J. Zobel. Efficient single-pass index construction for text databases. *Journal of the American Society for Information Science and Technology, v54 n8 p713-29*, 2003.

[13] Ron Jeffries. What is extreme programming? http://www.xprogramming.com/xpmag/whatisxp.htm, 2001.

[14] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. Introduction to information retrieval. http://nlp.stanford.edu/IR-book/html/htmledition/irbook.html, 2008.

[15] Alistair Moffat and Andrew Turpin. *Compression and coding algorithms*. http://books.google.com.co/books?id=86KkcSX5BuoC, 2002.

[16] Khaled Nagi. Bringing information retrieval back to database management systems. *Ubiquitous Computing and Communication Journal: Special Issue of IKE'07 Conference*, 2008.

[17] Oskars Ozols. Amberfish: Unicode support. http://sourceforge.net/forum/forum.php?thread_id=1185121&f 2006.

[18] Vladimir Támara Patiño. Ayudas para el desarrollo de programas. http://structio.sourceforge.net/guias/ayudadesprog/ayudadesprog.html.

[19] Vladimir Támara Patiño. Ganarle a google. experimentos de búsqueda con tries en ram. http://www.pasosdejesus.org/vtamara/estinf2006-1/proy-s/index.html, 2006.

[20] Vladimir Támara Patiño. Aprendiendo de jesús. distribución de openbsd. http://aprendiendo.pasosdeJesus.org, 2008.

[21] Editado por Philip Schaff. Early church fathers. christian classics ethereal library. http://www.ccel.org/fathers.html.

[22] Dimitri van Heesch. Doxygen: Source code documentation generator tool. http://www.stack.nl/ dimitri/doxygen/, 2008.

[23] Zend. Zend framework. http://framework.zend.com, 2008.