UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

COMPARACIÓN ENTRE ÍNDICE INVERTIDO Y WAVELET TREE COMO MÁQUINAS DE BÚSQUEDA

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

SENÉN ANDRÉS GONZÁLEZ CORNEJO

PROFESOR GUÍA:
SR. MAURICIO MARÍN CAIHUAN

MIEMBROS DE LA COMISIÓN:
SR. BENJAMIN BUSTOS CARDENAS
SR. GONZALO NAVARRO

SANTIAGO DE CHILE
SEPTIEMBRE 2009

RESUMEN DE LA MEMORIA
PARA OPTAR AL TITULO DE
INGENIERO CIVIL EN COMPUTACIÓN
POR: SENÉN GONZÁLEZ C.
FECHA:14/09/2009
PROF. GUIA : Sr. JUAN MAURICIO MARIN

"COMPARACIÓN ENTRE ÍNDICE INVERTIDO Y WAVELET TREE COMO MÁQUINAS DE BÚSQUEDA."

Las máquinas de búsqueda para la Web utilizan el índice invertido como estructura de datos que permite acelerar las búsquedas en grandes colecciones de texto. Para lograr tiempos de respuesta por consulta menores al medio segundo, tanto el índice como la colección de texto se particionan en dos grupos de máquinas distintos. Cada consulta se envía al primer grupo, las cuales responden con los documentos más relevantes para esa consulta. Desde estos resultados se seleccionan los documentos más relevantes en forma global. Dichos documentos se envían al segundo grupo de máquinas las cuales extraen de la colección de texto el resumen (o snippet) asociado a cada documento en el resultado y construyen la página Web final a ser presentada al usuario como respuesta a su consulta.

En este trabajo de memoria se propone un método alternativo de procesamiento de consultas, el cual ocupa un solo grupo de máquinas para realizar ambas operaciones, es decir, en un solo grupo de máquinas se realiza la determinación de los mejores documentos y la construcción de la página Web de respuesta.

Para esto se recurre al uso de estrategias de texto comprimido auto-indexado y memoria cache diseñada para mantener las listas invertidas de los términos más frecuentes en las consultas. El texto comprimido auto-indexado se utiliza para generar de manera on-line las listas invertidas y para generar el resumen asociado a cada documento en la respuesta a una consulta.

Los resultados experimentales muestran que en el mismo espacio ocupado por el índice invertido estándar es posible ubicar la memoria cache de listas invertidas y el texto comprimido, y alcanzar la misma tasa de respuestas por unidad de tiempo que se logra con el índice invertido. La ventaja está en que en el nuevo esquema no es necesario el uso del segundo grupo de máquinas, y por lo tanto se logra un mejor uso de los recursos de hardware, lo cual es relevante para la operación económica de los grandes centros de datos para máquinas de búsqueda.

Agradecimientos

En primer lugar quisiera agradecer a mi esposa Carmen y a mi hija Jazmín por darme siempre el apoyo necesario para seguir adelante.

También quisiera agradecer a Diego Arroyuelo y Mauricio Silva por ayudarme cuando más lo necesitaba.

Al profesor Mauricio Marín, que me permitió acercarme a este tema. Al profesor Benjamín Bustos por su confianza y apoyo.

Al profesor Gonzalo Navarro por ser parte de la comisión y ayudarme a mejorar aspectos del trabajo realizado.

Un agradecimiento especial a Susana Ladra quien fue de mucha ayuda en la realización de los programas asociados a este trabajo.

Índice general

1. Introducción **6**
- 1.1. Objetivos . . . 7
  - 1.1.1. Objetivo General . . . 7
  - 1.1.2. Objetivos Específicos . . . 7
- 1.2. Contribuciones . . . 7
- 1.3. Descripción de capítulos . . . 8

2. Antecedentes **9**
- 2.1. Recuperación de la información . . . 9
- 2.2. Motores de búsqueda Web . . . 9
- 2.3. Computación paralela sobre memoria distribuida . . . 11
- 2.4. Índices Invertidos . . . 12
  - 2.4.1. Particionado de índices . . . 13
    - 2.4.1.1. Índice particionado por documentos . . . 13
    - 2.4.1.2. Índice particionado por términos . . . 14
    - 2.4.1.3. Ventajas y desventajas de ambos métodos de distribución del índice . . . 14
- 2.5. Compresión de textos . . . 15
  - 2.5.1. Plain Huffman . . . 15
  - 2.5.2. End-Tagged Dense Code (ETDC) . . . 17
  - 2.5.3. Técnica de compresión a ocupar . . . 18
- 2.6. Autoíndices comprimidos . . . 18
  - 2.6.1. Rank y select . . . 19
  - 2.6.2. Wavelet Tree . . . 19
  - 2.6.3. Wavelet Tree con codificación de Huffman a la palabra . . . 20
    - 2.6.3.1. Construcción . . . 20
    - 2.6.3.2. Búsqueda . . . 22

3. Diseño de máquinas de búsqueda **24**
- 3.1. Arquitectura . . . 24
  - 3.1.1. Diseño del ambiente de comparación . . . 25
- 3.2. Diseño de la máquina de búsqueda . . . 25
- 3.3. Diseño de Índices . . . 28
  - 3.3.1. Índice invertido . . . 28

3.3.1.1. Extracción de vocabulario . . . 29
3.3.1.2. Lista de documentos por palabra . . . 29
3.3.1.3. Puntaje para el ranking . . . 29
3.3.2. Wavelet Tree . . . 30
3.3.2.1. Vocabulario . . . 30
3.3.2.2. Construcción del Wavelet Tree . . . 30
3.3.2.3. Lista de documentos por palabra y ranking on-line . . . 30
3.4. Funciones Comunes . . . 30
3.4.1. Intersección de listas de documentos . . . 30
3.5. Funciones de búsqueda . . . 31

4. Implementación **33**
4.1. Construcción de la maquina de búsqueda paralela . . . 33
4.1.1. Detalle de las funciones . . . 34
4.1.1.1. Lector de mensajes . . . 34
4.1.1.2. Separar consulta en términos . . . 35
4.1.1.3. Consultar términos al índice . . . 36
4.1.1.4. Intersecta respuestas . . . 36
4.1.1.5. Ordenar las respuestas según su puntaje . . . 37
4.1.1.6. Envía respuestas . . . 38
4.1.1.7. Lectura de nuevas consultas y envío de éstas . . . 38
4.1.1.8. Sincronización . . . 38
4.2. Construcción del prototipo con índice invertido . . . 38
4.2.1. Construcción . . . 38
4.2.2. Búsqueda . . . 39
4.3. Construcción del prototipo con Wavelet Tree . . . 39
4.3.1. Código . . . 39
4.3.2. Construcción . . . 40
4.3.3. Búsqueda . . . 41

5. Experimentos **44**
5.1. Objetivos de los experimentos . . . 44
5.2. Diseño de experimentos . . . 44
5.2.1. Configuración de los experimentos . . . 44
5.2.1.1. Índices a comparar . . . 44
5.2.1.2. Hardware . . . 46
5.2.1.3. Datos empleados . . . 46
5.2.1.4. Formato de los datos . . . 46
5.2.2. Tamaño del índice con respecto al número de documentos . . . 47
5.2.3. Promedio de consultas que se responden por segundo . . . 47
5.2.3.1. Configuraciones de los experimentos realizados . . . 47
5.3. Resultados . . . 48
5.3.1. Tamaño del índice con respecto al número de documentos . . . 48
5.3.2. Promedio de consultas que se responden por segundo . . . 48

5.3.2.1. Variando el tamaño de la Web . . . 48
5.3.2.2. Variando el número de procesadores . . . 49
5.3.2.3. Variando el número de máquinas y el tamaño de la Web . . . 49
5.4. Discusión . . . 50
5.5. Conclusiones . . . 53

6. Método híbrido **54**
6.1. Diseño . . . 54
6.1.1. Respuestas por segundo . . . 56
6.1.2. Una mejora al método . . . 56
6.2. Experimentación . . . 59
6.2.1. Configuración de sistema . . . 59
6.2.2. Tamaños para distintas configuraciones . . . 59
6.2.3. Consultas por segundo que es capaz de responder . . . 61
6.2.4. Escalabilidad . . . 61
6.2.4.1. Variando la cantidad de documentos que componen la Web 62
6.2.4.2. Variando el número de máquinas . . . 63
6.2.4.3. Variando el número de máquinas y el tamaño de la Web . . . 63
6.3. Comparación método híbrido e índice invertido . . . 64
6.3.1. Consultas respondidas por segundo . . . 64
6.4. Discusión y comparaciones de los métodos . . . 65

7. Conclusiones y trabajos futuros **66**
7.1. Conclusiones de los experimentos . . . 66
7.2. Conclusión general . . . 66
7.3. Trabajos futuros . . . 67

8. Bibliografía **68**

9. Apéndice **70**
9.1. Código índice invertido . . . 70
9.2. Código Cache dinámico . . . 71
9.3. Código Nodo_lista . . . 73

Índice de figuras

2.1. Esquema de un MBW. 10
2.2. Ejemplo de índice invertido para un conjunto de documentos. 13
2.3. Ejemplo de árbol de Huffman usando los datos de la Tabla 2.1. 17
2.4. Códigos asignados por palabras según el esquema ETDC, figura extraída de [1]. 18
2.5. Wavelet Tree formado con el alfabeto $\sum$ = {_, a, b, c, d, l, r} y el texto "*la_ cabra_ abracadabra*", figura extraída de [1]. 20
2.6. Wavelet Tree con codificación de Huffman orientada a la palabra para el texto "BELLA_ ROSA _ ROSA ,_ ¿ BELLA ? ¿ ROSA ? .", figura extraída de [1]. 22

3.1. Esquema de configuración del sistema de búsqueda. 25
3.2. Esquema de un broker y varias máquinas. 26
3.3. Proceso de búsqueda realizado por cada máquina, en cada superstep. 28
3.4. Proceso para intersectar dos listas ordenadas por Id de documento. 31

4.1. Diseño detallado de la máquina de búsqueda. 34
4.2. Módulo lector. 35
4.3. Módulo separar consulta en términos. 36
4.4. Modulo Intersecta respuestas. 37
4.5. Ordena respuestas y envío de mensajes. 38
4.6. Esquema de la clase Inverted_index. 39
4.7. Pasos de la construcción del índice Wavelet Tree. 41
4.8. Esquema de búsqueda de documentos en el Wavelet Tree. 42
4.9. Diagrama de la búsqueda de los documentos dada las posiciones. 43

5.1. Ejemplo de un índice invertido descomprimido. 45
5.2. Ejemplo de una lista de diferencias de id de documentos. 45
5.3. Ejemplo de una lista de diferencias codificadas. 45
5.4. Tamaño de los índices con respecto al número de documentos. 48
5.5. Muestra la velocidad de respuesta comparado con el tamaño del índice. . . . 51
5.6. Muestra la velocidad de respuesta comparado con el tamaño del índice. . . . 52

6.1. Diseño del método híbrido original. 55
6.2. Consultas por segundo que responde el esquema híbrido(foto referencial). . . 56
6.3. Diseño del método híbrido final. 58
6.4. Tamaño en RAM para distintas configuraciones del método híbrido comparado con el índice invertido. 60

6.5. Consultas por segundo que puede responder el método híbrido. 61
6.6. Escalabilidad del esquema híbrido con distintas configuraciones, para distintas cantidades de documentos en la Web. 62
6.7. Comparación entre las distintas configuraciones del método híbrido, variando la cantidad de máquinas. 63
6.8. Comparación entre las distintas configuraciones del método híbrido, variando la cantidad de máquinas y el número de documentos. 64
6.9. Consultas por segundo que pueden responder los métodos. 65

9.1. Código en c++ de la clase Índice invertido. 70
9.2. Código en c++ de la clase cache dinámico (parte 1). 71
9.3. Código en c++ de la clase cache dinámico (parte 2). 72
9.4. Código en c++ de la clase Nodo lista. 73

Capítulo 1

Introducción

La búsqueda eficiente de información útil sobre grandes colecciones de texto distribuido en un conjunto de computadores es un problema relevante cuya solución ha sido ampliamente estudiada y ha dado lugar a numerosos algoritmos y estructuras de datos para indexación de texto. Las máquinas de búsqueda para la Web constituyen una aplicación de este tipo y son quizás la aplicación de mayor difusión dado el volumen de usuarios que utilizan sus servicios cada día.

La necesidad de información de un usuario es representada por una secuencia de caracteres que llamaremos consulta, la cual está compuesta por un conjunto de palabras o términos. La respuesta a la consulta está constituida por un conjunto de documentos que son los más pertinentes a la consulta de acuerdo a una función de ranking de documentos. Por cada documento se debe presentar un pequeño resumen el cual contiene el texto que rodea a los términos de las consultas. Dicho resumen recibe el nombre de "snippet".

El problema estudiado en este trabajo de memoria se puede describir de la siguiente manera. Dada una colección de N documentos (por ejemplo, páginas Web), se desea construir una estructura de datos (o índice) y algoritmo de búsqueda para encontrar eficientemente los documentos donde se encuentran las ocurrencias de los términos contenidos en la consulta. La respuesta a la consulta debe estar ordenada de acuerdo a una función de ranking de documentos, donde el primer documento es el más relevante a la consulta, y por cada documento en la respuesta es necesario obtener el respectivo snippet.

La eficiencia de una máquina de búsqueda en texto construido para responder miles consultas por unidad de tiempo, donde el tiempo de respuesta de cada consulta individual no debe superar una cierta cota superior, está fuertemente correlacionada con la capacidad del sistema para mantener en memoria principal la mayor cantidad de datos útiles para responder las consultas activas en un periodo dado del tiempo. El supuesto básico es que el costo de acceso a memoria secundaria es mucho mayor que los accesos a memoria principal y que el espacio ocupado por todo el texto y el índice es mucho mayor que el espacio disponible en memoria principal.

La estrategia estándar de indexación y búsqueda sobre grandes colecciones de texto es el índice invertido. La estructura de datos está formada por una tabla de vocabulario con todos los términos distintos que aparecen en el texto. A cada uno de esos términos se le asocia una lista que contiene los documentos en donde aparece el término respectivo. Sin embargo

existen trabajos recientes [1,2], que proponen ocupar otras estructuras alternativas tales como autoíndices comprimidos para texto, los cuales son índices capaces de re-producir el texto sobre el cual fueron construidos y requieren espacio proporcional al del texto comprimido. Es decir, permiten buscar y extraer cualquier parte del texto sin tener que almacenarlo en forma separada como ocurre en el caso del índice invertido.

En el presente trabajo se comparan estas dos estrategias, es decir la estrategia de utilizar índice invertido más texto para responder consultas y la estrategia de utilizar autoíndices comprimidos para texto para responder las mismas consultas. La comparación se realiza en el contexto de máquinas de búsqueda para la Web.

1.1. Objetivos

1.1.1. Objetivo General

Implementar y realizar un estudio comparativo de dos estrategias de indexación y búsqueda sobre texto distribuido en un conjunto de máquinas formando un cluster de procesadores.

1.1.2. Objetivos Específicos

- Implementar una máquina de búsqueda con las estrategias de:
  - Índice invertido particionado por documentos.
  - Texto auto-indexado basado en Wavelet Tree, modificado para responder los documentos que contienen los términos de consulta.
- Realizar experimentos con las dos estrategias de indexación y búsqueda en varios procesadores, utilizando texto y consultas reales.

1.2. Contribuciones

En este trabajo se implementó una máquina de búsqueda que permite probar y evaluar distintos tipos de estrategias sobre un cluster de computadores. Un resultado importante producto de la evaluación de las dos estrategias estudiadas, es decir, la estrategia de índice invertido particionado por documentos y la estrategia de texto comprimido auto-indexado, tiene relación con el desarrollo de una estrategia híbrida que explota las ventajas de ambas estrategias.

Con la estrategia híbrida, en el mismo espacio que ocupa un índice invertido estándar se almacena un cache de listas invertidas y el texto comprimido auto indexado. Esto permite mantener en memoria principal tanto las listas invertidas más utilizadas para responder las consultas activas en un periodo dado del tiempo como el texto requerido para construir los snippets para las respuestas a dichas consultas.

En el caso del índice invertido estándar es necesario recurrir a memoria secundaria para producir los snippets o enviar mensajes a un servidor de documentos, lo cual requiere un costo

en tiempo de ejecución adicional al consumido en el índice. En el caso del texto comprimido auto-indexado mantenido en memoria principal, es posible obtener los snippets sin recurrir a memoria secundaria o servidores adicionales pero el tiempo de ejecución requerido para determinar los documentos que componen la respuesta a la consulta es mucho mayor que en el índice invertido.

La estrategia híbrida aprovecha las ventajas de ambas estrategias determinando los documentos que componen la respuesta a la consulta y los snippets a un tiempo de ejecución menor que las dos estrategias por separado. El cache de listas invertidas permite acelerar la determinación de los documentos que son parte de la respuesta a las consultas y el texto comprimido auto-indexado permite determinar los snippets desde memoria principal.

1.3. Descripción de capítulos

El presente trabajo está organizado de la siguiente manera:

- En el Capítulo 2 se describe el trabajo previo y los conceptos básicos que sirven de base para este trabajo de título.
- En los Capítulos 3, 4 y 5 se describe el diseño de los experimentos y su implementación, y se presentan y discuten los resultados de los experimentos.
- En el Capítulo 6 se propone un método híbrido que reune en una misma estrategia las ventajas de las estrategias de índice invertido y texto comprimido auto-indexado. En el Capítulo 7 se presentan experimentos sobre el nuevo método híbrido y se discuten sus resultados.
- Finalmente en el Capítulo 8 se presentan las conclusiones finales a partir de los resultados obtenidos en los capítulos anteriores.

Capítulo 2

Antecedentes

2.1. Recuperación de la información

En el contexto de la Web, donde se almacena una cantidad muy grande de documentos, la recuperación de información es el proceso por el cual se encuentran los documentos que contienen los términos (palabras) de la consulta ingresada por el usuario. Los documentos recuperados deberán satisfacer las necesidades de información que el usuario expresó en su consulta. Los problemas que enfrenta este tipo de búsqueda, son:

- Cuán bueno es lo recuperado.
- La eficiencia de recuperación.

La solución de estos problemas son objeto de estudio actualmente, estos tratan de dar una cierta *"calificación"* a la información recuperada [9], para poder realizar ranking de respuestas y así decidir cuáles resultados mostrar desde millones de respuestas posibles.

2.2. Motores de búsqueda Web

Los motores de búsqueda Web (MBW) son herramientas que permiten localizar y recuperar información que se encuentra en La Web, estructurando la información en bases de datos e índices. Dada una consulta, un MBW entrega una lista de los documentos relevantes a las consultas y un resumen de texto que hace referencia la palabra buscada dentro del documento. El MBW entrega una cantidad limitada de documentos por cada consulta, por ejemplo, los 10 primeros en el ranking de documentos.

El esquema actual de un MBW está conformado por el broker (o máquina recepcionista), un conjunto de procesadores y un servidor de texto, como se muestra en la Figura 2.1.

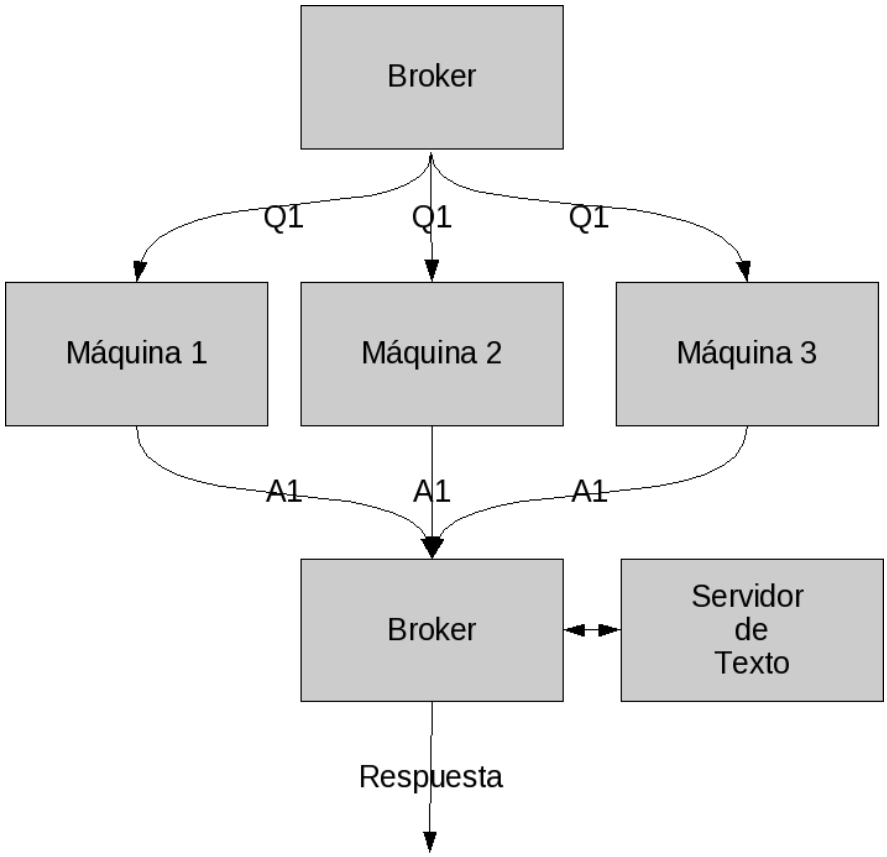

Figura 2.1: Esquema de un MBW.

La resolución de una consulta sobre este esquema puede resumirse en el siguiente proceso:

- Un usuario realiza una consulta a través de un browser.
- La consulta llega al broker que identifica y separa los términos de la consulta, luego envía estos a las máquinas que contienen el índice, para extraer sus listas de documentos.
- Las máquinas intersectan las listas de documentos de los términos, los ordenan de acuerdo a una función de ranking y envían la solución al broker.
- El broker mezcla las respuesta obteniendo una respuesta global.
- Luego el broker envía la petición de resúmenes de los documentos a los servidores de texto, por cada documento que se deba mostrar en el resultado.
- Luego el broker envía los resultados con sus resúmenes al browser del usuario que realizó la consulta.

2.3. Computación paralela sobre memoria distribuida

Si bien no todos los problemas son paralelizables de manera eficiente y la comunicación entre los distintos computadores que componen un cluster de procesadores con memoria distribuida implica un costo, sí existen muchas aplicaciones importantes que admiten una paralelización eficiente. Por ejemplo, buscar en un arreglo de manera secuencial toma tiempo proporcional al largo del arreglo, es decir tiempo $O(n)$. Si, por otro lado, se separa ese arreglo en P máquinas, cada una con arreglo de tamaño $\frac{n}{P}$, buscar en paralelo tomará tiempo $O(\frac{n}{P})$, lo que es un aumento de velocidad lineal en P.

Los buscadores para la Web presentan este grado de paralelización ideal similar a $O(\frac{n}{P})$ en general, donde n es el largo promedio de las listas invertidas asociadas a los términos de las consultas activas en un periodo del tiempo. Pero dada la magnitud del uso de recursos de hardware y el volumen de datos que manejan, la reducción de los factores constantes en $O(\frac{n}{P})$ es algo muy importante. Una reducción del 10 % o incluso menos, puede tener un impacto relevante en la reducción del costo de operación del centro de datos a lo largo del tiempo. Los buscadores más conocidos operan a una tasa de consultas de varios miles por segundo, lo cual requiere del uso de una cantidad importante de procesadores los cuales demandan un consumo de energía eléctrica y costos de operación y mantención importantes.

Al incrementar el número de procesadores P lo que se hace es aumentar el nivel de particionamiento del texto, es decir, se reduce n, lo cual permite reducir el tiempo de respuesta de consultas individuales. Por otra parte, cada uno de los P procesadores se replica D veces para mejorar la cantidad de consultas por segundo que es capaz de resolver el sistema, lo cual incrementa aun más el total de máquinas instaladas en el centro de datos.

Existen dos modelos de computación paralela sobre memoria distribuida:

- Modelo síncrono.
- Modelo asíncrono.

En el modelo síncrono se deben definir políticas de sincronización global entre los threads que colaboran para resolver cada consulta. En un cluster de memoria distribuida, los threads deben sincronizarse mediante mensajes u otro medio de comunicación. Las ventajas son que al no mantener recursos comunes, no hay que preocuparse de que dos o más procesos modifiquen partes sensibles del sistema. La desventaja es que para paralelizar tareas se deben establecer protocolos de comunicación entre ellas, y además sincronizar para que la temporalidad de la comunicación no afecte al sistema completo.

En el modelo asíncrono no existe un intervalo de tiempo constante entre cada evento. La ventaja de este esquema es que no hay que sincronizar los threads. La desventaja es que hay que tener cuidado con la temporalidad de las acciones, ya que dos o más threads pueden intentar modificar concurrentemente algún recurso vital del sistema. Para solucionar esto se implementan mecanismos de sincronización local, que no permiten que dos o más threads modifiquen partes sensibles del sistema al mismo tiempo, y esto produce que en ciertos intervalos las acciones se encolen, disminuyendo la paralelización del sistema.

Dada las características del problema, se escoge el esquema síncrono para el presente trabajo, dado que cada máquina puede responder las consultas con su parte de la información

y comunicar sus resultados o consultas a las otras máquinas, lo que corresponde bien con el esquema síncrono. La ventaja de utilizar el modelo síncrono para realizar la comparación entre las estrategias de procesamiento de consultas estudiadas en este trabajo de memoria, es que el modelo síncrono de computación paralela posee una estructura de cómputo bien definida lo cual facilita la evaluación experimental de algoritmos y permite que distintas estrategias puedan ser comparadas bajo la misma base.

Lo anterior es sólo para efectos de la comparación entre las estrategias estudiadas, puesto que las implementaciones de estas estrategias son fácilmente adaptables al modelo asíncrono, es decir pueden ser ejecutadas sobre ambos modelos de computación paralela prácticamente sin cambios de importancia. Simplemente en el caso síncrono existe un solo thread por procesador que está encargado de ejecutar los pasos asociados a la solución de cada consulta activa, y comunicarse y sincronizarse con todos los demás threads ubicados en los otros procesadores. En el caso asíncrono existe un thread por cada consulta activa y dichos threads se comunican con sus similares en los otros procesadores de manera completamente asíncrona formando pares respecto del paso de mensajes entre ellos.

2.4. Índices Invertidos

Un índice invertido [12,13] es una estructura de datos, la cual contiene una tabla de vocabulario con todos los términos distintos que aparecen en el texto. A cada uno de esos términos se le asocia una lista de posteos o lista invertida, la cual contiene los documentos en donde ocurre ese término y la frecuencia de la palabra en el documento o algún otro indicador que permita realizar un ranking entre los documentos.

En el ejemplo de la Figura 2.2, si consideramos la palabra "casa", su lista invertida es D1, D2, D4 y D5 lo cual significa que los documentos representados por los identificadores D1, D2, D4 y D5 contienen dicha palabra. Esta lista puede estar ordenada por la frecuencia de la palabra en el documento, o por la importancia de esta palabra dentro del documento (por ejemplo, si está en el título o en un apéndice). Estos índices muestran una gran eficiencia [4].

Este tipo de índice es altamente usado en los buscadores actuales, en donde al buscar más de una palabra, como por ejemplo *"casa usada"*, lo que se hace es obtener la lista de *"casa"* y luego la lista de *"usada",* las cuales se intersectan para obtener los documentos que contengan las dos palabras.

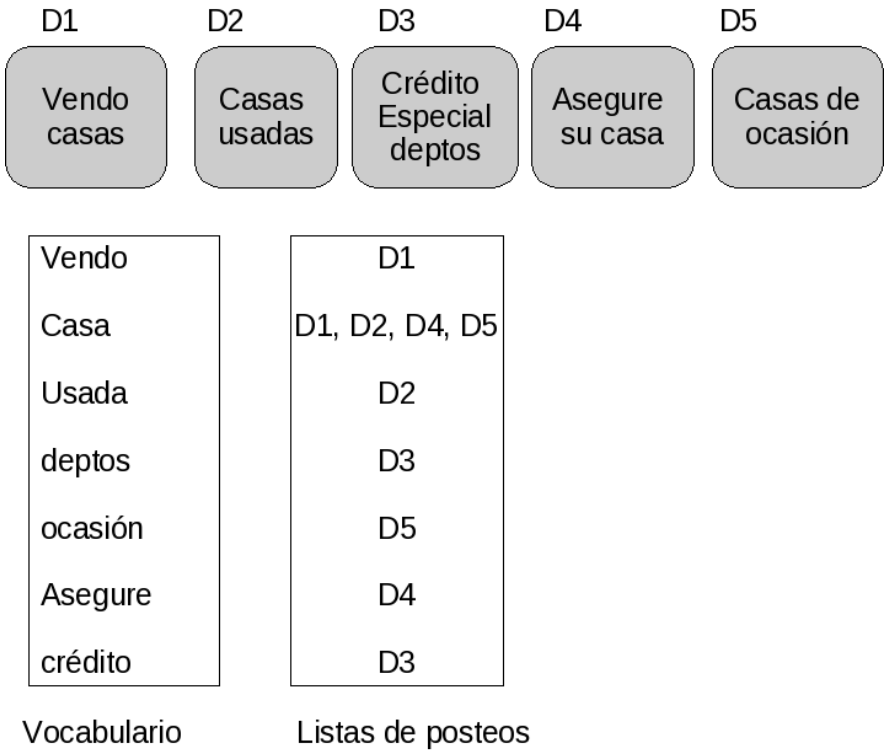

Figura 2.2: Ejemplo de índice invertido para un conjunto de documentos.

2.4.1. Particionado de índices

En un ambiente de computación paralela sobre memoria distribuida, se pueden aplicar dos tipos básicos de particionamientos en un índice:

- Distribución por documentos.
- Distribución por términos.

En ambos casos se pretende balancear la carga de trabajo entre las máquinas que conforman el cluster [5].

Dada una colección de N documentos y un servidor con P procesadores, dichas estrategias de particionamiento se definen de la siguiente manera.

2.4.1.1. Índice particionado por documentos

Los documentos se distribuyen entre los P procesadores, siguiendo alguna regla como *iddoc* mód P con lo que se consigue una distribución balanceada de los documentos. También se pueden usar técnicas más complejas de distribución de documentos como las que se muestran en [6].

Una vez distribuidos los documentos, cada uno de los P computadores construye un índice invertido con sus documentos locales (cada máquina tienen aproximadamente N/P

documentos). Para procesar una consulta, basta con enviarla a todas las máquinas para que éstas la resuelvan localmente y envíen una respuesta. Cabe señalar que se requiere un proceso adicional en el broker para ordenar todos los resultados locales de las máquinas y así obtener los mejores K documentos globales.

2.4.1.2. Índice particionado por términos

Se construye un índice de manera global, utilizando el vocabulario de todos los documentos, luego se distribuyen los términos entre los P procesadores siguiendo alguna regla como $idterm$ mód P con lo que se consigue una distribución balanceada de los términos. También se pueden usar técnicas más complejas de distribución de documentos como las que se muestran en [7].

Una vez distribuidos los términos, se construye en el broker un índice que señala qué procesador almacena cada término para que al realizar las consultas éstas sean enviadas sólo a los computadores que tienen estos términos y no a todos, disminuyendo así el costo de la comunicación.

2.4.1.3. Ventajas y desventajas de ambos métodos de distribución del índice

La ventaja del método de distribución por términos es que por cada consulta activa menos procesadores participan en la solución de la consulta, lo cual en total reduce la cantidad promedio de máquinas involucradas por consulta. Como se explica más abajo, esto tiene implicancias importantes en el uso eficiente de los recursos de hardware para resolver un conjunto dado de consultas. Sin embargo, si las listas invertidas son muy grandes la cota superior para el tiempo de respuesta para una consulta cualquiera puede ser imposible de alcanzar. También para el caso de consultas en que se requiera calcular la intersección de listas invertidas, el costo de la comunicación entre procesadores puede ser muy alto si los términos de las consultas están ubicados en procesadores distintos.

Por otra parte el costo de construcción del índice particionado por términos es extremadamente alto en comunicación puesto que es necesario poner la lista invertida completa de un término dado en un mismo procesador y la situación de inicio para la construcción es que el conjunto de documentos está uniformemente distribuido en los P procesadores.

La ventaja del método de distribución por documentos es que el costo de construcción y procesamiento de consultas es muy sencillo. En cada uno de los P procesadores se construye un índice invertido con los documentos almacenados en el procesador. Esto además tiene la ventaja de que las intersecciones entre listas invertidas siempre son entre listas almacenadas en el mismo procesador.

Sin embargo, cada consulta debe ser enviada a todos los P procesadores, es decir, cada consulta usa todos los recursos de hardware. Esto puede incrementar significativamente el costo pagado por latencias de dispositivos tales como discos y red, y otros tales como administración del estado de las consultas activas y planificación de los threads.

Esto porque en promedio si hay $Q \cdot P$ consultas activas en el sistema completo, entonces existen $Q \cdot P$ consultas siendo resueltas en cada procesador. Si el costo de resolver cada consulta es $O(\ell + S/P)$ donde S/P es el largo promedio de las listas invertidas en el procesador

y ℓ la latencia acumulada que produce la consulta en el procesador, entonces el costo total de procesar las Q consultas activas es $O(Q \cdot (P \cdot \ell + S))$. Por otra parte, el costo de procesar una consulta en el método de distribución por términos es $O(\ell + S)$ y si las $Q \cdot P$ consultas se distribuyen uniformemente en cada procesador de manera que cada procesador resuelve Q consultas en promedio, entonces el costo es $O(Q \cdot (\ell + S))$. Es decir, el costo de latencia acumulada en la estrategia de distribución por términos es P veces menor que en la estrategia de distribución por documentos.

En general, el método de distribución por términos es más eficiente que el de distribución por documentos para aplicaciones particulares de máquinas de búsqueda tales como Flickr[1] , el cual es un caso en que los documentos son textos muy pequeños y por lo tanto el ranking de documentos para las consultas se hace sin realizar la intersección de las listas invertidas respectivas.

En este trabajo de memoria las consultas de interés son aquellas en que es necesario realizar la intersección de listas invertidas y por lo tanto se utiliza el índice invertido particionado por documentos.

2.5. Compresión de textos

La compresión de textos es una técnica usada para representar un texto usando un menor espacio, tomando ventajas de las regularidades de un texto no aleatorio. Dos de las mayores ventajas son:

- Reduce el espacio requerido por el texto.
- Aumenta la eficiencia de la transmisión de texto.

Ambas ventajas son válidas tanto en la transmisión y almacenamiento de texto entre computadores a través de una red y entre memoria primaria y secundaria de un único computador. La mayor desventaja de la compresión es que el tiempo de procesamiento aumenta, esto bajo la premisa de que se debe descomprimir el texto para procesarlo.

2.5.1. Plain Huffman

La técnica de compresión o codificación de Huffman es una de las técnicas más usadas para comprimir textos debido a su enfoque de mínima redundancia [3]. Su aplicación consta de recodificar los caracteres que conforman un texto, asignándoles un código en bits, para después remplazar las apariciones de esos caracteres por dicha codificación. La codificación se ve beneficiada del hecho que se le asignan menos bits a los caracteres más frecuentes, codificando así gran parte del texto en pocos bits. Para asignar los códigos a los caracteres, se construye el árbol de Huffman asignando los código de acuerdo a la frecuencia de los caracteres, como se ve en el siguiente proceso, extraído de wikipedia[2]:

[1] www.flickr.com

[2] http://es.wikipedia.org/wiki/Algoritmo_de_Huffman.

1. Se crean varios árboles, uno por cada uno de los caracteres del alfabeto, consistiendo cada uno de los árboles en un nodo sin hijos, y etiquetado cada uno con su símbolo asociado y su frecuencia de aparición.

2. Se toman los dos árboles de menor frecuencia, y se unen creando un nuevo árbol. La etiqueta de la raíz será la suma de las frecuencias de las raíces de los dos árboles que se unen, y cada uno de estos árboles será un hijo del nuevo árbol. También se etiquetan las dos ramas del nuevo árbol: con un 0 la de la izquierda, y con un 1 la de la derecha.

3. Se repite el paso 2 hasta que sólo quede un árbol.

Para obtener el código asociado a un símbolo se debe proceder del siguiente modo:

1. Comenzar con un código vacío.

2. Iniciar el recorrido del árbol en la hoja asociada al símbolo.

3. Comenzar un recorrido del árbol hacia arriba.

4. Cada vez que se suba un nivel, añadir al código la etiqueta de la rama que se ha recorrido.

5. Tras llegar a la raíz, invertir el código, el resultado es el código Huffman deseado.

Para obtener un símbolo a partir de un código se procede como a continuación:

1. Comenzar el recorrido del árbol en la raíz de éste.

2. Extraer el primer símbolo del código a descodificar.

3. Descender por la rama etiquetada con ese símbolo.

4. Volver al paso 2 y avanzar al segundo símbolo en la codificación hasta que se llegue a una hoja, que será el símbolo asociado al código.

En la Tabla 2.1 se muestra un ejemplo hipotético de cómo, dado un alfabeto con sus frecuencias, se obtiene el árbol de Huffman que se muestra en la Figura 2.3.

Símbolo	Frecuencia
A	0.15
B	0.30
C	0.20
D	0.05
E	0.15
F	0.05
G	0.10

Cuadro 2.1: Tabla de alfabeto y su frecuencia.

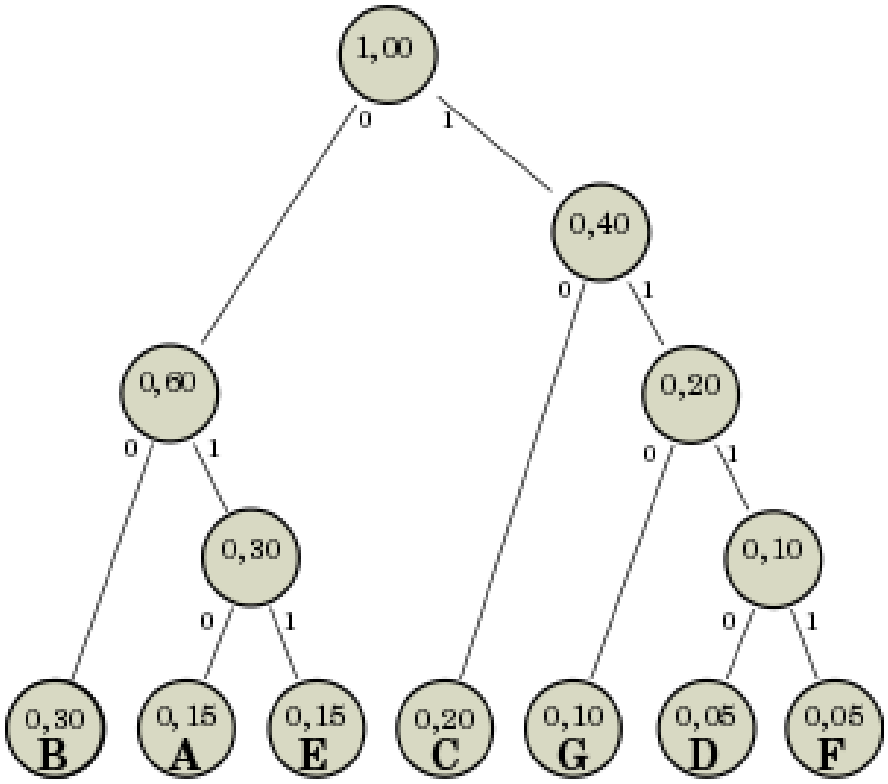

Figura 2.3: Ejemplo de árbol de Huffman usando los datos de la Tabla 2.1.

2.5.2. End-Tagged Dense Code (ETDC)

ETDC es una técnica de compresión orientada a bytes. Esto quiere decir que, dado un texto, se extrae el vocabulario y a cada palabra se le asigna un código formado por uno o más bytes. La codificación ETDC utiliza el primer bit de cada byte como marca de fin de código. Dicha marca permite distinguir cuáles bytes son los que terminan una palabra y cuáles no.

En particular ETDC marca en 1 el bit más significativo del último byte de cada código, y en 0 el bit más significativo de los restantes bytes. La marca al final del código es algo realmente importante pues implica que no existen 2 códigos repetidos. Dado que únicamente el bit de marca del último byte de cada código vale 1, ningún código podrá ser prefijo de otro.

Resumiendo, dado que existen 128 codificaciones de 8 bits que son de finalización (finalizador) y 128 codificaciones de 8 bits que son de continuación (continuadores), el proceso de codificación es el siguiente, como se ve en [1]:

- Se colocan las palabras del vocabulario en una lista.
- Se asigna a las primeras 128 palabras los bytes del 0 al 127. Las palabras desde la posición 128 hasta $(2^7 * 2^7) + 2^7 - 1 = 16.511$ reciben una codificación de 2 bytes donde el primero es alguno de los continuadores y el segundo un finalizador.

- Si las palabras del vocabulario sobrepasan las 16.511 +128, se les asignarán códigos de 3 bytes, donde los dos primeros serán continuadores y el último un finalizador, como se explica en la Figura 2.4.
- El proceso continúa asignando secuencialmente los códigos a todas las palabras del vocabulario. Sin embargo, en casos de interés práctico (por ejemplo, colecciones de documentos o paginas Web) basta con 3 bytes para codificar el vocabulario.

Pos. Palabra	código asignado
0	**1**0000000
1	**1**0000001
2	**1**0000010
...	...
$2^7 - 1 = 127$	**1**1111111
$2^7 = 128$	**0**0000000:**1**0000000
129	**0**0000000:**1**0000001
130	**0**0000000:**1**0000010
...	...
255	**0**0000000:**1**1111111
256	**0**0000001:**1**0000000
257	**0**0000001:**1**0000001
258	**0**0000001:**1**0000010
...	...
$2^7 2^7 + 2^7 - 1$	**0**1111111:**1**1111111
$2^7 2^7 + 2^7 = 16512$	**0**0000000:**0**0000000:**1**0000000
16513	**0**0000000:**0**0000000:**1**0000001
...	...

Figura 2.4: Códigos asignados por palabras según el esquema ETDC, figura extraída de [1].

Cabe destacar que si las palabras se ordenan por frecuencia, se aumenta el nivel de compresión, pues se le asignan códigos más cortos a las palabras que más se repiten en el texto.

2.5.3. Técnica de compresión a ocupar

La técnica que será ocupada en el presente trabajo es la compresión de Huffman a la palabra, pues como se muestra en [8], es más eficiente al caso particular de un conjunto grande texto.

2.6. Autoíndices comprimidos

Un autoíndice comprimido es una estructura que reemplaza el texto con una representación que ocupa un espacio proporcional al texto comprimido (por lo que si el texto es compresible ocupa menos espacio), también soporta búsquedas indexadas sobre el texto y

una rápida extracción de cualquier porción de éste [14] (el índice reemplaza al texto). Una importante característica es que los autoíndices pueden buscar sin necesidad de acceder al texto (pues el índice contiene al texto).

2.6.1. Rank y select

Rank y select son operaciones sobre arreglos de n bits $B = [b_1, b_2, b_3, ..., b_n]$, se definen como se presenta a continuación:

1. $Rank_{b_i}(B, k)$= la cantidad de veces que aparece b_i dentro de los primeros k elementos de B.

2. $Select_{b_i}(B, j)$= la posición donde aparece la j-ésima aparición de b_i.

Por ejemplo si $B = [100101001]$, entonces :

- $Rank_1(B, 4)$=2, lo que significa que en los primeros 4 elementos del arreglo B hay dos unos.

- $Select_0(B, 3)$=5, lo que significa que el tercer cero se encuentra en la quinta posición del arreglo B.

2.6.2. Wavelet Tree

Un Wavelet Tree [20] es un árbol binario balanceado, en donde cada hoja corresponde a un símbolo del alfabeto del texto que se está indexando.

La construcción de un Wavelet Tree se realiza mediante el siguiente algoritmo, como también se describe en [1]:

- La raíz del árbol es un vector de bits, de la misma longitud que el texto, en donde son 1 los caracteres que aparecen en la segunda mitad del alfabeto.

- De este modo los 0's corresponderán al hijo izquierdo del árbol y los 1's al derecho

- Se repite el proceso para cada uno de los nodos hijos (dividiéndose a la mitad el numero de caracteres indexados en cada nodo), hasta llegar a las hojas.

De esta manera el texto original está representado por el árbol y el alfabeto. En la Figura 2.5 se puede ver un ejemplo con el texto "la_cabra_abracadabra", siendo el alfabeto $\sum$ = {_, a, b, c, d, l, r}.

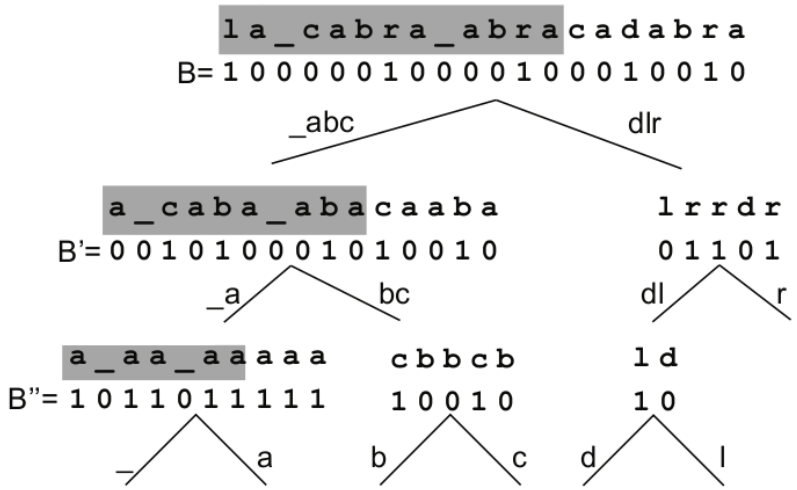

Figura 2.5: Wavelet Tree formado con el alfabeto $\sum$ = {_, a, b, c, d, l, r} y el texto "*la_ cabra_ abracadabra*", figura extraída de [1].

Dado el árbol, se puede obtener el símbolo de cualquier posición del texto original, además se puede saber de manera eficiente cualquier ocurrencia de un símbolo en el texto (puesto que todos los símbolos están en las hojas del árbol, basta con hacerles un seguimiento ascendente, con las operaciones de rank y select que se explican a continuación).

2.6.3. Wavelet Tree con codificación de Huffman a la palabra

Si bien en la Sección 2.5.3 se muestra la construcción de un Wavelet Tree con los términos del vocabulario, en [1,2] se muestra que es posible construirlo con la codificación Huffman de estos, aunque el árbol pierda la característica de balanceado. La idea es que el código asociado al término indique su posición en el Wavelet Tree,por lo que será más eficiente recuperar los términos más frecuentes pues estarán a una menor distancia de la raíz. Sobre la codificación a ocupar, como se demuestra en [8], si bien utilizar Huffman sobre texto supone tasas de compresión del 60 %, al cambiar el enfoque a codificar las palabras se obtuvieron tasas del 25-30 %, y dado que la codificación de Huffman basada en palabras y orientada al bit es óptima [8].

El proceso de construcción y búsqueda en este nuevo Wavelet Tree será detallado a continuación, como se ve en [1].

2.6.3.1. Construcción

En la raíz del Wavelet se tendrá un arreglo B de n bits (donde n es el número de palabras del texto), el valor de cada posición del arreglo estará dado por el valor del primer bit de la

codificación de la palabra. De esta manera si la i-ésima palabra tuviese codificación 01011 entonces $B_i = 0$, si fuese 11011 B_i sería 1, además la raíz B tiene dos hijos uno que tienen el primer bit de la codificación con 0 y otro con el primero bit de la codificación en 1, entonces se tendrá que todas las palabras cuyo código comience con 0 estarán en el hijo izquierdo y los que comiencen con 1 en el hijo derecho, y así sucesivamente por lo hijos de estos. Por lo que los pasos para construir el Wavelet Tree son los siguientes:

- Generar el vocabulario del texto, en este caso las palabras.
- Asignarles las frecuencias al vocabulario.
- Generar los códigos de Huffman para las palabras, dada las frecuencias de éstas.
- Colocar en la raíz del Wavelet Tree un arreglo con el primer bit de la codificación de cada palabra, este es el nivel 0 .
- En el hijo izquierdo de la raíz colocar los segundos bits de la codificación de las palabras de todas las codificaciones que comienzan con 0, este es el nivel 1.
- En el hijo derecho de la raíz colocar los segundos bits de la codificación de las palabras de todas las codificaciones que comienzan con 1, este es el nivel 1.
- Se repiten los pasos anteriores, hasta colocar todos los bits de la codificación de las palabras, quedando la altura del árbol igual a la codificación mas larga de Huffman. pero habrán ramas del árbol que serán mas cortas, y que pertenecerán a las palabras mas frecuentes del texto.

En la Figura 2.6 se muestra un ejemplo gráfico de la construcción de un Wavelet Tree codificado con Huffman a la palabra.

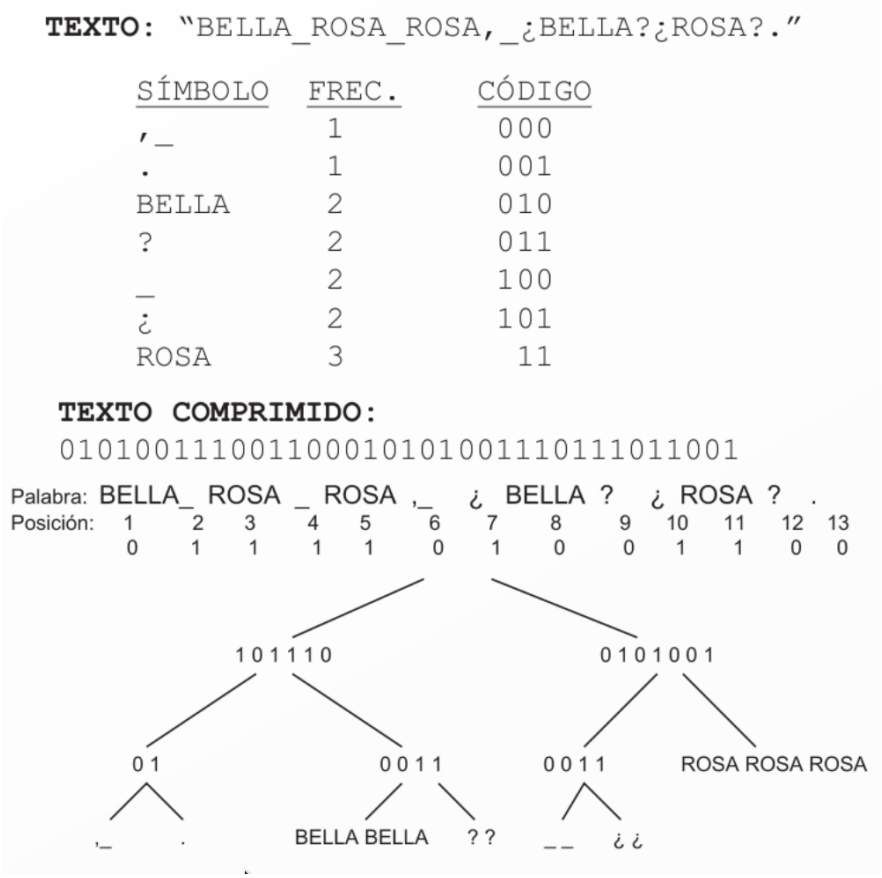

Figura 2.6: Wavelet Tree con codificación de Huffman orientada a la palabra para el texto "BELLA_ ROSA _ ROSA ,_ ¿ BELLA ? ¿ ROSA ? .", figura extraída de [1].

2.6.3.2. Búsqueda

Dada una palabra el proceso de búsqueda de ésta en el Wavelet Tree corresponde a :

- Encontrar su codificación Huffman.
- Una vez encontrada su codificación se localiza la hoja a la que pertenece, usando la codificación para descender, el tamaño del arreglo de bits en esa hoja muestra la cantidad de ocurrencias de la palabra en el texto
- Una vez ubicada la hoja que representa la palabra buscada, y suponiendo que la palabra está codificada en una secuencia de bits $P = [b_1 b_2 b_3 .....b_m]$, desde la hoja se realiza el siguiente procedimiento:
  - Para cada elemento en este arreglo se realiza $select_{b_m}$("padre del nodo hoja", j), donde j es la posición del arreglo.
  - Esto nos estrega las posiciones de las ocurrencia de la palabra en el arreglo padre, y cada posición sera un j'.

- luego para cada una de las posiciones j' dentro del nodo que es el padre de la hoja realizamos $select_{b_{m-1}}$(padre del nodo, j')
- Repetir este paso hasta llegar a la raíz, en donde las j' obtenidas serán las posiciones dentro del texto original.

Dentro del ejemplo de la Figura 2.6, se realizara la búsqueda de la palabra "ROSA".
Pasos de la búsqueda:

1. Buscamos el código de la palabra "ROSA", el cual es 11.
2. Identificamos la hoja que representa la palabra "ROSA" . La cantidad de elementos en la hoja es 3, por lo que, $J = \{0, 1, 2\}$.
3. El arreglo de bits de la codificación de ROSA es $B = [11]$, el tamaño es 2, el ultimo bit de la codificación es $b_2 = 1$, los select serán de la forma $\forall j \epsilon J$ $select_1$("padre del nodo hoja", j), como se explica en la Sección 2.4.3 la expresión $select_{b_i}$("A", j) devuelve la posición dentro del arreglo "A", del j-ésimo b_i.
4. Como $J = \{0, 1, 2\}$, $j_0 = 0$, $j_1 = 1$, $j_2 = 2$, los select y sus respuestas quedan :
   a) $select_1$("padre del nodo hoja", 0) = 1.
   b) $select_1$("padre del nodo hoja", 1) = 3.
   c) $select_1$("padre del nodo hoja", 2) = 6.
5. Los resultados del punto anterior son las posiciones dentro del arreglo padre, osea es el conjunto $J' = \{1, 3, 6\}$, como el nodo no es la raíz, repetimos el select, pero ahora se pregunta por el siguiente bit en B, que es $b_{1=1}$, y los select son $\forall j' \epsilon J'$ $select_1$("padre del nodo", j').
6. Ahora $J' = \{1, 3, 6\}$, $j'_0 = 1$, $j'_1 = 3$, $j'_2 = 6$, los select y sus respuestas quedan :
   a) $select_1$("padre del nodo", 1) = 3.
   b) $select_1$("padre del nodo", 3) = 5.
   c) $select_1$("padre del nodo", 6) = 11.
7. Pero el padre del nodo es la raíz, así que las posiciones devueltas por el select son las posiciones en el texto, lo que implica que la primera aparición de ROSA es en la palabra 3, la siguiente es en la palabra 5, y la ultima es en la palabra 11, lo que se puede comprobar en la Figura 2.6.

Capítulo 3

Diseño de máquinas de búsqueda

En el presente capítulo se describe el diseño del prototipo de máquina de búsqueda utilizado para realizar la implementación y evaluación de las estrategias de indexación y búsqueda estudiadas en este trabajo.

3.1. Arquitectura

Como estrategia de paralelización se determinó trabajar con el esquema que se utiliza actualmente en los buscadores Web, que es computación paralela con memoria distribuida, lo que implica utilizar una herramienta de comunicación entre las máquinas. El método a ocupar es el pasaje de mensajes utilizando la biblioteca MPI sobre C++. Las máquinas están distribuidas en la misma red y cada una tiene memoria RAM, disco y procesador propio.

Los experimentos son ejecutados sobre una muestra del la Web. Cada una de las P máquinas indexa una porción de la muestra de la Web, manteniendo el índice correspondiente en memoria principal para procesar las consultas. En tiempo de búsqueda, las consultas se reparten entre todas las máquinas. La Figura 3.1 muestra el esquema general para el caso donde hay replicación de procesadores y las consultas se distribuyen en todas las columnas. En cada columna se elige uno de los procesadores para resolver la consulta. Los procesadores de una misma columna mantienen la misma copia del índice.

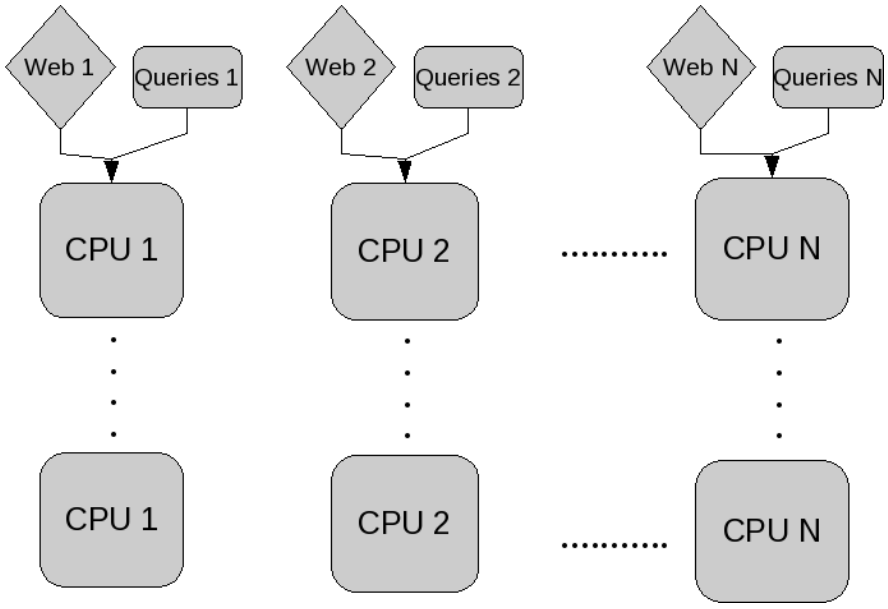

Figura 3.1: Esquema de configuración del sistema de búsqueda.

3.1.1. Diseño del ambiente de comparación

Como se describe anteriormente, las dos estrategias estarán montadas en un protocolo MPI con un diseño orientado a responder consultas. Como se requiere comparar las estrategias bajo exactamente las mismas condiciones de trabajo, se hace que la mayor parte de los componentes de la máquina de búsqueda sean comunes a las dos estrategias y que sólo difiera en aquéllos aspectos que atañen a uno u otro método de indexación y búsqueda. Para esto se diseña una máquina general en donde al consultar por los documentos asociados a una consulta, se realiza la búsqueda en uno u otro índice según sea necesario, lo cual solamente implica reemplazar las implementaciones de determinadas clases de objetos relacionadas con la indexación y búsqueda.

3.2. Diseño de la máquina de búsqueda

La máquina de búsqueda es un programa C++/MPI que recibe las consultas y devuelve los documentos en donde se encuentran contenidos los términos que componen la consulta.

En el caso del índice invertido se busca el término en éste y se responde con la lista de documentos asociada. En cambio si tenemos el esquema del Wavelet Tree, se consultan las posiciones del término dentro del texto y se extrae de esta información la lista de documentos a los cuales pertenece.

Como se aprecia, el resultado en ambos casos es una lista de documentos. En caso que la consulta esté formada por varios términos las listas resultantes se deben intersectar, obteniendo una lista resultante que será ordenada por un puntaje asociado a cada documento. Así se responde los documentos que contengan a todos los términos de la consulta ordenados por la relevancia de estos para dicha consulta.

El esquema de búsqueda consta de un broker que recibe la consulta, y la envía a todas las máquinas del cluster. Luego cada máquina responde esa consulta y envía la respuesta encontrada al broker, como se muestra en la Figura 3.2.

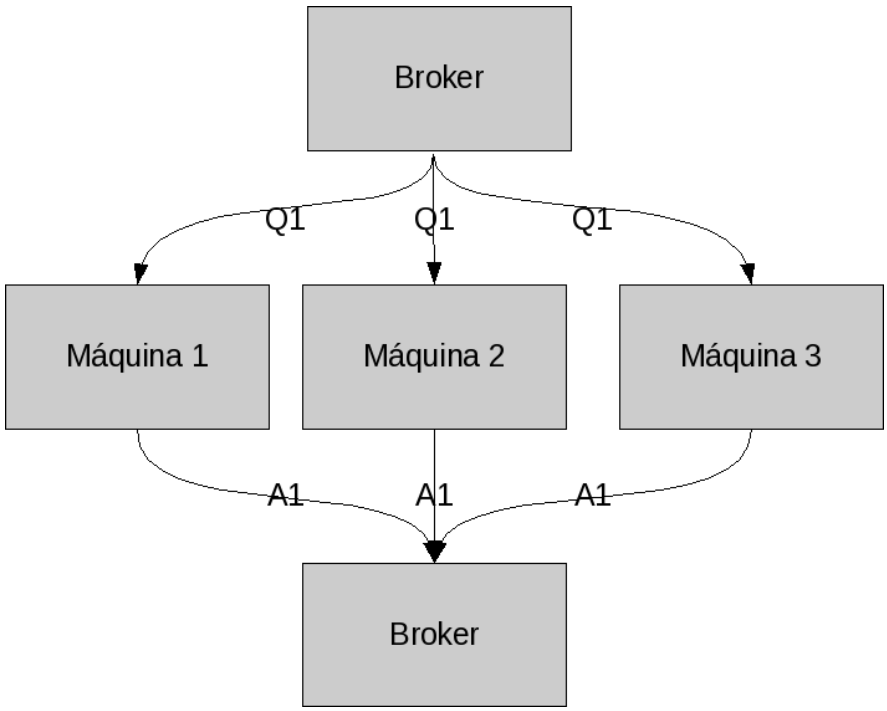

Figura 3.2: Esquema de un broker y varias máquinas.

En este trabajo todas las máquinas son brokers y máquinas de búsqueda a la vez. Esto no afecta la validez de los resultados y permite evitar las complicaciones de implementar un broker y la capa de comunicación entre broker y procesadores, el cual generalmente es otra interfaz de red distinta a la que se utiliza para la comunicación entre procesadores. El diseño consiste en que todas las máquinas realicen el mismo trabajo, permitiendo medir la comunicación y la sincronización entre ellos sin la interferencia de una máquina externa haciendo de broker a través de la misma interfaz de red. Entonces cada procesador simula la recepción de consultas y la procesa como si hubiesen sido enviadas por el broker para determinar los documentos que componen la solución.

El proceso que realiza cada procesador es el siguiente:

1. Leer consultas de la cola de mensajes de entrada: Se lee la cola de mensajes y se ve qué máquina envió qué consulta.

2. Responder las consultas: Los mensajes leídos son consultados al método de indexación y búsqueda correspondiente (ya sea Índice invertido o Wavelet Tree).

3. Encolar las respuestas: Las respuestas obtenidas en el punto 2 a los mensajes leídos en el punto 1, son encolados en la cola de mensajes salientes, enviándolos a la máquina que envió la consulta.

4. Obtención de consultas: Se obtienen nuevas consultas a ser resueltas las cuales simuladamente provienen de la máquina broker.

5. Se encolan las consultas : Las consultas obtenidas en el punto 4 se encolan en la cola de mensajes de salidas y son enviados a todas las máquinas.

6. Se sincronizan todas las máquinas.

La ejecución de los 6 pasos es llamado "superstep" y equivale a un paso de cómputo y comunicación de todas las máquinas. Si bien las máquinas pueden ejecutar los pasos con distintas velocidades, todas terminarán juntas y empezarán juntas, produciendo que no hayan pérdidas de mensajes. También se observa que las consultas son resueltas en un único superstep.

Aunque en el esquema presentado se pueden resolver consultas del tipo "AND" y del tipo "OR", para efectos del presente trabajo se estudian sólo las consultas del tipo "AND", las cuales son el tipo de consultas utilizado por buscadores tales como Google y Yahoo!.

En la Figura 3.3 se muestra gráficamente el proceso descrito anteriormente.

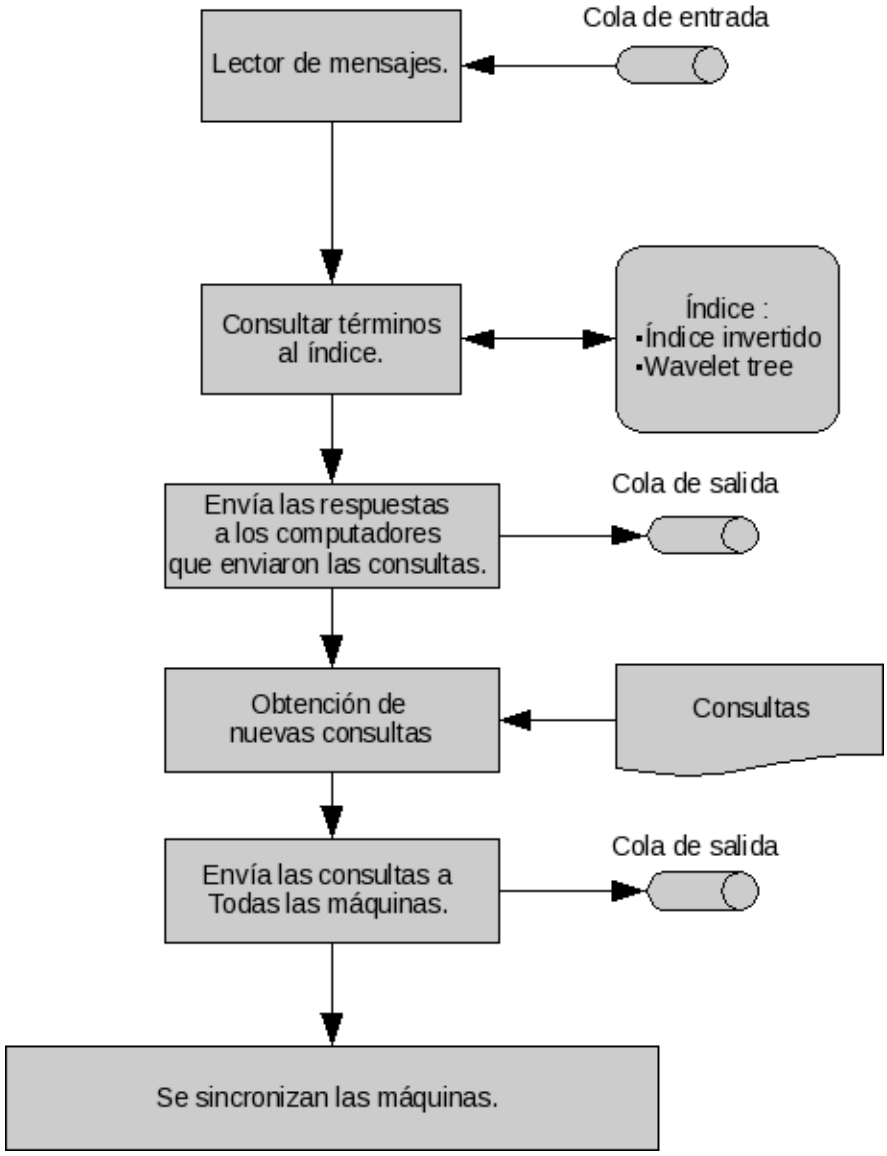

Figura 3.3: Proceso de búsqueda realizado por cada máquina, en cada superstep.

En el Capítulo 4, correspondiente a la implementación, se muestra un diagrama más detallado del proceso realizado por cada procesador.

3.3. Diseño de Índices

3.3.1. Índice invertido

Como se menciona en la Sección 2.4, un índice invertido es una tabla de vocabulario con todos los términos distintos que aparecen en el texto. A cada uno de esos términos se le asocia una lista de posteos o lista invertida, la cual contiene los documentos en donde ocurre ese

término y la frecuencia de la palabra en el documento o algún otro indicador que permita realizar un ranking entre los documentos.

Para construir esta estructura se realizan los pasos de: Extracción de vocabulario, luego a cada palabra se le asocia una lista de los documentos que la contienen y un puntaje que muestra la relevancia de esa palabra en el documento. Estos pasos se explican a continuación.

3.3.1.1. Extracción de vocabulario

El proceso para extraer el vocabulario se detalla a continuación:

- Se recorre todo el texto analizando caracter por caracter, como muestra el siguiente algoritmo, para ingresar las palabras al vocabulario:

  - Se distinguen 2 tipos de caracteres los alfa-numéricos (AN) y los no alfa-numéricos (NAN) los cuales no incluyen los caracteres fin de línea, fin de archivo, ni espacio pues estos son caracteres separadores.
  - Si se lee un caracter AN, éste se guarda en un buffer y se lee el siguiente caracter.
  - Si es un caracter separador el buffer es considerado una palabra y se ingresa al vocabulario, borrando el buffer y leyendo el siguiente caracter.
  - Si es un caracter NAN el buffer es considerado una palabra y se ingresa al vocabulario, borrando el buffer. El caracter recién leído es ingresado al buffer, se lee el siguiente caracter y se ingresa al buffer hasta que el caracter leído sea un caracter AN o separador, repitiendo el proceso anterior.

3.3.1.2. Lista de documentos por palabra

La lista de documentos por palabra se construye analizando nuevamente el texto, y al encontrar una palabra se almacena el documento en donde se encontró.

En esta parte también se calcula el puntaje de la palabra en el documento según alguna estrategia dada tal como el método vectorial de ranking de documentos. Esta es la ventaja del índice invertido, pues el puntaje de la palabra en el documento está precalculado, así luego de intersectar sólo se opera con estos valore pre-calculados para hacer el ranking de documentos. Sin embargo se requiere espacio extra.

Las distintas estrategias de cálculo del puntaje son realizadas en esta parte de la construcción del índice y no son modificables después, sólo se utilizan.

3.3.1.3. Puntaje para el ranking

Se tomará la frecuencia de la palabra dentro del documento como puntaje para realizar el ranking. Este puntaje será precalculado y guardado en el índice.

3.3.2. Wavelet Tree

La estrategia con Wavelet Tree es un poco más complicada y consta de pasos adicionales, pues éste entrega las posiciones en las cuales se encuentra el término buscado dentro de la colección de texto. En este caso la colección de texto es la Web, por lo que hay que realizar un segundo proceso, el cual transforma las posiciones del término en el documento correspondiente, con su puntaje de importancia en relación al término en el documento.

Los pasos para construir el Wavelet Tree son :

- Construir el vocabulario.
- Realizar la compresión.

Para buscar, se le consulta al Wavelet Tree la lista de posiciones de la palabra en la colección de texto y el resultado se transforma en una lista de documentos con su indicador de importancia.

A continuación se detallan los procesos.

3.3.2.1. Vocabulario

El vocabulario es extraído del texto como se detalla en la Sección 3.3.1.1.

3.3.2.2. Construcción del Wavelet Tree

Luego que se tienen todas las palabras que componen el texto, éstas se ordenan de mayor frecuencia en el texto a menor frecuencia, y se le asigna a cada palabra un código de Huffman y se comprime todo el texto en la estructura del Wavelet Tree, como se describe en la Sección 2.4.4 .

3.3.2.3. Lista de documentos por palabra y ranking on-line

Dado que el Wavelet Tree devuelve las posiciones de las palabras dentro de toda la colección de texto, esto no nos permite saber en qué documentos están, por lo que se utiliza una estructura externa que nos permite, dada la posición, saber el documento donde se encuentra la palabra. Esta estructura es cargada junto con el Wavelet Tree en memoria, y permite extraer una lista con los documentos que contienen el término de la consulta y sus frecuencias. De esta manera es posible generar en forma on-line los mismos valores para el ranking de documentos que el índice invertido ya tiene precalculado.

3.4. Funciones Comunes

3.4.1. Intersección de listas de documentos

Dado que responderemos a consultas del tipo “AND”, las lista de documentos en el índice invertido estarán ordenadas por el ID del documento en forma creciente, haciendo que la

función de intersección sea lineal con respecto a la suma de los tamaños de las listas a intersectar, como se muestra en la Figura 3.4.

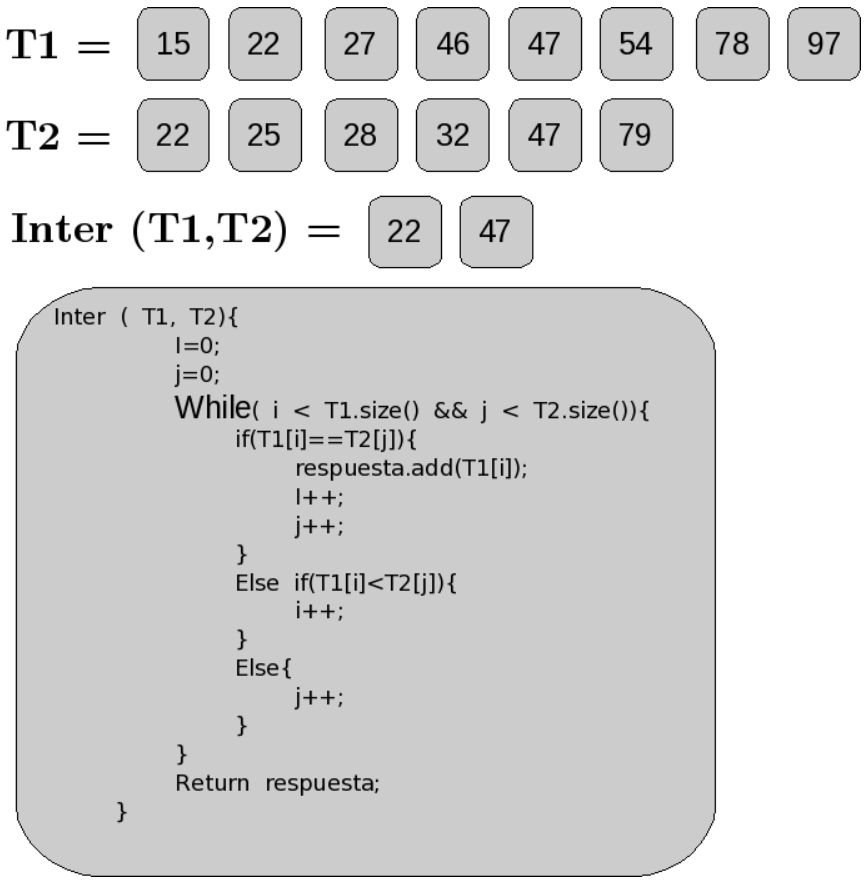

Figura 3.4: Proceso para intersectar dos listas ordenadas por Id de documento.

3.5. Funciones de búsqueda

Las funciones de búsqueda de los dos métodos son las encargadas de mostrar las diferencias de tiempos entre estos, para ambos casos se realiza el siguiente proceso, cuando se tenga una consulta *"term_1+term_2....term_n"*:

- Consultar al índice por las palabras ***term_1*** hasta ***term_n***. Si alguna de las palabras no está en el vocabulario, se responde que la intersección es nula.
- Una vez obtenidas las listas resultantes, éstas se intersectan y el resultado es ordenado, aplicando la función de ranking.
- Se responde la lista a la máquina que envió la consulta.

La tarea de consultar al índice dependerá del índice usado. En el caso del índice invertido:

1. Se buscan los términos dentro del vocabulario.

2. Se recuperan las lista de documentos.

En cambio, si se utiliza el Wavelet Tree.

1. Se busca la codificación de la palabra buscada.
2. Una vez obtenida la codificación, se chequea si está dentro de las hojas del árbol.
3. Luego se obtienen las ocurrencias del texto como se explicó en la Sección 2.6.3.2.
4. Finalmente se extrae la lista de documentos dadas las posiciones de los términos. Cabe destacar que es en este paso en donde se calcula la frecuencia de los documentos.

Luego que se tienen las listas de documentos resultantes, se realizan intersecciones dependiendo de las listas de las palabras buscadas y el resultado se ordena de acuerdo a la función de ranking.

Capítulo 4

Implementación

En este capítulo se presentan detalles sobre la construcción de la máquina de búsqueda utilizada para las pruebas.

4.1. Construcción de la maquina de búsqueda paralela

Como se puede apreciar en el Capítulo 3, para la comparación de las estrategias es necesario construir una máquina de búsqueda que pueda funcionar indistintamente con los dos tipos de índices, para así no favorecer a ninguna de las dos estrategias, y poder concluir que las diferencias de tiempo pertenecen a las estrategias y no a la implementación de la máquina de búsqueda específica.

A continuación en la Figura 4.1 se muestra un diagrama del diseño detallado de la máquina de búsqueda.

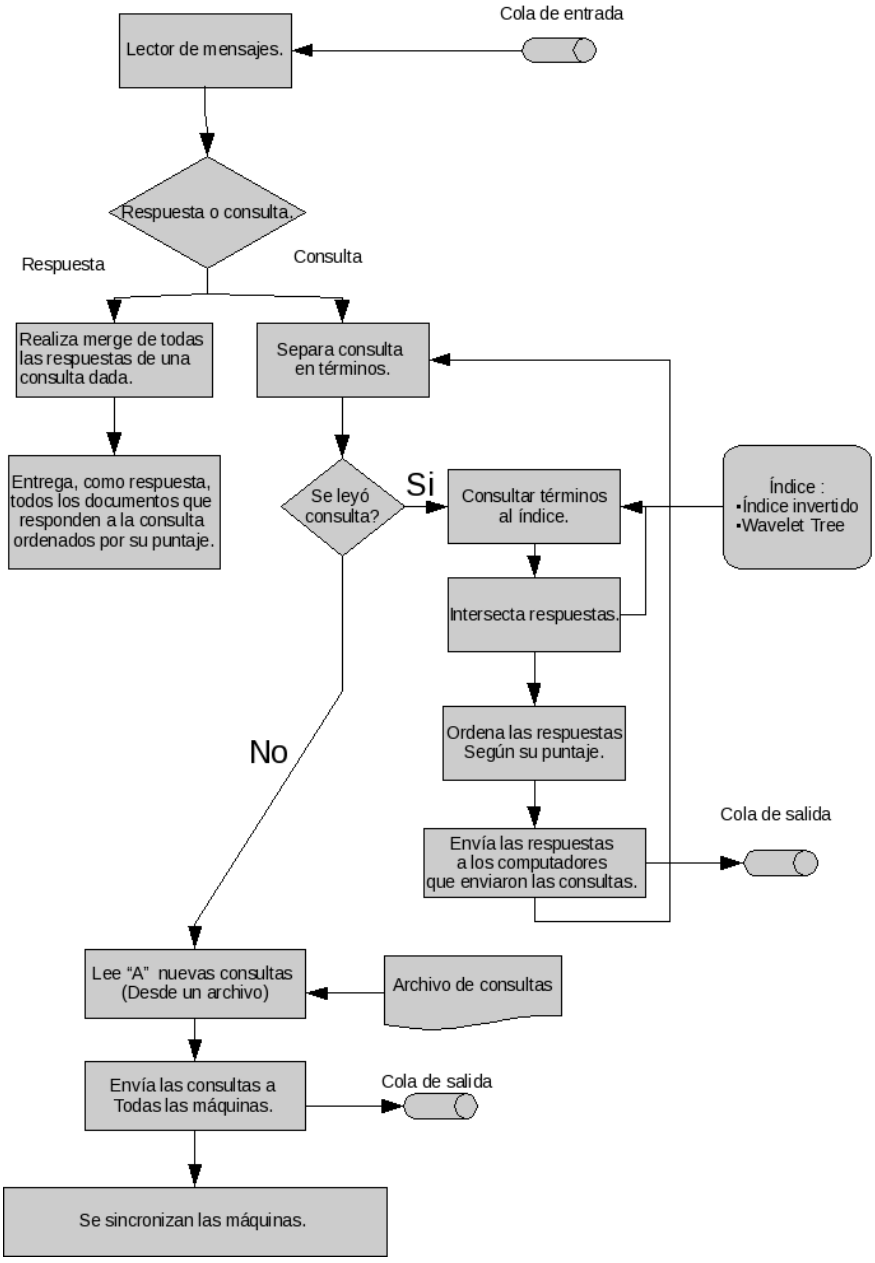

Figura 4.1: Diseño detallado de la máquina de búsqueda.

4.1.1. Detalle de las funciones

4.1.1.1. Lector de mensajes

El módulo de lectura de mensajes está dado por el siguiente proceso:

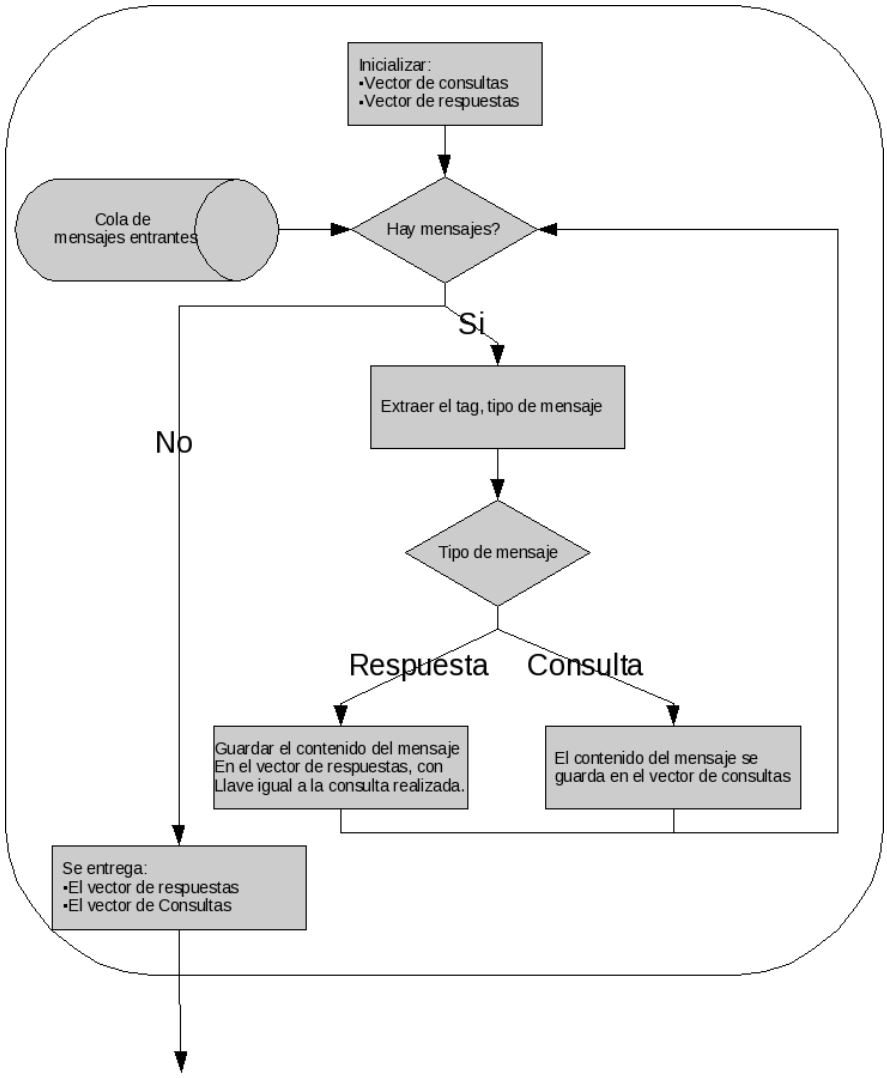

Figura 4.2: Módulo lector.

Como se observa en la Figura 4.2, el módulo lector inicializa los vectores de consultas y respuestas, además añade los datos que extrae de la cola de mensajes.

El vector de consultas es un vector de strings que contiene en cada posición una consulta que fue leída de la cola de mensajes y la máquina que la envió.

El vector de respuestas contiene un identificador de consulta y la respuesta de ésta por máquina.

Estos vectores serán usados después por los demás módulos para continuar con el proceso, como se aprecia en diagrama de la Figura 4.1. Los procesos que ocupan los vectores de respuesta y de consultas son independientes y no interfieren entre ellos.

4.1.1.2. Separar consulta en términos

Esta función toma cada entrada en el vector de consultas y extrae las palabras una a una, para consultarlas al índice, como muestra la Figura 4.3

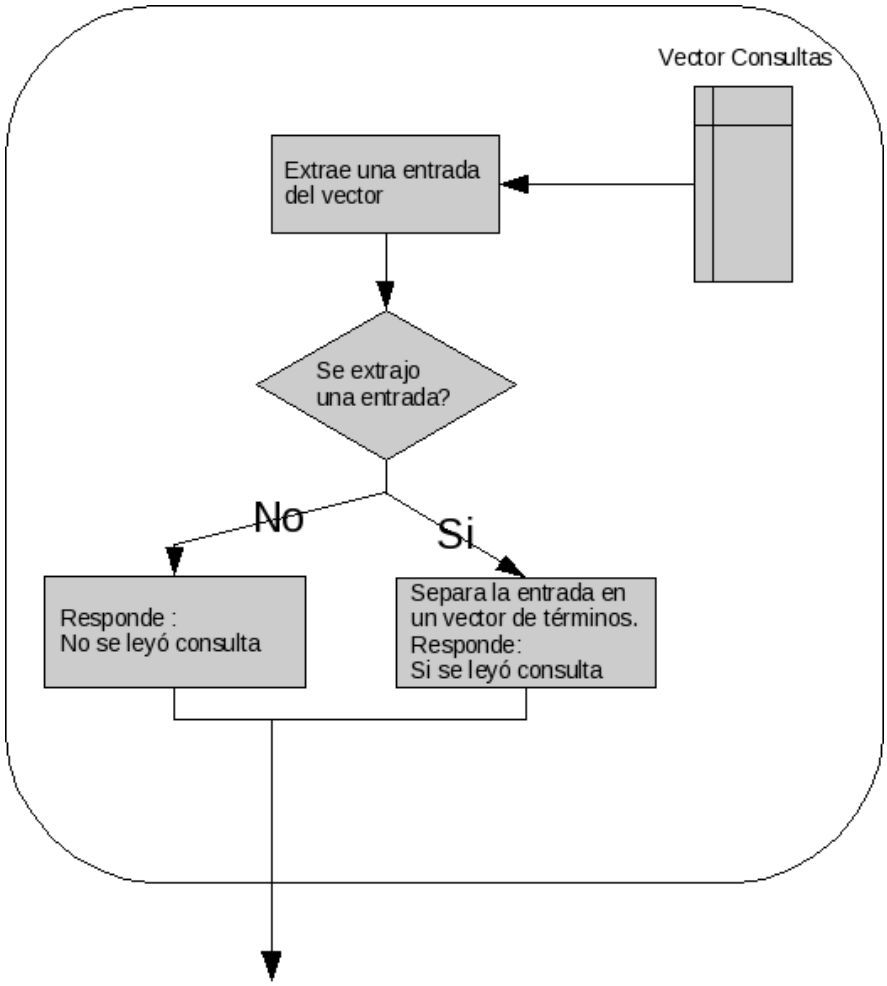

Figura 4.3: Módulo separar consulta en términos.

La salida de este módulo entrega la información de si hay o no más consultas para leer en el arreglo de consultas. Si las hay, pasa al módulo de consulta al índice, sino sigue al módulo de leer consultas.

4.1.1.3. Consultar términos al índice

Esta función será explicada en las secciones 4.2 y 4.3 correspondientes a las estrategias a comparar.

4.1.1.4. Intersecta respuestas

Esta función es la encargada de intersectar las listas de documentos obtenidos al consultar los términos al índice correspondiente. La función devuelve los documentos que estén en todas las listas de términos a consultar. En este módulo se van consultando los términos leídos del vector de términos e intersectando a medida que se van obteniendo las listas de documentos.

Si se obtiene una lista de documentos vacía o la intersección es vacía, la respuesta para esa consulta es vacía, respondiendo inmediatamente, como se muestra en la Figura 4.4.

Cabe destacar que la función de intersección es la que se muestra en la Figura 3.4.

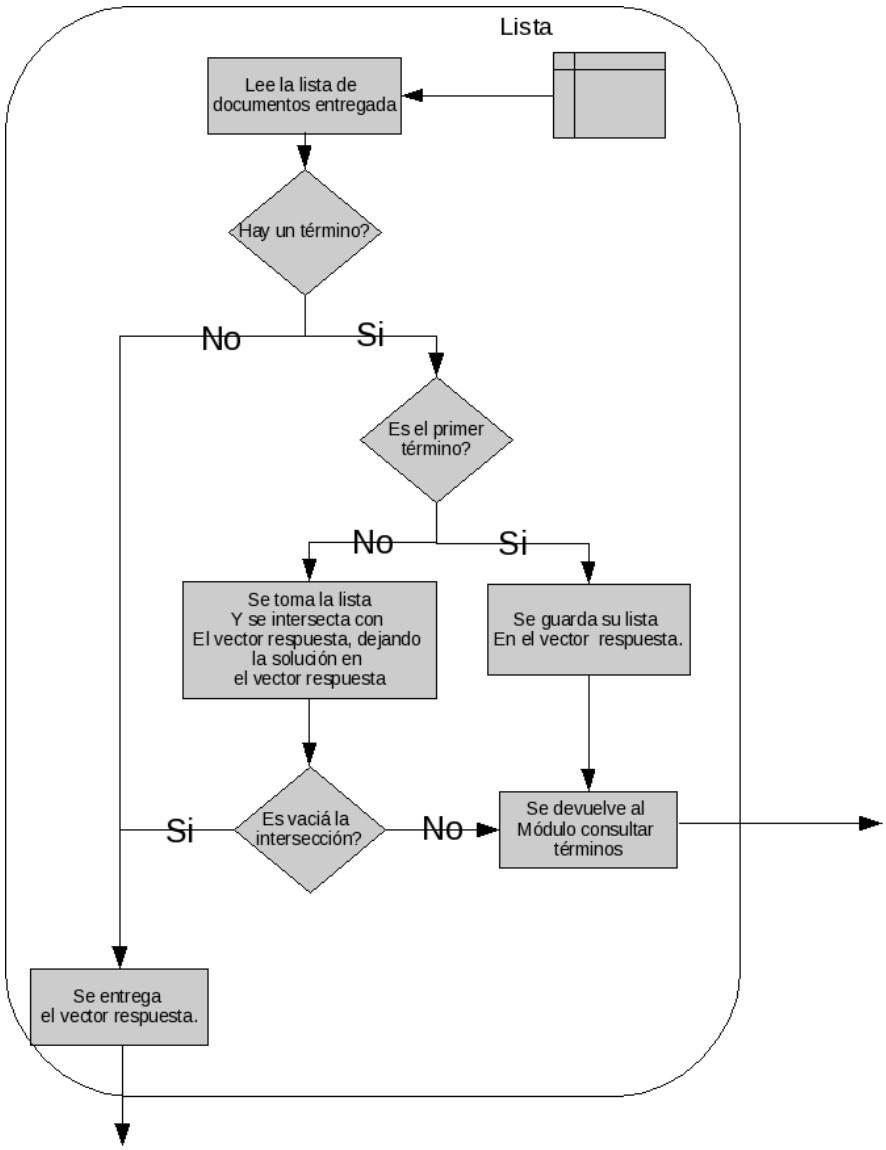

Figura 4.4: Modulo Intersecta respuestas.

4.1.1.5. Ordenar las respuestas según su puntaje

Este módulo toma el vector entregado por el módulo "Intersecta respuestas" y lo ordena según la calificación de cada documento, luego toma la consulta y crea un objeto respuesta, que contiene la consulta y el vector de documentos que la responde, ordenado por la calificación de los documentos como se muestra en la Figura 4.5.

4.1.1.6. Envía respuestas

Este módulo toma el objeto entregado por el módulo anterior y la máquina que envió esa consulta y coloca un mensaje en la cola de mensajes salientes, con destino a la máquina que envió la consulta y el contenido es el objeto entregado por el módulo anterior, como se muestra en la Figura 4.5.

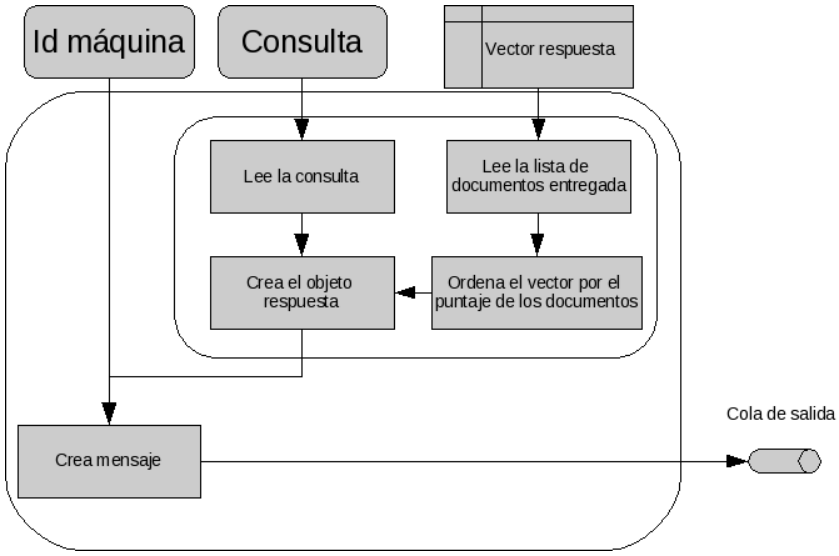

Figura 4.5: Ordena respuestas y envío de mensajes.

4.1.1.7. Lectura de nuevas consultas y envío de éstas

En este módulo se leen A líneas del archivo de consultas y por cada línea leída, se obtiene una consulta que hay que mandar a todas las máquinas.

El número de líneas A es un parámetro del programa.

4.1.1.8. Sincronización

En esta etapa del programa las máquinas se sincronizan, antes de leer los mensajes, asegurándose que no se pierdan, luego comienza nuevamente el ciclo completo.

4.2. Construcción del prototipo con índice invertido

4.2.1. Construcción

La creación del índice invertido sigue el modelo presentado en la Sección 3.3.1, en donde se extrae el vocabulario del texto y luego se crean las listas que indican en qué documento está cada palabra y con qué puntaje.

Para cargar el índice en RAM se lee el archivo donde están las palabras y sus listas, luego se utiliza la interfaz "*añadir_termino()*", Una vez que se ha leído todo el archivo, el índice invertido está listo para ser consultado.

4.2.2. Búsqueda

Para consultar el índice invertido se creó una clase en C++, cuyos métodos principales son:

- buscar(string *s).
- añadir_termino(string *s, vector<int > documentos).

En la Figura 4.6 se muestra un ejemplo del código usado para el índice invertido. En el apéndice se muestran todos los métodos empleados en esta clase.

```
class Inverted_index{
public:
    vector<string > *palabras;
    vector<vector<int> > *listas_de_documentos;
    int indice_actual;
    Inverted_index(int tam){
        palabras= new vector<string>(tam);
        listas_de_documentos= new vector<vector<int> >(tam);
        indice_actual=0;
    }
    ~Inverted_index(){
        palabras->clear();
        delete palabras;
        listas_de_documentos->clear();
        delete listas_de_documentos;
    }
    void agregar(string *_s, vector<int > *Documentos);
    vector <int> * busca(string *s);
};
```

Figura 4.6: Esquema de la clase Inverted_index.

La función de consulta del índice invertido queda definida con la interfaz mostrada en la figura anterior. El módulo de búsqueda recibe una palabra, realiza la acción *"índice_invertido.buscar(palabra)"*, y esto devuelve un vector con la lista de la palabra si la encuentra o un puntero a NULL si la palabra no estaba en el índice.

El costo de la búsqueda dentro del vocabulario es logarítmico, ya que se ocupa búsqueda binaria para encontrar la palabra.

Un proceso off-line es el encargado de ordenar el archivo por las palabras, así al cargarlo éste ya se encuentra ordenado.

4.3. Construcción del prototipo con Wavelet Tree

4.3.1. Código

Para gran parte del código C de este esquema se usó el código original de los autores de [2].

4.3.2. Construcción

La creación del Wavelet Tree consta de más fases que la creación del índice invertido, aunque existen algunas comunes,

1. Extracción de palabras del texto y sus frecuencias.
2. Las palabras son ordenadas por frecuencia.
3. Se aplica una codificación de Huffman a las palabras.
4. Se crea un Wavelet Tree con el texto usando la codificación de Huffman construida en el punto 3.
5. Mientras se crea el Wavelet Tree, se guardan las posiciones de fin de documento en el texto, para posteriormente poder ubicar los documentos a los cuales pertenecen las palabras.
6. Se almacena el texto codificado en binario en la estructura del Wavelet Tree, además del índice que indica los finales de documentos.

Los pasos anteriormente señalados pueden verse en la Figura 4.7.

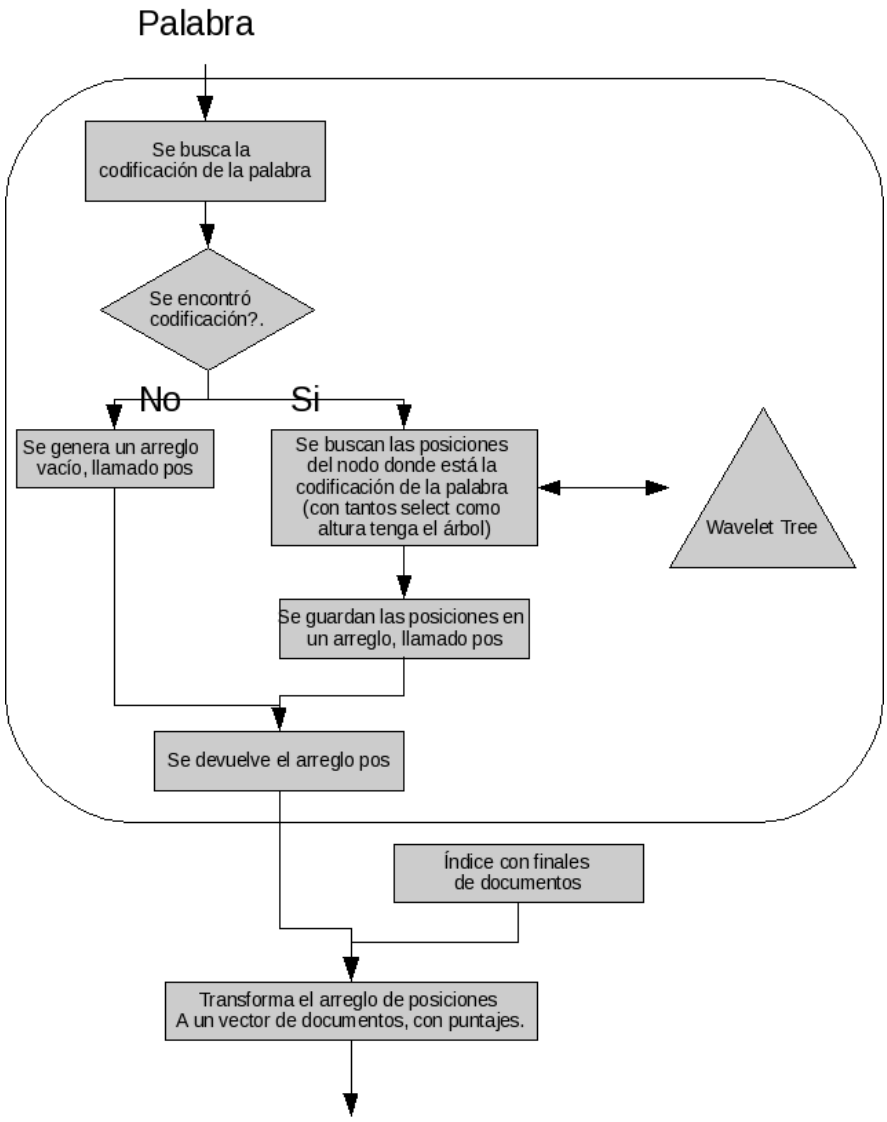

Figura 4.7: Pasos de la construcción del índice Wavelet Tree.

Cabe destacar que el árbol que se obtiene no es un árbol balanceado, tal como sí lo es un Wavelet Tree normal. Esto se debe a que en nuestro caso las palabras tienen codificaciones de largo distinto, debido a los códigos de Huffman. Dado que que las palabras más frecuentes tendrán una codificación más corta, estarán en las hojas más cercanas a la raíz que otras palabras que no sean tan frecuentes.

4.3.3. Búsqueda

Para la búsqueda en el Wavelet Tree, se implementó la misma interfaz, que en el índice invertido y los pasos que realiza ésta son mostrados en la Figura 4.8.

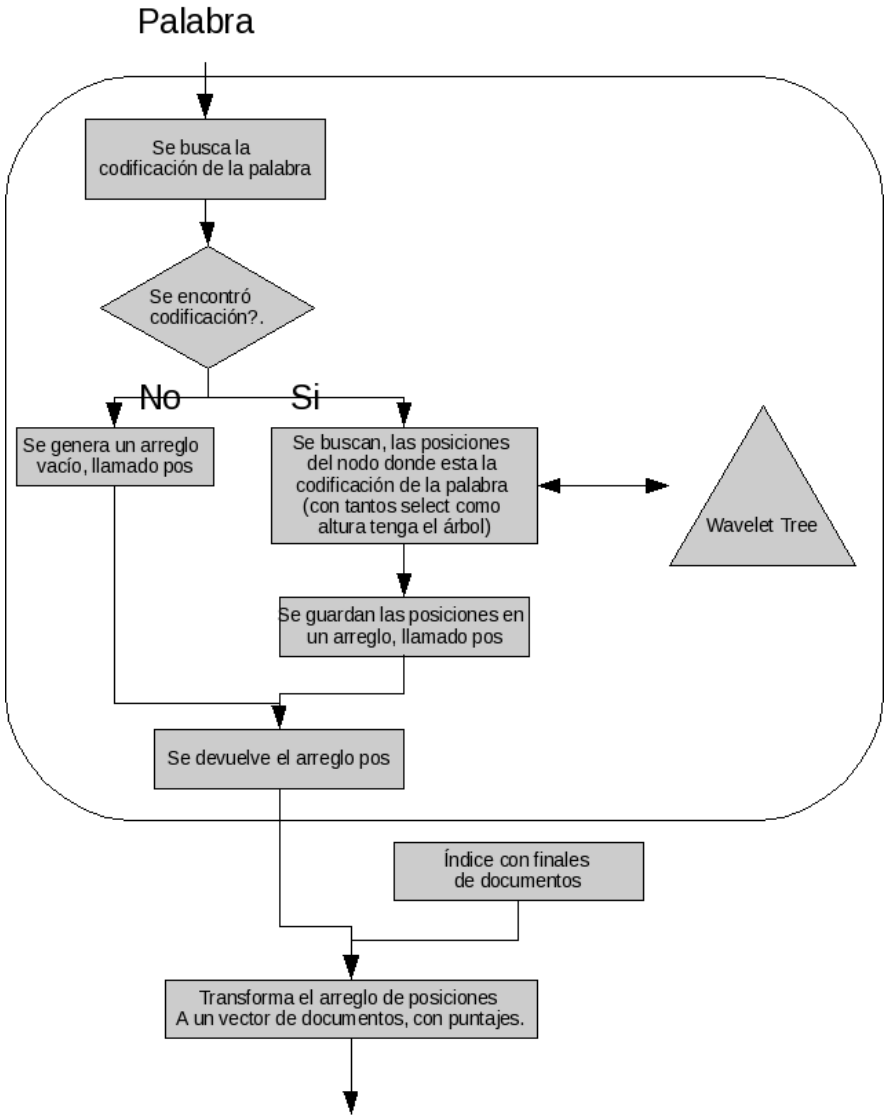

Figura 4.8: Esquema de búsqueda de documentos en el Wavelet Tree.

Uno de los desafíos que se encontró fue el de transformar el arreglo de posiciones a un vector con documentos y sus puntajes, para el proceso de ordenar los documentos de acuerdo a su importancia. Para resolver el problema se aprovechó que el Wavelet Tree responde las posiciones en orden, por lo que que se usó el índice que muestra los finales de documentos y se recorre buscando si la posición de ese final es mayor o menor que la posición devuelta por el Wavelet Tree. Una vez que se encuentra el primer documento de la primera posición, se realiza el mismo procedimiento pero en vez de comenzar de la posición 0 del índice se comienza desde la posición del último documento encontrado. Es aquí donde se calcula el puntaje del documento, ya sea por frecuencia o por las posiciones de las palabras dentro de éste. En la Figura 4.9 se muestra un ejemplo de cómo se realiza la búsqueda de los documentos.

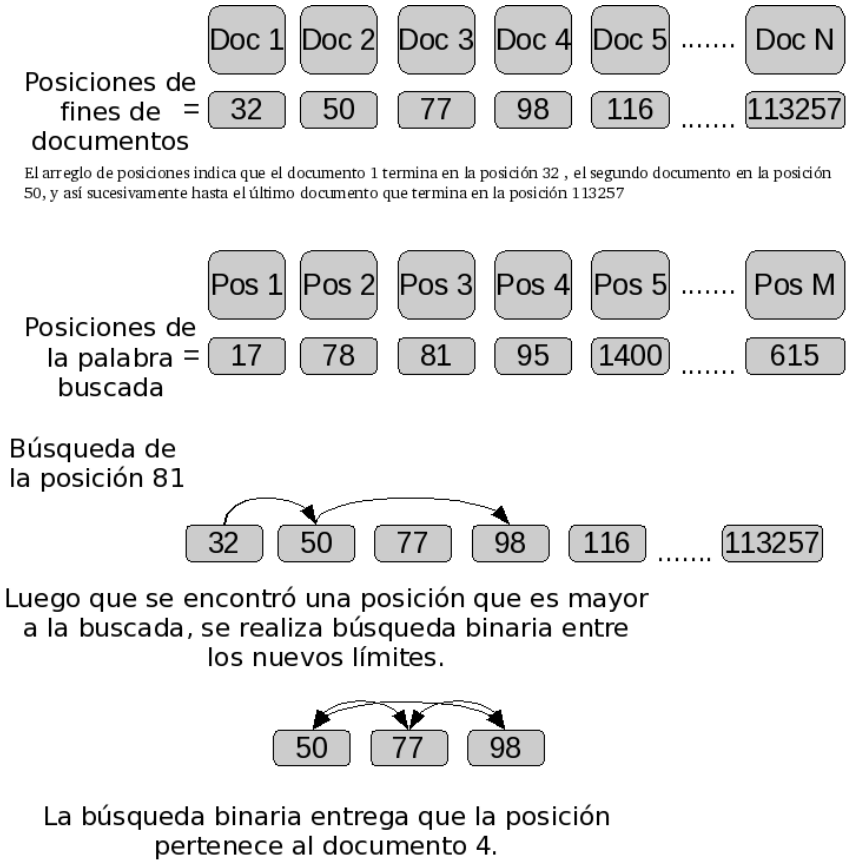

Figura 4.9: Diagrama de la búsqueda de los documentos dada las posiciones.

Luego que se tienen un vector con los documentos y sus puntajes, éste es devuelto al flujo normal de la máquina de búsqueda, como se muestra en la Figura 4.1

Como se puede apreciar en esta sección, el costo de búsqueda del Wavelet Tree se ve afectado por las siguientes condiciones:

1. Altura de la hoja de la codificación de la palabra a buscar.

2. Transformación del arreglo de posiciones a un vector de documentos.

3. Calcular la función de puntaje.

El costo del punto 1 depende del largo de la codificación de Huffman de la palabra, multiplicado por el costo del select. El transformar el arreglo en vector es logarítmico por el tamaño del vector de salida y el calculo del puntaje es proporcional al tamaño del vector de salida, produciendo que la búsqueda sea mucho más costosa que en el índice invertido puro.

Capítulo 5

Experimentos

En el presente capítulo se comparan experimentalmente las estrategias de índice invertido y Wavelet Tree.

5.1. Objetivos de los experimentos

Los experimentos del presente trabajo comparan las alternativas de índice invertido y Wavelet Tree, en el escenario de un motor de búsqueda. Las comparaciones son tanto en el espacio ocupado por cada una de las alternativas como en el tiempo para responder consultas, para observar cual es la más eficiente en distintos escenarios.

5.2. Diseño de experimentos

Se realiza una comparación de las estrategias en cuanto al espacio de memoria principal usada y la cantidad de consultas por segundo que son capaces de responder. Puesto que se está representando una MBW, se trata de maximizar la cantidad de consultas por segundo que es capaz de resolver. También se estudia la escalabilidad de las estrategias con respecto al número de procesadores que se están empleando, así como el tamaño de la colección de texto utilizada.

En la siguiente sección se describen los experimentos a realizar y los parámetros con los cuales se realizan.

5.2.1. Configuración de los experimentos

5.2.1.1. Índices a comparar

Se utilizan las siguientes instancias para cada estrategia:

Para la estrategia del autoíndice comprimido se utiliza un Wavelet Tree orientado a la palabra y comprimido con Huffman.

Para la estrategia de índice invertido se usará:

- Un índice invertido normal, sin comprimir, el cual está comprendido por una tabla de palabras y cada palabra tiene una lista de documentos con la frecuencia de la palabra dentro de cada uno de ellos, tanto las identificadores de documento como las frecuencias que se representan como enteros sin comprimir. Esta lista de documentos está ordenada por el identificador de éste, como se puede ver en la Figura 5.1.

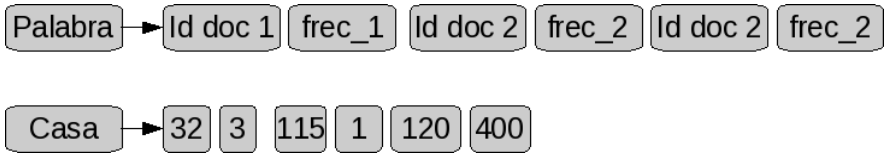

Figura 5.1: Ejemplo de un índice invertido descomprimido.

- Un índice invertido compactado: el cual consiste en una tabla de palabras y cada palabra tiene una lista de documentos con la frecuencia de la palabra dentro de cada uno de ellos. Esta lista de documentos está ordenada por el identificador de éste. Sin embargo en vez de guardar los identificadores de los documentos en cada posición, se almacena la diferencia entre los identificadores consecutivos. Un ejemplo con la lista de documentos de la Figura 5.1 se muestra en la Figura 5.2.

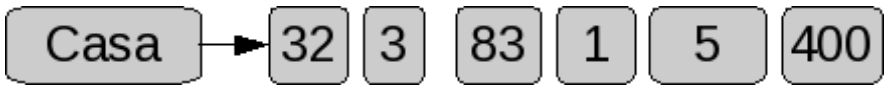

Figura 5.2: Ejemplo de una lista de diferencias de id de documentos.

Luego esas diferencias en la lista se codifican como un entero, pero empleando la cantidad exacta de bits necesaria para representar la máxima diferencia dentro de la lista. Lo mismo ocurre con las frecuencias. De esta manera, a cada palabra de la tabla de palabras se le asocia un arreglo de bits que contiene la lista de diferencias de documentos y las frecuencias codificadas en un menor número de bits, quedando como se muestra en la Figura 5.3.

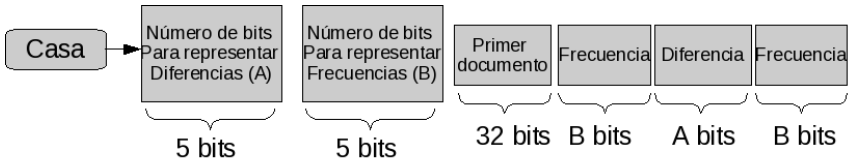

Figura 5.3: Ejemplo de una lista de diferencias codificadas.

- Un índice invertido comprimido: el cual consiste en una tabla de palabras y cada palabra tiene una lista de documentos con la frecuencia de la palabra dentro de cada uno de ellos. Esta lista de documentos está ordenada por el identificador de éste. Luego en

vez de guardar los identificadores de los documentos en cada posición, se guarda la diferencia con el identificador anterior como se muestra en la Figura 5.2. Luego, esa lista de diferencias y frecuencias se comprime usando la codificación de Golomb [16], quedando una sola lista de bits.

5.2.1.2. Hardware

El hardware en donde se ejecutaran los experimentos será:

- Cpu: x86_64 AMD Opteron(tm) Processor 275 AuthenticAMD.
- Ram: 5 GB.
- Sistema Operativo : Gentoo linux.[1].
- kernel: 2.6.25-gentoo-r7.

5.2.1.3. Datos empleados

La colección de documentos usada en las pruebas es una muestra de la Web de UK, extraída de una de las colectas de YAHOO! UK de marzo del 2006. Desde esta muestra se seleccionaron al azar un total de 10.000.000 de documentos, de los cuales para cada experimento se utilizó un máximo de 1.080.000 documentos por procesador.

El log de consultas con el que se trabajó contiene 36.389.576 consultas reales hechas por lo usuarios de YAHOO! UK entre el 01/03/2006 y el 01/06/2006, de las cuales:

- 9.900.102 son consultas únicas.
- 7.236.931 son consultas de una palabra.
- 11.329.505 son consultas de dos palabras.
- 9.314.324 son consultas de tres palabras.
- 8.505.341 son consultas de cuatro o más palabras.

5.2.1.4. Formato de los datos

La Web y el log de consultas están representados de la siguiente forma:

- Web: Está distribuida en archivos de texto en donde el contenido de cada documento (sólo el texto) está contenido en una sola línea, por lo que la cantidad de documentos en cada archivo es la cantidad de líneas que éste contiene.
- Consultas: Están distribuidas en un archivo en donde cada línea corresponde a una consulta. Las líneas están ordenadas por el tiempo en que la consulta fue realizada, lo cual permite reproducir una secuencia real de consultas hechas por los usuarios.

[1] http://www.gentoo.org/

- Cada procesador almacena un archivo que representa la Web y uno que representa las consultas. Debido a que se usa partición por documentos, cada procesador procesa la mismas consultas pero éstas son ejecutadas sobre un índice y colección de texto distinta como se muestra en la Figura 3.1.

5.2.2. Tamaño del índice con respecto al número de documentos

Es importante estudiar la diferencia en la constante de crecimiento en el tamaño de los índices. Esto se probó construyendo los índices con distintos números de documentos y observando el espacio ocupado en RAM.

Las cantidades son de:

- 20.000, 40.000, 80.000, 100.000, 150.000, 200.000, 250.000, 300.000, 321.000 documentos.

5.2.3. Promedio de consultas que se responden por segundo

Como se están comparando las estrategias en el contexto de los requerimientos para máquinas de búsqueda para la Web, es importante estudiar la cantidad de consultas que el índice es capaz de responder por segundo. Para esto, las dos estrategias responden las mismas consultas (tomadas de nuestro log) sobre una misma muestra de la Web, con distintas configuraciones para ver la escalabilidad de las estrategias.

5.2.3.1. Configuraciones de los experimentos realizados

Realizamos pruebas para estudiar cómo escala la eficiencia de los índices para responder consultas, con respecto a:

- El número de documentos de la Web, en una sola máquina.
- La cantidad de máquinas involucradas, con 270.000 documentos.

Esto implica que los experimentos muestran cómo varía el promedio de respuestas por segundo que entregan las dos estrategias, con una Web de:

- 20.000, 80.000 y 200.000 documentos.

Para la segunda parte, se muestra cómo varía el promedio de respuestas por segundo para distintas cantidades de máquinas involucradas en la búsqueda, con una Web de 270.000 documentos. Las distintas configuraciones son:

- 1 máquina con 270.000 documentos.
- 2 máquinas, cada una con 135.000 documentos.
- 4 máquinas, cada una con 67.000 documentos.

Por último se realizaron experimentos aumentando el tamaño de la Web y las máquinas, donde se tiene:

- 1 máquina con 270.000 documentos en total.
- 2 máquinas, cada una con 270.000 documentos, con 540.000 documentos en total.
- 4 máquinas, cada una con 270.000 documentos, con 1.080.000 documentos en total.

Para todas las configuraciones anteriores, se trabajó con 400.000 consultas extraídas de nuestro log de consultas.

5.3. Resultados

5.3.1. Tamaño del índice con respecto al número de documentos

En la Figura 5.4 se muestra el espacio que ocupan los índices en la RAM, dada una Web con un número determinado de documentos.

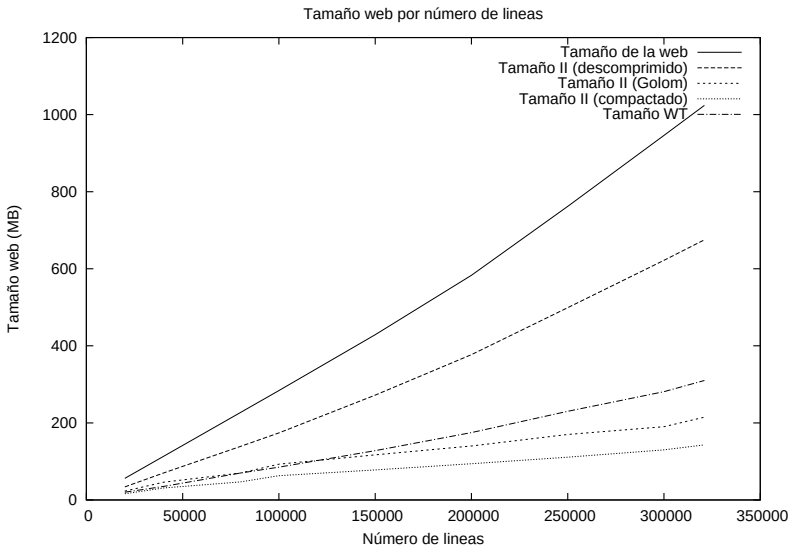

Figura 5.4: Tamaño de los índices con respecto al número de documentos.

5.3.2. Promedio de consultas que se responden por segundo

5.3.2.1. Variando el tamaño de la Web

Los resultados obtenidos se muestran en el Cuadro 5.1.

Documentos en la Web	II (Descomprimido)	II (Golomb)	II (Compactado)
20.000	42.552	95	156
80.000	4.799	22	40
200.000	1.413	11	17

Documentos en la Web	Wavelet Tree
20.000	76
80.000	30
200.000	15

Cuadro 5.1: Consultas por segundo que responden las estrategias para distintas cantidades de documentos.

5.3.2.2. Variando el número de procesadores

Los resultados obtenidos se muestran en el Cuadro 5.2.

Procesadores	Documentos por máquina	II (Descomprimido)	II (Golomb)	II (Compactado)
1	270.000	998	10	15
2	135.000	1.025	12	19
4	66.000	1.040	15	25

Procesadores	Documentos por máquina	Wavelet Tree puro
1	270.000	9
2	135.000	10
4	66.000	11

Cuadro 5.2: Consultas por segundo que responden las estrategias para distintas cantidades máquinas.

5.3.2.3. Variando el número de máquinas y el tamaño de la Web

Los resultados obtenidos se muestran en el Cuadro 5.3.

Procesadores	Documentos en total	II (Descomprimido)	II (Golomb)	II (Compactado)
1	270.000	998	10	15
2	540.000	503	6	9
4	1.080.000	257	3	5

Procesadores	Documentos en total	Wavelet Tree puro
1	270.000	9
2	540.000	5
4	1.080.000	3

Cuadro 5.3: Consultas por segundo que responden las estrategias para distintas cantidades de procesadores.

5.4. Discusión

Como se muestra en la Figura 5.4, si bien el índice invertido comprimido y el compactado requieren menos espacio que el Wavelet Tree, la cantidad de consultas que responden por segundo es muy pequeña. Esto no es aceptable en una máquina de búsqueda puesto que uno de los principios básicos es que el hardware instalado sea capaz de entregar la mejor tasa de consultas resueltas por segundo que sea posible. Las implicancias de una tasa deficiente en el costo de mantención y consumo de energía son relevantes.

Como se puede apreciar en la Figura 5.4 el índice invertido descomprimido, es decir sin comprimir o compactar, tiene un tamaño mayor que el Wavelet Tree. Sin embargo, los índices invertidos compactados y comprimidos tienen un tamaño menor al del Wavelet Tree. Si bien el índice invertido descomprimido responde muchas más consultas por segundo que las demás alternativas, su tamaño es mucho mayor también. Los Cuadros 5.1, 5.2 y 5.3 muestran que los índices comprimidos y compactados responden una cantidad de consultas comparable con el Wavelet Tree, por lo que la pregunta es si el Wavelet Tree es una buena alternativa. La discusión de las tres ultimas alternativas se refleja en la Figura 5.5.

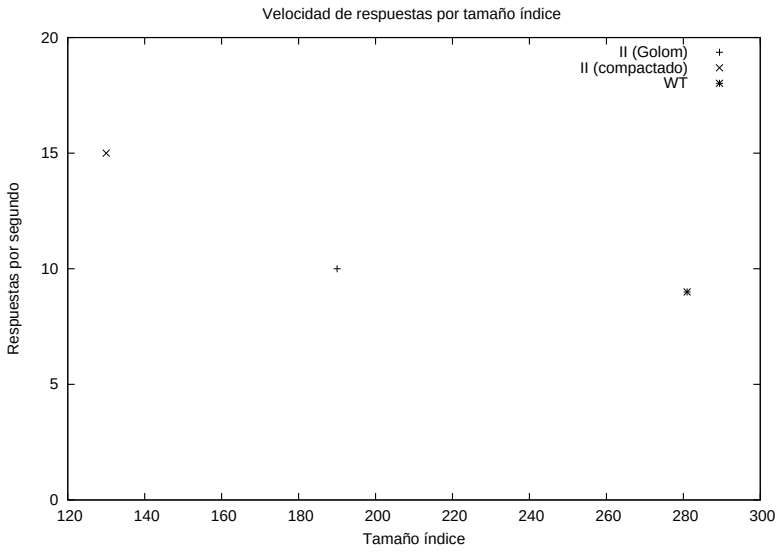

Figura 5.5: Muestra la velocidad de respuesta comparado con el tamaño del índice.

Claramente el gráfico anterior muestra que las mejores alternativas son los índices comprimido y compactado, pues ocupan menos espacio y responden un poco más de consultas por segundo. Pero el Wavelet Tree contiene el texto, el cual puede ser consultado y también extraído para formar los snippets, característica de la cual carecen los índices invertidos. Es por eso que debemos sumar el tamaño del texto a los índices. Para el cálculo de cuanto se debe sumar, se determinó el tamaño del texto comprimido con Huffman orientado a la palabra (el cual fue de 180 megas). Actualizando los datos obtenemos un nuevo gráfico que se muestra en la Figura 5.6.

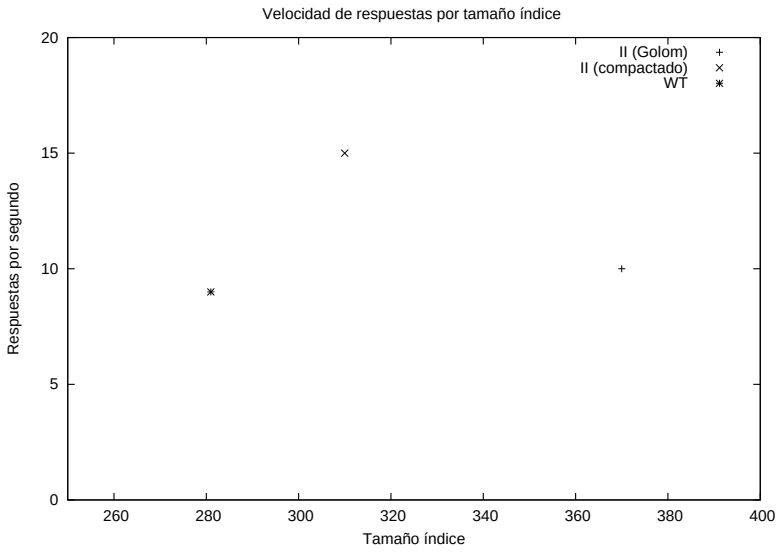

Figura 5.6: Muestra la velocidad de respuesta comparado con el tamaño del índice.

Allí se puede ver, que con la inclusión del texto el Wavelet Tree es ahora el mejor representante en tamaño.

Si bien los índices invertidos comprimidos constituyen una interesante alternativa, su tasa de respuestas por unidad de tiempo es baja, y para una MBW la velocidad es una de más importantes características, es por eso que para los siguientes experimentos se escogió el índice invertido descomprimido como representante de los índices invertidos, por tener la mayor velocidad de consultas, para comparar con el Wavelet Tree que ocupa el menor espacio.

En los Cuadros 5.1, 5.2, 5.3 se puede ver que el índice invertido descomprimido es capaz de responder más consultas por segundo que el Wavelet Tree, lo cual se debe a que almacena, para cada término del vocabulario, los documentos y sus puntajes precalculados. Sin embargo, el índice invertido descomprimido ocupa un espacio de memoria mucho más grande que el Wavelet Tree, como se puede apreciar en la Figura 5.1. Además la característica de precalcular el ranking hace que si se quiere cambiar la función de ranking deba precalcularse todo el índice o guardar más información para poder ocupar otra función, lo que lo haría aún más costoso en espacio. No es difícil imaginar que algunos MBW utilizan varias funciones de ranking las cuales son aplicadas en conjunto o separadamente dependiendo del contenido de la consulta.

Como se puede observar en los Cuadros 5.1, 5.2, 5.3 y la Figura 5.1 si bien el tamaño del Wavelet Tree en RAM es mucho menor, y crece a una tasa más lenta con respecto al número de documentos, éste responde muy pocas consultas por segundo en comparación con el índice invertido descomprimido, lo que se debe a que para extraer los documentos primero se deben encontrar todas las ocurrencias de la palabra en el texto, lo cual es costoso en tiempo de ejecución, pues por cada consulta debe realizarse todo el trabajo que en el índice invertido ya se tiene precalculado.

Si bien técnicas de reducción de vocabulario [10,11], técnicas de compresión y técnicas de compactación pueden hacer que el índice invertido se reduzca en espacio, la fortaleza del Wavelet Tree está en ocupar menos espacio para mantener tanto el índice como el texto en memoria principal. Si bien el tener que calcular los puntajes al vuelo es el otro factor importante que produce la baja cantidad de respuestas por segundo, es una ventaja con respecto al índice invertido en ciertas aplicaciones, pues permite generar dinámicamente un puntaje para las búsquedas. Por ejemplo, esto tiene aplicaciones en casos en los que se debe realizar ranking con perfiles de usuario, o con características variables sin tener que guardar más información. Existen trabajos [17,18] que presentan la problemática de tener que variar o cambiar dinámicamente la función de ranking para mejorar los resultados. Es por eso que la característica del Wavelet Tree de calcular al vuelo la función de ranking se vuelve interesante, permitiendo implementar varias estrategias de ranking sin aumentar el espacio ocupado. Además cambiar de función de ranking no afecta en sí a la estructura del índice, por lo que no hay que regenerar nada de ésta.

5.5. Conclusiones

Después de evaluar experimentalmente los métodos, se puede concluir que el trabajo realizado por el preproceso en el índice invertido es beneficioso en tiempo de búsqueda. Este resultado es evidente pero los experimentos muestran la magnitud de este beneficio. Pero esto le juega en contra a la flexibilidad de su ranking en un ambiente en constante cambio de la función de ranking, pues para cada cambio en esta función hay que reconstruir los índices o es necesario almacenar más información pre-calculada lo que reduce la eficiencia del espacio ocupado. Por otro lado, si bien el Wavelet Tree es menos eficiente en tiempo de búsqueda, pues debe hacer cálculos adicionales al procesar cada consulta, esa característica juega un papel bastante importante, ya que nos permite realizar distintos tipos de ranking según sea conveniente, dado que los puntajes no necesitan fijarse al momento de construir el índice. Además, otra ventaja del Wavelet Tree es su capacidad de re-construir el texto a partir del índice mismo, por lo que se pueden extraer snippets del texto sin tener que consultar a otra máquina o acceder a memoria secundaria, y esto a un costo comparable al de la búsqueda de posiciones en el índice. Esta es una ventaja importante que permite reducir el espacio de almacenamiento y reducir costos, pues no se necesitan servidores para almacenar el texto, y además se reduce el costo de comunicación entre procesadores del cluster.

Capítulo 6

Método híbrido

Como se puede apreciar, las dos estrategias (Wavelet Tree e índice invertido) conllevan ventajas y desventajas. Por lo tanto es interesante estudiar un esquema que combine esas características, para obtener así una mejor alternativa. En este capítulo se propone un tercer método, el cual combina las características de los métodos mencionados para explotar sus ventajas y reducir el efecto de sus desventajas.

6.1. Diseño

El método híbrido consiste en usar el Wavelet Tree para generar, durante el procesamiento de las consultas, el índice invertido para los términos maś frecuentemente referenciados por las consultas activas en un periodo del tiempo. El objetivo es amortizar el costo de consultar el Wavelet Tree ya que las consultas subsecuentes para el mismo término serán respondidas usando esa información precomputada. Esto es posible dado que en las consultas de usuarios reales siempre existe una gran repitencia de términos. Algunos términos se repiten a lo largo de periodos muy largos de tiempo mientras que otros son populares en periodos cortos del tiempo. En cualquiera de los dos casos las listas invertidas de los términos involucrados tienden a ser referenciadas varias veces lo cual favorece que dichas listas invertidas sean mantenidas en un cache de propósito específico. En este cache se almacena el índice invertido generado dinámicamente por las consultas y el cache tiene una capacidad limitada. De esta manera se va generando un índice que contiene sólo con las palabras más frecuentemente buscadas, sin desperdiciar espacio con palabras que rara vez se consultan. Este esquema se muestra en la Figura 6.1.

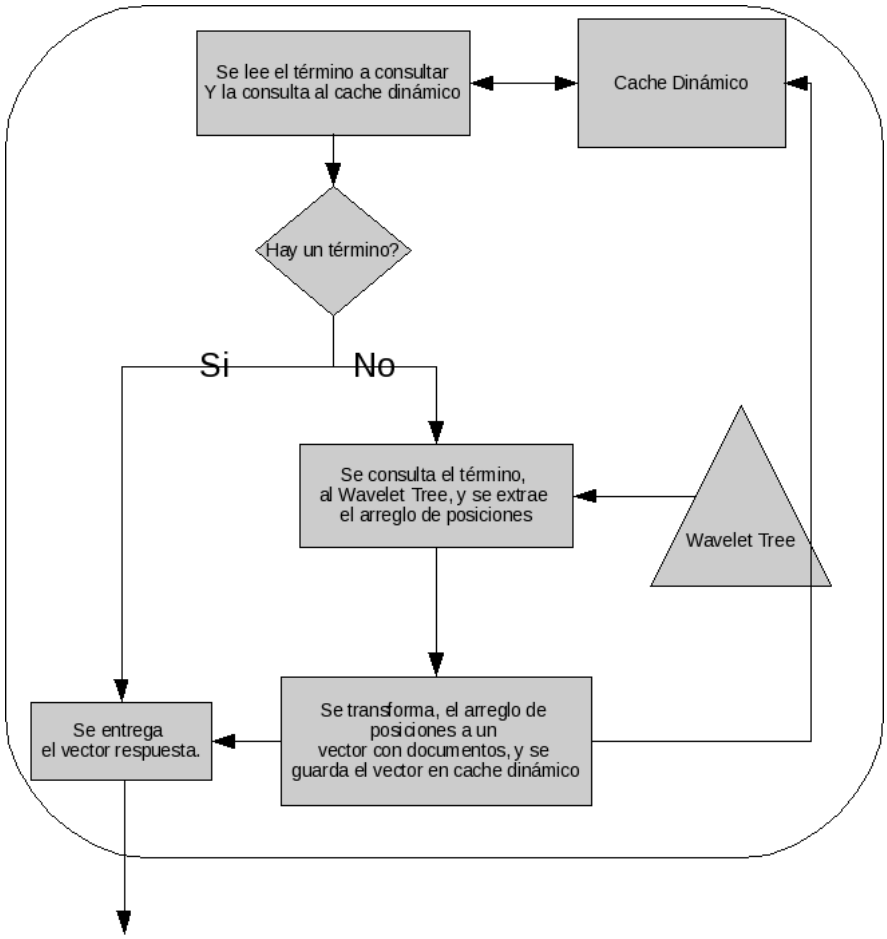

Figura 6.1: Diseño del método híbrido original.

El cache de listas invertidas está compuesto por una tabla que indica las palabras que lo componen y una lista que contiene los documentos asociados a estas palabras. Cada palabra tiene un puntero a un nodo de la lista para su fácil acceso y, cada vez que se consulta una palabra, el nodo al cual hace referencia se posiciona en la cabeza de la lista (con lo que el término adquiere más relevancia en la lista). Cada vez que se agrega una palabra con su lista invertida al cache, se añade la palabra a la tabla de palabras y se transforma en la cabeza de la lista. Esto emula de manera eficiente la política LRU de reemplazo de páginas en un cache. Para su eficiencia, el cache tiene un límite de palabras que puede almacenar. Cuando se alcanza este límite, se elimina el nodo que se encuentra al final de la lista (correspondiente a un término poco consultado), y se elimina la palabra de la tabla de palabras. De esta manera en el cache de listas invertidas están las palabras más recientemente consultadas y las que han consultado más veces, siendo eficiente en espacio al no albergar a palabras poco consultadas, como un esquema LRU[15].

Esto constituye un cache dinámico de listas invertidas, lo cual se puede complementar con cache estático que contiene las listas invertidas de los términos que son populares en todo momento. Tal como se discute más adelante en este capítulo, esta información se puede extraer desde el log de consultas de usuarios y la combinación de ambos tipos de memorias

cache puede mejorar significativamente la tasa de consultas resueltas por unidad de tiempo del buscador.

6.1.1. Respuestas por segundo

Las cantidad de respuestas por segundo que es capaz de responder el nuevo esquema se muestra en la Figura 6.2.

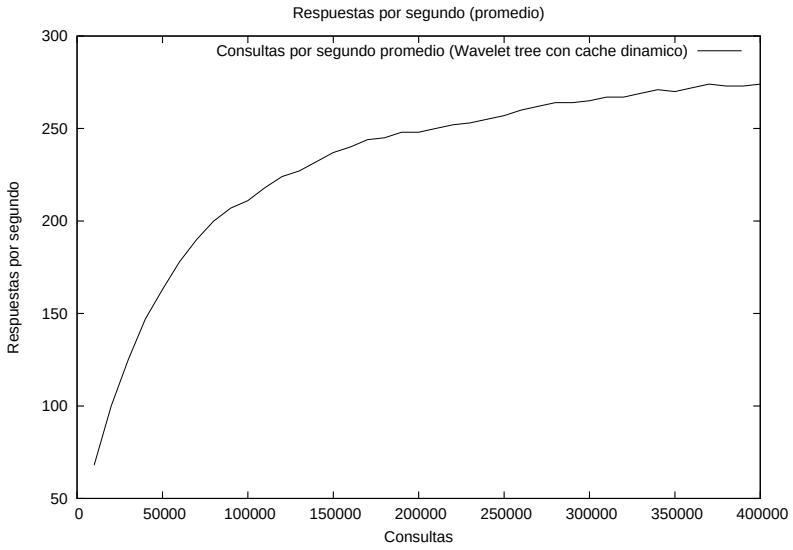

Figura 6.2: Consultas por segundo que responde el esquema híbrido(foto referencial).

Como se puede apreciar en la Figura 6.2, a medida que el cache dinámico se va llenando se comienza a responder cada vez más rápido. Sin embargo, el tiempo que le toma llegar a un régimen estable de respuestas es bastante grande.

6.1.2. Una mejora al método

Un análisis de las consultas en nuestro log mostró una característica bastante particular, la cual es que dentro de éstas existen términos que se repiten durante un intervalo de tiempo no muy grande, y luego ya no se repiten más, lo cual hace que el cache dinámico sea bastante útil. Sin embargo, existen términos que se repiten durante todo el periodo abarcado por el log de consultas, produciendo que siempre estén en el cache dinámico, acortando el espacio de éste para los términos que van variando.

Es por eso que se propone una mejora del método, la cual consta de utilizar entrenamiento y un cache estático de términos. Si hay términos que se preguntan casi siempre, estos se incluyen en un cache estático, quedando el diseño como se muestra en la Figura 6.3. Este

cache tiene la ventaja de que es más fácil de administrar, además es más eficiente pues puede colocarse en memoria de sólo lectura ya que no se modifica, permitiendo que varios threads lo consulten al mismo tiempo. Los términos que se agregan al cache estático son extraídos en nuestro caso del estudio de un trozo del log de consulta representativo de las tendencias de los usuarios.

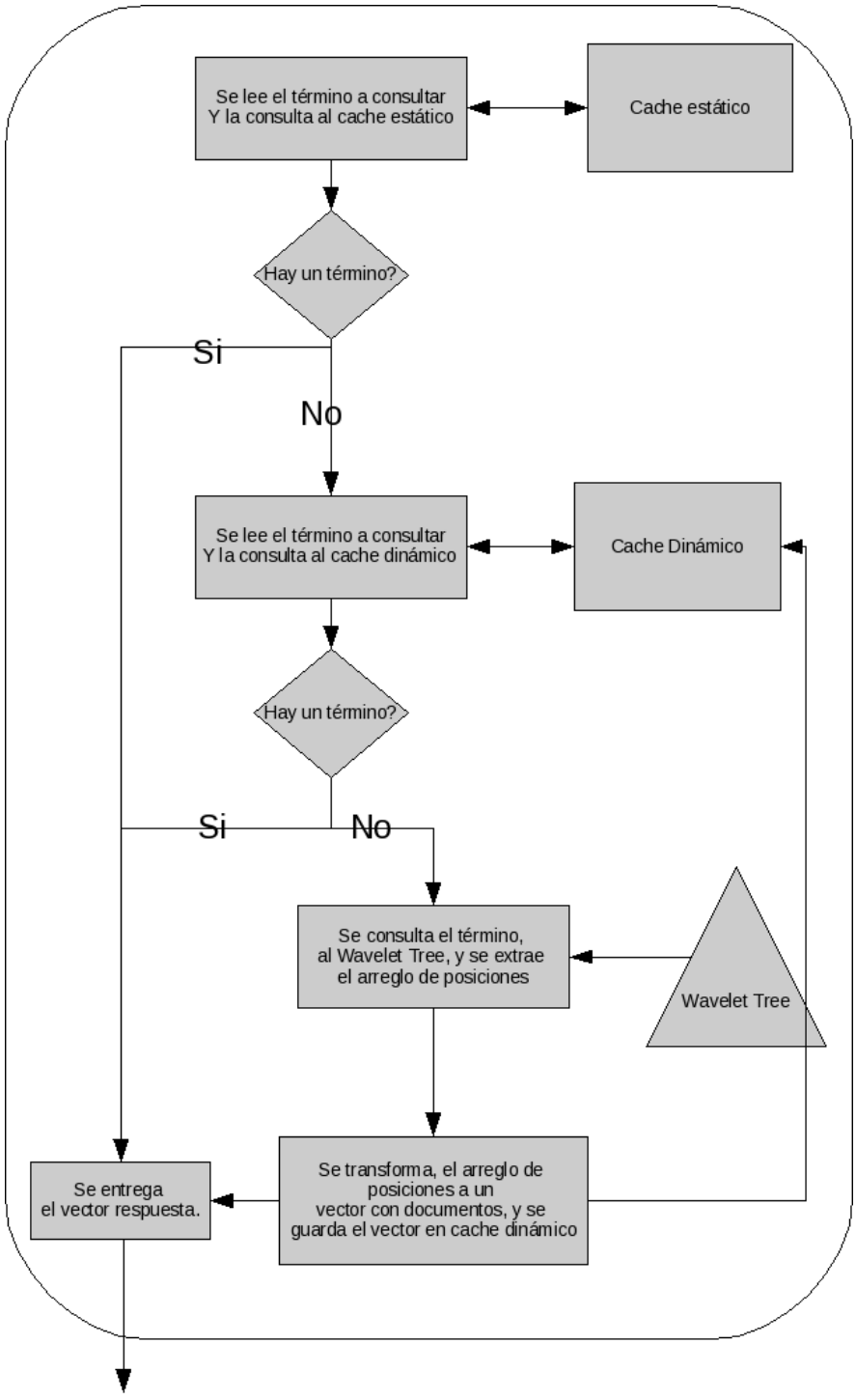

Figura 6.3: Diseño del método híbrido final.

6.2. Experimentación

Dentro de la experimentación del método híbrido, se realizaron las pruebas de tamaño en RAM con distintas configuraciones y consultas por segundo que puede responder el esquema. Como se tienen dos caches, se recurre al estado del arte para definir una configuración válida para estos. Es por eso que para los experimentos de este capítulo se ocupa una proporción de cache estático/dinámico entre aproximadamente 70 %/30 % y 80 %/20 %, la cual es una configuración bastante aceptada en la literatura [19]. No es el objetivo de este trabajo el estudio de otras configuraciones para los caches. No obstante, los experimentos sobre nuestros datos validan esta elección como la que conduce al mejor desempeño de nuestro sistema.

6.2.1. Configuración de sistema

La configuración del sistema y de los experimentos es la misma que se muestra en la Sección 5.2.3.1

6.2.2. Tamaños para distintas configuraciones

En esta sección se estudian los distintos tamaños que alcanza el esquema híbrido, dependiendo de las configuraciones entre cache estático y dinámico. Para que los tamaños sean reales, se toma el promedio de los tamaños desde que el cache dinámico está lleno. Estos experimentos fueron realizados mientras las distintas configuraciones del método híbrido estaban respondiendo consultas, en un único procesador con una Web de 270.000 documentos y un log de 9.000.000 de consultas.

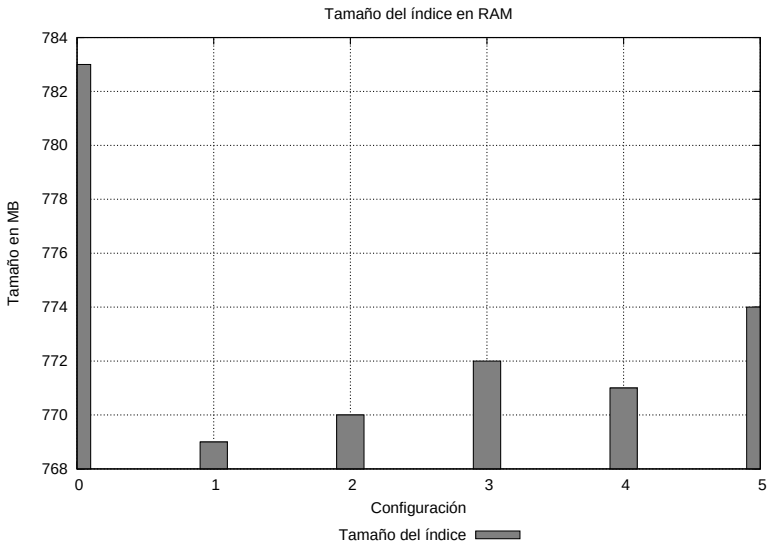

Figura 6.4: Tamaño en RAM para distintas configuraciones del método híbrido comparado con el índice invertido.

En la Figura 6.4 se muestra el tamaño resultante para las siguientes configuraciones:

- "0" corresponde al tamaño del índice invertido puro, que contiene las listas de todas las palabras existentes en el texto.
- "1" corresponde a la configuración 52.000 términos en el cache estático y 13.000 términos en el cache dinámico.
- "2" corresponde a la configuración 56.000 términos en el cache estático y 14.000 términos en el cache dinámico.
- "3" corresponde a la configuración 60.000 términos en el cache estático y 15.000 términos en el cache dinámico.
- "4" corresponde a la configuración 70.000 términos en el cache estático y 10.000 términos en el cache dinámico.
- "5" corresponde a la configuración 80.000 términos en el cache estático y 10.000 términos en el cache dinámico.

Como se puede observar en la Figura 6.4, si bien los tamaños en RAM son muy parecidos todas las configuraciones del esquema híbrido tienen un tamaño menor al del índice invertido.

6.2.3. Consultas por segundo que es capaz de responder

En esta sección se muestra la cantidad de consultas que puede responder el esquema híbrido, con distintas configuraciones, en intervalos de 10.000 consultas, con 9.000.000 de consultas y una Web de 270.000 documentos. Los resultados se aprecian en la Figura 6.5.

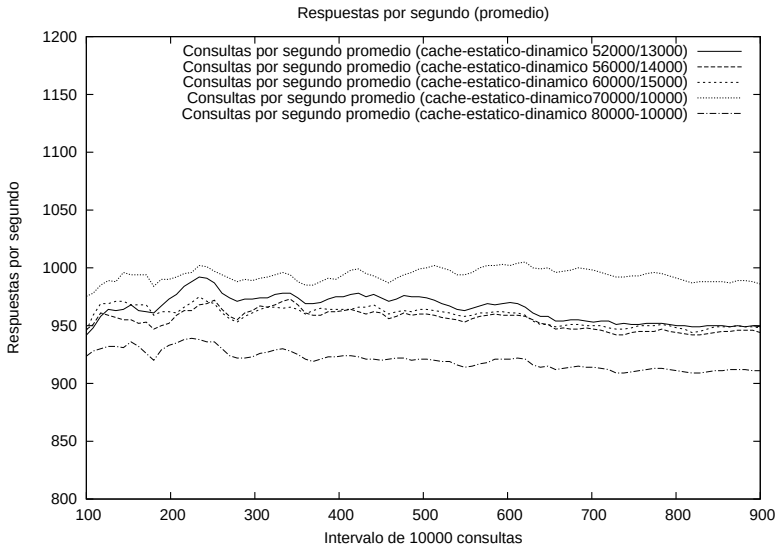

Figura 6.5: Consultas por segundo que puede responder el método híbrido.

Como se puede ver en la Figura 6.5, si bien todas las configuraciones tienen rendimientos parecidos,la configuración de cache estático con 70.000 términos y un cache dinámico con 10.000 términos es la que alcanza la mejor tasa de respuestas por segundo. Más adelante en este capítulo se establece una comparación entre este caso y el índice invertido descomprimido.

6.2.4. Escalabilidad

En esta sección se muestra cómo escala el esquema híbrido con respecto a la cantidad de documentos que conforman la muestra de la Web utilizada en los experimentos y el número de máquinas que componen el cluster.

Para las Secciones 6.2.4.1 y 6.2.4.2, se muestran los esquemas híbridos

1. Cache dinámico 2.000 términos y cache estático 6.000 términos.

2. Cache dinámico 3.000 términos y cache estático 9.000 términos.

Para la Sección 6.2.4.3, las configuraciones del esquema híbrido son:

1. Cache dinámico 10.000 términos y cache estático 70.000 términos.

2. Cache dinámico 10.000 términos y cache estático 80.000 términos.

Estas configuraciones fueron ocupadas para igualar el espacio ocupado por el índice invertido.

6.2.4.1. Variando la cantidad de documentos que componen la Web

Los resultados presentados en el Cuadro 6.1 muestran la cantidad promedio de consultas por segundo que responde el esquema híbrido para 1.000.000 de consultas con distintas configuraciones.

Cantidad de documentos	Dinámico 2.000 Estático 6.000	Dinámico 3.000 Estático 9.000
20.000	3.147	5.351
80.000	1.056	1.575
200.000	496	688

Cuadro 6.1: Consultas por segundo que responden las estrategias para distintas cantidades de documentos.

En la Figura 6.6 se ve una comparación gráfica del Cuadro 6.1.

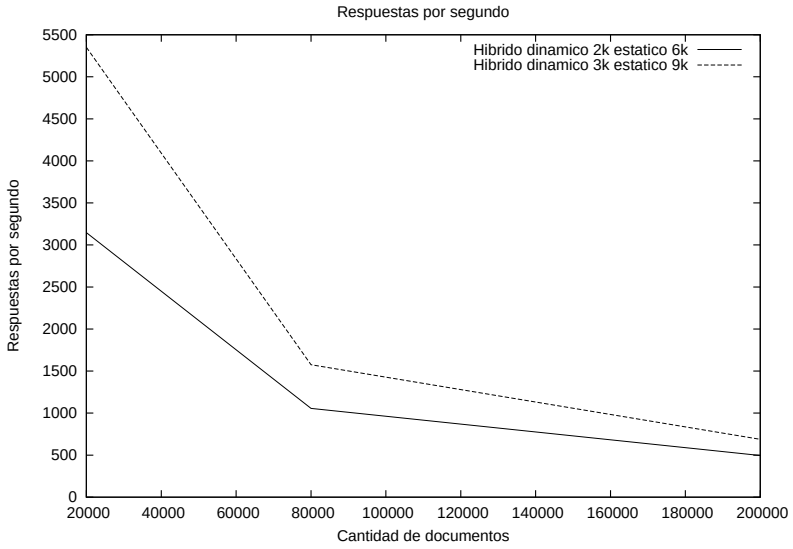

Figura 6.6: Escalabilidad del esquema híbrido con distintas configuraciones, para distintas cantidades de documentos en la Web.

Como se puede ver en el Cuadro 6.1, el comportamiento del esquema híbrido es muy parecido al del índice invertido, en relación a la cantidad de documentos que componen la Web.

6.2.4.2. Variando el número de máquinas

Los resultados obtenidos se muestran en el Cuadro 6.2.

Máquinas	Documentos en cada máquina	Dinámico 2.000 Estático 6.000	Dinámico 3.000 Estático 9.000
1	270.000	431	584
2	135.000	447	597
4	66.000	453	605

Cuadro 6.2: Consultas por segundo que responden las distintas configuraciones del método híbrido para distintas cantidades máquinas.

En la Figura 6.7 se ve una comparación gráfica del Cuadro 6.2.

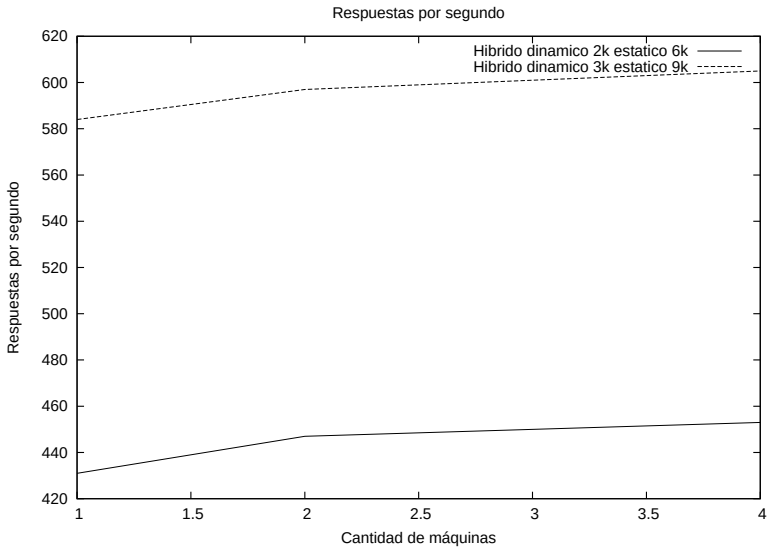

Figura 6.7: Comparación entre las distintas configuraciones del método híbrido, variando la cantidad de máquinas.

6.2.4.3. Variando el número de máquinas y el tamaño de la Web

Los resultados obtenidos se muestran en el Cuadro 6.3.

Máquinas	Documentos Totales	Dinámico 10.000 Estático 70.000	Dinámico 10.000 Estático 80.000
1	270.000	930	900
2	540.000	496	453
4	1080.000	251	230

Cuadro 6.3: Consultas por segundo que responden las distintas configuraciones del método híbrido para distintas cantidades máquinas y número de documentos.

En la Figura 6.8 se ve una comparación gráfica de las estrategias.

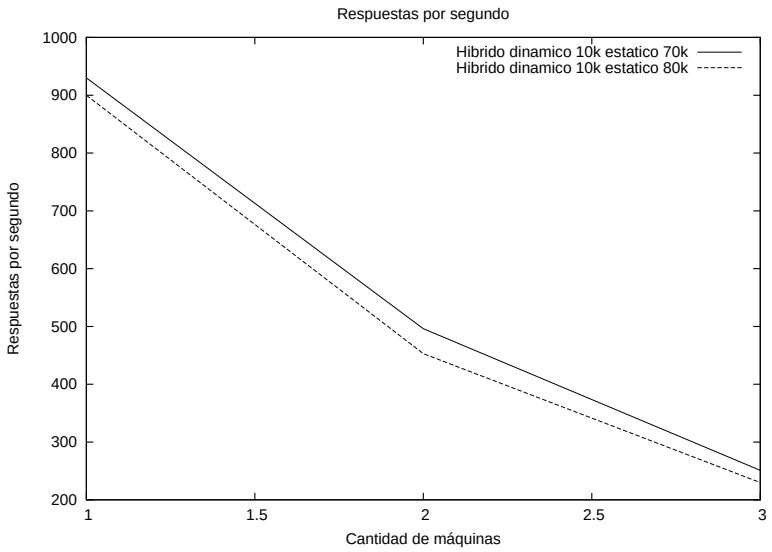

Figura 6.8: Comparación entre las distintas configuraciones del método híbrido, variando la cantidad de máquinas y el número de documentos.

Como se aprecia en las secciones 6.2.4.2 y 6.2.4.3, el comportamiento del esquema híbrido es similar al comportamiento del índice invertido en la escalabilidad de las máquinas y la variación de máquinas y número de documentos, por lo que se realizará ahora una comparación entre el esquema híbrido y el índice invertido en la característica más relevante para un MBW, es decir, la tasa de respuestas por segundo.

6.3. Comparación método híbrido e índice invertido

6.3.1. Consultas respondidas por segundo

Estos experimentos fueron realizados con la misma configuración de la Sección 6.2.3, haciendo que las distintas configuraciones del método híbrido ocupen el mismo espacio en

memoria RAM que el índice invertido.

Cabe destacar que el método híbrido además contiene el texto de toda la Web, pudiendo extraer los snippets, disminuyendo así el costo de almacenamiento y el de transferencia de texto.

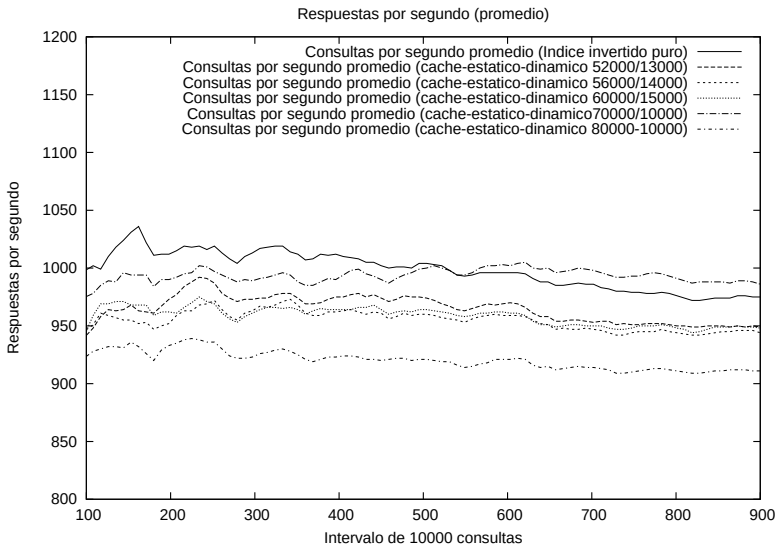

Figura 6.9: Consultas por segundo que pueden responder los métodos.

Como se puede apreciar en la Figura 6.9, el método híbrido responde casi la misma cantidad de consultas por segundo que el índice invertido, haciéndolos comparables pero enfatizando que en este punto, por cada consulta, el esquema híbrido es capaz de producir los snippets de los documentos desde el mismo espacio en memoria principal. Para producir los snippets, el índice invertido debe recurrir a espacio adicional para almacenar el texto en formato comprimido, texto que puede estar ubicado en memoria secundario o en la memoria principal de un servidor de documentos.

6.4. Discusión y comparaciones de los métodos

Como se puede apreciar en los experimentos, el método híbrido logra casi la misma eficiencia en responder consultas que el índice invertido, oscilando entre 960 y 1000 consultas por segundo, en comparación a las 980 y 1040 consultas por segundo que responde el índice invertido puro, usando similar espacio. Además, el método híbrido tiene la ventaja de contener al Wavelet Tree, pudiendo responder los trozos de texto que se necesitan sin necesidad de recurrir a otros medios para extraer los snippets, aumentando así la localidad y disminuyendo la comunicación en el servidor o los accesos a memoria secundaria. El método híbrido cuenta además con todas las ventajas del Wavelet Tree descritas en el Capítulo 5.

Capítulo 7

Conclusiones y trabajos futuros

7.1. Conclusiones de los experimentos

Como se aprecia en las discusiones de los capítulos anteriores, los dos métodos de indexación estudiados presentan ventajas y desventajas. Es por eso que se propone un método híbrido que aprovecha lo mejor de los dos métodos para responder consultas de una forma más eficiente que ambos métodos por separado. En el híbrido no es necesario re-hacer el índice para construir las listas invertidas cuando se modifica dinámicamente el método de ranking de documentos. Las listas se generan de manera on-line con el Wavelet Tree a la vez que se produce un índice invertido más eficiente en espacio puesto que se mantienen sólo las listas invertidas de los términos referenciados por las consultas en un periodo dado de tiempo.

7.2. Conclusión general

En este trabajo se implementó un prototipo de máquina de búsqueda de propósito general, la cual permite probar distintos tipos de estrategias, sin modificar el flujo de los datos. El objetivo de construir esta máquina de búsqueda es proporcionar un ambiente de pruebas para estudiar como se comportan las estrategias de búsqueda estudiadas en un ambiente similar a la realidad, sin tener que emular comportamientos de usuarios. En particular, el presente trabajo estuvo dedicado a comparar las estrategias de índice invertido y Wavelet Tree.

Si bien el Wavelet Tree mostró ser menos eficiente para responder consultas, este posee una alta flexibilidad al calcular on-line los documentos y sus puntajes. Sin embargo, el índice invertido descomprimido alcanza la mejor tasa de consultas resueltas por unidad de tiempo.

En base a los resultados de la comparación entre ambas estrategias, se desarrolló una técnica híbrida, la cual mejora las anteriores tanto en espacio como en flexibilidad, alcanzando una tasa similar de consultas resueltas por unidad de tiempo que el índice invertido, pero almacenando al mismo tiempo en memoria principal el texto comprimido auto-indexado desde donde extraer los snippets y permitiendo también cambiar dinámicamente el método de ranking dependiendo del contexto y contenido de las consultas.

7.3. Trabajos futuros

Dentro de los trabajos futuros que se pueden realizar se encuentran pruebas de otras técnicas de compresión para el Wavelet Tree, como "End-Tagged Dense Code", para aumentar la eficiencia de éste para responder consultas. Además, encontrar una manera adaptativa de que los términos más frecuentes del cache dinámico sean finalmente agregados en el cache estático, y eliminar términos del cache estático sin que esto afecte significativamente el rendimiento del esquema.

Otros trabajos que se proponen para el futuro son tratar de cambiar el esquema de particionado por documentos por esquemas particionados por términos, distribuyendo los términos entre las máquinas. De esta manera cada máquina tendrá toda la Web pero sólo con los términos que a ésta le corresponde, disminuyendo el tamaño del vocabulario por máquina y así aumentando tanto la compresión como disminuyendo la altura del Wavelet Tree lo cual permite optimizar las búsquedas.

Capítulo 8

Bibliografía

[1]: Brisaboa N. R., Cillero Y., Fariña A., Ladra S., Pedreira O. A New Approach for Document Indexing UsingWavelet Trees. A M. Tjoa and R.R. Wagner (Ed.). Proc. of the 18th International Workshop on Database and Expert Systems Applications (DEXA 2007), pp. 69-73. Regensburg, Germany, 2007.

[2]: Brisaboa N. R., Fariña A., Ladra S., Navarro G. Reorganizing compressed text. In Proc. of the 31th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'08), pp. 139-146. Singapore, 2008.

[3]: D. A. Huffman. A metod for the construction of minimum-redundancy codes. In Proc. Inst. Radio Eng., pages 1098– 1101, September 1952. Published as Proc. Inst. Radio Eng., volume 40, number 9.

[4]:M. Marin, G.V. Costa. "High-Performance Distributed Inverted Files", In ACM 16th Conference on Information and Knowledge Management (CIKM 2007), Lisbon, Portugal, Nov 6-9, 2007.

[5]: M. Marin and C. Gomez. Load Balancing Distributed Inverted Files, In ACM 9th International Workshop on Web Information and Data Management (WIDM 2007), held in conjunction with CIKM 2007, Nov. 6-9, 2007, Lisboa, Portugal.

[6]: Alistair Moffat, William Webber, and Justin Zobel. Load Balancing for Term-Distributed Parallel Retrieval. In SIGIR'06.

[7]: Jiangong Zhang, Torsten Suel. Optimized Inverted List Assignment in Distributed Search Engine Architectures. Polytechnic University Brooklyn.

[8]: Alistair Moffat. Word-based text compression. Software - Practice and Experience, 19(2):185–198, 1989.

[9]: Xiaolan Zhu, Susan Gauch. Incorporating quality metrics in centralized/distributed information retrieval on the World Wide Web. Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR'2000), pp 288 - 295. Athens, Greece, 2000.

[10]: Büttcher, S. and Clarke, C, A Document-Centric Approach to Static Index Prunning in Text Retrieval Systems, Proc. of CIKM, pp 182-189, 2006.

[11]: A. Soffer and D. Carmel and D. Cohen and R. Fagin and E. Farchi and M. Herscovici and Y. Maarek, Static Index Pruning for Information Retrieval Systems, In *Proc. 24th Annual International ACM SIGIR, Conference on Research and Development in Information*

Retrieval (SIGIR), pp 43-50, 2001.

[12]: R. BAeza-Yates ans B. Ribeiro-Neto. *Modern Information Retrival. 1999.*

[13]: J. Zobel and A. Moffat. Inverted files for text serach engines. ACM C. Surv.2006.

[14]: G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM C.Surv.* 2007.

[15]: Q. Gan and T. Suel. Improved techniques for result caching. *WWW*, 2009.

[16]: S. W. Golomb .Run-length encodings. *IEEE Trans Info Theory* ,1966.

[17]: Keke Chen, Ya Zhang, Zhaohui Zheng, Hongyuan Zha, Gordon Sun. Adapting ranking functions to user preference. *IEEE 24th International Conference on Data Engineering Workshop,* pp 580-587, 2008.

[18]: Weiguo Fan, Michael D. Gordon, Praveen Pathak, Discovery of Context-Specific Ranking Functions for Effective Information Retrieval Using Genetic Programmin, IEEE Transactions on Knowledge and Data Engineering, vol. 16, no. 4, pp. 523-527, Apr. 2004

[19]: X. Long, T. Suel. Three-Level Caching for Efficient Query Processing in Large Web Search Engines. *In 14th International World Wide Web Conference (WWW)*, 2005.

[20]:Roberto Grossi, Pisa Ankur Gupta, Jeffrey Scott Vitter. High-order entropy-compressed text indexes, *In Symposium on Discrete Algorithms (SODA)*,pp 670-678, 2003.

Capítulo 9

Apéndice

En el presente capítulo se muestra el código de las funciones de cache estático y dinámico que fueron ocupados para los experimentos.

9.1. Código índice invertido

```
class Inverted_index{
public:
    vector<string > *palabras;
    vector<vector<int> > *listas_de_documentos;
    int indice_actual;
    Inverted_index(int tam){
        palabras= new vector<string>(tam);
        listas_de_documentos= new vector<vector<int> >(tam);
        indice_actual=0;
    }
    ~Inverted_index(){
        palabras->clear();
        delete palabras;
        listas_de_documentos->clear();
        delete listas_de_documentos;
    }
    void agregar(string *_s, vector<int > *Documentos);
    vector <int> * busca(string *s);
};
```

Figura 9.1: Código en c++ de la clase Índice invertido.

9.2. Código Cache dinámico

```
#if !defined(_CACHE_H)
#define _CACHE_H
#include <string.h>
#include <vector>
#include <map>
#include "NodoLista.h"
using namespace std;
class Cache{
public:
	map<string,NodoLista *> cache_pal;
	NodoLista *cabeza, *cola;
	int largo;
	int largo_maximo;
	int cambio;
	Cache(){
		cabeza=NULL;
		cola=NULL;
		largo=0;
		largo_maximo=10;
		cambio=0;
	}
	Cache(int _largo_maximo){
		cabeza=NULL;
		cola=NULL;
		largo=0;
		largo_maximo=_largo_maximo;
		cambio=0;
	}
	~Cache(){
		cola=NULL;
		if(cabeza!=NULL)
			delete cabeza;
		cache_pal.clear();
	}
	void agregar(string *_s, int * lista_docs1, int cnt){
		if((*_s).size()==0){
			return;
		}
		if(largo_maximo==0)
			return;
		vector<int> *_v =new vector <int>((2*cnt),0);
		for(int i=0;i<cnt;i++){
			(*_v)[2*i]=lista_docs1[i];
		}
		if(cabeza==NULL){
			cabeza=new NodoLista(_s, _v);
			cola=cabeza;
			cache_pal[(*(cola->s))]=cola;
			largo++;
		}
		else{
			cola->sgte=new NodoLista(_s, _v);
			cola->sgte->ante=cola;
			cola=cola->sgte;
			cache_pal[(*(cola->s))]=cola;
			largo++;
		}
		if(largo>largo_maximo){
			NodoLista *eliminar;
			eliminar=cabeza;
			cabeza=cabeza->sgte;
			cabeza->ante=NULL;
			eliminar->sgte=NULL;
			eliminar->ante=NULL;
			cache_pal.erase(eliminar->get_string());
			delete eliminar;
			cambio++;
			largo--;
		}
	}
```

Figura 9.2: Código en c++ de la clase cache dinámico (parte 1).

```
        vector <int> * busca(string *s){
                map<string,NodoLista *>::iterator it=cache_pal.find(*s);
                if(it==cache_pal.end()){
                        return NULL;
                }
                else{
                        (*this).ascender(((*it).second));
                        ((*it).second)=cola;
                        return (((*it).second)->get_vector());
                }
        }
        vector <int> * vector_cola(){
                return cola->get_vector();
        }
        int get_cambios(){
                return cambio;
        }
        void ascender(NodoLista *nodo){
                if(nodo==cola){
                        //no hay nada que hacer
                }
                else if(nodo==cabeza){
                        cabeza=cabeza->sgte;
                        cabeza->ante=NULL;
                        cola->sgte=nodo;
                        nodo->ante=cola;
                        nodo->sgte=NULL;
                        cola=cola->sgte;
                }
                else{
                        NodoLista *p1=nodo->ante;
                        NodoLista *p2=nodo->sgte;
                        p1->sgte=p2;
                        p2->ante=p1;
                        cola->sgte=nodo;
                        nodo->ante=cola;
                        nodo->sgte=NULL;
                        cola=cola->sgte;
                }
        }
};
#endif
```

Figura 9.3: Código en c++ de la clase cache dinámico (parte 2).

9.3. Código Nodo_lista

```
#if !defined(_NODO_LISTA_H)
#define _NODO_LISTA_H
#include <string.h>
using namespace std;
class NodoLista{
public:
        string *s;
        vector<int> *v;
        NodoLista *sgte;
        NodoLista *ante;
        NodoLista(){
                s=NULL;
                v=new vector<int>();
                sgte=NULL;
                ante=NULL;
        }
        NodoLista(string *_s){
                s=new string(_s->data());
                v=new vector<int>();
                sgte=NULL;
                ante=NULL;
        }
        NodoLista(string *_s, vector<int> *_v){
                s=new string(_s->data());
                v=new vector<int>(_v->begin(), _v->end());
                sgte=NULL;
                ante=NULL;
        }
        ~NodoLista(){
                ante=NULL;
                if(s!=NULL){
                        s->clear();
                        delete s;
                }
                if(v!=NULL){
                        v->clear();
                        delete v;
                }
                if(sgte!=NULL){
                        delete sgte;
                }
        }
        vector<int> * get_vector(){
                return v;
        }
        string get_string(){
                return *s;
        }
};
#endif
```

Figura 9.4: Código en c++ de la clase Nodo lista.