UNIVERSIDAD DE SANTIAGO DE CHILE
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

Estrategias de planificación para motores de búsqueda verticales

Danilo Fernando Bustos Pérez

Profesor Guía: Dra. Carolina Bonacic Castro
Profesor Co-guía: Dr. Mauricio Marín Caihuán

Trabajo de Titulacíon presentado en conformidad a los requisitos para obtener el Título de Ingeniero Civil Informático

SANTIAGO DE CHILE
2014

AGRADECIMIENTOS

Dedicado a ... .

RESUMEN

Resumen en Castellano

Palabras Claves: keyword1, keyword2 .

ABSTRACT

Resumen en Inglés

Keywords: keyword1, keyword2 .

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO 1. INTRODUCCIÓN

1.1 ANTECEDENTES Y MOTIVACIÓN

La World Wide Web es conocida como una gran teleraña mundial en la que existen millones de computadores conectados y la que desde el año 1993 crece exponencialmente, a tal punto que es incapaz de detectar sus propios cambios. A medida que pasa el tiempo y la Web sigue creciendo, los motores de búsqueda se convierten en una herramienta cada vez más usada e importante para los usuarios. Estas máquinas ayudan a los usuarios a buscar contenido dentro de la Web, puesto que conocen en cuáles documentos de la ella aparecen qué palabras. Si estas máquinas no existieran, los usuarios estarían obligados a conocer los localizadores de recursos uniformes (*URL*) de cada uno de los sitios a visitar. Además, los motores de búsquedas en cierto modo conectan la Web, ya que existe un gran número de páginas que no tienen referencia desde otras, siendo el único modo de acceder a ellas a través de un motor de búsqueda (Baeza-Yates et al., 2008).

La tendencia actual es incluir lo que han llamado búsqueda en tiempo real, que consiste en incluir en los resultados de las búsquedas documentos actualizados en el pasado muy reciente, por ejemplo, en una ventana de tiempo de minutos. Esto permite incluir en los resultados de las búsquedas a sistemas muy activos respecto de publicación de nuevos documentos, tales como *Twitter*[1]. Esto presenta desafíos importantes para los motores de búsqueda puesto que no solo deben procesar eficientemente a decenas de miles de consultas por segundo, sino que también

[1]http://www.twitter.com

deben permitir que sus índices invertidos (Zobel & Moffat, 2006) sean actualizados de manera concurrente con las consultas que llegan al sistema. Por lo tanto, es relevante diseñar estrategias que permitan administrar a decenas de miles de consultas de usuarios concurrentes por segundo y a la vez ser eficientes.

Típicamente, un motor de búsqueda de gran escala como *Google*[2] o *Yahoo!*[3], recibe del orden de decenas de miles de consultas por segundo, a lo cual hay que sumarle la cantidad de actualizaciones a los datos en el índice. Esto genera cargas de trabajo que deben ser servidas por muchos procesadores organizados de manera de satisfacer dos métricas de eficiencias relevantes:

1. El tiempo de respuesta por operación. Este debe ser del orden de las pocas decenas de milisegundos en cada servicio, con el fin de poder responder al usuario en tiempos del orden de la fracción de segundo.

2. El número de transacciones de lectura y/o escrituras servidas por segundo (*throughput*), el cual no debe verse afectado con la intensidad de tráfico de operaciones que llegan al motor de búsqueda.

Los motores de búsquedas verticales hacen distinciones de tópicos específicos, estos son sometidos a la misma intensidad de trabajo que un motor de búsqueda horizontal como *Yahoo!*, la diferencia más importante entre ambos es que estos últimos poseen un índice invertido muy grande, es decir, que cada lista invertida tiene un tamaño lo suficientemente grande como para sustentar de manera eficiente el paralelismo. Por el contrario, esto no ocurre en el caso de motores de búsqueda verticales, donde la cantidad de documentos a ser indexados es menor que los que existen en la Web mundial.

Por otra parte, los procesadores actuales tienden a aumentar cada vez más la cantidad de núcleos, lo cual permite incrementar la cantidad de hilos o hebras de ejecución disponibles para procesar consultas. Un enfoque tradicional para realizar el procesamiento de consultas es

[2]http://www.google.com
[3]http://www.yahoo.com

asignar una hebra para resolver una consulta individual, aquí el paralelismo se logra procesando muchas consultas individuales en diferentes núcleos, una por cada hebra y se maximiza el número de consultas resolviéndose al mismo tiempo; sin embargo, esto hace que eventualmente una consulta se demore mucho tiempo en ser resuelta. Un enfoque alternativo es utilizar todos los hilos de ejecución disponibles en una máquina para resolver cada transacción de lectura que llega al sistema, de esta forma se minimizaría el tiempo de resolución por consulta, sin embargo, la resolución de consultas es secuencial. Quizás sea mejor un punto intermedio en que se pueda resolver consultas a un tiempo considerable y a la vez paralelizar el procesamiento de transacciones de lectura.

Hoy en día los motores de búsquedas junto con sus centros de datos consumen el 2 % de la energía mundial, y se espera que en los próximos años esta cifra aumente alrededor del 30 %, puesto que la Web se duplica cada ocho meses y el número de nuevos usuarios que se conectan a esta crece año a año, para lo cual se hace casi inevitable el aumento de máquinas (*hardware*) a los centros de procesamiento de datos de compañías dueñas de motores de búsqueda a gran escala. Es por esto que se hace imprescindible diseñar e implementar algoritmos que trabajen en paralelo, para de esta forma mejorar el *throughput* de los motores de búsqueda, así se podrá resolver transacciones de manera más rápida en períodos de altas cargas de trabajo y apagar aquellos servidores que fueron prendidos durante estos períodos. Llegar a soluciones que aporten a mejorar el tiempo de respuesta de una transacción de lectura y el *throughput* de un motor de búsqueda, requiere conocimientos de diferentes áreas de la ciencia de la computación como por ejemplo: Recuperación de información, computación paralela, compresión de datos, *scheduling*, entre otras.

Los desafíos expuestos anteriormente hace que exista una comunidad activa intentando encontrar soluciones eficientes para sistemas de recuperación de información como los motores de búsqueda.

1.2 DESCRIPCIÓN DEL PROBLEMA

Para proporcionar un tiempo de respuesta adecuado a cada una de las consultas de los usuarios, los motores de búsquedas garantizan una cota superior de tiempo para la respuesta a una consulta (Jeon et al., 2014). Para un motor de búsqueda es muy importante reducir el tiempo de ejecución de aquellas consultas que tomarán mucho tiempo en ser resueltas, esto es muchas veces más importante que reducir el tiempo medio de respuesta (Dean & Barroso, 2013). Por lo anteriormente descrito, se plantea la siguiente pregunta que guia el presente trabajo: "¿Es posible diseñar un modelo de procesamiento y de planificación de transacciones de lectura que (1) asegure una cota superior de tiempo de respuesta para las consultas de los usuarios, y (2) minimice el tiempo en procesar lotes de consultas?".

Los motores de búsqueda utilizan un método multihilo llamado Wand para obtener el conjunto de los mejores documentos (conjunto *top-K*) para una transacción de lectura, el cual posee dos enfoques: (1) Con *heap* locales, en el que cada hebra procesa una parte del índice invertido y obtiene su propio conjunto *top-K*, posteriormente la hebra maestra hace la combinación de resultatos para obtener el conjunto final; (2) Con *heap* compartido, en el que cada hebra procesa una parte del índice invertido y cada vez que ella encuentra un documento que debe estar dentro del conjunto *top-K* final, se pide acceso exclusivo al *heap* compartido y se inserta. Existen trabajos en donde se estudia ambos enfoques (Rojas et al., 2013), sin embargo, aún no es posible ser categórico en decir cuál enfoque es mejor que otro. Por lo tanto, para poder crear un modelo eficiente de procesamiento y de planificación de transacciones de lectura, primero se debe analizar ambos enfoques y decidir cuál se ajusta de mejor forma al presente contexto.

Decidir si utilizar una o todas las hebras de una máquina para resolver una transacción de lectura dependerá de los objetivos del sistema de recuperación de información, generalmente estos dos enfoques son limitados. Encontrar un punto medio en donde se pueda cumplir con

(1) una cota superior aceptable de tiempo para responder a cada consulta y (2) tener un buen *throughput*, es un desafío importante para un motor de búsqueda. Por lo tanto, a través del presente trabajo se intenta encontrar una forma de asignación eficiente de hebras a consultas de manera tal que se cumplan los dos requerimientos mencionados anteriormente.

1.3 SOLUCIÓN PROPUESTA

La solución propuesta en este trabajo consta del diseño e implementación de estrategias de planificación de transacciones de lectura para motores de búsqueda vertical. A su vez, la estrategia generada como solución tiene como foco principal una sola máquina, en donde el procesamiento de consultras debe explotar el el paralelismo liviano.

1.3.1 Características de la solución

La solución propuesta incluye al la implementación de un método de predicción de tiempo de respuesta a transacciones de lectura para la asignación eficiente de hilos de ejecucion, con el objetivo de reducir el tiempo de procesamiento de las consultas.

Se implementaron dos modelos de procesamiento paralelo de consulta basado en el método Wand: Con *heap* compartido y *heaps* locales. De esta manera el sistema es flexible para trabajar con una o más hebras.

Se diseñó una estrategia capaz de reordenar dinámicamente las transacciones que llegan

al procesador.

Finalmente, se reunió en un solo esquema las mejores soluciones anteriormente descritas, y mediante experimentación se evaluó el rendimiento y efectividad de esta nueva estrategia. Se encontró el esquema óptimo de solución con las implementaciones descritas.

La métrica a optimizar es el número de consultas resueltas por unidad de tiempo, garantizando un tiempo de respuesta individual menor a una cota superior establecida y el tiempo medio en resolver conjuntos de transacciones. Por lo tanto, si se diseña una estrategia de procesamiento de consultas que le permita a cada máquina alcanzar una mayor tasa de resolución, la recompensa puede ser una reducción del total de nodos desplegados en producción.

1.3.2 Propósito de la solución

El primer propósito de la solución es asegurar una cota superior de tiempo en las respuestas de las consultas que llegan al motor de búsqueda, esto implica entregar al usuario tiempos aceptables en sus búsquedas.

Un segundo propósito de la solución es reducir el tiempo de procesamiento de lotes de transacciones de lectura. Esto traería como beneficio alcanzar un uso más eficiente de los recursos asignados a las operaciones de un motor de búsqueda en un centro de datos, ya que los procesadores serán utilizados de manera más eficiente, esto implicaría que un motor de búsqueda vertical estará preparado de mejor forma para resolver flujos grandes de transacciones y los tiempos en resolver cada transacción será menor.

Finalmente, se espera proveer un modelo de procesamiento y planificación de transacciones de lectura para un motor de búsqueda que sea flexible a la actualización concurrente de su índice invertido. De esta forma se permitirá incluir en las respuestas de las consultas a contenido creado

en el pasado muy reciente.

1.3.3 Alcances de la solución

1.4 OBJETIVOS DEL PROYECTO

1.4.1 Objetivo general

Diseñar, analizar, implementar y evaluar estrategias de procesamiento y planificación de transacciones de lectura para motores de búsqueda verticales para la Web. Estas deben ser capaces de resolver de manera eficiente el problema de procesar decenas de miles de consultas, asegurando una cota superior de tiempo de respuesta de cada operación.

1.4.2 Objetivos específicos

- Desarrollar algoritmos paralelos de procesamiento de transacciones de lectura que sean eficientes.

- Obtener estrategias *online* que permitan realizar la gestión eficiente de hebras a través de algoritmos que se ajusten a los tamaños de las estructuras de datos involucrados en el procesamiento de las transacciones de lectura.
- Obtener estrategias de reordenamiento dinámico de transacciones de lectura que llegan a un procesador con múltiples núcleos.
- Obtener un modelo de costo que permita analizar la eficiencia y escalabilidad de las estrategias diseñadas en motores de búsqueda verticales.

CAPÍTULO 2. MARCO TEÓRICO

En este capítulo se exponen los conceptos teóricos del presente trabajo de tesis. Primero se explica qué es un motor de búsqueda vertical. Luego se definen las estrategias de evaluación de transacciones de lectura, también conocidas como consultas o *queries*. Posteriormente se describen las diferentes operaciones sobre listas invertidas. Finalmente se explica el concepto de *ranking*.

2.1 MOTORES DE BÚSQUEDA VERTICALES

Un motor de búsqueda está construído por diversos componentes, su arquitectura típica se puede ver en la Figura 2.1. Existe un proceso denominado *crawling*, éste posee una tabla con los documentos iniciales en los que se extrae el contenido de cada uno de ellos. A medida que el *crawler* comienza a encontrar enlaces a otros documentos, la tabla de documentos a visitar crece. El contenido que se extrae en el procedimiento de *crawling* es enviado al proceso de indexamiento, este se encarga de crear un índice de los documentos ya visitados por el *crawler* (Croft et al., 2009).

Dado el volúmen de datos involucrado en el procesamiento, se debe tener una estructura de datos que permita encontrar cuáles documentos contienen las palabras presentes en la búsqueda que llega al sistema. Todo esto dentro de un período de tiempo aceptable. El índice invertido (Zobel & Moffat, 2006) es una estructura de datos que contiene un diccionario con todas las palabras que el proceso de *crawling* ha encontrado, asociado a cada palabra se tiene una lista de

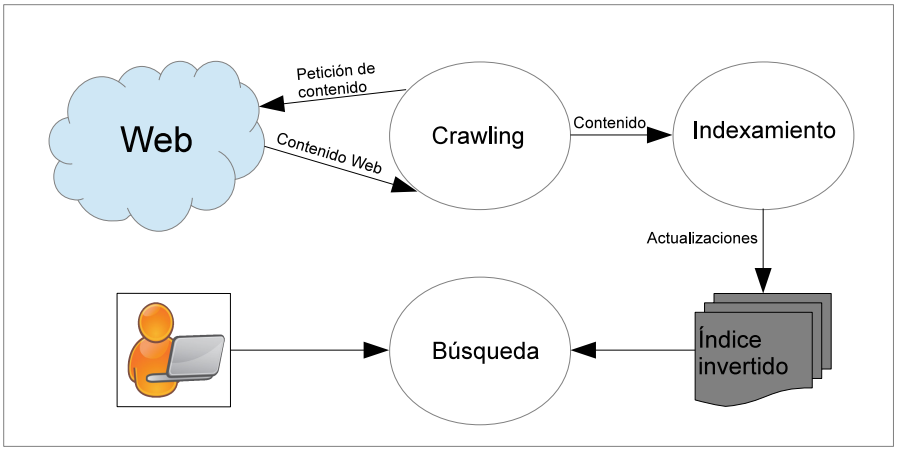

FIGURA 2.1: Arquitectura típica de un motor de búsqueda

todos los documentos en donde esta palabra aparece mencionada (conocida como lista invertida de un término). El motor de búsqueda construye esta estructura con el objetivo de acelerar el proceso de las búsquedas que llegan al sistema. El proceso de búsqueda es el encargado de recibir las transacciones de lectura, generar un *ranking* de los documentos que contienen las palabras de la consulta y finalmente generar una respuesta. Las diversas formas de calcular la relevancia de un documento será explicado en secciones posteriores.

En un motor de búsqueda se pueden encontrar diversos servicios tales como (a) cálculo de los mejores documentos para una cierta consulta; (b) construcción de la página Web en la que se mostrará al usuario los resultados; (c) publicidad relacionada con las transacciones de lectura; (d) sugerencias en el momento que el usuario está escribiendo la consulta en el sistema; entre muchos otros servicios.

En los sistemas de recuperación de información como los motores de búsqueda, lo que se hace hoy en día es agrupar computadores para procesar una transacción y obtener la respuesta para ésta. Este conjunto de computadores recibe el nombre de *cluster* (Dean, 2009).

La diferencia entre un motor de búsqueda vertical y uno general, es que el primero se centra solo en un contenido específico de la Web (Gil-Costa et al., 2013). El *crawler* debe extraer contenido solo de aquellas páginas Web que están dentro del dominio permitido. Al

ser un dominio acotado, los documentos a procesar serán menos y por lo tanto, la lista de los términos del índice invertido serán eventualmente de menor tamaño. Sin embargo, en un motor de búsqueda vertical las actualizaciones al índice invertido ocurren con mayor frecuencia.

2.2 ÍNDICE INVERTIDO

Es una estructura de datos que contiene todos los términos (palabras) encontrados por el *crawler*. A cada uno de los términos está asociado una lista invertida de documentos (páginas Web) que contienen dicho término. Adicionalmente, se almacena información que permita realizar el *ranking* de documentos para generar la respuesta a las consultas que llegan al sistema, por ejemplo, el número de veces que aparece el término en el documento.

Para construir un índice invertido (Baeza-Yates & Ribeiro-Neto, 2011; Salton & McGill, 2003) se debe procesar cada palabra que existe en un documento, registrando su posición y la cantidad de veces que éste se repite. Cuando se procesa el término con la información asociada correspondiente, se almacena en el índice invertido (ver Figura 2.2).

El tamaño del índice invertido crece rápido y eventualmente la memoria RAM se agota antes de procesar toda la colección de documentos. Cuando la memoria RAM se agota, se almacena en disco el índice parcial, se libera la memoria y se continúa con el proceso. Además, se debe hacer un *merge* de los índices parciales uniéndo las listas invertidas de cada uno de los términos involucrados. Es por esto que se han desarrollado algunas técnicas de compresión con el objetivo de guardar de una manera más eficiente el índice invertido (Arroyuelo et al., 2013; Baeza-Yates & Ribeiro-Neto, 2011; Yan et al., 2009).

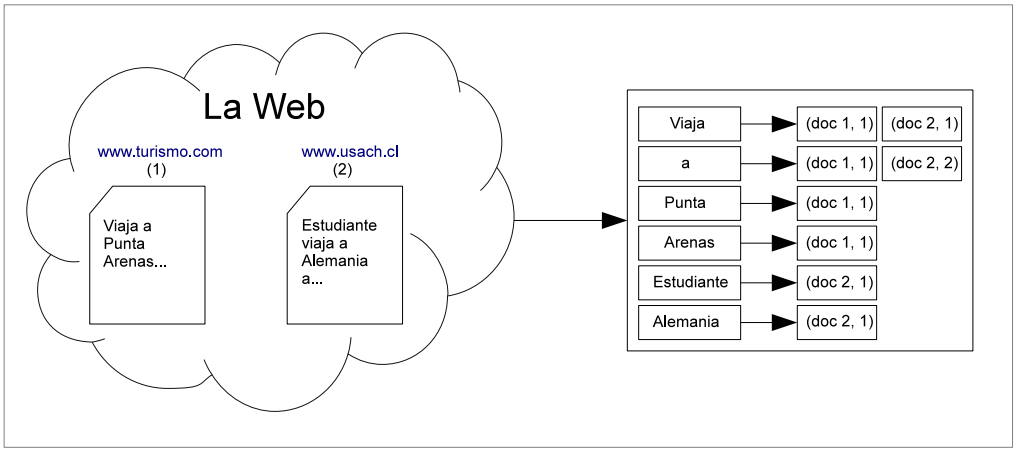

FIGURA 2.2: Índice invertido

2.3 ESTRATEGIAS DE EVALUACIÓN DE TRANSACCIONES DE LECTURA

Una de las tareas que un motor de búsqueda debe hacer para resolver una consulta es calcular el puntaje o *score* para aquellos documentos relevantes en la consulta y así poder extraer los mejores K documentos. Existen dos principales estrategias para recorrer las listas invertidas y calcular el puntaje de los documentos para una determinada consulta. Estas son (a) *term-at-a-time* (Buckley & Lewit, 1985; Turtle & Flood, 1995) y (b) *document-at-a-time* (Broder et al., 2003; Turtle & Flood, 1995).

2.3.1 Term at a time

Abreviada TAAT, este tipo de estrategia procesa los términos de las consultas una a una y acumula el puntaje parcial de los documentos. Las listas invertidas asociadas a un término

son procesadas secuencialmente, esto significa que todos los documentos presentes en la lista invertida del término t_i obtienen un puntaje parcial antes de comenzar el procesamiento del término t_{i+1}. La secuencialidad en este caso es con respecto a los términos contenidos en la transacción de lectura.

2.3.2 Document at a time

Abreviada DAAT, en este tipo de estrategias se evalúa la contribución de todos los términos de la *query* con respecto a un documento antes de evaluar el siguiente documento. Las listas invertidas de cada término de la consulta son procesadas en paralelo, de modo que el puntaje del documento d_j se calcula considerando todos los términos de la transacción de lectura al mismo tiempo. Una vez que se obtiene el puntaje del documento d_j para la consulta completa, se procede al procesamiento del documento d_{j+1}. Este tipo de estrategia posee dos grandes ventajas: (a) Requieren menor cantidad de memoria para su ejecución, ya que el puntaje parcial por documento no necesita ser guardado y (b) Explotan el paralismo de entrada y salida (I/O) más eficientemente procesando las listas invertidas en diferentes discos simultáneamente.

2.4 FUNCIONES DE *RANKING*

Los sistemas de recuperación de información como los motores de búsqueda deben ejecutar un proceso el cual asigna puntaje a documentos con respecto a una determinada transacción

de lectura, este proceso se denomina *ranking* (Baeza-Yates & Ribeiro-Neto, 2011). Como se puede ver en la Figura 2.3, este proceso toma como entrada la representación de las consultas y documentos, y asigna un *score* a un documento d_j dada una *query* q_i.

Un motor de búsqueda guarda billones de documentos que están formados por términos o palabras, estos términos no todos poseen la misma utilidad para describir el contenido del documento. Determinar la importancia de una palabra en un documento no es tarea sencilla, para ello se asocia un peso positivo $w_{i,j}$ a cada término t_i del documento d_j. De esta forma, para un término t_i que no aparezca en el documento d_j se tendrá $w_{i,j} = 0$. La asignación de pesos a los términos permite generar un *ranking* numérico para cada documento en la colección.

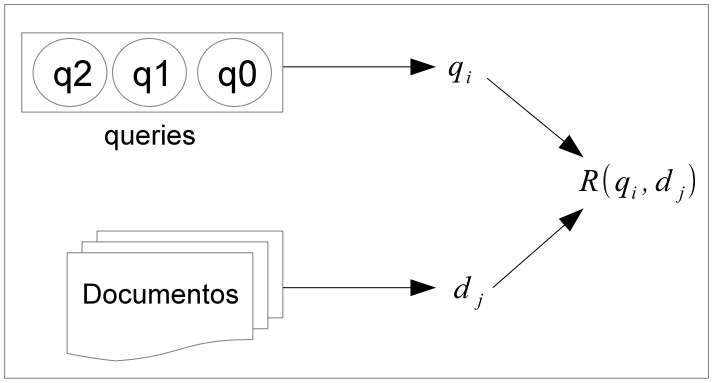

FIGURA 2.3: Proceso de *scoring* de documento

2.4.1 TF-IDF

$Tf - idf$ (*term frequency - inverse document frequency*) es un estadístico que tiene por objetivo reflejar cuán importante es una palabra para un documento en una colección o corpus

$$tf(t,d) = \frac{f(t,d)}{\text{máx}\, f(w,d) : w \in d} \tag{2.1}$$

El segundo término corresponde a la frecuencia inversa de documento (idf) y se utiliza para observar si es que el término es común en el corpus. El idf se obtiene calculando el logaritmo de la división entre el número total de documentos del corpus y el número de documentos que contienen el término.

$$idf(t, D) = log \frac{|D|}{1 + |d \in D : t \in d|} \tag{2.2}$$

De esta forma a partir de (2.1) y (2.2) se obtiene finalmente el $tf - idf$:

$$tf - idf(t, d, D) = tf(t, d) * idf(t, D) \tag{2.3}$$

Notar en (2.3) que el estadístico incrementa proporcionalmente al número de veces que la palabra aparece en el documento, sin embargo, es compensado por la frecuencia de la palabra en la colección completa de documentos o corpus. Esta compensación ayuda a controlar el hecho de que algunas palabras son generalmente más comunes que otras.

2.4.2 BM25

Es una función de *ranking* de documentos basada en los términos que aparecen en la consulta que llega al motor de búsqueda. *BM25* pertenece a una amplia gama de funciones de puntuación y está basada en los modelos probabilísticos de recuperación de la información (Baeza-Yates & Ribeiro-Neto, 2011).

Dada una *query* Q que contiene los términos $q_1, ..., q_n$, el *ranking BM25* del documento D se calcula como:

$$score(D,Q) = \sum_{i=1}^{n} IDF(q_i) * \frac{f(q_i,D) * (k+1)}{f(q_i,D) + k * (1 - b + b * \frac{|D|}{prom(docs)})} \tag{2.4}$$

En donde: $f(q_i, D)$ es la frecuencia en que aparece el término q_i en el documento D; $|D|$ es el número de palabras o términos en el documento D; $prom(docs)$ es la media de número de palabras de los documentos en el corpus; k y b son constantes que depende de las características del corpus en el que se está haciendo la búsqueda, por lo general se asignan los valores de $k = 2$ o $k = 1.2$ y $b = 0.75$; finalmente, $IDF(q_i)$ es la frecuencia inversa de documento para el término q_i.

2.5 OPERACIONES SOBRE LISTAS INVERTIDAS

Cuando una consulta llega al motor de búsqueda, cada término tiene asociado una lista con todos los documentos en los cuales aparece. El sistema debe decidir qué documentos se analizarán para obtener la respuesta y entregar al usuario una respuesta. A continuación se presenta los modos de operar las listas invertidas para una cierta transacción de lectura.

2.5.1 *OR*

Este operador toma las listas invertidas de cada uno de los términos de la *query* y ejecuta la disyunción entre ellas. El resultado de este operador es una lista invertida con todos los documentos que contengan al menos un término de la *query*. Finalmente, esta lista invertida se

ocupará para obtener los mejores K documentos. Un simple ejemplo se muestra en la Figura 2.4.

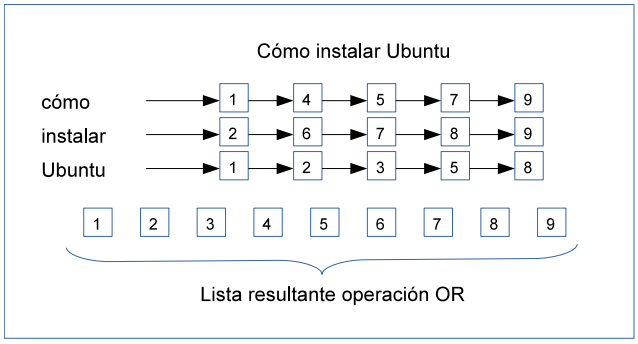

FIGURA 2.4: Operación OR

2.5.2 AND

Este operador ejecuta la conjunción entre las listas invertidas de los términos de una *query*. Se obtiene una lista invertida con los documentos que contengan todos los términos de la *query*. Se debe notar que aquí se obtiene una lista resultante de menor tamaño que la obtenida en el operador OR (Ver Figura 2.5).

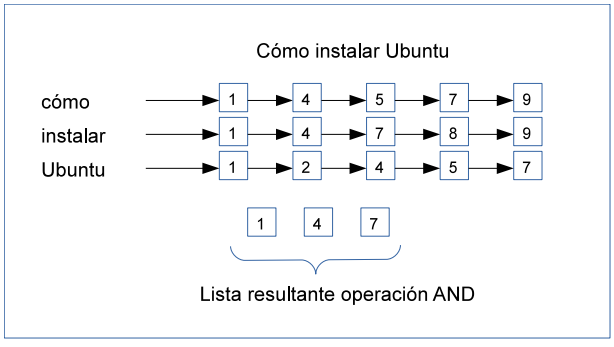

FIGURA 2.5: Operación AND

2.5.3 Wand

Algoritmo de evaluación de transacciones de lectura para obtener eficientemente el conjunto de K documentos que mejor satisfacen una consulta dada. *WAND* (Broder et al., 2003) es un proceso menos estricto que el método *AND* y está basado en dos niveles. Dentro del proceso de evaluación de una transacción de lectura, uno de los procesos más costoso en términos de tiempo es el de *scoring*, que consiste en entregarle a cada uno de los documentos analizados un puntaje que representa la relevancia del documento para una *query* dada, esto se denomina evaluación completa o cálculo del puntaje exacto del documento. El objetivo de WAND es minimizar la cantidad de evaluaciones completas de los documentos ejecutando un proceso de dos niveles. En el primer nivel se intenta omitir rápidamente grandes porciones de las listas invertida, lo que se traduce en ignorar el cálculo del puntaje exacto de grandes cantidades de documentos, esto porque en motores de búsqueda a gran escala, este un proceso que requiere de mucho tiempo para llevarse a cabo y depende de factores como la cantidad de ocurrencia del término dentro del documento, el tamaño del documento, entre otros. A este tipo de técnicas que intenta omitir partes de lista invertida se les conoce como técnica de poda dinámica (Broder et al., 2003; Persin, 1994; Turtle & Flood, 1995).

Para llevar a cabo el algoritmo WAND y así reducir el número de documentos completamente evaluados durante el proceso de *ranking* de documentos, se necesita calcular los valores estáticos de límite superior (*upper-bounds*), en donde para cada uno de los términos del índice invertido, se toma la lista invertida correspondiente y se extrae el puntaje máximo de contribución de algún documento con respecto al término. El cálculo de los *upper bounds* se lleva a cabo cuando se construye el índice invertido y en donde a cada término del índice se asocia el puntaje máximo que existe en la lista invertida.

WAND usa un índice invertido ordenado por los identificadores de documentos. En el primer nivel se itera sobre los documentos del índice invertido de cada término y se identifica

los potenciales candidatos usando una evaluación aproximada. En el segundo nivel, aquellos documentos candidatos son completamente evaluados y su puntaje exacto es calculado. De esta forma se obtiene el conjunto final de documentos. Se utiliza un heap como estructura de datos para almacenar el conjunto de los mejores K documentos, en donde el elemento superior corresponde al documento con menor puntaje y es el que se utilizará como umbral (*threshold*) para decidir si los siguientes documentos deben ser completamente evaluados o no.

En la Figura 2.6 se puede ver un ejemplo sencillo de cómo el algoritmo Wand trabaja en la resolución de una transacción de lectura de tres términos: 'casa', 'perro' y 'gato'. Como la *query* está compuesta por tres términos, existen tres punteros que recorren cada una de las listas invertidas (notar que cada puntero recorre una lista invertida diferente). Lo primero que se hace es ordenar las listas invertidas de acuerdo a los identificadores de documentos que se están apuntando, razón por la cual en la Figura 2.6 la lista invertida de 'casa' (puntero referenciando al documento con identificador 125), aparece primero que la lista invertida de 'perro' (puntero haciendo referencia al documento con identificador 503). Luego se suma los *uppers bounds* de los términos en orden hasta que se obtiene un valor mayor o igual al *threshold*. De esta manera el término 'perro' es escogido como término pivote ($2.0 + 4.4 \geq 5.7$) y el actual documento al cual se está apuntando es escogido como documento pivote (documento con identificador 503). Si las dos primeras listas invertidas no contienen el documento 503 entonces se procede a seleccionar el siguiente pivote, en otro caso se calcula el puntaje completo del documento. Finalmente, si el puntaje es mayor o igual al *threshold*, se actualiza el *heap* eliminando el elemento superior y se añade el nuevo documento. Este algoritmo es repetido hasta que no hayan más documentos a procesar o hasta que no exista un documento que supere el actual *threshold*. De esta manera se evita procesar las listas completas (Blanco & Barreiro, 2010).

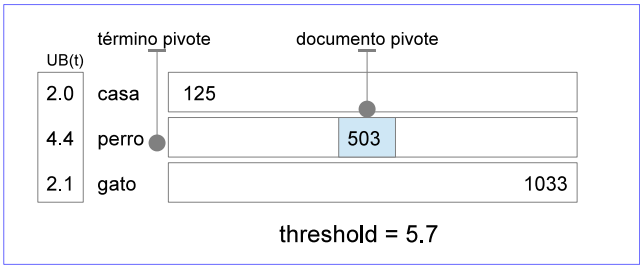

FIGURA 2.6: Ejemplo de ejecución de algoritmo *Wand*

2.5.4 Block Max Wand

Como se explicó en la sección anterior, la diferencia entre un método exhaustivo de evaluación de documentos y el método Wand, es que este último es una técnica DAAT de poda dinámica (Moffat & Zobel, 1996) en la que se intenta omitir la mayor cantidad de evaluaciones de documentos haciendo uso de una estrategia de movimientos de punteros pivotes. Bajo la premisa que Wand tradicional es limitado por el hecho que usa los máximos puntajes de las listas invertidas (*Upper bounds*) para podar, puesto que estos pueden ser mucho más grandes que el promedio de puntaje en ellas, se propone un método llamado *Block-Max-Wand* (BMW) (Ding & Suel, 2011). Este método utiliza una estructura de datos llamada índice *Block-Max*, en donde el índice invertido estará particionado en bloques y para cada bloque se almacena la máxima contribución de algún documento dentro del bloque. En otras palabras, se tendrán tantos *upper bounds* locales como bloques existan en la lista invertida.

Este método utiliza una variación del algoritmo Wand tradicional para que trabaje correctamente con la nueva estructura *Block-Max-Wand*. Remplazar el uso de los *upper bounds* por cada bloque por el *upper bound* global no garantiza la correctitud del algoritmo. En la Figura 2.7 se muestra un ejemplo de por qué mirar solo los *upper bounds* locales no garantiza obtener los resultados correctos, aquí no se puede concluir que el documento 4868 es el documento más pequeño que puede estar dentro del conjunto *top-K*, ya que $2.5 + 2.0 + 3.5 \geq 7.0$ (conclusión

que sí es válida utilizando Wand tradicional y *upper bounds* globales), porque es posible que el bloque siguiente al bloque del docID 275 (en la primera lista), tenga un *upper bound* local mayor. Por lo tanto, aplicar solo las máximas contribuciones por bloque no permite al algoritmo omitir documentos de forma segura.

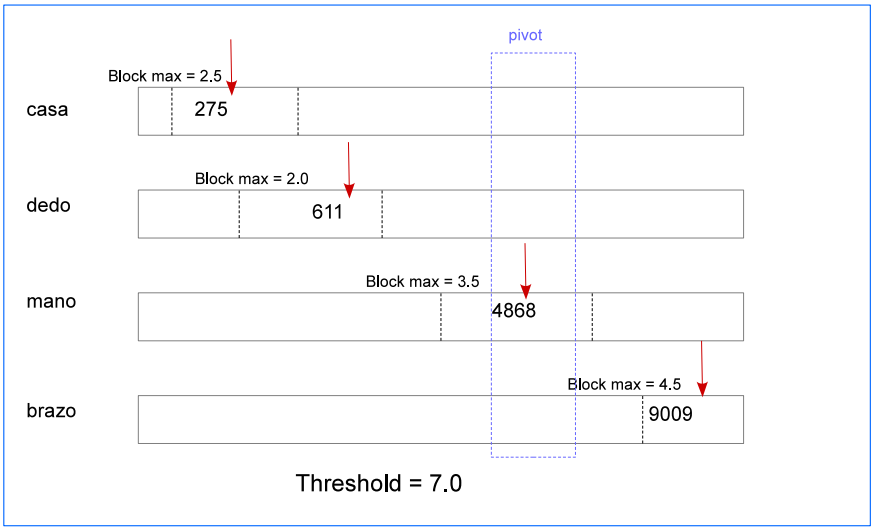

FIGURA 2.7: Ejemplo del proceso Block-Max-Wand

2.6 PREDICCIÓN DE TIEMPO DE RESPUESTAS DE TRANSACCIONES DE LECTURAS

El rendimiento (*performance*) de una consulta puede medirse de dos formas: Efectividad y eficiencia. La efectividad tiene relación con la calidad de los documentos extraídos para una cierta consulta y la eficiencia corresponde al tiempo que conlleva procesarla. El tiempo que le toma al sistema en resolver una consulta puede variar considerablemente. Con el objetivo de retornar los resultados al usuario dentro de una cota superior de tiempo, aquellas consultas

que toman una mayor cantidad de tiempo en ser procesadas se requiere una mayor cantidad de procesadores para resolverla, de esta forma podemos asegurar esta cota de tiempo. El tener un buen predictor de la eficiencia de una *query* es muy útil, por ejemplo, si pensamos en un sistema con réplicas, podemos planificar la consulta en el servidor que se desocupará más pronto.

Existen estudios en los cuales el rendimiento es inferido usando *clarity score* (Cronen-Townsend et al., 2002), que es una forma para evaluar la pérdida de ambigüedad de una transacción con respecto a la colección. En (He & Ounis, 2004) se propone un conjunto de predictores para el rendimiento de cada consulta. Técnicas de aprendizaje de máquina también han sido estudiadas para predecir el rendimiento de transacciones de lectura (Si & Callan, 2002). Todos los estudios mencionados anteriormente se han centrado en la efectividad para ser predicciones de rendimento de *queries*. La eficiencia de una transacción de lectura también ha sido objeto de estudio, identificando las principales razones que tienen impacto sobre el tiempo de respuesta y evaluando estos factores para predecir el comportamiento de futuras consultas (Tonellotto et al., 2011). En (Macdonald et al., 2012) se propone un método de predicción de tiempo de respuesta para consultas basado en datos estadísticos disponibles en las respectivas listas invertidas de los términos. Finalmente, en (Jeon et al., 2014) además de utilizar estadísticos disponibles en las listas invertidas de los términos, se agregan estadísticos propios de las *queries* para la creación de un predictor.

2.7 SCHEDULING EN MOTORES DE BÚSQUEDA

Los motores de búsqueda no solo se preocupan de la calidad de los resultados de las búsquedas (efectividad), sino que también de la velocidad con la que los resultados son

obtenidos (eficiencia). Existen varias estrategias para mejorar la velocidad en la obtención de los resultados, una de ellas muy utilizada es el *caching*. Consiste en guardar en memoria de acceso rápido (memoria caché) datos temporales, que luego pueden ser sobrescrito. Una opción es hacer *caching* de los resultados de las búsquedas, de esta forma cuando una *query* es encontrada en caché el motor de búsqueda puede generar la respuesta rápidamente, reduciendo los tiempos de calculos considerablemente. Otra opción es, guardar en caché la intersección de las listas invertidas de pares comunes de términos que llegan al motor de búsqueda. Por ejemplo, si llega al sistema una consulta con los términos ('casa', 'árbol', 'perro'), se puede guardar en caché la intersección de las listas de 'casa' y 'árbol', para luego reutilizar esta información en otras *queries* que lleguen en el futuro. Para ver más técnicas de *caching* y ver el detalle de las técnicas mencionadas, ver (Büttcher et al., 2010).

Otra estrategia para acelerar el proceso de resolución de *queries* que llegan al sistema es el uso de algoritmos de planificación (scheduling). Un algoritmo de *scheduling* es el proceso en el cual se cambia el orden en que llegan las *queries* al motor de búsqueda con el objetivo de mejorar la eficiencia.

Existen dos clases de algoritmos de *scheduling*, estáticos y dinámicos. Los estáticos son aquellos en que se conoce el conjunto completo de tareas y las características de cada una de ellas, como por ejemplo, el tiempo de procesamiento. Los algoritmos de *scheduling* dinámicos son aquellos en que no se conoce las tareas que llegarán en el futuro, también se desconoce el momento en que éstas llegarán. La filosofía de los algoritmos de *scheduling* dinámicos es ajustarse a los cambios que pueden haber en el sistema.

En el contexto del presente trabajo de tesis, el objetivo de hacer *scheduling* es minimizar el tiempo en que las *queries* son procesadas por un motor de búsqueda. Los motores de búsqueda como *Google*[1] o *Yahoo!*[2] trabajan en un contexto *online*. Esto significa que cuando las *queries* llegan al sistema (una a una), éste está obligado a tomar una decisión para planificarla sin

[1]http://www.google.com
[2]http://www.yahoo.com

saber cuáles *queries* llegarán en un momento posterior. A esto se le conoce como algoritmo de *scheduling online* (Albers, 2003; Borodin & El-Yaniv, 1998).

Los sistemas de recuperación de información a gran escala despliegan una arquitectura distribuída (Dean, 2009), en donde el índice invertido está particionado (Barroso et al., 2003) a lo largo de servidores (*shard servers*), los cuales están encargados de procesar las *queries* que llegan al sistema. Es fácil notar que resolver una *query* con varios *shard servers* mejoraría la eficiencia. Ahora bien, para asegurar un alto rendimiento (*throughput*) del sistema, cada uno de los *shard servers* posee réplicas, de esta forma, más *queries* pueden ser procesadas en paralelo en copias idénticas del mismo *shard server*. Esto implica que el tiempo de espera de las *queries* que vienen llegando al sistema se reduce.

Como en un sistema con arquitectura como el de la Figura 2.8, una *query* puede ser procesada por varios *shard servers*, el *broker* debe escoger la réplica más apropiada para procesar la parte de la *query* asignada al *shard server*, con el objetivo de reducir el tiempo de espera de ésta. El *broker* podría seleccionar el *shard server* con el menor número de *queries* en la cola, sin embargo, este no es un parámetro adecuado, ya que el tiempo de respuesta de las *queries* puede variar considerablemente, especialmente si se usa poda dinámica (Broder et al., 2003; Moffat & Zobel, 1996).

2.7.1 Trabajo relacionado

El estudio (Broccolo et al., 2013) analiza métodos de *dropping* y *stopping* para el procesamiento de *queries* bajo altas carga de trabajo en un sistema distribuído donde existen múltiples servidores en el que cada uno resuelve una parte de la *query* para luego enviar las consultas al *broker* y éste hace el *merge* de los resultados de acuerdo al *score* de los documentos.

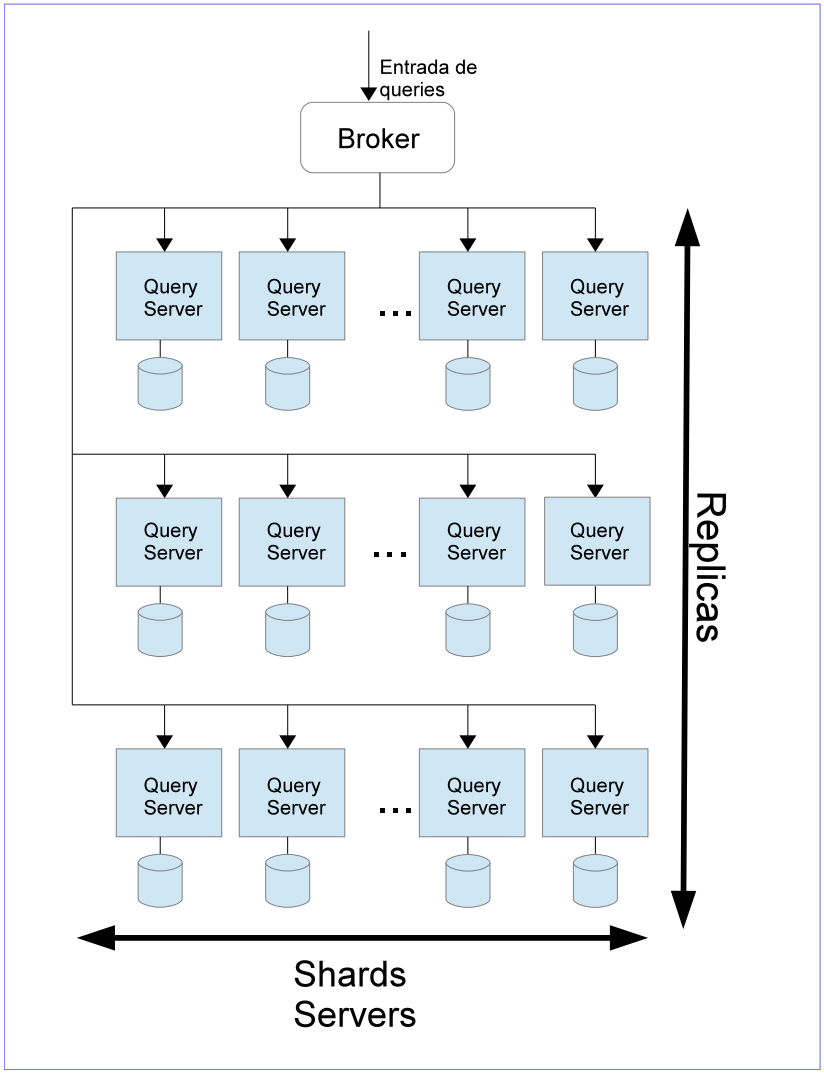

FIGURA 2.8: Arquitectura de un sistema de recuperación de la información con réplicas

Se define un tiempo T, en el que la suma de el tiempo de espera de la *query* para ser procesada (t_w) y el tiempo de procesamiento de la misma (t_p) deben ser menor a T. Si es que se sobrepasa este tiempo, se tienen dos opciones (1) la *query* es desechada y se envía al *broker* una lista vacía, (2) se detiene el procesamiento de la *query* y se envía los resultados parciales hasta el momento. Finalmente se propone un método basado en la predicción de tiempo de respuesta ($\hat{pt}(q)$) de una *query* (Macdonald et al., 2012) de modo que si se cumple $\hat{pt}(q) \leq T - wt(q)$, entonces la *query* es desechada antes de comenzar a procesarse y se toma la siguiente desde la cola de espera. Notar que en estos métodos existe una pérdida de efectividad, puesto que eventualmente los servidores muchas veces no enviarán sus mejores documentos al *broker*, esto implica que el *broker* responderá al usuario un conjunto de K documento que no necesariamente son los mejores dentro del corpus completo.

En (Freire et al., 2012) se estudia el impacto que tiene la técnica de predicción de tiempos de respuestas para *queries*, (Tonellotto et al., 2011) en sistemas de recuperación de la información con réplicas. En este estudio, se llega a la conclusión que usando una buena predicción, se puede reducir el tiempo que la *query* tiene que esperar para ser procesada (t_w), y también se puede reducir el tiempo total requerido para procesar el conjunto (*log*) completo de *queries* (*completion time*). En (Freire et al., 2013), se propone un modelo híbrido de *scheduling* de *queries* a través de réplicas, en el que cuando el sistema se encuentre bajo altas cargas de trabajo, se utilice política de *scheduling* basada en la predicción de tiempo de respuesta de las *queries* (Macdonald et al., 2012) y cuando el sistema se encuentre con una baja carga de trabajo, se utilice una politica de *scheduling* sencilla y de menor costo como *Round Robin*.

CAPÍTULO 3. WAND *MULTI-THREADED*

Dado que el método Wand (Broder et al., 2003) consiste en el método del estado del arte ocupado hoy en día por los motores de búsqueda para obtener eficientemente los mejores K documentos a una transacción de lectura, en esta investigación se asume un sistema que usa este método . Este algoritmo usa un *ranking* basado en una evaluación de dos niveles. En el primer nivel, este usa una cota superior (*upper bound*) al puntaje de cada documento para intentar descartarlos eficientemente. En el segundo nivel se computa el puntaje real de los documentos que pasa el primer nivel. Se utiliza una estructura de datos llamada *heap* que va guardando el conjunto de los mejores K documentos hasta un determinado instante. El menor puntaje de este conjunto es usado como umbral (*threshold*) para las evaluaciones del primer nivel, de esta forma se descarta rápidamente documentos que no pueden ser parte del conjunto final de los *top-K* documentos. Esto permite un eficiente y a la vez seguro proceso de descarte que asegura que en el resultado final se encontrará el conjunto correcto y no se perderán documentos relevantes.

Existe una variación al método Wand tradicional que intenta hacer una poda más agresiva, en otras palabra, lo que se intenta es tratar de omitir una mayor cantidad de documentos a la hora de resolver una transacción de lectura. Este método llamado Block Max Wand requiere que cada una de las listas invertidas este particionada en bloques (generalmente 64 o 128 bloques), en donde se tiene un upper bound por cada bloque. La lógica es la misma que en el método original y en la primera fase también se ocupa el máximo puntaje por lista para descartar documento, sin embargo, ahora existe una tercera fase en donde se utiliza el upper bound por bloques. De esta forma se intenta omitir una mayor cantidad de documentos. Más detalle estos dos métodos los puede encontrar en las secciones 2.5.3 y 2.5.4 del presente trabajo.

Wand y Block Max Wand son métodos que lógicamente son parecidos y que utilizan los mismos métodos, es por esto que el diseño de la implementación es como lo muestra la Figura

3.1; aquí se puede apreciar dos tipos de Wand: WandScorer y WandMaxBlock. WandScorer implementa el método tradicional utilizando el método next, que retornará algún documento que merezca estar en el conjunto top-K en ese momento. Por su parte, WandMaxBlock además de utiliza la función next(), usa la función nextShallow(), que moverá el puntero del documento actual de la lista a la posición inicial del bloque en donde debería encontrarse el documento que se le entrega como parámetro. La ventaja de este diseño es que ambas opciones son flexibles a utilizar cualquier functión de ranking que se desee, en el diagrama se observa la clase que utilizará Okapi BM25.

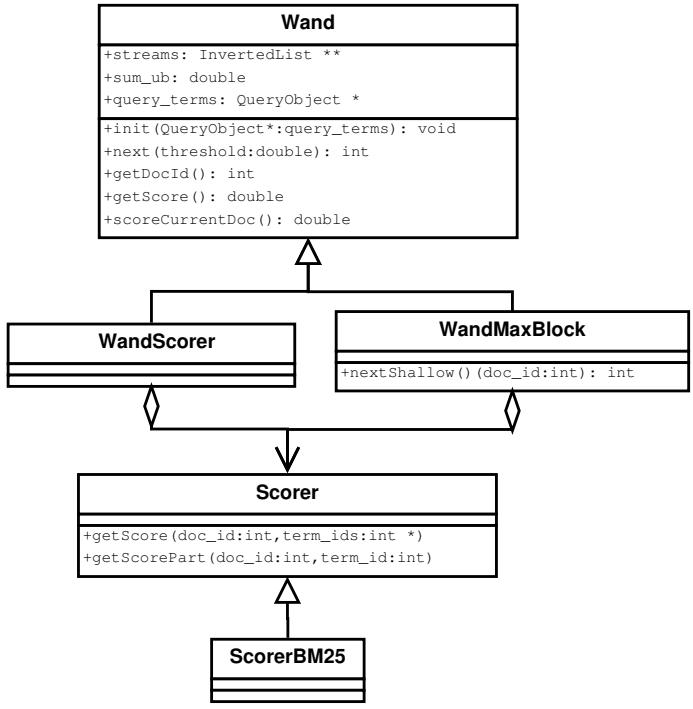

FIGURA 3.1: Diseño de clases para Wand y Block Max Wand

Existen dos formas de implementar Wand. Una de ellas es usando *heaps* locales (LH), es decir, un *heap* por hilo de ejecución (*thread*) y el otro es usando *heaps* compartidos (SH). El estudio en (Rojas et al., 2013) se muestra indicios que el esquema SH es generalmente más eficiente. Logrando rápidamente un óptimo valor para el threshold, el esquema SH posee las siguientes ventajas: (1) Se puede reducir el número de calculo de puntajes completos y (2) se

ejecutan pocas operaciones de actualización del heap (reduciendo el número de *locks* que se hace a la estructura de dato). A continuación se presenta el diseño llevado a cabo para ambos esquemas.

A continuacion se muestra las dos formas en que se implementó Wand y también la implementación de Block Max Wand.

3.1 WAND CON HEAPS LOCALES

En el esquema LH, cada *thread* procesa una porción del índice invertido mientras mantiene un *heap* local con los mejores K documentos que el específico hilo de ejecución ha encontrado hasta ahora. Al final del proceso, los resultados deben ser reunido en un solo conjunto final global. Los resultados en (Rojas et al., 2013) muestran que el esquema LH es más eficientes para aquellas transacciones que toman poco tiempo en ser resueltas. En la Figura 3.2 se muestra el esquema de ejecución para *heaps* locales explicado anteriormente.

El diseño aplicado para implementar el esquema LH se puede ver en la Figura 3.3. La clase principal es la TopKMultiThreadWandOperatorLocal, que es la encargada de controlar el paralelismo en la resolución de las transacciones. Para explicar de mejor manera cada una de las clases involucradas en la implementación, se presenta el siguiente diccionario de datos.

TopKMultiThreadWandOperatorLocal. Clase encargada de devolver los mejores K documentos para una query dada. Si es que la query debe ser resuelta en forma paralela, esta clase además debe controlar el paralelismo que se produce en la resolución de ésta, inicializando las variables correspondientes para lanzar los hilos de ejecución y luego escogiendo los mejores documentos desde todos los heaps creados por los diferentes threads

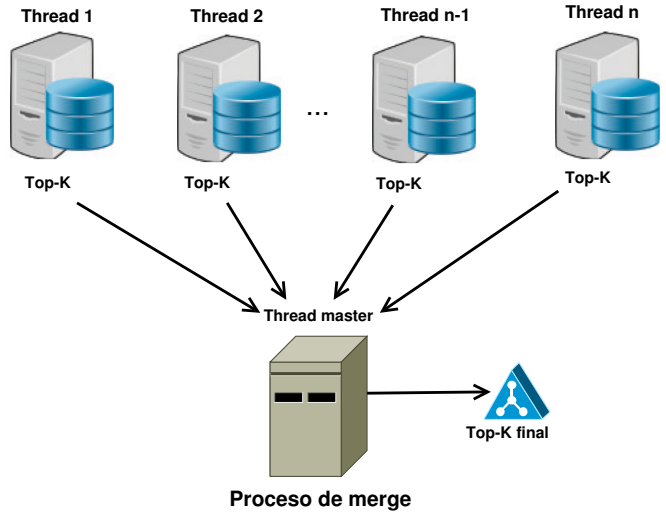

FIGURA 3.2: Esquema de ejecución de algoritmo WAND con heaps locales

(proceso de merge). En esta clase se define un mapa que asocia cada término del índice invertido con el puntaje del mejor documento en esa lista invertida (upper bound de la lista invertida) y además se define cuántos documentos se van a retornar al final del proceso (atributo K). El método execute inicializa las variables locales para los diferentes threads, posteriormente hace el llamado al método *thread-execute* (en el cual se llevará a cabo la resolución de la transacción de lectura en forma paralela), finalmente se toman los resultados parciales de cada uno de los hilos de ejecución y se ejecuta el proceso que mezcla los resultados, retornando solo los mejores K documentos.

PartitionedInvertedIndex. Clase que tiene la tarea de almacenar el índice invertido y extraer desde aquí las listas invertidas de documentos para cada uno de los términos de las transacciones de lectura. El almacenamiento el índice se lleva a cabo mediante un mapa cada término su lista invertida correspondiente y para la extracción de estas listas se usa el método getList.

TopKWandOperator. Cada thread tendrá su propio objeto TopKWandOperator encargado de obtener los mejores K documentos. El cálculo de este conjunto se realiza en

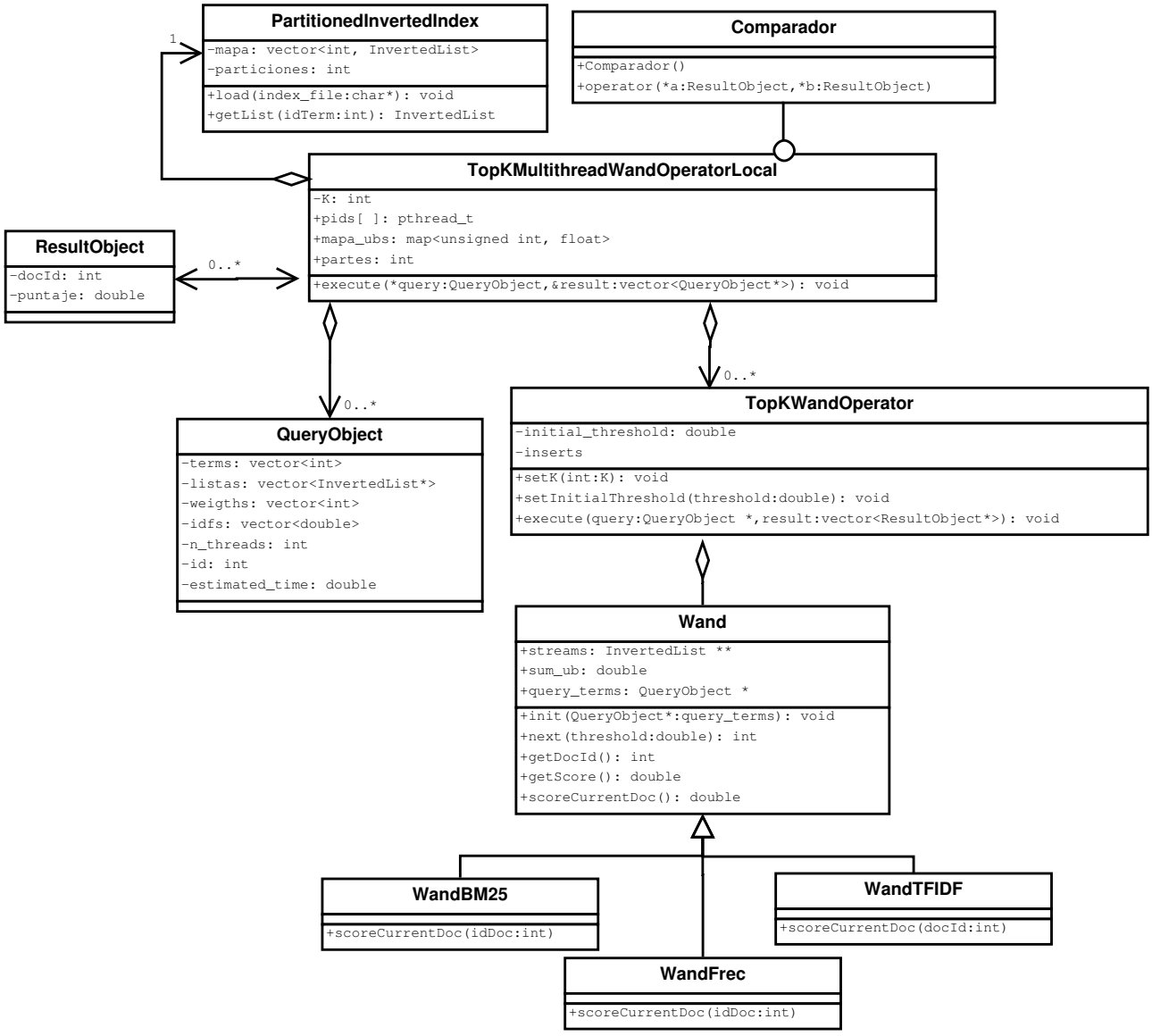

FIGURA 3.3: Diagrama de clases para el esquema LH

el método execute con la ayuda de un objeto de tipo Wand asociado.

Wand. Clase que controla la lógica del algoritmo wand. Lleva a cabo el proceso de inserción de documentos en el heap y todo lo que esto conlleva. Existen diferentes tipos de objetos Wand que se pueden utilizar, entre ellos están WandBM25, WandFrec y

WandTFIDF, donde la única diferencia entre ellos es el método de que calcula el puntaje de cada documento. Por ejemplo, WandBM25 utiliza BM25 (citar) y WandTFIDF utiliza tf-idf (citar también).

ResultObject. Clase que se utiliza para guardar los mejores K documentos.

QueryObject. Clase que representa una transacción de lectura. Está constutuída sus términos, las respectivas listas invertidas y pesos de cada uno de ellos, la cantidad de threads con los cuales se resolverá dicha transacción y el tiempo estimado de procesamiento (este tiempo se predice al momento de resolver la query).

3.2 WAND CON HEAP COMPARTIDO

En el esquema SH cada thread procesa una porción del índice. Sin embargo, ahora un solo heap es creado y accedido por todos los threads. En este caso no se requiere de mezclar los resultados y el proceso de descarte tiende a ser más eficiente porque los documentos con mayor puntaje tienden a estar en el heap. Acceder al heap debe ser controlado por un lock o algún método similar que garantice el acceso exclusivo de los threads al heap. Este esquema es más eficiente que el LH en queries que toman mayor tiempo en ser resueltas.

El diseño implementado para este esquema posee como clase principal a TopKMultiThreadWandOperatorLocks y difiere del modelo implementado para el esquema LH en el sentido que ahora se debe controlar el acceso concurrente a los datos compartidos como el heap y el threshold. A continuación se presenta el diccionario de datos del esquema SH mostrado en la Figura 3.5.

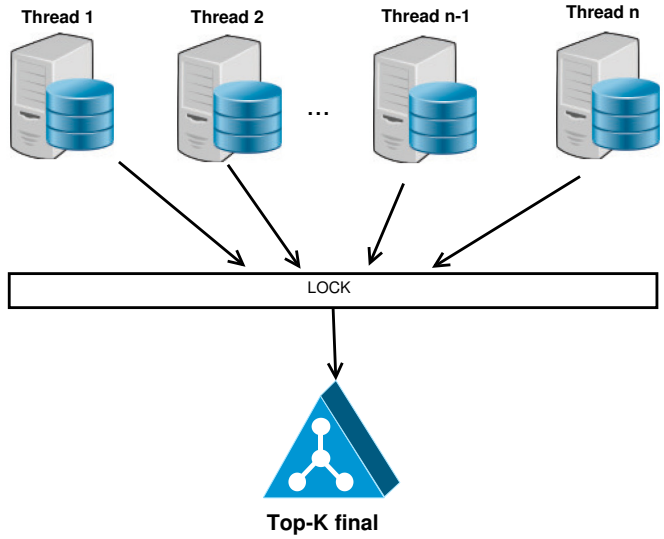

FIGURA 3.4: Esquema de ejecución de algoritmo WAND con heap compartido

TopKMultiThreadWandOperatorLocks. Clase encargada de inicializar las variables compartidas y de lanzar los threads requeridos para procesar la transacción de lectura.

WandThreadData. Clase anidada a TopKMultiThreadWandOperatorLocks que contendrá todas las variables compartidas para el procesamiento de las consultas. Dentro de los atributos más importantes destaca el mutex utilizado para controlar el acceso al heap compartido y además al threshold (en este esquema es un threshold global y compartido a todos los threads).

Wand. Al igual que en el esquema anterior, esta clase se encarga de llevar a cabo el proceso de inserción de documentos en el heap y de las actualizaciones del threshold. El método scoreCurrentDoc es el encargado de entregarle un puntaje a cada documento y dependerá de qué tipo de Wand se este utilizando (BM25, WandFrec, WandTFIDF).

PartitionedInvertedIndex. Clase encargada de almacenar el índice invertido. Posee un método llamado getList que recibe como parámetro el identificador de un documento y retorna la lista invertida asociada.

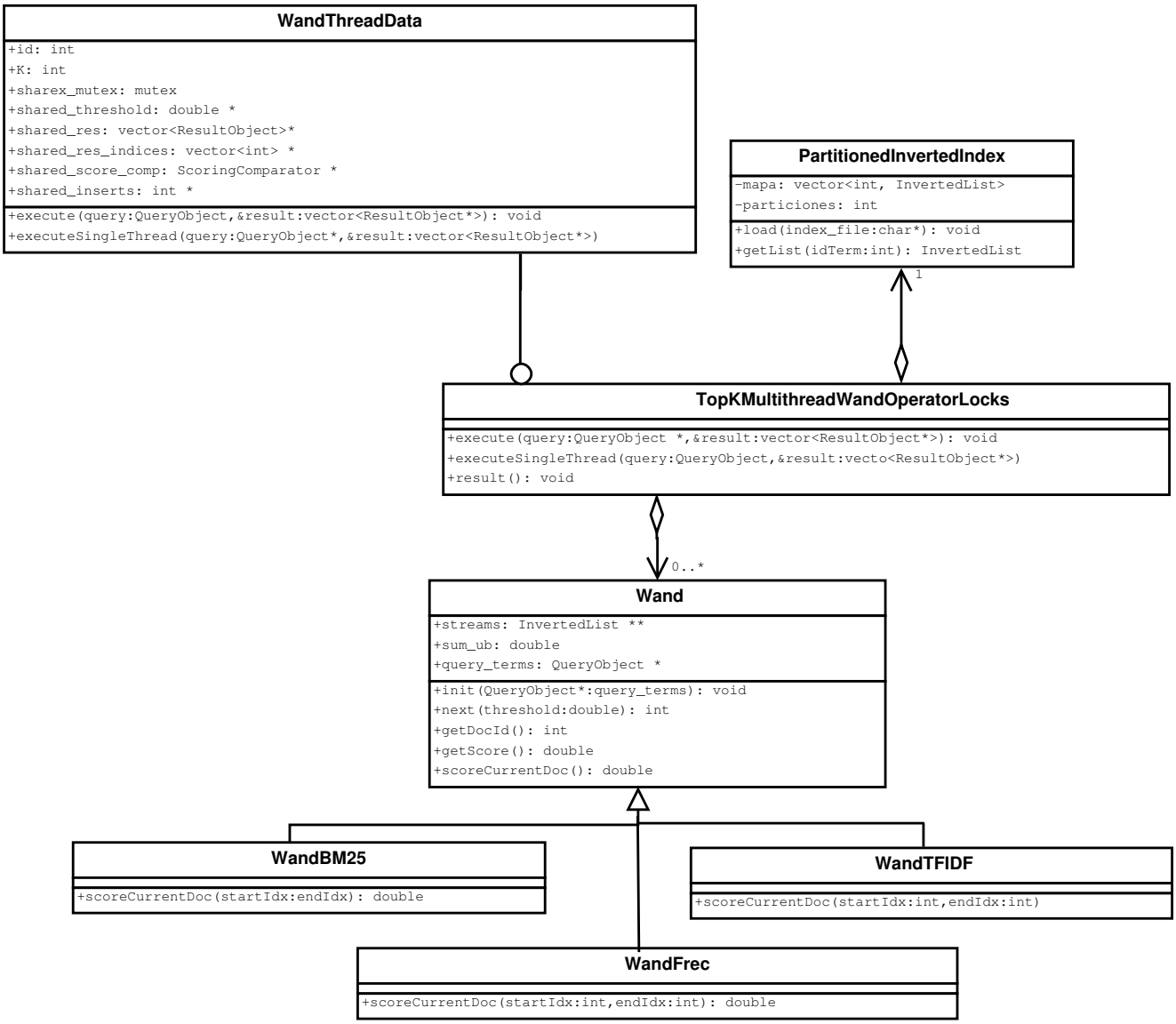

FIGURA 3.5: Diagrama de clases para el esquema SH

3.3 BLOCK MAX WAND

Recordar que en el método de Wand para descartar documentos y encontrar un documento que potencialmente podría estar en el conjunto *top-K*, lo que se hace es usar los *upper bounds* globales de cada lista, es decir, la máxima contribución (puntaje o *score*) de algún documento

de la lista invertida. Además, Wand tradicional es una estrategia DAAT, por lo que por cada lista invertida ocupa un puntero al documento actual que se desea evaluar. Además, usa un método que recibe como entrada un identificador del documento $docID$ y una lista invertida L, y retorna el primer $docID'$ que sea mayor o igual al documento $docID$. A esto se le conoce como movimiento de puntero profundo (*deep pointer movement*) debido a la razón que generalmente implica una descompresión del boque en el que se encuentra el documento.

Sin embargo, como se dijo anteriormente en 2.5.4, usando solo las máximas contribuciones por cada bloque no hará que el método funcione correctamente, puesto que hará que eventualmente se pierdan documentos que podrían estar en el conjunto final de los mejores K documentos. Como ahora se tiene las máximas contribuciones por cada bloque, BMW utiliza otra función la cual recibe como parámetro un identificador de documento $docID$ y una lista invertida. Lo que se hace es mover el puntero actual al correspondiente bloque donde eventualmente se debería encontrar el documento $docID$. A esta función se le conoce como movimiento de puntero superficial (*shallow pointer movement*), por la razón que no involucra una descompresión de bloque. Se debe notar que para que esta función trabaje correctamente se requiere tener almacenada las fronteras de cada uno de los bloques de las listas invertidas.

BMW utiliza dos principales ideas en su diseño: (1) Se usa los *upper bounds* globales para determinar un pivote candidato (como en Wand tradicional), para luego usar los *upper bounds* locales para determinar si es que el pivote candidato es un pivote real o no, y (2) Se intenta siempre utilizar *shallow pointer movement* por sobre *deep pointer movement*.

En el Algoritmo 3.1 se puede apreciar cómo el método *Block-Max-Wand* trabaja. Recordar que todas las listas invertidas poseen un puntero al documento actual que se desea evaluar (*currentDoc*). Lo primero que se hace es ordenar en orden creciente las listas invertidas de acuerdo a su correspondiente *currentDoc*. La función *findPivot()* es la misma que se utiliza en el método Wand tradicional (2.5.3), se itera sobre las listas invertidas y se retorna la posición de la lista en donde se cumple que la suma de los *upper bounds* globales es mayor al *threshold*

(θ). Luego la función *NextShallow()* se encarga de avanzar los punteros de las listas invertidas al inicio del bloque que debería contener el documento d. Posteriormente la función *isRealPivot()* verifica si es que el pivote p encontrado es un pivote real o no, para cada una de las listas desde la posición 0 hasta la posición p, se suma los *upper bounds* de los bloques en donde se encuentran los punteros (recordar que con *NextShallow()* los punteros de las listas quedaron apuntando a los bloques en donde se debería encontrar el documento d), si la suma es mayor al threshold entonces retorna verdadero, de lo contrario retorna falso. El método *scoreDoc()* calcula el puntaje del documento que se le pasa por parámetro.

Cuando el método se da cuenta que p no es un pivote real, lo que se hace es buscar un nuevo candidato a través de la función *getNewCandidate()*, la cual hace avanzar los punteros de las listas invertidas hasta el bloque siguiente que contenga el mínimo $docID$. Para ver explicar de mejor manera esta idea se presenta la Figura 3.6, aquí se puede ver que el documento 4868 es el pivote, cuando este documento no es un pivote real (la función *isRealPivote* retorna falso), lo que se hará es escoger un documento d' tal que $d = min(d1, d2, d3, d4)$ en donde $d1, d2, d3$ son la frontera del bloque actual más uno (inicio del bloque siguiente) y $d4$ es el *currentDoc* de la cuarta lista. Notar que para hacer un descarte seguro de documentos, siempre se debe incluir a la elección del nuevo candidato el *currentDoc* de la lista inmediatamente siguiente a la lista pivote (en este caso 9009).

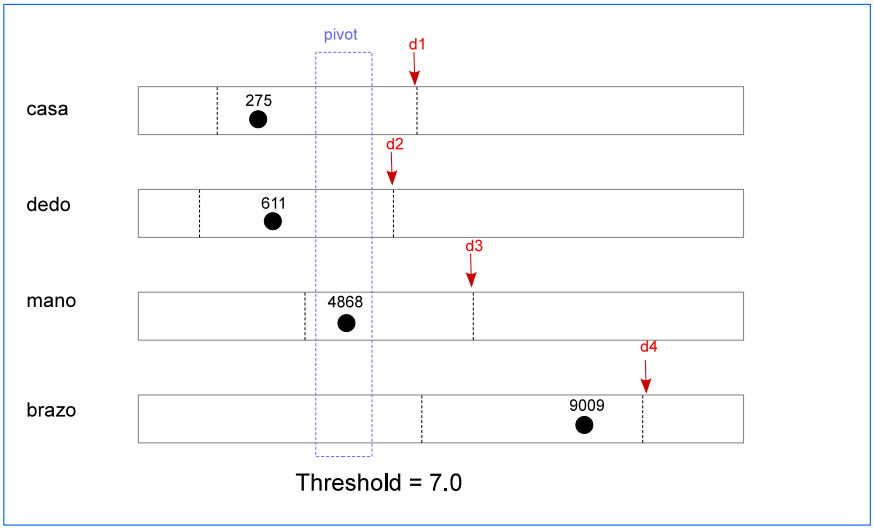

FIGURA 3.6: Ejemplo de cómo opera la functión getNewCandidate()

Algoritmo 3.1: *$BMW(\theta, L, docID)$: Block Max Wand*

Entrada: Un *threshold* θ, listas invertidas L de los términos en la consulta
Salida: $docID$, si existe un documento $docID$ tal que $score(docID) \geq \theta$. de lo contrario END-OF-FILE

```
1:  while true do
2:    Sort(L);
3:    p = findPivot(L, θ);
4:    d = L[p] → currentDoc;
5:    if d == END-OF-FILE then
6:      break;
7:    end if
8:    for i = 0...p do
9:      NextShallow(d, L[i]);
10:   end for
11:   if isRealPivot(θ, p); then
12:     if L[0] → currentDoc == d then
13:       scoreDoc(d, p);
14:       for i = 0...p do
15:         Next(d + 1, L[i]);
16:       end for
17:     else
18:       while List[p − 1] → currentDoc == p do
19:         p = p − 1;
20:       end while
21:       for i = 0...p do
22:         Next(d, L[i]);
23:       end for
24:     end if
25:   else
26:     d' = getNewCandidate();
27:     for i = 0...p do
28:       Next(d', L[i]);
29:     end for
30:   end if
31: end while
```

CAPÍTULO 4. PREDICCIÓN DE RENDIMIENDO DE TRANSACCIONES DE LECTURA

Lograr bajos tiempos de respuestas es uno de los objetivos principales en el diseño de un motor de búsqueda, ya que de esta forma se le puede entregar una respuesta oportunda al usuario. Además estos poseen acuerdos de nivel de servicio (SLA), por ejemplo, que el 99 % de las consultas sean respondidas en $100ms$. Por lo tanto, aquellas transacciones de lectura que requieren una gran cantidad de tiempo para ser resueltas degradan considerablemente la satisfacción del usuario, y es por esto que las máquinas de búsqueda están optimizadas para reducir el percentil más alto de los tiempos de respuesta (también llamado *tail latency*). Paralelizar el procesamiento de cada consulta es una solución promitente para reducir el tiempo de ejecución (Jeon et al., 2013; Tatikonda et al., 2011). Esto es posible con los modernos servidores que existen hoy en día que poseen múltiples núcleos, en donde se puede resolver una consulta paralelizando múltiples hilos de ejecución, reduciendo el tiempo de ejecución de esta.

Conocer de antemano la eficiencia de una *query* es una ventaja muy importante, puesto que aquellas consultas que tomaran una mayor cantidad de tiempo en ser resueltas se les puede asignar un mayor número de *threads* para procesarla, de esta manera se reduce el tiempo de procesamiento de las consultas y se cumple con la cota superior de tiempo prometida al usuario. Adicionalmente, que un sistema de recuperación de la información como un motor de búsqueda conozca anticipadamente cuánto tardará una consulta en ser procesada, permite implementar técnicas efectivas de planificación de transacciones de lecturas, por ejemplo, en el contexto de procesamiento paralelo de *queries* por lotes (*batches*) se pueden crear grupos de consultas que posean tiempos de respuesta parecidos, así se tiende a disminuir tanto el desbalance de carga entre los procesadores como el tiempo en procesar el *batch* completo.

Existen trabajos en donde es estudia la correlación de algunos estadísticos presentes en

las listas del índice invertido con el tiempo de respuesta de una transacción de lectura (citar trabajos que estudian los estadísticos). El más intuitivo es el número de documentos que hay en una lista invertida, mientras más larga es una lista invertida mayor es el tiempo que toma en ser resuelta. A continuación se presenta los métodos de predicción de eficiencia de una transacción de lectura implementados.

4.1 MÉTODO DE PREDICCIÓN GLASGOW

Este método se ocupa para predecir el tiempo de respuesta de una transacción de lectura, está basado en una regresión lineal múltiple con 42 características independientes. Como la respuesta a una consulta debe ser rápida, los estadísticos obtenidos desde las listas invertidas de los términos son previamente calculados en la fase de indexamiento, y en ningún caso es parte del proceso de resolución de la consulta. Los puntajes de los documentos son obtenidos mediante el método de *ranking BM25*.

Es importante señalar que de los 42 estadísticos utilizados en la regresión tienen relación lineal independiente con el tiempo de respuesta de una consulta y que son solo 14 estadísticos los que deben ser extraídos desde las listas invertidas, ya que los 42 estadísticos se obtienen aplicando funciones de agregación. El estudio y el análisis estadístico de cada una de las variables involucradas en la regresión y su impacto en el tiempo está disponible en (Macdonald et al., 2012; **?**; He & Ounis, 2004). A continuación se describe cada uno de los estadísticos $s(t)$ calculados en el proceso de indexamiento (Croft et al., 2009) de un sistema de recuperación de la información.

Media aritmética. Se calcula la media aritmética del puntaje de los documentos.

Media geométrica. Se calcula la media geométrica del puntaje de los documentos.

Media harmónica. Se calcula la media harmónica del puntaje de los documentos.

Máximo puntaje. Se obtiene el puntaje máximo perteneciente a algún documento dentro de la lista invertida. En otras palabras, se obtiene el *upper bound* UB_t de la lista.

Varianza del puntaje. Se extrae la varianza de puntaje de los documentos desde la lista invertida del término t.

Número de documentos. Se calcula el largo de la lista invertida.

Número de maximos. Se obtiene el número de veces en que aparece un puntaje máximo, es decir, el número de veces en que se actualiza el puntaje máximo.

Número de documentos mayor a la media. Se extrae el número de documentos que sobrepasa en puntaje al puntaje promedio.

Número de documentos con puntaje máximo. Se calcula el número de documentos que tienen el puntaje máximo dentro en la lista invertida del término t.

Número de documentos dentro del 5 % más alto. Se obtiene el número de documentos cuyos puntajes están dentro del 5 % superior de la lista invertida.

Número de documentos dentro del 5 % del umbral (*threshold*). Se calcula el número de documentos cuyos puntaje están dentro del 5 % superior o inferior al umbral. Recordar que el *threshold* es el puntaje de documento más bajo dentro del conjunto de *top-K*.

Número de inserciones en el conjunto de los mejores K documentos. Para obtener este estadístico se asume que el término t es una consulta con un solo término, se resuelve esta *query* con el método *Wand* y se calcula el número de inserciones de

documentos que se hizo al *heap*. Recordar que las inserciones al *heap* ocurren cuando el puntaje completo del documento supera el puntaje más bajo que hay en el *heap* en ese momento (umbral o *threshold*).

Frecuencia inversa de documento del término. Se calcula el *idf* del término t.

Tiempo en ser procesado el término. Se obtiene el tiempo que toma ser procesado el término como una *query* de un solo término.

Los 14 estadísticos descritos anteriormente son la base para la implementación del predictor y estos son calculados por cada término del índice invertido. Adicionalmente se definen tres funciones de agregación que se usará por *query*: máximo, varianza y suma. El proceso es el siguiente: Para cada consulta que llega al sistema, se toman los 14 estadísticos de cada uno de los términos que la conforman, posteriormente se aplica las funciones de agregación a los estadísticos de los términos. Por ejemplo, suponga que llega dos consultas al sistema q_1 y q_2, ambas tendrán asociada un vector de 14 estadísticos E_{q_1} y E_{q_2} respectivamente, las funciones de agregación para el estadísticos de la media aritmética será calculado como sigue: $e_1 = maxE_{q_1}(0), E_{q_2}(0), e_2 = varE_{q_1}(0), E_{q_2}(0), e_3 = sumE_{q_1}(0), E_{q_2}(0)$. De esta forma, con solo el primer estadistico se obtiene tres variables independientes. Si se extrapola a cada estadisticos se obtienen los 42 requeridos por el metodo.

CAPÍTULO 5. ESTRATEGIAS DE PLANIFICACIÓN DE QUERIES

Los motores de búsqueda verticales son sistemas dedicados a un solo propósito e ideado con el propósito de lidiar con cargas de trabajos dinámicas. Un ejemplo de un motor de búsqueda vertical es un motor de publicidad que ejecuta una consulta cada vez que un usuario abre un correo electrónico en por ejemplo, el servicio de *Yahoo! mail*; de esta forma se muestra publicidad de acuerdo al contenido del correo electrónico. Eventualmente millones de usuarios concurrentes están conectados a sus correos electrónicos, por lo que la carga de trabajo esperada para el motor de búsqueda puede llegar a órdenes de las cien mil consultas por segundo (Gil-Costa et al., 2013). Adicionalmente, el hecho que las actualizaciones en un motor de búsqueda vertical ocurran con mayor frecuencia que en uno de propósito general, hace que el diseño de los algoritmos para procesar las *queries* sea diferente; también se debe permitir la actualización del índice invertido.

Por lo anteriormente mencionado, se hace imperioso tener un sistema diseñado que soporte altas cargas de trabajo, y las respuestas a consultas esten en una cota de tiempo aceptable para el usuario sin mermar la calidad de los resultados obtenidos. También es necesario que las estructuras de datos y los algoritmos implementados soporten la concurrencia entre las transacciones de lecturas y escrituras; dicho de otra forma, eventualmente el motor de búsqueda tendrá que dejar de procesar consultas para poder servir las transacciones de escritura que actualizan el índice invertido.

A continuación se muestra las diferentes estrategias de planificación de transacciones de lectura abordadas en el presente trabajo utilizando diferentes enfoques; se presentan tres enfoques diferentes: El primero consiste en crear bloques de consultas en donde previamente a cada una de ellas se le asigna el número de hebras que utilizará en su resolución, luego el

bloque es procesado en paralelo por los diferentes hilos de ejecución asignados; El segundo enfoque corresponde a unidades de trabajos, en la que a cada *query* se le asigna un número determinado de unidades de trabajo y los *threads* compiten por ellas desde una cola; (3) El tercer enfoque y último es el más básico, cada hilo de ejecución se hace cargo de una consulta y lleva a cabo su procesamiento, en este enfoque la competencia entre los hilos de ejecución es por las consultas.

5.1 ESTRATEGIAS POR BLOQUES

Un sistema de planificación de un motor de búsqueda trabaja en un contexto *online*, esto significa que desconoce las transacciones que vendrán en el futuro y que cuando llega una nueva transacción de lectura, se debe tomar una decisión rápida acerca de qué hacer con ella. Adicionalmente, una transacción de lectura debe ser resuelta dentro de una cota superior de tiempo, al cual llamaremos t_{limite}. En el contexto del presente trabajo, para que el planificador tome una decisión con respecto a una *query*, debe conocer de ella (1) su tiempo de ejecución y (2) el número de hebras con los que será resuelta. El tiempo de ejecución de cada consulta se obtiene utilizando los métodos de predicción de tiempos mostrados en el Capítulo 4; una vez que se predice el tiempo esperado $t_{esperado}$ de cada *query* para 1,2,4,8 y 16 *threads*, se asigna el numero de hilos de ejecución tal que se cumpla que $t_esperado < t_limite$, de esta forma se satisface la condición de que todas las consultas deben ser resueltas en una cota superior de tiempo previamente definida.

Bajo el contexto de un motor de búsqueda en el que se debe planificar transacciones de lecturas que eventualmente serán resueltas de forma paralela por diferentes hilos de ejecución,

existe una estrategia teórica llamada FR que aborda este problema (Ye & Zhang, 2007) y se adapta a nuestro escenario de un motor de búsqueda vertical; esta estrategia del estado del arte da pie para que en el presente trabajo de tesis se proponga dos nuevas estrategias siguiendo el mismo enfoque de FR, pero estas enfocadas principalmente en mejorar la asignación de consultas a bloques, para así reducir el tiempo ocioso de las hebras.

En la Figura 5.1 se muestra el proceso completo del presente enfoque; las consultas que llegan al sistema las recibe el planificador *scheduler* y las envía al estimador (*estimator*), que calcula el número adecuado de hilos de ejecución para la consulta tal que la consulta sea resuelta en un tiempo inferior al tiempo límite t_limite. Una vez que a la *query* se le predice el tiempo de ejecución y el número de *threads* a utilizar, este planifica la consulta en algún bloque correspondiente depenediendo de la política que se este utilizando. A continuación se muestran las diferentes estrategias propuestas.

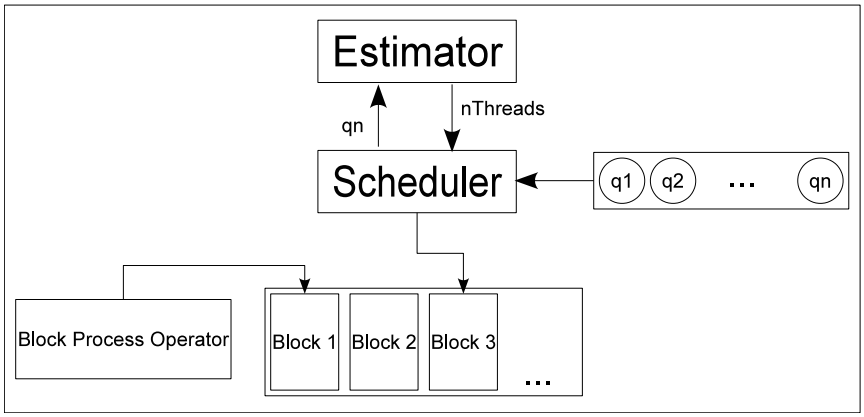

FIGURA 5.1: Enfoque de planificación para estrategias por bloques

5.1.1 Estrategia FR

La estrategia FR posee como requisito que cada una de las consultas a planificar se le haya asignado el número de hebras con las cuales se resolverá; esto se hará siguiendo el esquema 5.1. Como se dijo anteriormente, a cada consulta se asignará la cantidad mínima de hebras tal que el tiempo en resolver la consulta sea menor al tiempo límite t_limite.

Por lo explicado en el párrafo anterior, el algoritmo FR asume que cada *query* que llega al motor de búsqueda posee el número de hebras que debe utilizarse en su resolución. Utilizando esta información, la estrategia hace una clasificación de cada consulta entre *Big* y *Small*, con el objetivo de crear estructuras de datos denominadas *Rooms* y Walls, en donde cada *Wall* y cada Room estará formado solo por consultas de tipo *Big* y *Small* respectivamente. Ambas estructuras de dato tienen un número máximo de máquinas disponibles para procesar las transacciones de lectura. Una *query* es *Big* si el número de máquinas requeridas para procesarla es m (siendo m el número de máquinas disponibles en el sistema), de lo contrario la *query* es *Small*. La idea del algoritmo es crear bloques de consultas (*Wall* y *Room*), que serán procesadas en paralelo por el proceso que resuelve las consultas.

Como se puede ver en el Algoritmo 5.1, cuando una nueva *query* llega al sistema, esta se analiza si es de tipo *Big* o *Small*; esto se hace en el método *isBig()*, que retorna verdadero si es que el número de máquinas requeridas para procesar la consulta es igual al máximo de máquinas disponibles en el sistema, de lo contrario retorna falso y la transacción es clasificada como *Small*. Si la consulta es *Big*, entonces se crea una estructura de dato *Wall*, se planifica la *query* en el bloque y esta se inserta en la lista de planificación *SchedulingList* que contendrá todos los bloques con las consultas ya planificadas. Si se está en presencia de una transacción de lectura de tipo *Small*, se busca algún bloque de tipo *Room* para planificarla; para planificar esta consulta, el bloque debe satisfacer dos condiciones: (1) No debe estar completo, es decir, debe tener threads disponibles, y (2) No debe haber sido procesado aún. Finalmente, si eventualmente no

se encuentra algún bloque disponible para planificar la *query*, entonces se crea un nuevo bloque *Room*, se planifica la consulta al bloque y este bloque es insertado en lista de *scheduling*. Cabe destacar que se dice que un bloque está abierto (*isOpen()*) cuando las consultas presentes en el bloque no han ocupado todos los hilos de ejecución disponibles o cuando el proceso de ejecución ya ha procesado el bloque. Es importante también notar que las estructuras de datos de tipo *Wall* estarán formadas solo por una transacción de lectura de tamaño máximo.

Algoritmo 5.1: *schedulerFR* :: *assignQuery*(L, Q)*: Planificación de consulta*

Entrada: Una SchedulingList L en donde se hará la planificación, QueryObject Q a planificar
Salida: SchedulingList L con la nueva query planificada

```
1: if isBig(query) then
2:    block = newWall();
3:    block → addQuery(query);
4:    L → addBlock(block);
5: else
6:    asignada = false;
7:    for i = L → firstOpenBlockLocked()...L → size() do
8:       room_block = L → getBlockLocked(i);
9:       if (room_block → isOpen())&&(room_block → freeThreads() >= query →
         getThreads()) then
10:         room_block → addQuery(query)
11:         asignada = true
12:         break;
13:      end if
14:   end for
15:   if !(asignada) then
16:      block = newRoom();
17:      block → addQuery(query);
18:      L → addBlockLocked(block);
19:   end if
20: end if
```

En la Figura 5.1 se presenta un ejemplo de ejecucion la estrategia FR. Ya han llegado al sistema consultas: q0 (2 threads), q1 (16 threads), q2 (4 threads), q3 (2 threads), q4 (2 threads), q5 (8 threads) y q6 (16 threads). Se puede ver cómo se van formando las estructuras de datos llamadas *Rooms* y *Walls*. Suponer que eventualmente arriba al sistema una nueva consulta que

será resuelta con 8 *threads*, entonces el algoritmo verifica en primera instancia la $Room_0$, sin embargo, en esta estructura no hay suficientes *threads* disponibles para procesar la consulta (posee solo 6 disponibles). Finalmente la planifica en la $Room_1$.

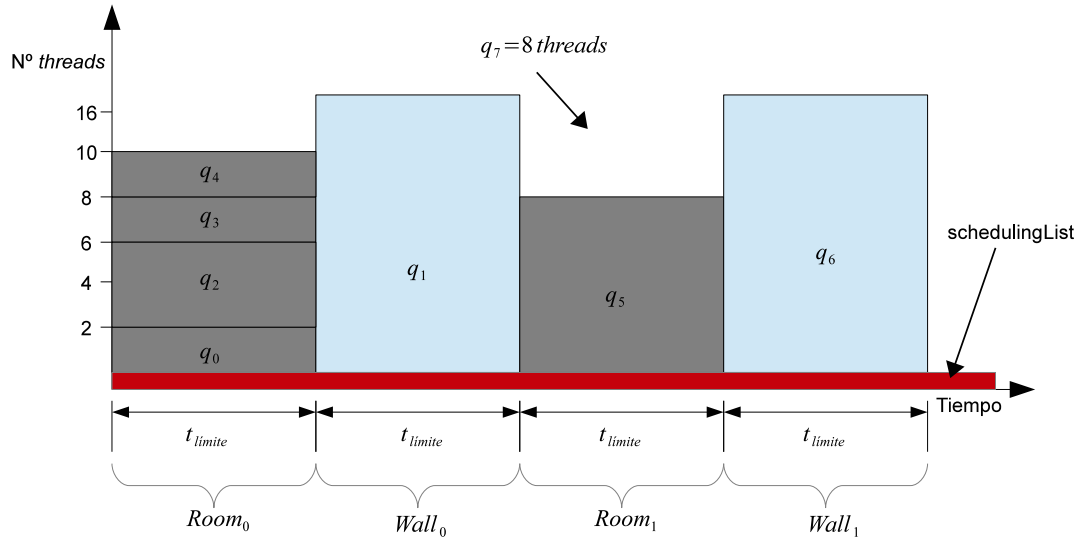

FIGURA 5.2: Ejemplo de procesamiento de la estrategia FR

5.1.2 Estrategia Times

Se intuye que una de las desventajas de la estrategia FR es que al planificar una consulta en algún bloque, solo verifica si es que este posee el número de threads desocupados suficientes tal que sea mayor o igual al número de threads requerido por la consulta. Esto puede generar pérdida de eficiencia importante en los procesamientos de los bloques, puesto que algunas transacciones de lecturas pueden tomar mayor tiempo en ser procesadas y los hilos de ejecución que ya han terminado su trabajo estarán ociosos esperando por otros para continuar con el siguiente bloque. Para abordar esta posible pérdida de eficiencia, se diseña una política de

planificación alternativa en donde además de tomar en cuenta el número de threads disponible en cada bloque (como en la estrategia anterior), se toma en cuenta el tiempo esperado de la transacción de lectura.

Cada consulta q tiene asociado un número de threads NT_q y un tiempo $t_predicho$, que es el tiempo en que se espera que la consulta sea resuelta con NT_q hilos de ejecución. La idea de esta estrategia es separar las transacciones de lectura que tengan tiempos de procesamiento muy diferentes en bloques distintos, es decir, se crearán bloques con consultas que tengan poca diferencia de tiempo unas de otra, de esta forma se quiere reducir el tiempo que se podría perder entre un bloque y otro por el desbalance de carga de los *threads*. Cada bloque B tendrá un tiempo t_B, que será el tiempo de la consulta con menor tiempo dentro del bloque. La métrica establecida para que una transacción de lectura que llega al sistema sea planificada en un bloque, es que el tiempo del bloque t_B no sea el doble del tiempo de la consulta t_q entrante, y viceversa. Si esta condición falla, entonces significa que la consulta q que se está intentando planificar posee tiempos que se escapa a los rangos de tiempo del bloque B.

El Algoritmo 5.1.2 muestra el funcionamiento de la estrategia *Times*, esta recibe como entrada la consulta a planificar y la lista de bloques (*SchedulingList*). El algoritmo Times trabaja de manera similar a la estrategia FR, la diferencia es que en esta estrategia una consulta puede ser planificada en un bloque siempre y cuando este tenga *threads* disponibles suficientes para procesarla y que el tiempo de la consulta no doble al tiempo mínimo dentro del bloque perteneciente a alguna *query* ya planificada; si esta no puede ser planificada, entonces el bloque se desecha y se busca por otro bloque. Existirá un número limitado de bloques que se pueden desechar ($MAX\_BLOCKS\_CHECKED$), si eventualmente se llega a este valor, se escoge aquel bloque con la mínima diferencia de tiempo con la consulta. En el peor de los casos, ningún bloque tendrá espacio suficiente para planificar la consulta y se deberá crear uno nuevo. Notar que en esta estrategia ya no se clasifican las queries de acuerdo al número de *threads* que utilizan.

Algoritmo 5.2: *schedulerTimes* :: *assignQuery*(L, Q)*: Planificación de consulta*

Entrada: Una SchedulingList L en donde se hará la planificación, QueryObject Q a planificar
Salida: SchedulingList L con la nueva query planificada

```
1:  blocks_viewed = 0
2:  blockValid = false;
3:  best_diff = INF;
4:  for i = L → firstOpenBlockLocked()...L → size() do
5:     block = L → getBlockLocked(i);
6:     if blockblock → freeThreads() ≥ query → getThreads() then
7:        tiempo_min = block → getMinimumTime()
8:        if block → isSchedulable(query) then
9:           L → addQuery(query);
10:          assigned = true;
11:          break;
12:       end if
13:       blocks_viewed + +;
14:       if blocks_viewed ≥ MAX_BLOCKS_CHECKED then
15:          break;
16:       end if
17:    end if
18: end for
19: if !(assigned)&&(blocks_viewed ≥ MAX_BLOCKS_CHECKED) then
20:    block = newQueryBlock();
21:    block → addQuery(query);
22:    L → addBlockLocked(block);
23: end if
```

En la Figura 5.3 se muestra un ejemplo de la estrategia *Times*, en el que una consulta llega al sistema y debe ser planificada. El estimador utilizado predijo que la transacción de lectura entrante se demorará 135 ms. con 8 *threads*; en otras palabras, 8 hilos de ejecución es el número mínimo con el que se cumple que el tiempo de la consulta (135 ms.) es menor que la cota superior de tiempo (140 ms.). El algoritmo intenta planificar la consulta en primera instancia en el bloque B_0, sin embargo, el tiempo de la consulta (135 ms.) es el doble del mínimo tiempo en el bloque (55 ms.). Posteriormente, el bloque B_1 no posee *threads* disponibles. Finalmente la *query* q_7 es planificada en el bloque B_2, ya que reúne todas las condiciones necesarias explicadas anteriormente en el algoritmo .

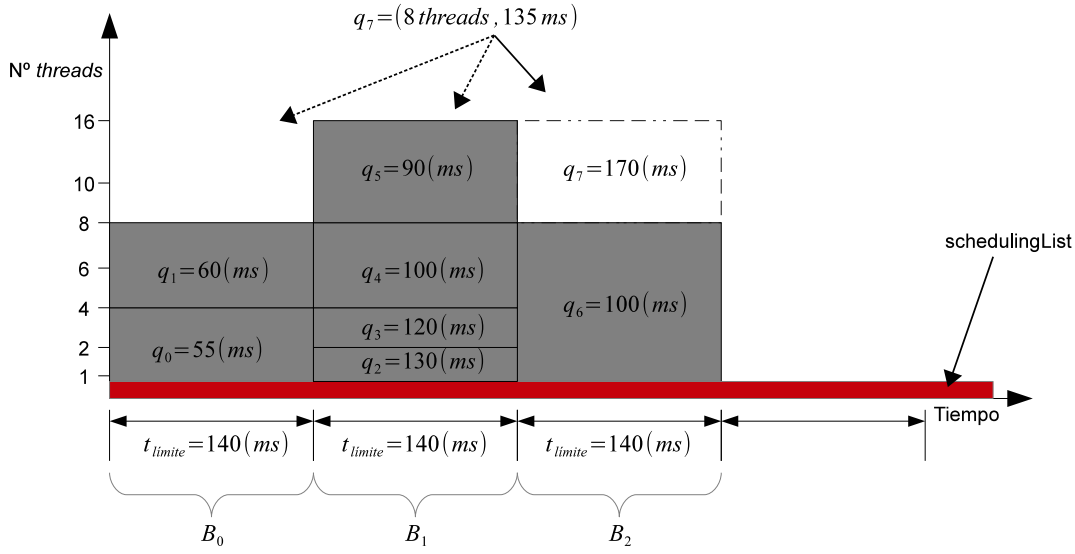

FIGURA 5.3: Ejemplo de procesamiento de la estrategia Times

5.1.3 Estrategia TimesRanges

Esta estrategia también intenta disminuir la posible perdida de eficiencia de la estrategia *FR*. La idea de *TimesRanges* es clasificar y agrupar las consulta de acuerdo a rangos de tiempos; Para planificar una transacción de lectura que arriba al sistema, se debe encontrar un bloque que cumpla con: (1) Número de *threads* libres suficientes, y (2) que el bloque sea del mismo rango de tiempo que la consulta. Inicialmente se define tres tipos de rangos: (1) aquellas que se demoren menos del 10 % del tiempo límite; (2) aquellas que se demoren más (o igual) del 10 % y menos del 25 % del tiempo límite; (3) aquellas que se demoren más (o igual) del 25 % y menos del 50 % del tiempo límite; y (4) aquellas que se demoren más (o igual) del 50 % del tiempo límite. De esta forma reducimos la diferencia de tiempos entre las consultas pertenecientes a un mismo bloque, lo que significa que consultas dentro de un mismo bloque deberían ser resueltas en tiempos muy parecidos. Recordar que el tiempo límite es la cota superior de tiempo en que una consulta debe ser resuelta.

El procedimiento de la estrategias *TimesRanges* se puede ver en el algoritmo 5.3. Para

que una transacción de lectura sea planificada bajo la presente estrategia, lo primero es obtener el rango de tiempo en que se encuentra la consulta entrante; posteriormente, se busca algún bloque que no haya sido procesado y que además posea un número de threads disponibles suficiente para procesar la quer, y se planifica la consulta. Si no se encuentra un bloque que satisfaga las condiciones de la consulta entrante, entonces se crea uno nuevo paa planificarla.

Algoritmo 5.3: *schedulerTimesRanges* :: $assignQuery(L, Q)$*: Planificación de consulta*

Entrada: Una SchedulingList L en donde se hará la planificación, QueryObject Q a planificar
Salida: SchedulingList L con la nueva query planificada

```
1:  range = getQueryRange(query);;
2:  for i = L → firstOpenBlockLocked()...L → size() do
3:     block = L → getBlockLocked(i);
4:     if blockblock → freeThreads() ≥ query → getThreads()&&block_ranges[i] == range
       then
5:        block = L → addQuery(query);
6:        asignada = true;
7:        break;
8:     end if
9:  end for
10: if !(asignada) then
11:    block = newQueryBlock();
12:    block− > addQuery(query);
13:    L → addBlockLocked(block);
14:    block_ranges[L → size − 1] = range;
15: end if
```

5.2 ESTRATEGIA UNIDADES DE TRABAJO

Decir que aquí para planificar se necesita conocer las unidades de trabajos requeridas en vez de threads como en las estrategias por bloques.

Con respecto a los esquemas explicados hasta ahora, el esquema 1TQ tiene la ventaja que no solo requiere menos control, sino que también permite a los hilos de ejecución trabajar sin pausa mientras un *batch* de consultas está siendo procesado. En esta sección se propone un esquema híbrido basado en unidades de procesamiento (*Processing Units*) que aproveche las ventajas de ambos enfoques. (se requiere ver el tema de bloques). En este nuevo esquema de planificación, las consultas pasan a través de una fase en la cual cada *query* es evaluada y se determina un apropiado número de unidades de procesamiento (*processing units*) para poder resolver dicha consulta. Este proceso es llevado a cabo de manera similar al proceso en donde se determina la cantidad de *threads* apropiados para resolver una determina transacción de lectura. Este número de unidades de procesamiento es creado y asociado a cada consulta, finalmente se guarda en una cola de unidades de trabajo. Un conjunto de *threads* consumidores extraen las unidades desde la cola y las procesa independientemente. Cuando un *thread* finalice el procesamiento de la unidad de trabajo actual automáticamente leerá la siguiente unidad de trabajo desde la cola. Generalmente lo que se hace habitualmente es estimar el número de *threads* con el que se resolverá la consulta, como se muestra en la Figura 5.4 en este nuevo enfoque se intenta estimar el número de unidades de trabajo con el que se resolverá cada consulta. Además, se debe controlar el acceso concurrente de los hilos de ejecución a la cola de unidades de trabajo, de tal manera que solo un thread tenga acceso exclusivo a la estructura de datos.

El procesamiento de cada hilo de ejecución es una versión de Wand con heap compartido (SH), adaptado de manera tal que cada unidad de trabajo es resuelta independientemente de si existen otras unidades siendo procesada al mismo tiempo o no. La única excepción es que la unidad que inicializa la consulta es siempre ejecutada antes del resto de las otras unidades de la misma consulta y la entrega de resultados se hace una vez que todas las unidades de trabajo de la *query* han finalizado. Este enfoque híbrido permite reducir el tiempo perdido al final de cada *batch* sin generar una importante pérdida de trabajo mientras las *queries* del *batch* están

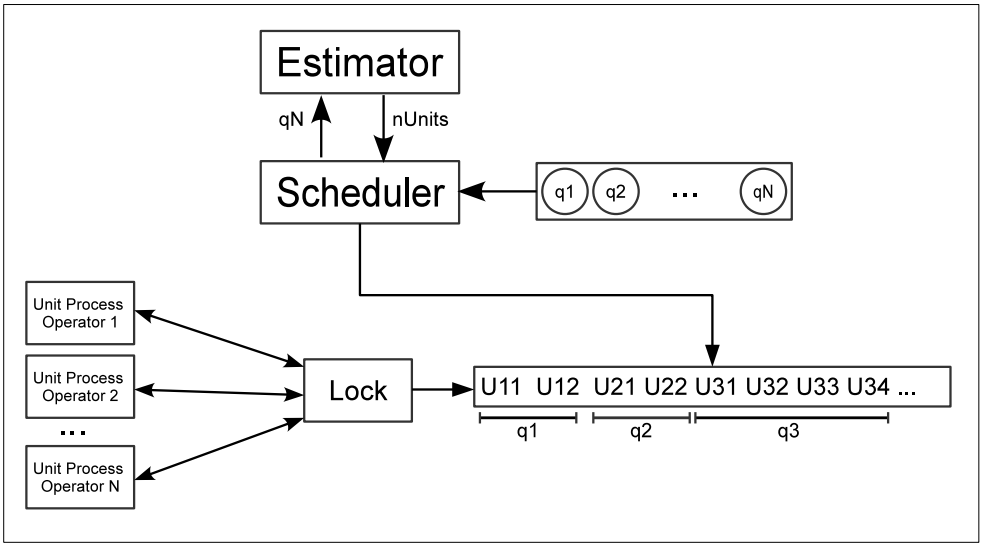

FIGURA 5.4: Procesamiento de consultas utilizando unidades de trabajo

siendo procesadas.

5.3 ESTRATEGIA *1TQ*

Un simple camino para construir un sistema que responda a múltiples consultas simultáneamente usando múltiple hilos de ejecución, es usando estos hilos de manera independiente. Para hacer esto se debe mantener un conjunto de *threads* consumidores que trabajarán en paralelo y se encargarán de resolver las *queries* secuencialmente (una a una) desde una misma cola, esto es lo que en este trabajo se denomina estrategia de Un Thread Por Query (1TQ). En la Figura 5.5 se puede apreciar el esquema de ejecución en donde cada uno de los procesos genera una petición de alguna consulta en la cola, si quedan *queries* por procesar entonces se le asigna al proceso una consulta que tendrá que resolver de manera secuencial. Se debe tener en cuenta que cada vez que un proceso genera una solicitud de *query*, se bloquea

la estructura de datos que contiene las consultas a procesar y luego se procesa la solicitud, de esta forma se asegura un acceso seguro por parte de los distintos *threads*.

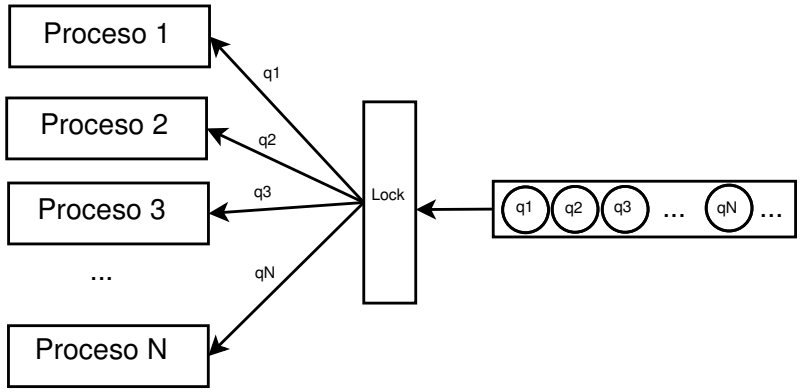

FIGURA 5.5: Ejemplo de procesamiento estrategia 1TQ

Este esquema tiene la ventaja que es simple y fácil de implementar y controlar. Sin embargo, existen sistemas de recuperación de la información como los motores de búsqueda verticales que cuando están ejecutando *batches* de *queries* deben parar su ejecución porque transacciones de escritura han llegado al sistema, y este deben actualizar la información del índice invertido. Solo después de la fase de actualización el sistema es capaz de ejecutar el siguiente *batch* de transacciones de lectura. Al final de cada conjunto de consultas, es posible que algunos hilos de ejecución del sistema finalicen su trabajo y que no tengan más *queries* para procesar, por lo que ellos tienen que esperar que los *threads* restantes finalicen su trabajo antes que el sistema entre en la fase de actualización de su índice invertido o bien, se pase a la ejecución del siguiente *batch* de consultas. Sin embargo, aunque cada hilo de ejecución está secuencialmente ejecutando una transacción de lectura diferente, algunas de estas operaciones puede tomar un tiempo cosiderable, de esta forma se produce una importante pérdida de eficiencia, aunque la intuición nos dice que esto se puede mitigar con *queries* que requieran poca cantidad de tiempo para ser procesada (trabajos pequeños o *small jobs*). En la Figura 5.6 queda reflejado lo dicho en el párrafo anterior. Si los trabajos que cada *thread* está ejecutando son pequeños, entonces probablemente la pérdida de trabajo al final de cada

batch de consultas será menor al trabajo que se pierde cuando los trabajos son grandes (ver Figura 5.7).

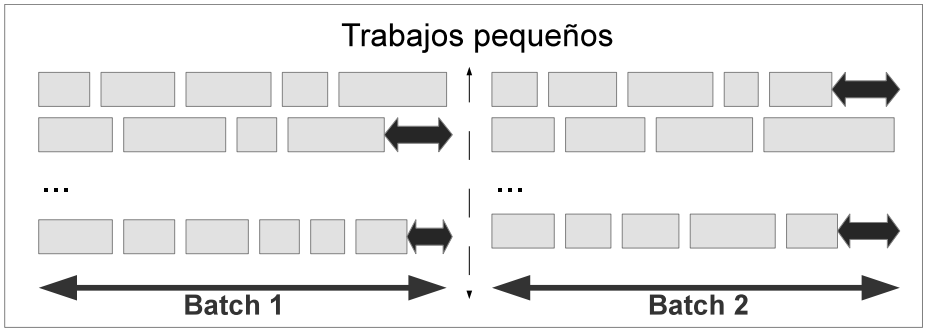

FIGURA 5.6: Ejecución en paralelo de *small jobs*

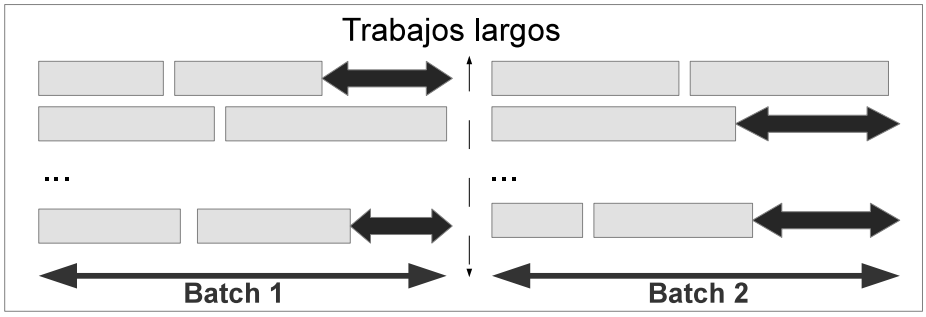

FIGURA 5.7: Ejecución en paralelo de *large jobs*

CAPÍTULO 6. EVALUACIÓN EXPERIMENTAL

En el presente capítulo se presenta los resultados obtenidos de las diferentes implementaciones para los métodos propuestos en la secciones anteriores. Se comienza por la implementación de los métodos de procesamiento de transacciones de lectura, posteriormente se muestra los resultados obtenidos para los diferentes métodos de predicción de tiempos de respuestas para consultas y el comportamiento que tienen para diferente conjunto de datos. Finalmente se presenta el comportamiento de las estrategias de planificación presentadas para diferentes tipos de escenarios.

6.1 HARDWARE Y CONJUNTO DE DATOS

Los experimentos fueron ejecutados en un Intel Xeon E5620 2.4 *Ghz.* con 8 núcleos físicos, tecnología *hyper-threading* y 90 gigabytes de memoria de acceso aleatorio (RAM). Se utilizaron dos conjuntos de datos para llevar a cabo los experimentos, estos son frecuentemente usados por la comunidad del área de Information Retrieval. El primero de ellos es *GOV2*, este conjunto es una colección de aproximadamente 25 millones de páginas *Web* obtenida desde los dominios *.gov* y que pesa 426 *GB*. La otra colección de datos utilizada es la *ClueWeb09*, la cual fue creada para apoyar la investigación en recuperación de la información y las tecnologías relacioanadas con el lenguaje humano, consiste en alrededor de un billón de páginas en diez lenguajes diferentes y 50 millones en inglés. *ClueWeb09* pesa alrededor de 5 *TB* en forma comprimidad y 25 *TB* descomprimida.

6.2 WAND MULTITHREADED

En esta sección se muestra la implementación de dos enfoques para el procesamiento de consultas a través del algoritmo Wand (Broder et al., 2003). El primer enfoque es el esquema de *heap* locales (LH), en el que cada hebra obtiene sus mejores documentos para una consulta dada y luego una hebra maestra se encarga de mezclar todos los resultados de cada uno de los hilos de ejecución para construir el conjunto *top-K* final; el segundo enfoque es el enfoque de *heap* compartido (SH), en el que se tiene un *heap* visible a todos los hilos de ejecución y en donde ellos compiten por el acceso a esta estructura de datos. El detalle del diseño de los enfoques LH y SH están disponibles en 3.1 y 3.2.

6.2.1 Esquema LH

En el esquema LH todos los hilos de ejecución tienen sus propias estructuras de datos y variables que soportan la resolución de una transacción de lectura. La clase *TopKMultithreadWandOperatorLocal* es la encargada de administrar la lógica de ejecución, además prepara las variables e inicia los hilos de ejecución. El Código 6.1 muestra la implementación, en el que existe un método llamado *execute*, este método es el encargado de llevar a cabo la resolución de la consulta, recibe como entrada la consulta a ser resuelta y un vector de resultados, en el que se almacenará los resultados obtenidos. Adicionalmente, este método es el encargado de lanzar las hebras con que se resolverá cada consulta y a cada uno de ellas le asiga un objeto de tipo *TopKWandOperator* ($arr\_ops[pid\_thread]$) para obtener los resultados. Todo este proceso es llevado a cabo usando K como tamaño del conjunto que se

quiere obtener. Además se definen variables como *mapas_ubs*, el cual asocia a cada término los *upper bounds* con los que el método Wand trabajará y *query_partes*, variable que define en cuántas partes la consulta debe ser dividia y está supeditada al número de hebras con que esta será resuelta.

CÓDIGO 6.1: Implementación de la clase TopKMultithreadWandOperatorLocal.h

```
class TopKMultithreadWandOperatorLocal {
    private:
        // Variable que controla el tama o del heap
        unsigned int k;
        // Indice invertido particionado
        PartitionedInvertedIndex *indice;

        // Clase anidada que se utiliza para ordenar los documentos dentro del heap
        class Comparador : public std::binary_function<const ResultObject*,
                           const ResultObject*, bool> {
            public:
                Comparador(){ }
                inline bool operator()(const ResultObject *a, const ResultObject *b){
                    if (a->getScore() == b->getScore()){
                        return a->getDocId() > b->getDocId();
                    }
                    return a->getScore() > b->getScore();
                }
        };

        // Objeto Comparador
        Comparador *comp;


    public:
        // Valor del threshold inicial. Controlara los documentos que deberian
        // ser parte de los top-K.
        double initial_threshold;

        // Arreglo de pthread_t para los hilos de ejecucion
        pthread_t t[max_threads];

        // Arreglo donde se guardaran los identificadores de los arreglos
        int pids[max_threads];

        unsigned int *indices;

        // Arreglo de operadores. Cada uno de los threads tendra un operador (1thread)
        static TopKWandOperator **arr_ops;

        // Vector en donde se guardaran los resultados por thread
        static vector<ResultObject*> **arr_results;

        // Variable que mapea cada termino t a su respectivo upper_bound global
        static map<unsigned int, float> **mapas_ubs;

        // Arreglo de punteros a cada query
        static QueryObject ***query_terms;

         // Partes en la que se dividira cada query
        static unsigned int partes;

        // Constructor
```

```
54          TopKMultithreadWandOperatorLocal(PartitionedInvertedIndex *_indice,
55              map<unsigned int, unsigned int> *_mapa_docs,
56              map<unsigned int, float> **_mapas_ubs,
57              unsigned int _k = 10,
58              unsigned int _max_terms = 128);
59
60          // Destructor
61          ~TopKMultithreadWandOperatorLocal();
62
63          // Metodo que se encarga de resolver la query. Los resultados
64          // quedaran en la variable result
65          virtual void execute(QueryObject *query, vector<ResultObject*> &result);
66  };
```

En la Figura 6.2 se ejemplifica la resolución de una consulta con cuatro hilos de ejecución. Una vez que el sistema asigna el número de hebras que se utilizará para la resolución de la consulta, esta es tomada por el objeto *TopKMultithreadWandOperatorLocal* y hace un llamado al método *execute*, en el que se hace una preparación de variables y se lanzan los hilos de ejecución; cada uno de ellos tiene asignado dos objetos: (1) *TopKWandOperator*, el cual se encarga de que cada hilo de ejecución solo resuelva la parte de la consulta que se le asignó, y (2) *Wand*, el cual se encarga de obtener los mejores K documentos guardándolos en un *heap*.

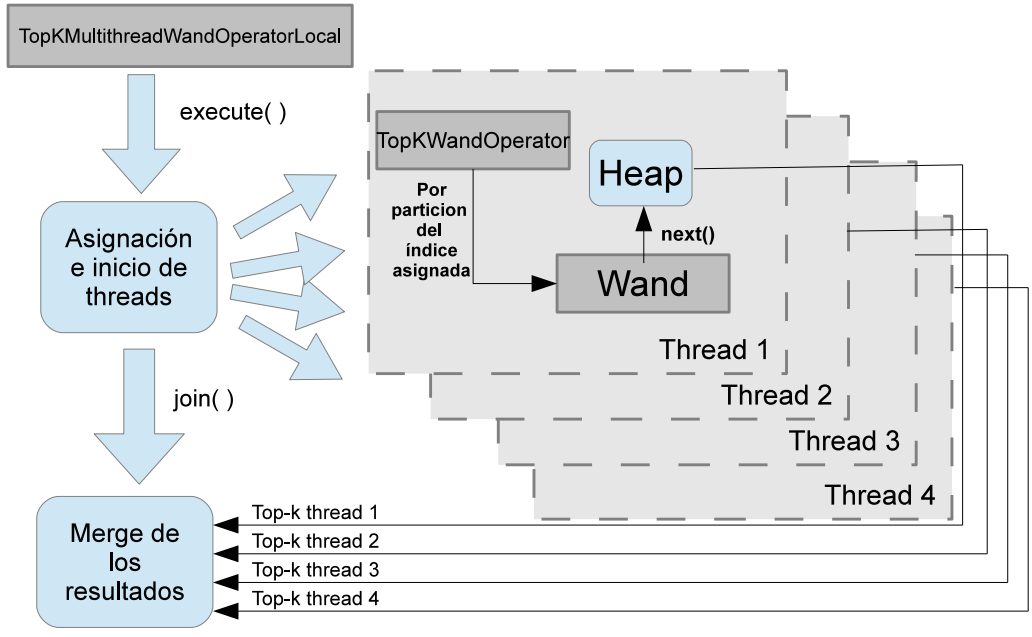

FIGURA 6.1: Esquema de ejecución enfoque LH

6.2.2 Esquema SH

En el esquema SH los hilos de ejecución trabajan con variables compartidas, incluído el *heap* en donde se almacenan los resultados. La ejecución de este enfoque es llevada a cabo por la clase *TopKMultithreadWandOperatorLocks*, la cual podemos ver en el Código 6.2; en esta implementación se puede observar la declaración de una clase anidada la que contiene las variables que serán compartidas por los hilos de ejecución. Dentro de las variables más importantes está el *heap*, el umbral utilizado para decidir si un documento debe estar dentro del *heap* y la variable de tipo *mutex* que permite el acceso exclusivo a las variables compartidas.

CÓDIGO 6.2: Implementación de la clase TopKMultithreadWandOperatorLocks.h

```
#ifndef _TOPK_MULTITHREAD_WAND_OPERATOR_LOCKS_H
#define _TOPK_MULTITHREAD_WAND_OPERATOR_LOCKS_H

#include "TopKMultithreadOperator.h"

#include "Wand.h"
#include "WandBM25.h"

using namespace std;

class TopKMultithreadWandOperatorLocks : public TopKMultithreadOperator{

    protected:
        // Un objeto wand para que cada hebra resuelva la consulta
        Wand **arr_wands;

    public:
        // Clase anidada que administrar  las variables compartidas
        class WandThreadData{
            public:

                WandThreadData(){}

                ~WandThreadData(){
                    wands.clear();
                }

                unsigned int id;
                unsigned int k;

                // Un objeto wand para cada hebra
                vector<Wand*> wands;

                // Controla el acceso concurrente al heap
                mutex *shared_mutex;

                // Umbral que es compartido por todos los hilos
                double *shared_threshold;
```

```
39
40                  // Vector de resultados
41                  vector<ResultObject> *shared_res;
42                  vector<unsigned int> *shared_res_indices;
43
44                  ScoringComparator *shared_score_comp;
45          };
46
47          TopKMultithreadWandOperatorLocks(PartitionedInvertedIndex *_indice, map<unsigned int,
                unsigned int> *_mapa_docs, map<unsigned int, float> **_mapas_ubs, unsigned int _k
                = 10);
48          virtual ~TopKMultithreadWandOperatorLocks();
49
50          // M todo que hara la resoluci n de la query
51          virtual void execute(QueryObject *query, vector<ResultObject*> &result);
52          virtual void executeSingleThread(QueryObject *query, vector<ResultObject*> &result);
53          virtual void reset();
54
55
56  };
57
58
59
60  #endif //_TOPK_MULTITHREAD_WAND_OPERATOR_LOCKS_H
```

La Figura 6.2 muestra un ejemplo de resolución de consulta utilizando cuatro hilos de ejecución y el enfoque SH. Al igual que en el esquema anterior, la clase principal inicializa variables e inicia los hilos de ejecución; el objeto *TopKWandOperator* asignará a cada hebra la parte del índice invertido con la que cada hebra resolverá la consulta. Cada vez que un hilo de ejecución utilizando el objeto Wand encuentre un documento candidato para estar en el conjunto *top-K* final, debe pedir acceso exclusivo a las estructuras de datos involucradas (*heap* y umbral), de esta forma se evita resultados erróneos en el conjunto final producto del paralelismo entre los hilos de ejecución.

6.2.3 Resultados obtenidos

En la Figura 6.3 se puede observar el tiempo promedio del enfoque LH y el enfoque SH en resolver un conjunto de 10,000 consultas de la colección *GOV2*. A medida que crece el número

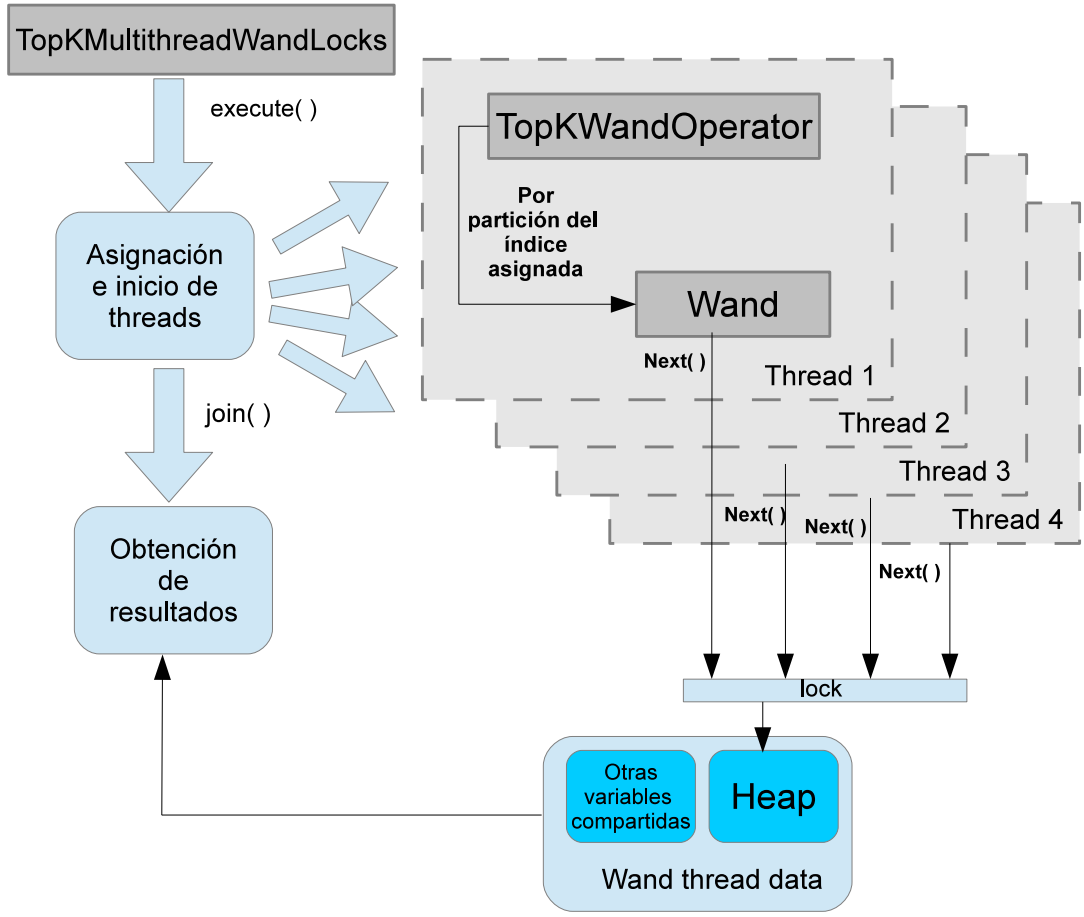

FIGURA 6.2: Esquema de ejecución enfoque SH

de hilos de ejecución, el enfoque de *heaps* compartidos toma ventaja por sobre el enfoque de *heaps* locales, sin embargo, cuando se utiliza un hilo de ejecución se puede observar que LH (117.486ms) requiere un tiempo menor que SH (130.591ms), esto se debe porque en LH no se usa variables compartidas que retrase a los hilos de ejecución esperando a que otros las libere. LH requiere menos tiempo en resolver el *log* de consultas para 2,4,8 y 16 hebras. El esquema LH puede estar muy supeditado a la distribución de documentos en las listas del índice invertido, ya que si un hilo de ejecución procesa su corrrespondiente parte del índice invertido en donde los mejores puntajes se encuentran al final, entonces el *heap* tendrá un umbral bajo al comienzo del proceso, eso implica un proceso de descarte de documentos menos eficiente y el tiempo de ejecución requerido será mayor, retrasando el proceso que mezcla los resultados para obtener

el conjunto *top-K* final. Como el esquema SH ocupa un solo *heap* para obtener los mejores K documentos, el *heap* tiende a llenarse rápidamente con los mejores documentos globales, esto implica que el puntaje mínimo del *heap* (umbral) tiende a crecer rápidamente, permitiendo un mejor descarte de documentos y menor tiempo de ejecución para las hebras.

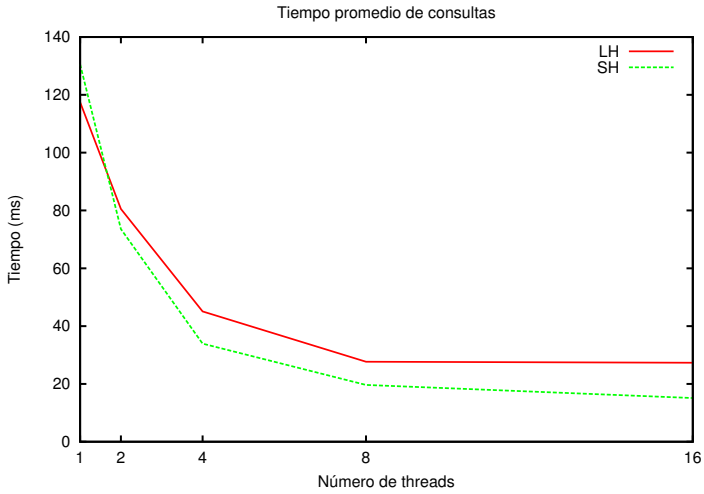

FIGURA 6.3: Tiempos promedios de las consultas

Adicionalmente en la Figura 6.4 se puede ver en forma general que con la estrategia de enfoques compartidos se obtiene mejores eficiencias que con la estrategia LH. Con SH la mejor eficiencia que se obtiene es con 4 hilos de ejecución ($0.962ms$), mientras que con 2 y con 8 hebras se obtiene una eficiencia de 0.887 y $0.831milisegundos$; en general se obtiene buenas eficiencias para 1,2,4 y 8 hebras, sin embargo, con 16 hilos de ejecución la eficiencia baja considerablemente (0.5403) con respecto a las anteriores, esto se debe principalmente a la tecnología thyperthreading de la máquina utilizada. También es interesante ver que el uso exclusivo del *heap* compartido por parte de los *threads* no tiene un fuerte impacto en el rendimiento. La eficiencia baja de LH se debe porque para obtener el conjunto *top-K* final de una consulta debe haber una sincronización de todos los hilos de ejecución involucrados en que cada uno de ellos envíe sus *top-K* locales a la hebra maestra, y además porque existe un costo adicional de calcular el conjunto *top-K* final entre los $P \times K$ documentos seleccionados (siendo

P el número de procesadores).

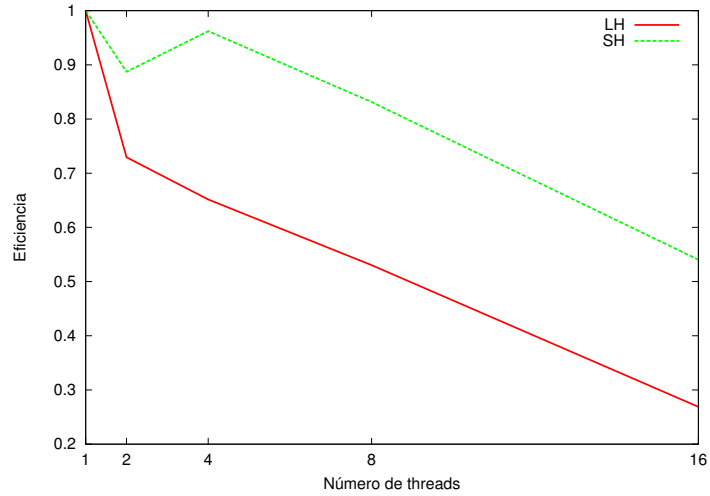

FIGURA 6.4: Eficiencias para Wand con heaps compartido y locales

6.3 PREDICCIÓN DE TIEMPOS

En la Tabla 6.1 se puede apreciar los resultados obtenidos con la regresión lineal múltiple con distintos número de hilos de ejecución. Los experimentos se llevaron a cabo utilizando el conjunto de datos GOV2 y el método Wand tradicional. Se llega a buenos valores del coeficiente de determinación a pesar que son 42 variables independientes; todos los valores del coeficientes están sobre el 80

TABLA 6.1: Resultados obtenidos de la regresión lineal múltiple

	1 thread	2 threads	4 threads	8 threads	16 threads
Rcuadrado	0.7272	0,7428135071	0,7135704064	0,7036605024	0,6990570822
ECM	7242,4204588977	2251,2801134239	475,6894968954	145,2615752321	82,8386825923

Adicionalmente se implementó un modelo de red neuronal backpropagation, en donde se obtuvieron mejores resultados que en el método basado en la regresión lineal múltiple.

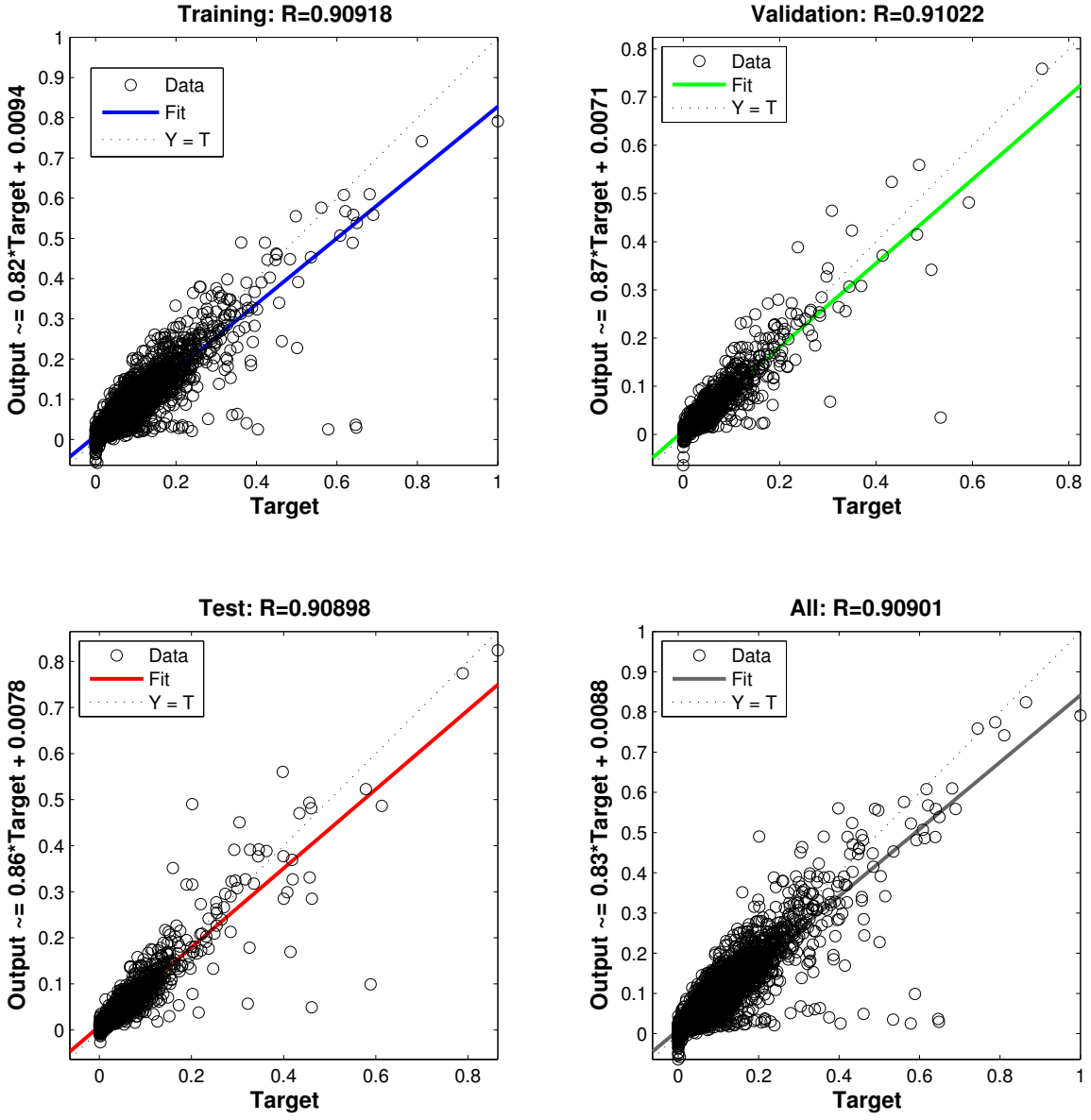

FIGURA 6.5: Regresiones obtenidas por la red neuronal backpropagation

TABLA 6.2: Resultados obtenidos de la red neuronal backpropagation

	1 thread	2 threads	4 threads	8 threads	16 threads
ECM	4593,5493672152	1528,1706707118	335,1918550275	100,0618347161	54,786696642

6.4 ESTRATEGIAS DE SCHEDULING

CAPÍTULO 7. CONCLUSIONES

REFERENCIAS

Albers, S. (2003). Online algorithms: a survey. *Mathematical Programming*, *97*(1-2), 3–26.
URL http://dx.doi.org/10.1007/s10107-003-0436-0

Arroyuelo, D., González, S., Oyarzún, M., & Sepulveda, V. (2013). Document identifier reassignment and run-length-compressed inverted indexes for improved search performance. En *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '13, (pág. 173–182). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/2484028.2484079

Baeza-Yates, R. A., Arenas, M., Gutiez, C., Hurtado, C., Mar M., Navarro, G., Piquer, J., Rodrez, A., Ruiz-del Solar, J., & Velasco, J. (2008). *Cunciona la Web*. Centro de Investigaci la Web.

Baeza-Yates, R. A., & Ribeiro-Neto, B. (2011). *Modern Information Retrieval*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Barroso, L. A., Dean, J., & Hölzle, U. (2003). Web search for a planet: The google cluster architecture. *IEEE Micro*, *23*(2), 22–28.
URL http://dx.doi.org/10.1109/MM.2003.1196112

Blanco, R., & Barreiro, A. (2010). Probabilistic static pruning of inverted files. *ACM Trans. Inf. Syst.*, *28*(1), 1:1–1:33.
URL http://doi.acm.org/10.1145/1658377.1658378

Borodin, A., & El-Yaniv, R. (1998). *Online Computation and Competitive Analysis*. New York, NY, USA: Cambridge University Press.

Broccolo, D., Macdonald, C., Orlando, S., Ounis, I., Perego, R., Silvestri, F., & Tonellotto, N. (2013). Query processing in highly-loaded search engines. En O. Kurland, M. Lewenstein, & E. Porat (Editores) *String Processing and Information Retrieval*, vol. 8214 de *Lecture Notes in Computer Science*, (pág. 49–55). Springer International Publishing.
URL `http://dx.doi.org/10.1007/978-3-319-02432-5_9`

Broder, A. Z., Carmel, D., Herscovici, M., Soffer, A., & Zien, J. (2003). Efficient query evaluation using a two-level retrieval process. En *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, CIKM '03, (pág. 426–434). New York, NY, USA: ACM.
URL `http://doi.acm.org/10.1145/956863.956944`

Buckley, C., & Lewit, A. F. (1985). Optimization of inverted vector searches. En *Proceedings of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '85, (pág. 97–110). New York, NY, USA: ACM.
URL `http://doi.acm.org/10.1145/253495.253515`

Büttcher, S., Clarke, C., & Cormack, G. V. (2010). *Information Retrieval: Implementing and Evaluating Search Engines*. The MIT Press.

Chambers, J. M. (1991). *Statistical Models in S*. Boca Raton, FL, USA: CRC Press, Inc.

Croft, B., Metzler, D., & Strohman, T. (2009). *Search Engines: Information Retrieval in Practice*. USA: Addison-Wesley Publishing Company, 1st ed.

Cronen-Townsend, S., Zhou, Y., & Croft, W. B. (2002). Predicting query performance. En *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '02, (pág. 299–306). New York, NY, USA: ACM.
URL `http://doi.acm.org/10.1145/564376.564429`

Dean, J. (2009). Challenges in building large-scale information retrieval systems: Invited talk. En *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, WSDM '09, (pág. 1–1). New York, NY, USA: ACM.
URL `http://doi.acm.org/10.1145/1498759.1498761`

Dean, J., & Barroso, L. A. (2013). The tail at scale. *Commun. ACM*, *56*(2), 74–80.
URL `http://doi.acm.org/10.1145/2408776.2408794`

Ding, S., & Suel, T. (2011). Faster top-k document retrieval using block-max indexes. En *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, (pág. 993–1002). New York, NY, USA: ACM.
URL `http://doi.acm.org/10.1145/2009916.2010048`

Freire, A., Macdonald, C., Tonellotto, N., Ounis, I., & Cacheda, F. (2012). Scheduling queries across replicas. En *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '12, (pág. 1139–1140). New York, NY, USA: ACM.
URL `http://doi.acm.org/10.1145/2348283.2348508`

Freire, A., Macdonald, C., Tonellotto, N., Ounis, I., & Cacheda, F. (2013). Hybrid query scheduling for a replicated search engine. En *Proceedings of the 35th European Conference on Advances in Information Retrieval*, ECIR'13, (pág. 435–446). Berlin, Heidelberg: Springer-Verlag.
URL `http://dx.doi.org/10.1007/978-3-642-36973-5_37`

Gil-Costa, V., Inostrosa-Psijas, A., Marin, M., & Feuestein, E. (2013). Service deployment algorithms for vertical search engines. En *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, PDP '13, (pág. 140–147). Washington, DC, USA: IEEE Computer Society.
URL `http://dx.doi.org/10.1109/PDP.2013.28`

He, B., & Ounis, I. (2004). Inferring query performance using pre-retrieval predictors. En A. Apostolico, & M. Melucci (Editores) *String Processing and Information Retrieval*, vol. 3246 de *Lecture Notes in Computer Science*, (pág. 43–54). Springer Berlin Heidelberg.
URL http://dx.doi.org/10.1007/978-3-540-30213-1_5

Jeon, M., He, Y., Elnikety, S., Cox, A. L., & Rixner, S. (2013). Adaptive parallelism for web search. En *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, (pág. 155–168). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/2465351.2465367

Jeon, M., Kim, S., Hwang, S.-w., He, Y., Elnikety, S., Cox, A. L., & Rixner, S. (2014). Predictive parallelization: Taming tail latencies in web search. En *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR '14, (pág. 253–262). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/2600428.2609572

Macdonald, C., Tonellotto, N., & Ounis, I. (2012). Learning to predict response times for online query scheduling. En *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '12, (pág. 621–630). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/2348283.2348367

Moffat, A., & Zobel, J. (1996). Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, *14*(4), 349–379.
URL http://doi.acm.org/10.1145/237496.237497

Persin, M. (1994). Document filtering for fast ranking. En *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '94, (pág. 339–348). New York, NY, USA: Springer-Verlag New York, Inc.
URL http://dl.acm.org/citation.cfm?id=188490.188597

Rojas, O., Gil-Costa, V., & Marin, M. (2013). Efficient parallel block-max wand algorithm. En F. Wolf, B. Mohr, & D. an Mey (Editores) *Euro-Par 2013 Parallel Processing*, vol. 8097 de *Lecture Notes in Computer Science*, (pág. 394–405). Springer Berlin Heidelberg.
URL `http://dx.doi.org/10.1007/978-3-642-40047-6_41`

Salton, G., & McGill, M. J. (2003). *Introduction to Modern Information Retrieval*. New York, NY, USA: McGraw-Hill, Inc.

Si, L., & Callan, J. (2002). Using sampled data and regression to merge search engine results. En *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '02, (pág. 19–26). New York, NY, USA: ACM.
URL `http://doi.acm.org/10.1145/564376.564382`

Tatikonda, S., Cambazoglu, B. B., & Junqueira, F. P. (2011). Posting list intersection on multicore architectures. En *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, (pág. 963–972). New York, NY, USA: ACM.
URL `http://doi.acm.org/10.1145/2009916.2010045`

Tonellotto, N., Macdonald, C., & Ounis, I. (2011). Query efficiency prediction for dynamic pruning. En *Proceedings of the 9th Workshop on Large-scale and Distributed Informational Retrieval*, LSDS-IR '11, (pág. 3–8). New York, NY, USA: ACM.
URL `http://doi.acm.org/10.1145/2064730.2064734`

Turtle, H., & Flood, J. (1995). Query evaluation: Strategies and optimizations. *Inf. Process. Manage.*, *31*(6), 831–850.
URL `http://dx.doi.org/10.1016/0306-4573(95)00020-H`

Yan, H., Ding, S., & Suel, T. (2009). Inverted index compression and query processing with optimized document ordering. En *Proceedings of the 18th International Conference on World*

Wide Web, WWW '09, (pág. 401–410). New York, NY, USA: ACM.

URL http://doi.acm.org/10.1145/1526709.1526764

Ye, D., & Zhang, G. (2007). On-line scheduling of parallel jobs in a list. *J. of Scheduling*, *10*(6), 407–413.

URL http://dx.doi.org/10.1007/s10951-007-0032-x

Zobel, J., & Moffat, A. (2006). Inverted files for text search engines. *ACM Comput. Surv.*, *38*(2).

URL http://doi.acm.org/10.1145/1132956.1132959