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

Tesis de Grado presentado en conformidad
a los requisitos para obtener el Grado de
Magíster en Ingeniería Informática

SANTIAGO DE CHILE
2014

AGRADECIMIENTOS

Agradezco a cada uno de los profesores del Departamento de Ingeniería Informática de la Universidad de Santiago de Chile, especialmente al profesor Fernando Rannou, Max Chacón y Felipe Bello por la disposición mostrada hacia mi persona cada vez que requerí de su ayuda.

Quiero agradecer al profesor Edgardo Sepúlveda por los consejos, la dedicación y su excelente voluntad a enseñarme durante mi formación como profesional.

Agradecimiento especial a mis profesores guías Carolina Bonacic y Mauricio Marín por su tiempo y voluntad para guiarme durante el desarrollo de esta tesis; extiendo este agradecimiento especial para la profesora Carolina, por su preocupación que tuvo hacia mi persona cuando tuve problemas.

Agradezco también a Víctor Sepúlveda por el apoyo técnico y por entregarme conocimientos muy importantes a lo largo del desarrollo de esta tesis, sin su apoyo el desarrollo de este trabajo de tesis no hubiese sido el mismo.

Agradezco también a Bárbara Chávez y Carolina Bustos.

Quiero agradecer a mis padres Alex y Angélica, a Verónica Pérez y Jorge Pérez por el apoyo y el cariño mostrado siempre hacia mi persona. También agradecer a las dos personas más importantes en mi vida Rubén Pérez Vera y Flor Magdalena Warner, por haber sido una pieza fundamental en mi vida durante todos estos años.

Finalmente agradezco a Dios por darme la oportunidad de crecer como persona y como profesional.

Para Rubén, Flor, Tía Tita, familia y amigos

RESUMEN

El procesamiento de transacciones de lecturas en motores de búsqueda demanda el uso eficiente de recursos de *hardware* para hacer frente a altas y dinámicas cargas de trabajo por parte de los usuarios. Estos sistemas son generalmente desplegados en grupos de máquinas con múltiples procesadores cada una, de modo que puedan responder múltiples consultas simultáneamente. A medida que la Web crece, los motores de búsqueda toman mayor importancia en la búsqueda de información dentro de grandes cantidades de datos.

En el presente trabajo se abordan diferentes estrategias de procesamiento y planificación de transacciones de lectura, así como también técnicas de asignación de recursos para resolverlas, enfocadas principalmente en (1) el acceso a grandes índices invertidos para obtener el conjunto de los mejores K documentos para una consulta utilizando el algoritmo Wand, y (2) el uso de predictores de eficiencia para transacciones de lectura con el objetivo de reducir el tiempo de procesar lotes de consultas.

Los resultados obtenidos muestran que se puede reducir el tiempo de procesamiento de grandes conjuntos de consultas utilizando métodos que predigan el costo en tiempo de estas y que algunos métodos de aprendizaje pueden llegar a ser muy dependiente de los datos. Además se obtiene que en el contexto de un motor de búsqueda, las técnicas de planificación disponibles en el estado del arte son muy dependientes de la precisión del predictor, por lo cual se propone un enfoque de procesamiento de consultas basado en unidades de trabajo con el que se obtienen mejoras significativas en los tiempos totales de procesamiento.

Palabras Claves: recuperación de información, motores de búsqueda, Wand .

ABSTRACT

Processing queries in Web search engines demands an efficient use of hardware resources to cope with high and dynamics workload of users traffic. These systems are usually deployed on dedicated clusters of multiprocessor servers, so that they can respond multiple queries simultaneously. As the Web becomes bigger, search engines are becoming increasingly important to find information in large amounts of data.

This work discusses different query processing and scheduling strategies, it also studies resource allocation techniques to resolve them, focused mainly on (1) access to large inverted index data structure to obtain the top-K most pertinent results for any query using Wand algorithm, and (2) the use of different query efficiency predictors in order to process batches of queries.

The results show that query efficiency predictors can lead to reduced processing time of large query batches and that some query predictors may become data dependents. This work also shows that, in the context of search engine, state of the art algorithms for query scheduling are very dependent on predictor accuracy, whereby this work presents a query processing technique based on work units that yields significant improvements in total processing times.

Keywords: information retrieval, search engines, Wand .

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE ALGORITMOS

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 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, ya que 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, esta resolución 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*) en los centros de procesamiento de datos. 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 podría resolver transacciones de manera más rápida en períodos de altas cargas de trabajo y luego 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 guía 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 resultados 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 consultas debe explotar el paralelismo.

1.3.1 Características de la solución

La solución propuesta incluye la implementación de un método de predicción de tiempo de respuesta a transacciones de lectura para la asignación eficiente de hilos de ejecución, 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 de contenido

creado en el pasado muy reciente.

1.3.3 Alcances de la solución

En la solución propuesta no se considera un sistema distribuído de procesamiento de consultas en donde existe un costo de comunicación entre las máquinas, sino que la resolución de consultas se llevará a cabo en una máquina distribuyendo la carga en sus unidades de procesamiento.

La solución propuesta solo considera la planificación y procesamiento de transacciones de lectura, debido a que las transacciones de escritura requieren un tratamiento diferente al que se abordará en el presente trabajo.

Las técnicas ocupadas para la extracción de documentos ocupan un número fijo de ellos $K = 100$. Esto significa que todas las estrategias abordadas en este trabajo se utiliza la extracción de 100 documentos.

1.4 OBJETIVOS DEL PROYECTO

1.4.1 Objetivo general

Analizar, desarrollar, comparar 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 en cada operación.

1.4.2 Objetivos específicos

1. Desarrollar algoritmos paralelos de procesamiento de transacciones de lectura que sean eficientes.
2. 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.
3. Obtener estrategias de reordenamiento dinámico de transacciones de lectura que llegan a un procesador con múltiples núcleos.
4. Obtener un modelo de costo que permita analizar la eficiencia y escalabilidad de las estrategias diseñadas en motores de búsqueda verticales.

1.5 METODOLOGÍA Y HERRAMIENTAS A UTILIZAR

1.5.1 Metodología

El presente trabajo posee un enfoque investigativo, en donde en primera instancia se busca en la literatura las estrategias de planificación de carácter teórico que existen, se genera un diseño adaptado al entorno de un motor de búsqueda para ser implementado. Luego estas estrategias son evaluadas, y en base a observación y experimentación, se buscan oportunidades de mejora de ellas, para posteriormente compararlas bajo ciertos parámetros establecidos.

Antes de comenzar la investigación de las diferentes estrategias de planificación, se construye el entorno de un motor de búsqueda, esto implica (1) el diseño, desarrollo y evaluación de diferentes métodos de procesamiento de transacciones de lectura, y (2) implementación y evaluación de métodos predictores de eficiencia para transacciones de lectura.

Bajo dicha perspectiva de trabajo, el método científico es particularmente útil para guiar la totalidad del trabajo. Adicionalmente para las etapas en donde se requiera la construcción de *software*, se utilizará una metodología que consta de cuatro fases: (1) concepción, en donde se estudian las variables involucradas y se establecen los requerimientos principales; (2) elaboración, en la cual se diseña los modelos y componentes involucrados; (3) construcción, en donde se implementa lo definido en la fase de elaboración; (3) transición, etapa en la cual se reúne los resultados obtenidos y se hacen ajustes si es necesario.

1.5.2 Herramientas a utilizar

Las herramientas a utilizar se dividen en herramientas de *software* y *hardware*, a continuación se describe las utilizadas en el desarrollo del trabajo.

1.5.2.1 Herramientas de software

Para la escritura de documento se utilizan las siguientes herramientas:

1. Sistema operativo OS X Yosemite 10.10.
2. Texmaker.
3. Apache OpenOffice 4.0.1.
4. Gnuplot 4.2.

Para el desarrollo de los experimentos:

1. Fedora 13 (goddard).
2. GCC 4.4.5.
3. Editor de texto VIM.

1.5.2.2 Herramientas de hardware

El desarrollo y ejecución se lleva a cabo en una máquina del laboratorio *Yahoo! Research* Santiago de Chile con las siguientes características:

1. Intel Xeon E5620 2.4 GHz, el cual consiste de 8 núcleos físicos y tecnología *hyperthreading*.
2. 90Gb de RAM.

1.6 ORGANIZACIÓN DEL DOCUMENTO

El presente documento se divide en 7 capítulos. En el primer capítulo se presenta la problemática y la solución respectiva; además se muestran los objetivos y la metodología a utilizar. En el segundo capítulo se expone los conceptos teóricos involucrados. Durante el tercer capítulo se aborda los métodos de procesamiento de transacciones de lectura utilizados, explicando en detalle cómo funciona cada uno de ellos y presentando el diseño creado para su posterior implementación. En el cuarto capítulo se muestran los métodos predictores de eficiencia para transacciones de lectura implementados. En el quinto capítulo se presenta el diseño y esquema de ejecución para cada una de las estrategias de planificación abordadas. En el capítulo seis se explica la fase de experimentación y se muestra los resultados obtenidos para las diferentes estrategias de planificación bajo diferentes métodos de procesamiento de consultas y diferentes conjunto de datos. Finalmente en el capítulo siete, se exponen las respectivas conclusiones obtenidas a partir del presente trabajo.

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*, este 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 los términos o 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 los términos que el proceso de *crawling* ha encontrado, asociado a cada uno de ellos

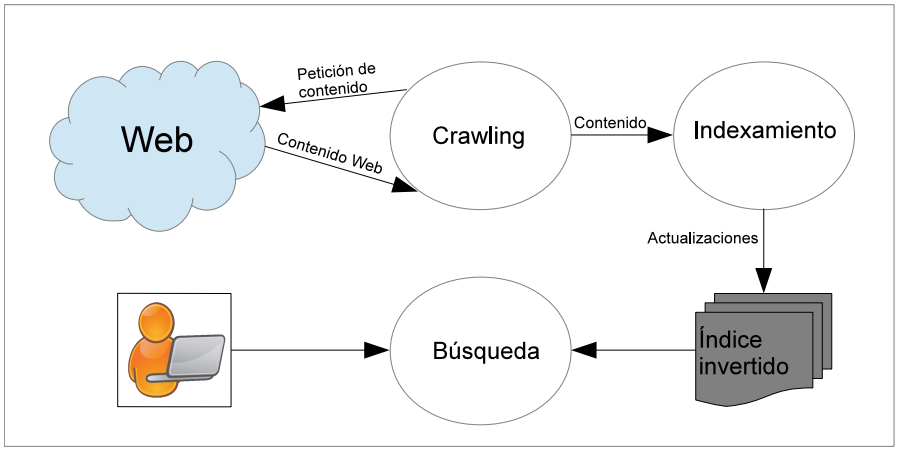


FIGURA 2.1: Arquitectura típica de un motor de búsqueda.

se tiene una lista de todos los documentos en donde el término aparece mencionado (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 los términos 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 esta. 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, como 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 término que existe en un documento, registrando su posición y la cantidad de veces que este 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).

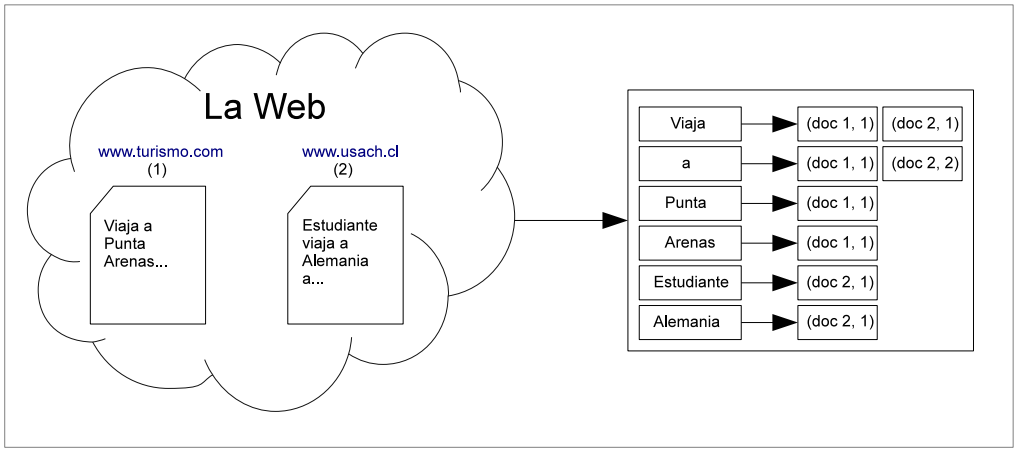


FIGURA 2.2: Índice invertido.

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 uno a uno 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 transacción de lectura con respecto a un documento antes de evaluar el siguiente. 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 paralelismo 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

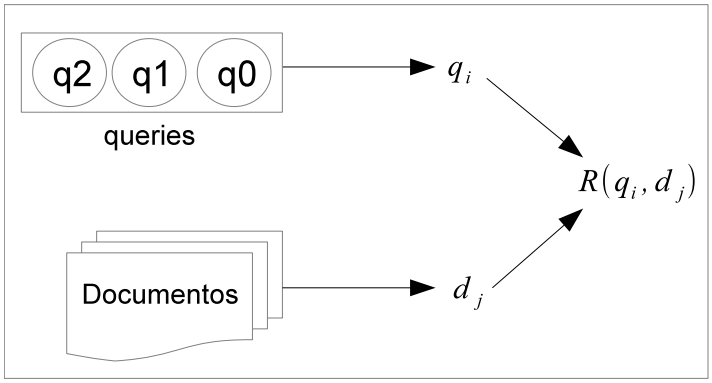


FIGURA 2.3: Proceso de *scoring* de documento.

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 consulta q_i.

Un motor de búsqueda guarda billones de documentos que están formados por términos o palabras, dentro de estos términos no todos poseen la misma utilidad para describir el contenido del documento. Determinar la importancia de estos 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.

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. Este estadístico se divide en dos partes, el primero corresponde a la frecuencia del término en un documento (tf) y que en su versión más sencilla se utiliza la frecuencia bruta del término

t en el documento d ($f(t,d)$) dividido por la frecuencia de la palabra que más se repite en el documento d.

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

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 información (Baeza-Yates & Ribeiro-Neto, 2011).

Dada una consulta 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 entregársela al usuario. A continuación se presenta los modos de operar las listas invertidas para una cierta transacción de lectura.

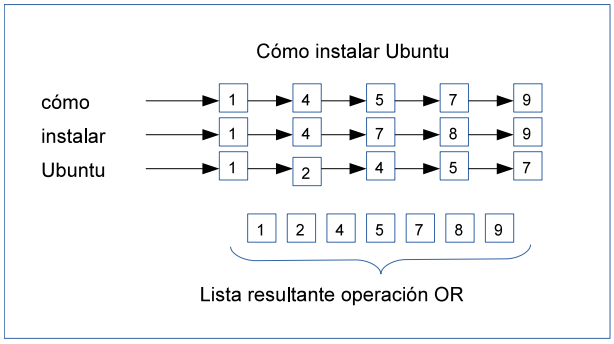


FIGURA 2.4: Operación OR.

2.5.1 *OR*

Este operador toma las listas invertidas de cada uno de los términos de la transacción de lectura 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 consulta. Finalmente, esta lista invertida se ocupará para obtener los mejores K documentos. Un simple ejemplo se muestra en la Figura 2.4.

2.5.2 AND

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

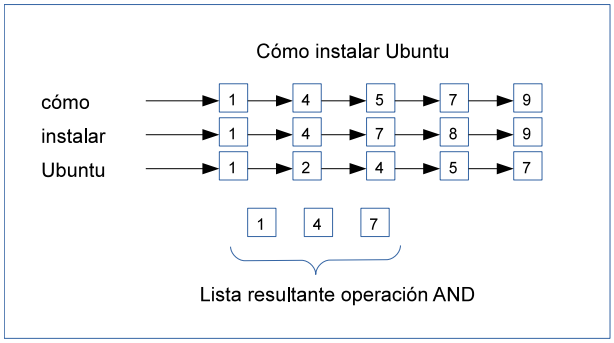


FIGURA 2.5: Operación AND.

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 costosos en términos de tiempo es el de *scoring*, el cual consiste en entregarle a cada uno de los documentos analizados un puntaje que representa la relevancia de estos para una transacción de lectura 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 invertidas, lo que se traduce en ignorar el cálculo del puntaje exacto de grandes cantidades de documentos, debido a que en motores de búsqueda a gran escala, este es 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*, 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 identifican 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 consulta 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

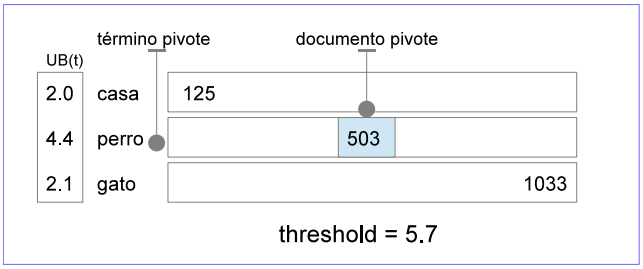


FIGURA 2.6: Ejemplo de ejecución de algoritmo Wand.

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).

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

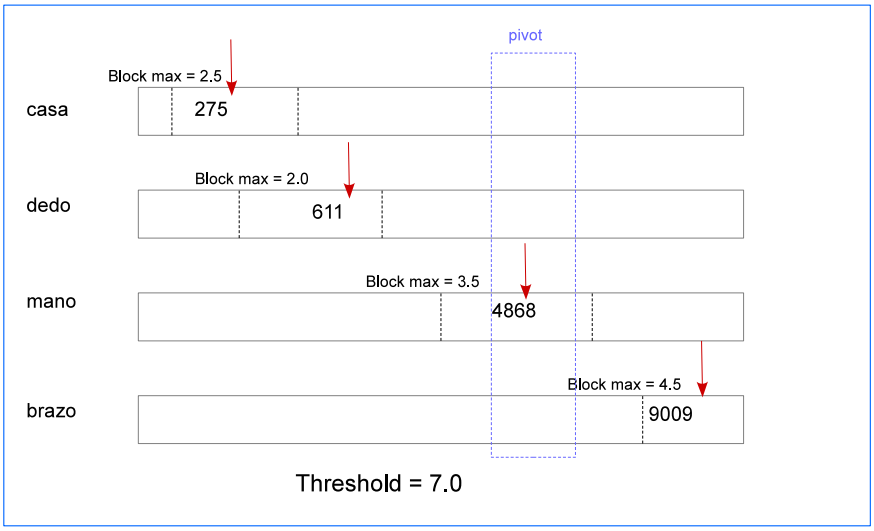


FIGURA 2.7: Ejemplo del proceso *Block-Max-Wand*.

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* de 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.

2.6 PREDICCIÓN DE TIEMPO DE RESPUESTAS DE TRANSACCIONES DE LECTURA

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, en 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 transacción de lectura 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 hacer predicciones de rendimento de transacciones de lectura. 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 consultas para la creación de un predictor.

2.7 PLANIFICACIÓN

Planificación o *scheduling* es la tarea de determinar cuando una operación comienza y finaliza; en donde cada una de estas está en posible competición con otras operaciones por los recursos disponibles. *Scheduling* también se podría definir como la asignación de recursos para ejecutar una colección de tareas, y es caracterizada por un amplio número de problemas tipo (Baker, 1974; Bazewicz & Al, 2001; Rinnooy Kan, 1976; Pinedo, 2008).

En el proceso tradicional de diseño y análisis de algoritmos, se asume que un algoritmo genera una salida con el conocimiento completo de la entrada. Sin embargo, esta suposición es muchas veces poco realista en aplicaciones prácticas. Muchos de los problemas algorítmicos en la práctica son *online*, es decir, la entrada es solo parcialmente conocida, ya que datos que son importantes en la entrada arribarán en el futuro. Un algoritmo *online* debe generar una salida sin el conocimiento de la entrada en forma completa. Ejemplos en el área de la ciencia de la computación en donde surgen problemas *online* pueden ser vistos en (Albers, 2003). Para efectos del presente trabajo de tesis, el problema se define como una secuencia de trabajos (consultas) que deben ser planificados en un conjunto de unidades de procesamiento. Los trabajos arriba uno a uno al sistema y deben ser planificados inmediatamente sin conocimiento de los futuros trabajos que llegarán.

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 y 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.

2.7.1 Planificación 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 sobrescritos. Una opción es hacer *caching* de los resultados de las búsquedas, de esta forma cuando una consulta es encontrada en caché el motor de búsqueda puede generar la respuesta rápidamente, reduciendo considerablementelos tiempos de cálculos. 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 consultas 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 transacciones de lectura 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 consultas al motor de búsqueda con el objetivo de mejorar la eficiencia.

En el contexto del presente trabajo de tesis, el objetivo de hacer *scheduling* es minimizar el

tiempo en que las consultas 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 consultas llegan al sistema (una a una), éste está obligado a tomar una decisión para planificarla sin saber cuáles transacciones de lectura 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 transacciones de lectura que llegan al sistema. Es fácil notar que resolver una consulta con varios *shard servers* mejoraría la eficiencia. Ahora bien, para asegurar un alto rendimiento (*throughput*) del sistema, cada uno de los *shard servers* poseen réplicas, de esta forma más consultas pueden ser procesadas en paralelo en copias idénticas del mismo *shard server*. Esto implica que el tiempo de espera de las transacciones de lectura que vienen llegando al sistema se reduce.

En un sistema con arquitectura como el de la Figura 2.8, una transacción de lectura puede ser procesada por varios *shard servers*, el *broker* debe escoger la réplica más apropiada para procesar la parte de la consulta 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 consultas en la cola, sin embargo, este no es un parámetro adecuado, ya que el tiempo de respuesta de las transacciones de lectura puede variar considerablemente, especialmente si se usa poda dinámica (Broder et al., 2003; Moffat & Zobel, 1996).

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

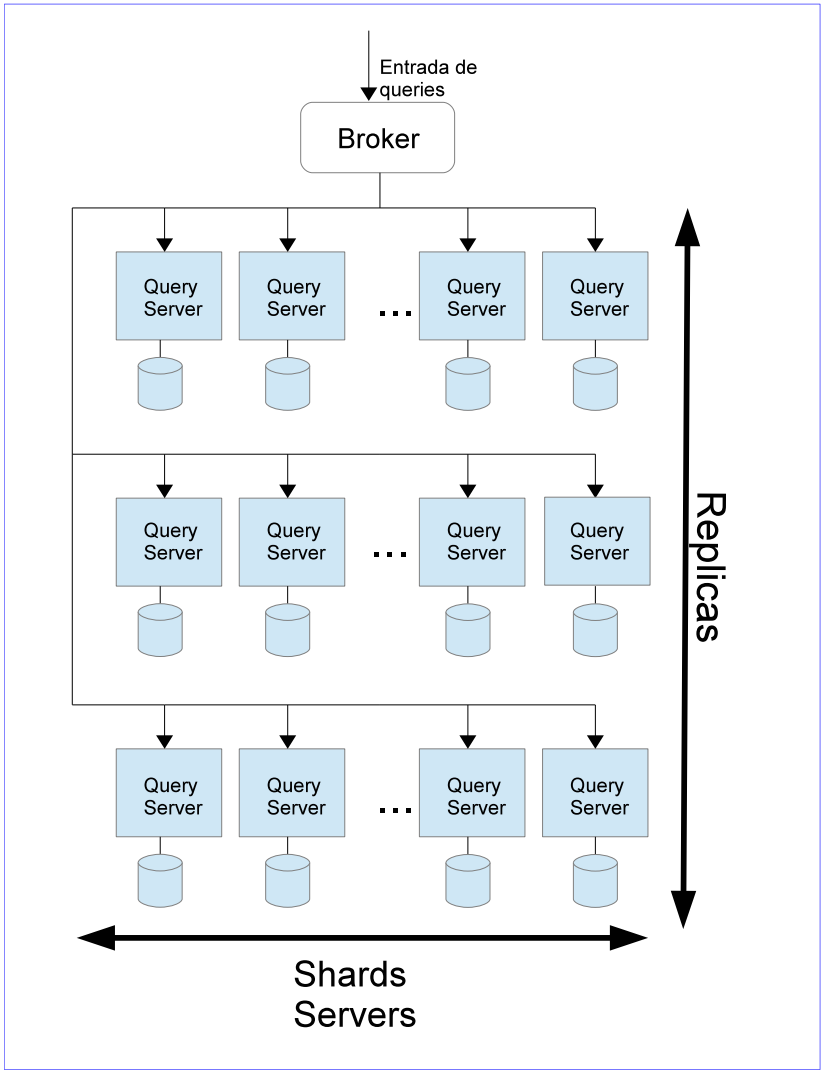


FIGURA 2.8: Arquitectura de un sistema de recuperación de información con réplicas.

2.7.2 Trabajo relacionado

El estudio (Broccolo et al., 2013) analiza métodos de *dropping* y *stopping* para el procesamiento de consultas bajo altas cargas de trabajo en un sistema distribuído donde existen múltiples servidores en el que cada uno resuelve una parte de la consulta para luego enviar las consultas al *broker* y éste hace el *merge* de los resultados de acuerdo al *score* de los documentos. Se define un tiempo T, en el que la suma de el tiempo de espera de la consulta 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 consulta es desechada y se envía al *broker* una lista vacía, (2) se detiene el procesamiento de la transacción de lectura y se envían los resultados parciales hasta el momento. Finalmente se propone un método basado en la predicción de tiempo de respuesta ($\hat{pt}(q)$) de una consulta (Macdonald et al., 2012) de modo que si se cumple $\hat{pt}(q) \leq T - wt(q)$, entonces la consulta 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 documentos 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 consultas, (Tonellotto et al., 2011) en sistemas de recuperación de información con réplicas. En este estudio, se llega a la conclusión que usando una buena predicción, se puede reducir el tiempo que la consulta tiene que esperar para ser procesada (t_w), y también se puede reducir el tiempo total requerido para procesar el conjunto (*log*) completo de transacciones de lectura (*completion time*). En (Freire et al., 2013), se propone un modelo híbrido de *scheduling* de consultas a través de réplicas, en el que cuando el sistema se encuentre bajo altas cargas de trabajo, se utilice una política de *scheduling* basada en la predicción de tiempo de respuesta de las consultas (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 MULTIHILO

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, en este trabajo se utiliza un sistema que trabaja con este método. Este algoritmo usa un *ranking* basado en una evaluación de dos niveles. En el primer nivel, 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 palabras, 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 de estos métodos se pueden encontrar en las secciones 2.5.3 y 2.5.4.

Wand y Block Max Wand son métodos lógicamente parecidos en el sentido que trabajan con *upper bounds* para poder descartar documentos, es por esto que el diseño de

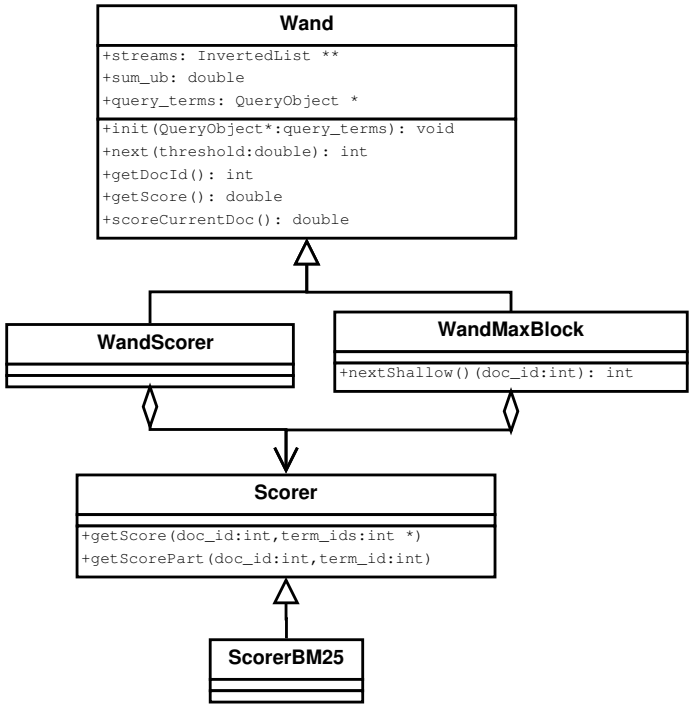


FIGURA 3.1: Diseño de clases para Wand y Block Max Wand.

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 utilizar 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 función de *ranking* que se desee, en el diagrama se observa que se utilizará BM25.

Existen dos formas de implementar Wand. Una de ellas es usando *heaps* locales (LH), es decir, un *heap* por hilo de ejecución y el otro es usando *heaps* compartidos (SH). En el estudio (Rojas et al., 2013) se muestran 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 cálculos de puntajes completos y (2) se ejecutan

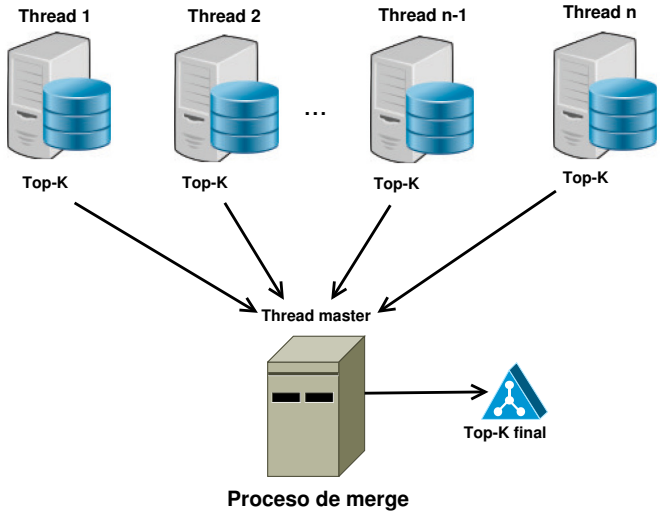


FIGURA 3.2: Esquema de ejecución de algoritmo WAND con *heaps* locales.

pocas operaciones de actualización del *heap* (reduciendo el número de *locks* que se hace a la estructura de datos). A continuación 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 hebra procesa una parte del índice invertido mientras mantiene un *heap* local con los mejores K documentos. Al finalizar este proceso, los resultados se unen en un solo conjunto final global. Los resultados en (Rojas et al., 2013) muestran que el esquema LH es más eficiente 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 consulta dada. Si es que la consulta 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 hilos de ejecución (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 las diferentes hebras, 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 del índice se lleva a cabo mediante un mapa, en donde cada término tiene asociado su lista invertida correspondiente y para la extracción de estas listas se usa el método getList.

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

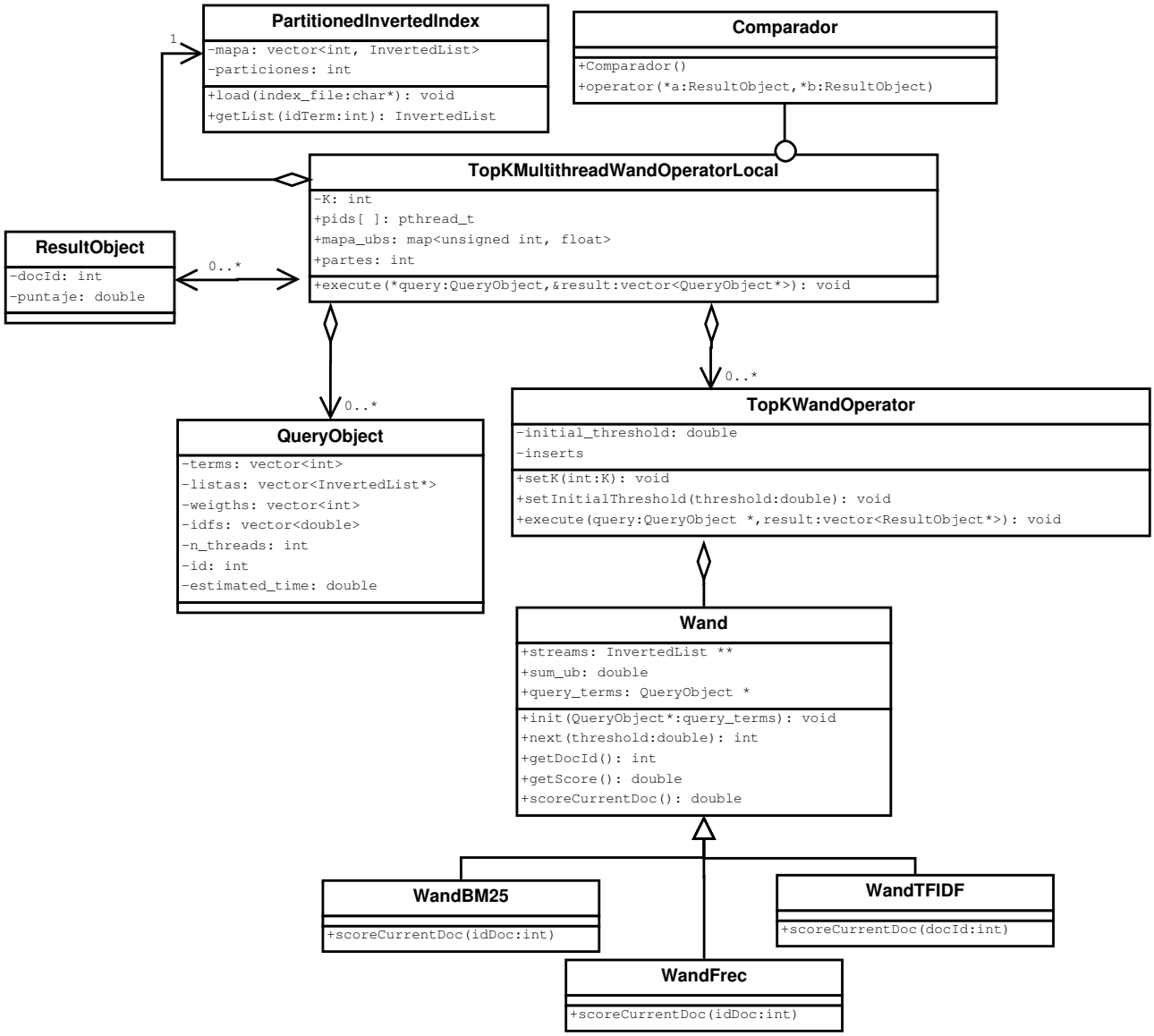


FIGURA 3.3: Diagrama de clases para el esquema LH.

de objetos Wand que se pueden utilizar, entre ellos están WandBM25, WandFrec y WandTFIDF, donde la única diferencia entre ellos es el método con que se calcula el puntaje de cada documento. Por ejemplo, WandBM25 utiliza BM25 y WandTFIDF utiliza tf-idf.

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

QueryObject. Clase que representa una transacción de lectura. Está formada por términos y sus respectivas listas invertidas, la cantidad de hebras con las cuales se resolverá dicha transacción y el tiempo estimado de procesamiento (este tiempo se predice al momento de resolver la consulta).

3.2 WAND CON *HEAP* COMPARTIDO

En el esquema SH cada hebra procesa una parte del índice. Sin embargo, ahora un solo *heap* es creado y accedido por todos los hilos de ejecución. En este caso no se requiere 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*. El acceso al *heap* debe ser controlado por un *lock* o algún método similar que garantice el acceso exclusivo de los hilos al *heap*. Este esquema es más eficiente que el LH en consultas 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 diagrama de clases mostrado en la Figura 3.5.

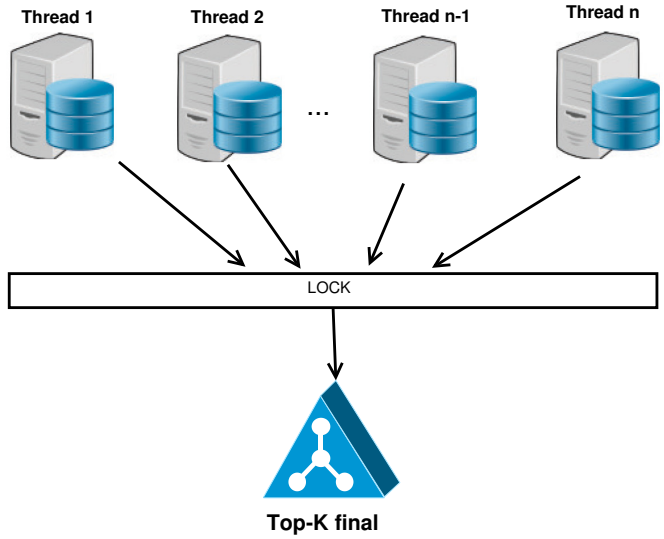


FIGURA 3.4: Esquema de ejecución de algoritmo WAND con *heap* compartido.

TopKMultiThreadWandOperatorLocks. Clase encargada de inicializar las variables compartidas y de lanzar los hilos de ejecución 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 por todas las hebras).

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.

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*, utiliza 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; también 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 que generalmente implica una descompresión del bloque 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

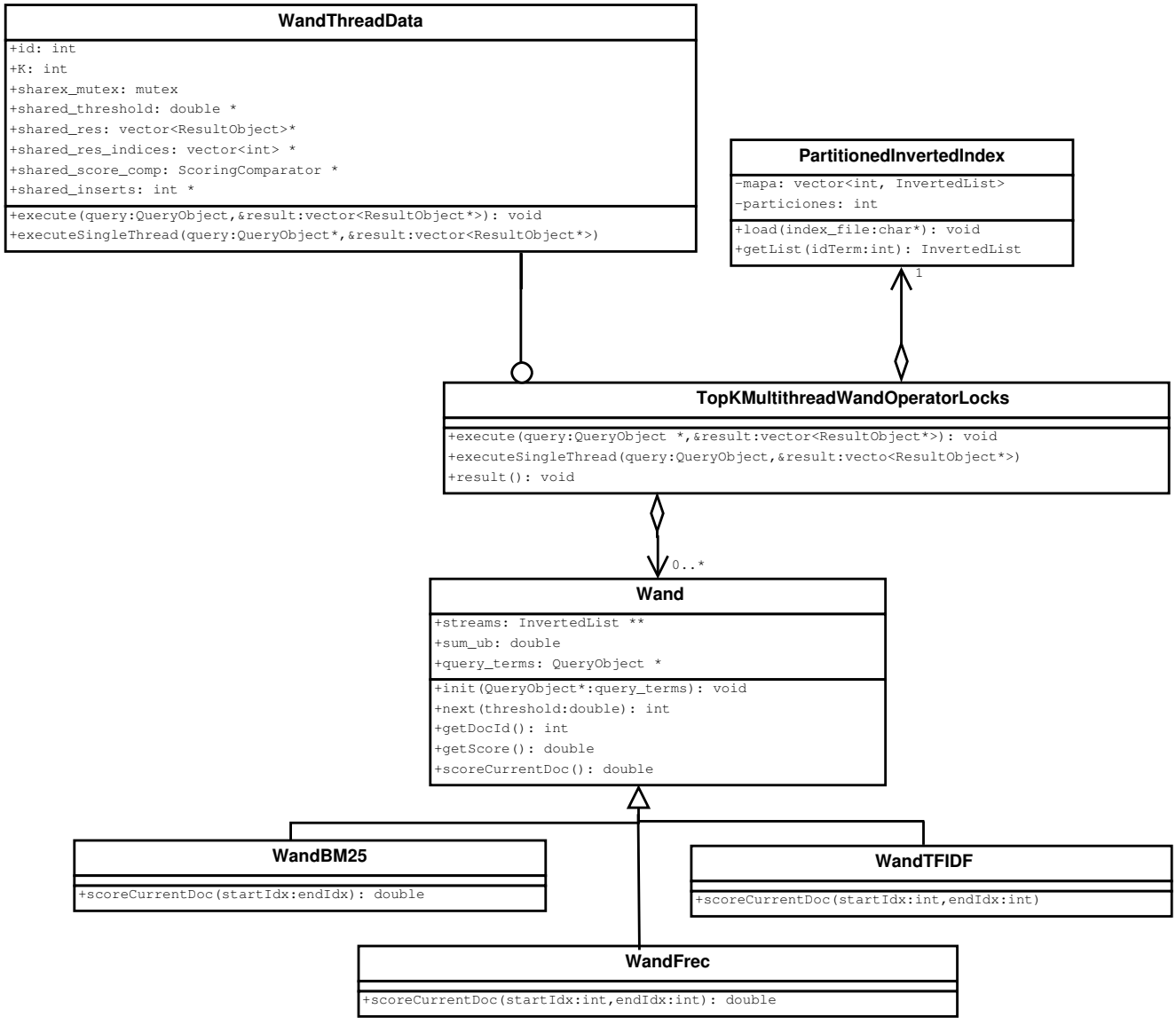


FIGURA 3.5: Diagrama de clases para el esquema SH.

que todas las listas invertidas poseen un puntero al documento actual que se desea evaluar (*currentDoc*). Lo primero que se hace es ordenar de manera 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 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).

Algoritmo 3.1: $BMW(\theta, L, docID) : BlockMaxWand$

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
```

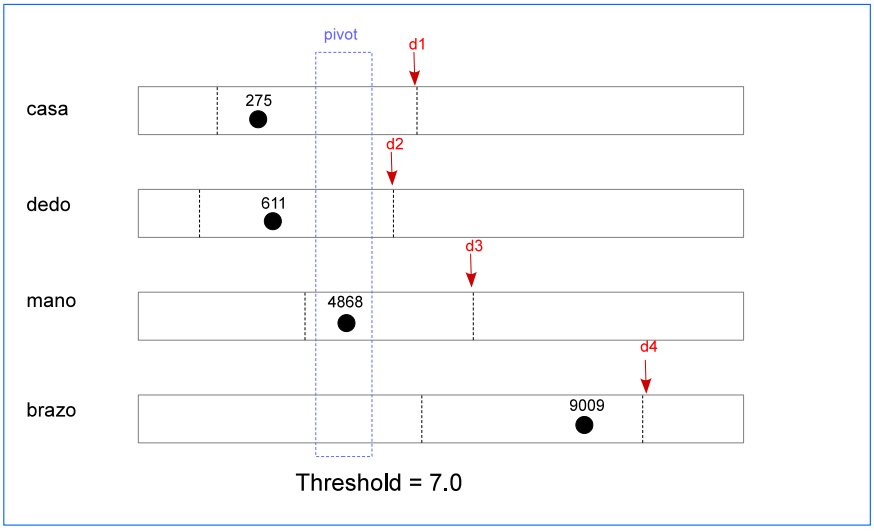


FIGURA 3.6: Ejemplo de cómo opera la functión getNewCandidate().

CAPÍTULO 4. MÉTODOS DE PREDICCIÓN DE RENDIMIENTO

Lograr bajos tiempos de respuesta para transacciones de lectura es uno de los principales objetivos en el diseño de un motor de búsqueda, ya que de esta forma se le puede entregar una respuesta oportuna al usuario. Aquellas transacciones de lectura que requieren una gran cantidad de tiempo en 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 (también llamado *tail latency*) (Jeon et al., 2014). Paralelizar el procesamiento de cada consulta es una solución promitente para reducir el tiempo de ejecución de estas (Jeon et al., 2013; Tatikonda et al., 2011), lo cual es posible gracias a los modernos procesadores que existen hoy en día que poseen múltiples núcleos, en donde se puede resolver una consulta paralelizando múltiples hilos de ejecución.

Conocer de antemano la eficiencia de una transacción de lectura es una ventaja muy importante, puesto que aquellas consultas que toman una mayor cantidad de tiempo en ser resueltas se les asigna un mayor número de hilos de ejecución para resolverla, de esta manera se reduce el tiempo de procesamiento de las consultas y se cumple con la cota superior de tiempo establecida. Permite implementar técnicas efectivas de procesamiento y de planificación de transacciones de lectura, por ejemplo, en el contexto de procesamiento paralelo de consultas por lotes (*batches*), se pueden crear grupos de consultas que posean similares tiempos de respuesta, así se tiende a disminuir tanto el desbalance de carga entre los procesadores como el tiempo en procesar el lote completo.

Con el objetivo de construir estrategias de procesamiento y planificación de consultas eficientes, en el presente trabajo se lleva a cabo la implementación de dos métodos de predicción de rendimiento para transacciones de lectura. La construcción de estos métodos de predicción

se lleva a cabo con el objetivo de disminuir el tiempo en resolver conjuntos de consultas y asegurar una cota superior de tiempo para cada una de ellas. Adicionalmente, se busca estudiar cómo estos métodos afectan el rendimiento de un motor de búsqueda bajo el contexto de procesamiento de consultas por lotes utilizando el método Wand (Broder et al., 2003) y Block Max Wand (Ding & Suel, 2011).

4.1 MÉTODO DE PREDICCIÓN MULTILINEAL

Este método predice el tiempo de respuesta de una transacción de lectura y está basado en una regresión lineal múltiple con 42 variables independientes (Macdonald et al., 2012). Como la respuesta a una consulta debe ser rápida, los estadísticos obtenidos desde las listas invertidas 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.

Si bien es cierto que la regresión lineal posee 42 variables independientes, desde las listas invertidas se extraerán solo 14 estadísticos, ya que las 42 variables independientes se forman aplicando funciones de agregación sobre estos estadísticos. 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; Hauff, 2010; 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 información.

Media aritmética.

Media geométrica.

Media harmónica.

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 desde la lista invertida del término t, la varianza del puntaje de los documentos.

Número de documentos. Largo de la lista invertida.

Número de máximos. Número de veces en que aparece un nuevo puntaje máximo, es decir, el número de veces en que el *upper bound* es actualizado.

Número de documentos mayor a la media. Número de documentos con puntaje superior al puntaje promedio.

Número de documentos con puntaje máximo. Número de documentos que poseen el puntaje máximo en la lista invertida del término t.

Número de documentos dentro del 5 % más alto. Número de documentos cuyos puntajes están dentro del 5 % superior.

Número de documentos dentro del 5 % del umbral (*threshold*). Número de documentos cuyos puntaje están dentro del 5 % superior o 5 % inferior al *threshold*. Recordar que el *threshold* es el puntaje 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 consulta 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 al *threshold*.

Frecuencia inversa de documento del término. Se calcula el *idf* del término t (Baeza-Yates & Ribeiro-Neto, 2011).

Tiempo en ser procesado el término. Tiempo que toma en ser procesado el término como si fuese una consulta 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 usarán por cada consulta: 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, luego se aplican las funciones de agregación a los estadísticos de los términos. Por ejemplo, suponga que llegan dos consultas al sistema q_1 y q_2, ambas tendrán asociadas un vector de 14 estadísticos E_{q_1} y E_{q_2} respectivamente, las funciones de agregación para el estadístico de la media aritmética será calculado como sigue: $e_1 = max\{E_{q_1}(0), E_{q_2}(0)\}$, $e_2 = var\{E_{q_1}(0), E_{q_2}(0)\}$, $e_3 = sum\{E_{q_1}(0), E_{q_2}(0)\}$. De esta forma, con solo el primer estadístico (la media aritmética) se obtienen tres variables independientes (e_1, e_2, e_3). Si esto se extrapola a cada estadístico, se obtienen los 42 requeridos por el método. La Tabla 4.1 muestra un resumen de lo escrito anteriormente en donde se muestra cada uno de los estadísticos y los agregadores a utilizar.

4.2 MÉTODO DE PREDICCIÓN NEURONAL

Se implementa un método de predicción basado en una red neuronal *backpropagation* (Rumelhart et al., 1988). La característica de este tipo de redes es que utilizando al menos una capa oculta, se puede aproximar cualquier tipo de función o relación continua entre un grupo de

TABLA 4.1: Resumen de los estadísticos para la predicción multilineal

Estadísticos de términos $s(t)$
1. Media aritmética
2. Media geométrica
3. Media harmónica
4. Puntaje máximo
5. Varianza del puntaje
6. Número de documentos
7. Número de máximos
8. Número docs > media
9. Número docs = máximo puntaje
10. Número docs dentro del 5 % más alto
11. Número docs dentro del 5 % del *threshold*
12. Número de inserciones al conjunto *top-K*
13. IDF
14. Tiempo en resolver t como consulta
Agregadores A()
a. Máximo
b. Varianza
c. Suma

variables de entrada y salida. Este tipo de redes neuronales utilizan un método de entrenamiento en el cual se propaga el error hacia atrás para ajustar los pesos de las diferentes neuronas del modelo, de esta forma la red neuronal va generando una asociación entre la entrada y salida (Fausett, 1994).

Se implementa el modelo neuronal usando las mismas 42 variables independientes del método anterior, debido a que ya se ha demostrado que existe una relación lineal entre estas variables y la variable tiempo (Macdonald et al., 2012; Hauff, 2010; He & Ounis, 2004). El modelo consiste de dos neuronas en una capa oculta, la idea es que el tiempo que toma la predicción no genere un impacto negativo en el tiempo de procesamiento de la consulta, es por esto que se decide utilizar solo dos neuronas, sin embargo, en el Capítulo 6 también se hace un análisis del modelo con 10 y 20 neuronas en la capa oculta. El objetivo es minimizar el error de la estimación de tiempo y que la predicción no genere un impacto negativo en términos de tiempo.

CAPÍTULO 5. ESTRATEGIAS DE PLANIFICACIÓN DE CONSULTAS

Los motores de búsqueda verticales son diseñados con el propósito de lidiar con cargas dinámicas de trabajo. 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 consultas 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 algoritmos implementados soporten la concurrencia entre las transacciones de lecturas y escrituras; ya que 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 muestran las diferentes estrategias de planificación de transacciones de lectura abordadas en el presente trabajo utilizando 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 sirve de *baseline*, 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; El tercer y último enfoque corresponde a unidades de trabajo, en la que a cada transacción de lectura se le asigna un número determinado de unidades de procesamiento y los hilos de ejecución compiten por ellas obteniéndolas desde una cola.

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 se llamará t_{limite}. En el contexto del presente trabajo, para que el planificador tome una decisión con respecto a una consulta, 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 consulta para 1, 2, 4, 8 y 16 hebras, se asigna el número 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 RW que aborda este problema (Ye & Zhang, 2007) y se

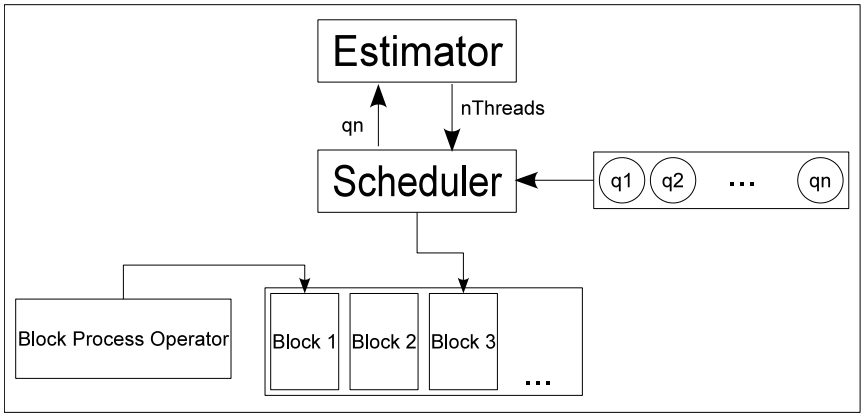


FIGURA 5.1: Enfoque de planificación para estrategias por bloques.

adapta a este 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 RW, 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 de este 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 ésta sea resuelta en un tiempo inferior al tiempo límite t_{limite}. Una vez que a la transacción de lectura se le predice el tiempo de ejecución y el número de hebras a utilizar, este planifica la consulta en algún bloque correspondiente dependiendo de la política que se este utilizando. A continuación se muestran las diferentes estrategias propuestas.

5.1.1 Estrategia Rooms y Walls

La estrategia de *Rooms* y *Walls* (RW) posee como requisito que a 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 le asignará la cantidad mínima de hebras tal que el tiempo en resolver la consulta sea menor al tiempo límite t_{limite}.

Por lo explicado anteriormente, el algoritmo RW asume que cada consulta 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 datos tienen un número máximo de máquinas disponibles para procesar las transacciones de lectura. Una consulta 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 consulta 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 transacción de lectura 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 consulta 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 planificar esta consulta; para realizar lo anterior, el bloque debe satisfacer dos condiciones: (1)

Algoritmo 5.1: *schedulerRW* :: *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(Q) then
2:    block = newWall();
3:    block → addQuery(Q);
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() >= Q → getThreads())
        then
10:       room_block → addQuery(Q)
11:       asignada = true
12:       break;
13:     end if
14:   end for
15:   if !(asignada) then
16:     block = newRoom();
17:     block → addQuery(Q);
18:     L → addBlockLocked(block);
19:   end if
20: end if
```

no debe estar completo, es decir, debe tener hebras disponibles, y (2) no debe haber sido procesado aún. Por último, si eventualmente no se encuentra algún bloque disponible para planificar la consulta, 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.

En la Figura 5.1 se presenta un ejemplo de ejecución la estrategia RW. Han llegado al

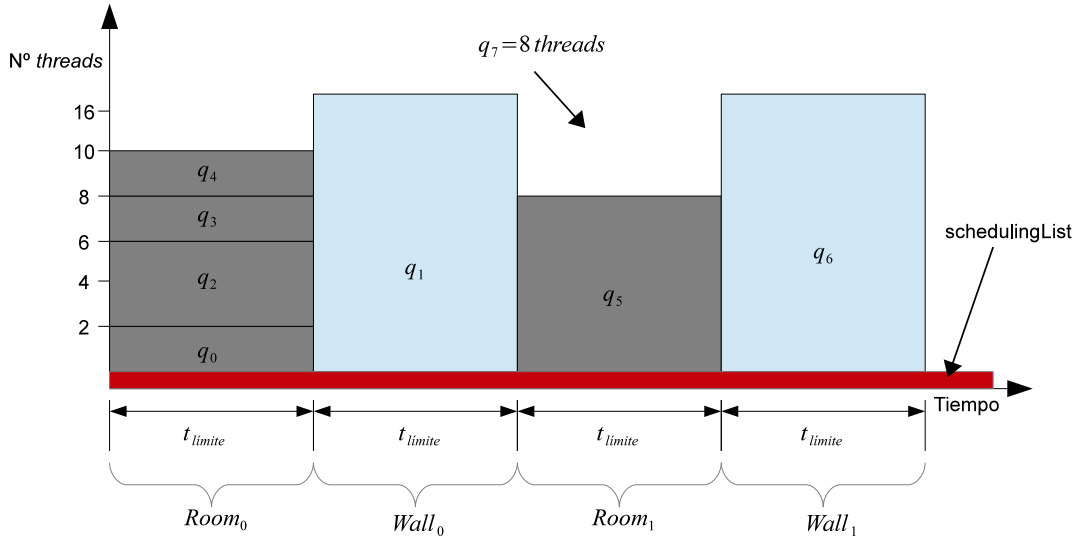


FIGURA 5.2: Ejemplo de procesamiento de la estrategia RW.

sistema consultas: q_0 (2 hebras), q_1 (16 hebras), q_2 (4 hebras), q_3 (2 hebras), q_4 (2 hebras), q_5 (8 hebras) y q_6 (16 hebras). Se puede ver cómo se van formando las estructuras de datos llamadas *Rooms* y *Walls*. Suponer que eventualmente arriba al sistema una nueva consulta (q_7) que será resuelta con 8 hebras, entonces el algoritmo verifica en primera instancia la $Room_0$, sin embargo, en esta estructura no hay suficientes hilos de ejecución disponibles para procesar la consulta (posee solo 6 disponibles). Finalmente la planifica en la $Room_1$.

5.1.2 Estrategia Times

Se intuye que una de las desventajas de la estrategia RW es que al planificar una consulta en algún bloque, solo verifica si es que este posee el número de hebras desocupadas suficientes tal que sea mayor o igual al número de hilos requeridos 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 hebras 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 hilos de ejecución 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 hilos. 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 5.2 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 RW, la diferencia es que en esta estrategia una consulta puede ser planificada en un bloque siempre y cuando este tenga hebras disponibles suficientes para procesarla y que el tiempo de la consulta no doble al tiempo mínimo dentro del bloque perteneciente a alguna consulta 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

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() ≥ Q → getThreads() then
7:       tiempo_min = block → getMinimumTime()
8:       if block → isSchedulable(Q) then
9:          L → addQuery(Q);
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(Q);
22:   L → addBlockLocked(block);
23: end if
```

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 consultas de acuerdo al número de hilos de ejecución que utilizan.

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 hebras; 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

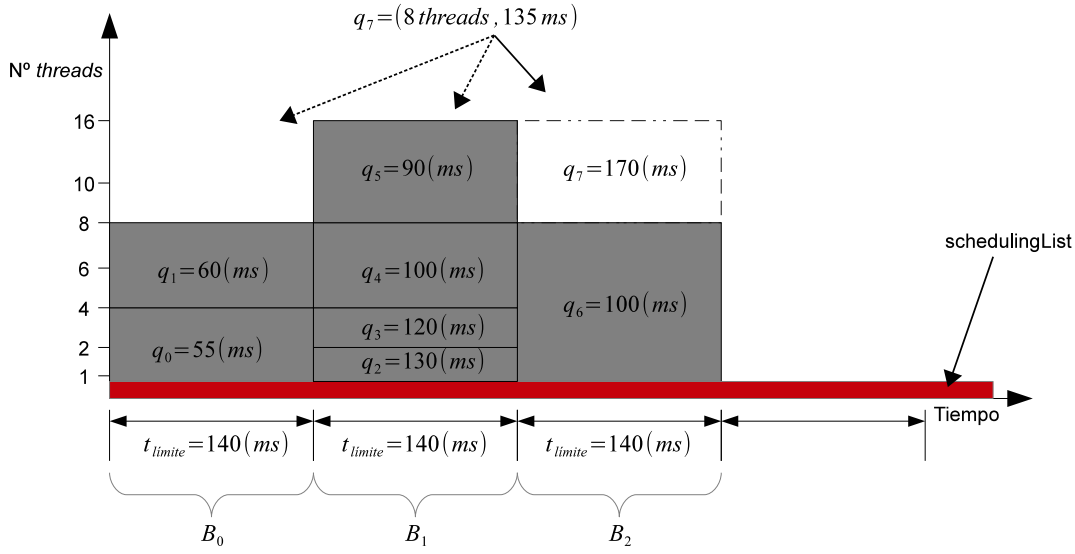


FIGURA 5.3: Ejemplo de procesamiento de la estrategia Times.

en el bloque (55 ms.). Posteriormente, el bloque B_1 no posee hebras disponibles. Finalmente la transacción de lectura q_7 es planificada en el bloque B_2, ya que reúne todas las condiciones necesarias explicadas anteriormente en el algoritmo 5.2.

5.1.3 Estrategia Times Ranges

Esta estrategia también intenta disminuir la posible pérdida de eficiencia de la estrategia RW. La idea de *TimesRanges* es clasificar y agrupar las consultas 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 hilos de ejecución 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

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

(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 se reduce 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 estrategia *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 hebras disponibles suficiente para procesarla, y se planifica la consulta. Si no se encuentra un bloque que satisfaga las condiciones de la consulta entrante, entonces se crea uno nuevo para planificarla.

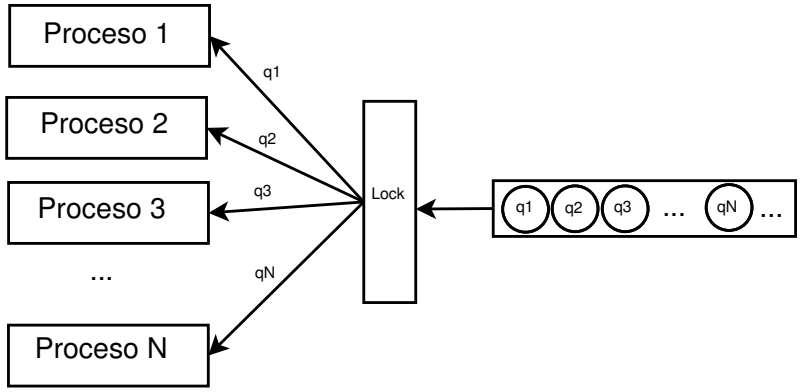


FIGURA 5.4: Ejemplo de procesamiento estrategia 1TQ.

5.2 ESTRATEGIA UN THREAD POR QUERY

Un simple camino para construir un sistema que responda a múltiples consultas simultáneamente usando múltiples hilos de ejecución, es usando estos hilos de manera independiente. Para hacer esto se debe mantener un conjunto de hilos de ejecución consumidores que trabajarán en paralelo y se encargarán de resolver las transacciones de lectura 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 5.4 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 consultas 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 consulta, 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 hilos de ejecución.

Este esquema tiene la ventaja que es simple y fácil de implementar y controlar. Sin embargo, existen sistemas de recuperación de información como los motores de búsqueda verticales que cuando están ejecutando *batches* de consultas deben parar su ejecución porque

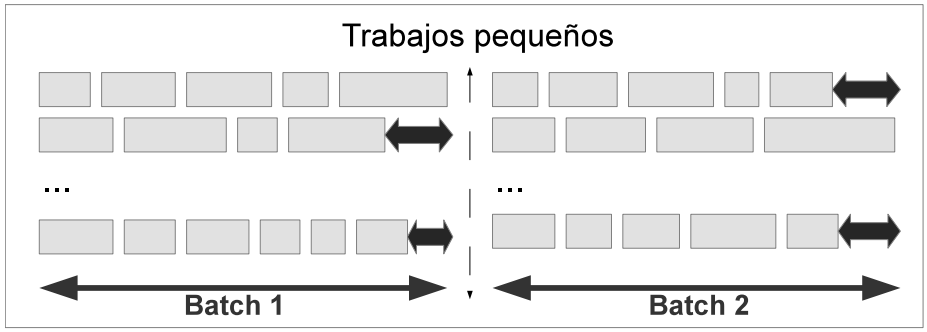


FIGURA 5.5: Ejecución en paralelo de *small jobs*.

transacciones de escritura han llegado al sistema, y estos 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 consultas para procesar, por lo que ellos tienen que esperar que los hilos 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 transacciones de lectura que requieran poca cantidad de tiempo para ser procesadas (trabajos pequeños o *small jobs*). En la Figura 5.5 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 5.6).

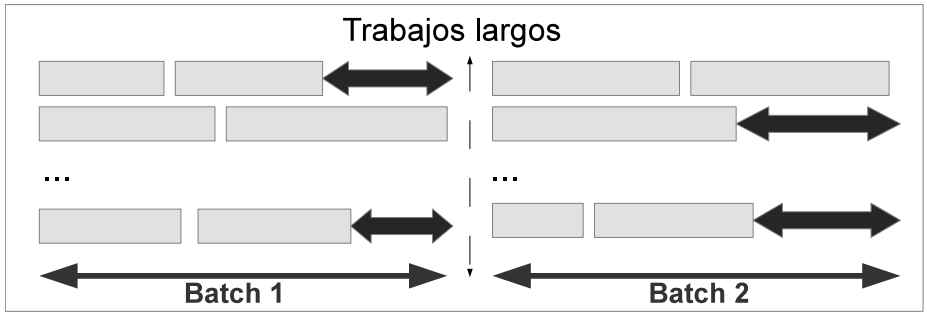


FIGURA 5.6: Ejecución en paralelo de *large jobs*.

5.3 ESTRATEGIA UNIDADES DE TRABAJO

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 se resuelven transacciones de lectura por lotes; sin embargo, no se garantiza una cota superior de tiempo para cada consulta. En esta sección se propone un esquema en donde la consulta es dividida en unidades de procesamiento o de trabajo, en el que a cada unidad se le asigna la parte del índice invertido con el cual se debe trabajar. Además existe un nivel de datos compartido entre las unidades de procesamiento, como por ejemplo, el *heap* en donde se guardará el conjunto *top-K*.

En este nuevo esquema de planificación, las consultas pasan a través de una fase en la cual se predicen sus tiempos de ejecución y se les asigna el mínimo número de hebras tal que se cumpla con la cota superior de tiempo establecida. Posteriormente se estima para cada consulta el número de unidades de procesamiento que se utilizará en su resolución, este número será igual al número de hilos de ejecución estimado (Ver Figura 5.7). Un conjunto de hilos consumidores extraen las unidades desde la cola y las procesan de forma independiente. Cuando una hebra

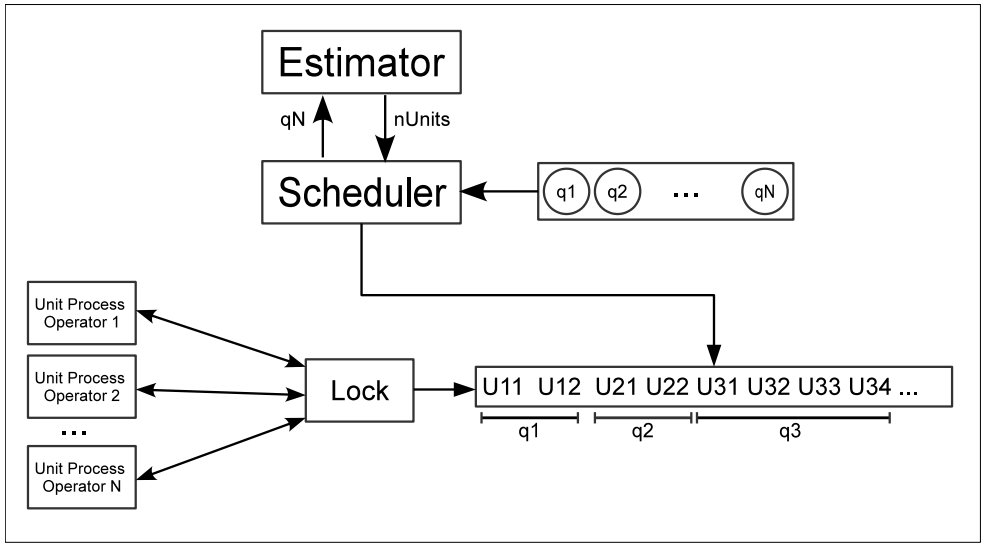


FIGURA 5.7: Procesamiento de consultas utilizando unidades de trabajo.

finalice el procesamiento de la unidad de trabajo actual, automáticamente extraerá la siguiente unidad de trabajo desde la cola. De esta manera se forma una competencia entre los hilos de ejecución por las unidades de trabajo de la cola, por lo que se debe controlar el acceso concurrente de los hilos al nivel compartido de datos entre las unidades de procesamiento, de tal manera que solo una hebra tenga acceso exclusivo a este objeto para poder actualizar los datos a medida que cada una de las unidades de la transacción de lectura es resuelta.

Se muestra un ejemplo en la Figura 5.8 en el que se resuelven cuatro consultas. La consulta 0 fue dividida en cuatro unidades de trabajo; la consulta 1 y 2 en dos unidades de trabajo; y finalmente la consulta 3, en ocho unidades de trabajo. Los hilos de ejecución competirán por la extracción de unidades de trabajo desde la lista.

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 procesadas al mismo tiempo. La única excepción es que la unidad de procesamiento que inicializa la consulta es siempre ejecutada antes del resto de las otras unidades de la misma consulta, y también la entrega de resultados se hace una vez que todas las unidades de procesamiento de la consulta han finalizado. Este enfoque híbrido permite

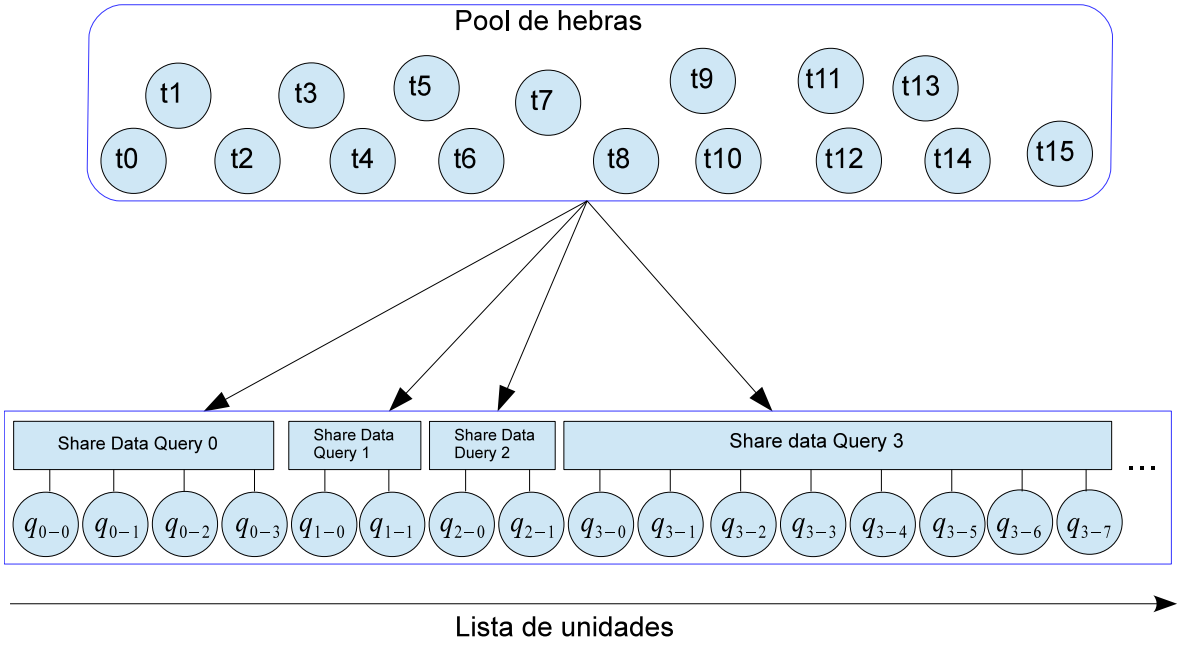


FIGURA 5.8: Esquema de ejecución estrategia unidades de procesamiento.

reducir el tiempo perdido al final de cada *batch* de consultas sin generar una importante pérdida de trabajo mientras las consultas del *batch* están siendo procesadas.

En la Figura 5.9 se presenta el diagrama de clases del planificador de unidades de procesamiento:

TopKMultiThreadWandOperatorLocal. Clase encargada de devolver los mejores K documentos para una transacción de lectura dada. Si es que la consulta 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 hilos (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 las diferentes hebras, 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.

QueryObject. Clase que representa las transacciones de lectura a las que se le asignará unidades de procesamiento.

QueryUnit. Clase que representa las unidades de procesamiento de transacciones de lectura. Cada uno de estos objetos tiene asociado un nivel de datos compartido (QueryUnitShareData) con otras unidades de procesamiento que resolverán la misma consulta.

QueryUnitShareData. Clase que permite la creación de datos compartidos entre las unidades de procesamiento. Todas las variables compartidas por las unidades de trabajo estarán albergadas en este objeto.

QueryEstimator. Clase que se encarga de estimar el número de unidades de procesamientos requeridas para una cierta consulta.

QueryUnitScheduler. Clase principal del planificador, esta se encarga de tomar las transacciones de lectura y asignarle a cada una de ellas el número de unidades de procesamiento adecuado. Finalmente, las unidades de trabajo de planifican en una lista.

Finalmente se diseña un diagrama de clases para el ejecutador de unidades de procesamiento (Ver Figura 5.10). Aquí se pueden observar los objetos involucrados en la resolución de cada consulta; cada objeto QueryProcessingOperator es el encargado de procesar una unidad, y habrán tantos objetos de este tipo como hilos de ejecución disponibles en el sistema. Cada vez que una unidad es procesada, si es que restan unidades por resolver, el hilo de ejecución toma inmediatamente la siguiente unidad de trabajo, de lo contrario debe retornar el conjunto final de documentos que se encuentra en el *heap* compartido.

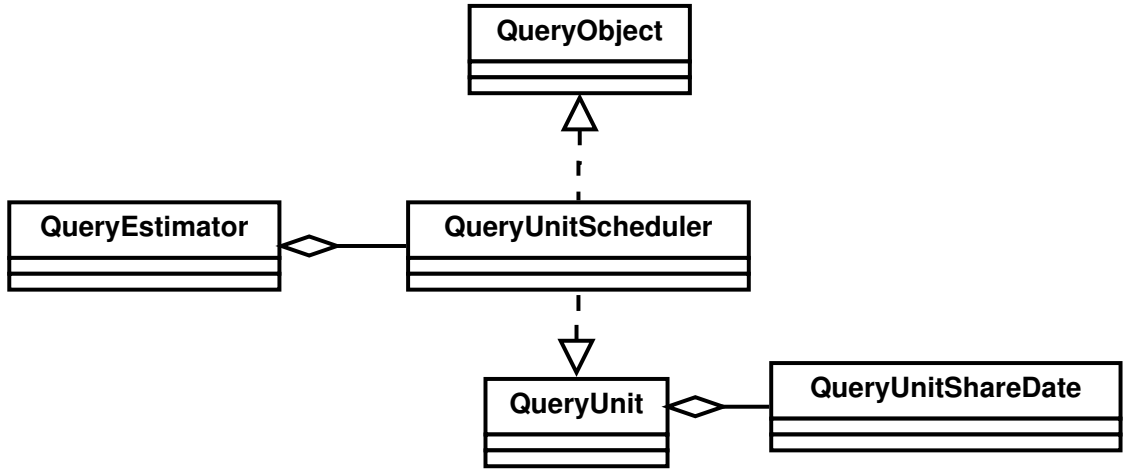


FIGURA 5.9: Diagrama de clases del planificador de unidades de procesamiento.

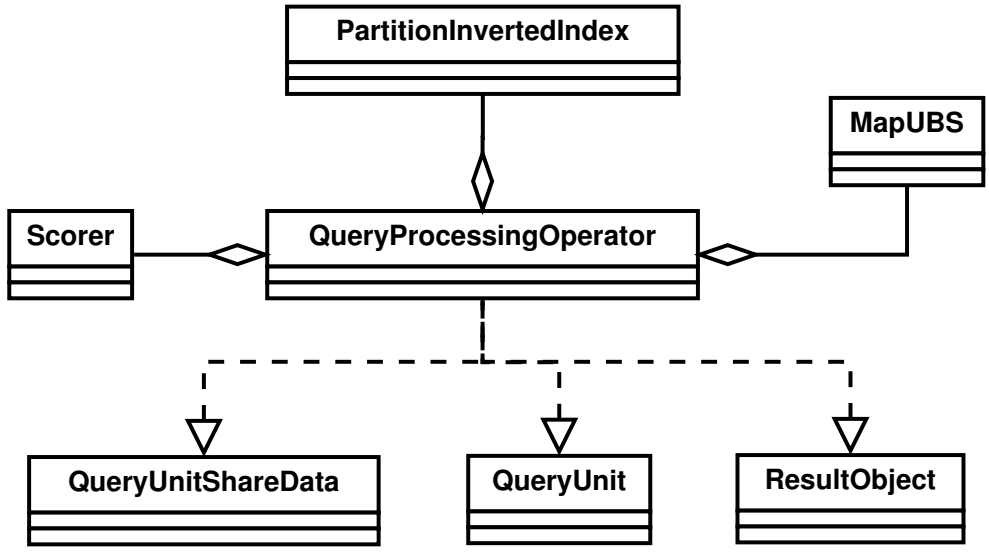


FIGURA 5.10: Diagrama de clases del ejecutador de unidades de procesamiento.

Se han descrito las estrategias de planificación y de procesamiento de transacciones de lectura que se evaluarán mediante experimentación. Se intuye que dentro de las estrategias de planificación por bloques, la estrategia *Times Ranges* debería tener mejor rendimiento que las estrategias RW y *Times*; por otro lado, se propone un enfoque que divide una transacción de lectura en unidades de trabajo y en el que las hebras compiten por procesar estas unidades. Finalmente se propone un enfoque básico que sirve como punto de comparación en el que una consulta siempre será procesada por solo un hilo de ejecución. Las comparaciones de las diferentes estrategias descritas en el presente capítulo se muestra en el Capítulo 6.

CAPÍTULO 6. EVALUACIÓN EXPERIMENTAL

En este capítulo se presentan 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 muestran los resultados obtenidos para los diferentes métodos de predicción de tiempos de respuestas para consultas y el comportamiento que tienen para diferentes conjuntos de datos. Finalmente se presentan los tiempos de ejecución de las estrategias de planificación 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 *hyperthreading* 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 recuperación de información. 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 *gigabytes* de espacio en disco. El segundo conjunto de datos utilizado es *ClueWeb09*, el cual fue creado para apoyar la investigación en recuperación de información y las tecnologías relacionadas con el lenguaje humano, consiste en alrededor de un billón de páginas en 10 lenguajes diferentes y 50 millones en inglés. *ClueWeb09* pesa alrededor de 5 *terabytes* de espacio en disco en forma comprimida y 25 *terabytes* en forma descomprimida.

6.2 WAND MULTIHILO

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 *heaps* locales

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 en el que se almacenarán los resultados obtenidos. Adicionalmente, este método es el encargado de lanzar las hebras con que se resolverá cada consulta y a cada una de ellas le asigna 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.

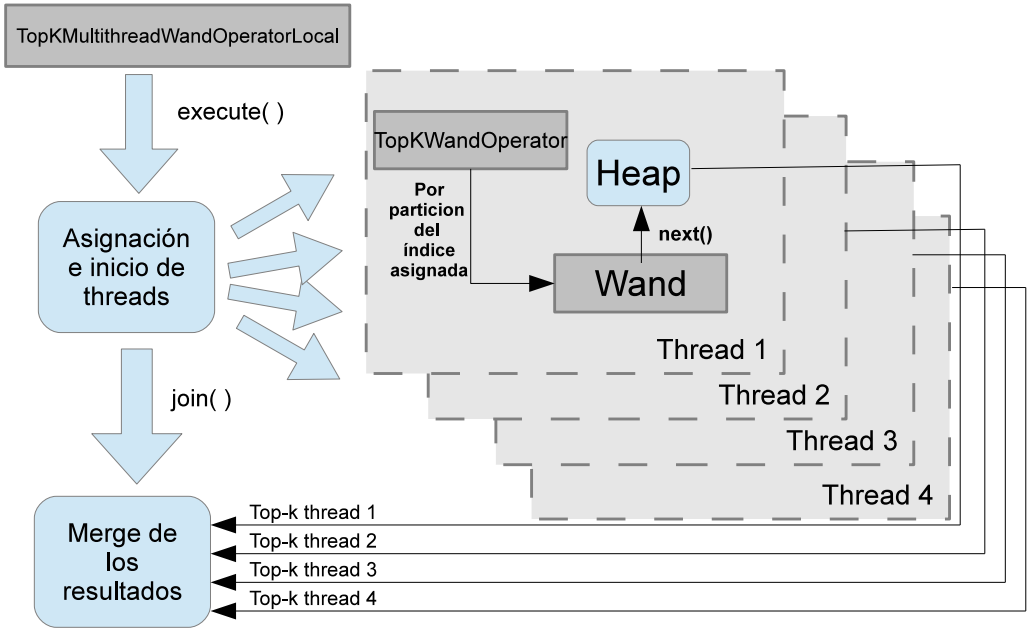


FIGURA 6.1: Esquema de ejecución enfoque LH.

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 dividida y está supeditada al número de hebras con que esta será resuelta.

En la Figura 6.1 se ejemplifica la resolución de una consulta con cuatro hilos de ejecución. Una vez que el sistema asigna el número de hebras que se utilizarán 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*.

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
        TopKMultithreadWandOperatorLocal(PartitionedInvertedIndex *_indice,
            map<unsigned int, unsigned int> *_mapa_docs,
            map<unsigned int, float> **_mapas_ubs,
            unsigned int _k = 10,
            unsigned int _max_terms = 128);

        // Destructor
        ~TopKMultithreadWandOperatorLocal();

        // Metodo que se encarga de resolver la query. Los resultados
        // quedaran en la variable result
        virtual void execute(QueryObject *query, vector<ResultObject*> &result);
};
```

6.2.2 Esquema *heap* compartido

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*, lo cual se puede ver en el Código 6.2; en esta implementación se puede observar la declaración de una clase anidada, la cual 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.

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 una 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 de hilos de ejecución, el enfoque de *heaps* compartidos toma ventaja por sobre el enfoque de

CÓDIGO 6.2: Implementación de la clase TopKMultithreadWandOperatorLocks.h

```
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;

                // Vector de resultados
                vector<ResultObject> *shared_res;
                vector<unsigned int> *shared_res_indices;

                ScoringComparator *shared_score_comp;
        };

        TopKMultithreadWandOperatorLocks(PartitionedInvertedIndex *_indice, map<unsigned int,
            unsigned int> *_mapa_docs, map<unsigned int, float> **_mapas_ubs, unsigned int _k
            = 10);
        virtual ~TopKMultithreadWandOperatorLocks();

        // M todo que hara la resoluci n de la query
        virtual void execute(QueryObject *query, vector<ResultObject*> &result);
        virtual void executeSingleThread(QueryObject *query, vector<ResultObject*> &result);
        virtual void reset();
};
```

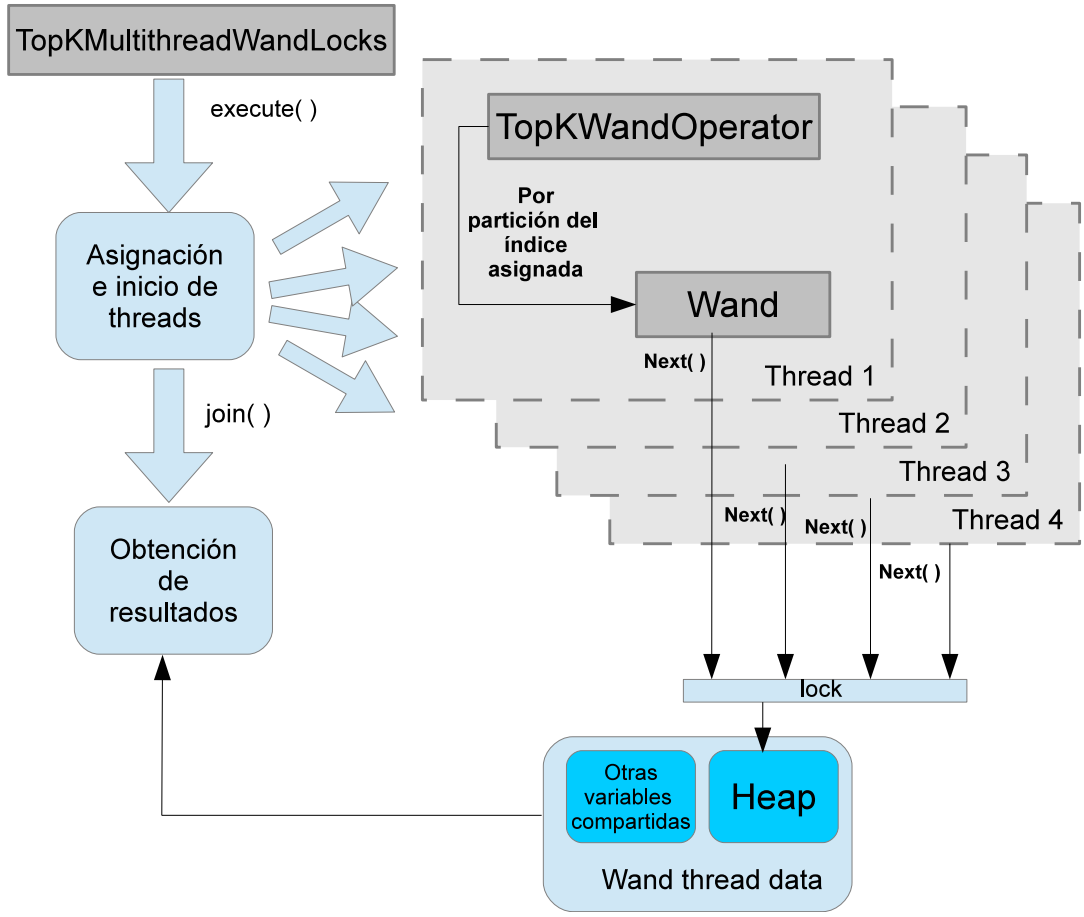


FIGURA 6.2: Esquema de ejecución enfoque SH.

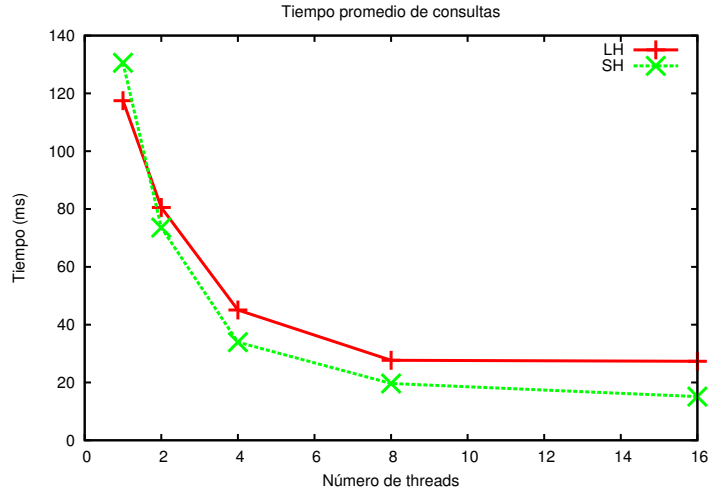


FIGURA 6.3: Tiempos promedios de las consultas.

heaps locales, sin embargo, cuando se utiliza un hilo de ejecución se puede observar que LH (117.486 ms) requiere un tiempo menor que SH (130.591 ms), esto se debe a que en LH no se usan variables compartidas que retrasen a los hilos de ejecución esperando a que otros las liberen. 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.

Adicionalmente en la Figura 6.4 se puede ver en forma general que con la estrategia de enfoques compartidos se obtienen mejores eficiencias que con la estrategia LH. Con SH la mejor eficiencia que se obtiene es con 4 hilos de ejecución (0.962 ms), mientras que con 2 y

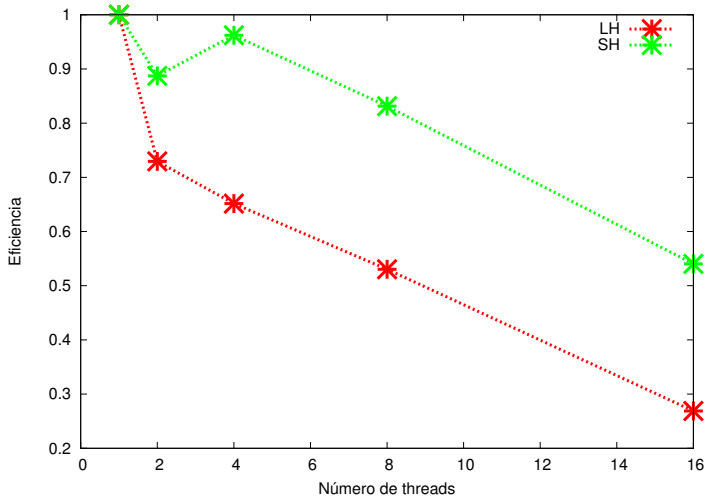


FIGURA 6.4: Eficiencias para Wand con heaps compartido y locales.

con 8 hebras se obtiene una eficiencia de 0.887 y 0.831 milisegundos; 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 *ms*) con respecto a las anteriores, esto se debe principalmente a la tecnología *hyperthreading* 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).

TABLA 6.1: Resultados método ML utilizando el conjunto de datos Gov2 y método de procesamiento Wand.

Estimador ML – GOV2 – WAND					
	1 hebra	2 hebras	4 hebras	8 hebras	16 hebras
r	0,8395167242	0,8487027167	0,8363882022	0,8220135644	0,8189326373
RMSE	89,8191624638	49,0317178648	22,0735989696	12,5486639038	9,1536416766

6.3 PREDICCIÓN DE TIEMPOS

En esta sección se muestran los resultados obtenidos de las implementaciones hechas para los métodos de predicción basados en regresión lineal múltiple (ML) y red neuronal (RN). El proceso de entrenamiento para ambos casos se llevó a cabo con 10000 instancias, cada instancia es una transaccion de lectura real desde la cual se precalcularon los 42 descriptores definidos en el Capítulo 4.

Las Tablas 6.1 y 6.2 muestran los valores obtenidos en el proceso de construcción de los modelos ML y RN en términos del coeficiente de Pearson y la raíz del error cuadrático medio (RMSE) en milisegundos; en ambos procesos mostrados en las tablas anteriores se utilizó el *dataset* Gov2 y el método Wand. Los resultados para los modelos restantes son presentados en el Anexo A.

A pesar de que son 42 variables independientes, en forma general se pueden observar buenos valores de los coeficientes de regresión de *Pearson* para cada uno de los modelos, lo que significa que existe una relación lineal entre el tiempo de las consultas y el modelo. Adicionalmente calculando el coeficiente de determinación, se puede notar que en el peor caso del modelo ML el porcentaje de variabilidad del tiempo explicado por el modelo alcanza un 67 % $((0.819^2) * 100)$ y en el caso del modelo RN alcanza un 79.5 % $((0.892^2) * 100)$. Con el conjunto de datos Clueweb se obtienen mejores coeficientes de correlación (ver Anexo A).

TABLA 6.2: Resultados método RN utilizando el conjunto de datos Gov2 y método de procesamiento Wand.

Estimador RN – GOV2 – WAND					
	1 hebra	2 hebras	4 hebras	8 hebras	16 hebras
r	0,9090139343	0,9081674727	0,8937170981	0,8920354081	0,8945066549
RMSE	67,7757284521	90,4311959153	66,5232866794	23,6964743214	7,6274795547

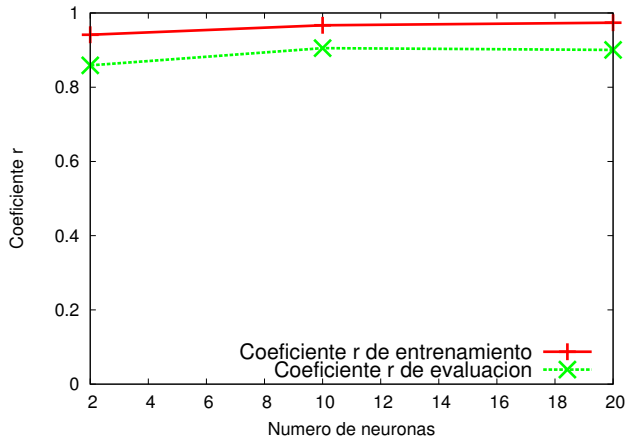
TABLA 6.3: Comparación de proceso entrenamiento versus proceso de validación, utilizado conjunto de datos GOV2 y método de procesamiento Wand

	Modelo ML		Modelo RN	
	Entrenamiento	Validación	Entrenamiento	Validación
RMSE	36,5253569757	49,405170938	51,2108329846	73,2588845444
ERP (%)	40,7309501561	48,528654817	75,1585277033	78,7569645709

A pesar de que los resultados mostrados anteriormente muestran que ambos modelos explican un porcentaje aceptable del fenómeno, la manera de evaluar estos métodos será mediante el porcentaje de error que se arroja para un segundo conjunto de consultas, usado exclusivamente para la evaluación. Se utilizaron dos conjuntos diferentes de 1000 consultas tanto de los *datasets* Gov2 como de Clueweb. Los parámetros utilizados para la evaluación fueron el RMSE (en milisegundos) y también el error relativo promedio porcentual (ERP) definido como $(\frac{ErrorAbsolutoMedio}{TiempoRealPromedio}) * 100$. La Tabla 6.3 muestra un resumen de los resultados obtenidos, aquí se muestra que tanto el RMSE como el ERP son menores para el modelo ML; lo que indica que el modelo multilineal generaliza de mejor manera para el conjunto de datos Gov2.

El detalle de los resultados obtenidos en el proceso de evaluación de ambos métodos de aprendizaje para los diferentes escenarios son mostrados en el Anexo B.

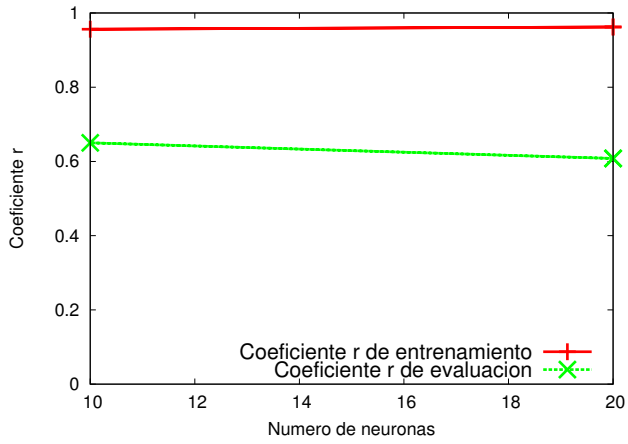
Finalmente con el objetivo de entender el por qué del valor de los errores obtenidos anteriormente para el modelo RN, se hizo un análisis exclusivo del coeficiente de correlación de este modelo con las dos muestras de la Web disponibles (Gov2 y Clueweb). Se tomó este



Clueweb		
Nro. neuronas	r entrenamiento	r evaluación
2	0,9413	0,8589
10	0,9669	0,9052
20	0,9738	0,9004

FIGURA 6.5: Valores del coeficientes de correlación para el *dataset* Clueweb.

estadístico para el proceso de entrenamiento y evaluación para distintos números de neuronas en la capa oculta (2, 10 y 20). Los valores obtenidos para la Clueweb se muestran en la Figura 6.5, en donde se puede apreciar que la diferencia entre el coeficiente de correlación de entrenamiento y el calculado desde el conjunto de evaluación no parece ser muy importante; sin embargo, cuando se observan los resultados para la Gov2 (Figura 6.6), se puede apreciar una gran diferencia entre sus coeficientes de correlación de entrenamiento y evaluación, por ejemplo, para 20 neuronas se observa un coeficiente de entrenamiento de 0.96, mientras que el de evaluación es 0.61. Lo explicado anteriormente muestra que el modelo RN es realmente dependiente de los datos, por lo que no es confiable utilizarlo en todos los escenarios. Además se puede observar que al aumentar significativamente el número de neuronas en la capa oculta, los resultados no son muy diferentes e incluso son peores cuando se ocupa los datos de Gov2, lo que podría ser muestra de un sobreentrenamiento del modelo. Sin embargo, esto no descarta una solución basada en redes neuronales, sino que es necesario encontrar un conjunto más preciso de descriptores.



Gov2		
Nro. neuronas	r entrenamiento	r evaluación
2	0,8995	0,6803
10	0,9563	0,6502
20	0,9623	0,6080

FIGURA 6.6: Valores del coeficientes de correlación para el *dataset* Gov2.

6.4 ESTRATEGIAS DE PROCESAMIENTO Y PLANIFICACIÓN

Se compararon las estrategias de planificación por bloques (Ye & Zhang, 2007) usando el predictor multilineal descrito en la sección 4.1 y además un predictor perfecto, el cual sabe de antemano el tiempo real de las consultas. Para determinar el número de hebras a ser usado en resolver cada consulta, el sistema predice el tiempo que esta tomará, y escoge el mínimo número de hebras necesarias con el que el tiempo predicho es menor o igual a la cota superior de tiempo establecida. Se experimentó con diferentes valores de cota superior y los mejores resultados fueron obtenidos con el tiempo promedio de las consultas. Para la resolución de consultas se utilizó el método Wand.

En la Figura 6.7 se puede observar el comportamiendo que tienen estas estrategias al resolver consultas divididas en conjuntos de diferentes tamaños (1000, 500, 200, 100 y 50). En cada caso las estrategias resolvieron un total de 10000 consultas con una pausa entre lotes, es decir, en el primer caso se resuelven 200 lotes de 50 consultas cada uno; en el segundo caso se resuelven 100 lotes de 100 consultas cada uno; así sucesivamente hasta llegar a procesar 10 lotes de 1000 consultas cada uno. El tiempo de ejecución total tomado corresponde a la suma

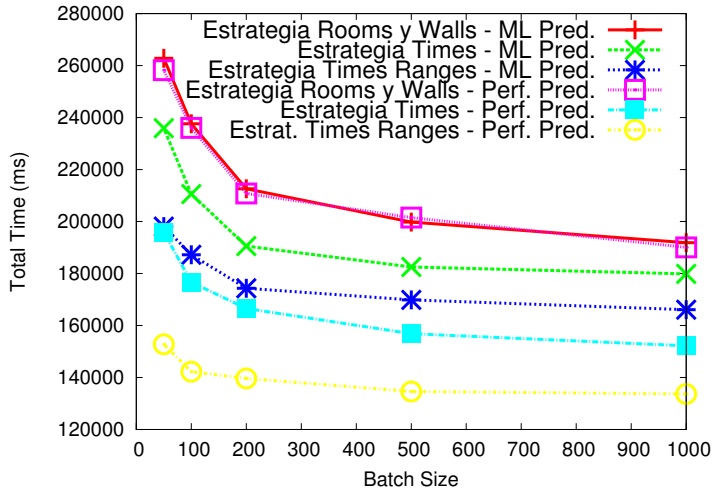


FIGURA 6.7: Tiempos de estrategias de planificación por bloques.

de tiempos de ejecución de los lotes. El mejor resultado fue obtenido por la estrategia *Times Ranges* usando el predictor perfecto, ya que al crear bloques de consultas que poseen rangos de tiempos similares, se disminiuye el tiempo pérdido por las hebras al final de cada uno de ellos.

Notar que en estrategias bajo el enfoque de bloques, un predictor con mayor precisión puede conducir a la reducción del tiempo de procesamiento de las consultas, por lo que el rendimiento de estas estrategias está supeditado a la calidad del predictor.

En base a experimentación se ha encontrado que en general procesar transacciones de lectura bajo un enfoque de bloques (Ye & Zhang, 2007) es mucho menos eficiente que con el enfoque de 1 hebra por consulta (1TQ) y unidades de trabajo (*Query Units*). Es por esta razón que en las siguientes experimentaciones y comparaciones, solo se utiliza aquella que arrojó mejores resultados: *Times Ranges* con predictor perfecto.

Posteriormente se hace una comparación *Times Ranges* con las estrategias de unidades de trabajo y 1TQ. Los resultados de dicha comparación se muestran en la Figura 6.8, aquí se ve claramente que bajo un contexto de un predictor perfecto, la estrategia de unidades de trabajo posee un mejor rendimiento que *Times Ranges*, ya que para todos los escenarios (lotes de diferentes tamaños), se requiere de menos tiempo para el procesamiento del conjunto total de

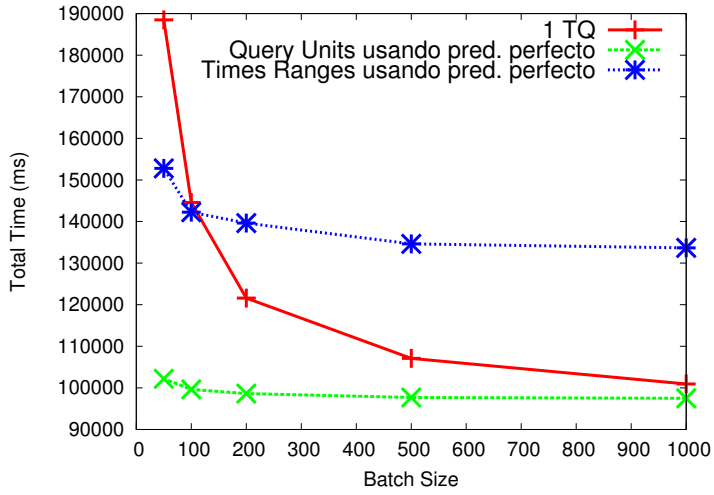


FIGURA 6.8: Tiempos de estrategias de planificación por bloques.

10000 consultas. Es importante notar que a medida que el tamaño de los lotes de consultas crece, la estrategia 1TQ se acerca al rendimiento de unidades de trabajo; esto se debe principalmente a que la estrategia 1TQ no posee un método inteligente de asignación de hebras a consultas, esto implica que eventualmente al final de cada lote, algunas hebras procesen consultas costosas en tiempo, retrasando al conjunto completo de hebras en la sincronización para comenzar el procesamiento del siguiente bloque. De esta forma, mientras mayor sea la cantidad de consultas por lotes, menor será el número de lotes a procesar y por ende menor el tiempo perdido por la estrategia 1TQ.

Finalmente se hizo una comparación del impacto que tienen los métodos de predicción ML y RN sobre el sistema de unidades de trabajo, estos resultados se pueden observar en la Figura 6.9; aquí se muestra que el uso de un predictor se hace importante a medida que el tamaño de los lotes decrece, puesto que es cuando la diferencia de tiempos entre 1TQ (que no usa una asignación de hilos de ejecución) y las otras dos estrategias se hace cada vez mayor, siendo mucho menores los tiempos de aquellas estrategias que sí usan un método de asignación de recursos basado en el tiempo predicho de la consulta (aunque este no sea del todo preciso). Por otro lado, a pesar de que existen diferencias de precisión entre ambos métodos (como se

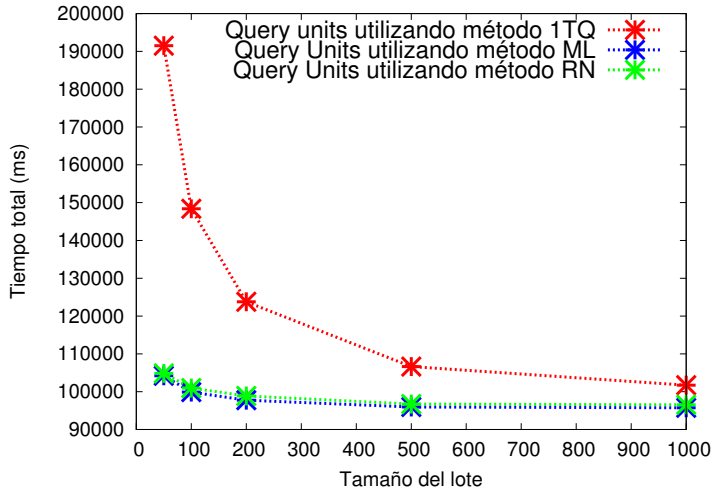


FIGURA 6.9: Tiempos de estrategias de planificación por bloques.

mostró en la sección anterior), en el enfoque de unidades de trabajo, esta diferencia no genera un impacto negativo importante en el tiempo total de procesamiento de las consultas; en otras palabras, el rendimiento de la estrategia propuesta no es demasiado sensible a los errores que puedan cometer los predictores de tiempo.

CAPÍTULO 7. CONCLUSIONES

Por medio del presente trabajo se ha llevado a cabo un estudio del procesamiento y planificación de transacciones de lectura que llegan a un motor de búsqueda haciendo uso de una máquina multinúcleo, en el que se adaptaron diferentes estrategias de planificación del estado del arte al contexto de un motor de búsqueda, y se evaluaron los rendimientos de cada una de ellas mediante el procesamiento de consultas por lotes. Adicionalmente se propone una estrategia de procesamiento de consultas basada en unidades de trabajo, en el que cada consulta es dividida en unidades de procesamiento y los diferentes hilos de ejecución compiten por procesar cada unidad.

El sistema implementado para resolver una transacción de lectura que llega al sistema, es flexible a hacer uso de diferentes números de hilos de ejecución; esta capacidad se logra debido a que se implementó dos versiones paralelas del algoritmo Wand, la primera es una versión con *heaps* locales y la otra hace uso de un solo *heap* compartido. Los resultados arrojan que utilizando la muestra de la Web Gov2 y obteniendo el conjunto de los *top-100* mejores documentos, el enfoque con *heap* compartido posee mejor rendimiento.

Con respecto a los predictores de rendimiento de transacciones de lectura, el método ML basado en una regresión lineal múltiple obtiene mejor rendimiento en la predicción que el método RN basado en redes neuronales, esto utilizando el mismo conjunto de 42 descriptores para ambos métodos. Lo mencionado anteriormente descarta que exista una mejor solución para el método RN que incluso pueda mejorar en rendimiento al método ML, para esto se debe hacer una mejor clasificación de descriptores utilizados para la creación del modelo.

Las estrategias de planificación por bloques no poseen un buen rendimiento cuando el objetivo es procesar grandes cantidades de transacciones de lectura por lotes, debido a la pérdida de tiempo que existe entre la sincronización de bloques. Por otro lado, la estrategia

de procesamiento de consultas por unidades de trabajo parece ser la mejor forma de reducir el tiempo total de procesar grandes cantidades de consultas por lotes y al mismo tiempo asegurar una cota superior de tiempo para cada una de ellas.

En condiciones ideales de predicciones de tiempo, la estrategia de unidades de trabajo posee mejor rendimiento que la estrategia 1TQ creada como *baseline*. A medida que el tamaño de los lotes crece, la diferencia de rendimiento es menor, debido a que existen menos sincronizaciones entre lotes, lo que implica una menor pérdida de eficiencia para la estrategia 1TQ.

En el presente contexto de procesamiento de transacciones de lectura en una máquina multicore, probablemente sea una mejor idea enfocar esfuerzos en crear un predictor más simple y preciso que los vistos en el presente trabajo por sobre intentar reordenar las consultas, de esta manera se obtendrán mejores tiempos para el procesamiento del conjunto completo de consultas.

Finalmente es posible afirmar que se ha cumplido con todos los objetivos planteados al comienzo de este trabajo. Se ha desarrollado dos algoritmos de procesamiento de transacciones de lectura basados en el algoritmo Wand, una con *heap* compartido y otra con *heaps* locales. Se han creado estrategias de planificación *online* que reordenan y adaptan dinámicamente las consultas al tamaño de las estructuras de datos disponibles; estas estrategias fueron adaptadas al contexto de un motor de búsqueda vertical en el que se procesan transacciones de lectura por lotes. Finalmente la evaluación para los métodos de predicción se hizo mediante la medición del RMSE y error relativo porcentual promedio (ERP); por otro lado, la evaluación para los métodos de planificación y procesamiento se hizo en base al tiempo que tarda cada una de ellas en procesar el conjunto completo de consultas.

7.1 TRABAJO FUTURO

Con respecto al trabajo futuro, sería interesante analizar el comportamiento que tienen las estrategias de procesamiento y de planificación para diferentes tamaños del conjunto *top-K* y el impacto que pueda tener sobre los métodos de aprendizaje. También es importante trabajar en la creación de un método de predicción con mayor precisión que los presentados en este trabajo, resulta interesante estudiar si es posible reducir el número de variables del modelo multilineal manteniendo o mejorando el error. Adicionalmente hacer un análisis detallado de cada uno de los descriptores utilizados y ver si es posible crear un método de predicción nuevo más simple y preciso. El modelo creado hasta ahora para procesar transacciones de lectura es flexible a hacer pausas entre los lotes; sería interesante agregar el procesamiento de las transacciones de escritura a este modelo y estudiar el comportamiento y el impacto que tienen estas transacciones sobre el sistema.

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., Arenas, M., Gutiérrez, C., Hurtado, C., Marín, M., Navarro, G., Piquer, J., Rodríguez, A., Ruiz-del Solar, J., & Velasco, J. (2008). *Cómo funciona la Web*. Centro de Investigación de la Web.

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

Baker, K. (1974). *Introduction to sequencing and scheduling*. Wiley.
URL http://books.google.cl/books?id=o8lTAAAAMAAJ

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

Bazewicz, J., & Al, E. (2001). *Scheduling Computer and Manufacturing Processes*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2nd ed.

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.

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`

Fausett, L. (Ed.) (1994). *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

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

Hauff, C. (2010). *Predicting the Effectiveness of Queries and Retrieval Systems*. Tesis de Doctorado, University of Twente, Enschede.

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`

Pinedo, M. L. (2008). *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3rd ed.

Rinnooy Kan, A. H. G. (1976). *Machine scheduling problems : classification, complexity and computations*. The Hague: Martinus Nijhoff. Result of a doctoral dissertation ... University of Amsterdam.
URL `http://opac.inria.fr/record=b1084866`

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`

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1988). Neurocomputing: Foundations of research. cap. Learning Representations by Back-propagating Errors, (pág. 696–699).

Cambridge, MA, USA: MIT Press.
URL http://dl.acm.org/citation.cfm?id=65669.104451

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`

APÉNDICE A. RESULTADOS DEL PROCESO DE ENTRENAMIENTO

TABLA A.1: Resultados método ML utilizando el conjunto de datos Gov2 y método de procesamiento Block Max Wand.

Estimador ML – GOV2 – BMW					
	1 hebra	2 hebras	4 hebras	8 hebras	16 hebras
r	0,8782952873	0,8809618279	0,8479348273	0,7771884041	0,7377811742
RMSE	72,364708101	40,8927943754	20,1217578763	13,7608115407	12,4521027766

TABLA A.2: Resultados método ML utilizando el conjunto de datos ClueWeb y método de procesamiento Wand.

Estimador ML – ClueWeb – WAND					
	1 hebra	2 hebras	4 hebras	8 hebras	16 hebras
r	0,8613156155	0,8726350536	0,8646059611	0,8598639269	0,8497258186
RMSE	91,9765237227	48,1189862101	21,9652740764	12,1717738001	9,3846426006

TABLA A.3: Resultados método ML utilizando el conjunto de datos ClueWeb y método de procesamiento Block Max Wand.

Estimador ML - ClueWeb – BMW					
	1 hebra	2 hebras	4 hebras	8 hebras	16 hebras
r	0,8828211665	0,8891976969	0,808606576	0,823249926	0,7451258225
RMSE	64,7039723565	35,281001295	25,7540777939	15,8306946733	17,9398672123

TABLA A.4: Resultados método RN utilizando el conjunto de datos Gov2 y método de procesamiento Block Max Wand.

Estimador RN – GOV2 – BMW					
	1 hebra	2 hebras	4 hebras	8 hebras	16 hebras
r	0,932476451	0,9360700621	0,8966995703	0,827613008	0,7880014511
RMSE	54,7912225707	82,2905244753	60,3315527261	21,882569362	5,7758056986

TABLA A.5: Resultados método RN utilizando el conjunto de datos Clueweb y método de procesamiento Wand.

Estimador RN – ClueWeb – Wand					
	1 hebra	2 hebras	4 hebras	8 hebras	16 hebras
r	0,9214415134	0,928326314	0,9547375955	0,9520042927	0,9498575917
RMSE	70,5610058313	98,6489306355	65,1112021339	24,172402818	8,4319553251

TABLA A.6: Resultados método RN utilizando el conjunto de datos Clueweb y método de procesamiento Block Max Wand.

Estimador RN – ClueWeb – BMW					
	1 hebra	2 hebras	4 hebras	8 hebras	16 hebras
r	0,9583572968	0,9581178412	0,8717897021	0,9019796766	0,8192397311
RMSE	39,546150466	75,4843974473	48,4467865615	24,9504558614	17,0429025714

APÉNDICE B. RESULTADO DEL PROCESO DE EVALUACIÓN DE LOS MODELOS DE APRENDIZAJE

TABLA B.1: Errores obtenidos método ML utilizando conjunto de datos Gov2 y algoritmo Wand

	Estimador ML				
	1 hebra	2 hebras	4 hebras	8 hebras	16 hebras
RMSE	93,4213321631	55,2226746394	35,8065152454	31,5809909101	30,9943417318
ERP (%)	46,7679492043	48,3620334358	49,158464109	54,3328274289	54,4780442408

TABLA B.2: Errores obtenidos método ML utilizando conjunto de datos Gov2 y algoritmo Block Max Wand.

	Estimador ML				
	1 hebra	2 hebras	4 hebras	8 hebras	16 hebras
RMSE	88,5781924513	50,2281785684	65,3589630276	85,0273335875	104,9557422762
ERP (%)	40,9396043829	43,5946339982	57,8712066627	73,6999382771	80,2156076147

TABLA B.3: Errores obtenidos método ML utilizando conjunto de datos Clueweb y algoritmo Block Max Wand.

	Estimador ML				
	1 hebra	2 hebras	4 hebras	8 hebras	16 hebras
RMSE	105,0379461837	55,108354518	32,3000081798	24,7425815335	29,7917310828
ERP (%)	38,2730089817	37,9789108856	39,2179670254	40,2465224632	47,8721024955

TABLA B.4: Errores obtenidos método ML utilizando conjunto de datos Clueweb y algoritmo Block Max Wand.

	Esimador ML				
	1 hebra	2 hebras	4 hebras	8 hebras	16 hebras
RMSE	63,2614531088	35,318259738	19,7612998424	18,0902452623	21,9105063561
ERP (%)	31,3197074579	32,8018021931	34,9491116301	33,1906288908	36,8597795426

TABLA B.5: Errores obtenidos método RN utilizando conjunto de datos Gov2 y algoritmo Wand.

	Estimador RN				
	1 hebra	2 hebras	4 hebras	8 hebras	16 hebras
RMSE	83,3427688489	127,7971510158	78,5508679211	44,0078238263	32,5958111096
ERP (%)	39,6467173103	109,5123072913	141,1360887399	123,600300482	62,1680304214

TABLA B.6: Errores obtenidos método RN utilizando conjunto de datos Gov2 y algoritmo Block Max Wand.

	Estimador RN				
	1 hebra	2 hebras	4 hebras	8 hebras	16 hebras
RMSE	75,2655924609	104,8966721749	64,0202898703	79,2415135861	100,5199807231
ERP (%)	33,2886231241	93,0690850804	70,2284168389	68,6799700357	77,8690931682

TABLA B.7: Errores obtenidos método RN utilizando conjunto de datos Clueweb y algoritmo Wand.

	Estimador RN				
	1 hebra	2 hebras	4 hebras	8 hebras	16 hebras
RMSE	89,9512834646	117,3863878347	62,5147780537	24,6101410147	23,1199566895
ERP (%)	35,065691535	108,6941782636	93,9594017925	52,8321098876	30,1810159761

TABLA B.8: Errores obtenidos método RN utilizando conjunto de datos Clueweb y algoritmo Block Max Wand.

	Estimador RN				
	1 hebra	2 hebras	4 hebras	8 hebras	16 hebras
RMSE	42,167317322	86,715284809	21,4293356118	49,3822498013	22,255611322
ERP (%)	22,1467849369	96,6915623431	40,6604395593	148,9848225291	40,2814151336