UNIVERSIDAD DE SANTIAGO DE CHILE

FACULTAD DE INGENIERÍA

DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

Estrategias de planificación para motores de búsqueda verticales

Danilo Fernando Bustos Pérez

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

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

SANTIAGO DE CHILE
2013

AGRADECIMIENTOS

Dedicado a ... .

RESUMEN

Resumen en Castellano

Palabras Claves: keyword1, keyword2 .

ABSTRACT

Resumen en Inglés

Keywords: keyword1, keyword2 .

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO 1. INTRODUCCIÓN

1.1 ANTECEDENTES Y MOTIVACIÓN

1.2 DESCRIPCIÓN DEL PROBLEMA

1.3 OBJETIVOS Y SOLUCIÓN PROPUESTA

1.3.1 Objetivo General

1.3.2 Objetivos Específicos

1.3.3 Alcances

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

A medida que pasa el tiempo y la Web sigue creciendo, los motores de búsqueda se convierten en una herramienta cada vez más importante para los usuarios. Estas máquinas ayudan a los usuarios a buscar contenido dentro de la Web, puesto que conocen en cuales documentos de la Web 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 Web que no tienen referencia desde otras páginas, siendo el único modo de acceder a ellas a través de un motor de búsqueda (Baeza-Yates et al., 2008).

Un motor de búsqueda está construído por diversos componentes. Su arquitectura típica se puede ver en la Figura 2.1. Existe un proceso denominado *crawling*, éste posee una tabla con los documentos Web 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 Web, 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).

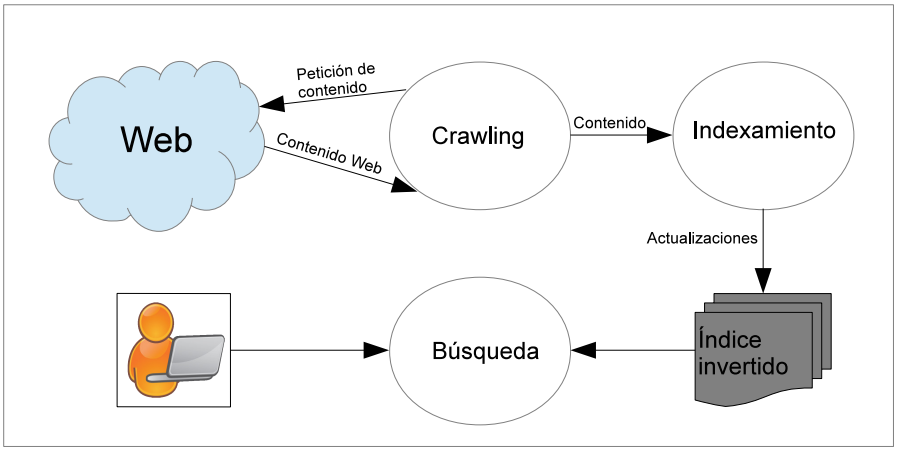

FIGURA 2.1: Arquitectura típica de un motor de búsqueda

Dado el volúmen de datos involucrado en el procesamiento, se debe tener una estructura de datos que permita encontrar cuáles documentos contienen las palabras presentes en la búsqueda que llega al sistema. Todo esto dentro de un período de tiempo aceptable. El índice invertido (Zobel & Moffat, 2006) es una estructura de datos que contiene un diccionario con todas las palabras que el proceso de *crawling* ha encontrado, asociado a cada palabra se tiene una lista de todos los documentos Web en donde esta palabra aparece mencionada (conocida como lista invertida de un término). El motor de búsqueda construye esta estructura con el objetivo de acelerar el proceso de las búsquedas que llegan al sistema. El proceso de búsqueda es el encargado de recibir las transacciones de lectura, generar un *ranking* de los documentos Web que contienen las palabras de la consulta y finalmente generar una respuesta. Las diversas formas de calcular la relevancia de un documento será explicado en secciones posteriores.

En un motor de búsqueda se pueden encontrar diversos servicios tales como (a) cálculo de los mejores documentos Web 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; (e) sugerencias en el momento que el usuario está escribiendo la consulta en el sistema; entre muchos otros servicios.

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

La diferencia entre un motor de búsqueda vertical y uno general, es que el primero se centra solo en un contenido específico de la Web. El *crawler* debe extraer contenido solo de aquellas páginas Web que están dentro del dominio permitido. Al ser un dominio acotado, los documentos Web a procesar serán menos y por lo tanto, la lista de los términos del índice invertido serán eventualmente de menor tamaño. Sin embargo, en un motor de búsqueda vertical las actualizaciones al índice invertido ocurren con mayor frecuencia.

2.2 ÍNDICE INVERTIDO

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

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

asociada correspondiente, se almacena en el índice invertido (ver Figura 2.2).

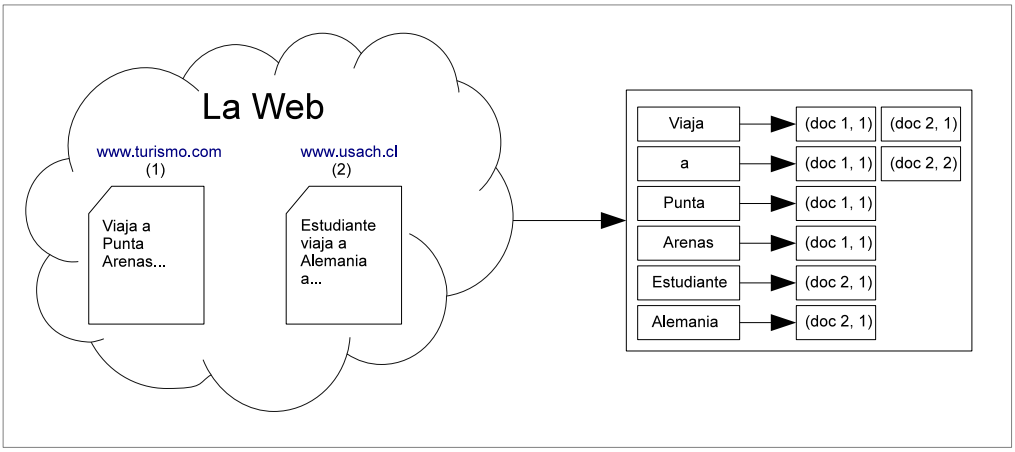

FIGURA 2.2: Índice invertido

El tamaño del índice invertido crece rápido y eventualmente la memoria RAM se agotará antes de procesar toda la colección de documentos. Cuando la memoria RAM se agota, se almacena en disco el índice parcial hasta aquel momento, 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).

2.3 ESTRATEGIAS DE EVALUACIÓN DE TRANSACCIONES DE LECTURA

Una de las tareas que un motor de búsqueda debe hacer para resolver una consulta es calcular el puntaje o *score* para aquellos documentos relevantes en la consulta y así poder

extraer los mejores k documentos. Existen dos principales estrategias para recorrer las listas invertidas y calcular el puntaje de los documentos para una determinada consulta. Estas son (a) *term-at-a-time* (Buckley & Lewit, 1985; Turtle & Flood, 1995) y (b) *document-at-a-time* (Broder et al., 2003; Turtle & Flood, 1995).

2.3.1 Term at a time

Abreviada TAAT, este tipo de estrategia procesa los términos de las consultas una a una y acumula el puntaje parcial de los documentos. Las listas invertidas asociadas a un término son procesadas secuencialmente, esto significa que todos los documentos presentes en la lista invertida del término t_i obtienen un puntaje parcial antes de comenzar el procesamiento del término t_{i+1}. La secuencialidad en este caso es con respecto a los términos contenidos en la transacción de lectura.

2.3.2 Document at a time

Abreviada DAAT, en este tipo de estrategias se evalúa la contribución de todos los términos de la *query* con respecto a un documento antes de evaluar el siguiente documento. Las listas invertidas de cada término de la consulta son procesadas en paralelo, de modo que el puntaje del documento d_j se calcula considerando todos los términos de la transacción de lectura al mismo tiempo. Una vez que se obtiene el puntaje del documento d_j para la consulta

completa, se procede al procesamiento del documento d_{j+1}. Este tipo de estrategia posee dos grandes ventajas: (a) Requieren menor cantidad de memoria para su ejecución, ya que el puntaje parcial por documento no necesita ser guardado y (b) Explotan el paralismo de entrada y salida (I/O) más eficientemente procesando las listas invertidas en diferentes discos simultáneamente.

2.4 FUNCIONES DE *RANKING*

Los sistemas de recuperación de información como los motores de búsqueda deben ejecutar un proceso el cual asigna puntaje a documentos con respecto a una determinada transacción de lectura, este proceso se denomina *ranking* (Baeza-Yates & Ribeiro-Neto, 2011). Como se puede ver en la Figura 2.3, este proceso toma como entrada la representación de las consultas y documentos, y asigna un *score* a un documento d_j dada una *query* q_i.

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

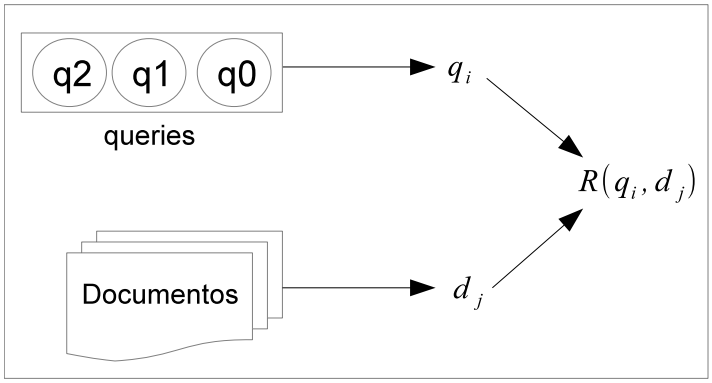

FIGURA 2.3: Proceso de *scoring* de documento

2.4.1 TF-IDF

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

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

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

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

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

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

2.5 OPERACIONES SOBRE LISTAS INVERTIDAS

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

2.5.1 *OR*

Este operador toma las listas invertidas de cada uno de los términos de la *query* y ejecuta la disyunción entre ellas. El resultado de este operador es una lista invertida con todos los documentos que contengan al menos un término de la *query*. Finalmente, esta lista invertida se ocupará para obtener los mejores K documentos. Un simple ejemplo se muestra en la Figura 2.4.

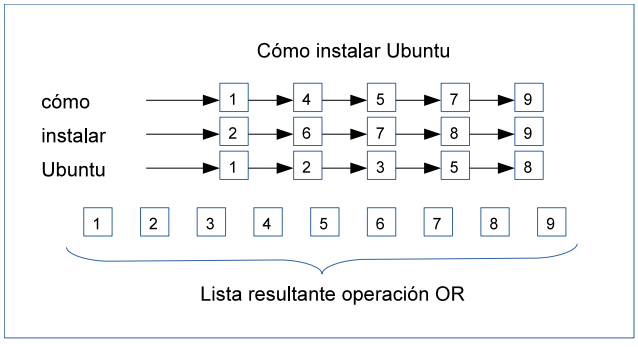

FIGURA 2.4: Operación OR

2.5.2 AND

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

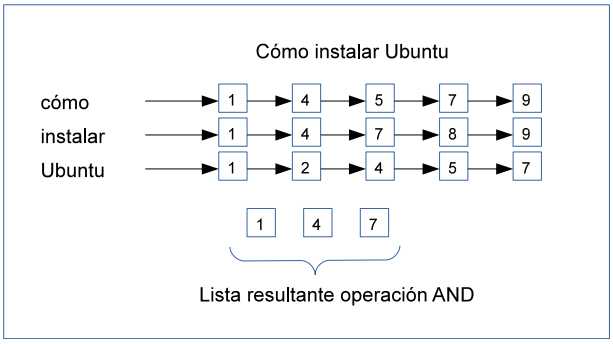

FIGURA 2.5: Operación AND

2.5.3 Wand

Algoritmo de evaluación de transacciones de lectura para obtener eficientemente el conjunto de K documentos que mejor satisfacen una consulta dada. *WAND* (Broder et al., 2003) es un proceso menos estricto que el método *AND* y está basado en dos niveles. Dentro del proceso de evaluación de una transacción de lectura, uno de los procesos más costoso en términos de tiempo es el de *scoring*, que consiste en entregarle a cada uno de los documentos analizados un puntaje que representa la relevancia del documento para una *query* dada, esto se denomina evaluación completa o cálculo del puntaje exacto del documento. El objetivo de

WAND es minimizar la cantidad de evaluaciones completas de los documentos ejecutando un proceso de dos niveles. En el primer nivel se intenta omitir rápidamente grandes porciones de las listas invertida, lo que se traduce en ignorar el cálculo del puntaje exacto de grandes cantidades de documentos, esto porque en motores de búsqueda a gran escala, este un proceso que requiere de mucho tiempo para llevarse a cabo y depende de factores como la cantidad de ocurrencia del término dentro del documento, el tamaño del documento, entre otros. A este tipo de técnicas que intenta omitir partes de lista invertida se les conoce como técnica de poda dinámica (Broder et al., 2003; Persin, 1994; Turtle & Flood, 1995).

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

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

En la Figura 2.6 se puede ver un ejemplo sencillo de cómo el algoritmo Wand trabaja en la resolución de una transacción de lectura de tres términos: 'casa', 'perro' y 'gato'. Como la *query* está compuesta por tres términos, existen tres punteros que recorren cada una de las

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

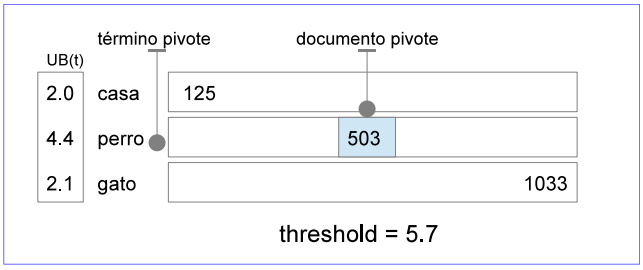

FIGURA 2.6: Ejemplo de ejecución de algoritmo *Wand*

2.5.4 Block Max Wand

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

Este método utiliza una variación del algoritmo Wand tradicional para que trabaje correctamente con la nueva estructura *Block-Max-Wand*. Remplazar el uso de los *upper bounds* por cada bloque por el *upper bound* global no garantiza la correctitud del algoritmo. En la Figura 2.7 se muestra un ejemplo de por qué mirar solo los *tupper bounds* locales no garantiza obtener los resultados correctos, aquí no se puede concluir que el documento 4868 es el documento más pequeño que puede estar dentro del conjunto *top-K*, ya que $2.5 + 2.0 + 3.5 \geq 7.0$ (conclusión que sí es válida utilizando Wand tradicional y *upper bounds* globales), porque es posible que el bloque siguiente al bloque del docID 275 (en la primera lista), tenga un *upper bound* local mayor. Por lo tanto, aplicar solo las máximas contribuciones por bloque no permite al algoritmo omitir documentos de forma segura.

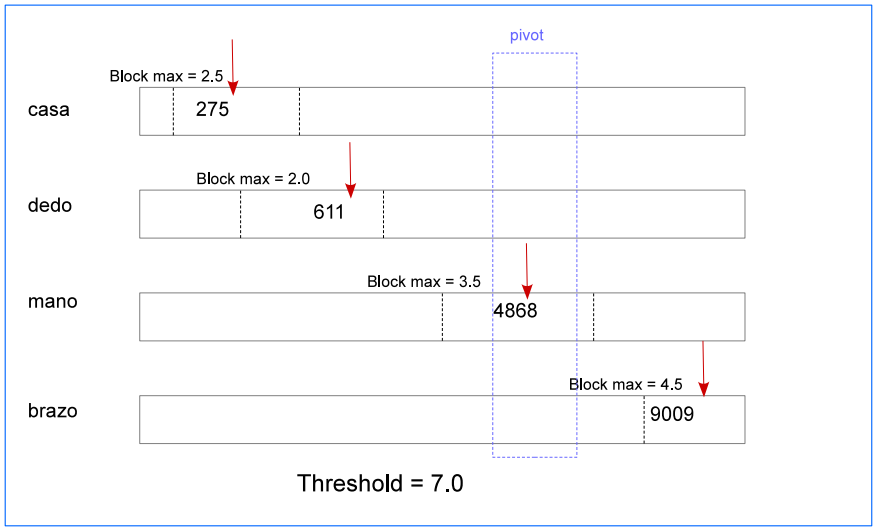

FIGURA 2.7: Ejemplo del proceso Block-Max-Wand

2.6 PREDICCIÓN DE TIEMPO DE RESPUESTAS DE TRANSACCIONES DE LECTURAS

El rendimiento (*performance*) de una consulta puede medirse de dos formas: Efectividad y eficiencia. La efectividad tiene relación con la calidad de los documentos extraídos para una cierta consulta y la eficiencia corresponde al tiempo que conlleva procesarla. El tiempo que le toma al sistema de recuperación de la información en resolver una consulta puede variar considerablemente. Con el objetivo de retornar los resultados al usuario dentro de una cota superior de tiempo, aquellas consultas que toman una mayor cantidad de tiempo en ser procesadas se requiere una mayor cantidad de procesadores para resolverla, de esta forma podemos asegurar esta cota de tiempo. El tener un buen predictor de la eficiencia de una *query* es muy útil, por ejemplo, si pensamos en un sistema con réplicas, podemos planificar la consulta en el servidor que se desocupará más pronto.

Existen estudios en los cuales el rendimiento es inferido usando *clarity score* (Cronen-Townsend et al., 2002), que es una forma para evaluar la pérdida de ambigüedad de una

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

2.7 SCHEDULING EN MOTORES DE BÚSQUEDA

Los sistemas de recuperación de la información como los motores de búsqueda, no solo se preocupan de la calidad de los resultados de las búsquedas (efectividad), sino que también de la velocidad con la que los resultados son obtenidos (eficiencia). Existen varias estrategias para mejorar la velocidad en la obtención de los resultados, una de ellas muy utilizada es el *caching*. Consiste en guardar en memoria de acceso rápido (memoria caché) datos temporales, que luego pueden ser sobrescrito. Una opción es hacer *caching* de los resultados de las búsquedas, de esta forma cuando una *query* es encontrada en caché el motor de búsqueda puede generar la respuesta rápidamente, reduciendo los tiempos de calculos considerablemente. Otra opción es,

guardar en caché la intersección de las listas invertidas de pares comunes de términos que llegan al motor de búsqueda. Por ejemplo, si llega al sistema una consulta con los términos ('casa', 'árbol', 'perro'), se puede guardar en caché la intersección de las listas de 'casa' y 'árbol', para luego reutilizar esta información en otras *queries* que lleguen en el futuro. Para ver más técnicas de *caching* y ver el detalle de las técnicas mencionadas, ver (Büttcher et al., 2010).

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

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

En el contexto del presente trabajo de tesis, el objetivo de hacer *scheduling* es minimizar el tiempo en que las *queries* son procesadas por un motor de búsqueda. Los motores de búsqueda como *Google*[1] o *Yahoo!*[2] trabajan en un contexto *online*. Esto significa que cuando las *queries* llegan al sistema (una a una), éste está obligado a tomar una decisión para planificarla sin saber cuáles *queries* llegarán en un momento posterior. A esto se le conoce como algoritmo de *scheduling online* (Albers, 2003; Borodin & El-Yaniv, 1998).

Los sistemas IR a gran escala despliegan una arquitectura distribuída (Dean, 2009), en donde el índice invertido está particionado (Barroso et al., 2003) a lo largo de servidores (*shard servers*), los cuales están encargados de procesar las *queries* que llegan al sistema. Es fácil notar que resolver una *query* con varios *shard servers* mejoraría la eficiencia. Ahora bien, para

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

asegurar un alto rendimiento (*throughput*) del sistema, cada uno de los *shard servers* posee réplicas, de esta forma, más *queries* pueden ser procesadas en paralelo en copias idénticas del mismo *shard server*. Esto implica que el tiempo de espera de las *queries* que vienen llegando al sistema se reduce.

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

2.7.1 Trabajo relacionado

El estudio (Broccolo et al., 2013) analiza métodos de *dropping* y *stopping* para el procesamiento de *queries* bajo altas carga de trabajo en un sistema distribuído donde existen múltiples servidores en el que cada uno resuelve una parte de la *query* para luego enviar las consultas al *broker* y éste hace el *merge* de los resultados de acuerdo al *score* de los documentos. Se define un tiempo T, en el que la suma de el tiempo de espera de la *query* para ser procesada (t_w) y el tiempo de procesamiento de la misma (t_p) deben ser menor a T. Si es que se sobrepasa este tiempo, se tienen dos opciones (1) la *query* es desechada y se envía al *broker* una lista vacía, (2) se detiene el procesamiento de la *query* y se envía los resultados parciales hasta el momento. Finalmente se propone un método basado en la predicción de tiempo de respuesta ($\hat{pt}(q)$) de una *query* (Macdonald et al., 2012) de modo que si se cumple $\hat{pt}(q) \leq T - wt(q)$,

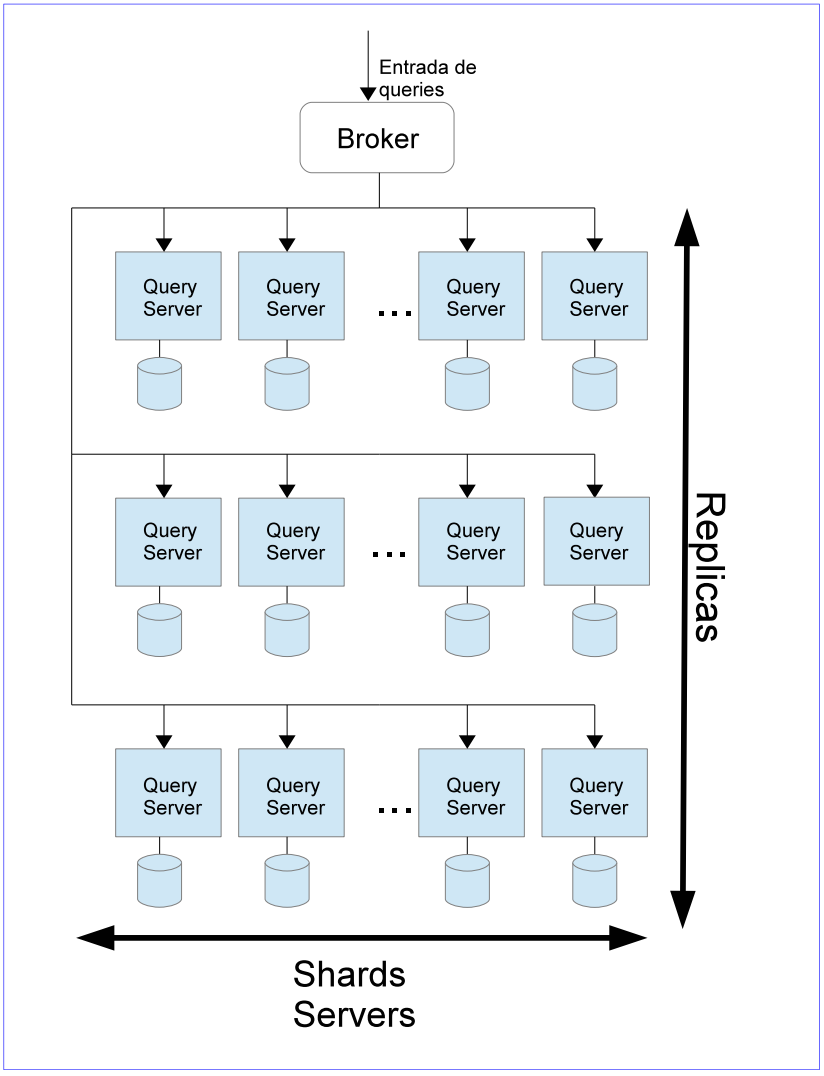

FIGURA 2.8: Arquitectura de un sistema de recuperación de la información con réplicas

entonces la *query* es desechada antes de comenzar a procesarse y se toma la siguiente desde la cola de espera. Notar que en estos métodos existe una pérdida de efectividad, puesto que eventualmente los servidores muchas veces no enviarán sus mejores documentos al *broker*, esto implica que el *broker* responderá al usuario un conjunto de K documento que no necesariamente son los mejores dentro del corpus completo.

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

CAPÍTULO 3. ESTRATEGIAS DE PLANIFICACIÓN DE QUERIES

Decir por lotes

3.1 PREDICCIÓN DE RENDIMIENDO DE TRANSACCIONES DE LECTURA

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

Conocer de antemano la eficiencia de una *query* es una ventaja muy importante, puesto que aquellas consultas que tomaran una mayor cantidad de tiempo en ser resueltas se les puede asignar un mayor número de *threads* para procesarla, de esta manera se reduce el tiempo de

procesamiento de las consultas y se cumple con la cota superior de tiempo prometida al usuario. Adicionalmente, que un sistema de recuperación de la información como un motor de búsqueda conozca anticipadamente cuánto tardará una consulta en ser procesada, permite implementar técnicas efectivas de planificación de transacciones de lecturas, por ejemplo, en el contexto de procesamiento paralelo de *queries* por lotes (*batches*) se pueden crear grupos de consultas que posean tiempos de respuesta parecidos, así se tiende a disminuir tanto el desbalance de carga entre los procesadores como el tiempo en procesar el *batch* completo.

A continuación se presenta la implementación de métodos de predicción de eficiencia de queries que llegan a un motor de búsqueda.

3.1.1 Método de predicción Glasgow

El presente método (Macdonald et al., 2012) se basa en estadísticos obtenidos previamente calculados desde el índice invertido. Los puntajes de los documentos son obtenidos mediante el método BM25. Para cada consulta que llega al sistema, se toman los términos y para cada uno de ellos desde su propia lista invertida se obtiene los siguientes estadísticos $s(t)$:

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

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

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

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

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

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

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

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

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

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

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

Número de inserciones en el conjunto de los mejores K documentos. Para obtener este estadístico se asume que el término t es una consulta con un solo término, se resuelve esta *query* con el método *Wand* y se calcula el número de inserciones de documentos que se hizo al *heap*. Recordar que las inserciones al *heap* ocurren cuando el puntaje aproximado del documento supera el puntaje más bajo que en el *heap* en ese momento (umbral o *threshold*).

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

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

Cada uno de los 14 estadísticos mencionados anteriormente están relacionados linealmente con la eficiencia de una transacción de lectura y son la base para la implementación del predictor, que está basado en un modelo de regresión lineal múltiple (Chambers, 1991). A continuacion se muestra un resumen de estos estadísitcos en la Tabla 3.1.

TABLA 3.1: Resumen de los estadísticos que se deben extraer desde el índice invertido

Estadísticos del término s(t)
1. Media aritmética
2. Media geométrica
3. Media harmónica
4. Máximo puntaje
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 umbral
12. Número de incerciones al conjunto top-K
13. IDF
14. Tiempo en resolver t como *query*

3.1.2 Método de predicción SIGIR

En el contexto de un motor de búsqueda una consulta se puede clasificar como larga y como corta. Una query larga es aquella que toma un tiempo considerable en ser procesada y una query corta es aquella que se puede resolver rápidamente. Cuando un sistema está con una

baja carga de trabajo paralelizar todas las consultas no tiene un impacto en el rendimiento, sin embargo, cuando un sistema está sometido a una moderada o alta carga de trabajo paralelizar todas las queries entrantes es ineficiente (Jeon et al., 2014), debido a que el costo de paralelizar queries cortas es alto.

3.2 WAND *MULTI-THREADED*

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

Existen dos formas de implementar Wand *multithreaded*. Una de ellas es usando *heaps* locales (LH), es decir, un *heap* por hilo de ejecución (*thread*) y el otro es usando *heaps* compartidos (SH). El estudio en (Rojas et al., 2013) se muestra indicios que el esquema SH es generalmente más eficiente. Logrando rápidamente un óptimo valor para el threshold, el

esquema SH posee las siguientes ventajas: (1) Se puede reducir el número de calculo de puntajes completos y (2) se ejecutan pocas operaciones de actualización del heap (reduciendo el número de *locks* que se hace a la estructura de dato). A continuación se presenta el diseño llevado a cabo para ambos esquemas.

3.2.1 Block max wand

Recordar que en el método de Wand para descartar documentos y encontrar un documento que potencialmente podría estar en el conjunto *top-K*, lo que se hace es usar los *upper bounds* globales de cada lista, es decir, la máxima contribución (puntaje o *score*) de algún documento de la lista invertida. Además, Wand tradicional es una estrategia DAAT, por lo que por cada lista invertida ocupa un puntero al documento actual que se desea evaluar. Además, usa un método que recibe como entrada un identificador del documento $docID$ y una lista invertida L, y retorna el primer $docID'$ que sea mayor o igual al documento $docID$. A esto se le conoce como movimiento de puntero profundo (*deep pointer movement*) debido a la razón que generalmente implica una descompresión del boque en el que se encuentra el documento.

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

una descompresión de bloque. Se debe notar que para que esta función trabaje correctamente se requiere tener almacenada las fronteras de cada uno de los bloques de las listas invertidas.

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

En el Algoritmo 3.1 se puede apreciar cómo el método *Block-Max-Wand* trabaja. Recordar que todas las listas invertidas poseen un puntero al documento actual que se desea evaluar (*currentDoc*). Lo primero que se hace es ordenar en orden creciente las listas invertidas de acuerdo a su correspondiente *currentDoc*. La función *findPivot()* es la misma que se utiliza en el método Wand tradicional (2.5.3), se itera sobre las listas invertidas y se retorna la posición de la lista en donde se cumple que la suma de los *upper bounds* globales es mayor al *threshold* (θ). Luego la función *NextShallow()* se encarga de avanzar los punteros de las listas invertidas al inicio del bloque que debería contener el documento d. Posteriormente la función *isRealPivot()* verifica si es que el pivote p encontrado es un pivote real o no, para cada una de las listas desde la posición 0 hasta la posición p, se suma los *upper bounds* de los bloques en donde se encuentran los punteros (recordar que con *NextShallow()* los punteros de las listas quedaron apuntando a los bloques en donde se debería encontrar el documento d), si la suma es mayor al threshold entonces retorna verdadero, de lo contrario retorna falso. El método *scoreDoc()* calcula el puntaje del documento que se le pasa por parámetro.

Cuando el método se da cuenta que p no es un pivote real, lo que se hace es buscar un nuevo candidato a través de la función *getNewCandidate()*, la cual hace avanzar los punteros de las listas invertidas hasta el bloque siguiente que contenga el mínimo $docID$. Para ver explicar de mejor manera esta idea se presenta la Figura 3.1, aquí se puede ver que el documento 4868 es el pivote, cuando este documento no es un pivote real (la función *isRealPivote* retorna falso), lo que se hará es escoger un documento d' tal que $d = min(d1, d2, d3, d4)$ en donde $d1, d2, d3$

son la frontera del bloque actual más uno (inicio del bloque siguiente) y $d4$ es el *currentDoc* de la cuarta lista. Notar que para hacer un descarte seguro de documentos, siempre se debe incluir a la elección del nuevo candidato el *currentDoc* de la lista inmediatamente siguiente a la lista pivote (en este caso 9009).

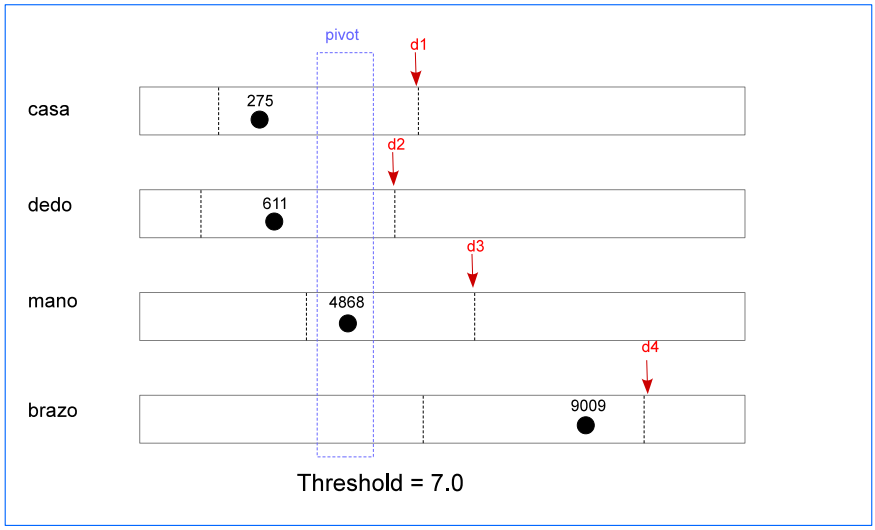

FIGURA 3.1: Ejemplo de cómo opera la función getNewCandidate()

3.2.2 Wand con heaps locales

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

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

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

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

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

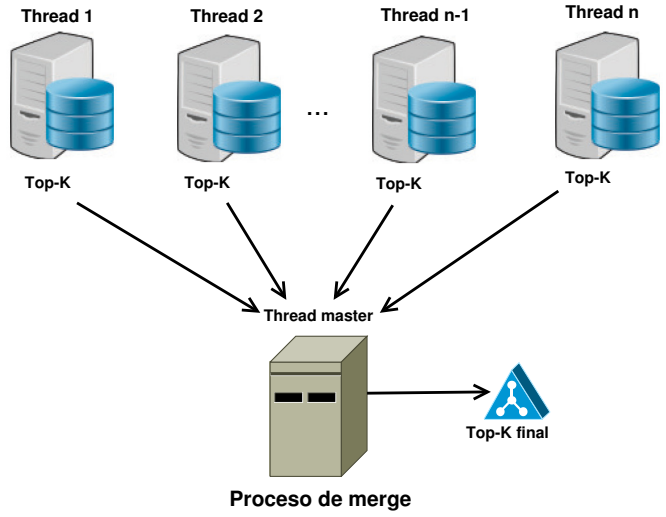

FIGURA 3.2: Esquema de ejecución de algoritmo WAND con heaps locales

las clases involucradas en la implementación, se presenta el siguiente diccionario de datos.

TopKMultiThreadWandOperatorLocal. Clase encargada de devolver los mejores K documentos para una query dada. Si es que la query debe ser resuelta en forma paralela, esta clase además debe controlar el paralelismo que se produce en la resolución de ésta, inicializando las variables correspondientes para lanzar los hilos de ejecución y luego escogiendo los mejores documentos desde todos los heaps creados por los diferentes threads (proceso de merge). En esta clase se define un mapa que asocia cada término del índice invertido con el puntaje del mejor documento en esa lista invertida (upper bound de la lista invertida) y además se define cuántos documentos se van a retornar al final del proceso (atributo K). El método execute inicializa las variables locales para los diferentes threads, posteriormente hace el llamado al método *thread-execute* (en el cual se llevará a cabo la resolución de la transacción de lectura en forma paralela), finalmente se toman los resultados parciales de cada uno de los hilos de ejecución y se ejecuta el proceso que mezcla los resultados, retornando solo los mejores K documentos.

PartitionedInvertedIndex. Clase que tiene la tarea de almacenar el índice invertido y extraer desde aquí las listas invertidas de documentos para cada uno de los términos

de las transacciones de lectura. El almacenamiento el índice se lleva a cabo mediante un mapa cada término su lista invertida correspondiente y para la extracción de estas listas se usa el método getList.

TopKWandOperator. Cada thread tendrá su propio objeto TopKWandOperator encargado de obtener los mejores K documentos. El cálculo de este conjunto se realiza en el método execute con la ayuda de un objeto de tipo Wand asociado.

Wand. Clase que controla la lógica del algoritmo wand. Lleva a cabo el proceso de inserción de documentos en el heap y todo lo que esto conlleva. Existen diferentes tipos de objetos Wand que se pueden utilizar, entre ellos están WandBM25, WandFrec y WandTFIDF, donde la única diferencia entre ellos es el método de que calcula el puntaje de cada documento. Por ejemplo, WandBM25 utiliza BM25 (citar) y WandTFIDF utiliza tf-idf (citar también).

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

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

3.2.3 Wand con heap compartido

En el esquema SH cada thread procesa una porción del índice. Sin embargo, ahora un solo heap es creado y accedido por todos los threads. En este caso no se requiere de mezclar los

resultados y el proceso de descarte tiende a ser más eficiente porque los documentos con mayor puntaje tienden a estar en el heap. Acceder al heap debe ser controlado por un lock o algún método similar que garantice el acceso exclusivo de los threads al heap. Este esquema es más eficiente que el LH en queries que toman mayor tiempo en ser resueltas.

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

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

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

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

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

3.3 ESTRATEGIA *BASELINE*

Un simple camino para construir un sistema que responda a múltiples consultas simultáneamente usando múltiple hilos de ejecución, es usando estos hilos de manera independiente. Para hacer esto se debe mantener un conjunto de *threads* consumidores que trabajarán en paralelo y se encargarán de resolver las *queries* secuencialmente (una a una) desde una misma cola, esto es lo que en este trabajo se denomina estrategia de Un Thread Por Query (1TQ). En la Figura 3.6 se puede apreciar el esquema de ejecución en donde cada uno de los procesos genera una petición de alguna consulta en la cola, si quedan *queries* por procesar entonces se le asigna al proceso una consulta que tendrá que resolver de manera secuencial. Se debe tener en cuenta que cada vez que un proceso genera una solicitud de *query*, se bloquea la estructura de datos que contiene las consultas a procesar y luego se procesa la solicitud, de esta forma se asegura un acceso seguro por parte de los distintos *threads*.

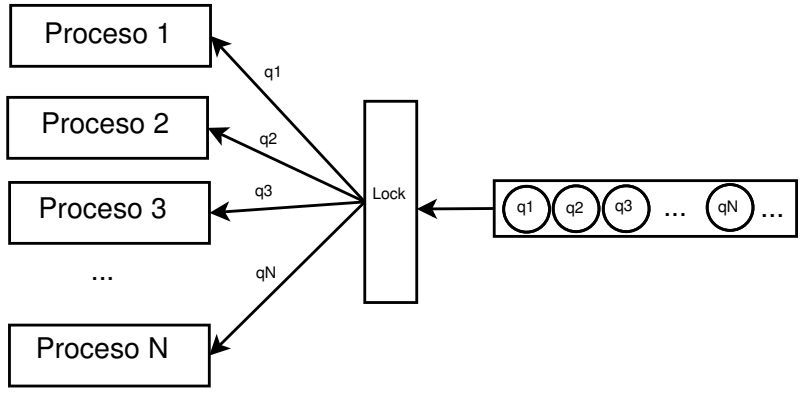

FIGURA 3.6: Ejemplo de procesamiento estrategia 1TQ

Este esquema tiene la ventaja que es simple y fácil de implementar y controlar. Sin embargo, existen sistemas de recuperación de la información como los motores de búsqueda verticales que cuando están ejecutando *batches* de *queries* deben parar su ejecución porque transacciones de escritura han llegado al sistema, y este deben actualizar la información del

índice invertido. Solo después de la fase de actualización el sistema es capaz de ejecutar el siguiente *batch* de transacciones de lectura. Al final de cada conjunto de consultas, es posible que algunos hilos de ejecución del sistema finalicen su trabajo y que no tengan más *queries* para procesar, por lo que ellos tienen que esperar que los *threads* restantes finalicen su trabajo antes que el sistema entre en la fase de actualización de su índice invertido o bien, se pase a la ejecución del siguiente *batch* de consultas. Sin embargo, aunque cada hilo de ejecución está secuencialmente ejecutando una transacción de lectura diferente, algunas de estas operaciones puede tomar un tiempo cosiderable, de esta forma se produce una importante pérdida de eficiencia, aunque la intuición nos dice que esto se puede mitigar con *queries* que requieran poca cantidad de tiempo para ser procesada (trabajos pequeños o *small jobs*). En la Figura 3.7 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 3.8).

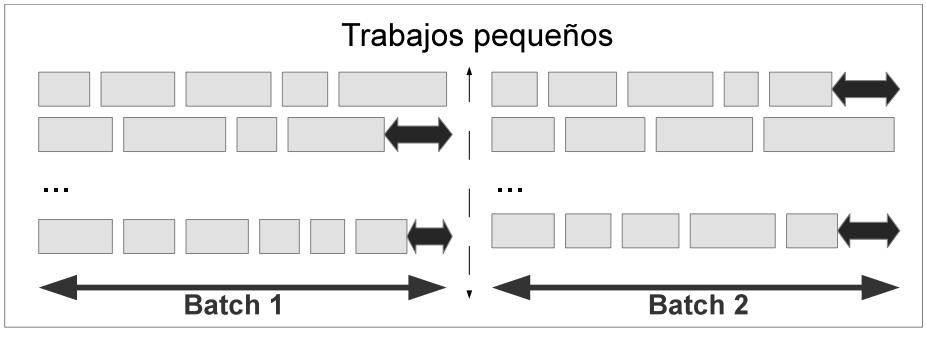

FIGURA 3.7: Ejecución en paralelo de *small jobs*

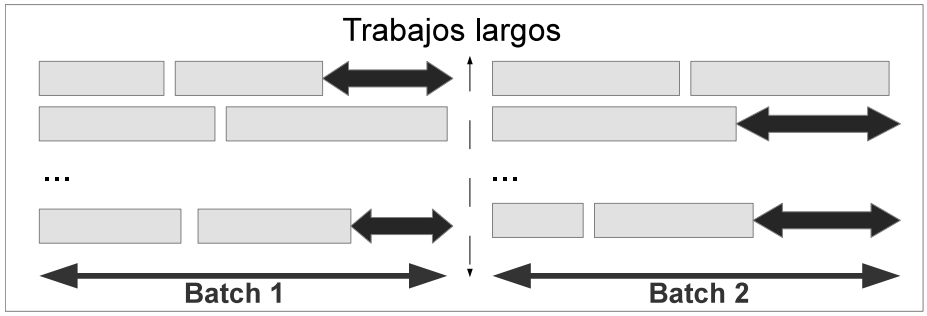

FIGURA 3.8: Ejecución en paralelo de *large jobs*

3.4 ESTRATEGIAS DE *SCHEULING*

nosotros optamos por un enfoque de Wand Heap Compartido para ser usado en los experimentos.

3.5 ESTRATEGIA DE 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 un *batch* de consultas está siendo procesado. En esta sección se propone un esquema híbrido basado en unidades de procesamiento (*Processing Units*) que aproveche las ventajas de ambos enfoques. (se requiere ver el tema de bloques). En este nuevo esquema de planificación, las consultas pasan a través de una fase en la cual cada *query* es evaluada

y se determina un apropiado número de unidades de procesamiento (*processing units*) para poder resolver dicha consulta. Este proceso es llevado a cabo de manera similar al proceso en donde se determina la cantidad de *threads* apropiados para resolver una determina transacción de lectura. Este número de unidades de procesamiento es creado y asociado a cada consulta, finalmente se guarda en una cola de unidades de trabajo. Un conjunto de *threads* consumidores extraen las unidades desde la cola y las procesa independientemente. Cuando un *thread* finalice el procesamiento de la unidad de trabajo actual automáticamente leerá la siguiente unidad de trabajo desde la cola. Generalmente lo que se hace habitualmente es estimar el número de *threads* con el que se resolverá la consulta, como se muestra en la Figura 3.9 en este nuevo enfoque se intenta estimar el número de unidades de trabajo con el que se resolverá cada consulta. Además, se debe controlar el acceso concurrente de los hilos de ejecución a la cola de unidades de trabajo, de tal manera que solo un thread tenga acceso exclusivo a la estructura de datos.

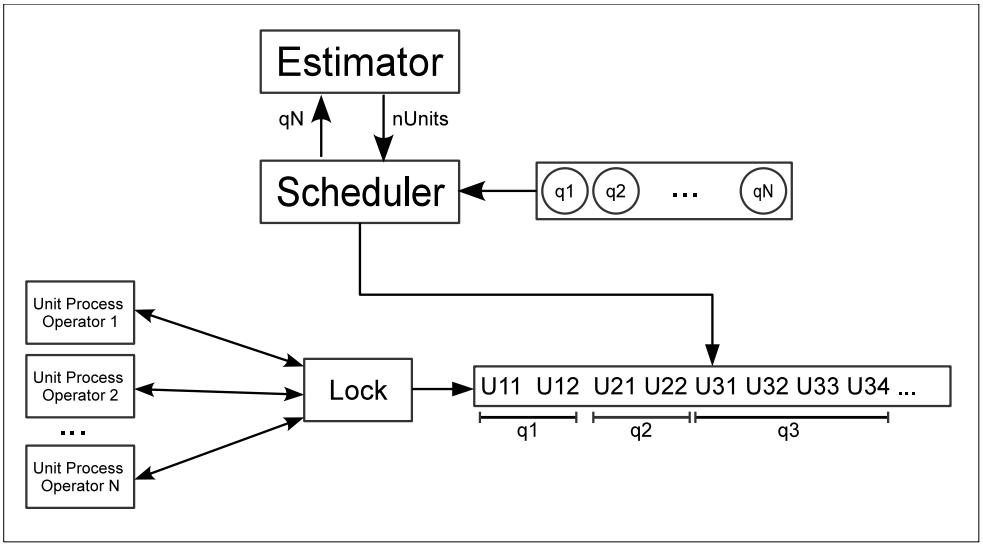

FIGURA 3.9: Procesamiento de consultas utilizando unidades de trabajo

El procesamiento de cada hilo de ejecución es una versión de Wand con heap compartido (SH), adaptado de manera tal que cada unidad de trabajo es resuelta independientemente de si existen otras unidades siendo procesada al mismo tiempo o no. La única excepción es que la

unidad que inicializa la consulta es siempre ejecutada antes del resto de las otras unidades de la misma consulta y la entrega de resultados se hace una vez que todas las unidades de trabajo de la *query* han finalizado. Este enfoque híbrido permite reducir el tiempo perdido al final de cada *batch* sin generar una importante pérdida de trabajo mientras las *queries* del *batch* están siendo procesadas.

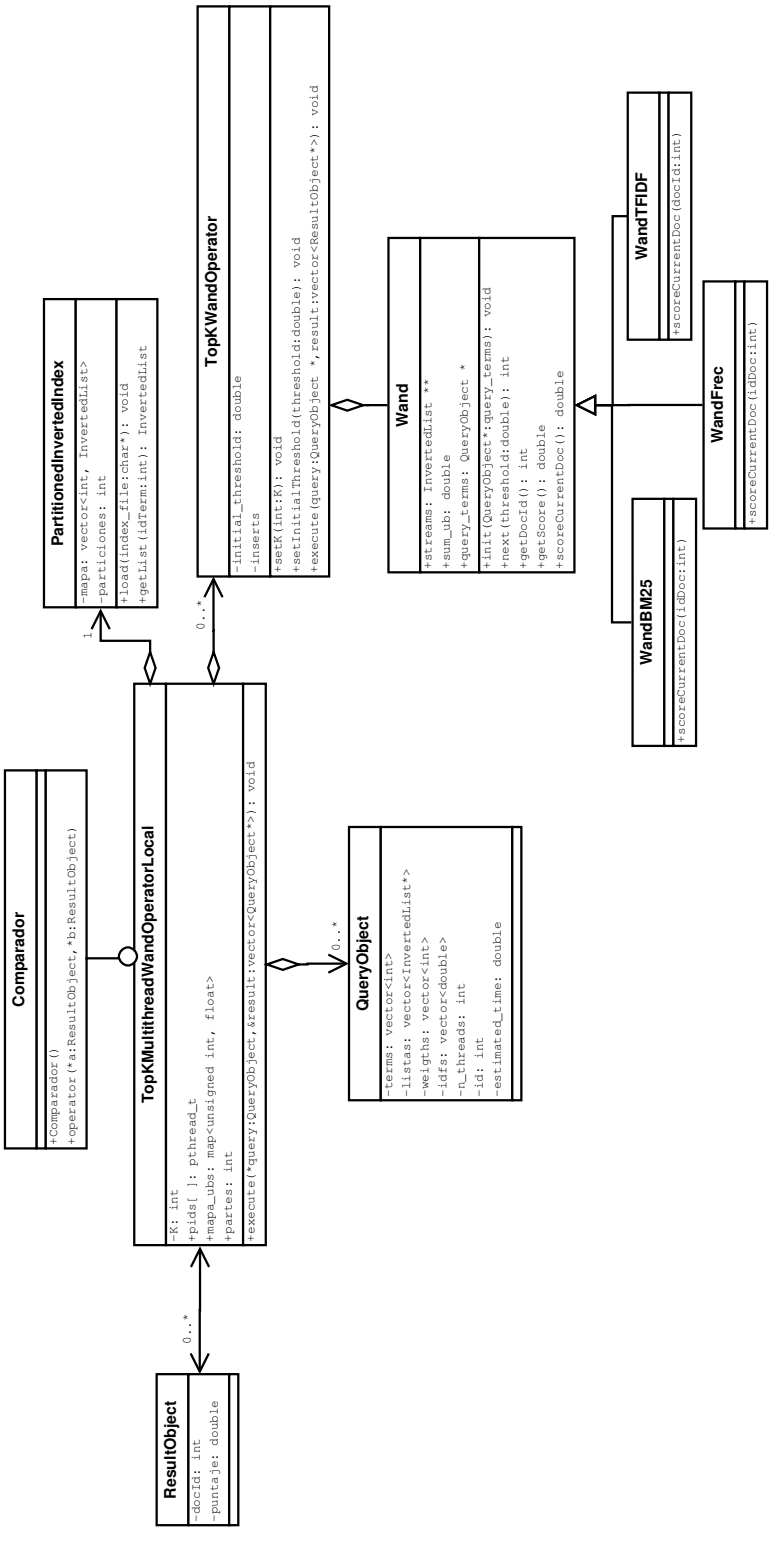

FIGURA 3.3: Diagrama de clases para el esquema LH

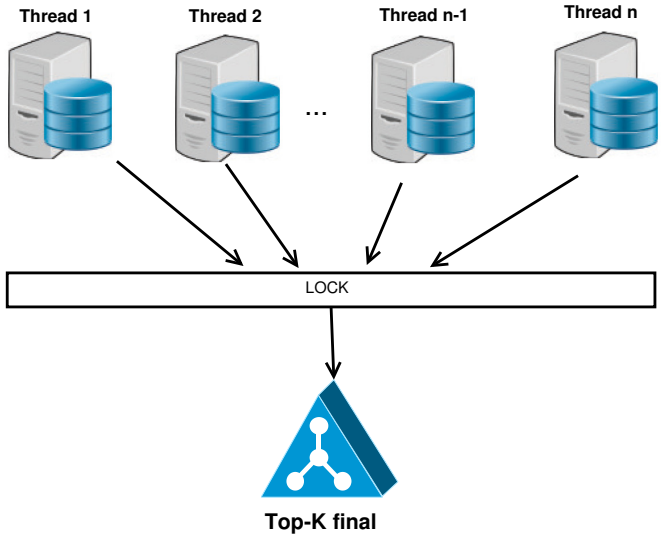

FIGURA 3.4: Esquema de ejecución de algoritmo WAND con heap compartido

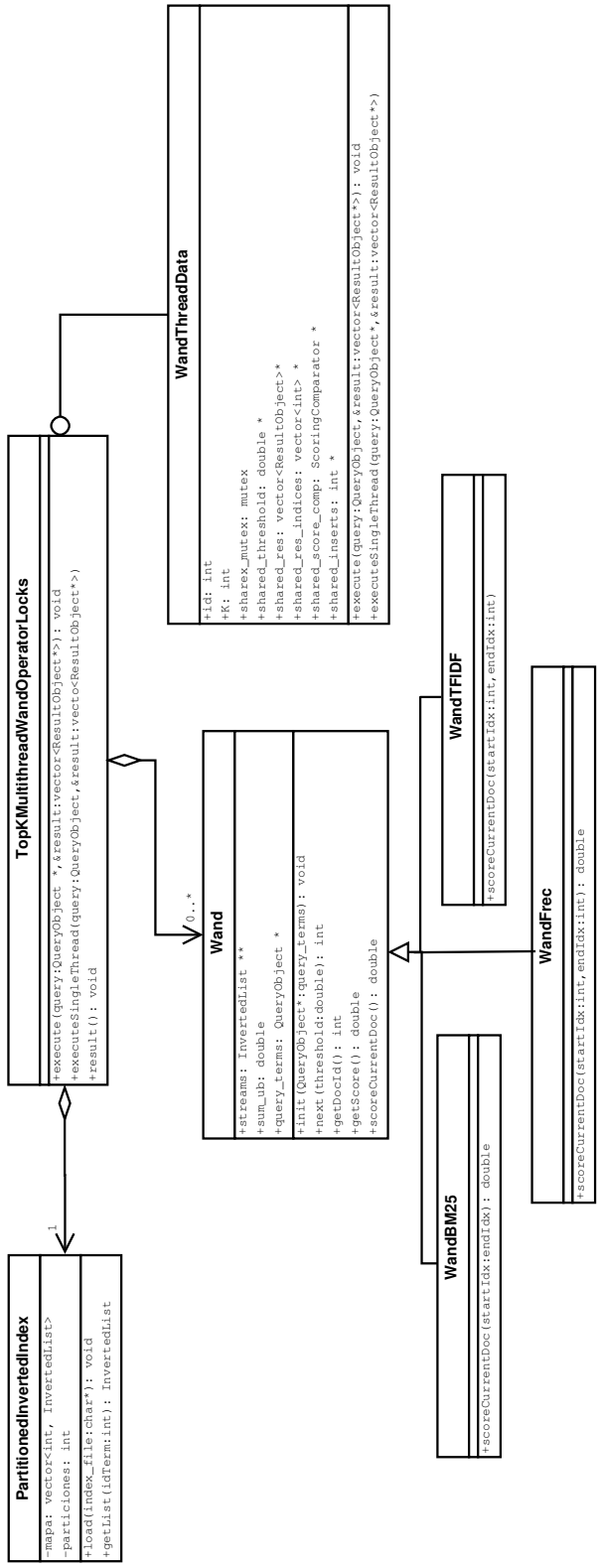

FIGURA 3.5: Diagrama de clases para el esquema SH

CAPÍTULO 4. EVALUACIÓN EXPERIMENTAL

Se hace una introducción.

4.1 PREDICCIÓN DE TIEMPO DE RESPUESTA A TRANSACCIÓN DE LECTURA

Se hace una breve introducción al cpaítulo anterior.

Se menciona los resultados obtenidos con la regresión, se dice que no se tuvieron muy buenos resultados con la regresión, se deja a ver por qué no se obtuvieron muy buenos resultados (índice con los que se hicieron experimentos, consultas, etc.).

4.1.1 Predictor perfecto

Decir que no es el foco de esta tesi, que los resultados obtenidos no fueron muy buenos s y que para evaluar los algoritmos de scheduling también se usará un predictor perfecto. Decir cómo se obtuvo un predictor perfecto y ojalá mostrar algún algoritmo.

4.2 WAND MULTITHREADED

Introducción al capítulo anterior, dos esquemas, etc. Decir cómo se llevaron a cabo los experimentos, datos que se utilizaron (o quizás decirlos una sola vez).

4.2.1 Wand heap compartido

Se habla un poco de las ventajas que se tenía con este esquema nuevamente, qué se hizo para la implementación, cómo se programó, etc.

Se muestra el código y ojalá se muestra algún flujo de ejecución para una query específica.

```
class TopKMultithreadWandOperatorLocal {
    private:
        unsigned int k; // Variable que controla el tama o del heap
        unsigned int max_particiones; // M ximo de particiones del  ndice  invertido (se usa 16)
        PartitionedInvertedIndex *indice; // Indice invertido particionado

    // 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();
            }
    };

    Comparador *comp; // Objeto Comparador

    //parts x terms > 256
    // Variables que se utilizan como buffer de las listas invertidas
    const static unsigned int n_buffers = 1024;
    const static unsigned int buffer_size = 256;
    unsigned int **buffers;

    public:
        double milis_init; //tiempo preparacion de threads antes de procesar la query
        double milis_threads; //tiempo de los threads en procesar la query (desde create hasta join)
```

```
30          double milis_merge; //tiempo que se demora en el merge de documento
31          double milis_inner; //tiempo interno del thread mas largo (en el proceso de
                paralelizaci n)
32
33          unsigned int inserts; // Variable que se utiliza para contar las inserciones de
                documentos al heap
34          double initial_threshold; // Valor del threshold inicial
35
36          static double *milis_each_thread; // Variable que se utiliza para guardar los tiempos
                de cu nto se demor  cada thread
37
38          pthread_t t[max_threads]; // Arreglo de pthread_t para los hilos de ejecuci n
39          int pids[max_threads]; // Arreglo donde se guardar n los identificadores de los
                arreglos
40          unsigned int *indices;
41
42          static TopKWandOperator **arr_ops; // Arreglo de operadores. Cada uno de los threads
                tendr  un operador (1thread)
43          static vector<ResultObject*> **arr_results; // vectores de resultados, por thread
44          static map<unsigned int, float> **mapas_ubs; // Variable que mapea cada t rmino a su
                upper_bound
45          static QueryObject ***query_terms; //Este es un arreglo de punteros a cada query
46
47          static unsigned int partes; // Partes en la que se dividir  cada query
48
49          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);
50          ~TopKMultithreadWandOperatorLocal();
51
52          // M todo que se encargar  de resolver la query. Los resultados quedar n en la
                variable result
53          virtual void execute(QueryObject *query, vector<ResultObject*> &result);
54  };
```

Se muestra un gráfico y tabla de eficiencia.

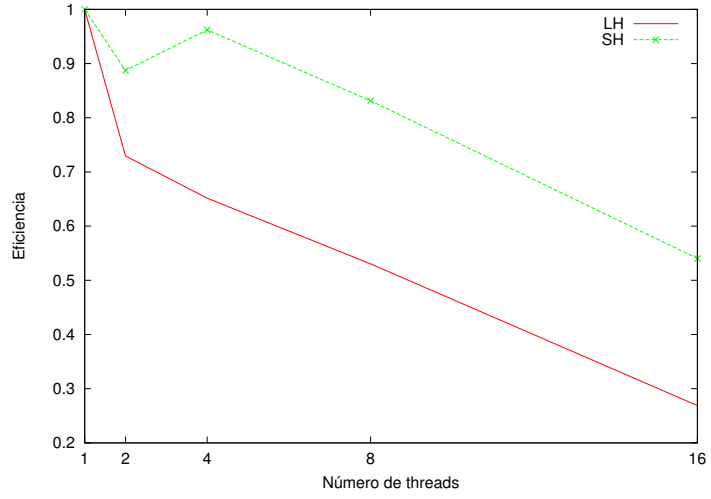

FIGURA 4.1: Eficiencias para Wand con heaps compartido y locales

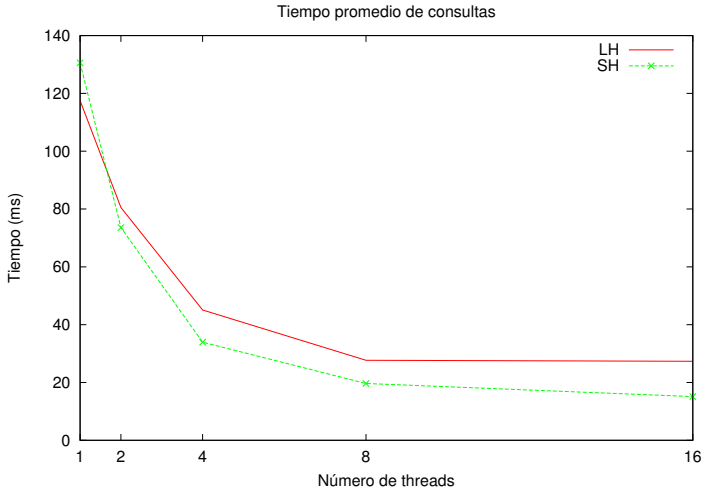

FIGURA 4.2: Tiempos promedios de las consultas

4.2.2 Wand heap local

Se habla un poco de las ventajas que se tenía con este esquema nuevamente, qué se hizo para la implementación, cómo se programó, etc.

Se muestra el código y ojalá se muestra algún flujo de ejecución para una query específica.

Se muestra un gráfico y tabla de eficiencia.

Meter algo para comparar ambos enfoques en un mismo gráfico o algo así. (no se si sea buena idea tener dos gráficos de eficiencia o solo 1).

4.3 ESTRATEGIAS DE SCHEDULING

Hablar separado cada una de ellas, mostrando implementación y cómo se llevaron a cabo los experimentos.

1. Comparar las tres estrategias de scheduling (decir que TimesRanges es mejor) ==¿Con predictor perfecto también? 2. Comparar TimesRanges con baseline ==¿Con Predictor perfecto también. 4. Decir los problemas que existen en cada una de las estrategias que se pierden tiempos. 3. Sacar a la luz la nueva unidades de trabajo ==¿Predictor perfecto también. 4. Comparación unidades de trabajo - baseline - TimesRanges.

Conclusiones para cada uno de los gráficos realizados.

CAPÍTULO 5. CONCLUSIONES

REFERENCIAS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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., Kim, S., Hwang, S.-w., He, Y., Elnikety, S., Cox, A. L., & Rixner, S. (2014). Predictive parallelization: Taming tail latencies in web search. En *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR

'14, (pág. 253–262). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/2600428.2609572

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

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

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

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

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

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

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`

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`