

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

SANTIAGO DE CHILE  
2013

© Danilo Fernando Bustos Pérez

Se autoriza la reproducción parcial o total de esta obra, con fines académicos, por cualquier forma, medio o procedimiento, siempre y cuando se incluya la cita bibliográfica del documento.

# AGRADECIMIENTOS



*Dedicado a ... .*



# RESUMEN

Resumen en Castellano

**Palabras Claves:** keyword1, keyword2 .

# ABSTRACT

Resumen en Inglés

**Keywords:** keyword1, keyword2 .



# ÍNDICE DE CONTENIDOS

Índice de Figuras	v
-------------------	---

Índice de Tablas	vii
------------------	-----

<b>1. Introducción</b>	<b>1</b>
------------------------	----------

1.1. Antecedentes y motivación . . . . .	2
--	---

1.2. Descripción del problema . . . . .	2
---	---

1.3. Objetivos y solución propuesta . . . . .	2
---	---

1.3.1. Objetivo General . . . . .	2
-----------------------------------	---

1.3.2. Objetivos Específicos . . . . .	2
--	---

1.3.3. Alcances . . . . .	2
---------------------------	---

1.3.4. Solución propuesta . . . . .	2
-------------------------------------	---

1.3.5. Características de la solución . . . . .	2
---	---

1.3.6. Propósito de la solución . . . . .	2
---	---

1.4. Metodología y herramientas de desarrollo . . . . .	2
---	---

1.4.1. Metodología . . . . .	2
------------------------------	---

1.4.2. Herramientas de desarrollo . . . . .	2
---	---

1.5. Resultados obtenidos . . . . .	2
-------------------------------------	---

1.6. Organización del documento . . . . .	2
---	---

<b>2. Marco teórico</b>	<b>3</b>
-------------------------	----------

2.1. Motores de búsqueda verticales . . . . .	3
---	---

2.2. Índice invertido . . . . .	5
---------------------------------	---

2.3. Estrategias de evaluación de transacciones de lectura . . . . .	6
--	---

<i>ÍNDICE DE CONTENIDOS</i>	ii
2.3.1. Term at a time . . . . .	7
2.3.2. Document at a time . . . . .	7
2.3.3. Consideraciones . . . . .	8
2.4. Funciones de <i>Ranking</i> . . . . .	8
2.4.1. TF-IDF . . . . .	9
2.4.2. BM25 . . . . .	10
2.5. Operaciones sobre listas invertidas . . . . .	11
2.5.1. <i>OR</i> . . . . .	11
2.5.2. <i>AND</i> . . . . .	12
2.5.3. <i>WAND</i> . . . . .	12
2.6. Scheduling en motores de búsqueda . . . . .	16
2.6.1. Trabajo relacionado . . . . .	19
<b>3. Estrategias de planificación de queries</b>	<b>21</b>
3.1. Predicción de rendimiendo de transacciones de lectura . . . . .	21
3.2. Wand <i>multi-threaded</i> . . . . .	24
3.2.1. Wand con heaps locales . . . . .	25
3.2.2. Wand con heap compartido . . . . .	28
3.3. Estrategia <i>baseline</i> . . . . .	30
3.4. Estrategias de <i>scheuling</i> . . . . .	31
3.4.1. FR . . . . .	31
3.4.2. Times . . . . .	31
3.4.3. TimesRanges . . . . .	31
3.5. Estrategia de unidades de trabajo . . . . .	31
<b>4. Evaluación experimental</b>	<b>33</b>
4.1. Predicción de tiempo de respuesta a transacción de lectura . . . . .	33

---

4.1.1. Predictor perfecto . . . . .	33
4.2. Wand multithreaded . . . . .	34
4.2.1. Wand heap compartido . . . . .	34
4.2.2. Wand heap local . . . . .	35
4.3. Estrategias de scheduling . . . . .	36
<b>5. Conclusiones</b>	<b>37</b>
<b>Referencias</b>	<b>39</b>



# ÍNDICE DE FIGURAS

2.1. Arquitectura típica de un motor de búsqueda . . . . .	4
2.2. Índice invertido . . . . .	6
2.3. Operación AND . . . . .	9
2.4. Operación OR . . . . .	11
2.5. Operación AND . . . . .	12
2.6. Ejemplo de ejecución de algoritmo <i>Wand</i> . . . . .	15
2.7. Arquitectura de un sistema de recuperación de la información con réplicas . . .	18
3.1. Esquema de ejecución de algoritmo WAND con heaps locales . . . . .	25
3.2. Diagrama de clases para el esquema LH . . . . .	27
3.3. Esquema de ejecución de algoritmo WAND con heap compartido . . . . .	28
3.4. Diagrama de clases para el esquema SH . . . . .	29



# 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 construido 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

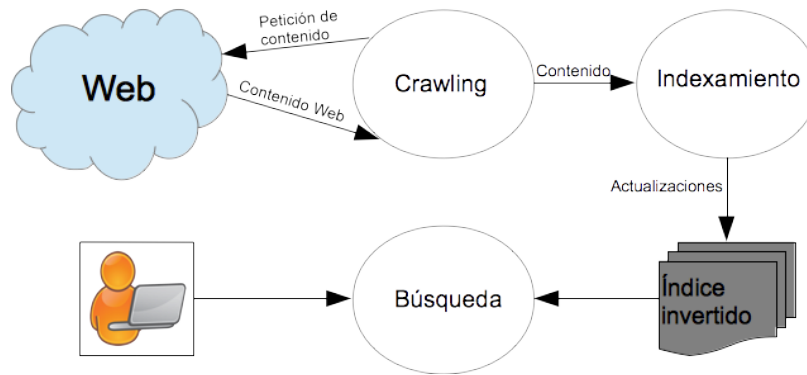


FIGURA 2.1: Arquitectura típica de un motor de búsqueda

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 volumen 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).

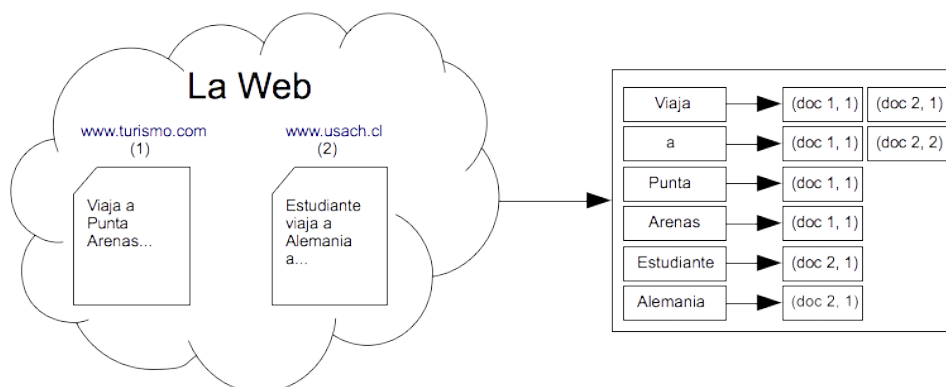


FIGURA 2.2: Índice invertido

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 uniéndolos 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 *query* completa, se procede al procesamiento del documento  $d_{j+1}$ .

### 2.3.3 Consideraciones

Cuando se tiene un índice invertido pequeño, las estrategias TAAT rinden adecuadamente, sin embargo cuando los índices invertidos son de gran tamaño las estrategias DAAT poseen 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. Además existen técnicas para optimizar el proceso de las estrategias recientemente descritas (Turtle & Flood, 1995).

## 2.4 FUNCIONES DE *RANKING*

Los sistemas de recuperación de información como los motores de búsqueda deben ejecutar un proceso el cual asigna un puntaje a documentos con respecto a una determinada *query*, 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 *queries* y documentos, y asigna un puntaje (*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.



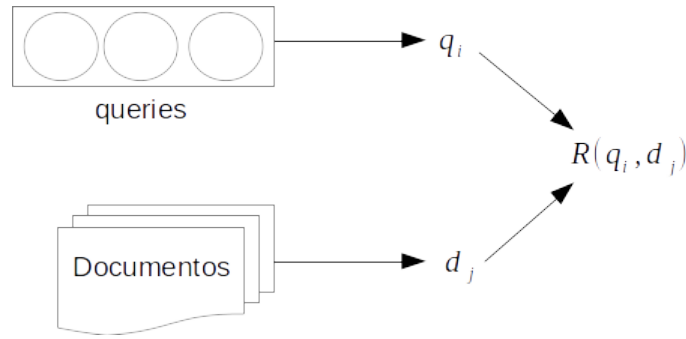


FIGURA 2.3: Operación AND

### 2.4.1 TF-IDF

El 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 de la palabra o término en un documento (tf) y que en su versión más sencilla se utiliza la frecuencia bruta del término  $t$  en el documento  $d$  ( $f(t,d)$ ). 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 obtiene calculando el logaritmo de la división entre el número total de documentos del corpus y el número de documentos que contienen el término. De esta forma se tiene:

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

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

$$tfidf(t, d, D) = tf(t, d) * idf(t, D)$$

Por lo que el estadístico *TF-IDF* 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. Notar que la 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 *query* 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  $q_1, \dots, q_n$ , el *ranking* *BM25* del documento  $D$  se calcula como:

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

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  $q_i$ .

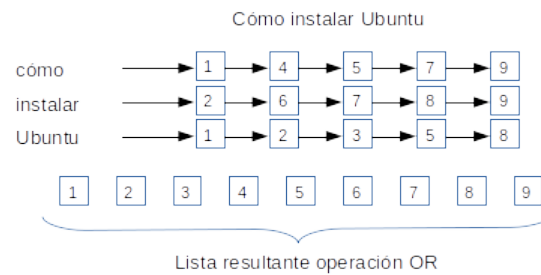


FIGURA 2.4: Operación OR

## 2.5 OPERACIONES SOBRE LISTAS INVERTIDAS

Cuando una *query* 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 con el conjunto de los K mejores. A continuación se presentan las diferentes formas de operar las listas invertidas de una *query*.

### 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.

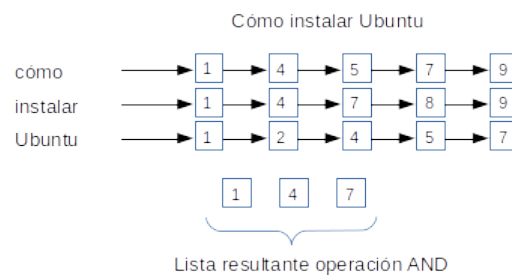


FIGURA 2.5: Operación AND

### 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).

### 2.5.3 WAND

Algoritmo de evaluación de transacciones de lectura para obtener eficientemente el conjunto de  $K$  documentos que mejor satisfacen una *query* 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

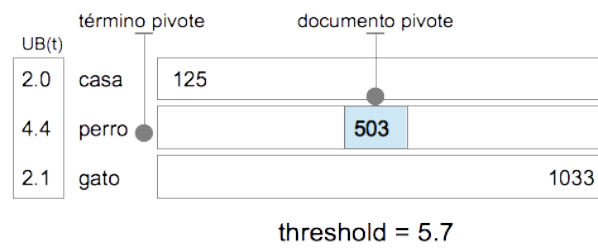
dos niveles. En el primer nivel se intenta omitir rápidamente grandes porciones de las listas invertida, lo que se traduce en ignorar el cálculo del puntaje exacto de grandes cantidades de documentos, esto porque en motores de búsqueda a gran escala, este es un proceso que requiere de mucho tiempo para llevarse a cabo y depende de factores como la cantidad de ocurrencia del término dentro del documento, el tamaño del documento, entre otros.

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*

## 2.6 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*<sup>1</sup> o *Yahoo!*<sup>2</sup> 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 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 2.7, 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).

---

<sup>1</sup><http://www.google.com>

<sup>2</sup><http://www.yahoo.com>

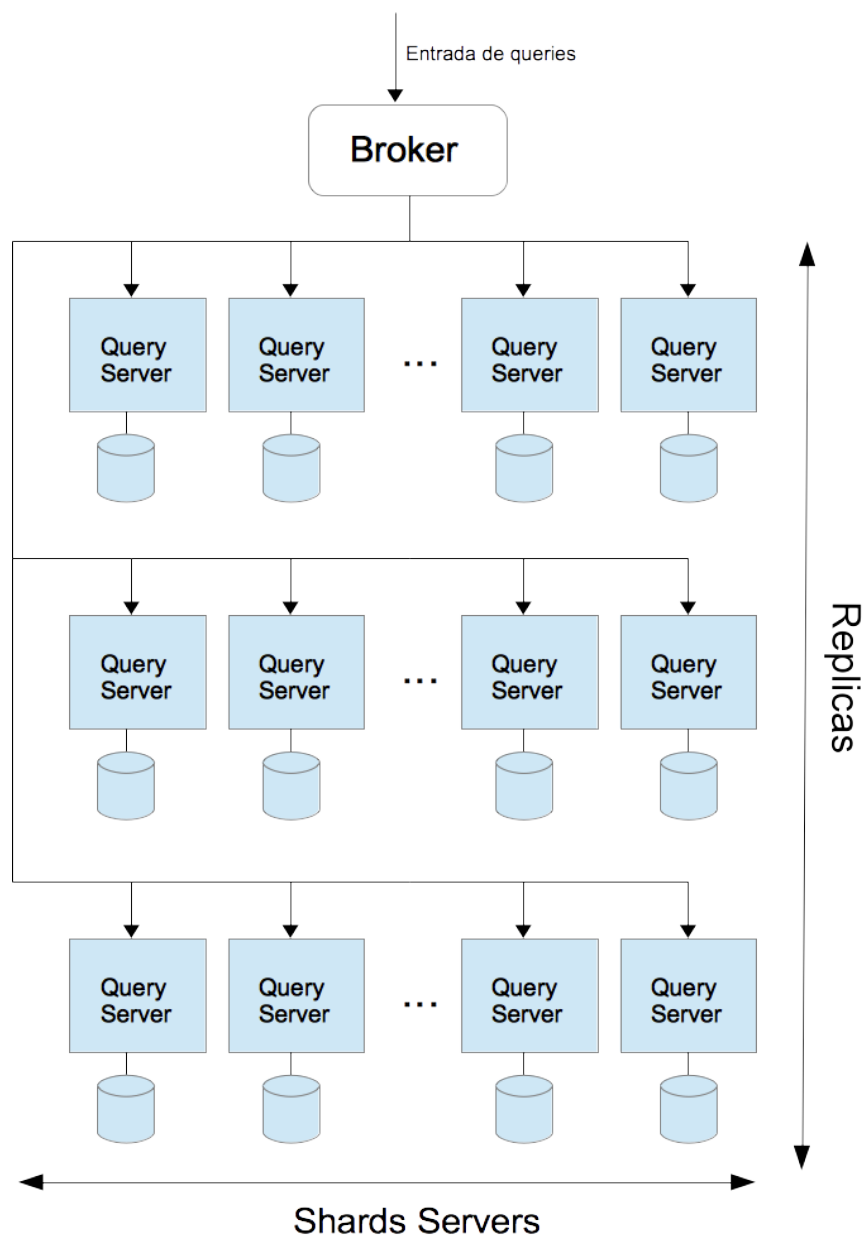


FIGURA 2.7: Arquitectura de un sistema de recuperación de la información con réplicas

### 2.6.1 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 distribuido 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  $\hat{pt}(q) \leq T - wt(q)$ , 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

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. Conocer de antemano el rendimiento es una ventaja importante para poder usar técnicas efectivas de planificación en motores de búsqueda. Por ejemplo, en el contexto de procesamiento paralelo de *queries* por lotes (*batches*) se pueden crear grupos de consultas que posean rendimientos parecidos, así se tiende a disminuir tanto el desbalance de carga entre los procesadores como el tiempo en procesar el *batch*.

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 rendimiento de *queries*. La eficiencia también ha sido utilizada para predecir el rendimiento de una consulta, 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 transacciones de lectura basado en datos estadísticos disponibles en las respectivas listas invertidas de los términos. La idea del método es que para cada consulta que llega al sistema, se tomen los términos y para cada uno de ellos calcular desde su propia lista invertida los siguientes estadísticos  $s(t)$ :

**Puntaje medio aritmético.** Se calcula la media aritmética del puntaje de los documentos.

**Puntaje medio geométrico.** Se calcula la media geométrica del puntaje de los documentos.

**Puntaje medio armónico.** Se calcula la media armónica 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 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 mayor a la media.** Se extrae el número de documentos que sobrepasa en puntaje al puntaje promedio.

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

**Ultimo estadístico.** Pregúntarselo a Victor.

Cada uno de los 14 estadísticos mencionados anteriormente está relacionados linealmente con el tiempo que se demora un motor de búsqueda en resolver una transacción de lectura y son la base para la implementación predictor que está basado en un modelo de regresión lineal (Chambers, 1991). En el siguiente capítulo se muestra el detalle sobre la implementación del modelo predictor y los resultados obtenidos.

### 3.2 WAND *MULTI-THREADED*

Dado que el método WAND (Broder et al., 2003) consiste en el método del estado del arte ocupado hoy en día por los motores de búsqueda, 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. Para hacer que este proceso trabaje eficientemente, se usa una estructura de datos llamada *heap* que va guardando el conjunto de los mejores K documentos hasta determinado instante. El menor puntaje de este conjunto es usado como umbral 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*. Uno de ellos es usando *heaps* locales (LH), es decir, un *heap* por *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) son ejecutadas 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.



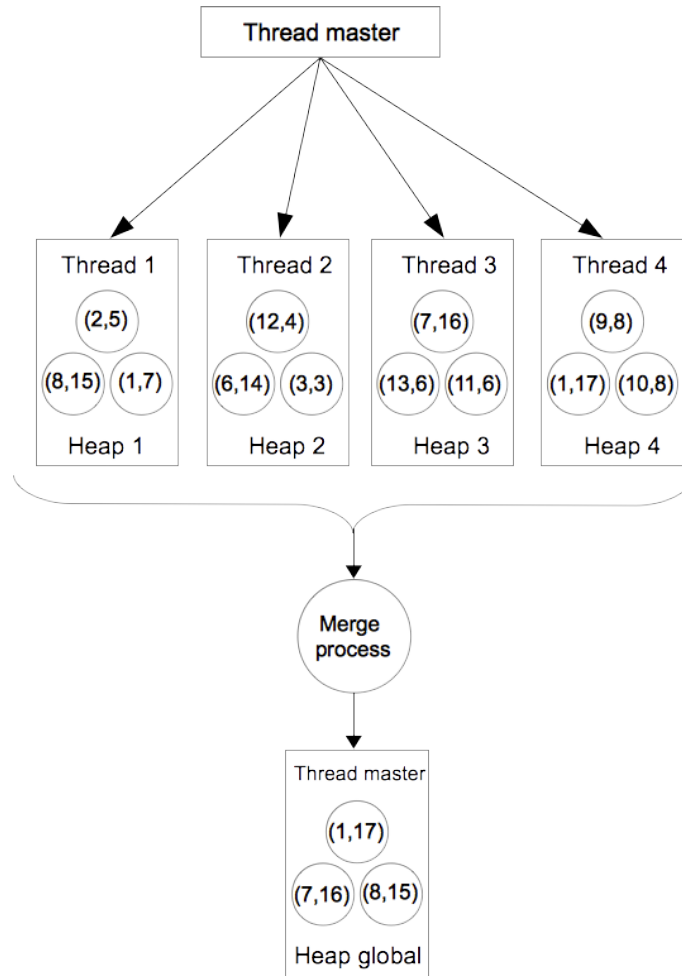


FIGURA 3.1: Esquema de ejecución de algoritmo WAND con heaps locales

### 3.2.1 Wand con heaps locales

En el esquema LH, cada thread procesa una porción del índice invertido mientras mantiene un heap local con los mejores K documentos que el específico thread ha encontrado hasta ahora. Al final del proceso, el resultado de cada thread debe ser reunido 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 3.1 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 3.2. La clase principal es la `TopKMultiThreadWandOperatorLocal`, que es la encargada de controlar el paralelismo en la resolución de las transacciones. Para explicar de mejor manera cada una de las clases involucradas en la implementación, se presenta el siguiente diccionario de datos.

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

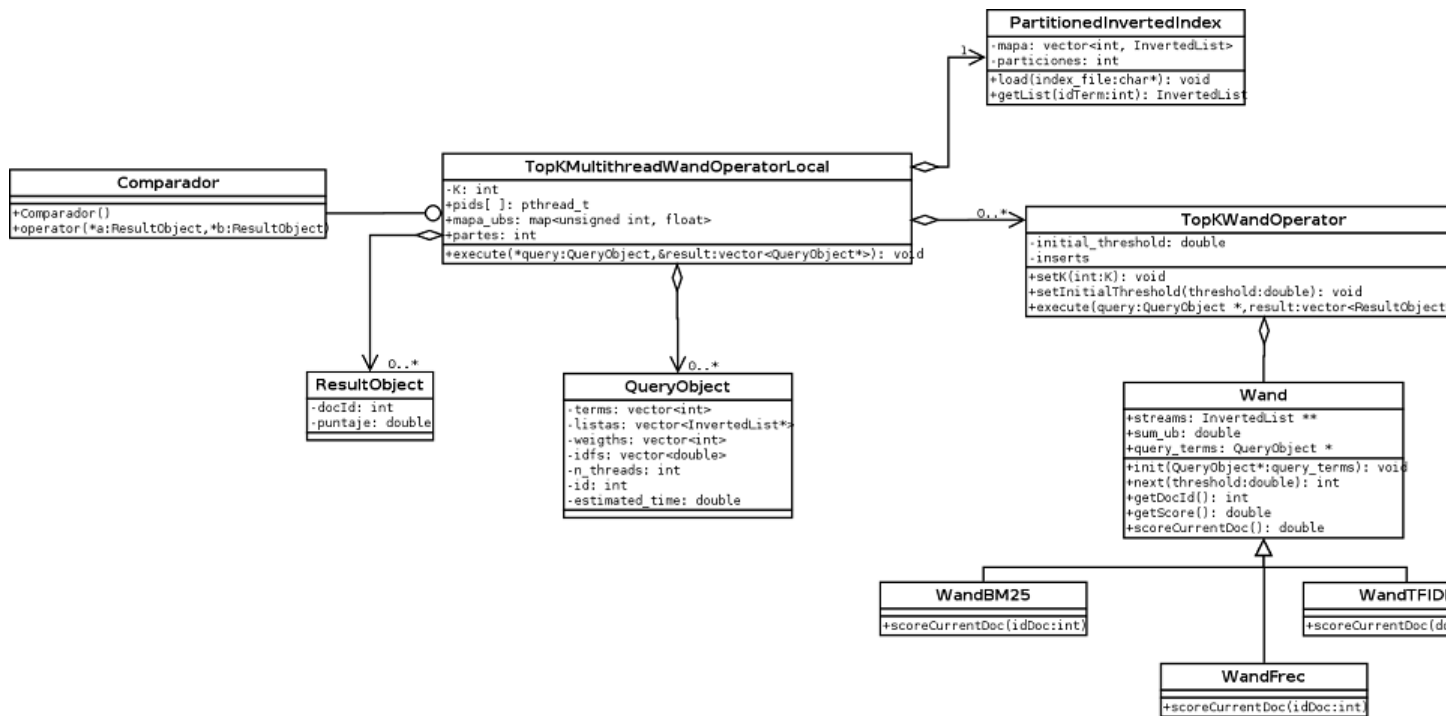


FIGURA 3.2: Diagrama de clases para el esquema LH

### 3.2.2 Wand con heap compartido

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

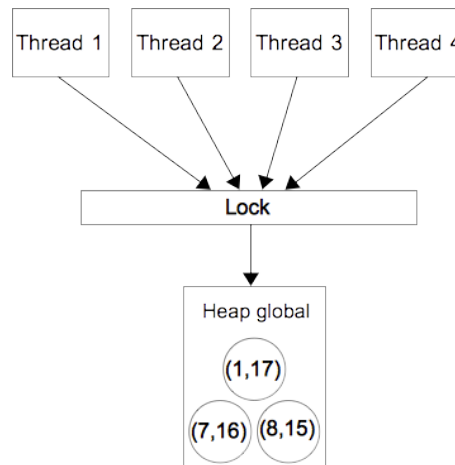


FIGURA 3.3: Esquema de ejecución de algoritmo WAND con heap compartido

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 3.4.

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

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

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

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

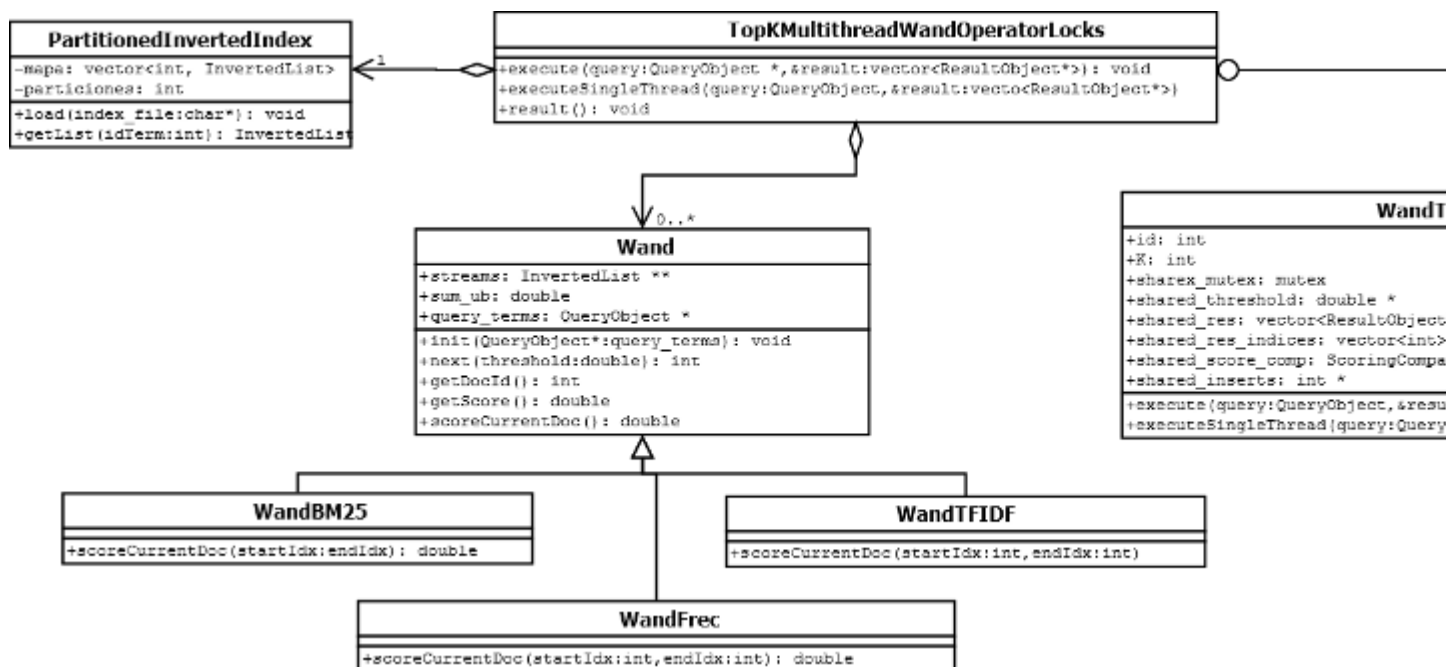


FIGURA 3.4: Diagrama de clases para el esquema SH

### 3.3 ESTRATEGIA *BASELINE*

Una simple manera de construir un sistema que responda a múltiples queries simultáneamente usando múltiples threads es, usando los threads de manera independiente. Para hacer esto, se debe mantener un conjunto de threads consumidores, cada uno de ellos se encargará de responder a queries secuencialmente y todos ellos trabajan en paralelo leyendo las queries desde la misma cola. Esto es lo que en este trabajo se denomina Un Thread Por Query (1TQ), ver Figura X (Profundizar más en esta figura)

Este esquema simple tiene ventajas y desventajas, además que es fácil de implementar y controlar.

Existen sistemas que tienen que ejecutar un conjunto de queries de un cierto tamaño y luego parar para actualizar la información del índice invertido. Solo después de la fase de actualización, éste es capaz de ejecutar el siguiente conjunto de queries (batch de queries). Al final de cada batch, es posible que algunos threads 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.

Sin embargo, aunque cada thread está secuencialmente ejecutando una query diferente, algunas de estas operaciones puede tomar un tiempo considerable, de esta forma se produce una importante pérdida de eficiencia, aunque la intuición nos dice que esto se puede amortizar con trabajos pequeños. Ver Figura 2 (Explicar)

### 3.4 ESTRATEGIAS DE *SCHEULING*

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

#### 3.4.1 FR

#### 3.4.2 Times

#### 3.4.3 TimesRanges

### 3.5 ESTRATEGIA DE UNIDADES DE TRABAJO





# 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 capítulo 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.

```

1 class TopKMultithreadWandOperatorLocal {
2     private:
3         unsigned int k; // Variable que controla el tamaño del heap
4         unsigned int max_particiones; // Máximo de particiones del índice invertido (se usa
5             16)
6         PartitionedInvertedIndex *indice; // Índice invertido particionado
7
8         // Clase anidada que se utiliza para ordenar los documentos dentro del heap
9         class Comparador : public std::binary_function<const ResultObject*, const ResultObject*,
10             bool> {
11             public:
12                 Comparador(){ }
13                 inline bool operator()(const ResultObject *a, const ResultObject *b){
14                     if (a->getScore() == b->getScore()){
15                         return a->getDocId() > b->getDocId();
16                     }
17                     return a->getScore() > b->getScore();
18                 }
19             };
20
21         Comparador *comp; // Objeto Comparador
22
23         //parts x terms > 256
24         // Variables que se utilizan como buffer de las listas invertidas
25         const static unsigned int n_buffers = 1024;
26         const static unsigned int buffer_size = 256;
27         unsigned int **buffers;
28
29     public:
30         double millis_init; //tiempo preparacion de threads antes de procesar la query
31         double millis_threads; //tiempo de los threads en procesar la query (desde create hasta
32             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.

#### 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](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>
- 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](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](http://dx.doi.org/10.1007/978-3-540-30213-1_5)
- 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>

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](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](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>