Capítulo 4

Transacciones Concurrentes

En este capítulo se propone realizar procesamiento sincrónico por lotes a nivel de threads como una alternativa más eficiente y escalable que el enfoque convencional basado en threads asincrónicos. Se muestra que este tipo de paralelismo es eficiente para procesar transacciones concurrentes de lectura y escritura en nodos de búsqueda implementados sobre índices invertidos. Sobre esta forma de paralelismo se diseñan algoritmos de procesamiento de consultas (transacciones de lectura) y actualización on-line del índice (transacciones de escritura) que muestran ser particularmente eficientes frente a situaciones de tráfico alto de transacciones. Las estrategias propuestas son generales y pueden ser aplicadas a cualquier tipo de motor de búsqueda.

Para un cluster con $P \times D$ nodos, donde cada nodo contiene procesadores multi-core que permiten ejecutar eficientemente T threads, y donde los nodos se utilizan para mantener P particiones del índice invertido, cada partición replicada D veces, se le llama "nodo de búsqueda" a cada nodo en que se despliega un servicio de índice que opera en modo exclusivo en el nodo. Se asume que a cada nodo llegan mensajes conteniendo transacciones de lectura y escritura que deben ser procesadas en tiempo real. Dichos mensajes son depositados en una única cola de transacciones, las cuales son retiradas por los threads para darles servicio. Para el conjunto de $P \times D$ nodos de búsqueda, se asume que el valor de P es tal que permite a un solo thread procesar completamente una transacción de manera secuencial dentro de una cota superior para el tiempo de respuesta. Se asume que el nivel de replicación D es tal que es posible alcanzar el throughput objetivo para la tasa de llegada de transacciones en régimen permanente. Lo que propone este capítulo son soluciones eficientes para enfrentar alzas bruscas en el tráfico de transacciones, especialmente alzas

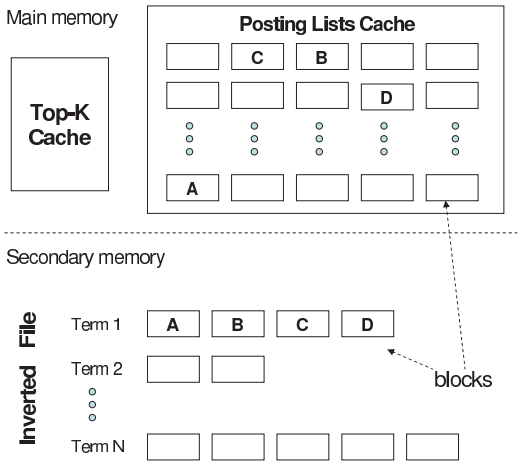

Figura 4.1: Arquitectura de un Nodo de Búsqueda.

en las transacciones de sólo lectura [13, 49, 94] las cuales representan las consultas de los usuarios del motor de búsqueda.

4.1. Planteamiento del Problema

En la literatura para el diseño de motores de búsqueda es posible encontrar numerosas estrategias de indexación y caching, las cuales pueden ser combinadas de distintas maneras para alcanzar un rendimiento eficiente y escalable a sistemas de gran tamaño y tráfico de consultas. La combinación específica depende de los requerimientos que dependen de aspectos tales como el tamaño del índice, el uso de memoria secundaria y/o compresión del índice, y uso de distintos tipos de caches. En particular, el diseño de un servicio de índice para un motor de búsqueda puede tener la forma mostrada en la Figura 4.1. El cache de listas invertidas está formado por un gran número de bloques que se utilizan para almacenar grupos de ítemes de las listas invertidas del tipo (doc_id, term_freq). Cada lista, por lo general ocupa varios de estos bloques. Un cache secundario almacena los resultados top-K de las consultas más frecuentes resueltas por el nodo de búsqueda. La lista invertida completa de cada término se puede mantener en el disco local al nodo o en memoria principal en formato comprimido, es decir, se trata de una zona de memoria relativamente más lenta que la memoria proporcionada por los caches.

Un camino factible para los threads se ilustra en la Figura 4.2. En este ejemplo, las consultas llegan a la cola de entrada del nodo de búsqueda. Un número determinado de

threads se encarga de resolver las consultas. Cada vez que un thread obtiene una nueva consulta desde la cola de entrada, se comprueba si la misma ya está almacenada en el cache de top-K (1). Si hay un hit en este cache, el thread responde con los identificadores de los K documentos almacenados en la entrada correspondiente (2). De lo contrario, el thread verifica si todos los bloques de las listas invertidas de los términos de la consulta están en el cache de listas (3). Si es así, el thread utiliza esos bloques para resolver la consulta mediante la aplicación de un algoritmo de ranking de documentos (3.a). Una vez que el thread obtiene los identificadores de los documentos top-K para la consulta, se guarda esta información en el cache de top-K aplicando una política de reemplazo de entradas del cache (4) y se envía un mensaje con la respuesta a la consulta (5). Si faltan bloques de listas invertidas en el cache de listas (3.b) el thread deja la consulta en una cola de requerimientos de memoria secundaria o índice comprimido (otro thread gestiona esta transferencia de bloques) y comprueba si en esta segunda cola hay otra consulta cuya transferencia de bloques al cache de listas ha sido completada para proceder con su solución (3.c), (4) y (5).

El enfoque anterior de procesamiento de consultas utilizando varios threads, supone la existencia de una estrategia capaz de abordar adecuadamente la ejecución concurrente de las operaciones de lectura/escritura causadas por los threads en los caches y las colas. Dado que las entradas de los caches son reemplazadas por nuevos datos constantemente, es muy posible encontrar lecturas y escrituras concurrentes siendo ejecutadas en la misma entrada de un cache, y por lo tanto es necesario imponer un protocolo que permita sincronizar los accesos de los threads a los recursos compartidos para prevenir conflictos de lectura/escritura. Una solución intuitiva es aplicar locks a las entradas del cache. Sin embargo, el problema es más complicado que eso como se describe a continuación.

Luego de calcular los identificadores de los documentos top-K para una consulta, es necesario desalojar una entrada del cache de top-K para almacenar los nuevos identificadores top-K. Durante el mismo intervalo de tiempo, otros threads pueden estar leyendo las entradas del cache para decidir si calcular o no los resultados de sus respectivas consultas. Si el control de concurrencia se realiza a nivel grueso utilizando un lock o unos pocos locks para proteger todas las entradas del cache, los threads pueden ser serializados severamente. Por otro lado, mantener un lock por cada entrada del cache incrementa la concurrencia significativamente. Sin embargo, hacer que cada thread ejecute un lock por cada entrada que lee mientras recorre el cache secuencialmente, puede también producir un efecto de serialización de threads y dado que el costo de cada lock no es despreciable, el tiempo de ejecución se puede degradar y aumentar con el número de threads. Como se ha

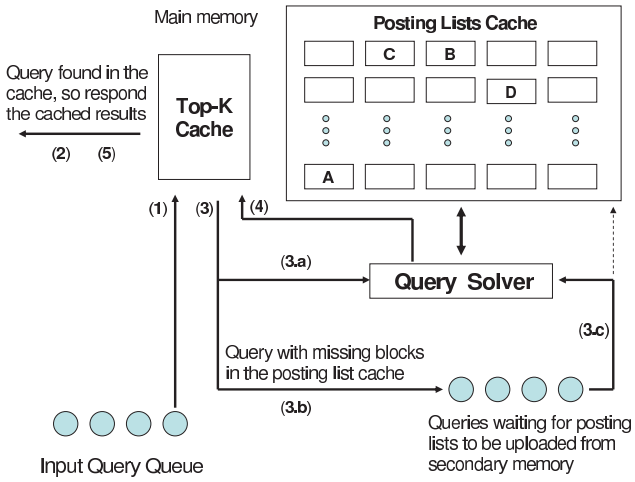

Figura 4.2: Camino seguido por las consultas.

señalado en [27], este problema puede aliviarse utilizando estrategias como SDC donde un 80 % de las entradas son de sólo lectura (cache estático) y un 20 % son de lectura/escritura (cache dinámico).

En el cache de listas invertidas, es necesario seleccionar rápidamente un número suficientemente grande de bloques de cache para reemplazarlos por los bloques que contienen los pares (doc_id, term_freq) de la nueva lista invertida siendo recuperada desde memoria secundaria o desde un índice comprimido. Para este propósito se puede utilizar una cola de prioridad la cual permite determinar de manera exacta y eficientemente los bloques de mayor prioridad de desalojo en el cache. Alternativamente, es posible utilizar una lista enlazada administrada con la heurística "mover al frente", la cual permite obtener de manera aproximada los bloques con mayor prioridad de ser desalojados del cache. Para la política LRU, la estrategia de mover al frente debería funcionar relativamente bien, mientras que para otras políticas de reemplazo tales como Landlord [30], la cola de prioridad es la alternativa de mejor rendimiento.

Si los bloques que contienen los ítemes de las listas invertidas son modificados por threads escritores, el problema de concurrencia se agrava, con la dificultad adicional que debe haber consistencia entre los bloques almacenados en las entradas del cache y las respectivas listas invertidas mantenidas en memoria secundaria o en memoria principal en formato comprimido. Por ejemplo, si una estrategia de control de concurrencia se basa en el bloqueo de las listas invertidas asociadas con los términos de la consulta, entonces, esto también debería provocar el bloqueo de todas las entradas del cache que mantienen los

bloques de las respectivas listas invertidas (o alternativamente el ocultamiento de estas entradas de el algoritmo de reemplazo de entradas).

Además, mientras un thread lee o actualiza una lista invertida ℓ, el algoritmo de administración de cache no debe seleccionar para reemplazo los bloques asignados a ℓ. Al igual que en el caso anterior, una solución sencilla es ocultar temporalmente todos los bloques de ℓ antes que el thread los utilice. Este proceso también requiere de una correcta sincronización de los threads y por lo tanto puede degradar el tiempo de ejecución del thread que está operando sobre la lista ℓ.

El modificar una lista invertida también presenta problemas de coherencia entre la nueva versión y las posibles entradas para el mismo término que ya están almacenadas en el cache de top-K. Si el nuevo ítem que se inserta en la lista invertida tiene una frecuencia lo suficientemente alta o mayor a la versión anterior, entonces es probable que el documento asociado pueda también estar presente en la respectiva entrada del cache de top-K. En este caso, es más práctico invalidar todas las entradas del cache relacionadas con los términos que aparecen en las respectivas consultas, ya que la única manera de estar seguros es volver a calcular el ranking de la consulta. Esto genera requerimientos adicionales de control de concurrencia en el cache, los cuales imponen una carga adicional en uso de locks o algún otro mecanismo de sincronización de threads.

4.2. Solución Propuesta

La solución propuesta está basada en la observación de que si las consultas y actualizaciones del índice fueran procesadas de manera secuencial por un solo proceso o thread principal, no sería necesario formular una solución para cada uno de los problemas anteriores. Por lo tanto, lo que se propone es asignar un thread principal en el nodo de búsqueda a seguir secuencialmente los pasos descritos en la Figura 4.2, el cual recurre al uso de tantos threads como núcleos tenga el procesador para paralelizar las operaciones de mayor costo tales como el ranking de documentos, actualización del índice y la administración de los caches. La granularidad de las demás operaciones en términos de costo es muy pequeña y las puede ejecutar el mismo thread principal secuencialmente utilizando uno de los núcleos del nodo sin pérdida de eficiencia tal como fue comprobado en las estrategias presentadas en el Capítulo 3. Se utiliza paralelismo sincrónico para organizar de manera eficiente las operaciones realizadas por los threads. El número N_t de threads disponibles para ser utilizados por el thread principal puede variar dinámicamente en todo momento

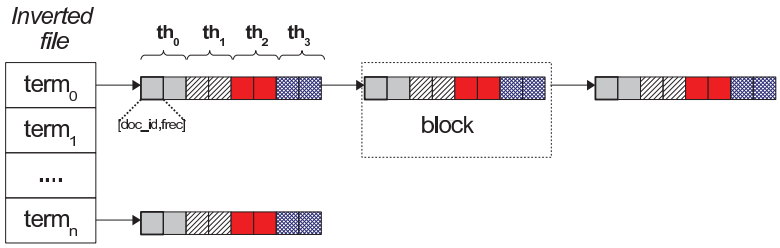

Figura 4.3: Organización del índice invertido donde cada lista invertida es almacenada en un número de bloques y cada bloque se encuentra lógicamente dividido en trozos que son asignados a cada threads. Cada trozo está compuesto por un número de ítemes de la lista invertida, (doc_id, freq). En este ejemplo, el primer bloque el término *term_0* es procesado en paralelo por los threads *th0*, *th1*, *th2* and *th3*.

con el objetivo de permitir la ejecución de software de mantención y administración en el nodo de búsqueda durante su operación en producción.

4.2.1. Query Solver

Las consultas son resueltas en el componente llamado *query solver*, el cual es el encargado de recorrer las listas invertidas de todos los términos de la consulta y aplicar el método de ranking de documentos. Los elementos de cada lista invertida se agrupan en bloques consecuentes con el tamaño de las entradas del cache de listas invertidas. Se utilizan N_t threads para hacer el ranking de los documentos y cada thread trabaja sobre una sección distinta de cada lista invertida, emulando un índice particionado por documentos. Por ejemplo, la Figura 4.3 (organización de la estructura de datos utilizada) muestra esta situación. La figura muestra que para el primer bloque del término *term_0*, existen $N_t = 4$ threads trabajando en paralelo en el mismo bloque.

Dado que el rendimiento de los procesadores multi-core es altamente dependiente de la localidad de datos, durante el procesamiento de una consulta todos los threads deberían maximizar el acceso a memoria local, es decir, accesos a datos almacenados en el cache privado de los respectivos núcleos o CPUs donde los threads están siendo ejecutados. Con este fin, cada thread mantiene una estructura de datos llamada *fast_track* destinada a aumentar la localidad de referencias y por lo tanto las consultas se procesan iterativamente siguiendo dos pasos principales:

- **Fetching**. El primer paso consiste en leer desde la memoria principal del nodo (cache de listas) un trozo de lista invertida de tamaño K por cada término presente en la consulta, y por cada uno de ellos almacenar un trozo distinto de tamaño K/N_t en la memoria *fast_track* de cada uno de los N_t threads que participan en la solución de la consulta. En rigor, tan pronto como un segmento de tamaño K ha sido almacenado en las N_t memorias locales, él o los bloques respectivos del cache pueden quedar disponibles para operaciones de actualización dirigidas a esa sección del índice o pueden ser desalojadas del cache.

- **Ranking**. En el segundo paso, los N_t threads ejecutan en paralelo el ranking de documentos y, si es necesario, se solicitan nuevos trozos de tamaño K de las listas invertidas (fetching) para finalmente producir los K documentos mejor rankeados para la consulta. El proceso de ranking puede implicar la determinación de los documentos que contienen todos los términos de la consulta, lo que implica realizar una operación de intersección entre las listas invertidas involucradas.

De esta manera el proceso de ranking puede requerir varias iteraciones de la secuencia (*Fetch*, *Rank*) en ser completado. Los documentos son asignados a los threads utilizando la regla *id_doc* módulo N_t, es decir, partición por documentos, lo cual aumenta la localidad de operaciones costosas tales como la intersección de listas invertidas. Para facilitar el uso de memoria contigua en las transferencias, los bloques que mantienen los pares (doc_id, term_freq) de las listas invertidas se pueden mantener explícitamente particionados en N_t sub-bloques, donde N_t indica el total de threads a ser utilizados en régimen permanente, o implícitamente divididos en N_t particiones almacenando todos los pares (doc_id, term_freq) asignados a una partición dada en una región contigua del bloque.

Por otro lado, en la cola de entrada del nodo de búsqueda también pueden haber operaciones o transacciones de escritura que representen el caso en que nuevos documentos son insertados en el índice (inserción), o que documentos existentes sean reemplazados por una nueva versión (actualización). La Figura 4.4 ilustra el caso en que un nuevo documento es insertado en el índice, donde se muestra el efecto de la inserción en un conjunto de términos y secciones de las respectivas listas invertidas. Una operación de escritura puede modificar cualquier bloque de una lista invertida y este puede encontrarse en el cache o en memoria secundaria o memoria comprimida. Una transacción de escritura se procesa distribuyendo uniformemente los términos en los N_t threads disponibles y cada thread modifica las respectivas listas en paralelo.

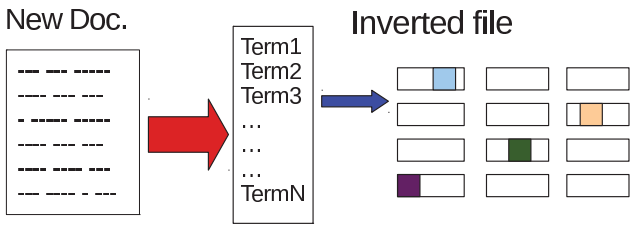

Figura 4.4: Inserción/Actualización de documentos.

Para consultas de tipo OR (*disjunctive queries*), las listas invertidas son ordenadas por frecuencia del término en los documentos, mientras que para consultas de tipo AND (*conjunctive queries*) las listas son ordenadas por identificador de documento. Esto último requiere de la intersección de todas las listas invertidas involucradas. En este caso, es útil mantener las listas ordenadas por identificador de documento ya que las operaciones de intersección pueden tomar tiempo lineal respecto de la longitud de las listas. Para apoyar la inserción eficiente en listas ordenadas por frecuencia, se mantienen espacios vacíos en los bloques y se aplica una estrategia similar a la empleada en el B-Tree. Es decir, cuando un bloque se llena, se crea uno nuevo bloque moviendo al nuevo bloque elementos tomados desde el bloque afectado y el bloque adyacente a éste.

4.2.2. Algoritmo Bulk Processing (BP)

La estrategia propuesta utiliza paralelismo sincrónico por lotes (*bulk-synchronous parallelism* [99]) para posibilitar el paralelismo a nivel de threads y donde las barreras de sincronización de threads permiten evitar los posibles conflictos de lectura y escritura entre las transacciones. Básicamente los threads son sincronizados al final de una secuencia de iteraciones realizadas para resolver una consulta o al final de la inserción o actualización de un documento en el índice invertido. Entre dos barreras de sincronización consecutivas se procesa completamente una transacción de lectura o escritura. Los detalles se presentan en la Figura 4.5, la cual presenta un caso donde el *query solver* tiene en su cola de entrada un lote de transacciones de lectura y de escritura, y las procesa todas de una vez antes de entregar los resultados.

En el algoritmo presentado en la Figura 4.5 se utiliza una forma relajada de sincronización de barrera que se ha diseñado para reducir el costo de la sincronización. Típicamente, una barrera de sincronización estándar para memoria compartida se implementa utilizando una operación "conditional wait" para hacer que los threads se bloquen en el punto

F es un trozo de tamaño $O(K)$ de memoria local al procesador.
D conjunto de documentos rankeados para una consulta.
R documentos mejor rankeados para una consulta.
L_t es la lista invertida de un término t.
tid es el identificador del thread.
pair(d, f_t) es un ítem de lista invertida, donde f_t es la frecuencia del término t en el documento d.

```
BulkProcessing( tid )

  for each transaction Tn in input queue do
    if Tn.type == QUERY then /* read transaction */
      empty( D )
      for each term t in Tn.query do
        for each block b in L_t[tid] do
          F = fetch( L_t[tid][b] )
          rank( F, D )
        endfor
      endfor
      R_tid = getLocalTopK( D )
    else /* write transaction */
      for each term t in Tn.termsForThread[tid] do
        d = Tn.docId
        if Tn.type == UPDATE then
          update( L_t[tid], pair(d,f_t) )
        else
          insert( L_t[tid], pair(d,f_t) )
        endif
      endfor
    endif
    ObliviousBarrier( tid, Tn.id mod N_t )
    if (Tn.type == QUERY) and (Tn.id mod N_t == tid) then
      R_global = getGlobalTopK( { R_1, R_2, ..., R_Nt } )
      ouputQueue[ tid ].store( R_global )
    endif
  endfor

EndBulkProcessing
```

Figura 4.5: Pasos ejecutados por cada thread en paralelo con los otros $N_t - 1$ threads.

de sincronización. El último threads despierta a todos los otros para lo cual también se utiliza un contador protegido mediante un lock de acceso exclusivo. Cuando el valor del contador es igual al total de threads, se indica el fin de la barrera y los $N_t - 1$ threads bloqueados se activan. Para un sistema con gran número de threads, la competencia por acceder al contador y a la instrucción de espera condicional puede degradar el rendimiento al transformarse en una fuente de serialización de threads.

Basados en la idea de sincronización relajada (*oblivious synchronization*) para el modelo BSP sobre memoria distribuida [12], en la Figura 4.6 se propone una versión del mismo concepto para memoria compartida, la cual usa locks y esperas condicionales. Básicamente la idea consiste en ampliar a N_t el total de locks y esperas de manera de reducir la competencia de los threads por acceder a esos recursos compartidos. Las operaciones *condWait*(a, b) y *condSignal*(a) tienen la misma semántica que sus homólogos en Posix-threads, es decir, se bloquea b y se duerme el thread con una variable de condición a y posteriormente se despierta a los threads que esperan en la variable a.

Aparte de la mejora en rendimiento proveniente de la reducción de competencia por recursos, el costo de los locks y esperas condicionales es amortizado por la vía del siguiente concepto que el algoritmo BP explota para lotes de N_t o más consultas. La última parte del procesamiento de las transacciones de lectura contiene una parte secuencial en que se determinan los mejores K documentos para la consulta. Una ventaja de la sincronización relajada es que permite que esa fase sea realizada para N_t consultas en paralelo. Lo mismo es posible con la sincronización de barrera estándar, pero la diferencia aquí es que cada threads puede ingresar a esa fase tan pronto como los otros threads han terminado de calcular sus mejores documentos locales para la consulta. Este solapamiento tiene el efecto de amortizar los costos de la sincronización.

4.2.3. Alzas bruscas en el tráfico de consultas

Un nodo de búsqueda debe asegurar que el tiempo de respuesta de consultas individuales no esté por encima de un determinado límite superior, incluso en una situación de tráfico alto de consultas. En este escenario, es conveniente retrasar las operaciones de escritura hasta un periodo de tráfico moderado o bajo.

La estrategia BP es lo suficientemente flexible enfrentar para estos casos puesto que puede acomodar varias heurísticas orientadas a absorber periodos de saturación de tráfico. Por ejemplo, el *query solver* puede imponer un límite superior para el número de bloques

```
ObliviousBarrier( my_tid, tid )

  set_Lock( mutex[tid] )
    counter[tid]++
    if my_tid == tid then
      if counter[tid] < N_t then
        condWait( cond[tid], mutex[tid] )
      else
        counter[tid]=0
      endif
    else if counter[tid] == N_t then
      counter[tid]=0
      condSignal( cond[tid] )
    endif

  unset_Lock( mutex[tid] )
EndObliviousBarrier
```

Figura 4.6: Sincronización relajada ejecutada por cada thread con identificador *my_tid*.

que son recorridos durante el procesamiento de cada consulta y puede mantener cola round-robin de consultas en ejecución. El límite superior se puede ajustar en función del número de consultas siendo procesadas en el *query solver*. Esto es importante ya que la longitud de las listas invertidas siguen la ley de Zipf, donde un número reducido de términos tienen listas muy grandes mientras que los restantes son tamaños relativamente pequeños. Así, un nodo de búsqueda debe garantizar una cota superior para el tiempo de respuesta de consultas individuales, llegando incluso a producir respuestas aproximadas para las consultas con listas con un número muy grande de bloques, mientras que debido a la aplicación de round-robin puede resolver de manera exacta las consultas con listas invertidas más pequeñas.

4.3. Un Caso Especial: BP global

La estrategia BP descrita en la sección anterior apoya el enfoque de que cada transacción es procesada con la ayuda de todos los N_t threads. La idea es procesar una o más transacciones en el recorrido secuencial del thread principal por los distintos componentes ejemplificados en la Figura 4.2. Esto supone la existencia de una cantidad suficiente de paralelismo en las transacciones de manera que sea posible utilizar eficientemente los N_t

threads en cada transacción. No obstante, tal como se muestra en el Capítulo 3, Figura 3.5, cuando el nodo de procesadores multi-core puede utilizar eficientemente una cantidad N_t grande de threads, es posible también de manera eficiente dividir los threads en grupos y aplicar paralelismo en cada grupo. En el caso de BP, es fácilmente posible aplicar paralelismo sincrónico utilizando para cada transacción una cantidad de N_t/n threads de manera de mantener n instancias de BP tomando transacciones desde la cola de entrada de transacciones. Siempre el objetivo es resolver la mayor cantidad de transacciones posibles en cada visita que el thread principal hace al *query solver* de la Figura 4.2.

Por otra parte, existen servicios de índice, especialmente en motores de búsqueda vertical (e.g., publicidad on-line), que están diseñados para trabajar con todo el índice descomprimido almacenado en memoria principal. Es decir, no utilizan ningún tipo de cache y resuelven completamente cada consulta que reciben sin importar si son repetidas. Por lo general el índice es pequeño y no posee una cantidad de paralelismo lo suficientemente grande como en el caso supuesto para la estrategia BP. Notar que BP utiliza indexación particionada por documentos. No obstante, para el caso descrito en esta sección, es perfectamente posible utilizar indexación particionada por términos para las listas invertidas pequeñas y asumir indexación particionada por documentos para listas de gran tamaño que permitan un buen nivel de paralelismo. En definitiva, ambos enfoques de indexación pueden ser utilizados intercambiablemente puesto una copia completa del índice está almacenada en la memoria principal del nodo.

Cuando no existe el requerimiento de interactuar con otros elementos como caches, entonces es posible relajar el requerimiento de procesar una transacción por cada sincronización. La idea es mantener, en todo momento, varias transacciones en distintos grados de avance respecto de su solución completa. Sin embargo, es necesario contar con una solución al problema de la concurrencia de lectores y escritores, respetando al menos el concepto de transacción atómica en el sentido que si una transacción de lectura llega primero al nodo, entonces un escritor que llega posteriormente debería modificar después de la transacción de lectura una lista invertida que tenga un término presente en las dos transacciones. Si hay más de un término en común, esta regla debería respetarse en todas las listas invertidas. Si se trata de un motor de búsqueda donde el cumplimiento de esta restricción de serialidad no es relevante, al menos debería respetarse la restricción a nivel de lista invertida individual para evitar que la transacción de lectura omita documentos durante el proceso de ranking.

Debido a que la estrategia BP puede avanzar el proceso de una transacción a lo largo de varios supersteps o sincronizaciones de barrera, entonces es posible ajustar el momento en

que una transacción de escritura modifica una lista invertida. Cuando se trata de consultas de tipo OR las listas invertidas se mantienen ordenadas por frecuencia de ocurrencia de los términos en los documentos. En este caso, la transacción de escritura debe recorrer la lista invertida secuencial examinando el primer y último ítem (doc_id, term_freq) de cada bloque de la lista con el fin de determinar el bloque en que debe ser insertado el nuevo ítem (doc_id, term_freq). Si cada thread mantiene una cola FIFO conteniendo las transacciones que deben ser ejecutadas en el superstep actual, entonces el proceso de búsqueda del bloque e inserción en el bloque seleccionado se puede realizar en el orden necesario para respetar la restricción de serialidad y atomicidad de las transacciones.

Si se impone que a cada transacción se le permita procesar hasta una cierta cantidad B de bloques en cada superstep, entonces es suficiente con procesar las transacciones en el orden dado por la cola FIFO para respetar la restricción de serialidad tanto a nivel de lista individual como a nivel de todas las listas de términos comunes a ambas transacciones. Para mejorar el balance de carga, además de imponer que cada transacción visite B bloques durante un superstep, también se puede imponer que en el superstep no se procesen más de una cierta cantidad Q de términos. Con esto se puede acotar la cantidad máxima de trabajo realizada por cualquiera de los threads durante el superstep.

En la Figura 4.7 se presenta el algoritmo propuesto para el caso de procesamiento sincrónico particionado por términos (indexación global). Este es el algoritmo recomendado para métodos de ranking que presentan un nivel bajo de paralelismo tales como servicios de índice que contienen índices invertidos muy pequeños (e.g., publicidad) donde una gran mayoría de las listas frecuentemente referenciadas por las consultas son listas con muy pocos ítemes (doc_id, term_freq) o para casos en que el método de ranking utiliza una estrategia de terminación temprana muy agresiva que en la práctica conduce a examinar pocos ítemes.

Para el caso de consultas de tipo AND, dado que las listas se mantienen ordenadas por el identificador de documentos, la inserción de un nuevo documento es sencilla puesta que debe ir directamente al último bloque de la lista (o crear uno nuevo al final si dicho bloque si está completo). La actualización de un documento existente requiere de un procedimiento similar al empleado para las listas ordenadas por frecuencia, es decir, es necesario realizar una búsqueda saltando de bloque en bloque.

Por lo general, las eliminaciones de documentos en las listas invertidas requieren reubicar ítemes (doc_id, term_freq) y posiblemente reducir la cantidad de bloques y es por lo tanto una operación muy cara en tiempo de ejecución. Especialmente si las listas están

```
BulkProcessing-TermPartitioning( tid )

  while true do
    for each operation Op in the thread FIFO queue do
      for each block b in the next B blocks of list L_t with t = Op.term do
        if Op.type == QUERY then
          D = rank( F_t[b] )
          R_t = updateLocalTopK( R_t, D )
        else
        if  f_t ≥ F_t[b][0]  and  f_t ≤ F_t[b][K − 1]  then
          d = Op.doc_id
          if Tn.type == UPDATE then
            update( L_t[b], pair(d,f_t) )
          else
            insert( L_t[b], pair(d,f_t) )
          endif
        endif
      endfor
    endfor
    Barrier()
    for each finished transaction Tn do
      if (Tn.type == QUERY) and (Tn.id mod N_t == tid) then
        for each term t in Tn do
          R_global = updateGlobalTopK( R_global, R_t )
        endfor
        ouputQueue[ tid ].store( R_global )
      endif
      freeTransactionState( Tn )
    endfor
    if tid == 0 then
      Assign newly arriving transactions to each thread FIFO queue
       so that each read/write operation on a term t is assigned to
       thread with tid = t mod N_t. At all time each queue maintains
       upto Q terms, so when a queue gets full the remaining terms
       are assigned in the next supersteps in FIFO order.
    endif
    Barrier()
  endwhile

  EndBulkProcessing
```

Figura 4.7: Pasos ejecutados por cada thread en paralelo con los otros $N_t - 1$ threads.

comprimidas. En términos prácticos es más eficiente marcar como "borrado" los documentos que son eliminados y realizar periódicamente, en momentos de tráfico bajo de consultas, la recolección de basura en las listas invertidas que hayan alcanzado un cierto porcentaje de relevante de elementos eliminados. Por otra parte, si un determinado bloque contiene elementos eliminados, es posible que esos espacios sean ocupados por la inserción de nuevos documentos de manera que realizar recolección de basura a intervalos largos de tiempo puede ayudar a amortizar el costo total de las eliminaciones.

4.4. Sobre la Administración de los Caches

Para evitar la fragmentación del espacio de memoria principal, los caches se implementan como un conjunto de bloques del mismo tamaño. En nuestro caso, estos bloques coinciden con los bloques de las listas invertidas.

Independientemente de la política de admisión y desalojo aplicada en los caches del nodo de búsqueda, el tipo de dato abstracto empleado para seleccionar los elementos de mayor prioridad de ser reemplazados, debe implementar eficientemente la recuperación de los n_E elementos de mayor prioridad y la inserción de n_I elementos con prioridad aleatoria. En particular, para el caso del cache de listas normalmente será necesario extraer $n_E \gg 1$ elementos e inserta $n_I = n_E$ elementos con la misma prioridad, donde siempre se aplica una extracción inmediatamente seguida de una inserción. En este caso, los elementos son bloques de listas invertidas y cuando se trata de bloques recuperados desde memoria secundaria, es conveniente que dichos bloques sean almacenados en memoria contigua para reducir la cantidad de accesos a disco y aumentar la velocidad de transferencia de bloques entre el disco y la memoria principal. Para el cache top-K conviene extraer $n_E = N_t$ elementos en un solo acceso cuando el *query solver* trabaja con N_t threads en paralelo e insertar $n_I = N_t$ nuevos resultados top-K en esas n_E entradas del cache. Por otra parte, generalmente los caches para nodos de búsqueda se diseñan para un número fijo de entradas. Se asume que el cache tiene un total de $n = m \cdot h$ entradas y que a cada entrada se le asocia un valor numérico que representa la prioridad de ser reemplazada con un nuevo contenido.

A continuación, se propone una cola de prioridad que permite realizar las operaciones de extracción e inserción anteriormente descritas de manera eficiente. La estructura de datos es ilustrada en la Figura 4.8. Los nodos ovalados verticales llamados *buckets*, son todos del mismo tamaño y contienen m elementos donde cada elemento está asociado

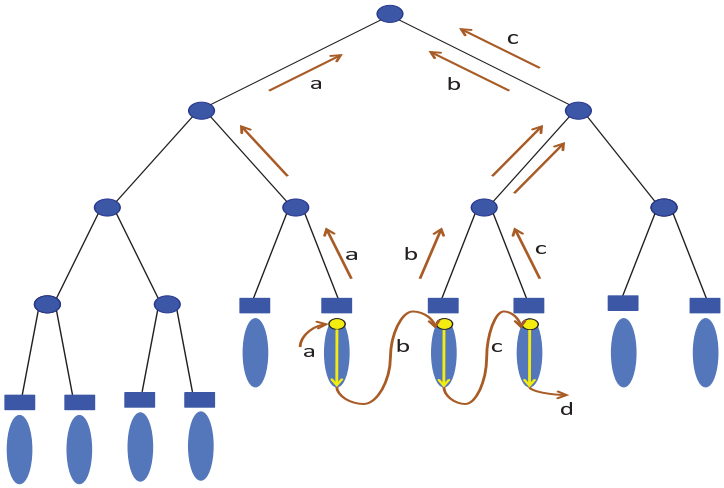

Figura 4.8: Cola de prioridad para la administración de caches.

con un bloque del cache. Los m elementos referencian a m bloques ubicados en memoria contigua. Los bloques de las listas invertidas son asociados a estos elementos de manera dinámica de acuerdo con la política de administración del cache.

Sobre los h buckets existe un árbol binario completo destinado a seleccionar en tiempo $O(\log h)$ el nodo que contiene el elemento de mayor prioridad. Cada hoja del árbol es asociada con un bucket. Se define como elemento de mayor prioridad al elemento de menor valor numérico. Los elementos almacenados en los nodos son pares (valor_prioridad, puntero_bloque_cache). Se mantiene un arreglo Prio[1 ... h] que contiene el valor numérico del elemento de mayor prioridad en cada uno de los h buckets. La función del árbol es determinar el mínimo global entre los h mínimos locales, es decir, el elemento de mayor prioridad entre los h elementos que son representantes de sus respectivos buckets.

El árbol es un arreglo de enteros CBT[1 ... $2\,h - 1$] tal que cualquier elemento CBT[i] del arreglo puede contener un valor en el rango 1 ... h. En particular, el valor Prio[CBT[1]] es el menor valor numérico entre las $n = m \cdot h$ prioridades almacenadas en la cola de prioridad. El árbol está representado de manera implícita en el arreglo donde cualquier nodo interno en la posición CBT[p] tiene a sus hijos en las posiciones CBT[$2\,p$] y CBT[$2\,p + 1$].

Para hacer que la raíz CBT[1] del árbol haga referencia a la mejor prioridad en todo el conjunto, recursivamente desde cada hoja hasta la raíz, se aplica la siguiente asignación a cada nodo interno CBT[p] del árbol,

if Prio[CBT[$2p$]] < Prio[CBT[$2p+1$]] **then**
  CBT[p] = CBT[$2p$]
else
  CBT[p] = CBT[$2p+1$]
endif

Es decir, el árbol se usa para mantener un torneo binario entre los valores almacenados en el arreglo Prio[1 ... h]. Cada vez que se modifica un valor Prio[i] se debe re-establecer el invariante a partir del nodo interno ubicado en la posición $\lfloor i \rfloor$ hasta la raíz. Esta operación toma costo $O(\log h)$. La ventaja sobre el heap clásico es que cada paso de actualización en la estructura de datos requiere de una comparación entre dos valores de prioridad. También, todas las actualizaciones no requieren llegar hasta la raíz del árbol.

Las inserción de un nuevo par (valor_prioridad, puntero_bloque_cache) se hace en el bucket referenciado por i = CBT[1]. El nuevo par ocupa el lugar del par con la menor prioridad en el bucket i y por lo tanto se modifica el valor de Prio[i], y entonces es necesario actualizar el camino hacia la raíz del árbol CBT. Dado que típicamente, por ser una aplicación para caches, el nuevo par tendrá la mayor prioridad en el bucket, lo cual conduce a la realización de un recorrido secuencial de los $m-1$ pares almacenados en bucket i para determinar el par con la menor prioridad. Sin embargo, si se trata de la inserción de una secuencia de pares con la misma prioridad este costo $O(m)$ será amortizado rápidamente.

Por ejemplo, cuando son bloques de una lista invertida éstos podrían incluso superar la capacidad del bucket i, involucrando a nuevos buckets seleccionados por los sucesivos valores de CBT[1] obtenidos luego de cada actualización del árbol. La Figura 4.8 muestra tal situación, en la cual la inserción de los bloques de una lista invertida involucra ocupar tres buckets completos (a), (b) y (c), y parte de un cuarto bucket (d) no identificado en la figura. También se muestran las activaciones de la rutina de actualización del árbol. El resultado de cada actualización le permite a la rutina de inserción saber cual es el siguiente bucket.

La paralelización de esta cola de prioridad puede ser realizada en forma exacta o de manera aproximada para producir un mejor rendimiento. A continuación se describen dos propuestas de métodos de paralelización.

4.4.1. Paralelización con N_t threads

Para sistemas de memoria distribuida, una estrategia de paralelización del problema de colas de prioridad que alcanza un buen rendimiento en la práctica, consiste en simplemente mantener P colas de prioridad secuenciales. Desde cada cola se extraen n_E elementos para formar un conjunto con $n_E \cdot P$ elementos y extraer desde ellos los n_E elementos de mejor prioridad. Los restantes $n_E \cdot (P-1)$ elementos se re-insertan en las P colas de prioridad. La inserción de n_I elementos se hace simplemente insertando n_I/P elementos en cada cola.

Para una cola de prioridad con N elementos y suponiendo $n = n_E = n_I$, el costo BSP de esta estrategia es $O(\,(n \cdot P) \cdot \log(n \cdot P) + \log(N/P) + (n \cdot P) \cdot g + \ell\,)$ donde el g es el costo de la red comunicación y ℓ la latencia de la sincronización entre procesadores. Los factores $(n \cdot P)$ pueden ser mejorados a $O(n)$, con gran probabilidad, realizando unas pocas iteraciones en que cada procesador extrae $O(n_E/P)$ elementos, y al final de cada iteración preguntar a los P procesadores si tienen elementos con mejor prioridad que los actuales n_E elementos seleccionados como los de mejor prioridad en forma global.

La misma estrategia puede ser implementada en la memoria compartida de un nodo de búsqueda con la ventaja que (**a**) $g \gg g_1 + g_2$ donde g_1 representa el costo de transferir datos desde el cache L2 al cache L1 en el procesador multi-core y g_2 el costo de transferir datos desde la memoria compartida al cache L2 del procesador, y que (**b**) las colas de prioridad individuales aumentan la localidad de referencias de los threads.

Si en el cache se requiere insertar un total de n elementos provenientes de una lista invertida, tal que $n \gg N_t$, entonces una estrategia razonable es mantener $P = N_t$ árboles CBT y utilizarlos para determinar los N_t buckets de menor prioridad. Cada CBT se hace cargo de h/N_t buckets distintos. Una vez que se han obtenido los N_t buckets, cada thread reemplaza los elementos con prioridad igual al elemento de menor prioridad en el bucket. Tan pronto como se encuentra un elemento con prioridad distinta en cada bucket o uno o más de éstos se repletan de nuevos elementos, es necesario volver a repetir este proceso determinando los siguientes N_t buckets de menor prioridad. Cada uno de estos ciclos es delimitado por una barrera de sincronización de threads o superstep. Esta estrategia produce resultados exactos sólo cuando en los buckets se almacenan elementos con la misma prioridad.

La estrategia de los N_t CBTs falla cuando se trata de un sistema diseñado para utilizar un número variable de threads. Para este caso, una solución es utilizar la estrategia propuesta en [58, 57]. En este caso, los nodos internos del CBT contienen $c \cdot N_t^S$ elemen-

tos, donde N_t^S es el total de threads utilizados en régimen permanente y $c \geq 1$. En cada nodo interno se mantiene el invariante de que un nodo padre tiene mayor prioridad que un nodo hijo si todos los elementos referenciados por el padre tienen menor prioridad que los que referencia el nodo hijo. Cuando se viola el invariante lo que se hace es intercambiar elementos entre los dos nodos hasta re-establecerlo. Una actualización toma un total de $\log(\, h/(c \cdot N_t^S)\,)$ pasos, donde en cada paso un número variable de threads $N_t \leq N_t^S$ puede participar en el re-establecimiento del invariante. Una implementación para Posix-PThreads de esta estrategia para el procesador Intel fue presentada en [67]. Los resultados muestran que esta estrategia escala razonablemente bien con el número de threads.

Cuando los buckets son de un tamaño lo suficientemente grande como para hacer probable la existencia de elementos con distintas prioridades, las estrategias anteriores que utilizan tanto el enfoque de N_t colas de prioridad para seleccionar los buckets o una sola cola de prioridad con nodos internos conteniendo referencias a $c{\cdot}N_t^S$ buckets, se obtiene una solución que produce resultados aproximados. En teoría, es posible que existan resultados aproximados, pero en la práctica, para índices con listas invertidas conteniendo una gran cantidad de bloques con la misma prioridad, es muy poco probable que esto ocurra (en la experimentación presentada en el Capítulo 5 no se produjo ningún caso).

Un solución pragmática para listas invertidas y para sistemas que utilizan N_t threads en forma permanente, es aplicar los principios de indexación local manteniendo N_t colas de prioridad independientes, cada una compuesta de los arreglos CBT[1 ... $h/N_t - 1$] y Prio[1 ... h/N_t] y un conjunto de h/N_t buckets de tamaño m/N_t. Los bloques de las listas invertidas se distribuyen uniformemente al azar en las N_t colas de prioridad utilizando una regla tal como $tid = id_bloque$ módulo N_t. La experimentación presentada en el Capítulo 5 considera esta implementación puesto que puede trabajar de manera natural con la estrategia BP de la Figura 4.5.

Por otra parte, una estrategia aproximada para un sistema con número variable de threads puede utilizar el CBT con nodos internos de tamaño $c \cdot N_t^S$ y asignar un thread a cada bucket de tamaño m, es decir, en este caso se aplica el principio de indexación global.

4.5. Conclusiones

Las estrategias propuestas respetan el principio básico de toda transacción, es decir, la atomicidad. En nuestro caso, esto equivale a que el resultado de una transacción sea consistente con el estado de las listas invertidas al momento en que se inicia la ejecución de la transacción. Es decir, durante el recorrido de las listas invertidas y respectivo ranking de documentos, ningún otro thread escritor modifica el contenido de la lista. Esto es evidente para la estrategia BP presentada en la Figura 4.5, puesto que todos los threads procesan en paralelo cada transacción.

En la estrategia BP presentada en la Figura 4.7, el empleo de la cola FIFO en cada thread y la manera en que son procesados los bloques de las listas invertidas, cualquier escritor que haya arribado al *query solver* después de un lector, va a modificar la lista invertida después de que el lector la haya leído.

La serialidad de transacciones tiene que ver con el concepto de que el resultado final de un conjunto de transacciones concurrentes debe ser equivalente al resultado que se obtiene cuando las transacciones son ejecutadas secuencialmente. Las estrategias BP son seriales puesto que toman secuencialmente las transacciones desde la cola de entrada y respetan ese orden en la ejecución de las operaciones de lectura y escritura sobre las listas invertidas.

Para la cola de prioridad propuesta en este capítulo, la recomendación es utilizar el método exacto para caches top-K debido a su poco volumen de datos, y el método aproximado para caches de listas debido a su gran volumen de datos. Típicamente en una sistema en producción, el tamaño del cache de listas es de alrededor del 20 % del tamaño del índice comprimido, mientras que el tamaño del cache de top-K es de un 5 %.

Capítulo 5

Comparación de Estrategias

En este capítulo se presenta un estudio que compara la estrategia BP con diversas alternativas para el mismo problema tomadas desde la literatura de bases de datos y sistemas operativos. Se utilizaron implementaciones reales ejecutadas sobre el procesador Intel. Como cargas de trabajo se utilizaron benchmarks diseñados para simular motores de búsqueda en texto (*index service*) y un servicio de *user click-through* el cual debe responder en tiempo real con información de documentos visitados por usuarios anteriores para mejorar el ranking de documentos realizados por el motor de búsqueda.

5.1. Estrategias alternativas

Para comparar el algoritmo BP de la Figura 4.5 se han escogido de la literatura actual una serie de estrategias de control de concurrencia para transacciones de lectura y escritura, las cuales fueron adaptadas al contexto de esta tesis. Sus principales características se resumen en la Tabla 5.1. Los pseudo-códigos con los detalles de cada estrategia se presentan en el Apéndice B. En esta tabla, la primera columna indica si la estrategia de control de concurrencia es serializable o no. La segunda columna indica el tipo de paralelismo explotado en el procesador multi-core, es decir, un thread por transacción (prefijo "C") o todos los threads sobre una única transacción (prefijo "P"). La tercera columna indica la primitiva de sincronización que se utiliza para sincronizar los threads y así prevenir conflictos entre lectores y escritores operando sobre las listas invertidas. En nuestro caso, un conflicto es un evento desastroso puesto que durante el desarrollo de las implementaciones `C++` un sólo conflicto simplemente provocaba la caída del programa en ejecución.

Estrategia	Serializable	Tipo de Parallelismo	Sincronización
BP	Yes	PRT and PWT	Thread Barrier
CR	Yes	CRT and PWT	Thread Barrier
TLP1	Yes	CRT (term sharing) and CWT	R/W List Locking
TLP2	Yes	CRT (no sharing) and CWT	Exclusive List Locking
RBLP	No	non-atomic: CRT and CWT	Block Locking
RTLP	No	non-atomic: CRT and CWT	Term Locking

CRT= concurrent read transactions PRT= read transaction in parallel
CWT= concurrent write transactions PWT= write transaction in parallel

Tabla 5.1: Estrategias de sincronización de transacciones.

Las estrategias más restrictivas garantizan la serialización de las transacciones, es decir, se mantiene la ilusión de la ejecución en serie. Las otras estrategias pueden ser aplicadas en escenarios donde no se exige serialidad, y potencialmente pueden ofrecer un mejor rendimiento puesto que son más relajadas en exigencias de sincronización. Al igual que la estrategia BP, todas las estrategias utilizan el mismo esquema de *fetching* y *ranking* sobre la estructura de *fast-track* del procesador. Para hacer más justa la comparación, todas las estrategias contienen una cola de entrada de transacciones donde, en varios casos, el paso fundamental hacia la serialización es hacer un lock de acceso exclusivo a esta cola para retirar una transacción y luego liberar el lock tan pronto como el respectivo protocolo de sincronización lo permita.

El programa `C++` utilizado en las implementaciones es, en su mayor parte, idéntico para cada estrategia y los únicos cambios en el código tienen que ver con las primitivas de sincronización y algunas estructuras de datos mínimas de apoyo. En otras palabras, los tiempos de ejecución de cada estrategia se ven afectados solamente por los efectos del protocolo de sincronización y no por detalles de programación. En cualquier caso, las implementaciones son eficientes puesto que logran aceleraciones muy cercanas al óptimo cuando se aumenta el número de threads.

La estrategia *Concurrent-Reads (CR)* es similar a la estrategia BP de la Figura 4.5 en términos de la realización de las operaciones de escritura utilizando todos los threads en paralelo. La diferencia es que permite el solapamiento entre transacciones de lectura a lo largo del tiempo mediante la asignación de un thread a cada consulta. Cada thread resuelve de principio a fin la consulta asignada. Antes de procesar una transacción de escritura, CR espera a que todas las transacciones concurrentes de lectura en curso hayan finalizado (más detalles en la Figura B.1 del Apéndice B). De esta manera, la estrategia CR explota el paralelismo disponible desde todas consultas activas (transacciones de sólo lectura) en

un periodo de tiempo. Dado que algunos threads pueden quedar sin trabajo antes del comienzo de la siguiente transacción de escritura, se probó una versión de CR donde los threads sin trabajo cooperan con los demás para terminar las transacciones de lectura en curso. Sin embargo, no se observó mejoras en el rendimiento debido a la computación extra que es necesario hacer para la planificación de threads. En un sentido esta estrategia es similar a la estrategia BP presentada en la Figura 4.7 y puede ser utilizada como una alternativa a la estrategia BP de la Figura 4.5 cuando las listas invertidas son relativamente pequeñas.

La estrategia *Term-Level-Parallelism (TLP)* permite el solapamiento entre transacciones de lectura y escritura. Cada transacción es ejecutada por un único thread y se utilizan locks de exclusión mutua para proteger las listas invertidas involucradas. El lock sobre un término en realidad implica un lock sobre la respectiva lista invertida en forma completa. Para garantizar la serialidad de transacciones, un thread dado no libera el lock sobre la cola de entrada de transacciones hasta que no ha adquirido los locks de todos los términos que debe utilizar. Posteriormente, cada lock es liberado tan pronto como la respectiva lista invertida ha sido procesada. Se implementaron dos versiones de TLP.

La primera versión, llamada *TLP1*, permite transacciones simultáneas de sólo lectura sobre las mismas listas invertidas. Para esto fue necesario implementar locks de lectura, es decir, locks que no bloquean a los threads conteniendo transacciones de lectura y que tienen términos en común, mientras que los threads conteniendo transacciones de escritura son bloqueados si comparten términos con las transacciones de lectura. Las transacciones de escritura utilizan locks de acceso exclusivo a las listas invertidas. Los locks son servidos en modo FIFO lo cual garantiza la serialidad (más detalles en la Figura B.2).

La segunda versión, llamada *TLP2*, es mucho más fácil de implementar puesto que se impide el solapamiento de transacciones de lectura o escritura que tengan términos en común. Esta estrategia es similar al protocolo de 2 fases de bases de datos. Una vez que el thread adquiere el lock sobre la cola de entrada de transacciones, procede a solicitar un lock de acceso exclusivo para cada uno de los términos involucrados en la transacción. No existe posibilidad de deadlocks debido a que el lock de acceso exclusivo a la cola de entrada de transacciones no se libera hasta que el thread ha adquirido todos los locks (más detalles en la Figura B.3).

También se estudiaron dos estrategias adicionales que relajan los requisitos de serialidad y atomicidad de las transacciones. Al no adherir a estos dos requisitos, el nivel de concurrencia de los threads aumenta significativamente. Para motores de búsqueda, donde

las consultas de usuarios son respondidas en una fracción de segundo, ambos requisitos podrían ser prescindibles. Para otras aplicaciones tales como sistemas de bolsa electrónica no es tan claro que estos requisitos sean prescindibles. Por otra parte, estas dos estrategias son buenos representantes del mejor rendimiento que podría alcanzar una estrategia optimista que cumpla al menos el requisito de atomicidad por la vía de abortar y reejecutar una transacción para la cual se detecta que las versiones de las listas invertidas involucradas no son las que existían al momento de iniciar su ejecución.

La primera estrategia, llamada *Relaxed-Block-Level-Parallelism (RBLP)*, es similar a TLP pero no obliga a los threads a hacer un lock de la lista invertida completa por cada término involucrado tanto para transacciones de lectura como para transacciones de escritura. Las listas invertidas están compuestas de un conjunto de bloques. Por lo tanto, los threads van haciendo locks exclusivos sólo de los bloques de las listas invertidas que se están procesando en un momento dado del tiempo. Es decir, es minimal, se aplica el lock sólo en la sección de la lista invertida donde es necesario prevenir que dos o más threads lean y escriban los mismos ítemes (doc_id, term_freq) de la lista. En la implementación de esta estrategia, debido a que los bloques pueden eventualmente llenarse de ítemes y por lo tanto dividirse, lo que se hace es aplicar locks exclusivos sobre dos bloques consecutivos en los casos en que el número de ítemes dentro del bloque actual esté más allá de un valor umbral $B - N_t$ donde B es el tamaño del bloque (es decir, existe una probabilidad de que una transacción de escritura divida el bloque que se ha llenado siguiendo el algoritmo descrito al final de la Sección 4.2.1).

Dado que no existe el requerimiento de serialidad y atomicidad, existen tres optimizaciones adicionales que incrementan el nivel de concurrencia. Primero, los threads liberan el lock de acceso exclusivo sobre la cola de entrada de transacciones tan pronto como copian los datos de la transacción a memoria local. Segundo, los threads liberan los locks sobre los bloques tan pronto como van siendo copiados al *fast-track*, y por lo tanto el ranking de los documentos presentes en el bloque puede involucrar una gran cantidad de cómputo sin que este retardo afecte el nivel de concurrencia. En los experimentos realizados se detectó que el costo del ranking de documentos es la parte dominante en el tiempo de ejecución. Tercero, durante la solución de una consulta, el ranking de documentos avanza considerando un sólo bloque por cada término en oposición al caso de primero procesar todos los bloques de un término para pasar al siguiente. Esto reduce el nivel de contención entre threads que comparten términos. Los detalles de RBLP se presentan en la Figura B.5.

El segundo enfoque, llamado *Relaxed-Term-Level-Parallelism (RTLP)*, es similar a RBLP pero antes de acceder a un bloque se hace un lock del término en lugar de lo-

ck de bloques como en RBLP. Esto evita la necesidad de tener que decidir si hacer un lock de dos bloques consecutivos durante el recorrido de cada lista invertida. Otra ventaja práctica es la significativa reducción en el tamaño del arreglo de locks que es necesario definir en la implementación. En rigor, en RTLP se requiere definir una variable de tipo lock por cada término del índice invertido, mientras que en RBLP es mucho más que esa cantidad puesto que por cada término se tienen muchos bloques. En la implementación de RBLP y RTLP en realidad se utiliza un arreglo de variables de tipo lock mucho menor y se utiliza hashing sobre ese arreglo para requerir un lock de término y de bloque. Naturalmente, para un mismo tamaño de arreglo, la probabilidad de colisiones es mucho menor en el caso de RTLP que en RBLP. Los detalles de la estrategia RTLP se presentan en la Figura B.4.

5.2. Evaluación utilizando un Servicio de Indice

En los experimentos que siguen se considera que siempre el conjunto de listas invertidas requeridas para procesar una transacción se encuentra disponible en el cache de listas. Las evaluaciones realizadas no consideran el costo de administración del cache. En una experimentación separada se evalúa el rendimiento de la propuesta de cola de prioridad para la administración del cache de listas. La motivación para separar ambos experimentos es en primer lugar simplicidad y precisión de los resultados, y en segundo lugar es el hecho de que si la estrategia BP se comporta mejor que las alternativas asincrónicas en el caso de listas siempre residentes en memoria principal, entonces éste mismo enfoque debe ser aplicado a la administración del cache.

Los experimentos se realizaron en el procesador Intel descrito en el Apéndice A. Se asume que la tasa de llegada de transacciones es lo suficientemente alta como para mantener ocupado a todos los threads. La métrica principal de interés es la tasa de salida de transacciones, es decir, el throughput. Esto porque todas las estrategias son capaces de resolver una transacción en una fracción de segundo.

Las listas invertidas fueron construidas desde una muestra de la Web de UK proporcionada por Yahoo!. La Figura 5.1 muestra la distribución del largo de las listas invertidas y la distribución de términos en los documentos. Para las transacciones de lectura se utilizó un log de consultas real que contiene búsquedas de usuarios sobre dicha Web. El texto de los documentos que participan en las transacciones de escritura fue tomado desde la misma Web. La generación del flujo de transacciones (traza) que llegan al nodo de búsqueda se