FACULTAD DE INGENIERÍA

INGENIERÍA CIVIL INFORMÁTICA

Una arquitectura cache para aplicaciones web de gran escala basada en el comportamiento del usuario

Proyecto de título para optar al grado de ingeniero civil en informática.

Autor

Daniela Lucía Baeza Rocha

Profeso guía: Carlos Gómez-Pantoja

Santiago de Chile

Octubre, 2016

Índice general

Índice de figuras

Índice de tablas

Resumen

El *Web Caching* nació como una tecnología que permite disminuir el tráfico redundante en la web. Las aplicaciones web de gran escala utilizan esta tecnología para disminuir el acceso al servidor web (*back-end*), donde se encuentran las respuestas a las consultas solicitadas por los usuarios, al mismo tiempo reduce los tiempos de respuesta hacia el cliente. Esto se logra por medio de la existencia de nodos cache con capacidad limitada, donde son almacenadas respuestas pre-computadas a las consultas de usuario. La limitación en el espacio del cache, hace necesario decidir que consultas deben ser almacenadas en cache, privilegiando aquellas entradas que puedan ser referenciadas nuevamente en un futuro cercano.

Las consultas se pueden clasificar según su variación en tiempo y frecuencia en que son solicitadas. Existiendo consultas que son referenciadas de manera permanente por los usuarios y otras que tienen un aumento explosivo teniendo una duración de tiempo acotado (consultas tipo ráfaga). Mantener estas consultas en cache se hace cada vez más crucial para disminuir los accesos al back-end, lo que se traduce en la mejora en los tiempos de respuesta hacia el usuario.

En este proyecto de título se propone una arquitectura cache para aplicaciones web de gran escala que tome en consideración el comportamiento del usuario, por medio de la asignación de secciones donde se almacenen consultas permanentes, consultas ráfaga y consultas variables. Los experimentos muestran que la arquitectura cache propuesta tiene un mejor rendimiento con respecto a la utilización de estrategias básicas de cache. Esto es un 10,8% promedio de hits en cache con respecto al algoritmo base LRU y un 36% promedio de hits en cache con respecto al algoritmo base LFU.

Capítulo 1

1. Introducción

1.1. Motivación

El nacimiento de la *World Wide Web* (*Web*) en el año 1990 permitió la distribución de documentos de hipertexto vía Internet, los que unos años más tarde fueron visualizados por usuarios en distintas partes del mundo por medio de navegadores Web. La posibilidad de compartir diferentes tipos de documentos e información en la Web y la interconexión entre distintos usuarios en diferentes zonas geográficas ha tenido como consecuencia la masificación en el uso de la Web, llegando a ser utilizadas por personas de todo el mundo que se conectan de manera diaria.

El continuo crecimiento de la Web ha conllevado la aparición de aplicaciones web de gran escala que deben interactuar con una gran cantidad de usuarios y procesar altos volúmenes de información. El intenso tráfico generado entre los usuarios y la aplicación, provoca la existencia de demoras en el acceso a los distintos objetos de la Web, esto se puede deber no solo a la gran cantidad de clientes que se conectan al servidor web para obtener algún objeto, sino que también a que muchos de ellos desean adquirir la misma información en un corto periodo de tiempo. Con el fin de mejorar el acceso a la Web se comenzó a utilizar distintas tecnologías, una de ellas es el uso del *Web Caching*.

El *Web Caching* es una tecnología utilizada para disminuir el tráfico redundante en la red aumentando la disponibilidad del ancho de banda y reduciendo la congestión, mejorar los tiempos de respuesta a los usuarios, ahorrar costos (p.ej.: costo de ancho de banda), entre otros (Nagaraj, 2004, pág. xxii). Se compone de varios nodos cache donde se almacenan los distintos documentos web que pueden ser solicitados por los usuarios en un futuro cercano, tal que no se deba buscar constantemente la respuesta al servidor web que la contiene (*back-end*). Esto quiere decir, mientras mayor sea la cantidad de peticiones a las que se pueda dar respuesta en un nodo cache (hit en cache), menor será el ingreso al servidor web, disminuyendo la congestión en la red, el ancho de banda y el tiempo de respuesta a los usuarios de la red.

Las peticiones realizadas por los usuarios en la web sigue un comportamiento dinámico con un alto sesgo entre los distintos grupos de consultas, es decir, existen peticiones que son constantemente solicitada por los usuarios en distintos periodos de tiempos en comparación con otras que solo son pedidas una única vez. Además, existen ocasiones en que ciertas peticiones aparecen de manera explosiva, generado por un acontecimiento de interés popular que tiene como consecuencia una alta demanda a los nodos cache en un corto periodo de tiempo. Al no contar con algún tipo de control puede producir latencia y congestión en la red, incluso el colapso de algunos componentes del servicio.

Lo descrito en el párrafo anterior lleva a determinar que es importante tener en consideración el comportamiento del usuario a la hora de diseñar una arquitectura cache. Es decir, a la hora de decidir que consultas deben ser almacenadas en cache, se tome en cuenta el dinamismo de las peticiones hecha por los usuarios, teniendo en especial consideración aquellas que tienen una alta frecuencia en cortos periodos de tiempo.

Este proyecto de título se enfoca en proponer una nueva arquitectura cache que tome en consideración el comportamiento del usuario y logre mejorar la eficiencia del cache para las aplicaciones web de gran escala.

1.2. Antecedentes preliminares

Las arquitecturas cache para aplicaciones web de gran escala deben ser capaces de dar respuesta a una gran cantidad de peticiones realizadas por los usuarios de manera diaria. Además, el cache posee un espacio de almacenamiento limitado, por lo que debe priorizar mantener en cache aquellas respuestas a consultas que puedan ser referenciadas por los usuarios nuevamente (generando un hit en cache) y así disminuir el tráfico hacia el back-end.

Las consultas efectuadas por los usuarios en la web siguen un comportamiento Zipfiano, esto quiere decir que existen grupos de palabras a los que se hace referencia de manera frecuente, mientras que otros grupos de palabras son referenciadas ocasionalmente.

En la literatura los tipos de consultas realizadas por los usuarios se clasifican según su variación en tiempo y frecuencia, estas corresponden a: las consultas permanentes, son peticiones que se referencian de manera frecuente y perduran en el tiempo; las consultas en ráfaga, son aquellas en las que su frecuencia sube de forma abrupta en un periodo de tiempo acotado para terminar desapareciendo de la misma forma; las consultas variables son aquellas que poseen un dinamismo propio y cambian según los intereses momentáneos de los usuarios; entre otras.

Las consultas en ráfaga pueden generar distintos problemas a nivel de cache, esto se debe a que un gran número de usuarios hace referencia a un

tópico en particular y si el cache no se encuentra bien balanceado a la hora de recibir estas consultas, puede producir latencia en la entrega de las respuestas a los usuarios y/o fallos en algunos de los nodos cache. Hoy en día existen mecanismos que permiten detectar de manera temprana y de forma *on-line* las consultas en ráfaga.

Este trabajo estará enfocado en considerar el comportamiento del usuario a la hora de diseñar la nueva arquitectura cache. Para esto se consideraran los diferentes tipos de consultas de usuario que serán almacenadas en el cache.

1.3. Contribución de la tesis

La contribución del proyecto de título, es la creación de una nueva arquitectura cache para aplicaciones web de gran escala, capaz de considerar el comportamiento de los usuarios a la hora de decidir los objetos a ser almacenados en el cache, mejorando la tasa de hit en comparación con otras arquitecturas cache. Específicamente las contribuciones de este trabajo son:

- Diseño de una arquitectura cache subdividido en tres secciones, con el fin de almacenar distintos tipos de consultas, en cada una de ellas, aumentando la tasa de hit en cache. La división del cache es la siguiente: una sección para almacenar consultas del tipo ráfaga, otra sección para guardar consultas del tipo permanente y por último una sección para mantener las consultas variables.

- Implementación de un mecanismo on-line de detección de consultas en ráfaga propuesto por Gómez Pantoja (2014), el que será utilizado para detectar y almacenar este tipo de consultas en el cache ráfaga de la arquitectura propuesta.

1.4. Estructura de la tesis

El capítulo 2 contiene la descripción del problema, se define el objetivo general, los específicos, la hipótesis y los alcances del proyecto de título. Además, se describe la metodología de trabajo utilizado para el desarrollo del proyecto.

El capítulo 3 contiene el marco teórico donde se definen distintos conceptos necesarios para abordar el proyecto de título. Entre los conceptos definidos se encuentra: las aplicaciones web de gran escala, los motores de búsqueda web, las redes sociales, el cache, la organización del cache donde se describen las distintas políticas de cache existentes, la estructura del cache y la clasificación del cache, por último se encuentra el comportamiento del usuario donde se explica la ley de Zipf y se describen los tipos de peticiones de usuario.

En el capítulo 4 se presenta una revisión bibliográfica de los trabajos realizados sobre las arquitecturas cache y las políticas de cache, con el fin de diseñar una arquitectura de cache que no exista en la literatura.

El capítulo 5 contiene el desarrollo de la solución, en las primeras dos secciones se describe la arquitectura cache propuesta. En la sección 5.3 se mencionan las diferencias que tienen con otras arquitecturas cache existentes.

El capítulo 6 se presentan las pruebas y resultados de la solución. En la sección 6.1 se dan a conocer las características de las pruebas, los algoritmos utilizados para el desarrollo de las pruebas. En la sección 6.2 se da a conocer los resultados obtenidos en la experimentación.

El capítulo 7 comprende las conclusiones del trabajo desarrollado según los resultados obtenidos en capítulo y una propuesta de trabajos futuros a desarrollar.

Capítulo 2

2. Descripción del problema

2.1. Contextualización

El *Web Caching* es una tecnología desarrollada para disminuir el tráfico redundante en la red, lo que genera distintos beneficios como el aumento en la disponibilidad del ancho de banda, reducir la congestión, disminuir los tiempos de respuesta, etc. Esto se logra por medio de la existencia de distintos nodos cache que se pueden situar en el cliente, en el servidor o en localidades intermedias de la red.

En los nodos cache se almacenan entradas que contienen respuestas pre-computadas de las peticiones realizadas por los usuarios, así cuando un usuario hace una petición la respuesta se encuentra en uno de estos nodos cache y es devuelta al usuario, reduciendo los costos y recursos utilizados al no tener que acceder al servidor web (*back-end*) por una respuesta.

Las consultas realizadas por los usuarios en la web varían en el tiempo, existiendo distintos tipos de consultas que se dividen según su variación en tiempo y frecuencia. Algunas de ellas son: las consultas permanente que son realizadas de manera diaria, las consultas en ráfaga que tienen una alta frecuencia en un corto periodo de tiempo, las consultas variables o dinámicas las que se realizan acorde a la necesidad del usuario en ese instante, etc.

El proyecto de título tiene como finalidad proponer e implementar una arquitectura cache para aplicaciones web de gran escala que sea capaz de considerar el comportamiento del usuario, en las que se asignará un espacio del cache para tres tipos de consultas (ráfaga, permanente y dinámicas).

Además se utilizará un mecanismo online para la detección de consultas en ráfaga eficiente en tiempo y espacio, y un mecanismo offline para determinar las consultas permanentes que serán almacenadas a priori en el cache.

La solución desarrollada será comparada en relación a estrategias básicas vistas en el estado del arte, en las que se obtendrá la tasa de hit en cache de ambas estrategias utilizando registros históricos, y así determinar la eficiencia de la solución propuesta.

2.2. Enunciado del problema

Las aplicaciones web de gran escala deben procesar grandes cantidades de peticiones realizadas por los usuarios de manera constante. Para disminuir la latencia producida por la búsqueda de la respuesta hacia el back-end, se utilizan nodos cache que se encuentran interconectados unos con otros.

El cache al tener un espacio limitado no puede almacenar todas las respuestas de las consultas de usuario. Es por esto, que una arquitectura cache debe privilegiar almacenar objetos que en un futuro cercano sea referenciado nuevamente produciendo una mayor tasa de hit en cache.

En trabajos como los de Bresalau, Cao, Fan, Phillips y Shenker (1999) muestran que las consultas de usuario siguen un comportamiento Zipfiano. Esto quiere decir que existen peticiones que van a ser realizadas con una mayor

frecuencia, mientras que otras van a ser referenciadas con una frecuencia muy baja, lo que puede producir un desbalance entre los distintos nodos cache puesto que unos serían más visitados que otros.

Además, las consultas de usuario poseen un dinamismo propio habiendo diferencias en la frecuencia y el contenido de las peticiones que son realizadas en el tiempo. En este contexto también existen peticiones que son realizadas de manera constante prevaleciendo en el tiempo, mientras que otras aparecen de manera abrupta y tienen una duración de tiempo acotado. Es este tipo de consulta que al producir un alza abrupta en el tráfico, puede generar congestión en los nodos cache que contiene dicha respuesta.

Al no tener en cuenta los distintos componentes estructurales del comportamiento de los usuarios, puede generar la congestión de los nodos cache aumentando la carga de trabajo y produciendo la posibilidad de saturación y fallas en los nodos, aumentando los tiempos de respuesta a los usuarios de la aplicación web, afectando directamente en la eficiencia del cache.

En la figura 2.1 se muestra el diagrama de Ishikawa con el fin de comprender los problemas relacionados con la ineficiencia del cache.

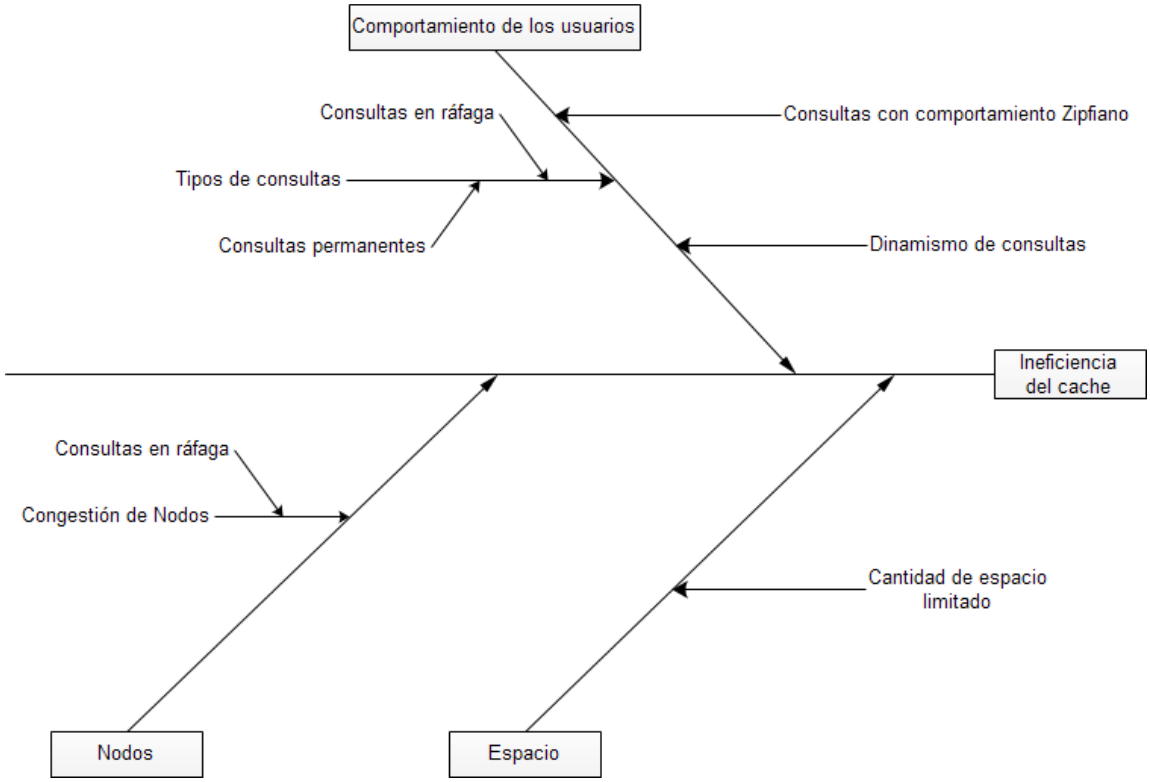

Figura 2.1: Diagrama Ishikawa. Fuente: Elaboración propia.

2.3. Objetivos e hipótesis

2.3.1. Objetivo general

Mejorar la eficiencia del cache para aplicaciones web de gran escala mediante el diseño de una arquitectura cache basada en el comportamiento de los usuarios.

2.3.2. Objetivos específicos

1. Determinar los componentes estructurales presentes en el comportamiento de usuario de aplicaciones web de gran escala.

2. Proponer una arquitectura cache que considere los distintos componentes estructurales del comportamiento de usuario.

3. Mejorar la tasa de hit respecto a una estrategia básica de cache en la literatura.

2.3.3. Hipótesis

Existen organizaciones lógicas del cache que considere el comportamiento de usuario y que permita mejorar la tasa de hit en relación a una estrategia básica del estado del arte.

2.4. Metodología de trabajo

La metodología utilizada en el proyecto de título corresponde al método empírico-analítico, basado en la experimentación y la lógica empírica el que consta de una serie de pasos ordenados que permiten determinar las características del suceso a estudiar. A partir del análisis del suceso se deducen conclusiones, las que serán sometidas a pruebas para determinar su validez. A continuación se desglosará a grandes rasgos las actividades realizadas en el proyecto.

Fase 1: Identificación de un problema de investigación

- Formulación de un problema.
- Identificación de factores importantes:
    - Construcción del Marco teórico.
    - Investigación del estado del arte.

Fase 2: Formulación de hipótesis

- Planteamiento del problema a analizar.
- Desarrollo de objetivos e hipótesis.

Fase 3: Probar hipótesis

- Proponer una solución.
- Desarrollo de la solución propuesta.

Fase 4: Resultados

- Evaluación de los resultados de la solución.
- Obtener conclusiones.

En el anexo 1 se encuentra la planificación utilizada en el proyecto de título.

2.4.1. Herramientas y entorno de trabajo

Las herramientas utilizadas para el desarrollo del proyecto de título son:

- Lenguaje de programación C++ compilador g++ 4.9.
- Lenguaje de programación C#.
- Entorno de desarrollo Microsoft Visual Studio 2012.
- Microsoft Office Word 2007.
- Microsoft Office Proyect 2007.

2.5. Alcances

El proyecto de título comprende la propuesta y experimentación de una arquitectura cache para aplicaciones web de gran escala. Donde se utilizará un mecanismo on-line para la detección de consultas en ráfaga y un mecanismo off-line para la detección de consultas permanentes.

En el contexto de este trabajo se utilizarán políticas (desalojo, admisión y de eliminación de resultados antiguos) básicas para el cache dinámico. En específico, la política de desalojo usada en las pruebas para la sección

dinámica de la arquitectura propuesta corresponden a *Least Recentently Used* (LRU) y *Least Frequently Used* (LFU). Las políticas a utilizar en el cache dinámico dependen directamente del tipo de aplicación web en la que se va a utilizar la solución propuesta, ya que existen políticas de cache que tienen mejores resultados dependiendo del tipo de aplicación web en las que son utilizadas. Se dejará como trabajo futuro la decisión de implementar una política específica acorde a la aplicación web en la que se desee utilizar la arquitectura cache propuesta.

Existen distintas vías de obtener de manera off-line las consultas permanentes. En este trabajo se obtendrán las consultas por medio de un registro histórico de un mes, donde serán seleccionadas las peticiones realizadas con mayor frecuencia por los usuarios para ser almacenadas en el cache estático. Se dejará como trabajo futuro buscar nuevas estrategias que permitan obtener una mayor cantidad de hits en cache.

Capítulo 3

3. Marco Teórico

3.1. Aplicación web de gran escala

A las aplicaciones web que manejan un elevado número de usuarios y un gran volumen de transacciones de datos por segundos, se les denomina aplicaciones web de gran escala. En estos sistemas predominan las arquitecturas distribuidas las que se caracterizan por tener una alta disponibilidad y escalabilidad (Díaz Ramos & Orjuela Castillo, 2014). Esto permite soportar el gran número de usuarios concurrentes, compartir grandes cantidades de recursos, tener una mayor tolerancia a fallas, entre otros.

Las aplicaciones web de gran escala están conformadas por un *front-end* (parte del software que interactúa con el usuario), un *back-end* (tecnología que se encuentra por el lado del servidor), y además, pueden contar con capas intermedias, como por ejemplo una capa de aplicación en donde se puede utilizar contenido web dinámico (ASP, PHP, etc.).

Algunas aplicaciones web a gran escala se pueden encontrar en motores de búsqueda (Google, Yahoo!, etc.) y en las redes sociales (Facebook, Twitter, etc.).

3.2. Motores de búsqueda web

Los motores de búsqueda web (WSE) son sistemas grandes y complejos encargados de entregar un resultado a las peticiones de los usuarios. Están compuestos de una gran cantidad de servidores interconectados a través de diferentes redes, localizados generalmente en múltiples *data centers* (Skobeltsyn, Junqueira, Plachouras, & Baeza-Yates, 2008) ubicados en zonas geográficas específicas. Los principales componentes de los WSE son: Servicio de Front-End (FS), Servicio de Cache (CS) y Servicio de Índice (IS) (Gómez Pantoja, 2014).

Algunos de los motores de búsqueda más conocidos son:

- **Google**

  Google es una empresa que provee productos y servicios relacionados con Internet, software, etc. Los principales servicios proporcionados por Google son: Gmail, Google Maps, Google Search, Google+, Google Drive, entre otros. Google Search, es el motor de búsqueda web más visitado en la actualidad, siendo en febrero del año 2016 el buscador más utilizado en Estados Unidos con 11,2 billones de consultas (Lella, 2016).

  Google debe manejar más de 100 billones de búsquedas al mes, más de 3 billones de búsqueda por día y debe rastrear más de 20 billones de páginas al día para mantener actualizados los constantes cambios en la web (Biswas, 2013).

- **Yahoo!**

  Yahoo! es una empresa que posee múltiples servicios. Entre ellos se puede encontrar: AskYahoo, blogs, Yahoo! Answers, Yahoo! Search, entre otros. Yahoo! Search es uno de los motores de búsqueda web que le sigue en popularidad a Google, siendo en febrero del año 2016 el tercer buscador más utilizado en Estados Unidos con 2,2 billones de consultas realizadas por medio de este buscador (Lella, 2016).

De lo anterior, se observa que ambos buscadores deben manejar mensualmente una alta cantidad de usuarios y un gran volumen de intercambio de información para responder las consultas de los usuarios de forma rápida, precisa y con un contenido actualizado. Los motores de búsqueda web necesitan tener una arquitectura robusta, que utilice los algoritmos y las políticas necesarias que permitan el acceso a la información de manera rápida y eficiente.

3.3. Redes sociales

Las redes sociales son sitios basados en servicios que permiten a los usuarios crear un perfil público, definir una lista de usuarios con los que comparte una conexión, visualizar y recorrer su lista de conexiones y las hechas por otros (Bhattacharya, 2014). Es una plataforma para construir relaciones con personas que, por ejemplo, compartan mismos intereses, actividades, antecedente o conexiones de la vida real. Entre las redes sociales más conocidas se encuentran:

- **Facebook**

  Facebook es fundado el año 2004, en un inicio es creado para que los estudiantes Harvard se pudieran comunicar y compartir información entre ellos. Se hizo tan popular que al poco tiempo se extendió más allá del ambiente universitario, alcanzando en diciembre del año 2015, 1,59 billones de usuarios activos al mes, que incrementa un 17% año a año (Facebook, 2016).

  Facebook debe soportar diariamente 1,04 billones de usuarios activos (Facebook, 2016), los que comparten mensajes, imágenes, videos, etc.

- **Twitter**

  Twitter es una red social que marca la tendencia de servicios en tiempo real, donde los usuarios pueden comunicar al mundo lo que están haciendo o lo que está en sus mentes en un límite de 140 caracteres (Bhattacharya, 2014) (*tweets*). Twitter soporta mensualmente 320 millones de usuarios activos (Twitter, 2016) los que diariamente publican mensajes, se suscriben a los tweets de otros usuarios (*follow*), re-envían mensajes de otros usuarios (*retweet*), etc.

Las redes sociales como Facebook y Twitter deben tener una infraestructura capaz de permitir comunicaciones en tiempo real, agregar contenido de múltiples fuentes, acceder y actualizar los contenidos más populares y compartidos, y ser capaz de procesar millones de solicitudes de usuarios por segundo (Nishtala, y otros, 2013).

3.4. Cache

Las aplicaciones web de gran escala deben ser capaces de soportar una gran cantidad de usuarios y un gran volumen de transacciones de forma diaria (millones/billones), en que las peticiones que se han realizado recientemente tienen una alta probabilidad de ser requeridas nuevamente en un futuro cercano. Una de las tecnologías que permite disminuir el continuo acceso a los servidores producido por la búsqueda de esta información redundante solicitada por los usuarios es el *Web Caching*. Esta tecnología permite almacenar en cache respuestas pre-computadas de los objetos solicitados por los usuarios.

Según Wang (1999) algunas de las ventajas del Web Caching son:

1. Reducir el consumo de ancho de banda, lo que permite disminuir la congestión y el tráfico en la red.

2. Reducir la latencia de acceso a los documentos, esto se debe a que los objetos que son accedidos con mayor frecuencia se encuentran almacenados en un nodo de cache cercano, lo que disminuye el tráfico en la red, permitiendo que los objetos que no están en cache puedan también ser recuperados con mayor rapidez.

3. Reducir la carga de trabajo en los servidores web.

Existen 3 clases de cache en la web: (i) cache en el cliente, que son integrados en los navegadores y ayudan a mejorar la percepción del usuario en cuanto a eficiencia en ese cliente; (ii) el cache en los servidores, ubicados en los servidores web los que ayudan a mejorar los tiempos de respuesta; y (iii) los proxy cache, ubicado en una localidad intermedia y son compartidos por un

grupo de usuarios que mantienen algún tipo de relación (ubicados en la misma región, país, empresa, etc..).

Según Wang (1999), entre las propiedades deseables que tenga el sistema de *caching* se encuentran: rapidez en el acceso (reducir la latencia), robustez (capaz de recuperarse en caso de fallos), transparente a la vista del usuario, escalable, eficiente, adaptable, estable, contar con técnicas para el balance de carga (tal que no se formen cuellos de botella en algún nodo), hacer frente a la heterogeneidad (en cuanto a arquitecturas de red) y simple (de implementar).

Uno de los objetivos más importantes en el cache es alcanzar una alta tasa de hit, es decir que una alta cantidad de las peticiones que acceden al cache tenga una respuesta pre-computada y no deba entrar a otro tipo de servicio a buscar una respuesta.

En los WSE de gran escala, el cache juega un papel importante a la hora de realizar las búsquedas a las peticiones de usuarios. Esto se debe a que sin la existencia del cache, el motor de búsqueda procesa un gran volumen de datos recorriendo los miles de servidores interconectados en diferentes redes y en múltiples data center. Para esto se utiliza el cache o servicio cache que permite obtener una respuesta rápida, incrementar el *throughput* de la consulta y reducir la carga del IS (Skobeltsyn, Junqueira, Plachouras, & Baeza-Yates, 2008; Gómez Pantoja, 2014; Cambazoglu, y otros, 2010; Gan & Suel, Abril 2009).

En los WSE el cache es utilizado para almacenar respuestas pre-computadas de peticiones previamente realizadas por los usuarios (Skobeltsyn, Junqueira, Plachouras, & Baeza-Yates, 2008; Gómez Pantoja, 2014; Cambazoglu, y otros, 2010). Su finalidad es obtener la respuesta a las

peticiones hechas por los usuarios de manera rápida, reduciendo al mínimo el acceso a los cientos/miles de servidores que se encuentren en el back-end. A grandes rasgos, el proceso que se lleva a cabo cuando un usuario realiza una petición es el siguiente: (i) la consulta es buscada en el cache, (ii) si la respuesta se encuentra en el cache (hit en cache), ésta es enviada al servicio de front-end (FS) donde es mostrada al usuario, (iii) en caso de miss en cache el FS debe ir a buscar la respuesta al servicio de índice (IS).

3.5. Organización del cache

El cache se encuentra en múltiples nodos de los *clusters* disponibles para este servicio. Estos pueden aceptar un límite de peticiones por segundo. El espacio de memoria que se encuentra en cache es limitado. Cada entrada del cache es un par <*query*, *answer*>, *query* corresponde a la consulta realizada por el usuario y *answer* a su respuesta.

3.5.1. Políticas para cache

Dado que el cache tiene un espacio limitado, las entradas deben ser constantemente modificadas, dando paso al ingreso de nuevas entradas que puedan ocasionar un hit en cache en un futuro cercano. Para esto existen distintas políticas que permiten admitir, reemplazar y eliminar las entradas del cache.

3.5.1.1. Políticas de desalojo o algoritmos de reemplazo

Las políticas de desalojo deben permitir eliminar una entrada existente del cache cuando ha alcanzado su límite de espacio. La idea es descartar la entrada que probablemente no logre ser un hit en cache (Baeza-Yates, Junqueira, Plachouras, & Witschel, 2007).

Según Podlipnig y Böszörmenyi (2003) los objetos web tienen distintos factores que pueden influenciar en el proceso de reemplazo. Estos son la frecuencia en que se les hace referencia, el tiempo desde la última referencia, el tamaño, el costo de ir a buscar un objeto, el tiempo desde su última modificación y el tiempo de expiración (heurística).

Las políticas de desalojo más conocidas son *Least Recentently Used* (LRU) y *Least Frequently Used* (LFU) (Gómez Pantoja, 2014; Cambazoglu, y otros, 2010; Gan & Suel, Abril 2009; Baeza-Yates, Junqueira, Plachouras, & Witschel, 2007). Mientras LRU desaloja la entrada que fue menos usada recientemente, LFU desaloja la entrada menos frecuentemente usada.

3.5.1.2. Políticas de admisión

Las políticas de admisión deben permitir que no sean ingresadas al cache aquellas entradas que no serán frecuentadas en un futuro cercano, puesto que utilizarán un espacio de memoria sin generar un hit en cache (Baeza-Yates, Junqueira, Plachouras, & Witschel, 2007), perjudicando el rendimiento de ese nodo. Esto es, cuando el cache no tiene espacio para almacenar una nueva consulta, se activan las políticas de desalojo correspondientes para eliminar una entrada del cache e ingresar la consulta entrante. Como esta nueva entrada puede no generar hits, se deben aplicar políticas de admisión que decidan si la nueva entrada es almacenada en cache, o se conservará en cache la entrada propuesta a ser eliminada por la política de reemplazo.

3.5.1.3. Eliminación de resultados antiguos

Las políticas de eliminación de resultados antiguos deben permitir mantener actualizadas las respuestas de las entradas del cache, tal que no exista el problema de *cache invalidation* debido a que el índice de alguna

página haya sido modificado. Esto se debe a que la web se encuentra en constante cambio (Cambazoglu, y otros, 2010; Blanco, Roi, Junqueira, Lempel, Telloli, & Zaragoza, Julio 2010). Una de las soluciones propuesta para la eliminación de resultados antiguos es la de *Time-To-Live* (TTL), la cual asocia a cada entrada del cache un tiempo de vida para que la respuesta de la entrada sea válida (Cambazoglu, y otros, 2010; Blanco, Roi, Junqueira, Lempel, Telloli, & Zaragoza, Julio 2010; Alici, Sengor Altingovde, Ozcan, Cambazoglu, & Ulusoy, Julio 2011). Una vez que expire su tiempo de vida, los resultados vuelven a ser computados para que se mantengan actualizados (Gómez Pantoja, 2014).

3.6. Estructura del cache

Es posible clasificar el cache en dos categorías: cache estático y cache dinámico (Ozcan R. , Altingovde, Cambazoglu, Junqueira, & Ulusoy, 2012).

3.6.1. Cache estático

El cache estático almacena contenido que no cambia frecuentemente, se basa en la información histórica y es actualizada de manera periódica.

Según Ozcan, Altingovde y Ulusoy (2008), el cache estático en los WSE almacena en cache las consultas más populares, esto se logra analizando los registros históricos de las peticiones realizadas por los usuarios. Una vez obtenidas las consultas más populares son almacenadas en cache, y deberá ser actualizado periódicamente. Algunas de las características que deben tener las consultas (proveniente de los registros) para ser ingresadas al cache estático son: la frecuencia en que es solicitada la consulta, la cantidad de usuarios que han hecho la misma consulta, las URL de resultados de consultas

con mayor cantidad de clicks hechos por los usuarios y la estabilidad en la que permanece la frecuencia de la consulta en el tiempo.

3.6.2. Cache dinámico

En el cache dinámico se trata de almacenar los datos que probablemente van a ser solicitados en un futuro cercano. Al tener que lidiar con contenido dinámico, las entradas del cache deben ser frecuentemente actualizadas en comparación con los del cache estático. En los WSE el reto del cache dinámico está en utilizar políticas de desalojo efectivas para mantener las entradas más recientes (Ozcan R. , Altingovde, Cambazoglu, Junqueira, & Ulusoy, 2012).

3.7. Clasificación de cache según el almacenamiento

Según Baeza-Yates R., Gionis, Junqueira, Murdock, Plachouras y Silvestri (2007), en los WSE existen dos formas de utilizar la memoria cache: una forma es **almacenando las respuestas** de consultas que pueden ser realizadas en el futuro; la segunda forma es **almacenando términos**. Es decir cuando el motor analiza una consulta en particular puede decir si guarda en memoria una *posting list* de los términos de la consulta. Si el conjunto de *posting list* es mayor a la memoria que posee el cache, entonces selecciona una pequeña cantidad que es mantenida en memoria.

Según Zhou, Li, & Xinhua (2015), en los motores de búsqueda web los cache de un nivel pueden ser clasificado en: cache de resultado, cache de puntuación (*score cache*), cache de intersección, cache de proyección, cache de listas, *cache snippet* y cache de documentos.

Para cada uno de los caches nombrados anteriormente es posible aplicar distintas políticas y algoritmos que permitan mejorar su rendimiento. Además, existen distintos trabajos que complementan estos cache generando distintas arquitecturas que permitan mejorar el rendimiento. A continuación se detallarán el cache de resultado, cache de listas y el cache de intersección.

3.7.1. Cache de resultado (RC - *Result cache*)

A medida que el motor de búsqueda web retorna el resultado de una consulta en particular, este decide si almacenar o no el resultado en el cache para resolver futuras peticiones. El almacenamiento de respuestas a las consultas previamente realizadas en el cache permite responder de manera más eficiente las peticiones realizadas por los usuarios, ya que no se buscará una respuesta al servicio de listas invertidas. El cache de resultados debe ser actualizado periódicamente (Baeza-Yates & Jonassen, Modeling static caching in web search engines, 2012). El manejo de RC es sencillo, pero a medida que aumenta el volumen de los datos disminuye muy rápidamente su tasa de hits (Zhou, Li, & Xinhua, 2015).

3.7.2. Cache de listas (PLC - *Posting list cache*)

Es una lista de resultados por término. Consiste en un término t y $n \geq 1$ *postings*, cada una contiene el *id* de un documento donde t es referenciado, y alguna otra información que sea requerida para la función de puntuación (*score*) del motor de búsqueda (p.ej.: la frecuencia de t en cada documento). Además, puede reducir la cantidad de disco I/O (*input/output*) en el procesamiento de las consultas, proporciona una mayor utilización del cache obteniendo una tasa de hit mayor al cache de resultado y puede combinar términos para responder a consultas entrantes (Petersen, Simonsen, & Lioma, 2015). El manejo de PLC es

complicado y necesita una cantidad considerable de cálculos para retornar el resultado de las consultas.

Según Baeza-Yates R., Gionis, Junqueira, Murdock, Plachouras y Silvestri (2007), el almacenamiento en cache de las *posting lists* tiene desafíos adicionales. Dado que poseen distintos tamaños (el tamaño de su distribución sigue la ley de Zipf), almacenar estas listas en cache de forma dinámica no es muy eficiente. Esto se debe a la complejidad en términos de eficiencia y espacio, y a la desigual distribución del flujo de la consulta. El almacenamiento estático de las *posting lists* plantea aún más desafíos a la hora de decidir qué términos se deben almacenar, si los términos que son frecuentemente consultados o los términos con pequeñas *posting lists* que ahorran espacio. Por último, se debe adoptar una política para el cache estático que analicen que las características no cambien rápidamente en el tiempo.

3.7.3. Cache de intersección (IC - *Intersection cache*)

El cache de intersección es la intersección de una lista de resultado de varios términos que aparecen juntos en una consulta. Mejora el rendimiento en la recuperación y puede ser complementado con el RC y el PLC. La combinación de términos puede ser muy grande para ser mantenido en memoria, tal que debe consumir gran cantidad de espacio en disco. Esto hace que la elección de datos de intersección que deben ser almacenados en cache sea un proceso complejo.

Un ejemplo del funcionamiento del cache de intersección se puede ver mencionado en Long y Suel (2006) es el siguiente: se tienen los términos "hola" y "mundo", donde cada lista invertida posee en primer lugar el *id* del documento, mientras que los otros elementos corresponden a información

adicional como la frecuencia del documento (<*id*,r_1,$r_{2,...}$>). Entonces se tiene que el término "hola" tiene las siguientes *posting lists*:

hola :{ <2,2,4,6>,<3,1,7>,<7,2,1,6>}

Por otra parte, el término "mundo" posee las siguientes *posting lists*:

mundo : {<2,1,7>,<5,2,1,4><7,1,11>}

Al ser el primer elemento la *id* del documento este será el que define la intersección de los *posting lists* entre ambos términos. Esto quiere decir que, el término "hola" se encuentra en los documento con *id* 2, 3 y 7; en cambio el término "mundo" se encuentra en los documentos con *id* 2, 5 y 7. Por lo tanto, la intersección de ambos términos es la siguiente:

hola ∩ mundo : {<2;2,4,5;1,7>,<7;2,1,6;1,11>}

3.8. Comportamiento de usuario

3.8.1. Ley de Zipf

El 6% de todas las palabras que una persona dice, escribe y lee (en el lenguaje de habla inglesa) corresponde al término "the", la segunda palabra más usada corresponde a la palabra "of" la cual es utilizada la mitad de las veces que "the" (3%), la tercer palabra corresponde a la palabra "and" que es usada un tercio de las veces que la palabra "the" (2%), la cuarta palabra más usada es "to" la cual aparece un cuarto de las veces que el término "the" (1,5%) y así sucesivamente. El comportamiento que tienen estas palabras siguen la formula:

$$\frac{1}{n}$$

Donde *n* corresponde a la *n-ésima* palabra ordenada por frecuencia decreciente. Este fenómeno es conocido como la Ley de Zipf, ley empírica no solo aplicable para el habla inglesa si no que para todos las lenguas. No solo se ve en palabras si no que también en diversos escenarios, por ejemplo: en la intensidad de las llamaradas solares, en la magnitud de los terremotos, en la ocurrencia de nombres, en el número de visitas por página web (Newman, 2005), etc.

En la figura 3.1 se observan gráficos de distintos escenarios. Según la aparición de un suceso (palabra, acontecimiento...) y la cantidad de ocurrencias de este, se observa que siendo escenarios tan diferentes unos de otros siguen la ley de Zipf.

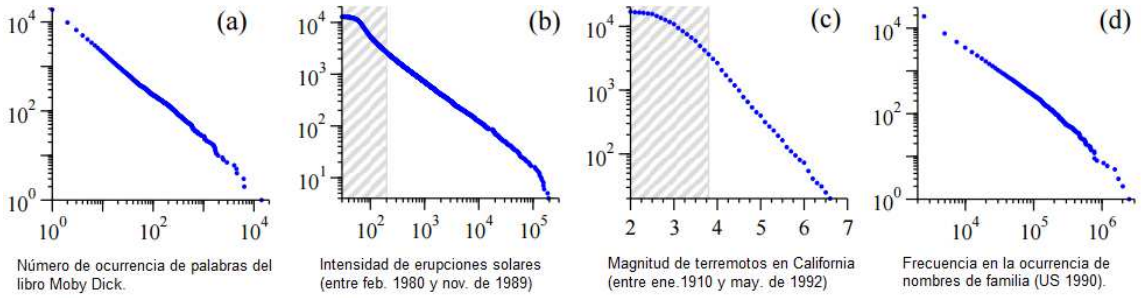

Figura 3.1: Gráficos rank v/s frecuencia de cuatro escenarios que siguen la ley de Zipf. Fuente: (Newman, 2005)

George Kingsley Zipf formuló esta ley empírica sobre libros de texto escritos en inglés, que modela la frecuencia de las palabras en un texto. En 1936, M. Joos modificó la fórmula de una forma más generalizada (Shi, Gu, Wei, & Shi, 2006):

$$P_n = \frac{c}{n^{\alpha}}$$

Donde *C* es una constante mayor que 0, *n* corresponde a la *n-ésima* palabra ordenada por frecuencia decreciente *n* = 1,... *N*, 0 < α ≤ 0 es el exponente característico de la distribución de palabras.

Diferentes estudios han demostrado que la ley de Zipf es aplicable al comportamiento del usuario en la web, y puede ser usado para describir la popularidad de los objetos web (Shi, Gu, Wei, & Shi, 2006). Esto es importante, puesto que puede ser usado para optimizar distintas aplicaciones en las que el usuario participe directamente. Sin ir más lejos la ley de Zipf podría ser utilizada junto a políticas de admisión del servicio cache para establecer que entrada permanece en cache y cual no, según la frecuencia de utilización de esa frase o palabra de la entrada con respecto a otra.

3.8.2. Tipos de peticiones de usuario

La cantidad de peticiones efectuadas por los usuarios en las aplicaciones web de gran escala varía en función de distintos factores, tales como el horario, los eventos sociales, la economía, el deporte, los desastres naturales, la temporada, entre otros. Tomando en cuenta el comportamiento del usuario, el interés que manifiesta por distintos tópicos puede cambiar o mantenerse en el tiempo, el que puede ser medido obteniendo la frecuencia en que se realiza una petición a un tópico en particular. Las diferentes frecuencias que se obtienen de

los tópicos consultados por los usuarios permiten clasificar los distintos tipos de consultas. Las clasificaciones de consultas más conocidas corresponden a las consultas periódicas, las consultas en ráfaga y las consultas permanentes, las que serán detalladas a continuación.

3.8.2.1. Consultas periódicas

Las consultas periódicas son aquellas que aparecen siempre en la misma fecha (p.ej.: navidad, fiestas patrias, etc.) para luego decaer y no volver a aparecer hasta su próximo periodo. Al tener en conocimiento las consultas que solo se realizan en ciertos períodos de tiempo permite integrarlas con antelación a la fecha del suceso en el cache, sin la necesidad de aplicar algún tipo de algoritmo para su admisión. En la Figura 3.2 se ve un ejemplo de una consulta periódica correspondiente a "navidad". Se observa que todos los años en las fechas de este acontecimiento crece abruptamente la frecuencia en que se realiza esta consulta.

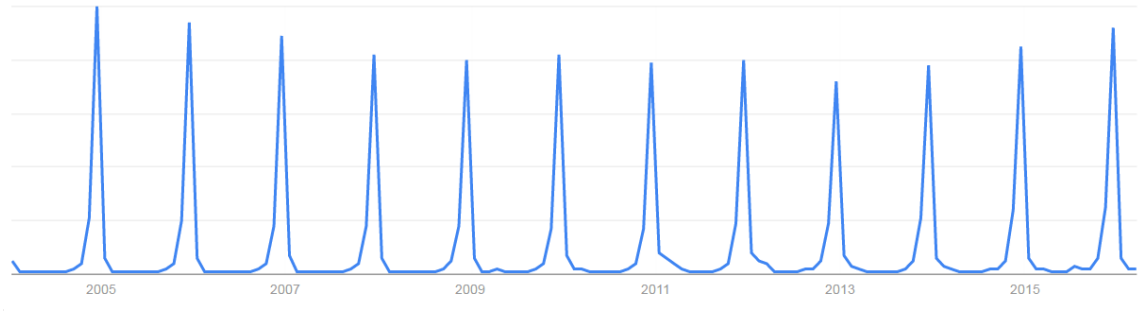

Figura 3.2: Frecuencia de consulta periódica: "navidad". Fuente: Google Trends.

3.8.2.2. Consultas en ráfaga

Las consultas en ráfaga son aquellas en que su comportamiento habitual son prácticamente nulas, para luego incrementar abruptamente debido a un suceso inesperado que llame la atención del público en general (p.ej.: la muerte de un famoso). Una vez que alcanza un pick va disminuyendo a medida que pierde el interés de los usuarios. En la Figura 3.3 se observa una consulta en ráfaga correspondiente a la "muerte de David Bowie", la que es altamente referenciada el día 10 de enero decayendo en su frecuencia el día 13 de enero.

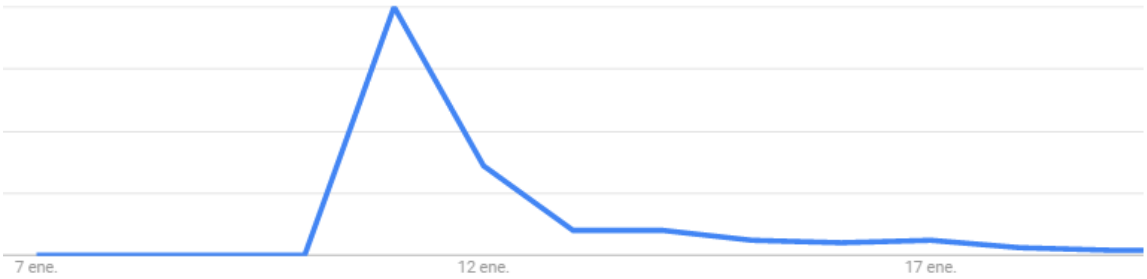

Figura 3.3: Frecuencia de consulta en ráfaga: "muerte de David Bowie". Fuente: Google Trends.

Las consultas en ráfaga no solo pueden corresponder a una sola consulta, sino a un grupo de consultas que tienen relación con ese mismo tópico. Por ejemplo, para la consulta en ráfaga "muerte de David Bowie", también en ese intervalo de tiempo fueron consultas en ráfaga: "David Bowie", "biografía de David Bowie", "música de David Bowie", entre otras consultas relacionadas con el mismo tópico.

A nivel de cache, uno de los problemas que ocasiona este tipo de consultas al aumentar el interés de los usuarios con respecto a un tópico en particular de forma abrupta, es la posibilidad de generar un desbalance en los

nodos que contienen la respuesta de dicha consulta. Esto se debe a que el volumen de peticiones que reciben los nodos es mayor a la cantidad máxima que puede soportar. Una posible solución es que al detectar este tipo de consultas sea ingresada al cache sin tener que pasar por alguna política de admisión siendo distribuida en varios nodos, disminuyendo el ingreso al back-end y el desbalance de los nodos.

Un mecanismo para la detección temprana de consultas en ráfaga es el propuesto por Gómez Pantoja (2014), basado en: el promedio móvil (si la frecuencia está por sobre el promedio móvil de la serie se considera que es un alza tipo ráfaga de la frecuencia), la desviación estándar móvil (una alta desviación estándar indica una alta dispersión de los datos, que permite detectar series de tiempo anómalas), y en el coeficiente de variación móvil (su valor es alto al tratarse de una ráfaga).

3.8.3. Consultas permanentes

Las consultas permanentes, corresponden a aquellas peticiones que son realizadas constantemente por los usuarios (p.ej.: Facebook, Google, etc.). Es decir, son las consultas que se hacen de manera frecuente trascendiendo en el tiempo y que representan estacionalidad. En la Figura 3.4 se observa un ejemplo de consulta permanente correspondiente a "facebook" donde constantemente es buscada por los usuarios.

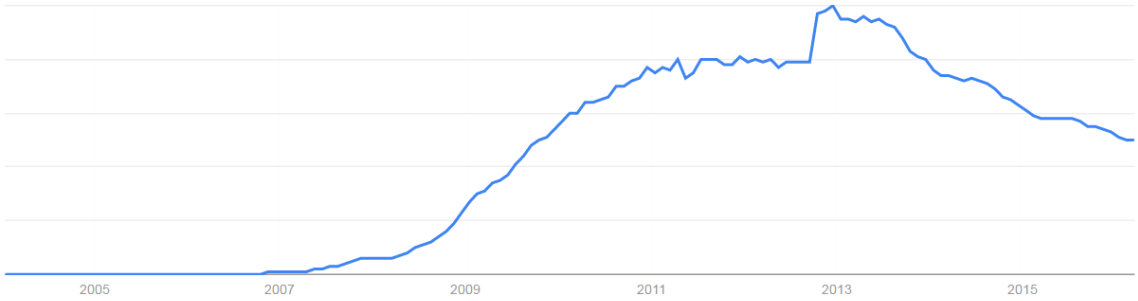

Figura 3.4: Frecuencia de consulta permanente: "facebook". Fuente: Google Trends.

A nivel de cache, trabajos como los realizados por Zhang, He, y Ye (2013); Baeza-Yates, y otros (2007); Markatos (2001); Ozcan R., Altingovde, Cambazoglu, Junqueira, y Ulusoy (2012); y Kumar y Norris (2008) han demostrado que dedicar un espacio estático en cache para el almacenamiento de consultas permanentes mejora el rendimiento del cache aumentando la tasa de hit que se obtiene en el caso de no ser almacenadas. Para obtener las consultas permanentes que van a ser ingresadas al cache estático generalmente se procede a revisar los registros históricos de la aplicación web, utilizando métodos que determinan que consultas son permanentes y que dependen de la variación en la frecuencia de la consulta.

Capítulo 4

4. Estado del Arte

4.1. Arquitectura de Cache

Las aplicaciones web a gran escala cuentan con una gran cantidad de personas que interactúan a diario con el sistema, mientras más usuarios tiene el sistema mayor es la probabilidad de que se repitan las peticiones realizadas con anterioridad. La existencia de una arquitectura cache capaz de almacenar las respuestas a las peticiones de usuario que puedan ser solicitadas en un futuro cercano, permite disminuir el ingreso innecesario a otras capas del sistema que generan distintos costos de procesamiento de la consulta, tales como, costos de tiempo, costos de utilización de ancho de banda, entre otros.

A continuación, se darán a conocer en forma general algunas arquitecturas de *Caching*:

Arquitectura de cache jerárquico: Las arquitecturas de cache jerárquica se puede asimilar a una estructura de árbol en donde los caches son clasificados en cache padres y cache hijos. Los padres no consultan a sus hijos aunque pueden ser solicitados por ellos. Según Rodriguez, Spanner y Biersack (2001), en estas arquitecturas el cache se sitúa en múltiples niveles de la red. Por ejemplo, hay cuatro niveles de cache: el inferior (conectado directamente al cliente), el institucional, el regional y el nacional. Cuando una petición no es solucionada por el cache del cliente, la petición es redirigida al cache

institucional, si no es encontrado ahí es enviada al cache regional y si aún así no es encontrada es destinada al cache nacional. En el caso que el documento no se encuentre en este cache es enviado directamente al servidor original. Cuando el documento es encontrado se desplaza hacia abajo en la jerarquía y deja una copia en cada uno de los nodos intermedios. Además, las solicitudes realizadas para el mismo documento van recorriendo hacia arriba la jerarquía del cache hasta encontrar el documento. Algunos problemas de esta arquitectura son: (i) cada jerarquía produce retrasos adicionales, (ii) los cache ubicados en niveles más altos pueden convertirse en cuellos de botella y tener gran retraso, (iii) varias copias del mismo documento son almacenadas en diferentes niveles del cache. En la figura 4.1 se observa la topología que puede seguir una arquitectura de cache jerárquico.

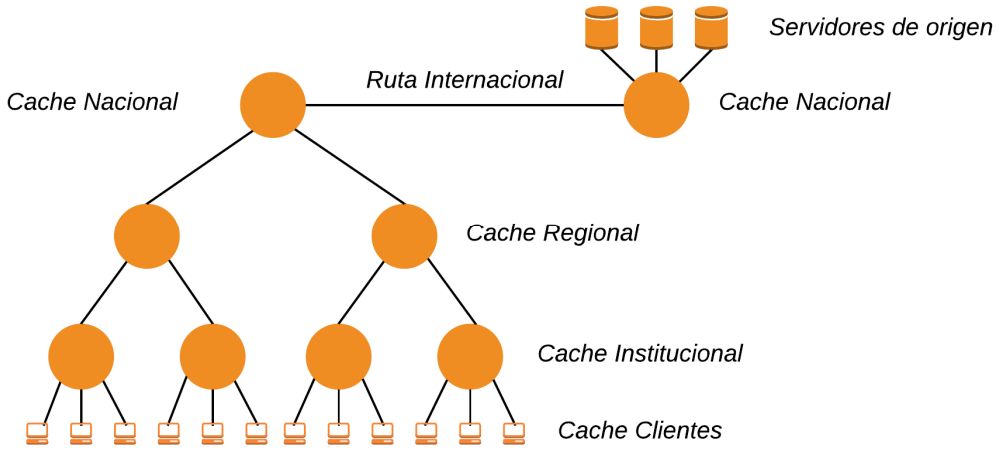

Figura 4.1: Topología de cache jerárquico. Fuente: (Sosa Sosa & Navarro Moldes, 2002).

Arquitectura de cache distribuido: En los caches distribuidos no hay caches intermedios solo hay caches institucionales. Todos estos caches están formados por metadatos que contiene información de los otros nodos del cache, permitiendo así ir a buscar el documento a otro nodo que la pueda contener, en el caso de existir un *miss* en cache. La arquitectura de cache jerárquico puede ser utilizada en conjunto para obtener un enfoque escalable para la dispersión

de los metadatos, solo siendo utilizada en los directorios en los que el contenido esté disponible. Algunas ventajas que tiene el cache distribuido sobre el cache jerárquico son: (i) el espacio en disco usado en las arquitecturas distribuidas es menor que el usado en arquitecturas jerárquicas, (ii) tiene un mayor grado de tolerancia a fallas y un mayor balance de carga (Nagaraj, 2004). En la figura 4.2 se observa la topología que puede seguir una arquitectura de cache distribuido.

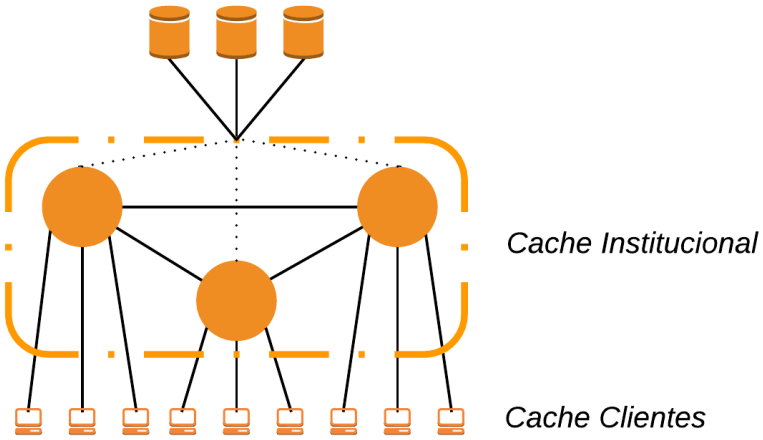

Figura 4.2: Topología de cache distribuido. Fuente: (Sosa Sosa & Navarro Moldes, 2002).

Arquitectura de cache híbrida: Los cache híbridos son una combinación del cache jerárquico y el cache distribuido. Los nodos cache pueden cooperar con otros caches que se encuentren en el mismo nivel de jerarquía o en un nivel superior a través del cache distribuido. En la figura 4.3 se observa la topología que puede seguir una arquitectura de cache hibrido.

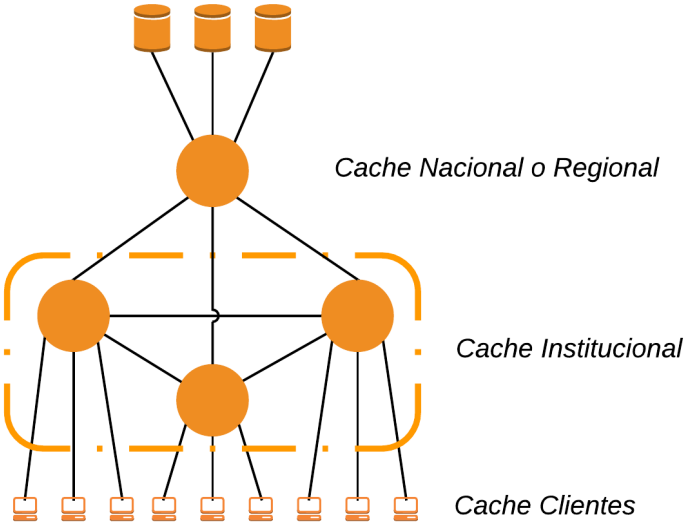

Figura 4.3: Topología de cache híbrido. Fuente: (Sosa Sosa & Navarro Moldes, 2002).

Para las arquitecturas nombradas anteriormente, existen distintos enfoques proporcionado por un sin número de autores que han buscado nuevas formas para mejorar el rendimiento del cache como tal, disminuir la latencia generada al ir a buscar peticiones en la web, aumentar la escalabilidad, mejorar el balance de carga, entre otros.

A continuación se da a conocer algunos de los trabajos realizados de arquitecturas cache que buscan mejorar su rendimiento.

4.1.1. Two level caching technique for improving result ranking

En Singh, Hussain y Ranjan (2011) proponen un esquema que reduce el cómputo y los requerimientos de entrada/salida, mejorando la escalabilidad de los motores de búsqueda web sin alterar las características del ranking. Es un esquema cache de dos niveles que combina de forma simultánea el cache de resultados y el cache de listas invertidas.

En el cache de resultados la estrategia que utilizan es de mantener en el cache una lista de documentos asociados a la consulta, en el que para cada documento se almacena la URL, el título y 250 caracteres. La cantidad máxima de documentos que se asocia a cada petición es de 50 referencias (25 kilobytes de resultado por consulta en cache).

En el cache de listas invertidas la estrategia que utilizan es de mantener en memoria una lista de documentos web asociados a un término de la consulta. Al llegar una petición con el mismo término, ésta puede ser resuelta con la lista previamente almacenada (la lista es almacenada en un buffer especial o generalmente en una memoria secundaria), y así evitar el acceso al disco. Además, introducen una técnica para el almacenamiento de las listas invertidas, en el caso que sean de gran tamaño ya que podría producir fallas en este cache. Esta técnica consiste en ordenar la lista invertida según la frecuencia en que aparece el término en cada documento, tal que al procesar la consulta se usan solo los documentos cuyos términos tengan la mayor frecuencia. Es decir, la lista no es recorrida completamente o no es recorrida en absoluto dependiendo de la relevancia del término. También, dividen las listas en bloques de documentos donde se encuentran los términos con la misma frecuencia. El primer bloque de cada lista es pequeño y corresponde a los documentos a lo que se accede de manera frecuente. Al final de la lista se encuentran los bloques que contienen los documentos donde el término aparece un par de veces. La distribución de tamaño de los bloques sigue varias órdenes de magnitud, que hace el almacenamiento mucho más complejo.

La política de reemplazo que utilizan en el caso que el cache alcance su límite de entradas corresponde a LRU. Esto se debe a que el motor de búsqueda web (TodoBR utilizado para las simulaciones) tiene buena localidad temporal.

El esquema de *Two level cache* propuesto (ver figura 4.4) combina el cache de resultado con el cache de listas invertidas, donde cada vez que se realiza una petición primero se ve si la respuesta se encuentra en el cache de resultados. Si es un hit en cache la consulta es respondida inmediatamente, sino la petición es procesada en el cache de listas invertidas, reduciendo así el número de acceso al disco. Si se produce un miss, accede al disco a buscar la respuesta.

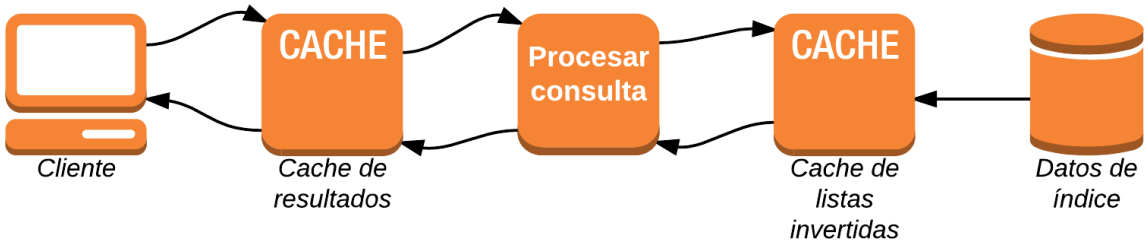

Figura 4.4: Esquema de dos niveles. Fuente: (Singh, Hussain, & Ranjan, 2011).

Los experimentos realizados en los *logs* obtenidos del motor de búsqueda web "TodoBR" muestran que el *throughput* del esquema cache de dos niveles es un 52% más alto que la implementación de solo listas invertidas y un 36% más alto si solo se utilizará el cache de resultados.

4.1.2. A two-level cache for distributed information retrieval in search engines

En Zhang, He y Ye (2013) proponen una estructura de dos niveles basada en los registros de consultas de usuario, los que son integrados a un sistema de cache distribuido. La estructura cache propuesta consiste en lo siguiente: cada *clúster* de cada servidor cache adopta una estructura de dos niveles, en que cada cache se compone de un cache estático y un cache dinámico. El cache estático almacena los pares petición/respuesta más

frecuentes, los que son extraídas de los registros de consultas de usuario. Las entradas solo cambian si deben ser reemplazadas o actualizadas. El cache dinámico cambia según las consultas de usuario, almacenando los pares petición/respuesta más frecuentes y por medio de la utilización de algoritmos de reemplazo en el caso que sea necesario. Lo que hace esta estrategia es que cuando el usuario realiza una consulta entra primero al cache estático, si no se encuentra la respuesta (miss) pasa a buscar al cache dinámico.

La estructura y la estrategia utilizadas consiste en (ver figura 4.5) los siguientes pasos:

i. El módulo de operación de cache crea en cada servidor cache un cache estático y otro dinámico con un buffer de memoria.

ii. Utilizar la estrategia de distribución de datos (ver tabla 4.1). Esta estrategia consiste en que por medio del análisis de los registros de las consultas el sistema cache va a calcular el HotValue de cada consulta. Luego se abre el log de consulta, lee el contenido y extrae todos los ítems de las consultas. Primero se calcula la frecuencia (*Freq*) en que es referenciada la consulta, luego se calcula el intervalo de tiempo (IntervalTime), también se calcula el ciclo de vida de la petición (LiveTime), el tiempo desde la última vez que se realizó la consulta (NotActiveTime). Por último, realiza los cálculos HotValue para cada consulta y el sistema ordena el resultado de manera descendente.

	Fórmulas
Frecuencia de la consulta	$Freq = \frac{QueryNum}{24 \cdot 360}$ (QueryNum: número de consultas realizadas en el tiempo; 24 es la cantidad de horas en que corre el sistema)
Intervalo de tiempo	$IntervalTime = \frac{1}{Freq}$
Ciclo de vida de la consulta	$LiveTime = LastTime - FirsTime$ (LastTime: tiempo de la última ocurrencia de la consulta; FirstTime: tiempo de la primera ocurrencia de la consulta)
Tiempo desde que la consulta no ha sido referenciada	$NotActiveTime = CurrentTime - LastTime$ (CurrentTime: tiempo actual; LastTime: tiempo de la última consulta)
Valor del comportamiento de la consulta, es alto (hot) si actualmente es referenciada frecuentemente.	$HotValue = Freq \cdot LiveTime \cdot \frac{1}{NotActiveTime}$

Tabla 4.1: Fórmulas para la estrategia de distribución de datos. Fuente: (Zhang, He, & Ye, 2013).

iii. Inicializar el cache estático y dinámico. Una vez ordenados los valores HotValue, la inicialización de las consultas en el cache estáticos consta de dos partes. La primera es ingresar al cache las consultas nativas con su respectiva respuesta y segundo almacenar las peticiones con mayor valor (HotValue) y su respectiva respuesta de otros clúster. El cache dinámico se deja vacío al inicio.

iv. Comunicación entre el cache estático y el cache dinámico. Al llegar una consulta de usuario entra al cache estático. Si es un hit devuelve la respuesta al usuario, en caso de un miss accede al cache dinámico. En caso de un miss en ambos caches la petición es procesada en el clúster. Una vez

obtenida la respuesta desde los clúster se utilizan algoritmos de reemplazo para el intercambio de entradas en los caches.

v. Actualización de cache estático y dinámico. El sistema usa la estrategia de inicialización sincronizada de buffer para mantener actualizados los índices cada 24 horas. En cuanto los índices son actualizados, se eliminan ambos caches, y se utiliza la misma estrategia para el ingreso de datos al cache.

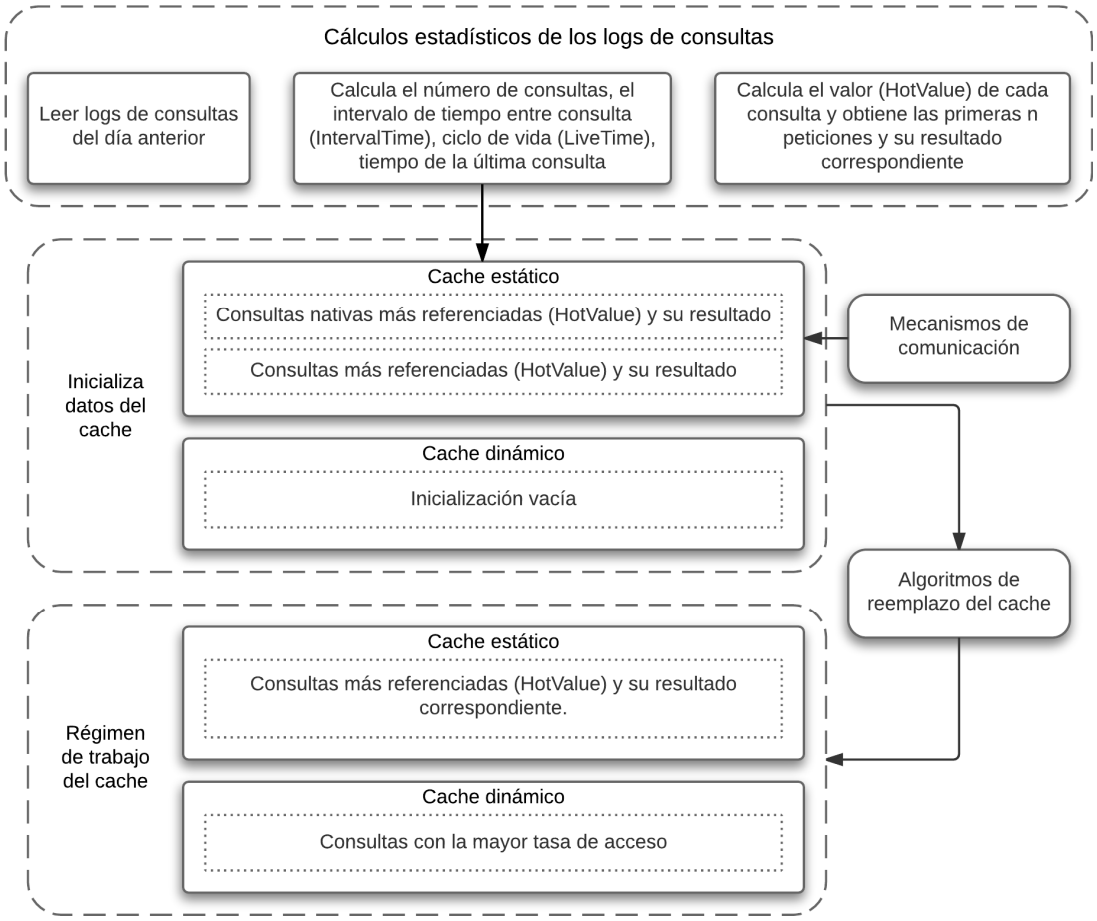

Figura 4.5: Distribución estrategia del cache estático y dinámico. Fuente: (Zhang, He, & Ye, 2013).

En las pruebas de rendimiento realizadas de la estrategia propuesta, se obtuvo que la estructura de dos niveles es un 28% más rápida y tiene una mayor tasa de hit que al usar solo una estructura de cache dinámico. Un problema que surge de esta estrategia es que los datos del cache estático son almacenados en un periodo de tiempo, pero puede ser que esos datos no sean referenciados al siguiente día. Esto se puede solucionar utilizando políticas de desalojo para el cache estático.

4.1.3. TLMCA - An intersection cache based on frequent itemset mining in large scale search engines

En Zhou, Li y Xinhua (2015), denotaron un problema en la baja velocidad de acceso I/O (Input/Output) al disco por lo que proponen una arquitectura cache de tres niveles llamada TLMCA. Esta estructura combina el cache de intersección (IC), el cache de resultado (RC) y el cache de listas (PLC). Además, introducen una nueva política de selección de datos del cache de intersección basada en la minería de los top-N elementos más frecuentes (*Frequent itemset mining* - FIMI), mientras que en el PLC y el RC adoptan políticas de reemplazo basadas en la frecuencia.

El diseño de la arquitectura TLMCA es la siguiente (ver figura 4.6): primero añadieron el cache de intersección estático en la base del cache de resultados y el cache de listas en la memoria. Los pasos a seguir cuando el buscador web recibe una consulta con n términos ($t_1, t_2, ..., t_n$) son los siguientes:

i. Los términos de la consulta son procesados en el cache de resultados.

ii. Los términos de la consulta son chequeados en el cache de intersección, donde el sistema extrae los k-conjuntos de elementos ($2 \leq k \leq m$, donde m es la máxima longitud del IC), ve si coincide con el cache de

intersección desde el más largo al más pequeño, y al producirse un hit en un elemento retorna de inmediato la lista de intersección ($I(t_1, t_2, t_3)$).

iii. Se revisa el PLC con los últimos términos restantes ($t_4, t_5,..., t_n$). El término *t4* de la lista de términos, es extraído del cache, y se buscarán las otras listas de términos ($PL(t_5)$,..., $PL(t_n)$) por medio del acceso al disco.

iv. El servicio de índice retorna la respuesta ($r_1, r_2,..., r_k$) al servidor web después de realizar las operaciones de intersección ($I(t_1, t_2, t_3) \cap PL(t_4) \cap PL(t_5) \cap ... \cap PL(t_n)$), el ranking de las páginas y la generación de *snippet*, y así sucesivamente. El servidor web recibe resultados parciales de cada servicio de índice.

Por último, se generan los nuevos top-*k* resultados de las consultas ($r_1, r_2,..., r_k$) y se forma la página de resultados final la que es devuelta al usuario.

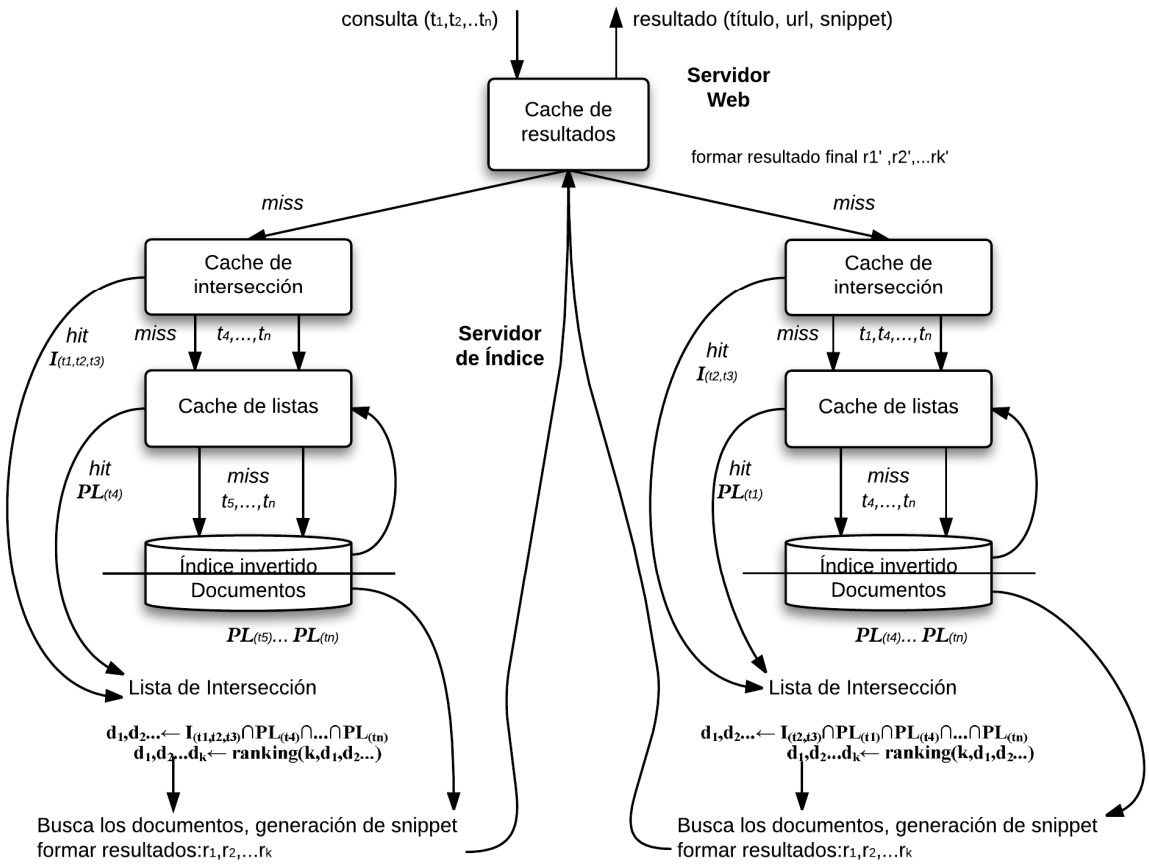

Figura 4.6: Sistema de arquitectura caching en servidores web. Fuente: (Zhou, Li, & Xinhua, 2015).

La estrategia de selección de datos del cache de intersección utiliza el algoritmo FP-Growth para extraer los top-*n* elementos más frecuentes. La estrategia de reemplazo del cache de intersección se basa en el incremento de la frecuencia de los elementos de la estructura Trie-Tree.

En las pruebas de rendimiento realizadas a la propuesta de TLMCA junto con las estrategias utilizadas en el cache de intersección, mejora un 27% el rendimiento en comparación de arquitecturas de dos niveles.

4.1.4. A five-level static cache architecture for web search engines

En Ozcan R., Altingovde, Cambazoglu, Junqueira y Ulusoy (2012), proponen una arquitectura multinivel de cache estático que almacena cinco tipos de ítems: cache de resultado, cache de puntuación, cache de listas invertidas, cache de intersección y cache de documentos. Cada cache almacena un par (*key*, *value*) y proporciona ahorro en los costos de proceso de las consultas. Además, proponen una heurística voraz capaz de priorizar los diferentes ítems que van a ser ingresados en el cache, tomando en cuenta la interdependencia entre el costo del proceso y la frecuencia de acceso de los ítems. La heurística voraz (algoritmo de almacenamiento en cache *cost-based mixed-order*) utilizada consta de 2 pasos:

Primer paso: calcula el valor de la posible ganancia inicial que se obtiene al almacenar el ítem en cache (usando estadísticas de los registros anteriores) y luego los ítems son insertados en una cola en que su prioridad se basa en este valor.

Segundo paso: los ítems con mayores ganancias son seleccionados y se llevan a cabo actualizaciones de las ganancias obtenidas en los ítems restantes. Esto se realiza de manera iterativa.

La figura 4.7 muestra el flujo de trabajo realizado por el simulador al procesar las consultas. A grandes rasgos la consulta es buscada en el cache de resultados, el cache de puntuación, el cache de documentos, el cache de intersección y el cache de listas, pero no ingresará al siguiente cache a menos que exista un miss en el cache anterior, con excepción del cache de puntuación donde una vez que la respuesta a la consulta es encontrada se debe acceder al cache de documentos para acceder a la página de resultados. También se observa en el flujo la existencia de 5 pasos, en que cada paso ayuda al

procesamiento de la consulta para generar la página de resultados final. A cada uno de estos pasos se le asignó un costo de I/O (Input/Output) para ayudar a medir el tiempo de procesamiento en cada cache.

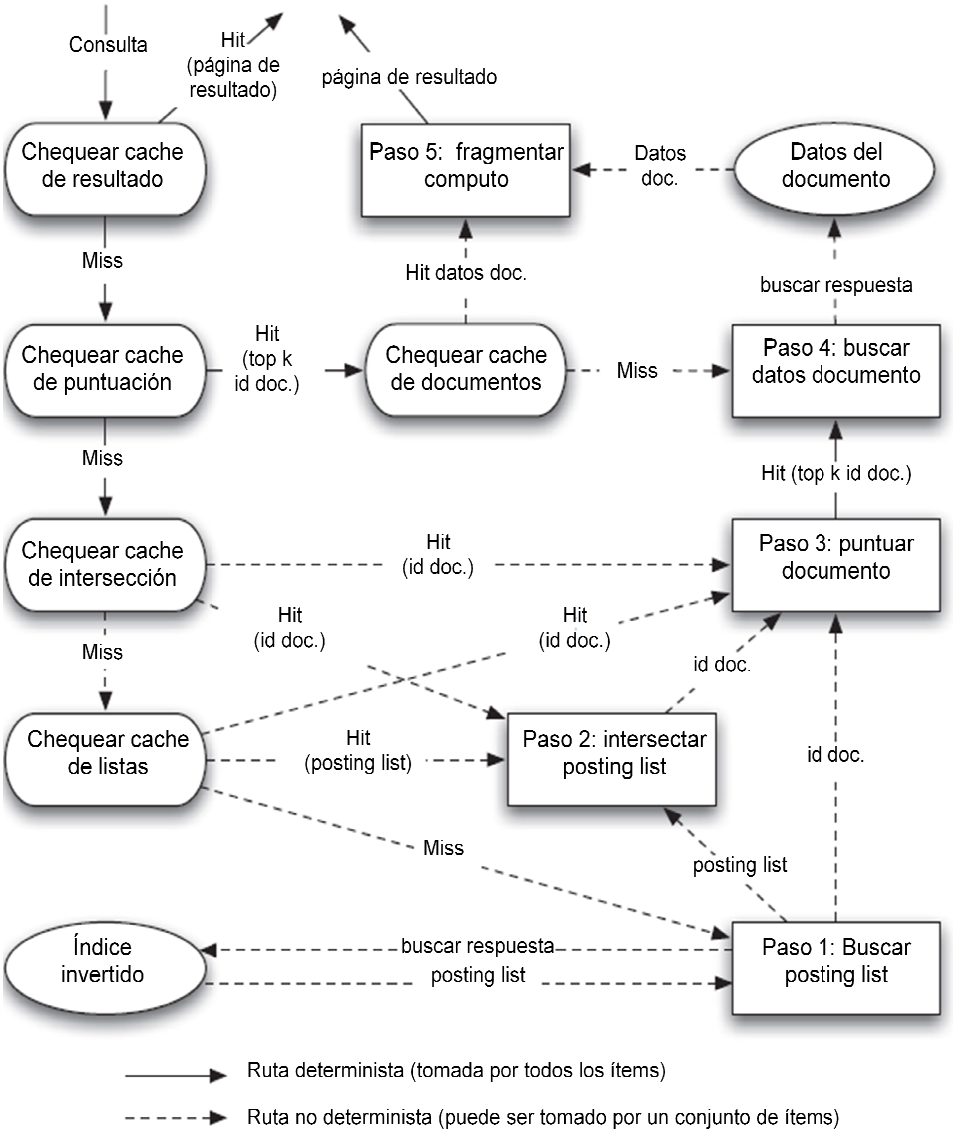

Figura 4.7: Flujo de trabajo utilizado para simular la arquitectura cache. Fuente: (Ozcan R. , Altingovde, Cambazoglu, Junqueira, & Ulusoy, 2012).

El algoritmo propuesto reduce el proceso total realizado por las consultas en el cache un 18,4% para arquitecturas de dos niveles propuesto por Correia Saraiva et. al (2001) y un 9,1% para arquitecturas de tres niveles propuesta por Long y Suel (2006). La mayor mejora se logró con respecto al algoritmo *baseline five – level* cache que es de un 4,8%.

4.1.5. Time-based query classification and its application for page rank

En Chen, Yang, Ma, Lei y Gao (2011) proponen una taxonomía a las consultas y a las páginas para mejorar el resultado del ranking. Esto lo logran por medio de dos pasos, el primero es crear un algoritmo de clasificación para establecer la categoría de la consulta basada en su frecuencia según los registros de las peticiones realizadas anteriormente. El segundo es utilizar un modelo dinámico teórico para establecer el *rank* de las páginas que toma en consideración la relevancia entre la información temporal contenida en la categoría de consultas y el tiempo de la publicación en la web.

En la Figura 4.8 se observa la clasificación hecha a las consultas según sus características temporales. Estas son: consultas permanentes, consultas periódicas, consultas en ráfaga y consulta en ráfaga múltiples.

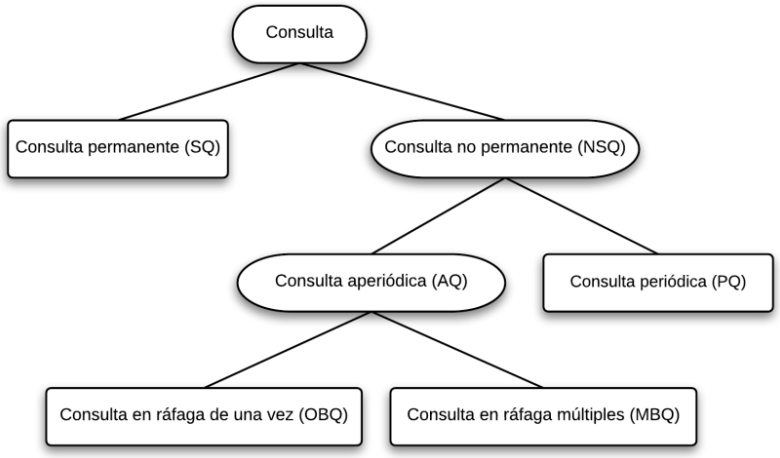

Figura 4.8: Taxonomía de consultas basadas en el tiempo. Fuente: (Chen, Yang, Ma, Lei, & Gao, 2011).

Al llegar una nueva consulta, es utilizado el algoritmo propuesto (basado en la frecuencia de búsqueda de la consulta y la tendencia registrada) el que determina la clase (permanente, periódica, ráfaga o ráfaga múltiple) a la que pertenece la consulta ingresada. Una vez obtenida la clase a la que pertenece la consulta es utilizado el modelo propuesto (*time-sensitive*) para determinar el *rank* de la página, cuya ecuación varía según al tipo de clase a la que pertenece la consulta.

Los métodos de ranking utilizados en este modelo son mejores que el método *baseline ranking*.

4.1.6. Facebook memcached escalable

En Nishtala, et al (2013), describen como Facebook mejora la versión de memcached propuesta por (Fitzpatrick, 2004) y la usa para construir un almacén distribuido del par *key-value*, escalando desde un solo *clúster* de servidores a múltiples clúster distribuidos geográficamente. La memcached es un sistema de código libre que implementa una tabla hash en memoria, ésta

posee 3 simples operaciones *set, get* y *delete*. Además, es capaz de proveer una baja latencia al acceder a conjuntos de almacenamiento compartido a un bajo costo.

En la figura 4.9 se ve que dividen los servidores web junto con la memcache[1] en múltiples clúster de front-end. Estos clúster, junto con los clúster de almacenamiento (contienen la base de datos), definen una región (una región tiene base de datos maestra, mientras que las base de datos de las demás regiones son réplicas). Al ser separadas en regiones, permite la existencia de menores fallos de dominios y una configuración de red tratable. Cada cliente mantiene una lista de todos los servidores disponibles y tiene distintas funciones que incluyen serialización, compresión, petición de enrutamiento, manejo de errores y petición de procesamiento por lotes. Los servidores memcached no se comunican entre ellos. La lógica del cliente tiene 2 componentes: una librería que puede ser embebido en las aplicaciones o un proxy autónomo llamado mcrouter. Este proxy presenta una interfaz del servidor memcached y recorre las peticiones/respuestas hacia/desde otros servidores. El cliente usa el protocolo UDP para las peticione *get* con el fin de reducir la latencia y los costos que puede generar. Por seguridad, el cliente realiza las peticiones set y delete por medio del protocolo TCP mediante una instancia de mcrouter ya que corre en la misma máquina que el servidor web.

El proceso a grandes rasgos que se sigue para generar una petición es el siguiente: si un cliente memcached no recibe respuesta de su petición get, el cliente asume que el servidor falló y envía una segunda petición a un grupo (pool) especial llamado Gutter. Si la segunda petición falla, el cliente va consultar a la base de datos para después insertar el par *key-value* apropiado

[1] Se hablará de memcached para referirse al código fuente que se ejecuta; mientras que memcache corresponde al sistema distribuido como tal.

en la máquina Gutter (las entradas en Gutter expiran rápido para no tener problemas de invalidación de Gutter).

En caso que se desee actualizar la información que afecta a una *key k* del servidor web, ese servidor establece una marca remota en la región (así en el caso que llegue otra petición desde otro servidor web busque la respuesta directamente en la base de datos maestra), luego el servidor web le escribe a la base de datos maestra y borra el par *key-value* de la memcache, la base de datos maestra envía una replicación Mysql a la base de datos replicada, la que luego quita el marcador remoto a la región.

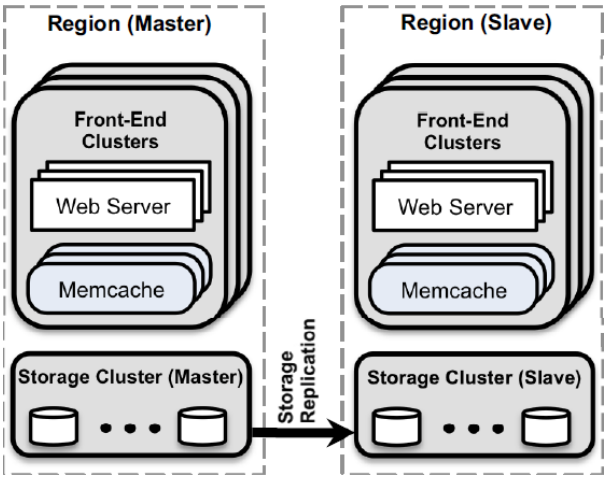

Figura 4.9: Arquitectura general facebook. Fuente: (Nishtala, y otros, 2013).

4.2. Organización y gestión de cache

El cache al poseer un espacio limitado de almacenamiento de objetos que contienen las respuesta pre computadas a las peticiones realizadas por los usuarios, debe contar con políticas que permitan mantener en cache las entradas actualizadas y que permitan generar hits en un futuro cercano. Para

esto se deben utilizar políticas que por medio de distintos procedimientos permitan el desalojo de consultas que pueden producir miss en cache y el ingreso de aquellas que logren generar hits en cache para así disminuir el acceso al back-end. A continuación se darán a conocer las distintas políticas para cache existentes en la literatura.

4.2.1. Políticas de desalojo

El cache tiene una cantidad limitada de entradas lo que significa que al existir un miss en cache este debe adquirir e ingresar el nuevo objeto. Para esto se deben utilizar políticas de desalojo, las que deben considerar diferentes factores al momento de tomar la decisión sobre si la nueva entrada es aceptada en el cache y determinar si uno o más objetos son removidos de este. En el caso de que la nueva entrada no es aceptada por el cache (según su política de admisión) el contenido queda sin modificar. El principal objetivo de estas políticas es disminuir los miss en cache. Esto se puede lograr generalmente eliminando aquellos objetos/documentos que no son referenciados en cache.

Se pueden evaluar las políticas de desalojo por medio de distintas métricas que miden el rendimiento obtenido al ser utilizados para realizar un reemplazo de los objetos en cache. Según Balamash y Krunz (2004) algunas métricas son (ver tabla 4.2): *Hit Ratio* (HR - cantidad de documentos que tienen hit en cache según el número de peticiones que se ha realizado al mismo), *Byte Hit Ratio* (BHR - cantidad de información que ha tenido hit en cache en relación con la cantidad de información procesada), *Delay Saving Ratio* (DSR - corresponde a la fracción de retardo ahorrado al tener un hit en cache y no tener que ir a buscar la petición al servidor web), *Average Download Time* (ADT - tiempo medio de descarga), *Saved Bandwidth* (ancho de banda ahorrado), entre otros.

Métricas	Notación
Hit Ratio	si : tamaño del documento i. fi : número total de peticiones al documento i. hi : número total de hits que ha producido al documento i. di: retardo de obtener el documento i desde el servidor web correspondiente. R: conjunto de todos los documentos a los que se accedieron. $\|R\|$: tamaño de R.
$\frac{\sum_{i \in R} h_i}{\sum_{i \in R} f_i}$	
Byte Hit Ratio	
$\frac{\sum_{i \in R} s_i \cdot h_i}{\sum_{i \in R} s_i \cdot f_i}$	
Delay Saving Ratio	
$\frac{\sum_{i \in R} d_i \cdot h_i}{\sum_{i \in R} d_i \cdot f_i}$ (Tiempo de validación ignorado).	
Average Download Time	
$\frac{\sum_{i \in R} d_i \cdot (1 - h_i / f_i)}{\|R\|}$	
Saved Bandwidth	
Relacionado directamente con el BHR	

Tabla 4.2: Ejemplos de métricas de rendimiento usadas en políticas de desalojo. Fuente: (Balamash & Krunz, 2004).

Por lo tanto, si se quiere disminuir el número de peticiones que se hacen al servidor, se debe incrementar el Hit Ratio, mientras que si se desea disminuir el ancho de banda requerido para responder una consulta, se debe aumentar el Byte Hit Ratio.

Uniendo los trabajos de Podlipnig y Böszörmenyi (2003) y Wong (2006) las políticas de desalojos se pueden clasificar en:

i. Estrategias basadas en lo reciente

Las estrategias basadas en lo reciente siguen la lógica de que los objetos que han sido accedidos recientemente tienen mayor probabilidad de ser accedidos nuevamente en un futuro cercano. Esta estrategia incorpora el tiempo desde

que fue referenciado el objeto (y tamaño y/o costo) en el proceso de reemplazo. Algunas políticas de reemplazo que siguen esta estrategia son.

- ***Least Recentently Used (LRU)***

  Este algoritmo está basado en la regla de localidad temporal que define que "Los objetos que no han sido referenciados recientemente, no se espera que sean referenciado en un futuro cercano" (Vakali, 2000). Esta política remueve del cache la entrada que ha sido recientemente menos referenciada. Según Wong (2006), LRU es una política popular debido a su simpleza y el buen rendimiento que tiene en muchas situaciones. Esta política funciona particularmente bien en peticiones web que muestran una alta localidad temporal. Esto pasa cuando varios clientes tienen interés en un conjunto común de objetos web. La complejidad de LRU es de *O(1)*.

- ***SLRU***

  Esta política es introducida en el trabajo de Arlitt, Friedrich y Jin, (1998). Dividen el cache en 2 segmentos, un segmento protegido (reservado para los documentos más populares) y otro desprotegido. Cuando un objeto es referenciado por primera vez, éste es guardado en el espacio desprotegido. Cuando el documento produce un hit en cache es desplazado al segmento protegido del cache. Ambos segmentos son manejado por medio de la política de reemplazo LRU pero solamente los objetos del segmento desprotegido son desalojados. En el caso que un documento sea removido del segmento protegido al desprotegido, este documento es añadido en la posición en que va el objeto más recientemente usado (así permanece en cache más tiempo, en el caso que vuelve a ser popular). Si para ingresar este documento se debe eliminar una entrada del segmento desprotegido, el documento menos

recientemente referenciado es eliminado primero. En Aggarwal, Wol y Yu (1999) proponen una heurística para determinar qué documento debe ser reemplazado desde el segmento desprotegido. Esta función es la siguiente:

$$v_i = \frac{\left(c_i \cdot (1 - \gamma_i)\right)}{s_i \Delta_{i,k}}$$

donde $\Delta_{i,k}$ es el número de acceso desde la última vez que fue referenciado el documento *i* , s_i es el tamaño del documento, c_i es el costo, γ_i es un factor de refresco de la sobrecarga de un objeto *i* entrante. Para el reemplazo se compara el valor del documento que se desea dejar en cache con el valor del conjunto de objetos candidatos a dejar el cache, admitiendo el objeto si es rentable hacerlo. La complejidad de SLRU es de *O(n)*.

ii. Estrategias basadas en la frecuencia

Las estrategias basadas en la frecuencia mantienen en cache los objetos que han sido referenciados con más frecuencia, y así lograr que una mayor cantidad de peticiones puedan ser satisfechas. Esta estrategia incorpora la frecuencia en que ha sido invocado el objeto (y tamaño y/o costo) en el proceso de reemplazo. Una de las políticas de reemplazo que siguen esta estrategia es:

- ***Least Frequently Used (LFU)***

  LFU remueve la entrada menos frecuentemente referenciada. Según Wong (2006), no es recomendado usar LFU puesto que sufre de problemas de corrupción de cache (p.ej. cuando un objeto acumula un recuento de referencias muy alto y se convierte en impopular debe pasar

un largo tiempo antes de ser candidato para ser eliminado). La complejidad de LFU es de O(log(n)).

iii. Estrategias basadas en el tamaño del objeto

En esta estrategia usualmente son reemplazados los objetos de mayor tamaño primero. Esto es porque la mayoría de los objetos web tienen un tamaño pequeño, tal que al borrar el mayor, podría dar acceso a múltiples objetos de menor tamaño. Una de las políticas de reemplazo que sigue esta estrategia es:

- ***Size***

  Size remueve la entrada de mayor tamaño primero, esta estrategia debe ser implementada manteniendo una cola de prioridad basado en el tamaño de los objetos. Debido a que el tamaño de los objetos es fijo, se requiere una constante de tiempo para los hits en cache (Wong, 2006). La complejidad de *Size* es de *O(log(n))*.

iv. Estrategias basadas en lo frecuente/reciente

Incorporan las dos estrategias nombradas anteriormente basadas en la frecuencia y en lo reciente, asumiendo el costo/tamaño que puede ser fijo o variable. Una de las políticas de reemplazo que sigue esta estrategia es:

- ***Hyper-G***

  Esta política es introducida en el trabajo de Williams, Abrams, Standridge, Abdulla y Fox, (1996) donde combinan LRU, LFU y SIZE. Al inicio el objeto menos frecuentemente referenciado es seleccionado. Si más de un objeto cumple este criterio, el cache selecciona el objeto menos referenciado recientemente. En el caso que no dé un solo objeto,

el documento de mayor tamaño es elegido para ser reemplazado. El principal objetivo de esta política es aumentar la tasa de hits. La complejidad de *Hyper-G* es de *O(log(n))*.

v. Estrategias basadas en funciones

Estas estrategias incorporan funciones específicas con diferentes factores (tiempo, tamaño, frecuencia, costo, latencia y distintos parámetros). Generalmente los objetos con el menor valor son reemplazadas primero. Algunas políticas de reemplazo que siguen esta estrategia son:

- ***Hybrid***

  Esta política es introducida en el trabajo de Wooster y Abrams (1997). El objetivo de esta política es reducir la latencia de acceso. Hybrid reemplaza del cache los documentos que tienen menor valor dada la siguiente función:

$$v_i = \left(c_s + \frac{W_b}{b_s}\right) \cdot \frac{(f_i)^{W_n}}{s_i}$$

  Donde c_s es el tiempo de conexión con el servidor s para obtener el documento i, b_s es el ancho de banda con el servidor, f_i es el número de veces que se ha referenciado el documento i desde que está en cache, s_i es el tamaño del documento i, W_b y W_n son constantes que establecen la importancia relativa de las variables b_s y f_i respectivamente. Por lo tanto, Hybrid es menos probable que reemplace aquellos documentos que se encuentran en un servidor que toma mucho tiempo en conectarse y/o el ancho de banda de conexión es bajo y/o el documento es referenciado varias veces y/o el tamaño del documento es pequeño. La complejidad del algoritmo utilizado por Hybrid es de *O(log(n))*.

- ***Least Unified Value (LUV)***

  Esta política es introducida en el trabajo de Bahn, Koh, Noh y Lyul Min (2002). Intentan obtener los beneficios de LRU y LFU en un esquema unificado, usando la información de todos los objetos referenciados recientemente y la frecuencia en que han sido referenciado dichos objeto. LUV asigna un valor *V(i)* a cada documento *i*:

  $$V(i) = W(i) \cdot p(i)$$

  donde *W(i)* es el costo relativo que se tiene al ir a buscar el objeto al servidor original el cual es definido como:

  $$W(i) = \frac{c_i}{s_i}$$

  donde c_i y s_i representan el costo y tamaño del objeto respectivamente. De la primera ecuación, se tiene que *p(i)* es la probabilidad de que un objeto *i* sea referenciado en el futuro, y se calcula de la siguiente forma:

  $$p(i) = \sum_{k=1}^{f_i} F(t_c - t_k)$$

  donde t_c es el tiempo actual y t_k es el tiempo en que se referenció el documento en el instante *k*. *F(x)* debe ser una función decreciente para asignar un mayor peso a las referencias más recientes:

  $$F(x) = \left(\frac{1}{2}\right)^{\lambda x} \text{ donde } 0 \leq \lambda \leq 1$$

el valor de λ (que es calculado de manera offline) permite a LUV representar distinta políticas de almacenamiento en cache que tiene en cuenta información de carácter reciente a corto plazo y la información de frecuencia a largo plazo de los documentos. A medida que λ se va acercando a 1, LUV le da más énfasis a la información reciente. En cambio a medida que λ se acerca 0, LUV da más énfasis a la información de frecuencia.

Según Balamash y Krunz (2004) el valor del documento *i* en la *k-ésima* referencia se puede definir recursivamente como una función de su valor en el (*k*-1)-*ésima* referencia, por lo tanto LUV reemplaza el documento de menor valor siguiendo la función:

$$V_i(k) = F(t_i)V_i(k-1) + \frac{c_i}{s_i}$$

donde t_i es el tiempo desde el último acceso. El valor del documento solo cambia cuando se accede a él. El valor de λ es computado fuera de línea tal que maximice el rendimiento deseado de la métrica. La complejidad de LUV es de *O(log(n))*.

- ***Mix***

Esta política es introducida en el trabajo de Niclausse, Liu y Nain (1998), donde tratan de encontrar una política que tome en cuenta los parámetros importantes para todo todos los documentos en el cache. Mix reemplaza del cache los documentos que tienen menor valor dada la siguiente función:

$$v_i = \frac{\left(lat_i^{r_1} \cdot f_i^{r_2}\right)}{\left(tref_i^{r_3} \cdot s_i^{r_4}\right)}$$

Donde lat_i es la latencia de acceso en la descarga del documento *i*, f_i es el número de veces que se ha referenciado el documento *i*, $tref_i$ es el tiempo desde que se realizó la última referencia al documento *i*, s_i es el tamaño del documento *i*, r_1, r_2, r_3 y r_4 son valores determinados experimentalmente (los autores utilizaron $r_i = 1$, para $i = 2, 3$ y 4, y $r_1 = 0{,}1$). Por lo tanto, Mix prefiere mantener en cache aquellos documentos que son costosos (tiempo) de obtener y/o que se han referenciado frecuentemente, antes que aquellos documentos que sean de gran tamaño y/o no se hayan referenciados durante un largo período de tiempo. La complejidad del algoritmo utilizado por Mix es de *O(n)*.

- ***Greedy Dual Size (GDS o GD-Size)***

  Esta política es introducida en el trabajo de Cao e Irani (1997). GDS mantiene para cada documento un valor característico H_i. Si se realiza una nueva referencia al objeto *i* (o una nueva referencia), es necesario volver a calcular el valor de H_i. GDS reemplaza del cache el documento que tiene menor valor dada la siguiente función:

$$H_i = \frac{c_i}{s_i} + L$$

  donde c_i es el costo de traer el objeto al cache, s_i es el tamaño del objeto y *L* es un valor inflado que se define en el momento de decidir el reemplazo. El objetivo de GDS es que los objetos que se demoran mucho tiempo en ser recuperados del servidor se mantengan más tiempo en cache. El algoritmo GDS sigue ciertos pasos cuando un documento no es encuentra en el cache y se desea ingresar un nuevo documento, *L* es definido en el momento de realizar el reemplazo y es inicializado como H_{min}, una vez que el documento *i* accede al cache, el valor de *H* es

reseteado a su valor inicial utilizando la formula H_i, los demás documentos mantienen su valor. La complejidad de GDS es de *O(log(n))*.

vi. Estrategias de asignación al azar

Las estrategias de asignación al azar son un enfoque diferente (no determinista) al reemplazo en cache. Al ser aleatorio el desalojo del objeto no requiere estructuras de datos para apoyar le decisión de reemplazo. Algunas políticas utilizan funciones para clasificar los objetos del cache y así seleccionar aleatoriamente uno de los mayores valores para ser reemplazado. Una de las políticas de reemplazo que sigue esta estrategia es:

- ***LRU-S***

  Esta política es introducida en el trabajo de Starobinski y Tse (2001), definen $s_{min} = \min\{s_1, s_2, \ldots, s_N\}$ correspondiente al tamaño mínimo de los *N* objetos almacenados en el cache $d_i = s_{min}/s_i$ corresponde a la densidad normalizada del objeto *i*. Cuando se referencia el objeto *i*, el algoritmo LRU-S actúa como LRU, los objetos cambian de posición en la cola usando la probabilidad d_i. En otro caso no se mueven. La complejidad de LRU-S es de O(1).

En la tabla 4.3 se observa un resumen de las políticas de desalojo nombradas anteriormente, el tipo de reemplazo que se utiliza, la clasificación y su complejidad.

Política	Estrategia	Complejidad	Reemplaza
LRU	Basado en lo reciente	O(1)	La primera entrada menos recientemente accedida.
LFU	Basado en la frecuencia	O(log(n))	La primera entrada menos frecuentemente accedida.
Size	Basado en el	O(log(n))	La entrada de tamaño más grande.

	tamaño		
Hyper-G	Basado en lo frecuente/ reciente	O(log(n))	La primera entrada menos frecuentemente accedida y luego la primera entrada menos recientemente accedida entre documentos de igual tamaño.
Hybrid	Basado en funciones	O(log(n))	El menor valor primero de acuerdo con la función: $v_i = (c_s + W_b/b_s)(f_i)^{W_n}/s_i$
Mix	Basado en funciones	O(n)	El menor valor primero según: $v_i = \left(lat_i^{r_1} \cdot f_i^{r_2}\right)/\left(tref_i^{r_3} \cdot s_i^{r_4}\right)$
SLRU	Basado en lo reciente	O(n)	El menor valor primero según: $v_i = \left(c_i \cdot (1-\gamma_i)\right)/s_i\Delta_{i,k}$
LUV	Basado en funciones	O(log(n))	El menor valor primero según: $V_i(k) = F(t_i)V_i(k-1) + c_i/s_i$ Donde $F(x) = (1/2)^{\lambda x}$
LRU-S	Asignación al azar	O(1)	El primer documentos que se encuentra al final del *stack* LRU (asumiendo que ha llegado un nuevo documento y es admitido).
GDS	Basado en funciones	O(log(n))	El menor valor primero según: $v_i = \frac{c_i}{s_i} + L$

Tabla 4.3: Resumen de políticas de reemplazo. Fuente: (Balamash & Krunz, 2004).

En las figuras 4.10 a la 4.15 se observa el resultado de las simulaciones realizadas por Balamash y Krunz (2004) en algunas de las políticas de desalojo que se nombró con anterioridad, específicamente LUV, GDS, Hyper-G, LRU, SIZE e Hybrid. Para comparar el rendimiento de cada una de las políticas se utilizaron las métricas de *Hit Ratio* (HR)*, Byte Hit Ratio* (BHR) y *Delay Saving Ratio* (DSR). Las trazas utilizadas fueron obtenidas del *Laboratory for Applied Network Research* (NLANR), las que corresponden al periodo de tiempo entre el 20 de marzo del 2002 y el 26 de marzo del 2002 con un total de 3.810.537 (30.6 GB) peticiones de las que 1.733.794 (15.6GB) son peticiones únicas; y del proxy público *Digital Equipment Corporation* (DEC) tomadas en el periodo del 1 de septiembre de 1996 y 10 de septiembre de 1996, con un total de 6.056.025

(51.7GB) peticiones de las que 3.070.404 (33 GB) son peticiones únicas. Solo se tomaron las peticiones que se almacenarán en cache. A simple vista se puede observar que la política LUV sobresale de las demás políticas en todas las métricas (se debe a la modificaciones fuera de línea que se les hace a λ).

En la figuras 4.10 y 4.11 se muestra el rendimiento de las políticas según la métrica HR. Se observa que LUV tiene la mejor tasa de hit, y le sigue GDS (con costo unitario) que sobresale de LRU y Hyper-G. Para cache de menor tamaño, SIZE tiene mejor tasa de hit que Hybrid, pero a medida que aumenta el tamaño del cache el rendimiento es similar. En caches de mayor tamaño Hybrid y SIZE comienzan a mejorar su tasa de hit con respecto a LRU y Hyper-G.

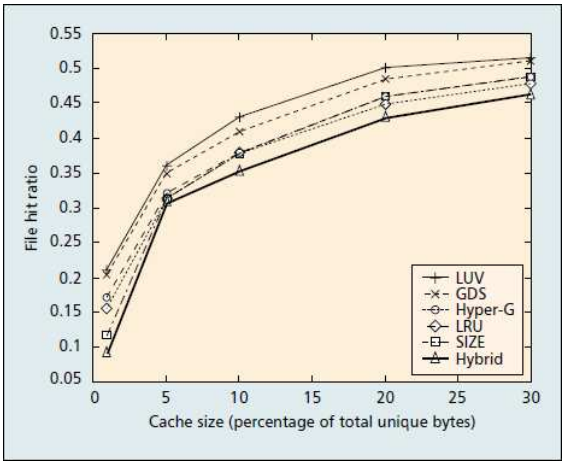

Figura 4.10: Traza NLANR métrica HR.
Fuente: (Balamash & Krunz, 2004).

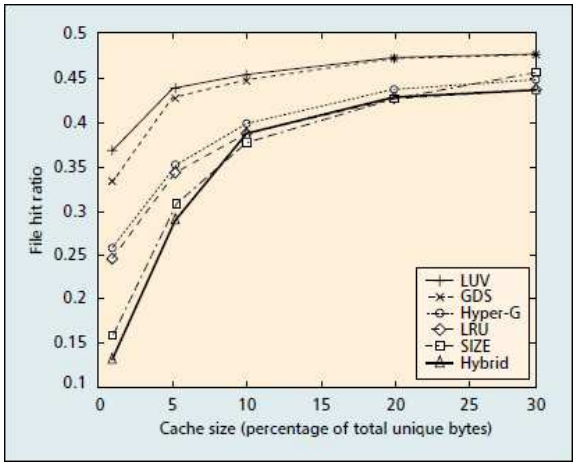

Figura 4.11: Traza DEC métrica HR.
Fuente: (Balamash & Krunz, 2004).

En las figuras 4.12 y 4.13 se muestra el rendimiento de las políticas según la métrica BHR, el mejor rendimiento lo tiene la política de reemplazo LUV, y lo sigue muy de cerca, a medida que aumenta el tamaño del cache, Hyper-G, LRU va en tercer lugar superando a GDS, para terminar con Hybrid y SIZE en quinto y sexto lugar respectivamente.

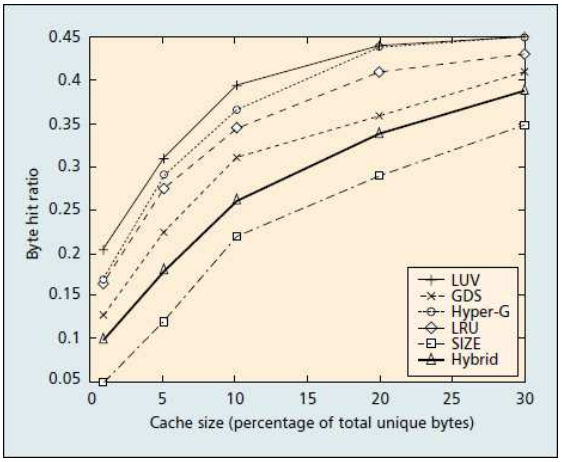

Figura 4.12: Traza NLANR métrica BHR.
Fuente: (Balamash & Krunz, 2004).

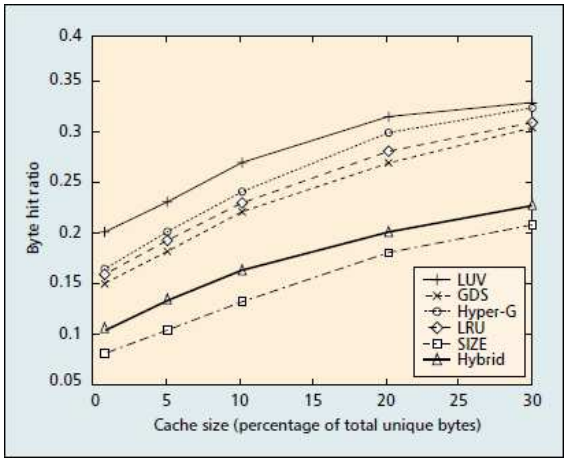

Figura 4.13: Traza DEC métrica BHR.
Fuente: (Balamash & Krunz, 2004).

Por último en las figuras 4.14 y 4.15 se muestra el rendimiento de las políticas según la métrica DSR. LUV continua mostrando el mejor rendimiento con respecto a las otras políticas, le sigue GDS que va mejorando a medida que aumenta el tamaño del cache, en tercer lugar esta Hyper-G, al que le sigue muy de cerca LRU, por último se encuentra Hybrid y SIZE.

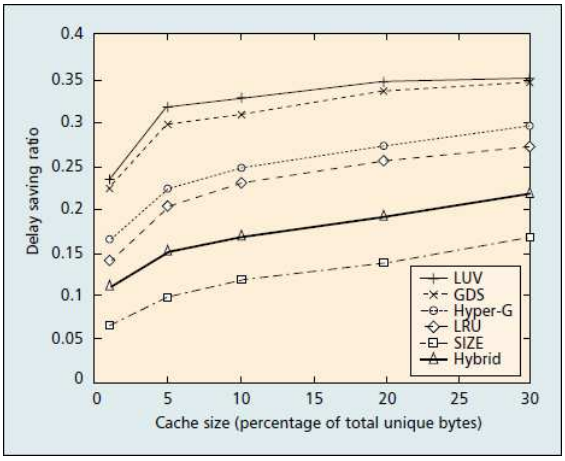

Figura 4.14: Traza NLANR métrica DSR.
Fuente: (Balamash & Krunz, 2004).

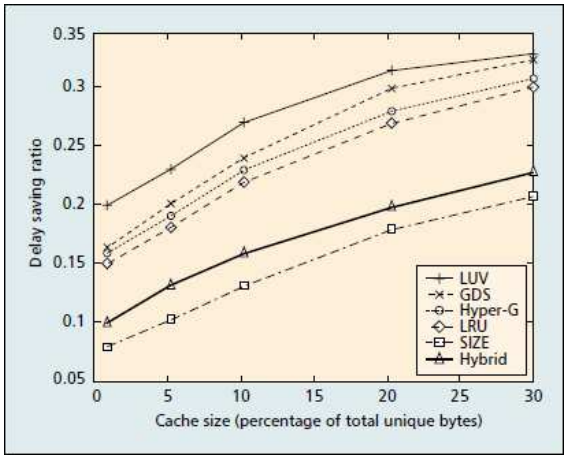

Figura 4.15: Traza DEC métrica DSR.
Fuente: (Balamash & Krunz, 2004).

Según Wong (2006) las políticas de desalojo tienen mejor rendimiento en algunos ambientes que en otros, algunos escenarios en los que se puede determinar la política que es conveniente utilizar para cada caso son (ver resumen en tabla 4.4):

- Caches de reducido tamaño: los caches pequeños tienen una baja HR, puesto que pueden mantener una pequeña cantidad de objetos en cache comparado con la cantidad de objetos existente en la carga de trabajo. Se sugiere utilizar políticas basadas en lo reciente ya que es más probable que se realicen referencias a un mismo objeto a corto plazo.

- Caches de gran tamaño: para caches de gran tamaño distintos estudios han mostrado que el rendimiento de distintas políticas es similar, en este caso una buena opción es la utilización de LRU como política de desalojo. Esto es para el almacenamiento de objetos web en general, para otros casos en que se requiera almacenar objetos de gran tamaño (vídeo) se deberán aplicar otras técnicas.

- Banda de ancho limitado: en caso de que el ancho de banda sea limitado se requiere disminuir el tráfico en la red, se sugiere utilizar políticas como GDS que provee de un bajo BHR en comparación con otras políticas y aumenta la HR.

- Proxy en pequeñas organizaciones: para este tipo de ambiente en que generalmente tiene una pequeña cantidad de usuarios y un gran tamaño de cache se sugiere utilizar LRU.

- Proxy a nivel de ISP (Internet *Service* Provider): poseen una gran cantidad de usuarios y por lo tanto una mayor localidad temporal a corto plazo, por lo que se sugiere utilizar LRU como política de desalojo.

- Proxy *root-level*: en caches jerárquicos el tráfico tiene una baja localidad temporal, por lo que para incrementar la HR se recomienda utilizar políticas basadas en funciones y que tomen en cuenta el tamaño de los objetos.

- Proxy basado en el comercio: para este tipo se recomienda utilizar una política simple como lo es LRU.

Características del cache	Estrategia
Cache de tamaño reducido	Políticas basadas en lo reciente.
Cache de gran tamaño	LRU, etc.
Ancho de banda limitada	GDS, etc.
Proxy en pequeñas organizaciones	LRU, etc.
Proxy a nivel de ISP	Políticas basadas en lo reciente.
Proxy root-level	Políticas basadas en funciones que consideren el tamaño de los objetos.
Proxy basado en el comercio	LRU, etc.

Tabla 4.4: Políticas de desalojo sugeridas para distintas características del cache. Fuente: (Wong, 2006).

4.2.2. Políticas de admisión

Al llegar una nueva consulta que no se encuentra almacenada en cache, se activan las correspondientes políticas de desalojo (en el caso en que el cache se encuentre lleno) para eliminar una entrada del cache e ingresar la nueva petición. Existen casos en que la nueva consulta puede no producir un hit en cache eliminando una entrada que si los produce, para estos casos es necesario utilizar políticas de admisión con el fin de evitar que esta entrada se guarde en cache y se mantengan aquellas que tienen más probabilidad de generar un hit en cache en el futuro. Algunos trabajos de políticas de admisión son:

4.2.2.1. Admission Policies for Caches of Search Engine Results

En Baeza-Yates, Junqueira, Plachouras y Witschel (2007) proponen una política de admisión dinámica capaz de prevenir consultas infrecuentes o simples ingresen al cache en motores de búsqueda web. Esto lo logran por medio de la utilización de un estimador que predice si la consulta es infrecuente

o lo suficientemente frecuente como para ser almacenada en el cache. El estimador puede usar características sin estado las que depende de cada consulta (es decir se puede calcular a partir de la propia secuencia de consulta sin usar información adicional y sin hacer seguimiento de estadísticas) o características con estado las que son calculadas a partir de la información de uso (estas características deben ser almacenadas y generalmente se relacionan con la frecuencia de la consulta).

La política de gestión de cache propuesta abarca tanto la política de admisión como políticas de desalojo, y divide al cache en dos partes: la primera parte se encuentran aquellas consultas que la política de admisión estima que podría tener hits en cache a futuro llamado cache controlado (CC); en la segunda parte se encuentran todas las demás consultas llamado cache descontrolado (UC). El resultado obtenido de las pruebas realizadas de los registros históricos obtenidos en la web Altavista (datos obtenidos en una semana) y yahoo.co.uk (consultas realizadas durante un año), muestran una mejora de un 21% sobre *Least Recentently Used* (LRU) y un 4% sobre *Static Dynamic Cache* (SDC) en 100.000 consultas en Altavista y mejoras en un 6% sobre LRU y de un 5% sobre SDC en 500.000 de consultas en yahoo.com.uk.

4.2.2.2. TinyLFU: A highly efficient cache admission policy

En Einziger y Friedman (2014) propone utilizar políticas de admisión basadas en la frecuencia aproximada, con el fin de aumentar la eficacia del cache que está sujeta a distribuciones de acceso sesgadas (un pequeño número de objetos son más probable a ser accedidos que otros). El esquema propuesto se llama TinyLFU, este mantiene un gran historial reciente (contiene estadísticas de las frecuencia de los ítems) y decide si se debe permitir el acceso de una nueva entrada a expensas del candidato a desalojar del cache ("victima"), con el fin de aumentar la tasa de hit (ver figura 4.18).

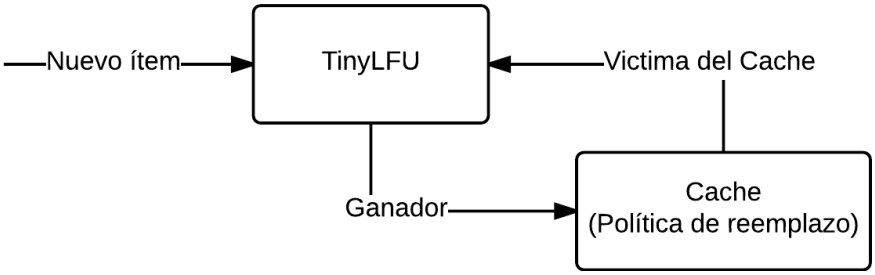

Figura 4.16: Arquitectura TinyLFU. Fuente: (Einziger & Friedman, 2014).

Almacenar las estadísticas de la frecuencia de los ítems tiene un alto costo, es por esto que emplean técnicas de recuento aproximado para estas frecuencias y así disminuir el consumo de memoria cache. Una de estas técnicas es la de mínimo incremento de *Counting Bloom Filter* (CBF).

El *Bloom Filter* es una estructura de datos probabilístico que permite preguntar, de manera eficiente, si un dato existe o no en un conjunto. Puede dar falsos positivos pero no falsos negativos. Funciona con un array de *m* bits inicializados en 0 y un conjunto de *k* funciones hash, que dado un dato generarán números entre 0 y *m* -1. Por otro lado, el CBF es un Bloom Filter en que cada entrada en el vector es un contador en lugar de 1 solo bit, donde las entradas son incrementadas por medio de una operación add/insert.

El mínimo incremento de CBF soporta dos métodos *Add* y *Estimate*. Ambos métodos se realizan mediante el cálculo de los *k* diferentes valores hash de la clave. Para el método de *Estimate* cada valor hash es tratado como un índice y es leído el contador del índice, donde el contador de menor valor es devuelto. El método Add lee todos los contadores *k* y solo incrementa los contadores mínimos (esto ayuda a que ítems que tienen baja frecuencia y tienen un comportamiento en ráfaga aumenten abruptamente su contador produciendo una mejor estimación).

El esquema de aproximación se debe mantener actualizado, para esto proponen un mecanismo llamado *reset*, que consiste en que cada vez que agregan un ítem al esquema incrementan el contador, una vez que el contador llegue al tamaño *W* dividen todos los contadores en 2. Al usar este mecanismo se sincroniza la operación *reset* con el cache y se hace un *reset* en las frecuencias de los ítems del cache.

La reducción de espacio se da por medio de dos ejes, el primero es reduciendo el tamaño de cada uno de los contadores del esquema de aproximación y segundo reduciendo el número total de contadores alojados por el esquema usando el mecanismo *doorkeper* (es un *bloom filter* situado antes del esquema de aproximación, cuando llega un ítem si no se encuentra el *doorkeper* lo almacena en el, si se encuentra en el *doorkeper* lo almacena en la estructura de aproximación), este mecanismo permite no mantener entradas únicas en el esquema de aproximación.

En los experimentos realizados para TinyLFU se tomaron trazas de distintas fuentes entre las que se encuentra YouTube, Wikipedia, motores de búsqueda (S3,WS1,WS2 y WS3), DS1 (base de dato), aplicaciones de instituciones financieras (F1 y F2), entre otros. En los resultados se comparan distinta políticas de desalojo que incorporan o no TinyLFU como política de admisión, entre estas políticas están: LRU, Random, TLRU (LRU incorpora TinyLFU), TRandom (Random incorpora TinyLFU), TLFU (LFU incorpora TinyLFU), W-TinyLFU, LIRS (ver (Jiang & Zhang, 2002)) y ARC (ver (Megiddo & Modha, 2003)). El resultado obtenido en la tasa de hit de cada política es mejor para aquellas que utilizan la política de admisión, destacando W-TinyLFU y TLRU.

4.2.3. Políticas de eliminación de resultados antiguos

La web se encuentra en constante cambio, lo que hace que existan entradas en el cache que al momento de acceder a ellas su resultado ha expirado, produciendo un miss en cache. Un ejemplo concreto de esta situación se da en los motores de búsqueda web, estos deben actualizar sus índices periódicamente para poder incorporar los cambios en la web, por lo que acceder a una entrada cuyo resultado ha sido modificado genera un *"cache invalidation"* provocando un miss en cache. Para poder evitar este problema se deben implementar políticas de eliminación de resultados antiguos, que permiten actualizar los resultados de las entradas almacenadas en el cache.

A continuación se dará a conocer algunos trabajos de política de eliminación de resultados antiguos.

4.2.3.1. A Refreshing Perspective of Search Engine Caching

En Cambazoglu, et al (2010), proponen una estrategia basada en asignar un valor *Time-To-Live* (TTL) en conjunto de un algoritmo que permite mantener actualizadas las entradas del cache. La solución que ellos proponen es un enfoque desacoplado que consiste en invalidar las entradas del cache sin saber si cambiaron o no su índice. Es decir, ya que los motores no están procesando peticiones en su capacidad máxima, se aprovecharán los ciclos inactivos en los servidores del back-end para re-procesar las respuestas y refrescar las entradas del cache.

La política que decidieron utilizar para asignar el orden en que van a actualizar cada entrada consiste en que las consultas que aparecen más frecuentemente y han estado una mayor cantidad de tiempo en cache van a tener una mayor prioridad a la hora de ser actualizadas. Para esto utilizan un mecanismo de actualización de dos dimensiones, correspondientes a la

frecuencia de la consulta y el tiempo que lleva la entrada en cache, utilizando la latencia (existente en CPU y memoria usada en los nodos del back-end) para ajustar la tasa de actualización del cache. Además, utilizan la heurística *cyclic refresh* para que el escaneo de las entradas del cache continúe desde donde fue dejado anteriormente y no vuelva a actualizar desde el inicio. Una vez que todas las entradas han sido escaneadas, comienza nuevamente desde el inicio. Esta política fue implementada en producción en los motores de búsqueda de Yahoo! aumentando la tasa de hit entre un 7% a un 10%. En otras palabras permite mejorar el tiempo de respuesta a la búsqueda del usuario y reduce las peticiones a los servidores que se encuentran en el back-end.

4.2.3.2. Caching search engine results over incremental indices

En Blanco, et al (2010) proponen una infraestructura digital para el desarrollo de un predictor de invalidaciones y definen las métricas para la evaluación del esquema de invalidación, esto lo hacen por medio de la incorporación de un nuevo componente a la arquitectura de los motores de búsqueda web llamado *Cache Invalidator Predictor* (CIP). CIP se encarga de invalidar del cache las entradas de forma selectiva usando los documentos entrantes y actúa como puente entre los procesos de indexación y de tiempo de ejecución de los motores de búsqueda web, para esto el cache debe estar al tanto de los documentos que provienen del proceso de indexación. CIP es constituida en dos piezas principales:

El generador de sinopsis: reside en el proceso de indexación y prepara una sinopsis (compacto de información sobre los atributos de puntuación del documento) de los nuevos documentos entrantes. En su salida envía un vector de los documentos con mayor puntuación (TF-IDF), estos son los términos por los cuales el documento podría tener una mayor puntuación. Para controlar la longitud de la sinopsis el generador envía una fracción η de los términos top de

cada documento en el vector (para $\eta = 0$ sinopsis vacío; para $\eta = 1$ sinopsis llena, todos los términos).

El invalidador: implementa la política de invalidación, recibe la información de los atributos de los documentos provenientes del generador de sinopsis y por medio de la interacción que tiene con el sistema en ejecución decide que entradas del cache invalidar. La interacción puede ser simple (ignora sinopsis) o compleja (evalúa cada par consulta-sinopsis).

Al llegar un documento *"D"*, este es procesado y se genera una sinopsis del documento, el invalidador identifica todas las consultas que coinciden con la sinopsis y luego decide si invalidar o no estas entradas del cache. CIP, también aplica invalidaciones basadas en la antigüedad de la entrada, asignando un tiempo de vida τ. Por último, elimina todas aquellas entradas en que su resultado ha sido borrado.

Las métricas de rendimiento se pueden observar en la tabla 4.5, la tasa de falsos positivos corresponde a los *q* resultados invalidados erróneamente lo que genera un costo computacional al tener que volver a procesar estas entradas; la tasa de falsos negativos es cuando CIP deja los resultados intactos dejando entradas antiguas en cache, perjudicando la calidad de los resultados.

Métricas	Notación
Tasa de falso positivo (FP)	PN : set de consultas de falsos positivos
$\frac{PN}{Q}$	NP: set de consultas falsos negativos
Tasa de falso negativo (FN)	Q : tamaño de set de consultas Q
$\frac{NP}{Q}$	q: consulta
Tasa de tráfico antiguo (ST)	S: set de consultas antiguas
$\sum_{q \in S} \frac{f_q}{F}$	f_q: frecuencia de la consulta
	F: Suma de todas las frecuencias de las consultas

Tabla 4.5: Métricas de rendimiento. Fuente: (Blanco, Bortnikov, Junqueira, Lempel, Telloli, & Zaragoza, 2010).

Los resultados de los experimentos realizados (utilizando registros históricos del motor de búsqueda Yahoo! sobre documentos de Wikipedia) muestran que para cada tráfico hacia un documento antiguo (ST) se han reducido los falsos positivos (FP) entre un 25% y un 30% en comparación con esquemas que utilizan de base el método TTL (Time To Live).

Capítulo 5

5. Descripción de la solución

La solución propuesta consiste en crear una arquitectura cache para aplicaciones web de gran escala tomando en consideración el comportamiento de los usuarios.

Una de las razones por las que se considera tomar en cuenta el comportamiento de los usuarios se debe a que las peticiones realizadas por ellos sigue un comportamiento dinámico. A esto se le puede agregar el comportamiento Zipfiano que siguen los usuarios en la web, donde existe un conjunto de términos de consultas que se realizan de forma permanente y en el otro extremo consultas que son realizadas una única vez sin llegar a generar una segunda petición en un futuro cercano. Por lo tanto, al tomar en cuenta el comportamiento de los usuarios se puede formar una estructura cache que mantenga almacenadas las consultas que son realizadas con mayor frecuencia durante largos periodos de tiempo (consultas permanentes), consultas que son realizadas frecuentemente durante un periodo de tiempo corto (consultas en ráfaga) y que cambian según los intereses momentáneos de los usuarios (consultas variables), con el fin de obtener una alta tasa de hit en cache.

Para construir una arquitectura capaz de cumplir las propiedades deseables de un sistema cache (rápido acceso, robustez, etc.) se tiene que considerar integrar distintas técnicas, algoritmos y estructuras para la administración de los objetos en el cache. Además, se debe considerar los

distintos tipos de peticiones realizadas por los usuarios, las que varían en el tiempo tales como las consultas en ráfaga y las consultas permanentes.

5.1. Estructura propuesta del cache

En trabajos realizado por Kumar y Norris (2008); Zhang, He y Ye (2013); Ozcan R., Altingovde, Cambazoglu, Junqueira y Ulusoy (2012); y Long y Suel (2006), proponen estructuras de múltiples niveles para el cache que permiten mejorar su rendimiento en comparación con arquitecturas que utilizan otros mecanismos para el almacenamiento en cache, como aquellos que usan solamente un cache dinámico.

En el trabajo de Kumar y Norris (2008) proponen una sección cuasi-estático y otra sección dinámica de almacenamiento en cache a nivel de web proxy. El cache cuasi-estático obtiene la información de consultas de registros históricos por medio de un mecanismo que obtiene las peticiones frecuentes, que no cambian su URL pero si su contenido, y que considere los costos asociados de ser almacenados en cache. El cache dinámico maneja las peticiones variables de los usuarios. Las pruebas realizadas fueron por medio de un log de *New York IRCache proxy server* entre el 29 de abril y el 30 de Junio del 2004 y corresponden a 2.567.818 consultas con un tamaño de almacenamiento de cache de 1000. Los resultados obtenidos muestran que el rendimiento aumenta en un 50% en términos de costos totales con respecto a la utilización de solo la política LRU.

En Zhang, He, y Ye (2013) proponen una estructura de dos niveles para motores de búsqueda web que posee un cache estático y un cache dinámico. El cache estático almacena las consultas más frecuentes que se obtienen de registros históricos de consultas de usuario. El cache dinámico cambia según las consultas de usuarios por medio de políticas de reemplazo. Se realizaron

varias pruebas con diferente cantidad de consultas entrantes que varían entre las 1000 y 80000 consultas; y se le asignó a cada cache 500 entradas de capacidad de almacenamiento. Los resultados del rendimiento obtenidos muestran que esta estructura es un 28% más rápida que usar solo una estructura de cache dinámico.

Los trabajos mencionados anteriormente, mejoran el rendimiento del cache al tener dos secciones donde se almacenan distintos tipos de consultas. Tomando en consideración lo antes expuesto y el comportamiento del usuario en la web, se decidió proponer una estructura que mantenga estas dos secciones y agregar una nueva sección en donde se almacenarán las consultas en ráfaga.

La arquitectura cache propuesta divide el cache en tres secciones (ver figura 5.1) que corresponden al cache estático, cache en ráfaga y al cache dinámico (la elección de cada una de estas secciones en específico será explicado más adelante). A cada una de estos segmentos se le asignará una cantidad de espacio para el almacenamiento de las entradas en cache, con el fin de obtener una alta tasa de hit. En el caso del cache ráfaga y el cache dinámico, la cantidad de espacios será compartida e irán variando a medida que se ingresen o eliminen entradas del cache en ráfaga. La cantidad de entradas asignadas para cada sección del cache dependerá directamente del tipo de aplicación en que utilice la arquitectura propuesta y el dispositivo en el que se esté ejecutando, siendo de menor tamaño para dispositivos móviles con baja capacidad de almacenamiento.

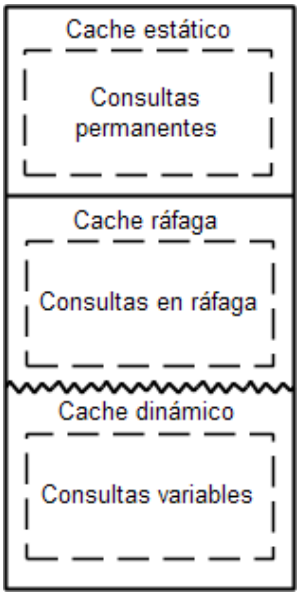

Figura 5.1: Estructura cache. Fuente: Elaboración propia.

Hay que tener especial cuidado con las consultas en ráfaga ya que se pueden confundir con consultas permanentes puesto que ambas poseen una alta frecuencia. Es importante a la hora de ingresar este tipo de consultas determinar si pertenecen a consultas en ráfaga o a consultas permanentes, por medio de la utilización de distintas técnicas. A continuación se describirá cada uno de los caches propuestos.

5.1.1. Cache estático

La utilización del cache estático ha sido estudiada en múltiples trabajos tales como los realizados por Zhang, He, y Ye (2013); Baeza Yates, et al (2007); Markatos (2001); Ozcan R., Altingovde, Cambazoglu, Junqueira, y Ulusoy (2012); y Kumar y Norris (2008). Ellos observaron que al mantener una sección estática en el cache, donde se almacenen las consultas más frecuentes realizadas por los usuarios (obtenidas de manera off-line desde un registro histórico de consultas), permite mejorar el rendimiento del cache junto con su

tasa de hit. Esto se debe a que las consultas que son realizadas de manera frecuente por los usuarios, permiten disminuir los costos (utilización de ancho de banda, tiempo de respuesta, entre otros) que se tienen al ir a buscar constantemente la respuesta al back-end.

La propuesta para el cache estático consiste en almacenar los resultados de las consultas permanentes, que se obtendrán inicialmente de manera off-line por medio de registros históricos de consultas.

En cuanto a las políticas para el cache estático, no se utilizarán políticas de admisión ni de desalojo puesto que son consultas con alto impacto y que se mantienen en el tiempo. Esto se debe a que podrían haber consultas o peticiones permanentes que no calcen dentro de las políticas de admisión.

Por otro lado, cada cierto tiempo se deberán analizar nuevamente los registros históricos de consultas en el caso que alguna entrada del cache estático deba ser reemplazada. Esto se puede deber a que la consulta no alcanza un umbral mínimo de frecuencia para ser considerada en el grupo de consultas permanentes, así como la aparición de nuevas consultas del tipo permanente que deban ser ingresadas en este cache. Esta medida se ejecutará durante periodos de tiempo largo (utilizando el mecanismo off-line antes mencionado) debido a que las consultas permanentes no cambian en periodos cortos de tiempo.

Se emplearán políticas de eliminación de resultados antiguos a largo plazo con el fin de mantener los resultados del cache actualizados. La política es realizada a largo plazo puesto que, al ser entradas que se encuentran permanentemente en cache, es poco probable que su resultado cambie. Para este cache se podría considerar utilizar *Time-To-Live* asignando un tiempo *t* a

cada entrada del cache estático y una vez que el tiempo se ha cumplido el resultado de la entrada es actualizado.

5.1.2. Cache ráfaga

Las consultas en ráfaga son aquellas peticiones que en un corto periodo de tiempo crece abruptamente la frecuencia en que son referenciadas y que son solicitadas por un periodo de tiempo acotado. Algunas de las razones por las que se decidió incorporar un cache ráfaga en la arquitectura propuesta son:

i. Las consultas en ráfaga son referenciadas frecuentemente por los usuarios. Al mantener estas consultas en cache durante su periodo de vida va a reducir el acceso al back-end aumentando la tasa de hit en cache.
ii. Las consultas en ráfaga que son almacenadas en el cache dinámico con políticas basadas en la frecuencia (por ejemplo: LFU), permanecerán en este cache de manera indefinida sin generar hits en cache una vez que los usuarios pierdan el interés sobre dicha consulta. Al tener conocimiento de las consultas que tienen características de ráfaga, permite eliminar la entrada del cache cuando la consulta pierde el interés por parte de los usuarios, dando espacio al ingreso de nuevas consultas que puedan ser referenciadas en un futuro cercano.

En el cache ráfaga se almacenará el par <consulta, respuesta> de las consultas en ráfaga y el tiempo de vida (*Time-To-Live)* que tiene la entrada en cache, el que será modificado según el flujo que siga la consulta (ver sección 5.2.4). En una primera instancia esta sección se encontrará vacía y se irán añadiendo entradas a medida que van surgiendo consultas en ráfaga. Para esto

se utilizará un mecanismo de detección temprana de consultas en ráfaga, que detecta las consultas con características de ráfaga. El mecanismo que se utilizará para la detección de ráfagas corresponde al algoritmo descrito por Gómez Pantoja (2014). Además, se contará con una política de eliminación de ráfaga, que debe ser capaz de determinar cuando la consulta será eliminada del cache ráfaga.

La cantidad de entradas del cache ráfaga será inicializada en cero. El espacio asignado a la sección ráfaga es compartida con la sección dinámica tal que al añadir una entrada a uno de estos caches el otro disminuye su tamaño. Por otro lado, no se espera tener grandes cantidades de consultas en ráfaga al mismo tiempo, puesto que aproximadamente un 0,00000182% de un log de consultas presentan características de ráfaga (Gómez Pantoja, 2014).

En cuanto a las políticas para cache de ráfagas, se utilizará políticas de eliminación de resultados antiguos para mantener actualizados los resultados de la sección ráfaga que se encuentra en constante cambio. El tiempo de actualización de los resultado antiguos debe ser a corto plazo puesto que son resultados que se mantienen en constante cambio. No se utilizarán políticas de admisión ya que se recurrirá a un mecanismo que permita detectar e ingresar ráfagas en cache. La eliminación de las consultas ráfagas será por medio de una política de desalojo del cache ráfaga que será explicada más adelante (ver sección 5.2.4).

5.1.3. Cache dinámico

En los trabajos de Markatos (2001); Baeza-Yates, Gionis, Junqueira, Murdock y Plachouras (2008); y Zhang, He, y Ye (2013) determinan que es necesario usar en conjunto al cache estático un cache dinámico que ayude a optimizar el rendimiento del cache.

Es necesario mantener en una parte del cache las consultas recientes que han realizado los usuarios y que no corresponda al grupo de consultas ráfaga y permanentes, tal que permitan mantener en cache los intereses que van adquiriendo los usuarios por distintos tópicos en el tiempo y que puedan generar en un futuro cercano hits en cache.

La propuesta para el cache dinámico consiste en almacenar el par <consulta, respuesta> de consultas variables. En un inicio el cache se encontrará vacío y se irán almacenando las distintas entradas (pares) de las peticiones recientes realizadas por los usuarios, que no pertenecen al set de consultas en ráfaga ni permanentes. La capacidad del cache dinámico irá variando a medida que las consultas en ráfaga sean ingresadas o desalojadas del cache ráfaga.

En el cache dinámico se utilizarán políticas de eliminación de resultados antiguos a mediano plazo para mantener actualizadas los resultados del cache, sobre todo aquellas entradas que tienen una mayor antigüedad. Para esta política también se podría considerar utilizar un *Time-To-live* (Alici, Sengor Altingovde, Ozcan, Cambazoglu, & Ulusoy, Julio 2011) en que se le asigna un tiempo *t* a cada respuesta de las consultas entrantes al cache. Cada vez que el tiempo *t* expire la entrada será actualizada. Para aplicaciones correspondientes a motores de búsqueda web, se puede considerar utilizar los métodos mencionados en el capítulo 4 sobre eliminación de resultados antiguos y que

fueron propuestos por Cambazoglu, et al (2010) que propone asignar un TTL a cada entrada aprovechando los ciclos inactivos para actualizar todas las respuestas del cache.

El cache dinámico utilizará políticas de desalojo que permita eliminar entradas del cache (una vez que alcance su tope máximo) para ingresar las nuevas entradas, que en un futuro cercano puedan ocasionar un hit en cache.

Existen distintas políticas de desalojo que se pueden utilizar para reemplazar las entradas en el cache dinámico, las que estarán relacionadas al tipo de aplicación web, el tamaño del cache, el tipo de cache, entre otros. En la tabla 4.4 del capítulo 4 se mencionan algunas políticas que se pueden utilizar dependiendo de los factores antes nombrados.

Una buena opción de política de desalojo a utilizar corresponde a LRU, la cual es recomendada para varios tipos de cache debido a su simpleza, pero se debe tener en cuenta que existen políticas que tienen un mejor rendimiento en ciertos tipos de aplicaciones.

El cache dinámico será la única sección en utilizar políticas de admisión, se utilizarán con el fin de no ingresar en cache entradas que probablemente no sean referenciadas próximamente, y así aumentar la tasa de hit en el cache dinámico. Algunas políticas de admisión nombradas con anterioridad en el capítulo 4 que se podrían utilizar son TinyLFU propuesta por Einziger y Friedman (2014). Para motores de búsqueda se podría utilizar la política propuesta por Baeza-Yates, Junqueira, Plachouras, y Witschel (2007) que previene que consultas infrecuentes entren al cache por medio de un estimador que predice si la consulta va a ser referenciada nuevamente.

5.2. Arquitectura propuesta

5.2.1. Políticas de la memoria cache

Existen tres políticas principales en una arquitectura cache:

i. Política de desalojo: Debido a que la memoria cache es un contenedor limitado, esta política constantemente libera espacio mediante la selección de una entrada para su desalojo. Esto hace posible la incorporación de nuevas entradas en cache.

ii. Política de eliminación de resultados antiguos: La memoria cache es un contenedor de respuestas pre-computadas. Estas respuestas podrían eventualmente ser obsoletas. Mediante esta política la respuesta es refrescada cada cierto intervalo de tiempo.

iii. Política de admisión: Debido al espacio limitado del cache, es necesario utilizar las entradas de manera efectiva. Por otro lado, las peticiones de usuario tienen patrones bien definidos, por lo que es posible analizar y establecer si una petición es poco frecuente o no. Esta política determina que entrada debe ser almacenada en la memoria cache.

En la tabla 5.1, se resumen algunas relaciones entre las políticas descritas anteriormente y las secciones de la arquitectura cache propuesta.

	Política de desalojo	Política eliminación resultados antiguos	Política de admisión
Intervalo de tiempo entre invocaciones	Cada vez que se debe insertar una entrada	Depende de la sección del cache	Cada vez que se deba insertar una entrada
Cache estático	No (conjunto fijo de entradas, recalculado cada cierto período de tiempo)	Sí (largo plazo)	No (toda petición permanente debe estar en cache)
Cache ráfaga	Sí (entrada deja de ser ráfaga)	Sí (corto plazo)	No (toda ráfaga debe ser admitida)
Cache dinámico	Sí (inserción de nueva entrada que es admitida)	Sí (mediano plazo)	Sí (toda entrada debe ser validada para entrar en cache)

Tabla 5.1: Resumen de políticas y secciones de cache. Fuente: creación propia.

Por otro lado, con respecto al dinamismo, las tres secciones exhiben distintos patrones. La sección dinámica del cache experimenta un alto dinamismo, ya que constantemente esta desalojando entradas para dar espacio a nuevas entradas. El dinamismo de la sección ráfaga es medio, ya que sólo entrará un número limitado de entradas. La entrada y salida de una petición en esta sección está dada por la detección de una ráfaga y su posterior eliminación. Finalmente, la sección permanente presenta bajo dinamismo, ya que su actualización tiene relación con una mirada de largo plazo.

5.2.2. Secciones de la memoria cache

La sección permanente (ver figura 5.2) debe ser poblada por peticiones con alta frecuencia y estables en el tiempo. Estas peticiones son obtenidas de forma off-line, mediante un mecanismo que analiza registros históricos [100]. Estas entradas y sus respuestas pre-computadas son almacenadas en el cache estático [104]. Este mecanismo debe ser ejecutado cada cierto tiempo (largo plazo, por ejemplo cada mes), para almacenar en cache posibles nuevas

consultas de ese tipo que surjan en el tiempo. Esta sección no cuenta con política de admisión, ya que todas las entradas deben ser consideradas, independiente de sus características. Finalmente, la política de eliminación de resultados antiguos [110] debe actualizar las respuestas pre-computadas de esta sección con una mirada a largo plazo (por ejemplo, cada día).

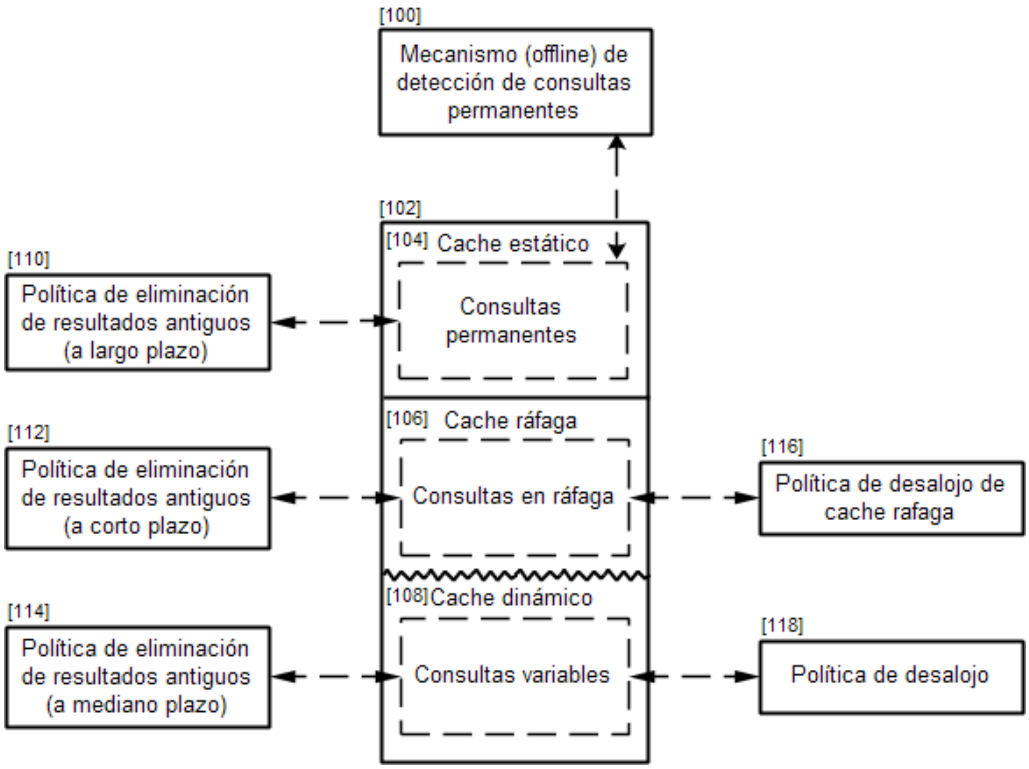

Figura 5.2: Arquitectura cache propuesta. Fuente: Creación propia.

La sección ráfaga no posee un mecanismo de llenado a priori, sino que es poblada dinámicamente con peticiones declaradas como ráfagas. En el trabajo de Gómez Pantoja (2014) se propone un esquema de detección de consultas en ráfaga eficiente en tiempo y espacio. Este esquema se basa principalmente en la obtención de los top-*k* aproximados durante un intervalo, el cual permite establecer una serie de tiempo acotada de frecuencias aproximadas de las consultas top-*k*. Luego, utilizando promedio y desviación

estándar sobre esta serie de tiempo acotada, se determina si una consulta es ráfaga. Es decir, este mecanismo determina el inicio del intervalo de vida de una ráfaga, por lo que debe insertarse en la sección ráfaga [106]. El esquema de detección de ráfaga contempla todas las consultas entrantes para actualizar el mecanismo. En nuestro caso no se considerarán las consultas que se encuentren en la sección permanente puesto que ya se encuentran en cache y se mantendría dentro de los top-*k* las consultas de las sección estática quitando lugar a otras peticiones que no se encuentren en cache. Por otro lado, utilizando las consultas top-*k* aproximados de cada intervalo se puede establecer si la ráfaga se encuentra activa en el tiempo, en caso contrario la política de desalojo de cache ráfaga [116] determinará su eventual eliminación del cache ráfaga (ver sección 5.2.4). Esta sección del cache no tiene una política de admisión, ya que las peticiones en ráfaga son prioritarias y no deben ser filtradas por esta clase de políticas. En términos de actualización de resultados, esta sección de cache debe re-ejecutar cada petición constantemente, ya que las peticiones alojadas en esta sección tienen un alza abrupta en el interés de los usuarios. Esto quiere decir que la política de eliminación de resultados antiguos [112] debe ser constantemente invocada (por ejemplo, pocos minutos).

La sección dinámica corresponde a una porción del cache en la que se almacenan las peticiones que no pertenecen a las dos secciones anteriores (permanente y ráfaga). Esta sección tiene un alto dinamismo, ya que constantemente se desalojan entradas y se insertan nuevas entradas. Por ende, la utilización de una política de desalojo [118] es mandatorio para dar cabida a nuevas entradas. En esquemas clásicos, esta política de desalojo debe invocarse cada vez que una nueva petición no se encuentre en cache. De este modo, toda nueva petición tiene una oportunidad de permanecer en cache durante un intervalo de tiempo. Una política de admisión, utilizada antes que una política de desalojo, hace más efectiva la utilización de esta sección, ya que

inhibe la entrada de peticiones poco útiles. Una política de eliminación de resultados antiguos [114] clásica es *Time-To-Live* (Alici, Sengor Altingovde, Ozcan, Cambazoglu, & Ulusoy, Julio 2011), que corresponde a una estampilla de tiempo asignada a cada entrada en la cual debe re-ejecutarse.

Es importante señalar que la única sección de tamaño fijo es la sección permanente, ya que el conjunto de peticiones a alojar es calculado y definido a priori, con una mirada de largo plazo. Por otro lado, el espacio restante es dividido entre la cache dinámica y la cache ráfaga. La idea de funcionamiento tiene relación con un complemento, es decir, si se requiere insertar una nueva entrada en la cache ráfaga, entonces se debe decrementar el tamaño de la cache dinámica. De este modo, el tamaño de la cache dispuesto para estas dos secciones no cambia. Inicialmente, la cache ráfaga inicia vacía, por lo que el cache dinámico es propietario de la totalidad de este espacio.

En términos lógicos, la sección dinámica y ráfaga se encuentran en porciones separadas como se observa en la figura 5.2. En términos prácticos, todas las entradas de estas dos secciones pueden ser organizadas en la misma sección de memoria, pero diferenciando la pertenencia a cada sección con una marca. Es decir, una entrada en cache puede tener una marca que indique si es ráfaga o no. Si tiene esta marca habilitada, entonces pertenece a la sección ráfaga. En caso contrario, es de la sección dinámica.

5.2.3. Flujo de mecanismo de inserción de ráfaga

El ingreso de las consultas al cache ráfaga se realiza por medio del mecanismo de detección de ráfaga propuesto en el trabajo de (Gómez Pantoja, 2014). El mecanismo de ingreso de ráfaga es ejecutado en cada intervalo de tiempo de un minuto, donde se obtienen las top-*k* consultas más frecuentes de

ese intervalo (ver figura 5.3). Para cada consulta perteneciente a los top-*k* se obtiene la frecuencia aproximada, la desviación estándar y el promedio de la serie de tiempo de los últimos 3 intervalos, y se determina si la consulta cumple con los criterios necesario para ser considerada como una consulta en ráfaga [200]. En el caso que sea una ráfaga se debe verificar que no se encuentre almacenada con anterioridad en el cache ráfaga [202]. Si la consulta se encuentra ya en el cache ráfaga se da por terminado el flujo [218]. Por otro lado, si la petición no se encuentra en el cache ráfaga, es posible que una consulta en ráfaga sea insertada en el cache dinámico en su preámbulo, es decir antes que el mecanismo detecte que corresponde a una consulta en ráfaga. En el caso que la consulta se encuentre en el cache dinámico [204], esta petición en la sección dinámica debe ser trasladada a la sección ráfaga, mediante su desalojo [206], el decremento del tamaño de la sección dinámica [208], el incremento del tamaño de la sección ráfaga [210], y la inserción de la petición en esta sección [212], dando por terminado el flujo [218]. Por el contrario, si la consulta no se encuentra en la sección dinámica [204] se verifica si el cache se encuentra lleno [214]. En el caso que no esté lleno es incrementado el tamaño de la sección ráfaga [210] e insertada en esta sección [212] dando por finalizado el flujo [218]. Si el cache dinámico se encuentra lleno se procede a utilizar políticas de desalojo [216] para decidir que par consulta-respuesta del cache variable debe ser eliminado, decrementando el tamaño de la sección dinámica [208], aumentado el tamaño de la sección ráfaga [210], para luego insertar la petición en esta sección [212], dando por finalizado el flujo [218].

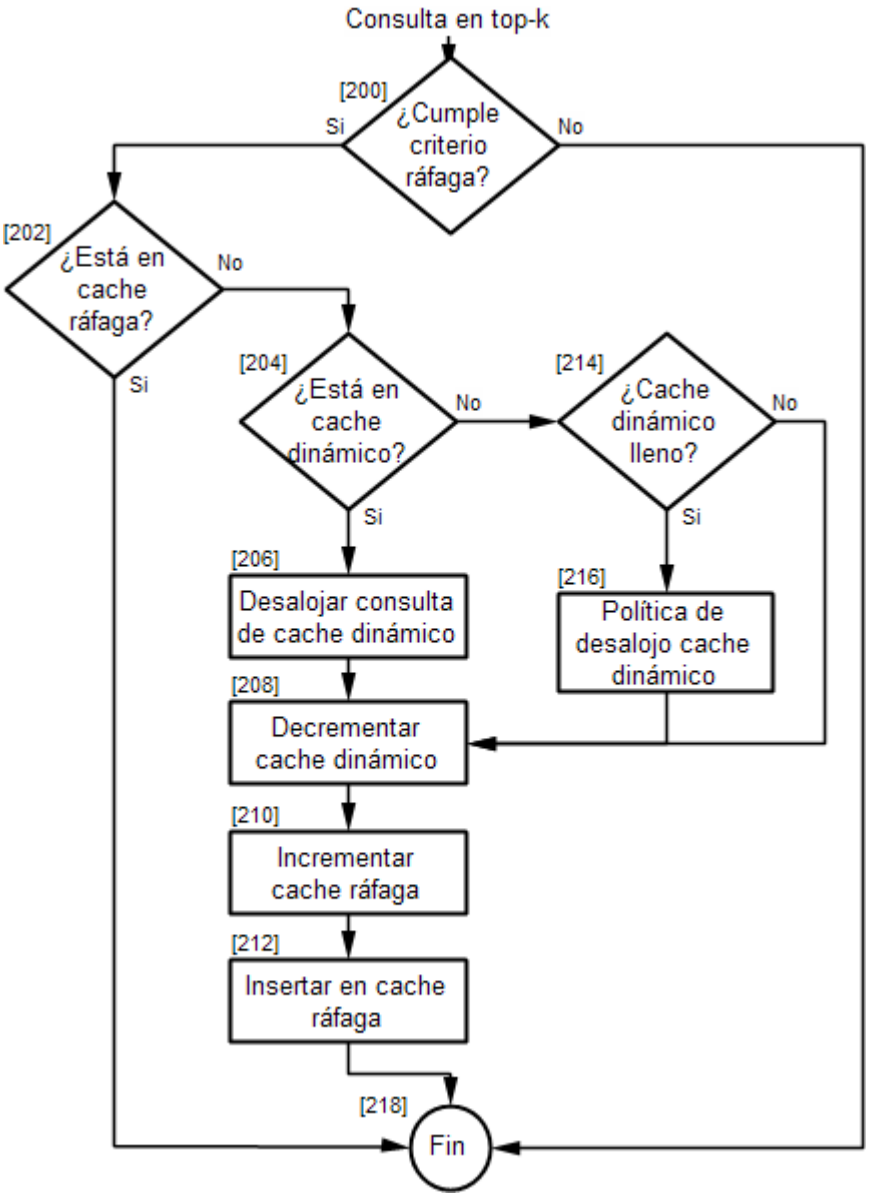

Figura 5.3: Mecanismo de inserción de ráfaga. Fuente: Creación propia.

5.2.4. Flujo de política de desalojo de cache ráfaga

El desalojo de la consulta en ráfaga es ejecutado cada intervalo de tiempo *t* (en nuestro caso un minuto), donde por cada consulta perteneciente a la sección ráfaga se realiza el flujo correspondiente a la figura 5.4. Durante cada intervalo, a las consultas pertenecientes a la sección ráfaga, se le asignará un tiempo de vida (*Time To Live* - TTL) el que será eliminado o irá disminuyendo a medida que las consultas que se encuentran en la sección ráfaga pertenezcan a los top-*k* consultas frecuentes del intervalo. Al terminar ese tiempo (TTL) la consulta será desalojada del cache ráfaga.

El desalojo de las consultas del cache ráfaga se hace por medio de la obtención de los top-*k* aproximados del mecanismo de detección de ráfaga durante un intervalo. En el caso que la consulta perteneciente a la sección ráfaga se encuentre dentro de los top-*k* [300], se debe verificar si posee un tiempo de vida (TTL) pre-asignado [302]. Si tiene un tiempo de vida asignado [302], el tiempo de vida es eliminado [304] de la entrada en ráfaga ya que la consulta sigue siendo solicitada por los usuarios y se da por terminado el flujo [322]. Si la consulta no tiene un TTL se finaliza el flujo [322]. Por otro lado, si la consulta no se encuentra dentro de los top-*k* [300] se verifica si el TTL tiene un valor de cero [308]. Si el TTL de la consulta es cero quiere decir que la consulta ya cumplió con su tiempo de vida y debe ser desalojada de la sección ráfaga [310], decrementando el tamaño de la cache ráfaga [312] e incrementando el tamaño de la sección dinámica [314], para luego insertar la consulta desalojada de la sección ráfaga en el cache dinámico [316], dando por finalizado el flujo [322]. Por el contrario si el par consulta-respuesta tiene un TTL distinto de cero [308], éste es disminuido en uno y se da por finalizado el flujo [322]. Por último, en el caso que la entrada no tiene establecido un TTL [306], se le asigna uno [320] y se finaliza el flujo de esa consulta.

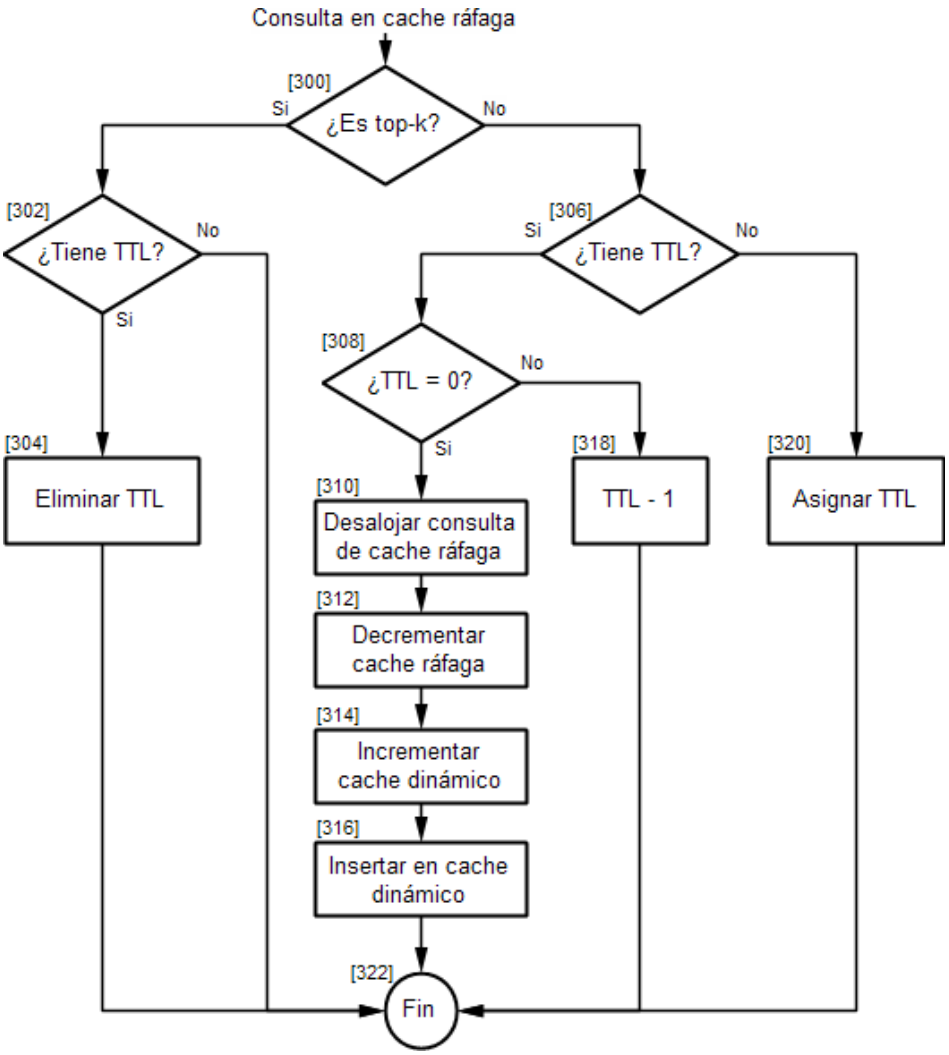

Figura 5.4: Política de desalojo cache ráfaga. Fuente: Creación propia.

5.2.5. Flujo de operación

A continuación, se describe la interacción entre las distintas secciones de cache al procesar una nueva entrada (ver figura 5.5). La acción más importante es preguntar si la petición se encuentra pre-computada en cache [400]. Si la entrada está en cache, se obtiene su respuesta pre-computada y la sección en cache donde está almacenada. Luego, la respuesta a la petición es devuelta al usuario [402]. Posteriormente, varias acciones toman lugar dependiendo de la

sección en cache donde está la petición (dado que es un *hit* en cache, está en alguna de las tres secciones). Si la petición se encuentra en el cache estático [404], se da por terminado el flujo [426]. En el caso contrario [404] la consulta entrante es ingresada al mecanismo de detección de ráfagas [406]. Si la petición se encuentra en cache dinámico [408], entonces se deben actualizar los atributos de esta petición dependiendo de la política de desalojo implementada [410]. Luego la consulta finaliza su ciclo [426]. En el caso que la petición no está en cache dinámico [408], quiere decir que pertenece a la sección ráfaga y se da por finalizado el flujo [426].

Si la petición no está en cache [400], se envía la petición al servicio *back-end* [412], se devuelve la petición al usuario cuando se procesa la petición y se actualiza el mecanismo de detección de ráfagas [416], para finalizar el flujo de la petición [426]. En paralelo al envío de la petición y su respuesta al usuario, se debe determinar si se admite o no la consulta en cache [418]. Si es admitida, se pregunta si la sección dinámica está llena [420]. Si está llena, entonces se debe aplicar la política de desalojo en la sección dinámica [422] e insertar la nueva petición en la sección dinámica [424]. Si no está llena, se debe realizar la inserción directamente en la sección dinámica [424], dando por finalizado el ciclo [426]. Por otro lado, si la consulta no es admitida se da por terminado el flujo [426].

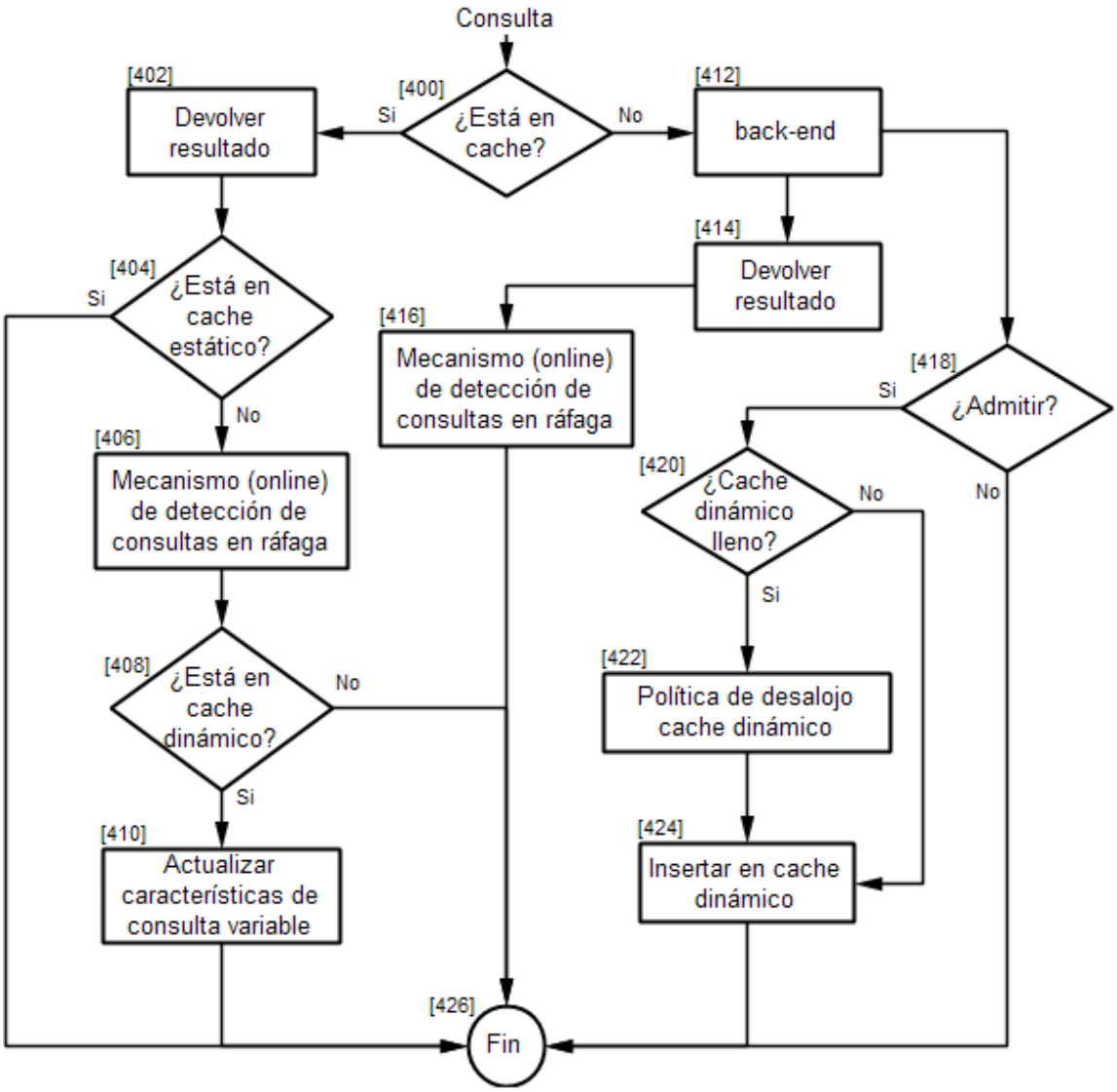

Figura 5.5: Flujo de arquitectura cache propuesta. Fuente: Creación propia.

5.3. Modificación a arquitecturas existentes

Al realizar un análisis de distintos tipos de arquitecturas existentes en el medio, se observó que existen varias propuestas con respecto a la división del cache en distintas secciones o niveles. Trabajos como los realizados por Zhang, He y Ye (2013); Baeza-Yates, et al (2007); Markatos (2001); Ozcan R. , Altingovde, Cambazoglu, Junqueira y Ulusoy (2012); Kumar y Norris (2008); Long y Suel (2006); entre otros, proponen mejorar el rendimiento del cache en diversos aspectos por medio de la existencia de diferentes caches que almacenen distintos tipos de consultas, por ejemplo: consultas dinámicas, estáticas, *posting lists*, intersección de listas, entre otros. Pero en ninguno de estos trabajos se propone manejar las consultas en ráfaga ni mucho menos mantener una sección del cache dedicada al almacenamiento de este tipo de consultas.

La solución propuesta se diferencia de otras arquitecturas en la inclusión de una sección de cache de ráfaga que utiliza mecanismo de detección de ráfaga propuesto por Gómez Pantoja (2014). Además, se propone manejar distintos tiempos para las políticas de eliminación de resultados antiguos correspondiente a largo plazo para el cache permanente, a mediano plazo para el cache dinámico y a corto plazo para el cache ráfaga.

La arquitectura propuesta está orientada a aplicaciones web de gran escala, tanto para dispositivos con grandes capacidades de almacenamiento en cache, como para aquellos que poseen un espacio de almacenamiento menor como dispositivos móviles.

Capítulo 6

6. Pruebas y resultados de la solución

6.1. Pruebas

6.1.1. Características de la entrada

Las pruebas se realizaron con registros históricos de consultas de un motor de búsqueda, que consiste en una colección masiva de consultas realizadas durante el mes de agosto del año 2012. Los datos incluyen la fecha en que se ha realizado la petición en formato timestamp Unix y los términos de la consulta.

Los términos que serán ingresados mediante un mecanismo off-line al cache permanente, corresponden a una colección de las primeras 50.000 peticiones más frecuentes realizadas en el motor de búsqueda durante el mes de julio del año 2012 ordenadas según su frecuencia.

Una vez obtenidos los registros de las pruebas preliminares se procedió a generar el código fuente de la arquitectura cache propuesta en el lenguaje de programación C++.

6.1.2. Algoritmos de la arquitectura cache propuesta

El algoritmo 1 corresponde al pseudocódigo del método off-line para la obtención de los top-*n* términos más frecuentes que serán ingresados al cache estático. El registro corresponde al mes anterior del período en que se comenzarán a realizar las pruebas, en nuestro caso el mes de julio del año 2012. En las pruebas realizadas se asignaron distintos tamaños a la sección estática, las que corresponden a una capacidad de: 300, 500, 1500, 2500, 3000, 5000, 15000, 25000, 30000 y 50000.

Algoritmo 1: Método off-line de ingreso de consultas al cache estático.

Entrada: TF términos *t* de consultas de log de muestra ordenadas de mayor a menor frecuencia, *te* tamaño máximo del cache estático, *ce* cache estático.

```
1.  i = 0
2.  read(TF)
3.  while (t ∈ TF) do
4.      if (te ≥ i ) then
5.          salir
6.      else
7.          ce.ingresar t
8.          i ++
9.      end if
10. end while
```

Una vez obtenido las consultas pertenecientes al cache estático, se procede a emular el procesamiento on-line de las entradas del registro en el que se va a llevar a cabo las pruebas de la arquitectura cache propuesta.

En el algoritmo 2 se observa el pseudocódigo del procesamiento de las consultas realizadas en el registro histórico del motor de búsqueda del mes de

agosto del año 2012. En este caso se invoca al mecanismo de detección de ráfaga para alimentar el esquema con todas aquellas consultas que no se encuentran a priori en el cache estático. Esto se hace con el fin de que el mecanismo de detección de ráfaga tome peticiones que no se encuentren en cache y así dar paso a un mayor universo de consulta en la búsqueda de las peticiones top-*k* más frecuentes.

Algoritmo 2: Flujo de operación de la arquitectura cache propuesta.

Entrada: TF términos *t* de consultas del log de muestras, *ce* cache estático, *cd* cache dinámico, *cr* cache ráfaga, *cv_max* tamaño máximo del cache variable.

1. $read(TF)$
2. $\boldsymbol{while}(t \in TF)\boldsymbol{do}$
3. $\quad\boldsymbol{if}(t \in ce \vee t \in cd \vee t \in cr)\boldsymbol{then}$
4. $\quad\quad devolver\_resultado$
5. $\quad\quad\boldsymbol{if}\big(!\,(t \in ce)\big)\boldsymbol{then}$
6. $\quad\quad\quad mecanismo\_detección\_ráfaga\ (insertar\ t)$
7. $\quad\quad\quad\boldsymbol{if}(t \in cd)\boldsymbol{then}$
8. $\quad\quad\quad\quad cd.actualizar\_caracteristicas\_consulta$
9. $\quad\quad\quad\boldsymbol{end\ if}$
10. $\quad\quad\boldsymbol{end\ if}$
11. $\quad\boldsymbol{else}$
12. $\quad\quad enviar\_petición\_back - end$
13. $\quad\quad devolver\_resultado$
14. $\quad\quad mecanismo\_detección\_ráfaga\ (insertar\ t)$
15. $\quad\quad\boldsymbol{if}(adimitir)\boldsymbol{then}$
16. $\quad\quad\quad\boldsymbol{if}(cd.tamaño\ + cr.tamaño\ \geq\ cv\_max)\boldsymbol{then}$
17. $\quad\quad\quad\quad cd.politica\_de\_desalojo$
18. $\quad\quad\quad\quad cd.insertar\ t$
19. $\quad\quad\quad\boldsymbol{end\ if}$
20. $\quad\quad\boldsymbol{end\ if}$
21. $\quad\boldsymbol{end\ if}$
22. $\boldsymbol{end\ while}$

La sección dinámica y ráfaga comparten el espacio de almacenamiento de las entradas, siendo prioritario el ingreso de las consultas ráfaga al cache. Esto se debe a que en su tiempo de vida generarán una mayor cantidad de hits en cache que una petición que no cumpla características de consulta en ráfaga ni permanente.

En las pruebas realizadas se le asignaron distintos tamaños a la sección dinámica-ráfaga que corresponde a la siguiente capacidades de almacenamiento de entradas del cache estos son: 500, 700, 2500, 3500, 5000, 7000, 25000, 30000, 35000, 50000, 70000.

En cuanto a las políticas de desalojo de la sección dinámica se realizarán pruebas con las políticas *Least Recentently Used* (LRU) y *Least Frequently Used* (LFU).

En el caso de LRU al llegar una petición que no pertenece a la sección estática ni a la sección ráfaga, la consulta será almacenada en la última posición del cache dinámico. Si el cache dinámico se encuentra a tope se desaloja la primera entrada de la sección, esto se debe a que la entrada que está en la primera posición ha permanecido durante más tiempo en el cache dinámico y debe ser reemplazada. En el caso que ingrese una consulta que pertenece al cache dinámico, se debe actualizar las características de la entrada, lo que se puede lograr reasignando la entrada a la última posición del cache.

En el caso de LFU al llegar una petición que no se encuentran en el cache ráfaga ni en el cache estático, se almacena en la sección dinámica y se inicializa en uno su frecuencia en caso de ser una nueva entrada. Si la sección dinámica llegó a su tope, se desaloja la entrada que posee la menor frecuencia

del cache. En caso que ingrese una consulta que ya se encuentra almacenada en la sección dinámica, se actualiza la frecuencia de la entrada.

A continuación se describirán los métodos utilizados en el mecanismo de detección de ráfaga, el posterior ingreso de la consulta al cache ráfaga y el método utilizado para ejecutar la política de eliminación de consultas en ráfaga.

6.1.2.1. Método de detección e ingreso de consultas en ráfaga

El mecanismo de detección de ráfaga propuesto en el trabajo de Gómez Pantoja (2014) se alimenta de todas las consultas entrantes del sistema, en nuestro caso solo se utilizará aquellos ítems que no se encuentran en cache estático. El esquema se ejecutará en intervalos de cada 60 segundos (ver figura 6.1), donde en cada intervalo procesará las consultas por medio del algoritmo *Space-Saving* (Metwally, Agrawal, & Abbadi, 2006) con el fin de obtener las *k* consultas más frecuentes junto con su frecuencia aproximada (top-*k* con *k* = 1000), las que serán ingresadas a una estructura de monitorización.

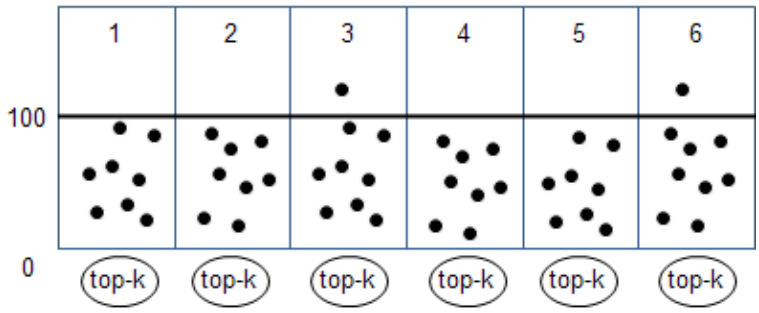

Figura 6.1: Mecanismo de detección de ráfaga top-k. Fuente: Creación propia.

El mecanismo necesita de una ventana de tres series de mediciones de frecuencia aproximada, que corresponde al mínimo de mediciones que necesita el método para declarar una consulta como ráfaga (Gómez Pantoja, 2014). Una vez que se obtengan tres series de tiempo (ver figura 6.2) se procede a verificar

si para cada serie de consultas se cumplen los criterios que indican si las consultas que se encuentran en la estructura de monitorización corresponden a una ráfaga o no.

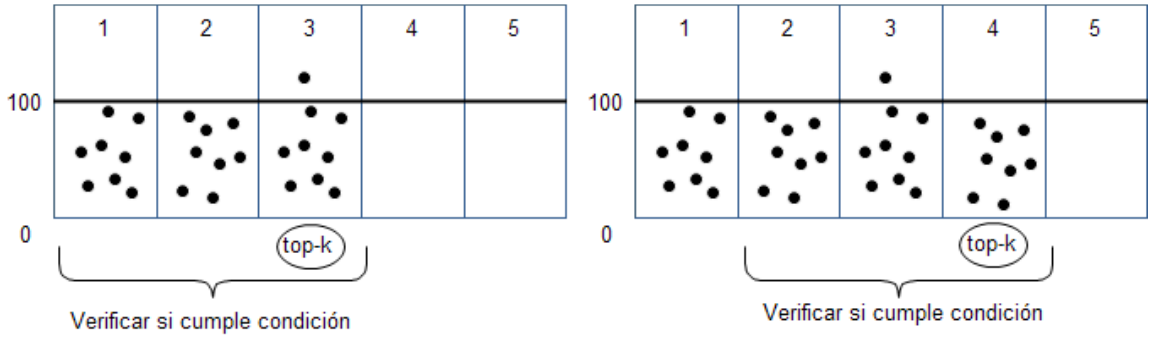

Figura 6.2: Mecanismo de detección de ráfaga *3*-series. Fuente: Creación propia.

Según Gómez Pantoja (2014), "Una consulta *q* es categorizada como ráfaga en el instante *t* si cumple las siguientes condiciones sobre su serie de tiempo $T_w(q)$ (de tamaño w)"

$$\left(f_t(q) \geq \overline{T_w(q)} + x \cdot \sigma(T_w(q))\right) \wedge \left(\frac{\sigma(T_w(q))}{T_w(q)} > r\right)$$

Donde $f_t(q)$ es la frecuencia aproximada de la consulta en el instante *t*, *r* indica si la desviación estándar móvil de la serie es *r* veces el promedio móvil, *x* indica si la frecuencia actual está *x* veces la desviación estándar móvil sobre el promedio móvil.

En las pruebas realizadas se utilizará un *r* = 0,3 y un *x* = 0,5. Además, la consulta *q* será catalogada como ráfaga si cumple con un umbral de 100, es decir, si la frecuencia aproximada de la consulta en el último intervalo de la serie es mayor que 100 y cumple las condiciones mencionadas anteriormente,

entonces la consulta será declarada como ráfaga y se ingresará al cache ráfaga.

El algoritmo 3 corresponde al mecanismo de detección de una ráfaga.

Algoritmo 3: Mecanismo de detección de ráfaga.

Entrada: SS estructura de monitorización de *t* términos top-*k*, *cr* cache ráfaga.

1. $\boldsymbol{for\ all}\ t \in SS\ \boldsymbol{do}$
2. $\quad \boldsymbol{if}\left((t.frec \geq \overline{t.serie} + 0.5 \cdot \sigma(t.serie)) \wedge \left(\left(\frac{\sigma(t.serie)}{\overline{t.serie}}\right) > 0.3\right) \wedge (t.frec > 100)\right)\boldsymbol{then}$
3. $\quad\quad \boldsymbol{if}(!(t \in cr))\ \boldsymbol{then}$
4. $\quad\quad\quad cr.ingresar\ t$
5. $\quad\quad \boldsymbol{end\ if}$
6. $\quad \boldsymbol{end\ if}$
7. $\boldsymbol{end\ for\ all}$

El algoritmo 4 contiene el método utilizado para el ingreso de una consulta declarada como ráfaga visto en el algoritmo 3.

Algoritmo 4: Mecanismo de ingreso de consulta al cache ráfaga.

Entrada: términos *t* declarado como ráfaga, *cr* cache ráfaga, *cd* cache dinámico, *cv_max* tamaño máximo del cache variable.

```
1.  if (! (t ∈ cr) then
2.      if(t ∈ cd)then
3.          cd.desalojar_consulta t
4.          cr.ingresar t
5.      else
23.         if(cd.tamaño + cr.tamaño ≥ cv_max)then
6.              cd.política_desalojo
7.              cr.ingresar t
8.          else
9.              cr.ingresar t
10.         end if
11.     end if
12. end if
```

Las consultas declaradas como ráfaga son de carácter mandatorio, puesto que siempre van a generar hits en cache durante el ciclo de vida de la consulta. Es por este motivo que se deber privilegiar el ingreso de consultas en ráfaga por sobre las consultas que se encuentran en cache dinámico.

En el caso del cache dinámico con políticas de desalojo LRU, al detectar una ráfaga que no se encuentre en la sección dinámica se desalojará la consulta que se encuentre en la primera posición del cache dinámico (siempre y cuando este en su máxima capacidad). Si la consulta en ráfaga se encuentra en cache dinámico antes de ser detectada como tal, se desalojará esa consulta en específico del cache dinámico dando un espacio a la sección ráfaga para su eventual ingreso.

En el caso de la sección dinámica con políticas de desalojo LFU al detectar una ráfaga que no se encuentre en la sección dinámica se eliminará del cache dinámico la consulta con menor frecuencia (siempre y cuando esté en su máxima capacidad). Si la consulta en ráfaga se encuentra en cache dinámico antes de ser detectada como tal, se desalojará esa consulta en específico de la sección dando un espacio a la sección ráfaga para su eventual ingreso.

6.1.2.2. Política de desalojo de consultas en ráfaga

La política de desalojo del cache ráfaga se basa en la asignación de un tiempo de vida (TTL) a las *n* consultas almacenadas en el cache ráfaga. Este tiempo de vida se asigna una vez que la consulta deja de pertenecer al grupo de peticiones top-*k* obtenido por el mecanismo de detección de ráfaga. Las consultas que pertenecen al cache ráfaga serán evaluadas para su desalojo en cada intervalo de tiempo de 60 segundos junto al mecanismo de detección de ráfaga.

En las pruebas se utilizará un TTL de 60 minutos. Esto quiere decir que si el término que se encuentra almacenado en cache ráfaga no aparece dentro de las consultas top-*k* durante el periodo de una hora (TTL = 0), ésta será eliminada del cache ráfaga e ingresada al cache dinámico. El ingreso al cache dinámico se debe a que la consulta puede seguir generando hits en un futuro cercano, aunque no se encuentre dentro de los top-*k*.

El algoritmo 5 contiene el método utilizado por la política de desalojo de consultas en ráfaga.

Algoritmo 5: Mecanismo de política de desalojo de consulta en ráfaga.

Entrada: términos *t* almacenados en el cache ráfaga, *cr* cache ráfaga, *cd* cache dinámico, *TTL* tiempo de vida de la consulta.

```
1.  for all t ∈ cr do
2.      if(t ∈ top_k)then
3.          if(t.tiene_TTL)then
4.              t.TTL = −1
5.          end if
6.      else
7.          if(t.tiene_TTL)then
8.              if(t.TTL = 0) then
9.                  cr.desalojar_consulta t
10.                 cd.ingresar_consulta t
11.             else
12.                 t.TTL − −
13.             end if
14.         else
15.             t.TTL = 60
16.         end if
17.     end if
18. end for all
```

6.1.3. Características de las pruebas

Las políticas de admisión y desalojo que serán utilizadas en el cache dinámico para las pruebas corresponde a las políticas de desalojo más conocidas que son *Least Recentently Used* (LRU) y *Least Frequently Used* (LFU) (Gómez Pantoja, 2014; Cambazoglu, y otros, 2010; Gan & Suel, Abril 2009; Baeza-Yates, Junqueira, Plachouras, & Witschel, 2007).

Se realizarán 20 pruebas en total con distintos tamaños asignados a la sección dinámica y ráfaga (ver tabla 6.1). De estas 20 pruebas 10 serán

asignadas a la arquitectura propuesta utilizando la política de desalojo LRU en la sección dinámica y las restantes 10 usando política de desalojo LFU en el cache dinámico.

Prueba	Capacidad sección estática	Capacidad sección dinámica- ráfaga	Capacidad total del cache
1	300	700	**1000**
2	500	500	**1000**
3	1500	3500	**5000**
4	2500	2500	**5000**
5	3000	7000	**10000**
6	5000	5000	**10000**
7	15000	35000	**50000**
8	25000	25000	**50000**
9	30000	70000	**100000**
10	50000	50000	**100000**

Tabla 6.1: Pruebas: tamaño de las secciones del cache. Fuente: Creación propia,

De la fórmula utilizada en el mecanismo de detección de ráfaga para determinar si una consulta corresponde a una ráfaga visto en la sección 6.1.2.1 se utilizarán en las pruebas los siguientes parámetros:

- x = 0,5
- r = 0,3
- umbral = 100

El tiempo de vida (TTL) asignado para la política de eliminación de ráfagas utilizado en las pruebas corresponde a 60 minutos.

6.2. Resultados

Los resultados obtenidos de las pruebas serán comparados con los algoritmos base de desalojo más conocidos correspondiente a LFU y LRU. Asignando la totalidad de la capacidad del cache a un cache dinámico con política de desalojo LFU, y otro con políticas de desalojo LRU. Para esto, se aplicó el mismo log de consultas utilizado en las pruebas de la arquitectura cache propuesta. De ahora en adelante se referirá a:

- **ACP-LRU:** arquitectura cache propuesta que hace uso de la política LRU en el cache dinámico.

- **ACP-LFU:** a la arquitectura cache propuesta que hace uso de la política LFU en la sección dinámica.

6.2.1. ACP-LRU v/s algoritmo base LRU

La tabla 6.2 muestra los resultados de ACP-LRU. En la primera columna contiene la cantidad total de entradas del cache, seguida por la columna "Prueba" donde se encuentra la capacidad asignada a la sección estática v/s la capacidad asignada a la sección dinámica en conjunto con la sección ráfaga. Se observa que a medida que va aumentando la capacidad de la sección estática se detectan una menor cantidad de ráfagas. Esto corresponde a una anomalía de la colección masiva de los datos de muestra y se puede deber a dos motivos: (i) algunas de las consultas almacenadas en el cache estático son ráfagas que fueron ingresadas en el mes anterior y que aún no han terminado su ciclo de vida y continúan con un comportamiento ráfaga en el mes que se realizó la muestra; (ii) existen consultas permanentes que no fueron ingresadas al cache estático y tienen un comportamiento de ráfaga en un instante de

tiempo donde son altamente solicitadas por los usuarios. A medida que la capacidad del cache aumenta, los hits producidos por la sección dinámica se asemejan a los obtenidos por el cache ráfaga. Esto no solo se debe a lo expuesto anteriormente sino que el aumento del tamaño del cache, permite que un mayor número de entradas puedan ser almacenadas en cache dinámico, lo que da lugar a que las entradas permanezcan por más tiempo en cache, dando la opción a que sean referenciadas en un futuro cercano.

Cantidad de entradas	Prueba	Hits sección estática	Hits sección ráfaga	Hits sección dinámica	Hits totales	Cantidad de ráfagas detectadas
1000	**300-700**	199.258.257	147.160.265	20.614.274	**367.032.796**	1280
	500-500	223.217.665	145.413.939	13.328.148	**381.959.752**	1267
5000	**1500-3500**	288.304.696	137.029.239	40.600.753	**465.934.688**	1201
	2500-2500	319.710.204	132.964.808	30.036.195	**482.711.207**	1166
10000	**3000-7000**	330.197.707	132.519.505	53.718.691	**516.435.903**	1164
	5000-5000	362.516.504	128.420.986	41.399.214	**532.336.704**	1124
50000	**15000-35000**	437.282.339	114.180.639	92.095.213	**643.558.191**	1040
	25000-25000	477.166.088	104.025.585	76.720.324	**657.911.997**	979
100000	**30000-70000**	490.798.000	101.578.659	105.763.304	**698.139.963**	963
	50000-50000	529.612.778	94.110.694	90.117.950	**713.841.422**	912

Tabla 6.2: Resultados ACP-LRU. Fuente: Creación propia.

Otra observación de la tabla 6.2 es que la mayor cantidad de hits en cache se obtiene de las entradas almacenadas en la sección estática, seguido en la mayoría de las pruebas por la sección ráfaga y por último la sección dinámica. Esto demuestra la alta cantidad de hits que genera una consulta en ráfaga durante su ciclo de vida y porque se le debe dar prioridad a que este tipo de consulta se mantenga en cache. Además, el aumento en capacidad del cache estático produce una mayor cantidad de hits en esta sección,

aumentando la cantidad de hits totales. Esto se debe a que al mantener en esta sección las consultas que son constantemente solicitadas por los usuarios aumente la tasa de hits en cache.

En la tabla 6.3 se muestra los resultados de hits totales obtenidos en la ACP-LRU v/s la utilización de un algoritmo base LRU. En cada prueba el algoritmo base utiliza la capacidad total del cache, es decir para la primera prueba "300-700" la capacidad total del cache para el algoritmo base es de 1000. En la última columna se observa la mejora producida ACP-LRU con respecto al algoritmo base, mejorando la tasa de hits en todos las pruebas. Se observa que en las pruebas donde la capacidad total del cache es el mismo y se asigna una mayor cantidad de espacio de almacenamiento al cache estático, aumenta la tasa de total de hits en cache.

Cantidad de entradas	Prueba	Hits arquitectura propuesta	Hits algoritmo base LRU	Mejora (% aprox.)
1000	300-700	367.032.796	315.626.558	**16,3%**
	500-500	381.959.752		**21%**
5000	1500-3500	465.934.688	419.790.612	**11%**
	2500-2500	482.711.207		**15%**
10000	3000-7000	516.435.903	468.631.609	**10%**
	5000-5000	532.336.704		**13,6%**
50000	15000-35000	643.558.191	592.507.547	**8,6%**
	25000-25000	657.911.997		**11%**
100000	30000-70000	698.139.963	646.044.741	**8%**
	50000-50000	713.841.422		**10,5%**

Tabla 6.3: ACP-LRU v/s algoritmo base LRU. Fuente: Creación propia.

En la figura 6.3 muestra el gráfico obtenido con los datos de la tabla 6.3, donde se observa que ACP-LRU, mejora la tasa de hits en cache al ser comparada con los hits obtenidos al hacer uso del algoritmo base LRU, haciendo uso del mismo registro de consultas en ambas estructuras. El mayor porcentaje de mejora se obtiene para los cache de menor tamaño, llegando a un 21% de incremento en su tasa de hits en el cache con capacidad 1000 (500-500). La menor mejora se obtiene para el cache con capacidad de 100.000 entradas (30000-70000), correspondiente a un 8% de incremento en su tasa de hits.

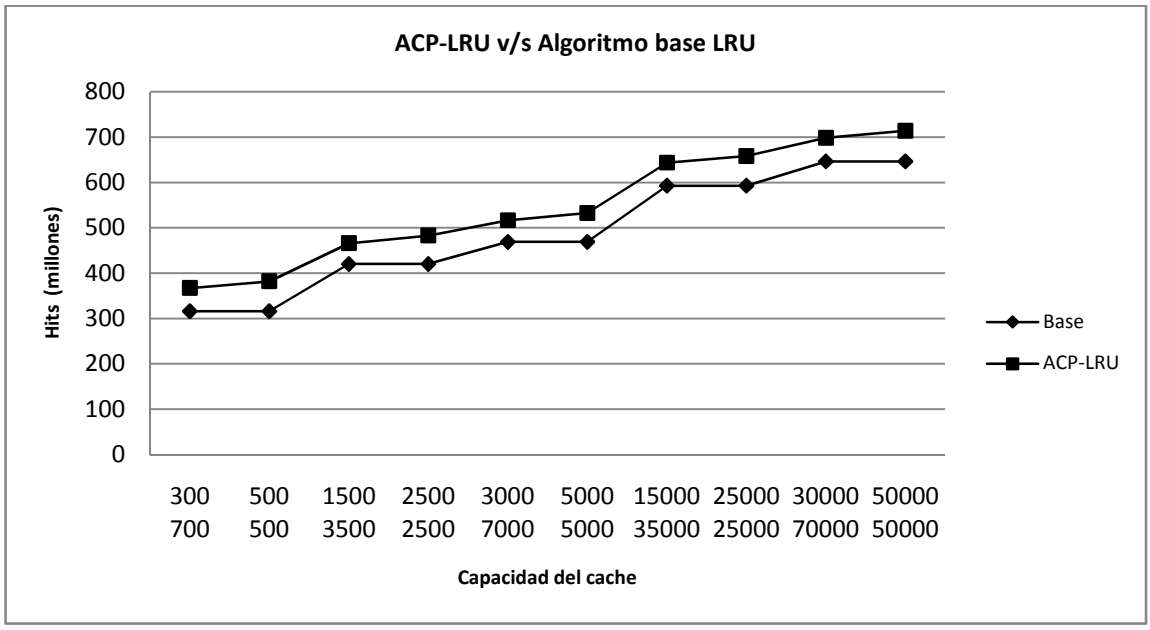

Figura 6.3: Gráfico ACP-LRU v/s Algoritmo base LRU. Fuente: Creación propia.

6.2.2. ACP-LFU v/s algoritmo base LFU

En la tabla 6.4 se muestran los resultados obtenidos de ACP-LFU. La primera columna contiene la cantidad total de entradas del cache, mientras que la columna "Prueba" contiene el tamaño asignado para la sección estática y

para la sección dinámica-ráfaga. En las columnas subsiguientes se observan los hits producidos en cada sección del cache y los hits totales producidos sumando todas las secciones del cache. La última columna corresponde a la cantidad de ráfagas ingresadas en el cache ráfaga.

En cuanto a los resultados obtenidos en la tabla 6.4 se observa que la mayor cantidad de hits en cache se dan en la sección estática, seguida por la sección ráfaga y por último la sección dinámica con valores muy por debajo de los obtenidos por las otras dos secciones. Esto se debe a que la política de desalojo LFU es basada en la frecuencia, por ende al agregar una sección ráfaga y una sección estática, éstas abarcarán la mayor parte de las consultas que son referenciadas con una mayor frecuencia durante su ciclo de vida. Al igual que en la tabla 6.2 de la sección 6.2.1, se observa que a medida que va aumentando la capacidad de la sección estática se detectan una menor cantidad de ráfagas. Esto corresponde a una anomalía de la colección masiva de los datos de muestra y se puede deber a dos motivos: (i) algunas de las consultas almacenadas en el cache estático son ráfagas que fueron ingresadas en el mes anterior y que aún no han terminado su ciclo de vida y continúan con un comportamiento ráfaga en el mes que se realizó la muestra; (ii) existen consultas permanentes que no fueron ingresadas al cache estático y tienen un comportamiento de ráfaga en un instante de tiempo donde son altamente solicitadas por los usuarios.

Cantidad de entradas	Prueba	Hits sección estática	Hits sección ráfaga	Hits sección dinámica	Hits totales	Cantidad de ráfagas detectadas
1000	300-700	199.258.257	147.160.265	42.860.537	**389.279.059**	1280
	500-500	223.217.665	145.413.939	23.998.623	**392.630.227**	1267
5000	1500-3500	288.304.696	136.951.826	31.664.303	**456.920.825**	1203
	2500-2500	319.710.204	132.952.917	16.488.321	**469.151.442**	1166
10000	3000-7000	330.197.707	132.519.505	28.363.014	**491.080.266**	1164
	5000-5000	362.516.504	128.367.641	15.596.977	**506.481.122**	1124
50000	15000-35000	437.282.339	114.180.659	28.781.083	**580.244.081**	1040
	25000-25000	477.166.088	104.025.585	15.844.570	**597.036.243**	979
100000	30000-70000	490.798.000	101.578.659	29.629.758	**622.006.417**	963
	50000-50000	529.612.778	94.110.694	16.484.531	**640.208.003**	912

Tabla 6.4: Resultados ACP-LFU. Fuente: Creación propia.

La tabla 6.5 se muestra los resultados de hits totales obtenidos ACP-LFU v/s la utilización de un algoritmo base LFU. En cada prueba el algoritmo base utiliza la capacidad total del cache, es decir, para la primera prueba "300-700" la capacidad total del cache para el algoritmo base es de 1000. En la última columna se observa la mejora producida por ACP-LFU con respecto al algoritmo base, mejorando la tasa de hits en todos las pruebas. Se observa que en las pruebas donde la capacidad total del cache es el mismo y se asigna una mayor cantidad de espacio de almacenamiento al cache estático, aumenta la tasa de total de hits en cache

Cantidad de entradas	Prueba	Hits arquitectura propuesta	Hits algoritmo base LFU	Mejora (% aprox.)
1000	300-700	389.279.059	245.109.780	**58,8%**
	500-500	392.630.227		**60,1%**
5000	1500-3500	456.920.825	318.637.164	**43,4%**
	2500-2500	469.151.442		**47,2%**
10000	3000-7000	491.080.266	359.577.076	**36,5%**
	5000-5000	506.481.122		**40,8%**
50000	15000-35000	580.244.081	483.713.566	**19,9%**
	25000-25000	597.036.243		**23,4%**
100000	30000-70000	622.006.417	548.808.946	**13,3%**
	50000-50000	640.208.003		**16,6%**

Tabla 6.5: ACP-LFU v/s algoritmo base LFU. Fuente: Creación propia.

En la figura 6.4 muestra el gráfico obtenido con los datos de la tabla 6.5, donde se observa que ACP-LFU, mejora la tasa de hits en cache al ser comparada con los hits obtenidos al hacer uso del algoritmo base LFU, haciendo uso del mismo registro de consultas en ambas estructuras. El mayor porcentaje de mejora se obtiene para los cache de menor tamaño, llegando a un 60% de incremento en su tasa de hits en el cache con capacidad 1000 (500-500). La menor mejora se obtiene para el cache con capacidad de 100.000 entradas (30000-70000), correspondiente a un 13.3% de incremento en su tasa de hits.

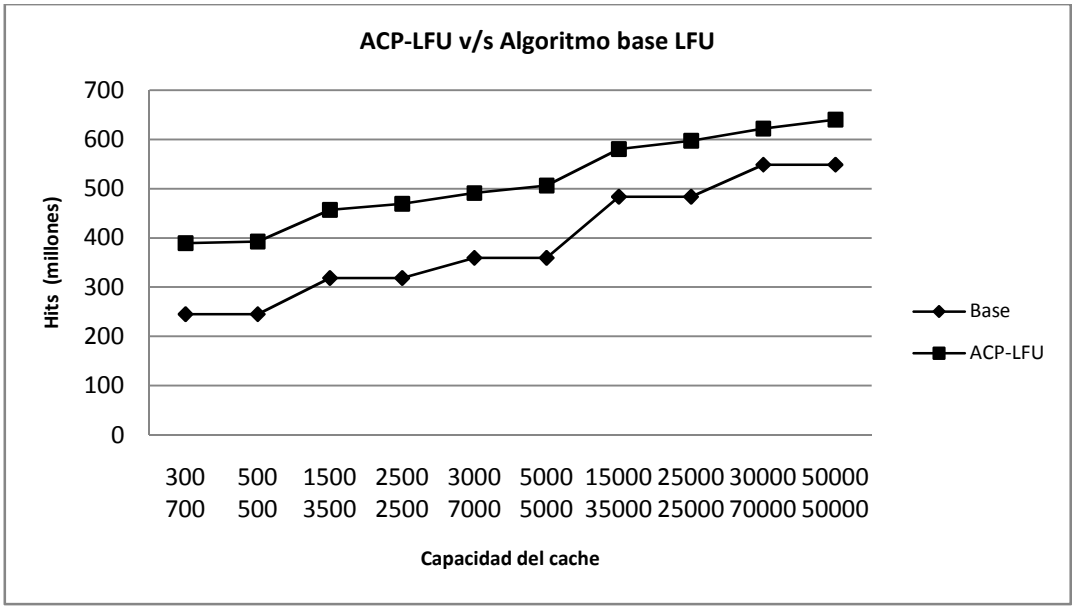

Figura 6.4: ACP-LFU v/s Algoritmo base LFU. Fuente: Creación propia.

6.2.3. Algoritmo LFU v/s Algoritmo LRU

En la figura 6.5 se muestra el gráfico de línea comparativo entre la utilización del algoritmo base LFU con respecto al algoritmo base LRU. Se observa la amplia superioridad en cantidad de hits en cache del algoritmo base LRU sobre LFU, logrando obtener alrededor de 100 millones más de hits en cache con respecto al algoritmo base LFU en todas las pruebas.

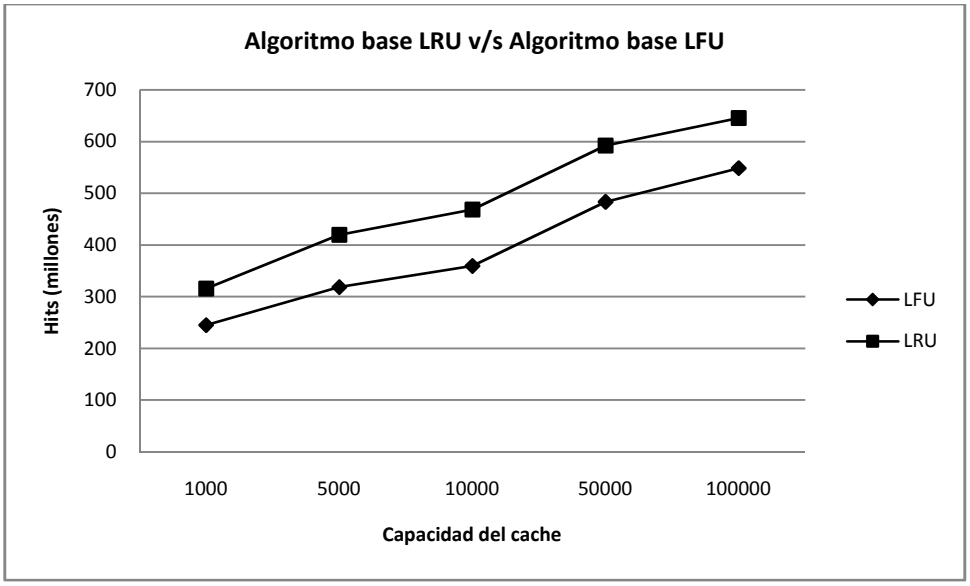

Figura 6.5: Algoritmo base LRU v/s Algoritmo base LFU. Fuente: Creación propia.

En la figura 6.6 se muestra el gráfico comparativo entre ACP-LRU y ACP-LFU. Se observa que el uso de LFU en la arquitectura propuesta solo muestra una pequeña superioridad en las dos primeras pruebas donde el cache tiene una capacidad de almacenaje pequeña. La superioridad (en cuanto a la cantidad de hits totales en cache) de las primeras dos pruebas, se debe a que ACP-LFU obtiene una mayor cantidad de hits en la sección de cache dinámico en comparación con ACP-LRU (ver tablas 6.2 y 6.4), situación que se revierte a medida que aumenta la capacidad del cache. Esto no sucede en el gráfico de la figura 6.5 donde el algoritmo base LRU es muy superior al algoritmo base LFU. Por otra parte, el uso de la política LRU a medida que la capacidad de almacenaje del cache aumenta, muestra una mayor superioridad que la política de desalojo LFU.

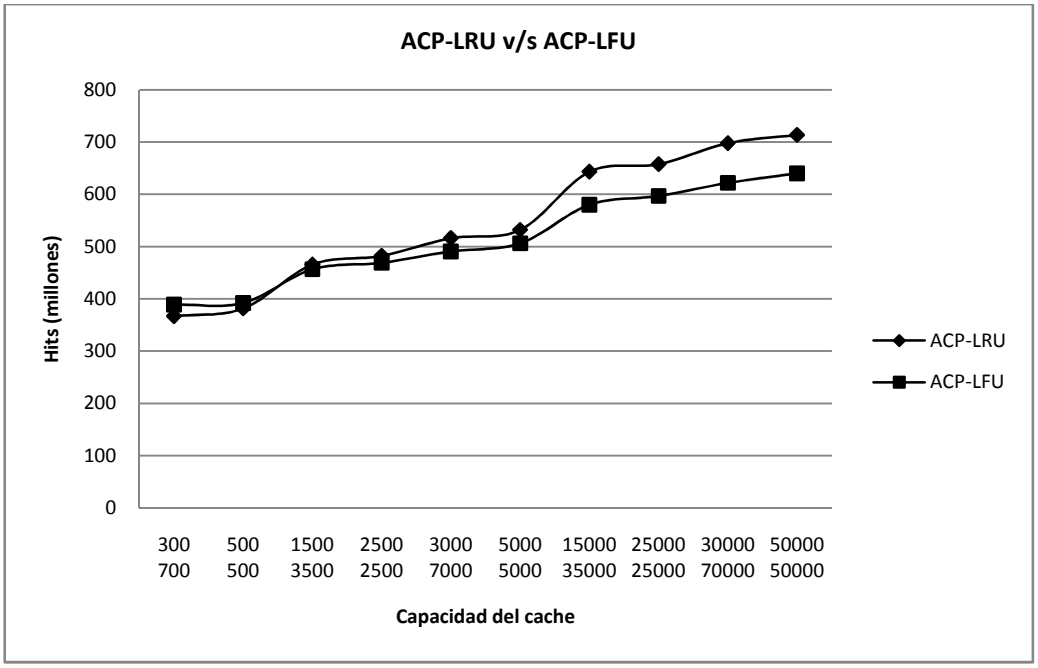

Figura 6.6: Gráfico ACP-LRU v/s ACP-LFU. Fuente: Creación propia.

6.2.4. Resultados finales

En la tabla 6.6 se muestra un resumen de los resultados obtenidos (en cantidad total de hits en cache) de las pruebas realizadas para ACP-LRU y ACP-LFU con respecto al algoritmo base correspondiente. Se observa que la arquitectura cache propuesta presenta una mejora (medida en cantidad de hits totales en cache) para cada una de las pruebas realizadas en comparación a su algoritmo base.

Cantidad de entradas	Prueba	Hits totales ACP-LRU	Hits algoritmo base LRU	Hits totales ACP-LFU	Hits algoritmo base LFU
1000	**300-700**	**367.032.796**	315.626.558	**389.279.059**	245.109.780
	500-500	**381.959.752**		**392.630.227**	
5000	**1500-3500**	**465.934.688**	419.790.612	**456.920.825**	318.637.164
	2500-2500	**482.711.207**		**469.151.442**	
10000	**3000-7000**	**516.435.903**	468.631.609	**491.080.266**	359.577.076
	5000-5000	**532.336.704**		**506.481.122**	
50000	**15000-35000**	**643.558.191**	592.507.547	**580.244.081**	483.713.566
	25000-25000	**657.911.997**		**597.036.243**	
100000	**30000-70000**	**698.139.963**	646.044.741	**622.006.417**	548.808.946
	50000-50000	**713.841.422**		**640.208.003**	

Tabla 6.6: Resultados finales de ACP-LRU y ACP-LFU. Fuente: Creación propia.

6.2.5. Modificación del umbral

Se hicieron pruebas especiales para cierta capacidades del cache. Se modificó el umbral del mecanismo de detección de ráfaga a 50. El umbral corresponde a la frecuencia aproximada mínima que debe tener una consulta para ser catalogada como ráfaga. Esto significa que se obtendrá una mayor cantidad de consultas catalogadas como ráfaga y la posible reinserción de otras peticiones que mantienen el comportamiento de ráfaga durante su ciclo de vida.

En el caso de ACP-LFU se hicieron pruebas que mostraron una mejora en la cantidad de hits en cache. La tabla 6.7 muestra la cantidad de hits totales y la diferencia porcentual aproximada, que se obtiene al utilizar un umbral de 50 y 100 en el mecanismo de detección de ráfaga. Se observa que existe una

pequeña mejora al disminuir el umbral a 50 en ACP-LFU, la que va disminuyendo a medida que la capacidad del cache va aumentando.

Cantidad de entradas	Prueba	Hits ACP-LFU umbral 100	Hits ACP-LFU umbral 50	Diferencia (% aprox.)
5000	1500-3500	456.920.825	459.795.515	0,63
	2500-2500	469.151.442	471.931.950	0,59
10000	3000-7000	491.080.266	493.419.846	0,48
	5000-5000	506.481.122	508.769.434	0,45
50000	15000-35000	580.244.081	582.111.817	0,32
	25000-25000	597.036.243	598.806.406	0,30
100000	30000-70000	622.006.417	623.470.528	0,24
	50000-50000	640.208.003	641.558.778	0,21

Tabla 6.7: ACP-LFU umbral 100 vs umbral 50. Fuente: Creación propia.

En el caso de ACP-LRU se hizo una prueba correspondiente a 30000-70000 de capacidad en el cache. En la tabla 6.8 se observa que la mejora es de 1288 hits más en cache, que es una cifra muy por debajo con respecto al uso en ACP-LFU.

Cantidad de entradas	Prueba	Hits ACP- LRU umbral 100	Hits ACP-LRU umbral 50
100000	30000-70000	698.139.963	698.141.251

Tabla 6.8: Arquitectura cache propuesta (LRU) umbral 100 vs umbral 50. Fuente Creación propia.

6.2.6. Comportamiento de las consultas ráfaga y tiempo de vida

En las figuras 6.7 y 6.8 se observa el comportamiento de dos consultas en ráfagas obtenidas de las pruebas realizadas. En ambas figuras se observa el ciclo de vida de las consultas en ráfaga del log de pruebas. Ambas consultas comienzan con un alza abrupta y es solicitada por los usuarios durante un periodo de tiempo acotado. Los intervalos que se muestran en los gráficos tienen una hora de diferencia entre cada uno y no corresponde al intervalo real en el que entra o es desalojada la consulta en ráfaga. En nuestro caso un intervalo real es de cada 60 segundos.

El mecanismo de detección de ráfaga ingresó la consulta "*kofi annan resigns*" (ver figura 6.7) en el intervalo 5 (2188 real) a la sección ráfaga, siendo desalojada de la sección ráfaga, por la política de desalojo del cache ráfaga entre los intervalos 28 y 30, se observa que después del tiempo de eliminación la consulta siguió siendo solicitada por los usuarios con una menor frecuencia, llegando a ser nula al final de su ciclo de vida.

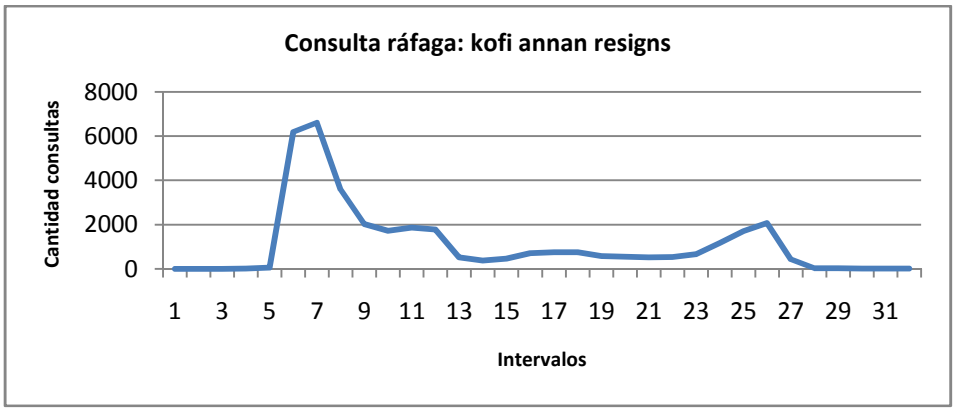

Figura 6.7: Consulta en ráfaga "kofi annan resigns". Fuente: Creación propia.

En el caso de la consulta ráfaga "*aussie rower sent home*" (ver figura 6.8), el mecanismo de detección de ráfaga ingreso la consulta en la sección ráfaga en el intervalo 4, y fue desalojada del cache ráfaga entre los intervalos 7 y 30 (dependiendo del tamaño del cache y los top-*k*). Se observa que entre los intervalos 6 y 18 bajan las peticiones realizadas por los usuarios hacia esta consulta volviendo a aumentar su interés en el intervalo 19 hasta declinar el interés por los usuarios con respecto a este tópico en el intervalo 30.

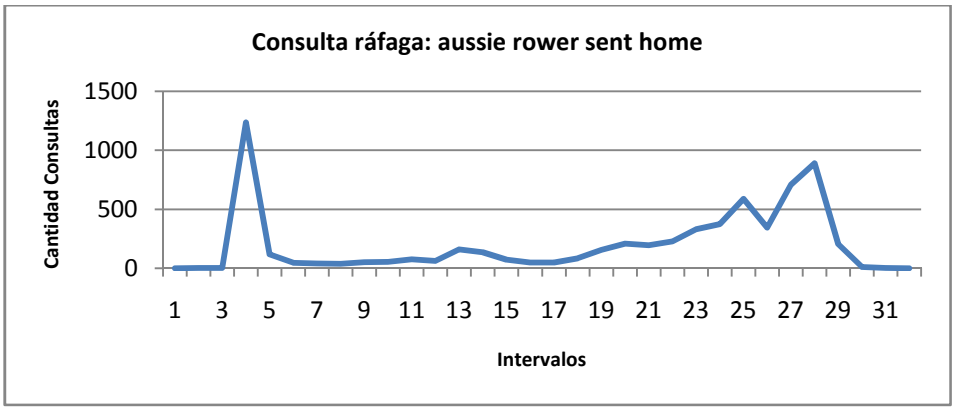

Figura 6.8: Consulta en ráfaga "aussie rower sent home". Fuente: Creación propia.

En la tabla 6.9 se muestra el intervalo en que son desalojadas las consultas en ráfaga nombradas anteriormente, entre paréntesis se encuentra el intervalo de tiempo real de su desalojo.

Cantidad de entradas	Prueba	Intervalo elimina "kofi annan resigns"	Intervalo elimina "aussie rower sent home"
1000	300-700	30 (3606)	7 (4076)
	500-500	30 (3606)	7 (4035)
5000	1500-3500	28 (3512)	30 (5430)
	2500-2500	28 (3512)	8 (4104)
10000	3000-7000	30 (3606)	30 (5430)
	5000-5000	30 (3606)	8 (4104)
50000	15000-35000	30 (3606)	7 (4076)
	25000-25000	28 (3501)	30 (5412)
100000	30000-70000	30 (3606)	7 (4076)
	50000-50000	30 (3606)	7 (4035)
100000	TTL = 120 30000-70000	33 (3796)	31 (5516)

Tabla 6.9: Intervalo de eliminación de consultas ráfaga. Fuente: Creación propia.

La consulta en ráfaga "*kofi annan resigns*" fue desalojada entre los intervalos reales 3501 y 3606 existiendo aproximadamente 76 consultas realizadas por los usuarios entre estos intervalos. En este caso la consulta se mantuvo durante el transcurso de su tiempo de vida en cache sin perder hits al ser desalojada antes del cache. Por otro lado, la consulta "*ausssie rower sent home*" fue desalojo en las distintas pruebas entre los intervalos 4035 y 5430 (alrededor de 23 horas de diferencia entre cada desalojo) con 5071 referencias aprox. hacia la consulta entre esos intervalos. La diferencia en los intervalos de desalojo de las consultas del cache ráfaga para cada prueba, depende directamente del tamaño asignado al cache y la obtención de las top-*k* consultas más frecuentes del mecanismo de detección de ráfaga. Esto quiere decir que la consulta en ráfaga no es almacenada en las top-*k* consultas más frecuente durante un intervalo mínimo de 60 minutos, tiempo necesario para

que se dé por finalizado el tiempo de vida de la consulta en ráfaga y sea desalojada de la sección ráfaga por la política de desalojo del cache ráfaga. En la última fila de la tabla muestra una prueba extraordinaria realizada para el cache con capacidad 30000-70000 que consiste en asignar el doble tiempo de vida a la política de desalojo de ráfaga (TTL = 120). Se observa que la consulta "aussie rower sent home" fue desalojada en el intervalo 5516 en vez del intervalo 4076 obteniendo 5114 hits mas en el cache ráfaga por esta consulta. Hay que recordar que una vez que la consulta es desalojada de la sección ráfaga, esta es ingresada a la sección dinámica, por lo que la consulta puede seguir generando hits en cache durante su tiempo de vida, esto depende directamente de la política de desalojo utilizada en la sección dinámica.

En cuanto al desalojo temprano de consultas de la sección ráfaga, se realizaron pruebas asignando distintos tiempos de vida en la política de desalojo. Estas pruebas se realizaron para ACP-LRU. La capacidad del cache para estas pruebas es de 30000-70000. En la tabla 6.10 se observa que a medida que se le asigna un mayor tiempo de vida a las consultas pertenecientes al cache ráfaga, aumenta la cantidad de hits en cache. La última prueba muestra la cantidad de hits ocasionados en cache que se obtienen al no eliminar del cache ráfaga las consultas. Si bien, la cantidad de hits aumenta en 170.217 hits con respecto a un tiempo de vida de 60 minutos, esto podría ser un error puesto que se mantendría en cache las consultas en ráfaga que una vez que terminan su ciclo no ocasionaran hits en cache, desperdiciando espacio para que ingrese una nueva consulta en la sección dinámica.

Prueba	Hits ACP-LRU
TTL = 60	698.139.963
TTL = 120	698.162.879
TTL = 180	698.184.367
TTL = 240	698.201.383
TTL = 300	698.214.481
TTL = 360	698.226.686
TTL = 420	698.239.558
TTL = 480	698.248.601
TTL = 540	698.255.994
Sin eliminar consultas ráfagas del cache	698.310.180

Tabla 6.10: Diferentes tiempos de vida asignado a consultas ráfaga en cache. Fuente: Creación propia.

La asignación de un tiempo de vida es importante para la generación de hits en cache durante el ciclo de vida de la consulta en ráfaga, puesto que es una petición que va a ser referenciada un número considerable de veces antes de que lo usuarios pierdan el interés sobre ese tópico. El tiempo de vida asignado en la política de desalojo de consultas en ráfaga puede diferir entre las aplicaciones web en que sea utilizada la arquitectura propuesta, siendo menor en aplicaciones web donde se tiene conocimiento a priori de su corta duración o mayor en caso contrario.

Capítulo 7

7. Conclusiones

En este trabajo se han estudiado las arquitecturas cache propuestas en la literatura, junto con las políticas de desalojo, eliminación de resultados antiguos y de admisión existentes para mantener el funcionamiento del cache. Además, la ley de Zipf nos dice que no todos los términos tienen el mismo impacto en una comunidad, existiendo a nivel de la web, peticiones que son altamente referenciadas en comparación con otras que tienen una muy baja ocurrencia, incluso llegando a consultas que son referenciadas solo una única vez.

En cuanto al comportamiento del usuario, la variación en el tiempo y la frecuencia en que los usuarios solicitan distintos tópicos permite la clasificación de las consultas. Entre los distintos tipos de clasificación se encuentran las consultas permanentes, las consultas en ráfaga y las consultas variables. El alto impacto producido por las consultas en ráfaga en aplicaciones web de gran escala, durante su ciclo de vida da pie a que sean altamente referenciadas en un tiempo acotado.

En este trabajo se propone una arquitectura cache para aplicaciones web de gran escala basada en el comportamiento del usuario, con el fin de mejorar la eficiencia del cache con respecto a los algoritmos base utilizados en el desalojo de entradas del cache. Para esto, se hizo enfoque en las consultas en ráfaga, permanentes y variables.

La arquitectura cache propuesta consiste en dividir el cache en tres secciones en que cada sección tendrá sus políticas de desalojo, eliminación de resultados antiguos y de admisión, las que serán diferentes para cada una de ellas. En una sección se dará lugar al almacenamiento de consultas con características permanentes, las segunda sección contiene las consultas en ráfaga y por último una sección encargada de manejar las consultas variables. Estas dos últimas secciones comparten el espacio de almacenamiento, esto quiere decir que si una consulta entra a la sección ráfaga, la sección dinámica debe disminuir su espacio en uno, desalojando una entrada del cache dinámico en el caso que fuera necesario.

Las consultas en ráfaga tienen un comportamiento especial, por este motivo deben ser tratada de distinta forma que las otras consultas. Para esto, se utilizó el mecanismo de detección de ráfaga eficiente en tiempo y en espacio propuesto en el trabajo de Gómez Pantoja (2014). El mecanismo fue utilizado para el ingreso temprano de consultas catalogadas como ráfaga y así tener conocimiento a priori de las peticiones que poseen estas características, las que serán mantenidas en la sección ráfaga hasta disminuir la frecuencia en la que es referenciada. Siguiendo este contexto, se decidió crear una políticas de desalojo del cache ráfaga, con el fin de mantener esta consulta en la sección ráfaga durante su tiempo de vida y así mejorar la eficiencia del cache al no ser desalojadas antes de tiempo.

En el transcurso del experimento se observó que el ingreso temprano de las consultas en ráfaga al cache y su mantención en el mismo durante su ciclo de vida, aumenta la tasa de hit en cache lo que se traduce en la disminución en el acceso al back-end para dar respuesta a esta petición.

Por otro lado, en los experimentos realizados las consultas permanentes ingresadas a la sección estática de forma off-line, corresponden a las consultas

más frecuentes obtenidas del mes anterior en que se realizaron las pruebas. En nuestro caso se usaron las consultas más frecuentes del mes de julio del año 2012 de un log de consultas del motor de búsqueda. En un inicio la sección ráfaga y dinámica se encuentran vacías.

Los experimentos se realizaron para diferentes tamaños del cache (1000-5000-10000-50000-10000), donde se asignó una cantidad de almacenamiento para la sección estática y una para la sección dinámica que comparte el espacio con la sección ráfaga. Para cada prueba se realizaron dos pruebas más: en la primera se asigna un 30% de capacidad para el cache estático y un 70% de capacidad para el cache dinámico-ráfaga; y en la segunda se asigna un 50% de capacidad para la sección estática y un 50% de capacidad para la sección dinámica-ráfaga. Además, se realizaron pruebas separadas de la arquitectura propuesta utilizando políticas de desalojo distintas para el cache dinámico correspondientes a LRU y LFU, lo que dio un total de 20 pruebas realizadas. El log de consultas utilizados en la pruebas corresponden al mes de agosto del año 2012 del motor de búsqueda.

Los resultados obtenidos para la arquitectura cache propuesta utilizando LRU como política de desalojo del cache dinámico, muestran una mejora promedio en el total de las pruebas del 10,87% en cantidad de hits en cache con respecto al algoritmo base LRU. Dando mejores resultados en los casos donde la división del cache es de un 50% para cada sección.

Por otro lado, los resultados obtenidos para la arquitectura cache propuesta utilizando LFU como política de desalojo del cache dinámico, muestran una mejora promedio del total de las pruebas de un 36% en cantidad hits en cache con respecto al algoritmo base LFU. Dando mejores resultados en los casos donde la división del cache es de un 50% para cada sección. Además, es la arquitectura obtuvo los mejores resultados con respecto a su

algoritmo base (en comparación con LRU), llegando a obtener un 58,8% y un 60% de mejora en las 2 prueba realizada con capacidad de almacenaje del cache de 1000.

En ambas arquitecturas se obtuvo mejores resultados para los cache con menor capacidad disminuyendo su eficiencia a medida que aumenta la capacidad total de almacenaje del cache. Esto hace ideal el uso de la arquitectura propuesta para aplicaciones web de gran escala que utilice dispositivos cache con pequeña capacidad de almacenaje.

En lo referente a los objetivos del proyecto de título en la tabla 7.1 se muestra los objetivos específicos del proyecto de titulo, la situación actual de cada objetivo, el resultado esperado, la métrica y por último el criterio de éxito que se debe cumplir para determinar el cumplimiento del objetivo.

En el primer objetivo específico se determinaron los componentes estructurales presentes en el comportamiento del usuario de aplicaciones web de gran escala correspondiente a las consultas de tipo ráfaga, permanentes, periódicas y variables. Por lo tanto, el número de componentes es mayor a 2, cumpliendo el criterio de éxito propuesto.

En El segundo objetivo específico la arquitectura cache propuesta se desarrolló en base a los componentes estructurales del comportamiento de usuario, al asignar una sección del cache al almacenamiento de consultas permanentes, otra sección para el almacenamiento de consultas en ráfaga y por último una tercera sección para el almacenamiento de consultas variables. El criterio de éxito asignado para el cumplimiento del objetivo se puede observar en la sección 5 del proyecto donde se muestran los distintos diagramas de flujo desarrollados para la arquitectura cache propuesta.

Por último, en el tercer objetivo específico los resultados obtenidos de la arquitectura cache propuesta mejoran la tasa de hit en cache con respecto a los algoritmos base LFU y LRU, en un 36% y un 10,87% del promedio del total de hits en cache de las pruebas realizadas respectivamente. Sin embargo, en la tabla 6.3 se observan dos excepciones que corresponden a un porcentaje de mejora de un 8,6% y 8%, que corresponde a un cache estático de 15000 y 30000 entradas respectivamente. Se recalca que esto corresponde a dos excepciones, ya que en la mayoría de los casos se observa una mejora por sobre el criterio de éxito establecido.

Situación actual	Objetivo específico	Resultado esperado	Métrica	Criterio de éxito
No existe un análisis de los principales patrones existentes para cache de gran escala.	Determinar los componentes estructurales presentes en el comportamiento de usuario de aplicaciones web de gran escala	Un conjunto de componentes estructurales relevantes del comportamiento de usuario presente en aplicaciones web de gran escala	Número de Componentes (NC)	NC ≥ 2
En la literatura existen trabajos que consideran dos componentes estructurales del comportamiento del usuario (permanente y variable)	Proponer una arquitectura cache que considere los distintos componentes estructurales del comportamiento de usuario	Una propuesta cache que considere los patrones relevantes de comportamiento de usuario	Propuesta de arquitectura cache	Diagramas de flujo y de arquitectura cache
Las estrategias básicas más utilizadas corresponden a LRU y LFU.	Mejorar la tasa de hit respecto a una estrategia básica de cache en la literatura	Una mejora porcentual sobre estrategias básicas de cache	Porcentaje de Mejora (PM)	PM ≥ 10%

Tabla 7.1: Objetivos específicos. Fuente: Creación propia.

La hipótesis del proyecto de título, se cumple al proponer una arquitectura que considera el comportamiento de usuario, mejorando la tasa de hits en cache (36% y 10,8%) en relación a una estrategia básica del estado del arte.

7.1. Trabajos futuros

Como trabajo futuro se propone determinar la política de desalojo ideal a utilizar en la sección dinámica según la aplicación web de gran escala donde se utilizará la arquitectura cache propuesta. Esto se debe a que existen políticas de desalojo que tienen un mayor rendimiento según la aplicación en donde se utilice.

Otro trabajo futuro corresponde a determinar el tiempo de vida asignado en la política de desalojo para cache ráfaga según el tamaño del cache y la aplicación web donde sea utilizado. Esto es para aquellas aplicaciones web de gran escala donde las consultas en ráfaga tienen un menor o mayor tiempo de vida según el comportamiento del usuario en dicha aplicación web.

Un posible trabajo futuro es buscar o crear un mecanismo off-line de almacenamientos de consultas permanentes que mejoren la tasa de hit del cache estático. En este trabajo se obtuvieron las consultas permanentes según la frecuencia en que son referenciadas en un mes, lo que puede llevar al ingreso de consultas ráfagas del mes anterior. Un posible mecanismo off-line puede ser aquel que detecte las consultas en ráfaga del mes y no las inserte en cache estático.

Otro posible trabajo futuro es estudiar el funcionamiento de la arquitectura cache propuesta en redes de sensores inalámbrico y el internet de las cosas así como en *smart cities*, que corresponden a varios nodos que "incluyen aplicaciones para dispositivos embebidos enfocados a la automatización de edificios, logísticas, mediciones de personal, o a diversos procesos de medición inteligente" (Cama, De la Hoz, & Cama, 2012).

Bibliografía

Aggarwal, C., Wol, J., & Yu, P. (1999). Caching on the world wide web. Knowledge and Data Engineering. *Knowledge and Data Engineering* , 94 - 107.

Alici, S., Sengor Altingovde, I., Ozcan, R., Cambazoglu, B., & Ulusoy, Ö. (Julio 2011). Timestamp-based result cache invalidation for web search engines. *In Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval* (págs. 973 - 982). Beijing, China: ACM.

Arlitt, M., Friedrich, R., & Jin, T. (1998). Performance evaluation of web proxy cache replacement policies. *Springer Berlin Heidelberg* , 193 - 206.

Baeza-Yates, R., & Jonassen, S. (2012). Modeling static caching in web search engines., (págs. 436-446).

Baeza-Yates, R., Gionis, A., Junqueira, F. P., Murdock, V., & Plachouras, V. (2008). Design trade-offs for search engine caching. *ACM Transactions on the Web (TWEB)* .

Baeza-Yates, R., Gionis, A., Junqueira, F., Murdock, V., Plachouras, V., & Silvestri, F. (2007). The impact of caching on search engines. *In Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval* (págs. 83 - 190). Amsterdam: ACM.

Baeza-Yates, R., Junqueira, F., Plachouras, V., & Witschel, H. F. (2007). Admission Policies for Caches of Search Engine Results. *In String Processing and Information Retrieval* (págs. 74-85). Springer Berlin Heidelberg.

Bahn, H., Koh, K., Noh, S. H., & Lyul Min, S. (2002). Efficient replacement of nonuniform objects in web caches. *Computer* , 65 - 73.

Balamash, A., & Krunz, M. (2004). An overview of web caching replacement algorithms. *Communications Surveys & Tutorials* (págs. 44 - 56). IEEE.

Bhattacharya, A. (2014). CHAPTER 2 WHAT IS SOCIAL NETWORK. En A. Bhattacharya, *Social Network Addiction Part I* (págs. 15-19). McGraw-Hill.

Biswas, S. (10 de Septiembre de 2013). *Digital Indians: Ben Gomes.* Obtenido de BBC: http://www.bbc.com/news/technology-23866614

Blanco, R., Bortnikov, E., Junqueira, F. P., Lempel, R., Telloli, L., & Zaragoza, H. (2010). Caching search engine results over incremental indices. *In Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval* (págs. 82-89). Geneva: ACM.

Blanco, R., Roi, E., Junqueira, F. P., Lempel, R., Telloli, L., & Zaragoza, H. (Julio 2010). Caching Search Engine Results over Incremental Indices. *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval* (págs. 82 - 89). Geneva, Switzerland: ACM.

Bresalau, L., Cao, P., Fan, L., Phillips, G., & Shenker, S. (1999). Web caching and Zipf-like distributions: Evidence and implications. *In INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* (págs. 126 - 134). Los Angeles: IEEE.

Cama, A., De la Hoz, E., & Cama, D. (2012). Las redes de sensores inalámbricos y el internet de las cosas. *INGE CUC* , 165.

Cambazoglu, B., Junqueira, F. P., Plachouras, V., Banachowski, S., Cui, B., Lim, S., y otros. (2010). A Refreshing Perspective of Search Engine Caching. *In Proceedings of the 19th international conference on World wide web* (págs. 181 - 190). North Carolina, USA: ACM.

Cao, P., & Irani, S. (1997). Cost-Aware WWW Proxy Caching Algorithms. *In Usenix symposium on internet technologies and systems* , 193 - 206.

Chen, Z., Yang, H., Ma, J., Lei, J., & Gao, H. (2011). Time-based query classification and its application for page rank. *J Comput Info Sys* , 3149 - 3156.

Correia Saraiva, P., Silva de Moura, E., Ziviani, N., Meira, W., Fonseca, R., & Ribeiro-Neto, B. (2001). Rank-preserving two-level caching for scalable search engines. *In Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval* (págs. 51 - 58). New Orleans: ACM.

Díaz Ramos, A. F., & Orjuela Castillo, J. J. (2014). Diseño de una arquitectura web distribuida de alta disponibilidad para sistemas de educación a distancia por medio de Oracle WebLogic Server. *Tlamati* , 92-95.

Einziger, G., & Friedman, R. (2014). Tinylfu: A highly efficient cache admission policy. *In Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on* (págs. 146-153). IEEE.

Facebook. (28 de Febrero de 2016). Obtenido de Facebook Reports Fourth Quarter and Full Year 2015 Results: http://investor.fb.com/releasedetail.cfm?ReleaseID=952040

Fitzpatrick, B. (2004). Distributed caching with memcached. *Linux journal* , 5.

Gan, Q., & Suel, T. (Abril 2009). Improved techniques for result caching in web search engines. *n Proceedings of the 18th international conference on World wide web* (págs. 431 - 440). Madrid, España: ACM.

Gómez Pantoja, C. L. (Abril de 2014). Servicios de cache distribuidos para motores de búsqueda web. *PhD thesis* . Universidad de Chile.

Jiang, S., & Zhang, X. (2002). LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review* , 31-42.

Kumar, C., & Norris, J. B. (2008). A new approach for a proxy-level web caching mechanism. *Decision Support Systems* , 52-60.

Lella, A. (23 de 02 de 2016). *comScore*. Obtenido de comScore: http://www.comscore.com/Insights/Market-Rankings/comScore-Releases-January-2016-US-Desktop-Search-Engine-Rankings

Long, X., & Suel, T. (2006). Three-level caching for efficient query processing in large web search engines. *World Wide Web* , 369 - 395.

Markatos, E. (2001). On caching search engine query results. *Computer Communications* , 137-143.

Megiddo, N., & Modha, D. S. (2003). ARC: A Self-Tuning, Low Overhead Replacement Cache. *In Proc. of the 2nd USENIX Conf. on File and Storage Technologies* (págs. 115-130). San Franciso: FAST.

Metwally, A., Agrawal, D., & Abbadi, A. E. (2006). An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Transactions on Database Systems (TODS)* , 1095-1133.

Nagaraj, S. V. (2004). WEB CACHING AND ITS APPLICATIONS. En S. V. Nagaraj, *WEB CACHING AND ITS APPLICATIONS* (págs. 24 - 27). Chennai: KLUWER ACADEMIC PUBLISHERS.

Newman, M. E. (2005). Power laws, Pareto distributions and Zipf's law. *Contemporary physics* , 323 - 351.

Niclausse, N., Liu, Z., & Nain, P. (1998). A new efficient caching policy for the World Wide Web. *In Proceedings of the Workshop on Internet Server Performance* , 119 - 128.

Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., C. Li, H., y otros. (2013). Scaling memcache at facebook. *In Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, (págs. 385 - 398).

Ozcan, R., Altingovde, I. S., & Ulusoy, Ö. (2008). Static Query Result Caching Revisited. *In Proceedings of the 17th international conference on World Wide Web* (págs. 1169 - 1170). Beijing: ACM.

Ozcan, R., Altingovde, S., Cambazoglu, B. B., Junqueira, F. P., & Ulusoy, Ö. (2012). A five-level static cache architecture fore web search engines. *Information Processing & Management* , 828 - 840.

Petersen, C., Simonsen, J. G., & Lioma, C. (2015). The Impact of using Combinatorial Optimisation for Static Caching of Posting Lists. *In Information Retrieval Technology* , 420-425.

Podlipnig, S., & Böszörmenyi, L. (2003). A survey of web cache replacement strategies. *ACM Computing Surveys (CSUR)* , 374-398.

Rodriguez, P., Spanner, C., & Biersack, E. W. (2001). Analysis of web caching architectures: hierarchical and distributed caching. *Networking, IEEE/ACM Transactions on* , 404 - 418.

Shi, L., Gu, Z., Wei, L., & Shi, Y. (2006). An applicative study of Zipf's law on web cache. *International Journal of Information Technology* , 49 - 58.

Singh, A., Hussain, M., & Ranjan, R. (2011). Two Level Caching Techniques for Improving Result Ranking. *Bharati Vidyapeeth's Institute of Computer Applications and Management* , 365 - 371.

Skobeltsyn, G., Junqueira, F., Plachouras, V., & Baeza-Yates, R. (2008). ResIn: a combination of results caching and index pruning for high-performance web search engines. *In Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval* (págs. 131-138). Singapore: ACM.

Sosa Sosa, V. J., & Navarro Moldes, L. (2002). Arquitectura para la distribución de documentos en un sistema distribuido a gran escala.

Starobinski, D., & Tse, D. (2001). Probabilistic methods for web caching. *Performance evaluation* , 125 - 137.

Twitter. (24 de Febrero de 2016). *Twitter*. Obtenido de Twitter: https://about.twitter.com/es/company

Vakali, A. I. (2000). LRU-based algorithms for Web cache replacement. *In Electronic Commerce and Web Technologies*, (págs. 409 - 418). Springer Berlin Heidelberg.

Wang, J. (1999). A survey of web caching schemes for the internet. *ACM SIGCOMM Computer Communication Review* , 36 - 46.

Williams, S., Abrams, M., Standridge, C. R., Abdulla, G., & Fox, E. A. (1996). Removal policies in network caches for World-Wide Web documents. *In Acm sigcomm computer communication review* , 293 - 305.

Wong, K.-Y. (2006). Web cache replacement policies: a pragmatic approach. *Network, IEEE* , 28 - 34.

Wooster, R. P., & Abrams, M. (1997). Proxy caching that estimates page load delays. *Computer Networks and ISDN Systems,* , 977 - 986.

Zhang, W., He, H., & Ye, J. (2013). A two-level cache for distributed information retrieval in search engines. *The Scientific World Journal* .

Zhou, W., Li, R., & Xinhua, D. (2015). An Intersection Cache Based on Frequent Itemset Mining in Large Scale Search Engines. *In Hot Topics in Web Systems and Technologies (HotWeb)* (págs. 19 - 24). IEEE.

Anexos

Anexo 1 : Planificación utilizada en el proyecto de título.

Id	Nombre de tarea	Duración	Comienzo	Fin	Nombres de los recursos
1	Definición de tema	1 hora	mié 17-02-16	mié 17-02-16	DBR
2	**Fase 1**	**46 días**	**lun 22-02-16**	**lun 25-04-16**	**DBR**
3	**Marco teórico**	**13 días**	**lun 22-02-16**	**mié 09-03-16**	**DBR**
4	Definicón aplicación web de gran escal	2 horas	lun 22-02-16	lun 22-02-16	DBR
5	Definicón de motor de búsqueda	2 horas	lun 22-02-16	lun 22-02-16	DBR
6	Definicón de Redes sociales	2 horas	mié 24-02-16	mié 24-02-16	DBR
7	Definicón de Caché	1 día	jue 25-02-16	jue 25-02-16	DBR
8	Definicón de organización del caché	1 día	vie 26-02-16	vie 26-02-16	DBR
9	Definición de polîticas de caché	1 día	lun 29-02-16	lun 29-02-16	DBR
10	Definición de las estrucutras del caché	1 día	mar 01-03-16	mar 01-03-16	DBR
11	Clasificiación del caché	1 día	mié 02-03-16	mié 02-03-16	DBR
12	Definición ley de Zip	3 días	jue 03-03-16	lun 07-03-16	DBR
13	Definición de tipo de peticiones de usua	2 días	mar 08-03-16	mié 09-03-16	DBR
14	**Estado del Arte**	**33 días**	**jue 10-03-16**	**lun 25-04-16**	**DBR**
15	Definición de arquitecturas cache	3 días	jue 10-03-16	lun 14-03-16	DBR
16	Revisión bibliografica de arquitecturas c	15 días	mar 15-03-16	lun 04-04-16	DBR
17	**Revision biblográfica de Politicas pa**	**15 días**	**mar 05-04-16**	**lun 25-04-16**	**DBR**
18	Politicas de reemplazo	7 días	mar 05-04-16	mié 13-04-16	DBR
19	Politicas de admisión	4 días	jue 14-04-16	mar 19-04-16	DBR
20	Politicas de eliminación de resultado	4 días	mié 20-04-16	lun 25-04-16	DBR
21	**Fase 2**	**11 días**	**mar 26-04-16**	**mar 10-05-16**	**DBR**
22	**Descripción del problema**	**6 días**	**mar 26-04-16**	**mar 03-05-16**	**DBR**
23	Contextualización	1 día	mar 26-04-16	mar 26-04-16	DBR
24	Enunciado del problema	1 día	mié 27-04-16	mié 27-04-16	DBR
25	Objetivos e hipotesis	2 días	jue 28-04-16	vie 29-04-16	DBR
26	Metodologia de trabajo	1 día	lun 02-05-16	lun 02-05-16	DBR
27	Alcances	1 día	mar 03-05-16	mar 03-05-16	DBR
28	**Introduccion**	**5 días**	**mié 04-05-16**	**mar 10-05-16**	**DBR**
29	Motivacion	1 día	mié 04-05-16	mié 04-05-16	DBR
30	Antecedebtes preliminares	2 días	jue 05-05-16	vie 06-05-16	DBR
31	Contribución de la tesis	1 día	lun 09-05-16	lun 09-05-16	DBR
32	Estructura de la tesis	1 día	mar 10-05-16	mar 10-05-16	DBR
33	**Fase 3**	**55 días**	**mié 11-05-16**	**mar 26-07-16**	**DBR**
34	**Descripción de la solución**	**55 días**	**mié 11-05-16**	**mar 26-07-16**	**DBR**
35	Definición de la arquitectura propuesta	15 días	mié 11-05-16	mar 31-05-16	DBR
36	Desarrollo de la solución	40 días	mié 01-06-16	mar 26-07-16	DBR
37	**Fase 4**	**23 días**	**mié 27-07-16**	**vie 26-08-16**	**DBR**
38	**Pruebas y resultados**	**21 días**	**mié 27-07-16**	**mié 24-08-16**	**DBR**
39	Definición de pruebas	1 día	mié 27-07-16	mié 27-07-16	DBR
40	Obtención de resultados	20 días	jue 28-07-16	mié 24-08-16	DBR
41	**Consclusiones**	2 días	jue 25-08-16	vie 26-08-16	DBR
42					
43	Reuniones Con CGP	25horas	mar 02-02-16	mar 23-08-16	DBR,CGP
44	Revisión del documento	2 días	vie 01-04-16	lun 29-08-16	DBR