UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

SERVICIOS DE CACHE DISTRIBUIDOS PARA MOTORES DE BÚSQUEDA WEB

TESIS PARA OPTAR AL GRADO DE
DOCTOR EN CIENCIAS MENCIÓN COMPUTACIÓN

CARLOS LUIS GÓMEZ-PANTOJA

PROFESOR GUÍA:
GONZALO NAVARRO BADINO

PROFESOR CO-GUÍA:
MAURICIO MARÍN CAIHUÁN

MIEMBROS DE LA COMISIÓN:
BENJAMÍN BUSTOS CARDENAS
CARLOS CASTILLO OCARANZA
LUIS MATEU BRULÉ

ESTE TRABAJO HA SIDO FINANCIADO PARCIALMENTE POR CONICYT

SANTIAGO DE CHILE
ABRIL 2014

Resumen

Los Motores de Búsqueda Web (WSEs) actuales están formados por cientos de nodos de procesamiento, los cuales están particionados en grupos llamados *servicios*. Cada servicio lleva a cabo una función específica, entre las que se destacan: (i) Servicio de *Front-End*; (ii) Servicio de Cache; y (iii) Servicio de Índice. Estos servicios forman el núcleo del sistema de procesamiento de consultas. El presente trabajo de tesis se focaliza en el diseño e implementación de servicios de cache distribuidos.

Varios aspectos del sistema y el tráfico de consultas deben ser considerados en el diseño de servicios de cache eficientes: (i) distribuciones sesgadas de las consultas de usuario; (ii) nodos que entran y salen de los servicios (de una forma planificada o súbitamente); y (iii) la aparición de consultas en ráfaga. Dada la arquitectura que se emplea en este trabajo, el Servicio de Cache es el más expuesto a los problemas mencionados, poniendo en riesgo la tasa de *hit* de este servicio clave y el tiempo de respuesta del WSE.

Este trabajo ataca los problemas mencionados anteriormente proponiendo mejoras arquitecturales, tales como un enfoque de balance de carga dinámico para servicios de cache altamente acoplados (desplegados en *clusters*) basados en *Consistent Hashing*, y un esquema para monitoreo y distribución de consultas frecuentes. El mecanismo de balance de carga propuesto es una nueva solución al problema de balance de carga en *clusters* de computadores que corren aplicaciones manejadas por los datos (*data-driven*). Además, se estudia cómo predecir la aparición de consultas en ráfaga para tomar acciones correctivas antes de que saturen o colapsen algunos nodos. Finalmente, se adopta la idea de un sistema tolerante a fallas para proteger información valiosa obtenida a través del tiempo. La idea fundamental es replicar algunas entradas de cache entre distintos nodos para que sean usados en caso de fallas.

Abstract

Current Web Search Engines (WSEs) comprise hundreds of processing nodes, which are partitioned into groups called *services*. Each service carries out a very specific task, among which we can highlight: (i) Front-End Service, (ii) Caching Service, and (iii) Index Service. These services form the core of the query processing system. The present thesis work focuses on the design and implementation of distributed caching services.

Several aspects of the system and query traffic must be considered in the design of efficient caching services: (i) skewed distributions of user queries, (ii) nodes that leave and join the system (in a planned and/or sudden fashion), and (iii) the appearance of bursty queries. Given the architecture we employ, the Caching Service is the most exposed component to the problems above, jeopardizing the hit rate of this key service and the response time of the WSE.

We attack the previous problems proposing several architectural improvements such as a dynamic load balancing approach for highly-coupled Caching Services based on Consistent Hashing, and a scheme for tracking and spreading the load of frequent permanent queries. The dynamic load balancing approach is a new solution to the problem of balancing load in clusters of computers that run data-driven applications. Also, we study how to predict the emergence of bursty queries to spread their load before they saturate or collapse some nodes. Finally, we adopt the idea of a fault-tolerant system to protect valuable information gathered through time. The fundamental idea is to replicate some selected information among nodes to be used in case of node failures.

A Kathy y Rafaela.

Agradecimientos

En estas últimas páginas escritas de la tesis quiero agradecer a las múltiples personas y organizaciones que de una u otra forma contribuyeron a su desarrollo.

Primero que todo, quiero agradecer infinitamente a mi esposa **Kathy** y mi hija **Rafaela**, quienes me regalaron mucho de su tiempo para el desarrollo de este trabajo. Ellas sin duda son los pilares de este trabajo debido a su amor y apoyo constante. Gracias por la compañía, la tolerancia y por entenderme.

Quiero agradecer profundamente a mis padres Ingrid y Antonio, a mis hermanos Gonzalo, Antonio y Vanessa, y a mi sobrino Lionel, quienes me soportaron en mis malos ratos e hicieron este desafío más fácil y alegre. También quiero agradecer a mis tíos Rosa y Mauricio, a mis primos, en especial a Cristian, a mis suegros Margarita y Luis, y a su hija Isasku, a Jocelyn y Juan, y sus hijas Sofía y Fernanda. Todo este círculo cercano me alentó constantemente con su preocupación y aguantó durante bastante tiempo mis largas ausencias.

A **Gonzalo Navarro** y **Mauricio Marín**, mis profesores guía de tesis, quienes me aceptaron como estudiante doctoral y me apoyaron constantemente, tanto en el aspecto académico como en aspectos personales. Ambos dejaron importantes huellas en mi formación académica y profesional.

Durante la estadía en el laboratorio de Yahoo! Research Latin America conocí mucha gente y debo un agradecimiento a todos ellos también. A Mauricio Marín por permitirme participar como estudiante doctoral y desarrollar mi tesis allí. A Sarita Quiñones por su alegría, paciencia y múltiples ayudas. También hice buenos amigos como Alonso Inostrosa y su esposa Carolina Anabalón, Andreas Eiselt, Emir Muñoz, Pablo Torres, Alejandro Figueroa y Roberto Solar. Todos ellos hicieron de mi estadía en el laboratorio una grata experiencia, llena de conversaciones y discusiones

en múltiples ámbitos.

A la gente del DCC, quienes siempre me guiaron y apoyaron en este trayecto, en especial a Angélica Aguirre, y también a los académicos que dejaron en mi formación una huella importante. Quiero destacar a Gonzalo Navarro, Claudio Gutiérrez, Éric Tanter y Ricardo Baeza-Yates.

Quiero agradecer a los miembros de la comisión, Gonzalo Navarro, Mauricio Marín, Benjamín Bustos, Luis Mateu y en especial a Carlos Castillo, por sus valiosas correcciones y comentarios que hicieron posible la mejora del documento final de tesis.

Finalmente, mis agradecimientos a CONICYT por las múltiples becas otorgadas y a la Universidad Andrés Bello por permitirme compatibilizar el trabajo con el desarrollo de la tesis.

Espero no dejar de lado a nadie. A todos ellos, gracias.

Índice general

Índice de Tablas

Índice de figuras

Introducción

1.1. Motivación y Objetivos de la Tesis

Los Motores de Búsqueda Web (*Web Search Engines*, WSEs) son una herramienta fundamental en la sociedad, presente en casi toda actividad que involucre la búsqueda de información. La idea de un WSE involucra tres conceptos fundamentales: (i) motor, (ii) búsqueda y (iii) La Web. El concepto (i) *Motor* se refiere, en un contexto genérico, a una entidad que desarrolla tareas específicas; el concepto (ii) *Búsqueda* consiste en encontrar información relacionada con un tópico en particular; y el concepto (iii) *La Web*, o más específicamente *World Wide Web*, se refiere a la Red Mundial de Información que contiene sitios Web compuestos por documentos y enlaces entre ellos, y accesible a través de Internet. En conclusión, los WSEs son sistemas computacionales que "almacenan" La Web y en que los usuarios realizan búsquedas sobre esta información almacenada. La respuesta del WSE a una búsqueda es una serie de documentos que la responden de mejor manera siguiendo una función de *ranking*.

Los usuarios de los WSEs ven a éstos como una *caja negra* en que dada una consulta, se obtienen los mejores documentos que la responden siguiendo los criterios del buscador. Las preguntas fundamentales en el funcionamiento de los WSEs son: ¿cómo se obtiene la información de la Web?, ¿cómo se almacena la información de la Web?, ¿cómo se consulta la información almacenada de la Web?, entre otras. A grandes rasgos, la información de la Web se recolecta mediante un proceso llamado *crawling* que consiste en visitar las páginas Web (y sus enlaces a otras páginas) y descargarlas [CGMP98, Cas05, BYCMR05]. Luego, se realiza un proceso de indexación en el cual se utiliza una estructura de datos, conocida como *índice invertido*, para organizar y almacenar el texto de la Web [BYRN99, MRS08]. Finalmente, para responder consultas de usuario se utiliza el índice invertido en conjunto con algoritmos de *ranking* para obtener una respuesta (conjunto de documentos) [CMS10, ZM06].

En general, el elemento principal en cualquier búsqueda es la *consulta*. En el contexto de los WSEs, la consulta está formada por un conjunto de palabras, o palabras clave, acerca de algún(os) tópico(s), tal como el nombre de una celebridad, un sitio Web con alto impacto, una marca, o una pregunta en lenguaje natural. Estas palabras clave deben ser contrastadas con la información almacenada en el WSE. La búsqueda es una operación costosa, que implica la obtención de los documentos en los cuales aparecen las palabras clave más una operación adicional de ordenamiento (*ranking*). Esta operación de ordenamiento es aplicada según el impacto que tiene cada palabra clave en el documento y a características específicas de los documentos, lo cual también es costoso. Más aún, el número de documentos que almacena un WSE es del orden de millones o miles de millones de documentos [LS06, GS09], lo cual encarece aún más la operación de búsqueda y que hace fundamental la correcta orquestación de los recursos.

En este trabajo se está interesado en WSEs de gran escala, donde se hace imprescindible el uso de múltiples máquinas para hacer frente a un alto volumen de consultas por unidad de tiempo [LS06]. Los WSE deben estar diseñados para, además de responder consultas de usuario, minimizar el tiempo de respuesta de las consultas (latencia), seguir funcionando en caso de fallas de elementos del buscador, almacenar páginas/documentos Web considerando *cobertura* y *frescura*, y utilizar eficientemente los recursos. Las preguntas enunciadas anteriormente son básicas acerca del funcionamiento de los WSEs, pero existen diversas preguntas más complejas relacionadas con la arquitectura de éstos y cómo están diseñados.

Para diseñar un WSE eficiente y que utilice correctamente los recursos, se debe lidiar con un problema de optimización cuya principal función objetivo es lograr el máximo *throughput* (tasa con la que las peticiones finalizan [Sta01]). En el contexto de este trabajo, el *throughput* es medido en consultas finalizadas por segundo. En una primera aproximación, se podrían utilizar la mayor cantidad de recursos computacionales posible, pero esto es inviable. Existen algunas restricciones adicionales que deben ser consideradas para este problema:

1. *Tiempo de Respuesta*: tiempo entre el envío de una petición y la aparición de su respuesta [Sta01]. Aspecto fundamental en el funcionamiento de los WSEs

[BYGJ+07, SSdMZ+01], ya que una incorrecta experiencia de usuario en el uso del buscador, hará que éste deje de utilizar los servicios. Se debe asegurar que el tiempo de respuesta individual de las consultas esté limitado por una cota superior de tiempo r, es decir, se busca que toda consulta sea resuelta antes de r unidades de tiempo. Esta cota superior de tiempo es establecida en la etapa de diseño y tiene relación con la calidad de servicio a proveer.

2. *Throughput*: la importancia radica en la necesidad de procesar altos volúmenes de consultas por unidad de tiempo [BDH03, LS06], por lo que se debe asegurar que el *throughput* del WSE sea como mínimo una cota inferior t_0. Esto tiene relación con la tasa de arribo de consultas al WSE. Suponer que las consultas de usuario llegan a una tasa de arribo t^* consultas por segundo, entonces el WSE debe estar diseñado para $t_0 > t^*$, ya que de otra forma el procesamiento de consultas en el WSE se tornaría un cuello de botella, acumulando consultas y eventualmente colapsando.

3. *Recursos*: se debe minimizar la cantidad de procesadores, tal que la utilización de ellos esté bajo un umbral $u < 1$. Esto principalmente implica mantener los procesadores con una utilización menor al 100 % en régimen. En la práctica, los WSEs están sobre-dimensionados [BH07, GCLIPM12], así están preparados para lidiar con alzas abruptas en el tráfico de consultas.

Entre las herramientas disponibles para asegurar cada uno de los puntos descritos anteriormente, en este trabajo se han estudiado:

- *Balance de Carga*. El desbalance genera condiciones que producen sesgo y alteran las mediciones. Para mitigarlo, herramientas tales como balance dinámico de carga deben ser utilizadas ya que, como es mencionado en [BYGJ+07, BYGJ+08, CJP+10, GS09, BYJ12, WLY+13, LM03, LM04, XO02, SSdMZ+01, AXF+12], el comportamiento de usuario es altamente sesgado (se utilizan pocas palabras/consultas) y dinámico (cambia en función de eventos económicos, sociales y naturales). Adicionalmente, el balance de carga es utilizado cuando existen cambios de estado, principalmente debido a salidas e ingresos, tanto

planificados como no planificados, de los nodos que componen los servicios. Finalmente, si se tiene un sistema balanceado y por ende menos susceptible a colapsos, se podría re-dimensionar la cantidad de recursos utilizados en el procesamiento de consultas. Este re-dimensionamiento implica una disminución de los recursos utilizados y tiene un impacto económico, pero el re-dimensionamiento debe ser tal que no comprometa el tiempo de respuesta.

- *Tolerancia a Fallas.* Los servidores fallan constantemente en aplicaciones de gran escala [LM10]. Los distintos componentes del WSE son vulnerables a las fallas, además alzas bruscas en el tráfico de consultas podrían dejar fuera de servicio algunos componentes por saturación. Se deben utilizar mecanismos que permitan seguir con la operación y que mitiguen en parte las consecuencias ante la aparición de este tipo de eventos. A través del funcionamiento del WSE, se genera información histórica que es útil (y valiosa) en el procesamiento de consultas de usuario, por lo que proveer mecanismos de protección de esta información ayudaría al WSE a mantener su desempeño en caso de fallas. Principalmente, la información protegida se encuentra en forma redundante en el WSE. Con eso se logra que, en el caso de fallas, se siga operando normalmente y con un mínimo de impacto en el desempeño del WSE.

- *Predicción de Alzas Abruptas.* Las alzas abruptas de tráfico son un factor importante y constante a considerar, como se verá en los posteriores capítulos. Las alzas abruptas podrían no ser cubiertas por los mecanismos de balance de carga, por lo que es necesario analizarlas y mitigarlas en caso de aparición. En el contexto de este trabajo, las alzas abruptas son debido a cambios bruscos y limitados en tiempo en las necesidades de información (consultas) de los usuarios [SC10, GS09].

Este trabajo intentará resolver las siguientes preguntas:

1. ¿Cómo está organizado un WSE de gran escala?
2. ¿Cuáles son los problemas experimentados en un WSE de gran escala?

3. ¿Cuáles son las causas que provocan pérdida de eficiencia en un WSE?

4. ¿Cuáles son los mecanismos para reducir o mitigar las causas que generan pérdida de eficiencia en un WSE?

1.2. Contribución de la Tesis

Como se ha mencionado anteriormente, la tesis está dedicada a estudiar mecanismos de balance de carga y tolerancia a fallas para Motores de Búsqueda Web de gran escala, específicamente en los servicios de cache distribuido que son un elemento clave en su funcionamiento. Como contribuciones específicas de este trabajo están las siguientes:

1. Se establecieron las debilidades que podrían ocasionar una degradación en el desempeño de los servicios de cache.

2. Se identificaron problemas de balance de carga en el Servicio de Cache. A partir de estos problemas, se propuso una organización y un esquema para la mitigación del desbalance basado en la repartición dinámica de consultas frecuentes. Además, se ideó un nuevo enfoque de balance dinámico de carga para aplicaciones en las cuales la asignación de tareas es dirigida por los datos (*data-driven tasks*).

3. Se detectó que entre las debilidades de los WSEs, las consultas en ráfaga (o *bursty queries*) son uno de los eventos que tienen más impacto en la degradación del desempeño del Servicio de Cache. La contribución de este trabajo en torno a este tópico, es la propuesta de una solución para detectar tempranamente una consulta en ráfaga en forma eficiente de espacio y tiempo, con el fin de establecer acciones correctivas.

4. Se estudió la característica de tolerancia a fallas dentro en los servicios de cache. Se determinó qué mecanismos son los apropiados para mantener el desempeño ante la ocurrencia de fallas. Se diseñó un esquema de protección de información

relevante, la cual es almacenada en forma redundante y se utiliza ante la falla de nodos.

5. Se enuncia una organización de los componentes y un conjunto de mecanismos que hace al WSE menos proclive a la degradación de rendimiento, debido a variaciones en el comportamiento de las consultas de usuario y fallas en componentes.

La realización de este trabajo de tesis ha dado lugar a las siguientes publicaciones, las cuales tienen relación con los problemas tratados en este trabajo:

1. *A Fault-Tolerant Cache Service for Web Search Engines: RADIC Evaluation* [GPRML12]. En 18° *International European Conference on Parallel Processing* (Euro-Par 2012): 298-310. Tasa de aceptación: 32,9 % (75/228), *h-index*[1]: 41. Tópicos: cache distribuido, balance de carga, tolerancia a fallas.

2. *A Fault-Tolerant Cache Service for Web Search Engines* [GPGCR+12]. En 10° *IEEE International Symposium on Parallel and Distributed Processing with Applications* (ISPA 2012): 427-434. Tasa de aceptación: 35,8 % (54/151), *h-index*: 13. Tópicos: cache distribuido, balance de carga, tolerancia a fallas.

3. *An Evaluation of Fault-Tolerant Query Processing for Web Search Engines* [GPMCB11]. En 17° *International European Conference on Parallel Processing* (Euro-Par 2011): 393-404. Tasa de aceptación: 29,9 % (81/271), *h-index*: 41. Tópicos: tolerancia a fallas.

4. *New Caching Techniques for Web Search Engines* [MGCGP10a]. En 19° *ACM International Symposium on High Performance Distributed Computing* (HPDC 2010): 215-226. Tasa de aceptación: 25,3 % (23/91), *h-index*: 56. Tópicos: cache distribuido.

5. *Load Balancing Distributed Inverted Files: Query Ranking* [GPM08]. En 16° *Euromicro International Conference on Parallel, Distributed and Network-Based*

[1] http://shine.icomp.ufam.edu.br/

Processing (PDP 2008): 329-333. Tasa de aceptación: 40,0 % (56/140), *h-index*: 27. Tópicos: balance de carga.

6. *Load Balancing Distributed Inverted Files* [MG07]. En 9° *ACM International Workshop on Web Information and Data Management* (WIDM 2007): 57-64. Tasa de aceptación: 25,0 % (20/80), *h-index*: 32. Tópicos: balance de carga.

Además, se está pronto a enviar los resultados de la tesis a una revista, cuyo manuscripto está en etapa final de revisión.

1.3. Organización de la Tesis

En el capítulo 2 se mencionan los principales componentes de los WSE, cómo están organizados, qué algoritmos y técnicas se utilizan, y cómo fluye la información entre los componentes para el procesamiento de consultas. En base a esto, se analizan las potenciales fuentes de degradación del desempeño en los WSEs. Entre los principales componentes de un WSE están el Servicio de *Front-End*, el Servicio de Cache y el Servicio de Índice.

En el capítulo 3 se explica cómo funciona el Servicio de Cache y los distintos mecanismos para habilitar este componente en un cluster de procesadores con memoria distribuida. También se estudian los distintos métodos existentes actualmente y se hace un análisis de ellos. Esto sirve como punto inicial para la construcción de un mecanismo que tome como base las mejores características actuales y se propongan mejoras para los puntos débiles.

Luego, en el capítulo 4 se describen los problemas de balance de carga detectados en un Servicio de Cache considerando *logs* de consultas reales, y se establece cuáles serán los principales componentes a estudiar. Además, los problemas detectados se dividen para ser estudiados por separado. Los problemas principales detectados son asignación poco balanceada de carga, consultas en ráfaga y tolerancia a fallas.

Con los problemas detectados en el capítulo anterior, en el capítulo 5 se proponen soluciones para mitigar el impacto de las consultas más frecuentes. La conclusión

principal de este capítulo es que este tipo de consultas debe ser distribuida en múltiples máquinas que absorban su alto tráfico. Cuatro sub-problemas son enunciados: (i) cómo detectar eficientemente estas consultas, (ii) cuántas máquinas necesita cada una de estas consultas, (iii) en cuáles máquinas replicar la consulta, y (iv) en cuál máquina asignar una nueva instancia de estas consultas.

A continuación, en el capítulo 6 se diseña un mecanismo dinámico para balancear la carga de trabajo entre las máquinas que forman un Servicio de Cache. Una asignación de consultas sin considerar este aspecto produce un alto grado de desbalance entre las máquinas, lo que sumado a la aparición de consultas en ráfaga, hace necesaria la aplicación de algoritmos dinámicos de balance de carga que consideren el estado actual de la carga de cada máquina.

Posteriormente, en el capítulo 7 se idea un mecanismo para detectar tempranamente consultas en ráfaga. Este mecanismo funciona en forma eficiente, utiliza poco espacio, y funciona a partir de infraestructura disponible en las grandes aplicaciones Web actuales. Estas consultas aparecen de forma abrupta e impredecible debido a eventos sociales, económicos o naturales. La idea es predecir su aparición y así mitigar su ocurrencia.

En el capítulo 8, se analiza la información sensible de perder ante fallas en los nodos y posteriormente se formula una solución para protegerla. Información sensible se refiere a información histórica que puede ser reutilizada en el futuro. Al ser información reutilizable, se reduce la cantidad de recursos consumidos (CPU, disco y red). Además, dado el sesgo de las consultas de usuario, en el cual pocas consultas tienen una alta frecuencia y muchas consultas tienen poca frecuencia, hacen que la protección de las consultas muy frecuentes tenga un impacto directo en el desempeño del WSE.

Finalmente, en el capítulo 9 se dan las apreciaciones finales de esta tesis y el trabajo futuro.

Arquitectura

2.1. Contexto

A medida que la información disponible en La Web crece, es necesario distribuir las funcionalidades y la información en múltiples nodos. Los Motores de Búsqueda Web (WSEs) están construidos como una colección de servicios desplegados en *clusters* de nodos altamente acoplados. Estos clusters están localizados en *datacenters*, los cuales atienden consultas de usuarios en una zona geográfica bien definida. Este trabajo estudia los WSEs como un conjunto de servicios, cada uno con una función específica, como en [BDH03, FPSO06, LS05, MGCGP10b]. El objetivo es la especialización y optimización de un grupo de nodos en una tarea en particular.

Como se menciona en el capítulo 1, los principales componentes en el procesamiento de consulta, y por ende servicios, en los WSEs son: Servicio de *Front-End* (FS), Servicio de Cache (CS) y Servicio de Índice (IS). Estos servicios deben mapearse en múltiples nodos de los clusters disponibles. Una opción es definir un WSE como un conjunto de nodos y que todos ellos compartan las funcionalidades. Mientras más distintas son las tareas y procesos, más complejo es modelar y analizar el sistema bajo estudio. Además, para nuestro contexto, mientras más variables se deban analizar, más complejo es el proceso de simulación. La mayoría del trabajo existente, considerando el despliegue de un motor de búsqueda en un entorno distribuido, mapea las principales funcionalidades de los WSEs en servicios, y estudia y analiza cada uno de estos servicios en forma aislada [BDH03, BP98, FPSO06, LS05, MGCGP10b, SJPBY08, WLY$^+$13, ZLS08].

Estos servicios tienen actividades bien definidas. El FS recibe consultas de usuario y las rutea entre los otros servicios para generar la respuesta final a las consultas (página HTML). El CS mantiene en memoria respuestas pre-computadas para las

consultas de usuario más importantes. Finalmente, el IS mantiene un índice invertido para calcular las respuestas a las consultas (no presentes en el CS).

2.2. Servicio de *Front-End* (FS)

El FS actúa como un coordinador para obtener las respuestas a las consultas de usuario: (i) recibe consultas de usuario, (ii) pregunta al CS para obtener respuestas pre-computadas a las consultas, y (iii) envía las consultas al IS (sólo si no es encontrada en el CS) para obtener la respuesta usando el índice invertido [BAA+10, GCIPMF13, MGCBS13]. El FS tiene una visión global de todos los nodos involucrados en la resolución de consultas.

Para hacer frente a miles de consultas por segundo, este servicio debe estar distribuido en múltiples máquinas. La forma más simple y eficiente de organizar estas máquinas, es la completa replicación: todas las máquinas asumen la responsabilidad de gestionar las consultas (un nodo que recibe una consulta, la gestiona hasta el final). La distribución de carga entre los nodos replicados del FS es mediante el algoritmo *Round-Robin*. Más aún, como cada tarea consume la misma cantidad de recursos en cada nodo del FS, este enfoque alcanza balance de carga casi perfecto (no hay nodos sobrecargados). Finalmente, este esquema es tolerante a fallas, ya que no existen responsabilidades específicas y, en caso de fallas, el servicio continúa operando normalmente, ya que los nodos restantes absorben la carga del nodo caído.

En general, en los sistemas de gran escala, existen mecanismos para determinar la falla de nodos [VDA+98]. Estos mecanismos implementan distintos protocolos para la detección de fallas en los nodos en forma asíncrona y distribuida, a través del paso de mensajes. La idea principal de estos mecanismos es que una vez que se ha detectado un conjunto de nodos como no disponible, entonces no se realizan más peticiones a ellos. Dado que este servicio tiene la visibilidad de los otros servicios y sus nodos, y gestiona el procesamiento de las consultas, es el lugar apropiado para usar mecanismos de balance de carga. Estos mecanismos no deben sobrecargar el FS, sino se transformaría en el cuello de botella del WSE. Es por esta razón que las soluciones propuestas en este trabajo deben considerar esta arista. Dentro de los

capítulos posteriores se hará un análisis de la sobrecarga en términos de tiempo y espacio necesario en este servicio para su implementación.

2.3. Servicio de Cache (CS)

El Cache es una tecnología clave para los WSEs [BYGJ+07, BYGJ+08, BBJ+10, CA12, LM03, OAU11, OSABC+12, SJPBY08, ZLS08] y, en general, para aplicaciones y servicios Web [Ada10, AJZ11, Fit04, NO08, Saa]. La idea principal del cache es almacenar respuestas pre-computadas a consultas que puedan ser utilizadas en el futuro. Entre las bondades del cache se tienen: (i) reduce la carga del IS; (ii) mejora la utilización de la red de comunicación; (iii) reduce el costo y tiempo necesario para procesar una consulta; e (iv) incrementa el *throughput* [BYGJ+08, BYJ12, BBJ+10, CA12, CJP+10, SJPBY08, XO02].

Una forma de optimizar el Servicio de Cache es con la utilización de aspectos semánticos o respuestas parciales pre-computadas para la resolución de consultas. Este aspecto consiste principalmente en la utilización de información disponible en el cache para resolver un *miss* en forma aproximada. Algunos trabajos en esta área son [FMM09, CRS99]. El Cache también es utilizado en el servicio de índice para acelerar el procesamiento de consultas [GS09, ZLS08, LS05], lo cual se da esencialmente por el almacenamiento en cache de intersecciones parciales y secciones de índice, entre otros. Estos dos aspectos no son considerados en este trabajo.

George Kingsley Zipf [Zip35, Zip49] formuló una ley empírica, denominada "Ley de Zipf", la cual establece que muy pocas palabras son muy frecuentes, mientras que muchas o la mayoría de ellas son raramente usadas. Este comportamiento también ha sido estudiado en el contexto de los WSEs y el resultado más importante es que las consultas de usuario siguen distribuciones "zipfianas" [BYGJ+07, BYGJ+08, BYJ12, CJP+10, GS09, LM03, LM04, SSdMZ+01, WLY+13, XO02]. Este hecho es crucial para el diseño de servicios de cache distribuidos eficientes, pero muy pocas veces ha sido considerado.

Un objetivo de diseño ingenuo podría ser alcanzar la tasa máxima de *hit* (TMH), la cual se calcula considerando un intervalo de tiempo, por ejemplo un mes, y obteniendo

el número de consultas totales (Q) y el número de consultas distintas (D). Entonces, la *TMH* considerando el intervalo de tiempo es $TMH = (Q - D)/Q$. De hecho, considerar un CS con *F2* entradas disponibles, siendo *F2* el número de consultas que se repite más de dos veces, es lo mismo que considerar un CS con cache infinita (ese CS alcanza la *TMH*).

Dimensionar un CS que alcance la *TMH* es una pérdida de recursos, ya que consultas con muy pocas apariciones (por ejemplo, 2, 3 o 4 veces), serán mantenidas en cache. A pesar de este hecho, cientos de miles de entradas deben ser empleadas para alcanzar una tasa de *hit* alta y aceptable. Esta arista no es la más importante, es necesario tomar en consideración los límites de los nodos que componen el servicio y el volumen de consultas de usuario por unidad de tiempo. Por ejemplo, un nodo no puede aceptar más que x peticiones por segundo; si el número de peticiones excede x, entonces el nodo podría colapsar o congestionarse. Pero tampoco se puede considerar un servicio con infinitos nodos. Existe un equilibrio entre la tasa de *hit* y la cantidad de recursos empleados [BYJPW07, CJP+10, SJPBY08]. En [BYGJ+07, BYGJ+08, BYJPW07, CJP+10, LM03, Mar01, SJPBY08] se determinan las tasas máximas de *hit* para *logs* de consultas específicos de distintos tamaños y períodos de tiempo.

Esta es la razón porque es tan importante balancear apropiadamente la carga en este servicio. Un CS bien balanceado permite usar la cantidad correcta de recursos, ya que es más simple dimensionarlos para cumplir con parámetros de desempeño. Además, un CS balanceado se vuelve menos susceptible a alzas bruscas de tráfico y a la falla de nodos. Por otro lado, la forma en que se mantiene balanceado el CS debe ser dinámica, ya que el comportamiento de usuario es impredecible y variable, incluso en función de husos horarios. Con balance dinámico de carga, se mitiga cualquier peligro relativo a fallas en los nodos de CS.

Todas las entradas del CS se mantienen en memoria y cada entrada es un par $<$*query*, *answer*$>$ que ocupa unos pocos kilobytes [AXF+12, GS09, SSdMZ+01, SJPBY08, WLY+13]. La clave de búsqueda en el CS es "query". Cada una de estas entradas es valiosa, ya que implican un ahorro en la utilización de recursos para su resolución. Ante la caída de un nodo, toda esta información valiosa se pierde, lo que abre la posibilidad de estudiar cómo proteger esta información. La primera pregunta es

qué proteger, pero dada la distribución zipfiana de las consultas de usuario, los candidatos a proteger son las consultas que tienen mayor frecuencia. Con esto, se da cobertura a un mayor número de consultas que no deben ser recalculadas al perder la información histórica en los nodos que fallan.

2.4. Servicio de Índice (IS)

Los datos fundamentales en los WSEs son los documentos Web. Los documentos se deben recoletar, *parsear* para obtener la información importante, organizar y guardar en almacenamiento secundario (no volátil). Una vez recolectados los documentos Web, a través del proceso denominado *Web crawling* [CGMP98, Cas05, BYCMR05], se deben examinar para obtener la información relevante e irrelevante (que es eliminada). Luego, esta información se debe organizar de forma tal que facilite y acelere el proceso de búsqueda. A este proceso se le denomina indexación [BYRN99, MRS08]. Un WSE comercial indexa del orden de millones o miles de millones de documentos [LS06, GS09].

La estructura clásica utilizada por los WSEs para el indexamiento es el *Índice Invertido* [BYRN99, MRS08]. Esta estructura de datos está compuesta por un *diccionario* y un conjunto de *listas invertidas* o *postings* (ver Figura 1). El diccionario es un conjunto de términos que aparecen en los documentos. Estos términos están normalizados y no contienen palabras muy comunes (denonimadas *stopwords* y que están previamente definidas) para permitir un mejor *ranking* de documentos. Por otro lado, cada término t del diccionario tiene asociado una lista invertida l_t. La lista invertida l_t indica el conjunto de documentos en los cuales aparece el término t. Este diseño tiene por finalidad permitir un acceso rápido a los documentos en los cuales está un término t. El conjunto de documentos para el término t se puede ver como una lista de pares (d_i, f_i), tal que f_i es la importancia que tiene el término t en el documento d_i. La finalidad de la importancia f_i es la elaboración del *ranking* de documentos, es decir, determinar qué documentos son más importantes que otros.

Dada la cantidad de información a almacenar y el flujo de consultas que debe resolver el IS, este servicio debe ser alojado en múltiples máquinas. El esquema clásico

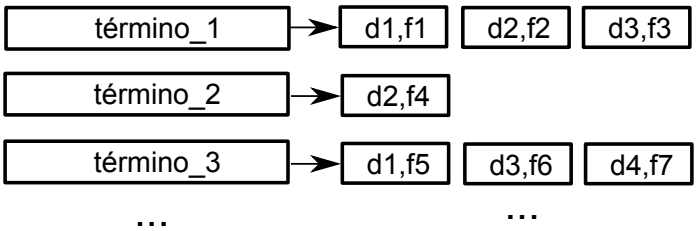


Figura 1: Índice invertido.

en que ha sido resuelto este problema es el *particionado por documentos* [PSL06, KC10]. Como su nombre lo indica, el particionado por documentos significa que, dada una colección de M documentos, éstos son repartidos entre P particiones siguiendo algún método. Los métodos clásicos son asignación aleatoria [PSL06], *clustering* de documentos [KC10], entre otras. Con la asignación aleatoria, cada máquina recibe M/P documentos aproximadamente. Independiente de la forma en que se realice la distribución de documentos, el siguiente paso es que cada máquina construya su propio índice invertido en forma local y autónoma considerando sus documentos. Al finalizar esta etapa, cada máquina tiene su propio índice invertido, con lo cual puede responder consultas. Este método de particionamiento tiene como fin el balance de carga [BYRN99].

Para responder una consulta considerando todos los documentos y dadas P particiones, cada una con su índice invertido, se deben consultar en forma independiente a todas las P particiones del IS. Cada partición del IS reportará sus mejores documentos para la consulta siguiendo una función de *ranking*. Luego, los P conjuntos de documentos deben ser unidos para generar la respuesta global a la consulta inicial. Como se aprecia, una consulta genera la misma carga de trabajo en todas las particiones. Si la distribución de documentos es uniforme, entonces la carga de resolver una consulta en cada una de las particiones es similar. Esto hace que no existan problemas graves de desbalance en este servicio.

Lo más obvio es asignar una partición a una máquina, pero al igual que en el CS, no es suficiente considerar sólo la dimensión de información a almacenar, sino que también las capacidades de las máquinas que componen el servicio. Es decir, dada una colección de M documentos y cada máquina con una capacidad de almacenaje de

C documentos, lo más probable es que sean necesarias mucho más que M/C máquinas para desplegar este servicio. Por otro lado, si se consideran infinitas máquinas, cada máquina tendrá muy pocos documentos, lo que se podría considerar una buena solución. Esta solución es inviable, ya que los costos asociados se incrementan en función del número de particiones. Por ejemplo, a medida que se aumenta el número de particiones, aumenta el número de latencias de disco y la transferencia en la red. Este *trade-off* es considerado en la investigación de Raiciu *et al.* [RHHR09].

Por otro lado, en caso de falla de alguna máquina, la información de los documentos alojados en esa máquina no estará disponible. Deben existir mecanismos para seguir funcionando en caso de fallos (tolerancia a fallas). El mecanismo clásico para proveer tolerancia a fallas es la replicación [Cap09, GS96, LGTT01, Tre05]. Es decir, cada partición es desplegada en D máquinas que contienen exactamente el mismo índice invertido. Con esto, el servicio de índice puede ser visto como una matriz de P particiones y D réplicas. Además de proveer tolerancia a fallas, la replicación sirve para reducir el tiempo de respuesta de una partición, ya que las x peticiones a ser resueltas en una partición, son distribuidas uniformemente en las D réplicas, con lo que cada réplica de la partición atenderá x/D peticiones. En conclusión, la replicación de particiones es un mecanismo que sirve para proveer tolerancia a fallas y para reducir el tiempo de procesamiento en el IS.

Para dimensionar correctamente el servicio de índice, es necesario considerar: (i) número de documentos a almacenar (relacionado con el número de particiones); (ii) cotas de tiempo de resolución de una consulta en una partición (cada partición debe responder una consulta en menos de t milisegundos considerando los documentos asignados a ella); y (iii) disponibilidad del servicio (proveer tolerancia a fallas).

En general, el costo computacional de una operación en el IS es varios órdenes de magnitud mayor que una operación en el CS. Por otro lado, la operación principal en el IS es la realización del *ranking* de documentos según una consulta y considerando los documentos asignados a una partición. Para la aceleración de este proceso, se utilizan comúnmente mecanismos de cache [LS05, OSABC$^+$12, SSdMZ$^+$01, ZLS08, ZS07], pero aún así la operación en el IS es costosa. Es por este último motivo que el estudio del Servicio de Cache, en términos de balance y tolerancia a fallas, es importante para

la operatoria global del motor de búsqueda.

2.5. Procesamiento de Consultas

A grandes rasgos y según lo descrito en las secciones anteriores, el flujo se puede describir con las acciones etiquetadas en la Figura 2. Cada petición hecha al WSE, arriba a un nodo fs_i del FS (100), el cual selecciona un nodo cs_j del CS y le envía la petición para determinar si cs_j aloja la respuesta a la petición (102) (dependiendo de la organización, el envío de la petición podría ser a más de un nodo). El nodo cs_j determina si mantiene en su memoria la respuesta a la petición y responde al nodo fs_i (FS) con un *miss* (si no la mantiene) o *hit* (si la tiene en su memoria) más la respuesta pre-computada (104). En caso de *hit*, la respuesta es presentada al usuario (106), y en caso de *miss*, la consulta es enviada a todas las particiones del IS para su resolución (108) y en cada partición se selecciona una réplica en paralelo (110). El IS computa la respuesta global a la consulta y la envía al nodo fs_i del FS (112), y éste último presenta la respuesta al usuario (106). En caso de *miss*, y además de enviarle la respuesta al usuario, el nodo fs_i envía la consulta y su respuesta al nodo cs_j (114) (aquel que generó el *miss*). El motivo de este envío es dar la oportunidad a la consulta de ser almacenada en cache. Por otro lado, una política de admisión [BYJPW07] puede ser implementada en el FS para impedir que consultas con ciertas características, por ejemplo poco frecuentes, sean almacenadas en cache. En este último caso, si la consulta no es admitida en cache, se envía directamente al IS (108) y los pasos marcados con [*] no toman lugar. Finalmente, una estrategia de invalidación de consultas puede ser implementada como un proceso *background* en el Servicio de Cache, así cada entrada detectada como inválida puede ser reenviada al IS para ser resuelta y volver a almacenarla en el CS [CJP+10]. Un ejemplo de este tipo de estrategias es *Time-To-Live* (TTL) [AAO+11].

Como se aprecia, estos servicios trabajan en forma independiente, por lo que se pueden analizar en forma individual y buscar las debilidades/fortalezas de cada servicio. Esta es la arquitectura propuesta y estudiada en este trabajo de tesis.

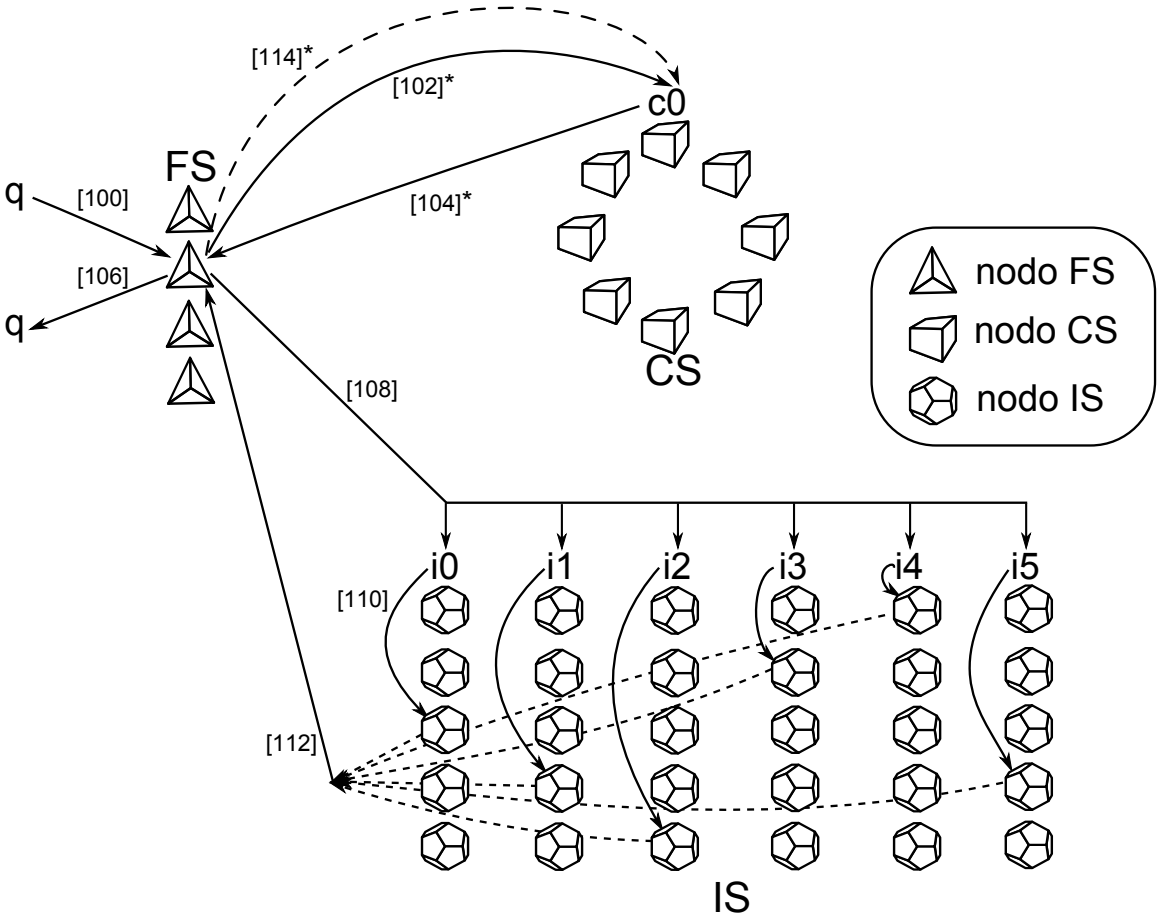


Figura 2: Principales componentes y su interacción en un WSE.

2.6. Análisis

Como fue descrito anteriormente, el FS no presenta mayores problemas: es un servicio replicado que sirve de interfaz entre el motor de búsqueda y el usuario. Debe rutear las consultas entre los distintos servicios para producir la respuesta final al usuario. Al ser el servicio que rutea y asigna las tareas en los distintos servicios, el FS es un elemento importante a considerar en cualquier mecanismo que tenga como fin el balance de carga del motor de búsqueda. Todas las consultas de usuario pasan por el FS, entonces en este servicio se podrían aplicar técnicas de monitoreo de consultas que, como se verá más adelante, resulta ser una herramienta efectiva en el control y balance de la carga de trabajo del Servicio de Cache.

Además, al actuar de ruteador entre los servicios, el FS mantiene una visión del estado de los nodos de los demás servicios. El estado de un nodo se refiere a si está activo/inactivo y también a la carga de trabajo de cada nodo. Al considerar mecanismos de balance dinámico, como los propuestos en este trabajo, se debe tener en consideración la situación actual de cada nodo. Uno de los mecanismos más utilizados para este fin, es el mecanismo de *heartbeats* [GLTT01, HHZ$^+$ch, LVD$^+$11, LWX06, Tre05, VDA$^+$98]. Este mecanismo hace que cada nodo del sistema reporte a un coordinador (en este caso el FS) su estado y carga de trabajo cada cierto intervalo de tiempo, generalmente del orden de pocos segundos. Si el coordinador no recibe la señal de *heartbeat* de un nodo durante n períodos consecutivos, se considera que dejó el servicio y no se hacen peticiones a este nodo. Por ejemplo, en [VDA$^+$98] los *heartbeats* son enviados cada 1,2 segundos, y si en 7,2 segundos (6 *heartbeats* consecutivos) no se tiene noticia de un nodo, entonces se marca como caído. Por otro lado, a modo de aproximación, la carga de trabajo de un nodo puede interpretarse en el FS como la cantidad de peticiones hechas a ese nodo, ya que como se menciona en [SJPBY08], la resolución de una consulta en el CS toma tiempo constante.

La arquitectura del IS hace que la carga de trabajo experimentada por el servicio sea balanceada. Una consulta genera carga de trabajo en todas las particiones (particionado por documentos). Además, como cada partición tiene asignada aproximadamente la misma cantidad de documentos, el procesamiento de una consulta

toma casi el mismo tiempo en cada partición. En los únicos casos en que se genera un leve desbalance, es en la situación en que alguna de las réplicas de una partición deja el servicio de índice (esa partición queda con un nodo menos). El servicio de índice ha sido altamente estudiado (estructuras de datos, compresión, cache, organización, etc.) y existe un consenso generalizado en la arquitectura de particionamiento utilizada más comúnmente y usada como estándar.

Finalmente, el Servicio de Cache (CS) es el elemento clave en el procesamiento de consultas en los motores de búsqueda Web. Este servicio funciona como un "amortiguador" de carga de trabajo del servicio de índice. Toda consulta encontrada en cache no es enviada al servicio de índice, lo cual reduce drásticamente el tiempo de respuesta de una consulta. Siguiendo esta idea, una solución válida sería tratar de mantener todas las consultas en cache y eliminar el servicio de índice. Esta solución no es factible debido a que siempre los usuarios generan nuevas consultas y a que las respuestas a cada consulta cambian cada cierto período de tiempo.

Como se verá en los siguientes capítulos, el Servicio de Cache presenta serios problemas de desbalance que se denominarán *desbalance estructural* (distribución sesgada de las consultas de usuario) y *desbalance dinámico* (variación a través del tiempo en la composición de las consultas de usuario). Además, la pérdida de nodos en el servicio agrava los problemas de balance. Este componente esencial en el funcionamiento de los motores de búsqueda será el objeto de estudio del presente trabajo.

2.7. Consultas

Una de las entradas esenciales para este trabajo son los registros de consultas. Estos registros provienen del tráfico de peticiones de usuarios almacenado por un motor de búsqueda comercial. Los datos almacenados contienen elementos de sesiones, consultas realizadas, clicks, etc., y se dividen en *buckets*, los cuales representan zonas geográficas bien definidas. Dentro de los *buckets* disponibles, se escogió el de Estados Unidos debido principalmente a su masividad de datos (volumen de consultas útiles). Dada la importancia de las consultas, a continuación se presentarán algunas características de los datos utilizados. La idea es que se tenga en mente la dimensión de

los datos, su distribución y cómo varían a través del tiempo.

En la Tabla 1 se indican algunos datos acerca de los registros utilizados. La escala temporal de cada conjunto es mensual y se consideraron tres meses consecutivos para estudiar cómo cambian las consultas en esos meses.

1. **Número de consultas distintas**. Considerando cada mes en forma aislada, se observa que existe poca variación en el número de consultas distintas (sólo se está hablando del volumen, no del conjunto ni composición de las consultas). Este parámetro debe ser evaluado para dimensionar el Servicio de Cache en términos de número de entradas. Además, se observa el mismo patrón en las consultas que se repiten más de dos veces y las que ocurren sólo una vez. Con esto se puede concluir que tanto el volumen como la distribución de consultas mes a mes, es relativamente el mismo en los meses analizados.

2. **Composición de consultas**. En promedio, las consultas tienen 2,97 términos y 18,13 caracteres alfanuméricos (contando espacios). Antes de utilizar las consultas, se realizó una etapa de pre-procesamiento según las prácticas más comunes [GS09]. *Grosso modo*, las consultas resultantes sólo se componen de caracteres, números y espacios. Estadísticas similares a las expuestas son encontradas en [BYGJ+07, BYGJ+08, BYJPW07, SMHM99, XO02].

3. **Impacto de la consulta más frecuente**. La consulta más frecuente ("facebook") resulta ser la misma en todos los *logs* utilizados, y la mayoría de las veces tiene el mismo impacto en los *logs* de cada mes (2,65 % en promedio). Esto implica que si esta consulta se almacena en cache, un 2,65 % de las peticiones no usará el Servicio de Índice (IS).

4. **Alto dinamismo**. Se observa que en cada mes se generan aproximadamente 80,8 millones de consultas nuevas distintas considerando el mes anterior (16,71 % de nuevas consultas cada mes en promedio). Este porcentaje fue calculado sólo considerando las consultas con frecuencia mayor o igual a 2. Esto implica una alta capacidad de los usuarios para cambiar el foco de las peticiones en función de eventos sociales, económicos y naturales. Esto último indica que el IS es

Tabla 1: Características de los *logs* de consulta utilizados ($f(q)$ es la frecuencia de q).

Características del *Log*	**Diciembre, 2011**	**Enero, 2012**	**Febrero, 2012**
N° consultas	1.917.556.179	2.032.123.082	1.893.555.146
N° consultas distintas	466.000.248	506.756.863	478.271.434
N° consultas con $f(q) = 1$	355.207.061	386.822.371	366.965.677
N° consultas con $f(q) \geq 2$	110.793.187	119.934.492	111.305.757
N° de términos por consulta	2,95	2,99	2,98
Largo promedio de consulta	17,99	18,27	18,12
% consulta más frecuente	2,65 %	2,60 %	2,71 %
Tasa máxima de *hit*	75,70 %	75,06 %	74,74 %
N° consultas nuevas (distintas)	77.991.118	87.007.209	77.477.308

necesario para la operación del WSE, ya que un porcentaje no despreciable de consultas deben ser resueltas de la información disponible en el IS (no estarán en cache).

5. **Tasa máxima de *hit***. Dado que el número de consultas que se repiten más de dos veces es similar, la tasa máxima de *hit* que puede alcanzarse es en promedio 75,17 %. Para alcanzar la tasa máxima de *hit* se debería considerar una cache con infinitas entradas disponibles. Estas estadísticas dependen del registro de consultas utilizado.

Un punto importante a señalar es que los *logs* de consultas utilizados en este trabajo representan un muestreo de todo el tráfico de un WSE y que es representativo del tráfico total (el muestreo es del orden del 1 %). Para corroborar este punto, en [BYGJ+08, LM03, LM04, SSdMZ+01] se indica que en los motores de búsqueda se someten millones de peticiones diariamente. Pero trabajos más actuales, del año 2012 y 2013, indican que importantes aplicaciones Web, entre los que se cuentan los motores de búsqueda, reciben miles de millones de peticiones por día [AXF+12, NFG+13]. Por este motivo, los números expuestos en la Tabla 1 podrían presentar cambios si se considera el tráfico total de un WSE.

Como se mencionó anteriormente, las consultas de usuario siguen una distribución Zipfiana, y este comportamiento se encuentra ampliamente evidenciado y aceptado en la comunidad científica. Este punto es importante, ya que permite tomar decisiones en el diseño de los componentes de los WSEs. Por ejemplo, al seguir distribuciones

Zipfianas, una pequeña cantidad de consultas tiene un alto impacto, por lo que si se implementa un mecanismo tolerante a fallas para la protección de las respuestas precomputadas, entonces proteger (replicar) una pequeña fracción de consultas genera que una alta cantidad de peticiones no deba recurrir al servicio de índice al momento de fallos.

En conclusión, estudiar el *log* de consulta da una idea inicial acerca de los parámetros necesarios para dimensionar el Servicio de Cache. Como se verá más adelante, es necesario considerar aspectos adicionales como capacidad de los nodos, tiempo máximo de respuesta, entre otros.

Análisis del Estado del Arte

3.1. Contextualización

Según los trabajos de [BYGJ+08, BYJ12, BBJ+10, CJP+10, CA12, SJPBY08, XO02], el Servicio de Cache es un elemento importante para:

- Reducir la carga de trabajo de los servidores *Back-End*.
- Mejorar la utilización de la red.
- Reducir la cantidad de recursos utilizados en el procesamiento de peticiones/consultas.
- Acelerar el tiempo de respuesta de las peticiones/consultas.
- Incrementar el *throughput* (peticiones/consultas resueltas por unidad de tiempo).

Esto implica que cualquier mejora en este servicio tiene un alto impacto en el funcionamiento global de los motores de búsqueda y de las grandes aplicaciones Web. En particular, en el capítulo 4 se dará evidencia empírica de los distintos problemas a los que está expuesto un Servicio de Cache. De ahí la importancia de hacer un análisis de las causas y proponer soluciones eficientes a ellas.

Para dimensionar el amplio espectro que abarcan las técnicas de cache, a continuación se detallan algunas áreas importantes en el contexto de La Web:

- **Políticas de desalojo**. La cache es implementada en memoria principal de un nodo, por lo que es un recurso limitado. Cada vez que la cache (memoria del nodo) se llena y se debe insertar una nueva entrada en ella, se selecciona alguna entrada en cache, mediante una política, para que sea desalojada de la memoria. A esta política se le denomina algoritmo de reemplazo o política de

desalojo. Entre las políticas de desalojo más comunes está *Least Recently Used* (LRU) y *Least Frequently Used* (LFU) [Den70]. Existen políticas que consideran características adicionales de las consultas, tales como el costo de su resolución en los servidores *Back-End* y/o servicios de índice [OAU11], componentes probabilísticas [LM03], estacionalidad y variabilidad de consultas [FPSO06], entre otros trabajos [BYGJ+08, GS09, Mar01]. La idea es optimizar la memoria disponible para cache y almacenar las consultas más frecuentes y/o más costosas de resolver. Estas políticas se implementan a nivel de nodo.

- **Políticas de eliminación de resultados "antiguos"**. Las consultas almacenadas en cache podrían tener un tiempo de vencimiento, es decir, un intervalo de tiempo o un tiempo límite de permanencia en cache. La idea es que los resultados vuelvan a ser computados para que sean "frescos" (que consideren los últimos resultados insertados en el servicio de índice). Algunos trabajos son [AAO+11, BBJ+10, CJP+10]. Estas políticas también son implantadas a nivel de un nodo en particular.

- **Políticas de admisión**. Ciertas entradas podrían perjudicar el rendimiento del cache si se permite que sean almacenadas. Por ejemplo, las consultas poco frecuentes impactan el desempeño, ya que mantener alojadas en memoria estas consultas no aumenta el número de *hits* en cache. La idea principal es bloquear la admisión en cache de entradas que generen poca ganancia al mantenerlas en cache. Algunos trabajos son [BYJPW07, LhLH+10, LS05].

- **Cache de localización**. Según el método de resolución de consultas en el servicio de índice, una consulta debe visitar un conjunto de nodos para ser resuelta. A veces este conjunto de nodos es extenso. Existen estructuras cache que son utilizadas como un medio para reducir el número de nodos visitados en servicios de índice. La idea principal es que ciertas consultas, en general las más frecuentes, visiten menos nodos en su procesamiento, con el objetivo de reducir la carga de trabajo. En [FMM09, MFM+09] se encuentran trabajos relacionados a esta área.

- **Cache distribuido**. El cache distribuido trata de la organización en forma cooperativa de un conjunto de nodos. Un esquema clásico utilizado es la organización de los nodos en una jerarquía, en que cada nivel de la jerarquía asume una función específica. Las funciones más importantes definidas en estas jerarquías son cache de resultados, cache de intersecciones, cache de listas invertidas, entre otras. Entre los trabajos en que se encuentran mecanismos de cache en múltiples niveles están [LS05, MGCGP10b, OSABC+12, SSdMZ+01, ZLS08]. Estos trabajos están orientados a jerarquías de cache en *clusters* (o conjunto de nodos altamente acoplados).

- **Cache en sistemas P2P**. Los sistemas *Peer-to-Peer* son sistemas a gran escala en que, a diferencia del punto anterior, el retardo de comunicación entre nodos/*peers* es alto. Además, todos los nodos no están conectados entre sí, sólo existe información parcial para rutear apropiadamente las peticiones. Ante esto, el cache resulta fundamental en el camino que tiene una petición hacia el nodo que almacena la información pedida. Algunos trabajos son [RHM12, SH06].

- **Web Caching**. Este es un mecanismo de almacenamiento temporal de documentos para reducir el ancho de banda y la carga de trabajo generada por las peticiones. Estos documentos son almacenados en los nodos intermedios desde el cliente (genera la petición) al servidor (responde la petición). Un trabajo interesante en esta área es [BCF+99], ya que hace un análisis del impacto de las distribuciones Zipfianas en esta técnica. En [PB03, Wan99] se encuentran *surveys* de los distintos mecanismos para Web Caching.

En el contexto de este trabajo, se está interesado en el cache distribuido, ya que el Servicio de Cache, así como los servicios restantes, están implementados sobre un conjunto de nodos fuertemente acoplados (cluster con una red de comunicación de alto ancho de banda). Cada uno de estos nodos funciona como una caja negra, en que se almacenan consultas y sus respuestas, y se permite la consulta de elementos. Es decir, áreas como las políticas de reemplazo, eliminación de resultados antiguos y mecanismos de cache de localización quedan fuera del alcance de este trabajo. El Servicio de Cache se verá desde una perspectiva de alto nivel, en que cada nodo tiene

un estado y es posible obtener métricas, tales como tasa de *hit* y utilización (que tan sobrecargado está).

Hasta donde conocemos, no existen trabajos que aborden la problemática de diseñar servicios de cache distribuidos eficientes. La mayoría de los trabajos están orientados a un nodo en particular, o sistemas distribuidos en que el retardo de comunicación es una variable importante (sistemas P2P).

Existen una serie de objetivos a alcanzar en el diseño de un Servicio de Cache. Cada objetivo tiene una métrica asociada. El principal objetivo de un Servicio de Cache es tener una alta tasa de *hit*. Esto implica que un alto porcentaje de las consultas realizadas al WSE estarán pre-computadas y podrán ser resueltas sin acceder al servicio de índice. El cumplimiento de este objetivo contribuye a fortalecer cada uno de los puntos importantes descritos al inicio de este capítulo.

Esta maximización de la tasa de *hit* no puede ser a cualquier costo, principalmente utilizando un número limitado de nodos, y por ende de memoria total disponible. Además, se debe considerar que existe una Tasa Máxima de *Hit* descrita en el capítulo 2, por lo que es importante un correcto dimensionamiento del servicio. Este dimensionamiento debe considerar cierta holgura para hacer frente a eventos, tales como fallos de los nodos.

Siguiendo la idea anterior, los fallos representan una constante dentro de los sistemas computacionales, por lo que se debe explorar el impacto de los fallos en el Servicio de Cache. Específicamente, un fallo en un nodo implica que toda la información de ese nodo se pierde. Esa información perdida representa recursos computacionales usados anteriormente para la generación de una respuesta a una consulta, por lo que se deben establecer mecanismos para mitigar este impacto. Cuando existan fallos, la idea principal es que la tasa de *hit* no se vea afectada en forma drástica y que la pérdida de información relevante tenga un efecto limitado. La métrica para medir el impacto de los fallos es principalmente la tasa de *hit*.

Por último, un sistema balanceado en términos de carga de trabajo de los nodos, permite la utilización correcta de los recursos computacionales. Además, un Servicio de Cache balanceado es menos proclive a fallos y/o congestión de nodos (son menos frecuentes), y queda menos expuesto a alzas abruptas de tráfico.

La solución óptima sería la combinación de: (i) máxima tasa de *hit*; (ii) balance de carga óptimo; y (iii) nulo transtorno en caso de fallas. Como se verá en el transcurso de este trabajo, existe un compromiso entre los puntos descritos anteriormente.

En la siguiente sección, se enunciarán las soluciones posibles que permiten cumplir con los objetivos planteados.

3.2. Soluciones

Considerar la siguiente situación: distribuir un conjunto de consultas en un grupo de nodos, tal que la tasa de *hit* sea máxima. Sea P el número de nodos del Servicio de Cache y N el número de entradas disponibles en cada nodo para cache.

La primera opción es que cada consulta que arribe al servicio, sea asignada en forma aleatoria a cualquiera de los P nodos. Como las consultas de usuario siguen una distribución Zipfiana (capítulo 2), a través del tiempo, en cada uno de los nodos quedarán alojadas en cache las consultas más frecuentes. Como la distribución es aleatoria, esto implica que cada nodo del servicio tendrá aproximadamente el mismo conjunto de consultas alojadas en cache, es decir, el número de entradas distintas del servicio será aproximadamente N. Esto no es escalable, ya que a medida que se aumenta el número de nodos, el número de entradas disponibles para cache, considerando todo el servicio, se torna constante en N. Es más, esta estrategia se comporta de la misma forma que tener sólo un nodo para cache (en términos de tasa de *hit*). Al existir menos entradas disponibles totales en el servicio, existe una tasa de *hit* más baja. Ante un fallo no existe pérdida de información relevante, ya que se actúa como un servicio totalmente replicado.

Otra opción está compuesta por los siguientes pasos al arribo de una consulta:

1. Calcular $pid = hash(q)\,\%P$ (el valor de pid varía desde 0 hasta $P - 1$).
2. Asignar la consulta q al nodo con identificador pid.

Lo descrito anteriormente tiene la ventaja de que siempre una consulta q será asignada al mismo nodo. Además, cada nodo es responsable por un conjunto disjunto de consultas. El número de entradas disponibles para cache de este servicio está dado

por $N \cdot P$, es decir, se maximiza la tasa de *hit* al tener la cantidad máxima de entradas disponibles para cache. Además, es simple y rápido de implementar. La gran desventaja de esta solución es la consecuencia de un fallo en un nodo, ya que al cambiar el número P implica un cambio de responsable para casi la totalidad de consultas, lo que implicaría una drástica caída en la tasa de *hit* durante un período de tiempo. Lo mismo sucede ante el ingreso de nodos al servicio (crece P).

Las estrategias descritas anteriormente funcionan bien en el caso de conjuntos fijos de nodos. En cambio, en un contexto real esto no sucede y el cambio en los conjuntos de nodos es debido a (i) caída de nodos y (ii) agregación de nodos. El punto (i) es referido a los fallos en los nodos, ya sea por sobrecarga o falla de hardware, y el punto (ii) a que se necesitan más nodos para hacer frente a una carga de trabajo determinada. El problema del cambio o inconsistencia de "vistas" [KLL$^+$97] se refiere al cambio que experimenta un cliente del mapeo de las claves a los nodos. Ciertamente, bajo la agregación o reducción de nodos, habrán cambios de vistas, pero para mantener una alta tasa de *hit* se necesita minimizar este impacto (reducir la cantidad de cambios en el mapeo de claves a nodos).

Los trabajos de Karger *et al.* [KLL$^+$97, KSB$^+$99] presentan Consistent Hashing (CH), una solución elegante e intuitiva al problema del cambio de vistas. Para implementar CH se debe escoger una función hash estándar que mapee cadenas de texto (consultas) a un rango $[0, ..., M)$. Luego, si se divide cada uno de los valores por M, se tiene que las cadenas son mapeadas al rango $[0, ..., 1)$. Este rango se mapea a una circunferencia, conocida como círculo o anillo unitario. Es decir, cada cadena/consulta es mapeada a un y sólo un punto en el anillo. Al mismo tiempo se realiza un mapeo de cada uno de los nodos disponibles al anillo. Finalmente, una cadena/consulta es asignada, desde su punto en el anillo, al primer nodo que se encuentre siguiendo el sentido de las agujas del reloj.

La Figura 3(a) muestra la asociación de distintos ítemes/consultas (qi) y nodos (Pj). Por ejemplo, las consultas *q5* y *q6* son mapeadas al nodo *P0*. El nodo *P1* es responsable por todas las consultas que intersecten el intervalo A (se sigue el sentido de las agujas del reloj). Por otro lado, en la 3(b) se expone la situación de un fallo. En este caso, el nodo *P1* falla, por lo que todas las consultas asociadas a ese nodo (rango

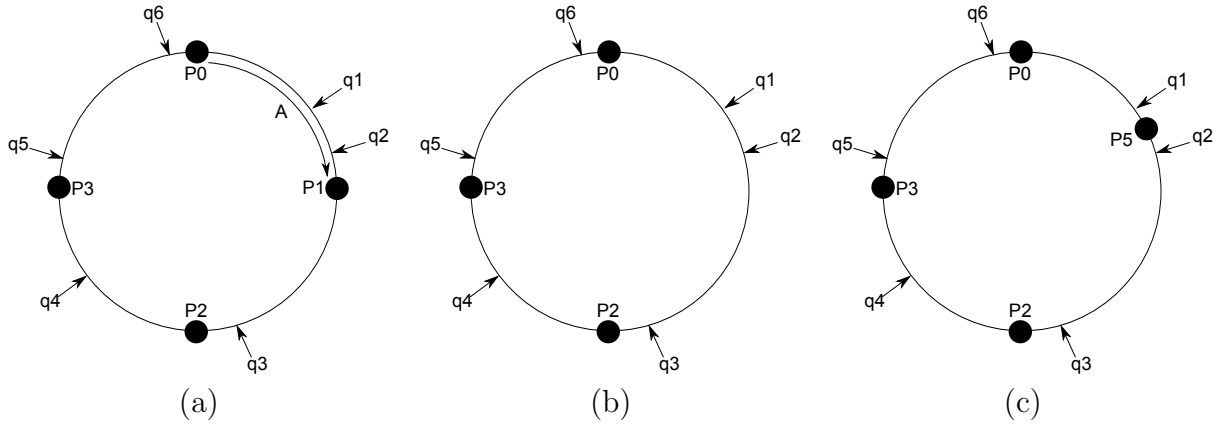


Figura 3: Vista lógica de Consistent Hashing: (a) mapeo de consultas a nodos; (b) falla de un nodo; y (c) agregación de un nodo.

A) cambian de responsable y son asociadas, desde el momento de la falla, al nodo *P2* (ahora el nodo *P2* atiende las consultas que intersectan A y su propio rango). Cabe señalar que las consultas que cambian de responsable sólo son las asociadas al rango A, y por ende se produce una leve alteración en la asociación entre consultas y nodos (se reduce el problema de cambios de vistas). Finalmente, en la Figura 3(c) se observan los cambios producidos ante la inserción de un nodo en el servicio. Se observa que el rango perteneciente a *P2* es dividido en dos partes, cuyos responsables son *P5* y *P2*. Con esa operación se reduce la carga de *P2*, y nuevamente la alteración en el mapeo de consultas a nodos es mínima.

En la implementación de CH, todos los valores que representan nodos del servicio son almacenados en un arreglo ordenado. Así, una vez calculado el hash de una consulta, se recorre vía búsqueda binaria para obtener el nodo responsable por la consulta. Esto implica que para implementar un servicio de n nodos, la búsqueda del nodo responsable para una consulta tarda $O(\log n)$. En [KSB+99] se indica que en la práctica, funciones hash que mezclan bien, como MD5, son suficientes. Un punto a considerar es que dos o más consultas que produzcan el mismo hash (colisiones) no tiene impacto, ya que sólo se necesita un mecanismo que haga el mapeo de una consulta a un nodo en particular (muchas consultas serán mapeadas a un nodo).

Una característica importante que será explotada en este trabajo es la asignación

fija de los puntos en el anillo. Como fue mencionado en el capítulo 2, las consultas de usuario varían a través del tiempo, en intervalos de minutos, horas y días. La política de mantener los puntos fijos no permite modificar la asignación de tareas a cada nodo dependiendo de la carga de trabajo experimentada por cada uno de ellos. Por otro lado, la distribución de puntos en el anillo ha sido clásicamente de dos maneras [DHJ+07a, DHJ+07b]: (i) asignación aleatoria; y (ii) asignación equidistante. Ninguna de estas asignaciones considera la carga de trabajo experimentada por cada nodo, y como se verá en los siguientes capítulos, no contribuyen a obtener un buen desempeño en el Servicio de Cache.

En nuestro contexto, el desempeño de Consistent Hashing está dado por la carga de trabajo generada por la distribución de consultas. Anteriormente se mencionó que se utiliza una función hash que mapee las consultas en el anillo. Según la evidencia empírica, utilizar una función simple como SHA-1 o MD5 genera que el universo de consultas sea distribuido uniformemente en el anillo. Lo anterior es la primera dimensión, el universo de consultas, pero la segunda dimensión tiene relación con el impacto de cada uno de esos ítemes en el anillo. Este impacto se da principalmente por la frecuencia de una consulta. Las consultas de usuario siguen distribuciones Zipfianas, lo que implica que una fracción muy pequeña de consultas tiene alta frecuencia, mientras la gran mayoría tiene una baja frecuencia. Este "desbalance estructural" de las consultas puede ser mejorado mediante la disposición inteligente de los puntos que representan nodos en el anillo. Es por esta razón que la distribución de puntos en el anillo mediante asignación aleatoria o equidistante falla. Por otro lado, las consultas varían a través del tiempo, por lo que la política de asignación fija de puntos también falla, así que se debe diseñar un método que permita la asignación de carga en forma dinámica considerando el estado de cada uno de los nodos del servicio.

Una técnica utilizada sobre CH es la inclusión de nodos virtuales [KSB+99, SMLN+03], la cual consiste en la replicación de nodos (puntos) en el anillo. Esto se observa en Figura 4, en que cada nodo tiene asociado una tonalidad (rango) distinta. Por ejemplo, el nodo asociado al color negro es responsable de tres rangos disjuntos en el anillo. La idea principal es que cada nodo tenga la responsabilidad de atención de consultas de forma distribuida en el anillo (no sólo una sección contigua), así la distribución de

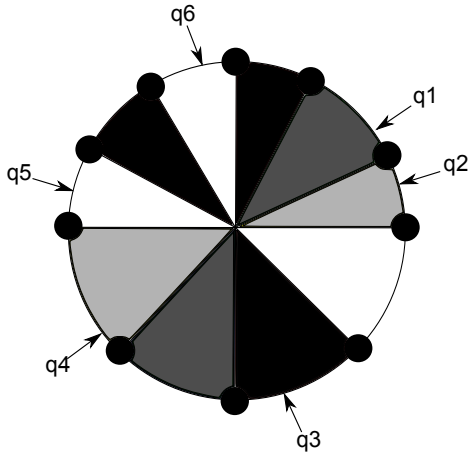


Figura 4: Nodos virtuales en Consistent Hashing.

consultas en los nodos es más homogénea. Esto además permite considerar aspectos de heterogeneidad entre los nodos, por ejemplo un nodo más potente en términos de procesamiento, puede ser asociado a más rangos que un nodo con menor poder de cómputo [DHJ$^+$07a, DHJ$^+$07b]. Esta técnica, según los autores, tiene un impacto positivo, pero anteriormente se mencionaron las debilidades de CH, y la inclusión de nodos virtuales no implica una mejora, ya que nuevamente la variación de las consultas de usuario y cantidad de carga de trabajo generada en cada uno de los nodos virtuales no es considerada. El costo para obtener el nodo de una consulta crece dependiendo del número de nodos virtuales, por ejemplo si por cada uno de los P nodos se insertan V nodos virtuales, entonces el costo de búsqueda es $O(\log PV)$.

Una alternativa práctica a los nodos virtuales, es el aplicación del paradigma "el poder de dos elecciones" (*power of two choices*) [ABKU99, MRS00] (o en forma genérica, el poder de d elecciones). Para entender este paradigma, suponer que se ponen secuencialmente n bolas en n cajas, colocando cada bola en alguna caja seleccionada aleatoriamente. Según [ABKU99], la caja más llena tiene $\ln n / \ln \ln n$ bolas, con alta probabilidad $(1 + o(1))$. Este es un problema de balance de carga clásico. Ahora suponer que por cada bola se escogen d cajas en forma aleatoria y se coloca la bola en la caja menos cargada, en el momento de la asignación. En [ABKU99] se muestra que, con alta probabilidad, la caja más llena tiene sólo $\ln \ln n / \ln d + O(1)$ bolas.

En el contexto de CH en sistemas Peer-to-Peer (P2P), es posible utilizar la idea de poder de d elecciones [BCM03]. La idea es que por cada ítem se utilizan $d \geq 2$ funciones hash distintas para generar d puntos en el anillo. Cada punto tiene asociado un nodo (en la mayoría de los casos distintos), y el ítem se asigna al nodo menos cargado de los d nodos seleccionados.

En [KLL+97] se propone el uso de un "árbol de caches" (*Tree of Caches*) para asegurar que muchas peticiones no vayan al mismo nodo. Para un servicio de C nodos y si se utiliza un árbol d-ario, entonces el árbol de caches tiene altura $\log_d C$. El nodo que es el responsable de la petición, es la raíz del árbol. El número de nodos en el árbol es igual al número de nodos del Servicio de Cache (todos los niveles están llenos, posiblemente con la excepción del último). Este árbol se construye en función del ítem/consulta, el nodo responsable y los demás nodos del servicio, así existe una alta cantidad de árboles de cache posibles. Cuando una petición arriba al Servicio de Front-End (FS), se obtiene el nodo responsable y su árbol de cache, luego se selecciona un camino en forma aleatoria desde un nodo hoja aleatorio hasta el nodo raíz (responsable). A este árbol se le denomina *Random Tree* y al camino aleatorio *Random Path*. Luego, en el peor caso, se tendrían que visitar $\log_d C$ nodos de cache (si en todos se tiene un *miss*), y en el mejor caso se debe visitar un nodo (si en el primer nodo del camino se tiene un *hit*). Dado el esquema anterior, se tiene la siguiente pregunta: ¿cuándo un nodo del camino almacena un par petición/respuesta? Para esto, se tiene otro parámetro, r, que representa el número de peticiones de una misma consulta que un nodo debe "ver" para almacenar una copia en su memoria local. El retardo que una petición experimenta está relacionado con la altura del árbol.

Como se mencionó en el capítulo 2, el Servicio de Front-End (FS) es el encargado de rutear las consultas en el CS. Las operaciones básicas que realiza el FS al CS son: (i) *buscar(q)*; (ii) *obtener(q)*; e (iii) *insertar(q,r)*. La operación *buscar(q)* permite determinar si una consulta q está almacenada en el CS o no. Si una consulta q está almacenada en cache, se obtiene su respuesta asociada con *obtener(q)*. Si una consulta q no está almacenada en cache, entonces se puede insertar, junto con su respuesta r, con la operación *insertar(q,r)*. Internamente, además de las operaciones descritas anteriormente, cada nodo del CS tiene las siguientes operaciones adicionales:

(i) *actualizar(q)*; y (ii) *desalojar(q)*. Con *actualizar(q)* se actualiza la prioridad de una consulta q. La actualización de una prioridad tiene relación con la política de desalojo implementada internamente en el nodo. Con *desalojar(q)* se elimina la consulta q y su respuesta desde la memoria del nodo. Esta operación tiene relación con hacer espacio para una nueva entrada en cache.

Anteriormente se definieron algunas soluciones para un cache distribuido. En función de lo anterior, varias organizaciones lógicas de los nodos del CS son factibles. La principal inquietud es alcanzar una alta tasa de *hit*, teniendo un servicio balanceado y tolerante a fallas. Sea *NCS* el número de nodos del CS, sea FS_i el nodo del FS que hizo la consulta al CS y sea *NE* el número de entradas disponibles para cache en cada nodo. Entonces las organizaciones lógicas posibles son:

1. *Difusión* (*Broadcast*). La idea es que bajo una operación de *búsqueda* de FS_i, la consulta es enviada a todos los nodos del CS, cada nodo examina su propia estructura y responde al nodo del FS que preguntó. En caso de *hit*, el nodo que contenga la respuesta envía adicionalmente la respuesta. En caso de *miss*, FS_i debe enviar la respuesta al Servicio de Índice (IS) y luego de ser resuelta en este servicio, la respuesta debe enviarse al CS para almacenarla junto con su consulta. Hay varias opciones para realizar esta inserción: enviarla al nodo que respondió con un *miss* o a todos los nodos. Si bien es cierto que el *broadcast* es simple y no genera desbalance, también es cierto que no es escalable ni eficiente. Más aún, si la operación *inserción* es enviada a un nodo aleatorio, este esquema reduce drásticamente la tasa de *hit*, ya que la probabilidad de que en el futuro la misma consulta sea enviada a la misma máquina seleccionada aleatoriamente, es muy baja. Sin embargo, dado el sesgo de las consultas de usuario, a medida que transcurre el tiempo el conjunto de consultas en los distintos nodos se torna similar. Si la operación *inserción* es enviada a todas las máquinas, las entradas no son usadas apropiadamente, ya que una consulta está replicada en todo el servicio (al igual que el mecanismo anterior). Por lo anterior, este mecanismo se torna tolerante a fallas. La latencia asociada a cada consulta por una operación de búsqueda son *NCS* búsquedas. El número de entradas disponibles totales para cache es aproximadamente *NE* (totalmente replicado).

2. *Particionado*. Otra opción es utilizar un método determinístico para seleccionar un nodo específico dada una consulta. Se dice que es "determinístico" ya que se necesita obtener el mismo nodo dada la misma consulta. Como se mencionó anteriormente, las opciones posibles son $hash(q)\,\%NCS$ y Consistent Hashing. Estos métodos particionan las consultas en conjuntos disjuntos y cada conjunto es asignado a un nodo diferente. Esto permite alcanzar la tasa de *hit* más alta. El gran problema, y mayor desventaja, es la asignación de carga desbalanceada dada la distribución sesgada de consultas de usuario. Además, la asignación se torna más injusta con el incremento del número de nodos (en el capítulo 4 se dará evidencia). Esta situación es acentuada por la aparición de consultas en ráfaga, las cuales tienden a sobrecargar o, en algunos casos, colapsar algunos nodos (todos aquellos relacionados con el tópico emergente). La operación de *búsqueda* es enviada a un solo nodo, lo cual mejora el desempeño y la utilización de los recursos. Cuando un nodo cae, todas sus entradas se pierden (sigue operando con degradación). El número de entradas disponibles para cache con este método es $NE \cdot NCS$ (máximo alcanzable).

3. *Jerárquico*. Otra posibilidad es establecer un orden de visita de los nodos (la visita implica preguntar por la consulta). Este ordenamiento implica visitar más de un nodo y explota este hecho para alcanzar una alta probabilidad de encontrar la consulta en un camino de nodos. Una asignación jerárquica posible es, usar la función hash clásica $h = hash(q)\,\%NCS$, la cual da un identificador de nodo, y entonces considerar como un camino $h, h+1, h+2, ..., h+M$ (todos en módulo NCS). Luego, el primer nodo del CS a visitar es h, si hay un *hit* se envía la respuesta a FS_i que hizo la pregunta. Si hay un *miss*, se visita el nodo $h+1$ y si hay un *hit* se envía la respuesta al nodo FS_i, sino se visita al nodo $h+2$ del CS. Este proceso se repite hasta visitar el nodo $h+M$ (se dan M chances de encontrar la consulta) y si sólo existen *miss* en el camino completo, se visita el IS para resolver la consulta. En el caso de un *miss* en todo el camino y teniendo la respuesta del IS, entonces existen algunas preguntas como ¿cuántos y cuáles nodos del camino se deben actualizar con la respuesta? Existen diversos

esquemas jerárquicos, tales como árboles, utilización de varias funciones hash, entre otros. En el peor caso, las latencias asociadas por cada consulta son del orden del camino de caches establecido. Dependiendo de cuál es la decisión de nodos a actualizar, es la utilización de entradas y la tolerancia a fallas que se tiene. El número de entradas disponibles varía entre NE y $NE \cdot NCS$. La variación depende del número de réplicas que tenga asociada cada consulta en el Servicio de Cache.

4. *Matriz*. En vez de particionar el espacio de consulta de acuerdo al número de nodos totales del CS, se definen P particiones (con $P < NCS$) y cada partición es replicada D veces (con $D = NCS/P$). Cuando una consulta arriba a FS_i, una de las P particiones es escogida mediante un método determinístico (como $hash(q)\,\%P$). La elección de una partición significa que esa consulta se encuentra en cualquiera de las réplicas asociadas a esa partición, ya que con el método determinístico cada partición maneja un conjunto disjunto de consultas. Luego, en la partición, una de las D réplicas es seleccionada aleatoriamente (la consulta es hecha solamente en ese nodo), y la respuesta (*hit* o *miss*) retorna al FS_i. Este método exhibe menos desbalance y la carga de una partición es absorbida en partes iguales por las D réplicas. Para alcanzar una alta tasa de *hit*, cada partición debe ser actualizada continuamente vía un protocolo de consistencia. La idea clave del protocolo de consistencia es implementar la replicación de cada partición [CLV11, Wan99], y así tener una probabilidad mayor de que la selección aleatoria de una de las D réplicas sea un *hit*. La implementación de un protocolo de consistencia en servicios replicados necesita transferencia de información a través de la red de comunicación para alcanzar un estado consistente entre réplicas [YV00]. La operación de *inserción* es enviada sólo a la misma réplica que generó el *miss* anteriormente, ya que mediante el protocolo de consistencia esta actualización será realizada en las demás particiones. La latencia asociada a una consulta es 1 búsqueda, pero el número de entradas totales disponibles para cache es $NE \cdot NCS/D$. Como cada partición está replicada, se tiene que el servicio es tolerante a fallas.

Tabla 2: Tabla de métodos y características.

Característica	Método			
	Difusión	Partición	Jerarquía	Matriz
N° de entradas totales para cache	NE	$NE \cdot NCS$	$NE..NE \cdot NCS$	$NE \cdot NCS/D$
Latencias	NCS	1	$\log_d NCS$	1
Tolerancia a fallas	Sí		Sí	Sí
Protocolo de consistencia				Sí

A continuación, en la Tabla 2 se exponen los métodos anteriormente descritos junto con una lista de atributos para caracterizar cada método. Esta tabla servirá posteriormente para construir una solución que permita tomar en consideración las ventajas y desventajas de cada método.

3.3. Estrategias para Cache

Pastry [RD01b] es un esquema de ruteo y localización para sistemas P2P. Las características de este esquema es que es escalable, descentralizado, tolerante a fallas y confiable. Cada nodo de Pastry tiene un identificador numérico único (*nodeId* de 128 bits). Cuando arriba una petición, se le asocia una clave numérica, y cada nodo en Pastry rutea los mensajes al *nodeId* que está numéricamente más cercano al valor de la clave numérica. Sólo se consideran los nodos activos en Pastry. Una propiedad importante de este sistema de ruteo es que el número esperado de pasos de ruteo es menor que $\lceil \log_{2^b} N \rceil$, donde N es el número de nodos Pastry en la red bajo operación normal y b es un parámetro de configuración, típicamente con valor 4. Al igual que en Consistent Hashing, el identificador *nodeId* es mapeado a un espacio circular de rango 0 a $2^{128} - 1$. Para el ruteo de mensajes, se utilizan tablas relacionadas a los nodos y sus prefijos. En cada nodo, la cantidad de entradas en las tablas de ruteo es $O(\log N)$, así como también el número de pasos de ruteo para la búsqueda de una clave. Pastry ha sido utilizado en la implementación de PAT (sistema de almacenamiento persistente distribuido), SCRIBE (sistema de publicación/suscripción escalable) y Squirrel [LCP+05].

Ratnasamy *et. al* [RFH+01] definen CAN (*Content-Addressable Network*), el cual es una infraestructura distribuida que provee la funcionalidad de una tabla hash a gran escala. CAN es escalable, tolerante a fallas y completamente auto-organizativo. El diseño se basa en un espacio de coordenadas Cartesiano virtual de d dimensiones sobre una red torus de d dimensiones. Este espacio es lógico y no guarda relación con las coordenadas físicas de los nodos. La idea es que este espacio es particionado dinámicamente entre todos los nodos del servicio, tal que cada nodo sea el único dueño de una zona del espacio total. Los pares clave/valor son mapeados a un punto P en el espacio virtual usando una función hash. Luego, el par es almacenado en el nodo que sea dueño de esa zona. Las peticiones generadas en cualquier nodo (punto del espacio virtual) son ruteadas a través del espacio sólo permitiendo la transferencia de mensajes entre zonas adjacentes (vecinos en la red) hasta encontrar el punto P. Cada nodo sólo mantiene información de ruteo de los vecinos en el espacio lógico. Para un espacio de d dimensiones particionado en n zonas iguales, el número de latencias o accesos a nodos para encontrar un elemento es $(d/4)(n^{1/d})$, es decir $O(dn^{1/d})$, y cada nodo mantiene información de $2d$ vecinos ($O(d)$). Para mejorar la disponibilidad de los datos, se utilizan k funciones hash para mapear un ítem a k puntos distintos en el espacio. Los ítemes más frecuentemente accedidos son manejados a través de cache y replicación.

PAST [DR01] es una utilidad P2P de almacenamiento de gran escala en Internet que provee escalabilidad, alta disponibilidad, además de persistencia y seguridad. Los nodos en PAST forman una red sobrepuesta auto-organizativa, con múltiples copias de los ítemes insertados. El número de nodos visitados para la búsqueda de un ítem es a lo más logarítmico en el número de nodos del servicio. La replicación de elementos, a través de estrategias aleatorizadas, se utiliza como mecanismo de balance de carga y alta disponibilidad. También se utiliza caching de elementos populares en distintos nodos. A cada nodo se asigna un identificador de 128 bits, y a cada ítem se le asigna una clave de 160 bits. Para replicar, se rutea el ítem a k nodos cuyos identificadores estén numéricamente más cercanos a los 128 bits más significativos del identificador del ítem. En [RD01a] se describe en más detalle el mecanismo de almacenamiento y caching de PAST. Los autores mencionan que un elemento altamente popular podría

necesitar más de k copias, para lo cual detallan que más copias podrían asignarse más cerca de los grupos de usuarios que estén haciendo esta petición popular.

Kademlia [MM02b] es un sistema P2P de búsqueda y almacenamiento de pares clave/valor, cuyo ruteo de consultas y localización de nodos se basa en una nueva métrica XOR para la distancia entre dos puntos del espacio de claves. Este sistema utiliza una función hash de 160 bits para las claves y nodos, y la idea es para cada clave encontrar en nodo más cercano. El concepto de cercanía varía de sistema en sistema, por ejemplo en Consistent Hashing el concepto de cercanía para una clave es el nodo más cercano siguiendo el sentido del reloj en el anillo. En Kademlia, la distancia entre dos identificadores x e y está dada por $d(x, y) = x \oplus y$. La métrica definida tiene algunas propiedades como simetría y cumple la propiedad triangular. Con esta métrica, los autores definen los mecanismos de ruteo, en el cual se contactan $O(\log n)$ nodos para una búsqueda.

Stoica *et al.* [SMLN$^+$03] presentan *Chord*, un protocolo de búsqueda distribuido, el cual permite localizar eficientemente el nodo que almacena un ítem en aplicaciones Peer-to-Peer (P2P). Chord asigna a cada nodo y clave un identificador fijo de m-bits usando una función hash, tal como SHA-1. En el caso de los nodos, se hace el hash por ejemplo de su dirección IP. Estos identificadores se mapean a un círculo con rango [0,2^m). Los autores discuten que la función hash de m-bits debe ser suficientemente grande para prevenir colisiones. Chord utiliza nodos virtuales para mantener el número de claves por nodo uniforme. Un problema de Consistent Hashing es que cada nodo es consciente de los otros nodos. Este enfoque no escala cuando se tiene una gran cantidad de nodos. Por el contrario, Chord necesita información de ruteo de unos pocos nodos (orden logarítmico). Como la tabla de ruteo es distribuida, un nodo se comunica con otros nodos para realizar una búsqueda. La idea es que en régimen, cada nodo mantiene información sólo de otros nodos y resuelve las búsquedas mediante el envío de mensajes a otros nodos. Esta información es actualizada a medida que nodos ingresan/salen del servicio. El esquema descrito en el trabajo requiere almacenar información con costo $O(\log N)$ en cada nodo, pero con esa información se asegura que la entrega de mensajes ocurre en $O(\log N)$ pasos con alta probabilidad (al menos $1 - O(1/N)$). Aplicaciones que utilizan Chord para la funcionalidad de

localización de ítemes distribuido son CFS (Cooperative File System) y Chord-based DNS [LCP+05].

Koorde [KK03] es una tabla hash distribuida basada en Chord y, a través de la utilización de grafos de Brujin, se mejoran las latencias asociadas a la búsqueda de elementos. Como en Chord, Koorde usa Consistent Hashing para mapear claves a nodos. El costo de una búsqueda en un servicio compuesto de n nodos, está relacionado con la cantidad de información que tenga sobre sus nodos vecinos. Si se tiene sólo 2 vecinos por nodo, entonces el número de latencias es $O(\log n)$. Si se tiene información de $O(\log n)$ vecinos, el número de visitas a otros nodos es $O(\log n / \log \log n)$.

Tapestry [ZHS+04] es una infraestructura de ruteo de mensajes para sistemas P2P que es eficiente, escalable e independiente de la localización. Una característica principal de Tapestry es que para el ruteo de mensajes sólo se utilizan recursos locales a los nodos. A diferencia de Chord y CAN, Tapestry toma en consideración las distancias lógicas para construir las tablas de ruteo. Estas tablas construidas son localmente óptimas desde el proceso de inicialización. Tapestry usa tablas locales en cada nodo para rutear los mensajes, cuyo número de entradas esperado es $O(\log N)$. Por otro lado, el método que se define en el trabajo garantiza que cualquier nodo en el servicio es alcanzado en a lo más $O(\log N)$ pasos lógicos (N es el número de nodos). Esta infraestructura ha permitido el despliegue de diversas aplicaciones, tal como OceanStore y Bayeux [LCP+05], entre otros.

En [RS04] se propone Beehive, un *overlay* (o red superpuesta) para sistemas P2P basada en replicación proactiva, que permite realizar búsqueda en $O(1)$ para distribuciones zipfianas. La solución tiene bajo requerimientos de almacenamiento, bajo *overhead* en comunicación y carga de trabajo. También se adapta a los abruptos cambios en la popularidad de los objetos. La principal estrategia es la replicación proactiva, que implica la propagación de copias de ciertos objetos a través de los nodos. Los autores indican que existe un compromiso entre la replicación y los recursos utilizados. A diferencia de otros sistemas, el nivel de replicación es determinado por cada objeto, el cual es establecido mediante optimización que considera las características de la distribución de consultas. Una vez establecido este nivel de replicación, las copias son situadas en ciertos nodos que comparten un esquema de prefijos con

el objeto. La meta de los autores es establecer el nivel mínimo de replicación de los objetos de acuerdo a su importancia para que el costo de una búsqueda en promedio sea un número constante de peticiones a otros nodos. Obviamente, la estrategia óptima involucra replicar más veces los objetos más populares y menos veces los objetos poco frecuentes. Los autores obtienen estos niveles analizando y modelando la distribución zipfiana de consultas. La propuesta se implementa mediante un algoritmo de replicación distribuido compuesto de dos fases: análisis y replicación. Los autores no consideran aspectos de balance de carga en la disposición de los ítemes replicados (al ser orientado a sistemas P2P).

Cycloid [SXC06] es una arquitectura P2P que permite la búsqueda en $O(d)$ usando $O(1)$ espacio con información relacionada a vecinos. Para ello, utilizan un grafo *CCC* d-dimensional (*cube-connected cycle*) que consiste en un cubo de d dimensiones en el que se reemplaza cada vértice del cubo por un ciclo de d nodos. Este grafo contiene $d \cdot 2^d$ nodos en total, cada uno de grado 3. Cycloid utiliza Consistent Hashing para hacer el mapeo entre claves y nodos, los cuales tienen identificadores uniformemente distribuidos en un espacio de identificadores de tamaño $d \cdot 2^d$. Los autores definen algoritmos de ruteo basados en la topología del grafo CCC y a la asignación de los identificadores de claves y nodos. También se alcanza un mejor balance de carga utilizando replicación en el nodo (numéricamente) más cercano. Los autores también estudian el balance de carga, definido como el número de consultas recibidas para búsqueda desde nodos diferentes.

Dynamo de Amazon es un sistema de almacenamiento de clave-valor de alta disponibilidad [DHJ+07a, DHJ+07b]. Uno de sus principales preocupaciones es el particionamiento y replicación de los datos. El esquema de particionamiento para distribuir la carga entre los nodos se basa en Consistent Hashing (CH), donde cada nodo tiene un valor aleatorio que representa una posición fija en el anillo de CH. También, cada ítem es mapeado al mismo anillo usando una función hash sobre su clave. El nodo responsable por esta clave es el primer nodo encontrado siguiendo el sentido del reloj en el anillo. Entonces, cada nodo está a cargo de una región fija entre él y su predecesor inmediato. Para lidiar con el problema de la heterogeneidad, los autores toman ventaja de los nodos virtuales (múltiples copias de cada nodo son asignadas al anillo

de acuerdo a su poder de cómputo). El número de réplicas de cada ítem es configurado usando un parámetro global, R, y el nodo coordinador está a cargo de replicar el ítem a los siguientes $R-1$ nodos sucesores inmediatos siguiendo el sentido del reloj. Esto es aplicado a todos los ítemes. El diseño de Dynamo asume que las distribuciones sesgadas (zipfianas) no serán un problema, ya que los ítemes más frecuentes de la distribución serán suficientes para repartir la carga uniformemente en los nodos. Esto significa que si se distribuyen los ítemes más accedidos en las particiones, la carga experimentada por cada partición será balanceada. No existen modificaciones de los rangos de los nodos, excepto cuando los nodos entran/salen del servicio. En este último caso, el único rango modificado es el sucesor inmediato del punto donde el nodo fue removido/insertado en el anillo (esto corresponde a una propiedad de CH).

ROAR [RHHR09] es un algoritmo distribuido que permite re-configurar un Servicio de Índice (IS) en línea (*on-the-fly*). Los autores declaran que existe un equilibrio entre el nivel de particionamiento y replicación (ver capítulo 2). El nivel de particionado reduce el tiempo de respuesta de una consulta, mientras que el nivel de replicación incrementa el *throughput* del sistema. A medida que se aumentan las particiones, aumentan las latencias asociadas al resolver una consulta. El objetivo es encontrar el punto en el cual el sistema alcanza una latencia de búsqueda balanceada. La estrategia hace frente a la agregación o remoción de servidores, enfrenta fallos, balancea la carga y trabaja con nodos heterogéneos. Todas estas características son logradas sin detener el servicio. También, el algoritmo maneja picos de consultas (*query spikes*). Los autores definen un intervalo $[0, 1)$ y cada nodo es responsable con una sección contigua del intervalo. La idea fundamental es que cada objeto es almacenado en todos los nodos que intersectan un arco de tamaño $1/P$ (donde P es el número de particiones). La tarea de balance de carga es manejada por un algoritmo simple, el cual consiste en reducir el rango de los nodos sobrecargados. En este trabajo el tamaño del intervalo asignado a cada nodo es de acuerdo a su poder de procesamiento.

Cassandra [LM10] es un sistema de almacenamiento distribuido que utiliza una gran cantidad de servidores básicos (*commodity*). El objetivo es proveer un servicio con alta disponibilidad y sin punto único de fallo. Este sistema es utilizado en

Facebook. Una de las principales características es la capacidad de escalar incrementalmente. Cassandra particiona los datos en los nodos usando Consistent Hashing, pero utiliza una función hash que preserva el orden. Los autores mencionan que la asignación aleatoria de nodos en el anillo lleva a una distribución no uniforme de datos y carga de trabajo. Existen dos formas de solución: nodos virtuales y mover nodos con poca carga a sectores del anillo con alta carga. Cassandra usa esta última solución. Este sistema utiliza replicación para alcanzar alta disponibilidad. Cada ítem es replicado en N nodos, donde N es un factor de replicación que se configura. La replicación está a cargo de un coordinador. Cassandra provee varias políticas de replicación.

El trabajo en [NS09] investiga el problema de balance de carga en *crawlers* distribuidos. Los autores enfatizan que el balance de carga no debe ser hecho considerando simples funciones hash o Consistent Hashing (CH) debido a la distribución Ley de Potencias (*Power Law*) de la estructura de La Web. Para esto, se analizan los trasfondos matemáticos de la Ley de Potencias y luego son aplicadas a CH. Las conclusiones son que existe la necesidad de considerar la estructura de La Web para alcanzar "equidad" en la distribución de la carga de trabajo. También en el trabajo de [BCSV04] se utiliza Consistent Hashing en el contexto de Web Crawling, para la asignación de URL/*hosts* a nodos de procesamiento.

3.4. Análisis

Hasta donde se ha investigado en este trabajo, no existe trabajo realizado en el área específica de servicios de cache distribuidos en clusters (conjunto de nodos comunicados por una red de alta velocidad), tal como los que se implementan en *data-centers*. Como fue mencionado en el capítulo 1, existen trabajos que analizan algoritmos y estructuras para cache en forma distribuida, pero ningún trabajo analiza temas de balance de carga y tolerancia a fallos. La gran parte del trabajo que aborda sistemas de cache distribuidos se encuentran orientados a sistemas *Peer-to-Peer* (P2P). La diferencia fundamental es que el retardo de comunicación y el procesamiento de peticiones tiene un retardo mucho más alto en los sistemas P2P. Además,

la determinación de qué nodos dejan el servicio y la carga de trabajo experimentada por cada nodo es mucho más rápida y fácil en un cluster.

Como las consultas de usuario siguen distribuciones Zipfianas y varían a través del tiempo, se deben considerar mecanismos de balance de carga dinámico en la construcción de servicios de cache distribuidos. Esto se debe a que, tanto la replicación total de ítemes como la no replicación, sumado a la disposición fija de puntos en el anillo para representar nodos, no son las herramientas óptimas para lidiar con las características del tráfico de usuario. Este balance de carga debe ser en tiempo real y tomando en consideración la carga reportada de cada nodo. A esto hay que agregar el hecho de la utilización de Consistent Hashing como un estándar *de facto*. Esto implica que el mecanismo de balance dinámico debe considerar una dependencia fuerte entre una consulta y un nodo, con el fin de maximizar la tasa de *hit*.

Como se ha visto en la revisión del estado del arte, existen algunas características transversales en la mayoría de trabajos analizados:

1. Uso transversal del concepto de Consistent Hashing para hacer el mapeo entre consultas y N nodos (rango circular de 0 a $2^h - 1$, con h la cantidad de bits de la función hash).

2. Varios autores mencionan que la definición primitiva de Consistent Hashing tiene problemas: (i) se asume distribución uniforme de consultas; (ii) se considera una distribución de carga uniforme en los nodos. Se debe recalcar que estos puntos no se refieren a lo mismo, ya que (i) implica que si se distribuye el universo total de consultas en un punto en el anillo (sin considerar frecuencias), el número de claves por nodo es equivalente. Por otro lado, el punto (ii) sí considera frecuencias, ya que la carga de trabajo de los nodos tiene relación con la cantidad de peticiones hechas al nodo.

3. La distribución de los puntos en el anillo siempre es fija, ya que variarlas implicaría cambios en las tablas de ruteo de los sistemas P2P. La asignación de puntos en el anillo es en forma aleatoria o equidistante.

4. La aparición de *hot spots* (peticiones con alta frecuencia) es manejada a través

de la replicación de las respuestas a las peticiones en múltiples nodos (en los nodos que conforman el camino al nodo responsable).

5. Visita de $O(\log N)$ nodos para consultar al nodo responsable de la petición si está o no en cache.

6. Cada entrada se replica para proveer tolerancia a fallas y alta disponibilidad. Existen esquemas fijos (todas las consultas se replican R veces) y variables (cada consulta se replica en base a características, como su popularidad).

7. Utilización de nodos virtuales como un mecanismo de balance de carga.

8. Poco análisis de la carga experimentada por los nodos, al ser sistemas P2P.

El número de accesos que se realiza en el Servicio de Cache para determinar la existencia de una respuesta, también debe ser considerado, ya que cada vez que se consulta un nodo se debe realizar una búsqueda en la memoria local del nodo, es decir, tiene un costo asociado. Además, se utiliza en mayor medida la red de comunicación. Lo óptimo es realizar un solo acceso, tal como en la definición de Consistent Hashing.

A grandes rasgos las principales diferencias entre los trabajos descritos y las soluciones propuestas en este trabajo:

- Se utilizan mecanismos de balance de carga dinámicos considerando la infraestructura de Consistent Hashing, específicamente manejando los rangos asociados a los nodos. Esto es para que no se introduzcan cambios profundos en el ruteo de consultas. El tamaño del rango asignado a cada nodo no está dado por la cantidad de claves asociadas a cada uno.

- Se determinan algoritmos para balancear la carga entre los nodos considerando la topología anillo. De este modo, se tiene un impacto muy bajo en la tasa de *hit*.

- Se asume una distribución zipfiana de consultas de usuario.

- No se utilizan nodos virtuales, ya que (i) impone *overhead* adicional, y (ii) no es claro cómo distribuir la carga de trabajo experimentada por un nodo en las distintas secciones correspondientes a los nodos virtuales. Además, según evidencia empírica analizada por el autor, no tiene un impacto real en los resultados.

- No se evalúa la estrategia "poder de d elecciones". Esta estrategia implica que en cada arribo de una instancia de una consulta, se debe evaluar el estado de d nodos y asignarla a la menos cargada. Esto es imposible con una tasa de arribo de miles de consultas por segundo. Sin embargo, una estrategia similar es evaluada, que consiste en que la decisión de asignar una petición a uno de d nodos no es por carga de trabajo sino aleatoria.

- No se utilizan "random trees", ya que (i) implica visita logarítmica de nodos, y (ii) por la imposibilidad de disponer de contadores asociados en cada nodo para determinar si una consulta debe ser alojada en el nodo o no.

- Se analizan y evalúan estrategias que realizan sólo una visita para localizar los ítemes en el Servicio de Cache, ya que así se reducen las latencias asociadas al procesamiento de consultas y la comparación es más justa en términos de costos.

- Es posible obtener la utilización de los nodos y su estado (vivo/muerto) para efectos de balance de carga, pero cada cierto intervalo de tiempo (orden de segundos o minutos). Esta funcionalidad está disponible en los sistemas de gestión de clusters (como el mecanismo de *heartbeats* descrito anteriormente).

Problemas de Balance de Carga

4.1. Evidencia

El desbalance, en términos de carga de trabajo, de los nodos de un Servicio de Cache es un hecho. Según evidencia recolectada, con los métodos actualmente utilizados se experimentan cambios en la carga de trabajo de cada uno de los nodos. Estos cambios suceden a nivel de días y horas, incluso llegando a cambios abruptos en cuestión de minutos.

Dado el esquema de asignación de consultas en el Servicio de Cache detallado en el capítulo 3, un aumento en la carga de trabajo de un nodo se debe a un incremento en el número de instancias de una consulta en particular (o un grupo de consultas si ellas están asignadas al mismo nodo). Al aumentar el número de instancias de consultas recibidas por un nodo, se aumenta la utilización de este nodo. Si se tiene un Servicio de Cache balanceado y se presentan casos como el descrito anteriormente, entonces se genera desbalance en el servicio.

En la Figura 5(a) se observa la carga de trabajo de cada nodo para un sistema de cache compuesto por 20 particiones con Consistent Hashing. La ejecución corresponde a un mes completo de consultas (Marzo de 2012) y cada curva en el gráfico representa la carga de una partición. Además, la carga de trabajo de cada nodo se presenta como una medida entre 0 y 1, con 0 sin utilización y 1 a máxima capacidad. Se observa claramente la variación de la carga de trabajo a través del período de tiempo, apareciendo episodios de abruptas alzas en términos de carga de trabajo, incluso llegando a la completa utilización de algunos nodos.

Para examinar más detalladamente el desbalance, las Figuras 5(b), (c) y (d) muestran la carga de trabajo de 1 semana, 1 día y 1 hora respectivamente (mismo mes de la Figura 5(a)). Con las figuras anteriores, se concluye que el desbalance es un problema constante de corto, mediano y largo plazo. Con esto, es posible inferir que

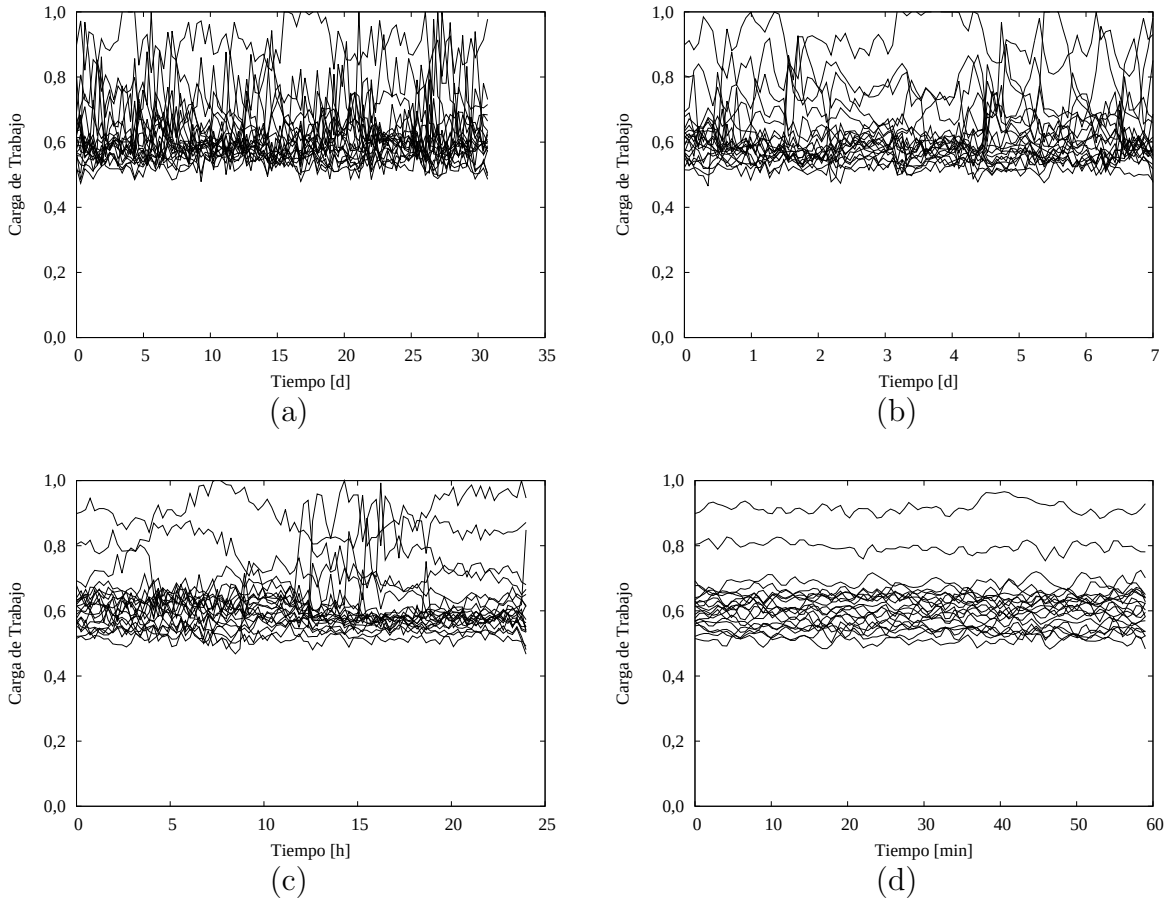


Figura 5: Carga de trabajo a nivel de partición experimentada por un CS de 20 particiones usando Consistent Hashing. Los resultados se muestran en distintas escalas: (a) mes; (b) semana; (c) día; (d) hora. *Log* correspondiente a Marzo de 2012.

la variable temporal es un factor importante a estudiar para solucionar el desbalance. A este desbalance se le denomina *desbalance dinámico* en este trabajo.

Por otro lado, se tiene que la carga de trabajo recibida por los nodos en un período de tiempo no es uniforme. Es decir, la suma de las peticiones hechas a un nodo en un período de tiempo, es (generalmente) distinta a la suma de peticiones hechas en otros nodos del Servicio de Cache en el mismo período de tiempo. Esto se debe principalmente al comportamiento Zipfiano de las consultas de usuario. Esto implica, que independiente del período de tiempo escogido, siempre habrá una asignación injusta de las peticiones a los nodos del Servicio de Cache.

Para evidenciar esta situación, en la Figura 6 se muestra el total de peticiones por nodo en un Servicio de Cache de $NCS = 20$ nodos, en diferentes intervalos de tiempo (mes completo, día, semana y hora). La situación que se observa es que durante el mes, la asignación total de peticiones a cada nodo no es balanceada. Esto también es observado a escalas de tiempo distintas como es presentado en la misma figura. Se puede concluir con la evidencia presentada que la asignación injusta de peticiones a los nodos existe, debido al comportamiento Zipf de las consultas de usuario, y es independiente de la escala temporal analizada. A este tipo de desbalance se referirá como *desbalance estructural*.

Para resaltar aún más la situación detallada anteriormente, la Figura 7(a) muestra cómo decae la eficiencia en la asignación de peticiones a medida que se aumenta el número de particiones. La eficiencia es calculada como el promedio del número de peticiones hechas a todos los nodos dividido por el número máximo de peticiones realizadas a un nodo. Este número varía entre 0 y 1, con 0 totalmente ineficiente y 1 totalmente eficiente. La idea de un sistema eficiente (eficiencia cercana a 1) es que el número máximo de peticiones en un nodo esté muy cercano al promedio de peticiones experimentado por todo el servicio. De la figura se concluye que la asignación injusta de peticiones se torna más grave a medida que aumentan las particiones.

Por otro lado, para evidenciar el impacto de las consultas frecuentes en el desempeño del servicio, la Figura 7(b) muestra cómo evoluciona la eficiencia de un servicio compuesto por $NCS = 20$ nodos a medida que se excluyen las N consultas más frecuentes del *log* ($N \geq 0$). La forma de obtener los resultados consta de los siguientes

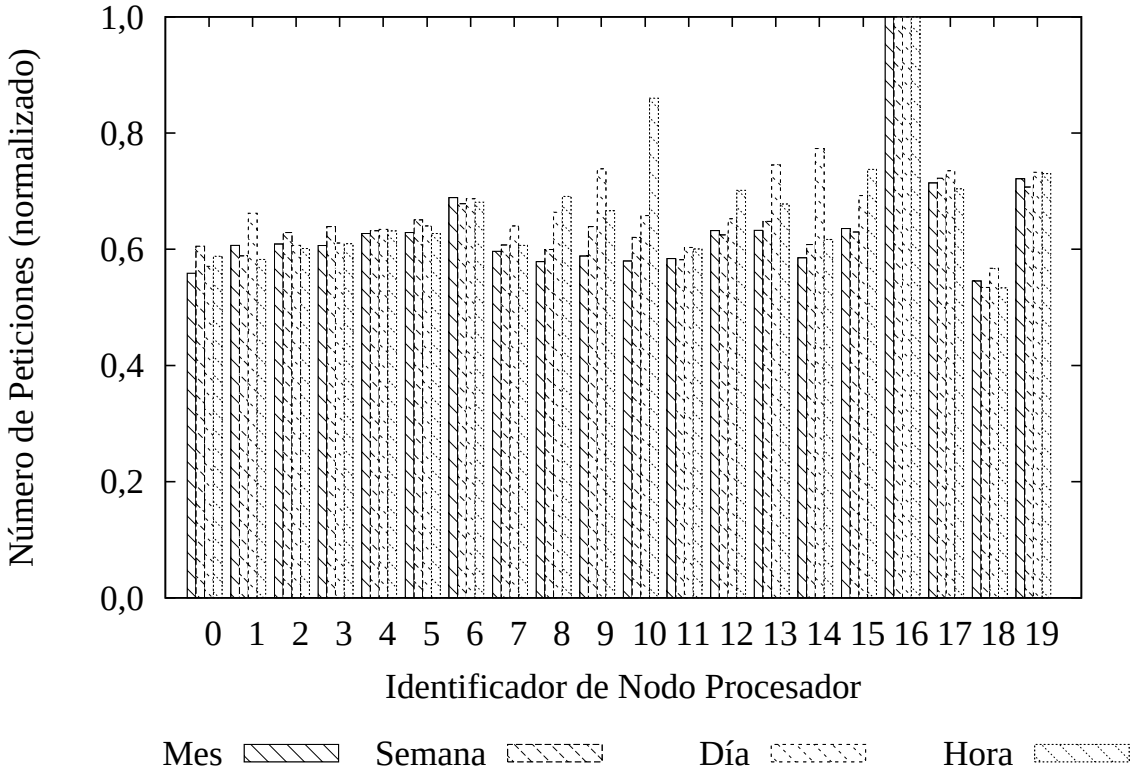


Figura 6: Número de peticiones normalizado en un servicio compuesto de 20 nodos utilizando el *log* de Marzo 2012 (identificador de nodo desde 0 a 19).

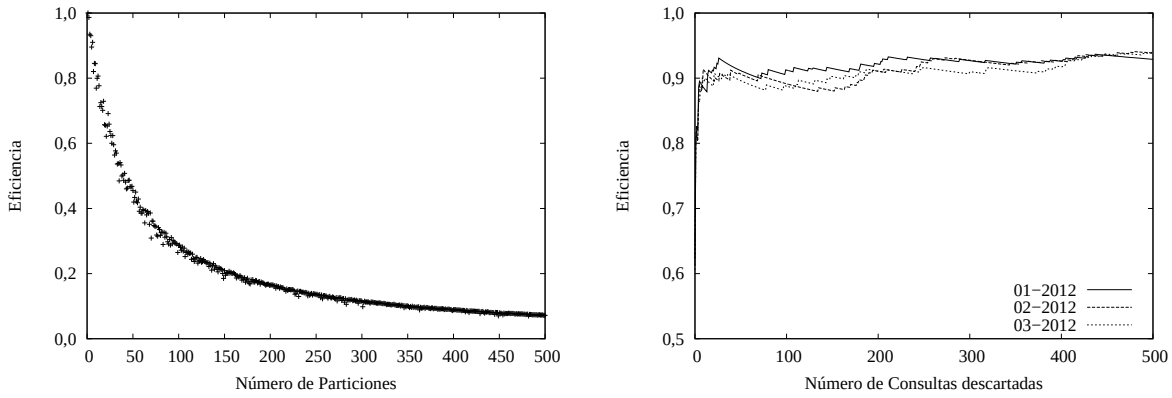


Figura 7: (a) Eficiencia de la asignación de consultas a medida que aumenta el número de particiones de Servicio de Cache (Enero de 2012). (b) Eficiencia a medida que se descartan las N consultas más frecuentes en un Servicio de Cache de $NCS = 20$ nodos (Enero, Febrero y Marzo de 2012).

pasos: (i) recorrer el *log* completo un mes, obtener las consultas distintas y ordenarlas por frecuencia; (ii) ejecutar el *log* de consultas de excluyendo en cada ejecución las N consultas más frecuentes. Los meses utilizados para obtener los datos de la figura son Enero, Febrero y Marzo de 2012. Se observa que las consultas más frecuentes sí tienen un impacto en el desempeño del Servicio de Cache. Aún más, estas consultas son las que tienen mayor impacto en la disminución de la eficiencia del servicio en un período de tiempo. Esto refuerza la idea de la distribución injusta de peticiones en los nodos del servicio.

La atención de los usuarios de los WSEs cambia en función de eventos sociales, ecónomicos, deportivos y naturales, entre otros. Es decir, cuando un evento es importante según la percepción del usuario, éste comienza a buscar información en relación a ese evento. La principal herramienta para buscar y recopilar información en la actualidad es el WSE. La consecuencia de esto es que en un intervalo de tiempo reducido, el WSE recibe una alta cantidad de peticiones en torno a un tópico en particular. En general, los usuarios tienen intereses muy variados, pero la atracción de una gran cantidad de usuarios hacia tópicos específicos es un suceso muy común que debe ser enfrentado por los WSEs. El tópico de interés (una celebridad, una localidad, un nuevo producto tecnológico, etc.) es compuesto de múltiples peticiones de

los usuarios asociadas a este tópico. Por ejemplo, cuando una celebridad muere, los usuarios comienzan a realizar consultas sobre sus fotografías, su biografía, motivos de la muerte, discografía (en caso de cantantes), películas (en caso de actores), etc. Es decir, un tópico de interés no genera sólo una clase de consulta (un *string*), sino que un conjunto de clases de consultas (varios *strings*) que giran en torno al tópico en cuestión. A estas consultas se les denomina *consultas en ráfaga*.

Para detallar la situación anterior y, aún más importante, el impacto de estos eventos en el Servicio de Cache, en la Figura 8(a) y (b) se muestra el desempeño, en términos de carga de trabajo, de un Servicio de Cache compuesto de 20 particiones. Para generar las figuras mencionadas, se utilizaron dos meses completos de ejecución correspondientes a Enero y Febrero de 2012, respectivamente. En ambas figuras, además de los cambios constantes de la carga de trabajo, se evidencian alzas abruptas en ciertos nodos. Para intentar encontrar una relación entre las consultas en ráfaga, en la Figura 8(c) y (d) se observa la frecuencia de un conjunto de consultas que se consideran como ráfaga (para los mismos meses anteriores). La idea a expresar es que la carga de trabajo del mes Enero de 2012 de la Figura 8(a), está influenciada por la alta frecuencia de un conjunto de consultas (Figura 8(c)). Se debe notar que ambas figuras están en la misma escala temporal. La misma situación sucede con las Figuras 8(b) y (d) para el mes Febrero de 2012.

Como se observa en las figuras, una consulta en ráfaga tiene un incremento abrupto de su frecuencia en un intervalo de tiempo reducido, alcanzando un *peak* y luego disminuyendo su frecuencia a medida que el tópico pierde la atención de los usuarios (los usuarios han satisfecho sus necesidades de información). Este incremento abrupto impacta el desempeño del Servicio de Cache en términos de balance, independientemente si el servicio se encuentra en un estado de balance o no. Si el servicio se encuentra desbalanceado, los efectos son mayores. Además, un alza abrupta de tráfico a un nodo en particular puede ocasionar un colapso de este nodo, y peor aún, una reacción en cadena de colapso de nodos (ya que las peticiones del nodo colapsado son re-direccionadas a un único nodo vecino).

Las consultas en ráfaga son un riesgo para el buen desempeño del Servicio de Cache. Para evitar las consecuencias, las principales medidas de mitigación son la

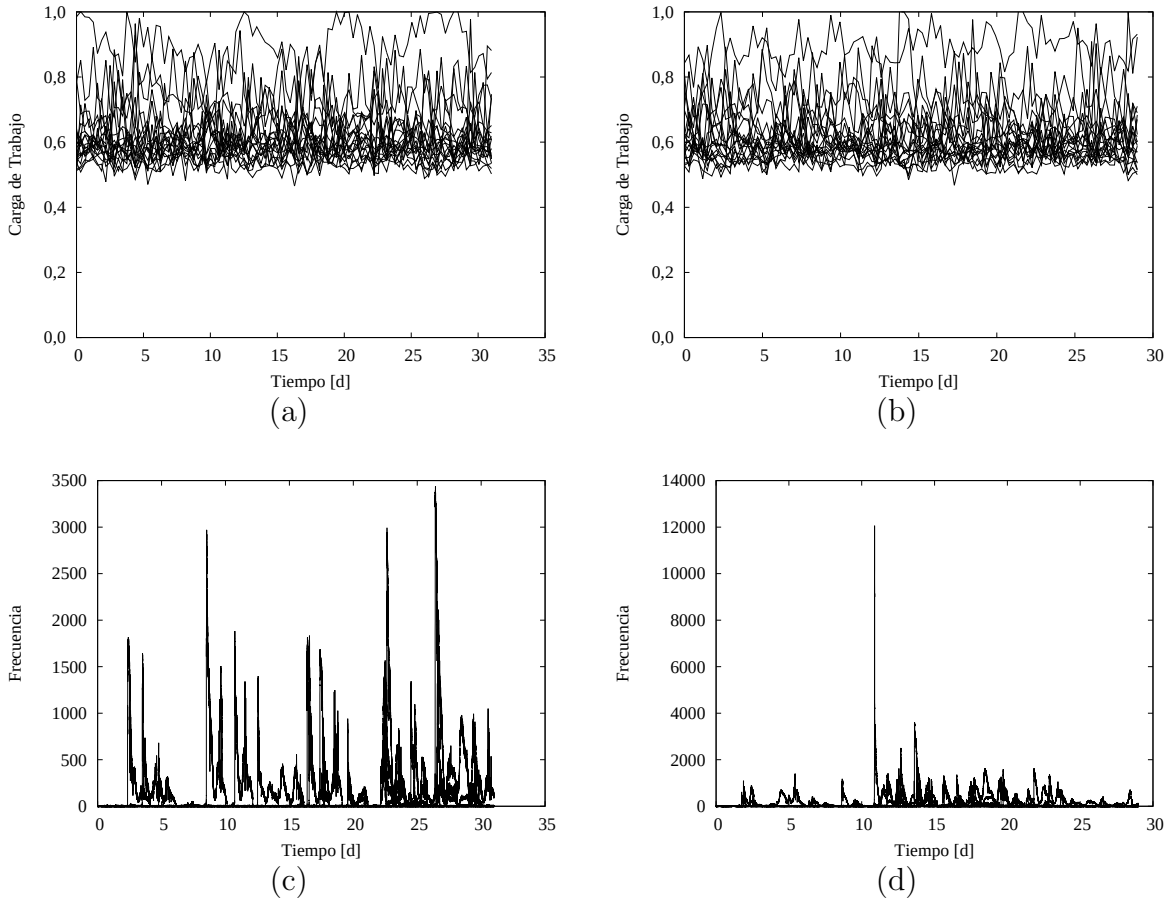


Figura 8: Carga de trabajo en un Servicio de Cache compuesto de 20 particiones para (a) Enero de 2012 y (b) Febrero de 2012. Frecuencia absoluta de un conjunto de 10 consultas en ráfaga en (c) Enero de 2012 y (d) Febrero de 2012.

detección temprana de este tipo de eventos y la distribución en múltiples nodos de las instancias de la consulta en ráfaga.

Finalmente, las fallas en los componentes computacionales deben considerarse, ya que la tasa de fallos aumenta en sistemas que están sometidos a altas cargas de trabajo. Además, como se mencionó anteriormente, existen episodios que podrían provocan congestión y posterior colapso en los nodos. Principalmente, este trabajo se enfoca en las fallas a nivel de nodo, es decir, una falla implica la pérdida total del nodo en cuestión y toda su información residente en memoria. Este último punto es el más importante: volatilidad de la información relevante.

Cabe recordar que en cada nodo de un Servicio de Cache, casi toda la memoria está dedicada a alojar respuestas pre-computadas a consultas realizadas anteriormente por los usuarios. La idea clave de esto es disminuir la utilización de recursos computacionales. Para hacer más rápido el proceso, esta información sólo se mantiene en memoria y ante una falla del nodo, esta información se pierde. Como cada entrada en el nodo representa respuestas pre-computadas, la pérdida de cada una de estas entradas implica que los recursos computacionales para generar esa respuesta deben ser nuevamente utilizados.

Para evidenciar el impacto de los fallos en el Servicio de Cache, la Figura 9 muestra el desempeño de un Servicio de Cache compuesto de 10 particiones y 8 réplicas por partición ($NCS = 80$ nodos en total). La Figura 9(a) muestra la situación ideal sin fallos en el servicio para los primeros 7 días del mes de Diciembre de 2011. Por otro lado, las Figuras 9(b), (c) y (d) detallan el desempeño del servicio en el mismo período de tiempo para una situación en que falla el 10 % , 20 % y 30 % de los nodos respectivamente (8, 16 y 24 nodos). Las fallas de los nodos son aleatorias en cada caso y ocurren simultáneamente (cerca del punto $x = 3$).

En la literatura, una medida que representa una caracterización de los fallos es el *tiempo medio entre fallos* (MTBF, *mean time between failures*). Esta métrica tiene correlación con el tipo e intensidad de la carga de trabajo experimentada por un nodo [SG06]. Por ejemplo, a medida que se incrementa la carga de trabajo, el MTBF disminuye. Diversos estudios que examinan la MTBF muestran que esta medida puede

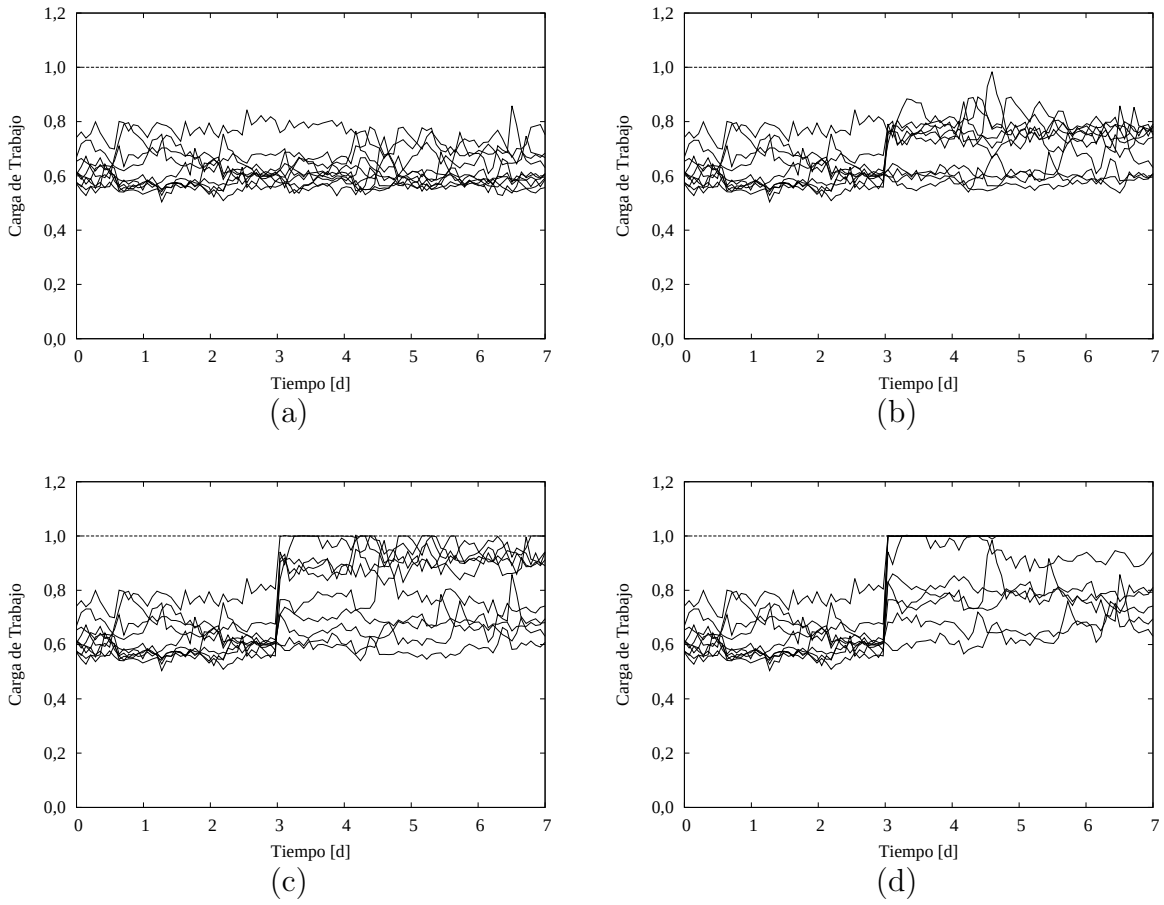


Figura 9: Desempeño de un Servicio de Cache con 10 particiones y 8 réplicas: (a) sin fallas; (b) 10 % de fallos; (c) 20 % de fallos; (d) 30 % de fallos.

variar desde varias horas [WMES08], pasando por algunos minutos [IJSE07] hasta varias fallas por minuto [FC07] en supercomputadores y centros de datos de gran escala. La idea de la inserción de un 10 %, 20 % y 30 % de fallas es observar la degradación del servicio con un incremento continuo de ellas, pero estas situaciones, aunque remotas, podrían suceder en aplicaciones de gran escala dados los antecedentes anteriores.

Específicamente, en la Figura 9(b) se observa un alza en la carga de trabajo de la mayoría de las particiones. Luego, en la Figura 9(c), cuando falla el 20 % de los nodos, se observa que algunas particiones se mantienen durante un período de tiempo con una utilización del 100 %, y también se observa que existe un alza en la carga

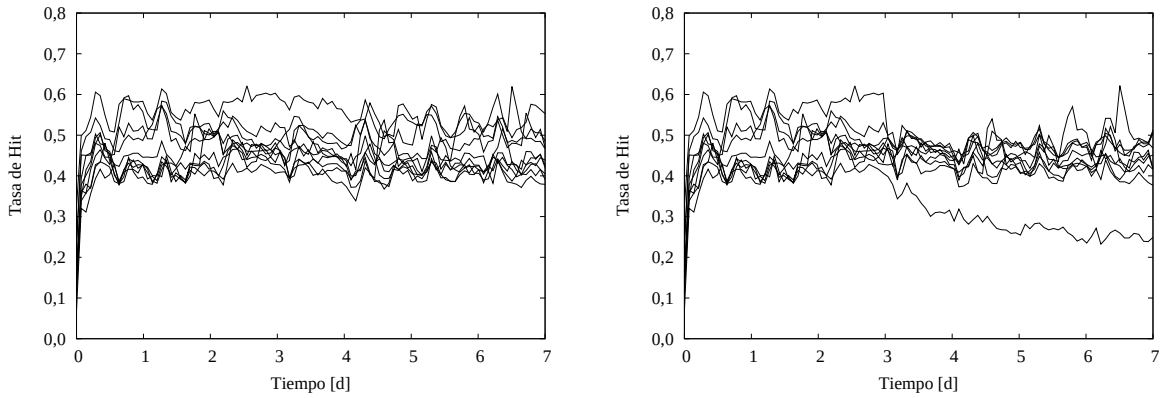


Figura 10: Tasa de *hit* de un Servicio de Cache con 10 particiones y 8 réplicas: (a) sin fallas; (b) 30 % de fallos.

de trabajo de la mayoría de las particiones. Finalmente, en la Figura 9(d) existe una situación más compleja en que 4 de las 10 particiones alcanzan la utilización máxima y ese comportamiento continua hasta el final del experimento.

En lo descrito anteriormente, es posible observar que el fallo de los nodos tiene un alto impacto en el desempeño del servicio. Una arista es la carga de trabajo de los nodos, pero otra arista muy importante es el comportamiento de la tasa de *hit*. Para analizar el impacto, la Figura 10(a) muestra la tasa de *hit* para la misma configuración de la Figura 9(a) (sin fallos). Posteriormente, en la Figura 10(b) se observa el desempeño cuando existe un 30 % de fallas.

Es posible observar cómo esta tasa decae en forma abrupta desde el momento en que suceden las fallas (punto $x = 3$ aproximadamente). Un decaimiento de la tasa de *hit* implica tiempo de respuesta más alto y mayor utilización del servicio de índice (al bajar el número de respuestas pre-computadas en cache). El ambiente cambiante, en términos de nodos que entran y salen del servicio, debe ser analizado y mitigado incorporando sistemas tolerantes a fallas para que, en caso de fallas, el servicio pueda mantener un alto nivel de desempeño, o al menos degradarse pero en forma controlada.

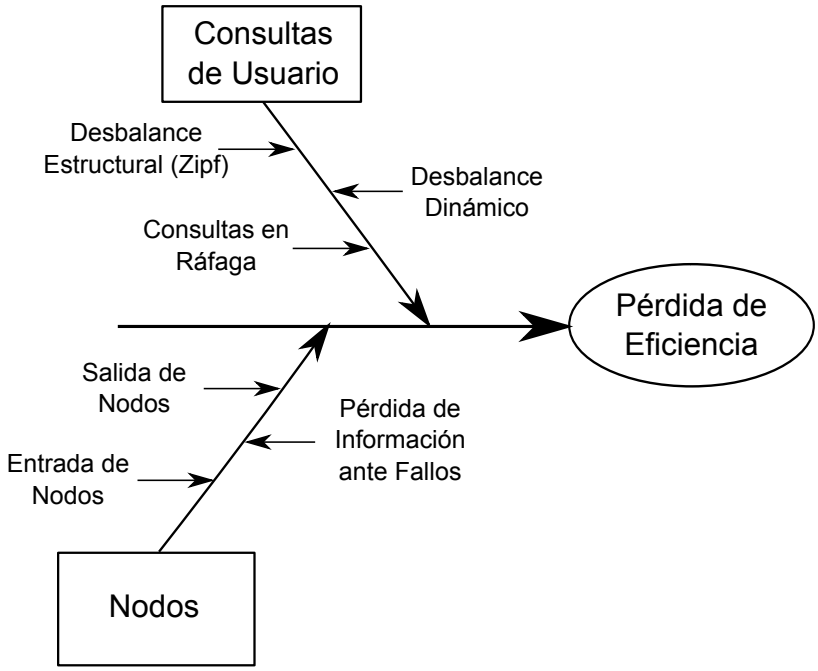


Figura 11: Diagrama causa/efecto.

4.2. Problema y Subproblemas

Según la evidencia dada en la sección anterior, el problema a resolver en este trabajo es la **pérdida de eficiencia en el Servicio de Cache** y se destacaron en forma empírica algunas causas que contribuyen a este problema. Para analizar más detalladamente este problema, se utilizará el diagrama Causa/Efecto (o diagrama de Ishikawa) que ayudará a visualizar gráficamente las causas. La idea de este diagrama es graficar las causas del problema en estudio, en donde cada una de las causas impacta en alguna medida a la generación del problema. En la Figura 11 se muestra a la izquierda las causas y a la derecha el efecto o problema a tratar.

Como se detalla, las principales causas del problema a tratar son:

- **Desbalance Estructural**. Debido al comportamiento Zipf de las consultas de usuario. Este punto es independiente de la componente temporal (siempre las consultas de usuario siguen esta distribución).

- **Desbalance Dinámico**. Debido a la variación en el tiempo del conjunto de

consultas de usuario, especialmente a la generación constante de nuevas consultas, y a la entrada/salida de nodos en forma controlada y no controlada.

- **Aparición de Consultas en Ráfaga**. Debido a la atracción de una gran cantidad de usuarios en tópicos específicos en un intervalo reducido de tiempo.
- **Pérdida de Información Relevante**. Debido a fallas en los nodos y a la inexistencia actual de un esquema tolerante a fallos para servicios de cache.

Cada una de las causas descritas anteriormente, es un subproblema complejo de tratar que debe ser resuelto para atacar el problema general. La idea de este trabajo es analizar cada uno de estos subproblemas y proponer una solución. Cada subproblema será analizado y resuelto en forma independiente. Como se discutirá en el desarrollo de este trabajo, cada solución (a los subproblemas enunciados) por sí misma impacta positivamente en el desempeño del servicio. Además, como se verá en el trabajo, existen estrategias que deben trabajar en *tandem* para el caso particular del Servicio de Cache.

4.3. Consecuencias

La importancia de un Servicio de Cache radica en que, como se mencionó en el capítulo 2, es un componente clave para los motores de búsqueda, y en general para cualquier servicio y/o aplicación en Internet a gran escala. Por ejemplo, Facebook [BFPS11, NO08, NFG+13, Saa], Twitter [Ada10], YouTube [AJZ11], Dynamo de Amazon [DHJ+07a], entre otros, utilizan servicios de cache principalmente para disminuir el tiempo de respuesta y para disminuir el estrés en los servicios que contienen los datos (por ejemplo, servicio de índice en los WSEs). En [Saa] se menciona que Facebook es la aplicación Web que probablemente posee uno de los servicios de cache más grandes en funcionamiento: 800 servidores con memcached [URL, Fit04] (2TB de memoria) para soportar 200 mil peticiones por segundo.

Por otro lado, un Servicio de Cache ahorra el uso de recursos computacionales. En particular, aplicaciones que necesitan alto cómputo (como los WSEs), se benefician del

uso de este tipo de sistemas al reutilizar información/datos que fueron anteriormente calculados. Además, reduce el tiempo de respuesta de aplicaciones en Internet, ya que la idea principal es que al estar en cache un elemento implica una búsqueda a grandes rasgos, que generalmente es más simple que el proceso completo de obtención de la información en el índice invertido. Finalmente, la correcta gestión de este servicio impacta positivamente en el desempeño de éste, aún más, las métricas de desempeño generalmente mejoran con la buena gestión. Es en este punto al cual apunta este trabajo.

La pérdida de eficiencia tiene consecuencias adversas para las métricas de desempeño de un Servicio de Cache. Estas consecuencias están relacionadas en gran medida con lo descrito en el párrafo anterior. Si se analizan las consecuencias de este problema, se tendrá a la vez la importancia de resolverlo. Se enfatiza que cada uno de los puntos detallados a continuación, es un desafío de resolver por sí mismo. Entre las principales consecuencias se destacan:

- Aumento del tiempo de respuesta. Al tener un CS desbalanceado, la varianza en el tiempo de atención de consultas es amplia, generándose un mayor tiempo de respuesta (mayor al tiempo de respuesta óptimo) en los nodos con más carga de trabajo.

- Aumento de la probabilidad de servidores congestionados y colapsados. A medida que un nodo recibe más peticiones para las que fue dimensionado, los servidores se congestionan. Si se siguen asignando peticiones a un servidor congestionado, sin considerar algún mecanismo de balance o reducción de carga, la congestión podría transformarse en colapso de nodos y dejar fuera del servicio a estos nodos. No tomar acciones correctivas en el caso de nodos que salen del servicio podría tornar más grave este problema, haciendo que otros nodos se congestionaran o colapsaran, ya que nodos del mismo servicio deben absorber la carga de los nodos que han caído. En el peor caso, se podría producir un colapso total del servicio.

- Mala utilización de los recursos computacionales. Un servicio desbalanceado impide utilizar los recursos de manera óptima. Además, en algunos casos el número

de nodos de un Servicio de Cache podría reducirse si se tiene un mecanismo que impida las alzas bruscas de carga de trabajo en los nodos.

4.4. Soluciones

Tenemos los subproblemas que gatillan la pérdida de eficiencia en el Servicio de Cache. A continuación, se analizará cada subproblema y se discutirá, sin entrar en mucho detalle, las opciones posibles para solucionarlos. La idea principal es dar una visión inicial de cómo será abordado este trabajo, ya que como se mencionó anteriormente, cada uno de los subproblemas será tratado en forma particular.

- **Desbalance Estructural**. Como todas las consultas no tienen el mismo impacto, el primer paso es identificar cuáles de ellas presentan la posibilidad de congestionar el servicio. Obviamente son las consultas más frecuentes. Para obtenerlas, se puede monitorear durante un período de tiempo todas las consultas que llegan al WSE, ordenarlas y obtener las más frecuentes. Con este método se tendría la frecuencia exacta, pero es inviable por dos razones: (i) debido a la cantidad de memoria y esfuerzo necesario para realizar este proceso; y (ii) debido a que las consultas cambian constantemente (generación de nuevas consultas y sus frecuencias). Como sólo se necesitan las consultas con mayor impacto, se puede utilizar alguna estrategia que permita monitorizar un flujo de consultas y obtener aquellas con mayor probabilidad de ocurrencia, o también algún método que permita obtener la frecuencia aproximada (con algún error) de ellas. Por otro lado, se tiene que las consultas frecuentes podrían congestionar o colapsar algunos nodos. Como el Servicio de Cache está distribuido en múltiples máquinas, entonces el mecanismo clásico para evitar las situaciones mencionadas anteriormente es el balance de carga. Para romper el desbalance estructural, se utilizará balance de carga en múltiples nodos para distribuir las consultas más frecuentes. Con esta acción se reduce el impacto que tendría una consulta frecuente en el Servicio de Cache.

- **Desbalance Dinámico**. Además de los efectos del Desbalance Estructural,

se tiene que las consultas de usuario cambian constantemente en contenido y frecuencia. Estos cambios suceden en el intervalo de tiempo del orden de minutos y/o días, y afectan el balance del Servicio de Cache. Ante estas variaciones dinámicas de carga de trabajo, la estrategia clásica utilizada es el balance de carga dinámico. Esta estrategia debe considerar la aplicación particular, ya que, como se verá más adelante, no todas las estrategias de balance de carga dinámico se pueden usar en este contexto, y por ello se ideará un mecanismo que permite solucionar este problema. La idea de trasfondo es tener cada una de las máquinas del servicio lo más balanceada posible. Además, ante la caída y entrada de nodos al servicio, se debe redistribuir la carga para alcanzar un estado balanceado. La idea final es que ante cualquier evento, el servicio siga estando balanceado y para eso el balance dinámico es fundamental.

- **Aparición de Consultas en Ráfaga**. Esto implica que los usuarios se sienten atraídos por nuevos tópicos que buscan en la Web (nuevas consultas), y que algunos de estos tópicos alcanzan una alta popularidad. Según la evidencia empírica, estas consultas tienen un alto impacto en el Servicio de Cache, ya que implica que un grupo muy alto de instancias de la misma consulta (mismo *string*) aparece en el servicio y podría colapsar muy rápidamente un nodo si no se toman las acciones apropiadas. Esto podría verse como el mismo caso que el Desbalance Estructural, pero la diferencia fundamental radica en el período de aparición. Mientras que la mayoría de las consultas frecuentes se mantienen como tal a través del tiempo, una consulta en ráfaga alcanza las mismas proporciones de una consulta frecuente (en algunos casos mayor), pero en un período muy acotado de tiempo. Luego del punto máximo alcanzado por estas consultas, se vuelven muy poco frecuentes por la pérdida de atracción por parte de los usuarios, incluso llegando en algunos casos a la desaparición de la consulta. Ejemplos clásicos son ocurrencia de fenómenos naturales (terremotos, meteoritos, etc.), y muerte o algún otro evento relacionado con celebridades. Se analizará la forma de atenuar el efecto de este tipo de consultas en el Servicio de Cache. La solución sería nuevamente distribuir la consulta en un grupo más

amplio de nodos, pero para eso se debe primero detectar este tipo de eventos.

- **Pérdida de Información Relevante**. Cada nodo del servicio mantiene información importante en su memoria principal, que en gran parte es dedicada a cache de resultados. Para obtener la respuesta a una consulta, y si su respuesta no está en cache, se debe visitar el servicio de índice y generar en este servicio la respuesta a la consulta. Como se mencionó en el capítulo 2, hacer el procesamiento de la consulta en el servicio de índice es mucho más costoso que en el de cache, por lo cual responder una consulta con un *hit* en cache le quita presión al servicio de índice. Es decir, una entrada en cache es muy valiosa y generarla nuevamente es costoso, por lo que la protección de estas entradas ante caída de nodos es un factor importante a considerar. Esta arista, denominada tolerancia a fallas, es abordada clásicamente mediante la replicación de datos en múltiples nodos. Dado que la distribución de consultas de usuario es sesgada, entonces sólo es importante proteger una fracción pequeña de consultas. La idea principal es que cuando un nodo del Servicio de Cache cae, las peticiones a este nodo puedan ser resueltas a través de la copia redundante en otros nodos del servicio.

Con las soluciones propuestas, que serán detalladas más adelante, se puede reducir la pérdida de eficiencia en el Servicio de Cache, atacando cada una de las causas que lo generan por separado. El análisis de un Servicio de Cache distribuido no ha sido estudiado hasta ahora. Específicamente el balance de carga dinámico y la protección de información relevante ha sido estudiado en otras áreas, pero no se han propuesto soluciones para los servicios de cache en los WSEs. Algunos trabajos tratan la detección temprana de consultas en ráfaga, pero en este trabajo se utilizará la infraestructura propuesta para identificar las consultas en ráfaga y tomar algunas acciones correctivas para mitigar el impacto de ellas.

4.5. Métricas

Las métricas permiten caracterizar la mejora en rendimiento experimentada por la aplicación de las distintas técnicas. Estas métricas principalmente son a nivel global del Servicio de Cache, ya que registrar información por cada instancia de consulta utilizada en la evaluación es complejo y requiere mayores recursos.

Entre las métricas a estudiar se encuentran:

1. **Tiempo de Respuesta**. Esta métrica se mide en milisegundos y representa el tiempo transcurrido entre la llegada de una consulta al FS y la salida de la consulta con su respuesta desde el FS. Esto permite observar cómo impactan las estrategias propuestas en la resolución de las consultas de usuario.

2. **Eficiencia**. La carga de trabajo de los nodos del servicio puede ser representada como la división entre la carga de trabajo promedio y la máxima carga de trabajo (considerando todos los nodos). Esta razón, entre 0 y 1, indica qué tanta diferencia existe entre la carga máxima y la promedio, por lo que una eficiencia cercana a 1 implica un servicio balanceado, mientras que 0 representa el máximo desbalance. El tiempo de respuesta y la eficiencia están relacionados, ya que un servicio mal balanceado presentará peor tiempo de respuesta. Ambas medidas son consideradas para evidenciar la ganancia de las estrategias propuestas desde dos puntos de vista diferentes.

3. **Tasa de *Hit***. La tasa de *hit* es uno de los parámetros clave del Servicio de Cache, ya que un *hit* implica tiempo de respuesta menor y una carga de trabajo menor para el servicio de índice. La idea de evaluar está métrica es determinar el impacto que tiene la implementación de las estrategias propuestas en el Servicio de Cache. Probablemente exista una reducción de la tasa de *hit* con las propuestas, que debería verse compensada por la disminución del desbalance y por ende del tiempo de respuesta.

4. **Congestión**. Si un nodo alcanza el 100 % de utilización, entonces está congestionado. Un servidor congestionado puede caer y salir del servicio, lo cual podría

producir una reacción en cadena en los demás nodos. La idea es determinar los servidores congestionados con la implementación de cada estrategia y evaluar la posible reducción de ellos.

4.6. Simulación

Para llevar a cabo la experimentación se utiliza simulación por eventos discretos. El simulador implementa exactamente las estrategias de cache y el funcionamiento global del motor de búsqueda Web, siendo capaz de predecir con precisión el *throughput* de consultas considerando los diferentes costos involucrados en su procesamiento. La precisión de los resultados de simulación con respecto a una implementación real viene del hecho que (i) la aplicación en estudio es de grano grueso y a que (ii) los costos de las operaciones relevantes pueden ser bien determinados por *benchmarks*.

Al ser de grano grueso, las operaciones involucradas en el procesamiento de consultas están bien definidas. En el contexto de este trabajo, las operaciones relevantes siguen una secuencia similar de ejecución en las consultas y esta secuencia es dada como entrada al simulador. Por ejemplo, una consulta arriba al FS, luego es asignado a un nodo en particular del CS. En este servicio, se debe determinar si una consulta está en cache. Un punto importante a señalar, es que independiente de la simulación, las políticas para cache son implementadas. Si una consulta está en cache, se envía al FS, en caso contrario se envía al IS.

Por otro lado, para obtener los costos de las operaciones relevantes se realizan ejecuciones de un registro de consultas en una implementación real en MPI-C++ sobre hardware real (cluster de computadores). Los resultados de estos *benchmarks* se denominan trazas de ejecución. Por ende, los simuladores utilizados en este trabajo son simuladores de eventos discretos guiados por trazas [MGCGP10b].

Los simuladores han sido implementados in C++ y los objetos concurrentes son simulados usando LibCppSim [Mar04]. La clase de simuladores empleada fue previamente validada en [GPRML12, GPGCR+12, MGCGP10b, GPMCB11]. En [MGCBS13] se encuentran extensiones de la clase de simuladores utilizada.

Distribución de Carga

En este capítulo se propone una solución para mitigar el desbalance estructural. En la sección 5.1 se hace un análisis más profundo del desbalance estructural y las consecuencias al utilizar Consistent Hashing. En la sección 5.2 se definen los subproblemas a resolver y se selecciona una solución desde la literatura para uno de ellos. Posteriormente, en la sección 5.3 se proponen soluciones para los subproblemas restantes, lo que permite construir una solución combinada para un conjunto de subproblemas. Esta solución es adaptiva al tráfico, eficiente en tiempo y espacio. La validación experimental en la sección 5.4 permite ver la mejora en la eficiencia de la estrategia propuesta en comparación con estrategias base. La experimentación se realiza utilizando *logs* de consultas reales y para servicios de cache de distinto tamaño, con el objetivo de medir principalmente balance de carga y tasa de *hit*. Posteriormente, las conclusiones del capítulo se encuentran en la sección 5.5.

5.1. Análisis del Desbalance Estructural

Las consultas de usuario siguen una distribución Zipfiana (capítulo 2). Esto implica que un porcentaje muy reducido de consultas de usuario tienen un alto impacto en el Servicio de Cache (alta frecuencia). Dado el esquema de localización de ítemes con Consistent Hashing (capítulo 3), una consulta con alta frecuencia por unidad de tiempo podría sobrecargar o dejar fuera de servicio un nodo.

Este factor puntual, el que una consulta sea frecuente, es el principal gatillador del desbalance experimentado por el Servicio de Cache. En la Figura 12 se observa un ejemplo del problema de consultas frecuentes en el anillo Consistent Hashing (para facilitar la presentación, el rango del anillo se presenta en forma lineal y los rangos están normalizados). Unos pocos puntos, que representan consultas muy frecuentes,

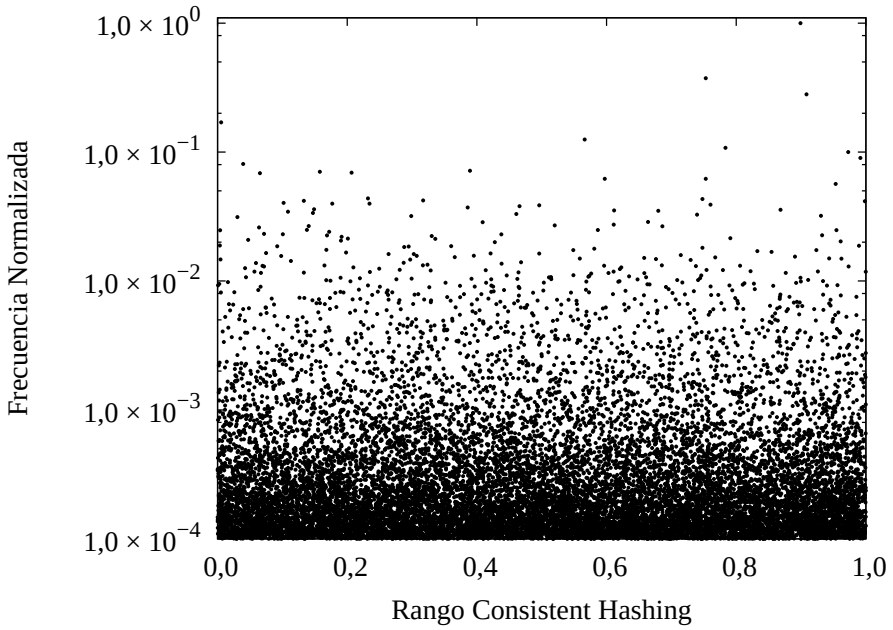


Figura 12: Frecuencia normalizada y su impacto en el anillo Consistent Hashing (Enero, 2012).

tienen gran cantidad de peticiones por unidad de tiempo. Esto es independiente de la función utilizada para calcular el hash de la consulta.

Para hacer explícito el desbalance, analizar la siguiente situación: si se particiona en dos el anillo Consistent Hashing (se parte el anillo en dos mitades cualquiera), es decir se tienen dos nodos en el servicio, la carga de trabajo experimentada por cada nodo será similar. La distribución injusta comienza cuando se tienen más particiones, lo cual es evidenciado en la Figura 7(a) del capítulo 4. En esta figura, se considera la asignación de las consultas de un mes completo en particiones fijas y se aumentan las particiones para ver el efecto en la eficiencia. La eficiencia se mide como el número de peticiones promedio entre todos los nodos dividido por el número máximo de peticiones experimentado por un nodo. Se observa que a más particiones, la asignación de consultas en las particiones se torna más desigual. Según evidencia empírica, no existe influencia entre la distribución de puntos en el anillo (particiones) y la reducción de la eficiencia a medida que se aumenta el número de particiones. Tampoco es un factor importante la función hash utilizada.

Las consultas frecuentes se clasifican en dos grupos: *permanentes* y *ráfagas* [FPSO06]. Las consultas frecuentes *permanentes* (o consultas permanentes) son aquellas que permanecen a través del tiempo con una alta frecuencia. Por otro lado, las frecuentes *ráfagas* (o consultas en ráfaga) tienen apariciones abruptas y en cortos períodos de tiempo según eventos sociales, económicos y naturales. Para mostrar evidencia de estos tipos de consultas, la Tabla 3 muestra la frecuencia porcentual de las 30 consultas más frecuentes en 3 meses consecutivos (cada mes es independiente de los otros). Las consultas permanentes tienen un alto porcentaje de incidencia a través del tiempo, son un conjunto reducido y su ocurrencia sucede a través del mes completo. Por otro lado, las consultas en ráfaga también alcanzan un alto porcentaje de incidencia, pero su ocurrencia es limitada a un intervalo de tiempo en particular. Este capítulo discutirá el tratamiento de las consultas permanentes. Las consultas en ráfaga serán tratadas posteriormente.

Aparte de la frecuencia de las consultas, se tienen límites en los nodos que implementan el Servicio de Cache. Como idea general, se puede mencionar que si se tiene que el número máximo de peticiones por unidad de tiempo que un servidor puede tolerar es x (o que se define en términos de calidad de servicio y tiempo de respuesta), y el número de peticiones por unidad de tiempo experimentado por una consulta frecuente es y, entonces el número de nodos necesarios para soportar esa carga de trabajo es $\lceil y/x \rceil$. Cualquier situación en que sobrepase la tasa x, aumenta la posibilidad de saturación y/o colapso de nodos.

El problema a tratar es la existencia de consultas con una alta tasa de ocurrencia por unidad de tiempo, lo que puede ocasionar congestión y colapso de nodos del Servicio de Cache. Este problema siempre estará presente (muy difícilmente los usuarios generarán consultas con distribución uniforme), por lo que se debe proponer una solución para mitigar el efecto de este tipo de consultas.

La única solución factible, a grandes rasgos, es la atención de estas consultas por múltiples nodos, es decir, distribuir la carga. Una solución poco eficiente es que todas las consultas sean distribuidas, por ejemplo, en todos los nodos. Esta simple solución disminuye la tasa de *hit*, ya que no todas las consultas tienen la misma tasa de aparición ni la misma importancia. Para explicar el problema de la tasa de

Tabla 3: Listado de consultas frecuentes y su frecuencia porcentual para tres meses consecutivos. Por ejemplo, la consulta "facebook" representa el 2,59 % del volumen de consultas en Enero de 2012.

N°	Enero 2012	Febrero 2012	Marzo 2012
1	2,59 facebook	2,70 facebook	2,77 facebook
2	0,97 google	0,93 google	0,89 google
3	0,72 youtube	0,72 youtube	0,73 youtube
4	0,44 craigslist	0,43 craigslist	0,43 craigslist
5	0,32 ebay	0,32 ebay	0,34 facebook com
6	0,28 yahoo	0,26 yahoo	0,32 ebay
7	0,25 yahoo mail	0,26 facebook com	0,24 gmail
8	0,23 gmail	0,24 yahoo mail	0,23 yahoo mail
9	0,21 facebook com	0,23 gmail	0,21 yahoo
10	0,18 you tube	0,21 bobbi kristina	0,18 you tube
11	0,18 facebook sign up	0,19 facebook sign up	0,18 facebook sign up
12	0,17 amazon	0,18 you tube	0,17 www facebook com
13	0,17 google search	0,17 google search	0,16 mega millions
14	0,16 hotmail	0,16 amazon	0,16 google search
15	0,16 khloe kardashian	0,15 hotmail	0,16 amazon
16	0,14 www facebook com	0,15 www facebook com	0,16 hotmail
17	0,11 facebook login	0,15 monica lewinsky	0,12 fb
18	0,11 youtube broadcast yourself	0,14 whitney houston	0,12 mapquest
19	0,10 pornhub	0,12 peyton manning	0,12 facebook login
20	0,10 fb	0,11 facebook login	0,11 youtube broadcast yourself
21	0,10 walmart	0,11 bobby brown	0,11 pornhub
22	0,10 demi moore	0,11 cissy houston	0,11 walmart
23	0,10 youporn	0,11 fb	0,10 sam wopat dies
24	0,10 mapquest	0,11 youtube broadcast yourself	0,10 leah remini fired
25	0,10 google com	0,11 walmart	0,10 xnxx
26	0,10 kim kardashian	0,11 mapquest	0,10 xhamster
27	0,09 yahoo com	0,10 pornhub	0,09 youporn
28	0,09 cher not dead	0,09 jennifer aniston	0,09 google com
29	0,09 bank of america	0,09 google com	0,09 bank of america
30	0,09 xhamster	0,09 youporn	0,09 bobbi kristina brown

aparición, considerar el siguiente ejemplo. Sea NC el número de entradas de cache disponibles en un nodo del servicio, y si se usa un algoritmo LRU, entonces una consulta poco frecuente q necesitará que pasen NC consultas distintas (que tengan una mejor posición con respecto a q) antes de ser desalojada. Una consulta frecuente no tendrá este problema.

Entonces la solución a desarrollar debe ser tal que no afecte la tasa de *hit*: el número de peticiones por unidad de tiempo de la consulta a distribuir debe ser suficiente para mantenerla en cache (que no sea desalojada de memoria), y así mantener su tasa de *hit*. Claramente las consultas frecuentes cumplen esta condición. Para ello debe existir un equilibrio entre las consultas que necesitan múltiples nodos y las que necesitan sólo un nodo.

Para impedir que un gran volumen de una misma consulta sea asignada a una misma máquina (como sería hecho con Consistent Hashing en su forma nativa), esta consulta debe ser distribuida en múltiples máquinas en el momento del ruteo de consultas en el Servicio de Cache. Según la arquitectura descrita en el capítulo 2, el componente encargado de rutear todas las consultas en el Servicio de Cache es el Servicio de Front-End (FS). Entonces, se debe diseñar un mecanismo en el FS tal que, para una consulta frecuente exista un grupo de máquinas y que cada instancia de esa consulta frecuente sea asignada a una máquina en particular de ese grupo. El mecanismo debe considerar un conjunto de consultas frecuentes y sus respectivos grupos de máquinas.

La hipótesis de este capítulo es que es posible reducir los efectos del desbalance estructural a través del diseño de un mecanismo de balance de carga de largo plazo, adaptivo, eficiente en tiempo y espacio, que considere la frecuencia de las consultas y que utilice infraestructura disponible en aplicaciones Web de gran escala.

5.2. Identificación de Subproblemas y Revisión del Estado del Arte

Desde lo descrito anteriormente, varios subproblemas deben ser resueltos: (i) cómo detectar las consultas frecuentes; (ii) cuántas consultas frecuentes considerar; (iii) en cuántos nodos distribuir una consulta frecuente; (iv) cómo seleccionar el grupo de nodos por cada consulta frecuente; y (v) en qué nodo del grupo de nodos distribuir una instancia de una consulta frecuente. Cada una de estas preguntas debe ser respondida para diseñar un esquema rápido, eficiente y compacto de localización de ítemes en el FS. El objetivo final es que este esquema de localización ayude a distribuir la carga de forma más homogénea en los nodos del Servicio de Cache.

A continuación, se realizará la selección de un método para la detección de consultas frecuentes, a través de una revisión del estado del arte. La idea principal es dar solución al subproblema (i) descrito anteriormente, pero siempre considerando aspectos de desempeño. Posteriormente, en la sección 5.3 se propondrán soluciones para los subproblemas restantes.

Para distribuir consultas frecuentes en múltiples máquinas, primero se debe saber cuáles consultas son frecuentes. Una solución básica es acumular la frecuencia de cada consulta asignada en el FS. Luego, cada cierto intervalo de tiempo, se realiza un ordenamiento de las consultas y aquellas con mayor frecuencia son seleccionadas para replicar. Este tipo de soluciones son inviables para aplicaciones con gran cantidad de consultas [MAEA05], ya que requieren conocimiento completo de las frecuencias de todos los elementos. En este trabajo, las consultas distintas son del orden de millones. Para tener una idea de este problema, en la Tabla 4 se muestran algunas características para *logs* de consultas de tres meses completos distintos. El dato más importante es el número de consultas distintas, ya que en función de las consultas distintas es el espacio en memoria utilizado (se necesita tener contadores de frecuencia por cada consulta). Por otro lado, la Figura 13 muestra el número promedio de consultas distintas a medida que se aumenta el intervalo de tiempo de medición. Mantener la frecuencia de todas las consultas que aparecen en un intervalo de tiempo se vuelve inmanejable, desde el punto de vista de la utilización de memoria y eficiencia en el FS. Además, no

Tabla 4: Características de *logs* de consultas para tres meses distintos ($f(q)$ es la frecuencia de q).

Atributo	Enero 2012	Febrero 2012	Marzo 2012
N° consultas	2.032.123.082	1.893.555.146	1.873.446.442
N° consultas distintas	506.756.863	478.271.434	477.878.560
N° consultas con $f(q) = 1$	386.822.371	366.965.677	368.497.559
N° consultas con $f(q) = 2$	62.644.642	58.379.347	57.377.068
N° consultas con $f(q) \geq 3$	57.289.850	52.926.410	52.003.933

tiene sentido almacenar todas las consultas, ya que como éstas siguen distribuciones zipfianas, muy pocas consultas tienen una alta frecuencia.

El primer problema es encontrar un método que detecte las consultas más frecuentes dentro de un flujo de consultas durante un intervalo de tiempo en forma rápida y compacta en términos de espacio.

Sea un *stream* $S = \{q_1, q_2, ..., q_N\}$ de elementos, donde cada elemento $q_i \in A = \{a_1, a_2, ..., a_m\}$ (A se conoce como el alfabeto). El objeto a_i aparece F_i veces en el stream S ($\sum_{i=1}^{m} F_i = N$), y en A se tiene un ordenamiento tal que $F_1 \geq F_2 \geq ... \geq F_m$ (el alfabeto está ordenado por frecuencia decreciente).

A continuación se definen formalmente los siguientes problemas. Dado un alfabeto A, un *elemento frecuente* E_i es un elemento cuya frecuencia F_i, en un stream S de tamaño N, excede una cota definida por el usuario $\lceil \phi N \rceil$, donde $0 \leq \phi \leq 1$. Los *elementos top-k* son los k elementos con mayor frecuencia, es decir, $\{a_1, a_2, ..., a_k\}$ del alfabeto A. Estos problemas son de los más estudiados en la investigación en flujos de datos (*data streams*) [CH08]. Además, es muy popular debido a su sencillez en la definición y a que a veces es utilizado como una subrutina en procedimientos más complejos en flujos de datos. Finalmente, estos dos problemas están relacionados pero son distintos, ya que en el problema de los elementos frecuentes se debe dar el umbral ϕ (el resultado es un conjunto de largo variable) y en el caso de los top-k se debe dar el parámetro k (el resultado es un conjunto de largo k). En algunos casos las frecuencias relativas de los ítemes no son relevantes.

La primera solución es obtener todos los contadores de los elementos del stream en un intervalo de tiempo y obtener los ítemes más frecuentes. Se necesita $\Theta(|A|)$ espacio para resolver el problema exacto sin considerar la distribución de los datos

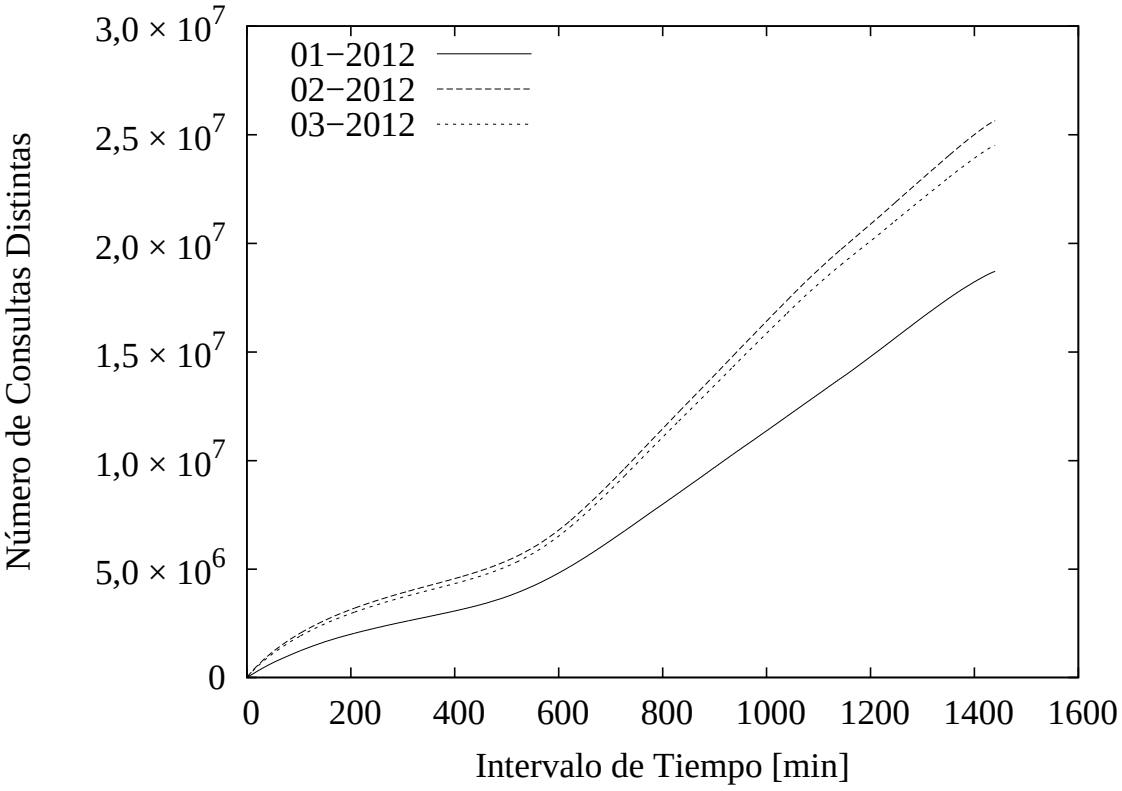


Figura 13: Consultas distintas por minuto para un día.

[HC11, MAA06, MAEA05]. Esto se torna impráctico en aplicaciones de gran escala con alfabetos masivos (consultas distintas en nuestro contexto).

En [CCFC02, CCFC04] se menciona que un algoritmo el cual obtiene una muestra uniforme aleatoria de los datos (algoritmo SAMPLING), tiene un requerimiento de espacio que depende de la distribución de la frecuencia de los ítemes del stream. Por ejemplo, para distribuciones Zipfianas con parámetro $z > 1$, el requerimiento de espacio de SAMPLING para obtener los k elementos más frecuentes con una probabilidad $1 - \delta$ es $k(\log \frac{k}{\delta})^{\frac{1}{z}}$ (Tabla 1 en [CCFC02]). El muestreo podría afectar tanto la proporción de consultas frecuentes como la frecuencia de éstas. Además, impone preguntas con respecto a cómo hacer el muestreo. Por ejemplo, ¿un muestro uniforme aleatorio le dará la importancia necesaria a las consultas en ráfaga? Esto es un tema relevante, ya que las consultas en ráfaga tienen una alta frecuencia en un tiempo reducido, por lo que un muestreo no permitiría captar su aparición. En el capítulo 4 se explicó la importancia de no pasar por alto este tipo de consultas. Es por esta razón, que se debe utilizar algún método que considere de alguna manera todas las consultas del *stream*.

Siguiendo el trabajo de [CCFC04], se definen dos nociones de aproximación al problema de los ítemes más frecuentes:

1. $CandidateTop(S, k, l)$
   - *Entrada*: un stream S, dos enteros k y l.
   - *Salida*: una lista de l elementos de S tal que los k elementos más frecuentes aparecen en la lista (sin garantizar el *ranking* de los $l - k$ elementos restantes).
2. $ApproxTop(S, k, \epsilon)$
   - *Entrada*: un stream S, un entero k y un real ϵ.
   - *Salida*: una lista de k elementos de S tal que todo elemento E_i en la lista tiene frecuencia $F_i > (1 - \epsilon)F_k$.

Para distribuciones genéricas, *CandidateTop* es díficil de resolver. Dada esa complejidad, se define la variante *ApproxTop*, que es una aproximación más práctica al problema de los top-k. En *ApproxTop* se pregunta por una lista de k elementos tal que cada elemento E_i en la lista tiene frecuencia $F_i > (1 - \epsilon)F_k$. Es decir, cada elemento E_i retornado dentro de los top-k tiene mayor frecuencia que el k-ésimo elemento del alfabeto (recordar que en el alfabeto A está ordenado por frecuencia decreciente). Obviamente el número real ϵ da cierto margen para las frecuencias de los top-k.

Este problema se encuentra presente en muchas aplicaciones Web actuales, por ejemplo Twitter (los ítemes frecuentes son los *trending topics* [LPN+11]) y motores de búsqueda Web (los ítemes frecuentes son las consultas populares [CH08, CCFC02, CCFC04]). Además, como se menciona en [CM05], este tipo de consultas tienen impacto en los sistemas de cache, en el balance de carga y en otros puntos del desempeño de sistemas.

Las soluciones para este problema asumen que el flujo de datos es suficientemente grande como para que las soluciones altamente intensivas en memoria, técnicas basadas en ordenamiento o mantener contadores por cada elemento del flujo no son factibles de realizar. Además, dado el flujo de datos, sólo una pasada sobre los datos es posible. Como este enorme flujo de datos es generado a altas velocidades, es importante utilizar algoritmos que procesen cada actualización rápidamente y ojalá sin bloqueos de instrucciones ni de estructuras de datos de por medio. Finalmente, las estructuras compactas en espacio tienen un mejor desempeño en términos de cache, lo que ayuda a aumentar el *throughput* (número de operaciones por unidad de tiempo).

El *survey* de Cormode *et al.* [CH10] realiza una comparación de los métodos existentes para el problema de los ítemes más frecuentes. La comparación se realiza bajo condiciones de experimentación uniforme. Además, mencionan que los métodos pueden ser implementados en pocos kilobytes de memoria y soportar del orden de millones de operaciones por segundo sobre hardware moderno.

Los autores realizan una división de los métodos en tres clases principales [CH10]:

- **Algoritmos basados en Contadores** (*Counter-based Algorithms*). Estos algoritmos mantienen contadores para una cantidad fija de elementos del flujo de datos. Sólo un número limitado de elementos es monitoreado. Si arriba un

ítem que está siendo monitoreado, entonces su contador se incrementa. Si no está siendo monitoreado, entonces el algoritmo decide si descartarlo o re-asignar un contador existente a este ítem.

- **Algoritmos de Cuantiles** (*Quantile Algorithms*). El ϕ-cuantil ($0 \leq \phi \leq 1$) de un stream S es un elemento x tal que $\phi|S|$ elementos de S son menores o iguales que x y los restantes $(1-\phi)|S|$ son más grandes que x. Los autores muestran que el problema de encontrar quantiles (aproximados) permite encontrar los ítemes frecuentes.

- **Algoritmos de *Sketch*** (*Sketch Algorithms*). Estos algoritmos utilizan un *sketch* del flujo de datos. Para esto, se utilizan técnicas como hashing para mapear los ítemes a un conjunto reducido de contadores. Estos métodos mantienen frecuencias aproximadas para todos los ítemes del flujo de datos.

Para justificar la elección de qué clase de algoritmos a utilizar, se revisarán los trabajos que estudian estos métodos. El trabajo de Cormode *et al.* [CH10], que es una extensión de [CH08, CH09], estudia en forma separada las tres clases de métodos descritos previamente. Una de las principales conclusiones del trabajo, es que los algoritmos de cuantiles no pueden competir con los algoritmos basados en contadores, ya que los primeros tienen estructuras más grandes, son más lentos en la actualización, y no estiman las frecuencias con precisión. Por otro lado, algunos métodos basados en *sketch* utilizan poco espacio y tienen un buen desempeño en la actualización, pero con una buena precisión sólo para distribuciones altamente sesgadas. Para los métodos basados en *sketch* existen dos parámetros, que se deben configurar correctamente para obtener un buen desempeño y que esta configuración es dependiente del caso de estudio (datos). Finalmente, los algoritmos basados en contadores presentan claras ventajas sobre los basados en *sketch* bajo las mismas condiciones de comparación.

En el trabajo de Minekar *et al.* [MP09] se realiza una evaluación experimental de algoritmos basados en contadores y en *sketch*. En la discusión relacionada a la comparación de estos métodos, los autores mencionan en que si la aplicación es estrictamente limitada a la búsqueda de los ítemes frecuentes, los algoritmos basados en contadores tienen un desempeño superior y son más fáciles de implementar que

los basados en *sketch* (los métodos basados en *sketch* proveen de más información estadística que podría ser importante en otras aplicaciones).

Como justificación de este trabajo para la elección de la clase de algoritmo, está el hecho de que no se necesita la frecuencia de todos los elementos del flujo de datos, sino que de un subconjunto de ellos: los más frecuentes. Existe un antecedente adicional que debe ser considerado: distribuciones Zipf. Esta situación hace que el espacio necesario para mantener contadores y los errores presentes en las estimaciones de las frecuencias obtenidas sean muy ajustadas a la realidad [MAA06]. Por la evidencia en trabajos del área y las razones descritas anteriormente, los algoritmos a utilizar en este trabajo son del tipo basado en contadores.

Dentro de los algoritmos basados en contadores, los más importantes son *Frequent* [GDD+03, KSP03, MG82], *Lossy Counting* [MM02a] y *Space-Saving* [MAA06, MAEA05].

Tanto en el trabajo de Minekar *et al.* [MP09] como en el de Cormode *et al.* [CH10], se indica que el algoritmo basado en contadores con mejor desempeño es *Space-Saving*, seguido por *Lossy Counting*. Evaluaciones adicionales de *Space-Saving* y otros algoritmos son encontradas en [MAA06]. A continuación se hará un resumen de los principales resultados entregados en los trabajos anteriores con respecto a estos algoritmos.

Los principales resultados de [CH10] son:

1. Para distribuciones zipfianas sintéticas con parámetro z entre 0,8 y 1,0 (nuestro contexto), *Space-Saving* muestra un desempeño similar a *Lossy Couting* en términos de tiempo de ejecución. Para datos reales (basados en tráfico de red), *Space-Saving* tiene un tiempo de ejecución aproximadamente 4 veces mayor que *Lossy Counting*.

2. En términos de espacio, *Space-Saving* con *heaps* tiene el mismo desempeño que *Lossy Couting* para distribuciones zipfianas sintéticas.

3. El *Recall*, tanto para *Space-Saving* como *Lossy Counting* es 100 % para distribuciones zipfianas sintéticas. Por otro lado, la *Precision* es un 100 % para

Space-Saving, mientras que es en promedio un 60 % para *Lossy Counting*. Para datos reales, *Space-Saving* tiene un 100 % de *Precision*, mientras que *Lossy Counting* tiene en promedio un 40 %.

4. Tanto *Lossy Counting* como *Space-Saving* tienen un error relativo promedio de las frecuencias reportadas de 0,0 % para distribuciones zipfianas sintéticas (con varianza igual a cero). Con datos reales, *Space-Saving* tiene un error relativo promedio de 0,2 %, mientras que *Lossy Counting* reporta un 5,5 %.

5. De dos implementaciones posibles, con *heaps* y listas enlazadas, la versión de *Space-Saving* con mejor desempeño, en términos de tiempo, es la con *heaps*.

6. Si el stream es *insert-only*, como en este trabajo, una clara conclusión es que *Space-Saving* es superior a los demás algoritmos.

Mientras que los principales resultados de [MP09] son:

1. Para distribuciones zipfianas sintéticas, la memoria utiliza por *Lossy Counting* es aproximadamente dos veces superior a la de *Space-Saving*.

2. Tanto *Lossy Counting* como *Space-Saving* tiene 100 % de *Recall* y 100 % de *Precision* para distribuciones zipfianas sintéticas. Lo mismo se repite para datos reales (excepto unas pocas divergencias perjudicando a *Lossy Counting*).

3. *Space-Saving* muestra un comportamiento estable ante la variación de parámetros en los experimentos.

4. *Space-Saving* tiene un *Precision* y *Recall* cercano al 100 % ante la variación de la memoria disponible para operar, característica no observada con *Lossy Counting*.

Space-Saving [MAA06, MAEA05] monitorea un conjunto de m elementos. Se inicializa con la frecuencia de los primeros m elementos distintos, y cada vez que arriba un nuevo elemento monitoreado se incrementa su contador. Si el nuevo elemento no está monitoreado, se reemplaza el ítem con menor contador *min* con el nuevo elemento y se incrementa su contador (a $min + 1$). Por cada elemento monitoreado e_i se asocia

Algoritmo 1 ***Space-Saving*** [MAA06, MAEA05].

Input: m contadores (con $m > 0$), *stream* S.
Output:

```
1: for cada elemento, e, en S do
2:     if e es monitoreado then
3:         Sea count_i el contador de e
4:         Incrementar-Contador( count_i )
5:     else
6:         Sea e_m el elemento con menos hits, min
7:         Reemplazar e_m con e
8:         Incrementar-Contador( count_m )
9:         Asignar ε_m el valor min
10:    end if
11: end for
```

ϵ_i que corresponde a la sobre-estimación máxima del ítem cuando fue insertado (esta sobre-estimación es inicializada con el valor del contador del elemento que fue desalojado, con contador *min*). Este valor de sobre-estimación toma importancia para dar algunas garantías de la salida del algoritmo. En el Algoritmo 1 se encuentra enunciado *Space-Saving* (*Incrementar-Contador* incrementa el contador dado como parámetro y realiza modificaciones a una estructura de datos utilizada, llamada *Stream-Summary*, para mantener los elementos ordenados por su frecuencia).

Desde el punto de vista teórico, en [MAA06] existen teoremas que indican los límites en términos de espacio necesario para la operación de *Space-Saving*.

Teorema 1 (4.5 de [MAA06]). Independiente de la distribución de los datos, para resolver el problema $ApproxTop(S, k, \epsilon)$, *Space-Saving* usa sólo mín$(|A|, \left\lceil \frac{N}{\epsilon F_k} \right\rceil)$ contadores ($|A|$ es el tamaño del alfabeto, N el número de elementos del *stream*, ϵ el error a tolerar y F_k la frecuencia del k-ésimo elemento del alfabeto ordenado decrecientemente). Se garantiza que cualquier elemento con frecuencia mayor que $(1 - \epsilon)F_k$ sea monitoreado.

Teorema 2 (4.6 de [MAA06]). Asumiendo una distribución Zipfiana sin ruido con parámetro $\alpha > 1$ (altamente sesgadas), para calcular los top-k exactos, el número de

contadores usados por *Space-Saving* está limitado por:

$$\text{mín}\left(|A|, O\left(\left(\frac{k}{\alpha}\right)^{\frac{1}{\alpha}} k\right)\right).$$

Cuando $\alpha = 1$, la complejidad de espacio es:

$$\text{mín}(|A|, O(k^2 \ln(|A|))).$$

Esto es independientemente de la permutación del stream. Además, el orden entre los elementos top-k es preservado.

Pero aún más importante es que, para el caso de distribuciones Zipfianas ligeramente sesgadas con $\alpha < 1$, como en nuestro contexto, los autores establecen para *Space-Saving* una cota de espacio para la obtención de top-k de:

$$O\left(\frac{k \ln N}{\epsilon}\right).$$

Los parámetros de esta cota corresponden a los parámetros de $ApproxTop(S, k, \epsilon)$, con lo cual se tiene una solución y el espacio necesario para el problema de los ítemes top-k (con cierto margen de error).

En los últimos años, existen trabajos como [CCT12, HC10], que abordan las problemáticas tratadas y que, según sus autores, mejoran los algoritmos descritos anteriormente (en forma marginal). La elección de *Space-Saving* en este trabajo está dada por los fundamentos teóricos descritos y por la amplia evidencia empírica existente también analizada.

5.3. Solución de Subproblemas

Con la selección del algoritmo *Space-Saving* se resuelve el subproblema (i) cómo detectar las consultas frecuentes. En esta sección, se propondrán soluciones para los subproblemas: (ii) cuántas consultas frecuentes considerar; (iii) en cuántos nodos distribuir una consulta frecuente; (iv) cómo seleccionar el grupo de nodos por cada

consulta frecuente; y (v) en qué nodo del grupo de nodos distribuir una instancia de una consulta frecuente. Cada una de estas preguntas deben ser respondidas para diseñar un esquema rápido, eficiente y compacto de localización de ítemes en el FS. El objetivo final es que este esquema de localización ayude a distribuir la carga de forma más homogénea en los nodos del Servicio de Cache.

5.3.1. Número de Consultas Frecuentes

En la Figura 7(b) del capítulo 4, se expone la eficiencia a medida que se descartan las N consultas más frecuentes. Se distinguen dos rangos importantes: 1-100, y 100-500. Estos rangos son independientes del *log* de consultas utilizado. Se observa que al descartar desde 1 a 100 consultas más frecuentes, se obtiene una mejora drástica en el desempeño del servicio. Luego de las 100 consultas más frecuentes, en el segundo rango, se obtiene una mejora pero marginal comparado con el rango anterior. Esto está en plena concordancia con la Tabla 3 que indica el porcentaje de impacto de las 30 consultas más frecuentes para los mismos *logs* utilizados.

Lo anterior implica que la mayor ganancia en replicar consultas frecuentes se obtiene en el primer rango, es decir, las consultas que deben ser consideradas con mayor prioridad son las primeras 100 más frecuentes. Con respecto a la monitorización, este factor es importante ya que ayuda a la configuración correcta de los parámetros del algoritmo *Space-Saving*. Entonces, la cantidad de consultas top-k necesaria nunca será mayor a una centena ($k < 100$).

Por otro lado, se debe definir el error ϵ en la determinación de los top-k. Existe un compromiso entre el error y la cantidad de contadores necesarios (menor error implica más contadores), pero no se debe perder de vista la idea principal: obtener los ítemes más frecuentes a un costo razonable de tiempo y espacio. Es decir, es posible sacrificar error en pos de reducir la cantidad de contadores. Es por esto que sólo se analizará la situación para $\epsilon = 0,1$ y $\epsilon = 0,05$, ya que errores más altos podrían perjudicar la estimación de las frecuencias y errores más bajos hacen que una alta cantidad de contadores sean necesarios.

El tercer parámetro es N, el número de elementos en el *stream*. Para estimar

Tabla 5: Contadores necesarios para top-k y $\epsilon = 0,1$.

k	N			
	50.000	10.0000	150.000	200.000
10	1.082	1.151	1.192	1.221
20	2.164	2.303	2.384	2.441
30	3.246	3.454	3.576	3.662
40	4.328	4.605	4.767	4.882
50	5.410	5.756	5.959	6.103
60	6.492	6.908	7.151	7.324
70	7.574	8.059	8.343	8.544
80	8.656	9.210	9.535	9.765
90	9.738	10.362	10.727	10.985
100	10.820	11.513	11.918	12.206

Tabla 6: Contadores necesarios para top-k y $\epsilon = 0,05$.

k	N			
	50.000	100.000	150.000	200.000
10	2.164	2.303	2.384	2.441
20	4.328	4.605	4.767	4.882
30	6.492	6.908	7.151	7.324
40	8.656	9.210	9.535	9.765
50	10.820	11.513	11.918	12.206
60	12.984	13.816	14.302	14.647
70	15.148	16.118	16.686	17.089
80	17.312	18.421	19.069	19.530
90	19.476	20.723	21.453	21.971
100	21.640	23.026	23.837	24.412

los valores correctos, se debe recurrir a los datos de la Tabla 4, que indican que en promedio se tienen aproximadamente 44.600 consultas por minuto. Entonces, se evaluará N en el orden de decenas de miles de elementos.

En la Tabla 5 y Tabla 6 se exponen los contadores con $\epsilon = 0,1$ y $\epsilon = 0,05$, respectivamente. Se varían los valores de k y N desde según lo descrito anteriormente.

Lo descrito da indicios que para obtener los top-k, con $k < 100$, con un error ϵ de entre 0,1 y 0,05 y para intervalos de tiempo que contienen del orden de decenas o centenas de miles de elementos, es necesario utilizar del orden de entre 10 mil y 20 mil contadores. Si se consideran contadores de 4 bytes y consultas de 256 bytes máximo, serían necesarios unos pocos Mbytes de memoria. Según los experimentos realizados, la memoria utilizada no pasa de 20 a 40 Mbytes.

5.3.2. Conjunto de Nodos

En esta subsección se analizarán los puntos (iii) en cuántos nodos distribuir una consulta frecuente (sin sobrecargar algún nodo); y (iv) cuáles máquinas son seleccionadas para distribuir la consulta frecuente.

Como fue mencionado en la sección 5.1, para que no se generen las condiciones de sobrecarga o colapso en un nodo, la cantidad de peticiones por unidad de tiempo no deben pasar un umbral. Este umbral es definido en términos de tiempo de respuesta y capacidad del nodo. Entonces la definición de en cuántos nodos se distribuye una consulta (punto (iii)), está determinado por la cantidad de veces que el número de peticiones por unidad de tiempo de una consulta supera este umbral.

Cada consulta frecuente tendrá asignada una cantidad de nodos dentro de $[1, NCS]$, es decir, como mínimo tendrá una máquina asignada (en teoría) y como máximo todo el servicio (NCS). Según se ha analizado, una consulta frecuente es poco probable que tenga una máquina del servicio asignada. Por otro lado, una consulta que tenga todo el servicio disponible también es un caso extremo. El número de peticiones de una consulta puede crecer indefinidamente, lo que se traduciría en que todos los nodos del Servicio de Cache recibirían a esa consulta. Como no existen más máquinas disponibles, cada una de ellas podría recibir más de sus peticiones máximas por unidad de tiempo. Este es un caso especial que sucede muy pocas veces y en un espacio de tiempo muy reducido, por lo cual no representa un problema mayor. Sin embargo, en el dimensionamiento de las capacidades de cada máquina, se debe considerar una holgura para este tipo de situaciones.

Para la decisión de en cuántos nodos replicar una consulta, se utiliza un mecanismo basado en el trabajo de Rosas *et al.* [RHMGC13]. Este método consiste en incrementar dinámicamente el número de responsables por una consulta, en base a la capacidad del nodo y a la frecuencia experimentada de una consulta. Así, es posible establecer el número de nodos necesarios para distribuir una consulta, contribuyendo a disminuir la posibilidad de congestión.

Definir una capacidad por nodo es complejo, ya que impone preguntas como: ¿cómo se caracterizan los nodos heterogéneos?, ¿cómo se incluye el tráfico variable?, ¿cómo incluir el impacto de las consultas en ráfaga?, y ¿la capacidad es fija o variable?

Además de esto, se tienen miles de peticiones por segundo y caída/agregación de nodos en el Servicio de Cache (esto último es una debilidad en un esquema de capacidad fija por nodo).

Si se sabe exactamente el número de peticiones por nodo y/o se tiene una distribución de consultas uniforme, la definición de una capacidad se torna más sencillo. Esto no es aplicable en el contexto de este trabajo, debido a la alta variación del volumen y composición de las consultas, y también debido a alzas abruptas en períodos de tiempo acotados.

Un punto importante es que la capacidad de un nodo es aplicada a todas las consultas que arriban a ese nodo. Es decir, si se define una capacidad C en un nodo y llegan más de C peticiones en un intervalo de tiempo, entonces se está sobrepasando la capacidad del nodo, independiente de la composición de las consultas, y se deben distribuir todas las peticiones de ese nodo en otro conjunto de nodos. Esto no es aplicable al contexto de los motores de búsqueda, ya que se tienen distribuciones zipfianas de consultas. Claramente este esquema es injusto en el caso de distribuciones zipfianas, ya que se trata de igual forma a todas las consultas. Peor aún, distribuir el conjunto total de peticiones en múltiples nodos disminuye la tasa de *hit*. Esta no es la solución correcta.

Se opta por un esquema similar al de Rosas *et al.* [RHMGC13], pero con una definición dinámica de capacidad. Como las consultas a distribuir en múltiples nodos son las top-k, entonces la capacidad es definida dinámicamente considerando las frecuencias aproximadas de este pequeño grupo de consultas. Entonces la capacidad dinámica C_d está dada por:

$$C_d = \frac{\sum_{q_i \in top-k} f(q_i)}{P},$$

con P el número de nodos activos del Servicio de Cache. Así, C_d representa el número de peticiones promedio que debería recibir cada nodo del servicio (considerando sólo las consultas top-k con más impacto). Esta capacidad es genérica y aplicada a cada consulta top-k para determinar el número de nodos a los cuales debe ser asignada. Cabe señalar que ya se ha descrito como obtener la frecuencia aproximada $f(q)$ de

Algoritmo 2 Replicación. Algoritmo propuesto para obtener el número de réplicas necesarias. Se ejecuta cada Δt unidades de tiempo.

Input: top-k: k consultas más frecuentes (*Space-Saving*); C_d: capacidad.
Output: *Rep* $\langle q, r(q)\rangle$: cantidad de réplicas por consulta.

```
1: for cada consulta q ∈ top-k do
2:     if q.frec > C_d then
3:         r(q) ← ⌈q.frec/C_d⌉
4:         Insertar ⟨q, r(q)⟩ en Rep
5:     end if
6: end for
```

una consulta q en un intervalo de tiempo. Cuando $f(q)$ excede C_d, se dice que el nodo p_i responsable de q está saturado por la consulta q. Para obtener el número de réplicas que necesita una consulta $r(q)$, se considera la siguiente expresión:

$$r(q) = \left\lceil \frac{f(q)}{C_d} \right\rceil$$

Este análisis se realiza a nivel de consulta para distribuir u homogeneizar aquellas con alto impacto. Esta homogeneización debe ser de acuerdo a la frecuencia experimentada por la consulta. El Algoritmo 2 muestra los pasos necesarios para fijar el número de réplicas por consulta.

Del paso anterior, se obtiene una serie de pares $\langle q, r(q)\rangle$, donde q es el *string* de una consulta frecuente y $r(q)$ es el número de nodos en que debe ser distribuida esa consulta. A $r(q)$ lo denominamos *nivel o grado de replicación* de una consulta q. Se quiere recalcar que q representa a todas las instancias con el mismo *string*, entonces ante el arribo de una instancia de q, se debe seleccionar una de las $r(q)$ máquinas asignadas a q para acceder al Servicio de Cache.

Una vez conocido el nivel de replicación de una consulta, el siguiente paso es seleccionar las máquinas del Servicio de Cache asignadas a cada consulta frecuente. A continuación, se analizará cómo realizar esta selección y las posibles estrategias.

El número mínimo de nodos que puede necesitar una consulta frecuente varía desde un hasta *NCS* nodos (el servicio completo). Dado que el comportamiento del usuario es variable, el grado de replicación de una consulta frecuente podría variar,

lo que implica que el conjunto de $r(q)$ nodos a atender la consulta podría incrementar/decrecer en función del impacto de la consulta. Más que un conjunto, por cada consulta q se establece una secuencia estática de NCS nodos *a priori*. A esta secuencia total se le denomina $sec(q)$ y puede ser establecida por mecanismos que serán tratados más adelante. Entonces, cuando una consulta q se torna frecuente y debe ser atendida por N nodos, esta consulta q es distribuida en los primeros N nodos de la secuencia $sec(q)$. A la subsecuencia de $r(q)$ nodos de $sec(q)$ se le denomina $real(q)$. Así, si $r(q)$ crece (o decrece) en M nodos, entonces se considerarán los $N + M$ primeros nodos de $sec(q)$ ($N - M$ si decrece). De este modo, respetando el orden de la secuencia, el mínimo de cambios se produce para la secuencia de nodos $real(q)$.

En la Figura 14, se muestra un ejemplo de una secuencia de nodos $sec(q)$ para una consulta q. Cada nodo es etiquetado con su posición en la secuencia. Por ejemplo, los nodos *P6* y *P2* son el primer y segundo nodo de la secuencia respectivamente. Se debe considerar que los nodos responsables de las consultas, en el ejemplo *P0* es el nodo responsable de q, son considerados como el primer elemento fijo e inamovible de la secuencia (a menos que falle). Siguiendo en el mismo ejemplo, si q debe ser atendida por $r(q) = 3$ nodos adicionales, entonces $real(q) = \langle P0, P6, P2, P7\rangle$ (en ese mismo orden). Si se aumenta a $N = 5$, entonces la secuencia real de nodos cambia a $real(q) = \langle P0, P6, P2, P7, P1, P5\rangle$. Por otro lado, si se reduce el número de nodos que debe atender q a $N = 2$, entonces se tiene que $real(q) = \langle P0, P6, P2\rangle$. Como se observa, el cambio en el conjunto de nodos a medida que $r(q)$ crece o disminuye es mínimo, lo que tiene impacto directo en la tasa de *hit* (similar al problema de cambio de vistas de Consistent Hashing). Finalmente, suponer que se tiene $r(q) = 5$, entonces $real(q) = \langle P0, P6, P2, P7, P1, P5\rangle$, y sucede una falla en el nodo *P7*. Como $r(q) = 5$, se debe agregar un nodo adicional, que es el siguiente de la secuencia (no considerando *P7*), por lo que se tiene que $real(q) = \langle P0, P6, P2, P1, P5, P3\rangle$.

En este problema, lo que se torna crítico es que una consulta sature un nodo, por lo que aumentar rápidamente su grado de replicación es crítico. No así el proceso de disminuir el grado de replicación de una consulta, ya que esto no afecta de manera importante el desempeño del Servicio de Cache. Para plasmar este punto, se utiliza una estructura jerárquica en función de los niveles de replicación y un esquema de

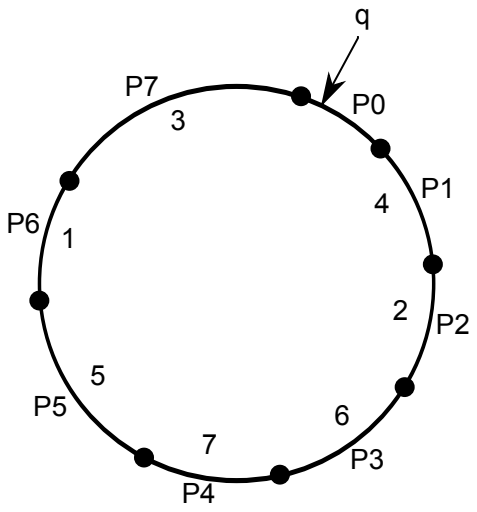


Figura 14: Servicio de Cache de 8 nodos y una secuencia de nodos para la consulta q.

envejecimiento (o *aging*). La idea es que el aumento del grado de replicación sea reflejado en forma instantánea, pero la disminución sea gradual y pausada en los niveles de la estructura. Cada vez que una consulta necesite más nodos, se asigna al nivel correspondiente. Por otro lado, cuando una consulta disminuye su nivel de replicación, a esa consulta se le da la oportunidad de permanecer A unidades de tiempo en el nivel actual n_i. Si transcurren A unidades de tiempo y la consulta se encuentra en el nivel n_j, con $n_j < n_i$, entonces baja al nivel n_{i-1}. Este proceso de envejecimiento se realiza hasta que la consulta alcance el nivel correspondiente a su grado de replicación necesario, o hasta que sea desalojada de la estructura (no necesitaría ser replicada). Con esto se previene la variación que puedan experimentar las consultas en términos de niveles de replicación.

Como se mencionó anteriormente, por cada consulta frecuente se debe seleccionar la secuencia de nodos. Las estrategias factibles para esta selección son las siguientes:

- $r(q)$ **nodos vecinos**: la idea más sencilla es que dada una consulta q con su respectiva asignación dentro del anillo Consistent Hashing, seleccionar los N vecinos consecutivos siguiendo el sentido de las agujas del reloj. En la Figura 15(a) se observa un ejemplo de la determinación de tres nodos para la consulta q con este método. En este caso, además de la máquina responsable *P0* de la

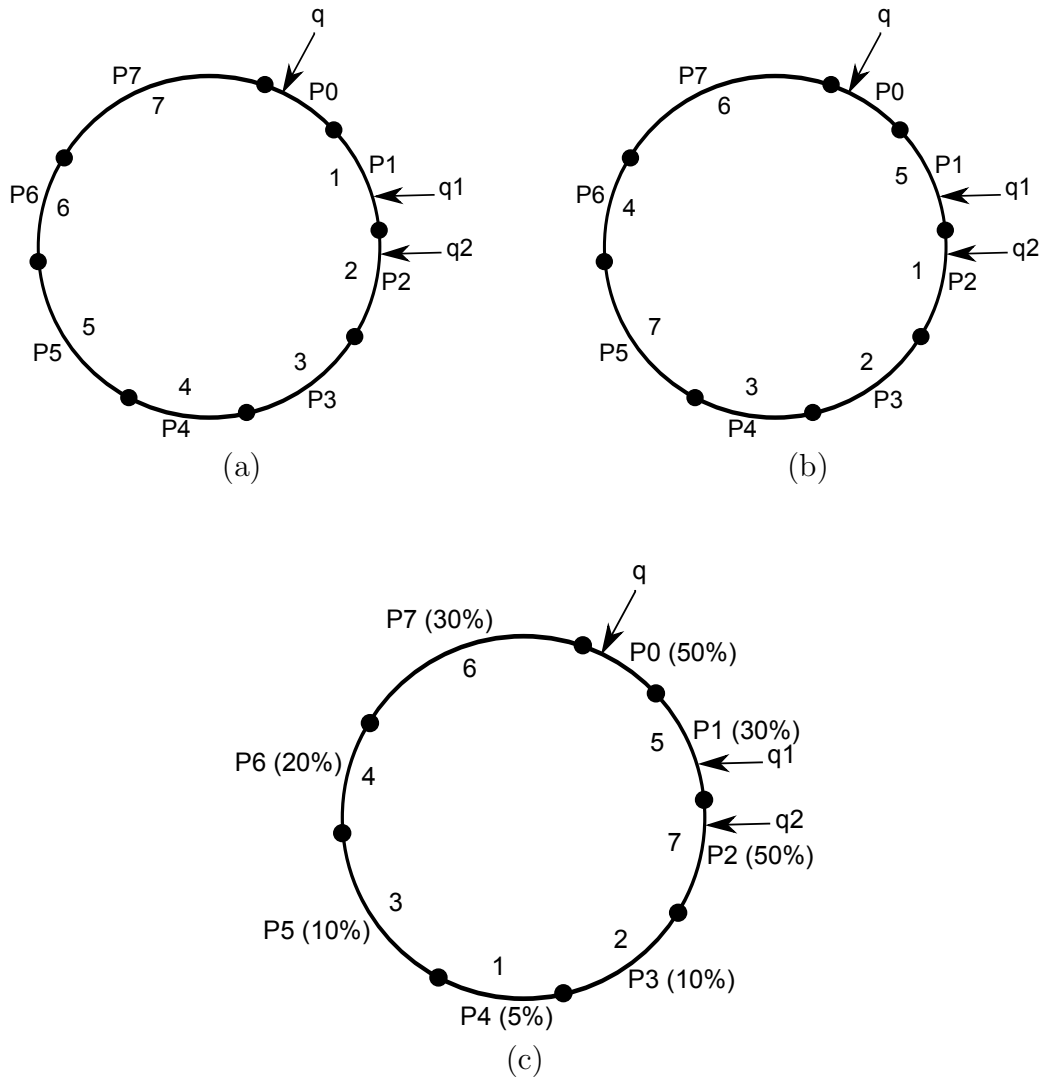


Figura 15: Métodos para obtención de secuencia de nodos: (a) vecinos; (b) azar; y (c) balance.

consulta, otras opciones de enviar la consulta son $P1, P2, P3$ ($r(q) = 3$). Entonces, ante el arribo de una instancia de la consulta q, esta instancia debe ser asignada a alguno de los nodos $P0, P1, P2, P3$ (la selección de qué nodo se detallará posteriormente). Sean NCS nodos en el Servicio de Cache y N el número de nodos en que debe ser distribuida una consulta, entonces con este método existen sólo NCS configuraciones posibles en el anillo. Es decir, existen NCS posibilidades de situar N nodos consecutivos en el anillo Consistent Hashing. Este método es simple, pero lo detallado anteriormente es una desventaja. Esta desventaja se da principalmente cuando dos consultas con alto impacto están muy "cerca" en el anillo. Por ejemplo, suponer que *q1* y *q2* son dos consultas frecuentes cuyas posiciones en el anillo son las indicadas en la Figura 15(a). Sean $r(q1) = 2$ y $r(q2) = 3$ el número de nodos en los cuales *q1* y *q2* se deben distribuir respectivamente. El problema es que el conjunto de nodos de $real(q1)$ es un subconjunto de los nodos de $real(q2)$. Esto aumentaría las posibilidades de asignar más carga de trabajo a esos conjuntos de nodos.

- $r(q)$ **nodos aleatorios**: el método anterior tiene la desventaja de sobrecargar algunos sectores del anillo en algunos casos, debido a que existen muy pocas combinaciones para la asignación de un conjunto de nodos. La idea natural para solucionar este problema es aumentar la cantidad de combinaciones posibles. Para esto, se necesita que para una consulta se obtenga una secuencia de nodos, en lo posible distinta para cada consulta. Esta secuencia de nodos no necesariamente debe ser contigua, al contrario, mientras más dispersa en el anillo, mejor. En la Figura 15(b) se observa un ejemplo para la consulta q y la secuencia $seq(q)$. Por motivos de legibilidad no se exponen las secuencias para *q1* y *q2*, pero es de fácil intuición entender que la posibilidad de que dos consultas tengan el mismo conjunto de máquinas disminuye. Esto es debido al espacio de secuencias posibles para un conjunto de $r(q)$ nodos. Sea NCS el conjunto de nodos totales del Servicio de Cache y $r(q)$ el conjunto de nodos de una consulta, entonces la cantidad de posibles secuencias de $r(q)$ nodos es $\binom{NCS}{r(q)}$. Dado esto, es muy poco probable que dos consultas frecuentes presenten

la misma secuencia (por el espacio de soluciones), y/o contengan subconjuntos de la secuencia en común. Para obtener esta secuencia, la opción más sencilla es utilizar un conjunto de funciones hash que sean dependientes del *string* mismo de las consultas.

- $r(q)$ **nodos con balance de carga**: los esquemas anteriores no consideran carga de trabajo de los nodos. La idea es que los $r(q)$ nodos a seleccionar para una consulta frecuente sean los $r(q)$ nodos con menos carga, pero en el momento en que se realiza la selección. Para eso, se debe tener la información de la carga de las máquinas, pero como se mencionó en el capítulo 2, esta información es manejada por el FS. Con esto se considera la carga de trabajo de los nodos en la asignación de consultas frecuentes. Se debe recalcar que la carga de trabajo de los nodos varía constantemente. En la Figura 15(c) se muestra la carga de trabajo experimentada por cada nodo del servicio en un instante de tiempo. Si la consulta q debe replicarse en $r(q) = 3$ nodos, entonces se tiene que $real(q) = \langle P0, P4, P3, P5 \rangle$. Notar que para las consultas *q1* y *q2*, sus secuencias comienzan con los nodos $\langle P4, P3, P5 \rangle$ (son los nodos que tienen menos carga en ese instante). Este método tiene la misma anomalía del primer método: una reducida cantidad de combinaciones distintas en un instante dado (para un instante de tiempo, la secuencia que involucra a todos los nodos es única). Es más, todas las consultas frecuentes tendrían la misma secuencia de nodos. Por último, para producir cambios en estas combinaciones se debería medir la carga de trabajo constantemente, lo cual podría imponer un alto *overhead*.

Dados los algoritmos anteriores, sus combinaciones posibles y a la incapacidad de medir la carga de trabajo del Servicio de Cache en intervalos reducidos, la opción que se utiliza es $r(q)$ nodos al azar. Si a esto se suma la capacidad dinámica C_d de cada nodo, se tiene que las consultas frecuentes serán distribuidas en forma balanceada entre los nodos del Servicio de Cache.

5.3.3. Selección de Nodos

Anteriormente se describió cómo seleccionar los nodos dado el nivel de replicación de una consulta. Este conjunto de nodos del CS indica los candidatos posibles de atención para una consulta frecuente. En el capítulo 3 se indica que lo óptimo es que cada petición visite sólo un nodo del CS. Entonces el problema es que dada una instancia de una consulta q y su conjunto de nodos $real(q)$, seleccionar un nodo de este conjunto siguiendo algún criterio. Se debe destacar que la selección de un nodo de $real(q)$ es independiente del método de obtención de una secuencia $seq(q)$. Una opción es hacer una selección aleatoria o mediante el algoritmo *Round-Robin* entre los nodos del conjunto $real(q)$. Otra opción más inteligente es usar algoritmos de balance de carga, como los descritos en [GPM08], es decir, considerar el estado actual de los nodos para determinar el nodo a visitar. Nuevamente se tiene la dificultad de que la carga de cada nodo se actualiza cada cierto período de tiempo, del orden de minutos, por lo que el nodo menos cargado lo seguirá siendo hasta la nueva medición. En el capítulo 2 se menciona que el costo de procesar una petición en cualquier nodo del servicio toma tiempo constante [SJPBY08] (a menos que existan nodos heterogéneos). Por esta razón, si se utiliza el algoritmo *Round-Robin* se tendrá que el conjunto de nodos $real(q)$ recibe el mismo número de peticiones (con una diferencia de uno), y por ende la carga de trabajo generada por q sea aproximadamente la misma en todos los nodos de $real(q)$.

Lo anterior indicó el modo de seleccionar un nodo desde un conjunto de nodos, con el fin de distribuir la carga de una consulta frecuente en forma más homogénea. Se debe recordar que el objetivo es consultar el nodo seleccionado para determinar si una consulta está en cache o no. Si es *hit*, entonces el nodo responde con la respuesta. Si es un *miss*, entonces la respuesta debe ser computada utilizando el servicio de índice. Una vez hecho esto, la respuesta debe ser enviada al CS. La pregunta que cabe acá es ¿a qué nodo del CS se envía la respuesta? La solución para esto es que siempre la actualización de una consulta (posterior a un *miss*) se realiza en el mismo nodo que reportó *miss*.

5.4. Evaluación Experimental

Antes de comenzar la evaluación experimental, se analizará el costo temporal de la propuesta. Sea n el número de contadores necesario para obtener los top-k. Para implementar la monitorización de elementos a través de Space-Saving se incurre en un costo promedio de $O(1)$, tanto para la inserción como la actualización (peor caso es $O(n)$). Para determinar los elementos que se deben replicar, se incurre en un costo $O(k)$, ya que sólo se deben obtener los k elementos más frecuentes. Esta última operación es cada Δt unidades de tiempo. Finalmente, se tiene un costo promedio de $O(1)$ para saber si una consulta debe replicarse o no, y obtener su nivel de replicación.

Anteriormente se definió que para obtener las $k = 100$ consultas más frecuentes con un error ϵ entre 0,1 y 0,05 era necesario utilizar entre 10 mil y 20 mil contadores. Para la experimentación se definió que se utilizaran 10 mil contadores, ya que según lo observado no representa un cambio drástico en el desempeño del Servicio de Cache. Este parámetro es independiente del número de nodos y de la política de cache implementada.

Para determinar el mejor valor de k, se ejecutan 100 millones de consultas de Enero de 2012 en un Servicio de Cache compuesto por 20 nodos. La eficiencia promedio se encuentra en la Figura 16. Al igual que fue analizado anteriormente, se observa que el desempeño tiene un aumento drástico desde el valor $x = 0$, que corresponde a la situación en que no se replican consultas. Los mejores valores están entre $k = 20$ y $k = 30$, tras lo cual se observa una reducción del desempeño, pero que igualmente es sobre un esquema sin replicación de consultas. Entre estos valores, la eficiencia promedio es aproximadamente un 85 %. En el experimento, el umbral C_d para la capacidad dinámica de los nodos fue calculado con la información de las k consultas más frecuentes.

Para actualizar el nivel de replicación de las consultas top-k (Algoritmo 2), se definió un intervalo de un minuto. La decisión se basa en que las frecuencias del intervalo actual sirven como un estimador de la frecuencia y el impacto de las consultas top-k, por lo que intervalos muy largos podrían no reflejar correctamente estos valores. Además, esta decisión también tiene relación con la detección de consultas

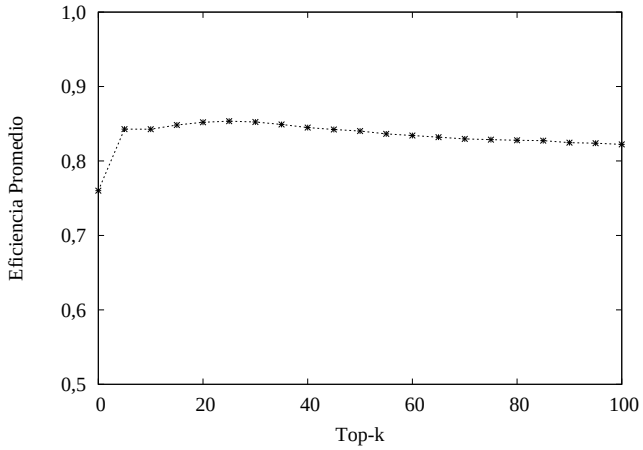


Figura 16: Eficiencia promedio versus top-k.

en ráfaga detallada más adelante, pero principalmente se debe analizar las frecuencias de las consultas para detectar valores atípicos en una ventana de tiempo. Para esto, se deben tener intervalos de evaluación reducidos. Por otro lado, cada vez que se obtienen los top-k y se analizan, se reinicia la estructura de monitorización. Esta decisión tiene relación con lo mencionado anteriormente (utilizar como un estimador de la frecuencia del próximo período). Además, si no se reinician los contadores, las consultas frecuentes permanentes tienen crecimiento sostenido (lineal) en el tiempo.

Para comenzar el análisis de la estrategia propuesta, en la Figura 17 se presenta una caracterización inicial del desempeño de un Servicio de Cache compuesto por 20 nodos con 100 mil entradas cada uno que implementa Consistent Hashing. Las consultas utilizadas corresponden a los días 2 y 3 de Enero de 2012. No se considera el primer día de Enero de 2012 para no sesgar los resultados con el *calentamiento* del Servicio de Cache. En la Figura 17(a) se observa la carga de trabajo asignada a cada nodo. Para ver con mayor claridad el resultado y su tendencia, en la Figura 17(b) se presenta la misma carga de trabajo pero con suavizado [1]. En la figura se observa que la estrategia básica presenta resultados muy variados en sus nodos, variando desde un 60 % de carga de trabajo hasta un 100 %. Específicamente, según los datos recolectados, 9 de los 20 nodos presentan una utilización del 100 % (saturación) en algún

[1] *smooth csplines* en gnuplot

intervalo de tiempo. Además del desbalance, se observan variaciones y algunas alzas abruptas. Este es un comportamiento típico de Consistent Hashing, experimentado en la mayoría de las simulaciones, debido a la nula respuesta ante el desbalance. Por otro lado, en la Figura 17(c) se tiene que la eficiencia del servicio se mantiene casi constante con un promedio de 77,1 %. Este mal resultado es debido principalmente a que la mayor parte del tiempo existen nodos con un 100 % de utilización (carga máxima). Finalmente, la tasa de *hit* promedio del servicio es 50,0 %, según los datos de la Figura 17(d).

El objetivo que se busca es reducir las diferencias entre la carga de trabajo experimentada por los nodos, para así aumentar la eficiencia del servicio. Como se verá más adelante, esto tiene un efecto colateral en la tasa de *hit*.

Según los parámetros descritos anteriormente para la estrategia propuesta, los resultados se encuentran en la Figura 18. Al igual que en el análisis previo, la Figura 18(b) representa el mismo resultado que la (a) pero con suavizado. Se observa que la estrategia propuesta presenta mejoras en gran parte del experimento, excepto en un par de intervalos en los cuales existe una utilización cercana al 100 % por parte de dos nodos del servicio (menor a los nueve nodos saturados de la versión básica). La diferencia principal radica en la duración de la saturación. Mientras que en la estrategia básica, la saturación en algunos nodos es prolongada, en la estrategia propuesta se tienen intervalos muy reducidos de saturación. Esta saturación es debida principalmente a alzas abruptas en el volumen de consultas, las cuales son detectadas e incluidas en la estrategia planteada, pero al ser esta una estrategia reactiva, las decisiones tomadas respecto a su asignación son tardías. Es por esta razón que la detección de consultas en ráfaga en forma temprana es un tópico a estudiar en el capítulo 7. El objetivo del mecanismo propuesto en este capítulo no es la mitigación inmediata de estas alzas abruptas, sino que realizar una asignación más justa de carga de trabajo. Podría ser visto como un balance de carga de largo plazo, que ataca directamente el desbalance estructural. Por otro lado, la eficiencia del servicio tiene un promedio de 88,9 % (Figura 18(c)), lo que representa una mejora de un 15,3 % sobre la estrategia básica. Esta diferencia será mejorada en los capítulos posteriores de este trabajo. Finalmente, la tasa de *hit* expuesta en la Figura 18(d) es en promedio

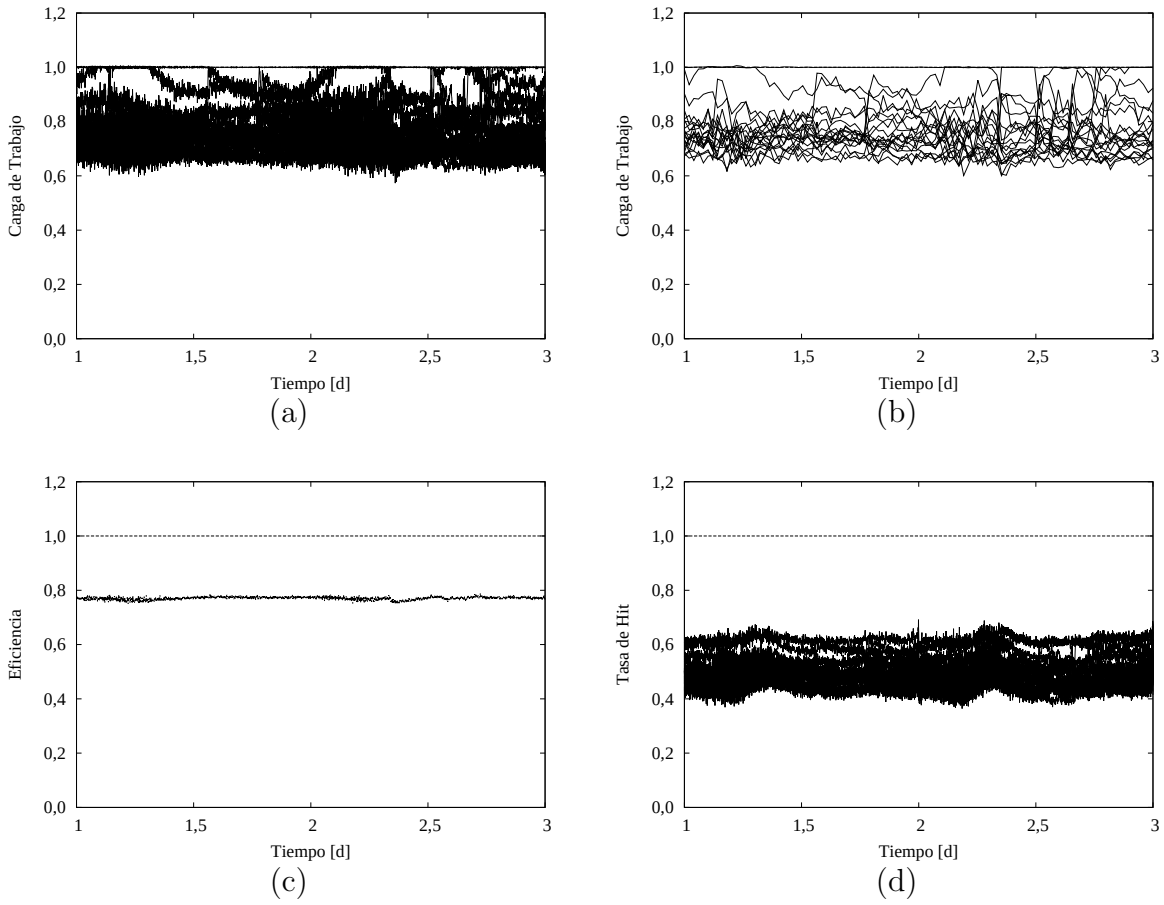


Figura 17: Carga de trabajo para un servicio compuesto de 20 nodos y 100 mil entradas (2 y 3 de Enero de 2012, aprox. 133 millones de consultas): (a) Consistent Hashing; (b) resultados de (a) con *smoothing*; (c) eficiencia; (d) tasa de *hit*.

51,0 % (versus 50,0 % de la estrategia básica).

La tasa de *hit* merece una discusión más profunda. Parece contradictorio que se alcance una mayor tasa de *hit* que con aquella estrategia que sólo presenta conjuntos disjuntos de entradas (en la estrategia propuesta existe replicación controlada). Se debe recordar que al replicar una consulta frecuente en un nodo adicional, esta consulta estará probablemente en la parte alta de la cache (más importante). Por otro lado, para que esta consulta frecuente replicada se mantenga en cache, se debe desalojar

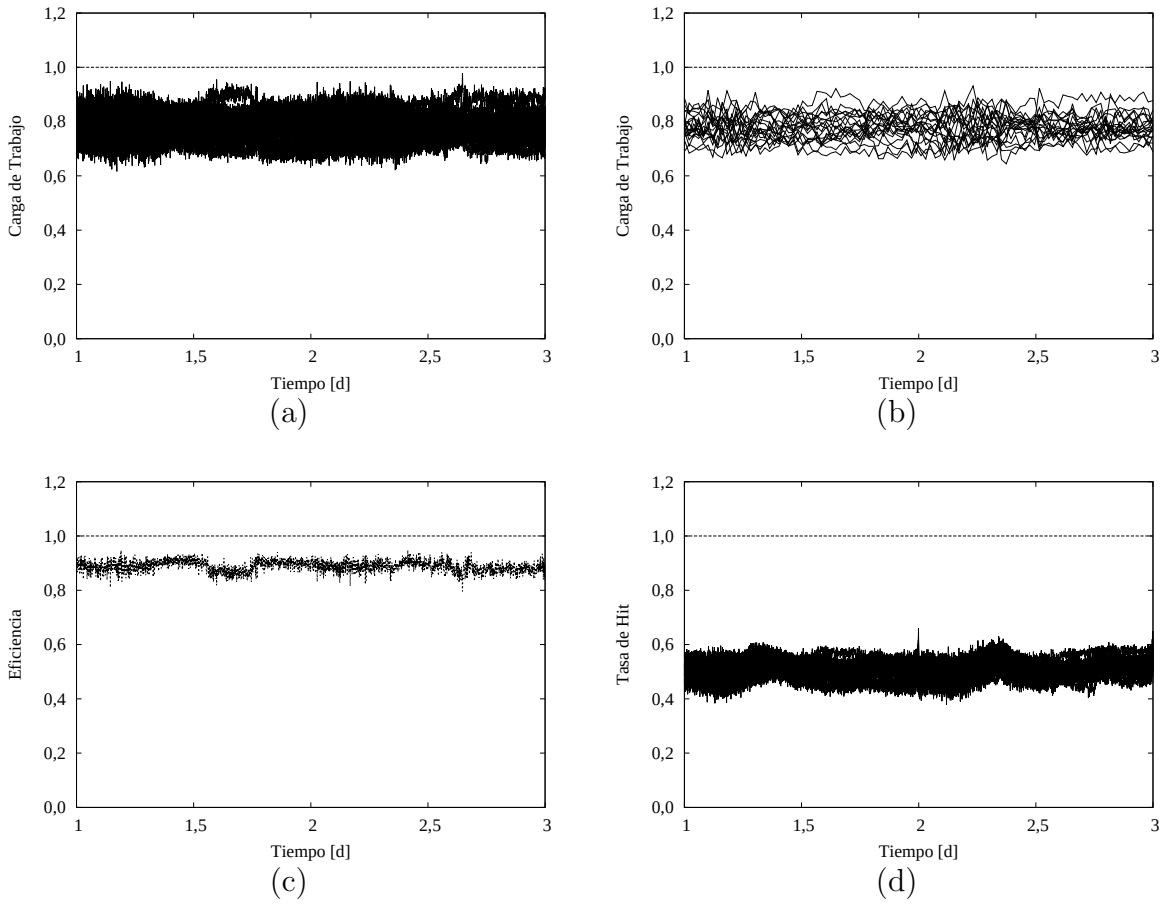


Figura 18: Carga de trabajo para un servicio compuesto de 20 nodos y 100 mil entradas (2 y 3 de Enero de 2012, aprox. 133 millones de consultas): (a) estrategia propuesta; (b) resultados de (a) con *smoothing*; (c) eficiencia; (d) tasa de *hit*.

una consulta. Las políticas de reemplazo seleccionan una entrada con menor prioridad para dar cabida a una de mayor prioridad, por lo que el número de *hits* de una consulta replicada, que es más frecuente, es más alto que una consulta desalojada, que es menos frecuente. Además, las consultas que se encuentran en la parte baja están constantemente siendo desalojadas o integradas a la estructura cache, por lo que existen períodos en que una misma consulta está y otros en que no está.

Es posible observar además que en la Figura 18(d) la tasa de *hit* experimentada por cada nodo en particular es más homogénea en contraste con la estrategia básica

(Figura 17(d)). Esto es muchas veces interpretado como un indicador de buen balance de un Servicio de Cache distribuido, ya que todos los nodos están operando a una tasa de *hit* similar, lo que implica que la cantidad de *hits* es similar en proporción a la cantidad de peticiones recibidas por cada nodo. Este es el efecto colateral y bondad adicional de la estrategia propuesta.

A continuación se compara la estrategia propuesta (etiqueta *PR*) con: (i) Consistent Hashing básico (*CH*); (ii) Esquema $P \times D$ con 10 particiones y 2 réplicas (*PD10*); (iii) Esquema $P \times D$ con 5 particiones y 4 réplicas (*PD5*); y (iv) Esquema $P \times D$ con 4 particiones y 5 réplicas (*PD4*). Todas estas estrategias realizan sólo una búsqueda en el CS (visitan un nodo). La estrategia matricial puede ser vista como una variación del paradigma "poder de d elecciones", pero en vez de establecer el nodo con menor carga de trabajo ante el arribo de cualquier instancia, se selecciona aleatoriamente una de las d posibilidades. Por ejemplo, en la estrategia (iii) $P \times D$ con 5 particiones y 4 réplicas (*PD5*), se selecciona con Consistent Hashing una de las 5 particiones y se elige aleatoriamente una de las $D = 4$ réplicas en esa partición (poder de $D = 4$ elecciones). Cabe señalar la implementación del protocolo de consistencia en las estrategias matriciales.

En la Figura 19(a) se muestra la eficiencia de las estrategias descritas y en las etiquetas se incluye la eficiencia promedio. Se observa que la aplicación de un esquema matricial tiene mejor desempeño que la estrategia CH, con valores desde un 82,1 % a un 93,2 %. Esto implica una mejora de un 20,9 % con la estrategia PD5, mayor que el 15,3 % reportado por la propuesta de este trabajo. Naturalmente, a medida que se disminuye el número de particiones se tiene un mejor balance, por lo que la estrategia CH (máximo número de particiones) tiene un peor desempeño que un esquema replicado. La eficiencia es una medida que analiza el comportamiento de cada estrategia en términos de balance pero en forma aislada, por lo que sólo es una primera línea de comparación. En la Figura 19(b) se observa la tasa de *hit* promedio de las distintas estrategias. Esta segunda comparación es quizás la más importante, ya que el objetivo principal de un Servicio de Cache es tener una alta tasa de *hit*. En este caso, el esquema matricial muestra una disminución de la tasa de *hit* y se observa que entre menos particiones (más réplicas), mayor es la disminución (desde un

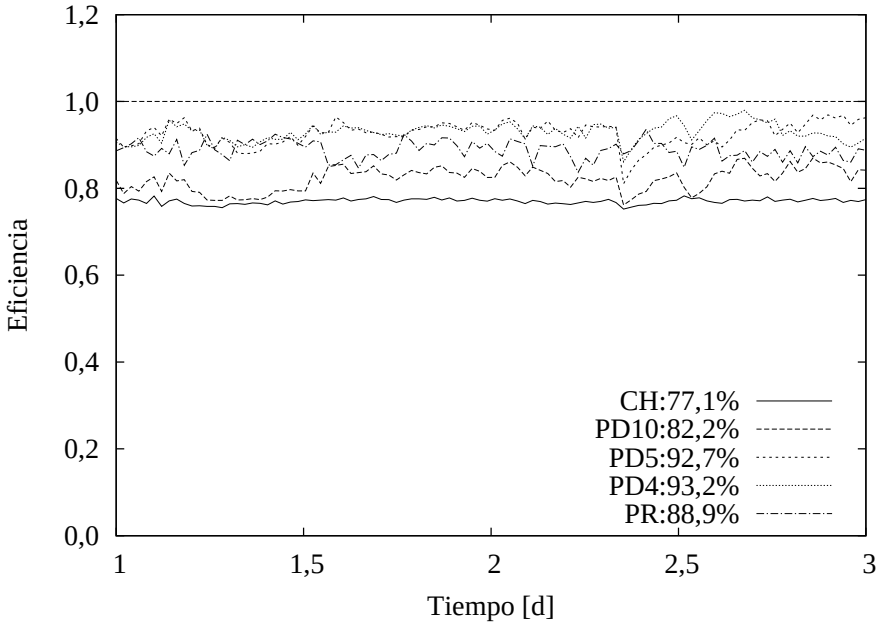
Tabla 7: Resumen de resultados para un Servicio de Cache compuesto de 20 nodos y 100 mil entradas (2 y 3 de Enero de 2012, aprox. 133 millones de consultas).

Estrategia	Tiempo de Respuesta Promedio [ms]	Tasa de *Hit*	Eficiencia	Saturación (# Nodos)
CH	36,0	50,0 %	77,1 %	9
PD10	31,4	47,4 %	82,2 %	2
PD5	32,1	45,3 %	92,7 %	0
PD4	32,6	44,7 %	93,2 %	0
PR	29,4	51,0 %	88,9 %	0

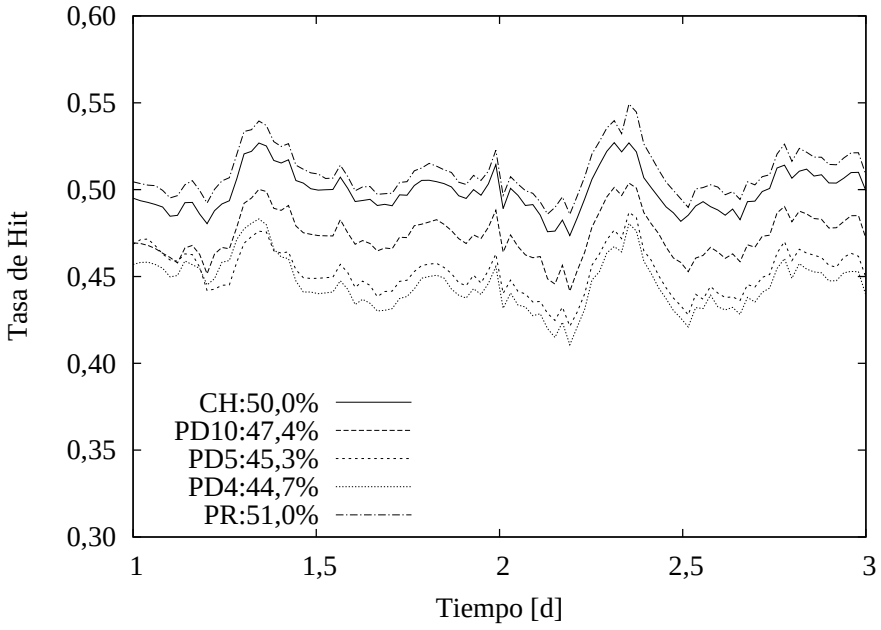
50,0 % a un 44,7 %). La estrategia propuesta se beneficia de un esquema con alta tasa de *hit* (CH), pero incluye un mecanismo para mitigar la desventaja de esta misma estrategia (alto desbalance). El balance que muestra la propuesta no es óptimo, pero en los siguientes capítulos esto será mejorado.

En la Tabla 7 se encuentra un resumen de los resultados obtenidos para las distintas estrategias, incluido el tiempo de respuesta para el conjunto de consultas. La estrategia propuesta presenta el mejor tiempo de respuesta con 29,4 [ms], lo que se traduce en una mejora del tiempo promedio de un 18 % en comparación con la estrategia básica CH y un 6,4 % con la estrategia matricial PD10. En la tabla es posible observar que una alta tasa de *hit* o un buen balance no implica el menor tiempo de respuesta promedio, sino que una combinación de ambos lo que es experimentado por la estrategia propuesta. La estrategia CH tiene una alta tasa de *hit*, pero la saturación experimentada, evidenciada por la eficiencia y el número de nodos saturados, hace que se alcance el peor tiempo de respuesta promedio. Por otro lado, las estrategias matriciales mejoran la eficiencia pero tienen una peor tasa de *hit*. Acá se debe notar un equilibrio entre la tasa de *hit* y el balance. Mientras más desbalanceado es el servicio, más larga será la espera de las consultas asignadas a los nodos con más carga de trabajo. Esto se ve agravado por la saturación de nodos, como se observa en la comparación de las estrategias CH y PD.

Finalmente, en la Figura 20 se muestran los resultados para un servicio de 80 nodos. Se observa que la estrategia propuesta mejora la estrategia CH en términos de utilización y tasa de *hit*, y también que la estrategia PD8 y PD10 tienen una



(a)



(b)

Figura 19: Servicio de Cache compuesto de 20 nodos y 100 mil entradas (2 y 3 de Enero de 2012, aprox. 133 millones de consultas): (a) eficiencia; (b) tasa de *hit*.

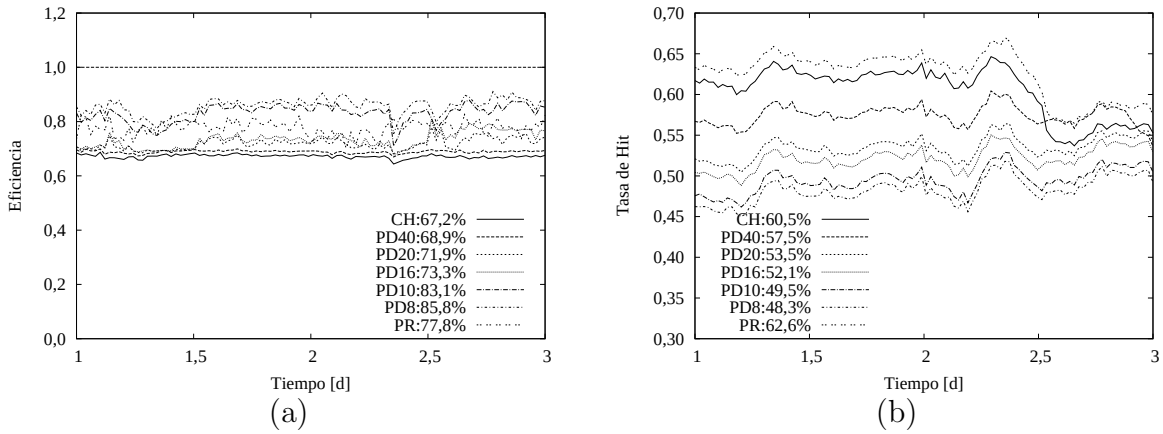


Figura 20: Resultados de distintas estrategias para un servicio compuesto de 80 nodos y 100 mil entradas (2 y 3 de Enero de 2012, aprox. 133 millones de consultas): (a) eficiencia; (b) tasa de *hit*.

mejor eficiencia que la propuesta, pero menor tasa de *hit*. Un punto importante a notar, es que a medida que se tienen más nodos, todas las estrategias experimentan una degradación de la eficiencia. Esto es debido a dos razones: (i) mientras más particiones se tienen, menor es la eficiencia; (ii) todas las estrategias son rígidas (no hay variación en sus configuraciones).

5.5. Conclusiones

La estrategia propuesta trata de rescatar dos elementos clave de las estrategias base analizadas: (i) utilización eficiente de las entradas (alta tasa de *hit*); y (ii) replicación controlada (mejorar balance). El uso correcto de las entradas hace que se reduzcan las peticiones al servicio de índice, pero esto tiene un costo: el alto desbalance por la asignación de consultas hecha por *Consistent Hashing*. Por otro lado, la replicación de nodos ayuda a balancear la carga, pero no se utilizan las entradas correctamente. El método propuesto realiza replicación controlada a las consultas más frecuentes, que son las que realmente tienen el mayor impacto en el desbalance detectado.

La replicación es el mecanismo clásico para abordar la disponibilidad de información, pero también como mecanismo de balance de carga y tolerancia a fallas. Específicamente, en este capítulo se busca que un grupo de consultas con alto impacto sean "homogeneizadas", es decir, en vez de que estas consultas aparezcan como un punto denso en el anillo, se distribuye la responsabilidad de esta consulta en varios puntos menos densos. Para ello, se monitoriza un conjunto de consultas (las más frecuentes) mediante el algoritmo *Space-Saving* y se proponen algoritmos para seleccionar cuántas réplicas deben atender una consulta, cuáles nodos seleccionar y cómo seleccionar un nodo ante el arribo de una instancia. Cada una de las decisiones fue analizada y se propuso una solución considerando aspectos de desempeño, eficiencia y *overhead* adicional de la estrategia propuesta. Por ejemplo, no es práctico obtener la carga de trabajo a intervalos muy pequeños de tiempo (del orden de milisegundos).

En resumen, la estrategia contempla dos pasos: (i) monitorización de consultas, y (ii) determinación de cómo distribuir las consultas frecuentes. Para el punto (i) se utiliza el algoritmo Space-Saving. Para obtener las k consultas más frecuentes, o consultas top-k, se deben utilizar c contadores. Para determinar las top-100 consultas, es necesario del orden de c=10.000 a c=20.000 contadores. Este algoritmo de monitorización es examinado cada Δt unidades de tiempo. Luego, en el punto (ii), se determinan las consultas top-k, y para cada una de ellas se establece el nivel de replicación y el conjunto de nodos en los cuales asignar las instancias de esas consultas. Finalmente, en tiempo de ejecución, ante el arribo de cada instancia de las consultas pertenecientes a los top-k, se debe seleccionar un nodo de los grupos definidos anteriormente.

La estrategia propuesta impone bajo *overhead*, ya que se utiliza una estructura de monitorización eficiente en tiempo y espacio. Además, el conjunto de decisiones detalladas en el párrafo anterior, se realiza cada cierto intervalo de tiempo, del orden de minutos, por lo que no representa una sobrecarga importante.

Este esquema propuesto es adaptivo, ya que a medida que sube el tráfico debido a los husos horarios, más nodos para una consulta frecuente son necesarios. Lo opuesto pasa a medida que disminuye el tráfico. A pesar de ser un esquema adaptivo, se ha notado que esto no es suficiente para mitigar completamente las variaciones dinámicas,

tanto abruptas como sostenidas, ya que el principal objetivo de esta estrategia es resolver el desbalance estructural, no el desbalance dinámico. El balance estructural puede ser visto como un balance de carga de largo plazo.

En el desarrollo de este trabajo, se ha notado que este problema, el del desbalance estructural, es la causa que tiene mayor impacto en el Servicio de Cache. Esta causa siempre estará presente, debido al comportamiento de las consultas de usuario. Por lo anterior, estrategias como la propuesta, que hagan más justa la asignación de trabajo a los nodos, deben ser consideradas en el diseño de servicios de cache. La estrategia reporta buen desempeño en términos de balance de carga, además hace que no sea necesaria la replicación de nodos completos, sino que la replicación es acotada y dependiente del impacto de las consultas.

Con respecto a los resultados obtenidos en la configuración descrita del Servicio de Cache, se reporta una mejora de un 18 % en términos de tiempo de respuesta y un 15,3 % en la eficiencia de la carga de trabajo; un incremento de un 2 % en la tasa de *hit* y la inexistencia de saturación de nodos. Todos estos resultados son en comparación con la estrategia base CH. Con respecto a la estrategia base que reporta el menor tiempo de respuesta promedio (PD10), se tiene una mejora de 6,4 %.

En este capítulo se hicieron algunos experimentos preliminares para mostrar que es posible mejorar el desempeño con la estrategia propuesta, sin embargo en el siguiente capítulo se analizará un nuevo esquema de balance de carga y que funciona en *tandem* con esta estrategia. Se verá posteriormente que el funcionamiento conjunto de ambas estrategias mejora aún más los resultados expuestos en este capítulo.

Finalmente, de los resultados se observa que la propuesta mejora visiblemente el desempeño de Consistent Hashing, pero aún queda espacio para nuevas optimizaciones que serán estudiadas en los siguientes capítulos. Específicamente mecanismos de balance de carga dinámico que reduzcan aún más el desbalance y la detección temprana de consultas en ráfaga.

Balance Dinámico de Carga

Al igual que en [SMLN+03, KK03, SXC06, DHJ+07a, DHJ+07b], en esta tesis se propone realizar la localización de ítemes en el Servicio de Cache (CS) a través de Consistent Hashing [KLL+97, KSB+99], pero se propone un nuevo esquema de balance de carga dinámico sobre esta estrategia para lidiar con las variaciones dinámicas, los fallos y la inserción de nodos en el servicio.

En la sección 6.1 se analiza el problema de variaciones dinámicas en el comportamiento de las consultas de usuario, lo cual tiene un impacto directo en el CS. En la sección 6.2 se diseña un nuevo método de balance de carga para aplicaciones basadas en Consistent Hashing que sufran de variaciones dinámicas en la carga de trabajo de los nodos. En la sección 6.3 se define la clase de algoritmos factibles de utilizar en este nuevo esquema, para posteriormente hacer una revisión del Estado del Arte de esa clase de algoritmos en la sección 6.4. Los algoritmos estudiados presentan propiedades de convergencia y estabilidad del proceso de balance de carga, por lo que en la sección 6.5 se describe el tipo de demostraciones utilizadas para determinar estas propiedades. El método propuesto sufre de una anomalía en el contexto de servicios de cache que es analizada en la sección 6.6. Esta anomalía es controlada por el funcionamiento conjunto del mecanismo de balance dinámico propuesto en este capítulo y la propuesta de distribución de ítemes frecuentes descrita en el capítulo 5. Es decir, el balance dinámico propuesto es un complemento a la técnica detallada en el capítulo anterior. Posteriormente, en la sección 6.7 se detallan los parámetros necesarios a configurar en el mecanismo propuesto, el cual es evaluado en la sección 6.8 en conjunto con la distribución de ítemes frecuentes. La experimentación permite ver cómo el balance dinámico mejora la distribución de ítemes frecuentes, y permite observar cómo el funcionamiento conjunto de las estrategias es más eficiente que otras alternativas base ante variaciones dinámicas presentes en los *logs* de consultas

reales utilizados. Además, la estrategia conjunta actúa efectivamente ante cambios en el número de nodos del CS. Tal como la distribución de consultas frecuentes fue categorizada como un balance de largo plazo, la propuesta de este capítulo realiza un balance de mediano plazo. Finalmente, las conclusiones del capítulo son expuestas en la sección 6.9.

6.1. Contexto

Las consultas de usuario varían a través del tiempo en su frecuencia y composición. El intervalo de variación puede ser desde días y horas hasta minutos y segundos. Una de las principales razones del tráfico variable es que la atención del usuario no está siempre puesta en los mismos tópicos de búsqueda. Otros motivos secundarios de la variación del tráfico de consulta son los horarios laborales y distintos husos horarios.

La localización de ítemes en el Servicio de Cache es hecha a través de Consistent Hashing [KLL+97, KSB+99], en el cual todo el espacio de consultas es mapeado en un anillo con ayuda de una función hash (capítulo 3). Cada nodo tiene asignado un intervalo continuo y que no se traslapa con los intervalos de otros nodos. Además, no existen agujeros (rangos sin asignar) en el anillo. Para asignar una consulta a un nodo, basta intersectar el punto que representa a la consulta con un intervalo. Luego, el nodo dueño de este intervalo es el asignado para procesar esa consulta.

La asignación de rangos a nodos ha sido clásicamente fija a través de dos técnicas principales: puntos equidistantes y puntos aleatorios. En el área de aplicaciones Peer-to-Peer (P2P), la razón de esta asignación es la premisa de que las consultas de usuario siguen una distribución uniforme (sección 3.3). Bajo las estrategias de asignación de rangos y una distribución uniforme de consultas, con una alta probabilidad todos los nodos experimentarán la misma asignación de carga de trabajo. Además, no existen trabajos en los cuales se estudien cómo afectan en el desempeño las variaciones en el tráfico de consultas. Finalmente, la asignación fija de estos rangos impide corregir la carga asignada a cada nodo dependiendo de las variaciones en el tráfico de consultas.

Otra técnica utilizada para balancear carga es la inclusión de nodos virtuales, que

consiste en replicar N veces un punto asociado a una partición en el anillo. Según la mayoría de los trabajos, esta técnica ayuda a balancear la carga pero en el contexto de distribuciones uniformes. Los nodos virtuales implican que distintos rangos disjuntos del anillo corresponden a una misma partición, por lo que una partición puede verse como un conjunto de rangos no contiguos. Cada vez que las consultas intersectan alguno de estos rangos, esas peticiones son asignadas a la misma partición. Si un sistema de cache se configura con N nodos virtuales, entonces cada partición tiene asignada N rangos disjuntos.

Además, dada alguna asignación fija de rangos, la carga experimentada por los nodos varía a través del tiempo. Para mostrar la variabilidad de la carga de trabajo, en la Figura 21 se observan cargas de trabajo para distintos intervalos de tiempo durante Febrero de 2012 (cada figura representa 2 días de carga de trabajo de un Servicio de Cache compuesto por 10 particiones). Es posible observar la alta variabilidad existente en la carga de trabajo de las distintas particiones. El caso más extremo se presenta en la Figura 21(b), en la cual se observa una abrupta alza de carga de trabajo en una partición. Esta abrupta alza es debido a la consulta en ráfaga "whitney houston", generada con la muerte de esa celebridad el día 11 de Febrero de 2012.

La hipótesis de este capítulo es que considerando la carga de trabajo de los nodos del Servicio de Cache y la asignación de ítemes a través de Consistent Hashing, es posible diseñar mecanismos de balance de carga dinámicos para mitigar los efectos de las variaciones en el tráfico de consultas, de las fallas y agregación de nodos, manteniendo un bajo tiempo de respuesta promedio.

6.2. Un Nuevo Método de Balance de Carga

El análisis de la sección anterior muestra que es necesario variar dinámicamente la carga de trabajo de cada nodo. El balance de carga clásico puede ser dividido en estático y dinámico [KA99]. Los algoritmos de planificación (*scheduling algorithms*) estáticos reciben comúnmente como entrada un grafo que representa las tareas y las dependencias entre ellas, y dan como salida una planificación que cumpla las restricciones y minimize el tiempo total de ejecución. Generalmente esto se realiza

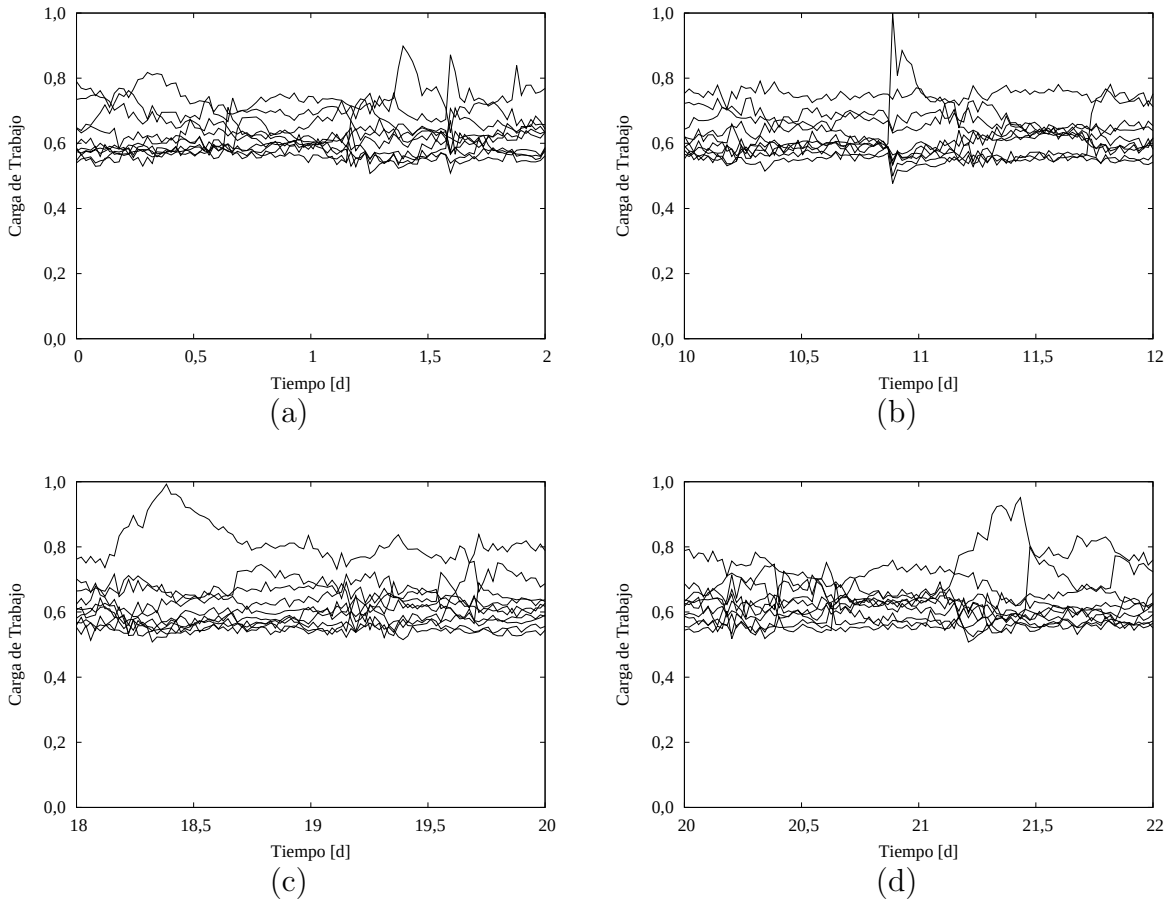


Figura 21: Carga de trabajo para distintos intervalos de dos días durante Febrero de 2012: (a) 1 y 2 de Febrero; (b) 11 y 12 de Febrero; (c) 19 y 20 de Febrero; y (d) 21 y 22 de Febrero.

en tiempo de compilación. Por otra parte, los algoritmos de planificación dinámicos reciben como entrada el estado actual del conjunto de nodos (en términos de carga) y una tarea (caracterizada generalmente con un tiempo de ejecución), y produce como salida la máquina en la que debe ser asignada la tarea.

El problema a resolver tiene un componente que los algoritmos detallados anteriormente no consideran: la dependencia entre consulta y nodo. Es decir, la asignación de una consulta debe hacerse a una máquina en particular, siguiendo un método determinístico, ya que ese es el procedimiento realizado a través de Consistent Hashing

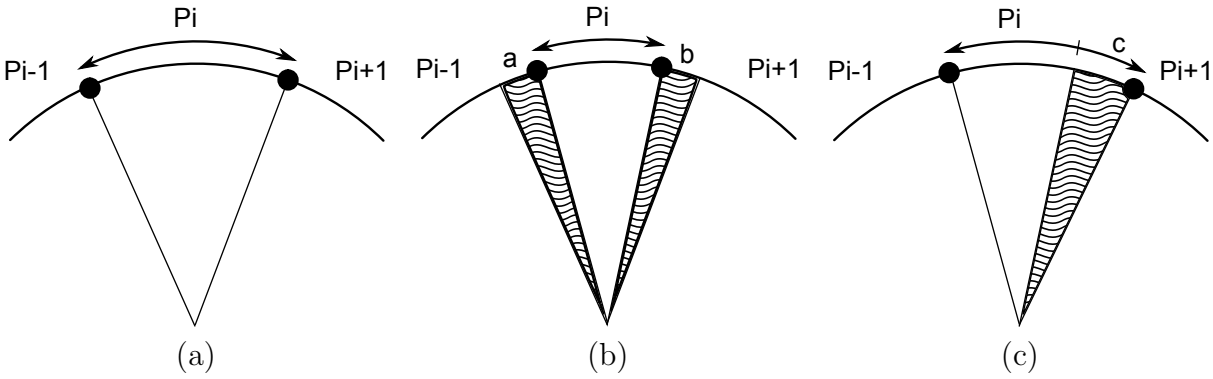


Figura 22: Balance de carga: (a) situación inicial con sobrecarga de P_i; (b) reducción de carga de P_i mediante el aumento de los rangos de P_{i-1} y P_{i+1}; (c) reducción de carga de P_{i+1} mediante el aumento del rango de P_i.

y además porque así se maximiza la tasa de *hit*.

El esquema de balance de carga dependiente de los datos propuesto (*data-dependent load balancing*), consiste en un anillo Consistent Hashing que comienza con asignación equidistante de rangos a nodos, pero la diferencia fundamental es la posibilidad de *aumentar* y *disminuir* el rango de cada nodo en función de la carga de trabajo experimentada por éste. *Aumentar* un rango a un nodo implica añadirle carga de trabajo, ya que el número de consultas que abarca ese rango más amplio es mayor. Por otro lado, *disminuir* un rango implica una reducción de la carga de trabajo. Ambas operaciones se observan en la Figura 22.

Inicialmente se tiene la situación de la Figura 22(a), en la cual al nodo P_i se asigna un rango que no se traslapa con sus vecinos (P_{i-1} y P_{i+1}). Si el FS determina que P_i está sobrecargado, entonces sólo los nodos P_{i-1} y P_{i+1} pueden ayudarlo a reducir su carga de trabajo (más adelante detalles de esto), lo que se aprecia en la Figura 22(b). En este caso, el nodo P_{i-1} aumenta su rango en a y P_{i+1} aumenta su rango en b. La elección de la amplitud de los rangos a y b depende de la carga de trabajo de los tres nodos en cuestión (y como se verá más adelante, del estado del servicio en general), ya que mientras más amplio el rango, más carga de trabajo implica. Análogamente, si P_i se encuentra con poca carga, puede alivianar la carga de P_{i+1} ampliando su rango c como se muestra en la Figura 22(c).

Una pregunta importante abierta en la explicación anterior es ¿por qué sólo los

vecinos pueden ayudar en la reducción de carga de un nodo? Dos razones principales responden esta pregunta:

- **Eficiencia en la Búsqueda**. En el capítulo 3 se indica que la implementación de la búsqueda en el anillo Consistent Hashing consiste en P números ordenados (uno por cada nodo del servicio). Cuando se necesita saber el nodo responsable para una consulta, se aplica una función hash a la consulta y se realiza la búsqueda binaria del valor hash sobre estos P números ordenados. A medida que aumenta P, aumenta el costo de la búsqueda, y considerando que son miles de consultas por unidad a buscar, entonces se requiere el mínimo costo de esta operación.

- **Facilidad en Balance de Carga**. Como será mostrado posteriormente, mientras menor sea P, más rápida es la obtención de una solución al problema de balance de carga, y también más rápida la convergencia a un estado balanceado. Es por esta razón que utilizar nodos virtuales y/o replicar un rango de una partición en otra sección del anillo no ayuda a un rápido balance de carga.

Por otro lado, la amplitud de las correcciones de rangos también merece análisis (rangos a, b y c en Figura 22(b) y (c)). Lo primero que se debe tener en mente es que al mover un rango, unas consultas cambian de nodo responsable. Por ejemplo, en la Figura 22(b) y considerando el rango a, todas las consultas que pertenecen a P_i ahora son manejadas por P_{i-1}. Esto implica dos cosas: (i) una reducción en la tasa de *hit*, aunque sea temporal, ya que los primeros accesos a las consultas del rango a en P_{i-1} serán *miss*; (ii) existen consultas del rango a que aún se encuentran en memoria cache de P_i y que pasará un tiempo antes de que sean desalojadas (dependiendo de la política de desalojo). Está claro que entre más abrupto el aumento/reducción del rango a, más notorios serán los dos efectos detallados anteriormente. Por lo anterior, sería ideal que los movimientos de los rangos fuesen incrementales, es decir, de a pequeños desplazamientos hasta resolver el desbalance. Los efectos del punto (i) podrían ser mitigados enviando las respuestas del rango a en P_i a P_{i-1}, pero esto implicaría: (a) informarle al nodo P_i que obtenga las consultas del rango a; (b) el nodo P_i recorre

toda su cache para obtener las consultas que intersectan a; y (c) enviar esas consultas a P_{i-1}. Esta opción no es considerada en este trabajo.

Finalmente, se debe responder la siguiente pregunta ¿con qué objetivo se mueven los rangos entre los nodos? Esta pregunta va con el trasfondo de esta tesis: tener un Servicio de Cache balanceado. Es decir, la idea de aumentar/reducir los rangos de los nodos tiene como fin fundamental conducir a un servicio en equilibrio en términos de carga de trabajo. El Servicio de Cache debe estar en continuo monitoreo para determinar cuál es el estado de éste y tomar acciones correctivas para balancearlo. Esta acción no representa una carga de trabajo importante para el Servicio de Cache.

Se ha propuesto la idea de aumentar/reducir los rangos en forma dinámica para balancear, pero no se ha determinado un algoritmo apropiado para encontrar un servicio balanceado. Cada nodo reporta una carga de trabajo distinta, la cual es monitoreada por el FS. Este último servicio puede determinar la carga óptima de trabajo de cada nodo: el promedio de la carga de trabajo del servicio. Esto es, dado un conjunto de cargas de trabajo de los nodos del servicio $L_1, L_2, ..., L_P$, la carga óptima de trabajo de cada nodo es $\sum_{i=1}^{P} L_i/P$.

En la siguiente sección se describirán los algoritmos adecuados para este problema y el análisis respectivo.

6.3. Clase de Algoritmos

La primera aproximación para encontrar un algoritmo para el problema de balance de carga descrito anteriormente, es que bajo la detección de un nodo sobrecargado P_i, sólo los nodos vecinos P_{i-1} y P_{i+1} apoyen en la reducción de carga. Es decir, que P_{i-1} y P_{i+1} aumenten su rango para absorber más carga de trabajo. Esta solución podría implicar que P_{i-1} y P_{i+1} ahora estén sobrecargados. Este tipo de balance miope y focalizado en el nodo sobrecargado tiene varias desventajas:

- *Iteración*. Existe la posibilidad de caer en iteración entre un nodo sobrecargado y su vecino, y también de dos nodos que estén sobrecargados.

- *Localidad*. Al ser focalizado, el proceso de balance de carga no considera el

estado de los demás nodos del servicio.

- *Balance Insuficiente*. Existe la posibilidad de que el apoyo de nodos vecinos no sea suficiente para balancear el nodo sobrecargado.

- *Carga Promedio*. Imposibilidad de alcanzar el objetivo de que cada nodo del servicio tenga la carga óptima (carga promedio).

Para solucionar los problemas descritos, la clase de algoritmos a utilizar deben considerar todos los nodos. Más aún, dado el esquema de balance de carga, la clase de algoritmos a utilizar deben ser aplicables en la topología anillo. Con este tipo de algoritmos, se puede hacer una asociación natural entre la entrada y salida de los algoritmos utilizados y el anillo de Consistent Hashing. Por ejemplo, en la Figura 23(a) se observa una situación inicial de un conjunto de 4 nodos conectados con una red que sigue la topología anillo, por ende cada nodo sólo puede compartir carga de trabajo con sus vecinos. Además, se indica la cantidad de trabajo que experimenta cada nodo. Los algoritmos utilizados dan como resultado la Figura 23(b), en la que se observa una distribución de carga de trabajo compuesta por tripletas <*fuente*, *destino*, α>. El nodo *fuente* aumenta en $\alpha\,\%$ su rango para reducir la carga de trabajo de *destino*. Se enfatiza nuevamente que en el contexto estudiado, no existe correlación entre tamaño de los rangos y carga de trabajo asociada a cada nodo. También se enfatiza que el esquema de compartición de carga entre nodos vecinos tiene relación con la lógica de Consistent Hashing, y no necesariamente con la lógica de la topología de conexión de los nodos.

La asociación entre la salida de los algoritmos y el anillo de Consistent Hashing es como se representa en la Figura 23(c). Es decir, cada tripleta <*fuente*, *destino*, α> indica que el nodo *destino* debe disminuir $\alpha\,\%$ de carga de trabajo y debe ser distribuida al nodo *fuente* (el nodo *destino* debe ser indicado, ya que existen 2 posibles destinos de la distribución de carga: vecino derecho e izquierdo). Luego, α debe ser mapeado porcentualmente al rango que experimenta el nodo *fuente* en el anillo Consistent Hashing.

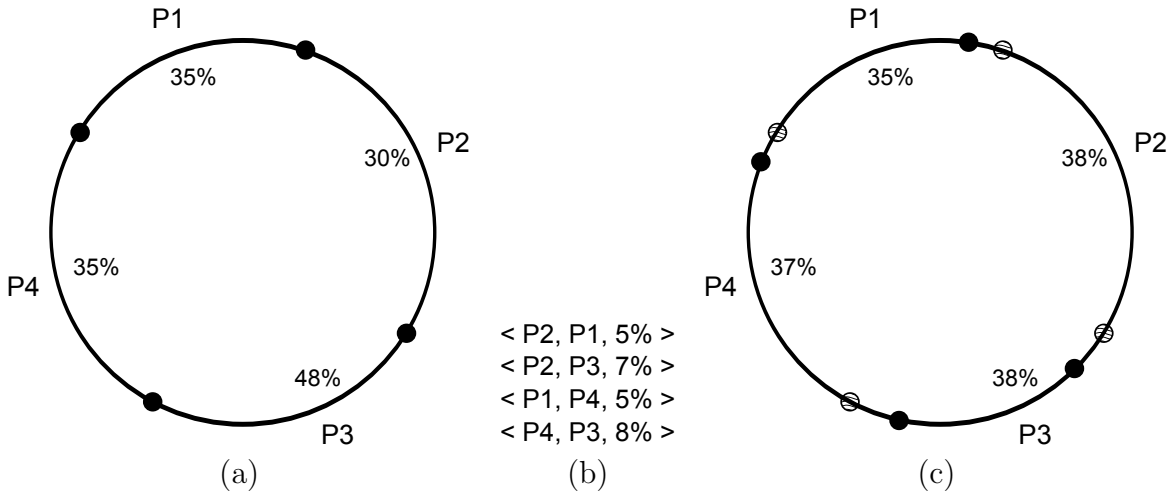


Figura 23: (a) Servicio de Cache desbalanceado con 4 particiones y su carga de trabajo; (b) salida de algoritmos de balance de carga; y (c) movimientos de rangos de acuerdo a la salida del algoritmo y la carga de trabajo resultante.

6.4. Algoritmos

En esta sección se estudian los algoritmos a utilizar para realizar el balance de carga. La idea fundamental es mover pequeñas secciones de rangos entre particiones consecutivas, más precisamente en los límites izquierdo y derecho de cada partición. Con este movimiento se minimiza el costo de administración de la propuesta a $O(\log n)$ para obtener la partición de cada consulta (como en el método original). Es decir, no se impone costo adicional a la búsqueda de la partición para una consulta. El único *overhead* que se impone con esta propuesta es la ejecución del algoritmo de balance de carga para obtener los movimientos correctos, pero como se verá más adelante, el algoritmo se ejecuta cada cierto intervalo de tiempo.

Siguiendo la clasificación de Xu *et al.* [zXL94], los algoritmos de balance de carga dinámicos pueden ser divididos en métodos *iterativos* y *directos*, de acuerdo a la forma en que la migración de carga toma lugar. Los métodos iterativos migran carga en pasos sucesivos y acotados, aproximando en cada paso la carga del sistema a la distribución global óptima de carga de trabajo. Estos métodos pueden ser clasificados en *determinísticos* y *estocásticos*. Por otro lado, los métodos directos dan un mapeo de los nodos y la carga a migrar en un paso.

En este trabajo se está interesado en los métodos directos e iterativos determinísticos, ya que en la evaluación se busca seleccionar un algoritmo que minimice la variación de los rangos (explicado posteriormente). La aleatoridad impuesta por los métodos estocásticos podría influir en los resultados y afectar el comportamiento de las simulaciones en sucesivas ejecuciones bajo el mismo *log* de consultas, por lo que no son considerados.

Existen tres clases predominantes de algoritmos iterativos determinísticos que serán consideradas en la evaluación: (a) Difusión; (b) Intercambio de Dimensión; y (c) Modelo Gradiente. Con los métodos de Difusión, un procesador intercambia carga de trabajo con todos sus vecinos al mismo tiempo en un solo paso. Los métodos de Intercambio de Dimensión intercambian carga de trabajo entre un nodo y sólo uno de sus vecinos en cada paso, y la nueva carga (debido a la migración de carga de trabajo) es empleada para continuar el proceso con los vecinos restantes. Los Modelos de Gradiente restringen la migración de carga al nodo menos cargado entre dos vecinos. Adicionalmente, un método directo es considerado: Intercambio de Dimensión Directa [WS97].

Métodos de Difusión

Con esta clase de métodos, en cada paso un nodo puede enviar carga a algunos vecinos y también recibir carga desde otros vecinos. Este proceso puede ser visto como una "difusión" de una fracción de la carga de trabajo de un nodo a sus vecinos inmediatos. Cybenko [Cyb89] estudió el modelo de difusión como un proceso iterativo, en el cual el modelo estático básico es gobernado por el parámetro de difusión α_{ij} ($0 < \alpha_{ij} < 1$). Este parámetro representa la porción de carga intercambiada entre los nodos i y j. El resultado principal del trabajo es que el proceso de iteración converge a una distribución de carga uniforme dada cualquier distribución inicial, bajo el supuesto de que no existan cambios en la carga de trabajo de cada nodo (no se terminen ni se agreguen nuevos trabajos) durante la ejecución del algoritmo.

Siguiendo el trabajo de Willebeek-LeMair *et al.* [WLR93], se han implementado dos algoritmos de difusión: Difusión iniciada por Fuente (*Sender Initiated Difussion*, SID) y Difusión iniciada por Destino (*Receiver Initiated Diffusion*, RID).

Intercambio de Dimensión

Este enfoque fue inicialmente analizado para topologías de hipercubo. La idea fundamental es que cada nodo intercambie carga en una dimensión a la vez con el vecino correspondiente. Cybenko [Cyb89] analizó el algoritmo en que la carga es dividida igualmente entre dos vecinos que pertenecen a la misma dimensión. El autor consideró cubos d-dimensionales y empleó etiquetas binarias que fueron asociadas con cada nodo (en la iteración i, con $1 \leq i \leq d$, los pares de procesadores que difieren en el i-ésimo bit intercambian carga de trabajo). El autor probó que el esquema de intercambio de dimensión determinístico tiene mejores propiedades de convergencia, al menos para hipercubos, que los métodos de difusión. En [HLM$^+$90], el método fue extendido a estructuras arbitrarias usando coloración de aristas en grafos.

La generalización del método de Intercambio de Dimensión usando un parámetro de intercambio de carga λ fue estudiado en [XL92]. El parámetro de intercambio λ sirve como un control a la cantidad de carga intercambiada entre dos vecinos. Los autores usaron coloración de aristas en grafos para analizar estructuras distintas a los hipercubos, y afirman que el parámetro de intercambio óptimo está altamente relacionado al tamaño y la topología de la red. Los mismos autores en [XL95] prueban que el parámetro óptimo para estructuras de anillo par de orden k es $\lambda_{opt}(k) = \frac{1}{1+\sin(\pi/m)}$, con $k = 2m$ y $k \neq 4$. También, demostraron que entre más nodos tenga el anillo par, más lenta se torna la convergencia a la carga promedio (Teorema 3.1 en [XL95]). Finalmente, con respecto a los casos impares, ellos concluyen que el comportamiento no es muy diferente de los casos pares, en términos de convergencia y valores óptimos para λ.

Modelo de Gradiente

La analogía de este método es como sigue: la diferencia de carga entre dos vecinos forma un contorno de los gradientes en las redes; entonces los procesadores altamente cargados (puntos altos del contorno) envían carga a los procesadores con poca carga siguiendo los gradientes (las regiones más bajas). El trabajo de Lin *et al.* [LK87] usa el gradiente superficial (*gradient surface*) que representa la distancia de cada procesador

al procesador con carga ligera más cercano.

Este método es iniciado por los procesadores con carga ligera y sólo ellos pueden interactuar con sus vecinos inmediatos. El balance global es alcanzado de una forma iterativa, ya que en cada iteración la información de carga local es propagada a los demás vecinos. Los procesadores con carga ligera reciben carga de trabajo desde los procesadores altamente cargados.

Métodos Directos

Los métodos de Intercambio de Dimensión son incapaces de alcanzar un estado balanceado en un paso. Wu *et al.* [WS97] proponen un algoritmo de balance dinámico directo llamado Intercambio de Dimensión Directo (Direct Dimension Exchange, DDE). Los autores mencionan que los métodos de intercambio de dimensión pueden tomar demasiadas iteraciones para converger a un estado balanceado. En contraste, su algoritmo toma sólo un *sweep* en alcanzar un estado completamente balanceado. La principal diferencia entre este método y los mencionados previamente, es la información necesaria para ejecutarse. La idea es calcular el número de tareas del sistema y determinar cuáles nodos están sobrecargados y cuáles están con poca carga. Entonces, el flujo de tareas entre los nodos es calculado considerando una *cuota* que representa el número de tareas que están sobre el promedio. Los autores idearon un algoritmo para estructuras de anillo, llamado *DDE-Ring*, que balancea la carga y minimiza el número de tareas transferidas entre los nodos. Para este fin, un algoritmo de flujo de costo mínimo es incorporado.

En la Tabla 8 se exponen algunas características de los algoritmos estudiados: (i) si se ejecutan de manera distribuida o centralizada; (ii) la cantidad de parámetros para la operación; y (iii) la cantidad de *sweeps* para llevar a la carga promedio. En la experimentación, cada algoritmo es configurado de acuerdo a los mejores parámetros establecidos en cada trabajo (notar que DM y DEM no necesitan parámetros).

Tabla 8: Características de los algoritmos estudiados (C: centralizado, D: distribuido).

Característica	DM [WS97]	DEM [Cyb89]	GDE [XL95]	GM [LK87]	RID [WLR93]	SID [WLR93]
Control	C	D	D	D	D	D
Parámetros	-	-	λ	LWM, HWM, δ	$L_{threshold}$	$L_{threshold}$
Sweeps	1	$f(n)$	$f(n, \lambda)$	-	-	-

6.5. Estabilidad y Convergencia

La mayoría de los algoritmos mencionados anteriormente garantizan estabilidad y convergencia del proceso de balance de carga. Esto implica que, para un sistema compuesto de N nodos con una carga total de L unidades distribuida en forma desbalanceada entre ellos, el funcionamiento de los algoritmos hará que la carga de cada procesador converga a L/N.

En esta sección, se analizará el algoritmo *Generalized Dimension Exchange* (GDE) [XL95] y se dará un bosquejo de la operación necesaria para demostrar estabilidad y convergencia. Considerar un grafo $G = (V, E)$ y se asignan a sus aristas k colores tal que cada par de aristas adjacentes tiene distinto color. Una *dimensión* corresponde a todas las aristas con el mismo color. El algoritmo GDE balancea la carga de una dimensión por iteración. Se define un grafo coloreado con k colores como $G_k = (V, E_k)$, tal que E_k es una tupla (i, j, c) que significa que la arista (i, j) tiene asignado el color c.

Se define w como la carga total de un procesador y λ como el parámetro de intercambio. Este parámetro se utiliza para intercambiar una cierta fracción de carga entre dos nodos de una misma dimensión. El rango de λ es $[0, 1]$. Dos temas deben ser abordados: (i) la condición de terminación del algoritmo; y (ii) eficiencia (número de pasos para alcanzar la carga promedio). El análisis es a través de un enfoque matricial iterativo.

En un enfoque matricial, la distribución de carga de los nodos se representa por un vector de carga. La política de balance es plasmada por una matriz, la que gobierna los

cambios de carga entre los nodos a través de sucesivas iteraciones. Sea w_i^t $(0 \leq i < n)$ la carga de trabajo del procesador i en el tiempo t. El vector de carga se denota $W^t = (w_0^t, w_1^t, ..., w_{n-1}^t)^T$, con W^0 la distribución de carga inicial.

Para aplicar el balance en una dimensión de color c, se tiene que $w_i^{t+1} = (1 - \lambda)w_i^t + \lambda w_j^t$ si $(i, j, c) \in E_k$, con $0 \leq \lambda \leq 1$. El cambio de carga en el sistema completo para una dimensión c es:

$$W^{t+1} = M_c(\lambda)W^t, 1 \leq c \leq k$$

Por lo que el cambio de carga en el sistema completo es (para todas las k dimensiones):

$$W^{t+k} = M(\lambda)W^t$$

Lo que implica:

$$W^{tk} = M^t(\lambda)W^0, t = 0, 1, 2, ...$$

A $M(\lambda) = M_k(\lambda) \times M_{k-1}(\lambda) \times ... \times M_1(\lambda)$ se le denomina matriz GDE del grafo G_k. Cabe recalcar que cada elemento de $M(\lambda)$ es un polinomio en λ.

Luego, el criterio de término del algoritmo se reduce a la convergencia de $\{M^t(\lambda)\}$ y la eficiencia se refleja en el radio de convergencia asintótico $R_\infty(M(\lambda))$.

Teorema (3.1 en [XL95]). Sea $M(\lambda)$ la matriz GDE de G_k, $\overline{M}$ una matriz de orden n cuyos elementos son todos iguales a $1/n$, y $\overline{W}$ un vector de distribución uniforme con todos los elementos iguales a 1. Entonces:

1. $lim_{t\to\infty} M^t(\lambda) = \overline{M}$ si y sólo si $\lambda \in (0, 1)$.

2. Para cualquier distribución inicial W^0 la secuencia $\{W^{tk}\}$ generada por el algoritmo GDE converge a $b\overline{W}$ si $\lambda \in (0, 1)$, donde $b = \sum_{0 \leq i < n}(w_i^0)/n$.

El teorema anterior dice que una condición necesaria y suficiente para la terminación del proceso de balance usando GDE es que $0 < \lambda < 1$. El teorema anterior es utilizado en múltiples trabajos para garantizar la convergencia de algoritmos de balance de carga.

El radio de convergencia asintótico $R_\infty(M(\lambda))$ tiene relación con la elección de un valor λ apropiado y de la estructura de $M(\lambda)$. La idea principal es hacer una selección de λ tal que $R_\infty(M(\lambda))$ sea máximo. Los autores en el trabajo determinan, a través del análisis de $M(\lambda)$, su radio espectral (máximo valor propio) y su valor propio subdominante, que para un conjunto de topologías de red (cadena, anillo, malla, entre otros) el valor del parámetro de intercambio óptimo λ_b es igual a $\frac{2-\sqrt{2(1-\cos(2\pi/n))}}{1+\cos(2\pi/n)}$.

Como se mencionó anteriormente, lo anterior es un bosquejo de los pasos para demostrar la convergencia y estabilidad del algoritmo GDE. Se recalca que cualquiera de los algoritmos descritos anteriormente, ostenta una demostración formal, ya sea basada en matrices u otra técnica, de la convergencia y estabilidad del proceso dado cualquier vector de carga inicial.

6.6. Anomalía

Este esquema de balance de carga es aplicable a otros ámbitos de balance de carga, pero en este contexto existe una anomalía que hace necesario un análisis más detallado.

Dada la utilización de Consistent Hashing para la localización de los ítemes, una consulta es asignada a un punto en el anillo. Suponer la situación de la Figura 24(a), en la cual la consulta q, que es frecuente y está asignada al nodo P_i, está cercana a un delimitador de rango del anillo (punto en el cual se separan las consultas que pertenecen a un nodo u otro). Esta situación continúa hasta que en un instante, se realiza corrección de los rangos para balancear la carga y el movimiento resultante es el expuesto en la Figura 24(b). Ahora toda la carga de trabajo generada por la consulta q pasa desde el nodo P_i al nodo P_{i+1}. Lo más probable es que, en una nueva ejecución del algoritmo de balance de carga, la situación resultante sea como la Figura 24(c). Esto implica que la carga de trabajo generada por la consulta q pasa nuevamente al nodo P_i, situación que puede reiterarse en futuras ejecuciones del algoritmo de balance, dado el impacto que tiene la consulta q en la carga absorbida por el nodo P_i.

La situación anómala descrita previamente, es sólo encontrada en situaciones en las cuales los ítemes no poseen la misma importancia o impacto. El nodo responsable

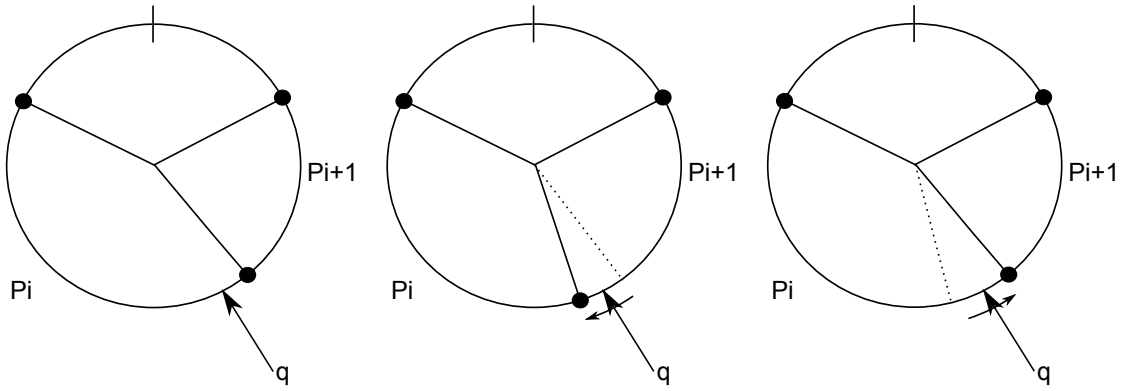


Figura 24: Anomalía de balance de carga.

por este tipo de ítemes tendrá una alta carga de trabajo, y mover estos ítemes a algún vecino tampoco es la solución, ya que se sobrecargará a éste último.

Esta es la razón por la que el mecanismo descrito en el capítulo 5 es tan importante: distribuyendo una fracción de las consultas más frecuentes en múltiples máquinas, hace que su impacto en una máquina en particular sea "dividida", y así el mecanismo de balance de carga propuesto en este capítulo no presenta la anomalía descrita. La estrategia de replicación es un mecanismo crucial para alcanzar un estado balanceado, pero el mecanismo de balance de carga es un refinamiento que sirve como una estrategia de balance de madiano plazo.

Es decir, los mecanismos de distribución de consultas frecuentes y el balance de carga dinámico deben trabajar en *tandem* con el objetivo de tener un Servicio de Cache balanceado.

Sin perjuicio de lo anterior, la nueva estrategia descrita en este capítulo sirve, por sí misma, como un mecanismo efectivo en el área de balance de carga dinámico para aplicaciones en que exista dependencia entre ítemes a asignar y los nodos.

6.7. Solución

La solución propuesta consiste en un mecanismo de balance de carga dinámico para servicios de cache que utilicen Consistent Hashing. Una cuestión importante

es cuándo activar el algoritmo de balance de carga, decisión que debe ser tomada considerando el estado del Servicio de Cache. Como se mencionó anteriormente, la información del estado de los nodos debe ser monitoreada y evaluada. En caso de ser necesario balancear la carga, ésta debe ser entregada al algoritmo de balance de carga en forma de entrada.

La evaluación del sistema toma lugar cada Δt unidades de tiempo. Este parámetro debe ser configurado de acuerdo a características del servicio, por ejemplo con la variación de la carga de trabajo del Servicio de Cache. Considerar unos pocos segundos para evaluar el desbalance del sistema, podría causar malas decisiones ya que una pequeña cantidad de información es analizada. Por otro lado, si se evalúa cada hora, la acciones correctivas a considerar podrían ser tardías. En general, sólo es necesario evaluar y corregir el balance cada pocos minutos. Otra opción posible, que no es considerada en este trabajo, es la definición del intervalo de tiempo en forma adaptiva, es decir, si existe poco desbalance el intervalo de medición crece, y si se detecta un alto grado de desbalance se reduce el intervalo de medición.

No siempre es necesario balancear el Servicio de Cache. Entonces se debe definir la condición bajo la cual ejecutar el algoritmo de balance de carga y aplicar los resultados de éste. En este trabajo se usa un umbral T_{load} que define el máximo grado de desbalance a ser tolerado en el sistema. Esto implica que si se cumple la relación $T_{load} > \bar{l}/max\{l_i\}$ (con $\bar{l}$ la carga promedio y l_i la carga de cada nodo, $1 \leq i \leq P$), entonces es necesario ejecutar el algoritmo con la carga del sistema como entrada. Este umbral puede ser fijo o variable. Se adopta un esquema fijo que simplifica la implementación del mecanismo. Este valor no puede ser muy restrictivo (por ejemplo un 99 %), ya que el algoritmo será ejecutado en cada evaluación, pero tampoco muy relajado (50 %) para dar espacio al balance. En las evaluaciones, los valores típicos son en un rango de $80\,\% - 95\,\%$. El mecanismo descrito se encuentra en el Algoritmo 3.

No hay una metodología para establecer un valor genérico de los dos parámetros descritos anteriormente. Más bien, estos parámetros deben ser configurados dependiendo de la aplicación a estudiar y realizando pruebas para obtener los valores óptimos para la aplicación. Sin embargo, los pasos seguidos en este trabajo van en esa

línea y podrían servir de guía del proceso de determinación de los parámetros en otros contextos.

Algoritmo 3 Balance. Mecanismo propuesto para balance de carga a ejecutarse cada Δt unidades de tiempo.

Input: T_{load} cota mínima de eficiencia, l vector de carga.

```
1: ef ← l̄/max{l_i}
2: if ef < T_load then
3:     M ← LoadBalancing( l )                    ▷ Conjunto de movimientos
4:     for each m ∈ M do
5:         Aplicar movimiento m al anillo Consistent Hashing
6:     end for
7: end if
```

6.8. Evaluación

Esta sección se dividirá en dos partes: (i) determinación del algoritmo apropiado junto con sus parámetros, y (ii) evaluación experimental en que se estudiarán distintas configuraciones para analizar el desempeño del mecanismo propuesto.

6.8.1. Algoritmos y Parámetros

Los algoritmos evaluados son *Naive* (Consistent Hashing con puntos fijos), *Dimension Exchange Method* (DEM) [Cyb89], *Generalized Dimension Exchange* (GDE) [XL95], *Gradient Model* (GM) [LK87], *Receiver Initiated Diffusion* (RID) [WLR93], *Sender Initiated Diffusion* (SID) [WLR93] y *Direct Method* (DM) [WS97].

Las métricas a evaluar de los algoritmos son:

- Utilización máxima y mínima de los nodos del servicio.
- Tasa de *hit*.
- Porcentaje de todos los movimientos, como un porcentaje sobre el rango total del anillo.

Para la evaluación de los algoritmos, se utilizó un *log* de 500 millones de consultas pertenecientes a los primeros 7 días de Mayo de 2011. Para dejar atrás efectos de calentamiento de cache y para lograr resultados en régimen, las mediciones comienzan una vez completadas 100 millones de consultas. La Tabla 9 muestra los resultados de la evaluación empírica. La definición de los parámetros de los algoritmos sigue las directrices mencionadas en cada uno de los artículos que definen los algoritmos (generalmente dependen del número de nodos). Varias conclusiones acerca del desempeño de los algoritmos son extraídas:

1. Un movimiento exagerado de los rangos no lleva al mejor desempeño. Por ejemplo, considerar el algoritmo DM, el cual mueve 23,22 % del anillo a través de la simulación. Esto es debido a la operación del algoritmo DM, el cual en un solo paso calcula la secuencia completa de movimientos para alcanzar un estado balanceado y estos pasos son aplicados de una sola vez. Algoritmos que realizan los movimientos iterativamente presentan mejor desempeño que DM.

2. Los algoritmos que realizan pocos movimientos (DEM y GDE) no alcanzan los mejores resultados.

3. Mientras más pequeña sea la diferencia entre la utilización máxima y mínima, mejor el desempeño, como por ejemplo el algoritmo RID. Esto implica un servicio mejor balanceado.

4. Casi todos los algoritmos preservan la tasa de *hit* a pesar del movimiento de los rangos (excepto el algoritmo DM debido al punto 1). Este último hecho es una de las principales razones de utilizar balance dinámico como se propone en este trabajo: variaciones pequeñas y controladas en el rango de cada nodo no impactan significativamente en la tasa de *hit*.

Para validar los resultados anteriores, en la Tabla 10 se muestra el resultado de la ejecución de un *log* de 200 millones de consultas desde el 1 al 3 de Enero de 2012 en un Servicio de Cache compuesto de 20 particiones con 10 mil entradas cada nodo. Se observa nuevamente que los algoritmos con mejores resultados son del tipo Difusión (SID y RID), el método directo DM tiene un mal desempeño debido al alto porcentaje

Tabla 9: Evaluación de los algoritmos de balance de carga. El sistema está compuesto de 20 particiones. $T_{load} = 95\,\%$ y el número de entradas de cache es un millón por nodo (500 millones de consultas del 1 al 7 de Mayo de 2011).

Algoritmo	Carga Máxima	Carga Mínima	Tasa de *Hit*	Movimientos
Naive	85,86 %	53,13 %	59,56 %	N/A
DEM	72,98 %	55,03 %	59,53 %	0,11 %
GDE	72,15 %	55,59 %	59,55 %	0,11 %
GM	66,70 %	58,10 %	59,36 %	2,40 %
RID	63,58 %	61,02 %	59,30 %	4,77 %
SID	65,33 %	59,94 %	59,20 %	11,55 %
DM	64,81 %	59,66 %	56,92 %	23,22 %

de movimientos, y los algoritmos que realizan pocos movimientos no tienen buenos resultados (DEM y GDE). Si se compara la Tabla 9 y 10, se observan diferencias en las distintas características expuestas. Esto es debido principalmente a las características del *log* de consultas utilizado y a la definición de los parámetros de simulación.

Dados los resultados de la Tabla 9 y 10, el algoritmo que se escoge es *Receiver Initiated Diffusion* (RID) [WLR93]. A pesar de que existen algoritmos que balancean la carga en un solo paso, RID presenta un mejor desempeño. RID inicia el balance de carga cuando un nodo detecta un desbalance entre él y sus vecinos (algoritmo distribuido que se adapta a un ambiente centralizado). Se piensa que este enfoque es mejor y tiene menos impacto en el sentido de que algunas veces no es necesario mover los rangos de todos los nodos, sino que pequeños movimientos son suficientes para alcanzar el umbral de eficiencia pre-definido. También, se debe recordar que en este trabajo se permite cierto nivel de desbalance para no ejecutar constantemente el algoritmo de balance de carga. Si se compara el algoritmo RID con el enfoque *Naive* se tiene que RID reduce efectivamente la utilización máxima en 25,95 %. Otro punto importante es notar que la carga de trabajo de los algoritmos implementados tiende a la carga promedio (en ningún caso diverge).

Existen varios puntos adicionales a considerar con respecto a los resultados obtenidos. Como fue mencionado en el capítulo 5, la utilización eficiente de las entradas del esquema básico genera desbalance. Al realizar movimientos de rangos, se tiene

Tabla 10: Evaluación de los algoritmos de balance de carga. El sistema está compuesto de 20 particiones. $T_{load} = 95\,\%$ y el número de entradas de cache es 10 mil por nodo (200 millones de consultas del 1 al 3 de Enero de 2012).

Algoritmo	Carga Máxima	Carga Mínima	Tasa de *Hit*	Movimientos
Naive	85,35 %	72,64 %	40,86 %	N/A
DEM	84,98 %	74,12 %	40,80 %	0,10 %
GDE	85,43 %	73,64 %	40,85 %	0,10 %
GM	85,12 %	78,25 %	40,15 %	0,95 %
RID	80,40 %	76,03 %	40,00 %	1,52 %
SID	82,46 %	75,57 %	39,94 %	1,61 %
DM	85,31 %	70,56 %	26,74 %	8,58 %

una pérdida inevitable en la tasa de *hit* y probablemente del tiempo de respuesta promedio. Lo ideal es que los movimientos sean iterativos y acotados, y también que la suma de los movimientos al terminar la ejecución sea reducida. Con movimientos reducidos se logra una alta tasa de *hit* mientras que se intenta balancear la carga de trabajo. Entonces existe nuevamente un compromiso entre la reducción de la tasa de *hit* y el total de movimiento de rangos. Esto está presente en la Tabla 9 si se consideran las estrategias RID (reducción acotada y movimientos acotados) y DM (alto movimiento de rangos y reducción drástica de la tasa de *hit*).

Se concluye que la pérdida acotada en la tasa de *hit* beneficia al servicio en términos de balance, lo que es observado en la mayoría de los algoritmos. Pero existe una arista más importante que no ha sido considerada: la capacidad de respuesta del servicio ante los fallos. Todas las estrategias estudiadas en el capítulo 5 son estáticas, por lo que la caída de nodos implicaría un empeoramiento del desempeño del servicio. Esto también se probará: aplicando el algoritmo de balance de carga dinámico propuesto es posible mitigar los efectos de los fallos, con la finalidad de llegar a un servicio balanceado luego de un período de tiempo. Esta arista debe ser analizada, ya que como es mencionado en [LM10], los servidores en aplicaciones de gran escala fallan constantemente.

El algoritmo RID es dependiente del parámetro $L_{threshold}$. Aparte del valor de este parámetro definido en [WLR93], se analizará su comportamiento para distintos

Tabla 11: Tiempo de respuesta promedio / Cuantil 0,90 [ms].

$L_{threshold}$	T_{load}		
	0,90	0,95	0,99
0,01	31,5 / 56,9	37,3 / 57,1	38,2 / 57,2
0,02	30,8 / 56,8	36,3 / 57,1	36,8 / 57,1
0,04	30,2 / 56,7	32,2 / 56,9	32,4 / 56,9
0,06	29,6 / 56,7	29,6 / 56,7	29,9 / 56,8
0,08	29,7 / 56,7	29,7 / 56,7	29,6 / 56,7
0,10	29,7 / 56,7	29,7 / 56,8	29,7 / 56,9
0,12	29,6 / 56,8	29,6 / 56,8	29,6 / 56,8
0,14	29,5 / 56,8	29,9 / 56,8	29,6 / 56,8
0,16	29,4 / 56,8	29,4 / 56,8	29,4 / 56,8

Tabla 12: Tasa de *hit*.

$L_{threshold}$	T_{load}		
	0,90	0,95	0,99
0,01	49,3	37,6	35,7
0,02	50,6	39,6	38,5
0,04	51,9	47,9	47,5
0,06	53,0	53,0	52,6
0,08	52,9	52,8	53,0
0,10	53,1	53,0	53,0
0,12	53,1	53,2	53,2
0,14	53,3	53,3	53,3
0,16	53,6	53,6	53,6

Tabla 13: Carga de trabajo mínima.

$L_{threshold}$	T_{load}		
	0,90	0,95	0,99
0,01	76,3	77,5	77,3
0,02	76,1	77,5	77,4
0,04	76,4	77,6	77,6
0,06	76,2	76,9	77,2
0,08	75,5	76,5	75,0
0,10	73,7	72,5	72,5
0,12	72,1	72,1	72,1
0,14	72,6	72,5	72,5
0,16	70,9	70,9	70,9

Tabla 14: Carga de trabajo máxima.

$L_{threshold}$	T_{load}		
	0,90	0,95	0,99
0,01	79,5	78,9	78,7
0,02	79,2	78,6	78,6
0,04	79,8	78,5	78,3
0,06	80,3	79,0	79,0
0,08	81,0	79,8	80,2
0,10	85,6	82,4	82,4
0,12	84,7	84,7	84,7
0,14	83,9	84,3	84,3
0,16	87,9	87,9	87,9

niveles de T_{load} para ver el impacto en conjunto. De esta forma se obtendrá la mejor configuración para la sección experimental.

En la Tabla 11 se expone el tiempo de respuesta promedio para la ejecución de 100 millones de consultas de Enero de 2012 en un servicio compuesto de 20 nodos y 100 mil entradas por nodo. El parámetro T_{load} indica el umbral de eficiencia tolerado bajo el cual se gatilla el algoritmo de balance de carga. Lo primero que se observa es que entre más exigente se torna la evaluación (T_{load} más alto), menos beneficios se tienen. Esto es debido a que, probablemente, en cada evaluación de la carga de

trabajo se gatilla el algoritmo de balance de carga, haciendo que se muevan constantemente secciones del anillo según la salida del algoritmo RID. Esto también afecta la tasa de *hit* presente en la Tabla 12. En contraste a este empeoramiento, se encuentra la utilización máxima y mínima experimentada (Tabla 13 y 14, respectivamente), la cual muestra que entre más exigentes las condiciones de balance, mejor es el desempeño observado. Como se ha mencionado, existe un compromiso entre estas medidas. El parámetro T_{load} es un parámetro global, que considera todos los nodos, mientras $L_{threshold}$ es un parámetro de desbalance a nivel de vecindad entre nodos. Si el desbalance de un nodo en comparación con sus nodos vecinos es menor que $L_{threshold}$, entonces se comparte carga entre el nodo y sus vecinos (los detalles del algoritmo se encuentran en [WLR93]). Entonces una condición más exigente ($L_{threshold}$ más bajo), implica que se realiza más frecuentemente balance entre vecinos, lo que conlleva a una pérdida en la tasa de *hit* y un aumento en el tiempo de respuesta.

Considerando las Tablas 11 y 12, se observa que el umbral para la eficiencia T_{load} debe encontrarse entre 0,90 y 0,95. Si se considera que 29,4 [ms] es el tiempo promedio mínimo y 53,6 % la máxima tasa de *hit*, se observa que un $L_{threshold}$ entre 0,06 y 0,12 generan un deterioro muy acotado (un 1 % para el tiempo promedio y un 1 % para la tasa de *hit*). Esto tiene como resultado que los resultados obtenidos para la utilización mínima y máxima sean mejores (más ajustados), como es expuesto en las Tablas 13 y 14. Según los resultados anteriores, se concluye que la estrategia es factible de implementar y que cumple el objetivo del mecanismo propuesto que es balancear la carga.

6.8.2. Evaluación Experimental

Con respecto a la complejidad computacional, las principales operaciones de la estrategia propuesta tienen que ver con el establecimiento de la eficiencia del servicio, con la ejecución del algoritmo de balance de carga y con la aplicación de los movimientos al anillo de Consistent Hashing. Estas operaciones son aplicadas como máximo a todos los nodos del servicio, por lo que la solución propuesta incurre en un costo de $O(P)$ cada Δt unidades de tiempo (incluido el algoritmo *RID*).

La estrategia de balance dinámico de carga se utilizará en conjunto con la estrategia propuesta en el capítulo 5 para distribuir las consultas más frecuentes. Lo primero que se realizará será analizar cuál es la ganancia, en términos de desempeño, al agregar el mecanismo de balance de carga dinámico. Para ello, se configura un Servicio de Cache compuesto por 30 nodos. En la Figura 25(a) se observa la utilización del esquema de distribución de consultas frecuentes. Se observa que esta acción balancea efectivamente la carga de trabajo de los nodos, pero estos resultados mejoran al aplicar balance de carga dinámico (Figura 25(b)). En la misma figura, se observa la comparación en términos de eficiencia de ambas estrategias (Figuras 25(c) y (d)). En promedio, la eficiencia sin balance de carga dinámico es 86,1 % versus un 88,6 % al aplicar este método.

Con respecto a la tasa de *hit*, y como fue analizado anteriormente, existe una reducción desde un 53,1 % a un 52,3 % (habilitando el balance de carga dinámico). También existe un aumento del tiempo de respuesta promedio desde 27,9 a 28,3 milisegundos. Esto es debido al movimiento de los rangos en el anillo de Consistent Hashing. Esto nuevamente comprueba la relación que existe entre tasa de *hit*, balance de carga y tiempo de respuesta promedio.

Las fallas en los nodos podrían cambiar la situación descrita anteriormente, por lo que el siguiente caso analizado corresponde a la misma configuración anterior bajo la inserción de fallas en 5 nodos seleccionados aleatoriamente. Estas 5 fallas suceden en distintos nodos de los 30 disponibles, es decir, equivale a un 16,7 % del servicio. Estas fallas son simultáneas y son insertadas en el valor $x = 2$ (el minuto entre el segundo y tercer día considerando el *log* de consulta utilizado).

En la Figura 26 se observa la comparación de las dos estrategias propuestas ante fallos. Se observa en la Figura 26(a) que la distribución de consultas frecuentes realiza una corrección de la carga de trabajo, pero es de largo plazo como fue analizado. Existe un conjunto de nodos que se satura en el instante en que se insertan las fallas, pero no logra ser balanceado en forma rápida y acorde a la carga de trabajo. Por el contrario, la carga de estos nodos es corregida lentamente, lo cual impacta significativamente en las métricas analizadas. En la Figura 26(c) se evidencia el empeoramiento de la eficiencia del servicio al insertar fallas, que baja desde un 86,8 % a un 72,3 % (en

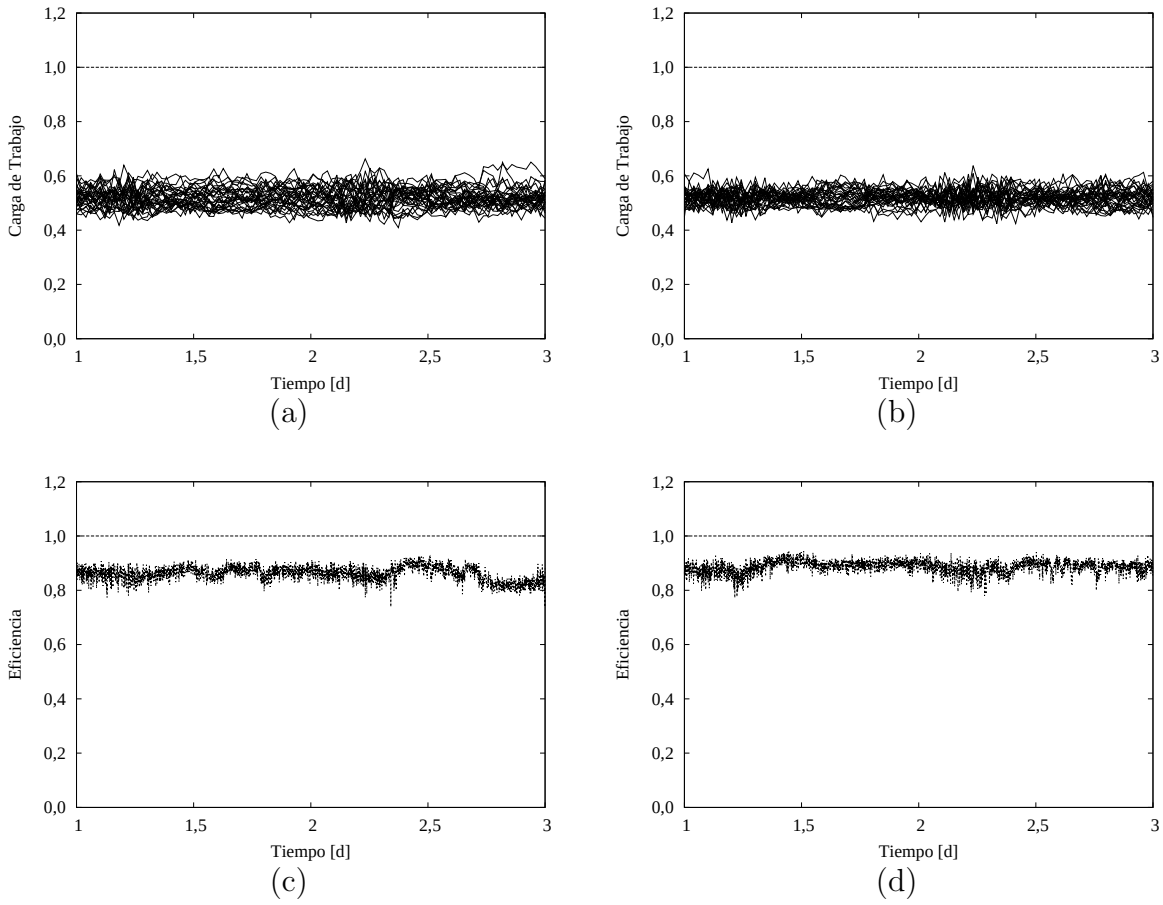


Figura 25: Resultados para el 2 y 3 de Enero de 2013 para un Servicio de Cache de 30 nodos y 100 mil entradas por nodo: (a) y (c) sin balance dinámico (propuesta capítulo 5); (b) y (d) con balance de carga dinámico.

promedio). La saturación experimentada es de 4 nodos, y la tasa de *hit* es de un 50,1 %. El tiempo de respuesta promedio sube 27,9 a 31,9 milisegundos.

Por otro lado, se tiene que la configuración de distribución de consultas frecuentes es mejorada con la aplicación de algoritmos de balance de carga dinámico. Esto es evidenciado por la utilización (Figura 26(b)) y la eficiencia (Figura 26(d)). Primero que todo, en el momento de las fallas se ve un alza en la utilización y un desbalance abrupto, llevando a algunos nodos a saturación (2 nodos específicamente). Esto es solucionado rápidamente por el balance dinámico, alcanzando en el corto plazo una

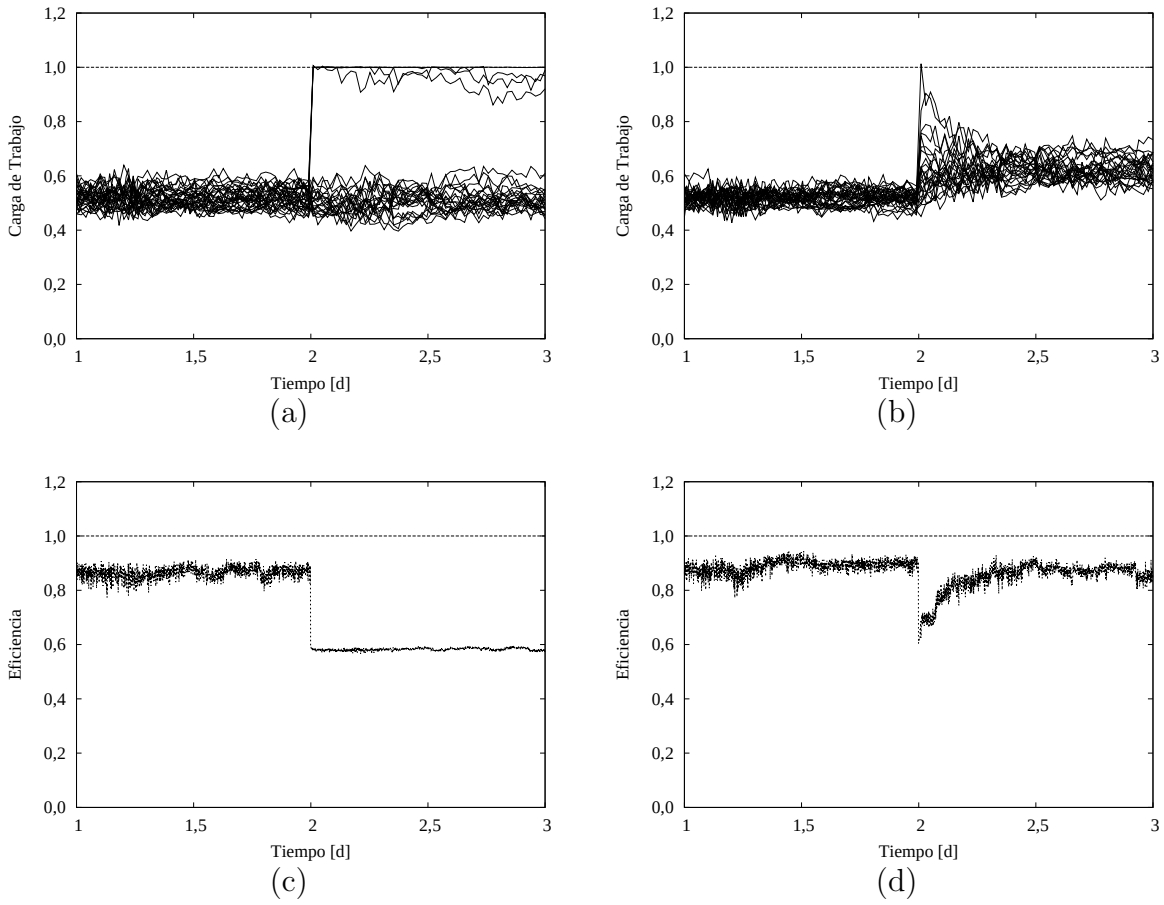


Figura 26: Resultados para el 2 y 3 de Enero de 2013 para un Servicio de Cache de 30 nodos, 100 mil entradas por nodo y 5 fallas simultáneas ($x = 2$): (a) y (c) sin balance dinámico (propuesta capítulo 5); (b) y (d) con balance de carga dinámico.

situación estable. En números, el tiempo promedio sube de 28,3 a 29,6 milisegundos, la eficiencia promedio es de un 86,8 % (comparado a un 88,6 % sin fallas), y la tasa de *hit* es un 49,1 %.

En la Tabla 15 y 16 se resumen los resultados de la estrategia de distribución de consultas más frecuentes con y sin aplicación del balance de carga. También se exponen los resultados con y sin fallas. La idea de estas tablas es comparar el deterioro parcial impuesto por el mecanismo de balance de carga dinámico (sin y con fallas), y también comparar la diferencia de cada estrategia de una situación sin fallas a una con

Tabla 15: Resultados sin fallas (D: distribución de ítemes frecuentes sin balance, $D + B$: distribución de ítemes frecuentes con balance).

	D	D+B	Diferencia
T. de respuesta promedio / Cuantil 0,90 ([ms])	27,9 / 55,9	28,3 / 55,4	1,4 % / 0,9 %
Tasa de *Hit* (%)	53,0	52,3	1,3 %
Saturación (Nodos)	0	0	0 %
Eficiencia Promedio (%)	86,1	88,6	2,9 %

Tabla 16: Resultados con fallas (D: distribución de ítemes frecuentes sin balance, $D + B$: distribución de ítemes frecuentes con balance).

	D	D+B	Diferencia
T. de respuesta promedio / Cuantil 0,90 ([ms])	31,9 / 56,3	29,6 / 55,5	7,2 % / 1,4 %
Tasa de *Hit* (%)	50,9	49,1	3,5 %
Saturación (Nodos)	4	2	50 %
Eficiencia Promedio (%)	72,4	86,8	19,9 %

fallas. El resultado más importante corresponde a que el funcionamiento en *tandem* de las dos estrategias propuestas en este trabajo, hace que en una configuración sin fallas con un tiempo de respuesta promedio de 28,3 milisegundos se suba a 29,6 milisegundos con la falla simultánea del 16,7 % de nodos del servicio. Esto es un incremento en tiempo de 4,6 % (1,3 milisegundos). Del análisis anterior, se concluye que la propuesta de balance de carga dinámico contribuye a mitigar los efectos de la caída de nodos, y además tiende a balancear la carga tal como es el objetivo de los algoritmos estudiados. De aquí en adelante, sólo se utilizará la versión que incluye las dos estrategias propuestas para realizar comparaciones.

El siguiente paso es la comparación con las estrategias base (Consistent Hashing y Matricial). Para esto, se ejecutará la misma configuración anterior: 30 nodos, 100 mil entradas por nodo y los días 2 y 3 de Enero de 2013. Como se tienen 30 nodos, las configuraciones en el esquema de matriz son: 15, 10, 6 y 5 particiones (siempre manteniendo los 30 nodos). La estrategia propuesta sigue los mismos parámetros definidos en las secciones anteriores.

En la Figura 27(a) se expone la eficiencia de las distintas estrategias, donde destaca el esquema de 5 particiones (PD5) y la estrategia propuesta (PR), con un 92,7 % y 88,6 % de eficiencia respectivamente. Se debe recordar que a medida que se aumenta

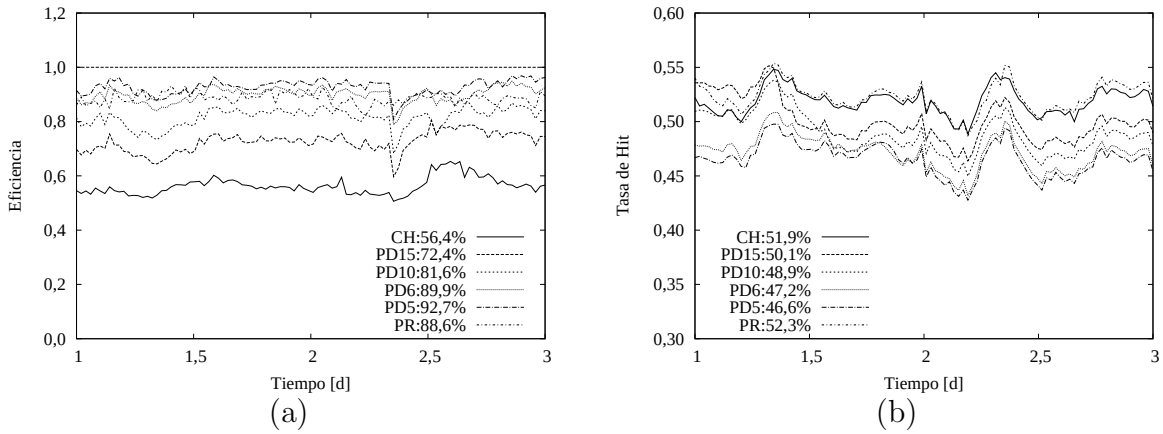


Figura 27: Resultados para el 2 y 3 de Enero de 2013 para un Servicio de Cache de 30 nodos y 100 mil entradas por nodo (sin fallas): (a) eficiencia; (b) tasa de *hit*.

el número de particiones, se tiene una menor eficiencia. Por otro lado, en la Figura 27(b) se tiene que la tasa de *hit* de estas dos mismas estrategias es de 46,6 % y 52,3 % respectivamente.

En la Tabla 17 se observa un resumen de los resultados obtenidos para la configuración de la Figura 27, que permite obtener diversas conclusiones. Primero que todo, el impacto de las fallas es variado, siendo las estrategias que tienen un esquema matricial las menos afectadas en aspectos como tasa de *hit* y tiempo promedio. Por ejemplo, PD10, PD6 y PD5 tienen variaciones mínimas en estas métricas. Esto es principalmente debido a la replicación de particiones, que permite tener información en múltiples nodos y también a la capacidad de distribuir una instancia de una consulta en un conjunto de nodos. Así, al fallar uno de los nodos de una partición, existe un conjunto de nodos que puede seguir respondiendo las peticiones asociadas a esa partición. La replicación es a la vez una debilidad, ya que disminuye el número total de entradas distintas para cache. Como en situaciones anteriores, la tasa de *hit* en esquemas sin replicación es alta (CH y PR), pero no implica el mejor desempeño, excepto la estrategia propuesta.

Sin fallas, se observa un comportamiento similar al observado en las secciones anteriores: (i) una alta tasa de *hit* en CH y PR; (ii) un buen balance en las estrategias

Tabla 17: Resultados de Servicio de Cache de 30 nodos y 100 mil entradas por nodo (2 y 3 de Enero de 2013). SF: sin fallas, CF: con 5 fallas.

Estrategia	Saturación (# nodos)		Eficiencia (%)		Tasa de *Hit* (%)		Tiempo de Respuesta Promedio ([ms])	
	SF	CF	SF	CF	SF	CF	SF	CF
CH	2	5	56,4	56,9	51,9	50,1	30,3	53,2
PD15	0	4	72,4	69,2	50,1	50,0	29,6	44,4
PD10	0	1	81,6	72,5	48,9	47,9	30,5	31,9
PD6	0	1	89,9	80,3	47,2	47,2	31,6	31,8
PD5	0	0	92,7	87,3	46,6	46,6	32,0	32,0
PR	0	2	88,6	86,8	52,3	49,1	28,3	29,6

matriciales y PR; (iii) inexistencia de saturación excepto en CH. Considerando el caso con fallas, la estrategia CH sube su tiempo de respuesta promedio de 30,3 a 53,2 milisegundos (un 75,6 %). Por otro lado, la estrategia propuesta sólo muestra un incremento de un 4,6 %, lo cual es un aumento marginal si se considera la magnitud de las fallas (5 fallas simultáneas en un servicio de 30 nodos). La estrategia con segundo mejor tiempo de respuesta muestra un incremento de 31,6 a 31,8 milisegundos (un 0,6 %). Aún así, la estrategia propuesta muestra una reducción del tiempo promedio de un 6,9 % (comparada con PD6). Finalmente, en la mayoría de los casos existe saturación de nodos, pero su impacto es acotado en las estrategias matriciales y la propuesta.

En la Figura 28(a) se observa el desempeño de las estrategias, en términos de eficiencia, de lo cual se obtuvieron los resultados expuestos en la Tabla 17 (inserción de 5 fallas en forma simultánea). Se distingue claramente el comportamiento de los esquemas rígidos (CH y PD), en que en el momento que sucede una falla, estas estrategias conservan el desempeño y no realizan acciones correctivas para mejorar esta situación. Por el contrario, el mecanismo de balance de carga dinámico propuesto permite corregir las consecuencias de las fallas. Esta corrección no es instantánea, pero a través del tiempo se observa una tendencia que permite tener mejor desempeño que las estrategias base. Por otro lado, en la Figura 28(b) se observa la eficiencia de las estrategias ante la inserción de 10 fallas en forma simultánea (un 33,3 % de los nodos del servicio fallan). En esta figura se observa el mismo patrón anterior, con la diferencia

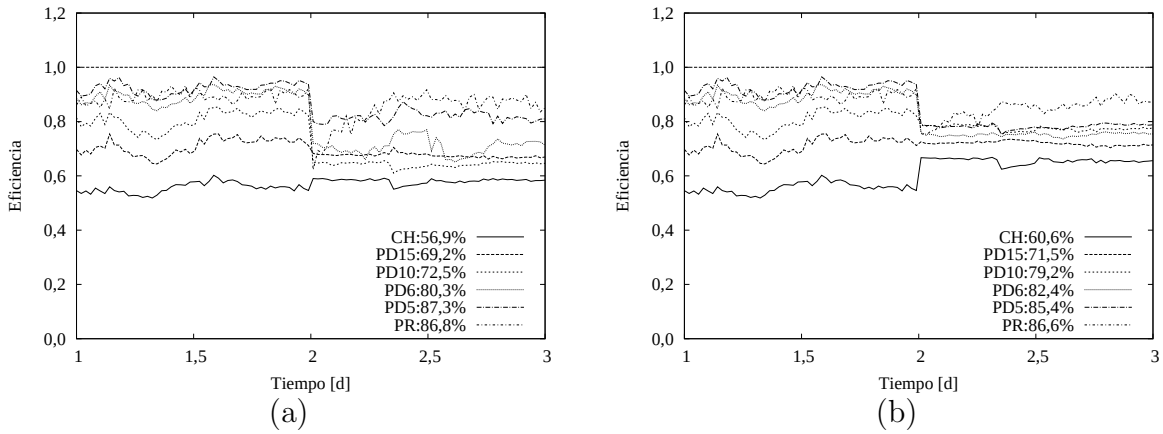


Figura 28: Eficiencia para el 2 y 3 de Enero de 2013 para un Servicio de Cache de 30 nodos y 100 mil entradas por nodo: (a) 5 fallas simultáneas en $x = 2$; (b) 10 fallas simultáneas en $x = 2$.

que la estrategia propuesta tiene un estabilización más lenta. Como antecedente final, en la Figura 29 se tiene carga de trabajo de las distintas particiones de las estrategias analizadas (CH, PD15, PD10, PD6, PD5 y PR). Esto permite ver la magnitud de los fallos en las particiones, y también permite ver la corrección de la carga de trabajo en la estrategia propuesta.

Los experimentos anteriores fueron para una configuración de 30 nodos. A continuación, se variarán tanto el modo de inserción de fallos como el número de nodos para verificar la validez de la estrategia propuesta. En la Figura 30 se exponen los resultados de la estrategia propuesta cuando las fallas son insertadas en distintos instantes de tiempo. Específicamente, el servicio se compone de 30 nodos, y se insertan 3 fallas aproximadamente en $x = 1$ y $x = 2$. La utilización de los nodos y la eficiencia muestran que la estrategia propuesta permite balancear la carga dinámicamente, en función de la carga de trabajo, y por lo tanto se impide que los nodos se saturen.

En la Figura 31 se observa la eficiencia de un servicio compuesto por 80 nodos y 100 mil entradas por nodo. Se observa que en el caso sin fallas (a) existe un buen balance, pero con algunas variaciones. Como fue mencionado en el análisis de los algoritmos, a medida que se incrementa el número de nodos, más demora la estrategia en converger a la carga promedio. En la Figura 31(b) se observa la eficiencia para la configuración

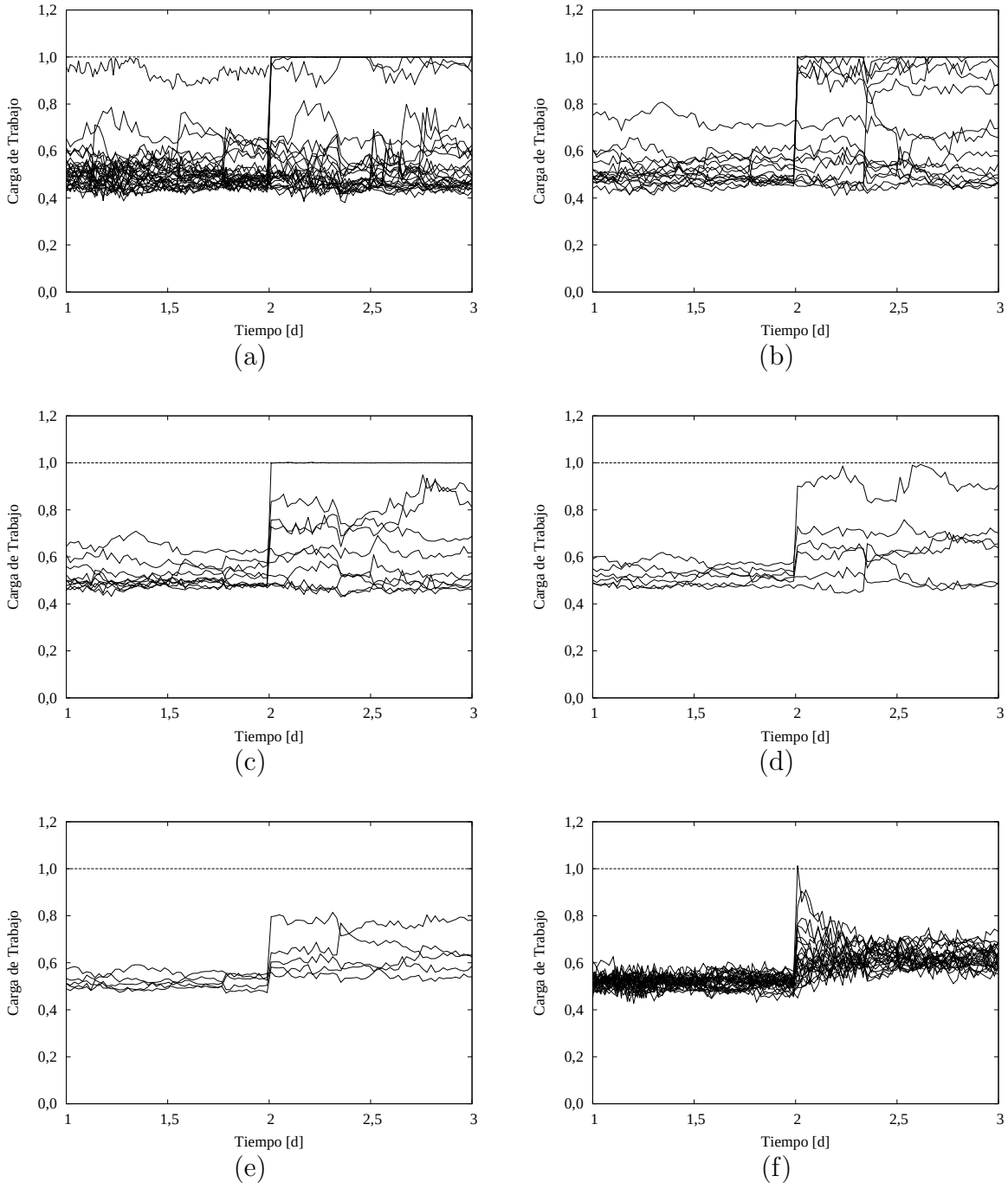


Figura 29: Resultados para el 2 y 3 de Enero de 2013 para un Servicio de Cache de 30 nodos, 100 mil entradas por nodo y 5 fallas simultáneas en $x = 2$: (a) CH; (b) PD15; (c) PD10; (d) PD6; (e) PD5; (f) PR.

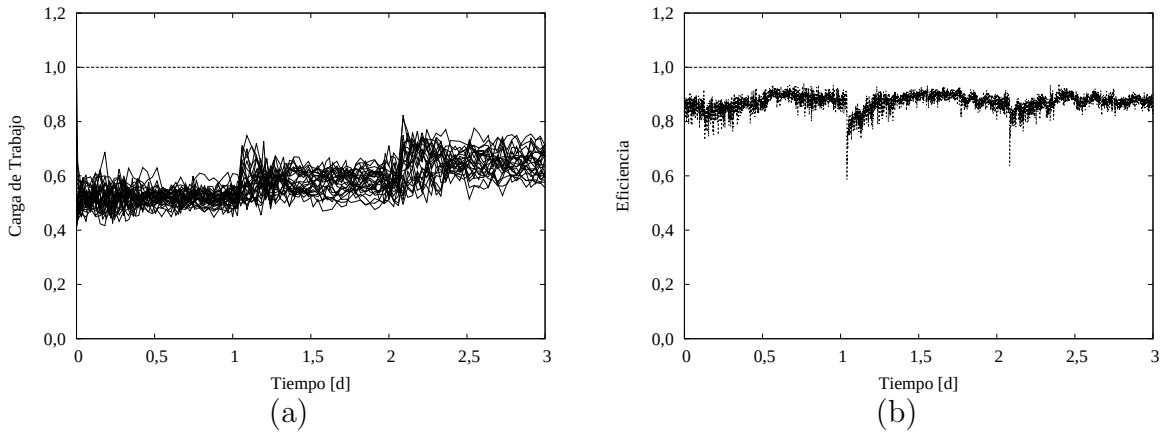


Figura 30: Resultados de la propuesta para el 1, 2 y 3 de Enero de 2013 para un Servicio de Cache de 30 nodos y 100 mil entradas por nodo: (a) utilización; (b) eficiencia. Se insertan 3 fallas simultáneas en $x = 1$ y $x = 2$ aproximadamente.

anterior, pero con la inserción de 10 fallas en forma simultánea. Nuevamente se observa que la estrategia propuesta alcanza buen desempeño en términos de balance de carga.

Como experimento final y para completar la validación del modelo de balance de carga, en la Figura 32 se muestran los resultados para la inserción de nodos en un Servicio de Cache compuesto inicialmente por 20 nodos. Esta característica de los mecanismos de balance de carga es muy importante, al igual que el estudio de las situaciones en que nodos salen del servicio (caen), ya que permiten tener una idea de qué tan adaptable es el mecanismo propuesto a las variaciones del entorno (conjunto de nodos). Para la inserción de un nodo se selecciona uno en forma aleatoria y su rango se divide en dos partes iguales: un subrango sigue perteneciendo al nodo y el otro subrango es asignado al nuevo nodo. Además, cuando un nodo entra en servicio lo hace sin entradas pre-computadas en su memoria, por lo que para alcanzar una tasa de *hit* comparable a la de los demás nodos del servicio, debe haber un período de calentamiento. Según los experimentos realizados, este período tiene una duración de entre 2 y 3 horas (para 100 mil entradas).

En la Figura 32(a) se observa que el balance dinámico permite mitigar el efecto de la inserción de 5 nodos en forma simultánea (aproximadamente en $x = 1$), haciendo que la carga de trabajo sea distribuida en forma homogénea. Naturalmente, a medida

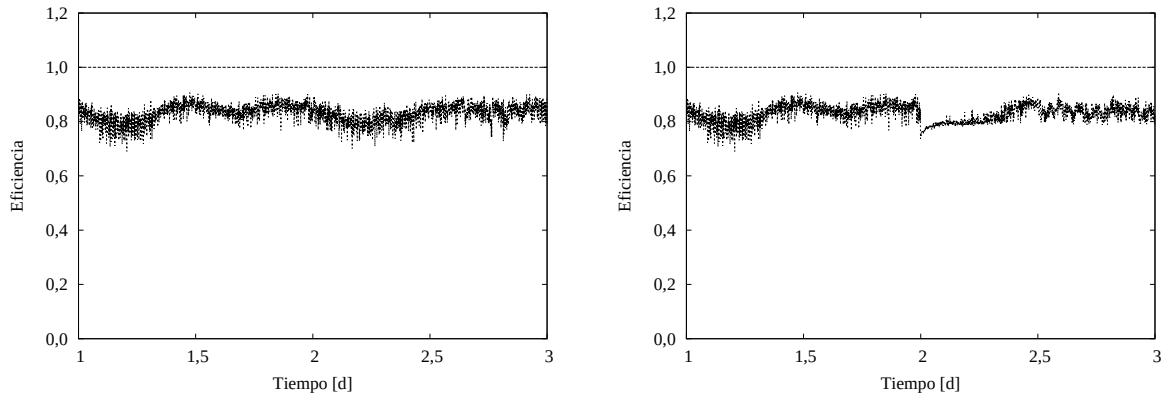


Figura 31: Eficiencia de la propuesta para el 2 y 3 de Enero de 2013 para un Servicio de Cache de 80 nodos y 100 mil entradas por nodo: (a) sin fallas; (b) inserción de 10 fallas simultáneas en $x = 2$.

que se insertan más nodos, baja en parte la utilización de los nodos del servicio. En un contexto similar, en la Figura 32(b) se observan los resultados para la inserción de 3 nodos en dos instantes distintos (aproximadamente en $x = 1$ y $x = 3$). Nuevamente, en este contexto se observa que la propuesta balancea la carga efectivamente. Esto refuerza la idea de que el mecanismo se adapta rápidamente a los cambios en el servicio, y que dado cualquier contexto de inserción de nodos, el servicio alcanza un estado de balance en un intervalo de tiempo limitado.

6.9. Conclusiones

En este capítulo se ha propuesto un mecanismo de balance de carga dinámico para servicios de cache basados en *Consistent Hashing*, que funciona de manera conjunta con la propuesta de distribución de consultas frecuentes descrita en el capítulo 5. El funcionamiento conjunto de las estrategias propuestas implica la reducción tanto del desbalance estructural como del desbalance dinámico, alcanzando un buen desempeño en distintas situaciones previstas en servicios de cache. Aún así, existen episodios puntuales que no son cubiertos por ambas estrategias y que podrían generar desbalance. Estos episodios serán abordados en el siguiente capítulo.

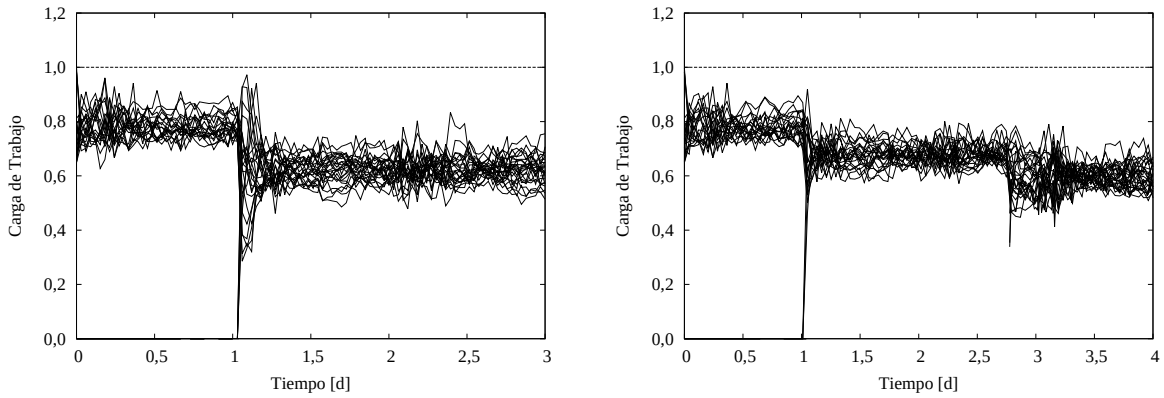


Figura 32: Carga de trabajo de la propuesta para un Servicio de Cache de 20 nodos y 100 mil entradas por nodo: (a) se insertan 5 nodos simultáneamente en $x = 1$ aproximadamente (1 al 3 de Enero de 2013); (b) se insertan 3 nodos en $x = 1$ y 3 nodos en $x = 3$ aproximadamente (1 al 4 de Enero de 2013).

La estrategia propuesta necesita como entrada la utilización de cada nodo del servicio cada Δt unidades de tiempo. Luego, la eficiencia se obtiene a partir del estado general del servicio. Si la eficiencia está bajo el umbral T_{load}, entonces se ejecuta el algoritmo RID que recibe como entrada los nodos y su utilización. Este algoritmo determina un conjunto de movimientos en el anillo, el cual tiende a mejorar el balance del servicio. Se debe considerar que el algoritmo RID garantiza convergencia y estabilidad del proceso, y se necesita más de una invocación del algoritmo para alcanzar un estado balanceado. Finalmente, estos movimientos son aplicados en el anillo de Consistent Hashing para ser considerados en el ruteo de consultas.

Los movimientos de rangos entre particiones son:

- Libres: ya que no siempre se balancea con ambos vecinos a la vez, y además una partición que alguna vez recibió más carga de trabajo podría en otro movimiento delegar carga de trabajo a otra partición.

- Bidireccionales: ya que los movimientos de los rangos suelen ocurrir en ambos sentidos.

El esquema de balance de carga propuesto permite solucionar varios problemas:

- Distribución de carga sólo entre vecinos, que se torna un requisito importante para la mantención de una alta tasa de *hit*. La propuesta no incluye el movimiento de rangos a otras secciones del anillo, ya que así se conserva el costo $O(\log n)$ para la búsqueda de un ítem.

- Variaciones dinámicas de carga en los nodos, que permiten ajustar el desempeño del Servicio de Cache a la variabilidad en el tráfico de consultas.

- Balance considerando dependencia de datos, ya que la asignación de una consulta a un nodo no es libre (Consistent Hashing).

En los problemas de balance de carga, en general la idea es converger a la carga de trabajo promedio en todos los nodos. Para esto, se utilizan algoritmos de balance de carga sujetos a la topología anillo y el mapeo resultante es aplicado al anillo Consistent Hashing.

Una conclusión importante de la evaluación experimental es que para tener un buen desempeño es necesario un equilibrio entre la tasa de *hit* y la eficiencia. Esto es crucial para mantener un bajo tiempo de respuesta promedio. Esto principalmente es la desventaja de las estrategias base, que mantienen un buen desempeño por separado en sólo una de estas dos características. La estrategia propuesta alcanza buenos indicadores en ambas características.

Los nodos que salen y entran al servicio no representan un problema para esta estrategia, ya que cada ejecución del algoritmo es completamente independiente de las otras invocaciones (sólo se necesita como entrada los nodos y su carga de trabajo). Por esta razón, el esquema propuesto permite lidiar con la agregación y fallas de nodos, y sea cual sea el contexto de la carga de trabajo y estado/número de los nodos, el mecanismo siempre tenderá a balancear el sistema a la carga promedio, debido a las condiciones de estabilidad y convergencia de los algoritmos.

Cuando el algoritmo da la salida en forma de movimientos de rangos entre particiones, éste estima que con esos movimientos se alcanzará la carga promedio, ya que cada sección del anillo "pesa" lo mismo. Es decir, para los algoritmos estudiados, la carga es uniformemente distribuida en el anillo, cosa que no es cierto en esta aplicación. Por esta razón es importante que cada consulta que tenga alto impacto

en el anillo, sea distribuida en más particiones (capítulo 5). De esta forma, lo que se trata de realizar es homogeneizar los ítemes con más impacto en el anillo. Así, este mecanismo de balance de carga alcanza buenos resultados previa homogeneización de los ítemes (trabajando en *tandem*).

Finalmente, este esquema tiene el mismo funcionamiento en caso de que se utilicen nodos virtuales en el Servicio de Cache, la cual es una solución adoptada por otras estrategias del estado del arte, pero se debe considerar que entre más secciones componen en anillo, se necesitan más pasos para alcanzar la carga promedio. Sin embargo, se tiene una complicación adicional: ¿cuál es el peso de cada sección/nodo virtual asociado a una partición? Es decir, se puede obtener la carga de trabajo de un nodo, pero ¿cómo se divide esta carga entre las distintas secciones/nodos virtuales? Es por estos motivos que la utilización de nodos virtuales no implica una mejora en este contexto de aplicación.

Consultas en Ráfaga

En este capítulo se propone una solución para detectar tempranamente consultas en ráfaga a partir de información reducida de un conjunto compacto de consultas (más frecuentes). Esto impide el alza abrupta de tráfico hacia un nodo en particular.

En la sección 7.1 se describe el contexto de aparición de las consultas en ráfaga, su impacto y una estimación acerca del número de eventos detectados como ráfaga en un intervalo de tiempo. En la sección 7.2 se determinan características de las consultas más frecuentes, con la finalidad de establecer diferencias entre ellas. La base de la solución propuesta es la estructura de monitorización utilizada en el capítulo 5 que permite obtener las consultas más frecuentes con cierto margen de error. Para diferenciar la solución propuesta, en la sección 7.3 se hace una revisión del Estado del Arte de los mecanismos para la detección de consultas en ráfaga. A grandes rasgos, la solución propuesta considera series de tiempo a partir de frecuencias aproximadas de consultas y elementos estadísticos sobre esta serie para declarar ráfagas. Esto es descrito en la sección 7.4. En la sección 7.5 se realiza una comparación con un mecanismo de detección de ráfagas *off-line*, lo que permite ver la eficiencia en la detección de la solución propuesta. Esta comparación se realiza con un conjunto de consultas en ráfaga obtenidas desde *logs* de consultas reales. Posteriormente, en la sección 7.6 se introduce el mecanismo propuesto en un Servicio de Cache para observar cómo actúa la detección y mitigación de este tipo de eventos. Las conclusiones del capítulo se encuentran en la sección 7.7.

7.1. Motivación

Las consultas en ráfaga generan las consecuencias más dañinas en términos de balance de carga para un sistema de cache, en comparación con el desbalance estructural

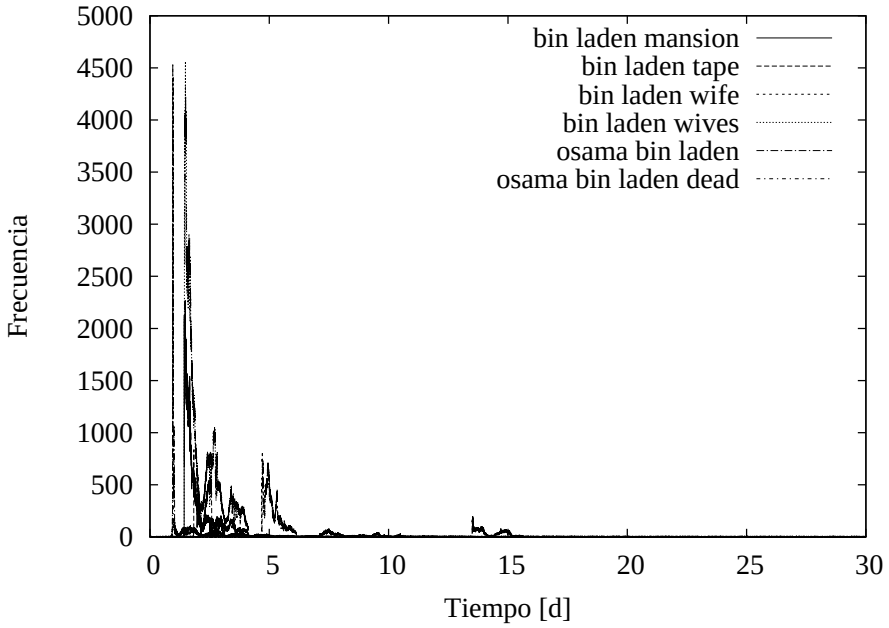


Figura 33: Consultas relacionadas al tópico "osama bin laden" (Mayo de 2011).

y la caída/agregación de nodos. Esto es debido a que múltiples instancias de la misma consulta son asignadas a la misma máquina en un intervalo de tiempo muy reducido. Este genera desbalance y un aumento de la posibilidad de servidores congestionados, por lo que es necesario tratarlas para lograr un Servicio de Cache balanceado.

Estas consultas a menudo suceden por eventos noticiosos relacionados con celebridades, economía, la sociedad, entre otros, que hacen que los usuarios quieran profundizar en la búsqueda de información de estos eventos. Según la evidencia recolectada, un tópico en particular no genera una sola consulta, sino que produce un conjunto de consultas en torno a ese tópico. Por ejemplo, en el momento en que se hizo pública la muerte de Osama Bin Laden, se generaron consultas relacionadas con la biografía, fotos, videos, entre otros, que también se transformaron en consultas en ráfaga. Esto puede ser observado en la Figura 33. Esto implica que la aparición de consultas en ráfaga (relacionadas con un tópico) produce desajustes en un conjunto de nodos. Esto último es otro factor importante para el estudio de este tipo de consultas y la aplicación de medidas mitigatorias ante su aparición. En [SC13] se encuentran patrones similares a los presentes en la Figura 33.

Es imposible predecir este tipo de consultas con días u horas de anticipación. Una alternativa es el procesamiento en tiempo real del flujo de consultas, obtener estadísticas y determinar con algún método si es ráfaga o no. Al ser en tiempo real, existen restricciones fuertes de tiempo y espacio para ello. Este tipo de análisis miope (en un instante en particular), no permite ver la historia pasada de una consulta. La historia pasada de una consulta permite saber más de ella, ya que si una consulta en particular no ha aparecido anteriormente y además de un instante a otro tiene un alto impacto, probablemente se trate de una consulta en ráfaga.

Como se mencionó anteriormente, el análisis de un flujo de consultas presenta restricciones de tiempo y espacio para hacer el proceso eficiente. Tanto el número de consultas a analizar, como el largo de la historia de cada consulta, debe ser analizado y determinado de forma tal que no represente una sobrecarga importante para el sistema.

Esta es la hipótesis de trabajo de este capítulo: la historia acotada de un cierto conjunto de consultas ayuda a determinar tempranamente la aparición de consultas en ráfaga, a través de una solución eficiente en tiempo y espacio.

Del análisis de un *log* de consulta y utilizando el método descrito en el trabajo de Vlachos *et al.* [VMVG04], se determinó que aproximadamente un 0,00000182 % de consultas presentan una características de ráfagas (considerando el total de consultas distintas). Si bien estas consultas no representan un gran porcentaje, su impacto en términos de balance de carga en el Servicio de Cache hacen necesario su estudio.

7.2. Evidencia

En el capítulo 5 se discutió el uso de un mecanismo rápido y compacto en espacio para la determinación de los ítemes más frecuentes. A través del análisis de esta estructura, se han evidenciado las siguientes características de las consultas monitorizadas que pertenecen a ella:

- **Dinámica**. El conjunto de consultas varía a través del tiempo, y también varía el impacto que tiene cada una de ellas dentro de este conjunto.

- **Estacionalidad**. La mayor porción de consultas permanece monitorizada a través del tiempo, pero existen consultas que aparecen como más frecuentes repentinamente y luego desaparecen de la estructura. De lo anterior, se definen dos sub-clases de las consultas frecuentes: (i) consultas frecuentes permanentes; (ii) consultas frecuentes en ráfaga. Esta misma clasificación (Ráfaga o Estable) es encontrada en [SC13].

Las consultas frecuentes permanentes (o consultas permanentes) son las consultas más frecuentes de los *logs* de consultas que trascienden cualquier escala de tiempo. Estas consultas son las indicadas para distribuir en múltiples máquinas como fue detallado en el capítulo 5. Por otro lado, las consultas con alto impacto en un intervalo de tiempo acotado, llamadas consultas frecuentes en ráfaga (o consultas en ráfaga), deben ser declaradas como tal antes de que alcancen su punto máximo. En la Figura 34 se aprecian dos consultas distintas a través del tiempo, una es permanente ('craigslist') la otra es ráfaga ('khloe kardashian'). Esta información fue obtenida graficando la frecuencia aproximada de la estructura para esas consultas en intervalos de un minuto. Estos patrones también son encontrados en [SC13].

Una consulta monitorizada presenta una historia a través del tiempo en la estructura. Claramente los dos tipos de consulta de la clasificación anterior presentan dos historias distintas, específicamente, en su forma de aparición dentro de la estructura. Cada una de las consultas frecuentes puede ser representada como una serie de tiempo, en que cada uno de los puntos de la serie corresponde a la frecuencia aproximada dada por la estructura en distintos intervalos de tiempo (contiguos siguiendo el orden temporal). Esta es la idea que se explotará: dado que se tiene una estructura de monitorización que opera en forma rápida y compacta en espacio, se podría formar una serie de tiempo por cada consulta presente en ella y aplicar algún método que facilite la clasificación de una consulta como ráfaga en base a esta información.

Para no incrementar en forma dramática el espacio, se define una ventana de tiempo en la cual se tiene la información relacionada a la frecuencia aproximada de cada consulta. Como se verá en la evaluación, la serie de tiempo por cada consulta no debe ser extensa para obtener buenos resultados.

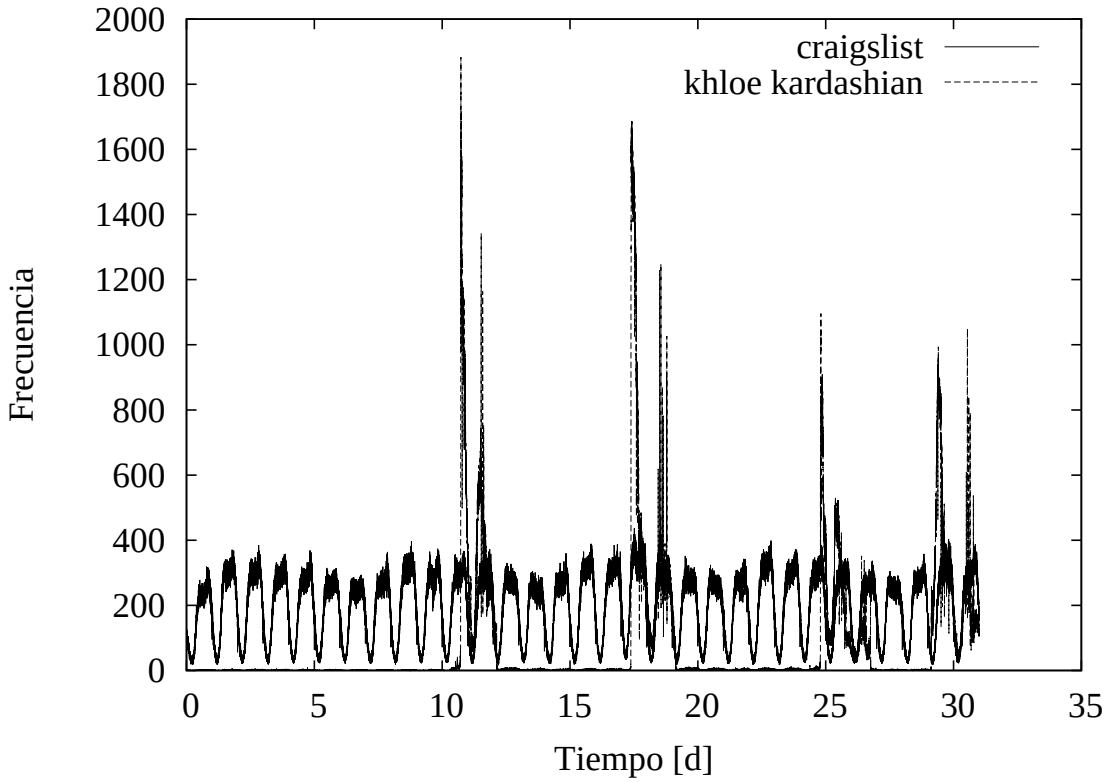


Figura 34: Consultas permanente y ráfaga (Enero de 2012).

Tabla 18: Caracterización de las consultas frecuentes en la estructura.

Característica	**Permanente**	**Ráfaga**
Permanencia en la Estructura	Infinita	Algunos minutos u horas
Cambios en la Frecuencia Aproximada	Acotados	Alzas abruptas hasta el punto máximo, disminuciones graduales desde el punto máximo
Promedio de la Serie de Tiempo	Siempre Alto	Bajo al inicio, sube abruptamente
Desviación Estándar de la Serie de Tiempo	Baja	Alta

En principio, existen algunos patrones que diferencian cada tipo de consulta para obtener una caracterización. Según la evidencia recolectada, en la Tabla 18 se resumen algunas características principales que permiten caracterizar las dos clases de consultas frecuentes.

La importancia del punto máximo de una consulta en ráfaga es para clasificarla como tal antes que se alcance este punto crítico.

7.3. Estado del Arte

La detección de consultas en ráfaga ha sido estudiada con diversos enfoques. En esta sección se hará un resumen de los trabajos más importantes.

Kleinberg [Kle03] propone un método para detectar y ordenar ráfagas en flujos de texto. Su enfoque se basa en el uso de un autómata de estados infinitos como estructura jerárquica para clasificar la frecuencia de palabras. El estado de cada palabra se determina por aquel que minimiza una función de costo basada en probabilidad. También se prueba que si no se necesita ordenar las ráfagas, un autómata de dos estados es suficiente para identificarlas.

Zhu y Sasha [ZS03] proponen un método para detectar ráfagas en series de tiempo utilizando una ventana elástica. Los autores proponen una estructura de datos, denominada *Shifted Wavelet Tree*, para asegurar tiempo lineal tanto de búsqueda como de actualización de los valores en la ventana. Cada ventana utilizada posee su propio

threshold que se adecúa según los datos presentes en ella utilizando el promedio y la desviación estándar. Una desventaja visualizada del método es su sensibilidad al tamaño de la ventana, ya que afectan los períodos reportados como en ráfaga.

Vlachos *et al.* [VMVG04] estudian las consultas de un motor de búsqueda como series de tiempo, proponiendo un método para descubrir períodos importantes e identificar ráfagas. Su trabajo se basa en el uso de la Transformada Discreta de Fourier para la extracción de características. La detección de las ráfagas se realiza comparando el valor del promedio móvil con un *threshold* calculado en base a la media y la desviación estándar. La información del *threshold* es calculada *off-line*. Sólo se guardan las partes representativas de la ráfaga, almacenando la intensidad de la ráfaga calculada como el promedio de los valores durante dicho período y el tiempo de inicio y fin. Posteriormente, está información se utiliza para obtener consultas de características similares.

También utilizando un enfoque probabilístico, Cheong *et al.* [FYYL05] identifican ráfagas en las palabras o características de un flujo de texto, agrupando las que son similares y reconociéndolas como parte de un mismo fenómeno. Dado que los datos para el flujo de texto se obtienen de artículos de un diario, los eventos en ráfaga corresponden a noticias relevantes y las características que la componen a palabras clave de las mismas. Este último método supone una mejora sobre el modelo de Kleinberg, ya que además de determinar las características en ráfaga, es capaz de agruparlas por eventos sin preocuparse del ajuste de parámetros.

En el contexto de servidores proxy, Yoo et al. [YC07] utilizan un modelo neuronal para anticipar la aparición de hot spots en un sistema de cache distribuido. Al comparar en un simulador el rendimiento del sistema actual con el que incorpora la lógica de predicción se aprecia un mejor desempeño. No se entrega mayor detalle sobre las características del predictor, como número de mediciones previas requeridas, implementación de la red neuronal y horizonte de predicción.

Parikh y Sundaresan [PS08] estudian la detección y clasificación de ráfagas en un sitio de eCommerce utilizando el autómata de dos estados descrito en el trabajo de Kleinberg [Kle03], pero proponiendo un método propio para la clasificación y *ranking*. La clasificación se realiza utilizando agrupamiento no supervisado. En primer lugar se

utiliza la Transformada de Daubechies para extraer las características de cada serie de tiempo, luego se agrupa utilizando K-medias y distancia Euclidiana. Entre las ventajas citadas del enfoque se encuentra la detección de ráfagas en consultas no tan populares y en aquellas moderadamente populares por períodos pequeños.

Klan *et al.* [KKP+09] mejoran el método de detección ráfagas por ventana elástica de Zhang y Shasha [ZS06], añadiendo capacidad de predicción para superar la poca adaptación del *threshold* ante datos no estacionarios en la ventana. Los métodos de predicción utilizados son suavizado simple y doble para la predicción, los cuales añaden más parámetros que configurar. Los resultados indican que el método debe adaptarse para cada caso particular, ya que además de la tendencia de los datos se puede requerir ajustes en los parámetros de los predictores y el tamaño de ventana. En cuanto al tiempo de procesamiento y memoria necesarios, los autores reportan que son lineales en función del flujo de datos procesados.

En su estudio del comportamiento de los usuarios durante las consultas en ráfaga, Subasic y Castillo [SC10] utilizan un método de *threshold* basado en una métrica propuesta, denominada *burst intensity*, para medir intensidad de ráfaga comparándola con el promedio de los valores en la ventana. Además de esto, los autores tratan de obtener clasificaciones y otras características representativas basadas principalmente en las sesiones de usuario y sus consultas.

7.4. Solución

Para obtener un método que permita la detección temprana de consultas en ráfaga, se analizarán consultas frecuentes de un *log* de consulta y se clasificarán como permanentes y ráfagas. La idea de esto es determinar las características diferenciadoras entre ambas. Se estudiarán desde dos puntos de vista: (i) frecuencia absoluta; (ii) frecuencia aproximada. Frecuencia absoluta corresponde a la frecuencia exacta de una consulta en cualquier instante de tiempo (impracticable), y con frecuencia aproximada se refiere a la generada por el algoritmo de monitorización. Con esto último se verá qué tanta precisión se alcanza con un método aproximado de obtención de frecuencias.

El primer paso es definir qué es ráfaga y qué no. Para esto se utiliza el mecanismo basado en promedio móvil de Vlachos *et al.* [VMVG04]. Para descubrir regiones de ráfagas para una consulta q en una secuencia, los autores utilizan el promedio móvil de largo w (MA_w) para una secuencia de puntos $\{t_1(q), t_2(q), ..., t_n(q)\}$, con $n > w$, que corresponden a la misma consulta q. Es decir, $t_i(q)$ corresponde a la frecuencia de la consulta q en el intervalo i, y $MA_w(t_i(q))$ corresponde al promedio móvil de largo w para la consulta q considerando los puntos $t_{i-w}(q), t_{i-(w-1)}(q), ..., t_{i-1}(q), t_i(q)$. Una vez calculados todos los promedios móviles ($MA(q)$), se obtiene el promedio de los promedios móviles $\overline{MA(q)}$ y la desviación estándar $std(MA(q))$. Por cada consulta se define un umbral $thres(q) = \overline{MA(q)} + x \cdot std(MA(q))$, con $x = 2,0$. El conjunto de puntos declarados como ráfaga para q es $B(q) = \{t_i(q) | MA_w(t_i(q)) > thres(q)\}$. La idea principal es la detección de consultas en ráfaga mediante la presencia de valores atípicos, en términos de frecuencia, considerando el promedio de la consulta.

Este método es *off-line*, ya que se necesita tener la secuencia completa de puntos para obtener el promedio y varianza de los promedios móviles. Además, es impracticable para la detección *on-line* de consultas en ráfaga, debido a la cantidad de información que se debe almacenar para su operación (por cada consulta se debe almacenar su umbral).

En virtud de la eficiencia, se aplican las siguientes restricciones. Por cada *log* utilizado a nivel de mes, se seleccionan las 100 mil consultas más frecuentes para realizar los experimentos. Esto nos da una cobertura de aproximadamente un 50 % del tráfico de cada mes, y además la consulta menos frecuente de este conjunto tiene aproximadamente 1.000 ocurrencias en el mes (no representará un impacto importante ni tampoco se transformará en ráfaga). Finalmente, en cada intervalo de tiempo sólo se consideran las consultas cuya frecuencia aproximada es mayor que una cota inferior definida. Por ejemplo, si el intervalo de tiempo es un minuto, entonces la cota inferior considerada es 100 (sólo se consideran consultas con frecuencia aproximada mayor que 100).

Se estima que la aplicación de las restricciones descritas no afectan los resultados, ya que las consultas descartadas no aparecen en la estructura de monitorización ni impactan en el desempeño del Servicio de Cache, ambas razones debido a su baja

Tabla 19: Consultas en ráfaga usando método *off-line* (Vlachos *et al.* [VMVG04]).

Tamaño de Ventana	F. Absoluta		
	12/2011	01/2012	02/2012
$w = 5$	926	925	971
$w = 10$	923	921	970
$w = 15$	920	919	969

frecuencia.

El análisis inicial consiste en examinar la efectividad de la detección de consultas en ráfaga utilizando el umbral $thres(q)$ de Vlachos *et al.* [VMVG04] y evaluar su desempeño con frecuencias absolutas. El primer paso es calcular el umbral $thres(q)$ que se realiza en forma *off-line* y en una pasada completa del *log* de consultas. Las mediciones de frecuencia se realizan cada un minuto y el tamaño de la ventana w considerado es 5, 10 y 15. Es decir, el promedio móvil para calcular el umbral $thres(q)$ es de largo 5, 10 y 15. El segundo paso es procesar el *log* de consultas considerando el umbral $thres(q)$ calculado *off-line* y el promedio móvil *on-line* para cada consulta (nuevamente el largo del promedio móvil es 5, 10 y 15). Los experimentos consideran el promedio móvil *on-line* utilizando frecuencias absolutas.

Con el proceso descrito anteriormente, se tienen 925 consultas categorizadas como ráfaga (0,00000182 % del total de consultas distintas) utilizando la frecuencia absoluta, durante el mes de Enero de 2012 y con una ventana de tamaño $w = 5$. En la Tabla 19 se expone el número de consultas categorizadas como ráfagas en las distintas configuraciones de w y para distintos *logs* de consultas.

En la Tabla 19 se observa que aumentar el tamaño de la ventana de tiempo tiene un impacto limitado en el número de consultas en ráfaga detectadas. Dados estos resultados y el poco impacto en el incremento del largo de la ventana de tiempo, se continuará desde aquí en adelante sólo con $w = 5$.

Como fue mencionado anteriormente, mantener la información relativa al umbral $thres(q)$ significa un *overhead* importante al implementar un sistema de detección de alto desempeño. Dada la gran cantidad de consultas distintas, esto se torna impracticable. Además, se ha demostrado que las consultas de usuario muestran un alto

dinamismo, lo que implica un sesgo en la determinación *off-line* del umbral. Finalmente, se debe lidiar con el largo del promedio móvil y cada cuánto tiempo se debe actualizar la información.

Adicionalmente a lo anterior, existe una seria anomalía al instaurar este tipo de mecanismo. Considerar tres consultas permanentes cuya frecuencia absoluta se muestra en la Figura 35 para la primera semana de Enero de 2012. Se observa que la frecuencia de las consultas varía, teniendo las mayores frecuencias durante el día y las menores durante la noche. En general, este patrón se repite a través del tiempo en la mayoría de las consultas permanentes. En el trabajo de Vlachos *et al.* [VMVG04], los resultados obtenidos consideran períodos extensos de tiempo: un año completo con promedio móvil de 7 a 30 días. La unidad de tiempo que los autores consideran para obtener las frecuencias es un día, por lo que la detección de consultas en ráfaga es en la misma escala de tiempo. Se recalca además que el patrón observado en la Figura 35 no está presente en ese trabajo [VMVG04]. Específicamente, si el umbral $thres(q)$ es calculado en base al promedio global y la desviación estándar, se tendrá un umbral que no representa correctamente el punto de corte para clasificar algo como ráfaga o no. Sería ideal contar con un mecanismo que se adapte al volumen de tráfico, que no imponga un excesivo *overhead* en términos de espacio y que no requiera de información que deba calcularse en forma *off-line*.

Como fue descrito en el capítulo 5, la estructura de monitorización permite obtener las K consultas más frecuentes y su frecuencia aproximada con un margen de error. La idea es obtener una serie de tiempo por cada consulta presente en la estructura. Sea T el tamaño de la ventana de tiempo, entonces cada consulta presente en la estructura puede ser representada por una serie de tiempo, desde el instante actual hasta $T - 1$ unidades de tiempo atrás.

El espacio utilizado para tener K consultas con una serie de tiempo compuesta por T puntos, está acotado a $O(T \cdot K)$. Existen algunas pequeñas variaciones cuando una nueva consulta entra a la estructura y alguna otra sale de ella, pero esta variación es marginal. Como K es pequeño, del orden de centenas, y además las series de tiempo usadas en este trabajo también son muy pequeñas, menor a una decena, entonces el espacio utilizado no representa una sobrecarga importante.

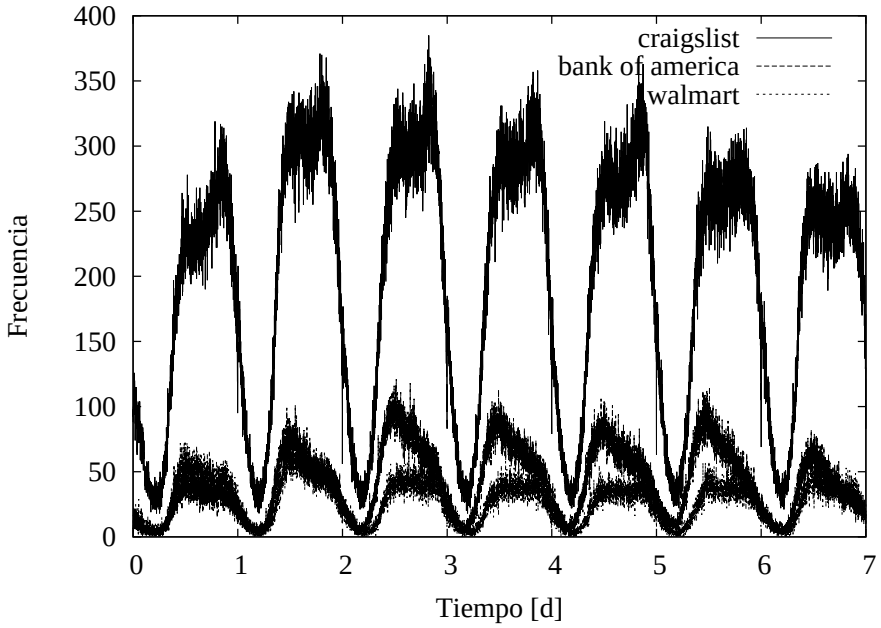


Figura 35: Consultas permanentes (primera semana de Enero de 2012).

Por otro lado, una serie de tiempo acotada a T intervalos de tiempo hacia atrás no necesita información histórica de largo plazo. Además, una serie de tiempo acotada se adapta naturalmente a las condiciones de tráfico detalladas anteriormente. La idea es utilizar el concepto de promedio móvil para cada consulta, pero sólo con la información disponible en la serie de tiempo. Además, existe un factor importante a considerar que fue mencionado en la Tabla 18: promedio y desviación estándar de las consultas permanentes versus las ráfagas.

Considerando una serie de tiempo para una consulta permanente, se tiene que las variaciones en esta serie son acotadas y graduales. Esto implica que, además de tener un promedio alto, la desviación estándar de esta serie es baja. Por otro lado, las consultas en ráfaga presentan una alta desviación estándar, debido a la variación de los puntos de la serie de tiempo en comparación con el promedio (de la serie). En la Tabla 20 se muestran algunas series de tiempo para consultas permanentes y en ráfaga (frecuencia absoluta en intervalos de un minuto).

La idea principal de este trabajo para detectar una consulta en ráfaga es considerar dos hechos respecto a las series de tiempo: (i) promedio móvil; y (ii) desviación

Tabla 20: Series de tiempo para consultas permanentes y ráfagas.

Permanente			**Ráfaga**		
Serie	Promedio	Desv. Est.	Serie	Promedio	Desv. Est.
65 57 55 79 68	66	8,59	9 28 71 123 151	76	54,13
111 106 113 94 126	110	10,37	34 66 113 239 400	170	134,32
262 273 297 267 285	277	12,68	94 138 286 383 725	325	225,04
1104 1140 1057 1098 1163	1112	36,52	145 208 415 850 1388	601	464,40

estándar de la serie. Con (i) promedio móvil se refiere a que si la frecuencia actual está sobre el promedio móvil de la serie, implica una anomalía que podría traducirse en una alza en tipo ráfaga de la frecuencia. Con (ii) alta desviación estándar se refiere a que exista alta dispersión de los datos, que podría implicar una condición clave para la detección de una serie de tiempo anómala. Además, si sólo se consideran series de tiempo de tamaño limitado (sin ninguna otra información histórica), se tiene que las estadísticas obtenidas de ellas se adaptan a la variación en el volumen del tráfico de las consultas de usuario.

La desviación estándar es una medida de variación de todos los valores de un conjunto con respecto a la media [Tri09]:

$$\sigma = \sqrt{\frac{\sum (x - \mu)^2}{N}}$$

Esta medida es igual a cero cuando los valores son iguales a su media, y es alta cuando existe mayor cantidad de variación en los datos. Además, el valor de la desviación estándar puede aumentar de manera drástica si se incluyen uno o más valores lejanos con respecto a los demás. Finalmente, la unidad de medida de la desviación estándar es la misma que los datos de la muestra.

En estadística, existe la denominada "regla práctica del intervalo" [Tri09], que se basa en el principio de que para una amplia cantidad de conjuntos, la mayoría (tanto como un 95 %) de los valores muestrales se ubican dentro de dos desviaciones estándar a partir de la media. Este principio es el utilizado en el trabajo de Vlachos *et al.* [VMVG04] para su categorización ráfaga / no ráfaga.

Este trabajo utilizará la misma idea, pero sólo considerando los datos de la serie de tiempo de cada consulta. Es decir, por cada consulta se tiene asociada una serie

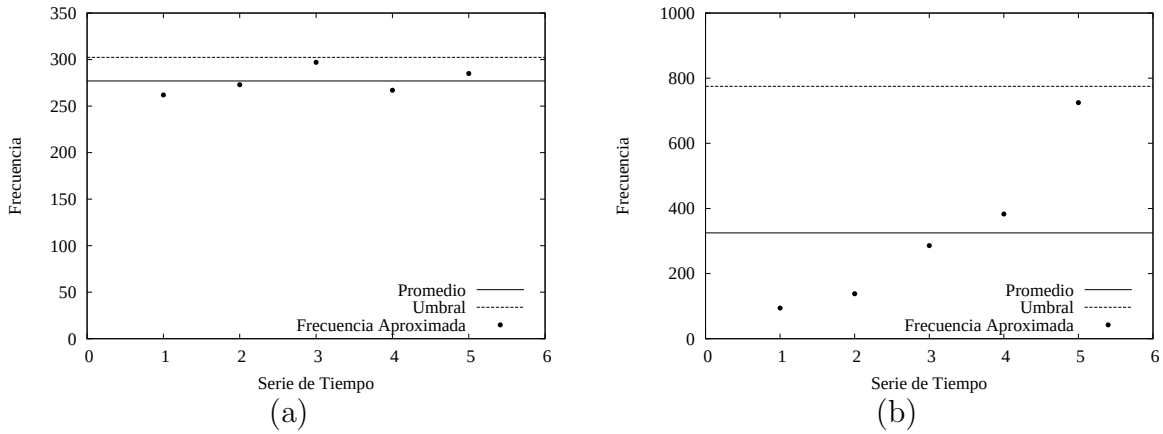


Figura 36: Promedio móvil y umbral del primer filtro para dos series de tiempo: (a) permanente; y (b) en ráfaga.

de tiempo de largo w, con lo que es posible obtener la media y la desviación estándar. Sea $f_t(q)$ el valor para la frecuencia en el intervalo actual para la consulta q, y sea $T_w(q) = \{f_{t-w}(q), f_{t-w+1}(q), ..., f_{t-1}(q)\}$ la serie de tiempo de largo w, cuyo promedio es $\overline{T_w(q)}$ con desviación estándar $\sigma(T_w(q))$. El primer filtro que se utilizará para la detección de una consulta en ráfaga es $f_t(q) \geq \overline{T_w(q)} + x \cdot \sigma(T_w(q))$, con x un parámetro que indica la amplitud desde el valor medio en términos de la desviación estándar.

En la Figura 36 se observan dos series de tiempo a las cuales se les ha obtenido el umbral que se define para el primer filtro. El parámetro utilizado es $x = 2$, es decir, promedio móvil más dos veces la desviación estándar de la serie. En la Figura 36(a) se observa el comportamiento de una consulta permanente, cuya variación es mínima con respecto al promedio de la serie. Por el contrario, la Figura 36(b) muestra un patrón de comportamiento abrupto al alza, lo que implica una alta variación en los datos y un posible rompimiento del umbral en el siguiente punto de la serie. Cabe señalar que el umbral calculado para ese conjunto de puntos se compara con la frecuencia aproximada del siguiente período.

Por otro lado, la desviación estándar es un factor importante para determinar la variación de valores respecto a la media. Existen algunas tasas que relacionan el promedio y la desviación estándar. Una tasa importante es el Coeficiente de Variación

c_v definida como la división entre la desviación estándar σ y la media μ [Tri09]:

$$c_v = \frac{\sigma}{\mu}$$

El Coeficiente de Variación es una medida normalizada (también puede ser porcentual) de dispersión de una población en relación con la media. Este coeficiente es independiente de la unidad de medida de la serie. La desventaja del Coeficiente de Variación es que el cálculo del promedio μ considera toda la muestra (ventana de tiempo infinita), que es la misma desventaja que el trabajo de Vlachos *et al.* [VMVG04]. Otras estadísticas utilizadas, en otros contextos, para el estudio de ráfagas son el Factor de Fano [GC10], el Coeficiente de Asimetría [Can03], entre otros.

La idea en este trabajo es que las características obtenidas de los datos sean calculadas en función de la serie de tiempo. El promedio y la desviación estándar son obtenidos para el primer filtro, por lo que se utilizará el Coeficiente de Variación sobre una ventana de tiempo como segundo filtro. A este valor se le denomina Coeficiente de Variación Móvil (*Moving Coefficient of Variation*). La segunda condición para la detección de una consulta en ráfaga es:

$$c_v(q) = \frac{\sigma(T_w(q))}{\overline{T_w(q)}} > r$$

Con r un parámetro que indica la cota mínima que debe tener la desviación estándar en proporción al promedio móvil.

En la Tabla 21 se exponen estadísticas, tanto el umbral como el coeficiente de variación, para las series de tiempo descritas anteriormente en la Tabla 20. Es claro que en las series expuestas existen diferencias sustanciales en las medidas descritas para la detección de ráfagas. Por un lado se tiene que la desviación estándar exhibe un valor elevado en las ráfagas, y también el coeficiente de variación es alto, por lo que indica que existe una correlación entre estas medidas y el tipo de consultas frecuente.

El coeficiente de variación tiene un promedio de 7,5 % para las consultas permanentes versus un 74 % para las ráfagas. Esta comparación porcentual es válida, ya que el coeficiente de variación permite comparar dos distribuciones cuando la escala de medición difiere de forma considerable entre éstas [Can03].

Tabla 21: Parámetros para consultas permanentes y ráfagas.

Permanente				**Ráfaga**			
Promedio	Desv. Est.	Umbral ($x = 2$)	$c_v(q)$	Promedio	Desv. Est.	Umbral ($x = 2$)	$c_v(q)$
66	8,59	83,18	13 %	76	54,13	184,26	71 %
110	10,37	130,74	9 %	170	134,32	438,64	79 %
277	12,68	302,36	5 %	325	225,04	775,08	69 %
1112	36,52	1185,04	3 %	601	464,40	1529,8	77 %

Las condiciones descritas anteriormente permiten declarar una consulta como ráfaga. Se enfatiza que en el preludio de la ráfaga estas condiciones tienen mayor validez, ya que a medida que la consulta aumenta su frecuencia también cambian sus características. A través de la experimentación, se detectó que validar el cumplimiento de las condiciones constantemente después de que una consulta es declarada como ráfaga, no es una buena política, ya que las características de la serie (promedio y desviación estándar) varían, siendo muy importantes al inicio del episodio ráfaga y no tan importantes después o en el transcurso de ésta. Entonces el cumplimiento de las condiciones es un "gatillador", es decir, un aviso de que la consulta es una ráfaga. Una vez que la consulta es declarada como tal, se mantiene como ráfaga mientras la frecuencia aproximada de la consulta en el tiempo sea mayor a una cota inferior c. Esto permite declarar una consulta como ráfaga durante un intervalo de tiempo en forma clara y precisa, como se verá más adelante. El Algoritmo 4 describe la propuesta.

Para el correcto funcionamiento del Algoritmo 4, $\sigma(q.serie)$ debe estar definida, por lo que $q.serie$ debe tener al menos dos mediciones de frecuencias aproximadas. Es decir, el método propuesto puede declarar una consulta como ráfaga con tres mediciones como mínimo. El costo temporal de la propuesta está dado principalmente por la monitorización de elementos (detallada en el capítulo 5) y por el examen de series de tiempo. Este último procedimiento tiene un costo de $O(K \cdot T)$, con K el número de consultas top-k y T el largo de la serie de tiempo.

7.4.1. Estacionalidad

Dentro de las consultas de usuario existen grupos de consultas, específicamente las permanentes, que generan componentes estacionales en la serie de tiempo, lo cual

Algoritmo 4 Ráfaga. Algoritmo propuesto para la declaración de una ráfaga a ejecutarse cada Δt unidades de tiempo.

Input: : SS estructura de monitorización, x amplitud desde el valor medio en términos de la desviación estándar, r cota mínima para coeficiente de variación, c cota inferior de frecuencia.

1: **for all** $q \in SS$ **do**
2: **if** $q.rafaga = true$ **then**
3: **if** $q.frec < c$ **then**
4: $q.rafaga \leftarrow false$
5: **end if**
6: **else**
7: **if** $\left(q.frec \geq \overline{q.serie} + x \cdot \sigma(q.serie)\right) \wedge \left(\frac{\sigma(q.serie)}{\overline{q.serie}} > r\right)$ **then**
8: $q.rafaga \leftarrow true$
9: **end if**
10: **end if**
11: **end for**

podría llevar a pensar que no existe desbalance para estas consultas. Sin embargo, aún con la presencia de grupos de consultas estacionales, el desbalance existe tal como lo muestran los resultados en esta sección.

En general, las series de tiempo pueden presentar componentes tendenciales, estacionales y/o aleatorias [BD02]. Las consultas de la Figura 35 no presentan tendencia (cambio a largo plazo en relación al nivel medio), sino que son series estacionarias discretas, ya que el valor medio y las variaciones son constantes a lo largo del tiempo. Esto se refleja gráficamente en que las series presentan oscilación alrededor de una media constante y las variaciones considerando esa media también son constantes. En la Figura 35 existen tres consultas permanentes que tienen una clara variación periódica. Para estudiarlas desde el punto de vista de la estacionalidad, se desestacionalizarán dos de ellas: (i) "craigslist" y (ii) "walmart". La idea es observar las variaciones sin la componente estacional para distintos intervalos de tiempo de las consultas. Estos intervalos de medición fueron minutos, horas y días.

Cada consulta tiene la medición total de un mes completo, por lo que existen varios períodos en cada serie. El procedimiento consiste en:

1. Determinar el período de la serie. Los períodos estudiados son día y semana.

Cada período tiene un conjunto de mediciones, por ejemplo, el período "día" tiene 1440 minutos o 24 horas.

2. Obtener el promedio para cada medición de un período considerando todos los períodos. Por ejemplo, si el período es una semana, en un mes existen 4 de ellas, entonces por cada día se debería sacar el promedio considerando las 4 semanas.

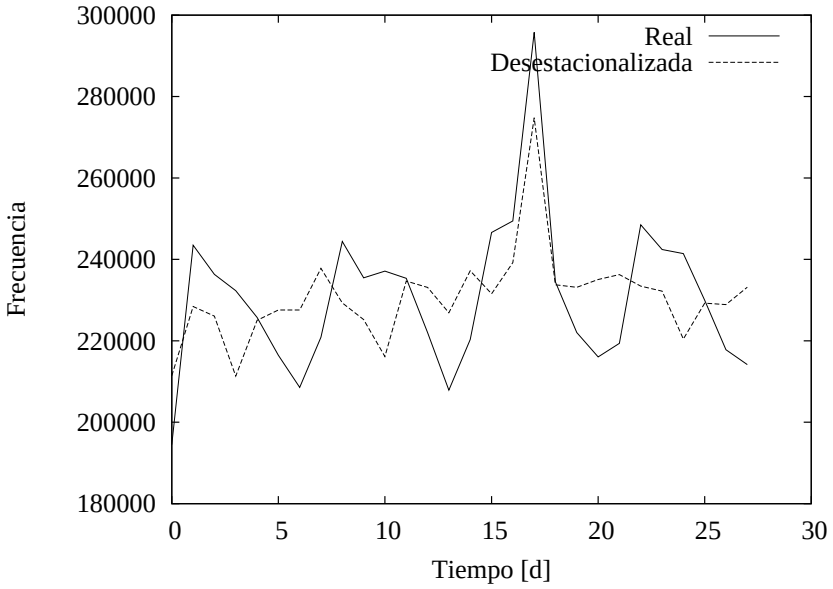
3. Obtener el promedio total de las observaciones.

4. Obtener el coeficiente estacional por cada medición del período, restando su promedio al promedio total de las observaciones.

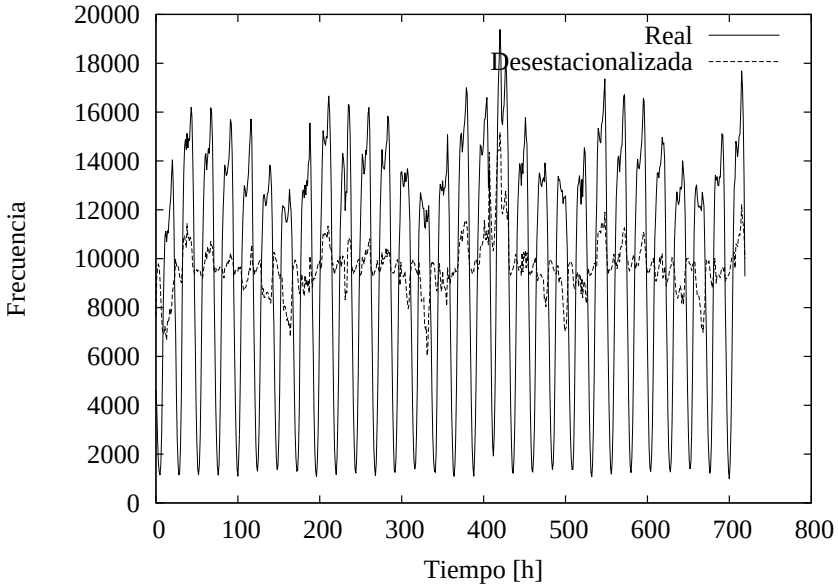
5. Por cada observación, se resta el coeficiente estacional (de acuerdo a la medición del período).

En la Figura 37(a) se muestra la consulta "craigslist" en Enero de 2012 considerando a la semana como período y día como medición. Esto quiere decir que en el mes completo existen 4 períodos, y es posible observar las variaciones diarias sin la componente estacional. En la Figura 37(b) se observa la misma situación pero considerando períodos de un día y mediciones a nivel de hora. En las dos figuras existen variaciones no predecibles a distintas escalas de tiempo. En particular, se observa un alza abrupta de esta consulta permanente alrededor de $x = 5$ en la Figura 37(a). Para complementar lo anterior, en la Figura 38(a) se observa el mismo proceso anterior, con períodos de un día con mediciones a nivel de minutos. Más claramente, la Figura 38(b) muestra el suavizado de la situación anterior. En las Figuras 39 y 40 se aplica el mismo procedimiento en los mismos períodos e intervalos de medición para la consulta "walmart". Contrastando ambos casos es posible observar que cada consulta tiene un comportamiento distinto y es difícil generalizar el comportamiento de las consultas permanentes.

A pesar de la eliminación de la estacionalidad con el método de promedios descrito, existen variaciones de distintas magnitudes, en distintos intervalos de medición y que no son previsibles que deben ser abordados por mecanismos de balance de carga. En este trabajo se adopta un mecanismo dinámico y adaptivo al volumen y variaciones

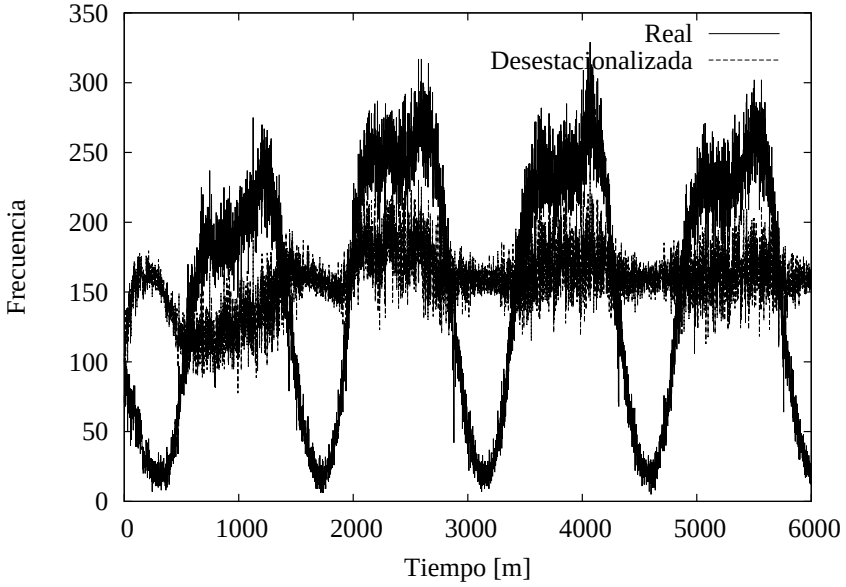


(a)

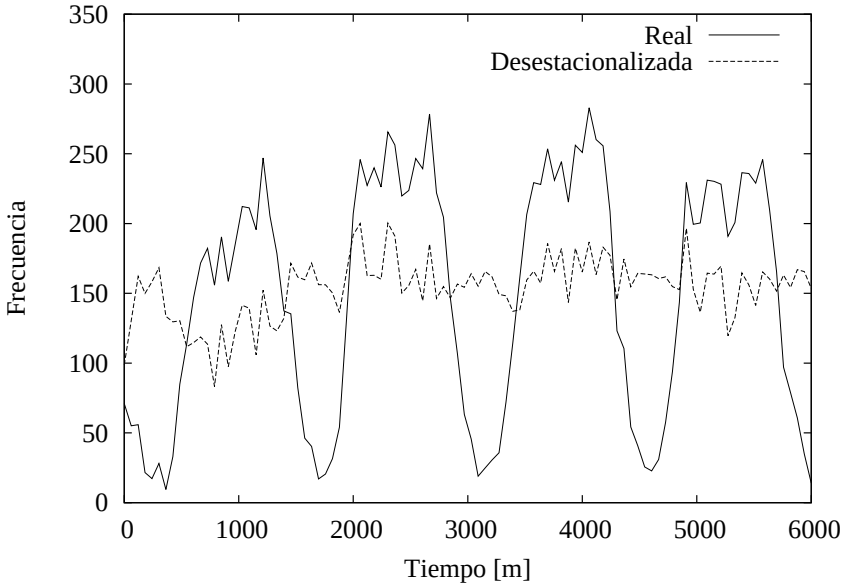


(b)

Figura 37: Serie real y desestacionalizada para la consulta “craigslist” en Enero de 2012: (a) período semana, medición día; (b) período día, medición hora.

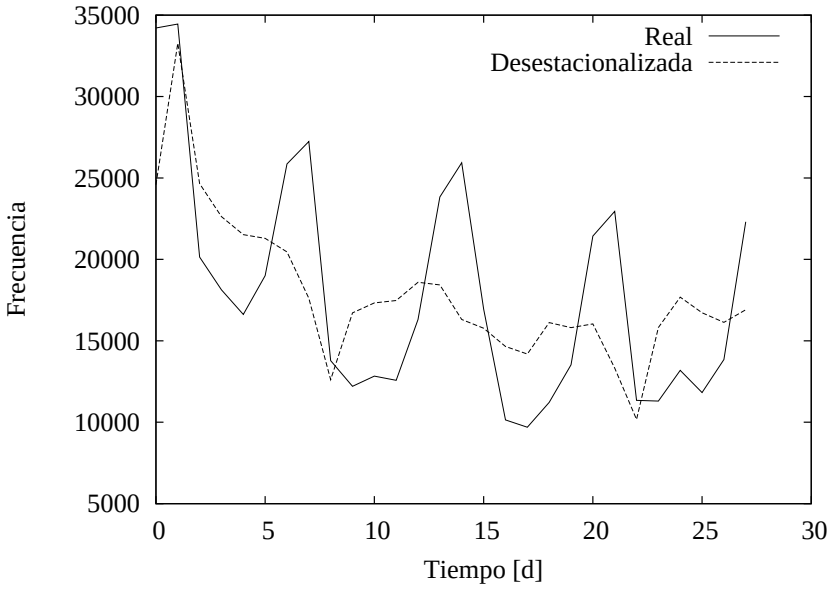


(a)

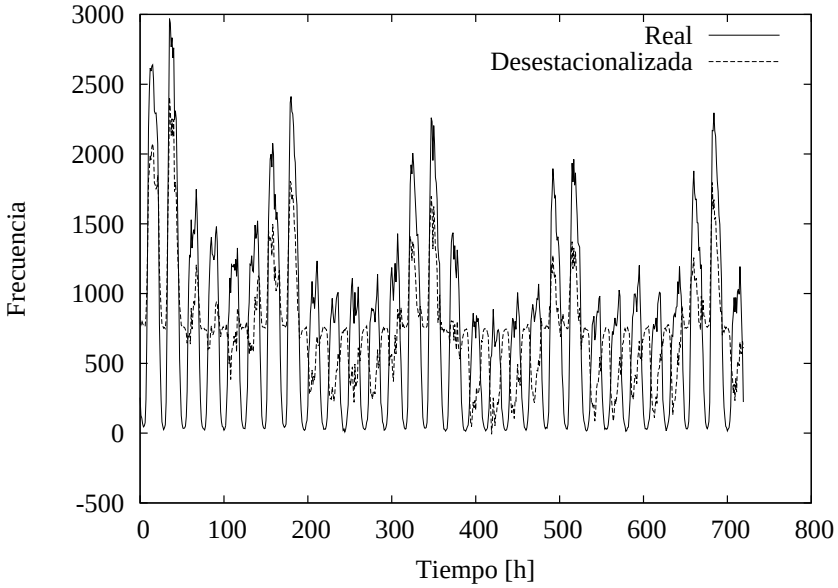


(b)

Figura 38: Serie real y desestacionalizada para la consulta "craigslist" en Enero de 2012 (sólo se grafican los primeros cuatro días): (a) período día, medición minuto; (b) suavizado de (a).

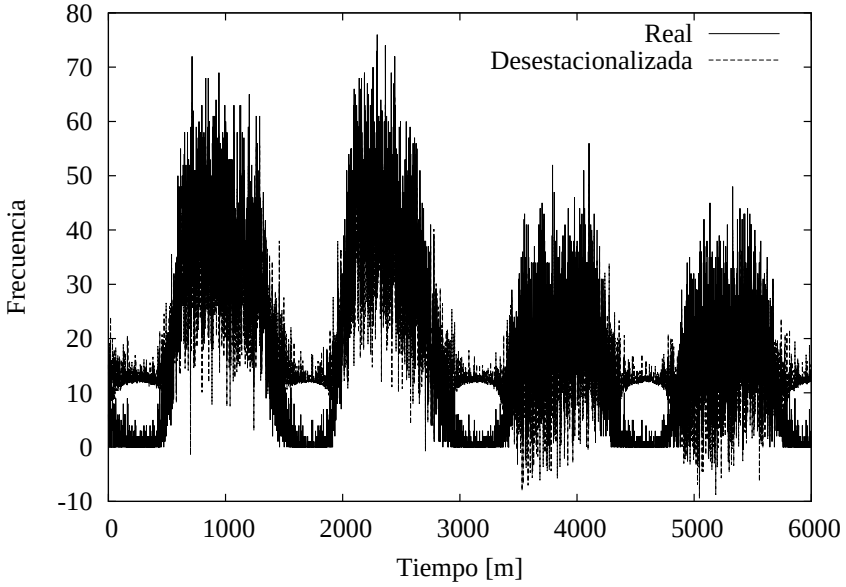


(a)

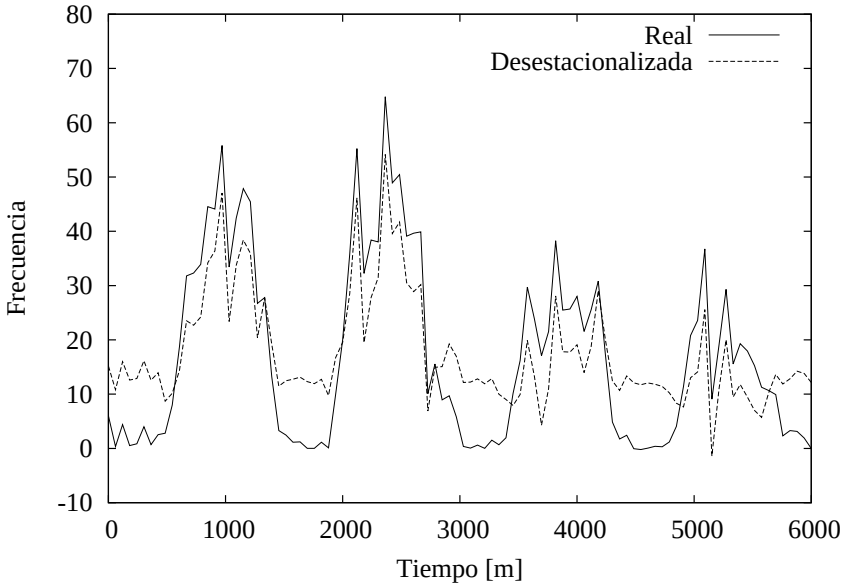


(b)

Figura 39: Serie real y desestacionalizada para la consulta "walmart" en Enero de 2012: (a) período semana, medición día; (b) período día, medición hora.



(a)



(b)

Figura 40: Serie real y desestacionalizada para la consulta "walmart" en Enero de 2012 (sólo se grafican los primeros cuatro días): (a) período día, medición minuto; (b) suavizado de (a).

de las consultas (ver capítulo 5), por lo que la característica de estacionalidad es abordada en forma automática. Esto último quiere decir que el mecanismo propuesto es independiente de si la serie tiene tendencia, estacionalidad o ruido. Por otro lado, las consultas en ráfagas no presentan los componentes mencionados anteriormente, por lo que la estacionalidad no aplica, de ahí la importancia de contar con mecanismos que permitan detectar este tipo de eventos que tienen características no previsibles.

7.4.2. Análisis

El coeficiente de variación y el umbral están compuestos por el promedio móvil y la desviación estándar. Existen cuatro casos a analizar con respecto a estas medidas, pero se debe pensar que en esta propuesta, el cumplimiento de ambos filtros será considerado como una consulta en ráfaga.

1. Desviación estándar baja - Promedio móvil bajo: consulta permanente con poca frecuencia.
    - Tener un valor más arriba que el umbral es simple de cumplir.
    - Implica que c_v es cercano a 0.
2. Desviación estándar baja - Promedio móvil alto: consulta permanente con alta frecuencia.
    - Tener un valor más arriba que el umbral también es simple de cumplir.
    - Implica que c_v es muy cercano a 0.
3. Desviación estándar alta - Promedio móvil bajo: consulta en ráfaga.
    - Tener una frecuencia más alta que el umbral se torna un poco difícil, por lo que el cumplimiento implica que tiene características atípicas a las consultas permanentes (alta posibilidad de que sea ráfaga).
    - En este caso, el c_v se aleja de 0 y mientras más se aleje, más características de ráfaga tiene la consulta. Según la evidencia examinada, la mayor parte de las consultas en ráfaga comienza con intervalos con frecuencia

baja (promedio móvil bajo) seguidos de abruptas alzas en frecuencia (alta desviación estándar), y en este caso el índice alcanza valores mayores que 1.

4. Desviación estándar alta - Promedio móvil alto: consulta permanente con alta frecuencia.

   - Al contrario del caso anterior, este caso es aún más difícil de cumplir, ya que es poco probable que una consulta con un promedio móvil alto se transforme en ráfaga.
   - El c_v es cercano a 0, ya que en esta medida predominaría el promedio móvil.

Existen dos parámetros en las condiciones para la detección de un ráfaga: (i) x (veces que se aleja del promedio móvil en términos del valor de la desviación estándar); y (ii) r (cota inferior de la relación entre desviación estándar y promedio móvil). Se evaluarán algunos valores de estos parámetros para determinar el impacto en la detección. También el largo de la serie se someterá a evaluación, pero anteriormente se observó que el impacto de una serie de tiempo muy larga es limitado.

7.5. Resultados y Discusión

Una consulta q es categorizada como ráfaga en el instante t si se cumplen las siguientes condiciones sobre su serie de tiempo $T_w(q)$ (de tamaño w):

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

Con $f_t(q)$ la frecuencia aproximada en el instante t. Los parámetros son:

1. x: si la frecuencia actual está x veces la desviación estándar (móvil) sobre el promedio móvil.

2. r: si la desviación estándar (móvil) es r veces el promedio móvil.

Tabla 22: Número de consultas *on-line*. x indica la cantidad de desviaciones estándar (móvil) sobre el promedio móvil y r la cota mínima para el coeficiente de variación móvil (Enero de 2012).

x	r						
	0,10	0,15	0,20	0,25	0,30	0,35	0,40
0,5	909	871	827	810	795	785	776
1,0	909	861	821	809	792	782	774
1,5	903	838	804	792	776	764	758
2,0	877	808	777	761	743	730	723
2,5	823	747	725	715	697	682	674
3,0	739	676	661	647	628	612	606
3,5	668	613	594	577	561	545	536

La experimentación se realiza con distintos valores de x y r. En la Tabla 22 se encuentra el número de consultas detectadas como ráfagas con distintos x y r para la ejecución completa de Enero de 2012. Recordar que en la Tabla 19, para este mismo mes, se categorizaron 925 consultas como ráfagas usando el método de Vlachos *et al.* (a este conjunto de consultas se le llamará *Vlachos*). Lo primero que se observa en la Tabla 22 es que la variación de los parámetros afecta el desempeño del método, haciendo que se detecten menos consultas cuando las condiciones son más restrictivas.

El número de consultas detectadas no es la métrica más importante, ya que podrían existir falsos positivos y verdaderos negativos. Considerando que el conjunto *Vlachos* contiene las consultas en ráfaga con cierto nivel de confianza, en la Tabla 23 se observa la tasa de consultas detectadas *on-line* que se intersectan con *Vlachos* (verdaderos positivos). Se observa que existe un alto número de consultas detectadas positivamente (comparadas con las 925 de *Vlachos*), considerando frecuencias aproximadas y una ventana de tiempo de largo $w = 5$.

Otro valor importante es la tasa de falsos negativos definida como el número de consultas en ráfaga de *Vlachos* que no fueron detectadas por el mecanismo propuesto. Esto se encuentra en la Tabla 24. Finalmente en la Tabla 25 se encuentra la tasa de falsos positivos (consultas detectadas como ráfagas pero que no lo son considerando *Vlachos*). Cabe señalar que no tiene sentido calcular los verdaderos negativos.

Tabla 23: Tasa de *verdaderos positivos* (VP): *on-line* $\cap$ *Vlachos* (Enero de 2012).

x	r						
	0,10	0,15	0,20	0,25	0,30	0,35	0,40
0,5	0,98	0,94	0,89	0,87	0,85	0,84	0,83
1,0	0,98	0,92	0,88	0,87	0,85	0,84	0,83
1,5	0,97	0,90	0,86	0,85	0,83	0,82	0,81
2,0	0,94	0,87	0,83	0,82	0,80	0,78	0,78
2,5	0,88	0,80	0,78	0,77	0,75	0,73	0,72
3,0	0,79	0,72	0,71	0,69	0,67	0,66	0,65
3,5	0,72	0,66	0,64	0,62	0,60	0,58	0,57

Tabla 24: Tasa de *falsos negativos* (FN): *Vlachos* - *on-line* (Enero de 2012).

x	r						
	0,10	0,15	0,20	0,25	0,30	0,35	0,40
0,5	0,01	0,05	0,10	0,12	0,14	0,15	0,16
1,0	0,01	0,07	0,11	0,12	0,14	0,15	0,16
1,5	0,02	0,09	0,13	0,14	0,16	0,17	0,18
2,0	0,05	0,12	0,16	0,17	0,19	0,21	0,21
2,5	0,11	0,19	0,21	0,22	0,24	0,26	0,27
3,0	0,20	0,27	0,28	0,30	0,32	0,33	0,34
3,5	0,28	0,33	0,35	0,37	0,39	0,41	0,42

Tabla 25: Tasa de *falsos positivos* (FP): *on-line* - *Vlachos* (Enero de 2012).

x	r						
	0,10	0,15	0,20	0,25	0,30	0,35	0,40
0,5	0,002	0,001	0,001	0,001	0,001	0,001	0,001
1,0	0,002	0,001	0,001	0,001	0,001	0,001	0,001
1,5	0,002	0,001	0,001	0,001	0,001	0,001	0,001
2,0	0,002	0,001	0,001	0,001	0,001	0,001	0,001
2,5	0,002	0,001	0,001	0,001	0,001	0,001	0,001
3,0	0,002	0,001	0,001	0,001	0,001	0,001	0,001
3,5	0,002	0,001	0,001	0,001	0,001	0,001	0,001

Tabla 26: *Precision*: VP / (VP + FP) (Enero de 2012).

x	r						
	0,10	0,15	0,20	0,25	0,30	0,35	0,40
0,5	1,00	1,00	1,00	1,00	1,00	1,00	1,00
1,0	1,00	1,00	1,00	1,00	1,00	1,00	1,00
1,5	1,00	1,00	1,00	1,00	1,00	1,00	1,00
2,0	1,00	1,00	1,00	1,00	1,00	1,00	1,00
2,5	1,00	1,00	1,00	1,00	1,00	1,00	1,00
3,0	1,00	1,00	1,00	1,00	1,00	1,00	1,00
3,5	1,00	1,00	1,00	1,00	1,00	1,00	1,00

Tabla 27: *Recall*: VP / (VP + FN) (Enero de 2012).

x	r						
	0,10	0,15	0,20	0,25	0,30	0,35	0,40
0,5	0,98	0,94	0,89	0,87	0,86	0,85	0,84
1,0	0,98	0,93	0,89	0,87	0,86	0,84	0,84
1,5	0,97	0,90	0,87	0,86	0,84	0,82	0,82
2,0	0,95	0,87	0,84	0,82	0,80	0,79	0,78
2,5	0,89	0,81	0,78	0,77	0,75	0,74	0,73
3,0	0,80	0,73	0,71	0,70	0,68	0,66	0,65
3,5	0,72	0,66	0,64	0,62	0,61	0,59	0,58

Considerando los valores expuestos anteriormente, dos métricas posibles para evaluar clasificación binaria son *Precision* y *Recall* (o sensibilidad). El primero sirve como una medida de exactitud o calidad, mientras que el segundo como una medida de completitud o cantidad. En el presente contexto, el *Precision* mide la calidad de las consultas detectadas como ráfagas, mientras que el *Recall* mide la capacidad del método para detectar consultas en ráfaga. Estos indicadores están definidos como:

$$Precision = \frac{VP}{VP + FP} \qquad Recall = \frac{VP}{VP + FN}$$

Ambos indicadores varían entre 0 y 1, con 1 siendo el mejor valor. En la Tabla 26 se expone el *Precision* del método propuesto, mientras que en la Tabla 27 se observa el *Recall*.

El método de Vlachos *et al.* da como salida un conjunto de consultas en ráfaga

Tabla 28: *Weighted Recall* (Enero de 2012).

x	r						
	0,10	0,15	0,20	0,25	0,30	0,35	0,40
0,5	0,99	0,98	0,96	0,92	0,69	0,58	0,52
1,0	0,99	0,98	0,95	0,92	0,63	0,56	0,51
1,5	0,99	0,96	0,91	0,84	0,60	0,52	0,47
2,0	0,98	0,95	0,90	0,76	0,56	0,47	0,45
2,5	0,97	0,90	0,81	0,72	0,49	0,44	0,42
3,0	0,95	0,86	0,56	0,50	0,43	0,41	0,41
3,5	0,94	0,77	0,49	0,44	0,40	0,39	0,38

considerando información histórica. Este conjunto de consultas sirve como una referencia para considerar qué es ráfaga y qué no. Como se ha mencionado, no existe un método estándar ni aceptado para determinar este conjunto. Considerando este conjunto, en la Tabla 25 y 26 (*Precision*) se observa que el método propuesto tiene la capacidad de detectar muy pocos falsos positivos, es decir, casi la totalidad de las consultas categorizadas como ráfagas lo son. Por otro lado, se observa en la Tabla 27 (*Recall*) que a medida que se aumentan las cotas de detección, se reduce el número de consultas detectadas como ráfagas. Se cree que los valores para una detección eficiente varían para x desde 0,5 a 2,0, y para r desde 0,10 a 0,20. Con esos valores se tendrá una alta cantidad de consultas detectadas como ráfagas. Para evidenciar aún más la contribución de los resultados anteriores, en la Tabla 28 se determina el *Weighted Recall* definido como:

$$Weighted\ Recall = \frac{\sum_{q\ \in Vlachos\ \cap\ on-line} w_q}{\sum_{q\ \in Vlachos} w_q}$$

En vez de sólo contar el número de consultas como en *Recall*, el *Weighted Recall* incluye el volumen de cada consulta (w_q o frecuencia de q), por lo que es posible observar cuál es el impacto, en términos de volumen, de las consultas declaradas como ráfaga por el método propuesto. Esto reafirma lo enunciado anteriormente con respecto a los mejores valores para x y r.

Las tablas presentadas anteriormente muestran la factibilidad de realizar detección de consultas en ráfaga con el método propuesto. A continuación, se exhibirán

algunas consultas y en qué instante el método realizó la detección. Se realizará una comparación del método propuesto con la estrategia de Vlachos *et al.* con respecto al momento en que son declaradas ráfagas. Aún así, esto no tiene tanta importancia, ya que son enfoques completamente distintos. En este trabajo se detectan las consultas en ráfaga con espacio reducido y en forma eficiente, sólo considerando información acotada de las series de tiempo de las consultas. La comparación será con fines de mostrar evidencia.

Para seleccionar las consultas, se consideraron tres aspectos. La idea es seleccionar consultas distintas en cada aspecto y con características que se obtengan sólo de las series de tiempo.

1. Alta Frecuencia: se selecciona un conjunto de tres consultas que experimentaron las frecuencias más altas: 'khloe kardashian', 'demi moore' y 'kim kardashian'.

2. Alta Desviación Estándar: tres consultas tal que tuvieron la más alta desviación estándar en las series de tiempo: 'cher not dead', 'nicole brown simpson' y 'kristy mcnichol'.

3. Alto Coeficiente de Variación: conjunto de tres consultas tal que tuviesen el coeficiente de variación más alto (mayor que 1,0): 'emmy rossum', 'jay z' y 'yeti crabs'.

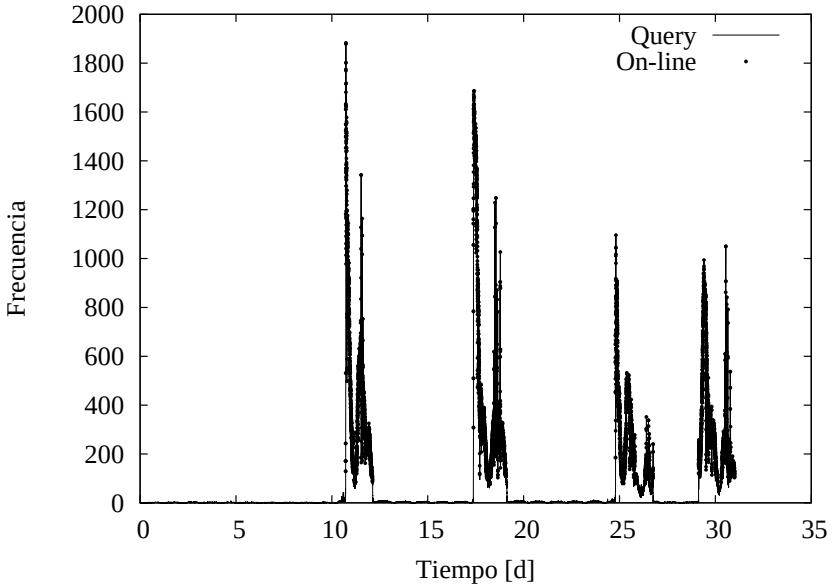
7.5.1. Alta Frecuencia

En la Figura 41 se observa la primera consulta seleccionada por su alta frecuencia ('khloe kardashian'). Se observa que la Figura 41(a) tiene múltiples episodios en que se torna ráfaga durante el período de tiempo. En la Figura 41(b) se tiene un ajuste al período de ocurrencia del primer episodio en ráfaga de la misma consulta. Cada uno de los puntos indica que el método propuesto categoriza la consulta como ráfaga. Es posible observar también que el objetivo de detectar tempranamente este tipo de eventos, antes de que alcancen el punto máximo, se cumple. El punto máximo visible en este episodio, ocurre en el minuto 15.452 (cerca de $x = 10,7$) alcanzando una frecuencia máxima por minuto de $y = 1.883$. El método propuesto categoriza la

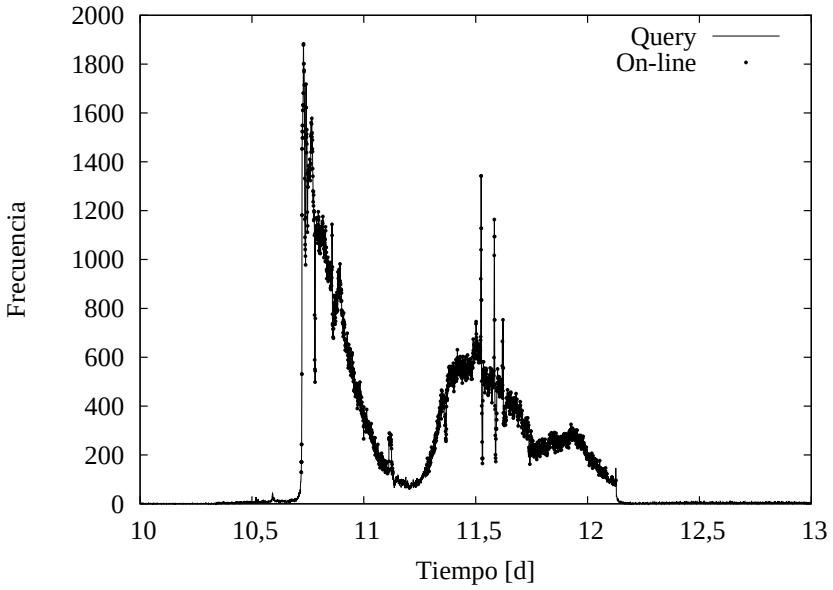
consulta como ráfaga (primer punto visible) en el minuto 15.437, cuando la consulta tiene una frecuencia por minuto de $y = 129$. Esto implica que el método categoriza la consulta como ráfaga 15 minutos antes de que alcance el punto máximo. Esto es tiempo suficiente para tomar acciones mitigatorias en el Servicio de Cache. Se debe considerar que esta consulta es una de las que presenta más alta frecuencia por minutos de las detectadas como ráfaga, por lo que mientras más temprana es la detección, más grandes los beneficios. Según la información de los experimentos, el método de Vlachos *et al.* categoriza la consulta como ráfaga en el minuto 15.444, cuando se tiene una frecuencia por minuto de $y = 1.524$. Es decir, la propuesta se adelanta en 7 minutos. Esto evidencia que la detección en base a una ventana de tiempo es factible, y en algunos casos tiene mejor desempeño.

En la Figura 42(a) se observa la frecuencia de la consulta 'demi moore' y los puntos de detección de la consulta como ráfaga. En la Figura 42(b) se muestra el período correspondiente al primer episodio en ráfaga. Nuevamente se observa una detección temprana, pero no tanto como en la consulta anterior. Esto indica que la detección con nuestra propuesta varía de consulta a consulta. En este caso, la detección se hace en el minuto 3.369 (aproximadamente en $x = 2,3$) cuando la frecuencia por minuto es $y = 540$. Existe un primer punto máximo en el minuto 3.375, y un segundo punto máximo en el minuto 3.389. Esto implica que la detección se hace en 6 y 20 minutos antes respectivamente. Esta consulta presenta un comportamiento diferente a la consulta anterior, ya que aquella tuvo un alza abrupta desde 0 al punto máximo, sin "descansos". Por el contrario, la consulta actual presenta puntos máximos intermedios (en el minuto 3.375 y 3.389) antes de llegar al punto máximo real del episodio. Este punto máximo está en el minuto 3.457, lo que implica que el método detecta la consulta 88 minutos antes de este punto máximo. Finalmente, el método declara como ráfaga a la consulta 3 minutos antes que el método *off-line*.

En la Figura 43 se observa la frecuencia de la consulta 'kim kardashian' y una reducción del intervalo de tiempo. El método propuesto categoriza la consulta como ráfaga 9 minutos antes del primer punto máximo y 20 minutos antes que el máximo global. Por otro lado, el método propuesto declara la consulta como ráfaga 3 minutos antes que Vlachos *et al.*.

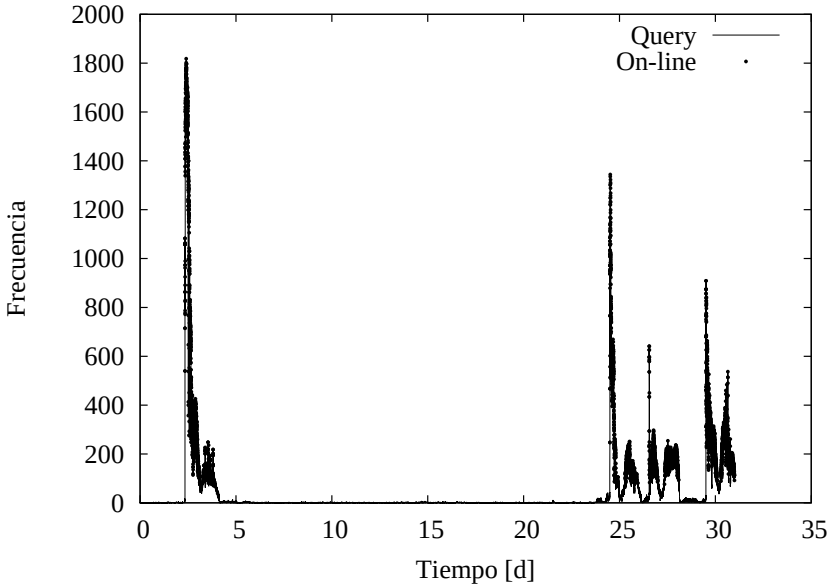


(a)

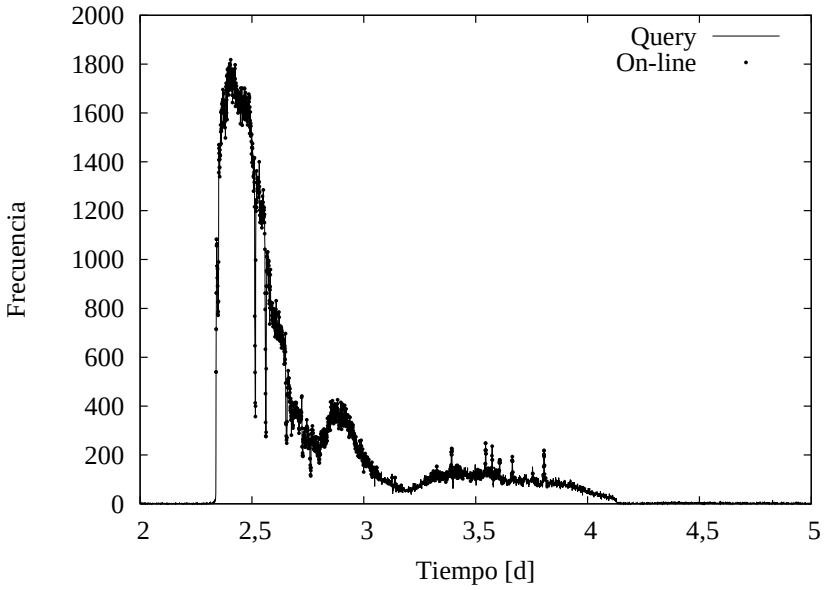


(b)

Figura 41: Frecuencia absoluta de consulta 'khloe kardashian' durante Enero de 2012: (a) período completo; (b) período acotado al primer episodio ráfaga. La línea continua representa la frecuencia absoluta, los puntos sobrepuestos el indicador de nuestra propuesta para ráfagas.

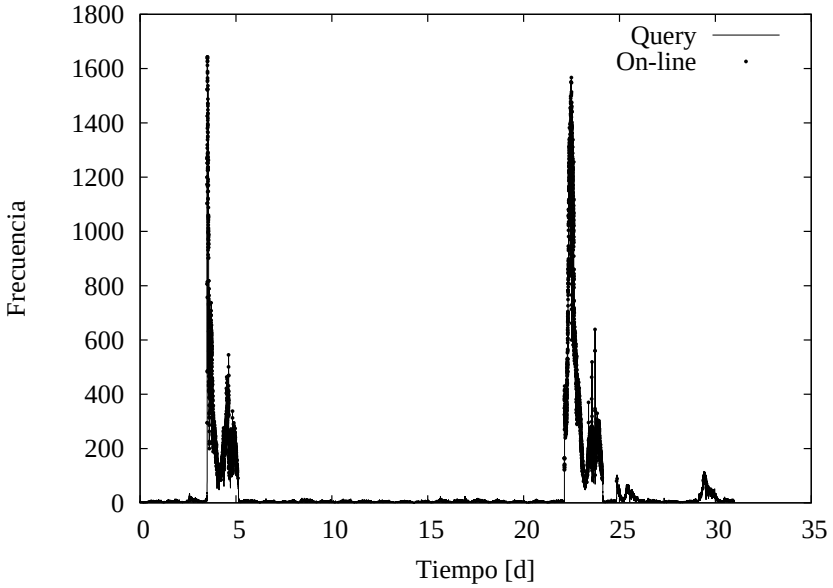


(a)

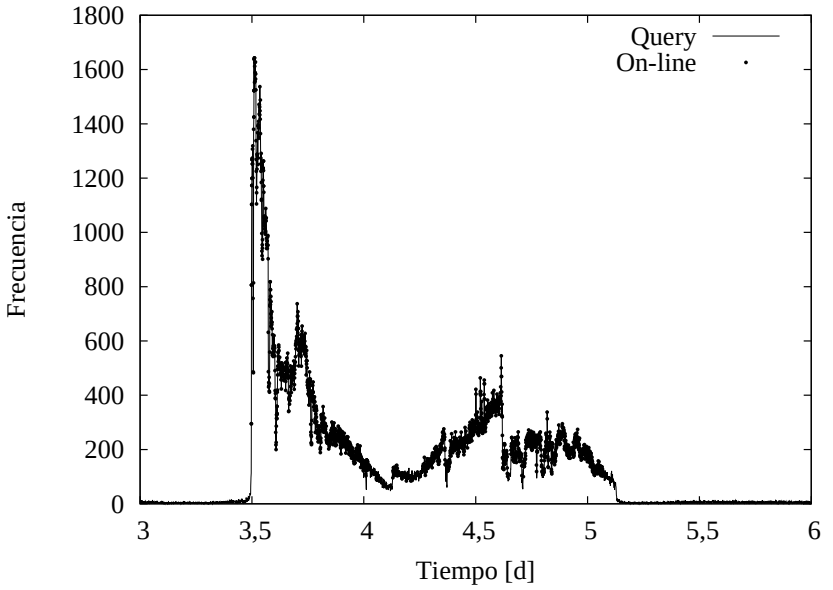


(b)

Figura 42: Frecuencia absoluta de consulta 'demi moore' durante Enero de 2012: (a) período completo; (b) período acotado al primer episodio ráfaga. La línea continua representa la frecuencia absoluta, los puntos sobrepuestos el indicador de nuestra propuesta para ráfagas.



(a)



(b)

Figura 43: Frecuencia absoluta de consulta 'kim kardashian' durante Enero de 2012: (a) período completo; (b) período acotado al primer episodio ráfaga. La línea continua representa la frecuencia absoluta, los puntos sobrepuestos el indicador de nuestra propuesta para ráfagas.

7.5.2. Alta Desviación Estándar

Una alta desviación estándar implica alta variación de los datos en torno al promedio, lo que a su vez podría indicar un comportamiento en ráfaga en un intervalo de tiempo muy acotado (alza abrupta). La primera consulta 'cher not dead' se encuentra en la Figura 44(a), así como la ampliación del episodio en ráfaga en la Figura 44(b). Se observa una alza abrupta, incluso alcanzando una frecuencia mayor a las consultas analizadas anteriormente. La consulta es declarada ráfaga en el minuto 38.020 (cerca de $x = 26, 4$), 3 minutos antes que el método *off-line*, 13 minutos antes que un primer punto máximo (en el minuto 38.933), y 31 minutos antes que el punto con mayor frecuencia (en el minuto 38.051).

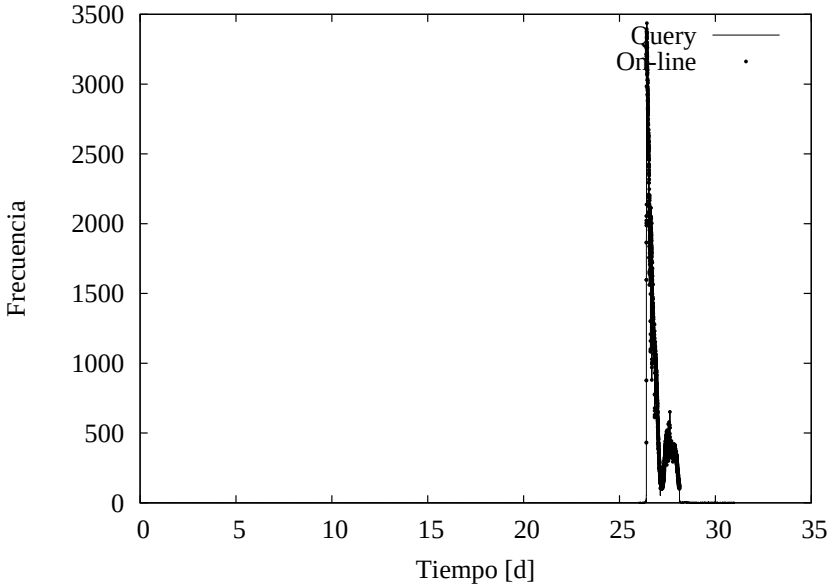
La Figura 45 muestra los resultados para la consulta 'nicole brown simpson'. En resumen, se detecta como ráfaga en el minuto 32.559 (cerca de $x = 22, 6$) y 8 minutos antes que el método de Vlachos *et al.*, 15 minutos antes que el primer punto máximo (en el minuto 32.573), y 38 minutos antes que el punto máximo (en el minuto 32.597). Finalmente, en la Figura 46, la consulta 'kristy mcnichol' es declarada como ráfaga en el minuto 12.244 (cerca de $x = 8, 5$), 5 minutos antes que el método *off-line*, 10 minutos antes que el primer punto máximo (en el minuto 12.254) y 25 minutos antes que el punto máximo global (en el minuto 12.269).

De los resultados descritos anteriormente, se concluye que una alta desviación estándar es un indicador importante en la detección, ya que según los datos anteriores, permite una detección más temprana de las consultas.

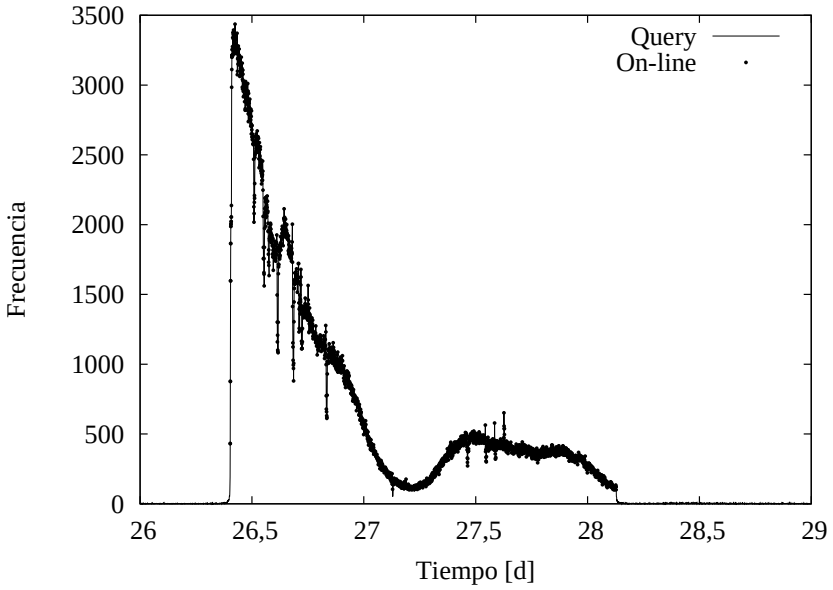
7.5.3. Alto Coeficiente de Variación

Un alto coeficiente de variación implica alta variación de las frecuencias de la serie de tiempo en comparación con el valor promedio de la serie. En la Figura 47 se observa que, con la consulta 'emmy rossum', se tiene una detección no tan temprana, sólo 3 minutos antes del punto máximo (1 minuto antes que el método *off-line*), cerca del punto $x = 4, 4$. Esto es debido a la abrupta alza de las frecuencias que experimenta la consulta.

Nuevamente en la Figura 48 se observa una abrupta alza para la consulta 'jay z',

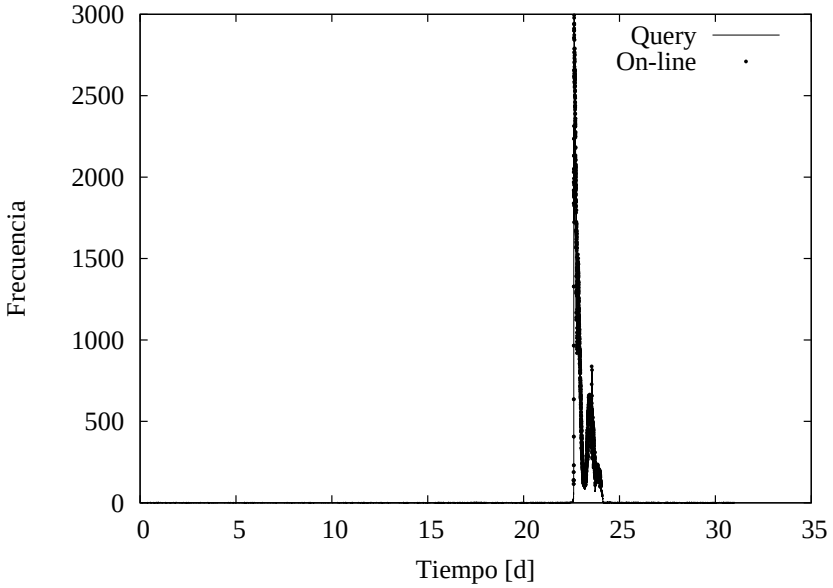


(a)

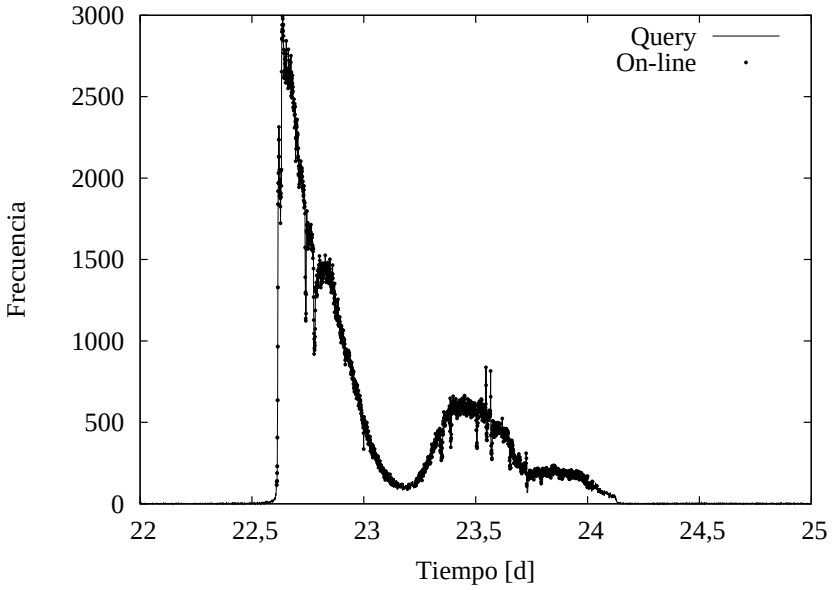


(b)

Figura 44: Frecuencia absoluta de consulta 'cher not dead' durante Enero de 2012: (a) período completo; (b) período acotado al episodio ráfaga. La línea continua representa la frecuencia absoluta, los puntos sobrepuestos el indicador de nuestra propuesta para ráfagas.

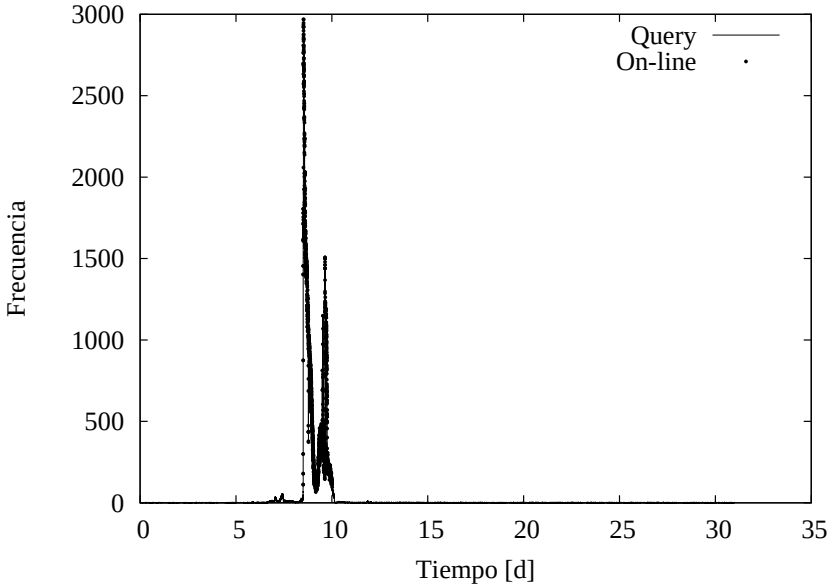


(a)

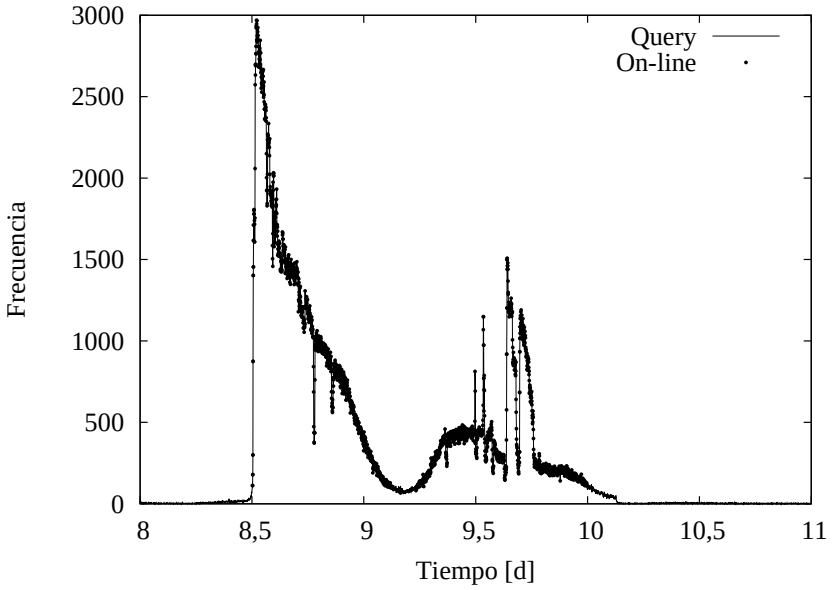


(b)

Figura 45: Frecuencia absoluta de consulta 'nicole brown simpson' durante Enero de 2012: (a) período completo; (b) período acotado al episodio ráfaga. La línea continua representa la frecuencia absoluta, los puntos sobrepuestos el indicador de nuestra propuesta para ráfagas.



(a)



(b)

Figura 46: Frecuencia absoluta de consulta 'kristy mcnichol' durante Enero de 2012: (a) período completo; (b) período acotado al episodio ráfaga. La línea continua representa la frecuencia absoluta, los puntos sobrepuestos el indicador de nuestra propuesta para ráfagas.

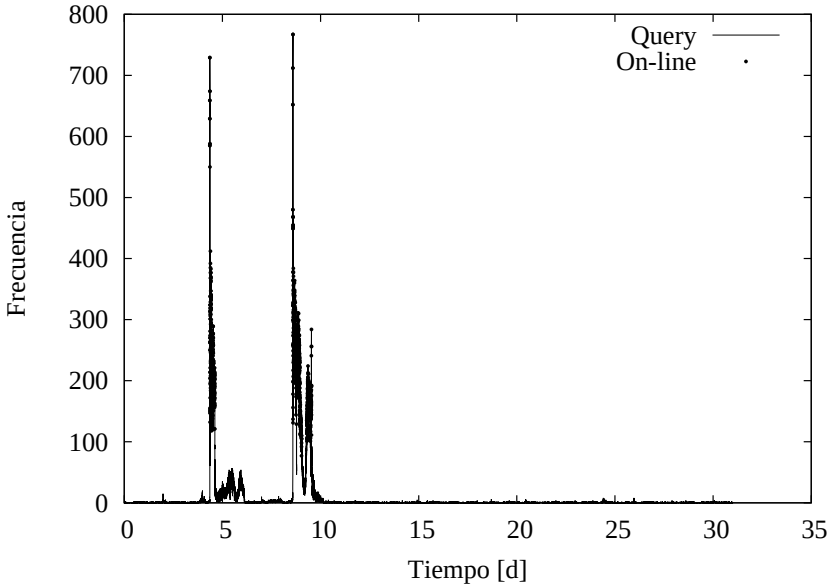
declarándola como ráfaga 5 minutos antes que el primer punto máximo y 9 minutos antes que el segundo punto máximo (en el minuto 190 y 194 respectivamente, cerca del punto $x = 0,1$). Se detecta 3 minutos antes que el método *off-line*. El mismo comportamiento de alza abrupta se observa en la Figura 49 (consulta 'yeti crabs'), en el que la propuesta detecta la consulta en ráfaga en el minuto 4.887 y alcanza su valor máximo en el minuto 4.895 (aproximadamente en $x = 3,4$), es decir, 8 minutos antes. Por otro lado, esta detección se realiza 1 minuto antes que Vlachos *et al.*.

Un alto coeficiente de variación implica una alza muy abrupta de las frecuencias según los datos observados. Aún así, el método propuesto permite adelantarse unos minutos a la aparición de los puntos máximos de las frecuencias.

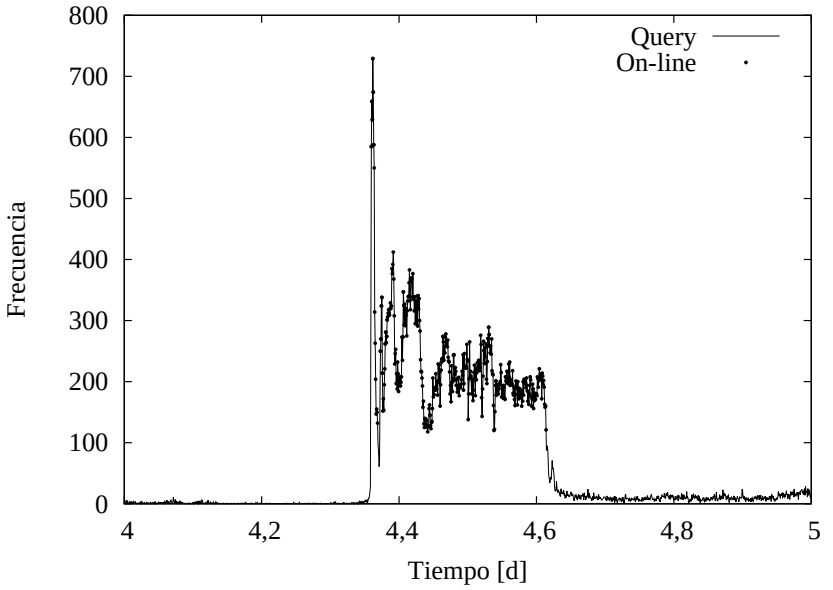
7.6. Evaluación Experimental

Los episodios que generan consultas en ráfaga suceden ocasionalmente. El método propuesto permite detectar tempranamente las consultas en ráfaga para evitar alzas en la utilización de un conjunto de nodos. Para mostrar la diferencia en términos de desempeño de la implementación de la propuesta, se han escogido dos episodios importantes dentro de los *logs* utilizados. El primer evento corresponde a la muerte de Osama Bin Laden (2 de Mayo de 2011), y el segundo corresponde a la muerte de Whitney Houston (11 de Febrero de 2012). Estos casos corresponden a episodios en que, como se verá, los mecanismos de balance de carga dinámico propuestos anteriormente no son suficientes para cubrirlos. Para evaluar el desempeño, se utilizarán consultas cercanas a la fecha de ocurrencia de estos eventos, ya que en un contexto de normalidad (sin consultas en ráfaga), casi no se aprecian cambios al implementar la estrategia propuesta.

En la Figura 50(a) se observa la utilización de las particiones con las estrategias propuestas (distribución de ítemes frecuentes y balance de carga). En la figura descrita, se observan tres episodios importantes de alzas de utilización en un conjunto de particiones. Esto se debe a que una consulta frecuente es distribuida en múltiples nodos. Es importante notar además que, según estos resultados, las estrategias de

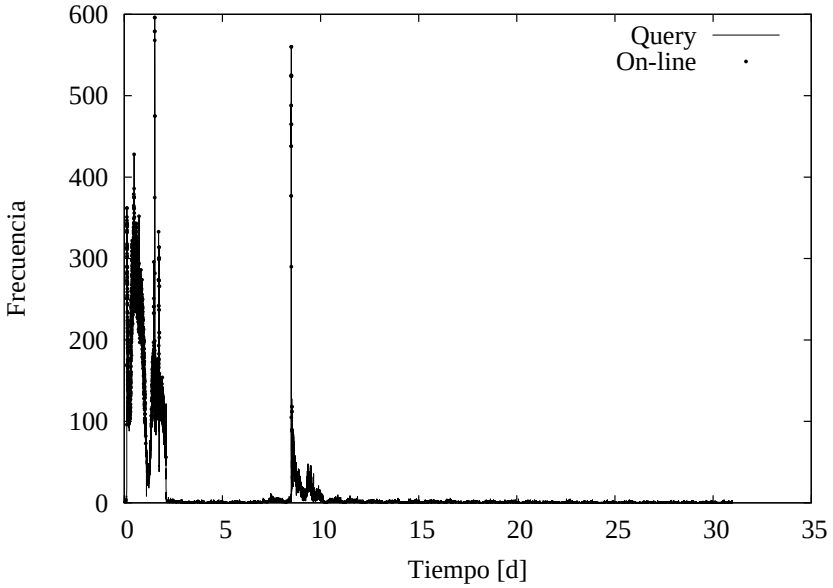


(a)

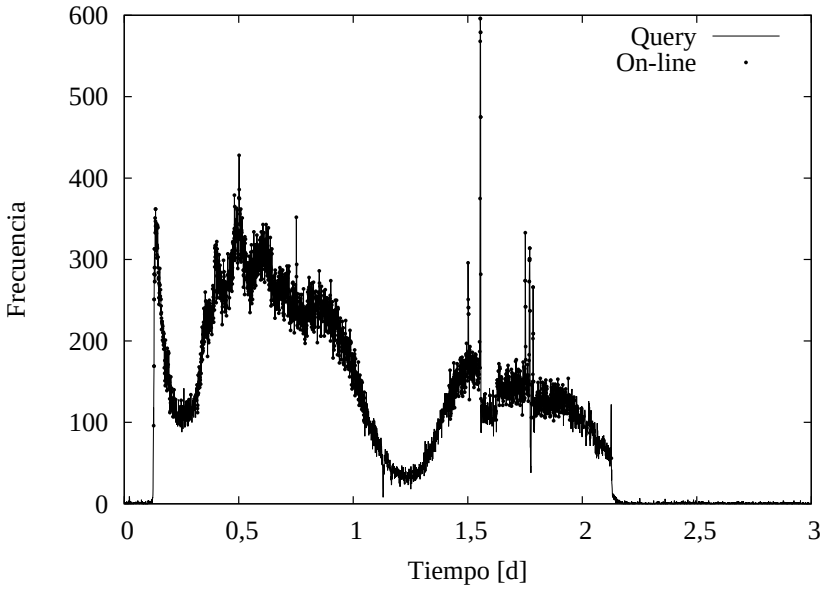


(b)

Figura 47: Frecuencia absoluta de consulta 'emmy rossum' durante Enero de 2012: (a) período completo; (b) período acotado al primer episodio ráfaga. La línea continua representa la frecuencia absoluta, los puntos sobrepuestos el indicador de nuestra propuesta para ráfagas.

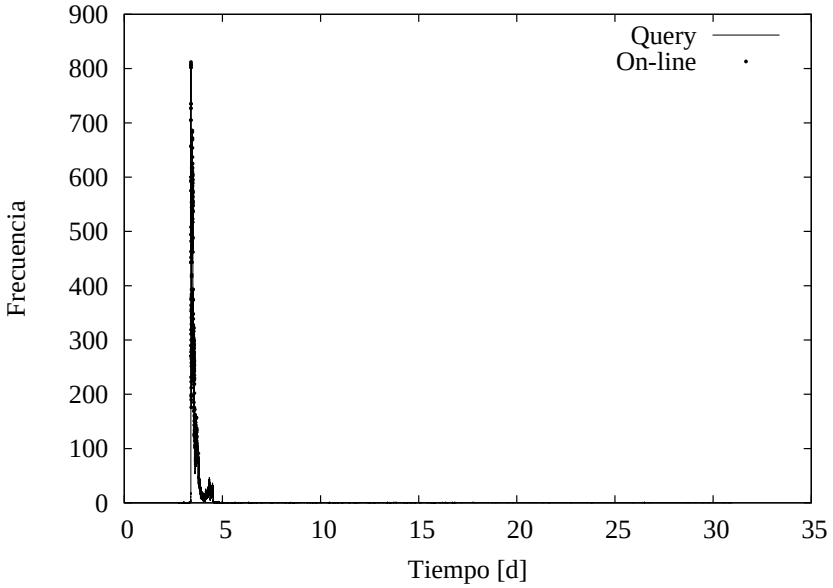


(a)

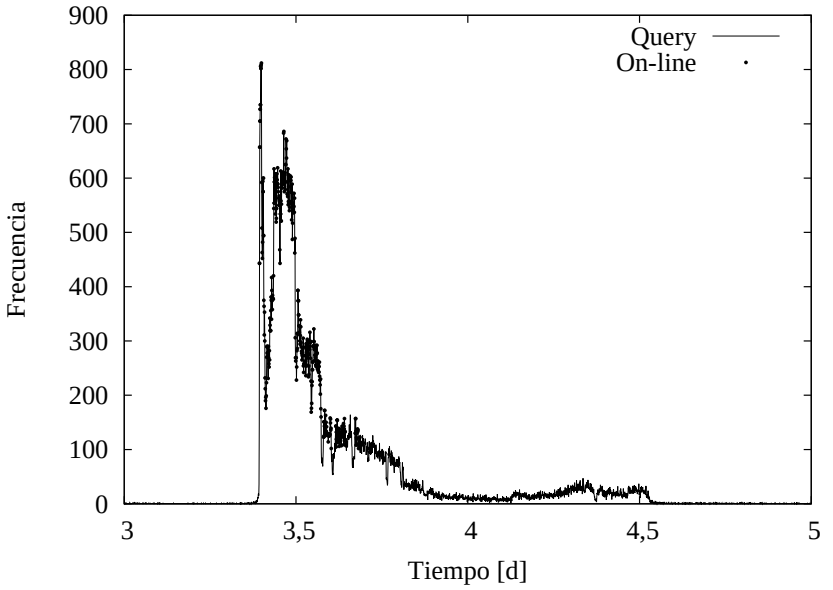


(b)

Figura 48: Frecuencia absoluta de consulta 'jay z' durante Enero de 2012: (a) período completo; (b) período acotado al primer episodio ráfaga. La línea continua representa la frecuencia absoluta, los puntos sobrepuestos el indicador de nuestra propuesta para ráfagas.



(a)



(b)

Figura 49: Frecuencia absoluta de consulta 'yeti crabs' durante Enero de 2012: (a) período completo; (b) período acotado al episodio ráfaga. La línea continua representa la frecuencia absoluta, los puntos sobrepuestos el indicador de nuestra propuesta para ráfagas.

balance de carga funcionan en balancear la carga a largo plazo, pero no logran impedir estas alzas abruptas. Como fue descrito en los capítulos anteriores, la idea de las estrategias de balance dinámico no era actuar en casos de alzas abruptas, sino que actuar iterativamente hasta conseguir la carga promedio. Según los experimentos, la consulta 'osama bin laden' (y algunas variaciones) comienza a posicionarse dentro de las consultas más frecuentes en la estructura de monitorización alrededor del valor $x = 0,9$. Posteriormente, en los valores $x = 1,4$, $x = 1,5$ y $x = 1,6$ aproximadamente aparecen otras consultas relacionadas con el tópico en cuestión. En la Figura 50(b) se observa la misma situación, pero con la estrategia de detección de consultas en ráfaga. Se observa la disminución de las alzas descritas anteriormente, mediante la distribución de este tipo de consultas en más nodos. Esto hace que la detección de este tipo de consultas deba ser considerado en el diseño de servicios de cache eficientes, con el fin de prevenir alzas abruptas en la utilización de un conjunto de nodos. Para hacer más clara la diferencia, en la Figura 50(c) y (d) se muestra la eficiencia de la situación sin y con detección de consultas en ráfaga respectivamente.

Las acciones mitigatorias ante consultas en ráfaga tienen efectos colaterales positivos, ya que a partir de los datos en ese intervalo de tiempo se tiene: (i) una reducción desde 30,4 a 29,8 milisegundos (2 % de reducción); (ii) un aumento de la eficiencia de 87,9 % a 89,8 % (un 1,9 % más); y (iii) la reducción de 6 nodos que presentan saturación a 0. Si bien es cierto que estas mejoras no son cuantiosas, ellas contribuyen a mantener un Servicio de Cache balanceado y que es capaz de adelantarse a alzas abruptas de tráfico. Se debe recordar además que estas mejoras son sobre el esquema de balance de carga dinámico propuesto en los capítulos anteriores.

Siguiendo la misma idea anterior, en la Figura 51 se muestran los resultados para un conjunto de tópicos, siendo el más importante la muerte de Whitney Houston. Las consultas relacionadas a este último evento[1], comienzan a ser frecuentes aproximadamente en el valor $x = 10,8$, por lo cual al inicio de la Figura 51(a) se observan algunas alzas alrededor de este valor. Posteriormente, se observan algunas alzas alrededor de los puntos $x = 11,8$ y $x = 12,5$. El primer punto está relacionado con los tópicos

[1] 'whitney houston', 'whitney houston dead', 'whitney houston news', 'whitney houston died', 'whitney houston death', 'whitney houston dies' y 'did whitney houston die'.

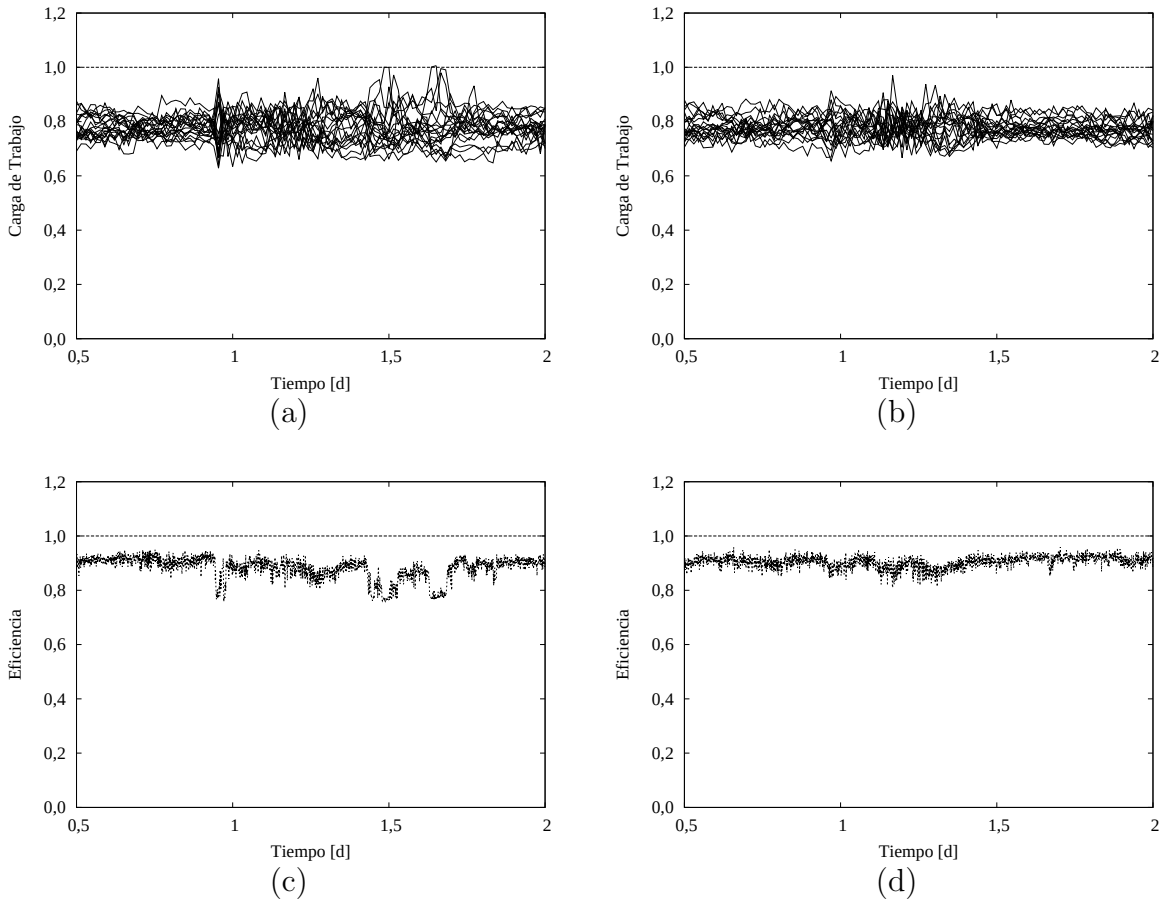


Figura 50: Resultados para primer tópico en un Servicio de Cache de 20 nodos y 100 mil entradas por nodo: (a) utilización sin detección; (b) utilización con detección; (c) eficiencia sin detección; y (d) eficiencia con detección.

'bobbi kristina' y 'bobby brown', mientras que el segundo punto está relacionado con el tópico '336 million powerball jackpot'. Cabe señalar que en este último punto, aún las consultas relacionas a los tópicos del primer punto son categorizadas como ráfagas. Es posible observar en la Figura 51(b) que estas alzas son atenuadas mediante la detección de las consultas que las generan. En la Figura 51(c) y (d) se muestra la eficiencia en ambos casos. La eficiencia promedio es 88,7 % y 90,3 %, respectivamente, mientras que no existe saturación de nodos en ambos casos.

Como ejemplo de la detección se han expuesto las consultas con más impacto,

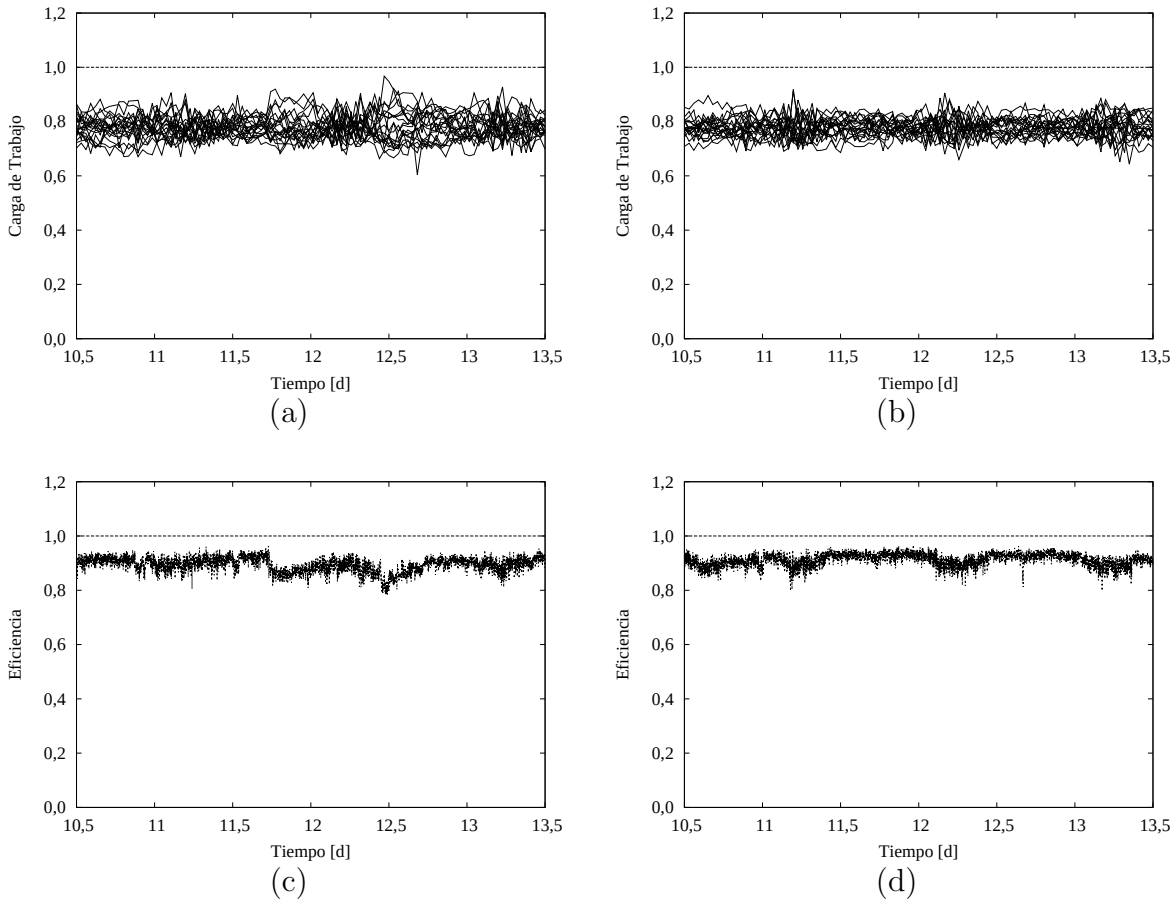


Figura 51: Resultados para segundo tópico en un Servicio de Cache de 20 nodos y 100 mil entradas por nodo: (a) utilización sin detección; (b) utilización con detección; (c) eficiencia sin detección; y (d) eficiencia con detección.

pero cabe señalar que además de estas consultas existen algunas con menor impacto que son detectadas y declaradas como ráfagas. Como se mencionó anteriormente, el objetivo de esta propuesta no es balancear más la carga de trabajo, sino detectar tempranamente eventos atípicos y realizar acciones mitigatorias para que las alzas asociadas no generen condiciones de desbalance y saturación de nodos. Además, según la evidencia empírica, el método funciona al declarar como ráfaga un conjunto de consultas (*string*) en torno a un tópico en común.

7.7. Conclusiones

Las consultas en ráfaga tienen un alto impacto en los servicios de cache si es que no son tratadas de manera distinta a las demás consultas. Esto es debido a su alta frecuencia en un período acotado de tiempo y al uso estándar que tiene Consistent Hashing para hacer el mapeo de consultas a nodos. Por otro lado, no existe un método aceptado estándar ni un método óptimo para la detección de estas consultas, sino que la mayoría de los métodos deben ser analizados para el área en particular y sus parámetros deben ser ajustados para ello. En particular, el trabajo de Vlachos *et al.* [VMVG04] permite categorizar una consulta como ráfaga en base a información histórica, lo que se torna imposible en un sistema de alto volumen de consultas como los motores de búsqueda Web. Aún así, este método *off-line* fue utilizado como el generador de un conjunto de consultas que sirve como base de comparación.

Este trabajo utilizó la infraestructura definida en el capítulo 5 para construir un método compacto en espacio y eficiente en tiempo de ejecución, lo cual es un giro respecto al trabajo mencionado anteriormente. En ese mismo capítulo se mencionó que esta infraestructura sí está disponible en grandes aplicaciones Web. La idea es que por cada consulta de la estructura de monitorización, se establezca una serie de tiempo acotada, que permita conocer su historia reciente como consulta. Distintas clases de consultas tienen distinta historia, por lo que se ideó un método para declarar una consulta como ráfaga utilizando la misma información de la serie de tiempo y estadísticas matemáticas sobre esta secuencia. Además de lo mencionado, otra bondad del método propuesto es que es adaptable al volumen de tráfico, ya que el impacto de una consulta varía según el tráfico observado. Esta propuesta permite el balance de corto plazo.

Desde el punto de vista del conjunto declarado como ráfaga, la propuesta de este trabajo de tesis logra buenos resultados con métricas como *Precision* y *Recall*. Además, según la evidencia expuesta en este capítulo, las consultas son declaradas como ráfagas antes que el método *off-line* (aunque sean enfoques completamente distintos y el método *off-line* cuente con mucha más información para realizar la identificación de consultas en ráfaga) y, lo más importante, son declaradas como tal

con suficiente antelación con el fin de implementar medidas mitigatorias en el Servicio de Cache tales como realizar la replicación pro-activa de estas consultas. El fin de un sistema en tiempo real de detección temprana de consultas en ráfaga es mitigar los efectos de estas consultas distribuyéndolas en más de un nodo (la misma idea del capítulo 5). Así el impacto de estas consultas en un solo nodo es reducido.

El Servicio de Cache no es un sistema crítico, es decir, identificar una consulta como ráfaga cuando realmente no lo es (falso positivo), no tiene efectos colaterales que perjudiquen el desempeño del servicio (sólo una despreciable reducción en el número de *hits* y aumento de comunicación para replicar unas pocas consultas pro-activamente). Por otro lado, la identificación correcta de una consulta en ráfaga previene la congestión de un conjunto de máquinas y por ende reduce los problemas de balance en un Servicio de Cache.

Por otra parte, no todas las consultas en ráfaga tienen el mismo impacto. El trabajo de Vlachos *et al.* [VMVG04] categoriza las consultas como ráfaga y no ráfaga. Este trabajo establece niveles para este tipo de consultas, con el fin de una consulta tenga características de ráfagas más alta sea tratada de forma distinta que una que tenga más reducidas estas características. Analizando las métricas utilizadas, se tiene que el coeficiente de variación móvil es independiente de la frecuencia actual de la consulta y tiene un rango desde 0 a 2 (al contrario de la cota relacionada con la frecuencia que varía dependiendo de x). Se puede usar este valor para definir distintos niveles de ráfagas. Si este rango se divide por ejemplo en 4 rangos, entonces una consulta que pertenezca a la primera categoría [0,5-1,0) necesitaría menos nodos a ser distribuidos que una consulta que esté en la tercera categoría [1,0-1,5).

Con respecto a la eficiencia de la técnica propuesta, hay que notar que en momentos en que no se declaran consultas como ráfagas, con esta estrategia no existen mejoras al balance dinámico planteado en los capítulos anteriores. Las mejoras se presentan en la presencia de consultas en ráfaga, lo cual fue validado con los experimentos en un Servicio de Cache. Las mejoras en estos experimentos fueron marginales, alrededor de un 2 %, pero lo que se debe recalcar es que el método sirvió para reducir el desbalance y para eliminar los servidores congestionados ante estos eventos. Ese

Tabla 29: Consulta y cantidad de minutos antes del primer máximo por método *on-line* y *off-line* (Enero de 2012).

Minutos		Query
On-line	*Off-line*	
18	8	'nicole brown simpson'
14	7	'john 3 16'
11	6	'blue ivy godmother'
11	5	'bill murray'
11	4	'kim kardashian'
11	11	'cher not dead'
10	6	'vanessa paradis'
10	5	'katy perry'
9	9	'mariah carey'
9	7	'william shatner'
9	6	'camille grammer'
9	2	'sausage and cancer'

es el fin principal perseguido en este capítulo, ya que el mecanismo propuesto (balance de corto plazo) en conjunto con el balance de mediano y largo plazo, permite implementar un Servicio de Cache menos proclive a desbalance y saturación ante las condiciones generadas por las consultas de usuario.

El número de consultas detectadas o métricas como *Precision/Recall* no son las únicas características importantes, ya que la idea principal de este trabajo es declarar una ráfaga como tal lo antes posible. Para este análisis se considera el tiempo promedio en minutos en que cada método declara una ráfaga antes del primer punto máximo (alza sostenida hasta una disminución de frecuencia). Cabe señalar que en gran parte de los casos, este primer máximo no representa el punto máximo sino que una especie de descanso hasta encontrar el punto máximo del episodio ráfaga. Estos resultados fueron calculados para las consultas detectadas como ráfagas y sólo para el primer episodio experimentado por cualquier consulta:

- Vlachos *et al.*: 3,2 minutos antes del primer máximo.
- Propuesta: 3,7 minutos antes del primer máximo.

Para complementar lo anterior, en la Tabla 29 se exponen las consultas con mayor impacto y la cantidad de minutos en que se declaró como ráfaga para ambos métodos estudiados. Con los resultados anteriores, se tiene que la propuesta realiza una detección más temprana (13,5 % en promedio), y en casos con alto impacto el método propuesto detecta las consultas en ráfaga con bastante anticipación. Con esto se concluye que la estrategia propuesta presenta un mejor desempeño que el método *off-line*, utilizando una infraestructura eficiente y con espacio limitado, y que además permite mitigar las alzas abruptas de carga en un Servicio de Cache.

Tolerancia a Fallas

En este capítulo se propone un esquema que permite mitigar el efecto de los fallos en un Servicio de Cache (CS). La idea principal es replicar en forma proactiva un grupo de respuestas pre-computadas en nodos de respaldo.

En la sección 8.1 se detalla el problema producido por las fallas de nodos y las opciones para proveer tolerancia a fallas. Junto a ello, se explican las desventajas de realizar replicación completa de nodos, como en un esquema matricial. En la sección 8.2 se detalla el mecanismo propuesto que consiste en la replicación controlada de información relevante en nodos de respaldo. Así, ante la ocurrencia de fallos, las peticiones del nodo caído son re-dirigidas al nodo de respaldo. La información replicada tiene relación con las consultas frecuentes. En la sección 8.3 se enuncian los algoritmos diseñados y los distintos parámetros para el funcionamiento de la propuesta. Posteriormente, en la sección 8.4 se realiza una evaluación experimental acotada para medir el costo adicional de la propuesta sobre otras estrategias base. Finalmente, en la sección 8.5 se establecen las conclusiones del capítulo.

8.1. Motivación

El Servicio de Cache (CS) almacena información relevante para el procesamiento de consultas, ya que cada par $\langle consulta, respuesta \rangle$ implica la utilización previa de recursos computacionales (CPU, red, disco). Esta utilización de recursos computacionales se produce en gran medida en el Servicio de Índice (IS), que es el servicio en donde se obtiene la respuesta a las consultas utilizando el índice invertido (ver capítulo 2). El índice invertido es una estructura de datos masiva, y cuando una consulta debe ser resuelta ahí, se consume gran cantidad de recursos computacionales.

De ahí el impacto del CS, ya que una entrada en cache implica el ahorro, en términos computacionales, de recursos. Pero más importante aún, es el ahorro de tiempo de procesamiento de una consulta al estar en cache, lo que en términos prácticos se traduce en respuestas más rápidas al usuario.

Si bien es cierto que un sistema de cache *sobrevive* (sigue operando) a fallas en los nodos, se presenta un deterioro en el desempeño del CS, ya que toda la información histórica en el nodo que falla se pierde. Información histórica se refiere a las consultas y sus respuestas almacenadas en cache. Este deterioro en el desempeño del CS se debe a que existe una reducción de la tasa de *hit* debido a la pérdida del nodo y todas sus respuestas pre-computadas, lo que a su vez se traduce en un aumento en el tiempo de respuesta promedio. Por otro lado, una falla en uno o varios nodos implica un cambio en la configuración que se consideró en el dimensionamiento del servicio para cumplir con el tiempo de respuesta y *throughput* requeridos.

El esquema clásico de tolerancia a fallas es la replicación de la información a proteger, es decir, si existe información sensible de ser perdida, ésta es replicada completamente en nodos de respaldo. Entonces, ante un fallo los nodos de respaldo son quienes asumen la carga del nodo que falla. Con esto se consigue que, ante fallos el desempeño del Servicio de Cache disminuya, pero en términos acotados y no de forma imprevista y sin control.

Las razones de la replicación de datos son variadas, siendo las más importantes: (i) disponibilidad (continuidad operacional), (ii) confiabilidad (integridad de los datos), (iii) desempeño (tiempo de respuesta y *throughput*), y (iv) escalabilidad (expansión). En el contexto de este trabajo, la disponibilidad (i) de la información relevante ante los fallos (respuestas pre-computadas) es la razón principal de la replicación. La replicación permite continuar la operación, con una reducción parcial del desempeño, en caso de fallas de nodos. Esto debido a que la información relevante, que serían respuestas pre-computadas a consultas previamente realizadas por los usuarios, se encuentra replicada en múltiples nodos, con lo que éstos pueden responder las peticiones hechas al nodo que falla (estas peticiones deben ser re-direccionadas).

La replicación también es utilizada en el capítulo 5 pero tiene más relación con la arista desempeño (iii), ya que una consulta con alta frecuencia es replicada en

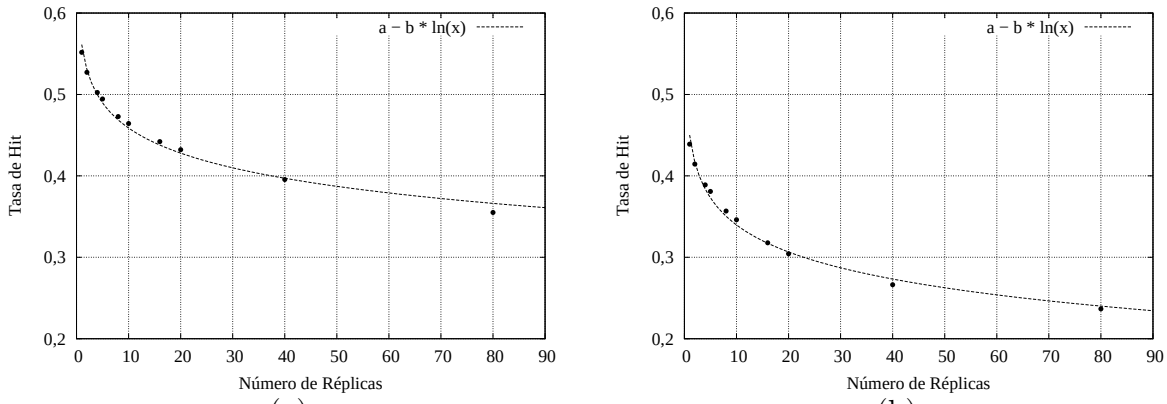
múltiples nodos como una forma de atenuar su impacto en un solo nodo del Servicio de Cache.

El costo a pagar en la implementación de un sistema completamente replicado en el caso de un Servicio de Cache, es la pérdida de entradas totales disponibles, y la implementación de un protocolo de consistencia entre nodos de respaldo. Suponer un Servicio de Cache sin considerar la tolerancia a fallas, y sean NCS el número de nodos de este servicio y que cada nodo tiene N entradas disponibles en cache. El número máximo de entradas de cache disponible en este servicio es $NCS \cdot N$ (todas las entradas en cache son distintas utilizando Consistent Hashing con NCS particiones, ver capítulo 3). En este esquema, la pérdida de un nodo implica la pérdida de las N entradas del nodo que falla.

Si se quiere un Servicio de Cache tolerante a fallas, una idea factible es reducir el número de particiones a NCS/D y que cada partición esté compuesta por D nodos que funcionan como réplicas (así se aprovechan los NCS nodos del servicio). El modo de operación de este esquema es que cada consulta sea asignada a alguna de las NCS/D particiones usando Consistent Hashing, y luego uno de los D nodos réplica es seleccionado al azar para atender la consulta. Como los D nodos de la partición son réplicas, da lo mismo enviar una consulta a cualquiera de los D nodos de la partición. En este caso, el servicio no pierde información relevante ante el fallo de un nodo en una misma partición, ya que existen $D - 1$ nodos de respaldo con la misma información. Sólo en el caso fallen los D nodos de la partición, se pierde la información relativa a una partición.

Existen dos desventajas en un esquema como el propuesto anteriormente: (i) pérdida de entradas totales disponibles, e (ii) implementación de un protocolo de consistencia para las réplicas de una partición. La pérdida de entradas totales disponibles para cache es evidente, ya que con un sistema con NCS/D particiones, D réplicas por partición y N entradas disponibles por nodo, el número total de entradas disponibles se reduce a $NCS \cdot N/D$. Si se reducen las entradas disponibles en cache, se reduce la tasa de *hit*, lo cual incrementa la utilización de recursos, principalmente en el IS, y el tiempo de respuesta de las consultas.

Para evidenciar la pérdida de entradas totales, en la Figura 52 se observa cómo



(a) (b)

Figura 52: Evolución de la tasa de *hit* a medida que se aumenta el número de réplicas manteniendo constante el número de nodos del Servicio de Cache (Enero de 2012 y $NCS = 80$) y su ajuste a la función $f(x) = a - b \cdot \ln(x)$: (a) 100 mil entradas por nodo, con $a = 0,561308$ y $b = 0,0445142$; (b) 10 mil entradas por nodo, con $a = 0,450203$ y $b = 0,0479532$.

evoluciona la tasa de *hit* a medida que se aumenta el número de réplicas por partición. El supuesto inicial es que siempre se tienen $NCS = 80$ nodos en el Servicio de Cache, por lo que las configuraciones del Servicio de Cache varían desde $P = 1$ y $D = 80$ hasta $P = 80$ y $D = 1$. En la Figura 52(a) se tiene un sistema compuesto por 100 mil entradas por nodo, y se observa que la tasa máxima de *hit* alcanzada es de un 55,2 % (80 particiones sin réplicas) y la mínima tasa de *hit* es de un 35,5 % (cuando existe una partición con 80 réplicas). En la Figura 52(b) se observa el mismo experimento anterior pero cada nodo tiene 10 mil entradas en cache (43,9 % y 23,7 %). Notar el impacto que tiene la replicación cuando las restricciones de espacio son más exigentes (100 mil versus 10 mil). En esta situación, al tener menos entradas disponibles para cache, la replicación hace que se reduzcan aún más las entradas disponibles para cache. Además, se hace una mala utilización de los recursos, debido a que no todas las entradas tienen la misma importancia como se mencionó previamente. En la Figura 52, además de los datos, se presenta la función que se ajusta a esos datos. Esta función está dada por $f(x) = a - b \cdot \ln(x)$, lo que implica que la tasa de *hit* disminuye en forma logarítmica a medida que se aumenta el número de réplicas.

A continuación se hará un análisis de la situación experimentada en la Figura 52(a). Considerar los siguientes puntos en el gráfico (cada punto indica el número de particiones, réplicas y la tasa de *hit*): (i) $\langle P = 80, D = 1\rangle = 55,2\,\%$; (ii) $\langle P = 40, D = 2\rangle = 52,7\,\%$; y (iii) $\langle P = 20, D = 4\rangle = 50,2\,\%$. Se debe considerar que aumentando el número de réplicas, el servicio se vuelve más resistente a las fallas. Por ejemplo, si se tienen sólo dos réplicas y sucede una falla en una partición cualquiera, el servicio se vuelve proclive a la pérdida de toda la información de esa partición. De los datos expuestos, se observa que al pasar de un servicio sin replicación a uno con dos réplicas, se percibe una pérdida de 4,5 % en la tasa de *hit* y al pasar a cuatro réplicas, se produce una pérdida de 9,1 %. Estos resultados en la Figura 52(b) son 5,5 % y 11,4 % respectivamente. Esto da una primera idea de que la replicación total de nodos en una partición no es una buena estrategia para proveer tolerancia a fallas y disponibilidad. Esto también confirma que cuando existen restricciones con respecto a la cantidad de entradas en cache, los costos de una mala administración de los recursos son mayores.

La otra desventaja de la replicación es la implementación de un Protocolo de Consistencia en cada partición, debido a que cada una de ellas debe estar completamente replicada (en caso de que exista más de un nodo por partición). La replicación es una tecnología clave en los sistemas distribuidos que permite aumentar la disponibilidad y mejorar el desempeño [SS05], además de proveer tolerancia a fallas y permitir el acceso concurrente a múltiples nodos con la misma información. Un Servicio de Cache con replicación permite que las respuestas almacenadas en nodos que fallan en una partición específica, no se pierdan y estén disponibles en las otras máquinas de esa misma partición. Asimismo, es necesario implementar un Protocolo de Consistencia que lidie con las réplicas. Varios Modelos de Consistencia [TS06] han sido propuestos para llevar a cabo la replicación. La Consistencia Estricta (Strict Consistency) es el modelo de consistencia más riguroso, ya que implica que todas las escrituras/modificaciones estén inmediatamente visibles en todas las réplicas. Este modelo se apoya en un tiempo global absoluto o cerraduras que reducen la concurrencia.

Otros modelos factibles a considerar son Consistencia Causal (*Causal Consistency*) y Consistencia Débil (*Weak Consistency*) [SS05]. Estos modelos de consistencia son más flexibles y pueden ser apropiados para un Servicio de Cache. Aquí cabe analizar la diferencia entre la aplicación estudiada en este trabajo y las aplicaciones que necesitan implementar Consistencia Estricta. Existen aplicaciones en las cuales debe existir una precisa sincronización en la ejecución de operaciones sobre los datos almacenados, principalmente en aplicaciones transaccionales como bases de datos financieras. Si se quiere leer un dato que fue escrito anteriormente, entonces desde el punto de vista transaccional, la escritura debe realizarse antes que la lectura. Si los datos almacenados están replicados en diversos nodos, entonces la sincronización y coordinación debe ser más férrea, ya que cada vez que exista una modificación de un dato, esta modificación debe estar visible inmediatamente en los otros nodos (en caso que exista una lectura posterior). En este tipo de aplicaciones se debe usar Consistencia Estricta. Por otro lado, en la aplicación estudiada suponer la siguiente situación: se tiene una partición replicada con D nodos. Como esta partición está replicada, entonces una petición q puede ser enviada a cualquier réplica de la partición. Si la petición q no ha aparecido anteriormente en esa partición, entonces el nodo que recibe la petición resuelve con un *miss* y luego cuando esa petición ha sido respondida por el servicio de índice, se debe insertar en el nodo que respondió *miss* en la partición. Esto implica que sólo un nodo tiene la consulta q y su respuesta en cache, y los restantes $D-1$ nodos no la tienen. La inconsistencia descrita anteriormente no causa ningún problema crítico en la partición, ya que la aplicación no es transaccional y un nodo sólo responde una petición con *hit* o *miss*. Es decir, no es crítico que existan tales inconsistencias. Lo que sí se quiere asegurar es que enviando una petición a cualquier réplica de una partición, ésta tenga una alta probabilidad de ser encontrada en cache. Es por esta razón que un protocolo de Consistencia Débil es el adecuado para servicios de cache distribuido con replicación.

La desventaja de cualquier protocolo de consistencia es la utilización de la red de comunicación para implementarlo [YV00], para así alcanzar un estado consistente entre las réplicas. Finalmente, mientras más réplicas involucre el protocolo de consistencia, más complejo es el proceso en términos de cómputo y transferencia por la red

(el proceso no es escalable en el número de réplicas D).

Este trabajo se aleja de la idea de proteger un nodo completo (no habrá replicación de nodos), como es la idea de tolerancia a fallas en aplicaciones paralelas sobre clusters de computadores [RL10]. Estas aplicaciones corresponden generalmente a aplicaciones de gran escala de cómputo científico, en que la falla de un nodo implicaría la pérdida de una alta cantidad de recursos computacionales. La propuesta es la replicación de ciertas entradas de cache en un servicio totalmente particionado. Es decir, en vez de considerar un esquema de P particiones y D réplicas ($P \cdot D = NCS$), este trabajo considerará que todos los nodos funcionan como partición y se replica cierta información en cada nodo, con la finalidad de no reducir drásticamente el desempeño ante la ocurrencia de fallas.

La hipótesis a probar es que la protección de información relevante del Servicio de Cache, basado en el comportamiento zipfiano de las consultas de usuario y replicación controlada, ayuda a mitigar los efectos de las fallas de nodos.

8.2. Solución

Para idear un Servicio de Cache (CS) que tolere de mejor forma los fallos, se debe considerar un factor importante de las consultas de usuario: distribución Zipf. Esto último implica que unas pocas consultas tienen un alto impacto (frecuencia) en el CS, mientras que muchas consultas tienen bajo impacto. Aún cuando se tenga una cantidad limitada de entradas de cache en cada nodo, se tiene que cada entrada no tiene el mismo impacto. Esto tiene implicancia inmediata en el diseño del CS: no es necesario proteger todas las respuestas pre-computadas, sólo aquéllas con alto impacto. Una consulta de alto impacto implica que tenerla en cache impide que una alta cantidad de peticiones visiten el Servicio de Índice (IS) para su resolución. Por otro lado, cada nodo del CS utiliza una estructura de datos en memoria que representa el cache del nodo. Esta estructura de datos está diseñada del tal modo que esté orientada a un algoritmo de cache, es decir, que las operaciones realizadas por el algoritmo de cache sean rápidas y contribuyan a mantener un ordenamiento consistente con el algoritmo.

La mayoría de los algoritmos de cache instauran lógicamente una estructura jerárquica, en que las entradas más importantes para el algoritmo se encuentran en el tope de la estructura, mientras que las menos importantes y prescindibles se encuentran en el fondo de la estructura. Por lo tanto, la información más relevante a ser protegida se encuentra en el tope de esta estructura de datos. Con esto se quiere decir que la información a proteger está relacionada con el algoritmo, ya que cada uno de ellos prioriza las entradas de acuerdo a una característica. Algunas características posibles son la frecuencia de una consulta o su costo de resolverla en el IS.

Por ejemplo, para los algoritmos LRU y LFU las entradas más importantes están en el tope de la estructura de datos: las más recientemente utilizadas y las más frecuentes. Mientras que las menos importantes están hacia el fondo de la estructura de datos. Las consultas a proteger en estos casos, son las que están en el tope de la estructura. En este trabajo no es importante qué algoritmo se utiliza, ya que como fue descrito en el capítulo 2, existen muchos algoritmos que tienen distintas virtudes y debilidades. Lo importante en este trabajo es que la mayoría de los algoritmos tienen una estructura jerárquica, que puede ser aprovechada para identificar la información relevante a ser protegida. Finalmente, este trabajo se beneficia de la distribución zipfiana de consultas de usuario para optimizar el proceso de protección.

Un problema es la determinación de la porción de consultas a proteger. Para ser justos en términos de comparación, siempre las estrategias analizadas tienen la misma cantidad N de entradas en cache. Es decir, si se ha seleccionado como α la porción de entradas (sobre N) a proteger en cada nodo, entonces cada nodo sólo debe tener disponible $(1-\alpha)N$ entradas para cache.

La determinación del nivel óptimo depende principalmente del *log* de consulta, ya que desde ahí se tiene variables como el número de consultas distintas, el *slope* de la distribución zipfiana, entre otras. Otros elementos importantes son el algoritmo de cache utilizado en cada nodo y el número de particiones. El procedimiento que se siguió para la determinación del nivel es empírico, es decir, basado en experimentos sobre los parámetros descritos anteriormente.

La Figura 53(a) presenta el impacto en la tasa de *hit* de una reducción porcentual determinada (desde 0 % a 20 %), y la Figura 53(b) presenta la ganancia que se obtiene

al hacer protección con las reducciones porcentuales descritas. Notar el impacto en la reducción de $x\,\%$ de entradas de cache en un nodo, y la ganancia que se obtendría con ese mismo $x\,\%$ de entradas en caso de existir un fallo (estas últimas entradas estarían alojadas en algún nodo y todas las peticiones del nodo que falló serían atendidas por esta sección). Las mediciones fueron realizadas cada minuto durante el 2 de Enero de 2012, la protección de entradas tomó lugar cada 5 minutos y el servicio está compuesto de 20 nodos con 100 mil entradas para cache por nodo. A primera vista en la Figura 53(a) se observa que reducir el número de entradas en cache no tiene un gran impacto en la tasa de *hit*, reforzando la idea de las distribuciones Zipfianas en las consultas de usuario. Esta reducción tiene un beneficio importante que se observa en la Figura 53(b), ya que, por ejemplo, con un $1\,\%$ de reducción de entradas (desde $50{,}2\,\%$ a $50{,}1\,\%$ en la tasa de *hit*) se alcanza una protección del $34{,}4\,\%$ de las peticiones (en caso de fallos). Se debe tener claro que esta ganancia se produce sólo en caso de fallos, por ende no es apropiado reducir las entradas en forma drástica (hasta un $50\,\%$), ya que esto implicaría una mayor utilización del servicio de índice. Sin duda existe un equilibrio.

En la Figura 53(c) y (d) se detalla la misma situación, pero con 10 mil entradas por nodo destinadas a cache. Notar la reducción que existe en la tasa de *hit* considerando la Figura 53(a) y (c). Una pregunta que asoma al observar la Figura 53(c), es por qué la tasa de *hit* es tan ajustada ante la reducción del cache. Esto es debido a la restricción que impone la reducción de entradas, ya que con menos entradas sólo las con mayor impacto quedan en cache, por lo que se podría ver una especie de homogeneización de las zonas de importancia, y por ende un impacto acotado ante la reducción.

La Figura 53 demuestra que la idea planteada en este capítulo sí tiene sustento empírico, ya que se observa que la protección de información relevante, a través de la replicación controlada, ayudará en forma importante a no experimentar una pérdida drástica de la tasa de *hit*. Esta estrategia está basada principalmente en la utilización correcta de las distintas zonas de la memoria cache, ya que no todas tienen la misma importancia. Un hecho relevante que da la evidencia empírica es que el sacrificio de un porcentaje de la tasa de *hit*, a través de la disminución de entradas, tiene una

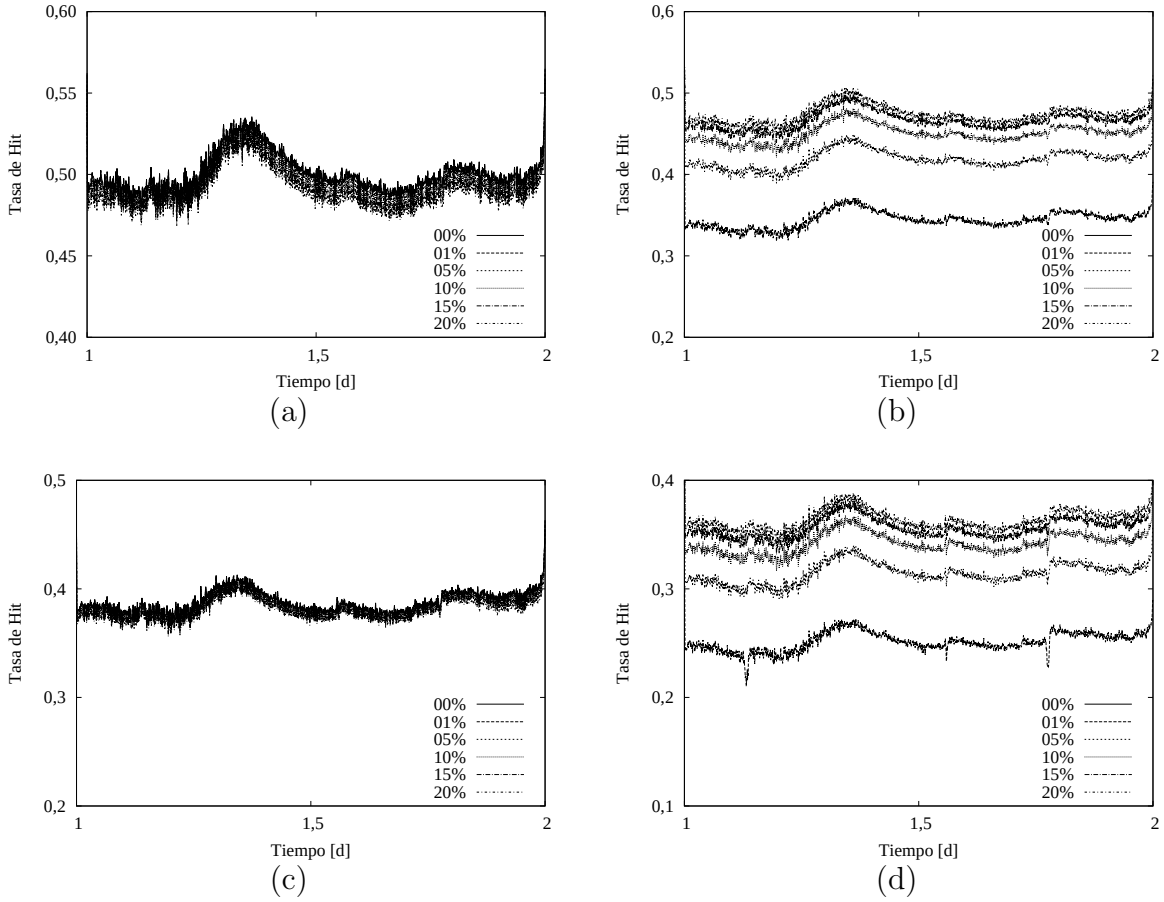


Figura 53: Impacto en la tasa de *hit* con reducción porcentual de cache: (a) 100 mil, y (c) 10 mil entradas. Ganancia obtenida con protección de las consultas más frecuentes: (b) 100 mil, y (d) 10 mil entradas. Las consultas utilizadas corresponden al 2 de Enero de 2012.

Tabla 30: Tasa de *hit* promedio porcentual (HR) y ganancia en la tasa de *hit* promedio porcentual (G) para distintas reducciones porcentuales. Datos obtenidos de la Figura 53.

Porcentaje	100M		10M	
	HR	G	HR	G
0 %	50,2	0,0	38,9	0,0
1 %	50,1	34,4	38,8	24,7
5 %	50,0	41,6	38,7	31,3
10 %	49,7	44,7	38,4	33,9
15 %	49,5	46,3	38,1	35,4
20 %	49,2	47,2	37,9	36,2

ganancia no lineal a este sacrificio. Es decir, para un caso particular, si se experimenta una pérdida de 0,1 % en promedio en la tasa de *hit*, la ganancia obtenida en la tasa de *hit* es en promedio de 34,4 % (en el caso de fallas). La Tabla 30 muestra los datos promedio para cada caso de la Figura 53. Según esos datos, el mejor porcentaje de protección se encuentra entre un 1 % y 10 %.

El siguiente problema a tratar es la determinación del nodo de respaldo ante una falla. Para eso, se aprovecha la funcionalidad de Consistent Hashing. Es decir, cuando en un servicio falla un nodo, las peticiones realizadas a ese nodo son re-dirigidas al primer nodo encontrado en el anillo siguiendo el sentido de las agujas del reloj. Aprovechando esta forma de operar, el nodo de respaldo para cualquier nodo en el Servicio de Cache será el siguiente nodo en el anillo. Este filosofía también puede ser encontrada en el esquema RADIC [FSD+09, SDRL08] (*Redundant Array of Distributed Independent Fault Tolerance Controllers*), que es una arquitectura que provee tolerancia a fallos en un sistema de paso de mensajes. Esta arquitectura ofrece alta disponibilidad en forma transparente, descentralización, flexibilidad y escalabilidad.

Entonces, tanto el respaldo de la información sensible como la re-dirección de las consultas ante nodos que fallan, siguen el mismo esquema de trabajo. Con este mecanismo, no se requiere infraestructura adicional en lo que se refiere a ruteo de consultas. Sólo se sigue en forma natural el procesamiento de consultas con Consistent Hashing.

En la Figura 54 se muestra el funcionamiento del esquema propuesto. La Figura

54(a) representa un Servicio de Cache compuesto por seis nodos. Cada nodo tiene en su memoria principal una estructura de cache y, como se mencionó anteriormente, se pueden identificar zonas más importantes que otras en función del algoritmo de cache a utilizar. Cada zona de importancia es identificada con un tono distinto en cada nodo, siendo las zonas más oscuras las más importantes. En la Figura 54(b) se muestra cómo el mecanismo de protección propuesto respalda la información relevante (zona más oscura debido al comportamiento zipfiano de las consultas de usuario) y la replica en el nodo protector utilizando la misma lógica que Consistent Hashing (el siguiente nodo siguiendo el sentido de las agujas del reloj). Posteriormente, la información de respaldo recibida por cada nodo es almacenada en memoria como es indicado en la Figura 54(c). Cabe señalar que si se determina que se protegerá un 20 %, ese mismo porcentaje debe ser reducido de cache para fines de protección. El flujo descrito anteriormente es el que permite la protección de información relevante en el Servicio de Cache, y este proceso toma lugar cada Δt unidades de tiempo.

La utilidad del proceso definido anteriormente toma relevancia en el momento de las fallas. Suponer una falla del nodo *P2* como es descrito en la Figura 54(d). En este caso, toda la información del nodo en cuestión se pierde, excepto la protegida en el nodo de respaldo (*P3*). En el momento de la falla de *P2*, todas las peticiones que son asignadas a este nodo vía Consistent Hashing son re-dirigidas al nodo *P3*. Este último nodo puede responder en parte las peticiones con un *hit*, ya que como se observa en la misma figura, la sección en color negro etiquetada con *P2* contiene respuestas pre-computadas pertenecientes a este último nodo. De esta forma se mitiga la pérdida en la tasa de *hit* del Servicio de Cache debido a los fallos.

Posteriormente, en la Figura 54(e) se observa una nueva iteración del mecanismo de protección sin considerar el nodo caído *P2*. Es importante señalar que, desde el instante de tiempo en que falla el nodo *P2* hasta que se realiza una nueva iteración del mecanismo de protección, el nodo *P1* no tiene información de respaldo en el CS, ya que ésta estaba en el nodo que falla *P2*. Este período de tiempo es acotado y el CS vuelve a un estado normal de funcionamiento, en términos de la protección de información relevante, en la siguiente iteración como es descrita en la Figura 54(e). Finalmente, el servicio trabaja normalmente como en la Figura 54(f).

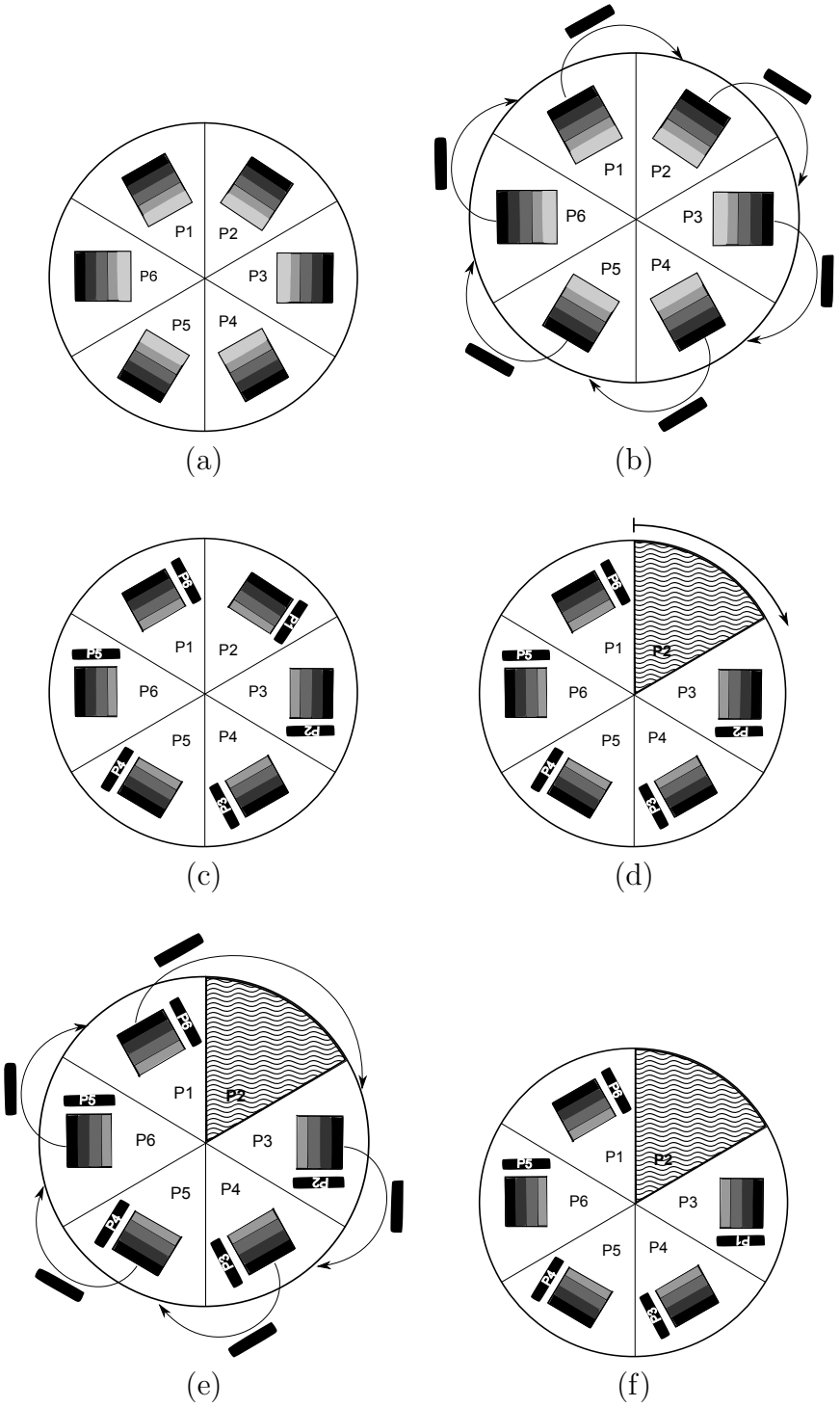


Figura 54: Mecanismo de protección propuesto en este trabajo.

Algoritmo 5 Protección. Mecanismo propuesto para protección a ejecutarse cada Δt unidades de tiempo en cada nodo.

Input: P_{prot} porcentaje de protección, *Cache* estructura cache local al nodo.

```
1: top ← Cache.GetTopCache( P_prot )
2: neig ← GetMyNeighbor( )
3: for each p ∈ top do
4:     Enviar a neig un mensaje de protección < p.query, p.answer >
5: end for
```

8.3. Algoritmos

El Algoritmo 5 muestra los pasos necesarios para llevar a cabo localmente en cada nodo del CS. El primer paso es obtener las entradas de la cache local destinada a protección (cada entrada está compuesta por *query* y *answer*). La cantidad está determinada por el parámetro determinado P_{prot} que debe ser configurado, mientras que la selección de las entradas depende de la política de reemplazo. Cada nodo tiene la posibilidad de obtener su vecino en el anillo Consistent Hashing (este nodo representa el nodo *protector*). Este valor puede ser actualizado constantemente por el Servicio de Front-End (FS) dependiendo de los nodos que se encuentren activos. Por cada entrada seleccionada, se debe enviar un mensaje de tipo *protección* al vecino en el anillo. Este conjunto de pasos se realiza cada cierto intervalo de tiempo que también debe ser configurado, pero la escala debería ser del orden de minutos. El esquema propuesto es asíncrono, ya que cada petición de protección se envía mediante un mensaje que debe ser transmitido y procesado por el nodo protector. El nodo que envía el mensaje en ningún momento se bloquea esperando por una respuesta. Se escoge este esquema asíncrono basado en mensajes para la protección, ya que permite mayor flexibilidad y un funcionamiento en forma transparente, y no son necesarias primitivas de sincronización para llevarlo a cabo.

El Algoritmo 6 detalla el funcionamiento de cada nodo al recibir las distintas peticiones. Existen principalmente tres peticiones con el esquema propuesto: (i) buscar una clave; (ii) insertar una clave; y (iii) proteger un par consulta/respuesta. Al realizar una búsqueda, el nodo debe consultar su estructura cache. En caso de estar en la estructura principal, se actualiza la prioridad de la consulta y se responde con la

Algoritmo 6 NodoCS. Funcionamiento de un nodo del CS para incluir la protección.

Input: P cola de peticiones al nodo, *cache* estructura cache del nodo, *cache-prot* estructura de protección alojada en el nodo.

```
1: for each p ∈ P do
2:     if p es búsqueda then
3:         if cache.Exists( p.query ) then
4:             cache.UpdatePriority( p.query )
5:             return cache.GetAnswer( p.query )
6:         else
7:             if cache-prot.Exists( p.query ) then
8:                 cache.Insert( p.query, cache-prot.GetAnswer( p.query ) )
9:                 return cache-prot.GetAnswer( p.query )
10:            else
11:                return miss
12:            end if
13:        end if
14:    end if
15:    if p es inserción then
16:        cache.Insert( p.query, p.answer )
17:    end if
18:    if p es protección then
19:        if cache-prot.Exists( p.query ) then
20:            cache-prot.UpdatePriority( p.query )
21:        else
22:            cache-prot.Insert( p.query, p.answer )
23:        end if
24:    end if
25: end for
```

respuesta. En caso contrario, se consulta la estructura de protección. Si se encuentra en esta estructura, entonces se inserta el par consulta/respuesta en la cache local y se responde con la respuesta a la consulta. En caso de que no esté en ninguna de las estructuras, se responde con un *miss*. Se tienen dos casos importantes a detallar:

1. El nodo p realiza la consulta a la estructura de protección, ya que la petición que está siendo procesada puede estar relacionada a un fallo del nodo v que se está protegiendo (v replica sus entradas en p, p protege a v). En caso de falla de v, todas las peticiones hechas a este nodo serán re-direccionadas al nodo p. Esta consulta a la estructura de protección, podría saltarse si el FS informa de los nodos que han salido del servicio.

2. Si la consulta está en la estructura de protección (v sale del servicio), el par consulta/respuesta es insertado inmediatamente en la cache del nodo p. Con esto se comienza a hacer un traslado natural de las entradas de v a la cache del nodo p, con el fin de que este último nodo comience a alojar en su estructura local las peticiones que pertenecían a v.

Considerar la siguiente situación: se tiene una secuencia de nodos en el anillo u, v y p (u protege sus entradas en v y v en p). La protección de entradas se realiza cada Δt unidades de tiempo, es decir, cada Δt unidades de tiempo se ejecuta el Algoritmo 5 en cada nodo en forma concurrente. Cuando sucede un fallo en v, todas las peticiones hechas a este nodo son asignadas a p, siguiendo el mecanismo de Consistent Hashing. Como la protección se realiza cada Δt unidades de tiempo, sólo se tiene un tiempo acotado en que las entradas protegidas de v en p están alojadas. En el siguiente intervalo de tiempo, p protegerá las entradas de u, haciendo que las de v se pierdan. En conclusión, independiente del instante de tiempo, las entradas protegidas sólo duran un intervalo de tiempo en la presencia de fallos. Es por esta razón que cada vez que una consulta es encontrada en *cache-prot* en el Algoritmo 6 debe ser insertada en la cache local del nodo. De esta forma se tiene que las entradas presentes en las distintas estructuras se hace consistente en forma natural y gradual, ya que, desde detectada la falla de un nodo hasta el siguiente período de protección, las entradas son insertadas en la cache local cuando se produce un *hit*. Luego, como las entradas

protegidas son insertadas en la cache local, estas entradas ya forman parte de la cache del nodo y serán *hit* en la cache local (no en la protegida). Por último, esto también hace que la cantidad de peticiones al espacio de protección se reduzca a través del tiempo hasta la siguiente protección, ya que las peticiones con *hit* relativas al nodo caído son encontradas en la cache local.

Esto abre la pregunta de cuánto es la duración del intervalo de tiempo para protección. Por un lado, si es muy pequeño se sobrecarga la red de comunicación y la utilidad de las entradas protegidas en caso de fallas se reduce. Si el intervalo es demasiado grande, las entradas protegidas no se actualizan constantemente y, en caso de fallas, se utilizan ineficientemente, ya que como se mencionó anteriormente, cuando sucede un *hit* en esta sección, la entrada se traslada a cache local. En cualquier caso, la protección debe llevarse a cabo en el orden de minutos.

Existe la posibilidad de proteger en más de un nodo, así en caso de falla de dos nodos consecutivos, se tiene un segundo nodo en que se aloja las entradas protegidas. En el ejemplo anterior, suponer que u protege sus entradas en v y p. En caso de falla de u y v, se tiene el nodo p que contiene entradas protegidas de los dos nodos caídos.

En el Algoritmo 6 también se encuentra la inserción, que corresponde a que una consulta hizo *miss* anteriormente y se resolvió en el Servicio de Índice (IS). Finalmente, si una petición es de protección, significa que un nodo está enviando un par consulta/respuesta para proteger en la zona especial del nodo.

8.4. Experimentos

Para evaluar el *overhead* o costo adicional de implementar el mecanismo de protección, se realizó una ejecución real sobre un cluster. Se utilizó un *log* de 100 millones de consultas que corresponden a dos días de operación (1 y 2 de Enero de 2012).

En el experimento se midió el tiempo necesario para finalizar un conjunto de consultas y la tasa de *hit* resultante en un servicio compuesto por 5 nodos. La idea es probar un esquema de Consistent Hashing (CH) puro y el esquema de CH más la propuesta de protección. Además, se inyectan dos fallas en $x = 1.500$ y $x = 6.000$ (sólo los 10 mil primeros puntos de la ejecución son graficados). Después de cada falla,

el nodo es re-insertado en el servicio, en los puntos $x = 1.600$ y $x = 6.500$. La idea de esto es evaluar el desempeño de las estrategias durante un intervalo reducido en la presencia de fallas. El servicio sin el sistema de protección tiene 100 mil entradas por nodo, y la implementación del sistema de protección hace que se tengan 95 mil entradas para cache y 5 mil entradas para protección por nodo.

La Figura 55(a) muestra el tiempo requerido para finalizar las consultas. El *overhead* impuesto por la estrategia propuesta (etiquetada con 'Radic') es 1,8 % en promedio, que no representa un impacto severo para el servicio. Cada proceso de protección toma lugar cada 1.000 pasos y sólo el 5 % de las consultas más importantes son consideradas. El proceso de protección se optimiza mediante un esquema *pipeline*, es decir, el 5 % de pares consulta/respuesta es enviado al nodo correspondiente en múltiples pasos. El desempeño de la protección de entradas se hace visible en el caso de fallos, lo cual es presentado en la Figura 55(b). Se observa que la tasa de *hit* es menos afectada por fallas, ya que la propuesta mitiga parte de las entradas perdidas en otro nodo. La tasa de *hit* promedio de la propuesta es 47,2 % y un 46 % en el caso de CH (sin protección). Esto representa un incremento de 2,6 % ante la existencia de fallos.

Finalmente, la Figura 55(c) muestra el contraste entre el costo asociado de la implementación de protección de entradas de la propuesta (etiqueta 'Radic') y un protocolo de consistencia optimista (etiqueta "Optimista"). Este experimento fue realizado teniendo el cuidado de que cada proceso fuese alojado en un nodo distinto (así la comunicación es a través de la red de comunicación), y tiene como objetivo probar la escalabilidad de los distintos modos de replicación de entradas. El propósito de este experimento es evaluar los efectos de la comuncación extra requerida para llevar a un estado consistente las réplicas de una partición en tiempo de ejecución. La curva etiquetada como "S/M" (sin método) es el mejor desempeño posible, ya que no implementa el protocolo de consistencia ni la protección de entradas. Notar que cada réplica recibe el mismo número de consultas, por lo que a medida que se aumenta el número de réplicas, se aumenta el número total de consultas procesadas. El protocolo de consistencia es ejecutado cada 1 y 10 minutos, al igual que el mecanismo de protección. De acuerdo a los resultados de la figura, se observa que la propuesta tiene mejor desempeño que el protocolo optimista en ambos casos, y específicamente

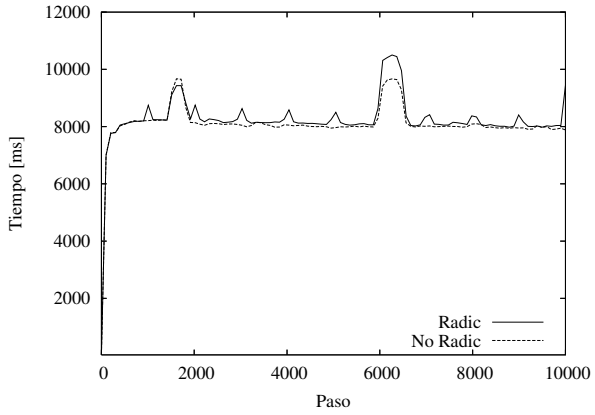
en el caso de la protección cada 10 minutos, el desempeño es muy similar al caso óptimo.

8.5. Conclusiones

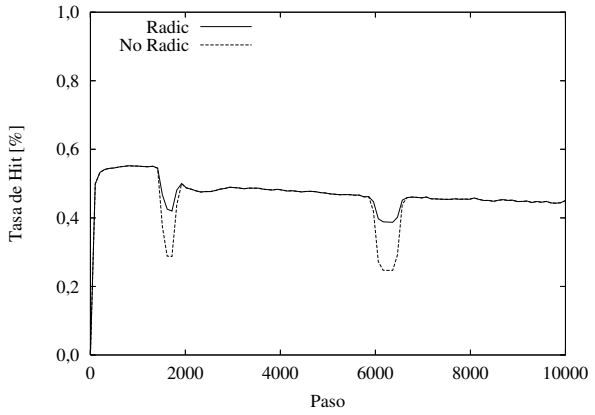
Debido a las consultas zipfianas, no todas las consultas tienen el mismo impacto. Esto también se repite en las consultas alojadas en cache. Cada entrada alojada en cache, implica la utilización de recursos en el Servicio de Índice (IS). Ante la falla de un nodo, toda la información alojada en cache (memoria principal) se pierde, por lo que proveer un mecanismo de protección proactivo que permita reducir esta pérdida fue el objetivo de este capítulo.

El mecanismo propuesto se basa en el esquema clásico de tolerancia a fallas: la replicación. La idea clave es considerar lo expuesto en el párrafo anterior: no todas las entradas tienen el mismo impacto, por lo que replicar en forma controlada y precisa algunas de ellas, específicamente las más importantes, provee ventajas con respecto a la replicación completa de un nodo. Este mecanismo de protección hace uso de la misma infraestructura presente en el ruteo (Consistent Hashing): las entradas replicadas son asignadas al siguiente nodo siguiendo el sentido del reloj en el anillo. De este modo, ante una falla, las entradas cuyo responsable es el nodo caído son asignadas exactamente al nodo que protege las entradas del nodo protegido. Este mismo mecanismo puede ser extendido para que la protección se realice en más de un nodo, y así tener un servicio más robusto.

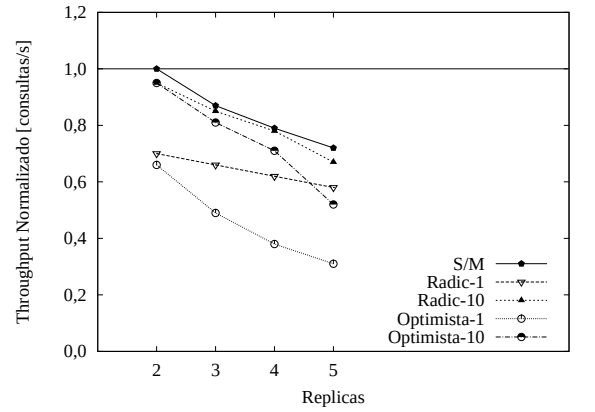
Existen dos parámetros a configurar en la propuesta: (i) el porcentaje a proteger; (ii) el intervalo de protección. Al realizar protección, por cada entrada a proteger se debe reducir una entrada en la cantidad total de cache del nodo. Esto es para evaluar en las mismas condiciones todas las estrategias. La entrada a proteger es seleccionada desde la sección de más importancia de la cache, y por otro lado, si se reduce una entrada el tamaño de la cache, las entradas afectadas con esta reducción de espacio son las menos importantes. Es decir, el impacto de las entradas protegidas/replicadas es mucho mayor que el de las entradas afectadas por la reducción del espacio en cache. Esto es evidenciado en la Figura 53. El porcentaje a proteger depende fuertemente



(a)



(b)



(c)

Figura 55: Evaluación de *overheads* mediante implementación real: (a) tiempo de ejecución; (b) tasa de *hit*; (c) contraste de la propuesta con un protocolo de consistencia optimista.

de la distribución de las consultas, y debe ser configurado en base a experimentos y análisis de los costos/ganancias obtenidos. Según los experimentos realizados, un porcentaje de entre 5 % y 10 % tiene buenos resultados.

Por otro lado, el intervalo de protección incide en la utilidad de la zona dedicada a protección, ya que cada vez que una nueva etapa de protección toma lugar, las entradas anteriores se eliminan. Si se considera un intervalo de protección muy reducido, la utilización de la red de comunicación aumenta y la utilidad se ve reducida. Por lo anterior, el intervalo debe ser de mediano plazo. En la experimentación, este intervalo fue de 5 y 10 minutos. El incremento de las comunicaciones con la propuesta es $O(N \cdot A)$, con N el número de entradas a proteger y A el tamaño de una respuesta pre-computada. Esta transferencia es en cada nodo, pero es posible realizar esta transferencia en paralelo, por lo que es independiente del número de nodos. Por otro lado, el costo temporal de la propuesta está dado por la selección de las entradas a proteger, es decir $O(N)$ cada Δt unidades de tiempo.

Las desventajas de la estrategia propuesta son:

1. Impone *overhead* en comunicación y utilización.
2. Baja el desempeño en el procesamiento de consultas.

Pero las ventajas son:

1. Replicación controlada e inteligente de entradas, sin establecer un protocolo de consistencia.
2. Degradación acotada de la tasa de *hit* y del tiempo de respuesta promedio en caso de fallas, en contraste con las estrategias base.
3. Utilización limitada y eficiente de los recursos, principalmente de la red.
4. El diseño del mecanismo permite protección transparente, con pocas modificaciones y utilizando la infraestructura existente (Consistent Hashing).

Se debe tener en consideración que el mecanismo propuesto impone un *overhead* adicional, razón por la cual los experimentos se realizaron sobre espacios acotados

de tiempo. Es decir, en una situación que considere fallos, la estrategia propuesta tiene mejor desempeño pero acotado a un intervalo de tiempo después del fallo. Si se consideran largos períodos de tiempo, probablemente otras estrategias tendrán mejor desempeño. Lo que importa, desde el punto de vista de la tolerancia a fallas, es ver cómo se comportan las distintas estrategias en el corto plazo y bajo la ocurrencia de fallas.

Finalmente, hasta donde se ha investigado, los esquemas clásicos de servicios de cache no consideran ningún mecanismo proactivo de protección, y en caso de pérdidas esa información debe ser nuevamente computada.

Conclusiones

Este trabajo de tesis ha propuesto una estrategia para la gestión eficiente de consultas almacenadas en un Servicio de Cache para Motores de Búsqueda Web. El diseño de estos sistemas para clusters de procesadores debe considerar la solución a problemas de balance de carga, capacidad para procesar cientos de miles de consultas por segundo garantizando cotas al tiempo de respuesta de consulta individual, y tolerancia a fallas.

La propuesta de la tesis consiste de soluciones específicas para los siguientes subproblemas detectados como relevantes a partir del análisis de grandes cantidades de consultas realizadas por usuarios reales: desbalance estructural (capítulo 5), desbalance dinámico (capítulo 6), consultas en ráfaga (capítulo 7), y tolerancia a fallas (capítulo 8).

En la Figura 56 se muestra en qué lugar de los distintos servicios principales del WSE están insertas las estrategias individuales propuestas en este trabajo de tesis, y se hace énfasis en que la propuesta global de esta tesis consiste en la operación combinada (*tandem*) de estas estrategias individuales. De esta manera, tal como lo muestra la extensa experimentación presentada en la tesis, la propuesta conduce a la realización de servicios de cache para motores de búsqueda que son eficientes, tolerantes a fallas y con rendimiento general significativamente independiente de las fuertes variaciones en intensidad de tráfico y contenido que presentan las consultas de usuarios a lo largo del tiempo. Para esto fueron diseñadas estrategias que funcionan en conjunto y que consideran los tres aspectos del balance de carga: de largo, mediano y corto plazo.

En el Servicio de Cache (CS) la única variación es la modificación correspondiente a la protección de entradas, lo cual no representa un *overhead* importante. Por otro

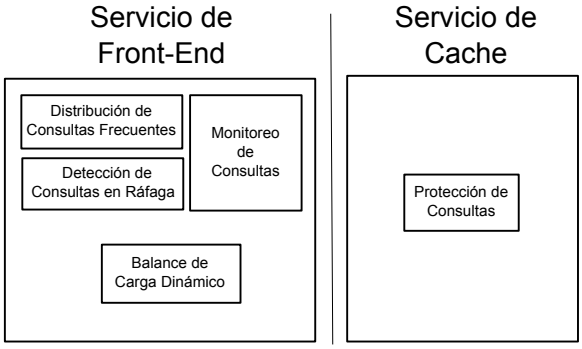


Figura 56: Mecanismos propuestos insertos en la arquitectura de un WSE.

lado, el Servicio de *Front-End* (FS) no realiza funciones de procesamiento de consultas, sólo de ruteo, por lo que sí es posible implantar algunos mecanismos eficientes en espacio y tiempo de ejecución. Además se debe considerar que muchos de los mecanismos son ejecutados cada cierto Δt unidades de tiempo, por lo que tampoco presenta una sobrecarga de consideración. La propuesta descansa en dos técnicas: (i) reporte periódico del estado y utilización de los nodos; y (ii) monitorización de ítemes más frecuentes. Como se ha visto en el desarrollo de este trabajo, estas técnicas se encuentran implementadas en clusters de computadores y aplicaciones Web de gran escala.

Los actuales mecanismos de gestión de servicios de cache distribuido presentan múltiples problemas de eficiencia. En el contexto de servicios de cache distribuido en clusters de computadores, no existe mucho trabajo que analice el desempeño de estos servicios. Aún aplicando técnicas encontradas en sistemas *Peer-to-Peer* (P2P), el desempeño mostrado con esquemas clásicos es pobre, dando lugar a un Servicio de Cache desbalanceado, y susceptible a fallas y alzas abruptas de tráfico. Esto último fue el objetivo de la tesis: analizar las causas que originan estas vulnerabilidades y proponer soluciones, dentro de un contexto de utilización eficiente de recursos.

Dentro de las causas con más impacto, la distribución sesgada de consultas de usuario (Zipf) y su dinamismo, sumado a la inexistencia de mecanismos dinámicos de balance de carga toman protagonismo en la ineficiencia de los servicios de cache. Las principales mejoras experimentadas en este trabajo son: (i) distribución homogénea de consultas frecuentes; (ii) balance de carga dinámico; (iii) detección temprana de

consultas en ráfaga; y (iv) protección de consultas en cache. Estas mejoras atacan directamente cada una de las causas de la pérdida de eficiencia. Mediante la experimentación, se estableció que cada una de las soluciones propuestas contribuye a aumentar la eficiencia en el CS.

La mejora en el punto (i) consiste en la monitorización de los ítemes más frecuentes y la distribución de ellos en más de un nodo. Para ello se utiliza el algoritmo *Space-Saving* que ha mostrado su ventaja en términos teóricos y prácticos. Esto permite una distribución más homogénea de consultas en el anillo, reduciendo el desbalance experimentado con esquemas básicos. A esta propuesta se le denomina balance de largo plazo. Con el mecanismo propuesto se tiene una eficiencia del 80 % versus el 63 % del esquema básico con Consistent Hashing. Esto da espacio para mejorar aún más el desbalance.

Para el balance de carga dinámico, punto (ii), se utilizan algoritmos de balance para estructuras de anillo que aseguran convergencia y estabilidad del proceso de balance. Esta propuesta corresponde a un balance de mediano plazo. Este mecanismo complementa la propuesta de distribución de ítemes frecuentes, lo que permite mejorar aún más el balance. Aún así, todavía existen episodios que podrían poner en riesgo el balance alcanzado. La eficiencia reportada en la implementación de esta estrategia es en promedio un 90 %, debido al umbral tolerado de desbalance.

La detección de consultas en ráfaga, punto (iii), se realiza utilizando la misma estructura de monitorización y estableciendo una serie de tiempo para las consultas más frecuentes. Esta serie de tiempo es acotada a unos pocos puntos, por lo que se torna eficiente en tiempo y espacio. La historia de una consulta en ráfaga es distinta de una consulta frecuente permanente, y eso es efectivo en los datos obtenidos de la serie de tiempo. El mecanismo propuesto se basa en promedio y desviación estándar móvil, y también en un coeficiente de variación móvil. Dado el esquema de reacción de esta propuesta, se le ha denominado balance de corto plazo. El mecanismo propuesto reporta una detección de una consulta en ráfaga de 3,7 minutos en promedio antes del primer máximo versus un 3,2 de un método *off-line* (inviable de implementarlo para detección en tiempo real). Por otro lado, los experimentos muestran que este método permite mitigar las alzas abruptas no cubiertas por los dos mecanismos de

balance descritos anteriormente.

Finalmente, el punto (iv) que corresponde a la protección de consultas, permite la operación ante la ocurrencia de fallos con un pequeño deterioro de la tasa de *hit*. La filosofía es que dado las distribuciones zipfianas de consultas de usuario, con una pequeña porción de entradas en cache replicadas en otros nodos, se tiene una alta ganancia en términos de tasa de *hit*. Esto es debido a que la porción replicada corresponde a las consultas más frecuentes. En la experientación se observa que el *overhead* impuesto para proteger un 5 % de consultas es de sólo un 1,8 % en promedio (en tiempo), lo que permite que en casos de fallos la tasa de *hit* promedio sea de un 47,2 % comparado contra un 46 % de una estrategia base (un 2,6 % superior).

El servicio resultante es un Servicio de Cache eficiente, balanceado y tolerante a fallas, cuyas características son: (i) utilización eficiente de las entradas de cache (alta tasa de *hit*); (ii) menos susceptibilidad a las alzas de tráfico (al estar balanceado y detectando tempranamente las consultas en ráfaga); (iii) menos impacto de las fallas en el desempeño (protección de entradas); y (iv) requiere pocas modificaciones a infraestructura existente como Consistent Hashing.

Esto tiene importantes consideraciones, ya que además de proveer una reducción del tiempo de respuesta promedio con poca variación entre estos tiempos, se puede determinar el número óptimo de nodos a ser utilizados en un Servicio de Cache. Obviamente este número depende de: (a) el volumen y distribución de consultas; y (b) de la capacidad de los nodos; pero al experimentar una carga similar en los nodos, se puede estimar con mayor certidumbre la capacidad del servicio.

9.1. Trabajo Futuro

Como trabajo futuro, múltiples áreas de trabajo quedan abiertas, específicamente para mejorar las optimizaciones diseñadas y para validar más completamente los modelos propuestos.

1. *Balance de Carga Dinámico*. Existen mecanismos que permiten hacer el intervalo de medición variable en función del desbalance detectado. A menor desbalance, el intervalo de medición es mayor. Esto permitiría una mejor respuesta

del mecanismo propuesto a las condiciones de carga de trabajo, es decir, que la ejecución del algoritmo de balance de carga se ejecute de acuerdo al nivel de desbalance del servicio. Por otro lado, el volumen total de consultas varía a través del tiempo, con más énfasis durante el día y la noche. Esto hace que el número de nodos óptimo sea distinto en cada caso (en el día se necesitan más nodos que en la noche), por lo que considerar elementos de planeamiento de capacidad en función del volumen de consultas sea una arista importante a tratar. Esto necesita de un mecanismo dinámico para balancear la carga en caso de inserción y exclusión de nodos en forma controlada. Este último tópico fue solucionado en este trabajo.

2. *Detección de Consultas en Ráfaga*. Existen múltiples métodos de detección de consultas en ráfaga, todos con distintas características de parámetros, recursos utilizados, etc. Según nuestra experiencia, métricas sobre una ventana de tiempo acotada dan evidencias de peso para detectar tempranamente una consulta en ráfaga, así como también para aumentar la probabilidad de acierto. Estudiar otras características y compararlas con otros métodos queda como trabajo pendiente para mejorar aún más el método propuesto. Por otro lado, existen casos particulares en que un grupo de consultas se torna en ráfaga en un instante de tiempo en particular, pero tienen relación con un mismo tópico. Existen algoritmos para la detección de tópicos emergentes que podrían fortalecer el método de detección propuesto. Con esto, la orientación de la detección sería en base a tópicos y no a consultas aisladas como en este trabajo.

3. *Implementación Real*. La evidencia empírica muestra que los mecanismos propuestos mejoran el desempeño en el CS, por lo que una arista pendiente es la implementación de estas estrategias en un contexto real y probar su factibilidad. Según los experimentos llevados a cabo en un cluster, existe ganancia en la implementación de algunos aspectos, pero queda una implementación y prueba más exhaustiva de los métodos propuestas en este trabajo. En particular, la construcción de un balanceador de carga es un desafío importante, ya que se necesita contar con la insfraestructura suficiente (cluster de computadores

compuesto por decenas de nodos). *Memcached* es la herramienta más popular para desplegar este tipo de servicios, por lo que es esta herramienta en donde se debería implementar el mecanismo de protección.

4. *Nodos de Cache*. Una arista importante no explorada en este trabajo es la mejora de un nodo de cache mediante la utilización de programación concurrente, es decir, utilizar eficientemente los núcleos de procesamiento de un nodo. Esto es posible a través de la utilización de hilos y estructuras libres de *deadlock* a nivel de nodo. Las alzas abruptas de tráfico hacia un nodo también podrían ser tratadas, por ejemplo a través del modelo BSP [Val90] (*Bulk Synchronous Parallel*). Otra opción es agrupar y organizar las operaciones sobre la estructura cache para reducir los bloqueos y esperas sobre ella.

5. *Componentes Tendenciales, Estacionales y/o Aleatorias*. Las consultas permanentes poseen distintas componentes que caracterizan su comportamiento. Según lo observado en los datos utilizados, las consultas permanentes no poseen tendencia sino que presentan estacionalidad. Por un lado existen a nivel diario, pero otro otro lado también se observó a nivel de semana. En el otro extremo, las consultas en ráfagas no presentaron estos componentes, ya que estan relacionadas a eventos específicos, no siguen patrones. Estudiar y aplicar los mecanismos del estado del arte para analizar las series y extraer sus características, permitiría obtener más conocimiento sobre ellas, y podrían ser incluidas en los mecanismos de balance de carga para mejorar el desempeño.

9.2. Implicancias Prácticas

Tanto la replicación total como la no replicación de ítemes en un Servicio de Cache implican una degradación de la eficiencia (desbalance y reducción de la tasa de *hit*). La primera implicancia práctica de este trabajo es que deben existir mecanismos adaptivos para la replicación controlada de ítemes, con lo que se llega a un punto intermedio entre los dos enfoques descritos anteriormente (replicación nula y total). La solución adoptada en un contexto real debe tomar en consideración el volumen

de tráfico y el impacto de los distintos ítemes, para lo que se necesitan algoritmos y estructuras de datos eficientes en tiempo y espacio. Este mecanismo es aplicable a cualquier servicio a gran escala en que la carga de trabajo generada en los nodos sea guiada por el tráfico de usuario (redes sociales, servicios de *microblogging*, etc.).

El balance de carga es un mecanismo clásico que se aplica en sistemas distribuidos. El enfoque dinámico es el indicado cuando existen variaciones en la carga de trabajo experimentada por los nodos. En un Servicio de Cache, y en general en aplicaciones y servicios Web de gran escala, se debe monitorear constantemente el estado de los nodos para determinar el estado global del servicio, con lo que es posible utilizar estrategias dinámicas para corregir la carga de trabajo en línea. Las virtudes de un servicio balanceado son múltiples, entre las que se pueden destacar la reducción de fallos y el aumento de la eficiencia en la utilización de recursos.

En la mayoría de las aplicaciones que lidian con peticiones generadas por usuarios existen alzas abruptas de distintas magnitud. Independiente de la aplicación, esto representa un peligro ya que una alta cantidad de requerimientos aparece en un intervalo de tiempo reducido. La recomendación para la gestión de aplicaciones y servicios Web es establecer mecanismos para la detección temprana de alzas de tráfico y definir medidas de contingencia ante estas alzas. La detección temprana se realiza examinando el tráfico cada cierto intervalo de tiempo y extrayendo características de él. Por otro lado, las medidas de contingencia tienen relación con el aumento de la capacidad del servicio o con la atención de un conjunto de peticiones frecuentes por parte de múltiples nodos.

Bibliografía

[AAO+11] Sadiye Alici, Ismail Sengor Altingovde, Rifat Ozcan, B. Barla Cambazoglu, and Özgür Ulusoy. Timestamp-based cache invalidation for search engines. In *Proceedings of the 20th international conference companion on World wide web*, WWW '11, pages 3–4, New York, NY, USA, 2011. ACM.

[ABKU99] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, September 1999.

[Ada10] John Adams. Billions of hits: Scaling twitter. *slide presentation presented at the Chirp*, 2010.

[AJZ11] Vijay Kumar Adhikari, Sourabh Jain, and Zhi-Li Zhang. Where do you "tube"? uncovering YouTube server selection strategy. In Haohong Wang, Jin Li, George N. Rouskas, and Xiaobo Zhou, editors, *ICCCN*, pages 1–6. IEEE, 2011.

[AXF+12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. *SIGMETRICS Perform. Eval. Rev.*, 40(1):53–64, June 2012.

[BAA+10] Claudine Santos Badue, Jussara M. Almeida, Virgilio Almeida, Ricardo A. Baeza-Yates, Berthier A. Ribeiro-Neto, Artur Ziviani, and Nivio Ziviani. Capacity planning for vertical search engines. *CoRR*, abs/1006.5059, 2010.

[BBJ+10] Roi Blanco, Edward Bortnikov, Flavio Junqueira, Ronny Lempel, Luca Telloli, and Hugo Zaragoza. Caching search engine results over incremental indices. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '10, pages 82–89, New York, NY, USA, 2010. ACM.

[BCF+99] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. 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*, volume 1, pages 126–134 vol.1, 1999.

[BCM03] John Byers, Jeffrey Considine, and Michael Mitzenmacher. Simple load balancing for distributed hash tables. In M.Frans Kaashoek and Ion Stoica, editors, *Peer-to-Peer Systems II*, volume 2735 of *Lecture Notes in Computer Science*, pages 80–87. Springer Berlin Heidelberg, 2003.

[BCSV04] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: a scalable fully distributed web crawler. *Softw. Pract. Exper.*, 34(8):711–726, July 2004.

[BD02] Peter J Brockwell and Richard A Davis. *Introduction to time series and forecasting*, volume 1. Taylor & Francis, 2002.

[BDH03] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, March 2003.

[BFPS11] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, IGCC '11, pages 1–8, Washington, DC, USA, 2011. IEEE Computer Society.

[BH07] L.A. Barroso and U. Holzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, April 1998.

[BYCMR05] Ricardo Baeza-Yates, Carlos Castillo, Mauricio Marin, and Andrea Rodriguez. Crawling a country: better strategies than breadth-first for web page ordering. In *Special interest tracks and posters of the 14th international conference on World Wide Web*, WWW '05, pages 864–872, New York, NY, USA, 2005. ACM.

[BYGJ+07] Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. The impact of caching on search engines. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '07, pages 183–190, New York, NY, USA, 2007. ACM.

[BYGJ+08] Ricardo Baeza-Yates, Aristides Gionis, Flavio P. Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. Design trade-offs for search engine caching. *ACM Trans. Web*, 2(4):20:1–20:28, October 2008.

[BYJ12] Ricardo Baeza-Yates and Simon Jonassen. Modeling static caching in web search engines. In *Proceedings of the 34th European conference on Advances in Information Retrieval*, ECIR'12, pages 436–446, Berlin, Heidelberg, 2012. Springer-Verlag.

[BYJPW07] Ricardo Baeza-Yates, Flavio Junqueira, Vassilis Plachouras, and Hans Friedrich Witschel. Admission policies for caches of search engine results. In *Proceedings of the 14th international conference on String processing and information retrieval*, SPIRE'07, pages 74–85, Berlin, Heidelberg, 2007. Springer-Verlag.

[BYRN99] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[CA12] B. Barla Cambazoglu and Ismail Sengor Altingovde. Impact of regionalization on performance of web search engine result caches. In *Proceedings of the 19th international conference on String Processing and Information Retrieval*, SPIRE'12, pages 161–166, Berlin, Heidelberg, 2012. Springer-Verlag.

[Can03] G.C. Canavos. *Probabilidad y estadística: aplicaciones y métodos*. McGraw-Hill, 2003.

[Cap09] Franck Cappello. Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities. *Int. J. High Perform. Comput. Appl.*, 23(3):212–226, August 2009.

[Cas05] Carlos Castillo. Effective web crawling. In *ACM SIGIR Forum*, volume 39, pages 55–56. ACM, 2005.

[CCFC02] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, ICALP '02, pages 693–703, London, UK, UK, 2002. Springer-Verlag.

[CCFC04] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3 – 15, 2004. Automata, Languages and Programming.

[CCT12] Ling Chen, Yixin Chen, and Li Tu. A fast and efficient algorithm for finding frequent items over data stream. *JCP*, 7(7):1545–1554, 2012.

[CGMP98] Junghoo Cho, Hector Garcia-Molina, and Lawrence Page. Efficient crawling through url ordering. *Computer Networks and ISDN Systems*, 30(1):161–172, 1998.

[CH08] Graham Cormode and Marios Hadjieleftheriou. Finding frequent items in data streams. *Proc. VLDB Endow.*, 1(2):1530–1541, August 2008.

[CH09] Graham Cormode and Marios Hadjieleftheriou. Finding the frequent items in streams of data. *Commun. ACM*, 52(10):97–105, October 2009.

[CH10] Graham Cormode and Marios Hadjieleftheriou. Methods for finding frequent items in data streams. *VLDB J.*, 19(1):3–20, 2010.

[CJP+10] Berkant Barla Cambazoglu, Flavio P. Junqueira, Vassilis Plachouras, Scott Banachowski, Baoqiu Cui, Swee Lim, and Bill Bridge. A refreshing perspective of search engine caching. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 181–190, New York, NY, USA, 2010. ACM.

[CLV11] G. Chockler, G. Laden, and Y. Vigfusson. Design and implementation of caching services in the cloud. *IBM J. Res. Dev.*, 55(6):422–432, November 2011.

[CM05] Graham Cormode and S. Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. *ACM Trans. Database Syst.*, 30(1):249–278, 2005.

[CMS10] W Bruce Croft, Donald Metzler, and Trevor Strohman. *Search engines: Information retrieval in practice*. Addison-Wesley Reading, 2010.

[CRS99] Boris Chidlovskii, Claudia Roncancio, and Marie-Luise Schneider. Semantic cache mechanism for heterogeneous web querying. *Computer Networks*, 31(11-16):1347–1360, 1999.

[Cyb89] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.*, 7(2):279–301, October 1989.

[Den70] Peter J. Denning. Virtual memory. *ACM Comput. Surv.*, 2(3):153–189, September 1970.

[DHJ+07a] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41:205–220, October 2007.

[DHJ+07b] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, 2007.

[DR01] P. Druschel and A. Rowstron. Past: a large-scale, persistent peer-to-peer storage utility. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 75–80, 2001.

[FC07] Wu-chun Feng and Kirk Cameron. The green500 list: Encouraging sustainable supercomputing. *Computer*, 40(12):50–55, December 2007.

[Fit04] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, August 2004.

[FMM09] Flavio Ferrarotti, Mauricio Marin, and Marcelo Mendoza. A last-resort semantic cache for web queries. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval*, SPIRE '09, pages 310–321, Berlin, Heidelberg, 2009. Springer-Verlag.

[FPSO06] Tiziano Fagni, Raffaele Perego, Fabrizio Silvestri, and Salvatore Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans. Inf. Syst.*, 24(1):51–78, January 2006.

[FSD+09] Leonardo Fialho, Guna Santos, Angelo Duarte, Dolores Rexachs, and Emilio Luque. Challenges and issues of the integration of radic into open mpi. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 73–83, Berlin, Heidelberg, 2009. Springer-Verlag.

[FYYL05] Gabriel Pui Cheong Fung, Jeffrey Xu Yu, Philip S. Yu, and Hongjun Lu. Parameter free bursty events detection in text streams. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 181–192. VLDB Endowment, 2005.

[GC10] F. Gabbiani and S.J. Cox. *Mathematics for Neuroscientists*. Elsevier science & technology books. Elsevier Science, 2010.

[GCIPMF13] Veronica Gil-Costa, Alonso Inostrosa-Psijas, Mauricio Marin, and Esteban Feuestein. Service deployment algorithms for vertical search engines. In *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, PDP '13, pages 140–147, Washington, DC, USA, 2013. IEEE Computer Society.

[GCLIPM12] Veronica Gil-Costa, Jair Lobos, Alonso Inostrosa-Psijas, and Mauricio Marin. Capacity planning for vertical search engines: An approach based on coloured petri nets. In Serge Haddad and Lucia Pomello, editors, *Application and Theory of Petri Nets*, volume 7347 of *Lecture Notes in Computer Science*, pages 288–307. Springer Berlin Heidelberg, 2012.

[GDD+03] Lukasz Golab, David DeHaan, Erik D. Demaine, Alejandro Lopez-Ortiz, and J. Ian Munro. Identifying frequent items in sliding windows over on-line packet streams. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, IMC '03, pages 173–178, New York, NY, USA, 2003. ACM.

[GLTT01] Daniel Goldberg, Ming Li, Wenchao Tao, and Yuval Tamir. The design and implementation of a fault-tolerant cluster manager. *Tech. Rep., Xerox Systems Institute*, 2001.

[GPGCR+12] Carlos Gómez-Pantoja, Veronica Gil-Costa, Dolores Rexachs, Mauricio Marin, and Emilio Luque. A fault-tolerant cache service for web search engines. In *Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, ISPA '12, pages 427–434, Washington, DC, USA, 2012. IEEE Computer Society.

[GPM08] Carlos Gómez-Pantoja and Mauricio Marín. Load balancing distributed inverted files: Query ranking. In *PDP*, pages 329–333. IEEE Computer Society, 2008.

[GPMCB11] Carlos Gómez-Pantoja, Mauricio Marín, Veronica Gil Costa, and Carolina Bonacic. An evaluation of fault-tolerant query processing for web search engines. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par (1)*, volume 6852 of *Lecture Notes in Computer Science*, pages 393–404. Springer, 2011.

[GPRML12] Carlos Gómez-Pantoja, Dolores Rexachs, Mauricio Marin, and Emilio Luque. A fault-tolerant cache service for web search engines: Radic evaluation. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 298–310, Berlin, Heidelberg, 2012. Springer-Verlag.

[GS96] Rachid Guerraoui and André Schiper. Fault-tolerance by replication in distributed systems. In Alfred Strohmeier, editor, *Reliable Software Technologies Ada Europe '96*, volume 1088 of *Lecture Notes in Computer Science*, pages 38–57. Springer Berlin Heidelberg, 1996.

[GS09] Qingqing Gan and Torsten Suel. Improved techniques for result caching in web search engines. In *Proceedings of the 18th international*

conference on World wide web, WWW '09, pages 431–440, New York, NY, USA, 2009. ACM.

[HC10] Nuno Homem and Joao Paulo Carvalho. Finding top-k elements in data streams. *Inf. Sci.*, 180(24):4958–4974, December 2010.

[HC11] Nuno Homem and João Paulo Carvalho. Finding top-k elements in a time-sliding window. *Evolving Systems*, 2(1):51–70, 2011.

[HHZ+ch] Zonghao Hou, Yongxiang Huang, Shouqi Zheng, Xiaoshe Dong, and Bingkang Wang. Design and implementation of heartbeat in multi-machine environment. In *Advanced Information Networking and Applications, 2003. AINA 2003. 17th International Conference on*, pages 583–586, March.

[HLM+90] S. H. Hosseini, B. Litow, M. Malkawi, J. McPherson, and K. Vairavan. Analysis of a graph coloring based distributed load balancing algorithm. *J. Parallel Distrib. Comput.*, 10(2):160–166, September 1990.

[IJSE07] Alexandru Iosup, Mathieu Jan, Ozan Sonmez, and Dick H. J. Epema. On the dynamic resource availability in grids. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, GRID '07, pages 26–33, Washington, DC, USA, 2007. IEEE Computer Society.

[KA99] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, December 1999.

[KC10] Anagha Kulkarni and Jamie Callan. Document allocation policies for selective searching of distributed indexes. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, CIKM '10, pages 449–458, New York, NY, USA, 2010. ACM.

[KK03] M.Frans Kaashoek and DavidR. Karger. Koorde: A simple degree-optimal distributed hash table. In M.Frans Kaashoek and Ion Stoica, editors, *Peer-to-Peer Systems II*, volume 2735 of *Lecture Notes in Computer Science*, pages 98–107. Springer Berlin Heidelberg, 2003.

[KKP+09] Marcel Karnstedt, Daniel Klan, Christian Pölitz, Kai-Uwe Sattler, and Conny Franke. Adaptive burst detection in a stream engine. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1511–1515, New York, NY, USA, 2009. ACM.

[Kle03] Jon Kleinberg. Bursty and hierarchical structure in streams. *Data Min. Knowl. Discov.*, 7(4):373–397, October 2003.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[KSB+99] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web caching with consistent hashing. *Comput. Netw.*, 31(11-16):1203–1213, May 1999.

[KSP03] Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28(1):51–55, March 2003.

[LCP+05] Eng Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys Tutorials, IEEE*, 7(2):72–93, 2005.

[LGTT01] Ming Li, Daniel Goldberg, Wenchao Tao, and Yuval Tamir. Fault-tolerant cluster management for reliable high-performance computing.

In *Proceedings of International Conference on Parallel and Distributed Computing and Systems*, pages 480–485, 2001.

[LhLH+10] Hui Li, Cun hua Li, Yun Hu, Shu Zhang, and Xia Wang. Admission polices for outperforming the efficiency of search engines. In *Intelligent Computing and Intelligent Systems (ICIS), 2010 IEEE International Conference on*, volume 1, pages 445 –449, oct. 2010.

[LK87] F.C.H. Lin and R.M. Keller. The gradient model load balancing method. *Software Engineering, IEEE Transactions on*, SE-13(1):32 – 38, jan. 1987.

[LM03] Ronny Lempel and Shlomo Moran. Predictive caching and prefetching of query results in search engines. In *Proceedings of the 12th international conference on World Wide Web*, WWW '03, pages 19–28, New York, NY, USA, 2003. ACM.

[LM04] Ronny Lempel and Shlomo Moran. Competitive caching of query results in search engines. *Theor. Comput. Sci.*, 324(2-3):253–271, 2004.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[LPN+11] Kathy Lee, Diana Palsetia, Ramanathan Narayanan, Md. Mostofa Ali Patwary, Ankit Agrawal, and Alok Choudhary. Twitter trending topic classification. In *Proceedings of the 2011 IEEE 11th International Conference on Data Mining Workshops*, ICDMW '11, pages 251–258, Washington, DC, USA, 2011. IEEE Computer Society.

[LS05] Xiaohui Long and Torsten Suel. Three-level caching for efficient query processing in large web search engines. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 257–266, New York, NY, USA, 2005. ACM.

[LS06] Xiaohui Long and Torsten Suel. Three-level caching for efficient query processing in large web search engines. *World Wide Web*, 9(4):369–395, 2006.

[LVD+11] Tonglin Li, Raman Verma, Xi Duan, Hui Jin, and Ioan Raicu. Exploring distributed hash tables in highend computing. *SIGMETRICS Perform. Eval. Rev.*, 39(3):128–130, December 2011.

[LWX06] Yutong Lu, Min Wang, and Nong Xiao. A new heartbeat mechanism for large-scale cluster. In HengTao Shen, Jinbao Li, Minglu Li, Jun Ni, and Wei Wang, editors, *Advanced Web and Network Technologies, and Applications*, volume 3842 of *Lecture Notes in Computer Science*, pages 610–619. Springer Berlin Heidelberg, 2006.

[MAA06] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Trans. Database Syst.*, 31(3):1095–1133, September 2006.

[MAEA05] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the 10th international conference on Database Theory*, ICDT'05, pages 398–412, Berlin, Heidelberg, 2005. Springer-Verlag.

[Mar01] E.P Markatos. On caching search engine query results. *Comput. Commun.*, 24(2):137–143, February 2001.

[Mar04] M. Marzolla. Libcppsim: A simula-like, portable process-oriented simulation library in c++. In *Proceedings of ESM*, 2004.

[MFM+09] Mauricio Marin, Flavio Ferrarotti, Marcelo Mendoza, Carlos Gomez-Pantoja, and Veronica Gil-Costa. Location cache for web queries. In *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 1995–1998, New York, NY, USA, 2009. ACM.

[MG82] Jayadev Misra and David Gries. Finding repeated elements. Technical report, Ithaca, NY, USA, 1982.

[MG07] Mauricio Marin and Carlos Gomez. Load balancing distributed inverted files. In *Proceedings of the 9th Annual ACM International Workshop on Web Information and Data Management*, WIDM '07, pages 57–64, New York, NY, USA, 2007. ACM.

[MGCBS13] Mauricio Marin, Veronica Gil-Costa, Carolina Bonacic, and Roberto Solar. Approximate parallel simulation of web search engines. In *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation*, SIGSIM-PADS '13, pages 189–200, New York, NY, USA, 2013. ACM.

[MGCGP10a] Mauricio Marin, Veronica Gil-Costa, and Carlos Gomez-Pantoja. New caching techniques for web search engines. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 215–226, New York, NY, USA, 2010. ACM.

[MGCGP10b] Mauricio Marin, Veronica Gil-Costa, and Carlos Gomez-Pantoja. New caching techniques for web search engines. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 215–226, New York, NY, USA, 2010. ACM.

[MM02a] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 346–357. VLDB Endowment, 2002.

[MM02b] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from*

the First International Workshop on Peer-to-Peer Systems, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag.

[MP09] Nishad Manerikar and Themis Palpanas. Frequent items in streaming data: An experimental evaluation of the state-of-the-art. *Data Knowl. Eng.*, 68(4):415–430, April 2009.

[MRS00] Michael Mitzenmacher, Andréa W. Richa, and Ramesh Sitaraman. The power of two random choices: A survey of techniques and results. In *in Handbook of Randomized Computing*, pages 255–312. Kluwer, 2000.

[MRS08] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[NFG+13] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, nsdi'13, pages 385–398, Berkeley, CA, USA, 2013. USENIX Association.

[NO08] Lucas Nealan and Portland OR OSCon. Caching & performance: Lessons from facebook. O'Reilly Open-Source Convention (OSCon), 2008.

[NS09] Mitra Nasri and Mohsen Sharifi. Load balancing using consistent hashing: A real challenge for large scale distributed web crawlers. In *Proceedings of the 2009 International Conference on Advanced Information Networking and Applications Workshops*, WAINA '09, pages 715–720, Washington, DC, USA, 2009. IEEE Computer Society.

[OAU11] Rifat Ozcan, Ismail Sengor Altingovde, and Özgür Ulusoy. Cost-aware strategies for query result caching in web search engines. volume 5, pages 9:1–9:25. ACM, New York, NY, USA, May 2011.

[OSABC+12] Rifat Ozcan, I. Sengor Altingovde, B. Barla Cambazoglu, Flavio P. Junqueira, and ÖZgüR Ulusoy. A five-level static cache architecture for web search engines. *Inf. Process. Manage.*, 48(5):828–840, September 2012.

[PB03] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, December 2003.

[PS08] Nish Parikh and Neel Sundaresan. Scalable and near real-time burst detection from ecommerce queries. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '08, pages 972–980, New York, NY, USA, 2008. ACM.

[PSL06] Diego Puppin, Fabrizio Silvestri, and Domenico Laforenza. Query-driven document partitioning and collection selection. In *Proceedings of the 1st international conference on Scalable information systems*, InfoScale '06, New York, NY, USA, 2006. ACM.

[RD01a] Antony Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. *SIGOPS Oper. Syst. Rev.*, 35(5):188–201, October 2001.

[RD01b] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK, 2001. Springer-Verlag.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, August 2001.

[RHHR09] Costin Raiciu, Felipe Huici, Mark Handley, and David S. Rosenblum. Roar: increasing the flexibility and performance of distributed search. *SIGCOMM Comput. Commun. Rev.*, 39:291–302, 2009.

[RHM12] Erika Rosas, Nicolas Hidalgo, and Mauricio Marin. Two-level result caching for web search queries on structured p2p networks. In *ICPADS*, pages 221–228. IEEE Computer Society, 2012.

[RHMGC13] Erika Rosas, Nicolas Hidalgo, Mauricio Marin, and Veronica Gil-Costa. Web search results caching service for structured {P2P} networks. *Future Generation Computer Systems*, (0):–, 2013.

[RL10] Dolores Rexachs and Emilio Luque. High availability for parallel computers. volume 10, pages 110–116, 2010.

[RS04] Venugopalan Ramasubramanian and Emin Gün Sirer. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 8–8, Berkeley, CA, USA, 2004. USENIX Association.

[Saa] Paul Saab. Scaling memcached at Facebook. http://www.facebook.com/note.php?note_id=39391378919.

[SC10] Ilija Subasic and Carlos Castillo. The effects of query bursts on web search. In *Proceedings of the 2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology - Volume 01*, WI-IAT '10, pages 374–381, Washington, DC, USA, 2010. IEEE Computer Society.

[SC13] Ilija Subasic and Carlos Castillo. Investigating query bursts in a web search engine. *Web Intelligence and Agent Systems*, 11:107–124, January 2013.

[SDRL08] Guna Santos, Angelo Duarte, Dolores Rexachs, and Emilio Luque. Providing non-stop service for message-passing based parallel applications with radic. In Emilio Luque, Tomàs Margalef, and Domingo Benitez, editors, *Euro-Par*, volume 5168 of *Lecture Notes in Computer Science*, pages 58–67. Springer, 2008.

[SG06] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pages 249–258, Washington, DC, USA, 2006. IEEE Computer Society.

[SH06] Osama Saleh and Mohamed Hefeeda. Modeling and caching of peer-to-peer traffic. In *ICNP*, pages 249–258. IEEE Computer Society, 2006.

[SJPBY08] Gleb Skobeltsyn, Flavio Junqueira, Vassilis Plachouras, and Ricardo Baeza-Yates. 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*, SIGIR '08, pages 131–138, New York, NY, USA, 2008. ACM.

[SMHM99] Craig Silverstein, Hannes Marais, Monika Henzinger, and Michael Moricz. Analysis of a very large web search engine query log. *SIGIR Forum*, 33(1):6–12, September 1999.

[SMLN+03] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, February 2003.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, March 2005.

[SSdMZ+01] Paricia Correia Saraiva, Edleno Silva de Moura, Novio Ziviani, Wagner Meira, Rodrigo Fonseca, and Berthier Riberio-Neto. 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*, SIGIR '01, pages 51–58, New York, NY, USA, 2001. ACM.

[Sta01] William Stallings. *Operating systems - internals and design principles (4. ed.)*. Prentice Hall, 2001.

[SXC06] Haiying Shen, Cheng-Zhong Xu, and Guihai Chen. Cycloid: a constant-degree and lookup-efficient p2p overlay network. *Perform. Eval.*, 63(3):195–216, March 2006.

[Tre05] Michael Treaster. A survey of fault-tolerance and fault-recovery techniques in parallel systems. *CoRR*, abs/cs/0501002, 2005.

[Tri09] Mario F. Triola. *Estadística*. Pearson Educación, 2009.

[TS06] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[URL] http://memcached.org/.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[VDA+98] Werner Vogels, Dan Dumitriu, Ashutosh Agrawal, Teck Chia, and Katherine Guo. Scalability of the microsoft cluster service. In *Proceedings of the 2nd conference on USENIX Windows NT Symposium - Volume 2*, WINSYM'98, pages 2–2, Berkeley, CA, USA, 1998. USENIX Association.

[VMVG04] Michail Vlachos, Christopher Meek, Zografoula Vagena, and Dimitrios Gunopulos. Identifying similarities, periodicities and bursts for online search queries. In *Proceedings of the 2004 ACM SIGMOD international*

conference on Management of data, SIGMOD '04, pages 131–142, New York, NY, USA, 2004. ACM.

[Wan99] Jia Wang. A survey of web caching schemes for the internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46, October 1999.

[WLR93] M. H. Willebeek-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans. Parallel Distrib. Syst.*, 4(9):979–993, September 1993.

[WLY+13] Jianguo Wang, Eric Lo, Man Lung Yiu, Jiancong Tong, Gang Wang, and Xiaoguang Liu. The impact of solid state drive on search engine cache management. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '13, pages 693–702, New York, NY, USA, 2013. ACM.

[WMES08] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive process-level live migration in hpc environments. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 43:1–43:12, Piscataway, NJ, USA, 2008. IEEE Press.

[WS97] Min-You Wu and Wei Shu. Dde: A modified dimension exchange method for load balancing in k-ary n-cubes. *Journal of Parallel and Distributed Computing*, 44(1):88 – 96, 1997.

[XL92] C. Z. Xu and F. C. M. Lau. Analysis of the generalized dimension exchange method for dynamic load balancing. *Journal of Parallel and Distributed Computing*, 16:385–393, 1992.

[XL95] Cheng-Zhong Xu and Francis C. M. Lau. The generalized dimension exchange method for load balancing in k-ary n-cubes and variants. *J. Parallel Distrib. Comput.*, 24(1):72–85, January 1995.

[XO02] Yinglian Xie and David O'Hallaron. Locality in search engine queries and its implications for caching. In *INFOCOM 2002. Twenty-First*

Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, volume 3, pages 1238–1247. IEEE, 2002.

[YC07] Sung Goo Yoo and Kil-To Chong. Hot spot prediction algorithm for shared web caching system using nn. In *Information Technology Convergence, 2007. ISITC 2007. International Symposium on*, pages 125–129, 2007.

[YV00] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 21–21, Berkeley, CA, USA, 2000. USENIX Association.

[ZHS+04] B.Y. Zhao, Ling Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *Selected Areas in Communications, IEEE Journal on*, 22(1):41–53, 2004.

[Zip35] George Kingsley Zipf. *The Psychobiology of Language, an Introduction to Dynamic Philology.* Houghton-Mifflin, Boston, 1935.

[Zip49] George Kingsley Zipf. *Human behavior and the principle of least effort: an introduction to human ecology.* Addison-Wesley Press, 1949.

[ZLS08] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th international conference on World Wide Web*, WWW '08, pages 387–396, New York, NY, USA, 2008. ACM.

[ZM06] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM computing surveys (CSUR)*, 38(2):6, 2006.

[ZS03] Yunyue Zhu and Dennis Shasha. Efficient elastic burst detection in data streams. In *Proceedings of the ninth ACM SIGKDD international*

conference on Knowledge discovery and data mining, KDD '03, pages 336–345, New York, NY, USA, 2003. ACM.

[ZS06] Xin Zhang and D. Shasha. Better burst detection. In *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*, pages 146–146, 2006.

[ZS07] J. Zhang and T. Suel. Optimized inverted list assignment in distributed search engine architectures. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1 –10, march 2007.

[zXL94] Cheng zhong Xu and Francis C.M. Lau. Iterative dynamic load balancing in multicomputers. *Journal of Operational Research Society*, 45:786–796, 1994.