Indexación y Compresión de Datos para motores de búsqueda

Diego Arroyuelo

Académico Departamento de Informática, Universidad Técnica Federico Santa María. Doctor en Ciencias Mención Computación, Universidad de Chile; Licenciado en Ciencias de la Computación, Universidad Nacional de San Luis, Argentina. Líneas de Investigación: Estructuras de Datos y Algoritmos, Compresión y Búsqueda en Texto, Estructuras de Datos Comprimidas, Índices Comprimidos para Recuperación de la Información.

darroyue@inf.utfsm.cl

Probablemente muy pocos usuarios de un motor de búsqueda Web se han detenido a pensar, respecto de los recursos y tecnologías involucrados en la generación de la página de resultados de su consulta. La misma debe ser generada en unas pocas décimas de segundo y debe contener diez resultados relevantes para el usuario, ojalá los más relevantes. Además, debajo del título y la URL de cada uno de los resultados encontrados, la página debe mostrar pequeñas porciones del documento ─conocidas como *snippets*─ que contienen las palabras usadas en la consulta. Para complicar más la situación, usualmente decenas de miles de consultas arriban por segundo a un motor de búsqueda (aproximadamente 38 mil consultas por segundo para Google en 2012, lo que equivale a 2.4 millones por minuto, o 3.3 mil millones por día [1, 2]). Además, estas búsquedas deben realizarse sobre varias decenas de miles de millones de páginas web [3]. En este escenario, tanto la Indexación como la Compresión de Datos son indispensables para el funcionamiento de los motores de búsqueda actuales. Esto impacta en los tiempos de respuesta a las consultas, en el ahorro de espacio de almacenamiento, en la consecuente reducción del uso de espacio físico, en la reducción de los tiempos de transferencia de datos y en el ahorro de energía usada [14].

En el presente artículo mostraré los avances más recientes y los desafíos más importantes en el área de Indexación y Compresión de Datos en motores de búsqueda, particularmente aquellos en los que he estado involucrado con mi investigación. Intentaré dar pistas que permitan comprender cómo los motores de búsqueda actuales son capaces de manejar grandes volúmenes de datos y la carga de trabajo antes mencionada. Todos los resultados experimentales aquí mostrados, han sido obtenidos dentro del marco de mi grupo de investigación y validados con los resultados similares en el estado del arte.

Dada una base de datos de documentos de texto (las páginas web), denotamos con v al vocabulario con todas las palabras distintas que aparecen en dichos documentos. Para facilitar su manipulación, a cada documento se le asigna un identificador (o docID), que es un número entero que lo identifica. Los usuarios presentan consultas q a la base de datos, las que constan de una o más palabras. El objetivo es encontrar documentos relacionados con esas palabras. Existen dos variantes principales de consultas: las consultas tipo AND (que buscan los documentos que contengan a todas las palabras de q) y las consultas tipo OR (que buscan los documentos que contengan *al menos* una de las palabras de q). Para mostrar los resultados en orden de relevancia para el usuario, el motor de búsqueda debe hacer un ranking de los mismos. Ésta es una etapa muy importante para la búsqueda. Sin embargo, en este artículo no discutiremos aspectos relacionados con el ranking de resultados.

LA INDEXACIÓN DE BASES DE DATOS DE DOCUMENTOS

Los documentos de la base de datos se conocen de antemano a las búsquedas, por lo que se pueden indexar, esto es, construir una estructura de datos que acelere las respuestas a las consultas. Las estructuras de datos clásicas en este caso son los índices invertidos [9, 13, 22, 30]. Por cada palabra p del vocabulario, el índice invertido tiene una lista invertida v que almacena los docIDs de los documentos que contienen a p. Esto es similar al índice que podemos encontrar al final de la mayoría de los libros técnicos. Por cada docID en $L(p)$, también se almacena información útil para el ranking de resultados, como la cantidad de ocurrencias de p en cada uno de los documentos.

El grueso de las listas invertidas usualmente se almacena en memoria secundaria, aunque algunas listas (generalmente las más consultadas) se almacenan en memoria principal, en lo que se llama el caché de listas invertidas. Muchos motores de búsqueda incluso precalculan los resultados de las consultas más frecuen-

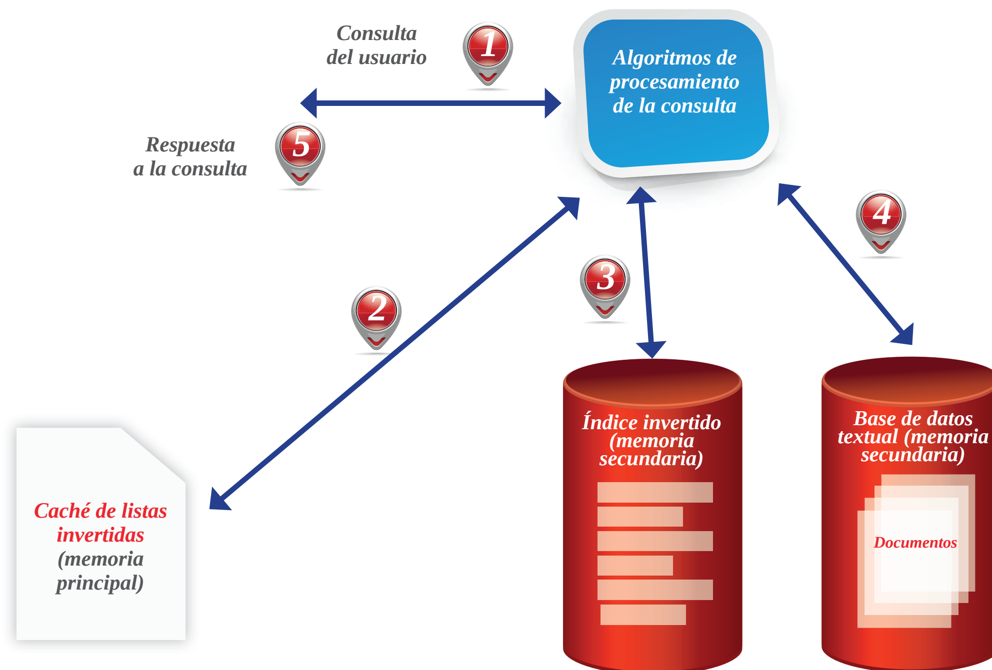


Figura 1 • Pasos para responder una consulta en un motor de búsqueda.

tes y las almacenan en caché. El motor de búsqueda también debe almacenar la base de datos de documentos en memoria secundaria. Ésta es necesaria tanto para la generación de los snippets como para mostrar la versión "en caché" de la página, entre otros usos. La Figura 1 resume el proceso de búsqueda, con los siguientes pasos: (1) La consulta llega al procesador de consultas; (2) el procesador intenta resolver la consulta usando las listas almacenadas en caché; (3) si el caché no contiene la lista invertida de alguna de las palabras de la consulta, ésta debe recuperarse desde la memoria secundaria; (4) después de usar el índice para encontrar las diez mejores páginas para la respuesta (pueden ser más), éstas deben recuperarse desde la memoria secundaria para generar los snippets; (5) la página de respuesta es enviada al usuario.

A continuación estudiaremos cómo se logra realizar el proceso de la Figura 1 de manera eficiente (en menos de un segundo el usuario debe tener su respuesta), describiendo los principales desafíos que se presentan.

EL DESAFÍO DE COMPRIMIR LOS ÍNDICES DE BÚSQUEDA

Las listas invertidas requieren usualmente una gran cantidad de espacio de almacenamiento. Por ejemplo, las listas invertidas de docIDs para la reconocida colección de documentos TREC GOV2 requieren aproximadamente 23 GB, si usamos un entero de 32 bits por cada docID (en total son 5.750 millones de docIDs en todo el índice). Dicha colección contiene páginas del dominio .gov de Estados Unidos, con 25 millones de documentos y un total de 426 GB. A raíz de esto, es común que las listas invertidas se mantengan en el disco, como se mencionó anteriormente. Dada una consulta q, las listas invertidas de cada palabra de la consulta se transfieren a memoria principal para ser procesadas. Sin embargo, la baja latencia de acceso a la memoria secundaria puede influir negativamente en la latencia de una consulta.

Una solución a este problema es comprimir las listas invertidas [13, 30]. Esto permite emplear de mejor manera el espacio disponible para el caché de listas. La compresión también produce una importante mejora en el tiempo de transferencia de las listas desde el disco, ya que se transfieren menos bytes (véase, por ejemplo, el estudio en [13, sección 6.3.6]). En ambos casos, la compresión impacta favorablemente en el tiempo de respuesta a los usuarios.

La compresión de datos sin pérdida (es decir, aquella en la que al descomprimir se obtienen exactamente los mismos datos originales), en general funciona detectando regularidades en los datos a comprimir. Por ejemplo, la compresión de textos generalmente se basa en detectar las regularidades del lenguaje escrito y así reducir el espacio de la codificación. En el caso de las listas invertidas, la compresión se logra usando lo que se conoce como *gap encoding:* los docIDs se almacenan ordenados de manera creciente en las listas invertidas. Luego, se almacena el primer docID en la lista de manera absoluta (es decir, tal cual es). El resto de los elementos de la lista, en cambio, se almacenan como la diferencia entre docIDs consecutivos en la lista (esas diferencias se conocen como gaps). Como la lista original está ordenada por docIDs crecientes, el resultado es una lista de números (gaps) enteros positivos. Posteriormente, las listas se dividen en bloques de tamaño constante (por ejemplo, 128 docIDs por bloque) y se usa un compresor de números enteros sobre los gaps [5, 13, 14, 16, 19, 23, 29, 30, 33].

Dichos compresores codifican un número entero en una cantidad de bits proporcional al valor del entero: mientras menor es su valor, menos bits se usan. Dado que almacenamos los gaps en lugar de los docIDs, normalmente se obtienen distribuciones con muchos valores pequeños, logrando comprimir las listas. La división en bloques de las listas se hace para no descomprimir toda la lista en tiempo de búsqueda, sino sólo los bloques que contienen docIDs relevantes para una consulta. Los compresores más efectivos son aquellos que permiten una alta velocidad de descompresión —dado que esto impacta en el tiempo de respuesta— sumado a una tasa de compresión razonable, aunque no necesariamente óptima. Aquí se destacan los métodos VByte [29], S9 [5] y PForDelta [33].

Para dar una idea de la compresión que puede alcanzarse en la práctica, el índice invertido comprimido para TREC GOV2 usa entre 6,20 GB (PForDelta) y 7,35 GB (VByte). En promedio esto es entre 8,84 bits y 10,48 bits por docID, menor que los 32 bits originales. Respecto de la velocidad de descompresión de las listas, se obtienen entre 446 millones y 1.010 millones de docIDs por segundo. Una consulta tipo AND puede responderse en promedio en 12 a 14 milisegundos por consulta.

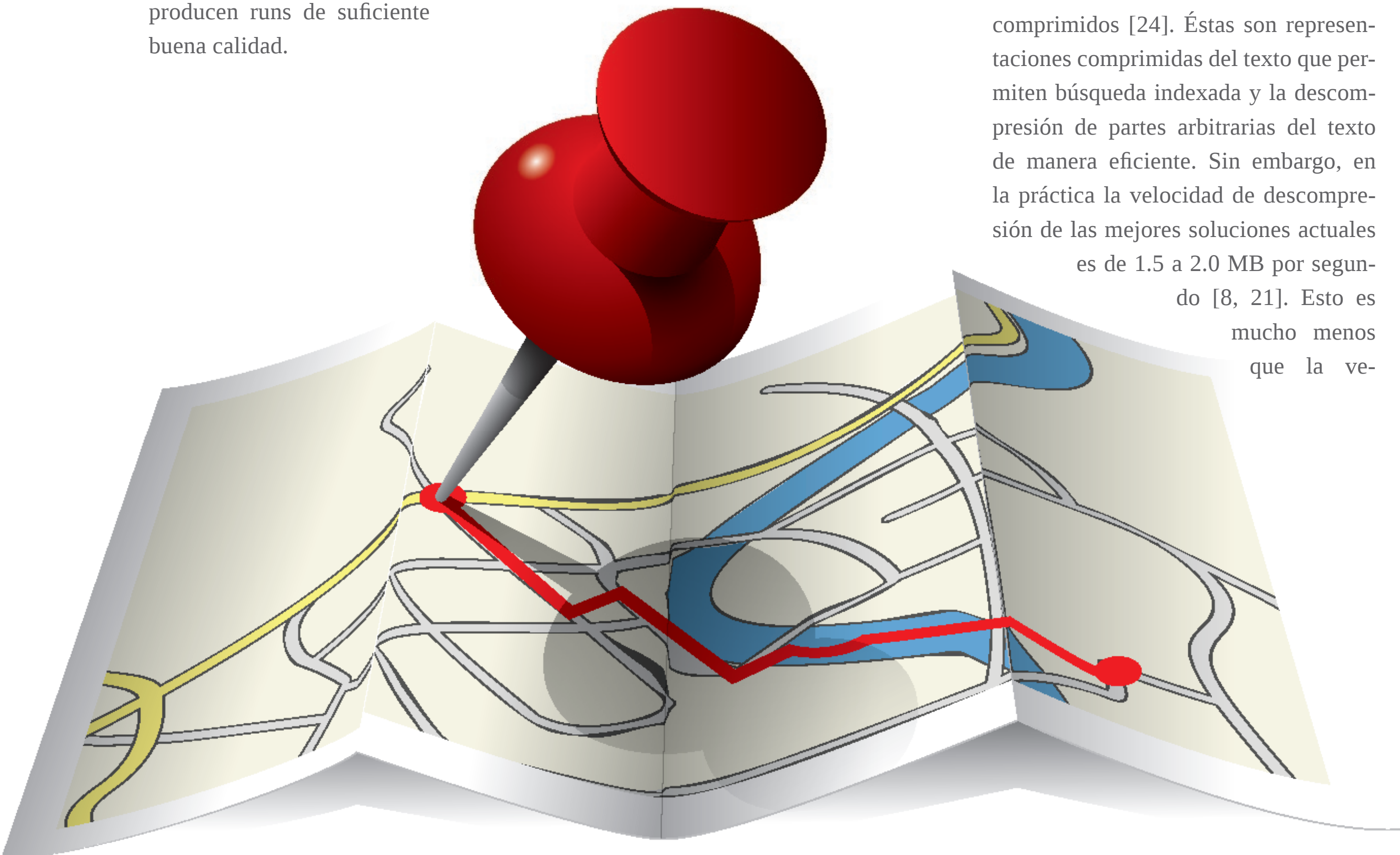
Algunos trabajos recientes [31] muestran cómo optimizar los resultados del párrafo precedente. Esto se logra generando gaps mucho más pequeños en las listas invertidas, mediante la asignación cuidadosa de docIDs a los documentos de la colección [10, 11, 15, 25, 26, 27]. Una de las técnicas más simples y efectivas es la de ordenar las páginas lexicográficamente por sus URLs, y luego asignar los docIDs siguiendo ese orden. De esta manera, las páginas que corresponden a un mismo sitio (y, por tanto, muy probablemente usan un vocabulario similar) van a tener docIDs consecutivos, o en su defecto muy similares. Esto se refleja en las listas invertidas, generando gaps más pequeños comparados con una asignación aleatoria de los docIDs, como la usada en los resultados del párrafo anterior. Para la colección TREC GOV2 enumerada de acuerdo a URLs, el espacio usado por el índice invertido es entre 3,69 GB (5,25 bits por docID) y 6,57 GB (9,35 bits por docID). La velocidad de descompresión es prácticamente la misma que la del párrafo anterior. El tiempo de respuesta para consultas tipo AND es ahora de entre 6 y 7 milisegundos por consulta, reduciendo a la mitad los tiempos del párrafo anterior (docIDs asignados aleatoriamente). Esta mejora se produce porque al asignar los docIDs cuidadosamente, menos bloques de la lista deben ser descomprimidos en tiempo de búsqueda [27, 31]. Nótese que este resultado permite duplicar la capacidad de respuesta, a la vez que permite usar menos espacio.

Dados estos resultados, vale la pena cuestionarse si existe una mejor manera de representar las listas invertidas cuando los docIDs han sido asignados cuidadosamente. Después de todo, los motores de búsqueda actuales trabajan casi exclusivamente en memoria principal [14], lo que hace que este tipo de mejoras sean importantes. Una alternativa puede ser usar métodos de compresión que logran buenas tasas de compresión, como Interpolative Encoding [23]. Sin embargo, este método no es eficiente al descomprimir las listas invertidas, lo que afectaría el tiempo de respuesta. Afortunadamente, hemos podido mostrar que existen mejores representaciones para las listas invertidas en esos casos [7], basándonos en la siguiente observación: el índice invertido para TREC GOV2 ordenada por URLs contiene aproximadamente 60% de gaps con valor 1 (comparado con un 10% de gaps con valor 1 para la colección ordenada aleatoriamente). Pero si el 60% de los gaps tienen valor 1, es probable que estos se agrupen en posiciones consecutivas de las listas invertidas. Llamamos *runs* a esos agrupamientos de 1s consecutivos. Los resultados empíricos en [7] muestran que esto es cierto en la práctica.

En nuestro trabajo [7] mostramos que en estos casos es mejor usar compresión *run-length* de los runs: en lugar de codificar los 1s en un run de mane-

ra explícita (como en los trabajos previos), es más eficiente representarlos implícitamente indicando la presencia del run (con algún tipo de marcador), seguido por la longitud del run. De esta manera se representan runs muy largos usando muy pocos bits. La compresión run-length es muy conocida [19] y ha sido usada con éxito en diversas áreas, como la compresión de imágenes [30]. Sin embargo, no había sido usada en el escenario planteado, aunque una técnica similar ya había sido empleada para comprimir grafos de la Web [12]. Sin embargo, dicha codificación no puede ser empleada directamente en nuestro caso. En cambio, tuvimos que adaptar los principales métodos de compresión de números enteros para soportar compresión tipo run-length. A pesar de que encontrar una asignación de docIDs que produzca la cantidad óptima de runs es un problema NP-hard [20], las enumeraciones cuidadosas de documentos producen runs de suficiente buena calidad.

Usando nuestros compresores adaptados para compresión run-length, el índice invertido para TREC GOV2 ahora requiere entre 3,31 GB (4,71 bits por docID) y 3,65 GB (5,20 bits por docID). Esto es una mejora de un 10% respecto del índice con docIDs asignados por URLs (y un 49% de mejora respecto del índice con documentos enumerados aleatoriamente). Estos métodos descomprimen entre 472 millones y 1.351 millones de docIDs por segundo (una mejora de hasta 34% respecto a los índices ordenados por URL). Esta mejora se explica porque los runs de 1s no se descomprimen de manera explícita, lo cual es útil para muchas aplicaciones. Finalmente, el tiempo de búsqueda es similar (y, en algunos casos, levemente superior) al obtenido para orden por URL, es decir 6 a 7 milisegundos por consulta. Aquí nuevamente el algoritmo que procesa la consulta no descomprime los runs de manera explícita.



EL DESAFÍO DE COMPRIMIR LA WEB TEXTUAL

Para poder generar snippets, los motores de búsqueda deben almacenar en sus servidores el texto de las páginas web. Obviamente, esto usa mucho espacio, por lo que debe comprimirse. Además, para responder a una consulta se deben extraer los diez mejores documentos encontrados y generar sus snippets. El principal problema aquí es cómo comprimir textos de gran tamaño y soportar la descompresión de páginas arbitrarias eficientemente. Conviene recordar que la compresión se basa en detectar regularidades en los datos. Cuando el volumen de datos es muy grande, esto no es una tarea fácil [18].

Un primer enfoque para soportar dicha funcionalidad es el de los auto índices comprimidos [24]. Éstas son representaciones comprimidas del texto que permiten búsqueda indexada y la descompresión de partes arbitrarias del texto de manera eficiente. Sin embargo, en la práctica la velocidad de descompresión de las mejores soluciones actuales es de 1.5 a 2.0 MB por segundo [8, 21]. Esto es mucho menos que la ve-

locidad de descompresión lograda por compresores estándar como snappy o LZ4, que llega a ser de 1.0 a 1.5 GB por segundo [4], o incluso la velocidad de descompresión de gzip que llega a ser de cientos de MBs por segundo. Por esta razón los motores de búsqueda prefieren usar compresores estándar, en particular de la familia Lempel-Ziv de 1977 [32], como todos los mencionados anteriormente. Una solución ineficiente es concatenar las páginas web formando un único texto, y luego usar un compresor estándar. Sin embargo, para extraer una página web debemos descomprimir todo el texto, algo absurdo en el caso de la Web. Otra posible solución es comprimir las páginas por separado. Pero esto afecta notablemente la tasa de compresión, ya que las regularidades entre páginas no son comprimidas.

Una solución bastante aceptada en la literatura [18, 6] corresponde a un punto intermedio entre los enfoques anteriores. Se concatenan las páginas web, pero esta vez formando bloques de un tamaño máximo definido (generalmente entre 50 KB y 1 MB). Luego, cada bloque es comprimido por separado. Para obtener una página web sólo se debe descomprimir el bloque que la contiene, lo cual es eficiente en tiempo de búsqueda. Además, se logra una buena tasa de compresión, porque las regularidades entre las páginas que conforman un bloque son eventualmente detectadas por un compresor. Los resultados de Ferragina y Manzini [18] muestran que la Web textual puede comprimirse a un 5% de su tamaño original usando compresores estándar. Esto fue muy comentado en su momento, ya que es bastante menos que las tasas de compresión de 20% a 25% observadas en documentos convencionales. La razón es que el gran volumen de texto de la Web contiene muchas repeticiones y regularidades que pueden aprovecharse para mejorar la compresión. Sin embargo, lograr esa tasa de compresión tiene su precio: nuestro trabajo [6] mostró que, en el escenario de un motor de búsqueda, extraer las diez primeras páginas de respuesta a una consulta usando dicho enfoque toma 25 milisegundos por consulta. Esto es muy costoso y afecta al tiempo total de respuesta.

Otra solución efectiva es la de Turpin et al. [28], donde proponen enumerar el vocabulario de la colección de acuerdo a la frecuencia con la que las palabras aparecen en las páginas web: la palabra más frecuente recibe un identificador de palabra 0, la segunda palabra más frecuente recibe un identificador 1, y así siguiendo. Luego, las palabras de cada página son reemplazadas por su correspondiente identificador, obteniendo una secuencia de números enteros. Dado que las palabras más frecuentes tienen identificadores con menor valor, se utiliza un método de compresión de números enteros (tal como los usados para comprimir listas invertidas) sobre estas secuencias, obteniendo compresión. Sin embargo, las tasas de compresión alcanzadas son menores a las logradas con compresores estándar: 29% para TREC GOV2. Esto tiene una explicación teórica: este tipo de compresión basada sólo en la frecuencia de las palabras se conoce como compresión de orden 0. Los compresores estándar, por otra parte, alcanzan compresión de orden superior: son capaces de detectar los contextos en que aparecen las palabras, por lo tanto cuentan con mayor información que les permite una mejor compresión. Para una consulta, los 10 mejores resultados pueden descomprimirse en sólo 0.2 milisegundos, lo cual no afecta el tiempo total de una consulta [6]. La practicidad de este método ha hecho que algunos motores de búsqueda lo adopten para almacenar sus colecciones de documentos.

Una manera efectiva de mejorar la tasa de compresión alcanzada con Turpin et al. es usar una compresión de dos fases [6]. Es importante notar que el método de Turpin et al. comprime el texto modificando la manera en que codifica las palabras del mismo. Esto quiere decir que la estructura original del texto (y sus potenciales regularidades) se mantienen y, por ende, es posible aplicar un método de compresión estándar sobre él [17]. La compresión de dos fases entonces significa: en la primera fase se comprime con Turpin et al., mientras que en la segunda fase se aplica un compresor estándar al resultado de la primera fase. En particular, en [6] los mejores compromisos entre tasa de compresión y velocidad de descompresión se lograron usando el compresor snappy en la segunda fase. Dicho compresor es de los más rápidos para descomprimir, pero tiene tasas de compresión del 40% - 50%, lo cual es poco eficiente [4]. Las malas tasas de compresión se deben a que snappy puede detectar regularidades dentro de una ventana muy pequeña del texto (aproximadamente 32 KB). Pero si en la primera etapa el texto se comprime con Turpin et al., esto significa que artificialmente estamos permitiendo que más regularidades quepan dentro de la ventana de compresión. Esto ha mostrado ser muy efectivo, ya que se alcanzan tasas de compresión del 12% con una velocidad de descompresión de 4 milisegundos por consulta, e incluso tasas de compresión del 16% con una velocidad de descompresión de 0.5 milisegundos por consulta [6]. Esto explica en parte cómo un motor de búsqueda puede manejar grandes volúmenes de texto, y a la vez responder consultas y mostrar snippets en pocas décimas de segundo.

CONCLUSIONES Y TRABAJOS FUTUROS

La indexación y la compresión de datos son vitales para el funcionamiento de los motores de búsqueda actuales. Los resultados mostrados permiten comprender (a grandes rasgos) cómo hacen los motores de búsqueda para responder consultas de manera muy rápida sobre grandes volúmenes de texto. Nuestro trabajo apunta a mejorar la eficiencia general de un motor de búsqueda. Nuestros principales resultados muestran mejores compromisos entre espacio usado y tiempo requerido para soportar las búsquedas [6, 7]. En particular, buscamos hacer más eficientes los pasos (2), (3) y (4) de la Figura 1. Un tema de investigación que estamos desarrollando (y que no discutimos aquí) es cómo agrupar los documentos en bloques. Por un lado, quisiéramos que documentos sintácticamente similares pertenezcan al mismo bloque, para comprimirlos mejor. Por otro, quisiéramos que los documentos que van a ser parte de la respuesta a una consulta también estén almacenados en el mismo bloque, para reducir la cantidad de bloques que se descomprimen y mejorar el tiempo de extracción. Respecto del procesamiento de la consulta en la Figura 1, estamos estudiando la manera de usar la representación run-length para mejorar algorítmicamente el proceso. La idea es considerar cada lista invertida como un conjunto de intervalos. Estos conjuntos deben intersectarse (consultas tipo AND) o unirse (consultas tipo OR). Algunos resultados preliminares indican que el tiempo de las consultas tipo OR puede reducirse a casi la mitad del tiempo original. Sin embargo, estamos hablando de consultas OR exhaustivas y sin ranking. Es necesario estudiar cómo agregar ranking sin afectar este resultado.

AGRADECIMIENTOS

Este trabajo es financiado por el Proyecto Fondecyt 11121556 de Iniciación en la Investigación. El grupo de investigación está conformado por Senén González (estudiante de Magíster, DCC, Universidad de Chile), Mauricio Oyarzún (estudiante de Doctorado, Universidad de Santiago de Chile) y Víctor Sepúlveda (Yahoo! Labs Santiago). BITS

Referencias

[1] http://en.wikipedia.org/wiki/Google_Search

[2] http://searchengineland.com/google-search-press-129925

[3] http://www.worldwidewebsize.com/

[4] http://mattmahoney.net/dc/text.html

[5] V. N. Anh y A. Moffat. Inverted index compression using word-aligned binary codes. Information Retrieval, 8(1):151–166, 2005.

[6] D. Arroyuelo, S. González, M. Marín, M. Oyarzún y T. Suel. To index or not to index: Time-space trade-offs in search engines with positional ranking functions. In Proc. ACM SIGIR, pp. 255-264. 2012.

[7] D. Arroyuelo, S. González, M. Oyarzún y V. Sepúlveda. Document Identifier Reassignment and Run-Length-Compressed Inverted Indexes for Improved Search Performance. En proc. ACM SIGIR, páginas 173-182, 2013.

[8] D. Arroyuelo y G. Navarro. Practical approaches to reduce the space requirement of Lempel-Ziv-based compressed text indices. ACM J. of Experimental Algorithmics, Volume 15, article 1.5, 2010.

[9] R. Baeza-Yates y B. Ribeiro-Neto. modern information retrieval: The concepts and technology behind search, Second edition. Pearson Education Ltd. 2011.

[10] R. Blanco y A. Barreiro. Document identifier reassignment through dimensionality reduction. In Proc. ECIR, pp. 375–387, 2005.

[11] D. Blandford y G. Blelloch. Index compression through document reordering. In Proc. DCC, pp. 342–351, 2002.

[12] P. Boldi y S. Vigna: The webgraph framework I: compression techniques. In Proc. WWW, pp. 595-602, 2004.

[13] S. Büttcher, C. Clarke y G. Cormack. Information Retrieval: Implementing and evaluating search engines. The MIT Press. 2010.

[14] J. Dean. Challenges in building large-scale information retrieval systems: invited talk. In Proc. WSDM, pp. 1, 2009.

[15] S. Ding, J. Attenberg y T. Suel. Scalable techniques for document identifier assignment in inverted indexes. In Proc. WWW, pp. 311–320, 2010.

[16] P. Elias. Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory, 21(2):194–203, 1975.

[17] A. Fariña, G. Navarro y J. Paramá. Boosting text compression with word-based statistical encoding. The Computer Journal, 55(1):111-131, 2012.

[18] P. Ferragina y G. Manzini. On compressing the textual web. In Proc. WSDM, pp. 391–400, 2010.

[19] S. Golomb. Run-length encoding. IEEE Transactions on Information Theory, 12(3):399–401, 1966.

[20] D. Johnson, S. Krishnan, J. Chhugani, S. Kumar y S. Venkatasubramanian. Compressing large boolean matrices using reordering techniques. In Proc. VLDB, pp. 13–23, 2004.

[21] S. Kreft y G. Navarro. On Compressing and indexing repetitive sequences. Theoretical Computer Science 483:115-133, 2013.

[22] C. Manning, P. Raghavan y H. Schütze. Introduction to information retrieval, Cambridge University Press. 2008.

[23] A. Moffat y L. Stuiver. Binary interpolative coding for effective index compression. Information Retrieval, 3(1):25–47, 2000.

[24] G. Navarro y V. Mäkinen: Compressed full-text indexes. ACM Computing Surveys, 39(1), 2007.

[25] W.-Y. Shieh, T.-F. Chen, J. Shann, y C.-P. Chung. Inverted file compression through document identifier reassignment. Information Processing and Management, 39(1):117–131, 2003.

[26] F. Silvestri. Sorting out the document identifier assignment problem. In Proc. ECIR, pp. 101–112, 2007.

[27] F. Silvestri, S. Orlando y R. Perego. Assigning identifiers to documents to enhance the clustering property of full-text indexes. In Proc. ACM SIGIR, pp. 305–312, 2004.

[28] A. Turpin, Y. Tsegay, D. Hawking, H. Williams: Fast generation of result snippets in web search. In Proc. SIGIR, pp. 127–134, 2007.

[29] H. Williams y J. Zobel. Compressing integers for fast file access. The Computer Journal, 42(3):193–201, 1999.

[30] I. Witten, A. Moffat y T. Bell. Managing gigabytes: Compressing and indexing documents and images, Second Edition. Morgan Kaufmann, 1999.

[31] H. Yan, S. Ding y T. Suel. Inverted index compression and query processing with optimized document ordering. In Proc. WWW, pp. 401–410, 2009.

[32] J. Ziv y A. Lempel: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory, 23(3): 337-343, 1977.

[33] M. Zukowski, S. Héman, N. Nes y P. Boncz. Super-scalar RAM-CPU cache compression. In Proc. ICDE, pp. 59, 2006.