Multicompresión de Grandes Listas de Enteros para Sistemas de Búsquedas

Agustín Gonzalez[1] and Gabriel Tolosa[1,2]
agustin.ang.92@gmail.com tolosoft@unlu.edu.ar

[1] Departamento de Ciencias Básicas, Universidad Nacional de Luján, Argentina
[2] CIDETIC, Universidad Nacional de Luján, Argentina

Resumen La búsqueda en grandes repositorios de documentos (como la web) exige que los sistemas se ejecuten bajo estrictas restricciones de performance. En la actualidad, dada la cantidad de documentos que un sistema gestiona, resulta indispensable aplicar técnicas tales como la compresión de las estructuras de datos. Particularmente, aquí se aborda el problema de la compresión de un índice invertido mediante un esquema "multicompresión"que procesa diferentes porciones de una lista utilizando diversos *codecs*. Los resultados preliminares muestran que es posible compensar el *overhead* requerido para mantener este esquema, mientras que se mejora el tiempo de descompresión.

Keywords: Índice Invertido · Compresión · Codec.

1. Introducción

Los sistemas de búsqueda de escala web se ejecutan bajo estrictas restricciones de performance que les exigen mantener estructuras de datos y algoritmos extremadamente eficientes. Esto se requiere para satisfacer la alta demanda de consultas por parte de los usuarios, mientras deben poder gestionar la creciente cantidad de documentos que indexan[3]. Además, la disponibilidad de grandes espacios de almacenamiento (por ejemplo, grandes *datacenters*) sugiere que la cantidad de datos online continuará creciendo.

Una de las técnicas ampliamente usadas para manejar grandes volúmenes de datos es la compresión que, en los últimos años, ha dado origen a diversas familias de algoritmos (*codecs*). En el ámbito de los sistemas de búsqueda, las técnicas de compresión son específicas para los datos que se almacenan. Para soportar las búsquedas sobre los documentos, la mayoría de los sistemas utiliza un índice invertido como estructura de datos [13]. De forma simple, el índice se encuentra dividido en un vocabulario (V) que contiene la lista de términos existentes (más información de gestión) y un conjunto de *posting lists*, P. Cada término $t_i \in V$ tiene asociado una lista $p_i \in P$ compuesta por pares de enteros (d_j, tf_{ij}) que identifican a los documentos donde aparece dicho término junto

[3] Ej.: índice de Google, 55.000 millones de docs. https://www.worldwidewebsize.com/

a información de su frecuencia, usada para el ranking[4]. Dado que V es muy pequeño respecto de P, resulta indispensable comprimir las posting lists y, para ello, se requieren de métodos eficientes sobre listas de números enteros.

Los beneficios de un índice eficientemente comprimido son directos [2]: por un lado, menos almacenamiento y, por el otro, mayor velocidad de transferencia entre memoria secundaria y principal; lo que también genera una mayor performance en el recorrido de las listas durante una búsqueda. Además, gestionar un espacio comprimido incrementa la posibilidad de que *ciertos* datos se encuentren en cache, evitando el acceso a disco. El *costo* de mantener un índice comprimido es, luego, el tiempo de descompresión. No obstante, la disponibilidad de CPUs cada vez más potentes minimiza este inconveniente.

En este trabajo se aborda el problema de la compresión de un índice invertido utilizando un enfoque híbrido en el cual una lista es dividida en bloques que se comprimen con diferentes codecs. Esto requiere, por un lado, considerar que se incurre en cierto *overhead*, ya que resulta necesario almacenar información referida al codec particular en cada caso; y, por el otro lado, el algoritmo de descompresión debe gestionar eficientemente esta situación para mantener (o mejorar) el tiempo de descompresión.

El objetivo principal es presentar un esquema híbrido, denominado "multicompresión", que aborda el problema de la codificación de posting lists, utilizando diferentes codecs en cada una de sus particiones. A tales efectos, primero se revisa la eficiencia de diferentes codecs, variando también el tamaño de bloque, y mostrando que es interesante tener en cuenta este parámetro. El resto del trabajo se organiza de la siguiente manera: en la Sección 2 se describe la propuesta. Luego, se presentan los experimentos y resultados preliminares (Sección 3), mientras que la Sección 4 ofrece las ideas para la discusión.

2. Multicompresión de Listas

El esquema multicompresión propuesto, cuya versión aquí presentada ha sido denominada mc@0, divide cada posting list en bloques de tamaño fijo, y luego codifica cada partición del bloque de documentos y de frecuencias seleccionando la codificación que genere la menor cantidad de bytes. Más específicamente, de un conjunto de codecs considerados en el estado del arte, mc@0 utiliza *Interpolative* [7], *OptPFD* [11], *QMX* [10], *Simple16* [12], *Simple8b* [1], *MaskedVByte* [8], *StreamVByte* [5] y *VarintGB* [3]. Además, para aquellos bloques cuyo tamaño es divisible por 128, el esquema contempla a *SIMD-BP128* [4], un codec eficiente en tiempo de descompresión. Por otro lado, en caso de que un bloque esté compuesto por únicamente valores 1s, el esquema considera su cardinalidad lo suficientemente expresiva como para su reconstrucción, lo cual tiene un costo de almacenamiento de 0 bytes. Asimismo, si un bloque consta de al menos un 25 % de 1s, se contempla el uso de *many-ones*, un codec propuesto para este esquema

[4] Habitualmente, en un índice sin comprimir se utilizan 4 bytes para el identificador de documentos (docID) y otros 4 para la frecuencia

en particular que se vale de Simple16 para codificar aquellos valores mayores a 1, junto a los *gaps* de las posiciones asociadas.

La información de los codecs utilizados en una partición documento-frecuencia se almacena en 1 byte, del cual se utilizan 4 bits para los documentos, y los restantes 4 para las frecuencias. La excepción de lo aquí descripto son las listas de longitud 1, cuyo costo de almacenamiento es de 0 bytes para el caso del bloque de documentos (ya que este valor puede tomarse directamente desde la suma de identificadores, que es parte del overhead); y de como máximo 5 bytes para las frecuencias (ya que en este caso, siempre se utiliza *VByte* [8]). Por su parte, el proceso de decodificación se lleva a cabo empleando un selector eficiente de descompresores para evitar, en lo posible, el *branching* [14].

El nuevo esquema se incorporó como una rama de desarrollo propia del motor de búsquedas PISA (Performant Indexes and Search for Academia) [6], el cual está optimizado para ser eficiente tanto en compresión como en recuperación.

3. Experimentos y Resultados Preliminares

Los experimentos se llevan a cabo sobre el motor de búsquedas PISA y como conjunto de datos de prueba, se usa la colección Clueweb12b[5], ampliamente utilizada en la evaluación de técnicas de indexación y recuperación. La misma está formada por 52.343.021 documentos, 133.248.235 términos y un total de 14.173.094.439 postings en el índice. Previo a la compresión, la colección se reordena por la URL [9] de los documentos, ya que está mostrado que el reordenamiento de los docIDs produce índices más compactos, incrementando la velocidad de descompresión.

En el primer experimento, se explora la compresión del dataset utilizando diferentes tamaños de bloque. La literatura del área propone el uso de bloques de 128 enteros de 4 bytes cada uno, lo que genera unidades de transferencia con el dispositivo de almacenamiento de 512 o 1024 bytes si se consideran solo docIDs o docIDs+Frecuencias, respectivamente. Sin embargo, para el esquema de multicompresión propuesto, una de las posibilidades es variar este tamaño, por lo que aquí se comprime la colección completa utilizando bloques de 64, 128 y 256 elementos. Los resultados se presentan en la Tabla 1. La primera observación interesante es que, a diferencia de lo comúnmente establecido, bajo la configuración propuesta, los bloques de 256 elementos permiten reducir el tamaño final del índice en todos los casos analizados; hasta en un 6 % para OptPFD donde, además, se evidencia la mayor diferencia (10 %) entre bloques de 64 y 128 elementos. Abusando de la notación, se puede establecer el orden y las diferencias como: $64 >_{[3-10\,\%]} 128 >_{[1-6\,\%]} 256$.

Comparando el esquema multicompresión respecto del codec que mejor prestaciones ofrece en cuanto a compresión (Interpolative) se obtiene que mc@0 reduce el tamaño final del índice en 0.26 % y 0.55 % para 128 y 256 ítems por bloque, respectivamente (aunque lo incrementa en 0,35 % para bloques de 64). Un

[5] https://lemurproject.org/clueweb12/

	Tamaño (GB)			Diferencias (%)	
Codec	**64**	**128**	**256**	**128/64**	**128/256**
Interpolative	12,12	11,36	10,99	0,94	1,03
OptPFD	15,60	13,97	13,21	0,90	1,06
Simple16	17,38	16,19	15,62	0,93	1,04
Simple8b	20,51	18,88	18,14	0,92	1,04
QMX	22,31	20,37	19,37	0,91	1,05
MaskedVByte	30,24	29,33	28,82	0,97	1,02
VarintGB	36,26	35,32	34,77	0,97	1,02
StreamVByte	38,13	36,45	36,05	0,96	1,01
mc@0	12,16	**11,33**	**10,93**	0,93	1,04

Tabla 1. Tamaños finales del índice (y diferencias) utilizando diferentes codecs de compresión y variando el tamaño de bloque. La última línea corresponde al esquema multicompresión propuesto.

análisis más detallado muestra que, usando bloques de 128 enteros, el esquema multicompresión permite reducir hasta un 3.36 % la información de frecuencias (y un 0.36 % los docIDs). Como resultado global, se puede considerar que ambas performances son equivalentes, aunque existe una variación en cuanto a los tamaños de bloque, por lo que se considera un factor importante a tener en cuenta. Por otro lado, las prestaciones del modelo multicompresión se ven afectadas por la información de *overhead* que hay que introducir en los bloques que determinan el codec usado para comprimirlo.

En el mismo sentido, teniendo en cuenta los tamaños finales de los índices, se analiza el tiempo total de descompresión para los mejores codecs en cuanto a espacio (Interpolative y mc@0) al utilizar bloques de 128 y 256. La Tabla 2 permite visualizar que el método basado en multicompresión mejora sobre Interpolative casi un 9 % y 5 % para particiones de 128 y 256 enteros, respectivamente. Finalmente, la Figura 1 muestra la ubicación del método propuesto en cuanto al *tradeoff* entre espacio de almacenamiento y tiempo de descompresión.

	Bloque	
Codec	**128**	**256**
Interpolative	6,57	6,51
mc@0	5,98	6,17
Diferencia (%)	8,98	5,22

Tabla 2. Tiempo total de descompresión (m). mc@0 vs Interpolative.

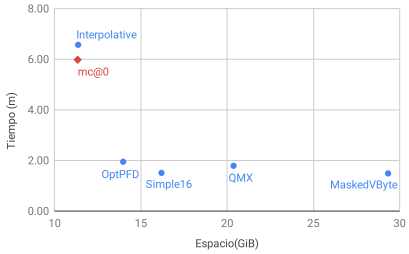

Figura 1. *Tradeoff* espacio/tiempo. mc@0 (diamante rojo) mejora sobre Interpolative el tiempo de decompresión.

4. Discusión

En este artículo se presenta el avance de un modelo de multicompresión para grandes listas de números enteros como las que conforman los índices invertidos (docIDs y frecuencias) de los sistemas de búsqueda de gran escala. Para ello, inicialmente se revisa el tamaño de partición a utilizar, evidenciando que bloques de 256 elementos pueden ser competitivos más allá del valor estándar habitualmente utilizado (128). Por otro lado, se muestra que el esquema de multicompresión

con el que se está trabajando puede mejorar el espacio ocupado (considerando el *overhead* necesario para soportar múltiples codecs) mientras que resulta más eficiente (hasta 9 %) para la descompresión del índice.

Actualmente, se está trabajando en un esquema de tamaño de bloque variable y en un algoritmo que estima el *mejor* codec a utilizar en función de características propias de las secuencias de enteros que conforman las listas.

Referencias

1. Anh, V., Moffat, A.: Index compression using 64-bit words. Softw. Pract. Exper. **40**(2), 131–147 (Feb 2010)
2. Catena, M., Macdonald, C., Ounis, I.: On inverted index compression for search engine efficiency. In: Rijke, M., Kenter, T., de Vries, A., Zhai, C., de Jong, F., Radinsky, K., Hofmann, K. (eds.) Advances in Information Retrieval. pp. 359–371. Springer International Publishing, Cham (2014)
3. Dean, J.: Challenges in building large-scale information retrieval systems: Invited talk. In: Proceedings of the Second ACM International Conference on Web Search and Data Mining. p. 1. WSDM '09, Association for Computing Machinery, New York, NY, USA (2009)
4. Lemire, D., Boytsov, L.: Decoding billions of integers per second through vectorization. Softw. Pract. Exper. **45**(1), 1–29 (Jan 2015)
5. Lemire, D., Kurz, N., Rupp, C.: Stream vbyte: Faster byte-oriented integer compression. Information Processing Letters **130** (09 2017)
6. Mallia, A., Siedlaczek, M., MacKenzie, J., Suel, T.: PISA: Performant indexes and search for academia. In: CEUR Workshop Proceedings. vol. 2409, pp. 50–56 (2019)
7. Moffat, A., Stuiver, L.: Binary interpolative coding for effective index compression. Inf. Retr. **3**(1), 25–47 (Jul 2000)
8. Plaisance, J., Kurz, N., Lemire, D.: Vectorized vbyte decoding. CoRR **abs/1503.07387** (2015)
9. Silvestri, F.: Sorting out the document identifier assignment problem. In: Amati, G., Carpineto, C., Romano, G. (eds.) Advances in Information Retrieval. pp. 101–112. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
10. Trotman, A.: Compression, simd, and postings lists. In: Proceedings of the 2014 Australasian Document Computing Symposium. p. 50–57. ADCS '14, Association for Computing Machinery, New York, NY, USA (2014)
11. Yan, H., Ding, S., Suel, T.: Inverted index compression and query processing with optimized document ordering. In: Proceedings of the 18th International Conference on World Wide Web. p. 401–410. WWW '09, Association for Computing Machinery, New York, NY, USA (2009)
12. Zhang, J., Long, X., Suel, T.: Performance of compressed inverted list caching in search engines. In: Proceedings of the 17th International Conference on World Wide Web. p. 387–396. WWW '08, Association for Computing Machinery, New York, NY, USA (2008)
13. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Computing Surveys **38**(2), 6—-62 (2006)
14. Zukowski, M., Heman, S., Nes, N., Boncz, P.: Super-scalar ram-cpu cache compression. In: Proceedings of the 22nd International Conference on Data Engineering. p. 59. ICDE '06, IEEE Computer Society, USA (2006)