Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA (SG)

Grado en Ingeniería Informática de Servicios y Aplicaciones

Diseño e Implementación de Diccionarios Comprimidos de Texto para Vocabularios Ordenados

Alumno: Álvaro Alonso Isla

Tutor: Miguel Ángel Martínez Prieto

AGRADECIMIENTOS

Este trabajo es un proyecto dedicado a iniciarme en la investigación, y no habría sido posible ni siquiera la idea de desarrollarlo sin la ayuda y constante apoyo de mi tutor Miguel Ángel, que ha tenido la paciencia de aguantarme todas las semanas reuniéndose conmigo para resolver las innumerables dudas que tenía. Él es el responsable de mi interés en los datos, y más concretamente en su compresión, interés que me ha empujado a decidirme a realizar este trabajo.

También agradecerle a mi tutor el haberme facilitado la librería con las estructuras de datos y los diccionarios que he estudiado en mi fase de investigación, así como sus apuntes y artículos relacionados con el tema, sin los cuales habría sido mucho más complicado entender las bases necesarias.

Por otro lado, a pesar de no conocerle personalmente, quiero agradecer que me dejara desarrollar su idea a Gonzalo Navarro, profesor de Chile y uno de los mayores expertos en este campo.

Y por último no puedo olvidarme de mi familia y amigos, que me han apoyado y animado a seguir adelante, incluyendo a mi novia Ainoa, que ha tenido que escucharme hablar sobre temas que ni siquiera conoce y siempre ha intentado entenderlos y ayudarme en todo lo que podía.

Resumen: El proyecto actual aborda el problema de utilizar diccionarios comprimidos de texto para organizar vocabularios de strings con un ranking previamente establecido. Esta línea de investigación plantea un reto de interés dado el uso de este tipo de vocabularios en aplicaciones como los buscadores web. Particularmente, nuestra operación de *top-K* (no soportada por ningún otro diccionario en el estado del arte) resulta de interés para implementar operaciones de autocompletado y obtención de sugerencias en las búsquedas. En este trabajo hemos desarrollado dos diccionarios comprimidos de texto que abordan la problemática de gestionar vocabularios con ranking preestablecido, el primero de los cuales (referido como RankedDictionary-Simple) plantea una extensión simple de algunos de los diccionarios ya existentes. Por otro parte, la segunda de nuestras propuestas (referida como RankedDictionary-Advanced) plantea una nueva aproximación para la compresión de los vocabularios tratados en este proyecto y, además, ofrece la posibilidad de resolver la operación de búsqueda *top-K* de una manera eficiente. Ambas propuestas se han evaluado utilizando vocabularios de referencia en el estado del arte y se han comparado contra otros diccionarios, pudiendo concluir las ventajas y desventajas de nuestras propuestas con respecto a las ya existentes.

Palabras clave: Diccionarios comprimidos ordenados, estructuras de datos compactas, top-K, Re-Pair.

Abstract: The current project tackles the problem of using compressed string dictionaries to organize string vocabularies with a previously stablished ranking. This research line sets out a challenge because of the use of this type of vocabularies in applications such as web browsers. Partycularly, our operation top-K (not supported by any other existing dictionary) is interesting for implement autocomplete operations and suggestion extracting in searches. In this project, we have developed two compressed string dictionaries which try to solve the problem of managing ranked string vocabularies. The first one (called RankedDictionary-Simple) sets out a simple extension of some of the existing dictionaries. And the other proposal (called RankedDictionary-Advanced) lays out a new way for the compression of the dealt vocabularies in this project; in addition, it offers the possibility of resolving the *top-K* search operation efficiently. Both proposals have been evaluated using reference vocabularies and have been compare with other dictionaries, obtaining the advantages and disadvantages of our proposals in relation to the existing ones.

Keywords: Orderd compressed dictionaries, compact data structures, top-K, Re-Pair.

Índice

CAPÍTULO 1
INTRODUCCIÓN

Desde la aparición de los ordenadores se ha iniciado una fase de transición de los datos en formato físico (principalmente en papel) a formato digital, pasando a ser ficheros de un ordenador. En un principio estos datos se han almacenado de tal manera que representaran símbolos entendibles y legibles para los humanos, pero con el paso del tiempo la cantidad de datos ha ido creciendo demasiado y este tipo de almacenamiento deja de ser adecuado.

Este problema ha motivado a la creación de técnicas de compactación que permitan el manejo de información reduciendo el espacio necesario para almacenar dicha información. En el presente proyecto se diseñan e implementan algunas que no han sido probadas hasta el momento.

1.1. Motivación

Como ya se ha mencionado, desde hace unos años la cantidad de datos que se almacenan es cada vez mayor y crece cada día a mayor velocidad, en gran medida gracias a la posibilidad de que cualquier usuario pueda generar contenido. Todo esto ha dado lugar a la problemática del *Big Data*.

Nos referimos a *Big Data* como "cualquier colección de datos que excede la capacidad de cómputo de un sistema gestor de bases de datos tradicional" [1]. Estas colecciones de *Big Data* tienen tres características particulares (las tres V de *Big Data*):

- **Volumen**: se refiere a la enorme cantidad de datos que se almacenan en las colecciones. Es su característica más obvia, ya que hoy en día la generación de datos es enorme; por ejemplo, cada minuto se crean más de 300.000 tweets, y se suben más de 100 horas de video a Youtube [2]. Debido a este crecimiento de los datos, la escalabilidad se convierte en el principal objetivo relacionado con el volumen del Big Data; y para conseguir esta escalabilidad se necesitan mecanismos de almacenamiento efectivos. Sin embargo, hay que tener en cuenta en estos mecanismos que las decisiones en el almacenamiento afectan a la recuperación de datos (velocidad), y los usuarios quieren que la recuperación de los datos sea lo más rápida posible.
- **Velocidad**: se refiere a la frecuencia de generación y solicitud de los datos, que se producen generalmente en entornos distribuidos, ya que tanto los datos nuevos como los resultados de las solicitudes se transmiten por la red. Sin embargo, los recursos de red no crecen en la misma proporción que lo hacen los datos generados. Debido a esto se necesitan técnicas para mejorar la velocidad de transmisión de datos. Para la generación de datos el objetivo es conseguir el procesamiento de datos en *streaming* (en tiempo real); mientras que para las consultas el almacenar los datos en memoria principal ayudaría a conseguir esa velocidad deseada.

- **Variedad**: esta característica se refiere a la diversidad en la estructura que se puede encontrar en colecciones de Big Data; y es que más que diversidad en la estructura se puede hablar de ausencia de la misma, ya que alrededor del 80% de los datos que existen se definen como no estructurados, además de que su tasa de crecimiento es mucho mayor a la que presentan los datos estructurados, alrededor de 15 veces más rápido [3]. La existencia de esta enorme cantidad de datos sin estructurar hace necesario el uso de modelos con los que se puedan organizar datos de diversos tipos en una única representación, independiente de la estructura que tengan dichos datos.

A la hora de trabajar con Big Data se debe lidiar con las tres características nombradas anteriormente. El problema de la variedad es un problema a nivel lógico, pero tanto el volumen como la velocidad requieren de optimizaciones a nivel físico. En este proyecto estudiaremos el uso de la compresión de texto para afrontar los problemas de volumen y velocidad. Más concretamente, diseñaremos e implementaremos diferentes diccionarios de texto comprimidos. Como se explicará a continuación, esta estructura de datos compacta se utiliza en numerosos escenarios de interés.

1.2. Objetivos

Una vez se conoce la necesidad de compactar los datos pasamos a explicar los objetivos que se pretenden alcanzar con el desarrollo del proyecto. Para ello es necesario introducir lo que es un diccionario de texto, aunque en el capítulo 3 se explica con mayor detalle. Un diccionario de texto es una estructura de datos que organiza una serie de strings asignando a cada uno un identificador único.

El primer objetivo es añadir un valor que no tienen la mayoría de diccionarios existentes, y es la posibilidad de **mantener un orden o ranking** determinado por la aplicación a consumir. De esta manera los resultados de búsquedas en los diccionarios se podrán devolver en el orden deseado y no por orden alfabético. Y aunque esto se puede conseguir con alguna implementación de diccionarios ya existentes, la que se desarrolla en este proyecto permite también obtener un buen grado de compactación y permitir operaciones de búsqueda de prefijos (útiles en contextos de interés como la búsqueda web).

El segundo de los objetivos es conseguir el **manejo de grandes diccionarios en memoria principal** (llegando a poder trabajar incluso con *Big Data*). Esto se pretende conseguir a través de la compactación, por lo que el grado de compresión debe de ser elevado.

Finalmente se pretende una mejora en el **rendimiento** respecto a diccionarios que trabajen en memoria secundaria, para lo cual se debe conseguir que las operaciones necesarias para compactar aprovechen los ciclos de procesador que quedan libres durante el tiempo que tarda en moverse un bloque de datos desde el disco a la memoria principal.

1.3. Estructuración de la memoria y CD

Esta memoria comienza con una primera parte de introducción donde se motiva el proyecto, explicando la problemática existente hoy en día con los grandes volúmenes de datos y el conocido *Big Data*; una vez motivado el proyecto se indican los objetivos que se pretenden conseguir con el desarrollo del proyecto; y para concluir con la introducción se lleva a cabo una planificación y presupuesto del proyecto, seguida de los tiempos y costes reales que se han obtenido al finalizar.

Después de la introducción se dedica un capítulo completo a la compresión y a conceptos básicos necesarios para entender lo que son los diccionarios de texto, y comprender aquellos que se van a utilizar y desarrollar, donde se explica lo que es la compresión de texto y las técnicas de compresión de texto que se necesitarán; además se introducen también las estructuras compactas de datos y se explican algunas de ellas.

Una vez introducida toda la teoría sobre la compresión de datos, se explican los diccionarios de texto, donde se explica lo que son y se profundiza en el funcionamiento de algunos de los diccionarios comprimidos más relevantes para el proyecto, ya que varios de ellos serán la base para los diccionarios que se van a desarrollar.

Además acompañando a la memoria se incluye un CD, que contiene una copia de la memoria en formato pdf y un directorio "TFG_src" con el código fuente. Dentro del código fuente se debe saber que la mayoría de ficheros no han sido creados en este proyecto, sino que han sido proporcionados por M. A. Martínez-Prieto, autor de muchos de ellos. Dentro de este directorio se encuentran los ficheros de creación y pruebas de los diccionarios, y un directorio "src" que contiene todos los diccionarios; además está el directorio libcds que incluye todas las estructuras compactas de datos que se explican en esta memoria.

Los diccionarios que se han programado en este proyecto se han incluido junto con los ya existentes previamente, creando una nueva clase "RankedStringDictionary" de la que heredan todos los diccionarios nuevos que se han creado. También se han creado nuevas clases auxiliares, como iteradores específicos para los nuevos diccionarios. Por último se ha modificado la clase "WaveletTreeNoPtrs" perteneciente a la librería libcds.

1.4. Planificación y presupuesto

En este apartado se va a analizar el planteamiento inicial realizado antes de comenzar con el proyecto, así como el coste que tendría llevarlo a cabo. Posteriormente se mostrará la planificación real que se ha llevado a cabo, con su correspondiente coste, y se comparará con lo planeado en el inicio.

1.4.1. Planificación

En la planificación del proyecto se distinguen tanto las distintas tareas a realizar como los profesionales que las llevan a cabo. Se pueden distinguir tres tareas distintas: una primera fase de **investigación** y estudio del estado del arte y de las técnicas existentes, una fase de **análisis** por parte de un experto donde se plantea el funcionamiento y la forma de los diccionarios a desarrollar, y una última fase de **programación** y pruebas de los diccionarios; el analista experto también tendrá encargada una parte de **seguimiento** del

proyecto, comprobando que se implementa de forma correcta lo planificado. Tanto la investigación como la programación deben ser llevadas a cabo por la misma persona aunque sean tareas distintas, dada la dificultad de programar técnicas que no se conocen; mientras que la tarea de análisis será llevada a cabo por un experto en la materia que no necesita estudio previo.

- **Investigación**: Esta tarea se estima que se realizará durante 8 semanas con 5 horas de trabajo diario, no incluyendo los domingos. Será llevada a cabo por un ingeniero informático junior.
- **Análisis**: Esta tarea se estima de corta duración, pues se estima que en una semana con 4 horas de trabajo diarias se termine (sin incluir el domingo). Será llevada a cabo por el experto.
- **Programación**: También llevada a cabo por el informático junior, pero al ser una tarea distinta no se le aplica el mismo precio por hora ni las mismas horas de trabajo diarias. Se programa un periodo de 14 semanas a 4 horas diarias excluyendo domingos.
- **Seguimiento**: Tarea realizada por el experto para comprobar que los diccionarios se implementan según lo programado en la fase de análisis. Se llevará a cabo de todo el proceso de programación con un tiempo de 4 horas semanales.

Presupuesto

Este proyecto consiste en un proceso de investigación inicial seguido por la parte de implementación de nuevos diccionarios, por lo que no es un proyecto de desarrollo de software puro. Debido a esto no se va a realizar un presupuesto basado en líneas de código o centrada en la programación, sino un coste por horas trabajadas dividiendo el trabajo a realizar. El presupuesto realizado se divide en tres partes distintas: profesionales, hardware y software.

En cuanto a los **profesionales** basta con dirigirse a la parte de planificación y calcular el número de horas de cada profesional, sabiendo el salario neto por hora de trabajo de cada profesional. De esta manera salario de los profesionales sería el mostrado a continuación en la tabla 1.

Tarea	Horas totales	€/hora	Salario neto
Investigación	8*6*5 = 240 horas	10 €/h	240*10 = 2400€
Análisis	6*4 = 24 horas	30 €/h	24*30 = 720€
Programación	14*6*4 = 336 horas	15 €/h	336*15 = 5040€
Seguimiento	14*4 = 56 horas	25 €/h	56*25 = 1400€

Tabla 1 – Presupuesto s. neto profesionales

Sin embargo, hay que tener en cuenta la necesidad del pago de IRPF, de la contribución a la seguridad social (tanto del trabajador como de la empresa), que incluye las contingencias comunes, el desempleo y la formación profesional. Para obtener el salario bruto a partir del salario neto, el IRPF y la contribución seguridad social del trabajador (CSST) se tiene en cuenta la siguiente fórmula:

$$s.\ bruto * (1\text{-}IRPF\text{-}CSST) = s.\ neto$$

Siendo IRPF y CSST en tanto por ciento. Y despejando de esa fórmula se tiene que:

s. neto / (1-IRPF-CSST) = *s. bruto*

De esta manera se puede calcular el presupuesto total debido a los trabajadores. Este presupuesto se muestra en la tabla 2, donde se indica el salario neto, el IRPF, la contribución a la seguridad social del trabajador (CSST), el salario bruto, la contribución a la seguridad social de la de la empresa (CSSE), y el gasto total debido al trabajador. Para obtener el IRPF y las contribuciones a la seguridad social se ha acudido a las páginas oficiales [4] [5]. La contribución a la seguridad social por parte del trabajador (CSST) es de 4'70% (contingencias comunes) + 1'60% (desempleo) + 0'10% (formación profesional) = **6'4%**. Del mismo modo se tiene que la CSSE = 23'60% (contingencias comunes) + 6'70% (desempleo) + 0'60% (formación profesional) = **30'9%**

Tarea	S. neto	IRPF	CSST	S. bruto	CSSE	Coste por profesionales
Investigación	2400€	19%	6'4%	2400/(1-0'19-0'064) = 3216'16€	30'9%	3216'16*1'309 = 4209'95€
Análisis	720€	19%	6'4%	720/(1-0'19-0'064) = 965'15€	30'9%	965'15*1'309 = 1263'38€
Programación	5040€	19%	6'4%	5040/(1-0'19-0'064) = 6756'04€	30'9%	6756'04*1'309 = 8843'66€
Seguimiento	1400€	19%	6'4%	1400/(1-0'19-0'064) = 1876'68€	30'9%	1876'68*1'309 = 2456'57€

Tabla 2 – Presupuesto total profesionales

Esto hace un total de 4209'95 + 1263'38 + 8843'66 + 2456'57 = 16.773'56€ en profesionales.

En cuanto al **hardware** necesario, se muestra en la tabla 3 los materiales necesarios así como su tiempo estimado de vida, su uso total, su porcentaje de uso, su precio total y el precio por el uso dado en el proyecto.

Hardware	Tiempo de vida	Precio del hardware	Utilización	Porcentaje de uso	Coste por uso en el proyecto
Ordenador portátil MSI GE62 (1)	60 meses	1300€	6 meses	10%	1300*0'1= 130€
Ratón USB Logitech (1)	24 meses	20€	6 meses	25%	20*0'25 = 5€
Pendrive 16GB (2) Toshiba	48 meses	15€ * 2 = 30€	6 meses	12'5%	30*0'125 = 3'75€

Tabla 3 – Presupuesto hardware

Por lo que el precio por utilización de hardware es de 130+5+3'75 = ¿?€.

En la utilización de **software** se presupuesta de la misma forma que en el hardware, según se muestra en la tabla 4.

Software	Tiempo vida software	Precio del software	Utilización	Porcentaje de uso	Coste por uso en el proyecto
Microsoft Office 2013	60 meses	80€	6 meses	10%	80*0'1= 8€
Windows 10	30 meses	gratuito	6 meses	20%	gratuito
Ubuntu 14.04	30 meses	gratuito	6 meses	20%	gratuito
Codeblocks	60 meses	gratuito	6 meses	10%	gratuito
Valgrind debugger	60 meses	gratuito	6 meses	10%	gratuito

Tabla 4 – Presupuesto software

Por lo que el precio del software es de 8€. Un precio muy bajo al utilizar mayormente productos gratuitos, ya que sólo Microsoft Office es de pago, y al ser una versión antigua es más barato.

Por último hay que añadir ciertos **gastos materiales**, como los folios o gastos de impresión de la memoria del proyecto. Estos gastos se muestran en la tabla 5.

Material	Coste
Gastos de impresión (aproximadamente 100 hojas)	0'04*100 = 4€
Paquete de folios (100)	1'5€
Material de oficina	15€

Tabla 5 – Presupuesto material

Haciendo un coste en material de 4+1'5+15 = 20'5€.

Sumando el coste de todas las partes se obtiene el **coste total** del presupuesto.

16773'56+ 138'75 + 8 + 20'5 = **16.940'81€**

1.5. Tiempo y costes finales

Una vez visto el presupuesto y la planificación realizada al inicio del proyecto, se pasa a mostrar tanto los tiempos como los costes finales obtenidos al finalizar el proyecto.

Finalmente los tiempos necesarios para los profesionales han sido los mostrados en la tabla 6, incluyendo el salario neto real de cada uno de ellos. En dicha tabla se aprecia que se ha requerido de una semana más de investigación y 4 semanas más de programación y seguimiento, mientras que de análisis se ha reducido a 20 horas.

Tarea	Horas totales	€/hora	salario neto
Investigación	9*6*5 = 270 horas	10 €/h	270*10 = 2700€
Análisis	20 horas	30 €/h	20*30 = 600€
Programación	18*6*4 = 432 horas	15 €/h	432*15 = 6480€
Seguimiento	18*4 = 72 horas	25 €/h	72*25 = 1800€

Tabla 6 – Coste s. neto profesionales

De igual manera que en el presupuesto se deben tener en cuenta el IRPF y la contribución a la seguridad social, no cambiando los porcentajes de los descritos en el

presupuesto. Los costes finales debido a los profesionales son los mostrados en la tabla 7 mostrada a continuación.

Tarea	S. neto	IRPF	CSST	S. bruto	CSSE	Coste por profesionales
Investigación	2700€	19%	6’4%	2700/(1-0’19-0’064) = 3619’31€	30’9%	3619’31*1’309 = 4737’68€
Análisis	600€	19%	6’4%	600/(1-0’19-0’064) = 804’29€	30’9%	804’29*1’309 = 1052’82€
Programación	6480€	19%	6’4%	6480/(1-0’19-0’064) = 8686’32€	30’9%	8686’32*1’309 = 11370’40€
Seguimiento	1800€	19%	6’4%	1800/(1-0’19-0’064) = 2412’87€	30’9%	2412’87*1’309 = 3158’45€

Tabla 7 – Coste final profesionales

Esto hace un total de 4737’68 + 1052’82 + 11370’40 + 3158’45 = 20.319’35€ en profesionales.

En cuanto al **hardware** necesario, se muestra en la tabla 8 los materiales que finalmente han sido necesarios así como su tiempo estimado de vida, su uso total, su porcentaje de uso, su precio total y el precio por el uso dado en el proyecto. Se aprecia que se ha necesitado un mes más de lo planeado.

Hardware	Tiempo de vida	Precio del hardware	Utilización	Porcentaje de uso	Coste por uso en el proyecto
Ordenador portátil MSI GE62 (1)	60 meses	1300€	7 meses	11’67%	1300*0’1167= 151’67€
Ratón USB Logitech (1)	24 meses	20€	7 meses	29’17%	20*0’2917 = 5’83€
Pendrive 16GB (2) Toshiba	48 meses	15€ * 2 = 30€	7 meses	14’58%	30*0’1458 = 4’38€

Tabla 8 – coste hardware

Así, el precio final de hardware es de 151’67+5’83+4’38 = 161’88€.

En la utilización de **software** ha variado en que no se ha necesitado finalmente el software Codeblocks (aunque es gratuito y no varía el precio), y que Microsoft Office sólo se ha necesitado durante los últimos 5 meses, tal y como se aprecia en la tabla 9.

Software	Tiempo vida software	Precio del software	Utilización	Porcentaje de uso	Coste por uso en el proyecto
Microsoft Office 2013	60 meses	80€	5 meses	8’33%	80*0’1= 6’67€
Windows 10	30 meses	gratuito	6 meses	20%	gratuito
Ubuntu 14.04	30 meses	gratuito	6 meses	20%	gratuito
Valgrind debugger	60 meses	gratuito	6 meses	10%	gratuito

Tabla 9 – coste software

Por lo que el coste final por software es de 6’67€.

Por último, en los **gastos materiales** no se ha necesitado más de lo indicado en el presupuesto, no variando mucho el coste respecto a lo presupuestado, aunque se han prescindido de los folios, y se ha tenido en cuenta el precio de encuadernación y se ha aumentado el número de fotocopias. Estos cambios se muestran en la tabla 10.

Material	Coste
Gastos de impresión (200 hojas)	200*0'04=8€
Material de oficina	10€
Encuadernación	6€

Tabla 10 – coste material

Haciendo un coste en material de 8+10+6 = 24€.

Además de profesionales, hardware, software y gastos materiales se ha necesitado la contratación de un **servicio** de IaaS (Infraestructure as a Service) para poder disponer de un servidor en el cual poder llevar a cabo las pruebas necesarias.

El coste de este servicio es el mostrado en la tabla 11, donde se indica el servicio a contratar, el precio del servicio (en el caso del servidor de prueba en euros/mes), el tiempo que se ha contratado, y el coste final que ha supuesto la contratación del servicio dentro del proyecto.

Servicio	Precio del servicio	Utilización	Coste en el proyecto
Máquina virtual Azure (modelo D13)	580€/mes	1 mes	580*1= 580€

Tabla 11 – Coste servicios

Por lo tanto se tiene un coste por contratación de servicios de 580€.

Sumando el coste de todas las partes se obtiene el **coste total** de.

20319'35+ 161'88 + 6'67 + 24 + 580 = **21.091'9€**

Si comparamos este resultado obtenido con el presupuesto que se realizó al principio del proyecto, se puede observar que se infravaloró el coste final en 21091'9 - 16940'81 = 4151'09€. Este aumento del coste con respecto al presupuesto es debido principalmente a que ha sido necesario más tiempo para el desarrollo del proyecto de lo inicialmente planificado.

CAPÍTULO 2
COMPRESIÓN DE DATOS Y CONCEPTOS BÁSICOS

Se ha mencionado que la compresión puede ser una solución a los problemas de Big Data, y esto es posible por la **jerarquía de memoria**. La jerarquía de memoria es una manera de dividir las distintas formas de almacenamiento que existen en un sistema informático; y según esta jerarquía, se puede dividir en tres niveles distintos: primario, secundario y terciario. Dentro de cada uno de estos niveles se distinguen 6 distintas tecnologías (como se aprecia en la figura 1): *Almacenamiento distribuido, memoria secundaria, memoria de estado sólido, memoria principal, caché de la CPU, registros de CPU*.

- Almacenamiento distribuido: es el almacenamiento fuera del computador, típicamente se refiere al almacenamiento en la nube. Tiene un tamaño máximo indeterminado (se suele suponer ilimitado) y en ciertas ocasiones se puede conseguir de manera gratuita. El tiempo de acceso es muy lento (del orden de segundos o décimas de segundo, dependiendo de la conexión), pues los datos deben viajar a través de la red.
- Memoria secundaria: es el almacenamiento en los discos duros mecánicos. Hoy en día tienen una capacidad del orden de los TB. Tienen un tiempo de acceso relativamente lento, del orden de 10 milisegundos; el tiempo de acceso está sujeto a las propiedades físicas del disco mecánico. Tienen un precio bastante económico y cualquiera puede tener un disco duro de varios TB. Los datos almacenados no se eliminan una vez desconectado de la corriente (no-volátil).
- Memoria de estado sólido: también denominada memoria flash. Puede llegar también al orden de los TB, pero está por debajo de la capacidad de los discos duros. La velocidad de acceso es del orden de pocos microsegundos. Tienen un precio moderado, y éste crece en gran medida a partir de los 100GB de almacenamiento. También son sistemas de almacenamiento no volátiles, aunque tienen un número limitado de escrituras, pudiendo producirse la pérdida absoluta de los datos de forma inesperada; aunque generalmente este número de escritura es muy elevado (del orden del millón de escrituras).
- Memoria principal: típicamente la memoria RAM. Tiene una capacidad bastante más reducida, siendo por lo general no superior a 32 o 64 GB en ordenadores personales estándar. El tiempo de acceso es del orden de los cientos de nanosegundos. Su precio no es excesivamente caro, pero si se tiene en cuenta el precio por GB es muy superior a los anteriores. Al desconectarse de la corriente se pierden los datos almacenados, es una memoria volátil.
- Caché de la CPU: existen tres niveles de caché de CPU: L1, L2 y L3; siendo L1 el más rápido y con menor tamaño y L3 el más lento pero el que tiene mayor almacenamiento. El tiempo de acceso está en el orden de 0’1 a 1 nanosegundos. Su capacidad de almacenamiento es de unos pocos MB. La caché de la CPU es un almacenamiento volátil.
- Registros de CPU: es la memoria más rápida e inmediata. Tiene un tiempo de acceso del orden de picosegundos, pero su capacidad de almacenamiento es de sólo unos pocos bytes. Es volátil.

Entre el almacenamiento distribuido y la memoria secundaria podría distinguirse otra tecnología más: los soportes ópticos, que son básicamente los CDs y DVDs.

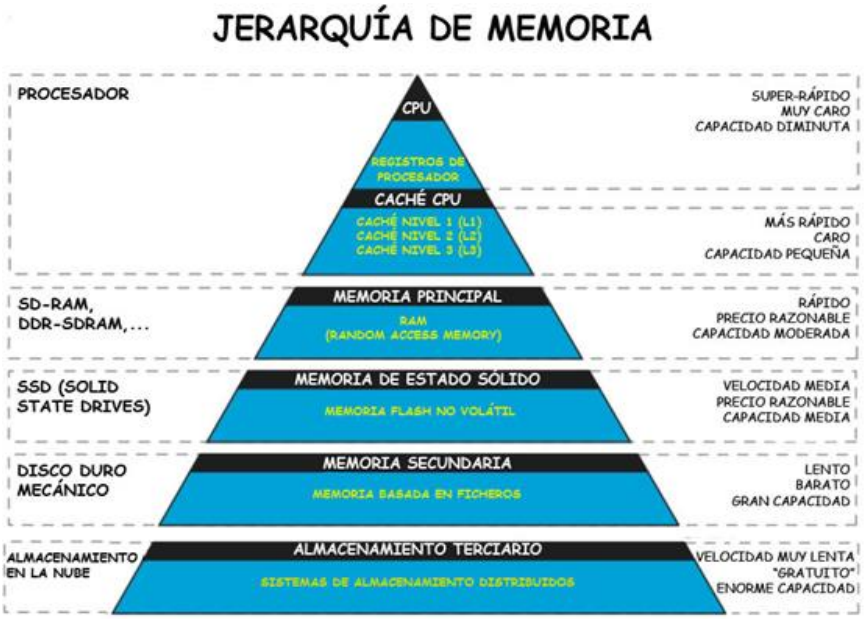

Figura 1 – Jerarquía de memoria

Visto esto se puede deducir que el volumen del *Big Data* fuerza a almacenar los datos en los niveles más bajos de memoria, siendo necesariorecurrir a almacenamiento terciario; sin embargo, viendo la jerarquía de memoria lo recomendable es trabajar en niveles superiores, ya que el tiempo de acceso en memoria secundaria es enormemente más lento que el tiempo de acceso en memoria principal. Aunque hay que tener en cuenta la volatilidad de la memoria principal, teniendo que hacer copias cada vez que se quiera tener persistencia de los datos.

Por lo tanto, la **compresión** permitirá gestionar un mayor volumen de información en niveles superiores de la jerarquía de memoria (generalmente subiendo de almacenamiento secundario a memoria principal). Gracias a esto se consigue mejorar la velocidad de acceso y procesamiento de los datos, a pesar de tener que realizar operaciones con los datos antes de poder utilizarlos. Además esto tiene como ventaja que el procesador se mantiene activo, en lugar de permanecer ocioso esperando a que lleguen los datos; también permite que el posterior almacenamiento en disco de los datos ocupe un menor espacio.

2.1. Definiciones

Antes de abordar las técnicas de compresión es necesario conocer algunos conceptos.

- **Símbolos**: elementos atómicos a partir de los cuales se obtiene la información. Para que un conjunto de símbolos contengan información deberán estar organizados. “A”, “B”, “C”,… son símbolos para la construcción de información en texto en lenguaje natural.
- **Alfabeto**: el conjunto de todos los símbolos utilizados para describir la información construida en los mensajes generados por una fuente de información. El código ASCII es un ejemplo de alfabeto.

- **Mensaje**: Contiene una determinada información expresada mediante la combinación de los símbolos de un alfabeto. Puesto que cada símbolo se puede representar de forma binaria, un mensaje también se puede escribir como un conjunto de bits.
- **Probabilidad**: de un símbolo en un mensaje; es un valor numérico que se calcula dividiendo el número de ocurrencias del símbolo entre el número total de símbolos del mensaje.
- **Entropía**: conociendo la probabilidad de los símbolos de un mensaje se puede calcular la entropía, que representa la cantidad de información contenida en un mensaje. Su fórmula se puede ver en la figura 2, y como se puede apreciar sólo depende de la probabilidad de los símbolos. La entropía es mayor cuanto más equiprobable sea la aparición de los símbolos, mientras que será menor si la distribución de los símbolos es sesgada, teniendo una gran probabilidad unos y muy poca otros. De esta manera se podrán obtener mayores niveles de compresión cuanto menor sea la entropía, ya que el mensaje es más predecible y se puede codificar utilizando menos bits.

$$\mathcal{H} = \sum_{i=1}^{i=\sigma} p_i \log_2 \left(\frac{1}{p_i}\right) = -\sum_{i=1}^{i=\sigma} p_i \log_2 p_i$$

Figura 2 – Entropía

- **Entropía de orden *k***: la entropía tiene en cuenta los símbolos de manera independiente, es decir, no tiene en cuenta la dependencia que pueden tener los símbolos entre sí, pero a veces es más probable la aparición de un determinado símbolo teniendo en cuenta los que le preceden. Por lo tanto si se conocen los *k* símbolos previos es más fácil predecir el siguiente; esta consideración se tiene en cuenta en la entropía de orden *k*, cuya fórmula es la que aparece en la figura 3. De igual manera que en la entropía se tiene que cuanto más predecible sea el mensaje menor será la entropía de orden *k*. Al igual que en la entropía, las mayores compresiones se obtienen con entropías de orden *k* menores.

$$\mathcal{H}_k = \sum_{x_1,\ldots,x_k} p(x_1,\ldots,x_k)\mathcal{H}(x_1,\ldots,x_k)$$

Figura 3 – Entropía de orden *k*

- **Código**: tanto la entropía como la entropía de orden *k* se utilizan para conocer en qué medida se puede comprimir un determinado mensaje. Sin embargo no aporta ninguna información de cómo construir los códigos. Un código es una función que asigna una secuencia de bits única a cada símbolo del alfabeto. De esta manera, ningún símbolo se codificará utilizando la misma secuencia de bits utilizada para otro símbolo.

2.2. Compresión de texto

En el ámbito de la informática, la información se representa a través de mensajes. Dichos mensajes se pueden considerar textos (con independencia de que sean textos en lenguaje natural, biológico, etc.), ya que un texto es una sucesión de caracteres, que realmente son símbolos de un alfabeto; por lo que para comprimir un mensaje cualquiera se pueden utilizar técnicas de compresión de texto. Por esta razón en esta sección se van a enseñar los distintos tipos de compresión de texto que existen, profundizando en algunas de las técnicas de mayor interés para el actual proyecto.

En una primera distinción se consideran dos grandes grupos en las técnicas de compresión: técnicas de compresión con pérdida y técnicas de compresión sin pérdida. Como su nombre indica, en las técnicas de **compresión con pérdida** la información obtenida cuando se descomprimen unos datos comprimidos con estas técnicas es distinta a la que se tenía antes de comprimir, aunque por lo general no son apreciables las diferencias; este tipo de técnicas se utilizan en datos cuya naturaleza permite renunciar a parte de la información sin que esto suponga una pérdida cualitativa de información (por lo general se utiliza en elementos multimedia). Mientras que las técnicas de **compresión sin pérdida** obtienen los mismos datos al descomprimir que los que se tenían antes de comprimirlos. Dentro de la compresión sin pérdida se encuentran la compresión estadística, la compresión basada en diccionario y la compresión basada en preprocesamiento.

Las técnicas de compresión con pérdida no son de interés en el proyecto, por lo que no se profundiza en ellas; sin embargo se utilizarán algunas de las técnicas existentes de compresión sin pérdida, por lo que a continuación se pasa a explicar los distintos tipos de compresión sin pérdida.

2.2.1. Compresión estadística

Las técnicas de compresión estadística se caracterizan en que se componen de dos fases: modelado y codificación. En el modelado se calcula la probabilidad de cada símbolo, de manera independiente o teniendo en cuenta los k símbolos precedentes (compresores desde orden 0 a orden k). Y en la codificación se construye un código según los valores obtenidos en la etapa de modelado (las codificaciones más cortas se le asignarán a los símbolos más frecuentes). Se pueden tener distintos tipos de compresión estadística según se combinen de una manera u otra las etapas de modelado y codificación.

El clásico ejemplo de compresión estadística es la codificación de **Huffman** [6], que consiste en la sustitución de los símbolos del alfabeto por secuencias de bits, sin que ninguna de esas secuencias sea prefijo de otra. Es un código de prefijo óptimo. Al sustituir los símbolos por sus códigos se podrá distinguir donde comienza y termina cada símbolo. Se puede ver un ejemplo de codificación de Huffman en la figura 4, donde se observa que el símbolo “a” se codifica con el código más corto al tener una frecuencia de aparición

muy alta, y los símbolos “d” y “r” se codifican con los códigos más largos al tener una frecuencia de aparición muy baja.

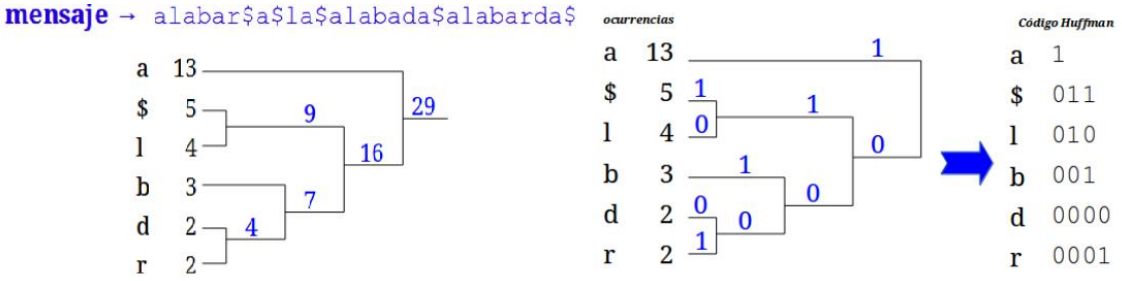

Figura 4 – Codificación de Huffman

Sin embargo, con la codificación de Huffman no se permite comparar lexicográficamente dos mensajes comprimidos con el mismo alfabeto, ya que el código de Huffman no almacena ningún tipo de información respecto al orden lexicográfico de los símbolos dentro del alfabeto. Para esto se creó la variante Hu-Tucker, que básicamente hace que la codificación de los símbolos esté lexicográficamente ordenada, de manera que para cualesquiera dos símbolos *x* e *y*, de manera que *x* sea anterior a *y* lexicográficamente, los códigos de dichos símbolos mantendrán ese orden lexicográfico, siendo el código de *x* anterior lexicográficamente al código de *y*.

2.2.2. Compresión basada en diccionarios

La compresión basada en diccionarios selecciona subsecuencias de símbolos dentro del mensaje y las sustituye por un identificador, teniendo así un diccionario que asigna a cada subsecuencia seleccionada un identificador. Para obtener el código se sustituyen las subsecuencias por su identificador correspondiente. La eficiencia de estos métodos reside en la localización de las subsecuencias adecuadas. La compresión Lempel Ziv (LZ) es un ejemplo de compresión basada en diccionarios, tanto LZ77 [7] como LZ78 [8].

Dentro de la compresión basada en diccionarios existe un tipo particular de técnicas, las basadas en gramáticas, que consisten en que el diccionario se crea a partir de una gramática que se obtiene a partir del mensaje. Se basan en la sustitución del mensaje utilizando unas reglas (usando para cada regla un identificador); estas reglas pueden ser de dos tipos:

- $R_x \rightarrow s_i$
- $R_x \rightarrow R_i|R_j$

Donde s_i es un símbolo del alfabeto, y las distintas R son reglas de la gramática.

Uno de los compresores basados en gramática más utilizados es **Re-Pair** [9], que consiste en inicializar las reglas de la gramática con los símbolos del alfabeto (generando cada regla por un símbolo); a continuación se localizan todos los pares de símbolos que aparecen en el mensaje al menos dos veces, seleccionando el que tiene un mayor número de ocurrencias para crear la siguiente regla; se sustituye el par seleccionado por la regla creada y se repite el proceso buscando el siguiente par de símbolos más frecuente (estos símbolos pueden ser símbolos del alfabeto o reglas generadas previamente). Este proceso se detiene en el momento en el que no hay ningún par que ocurra al menos dos veces.

En el ejemplo de la figura 5 se muestra este proceso, teniendo a la izquierda las reglas generadas, siendo de la 1 a la 6 las creadas antes de comenzar las iteraciones y las siguientes las que se crean según se encuentran los pares más frecuentes. A la derecha aparecen los mensajes que se van generando a lo largo del proceso, mostrando para cada vuelta dos mensajes, uno mostrando el estado actual del mensaje una vez sustituidas las reglas creadas (el que no tiene nada resaltado), y otro que muestra el par con mayor frecuencia en el mensaje (el que resalta los pares de mayor frecuencia). En la parte de los mensajes se observa que se comienza localizando el par más repetido, que es "a$", se crea su regla (regla 7) y se sustituye dicho par por "7"; una vez hecho esto se repite el proceso de nuevo, localizando que el par más repetido es "ab", creando la regla 8 y sustituyendo. Esto se repite hasta que no existan pares que aparezcan más de una vez, tal y como se ve al final de la imagen.

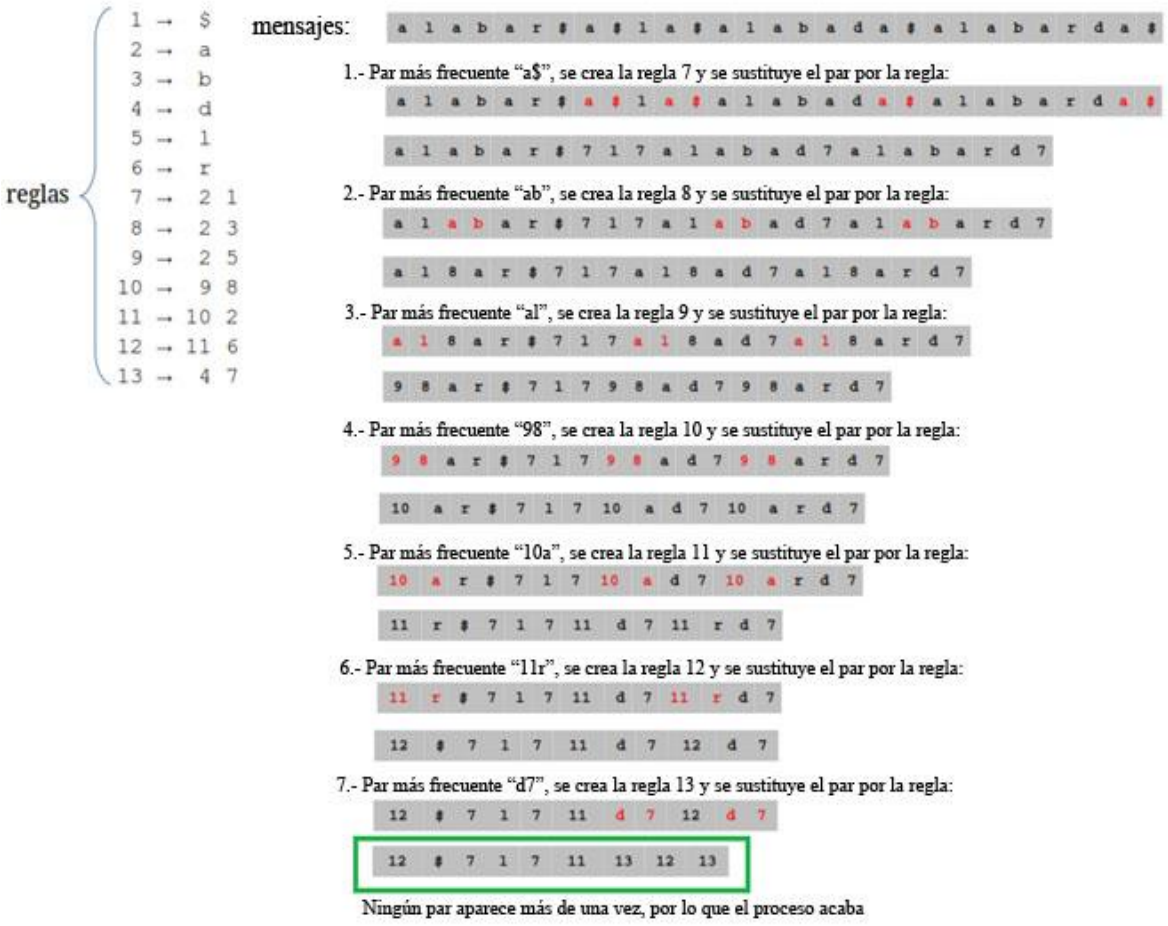

Figura 5 – Codificación Re-Pair

2.2.3. Técnicas basadas en preprocesamiento

Por último, las técnicas de compresión basadas en preprocesamiento realizan un procesamiento previo en el mensaje (M) para obtener una representación equivalente (M'), siendo esta representación equivalente más compresible que el mensaje original.

La **transformada de Burrows-Wheeler** (BWT) [10] es una técnica de preprocesamiento que en el preprocesamiento reorganiza los símbolos del mensaje para mejorar su compresión. Antes de realizar esta reorganización, se añade un símbolo # al final del mensaje, siendo este símbolo lexicográficamente anterior a todos los demás.

Para obtener la transformada de Burrows-Wheeler se realiza una permutación cíclica del mensaje, obteniendo una matriz con tantas filas como símbolos tiene el mensaje. Cada fila de la matriz es igual a la fila anterior movida una posición a la izquierda (de manera cíclica), además cada una de ellas representa (de izquierda a derecha) cada uno de los sufijos del texto. Posteriormente se ordenan las filas de la matriz lexicográficamente según la primera columna, tal y como se muestra en el ejemplo de la figura 6. De la matriz resultante se tiene que la primera columna es el array de sufijos del mensaje (F) y la última columna es la BWT (L).

Figura 6 – BWT

Para obtener de nuevo el mensaje original a partir de la BWT se necesita un elemento más aparte de L, un array C que almacena la posición de F en la que se cambia de símbolo como inicio del sufijo. Con esto se obtiene el mensaje original a partir del último símbolo, que se sabe, que es el primer elemento de L. Este proceso de recuperación del mensaje se explica en la sección de autoíndices (Sección 2.3.4).

2.3. Estructuras de datos compactas

Las estructuras de datos compactas son estructuras de datos modificadas para utilizar un espacio de almacenamiento menor. Aunque puede pensarse que esto simplemente es compresión, no es así, ya que estas estructuras de datos compactas no sólo comprimen los datos almacenados en su interior, sino que también facilitan las operaciones necesarias de acceso a la información. De esta manera se mantiene el acceso directo a la información reduciendo el tamaño de almacenamiento, pudiendo mejorar el rendimiento al poder trabajar en niveles superiores de jerarquía de memoria.

La ventaja principal que tienen las estructuras de datos compactas frente a las estructuras de datos tradicionales es la reducción del tamaño en la estructura necesaria para acceder de manera adecuada a los datos, ya que las estructuras de datos tradicionales suman una gran cantidad de volumen de almacenamiento a los datos originales. Esto se comprende mejor a través un ejemplo.

En 2015 el grafo de la web contenía unos 850 mil millones de nodos y cerca de 10 billones de links [11]. Para almacenar este grafo se necesitarían alrededor de 5TB

En las siguientes subsecciones se pasará a explicar algunas de las estructuras de datos de mayor interés para el actual proyecto, indicando su funcionamiento y operaciones que realizan. Estas estructuras son los bitmaps, secuencias, permutaciones y autoíndices [12].

2.3.1. Bitmaps

Los bitmaps son la base de las estructuras de datos compactas. Un bitmap es un array de bits de *n* posiciones. Los bitmaps además de permitir el acceso directo a cualquier posición del array, proporciona dos operaciones adicionales: *rank* y *select*.

La operación $rank_b(B, i)$ cuenta el número de bits b (0 o 1) que hay hasta la posición "i" dentro del bitmap *B*. En el ejemplo de la figura 7 se observa que $rank_1(B, 13)$ cuenta la cantidad de 1's que hay hasta la posición 13, siendo 7 el resultado; resultado que coincide con el mismo rank hasta la posición 20, ya que entre la posición 13 y la 20 no hay ningún 1.

La operación $select_b(B, i)$ obtiene la posición donde aparece la i-ésima ocurrencia del bit b. Igual que en el rank, si no se indica b se supone que vale 1. También en la figura 7 se tienen ejemplos de select, viendo que el primer 1 se encuentra en la posición 3, el séptimo en la posición 13, y el octavo en la posición 21.

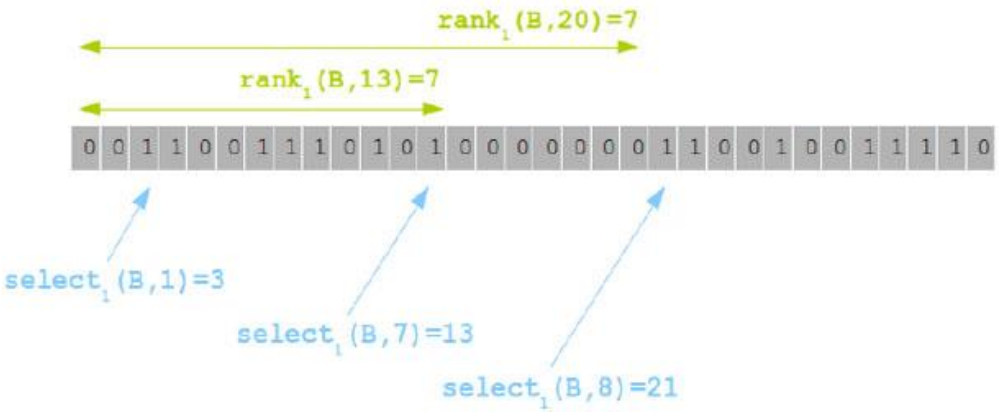

Figura 7 – rank y select

Tanto rank como select pueden resolverse en tiempo constante [13], lo cual asegura la escalabilidad de la estructura, ya que garantiza un rendimiento eficiente con independencia del tamaño del bitmap. El funcionamiento interno es complejo y no se abordará en este trabajo.

2.3.2. Secuencias

Las secuencias son una generalización de los bitmaps, sustituyendo el alfabeto {0, 1} por un alfabeto cualquiera; plantean una forma de representar una secuencia de símbolos procedentes de un alfabeto. Además, al ser una generalización de los bitmaps, estas estructuras proporcionan las operaciones descritas para los bitmaps (access, rank y select).

Para representar las secuencias de símbolos existen diversas estructuras de datos, siendo la más conocida el **wavelet tree**. El wavelet tree es una estructura basada en un árbol, y se basa en la división recursiva del alfabeto. Esta división del alfabeto se lleva a cabo dividiéndolo en dos subalfabetos; a partir de estos dos subalfabetos se crean dos submensajes, que se obtienen extrayendo del mensaje original únicamente los símbolos del subalfabeto correspondiente; los símbolos del subalfabeto izquierdo se codifican con un 0 y los del subalfabeto derecho con un 1. Esta división del alfabeto se realiza de manera recursiva sobre cada submensaje que se genera, hasta que se llega a subalfabetos con un único símbolo cada uno. De esta manera se puede obtener un árbol de altura log(σ), con σ el número de elementos del alfabeto.

A la hora de implementar el wavelet tree, para cada mensaje que se va a dividir se almacena un bitmap con un 0 en las posiciones de los elementos del mensaje que pertenecen al primer subalfabeto y con un 1 en las posiciones de los elementos que pertenecen al segundo subalfabeto. Esto se puede observar en el ejemplo de la figura 8, donde en un primer paso se divide el subalfabeto en dos, siendo el primero {$, a, b} y el segundo {d, l, r}; una vez dividido el alfabeto se crea el bitmap sobre el mensaje original, colocando un 0 en las posiciones del mensaje donde hay un “$”, una “a” o una “b” (primer subalfabeto), y un 1 en el resto (posiciones de los símbolos del segundo subalfabeto). De esta manera se obtienen los dos submensajes que aparecen en el segundo nivel del árbol; para cada uno de estos submensajes se repite la operación hasta llegar a subalfabetos con un único símbolo.

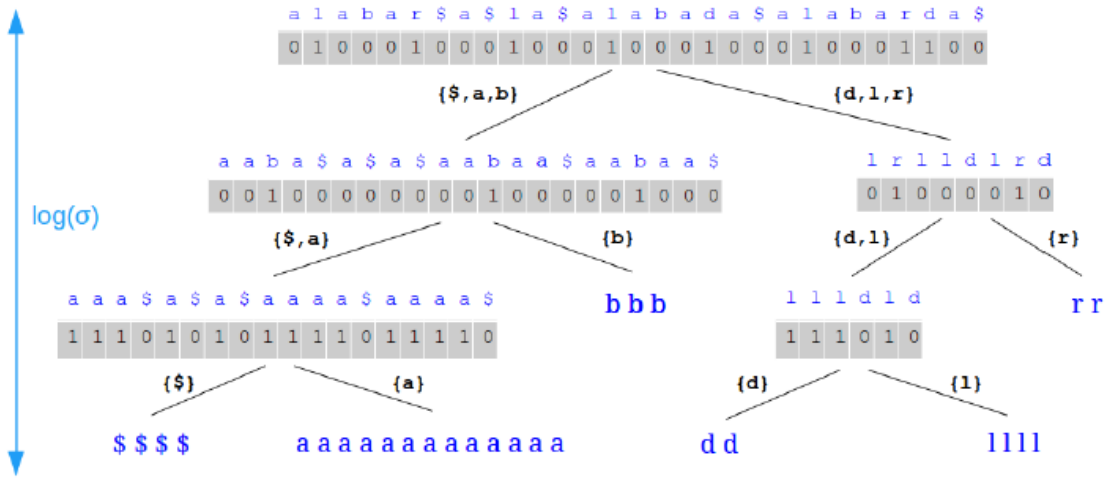

Figura 8 – wavelet tree

Para la realización de la operación de access se recorre el árbol de arriba hacia abajo. Para resolver access(i) se comienza en el primer bitmap (B), obteniendo el bit b=B[i], que indica el submensaje de destino de la siguiente iteración, siendo el primero si b=0 o el segundo si b=1; si el submensaje de destino no es una hoja del árbol (el alfabeto tiene más de un símbolo) se actualiza i=rank$_b$(B, i). A continuación se repite el proceso sobre el siguiente submensaje y se actualiza B al bitmap correspondiente al submensaje. En el momento en el que se alcanza una de las hojas se conoce el símbolo, ya que en las hojas el alfabeto está compuesto por un único símbolo, y ése es el resultado de access(i).

En el ejemplo mostrado en la figura 9 se tiene que se quiere realizar access(11) sobre el mensaje el ejemplo anterior con bitmap B=01000100010001000100010001000110; por lo tanto se tiene i=0, b=B[11]=0, y puesto que el submensaje siguiente no es una hoja hay que seguir; se actualiza i=rank$_0$(B, 11)=8 y se repite el proceso sobre el primer submensaje, ya que b=0. En la segunda iteración se tiene B=00100000000010000010000, por lo que b=B[8]=0, y puesto que el submensaje siguiente no es una hoja hay que seguir; se actualiza i=rank$_0$(B, 8)=7 y se repite el proceso sobre el primer submensaje, ya que b=0. En la siguiente iteración se tiene B=1110101011111011110, por lo que b=B[7]=1. Como el submensaje siguiente (el de la derecha) es una hoja no es necesario continuar, puesto que el resultado de access(11) es el único símbolo del subalfabeto de dicho submensaje, y se tiene access(11)="a".

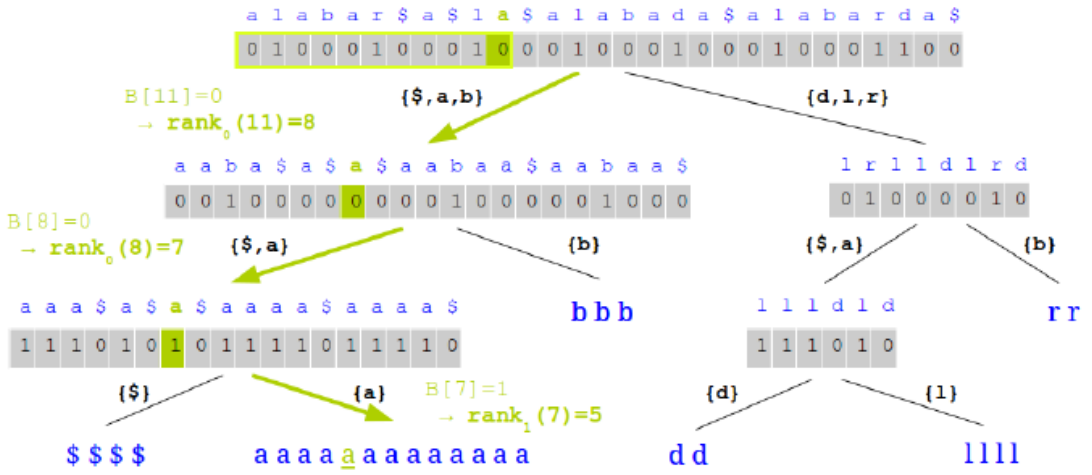

Figura 9 – access wavelet tree

Para la operación de rank se recorre el árbol de arriba hacia abajo al igual que se hace en access. Para resolver $rank_s(i)$ primero de todo se debe conocer en todo momento a qué subalfabeto corresponde el símbolo buscado *s*, o lo que podría considerarse su codificación. Con esto se comienza inicializando *B* al bitmap del mensaje actual, y *b*=0 si el símbolo *s* pertenece al primer subalfabeto o *b*=1 si pertenece al segundo; a continuación se actualiza $i=rank_b(B, i)$; si el subalfabeto siguiente contiene un único símbolo (es una hoja) se termina, siendo el resultado buscado *i*; en caso contrario se repite el proceso actualizando *B* y *b* como si el submensaje actual fuera el primer mensaje.

En el ejemplo de la figura 10 se realiza $rank_l(11)$. Se comienza con *i*=11, *B*=01000100010001000100010001000110, y puesto que "l" está en el segundo subalfabeto se tiene *b*=1; se actualiza $i=rank_b(i)=rank_1(11)=3$, y puesto que el siguiente submensaje no es hoja se realiza otra iteración. Se actualiza *B*=01000010 y *b*=0 (puesto que "l" está en el primer subalfabeto siguiente); se calcula la siguiente $i=rank_b(i)=rank_0(3)=2$, y puesto que el siguiente submensaje no es hoja se realiza otra iteración. Se actualiza *B*=111010 y *b*=1 (puesto que "l" está en el segundo subalfabeto siguiente); se calcula la siguiente $i=rank_b(i)=rank_1(2)=2$, y puesto que el siguiente submensaje es una hoja (sólo está "l" en el subalfabeto) se tiene que el resultado de $rank_l(11)=2$.

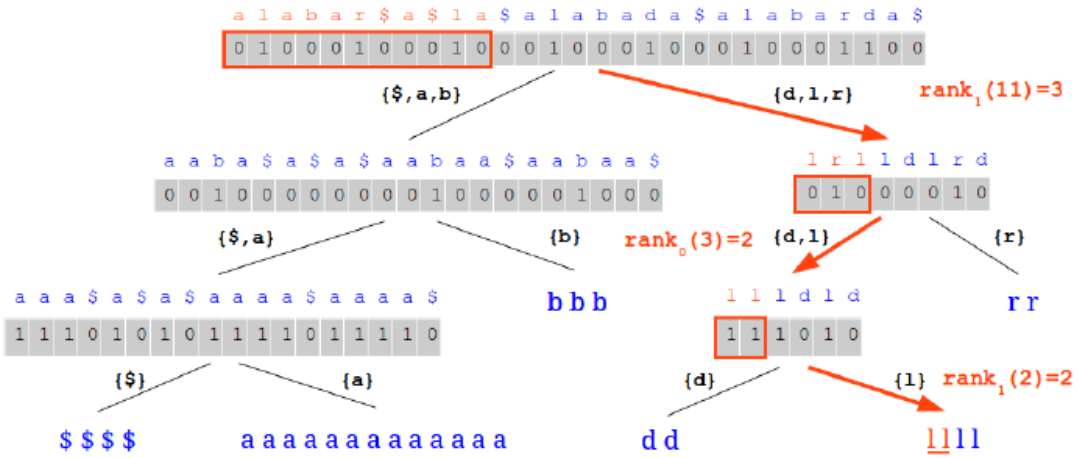

Figura 10 – rank wavelet tree

La operación de select se realiza de forma inversa a como se realiza rank, se recorre el árbol de abajo hacia arriba. Para calcular $select_s(i)$ se comienza en el último submensaje anterior al nodo hoja con el subalfabeto que contiene sólo a *s*, y se inicializa *B* al bitmap del submensaje, y *b*=0 si se proviene del primer submensaje o b=1 en caso contrario; se actualiza $i=select_b(i)$, y si el submensaje actual es el mensaje original se tiene que el resultado final es *i*; si no es el mensaje original aún, se repite el proceso.

En la figura 11 se muestra cómo se realizaría $select_b(2)$ sobre el ejemplo de los casos anteriores. Se comienza en el subalfabeto que contiene sólo "b", y se pasa al submensaje superior; puesto que provenimos del segundo submensaje se tiene *b*=1, y se inicializa *B*=00100000000001000001000; a continuación se actualiza $i=select_1(2)=12$. Al no estar en el mensaje original se repite el proceso; al siguiente submensaje se llega desde el primer submensaje, por lo que se actualiza *b*=0, y *B* al bitmap correspondiente *B*=01000100010001000100010001000110; se actualiza $i=select_0(12)=16$; y puesto que nos encontramos en el mensaje original se termina el proceso, teniendo como resultado $select_b(2)=16$.

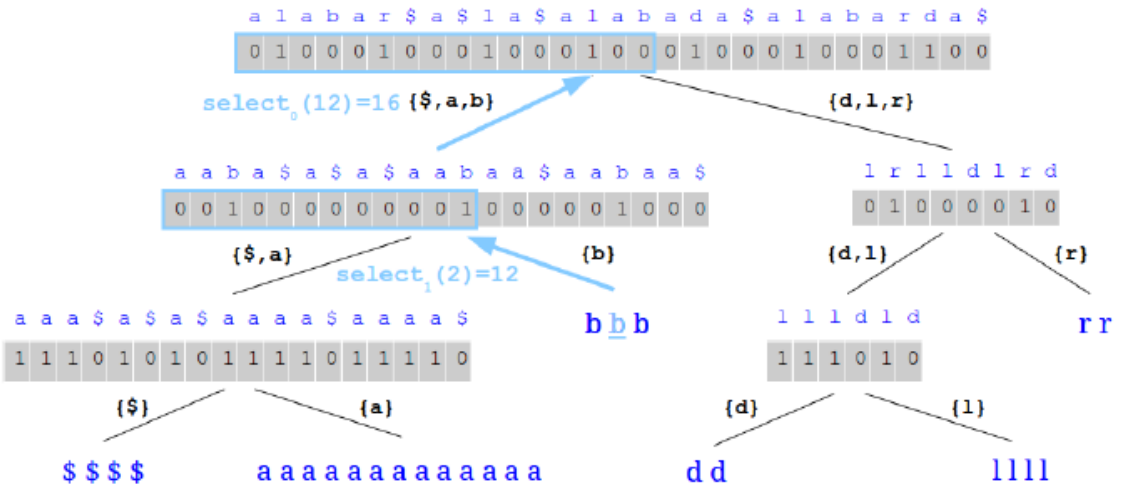

Figura 11 – select wavelet tree

La implementación vista para las operaciones access, rank y select permite la resolución de las mismas en tiempo logarítmico y con un espacio compacto, no requiriendo gran capacidad de almacenamiento. Incluso existe una alternativa al wavelet tree para ocupar aún menos espacio; esta implementación es el Huffman wavelet tree, que consiste en aplicar la codificación de Huffman a los símbolos del alfabeto, asignando así el camino necesario para llegar a ese símbolo, tal y como se muestra en la figura 12.

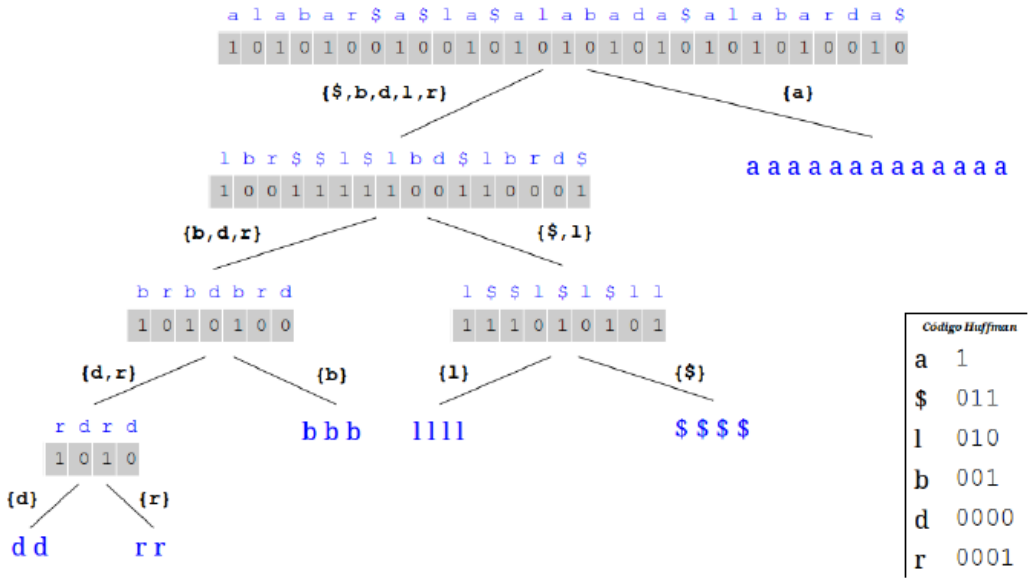

Figura 12 – Huffman wavelet tree

Además del wavelet tree existen otras estructuras de datos para almacenar secuencias. Entre ellas se encuentran el wavelet matrix [14], que es una variante del wavelet tree, mejorando el rendimiento de éste para alfabetos muy grandes. GMR [15] que está diseñada para operar con secuencias de alfabetos de gran tamaño, ocupando más que el wavelet tree, pero garantizando tiempo constante en operaciones de select. Alphabet Partitioning [16] combina GMR y wavelet tree obteniendo las ventajas de ambos, pudiendo operar con grandes alfabetos y con una buena compresión.

2.3.3. Permutaciones

Una permutación consiste en una secuencia que reordena los valores de una secuencia original. Una permutación puede verse como un array de valores de 1 a *n* donde cada valor aparece una única vez, o varias veces en el caso de permutaciones con repetición. Las permutaciones deben realizar dos operaciones básicas: $\pi(i)$ devuelve el valor almacenado en la i-ésima posición de la permutación; y $\pi^{-1}(i)$ devuelve la posición en la que ahora se encuentra el valor que antes ocupaba la posición *i*, siendo así inversa a la primera operación. Las permutaciones se suelen usar como componentes internos de otras estructuras compactas, como GMR o alphabet partitioning; sin embargo también se pueden utilizar de manera directa. Un ejemplo de esta utilización directa podría ser la creación de índices invertidos.

En el ejemplo de la figura 13 se puede observar un ejemplo de permutación, donde en este caso la secuencia estaría formada por palabras, y T sería el mensaje original, y Π el mensaje permutado (aunque en la práctica lo que se almacena es la permutación de los identificadores).

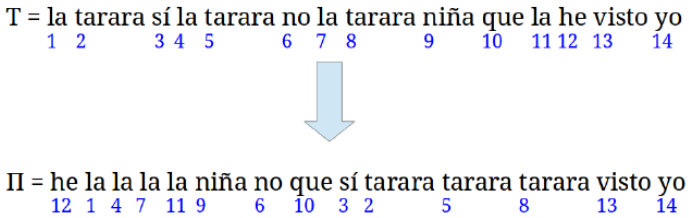

Figura 13 – Permutación

Existen distintas maneras de poder almacenar una permutación. La más sencilla pasaría por utilizar dos arrays, uno con los resultados de la operación π y otro con los de π^{-1}; sin embargo en cuanto a espacio se refiere no es una buena solución, a pesar de tener un tiempo de respuesta constante. Una implementación mejor pasaría por utilizar un wavelet tree, donde la operación $\pi(i)$=access(i), y $\pi^{-1}(i)$=select$_i$(1). Pero existe una manera mejor que el wavelet tree para implementar una permutación; esta implementación se basa en los ciclos de las permutaciones, y es que cualquier permutación se puede dividir en ciclos, volviendo a un elemento tras un determinado número de pasos, siendo el penúltimo paso del ciclo π^{-1}. De esta manera teniendo un único array con los resultados de π se puede resolver dicha operación en tiempo constante y π^{-1} en tiempo proporcional al número de pasos del ciclo. Pero esto conlleva a que si el ciclo es muy grande el tiempo de respuesta de π^{-1} se puede disparar; este problema se soluciona utilizando punteros cada t posiciones para ciclos mayores de t+1 posiciones (con t la que se desee). Esto se puede ver gráficamente en la figura 14, donde t=2. En dicha figura se pueden observar los 5 distintos ciclos que existen en la permutación utilizada como ejemplo; el primer y segundo ciclos son de 5 pasos, por lo que al estar utilizando t=2 se crean punteros cada dos posiciones apuntando hacia atrás, consiguiendo así "subciclos" de como mucho 3 pasos. En los tres últimos ciclos no se requiere crear ningún puntero, ya que ninguno tiene más de tres pasos.

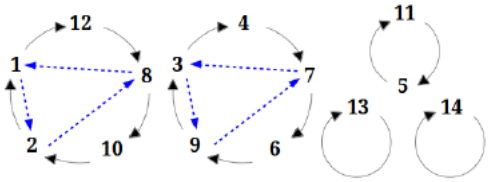

Figura 14 – ciclos permutación

2.3.4. DAC

Aunque la permutación permita reordenar a nuestro gusto los elementos de un mensaje, únicamente trabaja con identificadores. Para extraer el texto de los elementos de manera eficiente es necesario utilizar otro tipo de estructuras. El DAC (Directly Addressable Code) [17] es una estructura de datos que permite el almacenamiento y acceso a códigos de longitud variable; es decir, se pueden almacenar elementos de longitud distinta de manera eficiente. Se puede pensar que sería lo mismo tener todos los códigos de los elementos seguidos uno detrás de otro en un único array, pero para acceder a un elemento concreto no sería tan eficiente como el DAC.

El DAC consiste en dividir los códigos en distintos niveles, teniendo en total tantos niveles como el número de símbolos del elemento más largo. En cada nivel se almacena un array de símbolos (A_i) y un bitmap (B_i); en el array se almacena el *i*-ésimo símbolo de cada uno de los códigos (en orden), y si un código tiene menos de *i* símbolos se ignora (pero no se deja un hueco vacío en el array); en el bitmap se coloca un 0 en las posiciones de los códigos que tienen más de *i* símbolos, y un 1 para los códigos de *i* elementos (y por lo tanto se acaba de almacenar su último elemento en el array).

En el ejemplo de la figura 15 se puede ver cómo en el nivel 1 se almacena el primer símbolo de cada uno de los mensajes (que en este caso el código es el texto explícito), y en el bitmap se pone un 1 en la segunda posición, ya que el código "a" ha almacenado su último símbolo. En el segundo nivel sólo hay 4 símbolos, ya que uno de los códigos no tiene más símbolos, y se almacenan en el array los segundos símbolos de cada código, saltándose el segundo que ya terminó; en este segundo nivel termina el tercer código, y en su posición (la segunda) se añade un 1 al bitmap. De esta manera se continúa hasta que en el último nivel el bitmap sólo tiene unos.

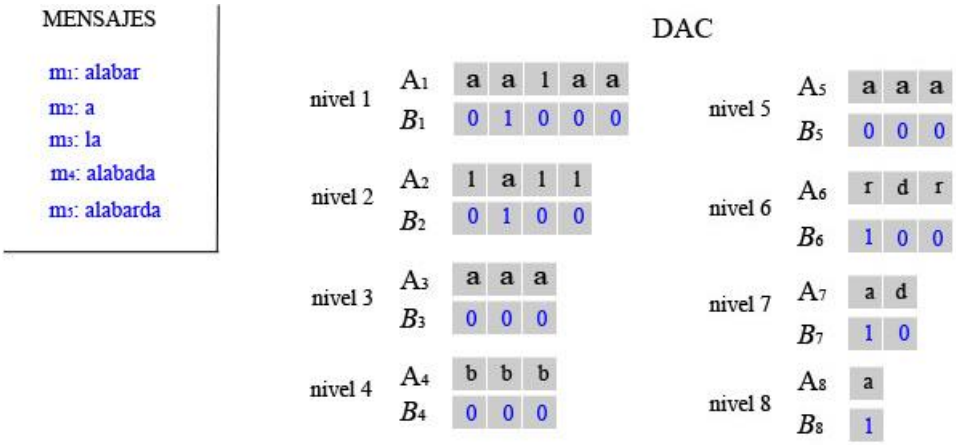

Figura 15 – DAC

La estructura DAC permite acceder a cada uno de los códigos almacenados de manera eficiente. Supongamos que se quiere acceder al código en la posición *x*; para obtener cada uno de sus símbolos se realiza la siguiente iteración:

1. Se comienza con i=1 y se inicializa x a la posición del código que se busca.
2. Se obtiene el siguiente símbolo del código buscado $s_i=A_i[x]$
3. Se comprueba si se ha terminado de extraer el código: $b=B_i[x]$.
    a. Si b=0 se continúa con el paso 4.
    b. Si b=1 se ha terminado de extraer el código.
4. Se actualiza x para la siguiente iteración con $x=\text{rank}_0(B_i, x)$.

Para el ejemplo de la figura 15, si se quiere extraer el tercer código se inicializaría i=1 y x=3; se extrae el primer símbolo s_1=A1[3]="l"; se calcula $b=B_1[3]$=0, que como es 0 significa que no se ha terminado de extraer el código y hay que seguir. Para la siguiente iteración se actualiza $x=\text{rank}_0(B_1, 3)$=2; se calcula el siguiente símbolo del mensaje s_2=A2[2]="a"; se calcula $b=B_2[2]$=1, y como tiene valor 1 significa que se ha terminado y el código buscado es C="la".

2.3.5. Autoíndices

La última estructura de interés que se va a explicar es el autoíndice [12]. Un autoíndice es una estructura que permite representar un texto o mensaje en un espacio proporcional al de su entropía, y además proporciona un acceso indexado a dicho texto. Esto significa que se puede descomprimir el texto (obteniendo cualquier fragmento del mismo), y además se realizan las operaciones necesarias para los textos de manera eficiente. Para ser capaz de esto, utiliza la transformada BWT.

Las operaciones que se realizan en un texto son: count (cuenta el número de ocurrencias de un patrón dado), *locate* (localiza las ocurrencias del patrón dado) y *extract* (extrae el fragmento o fragmentos de texto que contienen el patrón dado). Como ya se ha mencionado, estas operaciones se realizan de manera eficiente.

Existen diversas implementaciones de autoíndices, siendo alguna de ellas: FM-Index, basado en la BWT; autoíndices basados en la compresión LZ; SLP-Index, basado en gramáticas; CSA, que utiliza arrays de sufijos comprimidos. La más sencilla de ellas es FM-Index, que utiliza la transformada de Burrows-Wheeler, explicada anteriormente (sección 2.2.3).

FM-Index [18] utiliza estructuras compactas sobre las estructuras que se utilizan en la BWT. En la figura 16 se muestra cómo quedaría un texto después de aplicar BWT. Y de esta manera teniendo únicamente el array C y la BWT se pueden realizar búsquedas por substrings utilizando búsqueda inversa, es decir, empezando a buscar por el final del patrón, pues la BWT facilita localizar el símbolo que precede a uno dado. Además, la BWT se implementa utilizando un wavelet tree.

El siguiente algoritmo describe como se localizaría un patrón M en el mensaje original, teniendo la BWT, a la que llamaremos L, y el array C:

1. En primer lugar se inicializan ciertos valores:
    a. Al símbolo *s* se le da el valor del último símbolo del patrón.
    b. Se inicializa el rango de interés $R=[a, b]$, siendo $a=C[s]$, y $b=C[s+1]-1$. Si *s* es el último símbolo del array C (por lo que $C[s+1]$ no existe) se toma b igual al número de elementos de L.
2. Se actualiza *s* al símbolo precedente al actual del patrón.
3. Se calcula $x=rank_s(L, a-1)$; $y=rank_s(L, b)$.
4. Se actualiza el rango $R=[a, b]$, con los valores $a=C[s]+x$, y $b=C[s+1]+y-1$ (al igual que antes, si s es el último símbolo del array C se sustituye $C[s+1]-1$ por el número de elementos de L.
5. Se reitera desde el paso 2, terminando cuando se llega al paso 4 con el primer símbolo del patrón. Cuando se llega a este punto se tiene un rango R=[a, b] que indica el número y localización de las ocurrencias en la BWT.

Aunque puede parecer tedioso, es fácil de calcular. Teniendo la BWT del ejemplo de la figura 16, vamos a calcular las ocurrencias del patrón "laba":

- Se comienza iniciando *s*="a", último símbolo del patrón; y los límites del rango de interés *a*=C["a"]=6, *b*=C["a"+1]-1=18, obteniendo R=[5, 18] (nótese que C["a"+1]=C["b"]).
- Se actualiza *s* al símbolo anterior, s="b". Se calcula *x*=rank"b"(L, 6-1)=0; *y*=rank"b" (L, 18)=3. Finalmente se actualiza *a*=C["b"]+0=19, *b*=C["b"]+3-1=21.
- Se actualiza *s* al símbolo anterior, s="a". Se calcula *x*=rank"a"(L, 19-1)=4; *y*=rank"b" (L, 21)=7. Finalmente se actualiza *a*=C["a"]+4=10, *b*=C["a"]+7-1=12.
- Se actualiza *s* al símbolo anterior, s="l". Se calcula *x*=rank"l"(L, 10-1)=1; *y*=rank"b" (L, 12)=4. Finalmente se actualiza *a*=C["l"]+1=25, *b*=C["l"]+4-1=27.
- Como hemos llegado al último símbolo terminamos, teniendo que hay tres ocurrencias en las posiciones 25, 26 y 27 de la BWT.

C	#	$	a	b	d	l	r
	1	5	6	19	22	24	28

1	#alabar$a$la$alabada$alabarda
2	$a$la$alabada$alabarda#alabar
3	$alabada$alabarda#alabar$a$la
4	$alabarda#alabar$alaalabada
5	laalabada$alabarda#alabar$a
6	a#alabar$a$la$alabada$alabard
7	a$alabada$alabarda#alabar$a$l
8	a$alabarda#alabar$alaalabad
9	alaalabada$alabarda#alabar$
10	abada$alabarda#alabar$alaal
11	abar$a$la$alabada$alabarda#al
12	abarda#alabar$a$la$alabada$al
13	ada$alabarda#alabar$alaalab
14	alabada$alabarda#alabar$ala
15	alabar$a$la$alabada$alabarda#
16	alabarda#alabar$a$la$alabada$
17	ar$a$la$alabada$alabarda#alab
18	arda#alabar$a$la$alabada$alab
19	bada$alabarda#alabar$alaala
20	bar$a$la$alabada$alabarda#ala
21	barda#alabar$a$la$alabada$ala
22	da#alabar$a$la$alabada$alabar
23	da$alabarda#alabar$alaalaba
24	la$alabada$alabarda#alabar$a$
25	labada$alabarda#alabar$alaa
26	labar$a$la$alabada$alabarda#a
27	labarda#alabar$a$la$alabada$a
28	r$a$la$alabada$alabarda$alaba
29	rda$alabar$alaalabada$alaba

Figura 16 – Ejemplo FM-Index

CAPÍTULO 3
DICCIONARIOS DE TEXTO

Además de las estructuras de datos explicadas anteriormente, hay otra que es de vital importancia para en el proyecto actual, los diccionarios de texto. Un diccionario de texto es una estructura de datos que organiza un vocabulario de símbolos, que en los diccionarios de texto son strings, asignando a cada uno un identificador único. Y debido a que cada string tiene su propio identificador el diccionario debe implementar al menos dos operaciones básicas: localizar un identificador a partir del string (***locate***), y extraer el string dado su identificador (***extract***).

Usar diccionarios permite reemplazar cadenas de caracteres por simples números (IDs), que pueden ser manejados de manera más sencilla y eficiente; el lado negativo es que se requiere de un mapeo entre los strings y sus ids (de ahí las operaciones de *locate* y *extract*). Sin embargo, esta separación entre strings e ids es positiva a la hora de trabajar con problemas de Big Data, y es que cada vez existen diccionarios con mayor volumen de datos; para ello tener por un lado los identificadores numéricos y por otro los strings permite manejar los identificadores en lugar de los strings para realizar ciertas operaciones, y así conseguir trabajar con los diccionarios en memoria principal.

Como ejemplo para ilustrar el concepto de diccionario, supongamos que tenemos el mensaje T="la tarara sí la tarara no la tarara niña que la he visto yo". Del mensaje extraemos el vocabulario del mismo, que sería V={he, la, niña, no, que, sí, tarara, visto, yo}. A partir de ese vocabulario crearíamos el diccionario asignando a cada elemento del vocabulario un identificador, tal y como se muestra en la figura 17 (por ejemplo sl string "niña" tiene el identificador 3). Teniendo estos identificadores se puede codificar el mensaje original como T'="2 7 6 2 7 4 2 7 3 5 2 1 8 9", ocupando menos espacio y gestionándose de manera más eficiente.

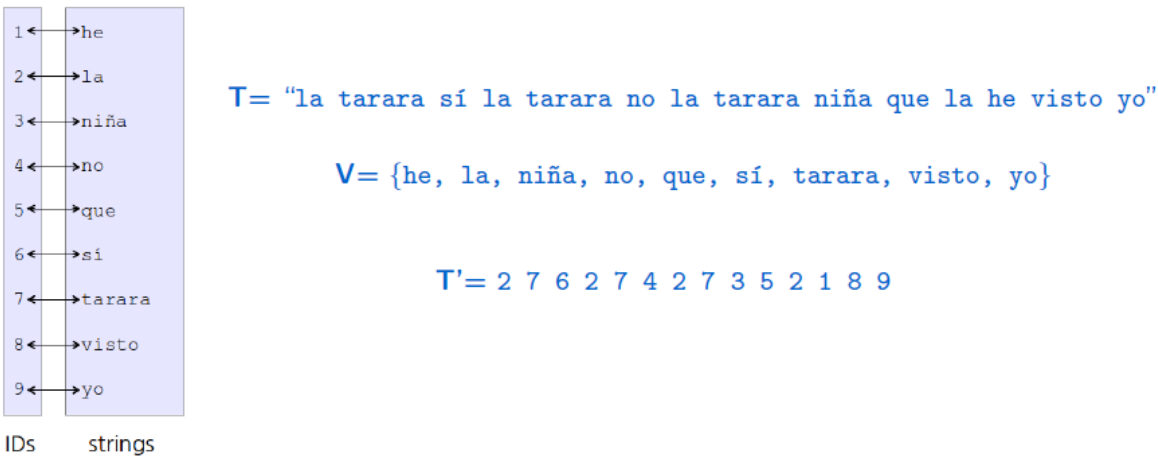

Figura 17 – Ejemplo diccionario

3.1. Motivación y campos de aplicación

Antes de profundizar sobre las distintas posibilidades de los diccionarios de texto conviene saber por qué existen y para qué se utilizan. Es fácil percatarse de que las utilidades de un diccionario de texto son muchas, principalmente en el campo de la recuperación de información. En este ámbito se necesita organizar una gran cantidad de datos, facilitando la extracción de pequeños fragmentos; tarea que encaja perfectamente con las que desempeña un diccionario de texto. Dentro de la recuperación de información los diccionarios se utilizan en varios ámbitos.

Aplicaciones de procesamiento de lenguaje natural, es una de las aplicaciones más clásicas de los diccionarios de texto. Tradicionalmente estos diccionarios no han representado un problema en lo que a volumen se refiere, ya que el tamaño de un diccionario de texto en lenguaje natural crece de manera sublineal, pudiendo tener un diccionario de unos pocos megabytes para un texto de un terabyte. Sin embargo esto no es extensible a otras aplicaciones de procesamiento de lenguaje natural, como por ejemplo en los buscadores web.

Los **buscadores web** utilizan también diccionarios de texto; pero los elementos de búsqueda en la web no son tan simples como un texto en lenguaje natural, ya que los buscadores web trabajan con textos "sucios" (palabras regulares, palabras mal escritas, palabras artificiales, múltiples idiomas,…); esto hace que el diccionario crezca mucho más rápido. Los diccionarios de los *web crawlers* pueden sobrepasar fácilmente el orden de los gigabytes.

Las **bases de datos NoSQL** (Not only SQL) están presentando un rendimiento más escalable que las bases de datos relacionales en el manejo de Big Data. Estas bases de datos incluyen una gran variedad de arquitecturas y tecnologías, la mayoría de ellas utilizando computación distribuida. Debido a esto los costes de transmisión son de vital importancia para el tiempo de resolución de las consultas. Una manera de reducir ese tiempo de consulta es transmitiendo IDs en lugar de datos, no necesitando transmitir tanta información y reduciendo el tiempo de transmisión. Esto se puede realizar si se dispone de un diccionario de texto común que traduzca esos IDs a los strings correspondientes.

En el ámbito de la **web semántica** también son ampliamente utilizados los diccionarios de texto, ya que la web semántica se basa en interconectar conjuntos de datos a través del lenguaje RDF, enlazando estos conjuntos de datos a través de hipervínculos. En este lenguaje las URIs son fundamentales, pero el almacenamiento de estas URIs en plano no es escalable, mientras que utilizando diccionarios de texto se pueden sustituir por IDs reduciendo enormemente el almacenamiento y el coste de transmisión de datos.

Otro ejemplo son los **sistemas de información geográfica (GIS)**, que manejan un gran número de strings. Y al igual que en los casos anteriores se pueden sustituir por identificadores con las consecuentes mejoras.

Y en general cualquier aplicación en la que se manejen gran cantidad de strings y se necesite su recuperación o transmisión obtiene una sustancial mejora utilizando diccionarios de texto, puesto que (como ya se ha explicado varias veces) es más fácil transmitir y recuperar identificadores numéricos que cadenas de caracteres, además de que ocupan mucho menos espacio de almacenamiento.

3.2. Operaciones básicas

Como ya se ha mencionado anteriormente, un diccionario de texto es una estructura de datos que representa un conjunto de strings distintos, permitiendo al menos dos operaciones básicas:

- ***locate*(s)**, siendo *s* un string. Devuelve el identificador correspondiente a *s*, si es que existe, en caso contrario se devuelve 0. Nótese que los identificadores de un diccionario de *n* elementos van de 1 a *n*, siendo números enteros.
- ***extract*(i)**, con i un número entero entre 1 y *n*. Devuelve el string correspondiente al identificador dado.

Aparte de estas dos operaciones que deben implementar todos los diccionarios de texto, existen otras operaciones que se pueden implementar, siendo de gran utilidad en muchas aplicaciones. Estas operaciones son las mismas operaciones de *locate* y *extract* ya mencionadas, pero realizando búsquedas por prefijos o por substrings. Todas ellas tienen como parámetro de entrada una cadena de caracteres (*s*), que indica el prefijo o el substring a buscar; en ninguna de las operaciones se pasa como parámetro un identificador numérico.

- *locatePrefix*(*s*). Devuelve los identificadores de aquellos strings que empiecen por el prefijo *s*. En el caso en el que el diccionario esté ordenado lexicográficamente sólo se necesita localizar el primer y el último string que comienza por el prefijo, ya que todos los demás estarán comprendidos entre ellos.
- *extractPrefix*(*s*). Es una operación equivalente a realizar un *locatePrefix*(*s*) y después realizar *extract* sobre cada uno de los identificadores obtenidos; sin embargo esta operación sobre diccionarios ordenados lexicográficamente se puede realizar de forma más optimizada.
- *locateSubstring*(*s*). Devuelve los identificadores de los strings que contengan el substring dado. No se puede resolver de manera tan trivial como el *locatePrefix* en un diccionario ordenado lexicográficamente.
- *extractSubstring*(*s*). Equivalente a realizar *locateSubstring*(*s*) y después hacer *extract* sobre los identificadores obtenidos. En esta ocasión no se puede optimizar con diccionarios ordenados lexicográficamente.

Estas operaciones tienen bastante utilidad en distintas aplicaciones, como en los buscadores web, que para las sugerencias de autocompletado se utilizan búsquedas por prefijos; o en la búsqueda de puntos de interés en los GIS, que realizan búsquedas por substrings.

A continuación en la figura 18 se muestran algunos ejemplos de las operaciones descritas, utilizando el ejemplo visto anteriormente en la figura 17. En la figura se realiza una operación de *locate*("tarara"), que devuelve el identificador de ese string, que es 7; también se realiza un *extract*(2), que extrae el segundo string ("la"). A continuación se realizan las dos operaciones por prefijos, buscando aquellos strings que comiencen por la letra "n", devolviendo los identificadores (*locatePrefix*) o los strings (*extractPrefix*) (3→"niña" y 4→"no"). Finalmente se buscan aquellos strings que contengan la letra "a" en cualquier parte del string (2→"la", 3→"niña" y 7→"tarara").

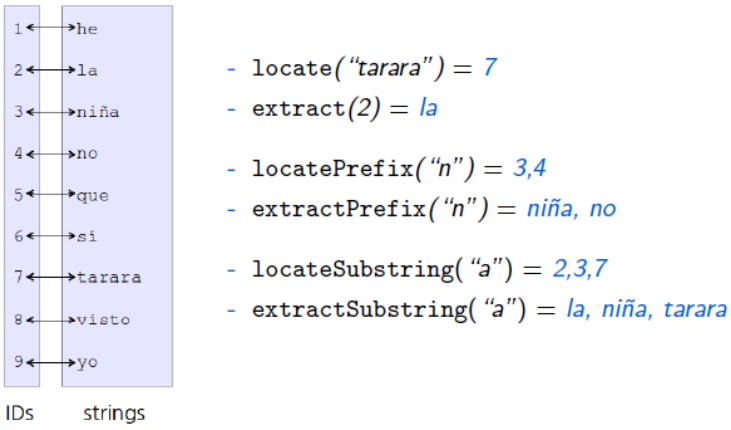

Figura 18 – operaciones de diccionarios

3.3. Estado del arte

En este proyecto se pretende crear un nuevo tipo de diccionario, pero para ello es necesario conocer qué diccionarios de texto existen actualmente, cómo funcionan y cuáles son sus características, así como sus fortalezas y debilidades. Se utilizará como referencia el *survey* recientemente publicado sobre diccionarios de texto comprimidos [19].

Todos los diccionarios que se describen en esta sección son diccionarios que fuerzan a que los strings se ordenen de una manera determinada, no permitiendo poder elegir nosotros el orden en que queremos que se guarden. Por este motivo es que ha llevado a cabo este proyecto, pues se pretende solucionar este problema (capítulo 4).

3.3.1. Hashing

El hashing [20] es una técnica clásicamente utilizada para la implementación de diccionarios, ya sean de texto como de cualquier otro tipo de datos. Este tipo de diccionario utiliza una tabla que almacena las claves de acuerdo al valor obtenido al aplicar una función (función hash) al string.

Para el tratamiento de colisiones se utiliza hashing cerrado (ante una colisión se busca otra posición distinta que se encuentre vacía), usando doble hashing para asignar siguientes posiciones. Como es de suponer, los diccionarios basados en hash no proporcionan búsquedas por prefijos o por substrings, ya que se necesita el string entero para poder obtener su clave y conocer su posición en la tabla hash.

Para construir un diccionario hash simple se aplica una función hash sobre los strings colocándose en una tabla hash normal, se almacenan posteriormente en un array en el orden que tengan en la tabla hash, asignándoles identificadores según el orden. Refinando esta construcción se pueden tener distintos tipos de diccionarios hash, aunque todos ellos pueden combinarse con la utilización de Huffman o Re-Pair para la compresión de los strings. Si se utiliza Huffman, se codifica antes de aplicar la función hash, mientras que si se usa Re-Pair la función hash se aplica sobre el texto plano, tal y como se muestra en las figuras 19 y 20.

En la figura 19 se aplica Huffman sobre los strings en primer lugar, y sobre los códigos obtenidos se aplica el hashing asignando a cada código una casilla de la tabla hash. Una vez realizado esto se comprime la tabla hash utilizando las estructuras H y S, donde H representa la tabla hash, indicando en cada casilla la posición de S donde comienza el string correspondiente, y S representa los strings codificados.

De manera similar se realiza en la figura 20, cambiando S por C, siendo C un array que almacena los strings codificados con Re-Pair. A diferencia que con Huffman, la codificación con Re-Pair se realiza después de aplicar hashing, es decir, la tabla hash se crea con los strings en plano sin codificar.

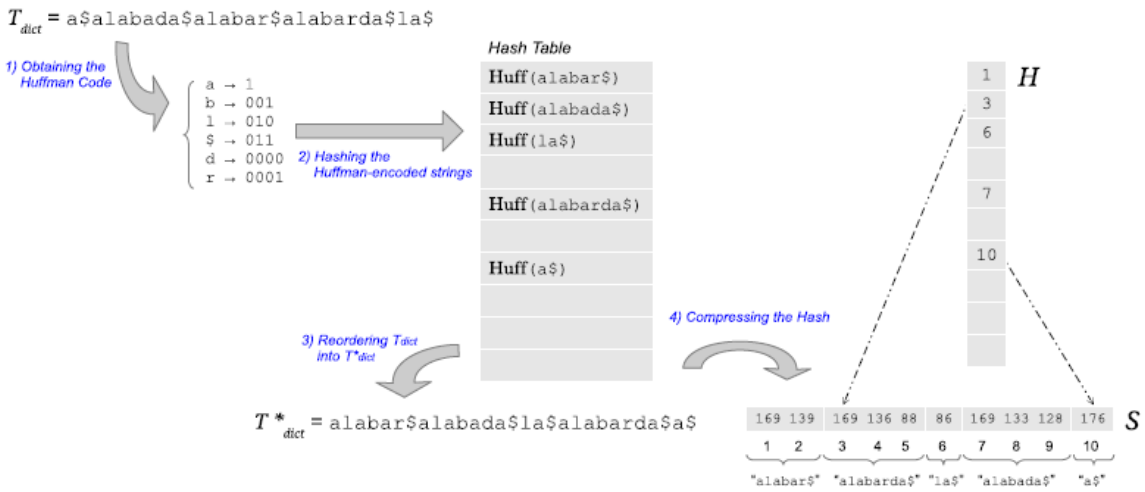

Figura 19 – Hash + Huffman

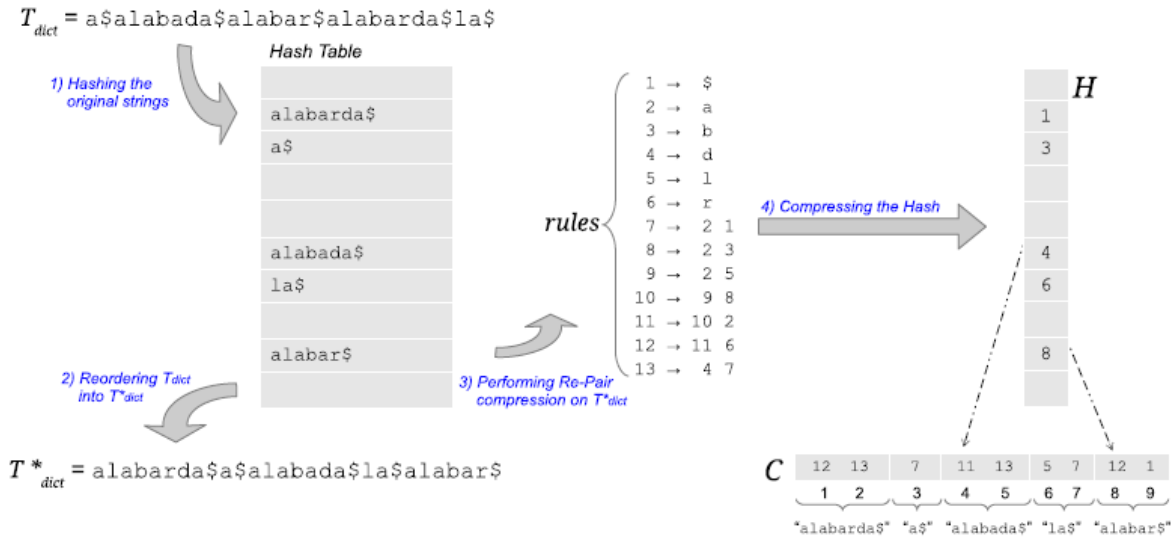

Figura 20 – Hash + Re-Pair

La **codificación en plano de la tabla hash (Hash)** es la manera más sencilla, la tabla se codifica en un array de *m* posiciones. Se utiliza un bitmap (*B*) de *m* posiciones para indicar las posiciones que se encuentran ocupadas. Para realizar un *locate*(*s*) se realiza la función hash sobre el string *s*, lo que da una posición de la tabla hash, se compara s con el elemento de dicha casilla; si coincide se devuelve la posición de la casilla (no la posición de la tabla hash, sino el $rank_1$ de la posición de la tabla hash); si no coincide se repite la operación aplicando el doble hashing. Para el *extract*(*B, i*) se devuelve el valor que haya en la posición $select_1(i)$. Sirva de ejemplo la figura 21, que representa el ejemplo anterior utilizando la codificación de Huffman.

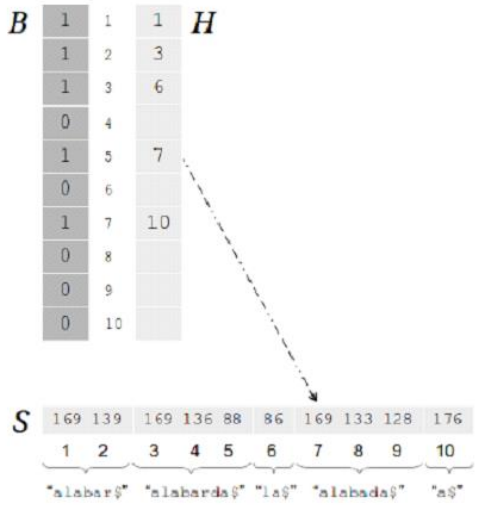

Figura 21 – Hash plano

También se puede implementar utilizando **compresión sobre la tabla (HashB)**. Básicamente consiste en eliminar las casillas vacías del array de la tabla hash, quedando un array compacto sin casillas sin utilizar; el bitmap se mantiene igual.

También se puede realizar una **recompresión sobre la tabla (HashBB)**, utilizando el hecho de que los valores de la tabla son crecientes (pues se asignan de forma creciente) se crea un bitmap con las mismas posiciones que S, teniendo 1 en las posiciones en las que comienza la codificación del string (posición igual al valor que había antes en la tabla).

La última implementación a explicar es el acceso directo a la tabla comprimida (**HashDAC**), que se basa en que el array S puede verse como un conjunto de n subsecuencias de longitud variable, que pueden reorganizarse utilizando la estructura de datos DAC, perdiendo la necesidad de almacenar un bitmap, ya que en el DAC se pueden distinguir las secuencias. El resultado de realizar esto sobre el diccionario utilizado hasta ahora como ejemplo es el mostrado en la figura 22. En esta figura se tiene B, representando las posiciones que están ocupadas, y la estructura DAC que representa los strings codificados con Huffman.

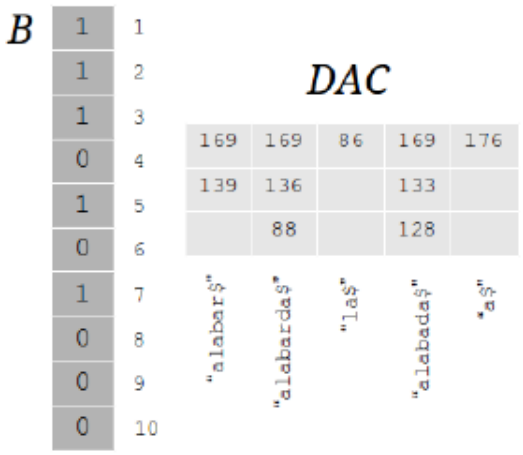

Figura 22 - HashDAC

Con lo visto se puede deducir que en cuanto a espacio los diccionarios Hash pueden llegar a ser competitivos (según los sistemas de compresión que se utilicen), y tienen la gran ventaja de poder realizar *locate* y *extract* en tiempo constante (de manera prácticamente directa), salvo en caso de colisiones; sin embargo tienen el inconveniente de que sólo admiten búsquedas completas, sin poder realizar búsquedas por prefijos o substrings.

3.3.2. Front-Coding

La técnica Front-Coding [21] plantea una codificación diferencial que aprovecha la similitud existente entre los prefijos de los strings consecutivos (siempre y cuando éstos estén ordenados lexicográficamente). La codificación se realiza utilizando dos valores para cada string: un valor entero que indica el número de símbolos del prefijo que comparte con su predecesor, y el resto del string en texto plano. Se debe saber que estos diccionarios se ordenan lexicográficamente para mejorar su rendimiento (desordenados no aportarían compresión alguna al no tener prefijos comunes).

La codificación diferencial consigue reducir el tamaño de un diccionario en gran medida; sin embargo no es eficiente ya que para realizar cualquier tipo de consulta es necesario realizar las búsquedas de manera secuencial, decodificando cada uno de los strings para poder decodificar el siguiente. Para evitar esto y no necesitar decodificar todos los strings anteriores para obtener uno en concreto se recurre a la división en buckets o grupos de strings contiguos; una vez dividido el diccionario, se codifica cada uno de estos buckets de manera independiente con Front-Coding, teniendo el primer elemento de cada bucket almacenado explícitamente.

Del mismo modo que en los diccionarios basados en hash, existen distintas formas de implementar diccionarios basados en Front-Coding.

La implementación más simple es el **Plain Front-Coding (PFC)** o Front-Coding en plano. Ésta es una técnica orienta a byte, que utiliza VBytes (variable-byte) [22] para codificar los valores de la longitud del prefijo común de los strings, y el resto de caracteres se codifican en plano utilizando un byte por cada carácter; el símbolo de finalización de

string (\0) también se codifica como un carácter más. Una vez realizada esta codificación se colocan todos los buckets en orden en un único array de bytes. Para conocer en qué posición comienza cada bucket se utiliza un array de punteros que señalan la primera posición del bucket correspondiente.

A pesar de representar todo en un único array se puede saber en qué elemento se encuentra en todo momento, pues se inicia una búsqueda en el principio de un bucket, donde se encuentra el primer string en plano (cabecera), terminando cuando se encuentra el byte de finalización de string (\0); los siguientes bytes corresponden al número que indica la longitud del prefijo común, siendo el último de ellos el byte que comienza por 0; después de esto empieza el resto de caracteres del string terminando con el carácter de finalización; así se continúa hasta el final del bucket o hasta que se localiza el elemento deseado.

Como ejemplo, en la figura 23 se muestra un ejemplo de diccionario PFC utilizando buckets de 4 strings.

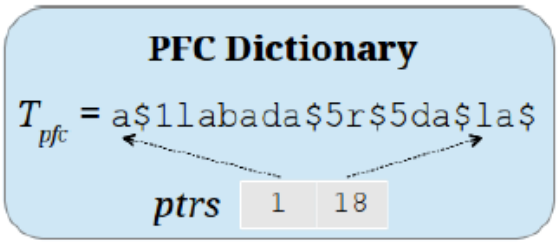

Figura 23 – Diccionario PFC

En el diccionario PFC existe bastante repetición de símbolos, y de esto se puede sacar provecho utilizando Hu-Tucker (similar a la codificación de Huffman, con la ventaja de poder comparar strings) para conseguir una representación más compacta, sacrificando algo de velocidad en las operaciones. Por lo tanto, aplicando esto a un diccionario PFC se obtiene un diccionario **Hu-Tucker Front-Coding (HTFC)**. Este diccionario se consigue codificando las cabeceras de los buckets con Hu-Tucker; el resto de strings del bucket se codifican con Huffman o Re-Pair (tanto el VByte como el sufijo). De esta manera se pueden comparar strings con las cabeceras sin necesidad de descomprimir. Su almacenamiento es similar al PFC, un array con todos los buckets (sólo que ahora están comprimidos) y un array de punteros hacia las distintas cabeceras.

Estos diccionarios además de las operaciones de *locate* y *extract* permiten las búsquedas por prefijos. En los diccionarios Front-Coding se realizan de la siguiente manera las operaciones:

- *locate*(*s*): Se realiza una búsqueda binaria sobre las cabeceras, localizando cual es la última anterior, siendo en su bucket en el que se debe encontrar la cadena solicitada. Se recorren i elementos en el bucket hasta encontrar el que coincide, y se devuelve $b*Bx+i$ como resultado del *locate*, siendo b el número de strings por cada bucket y Bx el número del bucket actual. Nótese que si la cabecera coincide con la cadena de búsqueda i=0. Si se recorre todo el bucket y no se encuentra nada se devuelve 0.

- *extract*(i): Primero se localiza el bucket, que es $Bx = \lceil i/b \rceil$. Una vez localizado el bucket se decodifican uno a uno los *(i-1)mod b* stings internos del bucket, siendo el último el resultado.
- *locatePrefix*(*s*): De manera similar al *locate*(*s*) se localizan el primer y el último string del diccionario que coinciden con *s*, devolviendo todo el rango entre ellos.
- *extractPrefix*(*s*): Se realiza un *locatePrefix*(*s*) y se realiza *extract* sobre los resultados.

Por todo lo visto se puede concluir que los diccionarios Front-Coding aportan la ventaja de poder realizar búsquedas por prefijos, cosa que en los diccionarios basados en hash no se podía. Además en muchos tipos de datos se obtienen muy buenos resultados de compresión (como por ejemplo en páginas web con muchos prefijos en común). En cuanto al rendimiento hay que tener en cuenta el tamaño del bucket, y es que a un tamaño de bucket muy grande se obtiene una mayor compresión, pero se pierde velocidad al realizar las operaciones; mientras que si se tiene un tamaño de bucket pequeño se tienen operaciones más rápidas, pero el nivel de compresión en menor; por lo que el tamaño de bucket es un valor que hay que tratar con cuidados y variará según el tipo de diccionario y las necesidades que se tengan en el mismo.

3.3.3. Re-Pair

Los diccionarios Re-Pair tienen como idea base un simple array donde se almacenan los distintos strings. A este array se puede representar utilizando DAC para no necesitar de símbolos de terminación de strings y poder acceder de manera directa a cualquier posición (haciendo más rápido la operación *extract*); y puesto que también se desea favorecer las operaciones de *locate*, se ordena el diccionario lexicográficamente. Pero esto que se tiene hasta el momento ocupa mucho espacio, pues se almacenan los strings explícitamente, por lo que a cada uno de ellos se codifica utilizando Re-Pair (se codifica usando Re-Pair antes de crear el DAC). De esta manera se consigue un diccionario **RPDAC**, que combina Re-Pair y estructuras compactas.

Este tipo de diccionarios tiene la ventaja de que consigue un buen nivel de compresión, pero a nivel de velocidad se queda bastante por detrás de los demás en la operación de *locate*, ya que para acceder a cada uno de los strings para comparar se necesita decodificarlos primero (al menos parcialmente). Sin embargo para la operación de *extract* tiene un buen rendimiento.

3.3.4. Autoíndices

Los diccionarios full-text utilizan el autoíndice FM-Index, visto en la sección 2.5. El texto utilizado para el FM-Index son los strings del diccionario (incluyendo el carácter de finalización), y también se añade un símbolo de terminación de string al principio del todo. De esta manera para hacer un *locate*(*s*) se realizaría la búsqueda de coincidencia mostrada en la sección 2.5 sobre "\0*s*\0", delimitando el inicio y el fin del string. También se implementa un array (*A*) que almacena la posición original array de sufijos (antes de ordenarlo). Así, para el *extract*(*i*) se sabe que el carácter de finalización del elemento anterior se encuentra en A[*i*+1], y el carácter de finalización suyo en A[*i*+2].

La principal desventaja de este tipo de diccionarios reside en que actualmente no se implementan con técnicas de compresión, por lo que no funciona muy bien para ahorrar espacio de almacenamiento.

3.3.5. Tries comprimidos

Un trie [23] consiste en la codificación de un diccionario utilizando un árbol, de tal manera que recorriéndolo de raíz a hojas se lea uno de los strings, obteniendo uno distinto en cada uno de los posibles caminos existentes. El camino de más a la izquierda correspondería con el primer elemento y el último con el camino de más a la derecha. Por lo tanto se tiene que cada rama codifica a un string, tal y como se muestra en la figura 24.

En un trie las consultas de *locate*(*s*) se resuelven recorriendo los nodos que coincidan con los caracteres de *s* desde la raíz hacia las hojas; el ID está asociado a la hoja en la que se acaba el recorrido. Mientras que el *extract*(*i*) realiza la operación inversa, comienza el recorrido en la iésima hoja y recorre el árbol hacia arriba, obteniendo el string en orden inverso, teniendo sólo que invertirlo para obtener el resultado. También permiten consultas por prefijos, aunque la mayoría no permite búsqueda por substrings (otros como el XBW sí lo permite).

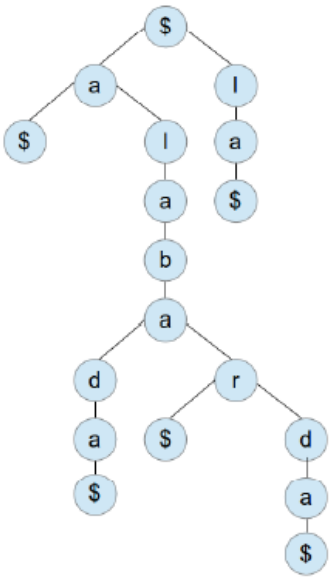

Figura 24 - Trie

El problema de los tries es que requieren de mucho espacio de almacenamiento si no se utiliza compresión y se almacena en plano, por lo que se hace imprescindible el uso de alguna técnica de compactación para que sea escalable. Una de las maneras de representar un diccionario de forma compacta utilizando tries es **XBW**, técnica que representa tries en un espacio comprimido. XBW está basado en FM-Index, diferenciándose en que está orientado a indexar un árbol etiquetado en lugar de un texto. Además añade las búsquedas por substrings de las que los tries normales carecen.

Para implementar XBW se utilizan dos estructuras: un array que almacena las etiquetas de los nodos del trie recorrido en preorden, y un bitmap que marca el último hijo de cada nodo (el situado más a la derecha) tal y como se muestra en el ejemplo de la figura 25. El array se implementa como un wavelet tree para facilitar sus operaciones.

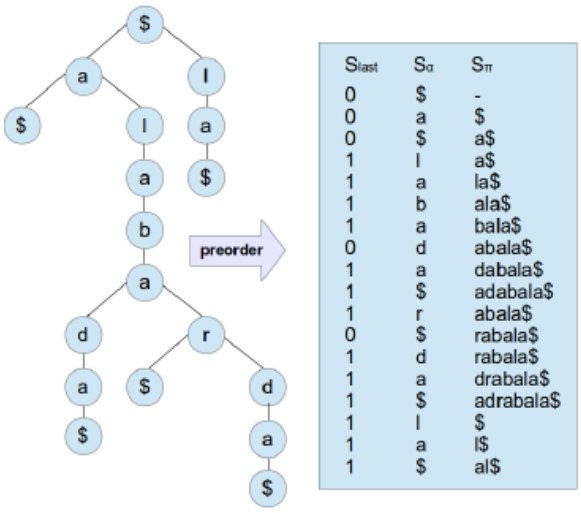

Figura 25 - XBW

3.4. Retos abiertos

Como se ha podido observar en la sección anterior, todos los diccionarios necesitan de un orden determinado (los front coding el orden lexicográfico, los hash el orden que tienen en la tabla hash,…). Esto no es un problema para aquellas aplicaciones que no necesiten tener un ranking en los strings. Sin embargo, si queremos tener un orden determinado para los strings que no sea el determinado por el diccionario hay que recurrir a otro tipo de diccionarios.

Actualmente no existen diccionarios comprimidos de texto que permitan mantener un orden en los strings. Por esto en este proyecto se han desarrollado diccionarios que sí lo permitan, y son los que se describen en el siguiente capítulo.

CAPÍTULO 4
DICCIONARIOS ORDENADOS

Tal y como se ha explicado anteriormente, los diccionarios que existen actualmente fuerzan a mantener un orden que el propio diccionario determina (como el orden lexicográfico en los Front Coding), sin que la aplicación final pueda escoger su propio criterio de ordenación. Por ello, en este proyecto se han desarrollado dos nuevos tipos de diccionarios que permiten utilizar cualquier tipo de ranking que se desee, es decir, no se necesita un orden determinado para poder crear el diccionario y la aplicación final puede elegir el orden de los strings en el diccionario. Este tipo de diccionarios son los que denominaremos **diccionarios ordenados**, ya que mantienen el orden en los strings que se desee.

El primero de los diccionarios que se ha desarrollado utiliza una idea muy sencilla: añadir una permutación sobre los diccionarios existentes, permitiendo que el diccionario mantenga el orden interno, y a través de la permutación obtener el ranking. Por otra parte, el segundo diccionario tiene una mayor complejidad, y se explica en la sección 4.2.

4.1. Ranked String Dictionary Simple

Como ya se ha mencionado, el primer diccionario que se ha desarrollado tiene un funcionamiento bastante sencillo, ya que consiste en una permutación y un diccionario cualquiera de entre los explicados en el capítulo 3 (que utilizan un orden específico no establecido por el usuario).

Para crear el Ranked String Dictionary Simple (RSD-S) se siguen los siguientes pasos:

1. Ordenar los strings según el orden del diccionario contenido, y crear una permutación para poder obtener el ranking original.
2. Crear el diccionario contenido a partir de los strings que se acaban de ordenar.

En la figura 26 se puede ver ilustrado el funcionamiento de este diccionario utilizando el ejemplo visto en anteriores ocasiones. En esta ocasión el ranking que se desea tener es el orden de aparición de las palabras en la oración “alabar a la alabada alabarda”. Para construir el diccionario se ordenan de manera lexicográfica (pues se utiliza un PFC), manteniendo para cada una de las palabras su identificador; una vez hemos realizado este paso tenemos lo que se ve en la parte superior de la figura 26 en “orden lexicográfico”; y a partir de esos identificadores se crea la permutación Π, que se encargará de transformar la posición en orden lexicográfico a la posición del ranking original. Finalmente, se utiliza un diccionario PFC para almacenar los strings, aunque podría utilizarse cualquier otro de los vistos en el capítulo 3 (pudiendo necesitar un orden distinto al lexicográfico).

De esta manera, para llevar a cabo las operaciones necesarias en el diccionario se realizarán esas mismas operaciones en el diccionario contenido por el RSD-S, pasando por la permutación. Aunque es de notar que el rendimiento del diccionario es peor que el propio diccionario contenido, ya que el tiempo de respuesta de las operaciones será igual al tiempo de dicho diccionario sumándole el tiempo de acceso a la permutación. Sin embargo, a pesar de este tiempo añadido al original (aunque no demasiado, ya que la permutación tampoco añade una gran carga) se tiene como ventaja que se permite mantener el orden que se desee, no estando forzado a mantener un orden lexicográfico.

De aquí en adelante al diccionario que se utilice dentro del RSD-S se le denominará diccionario contenido.

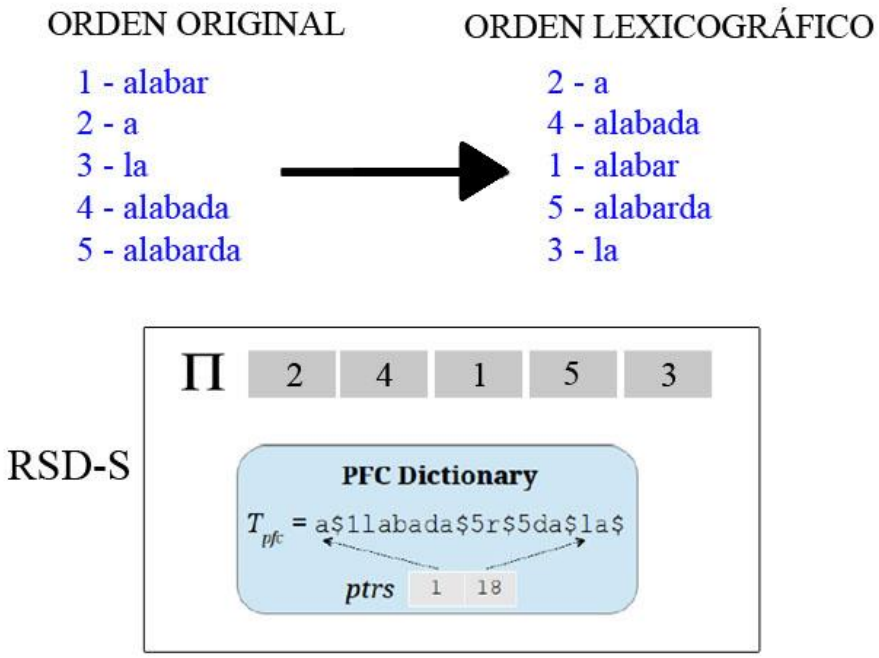

Figura 26 – Ranked String Dictionary - Simple

Como es de imaginar, este tipo de diccionario implementa las mismas operaciones que permita el diccionario contenido en el RSD-S, es decir, si el diccionario es un Hash, el RSD-S no permitirá las operaciones de prefijos, mientras que si se tiene un PFC como en el ejemplo anterior sí permitirá estas operaciones.

Para llevar a cabo la operación de ***locate*(s)** se realiza *locate*(s) en el diccionario contenido, y al resultado obtenido se le realiza la operación π, tal y como se puede observar en el ejemplo siguiente, que muestra cómo se obtiene *locate*("alabarda") en el ejemplo anterior:

- $i = locate_{PFC}$("alabarda") = 4
- $\pi(4) = 5$

Tal y como puede observarse, se tiene primero que en el PFC "alabarda" se encuentra en la posición 4, y al realizar la operación $\pi(4)$ se tiene que "alabarda" se encuentra en la quinta posición del ranking original.

La operación de ***extract*(i)** se realiza de forma similar al locate, con la diferencia de que en esta ocasión se accede a la permutación antes de solicitar el *extract* al diccionario contenido, de manera que se realiza $\pi^{-1}(i)$ y el resultado de esta operación de la permutación es lo que se le pasa al diccionario contenido para que realice el *extract*, siendo el resultado final el string que devuelva éste. En el siguiente ejemplo se muestra cómo se realizaría *extract*(1) usando el diccionario de la figura 26:

- $\pi^{-1}(1) = 3$
- $extract_{PFC}(3)$ = "alabar"

La operación de ***locate prefix*** (en el caso de que el diccionario dentro del RSD-S permita operaciones de prefijos) se implementa de forma similar a *locate*, se realiza *locate prefix* en el diccionario contenido y se realiza la operación π de la permutación sobre cada uno de los elementos que devuelva la operación anterior.

Y en cuanto a la operación ***extract prefix*** se realiza simplemente la operación de *locate prefix* del RSD-S descrita anteriormente y sobre cada uno de los resultados se realiza *extract*, obteniendo así los strings buscados, ordenados según el ranking original.

La conclusión que puede obtenerse finalmente al conocer este diccionario es que tiene las mismas características del diccionario que contenga, permitiendo realizar las operaciones que éste implemente y teniendo un rendimiento similar; sin embargo hay que tener en cuenta que el tiempo de respuesta de cualquiera de las operaciones será siempre superior, ya que es necesario realizar siempre una consulta en la permutación. Este tiempo extra gastado en la permutación es el coste de poder tener un diccionario rankeado con el orden que se desee.

En este proyecto se ha implementado el RSD-S con los diccionarios Front Coding, incluyendo el Plain Front Coding (PFC), Re-Pair Front Coding (RPFC), Hu-Tucker Front Coding (HTFC), Re-Pair Hu-Tucker Front Coding (RPHTFC), Huffman Hu-Tucker Front Coding (HHTFC), Re-Pair DAC (RPDAC), y Hash Re-Pair DAC (HASHRPDAC). Sin embargo se puede implementar con cualquiera del resto de diccionarios.

4.2. Ranked String Dictionary Advanced

Un vez se ha estudiado la manera más sencilla de crear un diccionario ordenado utilizando una permutación, se pasa a estudiar una manera más novedosa de crear este tipo de diccionarios, sin utilizar ninguno ya existente. Este nuevo diccionario será el denominado **Ranked String Dictionary Advanced (RSD-A)**.

Para crear este diccionario se siguen 4 pasos distintos, que se muestran en el ejemplo de la figura 27:

1. Ordenar el vocabulario del diccionario lexicográficamente.
2. Dividir el vocabulario en buckets de tamaño *b* (4 en el ejemplo de la figura 27).
3. Almacenar en una permutación P el bucket en el que se encuentra cada string (utilizando el orden original para almacenar el número de bucket en P).
4. Reordenar los elementos de cada bucket utilizando el ranking original, y almacenar en un array H la posición en la que se encuentra la cabecera (el primer elemento en orden lexicográfico).

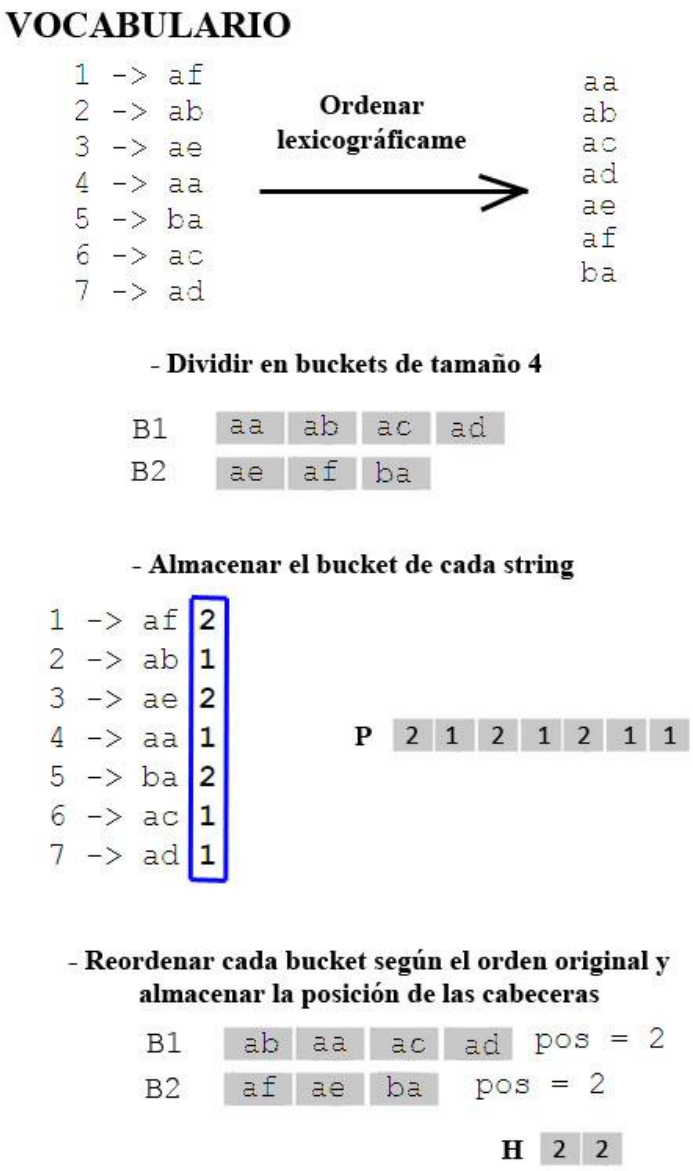

Figura 27 – Construcción RSD-A

Una vez terminado este proceso se tienen tres estructuras de datos distintas:

- El vocabulario (dividido en buckets según lo explicado) que se almacena directamente utilizando un DAC comprimido con Re-Pair (**RPDAC**).
- La permutación **P** que almacena el bucket de cada elemento y está ordenado según el orden original. Esta estructura se almacenará como un Wavelet Tree.
- El array H con la posición de las cabeceras, que se almacenará utilizando *log(b)* bits por cada elemento.

Utilizando el ejemplo anterior, el RSD-A quedaría según se puede observar en la figura 28, donde se muestran las 3 estructuras almacenadas y lo que representa cada una (el RPDAC se muestra en plano para facilitar la comprensión). Nótese que también debe conocerse el tamaño de los buckets para poder distinguir unos de otros en el RPDAC.

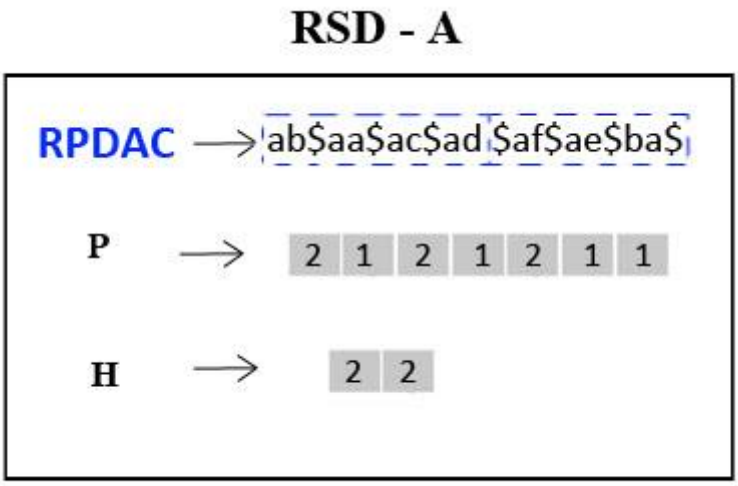

Figura 28 – Ranked String Dictionary Advanced

Con estas tres estructuras pueden realizarse tanto las operaciones básicas de *locate* y *extract* como operaciones por prefijos.

Para realizar ***locate(s)*** se siguen los siguientes pasos:

1. j = BusquedaBinaria(*s*) → Búsqueda del bucket que contiene *s*.
2. i = BusquedaSecuencial(*s*, *j*) → Búsqueda secuencial del string en el bucket.
3. id = P->select$_j$(*i*) → operación select sobre P para obtener el id final.

Como se puede observar, en el primer paso se realiza una búsqueda binaria por buckets para localizar el bucket (*j*) en el que se encuentra *s*, comparando el string *s* con la cabecera de los buckets. Una vez se ha localizado el bucket (*j*), éste se recorre de manera secuencial hasta localizar el string que coincida con *s*, obteniendo que se encuentra en la *i*-ésima posición del bucket. Una vez localizado el string en la posición *i* del bucket *j*, se obtiene el ID de ese elemento realizando select$_j$(*i*) sobre P. Si al recorrer el bucket no se encuentra ningún string que coincida con *s*, no se devuelve un valor nulo como resultado de *locate*(*s*). A continuación se muestra este proceso para resolver esta operación sobre el ejemplo anterior, buscando el resultado de *locate*("ac"):

- Se inicia la búsqueda binaria, obteniendo que la cabecera del primer bucket ("aa") es anterior al string buscado, y la cabecera del segundo bucket ("ae") es posterior, por lo que se deduce que el bucket donde se encuentra "ac" es el primero. *j*=1.
- Se recorre secuencialmente el bucket hasta localizar "ac" (si no se encontrara es que no existe el elemento en el diccionario). *i*=3.
- Se calcula sobre P select$_1$(3)=**6**, que se puede comprobar en la figura 27 que es el id original de "ac".

Para la resolución de ***extract(i)*** basta con seguir los siguientes pasos:

1. b = P[i] → Obtener el bucket.
2. x = P->rank$_j$(i) →Calcular la posición dentro del bucket haciendo rank en P.
3. RPDAC->extraer(b*(j-1)+x) → Se extrae el string del RPDAC.

Como se puede ver, primero se obtiene el bucket en el que se encuentra el elemento accediendo a P[*i*]=*j*. Una vez que se conoce el bucket se busca el número de ocurrencias de *j* hasta la posición *i* dentro de P, lo que nos proporcionará la posición del string dentro del bucket; esto se consigue calculando el valor x=rank$_j(i)$ en P. Con esto se puede obtener el string descomprimiendo el elemento $b*(j-1)+x$ en RPDAC. A continuación se muestra este proceso para resolver esta operación sobre el ejemplo anterior, resolviendo *extract*(5):

- j=P[5]=2, lo que indica que el string se encuentra en el segundo bucket.
- Calculamos sobre P x=rank2(5)=3, por lo que el string se encuentra en la tercera posición del segundo bucket.
- Se extrae de RPDAC el elemento en la posición 4*(2-1)+3=7, que es el string "**ba**", que como se comprueba en la figura 27 es el quinto elemento según el orden inicial.

Para la operación ***locatePrefix*(*s*)** se realiza según los siguientes pasos:

1. [*a*, *b*] = BusquedaBinaria(*s*) → búsqueda de los buckets que contienen el prefijo.
2. ∀*i* ∈ buket(*j*) | j ∈ [*a*, *b*] , if(*i*->contiene(*s*)) → id = P->select$_j(i)$ → Se recorren los buckets, y si el elemento contiene el prefijo se calcula su id.

Como se observa en el algoritmo anterior, lo primero que se realiza es una búsqueda binaria para localizar el primer y el último bucket que contienen el prefijo dado. Una vez se tiene el rango donde se localiza el prefijo, se recorren los buckets extrayendo el id de cada uno de los elementos, comprobando en el primer y último bucket si el elemento tiene el prefijo (en los buckets intermedios es seguro que todos los elementos contienen el prefijo.

Para la operación ***extractPrefix*(*s*)** se realiza de la misma manera que se realiza la operación de *locatePrefix*(*s*), con la diferencia de que en lugar de extraer el id, se extrae directamente el string.

Además de las dos operaciones de prefijos descritas, este diccionario permite la realización de otra operación de prefijos que tiene más sentido que las anteriores en ciertos ámbitos, y es la operación de **top_*k*(*s*)**, que extrae únicamente los *k* primeros strings que coincidan con el prefijo *s* dado. Esto permite reducir enormemente el tiempo de respuesta en tareas que únicamente requieran unos pocos resultados en grandes diccionarios, siendo el ejemplo más evidente el autocompletado en las búsquedas de los buscadores web.

A continuación se explicará el proceso para realizar la operación *locate-top_k*(*s*), que es similar a *locatePrefix*, pero devolviendo sólo los k primeros elementos. La operación de *extract-top_k*, se realizaría realizando *extract* sobre cada uno de los identificadores devueltos por la operación *locate-top_k*, por lo que no requiere de más explicación.

Para llevar a cabo la operación ***locate-top_k(s)*** se realiza lo explicado en el siguiente código:

1. [*x*, *y*] = BusquedaBinaria(*s*) → búsqueda de los buckets que contienen el prefijo.
2. *a* = siguienteIdBucketPrimero(*s*)
3. *b* = siguienteIdBucketUltimo(*s*)
4. *c* = primerIdBucketIntermedio(x-1,y-1)
5. for(int i=0; i<k; i++)
    a. resultado[i] = min(*a*, *b*, *c*)
    b. actualizarMenorId(*a*, *b*, *c*)

* actualizarMenorId(*a*,*b*,*c*) → Actualiza el menor de las ids que se le pasa como parámetro, llamando a "siguienteIdBucketPrimero(*s*)", "siguienteIdBucketUltimo(*s*)" o "siguienteIdBucketIntermedio()" según sea el id mínimo el primero, el último o el intermedio respectivamente.

Según se puede apreciar en el código anterior, se comienza obteniendo a través de una búsqueda binaria el rango de buckets que contienen el prefijo dado (al igual que en la operación de *locatePrefix*). Una vez se tiene el rango se calcula el primero de los ids del primer bucket que contiene el prefijo utilizando siguienteIdBucketPrimero(*s*) (operación que busca secuencialmente dentro del bucket el siguiente elemento con mismo prefijo y calcula su id), y a este valor le llamaremos *a*. Del mismo modo se calcula *c*, el primero de los ids del último bucket. Y mediante "primerBucketIntermedio()" se calcula el primer id de los buckets intermedios (*b*). A continuación se comprueba el menor de entre *a*, *b* y *c*, y ese menor se almacenará como resultado; si el menor es el del primer o el último bucket se busca el siguiente id del bucket de manera secuencial (comprobando que contenga el prefijo), y si es el del bucket intermedio se llama a la operación "siguienteIdBucketIntermedio()".

Las operaciones de primerBucketItermedio y siguienteBucketIntermedio son operaciones complejas, ya que requieren evaluar todos los buckets intermedios; no obstante, como cada bucket está ordenado internamente por el ranking, los valores se van encontrando en un recorrido izquierda-derecha de los buckets; para esto se propone una explicación exclusiva para dichas operaciones.

Si se recuerda la estructura del RSD-A, la permutación P se almacena utilizando un Wavelet Tree, donde cada hoja representa un bucket distinto. Para llevar a cabo las dos operaciones anteriormente mencionadas se creará un árbol auxiliar, que tendrá la misma forma que el subárbol del wavelet tree P que está comprendido entre los nodos hoja que representan al primer y al último buckets intermedios.

Para que la explicación sea más sencilla se utilizará un ejemplo distinto al anteriormente utilizado, y es el mostrado en la figura 29. Aunque en el ejemplo se utilizan strings tienen únicamente dos caracteres y no existiría compresión alguna con la técnica de Re-Pair, pero el ejemplo servirá para la comprensión de la creación del árbol auxiliar.

VOCABULARIO

1 -> ba
2 -> bf
3 -> bb
4 -> aa
5 -> bg
6 -> ac
7 -> be
8 -> ab
9 -> cb
10 -> bc
11 -> ca
12 -> da
13 -> bd
14 -> gb
15 -> cd
16 -> ga
17 -> db
18 -> ha
19 -> bh

RDS - A

	1	2	3	4	5	6	7	8	9	10
RPDAC	aa$	ab$	ba$	ac$	bb$	bc$	be$	bd$	bf$	bg$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
P	2	5	3	1	5	2	4	1	7	3	6	8	4	9	7	9	8	10	6

	1	2	3	4	5	6	7	8	9	10
H	1	2	2	2	1	2	1	1	2	1

Figura 29 – Ejemplo 2 – RSD-A

De este ejemplo, el Wavelet Tree que representa a P es el mostrado en la figura 30.

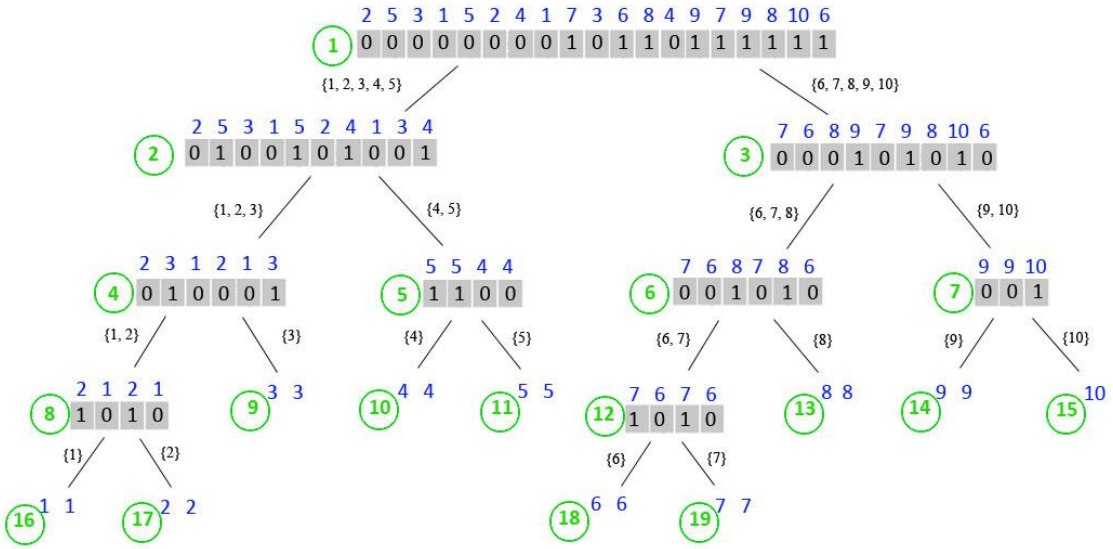

Figura 30 – Wavelet Tree “P”

Una vez que tenemos el ejemplo del diccionario, supongamos que se quiere realizar la operación top_8("b"), es decir, queremos los 8 primeros strings que comiencen por la letra "b". Para ellos realizamos la búsqueda binaria y localizamos que los buckets que tienen los strings que comienzan por "b", y son los del rango [2, 6]. Según lo descrito anteriormente, obtenemos el id correspondiente al primer elemento que comienza por "b" en el primer (*a*) y último bucket (*c*), realizando búsqueda secuencial, y se obtiene en este caso que *a*=1 y *c*=19. Los valores *a* y *c* (y el valor de *b* para los buckets intermedios) representan la posición en el ranking original. Como se ha mencionado antes, para obtener el identificador del primero de los strings de los buckets intermedios (*b*) se necesita un árbol auxiliar.

Para crear este árbol auxiliar se crea un árbol que represente al subárbol que contiene los nodos del intervalo [3, 5] (es decir, el árbol auxiliar representaría los nodos 2, 4, 5, 9, 10 y 11). Para cada uno de estos nodos se va a almacenar un valor al que llamaremos *ptr*, que comenzará siendo 1 en las hojas. Para el resto de los nodos se almacenarán también dos valores, I (izquierda) y D (Derecha), donde I=select$_0$(hijoIzquierdo->*ptr*) y D=select1(hijoDerecho->*ptr*), realizando la operación de select sobre el nodo del Wavelet Tree al que represente ese nodo; una vez se tienen los valores I y D, el valor ptr de ese nodo será el mínimo de ambos (*ptr*=min(I, D)). En el ejemplo actual, este árbol quedaría como se muestra en el primero de los árboles representados en la figura 31 (debe notarse que si un nodo no tiene hijo derecho o hijo izquierdo, su valor I o D, respectivamente, se igualará a infinito). Una vez construido el árbol, nos situamos en el nodo raíz (al que llamaremos nodo pivote), e iremos bajando el árbol, yendo a la derecha si D<I o a la izquierda si I<D; bajamos de esta manera hasta llegar a una hoja, que indica el bucket en el que se encuentra el string buscado (*j*). Para obtener el id que buscamos se realiza **id=P->select$_j$(*ptr*)**, siendo *ptr* el valor del nodo hoja al que hemos llegado. Una vez que se ha obtenido el id que buscamos (con lo que ya tendríamos el valor *b* del top_k) se suma 1 al valor *ptr* del nodo hoja, y si el ptr supera el tamaño de los buckets del diccionario, se actualiza a infinito.

Observando la figura 31 se puede ver cómo se va actualizando el árbol según se van extrayendo los ids de los buckets intermedios. De esta manera podríamos obtener la solución a la operación top_8("b") que se quería resolver:

1. En un primer instante se calcular *b*=P->select$_5$(1)=2 (pues se tiene que el bucket buscado es el 5 y tiene valor *ptr*=1). Entonces se tiene *a*=**1**, *b*=2 y *c*=19; como el menor es *a*, se almacena como primer id y se actualiza su valor. Al no haber más elementos con el prefijo "b" en el bucket se actualiza su valor a infinito.
2. Ahora se tiene *a*=∞, *b*=**2** y *c*=19; como el menor es *b*, se almacena como segundo resultado y se actualiza el árbol, y según se observa en el segundo paso de la figura 30, se extrae el primer elemento del bucket 3: *b* = P->select$_3$(1)=3.
3. Tenemos *a*=∞, *b*=**3** y *c*=19; de nuevo el menor es *b*, y al actualizar el árbol (obteniendo el paso 3 de la figura 31) se tiene *b* = P->select$_5$(2)=5.
4. Después se tiene *a*=∞, *b*=**5** y *c*=19; de nuevo el menor es *b*, se almacena y se actualiza el árbol (obteniendo el paso 4 de la figura 31) *b* = select$_4$(1)=7. Se puede observar en la figura que el nodo del bucket 5 ha pasado a ser infinito, pues al actualizarse cuando ptr ya era el tamaño del bucket se ha pasado a infinito.

5. Ahora tenemos $a=\infty$, $b=\mathbf{7}$ y $c=19$; de nuevo el menor es *b*, se almacena y se actualiza el árbol (obteniendo el paso 5 de la figura 31) $b = select_3(2)=10$.
6. Ahora tenemos $a=\infty$, $b=\mathbf{10}$ y $c=19$; de nuevo el menor es *b*, se almacena y se actualiza el árbol (obteniendo el paso 6 de la figura 31) $b = select_4(2)=13$. Ahora se puede ver que el nodo del bucket 4 también ha pasado a ser infinito al haber actualizado siendo *ptr*=2.
7. A continuación se tiene $a=\infty$, $b=\mathbf{13}$ y $c=19$; de nuevo el menor es *b*, se almacena y se actualiza el árbol, pero se tiene que en todos los nodos *ptr*=∞, por lo que se deduce que no existen más elementos en los buckets intermedios y se pasa a actualiza b=∞.
8. Finalmente tenemos $a=\infty$, $b=\infty$ y $c=\mathbf{19}$; en este paso no hay mucho donde elegir, ya que sólo se puede coger el valor de *c*, que se almacena. Al haber obtenido ya 8 identificadores no es necesario seguir con más pasos y se da por terminada la operación de *locate-top_8*(“b”).

Como resultado se tiene que ***locate-top_8*(“b”)={1, 2, 3, 5, 7, 10, 13, 19}**.

Es de notar que si en lugar de k=8 se hubiera elegido un k superior no variaría el resultado, pues después del paso 8 descrito anteriormente se actualizaría *c*, y se obtendría un valor infinito para esta variable; por lo tanto tendríamos $a=\infty$, $b=\infty$ y $c=\infty$, no pudiendo recuperar ningún valor más y dando por acabada la operación *locate-top_k*(“b”).

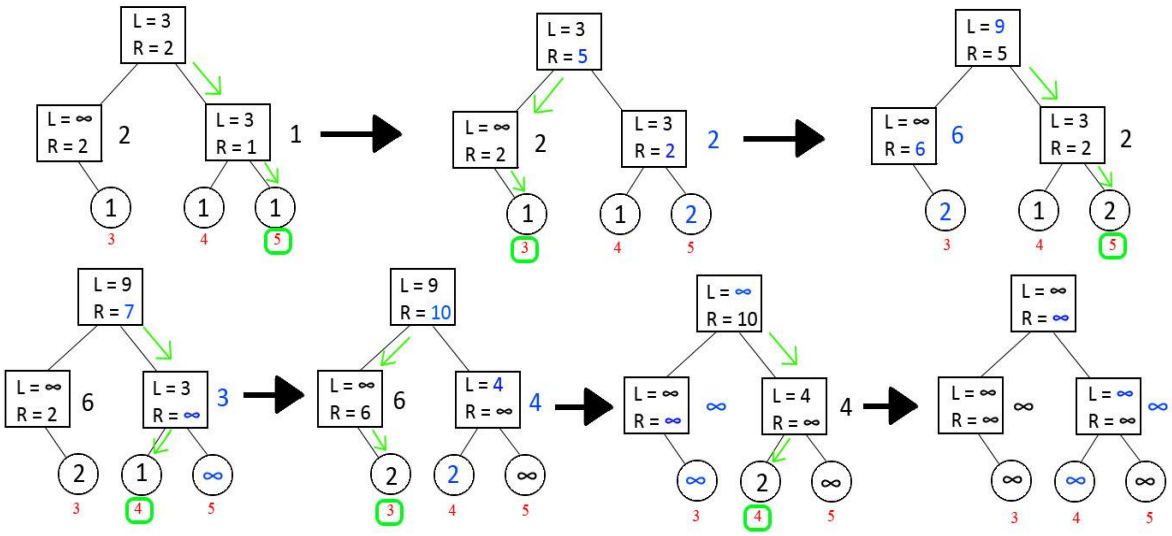

Figura 31 – Proceso árbol auxiliar top_k

Aunque en el ejemplo anterior sólo se han realizado 3 extracciones del árbol auxiliar, en la figura 31 se muestra todo el proceso hasta que no existen más ids para extraer. Si se llega el caso en que esto sucede, se le dará a b el valor infinito.

En el caso en que se llegue a la situación $a=\infty$, $b=\infty$ y $c=\infty$ se detendrá la ejecución de top_k, y en lugar de devolver k elementos se devolverán únicamente los que ya se hayan extraído.

CAPÍTULO 5
EVALUACIÓN EXPERIMENTAL

En este último capítulo se van a llevar a cabo experimentos para comprobar el rendimiento de los diccionarios desarrollados, comparándolos con los ya existentes (descritos en el capítulo 3), de manera que se pueda comparar un diccionario ordenado con otro ya existente que sea similar.

Antes de comenzar con los resultados de las pruebas realizadas se realizará un pequeño estudio acerca del entorno de experimentación (datos sobre la máquina sobre la cual se han llevado a cabo las pruebas) y los conjuntos de datos utilizados, así como una descripción sobre las pruebas que se han llevado a cabo.

5.1. Entorno de experimentación

Las pruebas se han llevado a cabo en una máquina virtual con las siguientes características técnicas: un procesador AMD a 2'3GHz; una memoria RAM de48GB; un disco duro de 1TB, de los cuales se disponía de 400GB libres; y un sistema operativo Ubuntu 14.04.4 LTS de 64 bits.

En cuanto a los conjuntos de datos utilizados para crear los diccionarios, se han tenido en cuenta 4 distintos:

- **Geonames**: contiene los diferentes nombres de todos los puntos geográficos. Este conjunto de datos está compuesto por 5.455.164 nombres geográficos y ocupa un espacio de almacenamiento de 81'62MB.
- **ADN**: contiene todas las subsecuencias de 12 nucleótidos encontrados en las secuencias de *S. Paradoxus* [24]. Contiene 9.202.863 subsecuencias y ocupa 114'09MB.
- **Word sequences**: consiste en un conjunto de palabras y frases obtenidas en un recopilatorio de traducción de inglés-español. Se han escogido las palabras y frases en español para la creación de los diccionarios. Este conjunto de datos está formado por 39.180.899 secuencias de palabras distintas, ocupando un total de 1127'87MB.
- **URLs**: consiste en un conjunto de URLs extraídas de un crawl de 2002 del dominio .uk. Contiene 18.520.486 URLs distintas con un tamaño de almacenamiento de 1372'06MB.

Las características técnicas de estos conjuntos de datos se muestran resumidos en la tabla 12 a continuación, donde se indica el nombre del conjunto de datos, el tamaño en almacenamiento en MB, el número total de strings que contiene, la longitud media de los strings (incluyendo el carácter especial de finalización $), el número de caracteres que utiliza (σ) y la entropía de orden 0 (H_0).

Diccionario	**Tamaño (MB)**	**Strings**	**Longitud media**	**σ**	**H_0**
Geonames	81'62	5455164	15'69	123	4'96
ADN	114'09	9202863	13	6	2'27
Word sequences	1127'87	39180899	30'18	138	4'35
URLs	1372'06	18520486	77'68	101	5'29

Tabla 12 – Descripción de los conjuntos de datos

5.2. Descripción de las pruebas

Para llevar a cabo las pruebas se han seleccionado únicamente algunos de los diccionarios desarrollados, sin la necesidad de tener que realizar pruebas sobre cada uno de ellos, ya que únicamente interesa la diferencia entre los diccionarios ordenados y los no ordenados. Los tipos de diccionario que se han utilizado son:

- Front-Coding: dentro de este tipo de diccionarios se han considerado los PFC, y los HTFC utilizando Re-Pair (RPHTFC); estos diccionarios se crearán tanto en la forma tradicional como siendo diccionarios contenidos dentro de un RSD-S. Para cada uno de estos diccionarios se considerarán distintos tamaños de bucket (4, 8, 16, 32, 64 y 128).
- Hash: Se utilizará el último de los diccionarios hash que se ha explicado, el HASHRPDAC, utilizando tres sobrecargas distintas (50%, 75% y 100% de sobrecarga).
- RSD-A: Por último se estudia también el comportamiento del último de los diccionarios que se ha desarrollado. También se considerarán distintos tamaños de bucket (4, 8, 16, 32, 64 y 128).

Se debe saber que los conjuntos de datos descritos en el apartado anterior originalmente están ordenados de manera lexicográfica (orden que no es de interés para los diccionarios ordenados), por lo que se ha realizado un desorden sobre cada uno de ellos para poder construir los diccionarios ordenados (para los diccionarios no ordenados se ha utilizado los datos ordenados lexicográficamente). Para **desordenar** los datos se ha construido un diccionario Hash (distinto a HASHRPDAC) con los datos ordenados lexicográficamente, y posteriormente se han realizado *extracts* sobre el diccionario creado (y puesto que el diccionario Hash utiliza un orden distinto al lexicográfico se tiene un orden "aleatorio").

Para realizar las pruebas se han creado para cada conjunto de datos 5 ficheros distintos: uno con 1.000.000 de strings del conjunto de datos para realizar *locate*, otro con 1.000.000 ids (entre 1 y el número de strings que se tienen) para realizar *extract*, y tres ficheros cada uno con 100.000 prefijos. Cada uno de los ficheros de los prefijos contiene prefijos de una longitud fija, siendo ésta el 40%, 60% y 80% de la longitud media de los strings del conjunto de datos.

Por último se debe saber que cada una de las pruebas que se han realizado se ha repetido 3 veces, utilizando como resultado final de la prueba la media aritmética de los tres resultados obtenidos. También se ha calculado el coeficiente de variación entre las 3 pruebas, repitiéndose la prueba si el coeficiente de variación era superior al 40%.

Se han realizado pruebas de *locate*, *extract*, *locatePrefix*, *extractPrefix* y *locateTop*_k. Los diccionarios front-coding realizan todas las operaciones menos *top_k*; los hash únicamente locate y extract; y RSD-A realiza todas las operaciones, siendo el único que puede realizar *top_k*.

5.3. Resultados experimentales

A continuación se muestran los resultados obtenidos en gráficas.

5.3.1. Locate

En la figura 32 mostrada a continuación se pueden observar los resultados obtenidos en la prueba de *locate* para los cuatro conjuntos de datos. Los resultados que se observan son el tiempo medio por consulta de *locate* (en microsegundos), mostrando también el porcentaje del tamaño original del conjunto de datos que tiene el diccionario. En las gráficas mostradas se pueden observar en el mismo color los diccionarios de mismo tipo, y en diferentes puntos según tengan mayor o menor tamaño de bucket o sobrecarga.

En esta figura 32 se puede observar que los mejores rendimientos son el de PFC y el HASHRPDAC, siendo mejor el HASHRPDAC al comprimir más y tener un rendimiento similar.

Aunque no se aprecia claramente en las gráficas, la diferencia entre los diccionarios básicos y los RSD-S del mismo tipo es de poco más de un microsegundo por consulta. Esta pérdida de rendimiento no es elevada y permite tener el diccionario ordenado, por lo que para esta operación parece un precio más que razonable por la ventaja de poder ordenar el diccionario de la manera que más nos convenga.

Además del tiempo añadido en cada consulta, se aprecia también que el coste en almacenamiento de la permutación en los RSD-S es notable, pudiendo llegar a ocupar hasta un 20% del tamaño original del conjunto de datos. Pero a pesar de esto se alcanzan niveles de compresión bastante buenos.

Es destacable también (aunque no tiene relación con los diccionario ordenados) que los diccionarios RPHTFC tienen una gran pérdida de rendimiento con tamaños de bucket superiores a 32 (incluyo con tamaño de bucket 32 el rendimiento empieza a ser peor), lo que incitaría a utilizar tamaños de bucket pequeños, pues aún con buckets pequeños consigue un muy buen nivel de compresión.

Por último mencionar que el RSD-A tiene un rendimiento bastante peor, ya que no está diseñado para la realización óptima de esta operación, tardando más que el resto de los diccionarios (aunque sigue manteniendo el orden de los pocos microsegundos por cada consulta).

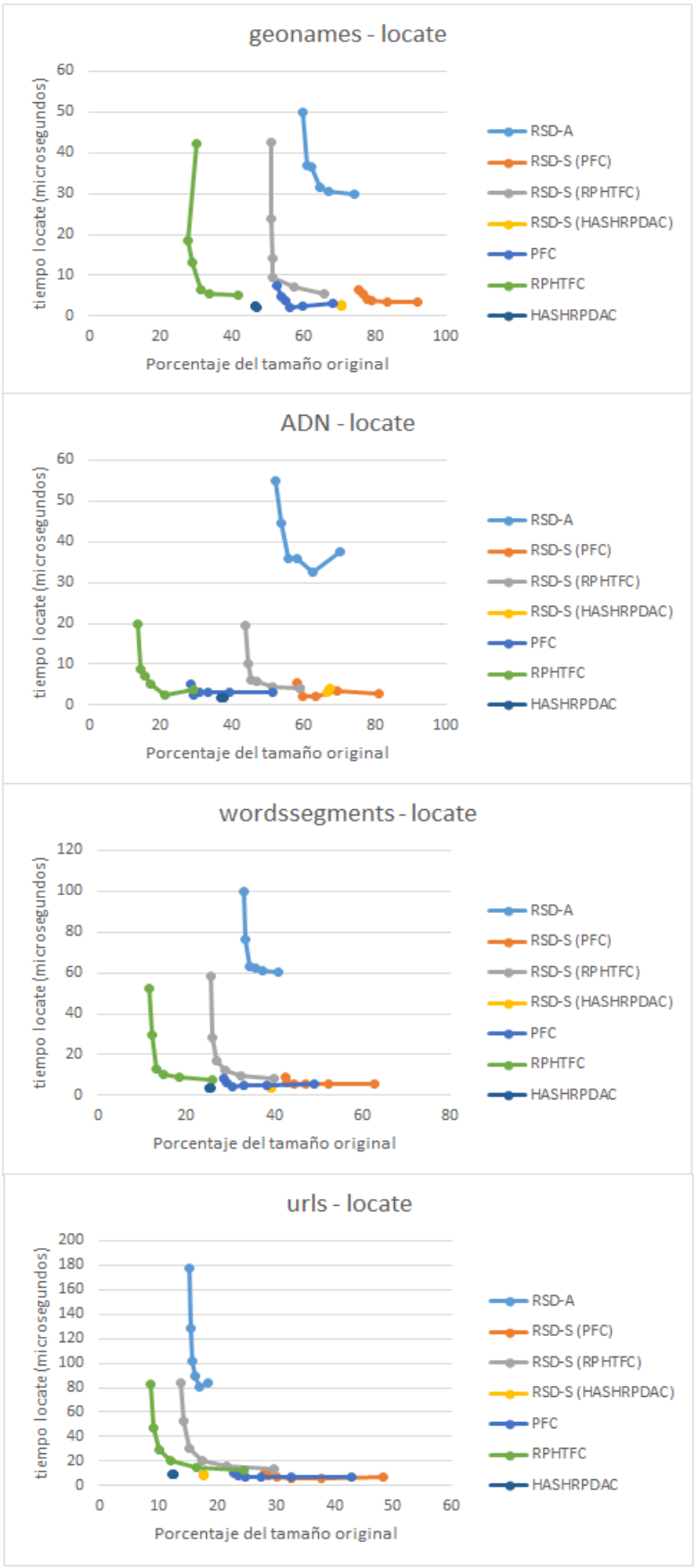

Figura 32 – locate

5.3.2. Extract

Para ver los resultados obtenidos en la prueba de extract, es necesario dirigirse a la figura 33. En esta figura aparecen las 4 gráficas correspondientes a cada uno de los conjuntos de datos, apareciendo en cada una los resultados de cada tipo de diccionario.

En esta ocasión no es necesario comentar demasiado la compresión que se obtiene con cada uno de los diccionarios ya que son los mismos que se tienen en la operación de locate, donde se explica por encima los niveles de compresión obtenidos.

De nuevo se repite la misma situación que en la operación *locate*, los mejores resultados en tiempo son para los PFC y HASHRPDAC, ya que estos diccionarios son óptimos para este tipo de operaciones, siendo PFC el más rápido de los front coding (aunque el que menos comprime) y HASHRPDAC de acceso prácticamente directo en operaciones de *locate* y *extract* (sin embargo no permite operaciones por prefijos ni por substrings).

De nuevo el diccionario RSD-A se encuentra entre los más lentos, únicamente superado en tiempo por los diccionarios RPHTFC con tamaños de bucket grandes. Sin embargo sigue encontrándose a unos niveles bastante aceptables, manteniéndose en el orden de los pocos microsegundos. El motivo de este rendimiento inferior al resto es el mismo que se ha explicado en las pruebas de la operación *locate*, y es que este diccionario no está optimizado para estas operaciones, se realizan de manera eficiente, pero no optimizadas.

También puede verse que, contrariamente a lo que sería de esperar, cuanto mayor es el tamaño de los buckets en este diccionario, a mayor velocidad realiza las consultas de *extract*. Seguramente a partir de un tamaño de bucket suficientemente grande empiece a empeorar el rendimiento, pero es un dato más que curioso.

De nuevo, los diccionarios RPHTFC sufren una gran pérdida de rendimiento en tamaños de bucket elevados, manteniendo un rendimiento muy bueno hasta tamaño 16 y empezando a dispararse los tiempos para tamaños de bucket 32 o superior. Sin embargo, no es necesario utilizar tamaños de bucket superiores a 16, ya que con buckets pequeños también alcanza un muy buen nivel de compresión.

Por último apreciar que los tiempos de respuesta en las consultas son inferiores a los obtenidos en *locate* (al menos en lo general) debido a que *extract* devuelve un string únicamente localizando la posición de éste, mientras que *locate* debe buscar un string que coincida con el dado como parámetro.

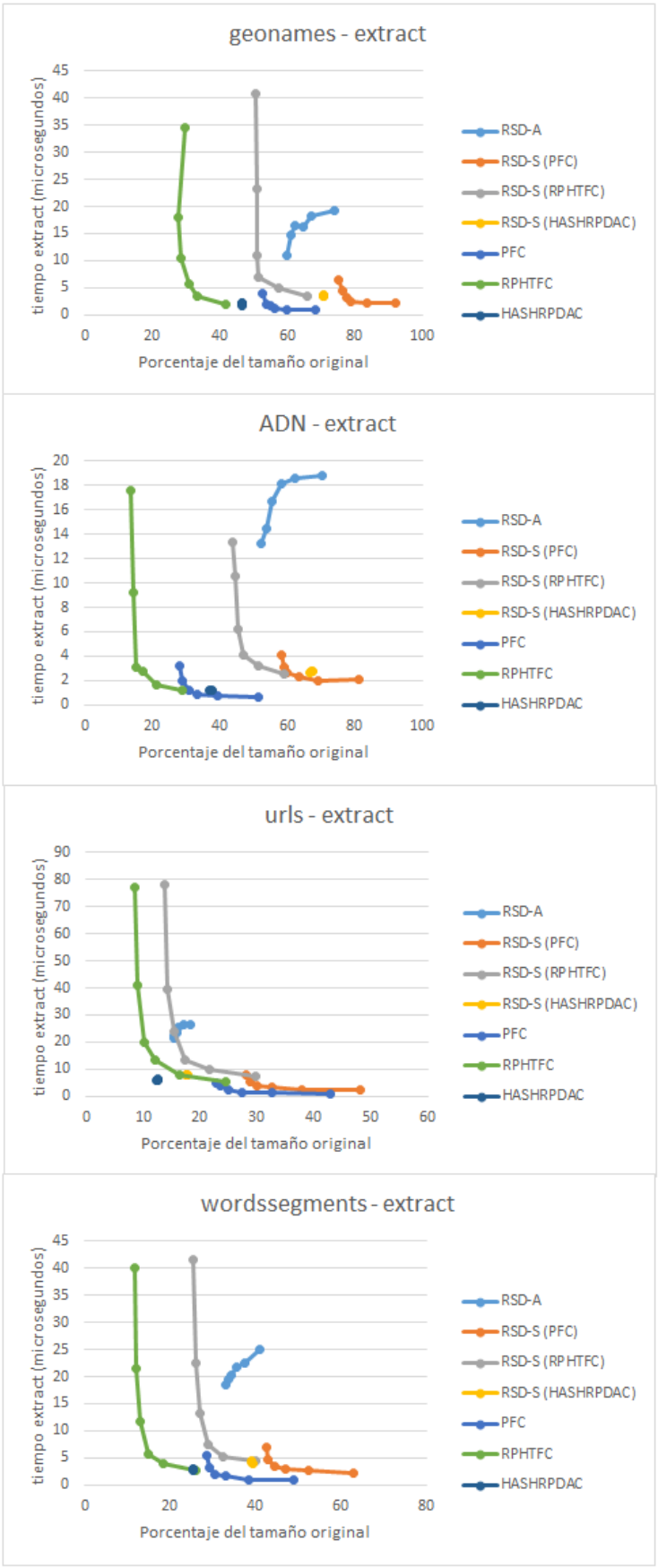

Figura 33 - extract

5.3.3. Prefijos

En las operaciones por prefijos se ha prescindido de los diccionarios basados en hash, ya que estos no permiten operaciones por prefijos (y por tanto RSD-S con diccionario hash como diccionario contenido tampoco permite estas operaciones).

Para estas operaciones se ha escogido un tamaño de bucket 8 para todos los diccionarios, pues es el tamaño considerado como óptimo, pues para todos los diccionarios tiene un buen nivel de compresión sin perder rendimiento. En las gráficas cada color representa un tipo de diccionario con las tres longitudes de prefijos utilizadas para las pruebas.

En las figuras 34 y 35 se pueden ver los resultados de las pruebas sobre operaciones por prefijos. En las gráficas situadas a la izquierda se muestra el tiempo que se ha necesitado de media para extraer cada uno de los ids (en nanosegundos), indicando en el eje x el número de strings que se han extraído por prefijo (de media). En las gráficas de la derecha se muestra el tiempo total necesario para llevar a cabo cada consulta (de media), mostrando en el eje x la longitud de los prefijos que se han utilizado.

Si nos fijamos en la figura 34 que muestra los resultados de *locatePrefix*, se puede observar que el resultado obtenido por RSD-A es mucho peor que el obtenido en el resto de diccionarios. Esto es debido a que este diccionario no está pensado para llevar a cabo esta operación, y la solución propuesta para resolver estas operaciones por prefijos (*locatePrefix* y *extractPrefix*) pasan por ejecutar un ordenamiento de todos los identificadores que se consiguen (pues en un principio se obtienen desordenados), siendo esta operación muy costosa y sumando un enorme tiempo en la resolución de la consulta. Es posible que exista alguna manera más optimizada de resolver esta consulta, pero al no ser de interés para este diccionario (pues la operación central de éste es *top-K*) no se ha contemplado optimizarla en este proyecto.

Siguiendo con *locatePrefix*, el resto de diccionarios tienen un rendimiento bastante similar al visto en las operaciones de *locate* y *extract*, siendo PFC el más rápido de todos. Sin embargo se aprecia que en esta ocasión el precio de la permutación en los diccionarios RSD-S es superior a uno o dos microsegundos por consulta; pero a pesar de esto no es mucho más y sigue siendo más que aceptable ese precio por poder mantener un diccionario ordenado.

En cuanto a la figura 35 que representa los resultados de *extractPrefix*, se aprecia que el diccionario RSD-A tiene un rendimiento bastante más aceptable, no teniendo un tiempo de respuesta muy superior al resto de diccionarios (a pesar de que la operación no está optimizada). Esto hace ver que el diccionario RSD-A tiene un rendimiento aceptable cuando lo que se quiere es el string y no el identificador (y por lo general lo que se desea obtener es el string).

Finalmente mencionar que el resto de diccionarios presentan un comportamiento muy similar al visto en la operación de *locatePrefix*; la única diferencia es que el tiempo es mayor, pero el rendimiento relativo entre ellos es el mismo.

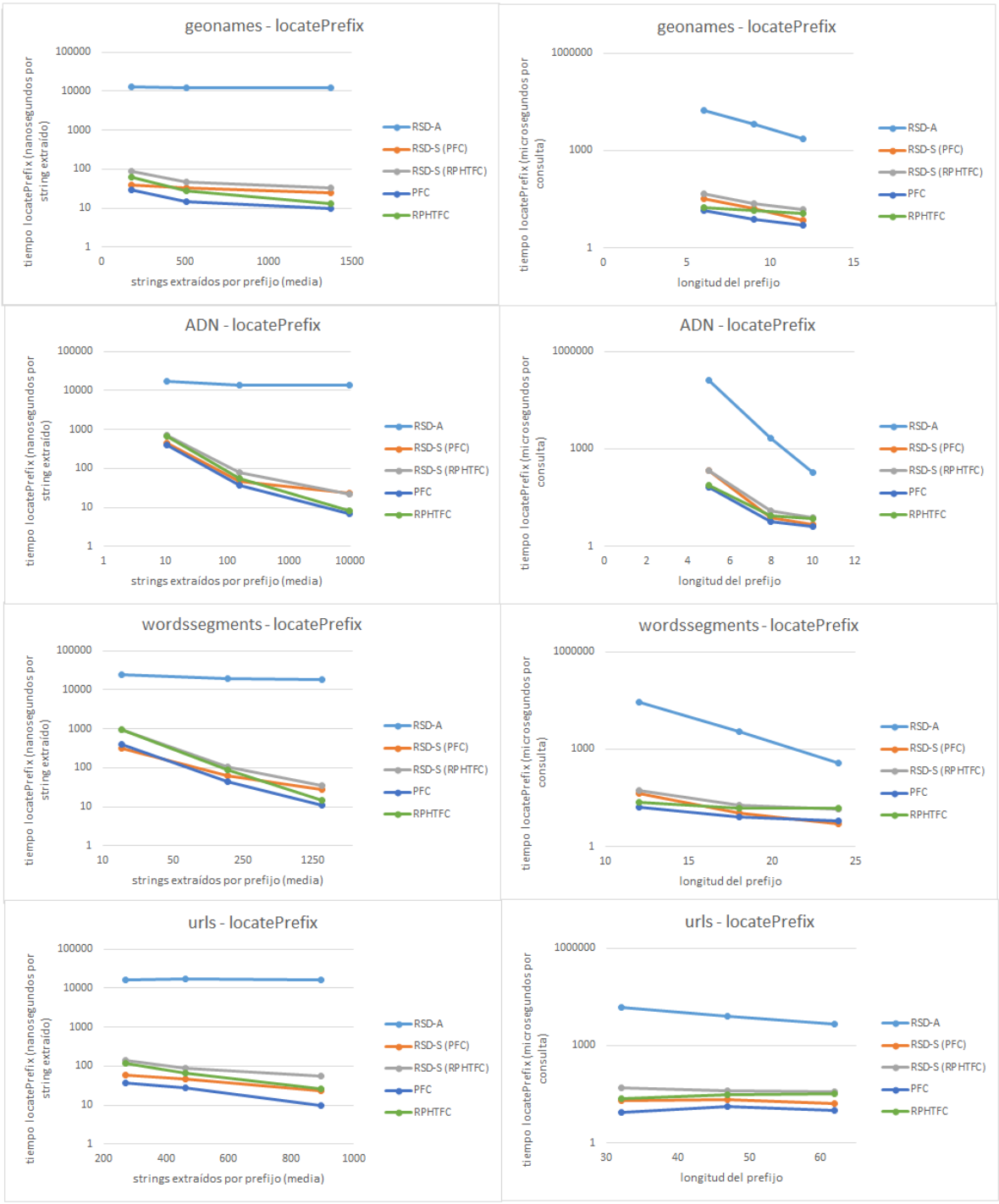

Figura 34 – locatePrefix

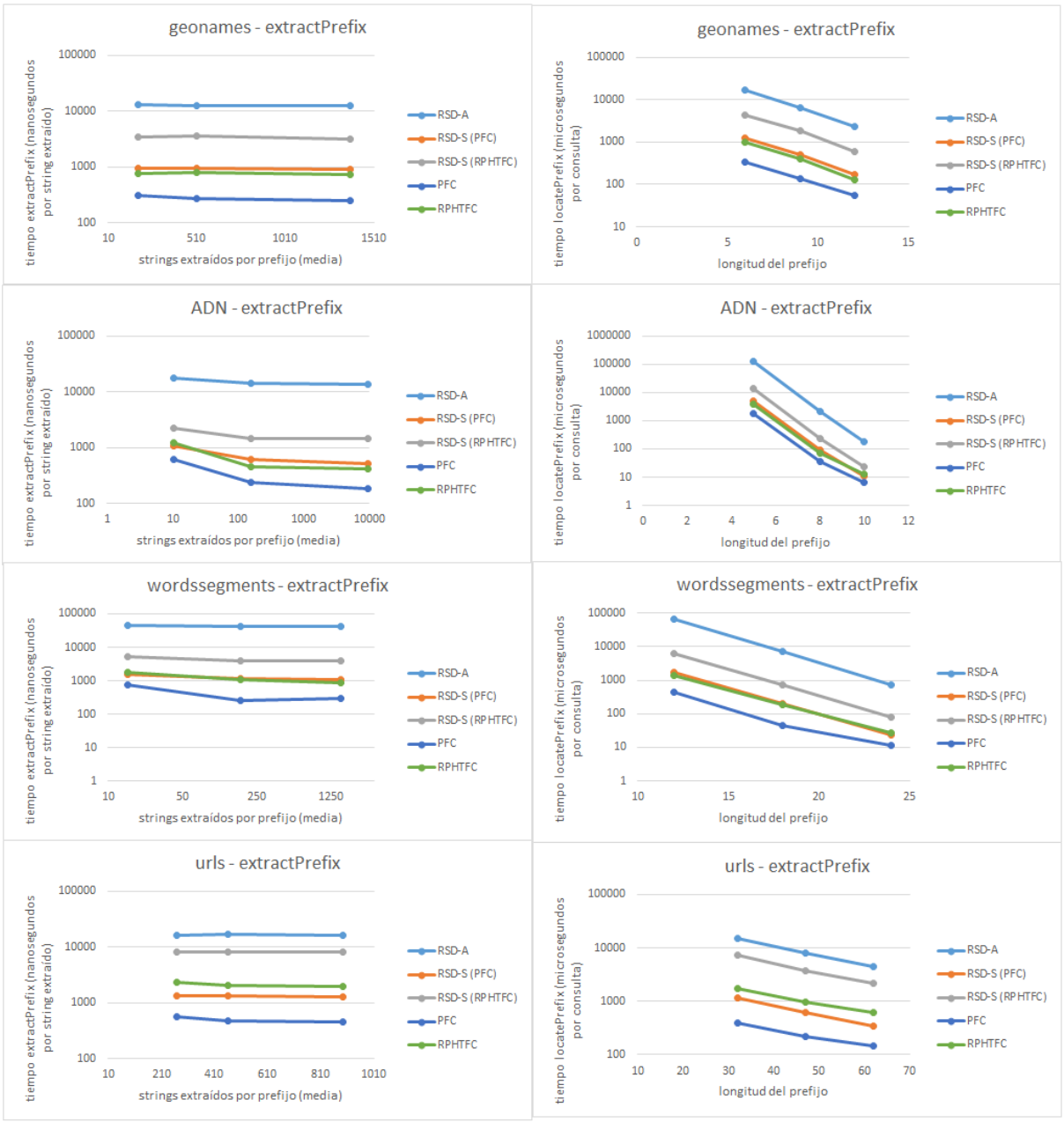

Figura 35 – extractPrefix

5.3.4. Top_k

Por último se han llevado a cabo pruebas en el diccionario RSD-A evaluando el rendimiento de la operación *top-K*, que es específica de este diccionario. Para ello se ha tenido en cuenta el tamaño de bucket 128, pues es el tamaño con el cual este diccionario ha presentado mejores rendimientos en operaciones por prefijos (y además al ser un tamaño de bucket grande se tiene un mejor nivel de compresión). Los resultados de estas pruebas se muestran en las figuras 36 y 37, siendo la figura 36 la correspondiente a *locateTop-K* y la figura 37 a *extractTop-K*.

Las gráficas de estas figuras son como las mostradas en las operaciones por prefijos: en la izquierda se muestra el tiempo medio en extraer cada string o identificador (en microsegundos), además, en el eje x se muestran los strings extraídos por cada operación de prefijo (de media); y en el lado derecho se muestra el tiempo medio de resolución de cada consulta (en microsegundos), y en el eje x la longitud de los prefijos con los que han lanzado las operaciones.

En cuanto a los resultados obtenidos, sería necesario compararlos con los vistos en las operaciones por prefijos. Si se compara con éstos, se puede observar que para la operación *locateTop-k* resulta más eficiente utilizar un diccionario RSD-S con la operación *locatePrefix* y devolver únicamente los k primeros resultados; esto hace pensar que quizá sea posible realizar alguna optimización más en las operaciones de *top-k*.

Sin embargo si lo deseado es obtener los strings y no los identificadores, se puede apreciar que es más rentable realizar las operaciones de *extractTop-k* a realizar un *extractPrefix* con otro diccionario (salvo cuando k es muy elevado y se devuelve un resultado similar a *extractPrefix*), siendo muy efectivo en situaciones en las que k no es muy elevado.

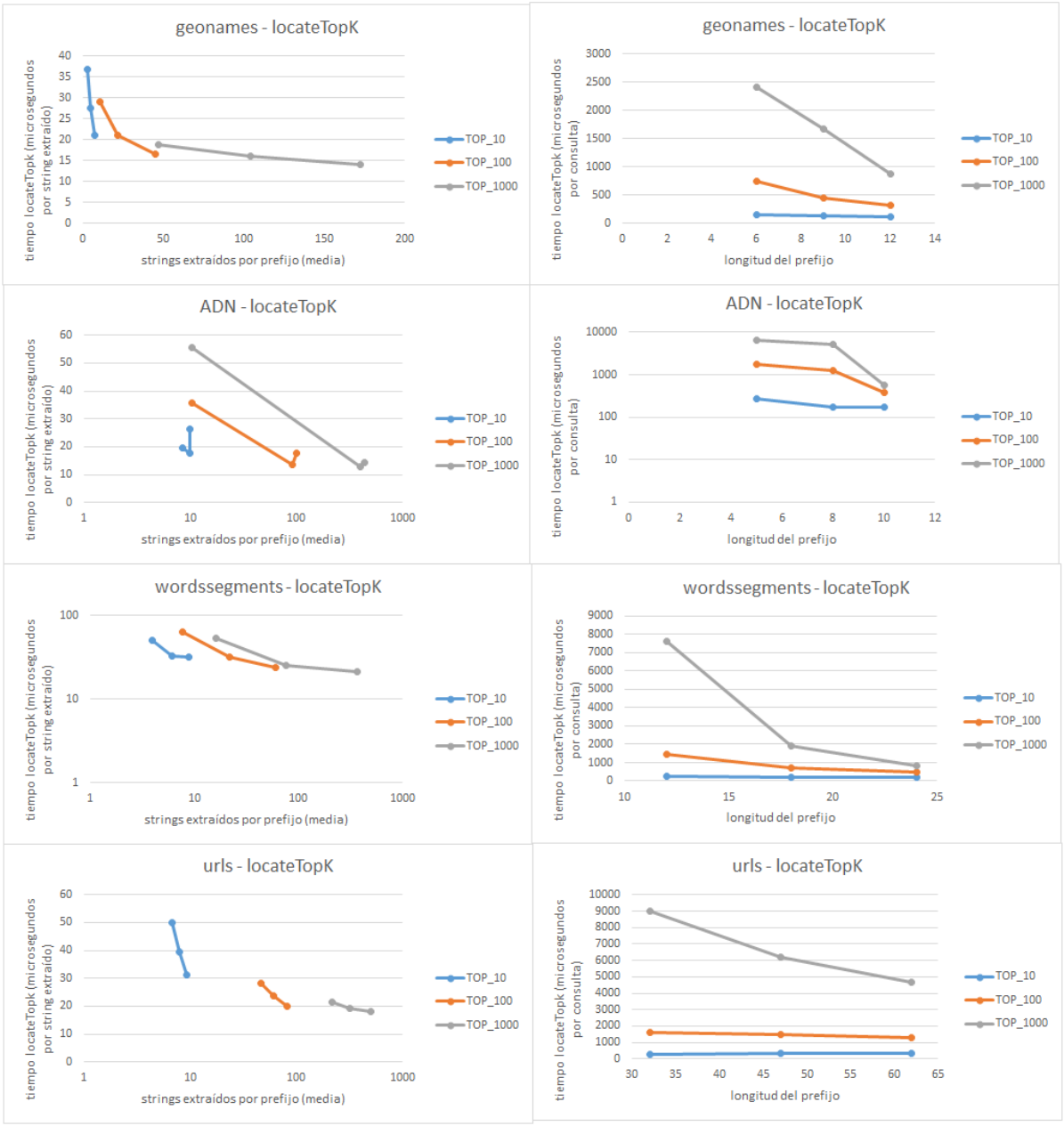

Figura 36 – locate top-K

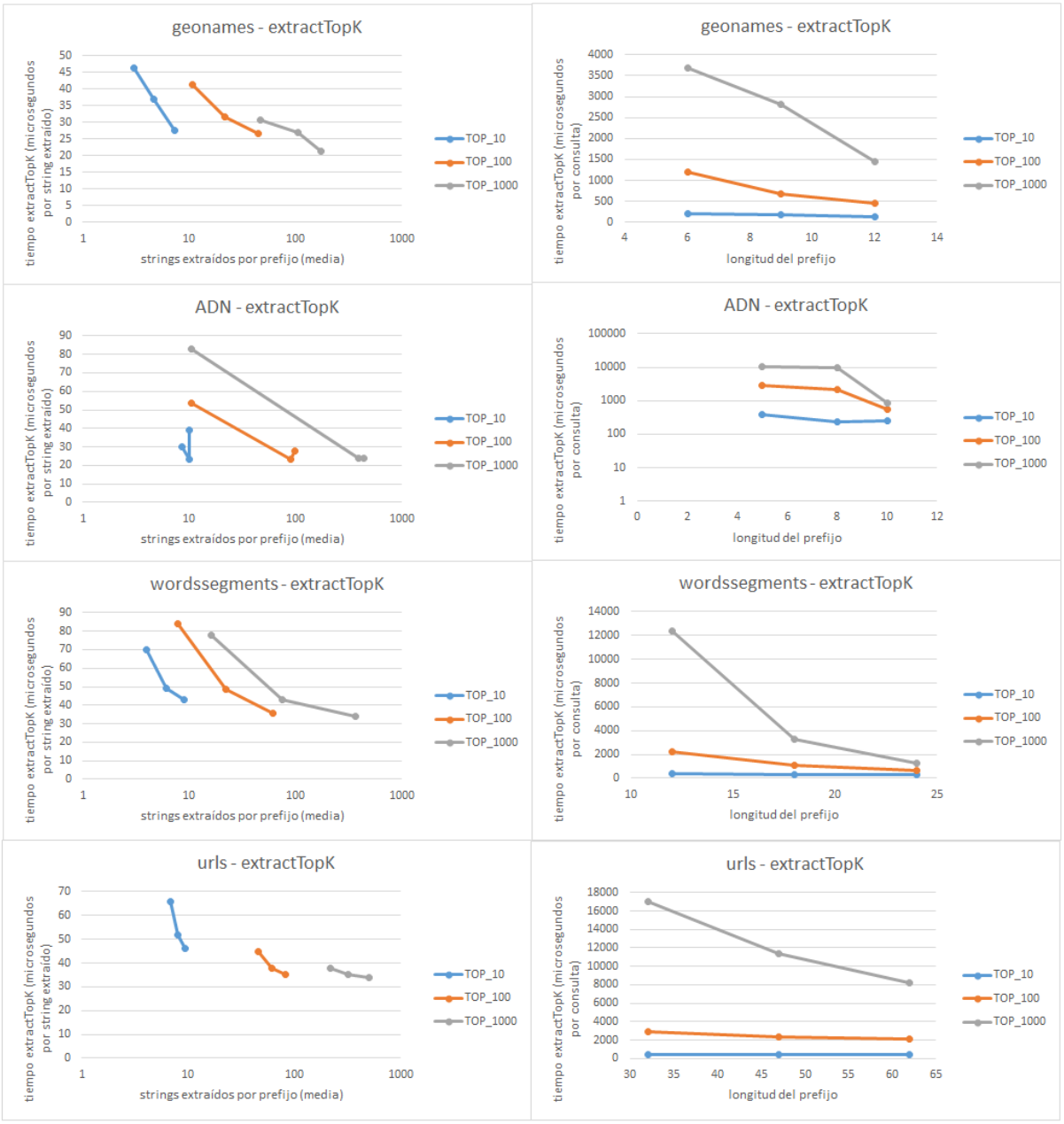

Figura 37 – extract top-K

5.4. Conclusiones

Con los resultados que se han obtenido se puede deducir, en primer lugar, que la solución sencilla de utilizar una permutación sobre un diccionario ya existente es una solución que prácticamente no añade tiempo de respuesta en operaciones y no supone una gran carga en almacenamiento. Por esto se puede concluir que el diccionario RSD-S es una perfecta solución para la gran mayoría de situaciones en las que se tiene un diccionario ordenado, ya que es muy versátil al poder contener cualquier tipo de diccionario.

Por otro lado se tiene que el diccionario RSD-A no tiene tan buenos resultados en la mayoría de operaciones (principalmente debido a que es un diccionario cuyo objetivo principal es la operación *top-k*), encontrándose entre los diccionarios más lentos en muchas de las pruebas, aunque con rendimiento más que aceptable. Sin embargo, en la operación de *extractTop-k* se ha podido comprobar que sí tiene una utilidad que puede ahorrar una gran cantidad de tiempo por consulta en aplicaciones que requieran únicamente una pequeña cantidad de resultados en búsquedas por prefijos (por ejemplo buscadores web, autocorrectores de smartphones,…).

Estos resultados dan pie a continuar en un futuro con el proyecto, comenzando por intentar optimizar las operaciones de *top-k*, pues es muy probable que existan maneras de mejorar la implementación de esta operación. Además sería de interés implementar operaciones de prefijos que no requieran reordenar todos los resultados encontrados antes de poder devolverlos, pudiendo mejorar así el rendimiento en las operaciones *locatePrefix* y *extractPrefix*. Por último se deberían realizar más pruebas sobre el diccionario RSD-A con tamaños de bucket mayores, para comprobar hasta qué punto mejora el rendimiento según aumente el tamaño de bucket (cosa que por lo general es al revés), y saber si en algún punto el rendimiento empieza a empeorar (lo cual sería lo lógico).

Una vez se hubiera realizado todo esto, que sería una fase de optimización y pruebas con mayor profundidad, sería de interés conocer cómo se comportan estos diccionarios ordenados en entornos reales, tales como buscadores web, tripleStores, aplicaciones de desambiguación semántica, etc.

REFERENCIAS

[1] E. Dumbil “What is big data? An introduction to the big data landscape”. 2012. Disponible en http://radar.oreilly.com/2012/01/what-is-big-data.html

[2] OBS Business School. “Big Data 2015”. 2015.

[3] P. Zikopoulos, C. Eaton, D. deRoos, T. Deutchs, G. Lapis. “Understanding Big Data”. 2011. Disponible en https://www.ibm.com/developerworks/vn/library/contest/dw-freebooks/Tim_Hieu_Big_Data/Understanding_BigData.PDF

[4] “Tablas IRPF 2016 y retenciones”. Recuperado 19/03/2016 http://www.irpf.eu/2016.html.

[5] “Seguridad social - Trabajadores”. Recuperado 19/03/2016 http://www.seg-social.es/Internet_1/Trabajadores/CotizacionRecaudaci10777/Basesytiposdecotiza36537/index.htm.

[6] D.A. Huffman. “A Method for the Construction of Minimum-Redundancy Codes”. Proceedings of the IRE, 40(9):1098–1101. 1952. Disponible en http://compression.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf.

[7] J. Ziv and A. Lempel. “A Universal Algorithm for Sequential Data Compression”. IEEE Transactions on Information Theory, 23(3):337–343. 1977. Disponible en https://www.cs.duke.edu/courses/spring03/cps296.5/papers/ziv_lempel_1977_universal_algorithm.pdf.

[8] J. Ziv and A. Lempel. “Compression of Individual Sequences Via Variable-Rate Coding”. IEEE Transactions on Information Theory, 24(5):530–536. 1978. Disponible en http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.14.2892&rep=rep1&type=pdf.

[9] N.J Larsson and A. Moffat. “Offline Dictionary-Based Compression”. Proceedings of the IEEE, 88(11):1722–1732. 2000. Disponible en http://www.larsson.dogma.net/dcc99.pdf.

[10] M. Burrows and D. Wheeler. “A block sorting lossless data compression algorithm”. Technical Report 124, Digital Equipment Corporation. 1994. Disponible en http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf.

[11] “Internetlivestats”. Recuperado 19/05/2016. http://www.internetlivestats.com/.

[12] G. Navarro and V. Mäkinen. “Compressed full-text indexes”. ACM Computing Surveys, 39(1):article 2. 2007. Disponible en http://www.dcc.uchile.cl/TR/2006/TR_DCC-2006-006.pdf.

[13] R. González, S. Grabowski, V. Mäkinen, G. Navarro. “Practical implementation of rank and select queries”. In Poster Proc. of WEA, 27–38. 2005. Disponible en http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.144.383.

[14] F. Claude, G. Navarro, and A. Ordóñez. “The wavelet matrix: An efficient wavelet tree for large alphabets”. Information Systems, 47:15–32. 2015. Disponible en http://lbd.udc.es/Repository/Publications/Drafts/1420793866417_is14.pdf.

[15] A. Golynski, J.I. Munro, and S.S. Rao. “Rank/select operations on large alphabets: A tool for text indexing”. In Proc. of SODA, pages 368–373. 2006. Disponible en http://dl.acm.org/citation.cfm?id=1109599.

[16] J. Barbay, F. Claude, T. Gagie, G. Navarro, and Y. Nekrich. “Efficient fully-compressed sequence representations”. Algorithmica, 69(1):232–268. 2014. Disponible en http://www.dcc.uchile.cl/~gnavarro/ps/algor13.pdf.

[17] N. Brisaboa, S. Ladra, and G. Navarro. “DACs: Bringing direct access to variable-length codes”. Information Processing and Management, 49(1):392–404. 2013. Disponible en http://www.dcc.uchile.cl/~gnavarro/ps/ipm12.pdf.

[18] P. Ferragina and G. Manzini. “Indexing compressed texts.” Journal of the ACM, 52(4):552–581. 2005. Disponible en http://web.cs.ucdavis.edu/~gusfield/cs224f11/ferragina2005.pdf.

[19] M. A. Martínez-Prieto, N. Brisaboa, R. Cánovas, F. Claude, and G. Navarro. “Practical compressed string dictionaries”. Information Systems, 56:73–108. 2016. Disponible en http://repositorio.uchile.cl/bitstream/handle/2250/138288/Practical-compressed-string-dictionaries.pdf?sequence=1.

[20] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. “Introduction to Algorithms”. MIT Press and McGraw-Hill, 2nd edition. 2001. Disponible en http://users.dcc.uchile.cl/~raparede/clases/iet121/cormen.pdf.

[21] I.H. Witten, A. Moffat, and T.C. Bell. “Managing Gigabytes: Compressing and Indexing Documents and Images”. Morgan Kaufmann. 1999.

[22] H.E. Williams and J. Zobel. “Compressing integers for fast file access”. The Computer Journal, 42:193–201. 1999. Disponible en http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.18.3782&rep=rep1&type=pdf.

[23] D.E. Knuth. “The Art of Computer Programming, volume 3: Sorting and Searching”. Addison Wesley. 1973.

[24] “Microbe Wiki - Saccharomyces paradoxus”. Recuperado 20/06/2016 https://microbewiki.kenyon.edu/index.php/Saccharomyces_paradoxus.