UNIVERSIDAD MICHOACANA DE
SAN NICOLÁS DE HIDALGO

FACULTAD DE INGENIERÍA ELÉCTRICA
DIVISIÓN DE ESTUDIOS DE POSGRADO

COMPRESIÓN DE BASES DE DATOS MÉTRICAS

TESIS

QUE PARA OBTENER EL GRADO DE:
MAESTRÍA EN CIENCIAS EN INGENIERÍA ELÉCTRICA

PRESENTA:
FABIOLA ARACELI VELÁZQUEZ AYALA

DIRECTOR DE TESIS:
DR. EDGAR LEONEL CHÁVEZ GONZÁLEZ

AGOSTO 2013

COMPRESIÓN DE BASES DE DATOS MÉTRICAS

Los Miembros del Jurado de Examen de Grado aprueban
la **Tesis de Maestría en Ciencias en Ingeniería Eléctrica** de ***Fabiola Araceli Velázquez Ayala***

Dr. Félix Calderón Solorio
Presidente del Jurado

Dr. Edgar Leonel Chávez González
Director de Tesis

Dr. José Antonio Camarena Ibarrola
Vocal

Dr. Mario Graff Guerrero
Vocal

Dr. Eric Sadit Téllez Ávila
Revisor Externo

Dr. J. Aurelio Medina Ríos
Jefe de la División de Estudios de Posgrado
de la Facultad de Ingeniería Eléctrica. UMSNH
(Por reconocimiento de firmas)

UNIVERSIDAD MICHOACANA DE SAN NICOLÁS DE HIDALGO
Abril 2013

COMPRESIÓN DE BASES DE DATOS MÉTRICAS

TESIS

Que para obtener el grado de

MAESTRÍA EN CIENCIAS EN INGENIERÍA ELÉCTRICA

presenta

Fabiola Araceli Velázquez Ayala

Dr. Edgar Leonel Chávez González

Director de Tesis

Universidad Michoacana de San Nicolás de Hidalgo

Agosto 2013

Resumen

Actualmente la mayoría de los sistemas de información requieren realizar búsquedas por similitud en repositorios de datos que contienen archivos multimedia y de texto. Utilizando un *modelo de bases de datos métricas* las aplicaciones pueden realizar consultas por similitud sobre estos repositorios, mediante la construcción de índices métricos. Una base de datos métrica es una colección de objetos digitales (de cualquier tipo), con una similaridad percibida entre estos objetos y donde tal similaridad es computada como una métrica [Chávez y Navarro, 2006]. A su vez, con la demanda de este tipo de aplicaciones se generan requerimientos de memoria para el almacenamiento de la base de datos métrica.

En el presente trabajo se propone un modelo de **compresión de bases de datos métricas**, entendiendo por compresión a la manera de representar un volumen de información utilizando un espacio menor en un dispositivo de almacenamiento y lograr un acceso más rápido a la misma, cuando esta representación puede residir en memoria principal.

El bosquejo del procedimiento básico es como sigue: primeramente se crea un espacio métrico sobre una colección o base de datos. Posteriormente los elementos del espacio métrico son organizados en grupos de elementos similares y en cada grupo se nombra a un elemento como representante, por medio de un método de partición que genera un índice métrico. Usando una métrica de distancia, la compresión de la base de datos se realiza codificando para cada zona o grupo al elemento representante de manera explícita y al resto de los elementos del grupo de manera implícita creando **secuencias de bits** (*bitmaps*) para codificar la distancia de cada elemento al correspondiente representante de su grupo. Utilizando **estructuras sucintas** se comprime el espacio de estos bitmaps, dichas estructuras permiten el acceso y manipulación de los bitmaps para descomprimir cada elemento implícitamente codificado. La representación que se propone, permite que los elementos codificados implícitamente sean decodificados sin pérdida de información. Bajo el esquema propuesto se puede evaluar la distancia de dos objetos arbitrarios sin la necesidad de descomprimir completamente la base de datos, esto permite que se pueda realizar una consulta sobre la base de datos comprimida descomprimiendo únicamente el grupo o grupos de interés donde el rango de alguna consulta intersecte.

En los experimentos realizados se obtuvo un porcentaje de compresión alrededor del 60 % con respecto al tamaño original de la base de datos métrica, realizando las búsquedas por similitud en un tiempo aproximadamente 10 veces mayor al tiempo de búsqueda sobre la base de datos sin compresión. Estos experimentos y sus resultados son expuestos con la finalidad de evaluar el esquema utilizado.

Abstract

Currently, most information systems need similarity search in data repositories containing text and multimedia files. Using a *metric database model* applications can perform queries by similarity over these repositories, by building metric index. A metric database is a collection of digital objects (of any type) with a perceived similarity between these objects and where such similarity is computed as a metric [Chávez y Navarro, 2006]. In turn, the demand for this type of application generate memory requirements for storing the metrics database.

In this document we propose a *metric database lossless compression model*, understanding compression as a way to represent the volume of information using less space on a storage device and achieve faster access to the same when this representation can reside in main memory.

The the basic procedure is as follows: first a metric space its created over a collection or database. By a partition method metric space elements area organized in groups of similar elements where each group as a representative element getting an index. Using a metric distance, the new database representation is made by a explicit representation of the representative in each group and the implicit representation of the rest of the elements trough **bitmaps** which coding the distance between each element and its representative. The space of this bitmaps is compressed using **succinct data structures** such structures allow access and manipulation of bitmaps to decode each element implicitly coded. The proposed representation enables that implicitly coded elements are decoded without loss of information. Under this scheme the distances between two arbitrary objects can be evaluated without the need to decompress the database completely, it permits to perform a query on compressed database and decompress only group or groups where the range query intersects.

The percentage of compression obtained in experiments is around 60 % and the search time is around 10 times over the search time in a original representation of the experiment database. The perform experiments and its results are exposed in order to evaluate the scheme used.

Contenido

Lista de Figuras

Lista de Tablas

Lista de Algoritmos

Lista de Símbolos

H	Entropía
$H_0(B)$	Entropía de orden cero
X	Fuente de información
X_c	Fuente de información comprimida
r	Razón de compresión
p	Porcentaje de compresión
$\mathbb{U}$	Universo válido de objetos
$\mathbb{S}$	Base de datos subconjunto de $\mathbb{U}$
n	Cardinalidad del subconjunto $\mathbb{S}$
$d(x, y)$	Función distancia entre dos elementos x, y.
$(\mathbb{U}, d)$	Espacio métrico
L_p	Familia de distancias L_p
L_1	Distancia de Manhatan
L_2	Distancia Euclidiana
L_∞	Distancia cuando p tiende a infinito
(q, r)	Consulta por rango
r	Radio de consulta o cobertura
I	Grupo de elementos en $\mathbb{S}$
$\vec{c}$	Elemento centro en $\mathbb{S}$
E	Elementos no centro $\mathbb{S}$
$cov(\vec{c})$	Cobertura de un centro
B	Secuencia de bits

Capítulo 1

Introducción

Con el avance de las tecnologías de información nos enfrentamos a una cantidad enorme de datos a administrar. Los datos que se utilizan en algunos sistemas de información pueden organizarse de diferentes formas, por ejemplo; llamamos modelo de información estructurada a los datos dispuestos mediante campos, usando este modelo, los *Sistemas de Gestión de Bases de Datos*, dividen la base de datos en registros o tuplas, donde cada registro tiene un campo clave. Las consultas a estas bases de datos devuelven todos los registros donde la clave coincide exactamente con la clave de búsqueda.

Existen bases de datos que tratan archivos multimedia [Chávez y Navarro, 2006]. Dichos archivos no pueden ser estructurados bajo registros o llaves cuando se requiere consultar su contenido o sus características y compararlo con otro archivo de consulta. La búsqueda por similitud soluciona este problema, utilizando este modelo de información, es posible, mediante una consulta por similitud encontrar el o los elementos más parecido(s) a una consulta dada. Para este tipo de sistemas y aplicaciones se utiliza el modelo de **Bases de Datos Métricas**.

Una *base de datos métrica* es una colección de objetos digitales (de cualquier tipo) con una similaridad percibida entre estos objetos y donde tal similaridad es computada como una métrica [Chávez y Navarro, 2006]. Muchos sistemas utilizan este modelo, por

ejemplo: *Google Images, Shazam, Amazon*, que haciendo uso de herramientas bajo este modelo realizan recuperación de información (*Information Retrieval*) y son capaces de encontrar similitudes, diferencias o patrones entre los elementos de sus bases de datos.

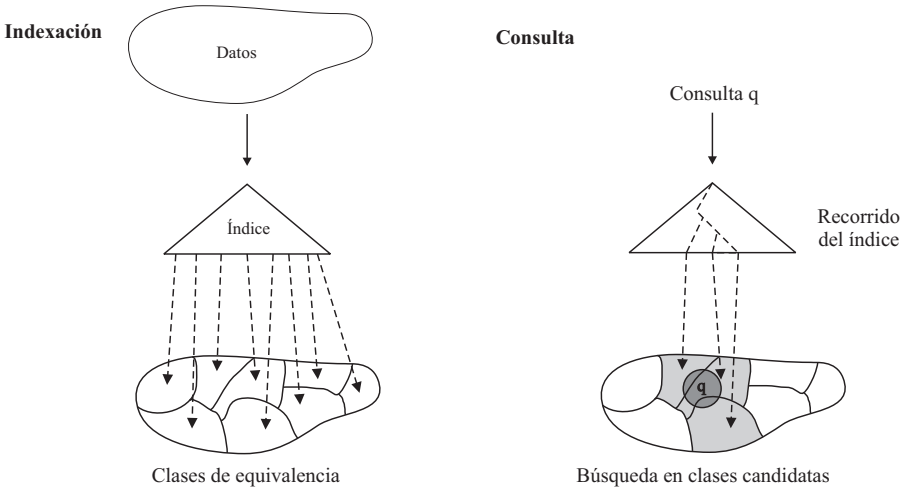


Figura 1.1: Modelo unificado para indexar y consultar en espacios métricos

En la figura 1.1 se representa el modelo general de indexación y búsqueda en espacios métricos, primeramente un algoritmo de indexación particiona el conjunto de datos $\mathbb{S}$ en subconjuntos creando una estructura de datos a la que llamamos *índice* este permite determinar un conjunto de subconjuntos candidatos donde pueden aparecer elementos relevantes a una consulta dada. Al momento de hacer una consulta, primero se busca en el índice para encontrar los subconjuntos relevantes a la consulta, posteriormente estos subconjuntos son revisados exhaustivamente [Chávez et al., 001b].

1.1. Planteamiento del Problema

Los costos de almacenamiento de información se han reducido significativamente en los últimos años, esto permite el uso de grandes colecciones de datos en los sistemas de recuperación de información, sin embargo la necesidad de algoritmos y representaciones más veloces nos obliga a compactar la información con la finalidad de aprovechar el uso de las jerarquías altas de memoria, es decir tener un acceso rápido a los datos [Navarro, 2006].

Existen estructuras de datos que permiten manipular la información en una representación compacta, dichas estructuras reciben el nombre de estructuras sucintas [Jacobson, 1988], recientemente, son aprovechadas en la compresión de índices de búsqueda por similitud en bases de datos métricas [Tellez et al., 2011, Santoyo, 2012]. Tales índices permiten la agrupación de objetos similares y facilitan la recuperación de información al realizar una consulta sobre la base de datos. Mientras los índices pueden compactarse y manipularse en memoria principal, los datos que forman la base de datos son almacenados directamente en disco, una manera de ahorrar el espacio en disco que utilizan los datos es mediante la aplicación de alguna técnica de compresión conocida como *zip* [1] o *gzip*[2], a toda la base de datos, sin embargo esta base de datos es descomprimida completamente para realizar consultas sobre ella.

De la observación anterior el problema a resolver se expresa de la siguiente manera: dada una base de datos métrica, diseñar un esquema de compresión de tal manera que la representación compacta de los datos pueda manipularse en memoria principal al realizarse una consulta. Revisando el estado del arte correspondiente a las bases de datos métricas así como el de las estructuras sucintas se concluyó que así como en los índices métricos las estructuras sucintas pueden ser útiles en la representación compacta de los datos.

La propuesta de solución que se presenta en este trabajo se basa en el siguiente argumento: cuando los elementos de una base de datos son indexados mediante alguna técnica de agrupación por similitud, se obtiene un índice de diferentes grupos donde sus elementos comparten características muy parecidas y por lo tanto diferencias pequeñas, aprovechando que las diferencias entre ellos es pequeña se propone hacer una nueva representación de la base de datos utilizando el agrupamiento por similitud y las diferencias de cada grupo. La nueva representación de la base de datos tomará de acuerdo al índice métrico de cada grupo un representante que será representado explícitamente y el resto de los elementos del grupo será representado con las diferencias entre estos y el elementos representante. La

[1] Formato de archivos comprimidos sin pérdida, creado por Phil Katz en 1989, http://www.info-zip.org/

[2] Formato de los archivos comprimidos por el programa de compresión *GNU ZIP* creado por Jean-loup Gailly y Mark Adler en 1992, http://www.gzip.org

nueva representación necesitará del uso de estructuras que permitan su manipulación de esta manera cuando una consulta sea realizada a la base de datos, el índice indicará que grupo o grupos descompactar en caso de ser similares a la consulta.

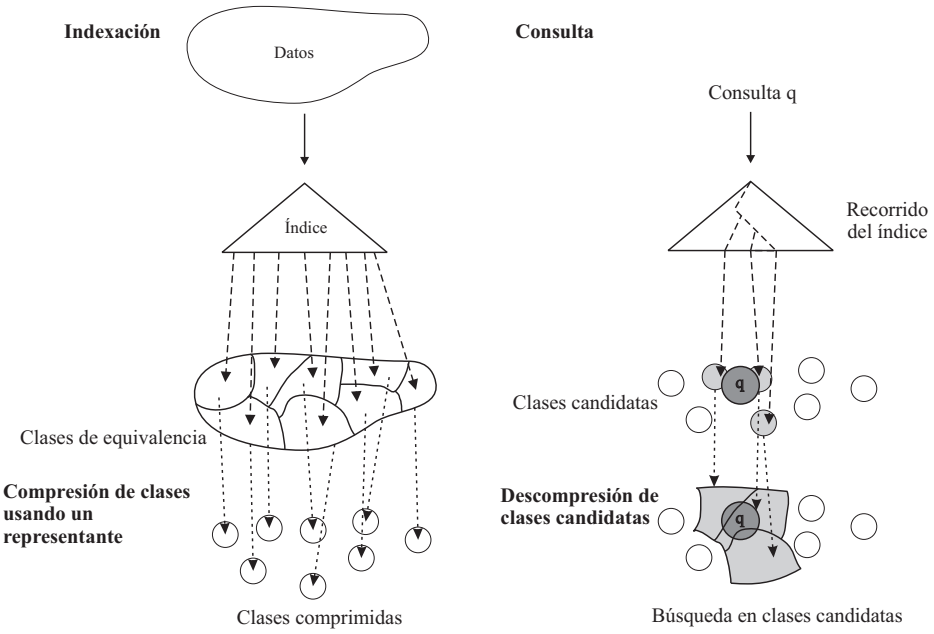


Figura 1.2: Esquema de compresión propuesto

La figura 1.2 muestra un esquema básico de la compresión propuesta en este trabajo. Como se observa en la figura por medio de este esquema de compresión sólo se descomprimen los grupos donde la consulta realizada intersecte.

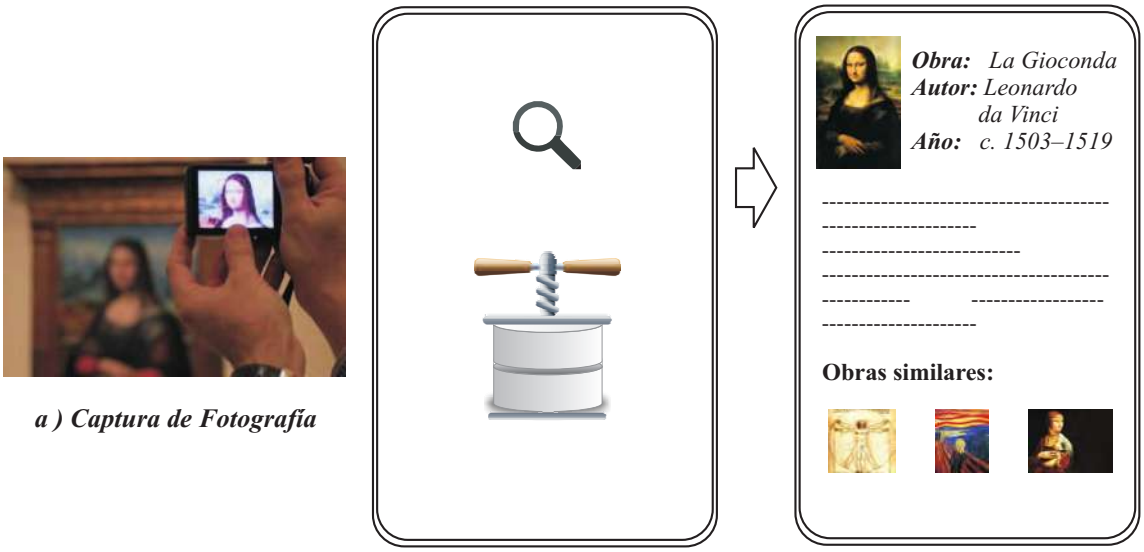
1.1.1. Posibles áreas de implementación

Siendo las bases de datos métricas un modelo de organización de la información recurrente en diferentes aplicaciones, la definición de un esquema de compresión como el que se propone resulta de utilidad en el desarrollo de aplicaciones móviles. Las aplicaciones móviles requieren acceso a información de manera local y remota. El acceso a datos de manera remota conlleva un costo económico para el usuario y de infraestructura para el proveedor del servicio móvil. Una de las razones por las que las aplicaciones móviles hacen uso de datos remotos es el espacio limitado con el que se cuenta en el dispositivo aún cuando esta información no es dinámica. Es en este caso donde el uso de un esquema de compresión

sobre una base de datos métrica puede ser una solución para aplicaciones móviles que requieran acceso a un volumen considerable de datos.

Ejemplo en una aplicación móvil

Suponiendo el caso de una aplicación de una red nacional de museos, la aplicación permite conocer la información de una obra buscada en una base de datos a través de una fotografía capturada por el usuario. La consulta de la obra se obtiene a través de la extracción de características de la fotografía, para su posterior comparación con los elementos de la base de datos comprimida. De esta manera sería posible tener la aplicación, el índice y la base de datos comprimida en un espacio menor o igual al de la base de datos original reduciendo el acceso a datos remotos en aplicaciones que así lo requieran.



a) Captura de Fotografía

b) Extracción de características y Consulta a la base de datos comprimida

c) Resultado de la consulta

Figura 1.3: Ejemplo de una posible aplicación móvil usando una base de datos métrica comprimida

En la figura 1.3 se ilustra el esquema básico de la posible aplicación descrita anteriormente. La investigación en la compresión de bases de datos métricas es una motivación en varias áreas, principalmente para ahorrar espacio y acceder en menor tiempo a los datos

como se puede observar en el ejemplo anterior.

Compresión en archivos

Otra manera de ver el modelo de compresión propuesto, es utilizarlo en datos que puedan ser agrupados por medio de representantes. Por ejemplo: dada una imagen puede interpretarse el ancho y alto de la imagen como una base de datos métrica vectorial, es decir, cada fila a lo ancho representaría un vector donde sus coordenadas son representadas por el valor de cada pixel.

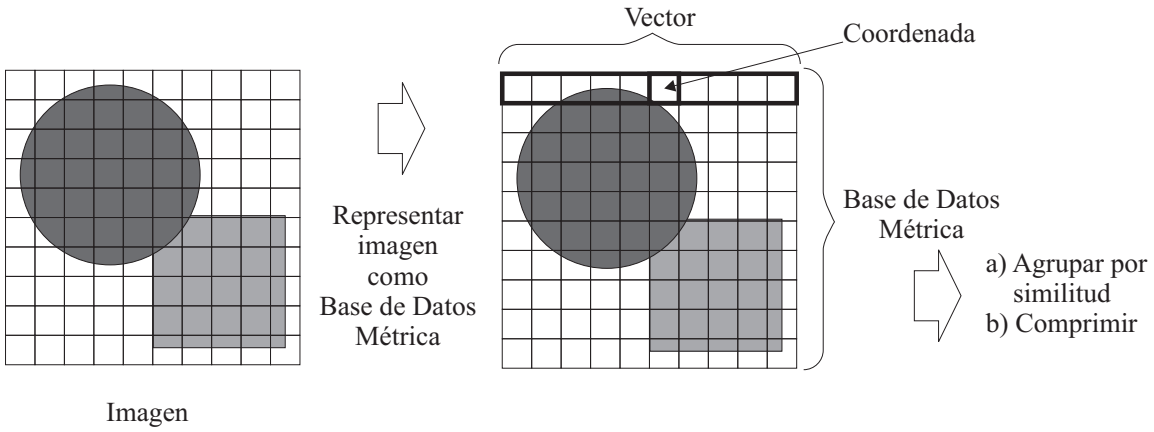


Figura 1.4: Compresión de una imagen usando diferencias

La agrupación y compresión por medio de diferencias puede aplicarse también en audio y en vídeo, tal vez no resulte útil para la compresión visual (como *jpg* o *gif*) de este tipo de archivos, sin embargo, la representación compacta por medio de diferencias y representantes podría resultar útil en la transmisión de información.

1.2. Antecedentes

A continuación se presentan algunos conceptos básicos relacionados con la compresión de datos y la teoría de información, puesto que serán aplicados para la evaluación del trabajo realizado.

Para presentar el contexto de la compresión de datos es importante mencionar la *jerarquía de memoria*, ya que su estructura motiva el diseño de esquemas de compresión los cuales son presentados en esta sección, así como el concepto de *entropía*, una medida de información que define la cantidad limite de compresión de una fuente de información.

1.2.1. La jerarquía de memoria

La información en un sistema de computo es almacenada en dispositivos de memoria, La jerarquía de memoria define la manera en que se procesa y administra la información y se define tomando en cuenta el tiempo de acceso, la capacidad y el costo. Un sistema ideal contendría una memoria de rápido acceso y de gran capacidad de almacenamiento, sin embargo esto no es posible, ya que el desarrollo tecnológico es costoso. Una solución económica para lograr el sistema ideal, es la *jerarquía de memoria*, misma que toma ventaja de la localidad y el intercambio para el costo-rendimiento de tecnologías de memoria. Según la ley de Moore [Moore et al., 1965] la velocidad de los procesadores de computo(CPU [3]) incrementa cada 18 meses, sin embargo los tiempos de acceso en disco no han mejorado mucho. La velocidad de las redes es comparable con la de transferencia en disco. Cada vez aparecen memorias cachés más rápidas y caras (aunque de poco espacio) [Navarro, 2006].

La figura 1.5 muestra un diagrama de una jerarquía de memoria multinivel, que incluye tamaños típicos y tiempos de acceso a los datos. [Hennessy y Patterson, 2012]

[3]Central Processing Unit

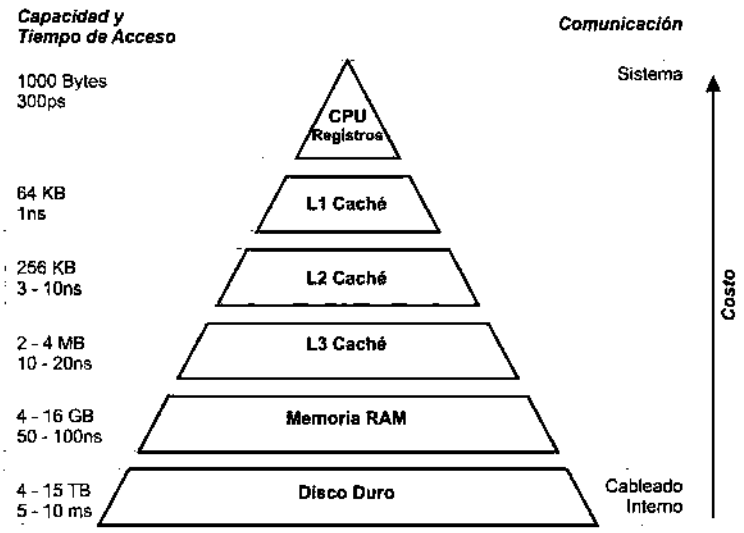


Figura 1.5: Jerarquía de Memoria del CPU

Es importante conocer la jerarquía de memoria para el buen funcionamiento de las aplicaciones y de la información que éstas procesan. Aunque el almacenamiento en varios dispositivos como el disco duro es muy grande, lo ideal es tener la aplicación y los datos en RAM para tener un tiempo de respuesta rápido, en el caso de una base de datos, podemos mejorar el acceso a la información creando estructuras de datos para indexar tal información, obteniendo así, tiempos de operación que mejoran cuando la información de la base de datos y la estructura para acceder a ellos se encuentra en la memoria RAM.

De la Primera ley de Parkinson:

"Work expands so as to fill the time available for its completion"

se deriva el siguiente corolario:

Corolario 1. *"La necesidad de transmisión y almacenamiento crece al doble de la mejora en capacidad de transmisión y almacenamiento".*

Esta proposición motiva el diseño de esquemas de compresión de datos, dichos esquemas toman ventaja de la jerarquía de memoria puesto que los datos comprimidos en disco duro se pueden almacenar también en memoria principal. La jerarquía de memoria además ha motivado el diseño de estructuras de datos compactas, dichas estructuras son

capaces de representar datos en poco espacio y mejorar su comportamiento al operar en memoria rápida [Navarro, 2006].

1.2.2. Compresión de datos

La compresión de datos es el proceso de convertir una cadena de datos de entrada (la cadena fuente o los datos originales a tratar), en otra cadena de datos (la salida, la cadena de bits, o la cadena comprimida), que tenga un tamaño más pequeño.

Sayood [Sayood, 2000] define la compresión de datos como: *"El arte o la ciencia de representar información de una forma compacta"*.

Cuando manejamos información de algún tipo: texto, imágenes, vídeo, audio, entre otros, observamos que algunos de los datos o contenidos son redundantes, este es un aspecto que se aprovecha para la compresión de datos con la finalidad de tener una representación compacta de la información. La redundancia se puede observar de varias formas: en una imagen pueden ser colores muy parecidos, en un texto palabras o letras que se repiten continuamente, en un archivo de audio pueden ser redundantes las frecuencias que el oído humano no percibe.

1.2.3. Esquemas de Compresión

Un esquema de compresión tiene las siguientes características:

1) Un algoritmo de compresión que toma una entrada X y genera una representación compacta X_c que ocupa menos bits.
2) Un algoritmo de descompresión que toma una entrada X_c y genera una reconstrucción Y.

En la figura 1.6 se puede observar claramente el proceso de compresión y descompresión para un conjunto de datos.

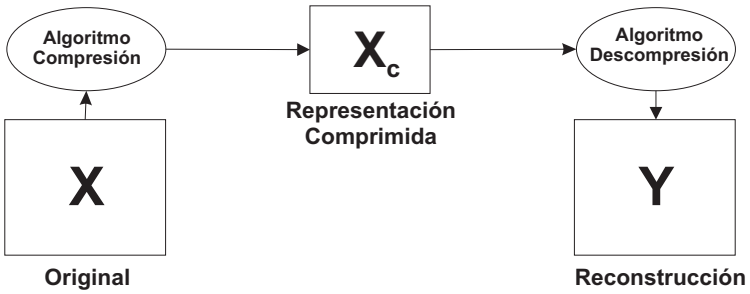


Figura 1.6: Esquema de Compresión

Los esquemas de compresión se dividen en dos grandes tipos:

- **Compresión sin pérdida**: Llamamos compresión sin pérdida a las técnicas o métodos donde se compacta la información de tal manera que al descompactarla se obtiene la misma información que se tenia antes de compactarla. Sin embargo esta condición puede limitar el grado de compresión. Observando la figura 1.6, cuando se tiene un esquema de compresión sin pérdida cuando $X = Y$.

- **Compresión con pérdida**: Son técnicas o métodos donde la información es compactada de tal manera que al descompactarla ésta no es igual a la información original, debido a que la información que no se toma en cuenta para el proceso de compresión es eliminada definitivamente. De igual manera que en la figura 1.6, en un esquema de compresión con pérdida $X \neq Y$.

Existen diferentes técnicas de compresión estas dependen del esquema de compresión *con/sin pérdida* y del tipo de dato a comprimir.

1.2.4. Evaluación de los esquemas de compresión

Para evaluar la calidad de un modelo de compresión se toma en cuenta:

- Complejidad del algoritmo.
- Necesidades de memoria.
- Tiempo de ejecución en una determinada plataforma.
- Grado de compresión.

- Similitud de la reconstrucción con la entrada original.

  La razón de compresión:

$$r = \frac{|X|}{|Y|} \tag{1.1}$$

  donde $|X|$ y $|Y|$, son el número de bits antes y después de la descompresión.

  El porcentaje de compresión

$$p = \frac{|Y| * 100}{|X|} \tag{1.2}$$

Cuando el esquema de compresión es con pérdida la diferencia entre los datos de la descompresión y los datos originales recibe el nombre de *distorsión.*

1.2.5. Información y Entropía

El concepto de compresión es natural cuando queremos representar algún tipo de información, para ello empleamos diferentes medios. Un ejemplo claro es el código Morse, Samuel Morse tomó en cuenta el hecho de que algunas letras como la e o la a son más frecuentes en palabras del idioma Inglés que las letras q o z, así creó códigos más cortos para palabras donde las letras son más frecuentes.

Claude E. Shanoon desarrolló una manera de estimar cuantitativamente la información y de esa manera tener una noción de ella [Sayood, 2003], es decir, cuando tenemos un texto en español, las palabras como "estas", "la", "que", no nos dicen mucho sobre el texto, en cambio las palabras con menor frecuencia como "árbol", "licencia", "reforestación" dan más información sobre el documento.

Llamamos entonces *entropía* a la medida de información usando la probabilidad de los símbolos que aparecen en un mensaje, con el ejemplo anterior sabemos que las palabras menos frecuentes aportan más información, entonces su entropía es más pequeña. Suponiendo que tenemos una serie de experimentos donde cada uno tiene las salidas $A_1, A_2, ..., A_m$, Shanoon asoció cada una de las salidas con una cantidad llamada *auto-información* (self-

information) definida como:

$$i(A_k) = log\frac{1}{P(A_k)} \tag{1.3}$$

donde $P(A_k)$ es la probabilidad de A_k.

Las unidades de *auto-información* son bits, nats o Hartleys dependiendo si la base del logaritmo es 2, e, 10.

Definimos formalmente entropía (H) como la cantidad promedio de información asociada con el experimento [Shannon, 1948].

$$H = E[i(A)] = \sum_{k=1}^{m} i(A_k)P(A_k) = \sum_{k=1}^{m} P(A_k)log\frac{1}{P(A_k)} \tag{1.4}$$

$$H = -\sum_{k=1}^{m} P(A_k)log(P(A_k)) \tag{1.5}$$

Codificación de los datos

Shanoon presentó el Teorema de codificación de la fuente *(Source-coding Theorem)* [Shannon, 1948], este teorema establece un límite fundamental sobre el porcentaje al que la salida de una fuente de información puede comprimirse sin que cause un error en la probabilidad. Como anteriormente se definió la entropía es la medida de incertidumbre o equivalencia del contenido de la información de alguna fuente, razón por la que la entropía tiene un rol importante en este teorema.

Teorema 1. *N i.i.d.[4] variables aleatorias, cada una con entropía $H(X)$, pueden comprimirse en más de $NH(x)$ bits, con un riesgo pequeño de pérdida de información si $N \to \infty$; inversamente si son comprimidas en menos de $NH(X)$ bits, es prácticamente cierto que se perderá información.*

Este teorema, demostrado por Shannon [Shannon, 1948], básicamente dice que un mensaje de n símbolos puede (en promedio) comprimirse hasta nH bits pero no en un

[4]Abreviación en inglés de: *independent and identically distributed*, se refiere a variables aleatorias independientes e idénticamente distribuidas.

número más pequeño [MacKay, 2003].

El teorema de Shannon representa el limite teórico de la compresión de una fuente de información, proporcionando las condiciones necesarias y suficientes para la existencia de códigos fuente, tomando en cuenta este teorema se han desarrollado esquemas de compresión y estructuras de datos compactas donde se busca comprimir la fuente en un orden cercano a la entropía.

1.3. Estado del arte

1.3.1. Compresión de bases de datos

La compresión de bases de datos es dependiente del tipo de información, es decir, tal vez no es fácil comprimir números reales, sin embargo puede ser menos complejo comprimir textos. El estudio en la compresión de bases de datos que se puede encontrar en la literatura se enfoca a las bases de datos relacionales así como a las bases de datos biológicas, sin embargo no se encuentra en la literatura un estudio especifico sobre la compresión de bases de datos métricas.

Actualmente algunas herramientas para la gestión de base de datos relacionales implementan técnicas de compresión como la compresión OLTP sigla en inglés de *Procesamiento de Transacciones En Línea* (OnLine Transaction Processing) [OLT, 2012] para aplicaciones cliente-servidor, específicamente es la compresión sobre tablas OLTP del manejador de bases de datos Oracle [5], el algoritmo está basado en la eliminación de datos duplicados para un bloque de base de datos, incluso a través de múltiples columnas.

Bases de datos biológicas son comprimidas utilizando algoritmos de compresión sobre las secuencias biológicas. Las técnicas de compresión sobre secuencias biológicas aprovechan la redundancia y repetición de los datos. A la vez que se utilizan algoritmos de compresión genéricos como LZ77 [Ziv y Lempel, 1977], han surgido algoritmos específicos para secuencias biológicas como: Pattern Hunter[Li et al., 2002] y Pattern Hunter II

[5] www.oracle.com

[Li et al., 2003] que realizan la compresión buscando repeticiones inexactas, es decir que el patrón redundante a buscar por ejemplo "*aaac*" pueda variar en algún carácter "*aagc*" creando una cadena binaria de la comparación "*1101*". Otro algoritmo DnaX [Manzini y Rastero, 2004] divide la secuencia en 20 fragmentos donde busca repeticiones exactas que sustituye por una referencia en el archivo comprimido.

En lo que respecta a las bases de datos métricas podemos encontrar técnicas de compresión sobre los índices, no así de los datos mismos. El estudio de la compresión de los índices es una técnica reciente, algunos índices buscan reducir en memoria el espacio del índice aumentando el número de pivotes en memoria como *Fixed Queries Array* [Chávez et al., 2001] y *Fixed Queries Three* [Baeza-Yates et al., 1994] ambos índices tratan de reducir el uso de memoria del *Fixed-Height Fixed Queries Tree* [Baeza-Yates et al., 1994]. Haciendo uso de estructuras sucintas en [Tellez et al., 2011, Téllez, 2012] se propone el índice List of Clusters comprimido, de igual forma se comprime el espacio que utiliza el índice Locality Sensitive Hashing (LSH) en [Santoyo, 2012].

1.4. Objetivos de la Tesis

1.4.1. Objetivo general

Diseñar un método de compresión sin pérdida de una base de datos métrica, que permita medir la distancia entre dos objetos arbitrarios sin descomprimir en su totalidad la base de datos únicamente las regiones o zonas que intersecten con el radio de alguna consulta realizada.

1.4.2. Objetivos particulares

- Aplicar el método de compresión sobre la base de datos métrica CoPhiR [Bolettieri et al., 2009] la base de datos CoPhiR es un repositorio de imágenes de Flickr creado por varios consorcios universitarios para realizar investigación sobre recuperación de información.

- Definir la evaluación de distancias de dos objetos arbitrarios en la base de datos sin

la necesidad de descomprimir completamente la base de datos.

- Agregar los métodos implementados a la librería Natix[6].

1.5. Organización de la Tesis

Este proyecto de tesis ha sido organizado en dos partes, la primera parte presenta los conceptos necesarios para comprender el trabajo realizado, en la segunda parte se describen los aportes de la tesis.

Primera parte: En los primeros tres capítulos se introducen los conceptos que contextualizan la propuesta de compresión de bases de datos metricas que se presenta.

- Capítulo 1 se presenta la descripción del problema a abordar y se definen los objetivos que se desean alcanzar para resolver el problema descrito, posteriormente se realiza una breve introducción al entorno de la compresión de datos mostrando las técnicas más utilizadas así como el estado del arte en lo que respecta a compresión de datos.
- Capítulo 2 explica las características y conceptos de las bases de datos métricas, así como la búsqueda por similitud y los métodos de indexación dando especial énfasis al índice LC y Rev-LC debido a que serán utilizados en la implementación.
- Capítulo 3 se definen las estructuras sucintas y se presenta el SArray, estructura que permitirá la representación compacta de la información en el proyecto.

Segunda Parte: En esta parte se realiza la propuesta para la compresión de bases de datos métricas, se presentan los resultados y finalmente las conclusiones.

- Capítulo 4 se presenta la estrategia para resolver el problema propuesto y los algoritmos utilizados.
- Capítulo 5 expone los resultados obtenidos del modelo de compresión implementado sobre dos muestras (1 millón y 10 millones) de la colección de CoPhIR.

[6] www.natix.org

- Capítulo 6 mencionando también recomendaciones para trabajos posteriores.

1.6. Conclusión

El propósito de esta introducción general ha sido definir y analizar desde un punto de vista teórico los conceptos básicos de la teoría de información y la compresión de datos para poder aplicarlos en la compresión de una base de datos métrica. Se puede observar con el desarrollo en los dispositivos de almacenamiento la jerarquía de memoria juega un papel importante para la implementación de esquemas de compresión de diferentes tipos de datos. En la revisión del estado del arte se observa que no hay un trabajo relacionado específicamente con la compresión de una base de datos métrica, sin embargo se han realizado representaciones compactas sobre los índices métricos lo que permite estudiar estas técnicas para aplicarlas en la compresión de las bases de datos métricas.

El esquema propuesto está motivado principalmente para su implementación futura en aplicaciones móviles, siendo el objetivo principal del trabajo: la definición y prueba del esquema sobre una muestra de la base de datos CoPhIR en particular. La característica principal del esquema que se propone es que, mediante la indexación por grupos sobre la base de datos métrica cada grupo sea comprimido en base a un representante del grupo, lo que permitirá que al realizar una consulta sólo se descompriman los grupos de interés y no la base de datos completa.

Capítulo 2

Bases de Datos Métricas

Una base de datos métrica es una colección de datos donde se ha definido una métrica que da información sobre la similitud entre sus elementos. Existen muchos tipos de colecciones, ello depende del tipo de aplicación que utilicemos, podemos crear bases de datos métricas de documentos de texto, imágenes, audio y vídeo principalmente. La finalidad de estas colecciones es resolver problemas tales como:

- Reconocimiento de patrones
- Indexación multimedia
- Estadísticas
- Clasificación

2.1. Espacio Métrico

Llamamos espacio métrico a un conjunto de datos (espacio) donde definimos una medida de semejanza entre sus elementos (distancia), medida que depende directamente de la aplicación y las características a evaluar.

Formalmente: Un espacio métrico $(\mathbb{U}, d)$ donde $\mathbb{U}$ es un universo válido de objetos y $\mathbb{S}$ un subconjunto finito de objetos de tamaño $n = |\mathbb{S}|$ tal que $\mathbb{S} \subseteq \mathbb{U}$, al que llamamos diccionario, *base de datos* o nuestro conjunto de objetos. [Chávez et al., 001b]

La función:

$$d : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}^{+} \tag{2.1}$$

denota la medida de distancia entre un par de objetos. Es decir: mientras más pequeña sea la distancia, los objetos son más parecidos.

Las funciones de distancia tienen las siguientes propiedades:

1) Positividad : $\forall\ x, y \in \mathbb{X},\ d(x, y) \geq 0$

2) Simetría : $\forall\ x, y \in \mathbb{X},\ d(x, y) = d(y, x)$

3) Reflexividad: $\forall\ x \in \mathbb{X},\ d(x, x) = 0$

En más casos:

4) Positividad Estricta: $\forall\ x, y \in \mathbb{X},\ x \neq y \rightarrow d(x, y) > 0$

5) Desigualdad Triangular: $\forall\ x, y, z \in \mathbb{X},\ d(x, y) \leq d(x, z) + d(z, y)$

Las propiedades anteriores aseguran la consistencia de la función distancia.

Si d es ya una métrica y satisface la propiedad $p5$) el par $(\mathbb{X}, d)$ es un *espacio métrico*. Para cada $x, y \in \mathbb{X}$, llamamos el número $d(x, y)$ la *distancia* ente x y y con respecto a la métrica d. Un *espacio pseudo-métrico* se presenta cuando no se satisface la propiedad $p4$)

2.2. Espacio Vectorial

Llamamos espacio vectorial a un espacio métrico particular donde los elementos u objetos ($\vec{x}$) son identificados con n coordenadas de valor real (x_1,x_2,x_3,...,x_n) y cada x_i es una "coordenada" del objeto. Existe un amplio desarrollo en el estudio de los espacios vectoriales aprovechando sus propiedades geométricas, sin embargo estos métodos no pueden aplicarse en todos los espacios métricos donde la información conocida es la distancia entre los objetos.

Existen diversas funciones de distancia en los espacios vectoriales, entre ellas la más utilizada es la familia de distancias llamada L_p o "Minkowski".

$$L_p(u_1, u_2, ..u_n), (v_1, v_2, ..v_n)$$

$$L_p(\vec{u}, \vec{v}) = (\sum_{i=1}^{n} |(u_i - v_i)|^p)^{\frac{1}{p}} \tag{2.2}$$

2.2.1. Distancia L_1

La distancia L_1 de la familia L_p también llamada distancia de "Manhattan" o "Minkowski" en particular si $p = 1$, se define como:

$$L_1(\vec{u}, \vec{v}) = \sum_{i=1}^{n} |(u_i - v_i)| \tag{2.3}$$

La utilización de distancia L_1 será muy importante para este trabajo.

Esta medida es igual a encontrar la distancia al trasladarse de una esquina a otra en una ciudad, de ahí el nombre Manhattan, ya que la distribución de las calles en esta isla aparenta una cuadrícula.

2.2.2. Distancia L_2

La distancia L_2 o *Euclidiana* es la representación espacial de la distancia, corresponde a la intuición humana de distancia.

$$L_2(\vec{u}, \vec{v}) = (\sum_{i=1}^{n} |(u_i - v_i)^2|)^{\frac{1}{2}} \tag{2.4}$$

2.2.3. Distancia L_∞

Otra distancia de la familia L_p muy usada es L_∞ que corresponde a la distancia cuando p tiende a infinito, es la distancia *máxima* entre dos puntos a lo largo de las coordenadas.

$$L_\infty((x_1, x_2, ..., x_n), (y_1, y_2, ..., y_n)) = max_{i=1}^{n} |x_i - y_i| \tag{2.5}$$

2.3. Búsquedas por similitud

Las aplicaciones que utilizan bases de datos multidimensionales requieren realizar búsquedas de objetos similares, las principales motivaciones principales de búsqueda son:

- Recuperar elementos particulares en la base de datos.
- Recuperar elementos similares en un rango definido o bien el k número de elementos más similares a un elemento dado.

Formalmente dichas operaciones se plantean como sigue:

- **Range query** *Consulta por rango* (q, r), $q \in \mathbb{U}$, $r \in \mathbb{R}^+$, devuelve todos los elementos que se encuentran a una distancia r de q, así $(q, r) = \{u \in \mathbb{S}, d(u, q) \leq r\}$
- **k nearest neighbors (k-NN)** *k vecinos más cercanos*, devuelve los k elementos de $\mathbb{S}$ cercanos a q, es decir, retorna el conjunto $C \subseteq \mathbb{S}$ tal que $|C| = k$ y $\forall\ x \in C,\ y \in \mathbb{S} - C,\ d(x, q) \leq d(y, q)$

El volumen definido por (q, r) es llamado radio de consulta (*query ball*), y los elementos dentro de él son reportados. Las consultas tipo *vecinos cercanos* (*nearest neighbors*) pueden implementarse usando consultas por rango.

Se han desarrollado diversas técnicas de indexación de elementos. La creación de índices es una manera de ordenar la información de tal manera que la consulta a los datos sea más rápida que revisar secuencialmente la base de datos. En el estudio de la búsqueda por similitud se presenta un problema conocido en matemáticas y estadística como *la maldición de la dimensión*, es decir, cuando los elementos u objetos de la base de datos son de altas dimensiones el proceso de búsqueda puede crecer exponencialmente con la dimensión, específicamente se refiere a la incertidumbre que existe en el resultado de una búsqueda.

Existen dos clases de técnicas utilizadas para realizar algoritmos de búsqueda por similitud: basada en pivotes y basada en particiones compactas, estas técnicas utilizan las distancias en lugar de las coordenadas de los objetos. Los principios que algunas de estas técnicas utilizan se basan en métodos de partición.

2.4. Métodos de partición del espacio métrico

El particionamiento, en general, es uno de los principios fundamentales para cualquier estructura de almacenamiento con el objetivo de dividir el espacio de búsqueda en subgrupos, de tal manera que, para una consulta dada, sólo algunos de estos grupos sean revisados [Zezula et al., 2006]. Dado un conjunto $\mathbb{S} \subseteq \mathbb{U}$ de objetos en un espacio métrico $(\mathbb{U}, d)$, Uhlmann [Jeffrey, 1991] define *partición en esfera* y *partición del hiperplano*, mientras Yianilos [Yianilos, 1999] sugiere la partición por exclusión. Brevemente se mencionan a continuación estas técnicas.

2.4.1. Partición en esfera

Esa partición divide el conjunto $\mathbb{S}$ en subconjuntos S_1 y S_2 usando un corte esférico respecto a $p \in \mathbb{U}$, donde p es un pivote, seleccionado arbitrariamente.
Sea d_m la mediana de $\{d(o_i, p), \forall o_i \in \mathbb{S}\}$. Entonces toda $o_j \in \mathbb{S}$ es distribuida a S_1 o S_2 siguiendo las siguientes reglas:

- $S_1 \leftarrow \{o_j | d(o_j, p) \leq d_m\}$,
- $S_2 \leftarrow \{o_j | d(o_j, p) \geq d_m\}$

La redundancia de las condiciones $\leq$ y $\geq$ aseguran el balance cuando la mediana no es única. Esto se logra asignando cada elemento a la distancia media de algunos de los subconjuntos arbitraria pero balanceadamente. La figura 2.1a, muestra el pivote p seleccionado y la distancia media d_m que determina la partición en esfera.

2.4.2. Partición del hiperplano

La partición del hiperplano generalizada puede considerarse como una partición en bolas ortogonal. Esta partición también divide el conjunto $\mathbb{S}$ en dos subconjuntos S_1 y S_2. Utilizando dos pivotes $p_1, p_2 \in \mathbb{U}$ seleccionados arbitrariamente todos los objetos $o_j \in \mathbb{S}$ son asignados a S_1 o S_2 dependiendo de las distancias a los pivotes siguiendo:

- $S_1 \leftarrow \{o_j | d(p_1, o_j) \leq d(p_2, o_j)\}$

- $S_2 \leftarrow \{o_j | d(p_1, o_j) \geq d(p_2, o_j)\}$

En contraste con la partición en bolas, el hiperplano generalizado no garantiza una división balanceada, por lo que una elección adecuada de los puntos de referencia es un gran desafío. La figura 2.1b es un ejemplo de una división balanceada de un conjunto hipotético.

2.4.3. Partición por exclusión

Excluded Middle Partitioning, Yianilos [Yianilos, 1999] divide el conjunto $\mathbb{S}$ en tres subconjuntos S_1, S_2 y S_3. La idea central es dejar los puntos cerca de d_m definiendo los dos subconjuntos S_1 y S_2. Los puntos excluidos forman el tercer sunconjunto S_3. Con esto la búsqueda para objetos similares siempre ignora al menos unos de los subconjuntos S_1 o S_2, siempre que la selectividad de la búsqueda sea más pequeña que el ancho de la zona de exclusión. Naturalmente los puntos excluidos no son perdidos, pueden formar un tercer subconjunto y si este subconjunto es grande sirve de base para una nueva partición. Dado el ancho de la zona de exclusión $2p$ la partición se define como sigue:

- $S_1 \leftarrow \{o_j | d(o_j, p) \leq d_m - p\}$
- $S_2 \leftarrow \{o_j | d(o_j, p) > d_m + p\}$
- $S_3 \leftarrow$ de otra manera.

La figura 2.1c, muestra una situación donde la división es balanceada aunque esto no siempre es garantizado.

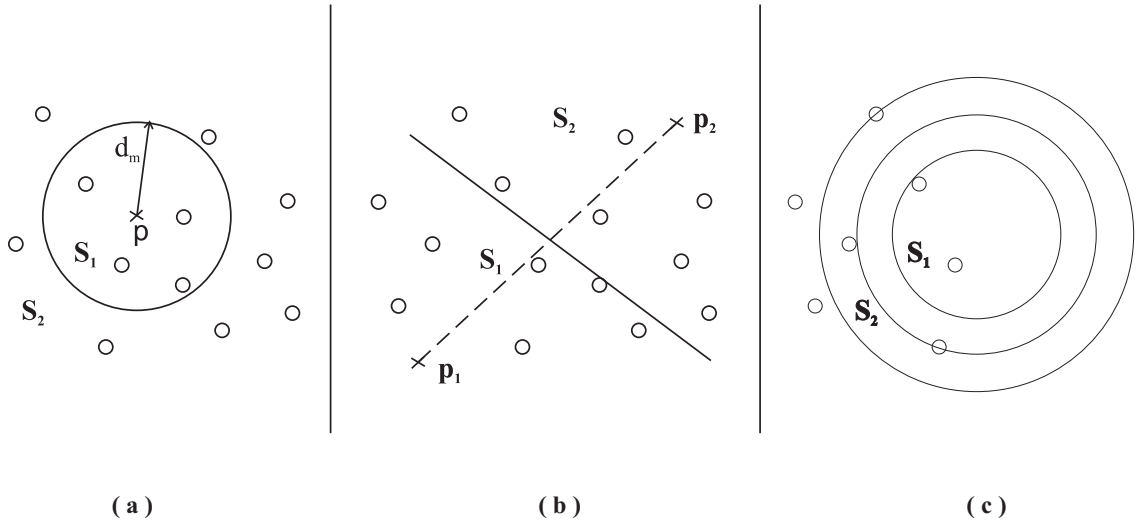


Figura 2.1: Ejemplo de particiones del espacio métrico

En la figura 2.1 se observa un ejemplo del tipo de particiones descritas anteriormente: *a)* Partición en esfera, *b)* Partición del Hiperplano, *c)* Partición por exclusión.

2.5. Métodos de indexación basados en pivotes

Estos algoritmos seleccionan un número de "pivotes" que son elementos de la base de datos y clasifican el resto de los elementos de acuerdo a la distancia con los mismos. A continuación se mencionan algunos de los principales métodos que utilizan esta técnica.

2.5.1. BKT

Burkhard-Keller Tree, es una de las primeras soluciones para realizar búsquedas en espacios métricos. Fue presentada en [Burkhard y Keller, 1973]. El árbol supone la función discreta de distancia y es construido recursivamente de la siguiente manera: Del conjunto indexado $\mathbb{S}$, un objeto arbitrario $p \in \mathbb{S}$ es seleccionado como el nodo raíz del árbol. Para cada distancia $i \geq 0$, los conjuntos $S_i = \{o \in \mathbb{S}, d(o,p)\}$ se definen como grupos de todos los objetos a una distancia i de la raíz p. Un nodo hijo de la raíz p es construido para cada conjunto no vacío S_i.

Todos los nodo hijos pueden reparticionarse recursivamente hasta que no sea posible crear un nuevo nodo hijo. Cuando un nodo hijo es dividido, algún objeto o_j del conjunto S_i es seleccionado como un representante del conjunto. Un nodo hoja es creado para cada conjunto S_i, S_i proporcionada no es reparticionada nuevamente. Un conjunto S_i ya no es dividido si contiene sólo un objeto. Los objetos seleccionados como raíces de subárboles(almacenados en nodos internos) son llamados pivotes. El algoritmo para consultas por rango es simple. La búsqueda por rango $R(q, r)$ comienza en la raíz del nodo del árbol y compara el objeto p con la consulta q. Si p satisface la consulta, es decir $d(p, q) \leq r$, el objeto p es devuelto. Posteriormente, el algoritmo ingresa todos los nodos hijos o_i y procede recursivamente hacia abajo. BKT tiene una complejidad de espacio lineal $O(N)$ y la complejidad de construcción es medida en términos del número de distancias calculadas $O(n \log n)$, esto bajo condiciones especiales. El tiempo de búsqueda también medido en términos de distancias calculadas es $O(n^{\alpha})$, donde α es un número real que satisface $0 < \alpha \leq 1$ lo que depende del radio de búsqueda y de la estructura del árbol.

2.5.2. FQT

Fixed Queries Tree, originalmente presentado en [Baeza-Yates et al., 1994] es una modificación del BKT. BKT usa pivotes individuales en diferentes niveles mientras que FQT utiliza un mismo pivote para todos los nodos al mismo nivel. Todos los objetos en un conjunto dado $\mathbb{S}$ son guardados en hojas y los nodos internos son utilizados para la navegación durante la búsqueda (o inserción). El algoritmo de búsqueda es el mismo que el de BKT. La ventaja de esta estructura es la reducción del número de distancias calculadas, ya que si más de un subárbol tiene que ser accedido para evaluar una consulta, sólo se evalúa una distancia entre la consulta y el pivote específico.

La complejidad de espacio es superlineal porque los objetos seleccionados como pivotes son duplicados, así que la complejidad varia desde $O(n)$ a $(n \log n)$. El número de cálculo de distancias requeridas para construir el árbol es de $O(n \log n)$. La complejidad de la búsqueda es $O(n^{\alpha})$, donde α esta en un rango $0 < \alpha < 1$ dependiendo también del radio de consulta y la distribución de los objetos en el espacio métrico.

2.5.3. FHFQT

Fixed-Height Fixed Queries Tree, propuesto en [Daniel et al., 1997] y [Baeza-Yates et al., 19
Es una variante del FQT. Esta estructura tiene todos sus nodos hoja al mismo nivel, es decir, la hojas están a la misma profundidad h, en otras palabras, caminos ordenados son extendidos por caminos adicionales. La ampliación del árbol mejora la implementación de la búsqueda ya que el proceso de búsqueda en caminos largos puede ser detenido antes de alcanzar la hoja. La complejidad de espacio del FHFQT es superlineal alrededor de $O(n)$ y $O(nh)$, donde h es la altura del árbol. Se construye en $O(nh)$ cálculos de distancia. La complejidad de la búsqueda es constante $O(h)$ que es el número de distancias evaluadas a h pivotes. El tiempo de calculo extra es proporcional al número de nodos transversales y resta $O(n^{\alpha})$, donde $0 < \alpha < 1$ dependiendo del rango de búsqueda y estructura del índice. El tiempo extra es utilizado en la comparación de distancias y en atravesar el árbol. En la práctica la altura óptima del árbol $h = O(\log n)$ no puede almacenarse debido a problemas de espacio.

2.5.4. FQA

Fixed Queries Array, fue presentado en [Chávez et al., 1999, Chávez et al., 2001]. A pesar que la estructura del FQA esta fuertemente relacionada al FHFQT, no es una estructura del árbol. Primeramente, el FHFQT con altura h es construido sobre un conjunto de objetos X. Si los caminos raíz-hojas del FHFQT son atravesados en orden de izquierda a derecha e ingresados en un array, el resultado es el FQA. Cada columna consiste en h números que representan las distancias a cada pivote utilizado en el FHFQT. La secuencia de h números es el camino de la raíz del FHFQT a su hoja. El FQA simplemente almacena los objetos de la base de datos lexicográficamente ordenados por su secuencia de distancias. Específicamente los objetos son ordenados inicialmente con respecto al primer pivote y aquellos a la misma distancia son ordenados respecto al segundo pivote y así sucesivamente. La búsqueda por rango es heredada del FHFQT. Cada nodo interno del FHFQT corresponde a un rango de elementos en el FQA. Los nodos hijo tienen un rango de elementos que es un subrango del rango de sus padres en el array. Naturalmente hay una similitud aproximada

entre el FQA, el *Suffix Tree* y el *Suffix Array* [Frakes y Baeza-Yates, 1992]. FQA requiere en espacio $n\dot{h}\dot{b}$ bits, donde b es el número de bits usado para almacenar una distancia. El número de distancias calculadas durante la búsqueda es $O(h)$. Como se probó en [**?**] el tiempo de cálculo extra del FHFQT es $O(n\alpha)$. El FQA tiene $O(n\alpha \log n)$ de complejidad extra donde $0 < \alpha < 1$. El tiempo extra de cálculo se debe a la búsqueda binaria del array.

Las estructuras presentadas (BKT, FQT, FHTQT y FQA) fueron diseñadas para funciones métricas discretas, ya que un nodo hijo es necesario para algún valor de distancia específico. Si se aplica esto a una métrica continua, el árbol se convertiría en un árbol plano cuya altura es uno y el algoritmo de búsqueda realizaría una búsqueda secuencial. Con el fin de transformar correctamente el caso continuo al discreto, se segmenta el dominio de los valores potenciales de distancia en un pequeño conjunto de subintervalos.
Dos esquemas para discretizar el FQA se han propuesto en [Chávez et al., 1999, Chávez et al., 2001]. El primero divide el rango de valores posible en porciones de igual anchura, a tal resultado se le llama *Fixed Slices Fixed Queries Array* (FSFQA) [Chávez et al., 2001].
Tal partición puede dar lugar a cortes vacíos donde ningún objeto de la base de datos se encuentre. Esto ha motivado a otro reciente enfoque, dividir el rango completo en segmentos, cada uno con el mismo número de objetos. En otras palabras el dominio se divide en cuantiles fijos. El FQA resultante es llamado *Fixed Quantiles Fixed Queries Array* (FQFQA) [Chávez et al., 2001].

2.5.5. VPT

Vantage Point Tree [Yianilos, 1993] fue diseñado expresamente para funciones de distancia continua, aunque también soporta distancias de funciones discretas con no muchas modificaciones. Se basa en el principio de partición en esfera que divide un conjunto en subconjuntos S_1 y S_2 basado en un objeto elegido p pivote, y la mediana d_m distancia de p a los objetos en $\mathbb{S}$. A partir de todo el conjunto de objetos $\mathbb{S}$ y de forma recursiva la aplicación de este procedimiento conduce a un árbol binario balanceado. El uso de la mediana para dividir un conjunto de datos en dos subconjuntos puede reemplazarse por una

estrategia que emplea la *media* de las distancias de p para todos los objetos en $\mathbb{S} \setminus \{p\}$. Este método, llamado punto medio *middle point* en [Chávez et al., 2001], puede dar un mejor rendimiento para los datos vectoriales de alta dimensión. Una desventaja de la estrategia del punto medio es que puede producir un árbol desequilibrado, impactando negativamente la eficiencia del algoritmo de búsqueda. El algoritmo de búsqueda para una consulta por rango $R(q, r)$ atraviesa el VPT desde la raíz a las hojas. Para cada nodo interno, evalúa la distancia $d(q, p)$ entre el pivote p y el objeto de consulta q Si $d(q, p) \leq r$, el pivote se informa a la salida. Para nodos internos, el algoritmo decide cual subárbol revisar. Para ello requiere establecer límites inferiores en las distancias de los objetos q en los subárboles izquierdos y derechos. Si el radio de consulta r es menor que el limite inferior, el algoritmo no visita el correspondiente subárbol. VPT utiliza $O(n)$ en espacio, la construcción $O(n \log n)$ y la búsqueda $O(\log n)$.

2.5.6. MW-VPT

Multi-Way Vantage Point Tree, propuesto por [Bozkaya y Ozsoyoglu, 1997] como una modificación del VPT donde se establece otra política de partición. En particular el método de partición en esfera es generalizado para producir mas de dos particiones. El árbol utiliza $k - 1$ umbrales $dm_1, ..., dm_{k-1}$ en lugar de la mediana dm para partir el conjunto en k subconjuntos con cortes esféricos. Sin embargo los experimentos demuestran que la implementación del MW-VPT no es siempre buena, pues los cortes esféricos resultan muy delgados, por ejemplo en el caso de dominios de alta dimensión donde las distancias entre algún par de objetos son prácticamente iguales. El algoritmo de búsqueda lleva a más ramas durante la ejecución de la consulta. Si los hijos i o k de un nodo tienen que ser buscados entonces se evalúan i distancias al siguiente nivel, ya que todas las distancias entre q y cada pivote seleccionado tienen que calcularse. La construcción de MW-VPT requiere $O(n \log_k n)$ en términos de cálculos de distancia. El espacio es $O(n)$ y la búsqueda es $O(\log_k n)$ esta complejidad para un radio 0 o muy pequeño y para una base de datos que le permita esa altura .

2.5.7. Excluded Middle Vantage Point Forest

Es otra estructura presentada en [Yianilos, 1998] basada en partición en esfera. La motivación para el VPF viene de la siguiente observación: Ya que la búsqueda del VPT es sublineal, su comportamiento depende no sólo del conjunto de datos, es decir la distribución de la distancia en $\mathbb{S}$, pero también en la selección del objeto de búsqueda q. El VPF admite el tiempo de búsqueda sublineal en el peor caso para consultas con un radio fijo hasta el máximo p, así el rendimiento no depende de la distribución de la consulta. Esta estructura introduce un nuevo concepto de exclusión de objetos a *middle distances* (distancias medias) modificando la técnica de partición.

$$bp_\rho(o) = \begin{cases} 0 & si\ d(o,p) \leq d_m - \rho \\ 1 & si\ d(o,p) > d_m + \rho \\ 2 & otro\ caso \end{cases}$$

El VPF es lineal en $O(n)$, con tiempo de construcción $O(n^{2-\alpha})$, donde $O(n^{1-\alpha})$ es el número de arboles en el VPT. Consultas por similitud son evaluadas en $O(n^{1-n}\log n)$ cálculos de distancia.

2.5.8. AESA

Approximating and Eliminating Search Algorithm, presentado en [Vidal, 1994] utiliza una matriz de distancias entre los objetos de la base de datos que han sido computados durante la construcción de la estructura AESA. La estructura es simple, una matriz $n \times m$ que almacena las distancias entre todos los pares n en la base de datos. Manteniendo la propiedad de simetría de las funciones de distancia, sólo la mitad de la matriz por debajo de la diagonal necesita ser guardada, resultando $n(n-1)/2$ distancias. A diferencia de los métodos anteriores cada elemento en AESA tiene el rol de pivote.

La operación de búsqueda por rango $R(q,r)$ (y similar para vecinos cercanos) selecciona un objeto aleatorio p y lo usa como pivote, la distancia de q a p es evaluada y utilizada para despreciar algunos objetos. Un objeto o puede ser despreciado si $|d(q,p) - d(p,o)| > r$. Esta condición solo utiliza distancias que han sido previamente calculadas. El algoritmo entonces selecciona otro pivote de los objetos restantes. La selección del nuevo pivote es definida

por el limite inferior $d(q,p) - d(p,o)$. Como se desea maximizar el despreciar objetos, se maximiza el limite inferior resultando en la selección del p más cercano a q. El nuevo pivote es utilizado con la condición para eliminar algunos objetos no descartados. AESA tiene un buen desempeño con respecto a otro métodos obteniendo un tiempo constante de búsqueda con respecto al tamaño de la base de datos ($O(1)$). Usa $O(n^2)$ en espacio y construcción. El tiempo extra al revisar la matriz es $O(n)$ a $O(n^2)$. Los cálculos de distancia sin embargo son mas costosos que revisar toda la matriz. AESA es recomendable para colecciones pequeñas. En contraste, búsquedas por rango con un radio grande o k-NN con una k alta requieren $O(n)$ cálculos, igual que la búsqueda lineal.

2.5.9. Linear AESA

Esta estructura resuelve el espacio cuadrático de AESA. [Micó y Oncina, 1994] LAESA conserva las distancias de los objetos sólo a un número fijo m de pivotes. Así la matriz de distancias es de $n \times m$ en lugar de $n(n-1)$ usadas en AESA. Sin embargo esto tiene nuevas complicaciones en la selección de nuevos pivotes. En [Micó y Oncina, 1994] la selección de los pivotes intenta seleccionar pivotes lo más lejanos entre si. La búsqueda es igual que en AESA a excepción de que algunos objetos no serán pivotes. El espacio y tiempo de construcción es de $O(mn)$ mientras que la complejidad de búsqueda es $m + O(1)$, bajo condiciones particulares.

2.6. Métodos de indexación basados en particiones compactas

Estos algoritmos están basados en la división del espacio (o *clustering* en Inglés) en *particiones* o *zonas* tan compactas como sea posible [**?**]. Cada zona guarda un punto representativo, llamado *centro* y unos pocos datos extras que permiten descartar zonas rápidamente en el tiempo de búsqueda, sin medir la distancia actual de los elementos de la zona al objeto de búsqueda. Al particionar el espacio se busca que los elementos de cada parte sean lo más cercanos posible entre si y que no existan demasiadas partes, por ejemplo, en el caso de obtener partes de un sólo elemento la partición quedaría igual que el espacio

y al realizar una búsqueda no se descartarían zonas de este rápidamente.

Cada zona puede particionarse recursivamente en más zonas, creando una jerarquía de búsqueda. Existen dos criterios generales para particionar el espacio: *Partición de Voronoi* y *Radio de cobertura*.

Criterio de partición de Voronoi: Un conjunto de m centros es seleccionado y el resto de los elementos son asignados a la zona del centro más cercano. Dada una consulta por rango (q, r), la distancia entre q y los m centros son calculadas. Sea c el centro más cercano a q. Cada zona del centro $c_i \neq c$ que satisface $d(q, c_i) > d(q, c) + 2r$ puede ser descartada, porque su área de Voronoi no puede tener intersección con el radio de la consulta *(query ball)*. La figura 2.2a, muestra un ejemplo del criterio de partición de Voronoi. Para q_1 la zona de c_4 puede ser descartada y para q_2 sólo la zona de c_3 debe ser visitada.

Criterio de radio de cobertura: El radio de cobertura $cr(c)$ es la distancia máxima entre el centro c y el elemento que pertenece a su zona. Dada una consulta por rango (q, r), si $d(q, c_i) - r > cr(c_i)$ entonces la zona i no puede tener intersección con el radio de la consulta y todos sus elementos son descartados. En la figura 2.2b, el radio de q_1 no tiene intersección con la zona del centro c, así que puede ser descartada. Para las esferas de las consultas q_2 y q_3, la zona no puede ser descartada porque hace intersección con estas esferas.

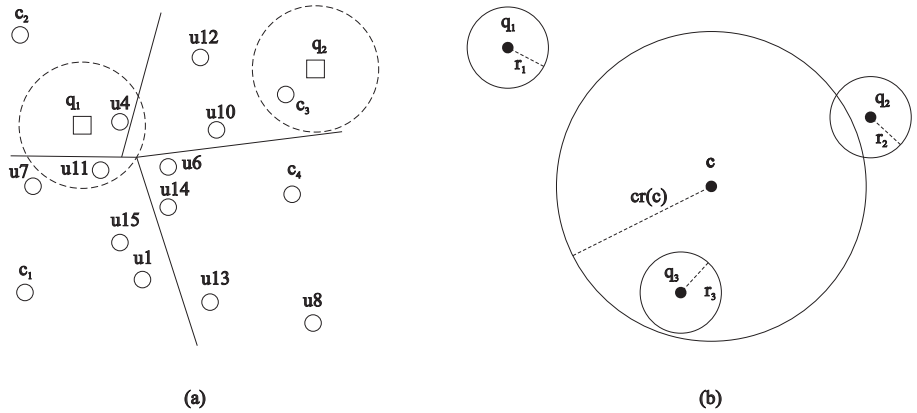


Figura 2.2: (a) Criterio de partición de Voronoi, (b) Criterio de radio de cobertura

2.6.1. Generalized Hyperplane Tree

GHT propuesto en [Jeffrey, 1991] utiliza el criterio de partición de Voronoi. GHT hace uso del hiperplano entre los pivotes p_1 y p_2 para decidir que subárbol visitar. En la búsqueda se atraviesa el árbol izquierdo si $d(q, p_1) - r \leq d(q, p_2) + r$. El subárbol izquierdo es visitado si $d(q, p_1) + r \geq d(q, p_2) - r$. Si $(d(q, p) + d(p, o)) \leq r$ se incluye o directamente en el resultado, sin calcular la distancia $d(q, o)$.

2.6.2. Bisector Tree

BST utiliza el criterio del radio de cobertura, propuesto en [Kalantar y McDonald, 1983]. Es un árbol binario construido recursivamente sobre un conjunto de datos $\mathbb{S}$ de la siguiente manera: Dos pivotes p_1 y p_2 son seleccionados a cada nodo y la partición del hiperplano es aplicada. Objetos mas cercanos al pivote p_1 que a p_2 forman el subárbol izquierdo, mientras que los objetos cercanos al pivote p_2 forman el subárbol derecho. Para cada pivote el radio de cobertura es guardado en los nodos respectivos. El algoritmo de búsqueda para una consulta por rango $R(q, r)$ revisa un subárbol si $d(q, p_i) - r$ no es más grande que el radio de cobertura r_i^c de p_i. Así una rama es descartada si la consulta no intersecta la esfera centrada en p_i con el radio de cobertura r_i^c.

Una variante del BST llamada *Monotonous Bisector Tree* [Noltemeier et al., 1992, Noltemeier H., 1993] propone que uno de los pivotes de cada nodo interno diferente a la raíz es heredado de su nodo padre. Específicamente, lo pivotes que representan el subárbol izquierdo y derecho son copiados al correspondiente hijo interno correspondiente. Esta técnica resulta en una estructura con pocos pivotes y así pocos cálculos de distancia son necesarios para ejecutar una consulta. BST está en espacio lineal $O(n)$ y requiere $O(n \log n)$ cálculos de distancia para construir el árbol.

2.6.3. VT

Voronoi Tree, es una mejora del BST [Dehne y Noltemeier, 1988]. Utiliza dos o tres pivotes en cada nodo interno y también tiene la propiedad de que el radio de cober-

tura es reducido mientras se visita el árbol hacia abajo. El autor del VT muestra que un VT balanceado puede obtenerse usando el algoritmo de inserción similar al de los B-trees [Comer, 1979].

2.6.4. M-Tree

Metric Tree, propuesto en [Ciaccia et al., 1997], puede manipular archivos que cambien de tamaño dinámicamente, lo que se convierte en una ventaja cuando las inserciones o eliminaciones de objetos son frecuentes. A diferencia de otros árboles métricos, el M-tree es construido de abajo hacia arriba dividiendo el tamaño de sus nodos fijos. Cada nodo es delimitado por una región esférica del espacio métrico. Un nodo hoja contiene un identificador del objeto, este dato es utilizado como argumento para el calculo de distancias y su distancia con un pivote que esta en el nodo padre. Cada nodo interno mantiene un nodo hijo, el radio de cobertura que limita a los objetos indexados abajo y su distancia al pivote correspondiente. Obviamente la distancia al pivote padre no tiene relevancia para el nodo raíz. El efecto de poda del árbol del algoritmo de búsqueda se maneja usando el radio de cobertura y las distancias de los objetos a sus pivotes en los nodos padres.

2.6.5. GNAT

Geometric Near-neighbor Access Tree presentado en [Brin, 1995], utiliza m pivotes en cada nodo interno. Específicamente, un conjunto de pivotes $P = \{p_1, ..., p_m\}$ es seleccionado y el conjunto $\mathbb{S}$ es dividido en $S_1, ..., S_m$ conjuntos, dependiendo de la distancia más corta a un pivote P. Es decir, para algún objeto $o \in \mathbb{S} \setminus P, o$ es un miembro del conjunto S_i si y sólo si $d(p_i, o) \leq d(p_j, o)$ para todo $j = 1, ..., m$. Aplicando este procedimiento recursivamente se construye un árbol $m - ario$.

2.6.6. SAT

Spatial Aproxximation Tree [Navarro, 1999, Navarro, 2002] está basado en partición de Voronoi, pero a diferencia del GHT y GNAT, busca una estructutra como el grafo de *Delanuay*. Dado un diagrama de Voronoi, un grafo de Dalanuay es un grafo donde cada

nodo representa una celda del diagrama de Voronoi y donde los nodos son conectados con aristas si las celdas de Voronoi son vecinas directas.
Para una colección $\mathbb{S}$ el SAT se define de la siguiente manera: Un objeto arbitrario p es seleccionado como la raíz del árbol y el conjunto más pequeño $N(p)$ de todos sus vecinos se determina por:

$$o \in N(p) \Leftrightarrow \forall o' \in N(p)\ o : d(o,p) < d(o,o')$$

Para un conjunto valido $N(p)$ (no necesariamente el más pequeño), cada objeto de $N(p)$ es cercano a p que a cualquier otro objeto en $N(p)$ y todos los objetos en $\mathbb{S} \setminus N(p)$ son cercanos a un objeto en $N(p)$ que a p. Para construir el árbol, un nodo hijo es definido para cada vecino y los objetos más cercanos al hijo se definen como se describió anteriormente. La distancia al objeto más lejano o de p también es almacenada en cada nodo.

2.6.7. List of Clusters

El índice de LC presentado por Chávez y Navarro [Chavez y Navarro, 2005] es un índice muy útil para datos con altas dimensiones. La construcción comienza tomando un "centro" $c \in \mathbb{S}$ y un radio r. Se define el radio del centro de (c, r) como el subconjunto de elementos de $\mathbb{U}$ que se encuentran a una distancia a lo mas r de c. Definiendo:

$$I_{\mathbb{S},c,cov(c)} = \{u \in \mathbb{S} \setminus \{c\} | d(c,u) \leq cov(c)\} \tag{2.6}$$

como el bloque de elementos "internos" cuyo centro es c y

$$E_{\mathbb{S},c,cov(c)} = \{u \in \mathbb{S} | d(c,u) > cov(c)\} \tag{2.7}$$

como el resto de los elementos, el proceso de construcción es recursivo sobre E y el resultado es la tripleta (c_i, r_i, I_i) a continuación se muestra el algoritmo de construcción del índice LC. Algoritmo 1

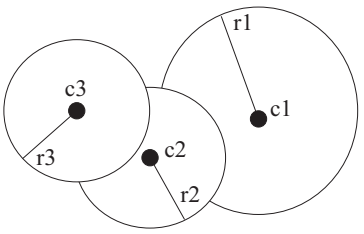
Algoritmo 1 Construcción de la Lista de Clusters

Entrada: El conjunto m de objetos a indexar. El operador "::" es el constructor de la lista.

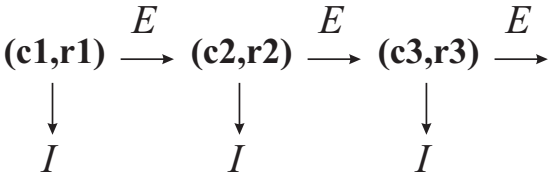
Salida: La lista de clusters

1: **Build**($\mathbb{S}$)
2: **si** $L = \emptyset$ **entonces regresar** Lista vacía
3: **si no**
4: Seleccionar $c \in \mathbb{S}$
5: Seleccionar un radio $cov(c)$
6: $I_{\mathbb{S},c,cov(c)} = \{u \in \mathbb{S} \setminus \{c\} | d(c,u) \leq cov(c)\}$
7: $E = \mathbb{S} \setminus I$
8: **fin si**
9: **regresar** $(c, cov(c), I) :: Build(E)$

La estructura construida pareciera simétrica, pero no es así. El primer centro seleccionado tiene preferencia sobre los centros posteriores en el caso de superposición de esferas. La figura 2.3a muestra todos los elementos en la esfera del primer centro c_1 son guardados en su bloque I, a pesar de encontrarse en los bloques de los centros posteriores c_2 y c_3. En la figura 2.3b se observa como los objetos se estructuran en una lista.



a) Las zonas de influencia de los tres centros tomados en el orden: ***c1,c2,c3***



b) El orden de la lista para los datos

Figura 2.3: Estructura del LC

Observación El algoritmo anterior puede representarse también de manera iterativa. En índice LC será utilizado en la implementación de este trabajo.

La selección de los centros y el radio en el índice LC no afecta el resultado correcto

de la construcción del índice, para la selección del *i-ésimo* centro se pueden utilizar las siguientes heurísticas:

(p1) Aleatoriamente
(p2) El elemento más cercano al c_{i-1} en el conjunto de elementos restantes.
(p3) El elemento más lejano al c_{i-1} en el conjunto de elementos restantes.
(p4) El elemento que minimice la suma de las distancias a los centros previos.
(p5) El elemento que maximice la suma de las distancias a los centros previos.

Para la selección de los radios de cobertura hay dos alternativas:

1) Particiones de radio fijo: Definir un radio r^* fijo para cada bloque.
2) Particiones de tamaño fijo: Definir un número fijo de elementos $m*$ para cada bloque.

El algoritmo de búsqueda en la figura 2.4 se comporta de la siguiente manera: Si el primer centro seleccionado es c y su radio de cobertura r_c, entonces la búsqueda para una consulta (q, r) inicia calculando $d(c, q)$ y agregando c al conjunto de resultados si está en el rango de búsqueda. Entonces se busca exhaustivamente en el bloque I sólo si el radio de consulta tiene intersección con el radio de cobertura del centro c. Dada la asimetría de la estructura, se puede podar la búsqueda en la otra dirección: si el radio de consulta está contenido totalmente en el radio de cobertura del centro c, no se considera E ya que por la construcción se conocen todos los elementos dentro de I.
Esta estructura permite conocer los bloques relevantes si el radio de consulta se encuentra contenido en un bloque.

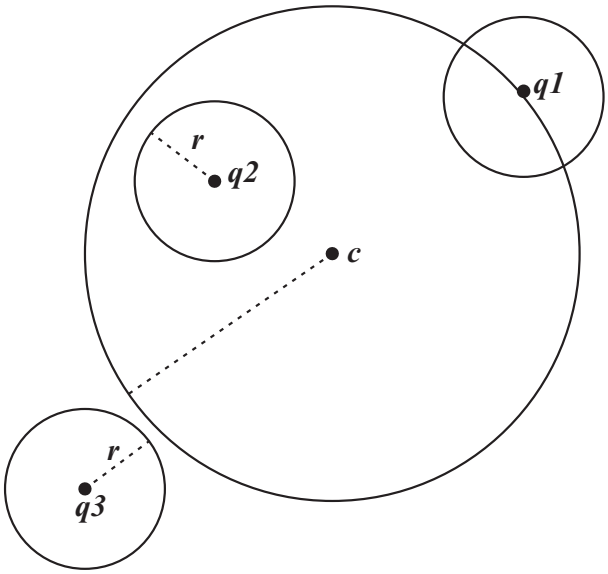


Figura 2.4: Tres casos de radio de búsqueda contra radio del centro

En la figura 2.3 se observa que para q_1 se considera el bloque actual y el resto de los centros. Para q_2 se considera el bloque actual. Para q_3 no se considera el bloque actual.

En el Algoritmo 2 se presenta el algoritmo de búsqueda del LC.

En índice LC tiene un tiempo de construcción de $O(n^2)$ y $O(n^\alpha)$ para algún $\alpha \leq 1$ cálculos de distancia en la búsqueda.

2.6.8. Rev-LC

Reverse Nearest Neighbor List of Clusters, propuesto en [Téllez y Chávez, 2012], es una mejora del LC sobre el tiempo de construcción. La idea principal es la selección de los m centros de antemano en lugar de obtenerlos recursivamente durante la construcción de la lista. La ventaja al seleccionar los centros de antemano es que se establece un orden y cada región alrededor de un centro puede asignarse de forma paralela.

La solución propuesta selecciona m centros de antemano y define un dominio de Dirichlet, como si fuera un GNAT de un nivel con alta aridad. Las tripletas del LC son separadas en tres componentes: $C = c_1, c_2, ..., c_3$ los bloques $I = I_1, I_2, ..., I_m$ y el radio de

Algoritmo 2 Búsqueda Lista de Clusters

Entrada: La lista de clusters L, la consulta $(q, r)_d$

Salida: El conjunto de resultados R

1: $R \leftarrow \emptyset$
2: **para todo** $(c, cov(c), I) \in L$ **hacer**
3: Seleccionar $c \in \mathbb{S}$
4: Seleccionar un radio $cov(c)$
5: $d_{cq} = d(c, q)$
6: **si** $d_{cq} \leq r$ **entonces**
7: $R \leftarrow \bigcup\{c\}$
8: **fin si**
9: **si** $d_{cq} \leq cov(c) + r$ **entonces**
10: **para todo** $u \in I$ **hacer**
11: **si** $d(u, q) \leq r$ **entonces**
12: $R \leftarrow R \bigcup\{u\}$
13: **fin si**
14: **fin para**
15: **fin si**
16: **si** $d_{cq} < cov(c) - r$ **entonces**
17: **regresar** R
18: **fin si**
19: **fin para**

cobertura $COV = cov(c_1), cov(c_2), ..., cov(c_m)$. Para simplificar el algoritmo I y COV son indexados con el i-ésimo número y el correspondiente i-ésimo centro c_i.

Algoritmo 3 Construcción del Rev-LC

Entrada: El número de centros m

Salida: El Rev-LC C, I y COV

1: C se inicializa seleccionando m centros aleatorios de S
2: **para todo** $i = 1$ **hasta** m **hacer**
3: $I_i \leftarrow \emptyset$
4: $COV_i \leftarrow 0$
5: **fin para**
6: **para todo** $u \in \mathbb{S} \setminus C$ **hacer**
7: Sea c_i el vecino más cercano a u en C ▷ Los empates se rompen arbitrariamente
8: $I_i \leftarrow I_i \bigcup \{u\}$
9: $COV_i \leftarrow max\{COV_i, d(c_i, u)\}$
10: **fin para**

La construcción del Rev-LC es mostrada en el Algoritmo 3. Si se supone que no hay empates para los vecinos cercanos (NN), una definición alternativa y sucinta es $I_c = \{u \in \{\mathbb{S} \setminus C\} | NN(u, C) = c\}$ y $COV_c = max\{d(c_i, u) | u \in I_c\}$, por ejemplo, los bloques son llenados con los vecinos reversos de cada $c \in C$. El algoritmo de búsqueda es muy parecido al Algoritmo 2 del LC, sólo evitando la última condición (línea 16) ya que el Rev-LC divide $\mathbb{S}$ en m bloques a diferencia de la división recursiva del LC ya que no hay un orden especial, como en la construcción del LC.

El costo de construcción del Rev-LC es $O(nm)$ en el peor caso, pero se puede lograr $O(nm^{\alpha})$ para algún $\alpha \leq 1$ dependiente de la base de datos.

2.6.9. Conclusión

En este capítulo se ha realizado un estudio analítico y extenso sobre algunas técnicas de indexación del espacio métrico primeramente se introduce el concepto de función distancia y sus propiedades. Se presenta la familia de distancias L_p comúnmente utilizada en el estudio de espacios vectoriales, dichos espacios, como se mencionó son un caso particular de los espacios métricos y es importante para este trabajo recordar su definición.

En lo correspondiente a métodos de partición de los espacios métricos, se realiza una división en dos clases; métodos de partición basados en pivotes y en particiones compactas, donde la mayoría de ellos constituyen el "estado del arte" con respecto a indexación de espacios métricos.

Dado que la propuesta de compresión anteriormente planteada se enfoca a la agrupación de un espacio métrico, en este capítulo podemos resaltar que los métodos de indexación en particiones compactas son relevantes para este estudio, podemos destacar que el índice Lista de Clusters utiliza poco espacio en memoria y tienen un buen manejo de la dimensión, además que proporciona la opción de agrupación por radio de cobertura o por número de centros. Tanto el índice LC como Rev-LC presentan buenos resultados de agrupación lo cual resulta útil en este trabajo, ya que lo que se necesita es particionar una base de datos métrica en grupos de elementos cuya similitud sea alta.

Capítulo 3

Estructuras de Datos Sucintas

Una *estructura de datos* es una manera de organizar la información representando eficientemente un conjunto de objetos y sus relaciones, donde además se definen métodos para acceder directamente a ésta. Las estructuras de datos permiten la ejecución eficiente de algoritmos sobre la información que manejan. Existen muchos tipos de estructuras y éstas tienen diferentes propiedades dependiendo de la aplicación en la que se requieren. Mientras que bases de datos pueden utilizar estructuras de tipo árbol un compilador puede hacer uso de una estructura tipo pila.

Jacobson [Jacobson, 1988] presentó las *estructuras de datos sucintas*. Este tipo de estructuras se caracterizan por hacer una representación compacta de la información. El objetivo principal de estas estructuras es poder utilizar estructuras tales como: árboles, grafos, índices de texto, en el menor espacio posible sin cambiar su funcionalidad. Es decir, a diferencia de los algoritmos de compresión sin pérdida, las estructuras sucintas conservan la habilidad de manejar la información sin descomprimirla.
Suponiendo que $\mathbb{Z}$ es la representación optima en bits necesaria para almacenar alguna fuente. La representación de esta fuente es llamada [Damaine, 2012]:

- **Implícita:** Si la representación ocupa $\mathbb{Z} + O(1)$ bits de espacio,
- **Sucinta:** Si la representación ocupa $\mathbb{Z} + o(\mathbb{Z})$ bits de espacio y
- **Compacta:** Si toma $O(\mathbb{Z})$

3.1. Secuencias de Bits

Un bit es la representación mínima de un dígito en el sistema binario y puede tomar dos valores: 0 o 1, a un conjunto de bits ordenados en sucesión le llamamos secuencia, cadena, mapa de bits o bitmap[1].

La entropía de orden cero H_0 de una secuencia de bits se define como [Navarro, 2007]:

$$H_0(B) = \frac{n_0}{n}\log_2\frac{n}{n_0} + \frac{n_1}{n}\log_2\frac{n}{n_1} = \frac{1}{n}\log_2\binom{n}{n_0} + O(\frac{\log_2 n}{n}) \tag{3.1}$$

La definición anterior hace una aproximación al limite teórico de la entropía, dada la siguiente observación; partiendo de una secuencia B de n bits $B[1,..,n]$, donde hay n_0 número de ceros y n_1 número de unos. Tal que:

$$n_0 + n_1 = n = |B|$$

Sea p la probabilidad de ceros en B:

$$p = \frac{n_0}{n}$$

Sea q la probabilidad de unos en B:

$$q = \frac{n_1}{n} = 1 - p$$

Como se definió en el Capítulo 1, la entropía es la medida de la incertidumbre de la información que se representa en una cierta base.

Entropía de orden cero [Shannon, 1948]:

$$H_0(x) = -p\log(p) + q\log(q) \tag{3.2}$$

Una secuencia o bitmap B de longitud n puede originar una combinación de n tomadas de np secuencias de bits como resultado de la modificación de np bits cualesquiera en B.

[1]Una cadena de bits que contiene ceros y unos recibe el nombre en Inglés de : bitmap, bitset, bit string, o bit vector

$$\binom{n}{np} = \frac{n!}{(n-np)!(np!)}$$

Aplicando la aproximación de Stirling [Hazewinkel, 2001][2] para el factorial: $n! \approx (\frac{n}{e})^n\sqrt{2\pi n}$ resulta:

$$\binom{n}{np} \approx \frac{(n/e)^n}{(np/e)^{np}(nq/e)^{nq}} \frac{\sqrt{2\pi n}}{\sqrt{2\pi np}\sqrt{2\pi nq}}$$

Donde: $q = (1-p)$ y $(n-np) = nq$

Simplificando:

$$\begin{aligned}
\binom{n}{np} &\approx \frac{\frac{(e^n)}{(n^n)}\frac{n^n}{e^n}}{\frac{(e^n)}{(n^n)}\frac{n^{np}p^{np}}{e^{np}}\frac{n^{nq}q^{nq}}{e^{nq}}} \frac{\sqrt{2\pi}\sqrt{n}}{\sqrt{2\pi}\sqrt{n}\sqrt{2\pi npq}} \\
&\approx \frac{1}{n^{np-n}e^{n-np}p^{np}\frac{n^{nq}q^{nq}}{e^{nq}}} \frac{1}{\sqrt{2\pi npq}} \\
&\approx \frac{1}{n^{-nq}e^{nq}p^{np}\frac{n^{nq}q^{nq}}{e^{nq}}} \frac{1}{\sqrt{2\pi npq}} \\
&\approx \frac{1}{p^{np}q^{nq}} \frac{1}{\sqrt{2\pi npq}}
\end{aligned}$$

Tomando logaritmo base dos:

$$\begin{aligned}
\log_2\binom{n}{np} &\approx \log_2(1) - (np\log_2(p) + nq\log_2(q)) + \log_2(1) - \frac{\log_2(2\pi npq)}{2} \\
&\approx -n(p\log_2(p) + q\log_2(q)) - \frac{\log_2(2\pi npq)}{2}
\end{aligned}$$

Dividiendo entre n:

$$\begin{aligned}
\frac{1}{n}\log_2\binom{n}{np} &\approx -(p\log_2(p) + q\log_2(q)) - \frac{\log_2(2\pi npq)}{2n} \\
&= H(p) + O(\frac{\log_2(n)}{n}) \qquad (3.3)
\end{aligned}$$

[2] Aproximación de Stirling. Encyclopedia of Mathematics. URL: http://www.encyclopediaofmath.org

Observamos que cuando n tiende a infinito:

$$\frac{1}{n}\log_2\binom{n}{np} + O(\frac{\log_2(n)}{n}) = H(p)$$

Para $0 \leq p \leq 1$ y donde H(p) es la entropía de orden cero.

Secuencias de símbolos

Dada una secuencia de símbolos S sobre un alfabeto Σ y sea n_c el número de ocurrencias del símbolo c en S, la entropía de orden cero de una secuencia de símbolos se define:

$$H_0(S) = \sum_{c \in S} \frac{n_c}{n} \log_2 \frac{n}{n_c} \tag{3.4}$$

3.1.1. Secuencias comprimidas

El espacio de una representación sucinta $n + o(n)$ de un bitmap B con tamaño n, puede mejorarse al notar que hay $\binom{n}{n_1}$ distintas subsecuencias de n con exactamente n_1 unos, y por lo tanto se puede representar B usando $\lceil \log \binom{n}{n_1} \rceil$ bits. Al comprimir esta representación se utilizan $nH_0(B) + O(\frac{n}{\log n})$ bits [Navarro, 2007] es decir, la información de una secuencia de bits puede compactarse con un orden de compresión cercano a la entropía.

La importancia de la representación de una secuencia de bits en una estructura sucinta radica en que se conserve el acceso a los datos originales, utilizando las operaciones principales definidas para este tipo de estructuras.

3.2. Operaciones principales de las estructuras sucintas

Las estructuras sucintas ocupan un espacio cercano a la entropía de orden cero. Las principales operaciones que las estructuras sucintas realizan sobre una secuencia de n bits $B[1, .., n]$ son **rank**, **select** y **access**:

- $rank_b(B, i)$: El número de veces que aparece el bit b hasta la posición i.
- $select_b(B, i)$: La posición del $i - ésimo$ bit b en B.

- $access(i)$: Devuelve el bit que se encuentra en la posición i.

En la figura 3.1 se observa un ejemplo de una secuencia de bits y las operaciones *rank*, *select* y *access*.

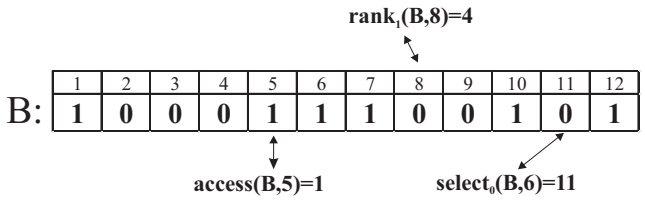


Figura 3.1: Ejemplo de *rank*, *select* y *access*

Estas operaciones tienen las siguientes propiedades simples:

- $rank_0(B, i) = i - rank_1(B, i)$.
- $B[i] = rank_1(B, i) - rank_1(B, i - 1)$
- $rank_b(B, select_b(B, i)) = i$.
- Si $B[i] = b$, $select_b(B, rank_b(B, i)) = i$.
- En general, $select_b(B, rank_b(B, i)) \leq i$.

Existe una gran cantidad de estructuras que realizan las operaciones *rank*, *select* y *access*. El estudio de las estructuras sucintas busca modelar estructuras que puedan evaluar las operaciones *rank*, *select* y *access* en tiempo o espacio mínimo. En la siguiente sección se mencionan algunas estructuras de datos sucintas más comunes.

3.3. Estructuras sobre secuencias de bits

3.3.1. GGMN

Presentada por González, Grabowski, Mäkinen y Navarro [González et al., 2005]. Esta estructura realiza *rank* en un tiempo $O(1)$ y *select* en $O(\log n)$. La idea es guardar los valores *rank* en 32 bits, cada $s = 20 \times 32 = 640$ bits y entonces revisar un byte a la vez y los 640 bits (80 bytes) usando una tabla precalculada. Para hacer *select* se hace una

búsqueda binaria en los valores muestreados y después realiza una revisión secuencial. Esta estructura es muy rápida para realizar las operaciones *rank* y *select*, requiere n bits, es decir, no reduce el espacio del bitmap, sin embargo la estructura resulta óptima cuando el número de bits ceros es muy parecido al número de bits unos.

3.3.2. SArray

Sparce Array presentado por Okanohada y Sadakane [Okanohara y Sadakane, 2006], la estructura está basada en el número m de unos que contiene la secuencia y que resulta óptimo cuando el m es pequeño. El método de codificación es el mismo que el propuesto por [Grossi y Vitter, 2005] sólo que con una representación más pequeña e implementación de la operación *rank*. Dada una secuencia de bits $B[1...n]$ con m unos tal que, $m << n$ se define $x[1, ...m]$ tal que $x[i] = select(i, B)$ para $1 \leq i \leq m$.
Dado un parámetro $t = \lceil \log(\frac{n}{m}) \rceil$, cada x es dividida en z bits altos y bajos $w = t$.

Los bits bajos son almacenados explícitamente en L usando $m \cdot w$ bits. Los bits altos son representados por un bitmap $H[1...2m]$ tal que $H[x_i/2^w + i] = 1$ y los otros valores son 0.

Usando H y L se puede calcular *select* haciendo: $select(i, B) = (select(i, H) - i) \cdot 2^w + L[i]$

El tamaño de H y L es $m + t + m(\log(\frac{n}{t}))$ bits cuando $t = m \log e \simeq 1.44m$. El tamaño entonces es: $1.92m + m \log \frac{n}{m}$ bits cuando $t = 1.44m$. Suponemos entonces que H es denso por que hay m unos y $t = 1.44m$ ceros en H.

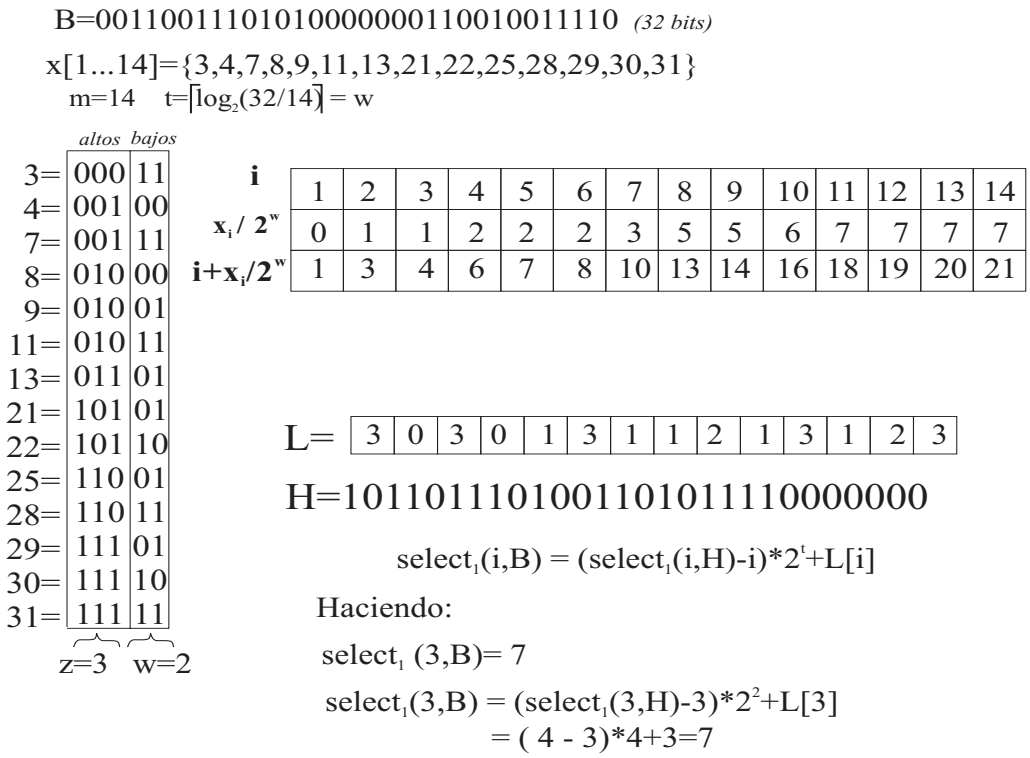


Figura 3.2: Codificación del SArray

La figura 3.2 es un ejemplo de SArray, x son las posiciones de unos en B.

3.3.3. DArray

Dense Array también propuesto por Okanohada y Sadakane [Okanohara y Sadakane, 2006] para un bitmap denso, $B[0...n-1]$ con $m \backsimeq \frac{n}{2}$ unos. Primeramente se particiona B en bloques de tal manera que cada bloque contenga L unos. Sean $P_l[0...\frac{n}{L-1}]$ arreglos tal que $P_l[i] := select(iL, B)$. Se clasifican estos bloques en dos grupos: si el tamaño de un bloque $(P_l[i] - P_l[i-1])$ es mayor que L_2, se almacenan todas las posiciones de unos explícitamente en S_l. Si el tamaño de un bloque es menor que L_2, se almacenan las posiciones de iL_3-ésimo unos $(i = 0...\frac{L_2}{L_3})$ en S_s. Estos valores se pueden almacenar en $\log L_2$ bits guardando únicamente valores relativos a los ya guardados para P_l.

Para hacer $select(i, B)$, se revisa $P_l[\lceil \frac{i}{L} \rceil]$ y se evalua si el block es mas grande que

L_2. Si así es se revisa el valor en S_l el cual es guardado explícitamente. En caso contrario se busca el valor correspondiente L_3-ésimo valor en S_s y se realiza una búsqueda secuencial en el bloque en un tiempo $O(\frac{L_2}{\log n})$ ya que se pueden leer $O(\log n)$ bits en RAM. A diferencia del SArray el DArray es mejor con conjuntos densos. Requiere el mismo espacio que GGMN, $n + o(n)$, realiza *select* en $O(\frac{\log^4 m}{\log n})$ y el *rank* en $O(1)$.

3.4. Secuencias Arbitrarias

A una secuencia de símbolos de un alfabeto Σ con tamaño σ, puede realizarse también las operaciones *rank*, *select* y *access*. [Claude y Gonzálo, 2008]. Existen varias estructuras para realizar estas operaciones:

3.4.1. Wavelet Tree

[Grossi et al., 2003, Navarro y Mäkinen, 2007] Es una estructura que almacena cadenas en un espacio comprimido. El árbol se define por la partición recursiva del alfabeto en conjuntos pares; las hojas corresponden a un sólo símbolo del alfabeto y cada nodo a un vector que almacena si un símbolo de la cadena pertenece a un conjunto o a otro. Para un alfabeto Σ de tamaño $\sigma = |\Sigma|$ una cadena $s \in \Sigma^*$ puede representarse en $nH_0(s)+o(|s| \log \sigma)$ donde $H_0(s)$ es la entropía de orden 0 de s. Si el árbol está balanceado las operaciones *rank*, *select* y *access* pueden realizarse en un tiempo $O(\log \sigma)$.

3.4.2. Golynski Munro Rao

Presentada en [Golynski et al., 2006], puede realizar las consultas *rank*, *select* y *access* en un tiempo $O(\log \log \sigma)$ usando un espacio de $n \log \sigma + n\ o(\log \sigma)$ bits. La idea principal es reducir el problema sobre una secuencia a $(\frac{n}{\sigma})$ fragmentos de tamaño σ. Para cada símbolo se concatenan, en unario, el número de sus ocurrencias en cada fragmento y posteriormente concatenar estas secuencias para todos los símbolos en un bitmap B. Compuesto con *rank* y *select* estructuras, el tamaño total de B es $2n + o(n)$ bits.
Con B es posible evaluar *rank* y *select* hasta alcanzar granularidad en el fragmento. Cada fragmento guarda σ caracteres usando un bitmap X y una permutación π. X almacena la

cardinalidad de cada símbolo del alfabeto en el fragmento, utilizando la misma codificación que en B. π almacena la permutación obtenida al ordenar la secuencia representada por el frangmento y utiliza una estructura de datos que permite cálculos de π^{-1} en un tiempo $O(\log\log\sigma)$ [Munro et al., 2003]. Esto agrega más de $2n+n\log\sigma+n\ o(\log\sigma)$ bits y permite completar todas las consultas *select* en tiempo constante y las consultas *rank* y *access* en $O(\log\log\sigma)$. La última complejidad necesita un *Y-Fast Trie* en cada fragmento para buscar π hacía la posición de interés entre las correspondiente ocurrencia de un símbolo en el fragmento. Esta estructura tiene un mejor comportamiento en alfabetos grandes.

3.5. Conclusión

En este capítulo se realiza una breve revisión de las estructuras sucintas sobre secuencias de bits y secuencias arbitrarias de símbolos. Las operaciones principales que estas estructuras son capaces de hacer son: *rank*, *select* y *access*, se observa además que estas estructuras pueden manipularse en memoria principal ya que algunas de ellas están diseñadas para comprimir el espacio de las secuencias sobre las que se construyen. El orden de compactación que algunas estructuras alcanzan es cercano a la entropía de orden cero. Con el avance de las tecnologías principalmente en el desarrollo de las jerarquías de memoria las estructuras sucintas brindan una opción favorable para el desarrollo de esquemas de compactación y compresión.

De las estructuras presentadas en este capítulo resultan importantes el SArray y el GGMN pues por su definición serán útiles para la representación de la base de datos comprimida, principalmente porque la propuesta de codificación buscará que estas estructuras funcionen en su forma óptima.

Capítulo 4

Compresión de una Base de Datos Métrica

Como se describió anteriormente, el presente trabajo pretende definir una técnica bajo un esquema de compresión sin pérdida sobre una base de datos métrica. Recordando la definición de espacio métrico y el caso particular de un espacio métrico vectorial, donde los elementos se identifican con coordenadas, podemos observar que si sobre este espacio se define una métrica de distancia de la familia *Lp* o *Minkowski* se percibe una similaridad entre un par de objetos de este espacio cuando la función distancia es pequeña (objetos cercanos) a su vez un par de objetos es menos similar cuando la función distancia es grande (objetos lejanos).

En la siguiente figura se muestra un ejemplo de dos pares de objetos en un espacio métrico de dos dimensiones.

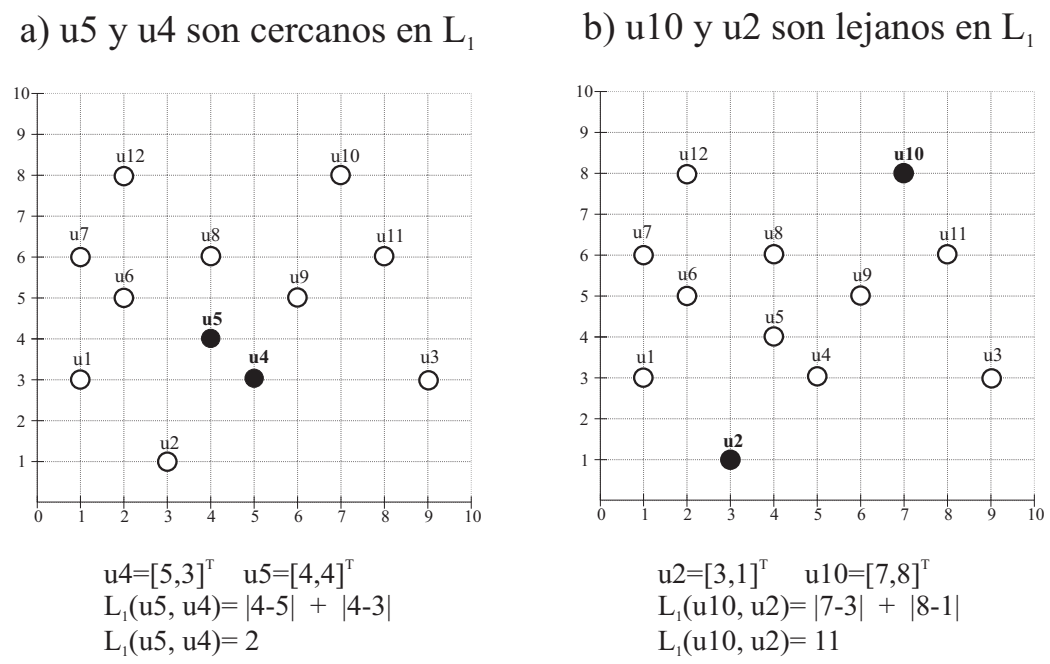


Figura 4.1: Objetos cercanos y lejanos en un espacio métrico de dos dimensiones

En la figura 4.1 se observan dos casos: **a)** La distancia entre el objeto *u5* y *u4* es pequeña por lo que los elementos son cercanos o similares, **b)** La distancia entre los objetos *u2* y *u10* es grande lo que indica que no son cercanos en L_1. Cuando utilizamos las distancias *Lp* en un espacio métrico la función distancia entre un par de objetos es inversamente proporcional a la similitud entre ellos.

Si un espacio métrico es agrupado o particionado de tal manera que se crean zonas o grupos donde los elementos son muy cercanos, se observa que las diferencias entre los elementos de cada grupo son pequeñas. De esta manera si en cada grupo se selecciona un representante y en una nueva representación de la base de datos los elementos de cada grupo son identificados implícitamente con la distancia a su representante como en la figura 4.2 se puede obtener una representación más pequeña. Para ello es necesario tomar en cuenta que con solo almacenar la distancia en la nueva representación no es posible obtener el elemento implícito, es decir, es necesario conservar el vector de diferencias. Este vector o vectores de diferencias puede entonces codificarse de alguna manera que no utilice el mismo espacio en memoria que un vector de un elemento explicito.

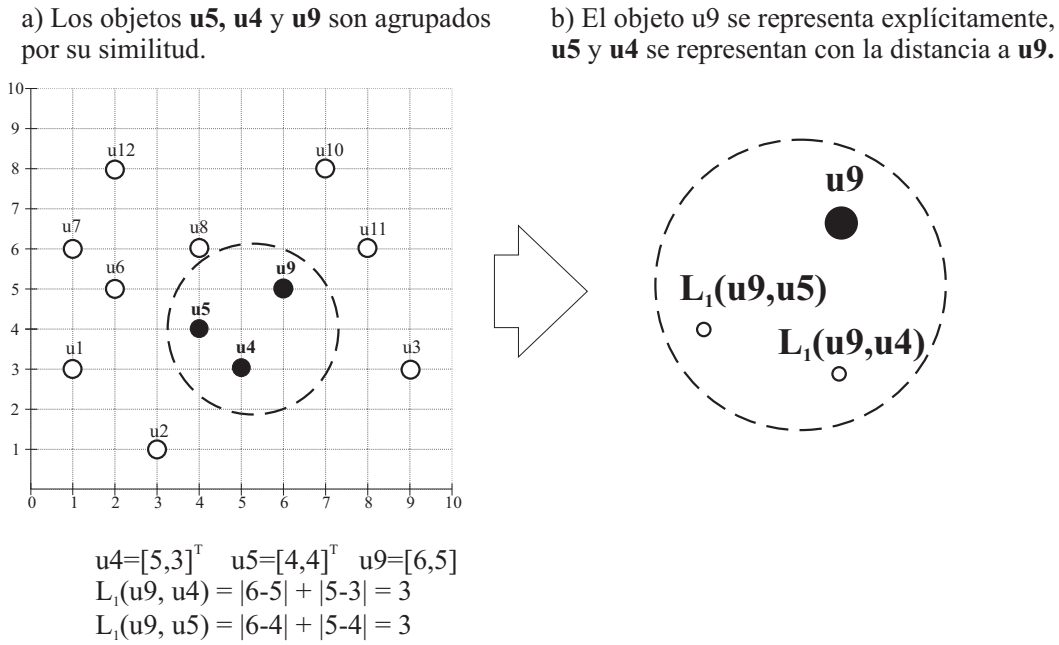


Figura 4.2: Representación de objetos implícita de objetos en base a la distancia.

En la figura 4.2 se observa un ejemplo de la agrupación de tres objetos por su similitud, posteriormente al seleccionar un representante el resto de los objetos será representado implícitamente por la distancia a su representante.

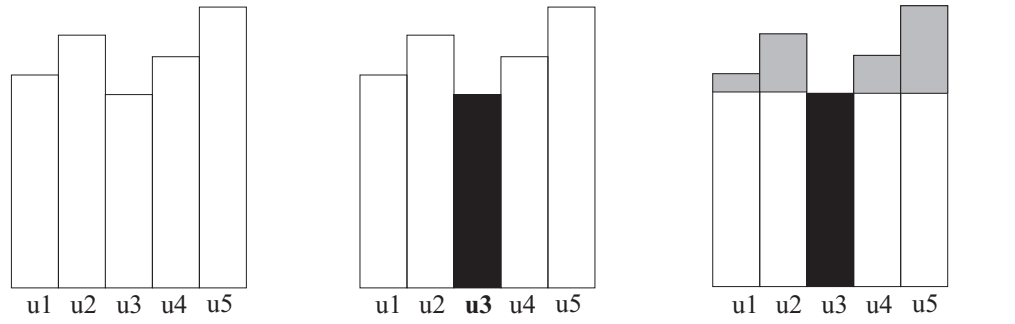
4.1. Técnica de Compresión

Suponiendo que la base de datos métrica $\mathbb{S}$ es particionada en grupos de elementos muy cercanos y cada grupo tiene un representante, una representación compacta de la base de datos se efectúa de la siguiente manera:

1 Agrupar los elementos de $\mathbb{S}$ en grupos, de tal manera que cada grupo contenga elementos cercanos, seleccionando a un elemento como *representante* del grupo.

2 En cada grupo se remplazará cada elemento con la distancia al elemento representante correspondiente.

Para ilustrar mejor esta propuesta, la figura 4.3a muestra un grupo de elementos muy parecidos (ilustrando a los objetos como barras de información), el grupo ilustrado tiene asignado un representante en este caso $u3$ (figura 4.3b) las diferencias del resto de los elementos con el representante se marcan con tono gris (figura 4.3c).



a) Grupo de elementos similares **b) u3 es el representante del grupo** **c) Diferencias de los elementos con u3**

Figura 4.3: Grupo de objetos similares y su diferencia con el representante

Observando la figura 4.3, se deduce que una representación compacta que incluya al representante explícitamente y al resto de los elementos implícitamente (utilizando sus diferencias al representante) se reduce el espacio para representar una base de datos métrica.

4.2. Selección de los representantes

La partición (o *clustering*) del espacio en una base de datos métrica con una distancia dada, representa uno de los problemas en recuperación de información que más se ha estudiado, ya que de una buena agrupación de los elementos depende el desempeño de muchas aplicaciones de búsqueda. En el caso de este esquema de compresión planteado, se busca un algoritmo de agrupamiento sobre un espacio métrico vectorial que resulte útil para la compresión y que además añada capacidades de búsqueda por similitud a la base de datos métrica. Esta propuesta utiliza el índice LC propuesto en [Chavez y Navarro, 2005] visto en el Capítulo 2 para lograr la agrupación de los elementos y a su vez hacer la selección de los representantes.

El índice LC sobre el espacio métrico puede construirse en base a un radio de cobertura o a un tamaño de bloque fijo. En este caso se propone utilizar un tamaño de bloque fijo ya que se obtienen mejores resultados en la experimentación.

Al crear el índice se obtiene:

- Una lista de m elementos centros, donde $m = \frac{|\mathbb{S}|}{n}$ siendo n el tamaño de bloque.
- El conjunto de tripletas (centro, radio de cobertura y conjunto de elementos del grupo) $(\vec{c_i}, r_i, I_i)$ donde $i = 1$ *hasta* m.

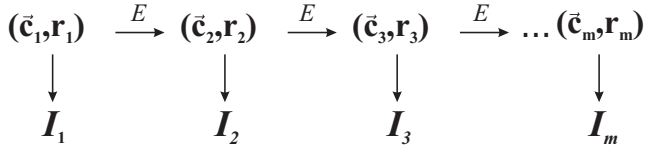


Figura 4.4: Construcción del índice LC

Usando el índice LC (figura 4.4) la representación implícita sobre los elementos de los subconjuntos I se realizará tomando como representante a los elementos *centro* $(\vec{c})$ de cada grupo los cuales son seleccionados de forma aleatoria.

4.3. Definición de la métrica y codificación

De la familia de distancias de Minkowski L_p (ecuación 2.2) se usará la distancia L_1 (ecuación 2.3) o distancia de Manhatan. Esta métrica es utilizada para crear el espacio métrico usando LC, posteriormente permitirá hacer la representación implícita de los elementos de cada grupo o bloque creados por el índice.

4.3.1. Codificación de las distancias

Una vez que los elementos de $\mathbb{S}$ han sido agrupados y se conoce el centro $\vec{c_i}$ de cada i-ésimo grupo, se realiza el cálculo de la distancia de los elementos de I_i a $\vec{c_i}$ utilizando la métrica L_1.

Sean:

$|\mathbb{S}|$ el número de elementos de la base de datos.

m el número de centros.

$\vec{c_i}$ un vector dado llamado centro del grupo i.

I_i el i-ésimo conjunto de elementos donde $\vec{c_i}$ es el vector representante o centro.

$\vec{u}$ un vector cualquiera contenido en I_i, $\vec{u} \in I_i$

dim la dimensión (número de coordenadas) de los elementos de la base de datos.

En cada grupo se realiza la diferencia de los vectores centro $\vec{c}$ al resto de los elementos $\vec{u}$ de su grupo obteniendo un vector de diferencias $\vec{p}$ utilizando la siguiente expresión:

$$\vec{p} = \vec{c} - \vec{u}$$

donde:

$$p_j = u_j - c_j \quad j = 1, 2, ...dim$$

Para hacer la representación de L_1 se harán dos codificaciones utilizando dos bitmaps:

- **Bitmap de diferencias** B_D: En este bitmap se codificará cada vector de diferencias $\vec{p}$ evaluado. Es decir se construirá con la codificación de todos los vectores de diferencias evaluados en cada grupo entre su centro y el resto de sus elementos. **Codificación del vector de diferencias:**

  - Para cada p_j hasta p_{dim}:

    - Escribir un 1 en B_D
    - Escribir el valor absoluto de p_j usando ceros en B_D.

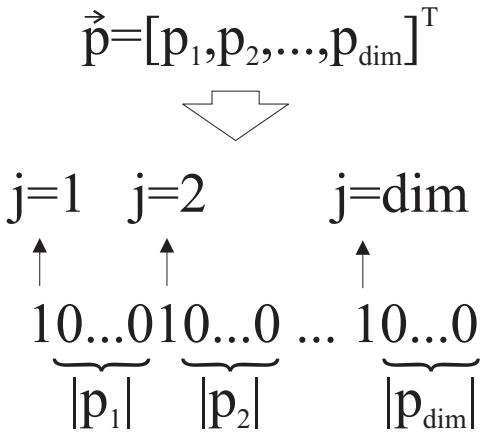


Figura 4.5: Codificación del vector de diferencias

En cada codificación de un vector de diferencias se representa $L_1(\vec{u}, \vec{c})$; es decir, el número de ceros de B_D corresponde a la distancia L_1 entre $\vec{u}$ y $\vec{c}$. Mientras que el número de unos a la dimensión del vector $\vec{u}$.

Construcción del bitmap de diferencias B_D

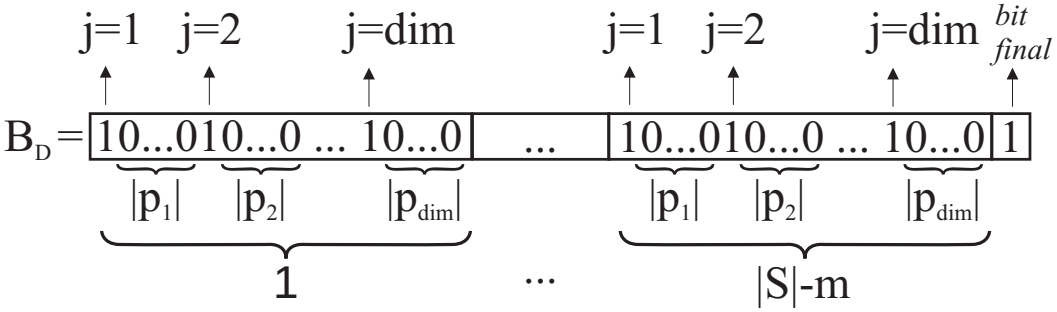


Figura 4.6: Representación de las diferencias de cada vector de diferencias evaluado

Una vez que todos las diferencias son evaluadas y codificadas en B_D se termina de construir el bitmap poniendo un bit 1 al final de la cadena. (Figura 4.6)

- **Bitmap de signos** B_S: A la vez que se construye el bitmap B_D se construye el bitmap de signos B_S, como se observa al construir el bitmap B_D no se conserva el signo de cada coordenada del vector de diferencias, este signo se guardará en B_S de la siguiente manera: si la diferencia es negativa se representará con un bit usando un

uno, si es positiva; con un bit usando un cero.

$$B_s += \begin{cases} 0 & si \;\; p_j \geq 0 \\ 1 & si \;\; p_j < 0 \end{cases}$$

Figura 4.7: Representación del vector de diferencias

B_S es una secuencia de bits que contiene el tipo de diferencia (positiva/negativa) por cada coordenada y que servirá para reconstruir los elementos implícitos.

Para ilustrar el proceso de codificación tenemos el siguiente ejemplo:

Ejemplo 1 Supongamos que tenemos vectores de cuatro coordenadas enteras y cada coordenada tiene un tamaño de 2 Bytes, necesitamos 8 Bytes para representar cada vector.

$$\vec{c} = [10708, 28500, 30584, 12566]^T$$

$$\vec{u} = [10700, 28502, 30587, 12561]^T$$

Calculamos L_1 entre c y u, guardaremos también el signo de la diferencia.

$$\overrightarrow{(u-c)} = \vec{p} = [\ -8,\ +2,\ +3,\ -5\ ]^T$$

La codificación de $\overrightarrow{(u-c)}$ se hará en dos partes:

1. Una cadena de bits B_D con los valores absolutos de cada coordenada, usando ceros separados por un uno para indicar la coordenada.

2. Una cadena de bits B_S que indica si el valor de cada coordenada es positivo o negativo usando 1 si el valor es negativo y 0 si es positivo.

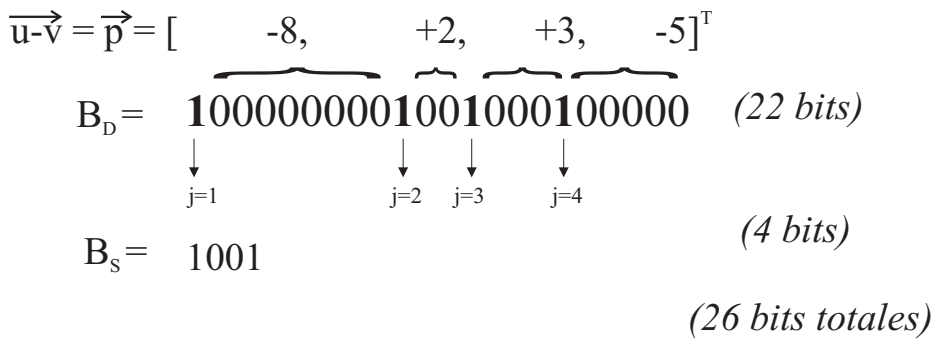


Figura 4.8: Representación de la distancia entre un elemento y su centro

Podemos observar que con esta codificación en lugar de 64 bits para representar el vector $\vec{u}$ estamos utilizando 24 bits para representarlo en base a la distancia de $\vec{u}$ al centro $\vec{c}$ ahorrando 40 bits.

La codificación de los bitmaps B_D y B_S **no es la representación final** de la base de datos, como se vio en el Capítulo 3 existen estructuras sucintas que permiten la manipulación de secuencias de bits algunas de ellas incluso compactan tal información, más adelante se describirá el tipo de estructura sucinta que se utilizará para los bitmaps creados.

4.3.2. ¿Por qué utilizar L_1?

Es importante mencionar que el ahorro de bits depende de la cercanía de los elementos ya que entre más cercanos la representación propuesta ocupa menos espacio en disco. La codificación de los vectores de diferencias que se propone puede realizarse con cualquier distancia de la familia L_p por ejemplo L_2 ecuación (2.4), sin embargo de acuerdo a la definición de esta distancia al codificar el vector de diferencias se necesitaría utilizar el cuadrado de la diferencia de las coordenadas, por lo que el número de ceros que se utiliza para representar cada coordenada aumentaría al buscar una representación compacta resulta mejor utilizar la distancia L_1. Con esta codificación para evaluar la distancia L_2 o L_3 de cualquier elemento a su centro, sólo será necesario elevar a la $\vec{p}$ correspondiente el valor de cada coordenada.

Ejemplo 2 Para ilustrar mejor esta representación sobre un espacio métrico, supongamos que el índice LC nos permite una representación de la base de datos como la de la siguiente figura: 4.9

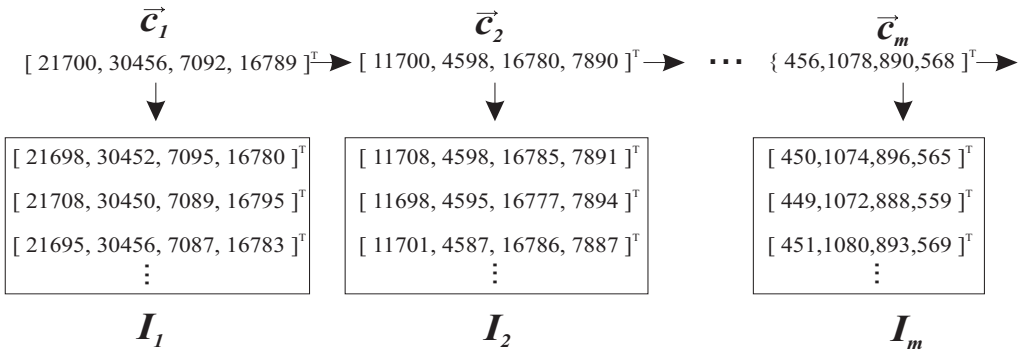


Figura 4.9: Ejemplo de agrupación sobre un espacio métrico

Los datos del grupo 1 compuesto por c_1 e I_1 explícitamente utilizan 256 bits, ya que se usan 2 bytes por coordenada, utilizando la codificación propuesta, el grupo 1 se representa usando 149 bits.

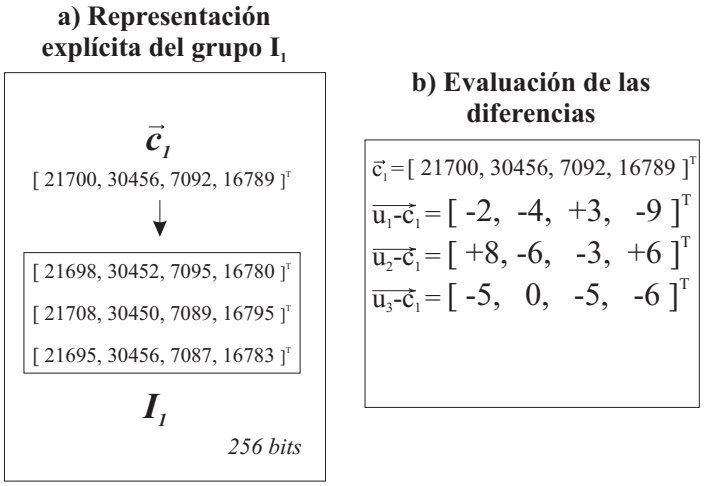
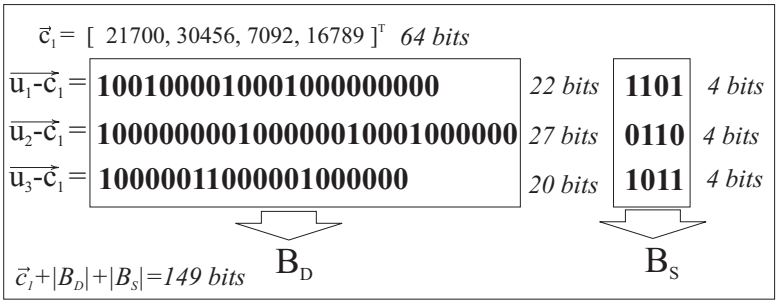


Figura 4.10: Ejemplo de representación explicita de un bloque (a) y evaluación de las diferencias de los elementos del bloque a su centro (b.)



c) Representación implícita del grupo 1

Figura 4.11: Ejemplo de representación implícita de un bloque sobre un índice LC

En la figura 4.11 se observa un ahorro de 107 bits en la codificación del grupo 1, la representación implícita del grupo 1 se compone por B_D , B_S y el centro c_1.
El ejemplo anterior muestra el número de bits que se pueden ahorrar al hacer la codificación del vector de diferencias, sin embargo esta codificación puede compactarse utilizando una estructura sucinta.

La codificación de los vectores de diferencias se realiza siguiendo la permutación del índice LC construido sobre la base de datos métrica, esta permutación indica en que orden los elementos de la base de datos forman la lista de clusters. Mientras se realiza esta codificación se construye un nuevo bitmap mismo que servirá para la descompresión de la base de datos, para construir este bitmap mientras se sigue la permutación del LC se sigue lo siguiente:
Construcción bitmap tamaño de bloque B_G

- Si el vector actual es un centro escribir un uno en B_G.

- Si el vector actual no es centro escribir un cero en B_G.

De esta manera se construye un bitmap que ayudará a conocer los vectores de diferencias que pertenecen a un grupo o a otro en el bitmap de diferencias.

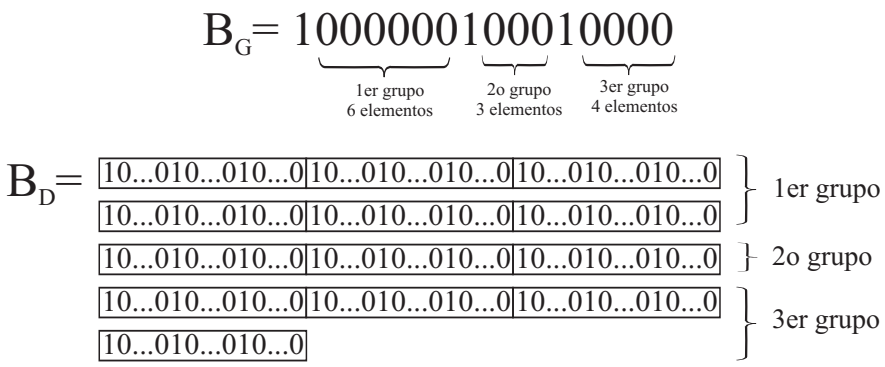


Figura 4.12: Correspondencia del bitmap de tamaño de bloque y el bitmap de diferencias

4.3.3. Estructura Sucinta

Para manipular y hacer una representación sucinta del bitmap de diferencias B_D y el bitmap de tamaño de bloque B_G se construyen un SArray y un GGMN, estas representaciones permitirán posteriormente decodificar los datos comprimidos.

Estructura SArray sobre el bitmap de diferencias

El uso de una estructura sucinta para la representación de las distancias es muy importante, pues permite mantener la codificación propuesta y acceder a los datos. El SArray, como se mencionó en el capítulo 3, permite crear una representación sucinta y es óptima cuando el bitmap contiene en su mayoría el bit cero, con una complejidad espacial de $1.92m + \log_2(\frac{n}{m}) + o(m)$ bits, donde m es el número de unos y m el número total de bits de un bitmap. Una vez que el bitmap de diferencias B_D es creado al codificar los vectores de diferencias se pueden definir las características de la información representada. Sea B_D el bitmap que representa las diferencias de los objetos a su correspondiente centro, dim la dimensión de la base de datos, $n0$ el número de ceros de B_D, $n1$ el número de unos de B_D y $|B_D| = n0 >> n1$ el número ceros y unos que contiene B_D. De acuerdo al esquema propuesto, el número de unos que contiene el bitmap B_D en la representación codificada de

una base de datos $\mathbb{S}$ particionada en m grupos es:

$$B_{D_{n1}} = (\mathbb{S} - m) * dim + 1 \tag{4.1}$$

El número de bits unos en la representación de diferencias B_D es igual al número de elementos de $\mathbb{S}$ que no son centros por la dimensión (número de coordenadas) más uno (que es el último bit para la codificación). El número de bits ceros en la representación de diferencias B_D resulta ser la sumatoria de las distancias L_1 evaluadas entre cada elemento a su centro correspondiente. Suponiendo que el número de ceros será mayor la implementación del SArray debe resultar óptima.

Estructura GGMN sobre el bitmap de tamaño de bloque

El bitmap de tamaño de bloque B_G guarda a medida que recorre la base de datos en el algoritmo de compresión un 1 si el elemento es centro y un 0 si no lo es. Si $n1$ es el número de bits unos en B_G y $n0$ el número de ceros, la representación total $|B_G| = |\mathbb{S}|$ bits. Utilizando GGMN sobre este bitmap se pueden realizar operaciones de *rank* y *select* eficientemente, donde la representación de bits es explicita. Posteriormente a estas representaciones, accederemos a los datos utilizando el índice de secuencias del LC. Con el uso de las estructuras sucintas se puede acceder a cualquier elemento de la base de datos, para ello se necesitará el número de elemento (o identificador), utilizando las operaciones *rank* y *select* descritas anteriormente.

Contenido de la representación compacta

Una vez que la base de datos se ha comprimido se tienen los siguientes datos:

- El índice LC y su permutación.
- Una lista de los centros C representados explícitamente.
- Un SArray SA_{B_D} que contiene la cadena de diferencias.
- Un GGMN $GGMN_{B_G}$ que contiene la cadena de tamaño de bloques.
- El bitmap B_S que contiene la representación de los signos.

Para acceder a cualquier elemento de la base de datos usando la representación anteriormente descrita, primero se buscará un elemento usando su identificador en el índice LC, ya que el algoritmo de compresión hace la representación en el orden del LC. Usando el índice LC se realiza el cálculo del orden de compresión y se calcula la posición de la representación del vector en la cadena de diferencias y de signos.

4.4. Algoritmo de compresión propuesto

En el Algoritmo 4 se representa el proceso de compresión de la base de datos que se propone en esta tesis. La idea es que siguiendo la permutación de el índice LC, se tome cada uno de los centros y se recorra el número total de elementos de este bloque, evaluando la distancia L_1 del elemento $\vec{u}$ a su centro $\vec{c}$, entonces la versión compacta de la base de datos contendrá los elementos centrales representados explícitamente y los elementos de cada bloque representados conforme a la distancia L_1 a su centro correspondiente.

Conforme se recorre el índice LC, se crea el bitmap **Tamaño de Bloque** B_G donde se indica con un bit 1 si el elemento actual es un centro y con un bit 0 en caso contrario.

En la línea 16, se agrega un bit uno al final de la representación de las distancias de todos los elementos, esta conveniencia técnica será utilizada para decodificar al último elemento representado, lo cual se mostrará en la siguiente sección. La función $Codificar(p, B_D, B_S)$ es la representación propuesta para realizar la compresión de los elementos y generar los bitmaps de diferencias y de signos, posteriormente se construyen dos estructuras sucintas para manipular el bitmap de diferencias B_D y el bitmap de tamaño de bloque B_G.

$Construir_{SArray}(B_D)$ y $Construir_{GGMN}(B_G)$ son las funciones de la creación de las estructuras sucintas para compactar y manipular el bitmap de diferencias y el bitmap de tamaño de bloque.

Algoritmo 4 Compresión de una base de datos métrica usando LC

Entrada: La lista de clusters L, la base de datos métrica $\mathbb{S}$

Salida: La lista de los elementos centros C, el SArray del bitmap de diferencias SA_{B_D}, el bitmap de signos B_S, el GGMN del bitmap de tamaño de bloques $GGMN_{B_G}$.

1: $Iniciar(B_D, B_S, B_G, C)$
2: **para todo** $(\vec{c}, I) \in L$ **hacer**
3: $Insertar(\vec{c}, C)$
4: $Concatenar(B_G, 1)$
5: Seleccionar $\vec{c} \in \mathbb{S}$
6: **para todo** $\vec{u} \in I$ **hacer**
7: Seleccionar $\vec{u} \in \mathbb{S}$
8: $CrearVector(\vec{p})$
9: $Concatenar(B_G, 0)$
10: **para todo** $j = 0$ **hasta** dim **hacer**
11: $p_j = u_j - c_j$
12: $Codificar(\vec{p}, B_D, B_S)$
13: **fin para**
14: **fin para**
15: **fin para**
16: $Concatenar(B_D, 1)$
17: $SA_{B_D} \leftarrow Construir_{SArray}(B_D)$
18: $GGMN_{B_G} \leftarrow Construir_{GGMN}(B_G)$
19: **regresar** $C, SA_{B_D}, B_S, GGMN_{B_G}$

4.4.1. Decodificación de un vector

La manera de decodificar un vector y obtener su información original es la siguiente: suponiendo que se tiene la colección como la de la figura 4.13 y su índice LC construido.

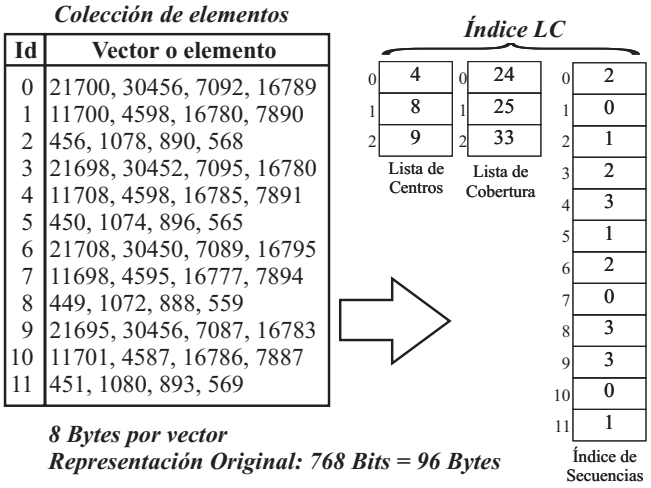


Figura 4.13: Ejemplo de una colección y su índice LC

La colección anterior tiene sólo doce elementos y cuatro coordenadas sin embargo servirá como ejemplo para entender la decodificación de un elemento. La representación de esta colección necesita 48 Bytes. El índice LC contiene tres elementos importantes:

- *La lista de centros:* en esta lista se encuentra el identificador del elemento que fue seleccionado como centro. (El ejemplo tiene tres centros)
- *La lista de cobertura:* asignado a cada centro se lista la distancia del elemento más lejano del bloque al centro.
- *El índice de secuencias:* contiene la asignación de bloque a cada elemento, si el elemento ya es un centro se asigna el total de los centros. Este índice permite hacer operaciones de *rank*, *select* y *access*.

Después de generar el índice LC, se crea la representación codificada figura 4.14, obteniendo la lista de centros, la cadena de tamaño de bloque, la cadena de diferencias y la cadena de signos, tal representación usa 27 Bytes al representar el bitmap de diferencias usando el SArray, siendo el porcentaje de compresión: $porcentaje_{compresión} = \frac{216*100}{768} = 28.125\,\%$.

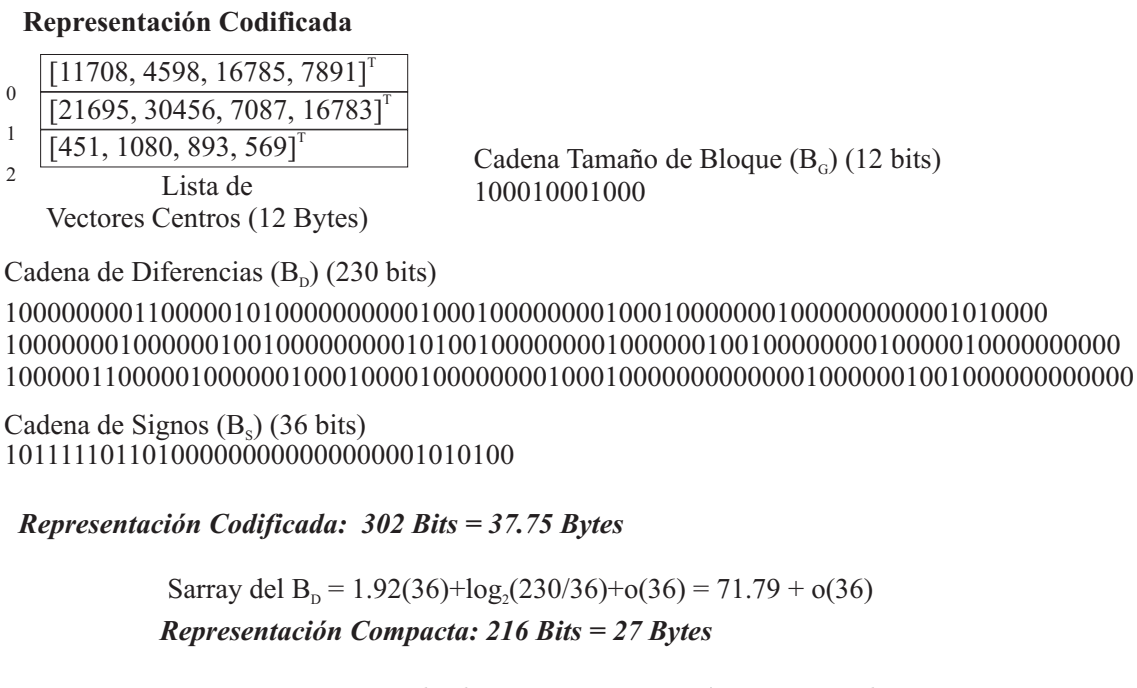


Figura 4.14: Ejemplo de una representación comprimida

Suponiendo que se desea extraer el elemento con $id = 6$, que en la versión original es: $[21708, 30450, 7089, 16795]^T$

Paso 1

El primer paso es buscar el elemento $id = 6$ en el índice de secuencias del LC, para saber si es un centro o si pertenece a un bloque. Figura 4.15

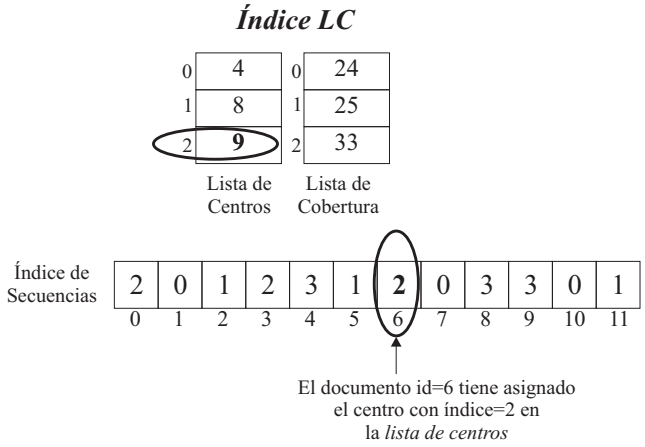


Figura 4.15: Ejemplo búsqueda de centro asignado

Para el ejemplo se observa que el elemento $id = 6$ esta asignado al bloque 2, revisando ese bloque (índice) en la lista de centros el representante de tal bloque es el elemento $id = 9$.

Haciendo un *rank* en el índice de secuencias se obtiene el número de elementos que pertenecen al bloque 2 hasta el elemento $id = 6$. Figura 4.16.

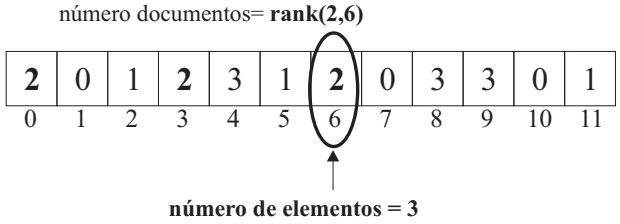


Figura 4.16: Ejemplo: obtener el número de elementos que pertenecen al bloque

Lo siguiente es hacer uso de la representación comprimida.

Paso 2

En la cadena de tamaño de bloque se busca la posición del centro haciendo $select_1(indice_{centro} + 1)$

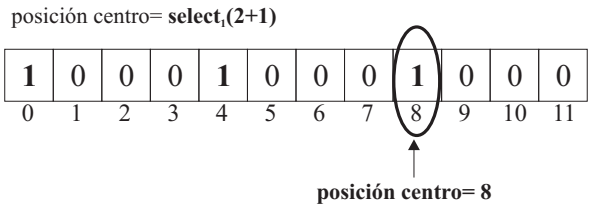


Figura 4.17: Ejemplo: buscar la posición del centro en la cadena de tamaño de bloque

Paso 3

Es necesario saber en que orden fue comprimido el elemento $id = 6$ para ubicar su representación en la cadena de diferencias y de signos, puesto que fue comprimido usando el orden del LC, esto se obtiene sumando la posición del centro y el número de elementos del bloque hasta el elemento de interés, restando a esta suma el número de bloque (índice del bloque) más uno, ya que los elementos representantes no son comprimidos sino guardados directamente en una lista.

$$ordinal_{vector} = posición_{centro} + número_{documentos} - (índice_{centro} + 1)$$

$$ordinal_{vector} = 8 + 3 - (2 + 1) = 8$$

El ordinal indica que 8 elementos fueron comprimidos antes del elemento $id = 6$, por lo que éste fue el noveno elemento en comprimirse.

Paso 4

Sabiendo que hay 8 elementos anteriores y que cada elemento se representa usando un bit uno por coordenada, para encontrar el elemento en la cadena de diferencias hay que calcular el número de unos anteriores el cual se obtiene haciendo:

$$posición_{vector} = ordinal_{vector} * dimensión$$

Paso 5

Haciendo la operación $select_1(posición_{vector} + 1)$ se obtiene la posición donde la cadena de

diferencias tiene un *rank* de unos anteriores más el uno de la primera coordenada que indica el vector o elemento que se está buscando. Figura 4.18.

$$Select_1(32+1) = 193$$

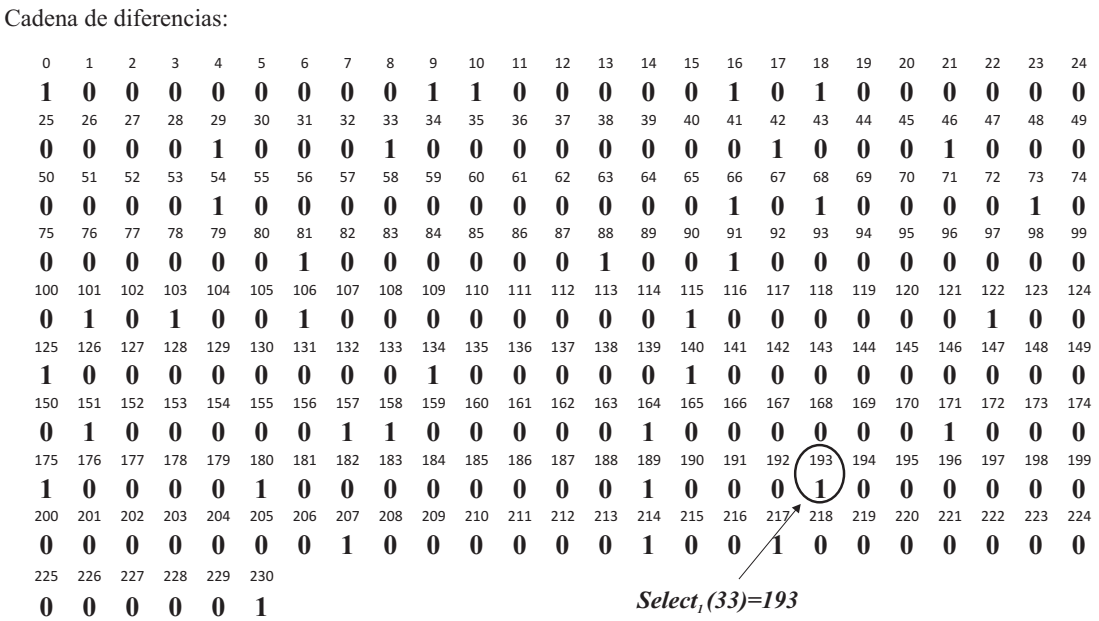


Figura 4.18: Ejemplo: posición de inicio de la representación comprimida

Paso 6

Cada coordenada se ha representado en la cadena como un uno, entonces el *rank* de unos puede incrementarse hasta obtener el total de coordenadas. Esta operación se efectúa *dimensión*+1 veces, ya que en la representación se hace: *un bit uno (que indica la coordenada)* seguido de la diferencia en ceros. En esta parte podemos observar que la concatenación de la representación de los vectores permite hacer la operación *select dimensión* + 1 veces, es por ello que al final de la representación se concatena un bit uno.

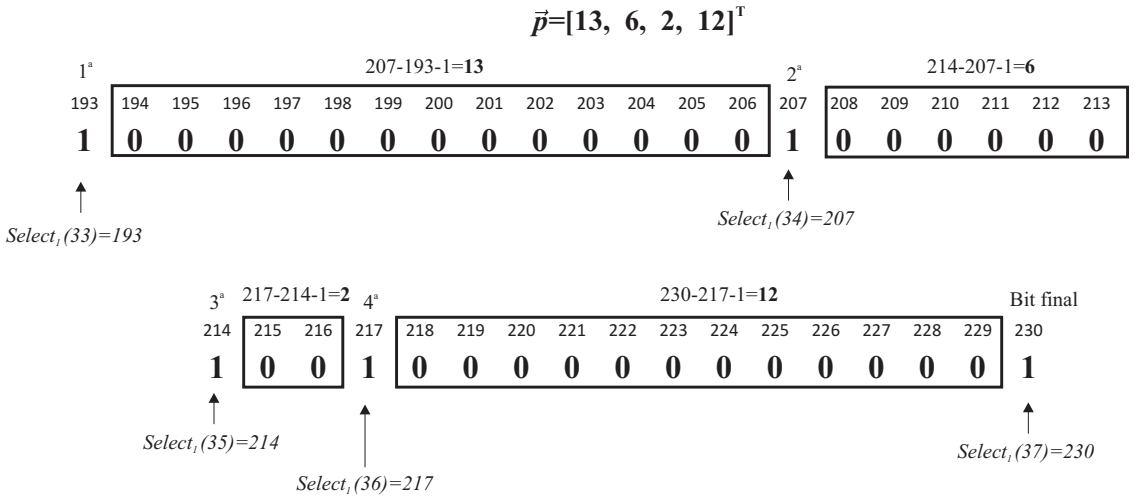


Figura 4.19: Ejemplo: obtener las posiciones de las coordenadas del vector

Ya que se obtienen las posiciones de las coordenadas se calcula el vector de diferencia al que llamaremos p. Figura 4.20

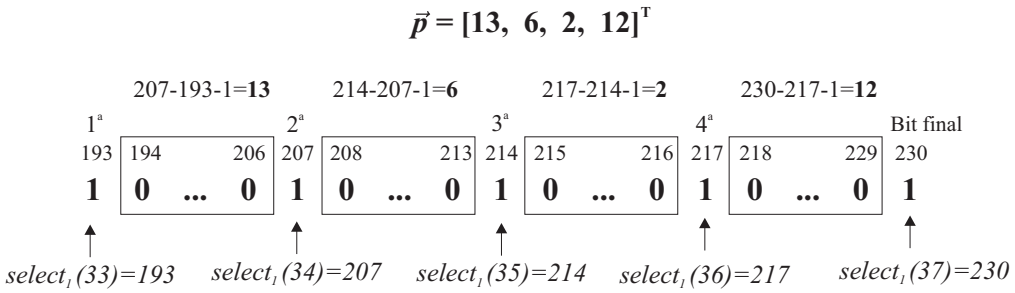


Figura 4.20: Ejemplo: obtener el vector de diferencias

Paso 7

Finalmente es necesario obtener el tipo de diferencia que se hará al vector centro para recuperar el elemento $id = 6$. En la cadena de signos se realiza la operación $access(posición_{vector})$ incrementando esta posición *dimensión* veces, figura 4.21.

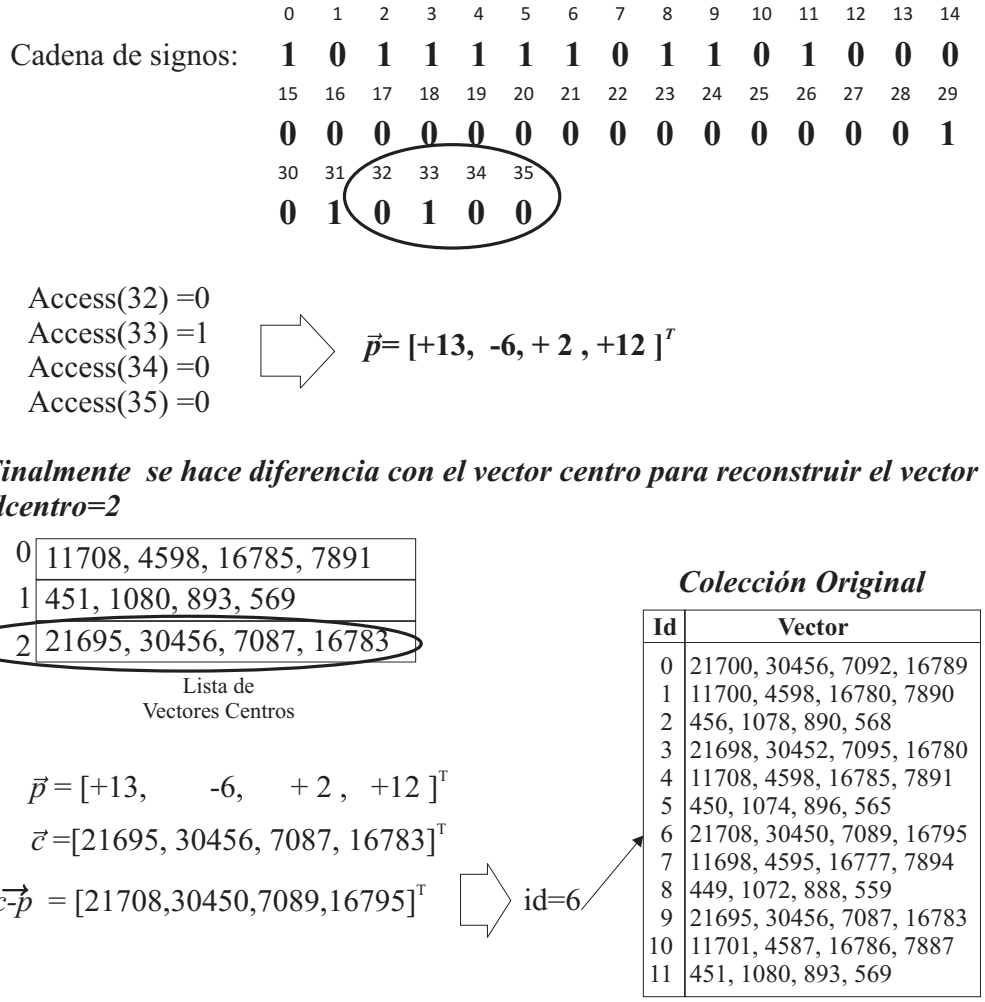


Figura 4.21: Ejemplo: obtener el vector de diferencias

De esta manera se recupera el vector $id = 6$.

4.4.2. Algoritmos de descompresión

El Algoritmo 5 presenta el proceso de descompresión de la base de datos usando el orden del LC. La base de datos descomprimida en la práctica deber ser comparada con la base de datos original, este proceso se realiza comparando la distancia de cada elemento de $\mathbb{S}'$ con el correspondiente elemento en $\mathbb{S}$ verificando que $d(x, x') = 0$.

Algoritmo 5 Descompresión de la base de datos

Entrada: La lista de clusters L,la lista de vectores centrales C, el SArray del bitmap de diferencias SA_{B_D}, el bitmap de signos B_S, el GGMN de tamaño de bloques GGM_{B_G}.

Salida: La base de datos descomprimida $\mathbb{S}'$

1: $pos_{signo} \leftarrow 0$
2: $rank1 \leftarrow 1$
3: $rank2 \leftarrow 2$
4: **para todo** $\vec{c} \in C$ **hacer**
5: $\quad Insertar(\vec{c}, \mathbb{S}')$
6: $\quad idx_{centro} \leftarrow Get_{indice}(c, C)$
7: $\quad$ **para todo** $\vec{u} \in I$ **hacer**
8: $\quad\quad$ **para todo** $j \leftarrow 1$ hasta dim **hacer**
9: $\quad\quad\quad signo \leftarrow Access(B_S, pos_{sig})$
10: $\quad\quad\quad u_j \leftarrow signo \times (Select(SA_{B_D}, rank2) - Select(SA_{B_D}, rank1)) - c_j$
11: $\quad\quad\quad pos_{signo} \leftarrow pos_{signo} + 1$
12: $\quad\quad\quad rank1 \leftarrow rank1 + 1$
13: $\quad\quad\quad rank2 \leftarrow rank1$
14: $\quad\quad$ **fin para**
15: $\quad\quad Insertar(\vec{u}, \mathbb{S}')$
16: $\quad$ **fin para**
17: **fin para**
18: **regresar** $\mathbb{S}'$

En el Algoritmo 6 se presenta la descompresión de un vector, dado únicamente su identificador, de ésta manera si el elemento dado es centro se extrae del conjunto de elementos almacenados explícitamente en caso contrario lo decodifica.

Algoritmo 6 Algoritmo de descompresión de vector dado su número de documento

Entrada: El identificador del documento en la base de datos id_{doc}, La lista de clusters L, la lista vectores centrales C, el SArray del bitmap de diferencias SA_{B_D}, el bitmap de signos B_S, el GGMN de tamaño de bloques GGM_{B_G}.

Salida: El vector $\vec{u}$ descomprimido

1: $GetVector(id_{doc})$
2: $CrearVector(\vec{u})$
3: $dim \leftarrow$ dimension de $\vec{u}$
4: **si** $id_{doc} \in C$ (Lista de centros) **entonces** ▷ Si el elemento es un centro
5: $\vec{u} \leftarrow$ Seleccionar $id_{doc} \in B$
6: **regresar** $\vec{u}$
7: **si no**
8: $idx_{centro} \leftarrow Buscar(id_{doc}, L)$
9: $num_{docs} \leftarrow Buscar(id_{doc}, L_{SEQ})$
10: $pos_{centro} \leftarrow Select_1(GGMN_{B_G}, idx_{centro} + 1)$
11: $ordinal_{doc} \leftarrow pos_{centro} + num_{docs} - (idx_centro + 1)$
12: $pos_{doc} \leftarrow ordinal_{doc} * dim$
13: $rank1 \leftarrow Select_1(SA_{B_D}, pos_{doc} + 1)$
14: $pos_{signo} \leftarrow pos_{doc}$
15: $rank2 \leftarrow rank1 + 1$
16: Seleccionar $\vec{c} \in C$
17: **para todo** $j = 0$ hasta dim **hacer**
18: $signo \leftarrow Access(B_S, pos_{signo})$
19: $u_j = (c_j) + (signo * (rank2 - rank1 - 1))$
20: $rank1 \leftarrow rank2$
21: $rank2 \leftarrow Select_1(SA_{B_D}, rank2 + 1)$
22: $pos_{signo} \leftarrow pos_{signo} + 1$
23: **fin para**
24: **fin si**
25: **regresar** $\vec{u}$

4.5. Búsqueda por similitud

El esquema de compresión propuesto basado en el índice LC permite realizar búsquedas por rango, cada consulta es comparada en un rango dado con la lista de centros de la representación comprimida obteniendo así una distancia, si esta distancia es menor o igual al rango dado se revisa el bloque que el centro representa, esto implica la descompresión de cada vector perteneciente al bloque. La métrica utilizada es L_1.

Algoritmo 7 Búsqueda por rango, en la base de datos comprimida usando Lista de Clusters

Entrada: La lista de clusters L,la consulta q

Salida: El conjunto de resultados R

1: $R \leftarrow \emptyset$
2: **para todo** $(\vec{c}, cov(\vec{c}), I) \in L$ **hacer**
3: Seleccionar $\vec{c} \in B$
4: Seleccionar un radio $cov(\vec{c})$
5: $d_{cq} = d(\vec{c}, q)$
6: **si** $d_{cq} \leq r$ **entonces**
7: $R \leftarrow \bigcup\{c\}$
8: **fin si**
9: **si** $d_{cq} \leq cov(\vec{c}) + r$ **entonces**
10: **para todo** $u \in I$ **hacer**
11: **si** $d(GetVector(\vec{u}, \vec{c}), q) \leq r$ **entonces**
12: $R \leftarrow R \bigcup\{\vec{u}\}$
13: **fin si**
14: **fin para**
15: **fin si**
16: **si** $d_{cq} < cov(\vec{c}) - r$ **entonces**
17: regresar R
18: **fin si**
19: **fin para**

El Algoritmo 7 realiza la búsqueda de una consulta q utilizando en índice LC y la

versión comprimida de la base de datos.

Se observa que al descomprimir el vector $\vec{u}$ se utiliza la función $GetVector$ donde además del identificador de $\vec{u}$ también se envía el identificador del centro $\vec{c}$.

La función $GetVector(\vec{u}, \vec{c})$ realiza la descompresión de un vector dado su identificador y el de su centro Algoritmo 8

Algoritmo 8 Algoritmo de descompresión de vector dado su número de documento y el índice del centro

Entrada: El identificador del documento en la base de datos id_{doc}, el identificador del centro idx_{centro}, La lista de clusters L, la lista de vectores centrales C, el SArray del bitmap de diferencias SA_{B_D}, el bitmap de signos B_S, el GGMN de tamaño de bloques GGM_{B_G}.

Salida: El vector $\vec{u}$ descomprimido

1: $GetVector(id_{doc}, idx_{centro})$
2: $u \leftarrow \emptyset$
3: $num_{docs} \leftarrow GetIndice(id_u, I)$
4: $pos_{centro} \leftarrow Select_1(GGMN_{B_G}, idx_{centro} + 1)$
5: $ordinal_{doc} \leftarrow pos_{centro} + num_{docs} - (idx_centro + 1)$
6: $pos_{signo} \leftarrow ordinal_{doc} * dim$
7: $rank1 \leftarrow Select_1(SA_{B_D}, pos_{doc} + 1)$
8: $rank2 \leftarrow rank1 + 1$
9: Seleccionar $c \in C$
10: **para todo** $j = 0$ hasta dim **hacer**
11: $\quad signo \leftarrow Access(B_S, pos_{signo})$
12: $\quad u_j = (c_j) + (signo * (rank2 - rank1 - 1))$
13: $\quad rank1 \leftarrow rank2$
14: $\quad rank2 \leftarrow Select_1(SA_{B_D}, rank2 + 1)$
15: $\quad pos_{signo} \leftarrow pos_{signo} + 1$
16: **fin para**
17: **regresar** $\vec{u}$

4.6. Análisis del esquema de compresión

4.6.1. Complejidad de las estructuras sucintas

El esquema de compresión propuesto, hace uso del **SArray** el cual realiza la operación *Select* en $O(\frac{log^4 m}{\log n})$ y *Rank* en $O(log\frac{n}{m}) + O(\frac{log^4 m}{\log n})$ y del **GGMN** que realiza la operación *Select* en $O(\log n)$ y *Rank* en $O(1)$.

Como se definió anteriormente la operación necesaria para descomprimir un vector haciendo uso de las dos estructuras es la operación *Select* la cual en los dos casos se realiza en un tiempo pequeño a comparación de otras estructuras.

4.6.2. Cantidad de Compresión

Al hablar de la cantidad de compresión que se puede alcanzar primeramente es necesario conocer el número de bits que se emplearán para la representación comprimida:

1) **Número de bits para hacer la representación basada en distancias**
   Llamamos B_D el bitmap que contiene la representación basada en distancias de la base de datos $\mathbb{S}$ donde:
   n es el número de elementos de una base de datos.
   dim la dimensión de la base de datos (número de coordenadas).
   m el número de bloques con tamaño k en que se ha agrupado la base de datos.
   $\vec{c_i}$ el vector centro correspondiente al i-ésimo bloque.

   El número de bits unos en B_D es:

$$B_{D_1} = (n - m) * dim + 1$$

   El número de bits ceros en B_D es:

$$B_{D_0} = \sum_{i=1}^{m} L_1(\vec{u_{ij}} - \vec{c_i})$$

   para $j = 1$ hasta k.

El número total de bits del bitmap de diferencias B_D es:

$$|B_D| = B_{D_1} + B_{D_0}$$

2) **Número de bits para hacer la representación de los signos de las diferencias**
Llamamos B_S el bitmap que contiene la representación de los signos de las diferencias de la base de datos S donde:

El número total de bits del bitmap de signos B_S es:

$$|B_S| = (n - m) * dim$$

3) **Número de bits para hacer la representación de tamaño de bloques**
Sea B_G el bitmap que contiene el tamaño de bloques de la base de datos S
El número de bits unos en B_G es:

$$B_{G_1} = m$$

El número de bits ceros en T es:

$$B_{G_0} = (n - m)$$

El número total de bits del bitmap de tamaño de bloques B_G es:

$$|B_G| = B_{G_1} + B_{G_0}$$

donde:

$$|B_G| = |S|$$

Finalmente la representación comprimida tiene:

1. Un archivo binario que contiene:
   (a) Una cadena con el nombre del índice.
   (b) El bitmap de diferencias B_D representado en un SArray.

(c) El bitmap tamaño de bloque B_G representado en un GGMN.

(d) El bitmap de signos B_S

2. Un archivo binario que contiene:

(a) La lista de los m vectores centros

Finalmente el espacio en bits de la nueva representación utiliza:

- **La representación del SArray**

$$1.92m + \log_2(\frac{n}{m}) + o(m)\ bits$$

- **La representación del GGMN**

$$n + o(n)\ bits$$

Donde m es el número de unos y n el número total de bits.

- **La representación de la lista de centros** C

Sea t el número de bits del tipo de dato para representar los valores de las coordenadas de un elemento de la base de datos $\mathbb{S}$.

$$|C| = m(t * dim)\ bits$$

- **Nombre del índice LC**

El nombre del índice LC se guardará en un tipo de dato *cadena* que llamaremos *st*

Espacio en bits de la base de datos comprimida El espacio en bits de la nueva representación R se expresa de la siguiente manera:

$$R = [1.92m + \log_2(\frac{n}{m}) + o(m)] + [n + o(n)] + |B_S| + |C| + |st| \tag{4.2}$$

Este esquema de compresión fue evaluado en una base de datos métrica de 1 millón de datos, los resultados se muestran en el siguiente capítulo.

4.7. Conclusión

En este capítulo se presenta el esquema propuesto para comprimir una base de datos, dicho esquema se basa en la agrupación de los elementos por su similitud usando el índice LC el cual proporciona una lista de grupos donde cada grupo tiene un vector representante. Una vez hecha la agrupación los elementos de cada grupo son evaluados con su representante usando la distancia L_1 obteniendo vectores de diferencias estos vectores de diferencias son codificados en dos bitmaps. Al realizar la codificación de la nueva representación se utilizan estructuras sucintas para lograr la manipulación y compactación de los bitmaps creados.

Con el esquema propuesto es posible determinar el espacio que ocupará la nueva representación de la base de datos. Los algoritmos de compresión y descompresión que se proponen en este trabajo se realizan sobre la base de datos completa, se presenta también un algoritmo de descompresión para obtener únicamente un vector de la base de datos dicho proceso es útil para realizar la búsqueda por rango que implementa el índice LC, de esta manera si la consulta q cae en el rango de un grupo I cualquiera se descomprimirán solo los vectores pertenecientes a este grupo de manera secuencial, este procedimiento puede retardar el proceso de búsqueda comparado con el proceso de búsqueda en una base de datos sin comprimir, sin embargo permite utilizar un espacio menor en memoria principal para manejar la base de datos y el índice LC.

Capítulo 5

Experimentos y Resultados

5.1. CoPhIR

La técnica de compresión propuesta fue probada utilizando la colección CoPhIR (Content-based Photo Image Retrieval)[Bolettieri et al., 2009] un repositorio de metadatos de imágenes de Flickr, creado por varios consorcios universitarios para realizar investigación en recuperación de información principalmente. CoPhIR tiene alrededor de 106 millones de imágenes y ocupa un espacio aproximado de 42 GB su índice ocupa 200 MB y comprimido 20 MB, la base de datos comprimida con algun formato de compresión conocido (gzip, zip, rar) ocupa un espacio de 5 GB, sin embargo es necesario descomprimirla para realizar consultas y acceder a la información. Cada registro de la colección CoPhIR es un XML con la siguiente estructura:

1. Un identificador que permite recuperar el enlace de la imagen correspondiente en el sitio Flickr
2. Los metadatos de la imagen: autor, título, descripción, localización GPS, etiquetas, comentarios, contador de vistas, entre otros.
3. Una subestructura XML que contiene la información relacionada con los descriptores visuales MPEG-7 [Sikora, 2001] de la imagen.
   (a) Descriptor de Color Escalable (Scalable Colour)

(b) Descriptor de la Estructura del Color (Colour Structure)

(c) Descriptor de Color de Diseño(Colour Layout)

(d) Descriptor de Textura No homogénea -Histograma de borde (Edge Histogram)

(e) Descriptor de textura homogénea (Homogeneous Texture)

Los algoritmos del modelo propuesto se implementaron en el lenguaje C# utilizando el entorno de desarrollo MonoDevelop 3.0 y la librería Natix [1],es una librería polifásica que implementa algunos métodos para búsqueda por similitud, compresión de datos, listas invertidas, estructuras sucintas y algoritmos de ordenamiento. El equipo utilizado fue un servidor Intel ®Xeon ®CPU E5620 @ 2.40 GHz con 32GB en RAM.

Se utilizó un subconjunto de la base de datos de CoPhIR con 1 millón de datos al que llamaremos *Cophir1M*, cada objeto de esta colección es un vector de 208 coordenadas. Dichos elementos fueron creados usando una combinación lineal de los cinco descriptores MPEG7 como se describe en [Bolettieri et al., 2009].

Utilizando la librería Natix se crearon cuatro índices LC para cophir1M, para cada índice se definieron los siguientes tamaños de bloque:

- 128 : usando 7813 centros.
- 256 : usando 3907 centros.
- 512 : usando 1954 centros.
- 1024 : usando 977 centros.

A continuación se muestran los histogramas de distancias que se evaluaron al realizar la compresión de la base de datos utilizando cada índice LC creado anteriormente. En el eje x se representan las 208 coordenadas de la base de datos, en el eje y la suma de todas las distancias de los vectores a sus centros.

[1]Natix: https://github.com/sadit/natix

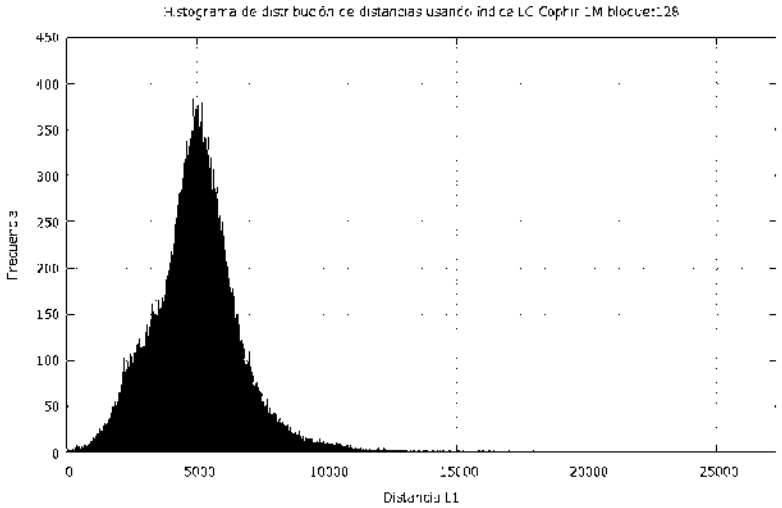


Figura 5.1: Histograma de distancias a los centros LC CoPhir1M: 128 centros

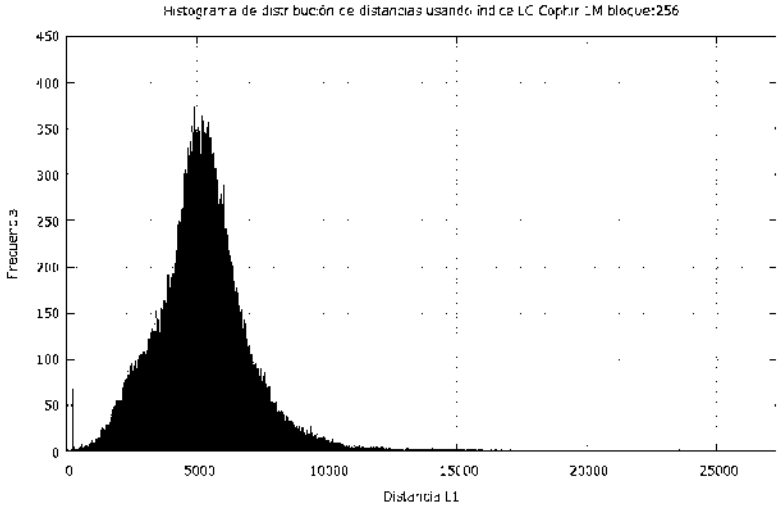


Figura 5.2: Histograma de distancias a los centros LC CoPhir1M: 256 centros

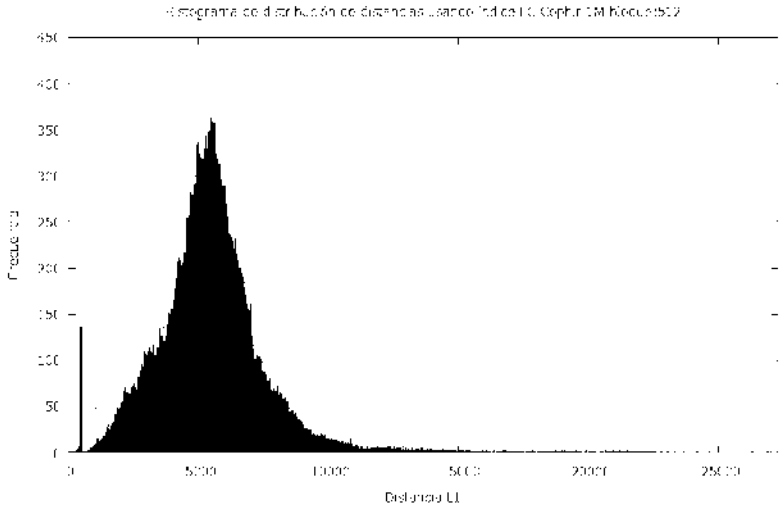


Figura 5.3: Histograma de distancias a los centros LC CoPhir1M: 512 centros

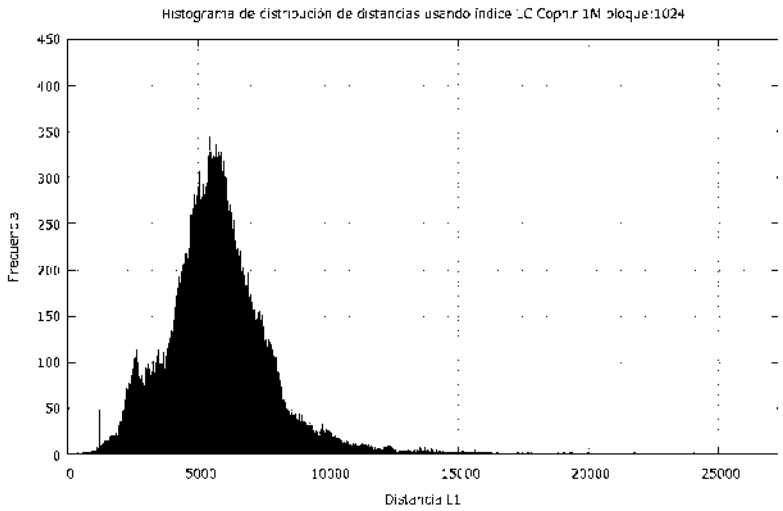


Figura 5.4: Histograma de distancias a los centros LC CoPhir1M: 1024 centros

Se puede observar que las distancias a los centros utilizando el índice LC con diferentes tamaños de bloque, siguen un comportamiento similar, lo cual también se observa en el comportamiento de compresión y descompresión de la base de datos.

El proceso de compresión y descompresión propuesto fue evaluado utilizando los índices anteriores. Se generaron cuatro archivos que representan la base de datos CoP-

hIR1M, cada uno utilizando la partición definida por el índice LC, es decir, 128, 256, 512 y 1024 centros. La figura 5.5 muestra el resultado obtenido del tiempo de compresión y descompresión en segundos.

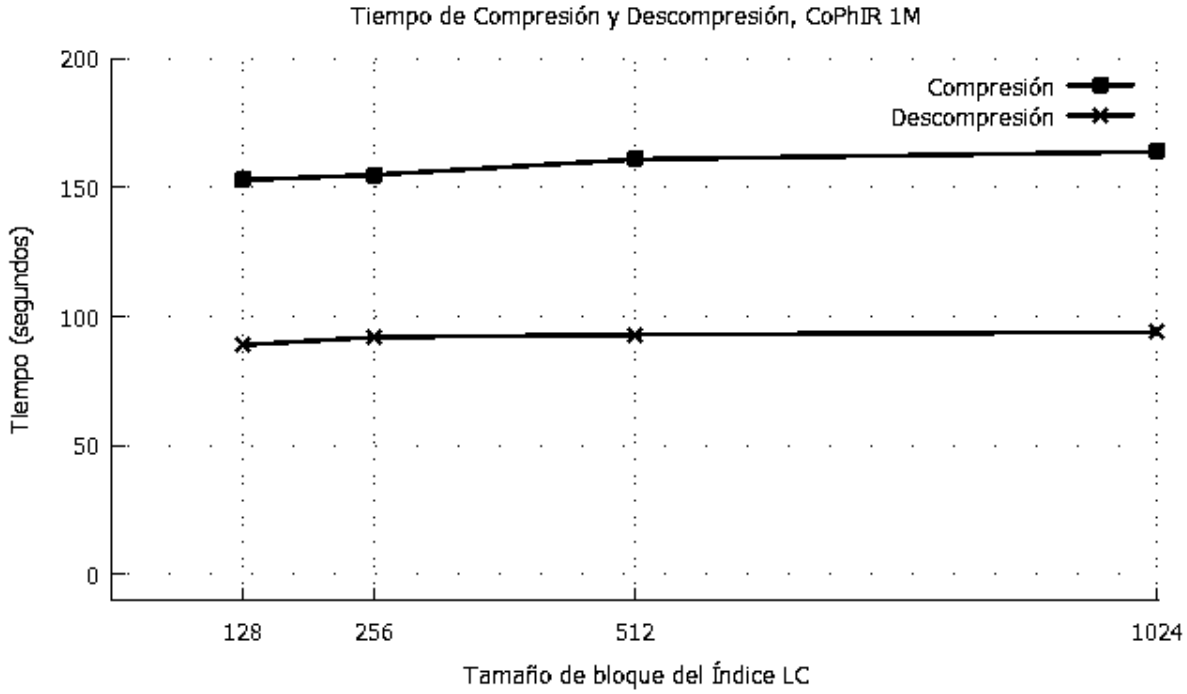


Figura 5.5: Tiempo de compresión y descompresión

El tiempo promedio de compresión en los 4 índices es de 2.63 minutos y de descompresión de 1.53 minutos. El tiempo de compresión resulta más grande debido a que después de la representación propuesta se construyen las estructuras sucintas.

Para evaluar el porcentaje de compresión obtenido en las 4 representaciones se comparó el número de bits en la base de datos original con las diferentes representaciones. Los resultados se observan en la Tabla 5.1.

Cophir 1M		
Tamaño BD: 407689.08624 KB		
LC Bloque	**Porcentaje Compresión**	**Tamaño en KB**
128	57.67 %	231808.80604 KB
256	57.50 %	232720.86068 KB
512	57.42 %	233176.88996 KB
1024	57.38 %	233405.01534 KB

Tabla 5.1: Porcentaje de compresión

De acuerdo al porcentaje de obtenido se deduce que la compresión es un poco mayor utilizando un índice con un tamaño de bloque de 128 elementos.

El tiempo de búsqueda se evaluó comparando el tiempo de búsqueda en la base de datos original y descomprimiendo el vector sólo cuando la consulta esté dentro de un rango dado.

Se realizaron 100 consultas utilizando un rango de 4000 sobre las diferentes versiones compactadas (128,256,512,1024) y con el indice correspondiente. Las consultas realizadas pertenecen a la colección de 10 millones de CoPhIR.

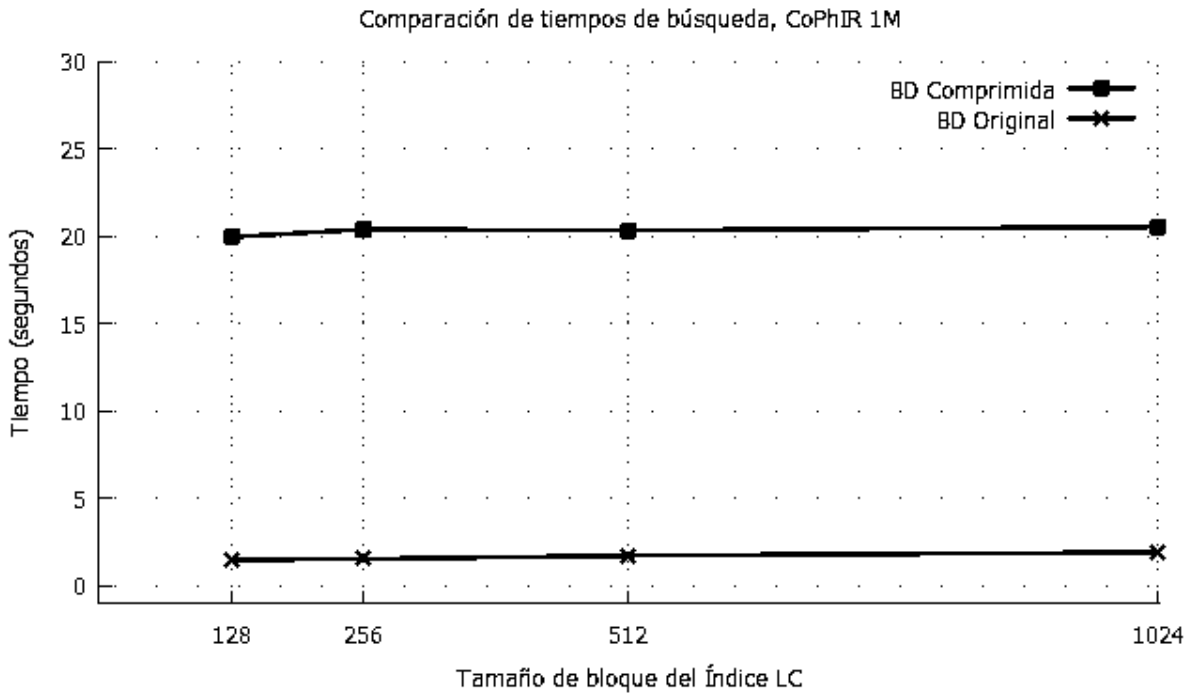


Figura 5.6: Tiempo de búsqueda con compresión y sin compresión para 100 consultas con radio 4000

Sobre las mismas versiones de la base de datos se realizaron 100 consultas con un rango 0, estas consultas pertenecen a la colección de 1 millón, por lo que cada consulta fue encontrada y el vector descomprimido para el resultado.

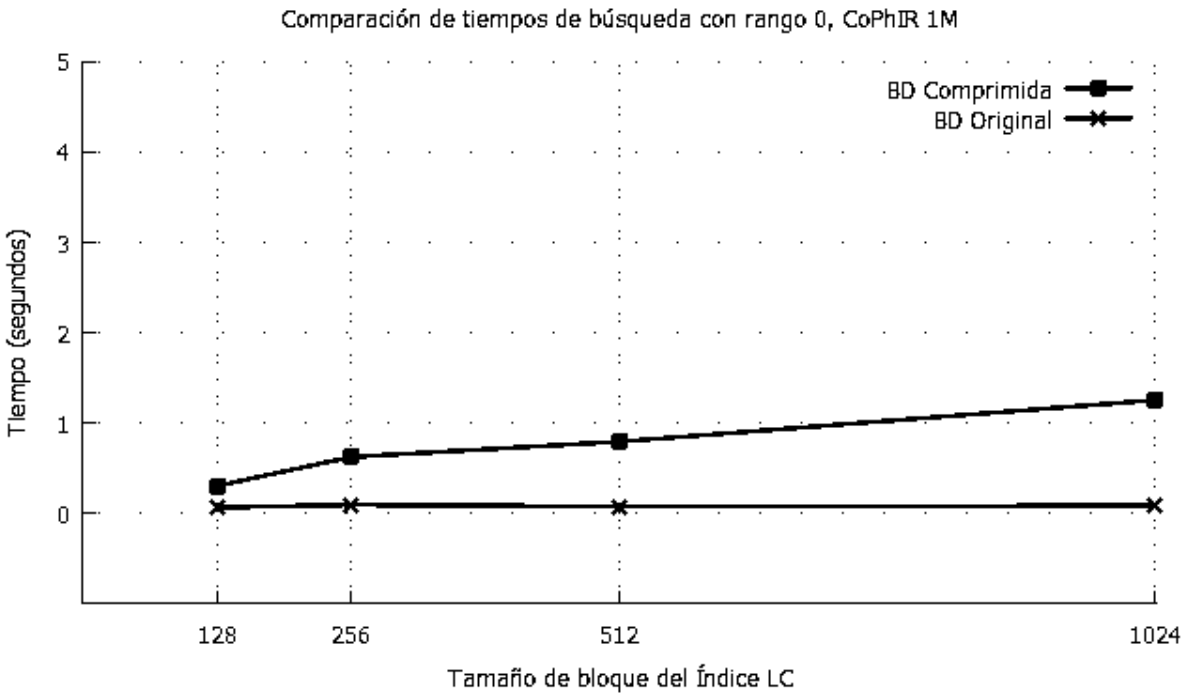


Figura 5.7: Tiempo de búsqueda con compresión y sin compresión para 100 consultas con radio 0

Observamos en los resultados anteriores que:

El porcentaje de compresión es menor cuando utilizamos un índice con un tamaño de bloque de 1024 elementos mientras que el tiempo de búsqueda es más pequeño cuando utilizamos un índice de 128 elementos.

Al evaluar la colección de 10 millones de CoPhIR (Cophir10M) se observó que la implementación basada en Natix sólo permite almacenar 2^{32} bits para el bitmap de diferencias, además que la creación del índice LC para un bloque de 128 elementos lleva un tiempo de construcción muy largo. Sin embargo es posible calcular el porcentaje de compresión que se puede alcanzar, ya que puede calcular la distancia L_1 de cada elemento a su centro correspondiente y obtener el número de ceros que este bitmap contendrá.

Utilizando el índice LC-IRNN [Téllez y Chávez, 2012] el cual tiene un tiempo de construcción más corto que el LC, se creó un índice para Cophir10M usando 1024 centros lo que permite que cada bloque contenga aproximadamente 9766 elementos.

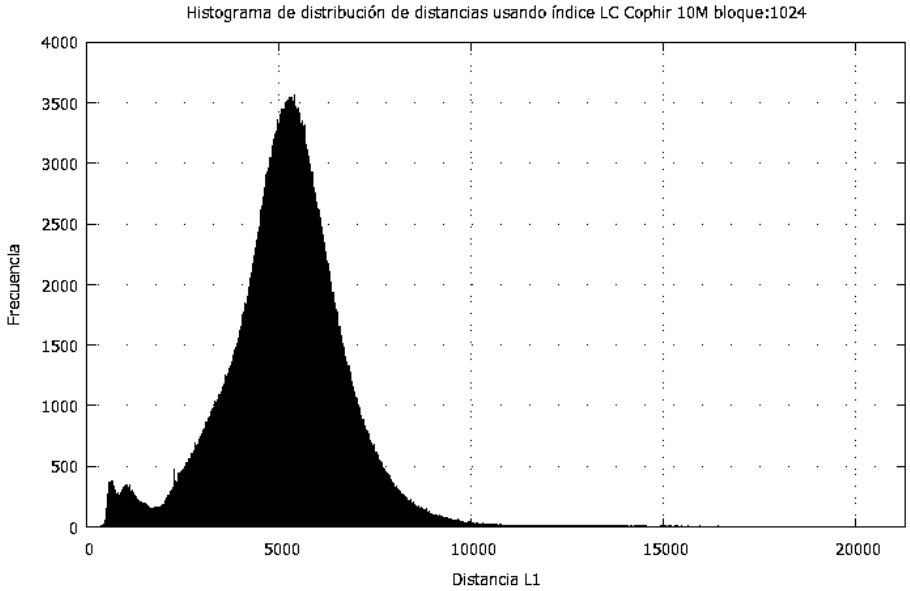


Figura 5.8: Histograma de distancias a los centros, LC-Rev CoPhIR-10M: 1024 centros

Haciendo la suma de diferencias L_1 de cada elemento a su centro se obtiene:

$$D_0 = 51526080512$$

El número de unos en el bitmap de diferencias es:

$$D_1 = (1 * 10^7 - 1024) * 208 + 1 = 2079787009$$

Por lo que el número de bits del bitmap de diferencias es:

$$D = 53605867521$$

el SArray

$$1.92(2079787009) + 2079787009 * \log_2(\frac{53605867521}{2079787009}) = 13742989693$$

El número de bits del bitmap de signos:

$$|B_S| = (1 * 10^7 - 1024) * 208 = 2079787008$$

El número de bit del bitmap de tamaño de bloque:

$$|B_G| = 10^7$$

La representación de la base de datos Cophir10M en Natix usa el tipo de dato *short* (16 bits) para cada coordenada, por lo que la representación de los vectores base es de:

$$|C| = 1024(16 * 208) = 3407872$$

La representación comprimida de la base de datos Cophir10M, utiliza aproximadamente:

$$R_{cophir10M} = 13742989693 + 2079787008 + 10^7 + 3407872 = 15836184573$$

$$R_{cophir10M} = 1.84GB$$

El porcentaje de compresión aproximado es:

$$p = \frac{1.84 * 100}{3.87} = 47.54$$

Este porcentaje es teórico, no toma en cuenta el espacio utilizado por las cabeceras de los archivos.

La representación propuesta puede implementarse para bases de datos grandes, donde es posible que las diferencias usen más de 2^{32} bits (que es la implementación realizada hasta el momento) representando el logaritmo base dos de la diferencia, en lugar del código unario de la diferencia.

Así una representación:

$$[4, 8, 2]^T$$

$$100001000000001001$$

puede ser

$$[2, 3, 1]^T$$

$$1001000101$$

Los resultados obtenidos muestran que se puede alcanzar un porcentaje de compresión entre el 50 % al 60 % del tamaño original de la base de datos.

5.2. Conclusión

Los experimentos realizados para evaluar el esquema de compresión propuesto en este trabajo fueron efectuados utilizando un subconjunto de la base de datos CoPhIR y lenguaje C#, con la ayuda de la librería Natix se implementaron los métodos de indexación LC. El conjunto de un millón de datos fue indexado utilizando cuatro tamaños diferentes de bloque, aquí se observó que el proceso de indexación requiere tiempo y que este tiempo aumenta con el tamaño de la base de datos.

Al llevar a cabo el proceso de compresión, búsqueda y descompresión sobre la base de datos de un millón se obtuvieron buenos resultados. Sin embargo este proceso no pudo implementarse sobre la base de datos de diez millones esto debido a la limitación del lenguaje puesto que la representación codificada requiere una capacidad mayor a 2^32 no obstante dicho esquema permite conocer el tamaño de la representación comprimida observando que también se alcanza un porcentaje de compresión cercano al 60 % como en la representación de un millón de datos.

Capítulo 6

Conclusiones

6.1. Conclusiones Generales

La aportación de este trabajo es un esquema de compresión sobre una base de datos métrica, específicamente en un espacio métrico vectorial donde los elementos de la colección son identificados con coordenadas. Se presentan los algoritmos de compresión y descompresión además de un algoritmo de descompresión de un solo vector lo que permite que al realizar una búsqueda se descompriman únicamente los grupos que intersecten con la consulta dada.

Después de los experimentos realizados bajo el esquema propuesto se observaron varios aspectos importantes, principalmente que el proceso de agrupación de la base de datos influye en el resultado de compresión así como la utilización de una estructura sucinta sobre la codificación realizada.

La implementación del modelo se realizó en una base de datos en particular CoPhIR para una colección de 1 millón de datos, utilizando como método de indexación el índice LC. Se observa que el porcentaje de compresión evaluado en la base de datos Cophir1M es alrededor del 60 % y el modelo permite tener un tiempo de búsqueda aproximadamente 10 veces mayor del tiempo de búsqueda en la base de datos normal.

El comportamiento de la compresión sobre una colección de 10 millones que se propone es teórico puesto que esta limitado por la implementación realizada.

Un aspecto importante para el modelo propuesto es la selección de los centros, pues mientras más representativos sean, se podrán obtener mejores resultados en la compresión. Este problema esta conectado con el problema de clustering con L_1 por lo que la mejora del modelo depende de la mejora del problema de clustering o partición.

El modelo de compresión de base de datos métricas propuesto puede utilizar a conveniencia de alguna aplicación algún otro modelo de partición e incluso otra estructura sucinta, tomando en cuenta los aspectos de tiempo sobre las operaciones como *rank*, *select* y *access*.

Aún cuando la búsqueda resulta 10 veces más grande que la de una base de datos sin comprimir, la representación comprimida más el indice utilizan un espacio casi menor al de la representación original, es decir esta representación podría utilizarse en un sistema con un dispositivo de memoria limitada, por ejemplo; un teléfono móvil, alcanzando una de las motivaciones de este trabajo.

6.2. Trabajos Futuros

El tema de compresión de bases de datos métricas es nuevo y amplio, puede utilizar muchos recursos ya existentes para el diseño de esquemas de compresión como algoritmos basados en gramáticas o en diccionarios y estructuras sucintas como Re-Pair [Navarro y Russo, 2008].

La partición del espacio métrico es importante para la compresión, esto abre una posibilidad para buscar una mejor opción para la selección de los representantes con la finalidad de mejorar el porcentaje de compresión bajo el esquema propuesto.

Si bien la codificación propuesta esta limitada por la implementación, se pueden diseñar nuevas forma de codificación, por ejemplo; el lugar de representar el valor de la coordenada de diferencias usando ceros utilizar el logaritmo base dos para la representación de los ceros y evaluar si esto altera el tiempo de búsqueda y descompresión.

Dado que el modelo de compresión se implementó en lenguaje C# es posible utilizarlo en el diseño de alguna aplicación móvil como la propuesta en la motivación de este trabajo.

Referencias

[OLT, 2012] (2012). Advanced compression with oracle database 11g. White Paper.

[Baeza-Yates et al., 1994] Baeza-Yates, R. A., Cunto, W., Manber, U., y Wu, S. (1994). Proximity matching using fixed-queries trees. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, CPM '94, pages 198–212. Springer-Verlag.

[Bolettieri et al., 2009] Bolettieri, P., Esuli, A., Falchi, F., Lucchese, C., Perego, R., Piccioli, T., y Rabitti, F. (2009). Cophir: a test collection for content-based image retrieval. *CoRR*, abs/0905.4.

[Bozkaya y Ozsoyoglu, 1997] Bozkaya, T. y Ozsoyoglu, M. (1997). *Distance-based indexing for high-dimensional metric spaces*, volume 26, pages 357–368. ACM Press.

[Brin, 1995] Brin, S. (1995). Near neighbor search in large metric spaces. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 574–584. Morgan Kaufmann Publishers Inc.

[Burkhard y Keller, 1973] Burkhard, W. A. y Keller, R. M. (1973). Some approaches to best-match file searching. *Commun. ACM*, 16(4):230–236.

[Chavez y Navarro, 2005] Chavez, E. y Navarro, G. (2005). A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*.

[Chávez et al., 1999] Chávez, E., Marroquín, J., y Navarro, G. (1999). *Overcoming the Curse of Dimensionality*, pages 57–64.

[Chávez et al., 2001] Chávez, E., Marroquín, J., y Navarro, G. (2001). Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications*, 14(2):113–135.

[Chávez y Navarro, 2006] Chávez, E. y Navarro, G. (2006). Metric databases. In *Encyclopedia of Database Technologies and Applications*, pages 367–372. Idea Group Inc.

[Chávez et al., 001b] Chávez, E., Navarro, G., Baeza-Yates, R., y Marroquin, J. (2001b). Searching in metric spaces. In *ACM Comput. Surv.*, volume 33, page 273.

[Ciaccia et al., 1997] Ciaccia, P., Patella, M., y Zezula, P. (1997). M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 426–435. Morgan Kaufmann Publishers Inc.

[Claude y Gonzálo, 2008] Claude, F. y Gonzálo, N. (2008). Practical rank/select queries over arbitrary sequences. In *In Proc. 15th SPIRE, LNCS 5280*, pages 176–187.

[Comer, 1979] Comer, D. (1979). Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137.

[Damaine, 2012] Damaine, E. (2012). Advanced data structures. University Lecture.

[Daniel et al., 1997] Daniel, B., duMouchel, W., Faloutsos, C., Haas, P. J., Hellerstein, J. M., Ioannidis, Y., Jagadish, H. V., Johnson, T., Ng, R., Poosala, V., Kenneth, R., y Kenneth, S. (1997). The new jersey data reduction report. *IEEE DATA ENG. BULL*, pages 3–45.

[Dehne y Noltemeier, 1988] Dehne, F. y Noltemeier, H. (1988). Syntactic and structural pattern recognition. chapter Voronoi trees and clustering problems, pages 185–194. Springer-Verlag New York, Inc.

[Frakes y Baeza-Yates, 1992] Frakes, W. y Baeza-Yates, R. (1992). *Information Retrieval: Data Structures and Algorithms*. Prentice Hall.

[Golynski et al., 2006] Golynski, A., Munro, J. I., y Rao, S. S. (2006). Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, SODA '06, pages 368–373. ACM.

[González et al., 2005] González, R., Grabowski, S., Mäkinen, V., y Navarro, G. (2005). *Practical implementation of rank and select queries*, volume 0109, pages 27–38. CTI Press and Ellinika Grammata.

[Grossi et al., 2003] Grossi, R., Gupta, A., y Vitter, J. S. (2003). High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850. Society for Industrial and Applied Mathematics.

[Grossi y Vitter, 2005] Grossi, R. y Vitter, J. S. (2005). Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407.

[Hazewinkel, 2001] Hazewinkel, M. (2001). *Encyclopaedia of Mathematics, Supplement III*, volume 13. Springer.

[Hennessy y Patterson, 2012] Hennessy y Patterson (2012). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 5a edition.

[Jacobson, 1988] Jacobson, G. (1988). *Succint Static Data Structures*. PhD thesis, Carregie Mellon University.

[Jeffrey, 1991] Jeffrey, U. (1991). Satisfying general proximity/similarity queries with metric trees. In *Information Processing Letters*, page 40.

[Kalantar y McDonald, 1983] Kalantar, I. y McDonald, G. (1983). A data structure and an algorithm for the nearest point problem.

[Li et al., 2003] Li, M., Ma, B., Kisman, D., y Tromp, J. (2003). Patternhunter ii: highly sensitive and fast homology search. *Genome informatics. International Conference on Genome Informatics*, 14.

[Li et al., 2002] Li, M., Ma, B., y Tromp, J. (2002). PatternHunter: faster and more sensitive homology search. 18.

[MacKay, 2003] MacKay, D. J. (2003). *Information theory, inference and learning algorithms*. Cambridge university press.

[Manzini y Rastero, 2004] Manzini, G. y Rastero, M. (2004). A simple and fast dna compressor. *Software - Practice and Experience*, 34:1397–1411.

[Micó y Oncina, 1994] Micó, M. L. y Oncina, José, a. V. E. (1994). A new version of the nearest-neighbour approximating and eliminating search algorithm (aesa) with linear preprocessing time and memory requirements. *Pattern Recognition Letters*, 15(1):9–17.

[Moore et al., 1965] Moore, G. E. et al. (1965). Cramming more components onto integrated circuits.

[Munro et al., 2003] Munro, J. I., Raman, R., Raman, V., y Rao, S. S. (2003). Succinct representations of permutations. In *Proceedings of the 30th international conference on Automata, languages and programming*, pages 345–356. Springer-Verlag.

[Navarro, 1999] Navarro, G. (1999). Searching in metric spaces by spatial approximation. In *Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware*, SPIRE '99, pages 141–. IEEE Computer Society.

[Navarro, 2002] Navarro, G. (2002). Searching in metric spaces by spatial approximation. *The VLDB Journal*, 11(1):28–46.

[Navarro, 2006] Navarro, G. (2006). Succinct data structures. Tutorial.

[Navarro, 2007] Navarro, G. (2007). Estructuras de datos compactas. Tutorial.

[Navarro y Mäkinen, 2007] Navarro, G. y Mäkinen, V. (2007). Compressed full-text indexes. *ACM Comput. Surv.*, 39(1).

[Navarro y Russo, 2008] Navarro, G. y Russo, L. (2008). Re-pair achieves high-order entropy.

[Noltemeier et al., 1992] Noltemeier, H., Verbarg, K., y Zirkelbach, C. (1992). Monotonous bisector* trees - a tool for efficient partitioning of complex scenes of geometric objects. In *Data Structures and Efficient Algorithms, Final Report on the DFG Special Joint Initiative*, pages 186–203. Springer-Verlag.

[Noltemeier H., 1993] Noltemeier H., Verbarg K., Z. C. (1993). Geometric modelling. chapter A data structure for representing and efficient querying large scenes of geometric objects: MB* trees, pages 211–226. Springer-Verlag.

[Okanohara y Sadakane, 2006] Okanohara, D. y Sadakane, K. (2006). Practical entropy-compressed rank/select dictionay.

[Santoyo, 2012] Santoyo, F. (2012). Indices comprimidos para búsqueda por proximidad en series de tiempo. Master's thesis, UMSNH.

[Sayood, 2000] Sayood, K. (2000). *Introduction to Data Compression*. Morgan Kaufmann Publishers, Burlington.

[Sayood, 2003] Sayood, K. (2003). Data Compression. In *Encyclopedia of Information Systems*, pages 423–444. Elsevier Science.

[Shannon, 1948] Shannon, C. E. (1948). A Mathematical Theory of Communication. *Bell system technical journal*, 27.

[Sikora, 2001] Sikora, T. (2001). The mpeg-7 visual standard for content description-an overview. *Circuits and Systems for Video Technology, IEEE Transactions on*, 11(6):696–702.

[Tellez et al., 2011] Tellez, E. S., Chávez, E., y Navarro, G. (2011). Succinct nearest neighbor search. In *Proceedings of the Fourth International Conference on SImilarity Search and APplications*, SISAP '11, pages 33–40. ACM.

[Téllez, 2012] Téllez, E. (2012). *Practical proximity searching in large metric databases*. PhD thesis, UMSNH.

[Téllez y Chávez, 2012] Téllez, E. S. y Chávez, E. (2012). The list of clusters revisited. In *Proceedings of the 4th Mexican conference on Pattern Recognition*, MCPR'12, pages 187–196. Springer-Verlag.

[Vidal, 1994] Vidal, E. (1994). New formulation and improvements of the nearest-neighbour approximating and eliminating search algorithm (aesa). *Pattern Recognition Letters*, 15(1):1–7.

[Yianilos, 1993] Yianilos, P. N. (1993). *Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces*, pages 311–321. Society for Industrial and Applied Mathematics.

[Yianilos, 1998] Yianilos, P. N. (1998). Excluded middle vantage point forests for nearest neighbor search. Technical report, NEC Research Institute, Princeton, NJ.

[Yianilos, 1999] Yianilos, P. N. (1999). Excluded middle vantage point forests for nearest neighbor search. In *In DIMACS Implementation Challenge, ALENEX'99*.

[Zezula et al., 2006] Zezula, P., Amato, G., Dohnal, V., y Batko, M. (2006). *Similarity Search - The Metric Space Approach*, volume 32. Springer.

[Ziv y Lempel, 1977] Ziv, J. y Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 23(3):337–343.