ÍNDICES COMPRIMIDOS PARA BÚSQUEDA POR PROXIMIDAD EN SERIES DE TIEMPO

TESIS

Que para obtener el grado de
MAESTRO EN INGENIERÍA ELÉCTRICA
Opción Sistemas Computacionales

Presenta
Francisco Alberto Santoyo Valdez

Dr. Edgar Leonel Chávez González
Director de Tesis

Universidad Michoacana de San Nicolás de Hidalgo
Facultad de Ingeniería Eléctrica
División de Estudios de Posgrado

Abril 2012

Resumen

Los Sistemas de Recuperación de Información (SRI) han aportado a los motores de búsqueda una nueva tecnología que ha mejorado las consultas de tal forma que gran parte de los usuarios se encuentran satisfechos. Los SRI se concentran en la representación, la búsqueda, y la manipulación de grandes colecciones de objetos como bases de datos de imágenes, audio, texto electrónico y otras representaciones multimedia.

Este proyecto propone la compresión del índice Locality Sensitive Hashing (LSH) para recuperación de información. Se reduce el espacio requerido por el índice LSH utilizando estructuras sucintas que ocupan espacio cercano a la entropía de orden cero H_0. Como segunda propuesta se modifica la estructura del índice y se implementa el índice LSH con un índice de secuencias para reducir el espacio aún más.

El índice LSH es un esquema para búsqueda aproximada basado en funciones hash. La idea principal es aplicar una función hash a todos los objetos de la base de datos de forma tal que asegure que la probabilidad de colisión sea mayor para objetos que son cercanos que para aquellos que son diferentes.

Se describe el uso de estructuras sucintas para hacer una representación comprimida del índice. Debido a que la información crece constantemente, los índices de búsqueda ocupan cada vez más espacio en memoria, por lo tanto, es importante poder almacenar índices que ocupen menor espacio. El espacio utilizado por el índice LSH comprimido se reduce cercano a la entropía de orden cero H_0, tiene un porcentaje de respuesta entre el 80 y 92 % y el tiempo de respuesta se encuentra en el orden de los milisegundos.

La segunda propuesta de compresión, utilizando el índice de secuencias, llamada Locality Sensitive Classification (LSC) reduce el espacio en menor razón que el índice LSH comprimido, el porcentaje de respuesta se mantiene entre el 80 % y 92 % y el tiempo de respuesta se encuentra en el orden de los milisegundos.

Abstract

Information Retrieval Systems have driven search engines with new technology that has improve queries in such way that most users are satisfied. IRS are concerned with representing, searching, and manipulating large collections of objects like databases of images, audio, electronic text and other multimedia representations.

This project propose de compressed index Locality Sensitive Hashing (LSH) for information retrieval. The space in the LSH is reduced by using succint structures that occupy space close to the zero order empirical entropy H_0. As second approach, the structure of the index is modified and the index LSH is implemented using a sequence index in order to reduce space even more.

The LSH index is a representation for similarity search based on hash functions. The main idea is to apply a hash function to all objects in the database in such way that it makes sure that the probability of collision is higher for similar objects than for those that are different.

The use of succinct structures is described to represent a compressed index. Because the increase of information is growing constantly, search index occupy each time more memory space, therefore, it is important to save indexes that occupy less space. The space occupied for the compressed index LSH is reduced close to the zero order empirical entropy H_0, it has a percentage of recall between 80 and 92 % and the time of recall is in the order of de milliseconds.

The second proposal of compression, using the sequence index, called Locality Sensitive Classification (LSC) reduces the space in menor proportion than the compressed index LSH, the percentage of recall is preserve between 80 and 92 % and the time of recall is in the order of the milliseconds.

Contenido

Resumen II
Abstract III
Contenido IV
Lista de Figuras VI
Lista de Tablas VIII
Lista de Algoritmos IX
Lista de Símbolos XI

1. Introducción 1
   1.1. Planteamiento del problema 2
   1.2. Objetivos de la Tesis 3
      1.2.1. Objetivo general 3
      1.2.2. Objetivos particulares 3
   1.3. Estado del arte 3
      1.3.1. Huellas de audio 4
      1.3.2. Búsqueda en espacios vectoriales 5
      1.3.3. Búsqueda en espacios métricos 12
      1.3.4. Aplicaciones existentes 14
   1.4. Descripción de capítulos 22

2. Búsqueda por similitud 23
   2.1. Similitud entre objetos 25
   2.2. El vecino más cercano 26
   2.3. Indices de búsqueda vectoriales 28
   2.4. Indices de búsqueda métricos 36
      2.4.1. Métodos de *particiones de esfera* 37
      2.4.2. Métodos de *particiones de hiperplano* 40
      2.4.3. Métodos *embedded* 43

3. Estructuras de datos sucintas 45
   3.1. Representación en bitmaps 46
      3.1.1. Bitmap de Raman, Raman y Rao 47
      3.1.2. Bitmap de Okanohara y Sadakane 47
      3.1.3. Bitmap de Claude y Navarro 50

3.2. Indices de secuencias 51
3.2.1. Indice de secuencias de Golynski, Munro y Rao 53

4. Indice Locality Sensitive Hashing comprimido 55
4.1. Locality Sensitive Hashing 57
4.2. Representación del índice LSH comprimido 58
4.3. Locality Sensitive Classification (LSC) 66

5. Experimentos y Resultados 69

6. Conclusiones 84
6.1. Conclusiones Generales 84
6.2. Conclusiones específicas 85
6.3. Trabajos Futuros 86

Referencias 87

Lista de Figuras

1.1. Extracción de características 6
1.2. Representación condensada del proceso de recuperación de audio. 7
1.3. Representación de una lista de ciudades con sus coordenadas x y y. 8
1.4. Representación del orden de las ciudades en base a su cordenada x y y. . . 9
1.5. Árbol multiatributo para una estructura con tres atributos. 11
1.6. Representación en celdas de la Figura 1.3. 12
1.7. A la izquierda se presenta un ejemplo de búsqueda por rango en un conjunto de puntos. A la derecha, el conjunto de puntos a la misma distancia de un centro, para diferentes distancias L_s. 14
1.8. Múltiples instancias de hash con tablas independientes. 19
1.9. Ejemplo de una instancia del índice LSH. 20

2.1. Funciones de probabilidad. 25
2.2. *Range tree* de una dimensión para el valor x de coordenadas en la Figura 1.3. 29
2.3. *Quadtree* que representa los datos presentados en la Figura 1.3. En el lado izquierdo se muestra la partición del espacio. En el lado derecho se muestra la representación en árbol. 30
2.4. Secuencia de descomposición de la construcción del *Quadtree* agregando Chicaco y Mobile, Toronto y Buffalo, Denver y Omaha, y Atlanta para los datos correspondientes a la Figura 1.3. 31
2.5. *MX Quadtree* que representa los datos presentados en la Figura 1.3. En el lado izquierdo se muestra la partición del espacio. En el lado derecho se muestra la representación en árbol. 32
2.6. *PR Quadtree* correspondiente a la Figura 1.3 que muestra del lado izquierdo la división del espacio y del lado derecho la representación en árbol. 33
2.7. Representación de un $\mathcal{K}d$-*tree* de 2 dimensiones para los datos en la Figura 1.3. El lado izquierdo muestra la partición del espacio. El lado derecho muestra la representación en árbol. 34
2.8. Representación de las cajas de mínima cobertura para un conjunto de objetos. 35
2.9. Ejemplo de un BKT. En el lado derecho se muestra la división del espacio tomando u_{11} como pivote. En el lado izquierdo se muestra el primer nivel del árbol BKT con u_{11} como raíz. 38
2.10. Espacio dividido utilizando un gh-tree 40

2.11. Ejemplo del primer nivel del GNAT para m =4. Con 4 pivotes, $u2, u3, u5, u9$. 41

3.1. Se muestran ejemplos de Rank y Select. Dado el vector de bits B. 46
3.2. Ejemplo de codificación en Recrank. El bitmap de entrada $\mathcal{B}^1$ convertido a bitmaps densos $\mathcal{B}^1_c$, $\mathcal{B}^2_c$ y $\mathcal{B}^2_e$. 48
3.3. Ejemplo del SArray. x marca las posiciones de los unos en B. 50
3.4. Ejemplo de un índice de secuencias para la secuencia $\mathcal{S}$ = alabar_a_la_alabarda. 52

4.1. Representación de un vector de 24 bits para un frame de la señal de audio. 60
4.2. Representación de la huella de una señal de audio. 60
4.3. Familia $\mathcal{H}$ de funciones hash para los Q-gramas de la base de datos. 61
4.4. Hash para un Q-grama de la base de datos. 62
4.5. Representación de la tabla hash para todas las combinaciones de $\mathcal{H}$ bits. . . 62
4.6. Representación del preprocesamiento de la tabla hash. 63
4.7. Representación de diferentes Familias $\mathcal{H}$. 64
4.8. Representación del bitmapsarray. 65
4.9. Representación del LSH comprimido. 66
4.10. Representación de asignación de símbolos en el índice LSC. 67
4.11. Representación de secuencia concatenando los símbolos de los objetos. . . . 67

5.1. Huellas de audio en la base de datos. 70
5.2. Representación de la base de datos concatenada. 70
5.3. Representación del audio dividido en Q-gramas. 70
5.4. Histograma de distancias para las diferentes configuraciones de Q. 72
5.5. Recall para la búsqueda de 100 consultas de 5 segundos. 73
5.6. Cardinalidad de las listas para una configuración Q=90. 75
5.7. Representación de instancias múltiples de LSH. 77
5.8. Espacio en MB para 1 tabla utilizando diferentes tamaños de familia $\mathcal{H}$. . . 78
5.9. Tiempo promedio en segundos al realizar la búsqueda de 100 consultas de 5 segundos para diferentes Familia $\mathcal{H}$. 79
5.10. Recall utilizando multiples tablas para diferentes configuraciones de LSH. . 80
5.11. División de consulta en varias consultas de tamaño Q. 80
5.12. Se le asigna un voto a cada consulta de objeto que corresponda al audio. . . 81
5.13. Esquema de votación para objetos de tamaño Q que pertenecen a un Identificador de Audio. 81
5.14. Bloques de votación para la localidad de las consultas. 81
5.15. Porcentaje de respuesta para 100 consultas de 5 segundos a nivel canción. . 82
5.16. Espacio para una tabla del LSC. 83

Lista de Tablas

5.1. Tiempo de consulta de 1 tabla para diferentes Q. 76
5.2. Tiempo promedio en segundos al realizar la búsqueda de 100 consultas con multiples tablas. 82

Lista de Algoritmos

1. Preprocesamiento del índice LSH 59
2. Búsqueda del vecino más cercano aproximado 59

Lista de Símbolos

b	Dimensión del espacio.
B	Vector de bits que representa el conjunto $\mathbb{U}$
c	Vocabulario en estructuras sucintas para $\{0, 1\}$
cl	Celda de un registro en la base de datos.
d	Función distancia en espacios métricos.
$d(x, y)$	Distancia entre dos objetos x, y.
DCT	Transformada Discreta Coseno.
DFT	Transformada Discreta de Fourier.
DWT	Transformada Discreta Wavelet.
E	Conjunto de aristas.
FFT	Transformada Rápida de Fourier.
$G = (V, E)$	Grafos generales.
$\mathcal{H}$	Familia de funciones hash.
H^b	Espacio métrico de Hamming de dimensión b.
$H_0(\mathbb{U})$	Entropía de orden cero.
k	k vecinos más cercanos.
l	Atributos de un registro en la base de datos.
LSC	Locality Sensitive Classification.
LSH	Locality Sensitive Hashing.
l_p	Distancia de Minkowski.
m	Número de elementos en $\mathbb{U}$.
M	Mediana de distancias en el índice Vantage Point Tree.
MFT	Fundamento Transformada de Medios.
MFCC	Coeficientes Cepstrales en la Frecuencia Mel.
n	Cardinalidad del subconjunto $\mathbb{U}$.
N	Número de registros en una base de datos.
NN	Búsqueda del vecino más cercano.
NN_k	Búsqueda de los k vecinos más cercanos.
q	Elemento de consulta.
Q	Tamaño de Q-grama en la configuración del índice.
$(q, r)_d$	Elementos a una distancia r respecto a q.
r	Registro de la base de datos.
$\mathbb{R}^b$	Espacio vectorial.
$\mathcal{S}$	Secuencia de símbolos para índices de secuencias.
u	Valores asociados a un atributo de un registro.
$\mathbb{U}$	Base de datos subconjunto de $\mathbb{X}$.
V	Conjunto de vértices.
$\mathbb{X}$	Universo de objetos válido.

Capítulo 1

Introducción

La Recuperación de Información paso de ser una disciplina académica primaria a una necesidad de gran importancia en la actualidad. Motores de búsqueda como Google, Bing y algunos otros, son por mucho los más populares y usados sistemas de recuperación.

La herramienta fundamental de Recuperación de Información es el índice de búsqueda; algoritmos que nos permiten realizar consultas en espacios métricos, vectoriales, etc. Dado un conjunto de objetos $\mathbb{U}$ y un objeto consulta q, se regresan los elementos más cercanos a q dentro $\mathbb{U}$ realizando la menor cantidad de operaciones.

Existen gran variedad de índices por proximidad exactos donde se regresa de manera exacta el objeto más cercano al objeto consulta. Sin embargo, las bases de datos actuales están formadas por objetos que tiene gran variedad de características, por lo que su representación en vectores se tiene de altas dimensiones y la mayoría de los métodos por proximidad exactos caen en un fenómeno conocido como la *maldición de la dimensión* presentado por Bellman en [Bellman, 1961].

Un enfoque distinto a la búsqueda por proximidad exacta es el de la búsqueda por proximidad aproximada donde se regresan objetos cercanos a la consulta, tal vez no el más cercano pero si un conjunto de los más cercanos. En este enfoque se intercambia la exactitud por el tiempo de respuesta o el espacio que ocupan los índices.

En la actualidad existe una gran cantidad de información multimedia que necesita ser consultada o tratada, por lo tanto, es necesario desarrollar índices de búsqueda que

ocupen menos espacio en memoria.

1.1. Planteamiento del problema

La búsqueda de un objeto en bases de datos requiere tiempo y espacio que son costosos. Ejemplo de ello es la búsqueda de un segmento de audio en una biblioteca de audio, buscar una imagen, un documento, etc.

La búsqueda de información es un problema de gran importancia en el área de las ciencias de la computación [Arroyuelo et al., 2012]. El volumen de información hace que su recuperación sea una tarea díficil. La búsqueda secuencial donde se revisan todos los elementos en la base de datos para encontrar el objeto exacto es lenta y no se usa debido a que la cantidad de información crece constantemente. De ahí la necesidad de diseñar estructuras de datos que permitan acelerar la recuperación de información.

Una manera de solucionar el problema de la búsqueda secuencial es utilizar índices por proximidad exactos, es decir, aquellos que regresan de manera exacta el vecino más cercano correspondiente a la consulta como un *Quadtree* presentado por Finkel et al. en [Finkel et al., 1974]. Sin embargo, para bases de datos con objetos de altas dimensiones, estos índices requieren revisar gran número de objetos y se dificulta distinguir objetos cercanos de objetos alejados. Un enfoque diferente que mejora este problema es el de la búsqueda por proximidad aproximada, donde se permite al algoritmo regresar los objetos similares al más cercano a la consulta. Esto debido a que en muchas ocasiones un resultado parecido es tan bueno como el vecino más cercano exacto.

En general, dado un conjunto de objetos $\mathbb{U}$ y un objeto consulta q se requiere encontrar todos los objetos similares a q en $\mathbb{U}$. La similitud está definida por una función distancia d que se utilice. Se pueden localizar todas las objetos similares midiendo la distancia d de la consulta contra todos los objetos del conjunto y quedarse con los objetos a distancia menor; pero es un procedimiento ineficiente. Lo que se busca con la implementación de un índice de búsqueda es acelerar la consulta descartando la mayoría de los objetos que no corresponden a la consulta.

La gran cantidad de objetos en las bases de datos genera índices de búsqueda que

ocupan espacio superior al espacio disponible en memoria principal. Se tiene que recurrir a memoria secundaria que es varios ordenes de magnitud más lenta que la memoria principal. Por lo tanto, es importante realizar investigación en compresión de índices de búsqueda.

La idea de este proyecto es comprimir el espacio que ocupa el índice Locality Sensitive Hashing (LSH) presentado por Gionis et al. en [Gionis et al., 1999] utilizando estructuras sucintas, e implementar el Locality Sensitive Classification (LSC) con índices de secuencias.

1.2. Objetivos de la Tesis

1.2.1. Objetivo general

Crear el índice LSH comprimido capaz de resolver consultas por proximidad.

1.2.2. Objetivos particulares

- Implementar el índice LSH (Locality Sensitive Hashing).
- Incrementar la calidad de recuperación de LSH utilizando tablas múltiples.
- Compresión de las tablas LSH utilizando bitmaps.
- Implementar el Locality Sensitive Classification, que produce índices más pequeños y con parámetros más flexibles.

1.3. Estado del arte

El crecimiento acelerado de la información multimedia ha hecho que se requiera realizar mejoras sobre los algoritmos de búsqueda y recuperación de información, utilizar nuevos enfoques para grandes colecciones de información con objetos de alta dimensionalidad. Un enfoque reciente a la búsqueda por proximidad exacta es el de la búsqueda por proximidad aproximada. La idea es que a veces un objeto parecido es tan bueno como el vecino más cercano exacto [Gionis et al., 1999]. Ejemplos de aplicaciones son las siguientes,

buscar una imagen en una base de datos, un documento en una biblioteca virtual, un segmento de audio en una colección de canciones, etc. Como caso particular para este proyecto de tesis se presenta el proceso para la búsqueda de segmentos de audio en una base de datos de audio ya que en el servidor del laboratorio se cuenta con una base de datos de 10, 252 canciones.

En la actualidad el proceso de búsqueda en archivos de audio se produce generalmente de la siguiente manera. En una primera etapa se obtiene una huella del audio (Audio Fingerprint, AFP) que es una representación de menor tamaño de algunas características representativas que identifican de manera única al audio como presenta Wold et al. en [Wold et al., 1996]. Ejemplo de esto puede ser la energía de la señal, el tono, el volumen, etc. La base de datos esta compuesta entonces por el conjunto de las huellas de audio.

En el peor de los casos al momento de realizar una consulta, se realiza una búsqueda secuencial donde se compara el segmento consulta contra todas las huellas en la base de datos, sin embargo, las bases de datos contienen gran cantidad de objetos lo que hace que la búsqueda secuencial sea muy lenta y costosa en el número de calculos realizados. Una manera diferente para resolver una consulta es regresar un conjunto de huellas similares para elegir al más parecido. Para determinar el parecido entre dos huellas de audio diferentes se utiliza una función distancia que capture la cercanía de objetos de manera correcta. La intención del índice es reducir el número de cálculos realizados para encontrar el vecino más cercano aproximado como menciona Samet en [Samet, 2005].

1.3.1. Huellas de audio

Una manera de representar una señal en menor espacio es extrayendo solo ciertas características que representen de manera única al audio. A esta representación se le llama huella de audio. La señal se divide en segmentos (en ocasiones traslapados) que capturan como va cambiando la señal. Wold et al. en [Wold et al., 1996] clasifica una serie de características acústicas perceptuales de una señal de sonido que se pueden analizar; volumen, la nota, el tono, ancho de banda, etc. En la Figura 1.1 se muestra el proceso de extracción de huellas de audio. Como primer paso se realiza el preproceso. Si la señal se encuentra en más de un canal, se cálcula el promedio de los canales y se convierte en un solo canal. Lo

siguiente es dividir la señal en marcos que por lo general se encuentran traslapados. Se aplica una transformación a la señal como por ejemplo Transformada de Fourier, Transformada Coseno, etc. Se realiza la extracción de características, ya sea la entropía, los MFCC, etc. Como siguiente paso se normalizan los valores y por último se elige una representación de las características que puede ser en forma de vectores, cadenas, etc.

La elección de la AFP que se utilice para la extracción de las características dependerá del problema. Las huellas de audio también conocidas como firmas de audio reemplazan a las señales de audio con fines de identificación. La huella más robusta encontrada en la literatura se basa en una medida de lo plano que resulta ser el espectro de la señal, esta huella forma parte del estándar de MPEG-7 [Allamanche et al., 2001]. La huella más citada en la literatura fue desarrollada por Haitsma y Kalker presentada en [Haitsma y Kalker, 2002] en los laboratorios de Philips Research. Recientemente Camarena en [Camarena y Chávez, 2006] presenta una huella basada en la entropía de la señal. En la Figura 1.2 se presenta de manera condensada el proceso de extracción de huella de audio. Primero, decidir cual característica refleja mejor la percepción de la señal. Luego analizar como modelar las características extraídas. Elegir una función distancia que permita definir un espacio de trabajo y por último crear un índice para el momento de realizar las consultas.

Una vez que se tiene una representación en forma de huella digital se procede a realizar la búsqueda. La búsqueda secuencial de huellas de audio es lenta para la gran cantidad de audio que existe en la base de datos. Se requiere utilizar índices aproximados para acelerar la búsqueda conservando la calidad en la respuesta. Se utilizan los índices para que al momento de la consulta se descarte la mayoría de los objetos que no corresponden o no son similares al objeto que se consulta. Las bases de datos se pueden representar de diferentes maneras. Dependiendo de la forma en que se representen, se tienen diferentes espacios de búsqueda. Se tienen espacios vectoriales donde los datos tienen una estructura definida, y espacios métricos, donde no existe una estructura pero se tiene una función distancia.

1.3.2. Búsqueda en espacios vectoriales

La representación de objetos en forma de vectores es un problema central en el diseño de las bases de datos. Estos vectores pueden representar localidades y objetos en el

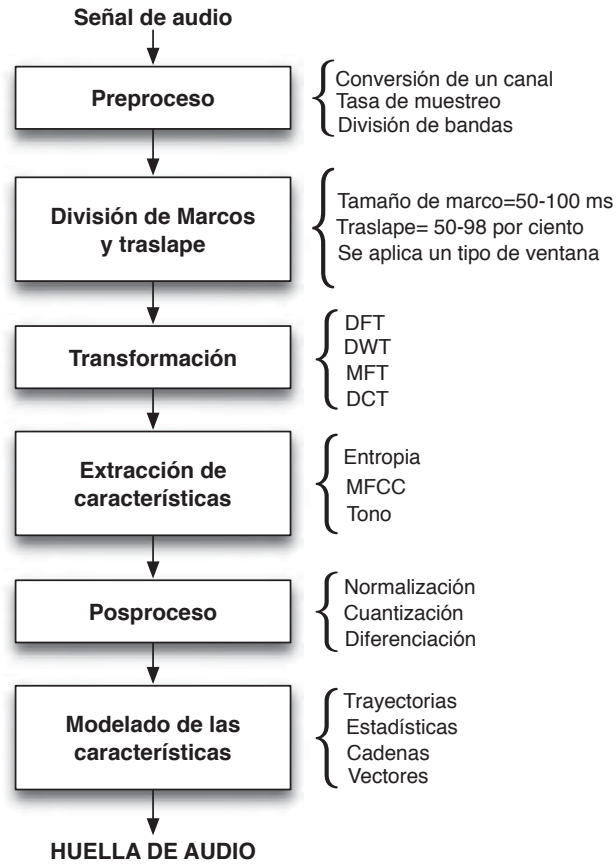


Figura 1.1: Extracción de características

espacio, así como también registros en una base de datos. Como ejemplo de un registro, se considera el registro de un empleado cuyos atributos o características son las siguientes: nombre, dirección, sexo, edad, altura, peso y número de seguro social. Este tipo de registros son manejados en la base de datos como vectores en un espacio en este caso de 7 dimensiones, es decir, una dimensión por cada uno de los atributos. Además las diferentes dimensiones son representadas por unidades de diferente tipo. Formalmente, una base de datos es una colección de registros comúnmente llamado archivo. Existe un registro por cada vector y cada vector contiene múltiples atributos. Para facilitar la recuperación de un registro, se asume cierto orden en alguno de los atributos que forman el vector. Es claro que encontrar un

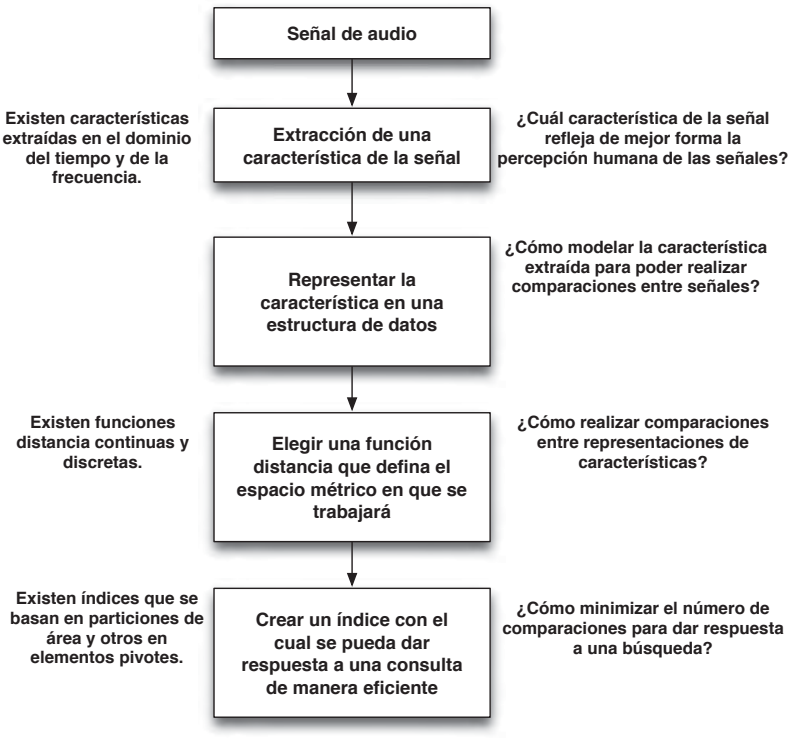


Figura 1.2: Representación condensada del proceso de recuperación de audio.

orden para cualquiera de los atributos no es un problema. El problema está en determinar en base a que atributo se quiere ordenar. Este problema depende generalmente de las siguientes cuestiones.

- Cual es el tipo de los datos, discretos, continuos, etc.
- Que operaciones se van a realizar con los datos.
- Se deben organizar los datos en base a un atributo.
- La base de datos es estática o dinámica.
- Podemos asumir que la base de datos cabe en memoria o se necesita acceso a disco.

Existen muchas representaciones posibles para una base de datos l-dimensional, donde l corresponde al número de atributos.. La facilidad para representar los objetos

depende del número de atributos que tiene. En un extremo se puede representar con un simple vector de bits para representar el atributo del espacio, con un bit reservado para cada registro posible en el espacio multidimensional que indique si se encuentra o no en el archivo. Esta representación es útil cuando el número de atributos l es pequeño ($l < 2$). Sin embargo, la mayoría de las aplicaciones tiene un número de atributos l mayor a dos. Por lo tanto, se necesitan diferentes representaciones.

Nombre	X	Y
Chicago	35	42
Mobile	52	10
Toronto	62	77
Buffalo	82	65
Denver	5	45
Omaha	27	35
Atlanta	85	15
Miami	90	5

Figura 1.3: Representación de una lista de ciudades con sus coordenadas x y y.

La manera más simple de representar la base de datos es en una lista secuencial. En este caso, no se necesita asumir un orden para algún atributo. Como ejemplo, se considera un conjunto de registros que consiste de 8 ciudades y sus localizaciones especificadas por los valores de sus coordenadas x y y como se muestra en la Figura 1.3. Dados N registros con l atributos para búsqueda, revisar todos los atributos requiere $O(N \cdot l)$ de tiempo, ya que, en el peor de los casos, se tienen que revisar los valores de todos los atributos del archivo como menciona Samet en [Samet, 2005].

Otra forma común de representarlos es utilizando un *archivo invertido* presentado por Knuth en [Knuth, 1998]. En este caso el papel de los registros y los atributos está invertido: en lugar de revisar los valores de los atributos en cada registro, se revisan registros que tienen un valor particular en algún atributo. Para facilitar la recuperación, se ordenan las listas de los registros con respecto al valor del atributo. El resultado se conoce como

lista invertida. Se tienen tantas listas invertidas como atributos tiene el registro.

X	Y
Denver	Miami
Omaha	Mobile
Chicago	Atlanta
Mobile	Omaha
Toronto	Chicago
Buffalo	Denver
Atlanta	Buffalo
Miami	Toronto

Figura 1.4: Representación del orden de las ciudades en base a su cordenada x y y.

Existen diferentes maneras de implementar un *archivo invertido*. La manera más simple es mantener separadas listas ordenadas por cada atributo. Por ejemplo, la Figura 1.4 es una representación de las ciudades en la Figura 1.3. Se presentan dos listas ordenadas: una para el valor de la coordenada x y otra para el valor de la coordenada y.

El uso más común de una *lista invertida* es limitar la búsqueda con respecto a un solo atributo. Es claro que la *lista invertida* no es útil para búsquedas que involucran varios atributos por que está agiliza la búsqueda solo para una lista. Se puede realizar la búsqueda por cada uno de los atributos y después realizar la intersección, sin embargo es un procedimiento costoso.

Se han propuesto varias soluciones alternativas. Estas soluciones se pueden dividir en dos clases: La primera clase sugiere una estructura que permita acceder a los valores del atributo primario y luego otra estructura que permita acceder a los siguientes atributos por cada uno de los valores del atributo primario. Un ejemplo de este enfoque incluye el Range Tree presentado por Bentley y Maurer en [Bentley y Maurer, 1979] y que se describe en el Capítulo 2. La segunda clase utiliza una representación de celdas l-dimensionales de rangos de valores para los atributos de modo tal que aquellos objetos cuyos valores de atributos se encuentran en el rango de una celda se encuentran de cierta manera asociados.

La diferencia entre las dos clases de soluciones, es que en la primera existe de manera explícita un orden en el valor de los atributos, mientras que en la segunda clase no existe tal orden.

En general, las soluciones de la primera clase están basadas en la construcción de un árbol u-ario que tiene un nivel por atributo. El primer nivel consiste de un nodo por cada uno de los u diferentes valores del primer atributo. Cada nodo interno de la raíz, digamos un hijo i con un valor asociado a para el primer atributo, sirve como raíz de un subarbol con un nodo interno por cada uno de los posibles valores del segundo atributo. El proceso se repite para los atributos restantes por un total de l hojas, asumiendo l atributos y que la raíz se encuentra a nivel 0. El resultado se conoce como *árbol multiatributo (MAT)* presentado por Kashyap en [Kashyap et al., 1977]. En la Figura 1.5 del lado derecho se muestra un archivo con tres diferentes atributos. El lado izquierdo muestra el árbol multiatributo correspondiente. Cada nodo en el árbol multiatributo tiene tantos nodos internos como valores diferentes tienen los atributos en el conjunto de registros a ese nivel. Así, para el primer atributo A1, se tiene un nodo interno para cada uno de sus valores A, B, C. Después, para los valores del segundo atributo A2, se tiene los valores para cada uno de los nodos internos del primer atributo y así sucesivamente para todos los atributos.

Una representación diferente con estructura de árbol basado en atributos es el *quintary tree* que presenta Lee y Wong en [Lee y Wong, 1980]. Este árbol utiliza una estructura para balancear los niveles o atributos. Se suponen l atributos y un archivo de N registros, la raíz corresponde al valor de primer atributo. Después se particiona el archivo en dos, L y R, con a lo más $N/2$ registros. Además, los dos archivos L y R, asumiendo un espacio l-dimensional, son divididos formando 2 archivos adicionales, LH y RH. Estos cuatro archivos (L, R, LH, RH) como también el archivo H de registros, forman los archivos de cada nodo en una estructura de cinco archivos, de ahí el nombre *quintary tree*.

Se observa que por la manera en que está construido el *quintary tree* se asegura de que en cada comparación por nodo se descarta al menos la mitad de los objetos. Esto permite obtener $O(l + log_2 N)$ tiempo de ejecución para una búsqueda exacta.

Una de las desventajas de utilizar estructuras de árbol es que al momento de una consulta ,en el peor caso, se necesita revisar cada nivel del árbol.

Atributos		
A1	A2	A3
A	1	b
A	1	c
A	2	a
A	2	d
A	3	a
B	2	a
B	3	b
B	3	c
C	3	d

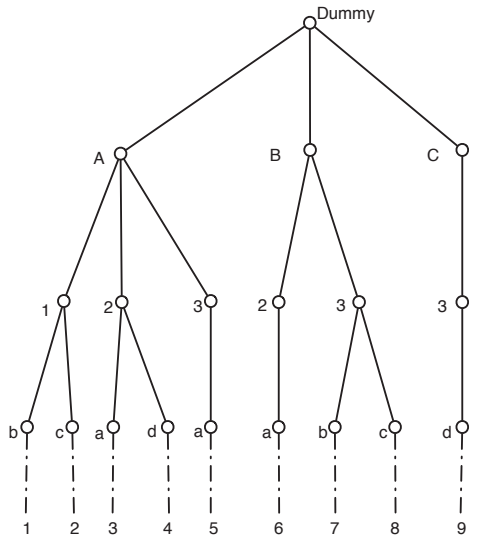


Figura 1.5: Árbol multiatributo para una estructura con tres atributos.

La segunda clase propone la construcción de celdas de rangos. En la Figura 1.6 se muestra un ejemplo de una representación de celdas uniforme correspondiente a los datos en la Figura 1.3 donde cada celda es de tamaño 20×20. Suponiendo un espacio de coordenadas de 100×100, se tienen 25 celdas de igual tamaño. Se considera la esquina inferior izquierda como el origen del espacio de coordenadas.

Los espacios vectoriales describen bases de datos que cuentan con una estructura definida. Existen diferentes métodos que nos permiten realizar consultas en espacios vectoriales. Algunos de ellos basados en árboles multiatributo y otros en celdas de rango. Ejemplos de estos métodos son los siguientes: Range Trees, Quadtrees, MX Quadtrees, PR Quadtrees, etc. Algunos de estos se explican con más detalle en el Capítulo 2.

Todos los métodos anteriores requieren que los datos en la base de datos tengan un estructura. Por otro lado, para las bases de datos que no cuentan con una estructura se tienen métodos basados en Espacios Métricos donde los único que se tiene es una función distancia.

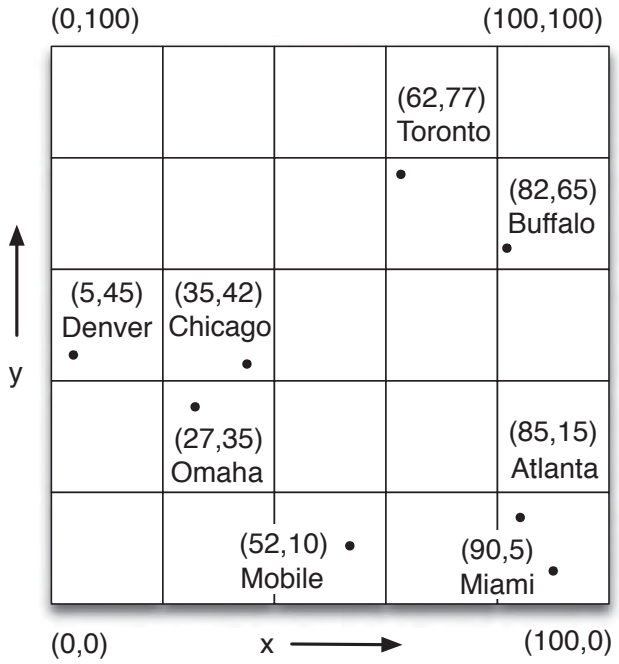


Figura 1.6: Representación en celdas de la Figura 1.3.

1.3.3. Búsqueda en espacios métricos

La búsqueda de objetos en un espacio métrico requiere la definición de espacio métrico. Chávez en [Chávez et al., 2001b] describe que en un espacio métrico $(\mathbb{X}, d)$, se tiene un conjunto de objetos de los que no se conoce su estrutura, solo tiene una función distancia. En el espacio métrico $(\mathbb{X}, d)$, $\mathbb{X}$ es el universo de objetos válidos. El conjunto de objetos donde se realiza la búsqueda es un subconjunto finito $\mathbb{U}$ de tamaño $n = |\mathbb{U}|$.

Se define una función que representa la distancia entre objetos.

$$d : \mathbb{X} \times \mathbb{X} \to \mathbb{R}$$

Si la distancia entre dos objetos es menor que la distancia a los demás objetos, los objetos se consideran similares o cercanos. La función distancia debe de cumplir las siguientes propiedades:

- $\forall x, y \in \mathbb{X}, d(x, y) > 0$ positiva,

- $\forall x, y \in \mathbb{X}, d(x, y) = d(y, x)$ simétrica,
- $\forall x \in \mathbb{X}, d(x, x) = 0$ reflexiva,

y en la mayoría de los casos,

- $\forall x, y \in \mathbb{X}, x \neq y \Rightarrow d(x, y) > 0$ estrictamente positiva

Si la función d es una métrica debe cumplir la desigualdad del triángulo,

- $\forall x, y, z \in \mathbb{X}, d(x, y) \leq d(x, z) + d(y, z)$

si cumple la propiedad anterior entonces el par $(\mathbb{X}, d)$ es un espacio métrico.

Los espacios métricos permiten diferentes tipos de búsqueda de los cuales se tratan los siguientes tres:

- *Búsqueda por rango* $(q, r)_d$. Regresa los elementos que se encuentran dentro de una distancia r respecto a q. Esto es $\{u \in \mathbb{U} \mid d(q, u) \leq r\}$
- *Búsqueda del vecino más cercano NN(q).* Esto es $\{u \in \mathbb{U} \mid \forall v \in \mathbb{U}, d(q, u) \leq d(q, v)\}$.
- *Búsqueda de los k vecinos más cercanos* $NN_k(q)$. Regresa los k elementos más cercanos a q. Esto es, regresa un conjunto $A \subseteq \mathbb{U}$ tal que $|A| = k$ y $\forall u \in A, v \in \{\mathbb{U} \backslash A\}, d(q, u) \leq d(q, v)$.

En la Figura 1.7 se muestra un ejemplo de búsqueda por rango y la representación de los puntos a la misma distancia de un centro bajo cierta distancia L_s. En el modelo de espacio métrico general se permite evaluar una distancia entre objetos, sin embargo si se tuvieran que evaluar todas las distancias sería costoso en el número de operaciones. Generalmente al conjunto de objetos donde se realiza la búsqueda se le llama base de datos, la mayoría de estas bases de datos son de múltiples dimensiones.

En los índices métricos donde no se tiene una estructura, solo se evalúan distancias. Estos índices se pueden clasificar, por ejemplo, están los índices basados en pivotes; en estos algoritmos se escogen arbitrariamente un subconjunto de elementos que se utilizan como referencia para encontrar los demás objetos. También existen los índices basados en particiones compactas donde se divide el espacio en base a ciertos criterios [Samet, 2005].

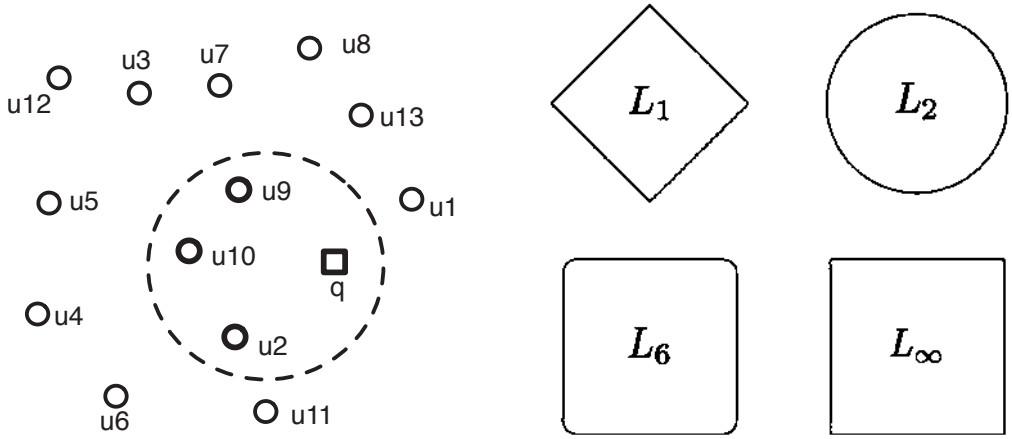


Figura 1.7: A la izquierda se presenta un ejemplo de búsqueda por rango en un conjunto de puntos. A la derecha, el conjunto de puntos a la misma distancia de un centro, para diferentes distancias L_s.

Algunos ejemplos de índices métricos son: BKT, FQT, FHQT, FQA, VPT, MVPT, VT. Estos índices se describen en el Capítulo 2.

Existen varias aplicaciones de la búsqueda multimedia. Algunas de las aplicaciones de los índices utilizando archivos de audio se presentan a continuación.

1.3.4. Aplicaciones existentes

Shazam

El primer proyecto que abordó el problema de búsqueda aproximada en audio fue la empresa Shazam Entertainment LTD. en el año 2000. Shazam abordó el problema de una manera comercial. La idea básica es brindar una herramienta para conectar a la gente con el reconocimiento de audio. La técnica se muestra en [Wang, 2003], es robusta ante el ruido ambiental y desplazamiento en el tiempo. La búsqueda se realiza en el orden de los milisegundos sobre una base de datos de más de dos millones de canciones. Esta aplicación graba 15 segundos de una canción reproducida en el ambiente por medio de un dispositivo móvil. Estos 15 segundos son la consulta y se regresa como resultado el vecino más cercano dentro de la base de datos. La base de datos se encuentra indexada.

La aplicación de Shazam extrae las huellas de las canciones para representar las señales. La respuesta se obtiene buscando una coincidencia entre el hash de la consulta contra los hashes de la base de datos.

La característica a representar en esta huella de audio son los picos de frecuencia altos que ocurren en el espectograma de la señal. Los picos altos soportan degradaciones o ruido dentro de la señal. Como primer paso, se pasa de un espectograma completo a una representación con los picos llamada *constelación*. Los puntos que forman esta *constelación* son pares en el tiempo y la frecuencia. Un punto es considerado un pico si al aplicar una ventana este conserva la frecuencia más alta.

Si se grafican tanto los puntos tiempo-frecuencia elegidos como picos de una consulta y los de una canción que pertenece a la base de datos. Se podría observar que en algún punto de la gráfica de la canción completa, para gráfica de la canción consulta en algún segmento, un número significativo de puntos comienzan a ser iguales con respecto a la gráfica de la canción.

Los picos de frecuencia se toman de acuerdo a un criterio de densidad con el cual se asegura que una serie de picos tiempo-frecuencia, representen de manera significativa el contenido de la señal. Los picos se toman de acuerdo al valor más alto con la justificación que es más probable que los picos de frecuencia se mantengan a pesar de que el espectro de la señal sufra deformaciones.

Para indizar, se toman en cuenta puntos de anclaje. Un punto de anclaje es un punto que se elige para combinarse con una zona objetivo. La combinación de estos valores de un punto de anclaje y una zona objetivo se lleva a cabo secuencialmente concatenando cada punto de anclaje con los puntos de la zona objetivo a la que pertenecen. Adicionalmente a la concatenación de los valores de frecuencia se agrega la diferencia de tiempos en que ocurre cada valor.

Cada combinación de un punto de anclaje con los puntos de una zona objetivo representa un valor hash y se representa con un entero de 32 bits. Además de estos 32 bits, se almancenan tanto el tiempo que existe desde comienzo de la canción al momento en el que ocurre el pico de frecuencia y también un identificador único en la colección para la canción, esto no es parte del valor hash sino información adjunta.

Para realizar la búsqueda, la técnica de obtención de las huellas de audio se aplica en una señal desconocida, de la cual se obtiene un conjunto de valores hash. Cada valor hash que se obtiene se utiliza para buscar y encontrar una colección de huellas de audio por valores hash iguales a los de la consulta. Para cada hash encontrado se toma en cuenta la información adicional con la que cuenta (para este caso la diferencia de tiempo con respecto al comienzo de la canción y el identificador). Los tiempos son agrupados en bins; cuando se terminan de buscar con todos los valores hash que se obtienen de una consulta, se tiene una lista de bins que contienen tiempos de ocurrencia de hashes que aparecen en una canción.

Para dar sentido en el tiempo de las ocurrencias de los valores hash en una canción candidata, se hace uso de histogramas. Para cada bin de la lista que obtiene de la búsqueda de hash en la tabla, se genera un histograma, donde como rangos de clase se toman en cuenta rangos de tiempo. Entonces, lo que se grafica es el desplazamiento de la ocurrencia del hash en una canción de la colección respecto a la señal de consulta. Supongamos que se encuentra un hash en una canción de la colección, el tiempo en el que se encuentra ese valor con respecto a la consulta es:

$$t'_k = t_k + \text{ desplazamiento en el tiempo}$$

Donde t_k es el tiempo en el que ocurre el hash en la colección y t'_k es el tiempo en el que ocurre el mismo hash en la consulta. Para cada par (t'_k,t_k), se calcula:

$$\delta t_k = t'_k - t_k$$

Después se calcula en el histograma de los valores δt_k y se busca por el pico más alto en el histograma. Esto lo hacen ordenando los δt_k y buscando el grupo más grande con el mismo δt_k. Como los valores en cada bin son pequeños debido a la manera en como se calculan los valores hash, el tiempo de generación de un histograma es del orden de microsegundos para un bin. La distancia entre una señal consulta y una canción de la base de datos es el pico más alto que se genera en el histograma.

El algoritmo que utiliza Shazam ha demostrado ser robusto ante niveles signifi-

cativos de ruido y distorsión. Puede identificar música aún cuando existe la presencia de voces, ruido de tráfico e incluso música de fondo.

Cheng Yang

Cheng Yang en [Yang, 2002] presenta una aplicación para reconocimiento de audio. La idea es que las personas en varias ocaciones escuchan una canción en la radio pero no saben el nombre. Esto es un ejemplo clásico de recuperación de audio, donde dado un segmento de una canción se encuentran todos los calces con una base de datos. En el marco de trabajo que se presenta, llamado MACSIS, cada archivo de audio es divido en pequeños segmentos y convertidos en vectores característicos. Todos los vectores son almacenados en un índice para grandes dimensiones conocido como LSH, un algoritmo probabilístico que hace uso de multiples funciones hash. Al momento de la consulta, pequeños segmentos de audio son regresados por el índice y se unen utilizando la transformada Hough presentada por Jain et al. en [Jain et al., 1995].

A continuación se describe el sistema de recuperación basado en índices que consiste de tres fases:

Fase 1. El problema es encontrar una representación que genere objetos que se puedan indexar en un espacio vectorial y cuya distancia mida la similitud entre objetos. Se propone un método basado en una *secuencia característica* que se genera de la siguiente manera;

- Se utiliza la Transformada de Fourier de Tiempo Corto (Short-Time Fourier Transform, STFT) para convertir cada señal en un espectrograma: se divide cada señal en segmentos de 1024-byte de longitud con un traslape del 50 %, se aplica una ventana a cada segmento y se aplica una Transformada Rápida de Fourier (Fast Foutier Transform, FFT) de 2048 bytes en cada segmento. Se toman los valores absolutos del resultado de la FFT y se obtiene un espectograma.

- Se grafica la energía de la señal como una función del tiempo.

- Se identifican los picos en la gráfica de la energía, donde cada pico es definido como un máximo local dentro de un vecindario definido previamente. Esto permite eliminar

máximos locales cuyos valores son seguidos inmediatamente por valores mayores.

- Se extraen componentes de frecuencia cercanos a cada pico. Se toman 180 muestras de frecuencia entre 200 Hz y 2000 Hz.

- Por cada vector u_j de dimensión 180, se convierte a un vector v_j de dimensión 24 que representa 24 diferentes niveles de tono en la siguiente forma: $v_{j,k}(k = 1, ..., 24)$ representa el tono p_k y es definido por: $v_{j,k} = \sum_{i=1}^{6} G_{j,ip_k}$, donde G_{j,ip_k} es el componente de frecuencia del vector u_j a una frecuencia ip_k Hz. Este paso genera una lista de vectores de dimensión 24, v_j, j=1,2,...,n, donde n es el número de picos obtenidos.

- Construir el conjunto de *secuencias características* como sigue: por cada dos picos cercanos separados por menos de D picos, identificar la secuencia de *follow-up* picos que mantienen distancia igual entre ellos, empezando por los dos picos originales. Formalmente una *secuencia característica* está dada por:

$$\{v_s, v_{s+d}, V_{t_s+2(t_{s+d}-t_s)}), V_{t_s+3(t_{s+d}-t_s)}), ..., V_{t_s+(M-1)(t_{s+d}-t_s)}\}$$

  donde M es la longitud de una secuencia, v_s es el punto inicial y d es el espaciado de los picos.

Fase 2. En esta fase se implementa el índice para grandes dimensiones donde se pueden recuperar vectores similares. El índice utilizado es el *Locality Sensitive Hashing (LSH)*, que computa varias funciones hash. En cada instancia, se le aplica una función hash al vector de forma tal que existe cierta probabilidad de que vectores similares tengan un hash con el mismo valor. La clave se encuentra en formar una familia de funciones hash de tal forma que vectores similares obtengan el mismo valor hash. En la Figura 1.8 presentada por Cheng Yang en [Yang, 2002] se ilustra la creación de instancias independientes de índices LSH con diferentes valores hash para el vector dando como resultado múltiples tablas.

El diseño de la función hash que se describe a continuación se presenta en la Figura 1.9 como muestra Cheng Yang en [Yang, 2002]:

- Primero, se aplica una reducción en la dimensión a cada uno de los vectores que elige aleatoriamente un subconjunto de sus dimensiones.

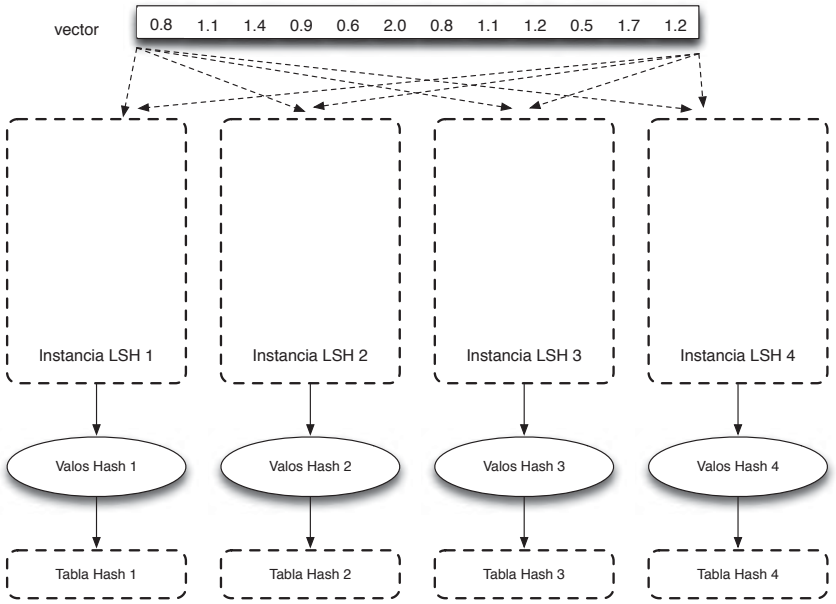


Figura 1.8: Múltiples instancias de hash con tablas independientes.

- Siguiente, se aplica una normalización al vector para una media con valor 0 y varianza con valor 1.
- Se pasa por un grid de baja resolución de tal forma que cada dimensión es convertida en un entero pequeño.
- Finalmente, al vector resultante se le aplica una función hash en donde cada dimensión es multiplicada por un peso aleatorio y se saca la suma para obtener un valor hash final. A este valor se le aplica la operación módulo con el número de buckets.

Fase 3. Se necesita una forma sistemática de evaluar similaridad. En el momento de la consulta es posible que se regresen varios objetos que tengan cierta similitud con la consulta y debemos tener una medida para elegir al más cercano. Para realizar la consulta, dividimos el segmento en un conjunto de *secuencias características* y realizamos la búsqueda en el índice. Cada búsqueda nos va a generar un conjunto de candidatos. Cada candidato es un par que indica el offsets en el tiempo de la consulta y el candidato. Si se grafican los pares en una gráfica 2-D. Si los dos segmentos se encuentran basados en el mismo ritmo, se deben

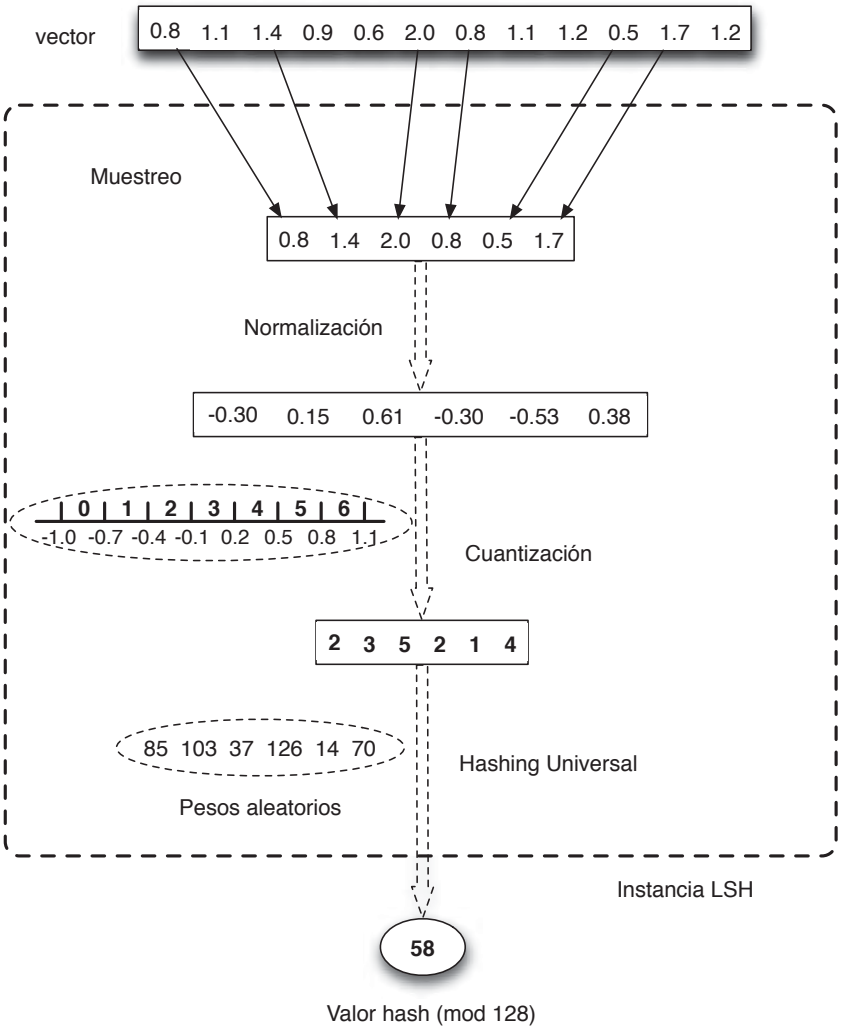


Figura 1.9: Ejemplo de una instancia del índice LSH.

presentar varios de los puntos presentados como líneas. Sin diferencia en el tiempo la línea se debe encontrar a un ángulo de 45 grados. Si existe cambio en el tiempo presentará un diferente ángulo pero debe seguir siendo recta. La línea más larga corresponde a la consulta. Para encontrar la línea recta entre los puntos se utiliza una técnica de visión en ciencias de la computación conocida como la Trasformada de Hough presentada por Jain et al. en [Jain et al., 1995] .

Haitsma y Kalker

Haitsma y Kalker en [Haitsma y Kalker, 2002] presenta un esquema de recuperación de audio. Plantea la situación de un usuario en su automóvil escuchando la radio. El usuario escucha una canción que capta su atención pero no sabe el título de la canción. La idea es crear una estructura que permita consultar un segmento pequeño de audio en una base de datos de canciones.

El esquema propuestó de extracción de la huella de audio se basa en lo siguiente. Se extraen sub-huellas de audio de 32-bits por cada intervalo de 11.6 milisegundos. Un bloque de huella de audio consiste de 256 sub-huellas correspondientes a una granularidad de 3 segundos.

La señal de audio primero se divide en marcos traslapados. Los marcos traslapados tienen una longitud de 0.37 segundos a los que se les aplica una ventana de Hanning con un factor de traslape de 31/32. El máximo traslape asegura que incluso en el peor de los casos las sub-huellas del segmento de audio pueden ser identificadas en las sub-huellas de la base de datos.

Las más importantes características que se perciben se encuentran en el dominio de la frecuencia. Por lo tanto, se aplica Transformada de Fourier a la señal de audio por marcos.

Para extraer las sub-huellas de 32-bits por cada marco, 33 bandas de frecuencia sin traslape son seleccionadas. Estas bandas se encuentran en un rango entre 300Hz a 2000Hz y tiene distribución logarítmica. Si se representa la energía de la banda υ del marco ω por $E(\omega, \upsilon)$ y el υ-ésimo bit de la sub-huella del marco ω por $F(\omega, \upsilon)$, los bits de la sub-huella son formalmente definidos como sigue:

$$F(\omega,\upsilon) = \begin{cases} 1 & si\ E(\omega,\upsilon) - E(\omega,\upsilon+1) - (E(\omega-1,\upsilon) - E(\omega-1,\upsilon+1)) > 0 \\ 0 & si\ E(\omega,\upsilon) - E(\omega,\upsilon+1) - (E(\omega-1,\upsilon) - E(\omega-1,\upsilon+1)) \leq 0 \end{cases}$$

Dos señales de audio de 3 segundos se consideran similares si la distancia de Hamming entre los dos segmentos es menor que cierto umbral T.

Al momento de la consulta se necesita encontrar la huella más similar. Se tiene una tabla con todas las posibles combinaciones de 32 bits para las sub-huellas como entrada. Se

modifica la tabla para convertirse en una tabla hash que apunta a una localidad de 32 bits.

Los índices presentados anteriormente muestran la importancia de crear índices en la recuperación de audio como se presenta en este proyecto de tesis. Sin embargo en ninguno de los casos anteriores se ha propuesto la compresión de esos índices. Este proyecto de tesis propone la compresión del índice utilizando como caso práctico la recuperación del audio. La recuperación multimedia es un problema que se sigue investigando.

1.4. Descripción de capítulos

El resto del documento guarda la estructura que a continuación se presenta. El Capítulo 2 comprende la descripción de técnicas de búsqueda en espacios métricos, búsqueda en espacios vectoriales y con coordenas. Introduce un enfoque reciente de búsqueda aproximada para bases de datos de altas dimensiones. El Capítulo 3 presenta las estructuras de datos sucintas y los índices de secuencias utilizados para la compresión de los índices. El Capítulo 4 describe la propuesta de investigación donde se plantea la compresión del índice aproximado Locality Sensitive Hashing (LSH) con múltiples tablas utilizando una estructuras sucintas. Se presenta un segundo enfoque llamado Locality Sensitive Classification (LSC) donde se utiliza un índice de secuencias para reducir el espacio. El Capítulo 5 presenta los experimentos y resultados obtenidos en los experimentos realizados con la recuperación de información, se muestra el porcentaje de respuesta, el tiempo de respuesta, el espacio utilizado con el uso de multiples tablas, el espacio con la compresión de las tablas y el espacio con la implementación del índice de secuencias. Finalmente, en el Capítulo 6 se proporcionan las conclusiones y se plantea trabajo futuro de investigación.

Capítulo 2

Búsqueda por similitud

Existen muchas áreas de aplicación donde los objetos a representar son de dimensiones muy altas como resultado de representar los objetos con sus características representativas formando vectores característicos. El objetivo de estas áreas es generalmente alguno de los siguientes:

- Encontrar objetos que tienen algunas características en particular.
- Encontrar objetos cuyos valores característicos se encuentran dentro de un rango o aquellos cuya distancia a un objeto es menor que cierto valor.
- Encontrar objetos cuyas características tienen valores similares a un objeto utilizado como consulta o un conjunto de consultas.

Al conjunto de consultas anteriores generalmente se les denomina como *Búsqueda por Similitud*. El problema es que conforme la dimensión de los objetos crece, la búsqueda se hace más difícil. Esto se debe a un efecto conocido como *la maldición de la dimensión*. Este término fué propuestó por Bellman en [Bellman, 1961] donde explica que el número de objetos que necesitan ser revisados crece exponencialmente con la dimensión de los objetos.

La maldición de la dimensión tiene un lazo directo con la búsqueda aproximada en el sentido de que en dimensiones muy altas surge el problema de que no se sabe si la búsqueda del vecino más cercano es representativa o no. Es decir, sea d una función distancia Beyer et al. en [Beyer et al., 1999] remarca que la búsqueda del vecino más cercano no tiene

sentido cuando la razón entre la varianza de la distancia entre dos puntos aleatorios p y q, tomados de las distribuciones de la base de datos y las consultas, y la distancia esperada entre ellos converge a cero cuando la dimensión b crece hasta infinito, esto es,

$$\lim_{b\to\infty} \frac{Varianza[d(p,q)]}{Esperada[d(p,q)]} = 0.$$

En otras palabras, la distancia al vecino más cercano y la distancia al vecino más lejano convergen mientras la dimensión crece.

Asumiendo que d es una distancia métrica y que se cumple la desigualdad del triángulo, una manera alternativa de ver la *maldición de la dimensión* es observando lo siguiente, cuando se trata con objetos de dimensiones altas, la densidad de la función de probabilidad (análoga a un histograma) de las distancias de los diferentes elementos es más concentrada en su media y el valor de su media es mayor. Esto significa que los algoritmos aproximados tendrían que realizar trabajo extra. En el peor de los casos, cuando $d(x,x) = 0$ y $d(y,x) = 1$ para toda $y \neq x$ significa que un algoritmo aproximado debería buscar el objeto consulta con cada objeto de la base de datos. Otra manera de ver por que probabilidades más concentradas se dirigen a búsquedas aproximadas más complicadas es revisando que la desigualdad del triángulo no se puede utilizar para descartar objetos. En particular, la desigualdad del triángulo implica que cada elemento x con $|d(p,q) - d(p,x)| > \epsilon$ no puede estar a una distancia ϵ o menor de un objeto q. Entonces, si examinamos la función de probabilidad de $d(p,x)$ encontramos que cuando ϵ es pequeño mientras la función de probabilidad es grande en $d(p,q)$, la probabilidad de descartar un elemento utilizando la desigualdad del triángulo es el área restante bajo la curva que en este caso es mínima. En la Figura 2.1 se muestra dos distribuciones de probabilidad, al lado izquierdo los objetos se concentran en la media, es decir la varianza es menor, el área que se descarta es el área que no se encuentra sombreada, por lo que se complica la búsqueda aproximada. Al lado derecho se muestra una distribución donde los datos no esán concentrados en la media lo que permite descartar gran parte de la base de datos al momento de la búsqueda aproximada.

Estas observaciones demuestran que las búsquedas del vecino más cercano para dimensiones muy altas son ineficientes ya que es muy difícil diferenciar entre el vecino más cercano y el resto de los elementos. Experimentos realizados por Beyer et al. en

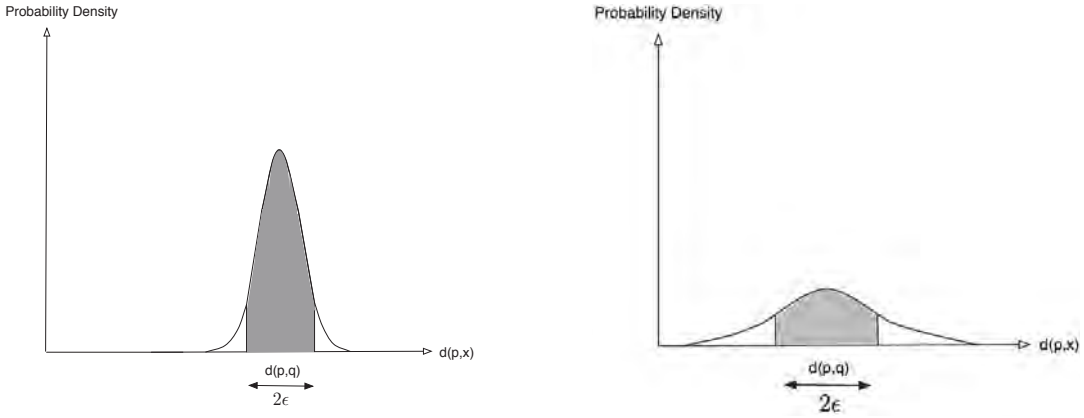


Figura 2.1: Funciones de probabilidad.

[Beyer et al., 1999] muestran que la *maldición de la dimensión* se hace presente para casos a partir de 10-15 dimensiones para una distribución uniforme.

La búsqueda en espacios de altas dimensiones consume tiempo. Además, el determinar similaridad requiere calcular distancias, y el proceso de calcular la distancia puede ser complejo. Por ejemplo, el calcular la distancia Euclideana entre dos puntos b-dimensionales, requiere de b multiplicaciones, $b-1$ sumas y de una raíz cuadrada (que se puede omitir).

Estas funciones de distancia permiten determinar que tan cercanos o similares son dos objetos entre sí.

2.1. Similitud entre objetos

En la mayoría de los casos es difícil determinar lo que significa similitud entre dos objetos. Lo único que se tiene es una función distancia. Por lo general, se requiere que la función distancia cumpla con la desigualdad del triángulo, sea positiva, y sea simétrica. Si cumple con las características antes mencionadas se refiere como una *distancia métrica*. Algunos ejemplos de distancias métricas incluyen las distancias de edición, la distancia de Levenshtein [Levenshtein, 1966] y la distancia de Hamming [Hamming, 1950] para cadenas, y la distancia de Hausdorff [Huttenlocher et al., 1993] que es una distancia para imágenes pero que se puede aplicar a un conjunto de objetos que cumplen cierta métrica entre ellos.

Dada una función distancia, se indizan los objetos utilizando su distancia a algunos elementos previamente seleccionados. Se utiliza el término *distance-based indexing* para describir estos métodos. La ventaja de estos métodos es que utilizan la función distancia para construir el índice, pero una vez construido, al momento de realizar una consulta, se determina un número menor de distancias que si se realizara una búsqueda secuencial.

Existen varios problemas al indizar objetos de altas dimensiones. Debido a la complejidad de manejar datos de altas dimensiones con índices multidimensionales, generalmente se encuentra que el costo de realizar consultas utilizando el índice es mayor que una búsqueda secuencial sobre la base de datos como presenta Beyer et al. en [Beyer et al., 1999]. Esto como resultado de la *maldición de la dimensión* mencionada anteriormente. Sin embargo, la *dimensionalidad intrínseca* de la base de datos es comúnmente menor que la dimensionalidad del espacio. Por ejemplo, los valores de algunos atributos pueden estar relacionados de alguna manera. Alternativamente, algunos atributos pueden no ser tan importantes como algunos otros en discriminar entre objetos. Por lo tanto, pueden ser ignorados como se propone por Hinneburg et al. en [Hinneburg et al., 2000]. Por lo tanto, existe gran interés en encontrar técnicas que permitan reducir la dimensionalidad del espacio.

2.2. El vecino más cercano

A continuación se presenta un esquema general para el problema del vecino más cercano. En lugar de encontrar los k vecinos más cercanos, el enfoque es primero encontrar el vecino más cercano y después utilizar el mismo procedimiento para seguir con el segundo, tercer, cuarto, etc. En otras palabras, se reportan los objetos en el conjunto $S \in \mathbb{U}$, uno por uno, ordenados por la distancia de la consulta $q \in \mathbb{U}$ basados en una función distancia d, donde $\mathbb{U}$ es la base de datos de objetos de donde se obtiene el conjunto. Este algoritmo es una versión general del algoritmo del vecino más cercano incremental presentado por Hjaltason y Samet en [Hjaltason y Samet, 1995].

La razón de buscar los vecinos de manera incremental, es que en ocasiones no se sabe cuantos vecinos se van a necesitar. El vecino más cercano incremental se aplica cuando el espacio de búsqueda puede ser representado de manera jerárquica. Se puede utilizar un R-

tree presentado por Guttman en [Guttman, 1984] para representar la búsqueda jerárquica. Se asume una implementación de un R-tree en donde cada nodo contiene un arreglo de entradas (*key,pointer*). El elemento *key* representa un hiperrectángulo que rodea los datos asociados a un subárbol correspondiente al *pointer*. En un nodo hoja del R-tree, el *pointer* es un identificador de objeto, mientras que un nodo interno apunta a un nodo hijo del siguiente nivel. El número máximo de entradas que existen por nodo se denomina *aridad* y puede ser diferente para nodos hoja y nodos internos.

La clave de la búsqueda incremental del vecino más cercano es que recorre el árbol como el clásico A^*-algoritmo que presenta Russel y Norvig en [Russell y Norvig, 2003]. A cualquier paso del algoritmo, éste visita el elemento con la menor distancia del objeto consulta con respecto a los elementos no visitados. Esto se logra manteniendo una lista global organizada en orden incremental sobre la distancia al objeto consulta, con el elemento más cercano a la consulta al frente de la lista.

El algoritmo del vecino más cercano incremental asume que la distancia entre dos objetos en el conjunto $\mathbb{S}$ es la distancia menor. Sin embargo, en muchas aplicaciones esta definición no se encuentra disponible completamente a causa de que tanto el conjunto $\mathbb{S}$ como el espacio donde se mide su proximidad se representan en una red tal como carreteras, corredores de aire, canales marinos, etc. Los ejemplos anteriores pueden ser modelados como *redes espaciales* presentadas en por Papadias en [Papadias et al., 2003]. En particular, las *redes espaciales* son extensiones de grafos generales $G = (V, E)$, donde ahora los vértices en el grafo tienen posiciones fijas en el espacio. Se han propuesto varias opciones para utilizar la búsqueda incremental del vecino más cercano en *redes espaciales* por Sankaranarayanan et al. en [Sankaranarayanan et al., 2005]. En otras palabras, para encontrar que un objeto o_l es vecino de un objeto o_i a una distancia d de o_i, debe existir algún camino en la *red espacial* entre o_i y o_l de distancia d.

Una manera muy simple de utilizarla es notar que la distancia Euclideana entre dos objetos q y u en la *red espacial* sirve como límite inferior en la distancia entre ellos. Esta observación es la base para el *Incremental Euclidean Restriction (IER)* algoritmo del vecino más cercano de [Papadias et al., 2003]. El algoritmo IER utiliza una cola de prioridad para implementar la búsqueda jerárquica. El algoritmo inicia aplicando la búsqueda incremental

del vecino más cercano para obtener un conjunto con los k candidatos a vecino más cercano. Después se calcula la distancia de la red del elemento q a los k vecinos.

En muchas aplicaciones, obtener resultados aproximados exactos no es crucial. Por lo tanto, los usuarios están dispuestos a intercambiar exactitud por desempeño.

2.3. Indices de búsqueda vectoriales

Los espacios vectoriales describen bases de datos que cuentan con una estructura definida, es decir, se conocen los atributos que forman los objetos y se forman vectores característicos en base al valor de sus atributos como se mencionó en el capítulo 1 en la sección 1.3.2. Existen diferentes métodos que nos permiten realizar una consulta en este tipo de espacio. Algunos de los métodos son los siguientes:

- *Range tree*. Este índice multidimensional fue propuesto por Bentley y Maurer en [Bentley y Maurer, 1979]. El *Range tree* se entiende mejor analizando el caso de la búsqueda por rango de una sola dimensión. El *Range tree* de una dimensión es un árbol binario balanceado donde los datos son almacenados en los nodos hoja y los nodos hoja están unidos en orden por medio de una lista doblemente ligada. Los nodos internos contienen rangos intermedios que permiten discriminar entre los nodos del lado izquierdo o el derecho. En la Figura 2.2 se muestra un ejemplo de un *Range tree* para los valores de la Figura 1.3 en el capítulo 1 con el valor de la coordenada x en las diferentes ciudades.

  Una búsqueda por rango de $[B : E]$ se realiza buscando el nodo con el mayor valor $\leq B$ o el menor valor $\geq B$, y recorriendo los nodos hasta encontrar un nodo hoja con un valor mayor que E. Por ejemplo, se quiere realizar una búsqueda para el rango de búsqueda [28:62] en el *Range tree* de la Figura 2.2. En este ejemplo, se tiene que el valor de búsqueda es la coordenada x de la Figura 1.3. Se desciende el árbol, en este caso, al nodo con el valor mayor ≤ 28, que es Omaha con (27,35). Lo siguiente es reportar los puntos ligados (35,42), (52,10), y (62,77) que corresponden a las ciudades de Chicago, Mobile y Toronto. Se detiene cuando encuentra (82,65), es decir, la ciudad de Buffalo.

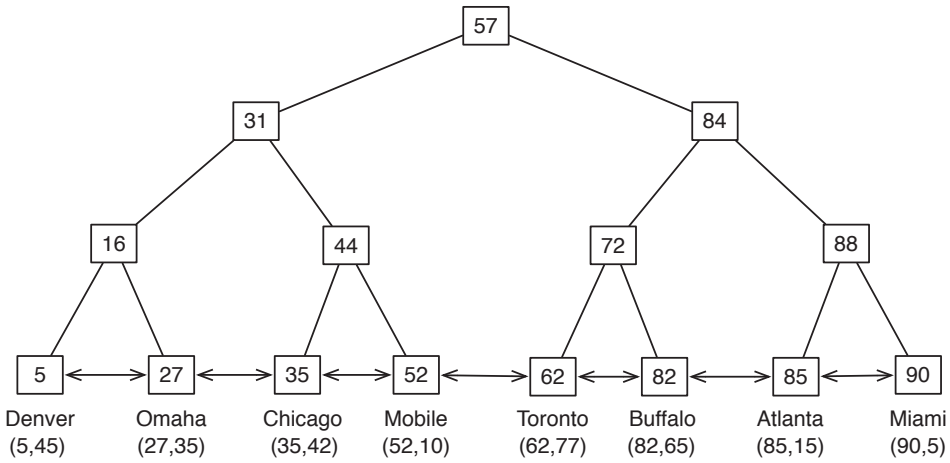


Figura 2.2: *Range tree* de una dimensión para el valor x de coordenadas en la Figura 1.3.

- *Quadtree*. Un *Quadtree* es el resultado de representar una estructura de árbol en forma de cuadrícula para que celdas que se encuentran vacías que son cercanas se puedan unir en una sola celda. Se tienen dos tipos de *Quadtree*. El primero es un *Quadtree* basado en puntos y el segundo es un trie. El *Quadtree* basado en puntos desarrollado por Finkel et al. en [Finkel et al., 1974] es un árbol binario bidimensional. El primer punto que se inserta sirve de raíz. El siguiente punto se inserta en el cuadrante del árbol relevante al punto insertado anteriormente. Ejemplo, en la Figura 2.3 se muestra un *Quadtree* para la Figura 1.6 que representan los datos en la Figura 1.3.

  Insertar el registro r con valores llave (a, e) es muy sencillo. Primero, si el árbol esta vacío, entonces se asigna un nuevo nodo que contenga a r y se regresa un árbol con esta llave como su único nodo. De otro modo, se hace la búsqueda de un nodo h que contenga el registro con los valores llave (a, e) para determinar el cuadrante del subárbol al que pertenece.

  Para realizar borrado existen varias maneras. La primera solución es simple. Sugerida por [Finkel et al., 1974]. Se reinsertan todos los nodos del árbol en el nodo que se elimina. y se realizan los arreglos necesarios. La Figura 2.4 muestra la secuencia de la descomposición en bloques cuando se construye el *Quadtree*, agregando Chicago y

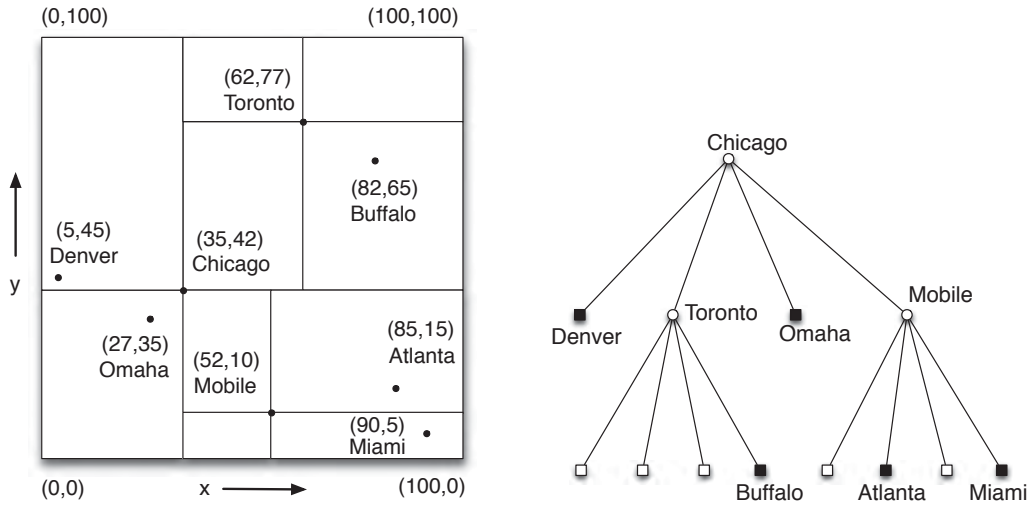


Figura 2.3: *Quadtree* que representa los datos presentados en la Figura 1.3. En el lado izquierdo se muestra la partición del espacio. En el lado derecho se muestra la representación en árbol.

Mobile, Toronto y Buffalo, Denver y Omaha, y Atlanta, datos de la Figura 1.3.

- *MX Quadtree*. Se tienen diferentes maneras de representar datos con una región *Quadtree*. Si el dominio de los datos es discreto, y los valores de todos los atributos son discretos y tienen el mismo tipo y rango, entonces se pueden tratar los puntos como si fueran pixeles negros en una región *Quadtree* que corresponde a una imagen cuadrada. Hunter y Steiglitz en [Hunter y Steiglitz, 1979] tratan los puntos como elementos que no son cero en una matriz cuadrada. Se utiliza el término *MX Quadtree* para describir la estructura que utiliza esta matriz. El *MX Quadtree* está estructurado de manera similar a una región *Quadtree*. La diferencia es que los nodos hoja representados de color negro en la representación de árbol corresponden a regiones de tamaño 1×1, y de blanco las vacías. Estos nodos hoja representan la presencia o ausencia de datos en esa posición de la matriz. La Figura 2.5 muestra una región *MX Quadtree* de $2^3 \times 2^3$ correspondiente a la Figura 1.3. Se obtiene aplicando un mapeo f, tal que $f(x) = x \div 12.5$ y $f(y) = y \div 12.5$, para los valores de coordenadas x y y, respectivamente. Los datos en el *MX Quadtree* son insertados búscando por ellos. Esta búsqueda se basa en la localización de los datos en la matriz. Se realiza de

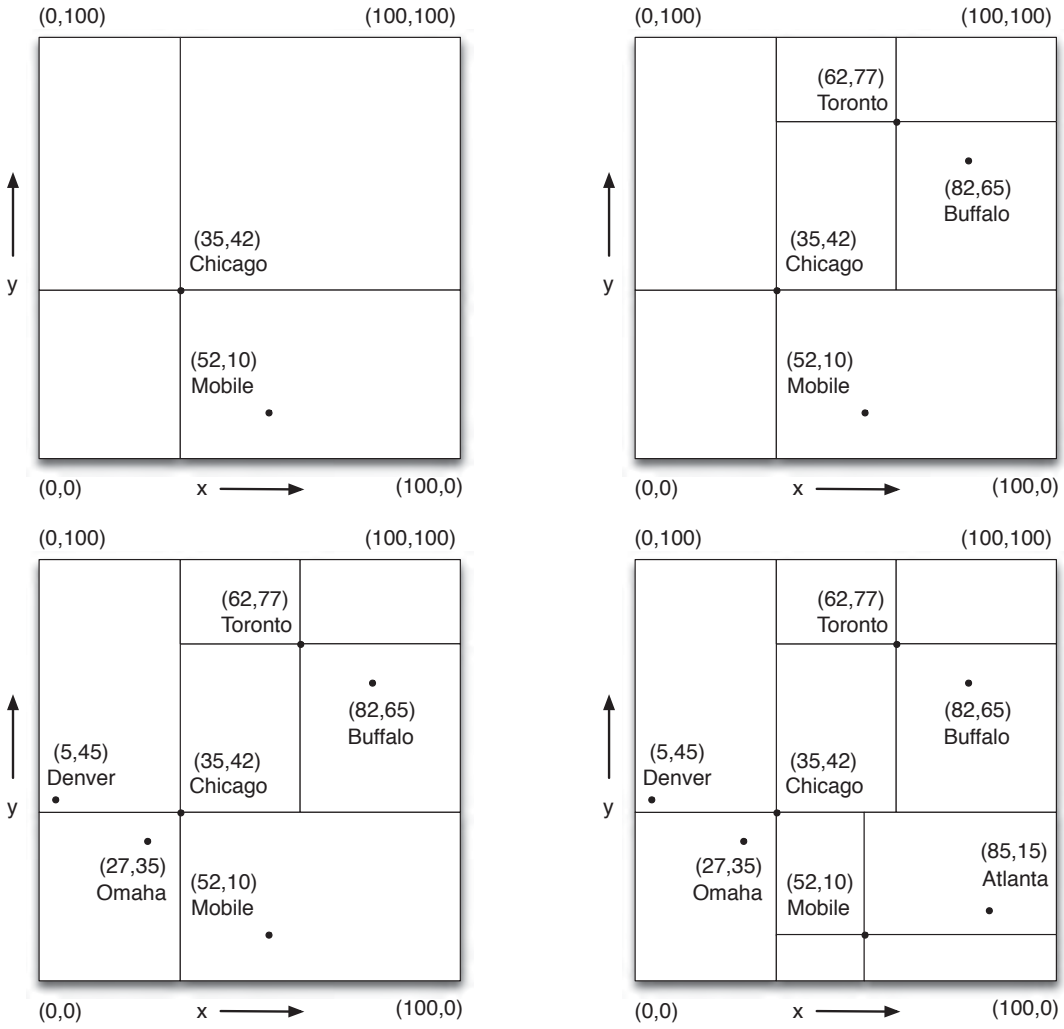


Figura 2.4: Secuencia de descomposición de la construcción del *Quadtree* agregando Chicaco y Mobile, Toronto y Buffalo, Denver y Omaha, y Atlanta para los datos correspondientes a la Figura 1.3.

manera recursiva descendiendo en el árbol comparando la localización de los datos, es decir de los valores de coordenadas para el ejemplo en Figura 1.3. Se revisa el valor de las coordenadas en la raíz del subárbol por el cual se desciende. El resultado de la comparación indica cual de los cuatro subtrees se tiene que descender.

- *PR Quadtrees*. El *MX quadtree* es factible cuando el dominio de los datos es discreto

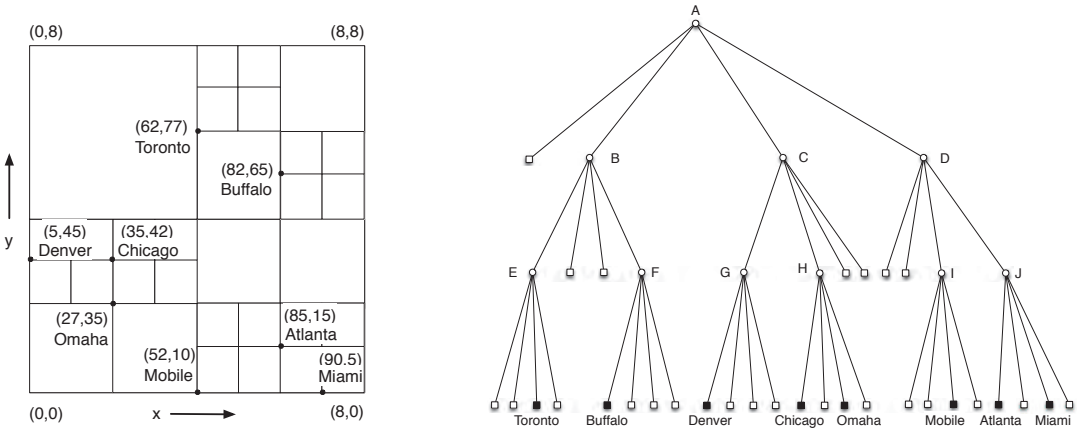


Figura 2.5: *MX Quadtree* que representa los datos presentados en la Figura 1.3. En el lado izquierdo se muestra la partición del espacio. En el lado derecho se muestra la representación en árbol.

y finito. Si este no es el caso, entonces los datos no pueden ser representados utilizando un *MX Quadtree* por que la mínima separación entre los datos es desconocida. Para resolver estos casos, se encuentra el *PR Quadtree* presentado por Anderson en [Anderson, 1983]. Esta organizado de manera similar que la región de un quadtree. La diferencia esta en que los nodos hoja son ya sea vacíos o contienen los valores de coordenadas de los datos. En la Figura 2.6 se muestra un *PR Quadtree* para los datos en la Figura 1.3. En la parte izquierda se muestra la partición del espacio y en la parte derecha se muestra representación en forma de árbol.

- $\mathcal{K}d$*-trees.* Mientras la dimensionalidad del número de atributos en el espacio se incrementa cada nivel de decomposición del quadtree genera muchas celdas. Esto se soluciona utilizando variantes del $\mathcal{K}d$*-tree* presentado por Bentley en [Bentley, 1975] donde para este algoritmo, $\mathcal{K}$ representa la dimensionalidad del espacio. En principio, el $\mathcal{K}d$*-tree* es un árbol binario donde el espacio es dividido en base al valor de solo un atributo en cada nivel. La diferencia entonces con el quadtree es que en el $\mathcal{K}d$*-tree* solo un valor de atributo es evaluado en cada nivel.

  Evaluar el valor de menos atributos en cada nivel tiene sus ventajas. Primero, se realiza solo una evaluación en lugar de d evaluaciones. Segundo, cada nodo hoja en el

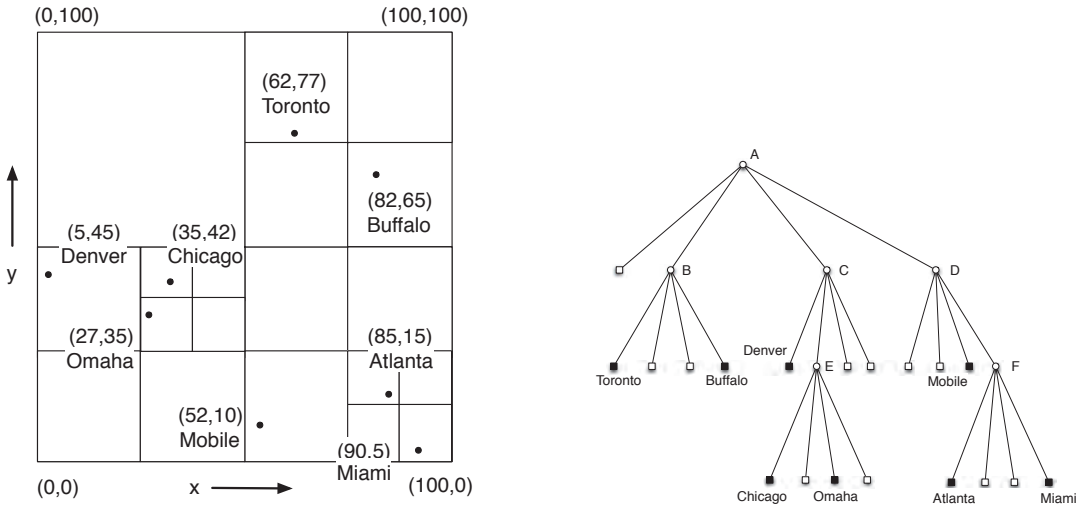


Figura 2.6: *PR Quadtree* correspondiente a la Figura 1.3 que muestra del lado izquierdo la división del espacio y del lado derecho la representación en árbol.

quadtree es costoso en espacio por el número de nulos que se tienen que almacenar.

En el $\mathcal{K}d$-*tree* cada registro es almacenado en un nodo hoja. Adicionalmente, cada nodo contiene dos apuntadores, que pueden ser ya sea null o apuntan a otro nodo en el $\mathcal{K}d$-*tree*. Asociado a cada nodo se tiene un discriminador, que es un número entero entre 0 y $\mathcal{K}-1$ inclusive. Se tiene la siguiente notación: los valores de los $\mathcal{K}$ atributos son llamados $\mathbb{K}_0(P), ..., \mathbb{K}_{\mathcal{K}-1}(P)$, para un nodo P. Los apuntadores son $LOSON(P)$ para el nodo izquierdo y $HISON(P)$ para el nodo derecho. El discriminador se define como $DISC(P)$. El orden impuesto por el $\mathcal{K}d$-*tree* es el siguiente: para cada nodo P en el $\mathcal{K}d$-*tree*, sea j el $DISC(P)$, entonces para un nodo Q en $LOSON(P)$, es verdad que $\mathbb{K}_j(Q) < \mathbb{K}_j(P)$. De la misma manera, para cualquier nodo R en $HISON(P)$, es verdad que $\mathbb{K}_j(R) > \mathbb{K}_j(P)$. Todos los nodos a cualquier nivel tienen el mismo discriminador. La raíz tiene discriminador 0, sus nodos hijo tienen discriminador 1 y así sucesivamente hasta el discriminador $\mathcal{K}-1$.

Insertar un registro r con valores llave (a, e) en un $\mathcal{K}d$-*tree* es muy simple. El proceso es muy similar al de un árbol binario. Primero, si el árbol se encuentra vacío, entonces se genera un nuevo nodo con el registro r, y se regresa el árbol con r como su único

nodo. De otra manera, se busca un nodo h con un registro con los valores llave (a, e). Si h existe, entonces r reemplaza el registro asociado con h.

Eliminar nodos de un $\mathcal{K}d$-*tree* es considerablemente más complicado de lo que es para un árbol binario. Se tiene el caso de que en el $\mathcal{K}d$-*tree* un subárbol no es necesariamente un $\mathcal{K}d$-*tree*. El proceso de borrado es el siguiente, se quiere borrar el nodo (a, e) del $\mathcal{K}d$-*tree*. Si los subárboles de (a, e) están vacías, se reemplaza el nodo (a, e) con un árbol vacío. En caso contrario, encontrar un nodo para reemplazar, sea (c, d) y recursivamente borrar (c, d) del $\mathcal{K}d$-*tree*. En la Figura 2.7 se muestra un ejemplo de un $\mathcal{K}d$-*tree* de 2 dimensiones para los datos en la Figura 1.3. En cada nivel del árbol se evalúa solo uno de los atributos.

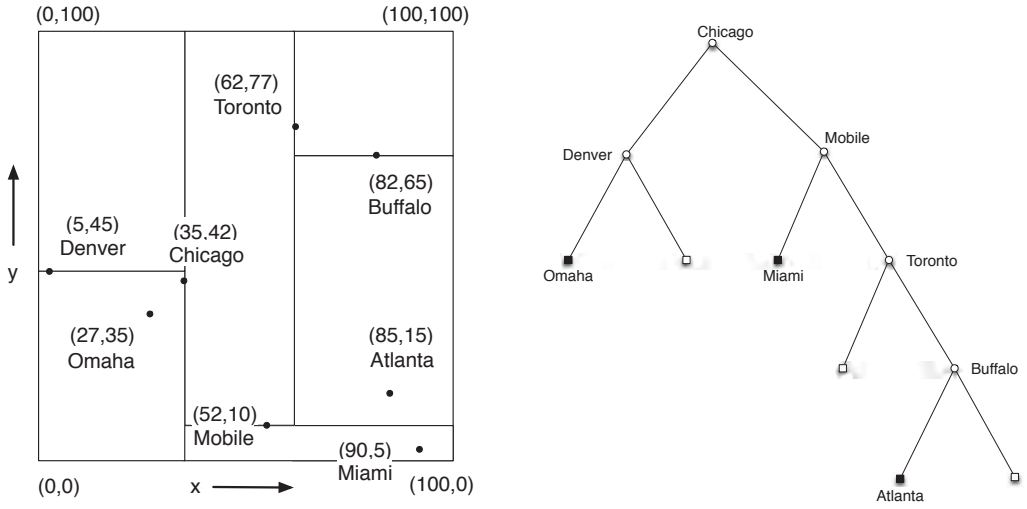


Figura 2.7: Representación de un $\mathcal{K}d$-*tree* de 2 dimensiones para los datos en la Figura 1.3. El lado izquierdo muestra la partición del espacio. El lado derecho muestra la representación en árbol.

- *R-tree*. La meta con este índice es poder tomar la localidad de un objeto a como entrada y regresar los objetos a los que a pertenece. Se asume la existencia de una caja de cobertura mínima para cada objeto. En particular, se agregan las cajas de cobertura de cada D objetos en una sola caja de cobertura mínima que los contiene a todos. Este proceso se repite recursivamente hasta que solo queda una caja. En la Figura 2.8 se muestra un conjunto de objetos {A, 1, E, B, C, D, 2, F, G} con sus

cajas de cobertura mínima S1, S2 y S3 y la caja de cobertura mínima que abarca S1, S2 y S3. El R-tree [Guttman, 1984] se basa en el ordenamiento de los objetos en base a algún criterio. Una vez ordenados los objetos, en los nodos hoja se almacenan los objetos en orden ascendente con los $D-1$ vecinos más cercanos que aún no han sido insertados. El proceso que se menciona anteriormente se describe de manera formal a continuación. Se tiene un conjunto $\mathbb{U}$ de objetos y w es el menor exponente de D tal que $D^w \geq \mathbb{U}$. Se asume que los agregados contienen D elementos con excepción del último que tal vez contiene menos objetos. La jerarquía de los objetos consiste en el conjunto T de conjuntos $\{T_i\}$ $(0 \leq i \leq w)$, donde T_w corresponde al conjunto de cajas de cobertura de los objetos individuales; T_{n-1} corresponde al resultado de agregar las $\mathbb{U}/D$ cajas de cobertura, cada una de las cuales contiene D cajas de cobertura. La jerarquía que se construye generalmente se denomina *pirámide de objetos*. Para realizar la búsqueda en la pirámide de objetos que consiste de T_i $(0 \leq i \leq w)$ para algún objeto en particular a se realiza de la siguiente manera. Se inicia con T_0 que consiste de una caja de cobertura b y se determina si a se encuentra en b. Si no se encuentra, entonces se sale de la caja. Si se encuentra, entonces se examinan los D elementos en D_1 que son cubiertos por b y se repite el proceso utilizando las cajas de cobertura.

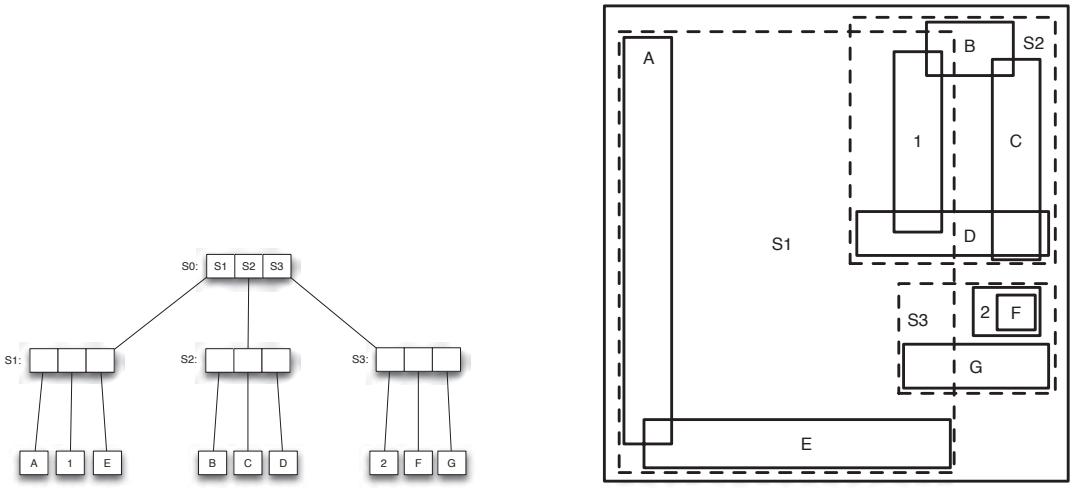


Figura 2.8: Representación de las cajas de mínima cobertura para un conjunto de objetos.

- *R^*-tree.* Presentado en [Beckmann et al., 1990], el R^*-tree es un R-tree que permite mejor descomposición en términos de menor traslape y menor espacio de almacenamiento. El primer cambio que se realiza es minimizar el traslape en el caso de los nodos hoja. La distinción entre nodos internos y nodos hoja es necesaria ya que el algoritmo de inserción inicia en la raíz del árbol y debe procesar nodos internos antes de encontrar un nodo hoja donde finalmente se inserta el objeto. El segundo cambio es que cuando un nodo p se encuentra sobrecargado en el R^*-tree, en lugar de dividir el nodo p como se realiza en el R-tree, primero se revisa si algunos de los objetos en p son más adecuados de estar en otro nodo.

Todos los métodos anteriores requieren que los datos en la base de datos tengan un estructura. Por otro lado, para las bases de datos que no cuentan con una estructura se tienen métodos basados en Espacios Métricos [Samet, 2005].

2.4. Indices de búsqueda métricos

En muchas aplicaciones, no se pueden determinar fácilmente las características de los objetos. La única información que se tiene es una función distancia que indica el grado de similitud entre dos objetos. Por lo tanto, crear el índice se facilita utilizando las distancias entre los objetos. Varios métodos han sido propuestos bajo este esquema. Generalmente, se tiene un conjunto $\mathbb{U}$ de n objetos y una función distancia d, es decir, un *espacio métrico*.

[Uhlmann, 1991] identificó dos esquemas de partición básicos: *particiones de esfera* y *particiones de hiperplano*. En *particiones de esfera*, la base de datos es dividida en base a las distancias sobre un objeto en particular, también llamado *objeto pivote*. En *particiones de hiperplano*, dos objetos p_1 y p_2 son elegidos del conjunto, la base de datos es dividida en base a cual de los dos objetos es más cercano a los objetos restantes. Es decir, todos los objetos en el subconjunto A son más cercanos al objeto p_1 que a p_2, mientras que los objetos en el subconjunto B son más cercanos a p_2.

Como se mencionó en el capítulo 1, los objetos en el índice deben ser parte de un espacio métrico $(\mathbb{U}, d)$. Esto significa que la función distancia d debe de cumplir las siguientes propiedades, donde x, y, $z \in S$:

- $d(x, y) = d(y, x)$ (simétrica)
- $d(x, y) \geq 0, d(x, y) = 0$ sii $x = y$ (positiva)
- $d(x, z) \leq d(x, y) + d(y, z)$ (desigualdad del triángulo)

De las propiedades anteriores, la desigualdad del triángulo es la clave para descartar objetos en una consulta dada.

2.4.1. Métodos de *particiones de esfera*

- *BKT*. Es presentado por Burkhard y Keller en [Burkhard y Keller, 1973]. Proponen una estructura de árbol donde un elemento arbitrario $p \in \mathbb{U}$ es seleccionado como la raíz del árbol. Para cada distancia $i > 0$, se define $\mathbb{U}_i = \{u \in \mathbb{U}, d(u, p) = i\}$ como el conjunto de todos los elementos a distancia i de la raíz p. A continuación, para cada $\mathbb{U}_i$ no vacío, construimos un hijo de p (etiquetado i), donde se construye recursivamente el BKT de $\mathbb{U}_i$. Todos los nodos internos, es decir, aquellos que no son nodos hoja se denominan *pivotes*.

  Dada una consulta q y una distancia r, se empieza a buscar en la raíz y se recorren todos los hijos i tal que $d(q, p) - r \leq i \leq d(p, q) + r$ y se continúa recursivamente. Si se llega a un nodo hoja se procede la búsqueda secuencialmente. En la Figura 2.9 se muestra la figura presenta por Chávez en [Chávez et al., 2001b]. Se divide el espacio tomando como pivote el objeto u_{11} además del primer nivel del árbol tomando como raíz a u_{11}. Se muestra una consulta q y se enfatizan las ramas del árbol que tienen que ser recorridas. En el lado izquierdo de la Figura 2.9 se muestra que tomando como pivote el elemento u_{11} los elementos en la primera rama del árbol son los que rodean a u_{11} en el lado derecho. Los de la segunda rama son los del siguiente radio de cobertura y así sucesivamente. Al momento de la consulta q, y un radio de cobertura r, solo se revisan las ramas correspondientes a los radios de cobertura que cubre r.

- *FQT*. Una modificación del *BKT* es el *Fixed Queries Tree FQT* propuesto por Baeza Yates et al. en [Baeza-Yates et al., 1994]. Esta estructura es básicamente un *BKT* donde solo existe un pivote al mismo nivel.

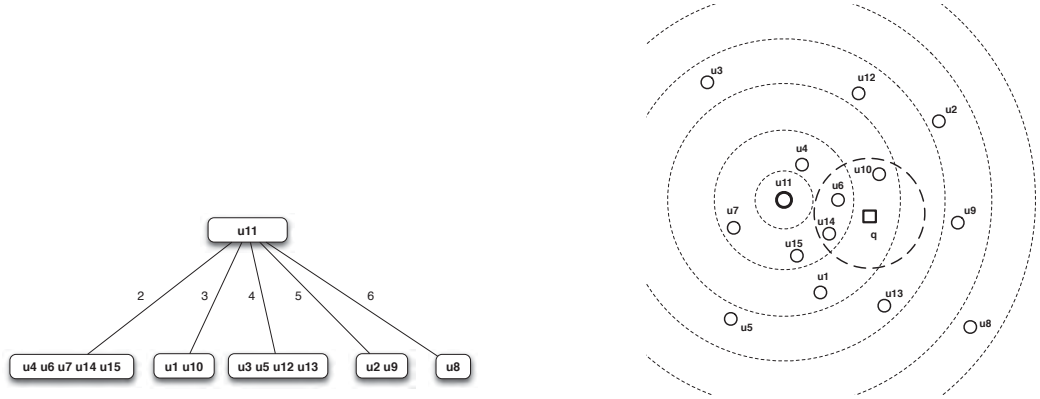


Figura 2.9: Ejemplo de un BKT. En el lado derecho se muestra la división del espacio tomando u_{11} como pivote. En el lado izquierdo se muestra el primer nivel del árbol BKT con u_{11} como raíz.

- *FHQT*. Baeza Yates et al. en [Baeza-Yates et al., 1994] propone una variante *Fixed Height FQT* donde todos los nodos hoja del árbol se encuentran a la misma profundidad h.

- *FQA*. Chávez et al. en [Chávez et al., 2001a] presenta el *Fixed Queries Array FQA*. El FQA es una representación compacta de *FHQT*. Dado cada elemento de la base de datos, se almacena una lista de su distancia a cada uno de los pivotes. Los elementos son ordenados en base a la distancia con el primer pivote, aquellos que se encuentran a la misma distancia se ordenan en base a su distancia con el segundo pivote y así sucesivamente. Se recorre el árbol de izquierda a derecha y se almacenan los elementos en un arreglo. Por cada elemento en el arreglo se calculan h números que representan las ramas a recorrer en el árbol para llegar al elemento desde la raíz.

- *Vp- Tree*. El *vantage point tree* presentado en [Yianilos, 1993], es una de las estructuras más utilizadas. En este método, se seleccionan un pivote p en $\mathbb{U}$, se calcula la mediana r de las distancias de los objetos restantes al objeto p, y se dividen los objetos restantes en dos subconjuntos de aproximadamente igual tamaño S_1 y S_2 como sigue:

$$S_1 = \{o \in S \backslash \{p\} | d(p, o) < r\}$$

$$S_2 = \{o \in S \backslash \{p\} | d(p, o) \geq r\}$$

Por lo tanto, los objetos en S_1 están dentro de la *bola* de radio r sobre p, mientras que los objetos en S_2 se encuentran fuera de la *bola*. Aplicando recursivamente este procedimiento obtenemos un árbol binario, donde un objeto pivote se almacena en cada nodo interno, los subárboles izquierdo y derecho que corresponden a los objetos dentro de la bola y los que se encuentran afuera.

En la búsqueda, por ejemplo, una consulta por rango de un objeto q y un radio de cobertura ϵ, se quiere determinar todos los objetos o tal que $d(q, o) \leq \epsilon$. En particular, cuando se visita un nodo n con un objeto pivote p y un radio de cobertura r, se debe decidir si se visita el subárbol izquierdo o derecho.

- *Mvp- Tree*. Uno de los aspectos más criticados del vp-tree es que tienen una baja *aridad*, es decir, el número de ramas por nivel. Yianilos en [Yianilos, 1993], propone dividir el conjunto $\mathbb{U}$ en m subconjuntos de aproximadamente igual tamaño, en lugar de solo dividirlo en dos. Es decir, $\mathbb{U}$ es divido en $S_1, S_2, ..., S_m$, donde $S_i = \{o \in \mathbb{U} \backslash \{p\} | r_{i-1} \leq d(p, o) < r_i\}$. Aplicando este proceso recursivamente tenemos un árbol m-ario.

- *BST*. En [Kalantari y McDonald, 1983], se propone el *Bisector Tree*. El *BST* es un árbol binario construido recursivamente como sigue. En cada nodo, dos *centros* c_1 y c_2 son seleccionados. Los elementos más cercanos a c_1 que a c_2 se insertan en el subárbol izquierdo y aquellos más cerca de c_2 se insertan en el subárbol derecho. Se almacena el *radio de cobertura* de cada uno de los dos centros, es decir, la máxima distancia del centro a cualquier elemento del subárbol. Al momento de realizar una consulta, accedemos a cada subárbol si $d(q, c_i) - r$ no es mayor que el radio de cobertura. Se descartan las ramas que exceden el radio de cobertura.

- *VT. The Voronoi Tree (VT)* se propone en [Dehne y Noltemeier, 1987] como una mejora del *BST* donde cada nodo tiene dos o tres elementos. Cuando se crea un nuevo nodo en el árbol, el elemento más cercano del nodo padre también es insertado. Los VTs tienen la propiedad de reducir el radio de cobertura conforme se desciende en el árbol.

2.4.2. Métodos de *particiones de hiperplano*

- *Gh-tree*. Uhlmann en [Uhlmann, 1991] definió un árbol métrico utilizando *particiones de hiperplano*, que fue nombrado *generalized hyperplane tree (gh-tree)*. Este método selecciona dos pivotes p_1 y p_2 de manera aleatoria y divide el conjunto de los objetos restantes por medio de la distancia al pivote más cercano:

$$S_1 = \{o \in S \backslash \{p_1, p_2\} | d(p_1, o\} \leq d(p_2, o)\}$$

$$S_2 = \{o \in S \backslash \{p_1, p_2\} | d(p_2, o\} < d(p_1, o)\}$$

En otras palabras, los objetos en S_1 son más cercanos a p_1 que a p_2 y los objetos en S_2 son más cercanos a p_2 que a p_1. Aplicando esto recursivamente obtenemos un árbol binario donde el subárbol izquierdo es S_1 y el subárbol derecho es S_2. En la Figura 2.10 se presenta la figura de un gh-tree. Se eligen como primeros pivotes al elemento a y b. Se divide el espacio en los elementos más cercanos a a y los elementos más cercanos a b. En cada división se eligen 2 pivotes. En la división del elemento a, se eligen los elementos c y d como siguientes pivotes. El espacio se divide en los elementos más cercanos a c y los elementos más cercanos a d. En la división del elemento b, se eligen los elementos e y f. El espacio se divide en los elementos más cercanos a e y los elementos más cercanos a f. Así sucesivamente para las demás divisiones.

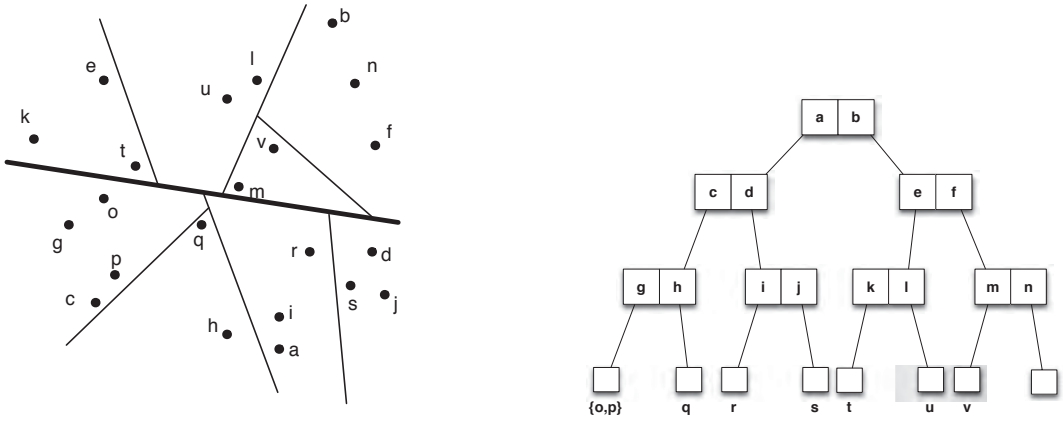


Figura 2.10: Espacio dividido utilizando un gh-tree

- *GNAT.* Presentado por Brin en [Brin, 1995] el *Geometric Near-neighbor Access Tree (GNAT)*, es una generalización del gh-tree, donde más de dos pivotes son seleccionados para dividir la base de datos en cada nodo. En particular, dado un conjunto de pivotes $P = \{p_1, ..., p_m\}$, se divide S en $S_1, ..., S_m$ basado en determinar cual de los objetos en P es el más cercano. Es decir, para cualquier objeto $o \in S \backslash P$, o es parte de S_i si $d(p_i, o) \leq d(p_j, o)$ para toda $j = 1, ..., m$. En caso de empates, i es el índice menor entre los participantes del empate.

  Además de los pivotes y los punteros a los hijos, los nodos del GNAT también almacenan información acerca de los rangos de las distancias entre los pivotes y los objetos en los subárboles. En la Figura 2.11 se muestra un ejemplo del GNAT con cuatro pivotes $u2, u3, u5, u9$ y los elementos asociados a estos.

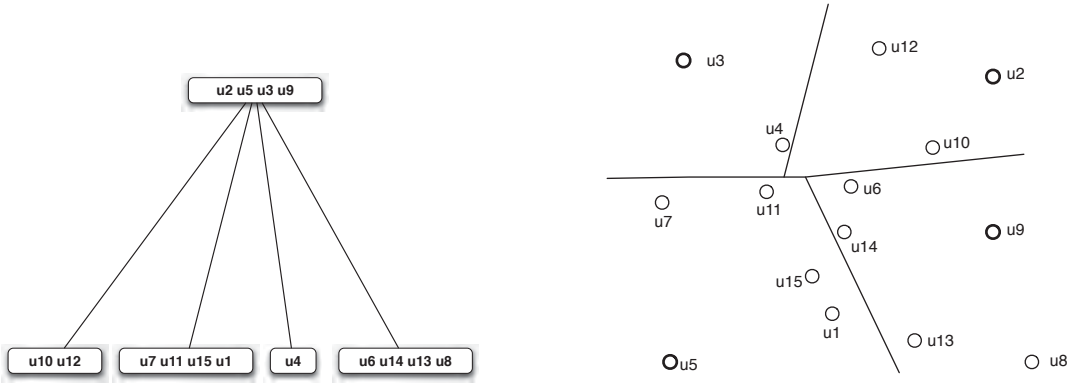


Figura 2.11: Ejemplo del primer nivel del GNAT para m =4. Con 4 pivotes, $u2, u3, u5, u9$.

- *M-Tree.* Los índices basados en la distancia descritos anteriormente son ya sea estáticos, desbalanceados o ambos. Por lo tanto, son inadecuados para situaciones dinámicas que involucran gran cantidad de información, donde el acceso a disco es necesario. El M-Tree presentado por Ciaccia y Patella en [Ciaccia y Patella, 2002] es un índice basado en la distancia que trata de resolver estas deficiencias. En el M-Tree todos los objetos que se van a indizar son representados en los nodos hoja, mientras que una entrada en un nodo interno tiene apuntadores a los nodos del siguiente nivel. Los pivotes en el M-Tree tienen una función similar a los pivotes en GNAT. Para un ob-

jeto o en el subárbol de un nodo b, el pivote p de ese subárbol no es siempre el más cercano a o. Para un nodo interno b, las entradas (p, r, D, T), donde p es el pivote, r es el correspondiente *radio de cobertura*, D es el valor de la distancia, y T es una referencia a un nodo hijo de b. Para todos los objetos o en el subárbol raíz en T, se tiene $d(p, o) \leq r$. El valor almacenado en D es la distancia $d(p, p\prime)$ entre p y el objeto padre $p\prime$ de b.

Al ser dinámico, el M-Tree puede ser construido gradualmente conforme aparecen nuevos datos. El proceso de insertar primero dirige un nuevo objeto a un nodo hoja b, para cada nodo interno en el camino, marcando los nodos hijo que mejor coinciden con el objeto.

Las búsquedas por rango de un objeto q y un radio de búsqueda ϵ se pueden realizar en el M-Tree con una búsqueda primero en profundidad empezando en la raíz. Sea b un nodo siendo visitado, y sea $p\prime$ su pivote padre, es decir, $p\prime$ es el pivote en el padre de b que apunta a b. Para que la distancia D sea aprovechada en las entradas de b, el valor de $d(q, p\prime)$ debe ser propagado en la búsqueda primero en profundidad mientras b es visitado. Se tienen dos casos:

- Si $\mid d(q, p\prime) - D \mid -r > \epsilon$, entonces el subárbol apuntado por T no necesita ser recorrido. En otras palabras, se encuentra un límite inferior en la distancia de q para un objeto en el subárbol apuntado por T.
- De otra manera, si $\mid d(q, p\prime) - D \mid -r \leq \epsilon$. En este caso, no se puede evitar calcular $d(p, q)$. Sin embargo, una vez calculada $d(q, p\prime)$, todavía se puede evitar visitar el nodo apuntado por T si el límite inferior en la distancia de q a cualquier objeto o en T es mayor que ϵ.

Los nodos hoja son procesados de manera similar. Para cada entrada (o, D) in b con un pivote padre $p\prime$, primero se revisa si $\mid d(q, p\prime) - D \mid \leq \epsilon$, y solo para tales entrada se calcula $d(q, o)$ y se revisa si $d(q, o) \leq \epsilon$.

Encontrar los vecinos más cercanos es más complicado. Para la búsqueda de los k-vecinos más cercanos [Ciaccia et al., 1997] propone utilizar la distancia del k-vecino que se encuentra más alejado en lugar de ϵ.

- *Sa-Tree*. Presentado por Navarro en [Navarro, 2002] el Sa-tree utiliza el diagrama de Voronoi que es ampliamente utilizado para la búsqueda del vecino más cercano. Por cada *sitio* p, la celda de Voronoi para p identifica el área más cercana a p que a cualquier otro sitio. Por lo tanto, dada una consulta q, la búsqueda del vecino más cercano solo implica encontrar la celda de Voronoi que contiene a q. En otras palabras, se nos proporciona un expacio $\mathbb{X}$ y un conjunto de objetos $\mathbb{U}$ tal que, por cada consulta q en $\mathbb{X}$ se desea encontrar los objetos más cercanos a q que se encuentran en $\mathbb{U}$.

  Utilizando el Sa-tree es fácil realizar búsquedas exactas.

- *AESA*. Ruiz presenta el Approximating and Eliminating Search Algorithm (AESA) en [Ruiz, 1986] y en [Ruiz, 1994] es un algoritmo de búsqueda del vecino más cercano que tiene un preprocesamiento donde se calculan $O(n^2)$ distancias entre objetos para todos los n objetos en $\mathbb{U}$ y los almacena en una matriz. Al momento de la consulta, la matriz de distancias se utiliza para determinar límites inferiores a objetos cuyas distancias aun no han sido calculadas, basándose en las distancias ya calculadas. El proceso se inicia calculando la distancia del objeto consulta a un objeto arbitrario. En otras palabras, AESA utiliza los n objetos como pivotes cuando se realiza la consulta. Aunque fue diseñado para encontrar el vecino más cercano, AESA puede ser utilizado casi sin modificación para realizar búsquedas por rango. AESA se puede extender fácilmente para realizar búsqueda de los k-vecinos al mantener una lista de los k candidatos para vecino más cercano.

- *LAESA*. AESA es impráctico para grandes bases de datos por el procesamiento necesario. LAESA propuesto por Micó et al. en [Micó et al., 1994] resuelve el problema utilizando un número fijo de pivotes Z cuyas distancias al resto de los objetos son calculadas. Por lo tanto, para n objetos, la matriz de distancias contiene $Z \cdot n$ entradas en lugar de $O(n^2)$ de AESA.

2.4.3. Métodos *embedded*

Los métodos *embedded* son diseñados para facilitar búsquedas aproximadas en un espacio donde solo se tiene el conjunto A de objetos y la distancia d, que indica los valores

entre ellos. La idea es embeber un espacio en otro reduciendo la dimensión. La motivación para desarrollar los métodos *embedded* es que, dado un valor suficientemente alto w, se puede encontrar una función F que traslada los A objetos en un vector de dimensión w, las distancias entre los vectores son preservadas utilizando una función distancia w-dimensional. En otras palabras, se tiene que para cualquier par de objetos a y b, $d(a,b) \approx d\prime(F(a), F(b))$. A continuación se presentan dos índices que siguen este enfoque.

- *Sparse Map.* [Gabriela y Martin, 1999] presenta el Sparse Map, es un método empotrado originalmente propuesto para trasladar una base de datos de proteinas al espacio Euclideano. El Sparse Map comprende dos heurísticas. La primera heurística reduce el número de distancias calculadas al calcular un límite superior $\hat{\mathrm{d}}(o, A_i)$ en lugar del valor exacto $d(o, A_i)$. La segunda heurística reduce el número de dimensiones utilizando un subconjunto de *alta-calidad* R en lugar del conjunto completo.

- *Locality Sensitive Hashing.* Locality Sensitive Hashing (LSH) es una algoritmo aleatorio para búsqueda aproximada desarrollado por [Gionis et al., 1999]. La meta del LSH es encontrar una función hash que preserve la función distancia. Indyk y Motwani implementan los objetos empotrados en $\mathbb{R}^d$ bajo alguna distancia de Minkowski L_p en el espacio de Hamming $\{0,1\}^i$ para alguna i, que es un espacio de vectores cuyos elementos son cadenas de ceros y unos y después utilizan la distancia L_1.

Los métodos anteriores se mencionan como antecedente de la búsqueda estructurada y de la búsqueda métrica. La búsqueda estructurada que generalmente se representa con estructuras de árbol y utiliza pivotes, para altas dimensiones cae comúnmente en el fenómeno de la *maldición de la dimensión*. Por otro lado, los espacios métricos que solo evalúan una función distancia para objetos de altas dimensiones es muy costosa la evaluación de las distancias. El índice que se implementa en este proyecto es el LSH para búsqueda aproximada que es un índice estructurado y que no utiliza pivotes. Su descripción a detalle se presenta en el Capítulo 4.

Capítulo 3

Estructuras de datos sucintas

En las ciencias de la computación, las estructuras de datos sucintas son estructuras de datos modificadas que ocupan espacio en memoria cercano al límite inferior marcado en teoría de la información y que además permite realizar consultas de manera eficiente. El término fue propuestó por Jacobson en [Jacobson, 1988] para codificar vectores de bits, árboles y grafos planos. A diferencia de los algoritmos de compresión sin perdida generalmente usados, las estructuras de datos sucintas retienen la habilidad de realizar las consultas sin descomprimir. Se supone Z como el número óptimo de bits necesarios para almacenar algunos datos de acuerdo con la teoría de la información. La representación de estos datos se denomina de las siguientes maneras:

- Implícita si requiere $Z + O(1)$ bits de espacio.
- Sucinta si requiere $Z + O(Z)$ bits de espacio.
- Compacta si requiere $O(Z)$ bits de espacio.

Los diccionarios de índices sucintos, también llamados diccionarios de Rank / Select, forman la base de un gran número de representaciones sucintas. Esto incluye árboles binarios, arboles K-arios, arreglos, etc. El problema consiste en almacenar el conjunto $\mathbb{U}$ de $\{0, ..., n-1\}$ en una estructura, usualmente representada como un vector de bits B de $\{0, ..., n-1\}$ donde $B[i] = 1$ si $i \in \mathbb{U}$ y $B[i] = 0$ si $i \notin \mathbb{U}$. Al vector de bits B generalmente

se le llama bitmap. Para un vocabulario $c \in \{0, 1\}$ las operaciones básicas del vector de bits B son:

- Rank_c(x, B). El número de elementos c hasta la posición x en B.
- Select_c(i, B). La posición del i-ésimo elemento c en B.

Por lo tanto, para el caso de $c = 1$, Rank_1(x, B) es el número de unos hasta la posición x en B . Select_1(i, B) es la posición del i-ésimo uno en B. En la Figura 3.1 se muestran ejemplos de Rank y Select. Dado el vector de bits B, la operación Rank regresa el número de unos hasta determinada posición. El $\mathsf{Rank}(13, B)$ regresa cuantos unos se tienen hasta la posición 13 en B. $\mathsf{Rank}(20, B)$ regresa el número de unos hasta la posición 20 en B. La operación Select regresa del posición del i-ésimo uno en B. $\mathsf{Select}(1, B)$ regresa la posición del primer uno en B. $\mathsf{Select}(7, B)$ regresa la posición del séptimo uno en B. $\mathsf{Select}(8, B)$ regresa la posición del octavo uno en B.

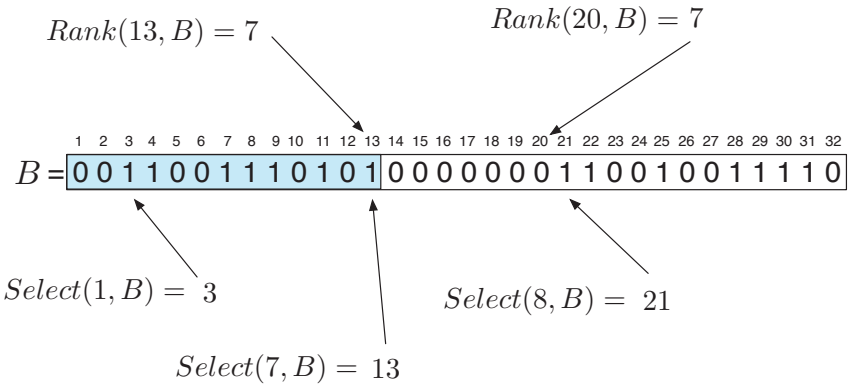


Figura 3.1: Se muestran ejemplos de Rank y Select. Dado el vector de bits B.

La representación anterior es óptima por que existen 2^n conjuntos diferentes en el universo y se necesitan log 2^n = n bits para distinguir entre los diferentes conjuntos.

3.1. Representación en bitmaps

Existen varias representaciones de diccionarios Rank / Select también llamados bitmaps. Diferentes autores han propuesto técnicas para construir bitmaps que permitan rea-

lizar operaciones de Rank y Select en diferentes ordenes de tiempo. A continuación se presentan las propuestas de varios autores.

3.1.1. Bitmap de Raman, Raman y Rao

Raman et al. en [Raman et al., 2007] introducen una estructura teórica de un bitmap que requiere $O(1)$ para Rank y Select y ocupa $\mathcal{B}$(n,m) + o(n) + $O(\log \log m)$ bits de espacio donde $\mathcal{B}$(n,m) es $\lceil \log \binom{m}{n} \rceil$. Se divide el conjunto $\mathbb{U}$ en bloques de $u = \frac{1}{2}\lfloor \log\ m \rfloor$ números. Se almacenan dos arreglos. El primer arreglo A almacena los números n_i en campos de igual tamaño $\lceil \log\ u \rceil$ bits. El segundo arreglo B almacena las cantidades $\mathcal{B}$(n,m); ya que $\mathcal{B}(n, m) \leq u$ estos números también se pueden almacenar en $\lceil \log\ u \rceil$. Para responder Select se realiza lo siguiente. Sea $p = \mid A \mid (= 2m/\log\ m)$. Se almacenan en A las posiciones explícitas de $\mathsf{Select}_1(i, (\log, p)^2)$ para $1 \leq i \leq p/(\log\ p)^2$. Aquí $\mathsf{Select}_1(j)$ da la posición de l en A tal que $\sum_{i=1}^{l-1} A[i] \leq j \leq \sum_{i=1}^{l} A[i]$. Se denomina un bloque denso si el tamaño del bloque es a lo más $(\log\ p)^4$ y disperso en caso contrario.

Para responder $\mathsf{Rank}_1(x, S)$ primero se calcula $i = \lfloor x/u \rfloor$, encontrar el número de elementos en $S_0 \cup ... \cup S_{i-1}$ utilizando la suma parcial para el arreglo A. Indice dentro de la cadena S para obtener la representación de S_i utilizando la suma parcial de B y encontrar Rank de x dentro del conjunto S_i utilizando una visita a la tabla.

3.1.2. Bitmap de Okanohara y Sadakane

Okanohara y Sadakane en [Okanohara y Sadakane, 2006] introducen 4 estructuras de bitmap basadas en diferentes ideas, por lo que cada una tiene diferentes ventajas en términos de tiempo, tamaño y simplicidad. Las cuatro estructuras son las siguientes:

- Esp (Estimating Pointer Information). Dado un bitmap B[0, ..., n-1] con m unos, sea L(B)=n la longitud de B. Se divide B en *super large blocks* (SLB) de tamaño $k = \log^3\ n$. Cada SBL se divide en *large blocks* (LB) de tamaño $l = \log^2\ n$. A su vez cada LB es dividido en *small blocks* (SB) de tamaño $s = \log\ n/2$. Después, de manera independiente se codifica cada SB. Se almacenan los Rank para los LB y SB.

- Recrank. Dado un bitmap B[0, ..., n-1] con m unos, se divide conceptualmente en

bloques $B_0, ..., B_{n/s}$ de tamaño s. Se denomina un bloque cero (ZB, zero block) un bloque donde todos los elementos son ceros, y un bloque no-cero (NZ, non-zero block) un bloque donde existe por lo menos un uno. El bitmap contraído B_c[0,...,n/s - 1] se define como, B_c[i] = 0 si es ZB, y B_c[i] = 1 si B_i es un bloque NZ. Se define un bitmap B_e formado por la concatenación de todos los bloques NZ en B en orden. Se calcula Rank de B utilizando B_c y B_e como sigue:

$$\mathsf{Rank}(x, B) = \mathsf{Rank}(\mathsf{Rank}(\lfloor x/s \rfloor, B_c) \cdot s + (x \bmod t) \cdot B_c[\lfloor x/s \rfloor], B_e)$$

En la Figura 3.2 se muestra un ejemplo de la codificación del bitmap utilizando Recrank. Dado el bitmap B^1, se divide en bloques de tamaño s. B^1_c representa el bitmap contraído formado utilizando los bloques ZB y NZ. Donde se pone un uno si el bloque contiene por lo menos un uno y un cero si el bloque tiene puros ceros. Después, B^1_e es la concatenación de los bloques NZ en orden. Se aplica de manera recursiva para B^1_e. Se tiene que B^1_e es B^2. Se divide B^2 en bloques ZB y NZ. Se determina B^2_c. Se pone un uno para bloques NZ y un cero para bloques ZB. B^2_e es la concatenación de los bloques NZ en B^2.

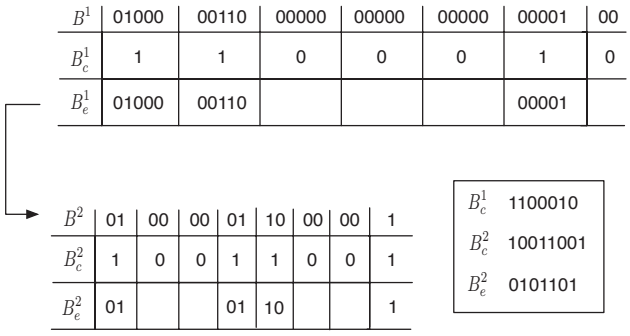


Figura 3.2: Ejemplo de codificación en Recrank. El bitmap de entrada $\mathcal{B}^1$ convertido a bitmaps densos $\mathcal{B}^1_c$, $\mathcal{B}^2_c$ y $\mathcal{B}^2_e$.

- Vcode. Dado un bitmap B[1,..., n-1] con m número de unos, primero se convierte en la secuencia de diferencias d[0,..., m-1], donde d[i] = Select(B, i+1) - Select(B,i) -1 (i = 0, ..., m-1). Después se divide d en bloques B_1, ..., $B_{m/p}$ de tamaño p = O(log

n). Sea T[0 ... m/p -1] arreglos tal que T[i] = $\log\lfloor max_{j=0...p-1} d[ip+j]\rfloor$, $V_i[j]$ son bitmaps de tamaño p que consisten en el conjunto del j-ésimo bit en d en el bloque B_i, y $S[0...m/p-1]$ son arreglos tal que $S[i] = \mathsf{Select}(B, ip)$. Se realiza Select utilizando T, V y S. Se tiene $b = i/p$ y $q = i \bmod p$. Select(S,i) = S[b] + q + $\sum_{i=bp}^{bp+q} d[\, i\,]$, se cuenta el número de unos en los primeros q bits de cada $V_b[0]$ ... $V_b[T]$, después se suman utilizando un shift.

La característica del Vcode es que todas las operaciones están alineadas en bytes, si establecemos p como un multiplo de 8, para cada arreglo de bytes.

- Sarray. Dado un bitmap B[0, ..., n-1] con m unos (m $\ll$ n), se define x[0, ..., m-1] tal que x[i] = Select(i + 1, B). Dado un parámetro t, cada x es divida en *upper* z = $\lceil \log t \rceil$ bits and *lower* w = $\lfloor \log \frac{n}{t} \rfloor$ bits. *Lower* bits son almacenados explícitamente en L[0 ... m - 1] usando $m \cdot w$ bits. *Upper* bits son representados por un bitmap H[0 ... m + t -1] tal que $H[x_i/2^w + i] = 1$ y los demás valores son 0. Utilizando H y L se puede calcular Select(i,B) = (Select(i, H) - i)$\cdot 2^w$ + L[i]. La Figura 3.3 muestra el ejemplo del Sarray presentado por [Okanohara y Sadakane, 2006]. Dado el bitmap B, se tiene el vector x con una longitud correspondiente al número de unos en B. Se almacena en x la posición de los unos en B. Los valores de las posiciones en x son divididos en *Lower* bits y en *Upper* bits. Los *Lower* bits se almacenan explícitamente en L. Los *Upper* bits se almacenan en H donde $H[x_i/2^w + i] = 1$ y los demás valores son cero.

- Darray. Dado un bitmap B[0, ..., n - 1] con $m \simeq n/2$ número de unos, primero se divide B en bloques de tal manera que cada bloque contiene L unos. Sea P_l[0 ... n/L - 1] arreglos tal que $P_l[i]$ = Select(iL, B). Se clasifican estos bloques en dos grupos: si ($P_l[i]$ - $P_l[i-1]$) es mayor que L_2, se almacenan las posiciones de los unos explícitamente en S_l. Si la longitud de un bloque es menor que L_2 se almacenan las posiciones de iL_3-ésimos unos (i=0 ... L_2/L_3) en S_s. Para realizar Select(i,B) en darray, se revisa si el bloque $P_l[\lceil i/L \rceil]$ es más largo que L_2. Si lo es, se busca el valor en S_l que fue almacenado explícitamente. Si no, se revisa el L_3-ésimo valor en S_s y luego se realiza una búsqueda secuencial en el bloque.

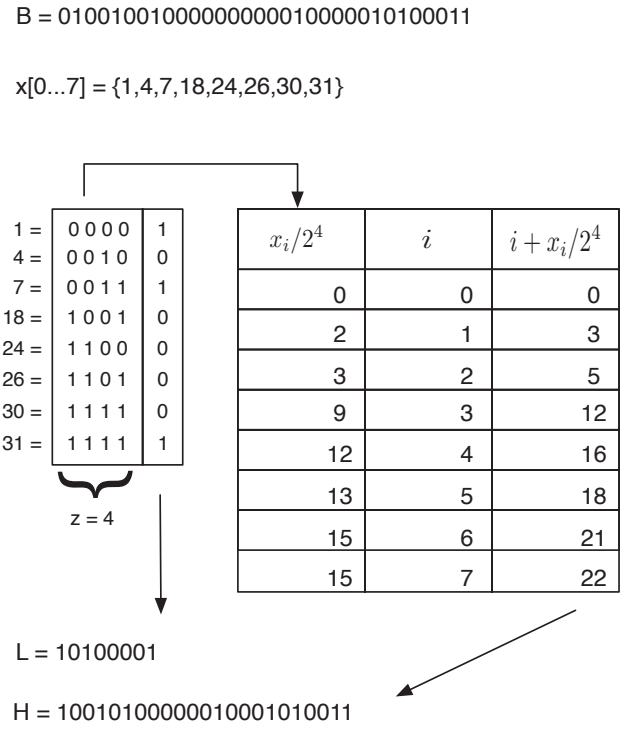


Figura 3.3: Ejemplo del SArray. x marca las posiciones de los unos en B.

3.1.3. Bitmap de Claude y Navarro

Claude y Navarro en [Claude y Navarro, 2009] presentan una implementación del bitmap de Raman et al. [Raman et al., 2007]. En Raman et al. [Raman et al., 2007] se divide la secuencia en bloques de tamaño $u = \frac{\log n}{2}$ y cada bloque se representa como una tupla (c_i, o_i). El primer componente, c_i, representa la *clase* del bloque, que corresponde al número de unos que tiene ese bloque. El segundo componente, o_i, representa el *offset* del bloque dentro de una lista de todos los posibles bloques en la clase c_i. Se definen tres tablas: E, R, y S. La tabla E corresponde a todas las posibles combinaciones de los u bits, ordenados por clase y por *offset* dentro de cada clase. También almacena todas las respuestas para Rank en cada posición de cada combinación. La tabla R corresponde a la concatenación de todas las clases utilizando $\lceil \log (u+1) \rceil$ bits por campo. La tabla S corresponde a la concatenación de todos los *offsets* utilizando $\lceil \log \binom{u}{c_i} \rceil$ bits por campo. En esta implementación se fija $u = 15$

de tal manera que las clases c_i necesitan 4 bits para representar las clases (0-15). Se almacena la tabla E usando enteros de 16 bits. La tabla R se representa por un arreglo utilizando 4 bits por campo. La tabla S almacena cada *offset* utilizando $\lceil \log \binom{u}{c_i} \rceil$ bits. Esta estructura también necesita sumas parciales, una para la tabla R y la otra para la longitud de los o_i en tabla S. Para la tabla R se toma la suma cada b valores y se almacenan estos valores en una nueva tabla cada $sumR$ usando $\lceil \log\ m \rceil$ bits por campo, donde m es el número de unos. Para obtener la suma parcial hasta la posición i se calcula $sumR[j] + \sum_{p=jk}^{i} c_p$ donde $j = \lfloor i/b \rfloor$, y la suma de los c_p se realiza secuencialmente sobre las entradas de R. Las posiciones en S se presentan de la misma manera. Para responder Rank hasta la posición i primero se calcula $sum(R, \lfloor i/u \rfloor) = \sum_{j=0}^{\lfloor i/u \rfloor} R_j$, el número de unos antes del inicio del bloque i. Y después Rank dentro del bloque hasta la posición i en la tabla E. Para Select se tiene la i-ésima ocurrencia del caracter a, se inicia en el nodo hoja donde se representa a y se realiza Select(x, i) donde, x es el bit de a correspondiente a este nivel. Al llegar a la raíz, la posición final es el resultado.

En los experimentos realizados se implementó el LSH utilizando el SArray por que el Select es muy rápido, el SArray comprime cercano a la entropía de orden cero H_0, sin embargo, a cierto tamaño de familia $\mathcal{H}$, como se muestra en los resultados del capitulo 4, el SArray resulta ocupar más espacio que la representación plana. Para esto se utiliza otro enfoque. Se realiza la implementación del LSH utilizando índices de secuencias. Se concatena la base de datos en forma de secuencia y se utiliza la variación del índice de secuencias presentado en [Golynski et al., 2006].

3.2. Indices de secuencias

Los índices de secuencias son una generalización de bitmap para alfabetos mayores a dos símbolos. Dada una secuencia $\mathcal{S}$ de tamaño n sobre un alfabeto de tamaño $|\sigma|$, para $c \in \sigma$ se pueden realizar las siguientes operaciones.

- Rank(c, p). Número de ocurrencias de el caracter c antes de la posición p.
- Select(c, r). Posición de la r-ésima ocurrencia del caracter c en la secuencia.

- Access(p). Regresa el p-ésimo caracter en la secuencia.

Estas estructuras de datos sirven para indizar secuencias y permiten los siguientes tipos de consultas:

- *Existencial*. Determina si el patrón ocurre en el texto.
- *Cardinalidad*. Regresa el número de ocurrencias del patrón en el texto.
- *Locate*. Regresa las posiciones donde inicia cada ocurrencia del patrón en el texto.
- *Contexto*. Regresa los carácteres que se encuentran inmediatamente después o antes de una ocurrencia.

Se almacenan σ bitmaps $B_c[1, .., n]$ tal que $B_c[i] = 1$ si y solo si $\mathcal{S}[i] = c$. Entonces, $\mathsf{Rank}_c(i, \mathcal{S}) = \mathsf{Rank}_1(i, B_c)$. $\mathsf{Select}_c(i, \mathcal{S}) = \mathsf{Select}_1(i, B_c)$. En la Figura 3.4 se muestra un ejemplo de índice de secuencias para la secuencia $\mathcal{S}$ = alabar_a_la_alabarda, con σ = {_, a, b, d, l, r }. Se tienen tantos bitmaps como elementos en sigma. En el conjunto de bitmaps se tiene un solo uno por columna que corresponden al carácter en esa posición.

$\mathcal{S}$ = a l a b a r _ a _ l a _ a l a b a r d a
σ = {_, a, b, d, l, r}

	a	l	a	b	a	r	_	a	_	l	a	_	a	l	a	b	a	r	d	a
_	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0
a	1	0	1	0	1	0	0	1	0	0	1	0	1	0	1	0	1	0	0	1
b	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
d	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
l	0	1	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
r	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	

Figura 3.4: Ejemplo de un índice de secuencias para la secuencia $\mathcal{S}$ = alabar_a_la_alabarda.

3.2.1. Indice de secuencias de Golynski, Munro y Rao

Para este proyecto de tesis se utiliza el índice de secuencias presentado por Golynski et al. en [Golynski et al., 2006]. En esta estructura se presenta la secuencia de tamaño n como una tabla T de $t \times n$ donde las filas se indizan de $1, ..., t$ y las columnas por su posición en la secuencia. La entrada $T[c, i]$ indica si el carácter c ocurre en la posición i en la secuencia. Sea A un bitmap de tamaño tn que se obtiene escribiento T en orden de las filas. Exite una relación simple entre las operaciones $\mathsf{Rank}(c, j)$ y $\mathsf{Select}(c, i)$ en la secuencia original con $\mathsf{Rank}_A(1, j)$ y $\mathsf{Select}_A(1, i)$ en el bitmap A:

- $\mathsf{Rank}(c, i) = \mathsf{Rank}_A(1, (c-1)n + i) - \mathsf{Rank}_A(1, (c-1)n)$
- $\mathsf{Select}(c, i) = \mathsf{Select}_A(1, \mathsf{Rank}_A(1, (c-1)n) + i)$

Se divide A en bloques de tamaño t. Se definen las operaciones Rank-b y Select-b en relación a estos bloques. Donde Rank-$b(i) = \mathsf{Rank}(i)$ si i es un múltiplo de t e indefinido de otra manera. Select-$b(i)$ es el número de bloque en donde ocurre el i-ésimo uno. Se define la cardinalidad de un bloque i como $k_i = \mathsf{Rank}$-$b(it) - \mathsf{Rank}$-$b((i-1)t)$. Se contruye el bitmap B escribiendo la cardinalidad de los bloques en unario. $B = 1^{k_1}01^{k_2}0...1^{k_n}0$. Se tiene entonces,

- Rank-$b(it) = \mathsf{Rank}_B(1, \mathsf{Select}_B(0, i))$,
- Select-$b(i) = \mathsf{Rank}_B(0, \mathsf{Select}_B(1, i))$.

Entonces el Rank y Select completos se determinan de la siguiente manera. Para el i-ésimo bloque, se define:

- Rank-$l_i(j) = \mathsf{Rank}_A(1, j + it) - \mathsf{Rank}_A(1, it)$,
- Select-$l_i(j) = \mathsf{Select}_A(1, j + \mathsf{Rank}_A(1, (i-1)t))$.

El Rank y Select completos se tienen como sigue:

- $\mathsf{Rank}(j) = \mathsf{Rank}$-$b(it) + \mathsf{Rank}$-$l_i(j - it)$, donde $i = \lfloor j/t \rfloor$,
- $\mathsf{Select}(j) = \mathsf{Select}$-$l_i(j - \mathsf{Rank}$-$b((i-1)t)) + (i-t)^t$, donde $i = \mathsf{Select}$-$b(j)$.

Para la compresión del índice LSH se utilizó el bitmap presentado por Okanohara y Sadakane en [Okanohara y Sadakane, 2006]. Para la modificación del índice LSH en el índice LSC se implementó una variación del índice de secuencias de Golynski et al. en [Golynski et al., 2006].

Capítulo 4

Indice Locality Sensitive Hashing comprimido

El problema del vecino más cercano surge en gran variedad de problemas con bases de datos, generalmente en problemas de búsqueda aproximada. Las bases de datos contienen objetos con gran variedad de características lo que causa que su representación sea de grandes dimensiones. La alta dimensionalidad de estos objetos limita la búsqueda exhaustiva donde se revisan todos los objetos de la base de datos por que es costosa en tiempo. Es por esto, que recientemente se ha incrementado el interés por diseñar estructuras índice/búsqueda que realicen busquedas aproximadas sobre bases de datos de alta dimensionalidad como son bases de datos de imágenes, colecciones de documentos, series de tiempo, y secuencias de genoma humano. Desafortunadamente, todas las técnicas actuales caen en el problema de *la maldición de la dimensión* que presenta Bellman en [Bellman, 1961].

El Locality Sensitive Hashing (LSH) presentado por Gionis en [Gionis et al., 1999] es un esquema para la búsqueda aproximada basado en funciones hash. La idea principal es aplicar una función hash a todos los puntos de la base de datos de forma tal que se asegure que la probabilidad de colisión sea mayor para objetos que son cercanos que para aquellos que son diferentes.

El problema de la búsqueda aproximada es de gran importancia en gran variedad de aplicaciones; algunos ejemplos son, compresión de datos, bases de datos y minería de

datos, recuperación de información, bases de datos de audio e imágenes, reconocimiento de patrones, y en estadística y análisis de datos. Usualmente las características de los vectores de interés son representadas como puntos en $\mathbb{R}^b$ donde b es el número de coordenadas del objeto vector y una distancia métrica es utilizada para medir la similitud entre dos objetos. La idea general es que en muchas ocasiones, no es necesaria la respuesta aproximada exacta a una búsqueda, determinar una respuesta por proximadad aproximada suele ser suficiente. Se han observado varias aplicaciones del vecino más cercano donde una respuesta aproximada es lo suficientemente buena. Gionis en [Gionis et al., 1999] introduce un nuevo método de índice para búsqueda aproximada. Se basa en la idea de *Locality Sensitive Hashing (LSH)*.

Se utiliza l_p^b para denotar el espacio Euclideano $\mathbb{R}^b$ bajo la norma l_p, cuando la longitud del vector es $(x_1, ..., x_b)$ se define como $(|x_1|^p + ... + |x_b|^p)^{1/p}$. Además, $d_p(p, q) = \sqrt[p]{\sum (x_i - q_i)^p}$ representa la distancia entre los puntos p y q en l_p^b. Sea H^b el espacio métrico de hamming de dimension b que representa la base de datos, es decir, el espacio de vectores binarios de longitud d. Para la distancia de Hamming, el número de bits diferentes entre p y q, se usa $d_H(p, q)$. El problema del vecino más cercano se define como sigue:

Definición 1 (Nearest Neighbor Search (NN)). *Dado un conjunto P de n objetos representados como puntos en un espacio l_p^b, preprocesar P para responder eficientemente consultas encontrando el punto en P más cercano a la consulta q.*

La definición se generaliza para el caso de regresar los $k > 1$ puntos. Especificamente, en el caso del k-Nearest Neighbor Search(k-NN), donde queremos regresar los k puntos en la base de datos más cercanos a la consulta. La versión aproximada se define como sigue:

Definición 2 (ϵ - Nearest Neighbor Search (ϵ-NN)). *Dado un conjunto de puntos P en un espacio de norma l_p^b, preprocesar P para regresar eficientemente el punto $p \in P$ para cualquier punto de consulta q, tal que $d(q, p) \leq (1 + \epsilon)d(q, P)$, donde $d(q, P)$ es la distancia de q al punto más cercano en P.*

De nuevo la definición se generaliza para la consulta de $k > 1$ vecinos más cercanos. En k-NN aproximado, se regresan los k puntos $p_1, ...p_k$ tal que las distancias de p_i a la

consulta q es a lo más $(1+\epsilon)$ veces la distancia del k-ésimo vecino a q.

4.1. Locality Sensitive Hashing

Sea C la coordenada mayor de todos los puntos en P. Se puede empotrar P dentro del cubo de Hamming $H^{b'}$ con $b' = Cb$, transformando cada punto $p = (x_1, ..., x_b)$ en un vector binario,

$$v(p) = Unario_c(x_1)...Unario_c(x_b),$$

Es decir, la distancia entre los objetos se preserva. Por lo tanto, se puede utilizar la búsqueda ϵ-NN en el espacio de Hamming $H^{b'}$.

Las funciones hash se definen como sigue. Para un entero l, se eligen l subconjuntos $I_1, ..., I_l$ de $\{1, ..., b^l\}$. Sea $p_{|I}$ la proyección del vector p en el conjunto de coordenadas I. Se calcula $p_{|I}$ seleccionando las posiciones de las coordenadas según I y concatenando los bits en esas posiciones. Se representa $g_j(p) = p_{|I_j}$. Para el preprocesamiento, se almacena cada $p \in P$ en el *bucket* $g_j(p)$, para $j = 1, ..., l$. Como el número total de *buckets* puede ser grande, se reduce el número de buckets utilizando *standard hashing*. Así, se utilizan dos niveles de hash: la función LSH mapea un punto p a un bucket $g_j(p)$, y una función hash estándar mapea el contenido de los *buckets* en una tabla hash de tamaño M.

Para procesar una consulta q, se realiza la búsqueda en todos los índices $g_1(q), ..., g_l(q)$ hasta que por lo menos encontremos $c \cdot l$ objetos donde c es el número de candidatos por índice. Sean $p_1, ...p_t$ los objectos encontrados en la consulta. Para la búsqueda aproximada $k - NN$, se regresan los p_i puntos más cercanos a q; en general, es posible regresar menos puntos si el número de puntos encontrados fue menor a K.

A continuación se presenta como elegir los subconjuntos I_j. Por cada $j \in \{1, ..., l\}$, el conjunto I_j consiste de los k elementos de $1, ..., b'$ tomados uniformemente de manera aleatoria con reemplazo.

El principio detrás del método esta basado en que la probabilidad de colisión de dos puntos p y q esta relacionada con la distancia entre ellos. Específicamente, entre más grande es la distancia entre dos objetos menor es la probabilidad de colisión. Formalmente se tiene $D(\cdot, \cdot)$ una función distancia para elementos de un conjunto S, y para cualquier

$p \in S$ sea $\mathcal{B}(p, r)$ el subconjunto de elementos del conjunto S a una distancia r de p.

Definición 3. Una *familia* $\mathcal{H}$ *de funciones hash de* S *a* U *es llamada* (r_1, r_2, p_1, p_2)*-sensitive para* $D(\cdot, \cdot)$ *si para cualquier* q, $p \in S$,

- si $p \in \mathcal{B}(q, r_1)$ entonces $Pr_{\mathcal{H}}[h(q) = h(p)] \geq p_1$,
- si $p \notin \mathcal{B}(q, r_2)$ entonces $Pr_{\mathcal{H}}[h(q) = h(p)] \leq p_2$.

Para que la familia *locality-sensitive* sea de utilidad, se necesita que se cumpla $p_1 > p_2$ y $r_1 < r_2$.

El Algoritmo 1 muestra el preprocesamiento del índice LSH. El Algoritmo 2 muestra el proceso de consulta. El índice LSH es rápido sin embargo, para que su nivel de respuesta mejore, es necesario utilizar múltiples instancias de LSH ocasionando que el espacio de memoria se incremente considerablemente.

Una solución al uso de multiples instancias, es utilizar estructuras sucintas. La representación del LSH se puede hacer utilizando esta nueva tendencia que se esta haciendo muy popular para las búsquedas de información. El uso de estructuras sucintas permite crear estructuras que ocupen menos espacio respetando la funcionalidad de los algoritmos.

4.2. Representación del índice LSH comprimido

Antes de implementar el índice LSH se necesita crear un espacio de trabajo. En el laboratorio de trabajo se cuenta con una base de datos de 10252 canciones. La idea del proyecto es que dado un segmento de una canción como consulta, se regrese la canción a la que corresponde el segmento consulta.

Para representar el espacio de trabajo, se extrae la huella de audio de cada una de las 10252 canciones utilizando una huella de audio basada en la entropía de la señal presentada por Camarena en [Camarena y Chávez, 2006]. La huella de audio se basa en que la señal de audio se debe medir en perspectiva al oído humano. Se divide la señal en frames (marcos) de 185 milisegundos con un traslape del 75 %. Se le aplica transformada

Algoritmo 1 Preprocesamiento del índice LSH

PREPROCESAMIENTO()

1 *P es el conjunto de puntos*
2 *l es el número de tablas hash*
3 **para** $i = 1$ *hasta* l
4 *Inicializar tabla hash* T_i *generando una función hash aleatoria* $g_i(\cdot)$
5 **para** $i = 1$ *hasta* l
6 **para** $j = 1$ *hasta* n
7 *Almacenar cada punto* p_j *en el bucket* $g_i(p_j)$ *de la tabla hash* T_i
8 **regresar** *Tablas hash* T_i *para* $i = 1, ..., l$

Algoritmo 2 Búsqueda del vecino más cercano aproximado

BÚSQUEDA()

1 *q es el punto de consulta*
2 *K es el número de vecinos más cercanos aproximados*
3 $S \leftarrow \emptyset$
4 **para** $i = 1$ *hasta* l
5 $S \leftarrow S \cup \{$*puntos encontrados en el bucket* $g_i(q)$ *de la tabla* $T_i\}$
6 **regresar** *K o menos vecinos más cercanos aproximados a q.*

de Fourier a cada frame y despúes se determina la entropía por cada banda crítica de Bark. Por cada frame de la señal de audio se obtiene un vector de 24 valores de entropía. En la Figura 4.1 se muestra el vector de 24 bits para un frame en la señal. La Figura 4.2 muestra la representación en frames de una señal de audio completa.

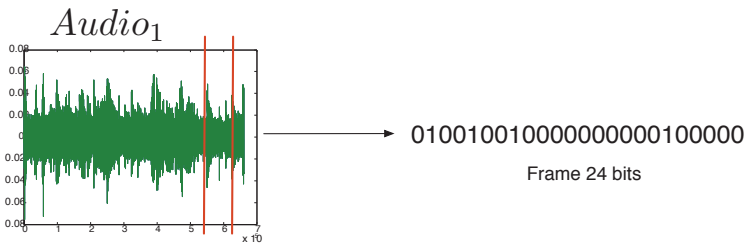


Figura 4.1: Representación de un vector de 24 bits para un frame de la señal de audio.

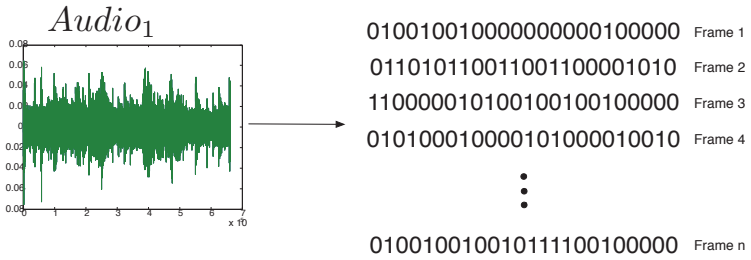


Figura 4.2: Representación de la huella de una señal de audio.

Para construir el espacio de trabajo se define un símbolo mínimo de tamaño 3 bytes que corresponde a los 24 bits de un frame. Se carga cada huella de audio de las diferentes canciones y se concatenan una tras otra formando una sola huella de gran tamaño. Se marca la división entre audio y audio utilizando un bitmap de diferencias que marca con un uno el primer frame de una huella y con tantos ceros como frames contenga la huella. Al inicio de la siguiente huella marca un uno para el primer frame y tantos ceros como frames tenga la huella así sucesivamente.

El siguiente paso es dividir la cadena concatenada de huellas de audio en segmentos de menor tamaño denominados Q-gramas. Estos Q-gramas son un conjunto de frames de 3 bytes. El número de frames tomados para un Q-grama se presenta en los experimentos del capítulo 5. A cada objeto de tamaño Q se le asigna un identificador de objeto denominado Obj_id, que se representa con un número entero de 32 bits y se asigna tal como se van formando los Q-gramas, es decir, el primer Q-grama tiene el Obj_id=1.

En base a estos Q-gramas se calculan distancias con una búsqueda secuencial y se

crea un histograma de distancias. Esto con la finalidad de revisar la forma de la distribución de las distancias y no caer en el fenómeno mencionado anteriormente, la *maldición de la dimensión* presentado por Bellman en [Bellman, 1961].

Una vez dividido el espacio en Q-gramas se obtiene una base de datos de millones de objetos. A continuación se presenta la construcción del índice LSH. El índice se basa en la construcción de una tabla hash donde a cada elemento se le aplica una función hash.

Para crear la función hash se selecciona una familia de funciones hash denominada Familia $\mathcal{H}$, donde cada función hash de la familia lo que hace es seleccionar un bit de manera aleatoria. Por lo tanto, la Familia $\mathcal{H}$ regresa $\mathcal{H}$ bits seleccionados de manera aleatoria pero fijos para todos los objetos pertenecientes a una misma instancia de LSH. La Figura 4.3 muestra $\mathcal{H}$ bits tomados de manera aleatoria pero fija para cada uno de los Q-gramas que forman la base de datos.

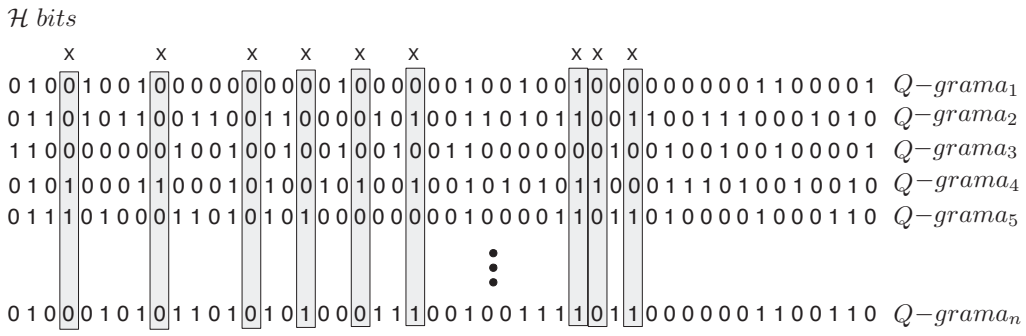


Figura 4.3: Familia $\mathcal{H}$ de funciones hash para los Q-gramas de la base de datos.

Los $\mathcal{H}$ bits tomados de un Q-grama se concatenan formando una cadena de bits de menor tamaño llamada *hash*. En la Figura 4.4 se muestra la concatenación de los bits aleatorios seleccionados de un Q-grama para formar el *hash* del Q-grama.

La tabla hash se conforma de un número de buckets de todas las combinaciones posibles de bits de tamaño $\mathcal{H}$. La tabla tiene una altura de $2^{\mathcal{H}}$. En la Figura 4.5 se muestra la tabla formada para los diferentes *hash*, es decir, para todas las posibles combinaciones utilizando $\mathcal{H}$ bits.

El preprocesamiento del índice se realiza aplicando la Familia $\mathcal{H}$ de funciones

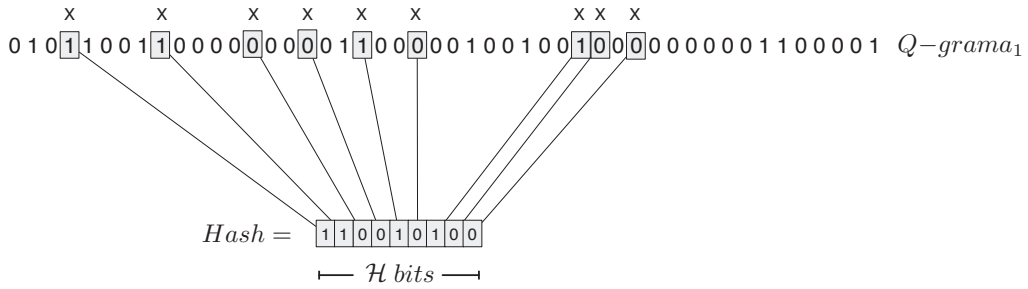


Figura 4.4: Hash para un Q-grama de la base de datos.

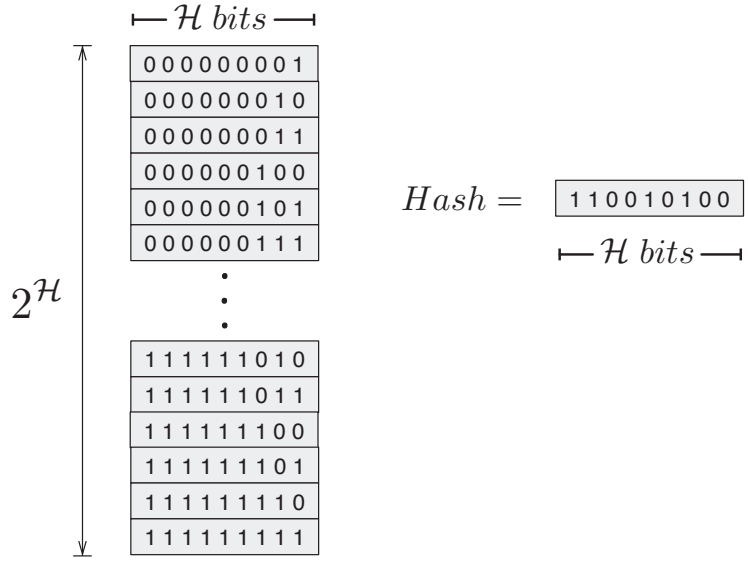


Figura 4.5: Representación de la tabla hash para todas las combinaciones de $\mathcal{H}$ bits.

hash a cada Q-grama. El *hash* que se obtiene apunta a una localidad en la tabla hash denominada bucket. En la representación sin compresión el bucket es un apuntador a una lista que contiene los identificadores de objeto Obj_id correspondientes a ese bucket. Por la manera de obtener la Familia $\mathcal{H}$ existe cierta probabilidad de que objetos similares obtengan el mismo *hash*. Por lo tanto, se encuentran en el mismo bucket. La Figura 4.6 representa los objetos dentro de un bucket en la tabla hash después de aplicar la función hash a los Q-gramas. Las listas son de diferente tamaño.

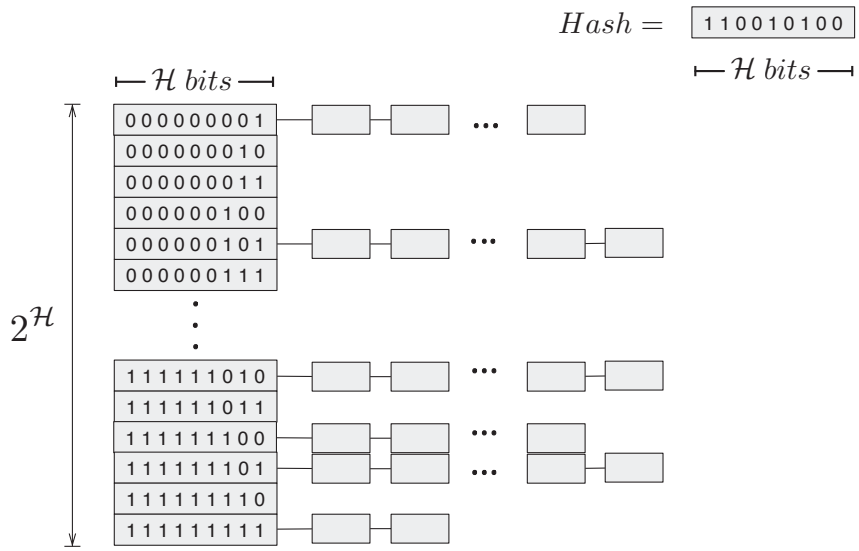


Figura 4.6: Representación del preprocesamiento de la tabla hash.

De ahí, la importancia de la Familia $\mathcal{H}$. Si $\mathcal{H}$ es pequeño el número de Q-gramas que se insertan en cada bucket es mayor. Esto mejora el porcentaje de respuesta ya que al momento de una consulta, con cierta probabilidad el vecino más cercano se encuentra dentro de ese bucket. Sin embargo, el tiempo de consulta se incrementa por que ahora hay que revisar más Q-gramas. Si $\mathcal{H}$ es mayor, el número de Q-gramas por bucket es pequeño, esto mejora el tiempo ya que se revisan menos Q-gramas pero afecta el porcentaje de respuesta. La configuración adecuada depende de la aplicación. Si se requiere porcentaje de respuesta lo mejor es utilizar una $\mathcal{H}$ de menor tamaño. Si se requiere mejorar el tiempo, entonces una Familia $\mathcal{H}$ mayor es la adecuada.

El espacio que ocupa esta representación sin compresión utiliza $2^{\mathcal{H}}$ apuntadores a listas de Q-gramas donde cada apuntador es un número entero de 32 bits. Se necesita el doble de espacio para almacenar las longitudes de las listas. Además se suman los n número de Q-gramas $\times$ 32 bits de un entero que representa los identificadores de objeto. El espacio que utiliza una instancia del LSH es mínima, sin embargo, para mejorar el porcentaje de respuesta se requieren múltiples tablas o instancias de LSH, por lo cual el espacio se incrementa. En la Figura 4.7 se muestra del lado izquierdo una Familia $\mathcal{H}$ menor por lo que

sus buckets tienen listas con mayor número de Q-gramas. En el lado derecho se muestra una Familia $\mathcal{H}$ mayor con buckets que tienen menor número de Q-gramas.

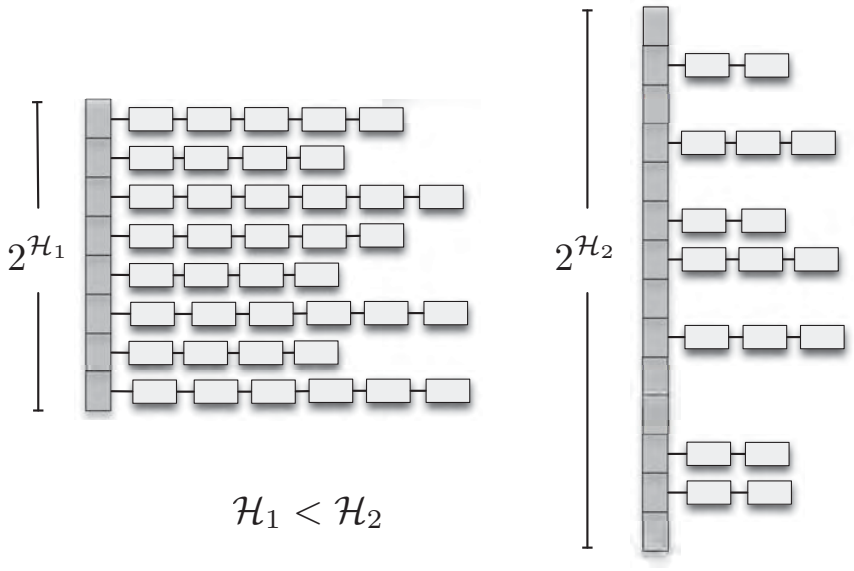


Figura 4.7: Representación de diferentes Familias $\mathcal{H}$.

Como solución al incremento en el espacio que se requiere por utilizar múltiples tablas o instancias del índice LSH, se plantea una nueva representación con la implementación de estructuras sucintas que ocupan espacio cercano a la entropía de orden cero H_0. Para un conjunto $\mathbb{U}$ se crea un bitamap B de tamaño n que es el número de Q-gramas. Se tiene $B[i] = 1$ si $i \in \mathbb{U}$ y $B[i] = 0$ si $i \notin \mathbb{U}$.

La entropía de orden cero se define como $H_0(B) = \frac{m}{n} \log \frac{n}{m} + \frac{n-m}{n} \log \frac{n}{n-m}$ donde m es el número de unos en B y n es el número de elementos.

La construcción del LSH comprimido se realiza haciendo las siguientes modificaciones: se mantiene la tabla hash. Tiene altura $2^{\mathcal{H}}$ que son todas las combinaciones posibles tomados $\mathcal{H}$ bits. Cada uno de los buckets es representado por un entero de 32 bits. La diferencia es que en la versión comprimida los buckets no son apuntadores a listas. Son apuntadores a bitmaps. Para este proyecto se utilizó el bitmap presentado por Okanohara y Sadakane en [Okanohara y Sadakane, 2006]. En la Figura 4.8 se muestra la representación

de las listas de identificadores de objeto *Obj_id* modificadas a la representación en bitmap. En la representación sin compresión los identificadores ocupan 128 bits por cuatro identificadores de objeto de 32 bits cada uno. Para la versión comprimida se tiene el bitmap B de tamaño variable para cada bucket. Tiene una una longitud de 1 al máximo identificador de objeto *Obj_id* dentro del bucket. $B[i] = 1$ si $i \in B$ y $B[i] = 0$ si $i \notin B$. Para la Figura 4.8 los identificadores de objeto se representan exactamente con solo 41 bits.

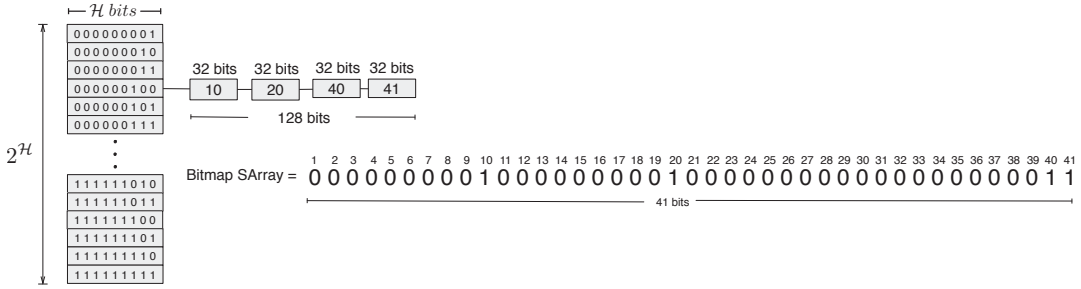


Figura 4.8: Representación del bitmapsarray.

Los bitmaps que se generan en el índice son bitmaps dispersos, es decir, el número de unos es significativamente menor al tamaño del bitmap. Esto debido a que se tienen más de 55 millones de objetos repartidos entre $2^{\mathcal{H}}$ buckets. Por eso se utiliza el bitmap SArray presentado por Okanohara y Sadakane en [Okanohara y Sadakane, 2006] ya que es SArray por Sparse Array.

En la Figura 4.9 se muestra la tabla hash haciendo referencia a bitmaps SArray en cada uno de los buckets en lugar de a listas de enteros de 32 bits.

Al momento de realizar una consulta se le aplica la Famili $\mathcal{H}$ al objeto consulta y se regresa el *hash*. Esto nos apunta a un bucket. Para recuperar el identificador de objeto de los Q-gramas dentro del bucket seleccionado primero se realiza la operación Rank para determinar el número de elementos dentro de ese bucket y después se aplica un Select para regresar la posición de los objetos que corresponde directamente al identificador de objeto Obj_id.

La representación comprimida del LSH mantiene su respuesta en porcentaje de respuesta y su desempeño en tiempo como lo muestran los resultados presentados en el

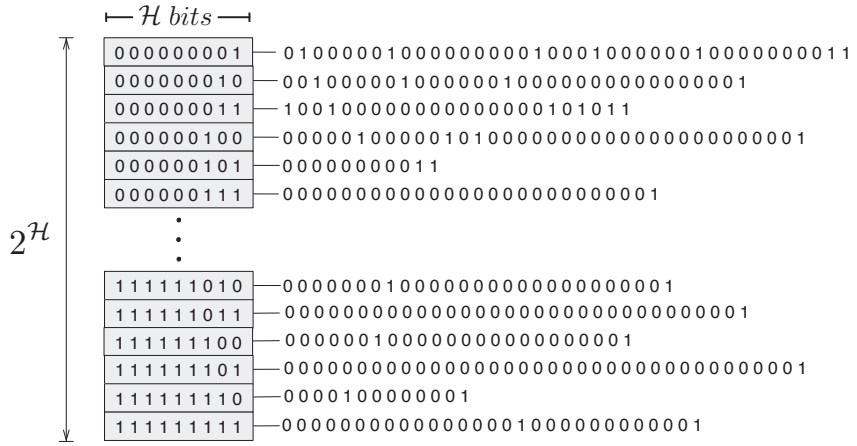


Figura 4.9: Representación del LSH comprimido.

capítulo 5.

Como segunda propuesta se utilizó un índice de secuencias para representar el índice LSH, a está propuesta se le denominó Locality Sensitive Classification (LSC).

4.3. Locality Sensitive Classification (LSC)

La implementación comprimida del índice LSH utilizando el bitmap de Okanohara y Sadakane [Okanohara y Sadakane, 2006] SArray se encuentra cercano a la entropía de orden cero H_0, sin embargo, a cierto tamaño de Familia $\mathcal{H}$ excede el espacio ocupado por la representación sin compresión debido a los overheads que se tiene en la tabla hash. La idea es eliminar esos overheads cambiando la representación de la base de datos. La solución propuesta es la implementación de la estructura llamada Locality Sensitive Classification (LSC). Se modifica la estructura en la representación de la base de datos y se transforma en una secuencia para después utilizar un índice de secuencias como el que presenta Golynski en [Golynski et al., 2006]. Cada uno de los buckets en la tabla hash se convierte en un símbolo de la secuencia. En el LSC se aplica la Familia $\mathcal{H}$ a los Q-gramas y se regresa el *hash*. El identificador de objeto no se inserta en el bucket correspondiente al *hash*, se le asigna un símbolo de secuencia a ese *hash*. En esta representación se tienen $2^{\mathcal{H}}$ símbolos,

$S_1, S_2, ..., S_{2^{\mathcal{H}}}$. La Figura 4.10 muestra como al aplicar la Familia $\mathcal{H}$ el *hash* es representado por un símbolo. El identificador de objeto *Obj_id* no se inserta en el bucket. Se le asigna un símbolo.

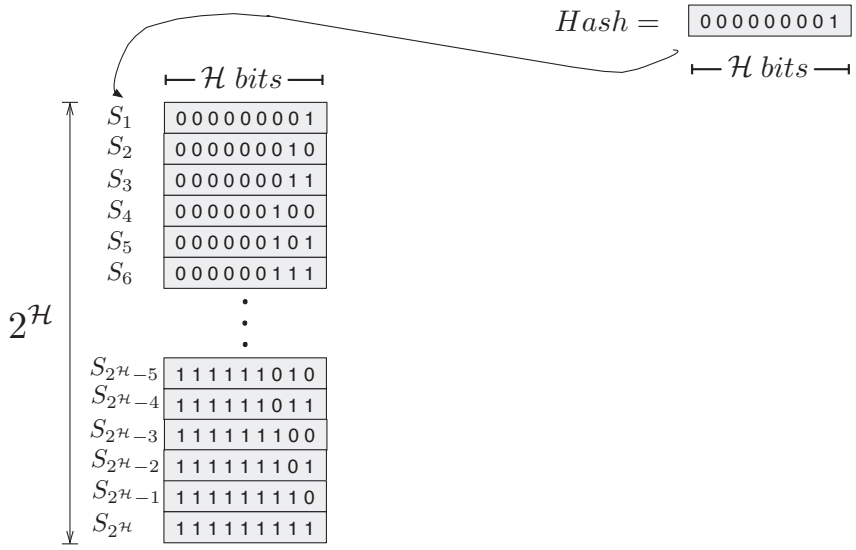


Figura 4.10: Representación de asignación de símbolos en el índice LSC.

La idea general es que para objetos similares cuyo *hash* con cierta probabilidad es el mismo, el símbolo con el que se representa también es el mismo. Objetos similares, son representados con el mismo símbolo. Se procesan todos los objetos en el orden en que se crean y se concatenan los símbolos asignados a los objetos creando una secuencia de símbolos. La Figura 4.11 muestra una secuencia creada concatenando los símbolos asignados a los objetos dependiendo de su *hash*.

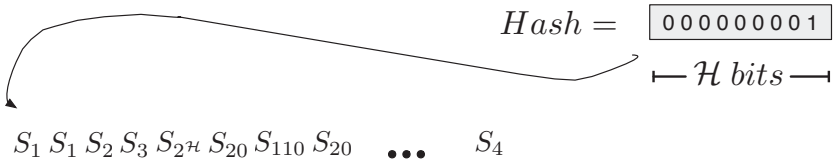


Figura 4.11: Representación de secuencia concatenando los símbolos de los objetos.

Al momento de realizar una consulta se le aplica la Famili $\mathcal{H}$ al objeto consulta y

se regresa el $hash$. Esto nos representa un símbolo. Para recuperar el identificador de objeto de los Q-gramas correspondientes a ese símbolo seleccionado primero se realiza la operación Rank para determinar el número de ocurrencias de ese símbolo y después se aplica un Select para regresar la posición de los símbolos que corresponden directamente al identificador de objeto Obj_id.

Esta representación se encuentra por debajo del espacio ocupado por la representación plana a cualquier tamaño de familia hash $\mathcal{H}$. Los resultados en el espacio se muestran en el capítulo 5.

Capítulo 5

Experimentos y Resultados

Los experimentos para probar la compresión del índice LSH se realizaron utilizando el caso de estudio de recuperación de audio aunque el LSH sirve de manera general. Se eligió el audio por que en el laboratorio se cuenta con una base de datos de 10 252 canciones.

Para modelar el problema, a cada canción se le extrae su huella de audio que es una representación en bits de la canción como se ilustra en la Figura 5.1. Para representar el espacio de trabajo, se extrae la huella de audio de cada una de las 10252 canciones utilizando una huella de audio basada en la entropía de la señal presentada por Camarena en [Camarena y Chávez, 2006]. La huella de audio se basa en que la señal de audio se debe medir en perspectiva al oído humano. Se divide la señal en frames (marcos) de 185 milisegundos con un traslape del 75 %. Se le aplica transformada de Fourier a cada frame y despúes se determina la entropía por cada banda crítica de Bark. Por cada frame de la señal de audio se obtiene un vector de 24 valores de entropía.

Como ejemplo, para una captura o segmento de 15 segundos de una canción, las huellas contienen información de 318 frames $\times$ 24 bits de cada banda crítica de Bark, dando un total de 7, 632 bits por huella. La base de datos esta formada por las huellas de audio de las 10 252 canciones.

La base de datos se representa concatenando todas las huellas de audio. Con un símbolo mínimo de 3 bytes por los 24 bits de cada frame. Se cargan las huellas una tras otra quedando como una sola cadena de gran tamaño como se muestra en la Figura 5.2.

Se crea una respresentación en bitmap que marca la división entre canción y canción. El bitmap pone un uno para el primer frame de una huella y después tantos ceros como frames tiene la huella. Para la segunda huella, pone un uno por el primer frame y después tantos ceros como frames tiene la huella y así sucesivamente para todas las huellas. De esa manera se puede recuperar donde inicia y termina cada huella de audio.

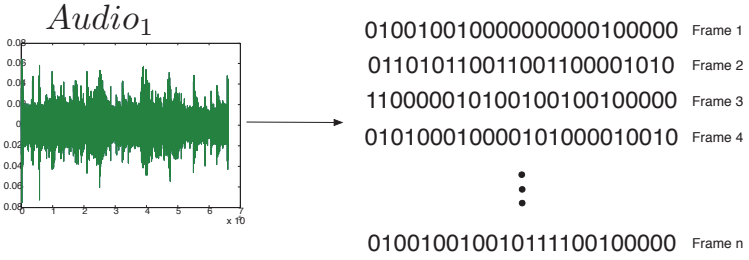


Figura 5.1: Huellas de audio en la base de datos.

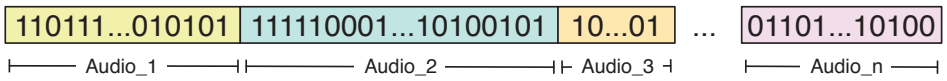


Figura 5.2: Representación de la base de datos concatenada.

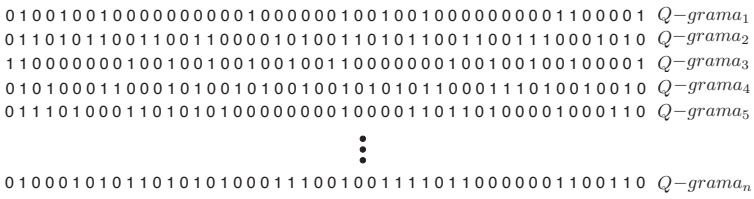


Figura 5.3: Representación del audio dividido en Q-gramas.

Se dividió la huella de audio concatenada en Q-gramas como se mencionó en el capítulo 4. La Figura 5.3 muestra la representación en Q-gramas de la base de datos. Se realizaron diferentes experimentos para determinar el tamaño adecuado de Q-grama. Se realizó una búsqueda secuencial para cada configuración de Q-grama y se calcularon las distancias de Hamming para realizar histogramas con esas distancias. En la Figura 5.4

se presentan histogramas para los más de 55 millones de objetos sobre la frecuencia de distancias que se tienen para los diferentes tamaños de Q-grama. Se realizaron experimentos para segmentos Q de tamaño 45, 60, 90, 120 y 150 bytes. Esto con la finalidad de revisar la forma de la distribución de las distancias. Las Figuras muestran como se va moviendo la media y la distribución se concentra en la media para algunas distribuciones.

Se necesita que la distribución en los histogramas no esté muy concentrada en la media para que no se presente el fenómeno de la *maldición de la dimensión*. Cuando la distribución está concentrada en la media y se presenta la *maldición de la dimensión*, no se pueden distinguir los objetos cercanos o similares de los objetos lejanos o que no se parecen a la consulta.

La elección de la Q adecuada depende también del porcentaje de respuesta. Para revisar el porcentaje de respuesta, se realiza una consulta con una búsqueda secuencial de k-vecinos más cercanos. Después se realiza la misma consulta utilizando el índice para el mismo número de k-vecinos. Se realiza la intersección para ver los resultados en que coinciden. En la Figura 5.5 se presentan los porcentajes de respuesta promedio de 10 corridas para diferentes configuraciones de Q de una consulta de 100 segmentos de 5 segundos. Los segmentos de 5 segundos se modificaron utilizando la herramienta ecasound. Se mezclaron con ruido blanco, de niños jugando, hablando, gritando y riendo, con sonido de fondo de padres hablando y video juegos. El porcentaje de respuesta es mejor para algunas configuraciones pero su histograma de distancias se encuentra concentrado en la media y eso afecta, o el tiempo de consulta es lento. Se debe tener equilibrio entre espacio, tiempo y porcentaje de respuesta. El porcentaje de respuesta varía entre 20 y 50 % para las diferentes configuraciones de Q-grama. Este porcentaje es deficiente por el uso de una sola instancia o tabla de LSH.

La elección del tamaño de Q-grama depende la aplicación. Se debe encontrar un equilibrio entre tiempo, espacio y porcentaje de respuesta. Para este caso de estudió se eligió la representación $Q = 90$, es decir, la Q está compuesta de 30 frames con un tamaño de símbolo de 3 bytes. Donde el tamaño del Q-grama es entonces de 90 bytes. Al final la base de datos consta de una colección de más de 55 millones de Q-gramas.

A continuación se tiene que realizar la configuración adecuada del índice al escoger la Familia $\mathcal{H}$ adecuada para la aplicación.

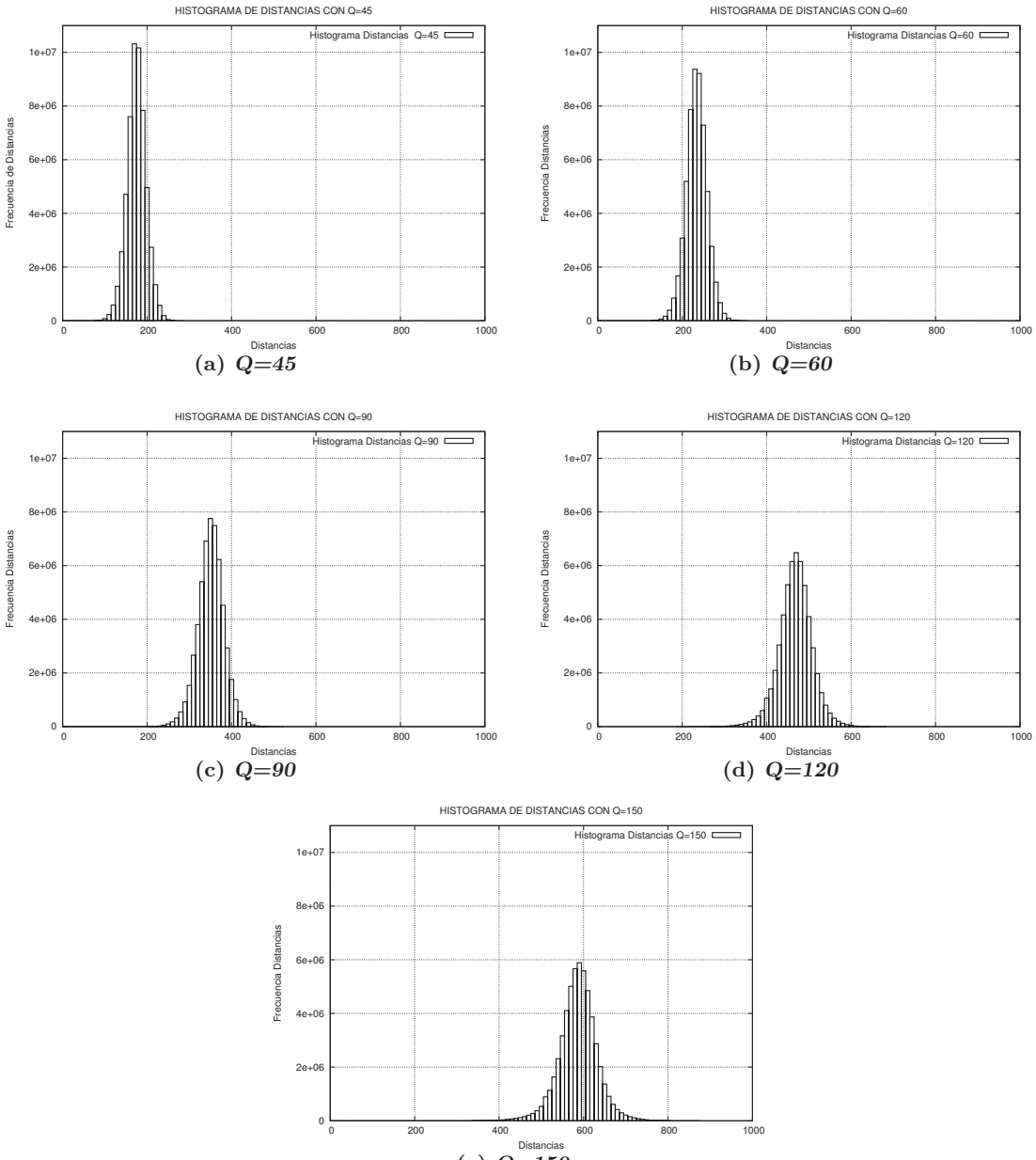
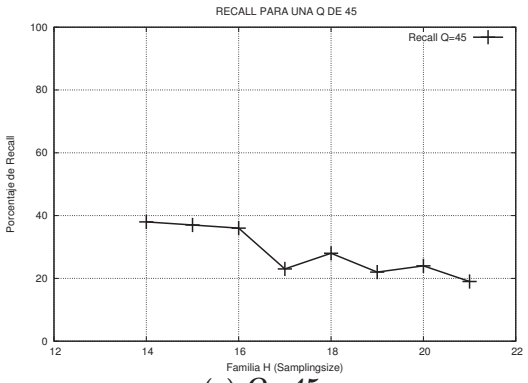
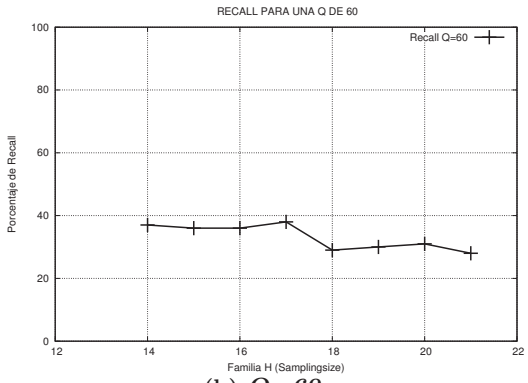


Figura 5.4: Histograma de distancias para las diferentes configuraciones de Q.

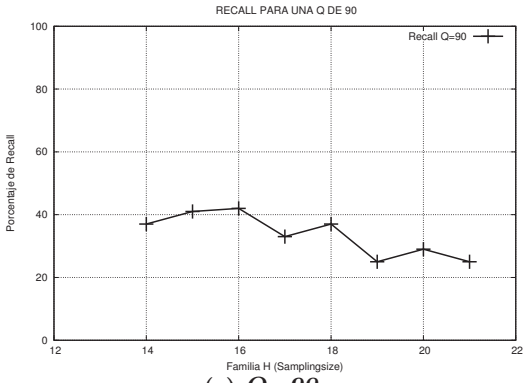
Una vez definida Q, se hicieron pruebas para seleccionar el tamaño adecuado Familia $\mathcal{H}$. Se realizaron experimentos sobre el tamaño de las listas o buckets que es impor-



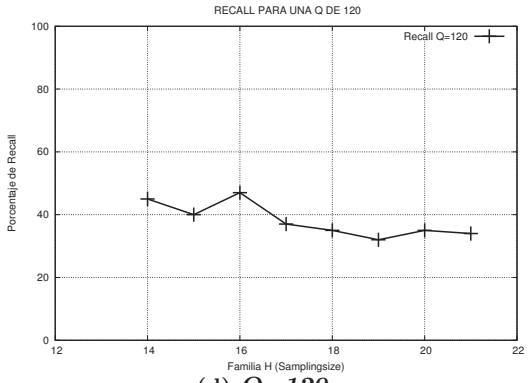
(a) ***Q=45***



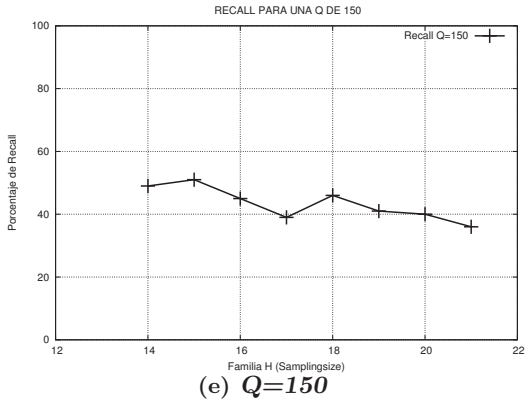
(b) ***Q=60***



(c) ***Q=90***



(d) ***Q=120***



(e) ***Q=150***

Figura 5.5: Recall para la búsqueda de 100 consultas de 5 segundos.

tante por que determina el número de posibles candidatos al momento de una consulta. Se consideran candidatos a consulta aquellos objetos dentro de un mismo bucket. Si la lista

correspondiente a un bucket es larga existen gran cantidad de candidatos similares a una consulta lo que con cierta probabilidad aumenta el porcentaje de respuesta. Sin embargo, al tener listas más largas se determinan distancias con más candidatos lo que hace más lenta la búsqueda. Por otro lado, si las listas son cortas la búsqueda es rápida pero el número de candidatos puede no es suficiente. Es decir, listas cortas mejoran el tiempo de consulta pero afectan el porcentaje de respuesta.

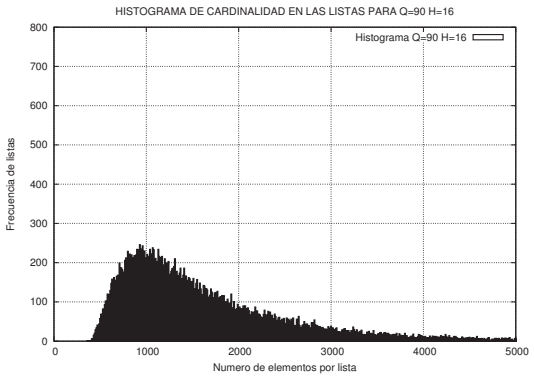
Existen varios factores a considerar para seleccionar los parámetros adecuados. Elegir la configuración con mejor distribución en sus listas no garantiza un mejor porcentaje de respuesta o un mejor tiempo. Debe existir un equilibrio entre porcentaje de respuesta, tiempo y espacio.

En la Figura 5.6 se muestra la cardinalidad de las listas para diferentes tamaños de $\mathcal{H}$. Por ejemplo, para una $\mathcal{H}$ de 16 se tienen que entre 200 a 250 listas tienen 1000 elementos. Para una $\mathcal{H}$ de 17 se tienen 3500 listas con 250 elementos en su lista. La $\mathcal{H}$ de 19 tiene 14000 listas tienen 100 elementos por lista.

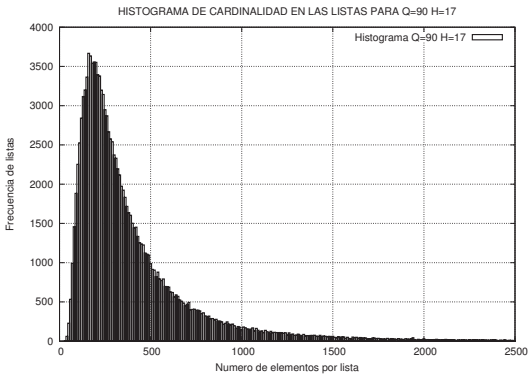
Dependiendo de $\mathcal{H}$ el número de elementos por lista crece o disminuye. Esto es importante tanto en el sentido de porcentaje de respuesta, tiempo de consulta y en el sentido de la compresión. Si las listas contienen gran cantidad de elementos el porcentaje de respuesta aumenta, pero el tiempo de respuesta también aumenta. Si las listas son cortas el tiempo es menor pero la compresión de pocos elementos no reduce el espacio.

En la Tabla 5.1 se presentan los resultados en el tiempo promedio que tarda cada configuración del índice, para Q de tamaño 45, 60, 90, 120 y 150 bytes, en realizar 100 consultas de 5 segundos de las canciones en la base de datos. Dependiendo del tamaño de la familia $\mathcal{H}$ el tiempo varia. Cuando $\mathcal{H}$ es pequeño, la tabla hash tiene menos buckets, pero las listas asociadas a cada bucket tienen un mayor número de candidatos o elementos por lo que la consulta requiere más tiempo. Si $\mathcal{H}$ es grande, la tabla hash tiene más buckets por lo que las listas en cada bucket tienen un menor número de elementos y requiere menos tiempo de búsqueda.

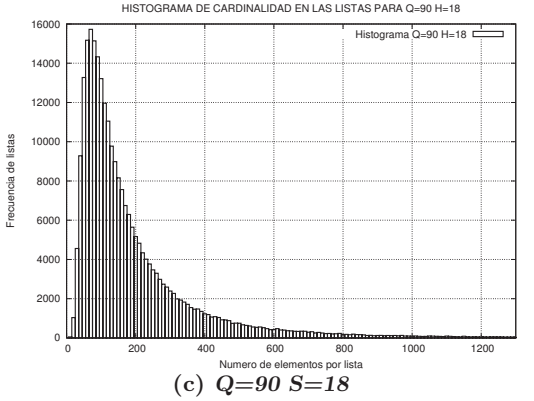
Como se observa en la Figura 5.5 el porcentaje de respuesta utilizando una sola tabla o instancia del LSH no es bueno. Se encuentra entre el 20 y el 50 %. La solución a este problema es incrementar el número de candidatos haciendo uso de múltiples tablas o



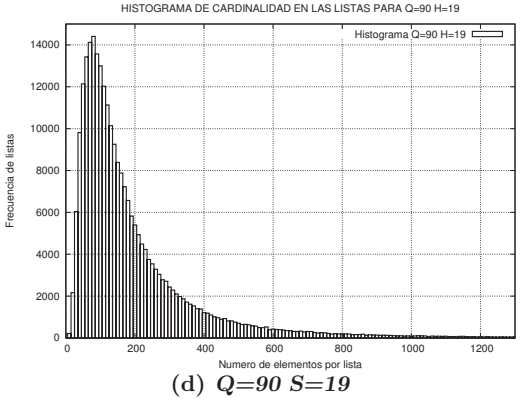
(a) ***Q=90 S=16***



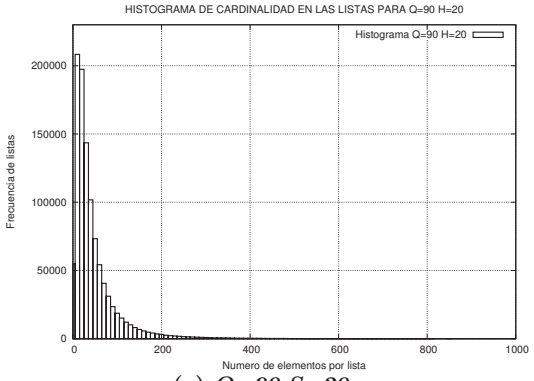
(b) ***Q=90 S=17***



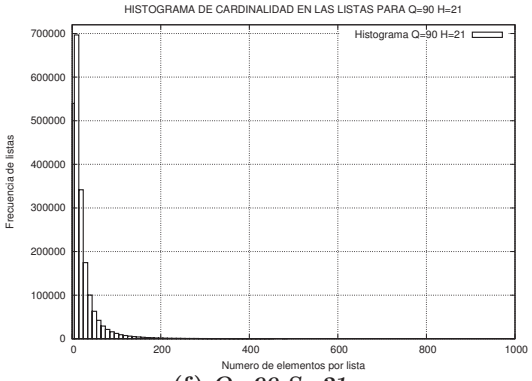
(c) ***Q=90 S=18***



(d) ***Q=90 S=19***



(e) ***Q=90 S=20***



(f) ***Q=90 S=21***

Figura 5.6: Cardinalidad de las listas para una configuración Q=90.

instancias de LSH.

El índice LSH está basado en probabilidad, por lo tanto, en cada instancia de

Tiempo (Segundos)					
Familia $\mathcal{H}$	Q = 45	Q = 60	Q = 90	Q =120	Q = 150
14	0.026	0.017	0.019	0.030	0.042
15	0.013	0.012	0.012	0.027	0.026
16	0.009	0.007	0.006	0.011	0.012
17	0.002	0.004	0.003	0.005	0.009
18	0.003	0.002	0.004	0.002	0.0018
19	0.001	0.002	0.0017	0.005	0.002
20	0.001	0.0018	0.0013	0.0013	0.0011
21	0.0006	0.0007	0.0009	0.0006	0.0014

Tabla 5.1: Tiempo de consulta de 1 tabla para diferentes Q.

LSH los elementos dentro de un bucket llegan a variar de tal forma que si se utilizan varias tablas, se puede mejorar el hecho de que el vecino más cercano aparezca dentro del bucket al momento de la consulta.

En la Figura 5.7 ilustra el efecto de agregar múltiples tablas LSH. El ejemplo muestra en un rectángulo el conjunto de elementos que forman la respuesta exacta a la consulta. Los círculos muestran los elementos de la respuesta exacta que abarca cada instancia LSH. La idea al agregar cada tabla es cubrir toda el área de la respuesta exacta.

El uso de múltiples tablas incrementa el espacio ocupado en memoria principal razón por la cual se plantea la implementación comprimida del índice LSH. El uso de estructuras de sucintas que compriman el espacio utilizado. Para este proyecto de tesis, se utiliza la estructura de bitmap presentada por Okanohara y Sadakane en [Okanohara y Sadakane, 2006] el Sparse array o SArray debido a que las listas en general tienen una distribución de $\frac{n}{2^{\mathcal{H}}}$. Por lo tanto, las listas son dispersas, es decir, que una vez creado el bitmap, el número de unos es significativamente menor n.

El espacio utilizado por una instancia LSH sin compresión, es decir, sin utilizar SArray , es de $2^{\mathcal{H}} \times 32$ bits que ocupa un entero, el doble por que almacena la longitud de las listas. Además los n objetos repartidos en las listas, se tiene $n \times 32$ bits. El espacio ocupado por el LSH con SArray sustituye cada lista o bucket por un bitmap SArray, lo cual reduce el espacio de cada lista con el espacio que ocupa el SArray, es decir, cercano

a la entropia de orden cero H_0. El espacio que ocupa es $\mathcal{H}n + O(n)$. En la Figura 5.8 se muestra el espacio ocupado por una instancia de LSH sin compresión y el espacio de la implementación comprimida utilizando el bitmap SArray.

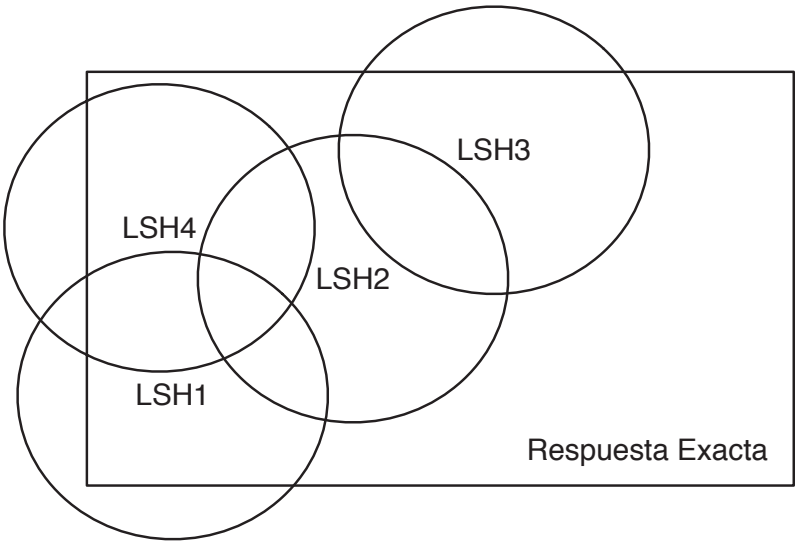


Figura 5.7: Representación de instancias múltiples de LSH.

El tiempo de consulta se incrementa con el uso de múltiples tablas. En la Figura 5.9 se muestra el tiempo promedio por consulta al realizar 100 consultas utilizando los segmentos de 5 segundos para múltiples tablas. El tiempo se encuentra en el orden de los milisegundos.

El uso de múltiples tablas mejora el porcentaje de respuesta significativamente. En la Figura 5.10 se muestran los diferentes porcentajes para diferente número de tablas utilizando la configuración antes seleccionada de Q=90 y diferentes tamaños de Familia $\mathcal{H}$.

Cabe mencionar que estos resultados de respuesta siguen siendo a nivel de Q-grama, no a nivel de canción. El porcentaje de respuesta a nivel de canción es mejor que los de la Figura 5.10 para las 100 consultas de 5 segundos con ruido.

La respuesta a nivel canción se realiza utilizando un esquema de votación. Se tiene una consulta y se divide en Q-gramas utilizando traslape máximo, es decir, recorriendo un

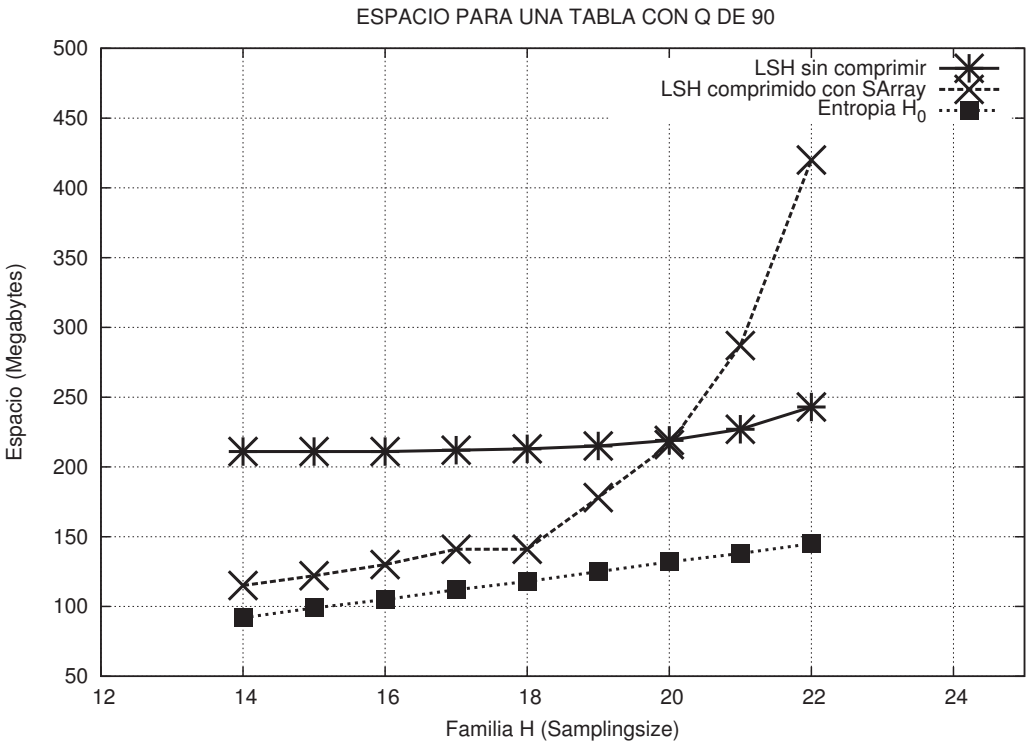


Figura 5.8: Espacio en MB para 1 tabla utilizando diferentes tamaños de familia $\mathcal{H}$.

frame, como se muestra en la Figura 5.11. Se consulta para cada uno de los segmentos y se revisa a que audio pertenece el objeto más cercano como se presenta en la Figura 5.12. Cada vez que se hace una consulta de un Q-grama, se le asigna un voto al identificador de audio que le corresponde. Al final se revisa que canción corresponde al identificador de audio con mayor número de votos como se ilustra en la Figura 5.13.

Como una mejora se utiliza la localidad de las consultas dividiendo en bloques. Se tienen casos aislados donde el identificador de audio incorrecto a la consulta tiene el mayor número de votos, sin embargo, estos votos se encuentran dispersos. En el caso del identificador de audio correcto los votos se mantienen a cierta distancia cercanos. Para aprovechar esto, se toman en cuenta solo los votos que estén cercanos en determinado rango de tiempo y un número mínimo de votos en ese rango. Como se muestra en la Figura 5.14 el $AudioId_1$ cuenta con tres votos igual que el $AudioId_2$, sin embargo, los votos del $AudioId_2$

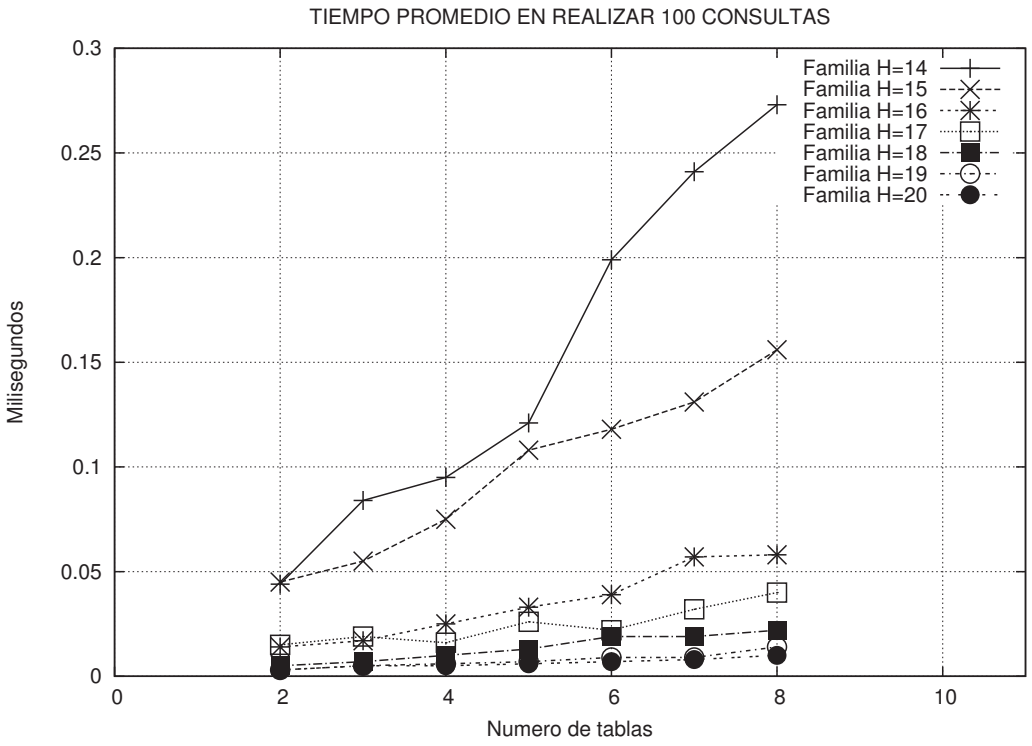


Figura 5.9: Tiempo promedio en segundos al realizar la búsqueda de 100 consultas de 5 segundos para diferentes Familia $\mathcal{H}$.

se encuentran fuera de determinado rango de tiempo por lo que los votos son descartados. Para el AudioId_1 los votos dentro del rango se consideran como 1 voto por lo que para la Figura 5.14 la consulta corresponde al $AudioId_1$. La Figura 5.15 muestra el porcentaje de respuesta a nivel canción donde se utiliza el esquema de votación. El porcentaje se encuentra entre el 80-92 % para diferentes configuraciones del LSH.

La Tabla 5.2 muestra el tiempo promedio al realizar 100 consultas de los segmentos de 5 segundos utilizando el esquema de votación para multiples tablas en la configuración Q=90 y diferente Familia $\mathcal{H}$. Se muestran resultados en tiempo para ocho tablas, sin embargo, para algunas configuraciones no son necesarias las ocho tablas ya que su porcentaje de respuesta se mantiene con menor número de tablas como se muestra en la Figura 5.10.

La implementación comprimida del índice utilizando el bitmap SArray que presenta Okanohara y Sadakane en [Okanohara y Sadakane, 2006] comprime cercano a la entropia

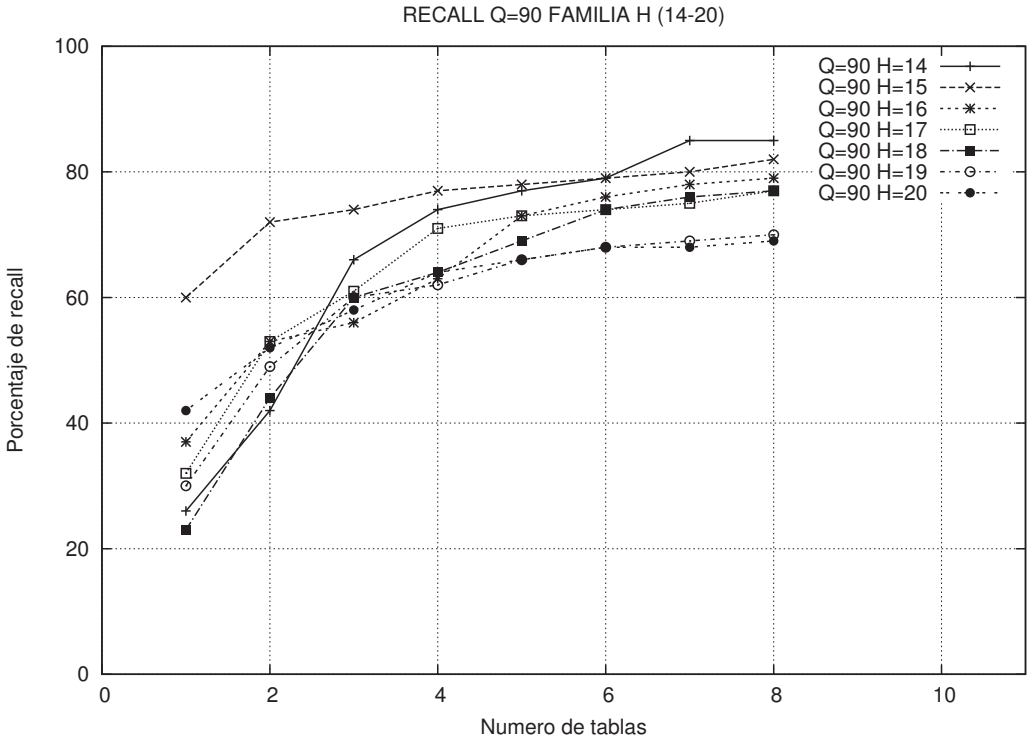


Figura 5.10: Recall utilizando multiples tablas para diferentes configuraciones de LSH.

Figura 5.11: División de consulta en varias consultas de tamaño Q.

de orden cero H_0 y reduce el espacio casi al 40 % del espacio que ocupa el LSH sin compresión, sin embargo, para determinados tamaños de Familia $\mathcal{H}$, el LSH comprimido con SArray ocupa más espacio que la representación sin compresión como se muestra en la Figura 5.8. Para dar solución a esto, se utiliza un segundo enfoque.

La implementación del Locality Sensitive Classification (LSC). El LSC es la repre-

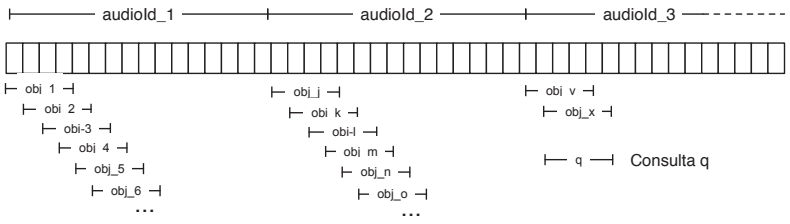


Figura 5.12: Se le asigna un voto a cada consulta de objeto que corresponda al audio.

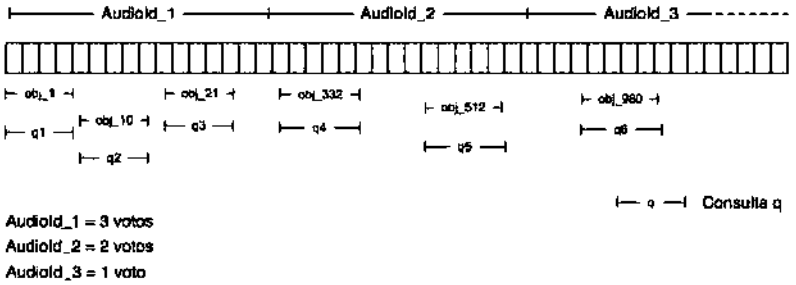


Figura 5.13: Esquema de votación para objetos de tamaño Q que pertenecen a un Identificador de Audio.

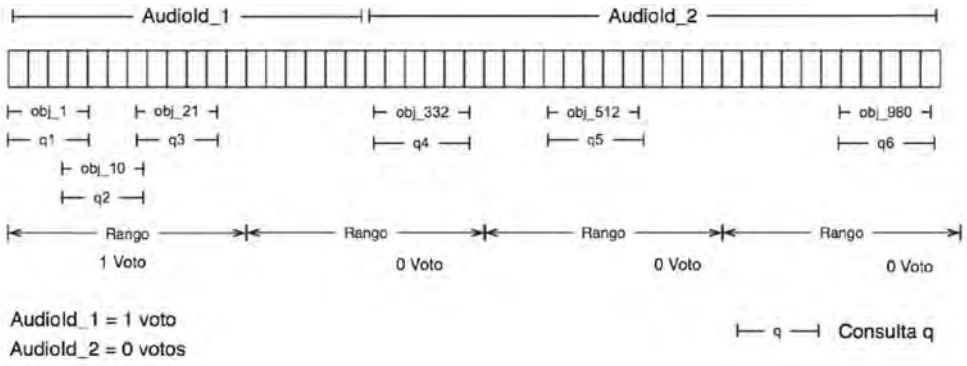


Figura 5.14: Bloques de votación para la localidad de las consultas.

sentación del índice LSH utilizando índices de secuencias. En esta nueva versión se utiliza el índice de secuencias que presenta Golynski et al. en [Golynski et al., 2006]. Este índice comprime en menor razón que la versión comprimida con el bitmap SArray, sin embargo, para el conjunto de Familias $\mathcal{H}$ con que se realizaron los experimentos, el LSC no sobrepasa

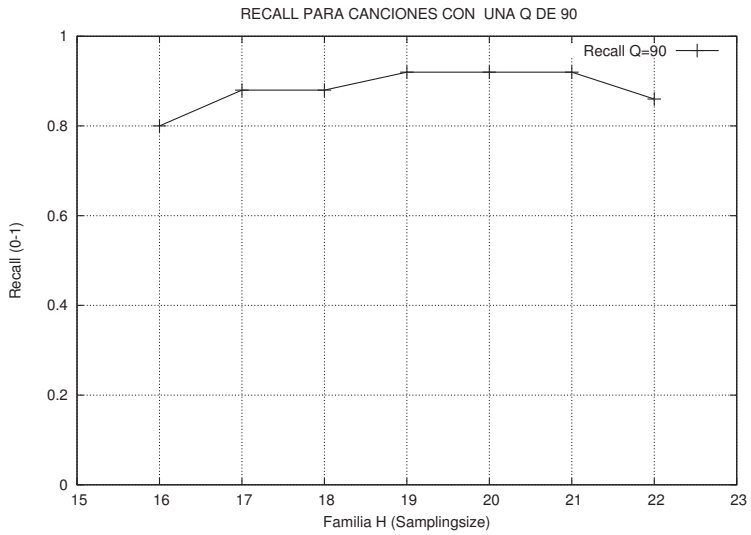


Figura 5.15: Porcentaje de respuesta para 100 consultas de 5 segundos a nivel canción.

8 Tablas para Q = 90	
Familia $\mathcal{H}$	Tiempo (Seg)
16	41.74
17	21.88
18	18.97
19	15.27
20	12.71
21	11.92
22	9.81

Tabla 5.2: Tiempo promedio en segundos al realizar la búsqueda de 100 consultas con multiples tablas.

el espacio utilizado por el LSH sin compresión. El espacio ocupado por la implementación del LSC se muestra en la Figura 5.16.

La compresión del índice LSH comprime cercano a la entropía de orden cero, sin embargo, no soporta tamaños de Familia $\mathcal{H}$ mayores a 20 por que sobrepasa el espacio ocupado por la versión del índice sin compresión. El LSC comprime en menor razón que la compresión del LSH pero soporta tamaños de Familia $\mathcal{H}$ mayores. Las dos propuestas lograron comprimir el índice.

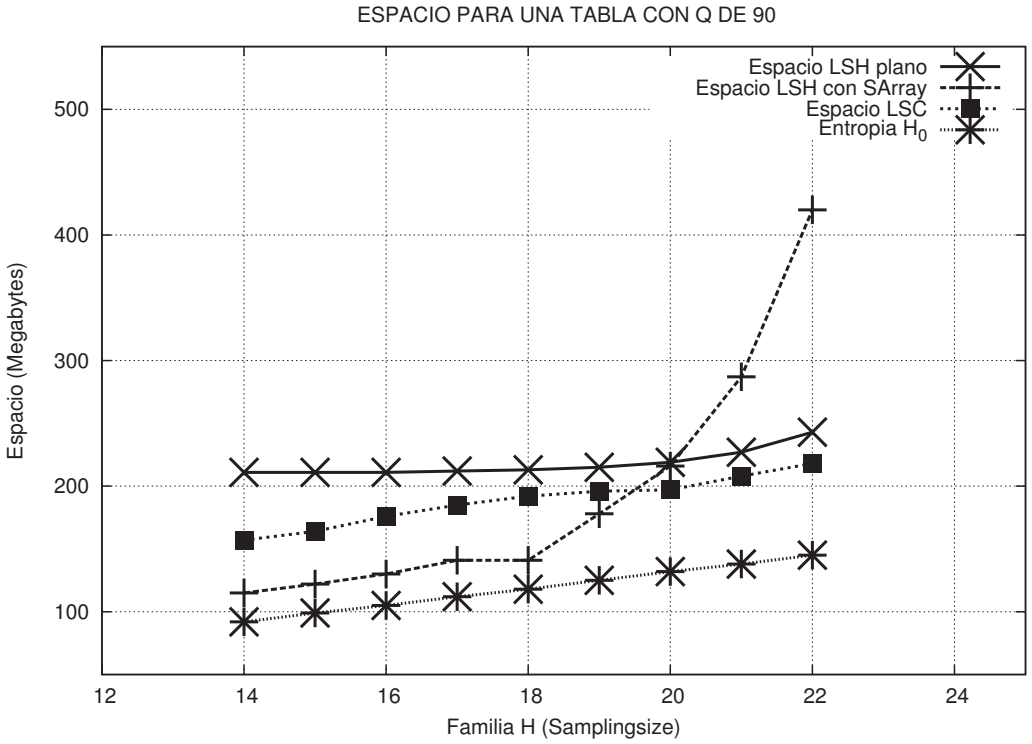


Figura 5.16: Espacio para una tabla del LSC.

Capítulo 6

Conclusiones

6.1. Conclusiones Generales

El estudio sobre índices de búsqueda, estructuras sucintas e índices de secuencias, se encuentra en constante desarrollo. En esta tesis se presentó una nueva representación del índice de búsqueda aproximada Locality Sensitive Hashing (LSH) de [Gionis et al., 1999] que ocupa menos espacio en memoria. Se trabajó con una base de datos de audio de 10, 252 canciones. Se representa la base de datos como segmentos de tamaño Q. Se realizaron experimentos para determinar la configuración adecuada del LSH. Se presentan resultados para determinar la cardinalidad de las listas que componen la tabla hash. La distribución de las distancias en las consultas. Al final la base de datos consta de un conjunto de mas de 55 millones de objetos de tamaño $Q = 90$ bytes. Se realizaron consultas de 100 segmentos de audio de 5 segundos para obtener el tiempo y el porcentaje de respuesta. La respuesta para las 100 consultas se encuentra en el orden de los milisegundos como se muestra en la Tabla 5.1. El porcentaje de respuesta se encuentra entre el 80 y 92 %. Para la reducción del espacio se propusieron dos casos de compresión. El primero, utilizando estructuras sucintas como el bitmap de [Okanohara y Sadakane, 2006] SArray donde las listas de la tabla hash se convierten en bitmaps y se recupera la información realizando operaciones de Rank y Select. En el segundo caso, se modificó la estructura de la base de datos a forma de secuencia. Cada uno de los buckets del hash representa un símbolo. . Los elementos que son similares tienen

alta probabilidad de ser representados por el mismo símbolo formando una secuencia. Se implementó una variación del índice de secuencias de [Golynski et al., 2006] y se recupera la información realizando operaciones de Rank y Select. Se realizaron experimentos para el tiempo, porcentaje de respuesta y el espacio para las nuevas representaciones. Las dos representaciones comprimieron la base de datos y conservaron su porcentaje de respuesta. Además el tiempo de consulta se mantiene en el orden de los milisegundos.

Con la gran cantidad de información disponible es importante seguir desarrollando técnicas para aumentar la velocidad de búsqueda y para reducir el espacio ocupado por los índices en memoria para las bases de datos.

6.2. Conclusiones específicas

1. Con las propuestas realizadas, se logró comprimir la implementación del índice LSH. El bitmap SArray de [Okanohara y Sadakane, 2006] comprimió el espacio cercano a la entropia de orden cero H_0 para determinados tamaños de familia $\mathcal{H}$. Para la operación Rank requiere $O(\log \frac{n}{m}) + O(\log^4 \ m/\log n)$ y para Select $O(\log^4 \ m/\log \ n)$. Lo que permite que se puedan crear múltiples tablas para mejorar el porcentaje de respuesta.

2. El índice de secuencias de [Golynski et al., 2006] comprimió el espacio en menor razón que el bitmap SArray de [Okanohara y Sadakane, 2006], sin embargo, soporta tamaños de familia $\mathcal{H}$ más grandes sin sobrepasar el espacio que ocupa la implementación del LSH plano. Por lo tanto, el tener un tamaño de familia $\mathcal{H}$ mayor nos permite incrementar la velocidad de respuesta reduciendo el espacio en el uso de las múltiples tablas. Las operaciones de Rank y Select se realizan en $O(\log \log \sigma)$ y $O(1)$ respectivamente.

3. La implementación del índice LSH con el bitmap SArray de [Okanohara y Sadakane, 2006] y el índice LSC con el índice de secuencias de [Golynski et al., 2006] tienen un desempeño similar en cuanto a tiempo al realizar la consulta de los 100 segmentos de audio de 5 segundos. Los dos realizan las consultas en el orden de los milisegundos para ciertos tamaños de familia $\mathcal{H}$.

4. El porcentaje de respuesta para las búqueda de las 100 consultas de 5 segundos mo-

dificadas con ecasound se mantiene en las dos propuestas. Se tiene un porcentaje de entre el 80-92 % para el bitmap SArray y para el índice de secuencias LSC.

6.3. Trabajos Futuros

1. Probar el índice en un espacio de trabajo diferente al audio para evaluar su desempeño en espacios con menor distorsión. Crear una tabla comparativa del desempeño que tienen las dos propuestas en cuanto al porcentaje de respuesta y al tiempo. Ejemplos de espacios diferentes son bases de datos de imágenes, documentos, secuencias de ADN, etc.

2. Aún cuando se eligió el bitmap de [Okanohara y Sadakane, 2006] por que su operación Select es rápida. Es recomendable la implementación de diferentes bitmaps para comparar el desempeño en el tiempo y el espacio ocupado.

3. Para el LSC se implementó una variación del índice se secuencias de [Golynski et al., 2006], sin embargo, es importante implementar diferentes índices de secuencias para realizar una tabla comparativa en el porcentaje de reducción del espacio y en su desempeño en el tiempo.

4. Para la variación del índice de secuencias [Golynski et al., 2006] se utilizó una sola instancia del índice LSC. Se plantea utilizar múltiples tablas. Esto se obtiene realizando permutaciones a la secuencia lo cual ocuparía menos espacio en memoria y nos permitiría mejorar el porcentaje de respuesta.

Referencias

[Allamanche et al., 2001] Allamanche E., Herre J., Hellmuth O., y Fröba B. (2001). Content-based identification of audio material using mpeg-7 low level description. En *In Proc. Int. Symposium on Music Information Retrieval (ISMIR.*

[Anderson, 1983] Anderson D.P. (1983). Techniques for reducing pen plotting time. *ACM Trans. Graph.*, 2:197–212. ISSN 0730-0301. doi:http://doi.acm.org/10.1145/357323.357327.

[Arroyuelo et al., 2012] Arroyuelo D., Navarro G., y Sadakane K. (2012). Stronger lempel-ziv based compressed text indexing. *Algorithmica*, 62(1-2):54–101.

[Baeza-Yates et al., 1994] Baeza-Yates R., Cunto W., Manber U., y Wu S. (1994). Proximity matching using fixed-queries trees. En M. Crochemore y D. Gusfield, editores, *Combinatorial Pattern Matching*, tomo 807 de *Lecture Notes in Computer Science*, páginas 198–212. Springer Berlin / Heidelberg. ISBN 978-3-540-58094-2.

[Beckmann et al., 1990] Beckmann N., Kriegel H.P., Schneider R., y Seeger B. (1990). The r*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19:322–331. ISSN 0163-5808. doi:http://doi.acm.org/10.1145/93605.98741.

[Bellman, 1961] Bellman R.E. (1961). *Adaptive control processes - A guided tour*. Princeton University Press, Princeton, New Jersey, U.S.A.

[Bentley, 1975] Bentley J.L. (1975). Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18:509–517. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/361002.361007.

[Bentley y Maurer, 1979] Bentley J.L. y Maurer H.A. (1979). A note on euclidean near neighbor searching in the plane. *Inf. Process. Lett.*, 8(3):133–136.

[Beyer et al., 1999] Beyer K.S., Goldstein J., Ramakrishnan R., y Shaft U. (1999). When is "nearest neighbor" meaningful? En *Proceedings of the 7th International Conference on Database Theory*, ICDT '99, páginas 217–235. Springer-Verlag, London, UK. ISBN 3-540-65452-6.

[Brin, 1995] Brin S. (1995). Near neighbor search in large metric spaces. En *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, páginas 574–584. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 1-55860-379-4.

[Burkhard y Keller, 1973] Burkhard W.A. y Keller R.M. (1973). Some approaches to best-match file searching. *Commun. ACM*, 16:230–236. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/362003.362025.

[Camarena y Chávez, 2006] Camarena A. y Chávez E. (2006). A Robust Entropy-Based Audio-Fingerprint. En *IEEE International Conference on Multimedia and Expo (ICME)*, páginas 1729–1731.

[Chávez et al., 2001a] Chávez E., Marroquín J.L., y Navarro G. (2001a). Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools Appl.*, 14:113–135. ISSN 1380-7501. doi:10.1023/A:1011343115154.

[Chávez et al., 2001b] Chávez E., Navarro G., Baeza-Yates R., y Marroquín J.L. (2001b). Searching in metric spaces. *ACM Comput. Surv.*, 33:273–321. ISSN 0360-0300. doi:http://doi.acm.org/10.1145/502807.502808.

[Ciaccia y Patella, 2002] Ciaccia P. y Patella M. (2002). Searching in metric spaces with user-defined and approximate distances. *ACM Trans. Database Syst.*, 27:398–437. ISSN 0362-5915. doi:http://doi.acm.org/10.1145/582410.582412.

[Ciaccia et al., 1997] Ciaccia P., Patella M., y Zezula P. (1997). M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. En *VLDB '97: Proceedings of the*

23rd International Conference on Very Large Data Bases, páginas 426–435. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 1-55860-470-7.

[Claude y Navarro, 2009] Claude F. y Navarro G. (2009). Practical rank/select queries over arbitrary sequences. En *Proceedings of the 15th International Symposium on String Processing and Information Retrieval*, SPIRE '08, páginas 176–187. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-540-89096-6. doi:http://dx.doi.org/10.1007/978-3-540-89097-3_18.

[Dehne y Noltemeier, 1987] Dehne F. y Noltemeier H. (1987). Voronoi trees and clustering problems. *Information Systems*, 12(2):171 – 175. ISSN 0306-4379. doi:10.1016/0306-4379(87)90041-X.

[Finkel et al., 1974] Finkel R.A., Bentley J.L., Finkel R.A., y Bentley J.L. (1974). Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9. ISSN 0001-5903. doi:10.1007/BF00288933.

[Gabriela y Martin, 1999] Gabriela H. y Martin F. (1999). Cluster-preserving Embedding of Proteins. Informe técnico.

[Gionis et al., 1999] Gionis A., Indyk P., y Motwani R. (1999). Similarity search in high dimensions via hashing. En *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, páginas 518–529. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 1-55860-615-7.

[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. En *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, SODA '06, páginas 368–373. ACM, New York, NY, USA. ISBN 0-89871-605-5. doi:http://doi.acm.org/10.1145/1109557.1109599.

[Guttman, 1984] Guttman A. (1984). R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14:47–57. ISSN 0163-5808. doi:http://doi.acm.org/10.1145/971697.602266.

[Haitsma y Kalker, 2002] Haitsma J. y Kalker T. (2002). A highly robust audio fingerprinting system. En *ISMIR*.

[Hamming, 1950] Hamming R. (1950). Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 26(2):147–160.

[Hinneburg et al., 2000] Hinneburg A., Aggarwal C.C., y Keim D.A. (2000). What is the nearest neighbor in high dimensional spaces? En *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, páginas 506–515. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 1-55860-715-3.

[Hjaltason y Samet, 1995] Hjaltason G. y Samet H. (1995). Ranking in Spatial Databases. En *Symposium on Large Spatial Databases*, páginas 83–95.

[Hunter y Steiglitz, 1979] Hunter G. y Steiglitz K. (1979). Linear transformation of pictures represented by quad trees. *Computer Graphics and Image Processing*, 10(3):289 – 296. ISSN 0146-664X. doi:10.1016/0146-664X(79)90008-X.

[Huttenlocher et al., 1993] Huttenlocher D.P., Klanderman G.A., y Rucklidge W.A. (1993). Comparing Images Using the Hausdorff Distance. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(9):850–863. ISSN 0162-8828. doi:10.1109/34.232073.

[Jacobson, 1988] Jacobson G.J. (1988). *Succinct static data structures*. Tesis Doctoral, Pittsburgh, PA, USA. AAI8918056.

[Jain et al., 1995] Jain R., Kasturi R., y Schunck B.G. (1995). *Machine vision*. McGraw-Hill, Inc., New York, NY, USA. ISBN 0-07-032018-7.

[Kalantari y McDonald, 1983] Kalantari I. y McDonald G. (1983). A data structure and an algorithm for the nearest point problem. *IEEE Trans. Softw. Eng.*, 9:631–634. ISSN 0098-5589. doi:10.1109/TSE.1983.235263.

[Kashyap et al., 1977] Kashyap R.L., Subas S.K.C., y Yao S.B. (1977). Analysis of the multiple-attribute-tree data-base organization. *IEEE Trans. Softw. Eng.*, 3(6):451–467. ISSN 0098-5589. doi:10.1109/TSE.1977.231175.

[Knuth, 1998] Knuth D.E. (1998). *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA. ISBN 0-201-89685-0.

[Lee y Wong, 1980] Lee D.T. y Wong C.K. (1980). Quintary trees: a file structure for multidimensional datbase sytems. *ACM Trans. Database Syst.*, 5:339–353. ISSN 0362-5915. doi:http://doi.acm.org/10.1145/320613.320618.

[Levenshtein, 1966] Levenshtein V.I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. Informe Técnico 8.

[Micó et al., 1994] Micó M.L., Oncina J., y Vidal E. (1994). A new version of the nearest-neighbour approximating and eliminating search algorithm (aesa) with linear preprocessing time and memory requirements. *Pattern Recogn. Lett.*, 15:9–17. ISSN 0167-8655. doi:10.1016/0167-8655(94)90095-7.

[Navarro, 2002] Navarro G. (2002). Searching in metric spaces by spatial approximation. *The VLDB Journal*, 11:28–46. ISSN 1066-8888. doi:http://dx.doi.org/10.1007/s007780200060.

[Okanohara y Sadakane, 2006] Okanohara D. y Sadakane K. (2006). Practical entropy-compressed rank/select dictionary. *CoRR*, abs/cs/0610001.

[Papadias et al., 2003] Papadias D., Zhang J., Mamoulis N., y Tao Y. (2003). Query processing in spatial network databases. En *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '2003, páginas 802–813. VLDB Endowment. ISBN 0-12-722442-4.

[Raman et al., 2007] Raman R., Raman V., y Satti S.R. (2007). Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3. ISSN 1549-6325. doi:http://doi.acm.org/10.1145/1290672.1290680.

[Ruiz, 1986] Ruiz E.V. (1986). An algorithm for finding nearest neighbours in (approximately) constant average time. *Pattern Recogn. Lett.*, 4:145–157. ISSN 0167-8655. doi:10.1016/0167-8655(86)90013-9.

[Ruiz, 1994] Ruiz E.V. (1994). New formulation and improvements of the nearest-neighbour approximating and eliminating search algorithm (aesa). *Pattern Recogn. Lett.*, 15:1–7. ISSN 0167-8655. doi:10.1016/0167-8655(94)90094-9.

[Russell y Norvig, 2003] Russell S.J. y Norvig P. (2003). *Artificial Intelligence: A Modern Approach*. Pearson Education. ISBN 0137903952.

[Samet, 2005] Samet H. (2005). *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 0123694469.

[Sankaranarayanan et al., 2005] Sankaranarayanan J., Alborzi H., y Samet H. (2005). Efficient query processing on spatial networks. En *Proceedings of the 13th annual ACM international workshop on Geographic information systems*, GIS '05, páginas 200–209. ACM, New York, NY, USA. ISBN 1-59593-146-5. doi:10.1145/1097064.1097093.

[Uhlmann, 1991] Uhlmann J.K. (1991). Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179.

[Wang, 2003] Wang A.L. (2003). An Industrial-Strength Audio Search Algorithm. En S. Choudhury y S. Manus, editores, *ISMIR 2003, 4th Symposium Conference on Music Information Retrieval*, páginas 7–13. The International Society for Music Information Retrieval, ISMIR, http://www.ismir.net.

[Wold et al., 1996] Wold E., Blum T., Keislar D., y Wheaton J. (1996). Content-based classification, search, and retrieval of audio. *IEEE MultiMedia*, 3:27–36. ISSN 1070-986X. doi:http://dx.doi.org/10.1109/93.556537.

[Yang, 2002] Yang C. (2002). Efficient acoustic index for music retrieval with various degrees of similarity. En *Proceedings of the tenth ACM international conference on Multimedia*, MULTIMEDIA '02, páginas 584–591. ACM, New York, NY, USA. ISBN 1-58113-620-X. doi:http://doi.acm.org/10.1145/641007.641125.

[Yianilos, 1993] Yianilos P.N. (1993). Data structures and algorithms for nearest neighbor search in general metric spaces. En *Proceedings of the fourth annual ACM-SIAM*

Symposium on Discrete algorithms, SODA '93, páginas 311–321. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. ISBN 0-89871-313-7.