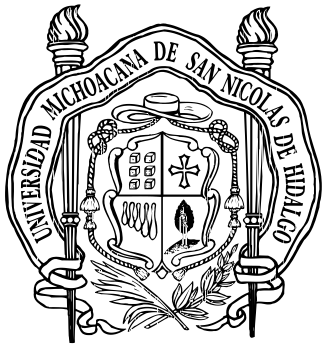


UNIVERSIDAD MICHOACANA DE
SAN NICOLÁS DE HIDALGO

FACULTAD DE INGENIERÍA ELÉCTRICA
DIVISIÓN DE ESTUDIOS DE POSGRADO

Estructuras Sucintas Para Búsquedas En Espacios Métricos: Spaguetti

TESIS

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

PRESENTA:
VICTOR HUGO GÓMEZ LÓPEZ

DIRECTOR DE TESIS:
DR. EDGAR LEONEL CHÁVEZ GONZÁLEZ

AGOSTO 2013

DIVISIÓN POSGRADO

ESTRUCTURAS SUCINTAS PARA BÚSQUEDAS EN ESPACIOS MÉTRICOS: SPAQUETTI

Los Miembros del Jurado de Examen de Grado aprueban
la **Tesis de Maestría en Ciencias en Ingeniería Eléctrica** de ***Víctor Hugo Gómez López***

Dr. José Antonio Camarena Ibarrola
Presidente del Jurado

Dr. Edgar Leonel Chávez González
Director de Tesis

Dr. Félix Caderón Solorio
Vocal

Dr. Mario Graff Guerrero
Vocal

Dr. Eric Sadit Téllez Ávila
Examinador Externo

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

UNIVERSIDAD MICHOACANA DE SAN NICOLÁS DE HIDALGO
JUNIO 2013

Estructuras Sucintas Para Búsquedas En Espacios Métricos: Spaguetti

TESIS

para obtener el grado de
Maestría en Ciencias en Ingeniería Eléctrica

presenta
Victor Hugo Gómez López

asesor
Dr. Edgar Leonel Chávez González

Universidad Michoacana de San Nicolás de Hidalgo

División de Estudios de Posgrado De La Facultad de Ingeniería Eléctrica

Morelia, Michoacán. 2013

Agradecimientos

Le doy gracias a mi madre por apoyarme en todo momento, por los valores que me ha inculcado. A mis hermanos por ser parte importante de mi vida y representar la unidad familiar. A Mamávi por haberme apoyado siempre, por su paciencia y amor incondicional. A mis tíos (Beto, Joel y Eli) por ser un ejemplo de desarrollo profesional a seguir, por llenar mi vida de alegrías y amor cuando más lo he necesitado.

A mi asesor Edgar por haber confiado en mi, por su apoyo en mi trabajo y su capacidad como guía; le agradezco también el haberme facilitado siempre los medios suficientes para llevar a cabo todas las actividades propuestas durante el desarrollo del trabajo que hemos realizado juntos.

A Eric por el apoyo y el ánimo que me brindó. Debo destacar su disponibilidad y paciencia. No cabe duda que su participación ha enriquecido en gran medida este trabajo.

A la IRB: Edgar, Eric, Faby, Jessica, Memo y Paco; porque cada uno con sus valiosas aportaciones hicieron posible este proyecto, con quienes además, ha significado el surgimiento de una amistad.

A todos y cada uno de los Doctores que me dieron clase; por sus enseñanzas, su dedicación y tiempo, por su rectitud como docentes, su visión critica y sus consejos, que ayudaron a mi formación profesional y personal.

A mis compañeros de generación por los momentos compartidos, las enseñanzas, experiencias y su amistad.

Resumen

En el presente trabajo se propone una modificación al algoritmo *Spaghetti* [Chávez González, 1999, Chávez et al., 1999]. La nueva estructura tiene como objetivo reducir el espacio necesario para almacenarse y calcular en menor número de comparaciones la intersección entre los conjuntos, proceso clave en la ejecución del *Spaghetti*. Para lograrlo se reestructura el índice almacenando la permutación que existe entre los elementos de una base de datos con un conjunto que identifica y da orden a estos elementos, esto elimina la dependencia entre las listas de candidatos de cada pivote existente en la estructura original *Spaghetti* y permite cambiar el orden en que se intersectan los conjuntos. Se probaron dos estrategias diferentes:

- SVS, pequeño contra pequeño, intersecta los conjuntos en orden ascendente por su cardinal.
- Aleatorio, calcula la intersección de los conjuntos seleccionando el orden de forma aleatoria, aprovechando la probabilidad conjunta de que un elemento que debe ser eliminado permanezca en la lista de candidatos con el avance iterativo del método.

Cada una de estas estrategias se probó usando dos técnicas diferentes de eliminación de elementos que no se encuentran en la intersección:

- Elemento a Elemento, selecciona un elemento del conjunto de elementos que aún no han sido eliminados, lo busca en los todos los conjuntos asociados a cada pivote antes de proceder al siguiente elemento.

- Por Bloque, revisa todos los elementos en conjunto de candidatos de un pivote, eliminando aquellos que no se encuentren dentro del rango de distancia establecido, antes de continuar su ejecución en el siguiente pivote.

Para lograr reducir el espacio de almacenamiento se implementa una estructura sucinta para almacenar los arreglos de permutaciones, el objetivo es almacenar las permutaciones de un conjunto utilizando tan poco espacio como sea posible pero manteniendo la capacidad de resolver la permutación directa y su inversa.

Abstract

We present a modification to the algorithm *Spaghetti* [Chávez González, 1999, Chávez et al., 1999]. The new structure aims to reduce the space required for storage and compute fewer comparisons in the intersection between the sets, key process in the execution of *Spaghetti*. To achieve this, we restructured the index, storing the permutation between elements of a database with a set that identifies and gives order to these elements, this eliminates the dependence between the candidate lists of each pivot, who exist in the original structure *Spaghetti*, so you can change the order in which the sets intersect. Two different strategies were tested:

- SVS, small vs small, intersects the sets in ascending order by the cardinal.
- Random selection, calculates the intersection by selecting a random order of sets, the joint probability that an element who should be removed remain in the candidates list after few iterations its going to be almost none.

Each of these strategies was tested using two different techniques of removing elements that are not found at the intersection:

- Element to element, select an element of the set of elements that have not been eliminated and looks in all sets associated with each pivot before proceeding to the next item.
- By Block, check all elements in a set of candidates provided by a pivot, eliminating those that are not within the distance range, before continuing its execution on the next pivot.

In order to reduce storage space, implements a succinct structure for storing permutations arrangements, the goal is to store the permutations using as little space as possible while maintaining the ability to solve the direct and inverse permutation.

Índice general

Índice de figuras

Capítulo 1

Introducción

Con el incremento del uso de las tecnologías de información también ha crecido el interés de buscar información en grandes bases de datos (e.g. un conjunto de documentos). Los sistemas de bases de datos relacionales no son capaces de manejar la complejidad de las consultas de este tipo de contenidos, debido a que es ineficiente almacenar en una tabla, haciendo uso de campos. Los sistemas de base de datos en sus inicios sólo permitían modelar bases de datos cuyos objetos se representaban con tipos de datos primitivos (enteros, flotante, carácteres, etc.). Con la necesidad de hacer uso de datos más complejos (colecciones de documentos, imágenes, sonidos. etc.) ha surgido la necesidad de encontrar nuevas técnicas que facilitan la búsqueda de estos, a fin de encontrar soluciones alternativas que permitan gestionar esta información. Por está razón los investigadores en el ramo se han enfocado en la búsqueda de nuevos métodos para realizar las consultas.

1.1. Espacios Métricos

La búsqueda de contenido requiere que estos contenidos sean representados de una forma sencilla, dicha representación es independiente de los datos originales, sin embargo, contiene atributos que abstraen el contenido de los objetos.

Haciendo uso de espacios métricos, la similaridad entre los objetos está modela-

da por una función definida de distancia, los objetos son vistos como cajas negras y la única operación que se permite es el cálculo de distancia con algún otro objeto. Usualmente el cálculo de esta distancia es computacionalmente muy costoso y se considera como la complejidad de realizar una búsqueda al número de cálculos de distancia que sean necesarios para realizarla, a este efecto muchos de los trabajos de investigación se centran en reducir el número de cálculos de distancia y el costo de los cómputos secundarios no es considerado.

Un espacio métrico es una tupla $(\mathbb{U}, d)$ donde $\mathbb{U}$ es el universo de objetos válidos, d denota una medida de "distancia" entre los objetos, $d : \mathbb{X} \times \mathbb{X} \to \mathbb{R}^+$. La base de datos o diccionario esta dada por $\mathbb{S} \subseteq \mathbb{U}$ de tamaño $n = |\mathbb{S}|$, éste el conjunto donde se harán las búsquedas. La función de distancia debe satisfacer las siguientes propiedades $\forall x, y, z \in \mathbb{U}$:

- Positividad, $d(x, y) \geq 0$ y $d(x, y) = 0 \iff x = y$ (positividad estricta)[1].
- Simetría, $d(x, y) = d(y, x)$.
- Desigualdad Triangular, $d(x, y) \leq d(x, z) + d(y, z)$

La propiedad de desigualdad triangular es la que permite descartar elementos sin ser comparados directamente contra la consulta, el resto de las propiedades sólo aseguran una definición consistente de la función de distancia.

Existen básicamente tres tipos de consulta en un espacio métrico:

(a) Regresa los elementos que se encuentran a lo más a distancia r de q, esto es, $\{u \in \mathbb{S} : d(q, u) \leq r\}$. Esta consulta se representa como $(q, r)_d$. éste es el tipo de consulta mas básico, se le conoce como consulta de rango o por proximidad, r es un número que indica el radio de la búsqueda (i.e. la tolerancia), q simboliza la consulta, forma parte del universo y no necesariamente de la base de datos $q \in \mathbb{U}$. La Figura 1.1 ilustra una consulta de rango en un conjunto de puntos en 2 dimensiones.

[1]Si la función de distancia no cumple con la propiedad de positividad estricta el espacio se considera un espacio pseudo-métrico

(b) Regresa el elemento más cercano a q en $\mathbb{S}$, $\{u \in \mathbb{S} : \forall v \in \mathbb{S}, d(q,u) \leq d(q,v)\}$, también podemos dar una distancia r máxima, en éste caso si el elemento más cercano esta más lejos que r el resultado será vacío.

(c) Regresa los k elementos en $\mathbb{S}$ más cercanos a q, $A \subseteq \mathbb{S}$ donde $|A| = k$ y $\forall u \in A, v \in \mathbb{S} - A, d(q,u) \leq d(q,v)$.

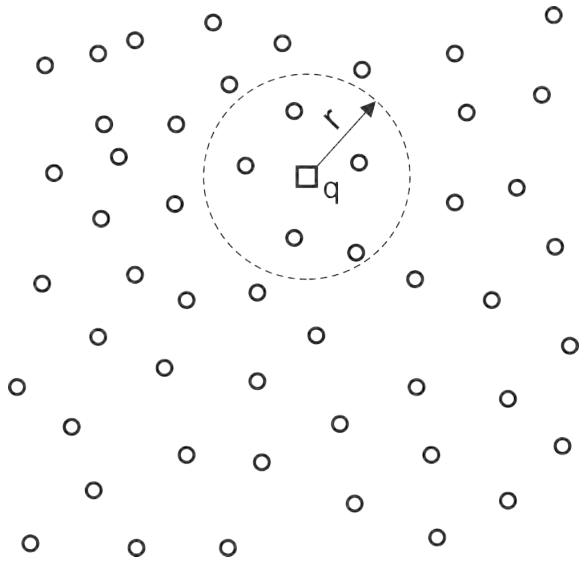


Figura 1.1: Representación de una consulta de rango $(q,r)_d$ en 2 dimensiones

El segundo y tercer tipo son conocidos como consulta por "vecinos cercanos" y podemos determinar la solución de estas con algoritmos que resuelven consultas por proximidad. Como ejemplo, haciendo una consulta de rango con r inicializada en infinito, es reducida hasta que el radio de búsqueda contenga k elementos. Esto se acompaña de técnicas para reducir los elementos tan rápido como sea posible.

Sin un índice todos los objetos de una colección tendrían que ser revisados para saber si cumple con un criterio de selección, los conjuntos de datos son usualmente muy grandes y esta práctica es inaceptable para la mayoría de los casos. Es debido a esto que se requiere de uso de índices, que permitan encontrar un objeto de manera eficiente sin la necesidad de revisar cada uno de ellos.

En los sistemas de bases de datos tradicionales ordenar los datos es la base de una posterior búsqueda eficiente, sin embargo en dimensiones altas este ordenamiento no se puede hacer de una forma tan obvia como se puede hacer con cadenas de texto, números o fechas, por ejemplo.

Los algoritmos que resuelve consultas por proximidad generalmente hacen uso de índices que permiten resolver la consulta sin la necesidad de comparar todos los elementos en la base de datos como lo haría una búsqueda secuencial.

1.1.1. Algoritmos Basados en Pivotes

Un *pivote* es un objeto distinguido dentro de la base de datos. Los pivotes sirven para filtrar objetos en una consulta sin medir directamente la distancia entre los objetos para descartarlos.

Sea $(\mathbb{U}, d)$ un espacio métrico y $\mathbb{S} \subseteq \mathbb{U}$ un conjunto de objetos. Dada una consulta por rango $(q, r)_d$ y un conjunto $\mathbb{P} = \{p_i : p_i \in \mathbb{S}\}$, $|\mathbb{P}| = k$, por desigualdad triangular para cualquier $x \in \mathbb{U}$, se cumple que $d(p_i, x) \leq d(p_i, q) + d(q, x)$, y además que $d(p_i, q) \leq d(p_i, x) + d(x, q)$. De estas ecuaciones obtenemos una cota inferior para $d(q, x) \geq |d(p_i, x) - d(p_i, q)|$. Los objetos de interés son aquellos que satisfacen $d(q, u) \leq r$, por lo que pueden ser excluidos todos los objetos que satisfagan la siguiente ecuación:

$$|d(p_i, u) - d(p_i, q)| > r, \ \textit{para algún } p_i \tag{1.1}$$

Una forma más precisa de describir este proceso, dado $\mathbb{P}$; podemos definir un mapeo del espacio métrico de $\mathbb{S}$ a $\mathbb{R}^k$ como $P : \mathbb{S} \to \mathbb{R}^k$ con $P(x) = (d(x, p_1), d(x, p_2), ..., d(x, p_k))$.

Este mapeo es claramente inyectivo, ya que cada sitio en el espacio métrico será asignado a algún punto en el espacio vectorial $\mathbb{R}^k$. Desafortunadamente el mapeo

no es biyectivo por que dos objetos en el espacio métrico pueden ser asignados al mismo punto en el espacio vectorial. Sin embargo podemos definir una pseudo-distancia o semi-métrica usando la regla de mapeo.

$$D(x,y)) = ||P(x) - P(y)||_\infty = max_j|d(x,p_j) - d(y,p_j)| \tag{1.2}$$

Abusando del lenguaje llamaremos a D una distancia; de cualquier manera dos elementos pueden tener distancia $D = 0$ ya que el mapeo P no es necesariamente biyectivo.

Esta nueva función de distancia es un límite bajo para la función de distancia original d. Se puede comprobar que $||P(x) - P(y)||_\infty \leq d(x,y)$ con x, y cualquier par de elementos de $\mathbb{S}$. Es decir si $D(q,u) \leq r$ entonces $d(q,u) \leq r$.

Haciendo uso de esto podemos definir el siguiente algoritmo:

Consultas de Rango
Preproceso:

- Seleccionar k elementos de la base de datos, etiquetarlos como $\{p_1, p_2, ..., p_k\}$. Seleccionarlos de forma aleatoria es un enfoque válido.
- Calcula y almacena los n vectores
  $\mathbf{u}_i = (d(u_i,p_i), d(u_i,p_2), ..., d(u_i,p_k))$,
  desde $i = 1$ hasta n

Resolver una consulta:
Dados x y r para obtener la lista $\{u_j : d(x_i,u_j) \leq r\}$

- Calcular el vector $\mathbf{x} = (d(x,p_1), d(x,p_2), ..., d(x,p_k))$
- Construir la lista de candidatos para satisfacer la consulta $\{v_l : D(\mathbf{x},\mathbf{u}_l) \leq r\}$

- Ajustar la lista de candidatos "v_l" usando la función de distancia original, para obtener la lista real "u_j".

Este algoritmo es simple y fácil de codificar, para construcciones prácticas es mucho mejor opción que los algoritmos de fuerza bruta. Los resultados experimentales muestran que el número de distancias que deben calcularse significativamente disminuye a medida que k, el número de pivotes, incrementa.

Algoritmos con Funciones de Distancia Discretas

- Burkhard-Keller Tree(BKT), [Burkhard y Keller, 1973]
- Fixed-Queries Tree (FQT), [Baeza-Yates et al., 1994]
- Fixed-Height FQT (FQHT), [Baeza-Yates et al., 1994]
- Fixed Query Array (FQA), [Chavez et al., 2001]

Algoritmos con Funciones de Distancia Continuas

- Vantage Point Tree (VPT), [Uhlmann, 1991]
- Multi Vantage Point Tree (MVPT), [Bozkaya y Ozsoyoglu, 1997]
- Excluded Middle Vantage Point Forest (VPF), [Yianilos, 1999]
- Approximating Eliminating Search Algorithm (AESA), [Vidal Ruiz, 1986]
- Linear AESA (LAESA), [Micó et al., 1994]
- Spaghetti, [Chávez González, 1999, Chávez et al., 1999]

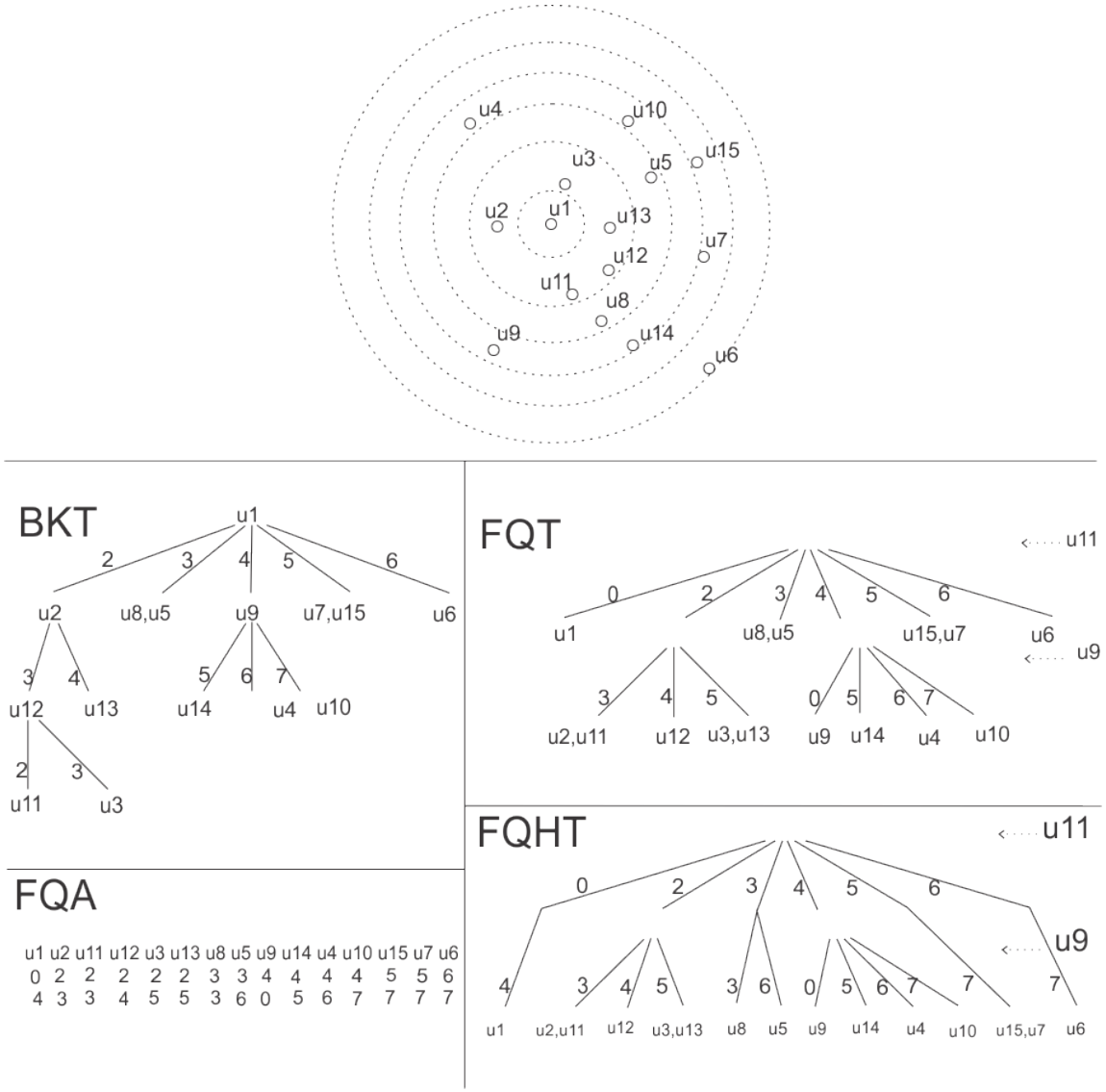


Figura 1.2: Comparativo entre los índices BKT, FQT, FQHT y FQA

1.1.2. Indices de Particiones Compactas

La idea de estos algoritmos es dividir el espacio en zonas, tan compactas como sea posible, es decir, seleccionar una conjunto de objetos $\{c_1, ..., c_k\} \subseteq \mathbb{S}$ tal que cada c_i es la raíz de un subárbol T_i, a estos elementos c_i se les conoce como "centros". El conjunto de centros es usado para hacer particiones en la base de datos de tal forma que cada T_i es compacta en espacio. Cada T_i puede a su vez ser recursiva-

mente particionada en más zonas, esto induce una jerarquía de búsqueda. El índice se compone por los centros, los elementos que pertenecen a cada zona y en algunos casos alguna información adicional que permita descartar zonas completas lo más rápidamente posible.

- Regiones de Voronoi, [Dehne y Noltemeier, 1987]
- Radio de cobertura
- Spatial Aproximation Tree (SAT), [Navarro, 1999]
- List of Clusters (LC), [Chávez y Navarro, 2000]

1.2. Dimensión Intrínseca

Considerando una función de distancia donde $d(x, x) = 0$ y $d(x, y) = 1\ \forall x \neq y$. Con esta función de distancia no obtenemos ninguna información para hacer comparaciones, exceptuando si el elemento que estamos evaluando es o no la consulta. Está claro que no es posible evitar la búsqueda secuencial, sin importar que tan bueno sea nuestro índice.

La dimensión intrínseca de un espacio métrico está definida en términos de su histograma como: $\rho = \frac{\mu}{2\sigma^2}$, donde μ y σ^2 son la media y la varianza del histograma de distancias respectivamente. [Chávez et al., 2001a]

La idea es que cuando la dimensión intrínseca es grande, la media crece y la desviación estándar disminuye. La Figura 1.3 nos da una ilustración intuitiva de porqué el problema es más difícil cuando el histograma está concentrado.

Si tenemos q una consulta aleatoria y un esquema de indexado basado en pivotes, entonces las posibles distancias entre un pivote p y la consulta q están distribuidas de acuerdo al histograma. La regla de eliminación dice que podemos descartar a todos los elementos que cumplan $d(p, u) \notin [d(p, q) - r, d(p, q) + r]$. Las áreas grises en la Figura 1.3 muestran los puntos que no pueden eliminarse. Mientras el histograma más

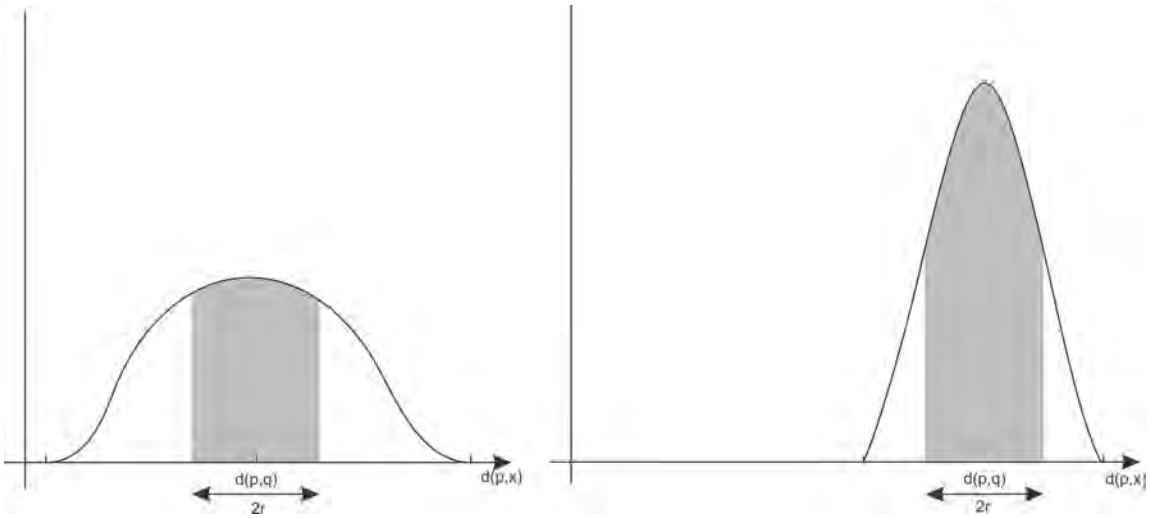


Figura 1.3: Histogramas de distancias de una consulta (q, r)d hacia un elemento p en dimensión baja y alta (izquierda y derecha, respectivamente). Las zonas sombreadas corresponden a los elementos que no podrían ser descartados sin compararse contra la consulta usando un índice y la desigualdad triangular.

se concentra en la media, menos son los puntos que pueden ser descartados usando la información provista por la función de distancia. Este fenómeno es independiente a la naturaleza del espacio métrico y no facilita una manera de cuantificar la dificultad de hacer una búsqueda en dicho espacio métrico.

1.3. Espacios Vectoriales

Los espacios vectoriales son casos particulares de los espacios métricos, si $(\mathbb{U}, d)$ es una tupla de números reales, entonces el par es llamado espacio vectorial de dimensiones finitas, por practicidad, espacios vectoriales. En un espacio vectorial de m-dimensiones los objetos son identificados con m coordenadas de m valores reales $(x_1, ..., x_m)$. Existen diversas funciones de distancia para usar, pero la más usadas son la familia de distancias de Minkowski o distancias L_p, definida como:

$$L_p((x_1, ..., x_m), (y_1, ..., y_2)) = \left(\sum_{i=1}^{m} |x_i - y_i|^p \right)^{\frac{1}{p}} \tag{1.3}$$

1.3.1. Métodos de Acceso Espacial

En los espacios vectoriales existen algoritmos que aprovechan el hecho de que la similaridad se interprete geométricamente. En general, los algoritmos para espacios vectoriales se basan en la geometría del espacio y la información de las coordenadas, algo no disponible en espacios métricos generales donde los algoritmos sólo usan la distancia entre objetos.

Existen algoritmos que aprovechan la interpretación geométrica de la similaridad entre los objetos en los espacios vectoriales, Estos métodos son conocidos como "métodos de acceso espacial" [Longley et al., 1999] (SAM, de sus siglas del ingles Spatial Access Methods). Estas técnicas hacen uso de la información de las coordenadas para clasificar y agrupar los objetos. Algunos ejemplos:

- KD-tree, [Bentley, 1975]
- KDB-tree, [Robinson, 1981]
- R-tree, [Guttman, 1984]

1.4. Aporte de la Tesis

Los índices son estructuras de datos usadas para dar cierto orden a los datos, y así descartarlos o conservarlos al momento de resolver una consulta.

El índice se precalcula y almacena una vez, por lo tanto, no se considera el costo de construcción cuando se resuelve una consulta. Para resolver una consulta se recorre el índice que nos devuelve una "lista de candidatos" que deberán ser comparados directamente con la consulta, estas se conocen como "comparaciones externas", esto con el propósito de descartar aquéllos elementos de la base de datos que no hayan sido excluidos por el índice.

La búsqueda de contenido en bases de datos multimedia requiere representar cada objeto de una forma que pueda buscarse de forma sencilla. Esta representación

es independiente de la información original, pero aún así debe contener atributos que describan al objeto inicial. Estos atributos son comúnmente almacenados como vectores de números reales, conjuntos, cadenas de símbolos, etc. Cuando se trata de miles o millones de objetos y la comparación entre ellos es costosa, hacer una búsqueda exhaustiva resulta impensable, una alternativa son los índices, que permiten localizar rápidamente los elementos relevantes. En otras palabras la búsqueda de estos datos requiere de técnicas como búsqueda métrica, donde una función de distancia métrica es definida entre dos objetos. La complicación se da en dos rubros:

- La complejidad de la representación.
- El tamaño del conjunto de datos.

Esto produce gran consumo memoria y aumenta considerablemente el tiempo de ejecución en la mayoría de los métodos de búsqueda métricos, ya que estos están diseñados para reducir el número de cálculos de distancia necesarios para resolver una consulta. Por otro lado, en general los conjuntos de datos son en altas dimensiones, la búsqueda por proximidad es mucho más costosa sobre este parámetro [Chávez et al., 2001b]. La información que es procesada se ha ido incrementado con el paso de los años, sin embargo el espacio requerido para ser almacenada es lejano a ser un problema dada la disponibilidad de dispositivos de almacenamiento masivo a un bajo costo, la problemática es que el tiempo de acceso a estos dispositivos no ha sido mejorada en mucho tiempo. Por ejemplo, actualmente el acceso a información almacenada en un disco puede llegar a ser un millón de veces más lenta que acceder a información que se encuentra en la memoria principal.

Jacobson introdujo el concepto de *Estructura de Datos Sucinta* [Jacobson, 1989] con el objetivo principal de codificar estructuras de árbol de una forma más compacta. Dichas estructuras han sido utilizadas satisfactoriamente para la codificación de árboles de sufijos y otras variantes relacionadas. Llamamos sucintas a las estructuras de datos que proporcionan ciertas funcionalidades usando un espacio proporcional al que utilizan los datos sin comprimir. De igual manera, llamamos estructura de datos compresa a aquellas estructuras de datos cuyo tamaño sea proporcional a la entropía

de los datos originales.

1.4.1. Objetivo

Este trabajo propone la combinación de técnicas específicas de estructuras de datos compactas con un algoritmo de búsqueda en espacios métricos (*Spaghetti*, [Chávez González, 1999, Chávez et al., 1999]). A fin de generar una nueva solución a partir de estas, que mejore tanto en espacio de almacenamiento como en tiempo de ejecución las soluciones conocidas.
Para lograrlo:

1.4.2. Objetivos Específicos

- Hacer uso de la representación sucinta de las permutaciones [Munro et al., 2003] que permiten mantener las operaciones necesarias para ejecutar el espagueti, logramos reducir el espacio necesario para almacenar el índice.

- Modificar la estructura original del espagueti, posibilitar la ejecución de estrategias para calcular una intersección, operación clave en la ejecución del *Spaghetti*, estas estrategias permiten disminuir el tiempo de ejecución.

  De forma más específica en este trabajo nos centramos en dos alternativas para encontrar dicha intersección:

  - *Pequeño contra pequeño* (SvS) [Culpepper y Listair, 2010], El método SVS es simple y efectivo, se identifica el conjunto más pequeño y este es intersectado con cada uno de los conjuntos restantes, ordenados de forma ascendente por su cardinal.

  - Selección Aleatoria, haciendo una selección aleatoria del siguiente pivote a intersectarse, reduce la probabilidad conjunta de que un elemento per-

manezca.

1.5. Organización de la Tesis

El presente trabajo ha sido dividido en capítulos, de la siguiente manera:

- En éste primer capítulo se presentó el problema de buscar los elementos de un conjunto que es cercano a una consulta bajo un criterio de similaridad, se hace una reseña del modelo formal sobre el cual estamos basando este trabajo, así como otros conceptos relacionados.

  - **Espacios Métricos.** Hacemos una revisión de los algoritmos para la búsqueda en espacios métricos. Se divide en dos familias: basadas en pivotes y basadas en particiones compactas. En cada una se presentan las bases formales, y las soluciones que aprovechan esta técnica.

  - **Dimensión Intrínseca.** El Capítulo finaliza con la definición y análisis de los efectos de la dimensión intrínseca en los algoritmos de búsqueda métricos.

  - **Espacios Vectoriales.** Mostramos otra alternativa para hacer búsquedas, los espacios vectoriales aprovechan la similaridad geométrica y la información de las coordenadas. Se revisan distintos algoritmos existentes.

  - **Aporte de la Tesis.** A grandes rasgos describimos el objetivo del presente trabajo.

- **Spaghetti**
  Se presenta una estructura basada en pivotes con capacidad para hacer inserciones y borrados tanto de elementos en la base de datos como de pivotes.

Esta característica es de utilidad en aplicaciones donde la base de datos no es estacionaria, y los pivotes deben ser cambiados periódicamente.

- **Conceptos de Interés.** Algunos conceptos necesarios para la elaboración de este trabajo son ilustrados en este Capítulo.

  - **Estructuras Sucintas.** Las estructuras sucintas son diseñadas para operar en RAM por lo que deben ser compactas, además deben ser capaces de resolver operaciones. Esta sección describe como se define una estructura sucinta, como funcionan y algunas operaciones útiles.

  - **Permutaciones.** Las permutaciones son un tema importante para las ciencias de la computación, es por esto que han sido foco de extenso estudio. Revisamos en esta sección, que son las permutaciones, propiedades de las permutaciones y operaciones que pueden realizarse.

  - **Permutaciones Sucintas.** Se presenta una solución propuesta en [Munro et al., 2003]para manejar las permutaciones en una estructura compacta.

- **Nueva representación del Spaghetti.** Describiremos una nueva manera de construir un Spagheti de forma compacta que no sólo permita almacenarlo en menor cantidad de espacio, sino que permita reducir el tiempo que toma calcular la intersección necesaria en la ejecución del método.

  - **Cambiando la estructura del Spaghetti.** Se detalla el proceso de construcción del nuevo espagueti, la idea general es reestructurarlo y aplicar la estructura compacta de las permutaciones descrita en el capítulo anterior.

  - **Cálculo de la Intersección.** Definimos dos estrategias distintas para establecer el orden de ejecución cuando se calcula la intersección.

- **Experimentación.** Se presentan una serie de experimentos con la intención de evaluar el desempeño del algoritmo tanto en tiempo de ejecución, como en espacio de almacenamiento, Variando sus parametros (número de pivotes y factor de compresión) y con diversas bases de datos, de distintos tamaños y dimensionalidades.

Capítulo 2

Spaghetti

La estructura de datos *Spaghetti* [Chávez González, 1999, Chávez et al., 1999] resuelve consultas por similaridad en espacios métricos. Sólo tienen un parámetro (el número de pivotes) esto lo hace adecuado para aquellos que necesitan una caja negra para agregar a sus aplicaciones. Incrementando el número de pivotes disminuye la cantidad de cálculos de distancias, comúnmente una medida de complejidad.

El algoritmo usa directamente las distancias a los pivotes. Es decir, este algoritmo no discretiza los valores de las distancias como lo hacen otras técnicas de indexado.

2.1. Cálculo de la Intersección

El conjunto de puntos en la lista de candidatos (i.e. $\{x : D(q, x) \leq r\}$)puede ser descrito como como los puntos que caen dentro de una región hipercúbica centrada en el punto $(d(p_1, q), ..., d(p_k, q))$ de tamaño k en el espacio vectorial $(\mathbb{R}, ||\cdot||_\infty)$. Con un enfoque secuencial, cada coordenada de un punto debe ser comparada con los límites del cubo para revisar si es que cae dentro de la región objetivo. Se puede observar que el procedimiento descrito es equivalente a encontrar la intersección de los k conjuntos, cada uno obtenido como un conjunto de puntos que caen dentro de los limites propios de la coordenada correspondiente.

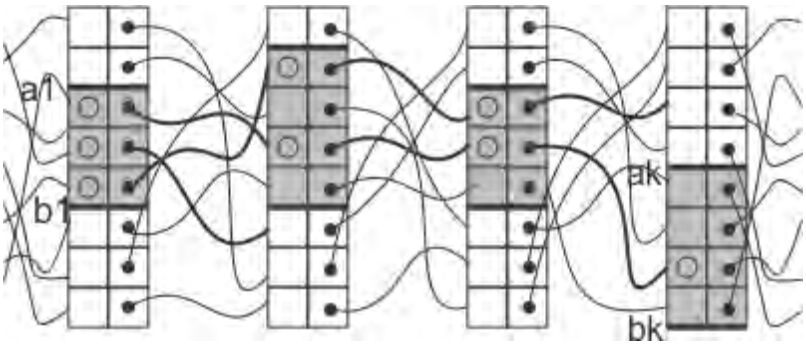


Figura 2.1: La estructura del *Spaghetti*. La probabilidad de recorrer todo el *Spaghetti* es el producto del tamaño de los intervalos dividido por el número de elementos en la base de datos.

En otras palabras para resolver una consulta $(q, r)_d$, encontramos k intervalos $[a_1, b_1], ..., [a_k, b_k]$ ($a_i = d(p_i, q) - r$ y $b_i = d(p_i, q) + r$), obtenemos los índices de los intervalos correspondientes a cada arreglo, $[I_1, J_1]...[I_k, J_k]$. Por último, es necesario seguir los elemento por los punteros para saber si caen dentro del intervalo o no, si cae dentro del intervalo en todos los arreglos entonces se encuentra en la intersección. Como se muestra en la Figura 2.1

El objetivo del algoritmo es encontrar esta lista de candidatos, calcular la intersección con menor esfuerzo del que toma calcularla haciendo por fuerza bruta. El algoritmo se describe brevemente en el *Algoritmo 1*:

2.2. Costo de Ejecución

2.2.1. Búsqueda Infructuosa

El costo de ejecutar el *Spaghetti* es lineal al número de pivotes ($O(km)$, con m el número de elementos dentro del subconjunto del primer pivote seleccionado), sin embargo para una consulta vacía podemos considerar que es menor ya que habrá una interrupción en la secuencia de ejecución en cada pivote. En promedio la mitad de la"secuencia de *Spaghetti*" debería ser revisada antes de que la ejecución termine, esto es lineal a k pero con una constante más pequeña.

Algoritmo 1 Spaghetti

Preproceso

- Para cada pivote calcular y guardar las distancias de cada elemento en la base de datos.
- Ordenar cada arreglo guardando los punteros a la posición de los elementos en el arreglo precedente (como se ilustra en la Figura 2.1)

Consulta

Dados k intervalos, define k conjuntos, $[a_1, b_1], ..., [a_k, b_k]$ (con $a_i = d(p_i - q) - ryb_i = d(p_i) + r$)

- Obtener los intervalos del índice $[I_1, J_1], ..., [I_k, J_k]$ correspondientes a cada conjuntos.
- Sigue cada elemento a través de los punteros para encontrar si cae dentro de todos los intervalos del índice.
- Si un punto cae en todos los intervalos forma parte de la intersección.

Por simplicidad consideremos que cada intervalo tiene en cada arreglo m elementos, de ahí que la fracción de elementos eliminados por cada intervalo sean $\varepsilon = \frac{m-s}{n}$ siendo n el tamaño de la base de datos y s el tamaño de la intersección, para el caso de una búsqueda infructuosa $s = 0$ por lo tanto tomemos $\varepsilon = \frac{m}{n}$. El peor de los casos se tiene que revisar cada elemento en todos los arreglos su complejidad sigue siendo $O(km)$.

Para encontrar el tiempo promedio hagamos c una variable aleatoria que describa el costo de ejecutar el *Spaghetti* para un elemento individual, c tomará valores discretos en el intervalo $[1, k-1]$. Si $k = 2$ para cada elemento en el primer intervalo pagamos 1 para ejecutar el *Spaghetti*. Si $k = 3$ entonces pagamos 1 con la probabilidad de encontrarlo en el segundo arreglo (e.g. $P(1) = 1 - \varepsilon$); pagamos 2 si lo encontráramos en el segundo (e.g. P(2)=ε^2). Para $k > 4$ el razonamiento es similar.

La fórmula para encontrar la probabilidad con una k arbitraria es:

$$P(c) = \left\{ \begin{array}{cc} \varepsilon^{c-1}(1-\varepsilon) & \text{si } c < k-1 \\ \varepsilon^{k-1} & \text{si } c = k-1 \end{array} \right\} \tag{2.1}$$

El valores esperado de c es:

$$E(c) = \sum_{i=1}^{k-1} cP(c) = c\sum_{i=1}^{k-1} \varepsilon^{i-1} = c\sum_{i=0}^{k-2} \varepsilon^{i} \tag{2.2}$$

De ahí que el costo promedio para una consulta vacía sea:

$$E(c) = \sum_{i=0}^{k-2} \varepsilon^{i} = \frac{1-\varepsilon^{k-1}}{1-\varepsilon} \leq k \tag{2.3}$$

note que para todo caso $m \leq n$.

2.2.2. Búsqueda Exitosa

Para una consulta exitosa, tomemos s como el tamaño de la intersección, el costo mínimo para encontrar la intersección es ks; ya que la ejecución no se puede terminar hasta comprobar que los s elementos se encuentren dentro de los k intervalos.

Tendremos que agregar el costo ks al costo ya calculado en caso de una consulta infructuosa, haciendo $\varepsilon = \frac{m-s}{n}$; Tomando en cuenta que los objetos descartados por los pivotes serán $m - s$. En todo caso $s \leq m$.

$$E(c) = \frac{1-(\frac{m-s}{n})^{k-1}}{1-(\frac{m-s}{n})} + ks \tag{2.4}$$

2.3. Capacidades Dinámicas

Nótese que en el *Spaghetti* deben encontrarse los intervalos en cada arreglo para resolver la consulta. Como los arreglos están ordenados, es posible realizar una búsqueda binaria para este objetivo; la búsqueda binaria deberá ejecutarse k ocasio-

nes, una para cada pivote. Sin embargo, si se desea incluir capacidades dinámicas, no es la mejor opción. Una lista ligada puede ser usada, para permitir la inserción y borrado.

2.3.1. Elementos de la base de datos

Borrado. Ya que la posición de los elementos puede cambiar en cada arreglo, es necesario localizarlos en cada uno de ellos; antes del proceso de borrado. El procedimiento es un poco complicado por que en general se requiere eliminar celdas en diferentes columnas (ver Figura 2.2).

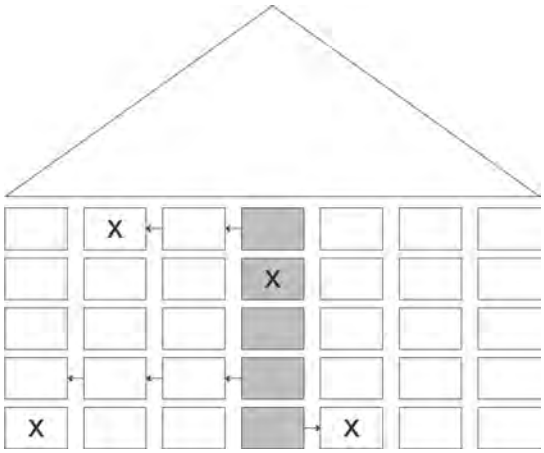


Figura 2.2: Esquema de Inserción/Borrado para la estructura del *Spaghetti*

Se selecciona una columna arbitraria (una selección justa es tomar una de las columnas cercanas a la mitad) y los elementos son cambiados a la derecha/izquierda para vaciar las celdas que serán eliminadas. En la Figura 2.2 la columna sombreada es seleccionada, y las celdas etiquetas con una 'X' son espacios vacíos.

Inserción. La inserción de un nuevo elemento en la base de datos es aproximadamente el inverso a hacer un borrado. Seleccionamos una nueva columna en el medio, localizamos la celda apropiada en cada columna y movemos en alguna dirección para

crear una celda vacía.

2.3.2. Pivotes

En esta estructura de árbol, como en el FHQT, es trivial insertar nuevos pivotes, (i.e. incrementar la altura del árbol) pero no es posible eliminar un pivote una vez que ha sido utilizado para el índice. En el *Spaghetti* por otra parte, la inserción y borrado pueden ser implementados de forma trivial sin necesidad de reconstruir el índice.

Primero debemos notar que eliminar un arreglo implica redireccionar los punteros, como se hace al eliminar un elemento en una lista ligada y así la estructura sigue funcionando. La inserción de igual forma es simplemente realizar un ordenamiento, el nuevo pivote (arreglo) es siempre insertado hasta el punto más a la izquierda del *Spaghetti*. el arreglo es inicializado con dos valores, la distancia al pivote y el puntero a la posición del elemento en el siguiente arreglo del *Spaghetti*; entonces el arreglo es ordenado usando como criterio las distancias al pivote.

2.4. Análisis

La construcción del *Spaghetti* tiene como costo kn cómputos de distancias. Las operaciones restantes (sin cálculos de distancia) están limitadas por $O(kn)$, ya que el costo de insertar un punto es $O(k \log_2 n)$ (encontrar la celda apropiada) más $O(kn)$ operaciones de intercambio. Note que cualquier algoritmo debe ejecutar $O(k)$ comparaciones de objetos en una inserción. De cualquier manera, existe un intercambio por hacer la estructura más dinámica, se incrementa el uso de espacio (como ocurre en las FQT).

2.5. Búsqueda Del Vecino Más Cercano

La búsqueda del vecino cercano puede ser construida a partir de búsquedas de rango utilizando la regla de "branch and bound".

Un buen algoritmo de indexado para búsquedas por rango pude ser adaptado para búsquedas de vecinos cercanos. El procedimiento general se describe a continuación.

Sea $\mathbb{N}$ un conjunto de puntos no revisados. Inicialmente $\mathbb{N} = \mathbb{U}$ el diccionario completo. Sea q la consulta.

- Seleccionamos n_i de $\mathbb{N}$. Sea $r = d(q, n_i)$.

- Ahora sea $\mathbb{N} = (q, r)_d$, si el número de elementos en $\mathbb{N}$ es mayor a 1, repetir el paso anterior, En caso contrario detener.

Para convertir este procedimiento en un algoritmo, es necesario hacer uso de una regla de selección para el "siguiente punto". La selección aleatoria es una alternativa no del todo inefectiva.

Una Regla de selección efectiva. Considere la Figura 2.3. Los puntos dentro de la región encerrada son el resultado de la consulta de rango $(q, d(q, n_i))_D$ usando dos pivotes. El proceso de selección elegirá el punto n_{i+1} tal que $d(q, n_{i+1}) < d(q, n_i)$. Podemos aproximar la distancia original $d(\cdot, \cdot)$ con la distancia máxima $D(\cdot, \cdot)$. En otras palabras, una heurística efectiva consiste en seleccionar n_{i+1} como el punto más cercano bajo $D(\cdot, \cdot)$. Otra alternativa es usar una distancia más estricta definida $\Delta(x, y) = \Sigma_i\{||d(x, p_i) - d(y, p_i)||\}$ o la norma L_1.

Una versión de este procedimientos se usa de manera eficiente en el algoritmo LAESA, de cualquier forma la selección del "siguiente mejor" punto se hace usando fuerza bruta, i.e realmente calculando $\Delta(\cdot, \cdot)$ para todo punto dentro del espacio

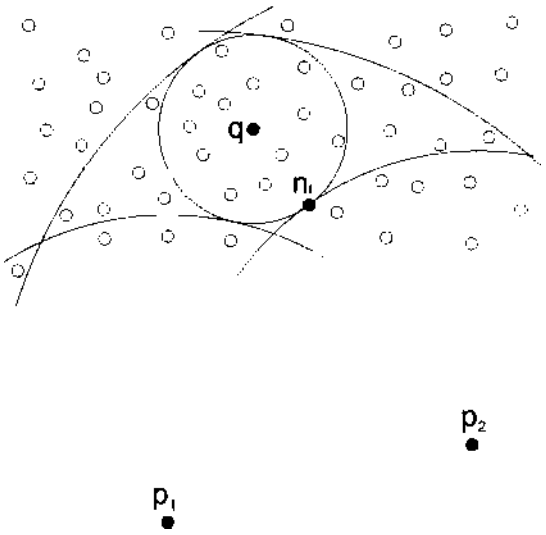


Figura 2.3: Un paso en el proceso de "Branch and Bound"

vectorial objetivo.

Haciendo uso del *Spaghetti* se puede seleccionar una lista de candidatos adecuada para el "siguiente mejor". El proceso se muestra a continuación:

- Encontrar q en cada arreglo del *Spaghetti*.
- Seleccionar ℓ sitios por encima y por debajo de la posición q.
- Seleccionar el mejor n_{i+1}, seguir cada puntero en cada sitio evaluando ya sea $D(\cdot,\cdot)$ o $\Delta(\cdot,\cdot)$, manteniendo el máximo o agregándole según corresponda.

Necesitamos inspeccionar ℓk^2 valores para obtener $D(\cdot,\cdot)$ o $\Delta(\cdot,\cdot)$.

El enfoque heurístico descrito no necesariamente da el mínimo valor, pero para pequeños valores de ℓ (3 *a* 5) se obtienen buenas aproximaciones. La aproximación mejora de forma monotónica a medida que ℓ incrementa.

2.6. Resumen del Capítulo

La estructura *Spaghetti* tiene tiempo de proceso y costo de almacenamiento lineales. Hemos mostrado como se construye y como se resuelve una consulta.

El *Spaghetti* utiliza una representación continua de las distancias en lugar de discretizarlas. Esta característica es muy útil para resolver consultas tanto de rango como de vecinos cercanos.

La inserción/borrado tanto de elementos en la base de datos como de pivotes ayuda a lidiar con las bases de datos no estacionarias. En particular la inserción y borrado de pivotes no puede hacerse en una estructura de árbol sin tener que reconstruirlo.

Capítulo 3

Conceptos de Interés

3.1. Estructuras Sucintas

Las estructuras de datos compactas son de utilidad en aplicaciones que son computacionalmente muy demandantes y requieren mantener los datos en la memoria principal, como es el caso de los motores de búsqueda. Siguiendo este contexto es importante almacenar los datos comprimidos y seguir siendo capaces de recuperarlos y actualizarlos de forma eficiente.

Una estructura sucinta, es una forma particular de almacenar y organizar los datos en una base de datos en espacio de almacenamiento proporcional al que toma almacenar los datos no comprimidos y provee de ciertas funcionalidades para mantener la capacidad de resolver consultas de forma eficiente. Una estructura de datos comprimida es aquella que su tamaño es proporcional al de la entropía de los datos originales (ver la siguiente sección). Por ejemplo, si una secuencia utiliza $n \log_2 \sigma$ bits de espacio. Una estructura de datos que utiliza $O(n \log \sigma)$ bits es sucinta, y otra que usa $O(nH_0(s))$ bits de espacio es comprimida. [González del Barrio, 2008]

Las estructuras sucintas son diseñadas para operar en RAM con un costo de palabra uniforme de tamaño $b = O(\log_2 n)$; en particular, un acceso a memoria puede escribir o leer una palabra de b bits. La estructura de datos debe ser capaz de resolver operaciones haciendo uso de $f(n) = o(n \log_2 \sigma)$ bits de espacio extra (i.e. $\lceil f(n)/b \rceil$

palabras de memoria). El uso típico de esos $f(n)$ es para almacenar directorios que permitan el acceso rápido a los datos y esto requiere descomprimir unas pocas palabras de la base de datos. El espacio usado por una estructura sucinta puede ser expresada como $n \log_2 \sigma + f(n) = n \log_2 \sigma + o(n \log_2 \sigma)$ bits.

3.1.1. Entropía Empírica

La entropía empírica es usada como medida de compresión para una cadena. Está definida para cualquier cadena individual y puede ser usada para medir el desempeño de un algoritmo de compresión.

Dada una cadena $S[1, n]$ en un alfabeto Σ de tamaño σ se define

$$H_0(S) = \sum_{c \in \Sigma} \frac{n_c}{n} \log_2 \frac{n}{n_c} \tag{3.1}$$

si hay n_c ocurrencias de c en S. Se entiende como la cota inferior a cualquier codificación de Σ que siempre asigne el mismo código al mismo símbolo. Esta definición puede ser generalizada al $k-esimo$ orden: sea Σ^k el conjunto de todas las secuencias de tamaño k que pueden formarse con un alfabeto Σ. Para cualquier cadena $w \in \Sigma k$, sea w_s la secuencia de caracteres que siguen a las ocurrencias de w en S. Entonces la entropía de orden k seria

$$H_k(S) = \frac{1}{n} \sum_{w \in \Sigma^k} |w_s| H_0(w_s) \tag{3.2}$$

esto es la cota inferior a codificaciones que consideren los k símbolos precedentes. [Cormen et al., 2001]

3.1.2. Rank y Select

Dado un vector de bits $\mathcal{B}[1, ..., n]$, definimos la operación $rank_0(\mathcal{B}, i)$(similar a $rank_1$) [González et al., 2005] como el número de 0's (1's) ocurridos hasta el la i-esima posición de $\mathcal{B}$.

$$rank_q(\mathcal{B}, i) = |\{k \in [0...x) : \mathcal{B}[k] = i\}|, \ q \in \{0, 1\} \tag{3.3}$$

La operación $select_0(\mathcal{B}, i)$(similar a $select_1$) [González et al., 2005] se define como la posición del i-esimo 0 (i-esimo 1) en $\mathcal{B}$.

$$select_q(\mathcal{B}, i) = min(\{k \in [0, ..., n) : rank_q(k) = i\}),\ q \in \{0, 1\} \tag{3.4}$$

3.2. Permutaciones

La permutación de un conjunto $p = \{1, ..., n\}$ es una lista de longitud n sin repetición formada por los elementos del conjunto p; al conjunto de todas estas listas lo llamaremos S_n:

S_n={listas formadas con los símbolos de conjunto p}

En estos términos podemos decir que una permutación es una (re)ordenación de los elementos del conjunto p. Existen $n!$ de permutaciones de p.

Otra forma de entender las permutaciones es como una función biyectiva $\pi : X \to X$ para un conjunto finito X.

Si π una permutación de p podemos representarla en la forma:

$$\pi = \begin{pmatrix} x_1 & x_2 & ... & x_n \\ x_{i1} & x_{i2} & ... & x_{in} \end{pmatrix}$$

donde $\pi(x_1) = x_{i1}, ..., \pi(x_n) = x_{in}$. Podemos simplificar la notación eliminando las x, para obtener

$$f = \begin{pmatrix} 1 & 2 & ... & n \\ i_1 & i_2 & ... & i_n \end{pmatrix}$$

Ejemplo tomado de [Rotman, 1996]

3.2.1. Composición de Permutaciones.

Sean f y g permutaciones del conjunto p, la composición de f y g se escribe como:

$$(f \circ g)(x) = f(g(x))$$

La aplicación de la permutación g sobre los elementos de p, devuelve los mismos elementos, quizá reordenados. Así que la acción posterior de f está igualmente bien definida. Pero además estas dos acciones sucesivas producen, en total, una nueva función biyectiva del conjunto (una nueva permutación).

La composición de las permutaciones nos permite definir una serie de propiedades interesantes:

- Si $f, g \in S_n$ entonces $f \circ g \in S_n$ (el conjunto s_n es cerrado para la composición).
- $f, g, h \in S_n$, entonces $(f \circ g) \circ h = f \circ (g \circ h)$, es asociativo.
- Existe el "elemento neutro" o identidad $\mathbb{I}$; es decir $\mathbb{I} \circ f = f$ y $f \circ \mathbb{I} = f, \forall f \in S_n$
- Para cada permutación $f \in S_n$, existe su inverso que denotaremos como f^{-1}, $f \circ f^{-1} = f^{-1} \circ f = \mathcal{I}$

En general la composición de permutaciones no es conmutativa. Es decir, es relevante el orden. Si f y g son de la forma

$$f = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 3 & 1 & 4 & 5 & 6 \end{pmatrix}$$

$$g = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 1 & 2 & 5 & 6 \end{pmatrix}$$

Entonces

$$f \circ g = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 1 & 3 & 2 & 5 & 6 \end{pmatrix}$$

$$g \circ f = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 4 & 2 & 3 & 5 & 6 \end{pmatrix}$$

$$f \circ g \neq g \circ f$$

Por lo tanto podemos decir que S_n no es abeliano, para $n > 2$.

3.2.2. Segmentación Cíclica

Sea i_1, i_2,...,i_n distintos enteros en $\{1, 2, ..., n\}$, si $\pi \in s_n$ tal que

$$\pi(i_1) = i_2,\ \pi(i_2) = i_3,...\pi(i_{n-1}) = i_n, f(i_n) = i_1$$

entonces π es una permutación cíclica de tamaño n, consideremos la permutación

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 5 & 3 & 1 \end{pmatrix}$$

la notación no ayuda a reconocer que f es una permutación cíclica de tamaño 5: $\pi(1) = 2$, $\pi(2) = 4$, $\pi(4) = 3$, $\pi(3) = 5$ y $\pi(5) = 1$, para esto hay una notación diferente

$$\pi = (i_1, i_2, ..., i_n)$$

para el ejemplo anterior $\pi = (1, 2, 4, 3, 5)$. Debemos notar que

$$\pi_1 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \end{pmatrix}$$

π_1 no es un ciclo, sin embargo, podemos representarlo como $\pi_1 = (1, 2)(4, 3)$.

3.3. Permutaciones Compactas

El objetivo es crear una estructura de permutaciones que requiera poco espacio para ser almacenada pero aun así permita realizar operaciones en ella rápidamente. Inicialmente se desea resolver la permutación π y su inversa π^{-1} en tiempo constante, Para lograr esto en [Munro et al., 2003] muestran como representar n números $a_1, ..., a_n$, donde $a_i \leq r$, $\forall i \leq n$ en $n \log_2 n + o(n)$ bits, y podamos visitar el i-esimo número en tiempo $O(1)$. (Una representación directa toma $n\lceil \log_2 r \rceil$ bits, estos es $O(n)$ bits más que $n \log_2 n$ para el peor de los casos.)

Tenemos una representación arbitraria de una permutación que toma $n \log n + o(n)$ bits y que permite resolver $\pi()$ en tiempo constante. Sea A esta representación y $t \geq 2$ un parámetro, almacenamos adicionalmente los ciclos de la estructura de la permutación, por cada ciclo cuya longitud sea al menos de t almacenamos un puntero a los elementos que se encuentren a una distancia múltiplo de t pasos de un punto de inicio arbitrario.

Ejemplo: Sea

$$\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 5 & 9 & 4 & 12 & 11 & 2 & 1 & 3 & 0 & 8 & 13 & 6 & 10 & 7 \end{pmatrix}$$

$$\pi = (2, 4, 11, 6, 1, 9, 8, 0, 5)(7, 3, 12, 10, 13)$$

si tenemos $x = 4$ calcular $\pi(x)$ lo hacemos en tiempo constante visitando la cuarta posición del arreglo de la permutación, sin embargo calcular $\pi^{-1}(x)$ sin que esta sea almacenada de forma explicita nos obliga a una búsqueda exhaustiva, esto es $O(n)$. Haciendo uso de la construcción sucinta de π con $t = 3$ que se muestra en la Figura 3.1, resolvemos $\pi{-}1(x)$ siguiendo el *Algoritmo 2* para una x dada, se resuelve en $O(t)$ pasos, para el ejemplo encontramos $\Pi^{-1}(x)$ siguiendo la permutación, es decir, $\pi(x) = 11$, $\pi(11) = 6$; $\pi(6) = 1$, sin embargo a este se le asigno un puntero hacia atrás, seguimos este puntero y decimos que $\pi(6) = 2$, finalmente $\pi(2) = 4$ de ahí que

$\pi^{-1}(4) = 2$.

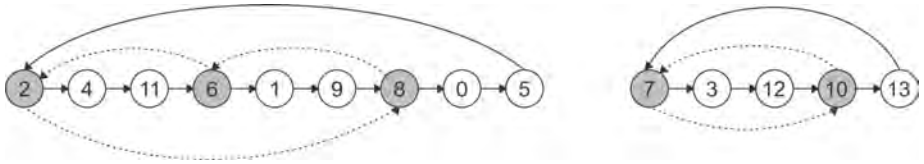


Figura 3.1: Permutaciones sucintas. Las líneas sólidas representan la permutación, las líneas discontinuas los punteros hacia atrás. Los nodos sombreados indican las posiciones que tienen punteros.

Podemos encontrar el valor del puntero en la secuencia de valores S, pero aún tenemos la necesidad de identificar los índices que tienen puntero, lo logramos si conocemos el *rank* de los índices que contienen punteros.

Algoritmo 2 Encontrar permutación inversa

```
i ← x
while π(i) ≠ x do
  if i tiene puntero then
    j = puntero(i)
  else
    j = π(i)
  end if
  i = j
end while
return i
```

Ya que tenemos un puntero por cada t elementos de un ciclo, el número de cálculos de π hechos por el algoritmo es de a lo mas $t+1$. Entonces el algoritmo para calcular π^{-1} toma a lo mas $O(t)$ pasos

3.4. Resumen del Capítulo

Una **estructura sucinta** es una forma particular de almacenar y organizar los datos haciendo uso de espacio de almacenamiento proporcional al que toma hacerlo

sin comprimir los datos y provee de algunas funcionalidades, son de utilidad en aplicaciones que son computacionalmente muy demandantes.

La operación Rank se define como el número de ocurrencias de un 1 o 0 ocurridos hasta un posición dada de una cadena de bits. Mientras que Select nos devuelve la posición en la que se encuentra el 0 o 1 para una ocurrencia dada.

Las **permutaciones** se pueden entender como una función biyectiva $f : X \to X$ para un conjunto finito X. Al conjunto de todas las permutaciones para un conjunto se le conoce como S_n.

La composición de permutaciones $(f \circ g) = f(g(x))$ produce una nueva permutación del conjunto y por lo tanto esta forma parte del conjunto S_n.

La separación cíclica de una permutación dice que esta puede se dividida en ciclos, decir que dada la permutación f un ciclo se da cuando $f(x) = y$, $f(y) = z$ y $f(z) = x$. Una permutación puede tener uno o más ciclos.

Permutaciones Compactas

El objetivo es crear una estructura para almacenar la permutación de un conjunto utilizando tan poco espacio como sea posible pero manteniendo la capacidad de resolver la permutación directa y su inversa.

Capítulo 4

Nueva Representación del "Spaghetti"

La primera etapa de todos los índices basados en pivotes consiste en encontrar los puntos dentro de una región hipercúbica en el espacio, estos puntos son la lista de candidatos. Cuando se usa el *Spaghetti* para resolver una consulta debemos encontrar los subconjuntos $S_i = \{x : |d(x, p_i) - d(q, p_i)| < r\}, i = 1, ..., k$, cada uno de estos subconjuntos son los elementos que se encuentran dentro de la región hipercúbica correspondiente a cada pivote, la lista de candidatos será la intersección de estos.

Nuestra meta es obtener un algoritmo que obtenga estos posibles candidatos y calcule la intersección con menor esfuerzo del que toma utilizar la estructura tradicional del *Spaghetti*, además de hacer uso de la estructura compacta para almacenar las permutaciones y así reducir el espacio necesario para almacenar el índice.

4.1. Cambiando la estructura del Spaghetti

Durante la construcción de la estructura del *Spaghetti* se calcula y guarda en arreglos la distancia de cada pivote a todos los elementos en la base de datos, estas son almacenadas en arreglos (uno por cada pivote), son ordenados teniendo cuidado de guardar los punteros a los elementos en cada uno de los arreglos de distancia.

En la nueva representación del *Spaghetti* cuando se ordenan los arreglos de distancia, los punteros entre los elementos son sustituidos por punteros que guardan la permutación de los elementos con respecto a un orden original dado (como se muestra en la figura 4.1), al que podemos entender como un arreglo de identidad, donde a cada elemento se le asigna un identificador único. Esto elimina la dependencia existente entre los arreglos en el *Spaghetti* original, lo que permite establecer técnicas que disminuyen el número de operaciones necesarias para realizar el cálculo de una intersección.

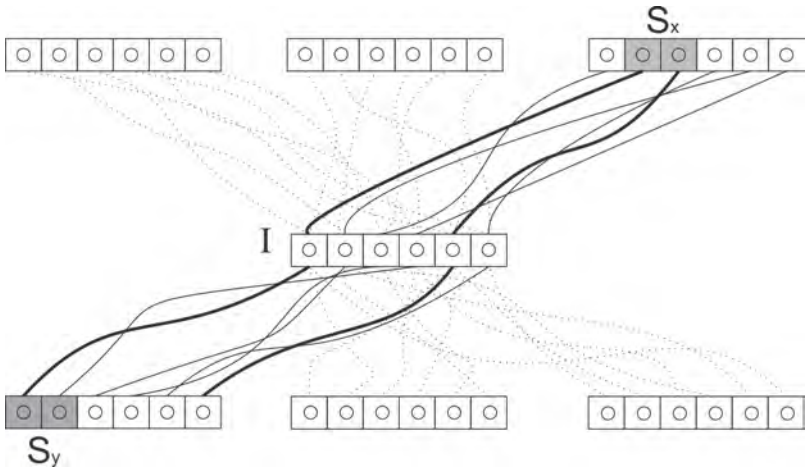


Figura 4.1: Representación de la nueva estructura del Spaghetti

De esta manera para encontrar donde se encuentra un elemento en un pivote conociendo su ubicación en otro pivote, primero deberemos localizar cual era su posición en el orden original, esto es calcular la permutación π, conociendo la posición en el orden original basta con calcular la permutación inversa π^{-1} a un pivote seleccionado.

4.1.1. Representación Sucinta

Hay diversas motivaciones para usar representaciones sucintas de datos, debido a que uno de los mayores problemas de los índices basados en pivotes, es que requieren de gran cantidad de espacio de almacenamiento, es deseable representar los arreglos de permutaciones utilizando la menor cantidad de espacio y aun así seguir

siendo capaces de calcular $\pi(i)$ y $\pi^{-1}(i)$ tan rápido como sea posible para cualquier i.

En [Munro et al., 2003] muestran un que es posible guardar la permutación en $(1 + \frac{1}{t})n \log_2(n) + O(\frac{n \cdot \log_2(\log_2(c))}{Lg(n)})$, calculando $\pi(i)$ en $O(1)$ y $\pi^{-1}(i)$ en $O(t)$, este proceso se detalla en la sección 3.3 de este documento.

Producto de la compresión se genera un nuevo parametro t, que se puede entender como el factor de compresión, cuando más grande es t menor es el espacio que requiere para almacenar las permutaciones, pero se incrementa el tiempo que lleva calcular una permutación inversa.

4.1.2. Capacidades Dinámicas

La nueva estructura mantiene las propiedades dinámicas del *Spaghetti*, la inserción y borrado de elementos se hace seleccionando una columna en el centro del *Spaghetti* y siguiendo la posición del elemento a borrar o buscando donde debe ser insertado, teniendo cuidado de mantener actualizadas las permutaciones.

En el caso de los pivotes es un proceso más sencillo que en el *Spaghetti* original, para eliminar o insertar un pivote, debido a que no existe relación directa entre los arreglos no es necesario redireccionar los punteros, simplemente para insertar un pivote se calcula las distancias de este a todos los elementos, se ordenan guardando el puntero a donde se localizan en el arreglo de identidad y se agrega al índice, para borrar só lo hace falta retirar del índice el arreglo correspondiente al pivote eliminado.

4.2. Cálculo de la Intersección

Dados k arreglos de distancia ordenados y una consulta q, se definen k intervalos, $[a_1, b_1]$,...,$[a_k, b_k]$(con $a_i = d(P_i, q) - r$ y $b_i = d(P_i, q) + r$), con estos intervalos obtenemos los índices del intervalo $[I_1, J_1]$,...,$[I_k, J_k]$ correspondientes a cada arreglo ordenado, esto nos lleva a encontrar los subconjuntos $S_i = \{x : |d(x, p_i) - d(q, p_i)| \leq r\}, i = 1, ..., k$. La intersección de estos se define como el conjunto de elementos que

se encuentra presente en los k distintos arreglos.

Se calcula la intersección para establecer una "lista de candidatos", la conformaran aquellos elementos que se encuentren dentro del intervalo $[a_i, b_i]$ de un primer pivote, mismos que fueron identificados al obtener los subconjuntos S_i. Es necesario verificar si los elementos de la lista de candidatos se encuentran dentro de los conjuntos S_i correspondiente a todos los pivotes, la intención es eliminar aquellos que no se encuentren dentro de dicho conjunto y así reducir la lista de candidatos, durante el proceso de construcción del índice se guardaron punteros a un arreglo de identidad, que para la nueva estructura del *Spaghetti* son representados por permutaciónes compactas, ahora podemos encontrar la posición de los elementos en los arreglos de distancia asociados a cada pivote sin la necesidad de recorrer el *Spaghetti*, entonces para encontrar la posición de los elementos de la lista de candidatos primero debemos encontrar su posición original encontrando su permutación π, para saber donde se encuentran en los arreglos restantes es necesario calcular la permutación $\pi-1$.

Para eliminar elementos de la lista de candidatos se utilizaron dos técnicas distintas, la primera de ellas es haciendo *eliminación elemento a elemento*. Esta consiste en tomar un elemento de la lista de candidatos y verificar si este se encuentra en los k conjuntos S_i asociado con los pivotes, interrumpiendo la ejecución en aquello que no lo haga, en caso contrario sabremos que forma parte de la intersección, repitiendo el proceso para cada uno de los elementos en la lista de candidatos, este proceso se muestra en el *Algoritmo 3*.

La segunda es haciendo un búsqueda de los elemento en la lista de candidatos en todos los conjuntos S_i. En cada iteración son eliminados de la lista de candidatos los elementos que no se encontraron en el conjunto que estaba siendo revisado. Después de k iteraciones los elementos en la lista de candidatos serán la intersección. El proceso completo haciendo uso de esta última técnica se muestra en el *Algoritmo 4*:

Algoritmo 3 Eliminación Elemento a Elemento

```
1: v ← 0
2: for i ← 0 to i = |S_0| do
3:    A[i] ← S_0(i)
4: end for
5: for j ← 0 to j = |A| do
6:    C ← S_i.Inversa(A[j]) n ← 1
7:    for i ← 1 to k do
8:       if C ∈ S_i then
9:          n ← n + 1
10:      else
11:         break
12:      end if
13:   end for
14:   if n = k then
15:      B[v] ← C
16:      v = v + 1
17:   end if
18: end for
19: return B
```

4.2.1. Pequeño Contra Pequeño(SVS)

En la práctica los conjuntos de candidatos que no han sido descartados en los intervalos de los pivotes pueden ser de cientos de miles de elementos, por consiguiente es de utilidad tener un algoritmo que sea eficiente y rápido en promedio.

Decidir el orden de visita de los pivotes para descartar elementos de la lista de candidatos es posible gracias al cambio en la estructura de construcción del *Spaghetti*. En este trabajo se utilizaron dos técnicas distintas, la primera de ellas se conoce como Small vs Small.

El algoritmo trabaja de forma iterativa con pares de conjuntos, como se muestra a continuación. Sea $S_i = \{x : |d(x, p_i) - d(q, p_i)| \leq r\}, i = 1, ..., k$, sin perder generalidad consideramos que $|S_1| \leq |S_2| \leq ... \leq |S_k|$ (i.e. Si $S_1 > S_2$ sólo renombramos $S_1 \leftrightarrow S_2$), calculamos $\bigcap_{i=1}^{i=k} S_i$. Ver *Algoritmo 5*

Algoritmo 4 Eliminación Por Bloque

```
1: for i ← 0 to i = |S_0| do
2:    A[i] ← S_0(i)
3: end for
4: for i ← 1 to k do
5:    n ← 0
6:    for j ← 0 to j = |A| do
7:       C ← S_i.Inversa(A[j])
8:       if C ∈ S_i then
9:          B[n] ← A[i]
10:         n ← n + 1
11:      end if
12:   end for
13:   A ← B
14: end for
15: return A
```

Algoritmo 5 SVS

```
1: Ordenar los conjuntos por tamaño |S_0| ≤ |S_1| ≤ ... ≤ S_k
2: A ← S_0
3: for i ← 2 to k do
4:    n ← 0
5:    for j ← 0 to j = |A| do
6:       if A[j] ∈ S_i then
7:          B[n] ← A[j]
8:          n ← n + 1
9:       end if
10:      A ← B
11:   end for
12: end for
13: return A
```

El método SVS es simple y efectivo, se identifica el conjunto más pequeño y este es intersectado con cada uno de los conjuntos restantes en orden ascendente por su cardinal. Así las lista de candidatos nunca será más grande que el conjunto S_i a inter-

sectarse, en el peor de los casos la intersección será igual al conjuntos S_x más pequeño.

4.2.2. Intersección Aleatoria

Podríamos tener la situación que el orden original nos lleve a revisar pivotes cercanos entre si, esto tendría como resultado que las listas de elementos descartados por cada uno de los pivote fuesen similares, incluso con el método SVS cabe la posibilidad de encontrar este fenómeno. Dicho de otro modo, existe la posibilidad de que $S_i \approx S_i \cap S_{i+1} \approx S_{i+1}$. Esto no es deseable ya que se desea que la lista de candidatos se reduzca lo más rápidamente como se pueda lograr, evitando el mayor número de operaciones posible.

Entonces, si tenemos un elemento donde debido al orden de selección de los pivotes permanece dentro de la lista de candidatos durante la primera mitad de la ejecución del *Spaghetti* para ser eliminado de la lista inmediatamente después, toma $\frac{k}{2} + 1$ operaciones, cambiando el orden de visita por un orden aleatorio nos da como resultado que dicho elemento tenga un 50 % de probabilidad de ser eliminado en la primer iteración del *Spaghetti*, y ya que el hecho de que haya o no sido eliminado por el primer pivote no afecta que sea o no eliminado por el segundo, la probabilidad de que el elemento no sea eliminado después de la segunda iteración es de 25 %, después de unas pocas iteraciónes la probabilidad que el elemento permanezca en la lista de candidatos es muy pequeña.

4.2.3. Análisis de la Complejidad en Tiempo

Por simplicidad consideremos que cada intervalo tiene en cada arreglo m elementos, tomemos s como el tamaño de la intersección, el costo mínimo para encontrar la intersección es ks; ya que la ejecución no se puede terminar hasta comprobar que los s elementos se encuentren dentro de los k intervalos. En [Chávez González, 1999, Chávez et al., 1999] se demuestra que el costo de calcular la intersección es de:

$$\frac{1 - (\frac{m-s}{n})^{k-1}}{1 - (\frac{m-s}{n})} + ks$$

Inconvenientemente debido a la nueva estructura hace falta agregar el costo de calcular π^{-1}, esto es t por cada elemento en cada una de las k iteraciones para calcular la intersección:

$$t \cdot \left(\frac{1 - (\frac{m-s}{n})^{k-1}}{1 - (\frac{m-s}{n})} + ks \right)$$

Invertir el orden de la permutación

Al reducir el costo de almacenamiento hemos aumentado el costo de ejecución, aun así es posible mejorar, la permutación inversa es calculada en el peor de los casos $m(k-1)$ ocasiones mientras que en el caso de la premutación directa solamente m, entonces si al construir el índice guardamos π^{-1} en cambio de π conseguiremos invertir estos tiempos como consecuencia, dicho de otra forma, sólo aumenta el tiempo con respecto al *Spaghetti* al encontrar la lista de candidatos:

$$\frac{1 - (\frac{m-s}{n})^{k-1}}{1 - (\frac{m-s}{n})} + ks + tm$$

4.2.4. Resumen del Capítulo

La nueva estructura del *Spaghetti* tiene como objetivo reducir el espacio necesario para almacenarse y calcular menor número de comparaciones, la intersección entre los conjuntos de puntos no eliminados con la información tomada de los pivotes, con respecto al *Spaghetti* tradicional.

Para lograrlo se reestructura el índice almacenando la permutación que existe con un conjunto de identidad, así se elimina la dependencia entre las listas de candidatos de cada pivote, lo que permite cambiar el orden en que se intersectan los conjuntos. Dos algoritmos se probaron:

- SVS, pequeño contra pequeño, intersecta los conjuntos en orden ascendente por su cardinal.

- Aleatorio, calcula la intersección de los conjuntos seleccionando el orden de forma aleatoria, aprovechando la probabilidad conjunta de que un elemento que debe ser eliminado permanezca en la lista de candidatos con el avance iterativo del método.

Cada uno de estos fue probado para dos algoritmos diferentes de eliminación de elementos en la lista de candidatos:

- Elemento a Elemento, toma un elemento de la lista de candidatos y revisa en cada pivote si es que forma parte de la intersección antes de proceder al siguiente elemento.

- Por Bloque, revisa todos los elementos en la lista de candidatos en un pivote, eliminando aquellos que no se encuentren dentro del rango de distancia establecido antes de continuar su ejecución en el siguiente pivote.

Capítulo 5

Evaluación de Rendimiento

5.1. Datos de Prueba

Los índices métricos son capaces de hacer usos de datos complejos (documentos, imágenes, sonido, etc.), sin embargo el desempeño del *Spaghetti* ya ha sido documentado en [Chávez González, 1999, Chávez et al., 1999], en este trabajo se desea mejorar el tiempo que toma calcular la intersección y reducir el espacio necesario para almacenar el índice, teniendo en cuenta esto para evaluar el comportamiento de nuestra técnica se generaron bases de datos sintéticas a partir de vectores sobre seis dimensionalidades diferentes (4, 8, 12, 16, 20 y 24). cada vector se generó usando números aleatorios tomando valores entre 0 y 1. Hay que notar que aun cuando las bases de datos son espacios vectoriales, no estamos usando las coordenadas para descartar elementos. Usamos las distancias como cajas negras. Esto permite trabajar con los datos sin tener en cuenta su representación, todo lo que necesitamos es una función de distancia para indexar los datos. En cuanto al tamaño de las bases de datos utilizamos 7 distintos con tamaños de 10^4, 10^5, 2.5×10^5, 10^6, $5 * 10^6$ y 10^7 objetos. Las pruebas fueron ejecutadas haciendo uso de un procesador Intel core i5-2450M, 2.50 GHz, RAM 6 GB.

Dimensión	$\frac{\mu}{2r^2}$	d_max	μ	r
4	11.18	1.81	0.426	0.138
8	20.29	2.19	0.509	0.112
12	29.62	2.55	0.546	0.096
16	38.31	2.81	0.58	0.087
20	45.14	2.95	0.607	0.082
24	54.67	3.21	0.615	0.075

Estadísticas de las bases de datos. La media μ, desviación estándar r y la dimensión intrínseca ($\frac{\mu}{2r^2}$) como la describe [Chavez et al., 2001]

Los conjuntos de consultas contienen 256 vectores aleatorios. Con alta probabilidad, las consultas y las bases de datos son conjuntos disjuntos.

5.2. Eliminación Elemento a Elemento vs Eliminación por Bloque

Todos los experimentos que se muestran en las secciones siguientes fueron corridos usando ambas técnicas de eliminación de elementos, realizan el mismo número de comparaciones, la intención es verificar que los procesos de carga de memoria y uso de estos datos hicieran alguna diferencia, debido a que para los experimentos el índice se almacenó completamente en RAM, obtuvimos tiempos indistinguibles entre ambos métodos.

5.3. Selección de Pivotes

Se ha demostrado [Baeza-Yates y Navarro, 1998] que existe un número óptimo de pivotes para usarse en un índice. Desgraciadamente dicho número no es viable ya que la cantidad de memoria requerida para almacenar el índice no es alcanzable, incluso en bases de datos pequeñas. Por lo tanto, optimizar la selección de pivotes permite tener el mejor rendimiento posible, y esto es una gran área de estudio en los algoritmos basados en pivotes.

Se debe encontrar un modelo matemático y hacer una experimentación minuciosa para encontrar un método práctico para hacer la selección de pivotes. Los *Spaghetti* pueden ser una herramienta útil ya que permite la inserción y borrado de pivotes, esta característica es requerida para buscar el conjunto de pivotes óptimo. Sin embargo, este no es el objetivo del presente trabajo, para las pruebas se seleccionó un conjunto de pivotes $\mathbb{P}$ dentro de la base de datos $\mathbb{S}$ de forma aleatoria, $\mathbb{P} \in \mathbb{S}$.

5.4. Tamaño del índice

Los experimentos de esta sección consistieron en establecer una comparativa entre el espacio requerido para almacenar el *Spaghetti* contra su nueva representación compacta.

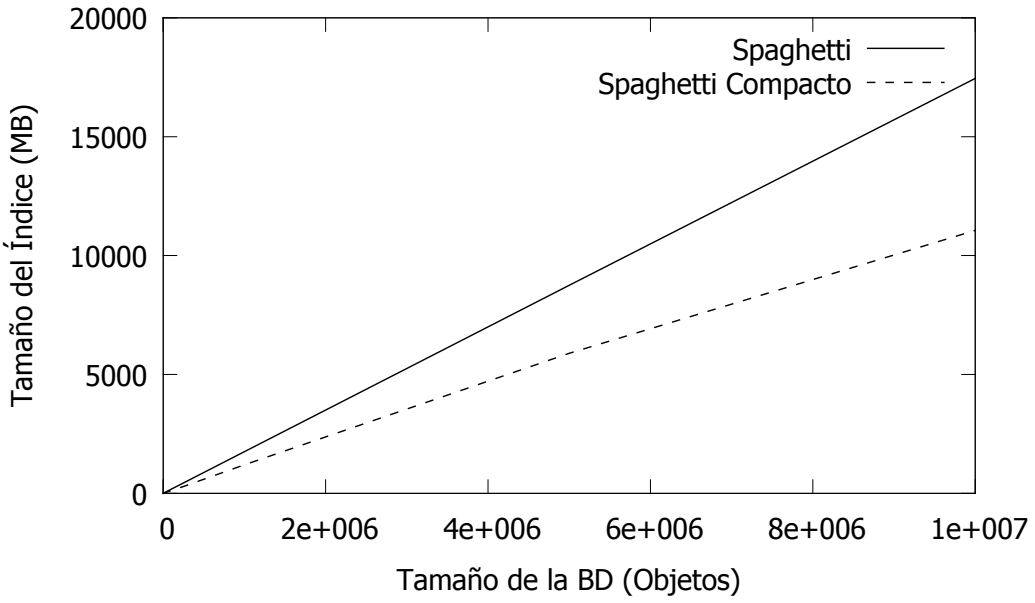


Figura 5.1: Tamaño del índice conforme aumenta el tamaño de la base de datos

La Figura 5.1 muestra el tamaño del índice conforme crece el número de elementos en la base de datos, para construirse se utilizaron bases de datos de dimension 20 con $t = 2$ (factor de compresión), con 120 pivotes, Mientras que en la Figura 5.2 se puede ver en como crece el tamaño del índice dependiendo de la cantidad de pivotes

usados para construirlo, se utiliza una base de datos de un millón de registros de dimension 20 y con $t = 2$.

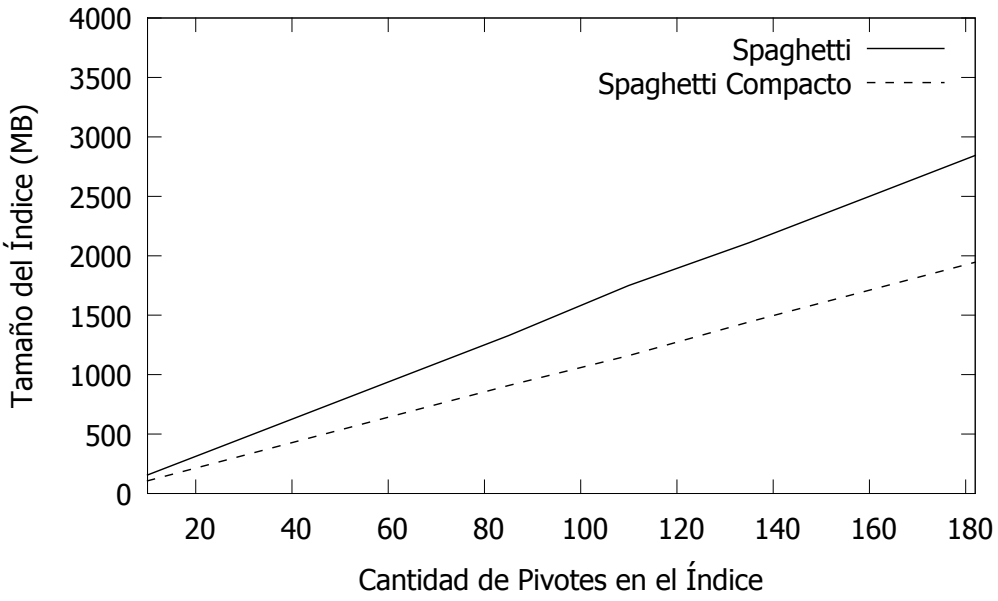


Figura 5.2: Tamaño del índice dependiendo de la cantidad de pivotes usados para construirlo

5.5. Parámetro t

El parámetro t es usado para construir los arreglos de permutaciones en su representación compacta durante el proceso de construcción del índice, se guarda un puntero por cada t elementos dentro de un "ciclo" de la permutación, esto permite calcular π^{-1} en $O(t)$.

En la Figura 5.3 podemos observar que el tamaño del índice se hace más pequeño a medida que t incrementa, sin embargo el tiempo que toma calcular la π^{-1} aumenta, esto tiene como resultado un incremento en el tiempo que toma resolver una consulta, esto se muestra en la Figura 5.4.

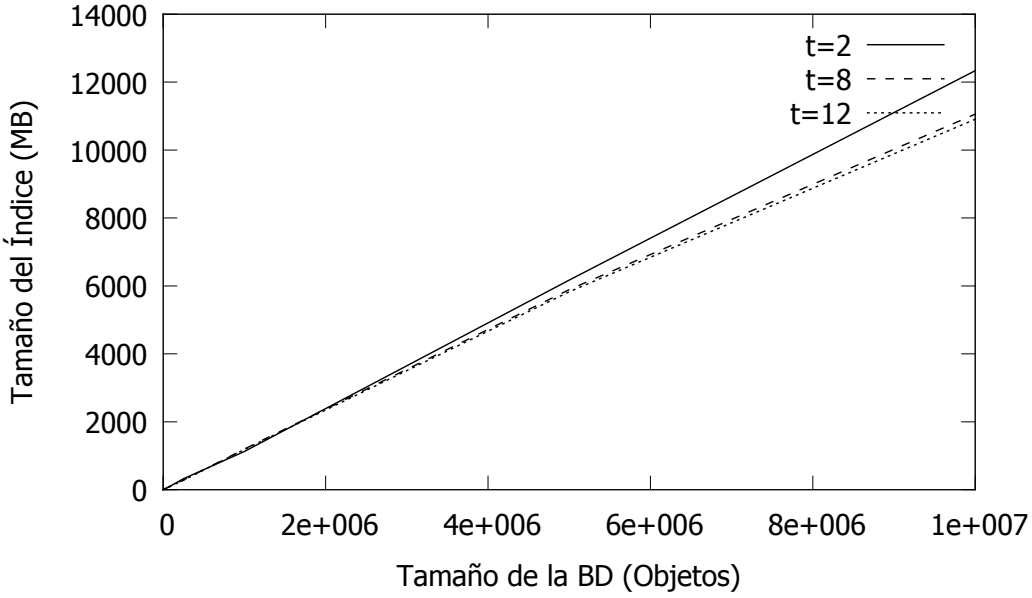


Figura 5.3: Tamaño del índice variando el parámetro de compresión t

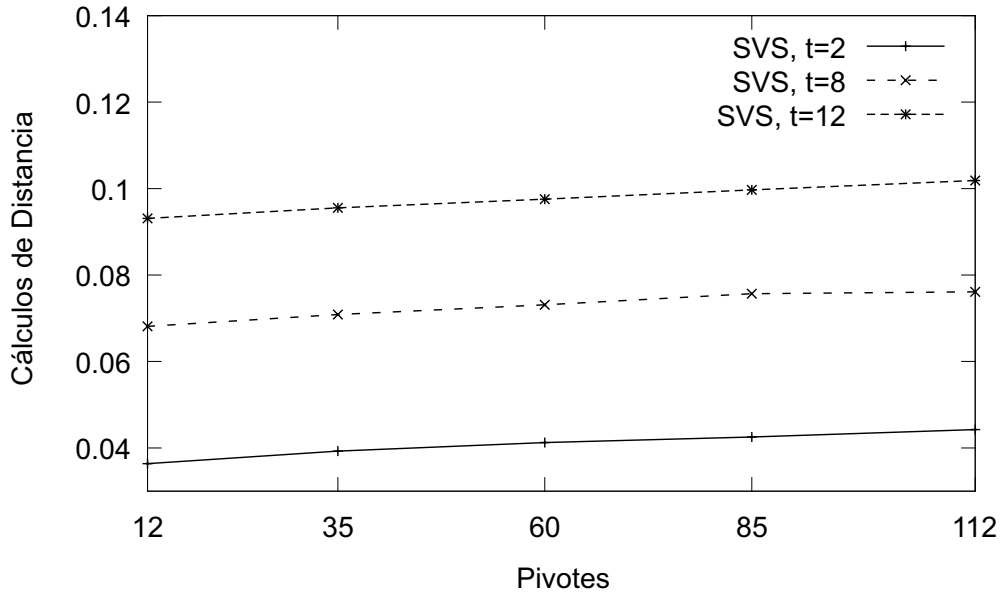


Figura 5.4: Tiempo para el cálculo de la intersección variando el parámetro t

5.6. Cálculos de Distancia

Uno de los objetivos de este trabajo es reducir el tiempo que toma calcular la intersección para obtener la lista de candidatos.

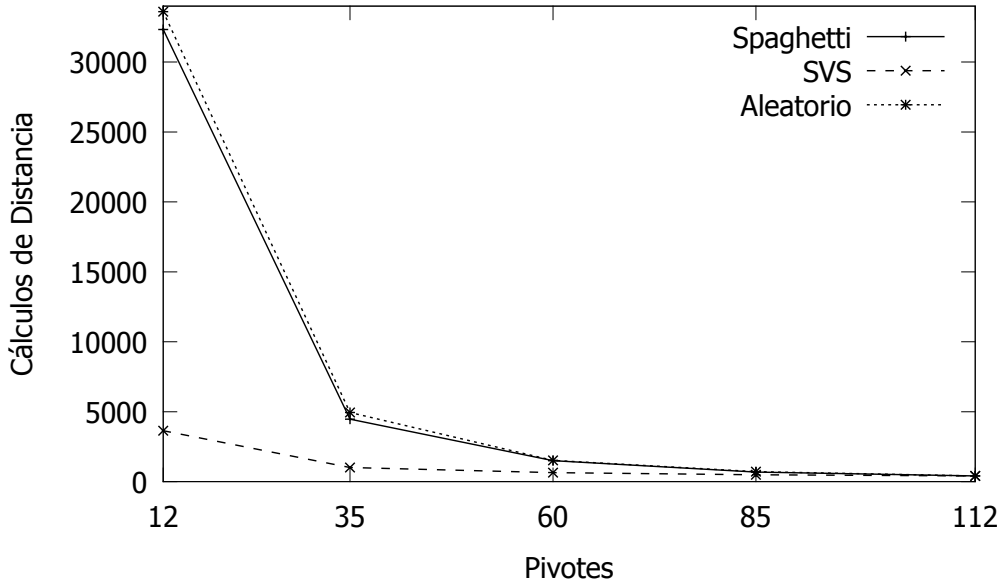
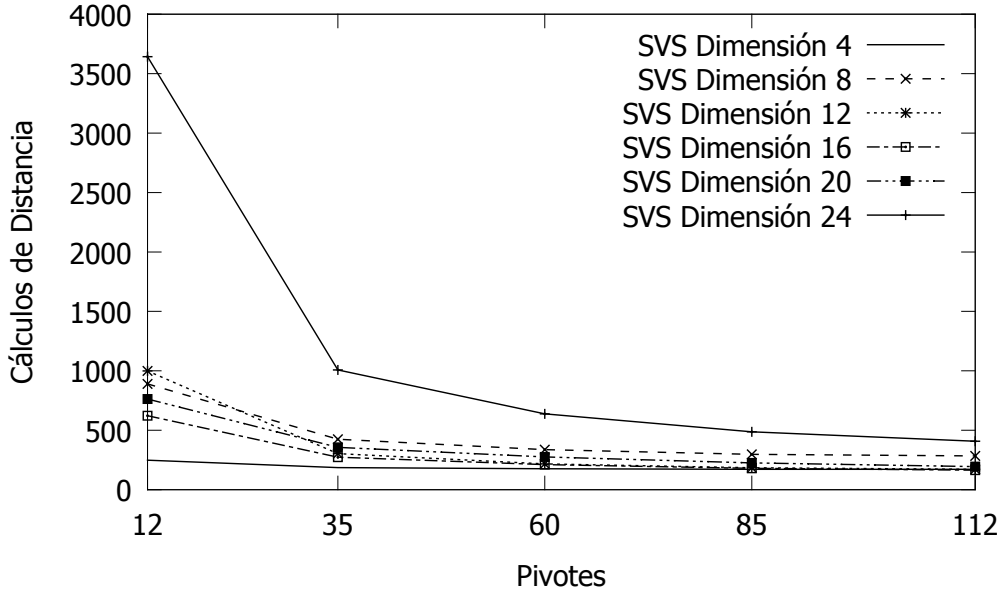


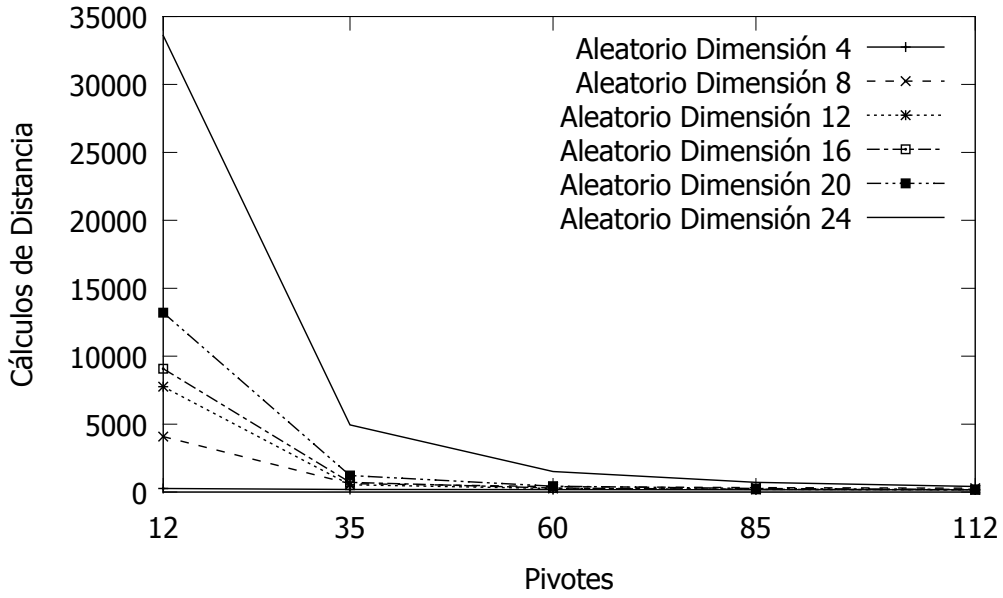
Figura 5.5: Aproximación a la intersección con el avance de la ejecución del Spaghetti

La Figura 5.5 muestra una comparativa entre la velocidad de aproximación a la intersección del *Spaghetti* y su nueva representación mediante las dos estrategias de selección del orden de los pivotes SVS y Aleatorio. Se puede observar que la estrategia SVS elimina una gran cantidad de puntos en las primeras iteraciones. Para esta prueba se utilizaron 112 pivotes y se tomó el número de candidatos después de 10, 35, 60, 85 y 112 iteraciones. Seleccionando de forma aleatoria el orden para calcular la intersección aproxima al comportamiento del *Spaghetti*, esto es por que originalmente los pivotes fueron seleccionados de forma aleatoria, esta estrategia tendría más sentido si se hubiese tomado alguna heurística para escoger los pivotes.

El desempeño de nuestra técnica respecto al número de puntos eliminados en progresión a los pivotes evaluados en distintas dimensiones se aprecia en la Figura



(a) SVS



(b) Selección Aleatoria

Figura 5.6: Efecto de la dimensión intrínseca en la ejecución del Spaghetti.

5.6. Nótese que a medida que la dimensión de los datos crece, se requieren más comparaciones para encontrar la intersección y el tamaño de esta también es más grande.

5.7. Cálculo de la Intersección

Esta sección muestra el tiempo que le toma al algoritmo calcular la intersección. Para estos ensayos se utilizó una base de datos de dimensión 24, con un millón de objetos. Los tiempos solo expresan el costo de calcular la intersección, el cálculo de distancias final para resolver la consulta no es contabilizado.

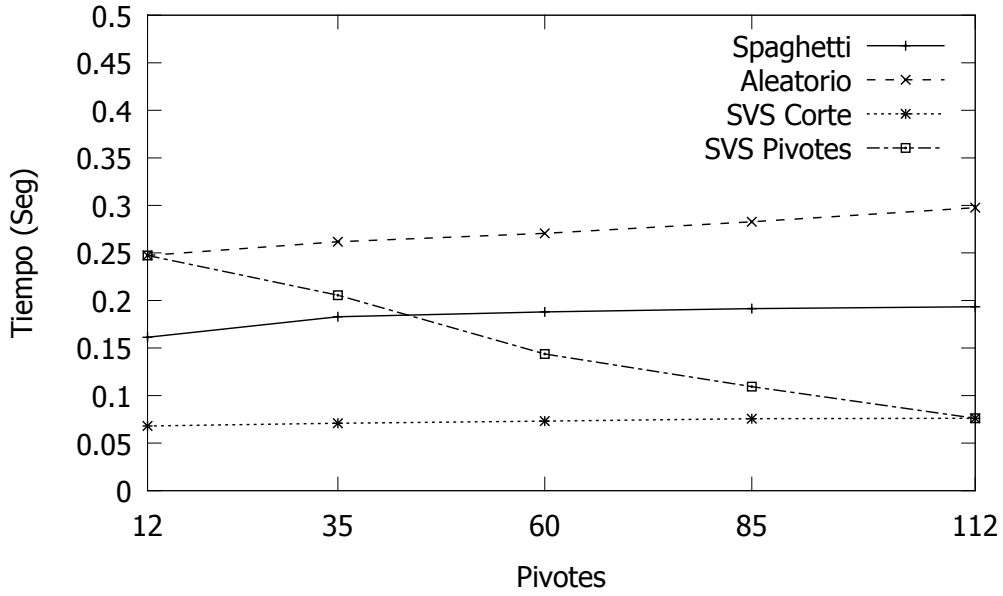
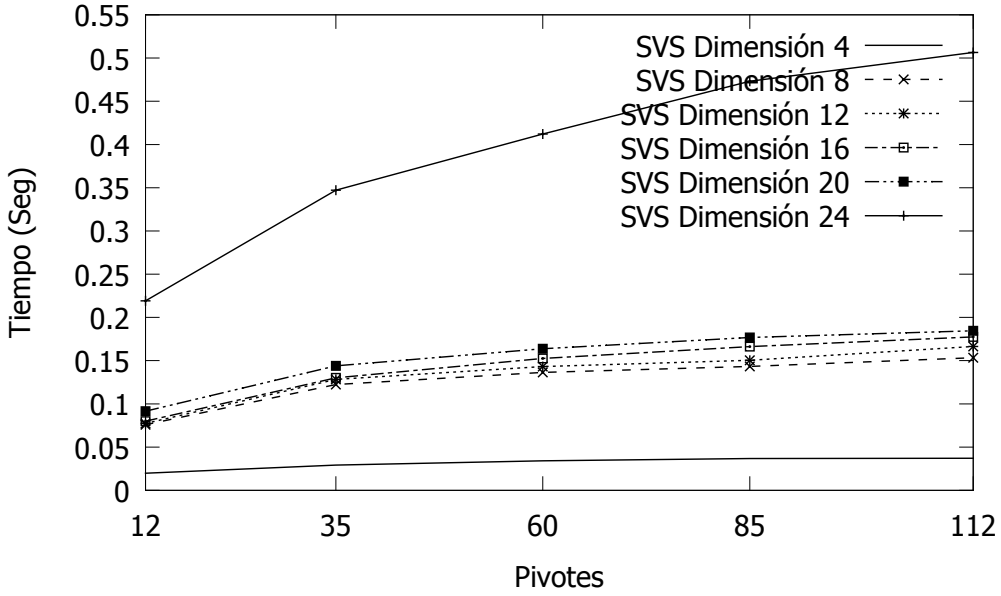
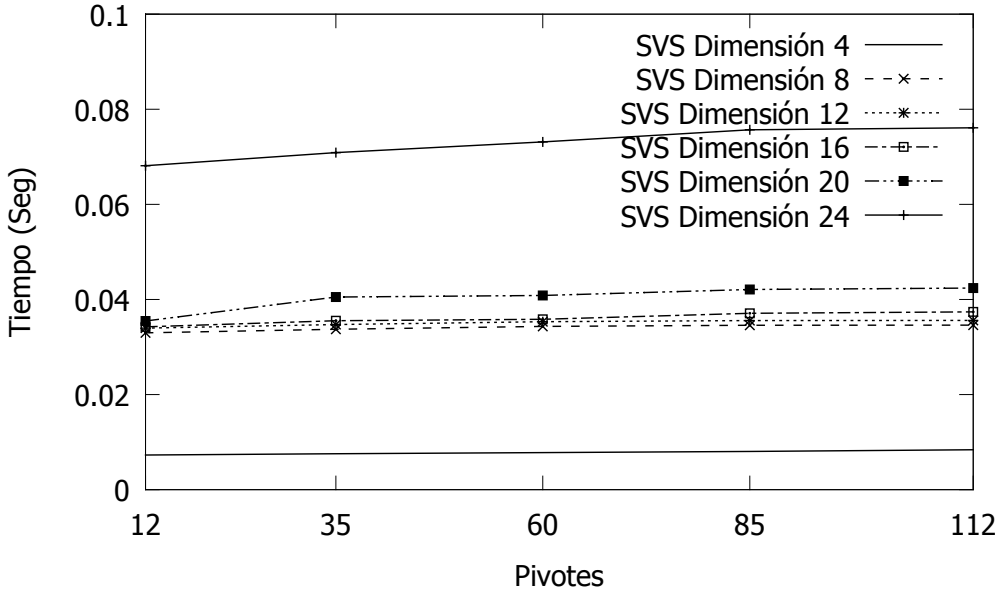


Figura 5.7: Tiempo para el cálculo de la intersección, comparativa entre estrategias, haciendo cortes en la ejecución para evaluar su desempeño después de algunas iteraciones. SVS pivotes refiere a la ejecución del algoritmo variando el número de pivotes que se usaron para construir el índice.

Podemos observar en la figura 5.7 el tiempo de ejecución utilizando la ejecución tradicional del *Spaghetti*, selección aleatoria y pequeño contra pequeño, para un índice de 120 pivotes, la ejecución se interrupe en diversos puntos para evaluar su desempeño; también se muestra como el número de pivotes afecta el rendimiento del algoritmo, haciéndolo únicamente para el SVS.

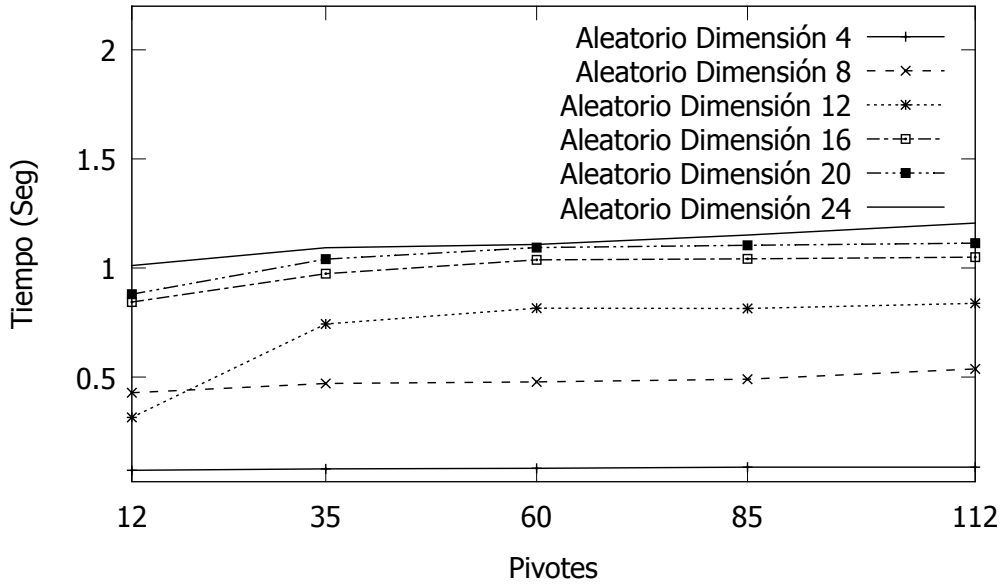


(a) Tiempos de Permutaciones NO Invetidas, SVS

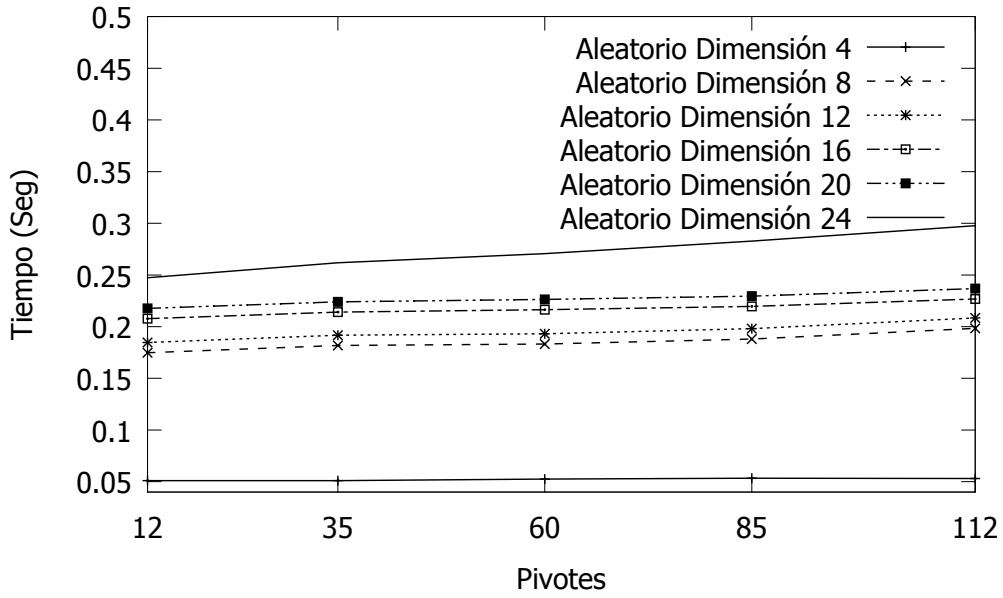


(b) Tiempos de Permutaciones Invetidas, SVS

Figura 5.8: Comparativa entre los tiempos de ejecución de la nueva estructura del *Spaghetti* haciendo uso de la estrategia SVS



(a) Tiempos de Permutaciones NO Invetidas



(b) Tiempos de Permutaciones Invetidas

Figura 5.9: Comparativa entre los tiempos de ejecución de la nueva estructura del *Spaghetti* haciendo uso de la estrategia aleatoria

Una comparativa en la Figura 5.8,5.9 del tiempo de ejecución del *Spaghetti* compacto invirtiendo las permutaciones cuando se construye el índice

En cuanto a calcular la intersección elemento a elemento o hacerlo por bloques no causa diferencia aparente en tiempo de ejecución.

5.8. Resultados

El *Spaghetti compacto* también se construye de igual manera que el *Spaghetti* en tiempo lineal sin embargo el espacio que se utiliza para almacenarse se reduce variando el parámetro t en la construcción de la permutación sucinta.

El tiempo que le lleva calcular la intersección también se reduce en el caso del algoritmo SVS. La intersección aleatoria tiene un comportamiento similar al del *Spaghetti*.

Las pruebas se corrieron en dos ocasiones, una eliminando candidatos elemento a elemento y la segunda eliminandoles por bloques, se concluyó que hacer el cálculo de la intersección elemento a elemento o por bloques completos no hace diferencia.

Capítulo 6

Conclusiones

En este trabajo se produjo una nueva técnica para la búsqueda por proximidad o similaridad en espacios métricos. Se tómo como base el *Spaghetti*, la idea consiste en hacer uso de estructuras compactas para almacenarlo y alterar la estructura del mismo.

Hemos presentado un estudio comparativo del comportamiento del *Spaghetti* comparándolo contra su nueva representación compacta como algoritmo basado en pivotes para resolver búsquedas por proximidad. Mostramos como es afectado en su rendimiento por la dimensión intrínseca y como su rendimiento es dependiente del número de pivotes.

Mostramos que es posible reducir el tiempo que toma calcular la intersección, mediante el uso de estrategias de selección del orden de ejecución.

6.1. Trabajo Futuro

Seleccionar los mejores pivotes. En este trabajo seleccionamos los pivotes de forma aleatoria, sin embargo hacer una selección mas cuidadosa de los pivotes podría mejorar el desempeño de los algoritmos basados en pivotes.

Comprimir las distancias. Si almacenamos distancias discretas entonces el *Spagheti* sería un Fixed Height Fixed Query Tree, entonces es deseable mantener el uso de distancias continuas, sin embargo se puede utilizar menor espacio para almacenarlas sin ganar mucho tiempo en ejecución, es decir es posible utilizar una estructura sucinta para almacenarlas.

Versión aproximada. El *Spaghetti* es un método de búsqueda exacto, es decir, que una consulta $(q, r)_d$ devuelve como resultado cabalmente los elementos que resuelven esta consulta. Se puede desarrollar una versión "más relajada" de algoritmo para resolver búsquedas donde se puede permitir perder respuestas relevantes o recibir algunas irrelevantes. Esto es una manera de lidiar con el efecto de la dimensión intrínseca en los índices métricos, algoritmos donde se intercambia precisión por velocidad.

Aplicación. Si se tiene en cuenta los requerimientos de memoria del *Spaghetti* es difícil pensar en aplicaciones del mundo real en donde pueda ser utilizado, sin embargo este tiene propiedades dinámicas muy interesante para ser usadas en un ambiente cambiante, cuando las bases de datos no son muy grandes. Una posible aplicación: es la compresión de vídeo, donde el índice debe de ser dinámico; puede ser usado para hacer búsqueda en imágenes o en audio, en la recuperación de información multimedia.

Otra aplicación interesante es la búsqueda de estrategias de selección de pivotes, es una muy buena opción ya que permiten la inserción y borrado de pivotes y elementos.

Bibliografía

[Baeza-Yates y Navarro, 1998] Baeza-Yates, R. y Navarro, G. (1998). Fast approximate string matching in a dicttionary. In *Procedings 5th South American Symposium on String Processing and Information Retrieval*.

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

[Bentley, 1975] Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Comunications of ACM*, 18.

[Bozkaya y Ozsoyoglu, 1997] Bozkaya, T. y Ozsoyoglu, M. (1997). Distance-based indexing for high-dimensional metric spaces. In *ACM SIGMOD international conference on Management of data*.

[Burkhard y Keller, 1973] Burkhard, W. A. y Keller, R. M. (1973). Some approaches to best-match file searching. *Comunications of ACM*, 16.

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

[Chávez et al., 1999] Chávez, E., Marroquín, J. L., y Baeza-Yates, R. A. (1999). Spaghettis: An array based algorithm for similarity queries in metric spaces. In *String Processing and Information Retrieval Symposium and International Workshop on Groupware*.

[Chávez y Navarro, 2000] Chávez, E. y Navarro, G. (2000). An effective clustering algorithm to index high dimensional metric spaces. In *IEEE CS Press, editor, 7th International Symposium on String Processing and Information Retrieval*.

[Chávez et al., 2001a] Chávez, E., Navarro, G., Baeza-Yates, R., y Marroquín, J. L. (2001a). Searching in metric spaces. *ACM Computing surveys*.

[Chávez et al., 2001b] Chávez, E., Navarro, G., Baeza-Yates, R., y Marroquín, J. L. (2001b). Searching in metric spaces. *ACM Computation Survey*, 33.

[Chávez González, 1999] Chávez González, E. (1999). *Búsquedas de proximidad en espacios métricos*. PhD thesis, Centro de investigaciones en matematicas.

[Cormen et al., 2001] Cormen, T. H., Leiserson, C. E., Rivest, R. L., y Stein, C. (2001). *Introduction to Algorithms*.

[Culpepper y Listair, 2010] Culpepper, A. S. y Listair, M. (2010). Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems*, 29.

[Dehne y Noltemeier, 1987] Dehne, F. y Noltemeier, H. (1987). Voronoi trees and clustering problems. *Information Systems*, 12.

[González et al., 2005] González, R., Grabowski, S., Mäkinen, V., y Navarro, G. (2005). Practical implementation of rank and select queries. *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*.

[González del Barrio, 2008] González del Barrio, R. R. (2008). *Autoíndices Comprimidos para Texto*. PhD thesis, Universidad de Chile.

[Guttman, 1984] Guttman, A. (1984). R-trees: a dynamic index structure for spatial searching. In *ACM SIGMOD international conference on Management of data*.

[Jacobson, 1989] Jacobson, G. (1989). Space-efficient static trees and graphs. In *Foundations of Computer Science, 30th Annual Symposium*.

[Longley et al., 1999] Longley, P. A., Googchild, M. F., Maguire, D. J., y Rhind, D. W. (1999). *Geographical Information Systems Principles*.

[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 Recognition Letters*, 15.

[Munro et al., 2003] Munro, J. I., Raman, R., Raman, V., y Rao, S. S. (2003). Succinct representations of permutations. *Automata, Languages and Programming Lecture Notes in Computer Science*, 2719.

[Navarro, 1999] Navarro, G. (1999). Searching in metric spaces by spatial approximation. In *IEEE CS Press, String Processing and Information Retrieval (SPIRE)*.

[Robinson, 1981] Robinson, J. T. (1981). The k-d-b-tree: a search structure for large multidimensional dynamic indexes. In *ACM SIGMOD international conference on Management of data*.

[Rotman, 1996] Rotman, J. J. (1996). *A first course in abstract algebra*.

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

[Vidal Ruiz, 1986] Vidal Ruiz, E. (1986). An algorithm for finding nearest neighbours in (approximately) constant average time. *Pattern Recognition Letters*, 4.

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