Universidad Nacional de San Luis
Facultad de Ciencias Físico Matemáticas y Naturales
Departamento de Informática

Bases de Datos Métricas

Nora Susana Reyes

Asesor Científico: Dr. Gonzalo Navarro

San Luis - Argentina

Mayo, 2016

Universidad Nacional de San Luis
Facultad de Ciencias Físico Matemáticas y Naturales
Departamento de Informática

Bases de Datos Métricas

Nora Susana Reyes

Asesor Científico : *Dr. Gonzalo Navarro*
Miembros del Tribunal : *Dr. Fernando Manuel Bulnes (Decano de la Facultad)*
Dra. Agma J. Machado Traina (Universidad de San Pablo, Brasil)
Dr. Benjamín Bustos (Universidad de Chile, Chile)
Dr. Marcelo Errecalde (Universidad Nacional de San Luis)

Tesis para optar al Grado de
Doctor en Ciencias de la Computación

San Luis - Argentina

Mayo, 2016

Agradecimientos

La verdad es que nunca es fácil empezar a escribir los agradecimientos. Uno pasa por la etapa de pensar mucho en el período de tiempo que dedicó al trabajo, que en mi caso fue extenso aunque con pausas, para tratar de rescatar todas aquellas personas o instituciones que de una u otra manera influenciaron el logro del objetivo y no olvidarse de nadie. En el transcurso lo invaden a uno muchas emociones distintas por los recuerdos que se vienen a la mente. Sin embargo, dada la altura de mi vida en la que este trabajo ha tomado lugar, hay dos aspectos que debe tener en cuenta quien quiera leer este agradecimiento: es preponderante para mí la buena gente que me ha acompañado, puedo fácilmente olvidar cosas y ya puedo tener a mi favor la impunidad de la edad.

He tenido la gran ventaja de poder elegir el área en que he investigado, aunque confieso que a veces creo que mi elección se ha debido principalmente a querer seguir trabajando con cierta gente que me hace mucho bien. Mis compañeros académicos son en su mayoría mis amigos. Gracias a la investigación he conocido a personas muy valiosas, tanto humana como académicamente. Buscamos juntos temas para investigar, principalmente para tener la excusa de continuar vinculados.

Además, el área de espacios métricos me ha posibilitado el reencontrarme con algunos compañeros y amigos de universidad, y el conocer a grandes amigos fuera de San Luis, cuya amistad no sólo me reconforta, sino que también me honra. Desde que inicié mis estudios de posgrado, primero la Maestría y ahora el Doctorado, confieso que esta temática me ha sido útil para acompañar a gente muy capaz y a la que aprecio mucho, en la conclusión de sus estudios de grado, para atrevernos también con algunos de ellos a ir más allá y encarar una maestría y principalmente para crecer junto a todos ellos: Aníbal Haissiner, Gabriela Romano, María Edith Di Genaro, Alejandro Gómez, Horacio Pistone, José Rapisarda, Dniel Viale, Juan Carlos Viales, Diego Arroyuelo, Marcelo Barroso, Cristian Bustos, Luis Britos, Natalia Miranda, Fernando Kasián, Manuel Hoffhein (Maha), Mariela Lopresti, Cintia Martínez y Hugo Gercek.

Por suerte, he sentido el apoyo de mucha gente en esta etapa. Algunos me impulsaban a terminar porque consideraban que debía lograr el reconocimiento que implica tener finalmente el título. Otros, porque siempre me apoyaron. Algunos otros, aún sin conocerme demasiado, me hicieron sentir su apoyo al reconocer el trabajo que he venido realizando desde hace años y que para ellos no podía tener otra conclusión lógica que finalizar el doctorado. Más aún, he sentido en gente, a la que no consideraba muy cercana, que sinceramente se alegra que yo logre finalizar mi doctorado. Claramente todo esto me ha hecho mucho bien.

Pasé en este tiempo por muchas pruebas, algunas más complicadas que otras, la dirección del Departamento de Informática por dos períodos, la enfermedad de mis padres, la llegada de mi nieto y luego el fallecimiento de mi Papá (a quien extraño muchísimo). Me fue difícil sobrellevar algunas de ellas y conseguir volver a avanzar con cierto ritmo en mis investigaciones. Pero llegar hasta aquí significa que con paciencia (sobre todo la de mi gran director) se puede.

Obviamente, el pilar más importante en mi vida es, ha sido y será mi familia. Sin embargo, he tenido la gran suerte de tener algunos otros pilares que me apuntalaron que, aunque no eran realmente familia, la vida ha hecho que ya los considere como parte de ella.

Primero quiero agradecer a mis padres, por hacerme creer desde muy chiquita (hace mucho de verdad) que siempre estarían para mí, por enseñarme con sus vidas los principales valores que alguien debe tener, por mostrarme que con esfuerzo las cosas se logran y por incentivarme a cada vez dar más y ser mejor. A mis hermanos, por ser grandes ejemplos de buenas personas, de buenos profesionales en lo suyo y principalmente porque me hacen sentir muy orgullosa de ellos. A mi amiga Patricia Roggero (Pato), que como decía mi papá debía ser mi hermana adoptiva, por su apoyo y por estar para mí y mi familia, como

una más de nosotros.

A Gabriela y a sus padres, una gran compañera y amiga desde el ingreso a la universidad, por la generosidad de hacernos sentir a Pato y a mí, cuando nos despegábamos de nuestras familias para venir a estudiar a San Luis, que su familia podía ser una segunda familia para nosotras. A mis amigos y compañeros de la universidad, en especial a Aníbal. A Carlos Casanova, quien en mis inicios como docente me contagió su placer por enseñar y aprender. A mis compañeras de trabajo, Norma Herrera, Olinda Gagliardi, Verónica Ludueña y María Teresa Taranilla, porque hemos compartido muchas cosas, porque fuimos creciendo juntas y con el tiempo hemos hecho crecer a nuestra área. A docentes-investigadores y amigos con quienes trabajo en la UNSL, Marcela Printista, Fabiana Piccoli, Roberto Guerrero, Marcelo Errecalde, a quienes respeto mucho y con quienes soñamos una universidad mejor. A Claudia Deco por permitirme trabajar con ella, y sus alumnos, en la Universidad Nacional de Rosario. A Verónica Gil-Costa, con quien hemos coincidido muchas veces en trabajos conjuntos.

A mis amigos académicos fuera de la UNSL: Gonzalo Navarro, Edgar Chávez, Rodrigo Paredes, Karina Figueroa, Nieves Brisaboa, Antonio Fariña, Benjamín Bustos, Roberto Uribe (Tito) y Óscar Pedreira, tanto por los momentos en que me permitieron investigar con ellos, como por los momentos también muy importantes que pudimos compartir fuera de la universidad. También a los padres de Karina (Leonila y Adán) por recibirme con tanto afecto y atenciones en mi visita a México, a la mamá de Rodrigo (Isa) por darme lugar en su casa en mis últimos viajes a Santiago y a Carme Fernández por hacerme pasar gratos y muy divertidos momentos en mis visitas a Coruña. Además debo agradecer a Edgar, Gonzalo, Karina y Rodrigo por facilitarme además acceso a sus servidores para realizar muchos de los experimentos de esta tesis. A Tito, Eric Sadit Téllez y Maha por ayudarme a entender y ejecutar algunos códigos de otras estructuras contra las que debía comparar mis aportes.

A Dolores Rexachs, porque aún sin ser conciente del efecto que produjeron en mí sus sinceras palabras de apoyo, estuvieron allí justo cuando necesitaba escucharlas: hay momentos en que nos viene bien que alguien levante nuestra autoestima para convencernos que podemos seguir avanzando.

Cuando inicié mi tesis de maestría tuve la suerte que, "por el tema que elegí", mi director resultara ser Gonzalo. Ese mismo día me presentó a su esposa Betina Giordano, quien desde el comienzo me brindó cálidamente un lugarcito junto a su familia, cuando la mía estaba lejos. Gonzalo es un lujo de director, porque además de su alto nivel académico es un gran ser humano y un amigo. No sólo hace que uno dé más de lo que uno mismo se cree que es capaz de dar, sino que tiene la paciencia para lograrlo, que en mi caso fue mucha. Agradezco también a Ricardo Baeza-Yates por hacer posible este encuentro director-alumno.

También, gracias a mi maestría, conocí a otro grande que me acompaña: Edgar. Desde que empezamos a trabajar juntos fuimos construyendo una hermosa amistad que nos permite acompañarnos mutuamente a la distancia. Durante todos estos años me ha hecho sentir que está para mí literalmente siempre, dispuesto a ayudarme, con la palabra justa cuando flaqueo, con la explicación que necesito; tanto es así que se tomó el trabajo de estar presente en mi defensa, sólo para acompañarme. Es otro de los lujos que he podido darme, tener su amistad y acceso a otra de las personas que más sabe en este tema.

Como dije, todo esto no hubiera sido posible sin el apoyo incondicional de mi familia. Mi marido, mis hijos, y ahora mi nieto, han vivido este largo proceso junto conmigo y me han permitido, aún a pesar de perder algunos momentos juntos, que yo dedique tiempo a esto y disfrute sin culpa de mi trabajo.

El que lea estos agradecimientos se dará cuenta de que muchas veces uso la palabra "suerte" y es porque realmente me siento muy afortunada por llegar hasta donde estoy llegando y con quienes lo estoy haciendo.

Agradezco a la Universidad Nacional de San Luis y a quienes fueron mis profesores, por la formación que me han dado. Tengo además la oportunidad de devolver algo de todo eso siendo docente e investigador en esta casa.

Además, quiero agradecer a Cecilia Ramos, artista plástica colombiana, por enviarme sus buenos deseos y versiones de mejor calidad de sus ilustraciones, cuando la contacté para pedirle autorización para usarlas en mi informe de tesis (las que aparecen en las conclusiones).

Finalmente agradezco a Marco Patella por haber aceptado ser revisor de mi plan de tesis y a Benjamín Bustos, Agma J. M. Traina y Marcelo Errecalde por formar parte del tribunal y permitir, con sus aportes y sugerencias, mejorar mi informe.

Nora

San Luis, 27 de mayo de 2016

A la mejor mamá del mundo: la mía

A mi papá, a quien extraño demasiado

A mis amores: Norita, Rocío, Sebastián, Tomás y Guillermo

Índice general

1. Introducción **1**

1.1. Objetivos . . . 3

1.2. Contribuciones de la Tesis . . . 3

1.2.1. Optimización de Índices Estáticos . . . 4

1.2.2. Dinamismo y Grandes Volúmenes de Datos . . . 4

1.2.3. Otras Operaciones . . . 5

1.3. Publicaciones Generadas . . . 6

2. Conceptos Básicos **9**

2.1. El Modelo de Espacios Métricos . . . 9

2.1.1. Funciones de Distancia . . . 11

2.2. Consultas por Similitud o Proximidad . . . 14

2.2.1. Consultas por Rango . . . 14

2.2.2. Consultas de k Vecinos Más Cercanos . . . 15

2.2.3. Otras Consultas por Similitud . . . 16

2.3. Estrategias de Búsqueda en Espacios Métricos . . . 18

2.4. Algoritmos de Indexación . . . 19

2.4.1. Estrategias de Indexación . . . 20

2.4.2. Almacenamiento . . . 29

2.4.3. Tipo de Respuesta . . . 31

2.4.4. Capacidades Dinámicas . . . 31

2.5. Clasificación de las Soluciones Existentes . . . 32

2.6. Técnicas de Búsqueda de los Vecinos más Cercanos . . . 33

2.6.1. Radio Incremental . . . 34

2.6.2. Backtracking con Radio Decreciente . . . 34

2.6.3. Backtracking con Prioridad . . . 35

2.7. Espacios Vectoriales . . . 35
2.8. Dimensionalidad y Dimensionalidad Intrínseca . . . 36
2.9. Otros Ejemplos de Espacios Métricos . . . 37
2.9.1. Modelo Vectorial para Documentos . . . 37
2.9.2. Diccionarios . . . 38
2.10. Espacios Métricos Utilizados en la Evaluación Experimental . . . 39
2.11. Detalles Generales de los Experimentos . . . 40

3. Estructuras Previas **41**
3.1. Árbol de Aproximación Espacial . . . 41
3.2. Acercamiento a la Aproximación Espacial . . . 41
3.3. El Árbol de Aproximación Espacial . . . 44
3.3.1. Proceso de Construcción . . . 44
3.3.2. Búsquedas . . . 45
3.3.3. Búsqueda de Vecinos más Cercanos . . . 47
3.4. Análisis . . . 49
3.5. Árbol de Aproximación Espacial Dinámico . . . 50
3.5.1. Inserciones y Búsquedas . . . 51
3.5.2. Eliminaciones . . . 55
3.5.3. Reconstrucción de Subárboles . . . 57
3.5.4. Usando Nodos Ficticios . . . 59
3.5.5. Hiperplanos Fantasmas . . . 60
3.6. Lista de Clusters . . . 62
3.6.1. Proceso de Construcción . . . 63
3.7. Búsquedas . . . 65
3.7.1. Lista de Clusters Recursiva . . . 66

4. Optimización de Índices **67**
4.1. Optimización al *SAT* . . . 67
4.1.1. Motivación . . . 68
4.2. Árbol de Aproximación Espacial Distal . . . 69
4.2.1. La Estrategia SAT^{+} . . . 72
4.2.2. La Estrategia SAT^{Glob} . . . 72
4.2.3. La Estrategia SAT^{Out} . . . 75

4.3. Resultados Experimentales . . . 76
4.3.1. Eligiendo la Raíz del Árbol . . . 77
4.3.2. Restringiendo la Aridad Máxima . . . 77
4.3.3. Diferentes Órdenes de Inserción . . . 77
4.3.4. Dependencia de la Dimensionalidad Intrínseca . . . 78
4.3.5. Escalabilidad . . . 80
4.4. Hiperplanos vs. Radios de Cobertura . . . 80
4.5. Comparación con otros Índices . . . 81
4.5.1. M–tree . . . 82
4.5.2. Pivotes . . . 84
4.5.3. Bisector–Tree . . . 85
4.5.4. Lista de Clusters . . . 86
4.5.5. Vantage-Point Tree . . . 88
4.5.6. Geometric Near-Neighbor Access Tree . . . 89
4.5.7. Aceleración . . . 91
4.6. Optimización a la *LC* . . . 95
4.6.1. Distancias entre Centros . . . 95
4.6.2. Objetos Candidatos . . . 99
4.6.3. Búsqueda de k Vecinos Más Cercanos . . . 101

5. Índices Dinámicos para Memoria Secundaria **104**
5.1. Conceptos Previos . . . 104
5.2. Configuración de los Experimentos . . . 106
5.3. Árbol de Aproximación Espacial Dinámico en Memoria Secundaria . . . 107
5.3.1. Diseño de la Estructura de Datos . . . 107
5.3.2. Inserciones . . . 108
5.3.3. Manejo de Rebalse de Página . . . 108
5.3.4. Asegurando el 50 % de Ocupación . . . 111
5.3.5. Búsquedas . . . 111
5.4. Una Variante Heurística: *DSAT+* . . . 112
5.4.1. Sintonización Experimental . . . 112
5.5. Lista de Clusters Dinámica en Memoria Secundaria . . . 114
5.5.1. Inserciones . . . 117

5.5.2. Políticas de División . . . 118
5.5.3. Búsquedas . . . 119
5.5.4. Eliminaciones . . . 119
5.5.5. Sintonización Experimental . . . 120
5.6. Conjunto Dinámico de Clusters en Memoria Secundaria . . . 122
5.6.1. Diseño de la Estructura de Datos . . . 123
5.6.2. Inserciones . . . 124
5.6.3. Búsquedas . . . 125
5.6.4. Eliminaciones . . . 125
5.6.5. Sintonización Experimental . . . 126
5.7. Comparación Experimental . . . 128
5.7.1. Comparando con el *M–tree* . . . 128
5.7.2. Eliminaciones . . . 130
5.7.3. Estudio de *DSC* sobre Espacios Masivos . . . 133
5.7.4. Comparación con los Nuevos Índices . . . 142
5.7.5. Escalabilidad . . . 143
5.8. Análisis Final de las Propuestas . . . 144

6. Índices para Bases de Datos Métricas **146**
6.1. Conceptos Previos . . . 147
6.2. Lista de Clusters Gemelos . . . 148
6.2.1. Resolviendo Consultas por Rango con *LTC* . . . 149
6.2.2. Construcción de *LTC* . . . 152
6.3. Consultas con *LTC* . . . 154
6.3.1. Join por Rango . . . 154
6.3.2. Join de k Pares de Vecinos Más Cercanos . . . 154
6.3.3. Consultas por Rango . . . 157
6.4. Evaluación Experimental . . . 157
6.4.1. Construcción de *LTC* . . . 159
6.4.2. Joins por Rango . . . 159
6.4.3. Join de k Pares Más Cercanos . . . 164
6.4.4. Consultas por Rango . . . 165

7. Conclusiones **167**

7.1. Aportes 168
7.2. Trabajos Futuros 170

Índice de figuras

2.1. Interpretación gráfica en $\mathbb{R}^2$ de la equidistancia entre puntos, de acuerdo a las distintas funciones de distancia de Minkowski ([ZADB06, Fig. 1.1]). 12

2.2. Consultas por rango, partiendo desde un q dado, con radio r. 15

2.3. Consulta de los k-vecinos más cercanos, partiendo desde un q dado y un valor de $k = 5$. 16

2.4. Diferencia entre las variantes de la operación de join por similitud ([BK04, Fig. 1]). . . . 18

2.5. Zonas para el pivote p_i respecto de una consulta (q, r). 22

2.6. Zona no descartada conjunta para el conjunto de pivotes $\mathbb{P} = \{p_1, p_2\}$. 23

2.7. Zona de no filtrado para un conjunto de tres pivotes, con dos pivotes similares entre sí. . 23

2.8. Otra zona de no filtrado para un conjunto de tres pivotes, con pivotes no similares entre sí. 24

2.9. Partición de un espacio en $\mathbb{R}^2$, desde un centro c con radio m. 25

2.10. Situación de una búsqueda (q, r) que no debe examinar la zona con centro c y radio de cobertura r_c. 25

2.11. Partición por hiperplano de un espacio en $\mathbb{R}^2$, respecto de dos centros. 27

2.12. Situación de una búsqueda (q, r) que no debe examinar la zona correspondiente al centro p_1, de acuerdo al criterio de hiperplano. 27

2.13. Criterios utilizados para particiones compactas. 28

2.14. Descarte en una búsqueda por rango (q, r) de acuerdo a los distintos criterios. 30

2.15. Taxonomía de los algoritmos existentes para búsqueda por similitud en espacios métricos. 33

2.16. Un histograma de distancias para un espacio métrico de dimensión baja (izquierda) y de dimensión alta (derecha). 37

2.17. Histogramas de distancia de los espacios métricos descargados de SISAP. 40

3.1. Un ejemplo del proceso de búsqueda con un grafo de Delaunay (arcos sólidos) correspondiente a una partición de Voronoi (áreas delimitadas por líneas punteadas). 43

3.2. Ilustración del Teorema 1. 44

3.3. Un ejemplo del proceso de búsqueda. 47

3.4. Políticas de selección de radios para *LC*. 63

3.5. Ejemplo de construcción de *LC* para tres centros c_1, c_2 y c_3 tomados en ese orden, y la lista resultante. 64

3.6. Posibles situaciones de una búsqueda por rango (q, r) en *LC*. 65

3.7. Construcción de la Lista de Clusters Recursiva. 66

4.1. Ejemplo de una base de datos métrica en $\mathbb{R}^2$. 68

4.2. Ejemplo de *SAT* obtenido al elegir a p_{12} como raíz. 68

4.3. Ejemplo de *SAT* obtenido al elegir a p_6 como raíz. 69

4.4. Ejemplo del *DSAT* obtenido con aridad máxima de 2. 70

4.5. Ejemplo del *DSAT* obtenido con aridad máxima de 6. 70

4.6. División del espacio al construir el *SAT* con raíz p_6 y con distintos primeros vecinos. . . 71

4.7. Ejemplo del *SAT*$^+$ seleccionando como raíz a p_{12}. 73

4.8. Ejemplo del *SAT*$^+$ al seleccionar a p_6 como la raíz del árbol. 73

4.9. Ejemplo del *SAT*^{Glob} seleccionando a p_{12} como raíz. 74

4.10. Ejemplo del *SAT*^{Glob} con p_6 seleccionado como raíz. 75

4.11. Ejemplo de *SAT*^{Out} con p_3 como la raíz del árbol. 76

4.12. Ejemplo de *SAT*^{Out} con p_8 como la raíz. 76

4.13. Comparación de los costos de búsqueda por rango entre el *SAT*, el *DSAT*, las mejores variantes del *DiSAT* y la versión del *SAT* que usa el algoritmo CSA para seleccionar la raíz. 78

4.14. Comparación de los costos de construcción, para el *SAT*, *DSAT* y la versión de *SAT* con aridad acotada. 79

4.15. Comparación de los costos de las búsquedas por rango, para *SAT*, *DSAT* y *SAT* con aridad acotada. 80

4.16. Comparación de los costos de construcción para las distintas variantes del *DiSAT*. 81

4.17. Comparación de costos de búsqueda por rango para las distintas variantes del *DiSAT*. . . 82

4.18. Comparación de costos de construcción del *DiSAT* a medida que crece la dimensión. . . 83

4.19. Comparación de los costos de búsqueda del *DiSAT* a medida que crece la dimensión. . . 83

4.20. Comparación de costos del *DiSAT* para la base de datos *Flickr*. 84

4.21. Comparación de costos de construcción entre el *DiSAT* y el *M–tree*. 86

4.22. Comparación de los costos de búsqueda entre el *DiSAT* y el *M–tree*. 87

4.23. Comparación de los costos de búsqueda del *DiSAT* con un algoritmo genérico de pivotes. 88

4.24. Comparación de costos de construcción del *DiSAT* con el *BST*. 89

4.25. Comparación de costos de búsqueda del *DiSAT* con el *BST*. 90

4.26. Comparación de los costos de construcción de *DiSAT* con la *LC*. 91

4.27. Comparación de los costos de búsqueda de *DiSAT* con la *LC*. 92
4.28. Comparación de costos de construcción del *DiSAT* con el *VPT*. 93
4.29. Comparación de costos de búsqueda del *DiSAT* con el *VPT*. 94
4.30. Comparación de costos de construcción entre el *DiSAT* y el *GNAT*. 95
4.31. Comparación de costos de búsqueda entre el *DiSAT* y el *GNAT*. 96
4.32. Comparación del speedup de las consultas y tiempo de construcción, a medida que crece la dimensión. 97
4.33. Comparación del speedup de las consultas y tiempo de construcción para un millón de elementos, a medida que la dimensión crece. 97
4.34. Comparación para la base de datos de vectores en dimensión 12, a medida que crece la cardinalidad de la base de datos. 98
4.35. Comparación de *DiSAT* con otros índices para la base de datos *Flickr*. 98
4.36. Comparación de costos de construcción para subconjuntos crecientes de la base de datos *CoPhIR*. 99
4.37. Comparación de costos de búsqueda en subconjuntos crecientes de la base de datos *CoPhIR*. 99
4.38. Comparación de costos de búsqueda entre *LC* y la versión que almacena la matriz de distancias entre centros *LC-DC*. 101

5.1. Estructura básica de un disco. 105
5.2. Ejemplo del diseño de los punteros $F(a)$ y $S(a)$ en un nodo. 108
5.3. Ejemplo antes y después de aplicar la política de ***traslado al padre***. 109
5.4. Ejemplo antes y después de aplicar la política de ***división vertical***. 110
5.5. Ejemplo antes y después de la aplicación de la política de ***división horizontal***. 111
5.6. Costos de búsqueda de las variantes para memoria secundaria del *DSAT*, en términos de evaluaciones de distancia (izquierda) y páginas de disco leídas (derecha). 113
5.7. Costos de construcción de las variantes del *DSAT* en memoria secundaria, en evaluaciones de distancia (izquierda) y operaciones de E/S (derecha). La tabla muestra la ocupación de espacio. 115
5.8. Ejemplo de una *DLC* en $\mathbb{R}^2$. 116
5.9. Detalles de los datos almacenados en memoria principal para la *DLC* de la Figura 5.8. . . 116
5.10. La posible reducción de una zona de cluster por la inserción de un nuevo elemento x. . . 117
5.11. Costos de búsqueda de las variantes de *DLC* en memoria secundaria, en evaluaciones de distancia (izquierda) y número de páginas de disco leídas (derecha). 121
5.12. Costo de construcción de las variantes en memoria secundaria de *DLC* en evaluaciones de distancia. La tabla muestra la ocupación de espacio. 122
5.13. Ejemplo de un *DSC* en $\mathbb{R}^2$. 123

5.14. Detalles del *DSAT* en memoria principal de la Figura 5.13. 124

5.15. Costos de búsqueda para las variantes de *DSC* en memoria secundaria, en términos de evaluaciones de distancia (izquierda) y páginas de disco leídas (derecha). 127

5.16. Costo de construcción de las variantes de *DSC* en memoria secundaria, en evaluaciones de distancia. La tabla muestra la ocupación de espacio. 128

5.17. Comparación de los costos de búsqueda de *DSC*, *DLC*, *DSAT+* y *M–tree*, considerando evaluaciones de distancia (izquierda) y cantidad de páginas leídas (derecha). 131

5.18. Costos de construcción de *DSC*, *DLC*, *DSAT+* y *M–tree*, considerando evaluaciones de distancia (izquierda) y operaciones de E/S (derecha). 132

5.19. Costos de eliminación para las mejores alternativas de *DSC*, considerando evaluaciones de distancia (izquierda) y operaciones de E/S (derecha). 133

5.20. Comparación de los costos de búsqueda a medida que la fracción de elementos eliminados de la base de datos aumenta, para las mejores alternativas de *DSC*. 134

5.21. Costos de inserción y búsqueda para las mejores alternativas de *DSC* sobre el espacio de Vectores en dimensiones 10 (izquierda) y 15 (derecha). 135

5.22. Costos de inserción y búsqueda para las mejores alternativas de *DSC* sobre el espacio de Vectores en dimensión 10 (izquierda) y 15 (derecha), considerando el tamaño de página de disco. 136

5.23. Costos de eliminación para la mejor variante de *DSC* sobre el espacio de Vectores, en dimensión 10 y 15. 137

5.24. Comparación de los costos de búsqueda a medida que crece el número de eliminaciones, para las mejores alternativas de *DSC*, para Vectores en dimensión 10. 138

5.25. Comparación de los costos de búsqueda a medida que crece el número de eliminaciones, para las mejores alternativas de *DSC*, para Vectores en dimensión 15. 139

5.26. Costos de inserción y búsqueda para las mejores alternativas de *DSC* sobre el espacio de *Flickr*. A la izquierda, como una función de la política usada. A la derecha, como una función del tamaño de página de disco. 140

5.27. Costos de eliminación para la mejor variante de *DSC* sobre *Flickr*. 141

5.28. Comparación de los costos de búsqueda a medida que crece el número de eliminaciones, para las mejores alternativas de *DSC* en el espacio de *Flickr*. 141

5.29. Comparación entre *DSC*, *eGNAT* y *MX–tree* sobre el espacio de *Flickr*. 142

5.30. Comparación de los costos de inserción del *DSAT+* y la mejor alternativa del *DSC* sobre los espacios de 10.000.000 de vectores de dimensiones 10 y 15. 143

5.31. Comparación de los costos de búsqueda para el *DSAT+* y la mejor política del *DSC* sobre los espacios de 10.000.000 de vectores de dimensiones 10 y 15. 144

5.32. Una comparación gráfica gruesa de los costos de inserción y los costos de búsqueda de las diferentes estructuras. 145

6.1. Clusters gemelos que se superponen entre ellos. 149

6.2. Resolución de la consulta para el centro de cluster c_a^i, usando las distancias almacenadas en *LTC*. 149

6.3. Construcción de *LTC* variando el radio de los clusters. 160

6.4. Comparación entre los diferentes radios de clusters considerados para la construcción del índice *LTC*. Notar las escalas logarítmicas. 162

6.5. Comparación entre todos los algoritmos de join por rango considerados, usando en cada caso el mejor radio de construcción determinado experimentalmente para el índice. Notar las escalas logarítmicas. 163

6.6. Comparación entre todos los algoritmos de búsqueda por rango considerados, usando un radio de join mayor que el radio de indexación; es decir $R < r$. 163

6.7. Comparación entre el join de k pares más cercanos y el join por rango equivalente. Notar las escalas logarítmicas. 165

6.8. Cálculo de las consultas por rango sobre *LTC* y con una o dos *LC*, variando los radios. . 166

Índice de tablas

2.1. Nombres de las estructuras de datos para búsquedas por similitud exactas en espacios métricos, las referencias a los artículos en donde se propone cada una de ellas y sus caracterśticas de capacidades dinámicas y de almacenamiento. 34

4.1. Comparación experimental de los costos de búsqueda de las nuevas heurísticas, medidos en evaluaciones de distancia, analizando la poda por hiperplanos y por radios de cobertura. 85

5.1. Utilización de espacio en disco de los índices seleccionados. 129

5.2. Uso de espacio para el espacio de Vectores, con diferentes tamaños de página. 137

5.3. Uso de espacio en disco para el conjunto de datos de *Flickr*, con diferentes tamaños de página. 139

6.1. Fracción del total del costo del join realizado por centros, objetos regulares y objetos no indexados. 164

6.2. Proporción del desempeño del JoinRango-LTC para todos los pares de bases de datos, en todos los radios usados, con respecto a los otros métodos de join por rango. 164

Índice de algoritmos

1. Proceso para construir un *SAT*. 46
2. Rutina para buscar q con radio r en un *SAT*. 48
3. Rutina para buscar los k vecinos más cercanos a q en un *SAT*. 49
4. Inserción de un nuevo elemento x en un *DSAT* cuya raíz es a, para la técnica de timestamp. 52
5. Búsqueda de q con radio r en un *DSAT* de raíz a, usando la técnica de timestamp. 52
6. Búsqueda de los k vecinos más cercanos de q en un *DSAT* con raíz a, usando timestamps. 53
7. Proceso de inserción de un nuevo elemento x en un *DSAT* cuya raíz es a, usando aridad acotada. 54
8. Búsqueda de q con radio r en un *DSAT* con raíz a, con aridad limitada. 54
9. Rutina para buscar los k vecinos más cercanos a q en un *DSAT* con aridad acotada. . . . 55
10. Inserción de un nuevo elemento x en un *DSAT* con raíz a, usando timestamp + aridad acotada. 56
11. Búsqueda de q con radio r en un *DSAT* con raíz a, usando timestamp + aridad acotada. . 56
12. Búsqueda de los k vecinos más cercanos de q en un *DSAT* con raíz a, usando timestamp + aridad acotada. 56
13. Proceso para eliminar x desde un *DSAT*, por reconstrucción de subárboles. 58
14. Proceso de búsqueda por rango para q with radius r en un *DSAT* con raíz a, modificado para considerar los hiperplanos fantasmas. 61
15. Proceso para eliminar a x desde un *DSAT*, usando hiperplanos fantasmas y encontrando un sustituto para x entre las hojas de su subárbol. 61
16. Proceso para eliminar x desde un *DSAT*, usando hiperplanos fantasmas, y eligiendo su reemplazo entre sus vecinos. 62
17. Eliminación de x desde un *DSAT*, usando hiperplanos fantasmas, y eligiendo su reemplazo como su elemento más cercano en su subárbol. 62
18. Proceso de construcción genérico de la *Lista de Clusters*. 64
19. Proceso de la búsqueda por rango en *LC*. 65
20. Proceso para construir un SAT^{+} con raíz a. 72
21. Proceso de construcción de un SAT^{Glob} con raíz a. 74
22. Proceso para seleccionar la raíz de un SAT^{Out}. 75
23. Proceso de la búsqueda por rango en *LC* usando las distancias almacenadas entre centros. 100
24. Proceso de la búsqueda por rango en *LC*, usando distancias almacenadas entre el centro y los objetos de cada cluster. 100
25. Proceso de búsqueda de k vecinos más cercanos en *LC*. 102
26. Consulta por rango para los centros de clusters. 150
27. Consulta por rango para objetos regulares. 151

28. Consulta por rango para objetos regulares. 152
29. Construcción de la *LTC*. 153
30. Cálculo del join por rango con *LTC*. 154
31. Rutina auxiliar para el cálculo del join de k pares más cercanos con *LTC*. 155
32. Cálculo del join de k pares más cercanos con *LTC*. 156
33. Uso de *LTC* para calcular consultas por rango. 158
34. Join por rango usando dos *Listas de Clusters*. 161

Capítulo 1

Introducción

En la actualidad es cada vez más evidente la necesidad de procesar conjuntos de datos, de manera tal de poder obtener información útil a partir de ellos. Sin embargo, las bases de datos en una computadora comenzaron a existir en la década de 1960, cuando se pudieron empezar a usar computadoras en las organizaciones, por ser una opción más efectiva en costo. Los primeros modelos de datos en esa década fueron un *modelo de red* (CODASYL) y un *modelo jerárquico* (IMS). En 1970 Edgar F. Codd publicó un importante trabajo que proponía el uso de un modelo de base de datos relacional [Cod70], y sus ideas cambiaron la manera de pensar sobre las bases de datos. En este modelo, el esquema de la base de datos, o la organización lógica, no está conectada con el almacenamiento físico, y se convirtió en el principio estándar para los sistemas de bases de datos. Este modelo de base de datos es el más utilizado en la actualidad para *bases de datos tradicionales*, donde siguiendo el modelo de Codd, es posible estructurar los datos en n-uplas, para luego hacer búsquedas exactas (por igualdad) respecto de valores de campos o atributos en las n-uplas. En muchas aplicaciones que realizan procesamiento de datos este modelo ha resultado ser muy exitoso, por su capacidad de modelar correctamente los conjuntos de datos de la realidad.

Sin embargo, las décadas pasadas han sido testigos de una alarmante velocidad de crecimiento de los datos disponibles en forma digital, así como un paralelo crecimiento de las capacidades de almacenamiento utilizables a precios moderados. Además, con la de las tecnologías de información y comunicación, han surgido depósitos no estructurados de información. No sólo se consultan nuevos tipos de datos tales como texto libre, imágenes, audio y video, sino que además ya no se puede estructurar más la información en atributos y n-uplas. Tal estructuración es muy dificultosa (tanto manual como computacionalmente) y restringe de antemano los tipos de consultas que luego se pueden hacer. Aún cuando sea posible una estructuración clásica, nuevas aplicaciones tales como minería de datos (*data mining*) requieren acceder a la base de datos por cualquier atributo, no sólo aquéllos marcados como "claves". Así, la recuperación desde estos repositorios requiere lenguajes de consulta más poderosos, los cuales exceden las capacidades de la tecnología tradicional de bases de datos. Por lo tanto, son necesarios nuevos modelos más generales de base de datos que permitan administrar y buscar en depósitos o almacenamientos de datos "no estructurados". Entre los nuevos tipos de datos se pueden mencionar huellas digitales, secuencias de audio, imágenes, video, modelos 3D, secuencias biológicas (ADN), etc. A las bases de datos capaces de almacenar este tipo de información se las denomina *bases de datos no tradicionales*.

Recientemente, los Sistemas Administradores de Bases de Datos (DBMS) incorporan la capacidad de almacenar nuevos tipos de datos tales como imágenes u "objetos multimedia", sin embargo la búsqueda se realiza todavía sobre un número predeterminado de claves de tipos numérico o alfabético y muy

raramente se los puede buscar por contenido. Un concepto unificador es el de *bases de datos métricas* que utiliza el concepto de "búsqueda por similitud" o "búsqueda por proximidad", es decir buscar elementos de la base de datos que sean similares o "próximos" a un elemento de consulta dado. Una *base de datos métrica* es una colección de objetos digitales (de cualquier clase) con una similitud *percibida* y una manera formal de calcular esta similitud percibida como una métrica. La similitud percibida es provista por expertos. Así, las bases de datos métricas, usando el modelo de espacios métricos, dan una base teórica para definir una manera significativa de recuperar datos multimedia.

Las aplicaciones de las bases de datos métricas van más allá de la recuperación multimedia y ello ha producido que, como los problemas han aparecido en áreas muy diversas, las soluciones también hayan surgido desde muchos campos no relacionados. Algunos ejemplos son bases de datos no tradicionales (donde no se usa el concepto de búsqueda exacta y se buscan objetos similares, v.g. bases de datos de imágenes, huellas digitales o clips de audio); aprendizaje de máquina y clasificación (donde se debe clasificar un nuevo elemento de acuerdo a su elemento más cercano existente); cuantización y compresión (donde sólo algunos vectores pueden ser representados y los demás deben ser codificados como su punto representativo más cercano); recuperación de texto (donde se buscan palabras en una base de datos de texto permitiendo un pequeño número de errores, o se buscan documentos que sean similares a una consulta o documento dado); biología computacional (donde se quiere encontrar una secuencia de proteína o ADN en una base de datos, permitiendo algunos errores debidos a variaciones típicas); predicción de funciones (donde se quiere buscar el comportamiento más similar de una función en el pasado, para predecir su probable comportamiento futuro); etc..

En algunas aplicaciones el espacio métrico es realmente un *espacio vectorial de dimensión finita* D, donde los elementos son vectores de coordenadas reales, es decir elementos de $\mathbb{R}^D$. Alrededor de este modelo de datos se han realizado muchos desarrollos, tanto de índices y operaciones, como de lenguajes de consulta. Sin embargo, generalmente estos desarrollos no pueden ser extendidos a las bases de datos métricas, donde sólo se dispone de la información de distancia entre los objetos [CNBYM01]. Más aún, a pesar de que recientemente al modelo de espacios métricos se le está dedicando cada vez más atención, se puede decir que aún no alcanza el nivel de desarrollo que ya poseen los espacios vectoriales [Sam05].

El área que más se ha desarrollado dentro de las bases de datos métricas ha sido el de las búsquedas por similitud [CNBYM01, ZADB06, Sam05]. Más aún, la mayoría de las soluciones propuestas para resolver las búsquedas por similitud considera el escenario simple en el que la base de datos se conoce completamente de antemano y por consiguiente se puede preprocesar para construir un índice. Por lo tanto, dicho índice no admite ni inserciones ni eliminaciones de elementos. Además, buscan la eficiencia sólo considerando mejorar los tiempos involucrados en los cálculos de distancia y algunas veces en los tiempos extras de CPU, por suponer que dichos índices y posiblemente también la base de datos, se almacena en memoria principal. Sin embargo, para poder considerar a las bases de datos métricas como un nuevo modelo de bases de datos no basta con que existan sólo soluciones al problema de búsqueda por similitud para conjuntos de datos cuya cardinalidad esté limitada por la capacidad de la memoria principal disponible, y más grave aún, que todos los elementos de la base de datos deban ser conocidos con anticipación.

Así, pensando en aplicaciones reales cada vez más demandantes de soluciones eficientes, el modelo de bases de datos métricas debe considerar:

- Dinamismo: los elementos pueden incorporarse a la base de datos en cualquier momento y se debe posibilitar la creación incremental de los índices. Además, en muchas aplicaciones puede ser indispensable que el índice sea capaz de eliminar elementos físicamente. Esto es particularmente problemático en un escenario donde los objetos podrían ser muy grandes (v.g. imágenes satelitales).

- Grandes volúmenes de datos: en muchos casos de interés los objetos son demasiado grandes y ellos deben alojarse en disco, o los objetos son tantos que el índice mismo no cabe en memoria principal.
- Otras operaciones: aún cuando se logren soluciones para buscar eficientemente en grandes bases de datos, que permitan que el índice que se construya admita que los elementos pueden insertarse o eliminarse en cualquier momento y que logren la eficiencia considerando que el almacenamiento del mismo será en memoria secundaria, en un modelo de bases de datos métricas se deben considerar otras operaciones, comunes en otros modelos de base de datos. En particular, en este modelo de base de datos métricas una operación que debería considerarse como una primitiva es el *join por similitud*.
- Mecanismos de control: considerando que en un sistema de base de datos diferentes usuarios tienen acceso a la base de datos al mismo tiempo, se deben definir mecanismos de control que permitan que cada usuario esté protegido de los otros. Por lo tanto, se debería considerar cómo asegurar la consistencia y seguridad de las transacciones sobre la base de datos, mientras se permite que varios usuarios puedan acceder concurrentemente a la misma.

1.1. Objetivos

En este trabajo el objetivo es atacar los tres primeros aspectos mencionados: dinamismo, grandes volúmenes de datos y otras operaciones; dejando el cuarto, mecanismos de control, como trabajo futuro. Más aún, el foco se ha puesto en considerar particularmente tres subproblemas: la optimización de estructuras existentes, las búsquedas por similitud en memoria jerárquica y el operador de join por similitud sobre bases de datos métricas. Así, en esta propuesta se busca contribuir a estas líneas de investigación proponiendo el diseño de estructuras de datos eficientes para resolver las operaciones de interés, dinámicas y conscientes de la existencia de una jerarquía de memoria, logrando así un modelo más completo para bases de datos métricas.

Para el desarrollo de este trabajo se consideraron las siguientes hipótesis:

a) Es posible mejorar la eficiciencia de las soluciones actuales para búsqueda por similitud en memoria jerárquica, ya sea en términos de tiempo, número de evaluaciones de la función de distancia, cantidad de operaciones de E/S, espacio, etc.

b) Es posible mejorar la eficiciencia de las soluciones actuales para el join por similitud, ya sea en términos de tiempo, número de evaluaciones de la función de distancia, cantidad de operaciones de E/S, espacio, etc.

Por lo tanto, se considera que esta tesis es una contribución al desarrollo de un modelo más maduro de bases de datos métricas.

1.2. Contribuciones de la Tesis

Existen numerosos métodos de indexación para búsquedas por similitud en espacios métricos [Sam05, ZADB06, CNBYM01, Het09]. Actualmente se encuentra disponible una biblioteca para espacios métricos [FNC07], que contiene la descripción de la misma, códigos fuentes de algunas de las estructuras y conjuntos de datos métricos a utilizarse como “benchmark”, para distintos tipos de datos.

1.2.1. Optimización de Índices Estáticos

Como se mencionó, la mayoría de las soluciones existentes son estáticas: una vez construido el índice para una base de datos dada, agregar nuevos elementos a la base de datos o eliminar un elemento desde ella, requiere de actualizaciones muy costosas sobre el índice. A pesar de ello, hay índices en los que parece no haberse aprovechado al máximo la ventaja de conocer de antemano todos los elementos de la base de datos y sobre los cuales si se realiza un análisis detallado de sus características más importantes, pueden aún optimizarse para que logren un mejor desempeño en las búsquedas, mientras mantienen las bondades del diseño original.

Por ejemplo, el *Árbol de Aproximación Espacial* (*SAT* por su sigla en inglés) [Nav02] es un índice estático, que obtiene un atractivo balance entre uso de memoria, tiempo de construcción y desempeño de búsqueda. Su versión dinámica, el *Árbol de Aproximación Espacial Dinámico* (*DSAT* por su sigla en inglés) [NR08], puede construirse por inserciones sucesivas con menor costo respecto del *SAT* (sin conocer de antemano todos los elementos de la base de datos), permite inserciones y eliminaciones y aún así supera el desempeño del *SAT* en las búsquedas.

Por otra parte, la *Lista de Clusters* (*LC*) [CN05] es también un índice estático, muy costoso de construir, con un muy buen desempeño en las búsquedas sobre espacios considerados difíciles. Sin embargo, la *LC* no es capaz de mejorar las búsquedas aprovechando la gran mayoría de los cálculos de distancia realizados durante la construcción, ni tampoco la disponibilidad de espacio adicional para el índice.

Se han realizado optimizaciones al *Árbol de Aproximación Espacial* [Nav02] y a la *Lista de Clusters* [CN05]. Los resultados se han publicado en [CRR09, CLRR11, CLRR14, CLRR16, GCMR08, GCMR09].

Las optimizaciones realizadas en el *SAT* logran mejoras significativas en las búsquedas respecto de la estructura original, gracias a cambiar la heurística de construcción. Este nuevo índice se establece como uno de los índices más competitivos respecto del estado del arte, no sólo por la eficiencia en las búsquedas, sino porque su principal ventaja frente a otras alternativas es que no necesita sintonizar ningún parámetro. Por otra parte, las optimizaciones a la *LC* han permitido que el índice pueda aprovechar la existencia de espacio de memoria disponible, y las distancias ya calculadas en la construcción, para mejorar el desempeño en las búsquedas.

1.2.2. Dinamismo y Grandes Volúmenes de Datos

Algunos índices toleran en principio inserciones, pero su eficiencia se degrada y requieren reconstrucción periódica. Otros soportan eliminaciones con el mismo problema de degradación en la eficiencia. Por lo tanto, existen muy pocos índices realmente dinámicos el *M-tree* [CPZ97], el *PM-tree* y el *MX-tree* (variantes del *M-tree*) [SPS04, Sko04, JKF13] y el *eGNAT* [UP05] (basado en el *GNAT* [Bri95]). Además, en [NH12] aparece una propuesta de aplicar el algoritmo de Bentley y Saxe [BS80], que es una técnica general de dinamización de índices, aplicado a algunos índices métricos conocidos, aunque los resultados obtenidos hasta el momento no son prometedores para aplicaciones realmente dinámicas.

Más aún, en muchos casos de interés para búsquedas por similitud en bases de datos métricas los objetos son demasiado grandes y ellos deben alojarse en disco, o los objetos son tantos que el índice mismo no cabe en memoria principal. Sin embargo, el problema de búsqueda en espacios métricos en memoria secundaria aún está inmaduro, aunque está recibiendo cada vez más atención. En este caso

uno se enfrenta al escenario donde el índice no cabe en RAM, y por lo tanto el modelo de contar sólo los cálculos de distancia es impráctico y se debe considerar también el número de operaciones de E/S. El área de las estructuras de datos con E/S eficiente es clásica en Ciencias de la Computación. Por esta razón, existe extensa literatura sobre ella, la que no se puede resumir significativamente en algunos párrafos. Un excelente libro es [Vit08]. Allí se introducen los modelos de costo de E/S más ampliamente utilizados y se consideran cuatro operaciones fundamentales en el paradigma de E/S: Explorar, Ordenar, Buscar, y Mostrar. La complejidad de esas operaciones está bien estudiada y analizada, y muchos problemas particulares pueden ser expresados como combinación de estas operaciones.

Hoy en día, la diferencia entre los tiempos de la CPU y disco es tan significativa que el esfuerzo de lograr eficiencia en las operaciones de E/S es imprescindible, aún logrando una pequeña disminución en el tiempo de E/S. Además, la amplia disponibilidad de los medioambientes de computadoras en red juega también un rol importante en este escenario, porque la transferencia de los datos sobre una red local cuesta aproximadamente lo mismo que la transferencia al disco.

Algunos de los índices para espacios métricos se han diseñado teniendo en mente que ellas se alojarán en la memoria secundaria, por ejemplo el *M-tree* [CPZ97], el *D-index* [DGSZ03a], el *eGNAT* [UPN09], la aproximación *iDistance* [JOT$^+$05], y las variantes del *M-tree*: el *Slim-tree* [JTSF00], el *PM-tree* [SPS04, Sko04] y el *MX-tree* [JKF13]. Además, existen otros métodos como por ejemplo la aproximación de Permutantes [CFN08], el *DSATCL* [BRP10], el *SSS* [BFPR06], la *LC* [CN05] y el *DSAT* [NR08], que permitirían una implementación eficiente en memoria secundaria.

Se han propuesto algunas variantes de inserción y eliminación nuevas para el *DSAT* [NR02]. Se han logrado implementaciones dinámicas y eficientes en memoria secundaria del *DSAT* y de la *LC*. Los resultados se han publicado en [NR08, NR09, NR14, NR16].

Las nuevas variantes de inserción y eliminación del *DSAT* permiten en algunos casos mejorar no sólo la eficiencia de dichas operaciones en sí, sino también que no se degrade el desempeño de la estructura en posteriores búsquedas. Además, existen muy pocas estructuras completamente dinámicas, como el *DSAT*, que luego de realizar inserciones y eliminaciones puedan mantener eficientes las búsquedas. Se presenta una versión para memoria secundaria del *DSAT* que permite realizar inserciones y búsquedas. La *Lista de Clusters Dinámica* (*DLC*), que extiende a la *LC*, permite inserciones, eliminaciones y búsquedas. Finalmente, el índice *Conjunto Dinámico de Clusters* (*DSC*) combina la *DLC* con el *DSAT* de memoria principal a fin de reducir la cantidad de evaluaciones de distancia en las inserciones. Las mejores variantes de estas estructuras generalmente superan a algunas de las buenas alternativas dinámicas y para memoria secundaria de la literatura.

1.2.3. Otras Operaciones

Desde el punto de vista de otras operaciones sobre bases de datos métricas, como el join por similitud, se han presentado algunos trabajos que permiten resolverlo sobre una clase particular de datos métricos como son los espacios de vectores multidimensionales [BBKK01, BKS93, KP03, LR96], sin embargo ninguna de estas estrategias es adecuada para espacios métricos debido a que usan información de las coordenadas que no necesariamente se encuentra disponible en bases de datos métricas. Con respecto al *join de k-vecinos más cercanos*, en el caso de espacios vectoriales multidimensionales, existen varias propuestas que utilizan la información de las coordenadas para calcular resultados aproximados [AP02, AP05, LL00]. Más aún, hasta lo que se sabe, existen pocos intentos de calcular el join de k-vecinos más cercanos en el contexto de bases de datos métricas. En [PCFN06] se presentan algoritmos de costo subcuadrático para construir el grafo de k-vecinos más cercanos de un conjunto métrico, el cual

puede verse como una variante del auto-join por similitud donde se buscan los k-vecinos más cercanos de cada objeto del mismo conjunto.

Por otra parte, si se analizan las diversas propuestas desde el punto de vista de los distintos tipos de join por similitud, se pueden clasificar las diferentes estrategias de implementación. Para el join por rango [Böh01, BK01, BKK02, DGZ03, DGSZ03b, BBKK01]; para join de k pares más cercanos [HS98]; para join de k vecinos más cercanos [BK04]. El join por rango ha sido el operador de join por similitud más estudiado, también se lo concoce con el nombre de θ-join. Entre sus técnicas de implementación más relevantes se encuentran las que se basan en el preprocesamiento de la o las bases de datos para construir un índice, tales como el *eD-index* [DGZ03], *D-index* [DGSZ03b]. Estas técnicas particionan los datos agrupando juntos los objetos similares. Sin embargo, el *eD-index* y el *D-index* se pueden aplicar directamente sólo en el caso de los auto-join. También se han propuesto diversos algoritmos que se basan en la no existencia de un índice para implementar el θ-join [BBKK01, DS01, JS08].

Una aproximación natural para resolver este problema en espacios métricos consiste en indexar uno o ambos conjuntos de datos métricos de manera independiente, usando alguno de los numerosos índices existentes (la mayoría de ellos revisados en [CNBYM01, Sam05, ZADB06, Het09]) y luego resolver consultas por rango para todos los elementos involucrados en uno de los conjuntos sobre el índice del otro.

Así, a pesar de la gran atención que hasta el momento la primitiva del join ha recibido en las bases de datos tradicionales o aún en las bases de datos multidimensionales [BKS93, LR96, BBKK01, KP03], poco se ha hecho para las bases de datos métricas generales [DGSZ03b, DGZ03, JS08, Wan10, CSO+15, GCL+15, SCO+15, SPC13, SPCR15].

Se ha desarrollado un índice, basado en la *LC*, especialmente diseñado para responder a operaciones de join por similitud. Los resultados se publicaron en [PR08, PR09].

El nuevo índice propuesto para join por similitud, la *Lista de Clusters Gemelos* (*LTC*), indexa conjuntamente a las dos bases de datos involucradas en la operación. El costo de construcción de la *LTC* es similar al de construir una única *LTC* para indexar una base de datos más grande. Se han presentado soluciones eficientes con *LTC* para dos clases de join por similitud: el join por rango y el join de k pares más cercanos, siendo la propuesta de solución del join de k pares más cercanos la primera existente en la literatura para resolver este tipo de consulta. Además, a pesar de ser *LTC* un índice para join, también permite resolver consultas por rango en cada uno de los dos conjuntos de datos métricos involucrados.

1.3. Publicaciones Generadas

Se considera que el conocimiento obtenido en el desarrollo de este trabajo abre nuevas posibilidades de investigación en el área de las bases de datos métricas. Las principales contribuciones de esta tesis se plasmaron en las siguientes publicaciones:

[CLRR16] Faster Proximity Searching with the Distal SAT. Edgar Chávez, Verónica Ludueña, Nora Reyes and Patricia Roggero. *Information Systems Journal*, Elsevier Science Publishers. Aceptada para su publicación. Disponible online desde el 9 de enero de 2016. ISSN: 0306-4379.

[NR16] New Dynamic Metric Indices for Secondary Memory. Gonzalo Navarro and Nora Reyes. *Information Systems Journal*, Elsevier Science Publishers. Aceptada para su publicación. Disponible online desde el 7 de abril de 2016. ISSN: 0306-4379.

[CLRR14] Faster Proximity Searching with the Distal SAT. Edgar Chávez, Verónica Ludueña, Nora Reyes and Patricia Roggero. Proceedings of the *7th International Conference on Similarity Search and Applications* (SISAP 2014). Lecture Notes in Computer Science series, Volume N °8821, Pages 58–69. Editors: Agma Juci Machado Traina, Caetano Traina Jr. Springer-Verlag Berlin Heidelberg, 2014. ISBN: 978-3-319-11987-8 (Print) 978-3-319-11988-5 (Online). [1]

[NR14] Dynamic List of Clusters in Secondary Memory. Gonzalo Navarro and Nora Reyes. Proceedings of the *7th International Conference on Similarity Search and Applications* (SISAP 2014). Lecture Notes in Computer Science series, Volume N °8821, Pages 94–105. Editors: Agma Juci Machado Traina, Caetano Traina Jr. Springer-Verlag Berlin Heidelberg, 2014. ISBN: 978-3-319-11987-8 (Print) 978-3-319-11988-5 (Online). [1]

[CLRR11] Reaching Near Neighbors with Far and Random Proxies. Edgar Chávez, Verónica Ludueña, Nora Reyes and Patricia Roggero. Proceedings of the *8th International Conference on Electrical Engineering, Computing Science and Automatic Control* (CCE 2011), IEEE Computer Society. Pages 574–581. ISBN: 978-1-4577-1011-7. 2011. Published in CD.

[CRR09] Delayed Insertion Strategies in Dynamic Metric Indexes. Edgar Chávez, Nora Reyes and Patricia Roggero. Proceedings of the *XXVIII International Conference of the Chilean Computer Science Society*, IEEE Computer Society. Pages 34–42. ISBN 978-0-7695-4137-2 ISSN 1522-4902. 2010. [2].

[NR09] Dynamic Spatial Approximation Trees for Massive Data. Gonzalo Navarro and Nora Reyes. Proceedings of the *Second International Workshop on Similarity Search and Applications* (SISAP 2009), IEEE Computer Society. Pages 81–88. ISBN 978-0-7695-3765-8. Editors: Tomás Skopal, Pavel Zezula, Vlatislav Dohnal.

[PR09] Solving Similarity Joins and Range Queries in Metric Spaces with the List of Twin Clusters. Rodrigo Paredes and Nora Reyes. *Journal of Discrete Algorithms*, Elsevier. Volume 7, Issue 1, March 2009, Pages 18–35. ISSN 1570-8667.

[GCMR09] Parallel Query Processing on Distributed Clustering Indexes. Veronica Gil-Costa, Mauricio Marin and Nora Reyes. *Journal of Discrete Algorithms*, Elsevier. Volume 7, Issue 1, March 2009, Pages 3–17. ISSN 1570-8667.

[GCMR08] An Empirical Evaluation of a Distributed Clustering-Based Index for Metric Spaces Databases. Verónica Gil Costa, Mauricio Marín and Nora Reyes. Proceedings of the *First International Workshop on Similarity Search and Applications* (SISAP 2008), IEEE. Pages 95–102. ISBN 0-7695-3101-6. [3]

[PR08] List of Twin Clusters: a Data Structure for Similarity Join in Metric Spaces. Rodrigo Paredes and Nora Reyes. Proceedings of the *First International Workshop on Similarity Search and Applications* (SISAP 2008), IEEE. Pages 131–138. ISBN 0-7695-3101-6. [3]

[1] El artículo obtuvo uno de los cinco "Certificate of Best Paper" y fue invitado a ser publicado en versión extendida para un "Special Issue" con los mejores artículos de SISAP 2014 en *Information Systems Journal* de Elsevier Science Publisher.

[2] El artículo fue invitado por selección del Comité entre los mejores artículos, y corresponde a una versión extendida y revisada del artículo *Data Buffering Strategies to Improve Dynamic Metric Indexing*, presentado en la XXVIII International Conference of the Chilean Computer Science Society en 2009.

[3] El artículo resultó invitado para enviar una versión extendida para un "Special Issue" dedicado a artículos seleccionados del SISAP 2008 en el Journal of Discrete Algorithms de Elsevier.

[NR08] Dynamic Spatial Approximation Trees. Gonzalo Navarro and Nora Reyes. The *ACM Journal of Experimental Algorithmics*. Acepted: August, 2007. Published: June 2008, Vol. 12, SECTION: 1 - Regular Papers, Artículo 1.5, Pages 1–68. ISSN 1084-6654.

Además, durante el desarrollo de esta tesis se realizaron, entre otras, las siguientes publicaciones internacionales en el área:

[NPRB16] An Empirical Evaluation of Intrinsic Dimension Estimators. Gonzalo Navarro, Rodrigo Paredes, Nora Reyes and Cristian Bustos. *Information Systems Journal*, Elsevier Science Publishers. En etapa de revisión. Invitado para su publicación entre los mejores trabajos de SISAP 2015.

[BNRP15] An Empirical Evaluation of Intrinsic Dimension Estimators. Cristian Bustos, Gonzalo Navarro, Nora Reyes and Rodrigo Paredes. Proceedings of the *8th International Conference on Similarity Search and Applications* (SISAP 2015). Lecture Notes in Computer Science series, Nº 9371, pages 125–137, 2015. Editors: Giuseppe Amato, Richard Connor, Fabrizio Falchi, Claudio Gennaro. Springer-Verlag Berlin Heidelberg 2015. ISBN 978-3-319-25086-1. ISBN 978-3-319-25087-8 (eBook). [4]

[MCPR13] (Very) Fast (All) k-Nearest Neighbors Without Indexing. Natalia Miranda, Edgar Chávez, Fabiana Piccoli and Nora Reyes. Proceedings of the *6th International Conference on Similarity Search and Applications* (SISAP 2013). Lecture Notes in Computer Science series, Nº 8199, pages 300–311, 2013. Editors: Nieves Brisaboa, Oscar Pedreira, Pavel Zezula. Springer-Verlag Berlin Heidelberg 2013. ISBN: 978-3-642-41061-1 (Print) 978-3-642-41062-8 (Online).

[BPR12] DSACL+-tree: A Dynamic Data Structure for Similarity Search in Secondary Memory. Luis Britos, A. Marcela Printista, Nora Reyes. Proceedings of the *5th International Conference on Similarity Search and Applications* (SISAP 2012). Lecture Notes in Computer Science series, Nº 7404, pages 116–131, 2012. Editors: Gonzalo Navarro, Vladimir Pestov. Springer-Verlag Berlin Heidelberg 2012. ISBN 978-3-642-32152-8. ISSN 0302-9743.

[BRP10] Enlarging Nodes to Improve Dynamic Spatial Approximation Trees. Marcelo Barroso, Nora Reyes and Rodrigo Paredes. Proceedings of the *3rd International Conference on Similarity Search and Applications* (SISAP 2010), ACM Press. Pages 41–48. ISBN 978-1-4503-0420-7.

[BFPR06] Similarity search using sparse pivots for efficient multimedia information retrieval. Nieves R. Brisaboa, Antonio Fariña, Oscar Pedreira and Nora Reyes. Proceedings of the *2nd IEEE International Workshop on Multimedia Information Processing and Retrieval* (MIPR 2006). December 2006. Pages 881–888. ISSN: 0-7695-2746-9

[4] El artículo obtuvo uno de los cinco "Certificate of Best Paper" y fue invitado a ser publicado en versión extendida para un "Special Issue" con los mejores artículos de SISAP 2015 en *Information Systems Journal* de Elsevier Science Publisher.

Capítulo 2

Conceptos Básicos

La teoría matemática de espacios métricos fue publicada por *Felix Hausdorff* en 1914 [1], *Maurice Fréchet* introdujo la noción de espacios métricos en 1906 [2] [DD09]. Sin embargo, sólo recientemente se ha considerado esta teoría como modelo para el problema de búsquedas por similitud.

Se introducen aquí los principales conceptos del modelo de espacios métricos y los tipos más comunes de búsquedas por similitud. Se describen también las distintas funciones de distancia que se usarán en la práctica para modelizar la "similitud" entre objetos. Además, se introduce el concepto de dimensionalidad intrínseca, el cual formaliza la idea de "facilidad" o "dificultad" para buscar en un espacio métrico dado [CNBYM01, ZADB06, Sam05, BNRP15]. Finalmente, se dan algunos ejemplos de espacios métricos y se describen los espacios métricos, junto con los detalles generales de los experimentos realizados, utilizados para la evaluación de las propuestas en este trabajo.

2.1. El Modelo de Espacios Métricos

El modelo formal de espacio métrico se define de la siguiente manera: sea $\mathbb{U}$ un *universo de objetos*, y d una función definida como $d : \mathbb{U} \times \mathbb{U} \to \mathbb{R}^+$ que denota una medida de *distancia* entre objetos (es decir, mientras más pequeña es la distancia d, más cercanos o similares son los objetos). Las funciones de distancia deben cumplir las siguientes propiedades:

(p1) $\forall x, y \in \mathbb{U}, d(x, y) \geq 0$ positividad,

(p2) $\forall x, y \in \mathbb{U}, d(x, y) = d(y, x)$ simetría,

(p3) $\forall x \in \mathbb{U}, d(x, x) = 0$ reflexividad,

y en muchos casos

(p4) $\forall x, y \in \mathbb{U}, x \neq y \Rightarrow d(x, y) > 0$ positividad estricta

Las propiedades enumeradas de la función sólo aseguran su definición consistente. Si d es en verdad una métrica, es decir si satisface:

[1] *Grundzüge der Mengenlehre*, de Felix Hausdorff (1868-1942), se publicó en 1914, en él aparecía la teoría de espacios métricos y topológicos.

[2] En 1906 la disertación doctoral *Sur quelques points du calcul functionnel* de Maurice Fréchet (1878-1973) se introdujo, dentro de un estudio sistemático de operaciones funcionales, la noción de espacio métrico.

(p5) $\forall x, y, z \in \mathbb{U}, d(x, y) \leq d(x, z) + d(z, y)$ desigualdad triangular,

entonces, el par $(\mathbb{U}, d)$ recibe el nombre de *espacio métrico*.

De aquí en más se utilizará el término *distancia* entendiendo que se hace referencia realmente a una métrica.

La desigualdad triangular es la propiedad más importante al momento de realizar las búsquedas por similitud, ya que permite obtener una cota inferior de la distancia entre dos objetos y así evitar calcular la distancia d entre ellos:

$$\forall x, y, z \in \mathbb{U}, |d(x, y) - d(y, z)| \leq d(x, z) \tag{2.1}$$

Formalmente, puede establecerse este resultado mediante el siguiente lema:

Lema 1 *Dado un espacio métrico* $(\mathbb{U}, d)$ *se tiene que* $\forall x, y, z \in \mathbb{U}$*:*

$$|d(x, z) - d(z, y)| \leq d(x, y) \leq d(x, z) + d(z, y)$$

Demostración:

Por la desigualdad triangular se tiene que:

$$d(x, z) \leq d(x, y) + d(y, z)$$

despejando se obtiene que:

$$d(x, z) - d(y, z) \leq d(x, y)$$

lo cual, dado que d cumple además con la propiedad de simetría, puede transformarse en:

$$d(x, z) - d(z, y) \leq d(x, y) \tag{2.2}$$

Ahora, se procede de manera análoga con $d(y, z)$, dado que por desigualdad triangular se cumple que:

$$d(y, z) \leq d(y, x) + d(x, z)$$

nuevamente, despejando se obtiene:

$$d(y, z) - d(x, z) \leq d(y, x)$$

y por la propiedad de simetría se llega a:

$$d(y, z) - d(x, z) \leq d(x, y) \tag{2.3}$$

Luego, a partir de las desigualdades 2.2 y 2.3, se puede deducir la cota inferior para $d(x, z)$:

$$|d(x, z) - d(y, z)| \leq d(x, y) \tag{2.4}$$

La cota superior se obtiene simplemente mediante la desigualdad triangular:

$$d(x, y) \leq d(x, z) + d(z, y) \tag{2.5}$$

□

Las propiedades de la función de distancia enumeradas, exceptuando la desigualdad triangular, sólo aseguran su definición consistente, y no pueden ser usadas para ahorrar evaluaciones de distancia en una consulta por similitud. Como se ha descripto, la desigualdad triangular es la propiedad que permite

evitar algunos cálculos de distancia, gracias a las cotas inferiores y superiores que pueden deducirse por medio de ella.

Además, la desigualdad triangular puede extenderse a varios elementos [WS90a]. Si se quiere obtener una cota inferior para la distancia $d(x, y)$, siendo $x, y, o_1, o_2, \ldots, o_k \in \mathbb{U}$, conociendo las distancias $d(x, o_1), d(o_1, o_2), \ldots, d(o_k, y)$, siendo d_M la mayor de las distancias conocidas; es decir, $d_M = \text{máx}\{d(x, o_1), d(o_1, o_2), \ldots, d(o_{k-1}, o_k), d(o_k, y)\}$; se puede utilizar el siguiente lema, el cual establece formalmente la *propiedad de desigualdad triangular generalizada*:

Lema 2 *Dado un espacio métrico* $(\mathbb{U}, d)$*, sean* $x, y, o_1, o_2, \ldots, o_k \in \mathbb{U}$ *y se conocen las distancias* $\mathbb{P} = \{d(x, o_1), d(o_1, o_2), \ldots, d(o_{k-1}, o_k), d(o_k, y)\}$*, siendo* $d_M = \text{máx}\{d(x, o_1), d(o_1, o_2), \ldots, d(o_k, y)\}$*, entonces:*

$$d(x, y) \geq d_M - \sum_{d \in \mathbb{P} - \{d_M\}} d$$

Demostración: Se puede demostrar por inducción sobre el número de distancias (objetos intermedios) en $\mathbb{P}$ y aplicación repetida de la desigualdad triangular y del Lema 1.

□

Claramente este límite inferior para $d(x, y)$ es inútil si $d_M - \sum_{d \in \mathbb{P} - \{d_M\}} d \leq 0$

2.1.1. Funciones de Distancia

Como se ha mencionado anteriormente, las funciones de distancia se usan para modelar el "*parecido*" entre dos objetos: mientras menor sea la distancia entre dos objetos dados, más similares ellos serán.

Existen varios ejemplos de funciones de distancia, que pueden ser aplicadas sobre distintos tipos de datos. A menudo las funciones de distancia se definen a medida para una aplicación específica o para un dominio de posibles aplicaciones. Así, la función de distancia es especificada por un experto, aún cuando la definición de la función de distancia no restringe el tipo de consultas que se pueden resolver. Se describen aquí algunas de las distintas funciones de distancia existentes [ZADB06]. Más aún, siguiendo la clasificación de distancias presentada en [ZADB06], las funciones de distancia pueden dividirse en dos grupos con respecto al valor retornado:

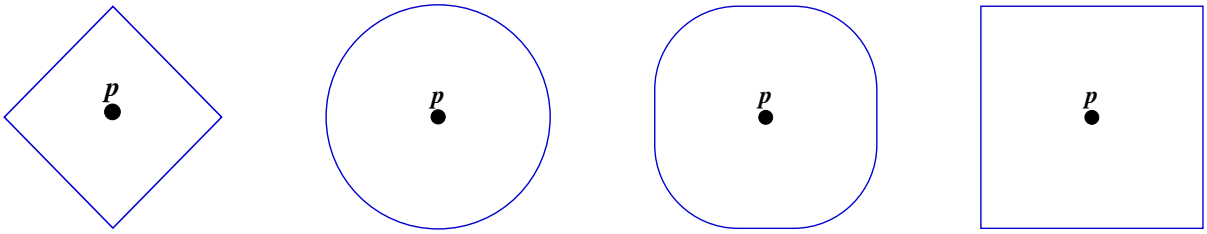
Discretas: la función de distancia devuelve sólo uno de un pequeño conjunto de valores posibles.

Continuas: la cardinalidad del conjunto de valores posibles de la función de distancia es infinito o muy grande.

Algunos ejemplos de funciones de distancia son las siguientes:

1. *Distancias de Minkowski*: Llamadas L_p, es una familia de funciones de distancia, de uso adecuado en espacios vectoriales de dimensión finita D, o D-dimensionales. En un espacio vectorial $\mathbb{R}^D$ los objetos son tuplas o vectores $\vec{x} = (x_1 \ldots, x_D)$, definidas por sus D coordenadas reales. Para $p \in \mathbb{Z}^+, \vec{x} = (x_1, \ldots, x_D), \vec{y} = (y_1, \ldots, y_D) \in \mathbb{R}^D$, se definen las funciones de distancia $L_p : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}^+$ como sigue:

$$L_p(\vec{x}, \vec{y}) = L_p((x_1, \ldots, x_D), (y_1, \ldots, y_D)) = \left(\sum_{i=1}^{D} |x_i - y_i|^p \right)^{1/p} \tag{2.6}$$



(a) L_1. (b) L_2. (c) L_6. (d) L_∞.

Figura 2.1: Interpretación gráfica en $\mathbb{R}^2$ de la equidistancia entre puntos, de acuerdo a las distintas funciones de distancia de Minkowski ([ZADB06, Fig. 1.1]).

Casos particulares de ésta son:

L_1: conocida como la *distancia de bloques* o *distancia de Manhattan*, ya que en dos dimensiones la distancia entre dos puntos se corresponde con la distancia que se debe caminar en una ciudad de manzanas rectangulares para llegar desde un punto al otro.

L_2: es la conocida *distancia Euclidiana*, y es nuestra noción de distancia espacial. En nuestros ejemplos sobre espacios vectoriales se usará la distancia L_2, la que se define, en particular, como:

$$L_2\left((x_1,\ldots,x_D),(y_1,\ldots,y_D)\right) = \sqrt{\sum_{i=1}^{D}|x_i-y_i|^2} \tag{2.7}$$

L_∞: conocida como *distancia de Máximo* o *distancia de Chebyshev*, para calcularla hay que tomar el límite de la fórmula de L_p, cuando p tiende a $+\infty$. La distancia viene dada por la mayor diferencia entre las coordenadas de los vectores. En el caso de posiciones de un tablero de ajedrez, corresponde al número mínimo de movimientos de la figura del rey para ir de una casilla a otra; por ello también se la conoce como "distancia del tablero de ajedrez". En símbolos:

$$L_\infty((x_l,\ldots,x_D),(y_1,\ldots,y_D)) = \max_{i=1}^{D}|x_i-y_i| \tag{2.8}$$

La Figura 2.1 ilustra gráficamente algunos de estos ejemplos, mostrando los puntos equidistantes a un centro p generados por las distintas variantes de las funciones, considerando puntos de un espacio de $\mathbb{R}^2$.

Estas distancias se utilizan en la práctica si las componentes de los vectores son independientes, como puede considerarse por ejemplo en las evaluaciones experimentales de los índices.

2. *Distancia Coseno*: se basa en una medida de similitud entre dos vectores llamada *similitud de coseno* que mide el coseno del ángulo que se forma entre ellos. En este caso, dos vectores con la misma orientación tienen una similitud de coseno de 1, dos vectores a 90°tienen similitud 0 y dos vectores opuestos tienen similitud -1. Sin embargo, la similitud de coseno se usa particularmente en el espacio positivo, donde la salida está limitada en el intervalo $[0,1]$.

   Dados dos vectores $\vec{x}$ y $\vec{y}$ de dimensión finita D, la similitud de coseno puede derivarse usando la fórmula del producto interno Euclídiano, siendo θ el ángulo formado entre los dos vectores:

$$\vec{x}\cdot\vec{y} = ||\vec{x}||||\vec{y}||\cos\theta,$$

entonces, se tiene que:

$$sim(\vec{x}, \vec{y}) = \cos\theta = \frac{\vec{x} \cdot \vec{y}}{||\vec{x}||||\vec{y}||} = \frac{\sum_{i=1}^{D} x_i \times y_i}{\sqrt{\sum_{i=1}^{D}(x_i)^2} \times \sqrt{\sum_{i=1}^{D}(y_i)^2}}.$$

Dado que $sim(\vec{x}, \vec{y})$ sólo es una medida de similitud, que en particular no cumple con la desigualdad triangular, se utiliza el ángulo formado entre los vectores $\vec{x}$ y $\vec{y}$. La función de *distancia coseno* se define como $d(\vec{x}, \vec{y}) = \arccos(sim(\vec{x}, \vec{y}))$ [BYRN11]

3. *Distancia de Edición*: también llamada distancia de *Levenshtein* [Lev65, Lev66] es una de las maneras de establecer la similitud entre cadenas de caracteres y tiene aplicaciones en diversas áreas, tales como recuperación de texto, reconocimiento de patrones, biología computacional y procesamiento de señales [Nav01].

   La *distancia de edición* entre cadenas de símbolos (o palabras) se define como el número mínimo de operaciones individuales (inserción, sustitución o eliminación de símbolos) necesario para transformar una cadena en la otra. Claramente ésta es una función de distancia *discreta*.

4. *Distancia de **Jaccard***: es aplicable en la comparación de conjuntos, se basa en el cálculo del *coeficiente de Jaccard* que mide la relación entre el cociente de las cardinalidades de la intersección y de la unión de ambos conjuntos [MRS08]. Formalmente, sean A y B dos conjuntos, el coeficiente de Jaccard se define como:

$$d(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}.$$

   La función de *distancia de Jaccard* se basa simplemente en la relación existente entre las cardinalidades de la intersección y de la unión de los conjuntos comparados.

5. *Distancia de **Hausdorff***: una medida de similitud de conjuntos, un poco más complicada, es la distancia de Hausdorff [HKR93], también llamada distancia de *Pompeiu-Hausdorff* [RWW09]. A diferencia del coeficiente de Jaccard, donde cualquier par de elementos de los conjuntos deben ser o iguales o completamente distintos, la distancia de Hausdorff empareja elementos, considerando una función de distancia entre elementos d_e. Así, dados dos conjuntos A y B y siendo

$$\begin{aligned} d_p(x, B) &= \inf_{y \in B} d_e(x, y) \\ d_p(A, y) &= \inf_{x \in A} d_e(x, y) \\ d_s(A, B) &= \sup_{x \in A} d_p(x, B) \\ d_s(B, A) &= \sup_{y \in B} d_p(A, y) \end{aligned}$$

   la *distancia de Hausdorff* se define formalmente como:

$$d(A, B) = \text{máx}\{d_s(A, B), d_s(B, A)\}.$$

   En particular, la distancia $d_e(x, y)$ definida entre elementos x e y cualesquiera de los conjuntos A y B puede ser una métrica arbitraria, por ejemplo la distancia Euclidiana, y es la que refleja el dominio particular de la aplicación. Expresado de manera más sencilla, las distancias de Hausdorff miden el grado en el cual cada punto del conjunto "modelo" A cae cerca de algún punto del conjunto "imagen" B y viceversa. En otras palabras, dos conjuntos A y B están entre ellos a distancia de Hausdorff de r si y sólo si cualquier punto de un conjunto está a distancia a lo más de r de cada punto del otro conjunto.

Un ejemplo de aplicación es la comparación de formas en procesamiento de imágenes, donde cada forma está definida por un conjunto de puntos de $\mathbb{R}^2$.

La distancia de Hausdorff en general puede ser infinita. Sin embargo, si los conjuntos A y B están limitados o acotados, se garantiza que $d(A, B)$ es finita.

El cálculo de todas estas funciones de distancia suele requerir mucho tiempo de procesamiento. En consecuencia, la mayoría de las técnicas de índexación para búsqueda en espacios métricos persigue minimizar el número de evaluaciones de distancia.

2.2. Consultas por Similitud o Proximidad

Dado un espacio métrico $(\mathbb{U}, d)$, se considera un subconjunto finito del universo $\mathbb{X} \subseteq \mathbb{U}$, de tamaño $n = |\mathbb{X}|$, al que se denomina *diccionario*, *base de datos*, o simplemente el conjunto de objetos o elementos.

Una consulta por similitud o proximidad se define explícita o implícitamente por un objeto de consulta o "query" $q \in \mathbb{U}$ y una restricción sobre la forma o grado de proximidad o similitud requerida, comúnmente expresada como una distancia. La respuesta a una consulta devuelve todos los objetos que satisfacen la condición o restricción de selección, los cuales presumiblemente serán aquellos objetos "cercanos" al objeto de consulta q dado.

A continuación se describen las consultas por similitud más comunes: *consultas por rango* y *consultas de k-vecinos más cercanos*. Luego se mencionarán otras clases de consultas que pueden ser de interés en aplicaciones sobre bases de datos métricas [ZADB06], algunas de las cuales son de especial interés en este trabajo.

2.2.1. Consultas por Rango

Posiblemente las consultas por rango sean el tipo más común de consulta por similitud. La consulta en este caso se especifica dando un objeto $q \in \mathbb{U}$ junto con un radio de consulta r, como la restricción de distancia. Este tipo de consulta consiste en encontrar todos los objetos que estén a una distancia menor o igual que r de un objeto q dado. Formalmente, una *consulta por rango* será un par (q, r) siendo q un elemento de $\mathbb{U}$, $r \in \mathbb{R}^+$ indicando el radio o tolerancia de la consulta y d una función de distancia definida sobre los elementos de $\mathbb{U}$:

$$(q, r) = \{x \in \mathbb{X} / d(q, x) \leq r\}.$$

Puede observarse, dado que $q \in \mathbb{U}$, que no es necesario que q pertenezca a la base de datos $\mathbb{X}$. La Figura 2.2 muestra gráficamente esta situación en $\mathbb{R}^2$ y utilizando distancia Euclidiana. En algunas aplicaciones puede ser necesario que los objetos individuales que pertenecen al conjunto respuesta sean ordenados de acuerdo a su distancia al objeto de consulta q, estableciendo así un ranking de los mismos.

Un ejemplo sería una aplicación geográfica en la que una consulta por rango podría ser: *Obtener todos las estaciones de servicio ubicadas a una distancia de a lo sumo un kilómetro desde donde se encuentra mi auto*.

En el caso particular en que el radio de búsqueda sea $r = 0$, la consulta por rango $(q, 0)$ se denomina una consulta puntual o una consulta de correspondencia exacta. En esta consulta se está buscando una

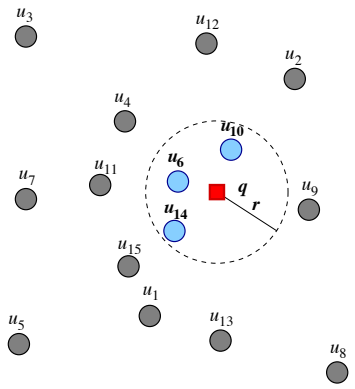


Figura 2.2: Consultas por rango, partiendo desde un q dado, con radio r.

copia idéntica del objeto de consulta q. El uso más habitual de este tipo de consulta es en los algoritmos de eliminación, cuando lo que se busca es ubicar un objeto particular que pertenece a la base de datos, para eliminarlo efectivamente de la misma.

2.2.2. Consultas de k Vecinos Más Cercanos

Cuando se desea realizar una consulta por objetos similares mediante una búsqueda por rango, se debe especificar una distancia máxima que deben cumplir los objetos. Sin embargo, en muchas aplicaciones puede resultar difícil especificar un radio sin tener algún conocimiento sobre los datos y la función de distancia. Por ejemplo, en un espacio de palabras y usando distancia de edición, puede ser fácil decidir cuál es la distancia máxima a la que se considerarían dos palabras como similares. Por ejemplo, se puede especificar un radio $r = 2$, lo cual representa buscar las palabras que tengan que hacer a lo sumo 2 operaciones de edición respecto del elemento de consulta q dado. En este caso particular la semántica de la consulta es clara. Sin embargo, al considerar que los objetos son secuencias de audio, una distancia es un número en $\mathbb{R}^+$ y por lo tanto no resulta fácil especificar un umbral que indique hasta qué punto se admite como similar a una secuencia de audio respecto de q. Si el radio de consulta r es demasiado chico, el conjunto respuesta podría ser vacío y se debería realizar una nueva búsqueda con un radio mayor para obtener algún resultado. Por otra parte, si r es demasiado grande la consulta podría ser muy costosa computacionalmente y el conjunto respuesta contendrá muchos elementos poco significativos (muy distintos a q).

Así, sería más sencillo especificar la similitud respecto del objeto de consulta q indicando cuántos elementos de los más similares se desean obtener. Entonces, una forma alternativa de buscar objetos similares es usar las que se conocen como *consultas de vecinos más cercanos*. En particular, en este tipo de consultas se debe especificar un valor k que indica cuántos de los elementos más similares a q se desean obtener en la respuesta. Entonces, una *consulta de los k-vecinos más cercanos* consiste en encontrar los k objetos en $\mathbb{X}$ más cercanos o similares a un objeto de consulta q dado y se denota con $kNN(q)$. La abreviatura *NN* viene del término inglés "Nearest Neighbors"(vecinos más cercanos). Formalmente, dado un objeto $q \in \mathbb{U}$ y $k \in \mathbb{N}$, se recupera un conjunto $A \subseteq \mathbb{X}$ tal que la cardinalidad $|A| = k$ y $\forall x \in A \wedge \forall y \in \mathbb{X} - A, d(q, x) \leq d(q, y)$. Cabe notar que cualquier conjunto de elementos que cumpla la condición satisface la consulta, por lo tanto la respuesta a este tipo de consulta no es necesariamente unívoca. Así, cuando varios objetos están a la misma distancia que el k-ésimo vecino más cercano, los empates se pueden resolver arbitrariamente.

La versión más elemental de esta consulta surge al querer recuperar el elemento más cercano o

similar a un objeto de consulta q dado; es decir, el vecino más cercano a q en $\mathbb{X}$. Esta consulta puede fácilmente considerarse como un caso particular de consulta $kNN(q)$ cuando $k = 1$.

Particularmente, una consulta kNN puede construirse sistemáticamente sobre consultas por rango. La Figura 2.3 ilustra gráficamente la búsqueda, nuevamente trabajando en $\mathbb{R}^2$ y utilizando distancia Euclidiana. En este caso, para hacer más evidente el tipo de consulta, se ha graficado también el rango necesario para encerrar al menos 5 puntos (aunque éste realmente encierra a más de 5 objetos). Así es posible observar que existirían, para esta consulta (dado q y con $k = 5$) otras posibles respuestas. Continuando con el ejemplo de la aplicación geográfica, se podría plantear la siguiente consulta: *Obtener las 3 estaciones de servicio más cercanas a donde está mi auto*.

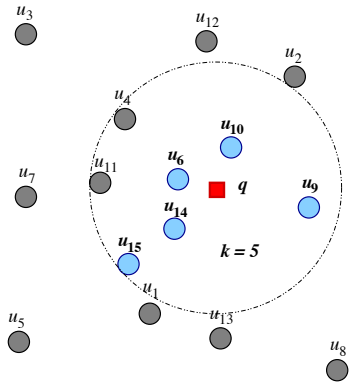


Figura 2.3: Consulta de los k-vecinos más cercanos, partiendo desde un q dado y un valor de $k = 5$.

Se dice que un algoritmo de búsqueda de vecino más cercano es *de rango óptimo* cuando el costo de una consulta $kNN(q)$ es exactamente el mismo que el de una búsqueda por rango (q, r), donde r es la distancia al k-ésimo vecino más cercano retornado. Los algoritmos de búsqueda de vecino más cercano de rango óptimo pueden derivarse sistemáticamente desde algoritmos de búsqueda por rango [HS00, HS03b, Sam05].

2.2.3. Otras Consultas por Similitud

Aunque las búsquedas por rango y de vecinos más cercanos son las clases de búsquedas más comunes, existen otros tipos de consultas por similitud que pueden ser de interés en espacios métricos [ZADB06, Het09]. De los otros tipos de consulta existentes, se describe aquí una que tiene especial interés para nuestro trabajo: el ensamble, o por su nombre en inglés "*join*", por similitud y algunas de sus variantes.

Join por Similitud

En la actualidad existe numerosa información disponible desde la web, la mayoría de ella no estructurada. Sin embargo, algunos servicios web requieren esta información como datos estructurados. Además, dada la variedad de fuentes de información, puede ocurrir que la calidad no sea la adecuada. Por lo tanto, un desafío importante es proveer datos consistentes y libres de errores, lo cual implica alguna clase de proceso de limpieza o integración de datos, básicamente implementado por medio de un proceso denominado *ensamble por similitud* o *join por similitud* [ZADB06].

Esto es, dadas dos bases de datos conteniendo el mismo tipo de datos, encontrar pares de objetos

formados por un elemento de cada base de datos, tal que satisfagan algún criterio o predicado de similitud. En el caso particular que ambas bases de datos coincidan, se habla de *join de auto-similitud*. Para ilustrar con otro ejemplo este concepto, puede considerarse una agencia de contratación de personal. Por una parte, la agencia tiene un conjunto de datos de perfiles y currículums de mucha gente que busca trabajo. Por otra parte, la agencia tiene un conjunto de datos de perfiles de trabajos solicitados por varias compañías que buscan empleados. Lo que la agencia tiene que hacer en ese caso es encontrar los pares persona-compañía, los cuales compartan un perfil similar.

Formalmente, utilizando notación propia de base de datos, la operación de *join por similitud* entre dos bases de datos $\mathbb{A} \subseteq \mathbb{U}$ y $\mathbb{B} \subseteq \mathbb{U}$, considerando que el criterio de similitud que deben satisfacer los pares de elementos es Φ, puede definirse como:

$$\mathbb{A} \bowtie_\Phi \mathbb{B} = \{(x, y)/(x \in \mathbb{A} \wedge y \in \mathbb{B}) \wedge \Phi(x, y)\} \subseteq \mathbb{A} \times \mathbb{B}$$

En el caso particular del auto-join por similitud sobre una base de datos $\mathbb{A} \subseteq \mathbb{U}$, se tendría que:

$$\mathbb{A} \bowtie_\Phi \mathbb{A} = \{(x, y)/x, y \in \mathbb{A} \wedge \Phi(x, y)\} \subseteq \mathbb{A}^2$$

A modo de ejemplo, el *join natural* de bases de datos tradicionales puede considerarse como un join por similitud en el que el criterio de similitud particular utilizado es la igualdad.

Algunas de las posibles variantes del join por similitud entre dos bases de datos métricas $\mathbb{A} \subseteq \mathbb{U}$ y $\mathbb{B} \subseteq \mathbb{U}$, son:

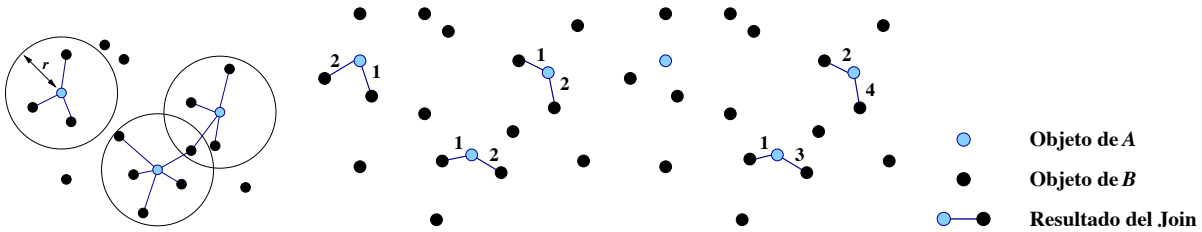
***Join por rango*:** dado un radio o umbral $r \geq 0$ encontrar todos los pares de objetos a distancia a lo más r entre sí; es decir, la operación de join por similitud donde el criterio de similitud es $\Phi = d(x, y) \leq r$. En este caso, se lo denota como $\mathbb{A} \bowtie_r \mathbb{B}$ por simplicidad. En el caso particular que $r = 0$, el join por rango coincide con la noción tradicional del join natural de bases de datos relacionales.

***Join de k-vecinos más cercanos*:** encontrar los pares de elementos de $A \times B$ tales que existen k pares por cada elemento x de $\mathbb{A}$, formados por x y cada uno sus k vecinos más cercanos en $\mathbb{B}$ [PR09]. El join de k-vecinos más cercanos se denota como $\mathbb{A} \bowtie_{kNN} \mathbb{B}$, en este caso $\forall x \in \mathbb{A}$, existen k pares de la forma:$(x, y_i), \forall y_i \in kNN(a) \subseteq \mathbb{B}$. Claramente, $|\mathbb{A} \bowtie_{kNN} \mathbb{B}| = n\,k$ si $|\mathbb{A}| = n$ y $|\mathbb{B}| \geq k$. Formalmente se define como:

$$\mathbb{A} \bowtie_{kNN} \mathbb{B} = \{(x, y_1), (x, y_2), \ldots, (x, y_k)/x \in \mathbb{A}, y_i \in kNN_{\mathbb{B}}(x)\}$$

***Join de k pares de vecinos más cercanos*:** encontrar los k pares de elementos más cercanos entre sí de $\mathbb{A} \times \mathbb{B}$. Se denota con $\mathbb{A} \bowtie_k \mathbb{B}$. Formalmente, sea $k \in \mathbb{N}$, $\mathbb{A} \bowtie_k \mathbb{B} = S \subseteq \mathbb{A} \times \mathbb{B}$ tal que la cardinalidad $|S| = k$ y $\forall (x, y) \in S \wedge \forall (v, w) \in (\mathbb{A} \times \mathbb{B}) - S, d(x, y) \leq d(v, w)$. Notar que, tal como ocurría con la consulta de kNN, satisface la consulta cualquier conjunto de pares que cumpla la condición, por lo tanto la respuesta a este tipo de consulta no es necesariamente unívoca. Así, cuando varios pares de objetos de $\mathbb{A} \times \mathbb{B}$ están a la misma distancia que el k-ésimo par más cercano, los empates se pueden resolver arbitrariamente.

Cabe destacar que la operación de join por similitud no es siempre simétrica. Dados $\mathbb{A} \subseteq \mathbb{U}$ y $\mathbb{B} \subseteq \mathbb{U}$, puede ocurrir que $\mathbb{A} \bowtie_\Phi \mathbb{B} \neq \mathbb{B} \bowtie_\Phi \mathbb{A}$. Además, a diferencia de la operación de join tradicional sobre bases de datos relacionales, el resultado de la operación no es un conjunto de objetos del mismo tipo de los conjuntos de objetos a los que se aplica; es decir, en la operación de join tradicional los operandos y el



(a) Join por Rango, radio r. (b) Join de k Vecinos más Cercanos, $k = 2$. (c) Join de k Pares más Cercanos, $k = 4$.

Figura 2.4: Diferencia entre las variantes de la operación de join por similitud ([BK04, Fig. 1]).

resultado son relaciones, mientras que en la operación de join por similitud los operandos son conjuntos de objetos y el resultado es una relación binaria definida entre ellos (conjunto de pares de objetos).

La Figura 2.4 ilustra los tres tipos de operaciones de join sobre las bases de datos $\mathbb{A} \subseteq \mathbb{R}^2$ y $\mathbb{B} \subseteq \mathbb{R}^2$ y utilizando distancia Euclidiana. Se puede observar la diferencia entre el resultado de un join por rango (con radio r), un join de k vecinos más cercanos (considerando $k = 2$) y un join de k-pares de vecinos más cercanos (considerando $k = 4$): Esta figura se basa en la **Figura 1** de [BK04].

Se pueden construir algoritmos para resolver las variantes de join por rango $\bowtie_r$, entre dos bases de datos $\mathbb{A} \subseteq \mathbb{U}$ y $\mathbb{B} \subseteq \mathbb{U}$, sobre la base de resolver consultas por rango en $\mathbb{B}$ ($(a, r)_{\mathbb{B}}$) para cada elemento $a \in \mathbb{A}$. De la misma manera, se pueden obtener algoritmos para resolver el join de k vecinos más cercanos $\bowtie_{kNN}$ entre $\mathbb{A}$ y $\mathbb{B}$, resolviendo consultas de kNN en $\mathbb{B}$ ($kNN_{\mathbb{B}}(a)$) para cada $a \in \mathbb{A}$. Más aún, es posible obtener el mismo conjunto de pares de un join por rango $\mathbb{A} \bowtie_r \mathbb{B}$, si se arman los pares (a, b) tales que $a \in (b, r)_{\mathbb{A}}$; es decir, mediante consultas por rango $(b, r)_{\mathbb{A}}$ para cada elemento $b \in \mathbb{B}$. Sin embargo, esto mismo no es aplicable para el join de k vecinos más cercanos $\mathbb{A} \bowtie_{kNN} \mathbb{B}$, dado que mediante consultas $kNN_{\mathbb{A}}(b)$ para todo $b \in \mathbb{B}$, la respuesta sería distinta; es decir:

$$\begin{aligned} \mathbb{A} \bowtie_{kNN} \mathbb{B} &= \{(x, y_1), (x, y_2), \ldots, (x, y_k)/x \in \mathbb{A}, y_i \in kNN_{\mathbb{B}}(x)\} \\ &\neq \{(x_1, y), (x_2, y), \ldots, (x_k, y)/y \in \mathbb{B} \wedge x_i \in kNN_{\mathbb{A}}(y)\} \end{aligned}$$

Para poder conseguir armar los mismos pares que $\mathbb{A} \bowtie_{kNN} \mathbb{B}$, realizando consultas desde elementos $b \in \mathbb{B}$ en la base de datos $\mathbb{A}$, se deberían realizar consultas que recuperaran para cada $b \in \mathbb{B}$ los elementos $a \in \mathbb{A}$ que tengan a b entre sus k vecinos más cercanos. A este tipo de consulta se la denomina *consulta de k vecinos más cercanos inversos* y se la denota como $kRNN$ [3]. Entonces, se podrían armar los pares (a, b) tales que $a \in kRNN_{\mathbb{A}}(b)$. Por otra parte, aunque para calcular $\mathbb{A} \bowtie_k \mathbb{B}$ se podrían obtener algoritmos basados en consultas de k vecinos más cercanos de elementos de $\mathbb{A}$ sobre la base de datos $\mathbb{B}$, posiblemente serían muy ineficientes porque sólo se están buscando k, y no $|\mathbb{A}|$ k, pares de elementos.

2.3. Estrategias de Búsqueda en Espacios Métricos

Para resolver una consulta por similitud, se deben encontrar todos aquellos elementos en $\mathbb{X}$ que sean similares a un elemento q del universo $\mathbb{U}$. Para ello simplemente se podría recorrer todo el conjunto $\mathbb{X}$ de principio a fin calculando la distancia de cada elemento a q (*algoritmo de fuerza bruta*). Sin embargo, se compararía a q con una gran cantidad de elementos que no hubiera tenido sentido considerar por la enorme distancia que tienen a q, y ésas son evaluaciones o consultas que es posible ahorrarse si los

[3] Formalmente, dado un objeto $q \in \mathbb{U}$ y $k \in \mathbb{N}$, $kRNN(q)$ recupera un conjunto $A \subseteq \mathbb{X}$ tal que $\forall x \in A, q \in kNN(x)$.

elementos en $\mathbb{X}$ hubiesen tenido algún tipo de organización o información que permitiera descartarlos sin necesidad de compararlos directamente con q.

En consecuencia, es necesario preprocesar el conjunto $\mathbb{X}$ para construir una estructura de datos o *índice*, de manera tal de disminuir la cantidad de evaluaciones de distancia llevadas a cabo durante las búsquedas. Los algoritmos que permiten realizar esta tarea son los llamados *algoritmos de indexación*, y es importante que éstos sean lo más eficientes posible. Todos los algoritmos de indexación particionan de alguna manera el conjunto $\mathbb{X}$ en subconjuntos más pequeños. Se construye el índice para permitir determinar un conjunto de *subconjuntos candidatos* donde se pueden encontrar los elementos que forman parte de la respuesta a la consulta. Durante la consulta, se busca en el índice para encontrar los subconjuntos relevantes, para luego inspeccionarlos exhaustivamente.

Todas estas estructuras trabajan sobre la base de descartar elementos usando la desigualdad triangular (la única propiedad que nos permite ahorrar evaluaciones de distancia). Notar que el preproceso para indexar $\mathbb{X}$ insumirá un tiempo, pero vale la pena invertirlo ya que se realiza una única vez para un mismo conjunto $\mathbb{X}$ y, en cambio, se espera que se efectúen varias consultas por similitud sobre el mismo índice.

El tiempo total T para responder a una consulta se puede calcular mediante la fórmula:

$$T = \textbf{N}^{\textbf{o}}\ \textbf{evaluaciones de}\ d()\ \textbf{*}\ \textbf{complejidad de}\ d()\ \textbf{+ tiempo extra CPU + tiempo E/S}$$

Por supuesto, se desea mantener un T tan pequeño como sea posible, para que se consideren eficientes las búsquedas. Pero, sucede que en algunos casos evaluar $d()$ es tan costoso que todas las demás componentes de la fórmula de costo se pueden ignorar. Sin embargo, cuando se trabaja con conjuntos de datos masivos donde es necesario almacenar datos y/o índice en memoria secundaria, la componente de tiempo de E/S también afecta significativamente los tiempos de resolución de las consultas y no se puede ignorar. Aunque ésta sería la ecuación utilizada para calcular los tiempos o costos de búsquedas en el presente trabajo, la medida de complejidad de los algoritmos que se utilizará será la **cantidad de evaluaciones de la función de distancia** si sólo se trabaja en memoria principal y la **cantidad de evaluaciones de la función de distancia** y la **cantidad de operaciones de E/S** cuando se usa almacenamiento sobre memoria secundaria, dejando de lado el costo de uso de CPU y otras medidas posibles.

Los criterios utilizados para descartar objetos y así no tener que compararlos con el objeto de búsqueda, son un elemento clave para reducir el número de evaluaciones de la función de distancia cuando se utilizan métodos de búsqueda por similitud sobre espacios métricos.

2.4. Algoritmos de Indexación

Como se ha mencionado, todos los algoritmos de indexación particionan el conjunto o base de datos $\mathbb{X}$ en subconjuntos. Se construye el índice para determinar un conjunto de subconjuntos candidatos, que pueden tener elementos de interés para la consulta. Entonces, ante una consulta, se busca en el índice para encontrar los subconjuntos relevantes y luego se inspeccionan exhaustivamente todos ellos. El costo de encontrar los subconjuntos candidatos es conocido como la *complejidad interna* del algoritmo de búsqueda, y el costo de examinar exhaustivamente esos subconjuntos candidatos es la denominada *complejidad externa* del algoritmo. Cabe recordar también que todas estas estructuras trabajan sobre la base de descartar elementos usando la desigualdad triangular.

Los algoritmos de indexación en espacios métricos generales se pueden dividir de acuerdo a diferentes criterios. Se mencionan aquí algunos de los criterios que son aplicables en este trabajo:

Por el tipo de estrategia de indexación que usan: *algoritmos basados en pivotes* y *algoritmos basados en particiones compactas* (ver [CNBYM01] para un análisis más detallado).

Por el tipo de almacenamiento para el que están diseñados: *para memoria principal* y *para memoria secundaria*.

Por el tipo de respuesta que dan a las consultas: *algoritmos de búsqueda por similitud exactos* y *algoritmos de búsqueda por similitud aproximados*.

Por si admiten o no actualizaciones en la base de datos: *algoritmos estáticos* y *algoritmos dinámicos*.

Así, en el caso general, un algoritmo de indexación va a pertenecer a una de las clases de acuerdo al criterio que se considere. Sin embargo, puede suceder que un algoritmo combine alternativas de la misma clase, porque posea características combinadas. Por ejemplo, existen algoritmos que combinan *pivotes* con *particiones compactas*, como el *PM–tree* [SPS04]; o algoritmos que permiten resolver eficientemente consultas por similitud exactas, pero también son capaces de responder eficientemente a consultas por similitud aproximadas, como las propuestas para *LC* planteadas en [BN04].

El objetivo común de cualquiera de estos algoritmos es reducir al mínimo el número de evaluaciones de distancia necesarias para responder a las consultas. Por lo tanto, los criterios utilizados en cada caso para descartar objetos y así no tener que calcular su distancia al elemento de la consulta q, son un elemento clave para reducir el número de evaluaciones de distancia.

2.4.1. Estrategias de Indexación

Como se ha mencionado, las estrategias de indexación más comunes son las *basadas en pivotes* y las *basadas en particiones compactas* [CNBYM01], aunque también existen algoritmos que combinan ambas. Se analizan y describen brevemente aquí cada una de ellas, junto con los criterios que en ellas se emplean para evitar algunos cálculos de distancia.

Algoritmos Basados en Pivotes

La idea es utilizar un conjunto $\mathbb{P} = \{p_1, p_2, \ldots, p_m\} \subseteq \mathbb{U}$ de m elementos distinguidos llamados "pivotes" y almacenar, para cada elemento x de la base de datos, su distancia a los m pivotes $(d(x, p_1), \ldots, d(x, p_m))$. Se suele llamar a las distancias $(d(x, p_1), \ldots, d(x, p_m))$ la *firma* de x y se denota como $\sigma(x)$.

Dada una consulta (q, r), se calcula su distancia a los m pivotes: $\sigma(q) = (d(q, p_1), \ldots, d(q, p_m))$; es decir, la firma de q. Luego para cada elemento $x \in \mathbb{X}$, si para algún pivote p_i se cumple que $|d(q, p_i) - d(x, p_i)| > r$, entonces por la desigualdad triangular se sabe que $d(q, x) > r$ y, por lo tanto, no es necesario evaluar explícitamente $d(x, q)$ porque su distancia desde q será mayor que r (ver Lema 1). Los elementos que no se puedan descartar por medio de esta regla se deben comparar directamente contra la consulta q.

Otra manera en que puede verse esta estrategia es considerando que se transforma la base de datos original $\mathbb{X}$ en otra, que está incluida en $\mathbb{R}^m$, compuesta de las firmas de los objetos de $\mathbb{X}$; es decir, vectores de m distancias para cada uno de los $x \in \mathbb{X}$. En este nuevo espacio transformado, se puede definir una nueva función de distancia $D : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}^+$ de la siguiente manera:

$$D(\sigma(x), \sigma(y)) = \max_{1 \leq i \leq m} |d(x, p_i) - d(y, p_i)| \ \ \forall x, y \in \mathbb{X}$$

De esta manera, se puede enunciar el siguiente resultado útil:

Lema 3 *Dado un conjunto de m pivotes $\mathbb{P} = \{p_1, p_2, \ldots, p_m\}$, para todo $x, y \in \mathbb{X}$, tal que $\sigma(x) = (d(x, p_1), \ldots, d(x, p_m))$ y $\sigma(y) = (d(y, p_1), \ldots, d(y, p_m))$, se cumple que:*

$$D(\sigma(x), \sigma(y)) \leq d(x, y)$$

Demostración:

Por aplicación del Lema 1, tomando a p_i como z, se tiene que:

$$|d(x, p_i) - d(p_i, y)| \leq d(x, y)$$

Como esta desigualdad se cumple cualquiera sea el pivote $p_i \in \mathbb{P}$, en particular se va a cumplir para el pivote p_j que maximice la diferencia $|d(x, pj) - d(p_j, y)|$; es decir, para $\max_{1 \leq i \leq m} |d(x, p_i) - d(y, p_i)|$. Pero como ésa es la definición de $D(\sigma(x), \sigma(y))$, entonces reemplazando se llega a que:

$$D(\sigma(x), \sigma(y)) = \max_{1 \leq i \leq m} |d(x, p_i) - d(y, p_i)| \leq d(x, y)$$

□

En particular, esta medida de distancia D puede considerarse como la distancia L_∞ sobre el espacio transformado de $\mathbb{R}^m$. Si ya se ha calculado la firma para cada elemento x de la base de datos y se conocen las distancias de q a los m pivotes $\sigma(q) = (d(q, p_1), \ldots, d(q, p_m))$, que son las distancias que intervienen en el cálculo de las cotas inferiores, puede determinarse si un x se debe descartar o no de la lista de candidatos sin realizar cálculos de distancia adicionales, porque está garantizado que si $D(\sigma(x), \sigma(q)) \leq d(x, q)$ y además se cumple que $D(\sigma(x), \sigma(q)) > r$, entonces se puede afirmar que $d(x, q) > r$.

Un índice genérico de pivotes almacena para cada $x \in \mathbb{X}$ su firma, o sea su distancia a los m pivotes de $\mathbb{P}$, $(d(x, p_1), \ldots d(x, p_m))$, por lo que un índice consiste de mn distancias, siendo $|\mathbb{X}| = n$. En una búsqueda es necesario computar la firma de q, que son las m distancias $d(q, p_i)$, entre los pivotes y la consulta, para poder aplicar la condición de exclusión. Así, la complejidad interna del algoritmo es una complejidad fija m, si existe un número fijo de pivotes. La lista de elementos candidatos $\{x_1, \ldots, x_s\} \subseteq \mathbb{X}$ que no pueden ser descartados por la condición de exclusión, deben ser comparados directamente contra la consulta q, porque el hecho de que no se los pueda descartar no significa que estarán necesariamente en la respuesta. Esas s evaluaciones de distancia $d(x_j, q)$, con $1 \leq j \leq s$, corresponden a la complejidad externa del algoritmo. La complejidad total del algoritmo de búsqueda es la suma de ambas complejidades, $m + s$ evaluaciones de distancia.

Por lo tanto, en este índice genérico de pivotes al aumentar el número de pivotes m crece el espacio necesario y la complejidad interna m del algoritmo, pero como mejoran los límites inferiores usados para descartar elementos sin necesidad de calcular distancias, es esperable que decrezca la complejidad externa s. Sin embargo, el uso de demasiados pivotes puede hacer que la complejidad interna sea demasiado grande para ser conveniente. Por consiguiente se dice que existe un compromiso, frente a la eficiencia en las búsquedas, entre espacio y tiempo. Generalmente, la cantidad de pivotes es restringida por el espacio de memoria disponible para almacenar la estructura de datos, aunque en otros casos implícitamente puede restringirse por la distribución de distancias del espacio [BFPR06].

Al conocer la distancia de un pivote $p_i \in \mathbb{P}$ a todos los elementos de la base de datos, cuando se considera una consulta por rango (q, r), se pueden clasificar los elementos en tres grupos: (1) los que

están cerca de p_i pero su distancia a q va a ser mayor que r, (2) los que pueden estar a distancia a lo más r de q y (3) los que están lejos de p_i y su distancia a q será mayor que r. Los elementos de las zonas (1) y (3) se descartarán sin compararse directamente con q. La zona (2) contendrá el conjunto de los elementos candidatos a estar incluidos en la respuesta, porque a esos elementos dicho pivote no es capaz de descartarlos.

La Figura 2.5 ilustra la situación para el pivote p_i, resaltando la zona que no se puede descartar (parte coloreada), formada por una corona circular con radio inferior $d(p_i, q) - r$ y radio superior $d(p_i, q) + r$. Esta zona contiene todos los elementos que para el pivote p_i pueden cumplir con el criterio de búsqueda (el conjunto $\{x_3, x_4, x_5\}$), lo cual incluye a todos los elementos que efectivamente estarían en la zona de consulta (q, r) (en este caso sólo x_5) y a otros que pueden no estarlo (en este caso x_3 y x_4), lo que en la figura se traduce en que la corona circular contiene al círculo que representa la búsqueda por rango (q, r). Los elementos a descartar serían $\{x_1, x_2, x_6, x_7\}$.

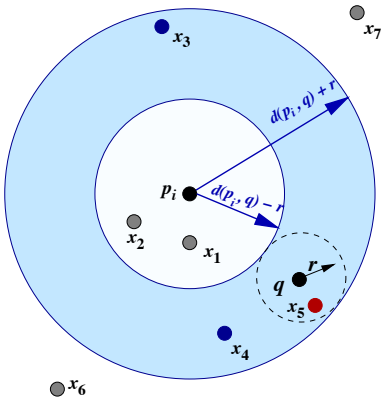


Figura 2.5: Zonas para el pivote p_i respecto de una consulta (q, r).

Si a un elemento $x \in \mathbb{X}$ no se lo puede descartar mediante un pivote $p_i \in \mathbb{P}$, se prueba descartarlo por otro; ya que por desigualdad triangular, para cada elemento x se pueden obtener tantas cotas inferiores como pivotes tenga $\mathbb{P}$. Entonces, si $|\mathbb{P}| = m$, hay m posibilidades de descartar a cada elemento de $\mathbb{X}$. Por lo tanto, considerando el Lema 3, en una consulta (q, r) un elemento $x \in \mathbb{X}$ aparecerá en la lista de candidatos si se cumple que:

$$D(\sigma(x), \sigma(y)) = \max_{p_i \in \mathbb{P}} |d(q, p_i) - d(p_i, x)| \leq r$$

Como se mencionó, cada pivote genera sus propias zonas, por lo cual a medida que se consideran otros pivotes, se pueden descartar más elementos porque los objetos que no se pueden descartar son los que pertenecen a la intersección de los anillos que representan las zonas de no descarte de cada pivote. Es decir, cuando se van aplicando diferentes pivotes disminuye la zona total a considerar para dicha consulta, porque es la intersección de todas ellas, con lo cual se disminuye la lista de candidatos. Entonces, la utilización combinada de las zonas de no descarte del conjunto total de pivotes $\mathbb{P}$ permite descartar con seguridad más elementos. La Figura 2.6 muestra el detalle de la zona a considerar para la consulta presentada (zona sombreada), que se obtiene considerando el conjunto de pivotes $\mathbb{P} = \{p_1, p_2\}$, para la consulta (q, r) dada. En este ejemplo puede observarse cómo se puede reducir drásticamente la zona en la que no pueden descartarse elementos, lo cual redundaría posiblemente también en una reducción significativa de la cardinalidad de la lista de candidatos.

Existen distintas propuestas, además de la selección aleatoria, para cómo elegir el conjunto $\mathbb{P}$ de pivotes [BNC03, BFPR06]. El número de pivotes a elegir, como ya se mencionó, puede determinarse como un número fijo m en función del espacio disponible, o como una cantidad variable en función de

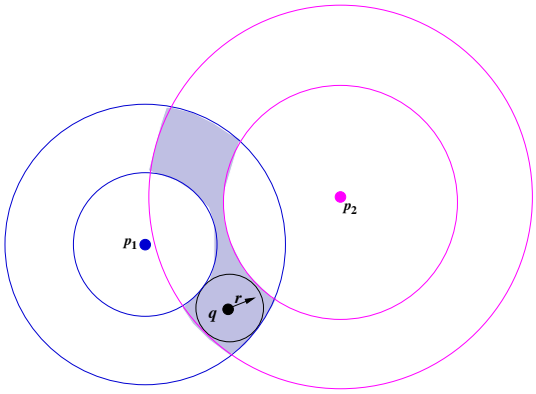


Figura 2.6: Zona no descartada conjunta para el conjunto de pivotes $\mathbb{P} = \{p_1, p_2\}$.

la distribución de distancias de la base de datos [BFPR06]. Sin embargo, otra característica que puede distinguir a los índices basados en pivotes es si usan una estructura de datos auxiliar (por ejemplo: un árbol o un arreglo) y qué datos almacenan. Además, ellos se pueden diferenciar también por si el conjunto de pivotes es *global* (es decir, para todos los elementos de la base de datos se usan los mismos pivotes) o locales (es decir, dentro de cada clase en que se divide el conjunto mediante el índice se considera un conjunto de pivotes distinto).

La elección de los pivotes juega también un rol clave en la eficiencia de las posteriores consultas. La Figuras 2.7 y 2.8 muestran dos ejemplos para la misma consulta (q, r) considerando un conjunto mayor de pivotes $\mathbb{P} = \{p_1, p_2\} \cup \{p_3\}$. Como se puede observar en Figura 2.7 la zona de elementos que no se podrán filtrar no disminuye significativamente respecto de lo filtrado con sólo $\{p_1, p_2\}$ (Figura 2.6). Por otra parte, la Figura 2.8 ilustra una situación distinta con otro conjunto de tres pivotes $\mathbb{P} = \{p_1, p_2, p_3\}$, siendo p_3 mejor seleccionado, que consigue disminuir significativamente la zona de no descarte para la consulta respecto de la situación de la Figura 2.6.

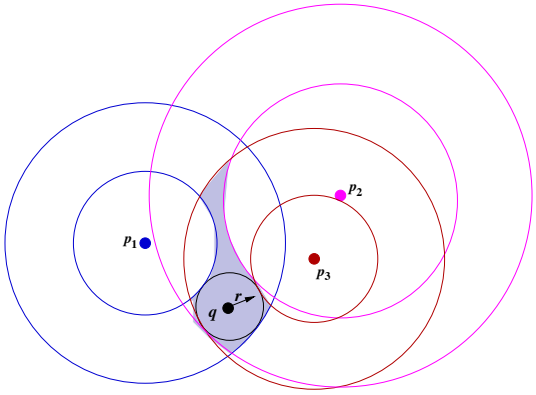


Figura 2.7: Zona de no filtrado para un conjunto de tres pivotes, con dos pivotes similares entre sí.

Algoritmos tales como *AESA* [Vid86], *LAESA* [MOV94], *Spaghettis* y sus variantes [CMBY99, NN97], *FQ-trees* y sus variantes [BYCMW94], *FQ-arrays* [CMN01] y *SSS* [BFPR06], son casi las implementaciones más directas de esta idea. Entre ellos difieren básicamente en la cantidad de pivotes que seleccionan, en la estructura extra que utilizan para reducir el costo de CPU de encontrar los puntos candidatos, en la manera de elegir los pivotes, pero no demasiado en el número de evaluaciones de distancia que realizan.

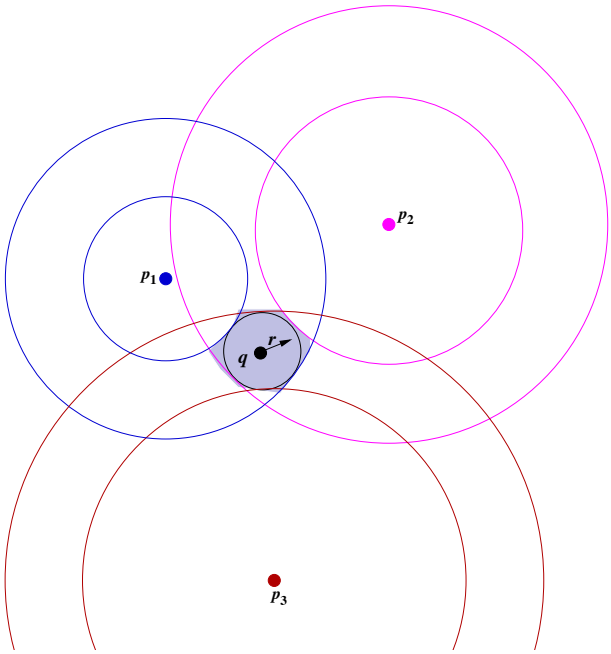


Figura 2.8: Otra zona de no filtrado para un conjunto de tres pivotes, con pivotes no similares entre sí.

Algoritmos Basados en Particiones Compactas

El otro enfoque para resolver las búsquedas por similitud en espacios métricos son los algoritmos basados en particiones compactas. Los mismos consisten en dividir el espacio subyacente en zonas o regiones tan compactas como sea posible. Normalmente esa división se realiza recursivamente, y se almacena un punto representativo ("centro") $c \in \mathbb{X}$ para cada zona, más algunos pocos datos extras que permitan descartar rápidamente la zona durante la búsqueda. Dada una consulta por rango (q, r), se denomina "bola" de la consulta q al conjunto de puntos que se encuentran a distancia a lo más r de q. Por lo tanto, el objetivo de este enfoque es que en el proceso de búsqueda puedan descartarse completamente todos los elementos de algunas de estas zonas, porque ellas no intersectan la bola de la consulta. Se suelen usar dos criterios para delimitar las zonas del espacio: *criterio de radio de cobertura* y *criterio de hiperplano* [CNBYM01]. En ambos casos se considera un objeto distinguido $c \in \mathbb{X}$ para cada zona, al que se denomina *centro*, en función del cual se determinará qué elementos pertenecen a dicha zona. En general, los elementos de la zona de c serán elementos similares a él.

Criterio del Radio de Cobertura: en este caso se considera que se puede particionar al espacio, respecto del centro c y de una distancia de corte o radio m, en dos subconjuntos: $\mathbb{X}_L$ (elementos internos) y $\mathbb{X}_R$ (elementos externos) [4]. El contenido de cada uno de estos conjuntos corresponde a:

- $\mathbb{X}_L = \{x \in \mathbb{X} / d(x, c) \leq m\}$
- $\mathbb{X}_R = \{x \in \mathbb{X} / d(x, c) > m\}$

La Figura 2.9 ilustra un conjunto de elementos en $\mathbb{R}^2$, utilizando distancia Euclidiana, donde se observa la partición del espacio con respecto a un objeto $c \in \mathbb{X}$ y una distancia m.

[4] Se usan aquí los subíndices L por izquierdo ("Left") y R por derecho ("Right"), por ser la posición en el histograma de distancias respecto de c de cada zona considerando la distancia m.

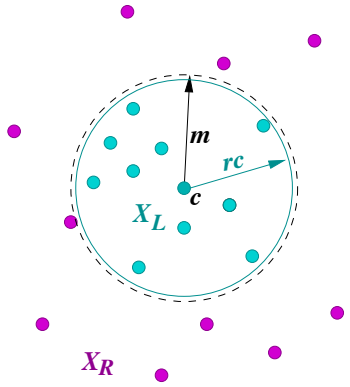


Figura 2.9: Partición de un espacio en $\mathbb{R}^2$, desde un centro c con radio m.

Considerando esta forma de particionar, cada zona almacena su centro c y su radio de cobertura $rc(c)$, el cual es la máxima distancia entre el centro c y un elemento de su zona $\mathbb{X}_L$. El número de zonas se incrementa eligiendo otros centros, o bien desde el inicio, o bien particionando recursivamente una región. Si durante una búsqueda (q, r) se cumple que $d(q, c) - r > rc(c)$, entonces no es necesario considerar los elementos de la zona correspondiente al centro c, porque no son relevantes para la consulta.

La Figura 2.10 ilustra, para el ejemplo de la Figura 2.9, la situación de una consulta (q, r) en la que la zona de centro c puede descartarse porque no puede contener elementos que puedan pertenecer a la bola de consulta. En este caso, claramente como $d(q, c) - r > rc(c)$ se puede observar que no hay intersección entre ambas zonas.

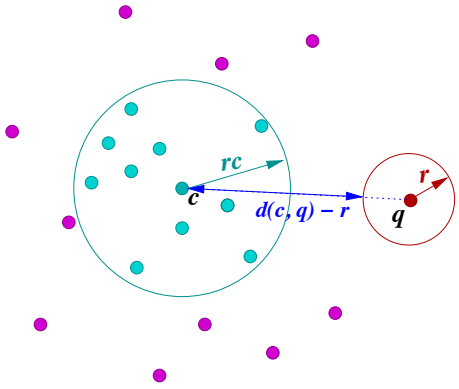


Figura 2.10: Situación de una búsqueda (q, r) que no debe examinar la zona con centro c y radio de cobertura r_c.

El siguiente lema formaliza las distintas cotas para la distancia entre un objeto de consulta q y cualquier elemento x que pertenezca a la zona con centro c y radio r_c [Are12].

Lema 4 *Sean $c, x \in \mathbb{X}$ tales que c es el centro de una zona cuyo radio de cobertura es r_c y x es un elemento de la zona de c; es decir $d(x, c) \leq r_c$. Dado $q \in \mathbb{U}$ se tiene que:*

$$\text{máx}\{d(q, c) - r_c, 0\} \leq d(q, x) \leq d(q, c) + r_c$$

Demostración:

Dado que d cumple la propiedad triangular, se tiene que:

$$d(q,c) \leq d(q,x) + d(x,c)$$

entonces despejando se cumple que:

$$d(q,c) - d(q,x) \leq d(x,c)$$

Sin embargo, por hipótesis se sabe que $d(x,c) \leq r_c$ y entonces:

$$d(q,c) - d(q,x) \leq d(x,c) \leq r_c$$

por lo tanto:

$$d(q,c) - r_c \leq d(q,x)$$

lo que combinado con las propiedades de la función de distancia, permite deducir la cota inferior:

$$\text{máx}\{d(q,c) - r_c, 0\} \leq d(q,x)$$

Nuevamente haciendo uso de la desigualdad triangular y de la definición de radio de cobertura, se logra deducir la cota superior para $d(q,x)$:

$$d(q,x) \leq d(q,c) + d(c,x) \leq d(q,c) + r_c$$

□

Por lo tanto, si el espacio se ha particionado usando el criterio de radio de cobertura, con estas cotas basta para determinar si se debe revisar o no cualquier zona con centro c y radio r_c para una consulta (q,r),

Criterio de hiperplano: este criterio es el más básico y el que se corresponde mejor con la idea de compacticidad [CNBYM01]. Se selecciona un conjunto de centros $\mathbb{C} = \{c_1, \ldots, c_m\}$ y se coloca cada elemento de la base de datos dentro de la región de su centro más cercano. Las áreas se delimitan con *hiperplanos* [5] y las zonas son análogas a las regiones de Voronoi en espacios vectoriales.

La Figura 2.11 ilustra una situación en $\mathbb{R}^2$ y con distancia Euclidiana, donde puede observarse un hiperplano que divide las zonas de dos centros c_1 y c_2. Los objetos se han coloreado dependiendo de la zona a la cual pertenecen.

Sea $\mathbb{C} = \{c_1, \ldots, c_m\}$ el conjunto de centros. La zona de c_i corresponde al conjunto $\{x \in \mathbb{X} / d(x,c_i) \leq d(x,c_j),\ \forall j \neq i\}$. En caso de empate, un elemento podrá asignarse a cualquiera de las zonas cuyos centros se encuentran a igual distancia. Durante una búsqueda (q,r) se evalúan $d(q,c_1), \ldots, d(q,c_m)$, se elige el centro $c \in \mathbb{C}$ más cercano a q y se descarta cada zona cuyo centro c_i satisfaga que $d(q,c_i) > d(q,c) + 2r$, porque su área de Voronoi no puede intersectar la "bola" de la consulta. Nuevamente aquí, el poder descartar una zona se basa también en el hecho de que la bola de consulta no intersecte dicha zona. Siendo c el centro más cercano, se puede descartar la zona de c_i si lo más cerca que puede estar un elemento de la bola de consulta respecto de c_i cae más allá de la zona de c_i (limitada por los hiperplanos). Lo más cerca que puede estar un elemento en la bola de consulta respecto de c_i sería $d(q,c_i) - r$ y si esa distancia es mayor que $d(q,c) + r$; en símbolos: $d(q,c_i) - r > d(q,c) + r$, dicha zona puede ser

[5] En particular en $\mathbb{R}^2$ el hiperplano se define como la recta que divide al plano en dos mitades.

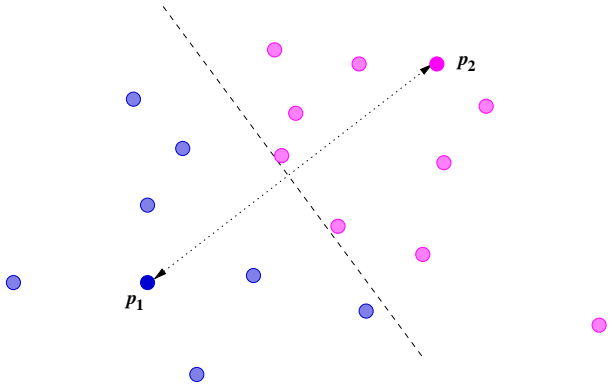


Figura 2.11: Partición por hiperplano de un espacio en $\mathbb{R}^2$, respecto de dos centros.

completamente descartada porque todos los elementos de la bola de consulta no caen en la zona de c_i. Otra manera de considerar esta condición es si $d(q, c_i) - d(q, c) > 2r$.

La Figura 2.12 muestra, para el ejemplo de la partición ilustrada en la Figura 2.11, la situación de una consulta (q, r). Como puede observarse, el centro más cercano a q es p_2. Lo más alejado que puede estar un elemento de la bola de consulta respecto de p_2 es a distancia $d(q, p_2) + r$. Lo más cerca que puede estar un elemento de la bola de consulta respecto de p_1 es a distancia $d(q, p_1) - r$. Sin embargo, como $d(q, p_1) - r > d(q, p_2) + r$ todos los elementos en la bola de consulta están más cerca de p_2 que de p_1 y así la bola de consulta no intersecta a la zona de p_1 y puede descartarse completamente.

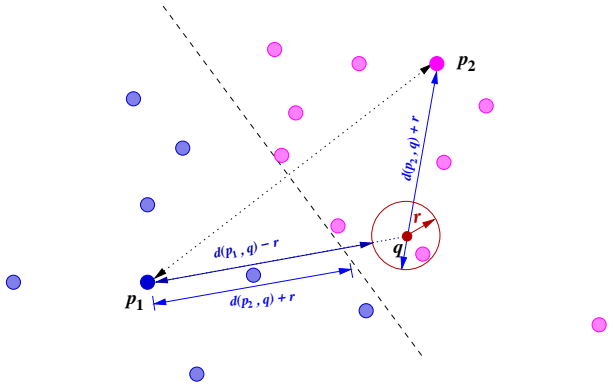


Figura 2.12: Situación de una búsqueda (q, r) que no debe examinar la zona correspondiente al centro p_1, de acuerdo al criterio de hiperplano.

El siguiente lema permite determinar una cota inferior para utilizar en las búsquedas, cuando se ha particionado el espacio por hiperplanos [ZADB06].

Lema 5 *Sean los objetos $x, c, c_i \in \mathbb{U}$ tal que se cumple que $d(c, x) \leq d(c_i, x)$. Dado un objeto de consulta q y conociendo las distancias $d(q, c)$ y $d(q, c_i)$, se puede establecer una cota inferior para $d(q, x)$ mediante:*

$$\text{máx}\left\{\frac{d(q, c) - d(q, c_i)}{2}, 0\right\} \leq d(q, x)$$

Demostración: Por las propiedades de la función de distancia de desigualdad triangular y simetría se

obtiene que:

$$d(q,c) \leq d(q,x) + d(c,x)$$

despejando se tiene que:

$$d(q,c) - d(q,x) \leq d(c,x)$$

aplicando nuevamente las propiedades de desigualdad triangular y simetría:

$$d(c_i,x) \leq d(c_i,q) + d(q,x) = d(q,c_i) + d(q,x)$$

aplicando la hipótesis que x cumple que $d(c,x) \leq d(c_i,x)$, se obtiene que:

$$d(q,c) - d(q,x) \leq d(c,x) \leq d(c_i,x) \leq d(q,c_i) + d(q,x)$$

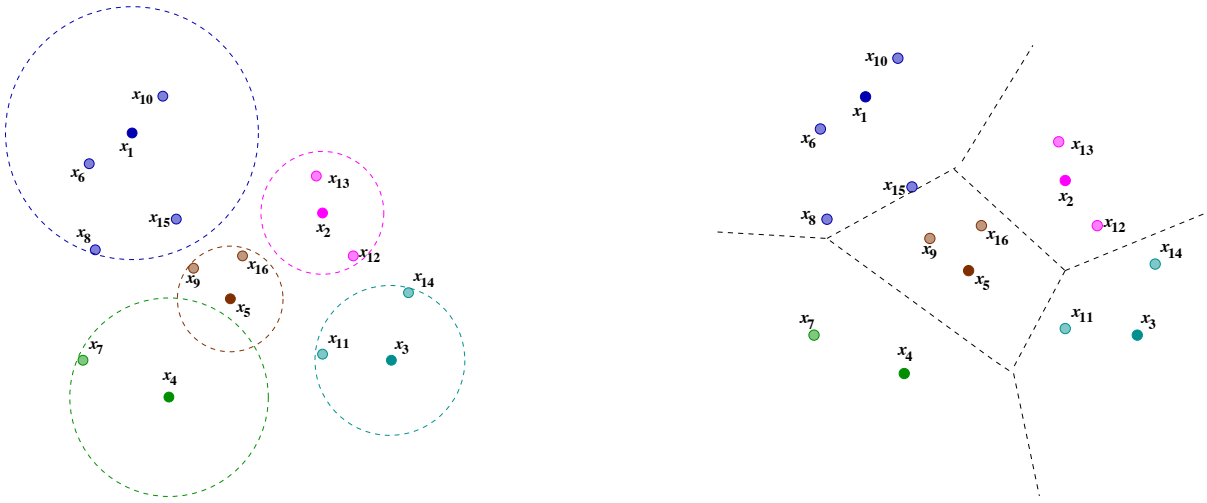
y así, aplicando transitividad de la desigualdad y despejando, se llega a que:

$$d(q,c) - d(q,c_i) \leq 2d(q,x)$$

Si $d(q,c) - d(q,c_i) > 0$, quiere decir que $d(q,c) > d(q,c_i)$, lo que implica que el punto q está más cerca de c_i que de c. En otro caso $d(q,c) - d(q,c_i) \leq 0$.

□

La Figura 2.13 ilustra ambos criterios de particionado en un espacio de ejemplo en $\mathbb{R}^2$, usando distancia Euclidiana. Para el ejemplo del criterio del radio de cobertura se muestra una partición de acuerdo a los centros $\mathbb{C} = \{x_1, x_2, x_3, x_4, x_5\}$. En el caso del ejemplo del criterio de hiperplanos, se muestra el *diagrama de Voronoi* del espacio en $\mathbb{R}^2$ utilizando el mismo conjunto de centros $\mathbb{C}$.



(a) Partición por radios de cobertura del espacio.

(b) Partición por hiperplanos del espacio.

Figura 2.13: Criterios utilizados para particiones compactas.

Recapitulando sobre las estrategias de particionado anteriormente expuestas, durante una búsqueda por rango (q,r), para descartar una zona con centro c_i en cada uno de los criterios se debe cumplir con:

- Criterio de radio de cobertura: siendo r_{c_i} el radio de cobertura de la zona:

$$d(q,c_i) - r_{c_i} > r \qquad \text{o expresado de otra forma:} \qquad d(q,c_i) - r > r_{c_i}$$

- Criterio de hiperplano: siendo c el centro más cercano a q:

$d(q, c_i) - d(q, c) > 2r$ o expresado de otra forma: $d(q, c_i) - r > d(q, c) + r$

En la Figura 2.14 se muestra la situación de búsquedas (q, r) sobre las particiones del espacio ilustradas en la Figura 2.13, de acuerdo al criterio de radio de cobertura (Figura 2.14(a)) y al de hiperplano (Figura 2.14(b)). En la Figura 2.14(a), se pueden descartar las zonas con centro x_2, x_3 y x_5, pero no las zonas de x_1 y de x_4 porque intersectan con la bola de consulta. Por otra parte, en la consulta que se muestra en la Figura 2.14(b), se pueden descartar las zonas de x_1 y x_4, pero no las zonas que corresponden a los centros x_2, x_3 y x_5. A modo de ejemplo, se repasa el motivo por el que puede descartarse la zona que se indica en cada caso:

- En la Figura 2.14(a): la zona de x_5 puede descartarse porque $d(x_5, q) - r > rc_{x_5}$, se puede observar esto gráficamente porque se ha detallado tanto el valor de $d(x_5) - r$ como el radio de cobertura de la zona de x_5.
- En la Figura 2.14(b): la zona de x_4 puede descartarse porque $d(x_4, q) - r > d(x_2, q) + r$, siendo x_2 el centro más cercano a q.

Las técnicas se pueden combinar, algunas que usan sólo hiperplanos son los *Generalized-Hyperplane Trees (GHT)* y sus variantes, y los *Voronoi Trees (VT)*. Existen técnicas que usan solamente el radio de cobertura son los *M–Trees*, las *Listas de Clusters* y los *Bisector Trees (BST)*. Los *Árboles de Aproximación Espacial* (*SAT* y *DSAT*) usan ambos criterios. Otros, como los *Geometric Near-neighbor Access Trees (GNAT)*, no utilizan ninguno de los criterios anteriores.

Combinando Particiones Compactas con Pivotes

Existen algunos índices que combinan ambas ideas dividiendo el espacio en zonas compactas y al mismo tiempo almacenando las distancias a algunos elementos distinguidos o pivotes.

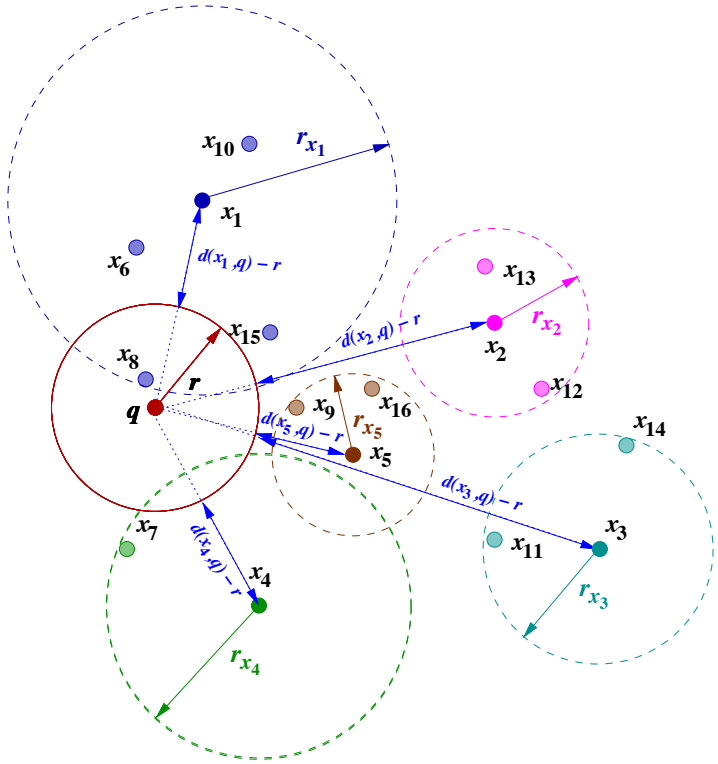
El *D-index* [DGSZ03a, Doh04] divide el espacio en particiones separables de bloques de datos y las combina con estrategias basadas en pivotes para decrementar los costos de E/S y evaluaciones de distancia realizadas durante las búsquedas. Soporta además almacenamiento en memoria secundaria, como así también eliminaciones de elementos. Sin embargo, la adaptación del *D-index* a aplicaciones particulares no es un proceso de parametrización trivial.

Otro ejemplo es el *Antipole–Tree*, presentado en [CFP+05]. Aunque los autores aseguran que soporta inserciones y eliminaciones, no se aclara cómo las pueden llevar a cabo eficientemente. Otra complicación es que la estructura no es tampoco fácil de parametrizar y los autores no mencionan cómo mantener una buena parametrización bajo un régimen dinámico de operaciones.

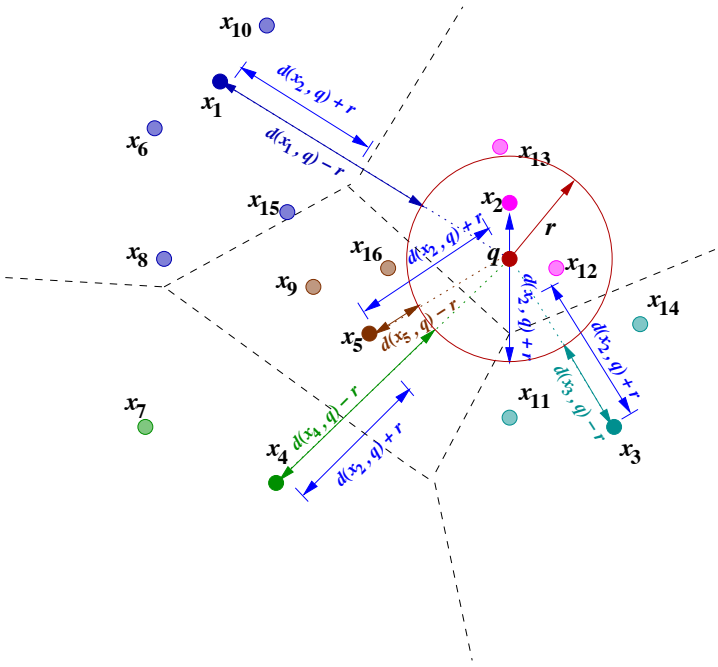
Otros ejemplos en este grupo se obtienen al agregar pivotes a alguna estructura basada en particiones compactas, tal como lo hace el *PM–Tree* [SPS04] y el *MX–Tree* [JKF13], las cuales se basan en la estructura del *M–Tree* [CPZ97].

2.4.2. Almacenamiento

Existen en la actualidad un gran número de estructuras de datos para espacios métricos. Sin embargo, la mayoría de ellas están diseñadas para memoria principal, por lo cual se preocupan principalmente de



(a) Descarte por radios de cobertura.



(b) Descarte por hiperplanos.

Figura 2.14: Descarte en una búsqueda por rango (q, r) de acuerdo a los distintos criterios.

minimizar la cantidad de evaluaciones de distancia que se realizan en construcción y en búsqueda. Sin embargo, existen aplicaciones sobre conjuntos de datos masivos donde, por el tamaño de los datos a almacenar o por la cantidad de los mismos, el índice no podrá alojarse en memoria principal sino en

memoria secundaria. Dado que el dispositivo más comúnmente utilizado para memoria secundaria es el disco, y que por su tecnología los tiempos necesarios para realizar sobre ellos operaciones de E/S no pueden despreciarse, las estructuras para memoria principal no resultan eficientes para aplicaciones sobre conjuntos masivos de datos.

Algunas pocas estructuras para búsqueda en espacios métricos han sido diseñadas especialmente para memoria secundaria, las cuales consideran la eficiencia considerando la cantidad de cálculos de distancia y a su vez la cantidad de operaciones de E/S necesarias para realizar las operaciones de interés. Entre ellas se pueden mencionar al *M–Tree* [CPZ97], y sus variantes el *Slim–Tree* [JTSF00], el *PM–Tree* [SPS04] y el *MX–Tree* [JKF13], el *D-index* [DGSZ03a], las variantes del *DSAT* para memoria secundaria *DSAT+* y *DSAT** [NR09, NR16], las variantes del *DSATCL* [BRP10] para memoria secundaria *DSATCL+*, *DSATCL** [BPR12] y el *EGNAT* [UN03, NU11] y las nuevas versiones dinámicas de *LC* [CN05] para memoria secundaria *DLC* y *DSC* [NR14, NR16] . Otras estructuras como la Aproximación Basada en Permutaciones (*PBA*) [CFN08] o *SSS* [BFPR06] admitirían una implementación eficiente en memoria secundaria.

2.4.3. Tipo de Respuesta

La mayoría de los índices existentes buscan disminuir el número de evaluaciones de distancia, y en algunos casos de operaciones de E/S, al responder a consultas por rango o de k vecinos más cercanos, en las cuales se debe responder el conjunto de "todos" los objetos de la base de datos $\mathbb{X}$ que cumplen con el correspondiente criterio de similitud. En este caso, se dice que la respuesta a las consultas por similitud es "exacta".

Sin embargo, en muchas aplicaciones es aceptable llevar a cabo una búsqueda por similitud "no exacta", donde se intercambia precisión o determinismo en la respuesta por un desempeño mejorado [ZADB06, Sam05, CNBYM01]. Esta alternativa a la búsqueda por similitud exacta se llama *búsqueda por similitud aproximada* [CP10] y comprende los algoritmos aproximados [KL04] y los probabilísticos [Cla99, KR02, CP00, CP02, CN03, BN04, CFN08].

La idea general de los algoritmos de aproximación es permitir relajar la precisión en la respuesta a una consulta a fin de obtener mayor eficiencia en la complejidad de tiempo de la consulta (evaluaciones de distancia y/u operaciones de E/S). Más aún, para la consulta se especifica un parámetro de precisión ϵ para controlar cuán lejos (en algún sentido) se admite que el resultado de la consulta se aleje del resultado correcto. Un comportamiento razonable para este tipo de algoritmos es que se aproxime asintóticamente a la respuesta correcta a medida que ϵ se aproxime a cero, y complementariamente el algoritmo de búsqueda es más rápido, perdiendo precisión, a medida que ϵ se aleja de cero. Por lo tanto, el éxito de una técnica aproximada subyace en resolver adecuadamente el compromiso entre calidad y tiempo de la respuesta [PC09].

El objetivo de este trabajo no está en los algoritmos aproximados, por consiguiente no se darán más detalles aquí. Más información sobre esta clase de algoritmos puede obtenerse en dos excelentes trabajos [PC09, WJ96].

2.4.4. Capacidades Dinámicas

La mayoría de los índices para espacios métricos están diseñados para construirse sobre un conjunto de datos estático; es decir, donde se conocen todos los elementos de antemano. En muchas aplicaciones ésta no es una situación razonable, porque se deben insertar y eliminar elementos dinámicamente. Algunas estructuras de datos soportan inserciones, pero casi ninguna soporta eliminaciones.

En [CNBYM01] se analizan las estructuras y sus capacidades dinámicas y se llega a las siguientes conclusiones:

> **Inserciones.** Entre las estructuras que allí se analizan, el *SAT* hasta ese momento *era* la menos dinámica [Nav99], porque necesitaba conocer completamente todo el conjunto para construir el índice. La familia *VP* (*VP–Tree*, *MVP–Tree* y *VP–Forest*) tiene el problema de basarse en estadísticas globales (tal como la mediana) para construir el árbol, por lo tanto luego se pueden hacer inserciones pero el desempeño de la estructura se puede degradar. Finalmente, el *FQ-array* necesita en principio insertar en la mitad de un arreglo, pero esto se puede realizar usando técnicas estándar. Todas las otras estructuras de datos pueden realizar inserciones de una manera razonable. Hay algunos parámetros de las estructuras que pueden depender de n y por ello requieren una reorganización estructural periódica.
>
> **Eliminaciones.** Las eliminaciones son un poco más complicadas. Además de las estructuras anteriores, que presentan problemas para la inserción, *BK–Trees*, *GH–Trees*, *BS–Trees*, *V–Trees*, *GNA-Trees* y la familia *VP* no admiten eliminaciones de un nodo interno del árbol, porque ellos juegan un rol esencial en la organización del subárbol. Desde luego esto se puede solucionar sólo marcando el nodo como eliminado y manteniéndolo para el propósito de elegir los caminos a seguir, pero la calidad de la estructura de datos se degrada a lo largo del tiempo. Más aún, en algunas aplicaciones no es aceptable dicha solución por cuestiones de espacio, esto puede ocurrir cuando los objetos son muy voluminosos y mantenerlos se hace imposible.

Por lo tanto, las otras estructuras además del *DSAT* [NR08], que soportarían completamente inserciones y eliminaciones son las de la familia *FQ* (*FQ–Trees* [BYCMW94], *FHQ–Trees* [BYCMW94, BY97, BYN98], *FQ-arrays* [CMN01], porque no poseen nodos internos), *AESA* [Vid86], *LAESA* [MOV94] y *SSS* [BFPR06] (porque son sólo vectores de coordenadas). También el *M–Tree*, que se diseñó con capacidades dinámicas en mente y cuyos algoritmos de inserción y eliminación hacen recordar a los del *B–Tree*,, una variante de los *GH–Trees*, el *C–Tree*, diseñada para soportar operaciones dinámicas [Ver95] y la variante del *GNAT* denominada *EGNAT* [UPN09, NU11]. Además, las basadas en el *M–Tree*, como son el *Slim–Tree* [JTSF00], el *PM–Tree* [SPS04] y el *MX–Tree* [JKF13], mantienen (o mejoran) las capacidades dinámicas de la estructura original. El *D-index* [DGSZ03a, Doh04] soporta inserciones y eliminaciones de elementos; aunque para remover elementos que actúan como pivotes es problemático. El *Antipole–Tree* [CFP$^+$05], que aunque los autores aseguran que soporta inserciones y eliminaciones, no es claro que las pueda llevar a cabo eficientemente. La *Lista de Clusters Recursiva* [Mam05], que es una versión dinámica de la *Lista de Clusters*, que aunque es capaz de soportar inserciones y eliminaciones, eliminar elementos que actúan como centros de clusters es problemático.

Más aún, existen métodos genéricos que permiten transformar una estructura de datos estática en una con capacidades dinámicas [BS80, OL81], los cuales fueron aplicados en [NH12] para obtener versiones dinámicas del *VP–Tree* [Yia93] y del *SSS–Tree* [BPS$^+$08]. Sin embargo, por ser estrategias genéricas, en algunos casos posiblemente degraden el desempeño de las inserciones (respecto de sus correspondientes costos estáticos) y las eliminaciones deben tolerar el mantener los elementos eliminados durante un tiempo, lo cual puede no ser aceptable en algunas situaciones.

2.5. Clasificación de las Soluciones Existentes

Para dar un amplio contexto al presente trabajo se mostrará, de acuerdo a la posibles clasificaciones antes mencionadas, cómo se encuadran las soluciones existentes más importantes para búsquedas exactas por similitud.

La Figura 2.15 ilustra la clasificación de los índices, planteada en [CNBYM01], con la incorporación de algunas nuevas propuestas, de acuerdo a sus características más importantes. Cabe aclarar, que esta taxonomía no pretende ser exhaustiva de todas las alternativas existentes, sino sólo la presentación de algunas de las consideradas más importantes. Además, cabe destacar que esta taxonomía se basa en la estrategia de indexación utilizada por los algoritmos (basados en pivotes o particiones compactas) y en sus posibles subcategorías.

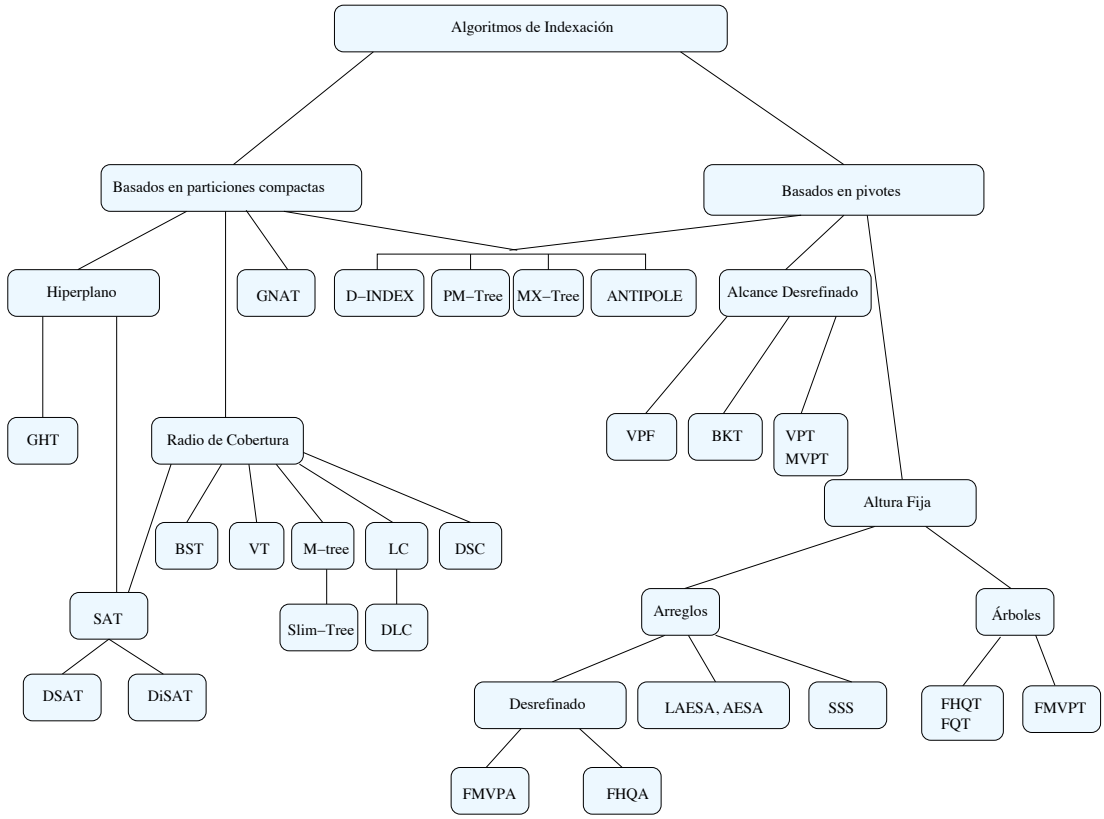


Figura 2.15: Taxonomía de los algoritmos existentes para búsqueda por similitud en espacios métricos.

Para mayor claridad en lo que sigue, en la Tabla 2.1 se muestra el nombre completo de cada una de las estructuras de datos para búsqueda por similitud en espacios métricos (que aparecen clasificadas en la Figura 2.15 desde [CNBYM01]) y algunas más nuevas, junto a la abreviatura que se usa habitualmente y la referencia del artículo en donde se propuso cada una de ellas. Además, se denota con D a aquéllas que son dinámicas, con MS las que están diseñadas para memoria secundaria y con MP las consideran su almacenamiento en memoria principal.

2.6. Técnicas de Búsqueda de los Vecinos más Cercanos

Hasta el momento se han descripto diversas soluciones para las consultas por rango. Se puede ver que la mayoría de las soluciones existentes para consultas de los vecinos más cercanos son construidas de manera sistemática sobre las técnicas de búsquedas por rango, y por ende pueden ser adaptadas a cualquiera de las estructuras de datos que se han mencionado [CNBYM01].

Nombre Abreviado	*Estructura*	*Referencia*	*Dinamismo*	*Almacenamiento*
BKT	Burkhard-Keller Trees	[BK73]	–	MP
FQT	Fixed Queries Trees	[BYCMW94]	–	MP
FHQT	Fixed Height FQ-Trees	[BYCMW94, BY97, BYN98]	–	MP
FQA	Fixed Queries Arrays	[CMBY99, CMN01]	–	MP
VPT	Vantage Point Trees	[Yia93, cC94]	–	MP
MVPT	Multi-Vantage Point Trees	[BO97]	–	MP
VP–forest	Excluded Middle Vantage Point Forest	[Yia99]	–	MP
BST	Bisector Trees	[KM83]	–	MP
GHT	Generalized-Hyperplane Trees	[Uhl91a, Uhl91b]	–	MP
GNAT	Geometric Near-neighbor Access Trees	[Bri95]	–	MP
EGNAT	Evolutive Geometric Near-neighbor Access Tree	[UPN09, NU11]	D	MS
VT	Voronoi Trees	[DN87]	–	MP
M–trees	M–Trees	[CPZ97]	D	MS
Slim–trees	Slim–Trees	[JTSF00]	D	MS
LC	List of Clusters	[CN00a, CN05]	–	MP
DLC	Dynamic List of Clusters	[NR14]	D	MS
DSC	Dynamic Set of Clusters	[NR16]	D	MS
SAT	Spatial Aproximation Trees	[Nav99]	–	MP
DSAT	Dynamic Spatial Approximation Trees	[NR02, NR08]	D	MP
AESA	Approximating Eliminating Search Algorithm	[Vid86]	–	MP
LAESA	Linear AESA	[MOV94]	–	MP
SSS	Sparse Spatial Selection	[BFPR06]	D	MP
ANTIPOLE	Antipole Trees	[CFP$^+$05]	D	MP
D-INDEX	D-Index	[DGSZ03a]	D	MS
PM–Trees	Pivoting M–Trees	[SPS04]	D	MS
MX–Trees	Extendable M–Trees	[JKF13]	D	MS

Tabla 2.1: Nombres de las estructuras de datos para búsquedas por similitud exactas en espacios métricos, las referencias a los artículos en donde se propone cada una de ellas y sus caracterśticas de capacidades dinámicas y de almacenamiento.

2.6.1. Radio Incremental

El algoritmo más simple para búsqueda de los vecinos más cercanos se basa en el uso de un algoritmo de búsqueda por rango, como se explica a continuación. Se busca con objeto de consulta q y radio fijo $r = a^i\varepsilon(a > 1)$, comenzando con $i = 0$ e incrementándolo hasta que al menos el número de elementos deseados se encuentre dentro del rango de búsqueda $r = a^{i-1}\varepsilon$. Más tarde, el radio es refinado entre $r = a^{i-1}\varepsilon$ y $r = a^i\varepsilon$ hasta que se haya incluido el número exacto de elementos.

Ya que la complejidad de una búsqueda por rango crece bruscamente con el radio de búsqueda, el costo de este método puede ser muy cercano al costo de una búsqueda en rango con el r apropiado (el cual no es conocido de antemano). Los incrementos pueden ser hechos más pequeños $(a \to 1)$ para evitar buscar con un radio mucho más grande que el necesario.

2.6.2. Backtracking con Radio Decreciente

La siguiente es una técnica más elaborada. Para la búsqueda del elemento más cercano $(1NN)$, se realiza una búsqueda por rango comenzando con radio $r = \infty$. Cada vez que q es comparada contra algún elemento $p \in S$, el radio de búsqueda es actualizado con $r \leftarrow min(r, d(q, p))$ y la búsqueda debe continuar con ese rango reducido.

A medida que se encuentran elementos más y más cercanos a q, se busca con radio más y más pequeño, por lo que la búsqueda se vuelve más “barata”. Por esta razón es importante encontrar rápidamente elementos cercanos a la consulta (algo que no es importante en las búsquedas por rango fijo). La manera de lograr esto es dependiente de la estructura de datos en particular. Al finalizar la búsqueda, r

es la distancia que hay entre q y el elemento más cercano.

Las consultas $kNN(q)$ se resuelven de manera similar, extendiendo la idea anterior y manteniendo en todo momento los k elementos más cercanos a q encontrados hasta el momento, siendo ahora el valor de r la máxima distancia entre q y alguno de esos k elementos. Cada vez que se encuentra un elemento cuya distancia es relevante, es insertado como uno de los k vecinos más cercanos conocidos hasta el momento (desplazando fuera de la lista a algún viejo candidato) y se actualiza r. Se comienza la búsqueda con $r = \infty$ y se mantiene ese valor hasta encontrar los primeros k candidatos.

2.6.3. Backtracking con Prioridad

La técnica anterior puede ser mejorada si se seleccionan de manera hábil cuáles elementos considerar primero. Por claridad se considerará que el backtracking se realiza en un árbol, aunque la idea es general. En lugar de seguir el orden normal de backtracking de una búsqueda por rango, modificando el orden en el cual se visitan los subárboles, se da mucha más libertad al orden en el cual son visitados los nodos del árbol. El objetivo es incrementar la probabilidad de encontrar rápidamente elementos cercanos a q y entonces reducir pronto el radio r^*.

En cada paso de la búsqueda se tiene un conjunto de posibles subárboles que pueden ser visitados (no necesariamente todos los subárboles son del mismo nivel). Se selecciona entre ellos un subárbol, siguiendo alguna heurística (por ejemplo, visitar primero los subárboles cuya raíz es la más cercana a q). Una vez que se ha seleccionado un subárbol, se compara a q con la raíz, se actualiza r^* y la lista de candidatos (si es necesario) y se determina cuál de los hijos de la raíz considerada merece ser visitado. A diferencia del backtracking normal, estos hijos no son inmediatamente visitados sino que son agregados al conjunto de subárboles que deben ser visitados en algún momento. Luego, se vuelve a seleccionar otro subárbol del conjunto usando la heurística de optimización.

La mejor manera de implementar esta búsqueda es con una cola de prioridades ordenada de acuerdo a la "bondad" indicada por la heurística. Los subárboles son insertados y removidos de esta cola. Se comienza la búsqueda con una cola vacía, insertando únicamente la raíz del árbol. Luego, se repite el paso de remover el subárbol más prometedor, se lo procesa y se insertan los subárboles relevantes hasta que la cola quede vacía o hasta que el límite inferior del mejor subárbol exceda a r^*.

2.7. Espacios Vectoriales

Anteriormente se aclaró que cuando los elementos del espacio métrico $(\mathbb{U}, d)$ son realmente tuplas de números reales, entonces el par se denomina *espacio vectorial*.

Un espacio vectorial de dimensión finita D es un espacio métrico particular donde los objetos se identifican por D números reales $(x_1, x_2, \ldots, x_D)$ y cada x_i es llamado "*coordenada*" del objeto. En adelante serán llamados espacios vectoriales o espacios *D-dimensionales*.

Como se ha mencionado previamente, existen distintas funciones de distancia que se pueden usar en un espacio vectorial, pero las más usadas son las de la familia de L_p o *familia de distancias de Minkowski* (definida por la ecuación 2.6): L_1 o *Distancia de Manhattan*, L_2 o *Distancia Euclidiana* y L_∞ o *Distancia de Máximo* o *distancia de Chebyshev*.

En muchas aplicaciones los espacios métricos son en realidad espacios vectoriales, es decir que los objetos son puntos D-dimensionales y la similitud se puede interpretar geométricamente. Un espacio vectorial permite más libertad al diseñar algoritmos de búsqueda, porque es posible usar la información

de coordenadas geométricas que no está disponible en los espacios métricos generales.

En el marco de los espacios vectoriales, existen algoritmos óptimos que permiten resolver búsquedas del punto más cercano. Las estructuras de datos para espacios vectoriales son llamadas métodos de acceso espacial (*SAM* en inglés). Entre las estructuras de datos más populares se pueden citar a los *KD–trees* [Ben75, Ben79], *R–trees* [Gut84], *QUAD–trees* [Sam84] y los *X–trees* [BKK96]. Todas estas técnicas hacen uso extensivo de la información de las coordenadas para agrupar y clasificar puntos en el espacio. Desafortunadamente las técnicas existentes son muy sensibles a la dimensión del espacio vectorial. Los costos de los algoritmos de búsqueda del punto más cercano tienen una dependencia exponencial respecto de la dimensión del espacio. Esto es conocido como la *maldición de la dimensionalidad*, aspecto que se mencionará a continuación.

2.8. Dimensionalidad y Dimensionalidad Intrínseca

Uno de los principales obstáculos para el diseño de técnicas de búsqueda eficiente en espacios métricos es la existencia y ubicuidad en aplicaciones reales de los llamados espacios de alta dimensionalidad. En las técnicas tradicionales de indexación como los *Kd–trees* [Ben75, Ben79] sus costos de búsqueda dependen exponencialmente de la dimensión de representación del espacio.

Existen métodos efectivos para buscar sobre espacios de dimensión finita D. Sin embargo, para 20 dimensiones o más dichas estructuras dejan de desempeñarse bien. El foco aquí, como ya se mencionó, se encuentra en espacios métricos generales aunque las soluciones planteadas son también adecuadas para espacios D–dimensionales.

En [CNBYM01, CN00b, CN01] se muestra que el concepto de dimensionalidad intrínseca se puede entender aún en espacios métricos generales, se da una definición cuantitativa de ella y se muestra analíticamente la razón para la llamada "*maldición de la dimensionalidad*".

Es interesante notar que el concepto de dimensionalidad está relacionado con la "*facilidad*" o "*dificultad*" para buscar en un espacio vectorial D-dimensional.

Los espacios de dimensionalidad alta tienen una distribución de probabilidad de distancias entre elementos cuyo histograma es más concentrado y con una media grande. Esto hace que el trabajo de cualquier algoritmo de búsqueda por similitud sea más dificultoso. En el caso extremo se tiene un espacio donde $d(x, x) = 0$ y $\forall y \neq x, d(x, y) = 1$. En este caso, la consulta q debe ser exhaustivamente comparada contra cada elemento en el conjunto. El concepto de dimensionalidad intrínseca es una extensión de esta idea. Se dice que un espacio métrico general es más "difícil"(dimensión intrínseca más alta) que otro cuando su histograma de distancia es más concentrado que el del otro.

La idea es que a medida que crece la dimensionalidad intrínseca del espacio, la media μ del histograma crece y su varianza σ^2 se reduce.

La Figura 2.16 nos muestra, de manera intuitiva, por qué los histogramas más concentrados producen espacios métricos más difíciles (alta dimensionalidad intrínseca), e inversamente por qué los histogramas menos concentrados caracterizan espacios métricos más fáciles (baja dimensionalidad intrínseca).

Sea p un elemento de la base de datos, q una consulta y $r > 0$ el radio de búsqueda. La desigualdad triangular implica que cada elemento x tal que $|d(q, p) - d(p, x)| > r$ no puede estar a distancia menor o igual que r de q, así se podría descartar a x. Sin embargo, en un histograma más concentrado las diferencias entre dos distancias aleatorias son más cercanas a cero y por tanto la probabilidad de descartar un elemento x es más baja. Las áreas sombreadas en la figura muestran los puntos que no se pueden

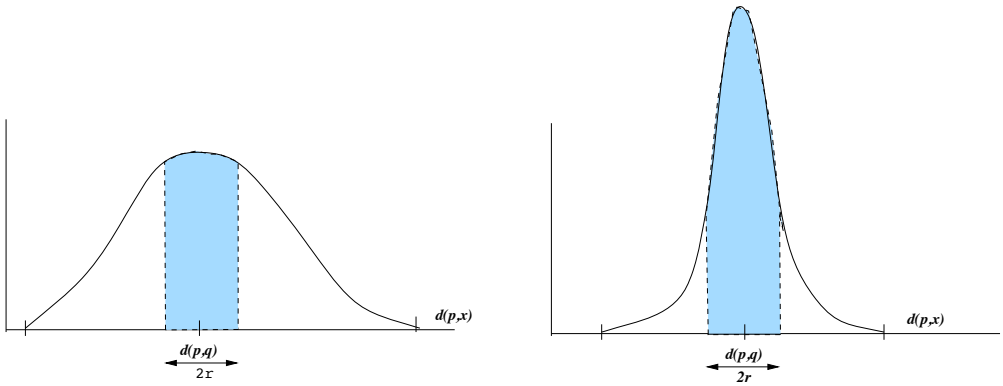


Figura 2.16: Un histograma de distancias para un espacio métrico de dimensión baja (izquierda) y de dimensión alta (derecha).

descartar. A medida que el histograma es más y más concentrado alrededor de esa media, menor cantidad de puntos se puede descartar usando la información brindada por $d(p, q)$.

Este fenómeno es independiente de la naturaleza del espacio métrico (en particular si es vectorial o no) y da una manera de determinar cuán difícil es buscar sobre un espacio métrico arbitrario.

La dimensionalidad intrínseca de un espacio métrico se define en [CN01] como $\rho = \mu^2/2\sigma^2$, donde μ y σ^2 son la media y la varianza del histograma de distancias. Lo importante de la fórmula es que la dimensionalidad intrínseca crece con la media y se reduce con la varianza del histograma. Esta definición además es coherente con la noción de dimensión en un espacio vectorial con coordenadas uniformemente distribuidas. Es decir, un espacio vectorial aleatorio con k dimensiones tiene una dimensión intrínseca $\Theta(k)$.

Si en un espacio vectorial sólo se utiliza la función de distancia para realizar las búsquedas, es decir no se utiliza la información de las coordenadas de los objetos del espacio, es sencillo simular las búsquedas en un espacio métrico general. Una ventaja adicional es que la dimensión intrínseca del espacio se muestra independiente de cualquier dimensión de representación; es decir, de la cantidad de coordenadas del vector.

En espacios métricos de dimensión muy alta, los índices en general son ineficientes. Numerosos autores se han dedicado a estudiar este hecho [BBK01], incluso se ha demostrado que si la dimensión es muy alta, la solución es lineal para muchas distribuciones [BGRS99, SR06].

Se puede evadir, de algún manera, a la maldición de la dimensionalidad utilizando algoritmos de aproximación o de respuesta no exacta. Si un espacio métrico es de alta dimensión, es posible conseguir eficiencia en las consultas relajando la precisión en la respuesta; porque, como se ha mencionado previamente, los algoritmos de aproximación logran reducir los costos de las consultas a costa de bajar la calidad de la respuesta.

2.9. Otros Ejemplos de Espacios Métricos

2.9.1. Modelo Vectorial para Documentos

En recuperación de la información [BYRN99, BYRN11] se define un *documento* como una "unidad de recuperación", la cual puede ser un párrafo, una sección, un capítulo, una página Web, un artículo o un

libro completo. Los modelos clásicos en recuperación de la información consideran que cada documento está descripto por un conjunto representativo de palabras claves llamadas *términos*, que son palabras cuya semántica ayuda a definir los temas principales del documento.

Uno de estos modelos, el *modelo vectorial*, considera un documento como un vector t-dimensional, donde t es el número total de términos de la colección. Cada coordenada i del vector está asociada a un término del documento, cuyo valor corresponde a un "peso" positivo w_{ij} si es que dicho término (el término i) pertenece al documento j y 0 en caso contrario. Si $\mathbb{D}$ es el conjunto de documentos y d_j es el j-ésimo documento perteneciente a $\mathbb{D}$, entonces $d_j = (w_{1j}, w_{2j}, \ldots, w_{tj})$.

En el modelo vectorial se calcula el *grado de similitud* entre un documento d y una consulta q, la cual se puede ver como un conjunto de términos o como un documento completo, como el grado de similitud entre los vectores $\vec{d_j}$ y $\vec{q}$. Esta correlación puede ser cuantificada, por ejemplo, como el coseno del ángulo formado entre ambos vectores:

$$sim(\vec{d_j}, \vec{q}) = \frac{\vec{d_j} \cdot \vec{q}}{|\vec{d_j}| \times |\vec{q}|} = \frac{\sum_{i=1}^{t} w_{ij} \times w_{iq}}{\sqrt{\sum_{i=1}^{t} w_{ij}^2 \times \sum_{i=1}^{t} w_{iq}^2}}$$

donde w_{iq} es el peso del i-ésimo término en la consulta q.

Los pesos de los términos pueden calcularse de varias formas. Una de las más importantes es utilizando *esquemas tf-idf*, en donde los pesos están dados por:

$$w_{ij} = f_{i,j} \times \log\left(\frac{N}{n_i}\right)$$

donde N es el número total de documentos, n_i es el número de documentos en donde el i-ésimo término aparece y $f_{i,j}$ es la *frecuencia normalizada* del i-ésimo término, dada por:

$$f_{i,j} = \frac{freq_{i,j}}{\text{máx}_{\ell=1\ldots t}(freq_{\ell,j})}$$

donde $freq_{i,j}$ es la frecuencia del i-ésimo término en el documento d_j, y $\text{máx}_{\ell=1\ldots t}(freq_{\ell,j})$ es el máximo valor de frecuencia sobre todos los términos contenidos en d_j.

Por lo tanto, si se considera que los documentos son puntos en un espacio métrico, el problema de la búsqueda de documentos similares a una consulta dada se reduce a una búsqueda por similitud en el espacio métrico correspondiente. Se utiliza en este caso como medida de similitud a la distancia coseno que mide el ángulo formado entre los vectores $\vec{d_j}$ y $\vec{q}$ [BYRN99, BYRN11]. Por lo tanto, $(\mathbb{D}, d)$ es un espacio métrico.

Como ejemplo de esta clase de espacios métricos, se ha utilizado para los experimentos un espacio métrico de documentos con la función de distancia coseno definida en Sección 2.1.1.

2.9.2. Diccionarios

Otro ejemplo posible de espacio métrico, que también se ha utilizado en los experimentos de este trabajo, corresponde a un conjunto de palabras o diccionario. En este caso los objetos son palabras (o strings) en un determinado lenguaje (por ejemplo, Inglés, Español, Francés, Italiano, etc.).

Para medir la similitud entre palabras se utiliza la función de distancia conocida como *distancia de edición* (o *distancia de Levenshtein*), definida en la Sección 2.1.1.

Si se consideran las palabras como puntos de un espacio métrico, el problema de buscar palabras similares a otra palabra dada, se reduce a una búsqueda por similitud en el espacio métrico. El caso particular de un diccionario es de interés en aplicaciones de correctores ortográficos y tiene muchas otras aplicaciones en recuperación de texto, procesamiento de señales y biología computacional [Nav01, Boy11].

2.10. Espacios Métricos Utilizados en la Evaluación Experimental

Se describen aquí los principales espacios métricos utilizados en este trabajo para realizar la evaluación experimental de las distintas propuestas. Algunos de ellos surgen desde aplicaciones reales y otros, llamados *sintéticos*, son generados con alguna distribución particular. Dentro de los espacios reales, se han seleccionado tres bases de datos métricas, usadas como puntos de referencia, todas descargadas desde la Biblioteca Métrica de SISAP [FNC07], disponibles desde `www.sisap.org`:

- *Imágenes de la NASA:* un conjunto de 40.700 vectores de características de dimensión 20, generados desde imágenes descargadas de la NASA. Se usa la distancia Euclidiana.

- *Diccionario:* un diccionario de 69.069 palabras en inglés. La distancia es la *distancia de edición* o *distancia de Levenshtein*. Esta distancia es útil, como ya se mencionó, en recuperación de texto para hacer frente a errores ortográficos, de deletreo y de reconocimiento óptico de caracteres (OCR).

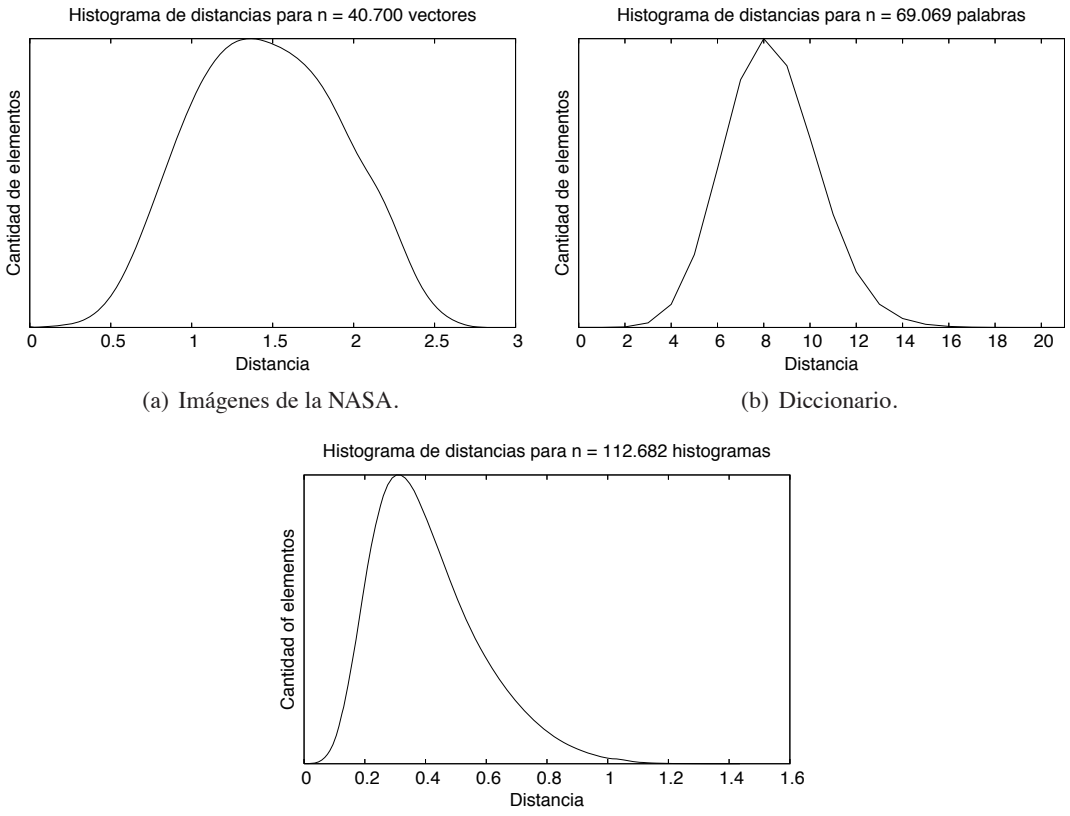
- *Histogramas de Color:* un conjunto de 112.682 vectores de características de dimensión 112, obtenidos desde histogramas de color desde una base de datos de imágenes. Cualquier forma cuadrática se puede usar como distancia, por lo tanto se ha elegido la distancia Euclidiana como la alternativa útil más simple.

La Figura 2.17 muestra los histogramas de distancia para cada espacio métrico considerado. Estos espacios son ejemplos interesantes de espacios métricos reales, con histogramas distintos (gaussiano, discreto y sesgado hacia la cola).

Además para investigar la escalabilidad de los índices, se ha considerado un conjunto de datos reales obtenido desde la colección de imágenes de SAPIR (Search in Audio Visual Content Using Peer-to-peer Information Retrieval) [FKM$^+$07]. Los descriptores basados en contenido extraídos desde las imágenes consisten en vectores de 208 componentes. La distancia es Euclidiana. La base de datos cuenta con un millón de imágenes. Este conjunto es un subconjunto de **CoPhIR** [BEF$^+$09], que contiene alrededor de 106 millones de imágenes extraídas desde Flickr y provistas por la colección *SAPIR*. En este trabajo a esta base de datos se la denomina *Flickr*.

Con el fin de analizar cómo la dimensionalidad intrínseca afecta el comportamiento de las búsquedas en los diferentes índices, se evalúan las propuestas sobre espacios métricos sintéticos donde se puede controlar la dimensión intrínseca. Se generan colecciones de vectores de distintas dimensiones, uniformemente distribuidos en el hipercubo unitario. En todos los casos de uso, explícitamente no se utiliza la información de coordenadas de cada vector, por considerar a cada vector como un objeto de un espacio métrico.

En algunos experimentos se han considerado otros espacios métricos, los cuales se describirán previamente a su utilización.



(a) Imágenes de la NASA.

(b) Diccionario.

(c) Histogramas de Color.

Figura 2.17: Histogramas de distancia de los espacios métricos descargados de SISAP.

2.11. Detalles Generales de los Experimentos

Para la evaluación experimental de las distintas propuestas que construyen índices para resolver consultas por similitud clásicas, se analizan los costos de construcción de los índices y los costos de las búsquedas por rango. Cabe aclarar que por la existencia de algoritmos de rango óptimo para la búsqueda de k-vecinos más cercanos y como la principal medida de complejidad considerada en este trabajo es el número de evaluaciones de la función de distancia, basta con considerar sólo las búsquedas por rango [6].

Para analizar los costos de construcción se construyen los índices con el 100 % de los elementos de la base de datos y los resultados que se muestran son el promedio de 10 ejecuciones de construcción del índice, usando diferentes permutaciones de la base de datos. Para analizar los costos de búsqueda, en todos los casos, se construye el índice con el 90 % de los objetos y se usa el otro 10 % (elegido aleatoriamente) como consultas. Todos los resultados son promediados sobre 10 ejecuciones de construcción del índice, usando diferentes permutaciones de los conjuntos de datos. Se han considerado consultas por rango que recuperan en promedio 0.01 %, 0.1 % y 1 % de la base de datos. Esto corrresponde a radios de 0,12, 0,285 y 0,53 para las Imágenes de la NASA, 0,051768, 0,082514 y 0,131163 para los Histogramas de Color. El Diccionario tiene una distancia discreta, por lo tanto se usan los radios desde 1 a 4, los cuales recuperan en promedio 0.00003 %, 0.00037 %, 0.00326 % y 0.01757 % de la base de datos respectivamente. Las mismas consultas son usadas para todos los experimentos sobre los mismos conjuntos de datos.

[6] Si se consideran otras medidas de complejidad, como por ejemplo el tiempo extra de CPU, los tiempos de ejecutar una búsqueda de k-vecinos más cercanos y una búsqueda por rango con el radio adecuado podrían ser distintos.

Capítulo 3

Estructuras Previas

Como existen numerosas aplicaciones en las que se puede utilizar el modelo de espacios métricos para resolver búsquedas por similitud, existe un amplio conjunto de métodos disponibles. Algunos de ellos aplican directamente los conceptos básicos en el área y otros combinan conceptos tratando de sacar el mayor provecho posible de las distancias que se almacenan o se calculan, con el fin de disminuir los costos de las búsquedas.

El objetivo de este capítulo no es dar un panorama general de los métodos disponibles, para lo cual pueden consultarse libros y surveys existentes [CNBYM01, Sam05, ZADB06], sino sólo de describir aquéllos que se han tomado como base en este trabajo. Todos ellos corresponden a técnicas basadas en particiones compactas. En secciones posteriores, cuando se realice la comparación de propuestas con otros índices existentes, se realizará una breve descripción de los mismos.

3.1. Árbol de Aproximación Espacial

La mayoría de los métodos existentes para búsqueda por similitud en espacios métricos se basan en dividir la base de datos, lo que se ha heredado desde las ideas clásicas de *dividir para conquistar* y de la búsqueda de datos típicos (v.g. árboles de búsqueda binaria). Sin embargo, se describe aquí una técnica con una aproximación diferente, que es específica de la búsqueda espacial, propuesta en [Nav99, Nav02]. Más que dividir el conjunto de candidatos durante la búsqueda, se trata de comenzar la búsqueda en algún punto del espacio y acercarse espacialmente al elemento de consulta q, encontrando elementos cada vez más cercanos a q. Para introducir el Árbol de Aproximación Espacial (*SAT*, por su nombre en inglés: *Spatial Approximation Trees*) es necesario entender en qué consiste la *aproximación espacial*.

3.2. Acercamiento a la Aproximación Espacial

Para entender qué es la aproximación espacial se analizan las consultas de 1-*vecino más cercano* o $1-NN$ (después se analizará cómo resolver todos los tipos de consultas). En lugar de los algoritmos conocidos para resolver las consultas por proximidad dividiendo el conjunto de candidatos, se trabaja con una aproximación diferente. En este modelo, se está ubicado en un elemento de $\mathbb{X}$ dado y se trata de acercar "espacialmente" al elemento consultado q. Cuando no se puede hacer más este movimiento, significa que se está posicionado en el elemento más cercano a q en todo el conjunto.

Las aproximaciones son realizadas sólo vía "vecinos". Cada elemento del conjunto $a \in \mathbb{X}$ tiene un conjunto de vecinos $N(a)$, y está permitido moverse sólo a los vecinos. La estructura natural para representar esta restricción es un grafo dirigido. Los nodos son los elementos del conjunto $\mathbb{X}$ y están conectados a sus vecinos por arcos. Más específicamente, existe un arco desde a hasta b si es posible moverse de a a b en un único paso.

Cuando tal grafo está definido adecuadamente, el proceso de búsqueda para un elemento de consulta q es simple: se comienza la búsqueda posicionado en un nodo aleatorio a y se consideran todos sus vecinos. Si ningún vecino está más cerca de q que de a, entonces, se entrega al elemento a como el vecino más cercano a q. En otro caso, se selecciona algún vecino b que esté más cerca de q que a y se continúa desde b.

Para que el algoritmo funcione, el grafo debe contener suficientes arcos. El grafo más simple que se puede utilizar es el grafo completo; es decir, todos los pares de nodos son vecinos. Sin embargo, esto implica hacer n evaluaciones de distancia sólo para verificar los vecinos del primer nodo. Por lo tanto, es preferible aquel grafo que tenga el *menor* número posible de arcos y que aún así permita responder correctamente a todas las consultas. Este grafo $G = (\mathbb{X}, \{(a,b), a \in \mathbb{X}, b \in N(a)\})$ debe cumplir la siguiente propiedad:

Propiedad 1 $\forall a \in \mathbb{X}, \forall q \in \mathbb{U}$, *si* $\forall b \in N(a),\ d(q,a) \leq d(q,b)$, *entonces* $\forall b \in \mathbb{X},\ d(q,a) \leq d(q,b)$.

Esto significa que, dado *cualquier* elemento q, si no se puede acercar más a q desde a yendo a través de sus vecinos, entonces es porque a ya es el elemento más cercano a q en todo el conjunto $\mathbb{X}$. Es claro que si el grafo G satisface la Propiedad 1 se puede buscar por aproximación espacial. Por lo tanto, se busca un grafo de esta clase que sea minimal (en cantidad de arcos).

Es posible ver esto de otra manera: cada elemento $a \in \mathbb{X}$ tiene un subconjunto de $\mathbb{U}$ donde él es la respuesta apropiada (es decir, el conjunto de objetos más cercanos a a que a cualquier otro elemento de $\mathbb{X}$). Esto es el análogo exacto de una "región de Voronoi" para espacios Euclídeos en geometría computacional [Aur91][1]. La respuesta a una consulta q es el elemento $a \in \mathbb{X}$ al cual pertenece la región de Voronoi en la que cae q. Si a no es la respuesta, se necesita poder moverse a otro elemento más cercano a q. Es suficiente para ello conectar a con todos sus "vecinos de Voronoi" (es decir, los elementos del conjunto $\mathbb{X}$ cuyas áreas de Voronoi comparten una frontera con la de a), ya que si a no es la respuesta, entonces un vecino de Voronoi estará más cerca de q (esto es exactamente lo que establece la Propiedad 1).

Se considera el hiperplano entre a y b. aquel que divide el área de puntos x más cercanos a a o más cercanos a b. Cada elemento b que se agregue como un vecino de a hará que la búsqueda se mueva desde a hacia b, si q está del lado del hiperplano de b. Por lo tanto, si y sólo si se le agregan todos los vecinos de Voronoi a a, la única zona en donde la consulta no se moverá más allá de a será exactamente el área donde a es el vecino más cercano.

Por lo tanto, en un espacio vectorial, el grafo minimal que se busca corresponde a la triangulación clásica de Delaunay, el cual es un grafo donde los elementos que son vecinos de Voronoi están conectados. Así, la respuesta ideal en términos de complejidad de espacio es el grafo de Delaunay, pero generalizado para espacios arbitrarios, y el cual permitiría también búsquedas rápidas. La Figura 3.1 muestra un ejemplo en el espacio $\mathbb{R}^2$. Si se comienza la búsqueda desde p_{11}, se alcanza a p_9, el nodo más cercano a q, siempre mediante movimientos a vecinos más y más cercanos a q, los cuales se muestran como los arcos coloreados.

Desafortunadamente, no es posible computar el grafo de Delaunay de un espacio métrico general

[1] El nombre correcto en un espacio métrico general es "Dominio de Dirichlet" [Bri95].

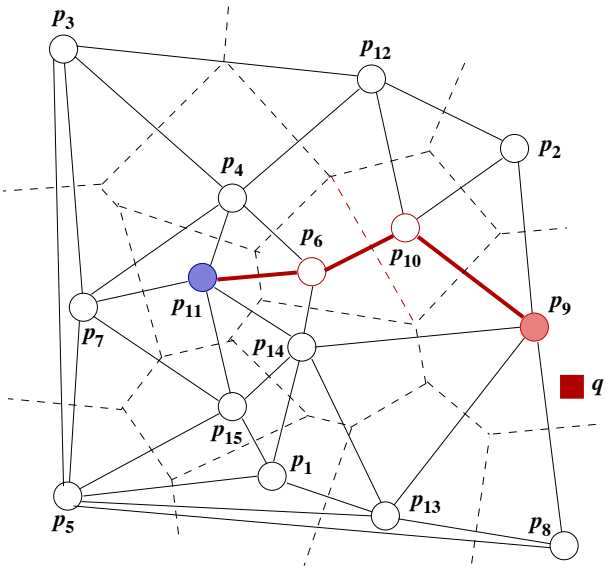


Figura 3.1: Un ejemplo del proceso de búsqueda con un grafo de Delaunay (arcos sólidos) correspondiente a una partición de Voronoi (áreas delimitadas por líneas punteadas).

dando sólo el conjunto de distancias entre los elementos de $\mathbb{X}$ sin ninguna indicación adicional de la estructura del espacio. Esto es porque, dado el conjunto de $|\mathbb{X}|^2$ distancias, diferentes espacios tendrán diferentes grafos. Más aún, no es posible probar que un único arco desde algún nodo a a un nodo b no está en el grafo de Delaunay, dando sólo las distancias. Por lo tanto, el único superconjunto del grafo de Delaunay que funciona para un espacio métrico arbitrario es el grafo completo, y como ya se explicó es inútil. Esto elimina la estructura de datos para aplicaciones generales. Se formaliza esta noción como un teorema.

Teorema 1 *Dadas las distancias entre cada par de elementos de un subconjunto finito $\mathbb{X}$ de un espacio métrico desconocido $\mathbb{U}$, entonces para cada $a, b \in \mathbb{X}$ existe una elección de $\mathbb{U}$ donde a y b están conectados en el grafo de Delaunay de $\mathbb{X}$.*

Demostración:

Dado el conjunto de distancias, se crea un nuevo elemento $x \in \mathbb{U}$ tal que $d(a, x) = M + \epsilon, d(b, x) = M$, y $d(y, x) = M + 2\epsilon$ para todos los otros y de $\mathbb{X}$. Esto satisface todas las desigualdades triangulares si $\epsilon \leq 1/2\ min_{y,z\in\mathbb{X}}\{d(y, z)\}$ y $M \geq 1/2\ max_{y,z\in\mathbb{X}}\{d(y, z)\}$. Por lo tanto, tal x puede existir en $\mathbb{U}$. Ahora, dada la consulta $q = x$ y estando posicionado en el elemento a, b es el elemento más cercano a x y la única manera de moverse hacia b sin irse más lejos de q es un arco directo desde a a b (ver Figura 3.2). Este argumento se puede repetir para cada par de elementos $a, b \in \mathbb{X}$.

□

Como la demostración del teorema supone que se pueden construir determinados elementos a distancia arbitraria entre dos dados, y como no todos los espacios métricos tienen esta propiedad, en realidad está afirmando que existen espacios donde no se puede construir un grafo de aproximación. Sin embargo, en ciertos espacios podría ser posible construirlo.

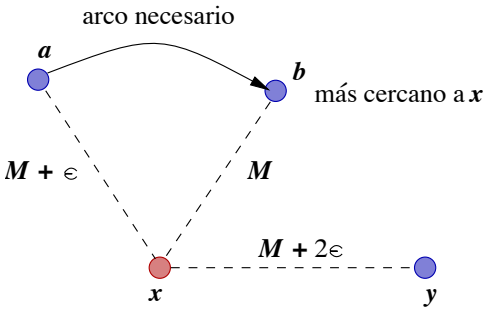


Figura 3.2: Ilustración del Teorema 1.

3.3. El Árbol de Aproximación Espacial

Para lograr una solución factible se hacen dos simplificaciones cruciales a la idea general. La simplificación resultante responde sólo a un conjunto reducido de consultas, es decir las consultas por el 1-vecino más cercano (*1-NN*) de $q \in \mathbb{X}$, que no es más que una búsqueda exacta. Sin embargo, luego en Sección 3.3.2 se mostrará cómo combinar la aproximación espacial con "backtracking" o retroceso para responder a cualquier consulta $q \in \mathbb{U}$ (no sólo $q \in \mathbb{X}$), para consultas por rango y consultas de k-vecinos más cercanos (*k-NN*).

(1) No se comienza atravesando el grafo desde un nodo aleatorio sino desde uno fijo, y por lo tanto no se necesitan todos los arcos de Voronoi.

(2) El grafo sólo podrá responder correctamente consultas por elementos $q \in \mathbb{X}$, es decir sólo elementos presentes en la base de datos.

Haciendo estas simplificaciones, se construye un grafo análogo al de Delaunay para buscar por aproximación espacial consultas *1-NN*. Realmente, el resultado no es un grafo sino un árbol, el cual se ha denominado *SAT* (*"Árbol de Aproximación Espacial"*). Luego se describe cómo usar este árbol para buscar cualquier elemento $q \in \mathbb{U}$ (no sólo para $q \in \mathbb{X}$) y cualquier tipo de consulta.

3.3.1. Proceso de Construcción

Para construir el árbol se elige aleatoriamente un elemento $a \in \mathbb{X}$ como la raíz. A continuación se selecciona un conjunto adecuado de vecinos $N(a)$ que satisfaga la siguiente propiedad:

Propiedad 2 *(dados a, $\mathbb{X}$)* $\forall x \in \mathbb{X}, x \in N(a) \Leftrightarrow \forall y \in N(a) - \{x\},\ d(x,y) > d(x,a)$.

Es decir, los vecinos de a forman un conjunto tal que cualquier vecino está más cerca de a que de cualquier otro vecino. La parte "sólo si" ($\Leftarrow$) de la definición garantiza que si es posible acercarse a algún $b \in \mathbb{X}$ entonces un elemento en $N(a)$ está más cerca de b que de a, porque se pusieron como vecinos directos a todos aquellos elementos que no están más cerca de cualquier otro vecino. La parte "si" ($\Rightarrow$) apunta a obtener además sólo los vecinos necesarios.

Notar que el conjunto $N(a)$ está definido no trivialmente en términos de sí mismo y que múltiples soluciones cumplen la definición. Por ejemplo, si a está alejado de b y c y éstos a su vez están cerca entre sí, entonces ambos $N(a) = \{b\}$ y $N(a) = \{c\}$ satisfacen la condición.

Encontrar el conjunto $N(a)$ más pequeño posible parece ser un problema de optimización combinatoria no trivial, ya que si se incluye un elemento es necesario sacar otros (esto ocurre entre b y c en el ejemplo del párrafo anterior). Sin embargo, heurísticas simples que agregan más vecinos que los necesarios trabajan bien. Se comienza con el nodo inicial a y su "bolsa" manteniendo todo el resto de $\mathbb{X}$. Como se espera que los nodos más cercanos sean más probablemente los vecinos, primero se ordena el conjunto de nodos por distancia creciente a a. Entonces, se van agregando nodos a $N(a)$ (el cual es inicialmente vacío). Cada vez que se considera un nuevo nodo b , se analiza si es más cercano a algún elemento de $N(a)$ que a a mismo. Si ese no es el caso, se agrega b a $N(a)$.

Al finalizar se obtiene un conjunto adecuado de vecinos. Notar que la Propiedad 2 se satisface gracias al hecho de haber considerado los elementos en orden creciente de distancia a a. La parte "$\Leftarrow$" de la condición claramente se cumple porque se inserta en $N(a)$ cualquier elemento que satisface la parte derecha de la cláusula. La parte "$\Rightarrow$" es más delicada. Sea $x \neq y \in N(a)$. Si y está más cerca de a que x entonces y se consideró primero. El algoritmo de construcción garantiza que si se inserta x en $N(a)$ entonces $d(x, a) < d(x, y)$. Si, por otro lado, x está más cerca de a que y, entonces $d(x, y) > d(y, a) \geq d(x, a)$ (es decir, un vecino no puede ser descartado por un nuevo vecino que se insertó después).

En este punto se debe decidir en cuál de las bolsas de los vecinos poner el resto de los nodos. Se puede poner cada nodo que no está en $\{a\} \cup N(a)$, como ya se estableció, en la bolsa de su elemento más cercano en $N(a)$ (*best-fit*). Se observa que esto requiere una segunda pasada cuando esté determinado completamente $N(a)$. Se ha mostrado en [Nav02] que también posible poner cada nodo en la bolsa del primer vecino más cercano que a (*first-fit*), y así no es necesaria una segunda pasada, pero la búsqueda se degrada. Por lo tanto, de aquí en más sólo se considera la estrategia *best-fit* para la selección del vecino en cuya bolsa se agregará al elemento.

Como ya se ha hecho con a, se procesan recursivamente todos sus vecinos, cada uno con los elementos de su bolsa. Notar que la estructura resultante no es un grafo sino un árbol, en el cual se puede buscar por cualquier $q \in \mathbb{X}$ por aproximación espacial para consultas del vecino más cercano. El mecanismo consiste en comparar q contra $\{a\} \cup N(a)$. Si a es el más cercano a q, entonces a es la respuesta, en otro caso se continúa la búsqueda por el subárbol del elemento más cercano a q en $N(a)$.

La razón por la que esto funciona es que, en tiempo de búsqueda, se puede replicar lo que pasó con q durante el proceso de construcción (v.g. entrar en el subárbol del vecino más cercano a q), hasta que se alcance a q. Esto es porque q ya está en el árbol, o sea, se realiza una búsqueda exacta.

Finalmente, es posible ahorrar algunas comparaciones durante la búsqueda almacenando en cada nodo a su radio de cobertura, es decir la máxima distancia $R(a)$ entre a y cualquier elemento en el subárbol con raíz a. La manera en que se usa esta información se hará evidente en la Sección 3.3.2.

El Algoritmo 1 muestra el proceso de construcción. Se invoca por primera vez como `Construir(`a`,` $\mathbb{X} - \{a\}$`)`, donde a es un elemento aleatorio del conjunto $\mathbb{X}$. Notar que, excepto para el primer nivel de recursión, se conocen todas las distancias $d(v, a)$ para cada $v \in \mathbb{X}$ y por lo tanto no es necesario recalcularlas. De manera similar, algunas de las $d(v, c)$ en la línea 10 ya se conocen desde la línea 6. La información que se almacena en la estructura de datos es la raíz a y los valores de $N()$ y $R()$ para todos los nodos. Por ser la estructura un árbol, el espacio necesario para su almacenamiento es $O(n)$.

3.3.2. Búsquedas

Desde luego es de poco interés buscar sólo elementos $q \in \mathbb{X}$. El árbol descripto puede, sin embargo, ser utilizado como un dispositivo para resolver consultas de cualquier tipo para cualquier $q \in \mathbb{U}$. Comencemos analizando las consultas por rango (q, r) con $q \in \mathbb{U}$ y radio r.

Algoritmo 1 Proceso para construir un *SAT*.

Construir (Nodo a, Conjunto de Elementos X)

1. $N(a) \leftarrow \emptyset$ /* vecinos de a */
2. $R(a) \leftarrow 0$ /* radio de cobertura */
3. For $v \in X$ en orden creciente de distancia a a Do
4. $R(a) \leftarrow \max(R(a), d(v,a))$
5. If $\forall b \in N(a),\ d(v,a) < d(v,b)$ Then $N(a) \leftarrow N(a) \cup \{v\}$
6. For $b \in N(a)$ Do $S(b) \leftarrow \emptyset$ /* subárboles */
7. For $v \in X - N(a)$ Do
8. $c \leftarrow \text{argmin}_{b \in N(a)} d(v,b)$
9. $S(c) \leftarrow S(c) \cup \{v\}$
10. For $b \in N(a)$ Do **Construir** (b,$S(b)$) /* construir subárboles */

La observación clave es que, aún si $q \notin \mathbb{X}$, las respuestas a la consulta son elementos $q' \in \mathbb{X}$ tales que $d(q,q') \leq r$. Así se usa el árbol para simular que se está buscando un elemento $q' \in \mathbb{X}$. No se conoce a q', pero como $d(q,q') \leq r$, se puede obtener a partir de q alguna información de distancia con respecto a q': por la desigualdad triangular se cumple que para cualquier $x \in \mathbb{U}$, $d(x,q) - r \leq d(x,q') \leq d(x,q) + r$.

Cuando se conoce el q que se busca, se avanza directamente al vecino de a más cercano a q. En este caso se busca un q' desconocido y no se sabe cuál es el vecino de a más cercano a q'. Por lo tanto, hay que explorar varios vecinos posibles. A algunos vecinos, afortunadamente, se los puede deducir como irrelevantes, en la medida que q' no los pudo haber elegido en la construcción si se cumple que $d(q,q') \leq r$.

En lugar de ir simplemente al vecino más cercano, se determina primero el vecino más cercano c de q entre $a \cup N(a)$. Así, para cada b en $\{a\} \cup N(a)$ se sabe que $d(c,q) \leq d(b,q)$. Sin embargo, como se explicó, es posible que $d(c,q') > d(b,q')$ y por lo tanto no se encontrará a q' sólo entrando en el subárbol de c. En su lugar, hay que entrar en *todos* los vecinos $b \in N(a)$ tal que $d(q,b) \leq d(q,c) + 2r$. Esto es porque el elemento virtual q' buscado puede diferir de q en a lo más r en cualquier evaluación de distancia, por ello se podría haber insertado dentro de tales nodos b. En otras palabras, un vecino b tal que $d(q,b) > d(q,c) + 2r$ satisface que $d(q',b) \geq d(q,b) - r > d(q,c) + r \geq d(q',c)$, así q' no se podría haber insertado en el subárbol de b. En cualquier otro caso, no se está seguro de ello y se debe entrar al subárbol de b.

Una manera diferente de ver este proceso es limitar inferiormente la distancia entre q y cualquier nodo x en el subárbol de b. Por la desigualdad triangular se cumple que $d(x,q) \geq d(x,c) - d(q,c)$ y $d(x,q) \geq d(q,b) - d(x,b)$. Resumiendo ambas desigualdades y manteniendo en mente que $d(x,b) \leq d(x,c)$ y $d(q,c) \leq d(q,b)$, se obtiene que $2d(x,q) \geq (d(q,b) - d(q,c)) + (d(x,c) - d(x,b)) \geq d(q,b) - d(q,c)$. Por lo tanto $d(x,q) \geq (d(q,b) - d(q,c))/2$. Si el último término es mayor que r, seguramente se puede descartar cada x en el subárbol de b. La condición es por ello $(d(q,b) - d(q,c))/2 > r$, o $d(q,b) > d(q,c) + 2r$.

El proceso garantiza que se comparará q contra cada nodo al que no se le puede probar que esté suficientemente lejos de q. De aquí, informando cada nodo q' que se ha comparado contra q y para el cual se cumple que $d(q,q') \leq r$, se puede asegurar que se informará cada elemento relevante.

Como se puede observar, lo que originalmente se concibió como una búsqueda por aproximación siguiendo un único camino se combina con *backtracking*, por ello se debe buscar en varios caminos. Éste es el precio de no construir un verdadero grafo de aproximación espacial. La Figura 3.3 ilustra

el proceso de búsqueda, comenzando desde p_{11} (la raíz). Sólo p_9 se encuentra en el resultado, pero se atraviesan todos los arcos coloreados.

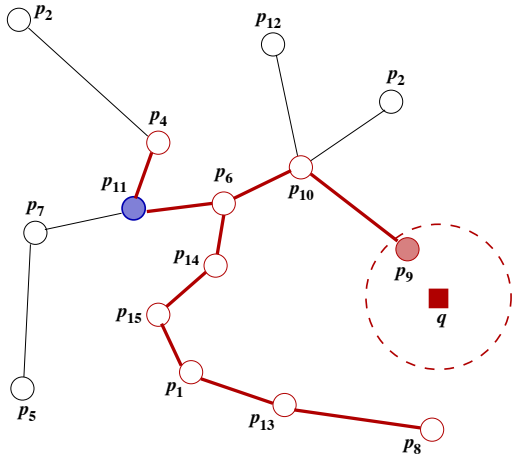


Figura 3.3: Un ejemplo del proceso de búsqueda.

El algoritmo de búsqueda se puede mejorar un poco más. Cuando se busca un elemento $q \in \mathbb{X}$ (es decir, una búsqueda exacta por un elemento del árbol), se sigue un único camino desde la raíz a q. En cada nodo a' en este camino, se elige el más cercano a q entre $\{a'\} \cup N(a')$. Por lo tanto, si la búsqueda se encuentra actualmente en el nodo a del árbol, sabemos que q está más cerca de a que de cualquier ancestro a' de a y también que de cualquier vecino de a'. De aquí, si se denota con $A(a)$ al conjunto de ancestros de a (incluyendo a a) y $N(A(a)) = \bigcup_{a' \in A(a)} N(a')$ se tiene que, durante la búsqueda, es posible evitar entrar en cualquier elemento $x \in N(a)$ tal que

$$d(q, x) > 2r + \text{mín}\{d(q, c),\ c \in N(A(a))\}$$

debido a que se puede mostrar usando la desigualdad triangular que ningún q' con $d(q, q') \leq r$ se pudo almacenar dentro de x. Esta condición es una versión más estricta de la condición original $d(q, x) > 2r + \text{mín}\{d(q, c),\ c \in \{a\} \cup N(a)\}$.

Esta observación se utiliza de la siguiente manera. En cualquier nodo b de la búsqueda se mantiene registro de la mínima distancia d_{min} a q vista hasta el momento a lo largo del camino, incluyendo vecinos. La búsqueda ingresa sólo en los vecinos que no estén más lejos que $d_{min} + 2r$ de q (criterio de hiperplano presentado en Sección 2.4.1).

Finalmente, el radio de cobertura $R(a)$ se usa para reducir más el costo de búsqueda. Nunca se debe entrar en un subárbol con raíz a en donde $d(q, a) > R(a) + r$, ya que esto implica que $d(q', a) > R(a)$ para cualquier q' tal que $d(q, q') \leq r$. La definición de $R(a)$ implica que q' no puede pertenecer al subárbol de a (criterio de radio cobertor presentado en Sección 2.4.1). El Algoritmo 2 muestra el proceso de búsqueda. Se invoca por primera vez como `BúsquedaRango(`a`,`q`,`r`,`$d(a,q)$`)` dónde a es la raíz del árbol. Notar que, en invocaciones recursivas, la distancia $d(a, q)$ ya está calculada.

3.3.3. Búsqueda de Vecinos más Cercanos

También es posible realizar búsquedas del vecino más cercano simulando una búsqueda por rango, donde se va reduciendo el radio de búsqueda a medida que se obtiene más información. Para resolver las consultas del vecino más cercano (*1-NN*), se comienza buscando con $r = \infty$, y se reduce r cada vez que se realiza una nueva comparación que da una distancia menor que r. Finalmente, se informa el elemento

Algoritmo 2 Rutina para buscar q con radio r en un *SAT*.

```
BúsquedaRango (Nodo a, Consulta q, Radio r, Distancia d_min)

1. If d(a,q) ≤ R(a) + r Then
2.     If d(a,q) ≤ r Then Informar a
3.     d_min ← mín {d_min} ∪ {d(q,c), c ∈ N(a)}
4.     For b ∈ N(a)
5.         If d(b,q) ≤ d_min + 2r Then BúsquedaRango (b,q,r,d_min)
```

más cercano visto durante toda la búsqueda. Para las consultas sobre los k vecinos más cercanos (k-*NN*) se almacena todo el tiempo una cola de prioridad con los k elementos más cercanos a q vistos hasta el momento. El radio r es la distancia entre q y el elemento más lejano en la cola (∞ cuando todavía se tienen menos que k candidatos). Cada vez que aparece un nuevo candidato se lo inserta en la cola, lo cual puede desplazar a otro elemento y por lo tanto reducir r. Al final, la cola contiene los k elementos más cercanos a q (recordar la Sección 2.6).

En una búsqueda por rango normal con radio fijo r, no es importante el orden en el que se realiza el backtrack en el árbol. Sin embargo, éste no es el caso aquí, porque se podrían encontrar rápidamente elementos cercanos a q para reducir pronto r. Una idea general propuesta en [Uhl91a] se adapta a esta estructura de datos. Se mantiene una cola de prioridad, donde los elementos más prometedores están primero. Inicialmente, se inserta la raíz del *SAT* en la estructura de datos. Iterativamente, se extrae la raíz del subárbol más prometedor, se la procesa e insertan todas las raíces de sus subárboles en la cola. Esto se repite hasta que la cola se vacíe o se pueda descartar su subárbol más prometedor (es decir, su "valor prometido" es suficientemente malo).

La medida más elegante de cuán prometedor es un subárbol es un límite inferior de la distancia entre q y cualquier elemento del subárbol. Cuando este límite inferior excede r es posible detener completamente el proceso. En realidad existen dos posibles límites inferiores:

1. Al encontrar el vecino más cercano c y se entra en cualquier otro vecino b tal que $d(q,b)-d(q,c) \leq 2r$, no hay que entrar en el subárbol con raíz b si no se cumple que $(d(q,b) - d(q,c))/2 \leq r$. De hecho, este c se toma sobre los vecinos de cualquier ancestro ($N(A(b))$).
2. Por el límite inferior de la distancia entre q y un elemento en el subárbol se cumple que $d(q,b) - R(b) \leq r$.

Como r se reduce durante la búsqueda, un nodo b puede parecer útil cuando se lo inserta en la cola de prioridad y ser inútil después, cuando se lo extrae de la cola para procesarlo. Así, se almacena junto con b el máximo de los dos límites inferiores, y se utiliza este máximo para ordenar los subárboles en la cola de prioridad, con el más pequeño primero. A medida que se extraen subárboles de la cola, se verifica si su valor excede a r, en cuyo caso el proceso completo se detiene y ya se sabe que todos los subárboles restantes contendrán elementos irrelevantes. Notar que es necesario mantener el rastro de d_{min} separadamente y que los nodos hijos heredan el límite inferior de sus padres.

El Algoritmo 3 presenta la búsqueda de vecinos más cercanos en un *SAT*, donde A es una cola de prioridad de pares $(nodo, distancia)$ ordenados por *distancia* creciente y Q es una cola de prioridad de ternas $(nodo, lbound, d_{min})$ ordenados por $lbound$ creciente [2]. Las rutinas $create$, $insert$, $size$,

[2] El Algoritmo original presentado en [Nav02] tiene un error en la línea 12, porque $(d(q,b) - d_{min})$ no aparece dividido por 2 como debería.

Algoritmo 3 Rutina para buscar los k vecinos más cercanos a q en un *SAT*.

BúsquedaNN (Árbol a, Consulta q, Vecinos Requeridos k)

```
1.  create(Q), create(A)
2.  insert(Q, (a, máx{0, d(q,a) − R(a)}, d(a,q)))     /* subárboles prometedores */
3.  r ← ∞
4.  While size(Q) > 0 Do
5.     (a, lbound, d_min) ← extractMin(Q)     /* elemento de Q con lbound más pequeño */
6.     If lbound > r Then Break     /* criterio global de parada */
7.     insert(A, (a, d(q,a)))
8.     If size(A) > k Then extractMax(A)     /* elemento de A con maxd más grande */
9.     If size(A) = k Then r ← máx(A)     /* valor de maxd más grande en A */
10.    d_min ← mín {d_min} ∪ {d(b,q), b ∈ N(a)}
11.    For b ∈ N(a) Do
12.       insert(Q, (b, máx{(d(q,b) − d_min)/2, d(q,b) − R(b), t}, d_min)
13. Informar A
```

$extractmin$ y $extractmax$ permiten trabajar sobre la cola de prioridad correspondiente.

3.4. Análisis

Se reproduce ahora un breve análisis de la estructura de *SAT*. Un análisis completo y mayores detalles respecto de lo que aquí se menciona se encuentra en [Nav02] (Apéndices A y B).

Éste es un análisis que se ha simplificado en varios aspectos, por ejemplo se supone que la distribución de distancia de los nodos que están en el árbol es la misma que en el espacio global. Tampoco se tiene en cuenta que se ordenan las bolsas antes de seleccionar los vecinos (los resultados son pesimistas en este sentido, dado que parece como si tuvieran más vecinos). Sin embargo, como se puede ver en los resultados experimentales [Nav02], la adecuación con la realidad es muy buena. Este análisis se hace para una función de distancia continua, aunque adaptarlo al caso discreto es inmediato.

Los resultados se pueden resumir como sigue. El *SAT* necesita espacio lineal $O(n)$, un tiempo de construcción $O(n \log^2 n / \log\log n)$ y tiempo de búsqueda sublineal $O(n^{1-\Theta(1/\log\log n)})$ en espacios difíciles (alta dimensionalidad intrínseca) y $O(n^\alpha)$ $(0 < \alpha < 1)$ en espacios sencillos (baja dimensionalidad intrínseca).

Se demuestra (en el Apéndice A de [Nav02]) que, bajo las simplificaciones ya mencionadas, el número promedio de vecinos es:

$$N(n) \;=\; \Theta(\log n)$$

La constante está entre 1,00 y 1,35. Existe también una parte constante que sería especialmente relevante en espacios sencillos. Sin embargo, para dicho análisis basta con $\Theta(\log n)$.

Esto permite determinar algunos parámetros del *SAT*. Por ejemplo, como en promedio $\Theta(n/\log n)$ elementos van en cada subárbol, la profundidad promedio de una hoja en el árbol es:

$$H(n) \;=\; 1 + H\left(\frac{n}{\log n}\right) \;=\; \Theta\left(\frac{\log n}{\log\log n}\right)$$

lo que se obtiene en detalle en el Apéndice B de [Nav02].

El costo de construcción, en términos de evaluaciones de distancia, es como sigue. La bolsa con los n elementos es comparada contra el nodo raíz. Se seleccionan $\Theta(\log n)$ como vecinos y luego todos los demás elementos se comparan contra los vecinos y se incorporan en una bolsa. Luego, todos los vecinos se construyen recursivamente. Por lo tanto, llamando $B(n)$ al costo de construir un *SAT* de n elementos, se obtiene la recurrencia

$$B(n) \;=\; n\log n + \log(n) B\left(\frac{n}{\log n}\right)$$

la que se resuelve como

$$B(n) \;=\; \Theta\left(\frac{n\log^2 n}{\log\log n}\right)$$

Como ya se mencionó previamente, el espacio necesario por el índice es $O(n)$, porque por ser un árbol se requiere ese número de vínculos.

Como existen $\Theta(\log n)$ vecinos, y se entra en cada uno con la misma probabilidad, el tamaño del conjunto dentro de un vecino es $\Theta(n/\log n)$. De aquí, si se denota con $T(n)$ al costo de buscar con n elementos, se tiene que (ver Apéndice B de [Nav02]):

$$T(n) \;=\; \Theta\left(n^{1-\Theta(1/\log\log n)}\right)$$

Esto muestra sublinealidad con respecto a n. Por otro lado, a medida que se incrementa el radio de búsqueda o se incrementa la dimensión intrínseca del espacio, el costo se aproxima al lineal.

Por otra parte, se hace notar que cuando el espacio es más sencillo (v.g. espacios de vectores con dimensión menor que $O(\log n)$), el número de vecinos $N(n)$ es más cercano a una constante porque no pueden existir tantos vecinos. En ese caso el análisis produce $T(n) = O(n^\alpha)$ para una constante $0 < \alpha < 1$. Aunque se prefiere, sin embargo, ser fiel a la complejidad más conservadora.

3.5. Árbol de Aproximación Espacial Dinámico

La construcción del *SAT* necesita conocer de antemano todos los elementos de la base de datos $\mathbb{X}$. En particular, una vez que se ha construido el árbol es dificultoso agregarle nuevos elementos bajo la estrategia de construcción elegida (*best-fit*). Si los elementos de la base de datos no se conocen de antemano, y luego de haber seleccionado la raíz del árbol, no es posible examinar todos los objetos para decidir cuáles de ellos satisfacen la Propiedad 2 y por lo tanto los vecinos deberían poder seleccionarse sobre una base de *primero en llegar primero en atenderse*.

Sin embargo, en [NR02, NR08] se han propuesto y evaluado empíricamente distintas variantes que permiten hacer dinámico al *SAT*. No sólo se ha conseguido crearlo por inserciones sucesivas, sino también poder eliminarle elementos. Esta versión totalmente dinámica del *SAT* se denomina *Árbol de Aproximación Espacial Dinámico* (*DSAT*). Sorprendentemente, el *DSAT* además de conseguir el dinamismo ha superado el desempeño del *SAT* en las búsquedas. En el siguiente capítulo (Capítulo 4) se analizan los motivos por los que las búsquedas sobre el *DSAT* son mejores que en el *SAT* y aprovechando dicho análisis se proponen otras versiones para el *SAT*.

Como se ha mencionado previamente, muy pocos índices para búsquedas por similitud soportan eliminaciones y de aquéllos que las soportan muchos sufren posteriormente que el desempeño de las búsquedas se degrada.

En esta sección se describirán las mejores variantes existentes para inserción y eliminación de elementos en el *DSAT*. De las variantes discutidas y evaluadas en [NR08] algunas son alternativas ingenuas

y/o que surgen del folklore de las estructuras de datos (por ejemplo, reconstruir los subárboles y usar áreas de rebalse), otras son más sofisticadas (por ejemplo, estampa de tiempo, inserción en las hojas y aridad acotada) y otras son combinación de variantes. Algunas de estas variantes, como la que se describe en la Sección 3.5.1 para las inserciones y en la Sección 3.5.5 para las eliminaciones, son aportes iniciales de esta tesis para la optimización del *DSAT*. Aunque existen alternativas clásicas para transformar una estructura estática en dinámica [BS80, GR93], las mismas no se han considerado porque a pesar de que logren no degradar significativamente los costos de inserción y búsqueda, las variantes del *DSAT* analizadas realmente mejoran los costos de construcción y búsqueda. Las demás variantes de inserción y eliminación de elementos no descriptas aquí aparecen en detalle en [NR08].

3.5.1. Inserciones y Búsquedas

La mejor estrategia para insertar elementos en el *DSAT* es el resultado de la combinación entre dos estrategias llamadas: *estampa de tiempo* (*timestamp*) y *aridad acotada*. Se describe a continuación cada una de ellas junto con los algoritmos de búsqueda que deben aplicarse, para luego detallar la combinación resultante.

Estampa de Tiempo

La alternativa de *timestamp* se basa en mantener una estampa del tiempo de inserción de cada elemento. Cuando se inserta un nuevo elemento, se lo agrega como un vecino en el punto de inserción apropiado usando la estrategia best-fit, pero no se reconstruye el árbol. Se considera que los vecinos se agregan al final, así leyéndolos de izquierda a derecha se tienen tiempos crecientes de inserción. Esto también hace que los padres sean siempre más antiguos que sus hijos.

Como se mencionó, un nuevo elemento se debe agregar como vecino en el punto de inserción adecuado. Por lo tanto, en cada nodo a desde la raíz del árbol se procede de la siguiente manera.

(a) Si $d(a,x) < d(b,x), \forall b \in N(a)$ se inserta a x como nuevo vecino de a; es decir, el nuevo conjunto de vecinos de a será $N(a) \cup \{x\}$.

(b) En otro caso, se determina el nodo $c \in N(A)$ tal que $d(c,x) \leq d(b,x), \forall b \in N(a) - \{c\}$; es decir, el vecino en $N(a)$ más cercano a x, y se continúa el proceso desde c hasta encontrarse en una situación como la descripta en **(a)** o hasta alcanzar una hoja.

Dicho de otro modo, al insertar x en cada nodo a se debe verificar si existe un $b \in N(a)$ tal que $d(b,x) \leq d(a,x)$. Si dicho b existe, se selecciona el c que minimiza la distancia a x; es decir $c \in N(A)$ tal que $d(c,x) \leq d(b,x), \forall b \in N(a) - \{c\}$ y se continúa el proceso de insertar x desde c. Por el contrario, si dicho b no existe, entonces x se debe convertir en un vecino del nodo corriente a.

El Algoritmo 4 muestra el proceso de inserción de un elemento x en un árbol cuya raíz es a. El *DSAT* puede construirse por inserciones sucesivas sobre un árbol inicial formado por un único nodo a donde $N(a) = \emptyset, time(a) = CurrentTime = 1$ y $R(a) = 0$. Así, el primer elemento en ser insertado se convertirá en la raíz del árbol.

Durante las búsquedas, se consideran los vecinos $\{b_1, \ldots, b_k\}$ de a desde el más antiguo hasta el más nuevo. En este caso la minimización, para aplicar el criterio de hiperplano, se lleva a cabo a medida que se atraviesan los vecinos. Es decir, se entra al subárbol de b_1 si $d(q,b_1) \leq d(q,a) + 2r$; al subárbol de b_2 si $d(q,b_2) \leq \min(d(q,a), d(q,b_1)) + 2r$; y en general al subárbol de b_i si $d(q,b_i) \leq$

Algoritmo 4 Inserción de un nuevo elemento x en un *DSAT* cuya raíz es a, para la técnica de timestamp.

InsertarTS (Nodo a, Elemento x)

1. $R(a) \leftarrow \max\{R(a), d(a,x)\}$
2. $c \leftarrow \text{argmin}_{b \in N(a)} d(b,x)$
3. If $d(a,x) < d(c,x)$ Then
4. $N(a) \leftarrow N(a) \cup \{x\}$
5. $N(x) \leftarrow \emptyset$, $R(x) \leftarrow 0$
6. $time(x) \leftarrow CurrentTime$
7. $CurrentTime \leftarrow CurrentTime + 1$
8. Else **InsertarTS** (c,x)

Algoritmo 5 Búsqueda de q con radio r en un *DSAT* de raíz a, usando la técnica de timestamp.

BúsquedaRangoTS (Nodo a, Consulta q, Radio r, Distancia d_{min}, Timestamp t)

1. If $time(a) < t \ \wedge \ d(a,q) \leq R(a) + r$ Then
2. If $d(a,q) \leq r$ Then Informar a
3. For $b_i \in N(a)$ Do /* en orden creciente de timestamp */
4. If $d(b_i,q) \leq d_{min} + 2r$ Then
5. $t' \leftarrow \min\{t\} \cup \{time(b_j), j > i \wedge d(b_i,q) > d(b_j,q) + 2r\}$
6. **BúsquedaRangoTS** (b_i,q,r,d_{min},t')
7. $d_{min} \leftarrow \min\{d_{min}, d(b_i,q)\}$

$\min(d(q,a), d(q,b_1), \ldots, d(q,b_{i-1})) + 2r$. Esto se debe a que entre la inserción de b_i y la de b_{i+j} pueden haber aparecido nuevos elementos que prefirieron a b_i sólo porque b_{i+j} todavía no se había convertido en vecino. Por lo tanto, se podría perder un elemento relevante a la búsqueda si no se entra al subárbol de b_i por la existencia de b_{i+j}.

Hasta el momento no se ha considerado realmente la necesidad de los timestamps, salvo para mantener los vecinos ordenados por ellos. Otro esquema más sofisticado es usar en las búsquedas los timestamps para reducir el trabajo realizado dentro de los vecinos más antiguos. Si suponemos que $d(q,b_i) > d(q,b_{i+j}) + 2r$, se debe entrar en b_i porque es más antiguo que b_{i+j}. Sin embargo, sólo los elementos cuyo timestamp sea menor que el de b_{i+j} deberían considerarse durante la búsqueda dentro del subárbol de b_i. Los elementos más nuevos que b_{i+j} lo han visto como vecino y ellos no pueden ser interesantes para la búsqueda si ellos eligieron a b_i. Como los nodos padres son más viejos que sus descendientes, tan pronto como se encuentre un nodo dentro del subárbol de b_i con timestamp mayor que el de b_{i+j} se puede parar la búsqueda en esa rama, porque su subárbol será aún más joven.

El Algoritmo 5 muestra la búsqueda por rango. El cálculo de d_{min} se realiza en la línea 7, a medida que se atraviesan los vecinos en orden ascendente de timestamp. El algoritmo inicialmente se invoca como BúsquedaRangoTS(a,q,r,$d(a,q)$,$CurrentTime$), donde a es la raíz del árbol. Notar que en las invocaciones recursivas la distancia $d(a,q)$ ya se ha calculado. A pesar de la naturaleza cuadrática de la iteración implícita en las líneas 3 y 5, la consulta es desde luego comparada sólo una vez con cada vecino.

Se considera ahora cómo afecta la técnica de timestamp a la búsqueda de vecinos más cercanos. Se debe expresar el manejo operativo de los timestamps como límites inferiores de distancias. En lugar de pensar en términos del máximo timestamp permitido de interés dentro de un nodo y, se piensa en

Algoritmo 6 Búsqueda de los k vecinos más cercanos de q en un *DSAT* con raíz a, usando timestamps.

BúsquedaNNTS (Árbol a, Consulta q, Vecinos Requeridos k)

```
1.  create(Q), create(A)
2.  insert(Q, (a, máx{0, d(q,a) − R(a)}, d(q,a)))
3.  r ← ∞
4.  While size(Q) > 0 Do
5.     (a, lbound, d_min) ← extractMin(Q)      /* elemento de Q con lbound más pequeño */
6.     If lbound > r Then Break      /* criterio global de parada */
7.     Insertar(A, (a, d(q,a)))
8.     If size(A) > k Then extractMax(A)      /* elemento de A con maxd más grande */
9.     If size(A) = k Then r ← máx(A)      /* valor de maxd más grande en A */
10.    For b_i ∈ N(a) Do      /* en orden creciente de timestamp */
11.       maxr ← máx {(d(q,b_i) − d(q,b_j))/2, j > i}
12.       insert(Q, (b_i, máx{maxr, (d(q,b_i) − d_min)/2, d(q,b_i) − R(b_i), t}, d_min)
13.       d_min ← mín{d_min, d(q,b_i)}
14. Informar la respuesta A
```

términos del máximo radio de búsqueda que permite entrar en y. Cada vez que se entra en un subárbol y de b_i, se buscan los hermanos b_{i+j} de b_i que sean más viejos que y. Sobre este conjunto, se calcula el máximo radio r_y que permite no entrar a y; es decir, $r_y = \max(d(q,b_i) - d(q,b_{i+j}))/2$. Si se cumple que $r < r_y$, entonces no se debe entrar al subárbol de y.

Suponiendo que se está procesando actualmente el nodo b_i y que se inserta su hijo y, se calcula r_y y se lo inserta junto con y en la cola de prioridad. Luego, cuando se procese y, se considera el radio corriente de búsqueda r y se descarta a y si $r < r_y$. Si se inserta un hijo z de y, entonces se almacena con el valor $\max(r_y, r_z)$. El Algoritmo 6 ilustra el proceso. La estructura A es una cola de prioridad de pares $(nodo, distancia)$ ordenados por *distancia* decreciente. Q es otra cola de prioridad de triuplas $(nodo, lbound, d_{min})$ ordenado por $lbound$ creciente.

Aridad Acotada

Si se remueve la parte $\Leftarrow$ de la Propiedad 2 (Sección 3.3.1; es decir, pueden existir algunos elementos más cercanos a a que a cualquier otro elemento $b \in N(a)$, y esos elementos no están aún en $N(a)$. Esta parte de la Propiedad 2 garantiza que, si q está más cerca de a que de cualquier otro vecino $b \in N(a)$, entonces se puede parar la búsqueda en ese punto debido a que q debería estar en $N(a)$ y no dentro de algún subárbol. Si se debilita la Propiedad 2 como se ha explicado, entonces no existe tal garantía. Aún si x está más cerca de a que de cualquier otro vecino $b \in N(a)$, x podría estar en el subárbol de su vecino más cercano en $N(a)$.

Por lo tanto, durante las búsquedas, en lugar de encontrar el c más cercano entre $\{a\} \cup N(a)$ y entrar en cualquier $b \in N(a)$ tal que $d(q,b) \leq d(q,c) + 2r$, se excluye la raíz $\{a\}$ del subárbol de la minimización. El beneficio de esto es que, durante una inserción, no se está más obligado a poner a ese nuevo elemento como un vecino de a cuando la Propiedad 2 lo pudiera requerir. Es decir, en la inserciones, aún si x está más cerca de a que de cualquiera de sus vecinos en $N(a)$, existe la posibilidad de no poner a x como un vecino de a, sino de insertarlo en su vecino más cercano en $N(a)$. Durante las búsquedas se alcanzará a x debido a que los procesos de inserción y búsqueda son similares.

Esta relajación de la Propiedad 2 puede utilizarse de diversas maneras, como se ha mostrado en

Algoritmo 7 Proceso de inserción de un nuevo elemento x en un *DSAT* cuya raíz es a, usando aridad acotada.

```
InsertarBA (Nodo a, Elemento x)

1. c ← argmin_{b∈N(a)} d(b,x)
2. If d(a,x) < d(c,x) ∧ |N(a)| < MaxArity Then
3.    Reunir in S los elementos del subárbol con raíz a
4.    ConstruirArbol(a, S ∪ {x})   /* rebuild the subtree */
5. Else
6.    R(a) ← máx{R(a), d(a,x)}
7.    InsertarBA (c,x)
```

Algoritmo 8 Búsqueda de q con radio r en un *DSAT* con raíz a, con aridad limitada.

```
BúsquedaRangoBA (Nodo a, Consulta q, Radio r)

1. If d(a,q) ≤ R(a) + r Then
2.    If d(a,q) ≤ r Then Informar a
3.    d_min ← mín {d(q,b_i), b_i ∈ N(a)}
4.    For b_i ∈ N(a) Do
5.       If d(b_i,q) ≤ d_min + 2r Then BúsquedaRangoBA (b_i,q,r)
```

[NR08]. Esto es particularmente interesante, por cómo se pueden reducir significativamente los costos de construcción mientras se retiene un desempeño competitivo en las búsquedas. Analizando cómo afecta esta relajación a los árboles, se descubrió que, en los casos donde la construcción dinámica mejora sobre la estática, la aridad promedio (número de hijos) de los nodos del árbol se reducía mucho. Esto parecía indicar que la razón por la cual el *SAT* tiene un pobre desempeño en algunos espacios es que su aridad era demasiado alta. Aún cuando el *SAT* adapta automáticamente su aridad al espacio, este mecanismo no es óptimo.

Así, es posible controlar directamente la aridad del árbol fijando una máxima aridad admisible, $MaxArity$. Cuando se inserte un nuevo elemento x y quiera convertirse en un vecino de un nodo a, se permite la inserción en ese punto sólo si $|N(a)| < MaxArity$, en otro caso se fuerza al elemento a elegir su vecino b más cercano en $N(a)$ y continuar la inserción desde ese nodo. Algoritmo 7 da el proceso de inserción para esta alternativa.

El Algoritmo 8 ilustra el proceso de una búsqueda por rango, considerando que el *DSAT* se ha construido con la técnica de aridad acotada. Se invoca como `BúsquedaRangoBA(`a`,`q`,`r`)` donde a es la raíz del árbol y en las invocaciones recursivas $d(a,q)$ ya se ha calculado. El algoritmo es muy similar al de la versión estática (Algoritmo 2), pero existe una diferencia importante como consecuencia de haber acotado la aridad. El valor de d_{min} no se hereda, lo cual significa que la raíz a no se incluye en la minimización, como se ha explicado. Esto también significa que los vecinos de los ancestros de a, $N(A(a))$, también se excluyen de la minimización. La razón es que, debido a haber relajado la Propiedad 2, no se puede garantizar que los elementos $b \in N(a)$ estén más cerca de a que del padre de a, o que de cualquier ancestro de a.

El algoritmo para realizar la búsqueda de vecinos más cercanos en un *DSAT*, construido con aridad acotada, puede obtenerse a partir del Algoritmo 3. No es necesario en este caso almacenar d_{min} junto con

Algoritmo 9 Rutina para buscar los k vecinos más cercanos a q en un *DSAT* con aridad acotada.

BúsquedaNNBA (Árbol a, Consulta q, Vecinos Requeridos k)

```
 1. create(Q), create(A)
 2. insert(Q, (a, máx{0, d(q,a) − R(a)}))     /* subárboles prometedores */
 3. r ← ∞
 4. While size(Q) > 0 Do
 5.    (a, lbound) ← extractMin(Q)     /* elemento de Q con lbound más pequeño */
 6.    If lbound > r Then Break     /* criterio global de parada */
 7.    insert(A, (a, d(q,a)))
 8.    If size(A) > k Then extractMax(A)     /* elemento de A con maxd más grande */
 9.    If size(A) = k Then r ← máx(A)     /* valor de maxd más grande en A */
10.    d_min ← mín {d(b,q), b ∈ N(a)}
11.    For b ∈ N(a) Do
12.       insert(Q, (b, máx{(d(q,b) − d_min)/2, d(q,b) − R(b), t}))
13. Informar A
```

los subárboles mantenidos en Q, y en la línea 10, el elemento $\{d_{min}\}$ se puede excluir de la minimización en el lado derecho de la asignación. El proceso adaptado se muestra en el Algoritmo 9

Timestamp + Aridad Acotada

La alternativa anterior para el *DSAT* logró mejorar costos de construcción respecto de la versión estática del *SAT*. Aunque la idea inicial surgió para limitar el tamaño del árbol a reconstruir, el efecto colateral fue que el árbol dinámico resultó más adecuado en algunos espacios. Por lo tanto, esta restricción sobre el punto de inserción se puede ver como una sintonización de parámetro por sí misma, no relacionada con el objetivo de limitar el tamaño del árbol a reconstruir. Más aún, el costo de reconstrucción mismo puede ser completamente evitado si se combina con la técnica de timestamp. De este modo, se podrían tener árboles que eviten cualquier reconstrucción, y al mismo tiempo limiten los posibles puntos de inserción con el objetivo de obtener un árbol de mejor forma.

La variante que combina timestamp con aridad acotada funciona como sigue. Se fija la aridad máxima del árbol y también se mantiene un timestamp del tiempo de inserción de cada elemento. El proceso de búsqueda para el punto de inserción es exactamente el utilizado en el Algoritmo 7, excepto que en las líneas 3 y 4 no se debe reconstruir el subárbol, sino que basta con agregar a x como el último vecino en la lista. El Algoritmo 10 muestra el proceso completo adaptado.

En las búsquedas se deben combinar las consideraciones realizadas para timestamp con las de aridad acotada. El Algoritmo 11 ilustra la búsqueda por rango para esta alternativa combinada. Notar que $d(a, q)$ siempre es ya conocida, excepto en la primera invocación.

También el algoritmo de búsqueda de k vecinos más cercanos se obtiene como una combinación de los respectivos algoritmos para timestamp y aridad acotada. El Algoritmo 12 muestra dicha combinación.

3.5.2. Eliminaciones

Para eliminar un elemento x, el primer paso es encontrarlo en el árbol. A diferencia de las estructuras de datos clásicas para problemas de búsqueda tradicionales, hacer esto no es equivalente a simular la

Algoritmo 10 Inserción de un nuevo elemento x en un *DSAT* con raíz a, usando timestamp + aridad acotada.

InsertarTBA (Nodo a, Elemento x)

1. $R(a) \leftarrow \max(R(a), d(a,x))$
2. $c \leftarrow \text{argmin}_{b \in N(a)} d(b,x)$
3. If $d(a,x) < d(c,x) \wedge |N(a)| < MaxArity$ Then
4. $\quad N(a) \leftarrow N(a) \cup \{x\}$
5. $\quad N(x) \leftarrow \emptyset$, $R(x) \leftarrow 0$
6. $\quad time(x) \leftarrow CurrentTime$
7. $\quad CurrentTime \leftarrow CurrentTime + 1$
8. Else **InsertarTBA**(c,x)

Algoritmo 11 Búsqueda de q con radio r en un *DSAT* con raíz a, usando timestamp + aridad acotada.

BúsquedaRangoTBA (Nodo a, Consulta q, Radio r, Timestamp t)

1. If $time(a) < t \ \wedge \ d(a,q) \leq R(a) + r$ Then
2. $\quad$ If $d(a,q) \leq r$ Then Informar a
3. $\quad d_{min} \leftarrow \infty$
4. $\quad$ For $b_i \in N(a)$ Do /* en orden ascendente de timestamp */
5. $\quad\quad$ If $d(b_i,q) \leq d_{min} + 2r$ Then
6. $\quad\quad\quad t' \leftarrow \min\{t\} \cup \{time(b_j), j > i \wedge d(b_i,q) > d(b_j,q) + 2r\}$
7. $\quad\quad\quad$ **BúsquedaRangoTBA** (b_i,q,r,t')
8. $\quad\quad\quad d_{min} \leftarrow \min\{d_{min}, d(b_i,q)\}$

Algoritmo 12 Búsqueda de los k vecinos más cercanos de q en un *DSAT* con raíz a, usando timestamp + aridad acotada.

BúsquedaNNTBA (Árbol a, Consulta q, Vecinos Requeridos k)

1. $create(Q)$, $create(A)$
2. $insert(Q, (a, \max\{0, d(q,a) - R(a)\}))$
3. $r \leftarrow \infty$
4. While $size(Q) > 0$ Do
5. $\quad (a, lbound) \leftarrow extractMin(Q)$ /* elemento de Q con $lbound$ más pequeño */
6. $\quad$ If $lbound > r$ Then Break /* criterio global de parada */
7. $\quad Insertar(A, (a, d(q,a)))$
8. $\quad$ If $size(A) > k$ Then $extractMax(A)$ /* elemento de A con $maxd$ más grande */
9. $\quad$ If $size(A) = k$ Then $r \leftarrow \max(A)$ /* valor de $maxd$ más grande en A */
10. $\quad d_{min} \leftarrow \infty$
11. $\quad$ For $b_i \in N(a)$ Do /* en orden creciente de timestamp */
12. $\quad\quad maxr \leftarrow \max\{(d(q,b_i) - d(q,b_j))/2, j > i\}$
13. $\quad\quad insert(Q, (b_i, \max\{maxr, (d(q,b_i) - d_{min})/2, d(q,b_i) - R(b_i), t\})$
14. $\quad\quad d_{min} \leftarrow \min\{d_{min}, d(q,b_i)\}$
15. Informar la respuesta A

inserción de x viendo en el árbol a dónde guía. La razón es que el árbol era diferente cuando se insertó x. Si x fuera insertado nuevamente, podría elegir entrar en un paso diferente en el árbol, el cual no existía al momento de la primera inserción.

Una solución elegante a este problema es realizar una búsqueda por rango con radio cero; es decir, una consulta $(x, 0)$. Esto es razonablemente barato y guiará a todos los lugares en el árbol donde x podría insertarse.

En otro sentido, si la búsqueda es necesaria o no es dependiente de la aplicación. La aplicación podría retornar un "manejador" cuando se inserta un objeto en el conjunto. Este manejador puede contener un puntero al correspondiente nodo del árbol. Agregar punteros al padre en el árbol podría permitir ubicar el paso sin costo (en términos de cálculos de distancias). Por lo tanto, no se considera la ubicación del objeto como parte del problema de eliminación, aunque se ha mostrado cómo se puede proceder si es necesario.

Se han estudiado en [NR02, NR08] varias alternativas para eliminar elementos en un *DSAT*. Se descartó desde el comienzo la alternativa trivial que marca el elemento como eliminado sin eliminarlo efectivamente, dado que se considera que es inaceptable en la mayoría de las aplicaciones. Se considera que el elemento debe ser físicamente eliminado. Sin embargo, si se desea, se puede mantener su nodo en el árbol, pero no el objeto en sí mismo.

Claramente, una hoja del árbol siempre se puede eliminar sin ningún costo ni complicación. Por lo tanto, el foco es cómo eliminar nodos internos del árbol. Sin embargo, cabe destacar que la mayoría de los nodos son hojas, especialmente si la aridad del árbol es alta. Así, cuando se consideren las eliminaciones se tendrá una motivación para usar aridades altas.

Aunque en [NR08] se han presentado diversas alternativas de eliminación, se describen aquí aquéllas que han resultado empíricamente más eficientes y que además poseen algunas características que las hacen más interesantes para este trabajo. Una de las alternativas desconecta el subárbol del nodo x eliminado, reconstruyéndolo exactamente como si x nunca se hubiera insertado, lo que garantiza la calidad del árbol luego de sucesivas eliminaciones. Como los costos de eliminación se vuelven significativos, es posible amortizar el costo de la reconstrucción sobre muchas eliminaciones, mientras que se mantiene la calidad deseada del árbol. La otra alternativa que se describe aquí reemplaza el elemento eliminado con otro que ocupa su lugar en el nodo, para no realizar reconstrucción. Sin embargo, se debe reconstruir periódicamente a fin de evitar la degradación del desempeño en las búsquedas.

3.5.3. Reconstrucción de Subárboles

En [NR08] se conjetura que la razón por la cual las búsquedas se degradan debido a las eliminaciones es geométrica. En un *DSAT* cada subárbol manipula los puntos más cercanos a él que a otros subárboles. Esto es una clase de particionado de Voronoi del espacio, donde cada raíz de subárbol actúa como el centro del área. Cuando se elimina uno de tales subárboles, sus elementos deben ser insertados en otro lugar, dado que el particionado en niveles superiores del árbol puede haber cambiado. En este caso, el área vacía es cubierta por otros vecinos del elemento eliminado x, el cual sin embargo no tiene elementos dentro. Esto reduce la precisión de la búsqueda debido a que aquellos vecinos que cubren las áreas vacías reciben muchas búsquedas inútiles para aquellas áreas. Alternativamente, es posible imaginar que los elementos en el subárbol de x caerán de nuevo en el mismo subárbol. Ellos se agregarían al final de la lista de vecinos, perdiendo su ubicación original y produciendo una asimetría hacia la derecha (es decir, hacia los vecinos más jóvenes). Esto significa que cualquier búsqueda dará prioridad a los primeros vecinos (los cuales aún cubren las áreas vacías) y entonces, adicionalmente, entrará en los vecinos más nuevos que realmente contienen los elementos en esa área. En un sentido, x estaba actuando como un

Algoritmo 13 Proceso para eliminar x desde un *DSAT*, por reconstrucción de subárboles.

```
Eliminar-R (Nodo x)

1. b ← padre(x)
2. Reunir en S los elementos más jóvenes que x del subárbol de raíz b
3. Ordenar S por timestamp creciente
4. N(b) ← N(b) − {x}
5. For y ∈ S Do InsertarTBA (b,y)    /* sin cambiar su timestamp */
```

tapón que prevenía que las búsquedas entraran innecesariamente en sus vecinos más jóvenes. Luego de eliminar dicho tapón, las búsquedas se vuelven más costosas.

Independientemente de si la conjetura anterior se mantenga, es claro que se debería encontrar un método que no degrade las búsquedas. La mejor manera de asegurar esto es logrando que el árbol resultante al eliminar x sea exactamente como si x nunca se hubiera insertado.

Cuando se elimina un nodo $x \in N(b)$, se desconecta a x desde el árbol principal. Por consiguiente, todos sus descendientes deben ser reinsertados. Más aún, los elementos en el subárbol de b que son más jóvenes que x se han comparado con x para determinar su punto de inserción. Por lo tanto, estos elementos, en ausencia de x, podrían elegir otro paso si se reinsertaran en el árbol. Entonces, se recuperan todos los elementos más jóvenes que x que descienden desde b (es decir, aquéllos cuyos timestamps sean mayores, los cuales naturalmente incluyen todos los descendientes de x) y se los reinserta en el árbol, dejando al árbol como si x nunca se hubiera insertado.

Si se reinsertan los elementos más jóvenes que x como si fueran elementos nuevos, esto es, si se les asignan nuevos timestamps, entonces se deben buscar los puntos de reinserción apropiados comenzando desde la raíz del árbol. Por otra parte, si se les mantienen sus timestamps originales, entonces se puede comenzar su reinserción desde b y así ahorrar muchas comparaciones. La razón es que se los está reinsertando como si el tiempo actual fuera el de su inserción original, cuando todas las elecciones más nuevas que aparecieron después no existían y así aquellos elementos deberían elegir lo mismo que eligieron en aquel momento, llegando de nuevo hasta b. A fin de dejar el subárbol resultante como si x nunca se hubiera insertado, se deben reinsertar los elementos en el orden original; es decir, en orden creciente de sus timestamps.

Por consiguiente, cuando se elimina el nodo $x \in N(b)$, se recuperan todos los elementos más jóvenes que x desde el subárbol con raíz b, desconectándolos desde el árbol, se los ordena por timestamp creciente, se los reinserta uno por uno, buscando su punto de inserción desde b. El Algoritmo 13 muestra el proceso de eliminación para esta alternativa.

Es posible hacer dos optimizaciones a la reconstrucción de subárboles, considerando que x es eliminado del subárbol cuya raíz es b (es decir $x \in N(b)$). La primera optimización hace un uso más inteligente de los timestamps. Se puede observar que existen algunos elementos más jóvenes que x que no cambiarán su punto de inserción cuando se los reinserte en el subárbol de b. Estos elementos son aquéllos más viejos que el primer hijo de x y también que el último hermano de x. Para aquellos elementos es posible evitar el cálculo de su nuevo punto de inserción. Para visualizar esto, se puede notar que son los primeros nodos insertados después que x. Esos nodos ya tuvieron la elección de entrar en x, pero ellos eligieron otras opciones (como ellos vinieron antes que el elemento que eligió ser el primer hijo de x). Todos esos nodos tienen, para su punto de reinserción, exactamente las mismas opciones que tuvieron en el momento en que se insertaron exceptuando a x, el cual no prefirieron de todos modos.

Así, ellos eligirán lo mismo de nuevo. La única excepción posible es que, debido a la aridad acotada, ellos se vean forzados a entrar en algún vecino aunque ellos hubieran preferido convertirse en un nuevo vecino de b. Ahora, la ausencia de x les deja espacio para convertirse en vecino de b. Esto es porque se puede asegurar la propiedad sólo hasta el tiempo de inserción del último hermano de x.

Una segunda optimización está basada en que se sabe que los elementos en $A(y)$ están más cerca de y que de cualquiera de sus vecinos más antiguos, y entonces sólo es necesario comparar a y con los vecinos más jóvenes (siempre y cuando se repita el mismo paso de inserción).

3.5.4. Usando Nodos Ficticios

Los costos de eliminación obtenidos son bastante más altos que los costos de inserción, debido a la reconstrucción de subárboles completos. De aquí, este método busca amortizar este costo entre muchas eliminaciones.

Una alternativa para eliminar un elemento x es dejar su nodo en el árbol (sin contenido) y marcarlo como eliminado. Tal nodo se dice ser *ficticio*. Aunque es un método barato y simple de eliminar, se debe averiguar cómo llevar adelante una búsqueda consistente cuando algunos nodos no contienen objetos.

Básicamente, si $b \in N(a)$ es ficticio, no se dispone de suficiente información para evitar entrar al subárbol de b cuando se ha alcanzado al nodo a. Entonces, no se puede incluir a b en la minimización y siempre se entra en su subárbol, excepto si se puede usar la información de timestamp de b para podar la búsqueda.

La búsqueda realizada en la inserción, por otra parte, tiene que seguir sólo un paso en el árbol. En este caso, se es libre de elegir que el nuevo elemento vaya en cualquier vecino del nodo ficticio corriente, o en el vecino más cercano que no sea ficticio. Una buena política, sin embargo, es tratar de no incrementar el tamaño de los subárboles, cuyas raíces sean nodos ficticios, porque ellos eventualmente tendrán que reconstruirse, y también porque se entrará en ellos más frecuentemente durante las búsquedas.

Por lo tanto, aunque la eliminación es simple, el desempeño del proceso de búsqueda se degrada. Por consiguiente, periódicamente se deben descartar los nodos ficticios y realmente eliminarlos. Notar que el costo de reconstrucción de un subárbol no sería demasiado diferente si éste contuviera muchos nodos ficticios, así se puede remover a todos los nodos ficticios con una única reconstrucción, amortizando de esta manera el alto costo de reconstrucción sobre muchas eliminaciones.

La idea es asegurar que cada subárbol tiene a lo sumo una fracción α de nodos ficticios. Se dice que tales subárboles son "densos", en otro caso ellos se consideran "ligeros". Cuando se marca un nuevo nodo $x \in N(a)$ como ficticio, se verifica si no se ha transformado el subárbol en ligero. En este caso, x es realmente eliminado desde el árbol. En el proceso de reinserción de elementos, también se descarta cada nodo ficticio que se encuentra.

Esta técnica tiene una propiedad de desempeño agradable. Si el número de elementos a reinsertar es m, esto es debido a que $\alpha\ m$ de ellos son ficticios y sólo se reinsertarán $(1-\alpha)m$ nodos reales. Por lo tanto, sólo se realizan $(1-\alpha)m$ reinserciones para cada grupo de $\alpha\ m$ eliminaciones que se han realizado. Las reinserciones se sacan de encima aquellos $\alpha\ m$ nodos ficticios, así realmente se está pagando un costo de eliminación amortizado el cual es $(1-\alpha)/\alpha$ veces el costo de una inserción. Asintóticamente, el árbol trabaja como si permanentemente tuviera una fracción α de nodos ficticios. así, se controla el compromiso entre costos de búsqueda y de eliminación.

Una pequeña complicación de este esquema es que al eliminar x puede hacer ligeros a varios ancestros de x, aún si x es sólo una hoja que puede ser removida directamente, aún si el ancestro no tiene

como raíz a un nodo ficticio. Como un ejemplo, se puede considerar un árbol unario de altura $3n$ donde todos los nodos a distancia $3i$ desde la raíz, $i \geq 0$, son ficticios. El árbol es denso para $\alpha = \frac{1}{3}$, pero al eliminar la hoja o marcarla como ficticia hace que cada nodo se vuelva ligero.

Se puede resolver este problema incrementalmente. Al marcar un nodo x como ficticio, se sigue un paso desde x hacia arriba a la raíz del árbol, verificando en cada nodo a si se vuelve ligero o no. Si se encuentra un nodo a que se vuelve ligero, se reconstruye el subárbol completo desde el padre de a, y se continúa verificando hacia arriba. La reconstrucción de un subárbol más bajo hace más probable que los subárboles más altos se vuelvan densos de nuevo, a un costo menor comparado con reconstruir directamente el subárbol ligero más alto.

3.5.5. Hiperplanos Fantasmas

Esta técnica está inspirada en la idea presentada en [UN03] para los *GNAT*s dinámicos [Bri95], llamada *hiperplanos fantasmas*. Este método reemplaza el elemento eliminado por una hoja, la cual es fácil de eliminar. De esta manera no es necesaria la reconstrucción, pero en cambio se debe incorporar alguna tolerancia cuando en una búsqueda se llega al nodo reemplazado.

Los vecinos de un nodo b en el *DSAT* particionan el espacio como en un diagrama de Voronoi, con hiperplanos. Si un elemento y reemplaza a un vecino x de b, los hiperplanos se desplazarán (levemente, si y está cerca de x). Se puede pensar de un hiperplano "fantasma", como el que corresponde al elemento eliminado x, y un hiperplano "real", como el que corresponde al nuevo elemento y. Los datos en el árbol inicial están organizados de acuerdo al hiperplano fantasma, pero nuevas inserciones considerarán el hiperplano real. Una búsqueda debe ser capaz de encontrar a todos los elementos, insertados antes o después de la eliminación de x.

Con este propósito, se mantiene una tolerancia $d_g(x)$ en cada nodo x. Cuando se inserta a x se inicializa en $d_g(x) = 0$. Cuando x se elimina y el contenido de su nodo se reemplaza por y, se coloca $d_g(x) = d_g(x) + d(x, y)$ (el nodo se llama aún x aunque su objeto ahora sea y). Se puede notar que sucesivos reemplazos pueden desplazar los hiperplanos en todas las direcciones de manera tal que la nueva tolerancia debe acumularse a las previas.

Durante las búsquedas, se debe considerar que cada nodo x realmente puede ser compensado por $d_g(x)$ cuando se determina si se debe entrar o no en un subárbol. Por lo tanto, se desea mantener cada valor de $d_g()$ tan pequeño como sea posible; es decir, se pretenden encontrar reemplazos que estén tan cerca del elemento eliminado como sea posible. El Algoritmo 14 ilustra el pseudo-código del algoritmo de búsqueda (Algoritmo 11) modificado para considerar la existencia de planos fantasmas.

Cuando se elimina un nodo x, se busca un sustituto en su subárbol para asegurar que se reducirá el tamaño del problema. En [UN03] se elige una hoja del subárbol descendiendo siempre por el hijo que está más cerca de x. Aunque esto no garantiza que y sea la hoja más cercana a x, se argumenta que realizar una búsqueda del vecino más cercano en el subárbol es demasiado costoso. Se considera esta alternativa de elegir el reemplazo entre las hojas con la misma política. El *DSAT*, sin embargo, tiene una ventaja interesante sobre el *GNAT* en el sentido que los vecinos (es decir, los hijos) de un nodo se eligen por ser cercanos a él, mientras que en el *GNAT* se eligen al azar o por estar alejados entre ellos. En un *DSAT* elegir el reemplazo entre los vecinos del elemento eliminado podría dar un buen candidato para reemplazo a un bajo costo. A continuación se explican estos métodos en detalle.

Eligiendo una hoja sustituta Se desciende en el subárbol de x todas las veces por el hijo más cercano a x. Cuando se alcanza una hoja y, se desconecta al nodo de y desde el árbol y se pone a y en el nodo

Algoritmo 14 Proceso de búsqueda por rango para q with radius r en un *DSAT* con raíz a, modificado para considerar los hiperplanos fantasmas.

BúsquedaRangoTBA-GH (Nodo a, Consulta q, Radio r, Timestamp t)

1. If $time(a) < t \ \wedge \ d(a,q) - d_g(a) \leq R(a) + r$ Then
2. If $d(a,q) \leq r$ Then Informar a
3. $d_{min} \leftarrow \infty$
4. For $b_i \in N(a)$ Do /* en orden creciente de timestamp */
5. If $d(b_i,q) - d_g(b_i) \leq d_{min} + 2r$ Then
6. $t' \leftarrow \min\{t\} \cup \{time(b_j), j > i \wedge d(b_i,q) - d_g(b_i) > d(b_j,q) + d_g(b_j) + 2r\}$
7. **Búsqueda RangoTBA-GH** (b_i,q,r,t')
8. $d_{min} \leftarrow \min\{d_{min}, d(b_i,q) + d_g(b_i)\}$

Algoritmo 15 Proceso para eliminar a x desde un *DSAT*, usando hiperplanos fantasmas y encontrando un sustituto para x entre las hojas de su subárbol.

EliminarGH1(Nodo x)

1. $b \leftarrow parent(x)$
2. If $N(x) \neq \emptyset$ Then
3. $y \leftarrow$ **EncontrarHojaSustituta** (x)
4. $d_g(x) \leftarrow d_g(x) + d(x,y)$
5. Copiar objeto de y el el nodo x
6. Else $N(b) \leftarrow N(b) - \{x\}$

EncontrarHojaSustituta (Nodo x): Nodo

1. $y \leftarrow x$
2. While $N(y) \neq \emptyset$ Do
3. $x \leftarrow y$
4. $y \leftarrow \text{argmin}_{c \in N(b)} d(c,x)$
5. $N(x) \leftarrow N(x) - \{y\}$
6. Retornar y

de x, reteniendo el timestamp original de x. Luego se actualiza el valor d_g del nodo. El Algoritmo 15 ilustra este proceso.

Eligiendo un vecino sustituto Se selecciona a y como el más cercano a x entre $N(x)$ y se copia el objeto y en el nodo de x como antes. Si el anterior y fuera una hoja se la elimina y termina. En otro caso se continúa recursivamente el proceso en ese nodo. Así, se vuelven *fantasmas* todos los nodos en un paso desde x a una hoja de su subárbol, siguiendo los vecinos más cercanos. A cambio, los valores de $d_g()$ deberían ser más pequeños. El Algoritmo 16 muestra este proceso de eliminación.

Eligiendo el elemento más cercano sustituto Se selecciona a y como el elemento más cercano a x entre todos los elementos en el subárbol de x y se copia el objeto y en el nodo de x anterior. Si el anterior nodo y fuese una hoja se la elimina y se termina. En otro caso se continúa recursivamente el proceso en ese nodo. Así, se vuelven *fantasmas* algunos nodos en un paso desde x a una hoja de su subárbol, siguiendo los elementos más cercanos. Los valores $d_g()$ deberían ser más pequeños que con las otras alternativas. El Algoritmo 17 muestra el proceso. BúsquedaNN (x,x,1) invoca al algoritmo que realiza una búsqueda 1-NN para la consulta x en el subárbol de x. Dada la versión de *DSAT* elegida, el algoritmo para realizar esta búsqueda es el que se muestra en el Algoritmo 12, el cual se invocaría como **BúsquedaNNTBA** (x, x,1).

Aún cuando ahora se tiene un elemento en el lugar del nodo del elemento eliminado, que permite no entrar en su subárbol en cada búsqueda, la tolerancia que introduce $d_g(x)$ es también un factor

Algoritmo 16 Proceso para eliminar x desde un *DSAT*, usando hiperplanos fantasmas, y eligiendo su reemplazo entre sus vecinos.

EliminarGH2 `(Nodo` x`)`

1. $b \leftarrow parent(x)$
2. `If` $N(x) \neq \emptyset$ `Then`
3. $\quad y \leftarrow \text{argmin}_{c \in N(x)} d(c, x)$
4. $\quad d_g(x) \leftarrow d_g(x) + d(x, y)$
5. $\quad$ `Copiar el objeto de` y `en el nodo` x
6. $\quad$ **EliminarGH2** `(`y`)`
7. `Else` $N(b) \leftarrow N(b) - \{x\}$

Algoritmo 17 Eliminación de x desde un *DSAT*, usando hiperplanos fantasmas, y eligiendo su reemplazo como su elemento más cercano en su subárbol.

EliminarGH3 `(Nodo` x`)`

1. $b \leftarrow parent(x)$
2. `If` $N(x) \neq \emptyset$ `Then`
3. $\quad y \leftarrow$ **BúsquedaNNTBA** `(`x`,`x`,`1`)`
4. $\quad d_g(x) \leftarrow d_g(x) + d(x, y)$
5. $\quad$ `Copiar el objeto de` y `en el nodo` x
6. $\quad$ **EliminarGH3** `(`y`)`
7. `Else` $N(b) \leftarrow N(b) - \{x\}$

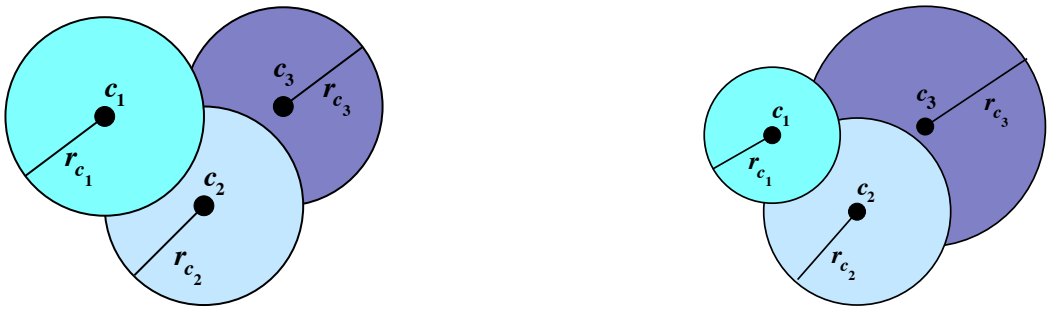
significativo en el empeoramiento de la calidad de la búsqueda. Así, en un régimen permanente que incluya eliminaciones, periódicamente hay que deshacerse de los hiperplanos fantasmas y reconstruir el árbol para eliminarlos. De la misma manera que con los nodos ficticios, cuando se reconstruye un subárbol se remueven todos los hiperplanos fantasmas que están dentro de él. Por lo tanto, se puede aplicar exactamente el mismo mecanismo usado en la Sección 3.5.4 para controlar la cantidad de nodos ficticios. Se establece una proporción máxima permitida de α hiperplanos fantasmas, y se reconstruye el árbol cuando se excede este límite. Además, los métodos que usan α permiten controlar los costos de eliminación esperados como una proporción del costo de inserción.

Experimentalmente en [NR08] se muestra que usando valores de $\alpha < 10\,\%$ se puede obtener un compromiso razonable entre costos de eliminación y búsqueda. La alternativa de reemplazar el nodo eliminado por un vecino tiene un desempeño levemente peor que las otras. Esto se debe probablemente al alto número de hiperplanos fantasmas que se introducen.

3.6. Lista de Clusters

La *Lista de Clusters* (*LC*) [CN00a, CN05] es una técnica efectiva y muy simple para indexar espacios métricos de mediana a alta dimensión. Pertenece también a la clase de algoritmos basados en particiones compactas.

La *LC* logra intercambiar costo de construcción por costo de búsqueda. Otras estructuras para poder lograr cierta eficiencia en espacios de alta dimensión utilizan más espacio para lograr mejorar las



(a) Particiones de radio fijo.

(b) Particiones de tamaño fijo.

Figura 3.4: Políticas de selección de radios para *LC*.

búsquedas. Sin embargo, el espacio de memoria a usar tiene un límite, que es más estricto que el del tiempo de construcción y además el espacio se necesitará durante todo el uso del índice mientras que el tiempo de construcción se gastará sólo una vez.

3.6.1. Proceso de Construcción

Para construir la *LC* se debe elegir un "centro" $c \in \mathbb{X}$ y un radio r_c. Se define la *zona centrada* o *bola centrada* (c, r_c) como el subconjunto de elementos de $\mathbb{U}$ que se encuentran a distancia a lo más r_c de c. Se definen además dos conjuntos de $\mathbb{X}$:

$$I_{\mathbb{X},c,r_c} = \{x \in \mathbb{X} - \{c\}, d(c,x) \leq r_c\}$$

como el "bucket" o "balde" de elementos *internos*, que caen dentro de la bola centrada (c, r_c), y:

$$E_{\mathbb{X},c,r_c} = \{x \in \mathbb{X}, d(c,x) > r_c\}$$

como los elementos *externos*. Entonces, el proceso se repite recursivamente con el conjunto E.

Existen dos maneras simples de particionar el espacio, considerando la selección del radio:

- Particiones de radio fijo: la alternativa más simple es seleccionar a r_c como un radio fijo r^* para todas las bolas de la lista. Esto implica que, avanzando en la lista, las bolas se vuelven más vacías.
- Particiones de tamaño fijo: otra elección es considerar un número fijo de elementos m dentro de cada bola centrada (c, r_c), y definir r_c en función de ello. Es decir, en este caso r_c corresponde a la máxima distancia entre el centro c y sus m vecinos más cercanos en $\mathbb{X}$. Esto permite además fijar el largo de la lista a $\frac{n}{(m+1)}$. Por lo tanto, a medida que se avance en la lista, las bolas serán espacialmente más amplias.

La Figura 3.4 ilustra las dos maneras que considera *LC* de particionar el espacio en $\mathbb{R}^2$, para los tres centros c_1, c_2 y c_3, tomados en ese orden.

Algunos de los aportes de este trabajo han utilizado como base a la *LC*. Así, se ha considerado tanto la versión que trabaja con la partición con tamaño fijo m, como también la versión con radio fijo para cada partición, dependiendo de lo que convenía a cada propuesta.

El proceso de construcción genérico, independiente de la política de selección de radio, se muestra en el Algoritmo 18. El operador ":" corresponde al constructor de la lista y Λ indica lista vacía. La salida

Algoritmo 18 Proceso de construcción genérico de la *Lista de Clusters*.

```
ConstruirLC (Conjunto de Elementos X)

1.  If X = ∅ Then Informar Λ    /* informar lista vacía */
2.  Seleccionar un centro c ∈ X
3.  X ← X − {c}
4.  Seleccionar un radio r_c
5.  I ← {x ∈ X, d(x, c) ≤ r_c}
6.  E ← X − I
7.  Informar (c, r_c, I): ConstruirLC (E)
```

de la construcción es una lista de triuplas (c_i, r_i, I_i), esto es *(centro, radio, bucket)*. La estructura de datos que se construye parece bastante simétrica, pero no lo es. El primer centro elegido tienen preferencia sobre los centros subsiguientes en caso que se superpongan sus zonas, como se puede observar en la parte izquierda de la Figura 3.5. Todos los elementos que caen dentro de la bola o zona del primer centro (c_1 en la figura) se almacenan en su bucket I (como es el caso del elemento x de la figura), a pesar que ellos podrían también caer dentro de los buckets I de los subsecuentes centros (c_2 y c_3 en la figura). Consecuentemente, este hecho queda reflejado en las búsquedas (Algoritmo 19). La parte derecha de la Figura 3.5 muestra también la estructura de lista resultante. El costo de construir la *LC* es $O(n^2/m)$ para particiones de tamaño fijo m y es $O(n^2/p^*)$ para particiones de radio fijo r^*, donde p^* es el tamaño promedio de las particiones obtenidas.

Para instanciar este algoritmo, de acuerdo a la política de selección de radio, sólo bastaría con modificar lo siguiente:

- Particiones de radio fijo: en la línea 4 se debe asignar a r_c el radio fijo r^* elegido: $r_c \leftarrow r^*$.
- Particiones de tamaño fijo: en este caso la línea 4 se debe sustituir por: $I \leftarrow kNN_X(c)$, $r_c \leftarrow \text{máx}_{x \in I}\{d(x, c)\}$.

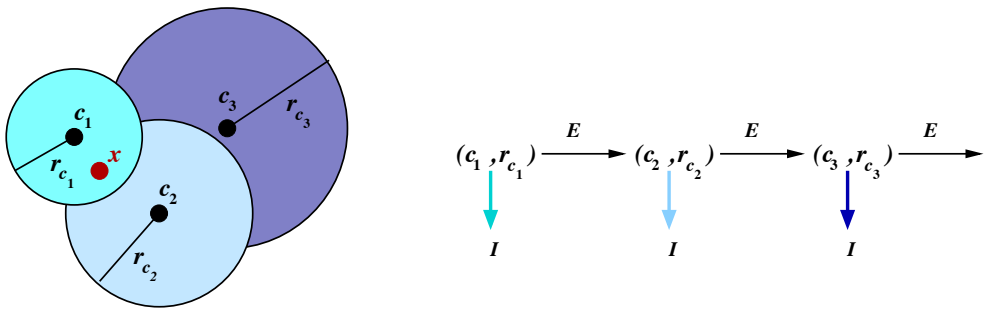
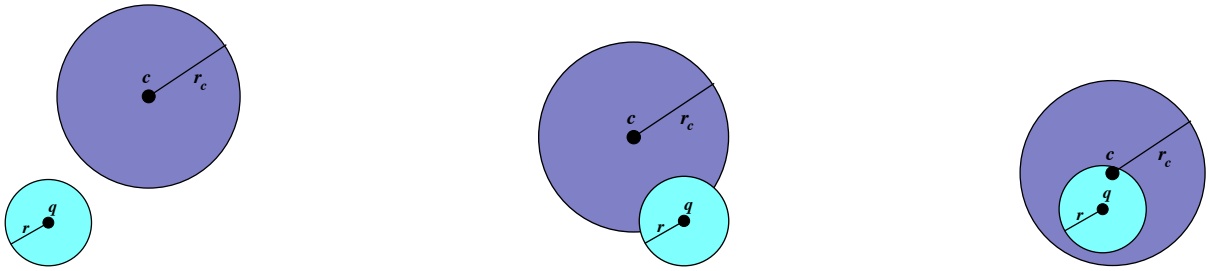


Figura 3.5: Ejemplo de construcción de *LC* para tres centros c_1, c_2 y c_3 tomados en ese orden, y la lista resultante.

Cabe destacar que, a pesar de que la *LC* se construya con particiones de tamaño fijo, en cada triupla se almacena el radio de cobertura real de la zona centrada en c, es decir la máxima distancia entre c y los elementos $x \in I$.

En [CN05] se han presentado diferentes heurísticas para seleccionar los centros (acción que se realiza en la línea 2 del Algoritmo 18), aunque experimentalmente allí se ha mostrado que la mejor manera



(a) No intersecta a I.

(b) Intersecta a I y sigue con E.

(c) Intersecta a I y no sigue con E.

Figura 3.6: Posibles situaciones de una búsqueda por rango (q, r) en *LC*.

hacerlo es elegir como centro al elemento que maximice la suma de las distancias a los centros anteriores.

3.7. Búsquedas

El proceso de búsqueda sobre la *LC* es también bastante simple y para evitar cálculos de distancia aplica el criterio de radio cobertor. La idea es que si el primer centro en ser elegido es c y su radio es r_c, la búsqueda de una consulta por rango (q, r) comienza calculando $d(q, c)$ y agregando c a la respuesta si $d(q, c) \leq r$. Luego, se debe buscar exhaustivamente en el bucket I de c sólo si la cola de consulta tiene intersección con la bola centrada en c. Luego de considerar el primer "cluster", se sigue la búsqueda con E. Sin embargo, dada la asimetría de la estructura de datos, se puede también podar la búsqueda de otra manera: *si la bola de consulta está estrictamente contenida en la bola de radio r_c centrada en c, entonces no se debe considerar a E, como por construcción se sabe que todos los elementos que caigan dentro de la bola de consulta se tienen que haber incluido en I*. Esta característica esencial no está presente en otros algoritmos basados en particiones compactas, donde la búsqueda debe entrar en todas las particiones o zonas que intersecten la bola de consulta. Con *LC*, la consideración de particiones relevantes se puede adelantar tan pronto la bola de consulta esté totalmente contenida en una partición. La Figura 3.6 ilustra las distintas situaciones que pueden aparecer al considerar la búsqueda de (q, r) respecto de un cluster (c, r_c).

El Algoritmo 19 exhibe el proceso de búsqueda por rango sobre *LC*. Cabe destacar que en la línea 3 se calcula la distancia $d(q, c)$ y luego en las líneas 4 y 7 ya es conocida y no vuelve a calcularse.

Algoritmo 19 Proceso de la búsqueda por rango en *LC*.

BúsquedaRangoLC (Lista L, Consulta q, Radio r)

```
1.  If L = Λ Then Return
2.  Sea L = (c, r_c, I) : E
3.  If d(c,q) ≤ r Then Informar c
4.  If d(c,q) ≤ r_c + r Then    /* búsqueda exhaustiva en I */
5.    For x ∈ I Do
6.      If d(q,x) ≤ r Then Informar x
7.  If d(c,q) > r_c − r Then BúsquedaRangoLC (E, q, r)
```

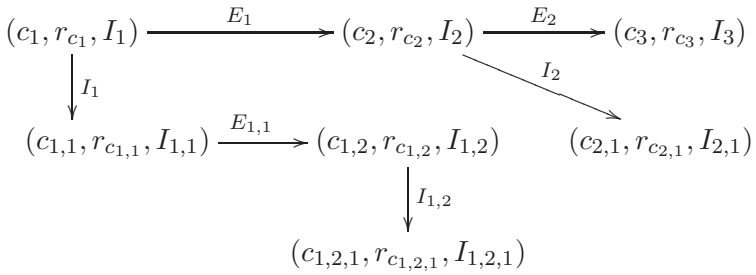


Figura 3.7: Construcción de la Lista de Clusters Recursiva.

3.7.1. Lista de Clusters Recursiva

En [Mam05] se propone una variante de *LC* denominada *Lista de Clusters Recursiva* (*RLC*). La *RLC* se basa en la versión de particiones de radio fijo de la *LC* y así, como ya se mencionó, los clusters resultantes pueden no tener la misma cantidad de elementos. Más aún, se puede verificar experimentalmente que los primeros clusters están más densamente poblados, mientras que los últimos contienen sólo el centro.

El algoritmo para construir la *RLC* es muy similar al de la *LC*. Cada nodo de la *RLC* es una triupla (c, r_c, I), correspondiendo a *(centro, radio fijo, bucket interno)*. Una vez seleccionado el conjunto I de objetos del bucket del nodo corriente, se continúan agregando nodos en la *RLC* al procesar el resto de los objetos E. La diferencia con *LC* se debe al siguiente hecho clave. Si el tamaño del bucket interno I es demasiado grande, mayor que alguna constante pequeña m, se construye recursivamente una *RLC* para los elementos de I. La Figura 3.7 muestra el inicio del proceso de creación de una *RLC*. Este algoritmo necesita tiempo $O(n \log_\beta n)$ para construir la *RLC*, para alguna constante $1 < \beta < 2$, lo cual es mejor que el costo de la *LC*.

El algoritmo de búsqueda, por lo tanto, debe ser levemente modificado para soportar el hecho de que el bucket interno I puede ser un conjunto de a lo sumo m elementos, o una *RLC* en sí misma. Así, basta con modificar en el Algoritmo 19 las líneas 5 y 6 para que consideren este hecho.

A pesar de que la *RLC* se presenta en [Mam05] como una mejora sobre *LC* y como una manera de permitir inserciones en la *LC*, se ha mostrado que el desempeño de las búsquedas de la *RLC* es levemente superior a *LC* en espacios de vectores de dimensión menor que 12, uniformemente distribuidos [Mam05].

Capítulo 4

Optimización de Índices

En este capítulo se presentan algunas optimizaciones obtenidas a estructuras de datos estáticas para búsquedas por similitud en espacios métricos que han permitido, además de lograr un mejor desempeño de las mismas, obtener un conocimiento mucho más profundo de las razones que determinan su comportamiento [CLRR11, CLRR14, CLRR16, GCMR08, GCMR09].

En uno de los casos la optimización surgió por la búsqueda de respuestas a los motivos que provocaban situaciones que eran contraintuitivas (Sección 4.1). En otro de los casos por la aplicación de técnicas propuestas en [UPN09, NU11] a otra estructura dinámica (Sección 3.5.1 y Sección 3.5.5). Finalmente, en el caso restante, por la posibilidad de almacenar distancias ya calculadas en la construcción del índice, si se dispone de espacio, para su posterior aprovechamiento en las búsquedas. En todos los casos las optimizaciones se han realizado sobre índices diseñados para memoria principal.

4.1. Optimización al *SAT*

Como se ha mencionado en el Capítulo 3, el *SAT* es un índice estático basado sobre una aproximación alternativa a la del común de las estructuras de datos para espacios métricos: más que dividir el espacio de búsqueda, acercarse espacialmente a la consulta. Se comienza desde un punto dado de la base de datos e iterativamente se acerca a la consulta [Nav99, Nav02, HS03a]. Además de ser el *SAT* algorítmicamente interesante por sí mismo, se ha mostrado que obtiene un atractivo balance entre uso de memoria, tiempo de construcción y desempeño de búsqueda.

Por otra parte, el *DSAT* [NR08] es una versión en línea del *SAT*. Se ha diseñado para ser totalmente dinámico, permitiendo inserciones y eliminaciones sin incrementar los costos de construcción respecto del *SAT*. Una muy sorprendente e inesperada característica del *DSAT* es que logró mejorar además el desempeño en las búsquedas. El *DSAT* es más rápido en las búsquedas aún si la construcción tiene menos información que la versión estática del índice. Para el *DSAT* la base de datos es desconocida de antemano y los objetos se insertan en el índice al azar al igual que las consultas. Una estructura de datos totalmente dinámica no puede hacer suposiciones fuertes sobre la base de datos y no tendrá estadísticas sobre la base de datos completa. Inversamente, el *SAT* es una estructura estática la cual, en principio, podría aprovechar el *conocimiento completo* de la base de datos. Sin embargo, en [NR08] se muestra que el *DSAT* es más eficiente en las búsquedas que el *SAT*. Este comportamiento aparentemente contraintuitivo ha sido desconcertante por algún tiempo.

4.1.1. Motivación

El *SAT* en su proceso de construcción elige la raíz del árbol y eso determina un árbol; es decir, hay un árbol fijo por cada elección de la raíz. Con el fin de ilustrar algunos conceptos, se presenta en la Figura 4.1 un ejemplo de una base de datos métrica en $\mathbb{R}^2$ con distancia Euclidiana. Las Figuras 4.2 y 4.3 muestran dos posibles *SAT*s sobre dicho espacio, que se obtienen al seleccionar p_{12} o p_6 como las raíces del árbol respectivamente. Se resaltan los vecinos de la raíz y se exhiben también sus radios de cobertura. Como puede observarse los árboles obtenidos son completamente distintos, lo cual probablemente provoque que puedan tener diferentes costos de búsqueda.

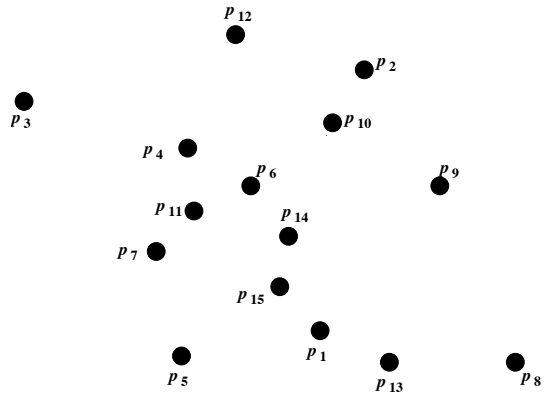


Figura 4.1: Ejemplo de una base de datos métrica en $\mathbb{R}^2$.

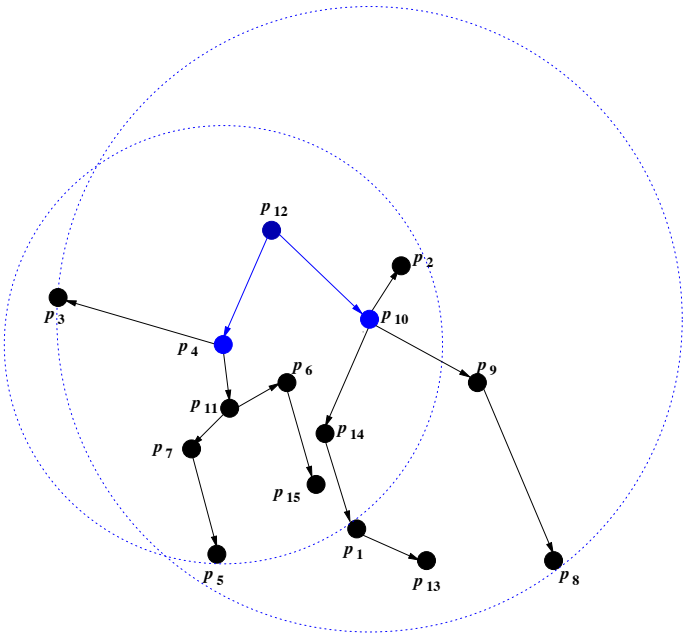


Figura 4.2: Ejemplo de *SAT* obtenido al elegir a p_{12} como raíz.

Cabe destacar que el proceso de construcción del *SAT* no persigue obtener una estructura de datos "balanceada". Aunque esto es claramente una deventaja en el ámbito de las estructuras de datos para búsqueda exacta, en estructuras de datos métricas parece que el "desbalance" acelera las búsquedas (para una discusión más detallada se pueden leer las Secciones 7.1 y 7.2 de [CN05]). Más aún, bajo el modelo de complejidad que mide la cantidad de cálculos de distancias, la estructura de datos más

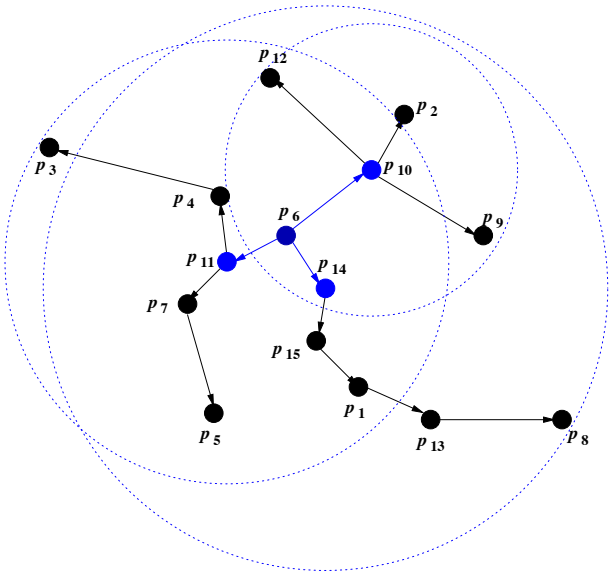


Figura 4.3: Ejemplo de *SAT* obtenido al elegir a p_6 como raíz.

eficiente para búsquedas por similitud en espacios métricos de alta dimensión es la *Lista de Clusters* (*LC*), la cual puede verse como un árbol extremadamente desbalanceado.

Si los objetos de la base de datos no se conocen de antemano, el *SAT* no puede construirse. Así el *DSAT*, en lugar de examinar todos los posibles elementos para determinar el conjunto de vecinos, los va seleccionando a medida que van llegando los elementos. La versión elegida para construir el *DSAT* fija la aridad máxima del árbol, aunque en [NR08] se estudian otras maneras de construirlo. En comienzo se consideró que la aridad jugaba un rol muy importante en la eficiencia de las búsquedas.

Las Figuras 4.4 y 4.5 muestran dos posibles *DSAT*s construidos sobre el espacio de la Figura 4.1. En este caso se ha considerado que los elementos $p_1, \cdots, p_{15}$ se insertaron de a uno a la vez. El primer elemento en ser considerado, p_1, será la raíz. La diferencia entre ambas figuras radica en la elección de la aridad máxima permitida, 2 y 6 respectivamente. Nuevamente, en ambos casos, se ilustran los radios de cobertura de los vecinos de la raíz. Algunos radios de cobertura son iguales a cero en la Figura 4.5, porque los respectivos subárboles sólo contienen la raíz. Claramente, los árboles dinámicos obtenidos tendrán probablemente diferentes costos de búsqueda.

En [NR08], se ha mostrado experimentalmente que el *DSAT* supera a la versión estática y se han sugerido un par de reglas prácticas para determinar el valor de su parámetro de máxima aridad:

(a) Bajas aridades son buenas para bajas dimensiones intrínsecas y para pequeños radios de búsqueda, y

(b) Altas aridades se pueden usar para espacios de alta dimensión intrínseca.

Se describen a continuación nuestros hallazgos y nuestras propuestas de optimización.

4.2. Árbol de Aproximación Espacial Distal

Existen dos diferencias conceptuales entre el *SAT* y el *DSAT* que podrían en comienzo justificar por qué el *DSAT* supera al *SAT* en las búsquedas:

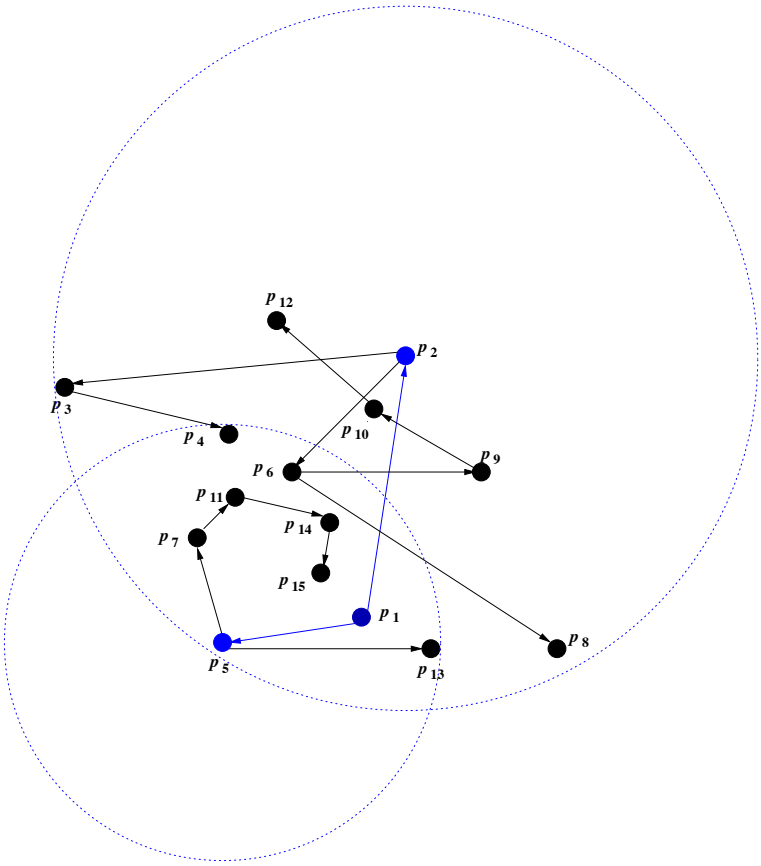


Figura 4.4: Ejemplo del *DSAT* obtenido con aridad máxima de 2.

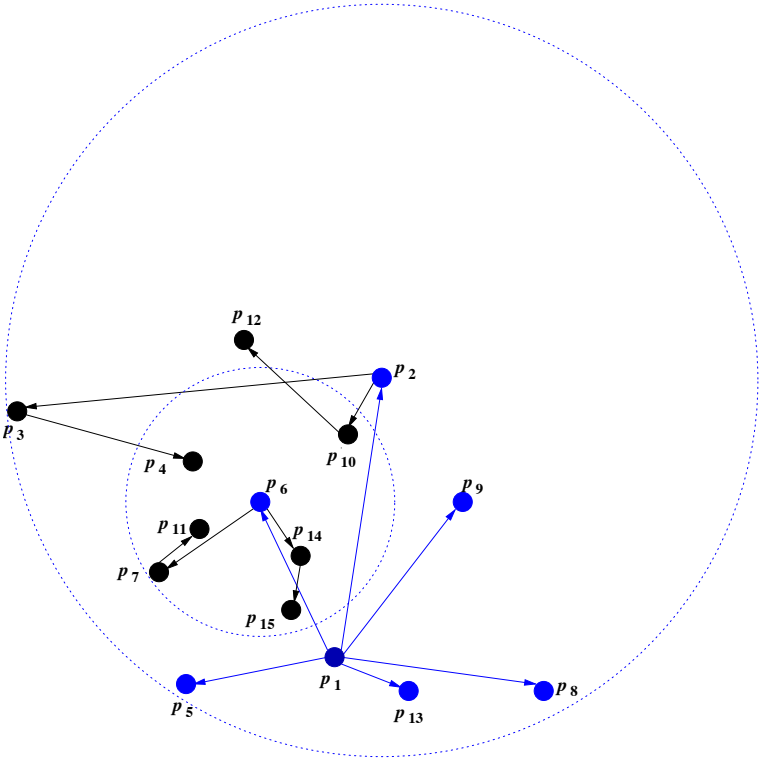
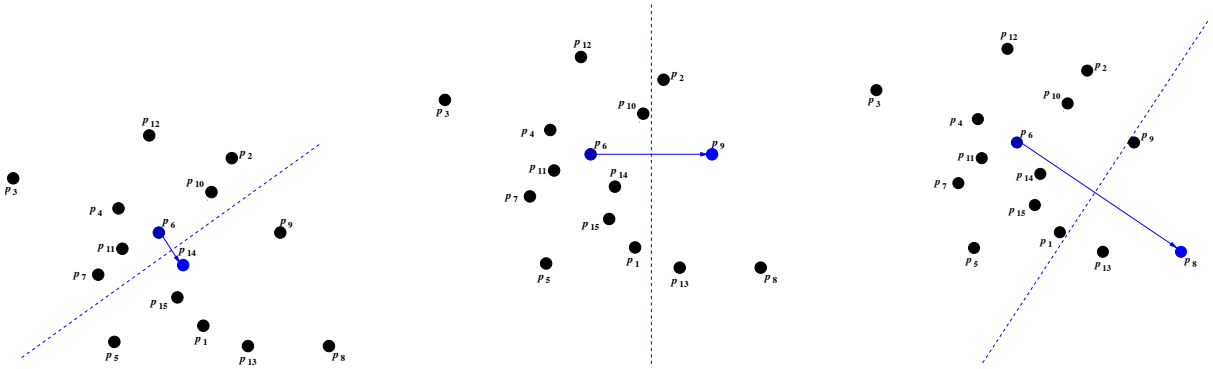


Figura 4.5: Ejemplo del *DSAT* obtenido con aridad máxima de 6.



(a) Eligiendo p_{14}. (b) Eligiendo p_9. (c) Eligiendo p_8.

Figura 4.6: División del espacio al construir el *SAT* con raíz p_6 y con distintos primeros vecinos.

(a) qué orden de los elementos se utiliza para determinar el conjunto de vecinos, y

(b) si la aridad máxima del árbol se fija (*DSAT*) o no (*SAT*).

A partir de la definición del *SAT* en el Algoritmo 1, inicialmente el conjunto de vecinos de la raíz a, $N(a)$, es vacío. Esto implica que se puede seleccionar *cualquier* elemento de la base de datos como primer vecino. Una vez que este primer vecino se ha establecido, la base de datos se divide en dos partes por el hiperplano definido por proximidad a a y al vecino recientemente seleccionado. Cualquier elemento en el lado de a se puede seleccionar como segundo vecino. Mientras que la zona de la raíz (aquellos elementos de la base de datos que están más cerca de la raíz que de los vecinos previos) no sea vacía, es posible continuar con la subsiguiente elección de vecino.

Las Figuras 4.6(a), 4.6(b) y 4.6(c) ilustran cómo se divide el espacio de la Figura 4.1 si se elige p_6 como raíz del *SAT* y como primer vecino a p_{14} (como lo hubiera elegido el *SAT* original), p_9 y p_8 respectivamente.

Ordenar los elementos en orden creciente de distancia a la raíz es sólo uno de las $(n-1)!$ posibles permutaciones de los elementos de la base de datos. Cada permutación de la base de datos se puede usar como un orden para la construcción del *SAT*. Cada orden de inserción producirá una versión *correcta* del *SAT*; es decir, un árbol que satisface la Propiedad 2 del *SAT* [Nav02], y por lo tanto se puede usar el mismo algoritmo de búsqueda. Es muy probable que el desempeño de las búsquedas sea diferente para cada permutación, entonces una pregunta natural es: ¿*cuál es la mejor permutación para una base de datos dada*?. En lugar de tratar ciegamente de probar cada permutación, se intentan optimizar las reglas de descarte del *SAT*. Se evita entrar a un subárbol utilizando dos reglas o criterios: hiperplanos y radios de cobertura. La característica clave del criterio de hiperplano es la separación entre los elementos que lo definen, debido a que la bola de consulta más probablemente caerá de un único lado del hiperplano y así todos los objetos en el lado opuesto se podrán descartar. Una buena separación de hiperplanos en los niveles superiores del árbol también implica que los radios de cobertura serán menores en los niveles más bajos del árbol. Entonces, se aprovechan estas dos observaciones usando varias heurísticas en el *Árbol de Aproximación Espacial Distal* (*DiSAT*). Llamativamente, la política original para el *SAT* trabaja exactamente en la dirección opuesta de esta estrategia de mejora. Aún una selección *aleatoria* del orden de inserción supera al *SAT* original; esto explica por qué la versión dinámica era mejor que la estática.

Algoritmo 20 Proceso para construir un *SAT*$^+$ con raíz a.

```
Construir (Nodo a, Conjunto de elementos X)

 1. N(a) ← ∅    /* vecinos de a */
 2. R(a) ← 0    /* radio de cobertura */
 3. Fijar un orden π en el conjunto X
 4. For v ∈ X de acuerdo al orden π Do
 5.    R(a) ← máx(R(a), d(v,a))
 6.    If ∀b ∈ N(a),  d(v,a) < d(v,b) Then N(a) ← N(a) ∪ {v}
 7. For b ∈ N(a) Do S(b) ← ∅    /* subárboles */
 8. For v ∈ X − N(a) Do
 9.    c ← argmin_{b∈N(a)} d(v,b)
10.    S(c) ← S(c) ∪ {v}
11. For b ∈ N(a) Do Construir (b, S(b))    /* construir subárboles */
```

4.2.1. La Estrategia *SAT*$^+$

El Algoritmo 20 da una descripción formal del proceso de construcción de la estructura de datos. La diferencia con el *SAT* está en la selección del orden de inserción π en la línea 3. El orden que se elige en este caso es de más lejano desde la raíz a más cercano. La búsqueda se realiza con el procedimiento estándar descripto en el Algoritmo 2.

Una permutación aleatoria, o equivalentemente un orden aleatorio, para la construcción del *SAT* es similar a insertar los elementos en un *DSAT* a medida que van llegando. La diferencia sólo será que tendrá un número *natural* de vecinos en lugar de una aridad arbitraria que se debe sintonizar. Esta versión del *DiSAT* se ha denominado *SAT*^{Rand}. Se ha verificado experimentalmente esta alternativa de construcción con el fin de explicar el comportamiento del *DSAT*.

Cuando se trabaja con hiperplanos para separar los datos es aconsejable usar pares de objetos lejanos uno de otro, como se documenta en [CNBYM01] para las estructuras de datos de *GNAT* y *GHT*. Usando las observaciones anteriores, se puede afirmar que se obtiene una buena separación de los hiperplanos implícitos al elegir el primer vecino como el elemento más alejado de la raíz. Claramente es aconsejable hacerlo recusivamente, en cada nodo del árbol. Vale la pena notar que esta heurística es exactamente la opuesta del orden original que se utiliza en la construcción del *SAT*.

En el ejemplo de la Figura 4.6(c) la elección del primer vecino corresponde a la estrategia de *SAT*$^+$, por ser p_8 el elemento más alejado a p_6. Esta elección hace que el primer hiperplano que se obtenga sea el más lejano a p_6 posible.

Las Figuras 4.7 y 4.8 ilustran el *SAT*$^+$ que se obtiene usando el mismo ejemplo de espacio que aparece en la Figura 4.1, con las mismas raíces p_{12} y p_6 respectivamente. Como puede observarse, las formas de los árboles son muy diferentes a los que mostraron en las Figuras 4.2 y 4.3. Se puede notar que los radios de cobertura de los vecinos de la raíz son significativamente más pequeños que los del *SAT* original. Experimentalmente se prueba esto en la próxima sección, para diferentes bases de datos.

4.2.2. La Estrategia *SAT*^{Glob}

Ordenar en cada nivel los elementos por distancia puede consumir tiempo. Por lo tanto, la estrategia *SAT*^{Glob} fija un orden de inserción de los elementos π por distancia a la raíz del árbol, más lejano

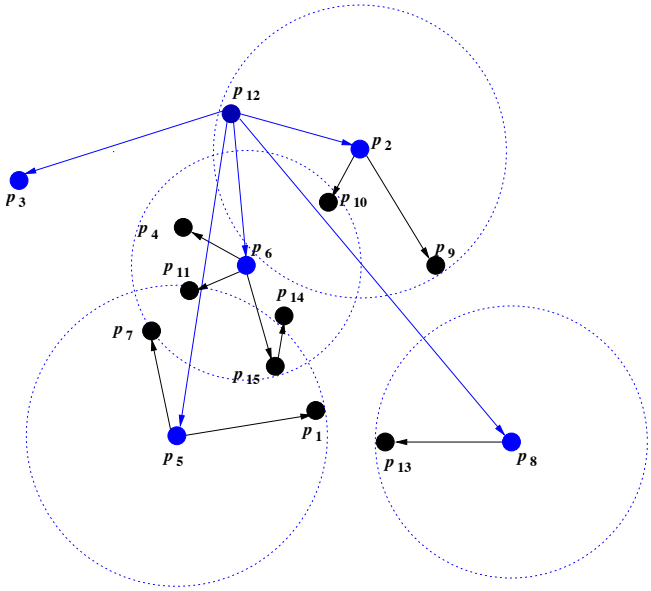


Figura 4.7: Ejemplo del *SAT*$^+$ seleccionando como raíz a p_{12}.

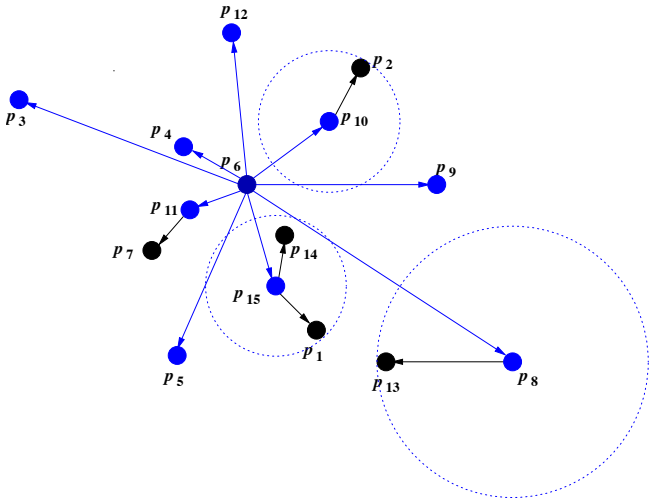


Figura 4.8: Ejemplo del *SAT*$^+$ al seleccionar a p_6 como la raíz del árbol.

Algoritmo 21 Proceso de construcción de un SAT^{Glob} con raíz a.

Construir (Nodo a, Orden π en el conjunto X)

1. $N(a) \leftarrow \emptyset$ /* vecinos de a */
2. $R(a) \leftarrow 0$ /* radio de cobertura */
3. For $v \in X$ de acuerdo al orden π Do
4. $R(a) \leftarrow \max(R(a), d(v,a))$
5. If $\forall b \in N(a),\ d(v,a) < d(v,b)$ Then $N(a) \leftarrow N(a) \cup \{v\}$
6. For $b \in N(a)$ Do $S(b) \leftarrow \emptyset$ /* subárboles */
7. For $v \in S - N(a)$ Do
8. $c \leftarrow \text{argmin}_{b \in N(a)} d(v,b)$
9. $S(c) \leftarrow S(c) \cup \{v\}$
10. For $b \in N(a)$ Do **Construir** (b, $S(b)$) /* construir subárboles */

primero. Este orden π fijo se usa en cada uno de los siguientes niveles del árbol. Por lo tanto, SAT^{Glob} y SAT^{+} son similares sólo en el primer nivel del árbol, en los siguientes niveles el orden π ya se ha determinado y se usa sin volver a realizar ningún nuevo ordenamiento. Esto también sirve para evaluar la necesidad recursiva de seleccionar buenos hiperplanos en cada nivel del árbol. El Algoritmo 21 muestra el proceso formal de la construcción del SAT^{Glob}. Nuevamente la búsqueda usa el Algoritmo 2.

Las Figuras 4.9 y 4.10 muestran ejemplos de SAT^{Glob}, sobre el mismo espacio métrico ilustrado en la Figura 4.1, con las mismas raíces p_{12} y p_6 respectivamente. Nuevamente, las formas de los árboles obtenidos son diferentes de las de los árboles de las Figuras 4.3 y 4.2. Se puede observar también que los radios de cobertura son significativamente más pequeños que los del *SAT* original.

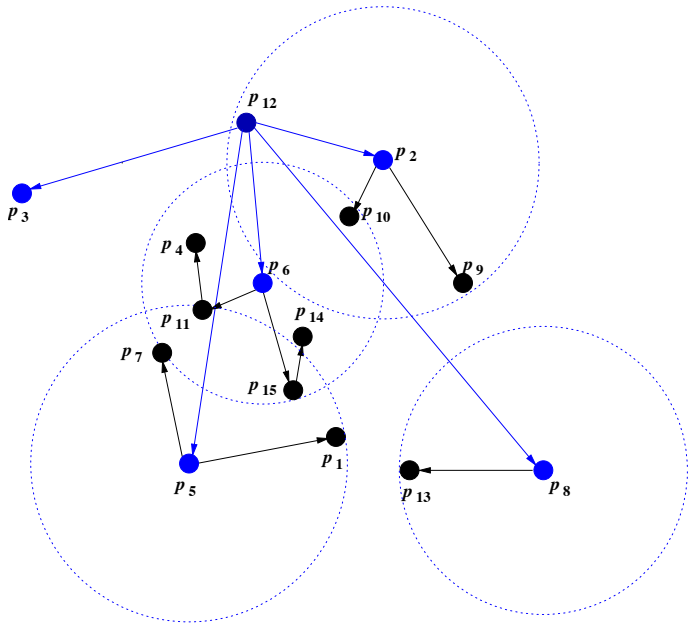


Figura 4.9: Ejemplo del SAT^{Glob} seleccionando a p_{12} como raíz.

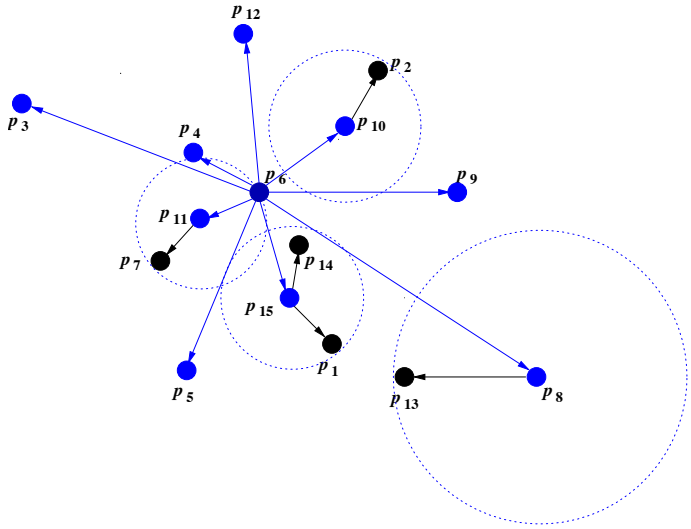


Figura 4.10: Ejemplo del SAT^{Glob} con p_6 seleccionado como raíz.

4.2.3. La Estrategia SAT^{Out}

Hasta ahora se ha seleccionado un elemento aleatorio como la raíz del árbol. Sin embargo, dado que se está buscando maximizar la separación del hiperplano entre la raíz y el primer vecino, tiene mucho sentido seleccionar el par más alejado en la base de datos como la raíz y el primer vecino respectivamente. De esta manera, existirá mucho espacio para seleccionar pares alejados en los niveles más bajos del árbol.

El problema de encontrar "el par de elementos más alejados" es bien conocido. Se buscan objetos $x, y \in \mathbb{X}$, tales que $d(x, y) \geq d(z, v), \forall z, v \in \mathbb{X}$. Esto puede hacerse comparando todos contra todos a los elementos de la base de datos, lo cual es prohibitivamente costoso porque involucra $O(n^2)$ cálculos de distancia. Una versión aleatorizada es muy efectiva y sólo necesita $O(n)$ cálculos de distancia. La idea es seleccionar un elemento u_0 al azar y ubicar su elemento más alejado u_1, determinando el par (u_0, u_1). Luego, dado u_1 encontrar su elemento más alejado u_2 (no necesariamente coincidirá con u_0), obteniendo el par (u_1, u_2) y así sucesivamente. Unas pocas iteraciones de este proceso obtendrán una buena aproximación al par de elementos más alejados.

Para construir un SAT^{Out} se invoca primero la selección de la raíz para determinar el elemento a

Algoritmo 22 Proceso para seleccionar la raíz de un SAT^{Out}.

Seleccionar-raíz `(Conjunto de elementos` X`)`
1. `Sean` x `e` y `elementos de` X
2. $d_{max} \leftarrow 0$
3. `While` $d(x, y) > d_{max}$ `Do`
4. $d_{max} \leftarrow d(x, y)$ /* distancia máxima actual */
5. $v \leftarrow \text{argmax}_{z \in X - \{y\}} d(x, z)$
6. `If` $d(x, v) > d(x, y)$ `Then`
7. $y \leftarrow v$ /* actual elemento más alejado de x */
8. `intercambiar (`x, y`)` /* intercambia el rol de x e y */
9. `Retornar` x

(como **Select-root** ($\mathbb{X}$)) y luego se invoca el Algoritmo 20 para construir un SAT^{+} con a y $\mathbb{X} - \{a\}$, y nuevamente se considera el orden π de $\mathbb{X} - \{a\}$ de más lejano a más cercano respecto de la raíz a.

Las Figuras 4.11 y 4.12 muestran el SAT^{Out} que se obtiene, usando nuevamente el mismo espacio ilustrado en la Figura 4.1. En este caso la raíz se selecciona invocando el Algoritmo 22. En la Figura 4.11 la búsqueda del par de objetos más alejados comenzó en p_6, y en la Figura 4.12 en p_5.

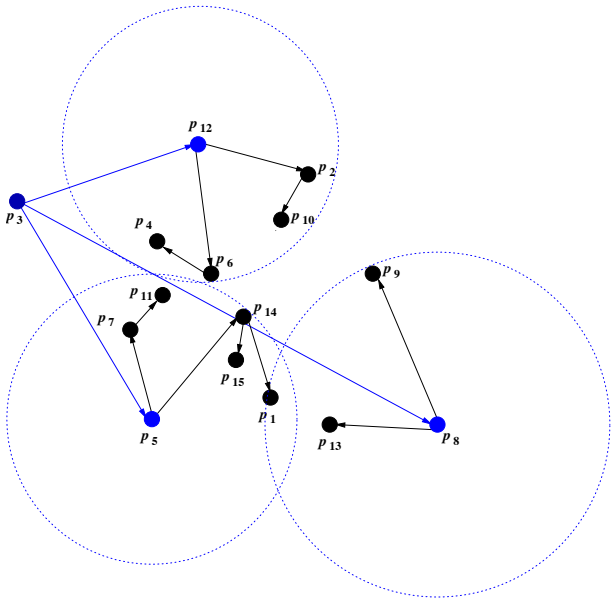


Figura 4.11: Ejemplo de SAT^{Out} con p_3 como la raíz del árbol.

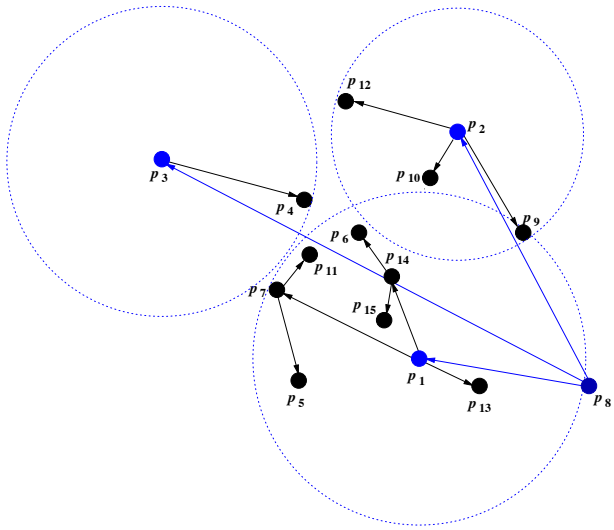


Figura 4.12: Ejemplo de SAT^{Out} con p_8 como la raíz.

4.3. Resultados Experimentales

Para los primeros experimentos en la evaluación se utilizan las tres bases de datos métricas reales, usadas como puntos de referencia, descargadas desde la Biblioteca Métrica de SISAP [FNC07]: *Imágenes de la NASA*, el *Diccionario* e *Histogramas de Color*.

Para realizar la evaluación experimental de los índices sobre estos espacios métricos se realizan los experimentos de construcción y búsqueda, tal como se describieron en la Sección 2.11.

Los códigos fuentes del *SAT* y *DSAT* utilizados se encuentran también disponibles en `www.sisap.org`. El parámetro de aridad del *DSAT*, el cual puede sintonizarse y corresponde al número máximo de vecinos de cada nodo del árbol, se seleccionó de acuerdo a la recomendación dada en [NR08]. Para los espacios de Imágenes de la NASA e Histogramas de Color la aridad es de 4 y para el Diccionario de 32.

4.3.1. Eligiendo la Raíz del Árbol

Como se ha mencionado, cada selección de la raíz en el *SAT* induce una forma diferente del árbol. Es posible que alguna de ellas tenga mejor desempeño en las búsquedas sobre el promedio. Intuitivamente, es posible que la mejor raíz pudiera ser un elemento "central" en el espacio; es decir, un elemento que minimice el radio de cobertura de la raíz, Sin embargo, es demasiado costoso seleccionar el elemento "central" del espacio métrico, porque se necesitan al menos $O(n^2)$ cálculos de distancia. Sin embargo, una buena heurística para un elemento central se usa en el *Algoritmo de Selección de Centroide* (*CSA* por su sigla en inglés) [PMCV06]. Este algoritmo se basa en el algoritmo *HF* presentado en la *Familia Omni* [FTJF01].

Con el fin de evaluar si es posible mejorar significativamente el desempeño de las búsquedas eligiendo la raíz de una manera más inteligente, se han comparado los costos de búsqueda de este método *SAT(CSA)* con el *SAT* y *DSAT* originales y con las variantes del *DiSAT*. La Figura 4.13 exhibe esta comparación.

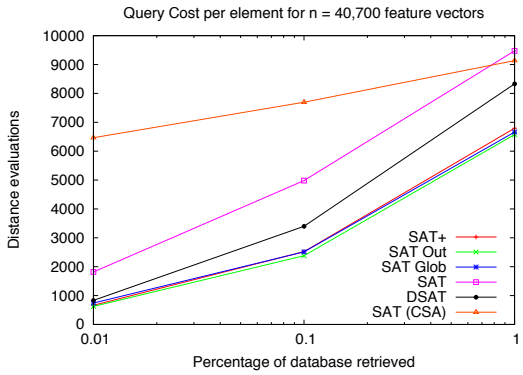
Como puede observarse, *CSA* sólo mejora el desempeño de las búsquedas en el *SAT* en uno de los tres espacios métricos considerados. Más aún, en los otros dos espacios obtiene o igual o peor desempeño que el *SAT*. Sin embargo, aún en ese espacio, su mejora es inferior a la obtenida por el SAT^{+}, el SAT^{Out} y el SAT^{Glob}. Por consiguiente, aunque la selección de la raíz pueda eventualmente mejorar al *SAT*, no es el aspecto clave para alcanzar generalmente una mejora significativa en los costos de búsqueda.

4.3.2. Restringiendo la Aridad Máxima

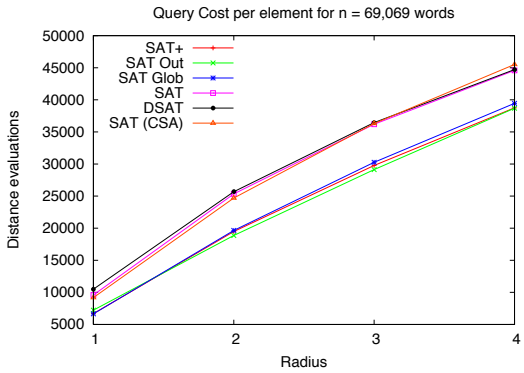
La Figura 4.14 contiene los resultados de los costos de construcción obtenidos en los experimentos sobre estos tres espacios métricos. Se muestra la comparación respecto de los costos del *SAT* original, el *DSAT* y una versión del *SAT* en la que se ha limitado la aridad máxima del árbol a $MaxArity = 4, 8, 16$ y 32. Como se puede observar *SAT* obtiene los peores costos de construcción en dos espacios. Esto se debe a que en aquellos espacios se producen árboles con aridad mayor, y como ya se ha mencionado, en [NR08] se muestra que los costos de construcción crecen con la aridad del árbol. Además, los costos de construcción del *SAT* son equivalentes a los obtenidos en el *SAT* con aridad máxima limitada a 32. La Figura 4.15 muestra que limitar la aridad máxima del árbol no es la razón clave por la cual el *DSAT* supera al *SAT* en las búsquedas. El *DSAT* tiene mejor desempeño en las búsquedas con la misma aridad que el *SAT* con aridad acotada en la mayoría de los espacios.

4.3.3. Diferentes Órdenes de Inserción

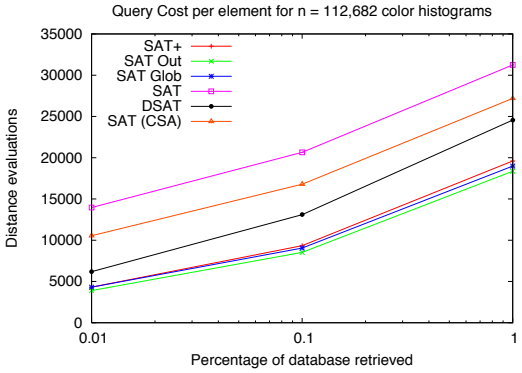
La Figura 4.16 contiene los resultados de los costos de construcción obtenidos en los experimentos para los tres espacios métricos. Se muestra la comparación de los costos de construcción para el *SAT* original, para el *DSAT* y para las nuevas alternativas que usan los nuevos criterios de construcción:



(a) Imágenes de la NASA.



(b) Diccionario.



(c) Histogramas de Color.

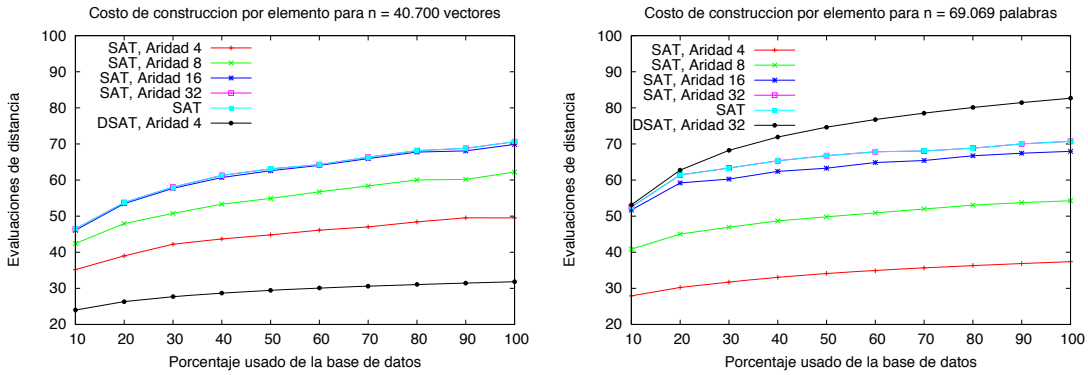
Figura 4.13: Comparación de los costos de búsqueda por rango entre el *SAT*, el *DSAT*, las mejores variantes del *DiSAT* y la versión del *SAT* que usa el algoritmo CSA para seleccionar la raíz.

SAT^{Rand}, SAT^{+}, SAT^{Glob} y SAT^{Out}. Como se puede observar, el SAT^{+} obtiene los peores costos de construcción. Este hecho puede ser explicado porque la aridad en esta opción es la más amplia [NR08]. Más aún, a pesar de que SAT^{Out} usa la misma política de selección de vecinos que SAT^{+}, SAT^{Out} obtiene mejores costos de construcción porque la aridad máxima del árbol es significativamente menor que la de SAT^{+} debido a que se selecciona más adecuadamente la raíz del árbol. La Figura 4.17 muestra que los nuevos SAT^{+}, SAT^{Glob} y SAT^{Out} mejoran significativamente los costos de búsqueda con respecto a las otras y que éstos son similares entre sí. Sin embargo, SAT^{Out} alcanza los costos de búsqueda más bajos.

Se postula que en los nuevos índices los vecinos de la raíz representan una muestra más precisa de las diferentes zonas en el espacio métrico y producen mejor separación de los hiperplanos en los dos sentidos, la separación entre hermanos y la separación entre los nodos y la raíz. Estas dos condiciones también implican radios de cobertura pequeños. Esto a su vez produce una partición más compacta del espacio, mejorando los costos de búsquedas.

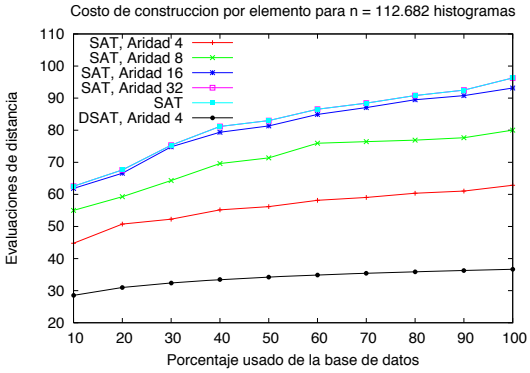
4.3.4. Dependencia de la Dimensionalidad Intrínseca

El principal desafío en las búsquedas por similitud es la dimensión. Cada índice conocido se degrada a una búsqueda secuencial si la dimensión intrínseca de los datos es mayor que un cierto umbral. Un buen índice métrico debería ser resistente a la maldición de la dimensión, o en otras palabras, se debería degradar el desempeño de la búsqueda lentamente a medida que se incrementa la dimensión intrínseca. Una manera de testear experimentalmente un índice es usando datos con una dimensión conocida, e



(a) Imágenes de la NASA.

(b) Diccionario.



(c) Histogramas de Color.

Figura 4.14: Comparación de los costos de construcción, para el *SAT*, *DSAT* y la versión de *SAT* con aridad acotada.

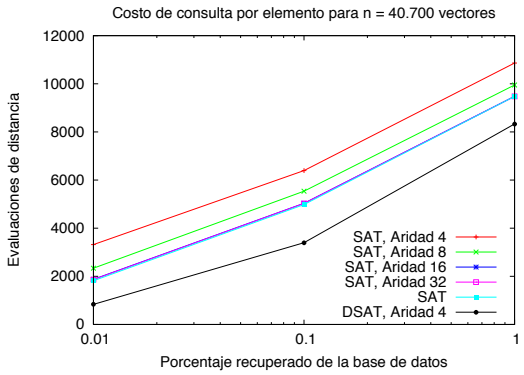
incrementarla de manera controlada.

Para analizar cómo la dimensionalidad intrínseca afecta el comportamiento de las diferentes técnicas, se evalúan todas las nuevas heurísticas de construcción sobre espacios métricos sintéticos. Se han usado colecciones de 100.000 vectores de dimensión $2, 4, 8$ y 16, uniformemente distribuidos en el hipercubo unitario. Como se ha mencionado, explícitamente no se utiliza la información de coordenadas de cada vector. En la comparación se usa para el *DSAT* la mejor aridad obtenida para cada dimensión; esto es, aridad 4 para las dimensiones 2 y 4, aridad 8 para dimensión 8 y aridad 32 para dimensión 16.

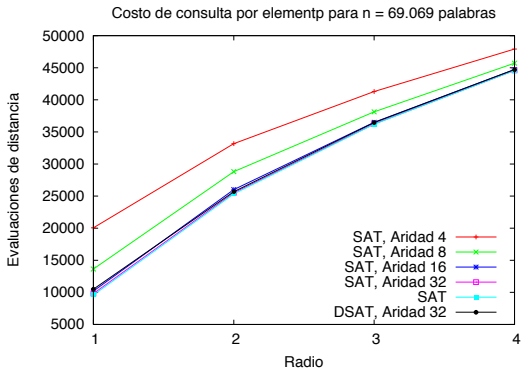
La Figura 4.18 muestra los costos de construcción obtenidos para cada dimensión. El *DSAT* obtiene mejores costos de construcción, seguido de cerca por el *SAT*, con excepción del espacio de dimensión 2. Las nuevas propuestas son más complejas de construir, siendo la alternativa de SAT^{Out} la menos costosa de ellas. Es sorprendente ver que el *SAT* para dimensión 2 es más costosa de construir que el *SAT* para dimensión 4. En el primer caso el árbol es más de 10 veces más profundo que en el segundo caso y también los radios de cobertura son más grandes.

La Figura 4.19 exhibe el comportamiento de las búsquedas de los diferentes métodos a medida que crece la dimensión. Las Figuras 4.19(a), 4.19(b) y 4.19(c) muestran respectivamente los costos de recuperar el $0.01\,\%$, $0.1\,\%$, y $1\,\%$ de la base de datos.

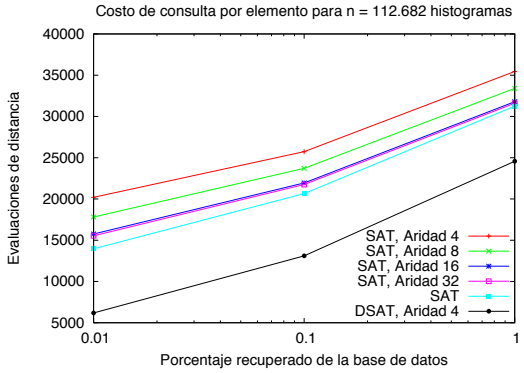
Como es posible observar, el SAT^{Out} supera a los demás métodos, seguido de cerca por SAT^{+} y SAT^{Glob}. La característica común en ellos es el uso de vecinos distantes.



(a) Imágenes de la NASA.



(b) Diccionario.



(c) Histogramas de Color.

Figura 4.15: Comparación de los costos de las búsquedas por rango, para *SAT*, *DSAT* y *SAT* con aridad acotada.

4.3.5. Escalabilidad

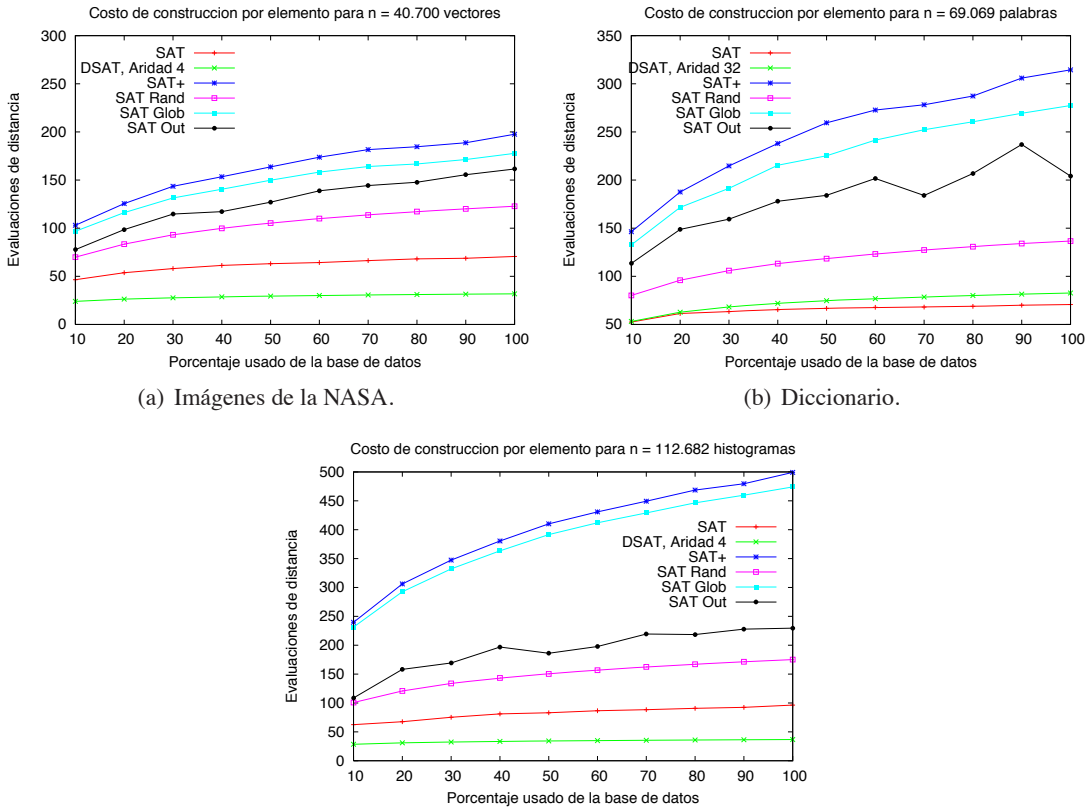
El conjunto previo de experimentos muestra que las nuevas estrategias de construcción superan al *SAT* original en las búsquedas y también son más resistentes a la maldición de la dimensionalidad. Para investigar la escalabilidad se considera el conjunto de datos reales *Flickr*, obtenido desde la colección de imágenes de SAPIR [FKM+07], descripto en la Sección 2.10. Como se ha mencionado, este conjunto contiene un millón de imágenes, cada una representada por un vector de 208 componentes.

Las Figuras 4.20(a) y 4.20(b) ilustran los costos de construcción y búsqueda respectivamente, para la base de datos de *Flickr*.

En la Figura 4.20(b) se puede notar que el SAT^{Out} es el más rápido en las búsquedas, seguido de cerca por SAT^{+} y SAT^{Glob}. Como era de esperar SAT^{Out} escala bien con el tamaño de la base de datos.

4.4. Hiperplanos vs. Radios de Cobertura

Una pregunta natural sobre la mejora del *SAT* es: ¿cuál regla es responsable de la eficiencia en las nuevas estructuras de datos? ¿Radios de cobertura o hiperplano? El *SAT* usa ambas reglas, como así también las versiones del *DiSAT* que usan distintas políticas de inserción. Se quiere investigar cuál regla es mejor para filtrar en las búsquedas. En este conjunto de experimentos se comparan los costos de búsqueda del *SAT* y de las variantes del *DiSAT*, contra las búsquedas usando sólo hiperplanos o radios



(a) Imágenes de la NASA.

(b) Diccionario.

(c) Histogramas de Color.

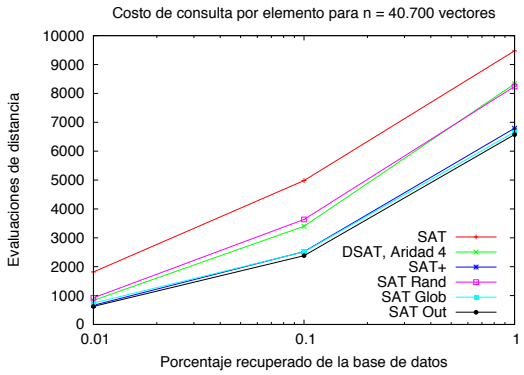
Figura 4.16: Comparación de los costos de construcción para las distintas variantes del *DiSAT*.

de cobertura; sólo uno de los dos. En estos experimentos se usan las consultas por rango con el radio de búsqueda del vecino más cercano. La Tabla 4.1 exhibe la comparación para todos los espacios métricos considerados en los experimentos previos, considerando la cantidad de evaluaciones de distancia.

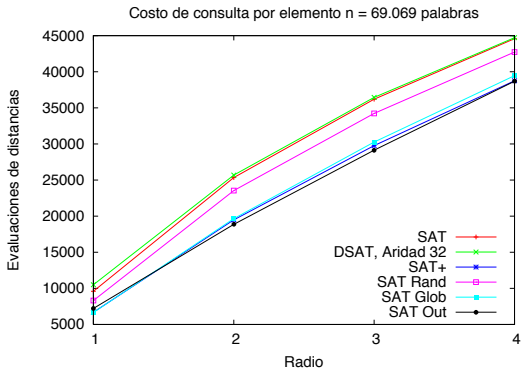
Como se mencionó, se ha mejorado explícitamente la regla del hiperplano. Sin embargo, es evidente desde los experimentos, que implícitamente también se han mejorado los radios de cobertura. Se puede observar que los costos de búsqueda podando sólo por hiperplanos aún son mejores que los costos de búsqueda del *SAT* que usa ambos criterios. Otro aspecto interesante es la observación que a medida que la dimensión intrínseca crece, la regla de hiperplano apenas se usa, a excepción de los hiperplanos de los vecinos de la raíz del árbol. Para dimensión intrínseca alta, usar sólo hiperplanos no trabaja demasiado bien, y descartar sólo por radios de cobertura da buenos resultados. En cualquier caso se reducen los costos de búsquedas del *SAT* usando un único criterio de descarte. La conclusión de los experimentos es que el radio de cobertura se mejora en los *DiSAT*, aún si sólo se manipula la separación de los hiperplanos.

4.5. Comparación con otros Índices

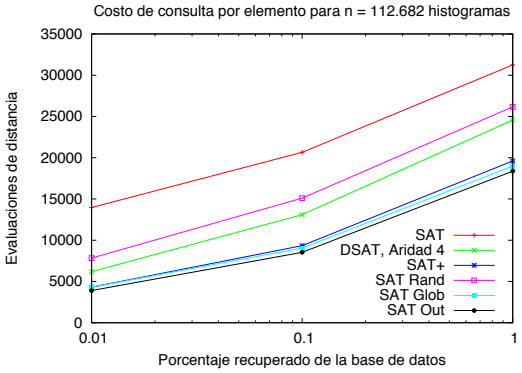
Se comparan las mejores alternativas de SAT^{+} y SAT^{Out}, con otros índices en la literatura. En la primera parte se miden sólo cálculos de distancia. Se han seleccionado estructuras de datos cuyos códigos fuentes se encuentran disponibles. La mayoría de estas estructuras son estáticas. Se incluye también al *M–tree*, que es una estructura de datos dinámica, debido a que es un importante referente en la literatura.



(a) Imágenes de la NASA.



(b) Diccionario.



(c) Histogramas de Color.

Figura 4.17: Comparación de costos de búsqueda por rango para las distintas variantes del *DiSAT*.

Entre todos los índices exactos AESA [Vid86] se establece como el límite inferior en cálculos de distancia; sin embargo, usa espacio cuadrático. Por lo tanto, para realizar una comparación justa, se utilizarán solamente índices que necesitan espacio lineal.

4.5.1. M–tree

El *M–tree* [CPZ97] es probablemente la estructura de datos dinámica más conocida. Su código fuente está disponible [1].

El *M–tree* también tiene buen desempeño en memoria secundaria, aunque en esta sección sólo es de interés medir el número de cálculo de distancias. Se ha usado la sintonización de parámetros sugerida por los autores [CPZ97]. Se han evaluado unas pocas alternativas de parametrización para minimizar los cálculos de distancia en las búsquedas (es posible reducir los costos de construcción incrementando los costos de búsqueda, pero se han priorizado las búsquedas). El *M–tree* usa más memoria que las alternativas de SAT^{+} y SAT^{Out}, debido a que los nodos internos replican elementos como objetos de enrutamiento. Además, si m es la aridad máxima del *M–tree*, cada nodo interno almacena $2m$ distancias y m punteros.

Se comparan los costos de construcción y búsqueda del *M–tree*, con las tres bases de datos: Imágenes de la NASA, Diccionario e Histogramas de Color. La Figura 4.21 muestra los resultados de la comparación de los costos de construcción sobre los tres espacios métricos y la Figura 4.22 exhibe los resultados

[1] At `http://www-db.deis.unibo.it/research/Mtree/`

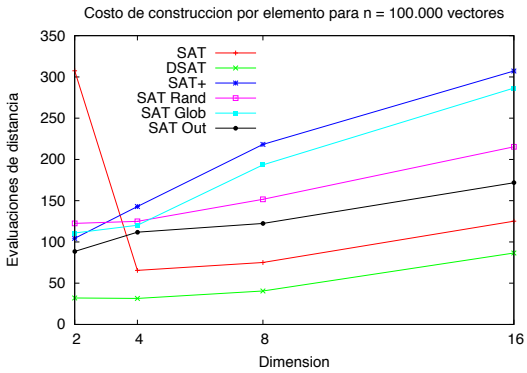
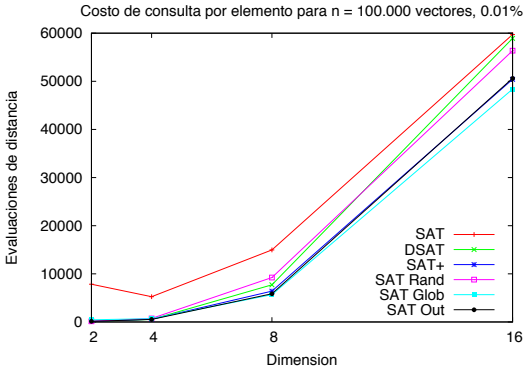
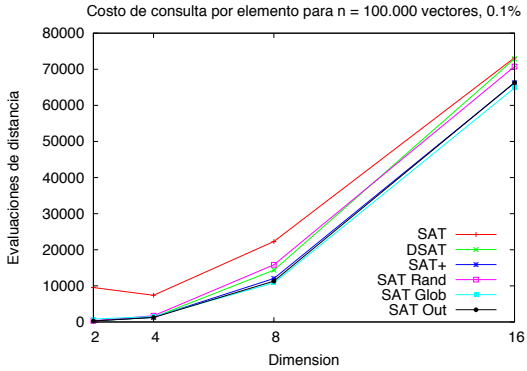


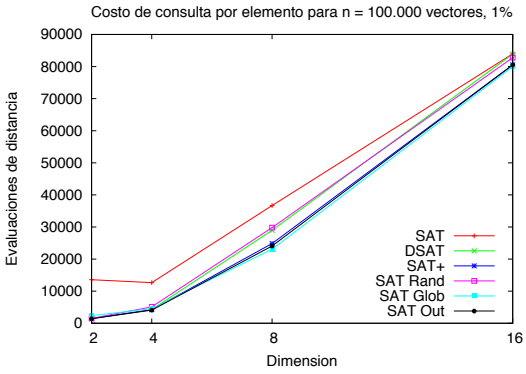
Figura 4.18: Comparación de costos de construcción del *DiSAT* a medida que crece la dimensión.



(a) Recuperando el 0.01 % de la base de datos.



(b) Recuperando el 0.1 % de la base de datos.

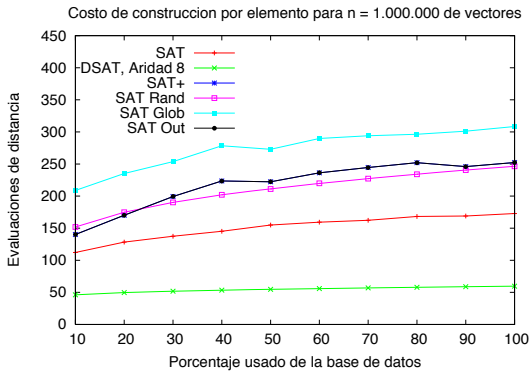


(c) Recuperando el 1 % de la base de datos.

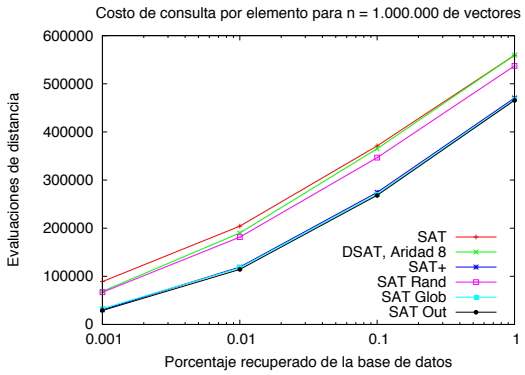
Figura 4.19: Comparación de los costos de búsqueda del *DiSAT* a medida que crece la dimensión.

de los experimentos de búsqueda.

Como se puede ver, el *M–tree* requiere hasta 10 veces menos cálculos de distancia que las alternativas de *DiSAT* para la construcción. Si por otra parte se considera el desempeño en las búsquedas, ambos SAT^{+} y SAT^{Out} superan al *M–tree*, el cual computa cerca de cuatro veces más distancias. Más aún, SAT^{+} y SAT^{Out} tienen otra ventaja práctica sobre el *M–tree*: no es necesario sintonizar ningún parámetro, mientras que la parametrización del *M–tree* no es trivial.



(a) Costos de construcción.



(b) Costos de búsqueda.

Figura 4.20: Comparación de costos del *DiSAT* para la base de datos *Flickr*.

4.5.2. Pivotes

Se realiza ahora la comparación contra un índice genérico de pivotes, una tabla de pivotes. El algoritmo genérico de pivotes elige k pivotes al azar. Para construir la tabla se necesitan kn cálculos de distancia y se supone que sólo es necesario el espacio para almacenar las kn distancias a los pivotes. Esto podría implicar un recorrido lineal durante las búsquedas, pero existen implementaciones sublineales de las tablas de pivotes [CMN01].

La comparación básica podría permitir que la tabla de pivotes use la misma cantidad de memoria que las alternativas de *SAT*$^{+}$ y *SAT*^{Out}. Se testea también cuánta más memoria necesitaría la tabla de pivotes para superar a *SAT*$^{+}$ y *SAT*^{Out}.

Como se ha establecido en [Nav02], para el *SAT* original, una implementación compacta de cualquiera de las versiones del *DiSAT* necesita lo siguiente: por cada objeto, un bit adicional que indique si es o no hoja más el identificador del objeto (20 bits bastan para 1.000.000 de elementos); para cada nodo interno, un radio de cobertura (32 o 64 bits, dependiendo de la arquitectura, con formato flotante o entero); un puntero al primer hijo (20 bits) y el número de hijos (9 bits es más que suficiente para estos experimentos). Esto permite una representación *en ancho* (*breadth-first*) para cualquier *SAT* sobre un arreglo. En estos experimentos, existen a lo sumo dos hojas por nodo interno, así en total se necesitan cerca de $41n$ bits (o $73n$ bits dependiendo de la arquitectura). Particularmente, para todos los espacios métricos considerados, el espacio de memoria necesario para los nuevos índices es realmente menor que 41 bits por elemento. Por otra parte, se necesitan 32 o 64 bits para representar una distancia, de acuerdo a la arquitectura utilizada, y así el espacio mínimo para k pivotes es de $32k$ bits (o $64k$ bits). Por lo tanto, permitiendo que el algoritmo de pivotes use el espacio de 2 pivotes (aunque esto realmente significa al menos un 56 % más espacio que *SAT*$^{+}$ o *SAT*^{Out}). Por consiguiente, en lo que sigue, *Pivotes(s)* equivale a decir que se usan $k = 2s$ pivotes, porque esto es una buena aproximación a usar s veces la cantidad de memoria necesaria por *SAT*$^{+}$ o *SAT*^{Out} (dándoles a los pivotes suficiente margen).

La Figura 4.23 compara los costos de búsqueda de *SAT*$^{+}$, *SAT*^{Out} y el algoritmo genérico de pivotes, usando valores para s desde 1 a 16. Como puede observarse, si se limita el número de pivotes al mismo espacio que necesitan las nuevas alternativas, *SAT*$^{+}$ y *SAT*^{Out} son siempre mejores. Para superarlas para todos los radios de búsqueda considerados, la tabla de pivotes necesita usar mucha más memoria, unas 8 veces para el espacio de Imágenes de la NASA, 16 veces para el Diccionario y 4 veces para los Histogramas de Color. Más aún, ambos *SAT*$^{+}$ y *SAT*^{Out} toleran mejor radios más grandes.

Tabla 4.1: Comparación experimental de los costos de búsqueda de las nuevas heurísticas, medidos en evaluaciones de distancia, analizando la poda por hiperplanos y por radios de cobertura.

(b) Imágenes de la NASA.

	Ambos	Hiper.	Radio Cober.
SAT	344.4	601.7	2369.8
SAT^{+}	266.1	299.9	1127.5
SAT^{Glob}	309.2	446.2	1048.5
SAT^{Out}	238.2	272.0	1025.0

(c) Diccionario.

	Ambos	Hiper.	Radio Cober.
SAT	9594.9	31793.6	17619.5
SAT^{+}	6689.2	16627.2	14586.5
SAT^{Glob}	6655.6	18855.1	14195.5
SAT^{Out}	7221.5	17835.3	14636.4

(e) Histogramas de Color.

	Ambos	Hiper.	Radio Cober.
SAT	4443.5	16422.7	13305.9
SAT^{+}	1133.8	1837.1	4733.0
SAT^{Glob}	1215.6	2501.0	4150.4
SAT^{Out}	916.2	1635.0	3324.3

(f) Base de Datos Flickr.

	Ambos	Hiper.	Radio Cober.
SAT	19512.3	90379.2	58046.2
SAT^{+}	5281.7	11929.6	21251.0
SAT^{Glob}	6231.4	21774.8	19213.0
SAT^{Out}	4994.1	11802.8	18598.6

(h) Vectores de dim. 2.

	Ambos	Hiper.	Radio Cober.
SAT	4931.5	12109.6	12175.2
SAT^{+}	111.6	115.9	194.1
SAT^{Glob}	287.5	540.6	680.9
SAT^{Out}	96.7	100.9	187.9

(i) Vectores de dim. 4.

	Ambos	Hiper.	Radio Cober.
SAT	3171.8	20671.1	4909.8
SAT^{+}	277.8	403.0	620.2
SAT^{Glob}	324.1	844.3	630.2
SAT^{Out}	247.5	370.4	571.2

(k) Vectores de dim. 8.

	Ambos	Hiper.	Radio Cober.
SAT	10513.3	77707.9	10652.0
SAT^{+}	3323.1	19990.8	4297.3
SAT^{Glob}	2966.1	24755.7	3602.0
SAT^{Out}	2928.7	19762.0	3742.8

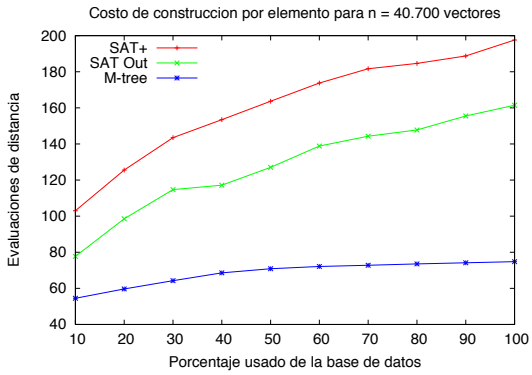
(l) Vectores de dim. 16.

	Ambos	Hiper.	Radio Cober.
SAT	46997.7	89998.0	46997.8
SAT^{+}	36984.6	89139.4	37027.2
SAT^{Glob}	34739.2	89145.2	34770.5
SAT^{Out}	37170.3	89553.2	37205.0

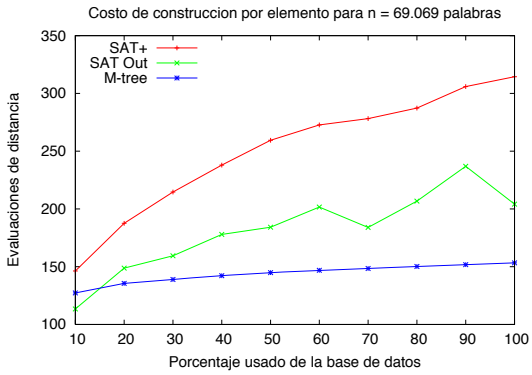
4.5.3. Bisector–Tree

El *Bisector–tree* (*BST*) [KM83] es uno de los índices basados en particiones compactas, que usa criterio de radio cobertor para podar en las búsquedas. Se puede mejorar el desempeño usando también el criterio del hiperplano. Este criterio es también compartido por el *Generalized-Hiperplane Tree* (*GHT*) [Uhl91b]. Los códigos fuentes de esta versión mejorada del *BST* están disponibles en la Biblioteca de SISAP [FNC07].

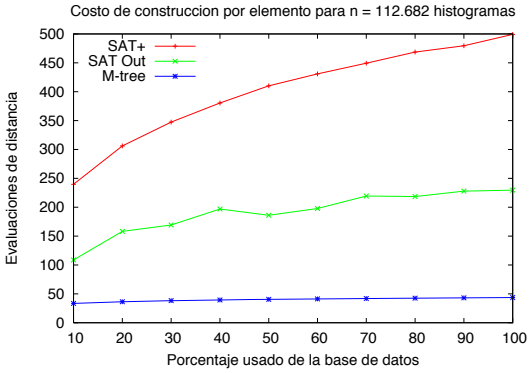
La Figura 4.24 compara los costos de construcción y la Figura 4.25 compara el desempeño de las búsquedas. El *BST* necesita también espacio $O(n)$, tal como el *DiSAT*. Como se puede notar, el *BST* obtiene mejores costos de construcción, SAT^{+} y SAT^{Out} necesitan hasta 11 veces más cálculos de distancia. Sin embargo, en las búsquedas, SAT^{+} y SAT^{Out} son hasta 3 veces más rápidos que el *BST*.



(a) Imágenes de la NASA.



(b) Diccionario.



(c) Histogramas de Color.

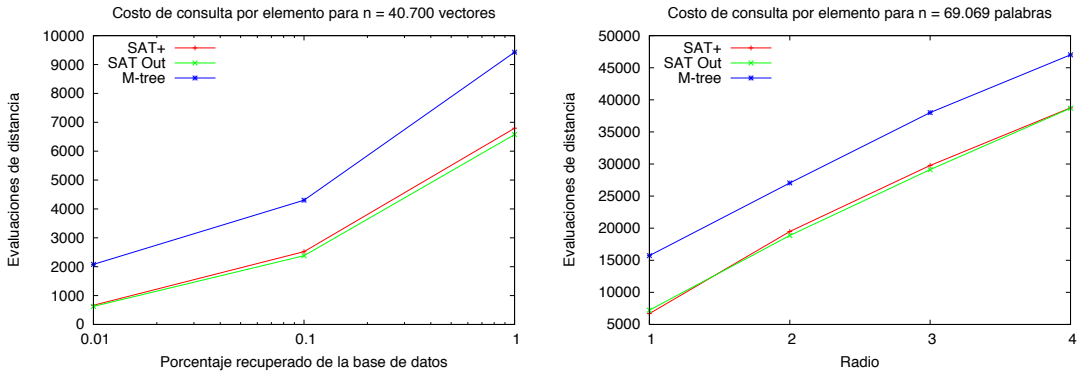
Figura 4.21: Comparación de costos de construcción entre el *DiSAT* y el *M–tree*.

4.5.4. Lista de Clusters

La *Lista de Clusters* (*LC*) [CN05], como ya se mencionó en la Sección 3.6, es otra estructura basada en particiones compactas que, bajo la medida de complejidad de cálculos de distancia y con la selección apropiada de su parámetro, se erige como uno de los índices destacados en la literatura. Se recuerdan aquí algunos detalles de *LC*, con el fin de poner en contexto la comparación.

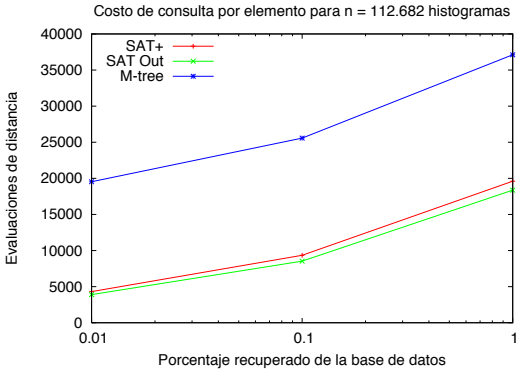
Para construir la *LC* se elige un elemento $p \in \mathbb{X}$ al azar. Luego, se determinan los m elementos más cercanos a p en $\mathbb{X}$ y se los asocia a p, éste es el *cluster* de p y m es su tamaño. El número m es el único parámetro que tiene la *LC*. El proceso continúa recursivamente con los restantes elementos hasta que se obtenga una lista de $\frac{n}{m+1}$ clusters. Cada cluster almacena su radio cobertura $cr()$. Cabe recordar que *LC* usa el criterio de radio cobertor para podar las búsquedas y además aprovecha también el orden de creación de los clusters para, en algunos casos, poder descartar clusters subsiguientes. Los autores sostienen que el desempeño de *LC* en las búsquedas es imbatible sobre espacios "difíciles", al costo de tiempo de construcción de $O(n^2/m)$. *LC* requiere tanto espacio de memoria como cualquiera de las alternativas de *DiSAT*.

La clave en el desempeño de la *LC* es la selección apropiada del tamaño del cluster. Un tamaño de cluster pequeño mejora el índice en altas dimensiones. Sin embargo, un tamaño de cluster pequeño trae aparejado que el precio de construcción del índice sea demasiado costoso. Se va a comparar a *DiSAT*, que no posee parámetros en ninguna de sus alternativas, con la *LC*, reflejando cuánto tiempo extra se está dispuesto a pagar en la construcción del índice.



(a) Imágenes de la NASA.

(b) Diccionario.



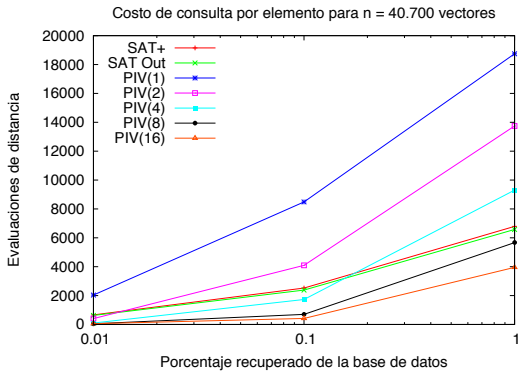
(c) Histogramas de Color.

Figura 4.22: Comparación de los costos de búsqueda entre el *DiSAT* y el *M–tree*.

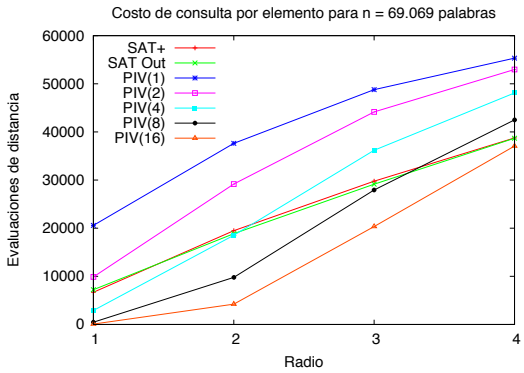
Las Figuras 4.26 y 4.27 comparan los costos de construcción y búsqueda, respectivamente, de SAT^{+}, SAT^{Out} y *LC*. Por ejemplo, en el Diccionario con un tamaño de cluster de 102 o 223 se obtiene el mismo costo de construcción de SAT^{+} o SAT^{Out}, respectivamente. Dado el costo de construcción t de SAT^{+}, esto implica tamaños m de cluster de $\frac{98}{t}$ para el espacio de Imágenes de la NASA, $\frac{102}{t}$ para el Diccionario y $\frac{105}{t}$ para el espacio de Histogramas de Color. En el caso de los costos de construcción del SAT^{Out}, éstos resultan en tamaños de cluster m de $\frac{118}{t}$ para las Imágenes de la NASA, $\frac{123}{t}$ para el Diccionario y $\frac{223}{t}$ para el espacio de Histogramas de Color.

Para el espacio de Imágenes de la NASA, SAT^{+} y SAT^{Out} superan a *LC* para todos los radios de búsqueda, aún con un tamaño de cluster $m = 25$ para *LC*, que implica aproximadamente 5 veces el tiempo de construcción de SAT^{+} y SAT^{Out}. Más aún, en esta base de datos no se logró obtener ningún tamaño que permita a *LC* ser superior a las alernativas del *DiSAT*, aún ignorando los costos de construcción. Vale la pena destacar que *LC* supera a SAT^{+} y SAT^{Out} con tamaño de $m = 100$ en todos los radios considerados sobre el Diccionario, usando 2.5 a 3 veces más tiempo de construcción. Más aún, en esta base de datos *LC* con costos de construcción similar (con $m = 200, 300$) supera a SAT^{+} y SAT^{Out} para radios amplios. Para la base de datos de Histograma de Color, de nuevo se puede ver que SAT^{+} y SAT^{Out} aventajan a *LC* para todos los radios considerados. Sin embargo, para $m = 50$ *LC* tiene costos levemente mayores de búsqueda que SAT^{+} y SAT^{Out}, usando un costo de construcción más de 6 veces mayor que estas alternativas.

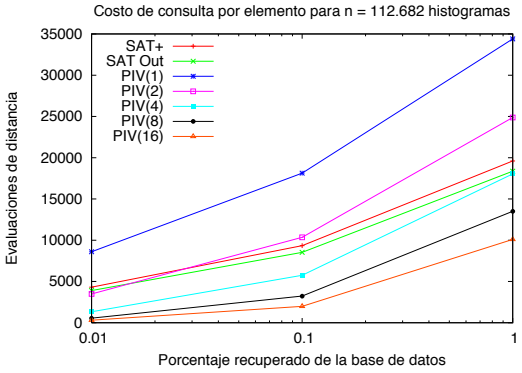
Con bases de datos masivas, un tiempo de construcción cuadrático es significativo y puede obstaculizar la adopción de un índice particular para una aplicación. Los costos de construcción de SAT^{+} y SAT^{Out} son mayores que los del *SAT* original, pero escalan mucho mejor que los de la *LC* (como se pue-



(a) Imágenes de la NASA.



(b) Diccionario.



(c) Histogramas de Color.

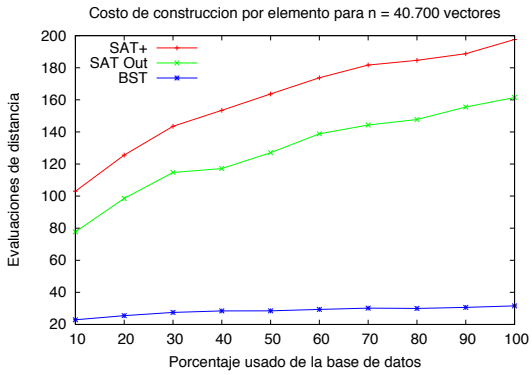
Figura 4.23: Comparación de los costos de búsqueda del *DiSAT* con un algoritmo genérico de pivotes.

de ver a continuación). Los experimentos también confirman que SAT^{+} y SAT^{Out} pueden manejar mejor que *LC* las búsquedas con radios grandes, con la característica adicional que no es necesario sintonizar ningún parámetro.

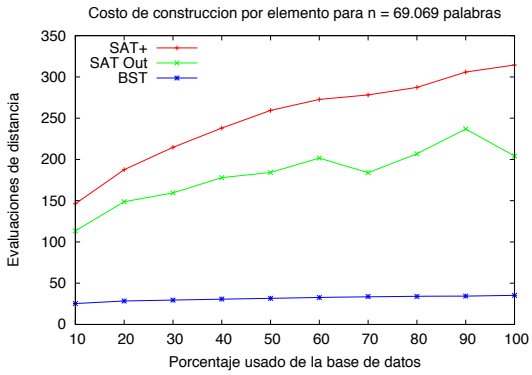
Así, en general, SAT^{+} y SAT^{Out} proveen un mejor compromiso entre eficiencia en las búsquedas y costos de construcción que la *LC*. Existen algunas instancias de base de datos donde *LC* no puede mejorar los costos de búsqueda a pesar de permitirle un tiempo de construcción arbitrario.

4.5.5. Vantage-Point Tree

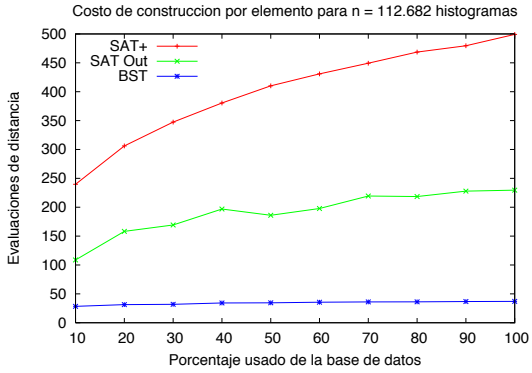
El *Vantage-Point Tree* (*VPT*) fue presentado en [Yia93], aunque la idea previamente se había introducido en [Uhl91b]. Esta estructura de datos es también un árbol, que se construye recursivamente, tomando un elemento arbitrario p como la raíz. Se calculan las distancias $d(p, u)$ de todos los elementos $u \in \mathbb{X} - \{p\}$. Se denomina M a la mediana de aquellas distancias. Todos los objetos tales que $d(p, u) \leq M$ se asignan al nodo que se conecta como hijo izquierdo de la raíz y el resto de los elementos al nodo que se conecta como hijo derecho. Luego, se continúa de manera recursiva hasta que el número de elementos sea menor que un cierto tamaño de balde o "bucket". El *VPT* necesita el mismo espacio que las alternativas del *DiSAT* y el tiempo de construcción podría ser $O(n \log n)$ en el peor caso, dado que es un árbol balanceado. Para resolver una consulta en el *VPT* se testea si la bola de consulta podría intersectar los subárboles de los nodos izquierdo y derecho. Es posible tener que entrar en ambos subárboles.



(a) Imágenes de la NASA.



(b) Diccionario.



(c) Histogramas de Color.

Figura 4.24: Comparación de costos de construcción del *DiSAT* con el *BST*.

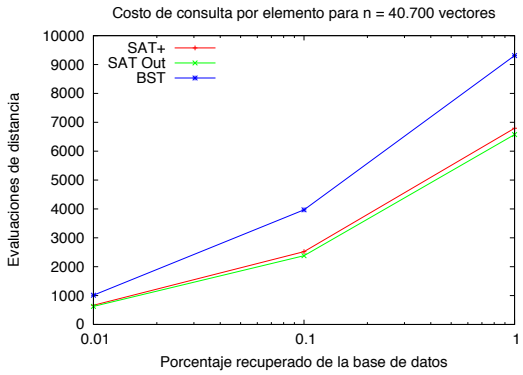
Las Figuras 4.28 y 4.29 muestra la comparación entre SAT^{+} y SAT^{Out} con el *VPT*. Se ilustra el comportamiento del *VPT* usando diferentes tamaños de bucket: *VPT(s)* indica que el *VPT* está usando un tamaño de bucket de s elementos. Se han considerado los siguientes valores de s: $10, 50, 100, 150$ y 200.

Como se puede notar, el *VPT* obtiene siempre mejores costos de construcción que SAT^{+} y SAT^{Out}, para todos los tamaños de bucket utilizados. Sin embargo, el *VPT* no puede superar el desempeño de las búsquedas de SAT^{+} y SAT^{Out}, excepto con el radio más chico de búsqueda considerado y con tamaño de bucket de 10. Se debe notar también que el desempeño del *VPT* varía con el tamaño de bucket elegido, mientras que SAT^{+} y SAT^{Out} no poseen parámetros a sintonizar.

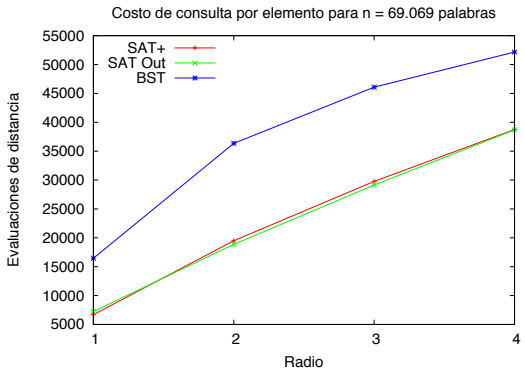
4.5.6. Geometric Near-Neighbor Access Tree

El *Generalized-Hyperplane tree* (*GHT*) [Uhl91b] se extiende en [Bri95] a un árbol m-ario, llamado *Geometric Near-Neighbor Access Tree* (*GNAT*), que mantiene la misma idea esencial. Para el primer nivel, se eligen m centros $c_1, \ldots, c_m$ y se determinan los conjuntos $\mathbb{X}_i = \{u \in \mathbb{X}, d(c_i, u) < d(c_j, u), \forall j \neq i\}$. Esto es, $\mathbb{X}_i$ contiene los elementos más cercanos a c_i que a cualquiera de los otros c_j. Desde la raíz, se construyen m hijos numerados $i = 1, \ldots m$ y recursivamente se construye un *GNAT* para cada $\mathbb{X}_i$. En la construcción el *GNAT* almacena en cada nodo una tabla:

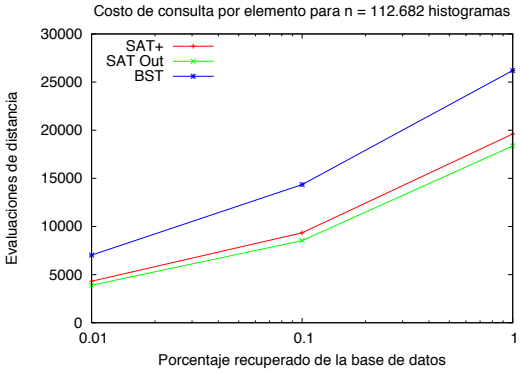
$$range_{i\,j} = [\min_{u \in \mathbb{U}_j}(c_i, u), \max_{u \in \mathbb{U}_j}(c_i, u)],$$



(a) Imágenes de la NASA.



(b) Diccionario.



(c) Histogramas de Color.

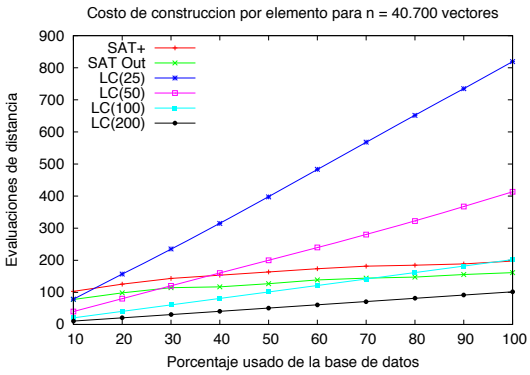
Figura 4.25: Comparación de costos de búsqueda del *DiSAT* con el *BST*.

de tamaño $O(m^2)$; es decir, la complejidad en espacio es $O(n\,m^2)$ y se construye en cerca de $O(n\,m\log_m n)$ tiempo.

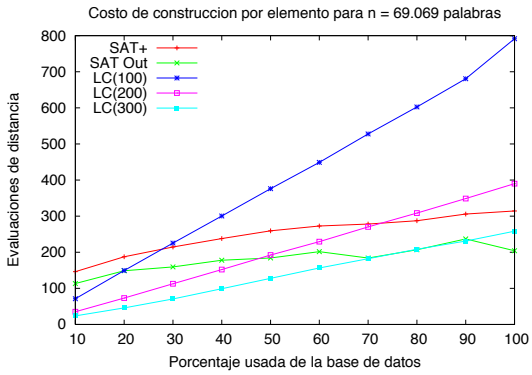
Durante las búsquedas, la consulta q se compara con algún centro c_i y luego se descarta cada otro centro c_j tal que $d(q,c_i) \pm r$ no intersecte a $range_{i,j}$. Todos los subárboles $\mathbb{X}_j$ se pueden descartar usando la desigualdad triangular [CNBYM01].

Las Figuras 4.30 y 4.31 ilustran la comparación entre *SAT*$^+$ y *SAT*^{Out} con el *GNAT*. El autor en [Bri95] sugiere que las mejores aridades para el árbol están entre 50 y 100. Por lo tanto, se ha testeado el *GNAT* con aridades 16, 32, 64 y 128. Así, en las gráficas *GNAT(a)* significa que se utiliza un *GNAT* con aridad a.

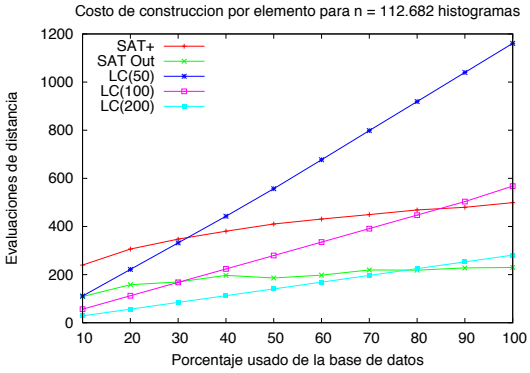
Como se puede observar el *GNAT* tiene mejores costos de construcción que *SAT*$^+$ y *SAT*^{Out}, con casi todas las aridades consideradas, excepto con aridad 128. Por otra parte, esta aridad es la mejor para el *GNAT* en las búsquedas. Particularmente, en el espacio del Diccionario (Figura 4.31(b)) *SAT*$^+$ y *SAT*^{Out} superan al *GNAT* con todas las aridades usadas. Sin embargo, *GNAT* necesita mucho más espacio que las alternativas del *DiSAT*. Como se mencionó, en un *GNAT* con aridad m almacena en cada nodo m^2 distancias, mientras que *SAT*$^+$ y *SAT*^{Out} almacenan sólo una distancia por nodo. Por ejemplo, un *GNAT* con la aridad más pequeña considerada (16), necesita 16 veces el espacio usado por *SAT*$^+$ y *SAT*^{Out}. Se puede notar que el *GNAT* podría ser más lento en las búsquedas que *SAT*$^+$ y *SAT*^{Out} si se elige una aridad no adecuada.



(a) Imágenes de la NASA.



(b) Diccionario.



(c) Histogramas de Color.

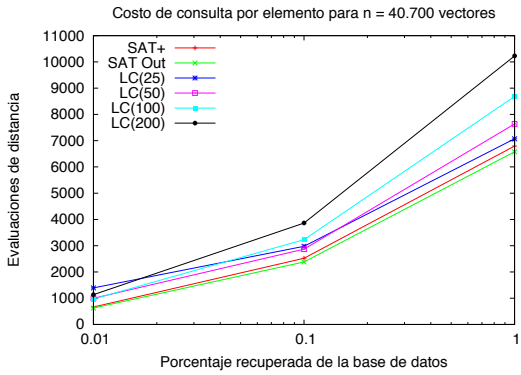
Figura 4.26: Comparación de los costos de construcción de *DiSAT* con la *LC*.

4.5.7. Aceleración

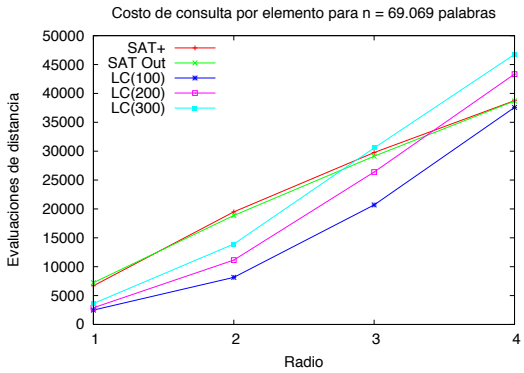
El ciclo más interno de un índice, o la complejidad interna, podría hacer una diferencia en la velocidad total en una consulta. Esto es especialmente cierto para distancias de baja complejidad. Cuando una función de distancia es compleja, el tiempo total de consulta es proporcional al número de cálculos de distancia. El tiempo total de consulta no es siempre útil cuando se compara un índice de otro autor, sin implementar el algoritmo desde cero. La razón es que el tiempo total de consulta depende de la velocidad de la máquina, el lenguaje de la implementación, el sistema operativo, la caché, etc. Con el fin de hacer comparable el tiempo total de consulta entre los sistemas, se ha normalizado por el tiempo usado por un algoritmo de fuerza bruta de recorrido secuencial; a esto se lo denomina *aceleración* o *speedup*.

Se compara el speedup de SAT^{+} y SAT^{Out} con otros índices en espacios de un millón de elementos, el de *Flickr* y sintéticos. La Figura 4.32 ilustra el speedup de los diferentes métodos a medida que crece la dimensión. Las Figuras 4.32(a), 4.32(b) y 4.32(c) muestran respectivamente los costos de búsqueda de recuperar el $0{,}01\,\%$, $0{,}1\,\%$ y $1\,\%$ de la base de datos. En todas las dimensiones se ha considerado para el *GNAT* la mejor aridad de 128 y para *LC* los tamaños de clusters de 256 para dimensión 2, 128 para dimensión 4 y 8, y 32 para dimensión 16. Como antes, para el *DSAT* se utiliza la mejor aridad máxima para cada dimensión; esto es, aridad 4 para las dimensiones 2 y 4, aridad 8 para dimensión 8 y aridad 32 para dimensión 16.

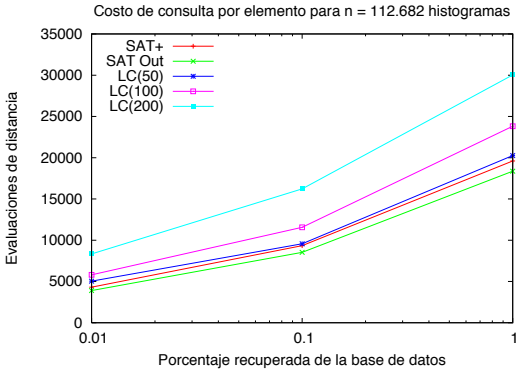
Es claro que la complejidad interna del *VPT* es menor que la de SAT^{+} y SAT^{Out}, logra un gran speedup con radios de búsqueda pequeña sobre los espacios de dimensión hasta 8. Sin embargo, a medida que el radio o la dimensión crece, *LC* supera a todos los otros y el speedup del *VPT* se vuelve similar al



(a) Imágenes de la NASA.



(b) Diccionario.



(c) Histogramas de Color.

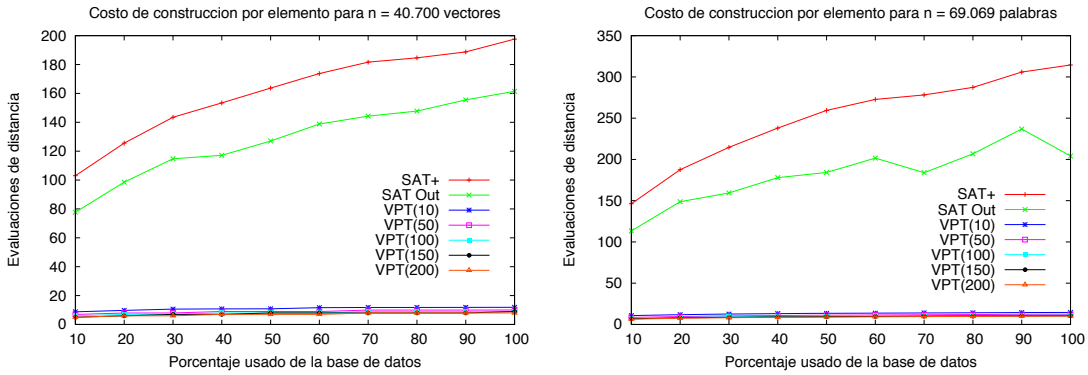
Figura 4.27: Comparación de los costos de búsqueda de *DiSAT* con la *LC*.

de SAT^{+} y SAT^{Out}. Notar que SAT^{+} y SAT^{Out} siempre están entre las cuatro mejores técnicas.

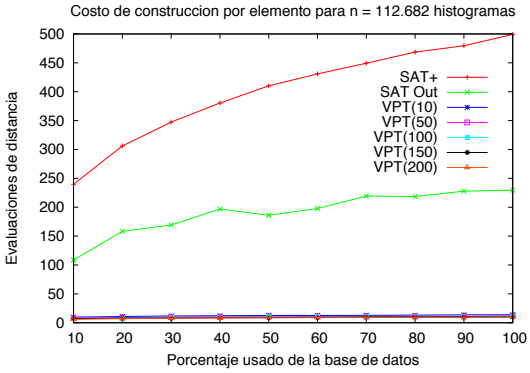
Para hacer una comparación completa y justa, la Figura 4.32(d) muestra los costos de construcción de todas las alternativas consideradas. La *LC* obtiene el peor costo de construcción para casi todas las dimensiones consideradas (notar que no se considera la sintonización del parámetro). El *SAT* tiene el peor costo de construcción en dimensión 2. Este comportamiento extraño es consistente con lo mostrado en la Figura 4.18. Los otros métodos tienen tiempos de construcción similar.

Se ha testeado el speedup con bases de datos grandes de un millón de vectores en diferentes dimensiones. La Figura 4.33 exhibe cómo se afecta el speedup (Figura 4.33(a)) y el tiempo de construcción (Figura 4.33(b)) a medida que la dimensión crece. Dado que la *LC* muestra previamente uno de los mejores speedup y su parámetro es el tamaño del cluster, se muestra su comportamiento con diferentes tamaños. Para estos experimentos se ha considerado también la *Extreme Pivot Table* (*EPT*) presentada en [RSC+13], como un método de selección de pivotes. Cada pivote en la *EPT* tiene asociados algunos elementos de la base de datos, las *regiones de pivotes*. La unión de aquellas regiones es la base de datos completa. La región asociada a un pivote está formada por los objetos que más problablemente son descartados por el pivote. La *EPT* es una secuencia de capas llamadas *grupos de pivotes* (*PG*). Con esto, cada objeto está en exactamente una región por *PG* independiente del número de pivotes. El espacio que se ahorra se puede usar para agregar más capas, mejorando el tiempo de búsqueda. En las gráficas presentadas, a la *EP* se le da más holgura que a los experimentos con pivotes, se han considerado hasta 32 grupos de pivotes; es decir, al menos 16 veces el espacio necesario para las alternativas del *DiSAT*.

Para dimensión 4 SAT^{+} obtiene el mejor speedup, pero para dimensiones 8, 12 y 16 la *LC* es más



(a) Imágenes de la NASA. (b) Diccionario.



(c) Histogramas de Color.

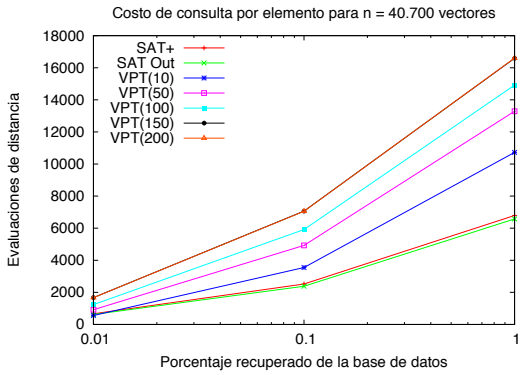
Figura 4.28: Comparación de costos de construcción del *DiSAT* con el *VPT*.

rápida con tamaño de cluster de 256, 128 y 64 respectivamente. Sin embargo, la *LC* es más que 20 veces más lenta de construir que el SAT^+. Nuevamente, es importante destacar que SAT^+ tiene la gran ventaja de no requerir la sintonización de ningún parámetro.

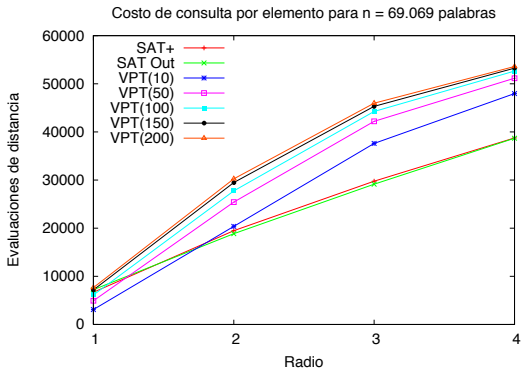
Para analizar cómo el tamaño de la base de datos afecta el speedup de los diferentes métodos en alta dimensión, se ha utilizado una base de datos de vectores en dimensión 12 con tamaños de bases de datos desde 100.000 (100m) hasta 3.000.000 (3M) de elementos. La Figura 4.34 ilustra la comparación del speedup (Figura 4.34(a)) y del tiempo de construcción (Figura 4.34(b)) para estos experimentos. Como puede notarse, la *LC* con tamaño de cluster de 64 obtiene el mejor speedup para casi todos los tamaños considerados. Para el tamaño de base de datos más grande con tamaño de cluster de 128 no es la más rápida. El tiempo de construcción de *LC* con tamaño de cluster de 64 es hasta 72 veces el tiempo necesario para construir el SAT^+ para tres millones (3M) de elementos. Más aún, el *VPT* es el otro índice competitivo, pero el speedup de las búsquedas del *VPT* apenas supera a SAT^+ y SAT^{Out}.

Finalmente. se considera la base de datos real de *Flickr*, con un millón de elementos, para comparar el speedup de cada una de las técnicas competitivas. La Figura 4.35 muestra la comparación de los costos de búsqueda para diferentes radios de búsqueda medido en cálculos de distancia (Figura 4.35(a)) y el speedup (Figura 4.35(b)). Se presentan también en la Figura 4.35(c) los tiempos de construcción.

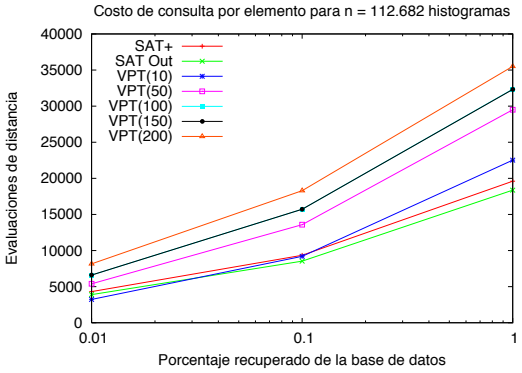
Como se puede notar, la *LC* obtiene el mejor desempeño de las búsquedas medido en cálculos de distancia, para ambos tamaños de cluster de 64 y 128. Las alternativas SAT^+ y SAT^{Out} del *DiSAT* vienen a continuación. Si se considera el speedup de la búsquedas, la *LC* de nuevo es el mejor método para todos los radios. El *VPT* obtiene el mejor speedup sólo para los radios de búsqueda más pequeños,



(a) Imágenes de la NASA.



(b) Diccionario.



(c) Histogramas de Color.

Figura 4.29: Comparación de costos de búsqueda del *DiSAT* con el *VPT*.

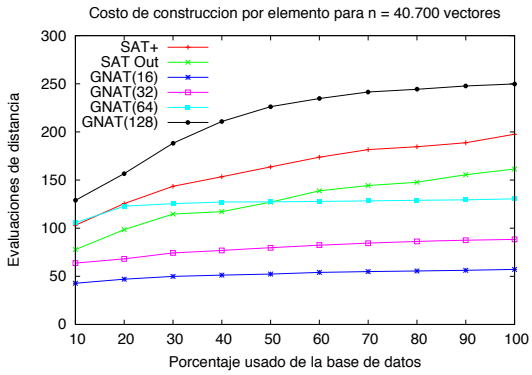
pero para los otros radios es más lento que *LC* y que SAT^{+} y SAT^{Out}. Sin embargo, si se considera el tiempo de construcción, la estructura más costosa de construir es la *LC*. Por ejemplo, la *LC* para los tamaños de clusters de 64 y 128 es cerca de 36 veces y 18 veces más lenta de construir que el SAT^{Out}, respectivamente.

Una Base de Datos Masiva

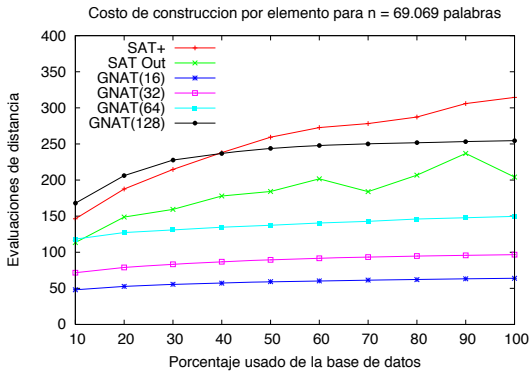
Adicionalmente, se han realizado experimentos para comparar la *LC* con SAT^{+} y SAT^{Out} sobre una base de datos masiva. Para estos experimentos, se ha utilizado un subconjunto de diez millones de imágenes desde la base de datos de *CoPhIR*. Para la *LC* se han usado clusters de tamaño 1024 y 2048. Se han construido los índices sobre tamaños crecientes de la base de datos para testear la escalabilidad a medida que crece el tamaño de la base de datos. En cada paso se duplica el tamaño. Se han reservado 200 elementos, los cuales no se han indexado, para usarlos como elementos de consulta. En todos los tamaños se ha usado el mismo radio de búsqueda $r = 200$ para consultas por rango, con $r = 200$ se recuperan en promedio más de 100 objetos. Se ha notado que la salida de la consulta ha sido vacía para los tamaños entre 100.000 y 400.000 elementos. La Figura 4.36 muestra los costos de construcción obtenidos y la Figura 4.37 exhibe los costos de búsqueda de los tres índices comparados.

Claramente, SAT^{+} y SAT^{Out} son más rápidas que la *LC* en las búsquedas, con costos de construcción órdenes de magnitud más pequeños.

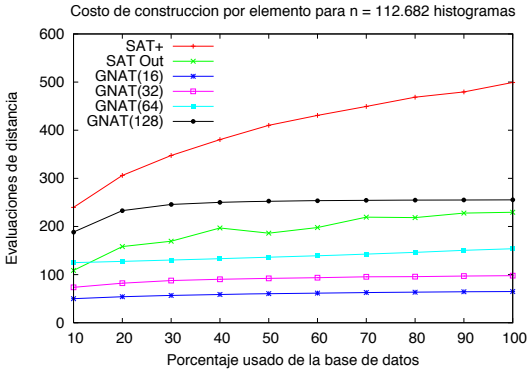
Como se puede observar, SAT^{+} y SAT^{Out} superan significativamente la *LC* en ambos tiempos de



(a) Imágenes de la NASA.



(b) Diccionario.



(c) Histogramas de Color.

Figura 4.30: Comparación de costos de construcción entre el *DiSAT* y el *GNAT*.

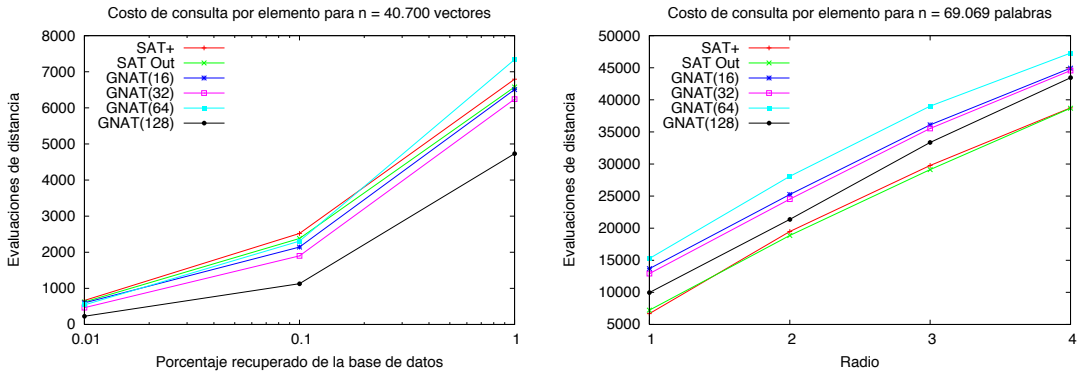
construcción y de búsqueda. Aún si los costos de construcción comienzan siendo más altos para tamaños más pequeños de la base de datos, los costos de construcción de SAT^{+} y SAT^{Out} no cambian tanto a medida que crece el tamaño de la base de datos. Para la *LC* los costos de construcción crecen muy rápidamente. Para las búsquedas la mejora es más destacable, se obtienen para SAT^{+} y SAT^{Out} mejores costos de búsqueda en todos los tamaños considerados.

4.6. Optimización a la *LC*

Se describen aquí algunas optimizaciones a la *LC* que permiten mejorar el desempeño de los algoritmos de búsqueda, que luego fueron utilizadas para una versión paralela y distribuida de la *LC*, y un algoritmo de búsqueda de k vecinos más cercanos para la *LC* [GCMR09]. El costo adicional de estas optimizaciones propuestas sobre la estructura es el espacio requerido para el índice. Estas optimizaciones se pueden aplicar conjuntamente o, en caso de tener restricciones de espacio, se puede aplicar sólo una de ellas.

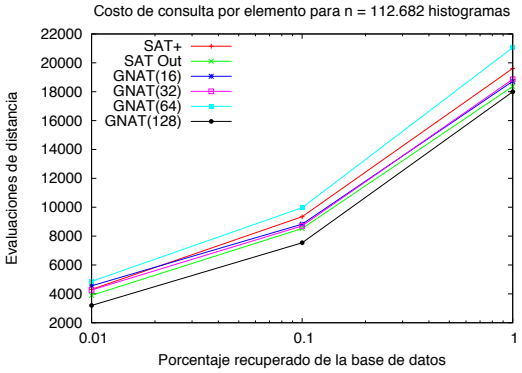
4.6.1. Distancias entre Centros

La *LC* original a pesar de ser costosa de construir por necesitar $O(n^2/m)$ para particiones de tamaño fijo m y ser $O(n^2/p^*)$ para particiones de radio fijo r^*, donde p^* es el tamaño promedio de las particiones obtenidas, no almacena casi ninguna de las distancias calculadas, excepto por el radio de cobertura del



(a) Imágenes de la NASA.

(b) Diccionario.

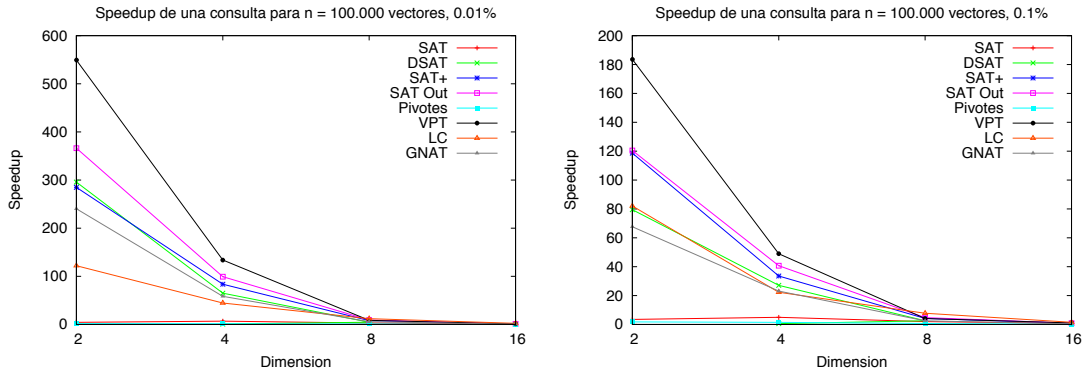


(c) Histogramas de Color.

Figura 4.31: Comparación de costos de búsqueda entre el *DiSAT* y el *GNAT*.

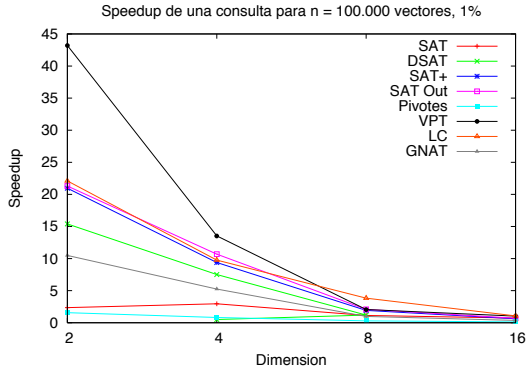
cluster. Sin embargo, si se quiere reducir el costo de las búsquedas y se dispone de memoria suficiente, se pueden almacenar algunas de las distancias calculadas durante la construcción a fin de evitar cálculos de distancia con los centros durante las búsquedas. Cuando se realiza una búsqueda por rango, a medida que se avanza en la lista de clusters, las distancias que no pueden evitar calcularse son las distancias a los centros. Estas distancias se necesitan no sólo para saber si el centro es o no un objeto relevante para la consulta sino también, junto con el radio de cobertura, para decidir si es necesario examinar su cluster y si la búsqueda debería seguir avanzando en la lista o no.

Sin embargo, si se tuviera una buena estimación entre la distancia entre el centro c y el elemento de consulta q, se podrían tomar decisiones como si realmente se hubiera calculado la distancia. Se puede lograr esta estimación manteniendo las distancias entre ciertos elementos que actúan como si fueran *pivotes*. En general, los elementos que pueden ser buenos pivotes para un elemento son los que están muy cerca o muy lejos de él [Cel08]. La heurística que usa *LC* para seleccionar los centros elige un nuevo centro como el elemento que maximiza las distancias a los centros previos [CN05], y durante la construcción se han calculado todas las distancias entre los centros. Por lo tanto, se podrían mantener estas distancias para reducir el número de evaluaciones de distancia durante las búsquedas. Debido a que la función de distancia d es reflexiva y simétrica, si e es el número de centros se necesitan $\frac{e(e-1)}{2}$ distancias. El proceso de búsqueda puede usar entonces estas distancias para estimar la distancia real entre la consulta q y un centro c. A esta estimación se la nombra como $D(q,c)$. Cuando $D(q,c) \leq r$ es posible que $d(q,c) \leq r$ y entonces se debe calcular realmente $d(q,c)$, pero en otro caso esto no es necesario. Por lo tanto, durante las búsquedas, los centros con los cuales la consulta q se ha comparado se pueden usar como pivotes para los centros restantes.

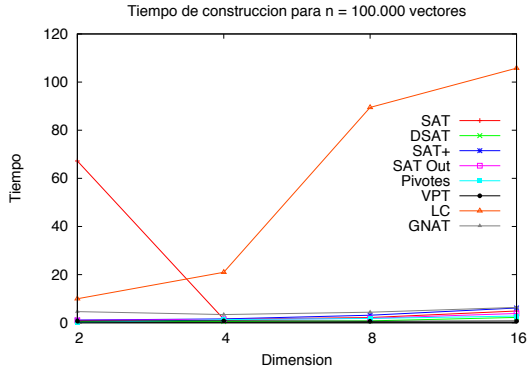


(a) Recuperando el $0{,}01\,\%$ de la base de datos.

(b) Recuperando el $0{,}1\,\%$ de la base de datos.

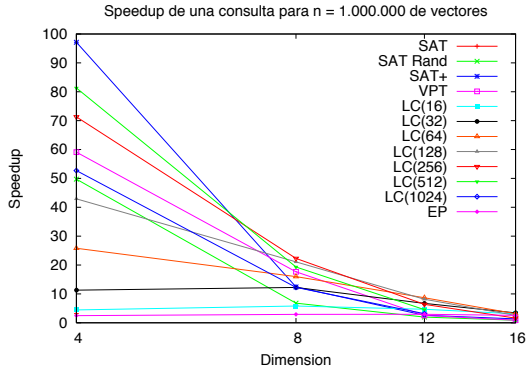


(c) Recuperando el $1\,\%$ de la base de datos.

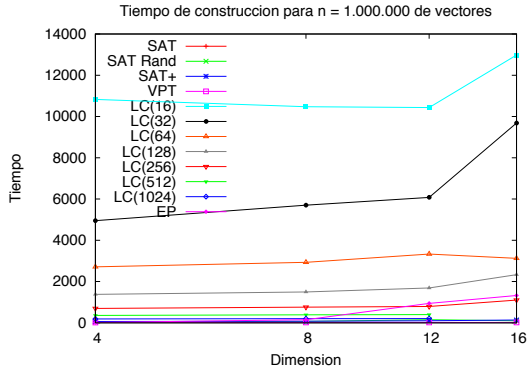


(d) Tiempo de construcción

Figura 4.32: Comparación del speedup de las consultas y tiempo de construcción, a medida que crece la dimensión.



(a) Speedup.

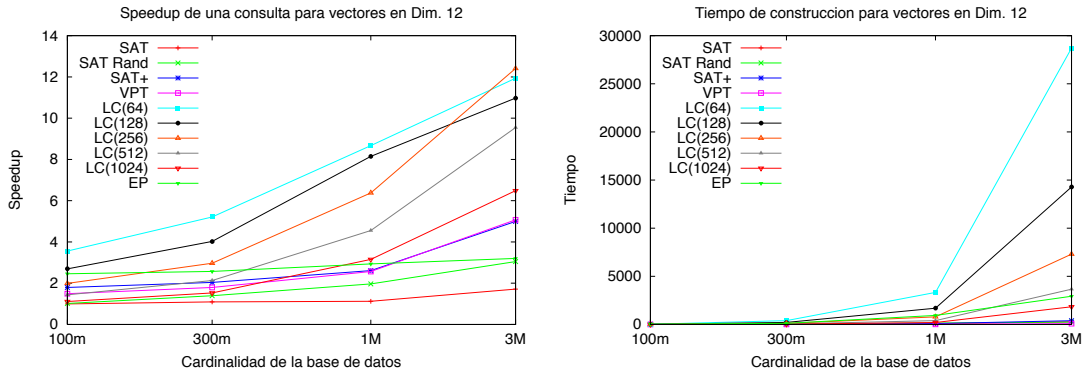


(b) Tiempo de construcción.

Figura 4.33: Comparación del speedup de las consultas y tiempo de construcción para un millón de elementos, a medida que la dimensión crece.

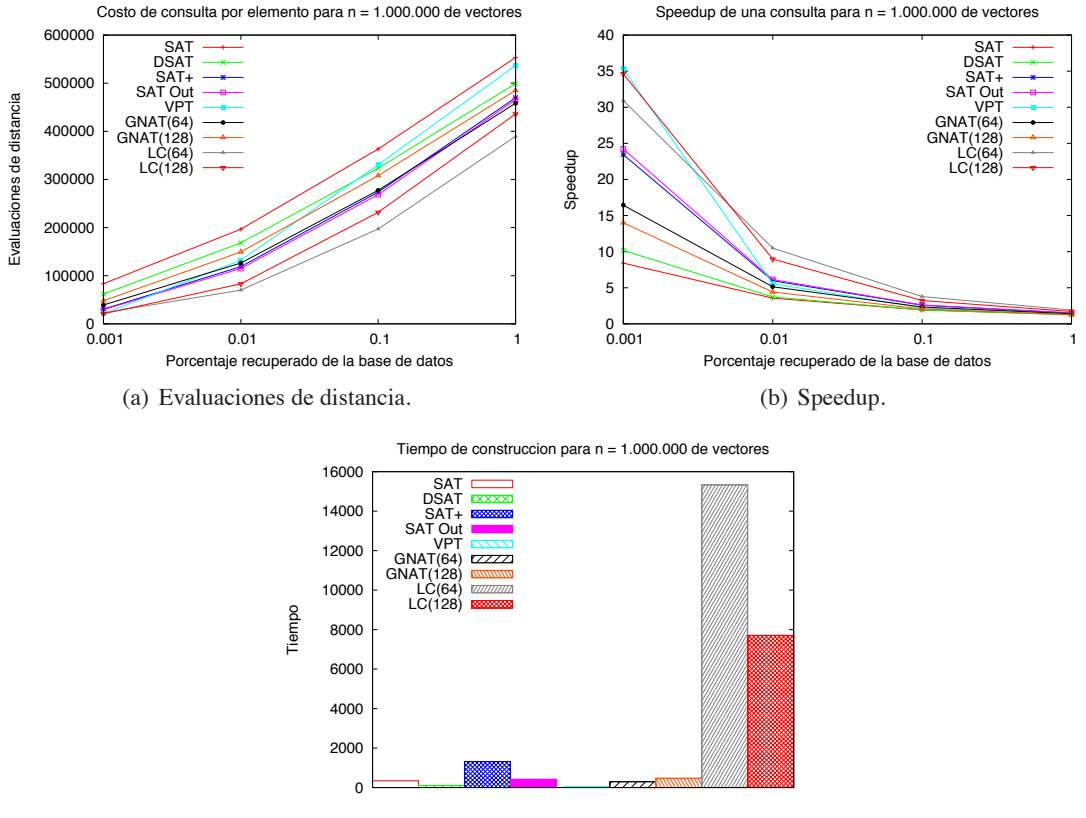
El Algoritmo 23, basado en el Algoritmo 19, ilustra el proceso de búsqueda que usa las distancias almacenadas entre los centros de la *LC*. En este caso, D_q es un conjunto de pares $(c_i, d(q, c_i))$ conteniendo la información de las distancias calculadas entre q y los centros c_i. A las distancias almacenadas entre centros se las denota como $d_{(c,c_i)}$.

La Figura 4.38 ilustra el desempeño en las búsquedas de la nueva estrategia que almacena las dis-



(a) Speedup. (b) Tiempo de construcción.

Figura 4.34: Comparación para la base de datos de vectores en dimensión 12, a medida que crece la cardinalidad de la base de datos.



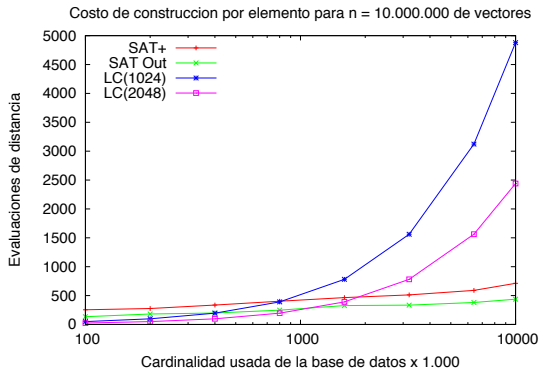
(a) Evaluaciones de distancia. (b) Speedup.

(c) Tiempo de construcción.

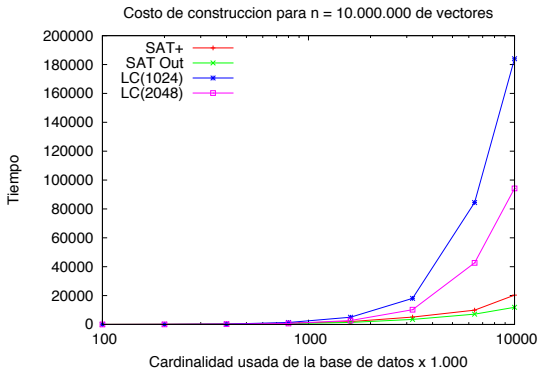
Figura 4.35: Comparación de *DiSAT* con otros índices para la base de datos *Flickr*.

tancias entre los centros, respecto de la *LC* original. Se han considerado los mejores tamaños de clusters para la *LC*. A modo de ejemplo, se muestran los resultados para dos de los espacios métricos reales. Como se puede observar, *LC-DC* puede superar significativamente a la *LC* por aprovechar las distancias almacanadas entre los centros.

El gasto adicional de espacio por almacenar la matriz de distancias entre centros se puede calcular como $O_M = \frac{S_M}{(S_I+S_M)}$, donde S_M es el tamaño de la matriz, con respecto al tamaño total del índice

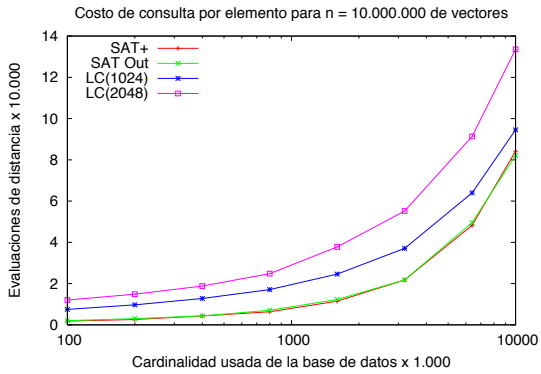


(a) Evaluaciones de distancia para construcción.

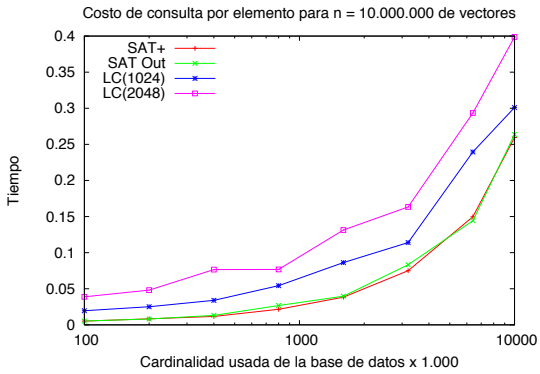


(b) Tiempo para construcción.

Figura 4.36: Comparación de costos de construcción para subconjuntos crecientes de la base de datos *CoPhIR*.



(a) Evaluaciones de distancia para búsqueda.



(b) Tiempos para búsqueda.

Figura 4.37: Comparación de costos de búsqueda en subconjuntos crecientes de la base de datos *CoPhIR*.

S_I. Este gasto depende del espacio métrico y de la cantidad de centros, lo cual a su vez depende del tamaño del cluster. Para estos espacios el gasto ha sido de $O_M \approx 0{,}7$ para el Diccionario y $O_M \approx 0{,}8$ para el espacio de imágenes de la NASA. En caso de no disponer de suficiente espacio adicional para almacenar el índice con la matriz de distancias entre centros, se pueden aplicar técnicas de compresión u organización más eficiente de los datos. Si se considera la razón $O_{IM} = \frac{(S_I+S_M)}{S_{\mathbb{X}}}$, siendo $S_{\mathbb{X}}$ el espacio utilizado para almacenar la base de datos, se ha obtenido que $O_{IM} \approx 1$ en todos los espacios, mientras que para la razón $O_I = \frac{S_I}{S_{\mathbb{X}}}$ se ha calculado que $O_I < 0{,}25$. Así, este esquema es más adecuado para objetos de base de datos que requieren una gran cantidad de memoria.

4.6.2. Objetos Candidatos

Un razonamiento similar al realizado para almacenar las distancias entre los centros, calculadas al construir la *LC*, con el fin de disminuir el número de evaluaciones de distancia en las búsquedas, se puede realizar sobre las distancias entre los centros y los objetos dentro de sus clusters. Así, luego del proceso de construcción del índice, se pueden mantener las distancias calculadas al armar los clusters. En particular, durante la construcción se ha calculado la distancia entre cada objeto y su centro de cluster y por lo tanto esa distancia se puede almacenar junto con el objeto en el cluster. El Algoritmo 24, basado en Algoritmo 19, refleja el uso de estas distancias almacenadas en cada cluster.

Algoritmo 23 Proceso de la búsqueda por rango en *LC* usando las distancias almacenadas entre centros.

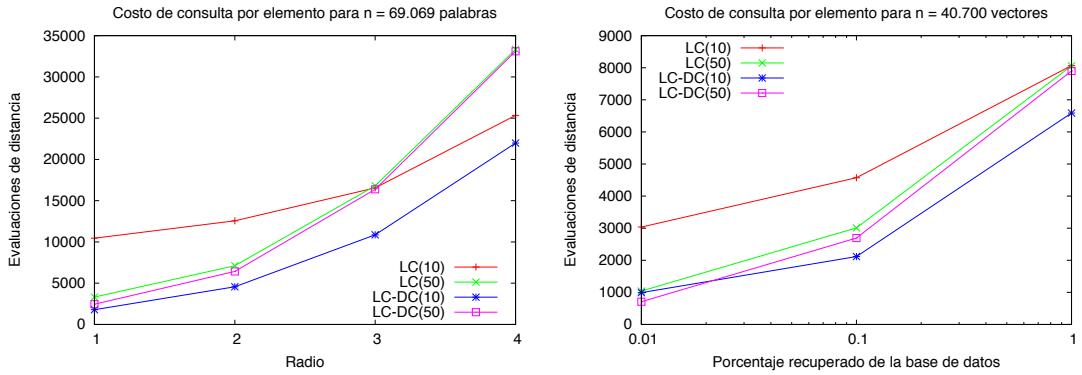
BúsquedaRangoLC-DC (Lista L, Consulta q, Radio r)

```
1. If L = Λ Then Return
2. Sea L = (c, r_c, I) : E
   /* calcular el límite inferior de d(q,c) */
3. lbound ← máx_{(c_i,d(q,c_i))∈D_q}{|d(q,c_i) − d_(c,c_i)|}
   /* calcular el límite superior de d(q,c) */
4. ubound ← mín_{(c_i,d(q,c_i))∈D_q}{d(q,c_i) + d_(c,c_i)}
5. If lbound ≤ r_c + r Then
6.     D_q ← D_q ∪ {(c, d(q,c))}     /* se debe calcular d(q,c) */
7.     If d(c,q) ≤ r Then Informar c
8.     lbound ← ubound ← d(q,c)
9. If lbound ≤ r_c + r Then    /* búsqueda exhaustiva en I */
10.    For x ∈ I Do
11.        If d(q,x) ≤ r Then Informar x
12. If ubound > r_c − r Then BúsquedaRangoLC-DC (E, q, r)
```

Algoritmo 24 Proceso de la búsqueda por rango en *LC*, usando distancias almacenadas entre el centro y los objetos de cada cluster.

BúsquedaRangoLC-D (Lista L, Consulta q, Radio r)

```
1. If L = Λ Then Return
2. Sea L = (c, r_c, I) : E
3. If d(c,q) ≤ r Then Informar c
4. If d(c,q) ≤ r_c + r Then    /* búsqueda exhaustiva en I */
5.    For x ∈ I Do
          /* verificar si debe calcularse la distancia d(q,x) */
6.        If |d(q,c) − d(c,x)| ≤ r Then    /* se debe calcular d(q,x) */
7.            If d(q,x) ≤ r Then Informar x
8. If d(c,q) > r_c − r Then BúsquedaRangoLC-D (E, q, r)
```



(a) Diccionario. (b) Imágenes de la NASA.

Figura 4.38: Comparación de costos de búsqueda entre *LC* y la versión que almacena la matriz de distancias entre centros *LC-DC*.

Además, se pueden mantener estos pares (*distancia-al-centro, objeto*) ordenados por distancia creciente al centro. Entonces, cada cluster estará compuesto por un centro y una tabla ordenada de pares (*distancia-al-centro, objeto*) y durante una búsqueda que deba examinar dicho cluster se puede usar la tabla para determinar los objetos que más probablemente sean parte de la solución y comparar sólo estos objetos con la consulta q. Por lo tanto, para una consulta dada (q, r), el rango de filas de la tabla del cluster I de c que pueden contener objetos del cluster candidatos son las que satisfacen que $d(x, c) \geq d(q, c) - r$ y $d(x, c) \leq d(q, c) + r$ para todos los objetos $x \in I$. Los límites del rango se pueden determinar eficientemente realizando dos búsquedas binarias en la tabla. Claramente, este esquema demanda espacio extra de almacenamiento, pero en sistemas de amplia escala compuesto de objetos complejos, este espacio extra puede ser relativamente insignificante. Además, se podría reducir la tabla manteniendo sólo un cierto número de filas a espacios regulares (esta idea, presentada en [GCMR09], se ha desarrollado de manera independiente con mayor detalle en [ABOPP12]). En el Algoritmo 24 se debe modificar la iteración de la línea de 5, para reflejar el uso de la tabla ordenada de pares (*distancia-al-centro, objeto*), sólo considerando los objetos $x \in I$ cuyas distancias almacenadas $d(x, c)$ cumplan con que $d(q, c) - r \leq d(q, x) \leq d(q, c) + r$; es decir, reemplazar la línea 5 por:

$$\texttt{For } x \in I \texttt{ tal que } d(q, c) - r \leq d(q, x) \leq d(q, c) + r \texttt{ Do}$$

.

4.6.3. Búsqueda de k Vecinos Más Cercanos

En [CN05] no aparece un algoritmo de búsqueda de k vecinos más cercanos, porque sistemáticamente se puede construir de una manera óptima [CNBYM01, HS00, HS03b]. Por lo tanto, siguiendo dichas técnicas, se puede adaptar el algoritmo de búsqueda por rango para obtener un algoritmo de búsqueda de k vecinos más cercanos [GCMR09]. Para ello, se puede considerar que el radio de la búsqueda por rango se inicializa en ∞ y a medida que se van calculando nuevas distancias a algunos elementos, el radio de búsqueda se actualiza con el valor de la distancia al k-ésimo elemento más cercano visto hasta ese momento (la distancia al más lejano de los k más cercanos vistos). Es decir, se puede simular la búsqueda de los k vecinos más cercanos realizando una búsqueda por rango de radio decreciente (Sección 2.6.2).

Si se conociera un radio inicial r para las consultas kNN, se puede mejorar el desempeño del algoritmo al considerar que se quieren obtener los k vecinos más cercanos de la consulta q, pero que ellos se encuentran a distancia a lo más r. En este caso, se inicializa el radio en r en lugar de en ∞. Una

mejora adicional es no recorrer la *LC* en el orden en que fue construida, sino comenzar la búsqueda por aquellos clusters que estén más cerca de la consulta q. Se puede determinar esto comparando la consulta con cada uno de los centros y usando sus radios de cobertura. Con estas distancias se obtienen cotas inferiores de las distancias entre q y cada elemento dentro de los clusters. El orden de recorrido de los clusters debería ser en orden creciente de dicho límite inferior y si ocurre un empate, los clusters con el mismo límite inferior se deberían ordenar en orden creciente de la distancia de sus centros a la consulta. Sin embargo, si se compara q con los centros de cada uno de los clusters, se pueden usar también esas distancias para reducir el radio de la consulta antes de atravesar los clusters en orden creciente del límite inferior de la distancia a la consulta.

El Algoritmo 25 muestra esta propuesta para el proceso de búsqueda de los k vecinos más cercanos de q en una LC. En la línea 7 se calcula la distancia $d(q, c)$, por lo tanto en las líneas 8 y 9 ya es conocida y no vuelve a calcularse. Se usan dos colas de prioridad adicionales. A es una cola de prioridad que almacena pares $(x, d(x, q))$ en orden creciente de distancias y se usa para mantener los k elementos más cercanos vistos hasta ese momento, y finalmente contendrá los k vecinos más cercanos de q que se obtendrán como respuesta. La otra es una cola de prioridad auxiliar Q que almacena triuplas de la forma $(I_c, lbound, d(q, c))$, ordenadas por orden creciente de $lbound$, donde $lbound$ es el límite inferior de la distancia entre q y cada elemento del cluster I con centro c. Aunque el Algoritmo 25 se ha desarrollado de manera independiente, se puede considerar que sería una aplicación de la propuesta presentada en [BN04].

Algoritmo 25 Proceso de búsqueda de k vecinos más cercanos en *LC*.

BúsquedaNNLC(Lista L, Consulta q, Vecinos requeridos k)

```
 1. create(Q), create(A)
 2. If L = Λ Then Informar la respuesta A
 3. r ← ∞
 4. For (c, r_c, I) ∈ L Do
 5.     insert(A, (c, d(q, c)))    /* se calcula la distancia d(q, c) */
 6.     If size(A) > k Then extractMax(A)    /* elemento de A con maxd más grande */
 7.     If size(A) = k Then r ← máx(A)    /* valor de maxd más grande en A */
 8.     For (c, r_c, I) ∈ L Do
            /* se verifica si I puede contener elementos relevantes */
 9.         lbound ← máx{(d(q, c) − r_c), 0}
10.         If lbound <= r Then insert(Q, (I, lbound, d(q, c)))    /* clusters prometedores */
11. While size(Q) > 0 Do
12.     (I, lbound, d_c) ← extractMin(Q)    /* elemento de Q con lbound más pequeño */
13.     If lbound > r Then Break    /* criterio global de parada */
14.     For x ∈ I Do    /* búsqueda exhaustiva en I */
15.         insert(A, (x, d(q, x)))    /* se calcula la distancia d(q, x) */
16.         If size(A) > k Then extractMax(A)
17.         If size(A) = k Then r ← máx(A)
18.  Informar la respuesta A
```

Si en cada cluster se mantienen las distancias entre los objetos del cluster x y su centro c, como se ha descripto anteriormente, entre las líneas 14 y 15 del Algoritmo 25 se debería considerar si es posible evitar el cálculo de la distancia $d(q, x)$, calculando su límite inferior como $lbound \leftarrow |d_c - d(c, x)|$ y sólo si $lbound \leq r$ realizar lo que indican las líneas 15, 16 y 17.

Notar que si se atraviesa la *LC* en el orden de construcción, se podría detener el proceso de búsqueda cuando se encuentre un cluster que contenga estrictamente la bola de consulta con centro q y radio actual

r. Para mantener el código del Algoritmo 25 simple, no se incluye en él esta mejora. Se puede usar esta restricción para evitar visitar los clusters restantes que contengan la bola de consulta.

Además, si k es mucho menor que el tamaño de los clusters y si no se está realmente interesado en los k vecinos más cercanos exactos de q, sino que bastaría con tener k elementos suficientemente cerca de q, alcanza con examinar el cluster cuyo centro sea el más cercano a la consulta y luego visitar los clusters previos en el orden de construcción que tengan intersección no vacía con la bola de consulta (considerando como radio r de la consulta a la distancia al k-ésimo elemento más alejado entre los k elementos más cercanos vistos hasta ese momento). Esta heurística es posible, debido a la asimetría de la estructura de datos *LC*.

Capítulo 5

Índices Dinámicos para Memoria Secundaria

Como se ha mencionado, existen numerosos índices métricos que permiten acelerar las búsquedas por similitud. La mayoría de ellos, sin embargo, son *estáticos* y trabajan en memoria principal. Los índices estáticos se deben reconstruir desde cero cuando el conjunto de elementos indexados se somete a inserciones y eliminaciones. Por otra parte, los índices diseñados para funcionar en memoria principal sólo pueden indexar conjuntos pequeños de datos, sufriendo de una seria degradación en su desempeño cuando los objetos están almacenados en disco. La mayoría de las aplicaciones de la vida real requieren de índices capaces de trabajar en disco y de soportar inserciones y eliminaciones de objetos intercaladas con las consultas.

Los pocos índices para espacios métricos que soportan dinamismo y diseñados para memoria secundaria se pueden también clasificar en aquéllos que usan pivotes [FTJF01, JOT+05, RSC+13], aquéllos que usan particiones compactas [CPZ97, JTSF00, NU11] y aquéllos que usan combinación de ambos [DGSZ03a, SPS04, JKF13].

En los espacios métricos de baja dimensión se pueden aplicar eficientemente soluciones basadas en pivotes. Sin embargo, las aproximaciones de pivotes fallan en espacios de alta dimensión, debido a que estos espacios son más difíciles de indexar. En esta etapa del trabajo, el interés está puesto en índices dinámicos para memoria secundaria en espacios de dimensión media a alta. El índice más famoso de esta familia es el *M–tree* [CPZ97]. Luego han surgido varios otros, algunos como variantes de índices existentes y otros nuevos, por ejemplo el *Slim–tree* [JTSF00], el *D–index* [DGSZ03a], el *PM–tree* [SPS04], el *EGNAT* [NU11] y el *MX–tree* [JKF13].

En este capítulo se proponen tres nuevos índices dinámicos, basados en particiones compactas, especialmente diseñados para memoria secundaria. El primero de ellos sólo soporta inserciones y búsquedas, lo cual es aceptable en algunos escenarios, mientras que los dos restantes son completamente dinámicos.

5.1. Conceptos Previos

Como se ha mencionado previamente, la distancia se supone costosa de calcular. Sin embargo, cuando se trabaja en memoria secundaria, la complejidad de la búsqueda debe también considerar el tiempo de E/S; pudiendo ignorar otros componentes tales como tiempo de CPU para cálculos extras.

Se describen brevemente aquí los conceptos básicos necesarios para poner en contexto el diseño para memoria secundaria de los índices, para mayores detalles se puede considerar la siguiente bibliografía:

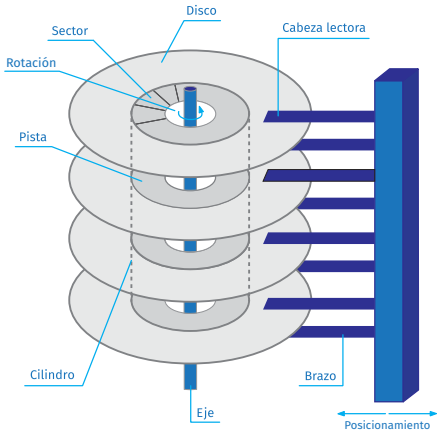


Figura 5.1: Estructura básica de un disco.

[Vit08, FRZ97, EN10].

El disco es el almacenamiento de memoria secundaria más usado, porque ofrece alta capacidad a un bajo costo. Por ser un dispositivo físico, con partes mecánicas, los accesos a disco siempre son más costosos que los accesos a memoria. La información sobre un disco se almacena sobre la superficie de uno o más *platos*. La disposición es tal que la información se almacena sobre *pistas* sucesivas sobre la superficie del disco. Cada pista a su vez se divide en *sectores*, que es la porción direccionable más pequeña en disco. Como los discos a menudo cuentan con varios platos, las pistas que se pueden acceder directamente por las cabezas de lectura/escritura se denominan un *cilindro*. La importancia del cilindro es que toda la información que contiene un único cilindro se puede acceder sin mover el brazo que mantiene las cabezas de lectura/escritura. El movimiento de este brazo se denomina *posicionamiento*. El movimiento del brazo es habitualmente la parte más lenta de leer la información desde el disco. Una *página de disco* corresponde a un *sector* de un cilindro; es decir, un sector de una pista en todos los platos. La Figura 5.1 ilustra la estructura básica de un disco [1].

Los factores que contribuyen al tiempo total T_{acc} necesario para acceder a una archivo almacenado en disco, desde el punto de vista físico, son: tiempo de posicionamiento (T_{pos}), tiempo demora rotacional (T_{rot}) y tiempo de transferencia (T_{transf}).

$$T_{acc} = T_{pos} + T_{rot} + T_{transf}$$

El tiempo de posicionamiento es el tiempo necesario para mover el brazo de acceso al cilindro correcto y depende de cuán lejos se deba mover. Si se accede secuencialmente a un archivo y el archivo está almacenado sobre varios cilindros consecutivos, el brazo sólo se moverá al terminar de procesar todo el cilindro y luego se necesita mover el brazo sólo en el ancho de una pista. Por otra parte, si se acceden dos posiciones al azar de un archivo, el brazo debe moverse desde el cilindro del primer acceso al cilindro donde se encuentra la segunda posición requerida, por lo cual el movimiento del brazo puede ser mayor que en el caso secuencial.

El tiempo de demora rotacional considera el tiempo de rotación que necesita el disco para hacer que el sector requerido quede bajo la cabeza de lectura/escritura, en promedio se considera la mitad del tiempo que necesita un disco para una revolución completa. Como en el caso del posicionamiento, este promedio se considera en accesos al azar; sin embargo, si se realiza acceso secuencial es posible que sólo este tiempo afecte al primer acceso y los demás, dentro del mismo cilindro, no deban considerarlo.

[1] Agradezco a mi hija, la diseñadora Nora Julia Aguirre-Reyes, por la realización de esta imagen.

El tiempo de transferencia corresponde al tiempo necesario para transferir los datos desde el disco, cuando la cabeza de lectura/escritura ya se encuentra ubicada en el cilindro y sector adecuados. Como el tiempo de una rotación del disco es fijo, si se desea transferir un sector, se calcula como el cociente entre tiempo de una rotación y la cantidad de sectores por pista.

Como se ha mencionado, el tiempo de posicionamiento es el que más influye en los tiempos de acceso a disco. Si un archivo ocupa N cilindros en disco, el tiempo promedio necesario entre dos accesos aleatorios es aproximadamente equivalente al tiempo de mover las cabezas de lectura/escritura $\frac{N}{3}$ cilindros. Por otra parte, si se accede secuencialmente al archivo, o se realizan accesos no aleatorios sino ordenados, el tiempo de posicionamiento a considerar se convierte en poco significativo.

En lo que sigue se considerará como medida de complejidad la cantidad de operaciones de E/S, cantidad de páginas de disco leídas o escritas, y no los tiempos, porque esta medida es independiente de la tecnología propia del disco utilizado y sirve para predecir los tiempos de acceso a disco del índice.

Por lo tanto, el tiempo de E/S está compuesto del número de páginas de disco de tamaño B que se leen y escriben. Dada una base de datos de $|\mathbb{X}| = n$ elementos de tamaño total N y considerando que las páginas de disco son de tamaño B, las consultas trivialmente se pueden responder realizando n cálculos de distancia y $\frac{N}{B}$ operaciones de E/S. El objetivo del índice en este caso es preprocesar la base de datos para responder las consultas con tan pocos cálculos de distancia y operaciones de E/S como sea posible.

En términos de uso de memoria, se considera la memoria extra requerida por el índice por sobre los datos y en el caso de memoria secundaria, la utilización de páginas de disco; es decir, la fracción de las páginas de disco que se utilizan efectivamente, en promedio.

En un escenario dinámico, el conjunto $\mathbb{X}$ puede sufrir inserciones y eliminaciones, y el índice se debe actualizar de acuerdo a ellas para las posteriores consultas. También es posible comenzar con un índice vacío y construirlo por sucesivas inserciones.

En algunos escenarios no ocurren eliminaciones. En otros, ellas son suficientemente escasas para permitir un enfoque simple que permita soportarlas: se marcan los objetos eliminados y se los excluye del resultado de las consultas. El índice se reconstruye cuando la proporción de elementos eliminados supera un umbral. Por el contrario, el caso más desafiante es donde ocurren frecuentemente eliminaciones y entonces se deben realizar físicamente, o cuando los objetos son tan grandes que resulta inaceptable retenerlos sólo por el propósito de la indexación.

5.2. Configuración de los Experimentos

Como en la Capítulo 4, para la evaluación de los índices primero se consideran los tres espacios métricos, obtenidos desde la Biblioteca Métrica de SISAP [FNC07], que fueron descriptos en la Sección 4.3. Estos espacios no tienen muchos elementos, pero son útiles para sintonizar los índices y tomar algunas decisiones de diseño basados en su desempeño aproximado. Más adelante, en la Sección 5.7, se consideran espacios más grandes o masivos.

Para la evaluación experimental de las distintas variantes de una misma estructura se utilizan los espacios métricos reales descriptos en la Sección 2.10, descargadas desde SISAP [FNC07]. Como se ha mencionado, para evaluar el comportamiento de los distintos índices los experimentos se realizan de acuerdo a lo descripto en la Sección 2.11.

El tamaño de la página de disco es $B = 4$KB en la mayoría de los casos; también se ha considerado en algunos experimentos un tamaño de 8KB. Todas las estructuras de datos de árbol mantienen en

memoria principal una copia de la raíz. Todos los índices se han construido por inserciones sucesivas.

Como base para las comparaciones, se ha utilizado el *M–tree* [CPZ97], por ser la estructura de datos dinámica y para memoria secundaria más famosa y porque además sus códigos están disponibles [2]. Se ha usado la sintonización de los parámetros del *M–tree* como sugieren los autores [3]. También se comparan las alternativas propuestas con el *eGNAT* [NU11] y con el *MX-tree* [JKF13].

5.3. Árbol de Aproximación Espacial Dinámico en Memoria Secundaria

Se propusieron en [NR09] variantes del *DSAT* para memoria secundaria, las cuales mantienen exactamente la misma estructura y realizan las mismas evaluaciones de distancia que la versión del *DSAT* para memoria principal descripta en la Sección 3.5.1. El desafío es cómo mantener un diseño en disco que minimice la cantidad de operaciones de E/S. Se describe primero la variante *DSAT** y a continuación la variante denominada *DSAT+*. Como se ha mencionado, no se han encontrado sobre estas variantes maneras eficientes de realizar eliminaciones. Todos los métodos de eliminación para el *DSAT*, descriptos en la Sección 3.5.2, requerirían numerosos accesos de lectura y escritura en disco, lo que no sería aceptable en este contexto. Por lo tanto, para esta estructura se han considerado sólo inserciones y búsquedas. En escenarios donde las eliminaciones no son frecuentes, se pueden soportar marcando los elementos eliminados de manera tal de omitirlos en los conjuntos resultados de las consultas y realizando periódicamente la reconstrucción de la estructura a fin de remover los elementos eliminados.

En ambas variantes se fuerza a que, para cualquier a, el conjunto de $N(a)$ se empaquete junto en una página de disco, lo cual asegure que el recorrido de $N(a)$ requiera sólo una lectura a disco. Más aún, por razones técnicas que se volverán claras más adelante, $MaxArity$ debe ser tal que una página de disco pueda mantener al menos dos vecindades $N()$ completas; es decir, dos listas de vecinos de máxima longitud. Aún se es libre de usar un valor considerablemente más bajo para $MaxArity$, lo cual puede ser beneficioso para el desempeño.

Para evitar la subutilización del disco, se permite que varios nodos compartan una única página de disco. Se definen políticas de inserción que permiten un particionado del árbol en páginas de disco que es eficiente para las búsquedas y no gasta demasiado espacio. Se garantizará una utilización mínima promedio de la página del disco de 50 %, y se logrará mucho más en la práctica.

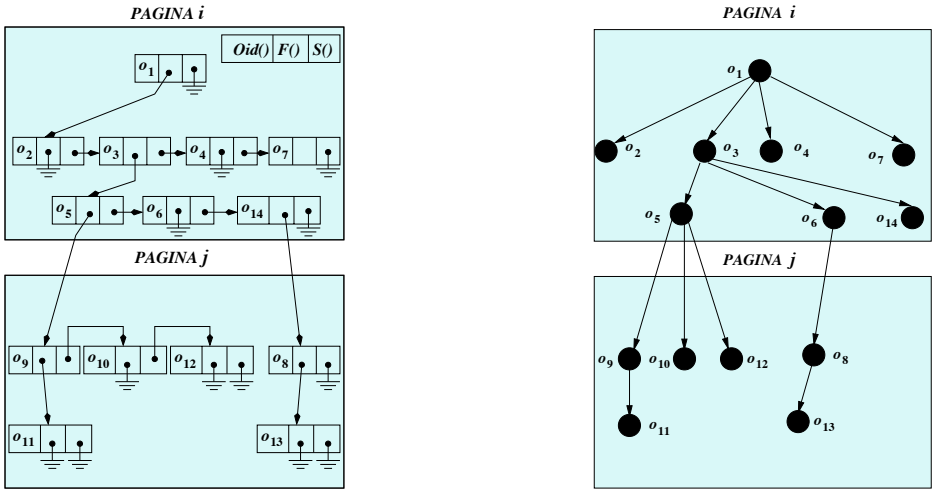
5.3.1. Diseño de la Estructura de Datos

Se representan los hijos de un nodo como una lista vinculada. Por lo tanto, cada nodo del árbol tiene un puntero al primer hijo $F(a)$ y uno al próximo hermano $S(a)$, donde el último siempre es local a la página de disco. Esto permite realizar más cambios a $N(a)$ sin acceder a a, el cual podría estar en otra página. Cada página en disco mantiene el número de nodos realmente utilizados. Punteros lejanos (o no locales) como $F(a)$ (es decir, punteros potencialmente a otra página) tienen dos partes: la página de disco y la posición del nodo dentro de esa página. La Figura 5.2 ilustra los punteros $F(a)$ y $S(a)$, mientras que se han omitido el timestamp $T(a)$ y el radio de cobertura $R(a)$.

En esta implementación los nodos tienen tamaño fijo, así objetos de largo variable como las cadenas de caracteres se rellenan a su máxima longitud. Es posible, con más esfuerzo de programación, almacenar tamaños variables para cada nodo en una página de disco. En este caso, la garantía de mantener al menos dos listas de vecindades $N()$ por página se traduce en un límite variable sobre la aridad de

[2] Los códigos fuentes del *M–tree* se encuentran disponibles en `http://www-db.deis.unibo.it/research/Mtree`

[3] `SPLIT_FUNCTION = G_HYPERPL, PROMOTE_PART_FUNCTION = MIN_RAD, SECONDARY_PART_FUNCTION = MIN_RAD, RADIUS_FUNCTION = LB, MIN_UTIL = 0.2.`



(a) Implementación. (b) Árbol en disco.

Figura 5.2: Ejemplo del diseño de los punteros $F(a)$ y $S(a)$ en un nodo.

cada nodo, de manera tal que a un nodo a no se le permite adquirir un nuevo vecino si el tamaño total de su lista de $N(a)$ podría superar la mitad del tamaño de la página de disco. En el caso, sin embargo, de objetos grandes que podrían forzar aridades muy bajas (o que simplemente no caben en la mitad de una página de disco), se pueden usar punteros a otra área de disco, como es usual en otras estructuras métricas, y tratar los punteros como objetos. En este caso, cada cálculo de distancia implica al menos un acceso a disco.

5.3.2. Inserciones

Para insertar un nuevo elemento x en el *DSAT** primero se procede exactamente como en el Algoritmo 10: se encuentra el punto de inserción en el árbol siguiendo un único camino, así cuando se determine que x debería agregarse a $N(a)$, se tienen ambas la página de a y la de $N(a)$ cargadas en memoria principal (estas páginas pueden ser la misma o diferentes). Luego se agrega a x al final de la lista de $N(a)$ en una página de disco. Si $N(a)$ fuera vacío, entonces x se convertirá en el primer hijo de a, así se modifica $F(a)$ y se inserta a x en la página de a. En otro caso, se debe agregar a x al final de la lista de $N(a)$, en la página de $N(a)$.

En cualquier caso, se debe agregar a x a una página existente y es posible que no haya suficiente espacio en la página. Cuando es éste el caso, una página debe dividirse en dos, o algunas partes de la página se deben insertar en una página existente. Se describe a continuación la política de manejo de *rebalse* u *overflow*.

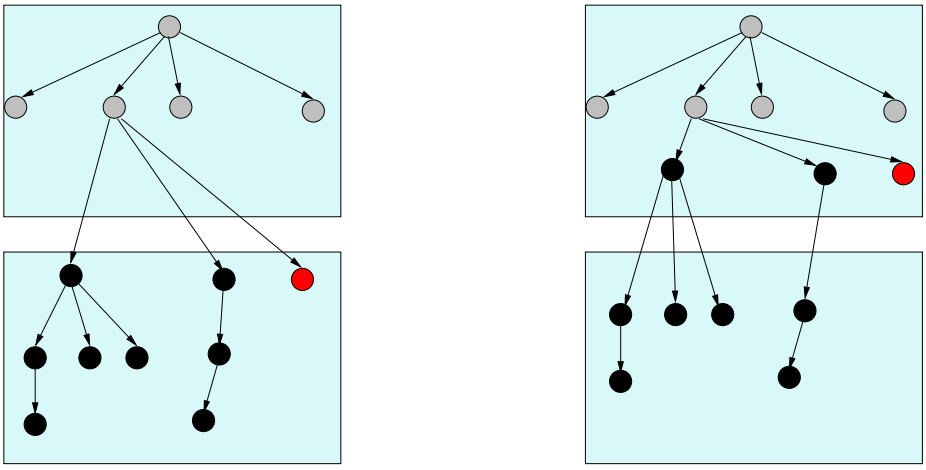
Debido a que cada $N(a)$ cabe en una única página de disco, el costo de E/S de una inserción es a lo sumo h lecturas de páginas más de 1 a 3 escrituras de página, donde h es la profundidad final de x en el árbol. Las lecturas pueden ser mucho menores que h porque a y $N(a)$ pueden estar en la misma página para muchos nodos a lo largo del camino.

5.3.3. Manejo de Rebalse de Página

Cuando la inserción de x en $N(a)$ produce un rebalse de página, se intentan las siguientes estrategias, en orden, hasta que alguna tenga éxito en resolver el rebalse. En lo que sigue, se supone que x ya se ha

insertado en $N(a)$, y $N(a)$ no cabe en su página de disco.

Primero (traslado al padre) Si a y $N(a)$ están almacenados en páginas distintas, y hay suficiente espacio libre en la página de a para mantener la vecindad $N(a)$ completa, entonces se mueve $N(a)$ a la página de a y se termina. Esto realmente mejora los tiempos de acceso de E/S para $N(a)$. Se realizan dos escrituras de páginas en este caso. La Figura 5.3 ilustra la situación.



(a) Antes. (b) Después.

Figura 5.3: Ejemplo antes y después de aplicar la política de ***traslado al padre***.

Segundo (División vertical) Si la página de $N(a)$ contiene subárboles con diferentes padres desde otra página, se hace lugar moviendo el subárbol completo donde ocurrió la inserción a una nueva página (es decir, se mueven los nodos que residían en esa página, a una nueva). Esto mantiene el número de lecturas a disco necesarias al atravesar el subárbol. Son necesarias hasta 3 escrituras de páginas, porque el puntero $F()$ del padre del subárbol está almacenado en otra página, y debe ser actualizado. Cabe destacar que se conoce dónde está el padre del subárbol a trasladar, porque se ha tenido que descender hasta $N(a)$. La Figura 5.4 ilustra este caso.

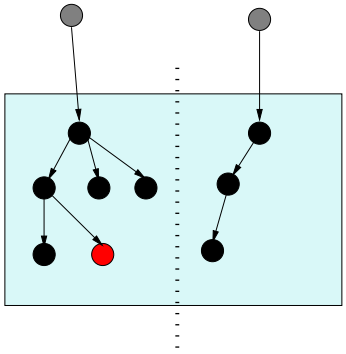
Para mantener una propiedad que se volverá evidente en la Sección 5.3.4, se evita usar la división vertical siempre que, luego de mover el subárbol elegido a una nueva página, quede menos de la mitad llena.

Tercero (División horizontal) Se mueven a la nueva página todos los nodos del subárbol al que se llegó, con profundidad local más grande que d, para el d más chico que deje al menos con la mitad llena a la página actual. La profundidad local es la profundidad dentro del subárbol almacenado en la página, y se puede calcular al vuelo en el momento de la división. Notar que:

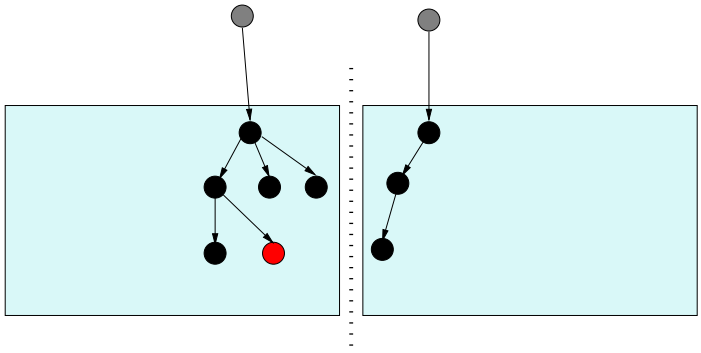
(i) los nodos cuyo padre está en otra página tienen el d más pequeño y así ellos no se mueven (en otro caso se podría mover el subárbol completo, lo cual es equivalente a la división vertical), de aquí basta con sólo dos escrituras de página;

(ii) ningún $N(b)$ se divide en ese proceso porque todos tienen el mismo d;

(iii) la página nueva contiene los hijos de los diferentes nodos y potencialmente de páginas diferentes después de futuras divisiones del nodo corriente.



(a) Antes.



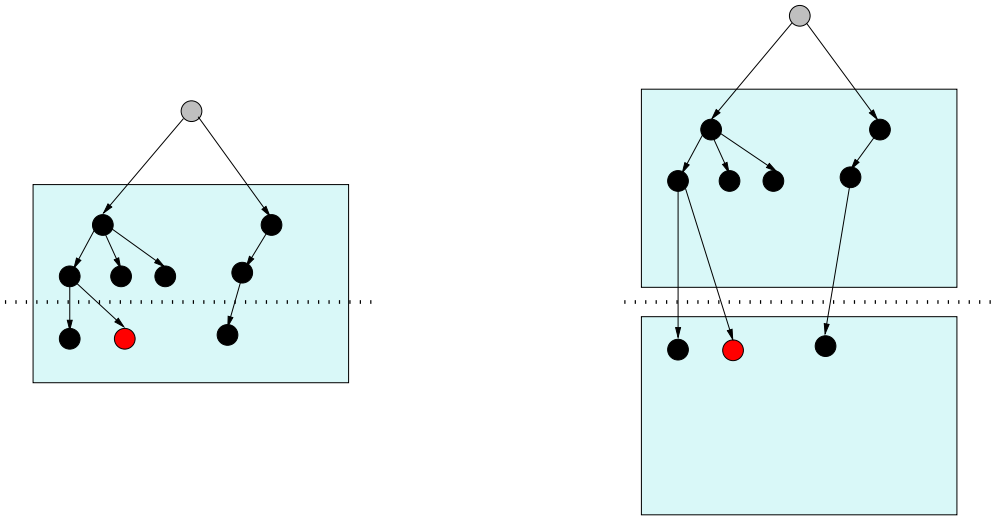
(b) Después.

Figura 5.4: Ejemplo antes y después de aplicar la política de ***división vertical***.

Se tiene que refinar la regla cuando aún el d más grande deja la página actual con menos que la mitad llena. En este caso se mueven sólo algunas de las listas $N(b)$ de profundidad d. Esto podría dejar todavía la página actual con menos de la mitad llena si existe sólo una $N(b)$ de profundidad máxima d, pero esto no puede ocurrir debido a que la capacidad de la página es al menos el doble de la longitud máxima de una lista $N()$. Otro problema potencial es que, si la máxima profundidad $d = 0$, entonces se moverá una lista $N(b)$ cuyo padre esté en otra página. Sin embargo, esto también es imposible debido a que el subárbol debería estar formado sólo por la lista de $N()$ de nivel alto, y dado que no puede ocupar más que la mitad de la página, en caso de rebalse se debería haber aplicado una división vertical.

La Figura 5.5 ilustra la aplicación de una división horizontal.

Notar que el *traslado al padre* se puede aplicar debido a que una *división vertical* u *horizontal* libera algún espacio en a dado que sus hijos se tienen que mover a una nueva página. Una *división vertical* se puede aplicar después de una *división horizontal* que pone varios nodos de diferentes padres juntos. Una *división horizontal* es siempre aplicable, debido a que cualquier $N(a)$ individual cabe en una página. Finalmente, se trata en el comienzo de poner $N(a)$ en la misma página de a (cuando $N(a)$ es creada) y luego, si es necesario, se la traslada fuera vía *división horizontal*. En términos generales, un *traslado al padre* mejora el desempeño en E/S (porque pone subárboles juntos), una *división vertical* lo mantiene (porque traslada subárboles juntos) y una *división horizontal* lo degrada. De aquí surge el orden en el cual se intentan las políticas.



(a) Antes. (b) Después.

Figura 5.5: Ejemplo antes y después de la aplicación de la política de ***división horizontal***.

5.3.4. Asegurando el 50 % de Ocupación

Las operaciones previas no aseguran que las páginas de disco tengan al menos la mitad llena. El caso de *traslado al padre* lo hace: como la página hija de $N(a)$ más el resto ha rebalsado, remover $N(a)$ no deja a la página hija con menos de la mitad llena, porque el tamaño de página es al menos el doble del tamaño de $N(a)$. En cambio, los particionados *vertical* y *horizontal* pueden crear nuevas páginas con menos de la mitad llena muchas veces, aunque garanticen que las páginas existentes queden con al menos la mitad llena.

Para forzar la ocupación de página deseada, no se permitirá la creación indiscriminada de nuevas páginas. Se apuntará todo el tiempo a *una* página de disco, la cual será *la única* a la que se le permitirá tener menos que la mitad llena (desde luego, ésta es inicialmente la página de la raíz). Ésta será llamada la página *apuntada* y siempre se mantendrá una copia de ella en memoria principal para evitar la relectura, además de mantenerla actualizada en disco.

Siempre que se deba crear una nueva página en disco, se intentará primero colocar los datos en la página apuntada. Si caben, no se creará ninguna página. Si no caben, se creará una nueva página para los nuevos datos, y ésta se convertirá en la página apuntada si y sólo si ésta contiene menos datos que la página apuntada (así sólo la página apuntada puede tener menos de la mitad llena).

A esta variante, que asegura el $50\,\%$ de ocupación de página (salvo en la página apuntada, que es la única que puede contener menos elementos), se la ha denominado como *DSAT**.

5.3.5. Búsquedas

Las búsquedas proceden igual que en el *DSAT*, por ejemplo para las búsquedas por rango se considera el Algoritmo 11. Sea T un subgrafo conectado con raíz de la estructura que se recorre durante la búsqueda, y sean L las hojas de T, las cuales no necesariamente son hojas en la estructura. Debido al diseño de la estructura en el disco, donde los nodos hermanos están siempre en la misma página, el número de páginas leídas en la búsqueda es a lo sumo $1 + |T| - |L|$, y usualmente mucho menos.

Se supone que se cuenta con suficiente espacio en memoria principal para almacenar las páginas de

disco que contengan los nodos desde el actual hasta la raíz, así que las páginas viejas de disco no se deben releer a lo largo del retroceso (*backtracking*). Esto no es un problema, porque la altura media del árbol es a lo sumo logarítmica [Nav02].

5.4. Una Variante Heurística: *DSAT+*

Como se ha mencionado, el *DSAT** que se ha descripto asegura el $50\,\%$ de la ocupación de página, pero paga un costo en términos de compacidad. Específicamente, aunque las políticas de división propuestas intentan evitarlo tanto como les sea posible, el árbol se puede fragmentar especialmente debido al uso del mecanismo de la página apuntada. En esta sección se propone una variante heurística, denominada *DSAT+*, la cual intenta obtener mejor localidad al precio de no asegurar el $50\,\%$ de ocupación (y en verdad, como se verá luego, se obtienen ocupaciones de página menores).

Las diferencias con respecto al *DSAT** son las que se detallan a continuación. En el *DSAT+*, cada subárbol con raíz en una página mantiene un puntero lejano (compuesto por página y ubicación en la página) a su padre y se mantiene un nivel global del árbol. La división vertical se aplica cada vez que falle un "traslado al padre" y existan más que una raíz de subárbol en la página. Esta estrategia divide los subárboles en dos grupos, de manera tal que la partición sea tan pareja como sea posible en cantidad de nodos, y crea una nueva página con uno de los dos grupos. Esta página es una nueva (no se utiliza ningún concepto de página apuntada). A fin de mover subárboles arbitrarios hacia otra página se usan sus punteros lejanos para actualizar los punteros al primer hijo de sus padres. Si existe sólo un subárbol en la página, la división horizontal usa el nivel global para mover todos los nodos cuyo nivel supere algún umbral a una página totalmente nueva, intentando que los tamaños sean lo más parejos posibles.

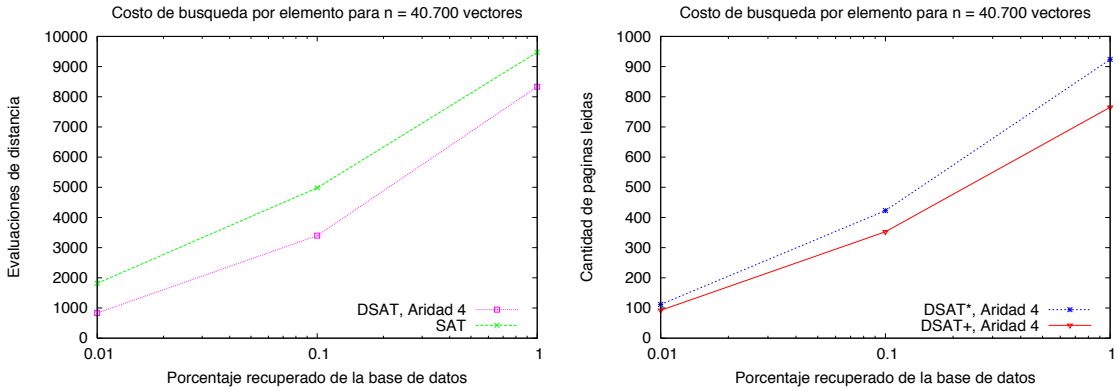
5.4.1. Sintonización Experimental

La Figura 5.6 compara los costos de búsqueda, considerando cálculos de distancia y cantidad de páginas leídas, del *DSAT** y del *DSAT+*. Se ha determinado experimentalmente la mejor aridad para cada espacio, la cual resultó ser de 4 para las Imágenes de la NASA y para los Histogramas de Color, y de 32 para el Diccionario, las mismas aridades que habían resultado mejores para la versión de memoria principal del *DSAT* [NR08]. Como base, se muestra el *SAT* original (estático y para memoria principal) [Nav02], el cual ya había sido superado por el *DSAT* (dinámico y para memoria principal) [NR08]. *DSAT**, *DSAT+* y *DSAT* tienen idénticos costos en cantidad de evaluaciones de distancia, porque el árbol es el mismo en todos los casos. Por lo tanto, en ambos sólo se ilustra el *DSAT*. Cada fila de la Figura 5.6 corresponde a los costos para un espacio.

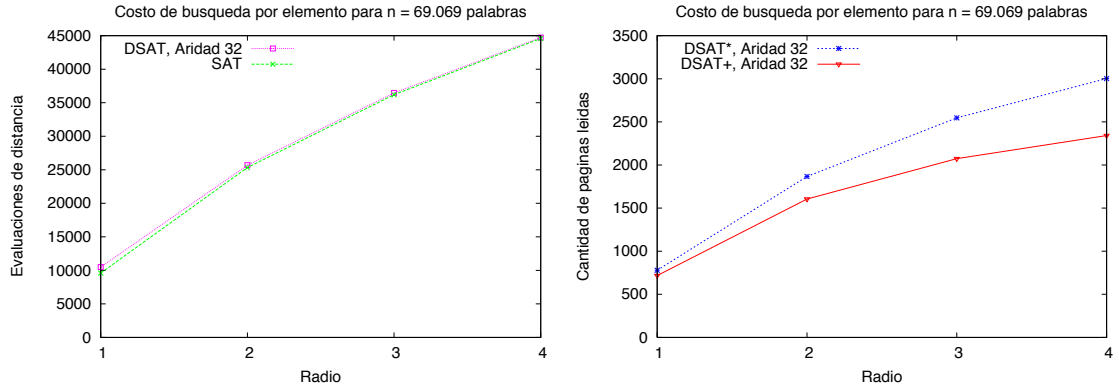
El *DSAT+* lee menos páginas que la variante *DSAT**, como una consecuencia de la mejora en la localidad y de tener menor fragmentación. La diferencia no es muy significativa en el espacio de Histogramas de Color, pero es notable en los otros dos espacios [4].

La Figura 5.7 compara los costos de construcción. El *SAT* estático se construye completo, mientras que los otros se construyen por inserciones sucesivas. Se puede observar que ambas variantes del *DSAT* se construyen rápido, en dos de los espacios las versiones dinámicas son más rápidas que la construcción del *SAT*. Cada inserción requiere unas pocas decenas de cálculos de distancia y unas pocas operaciones de E/S, y los costos crecen muy lentamente a medida que crece el tamaño de la base de datos (proporcionalmente a la profundidad del árbol). En este caso, la variante del *DSAT+* es más costosa en términos de E/S que el *DSAT**, porque éste último no modifica punteros en varias páginas de disco. La diferencia, sin embargo, es generalmente pequeña, siendo mayor en el espacio del Diccionario. Como ya se mencionó, todas las variantes del *DSAT* tienen los mismos costos en evaluaciones de distancia porque construyen

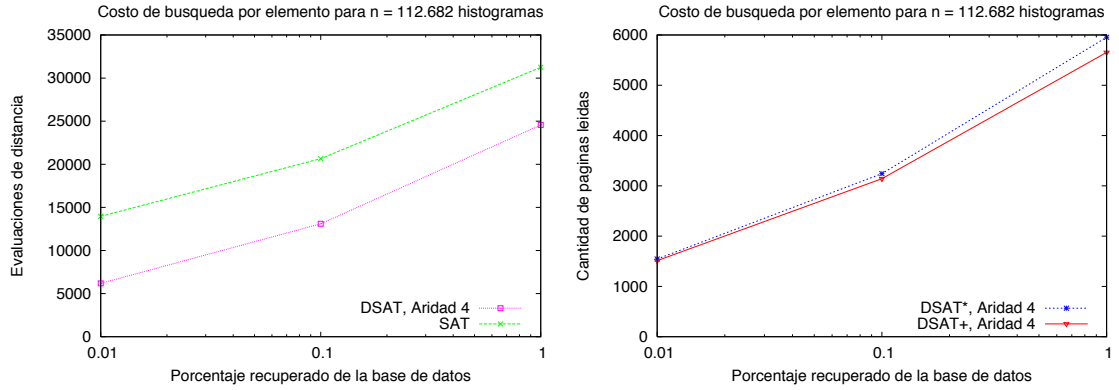
[4] Cabe destacar que el *DSAT** puede leer más páginas que la cantidad total que usa el índice. Esto se debe a que los subárboles comparten páginas y así la misma página se puede tener que leer varias veces a lo largo del proceso.



(a) Imágenes de la NASA.



(b) Diccionario.



(c) Histogramas de Color.

Figura 5.6: Costos de búsqueda de las variantes para memoria secundaria del *DSAT*, en términos de evaluaciones de distancia (izquierda) y páginas de disco leídas (derecha).

el mismo árbol. Por lo tanto, se usa sólo la leyenda *DSAT* para todas. Nuevamente, se muestra un espacio por fila. Al final, se muestra el uso del espacio en disco para las variantes del *DSAT* en memoria secundaria, mostrando tanto la ocupación promedio de las páginas como la cantidad total de páginas necesarias para almacenar el índice, para cada uno de los espacios métricos considerados.

Como se ha explicado previamente, para el *DSAT** se garantiza que la ocupación será de al menos $50\,\%$, pero en la práctica es del $75\,\%$ a más del $80\,\%$. Por otra parte, la ocupación obtenida por el *DSAT+* es algo inferior al $70\,\%$, lo cual coincide con la ocupación de páginas de disco del *árbol B* típico ($\frac{1}{\ln 2} \approx 69\,\%$). También se muestra el número total de páginas de disco usadas. El *DSAT** usa significativamente menos espacio que el *DSAT+*.

Para la comparación con otras estructuras, en la Sección 5.7, se utilizará al *DSAT+* con la mejor aridad para cada espacio, porque tiene mejor desempeño en las búsquedas a cambio de ser sólo un poco más lento de construir.

5.5. Lista de Clusters Dinámica en Memoria Secundaria

Se introduce ahora una variante dinámica para memoria secundaria de la *Lista de Clusters* (*LC*) [CN05], denominada *Lista de Clusters Dinámica* (*DLC* por su sigla en inglés: *Dynamic List of Clusters*), cuyo objetivo es indexar los espacios de mayor dimensión intrínseca. *DLC* por ser dinámica y para memoria secundaria, retiene las buenas propiedades de la *LC*, descripta en la Sección 3.6, con el agregado de requerir pocas operaciones de E/S para inserciones y búsquedas. La *DLC* se basa en la *LC* y también usa algunas ideas del *M–tree* [CPZ97]. El desafío es mantener un diseño de disco que minimice tanto los cálculos de distancia como las operaciones de E/S, y alcance una buena utilización de las páginas de disco.

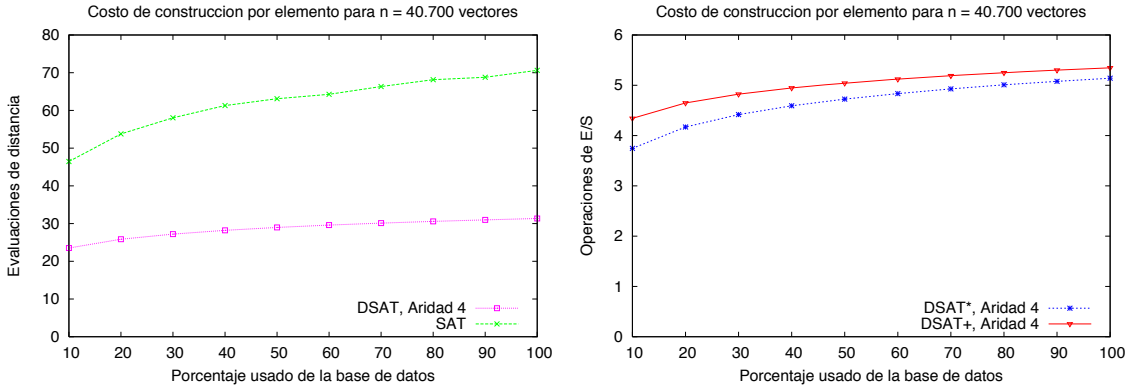
Se almacenan los objetos I de un cluster en una página de disco, de manera tal que la recuperación del cluster realice sólo una lectura de página del disco. Por lo tanto, se usan clusters de tamaño fijo m, el cual se elige de acuerdo al tamaño de página de disco B [5].

Para cada cluster C el índice almacena (1) el objeto centro $c = center(C)$; (2) su radio de cobertura $r_c = cr(C)$ (la máxima distancia entre c y cualquier objeto en el cluster); la cantidad de elementos en el cluster, $|I| = \#(C)$; y (4) los objetos en el cluster, $I = cluster(C)$, junto con las distancias $d(x, c)$ para cada $x \in I$ (se considera usar aquí la optimización a *LC* propuesta en Sección 4.6.2). Con el fin de reducir la cantidad de operaciones de E/S, se mantienen las componentes (1), (2) y (3) en memoria principal; es decir, un objeto y unos pocos números por cluster. Además se mantiene en memoria principal, junto con la información de cada cluster, la página de disco donde se encuentra efectivamente almacenado el cluster en el disco. De esta manera, se puede determinar si una zona debe ser revisada sin leer datos desde el disco. Los objetos del cluster y sus distancias al centro (la componente (4)) se mantendrán en la correspondiente página de disco.

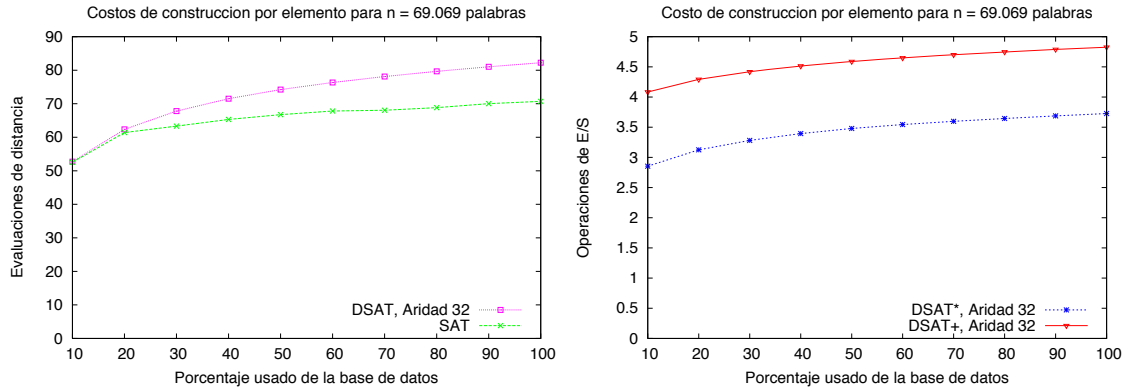
A diferencia de la *LC* estática, la estructura dinámica no garantiza que I contenga *todos* los objetos a distancia a lo sumo r_c de c, sino que sólo todos los objetos en I están a distancia a lo sumo r_c de c. Esto hace el mantenimiento mucho más simple, al costo de tener que considerar, en principio, a todas las zonas en cada consulta (esto es, se evita leer un cluster si éste no tiene intersección con la bola de consulta, pero no se puede detener la búsqueda cuando la bola de consulta está estrictamente contenida en el área del cluster).

La Figura 5.8 ilustra una *DLC*, con el conjunto de clusters en memoria secundaria y los centros en

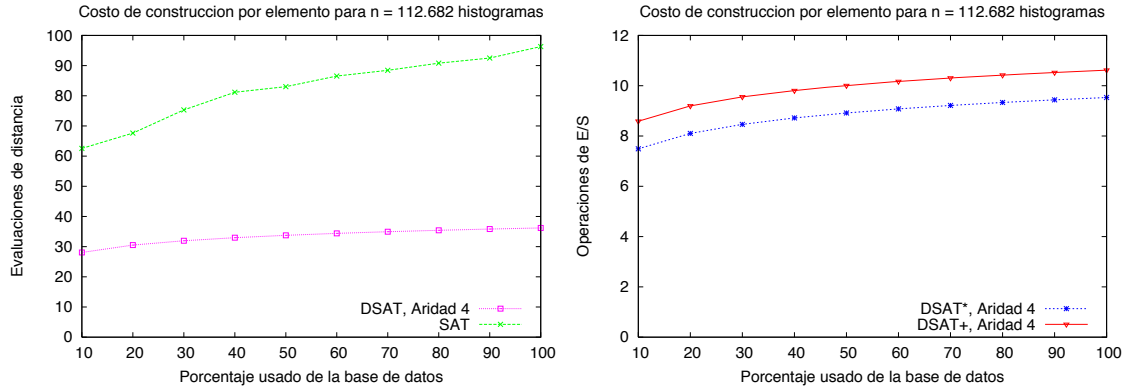
[5] En algunas aplicaciones, los objetos son tan grandes comparados a las páginas de disco, que se debe relajar esta suposición y se considera que un cluster cubre una cantidad constante de páginas de disco.



(a) Imágenes de la NASA.



(b) Diccionario.



(c) Histogramas de Color.

	Ocupación de página		**Total de páginas usadas**	
Espacio	*DSAT**	*DSAT+*	*DSAT**	*DSAT+*
NASA	80 %	67 %	1.271	1.726
Diccionario	83 %	66 %	904	1.536
Histogramas	75 %	67 %	18.781	21.136

(d) Ocupación de Espacio en Disco.

Figura 5.7: Costos de construcción de las variantes del *DSAT* en memoria secundaria, en evaluaciones de distancia (izquierda) y operaciones de E/S (derecha). La tabla muestra la ocupación de espacio.

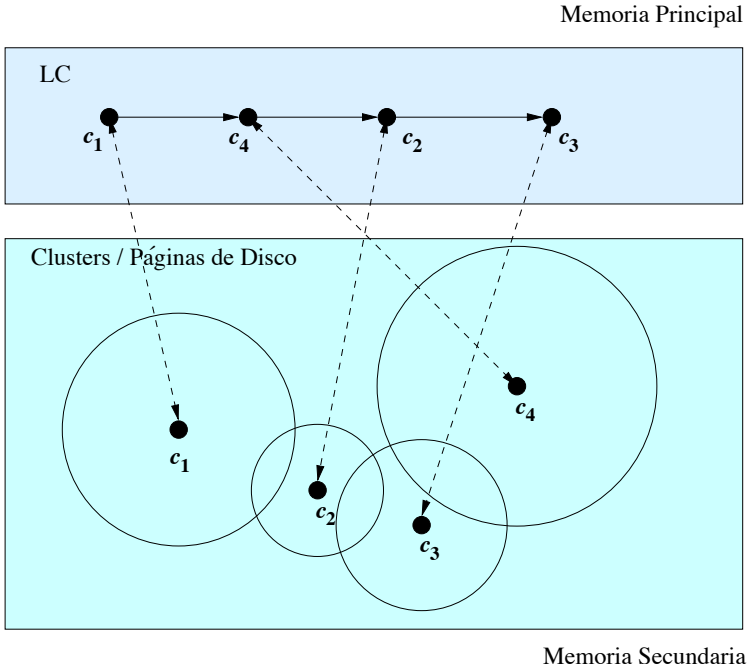


Figura 5.8: Ejemplo de una *DLC* en $\mathbb{R}^2$.

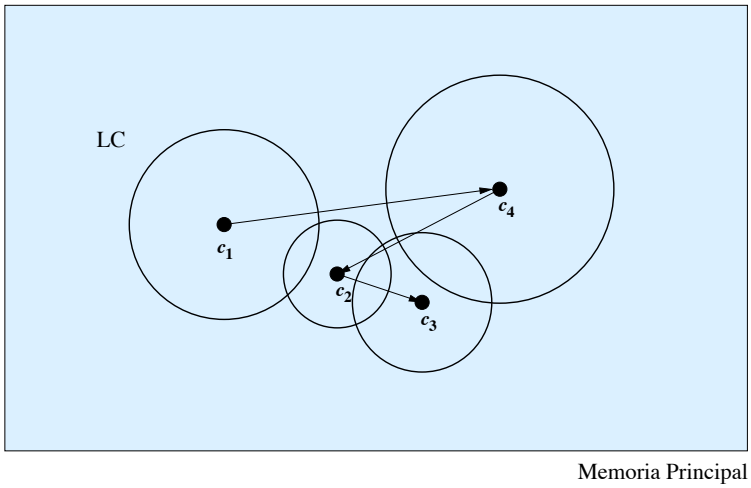


Figura 5.9: Detalles de los datos almacenados en memoria principal para la *DLC* de la Figura 5.8.

memoria principal. Cada cluster ocupa a lo sumo una página de disco. La Figura 5.9 muestra detalles de la lista de centros, donde cada centro almacena su centro de cluster $c = center(C)$, su radio de cobertura $cr(C)$, su número de elementos $\#(C)$ y la posición de la página donde se ha almacenado el cluster C.

La estructura comienza vacía y se va construyendo vía sucesivas inserciones. El primer elemento en llegar se convierte en el centro del primer cluster, y desde allí se aplica el mecanismo general de inserción que se describe a continuación.

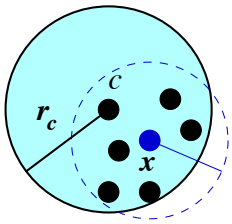


Figura 5.10: La posible reducción de una zona de cluster por la inserción de un nuevo elemento x.

5.5.1. Inserciones

Para insertar un nuevo objeto x se debe ubicar el cluster "más adecuado" para acomodarlo. La estructura del cluster podría mejorar por la inserción de x. Finalmente, si el cluster rebalsa en la inserción, debe dividirse de alguna manera.

Dos criterios ortogonales pueden determinar cuál es el cluster "más adecuado". Por una parte, elegir el cluster cuyo centro sea el más cercano a x produce zonas más compactas, las cuales menos probablemente serán leídas desde el disco y examinadas en las consultas. Por otra parte, elegir el cluster con menor ocupación de la página de disco produce mejor uso de disco, menos cluster en total, y un mejor valor para el costo de una lectura de una página. Se consideran las siguientes dos políticas para elegir el punto de inserción:

Compacidad: elige el cluster C cuyo $center(C)$ es el más cercano a x. Si hay empate, se elige aquél cuyo radio de cobertura deba incrementarse menos para incluir a x. Si aún existe empate, se elige al que tiene menos elementos.

Ocupación: elige el cluster C con menor $\#(C)$. Si existe empate, se elige aquél cuyo centro sea más cercano a x, y si aún hay empate, se elige el que deba incrementar menos su radio de cobertura.

Como es posible notar, para determinar el cluster donde el nuevo elemento x se insertará alcanza con la información que se mantiene en memoria principal, así no se realizan operaciones de E/S, sólo cálculos de distancia entre x y los centros de cluster. Cuando se ha determinado el cluster que recibirá la inserción, se incrementa el $\#(C)$ en memoria principal y se lee la correspondiente página desde memoria secundaria.

Antes de actualizar la página en el disco, se considera si x podría ser un mejor centro para C que $c = center(C)$: se calcula $cr_x = \text{máx}\{d(x,y),\ y \in I \cup \{c\}\}$, el radio de cobertura que tendría C si x fuera su centro. Si $cr_x < \text{máx}\{cr(C), d(x,c)\}$, se establece que $center(C) \leftarrow x$ y $cr(C) \leftarrow cr_x$ en memoria principal y se graba $I \cup \{c\}$ en el disco, con todas las distancias recalculadas entre los elementos y el (nuevo) centro. En caso contrario, se deja el centro actual como estaba, se actualiza $cr(C) \leftarrow \text{máx}\{cr(C), d(x,c)\}$, y se escribe $I \cup \{x\}$ en el disco, asociando x con la distancia $d(x,c)$.

Esta mejora de la calidad del cluster justifica la elección de "compacidad" de minimizar la distancia entre $d(x, center(C))$ contra, por ejemplo, elegir el centro del cluster de C con radio de cobertura resultante $cr(C)$ más chico luego de insertar a x: la inserción de elementos en los clusters de sus centros más cercanos reducirá, con el tiempo, los radios de cobertura de los clusters. La Figura 5.10 ilustra un ejemplo en $\mathbb{R}^2$.

Cuando el cluster elegido está lleno, el procedimiento a seguir es diferente. Se lo divide en dos clusters, el actual (C) y uno nuevo (N), se eligen centros para ambos (de acuerdo al "método de selección") y se elige cuáles elementos en el conjunto actual $\{c\} \cup cluster(C) \cup \{x\}$ permanecen en C y cuáles

van a N (de acuerdo al "método de partición"). Finalmente, se actualiza C y se agrega a N a la lista de clusters que se mantiene en memoria principal y se graban C y N en el disco. La combinación de un método de selección y uno de partición produce una *política de división*, varias de las cuales se han propuesto para el *M–tree* [CPZ97].

5.5.2. Políticas de División

El *M–tree* [CPZ97] considera varios requerimientos para las políticas de división:

Mínimo volumen: se refiere a minimizar el $cr(C)$;

Mínima superposición: para minimizar la cantidad de solapamiento entre dos clusters, y por ello la probabilidad que una consulta los visite a ambos; y

Máximo balance: para minimizar la diferencia en el número de elementos de ambos clusters.

El último de los requisitos es menos relevante para la *DLC*, porque no es un árbol sino una lista, pero aún es importante mantener una ocupación mínima de las páginas de disco.

El método de selección puede mantener el viejo centro c y sólo elegir uno nuevo c', denominada estrategia "confirmada" [CPZ97], o puede elegir dos nuevos centros, denominada estrategia "no confirmada". La estrategia confirmada reduce el costo de división en términos de cálculos de distancia, pero la estrategia no confirmada usualmente produce clusters de mejor calidad. De acuerdo a la notación utilizada en [CPZ97], se agrega _1 o _2 a los nombres de las estrategias dependiendo de si la estrategia de partición es confirmada o no.

RAND: El/los centro/s se eligen al azar, sin realizar cálculos de distancia.

SAMPL: Se elige una muestra aleatoria de s objetos. Para cada uno de los $\binom{s}{2}$ pares de centros, los m elementos se asignan al más cercano de los dos. Luego, los nuevos centros corresponden al par con menor suma de los dos radios de cobertura. Esto requiere $O(s^2m)$ cálculos de distancia ($O(sm)$ para la variante confirmada, donde un centro es siempre c). En los experimentos se ha usado $s = 0{,}1m$.

M_LB_DIST: Sólo para el caso confirmado. El nuevo centro es el elemento más alejado desde c. Como se han almacenado esas distancias, esto no requiere cálculos de distancia.

mM_RAD: Sólo para el caso no confirmado. Ésta es equivalente a la estrategia SAMPL con una muestra de $s = m$, su costo entonces es $O(m^2)$ cálculos de distancia.

M_DIST: Sólo para el caso no confirmado, y no se usa para el *M–tree*. Su objetivo es elegir como nuevos centros un par de elementos cuya distancia aproxime a la del par más alejado. Se selecciona un elemento del cluster al azar x, se determina el y más alejado de x, y se repite desde y el proceso durante una cantidad constante de iteraciones o hasta que la distancia al más lejano no se incremente. Los últimos dos elementos considerados serán los centros. El costo de este método es $O(m)$ cálculos de distancia.

Una vez que se han elegido los centros c y c', el *M–tree* propone dos métodos o estrategias de partición para determinar los nuevos contenidos de los clusters C y $C' = N$. El primero produce divisiones no balanceadas, mientras que la segunda no.

Partición por Hiperplano: (HYPER) asigna cada objeto a su centro más cercano.

Partición Balanceada: (BAL) comienza desde los elementos del cluster actual (exceptuando los nuevos centros) y, hasta que se hayan asignado todos los elementos, (1) mueve a C el elemento más cercano a c, (2) mueve a C' el elemento más cercano a c'.

Una tercera estrategia asegura una fracción mínima de ocupación αm, para $0 < \alpha < \frac{1}{2}$:

Partición Combinada: (MIXED) usa primero el particionado balanceado para los primeros $2\alpha m$ elementos, y luego continúa con el particionado de hiperplano con los elementos restantes.

5.5.3. Búsquedas

En una búsqueda por rango (q, r), se determinan los clusters candidatos como aquéllos cuyas zonas intersectan la bola de consulta, utilizando para ello los datos que se mantienen en memoria principal. Más precisamente, para cada C, se calcula $d = d(q, center(C))$, y si $d \leq r$ se informa como parte de la respuesta a $c = center(C)$. Independientemente, si $d - cr(C) \leq r$, se debe leer el cluster de elementos de C desde el disco y examinarlo. Cabe destacar que, en el caso dinámico, el recorrido de la lista no se puede parar si $cr(C) > d + r$, como se ha explicado previamente.

La examinación de un cluster de elementos posee una etapa de filtrado o poda: dado que se tienen almacenadas las distancias $d(x, c)$ para todo $x \in cluster(C)$, se calcula explícitamente $d(x, q)$ sólo cuando $|d(x, q) - d(q, c)| \leq r$. En otro caso, se sabe que $d(x, q) > r$ por desigualdad triangular.

Finalmente, a fin de realizar una lectura secuencial sobre el disco cuando se recuperan los clusters candidatos, y así evitar tiempos de posicionamiento innecesarios, se ordena primero toda la lista de clusters candidatos por su número de página de disco antes de comenzar a leerlos uno por uno.

Los algoritmos para realizar la búsqueda de los vecinos más cercanos se pueden obtener sistemáticamente usando las búsquedas por rango en una manera óptima [HS03b]. Para encontrar los k objetos más cercanos a q, la principal diferencia es que el conjunto de clusters candidatos se debe atravesar ordenado por el límite inferior de las distancias $d(q, center(C)) - cr(C)$, a fin de reducir el radio de búsqueda actual tan pronto como sea posible, y el proceso se detiene cuando el k-ésimo vecino más cercano conocido hasta ese momento está más cerca que el menor valor de $d(q, center(C)) - cr(C)$ de un cluster aún no explorado.

5.5.4. Eliminaciones

Un elemento x que se elimina desde la *DLC* puede ser un centro de cluster o un objeto interno de un cluster. Si x es un objeto interno de un cluster C, se lee su correspondiente página de disco, se remueve a x, y se escribe de vuelta la página al disco. En la estructura que se encuentra en memoria principal, se actualiza $\#(C)$ y se recalcula el $cr(C)$ si es necesario; es decir, si $cr(C) = d(x, center(C))$. Notar que no es necesario calcular ninguna nueva distancia para llevar a cabo este proceso.

Si, en cambio, x es el centro de un cluster C, se lee su página de disco y se elige un nuevo centro para C entre sus elementos. El nuevo centro será el elemento $y \in cluster(C)$ que minimice la máxima distancia a los demás elementos en $cluster(C) - \{y\}$, esto es el elemento cuyo cr sería el mínimo. Luego, se escribe de nuevo la página en el disco y se actualiza $center(C)$, $cr(C)$, y $\#(C)$ en memoria principal. En este caso se necesitan $O(m^2)$ cálculos de distancia. Como se puede notar, en ambos casos se necesitan sólo dos operaciones de E/S (una lectura y una escritura de página).

Si se desea asegurar una mínima proporción de ocupación $\alpha \leq 0{,}5$ en las páginas de disco, se debe también intervenir cuando la eliminación en un cluster deja su página de disco subutilizada. En este caso, se elimina el cluster completo y se reinsertan sus elementos en la estructura. Esto no es tan malo como podría parecer, debido a que los elementos del cluster tienden a ser cercanos entre ellos, y así es usual que varios elementos se inserten en la misma página, ahorrando operaciones de E/S. En el peor caso, sin embargo, se pueden necesitar $2\alpha m$ operaciones de E/S y αn evaluaciones de distancia. Si se elige no asegurar ninguna proporción de ocupación, entonces un cluster C se descartaría sólo cuando se quede sin elementos.

5.5.5. Sintonización Experimental

La Figura 5.11 ilustra los costos de búsqueda para un número de combinaciones de políticas de división. Sólo se muestran las mejores para evitar una inundación de gráficos. Como una línea de base, se muestra la *LC* estática para memoria principal [CN05]. Para la *LC*, se usa como tamaño de cluster el número máximo de elementos que caben en cada cluster de la *DLC*.

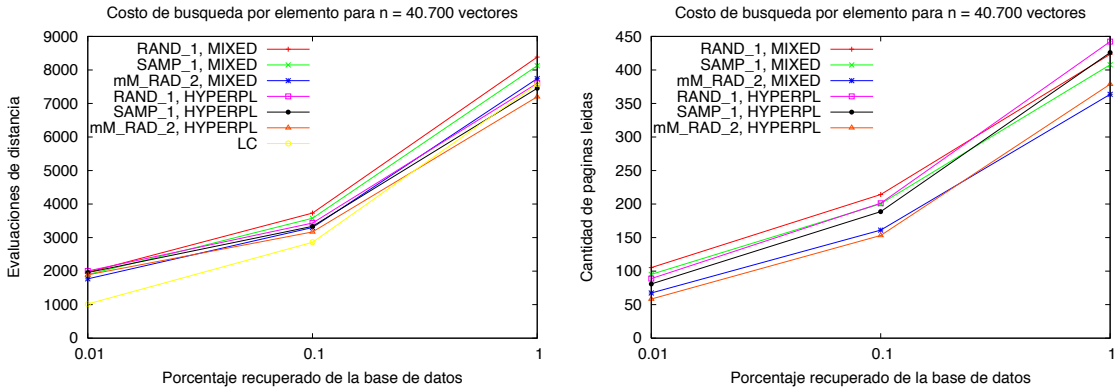
La política de compacidad siempre es mejor que la de ocupación cuando se busca el punto de inserción, así que sólo se muestra la primera. Similarmente, las particiones balanceadas obtienen peores costos de búsqueda que las otras, debido a que prioriza la ocupación por sobre la compacidad. Nuevamente, en la Figura 5.11 a cada espacio le corresponde una fila de gráficas.

En general, el desempeño de las búsquedas cambia sólo moderadamente de una política a la otra. En cambio, la *LC* estática, es significativamente mejor considerando evaluaciones de distancia, mostrando que en este caso la construcción al disponer de todos los elementos de la base de datos logra encontrar un mejor agrupamiento de los objetos que el que consigue la *DLC* al crearse por inserciones sucesivas.

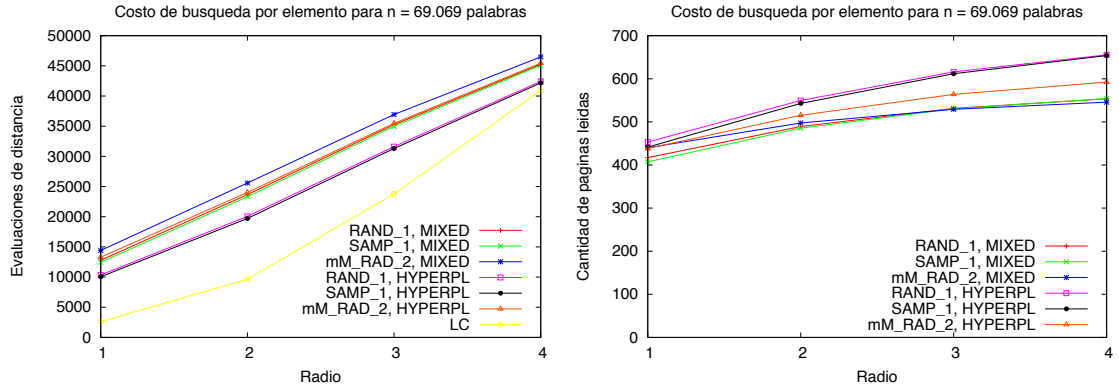
La Figura 5.12 exhibe los costos de construcción. Se debe prestar atención a que en dichas gráficas hay escalas logarítmicas en algunos ejes. Sólo se muestran las evaluaciones de distancia, porque los costos en operaciones de E/S son siempre 1 lectura de página y 1 escritura de página, más un muy pequeño número igual al número promedio de divisiones de páginas que se han producido, lo cual es la inversa del promedio de la cantidad de objetos por página de disco. Éste es básicamente el mínimo costo posible para una inserción individual en memoria secundaria. En cambio, el número de cálculos de distancia necesarios para insertar un elemento va desde cientos y aún decenas de miles de cálculos de distancia. Las peores son las estrategias no confirmadas, especialmente en casos donde los objetos son más pequeños y así las páginas de disco (es decir, los clusters) contienen un número mayor de elementos. Aún las mejores alternativas de la *DLC* requieren cientos de cálculos de distancia por inserción, un orden de magnitud mayor que las variantes del *DSAT*.

La Figura 5.12(d) muestra la ocupación de página de disco promedio obtenida con las estrategia SAMP_1 HYPERPL para el espacio de Imágenes de la NASA, SAMP_1 HYPERPL para el Diccionario y mM_RAD_2 MIXED para los Histogramas de Color. Se puede notar que las proporciones de ocupación de la *DLC* no son tan altas como las del *DSAT+*. Sin embargo, como se ha mencionado previamente, usando la estrategia de partición MIXED se puede garantizar, si se desea, una mínima ocupación en las páginas de disco.

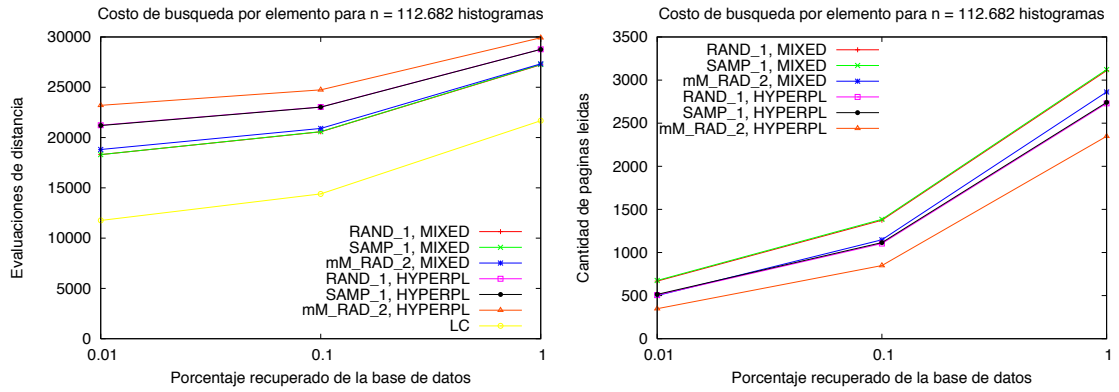
No es sencillo elegir un pequeño conjunto de variantes para la comparación que se brinda en la Sección 5.7. Se eligen las dos alternativas consideradas con mejor desempeño en cada espacio: SAMP_1 HYPERPL y SAMP_1 MIXED para el espacio de Imágenes de la NASA; SAMP_1 MIXED y SAMP_1 HYPERPL para el espacio del Diccionario; y mM_RAD_2 MIXED and SAMP_1 HYPERPL para los Histogramas de Color.



(a) Imágenes de la NASA.

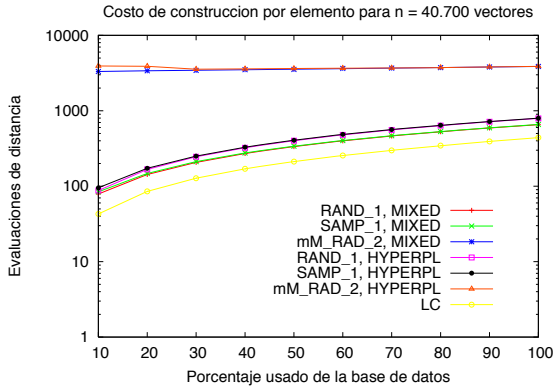


(b) Diccionario.

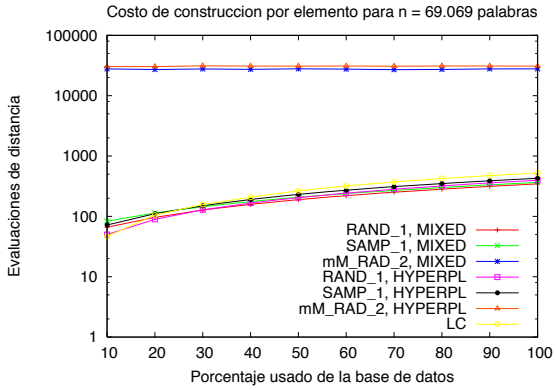


(c) Histogramas de Color.

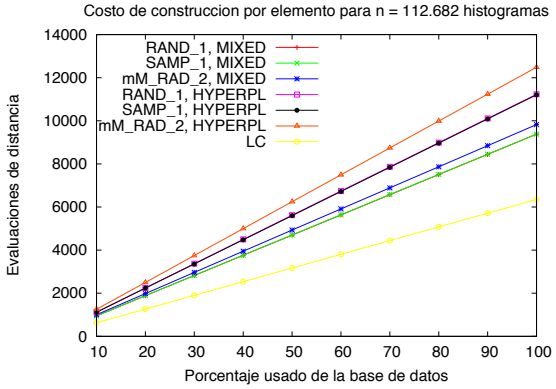
Figura 5.11: Costos de búsqueda de las variantes de *DLC* en memoria secundaria, en evaluaciones de distancia (izquierda) y número de páginas de disco leídas (derecha).



(a) Imágenes de la NASA.



(b) Diccionario.



(c) Histogramas de Color.

Espacio	Ocupación de página	Total de páginas usadas
NASA	53 %	1.450
Diccionario	60 %	713
Histogramas	59 %	19.855

(d) Ocupación de Espacio en Disco.

Figura 5.12: Costo de construcción de las variantes en memoria secundaria de *DLC* en evaluaciones de distancia. La tabla muestra la ocupación de espacio.

5.6. Conjunto Dinámico de Clusters en Memoria Secundaria

Como han evidenciado los experimentos de sintonización, las variantes de memoria secundaria del *DSAT* presentadas en la Sección 5.4 tienen buen desempeño para las inserciones y las búsquedas en términos de evaluaciones de distancia, pero el número de operaciones de E/S que realizan es alto. Por otra parte, la *DLC* descripta en la Sección 5.5 utiliza un bajo número de operaciones de E/S, pero el número de evaluaciones de distancia para inserciones es alto. Esto se debe a que se debe comparar la consulta con todos los centros de clusters c y ellos son $O(n/m) = O(n/B)$. Aunque esto ocurre en ambas consultas y actualizaciones, las consultas son usualmente costosas y esta penalización no es tan notable como en las actualizaciones.

Por lo tanto, la idea es reemplazar el recorrido secuencial de los centros por una estructura eficiente en memoria principal, para búsquedas y actualizaciones. Como los centros no se revisarán secuencialmente, su orden carece de importancia, y la estructura se denomina *Conjunto Dinámico de Clusters* (*DSC* por su sigla en inglés: *Dynamic Set of Clusters*). La estructura a utilizar debe ser capaz de soportar búsquedas, pero también inserciones y eliminaciones, para el caso donde los centros de los clusters cambien, o si se insertan nuevos clusters o se eliminan clusters existentes. Por ello, se ha elegido el *DSAT* descripto en Sección 3.5.

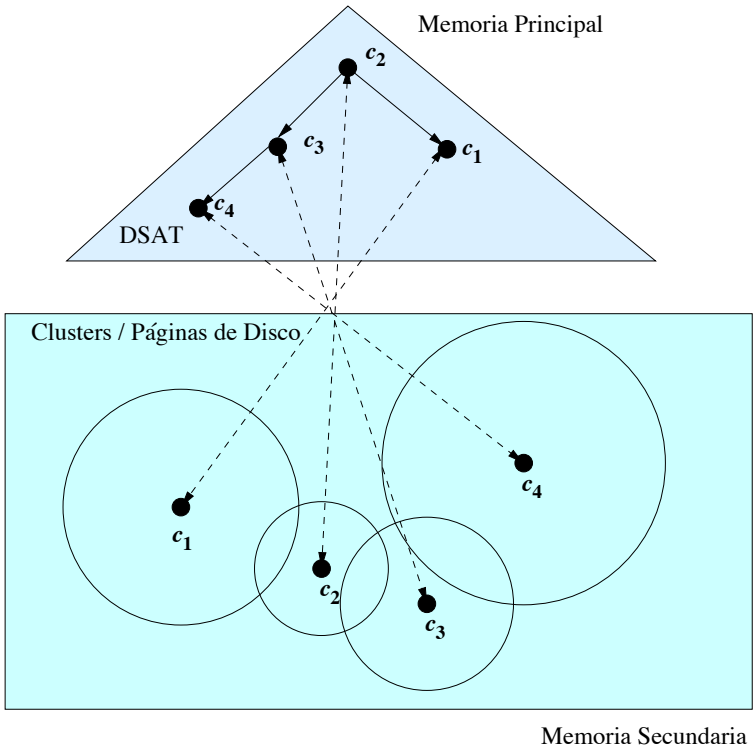


Figura 5.13: Ejemplo de un *DSC* en $\mathbb{R}^2$.

5.6.1. Diseño de la Estructura de Datos

El *DSC* usa la misma estructura de disco de la *DLC* y los clusters almacenan los mismos datos en memoria también. Sin embargo, en lugar de organizar los clusters como una lista, el conjunto de todos los centros $c = center(C)$ se mantendrá en una estructura de datos *DSAT* [NR08] en memoria principal. Además de $center(C), cr(C)$ y $\#(C)$, cada cluster almacenará en memoria principal el valor $CRZ(C)$, que es la máxima distancia entre c y cualquier elemento almacenado en el cluster de cualquier centro en el subárbol de c en el *DSAT*.

La Figura 5.13 ilustra un *DSC*, mostrando el conjunto de clusters en memoria secundaria y su *DSAT* en memoria principal. Cada cluster ocupa a lo sumo una página de disco. La Figura 5.14 expande detalles del correspondiente *DSAT*, donde se han mostrado los valores de $CRZ(C) \neq 0$. Además, también se muestra el valor de $CR(C) \neq 0$ de cada centro c en el *DSAT*, donde $CR(C)$ es el radio de cobertura de c en el *DSAT* en memoria.

La estructura comienza vacía y se construye por inserciones sucesivas. El primer elemento que arriba se convierte en el centro del primer cluster, y por lo tanto en la raíz del *DSAT* de centros. De ahí en más se aplican los mecanismos de inserción generales que se describen a continuación.

La única parte del índice que siempre está actualizada en memoria secundaria son los clusters. Además, para optimizar los accesos a disco, siempre están ubicados sobre páginas consecutivas del disco. Por lo tanto, si ocurre un problema y el *DSAT* de centros se pierde o se corrompe, se lo puede reconstruir desde la información de los clusters en un recorrido secuencial de clusters.

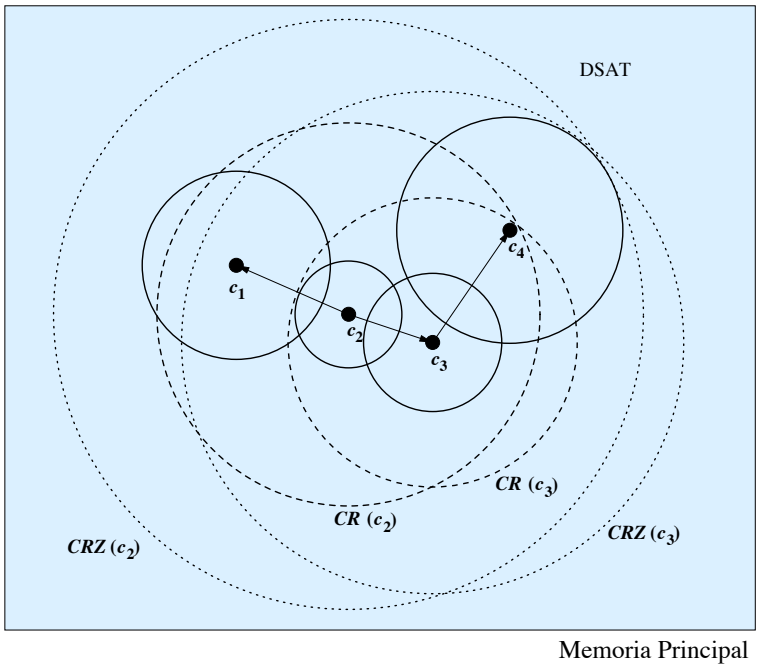


Figura 5.14: Detalles del *DSAT* en memoria principal de la Figura 5.13.

5.6.2. Inserciones

La estructura de *DSAT* soporta la política de insertar un nuevo elemento x en el cluster cuyo centro c sea el más cercano a x. Por lo tanto, sobre el *DSAT* una consulta de $k = 1$-vecino más cercano (usando el Algoritmo 12) obtiene el centro en cuyo cluster x debería insertarse. Esto no necesita ninguna operación de E/S, y requiere muchas menos evaluaciones de distancia que el recorrido secuencial que se debe realizar sobre la *DLC*.

Cuando se ha determinado el cluster C que recibirá la inserción, se lee la página correspondiente desde el disco, se agrega a x, se escribe de vuelta en el disco y se actualizan los valores de $\#(C)$ y $cr(C)$ en memoria, de la misma manera que lo hacía la *DLC*. Sin embargo, se debe considerar si x podría ser un mejor centro para C que $c = center(C)$. En tal caso, se reemplaza el centro por x, como se explicó en la Sección 5.5.1. Sin embargo, en el caso del *DSC*, se debe también actualizar el *DSAT* de centros, eliminando el viejo centro c desde el árbol e insertando a x como un centro nuevo. Estas actualizaciones sobre el *DSAT* requieren sólo cálculos de distancia. Para la eliminación en el *DSAT* [NR08] se utiliza el Algoritmo 13 y para las inserciones el Algoritmo10, descriptos en la Sección 3.5.

La inserción de un nuevo elemento x en un cluster puede incrementar el radio de cobertura $cr(C)$, y esto genera una actualización de los campos CRZ para todos los ancestros de c en el árbol. Para cada ancestro c', se establece $CRZ(c') \leftarrow \max\{CRZ(c'), d(c', x)\}$. Notar que todas estas distancias $d(c', x)$ ya se han calculado cuando la búsqueda alcanzó al nodo de c.

Cuando el cluster elegido para la inserción está lleno, el procedimiento es similar al de la *DLC*. Se divide el cluster en dos, el cluster actual C y uno nuevo N, cuyos centros para ambos, se eligen los centros de acuerdo al "método de selección" y se eligen cuáles elementos en el conjunto $\{c\} \cup cluster(C) \cup \{x\}$ permanecerán en C y cuáles van a ir a N, de acuerdo al "método de partición". Finalmente, se escribe a C y a N en el disco. Para completar el procedimiento, se remueve el viejo centro c del *DSAT* e inserta los nuevos elementos $center(C)$ y $center(N)$, pudiendo evitar la eliminación de c e insertar $center(C)$ si ambos son el mismo elemento.

Se usan los Algoritmos 10 y 13 para insertar un nuevo centro en el *DSAT* de centros y eliminar el viejo centro, respectivamente. Se usan también las optimizaciones descriptas para la eliminación en la Sección 3.5.3.

Cabe notar que, cuando se inserta un nuevo centro en el *DSAT* con su cluster ya definido (como ocurre después de una división o split), se podrían recalcular exactamente los campos CRZ de todos los nodos en el camino de inserción, comparándolos con cada elemento en el $cluster(C)$. Sin embargo, esto es demasiado costoso. En su lugar, cada campo es recalculado como $CRZ(c') \leftarrow \max\{CRZ(c'), d(c', c)+ cr(c)\}$, lo cual da un límite superior. Similarmente, cuando se eliminan elementos en el *DSAT*, los campos CRZ no se actualizan fácilmente. En este caso simplemente se dejan los viejos valores, ignorando el hecho que los nuevos podrían ser más pequeños. Recordar que esta estructura del *DSAT* es volátil; cuando se usa nuevamente el sistema, se construye un nuevo *DSAT* desde los centros almacenados en disco. Por lo tanto, esta ineficiencia en la actualización de los campos CRZ no es acumulativa.

5.6.3. Búsquedas

Las búsquedas de (q, r) también pueden aprovecharse de la existencia de la estructura del *DSAT* de centros para encontrar todas las zonas que intersecten la bola de consulta. La única diferencia es que cada nodo c representa una bola de radio $cr(c)$ alrededor de c. Por lo tanto, se puede usar esencialmente el mismo Algoritmo 11, con la diferencia que se debe usar $CRZ(a)$ en lugar de $R(a)$ en la línea 1. También, la búsqueda por rango colecciona cualquier elemento a tal que $d(q, a) \leq r + cr(a)$, en la línea 2.

Como se hace con la *DLC*, primero se determinan todos los centros c cuyos clusters deben ser examinados, se los ordena por su posición en disco y luego se los lee uno por uno desde disco (con el fin de reducir el tiempo de posicionamiento en disco), para finalmente examinarlos en memoria principal. Se reportan los centros que satisfacen que $d(q, c) \leq r$. Como antes, se usan las distancias almacenadas en los clusters C en disco para tratar de evitar el cálculo de $d(x, q)$ para cada $x \in cluster(C)$.

Las búsquedas de vecinos más cercanos pueden aprovechar también la estructura del *DSAT* de centros almacenda en memoria principal. En lugar de insertar a todos los centros c en la cola de prioridad, se inserta el centro de la raíz del *DSAT*. La prioridad de los nodos c será su límite inferior de la distancia a q, $d(c, q) - CRZ(c)$. Se extrae el próximo elemento, se examina exhaustivamente su cluster (usando el filtrado basado en la información de las distancias precalculadas $d(x, c)$ almacenadas), y se insertan sus hijos del *DSAT* en la cola.

5.6.4. Eliminaciones

Cuando se elimina un elemento x en el *DSC*, el proceso es muy similar al de la eliminación en la *DLC*; pero se actualiza, si es necesario, la información en el *DSAT* de centros. Si x no es el centro de su cluster C, no se necesitar hacer ninguna acción sobre el *DSAT* (de nuevo, no se tratan de recalcular los valores posiblemente más pequeños de los campos CRZ de todos los ancestros de $center(C)$).

En cambio, si $x = center(C)$, se elige un nuevo centro para C. En este caso, se elimina a x en el *DSAT* y se inserta el nuevo $center(C)$, con su correspondiente valor de cr. Esto no produce ninguna operación adicional de E/S.

Cuando un cluster debe descartarse y deben ser reinsertados sus elementos, porque su número de elementos es menor que un cierto umbral, se trata de evitar la búsqueda de los puntos de inserción uno por uno en el *DSAT* de centros. Se elige un elemento central x entre los elementos que deberán

reinsertarse y se determina su centro más cercano $c = center(C)$ en el *DSAT*. Luego, se inserta x en el $cluster(C)$ y a tantos elementos más cercanos a c, entre los elementos a ser reinsertados, como sea posible. Si $cluster(C)$ se llena, se reinicia el proceso para los restantes elementos. Sin embargo, si el cluster C se divide luego de insertar a x, se aprovecha el hecho que se tienen dos clusters cercanos medio llenos y se insertan los restantes elementos en los dos nuevos clusters. Cada elemento se inserta en el cluster de su centro más cercano.

En el peor caso, una eliminación puede forzar una reconstrucción completa del *DSAT* de centros en memoria principal. Si m es el tamaño máximo de un cluster y la base de datos contiene n elementos, la recontrucción total de este *DSAT* requiere $O((n/m)\log(n/m))$ cálculos de distancia. Por otra parte, se requieren sólo 5 operaciones de E/S para una eliminación en el peor caso. Éstas se pueden necesitar si se tiene que eliminar un cluster C por no contener suficientes elementos:

1. Para mantener las páginas consecutivas en disco, se lee la última página y se la escribe en la página del cluster eliminado:
2. se tiene que leer la página de disco del cluster cuyo centro sea el más cercano a $center(C)$;
3. cuando se insertan los elementos de C en este nuevo cluster, éste puede rebalsar. Dividirlo requiere escribir dos páginas en el disco.

5.6.5. Sintonización Experimental

Como para la *DLC*, se compara el efecto de las distintas políticas para el *DSC*, mostrando sólo aquéllas que tuvieron mejor desempeño. La Figura 5.15 muestra los costos de búsqueda para cada espacio, considerando la cantidad de evaluaciones de distancia y el número de páginas leídas. Cada fila representa a un espacio. Ambos costos son similares a los de la *DLC*.

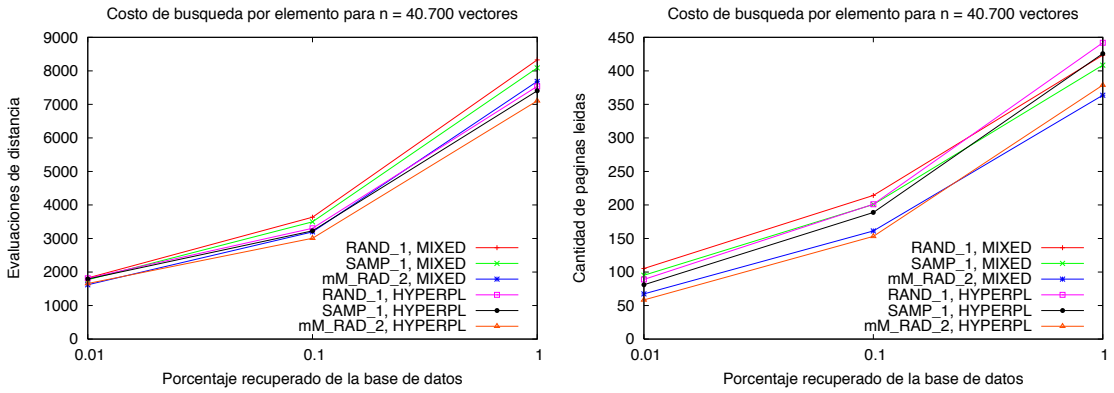
Las mejores alternativas son, en general, mM_RAD_2, RAND_1 y SAMP_1 para selección de centros y particionado de hiperplanos puro (HYPERPL) o combinado con balanceo (MIXED). Como es esperable, el particionado balanceado obtiene peores costos que los otros, porque prioriza la ocupación por sobre la compacidad.

La Figura 5.16 muestra los costos de construcción por elemento, considerando la cantidad de evaluaciones de distancia. Como para la *DLC*, la cantidad de operaciones de E/S es básicamente 1 lectura y 1 escritura de página por inserción. Como es posible observar, las estrategias de partición MIXED obtienen mejores costos en las inserciones y, como antes, la estrategia de selección de centros mM_RAD_2 es la más costosa.

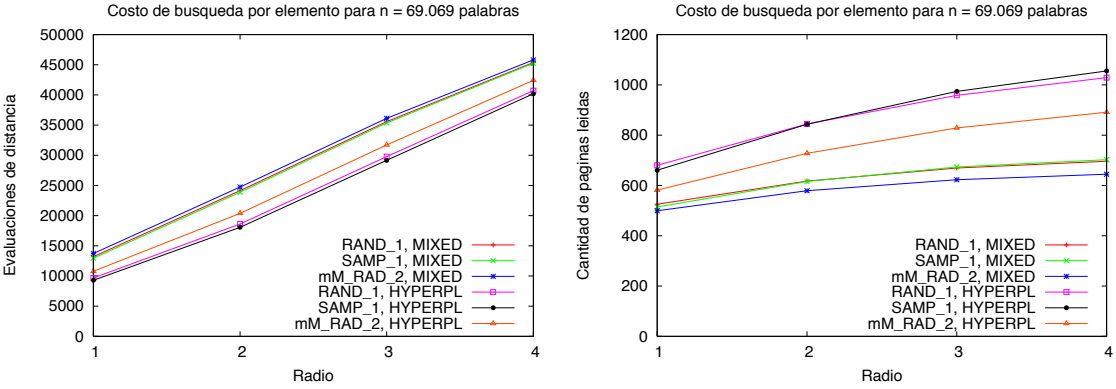
Mientras que los costos de construcción aún son de cientos de cálculos de distancia, y así un orden de magnitud mayores que los del *DSAT*, ellos son en general significativamente más bajos que los de la *DLC*. En particular, se puede notar que el incremento lineal en un costo de inserción de la *DLC* se ha convertido claramente en sublineal, aunque no es tan bajo como el costo aparentemente logarítmico de la inserción en el *DSAT*.

La Figura 5.16(d) muestra una tabla con la ocupación de espacio en disco, la cual es similar para *DLC* y *DSC*. Estos valores corresponden a usar las variantes mM_RAD_2 HYPERPL para el espacio de Imágenes de la NASA, mM_RAD_2 MIXED para el Diccionario y para el espacio de Histogramas de Color.

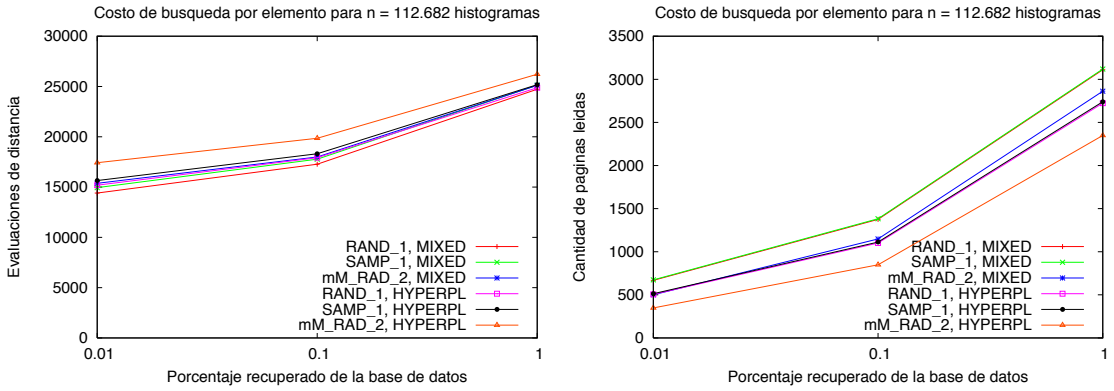
Para la comparación que se realiza en la Sección 5.7, se eligen las siguientes variantes, considerando un balance entre evaluaciones de distancia y operaciones de E/S: se selecionan mM_RAD_2 MIXED y



(a) Imágenes de la NASA.



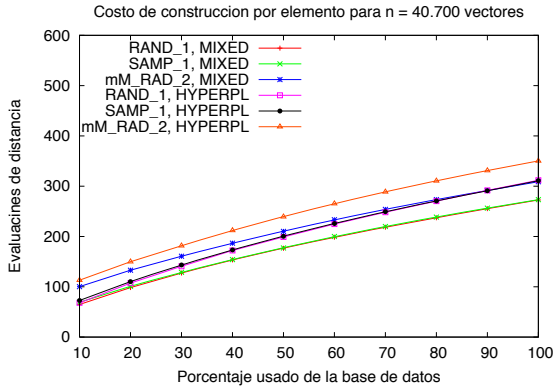
(b) Diccionario.



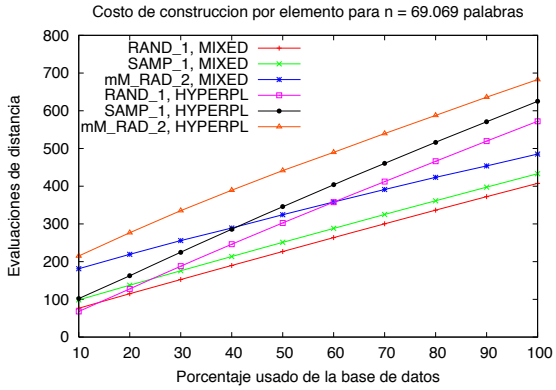
(c) Histogramas de Color.

Figura 5.15: Costos de búsqueda para las variantes de *DSC* en memoria secundaria, en términos de evaluaciones de distancia (izquierda) y páginas de disco leídas (derecha).

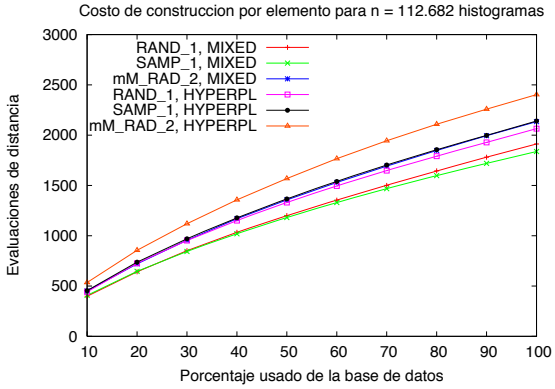
mM_RAD_2 HYPERPL para el espacio de Imágenes de la NASA, las cuales tienen costos de inserción excesivamente altos en *DLC* pero no en *DSC*; mM_RAD_2 HYPERPL y mM_RAD_2 MIXED para el Diccionario de palabras en inglés; y RAND_1 HYPERPL y mM_RAD_2 MIXED para los Histogramas de Color.



(a) Imágenes de la NASA.



(b) Diccionario.



(c) Histogramas de Color.

Espacio	Ocupación de página	Total de páginas usadas
NASA	53 %	1.429
Diccionario	64 %	665
Histogramas	59 %	17.656

(d) Ocupación de Espacio en Disco.

Figura 5.16: Costo de construcción de las variantes de *DSC* en memoria secundaria, en evaluaciones de distancia. La tabla muestra la ocupación de espacio.

5.7. Comparación Experimental

En esta sección se evalúan empíricamente las mejores alternativas de cada una de las estructuras de datos presentadas, comparándolas con trabajos previos sobre los espacios métricos pequeños usados para sintonizar parámetros y luego sobre espacios métricos masivos.

5.7.1. Comparando con el *M–tree*

Se comienza con una comparación con el punto de comparación clásico de la literatura, el *M–tree* [CPZ97], cuyos códigos están fácilmente disponibles.

La Figura 5.17 compara los costos de búsqueda. En términos de las evaluaciones de distancia, el *DSAT+* siempre supera al *M–tree*. El *DSAT+* trabaja mejor para las consultas más selectivas; es decir, de radios más chicos. Por ejemplo, sobre el espacio de Imágenes de la NASA, el *DSAT+* es la mejor opción para la mayoría de las consultas selectivas, pero para radios más grandes el *DSC* es mejor. En el espacio del Diccionario, en cambio, el *DSAT+* es superado por el *DSC* aún para el radio $r = 1$. Sin embargo, en el espacio de Histogramas de Color, el *DSAT+* es la mejor opción, aún para recuperar el $1\,\%$ de la base de datos. La *DLC* sigue de cerca al *DSC*, y ambas estructuras superan en casi todos los casos al *M–tree*.

	Ocupación de página			Total de páginas usadas			
Espacio	*DSAT+*	*DLC*	*DSC*	*DSAT+*	*DLC*	*DSC*	*M–tree*
NASA	67 %	53 %	53 %	1.726	1.450	1.429	1.973
Diccionario	66 %	60 %	64 %	1.536	713	665	1.608
Histogramas	67 %	59 %	59 %	21.136	19.855	17.656	31.791

Tabla 5.1: Utilización de espacio en disco de los índices seleccionados.

Cuando se considera la cantidad de páginas de disco leídas, ambos el *DSAT+* y el *M–tree* son superados por las variantes de *DLC* y *DSC*. Estas estructuras apenas se distinguen, excepto sobre el espacio del Diccionario, donde *DLC* es mejor. La comparación entre el *DSAT+* y el *M–tree* es variada. En el espacio de Histogramas de Color, el *M–tree* es dos veces más lento, en el Diccionario es dos veces más rápido y en el espacio de Imágenes de la NASA los resultados dependen de la selectividad de la consulta.

La Figura 5.18 compara los costos de construcción, en evaluaciones de distancia y en cantidad de operaciones de E/S. En términos de evaluaciones de distancia, el *DSAT+* es insuperable: requiere sólo unas pocas decenas de evaluaciones de distancia por inserción. *DSAT+* supera al *M–tree* aproximadamente por un factor de 2. En cambio, el *DSC* requiere unos pocos cientos de evaluaciones de distancia y usualmente supera a la *DLC*, la cual es la más costosa; excepto sobre el Diccionario, donde *DLC* es más rápida que el *DSC*. Es también evidente que el crecimiento del costo en el *DSAT+* y en el *M–tree* es mucho más lento que en el *DSC*, y éste es más lento que sobre la *DLC*. Es posible observar que, aunque el *M–tree* requiere menos evaluaciones de distancia para insertar un elemento que la *DLC* y el *DSC*, necesita más operaciones de E/S porque usa más páginas de disco. Esto es porque el *M–tree* es rápido para determinar el punto de inserción de un nuevo elemento, pero no encuentra el mejor. En cambio, el *DSC* y la *DLC* tienen una inserción más costosa porque gastan más en determinar el mejor cluster donde insertar el nuevo elemento. Los resultados experimentales en la Figura 5.17 validan esta conjetura, porque las búsquedas son más baratas, en operaciones de E/S y evaluaciones de distancia, en el *DSC* y la *DLC* que en el *M–tree*.

En términos de operaciones de E/S, la situación es básicamente la opuesta. Ambas estructuras, *DLC* y *DSC*, requieren casi 2 operaciones de E/S por inserción, mientras que el *DSAT+* requiere 4 a 10, y el *M–tree* requiere más, 6 a 20.

La Tabla 5.1 muestra el uso de espacio en disco de los distintos índices, en ocupación de páginas y en cantidad total de páginas de disco usadas. Es posible observar que el *DSAT+* logra la mejor ocupación de página de disco, alrededor del $67\,\%$. ésta cae a $53\,\% - 64\,\%$ para la *DLC* y el *DSC*. Curiosamente, cuando se considera el número total de páginas de disco usadas, resulta que el *DSAT+* es la estructura que más espacio consume, mientras que *DLC* y *DSC* necesitan menos páginas. Esto es se debe a que *DLC* y *DSC* almacenan menos datos por objeto. En esta comparación, el *M–tree* es la estructura de datos que más espacio consume en disco. Cabe destacar que como *DLC* y *DSC* utilizan mucho menos espacio en disco, realmente las operaciones de E/S que realiza cualquier operación sobre estas estructuras serán menos costosas que en los otros índices, porque el tiempo de posicionamiento en disco de cada acceso es directamente proporcional a la cantidad de espacio (cilindros) en los que debe moverse la cabeza de lectura/escritura.

En general, se pueden sacar las siguientes conclusiones:

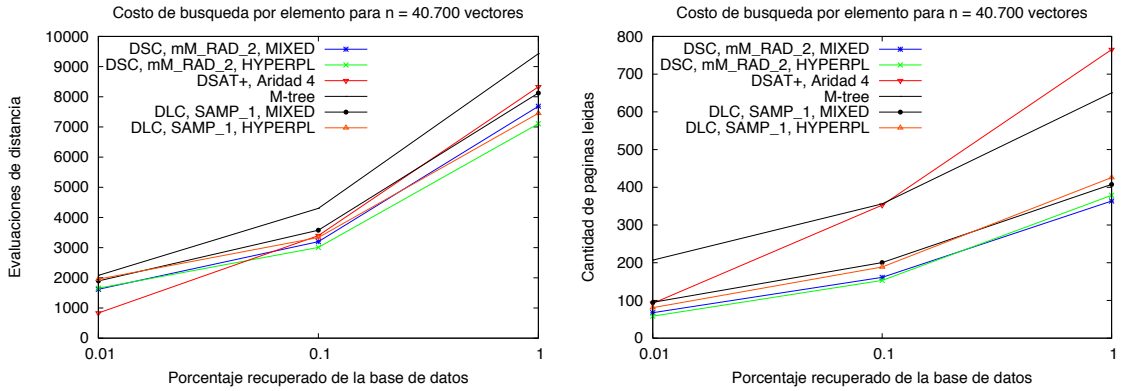
1. El *DSAT+* es una estructura que ofrece un buen balance entre costos de búsqueda e inserción, usualmente supera al *M–tree* (excepto, en algunos casos, en operaciones de E/S para las búsquedas). Tiene mejor desempeño sobre consultas más selectivas. Es más competitivo en evaluaciones de distancia, mientras que en operaciones de E/S generalmente es superado por la *DLC* y el *DSC*. Finalmente, el *DSAT+* carece de un soporte eficiente para las eliminaciones.

2. El *DSC* posee mejor desempeño cuando las consultas son menos selectivas, pero aún cuando no lo son, usualmente supera a las otras estructuras en las búsquedas, tanto en evaluaciones de distancia como en cantidad de páginas leídas. También supera a los otros índices en operaciones de E/S para inserciones, donde esencialmente realiza sólo dos operaciones de E/S. Su punto débil es la cantidad de evaluaciones de distancia requeridas para las insercions. Aunque esta cantidad es sublineal, y generalmente mejor que la que requiere la *DLC*, su valor es un orden de magnitud superior que los que requieren el *DSAT+* y el *M–tree*.

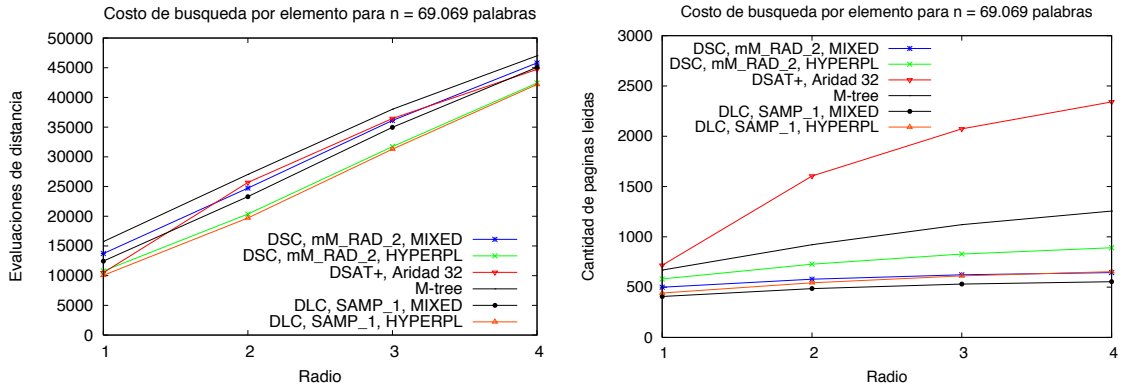
5.7.2. Eliminaciones

Dado que el *DSAT+* no soporta eliminaciones y la *DLC* generalmente es superada por el *DSC*, se estudia ahora el efecto de las eliminaciones sobre la última estructura propuesta. Se estudian ambos el costo de una eliminación y la degradación que sufren los costos de eliminación y búsqueda luego de haber realizado un número significativo de eliminaciones. Para estos experimentos, sobre un índice que contiene n elementos, se seleccionan aleatoriamente una fracción dada de elementos y se los elimina desde el índice, de manera tal que *después* de las eliminaciones el índice contenga n elementos. Por ejemplo, si se busca en la mitad del espacio del Diccionario luego de un $30\,\%$ de eliminaciones, significa que se insertaron 49.335 elementos y luego se removieron 14.800 de ellos, para dejar los mismos 34.534 elementos (la mitad del conjunto). El índice donde se busca contiene siempre la misma mitad del espacio. Para cada espacio métrico se usan las alternativas consideradas en la Tabla 5.1: mM_RAD_2 HYPERPL para Imágenes de la NASA, SAMP_1 HYPERPL para el Diccionario y mM RAD_2 MIXED para Histogramas de Color.

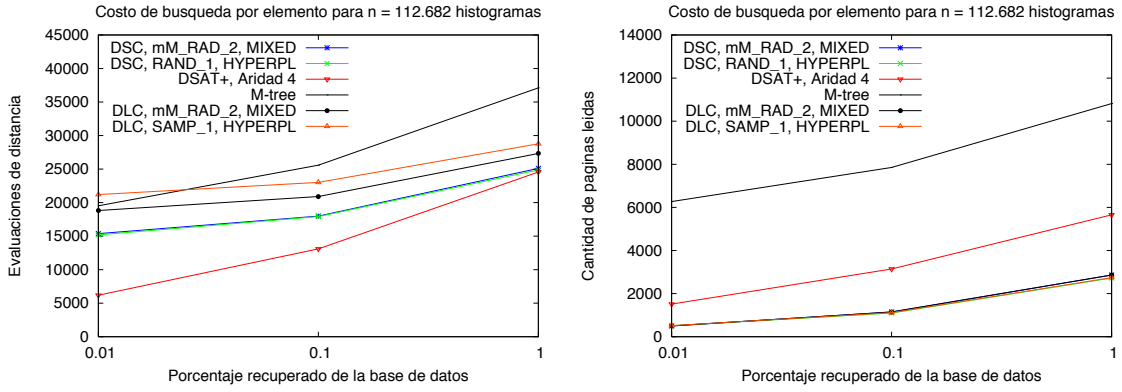
La Figura 5.19 muestra los costos de eliminación a medida que crece el porcentaje de eliminaciones. La cantidad de operaciones de E/S requerida por las eliminaciones se mantienen en 2–3, mientras que el número de cálculos de distancia necesarios para restablecer la corrección del índice luego de remover el elemento es relativamente bajo: unas pocas decenas para los espacios de Imágenes de NASA y para el Diccionario, y unos pocos cientos para el espacio de Histogramas de Color, lo cual es cercano al costo de una inserción. Estos números no cambian significativamente cuando el porcentaje de eliminaciones se incrementa.



(a) Imágenes de la NASA.

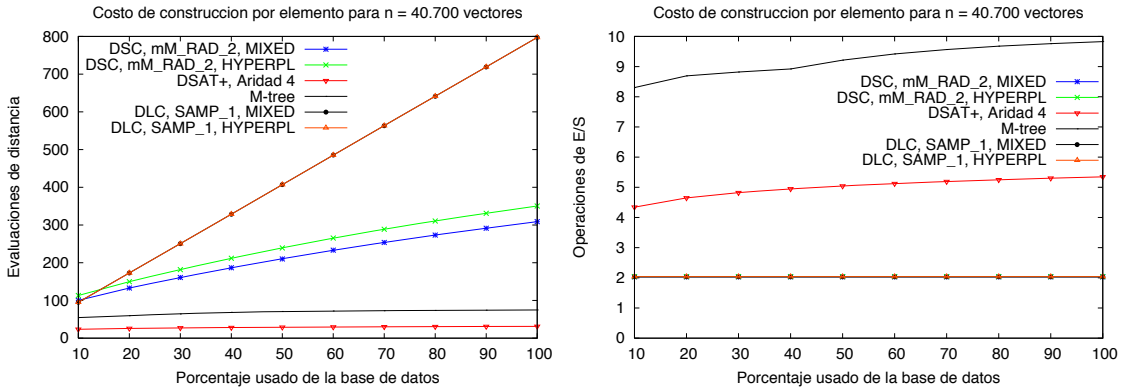


(b) Diccionario.

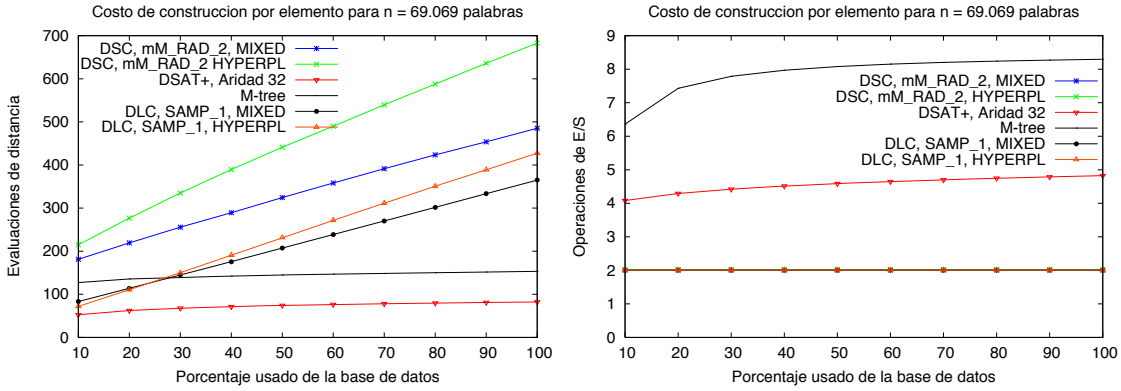


(c) Histogramas de Color.

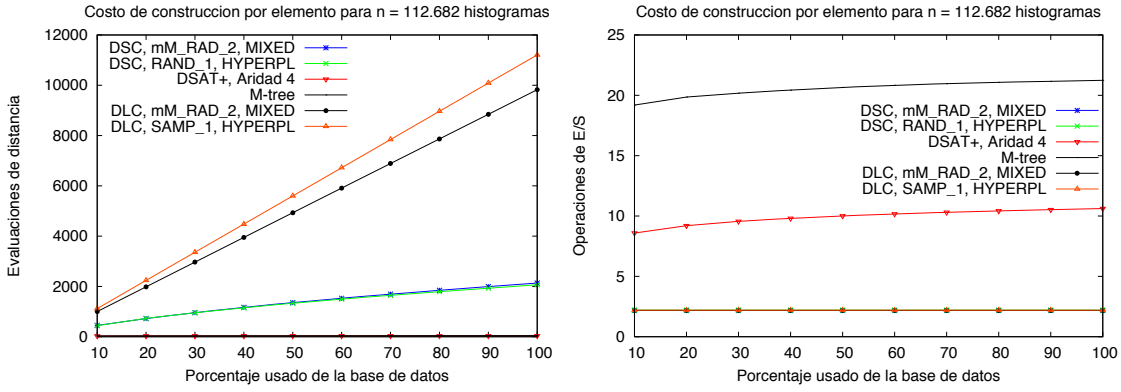
Figura 5.17: Comparación de los costos de búsqueda de *DSC*, *DLC*, *DSAT+* y *M–tree*, considerando evaluaciones de distancia (izquierda) y cantidad de páginas leídas (derecha).



(a) Imágenes de la NASA.

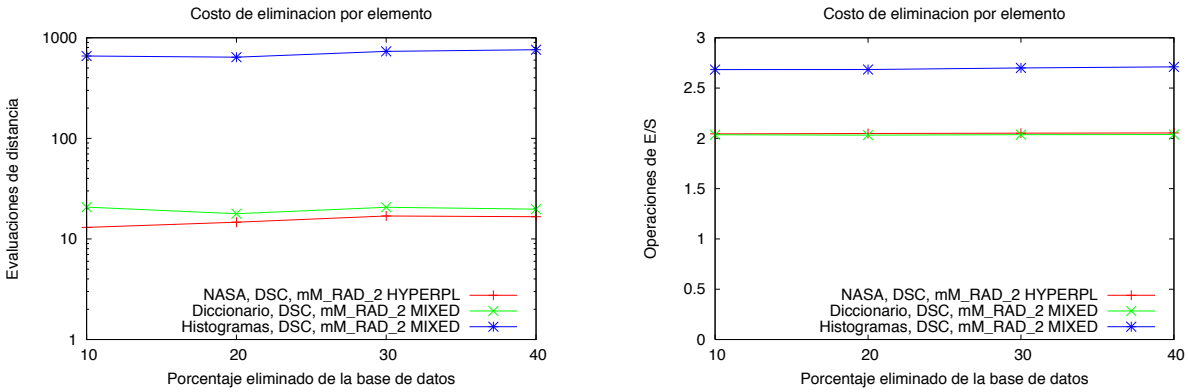


(b) Diccionario.



(c) Histogramas de Color.

Figura 5.18: Costos de construcción de *DSC*, *DLC*, *DSAT+* y *M–tree*, considerando evaluaciones de distancia (izquierda) y operaciones de E/S (derecha).



(a) Evaluaciones de Distancia.

(b) Operaciones de E/S.

Figura 5.19: Costos de eliminación para las mejores alternativas de *DSC*, considerando evaluaciones de distancia (izquierda) y operaciones de E/S (derecha).

La Figura 5.20 exhibe cómo las eliminaciones masivas degradan las búsquedas. La degradación se nota en algunos casos, tanto en evaluaciones de distancia como en operaciones de E/S, alcanzando como un $50\,\%$ extra en operaciones de E/S en el espacio de Imágenes de la NASA, cuando se ha eliminado un $40\,\%$ de la base de datos. Esta degradación se debe a dos hechos: las eliminaciones en el *DSAT* de centros pueden dejar los radios de cobertura CRZ sobreestimados y tener más clusters con menos elementos reduce el valor de cada lectura de página. Notar que la primera razón no es significativa en el largo plazo, dado que el *DSAT* de memoria principal se reconstruye cada vez que el motor de búsqueda es reiniciado.

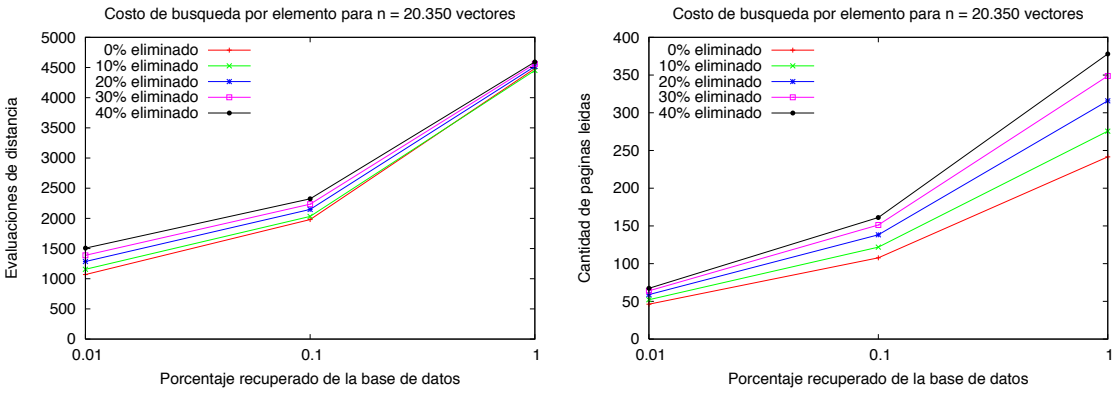
5.7.3. Estudio de *DSC* sobre Espacios Masivos

Se han usado hasta ahora conjuntos de datos relativamente pequeños. En esta sección se consideran conjuntos de datos un orden de magnitud más grandes. Primero se estudian espacios de 1.000.000 de vectores uniformemente distribuidos, con dimensiones 10 y 15. Como ya se ha mencionado, no se utiliza explícitamente la información de coordenadas de los vectores, sólo se los trata como objetos de caja negra con la distancia Euclidiana. Estos conjuntos de datos son llamados *Vectores*. Luego, se considera el espacio real de 1.000.000 de vectores denominado *Flickr* (descripto en Sección 2.10). Sobre estos espacios masivos, se verifican las conclusiones obtenidas para *DSC*, la mejor de las estructuras propuestas que soporta dinamismo completo; es decir, inserciones y eliminaciones de elementos de la base de datos. También se analiza un tamaño de página de disco de 8KB.

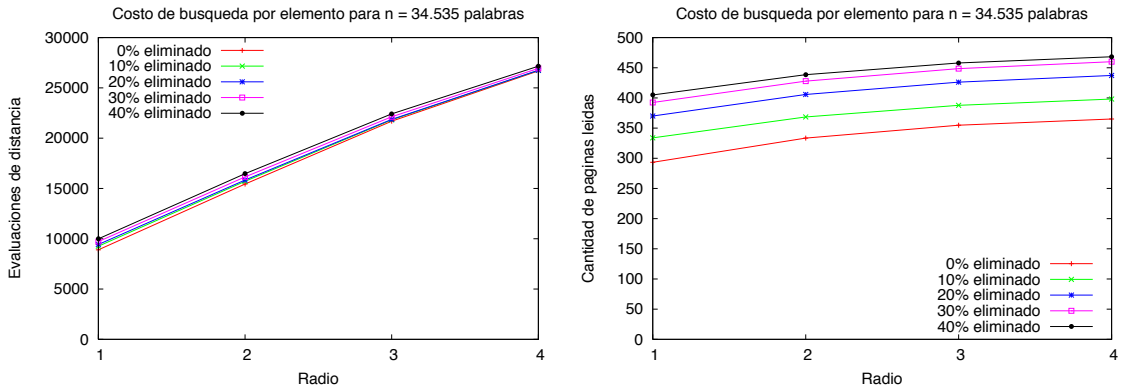
Espacios de Vectores Uniformes

La Figura 5.21 ilustra los costos de construcción y búsqueda en el espacio de Vectores. Los costos de construcción muestran el número de cálculos de distancia por elemento insertado; la cantidad de operaciones de E/S es siempre 2–3 y por lo tanto no se muestra. Para los costos de búsqueda se muestran ambas medidas: cantidad de evaluaciones de distancia y de páginas leídas.

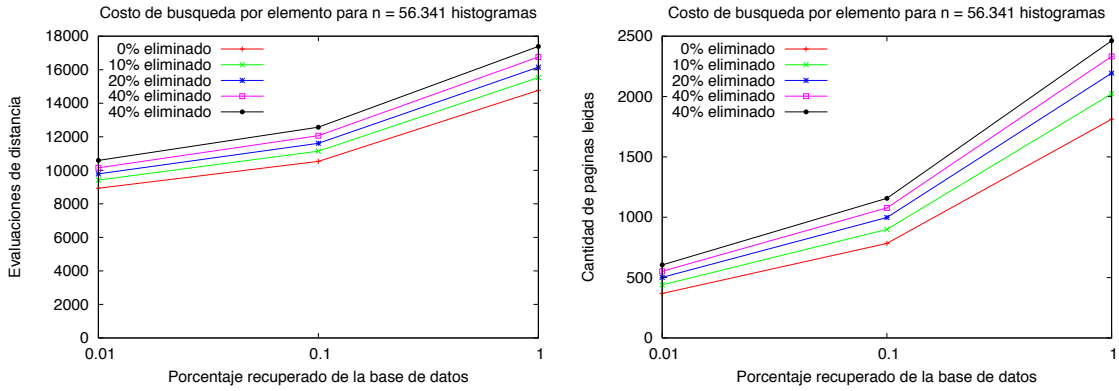
En estos espacios, el efecto de algunos grupos de políticas es claramente más visible. Por ejemplo, las estrategias MIXED producen mejores costos de construcción que las estrategias HYPERPL, pero también claramente se desempeñan peor en las búsquedas, en términos de evaluaciones de distancia. En



(a) Imágenes de la NASA.



(b) Diccionario.

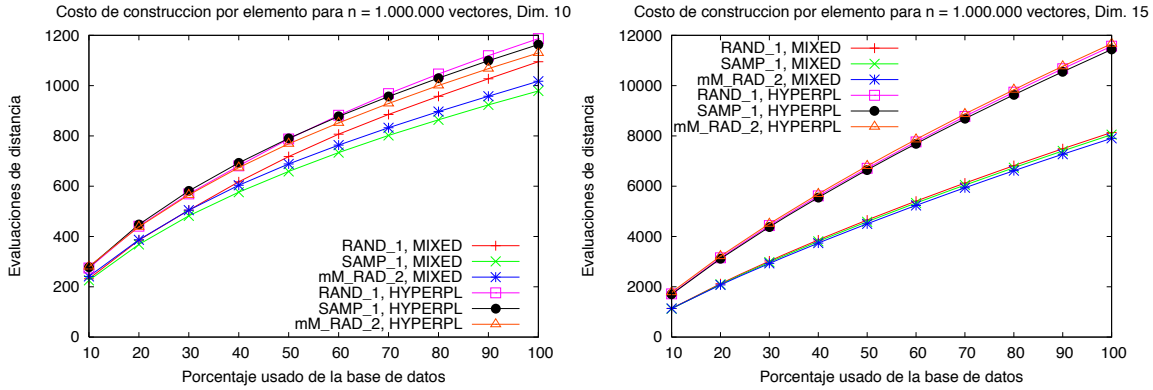


(c) Histogramas de Color.

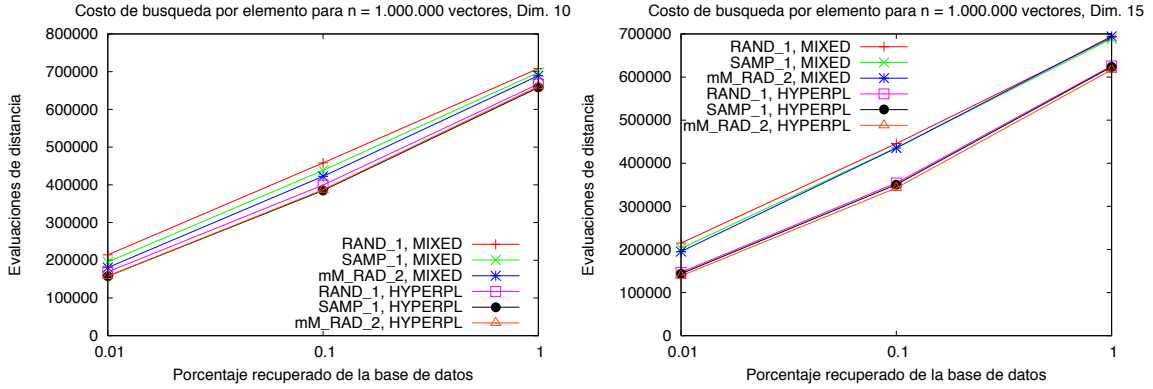
Figura 5.20: Comparación de los costos de búsqueda a medida que la fracción de elementos eliminados de la base de datos aumenta, para las mejores alternativas de *DSC*.

términos de E/S, las estrategias HYPERPL son levemente mejores sobre consultas más selectivas, pero para las menos selectivas las estrategias MIXED claramente toman el control. Este comportamiento es más notorio a medida que crece la dimensión.

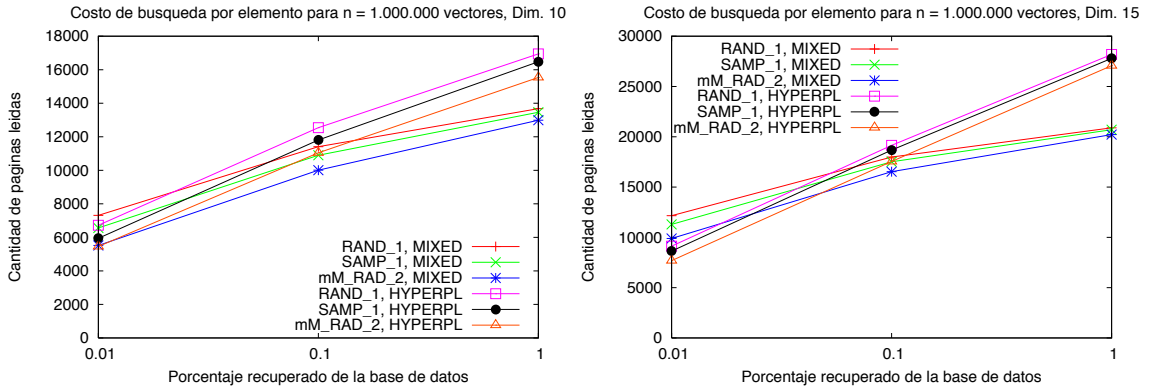
Otro efecto destacable es el crecimiento sublineal de los costos de inserción, para todas las estrategias. Por ejemplo, en dimensión 15, las estrategias MIXED comparan un objeto a ser insertado con alrededor del $1,1\,\%$ de la base de datos cuando ésta contiene 100.000 objetos, pero este porcentaje se decrementa a $0,8\,\%$ cuando se tiene 1.000.000 de objetos. Las estrategias HYPERPL también muestran



(a) Evaluaciones de Distancia en Construcción.



(b) Evaluaciones de Distancia en Búsquedas.

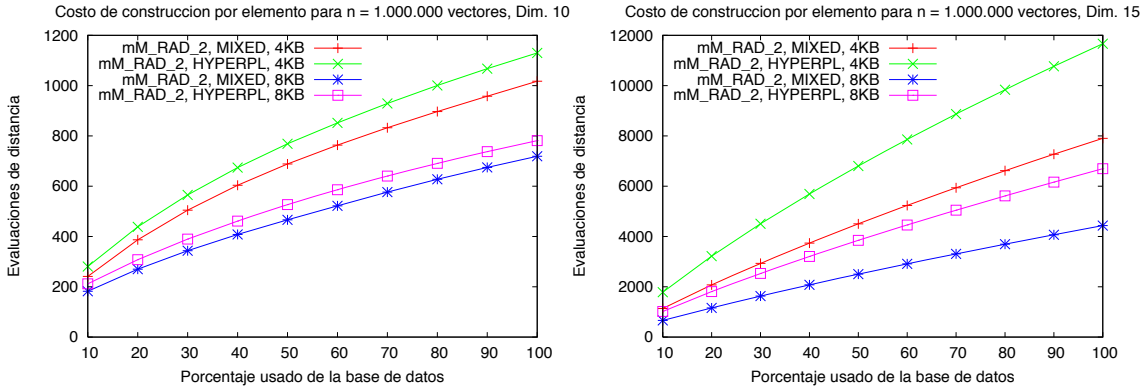


(c) Páginas Leídas en Búsquedas.

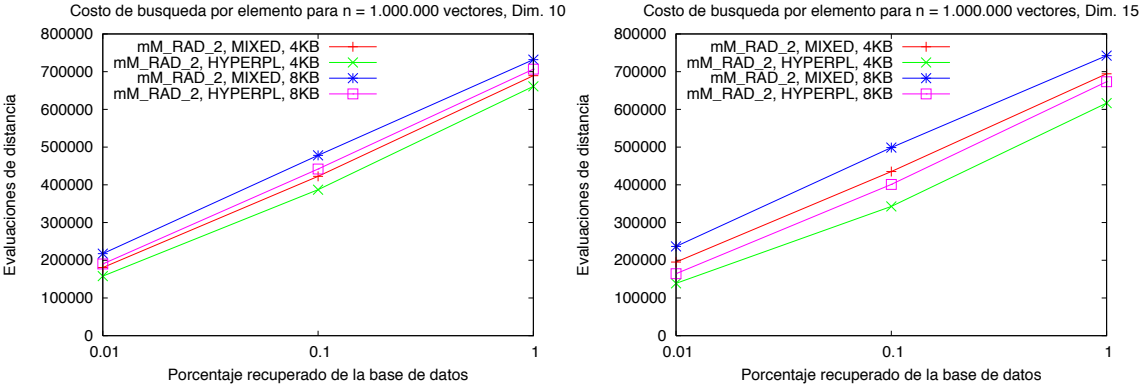
Figura 5.21: Costos de inserción y búsqueda para las mejores alternativas de *DSC* sobre el espacio de Vectores en dimensiones 10 (izquierda) y 15 (derecha).

una disminución desde alrededor del $1,9\,\%$ a menos que $1,2\,\%$. En dimensión 10, los porcentajes con las estrategias MIXED van desde $0,24\,\%$ sobre 100.000 objetos a menos que $0{,}1\,\%$ sobre 1.000.000 de elementos. Con la estrategia HYPERPL los porcentajes también decrecen, desde $0,28\,\%$ a $0,11\,\%$ aproximadamente.

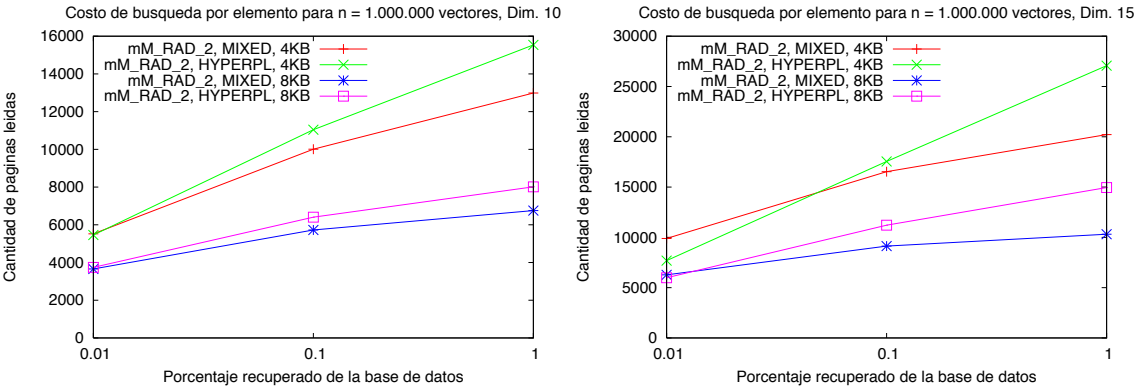
La Figura 5.22 muestra el efecto del tamaño de la página de disco, para las mejores estrategias. Se ha usado hasta ahora una página de disco de 4KB; ahora se considera el efecto de duplicarlo a 8KB. Las conclusiones no son tan simples como se podría esperar. Para los costos de construcción, dado que



(a) Evaluaciones de Distancia en Construcción.



(b) Evaluaciones de Distancia en Búsquedas.



(c) Páginas Leídas en Búsquedas.

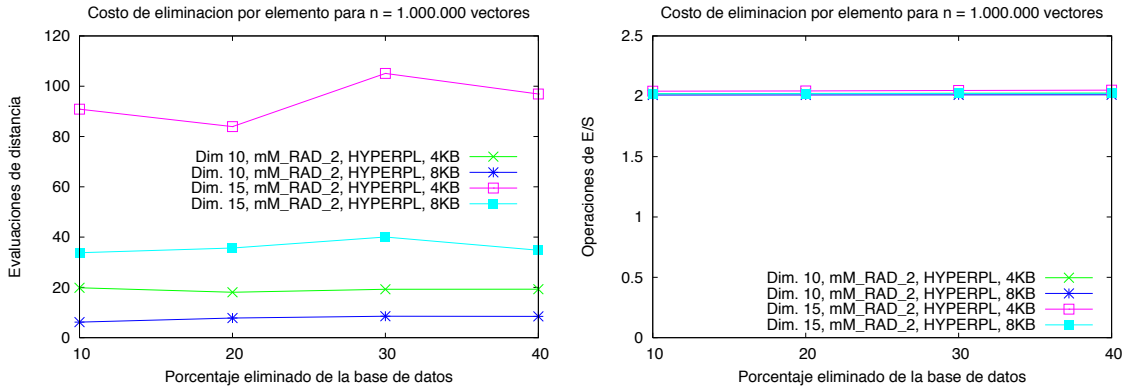
Figura 5.22: Costos de inserción y búsqueda para las mejores alternativas de *DSC* sobre el espacio de Vectores en dimensión 10 (izquierda) y 15 (derecha), considerando el tamaño de página de disco.

el *DSAT* en memoria principal posee la mitad de los centros con páginas de 8KB que con páginas de 4KB, el número de comparaciones decrece (menos que la mitad, porque el costo de búsqueda del *DSAT* es sublineal). Además, dado que en páginas de 8KB existe lugar para más elementos, la probabilidad de que una página se divida durante una inserción disminuye y así también los costos de inserción.

En las búsquedas, en cambio, dado que las páginas de 8KB tienen más elementos y aquéllos son todos comparados con la consulta, el número de cálculos de distancia se incrementa con el tamaño de página de disco, aunque no se duplica, porque a cambio se leen menos páginas, pero no la mitad. El

Dimensión	Alternativa	Ocupación de página		Total de páginas usadas	
		4KB	*8KB*	*4KB*	*8KB*
10	mM_RAD_2, MIXED	68 %	69 %	14.095	7.018
	mM_RAD_2, HYPERPL	54 %	56 %	17.755	8.553
15	mM_RAD_2, MIXED	67 %	67 %	21.272	10.548
	mM_RAD_2, HYPERPL	41 %	42 %	33.953	16.722

Tabla 5.2: Uso de espacio para el espacio de Vectores, con diferentes tamaños de página.



(a) Evaluaciones de Distancia.

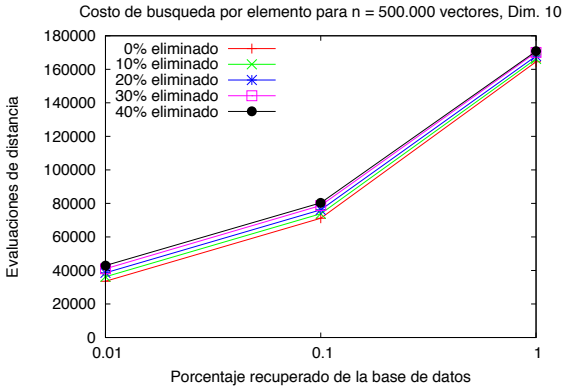
(b) Operaciones de E/S.

Figura 5.23: Costos de eliminación para la mejor variante de *DSC* sobre el espacio de Vectores, en dimensión 10 y 15.

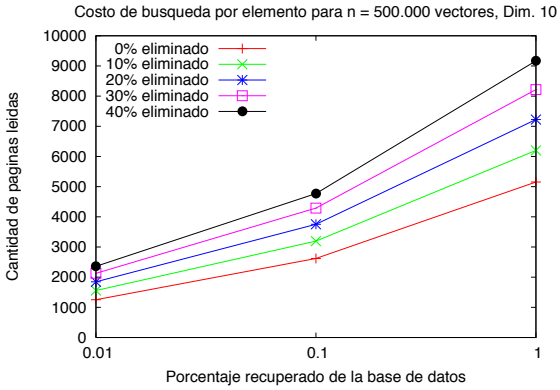
número de páginas de disco leídas, a su vez, es casi exactamente la mitad cuando se usan páginas más grandes de disco. Por lo tanto, usar páginas más grandes no necesariamente mejora el desempeño en las búsquedas en cálculos de distancia, pero sí lo hace en términos de operaciones de E/S. Por otra parte, la Tabla 5.2 muestra que, en ambas dimensiones 10 y 15, la ocupación de páginas y el uso de espacio no se ven afectados por el cambio en el tamaño de páginas de disco. Se muestra también que la estrategia MIXED siempre tiene un mucho mejor uso de espacio, como era de esperarse.

También se analizan los costos de eliminación para las mejores alternativas y diferentes tamaños de página. La Figura 5.23 muestra los costos de eliminación considerando evaluaciones de distancia y operaciones de E/S. Mientras que un tamaño de página mayor reduce a la mitad el número de cálculos de distancia, el número de operaciones de E/S permanece en 2 para ambas dimensiones. La razón para la cantidad reducida de evaluaciones de distancia es que el *DSAT* de memoria principal tiene la mitad del número de centros, y entonces las eliminaciones y las reinserciones de centros, cuando éstos cambian, cuestan cerca de la mitad.

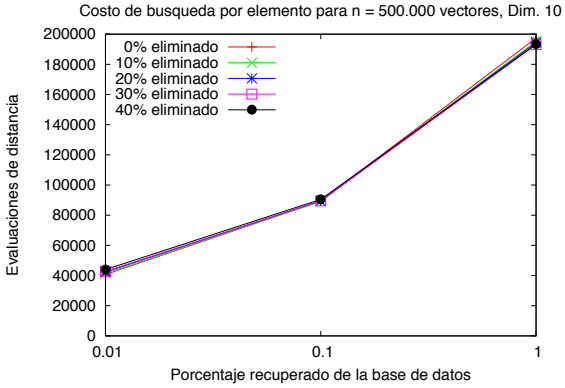
Finalmente, las Figuras 5.24 y 5.25 ilustran cómo se ven afectadas las búsquedas por rango a medida que la fracción de eliminaciones se incrementa, en dimensiones 10 y 15 respectivamente, para ambos tamaños de página de disco. Como se puede observar, mientras el número de evaluaciones de distancia se mantiene básicamente inalterado, la cantidad de operaciones de E/S se incrementa de manera sostenida a medida que el porcentaje eliminado de la base de datos crece hasta el 40 %. El tamaño de página de disco casi no tiene efecto en la manera en que los costos de búsqueda se degradan.



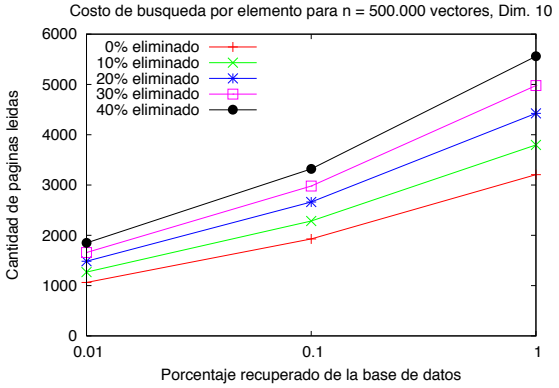
(a) Evaluaciones de Distancia, 4KB.



(b) Cantidad de Páginas Leídas, 4KB.



(c) Evaluaciones de Distancia, 8KB.



(d) Cantidad de Páginas Leídas, 8KB.

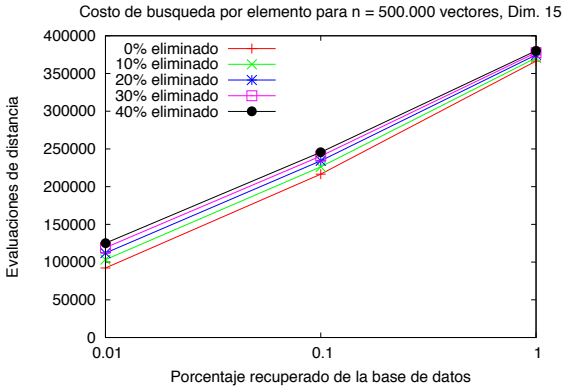
Figura 5.24: Comparación de los costos de búsqueda a medida que crece el número de eliminaciones, para las mejores alternativas de *DSC*, para Vectores en dimensión 10.

Un Espacio Real de Imágenes de Flickr

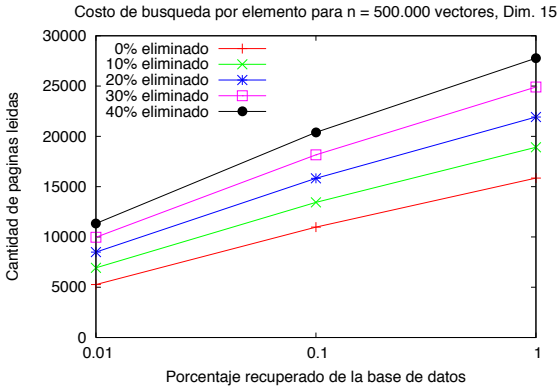
Se utiliza ahora, para evaluar a *DSC*, el conjunto de datos reales de *Flickr*, descripto en la Sección 2.10. Se ha comparado el efecto de varias políticas para *DSC*, pero se muestran aquí aquéllas que han tenido mejor desempeño. La Figura 5.26 (izquierda) ilustra los costos de construcción y búsqueda en este espacio. Los costos de construcción muestran el número de evaluaciones de distancia por elemento insertado; la cantidad de operaciones de E/S es siempre 2–3 y no se muestra. Para los costos de búsqueda, se muestran tanto la cantidad de evaluaciones de distancia como el número de páginas leídas. Como puede observarse, las estrategias de partición MIXED obtienen mejores costos en inserciones y, sorprendentemente, la estrategia de selección de centro de SAMP_1 con partición HYPERPL es la más costosa.

Las mejores alternativas son, en general, mM_RAD_2, RAND_1 y SAMP_1 para selección de centros y para particionado la distribución de hiperplano pura (HYPERPL) o la combinada con balanceo (MIXED). Como era de esperar, el particionado balanceado obtiene peores costos de búsqueda que las otras, debido a que prioriza la ocupación sobre la compacidad. La Tabla 5.3 muestra la ocupación de espacio en disco.

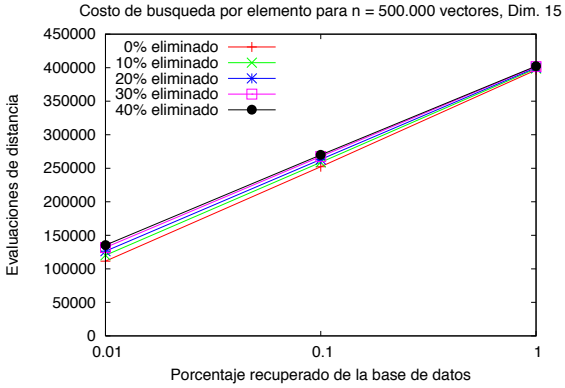
Para futuras comparaciones se elige la variante de mM_RAD_2 HYPERPL, porque tiene un buen



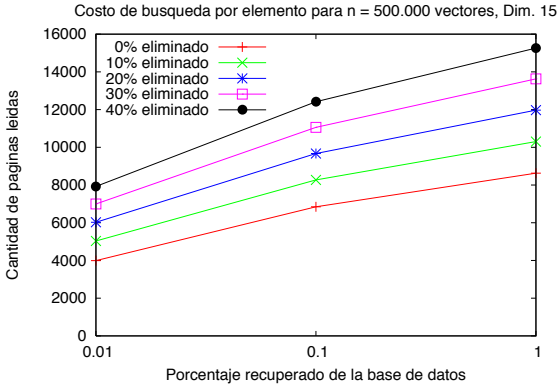
(a) Evaluaciones de distancia, 4KB.



(b) Cantidad de páginas leídas, 4KB.



(c) Evaluaciones de Distancia, 8KB.



(d) Cantidad de Páginas Leídas, 8KB.

Figura 5.25: Comparación de los costos de búsqueda a medida que crece el número de eliminaciones, para las mejores alternativas de *DSC*, para Vectores en dimensión 15.

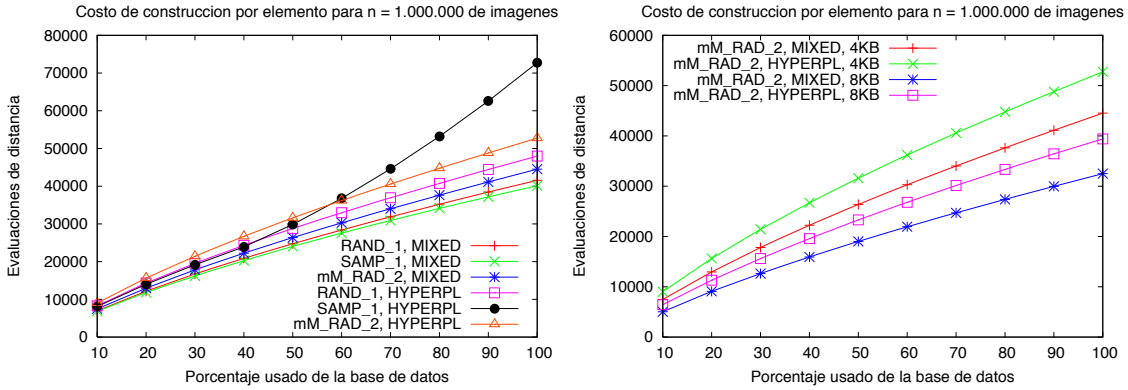
balance entre las evaluaciones de distancia y el número de páginas leídas en las búsquedas. Su costo de construcción no es generalmente el más bajo, pero es razonable.

También se consideran los costos de eliminación para la mejor alternativa, con diferentes tamaños de página. La Figura 5.27 ilustra los costos de eliminación considerando tanto evaluaciones de distancia como operaciones de E/S. Nuevamente, mientras un tamaño de página de disco más grande reduce a la mitad la cantidad de evaluaciones de distancia, el número de operaciones de E/S permanece en 2. La razón para el número reducido de evaluaciones de distancia es la misma que antes.

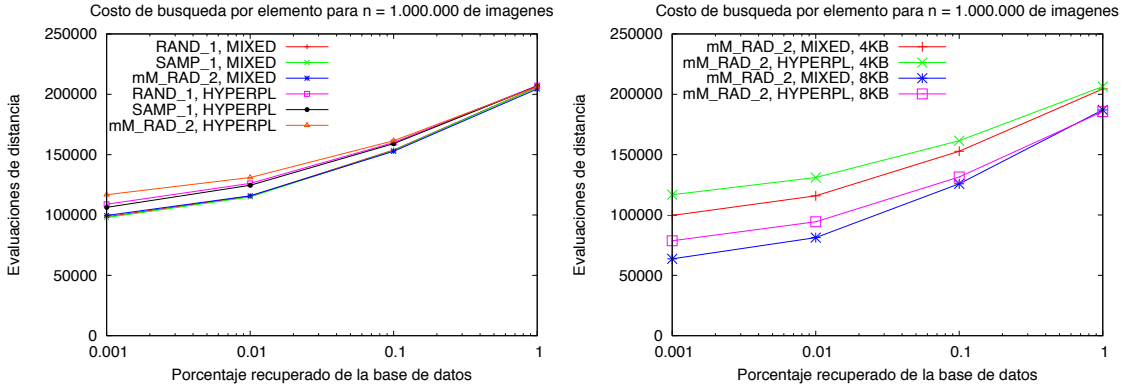
Finalmente, la Figura 5.28 ilustra cómo se ven afectadas las búsquedas por rango a medida que crece la fracción de elementos eliminados, para ambos tamaños de página de disco. Como puede observarse,

	Ocupación de página		**Total de páginas usadas**	
Alternativa	*4KB*	*8KB*	*4KB*	*8KB*
mM_RAD_2, MIXED	58 %	56 %	270.466	163.265
mM_RAD_2, HYPERPL	42 %	37 %	337.717	228.321

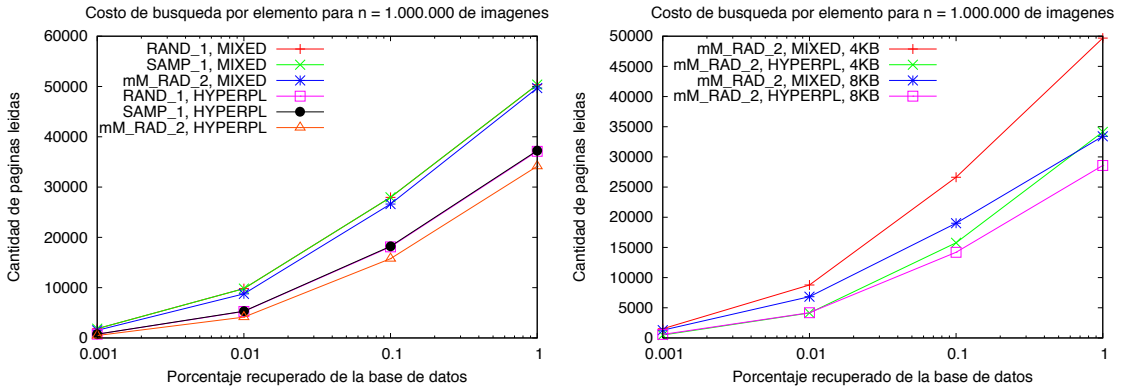
Tabla 5.3: Uso de espacio en disco para el conjunto de datos de *Flickr*, con diferentes tamaños de página.



(a) Evaluaciones de Distancia en Contrucción.



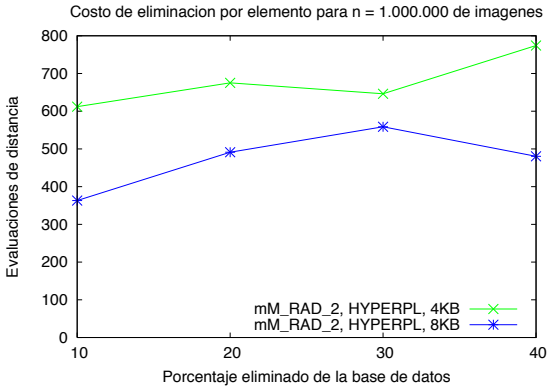
(b) Evaluaciones de Distancia en Búsquedas.



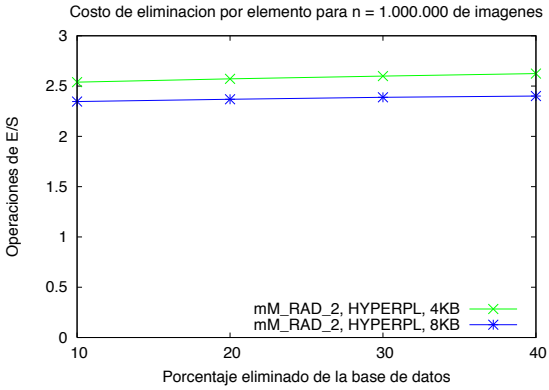
(c) Cantidad de Páginas Leídas en Búsquedas.

Figura 5.26: Costos de inserción y búsqueda para las mejores alternativas de *DSC* sobre el espacio de *Flickr*. A la izquierda, como una función de la política usada. A la derecha, como una función del tamaño de página de disco.

a diferencia de lo que ocurre en el espacio Vectores, el número de operaciones de E/S básicamente no se altera pero el número de evaluaciones de distancia se incrementa continuamente con el porcentaje de eliminaciones. Más aún, en este espacio el tamaño de página de disco afecta la manera en que los tiempos de búsqueda se degradan: el número de operaciones de E/S crece más notablemente con el tamaño de página de disco de 8KB que con la de 4KB. La razón es que, con un tamaño de página más grande, se toleran más páginas con pocos elementos sin removerlas. Por ejemplo, con un tamaño de

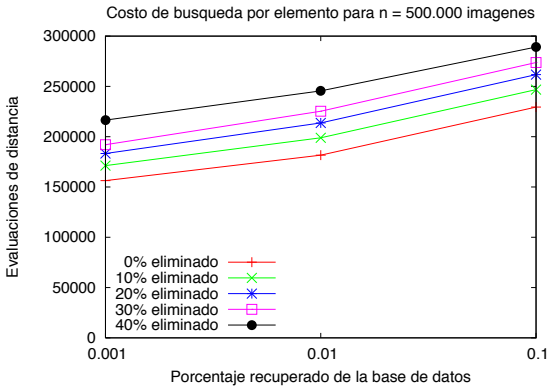


(a) Evaluaciones de Distancia.

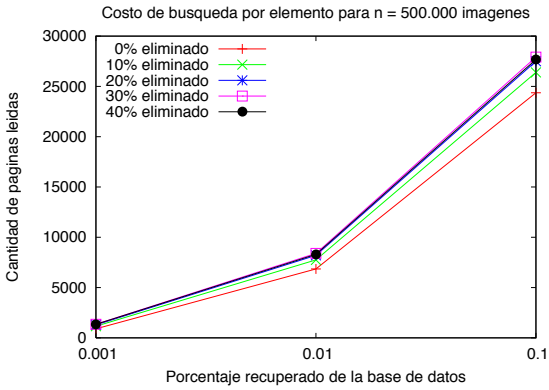


(b) Operaciones de E/S.

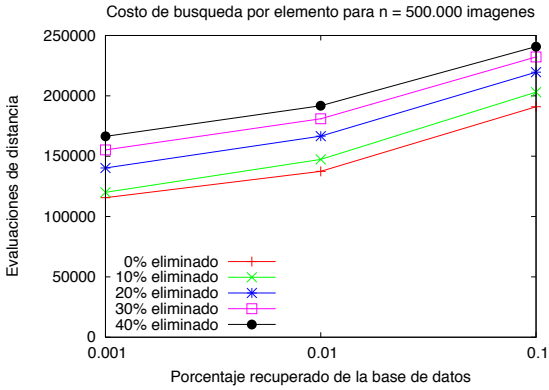
Figura 5.27: Costos de eliminación para la mejor variante de *DSC* sobre *Flickr*.



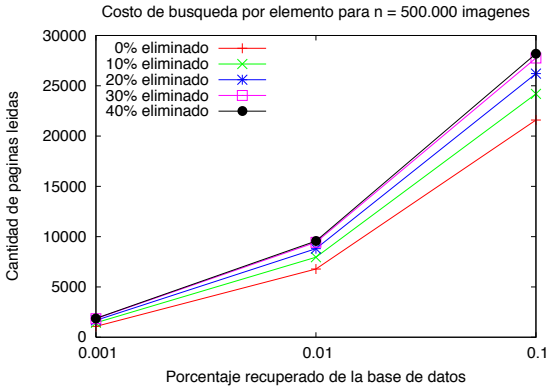
(a) Evaluaciones de Distancia, 4KB.



(b) Cantidad de Páginas Leídas, 4KB.



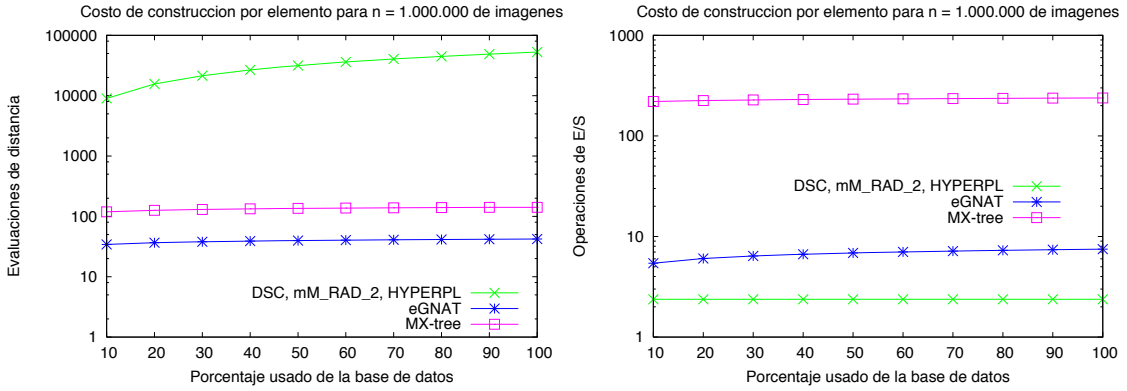
(c) Evaluaciones de Distancia, 8KB.



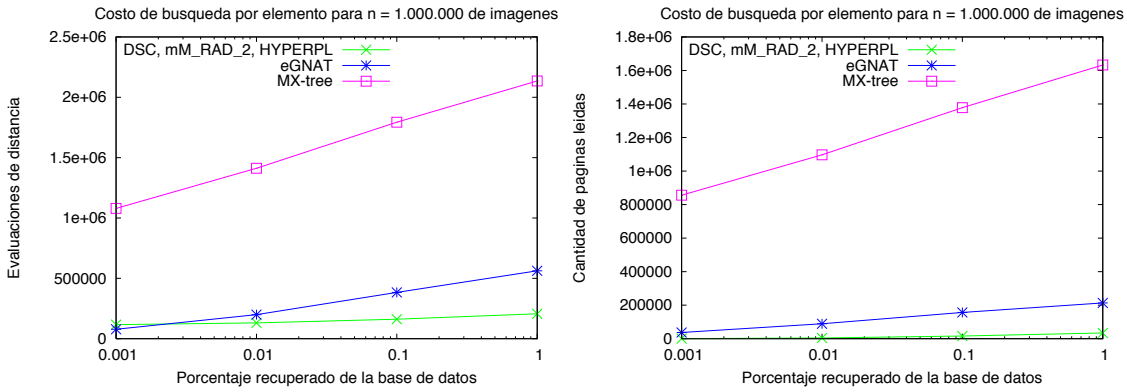
(d) Cantidad de Páginas Leídas, 8KB.

Figura 5.28: Comparación de los costos de búsqueda a medida que crece el número de eliminaciones, para las mejores alternativas de *DSC* en el espacio de *Flickr*.

página de 8KB la ocupación de página decrece de $36,6\,\%$ a $20,3\,\%$ luego de eliminar el $40\,\%$ de los elementos, porque no se remueven el $83\,\%$ de los clusters.



(a) Costos de Construcción.



(b) Costos de Búsqueda.

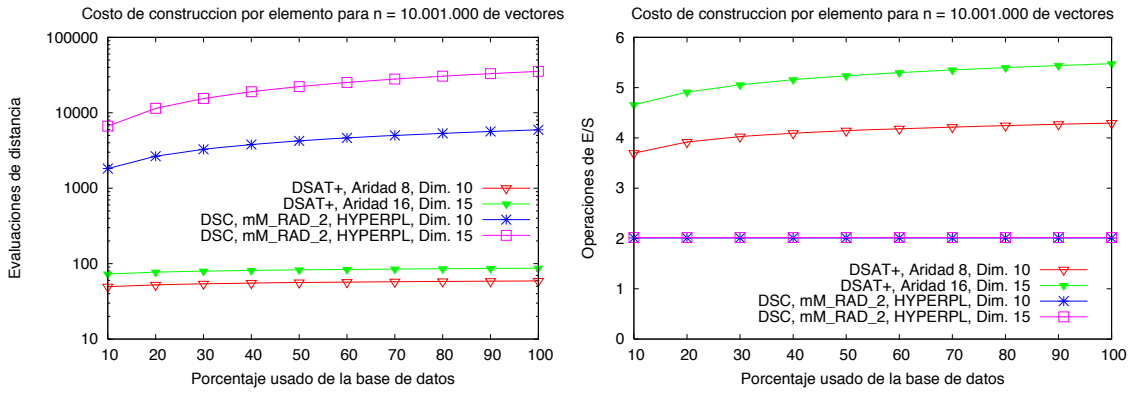
Figura 5.29: Comparación entre *DSC*, *eGNAT* y *MX–tree* sobre el espacio de *Flickr*.

5.7.4. Comparación con los Nuevos Índices

En esta sección se comparan los índices propuestos con otras alternativas que son más recientes que el *M–tree*, sobre el espacio de *Flickr* de un millón de imágenes: el *eGNAT* [NU11] y el *MX–tree* [JKF13] [6]. Para los experimentos se han considerado los parámetros de cada estructura como lo sugieren los autores para este espacio. Se utiliza en todos los índices un tamaño de página de 4KB.

La Figura 5.29 compara los costos de construcción y búsqueda de *DSC* con el *eGNAT* y el *MX–tree*, considerando tanto evaluaciones de distancia como operaciones de E/S. Como se puede observar, el *DSC* es significativamente más costoso en evaluaciones de distancia para inserciones. Sin embargo, *DSC* necesita menos de la mitad de las operaciones de E/S usadas por el *eGNAT* y aproximadamente el 10 % de las que necesita el *MX–tree*. Nuevamente, el trabajo extra realizado por *DSC* durante las inserciones reditúa en las búsquedas, dado que el *eGNAT* requiere más de 5 veces más operaciones de E/S y el doble de evaluaciones de distancia que las requeridas por *DSC*. El *MX–tree* es más costoso aún en las búsquedas, considerando ambas evaluaciones de distancia y operaciones de E/S. La razón posiblemente sea que el *MX–tree* usa copia de los elementos como objetos de enrutamiento y esto produce más operaciones adicionales de E/S. Para los experimentos de búsqueda, el índice del *MX–tree* ocupa 5.505.024 páginas de disco, mientras que el *eGNAT* necesita 393.216 y el *DSC* usa sólo 288.359 páginas.

[6] Los códigos del *MX-tree* se encuentran disponibles en `https://github.com/jsc0218/MxTree/`.



(a) Evaluaciones de Distancia.

(b) Operaciones de E/S.

Figura 5.30: Comparación de los costos de inserción del *DSAT+* y la mejor alternativa del *DSC* sobre los espacios de 10.000.000 de vectores de dimensiones 10 y 15.

5.7.5. Escalabilidad

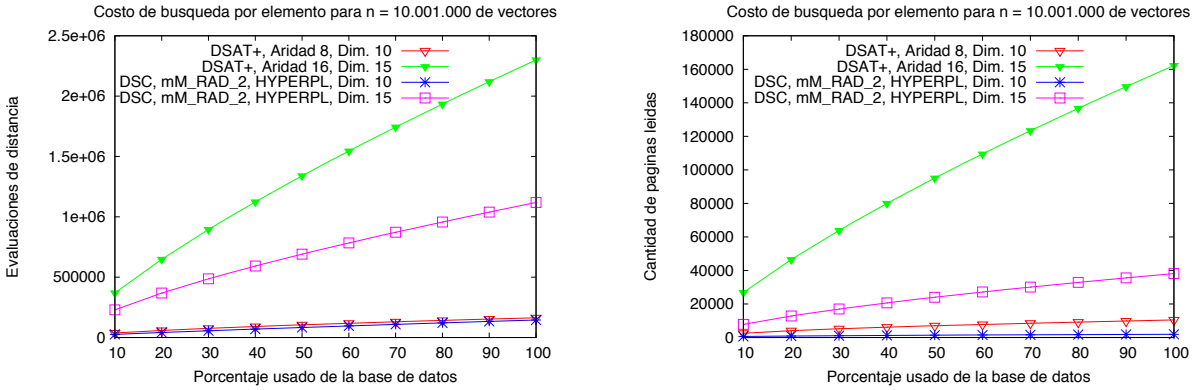
Finalmente, se consideran espacios significativamente más grandes para analizar cómo el desempeño de los mejores índices propuestos evoluciona con el tamaño de la base de datos. Para ello, se han generado espacios sintéticos de vectores uniformemente distribuidosen dimensiones 10 y 15, como en la Sección 5.7.3, pero ahora se han generado 10.000.000 de vectores. Se construye el índice para subconjuntos de la base de datos de tamaños crecientes, y se busca un conjunto fijo de 1.000 vectores aleatoriamente elegidos con radios $0,7$ y $0,65$, para dimensiones 10 y 15 respectivamente. Estas consultas recuperan a lo sumo 40 elementos en dimensión 10 con el conjunto de datos completo y 725 sobre el espacio en dimensión 15.

Se compara el *DSAT+* con sus mejores aridades de 8 para dimensión 10 y de 16 para dimensión 15, y el *DSC* con la política mM_RAD_2 HYPERPL, la cual tiene mejor desempeño en las búsquedas a pesar de ser levemente más costosa de construir. En este caso se ha utilizado un tamaño de página de 8KB para los experimentos.

La Figura 5.30 muestra los costos de construcción (notar la escala logarítmica). Mientras ambos costos de inserción en evaluaciones de distancia crecen sublinealmente, la velocidad de crecimiento del *DSAT+* es mucho más lenta, creciendo sólo desde 50 a 60 en dimensión 10 y de 73 a 87 en dimensión 15, cuando el tamaño de la base de datos crece de un millón a diez millones de elementos. Los costos de inserción de *DSC* también crecen sublinealmente, a pesar de que sean significativamente más altos: en dimensión 10 *DSC* se compara con un poco más de 1.800 elementos por inserción ($0,18\,\%$ de la base de datos) cuando se insertan un millón de elementos y termina examinando menos de 6.000 ($0,059\,\%$) cuando todos los diez millones de elementos se han insertado. De manera similar, en dimensión 15, se comienza examinando el $0,6\,\%$ de la base de datos y se termina examinando el $0,35\,\%$.

En términos de E/S, la situación es similar a la que se observó en conjuntos de datos más pequeños: el *DSC* requiere casi exactamente 2 operaciones de E/S por inserción, mientras que el costo del *DSAT+* crece muy lentamente, desde $3,7$ a $4,3$ y desde $4,7$ a $5,5$, en dimensiones 10 y 15 respectivamente.

Finalmente, la Figura 5.31 muestra cómo los costos de las búsquedas crecen, en términos de evaluaciones de distancia y de cantidad de páginas leídas. En las búsquedas, *DSC* claramente supera al *DSAT+* en ambos aspectos y en ambos espacios. Para el espacio de dimensión 15, por ejemplo, *DSC* usa cerca de la mitad de evaluaciones de distancia y un cuarto de las operaciones de E/S de las que necesita el *DSAT+*.



(a) Evaluaciones de Distancia.

(b) Cantidad de Páginas Leídas.

Figura 5.31: Comparación de los costos de búsqueda para el *DSAT+* y la mejor política del *DSC* sobre los espacios de 10.000.000 de vectores de dimensiones 10 y 15.

Más aún, aunque el *DSAT+* se desempeña mejor en espacios de menor dimensión, o en consultas más selectivas, el *DSC* se desempeña levemente mejor en ambos aspectos.

Notar que, a medida que crece la base de datos, los tiempos de búsqueda crecen sublinealmente en ambos índices y considerando ambas medidas de complejidad. Por ejemplo, el *DSC* para el espacio de dimensión 15 compara la consulta con el $23\,\%$ de la base de datos con un millón de elementos, pero con sólo el $11\,\%$ con diez millones de objetos. La fracción de páginas de disco leídas también decrece desde cerca de la mitad a menos del $20\,\%$. En dimensión 10, el *DSC* lee casi el $8\,\%$ de las páginas de disco con un millón de elementos, pero sólo menos del $2\,\%$ con diez millones de elementos. El porcentaje de objetos comparados con la consulta también decrece desde casi el $3\,\%$ a menos del $1,5\,\%$.

Cabe recordar que un espacio de dimensión 15 es considerado difícil de indexar, y los resultados muestran que los índices propuestos se desempeñarán mejor a medida que se manejen espacios más grandes. Esto plantea un contraste con los índices basados en pivotes, cuyo costo de búsqueda crece linealmente a medida que se incrementa el tamaño de la base de datos, a menos que el espacio para el índice crezca superlinealmente con el tamaño de la base de datos, lo cual se vuelve cada vez menos aceptable con conjuntos de datos más masivos.

5.8. Análisis Final de las Propuestas

El primer índice presentado se basa en el *Árbol de Aproximación Espacial Dinámico* (*DSAT*) [NR08], un índice dinámico para memoria principal que produce un balance atractivo entre uso de memoria, tiempo de actualización y desempeño en las búsquedas para espacios de mediana a alta dimensión. Se ha diseñado una variante de memoria secundaria, el *DSAT+*, que retiene el buen desempeño del *DSAT* en términos de evaluaciones de distancia, mientras necesita una cantidad moderada de operaciones de E/S. Esta estructura sólo soporta inserciones y búsquedas, lo cual es aceptable en algunos escenarios.

El segundo índice propuesto se basa en una estructura estática simple que tiene muy buen desempeño en espacios de alta dimensión, la *Lista de Clusters* (*LC*) [CN05]. Existe una versión dinámica de *LC*, llamada *Lista de Clusters Recursiva* (*RLC*) [Mam05], pero que tampoco ha sido diseñada para trabajar en memoria secundaria. Se ha diseñado una versión completamente dinámica de la *LC* que trabaja en memoria secundaria, a la cual se ha denominado *Lista de Clusters Dinámica* (*DLC*). Esta estructura

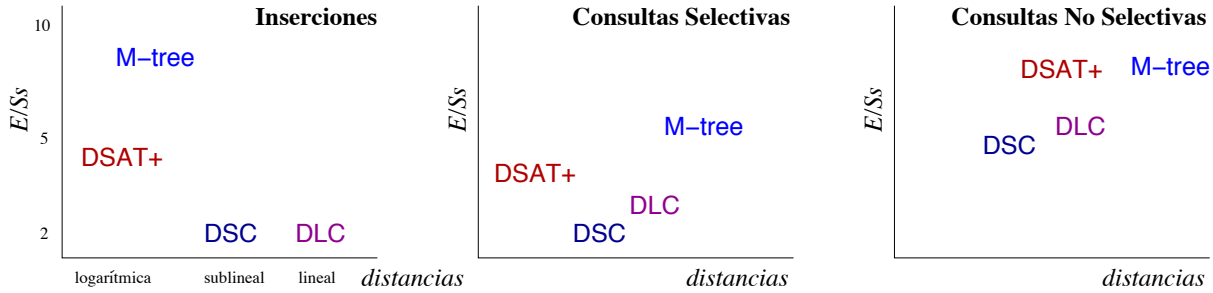


Figura 5.32: Una comparación gráfica gruesa de los costos de inserción y los costos de búsqueda de las diferentes estructuras.

requiere un muy bajo número de operaciones de E/S, pero para bases de datos grandes sufre de necesitar un número significativo de cálculos de distancia.

Por último, el tercer índice propuesto, denominado *Conjunto de Clusters Dinámico* (*DSC*), tiene por objetivo obtener lo mejor de ambos mundos. *DSC* combina una *DLC* con un *DSAT* en memoria principal para reducir el número de evaluaciones de distancia requeridas por la *DLC*. El resultado es también una estructura completamente dinámica para memoria secundaria, con un desempeño más balanceado en búsquedas y actualizaciones.

La comparación experimental muestra que las estructuras propuestas tiene una utilización de páginas de disco razonable y son competitivas con las alternativas representativas del estado del arte. Por ejemplo, la estructura de *DSAT+* es la más rápida de construir y requiere menos evaluaciones de distancia en consultas selectivas, superando al *M–tree* [CPZ97], la estructura alternativa más conocida, en todos los aspectos excepto en el número de operaciones de E/S en algunas búsquedas. La estructura *DSC* es más eficiente en consultas menos selectivas y en operaciones de E/S para construcción, donde también supera al *M–tree*, pero requiere más cálculos de distancia para construcción. La Figura 5.32 ilustra una comparación gráfica aproximada del desempeño de inserción y búsqueda de las diferentes estructuras propuestas, considerando tanto evaluaciones de distancia como operaciones de E/S.

Capítulo 6

Índices para Bases de Datos Métricas

Cuando se considera a las bases de datos métricas como un tipo de base de datos capaz de manejar datos no estructurados, surge la necesidad no sólo de poder responder a consultas clásicas de búsquedas por similitud, sino también a otro tipo de consultas más propias de bases de datos, como son las operaciones de join o ensamble. Sin embargo, en este modelo métrico donde sólo tiene sentido la similitud entre objetos, esta operación de join se establece considerando que dos elementos desde dos bases de datos se unen o combinan si cumplen con cierto criterio de similitud. Aunque los ensambles, o joins por similitud, se han convertido en operadores importantes de base de datos, hasta el momento han recibido poca atención de la comunidad del área. Los joins por similitud son útiles en varios escenarios, tales como detección de duplicados, emparejamiento de cadenas, integración de datos, soporte en minería de datos, entre otras [GCL+15, JS08, CSO+15].

Así, la existencia de servicios de Internet cada vez más complejos, los cuales utilizan fuentes de datos heterogéneas, requieren integración. Tales fuentes, como se ha mencionado, son generalmente no estructuradas; aunque en la mayoría de los servicios pretendidos se requieren datos estructurados. Por lo tanto, el principal desafío es proveer datos consistentes y sin errores, lo cual implica la "limpieza" de datos, que habitualmente se implementa con algún tipo de join o ensamble por similitud. A fin de ejecutar tales tareas, se especifican reglas de similitud para decidir si piezas de datos específicas pueden realmente considerarse el mismo dato o no. Sin embargo, cuando se trabaja con grandes volúmenes de datos, la limpieza de los datos puede tomar bastante tiempo, así el desempeño obtenido en tiempo de procesamiento es el factor más crítico que puede reducirse por medio de índices convenientes de búsqueda por similitud.

Sin embargo, a pesar de la gran atención que hasta el momento la primitiva del join ha recibido en las bases de datos tradicionales o aún en las bases de datos multidimensionales [BKS93, LR96, BBKK01, KP03], poco se ha hecho para las bases de datos métricas generales [DGSZ03b, DGZ03, JS08, Wan10, CSO+15, GCL+15, SCO+15, SPC13, SPCR15].

Hasta el momento se han presentado distintos índices diseñados para responder eficientemente consultas clásicas por similitud. En el Capítulo 4 cuando se consideran bases de datos conocidas de antemano y cuyo tamaño permite usar la memoria principal como almacenamiento y en el Capítulo 5 cuando las bases de datos son dinámicas y por el tamaño de los objetos o por la cantidad de los mismos, el índice debe almacenarse en memoria secundaria. En este capítulo se presenta un nuevo índice, la *Lista de Clusters Gemelos*, que permite avanzar en considerar a las bases de datos métricas como un modelo más maduro de bases de datos, porque logra resolver no sólo algunas variantes de la operaciones de join por similitud, sino también consultas por rango sobre la unión de dos bases de datos. Este nuevo índice ha sido presentado en [PR09].

6.1. Conceptos Previos

Como se ha descripto en la Sección 2.2.3, la operación de join por similitud entre dos bases de datos $\mathbb{A} \subseteq \mathbb{U}$ y $\mathbb{B} \subseteq \mathbb{U}$ es un subconjunto del producto cartesiano $\mathbb{A} \times \mathbb{B}$. Considerando que Φ es el criterio de similitud que deben satisfacer los pares de elementos de $\mathbb{A} \times \mathbb{B}$, el join se define formalmente como:

$$\mathbb{A} \bowtie_{\Phi} \mathbb{B} = \{(x, y)/(x \in \mathbb{A} \wedge y \in \mathbb{B}) \wedge \Phi(x, y)\} \subseteq \mathbb{A} \times \mathbb{B}$$

Claramente, la aproximación ingenua, o de fuerza bruta, para calcular un join por similitud entre $\mathbb{A}$ y $\mathbb{B}$ realizaría $|\mathbb{A}| \cdot |\mathbb{B}|$ cálculos de distancia; es decir, calcular la distancia entre todos los pares de elementos de $\mathbb{A} \times \mathbb{B}$ y seleccionar aquellos pares que satisfacen el criterio de similitud Φ. Este método es comúnmente llamado de *Iteración Anidada* o *Nested Loop* (*NL)*.

Recordando lo mencionado en la Sección 2.2.3, las variantes más comunes del join por similitud entre $\mathbb{A} \subseteq \mathbb{U}$ y $\mathbb{B} \subseteq \mathbb{U}$, son:

***Join por rango*:** dado un radio o umbral $r \geq 0$, se denota como $\mathbb{A} \bowtie_r \mathbb{B}$.

***Join de k-vecinos más cercanos*:** dado un valor $k \in \mathbb{N}$, se denota como $\mathbb{A} \bowtie_{kNN} \mathbb{B}$.

***Join de k pares de vecinos más cercanos*:** dado un valor $k \in \mathbb{N}$, se denota como $\mathbb{A} \bowtie_k \mathbb{B}$.

En espacios métricos, un enfoque natural para resolver este problema consiste en indexar uno o ambos conjuntos independientemente, usando alguno de los numerosos índices métricos disponibles que se encuentran mayormente compilados en [CNBYM01, HS03b, ZADB06, Sam05] y luego resolver consultas por rango para todos los elementos involucrados sobre los conjuntos indexados. De hecho, ésta es la estrategia propuesta en [DGSZ03b], donde los autores usan el *D–index* [DGSZ03a] para resolver auto-joins por similitud. Luego, presentan el *eD–index*, una extensión del *D–index* y estudian su aplicación a auto-joins por similitud [DGZ03].

Con respecto al join de k pares de vecinos más cercanos, en el caso de espacios vectoriales multidimensionales, existen algunas aproximaciones que se basan en la información de coordenadas para calcular resultados aproximados [LL00, AP02, AP05]. Esta operación es común en aplicaciones de la vida real tales como GIS, minería de datos y sistemas recomendadores [GCL$^+$15]. Sin embargo, no existían intentos previos de calcular el join $\mathbb{A} \bowtie_k \mathbb{B}$ en el contexto de espacios métricos. Recientemente, en [KTA11, GCL$^+$15], se presentan propuestas para el cálculo de consultas de k pares más cercanos. Por ejemplo, en [GCL$^+$15] se utilizan índices métricos dinámicos para memoria secundaria, junto con varias reglas de poda y barridos " best-first" (primero la mejor opción) y "depth-first" (primero en profundidad), para reducir costos de CPU y de E/S.

En [PCFN06], los autores dan algoritmos subcuadráticos para construir el grafo de k-vecinos más cercanos de un conjunto $\mathbb{A} \subseteq \mathbb{U}$, el cual puede verse como una variante de auto-join por similitud donde se buscan los k vecinos más cercanos de cada objeto en $\mathbb{A}$.

Sin embargo, también existen soluciones para resolver joins por similitud cuando se considera que ninguna de las bases de datos han sido indexadas. Una de estas soluciones es el *QuickJoin* (*QJ*) [JS08], inspirado en el conocido algoritmo de *quicksort*, que sólo permite resolver join por similitud por rango. La estrategia de *QJ* es dividir el espacio de búsqueda en regiones pequeñas, con el fin de reducir la complejidad de la aplicación de iteración anidada. En [FB15] se presenta una version mejorada de *QJ* que permite procesar el join de k vecinos más cercanos aproximados.

En [CSO$^+$15] se presenta una extensión de las operaciones de join por similitud a un conjunto más amplio de operadores binarios sobre bases de datos métricas, como por ejemplo la negación de

predicados. Además, dado que las respuestas a consultas de join suelen ser conjuntos muy grandes de pares de objetos y muchos de esos pares son muy similares entre sí, en [SCO+15] se introduce el concepto de *join por similitud diverso* como un operador de join por similitud que asegure un conjunto más pequeño y más diversificado de respuestas útiles. Estos desarrollos, entre otros, permiten acercar este nuevo modelo de bases de datos al nivel de madurez de las bases de datos tradicionales.

6.2. Lista de Clusters Gemelos

La idea básica de la *Lista de Clusters Gemelos* (*LTC* por su sigla en inglés: *List of Twin Clusters*) es que, para resolver el problema de join por similitud, se debe indexar a las bases de datos $\mathbb{A} \subseteq \mathbb{U}$ y $\mathbb{B} \subseteq \mathbb{U}$ conjuntamente en una única estructura de datos. Esto es porque, como se desean combinar objetos desde diferentes conjuntos que satisfagan algún criterio de similitud, las distancias que se calculen no deberían ser entre objetos del mismo conjunto. Además, como el join por similitud es una operación intrínsecamente costosa, si se calculan distancias entre elementos de la misma base de datos estas distancias no serán de utilidad directa para el resultado del join. Las distancias entre objetos de la misma base de datos pueden ser utilizadas para organizar los objetos dentro de ella y dicha organización puede aprovecharse para ahorrar cálculos de distancia en la operación de join con la otra base de datos. Sin embargo, no son distancias que el método ingenuo para el join calcularía. Por lo tanto, se considera como hipótesis de trabajo que el preproceso realizado para la construcción del índice debe involucrar sólo el cálculo de distancias que puedan ser de utilidad directa para la operación de join; es decir, sólo calcular distancias entre objetos que no pertenezcan a la misma base de datos.

La *LTC* es un índice métrico diseñado especialmente para resolver el join por similitud. Como su nombre sugiere, la *LTC* está basada en la *Lista de Clusters* (*LC*) [CN05], detallada en la Sección 3.6. La *Lista de Clusters*, como ya se ha mencionado, puede construirse usando particiones de tamaño fijo o particiones de radio fijo. A pesar de los resultados experimentales exhibidos en [CN05], se ha decidido usar los clusters de radio fijo. Cabe notar que, si se usara la opción de clusters de tamaño fijo, se obtendrían clusters de radios muy diferentes, especialmente en el caso en que los tamaños de los conjuntos de datos difieren considerablemente

Esencialmente, la *LTC* considera dos listas de clusters que se solapan, los que se denominan *clusters gemelos*. La Figura 6.1 ilustra esta situación. Los elementos de la base de datos $\mathbb{A}$ se han graficado como cuadrados y los de $\mathbb{B}$ como círculos. Los clusters con centros de $\mathbb{A}$ encierran círculos (desde $\mathbb{B}$) y los centros de $\mathbb{B}$ encierran cuadrados (desde $\mathbb{A}$). Se han resaltado en distintos colores los elementos que actúan como centros y aquéllos que integran ambos clusters.

Aunque R sea el radio fijo determinado para la construcción de la *LTC*, cada cluster determinará un *radio efectivo*, que es la distancia desde el centro al elemento más alejado que se encuentra en su bucket interno. Por lo tanto, cada cluster realmente será una triupla $(\textit{centro}, \textit{radio efectivo}, \textit{bucket interno})$. Por consiguiente, hay objetos en cada base de datos que actuarán como centros y otros objetos que integrarán los clusters. A los objetos internos de los clusters se los denominará aquí *objetos regulares*. Siguiendo la idea de la *LC*, se elige que cada objeto que es un centro no se incluya como elemento en su bucket gemelo.

De esta manera, cuando se resuelven consultas por rango, la mayoría de los objetos relevantes podrían pertenecer al cluster gemelo del objeto por el que se está consultando. Se han considerado estructuras adicionales para acelerar el proceso completo. Las estructuras de datos que integran la *LTC* son:

1. Dos listas de clusters gemelos CA y CB. Los centros de los clusters CA y CB pertenececen a la

base de datos $\mathbb{A}$ y $\mathbb{B}$, repectivamente y los objetos en sus clusters internos pertenecen a las bases de datos $\mathbb{B}$ y $\mathbb{A}$ respectivamente.

2. Una matriz D con las distancias calculadas entre todos los centros de la base de datos $\mathbb{A}$ a los centros de la base de datos $\mathbb{B}$ (considerando la propuesta de optimización a la *LC* planteada en Sección 4.6.1).

3. Cuatro arreglos $dAmax$, $dAmin$, $dBmax$ y $dBmin$ que almacenan el identificador de cluster y la distancia máxima o mínima para cada objeto desde una base de datos hacia todos los centros de cluster de la otra base de datos.

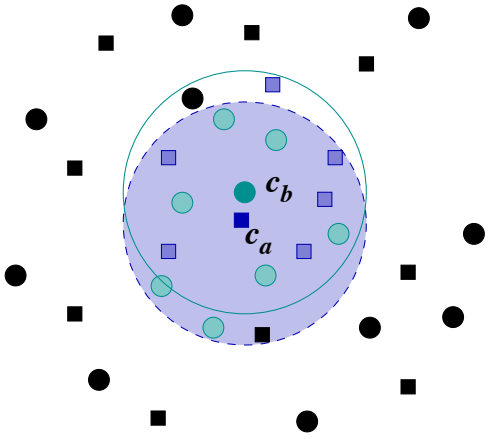


Figura 6.1: Clusters gemelos que se superponen entre ellos.

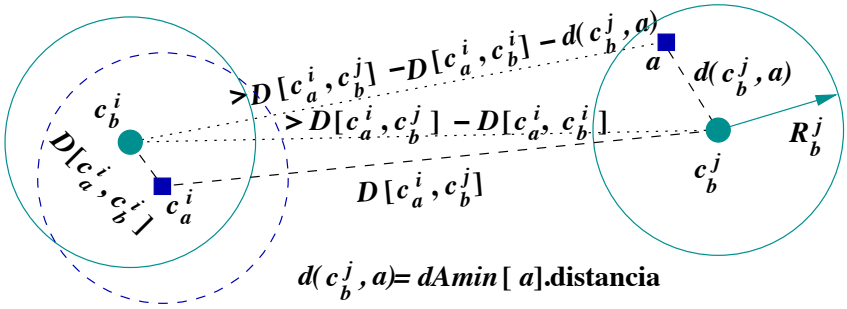


Figura 6.2: Resolución de la consulta para el centro de cluster c_a^i, usando las distancias almacenadas en *LTC*.

Se calculan las distintas variantes de join por similitud resolviendo consultas para los objetos desde una base de datos que recuperan objetos relevantes desde la otra. En la Sección 6.2.1, se describe cómo resolver las consultas por rango mediante las estructuras de la *LTC*. A continuación, en la Sección 6.2.2, se da el algoritmo de construcción de *LTC*. De aquí en adelante, r denota el radio del join por similitud y R el radio usado para indexar ambas bases de datos $\mathbb{A}$ y $\mathbb{B}$ conjuntamente con la *LTC*.

6.2.1. Resolviendo Consultas por Rango con *LTC*

Al resolver consultas por rango se tienen realmente tres clases de objetos: *centros de clusters*; *objetos regulares*, los objetos indexados en algún bucket interno; y *objetos no indexados*, los elementos que ni son centros ni son objetos regulares.

Algoritmo 26 Consulta por rango para los centros de clusters.

```
rqCenter (Entero i, Radio r)

1. If D[c_a^i, c_b^i] ≤ r Then Informar (c_a^i, c_b^i)     /* centro gemelo */
2. For b ∈ I_a^i Do    /* cluster propio */
3.     If dBmin[b].distancia ≤ r Then Informar (c_a^i, b)
4. For (c_a^j, R_a^j, I_a^j) ∈ CA,  j ← 1 To i − 1 Do    /* clusters previos */
5.     If |D[c_a^j, c_b^i] − D[c_a^i, c_b^i]| ≤ R_a^j + r Then
6.         For b ∈ I_a^j Do    /* bucket interno */
7.             If |D[c_a^j, c_b^i] − D[c_a^i, c_b^i]| − dBmin[b].distancia ≤ r ∧ d(c_a^i, b) ≤ r Then
8.                 Informar (c_a^i, b)
```

Para entender el concepto de los objetos no indexados, se debe considerar que cuando se construye la *LTC* no todos los objetos caen dentro de alguno de los clusters gemelos. Esto se debe a que la construcción de la *LTC* termina cuando una de las bases de datos se vacía, como se explica en la Sección 6.2.2. Así, todos los objetos restantes en el otro conjunto de datos se convierten en el conjunto de los *objetos no indexados*. De hecho, sólo se almacenan las distancias desde ellos hacia los centros más cercanos y más lejanos. Luego, se usan estas distancias a fin de tratar de evitar más cálculos de distancia cuando se resuelven joins por similitud y consultas por rango.

Resolviendo Centros de Clusters

Sea (c_a^i, R_a^i, I_a^i) el i-ésimo cluster de CA y c_b^i el centro gemelo de c_a^i. Luego de construir la *LTC*, cada centro $c_a^i \in \mathbb{A}$ se ha comparado con todos los objetos $b \in \mathbb{B}$ almacenados dentro de su propio bucket interno I_a^i y dentro de los buckets de los siguientes centros. Así, si el radio del join por similitud es menor o igual que el radio de construcción de la *LTC*; es decir, si $r \leq R$, a fin de resolver la consulta por rango para c_a^i, es necesario verificar si los siguientes objetos son relevantes:

(1) su centro gemelo c_b^i,

(2) los objetos dentro de su propio bucket interno y

(3) los objetos en los buckets de los clusters previos.

En otro caso, como $r > R$, se necesitarían revisar no sólo los objetos regulares sino también los centros de clusters de *todos* los clusters en la lista de CA, para terminar de calcular la consulta por rango para c_a^i.

Cuando se revisan los clusters previos, se puede evitar el cálculo de algunas distancias usando la *LTC* y la desigualdad triangular. La Figura 6.2 permite ilustrar esta situación. Se deben chequear los clusters previos $(c_a^j, R_a^j, I_a^j), j < i$, sólo si $|D[c_a^j, c_b^i] - D[c_a^i, c_b^i]| \leq R_a^j + r$; sino, el cluster (c_a^j, R_a^j, I_a^j) no es relevante para c_a^i. Dentro de un cluster relevante, se puede aún usar la desigualdad triangular para evitar una comparación directa. Dado un objeto b en el bucket I_a^j, si $|D[c_a^j, c_b^i] - D[c_a^i, c_b^i]| - dBmin[b]$.distancia $> r$, entonces b no es relevante. El Algoritmo 26 ilustra el proceso.

Algoritmo 27 Consulta por rango para objetos regulares.

rqRegular (Objeto a, Radio r)

1. $(c_b^i, d_1) \leftarrow dAmin[a]$ /* se obtiene el centro c_b^i y la distancia */
2. If $d_1 \leq r$ Then Informar (a, c_b^i) /* se chequea c_b^i, su cluster gemelo es (c_a^i, R_a^i, I_a^i) */
3. $d_2 \leftarrow D[c_a^i, c_b^i]$ /* d_2 es la distancia entre los centros gemelos */
4. For $b \in I_a^i$ Do /* verificar su cluster gemelo */
5. $d_3 \leftarrow dBmin[b]$.distancia, $d_s \leftarrow d_1 + d_2 + d_3$
6. If $d_s \leq r$ Then Informar (a, b)
7. Else If $2\,\text{máx}\{d_1, d_2, d_3\} - d_s \leq r \wedge d(a, b) \leq r$ Then Informar (a, b)

/* chequear otros clusters, d_1 no ha cambiado */

8. For $(c_a^j, R_a^j, I_a^j) \in CA$, $j \leftarrow 1$ To $|CA|, j \neq i$ Do
9. $d_2 \leftarrow D[c_a^j, c_b^i]$, $lb \leftarrow |d_2 - d_1|$, $d_3 \leftarrow D[c_a^j, c_b^j]$ /* c_b^j es el centro gemelo de c_a^j */
10. If $lb - d_3 \leq r$ Then /* primero verificar c_b^j */
11. $d_4 \leftarrow d(a, c_b^j), lb \leftarrow \text{máx}\{lb, |d_4 - d_3|\}$ /* actualizar lb si es posible */
12. If $d_4 \leq r$ Then Informar (a, c_b^j)
13. If $lb \leq R_a^j + r$ Then /* luego, chequear los objetos en el bucket I_a^j */
14. For $b \in I_a^j$ Do
15. If $lb - dBmin[b]$.distancia $\leq r \wedge d(a, b) \leq r$ Then Informar (a, b)

Resolviendo Objetos Regulares

Se considera que se está resolviendo la consulta por rango para un objeto regular $a \in \mathbb{A}$. Usando el arreglo $dAmin$, se determina a qué cluster (c_b^i, R_b^i, I_b^i) pertenece el objeto a. Así, se verifica si c_b^i es relevante. Sea (c_a^i, R_a^i, I_a^i) el cluster gemelo de c_b^i. Debido al algoritmo de construcción de la *LTC*, es probable que muchos objetos relevantes a a pertenezcan al bucket interno gemelo I_a^i; así, se verifican los objetos dentro de I_a^i. Luego, se verifican los otros clusters (c_a^j, R_a^j, I_a^j) en CA y sus respectivos centros gemelos. Cuando $r < R$, basta con chequear objetos regulares de los clusters previos, porque los centros de clusters están necesariamente más allá de r; en otro caso, se necesitarían verificar los centros de clusters.

Se usa $|D[c_a^j, c_b^i] - dAmin[a]$.distancia$|$ para acotar inferiormente la distancia entre a y c_a^j. Así, se limita inferiormente la distancia entre a y c_b^j, el centro gemelo de c_a^j con $|D[c_a^j, c_b^i] - dAmin[a]$.distancia$| - D[c_a^j, c_b^j]$, usando la desigualdad triangular generalizada [WS90b] (ver Lema 2 en la Sección 2.1). Más aún, si todavía es necesario computar $d(a, c_b^j)$, se usa este cálculo para (ojalá) mejorar el límite inferior de la distancia entre a y c_a^j. Finalmente, se verifica si el límite inferior corriente permite omitir al j-ésimo cluster; es decir, sólo se lo visita si el límite es menor o igual a $R_a^j + r$; sino, este cluster no contiene elementos relevantes. El Algoritmo 27 describe el proceso.

En el pseudo-código del Algoritmo 27 no se ha considerado si el cluster es previo o no, así cuando $r > R$ **rqRegular** no cambia.

Resolviendo Objetos no Indexados

En este caso es necesario chequear todos los clusters en CA y sus centros gemelos. Como en los algoritmos previos, para limitar inferiormente las distancias se usan las distancias entre los centros, las distancias al centro más cercano y al más lejano y la desigualdad triangular, evitando cuando sea posible la comparación directa. El Algoritmo 28 describe el proceso cuando los objetos no indexados provienen

Algoritmo 28 Consulta por rango para objetos regulares.

```
rqNonIndexedA (Objeto a, Radio r)

 1. (c_b^min, d^min) ← dAmin[a], (c_b^max, d^max) ← dAmax[a]
 2. For (c_a, R_a, I_a) ∈ CA Do    /* chequear todos los clusters */
 3. d_1 ← D[c_a, c_b^min], d_2 ← D[c_a, c_b^max], lb ← máx{|d_1 − d^min|, |d_2 − d^max|}
 4. d_3 ← D[c_a, c_b]    /* c_b es el centro gemelo de c_a */
 5. If lb − d_3 ≤ r Then    /* primero se chequea c_b */
 6.     d_4 ← d(a, c_b), lb ← máx{lb, |d_4 − d_3|}    /* y se actualiza lb si es posible */
 7.     If d_4 ≤ r Then Informar (a, c_b)
 8. If lb ≤ R_a + r Then    /* el cluster puede ser relevante */
 9.     For b ∈ I_a Do    /* luego, se chequea el bucket I_a */
10.         d_3 ← dBmin[b].distancia
11.         lb_1 ← 2 máx{d_1, d_3, d^min} − d_1 − d_3 − d^min
12.         lb_2 ← 2 máx{d_2, d_3, d^max} − d_2 − d_3 − d^max
13.         If máx{lb_1, lb_2} ≤ r ∧ d(a, b) ≤ r Then Informar (a, b)
```

de la base de datos $\mathbb{A}$, donde sólo se usan los arreglos $dAmax$ y $dAmin$. Si los objetos no indexados provienen de la base de datos $\mathbb{B}$ se usan los arreglos $dBmax$ y $dBmin$, y la situación es simétrica. Cuando $r > R$, **rqNonIndexedA/B** no cambia.

6.2.2. Construcción de *LTC*

Se ha asumido que la construcción del índice *LTC* es independiente del radio r del join por similitud por rango. Sea R el radio nominal de cada cluster en la *LTC*. El proceso de construcción de la *LTC* es como sigue.

Se comienza inicializando ambas listas de clusters CA y CB como vacías, y para cada objeto $a \in \mathbb{A}$ se inicializa $dA[a]$ en cero. Se usa el arreglo dA para elegir los centros de cluster para la *LTC*, desde el segundo al último cluster.

A continuación, se elige el primer centro c_a desde la base de datos $\mathbb{A}$ al azar y se agregan a su bucket interno I_a todos los elementos $b \in \mathbb{B}$ tales que $d(c_a, b) \leq R$. Luego, se usa el elemento $c_b \in I_a$ que minimiza la distancia a c_a como el centro del cluster gemelo de c_a, se elimina a c_b desde I_a, y se agregan a su bucket interno I_b todos los elementos $a \in \mathbb{A}$ tales que $d(a, c_b) \leq R$. La Figura 6.1 ilustra el concepto de los clusters gemelos. Para cada objeto $a \in \mathbb{A}$ se incrementan sus valores dA por $d(c_b, a)$; es decir, se actualiza su suma de distancias a los centros en $\mathbb{B}$. Cuando se han procesado las bases de datos $\mathbb{A}$ y $\mathbb{B}$ se agregan los clusters $(c_a, \text{máx}_{b \in I_a}\{d(c_a, b)\}, I_a)$ y $(c_b, \text{máx}_{a \in I_b}\{d(a, c_b)\}, I_b)$ de la forma (centro, radio efectivo, bucket) en la listas CA y CB, respectivamente. Ambos centros c_a y c_b, y los elementos insertados en sus buckets I_a y I_b se remueven de los conjuntos de datos $\mathbb{A}$ y $\mathbb{B}$. A partir de este punto, se elige como nuevo centro c_a el elemento que maximiza dA, pero se continúa usando el objeto $c_b \in I_a$ que minimiza la distancia a c_a como el centro del cluster gemelo de c_a. El proceso sigue de esta manera hasta que una de las bases de datos se vacíe.

Durante el proceso, se computa la distancia al centro de cluster más cercano y más lejano para todos los objetos. Por este motivo, se actualizan progresivamente $dAmin, dAmax, dBmin$ y $dBmax$ con las distancias mínimas y máximas conocidas hasta el momento.

Cabe destacar que para un objeto regular $a \in A$, y respectivamente $b \in B$, los arreglos $dAmin$ y

Algoritmo 29 Construcción de la *LTC*.

Construir-LTC (Base de datos A, Base de datos B, Radio R)

```
 1. CA ← ∅, CB ← ∅     /* las listas de clusters gemelos */
 2. For a ∈ A Do
 3.     dA[a] ← 0     /* suma de distancias a los centros en B */
        /* (centro más cercano y más lejano en B, distancia) */
 4.     dAmin[a] ← (Λ, ∞), dAmax[a] ← (Λ, 0)
 5. For b ∈ B Do
        /* (centro más cercano y más lejano en A, distancia) */
 6.     dBmin[b] ← (Λ, ∞), dBmax[b] ← (Λ, 0)
 7. While mín(|A|, |B|) > 0 Do
 8.     c_a ← argmax_{a∈A}{dA}, A ← A \ {c_a}
 9.     c_b ← Λ, d_{c,c} ← ∞, I_a ← ∅, I_b ← ∅
10.     For b ∈ B Do
11.         d_{c,b} ← d(c_a, b)
12.         If d_{c,b} ≤ R Then
13.             I_a ← I_a ∪ {(b, d_{c,b})}, B ← B \ {b}
14.             If d_{c,b} < d_{c,c} Then d_{c,c} ← d_{c,b}, c_b ← b
15.         If d_{c,b} < dBmin[b].distancia Then dBmin[b] ← (c_a, d_{c,b})
16.         If d_{c,b} > dBmax[b].distancia Then dBmax[b] ← (c_a, d_{c,b})
17.     I_a ← I_a \ {(c_b, d_{c,c})}     /* remover el centro c_b desde el bucket I_a */
18.     For a ∈ A Do
19.         d_{a,c} ← d(a, c_b)
20.         If d_{a,c} ≤ R Then I_b ← I_b ∪ {(a, d_{a,c})}, A ← A \ {a}
21.         Else dA[a] ← dA[a] + d_{a,c}
22.         If d_{a,c} < dAmin[a].distancia Then dAmin[a] ← (c_b, d_{a,c})
23.         If d_{a,c} > dAmax[a].distancia Then dAmax[a] ← (c_b, d_{a,c})
24.     CA ← CA ∪ {(c_a, máx_{b∈I_a}{d(c_a, b)}, I_a)}     /* (centro, radio efectivo, bucket) */
25.     CB ← CB ∪ {(c_b, máx_{a∈I_b}{d(a, c_b)}, I_b)}     /* (centro, radio efectivo, bucket) */
    /* sólo se conserva el arreglo dXmax para los objetos no indexados */
26. If |A| > 0 Then nonIndexed ← (''A'', A, dAmax)
27. Else nonIndexed ← (''B'', B, dBmax)
28. For c_a ∈ centers(CA), c_b ∈ centers(CB) Do D[c_a, c_b] ← d(c_a, c_b)
    /* las distancias d(c_a, c_b) ya se han calculado, entonces se las reusa */
29. Return (CA, CB, D, dAmin, dBmin, nonIndexed)
```

$dBmin$ almacenan su respectivo centro $c_b \in \mathbb{B}$ y $c_a \in A$ y la distancia desde el objeto a ese centro.

Notar también que se debe almacenar y mantener la matriz D, a fin de filtrar los elementos cuando se realizan joins por similitud y consultas por rango. Como estas distancias se calculan durante el proceso de construcción de *LTC*, se las puede reusar para llenar esta matriz.

Al final, se mantienen sólo las distancias máximas a los centros de clusters de los elementos no indexados. Así, si ellos provienen de la base de datos $\mathbb{A}$ se descarta el arreglo completo $dBmax$, y respectivamente si provienen de $\mathbb{B}$ se descarta el arreglo $dAmax$, y las distancias para los centros de clusters y objetos regulares desde $dAmax$ y respectivamente $dBmax$. Se hace esto en la triupla auxiliar $nonIndexed$ $(label, set, array)$. Si la base de datos $\mathbb{B}$ se vacía, entonces $nonIndexed \leftarrow$ ("A", $A, dAmax$) y se descarta el arreglo $dBmax$; en otro caso, se descarta el arreglo $dAmax$, así $nonIndexed \leftarrow$ ("B", $B, dBmax$).

El Algoritmo 29 describe el proceso de construcción.

Algoritmo 30 Cálculo del join por rango con *LTC*.

JoinRango-LTC `(Radio` r`)`

```
1. For c_a^i ∈ CA, I_b^i ∈ CB,  i ← 1 To |CA| Do
2.     rqCenter (i, r)    /* resolviendo el centro */
3.     For a ∈ I_b^i Do rqRegular (a, r)    /* resolviendo objetos regulares */
4. (label, set, array) ← nonIndexed
5. If label = ''A'' Then
6.     For a ∈ set Do rqNonIndexedA (a, r)
7. Else
8.     For b ∈ set Do rqNonIndexedB (b, r)
```

De acuerdo al análisis desarrollado en [CN05], los costos de construir la *LTC* es $O((\text{máx}\{|\mathbb{A}|, |\mathbb{B}|\})^2/p^*)$, donde p^* es el tamaño esperado de bucket.

6.3. Consultas con *LTC*

Como existe una simetría subyacente en el cálculo del join por rango (ver Sección 2.2.3) se puede considerar, sin pérdida de generalidad, que se están computando consultas por rango para elementos en $\mathbb{A}$, y que $|\mathbb{A}| \geq |\mathbb{B}|$. En otro caso se intercambian las bases de datos. En la Sección 6.3.1 se describe el algoritmo **RangeJoin** que permite calcular el join por rango $\mathbb{A} \bowtie_r \mathbb{B}$. Luego, en la Sección 6.3.2, se resuelven los join de k pares de vecinos más cercanos $\mathbb{A} \bowtie_k \mathbb{B}$ simulándolos como joins por rango con radios decrecientes. Finalmente, en Sección 6.3.3, se muestra cómo resolver consultas básicas por rango usando la *LTC*. Estas tres secciones suponen que dadas las bases de datos $\mathbb{A}$ y $\mathbb{B}$, y un radio R, ya se ha construido el índice *LTC* invocando **Construir-LTC** $(\mathbb{A}, \mathbb{B}, R)$.

6.3.1. Join por Rango

Dado un umbral r se calcula realmente el join por rango $\mathbb{A} \bowtie_r \mathbb{B}$ atravesando ambas listas CA y CB. Para los centros de clusters desde CA se invoca **rqCenter** (Algoritmo 26) y para los objetos regulares desde los buckets en CB se invoca **rqRegular** (Algoritmo 27). Finalmente, como aún no se han informado todos los pares coincidentes que consideran a los objetos no indexados, por medio de la triupla $nonIndexed$ se determina desde qué base de datos provienen los objetos no indexados y para todos aquellos elementos se invoca a **rqNonIndexedA** o **rqNonIndexedB**, de acuerdo a si provienen de $\mathbb{A}$ o de $\mathbb{B}$ (Algoritmo 28). El Algoritmo 30 ilustra el proceso completo.

6.3.2. Join de k Pares de Vecinos Más Cercanos

Como se ha mencionado en Sección 2.2.3, la idea básica de computar el join de k pares de vecinos más cercanos $\mathbb{A} \bowtie_k \mathbb{B}$ es como si fueran consultas de join por rango con radio decreciente. Por este motivo, se necesita una cola de prioridad auxiliar $heap$ que almacene triuplas de la forma (objeto, objeto, distancia), ordenadas en orden creciente de acuerdo a la distancia. Este $heap$ se inicializa con k triuplas $(\Lambda, \Lambda, \infty)$.

Al usar la estrategia de búsquedas por rango con radios decrecientes es conveniente disminuir los

Algoritmo 31 Rutina auxiliar para el cálculo del join de k pares más cercanos con *LTC*.

```
checkMax (Cola de Prioridad heap, Objeto a, Objeto b, Distancia dist)

1. If dist < heap.max().distancia Then
2.     heap.extractMax()
3.     heap.insert(a, b, dist)    /* se reduce el radio de búsqueda */
```

radios de búsqueda lo antes posible. Por lo tanto, antes de computar las consultas por rango es necesario reducir los radios de búsqueda. Para hacerlo, se puede poblar la cola de prioridad $heap$ con todas las distancias conocidas; es decir, con las distancias almacenadas en $dAmin$ y $dBmin$. Cada vez que se encuentra un par (a, b) de objetos que se encuentran más cerca que el par más lejano en el $heap$; es decir, distancia $d(a, b)$ menor que $heap.\mathbf{max}$.distancia, se elimina el par más lejano y se inserta en el $heap$ la triupla $(a, b, d(a, b))$. Si es necesario, es decir, cuando la máxima distancia almacenada en el $heap$ es mayor que el radio de construcción R de la *LTC*, se continúa la reducción del radio de búsqueda usando las distancias en D. Cabe destacar que todos los k pares de candidatos almacenados en el $heap$ se han encontrado "gratis", en términos de cálculos de distancia.

Luego de este preprocesamiento, se comienza realmente el cómputo de las consultas por rango para todos los objetos en $\mathbb{A}$. Se debe tener especial cuidado en evitar repetir pares de elementos que ya estén presentes en el $heap$. Una alternativa simple es fijar el radio inicial como $auxR \leftarrow heap.\mathbf{max}()$.distancia$+$ ε; considerando cualquier $\varepsilon > 0$; es decir, un radio levemente mayor que la distancia del par más alejado que actualmente está en el $heap$ con k triuplas $(\Lambda, \Lambda, auxR)$. Notar que esta alternativa sólo requiere tiempo de CPU, pero no cálculos de distancia. En el pseudo-código del Algoritmo 32 se usa esta alternativa por legibilidad, sin embargo en la implementación real se ha usado otra alternativa más eficiente [1]. Con esta alternativa, se pueden evitar verificaciones extras, simplificando el próximo proceso. Notar que en las siguientes etapas del procesamiento se encontrarán los mismos pares de objetos gratis, en términos de evaluaciones de distancia.

Como antes, el proceso considera resolver las consultas por rango de radio decreciente sobre los tres tipos de objetos de la base de datos: centros de clusters, objetos regulares y objetos no indexados. Así, se comienzan resolviendo consultas por rango para centros de clusters usando el radio $heap.\mathbf{max}()$.distancia. De nuevo, cada vez que se encuentre un par de objetos (a, b) tal que $d(a, b) <$ $heap.\mathbf{max}()$.distancia se modifica el $heap$, extrayendo su máximo y luego insertando la triupla $(a, b, d(a, b))$. Por lo tanto, se está resolviendo una consulta por rango de radio decreciente. Se continúa con el cálculo para objetos regulares y finalmente para aquellos objetos no indexados. Cuando el cálculo finaliza, $heap$ contiene los k pares del resultado. El Algoritmo 32 ilustra este proceso. Como los centros de clusters usualmente requieren menos trabajo para calcular sus consultas por rango que los objetos regulares o los no indexados, se comienza el cálculo del join de k pares más cercanos con ellos y además principalmente porque son los que con mayor probabilidad ayuden a reducir el radio de búsqueda más rápido.

Notar que, luego de revisar todas las distancias almacenadas en el índice *LTC*, es posible que el radio corriente del join sea más grande que el radio de indexación R. En este caso, se tienen que procesar los centros de clusters y los objetos regulares usando las variantes desarrolladas para este propósito particular.

Cabe destacar que, según [GCL$^+$15], a la fecha las únicas propuestas existentes para resolver este

[1] Por ejemplo, a fin de evitar la constante ε basta con reemplazar el signo "$<$" por el signo "$\leq$" en la línea 1 del procedimiento auxiliar **checkMax** (Algoritmo 31). Aunque esta modificación funciona bien con funciones de distancia contínuas, falla en descartar suficientes valores en el caso de distancias discretas.

Algoritmo 32 Cálculo del join de k pares más cercanos con *LTC*.

JoinkParesMasCercanos `(Entero` k`)`

```
 1. ColaPrioridad heap ← ∅    /* ordenada por distancia creciente (tercera componente) */
 2. For i ← 1 To k Do heap.insert(Λ, Λ, ∞)
    /* usando distancias en dAmin, dBmin, y D para reducir el radio de búsqueda */
 3. For a ∈ A Do (c_b, dist) ← dAmin[a], checkMax(heap, a, c_b, dist)
 4. For b ∈ B Do (c_a, dist) ← dBmin[b], checkMax(heap, c_a, b, dist)
 5. If heap.max().distancia > R Then
 6.     For c_a^i ∈ CA, c_b^j ∈ CB,  i, j ← 1 To |CA| Do
 7.         checkMax(heap, c_a^i, c_b^j, D[c_a^i, c_b^j])
 8. auxR ← heap.max().distancia + ε    /* estableciendo el radio inicial de búsqueda */
 9. heap ← ∅
10. For i ← 1 To k Do heap.insert(Λ, Λ, auxR)    /* se reinicia el heap */
11. For c_a^i ← CA,  i ← 1 To |CA| Do    /* revisar los centros */
12.     foundSet ← rqCenter(i, heap.max().distancia)
13.     For (b, dist) ∈ foundSet Do checkMax(heap, c_a^i, b, dist)
14. For I_b^i ← CB,  i ← 1 To |CB| Do    /* revisar los objetos regulares */
15.     For a ∈ I_b^i Do
16.         foundSet ← rqRegular(a, heap.max().distancia)
17.         For (b, dist) ∈ foundSet Do checkMax(heap, a, b, dist)
18. (label, set, array) ← nonIndexed    /* revisar los objetos no indexados */
19. If label = ''A''    /* los objetos no indexados son de A */
20.     For a ∈ set
21.         foundSet ← rqNonIndexedA(a, heap.max().distancia)
22.         For (b, dist) ∈ foundSet Do checkMax(heap, a, b, dist)
23. Else    /* los objetos no indexados son de B */
24.      For b ∈ set Do
25.         foundSet ← rqNonIndexedB(b, heap.max().distancia)
26.         For (a, dist) ∈ foundSet Do checkMax(heap, a, b, dist)
```

tipo de join sobre espacios métricos generales son: este algoritmo de *LTC* (que ha sido la primera propuesta de este tipo), la presentada en [KTA11] que aplica directamente una técnica divide y vencerás (llamada *Multi-Particionado Adaptativo*) y la que se propone justamente en [GCL$^+$15], que usa una extensión del *M–tree* llamada *COM–tree*.

6.3.3. Consultas por Rango

A pesar de que *LTC* indexa conjuntamente a las bases de datos $\mathbb{A}$ y $\mathbb{B}$, las listas CA y CB se pueden ver como una lista clásica de clusters [CN05] para las bases de datos $\mathbb{A}$ y $\mathbb{B}$, respectivamente. Por lo tanto, es posible derivar un algoritmo para búsquedas por rango tradicionales basado en la *LTC*, el cual atraviese ambas listas simultáneamente. Notar que, en este caso, no se puede usar directamente el criterio de parada de las consultas por rango de la *Lista de Clusters* (*LC*); es decir, cuando la bola de la consulta (q, r) esté estrictamente contenida por el cluster corriente. En cambio, se agregan variables lógicas o booleanas para controlar si es necesario buscar las listas.

Además, usando las distancias en la matriz D se pueden obtener cotas inferiores y superiores de las distancias entre la consulta q y los centros. Para hacerlo, durante el cálculo, se mantienen dos conjuntos de distancias DA y DB las cuales almacenan las distancias calculadas desde la consulta q a los centros en las listas CA y CB, respectivamente. Por lo tanto, usando las distancias calculadas y almacenadas en DB, la distancia $d(q, c_a)$ está limitada inferiormente por $\max_{(c_b, d(q,c_b)) \in DB}\{|d(q, c_b) - D[c_a, c_b]|\}$ y superiormente por $\min_{(c_b, d(q,c_b)) \in DB}\{d(q, c_b) + D[c_a, c_b]\}$. Simétricamente, se puede limitar inferiormente y superiormente la distancia $d(q, c_b)$. De esta manera, se modifica el algoritmo de consulta por rango de *LTC* de acuerdo a estos límites. Finalmente, se necesitan verificar los objetos no indexados. Por este motivo, se usa nuevamente la triupla $nonIndexed\ (label, set, array)$. Así, para los objetos (no indexados) almacenados en el conjunto set se calculan los límites inferiores de las distancias desde ellos hacia la consulta q usando los arreglos $dAmin$ o $dBmin$, de acuerdo a cuál sea el conjunto de datos de donde provienen; es decir, de acuerdo al valor de $label$. Además, se usa $array$ para mejorar los límites inferiores. Se debe recordar que $array$ almacena uno de los arreglos o $dAmax$ o $dBmax$, dependiendo de $label$. El Algoritmo 33 describe completamente el proceso.

Cabe destacar que, aunque el índice *LTC* fue diseñado especialmente para responder a consultas de join por similitud entre dos bases de datos $\mathbb{A} \subseteq \mathbb{U}$ y $\mathbb{B} \subseteq \mathbb{U}$, como indexa conjuntamente a ambas bases de datos, permite también responder a consultas por rango sobre $\mathbb{A} \cup \mathbb{B} \subseteq \mathbb{U}$.

6.4. Evaluación Experimental

Para evaluar experimentalmente a *LTC* se han seleccionado cuatro pares de bases de datos del mundo real desde tres clases distintas de espacios métricos, tales como: imágenes de rostros, palabras y documentos. En aplicaciones de Recuperación de Información [BYRN11] resultan de interés los últimos dos tipos de espacios métricos considerados: palabras y documentos. A continuación se decriben en detalle los conjuntos de datos utilizados:

Imágenes de rostros: un conjunto de 1.016 vectores de características, con 761 coordenadas, desde una base de datos de imágenes de rostros. En este espacio cualquier función de distancia de forma cuadrática se podría utilizar, así se ha elegido a la distancia Euclidiana como la alternativa significativa más simple. El conjunto completo posee cuatro imágenes del rostro de 254 personas y se lo ha dividido en dos subconjuntos: uno de ellos con tres imágenes del rostro por persona, por

Algoritmo 33 Uso de *LTC* para calcular consultas por rango.

ConsultaRango (Objeto q, Radio r)

```
 1. stop_A ← False, stop_B ← False
 2. For (c_a, R_a, I_a) ∈ CA, (c_b, R_b, I_b) ∈ CB,  i ← 1 To |CA| Do
 3.     If stop_A ∧ stop_B Then Break
 4.     lbA ← máx_{(c_b,d(q,c_b))∈DB}{|d(q,c_b) − D[c_a,c_b]|}   /* limitando inferiormente a d(q,c_a) */
 5.     ubA ← mín_{(c_b,d(q,c_b))∈DB}{d(q,c_b) + D[c_a,c_b]}    /* limitando superiormente a d(q,c_a) */
 6.     lbB ← máx_{(c_a,d(q,c_a))∈DA}{|d(q,c_a) − D[c_a,c_b]|}   /* limitando inferiormente a d(q,c_b) */
 7.     ubB ← mín_{(c_a,d(q,c_a))∈DA}{d(q,c_a) + D[c_a,c_b]}    /* limitando superiormente a d(q,c_b) */
 8.     If stop_A = False ∧ lbA ≤ R_a + r Then
 9.         dqa ← d(q,c_a), DA ← DA ∪ {(c_a,dqa)}, ubA ← dqa
10.         If dqa ≤ r Then Informar c_a
11.         If dqa ≤ R_a + r Then
12.             For b ∈ I_a Do
13.                 If d(q,b) ≤ r Then Informar b
14.     If stop_B = False ∧ lbB ≤ R_b + r Then
15.         dqb ← d(q,c_b), DB ← DB ∪ {(c_b,dqb)}, ubB ← dqb
16.         If dqb ≤ r Then Informar c_b
17.         If dqb ≤ R_b + r Then
18.             For a ∈ I_b Do
19.                 If d(q,a) ≤ r Then Informar a
20.     If ubA ≤ R_a − r Then stop_A ← True
21.     If ubB ≤ R_b − r Then stop_B ← True
     /* procesamiento de objetos no indexados */
22. (label, set, array) ← nonIndexed
23. For o ∈ set Do   /* revisión de los objetos no indexados */
24.     lb ← 0   /* se intenta limitar inferiormente d(q,o) con las distancias almacenadas en DA y DB */
25.     If label = ''A'' Then
26.         (c_b^min, d^min) ← dAmin[o], (c_b^max, d^max) ← array[o]
27.         If (c_b^min, dqb) ∈ DB Then lb ← máx{lb, |dqb − d^min|}
28.         If (c_b^max, dqb) ∈ DB Then lb ← máx{lb, |dqb − d^max|}
29.     Else
30.         (c_a^min, d^min) ← dBmin[o], (c_a^max, d^max) ← array[o]
31.         If (c_a^min, dqa) ∈ DA Then lb ← máx{lb, |dqa − d^min|}
32.         If (c_a^max, dqa) ∈ DA Then lb ← máx{lb, |dqa − d^max|}
33. If (lb ≤ r) ∧ d(q,o) ≤ r Then Informar o
```

brevedad se lo nombrará como FACES762 porque posee 762 imágenes de rostros; y el otro con la cuarta imagen de cada persona, el cual se nombrará como FACES254.

Palabras: un diccionario de palabras, donde la distancia es la distancia de edición [Lev65, Lev66]. Para este espacio métrico se han considerado dos pares de conjuntos de datos: el diccionario de 69.069 palabras en *Inglés* con un diccionario de 89.061 palabras en *Español* y el mismo conjunto de palabras en Inglés con un subconjunto de 494.048 términos de vocabulario desde una colección de documentos, denominado *Vocabulario*.

Documentos: un conjunto de 2.957 documentos, cada uno de los cuales es de aproximadamente 500 KB, obtenido al dividir los 1.265 documentos originales de la colección de TREC-3 [Har95], de manera tal que los subdocumentos obtenidos desde el mismo documento original se solapen cerca de un 50 %. Se han sintetizado los vectores que representan a los subdocumentos usando códigos provistos desde Biblioteca Métrica de SISAP [FNC07], disponibles desde `www.sisap.org`. Como es habitual, en este espacio se usa la distancia coseno [SWY75] para comparar dos documentos. Se ha dividido el conjunto de datos en dos subconjuntos: uno de ellos con 1.846 documentos, al que brevemente se nombrará como DOCS1846; y el otro de 1.111 documentos, al que se nombrará como DOCS1111.

Como ya se ha mencionado, se consideran aquí los dos tipos de joins por similitud: joins por rango $\mathbb{A} \bowtie_r \mathbb{B}$ y joins de k-pares más cercanos $\mathbb{A} \bowtie_k \mathbb{B}$. En los experimentos de join, se construye el índice con todos los objetos considerados para cada base de datos. Todos los resultados se han promediado entre 10 construcciones de índice usando diferentes permutaciones de las bases de datos. En los experimentos de consultas por rango, se ha construido el índice con el 90 % de los objetos desde ambas bases de datos y se usa el 10 % restante para las consultas por rango. Los resultados mostrados corresponden al promedio calculado sobre todas esas consultas.

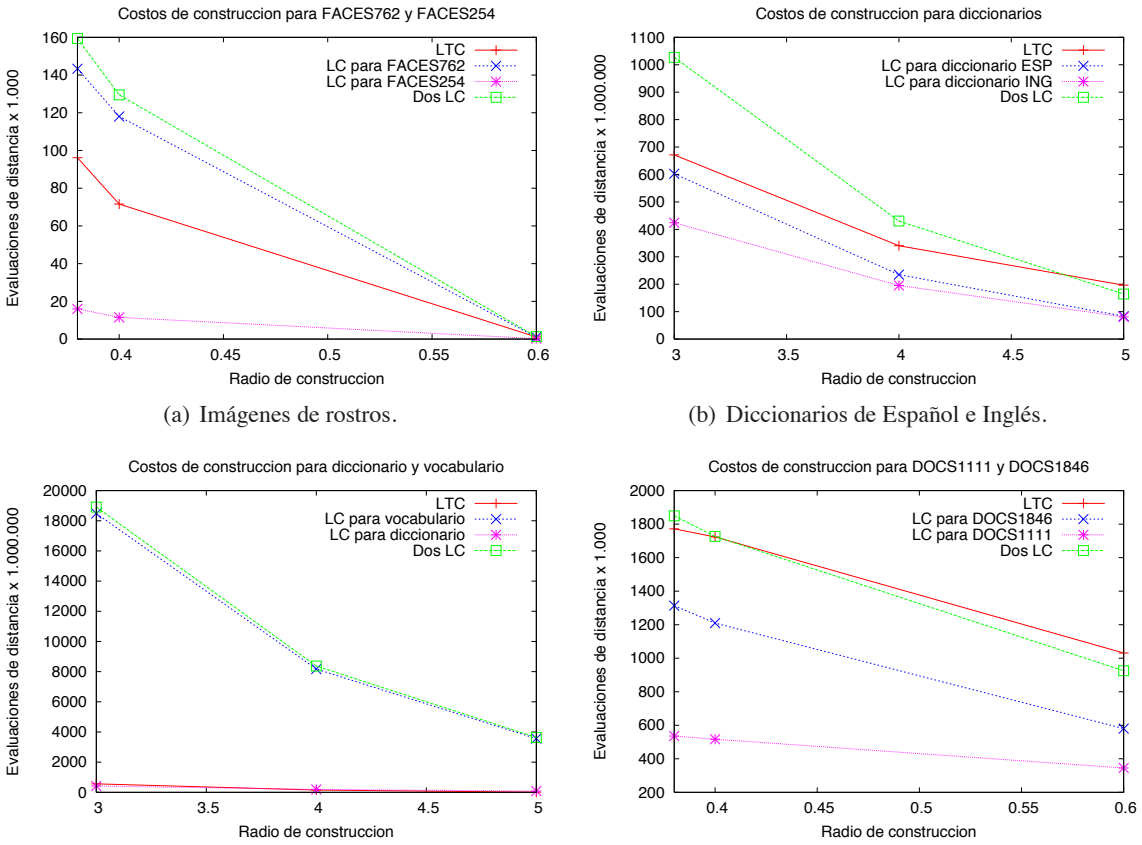
6.4.1. Construcción de *LTC*

Se comienza la evaluación experimental verificando que el costo de construir el índice *LTC* para cada par de bases de datos sea similar al necesario para indexar la base de datos más grande con una *LC* básica. Se han evaluado varios valores para el radio de construcción R. Para las imágenes de rostros, se muestran los resultados de construcción cuando se indexan con radios R de $0,38$, $0,40$, $0,60$ y $0,80$; para las palabras, radio desde 3 hasta 6; y para los documentos, radios $0,38$, $0,40$ y $0,60$. La Figura 6.3 ilustra los resultados obtenidos en la construcción de la *LTC*, para las bases de datos de imágenes de rostros (a), para los diccionarios de Español e Inglés (b), para el diccionario de Inglés y el Vocabulario (c) y para los documentos (d).

De ahora en adelante, los costos de los joins y de las consultas por rango no incluirán los costos de construir los índices *LTC* y *LC*, porque se considera que dicho costo sería amortizado entre varias consultas de joins y de rango.

6.4.2. Joins por Rango

En estos experimentos se han usado los siguientes parámetros. Para las imágenes de rostro se han considerado radios que recuperan en promedio 1, 5 y 10 imágenes relevantes desde la base de datos FACES762 por consulta por rango, cuando los elementos de consulta provienen desde la base de datos FACES254. Estos radios corresponden a valores de r iguales a $0,2702$, $0,3567$ y $0,3768$, respectivamente. Para los diccionarios se usan radios r de 1, 2 y 3, porque la distancia de edición es discreta. En los



(a) Imágenes de rostros.

(b) Diccionarios de Español e Inglés.

(c) Diccionario de Inglés y Vocabulario.

(d) Documentos desde TREC-3.

Figura 6.3: Construcción de *LTC* variando el radio de los clusters.

joins entre los diccionarios estos radios recuperan en promedio $0,05$, $1,5$ y 26 palabras en español por cada palabra en inglés, respectivamente. En los joins entre el diccionario en Inglés y el Vocabulario, éste recupera $7,9$, 137 y $1{,}593$ términos del vocabulario en promedio por cada palabra en inglés, respectivamente. Por otra parte, para el espacio de documentos, se han usado radios que recuperan en promedio 2, 3 y 30 documentos relevantes desde DOCS1846 por consulta por rango, cuando los elementos de consulta provienen de DOCS1111. Así, los valores de los radios r para el espacio de documentos son $0,25$, $0,265$ y $0,38$, respectivamente

Si se tiene sólo una base de datos indexada, se puede obtener trivialmente la respuesta al join por similitud $\mathbb{A} \bowtie_r \mathbb{B}$ ejecutando una consulta clásica por rango con radio r para cada elemento desde la otra base de datos. Debido a que la *LTC* se ha basado en la *LC*, en los experimentos se muestran también los resultados obtenidos con este algoritmo simple que considera una *LC* construida para una de las bases de datos. A este algoritmo de join se lo ha denominado *LC-join-rango*.

Más aún, si se tienen ambas bases de datos $\mathbb{A}$ y $\mathbb{B}$ indexadas, aunque se podría también en este caso aplicar la solución trivial (es decir, ignorando uno de los índices), se pueden evitar más cálculos de distancia si se usa toda la información de la que se dispone desde ambos índices. A fin de comparar la *LTC* con un ejemplo de esta clase de algoritmo, se ha considerado indexar ambas bases de datos usando una *LC* y luego aplicar un algoritmo de join que use toda la información disponible desde ambos índices para mejorar los costos del join. A este nuevo algoritmo se lo ha denominado *JoinRango-LC2* y la descripción del mismo se ilustra mediante el Algoritmo 34.

Algoritmo 34 Join por rango usando dos *Listas de Clusters*.

JoinRango-LC2 (Lista de Clusters L_1, Lista de Clusters L_2, Radio r)

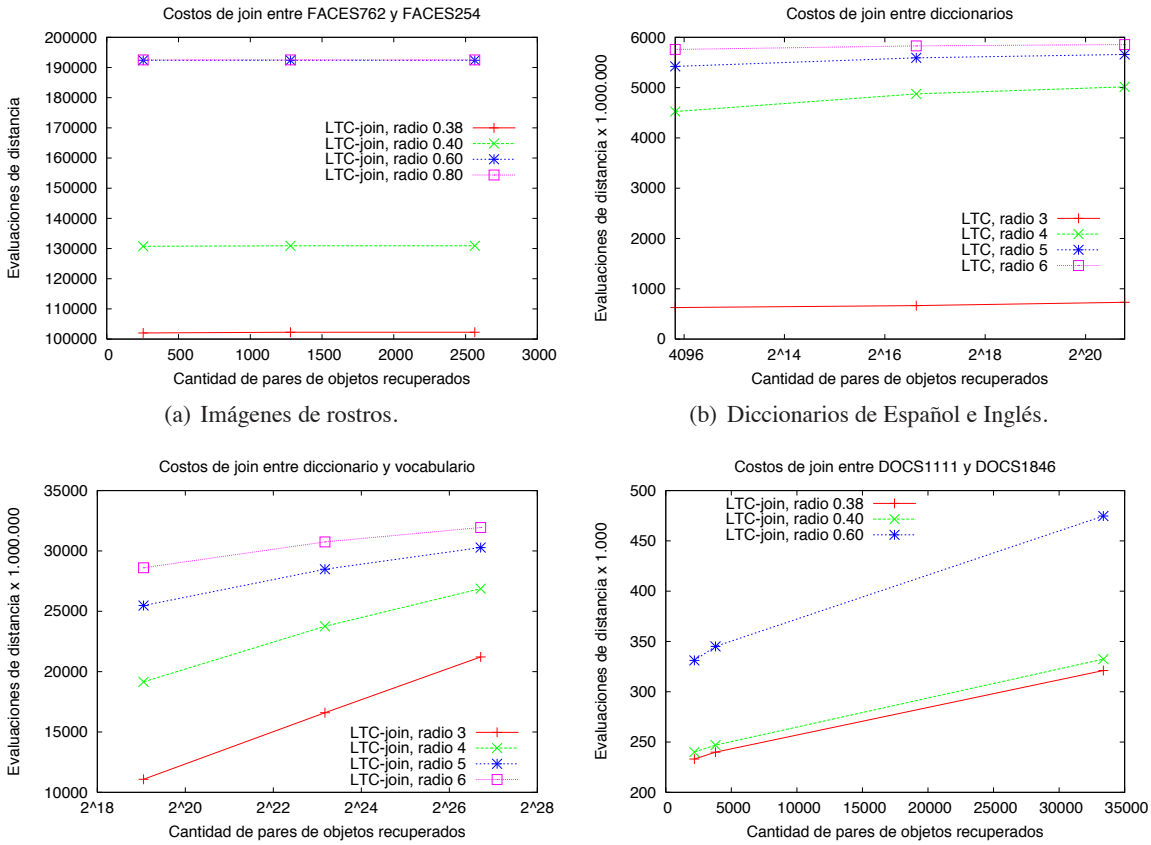
```
1.  For (c_i, r_{c_i}, I_{c_i}) ∈ L_1 Do
2.      For (c_j, r_{c_j}, I_{c_j}) ∈ L_2 Do
3.          d_cc ← d(c_i, c_j), d_s ← d_cc + r_{c_i} + r_{c_j}
4.          If d_cc ≤ r Then Informar (c_i, c_j)
5.          If 2 máx{d_cc, r_{c_i}, r_{c_j}} − d_s ≤ r Then   /* desigualdad triangular generalizada */
6.              For y ∈ I_{c_j} Do   /* d(c_j, y) está almacenada en L_2 */
7.                  If |d_cc − d(c_j, y)| ≤ r Then   /* verificar a y con el centro c_i */
8.                      d_y ← d(c_i, y)
9.                      If d_y ≤ r Then Informar (c_i, y)
10.             For x ∈ I_{c_i} Do   /* d(c_i, x) está almacenada en L_1, se verifican pares (x, y) */
11.                 d_s ← d_cc + d(c_i, x) + d(c_j, y)
12.                 lb ← 2 máx{d_cc, d(c_i, x), d(c_j, y)} − d_s
13.                 If d_y ''fue calculada'' Then lb ← máx{lb, |d_y − d(c_i, x)|}
14.                 If lb ≤ r ∧ d(x, y) ≤ r Then Informar (x, y)
15.             For x ∈ I_{c_i} Do   /* verificar todos los x ∈ I_{c_i} con el centro c_j */
16.                 If |d_cc − d(c_i, x)| ≤ r ∧ d(x, c_j) ≤ r Then Informar (x, c_j)
17.         If d_cc + r_{c_i} + r ≤ r_{c_j} Then
18.             Break   /* detiene la búsqueda de (c_i, r_{c_i}, I_{c_i}) sobre L_2 */
```

Debido a que se necesita establecer el radio de construcción R antes de construir los índices *LC* y *LTC*, en cada caso se han considerado diferentes radios y se ha elegido el que obtiene mejor costo de construcción para cada alternativa. Se han testeado varios casos donde el radio de construcción R es mayor o igual que el radio r más grande usado para $\mathbb{A} \bowtie_r \mathbb{B}$. También se ha incluido un breve test a fin de evaluar el desempeño cuando el radio de join es mayor que el de indexación; es decir, cuando $r > R$.

La Figura 6.4 ilustra el desempeño del JoinRango-LTC, considerando los diferentes radios en todos los pares de bases de datos, para las imágenes de rostros (a), los diccionarios de Español e Inglés (b), el diccionario de Inglés y el Vocabulario (c) y para los documentos (d). Se muestra además el número de pares de objetos recuperados en cada caso.

Como se puede notar, el mejor resultado para *LTC* se obtiene cuando el radio de construcción R es el valor más cercano al mayor valor de r considerado en cada caso. El JoinRango-LC2 tiene un comportamiento similar, pero en el caso del JoinRango-LC, el mejor radio puede variar un poco; de hecho, para el join por rango entre ambos diccionarios el mejor radio de construcción es $R = 4$ y para los documentos es de $R = 0,60$.

La Figura 6.5 muestra una comparación entre los tres algoritmos de join por rango, sin considerar costos de construcción, para los cuatro pares de bases de datos, usando el mejor valor del radio de construcción R determinado experimentalmente, para cada algoritmo de join por rango: para las bases de datos de imágenes de rostros (a), para los diccionarios de Español e Inglés (b), para el diccionario de Inglés y el Vocabulario (c) y para las bases de datos de documentos (d). Nuevamente, se muestra el número de pares de objetos recuperados. Se puede observar que el algoritmo de join por rango de *LTC* supera ampliamente a los otros dos algoritmos de join por rango considerados, en tres de los cuatro pares de bases de datos usados. Para el caso del join por rango entre el diccionario de palabras en Inglés y el Vocabulario, los algoritmos JoinRango-LC y JoinRango-LC2 son superiores a la *LTC*, a pesar de que el algoritmo JoinRango-LTC consigue una mejora significativa respecto del join de iteración anidada en todos los radios usados.



(a) Imágenes de rostros.

(b) Diccionarios de Español e Inglés.

(c) Diccionario de Inglés y Vocabulario.

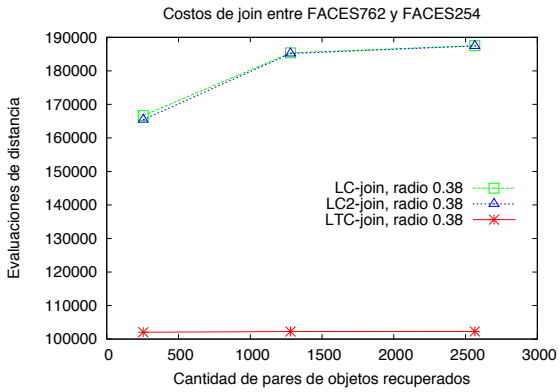
(d) Documentos desde TREC-3.

Figura 6.4: Comparación entre los diferentes radios de clusters considerados para la construcción del índice *LTC*. Notar las escalas logarítmicas.

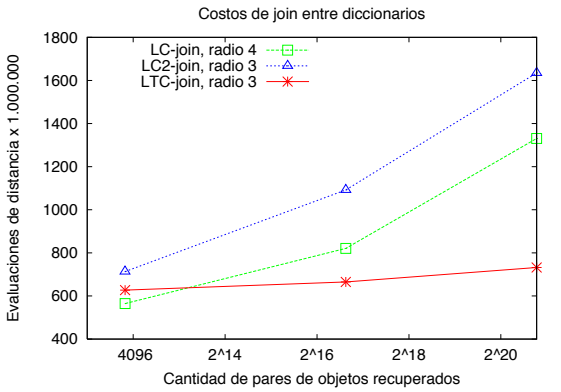
Se puede sospechar que este comportamiento no intuitivo que muestra que un algoritmo simple, el JoinRango-LC, supera al join por rango de *LTC* entre el Vocabulario y el diccionario de Inglés se puede explicar si se considera el número de objetos no indexados del índice *LTC*. En este caso, el 30 % de los términos del Vocabulario quedan sin indexar por la lista de clusters gemelos, mientras que en los otros casos, donde el join por rango de *LTC* es el mejor método, el porcentaje de objetos no indexados es más bajo. Por ejemplo, en los experimentos sobre las imágenes de rostros, sólo el 2 % de los rostros no son indexados; en el experimento de los diccionarios de Español e Inglés las palabras no indexadas por la lista de clusters gemelos representa el 23 % del conjunto y para los documentos el porcentaje de documentos no indexados es del 20 %.

A fin de ilustrar este comportamiento no intuitivo, se han dividido los costos del join en tres partes: (1) para centros, (2) para objetos regulares y (3) para objetos no indexados. Los valores de estos costos se muestran en la Tabla 6.1. Como se puede observar, en un caso favorable a *LTC* como el de las imágenes de rostros, la mayoría del trabajo se realiza entre objetos regulares. En cambio, en el join entre el Vocabulario y el diccionario de Inglés, la mayoría del trabajo se realiza cuando se resuelven los objetos no indexados.

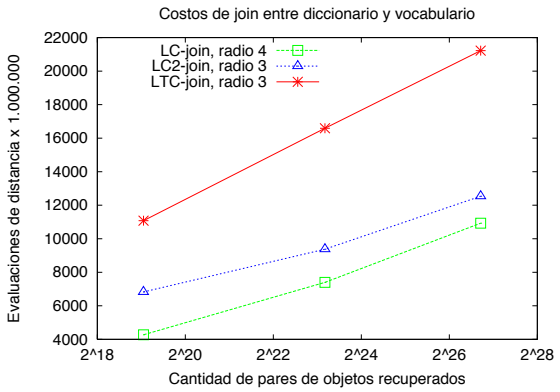
La Figura 6.6 exhibe una breve comparación entre los tres algoritmos por rango, sin los costos de construcción, cuando el radio de join es mayor que el de construcción; es decir, $r > R$. En dicha figura se muestran gráficas para las imágenes de rostros (a) y para los documentos (b). Nuevamente, se muestra el número de pares de objetos recuperados. Como era de esperar, se observa una degradación en el



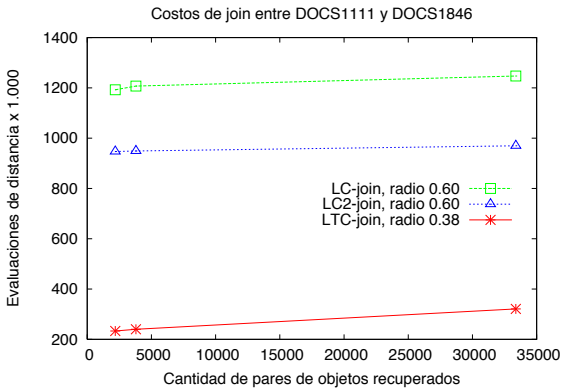
(a) Imágenes de rostros.



(b) Diccionarios de Español e Inglés.



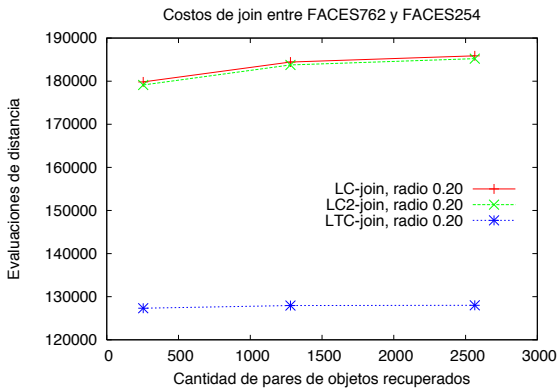
(c) Diccionario de Inglés y Vocabulario.



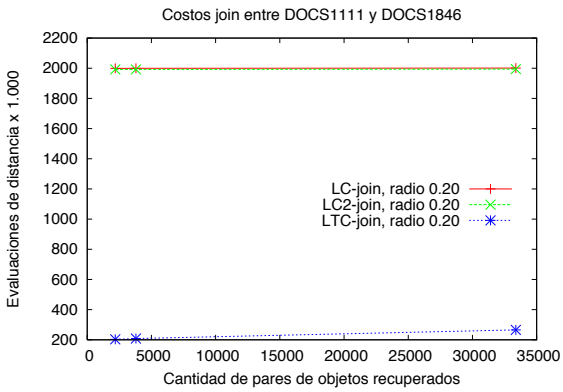
(d) Documentos desde TREC-3.

Figura 6.5: Comparación entre todos los algoritmos de join por rango considerados, usando en cada caso el mejor radio de construcción determinado experimentalmente para el índice. Notar las escalas logarítmicas.

desempeño del algoritmo de join por rango de *LTC*, que también se evidencia en ambos: JoinRango-LC y JoinRango-LC2, aunque aún sigue siendo la mejor alternativa de join por rango.



(a) Imágenes de rostros.



(b) Documentos desde TREC-3.

Figura 6.6: Comparación entre todos los algoritmos de búsqueda por rango considerados, usando un radio de join mayor que el radio de indexación; es decir $R < r$.

Tabla 6.1: Fracción del total del costo del join realizado por centros, objetos regulares y objetos no indexados.

(a) Join entre las bases de datos de imágenes de rostros.

	$r = 0,2702$	$r = 0,3567$	$r = 0,3768$
centros	1,8 %	1,8 %	1,8 %
objetos regulares	83,9 %	84,0 %	84,0 %
objetos no indexados	14,3 %	14,2 %	14,2 %

(b) Join entre el diccionario de Inglés y el Vocabulario.

	$r = 1$	$r = 2$	$r = 3$
centros	0,3 %	0,7 %	1,4 %
objetos regulares	27,8 %	29,6 %	30,8 %
objetos no indexados	71,9 %	69,7 %	67,8 %

Finalmente, la Tabla 6.2 brinda las proporciones de cálculos de distancia para los cuatro pares de bases de datos. Los valores son calculados de acuerdo a la siguiente fórmula:

$$\frac{\text{join} - \text{JoinRango-LTC}}{\text{join}} \cdot 100\,\%$$

Tabla 6.2: Proporción del desempeño del JoinRango-LTC para todos los pares de bases de datos, en todos los radios usados, con respecto a los otros métodos de join por rango.

(a) Join entre las bases de datos de imágenes de rostros.

Radio	JoinRango-LC	JoinRango-LC2	Iteración Anidada
0,2702	38 %	38 %	47 %
0,3567	44 %	44 %	47 %
0,3768	45 %	45 %	47 %

(b) Join entre los diccionarios de Español e Inglés.

Radio	JoinRango-LC	JoinRango-LC2	Iteración Anidada
1	−11 %	12 %	89 %
2	19 %	39 %	88 %
3	45 %	55 %	87 %

(c) Join entre el diccionario de Inglés y el Vocabulario.

Radio	JoinRango-LC	JoinRango-LC2	Iteración Anidada
1	−159 %	−62 %	67 %
2	−124 %	−76 %	51 %
3	−94 %	−69 %	38 %

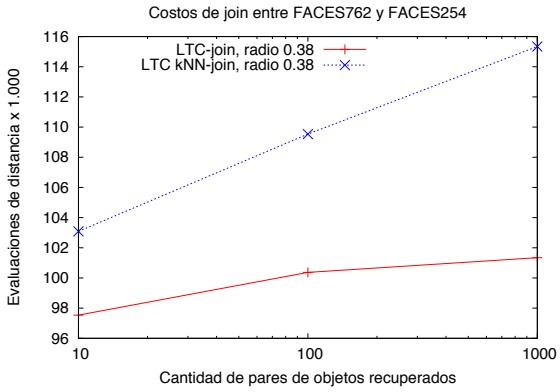
(d) Join entre las bases de datos DOCS1846 y DOCS1111.

Radio	JoinRango-LC	JoinRango-LC2	Iteración Anidada
0,25	80 %	75 %	88 %
0,265	80 %	74 %	88 %
0,38	74 %	67 %	84 %

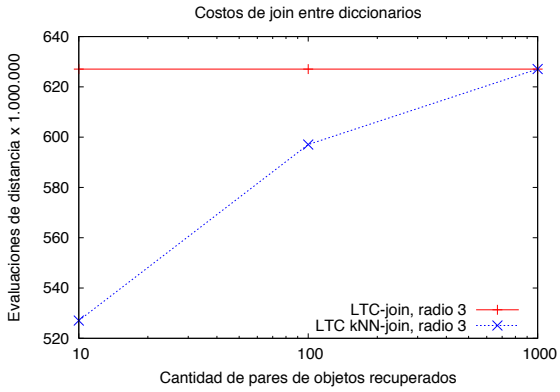
6.4.3. Join de k Pares Más Cercanos

En este caso, sólo se puede comparar el desempeño del algoritmo de join de k pares más cercanos $\mathbb{A} \bowtie_k \mathbb{B}$ basado en *LTC*, con el JoinRango-LTC, porque no se tiene otra alternativa para espacios métri-

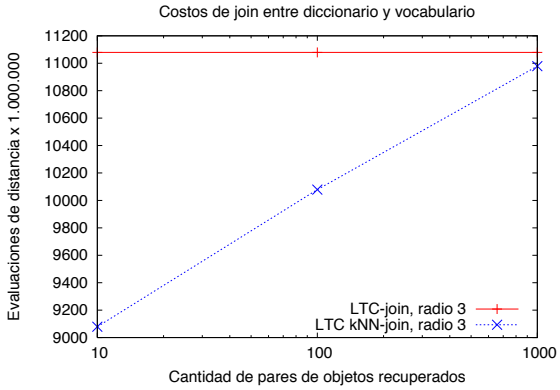
cos [2]. La Figura 6.7 muestra los resultados obtenidos cuando se recuperan los 10, 100 y 1.000 pares de elementos más cercanos entre $\mathbb{A}$ y $\mathbb{B}$, para los cuatro pares de bases de datos: para las imágenes de rostros (a), para los diccionarios de Español e Inglés (b), para el diccionario de Inglés y el Vocabulario (c) y para las bases de datos de documentos (d). Como es posible observar, el desempeño del $\mathbb{A} \bowtie_k \mathbb{B}$ es similar al del equivalente join por rango. Esto revela que la estrategia usada para reducir el radio de búsqueda funciona efectivamente. Es interesante notar que el desempeño del join de k pares más cercanos en el espacio de palabras es levemente mejor que el del join por rango. Esto se debe a que la distancia de edición es discreta y existen miles de palabras a distancia 1. Por lo tanto, el heap de los k pares de candidatos se llena tempranamente con pares a distancia 1 y las siguientes consultas por rango usan radio 0.



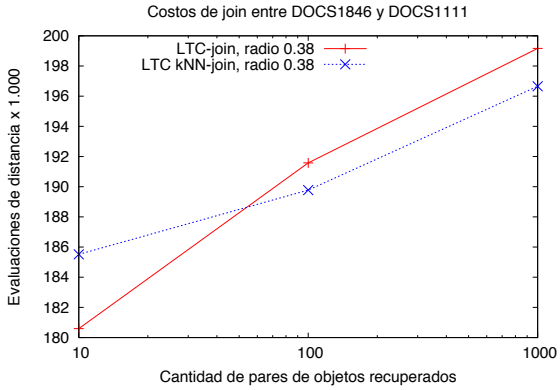
(a) Imágenes de rostros.



(b) Diccionarios de Español e Inglés.



(c) Diccionario de Inglés y Vocabulario.



(d) Documentos desde TREC-3.

Figura 6.7: Comparación entre el join de k pares más cercanos y el join por rango equivalente. Notar las escalas logarítmicas.

6.4.4. Consultas por Rango

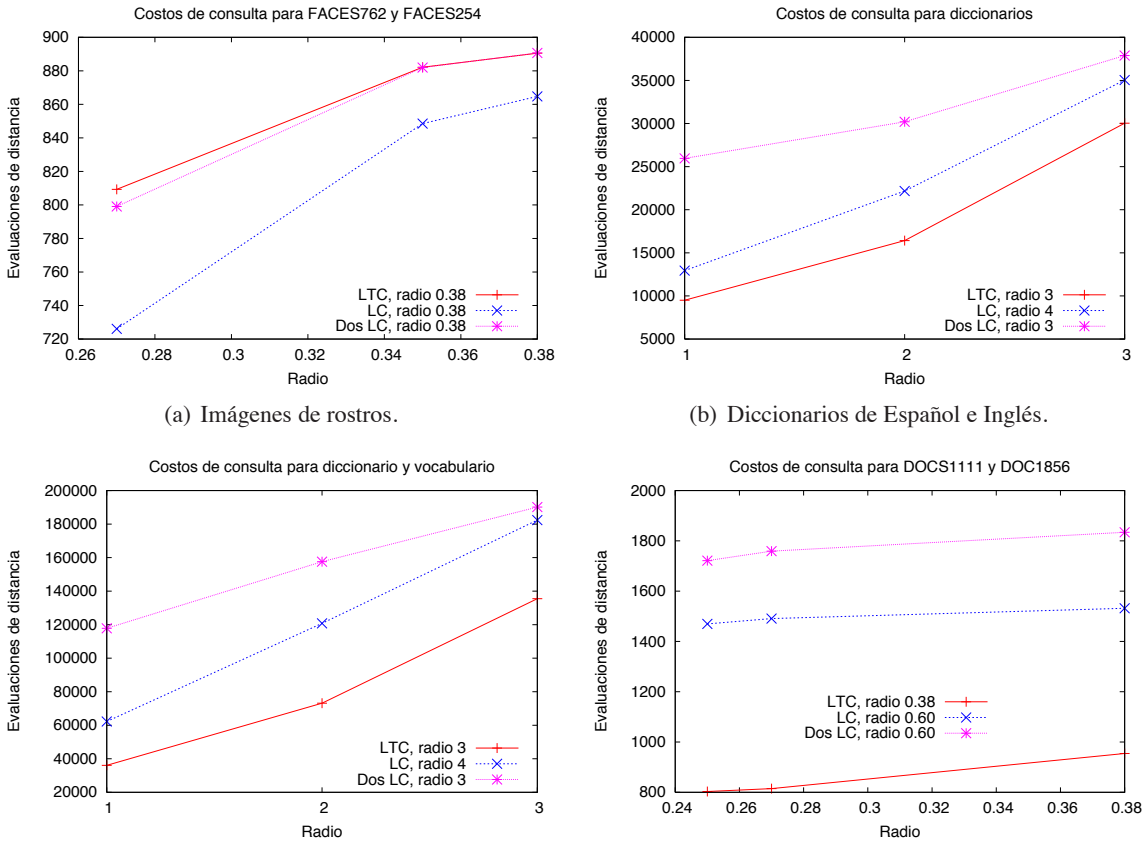
Se han calculado las consultas por rango usando los mismos radios de join de la Sección 6.4.2. Esto corresponde a, en el caso del espacio de imágenes de rostros a usar radios r iguales a 0,2702, 0,3567 y 0,3768, recuperando 0,7, 4,5 y 9,1 imágenes, respectivamente. Para las palabras se han usado los radios r de 1, 2, y 3, recuperando 2, 25 y 229 palabras para ambos diccionarios de Inglés y Español; y 7, 161 y 2.025 palabras desde el diccionario de Inglés y el Vocabulario, respectivamente. Para el

[2]Hasta donde se conoce, no han habido intentos previos de resolver esta variante de join.

espacio de documentos se han usado los radios de $0{,}25$, $0{,}265$ y $0{,}38$, recuperando 3, 5 y 47 documentos, respectivamente. Estos resultados se han promediado sobre los subconjuntos completos de consultas; es decir, sobre el $10\,\%$ de la unión de ambas bases de datos.

Las gráficas de la Figura 6.8, para las bases de datos de imágenes de rostros (a), diccionarios (b), el diccionario de Inglés y el Vocabulario (c) y los documentos (d), muestran una comparación del algoritmo de consultas por rango basado en la *LTC* con respecto a: (i) indexar la unión de ambas bases de datos usando una única *LC* y (ii) indexar cada base de datos con una *LC*. La alternativa (i) implica agregar un nuevo índice para soportar las consultas por rango, mientras que la alternativa (ii) es equivalente a la aproximación de la *LTC*, en el sentido que se reusan los índices con el fin de hacer frente a los joins por similitud y a las primitivas clásicas de búsquedas por similitud.

En la comparación, el algoritmo de consulta por rango basado en *LTC* muestra en buen desempeño cuando se lo compara con la aproximación básica de *LC*. El motivo de esto se explica cuando se considera que el índice *LTC* almacena más distancias que el índice *LC* básico. De hecho, la matriz de distancias entre centros permite reducir efectivamente la cantidad de cálculos de distancia realizados en tres casos. Con respecto a los competidores, se puede ver que sistemáticamente es mejor tener una única *LC* que indexa la unión de las bases de datos que dos *LC* que indexan a una base de datos cada una de manera independiente. Finalmente, como se puede observar en la Figura 6.8(a), sólo en el espacio de imágenes de rostros usar una única *LC* es levemente más rápida que las consultas por rango en la *LTC*.



(a) Imágenes de rostros.

(b) Diccionarios de Español e Inglés.

(c) Diccionario de Inglés y Vocabulario.

(d) Documentos desde TREC-3.

Figura 6.8: Cálculo de las consultas por rango sobre *LTC* y con una o dos *LC*, variando los radios.

Capítulo 7

Conclusiones

Cecilia Ramos, `http://www.lache.co/` [1]

Las bases de datos métricas recientemente se han vuelto populares como modelo para manipular varias bases de datos emergentes de objetos complejos no estructurados, en los cuales sólo la búsqueda por similitud puede ser significativa. Aunque se puede resolver cualquier búsqueda por similitud realizando una búsqueda exhaustiva, que compare la consulta con cada elemento de la base de datos, esta solución no es aplicable en una situación en que el número de objetos sea grande. Por lo tanto, con el fin de poder realizar búsquedas por similitud eficientes, es necesario diseñar índices. Han surgido varios *índices métricos* (también llamados *métodos de acceso métrico*, o *MAM* por su sigla en inglés); esto es, estructuras de datos que permiten acelerar las búsquedas por similitud. Estos índices pueden clasificarse de diferente manera, dependiendo de las diferentes características que pueden considerarse: la estrategia usada, el tipo de almacenamiento, el tipo de respuesta, si hay actualizaciones y el tipo de operación que soportan (sólo búsquedas o joins).

Así, el propósito de esta tesis ha sido principalmente el acercar las bases de datos métricas a un nivel mayor de madurez, a través de optimizar índices existentes, de conseguir índices eficientes y dinámicos en memoria secundaria y de proponer soluciones para algunas variantes del join por similitud, una operación muy relevante en bases de datos y que no se ha considerado mucho en bases de datos métricas.

[1] Agradezco a Cecilia Ramos por haberme autorizado a utilizar estas imágenes y por facilitarme versiones con buena resolución.

A continuación se resumen los principales aportes de esta tesis y luego se describen algunas posibilidades de trabajo futuro.

7.1. Aportes

Dentro del marco de este trabajo se han intentado abarcar algunas de las categorías de índices posibles, considerando como punto de partida a índices basados en particiones compactas y con tipo de respuesta por similitud exacta. Dentro de esta clase, se han considerado índices estáticos y dinámicos, para memoria principal y para memoria secundaria e índices para resolver búsquedas y joins por similitud.

En parte del Capítulo 3 y en los Capítulos 4, 5 y 6, se describen los aportes de esta tesis. Estos aportes se pueden resumir de la siguiente manera:

- Nuevos métodos de inserción y eliminación en el *Árbol de Aproximación Espacial Dinámico* (*DSAT*), que conjuntamente con los ya existentes, han permitido mostrar que una estructura "muy estática" como el *Árbol de Aproximación Espacial* (*SAT*) [Nav02] podía transformarse en una estructura más eficiente y "muy dinámica" como es el actual *DSAT*, ya que no sólo permite mejorar las búsquedas y la construcción respecto del *SAT* sino que además se puede lograr de muchas maneras distintas.

- Un nuevo y mejorado *SAT*, al que se ha denominado *Árbol de Aproximación Espacial Distal* (*DiSAT*). A esta nueva versión del *SAT* se pudo llegar gracias al profundo conocimiento adquirido en el estudio del *DSAT*. El *DiSAT*, con sus principales variantes SAT^{+} y SAT^{Glob}, ha conseguido establecerse como uno de los índices más eficientes en las búsquedas, cuya principal ventaja frente a otras alternativas es no necesitar de la sintonización de ningún parámetro. El *DiSAT* usa sólo una nueva heurística de construcción para el *SAT*. La regla es contraintuitiva y consiste en seleccionar objetos lejanos como vecinos en lugar de cercanos.

- Mejoras a la *Lista de Clusters* (*LC*) [CN05] que permiten aprovechar algunas de las distancias calculadas durante la construcción del índice para, si existe espacio disponible, mejorar el desempeño de las búsquedas. Es decir, distintas maneras de aprovechar el espacio disponible para mejorar las búsquedas, sin incrementar los costos de construcción. Además, se ha propuesto un algoritmo de búsqueda de k vecinos más cercanos.

- Se han presentado tres nuevos índices dinámicos para memoria secundaria. Los índices *DSAT** y el *DSAT+* extienden el *DSAT* a memoria secundaria, soportando inserciones y búsquedas. La *Lista de Clusters Dinámica* (*DLC*) extiende a la *LC*, el cual tiene buen desempeño en espacios de mediana a alta dimensión, pero que es costoso de construir. La *DLC* permite inserciones, eliminaciones y búsquedas, heredando ambas características de la *LC*, así su tiempo de construcción es muy alto en términos de evaluaciones de distancia, aunque el número de operaciones de E/S es esencialmente de 2 (el costo óptimo en operaciones de E/S para una inserción en una estructura en disco: una lectura y una escritura). Finalmente, el índice *Conjunto Dinámico de Clusters* (*DSC*) combina la *DLC* con el *DSAT* de memoria principal a fin de reducir la cantidad de evaluaciones de distancia en las inserciones.

  Las mejores variantes de estas estructuras generalmente superan a algunas de las alternativas prominentes de la literatura como el *M–tree* [CPZ97], el *eGNAT* [NU11] y el *MX-tree* [JKF13]. El *DSAT+* permite inserciones con unas decenas de evaluaciones de distancia y unas pocas operaciones de E/S, y su costo crece muy lentamente a medida que crece el tamaño de la base de datos.

Su costo de búsqueda es bueno cuando los espacios son de dimensión baja a mediana o las consultas son selectivas. La estructura *DSC* supera al *DSAT+* para las búsquedas, en evaluaciones de distancia y en operaciones de E/S, cuando los espacios son de alta dimensión o las consultas son de baja selectividad. Durante la construcción *DSC* requiere 2 operaciones de E/S por inserción, pero realiza muchas más evaluaciones de distancia que el *DSAT+*, aunque su costo crece también de manera sublineal.

- Se ha planteado una nueva aproximación para computar algunas variantes del join por similitud entre dos bases de datos, la cual consiste en indexar ambas bases de datos conjuntamente. Para ellos, se ha propuesto un nuevo índice métrico, denominada *Lista de Cluster Gemelos* (*LTC*). Se ha verificado experimentalmente que el costo de construir el índice *LTC* es similar al de construir una única *LC* para indexar una base de datos más grande.

  Se ha propuesto la solución de dos clases de join por similitud basada en el índice *LTC*: *join por rango* $\mathbb{A} \bowtie_r \mathbb{B}$: dado un radio r encontrar todos los pares de objetos (con un objeto desde cada base de datos) a distancia a lo sumo de r; y *join de k pares más cercanos* $\mathbb{A} \bowtie_k \mathbb{B}$: encontrar los k pares de objetos más cercanos (con un objeto desde cada base de datos) . Los resultados de la evaluación experimental del join por rango no sólo mejora significativamente sobre la alternativa ingenua básica de tiempo cuadrático (el algoritmo de iteración anidada o de fuerza bruta) sino también sobre los otros dos algoritmos de join, JoinRango-LC y JoinRango-LC2, para tres de los cuatro pares de bases de datos considerados.

  Por otra parte, con respecto al join de k pares más cercanos, los resultados de la evaluación experimental muestran que es bastante eficiente, dado que requiere un trabajo similar al que realiza un join por rango equivalente sobre el índice *LTC*. Este hecho se asemeja al desempeño de los algoritmos para las búsquedas de k vecinos más cercanos de rango óptimo [HS00, HS03b]. Además, tal como lo se establece en trabajos recientes [KTA11, FB15], esta propuesta ha sido la primera en la literatura para resolver este tipo de consulta.

  Finalmente, se ha mostrado que el algoritmo de consulta por rango basado en la *LTC* es competitivo con, y en algunos casos mejor que, el algoritmo de búsqueda de la *LC*.

  Por lo tanto, el índice *LTC* se establece como una estructura de datos práctica y eficiente para resolver dos casos particulares de joins por similitud, tales como $\mathbb{A} \bowtie_r \mathbb{B}$ y $\mathbb{A} \bowtie_k \mathbb{B}$, y como un índice para acelerar las consultas por rango clásicas. La *LTC* se puede usar para pares de bases de datos de cualquier espacio métrico y por lo tanto tiene un rango de aplicaciones más amplio que aquellos índices de join para espacios vectoriales.

Cabe mencionar aquí también que los principales resultados de nuestro trabajo han sido publicados en [NR08, GCMR08, PR08, GCMR09, CRR09, PR09, CLRR11, CLRR14, NR14, CLRR16, NR16].

7.2. Trabajos Futuros

Cecilia Ramos, `http://www.lache.co/`

A continuación se enuncian posibles extensiones a este trabajo, algunas de las cuales se van a encarar en el futuro, mientras que otras ya se están desarrollando:

- Analizar la posibilidad de aplicar, hasta donde sea posible, la heurística del *DiSAT* al *DSAT* tanto en las versiones para memoria principal como para memoria secundaria, de manera tal de producir subárboles más compactos, induciendo más localidad en las particiones por hiperplanos implícitas de los subárboles. Este factor podría impactar en las operaciones de E/S en las versiones para memoria secundaria del *DSAT*.

- Estudiar posibles aplicaciones para el *DiSAT*, dado que una posible consecuencia de una partición subyacente compacta como la que se logra, inducida por un radio de cobertura pequeño, es la posibilidad de producir clusters coherentes más adecuados para propósitos de estadísticas, minería de datos, reconocimiento de patrones y aprendizaje de máquina. Un aspecto generalmente considerado para un procedimiento de clustering es producir un clustering estable (que sea independiente de la elección de la raíz, por ejemplo), o alternativamente detectar de manera natural clusters libres de parámetros.

- En *DSAT+* sería posible aprovechar el trabajo realizado en una búsqueda para mejorar la ocupación de página de la parte alta del árbol (la que se accede más frecuentemente) y por consiguiente mejorar el desempeño de las búsquedas futuras. Durante el recorrido realizado en el árbol por una búsqueda es posible detectar, entre las páginas leídas, páginas a las que le sobra suficiente espacio como para "subir" una vecindad completa. Es decir, si se detecta una página i que dispone de espacio libre para almacenar n_i nodos más y desde uno de sus nodos x la búsqueda continúa en una vecindad $N(x)$ alojada en una página j, donde $|N(x)| \leq n_i$, dicha vecindad podría "subirse" a la página i para mejorar la ocupación de la página i. Sin embargo, para que este cambio no incremente demasiado los costos de las búsquedas, se puede considerar que sólo se realice en la primera página que cumpla con la posibilidad de realizarlo. De esta manera, las búsquedas sólo incrementarán sus costos en a lo sumo dos operaciones de escritura sobre las páginas involucradas. En un régimen donde las consultas se realizan en general luego de las inserciones, esta alternativa podría obtener importantes mejoras en las búsquedas.

- En *DSC* el desafío futuro más interesante sería lograr reducir el número de evaluaciones de distancia que requiere la inserción de un elemento, dado que es su punto débil.

- Reducir en *DSC* la degradación que sufre el índice cuando se han eliminado muchos elementos, aunque esto sólo sucede cuando una amplia fracción de la base de datos se ha eliminado y en este punto una reconstrucción completa puede tener un bajo costo amortizado. Además, se puede analizar cuánto más se pagaría en eliminación si se reinsertaran los elementos de un cluster que,

por contener muy pocos elementos, debe desaparecer y cuánto mejora esto el costo de las búsquedas que se realizan luego (hasta el momento se tratan de reinsertar grupos de elementos de dicho cluster por asumir que entre ellos son similares).

- Otro posible desafío para *DSC* es extender la estructura a casos donde la parte que se mantiene en memoria (un objeto por página de disco) tampoco cabe en memoria principal. En este caso, una estructura jerárquica permitiría manejar bases de datos extremadamente grandes; es decir, un *DSC* sobre los centros, en lugar de una estructura en memoria principal.

- Analizar la posibilidad de aprovechar el espacio disponible en memoria principal, en caso de existir, con el fin de mejorar el desempeño de la *DSC*, ya sea extendiendo el *DSAT* de centros a un *Árbol de Aproximación Espacial Híbrido* (HDSAT) [Arr14] o agregando *regiones de corte* sobre los clusters [LMČS14].

- Considerar para *DSC* otras posibles maneras de dividir los clusters luego de una inserción, como por ejemplo la estrategia del *Slim–tree* [JTSF00] para lograr clusters más compactos y más separados entre sí, en los que además sea posible mantener un balance sobre la cantidad de elementos.

- En el caso particular del auto-join por similitud no existe razón para construir un índice *LTC*. Por lo tanto, se podría diseñar especialmente otra variante de *LC* para esta clase de join.

- Analizar si es posible optimizar la *LTC* evaluando distancias "internas" en cada una de las dos bases de datos, porque durante la construcción de la *LTC* y cuando se evalúa el join por similitud, no se calcula ninguna distancia entre elementos de la misma base de datos. Se debe considerar aquí si se pueden mejorar los costos del join al calcular algunas distancias internas a fin de obtener mejores límites inferiores de las distancias "externas"; es decir, distancias entre elementos de distinta base de datos.

- Considerar otras maneras de seleccionar centros para la *LTC*. La mejor manera de seleccionar el centro gemelo de un centro es elegir su objeto más cercano en la otra base de datos. Sin embargo, se podrían evaluar otras maneras de elegir un nuevo centro, desde el último centro gemelo, a fin de representar el clustering real de la base de datos usando el número mínimo posible de centros de clusters. Además, eligiendo mejores centros se podría reducir la cantidad de memoria necesaria para la matriz de distancias entre centros.

- Desarrollar soluciones para otras clases de joins por similitud sobre la *LTC* o variantes que puedan surgir de ella. Por ejemplo, calcular el join de k vecinos más cercanos. Además, se puede analizar cómo resolver otros operadores, como los propuestos en [CSO$^+$15], o analizar la interacción con otros operadores de similitud o no de similitud en un ambiente físico de bases de datos [SPCR15].

- Estudiar estrategias, usando *LTC*, para resolver consultas por rango clásicas sobre una u otra base de datos (actualmente se resuelven sobre la unión de ambas).

- Como la *LTC* usa clusters de radio fijo, se ha observado experimentalmente que los primeros clusters están mucho más poblados que los siguientes en la lista. Por lo tanto, se podría evaluar la opción de construir la *LTC* con clusters de tamaño fijo, como así también desarrollar una version de *LTC* similar a la *Lista de Clusters Recursiva* [Mam05].

- Analizar la posible combinación de *DSC* o *DLC* con *LTC* para obtener un índice para join que sea dinámico, permitiendo que los elementos de cualquiera de las bases de datos se vayan insertando en el índice a medida que se vayan incorporando a la base de datos.

- Uno de los principales problemas que tiene la *LTC* es la existencia de muchos objetos no indexados, ya que debido a ellos se degrada el desempeño del join por rango, así que se podrían considerar otras alternativas para manejar los objetos no indexados. Por ejemplo, usar permutantes [CFN08].
- Un aspecto no considerado al resolver los joins por similitud con *LTC* es el de *diversidad en la respuesta*; es decir, responder al operador de *join por similitud diverso* [SCO+15]. Por lo tanto, se puede analizar cómo asegurar un conjunto más pequeño y más diversificado de respuestas útiles al realizar una consulta de join por similitud por rango con *LTC*.
- Considerar el desarrollo de mecanismos de control para bases de datos métricas, que permitan asegurar la consistencia y seguridad de las transacciones sobre la base de datos, mientras se permite que varios usuarios puedan acceder concurrentemente a la misma.
- Desarrollar variantes paralelas para los distintos índices propuestos u optimizados en este trabajo, buscando reducir en aplicaciones reales los tiempos de respuesta.

Bibliografía

[ABOPP12] Luis G. Ares, Nieves R. Brisaboa, Alberto Ordóñez Pereira, and Oscar Pedreira. Efficient similarity search in metric spaces with cluster reduction. In *Proceedings of the 5th International Conference on Similarity Search and Applications*, SISAP'12, pages 70–84, Berlin, Heidelberg, 2012. Springer-Verlag.

[AP02] Fabrizio Angiulli and Clara Pizzuti. Approximate k -closest-pairs with space filling curves. In *Proceedings of the 4th International Conference on Data Warehousing and Knowledge Discovery*, DaWaK 2000, pages 124–134, London, UK, UK, 2002. Springer-Verlag.

[AP05] Fabrizio Angiulli and Clara Pizzuti. An approximate algorithm for top-k closest pairs join query in large high dimensional data. *Data and Knowledge Engineering*, 53(3):263–281, 2005.

[Are12] Luis G. Ares. *Métodos de mellora do rendemento en buscas por similitude sobre espazos métricos*. PhD thesis, Facultade de Informática, Universidade da Coruña, 2012. Directores: Nieves R. Brisaboa, Óscar Pedreira Fernández.

[Arr14] Diego Arroyuelo. A dynamic pivoting algorithm based on spatial approximation indexes. In Agma Juci Machado Traina, Jr. Traina, Caetano, and Robson Leonardo Ferreira Cordeiro, editors, *Similarity Search and Applications*, volume 8821 of *Lecture Notes in Computer Science*, pages 70–81. Springer International Publishing, 2014.

[Aur91] Franz Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, 1991.

[BBK01] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, September 2001.

[BBKK01] Christian Böhm, Bernhard Braunmüller, Florian Krebs, and Hans-Peter Kriegel. Epsilon grid order: an algorithm for the similarity join on massive high-dimensional data. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, SIGMOD '01, pages 379–388, New York, NY, USA, 2001. ACM.

[BEF+09] Paolo Bolettieri, Andrea Esuli, Fabrizio Falchi, Claudio Lucchese, Raffaele Perego, Tommaso Piccioli, and Fausto Rabitti. CoPhIR: a test collection for content-based image retrieval. *CoRR*, abs/0905.4627v2, 2009.

[Ben75] Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Comm. of the ACM*, 18(9):509–517, 1975.

[Ben79] Jon L. Bentley. Multidimensional binary search trees in database applications. *IEEE Trans. on Software Engineering*, 5(4):333–340, 1979.

[BFPR06] Nieves R. Brisaboa, Antonio Fariña, Óscar Pedreira, and Nora Reyes. Similarity search using sparse pivots for efficient multimedia information retrieval. In *Proceedings of the Eighth IEEE International Symposium on Multimedia*, ISM '06, pages 881–888, Washington, DC, USA, 2006. IEEE Computer Society.

[BGRS99] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In Catriel Beeri and Peter Buneman, editors, *Database Theory - ICDT'99*, volume 1540 of *Lecture Notes in Computer Science*, pages 217–235. Springer Berlin Heidelberg, 1999.

[BK73] Walter A. Burkhard and Robert M. Keller. Some approaches to best-match file searching. *Comm. of the ACM*, 16(4):230–236, 1973.

[BK01] Christian Böhm and Hans-Peter Kriegel. A cost model and index architecture for the similarity join. In Georgakopoulos and Buchmann [GB01], pages 411–420.

[BK04] Christian Böhm and Florian Krebs. The *k*-nearest neighbour join: Turbo charging the kdd process. *Knowl. Inf. Syst.*, 6(6):728–749, 2004.

[BKK96] Stefan Berchtold, Daniel Keim, and Hans-Peter Kriegel. The X-tree: an index structure for high-dimensional data. In *Proc. 22nd Conference on Very Large Databases (VLDB'96)*, pages 28–39, 1996.

[BKK02] Christian Böhm, Florian Krebs, and Hans-Peter Kriegel. Optimal dimension order: A generic technique for the similarity join. In Yahiko Kambayashi, Werner Winiwarter, and Masatoshi Arikawa, editors, *DaWaK*, volume 2454 of *Lecture Notes in Computer Science*, pages 135–149. Springer, 2002.

[BKS93] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using R-trees. In *Proc. 1993 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'93)*, pages 237–246, 1993.

[BN04] Benjamín Bustos and Gonzalo Navarro. Probabilistic proximity search algorithms based on compact partitions. *Journal of Discrete Algorithms (JDA)*, 2(1):115–134, 2004.

[BNC03] Benjamin Bustos, Gonzalo Navarro, and Edgar Chávez. Pivot selection techniques for proximity searching in metric spaces. *Pattern Recognition Letters*, 24(14):2357–2366, 2003.

[BNRP15] Cristian Bustos, Gonzalo Navarro, Nora Reyes, and Rodrigo Paredes. An empirical evaluation of intrinsic dimension estimators. In Giuseppe Amato, Richard Connor, Fabrizio Falchi, and Claudio Gennaro, editors, *Similarity Search and Applications*, volume 9371 of *Lecture Notes in Computer Science*, pages 125–137. Springer International Publishing, 2015.

[BO97] Tolga Bozkaya and Meral Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proc. ACM Conference on Management of Data (ACM SIGMOD'97)*, pages 357–368, 1997. Sigmod Record 26(2).

[Böh01] Christian Böhm. The similarity join: A powerful database primitive for high performance data mining. tutorial, IEEE Int. Conf. on Data Engineering (ICDE), 2001.

[Boy11] Leonid Boytsov. Indexing methods for approximate dictionary searching: Comparative analysis. *J. Exp. Algorithmics*, 16:1.1:1.1–1.1:1.91, May 2011.

[BPR12] Luis Britos, A. Marcela Printista, and Nora Reyes. Dsacl+-tree: A dynamic data structure for similarity search in secondary memory. In Navarro and Pestov [NP12], pages 116–131.

[BPS+08] Nieves Brisaboa, Oscar Pedreira, Diego Seco, Roberto Solar, and Roberto Uribe. Clustering-based similarity search in metric spaces with sparse spatial centers. In *Proceedings of SOFSEM'08*, number 4910 in LNCS, pages 186–197, 2008.

[Bri95] Sergéi Brin. Near neighbor search in large metric spaces. In *Proc. 21st Conference on Very Large Databases (VLDB'95)*, pages 574–584, 1995.

[BRP10] Marcelo Barroso, Nora Reyes, and Rodrigo Paredes. Enlarging nodes to improve dynamic spatial approximation trees. In Paolo Ciaccia and Marco Patella, editors, *SISAP*, pages 41–48. ACM, 2010.

[BS80] Jon L. Bentley and James B. Saxe. Decomposable searching problems i. static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[BY97] Ricardo Baeza-Yates. Searching: an algorithmic tour. In A. Kent and J. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 331–359. Marcel Dekker Inc., 1997.

[BYCMW94] Ricardo Baeza-Yates, Walter Cunto, Udi Manber, and Sun Wu. Proximity matching using fixed-queries trees. In *Proc. 5th Combinatorial Pattern Matching (CPM'94)*, LNCS 807, pages 198–212, 1994.

[BYN98] Ricardo Baeza-Yates and Gonzalo Navarro. Fast approximate string matching in a dictionary. In *Proc. 5th South American Symposium on String Processing and Information Retrieval (SPIRE'98)*, pages 14–22. IEEE CS Press, 1998.

[BYRN99] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[BYRN11] Ricardo Baeza-Yates and Berhier Ribeiro-Neto. *Modern Information Retrieval: The Concepts and Technology Behind Search*. Addison Wesley, 2011.

[cC94] Tzi cker Chiueh. Content-based image indexing. In *Proc. of the 20th Conference on Very Large Databases (VLDB'94)*, pages 582–593, 1994.

[Cel08] Cengiz Celik. Effective use of space for pivot-based metric indexing structures. In *Data Engineering Workshop, 2008. ICDEW 2008. IEEE 24th International Conference on*, pages 402–409, April 2008.

[CFN08] Edgar Chávez, Karina Figueroa, and Gonzalo Navarro. Effective proximity retrieval by ordering permutations. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 30:1647–1658, 2008.

[CFP+05] Domenico Cantone, Alfredo Ferro, Alfredo Pulvirenti, Diego R. Recupero, and Dennis Shasha. Antipole tree indexing to support range search and k-nearest neighbor search in metric spaces. *IEEE Trans. on Knowl. and Data Eng.*, 17(4):535–550, April 2005.

[Cla99] Kenneth L. Clarkson. Nearest neighbor queries in metric spaces. *Discrete & Computational Geometry*, 22(1):63–93, 1999.

[CLRR11] Edgar Chávez, Verónica Ludueña, Nora Reyes, and Patricia Roggero. Reaching near neighbors with far and random proxies. *Proceedings of the 8th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE 2011)*, pages 574–581, 2011.

[CLRR14] Edgar Chávez, Verónica Ludueña, Nora Reyes, and Patricia Roggero. Faster proximity searching with the distal sat. In Agma Juci Machado Traina, Jr. Traina, Caetano, and Robson Leonardo Ferreira Cordeiro, editors, *Similarity Search and Applications*, volume 8821 of *Lecture Notes in Computer Science*, pages 58–69. Springer International Publishing, 2014.

[CLRR16] Edgar Chávez, Verónica Ludeña, Nora Reyes, and Patricia Roggero. Faster proximity searching with the distal sat. *Information Systems*, 2016.

[CMBY99] Edgar Chávez, José L. Marroquín, and Ricardo Baeza-Yates. Spaghettis: an array based algorithm for similarity queries in metric spaces. In *Proc. 6th International Symposium on String Processing and Information Retrieval (SPIRE'99)*, pages 38–46. IEEE CS Press, 1999.

[CMN01] Edgar Chávez, José L. Marroquín, and Gonzalo Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications*, 14(2):113–135, 2001.

[CN00a] Edgar Chávez and Gonzalo Navarro. An effective clustering algorithm to index high dimensional metric spaces. In *Proceedings of the 7th International Symposium on String Processing and Information Retrieval (SPIRE'2000)*, pages 75–86. IEEE CS Press, 2000.

[CN00b] Edgar Chávez and Gonzalo Navarro. Measuring the dimensionality of general metric spaces. Technical Report TR/DCC-00-1, Dept. of Computer Science, University of Chile, 2000.

[CN01] Edgar Chávez and Gonzalo Navarro. Towards measuring the searching complexity of metric spaces. In *Proc. Mexican Computing Meeting*, volume II, pages 969–978, Aguascalientes, M'exico, 2001. Sociedad Mexicana de Ciencias de la Computación.

[CN03] Edgar Chávez and Gonzalo Navarro. Probabilistic proximity search: fighting the curse of dimensionality in metric spaces. *Inf. Process. Lett.*, 85(1):39–46, January 2003.

[CN05] Edgar Chávez and Gonzalo Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.

[CNBYM01] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.

[Cod70] Edgar F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[CP00] Paolo Ciaccia and Marco Patella. Pac nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. *Data Engineering, International Conference on*, 0:244, 2000.

[CP02] Paolo Ciaccia and Marco Patella. Searching in metric spaces with user-defined and approximate distances. *ACM Trans. Database Syst.*, 27(4):398–437, December 2002.

[CP10] Paolo Ciaccia and Marco Patella. Approximate and probabilistic methods. *SIGSPATIAL Special*, 2(2):16–19, July 2010.

[CPZ97] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proc. of the 23rd Conference on Very Large Databases (VLDB'97)*, pages 426–435, 1997.

[CRR09] Edgar Chávez, Nora Reyes, and Patricia Roggero. Delayed insertion strategies in dynamic metric indexes. In *Chilean Computer Science Society (SCCC), 2009 International Conference of the*, pages 34–42, Nov 2009.

[CSO+15] Luiz Olmes Carvalho, Lucio F. D. Santos, Willian D. Oliveira, Agma Juci Machado Traina, and Jr. Traina, Caetano. Similarity joins and beyond: An extended set of binary operators with order. In Giuseppe Amato, Richard Connor, Fabrizio Falchi, and Claudio Gennaro, editors, *Similarity Search and Applications*, volume 9371 of *Lecture Notes in Computer Science*, pages 29–41. Springer International Publishing, 2015.

[DD09] Michel Marie Deza and Elena Deza. *Encyclopedia of Distances*. Springer Berlin Heidelberg, 2009.

[DGSZ03a] Vlatislav Dohnal, Claudio Gennaro, Pasquale Savino, and Pavel Zezula. D-index: Distance searching index for metric data sets. *Multimedia Tools and Applications*, 21(1):9–33, 2003.

[DGSZ03b] Vlatislav Dohnal, Claudio Gennaro, Pasquale Savino, and Pavel Zezula. Similarity join in metric spaces. In *Proc. 25th European Conf. on IR Research (ECIR'03)*, LNCS 2633, pages 452–467, 2003.

[DGZ03] Vlatislav Dohnal, Claudio Gennaro, and Pavel Zezula. Similarity join in metric spaces using eD-index. In *Proc. 14th Intl. Conf. on Database and Expert Systems Applications (DEXA'03)*, LNCS 2736, pages 484–493, 2003.

[DN87] Frank Dehne and Hartmut Noltemeier. Voronoi trees and clustering problems. *Information Systems*, 12(2):171–175, 1987.

[Doh04] Vlatislav Dohnal. An access structure for similarity search in metric spaces. In *EDBT Workshops*, pages 133–143, 2004.

[DS01] Jens-Peter Dittrich and Bernhard Seeger. Gess: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces. In *KDD*, pages 47–56, 2001.

[EN10] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. Addison-Wesley Publishing Company, USA, 6th edition, 2010.

[FB15] Kimmo Fredriksson and Billy Braithwaite. Quicker range- and k-nn joins in metric spaces. *Information Systems*, 52:189 – 204, 2015. Special Issue on Selected Papers from {SISAP} 2013.

[FKM+07] Fabrizio Falchi, Mouna Kacimi, Yosi Mass, Fausto Rabitti, and Pavel Zezula. SAPIR: Scalable and distributed image searching. In *SAMT (Posters and Demos)*, volume 300 of *CEUR Workshop Proceedings*, pages 11–12, Genoa, Italy, November 2007. CEUR-WS.org.

[FNC07] Karina Figueroa, Gonzalo Navarro, and Edgar Chávez. Metric spaces library, 2007. Available at `http://www.sisap.org/Metric_Space_Library.html`.

[FRZ97] Michael J. Folk, Greg Riccardi, and Bill Zoellick. *File Structures: An Object-Oriented Approach with C++*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1997.

[FTJF01] Roberto F. Santos Filho, Agma J. M. Traina, Caetano Traina Jr., and Christos Faloutsos. Similarity search without tears: The OMNI family of all-purpose access methods. In Georgakopoulos and Buchmann [GB01], pages 623–630.

[GB01] Dimitrios Georgakopoulos and Alexander Buchmann, editors. *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*. IEEE Computer Society, 2001.

[GCL+15] Yunjun Gao, Lu Chen, Xinhan Li, Bin Yao, and Gang Chen. Efficient k-closest pair queries in general metric spaces. *The VLDB Journal*, 24(3):415–439, 2015.

[GCMR08] Veronica Gil-Costa, Mauricio Marin, and Nora Reyes. An empirical evaluation of a distributed clustering-based index for metric space databases. In *Similarity Search and Applications, 2008. SISAP 2008. First International Workshop on*, pages 95–102, April 2008.

[GCMR09] Veronica Gil-Costa, Mauricio Marin, and Nora Reyes. Parallel query processing on distributed clustering indexes. *Journal of Discrete Algorithms*, 7(1):3 – 17, 2009. Selected papers from the 1st International Workshop on Similarity Search and Applications (SISAP).

[GR93] Igal Galperin and Ronald L. Rivest. Scapegoat trees. In *SODA '93: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 165–174, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[Gut84] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[Har95] Donna Harman. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conf. (TREC-3)*, pages 1–19, 1995. NIST Special Publication 500-207.

[Het09] Magnus Hetland. The basic principles of metric indexing. In Carlos Coello, Satchidananda Dehuri, and Susmita Ghosh, editors, *Swarm Intelligence for Multi-objective Problems in Data Mining*, volume 242 of *Studies in Computational Intelligence*, pages 199–232. Springer Berlin / Heidelberg, 2009.

[HKR93] Daniel P. Huttenlocher, Gregory A. Klanderman, and William J. Rucklidge. Comparing images using the hausdorff distance. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 15(9):850–863, Sep 1993.

[HS98] Gísli R. Hjaltason and Hanan Samet. Incremental distance join algorithms for spatial databases. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD Conference*, pages 237–248. ACM Press, 1998.

[HS00] Gísli R. Hjaltason and Hanan Samet. *Incremental Similarity Search in Multimedia Databases*. Number CS-TR-4199 in Computer science technical report series. Computer Vision Laboratory, Center for Automation Research, University of Maryland, 2000.

[HS03a] Gísli R. Hjaltason and Hanan Samet. Improved search heuristics for the sa-tree. *Pattern Recognition Letters*, 24(15):2785–2795, 2003.

[HS03b] Gísli R. Hjaltason and Hanan Samet. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems*, 28(4):517–580, 2003.

[JKF13] Shichao Jin, Okhee Kim, and Wenya Feng. MX-tree: A double hierarchical metric index with overlap reduction. In *Proc. ICCSA*, LNCS 7975, pages 574–589. Springer, 2013.

[JOT+05] Hosagrahar V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM Transactions on Database Systems*, 30(2):364–397, June 2005.

[JS08] Edwin H. Jacox and Hanan Samet. Metric space similarity joins. *ACM Trans. Database Syst.*, 33(2), 2008.

[JTSF00] Caetano Traina Jr., Agma J. M. Traina, Bernhard Seeger, and Christos Faloutsos. Slim-trees: High performance metric trees minimizing overlap between nodes. In Carlo Zaniolo, Peter C. Lockemann, Marc H. Scholl, and Torsten Grust, editors, *EDBT*, volume 1777 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 2000.

[KL04] Robert Krauthgamer and James R. Lee. Navigating nets: simple algorithms for proximity search. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '04, pages 798–807, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[KM83] Iraj Kalantari and Gerard McDonald. A data structure and an algorithm for the nearest point problem. *IEEE Transactions on Software Engineering*, 9(5), 1983.

[KP03] Dmitri V. Kalashnikov and Sunil Prabhakar. Similarity join for low-and high-dimensional data. In *Proceedings of the Eighth International Conference on Database Systems for Advanced Applications*, DASFAA '03, pages 7–, Washington, DC, USA, 2003. IEEE Computer Society.

[KR02] David Karger and Mathias Ruhl. Finding nearest neighbors in growth-restricted metrics. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 741–750, New York, NY, USA, 2002. ACM Press.

[KTA11] Hisashi Kurasawa, Atsuhiro Takasu, and Jun Adachi. Finding the k-closest pairs in metric spaces. In *Proceedings of the 1st Workshop on New Trends in Similarity Search*, NTSS '11, pages 8–13, New York, NY, USA, 2011. ACM.

[Lev65] Vladimir I. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8–17, 1965.

[Lev66] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[LL00] Mario A. Lopez and Swanwa Liao. Finding k-closest-pairs efficiently for high dimensional data. In *CCCG*, 2000.

[LMČS14] Jakub Lokoč, Juraj Moško, Přemysl Čech, and Tomáš Skopal. On indexing metric spaces using cut-regions. *Information Systems*, 43(0):1 – 19, 2014.

[LR96] Ming-Ling Lo and Chinya V. Ravishankar. Spatial hash-joins. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, SIGMOD '96, pages 247–258, New York, NY, USA, 1996. ACM.

[Mam05] Margarida Mamede. Recursive lists of clusters: A dynamic data structure for range queries in metric spaces. In *Proc. 20th Intl. Symp. on Computer and Information Sciences (ISCIS'05)*, LNCS 3733, pages 843–853, 2005.

[MCPR13] Natalia Miranda, Edgar Chávez, María Fabiana Piccoli, and Nora Reyes. (very) fast (all) k-nearest neighbors in metric and non metric spaces without indexing. In Nieves Brisaboa, Oscar Pedreira, and Pavel Zezula, editors, *Similarity Search and Applications*, volume 8199 of *Lecture Notes in Computer Science*, pages 300–311. Springer Berlin Heidelberg, 2013.

[MOV94] Luisa Micó, José Oncina, and Enrique Vidal. A new version of the nearest-neighbor approximating and eliminating search (AESA) with linear preprocessing-time and memory requirements. *Pattern Recognition Letters*, 15:9–17, 1994.

[MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[Nav99] Gonzalo Navarro. Searching in metric spaces by spatial approximation. In *Proc. String Processing and Information Retrieval (SPIRE'99)*, pages 141–148. IEEE CS Press, 1999.

[Nav01] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[Nav02] Gonzalo Navarro. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal (VLDBJ)*, 11(1):28–46, 2002.

[NH12] Bilegsaikhan Naidan and Magnus Lie Hetland. Static-to-dynamic transformation for metric indexing structures. In Navarro and Pestov [NP12], pages 101–115.

[NN97] Sameer A. Nene and Shree K. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 19(9):989–1003, 1997.

[NP12] Gonzalo Navarro and Vladimir Pestov, editors. *Similarity Search and Applications - 5th International Conference, SISAP 2012, Toronto, ON, Canada, August 9-10, 2012. Proceedings*, volume 7404 of *Lecture Notes in Computer Science*. Springer, 2012.

[NPRB16] Gonzalo Navarro, Rodrigo Paredes, Nora Reyes, and Cristian Bustos. An empirical evaluation of intrinsic dimension estimators. *Information Systems*, 2016.

[NR02] Gonzalo Navarro and Nora Reyes. Fully dynamic spatial approximation trees. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE 2002)*, LNCS 2476, pages 254–270. Springer, 2002.

[NR08] Gonzalo Navarro and Nora Reyes. Dynamic spatial approximation trees. *Journal of Experimental Algorithmics*, 12:1.5:1–1.5:68, June 2008.

[NR09] Gonzalo Navarro and Nora Reyes. Dynamic spatial approximation trees for massive data. In Skopal and Zezula [SZ09], pages 81–88.

[NR14] Gonzalo Navarro and Nora Reyes. Dynamic list of clusters in secondary memory. In Agma Juci Machado Traina, Jr. Traina, Caetano, and Robson Leonardo Ferreira Cordeiro, editors, *Similarity Search and Applications*, volume 8821 of *Lecture Notes in Computer Science*, pages 94–105. Springer International Publishing, 2014.

[NR16] Gonzalo Navarro and Nora Reyes. New dynamic metric indices for secondary memory. *Information Systems*, 2016.

[NU11] Gonzalo Navarro and Roberto Uribe. Fully dynamic metric access methods based on hyperplane partitioning. *Information Systems*, 36(4):734–747, 2011.

[OL81] Mark H. Overmars and Jan Van Leeuwen. Two general methods for dynamizing decomposable searching problems. *Computing*, 26:155–166, 1981.

[PC09] Marco Patella and Paolo Ciaccia. Approximate similarity search: A multi-faceted problem. *J. Discrete Algorithms*, 7(1):36–48, 2009.

[PCFN06] Rodrigo Paredes, Edgar Chávez, Karina Figueroa, and Gonzalo Navarro. Practical construction of k-nearest neighbor graphs in metric spaces. In *Proc. 5th Intl. Workshop on Experimental Algorithms (WEA'06)*, LNCS 4007, pages 85–97. Springer, 2006.

[PMCV06] José Peñarrieta, Patricio Morriberón, and Ernesto Cuadros-Vargas. Distributed spatial approximation tree - sat$*$. In *Proc. XXXII Latin American Conference on Informatics (CLEI 2006)*, page 155, Universidad de Santiago, August 2006. CD-ROM, ISBN 956-303-028-1.

[PR08] Rodrigo Paredes and Nora Reyes. List of twin clusters: a data structure for similarity joins in metric spaces. In *Proc. 1st Intl. Workshop on Similarity Search and Applications (SISAP'08)*, pages 131–138. IEEE Computer Society Press, 2008.

[PR09] Rodrigo Paredes and Nora Reyes. Solving similarity joins and range queries in metric spaces with the list of twin clusters. *Journal of Discrete Algorithms (JDA)*, 7:18–35, March 2009. doi:10.1016/j.jda.2008.09.012.

[RSC+13] Guillermo Ruiz, Francisco Santoyo, Edgar Chávez, Karina Figueroa, and Eric S. Tellez. Extreme pivots for faster metric indexes. In Nieves Brisaboa, Oscar Pedreira, and Pavel Zezula, editors, *Similarity Search and Applications*, volume 8199 of *Lecture Notes in Computer Science*, pages 115–126. Springer Berlin Heidelberg, 2013.

[RWW09] R.Tyrrell Rockafellar, Roger J.B. Wets, and Maria Wets. *Variational Analysis*. Grundlehren der mathematischen Wissenschaften. Springer Berlin Heidelberg, 2009.

[Sam84] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[Sam05] Hanan Samet. *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, 2005.

[SCO+15] Lucio F. D. Santos, Luiz Olmes Carvalho, Willian D. Oliveira, Agma J.M. Traina, and Jr. Traina, Caetano. Diversity in similarity joins. In Giuseppe Amato, Richard Connor, Fabrizio Falchi, and Claudio Gennaro, editors, *Similarity Search and Applications*, volume 9371 of *Lecture Notes in Computer Science*, pages 42–53. Springer International Publishing, 2015.

[Sko04] Tomáš Skopal. Pivoting m-tree: A metric access method for efficient similarity search. In Václav Snášel, Jaroslav Pokorný, and Karel Richta, editors, *DATESO*, volume 98 of *CEUR Workshop Proceedings*, pages 27–37. CEUR-WS.org, 2004.

[SPC13] Yasin N. Silva, Spencer S. Pearson, and Jason A. Cheney. Database similarity join for metric spaces. In Nieves Brisaboa, Oscar Pedreira, and Pavel Zezula, editors, *Similarity Search and Applications*, volume 8199 of *Lecture Notes in Computer Science*, pages 266–279. Springer Berlin Heidelberg, 2013.

[SPCR15] Yasin N. Silva, Spencer S. Pearson, Jaime Chon, and Ryan Roberts. Similarity joins: Their implementation and interactions with other database operators. *Information Systems*, 52:149 – 162, 2015. Special Issue on Selected Papers from {SISAP} 2013.

[SPS04] Tomáš Skopal, Jaroslav Pokorný, and Václav Snášsel. PM-tree: Pivoting metric tree for similarity search in multimedia databases. In *ADBIS (Local Proceedings)*, 2004.

[SR06] Uri Shaft and Raghu Ramakrishnan. Theory of nearest neighbors indexability. *ACM Trans. Database Syst.*, 31(3):814–838, September 2006.

[SWY75] Gerard Salton, Andrew Wong, and Chungshu Yang. A vector space model for automatic indexing. *Comm. of the ACM (CACM)*, 18(11):613–620, 1975.

[SZ09] Tomáš Skopal and Pavel Zezula, editors. *Second International Workshop on Similarity Search and Applications, SISAP 2009, 29-30 August 2009, Prague, Czech Republic*. IEEE Computer Society, 2009.

[Uhl91a] Jeffrey K. Uhlmann. Implementing metric trees to satisfy general proximity/similarity queries. In *Proc. Command and Control Symposium*, Washington, DC, 1991. Also published as Code 5570 NRL Memo Report (1991) at the Naval Research Laboratory.

[Uhl91b] Jeffrey K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.

[UN03] Roberto Uribe and Gonzalo Navarro. Una estructura dinámica para búsqueda en espacios métricos. In *Actas del XI Encuentro Chileno de Computación, Jornadas Chilenas de Computación*, Chillán, Chile, 2003. In Spanish. In CD-ROM.

[UP05] Roberto Uribe-Paredes. Manipulación de estructuras métricas en memoria secundaria. Master's thesis, Universidad de Chile, 2005. In Spanish. Gonzalo Navarro, advisor.

[UPN09] Roberto Uribe-Paredes and Gonzalo Navarro. Egnat: A fully dynamic metric access method for secondary memory. In Skopal and Zezula [SZ09], pages 57–64.

[Ver95] Knut Verbarg. The C-Tree: a dynamically balanced spatial index. *Computing Suppl.*, 10:323–340, 1995.

[Vid86] Enrique Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.

[Vit08] Jeffrey Scott Vitter. *Algorithms and Data Structures for External Memory*. Now Publishers Inc., Hanover, MA, USA, 2008.

[Wan10] Wei Wang. Similarity joins as stronger metric operations. *SIGSPATIAL Special*, 2(2):24–27, July 2010.

[WJ96] David A. White and Ramesh Jain. Similarity indexing: Algorithms and performance. In *Storage and Retrieval for Image and Video Databases (SPIE)*, pages 62–73, 1996.

[WS90a] Jason Tsong-Li Wang and Dennis Shasha. Query processing for distance metrics. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, pages 602–613, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[WS90b] Jason Tsong-Li Wang and Dennis Shasha. Query processing for distance metrics. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, pages 602–613, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[Yia93] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. 4th ACM-SIAM Symposium on Discrete Algorithms (SODA'93)*, pages 311–321, 1993.

[Yia99] Peter N. Yianilos. Excluded middle vantage point forests for nearest neighbor search. In *DIMACS Implementation Challenge, ALENEX'99*, Baltimore, MD, 1999.

[ZADB06] Pavel Zezula, Giussepe Amato, Vlatislav Dohnal, and Michal Batko. *Similarity Search: The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Springer, 2006.