UNIVERSIDAD DEL BÍO-BÍO

Facultad de Ciencias Empresariales
Departamento de Ciencias de la Computación y
Tecnologías de la Información

Cálculo de la consulta Skyline sobre Estructuras de Datos Compactas.

Por
Martita Paulina Muñoz Candia

Tesis para optar al grado de Magíster
en Ciencias de la Computación

Dirigido por:
Dr. Gilberto Gutiérrez
Universidad del Bío-Bío, Chillán, Chile

Co-Dirigido por:
Dr. Rodrigo Torres
Universidad del Bío-Bío, Chillán, Chile

2018 - II

A mi mamá, a mi familia y a mis amigos.

Agradecimientos

Quisiera comenzar agradeciendo a Dios por permitir lograr esta oportunidad de poder estudiar este postgrado en esta universidad, y lograr llegar hasta esta instancia de la vida.

También agradecer a mis directores de tesis. Realmente aprendí mucho trabajando con el profesor Gilberto y el profesor Rodrigo. Ambos fueron un aporte trascendental durante el desarrollo de esta tesis, quienes me ayudaron y guiaron en este camino que recién comienzo en el mundo de la investigación y divulgación científica.

Además, deseo agradecer a los demás profesores que participaron en mi formación académica, tanto cuando estudiaba mi carrera universitaria como en magíster que estoy concluyendo. El camino recorrido en estos años amparada por la Universidad del Bío Bío y sus docentes me han entregado las herramientas para ir poco a poco hasta llegar a esta importante instancia.

Quiero agradecer además a mi familia, especialmente a mi mamá, quien siempre ha estado a mi lado apoyándome en este camino. Mis amigos tampoco pueden faltar; gracias a ellos este camino se ha hecho más ameno y menos tedioso.

Finalmente, ha sido un largo camino para llegar a este punto, lleno de momentos únicos, en donde he logrado desarrollarme como persona y como investigadora. Pero nada podría haber logrado sin todos quienes me han acompañado y apoyado a lo largo de este trayecto.

Resumen

El operador (o consulta) Skyline ha recibido una importante atención en la comunidad de bases de datos. Dada una relación de dominancia en un conjunto de datos, esta consulta devuelve los objetos que no pueden ser dominados por ningún otro objeto. En la literatura han surgido diferentes algoritmos que resuelven esta consulta frente a datos almacenados en memoria secundaria. De todos los algoritmos presentados, *Branch and Bound Skyline* (BBS) es el más eficiente. Es un algoritmo que responde la consulta usando datos indexados en un R-tree, índice que también funciona sobre datos en memoria secundaria.

A raíz de la consulta Skyline, han surgido distintas variantes, como *Ranked Skyline query*, *Constrained Skyline Query* o *Dynamic Skyline Query*. De acuerdo a lo descrito en la literatura, es posible adaptar BBS para responder cada una de estas variantes sobe datos almacenados en memoria secundaria.

Por otro lado, el uso de las Estructuras de Datos Compactas ha tenido un auge en este último tiempo, debido a que permite reducir la cantidad de memoria utilizada para representar los datos, y además permite poder operar sobre ella sin la necesidad de ser descomprimida. La compactación es una propiedad muy importante debido a que permite manejar una cantidad mayor de datos en memoria principal, que se caracteriza por ser más rápida pero más pequeña, y evita trabajar en memoria secundaria, que se caracteriza por ser más grande pero hasta 10000 veces más lenta. De entre estas estructuras, el k^2-tree es una estructura diseñada para representar grafos Web, pero que además es muy útil para representar relaciones binarias.

En esta tesis se proponen cuatro algoritmos: dos que permite responder la consulta Skyline sobre datos almacenados en un k^2-tree de manera eficiente, otro que cuenta la cantidad de puntos contenidos dentro de un rango de puntos indexados en un k^2-tree, y que es utilizado en el último algoritmo que responde una de la variante: *Enumerating Skyline*. La complejidad de los algoritmos oscila entre $O(n \log_k n)$ y $O(kn^2 \log_k n)$, en donde n es el tamaño de la matriz que representa el k^2-tree. Por su parte, la cantidad de memoria utilizada oscila entre $O(n \log_k n)$ y $O(kn \log_k n)$. Cabe destacar que estos casos reflejan una cota superior que en la práctica no suele ser alcanzada.

Todos los algoritmos presentados fueron evaluados experimentalmente, midiendo tanto el tiempo de ejecución como la memoria requerida, y usando datos con distribución uniforme y clusterizada. Tras los experimentos se pudo comprobar que todos los algoritmos presentan una reducción del tiempo de respuesta en comparación con las propuestas de la literatura, a cambio de un uso adicional de memoria. También se pudo comprobar que existe una reducción significativa del uso de recursos cuando los datos presentan una distribución clusterizada.

Índice

Agradecimientos II

Resumen III

Índice VII

I Motivación. 1

1. Introducción. 2
1.1. Motivación. . . . 2
1.2. Hipótesis y objetivos. . . . 4
1.3. Alcance de la investigación. . . . 5
1.4. Metodología de trabajo. . . . 5
1.5. Contribución. . . . 6
1.6. Contenidos. . . . 7

II Trabajo Relacionado. 8

2. Estructuras de Datos Compactas. 9
2.1. Introducción. . . . 9
2.2. K^2-tree. . . . 10
2.2.1. Construcción del k^2-tree. . . . 12
2.2.2. Encontrar el hijo de un nodo en el k^2-tree. . . . 12
2.2.3. Operaciones sobre el k^2-tree. . . . 13
2.2.4. Contar puntos dentro de un rango. . . . 13

3. Consulta Skyline. 14
3.1. Definición y propiedades . . . 14
3.2. Algoritmos que responden la consulta Skyline. . . . 16
3.2.1. Block Nested Loop (BNL). . . . 18

3.2.2. Dividir para conquistar (D&C). . . . 19
3.2.3. Bitmap. . . . 20
3.2.4. Index. . . . 22
3.2.5. Nearest neighbor (NN). . . . 23
3.2.6. Branch and Bound Skyline (BBS). . . . 24
3.3. Evaluación de los algoritmos. . . . 25

4. Variantes del operador Skyline. 28
4.1. Constrained Skyline Query. . . . 28
4.2. Dynamic Skyline Query (DSQ). . . . 28
4.2.1. Spatial Skyline Query (SSQ). . . . 29
4.3. Reverse Skyline Query (RSQ). . . . 30
4.4. Ranked Skyline Query. . . . 30
4.5. Enumerating query y K-Dominating query. . . . 30
4.5.1. Enumerating Query. . . . 31
4.5.2. K-Dominating query. . . . 31
4.6. K-Skyband Query. . . . 32
4.7. Group-by Skyline. . . . 32
4.8. Resumen de las variantes. . . . 33

III Algoritmos propuestos. 34

5. Consulta Skyline en un k^2-tree. 35
5.1. Introducción. . . . 35
5.2. Antecedentes. . . . 35
5.3. Algoritmo propuesto: *BBS k^2-tree* (BBSk). . . . 36
5.3.1. Ejemplo. . . . 38
5.4. Análisis teórico. . . . 40
5.4.1. Condiciones de descarte. . . . 40
5.4.2. Tamaño del heap. . . . 42
5.4.3. Análisis temporal. . . . 44
5.4.4. Análisis espacial. . . . 47
5.5. Una nueva alternativa: *Skyline X*. . . . 48
5.5.1. Ejemplo. . . . 49
5.5.2. Análisis teórico. . . . 50
5.6. Conclusión. . . . 51

6. Contar puntos dentro de un rango en un k^2-tree. 53
6.1. Introducción. . . . 53
6.2. Antecedentes. . . . 53
6.3. Algoritmo propuesto: *Compact Count*. . . . 54

6.3.1. Un ejemplo. . . . 56
6.4. Análisis teórico. . . . 58
6.4.1. Contar puntos dentro de una submatriz. . . . 58
6.4.2. Análisis temporal. . . . 58
6.4.3. Análisis espacial. . . . 60
6.5. Conclusión. . . . 61

7. Variantes de la consulta Skyline. 62
7.1. Introducción. . . . 62
7.2. Enumerating Skyline. . . . 62
7.2.1. Algoritmo propuesto: *Enumerating Compact*. . . . 63
7.2.2. Un ejemplo. . . . 65
7.2.3. Análisis teórico. . . . 66
7.3. Constrained Skyline Query. . . . 68
7.4. Conclusión 70

8. Conclusiones de los algoritmos propuestos. 71

IV Implementación y Experimentación. 73

9. *BBS k^2-tree* (BBSk). 74
9.1. Implementación. . . . 74
9.2. Datos experimentales. . . . 75
9.3. Línea de comparación (Baseline). . . . 76
9.4. Entorno de pruebas. . . . 77
9.5. Resultados experimentales. . . . 77
9.5.1. Resultados experimentales: tiempo de ejecución. . . . 77
9.5.2. Resultados experimentales: memoria requerida. . . . 81
9.6. Conclusiones. . . . 82

10. *Compact Count*. 84
10.1. Implementación. . . . 84
10.2. Datos experimentales. . . . 84
10.3. Línea de comparación (Baseline). . . . 85
10.4. Entorno de pruebas. . . . 85
10.5. Resultados experimentales. . . . 85
10.5.1. Resultados experimentales: tiempo de ejecución. . . . 85
10.5.2. Resultados experimentales: memoria requerida. . . . 87
10.6. Conclusiones. . . . 87

11. *Enumerating Compact*. 89

11.1. Implementación. 89
11.2. Datos experimentales y entorno de pruebas. 89
11.3. Línea de comparación (Baseline). 90
11.4. Resultados experimentales. 90
11.4.1. Resultados experimentales: tiempo de ejecución. 90
11.4.2. Resultados experimentales: memoria requerida. 92
11.5. Conclusiones. 93

12. Conclusiones de los resultados experimentales. **95**

V Conclusiones y Trabajo Futuro. **97**

13. Conclusiones y Trabajo Futuro. **98**
13.1. Conclusiones. 98
13.2. Trabajo futuro. 100

Bibliografía **101**

Parte I

Motivación.

Capítulo 1

Introducción.

1.1. Motivación.

Suponga que una persona decide viajar de vacaciones a la playa durante una semana. Para ello, a la hora de buscar un hotel, encuentra varias opciones disponibles. Para escoger el mejor hotel de entre la lista, deberá escoger los criterios con los cuales decidirá cual es el mejor. En este caso suponga que la persona prefiere el hotel que se encuentre más cerca de la orilla de la playa, y a su vez sea el más barato. Estas son dos preferencias contradictorias debido a que un cuarto de hotel suele ser más caro en la medida en que se encuentre más cerca a la playa. Por lo tanto, la persona deberá conseguir lograr satisfacer ambas preferencias de tal forma de encontrar el mejor hotel en relación con sus preferencias.

Existen múltiples situaciones en que los usuarios deben tomar una decisión frente a una enorme cantidad de información, considerando una importante cantidad de criterios muchas veces contradictorios. Por ejemplo, escoger el mejor empleado de una empresa considerando la cantidad de horas que trabaja y la cantidad de trabajo que realiza; escoger la mejor vivienda a comprar considerando el precio y la distancia a una estación del metro; escoger la mejor ubicación para una sucursal considerando aspectos estratégicos de marketing y económicos; entre otros casos. Es así como como los usuarios deberán encontrar la mejor opción (o mejores opciones) de entre una enorme cantidad de información existente, utilizando una serie de criterios multidimensionales que deberán ser unificados para encontrar la mejor respuesta.

El rápido crecimiento de los sistemas de apoyo a la toma de decisiones multicriterio y el tamaño cada vez mayor de los datos multidimensionales ha llevado a los investigadores a buscar nuevos métodos eficientes para el procesamiento de datos con el fin de recuperar resultados útiles [23]. Muchos de estos sistemas se caracterizan por varios aspectos [47]:

- La consulta generalmente se basa en criterios múltiples y, a veces, contradictorios.
- A diferencia de los sistemas convencionales, puede que no haya una sola respuesta óptima.
- Para la misma consulta, diferentes usuarios, dictados por sus preferencias personales, pueden encontrar diferentes respuestas atractivas.

El operador (o consulta) Skyline se ha convertido en un área importante en la investigación de bases de datos para extraer objetos interesantes de conjuntos de datos multidimensionales [37].

Dado un conjunto de puntos, la consulta Skyline devuelve todos aquellos puntos que no son dominados por otros puntos. Un punto domina otro punto si es tan bueno o mejor en todas las dimensiones y mejor en al menos una dimensión [15]. La popularidad del operador de Skyline se debe principalmente a la simplicidad del paradigma y su aplicabilidad en el soporte de decisiones de múltiples criterios con respecto a las preferencias del usuario [23].

Esta consulta ha estado relacionada con otros problemas como *Top-k*, que recupera los mejores k puntos que minimizan una función de preferencia específica; los k vecinos más cercanos o *K-NN*, que responde con los k puntos más cercanos a un punto de consulta p; y *Convex Hull*, que responde con el polígono convexo mínimo que encierra a un conjunto de puntos [23].

Recientemente, algunas investigaciones se han centrado en cómo utilizar los puntos de Skyline en la toma de decisiones en el área de los negocios [38]. También se puede aplicar en economía [30] donde puede apoyar la extracción de datos microeconómicos.

Otra aplicación es la optimización de consultas en entornos distribuídos [20]. Esto puede ser particularmente útil en arquitecturas en la nube donde los datos se encuentran dispersos entre los servidores, o en el caso donde la calidad de los servicios web es el objetivo principal [1, 50, 21]. También puede ser útil en caso que los datos se encuentren distribuídos en redes P2P [20, 31, 48], y en redes inalámbricas [9, 46]. En este último caso, un objetivo adicional es la eficiencia energética de los sensores que se comunican de forma inalámbrica para responder la consulta [9].

La consulta Skyline también se puede usar para trabajar en un subespacio de atributos [47] con el fin de identificar el Skyline en un pequeño subconjunto de dimensiones dentro del espacio en donde se definen los datos. Esta consulta también es aplicable en una amplia variedad de tipos de datos, como atributos parcialmente ordenados [8, 22], incompletos [24], inciertos [2], o en *data stream* o flujo de datos [29].

En la literatura se han presentado diferentes tipos de algoritmos que responden esta consulta. Entre las estrategias presentadas, se encuentra preordenar el conjunto de puntos; o preprocesarlos utilizando alguna estructura o índice que almacene dichos puntos. Estas estrategias tienen por objetivo reducir la cantidad de comparaciones de dominancia entre los puntos del conjunto revisado y así reducir el costo de acceso a memoria secundaria y el costo de procesamiento [52, 22].

Por otro lado, las Estructuras de Datos Compactas son estructuras que buscan almacenar la mayor cantidad de información en el menor espacio posible. Lo importante es que estas estructuras puedan soportar consultas de interés sin descomprimir la información almacenada [16].

La compactación trae algunos beneficios como permitir mantener una cantidad mayor de información que en su representación original no cabría en memoria principal, y mejorar el rendimiento de la memoria caché permitiendo una mayor cantidad de información en ella [5].

Las Estructuras de Datos Compactas se utilizan para representar imágenes, relaciones en redes sociales, rasterdata (Mosaicos de datos), georreferencias, relaciones binarias, permutaciones de texto, diccionarios, sumas parciales, grafos web, secuencias de ADN, árboles binarios de decisión, filtros bloom, entre otras utilidades [40].

Las Estructuras de Datos Compactas pueden utilizarse en aplicaciones móviles, en donde la memoria es limitada y el acceso a internet también puede llegar a serlo. Suponga que la misma

persona que se encuentra en la playa desea recorrer los lugares de interés de la ciudad en la que se encuentra. No posee acceso a internet, pero si posee una aplicación móvil en su teléfono inteligente en donde tiene toda la información turística de esa ciudad. Una alternativa para almacenar toda esta información en un espacio tan reducido como la memoria de un teléfono inteligente son las Estructuras de Datos Compactas. Por otro lado, con la ayuda del operador Skyline, la persona podrá encontrar los lugares más interesantes para visitar. Por ejemplo, la persona puede consultar por el restaurante más recomendado y más barato, el museo más cercano y con mejor calificación, la estación del metro más cercana, entre otras consultas.

De forma particular, el uso de la Estructura de Datos Compacta k^2-tree promete ser una interesante propuesta para responder esta consulta. Fue presentada en [27] y fue diseñada originalmente para representar Grafos Web, pero que también es útil para representar relaciones binarias.

El k^2-tree es una estructura útil para responder la consulta Skyline porque permite aprovechar sus propiedades para reducir el número de comparaciones de dominancia, reduciendo así la cantidad de procesamiento. Por otro lado, al tratarse de una Estructura de Datos Compacta, permite almacenar una mayor cantidad de información utilizando una menor cantidad de memoria. Esto permite procesar una mayor cantidad de datos en memoria principal, reduciendo el acceso a memoria secundaria. Así por ejemplo, la información de los restaurantes con el precio de algunos de sus platos vs. la evaluación de los usuarios de la aplicación anteriormente mencionada puede estar almacenada en un k^2-tree y responder la consulta Skyline eficientemente.

En esta tesis se proponen, analizan, implementan y experimentan cuatro algoritmos: dos que permite responder la consulta Skyline sobre datos almacenados en un k^2-tree de manera eficiente, otro que cuenta la cantidad de puntos contenidos dentro de un rango de puntos indexados en un k^2-tree, y que es utilizado en el último algoritmo que responde una de las variantes de la consulta Skyline. La complejidad algorítmica en el peor de los casos oscila entre $O(n \log_k n)$ y $O(kn^2 \log_k n)$, en donde n es el tamaño de la matriz que representa el k^2-tree. Por su parte, la cantidad de memoria utilizada oscila entre $O(n \log_k n)$ y $O(kn \log_k n)$. Cabe destacar que estos casos reflejan una cota superior que en la práctica no suele ser alcanzada.

En este capítulo, se presentarán las siguientes secciones: en la Sección 1.2 se presenta la hipótesis y los objetivos; en la Sección 1.3 se presentan los alcances de la investigación; en la Sección 1.4 se presenta la metodología de trabajo aplicada en esta investigación; en la Sección 1.5 se muestran las contribuciones realizadas a partir de esta tesis; y finalmente en la Sección 1.6 se presentará la estructura del resto de este informe.

1.2. Hipótesis y objetivos.

Hipótesis.

El k^2-tree es una Estructura de Datos Compacta idónea para implementar y/o mejorar el rendimiento de la consulta Skyline y algunas de sus variantes.

Objetivo general.

Diseñar e implementar algoritmos eficientes que permitan responder la consulta Skyline y sus variantes en datos indexados en un k^2-tree.

Objetivos específicos.

- Adaptar e implementar las mejores estrategias y algoritmos presentes en la literatura respecto de la consulta Skyline sobre el k^2-tree.
- Diseñar e implementar un algoritmo que resuelva la consulta Skyline sobre datos indexados en un k^2-tree y que sea una mejora de los algoritmos implementados en el punto anterior o uno completamente nuevo.
- Evaluar el desempeño, tanto en rendimiento como en almacenamiento, de los algoritmos implementados.
- Explorar algunas de las variantes de la consulta Skyline y adaptar los principales algoritmos propuestos en la literatura que resuelvan algunas de ellas en un k^2-tree.

1.3. Alcance de la investigación.

Los algoritmos se han implementado bajo el lenguaje de programación C y C++, con las cuales se creará una librería que quedará disponible a la comunidad científica.

Para el proceso experimental de los algoritmos, se han usado datos sintéticos uniformes y clusterizados, para así verificar el comportamiento que pueden conseguir estos algoritmos sobre una u otra distribución. También se ha trabajado solamente en memoria principal, dejando afuera cualquier experimentación que involucre memoria secundaria.

Debido a la naturaleza de la estructura sobre la cual se encuentran indexado los datos, solo se contempla procesar y experimentar datos bidimensionales, quedando excluído el procesamiento de datos en 3 o más dimensiones.

1.4. Metodología de trabajo.

La metodología de trabajo utilizada en esta tesis consta de 6 etapas y que se describen a continuación:

1. **Revisión de la literatura.** Se ha realizado una exhaustiva revisión de la literatura analizando los algoritmos y estrategias existentes que logran responder la consulta Skyline. Como resultado de esta etapa se han escogido aquellos algoritmos que sean los más eficientes para su posterior implementación sobre el k^2-tree.
2. **Adaptación de los algoritmos seleccionados.** Una vez seleccionados aquellos algoritmos más eficientes encontrados en la literatura, se procedió a adaptar dichas estrategias para responder la consulta Skyline sobre datos indexados en un k^2-tree.

3. **Mejorar propuesta implementada.** Frente a los algoritmos adaptados en el k^2-tree, se procedió a buscar la forma de mejorarlo, o diseñar e implementar un nuevo algoritmo que resuelva la consulta. Lo anterior es con el propósito de aprovechar las ventajas de utilizar dicha estructura para resolver de manera eficiente la consulta.

4. **Explorar las variantes.** Se han explorado algunas de las variantes, en conjunto con los principales algoritmos presentes en la literatura responden dichas variantes, con el fin de adaptarlos e implementarlos para su uso en un k^2-tree.

5. **Análisis de los algoritmos propuestos.** Se ha desarrollado un análisis teórico de los algoritmos propuestos respecto al uso de recursos, tanto temporales como espaciales.

6. **Experimentación.** Se han realizado una serie de experimentos para evaluar el rendimiento de los algoritmos implementados, en términos de tiempo de ejecución y de almacenamiento requerido para resolver el problema.

1.5. Contribución.

Tras el desarrollo de esta tesis, se han logrado las siguientes contribuciones:

- Se proponen dos algoritmos que responde la consulta Skyline sobre datos indexados en un k^2-tree: BBSk^2-tree (BBSk) y *Skyline X*.
- Se propone un algoritmo que cuenta la cantidad de puntos dentro de un rango que trabaja en un k^2-tree: *Compact Count*, el cual es utilizado para responder la variante Enumerating Skyline.
- Se propone un algoritmo que responde la variante Enumerating Skyline sobre datos indexados en un k^2-tree: *Enumerating Compact*.
- Se implementan los algoritmos propuestos utilizando los lenguajes de programación C y C++.
- Se realizan las pruebas experimentales de los algoritmos implementados utilizando datos sintéticos, midiendo tanto el tiempo como la memoria que utilizan.
- Se presenta un análisis teórico de los algoritmos propuestos, tanto temporal como espacial.

De lo anterior se ha logrado la difusión de la investigación por medio de la participación en:

- Muñoz-Candia, M., Gutiérrez G., Torres R.: Cálculo del operador Skyline sobre Estructuras de Datos compactas. VII Encuentro de Investigación de Estudiantes de Postgrado. Universidad del Bío-Bío (2018).
- Está en desarrollo un artículo que será enviado próximamente a una revista científica todavía no definida.

1.6. Contenidos.

Este informe está dividido en cinco partes, las cuales a su vez están compuestas por uno o más capítulos. La Parte I, que corresponde a la motivación de esta tesis, está compuesta por un solo capítulo que es el que se está concluyendo en este punto.

La Parte II presenta el trabajo relacionado, mostrando la revisión de la literatura de esta tesis. En el Capítulo 2 está dedicado a describir las Estructuras de Datos Compactas, su funcionamiento y características, además de la principal estructura a utilizar en esta tesis: el k^2-tree. En el Capítulo 3 se define la consulta Skyline, sus propiedades y los algoritmos propuestos para responderla. Finalmente en el Capítulo 4 se muestran las principales variantes de la consulta Skyline encontradas en la literatura, su funcionamiento y los algoritmos y estrategias presentes en la literatura enfocados en resolverlas.

La Parte III se presentan y analizan las propuestas de esta tesis. En el Capítulo 5 se realiza un análisis de la consulta Skyline sobre un k^2-tree, presentando los dos algoritmos propuestos para responderla. En el Capítulo 6 se analiza el problema de contar puntos dentro de un rango para un conjunto indexado en un k^2-tree, presentando *Compact Count*. En el Capítulo 7 se ofrece una revisión de las variantes de la consulta Skyline, analizando dos variantes y presentando un algoritmo en concreto para responder con una de ellas. Finalmente, en el Capítulo 8 se muestran las conclusiones del análisis expuesto en esta parte.

La Parte IV presenta la implementación y experimentación de todos los algoritmos presentados. En el Capítulo 9 se explica la experimentación y los resultados de aquellos algoritmos que responden la consulta Skyline. En el Capítulo 10 se detalla la experimentación y resultados de los algoritmos que cuentan los puntos al interior de un rango. En el Capítulo 11 se describe la experimentación y resultados de aquellos algoritmos que responden la variante *Enumerating Skyline*. Finalmente en el Capítulo 12 se muestran las conclusiones tras los resultados experimentales encontrados.

La Parte V presenta las conclusiones y el trabajo futuro respecto de esta tesis, lo cual se ve detallado en el Capítulo 13.

Parte II

Trabajo Relacionado.

Capítulo 2

Estructuras de Datos Compactas.

Las Estructuras de Datos Compactas (o Estructuras de Datos Sucintas) son estructuras que representan datos (ya sean árboles, tablas, conjuntos, grafos, texto, etc.) utilizando el menor espacio posible, y que a su vez es capaz de resolver de forma eficiente las operaciones requeridas sobre los datos [27].

Una Estructuras de Datos Compactas esencial son los bitmaps, frente al cual se aplican dos operaciones básicas: *rank* y *select* y que será detallado en la Sección 2.1. Por otro lado, en la Sección 2.2 se describe el k^2-tree, la estructura de datos utilizada en esta tesis.

2.1. Introducción.

Un bitmap es un arreglo de bits que posee estructuras adicionales para responder las consultas de *rank* y *select* de manera eficiente [40]. Esta es una estructura básica que permite implementar otras Estructuras de Datos Compactas [27].

La operación *rank* se encarga de contar la cantidad de símbolos (ceros o unos) dentro de un bitmap hasta una cierta posición. Por otra parte la operación *select* indica en que posición se encuentra la i-ésimo símbolo (cero o uno) dentro de un bitmap [40].

De forma general, dada una secuencia de bits $B_{1,n}$, se pueden definir la operación *rank* como [27]:

Definición 2.1. *$Rank_b(B, i)$ devuelve la cantidad de bits b encontrados en el prefijo $B_{1,i}$.*

Por lo tanto, $Rank_0(B, i)$ entrega la cantidad de 0 que existen dentro de B entre la posición 1 e i, y $Rank_1(B, i)$ entrega la cantidad de 1 que existen en B entre la posición 1 e i.

La operación *select* se define como [27]:

Definición 2.2. *$Select_b(B, i)$ devuelve la posición en donde se ubica la i-ésima ocurrencia del bit b.*

Por lo tanto, $Select_0(B, i)$ responde con la posición en que se encuentra el i-ésimo 0, mientras que $Select_1(B, i)$ responde con la posición en que se encuentra el i-ésimo 1.

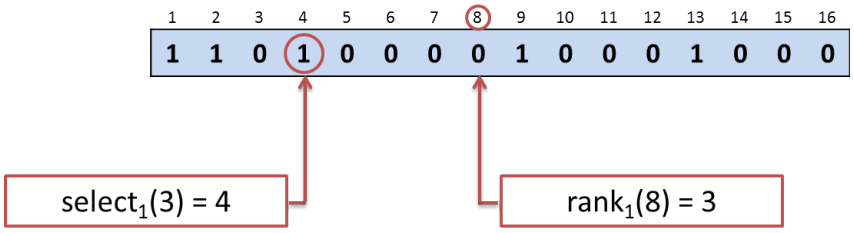


Figura 2.1: Ejemplo de un bitmap con sus operaciones escenciales [40].

En la Figura 2.1 se puede observar un ejemplo de un bitmap, en donde se aplican las dos operaciones ya vistas. En ella, cuando se desea saber cuántos unos hay hasta la posición 8, la consulta *rank* responde 3. Por otro lado, si se desea saber en qué posición se encuentra el tercer uno, la consulta *select* responde que el tercer uno se ubica en la posición 4.

Existe una extensa literatura referente a estas consultas [**?**, 32, 12, 35, 41, 19]. A grandes rasgos, *rank* es resuelto en un tiempo constante, mientras que *select* es resuelto en un tiempo de orden $O(\log \log n)$ utilizando un espacio de orden $n + o(n)$ [27]. El espacio es utilizado tanto para almacenar el bitmap como una estructura adicional que permite reducir el tiempo de respuesta de ambas consultas.

La estructura bitmap, en conjunto con sus dos consultas, *rank* y *select*, son la base para la construcción de otras Estructuras de Datos Compactas, como la que se revisará en la siguiente sección.

2.2. K^2-tree.

El k^2-tree es una estructura de datos para representar de forma compacta matrices binarias dispersas (que también se pueden considerar como cuadrículas de puntos) [34]. Originalmente fue presentada en [7] para representar grafos Web previamente indexado en una matriz de adyacencia. Su funcionamiento está basado en la estrategia seguida por los Quadtree [42]. Esta estructura está diseñada para comprimir grandes áreas de matriz con ceros [27]. Esto permite comprimir de forma más eficiente matrices cuya distribución sea clusterizada.

Sea una matriz $n \times n$, en donde n es potencia de k. La matriz es subdividida en k^2 submatrices de $\frac{n}{k} \times \frac{n}{k}$. Las submatrices se contabilizan de izquierda a derecha y de arriba hacia abajo. Cada k^2 submatriz es representada por un bit cuyo valor es 1 si contiene al menos un punto, y 0 si se encuentra completamente vacía. Aquellas submatrices representadas con un 1 vuelven a ser subdivididas en k^2 submatrices de forma recursiva. La subdivisión finaliza cuando se encuentra una submatriz completamente vacía o cuando se alcanzan las celdas individuales. En la Figura 2.2 muestra un ejemplo del uso del k^2-tree en donde $k = 2$ y $n = 4$.

En caso que n no sea potencia de k, la matriz puede ser extendida de $n \times n$ a $n' \times n'$ en donde n' es potencia de k, y los espacios vacíos producto de la extensión son completados con ceros. Por ejemplo, al tener una matriz de 11×11, puede ser extendida a una matriz de 16×16 en caso que $k = 2$ [7]. Un claro ejemplo se puede ver en la Figura 2.3.

Esta estructura es representada mediante el uso de dos bitmaps, llamados T y L. En T

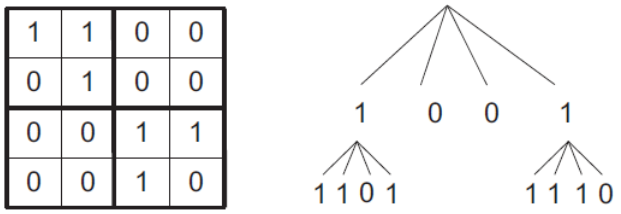


Figura 2.2: Ejemplo de un k^2-tree [7].

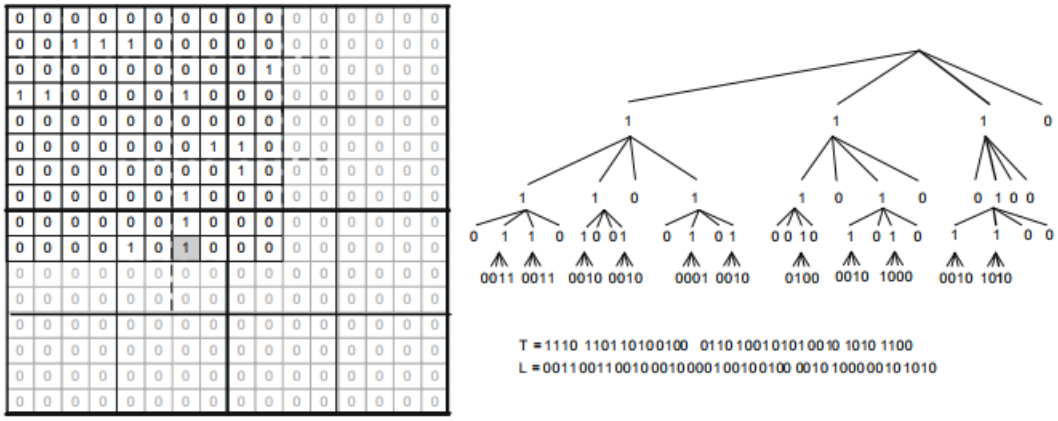


Figura 2.3: Ejemplo completo de un k^2-tree.

almacena todos los bits de los nodos excepto el último nivel, los cuales son almacenados en L. En la Figura 2.3 se puede revisar una implementación completa de un k^2-tree en donde se muestra la matriz binaria, el árbol conceptual y los dos bitmaps resultantes. Para almacenar los nodos en los bitmaps, se realiza un recorrido en anchura sobre el k^2-tree.

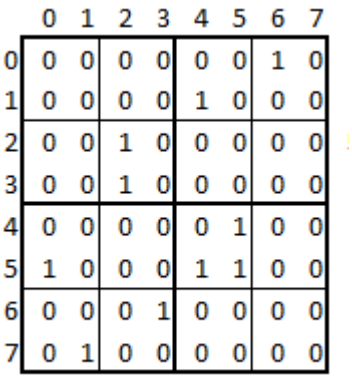
Para representar relaciones binarias como un plano cartesiano, el origen del eje de coordenadas se ubica en la esquina superior izquierda, mientras que las columnas representan el eje X, y las filas el eje Y, empezando a contar desde cero [34]. En la Figura 2.4 se puede observar un ejemplo de una matriz binaria de 8×8.

Para la implementación de algunas operaciones básicas que permiten navegar por el k^2-tree, se requiere la estructura necesaria para trabajar con la consulta *rank*, pero solo sobre el bitmap T. Esta es la causa de la separación de los nodos en dos bitmaps, puesto que no se requiere calcular *rank* sobre L [7].

El espacio que utiliza esta estructura para representar una matriz binaria en bits es de [7]:

$$k^2 m \left(\log_{k^2} \frac{n^2}{m} + O(1) \right)$$

En donde m es la cantidad de unos que tiene la matriz, n es el tamaño de la matriz a



	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	1	0
1	0	0	0	0	1	0	0	0
2	0	0	1	0	0	0	0	0
3	0	0	1	0	0	0	0	0
4	0	0	0	0	0	1	0	0
5	1	0	0	0	1	1	0	0
6	0	0	0	1	0	0	0	0
7	0	1	0	0	0	0	0	0

Figura 2.4: Representación de un plano cartesiano por un k^2-tree.

representar, y k es la división que se realiza sobre n para generar las submatrices.

2.2.1. Construcción del k^2-tree.

En [7, 27] se propone un algoritmo para construir el k^2-tree de forma recursiva a partir de la matriz de adyacencia. El tiempo que demora es de orden $O(n^2)$, mientras que el espacio utilizado es de $O\left(k^2 m \left(\log_{k^2} \frac{n^2}{m} + 1\right)\right)$, que es equivalente al tamaño de la estructura final [7].

Para reducir el tiempo de construcción, se propone usar una *lista de adyacencia* como formato para representar los puntos de entrada. De esta forma se puede construir el k^2-tree utilizando la misma cantidad de espacio, mientras que el tiempo utilizado se reduce a $O(\log m)$ por cada bit del árbol [7].

2.2.2. Encontrar el hijo de un nodo en el k^2-tree.

Para encontrar i-ésimo hijo de un nodo x del k^2-tree ($child_i(x)$) se obtiene de la siguiente manera [7]:

$$child_i(x) = rank_1(T, x)k^2 + i$$

sobre el bitmap $T : L$.

Por ejemplo, en la Figura 2.3, los primeros cuatro bits de T representan las cuatro primera submatrices en donde se dividió la matriz de adyacencia. Si se desea obtener el tercer hijo del nodo 2, se calcula:

$$child_3(2) = rank_1(T, 2) * 2^2 + 3 = 2 * 4 + 3 = 11$$

Y así se descubre que el tercer hijo del nodo 2 está ubicado en la posición 11 del arreglo T. Si se accede a esa posición, se descubre que el bit tiene el valor de 1, es decir, dicha submatriz no se encuentra vacía.

2.2.3. Operaciones sobre el k^2-tree.

En [7] se definen las operaciones para obtener los vecinos directos y reversos de un nodo de un grafo Web representado por esta matriz de adyacencia. En pocas palabras, la operación de los vecinos directos de un nodo p obtiene las celdas dentro de la fila p que tienen un 1. Por otro lado, los vecinos reversos de un nodo q obtiene las celdas dentro de la columna q que son 1. El análisis temporal indica que ambas operaciones pueden tener un orden temporal de $O(\sqrt{m})$ considerando una distribución uniforme de los puntos. En caso de encontrarse con una distribución clusterizada, estas operaciones pueden funcionar mucho mejor.

En [27] las operaciones sobre el k^2-tree son extendidas, con operaciones como *CheckLink* y *Range*. La primera es para conocer si la celda (p, q) se encuentra un 1, mientras que la segunda responde con las celdas que poseen un 1 dentro del rango $[p_1, p_2] \times [q_1, q_2]$.

Para resolver la consulta de *CheckLink* se requiere en el peor de los casos un tiempo de $O(\log_k n)$, debido a se requiere recorrer el árbol desde el nodo hasta la hoja en el caso en que exista un 1 en la celda (p, q).

2.2.4. Contar puntos dentro de un rango.

La operación *Count* responde con la cantidad de celdas que son 1 dentro del rango $[p_1, p_2] \times [q_1, q_2]$. En [27] se presenta *Range*, un algoritmo recursivo que identifica las coordenadas de todos los puntos que se encuentran al interior de un rango de consulta para datos indexados en un k^2-tree en un tiempo de orden $O(\log_k n)$ por cada punto al interior del rango. En [34] se explica que la única forma de contar los puntos dentro de un rango para datos indexados en un k^2-tree, es identificándolos uno por uno, es decir, utilizando *Range*.

En [13] se presenta un algoritmo que permite responder la operación *Count* sobre otra estructura de datos compactas llamada *Wavelet matrix*, basada en otra Estructura de Datos Compactas llamada *Wavelet tree* [33].

La operación *Count* puede llegar a tener varios usos como parte de otras consultas, como por ejemplo a la hora de resolver algunas variantes de la consulta Skyline como se verá más adelante. Como esta consulta, existen varias más las cuales requieren el uso de esta consulta, especialmente si se trabaja sobre datos indexados en alguna Estructura de Datos Compactas. Es por ello que es necesario diseñar un algoritmo que responda la consulta *Count* sobre datos indexados en un k^2-tree.

Capítulo 3

Consulta Skyline.

La consulta Skyline es una operación importante para muchas aplicaciones de toma de decisiones multicriterio, y ha recibido una importante atención en la comunidad de Base de Datos [17]. Es un popular y poderoso paradigma que permite extraer puntos de interés de un conjunto de datos multidimensionales [4].

En la Sección 3.1 se explicará la definición, funcionamiento y propiedades de esta consulta; mientras que en la Sección 3.2 se explicarán algunos de los algoritmos más importantes presentados en la literatura. Finalmente, en la Sección 3.3 se presentará una evaluación de todas las propuestas presentadas.

3.1. Definición y propiedades

Dado un conjunto de puntos, la consulta Skyline devuelve todos aquellos puntos que no son dominados por otros puntos. Un punto domina otro punto si es tan bueno o mejor en todas las dimensiones y mejor en al menos una dimensión [15]. La comparación entre dos puntos para saber cuál es mejor viene dada por las preferencias o criterios de selección de los usuarios [20].

En el Ejemplo 3.1.1 se puede observar una situación clásica en la literatura [6]. En este, se puede ver la aplicación de la consulta en un contexto real.

Ejemplo 3.1.1. Se desea buscar un hotel que se encuentre lo más cerca de la playa y que ofrezca las habitaciones más baratas. En la Figura 3.1 se presentan estos datos reflejados a través de un plano cartesiano, en donde en el eje X muestra la distancia de cada hotel de la playa y en el eje Y el precio de la habitación.

Como se puede apreciar en la Figura 3.1, los puntos marcados en negrita son el resultado de la consulta Skyline, los cuales reflejan aquellos hoteles más convenientes en cuanto a su relación distancia/precio.

El conjunto resultante de la consulta Skyline se le conoce como el conjunto *pareto-óptimo*, comúnmente utilizado en Economía e Ingeniería [38]. También es equivalente a resolver el problema del vector máximo en geometría computacional [23, 4].

La definición formal de Skyline es la siguiente [38, 4]:

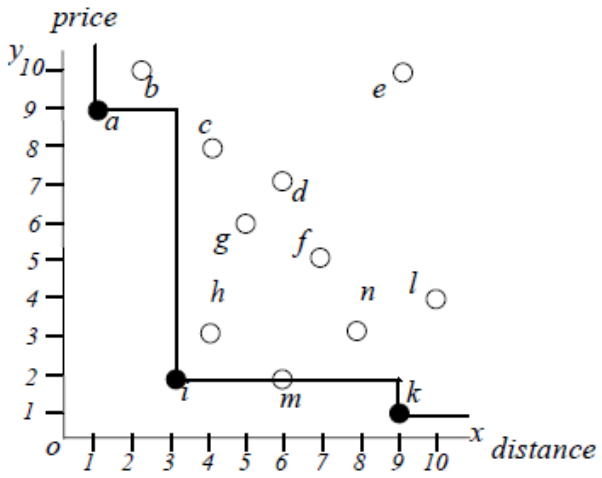


Figura 3.1: Ejemplo de resultado de consulta Skyline [37].

Definición 3.1. *Sea D un conjunto de puntos d-dimensionales, el resultado de la consulta Skyline S se define como:*

$$S = \{p \in D : \nexists q \in D : q \succ p\}$$

La relación de dominancia o de pareto dominancia $q \succ p$ se define de la siguiente manera [38]:

Definición 3.2. *Dado dos puntos d-dimensionales q y p, se dice que q domina a p, sí q es mejor o igual a p en todas las d dimensiones y q es estrictamente mejor que p en al menos una de las d dimensiones.*

Sin pérdida de generalidad y asumiendo que en cada atributo los valores más bajos son mejores, se puede definir de forma matemática la relación de dominancia de la siguiente manera [4]:

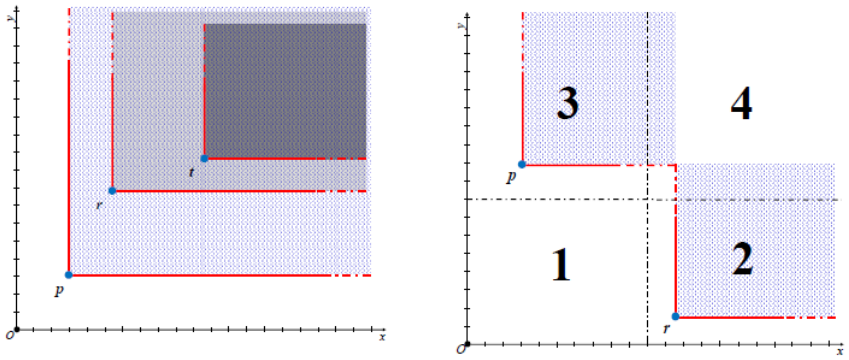
Definición 3.3. *Sean q y p dos puntos d-dimensionales, en donde $q = (q_1, ..., q_d)$ y $p = (p_1, ..., p_d)$, en donde q_i y p_i representan el valor del atributo de q y p respectivamente en la dimensión i, la relación de dominancia $q \succ p$ se define:*

$$q \succ p \Leftrightarrow \left(\bigwedge_{i=1}^{d} q_i \leq p_i\right) \wedge \left(\bigvee_{i=1}^{d} q_i < p_i\right)$$

La relación de dominacia tiene la propiedad de transitividad, la cual se puede observar en la Figura 3.2(a). Su definición es la siguiente [23]:

Propiedad 3.1.1. *Dado los puntos $p, r, t \in D$, si $p \succ r \wedge r \succ t \Rightarrow p \succ t$*

La transitividad se puede utilizar para eliminar de forma más rápida un solo punto o un grupo de puntos que están dominados por un punto p y que a su vez está dominado por un nuevo punto r [23].



(a) Transitividad (b) Incomparabilidad

Figura 3.2: Propiedades de la consulta Skyline [23].

Una segunda propiedad interesante de la relación de dominancia es la de la incomparabilidad, la cual se puede observar en la Figura 3.2(b), y se define como [23]:

Propiedad 3.1.2. *Dado dos puntos $p, r \in D$, si p no domina a r, (denotado como $p \prec\succ r$), y r no domina a p (denotado como $r \prec\succ p$), entonces p y r son incomparables (denotado $p \sim r$).*

Esta propiedad ayuda a determinar si uno o más puntos pueden ser Skyline, puesto que dos puntos Skyline son incomparables entre sí.

3.2. Algoritmos que responden la consulta Skyline.

El concepto de consultas Skyline se remonta a la década de 1960, en donde el conjunto Skyline era conocido como *el conjunto de Pareto*, *puntos admisibles* o *vectores máximos* [49]. Esta clase de problemas fue estudiado extensamente en la comunidad matemática durante aquel tiempo [23].

En los años ochenta y noventa, se propuso una variedad de algoritmos diferentes para situaciones específicas y que funcionan considerando que los datos se encuentran en memoria principal [23]. Estos algoritmos se vuelven ineficientes al aumentar el tamaño del conjunto de datos de entrada, por lo tanto no pueden funcionar en memoria secundaria [49].

En [18] se demostró que estos algoritmos basados en el enfoque de divide para vencer tiene un mal desempeño con respecto a la cantidad de dimensiones que tiene el conjunto de entrada debido a que no tienen en consideración las limitaciones de memoria [23]. Este tipo de enfoques sufre el problema de la *maldición de la dimensionalidad* [23].

El problema del vector máximo ha sido redescubierto posteriormente en el contexto de las bases de datos en [6]. Borzsony *et al.* propuso una extensión en SQL para el procesamiento de la consulta y los primeros algoritmos que resuelven este problema en las bases de datos.

A partir de entonces, han surgido una serie de algoritmos, los cuales se han clasificado de tres formas diferentes. La primera clasificación se refiere a la indexación del conjunto de entrada en alguna estructura de datos [4]. La segunda clasificación dice relación sobre la forma en que

resuelve el problema: si es basado en ordenamiento o particionamiento [28]. La tercera clasificación presenta cuatro grupos: basados en ordenamiento, en jerarquía, en divide y vencerás y en cubo [10].

La primera clasificación presenta las siguientes categorías [4]:

- **Algoritmos secuenciales o genéricos:** Estos algoritmos no requieren ningún paso de preproceso y, por lo tanto, se pueden aplicar incluso cuando la entrada es el resultado de alguna otra operación previa. El problema con los algoritmos secuenciales es que tienen que revisar todos los datos de entrada para calcular la consulta, lo cual es costoso en cuanto a recursos.
- **Algoritmos basados en índices u otra estructura:** Son más rápidos y menos costosos que las estrategias secuenciales sin índice, ya que evitan el acceso a todo el conjunto de datos. La desventaja está en que su aplicación está limitada por el requisito de indexación o del uso de una estructura de datos en particular [52].

La segunda clasificación presenta las siguientes categorías [28]:

- **Algoritmos basados en ordenamiento:** Los algoritmos basados en ordenamiento tienen como objetivo ordenar previamente el conjunto de entrada para eliminar los puntos que no forman parte del Skyline desde el principio. Como desventaja, estos algoritmos se centran únicamente en la optimización de la dominancia de los datos sin considerar la propiedad de incomparabilidad. En la medida en que disminuye la cantidad de puntos dominados entre si, aumenta la cantidad de puntos incomparables entre sí [28].
- **Algoritmos basados en particionamiento:** Estos algoritmos apuntan a agrupar los datos en subregiones que comparten puntos en común para llevar a cabo pruebas de dominancia al interior de la región. Si bien los algoritmos basados en particiones existentes exploran el potencial de optimización para la incomparabilidad, esto puede no alcanzar su máximo potencial si se realiza una mala partición.

La tercera clasificación presenta las siguientes categorías [10]:

- **Algoritmos basados en ordenamiento:** Los puntos se ordenan de acuerdo a una función de forma creciente o decreciente. Esto es una ventaja porque tras un cierto umbral, los puntos no pueden formar parte del Skyline, ahorrando así el uso de recurso computacional.
- **Algoritmos basados en jerarquía:** Utiliza estructuras como el R-tree o B-tree, permitiendo reducir de forma drástica el conjunto de candidatos Skyline.
- **Algoritmos basados en divide y vencerás:** La idea principal es dividir recursivamente un conjunto de datos dado en particiones hasta que pueda ser manejado en memoria y luego calcula el Skyline global mediante la fusión progresiva de los Skyline locales.
- **Algoritmos basados en cubo:** Se precalcula $2^d - 1$ consultas Skyline sobre un conjunto de d dimensiones para cada subespacio, ya que diferentes usuarios pueden tener diferentes preferencias de decisión óptima de múltiples criterios. Sin embargo, este precálculo del Skyline en cada subespacio puede incurrir en un costo prohibitivo.

A continuación se presentan los principales algoritmos existentes en la literatura enfocados en responder la consulta Skyline.

3.2.1. Block Nested Loop (BNL).

En [6] se presenta este algoritmo, el cual utiliza un búfer (o ventana) en memoria principal en donde almacena aquellos puntos que son candidatos a ser Skyline. El algoritmo basicamente revisa cada punto p almacenado en memoria secundaria, realiza un chequeo de dominancia, y de acuerdo al resultado de este chequeo, realiza los cambios correspondientes dentro de la ventana. Las reglas para dicho chequeo son las siguientes:

1. Si p está dominado por un punto en la ventana, significa que p no es un punto Skyline, por lo tanto, se descarta.

2. Si p domina uno o más puntos de la ventana, estos puntos se eliminan (ya que son dominados por p, por lo tanto, no pueden ser puntos Skyline), y p se inserta en la ventana.

3. Si p es incomparable con todos los puntos en la ventana (es decir, no domina ni está dominado), se inserta en la ventana, sí hay suficiente espacio en memoria o se escribe en un archivo temporal en el disco.

Una vez finalizada la revisión de todos los puntos del conjunto de entrada, se realiza una segunda revisión utilizando el archivo temporal como entrada (de haber uno). Lo anterior es debido a que no existe seguridad que los puntos del archivo temporal sean puntos Skyline, por lo tanto deben ser comparados con los existentes en la ventana.

En el mejor de los casos, este algoritmo posee una complejidad temporal de orden $O(n)$ (donde n es la cantidad de puntos del conjunto de entrada), y ocurre cuando todos los puntos necesarios caben dentro del búfer en memoria principal. Por otro lado, en el peor de los casos tiene una complejidad temporal de orden $O(n^2)$, en el caso en que posee un búfer en donde solo cabe un punto.

En [11] se presenta una mejora de este algoritmo, conocido como *Sort Filter Skyline* (SFS). En este algoritmo, el conjunto de entrada es ordenado utilizando una función monótona M. Este tipo de funciones garantiza que, dado dos puntos p y q, sí $M(p) \leq M(q)$, entonces q no dominará p [4]. Un par de ejemplos de funciones monótonas que pueden ser utilizadas son la norma L1 o la norma Euclideana [23].

El uso de este algoritmo disminuye la cantidad de comparaciones de dominancia a realizar, puesto que al preordenar el conjunto de entrada antes de revisarlos, si un punto p domina a un punto q, se obliga a revisar p antes que q [23]. La complejidad del algoritmo para el mejor de los casos es $O(dn + nlogn)$ y en el peor caso $O(dn^2)$, donde d es el número de dimensiones y n el tamaño del conjunto de datos. Este análisis involucra tanto la fase de ordenamiento como de revisión de los puntos [23].

En [3] y [4] se presenta *Sort and Limit Skyline algorithm* (SaLSa), que sigue el mismo principio de SFS: ordenar los puntos utilizando alguna función monótona y luego revisar cada uno de los puntos. Su principal diferencia es que posee una condición de detención que evita revisar

todo el conjunto de datos, asegurando que todos aquellos puntos que quedaron pendientes están dominados por al menos un punto del conjunto. Un factor clave que afecta el rendimiento de SaLSa es la elección de la condición de detención [4]. En [23] se presenta un ejemplo simple de la ejecución de este algoritmo.

3.2.2. Dividir para conquistar (D&C).

Algoritmo presentado inicialmente en [26, 39] que solo considera datos almacenados en memoria principal. Trabaja de la siguiente manera [6]:

1. Obtener la mediana del conjunto de entrada con respecto a una dimensión, y dividir la entrada de acuerdo a esa partición. Así se obtendrán los conjuntos P_1 y P_2.

2. Calcular el Skyline en P_1 y P_2, S_1 y S_2 respectivamente. Puede utilizarse otro algoritmo o volver a dividir el conjunto de acuerdo a otra dimensión. Se puede realizar dicha partición de forma recursiva hasta que queden muy pocos puntos dentro de la partición y sea trivial calcular el resultado de la consulta.

3. Fusionar el resultado Skyline de P_1 y P_2, comparando cada punto de S_1 con los puntos de S_2.

Una estrategia propuesta en [39] para realizar la fusión en el paso 3, es realizar una partición por cada dimensión, tal como aparece en la Figura 3.3(a):

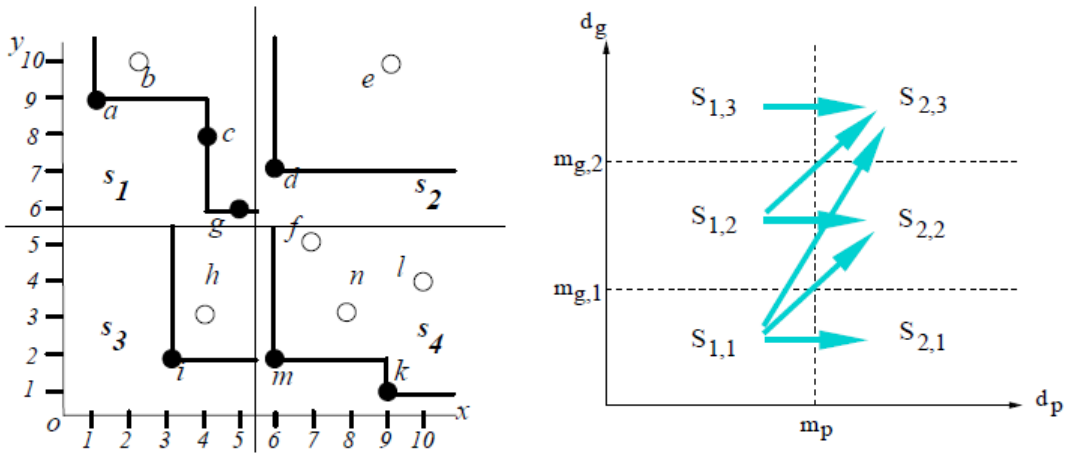
- Como en S_3 contiene puntos, el resultado final del Skyline contiene todos los puntos de S_3, mientras que el resultado encontrado en S_2 se descarta.

- Luego, se compara el resultado S_1 con S_3 para obtener aquellos puntos de S_1 que no sean dominados por S_3. De la misma forma se trabaja con S_4.

- En caso que S_3 esté vacío, los puntos de S_1 y S_4 forman completamente parte del resultado final del Skyline, y se deberá comparar S_2 con dicho resultado para descubrir si allí pueden existir más puntos existentes como resultado.

En la Figura 3.3(a) podemos ver que $S_1 = \{a, c, g\}$, $S_2 = \{d\}$, $S_3 = \{i\}$, $S_4 = \{k, m\}$, respectivamente. Para obtener el resultado final de la consulta S, se requiere eliminar los puntos que están dominados por algún punto en otras particiones. Por lo ya explicado, todos los puntos en el Skyline de S_3 deben aparecer en S, mientras que aquellos en S_2 se descartan inmediatamente porque están dominados por cualquier punto en S_3. Al comparar los puntos de S_1 sobre S_3, el punto a no está dominado por i por lo que está incluido en S. De manera similar, S_4 también se compara con puntos en S_3, lo que da como resultado la eliminación de m. Finalmente, el algoritmo termina con el conjunto de Skyline $S = \{a, i, k\}$.

En [6] se propone una partición de m-maneras, para así permitir que funcione con datos almacenados en memoria secundaria. En ella se realizan m particiones $P_1, ..., P_m$, de forma que P_i quepa en memoria principal y pueda ser procesado usando la división básica. Luego, en la etapa de fusión, es aplicada de modo que todas las particiones puedan fusionarse en la memoria



(a) División básica D&C. [37].

(b) División de 3-maneras D&C. [6].

Figura 3.3: Particionamiento para algoritmo D&C.

principal. En la Figura 3.3(b) se puede observar un ejemplo de una partición de 3-maneras, con las respectivas comparaciones entre las particiones que se deben realizar.

D&C es eficiente solo para pequeños conjuntos de datos. Para grandes conjuntos de datos, el proceso de partición requiere leer y escribir todo el conjunto de datos al menos una vez, lo que implica un costo significativo de lectura y escritura [53].

El algoritmo puede encontrar los puntos Skyline en $O(n)$ en el mejor de los casos, que es esencialmente el costo de fusión de las particiones divididas y en $O(n^2)$ en el peor de los casos, que es el costo de identificar los puntos del Skyline en cada subpartición y el coste de fusionar los resultados [23].

3.2.3. Bitmap.

En [47] se presenta *Bitmap*, un algoritmo que codifica el valor de cada atributo de cada punto en un bitmap para encontrar qué punto pertenece al Skyline. Sea $p = (p_1, p_2, ..., p_d)$ un punto del conjunto de datos, en donde d es la cantidad de dimensiones de p y p_i es el valor del atributo de p en la dimensión i. p es codificado en un bitmap de tamaño m de la siguiente manera:

- p_i es representado por k_i bits, en donde k_i representa la cantidad de valores diferentes que puede tener el conjunto de datos en la dimensión i, de tal forma que $m = \sum_{i=1}^{d} k_i$.
- Sea j_i la posición que ocupa p_i entre los k_i distintos valores ordenado de menor a mayor. Los $(k_i - j_i + 1)$ bits de más a la izquierda son 1, mientras que el resto son 0.

Sea BS_{ij} un bit-slice que corresponde a un bitmap que recoge todos los valores que poseen todos los puntos en el bit j de la dimensión i. A partir de estos bit-slice, se evalúa si un punto es Skyline de la siguiente manera:

ID	Coordenada	Representación en bitmap
a	(1,9)	(1111111**1**11, 1100000000)
b	(2,10)	(1111111**1**10, 1000000000)
c	(4,8)	(1111111**0**00, 1110000000)
d	(6,7)	(1111100**0**00, 1111000000)
e	(9,10)	(1100000**0**00, 1000000000)
f	(7,5)	(1111000**0**00, 1111100000)
g	(5,6)	(1111110**0**00, 1111000000)
h	(4,3)	(1111111**0**00, 1111111000)
i	(3,2)	(1111111**1**00, 1111111110)
k	(9,1)	(1100000**0**00, 1111111111)
l	(10,4)	(1000000**0**00, 1111110000)
m	(6,2)	(1111100**0**00, 1111111110)
n	(8,3)	(1110000**0**00, 1111110000)

Tabla 3.1: Ejemplo de codificación de los puntos en un Bitmap [37].

1. Sea $A = BS_{1j_1}\&...\&BS_{dj_d}$, en donde & corresponde a la operación *and* bit a bit, y j_i corresponde a la posición dentro del bitmap que ocupa el valor del atributo p_i.

2. Sea $B = BS_{1j_1-1}|...|BS_{dj_d-1}$, en donde a diferencia del cálculo de A, en B se utiliza el operador *or* y utilizan los bit-slice en una posición menor (un bit a la derecha) a los utilizados en el cálculo de A.

3. Sea $C = A\&B$. Sí C solo tiene ceros, significa que el punto a evaluar es Skyline.

En la Tabla 3.1 se encuentra un ejemplo de la codificación de un conjunto de datos. En este caso, tenemos dos dimensiones ($d = 2$). En cada dimensión, existen 10 posibles valores distintos ($k_1 = k_2 = 10$), por lo tanto se requieren 20 bits para representar cualquiera de los puntos del conjunto de datos ($m = 20$).

Para codificar un punto, por ejemplo $a = (1, 9)$, se sabe que a_1 es el primer menor valor posible en la dimensión 1 ($j_1 = 1$), mientras que a_2 es el noveno menor valor posible en la dimensión 2 ($j_2 = 9$). Para codificar a_1, se tiene que $(k_1 - j_1 + 1) = (10 - 1 + 1) = 10$, por lo tanto los 10 bits de más a la izquierda tienen el valor de 1, mientras que el resto es 0. En este caso los 10 bits que representan a este valor son 1. Para codificar a_2, se tiene que $(k_2 - j_2 + 1) = (10 - 9 + 1) = 2$, por lo tanto los dos primeros bits de más a la izquierda tienen el valor de 1, y el resto es 0.

Para averiguar si un punto es Skyline, por ejemplo el punto $i = (3, 2)$, se deberá calcular el bitmap A, B y C. Para eso, se sigue el siguiente procedimiento.

1. $A = B_{13}\&B_{22}$, en donde $B_{13} = 1100000010000$ y $B_{22} = 0000000011010$ (Ambos bit-slices están marcados en negrita en la Tabla 3.1). Esto da como resultado $A = 0000000010000$

2. $B = B_{12}|B_{21}$, en donde $B_{12} = 1100000000000$ y $B_{22} = 0000000001000$ (Estos bit-slices se encuentran a la derecha de aquellos bit-slices marcados en negrita). Esto da como resultado $B = 1100000001000$

3. $C = A\&B = 0000000000000$

4. Como C solo se compone de ceros, entonces el punto i pertenece al conjunto de Skyline. Sí hubiera aparecido al menos un 1, significa que el punto evaluado está dominado por algún punto, por lo tanto no forma parte del Skyline.

En [37] se realiza una simplificación de la operación, en donde solo consideran el cálculo de A, y si solo contiene un 1, de inmediato se considera como parte del Skyline. Este 1 resulta de la comparación del punto a evaluar consigo mismo.

La ventaja de este algoritmo radica en su velocidad al comprobar si un punto forma parte del Skyline, esto debido al uso de operaciones a nivel de bit [37]. Por otro lado, la principal desventaja de Bitmap radica tanto en el aumento de la cantidad de dimensiones a considerar como de los posibles valores distintos en cada dimensión. Ambos factores aumentan la cantidad de bits necesarios para representar cada punto y el cálculo pierde eficiencia [15].

3.2.4. Index.

En [47] se presenta *Index*, un algoritmo que utiliza un B-tree por cada dimensión d del conjunto de datos para responder la consulta Skyline. Funciona de la siguiente manera:

1. Los puntos del conjunto de datos se agrupan en d listas. Un punto $p = (p_1, ..., p_d)$ pertenece a la lista i sí y solo sí su atributo p_i es el menor comparado con el resto de los atributos de p. A este valor se le conoce como el $minC$ de p.

2. Cada lista se ordena de menor a mayor con respecto con el valor de su $minC$. Los puntos con el mismo $minC$ comparten la misma ubicación (o batch). Estas listas son indexadas en un B-tree.

3. El algoritmo comienza a recorrer las d listas, siguiendo siempre un orden estricto con respecto a su $minC$. En la medida en que va revisando los puntos, compara la dominancia con respecto a aquellos que han sido encontrados, y en caso de no ser dominados, se agrega al Skyline.

4. El algoritmo se detiene cuando todos los atributos del último punto encontrado son menores al siguiente $minC$ a revisar de su respectiva lista.

En la Tabla 3.2 se puede ver un ejemplo que refleja la formación de las listas y ubicación de cada punto dentro de su lista y su batch, acompañado de su $minC$. El algoritmo en este caso procede de la siguiente manera:

1. Se posiciona al principio de cada lista y escoge la lista cuyo valor de $minC$ sea menor. Como en este caso ambas listas tienen el mismo valor, se puede comenzar por cualquiera.

2. Se considera el primer batch de la Lista 1. Se revisa el punto a. Por ser el primer punto, se agrega al conjunto de solución, y se avanza al siguiente batch.

Lista 1	MinC	Lista 2	MinC
a(1,9)	1	k(9,1)	1
b(2,10)	2	i(3,2), m(6,2)	2
c(4,8)	4	h(4,3), n(8,3)	3
g(5,6)	5	l(10,4)	4
d(6,7)	6	f(7,5)	5
e(9,10)	9		

Tabla 3.2: Ejemplo de indexación de cada lista para ser procesada con Index [37].

3. Como en la Lista 1 el siguiente valor de $minC$ es 2, se continúa con la Lista 2. Tras encontrar el punto k y comprobar que no es dominado por los puntos ya encontrados, se agrega al conjunto solución y se avanza en dicha lista.

4. Ahora ambas listas se encuentran con el mismo valor de $minC$. Al continuar con la Lista 1, se encuentra el punto b, el cual al ser dominado por a, se descarta.

5. Al revisar el segundo batch de la Lista 2, se encuentran dos puntos: i y m. Al descubrir que m es dominado por i e i no es dominado por ninguno de los puntos ya encontrados, se agrega a i al conjunto de solución y se avanza al siguiente batch.

6. Como los siguientes valores de $minC$ son 4 y 3, las coordenadas de i son 3 y 2, $3 \leq 4$ y $2 \leq 3$, entonces el algoritmo se detiene, respondiendo con $\{a, k, i\}$ como resultado.

La principal ventaja de este algoritmo, es que permite responder la consulta evitando acceder a una cantidad importante de puntos, debido a que es capaz de percatarse cuando ya ha encontrado todos los puntos del Skyline [23]. Por otro lado, la principal desventaja de Index, es que pierde eficiencia al aumentar el número de dimensiones a considerar, debido a que con cada dimensión se debe manejar un B-tree en el proceso [15].

3.2.5. Nearest neighbor (NN).

En [25] se presenta un algoritmo basado en la búsqueda del vecino más cercano (*Nearest neighbor*). Trabaja sobre datos indexados en un R-tree y su objetivo es evitar consultas de dominancia innecesarias [23].

Básicamente, utiliza algún algoritmo de búsqueda del vecino más cercano con respecto al origen de la coordenada de puntos, utilizando alguna función monótona de distancia. El punto encontrado, además de pertenecer al Skyline, divide el espacio en 2^d subespacios. En la Figura 3.4 se puede observar un ejemplo para un espacio de 2 dimensiones. Aquí, 2 de los 4 subespacios son descartados: uno porque se encuentra vacío (región 1), y otro porque es la zona de dominancia del punto encontrado (región 4). Las regiones 3 y 2 son agregadas a una lista llamada *to-do list*. Posteriormente se continúa trabajando con la siguiente región de la lista, realizando el particionamiento de forma similar. En caso que una región se encuentre vacía, esta se descarta. El algoritmo finaliza cuando la lista se encuentre vacía.

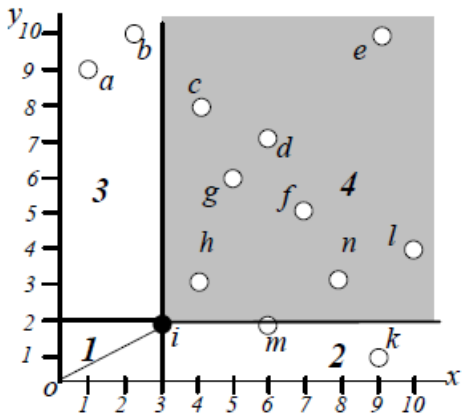


Figura 3.4: Partición del espacio tras encontrar al vecino más cercano. [37].

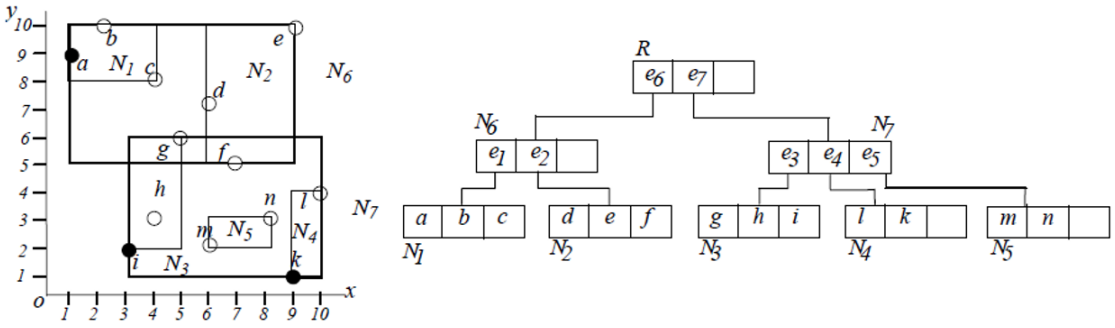
Uno de los problemas con este algoritmo, es que debido a la superposición de los espacios de búsqueda cuando la cantidad de dimensiones es mayor a 2, genera posibles puntos duplicados como resultado. Además, las regiones vacías pueden mantenerse innecesariamente dentro de *to-do list* durante mucho tiempo [37].

3.2.6. Branch and Bound Skyline (BBS).

En [36, 37] se presenta *Branch and Bound Skyline Algorithm* (BBS), un algoritmo basado en *Nearest neighbor* que recorre los nodos del R-tree. El algoritmo evita expandir nodos del R-tree que no contienen puntos de la solución. Se usa un heap H para el procesamiento, que mantiene las entradas de nodo o las entradas de datos ordanadas según su distancia mínima o *mindist* correspondiente desde el punto de origen, y un conjunto S para los objetos del Skyline.

El algoritmo trabaja de la siguiente manera:

1. Se expande la raíz del R-tree y se insertan sus hijos en H.
2. Se extrae de H la entrada e que se encuentre en el tope.
3. Si e es dominado por algún elemento de S, se descarta.
4. Si e no es descartado y se trata de un nodo interno, este se expande, en todos sus hijos e_i.
5. Se realiza la comprobación de dominancia de cada nodo e_i con respecto a S. Si es dominado, e_i se descarta. Si no es dominado, e_i ingresa a H.
6. Si e es un nodo de datos y no es dominado por ningún punto de S, este es ingresado a S.
7. El algoritmo termina cuando H está vacío.



Action	heap contents	S
access root	<e_7,4><e_6,6>	∅
expand e_7	<e_3,5><e_6,6><e_5,8><e_4,10>	∅
expand e_3	**<*i*,5>**<e_6,6><*h*,7><e_5,8> <e_4,10><*g*,11>	{*i*}
expand e_6	~~<*h*,7><e_5,8>~~<e_1,9><e_4,10><*g*,11>	{*i*}
expand e_1	**<*a*,10>** <e_4,10><*g*,11><*b*,12><*c*,12>	{*i*,*a*}
expand e_4	**<*k*,10>**~~<*g*,11><*b*,12><*c*,12><*l*,14>~~	{*i*,*a*,*k*}

Figura 3.5: Ejemplo de funcionamiento del algoritmo BBS [37].

En la Figura 3.5 se puede observar un ejemplo sobre la ejecución del algoritmo. Inicialmente entra la raíz a H. Al expandir la raíz, se insertan los nodos e_6 y e_7 con su respectivos valores *mindist*. En este caso para el cálculo del *mindist* se utiliza la distancia L1, es decir, la suma de las coordenadas. Como e_7 tiene un valor menor, es el siguiente en expandirse, obteniendo a e_3, e_4 y e_5. El siguiente nodo en ser expandido es e_3, obteniendo los nodos de datos g, h e i. El siguiente nodo a extraer es i, y como es un nodo de datos, se convierte en el primer punto Skyline encontrado.

El siguiente nodo a expandir es e_6, obteniendo los nodos e_1 y e_2. e_2 se descarta por estar dominado por un punto de S, al igual que h y e_5, que ya se encontraban dentro de H. El siguiente nodo en expandir es e_1, el cual contiene a, b y c. c no ingresa al heap por estar dominado por un punto de S. El siguiente nodo en el heap contiene el punto a, que termina formando parte del Skyline.

Tras expandir e_4 se encuentran los puntos k y l, de los cuales solo k ingresa a H, e inmediatamente es extraído del heap. Tras comprobarse que no es dominado por S, se convierte en el siguiente punto Skyline. Finalmente se revisan todos los demás nodos que quedan en H para comprobar finalmente que todos son dominados por algún punto de S. Finalmente H queda vacío y se termina el algoritmo.

3.3. Evaluación de los algoritmos.

En [25] se presenta una serie de criterios para evaluar el comportamiento y aplicación de un algoritmo que trabaja en tiempo real, es decir, en un entorno interactivo, como aplicaciones móviles [23].

Algoritmo / Propiedad	PRO	AFN	AFP	JUS	PRE	UNI
BNL	No	Si	No	Si	No	Si
SFS	Si	Si	Si	Si	No	Si
SaLSa	Si	Si	Si	Si	No	Si
D&C	No	Si	Si	Si	No	Si
Bitmap	Si	Si	Si	Si	No	Si
Index	Si	Si	Si	No	No	No
NN	Si	Si	Si	Si	Si	Si
BBS	Si	Si	Si	Si	Si	Si

Tabla 3.3: Evaluación de los algoritmos Skyline presentes en la literatura [23].

Los criterios son los siguientes:

- **Progresividad (PRO):** Una parte del conjunto perteneciente al Skyline debe ser entregado instantáneamente, y el resto debe aparecer de forma gradual.
- **Ausencia de falsos negativos (AFN):** El algoritmo debe entregar tarde o temprano la totalidad del conjunto Skyline.
- **Ausencia de falsos positivos (AFP):** El algoritmo debe asegurar que al responder con un punto, este forma parte del Skyline de manera definitiva, sin ser descartado posteriormente.
- **Justicia (JUS):** El algoritmo no debe favorecer aquellos puntos que sean particularmente buenos en una dimensión.
- **Incorporación de preferencia (PRE):** El usuario debe poder ingresar sus preferencias con respecto al orden en que quiere recibir los resultados de la consulta mientras el algoritmo está corriendo.
- **Universalidad (UNI):** El algoritmo debe poder aplicarse a cualquier distribución de datos y a cualquier dimensionalidad usando una estructura de datos y tecnología estándar.

En [23] se presenta una evaluación de los algoritmos aquí descritos, con los criterios presentados. Esto se ve reflejado en la Tabla 3.3. De todos los algoritmos, NN y BBS son los únicos que cumplen con todos los criterios, lo que significa que son los más aptos para ser utilizados en aplicaciones móviles.

En la Tabla 3.4 se presenta un resumen con la clasificación de los algoritmos con respecto a los tres criterios presentados en la Sección 3.1 además de su complejidad algorítmica [23]. Se puede observar que la mayoría de los algoritmos genéricos también son clasificados como algoritmos basados en ordenamiento, mientras que la mayoría de los algoritmos idexados también son clasificados como algoritmos basados en jerarquía.

Algoritmo	Clasificación según [4]	Clasificación según [28]. Basado en	Clasificación según [10]. Basado en	Complejidad
BNL	Genérico	Ordenamiento	Ordenamiento	$O(n^2)$
SFS	Genérico	Ordenamiento	Ordenamiento	$O(dn^2)$
SaLSa	Genérico	Ordenamiento	Ordenamiento	
D&C	Genérico	Particionamiento	Divide y conquista	$O(n^2)$
Bitmap	Indexados			
Index	Indexados		Jerarquía	
NN	Indexados	Particionamiento	Jerarquía	
BBS	Indexados	Particionamiento	Jerarquía	

Tabla 3.4: Tabla-resumen sobre clasificación de los algoritmos.

Capítulo 4

Variantes del operador Skyline.

En este capítulo se explicarán las principales variantes de las consultas Skyline. La idea principal y la noción de consulta de Skyline se mantienen. Cada variación delineada es aplicable y puede resolver diferentes aspectos de un problema.

4.1. Constrained Skyline Query.

Esta variante entrega los puntos del Skyline que se encuentren dentro de un espacio definido por una serie de restricciones, normalmente expresadas como un hiper-rectángulo d-dimensional [37]. En la Figura 4.1(a) se puede ver un ejemplo al respecto, en un conjunto de datos bidimensional.

Otro tipo de consultas con un nombre similar que podría confundirse con esta consulta son las consultas de Skyline con restricciones o *Skyline with Constrains* [51]. A diferencia de la consulta de restricción de Skyline, primero se calcula el operador Skyline, para luego aplicar la restricción [23]. En la Figura 4.1(b) se puede observar esta segunda variante.

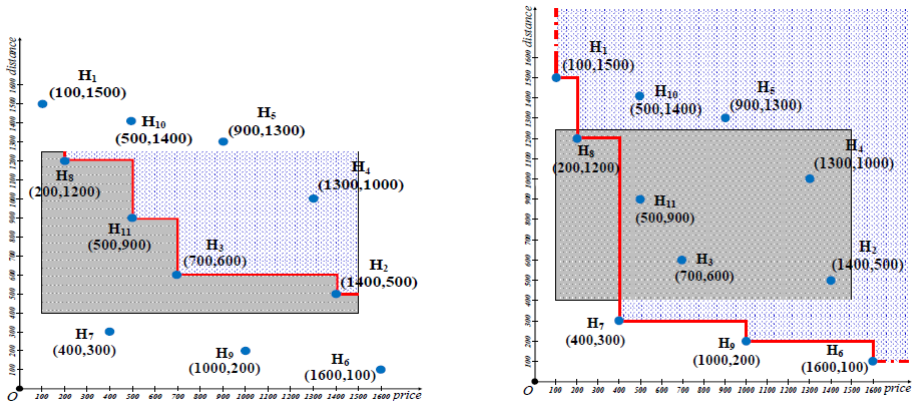
Esta consulta puede ser útil en el contexto del Ejemplo 3.1.1 en donde un usuario puede tener un presupuesto máximo para gastar en un hotel y/o permite una cierta distancia máxima de la playa. Es así como se puede obtener un nuevo Skyline considerando las nuevas restricciones y preferencias que tiene el usuario.

En [37] se propone utilizar BBS para responder esta variante, pero incluyendo las restricciones de la consulta a la hora de conservar un nodo.

4.2. Dynamic Skyline Query (DSQ).

Esta variación fue presentada en [36, 37]. Esta consulta responde el Skyline en un nuevo espacio m dimensional redefinido mediante una función $f : \mathbb{R}^d \to \mathbb{R}^m$. En el Ejemplo 4.2.1 se puede observar una aplicación de esta variante:

Ejemplo 4.2.1. Una base de datos almacena para cada hotel sus coordenadas X e Y y su precio, es decir, la base de datos contiene tres dimensiones. Suponga además que un usuario especifica



(a) Constrained Skyline Query.

(b) Skyline with Constrains.

Figura 4.1: Constrained Skyline Query v/s. Skyline with Constrains [23].

su ubicación actual (l_x, l_y) y desea conocer los hoteles más cercano con respecto a la distancia euclidiana y más barato. En este caso, cada punto P con coordenadas (p_x, p_y, p_z) en el espacio 3D original se transforma en un punto p' en el espacio 2D con coordenadas $(f_1(p_x, p_y), f_2(p_z))$ donde la dimensión las funciones f_1, f_2 se definen como:

$$f_1(p_x, p_y) = \sqrt{(p_x - l_x)^2 + (p_y - l_y)^2}$$

y

$$f_2(p_z) = p_z$$

Es así como en vez de obtener la consulta Skyline considerando la distancia del usuario y el precio, consulta más útil, en vez de considerar la distancia X e Y por separado.

4.2.1. Spatial Skyline Query (SSQ).

La consulta de Skyline espacial (SSQ) [43, 44] se puede considerar como un caso especial más restringido de la consulta de Skyline dinámico [37].

Un ejemplo en donde es aplicado este tipo de consultas es el presentado en el Ejemplo 4.2.2 [43, 44]:

Ejemplo 4.2.2. Suponga que un equipo de trabajo multidisciplinario se encuentra trabajando en diferentes oficinas ubicadas en diferentes lugares de una ciudad. Ellos deben decidir en qué restaurante realizarán sus almuerzos semanales. Para ello, al equipo de trabajo le interesan aquellos restaurantes que sea más cercano a todas las oficinas en cuanto a distancia.

El problema del Ejemplo 4.2.2 se vuelve más desafiante cuando los miembros del equipo se movilizan de oficina constantemente y cambian de lugar con el tiempo.

Dado un conjunto de datos D y un conjunto de puntos de consulta Q, la consulta SSQ recupera los puntos de D que no están dominados espacialmente por cualquier otro punto en D con respecto a Q. Un punto $p \in D$ domina espacialmente a un punto $r \in D$ con respecto a Q sí y solo sí p está más cerca de al menos un punto de $q \in Q$ en comparación de r, y tiene en el mejor caso es la misma distancia que r al resto de los puntos de consulta, es decir, ningún otro objeto está más cerca de todos los puntos de consulta dados simultáneamente [23].

En [44] se proponen dos algoritmos eficientes, denominados B^2S^2 y VS^2, para SSQ con objetos de consulta estáticos y un algoritmo, denominado VCS^2, para SSQ con objetos de consulta en movimiento. En [45] se propone una mejora de VS^2, argumentando que dicho algoritmo no responde con todos los puntos del Skyline, obteniendo así un resultado incorrecto.

4.3. Reverse Skyline Query (RSQ).

Reverse Skyline Query [14] recupera los puntos cuyos Skyline dinámicos contienen un punto de consulta determinado [23]. Considerando el Ejemplo 4.2.1, considere uno de los hoteles como q, la consulta sería descubrir si q se encuentra como respuesta a la consulta Skyline del usuario ubicado en (l_x, l_y). Esto con el propósito de decidir si vale la pena por ejemplo, enviarle publicidad respecto del hotel. Esta consulta puede ayudar en aplicaciones de investigación de mercado para descubrir si un producto específico atrae a los consumidores o para identificar la mejor ubicación para una nueva sucursal [14].

En [14] se propone *Branch and Bound Reverse Skyline* (BBRS) para el cálculo Reverse Skyline Query, que es una adaptación mejorada del algoritmo BBS original. También en [14] se presenta RSSA, el cual para reducir el costo computacional, utiliza un cálculo aproximado del Skyline dinámico para definir si un punto pertenece al Skyline Reverso.

4.4. Ranked Skyline Query.

Dado un conjunto de puntos en el espacio d-dimensional, *Ranked Skyline Query* o *top* k toma un parámetro k y una función monótona de preferencia f, y devuelve los K puntos del Skyline que tienen un puntaje mínimo según la función de entrada [37]. Considerando el Ejemplo 3.1.1, si se define $K = 2$ y $f = x + 3y^3$, la salida es $\langle k, 12\rangle, \langle i, 15\rangle$ en ese orden.

En [37] se argumenta que BBS puede manejar fácilmente este tipo de consultas modificando la definición de la distancia utilizada para ordenar los elementos del heap, de tal forma que pueda reflejar la función de preferencia.

4.5. Enumerating query y K-Dominating query.

Fueron propuestos en [36, 37]. Estos tipos de consultas no producen un conjunto Skyline, pero pueden funcionar como una medida de *bondad* en varios casos [23].

4.5.1. Enumerating Query.

Esta consulta devuelve, por cada punto del Skyline p, la cantidad de puntos dominados por p (denominado $num(p)$) [37]. Esta información puede ser relevante en aplicaciones donde la *bondad* de un punto del Skyline está determinada por el número de puntos que domina [37]. En el Ejemplo 3.1.1, el hotel i podría ser más interesante puesto que domina 9 puntos, frente a los 2 que dominan a y k.

En [36, 37] se propone calcular el Enumerating Query en los siguientes pasos:

1. Calcular el Skyline normal utilizando algún algoritmo tradicional (como por ejemplo BBS).
2. Por cada punto p perteneciente al Skyline, contabilizar la cantidad de puntos contenidos al interior de la región de dominancia de p.

Una forma de evitar visitar los varias veces los nodos del R-tree que forman parte de más de una región de dominancia de varios puntos del Skyline, en [37] se propone el proceso inverso: por cada punto q que no pertenece al Skyline, se consulta sí es dominado por cada punto p perteneciente al Skyline.

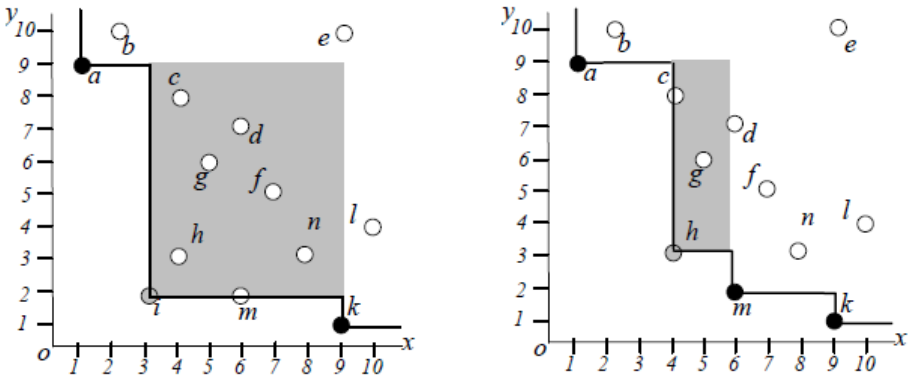
4.5.2. K-Dominating query.

Esta consulta responde con los K puntos que dominan la mayor cantidad de otros puntos, es decir, que posean los valores más altos de $num(p)$ [37]. En el Ejemplo 3.1.1, para $K = 3$, la consulta responde con los puntos i, h y m, porque $num(i) = 9$, $num(h) = 7$ y $num(m) = 5$.

En [36, 37] se propone un algoritmo llamado *k-dominating_ BBS*. A grandes rasgos, trabaja de la siguiente forma:

1. Calcula la consulta Enumerating Query, y se ordena el resultado de acuerdo a su $num(p)$.
2. Se extrae el punto cuyo valor $num(p)$ sea el mayor. Este punto forma parte K-Dominating Query.
3. Se calcula la consulta Enumerating Query con restricción, dentro de la región de dominancia del punto extraído del paso anterior. Los puntos resultantes se agregan al resultado obtenido anteriormente.
4. Se vuelve a repetir el paso 2 seguido del 3 hasta obtener los K puntos.

En la Figura 4.2(a), se puede observar el resultado del Skyline del Ejemplo 3.1.1. Como se puede observar, el punto i es el punto con un mayor valor de $num(p)$. Posteriormente en 4.2(b) se puede observar que se eliminó el punto i, y se calculó el Skyline dentro de su región de dominancia. A esta altura h se convierte en el punto con mayor cantidad de dominados con un total de 8 puntos.



(a) Skyline tras encontrar a i. (b) Skyline tras eliminar a i y encontrar a h.

Figura 4.2: Procesamiento de *k-dominating_ BBS* [37].

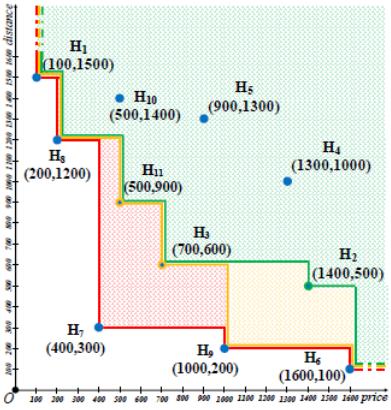


Figura 4.3: Ejemplo de 0-Skyband, 1-Skyband y 2-Skyband. [23].

4.6. K-Skyband Query.

Esta consulta responde con todos aquellos puntos que son dominados por a lo más K puntos [37]. Cuando $K = 0$, el resultado es el Skyline convencional. En la Figura 4.3 se puede observar el resultado de 0-Skyband (Marcado en rojo), 1-Skyband (Marcado en amarillo) y 2-Skyband (Marcado en verde). Cabe señalar que el resultado de 1-Skyband incluye el resultado de 0-Skyband, y el de 2-Skyband incluye los dos anteriores.

En [37] se propone una modificación de BBS para permitir responder esta consulta.

4.7. Group-by Skyline.

Esta consulta corresponde a una combinación de la consulta Skyline con Group by [37]. Suponga que a los datos del Ejemplo 3.1.1 se agrega un nuevo atributo referente a la clasificación

	D&C	BNL	SFS	Bitmap	Index	NN	BBS
Constrained	Si	Si	Si	Si	Si	Si	Si
DSQ	Si	Si	Si	No	No	Si	Si
Ranked	No	No	No	No	No	No	Si
K-Dominating	Si	Si	Si	Si	Si	Si	Si
K-Skyband	No	Si	Si	No	No	No	Si
Group-by	No	Si	Si	No	No	No	Si

Tabla 4.1: Comparación de aplicabilidad [37].

del hotel en estrellas. El resultado de la consulta será el cálculo de los puntos Skyline por el valor de cada estrella. Se obtendrá entonces el Skyline de todos los hoteles con una estrella, con dos estrellas, y así hasta cubrir todos los posibles valores que puede tomar la cantidad de estrellas.

4.8. Resumen de las variantes.

En [37] se presenta una comparación entre los algoritmos presentados en el el Capítulo 3 y las variantes presentadas en este capítulo, con el propósito de revisar si es posible de adaptar estos algoritmos para conseguir responder algunas de las variantes. El resultado de este análisis se puede observar en la Tabla 4.1.

Parte III

Algoritmos propuestos.

Capítulo 5

Consulta Skyline en un k^2-tree.

5.1. Introducción.

En este capítulo se presentan las propuestas de esta tesis para responder la consulta Skyline sobre datos almacenados en un k^2-tree. Estos algoritmos son una adaptación de *Branch and Bound Skyline Algorithm* (BBS) presentado en [36, 37]. Uno de ellos, *BBS k^2-tree* (BBSk), que de forma muy parecida a BBS, trabaja con los nodos del k^2-tree almacenándolos en un heap que los ordena de acuerdo a su distancia con respecto al origen. La otra propuesta, *Skyline X*, que ordena dichas submatrices de acuerdo a su valor en la coordenada X. En ambos casos, y utilizando las propiedades presentes en un k^2-tree, se utilizaron condiciones de poda que permiten reducir la cantidad de nodos a revisar, siendo un aporte significativo en la eficiencia de esta consulta.

En la Sección 5.2 se presentan algunos antecedentes presentes en la literatura y que serán de utilidad para la propuesta de esta tesis. En la Sección 5.3 se presenta BBSk, su funcionamiento y un ejemplo. En la Sección 5.4 se presenta un análisis teórico de BBSk, enfocado en estimar el costo temporal y espacial del algoritmo. En la Sección 5.5 se presenta *Skyline X*, una segunda alternativa para encontrar el Skyline sobre datos indexados en un k^2-tree. Finalmente en la Sección 5.6 se presentan las conclusiones de todo lo expuesto en este capítulo.

5.2. Antecedentes.

Branch and Bound Skyline Algorithm funciona sobre datos que se encuentran indexados en un R-tree. Un R-tree es una estructura de datos jerárquica que contiene tanto nodos internos definidos mediante un *minimum bounding rectangle* (MBR), como nodos hojas que representa los puntos almacenados.

Este algoritmo comienza expandiendo la raíz del R-tree, y en la medida en que va iterando, va expandiendo aquellos nodos que se encuentran más cercanos al origen de coordenadas. Para definir la distancia de cada nodo, se utiliza la métrica *mindist* de cada MBR, que corresponde a la distancia entre la esquina más cercana al origen con dicho origen de coordenadas. El algoritmo verifica por cada nodo para comprobar si contiene algún punto que forma parte de la solución. En caso de comprobar que no es así, procede a eliminarlo. La forma de comprobar sí un nodo

contiene puntos como parte de la solución es realizando la comparación de dominancia entre el nodo con respecto a los puntos ya encontrados del Skyline hasta ese instante.

Para realizar todo el proceso, el algoritmo utiliza un heap, dentro del cual almacena las entradas de aquellos nodos del R-tree que pudiesen contener puntos Skyline. En este heap, los nodos se ordenan de acuerdo a su *mindist*, de tal forma que siempre que se requiera procesar un nodo, el heap entregue aquel nodo que se encuentre más cerca del origen.

Algoritmo 5.2.1 BBS(R-tree R)

```
1: S = ∅
2: Insert all entries of the root R in the heap
3: while heap not empty do
4:     Remove top entry e
5:     if e is dominated by some point in S then
6:         Discard e
7:     else
8:         if e is an intermediate entry then
9:             for each child e_i of e do
10:                if e_i is not dominated by some point in S then
11:                    Insert e_i into heap
12:                else
13:                    Insert e_i into S
14:                end if
15:            end for
16:        end if
17:    end if
18: end while
19: return S
```

En el Algoritmo 5.2.1 se puede observar el pseudocódigo original de BBS. Cabe destacar que en la comparación de dominancia entre algún nodo de R-tree y los puntos de S se realiza dos veces: antes de ser insertado en el heap (línea 10) y antes de expandirlo (línea 5). Esta segunda comparación es necesaria porque algún nodo podría quedar dominado por un punto de S descubierto después de su inserción. Un ejemplo de esta situación se puede ver en la Figura 3.5, cuando el nodo e_5 ingresa al heap, pero al ser extraído se descubre que es dominado por el punto i que no había sido descubierto cuando el nodo se insertó.

5.3. Algoritmo propuesto: *BBS k^2-tree* (BBSk).

A continuación se presenta *BBS k^2-tree* (BBSk). Al igual que BBS, BBSk comienza con la raíz del k^2-tree, que corresponde a la matriz completa. Esta es subdividida en sus k^2 submatrices, de las cuales solo aquellas que superen las condiciones de descarte ingresan al heap. Este heap, de la misma forma que ocurre en BBS, ordena las submatrices de acuerdo a su distancia al origen de coordenadas.

De forma iterativa, se procede a evaluar el tope del heap de acuerdo a la dominancia de los puntos Skyline encontrados hasta el momento. Si no es dominado por dichos puntos, se subdivide en sus k^2 hijos, se aplican las condiciones de descarte por cada uno de ellos, y quienes las superen ingresan al heap o al conjunto solución S, dependiendo si son nodos internos u hojas del k^2-tree respectivamente.

Las condiciones de descarte de cada submatriz son claves para mejorar la eficiencia del algoritmo. Mientras más pronto sea descartado un nodo, menos regiones que no contienen puntos Skyline son revisadas. BBSk puede implementar más condiciones de descarte que BBS debido a las propiedades del k^2-tree. Según BBSk, una submatriz puede ser descartada si:

1. **Se encuentra dominada por algún punto Skyline encontrado hasta el momento**: Esta condición es idéntica a la presentada en BBS. Sí la submatriz es dominada por algún punto Skyline encontrado hasta el momento, significa que no contiene puntos que pertenezcan al Skyline, por lo tanto, es descartada.

2. **Se encuentra vacía**: Esta condición es posible gracias a las propiedades del k^2-tree, que permite identificar si una submatriz está vacía de forma rápida, accediendo a la ubicación que la representa dentro de los bitmaps T y L. Si una submatriz está vacía, no contiene puntos del Skyline, por lo tanto es descartada.

3. **Es dominada por alguno de sus hermanos no vacíos**: Esta condición hace referencia a una comparación entre las submatrices de un mismo padre, para mantener solo aquellas que no estén dominadas por sus hermanos no vacíos. Si una submatriz es dominada por uno de sus hermanos, quiere decir que no contiene puntos Skyline, por lo tanto es descartada. Esto reduce significativamente la cantidad de submatrices que ingresan al heap, mejorando la eficiencia de BBSk.

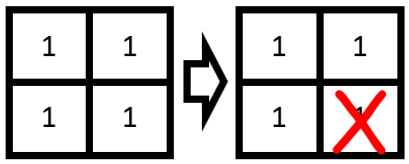
   En general, recordando que el origen de coordenadas en un k^2-tree se encuentra en la esquina superior izquierda, si uno de sus hermanos no vacíos se ecuentra sobre y a la izquierda de una submatriz, esta es descartada. Esta comparación puede realizarse con facilidad, accediendo a los bitmaps T y L.

La Figura 5.1(a) representa el caso más simple, con un $k = 2$. Se puede observar que la submatriz de la esquina superior izquierda domina a aquella ubicada en la esquina inferior derecha, por lo tanto de los cuatro hijos, tres ingresan al heap. Si la submatriz dominante fuera cero, la poda no podría realizarse, pero debido a la condición 2, siguen siendo tres los hijos que entran al heap.

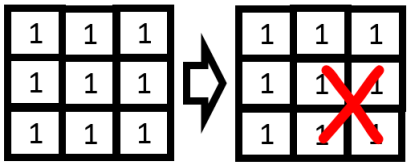
En la Figura 5.1(b) se puede observar una submatriz completa con $k = 3$. Se puede ver que la cantidad de nodos podados es 4 y de nodos conservados es 5.

En las Figuras 5.1(c) y 5.1(d) la submatriz no se encuentra completa, sino que tiene ceros justo en las submatrices que dominaban y por ende podaban las 4 submatrices de la Figura 5.1(b). A pesar de ello, la cantidad de submatrices conservadas se mantienen y las demás son podadas por la condición 2 ó 3. En general, la cantidad máxima de submatrices conservadas tras la aplicación de las condiciones de descarte está en función de k.

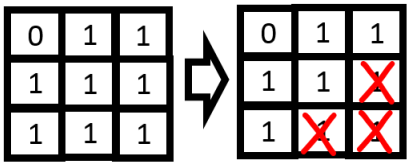
En el Algoritmo 5.3.1 se presenta el pseudocódigo de BBSk. Al igual que en BBS, se realizan dos chequeos de dominancia entre cada submatriz y S: en la línea 5 y 7. En la línea 7 se puede observar la aplicación de las tres condiciones de descarte. Las condiciones de descarte son aplicadas desde la que tiene un menor costo de aplicación hasta la más costosa. Tras revisar las condiciones de descarte, en la línea 8 se consulta si la submatriz en cuestión se trata de una hoja. De ser una hoja, ingresa de inmediato a S; en caso contrario, ingresa al heap para ser revisada a posterioridad.



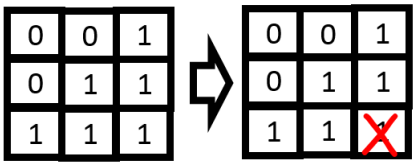
(a) Poda de una submatriz completa en una matriz con $k = 2$



(b) Poda de una submatriz completa en una matriz con $k = 3$



(c) Poda de una submatriz en una matriz con $k = 3$ y con un cero en el primer hijo



(d) Poda de una submatriz en una matriz con $k = 3$ y con tres ceros en la esquina superior izquierda

Figura 5.1: Poda de submatrices dominadas por sus hermanos.

Algoritmo 5.3.1 BBSk(ktree k)

```
1: Insert root in heap
2: S = ∅
3: while heap ≠ ∅ do
4:     n = heap top
5:     if n is not dominated by S then
6:         for each child n_i in n do
7:             if n_i ≠ ∅ and n_i is not dominated by its sibling not empty and n_i is not dominated by S then
8:                 if n_i is not a leaf then
9:                     heap = heap ∪ n_i
10:                else
11:                    S = S ∪ n_i
12:                end if
13:            end if
14:        end for
15:    end if
16: end while
17: return S
```

El heap almacena las submatrices del k^2-tree que contienen puntos Skyline. Por cada submatriz se almacena su índice que lo representa en el bitmap $T : L$, las coordenadas (x, y) de su esquina superior izquierda (y que es la que se encuentra más cercana al origen), y el tamaño de la submatriz.

5.3.1. Ejemplo.

En la Figura 5.2 se puede observar un ejemplo de un k^2-tree. Aparece la submatriz de 8×8, su representación en el k^2-tree, y su implementación en los bitmaps T y L. En este ejemplo, $k = 2$,

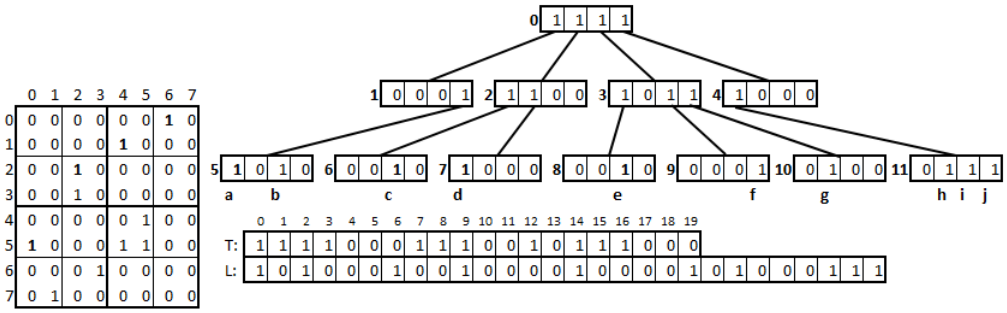


Figura 5.2: Ejemplo para la ejecución de BBSk.

Acción	**Contenido del heap**	**Contenido de S**
Inicio	$\langle 0,(0,0),0\rangle$	$\emptyset$
Expandir 0	$\langle 1,(0,0),0\rangle, \langle 2,(4,0),4\rangle, \langle 3,(0,4),4\rangle$	$\emptyset$
Expandir 1	$\langle 5,(2,2),4\rangle, \langle 2,(4,0),4\rangle, \langle 3,(0,4),4\rangle$	$\emptyset$
Expandir 5	$\langle 2,(4,0),4\rangle, \langle 3,(0,4),4\rangle$	$\langle a,(2,2)\rangle$
Expandir 2	$\langle 6,(4,0),4\rangle, \langle 3,(0,4),4\rangle, \langle 7,(6,0),6\rangle$	$\langle a,(2,2)\rangle$
Expandir 6	$\langle 3,(0,4),4\rangle, \langle 7,(6,0),6\rangle$	$\langle a,(2,2)\rangle, \langle c,(4,1)\rangle$
Expandir 3	$\langle 8,(0,4),4\rangle, \langle 9,(0,6),6\rangle, \langle 7,(6,0),6\rangle$	$\langle a,(2,2)\rangle, \langle c,(4,1)\rangle$
Expandir 8	~~$\langle 9,(0,6),6\rangle$~~, $\langle 7,(6,0),6\rangle$	$\langle a,(2,2)\rangle, \langle c,(4,1)\rangle, \langle e,(0,5)\rangle$
Expandir 7	$\emptyset$	$\langle a,(2,2)\rangle, \langle c,(4,1)\rangle, \langle e,(0,5)\rangle, \langle d,(6,0)\rangle$

Tabla 5.1: Ejecución de BBSk sobre el k^2-tree de la Figura 5.2.

los nodos internos fueron numerados desde el 0 al 11, mientras que las hojas fueron nombradas de la a a la j.

En la Tabla 5.1 se puede apreciar el proceso que realiza el algoritmo para encontrar los puntos Skyline. En cada fila se puede observar el contenido del heap y de S tras acceder a una submatriz determinada y procesarla. Para efectos de este ejemplo, cada nodo que ingresa al heap aparece bajo el formato $\langle n,(x,y),distancia\rangle$. n representa el número del nodo de acuerdo a la Figura 5.2. Las coordenadas (x,y) representan el punto que se encuentra en la esquina más cercana al origen de la submatriz. La distancia utilizada en este ejemplo corresponde a la norma L1, es decir, la suma de las dos coordenadas. Cada punto de S está representado bajo el formato $\langle nombre,(x,y)\rangle$. El nombre representa la letra con que se identifica el punto en el k^2-tree, y las coordenadas (x,y) representan la ubicación del punto.

Inicialmente dentro de heap solo se encuentra la raíz (nodo 0), y S está vacío. Al expandir el nodo 0, aparecen los nodos 1, 2, 3 y 4. El nodo 4 no es agregado al heap, porque está dominado por el nodo 1. El nodo 1 tiene prioridad por sobre sus hermanos debido a que posee una distancia menor.

Al expandir el nodo 1, aparece el nodo 5, el cual es el cuarto hijo del nodo 1. Sus hermanos se encuentran vacíos por lo tanto no ingresan al heap. Tanto el nodo 5, 2 y 3 poseen la misma distancia, por lo tanto cualquiera de los tres puede ser expandido. Al expandir el nodo 5, encuen-

tran cuatro nodos hojas. Primero se revisa a, el cual ingresa a S. Al revisar b se comprueba que es dominado por a y se descarta.

Al expandir el nodo 2, aparecen dos nuevos nodos: 6 y 7. El nodo 7 queda al final del heap debido a que el valor de su distancia es 6, mientras que su hermano tiene una distancia 4. Al expandir el nodo 6, solo aparece el punto c, que al no ser dominado por a, también ingresa a S.

El siguiente nodo a expandir es el 3. Allí aparecen los nodos 8, 9 y 10, de los cuales solo 8 y 9 ingresan al heap. El nodo 10 es dominado por su hermano, el nodo 8, por lo tanto queda descartado. De los nodos que quedan en el interior, el nodo 8 es el único que posee la distancia menor, por lo tanto es el aparece en el tope del heap. Al expandirlo, aparece el punto e, el cual al no ser dominado por ningún punto de S, es agregado al mismo conjunto.

Al revisar el nodo 9, se comprueba de inmediato que es dominado por el recién descubierto punto e, por lo tanto es descartado apenas es extraído del tope. El nodo que queda es el 7, del cual aparece el punto d, que al no ser dominado por ningún punto de S, se convierte en el último punto Skyline en ser descubierto.

5.4. Análisis teórico.

5.4.1. Condiciones de descarte.

Inicialmente, se revisarán las propiedades que presentan las tres condiciones de descarte, para demostrar que encuentran todos los puntos Skyline y trabaja de forma eficiente.

Lema 5.1. *BBSk revisa todas aquellas submatrices que en su interior contiene un punto de Skyline.*

Demostración. Suponga que existe una submatriz n que contiene un punto Skyline, pero que no es revisado por BBSk. Para que no sea revisada, debe ser descartada por alguna condición. La primera condición implica que debe existir un punto en S que la domine. Como n contiene un punto de Skyline, esto no puede ocurrir. Tampoco puede ser descartada por la condición 2 debido a que no está vacía. Para ser descartada por la tercera condición, debe existir otra submatriz hermana que lo domine, pero como n contiene un punto Skyline, esto no debe ocurrir. Como no existe otra condición de descarte, n no puede ser descartada. ■

Lema 5.2. *BBSk solo revisa aquellas submatrices que no estén vacías.*

Demostración. Es evidente debido al punto 2 presentado en la Sección 5.3 y que se ve reflejado en la línea 7 del Algoritmo 5.3.1. El chequeo se realiza accediendo al bitmap $T : L$ en la posición que representa a la submatriz a consultar en el k^2-tree. Si el valor almacenado en dicha posición es un 0, la submatriz está vacía y es descartada. ■

Para analizar la tercera condición de descarte, se debe definir cuáles submatrices se deben revisar para definir si una submatriz es descartada. A nivel de implementación del k^2-tree, se debe identificar a qué posiciones dentro de $T : L$ se debe acceder para definir el cumplimiento de esta condición.

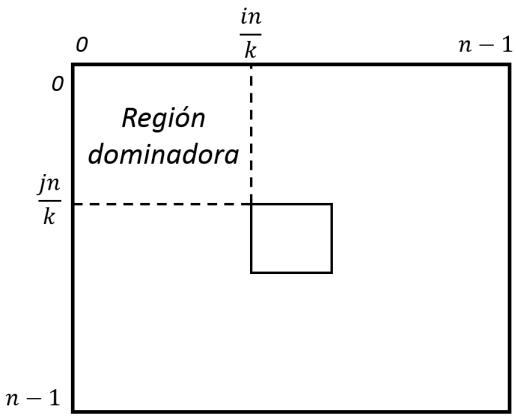


Figura 5.3: Regíon dominadora.

Sea una submatriz cuadrada $n_{i,j}$ de tamaño $\frac{n}{k}$, cuya esquina más cercana al origen se encuentra ubicada en las coordenadas $(i\frac{n}{k}, j\frac{n}{k})$, en donde i $(0 \leq i < k)$ y j $(0 \leq j < k)$ representan la columna y la fila en donde se ubica la submatriz con respecto a sus hermanos respectivamente. La región dominadora de $n_{i,j}$ se define como:

Definición 5.1. *La región dominadora de una submatriz es aquella cuyas coordenadas* (x, y) *son menores a la esquina más cercana al origen de dicha submatriz.*

Aplicado al caso de $n_{i,j}$, la región dominadora es representada por el rango $[0, i\frac{n}{k} - 1] \times [0, j\frac{n}{k} - 1]$. En la Figura 5.3 se puede observar de forma gráfica la región dominadora. De acuerdo a esto se puede asegurar lo siguiente:

Lema 5.3. *Una submatriz* $n_{i,j}$ *es podada si uno de sus hermanos no vacíos se encuentra dentro de la región dominadora de* $n_{i,j}$*.*

Demostración. Debido a la Definición 5.1, es evidente que cualquier punto perteneciente a dicha región domina a la esquina más cercana al origen de $n_{i,j}$, debido a que en ambas coordenadas sus valores son menores.

Por otra parte, para que la esquina más cercana al origen de $n_{i,j}$ cumpla con este requisito, sus valores (x, y) deben ser los menores dentro del rango que define a $n_{i,j}$. Por lo anterior, se puede concluir que dicha esquina domina a todos los puntos de $n_{i,j}$.

Por transitividad, si la región dominadora domina a la esquina más cercana al origen de $n_{i,j}$, y dicho punto domina a todos los puntos de $n_{i,j}$, significa la región dominadora domina a todos los puntos de $n_{i,j}$. ■

El costo de implementar la tercera condición de descarte es bajo. A lo más se deberán revisar por cada submatriz $n_{i,j}$ no vacía $(i-1)(j-1)$ submatrices. Esta revisión simplemente consta de acceder a $(i-1)(j-1)$ posiciones dentro del bitmap $T : L$. Si en su revisión se encuentra con al menos un 1, significa que existe una submatriz no vacía dentro de la región dominadora, $n_{i,j}$ es dominado por dicha submatriz, y se descarta.

Corollario 5.1. *En el peor de los casos, se deben revisar* $(k-1)^2$ *nodos para comprobar si una submatriz es podada por uno de sus hermanos.*

Esta situación ocurre cuando una submatriz se encuentra en la esquina más alejada en el origen, y es la única no vacía (submatriz $n_{k-1,k-1}$). Como la condición de detención para revisar la regíon dominadora es encontrar un 1 y al estar todas las submatrices vacías, se deben revisar las $(k-1)^2$ submatrices antes de determinar que $n_{k-1,k-1}$ no es dominada. Como esta consulta es considerada una operación a nivel de bits y k se considera una constante definida a la hora de construir el k^2-tree, se puede concluir que la aplicación de la tercera condición es constante.

Corollario 5.2. *Una submatriz puede podar hasta* $(k-1)^2$ *submatrices hermanas.*

En el lado opuesto, si la primera submatriz, ubicada en la esquina más cercana al origen (submatriz $n_{0,0}$), no está vacía. En este caso, al ser el primer nodo en ser revisado, puede descartar $(k-1)^2$ con tan solo un acceso al bitmap $T : L$ por cada nodo consultado.

Como se puede observar en la Figura 5.3, aquellas submatrices ubicadas en la misma fila y en la misma columna que $n_{i,j}$ no se encuentra en la región dominadora. Esto es porque ninguna de esas submatrices domina completamente a $n_{i,j}$. Lo anterior no significa que existan algunos puntos pertenecientes a alguna submatriz ubicada sobre o a la izquierda de $n_{i,j}$ que no pueda dominar a algún punto de $n_{i,j}$, pero para determinar esto se deben expandir las submatrices involucradas y realizar las comparaciones a niveles más profundos del k^2-tree.

Esto no funciona de la misma forma cuando las submatrices tienen tamaño 1, es decir, cuando se está trabajando a nivel de puntos. En este caso, los hermanos ubicados sobre o a la izquierda de la submatriz (o punto) si dominan a dicha submatriz. Un ejemplo de esto se puede observar en el nodo 5 de la Figura 5.2, en donde el punto a se encuentra ubicado inmediatamente sobre b. Un problema grave se puede formar si el algoritmo revisa primero b y luego a, debido a que erróneamente b podría entrar a S sin ser un punto Skyline.

Esto no es problema para BBSk, debido a que a la hora de revisar cada hijo de un nodo, los revisa de arriba hacia abajo, y de izquierda a derecha. Por lo anterior, ya sea que se esté trabajando con nodos internos o con nodos hojas, siempre a la hora de revisar un punto, ya se habrán revisado a sus hermanos que se encuentren por sobre o a la izquierda de ellos; y si alguno es Skyline, entrará a S antes de revisar a sus hermanos.

5.4.2. Tamaño del heap.

Para poder determinar una estimación teórica de los recursos que emplea BBSk, primero se debe establecer cuántos nodos pueden entrar al heap dada ciertas acciones. Esto por un lado permite determinar la cantidad máxima de nodos que BBSk deberá procesar, y por otro poder definir el tamaño máximo que puede alcanzar el heap durante el proceso.

Lema 5.4. *Por cada submatriz expandida, a lo más se insertarán* $2k-1$ *submatrices al heap.*

Demostración. Tras lo presentado en la Tabla 5.2 se puede apreciar cuál es la cantidad máxima de submatrices que ingresan al heap tras extraer una submatriz del tope, para distintos valores de

k	**Máxima cantidad de submatrices insertadas**
2	3
3	5
4	7
5	9
6	11
7	13
8	15
...	...
k	$2k-1$

Tabla 5.2: Cantidad máxima de submatrices insertadas al heap tras extraer un nodo en función de k.

Altura	**Tamaño máximo del heap**
0	1
1	$1-1+(2k-1)=2k-1$
2	$(2k-1)-1+(2k-1)=4k-3$
3	$(4k-3)-1+(2k-1)=6k-5$
4	$(6k-5)-1+(2k-1)=8k-7$
...	...
h	$2hk-(2h-1)=2(k-1)h+1$

Tabla 5.3: Cantidad máxima de nodos que ingresan al heap al descender desde una submatriz de altura h hasta las hojas.

k desde el 2 al 8. Estos valores se obtienen al asumir que la primera submatriz contiene puntos. En caso de estar vacía, ocurrirá algo similar a lo presentado en la Figura 5.1(c) y la Figura 5.1(d), es decir, que las submatrices vacías reemplazarán a las submatrices no podadas y la cantidad de nodos a ingresar al heap se mantendrá. Por inducción, se puede concluir que la cantidad máxima de submatrices a insertar en el heap es de $2k-1$. ■

De lo anterior también se puede deducir que cada vez que se extrae una submatriz del tope, el heap crece a lo más en $2k-2$ elementos, debido a que se extrae una e ingresan $2k-1$.

Sea una submatriz con un punto Skyline y de altura h $(0 < h \leq H)$, en donde H es la altura del k^2-tree. Al descender por esta submatriz y extraer el punto Skyline, se puede determinar cuál es la cantidad máxima de nodos que entrarán al heap.

Lema 5.5. *Al descender por una submatriz con un punto Skyline hasta llegar a dicho punto, el heap habrá crecido a lo más en $2(k-1)h+1$ nodos, en donde h es la altura de la submatriz.*

Demostración. Al descender el primer nivel, el nodo insertado se extrae, y en su lugar ingresan $2k-1$ hijos, que corresponde a la máxima cantidad de hijos que pueden ingresar de acuerdo al Lema 5.4. Posteriormente, al extraer un nuevo nodo, quedan $(2k-1)-1+(2k-1)=4k-3$.

En la Tabla 5.3 se puede observar los distintos casos para una submatriz de altura 1 a la 4. Por inducción se puede concluir que al bajar por una submatriz de altura h hasta encontrar el punto Skyline, el heap tendrá como máximo $2(k-1)h+1$ nodos nuevos. ■

Cuando BBSk comienza buscando el primer punto Skyline, la raíz es el único nodo que se encuentra en el heap, por lo tanto la altura que se debe recorrer es H, la altura del k^2-tree. Tras encontrar el primer punto, ya no existirá una matriz de altura H dentro del heap, sino que varios nodos con una altura menor. Dependiendo de la distancia de estos nodos al origen, se puede determinar cuál será el siguiente nodo a expandir, y dependiendo de su altura, se puede estimar la cantidad máxima de nuevos nodos que serán recibidos en el heap.

5.4.3. Análisis temporal.

Para poder determinar el costo temporal, es necesario determinar cuántos nodos son procesados por BBSk. Ya se ha determinado por los Lemas 5.4 y 5.5 que en la medida en que se van procesando los nodos, nuevos nodos van ingresando al heap.

Tras lo presentado en el Lema 5.5, la cantidad de nodos que ingresan al heap tras encontrar un punto Skyline, y que deberán ser revisado por BBSk, es $2(k-1)h+1$. Sea $|S|$ la cantidad de puntos Skyline existentes, la cantidad de nodos que deben ser procesados por BBSk es de:

$$\sum_{i=1}^{|S|}(2(k-1)h_i+1) = 2(k-1)\sum_{i=1}^{|S|}h_i + |S| \tag{5.1}$$

En donde h_i es la altura del nodo desde donde comienza la búsqueda del i-ésimo punto Skyline.

Debido a la primera condición de descarte, se deberá comprobar si la submatriz procesada no es dominada por algún punto Skyline ya encontrado. Para eso, se debe realizar una comprobación de dominancia entre la submatriz y cada punto Skyline encontrado hasta el momento. Sea G_j la cantidad de puntos Skyline encontrados a la hora de revisar el j-ésimo nodo. Desde el inicio de la revisión hasta el momento de encontrar el primer punto Skyline, $G_j = 1$. Aunque no se ha encontrado ningún punto Skyline, es errado afirmar que no existe costo a la hora de identificar el dicho punto. Por lo tanto, al encontrar el primer punto Skyline, se realizan $2(k-1)H+1$ operaciones.

En el proceso de encontrar el segundo punto Skyline, nuevamente $G_j = 1$, debido a que solo existe un punto Skyline con el cual se deben comparar todos los nodos, por lo tanto la cantidad de comparaciones de dominancia que deben realizarse para encontrar el segundo punto es $2(k-1)h_2+1$, en donde h_2 es la altura de la submatriz en donde comienza la búsqueda del segundo punto Skyline.

En general, por cada punto Skyline encontrado, G_j aumenta su valor en 1, pero siempre se cumplirá que $G_j \leq |S|$, debido a que no habrán más de $|S|$ puntos Skyline con los cuales aplicar

la primera condición de descarte. Considerando esto, el costo temporal está definido por:

$$T(n) \leq \left((2(k-1) \sum_{i=1}^{|S|} h_i + |S| \right) |S| \tag{5.2}$$

Recordando que $h_i \leq H$, se puede determinar que $\sum_{i=1}^{|S|} h_i \leq \sum_{i=1}^{|S|} H = |S|H$. Por lo tanto, aplicando esta nueva consideración, se tiene que:

$$T(n) \leq ((2(k-1)|S|H + |S|)\,|S| = 2(k-1)|S|^2 H + |S|^2 \tag{5.3}$$

Recordando que $H = \log_k n$, en donde n es el tamaño de la matriz, se puede determinar que el costo temporal, asociado a la cantidad de nodos que BBSk debe procesar está delimitado por el siguiente valor:

$$T(n) \leq 2(k-1)|S|^2 \log_k n + |S|^2 = O(k|S|^2 \log_k n) \tag{5.4}$$

De lo que se puede determinar en la ecuación 5.4, es que el costo temporal depende principalmente del tamaño de la matriz, el valor k de la estructura y de la cantidad de puntos Skyline.

Una cota inferior puede plantearse bajo el supuesto en que todos los puntos Skyline se encuentran circunscritos en una submatriz de tamaño c pequeño, ubicada en la esquina superior izquierda, es decir, que se encuentra apegada al origen del sistema de coordenadas. Inicialmente $h_1 = H$, en donde h_1 corresponde a la altura de la submatriz que debe ser revisada para encontrar el primer punto Skyline. Como dicha submatriz corresponde a la matriz completa, entonces la altura equivale a la altura del k^2-tree H. En esta etapa no se realizan comparaciones de dominancia con respecto a S, debido a que aún no se han encontrado puntos Skyline con los cuales comparar. De acuerdo al Lema 5.5, al heap ingresan $2(k-1)H + 1$ submatrices, y por lo tanto se realizan $2(k-1)H + 1$ operaciones.

Tras encontrar el primer punto, la siguiente submatriz en ser revisada presenta una altura $\log_k c = h$, asumiendo el supuesto inicial. Frente a esto, $h_2 = h$, y la cantidad de nuevas submatrices a ingresar es $2(k-1)h + 1$. Por cada una de las $2(k-1)h + 1$ submatrices se realiza una comparación de dominancia con el punto Skyline encontrado. Es por eso, que se puede concluir que para encontrar el segundo punto Skyline, se realizan $2(k-1)h + 1$ comparaciones de dominancia.

Asumiendo que el resto de las alturas de las submatrices en donde se ubican los puntos Skyline faltante es h, es decir, $h_i = h$, siempre ingresarán $2(k-1)h + 1$ submatrices nuevas al heap. Al buscar el tercer punto, se realizan dos comparaciones de dominancia por las $2(k-1)h+1$ submatrices ingresadas, dando un total de $2(2(k-1)h + 1)$ comparaciones de dominancia. De forma similar, al ubicar el cuarto punto Skyline, se realizan $3(2(k-1)h + 1)$ comparaciones de

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	1	0
2	0	0	0	0	0	1	0	0
3	0	0	0	0	1	0	0	0
4	0	0	0	1	0	0	0	0
5	0	0	1	0	0	0	0	0
6	0	1	0	0	0	0	0	0
7	1	0	0	0	0	0	0	0

Figura 5.4: Distribución de los puntos en el peor de los casos para BBSk, con $n = 8$

dominancia. En general, la cantidad de comparaciones de dominancia que se deben realizar es:

$$2(k-1)H + 1 + \sum_{i=1}^{|S|} i(2(k-1)h+1) = 2(k-1)H + 1 + (2(k-1)h+1)\left(\frac{|S|^2+|S|}{2}\right) \quad (5.5)$$

Finalmente, recordando que $H = \log_k n$ la cota inferior de la cota temporal es de:

$$T(n) \geq 2(k-1)H + 1 + (2(k-1)h+1)\left(\frac{|S|^2+|S|}{2}\right) = O(k\log_k n + k|S|^2 \log_k c) \quad (5.6)$$

En la medida que el valor $\log_k c$ sea pequeño, los puntos Skyline se encontrarán distribuídos en un espacio menor, por lo tanto tenderán a disminuir en cantidad.

Asumiendo que $c = k$, correspondiendo al menor tamaño de una submatriz sin ser un punto individual, finalmente se puede concluir que en general, el costo temporal de BBSk oscila entre $O(k\log_k n + k|S|^2)$ y $O(k|S|^2 \log_k n)$.

En el mejor de los casos, dentro de la matriz existe un único punto, el cual es Skyline. En este caso la cantidad de submatrices vacías permite una poda importante, por cada submatriz revisada solo ingresará una nueva submatriz, y solo se deberá revisar H nodos, y sobre estos nodos a lo más se realizará 1 comparación de dominancia con el el único punto encontrado. Esto implica que el costo de encontrar el Skyline es de orden $O(k\log_k n)$.

Por otro lado, en el peor de los casos, todos los puntos indexados en el k^2-tree se encuentran dentro del Skyline, ubicados de forma tal que todos los puntos sean incomparables entre si. El objetivo es determinar cuál es la mayor cantidad de puntos que pueden indexarse en el k^2-tree de tal forma que todos sean Skyline.

En la Figura 5.4 se encuentra un ejemplo de dicho caso. Los puntos están distribuídos de forma de maximizar la cantidad de puntos incomparables que pueden ser indexados en el k^2-

tree, generando una diagonal que va desde la esquina superior derecha hasta la esquina inferior izquierda. Cualquier punto ubicado sobre la diagonal dominará a los puntos de la diagonal, pero reducirá la cantidad de puntos pertenecientes al Skyline. Cualquier punto ubicado bajo la diagonal será dominado por al menos un punto de la diagonal, a menos que dichos puntos sean eliminados, pero esto también disminuye la cantidad de puntos Skyline.

Por lo anterior, se puede concluir que $|S| \leq n$. Utilizando esto en la ecuación 5.4, queda que la cantidad máxima de operaciones a realizar es de $2(k-1)n^2 \log_k n + n^2$. Finalmente el costo temporal en este caso es de orden $O(kn^2 \log_k n)$.

5.4.4. Análisis espacial.

De todas las estructuras adicionales utilizadas en BBSk, el heap es la más significativa en cuanto a la cantidad de recursos utilizados. De acuerdo al Lema 5.5, al encontrar un punto Skyline, a lo más ingresan $2(k-1)h+1$ nodos más. Tras encontrar el primer punto Skyline, el heap tendrá un tamaño máximo de $2(k-1)H+1$ nodos. Cuando comienza el proceso de ubicar el segundo punto Skyline, ingresarán $2(k-1)h_2+1$ nodos nuevos, pero a su vez, algunos nodos serán descartados producto del primer punto encontrado. Por lo tanto, la cantidad de nodos procesados por BBSk no es equivalente al tamaño máximo del heap, debido a que en la medida en que nuevos nodos van ingresando, otros se van retirando por la primera condición de descarte.

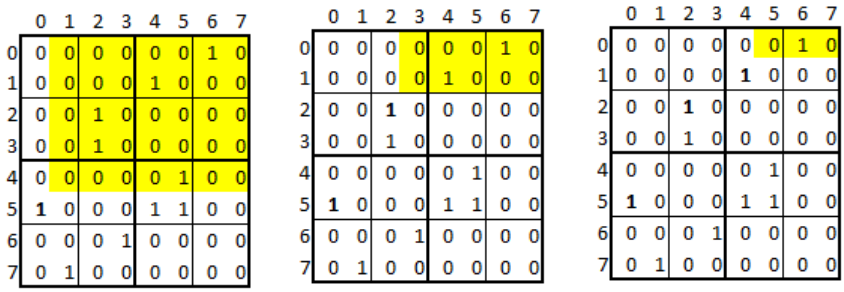
Determinar entonces el tamaño máximo del heap es difícil porque depende de la cantidad de nodos descartados mientras se busca un punto Skyline. Los nodos descartados dependen de la dominancia de los puntos en S, conjunto que crece de manera progresiva en la medida en que se van encontrando puntos Skyline. Mientras más cerca se encuentren los puntos Skyline al origen, mayor será su región de dominancia, descartarán más nodos, y en general el heap tendrá un tamaño pequeño.

Lo que si se determinar es una cota superior que delimita la cantidad de submatrices que habrá dentro del heap. Esta cota está definida por la cantidad de submatrices procesadas por BBSk. De acuerdo a lo presentado en la ecuación 5.1 y recordando que $\sum_{i=1}^{|S|} h_i \leq |S|H$ y que $H = \log_k n$, en donde n es el tamaño de la matriz, se puede determinar el costo espacial como:

$$T(n) \leq 2(k-1)\sum_{i=1}^{|S|} h_i + |S| = 2(k-1)|S|\log_k n + |S| = O(k|S|\log_k n) \tag{5.7}$$

En el mejor de los casos, cuando existe solo un punto que es Skyline, el heap solo conservará un nodo, nodo que se irá renovando en la medida en que el algoritmo descienda por el k^2-tree hasta llegar a las hojas. Por lo tanto en el mejor de los casos BBSk ocupa un espacio de orden constante.

Por otro lado, en el peor de los casos, los puntos en S no descartarán ningún nodo debido a que todos los puntos son Skyline, y el resto de los nodos serán podados ya sea por la segunda o la tercera condición de descarte. En este caso el espacio utilizado es de orden $O(kn\log_k n)$, en el caso en que se presente una situación desfavorable como la expuesta en la Figura 5.4.



(a) R tras encontrar el primer punto Skyline.

(b) R tras encontrar el segundo punto Skyline.

(c) R tras encontrar el tercer punto Skyline.

Figura 5.5: Ajuste de R en la medida en que se van encontrando puntos Skyline.

5.5. Una nueva alternativa: *Skyline X*.

A continuación se presenta otro algoritmo que también responde la consulta Skyline sobre datos indexados en un k^2-tree. Se trata de *Skyline X*, un algoritmo basado en BBS y en una propuesta de [34], en donde se busca los puntos dominados sobre un Wavelet Tree.

De forma similar a BBSk, utiliza un heap en el cual se almacenan las submatrices que pudieran contener puntos Skyline para revisarlas posteriormente. El orden que posee el heap y por ende el tipo de punto encuentra como primer punto Skyline varía con respecto a BBSk:

- *Skyline X* ordena los puntos priorizando aquellos con un menor valor en su coordenada X.
- En caso que dos puntos tengan el mismo valor de X, se ordenaran de acuerdo al menor valor en su coordenada Y.

Inicialmente se define un rectángulo R que cubre toda la matriz. Cada vez que encuentra un nuevo punto Skyline, R es reajustado cubriendo la región delimitada en su esquina inferior izquierda por el último punto encontrado. El espacio al interior de R es el único que puede contener nuevos puntos Skyline. El lado izquierdo como máximo contendrá otros puntos Skyline ya identificados. La esquina inferior derecha corresponde a la zona dominada por el último punto encontrado. La primera condición de descarte de BBSk es reemplazada por la intersección de la submatriz revisada con R. Esto es una mejora importante debido a que evita realizar las comparaciones de dominancia entre cada submatriz y los puntos Skyline encontrados durante la ejecucuón, reduciendo el tiempo de revisión de cada submatriz a un orden constante.

En el Algoritmo 5.5.1 se puede apreciar el pseudocódigo de *Skyline X*. Se puede observar que es muy similar a BBSk, salvo que la comprobación de dominancia sobre S es reemplazada por la intersección con R como se puede ver en las líneas 6 y 12. En las líneas 3 y 9 se puede observar la definición y redefinición de R respectivamente, realizada cada vez que un nuevo punto Skyline es encontrado.

Otra diferencia con BBSk, es que los nodos hojas si ingresan al heap. Si no ingresaran al heap, existe el peligro en donde un nodo padre de hijos hojas que se encuentra al tope del heap,

Algoritmo 5.5.1 SkylineX(ktree k)

```
1: Insert root in heap
2: S = ∅
3: R = [0, n]x[0, n]
4: while heap ≠ ∅ do
5:     n = heap top
6:     if n ∩ R ≠ ∅ then
7:         if n is a leaf then
8:             S = S ∪ n
9:             Readjust R
10:        else
11:            for each child n_i in n do
12:                if n_i ≠ ∅ and n_i ∩ R ≠ ∅ and n_i is not dominated by its sibling not empty then
13:                    heap = heap ∪ n_i
14:                end if
15:            end for
16:        end if
17:    end if
18: end while
19: return S
```

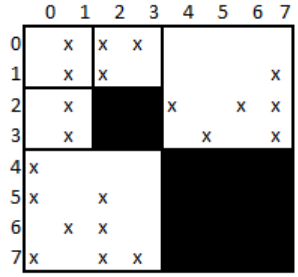


Figura 5.6: Ejemplo en donde los nodos hojas deben ingresar al heap en *Skyline X*.

reduzca el espacio de R y pode injustamente otros nodos con el mismo valor de X pero un valor mayor en Y y que poseen un punto Skyline. Un ejemplo de ello se encuentra en la Figura 5.6. Sea $n_1 = [0,1] \times [0,1]$ y $n_2 = [0,3] \times [4,7]$. De acuerdo al orden del heap, n_1 es revisado antes que n_2. n_1 posee un punto Skyline que es el $(1,0)$, esto reduce R hasta desaparecer, y el punto Skyline $(0,4) \in n_2$ es podado.

5.5.1. Ejemplo.

En la Tabla 5.4 se puede apreciar un ejemplo de la ejecución de *Skyline X* utilizando el k^2-tree presentado en la Figura 5.2. Como inicialmente R cubre toda la matríz, todas aquellas submatrices que no están vacías ni fueran dominadas por sus hermanos ingresan al heap. una vez encontrado el primer punto (en este caso e), R es reajustado como se ve en la Figura 5.5(a). Los puntos Skyline se entregan en orden de su ubicación desde la izquierda a la derecha.

Acción	**Contenido del heap**	**Contenido de S**	R
Inicio	$\langle 0,(0,0)\rangle$	$\emptyset$	$[0,7]\times[0,7]$
Expandir 0	$\langle 1,(0,0)\rangle, \langle 3,(0,4)\rangle, \langle 2,(4,0)\rangle$	$\emptyset$	$[0,7]\times[0,7]$
Expandir 1	$\langle 3,(0,4)\rangle, \langle 5,(2,2)\rangle, \langle 2,(4,0)\rangle$	$\emptyset$	$[0,7]\times[0,7]$
Expandir 3	$\langle 8,(0,4)\rangle, \langle 9,(0,6)\rangle, \langle 5,(2,2)\rangle, \langle 2,(4,0)\rangle$	$\emptyset$	$[0,7]\times[0,7]$
Expandir 8	$\langle e,(0,5)\rangle, \langle 9,(0,6)\rangle, \langle 5,(2,2)\rangle, \langle 2,(4,0)\rangle$	$\emptyset$	$[0,7]\times[0,7]$
Extraer e	~~$\langle 9,(0,6)\rangle$~~, $\langle 5,(2,2)\rangle, \langle 2,(4,0)\rangle$	$\{e\}$	$[1,7]\times[0,4]$
Expandir 5	$\langle a,(2,2)\rangle, \langle b,(2,3)\rangle, \langle 2,(4,0)\rangle$	$\{e\}$	$[1,7]\times[0,4]$
Extraer a	~~$\langle b,(2,3)\rangle$~~, $\langle 2,(4,0)\rangle$	$\{e,a\}$	$[3,7]\times[0,1]$
Expandir 2	$\langle 6,(4,0)\rangle, \langle 7,(6,0)\rangle$	$\{e,a\}$	$[3,7]\times[0,1]$
Expandir 6	$\langle c,(4,1)\rangle, \langle 7,(6,0)\rangle$	$\{e,a\}$	$[3,7]\times[0,1]$
Extraer c	$\langle 7,(6,0)\rangle$	$\{e,a,c\}$	$[5,7]\times[0,0]$
Expandir 7	$\langle d,(6,0)\rangle$	$\{e,a,c\}$	$[5,7]\times[0,0]$
Extraer d	$\emptyset$	$\{e,a,c,d\}$	$\emptyset$

Tabla 5.4: Ejecución de *Skyline X* sobre el k^2-tree de la Figura 5.2.

5.5.2. Análisis teórico.

Debido al orden en que los puntos Skyline son encontrados tanto por BBSk como por *Skyline X*, se puede determinar que *Skyline X* es menos eficiente que BBSk.

Suponga dos puntos Skyline p y q. p posee un valor muy bajo en la coordenada X, pero muy alto en la coordenada Y; mientras q posee un valor muy bajo en ambas coordenadas, de tal forma que $x_p < x_q$ y $y_p > y_q$. De acuerdo al funcionamiento de ambos algoritmos, *Skyline X* encuentra como primer punto a p, mientras que BBSk encuentra como primer punto a q. Debido a su alto valor en Y, la región que domina p es más pequeña que la región que domina q, por lo tanto la reducción del espacio de búsqueda de *Skyline X* es menor que el de BBSk.

En la medida en que ambos algoritmos se vayan ejecutando, *Skyline X* entregará los puntos que se encuentren más a la izquierda, sin importar el valor de Y. Por lo tanto, si el valor de Y tiende a ser muy alto, la capacidad de poda disminuye, y la región de búsqueda sigue siendo amplia.En cambio BBSk entrega los puntos más cercanos al origen, asegurando que ambas coordenadas tengan el valor más bajo posible. Esto asegura maximizar la región de dominancia, mejorando la capacidad de poda y reduciendo la región de búsqueda.

El costo temporal de *Skyline X* presenta una cota superior de $O(k|S|\log_k n)$. La primera condición de descarte no está presente en el algoritmo, por consecuente no existe una comparación con los puntos Skyline encontrados durante la ejecución. Por lo tanto, el proceso de cada submatriz n es $O(1)$, y el costo temporal está asociado a la cantidad de submatrices procesadas. Dicha cantidad de submatrices es equivalente a las submatrices procesadas por BBSk, puesto que la segunda y la tercera condición de descarte si son aplicadas en *Skyline X*, y es en particular la tercera condición la que delimita las submatrices procesadas. Finalmente, el costo temporal está definido por la ecuación 5.7.

A la hora de definir una cota inferior al costo temporal, se supondrá la misma situación que

la presentada en la Sección 5.4.3. Suponga que todos los puntos Skyline se encuentran ubicados en una submatriz de tamaño c ubicada en la esquina más cercana al origen de coordenadas. Como se explicó en la Sección 5.4.3, la altura a recorrer para encontrar el primer punto Skyline es igual a la altura del árbol H, mientras que para encontrar los otros $|S| - 1$ puntos Skyline, se deben procesar submatrices de tamaño $\log_k c = h$. Como la primera condición de descarte no se encuentra presente en *Skyline X*, la cota inferior será equivalente a la cantidad de submatrices a procesar en este caso. Dicha cantidad es:

$$2(k-1)H + 1 + (2(k-1)h + 1)(|S| - 1) = 2(k-1)(H + (|S| - 1)h) + |S| \tag{5.8}$$

Considerando lo planteado en la ecuación 5.8, se puede concluir que la cota temporal inferior es:

$$T(n) \geq O(k \log_k n + k|S| \log_k c) \tag{5.9}$$

Asumiendo $c = k$, se puede concluir que el costo temporal de *Skyline X* oscila entre $O(k \log_k n + k|S|)$ y $O(k|S| \log_k n)$.

De forma similar al costo temporal, el costo espacial también es $O(k|S| \log_k n)$, debido a que el funcionamiento del heap presenta similares características que el heap de BBSk.

5.6. Conclusión.

En este capítulo se presentaron dos algoritmos que responden la consulta Skyline sobre datos indexados en un k^2-tree. Estos algoritmos son *BBSk^2-tree* o BBSk, y *Skyline X*. Ambos se basan en la estrategia de *Branch and Bound Skyline Algorithm* o BBS, que recorre los nodos de un R-tree de una manera eficiente, encontrando los puntos Skyline de forma progresiva.

De forma similar a BBS, BBSk y *Skyline X* recorren los nodos del k^2-tree con la ayuda de un heap que almacena aquellos nodos que pueden contener puntos Skyline, y descarta aquellos que de antemano se descubre que no contienen puntos solución. La diferencia entre BBSk y *Skyline X* es la forma de ordenar los nodos dentro del heap. Mientras que BBSk prioriza las submatrices más cercanas al origen, *Skyline X* prioriza aquellas submatrices con menor valor de X.

En estos algoritmos se presentan tres condiciones de descarte que permite evitar revisar submatrices que no contienen puntos Skyline. De las tres condiciones, la tercera, que dice relación con la dominancia entre las submatrices hermanas, permite mejorar la eficiencia de BBSk y *Skyline X*. Esta condición es inherente al k^2-tree, ya que es gracias a su estructura, se puede descartar con facilidad grandes regiones del espacio que no contienen puntos Skyline, reduciendo así el espacio de búsqueda.

El costo temporal de BBSk está es de orden $O(k|S|^2 \log_k n)$, en donde n es el tamaño de la matriz y $|S|$ es la cantidad de puntos Skyline. Mientras, el costo espacial de BBSk y el costo tanto temporal como espacial de *Skyline X* es de orden $O(k|S|^2 \log_k n)$, que corresponde a la máxima cantidad de submatrices a procesar. *Skyline X*, a diferencia de BBSk, no implementa la primera condición de descarte, por lo tanto el costo de revisar cada submatriz es constante, mientras que para BBSk, el costo de revisar cada submatriz está definido por la cantidad de puntos dentro de

S hasta el momento, G_j.

A pesar de lo anterior, también se debe tener en consideración que debido al funcionamiento de ambos algoritmos, BBSk tiende a reducir rápidamente la región de búsqueda a diferencia de *Skyline X*, reduciendo así la cantidad de submatrices en el heap. Esta es una ventaja que permite que BBSk sea un algoritmo competitivo frente a *Skyline X* a pesar del costo temporal que presentan, especialmente en circunstancia en donde el valor de $|S|$ es reducido, ya que BBSk puede superar el costo cuadrático de su cota superior y resultar más eficiente.

La distribución clusterizada es ideal para ser indexadas en un k^2-tree puesto que mejoran su propiedad de compactación. Este tipo de distribución entrega una baja cantidad de puntos Skyline, y como este es un factor clave para la eficiencia tanto de BBSk y *Skyline X*, se puede concluir que ambos algoritmos también se ven beneficiados para este tipo de distribución.

En el peor de los casos, el costo temporal de BBSk es $O(kn^2 \log_k n)$, debido a que n es la mayor cantidad de puntos Skyline al interior de un k^2-tree. Por otro lado, el costo espacial de BBSk y el costo tanto temporal como espacial de *Skyline X* es de orden $O(kn \log_k n)$. Este es un resultado prometedor puesto que el peor de los casos es poco probable que se produzca, por lo que el Skyline suele tener una cantidad de puntos inferior a n. También se debe tener en consideración que al evitar la primera condición de descarte por parte de *Skyline X*, es posible mejorar la eficiencia de la consulta en comparación con la propuesta de BBSk.

Finalmente, debido a lo expuesto a lo largo de este capítulo, se puede concluir que tanto BBSk como *Skyline X* son algoritmos idóneos para responder el Skyline sobre datos indexados en un k^2-tree de manera eficiente, eficiencia que mejora en la medida en que el conjunto Skyline es reducido. Aunque, *Skyline X* pretende ser el más prometedor, BBSk presenta una mejor capacidad de poda, reduciendo el espacio de búsqueda de puntos Skyline, y mejorando su eficiencia. Los datos clusterizados, que son idóneos para ser compactados con esta estructura, también permiten aplicar ambos algoritmos de manera eficiente. En general, si el conjunto Skyline es reducido, se puede encontrar utilizando una menor cantidad de recursos.

En el Capítulo 9 se realizará el análisis experimental de los algoritmos presentados en este capítulo, en donde serán evaluados para determinar de forma más precisa su eficiencia, comparando los resultados con los resultados teóricos expuestos en este capítulo.

Capítulo 6

Contar puntos dentro de un rango en un k^2-tree.

6.1. Introducción.

Como se ha presentado en el Capítulo 4, existen diferentes variantes relacionadas con la consulta Skyline. Una de ellas es *Enumerating Skyline*, la cual será detallada en el Capítulo 7. Esta consulta responde con los puntos Skyline y la cantidad de puntos que domina cada punto Skyline. Para contar la cantidad de puntos que domina un punto, se requiere una consulta que permita contar los puntos existentes al interior de una región.

Dada una matriz binaria cuadrada de $n \times n$ con al menos un 1 en su interior e indexada en un k^2-tree, y un rango $r = [x_1, x_2] \times [y_1, y_2]$, el objetivo es encontrar la cantidad de unos dentro de la matriz que se encuentren dentro de r. En este capítulo se presenta *Compact Count*, un algoritmo que permite contar de forma eficiente la cantidad de puntos al interior de un rango en una matriz indexada en un k^2-tree.

En la Sección 6.2 se presentan los antecedentes de la literatura para responder esta consulta. En la Sección 6.3 se presenta *Compact Count*, su funcionamiento y un ejemplo. En la Sección 6.4 se presenta un análisis teórico del algoritmo, enfocado en encotrar el costo temporal y espacial de su funcionamiento. Finalmente, en la Sección 6.5 se presenta algunas conclusiones sobre lo expuesto en este capítulo.

6.2. Antecedentes.

Obtener la cantidad de puntos al interior de un rango en una matriz indexada en un k^2-tree no es algo que se haya presentado en la literatura con anterioridad. Existen otras propuestas que utilizan otra Estructura de Datos Compactas, como un Wavelet Tree [13].

En [27] se presenta un algoritmo llamado *Range*, que se encarga de identificar aquellos puntos que se encuentran al interior de r. De forma recursiva, divide cada submatriz y analiza sus hijos. Una forma de conseguir la cantidad de puntos existentes al interior de rango es ejecutar *Range* y contar la cantidad de puntos resultantes.

Una forma más simple de obtener este resultado es identificar los *Cuadrantes Maximales* que conforman el rango, y contar los puntos que contiene cada uno de manera eficiente.

Definición 6.1. *Un Cuadrante Maximal es una submatriz cuadrada del k^2-tree que se encuentra completamente contenida dentro del rango r, pero su nodo padre no cumple con esta condición.*

En este capítulo se presenta *Compact Count*, un algoritmo que identifica todos los Cuadrantes Maximales de r, y contabiliza de manera eficiente la cantidad de puntos que contiene cada uno de ellos.

6.3. Algoritmo propuesto: *Compact Count.*

El funcionamiento de este algoritmo aprovecha la estructuración del k^2-tree, gracias a la cual puede obtener con facilidad la cantidad de puntos almacenados en una submatriz cualquiera. Para contabilizar la cantidad de puntos al interior de una submatriz, se debe tener en cuenta la siguiente observación:

Observación 6.1. *Los descendientes hojas de una submatriz se encuentran ubicados de forma contigua dentro del bitmap L.*

Esta observación se produce debido a que a la hora de implementar el árbol conceptual que representa al k^2-tree, este es recorrido en anchura para almacenar sus nodos internos en el bitmap T. Lo anterior permite que los hijos de un nodo se encuentre ubicados de forma contigua en el bitmap, los hijos de dichos hijos se ubiquen de la misma forma, y de forma sucesiva se puede llegar a los últimos descendientes ubicados en el bitmap L.

Debido a lo anterior, el objetivo de contar cuántos puntos existen en una submatriz consiste en encontrar la ubicación del primer descendiente hoja en L, $firstChild$; y la ubicación del último descendiente hoja en L, $lastChild$. Finalmente, para obtener el resultado, simplemente se resta $rank_1(L, lastChild) - rank_1(L, firstChild)$. Para obtener dichas ubicaciones, se deben realizar diversas operaciones de $rank$ y y $select$ sobre T para descender por el k^2-tree.

Sea una n_i submatriz no vacía cuya ubicación en el bitmap T es la posición i, el primer hijo de n_i se ubica dentro del bitmap T en:

$$firstChild = k^2 rank_1(T, i) \tag{6.1}$$

Por otro lado, el último hijo se ubica dentro de T en:

$$lastChild = k^2 rank_1(T, i) + k^2 - 1 \tag{6.2}$$

La ecuación 6.1 y la ecuación 6.2 permiten obtener la ubicación tanto del primer hijo como del último hijo independiente si estos están vacíos o no. Continuar bajando utilizando la ubicación en T de un nodo vacío es un error, puesto que el k^2-tree no almacena esta información, y finalmente se llegaría a otro nodo no relacionado con el buscado. Por lo anterior, es necesario ajustar la búsqueda de los descendiente, y no solo ubicar al primer y último hijo, sino aquellos que no se encuentren vacíos.

Por lo anterior, se debe reajustar la búsqueda tanto del primer como del último hijo no vacío de tal forma que permita identificarlos en cualquier circunstancia. Es así como la ubicación en T del primer hijo no vacío de n_i es:

$$adjustFirstChild = select_1(T, rank_1(T, firstChild) + 1) \tag{6.3}$$

Mientras el último hijo no vacío se ubica dentro de T en:

$$adjustLastChild = select_1(T, rank_1(T, lastChild)) \tag{6.4}$$

Este reajuste no es necesario a la hora de llegar a los nodos hojas en el bitmap L, puesto que desde ese punto no se continuará bajando por el árbol, pero mientras se siga recorriendo el bitmap T, se debe realizar este reajuste para bajar por los nodos correctos.

Algoritmo 6.3.1 getN(bitmap T,bitmap L,int i)

```
1: if i is a leaf then
2:     return L[i]
3: end if
4: if T[i] = 0 then
5:     return 0
6: end if
7: firstChild = lastChild = i
8: while firstChild is not a leaf and lastChild is not a leaf do
9:     firstChild = k^2 rank_1(T, firstChild)
10:    if firstChild is not a leaf then
11:        firstChild = select_1(T, rank_1(T, firstChild) + 1)
12:    end if
13:    lastChild = k^2 rank_1(T, i) + k^2 − 1
14:    if lastChild is not a leaf then
15:        lastChild = select_1(T, rank_1(T, lastChild))
16:    end if
17: end while
18: return rank_1(T : L, lastChild) − rank_1(T : L, firstChild)
```

Es así como el Algoritmo 6.3.1 presenta el pseudocódigo de *getN*, un algoritmo que responde con la cantidad de puntos que se encuentran al interior de una submatriz representada en $T : L$ en la posición i. Inicialmente, en las líneas 1 y 4 se comprueba si el índice representa un nodo hoja y si la submatriz está vacía respectivamente. Ambas condiciones pueden comprobarse con rapidez, por lo que es lo primero en verificar. Posteriormente, en la línea 7 se puede observar que tanto el primer como el último hijo se ubican en n_i de manera inicial.

Posterior a ello, comienza el descenso por el k^2-tree. De forma iterativa se ubica tanto el primer como el último descendiente no vacío. Finalmente, en la línea 18 se ha encontrado el primer y último descendiente hoja no vacío. Finalmente se aplica un par de operaciones *rank* y se obtiene el resultado.

Cabe señalar, que de acuerdo a lo expuesto en el Capítulo 2, las estructuras necesarias para responder la consulta *rank* con un tiempo de orden constante no se encuentran implementadas sobre el bitmap L. Para responder tanto *Compact Count* como *getN* es necesario poder extender la estructura de tal forma de lograr *rank* sobre L, lo cual conlleva el uso de un poco más de espacio para mantener constante el tiempo de respuesta de esta consulta.

Algoritmo 6.3.2 CompactCount(ktree k, range r)

```
 1: total = 0
 2: Insert root in queue
 3: while queue ≠ ∅ do
 4:     n = queue tope
 5:     for each child n_i in n do
 6:         if n_i ≠ ∅ then
 7:             if n_i ⊆ r then
 8:                 if n_i is not a leaf then
 9:                     total = total + getN(k.T, k.L, n_i.i)
10:                 else
11:                     total = total + 1
12:                 end if
13:             else
14:                 if n_i ∩ r ≠ ∅ then
15:                     queue = queue ∪ n_i
16:                 end if
17:             end if
18:         end if
19:     end for
20: end while
21: return total
```

En el Algoritmo 6.3.2 se puede apreciar el pseudocódigo del algoritmo propuesto *Compact Count*. Este algoritmo subdivide las submatrices de forma iterativa, hasta encontrar aquellos Cuadrantes Maximales no vacíos para aplicar *getN* y sumar el resultado al *total*. Si una submatriz está vacía o no intersecta a r, es descartada. Si una submatriz intersecta a r pero no califica como un Cuadrante Maximal, es ingresada a una cola para ser revisada posteriormente. Esta cola, a diferencia del heap utilizado en el capítulo anterior, no ordena las submatrices bajo ningún criterio, debido a que este no es un tema que influya en la eficiencia de *Compact Count*.

Inicialmente la raíz del k^2-tree es ingresada a la cola. Cada vez que se extrae un nodo n de la cola, es subdividido en sus k^2 hijos n_i. Por cada hijo se consulta si el nodo está vacío (línea 6), y si consiste en un Cuadrante Maximal (línea 7). En caso de ser un Cuadrante Maximal, se llama a la función *getN* solo si no se trata de un nodo hoja. En caso de tratarse de una hoja, inmediatamente se suma 1 al total (recordando que ya se descartó que está vacía). Si no es un Cuadrante Maximal, se consulta si al menos intersecta a r (línea 14) para insertarlo a la cola.

6.3.1. Un ejemplo.

En la Figura 6.1 se puede observar un ejemplo de una matriz de 8×8 en donde se encuentra demarcado un rango $r = [0, 5] \times [0, 6]$ en su interior. También aparece su representación en un k^2-tree y su implementación en sus respectivos bitmaps T y L. Los espacios marcados en ambas representaciones muestran el camino que debe recorrer el algoritmo para encontrar la cantidad de puntos que existen dentro del rango.

En la Tabla 6.1 se puede observar los pasos realizados por el algoritmo para contabilizar los puntos al interior del rango marcado en la Figura 6.1. Para efectos de este ejemplo, los nodos que ingresan a la cola son representados bajo el formato $\langle n, (x, y), i\rangle$, en donde n indica el número que lo representa en el k^2-tree de la Figura 6.1, (x, y) son las coordenadas de la esquina más cercana al origen de la submatriz, e i muestra el índice que lo identifica dentro del bitmap T.

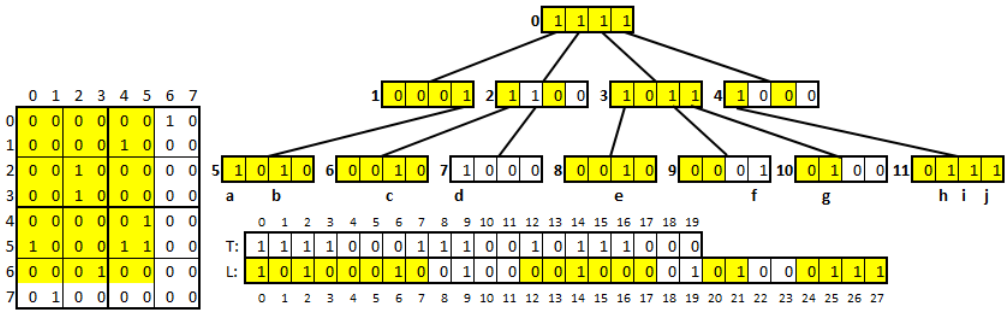


Figura 6.1: Ejemplo para la ejecución de *Compact Count*.

Acción	**Contenido de la cola**	*total*
Inicio	$\langle 0,(0,0),-1\rangle$	0
Expandir la raíz	$\langle 2,(4,0),1\rangle, \langle 3,(0,4),2\rangle, \langle 4,(4,4),3\rangle$	2
Expandir 2	$\langle 3,(0,4),2\rangle, \langle 4,(4,4),3\rangle$	3
Expandir 3	$\langle 4,(4,4),3\rangle, \langle 9,(0,6),14\rangle, \langle 10,(2,6),15\rangle$	4
Expandir 4	$\langle 9,(0,6),14\rangle, \langle 10,(2,6),15\rangle$	7
Expandir 9	$\langle 10,(2,6),15\rangle$	7
Expandir 10	$\emptyset$	8

Tabla 6.1: Ejecución de *Compact Count* sobre el k^2-tree de la Figura 6.1.

Inicialmente la raíz es insertada en la cola. Tras revisar sus hijos, el nodo 1 se encuentra completamente dentro del rango, por lo tanto se aplica la función *getN* para obtener la cantidad de puntos. Es aquí en donde el conteo asciende a 2. Las otras tres submatrices ingresan a la cola.

Al revisar el nodo 2, existen dos hijos que se encuentran completamente dentro del rango, y dos que se encuentran afuera. De los que se encuentran adentro, uno está vacío, y el otro (nodo 6), es contabilizada con *getN*, agregando un punto más al total.

Al revisar el nodo 3, dos nodos se encuentran completamente adentro del rango, y dos lo intersectan. De los nodos que se encuentran completamente contenidos en el rango, uno de ellos está vacío. El otro (nodo 8) agrega un punto más al conteo total. Los nodos 9 y 10 entran a la cola.

Al revisar el nodo 4, existe un nodo que se encuentra completamente contenido dentro del rango, y uno que lo intersecta. El nodo que se encuentra completamente contenido agrega 3 puntos más al total, mientras que el que lo intersecta está vacío.

Al revisar los nodos 9 y 10, el nodo 9 no aporta puntos al total, y el 10 aporta con un punto más, obteniendo un total de 8 puntos.

6.4. Análisis teórico.

6.4.1. Contar puntos dentro de una submatriz.

Inicialmente se discutirá respecto al costo que conlleva contabilizar la cantidad de puntos al interior de una submatriz n_i definida dentro del k^2-tree en la posición i dentro de $T : L$. En este caso, el costo es medido con respecto a la cantidad de operaciones *rank* y *select* realizadas.

Lema 6.1. *Contar la cantidad de puntos al interior de una submatriz n es de orden $O(h)$, en donde h corresponde a la altura de la submatriz.*

Demostración. Acceder al primer nodo hijo no vacío conlleva 3 operaciones de *rank* y *select*. De forma similar, se requieren 3 operaciones para encontrar al último hijo no vacío. Para poder descender por el k^2-tree hasta llegar a las hojas, se deben repetir estas operaciones h veces, haciendo un total de $6h$ operaciones. Esto equivale a un costo de orden $O(h)$. ■

Posteriormente, se debe determinar el costo de contar cuántos puntos posee un rango r, sumando la cantidad de puntos que puede aportar cada Cuadrante Maximal que lo compone.

Lema 6.2. *Un rango r al interior de una matriz indexada en un k^2-tree está compuesta por al menos un Cuadrante Maximal.*

Demostración. Al indexar una matriz en un k^2-tree, es subdividida en k^2 submatrices de forma recursiva hasta llegar a submatrices de 1×1. Por lo tanto si un rango r es de un tamaño 1×1, puede estar compuesto por al menos un Cuadrante Maximal de 1×1. Al considerar rangos más grandes, se pueden definir submatrices más grandes como Cuadrantes Maximales. ■

Lema 6.3. *La cantidad de puntos que posee un rango está compuesto por la suma de la cantidad de puntos que posee cada Cuadrante Maximal que lo compone.*

Demostración. Debido a lo expresado en el Lema 6.2, es evidente que si un rango está conformado por uno o más Cuadrantes Maximales, implica que la suma de la cantidad de puntos que contiene cada Cuadrante Maximal del rango es el total de puntos del rango. ■

Sea P la cantidad de puntos contenidos dentro de r, y N_i un Cuadrante Maximal que lo compone. Como r está conformado por m Cuadrantes Maximales ($m > 0$), se puede determinar que:

$$P = \sum_{i=1}^{m} N_i \tag{6.5}$$

6.4.2. Análisis temporal.

El costo temporal de obtener la cantidad de puntos al interior de un rango r está definido por la suma entre el costo de identificar los Cuadrantes Maximales y el costo de contar los puntos

dentro de dichos cuadrantes:

$$T(n) = T_{getM} + T_{countM} \tag{6.6}$$

Tras lo expresado en el Lema 6.1 y en la ecuación 6.5, se puede concluir que el costo de contar los puntos al interior de los Cuadrantes Maximales que conforman el rango r es de:

$$T_{countM} = 6 \sum_{i=1}^{m} h_i \tag{6.7}$$

En donde h_i corresponde a la altura del i-ésimo Cuadrante Maximal ($0 < h_i \leq H$).

Para identificar los Cuadrantes Maximales, se debe descender desde la raíz del k^2-tree hasta donde se encuentra cada uno de los m Cuadrantes Maximales. Sea H la altura del k^2-tree, por cada Cuadrante Maximal N_i debe recorrer $H - h_i$ niveles desde la raíz hasta la altura h_i de cada cuadrante. Por lo tanto, el costo de identificar todos los Cuadrantes Maximales es de:

$$T_{getM} = \sum_{i=1}^{m} (H - h_i) = mH - \sum_{i=1}^{m} h_i \tag{6.8}$$

Uniendo las ecuaciones 6.7 y 6.8 con 6.6, se tiene que:

$$T(n) = mH - \sum_{i=1}^{m} h_i + 6 \sum_{i=1}^{m} h_i = mH + 5 \sum_{i=1}^{m} h_i \tag{6.9}$$

Recordando que $h_i \leq H$, se puede concluir que $\sum_{i=1}^{m} h_i \leq \sum_{i=1}^{m} H = mH$. Aplicando esto a la ecuación 6.9, se puede definir una cota superior que indique el costo temporal:

$$T(n) \leq mH + 5mH = 6mH \tag{6.10}$$

Como $H = \log_k n$, en donde n es el tamaño de la matriz, finalmente el costo temporal es $T(n) \leq O(m \log_k n)$.

El costo temporal de *Compact Count* dependen tanto de la altura del k^2-tree como de la cantidad de Cuadrantes Maximales que componen el rango. En el mejor de los casos, el rango contiene solo un Cuadrante Maximal. Independiente del tamaño del cuadrante o de la cantidad de puntos que lo compone, el costo de contar los puntos es de orden $O(\log_k n)$.

Por otro lado, en el peor de los casos, se debe maximizar la cantidad de Cuadrantes Maximales m al interior del rango. En la Figura 6.2 se puede apreciar un ejemplo de ello. Aquí el rango $r = [0, 7] \times [1, 6]$ contiene 16 cuadrantes de 1×1 y 8 cuadrantes de 2×2, logrando un total de 24 Cuadrantes Maximales.

En general, en una matriz cuadrada de tamaño $n \times n$, el rango que maximiza m es toda la matriz menos las dos filas que se encuentran en la esquina superior e inferior, o las dos columnas ubicadas a la derecha e izquierda. La cantidad de Cuadrantes Maximales que componen este

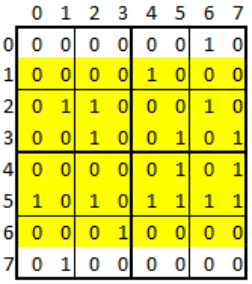


Figura 6.2: Ejemplo particular del peor caso para *Compact Count*.

rango es:

$$m = \sum_{i=0}^{H-2} \frac{2n}{k^i} = 2n \sum_{i=0}^{H-2} \left(\frac{1}{k}\right)^i \tag{6.11}$$

Como $\sum_{i=0}^{n-1} a^i = \frac{1-a^n}{1-a}$, entonces:

$$m = 2n \frac{1 - \left(\frac{1}{k}\right)^{H-1}}{1 - \left(\frac{1}{k}\right)} = \frac{2n(1 - k^{1-H})}{1 - k^{-1}} \tag{6.12}$$

En este caso se puede definir que el costo de contar los puntos al interior de este rango está definido por:

$$T(n) \leq 12nH \frac{1 - k^{1-H}}{1 - k^{-1}} \tag{6.13}$$

Con esto se puede concluir que en el peor de los casos la cota máxima del costo temporal de contar los puntos dentro de un rango es de orden $O(n \log_k n)$.

6.4.3. Análisis espacial.

Entre todas las estructuras utilizadas por *Compact Count*, destaca la cola en donde se almacenan las submatrices que intersectan al rango sin encontrarse completamente contenidas dentro de este.

En este sentido, asumiendo que los m Cuadrantes Maximales que conforman el rango provienen de padres diferentes, se puede concluir que a lo más la cola puede llegar a tener m elementos.

En el mejor de los casos, no existen submatrices que cumplan esta condición. El rango está compuesto solo por una submatriz, por lo tanto a lo más habrá solo una submatriz en la cola. En este caso, el espacio utilizado es orden constante.

Por otro lado, para el peor de los casos, m es un valor muy grande. El caso presentado en la Figura 6.2 refleja un buen ejemplo, pero solo basta un caso en donde se unan dos filas o columnas de extensión n y que cubran la frontera entre los k^2 hijos de la raíz. Un ejemplo aplicado a la

Figura 6.2 corresponde al rango $[0, 7] \times [3, 4]$. Con esto se puede concluir que el tamaño de la cola está acotada por $O(n)$.

También es importante considerar el espacio adicionar que requiere el k^2-tree para responder la consulta *rank* sobre el bitmap L. Esto requiere un espacio de orden $o(n)$, que es pequeño en comparación con la cola.

6.5. Conclusión.

En este capítulo se presentó *Compact Count*, un algoritmo que permite contar de manera eficiente la cantidad de puntos al interior de un rango para un conjunto de datos indexados en un k^2-tree. Para ello se basa en identificar los Cuadrantes Maximales que conforman el rango a contabilizar, y ayudado por sucesivas operaciones de *rank* y *select* sobre los bitmaps que implementan el k^2-tree cuenta la cantidad de puntos que posee cada Cuadrante Maximal de forma rápida y eficiente.

El costo temporal de *Compact Count* está definido por una cota superior de $O(m \log_k n)$, en donde n es el tamaño de la matriz y m es la cantidad de Cuadrantes Maximales del rango. Por otro lado, el costo espacial está definido por la cantidad de submatrices que intersectan tanto el rango como su complemento, los cuales son almacenados en una cola durante el proceso. La cantidad de submatrices que llegan a ser almacenadas es como máximo m, asumiendo que los padres de los m Cuadrantes Maximales son diferentes.

Por lo anterior, la eficiencia del algoritmo radica en la cantidad de Cuadrantes Maximales que compone el rango y no en la cantidad de puntos que posee. Puede que un rango cubra un área muy grande y con muchos puntos, pero si su interior está compuesto por una pequeña cantidad de Cuadrantes Maximales de gran tamaño, *Compact Count* puede obtener una importante ventaja frente a aquellos algoritmos cuya eficiencia si depende de la cantidad de puntos al interior del rango o en el tamaño del rango.

El peor de los casos ocurre cuando existen muchos Cuadrantes Maximales de tamaño mínimo al interior de un rango. Un caso bastante desfavorable es el presentado en la Figura 6.2. En estos casos, el costo temporal puede ascender al orden $O(n \log_k n)$, y orden lineal en cuanto al espacio. A pesar de ser un costo bastante eficiente, esta cota es dificilmente alcanzable, lo que demuestra que el comportamiento de *Compact Count* en la práctica es prometedor.

En conclusión *Compact Count* puede tener una amplia ventaja cuando tiene que revisar un rango con pocos Cuadrantes Maximales. El tamaño del rango y la cantidad de puntos que lo contiene no influye en la eficiencia del algoritmo, lo cual implica una importante ventaja de este algoritmo. Frente a lo escásamente propuesto en la literatura, *Compact Count* es una alternativa eficiente para aquellos problemas que requieren contar puntos al interior de un rango para un conjunto indexado en un k^2-tree.

Los resultados teóricos presentados en este capítulo serán contrastados con los resultados experimentales que serán expuestos en el Capítulo 10.

Capítulo 7

Variantes de la consulta Skyline.

7.1. Introducción.

La consulta Skyline presenta distintas variantes, las cuales resuelven diferentes problemas, pero manteniendo la escencia de la consulta. En el Capítulo 4 se pueden revisar las variantes más importantes que la literatura ha ofrecido. En este capítulo se revisarán algunas de estas variantes, y su funcionamiento en un conjunto de datos indexados en un k^2-tree.

La primera de ellas es *Enumerating Skyline*, que encuentra el Skyline de un conjunto, con la cantidad de puntos dominados por cada punto del Skyline. La cantidad de puntos que domina un punto Skyline p es conocido como num_p o $num(p)$. Esto es de utilidad debido a que este valor sirve como medida de *bondad* de p, y puede ser un dato adicional que ayude a los usuarios a tomar una decisión. En este capítulo se presenta *Enumerating Compact*, un algoritmo que permite obtener el resultado de la consulta Enumerating Skyline de manera eficiente, aprovechando las propiedades del k^2-tree y *Compact Count* presentado en el Capítulo 6.

La segunda variante a revisar es *Constrained Skyline Query*. Aqui se encuentra el Skyline al interior de un rango r. Este rango define las restricciones existentes por cada dimensión, dada por las limitaciones que poseen los usuarios. En el Ejemplo 3.1.1 puede existir una limitación en cuanto al dinero que el usuario posee para pagar un hotel, y la distancia límite en la que desea estar con respecto a la playa. En este capítulo se explicará la adaptación de BBSk para responder esta consulta.

En la Sección 7.2 se presenta un análisis de la variante Enumerating Skyline, y una propuesta para responder esta consulta en un k^2-tree. De forma similar, en la Sección 7.3 se presenta una propuesta para responder la consulta Constrained Skyline Query. Finalmente, en la Sección 7.4 se presentan algunas conclusiones sobre lo expuesto en este capítulo.

7.2. Enumerating Skyline.

En [36, 37] se propone calcular la consulta Enumerating Query sobre un R-tree realizando los siguientes pasos:

1. Calcular el Skyline normal utilizando algún algoritmo tradicional (como por ejemplo BBS).

2. Por cada punto p perteneciente al Skyline, contabilizar la cantidad de puntos contenidos al interior de la región de dominancia de p.

En caso de contabilizar los puntos en un k^2-tree, en donde puede ser útil utilizar *Compact Count*, presentado en el Capítulo 6, o *Range*, presentado en [27], sobre cada punto del Skyline. De esta forma, una solución eficiente es la presentada en el Algoritmo 7.2.1.

Algoritmo 7.2.1 enumeratingCount(ktree k)

```
1: S = BBSk(k)
2: for each point p in S do
3:     r = [p_x, n − 1] × [p_y, n − 1]
4:     num_p = compactCount(k, r) − 1
5: end for
6: return S
```

En la línea 1 se obtienen todos los puntos del Skyline utilizando el algoritmo presentado en la Sección 5.3. Posteriormente, por cada punto p Skyline, se define el rango de consulta como la región dominante de p, y posteriormente en la línea 4 se procede a contabilizar la cantidad de puntos existente dentro de dicho rango. Se resta 1 debido a que dicha consulta considera a p dentro de su cálculo.

Frente a esto se debe tener en cuenta que las regiones de dominancia de cada punto Skyline se superponen entre si. Al utilizar el Algoritmo 7.2.1 se están contando más de una vez la cantidad de puntos de aquellas submatrices no vacías que pertenecen a más de una región de dominancia.

7.2.1. Algoritmo propuesto: *Enumerating Compact.*

A continuación se presenta *Enumerating Compact*, un algoritmo que solo cuenta una vez la cantidad de puntos por cada submatriz perteneciente a la región de dominancia de S, y mantiene en memoria dicho resultado para tenerlo disponible en caso de ser consultado más de una vez.

Para mantener el valor de la cantidad de puntos sobre cada submatriz, se propone el uso de un arreglo N del tamaño equivalente a la cantidad de nodos internos del k^2-tree. Este valor se obtiene aplicando la consulta *rank* sobre la última posición de T. Para obtener la posición dentro de N en donde se almacena la cantidad de puntos de un nodo interno, se debe aplicar una operación *rank* sobre T en la posición i que representa a dicho nodo.

En la Figura 7.1 se puede apreciar el ejemplo de la Figura 5.2 con su respectivo arreglo N. Se puede apreciar que en las cuatro primeras ubicaciones de N aparece la cantidad de puntos que posee cada una de las submatrices hijas de la raíz. En la siguiente posición aparece la cantidad de puntos que posee la siguiente submatriz no vacía. Realizando el mismo recorrido en anchura sobre el k^2-tree, N muestra la cantidad de puntos que tiene cada una de las submatrices hasta llegar a aquellas que son de tamaño $k \times k$, o como en este ejemplo, 2×2.

Una forma eficiente de calcular los valores de N es durante la ejecución del algoritmo que responde la consulta. Se debe tener en cuenta que no se requieren todos los valores de N, sino solo aquellos que mantienen la cantidad de puntos de los Cuadrantes Maximales de la región de dominancia del Skyline. De acuerdo al Lema 6.1, calcular la cantidad de puntos que contiene una submatriz es de orden $O(h)$, donde h es la altura de dicha submatriz.

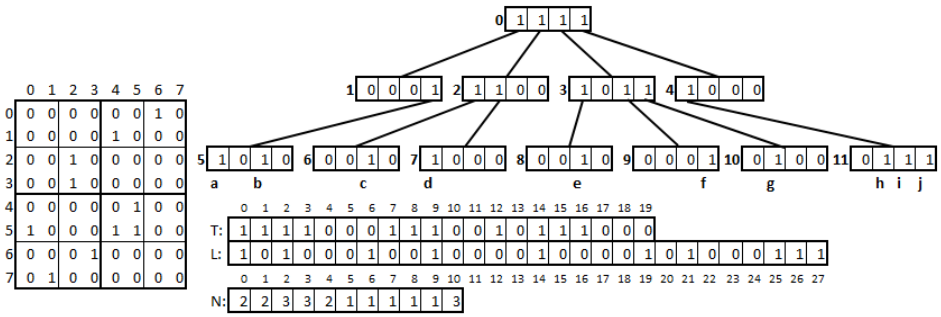


Figura 7.1: Ejemplo de la aplicación de N.

Algoritmo 7.2.2 enumeratingCompact(ktree k)

```
1: S = ∅
2: Insert root in heap
3: while heap ≠ ∅ do
4:     n = heap top
5:     if n is not dominated by S then
6:         for each child n_i in n do
7:             if n_i ≠ ∅ and n_i is not dominated by S then
8:                 if n_i is not dominated by its sibling not empty then
9:                     if n_i is not a leaf then
10:                        heap = heap ∪ n_i
11:                    else
12:                        r = [x_{n_i}, n − 1] × [y_{n_i}, n − 1]
13:                        num_{n_i} = enumartingCount(k, r) − 1
14:                        S = S ∪ n_i
15:                    end if
16:                else
17:                    if N[rank_1(k.T, i)] is not calculated then
18:                        N[rank_1(k.T, i)] = getN(k.T, k.L, i)
19:                    end if
20:                end if
21:            end if
22:        end for
23:    end if
24: end while
25: return S
```

En el Algoritmo 7.2.2 se presenta *Enumerating Compact*. Básicamente funciona de la misma manera que BBSk, pero con ciertas diferencias que permiten mejorar el cálculo de $num(p)$ sobre los puntos del Skyline. En la línea 8 se puede observar que la condición de dominancia entre nodos hermanos se encuentra separada del resto de las condiciones. Esto porque en caso de existir una submatriz dominada por sus hermanos, significa que sus puntos pertenecen a la región de dominancia del Skyline, por lo tanto es importante conocer la cantidad de puntos que posee. Por lo anterior, en la línea 18 se realiza el cálculo de la cantidad de puntos que posee la submatriz utilizando *getN*.

En la línea 13 es en donde, tras encontrar un punto Skyline, se cuenta la cantidad de puntos que domina. Se define su región de dominancia r, y se contabiliza utilizando la función *Enumerating Count*. Esta es una adaptación de *Compact Count*, con la diferencia que al obtener la cantidad

Acción	N	**Contenido de** S
Inicio	$[0,0,0,0,0,0,0,0,0,0,0]$	$\emptyset$
Expandir 0	$[0,0,0,3,0,0,0,0,0,0,0]$	$\emptyset$
Expandir 1	$[0,0,0,3,0,0,0,0,0,0,0]$	$\emptyset$
Expandir 5	$[0,0,0,3,0,0,0,0,0,0,0]$	$\langle a,5\rangle$
Expandir 2	$[0,0,0,3,0,0,0,0,0,0,0]$	$\langle a,5\rangle$
Expandir 6	$[0,0,0,3,0,0,0,0,0,0,0]$	$\langle a,5\rangle, \langle c,3\rangle$
Expandir 3	$[0,0,0,3,0,0,0,0,0,1,0]$	$\langle a,5\rangle, \langle c,3\rangle$
Expandir 8	$[0,0,0,3,0,0,0,0,1,1,0]$	$\langle a,5\rangle, \langle c,3\rangle, \langle e,4\rangle$
Expandir 7	$[0,0,0,3,0,0,0,0,1,1,0]$	$\langle a,5\rangle, \langle c,3\rangle, \langle e,4\rangle, \langle d,0\rangle$

Tabla 7.1: Ejecución de *Enumerating Compact* sobre el k^2-tree de la Figura 7.1.

de puntos perteneciente a cada Cuadrante Maximal no vacío, inicialmente consulta en N, y en caso de no existir dicho valor, lo calcula utilizando *getN* y lo almacena en N.

Algoritmo 7.2.3 enumartingCount(ktree k, range r)

```
1: total = 0
2: Insert root in queue
3: while queue ≠ ∅ do
4:     n = queue tope
5:     for each child n_i in n do
6:         if n_i ≠ ∅ then
7:             if n_i ⊆ r then
8:                 if n_i is not a leaf then
9:                     if N[rank_1(k.T, i)] is not calculated then
10:                        N[rank_1(k.T, i)] = getN(k.T, k.L, n_i.i)
11:                    end if
12:                    total = total + N[rank_1(k.T, i)]
13:                else
14:                    total = total + 1
15:                end if
16:            else
17:                if n_i ∩ r ≠ ∅ then
18:                    queue = queue ∪ n_i
19:                end if
20:            end if
21:        end if
22:    end for
23: end while
24: return total
```

En el Algoritmo 7.2.3 se puede apreciar el pseudocódigo de *Enumerating Count*. La diferencia con *Compact Count* radica en la línea 10, en donde se calcula la cantidad de puntos contenidos en la submatriz n_i en caso de no haber sido calculado previamente.

7.2.2. Un ejemplo.

En la Tabla 7.1 se puede apreciar el proceso de ejecución de *Enumerating Compact*. En este caso se revisa cómo evoluciona N en la medida en que se va ejecutando el algoritmo, y a su vez

se revisa el crecimiento de S, que incluye por cada punto su identificador y la cantidad de puntos que domina. Tras la ejecución se puede concluir que a es el mejor punto Skyline debido a que el que domina más puntos con un total de cinco.

Tras expandir la raíz, el nodo 4 es podado, por lo tanto, es necesario conocer la cantidad de puntos que posee, completando en $N[3] = 3$ puntos. De forma similar ocurre al expandir el nodo 3 y podar el nodo 10. Al encontrar a y c se utiliza el cálculo de $N[3]$. Al encontrar e se utiliza el cálculo de la cantidad de puntos sobre el nodo 10.

Al final fue necesario utilizar 3 de 11 espacios de N, con la ventaja de poder acceder a la cantidad de puntos por nodo en un tiempo constante. Esto también demuestra la eficacia de calcular los valores de N durante la ejecución, puesto que en este ejemplo hubo 8 submatrices sobre las cuales no fue necesario contar los puntos que contenían.

7.2.3. Análisis teórico.

El algoritmo *Enumerating Compact* es una combinación de BBSk con sucesivas aplicaciones de *Compact Count* sobre las regiones de dominancia de cada punto Skyline. Frente a esto, se debe tener en cuenta lo siguiente:

Lema 7.1. *La región de dominancia de dos puntos siempre se intersectará en al menos un cuadrante.*

Demostración. Considere dos puntos $p = (x_p, y_p)$ y $q = (x_q, y_q)$ cuyas regiones de dominancia no se intersectan. Para que esto ocurra, los rangos de ambas regiones ya sea en el eje X o en el eje Y no deben intersectarse. Sea n el tamaño de la matriz, en donde $x_p < n$, $x_q < n$, $y_p < n$ y $y_q < n$. La región de dominancia de p es $r_p = [x_p, n] \times [y_p, n]$, y la región de dominancia de q es $r_q = [x_q, n] \times [y_q, n]$. Como todas las coordenadas son menores a n, entonces $[x_p, n] \cap [x_q, n] \neq \emptyset$ y $[y_p, n] \cap [y_q, n] \neq \emptyset$. Como los rangos tanto en X como en Y de r_p y r_q se intersectan, entonces $r_p \cap r_q \neq \emptyset$. Sea $r = r_p \cap r_q$, debido al Lema 6.2, r está compuesto por al menos un cuadrante. ■

Es así como $m_t = m_{r_p} + m_{r_q} - m_r$, en donde m_{r_p} corresponde a la cantidad de Cuadrantes Maximales de r_p, m_{r_q} corresponde a la cantidad de Cuadrantes Maximales de de r_q, y m_r corresponde a la cantidad de Cuadrantes Maximales de r. Frente a esto se puede decir que:

Corollario 7.1. *La cantidad de Cuadrantes Maximales de la región de dominancia del Skyline es menor o igual que la suma de los Cuadrantes Maximales de las regiones de dominancia de cada punto del Skyline.*

Esto se traduce en:

$$m_t \leq \sum_{i=1}^{|S|} m_i$$

Lema 7.2. *Enumerating Compact cuenta la cantidad de puntos pertenecientes sobre m_t submatrices, que equivale a la cantidad de Cuadrantes Maximales de la región de dominancia del Skyline.*

Demostración. Esto es fácil de ver en la línea 10 del Algoritmo 7.2.3. Cada vez que se está por contar la cantidad de puntos dentro de una submatriz, primero se consulta si ya fue previamente calculado. De ser así, se evita todo cálculo y se consulta el valor en el arreglo N.

Por otro lado, en la línea 18 del Algoritmo 7.2.2 no es necesario consultar si la cantidad de puntos de aquella submatriz fue previamente calculada. Aquella submatriz se trata de un nodo hijo de una submatriz recién extraída del heap, por lo tanto no ha sido explorada. ■

Análisis temporal.

Enumerating Compact es una combinación de BBSk con sucesivas aplicaciones de *Compact Count* sobre las regiones de dominancia de cada punto de Skyline. Tras lo visto en el Lema 7.2, *Enumerating Compact* cuenta una sola vez por cada Cuadrante Maximal de la región de dominancia del Skyline. Por lo tanto, las sucesivas aplicaciones de *Compact Count* equivalen a realizar una vez *Compact Count* sobre un rango con m_t Cuadrantes Maximales.

Frente a esto, se puede concluir que el costo temporal del algoritmo está definido por:

$$T(n) = T_{bbsk} + T_{count} \tag{7.1}$$

Tras lo expresado en la ecuación 5.4, el costo temporal de aplicar BBSk es de:

$$T_{bbsk} \leq 2(k-1)|S|^2 \log_k n + |S|^2 = O(k|S|^2 \log_k) \tag{7.2}$$

Por otro lado, el costo temporal de aplicar *Compact Count* es de:

$$T_{count} \leq 6m_t H = O(m_t \log_k n) \tag{7.3}$$

Uniendo las ecuaciones 7.1, 7.2 y 7.3, se tiene que:

$$T(n) \leq 2(k-1)|S|^2 H + |S|^2 + 6m_t H = (2(k-1)|S|^2 + 6m_t)H + |S|^2 \tag{7.4}$$

Finalmente, el costo temporal de *Enumerating Compact* es de orden:

$$T(n) \leq O((k|S|^2 + m_t) \log_k n) \tag{7.5}$$

El costo temporal de *Enumerating Compact* depende principalmente del valor k del k^2-tree, la cantidad de puntos Skyline $|S|$ y la cantidad de Cuadrantes Maximales que componen la región de dominancia del Skyline m_t.

En el mejor de los casos, tanto BBSk como *Compact Count* utiliza un tiempo de orden $O(k \log_k n)$. Esto ocurre cuando solo existe un punto Skyline y su región de dominancia contiene solo un Cuadrante Maximal.

En el peor de los casos, considerando el peor de los casos de BBSk, presentado en la Figura 5.4, existen n puntos Skyline, y una cantidad significativa de Cuadrantes Maximales. Por otro lado el peor caso de *Compact Count*, se puede considerar que existe un punto Skyline, con una región de dominancia como la presentada en la Figura 6.2. De los dos casos presentados, el peor caso para *Enumerating Compact* es el mismo que el de BBSk, en donde el costo temporal es de

orden $O(kn^2 \log_k n)$.

Análisis espacial.

De la misma forma que como se lleva explicando en esta sección, el costo espacial de *Enumerating Compact* es equivalente al costo espacial de BBSk más el costo espacial de *Compact Count*.

Tras lo expresado en la Sección 5.4, el costo espacial de BBSk conserva una cota superior de orden $O(k|S| \log_k n)$. Por otro lado, el costo espacial de *Compact Count* es m, que equivale a la cola en donde se almacenan las submatrices que intersectan al rango sin encontrarse completamente contenida.

En este sentido, se debe considerar que cada vez que *Compact Count* es ejecutado, se utiliza un espacio equivalente a m_i submatrices, que corresponde a la cantidad de elementos de la cola. Cada cola de cada ejecución es independiente de la cola de otra ejecución.

Sea $m_{max} = \max_{i=1}^{|S|} m_i$ la cantidad de Cuadrantes Maximales de aquel punto Skyline cuya región de dominancia posee la mayor cantidad de Cuadrantes Maximales. Es así como el costo espacial de *Compact Count* es m_{max}.

Finalmente, el costo espacial de *Enumerating Compact* es de:

$$T(n) \leq O(k|S| \log_k n) + m_{max}$$

En el mejor de los casos, BBSk y *Compact Count* utilizan un espacio de $O(1)$, y ocurre en una situación similar al mejor caso temporal: solo existe un punto Skyline, y su región de dominancia solo contiene un Cuadrante Maximal.

En el peor de los casos, BBSk utiliza un espacio de orden $O(kn \log_k n)$, mientras que *Compact Count* utiliza un espacio de orden $O(n)$. En este sentido, el peor caso de BBSk es más costoso que el de *Compact Count*, por lo tanto el peor caso de BBSk afecta de peor manera el espacio utilizado en *Enumerating Compact*.

Cabe señalar que debido a la aplicación de la consulta *Compact Count*, el k^2-tree requiere aplicar la estructura necesaria para realizar la consulta *rank* sobre su bitmap L, lo cual requiere un espacio adicional del orden $o(n)$. Además, también se debe tener en cuenta el espacio necesario para trabajar con N, el cual también es de $o(n)$. En ambos casos, el tamaño depende de la cantidad de bits dentro de $T : L$, que está en directa relación con el tamaño del conjunto de datos.

7.3. Constrained Skyline Query.

Constrained Skyline Query es una variante en donde se obtiene el Skyline al interior de un rango r. Este es un rango rectangular en todas las dimensiones, y representa las restricciones presentadas por los usuarios.

En [36, 37] se explica BBS puede responder con esta consulta sobre datos indexados en un R-tree agregando la restricción presentada en r a la consulta. Por cada nodo revisado, si este no intersecta a r, es descartado.

De forma similar, BBSk puede ser adapatado para responder esta consulta sobre datos inde-

xados en un k^2-tree. De las tres condiciones de descarte presentada en la Sección 5.3, se incluye una cuarta condición en donde un nodo es descartado si no intersecta a r.

Por otro lado, la tercera condición de descarte presentada en la Sección 5.3 que dice relación con la dominancia entre nodos hermanos sufre una importante variación. En la medida en que se revisan los nodos del k^2-tree, solo se puede descubrir de forma rápida si están vacíos o no, pero no se puede identificar la ubicación de el o los puntos en su interior. Sea n una submatriz no vacía y que intersecta tanto a r como a su complemento (es decir, el espacio fuera de r). Saber que n no está vacía implica que existe al menos un punto en su interior, sin poder determinar en dónde se ubica. Si todos los puntos de n se ubican fuera de r, automáticamente estos puntos quedan fuera del rango de consulta de esta variante, y no podrán dominar puntos. Por la misma razón, si los puntos de n no se encuentran al interior de r, n no podrá podar a sus hermanos aunque los domine.

Por lo anterior, se debe considerar que la tercera condición de descarte solo podrá ser aplicada sobre un nodo si *su padre se encuentra completamente contenido en r*. De esta manera se asegura que todos sus hermanos se encuentren al interior de r, los puntos que contengan dichos hermanos también se encuentren al interior de r y poder aplicar sin problemas esta condición de descarte.

Algoritmo 7.3.1 BBSkConstrained(ktree k, range r)

```
 1: Insert root in heap
 2: S = ∅
 3: while heap ≠ ∅ do
 4:    n = heap top
 5:    if n is not dominated by S then
 6:       for each child n_i in n do
 7:          if n_i ≠ ∅ and n_i is not dominated by S then
 8:             if n ⊆ r and  then
 9:                if n_i is not dominated by its sibling not empty then
10:                   if n_i is not a leaf then
11:                      heap = heap ∪ n_i
12:                   else
13:                      S = S ∪ n_i
14:                   end if
15:                end if
16:             else
17:                if n_i ∩ r ≠ ∅ then
18:                   heap = heap ∪ n_i
19:                end if
20:             end if
21:          end if
22:       end for
23:    end if
24: end while
25: return S
```

En el Algoritmo 7.3.1 se presenta una adaptación de BBSk para responder la consulta Constrained Skyline. Como se puede apreciar en la línea 9, la tercera condición de descarte fue separada del resto, para ser aplicada luego de cerciorarse que su padre se encuentra al interior de r, como aparece en la línea 8. En este caso no se pregunta por la intersección de n_i con r, puesto que al confirmar que su padre se encuentra dentro de r, los hijos también lo estarán. Una situación diferente ocurre en la línea 18, en donde el nodo padre no se encuentra al interior de r. En este caso se aplica la cuarta condición de descarte, consultando por la intersección del nodo con r.

7.4. Conclusión

En este capítulo se estudiaron dos variantes de la consulta Skyline y su aplicación sobre datos indexados en un k^2-tree. La primera de ellas, *Enumerating Skyline* entrega el Skyline en conjunto con la cantidad de puntos que domina cada punto Skyline. La segunda, *Constrained Skyline Query*, encuentra los puntos Skyline al interior de un rango.

Para la consulta Enumerating Skyline se propuso un nuevo algoritmo: *Enumerating Compact*, el cual se basa en BBSk y *Compact Count* para responder con esta variante. Como una forma de mejorar su eficiencia, el algoritmo trabaja con un arreglo N que almacena la cantidad de puntos que tiene cada submatriz no vacía del k^2-tree. Esto permite evitar calcular dos veces la cantidad de puntos de una submatriz que intersecta la región de dominancia de dos puntos Skyline. El cálculo se realiza durante la ejecución para evitar calcular la cantidad de puntos al interior de submatrices que no forman parte de la región de dominancia del Skyline.

Enumerating Compact tiene un costo temporal de orden $O((k|S|^2 + m_t) \log_k n)$, en donde n es el tamaño de la matriz, $|S|$ es la cantidad de puntos Skyline, y m_t la cantidad de Cuadrantes Maximales de la región de dominancia del Skyline. Por otro lado, el costo espacial viene dado por $O(k|S| \log_k n) + m_{max}$, en donde m_{max} corresponde a la cantidad de Cuadrantes Maximales que posee la región de dominancia del punto Skyline con más Cuadrantes Maximales.

Es así como el aumento de la cantidad de puntos Skyline y la cantidad de Cuadrantes Maximales que posee la región de dominancia del Skyline definen la eficiencia del algoritmo. Este último punto depende de la ubicación que tienen los puntos Skyline dentro de la matriz. Por lo tanto, la cantidad y ubicación de los puntos Skyline definen la eficiencia del algoritmo.

En el peor de los casos, debido a la cantidad de puntos Skyline que puede existir dentro de la matriz, el costo temporal puede alcanzar un orden máximo de $O(kn^2 \log_k n)$, mientras que el costo espacial puede alcanzar un orden máximo de $O(kn \log_k n)$.

Estos resultados son los primeros que consideran el uso de una estructura de datos que compacta la información, como es el caso del k^2-tree. Además, debido a que se encuentra basado en BBSk, el algoritmo obtiene una importante ventaja al aplicarse sobre datos clusterizados, porque en esta distribución disminuye la cantidad de puntos Skyline.

Por otro lado, se propone una adaptación de BBSk para responder Constrained Skyline Query. En esta, la intersección del rango se convierte en una nueva condición de descarte a la hora de definir cuáles nodos revisar. Por otra parte, la aplicación de la tercera condición que dice relación con la dominancia entre nodos hermanos requiere una precondición que exige que el nodo padre se encuentre al interior del rango de consulta.

Finalmente, es posible responder algunas variantes de la consulta Skyline sobre datos indexados en un k^2-tree. Adaptar BBSk para responder estas variantes es posible, y permite responder la consulta ocupando los recursos de forma eficiente. *Enumerating Compact* permite mejorar el uso de recursos frente al uso independiente de BBSk y *Compact Count* para responder con esta consulta.

En el Capítulo 11 se presentará la evaluación experimental de *Enumerating Compact*, para determinar de forma empírica su comportamiento frente a la aplicación independiente de BBSk y *Compact Count*.

Capítulo 8

Conclusiones de los algoritmos propuestos.

En esta parte de la tesis se presentaron una serie de algoritmos que permiten resolver tanto la consulta Skyline como una de sus variantes frente a datos indexados en un k^2-tree. Las propiedades de esta Estructura de Datos Compactas permitió conseguir importantes mejoras en la eficiencia de los algoritmos propuestos, para así conseguir responder sus respectivas consultas en un menor tiempo.

En la Tabla 8.1 se puede apreciar un resumen de la complejidad algorítmica temporal de todos los algoritmos presentados en esta parte de la tesis. En general, el costo oscila entre $O(\log_k n)$ y $O(kn^2 \log_k n)$ en el mejor y peor de los casos respectivamente, en donde n es el tamaño de la matriz representada por el k^2-tree. El mejor de los casos representa el costo de tener que recorrer el k^2-tree desde la raíz hasta las hojas. El peor de los casos representa una cota superior frente a la que casi nunca se llegará a igualar. Esto es un resultado prometedor ya que por un lado se realizan propuestas que en la literatura no se han explorado; y por otro lado se demuestra que el uso de recursos de los algoritmos propuestos es inferior a la de aquellos algoritmos presentes en la literatura y que son comparables a los algoritmos propuestos en esta tesis. La complejidad de los algoritmos propuestos en esta tesis no depende de la cantidad de puntos del conjunto, sino de otros factores que generan un menor impacto en el costo temporal.

En la Tabla 8.2 se presenta la complejidad algorítmica del uso espacial de los algoritmos propuestos. El mejor de los casos es de orden constante, el cual es un resultado mucho mejor que el mejor de los casos temporal, y aceptable en comparación con los algoritmos presentados en la

Nombre	Costo temporal	Mejor Caso	Peor Caso
BBSk	$\leq O(k\|S\|^2 \log_k n)$	$O(k \log_k n)$	$\leq O(kn^2 \log_k n)$
Skyline X	$\leq O(k\|S\| \log_k n)$	$O(k \log_k n)$	$\leq O(kn \log_k n)$
Compact Count	$\leq O(m \log_k n)$	$O(\log_k n)$	$\leq O(n \log_k n)$
Enumerating Compact	$\leq O((k\|S\|^2 + m_t) \log_k n)$	$O(k \log_k n)$	$\leq O(kn^2 \log_k n)$

Tabla 8.1: Tabla resumen de la complejidad algorítmica temporal de los algoritmos presentados.

Nombre	Costo espacial	Mejor Caso	Peor Caso
BBSk	$\leq O(k\|S\|\log_k n)$	$O(1)$	$\leq O(kn\log_k n)$
Skyline X	$\leq O(k\|S\|\log_k n)$	$O(1)$	$\leq O(kn\log_k n)$
Compact Count	$\leq m$	$O(1)$	$O(n)$
Enumerating Compact	$\leq O(k\|S\|\log_k n) + m_{max}$	$O(1)$	$O(kn\log_k n)$

Tabla 8.2: Tabla resumen de la complejidad algorítmica espacial de los algoritmos presentados.

literatura. El peor de los casos por su parte, oscila entre el orden $O(n)$ y $O(n\log_k n)$, en donde n es el tamaño de la matriz. Estos resultados representan una cota superior del uso de los recursos de memoria y que de forma similar al costo temporal, difícilmente se llega a alcanzar. Aún así, este costo es superior al espacio utilizado por los algoritmos presentes en la literatura. En pocas palabras, los algoritmos propuestos en esta tesis presentan un *trade-off* entre almacenamiento y tiempo de ejecución.

En general, la complejidad de los algoritmos propuestos en esta parte dependen tanto de la cantidad de puntos Skyline como de la cantidad de Cuadrantes Maximales a analizar. La cantidad de puntos Skyline dependen de la distribución de los datos y de la cantidad de puntos indexados. Una distribución clusterizada y una cantidad baja de puntos indexados tiende a disminuir la cantidad de puntos Skyline. Por otra parte, la cantidad de Cuadrantes Maximales depende de la forma que tenga el rango de consulta, independiente de su extensión. La ubicación de los puntos Skyline determina la forma del rango de consulta en *Enumerating Compact*.

Finalmente, los algoritmos presentados en esta parte presentan una importante mejora frente a la complejidad temporal, a cambio de un costo superior en cuanto a la complejidad espacial. Esto puede ser contrarrestado gracias a las propiedades del k^2-tree y a sus propiedades de compactación. Cabe señalar que gracias a las mismas propiedades es posible realizar las mejoras correspondientes en cuanto al costo temporal.

Parte IV

Implementación y Experimentación.

Capítulo 9

BBS k^2-tree (BBSk).

En este capítulo se revisará el proceso de experimentación para medir la eficiencia tanto de BBSk como de *Skyline X*. El objetivo es medir el tiempo y la memoria utilizada en su ejecución y analizar los resultados obtenidos.

En la Sección 9.1 se presenta el proceso de implementación de los algoritmos que serán evaluados. En la Sección 9.2 se presentan los datos utilizados en la experimentanción, tanto su formato, características, configuraciones y transformaciones. En la Sección 9.3 se presentan los algoritmos que serán evaluados en esta experimentación. En la Sección 9.4 se presenta el entorno en donde se realizaron las pruebas. En la Sección 9.5 se presentan los resultados experimentales tras realizar las pruebas. Finalmente, en la Sección 9.6 se presentan algunas conclusiones de los resultados encontrados.

9.1. Implementación.

La librería utilizada en la experimentación para implementar la estructura k^2-tree es la implementada en [27]. Es una librería programada en C, y que se encarga de la implementación tanto de los bitmaps T y L como de las operaciones *rank* y *select*.

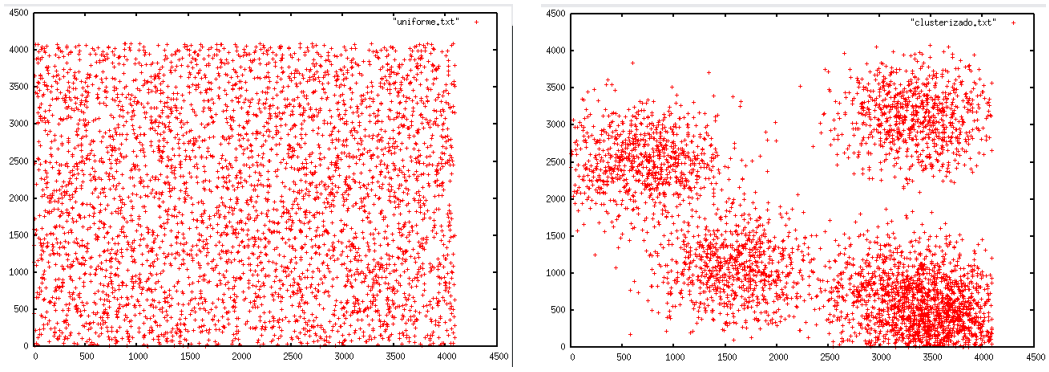
La implementación de los algoritmos fue realizada en C++, bajo el paradigma de orientación a objetos, permitiendo así facilitar la comprensión del código. Los algoritmos implementados fueron BBSk, *Skyline X* y una versión ingenua cuyo peudocódigo se presenta en el Algoritmo 9.1.1. BBSk fue implementado con la posibilidad de escoger dos funciones de distancia para ordenar las submatrices dentro del heap: la distancia de Manhattan o norma L1 $(x + y)$, y la distancia Euclideana $(x^2 + y^2)$ [1].

En lo que respecta al Algoritmo 9.1.1, trabaja sobre datos no indexados en ninguna estructura. En el peor de los casos, el algoritmo demora un tiempo de orden $O(n^2)$, en donde n corresponde a la cantidad de puntos de la lista.

[1] Aunque la verdadera distancia Euclideana incluye una raíz cuadrada, en esta experimentación se decidió no utilizar para facilitar el procesamiento computacional obteniendo el mismo efecto.

Algoritmo 9.1.1 skylineIngenuo(List L)

```
1: S = ∅
2: for each point p in L do
3:     pSkyline = true
4:     for each point q in L do
5:         if p is dominated by q then
6:             pSkyline = false
7:             break
8:         end if
9:     end for
10:    if pSkyline = true then
11:        S = S ∪ p
12:    end if
13: end for
14: return S
```



(a) Ejemplo distribucón uniforme (b) Ejemplo distribución clusterizada

Figura 9.1: Ejemplo de distribuciones utilizadas para la experimentación.

9.2. Datos experimentales.

Se trabajó con datos sintéticos generados usando una librería en java. Esta librería permite generar una cantidad n de puntos aleatorios diferentes cuyas coordenadas poseen valores entre el 0 y un cierto valor límite l. Los valores n y l son entregados por parámetro. Los puntos resultantes son almacenados en un archivo de texto plano, en donde por cada línea del archivo se coloca el valor X e Y de cada punto separado por un espacio.

Se generaron datos con dos distribuciones: uniforme y clusterizada. La diferencia entre cada una se puede apreciar en la Figura 9.1(a) y la Figura 9.1(b). La distribución clusterizada selecciona algunos puntos y genera clusteres en donde se concentran los puntos.

Por cada una de las distribuciones se generaron dos tipos de conjuntos de datos: conjuntos para evaluar tanto el efecto del tamaño como la densidad del conjunto. Los conjuntos para experimentar con la cantidad de puntos utilizaron una configuración con una cantidad de puntos que oscila entre los 100000 y 50000000 de puntos, al interior de un rango en donde $l = 10000000$ Los conjuntos para experimentar con la densidad, generaron puntos al interior de un rango en donde $l = 4096$, trabajando con una densidad del 1 %, 5 % y 10 %. Se generaron 10 archivos por

Cantidad de puntos	Archivo sin indexar (KB)	Archivo indexado (KB)
100000	1541	757
200000	3082	1462
300000	4622	2148
400000	6163	2821
500000	7704	3485
600000	9245	4141
700000	10786	4792
800000	12326	5437
900000	13867	6077
1000000	15408	6713
2000000	30816	12913
5000000	77040	30587
10000000	154079	58613
50000000	770399	263298

Tabla 9.1: Tamaño del conjunto sin indexar e indexado en función de la cantidad de puntos.

cada confguración, rango y cantidad de puntos definido. Por cada conjunto se construyó el k^2-tree considerando un valor $k = 2$.

En la Tabla 9.1 se presenta el tamaño en memoria de los archivos que almacenan los puntos de cada conjunto sin indexar vs. su correspondiente representación en el k^2-tree, en función de la cantidad de puntos. Se puede ver que aquellos archivos que almacenan la representación del k^2-tree poseen un tamaño inferior en comparación con los archivos que almacenan los datos sin compactar.

9.3. Línea de comparación (Baseline).

En esta experimentación, se realizará una comparación de los siguientes algoritmos:

- `Skyline Ingenuo`: Es el algoritmo presentado en la Sección 9.1, el cual es aplicado directamente sobre los datos sin compactar.
- `BBSk Manhattan`: Es el algoritmo *BBS k^2-tree* presentado en el Capítulo 5, utilizando la distancia Manhattan para ordenar las submatrices dentro del heap.
- `BBSk Euclidean`: Es el algoritmo *BBS k^2-tree* utilizando la distancia Euclideana para ordenar las submatrices dentro del heap.
- `Skyline X`: Es el algoritmo presentado en la Sección 5.5.

Cabe destacar que tanto BBSk como *Skyline X* trabajan con los datos indexados en un k^2-tree, mientras que el algoritmo Skyline Ingenuo trabaja sobre los datos sin compactar.

9.4. Entorno de pruebas.

Las pruebas de los algoritmos implementados en este capítulo fueron ejecutadas en un servidor con 4 procesadores Intel(R) Xeon(R) CPU E3-1225 de 3.30GHz, con 8192 KB de memoria caché. La RAM fue de tipo DDR3 de 32GB de capacidad. El servidor posee una instalación de Linux Debian 4.9.130 de 64 bits. Los algoritmos fueron implementados sobre el lenguaje de programación C y C++ y compilados con gcc 6.3.0.

Para la medición del tiempo, se utilizó la función `clock()` de la librería `times.h` implementada en C. En esta medición no se consideró el tiempo de lectura/escritura, por lo tanto previo a la medición, los datos fueron cargados a memoria principal. Por cada archivo se repitieron las operaciones 10 veces sobre las cuales se obtuvo un promedio. Como por cada configuración se generaron 10 archivos, finalmente por cada configuración se obtuvo un promedio por sobre 100 repeticiones.

Para la medición de la memoria, se utilizó el comando shell script `/usr/bin/time`, el cual con su opción `-f` permite obtener el peak de memoria utilizado por cada proceso. Para evitar considerar la memoria utilizada por los propios datos en memoria principal, se midió con este mismo comando el peak de carga de los datos por separado para restarlo al resultado de la medición del proceso completo. Por cada archivo las operaciones se realizaron una vez. Como por cada configuracón se generaron 10 archivos, finalmente por cada configuración se obtuvo un promedio por sobre 10 repeticiones.

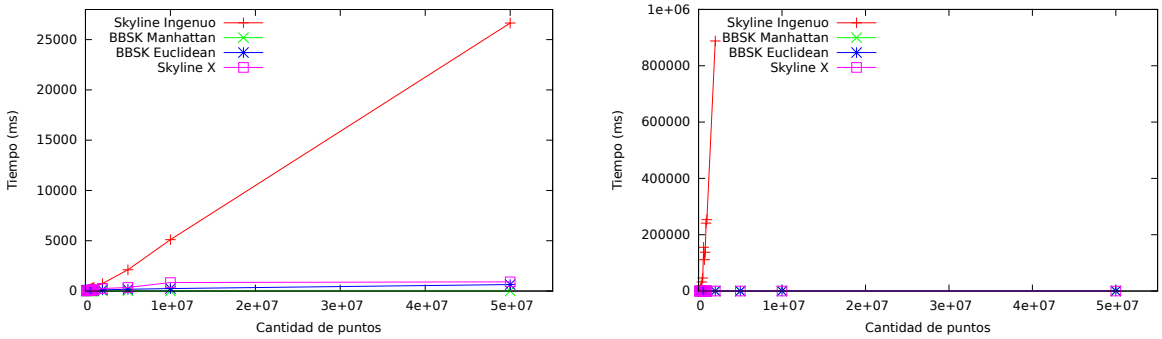
9.5. Resultados experimentales.

A continuación se presentan los resultados experimentales sobre los algoritmos que responden la consulta Skyline. Estos resultados muestran tanto el uso del tiempo como el uso de la memoria, como también el comportamiento de los algoritmos con respecto a la cantidad de puntos indexados como de la densidad. Además, se puede observar cómo la distribución de los datos afecta al comportamiento de los algoritmos.

9.5.1. Resultados experimentales: tiempo de ejecución.

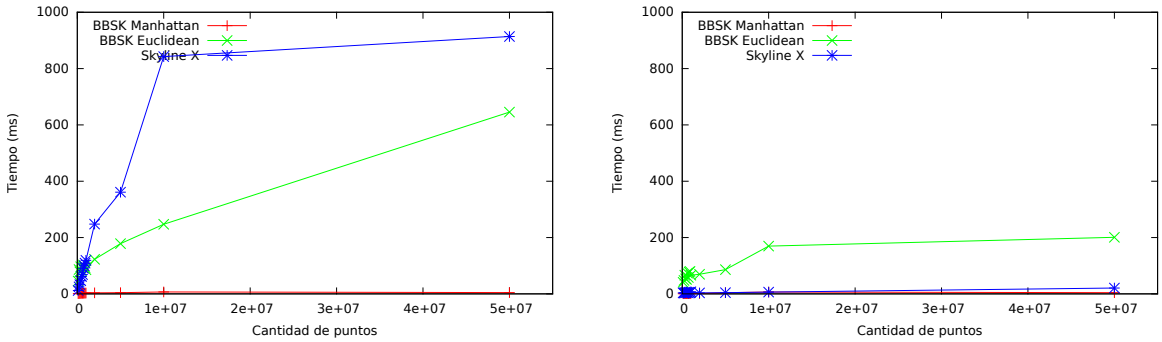
En la Figura 9.2(a) y la Figura 9.2(b) se presenta el tiempo utilizado por los algoritmos ejecutados en estas pruebas en función de la cantidad de puntos. Se puede observar que Skyline Ingenuo demora un tiempo muy alto en comparación con el resto, demorando hasta minutos en comparación con los milisegundos que utilizan los otros algoritmos. También se puede observar que Skyline Ingenuo se ve fuertemente desfavorecido frente a los datos clusterizados. Por un lado, para datos uniformes, el algoritmo demora 27 segundos aproximadamente en responder la consulta para 50 millones de datos. Por otro lado, para datos clusterizados, el algoritmo demora 15 minutos en responder la consulta con solo 1 millón de datos.

En la Figura 9.3(a) y la Figura 9.3(b) se puede apreciar el tiempo utilizado por los algoritmos en función de la cantidad de puntos. Se excluyó Skyline Ingenuo en este caso para observar mejor el comportamiento de los demás algoritmos. Lo primero que llama la atención es el poco tiempo



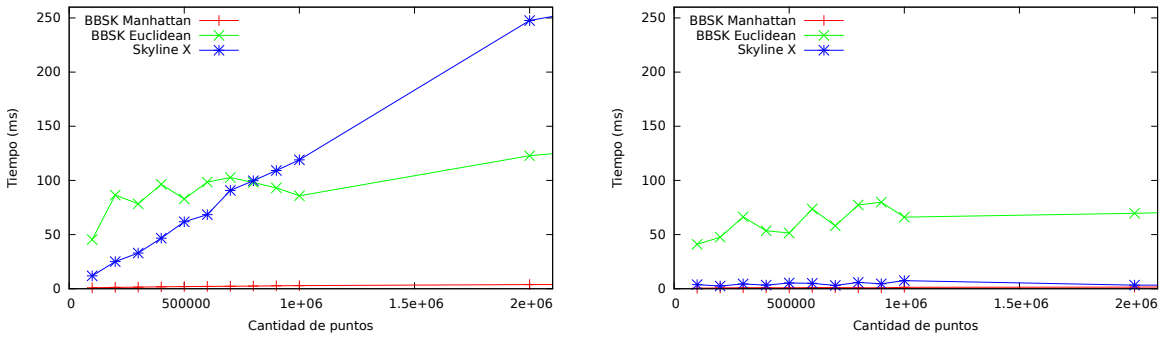
(a) Datos uniformes. (b) Datos clusterizados.

Figura 9.2: Tiempo utilizado por cantidad de puntos.



(a) Datos uniformes. (b) Datos clusterizados.

Figura 9.3: Tiempo utilizado por cantidad de puntos (excluyendo Skyline Ingenuo).



(a) Datos uniformes.

(b) Datos clusterizados.

Figura 9.4: Tiempo utilizado por cantidad de puntos (hasta 2 millones de puntos).

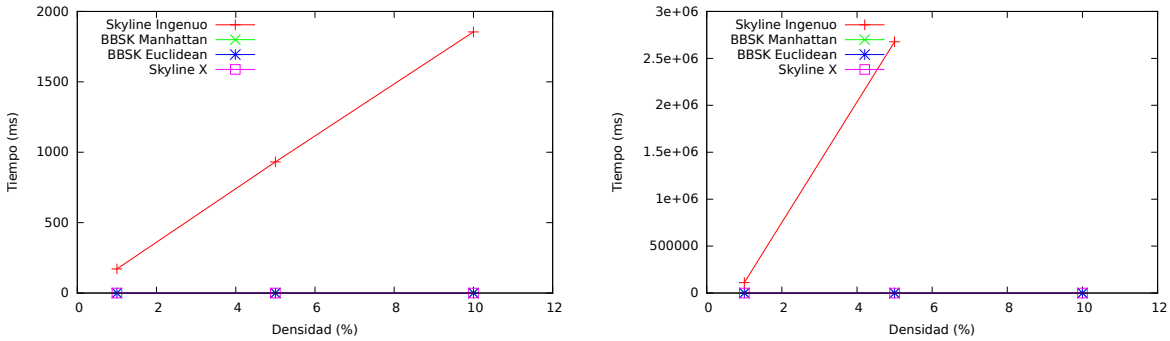
que demora BBSk con la distancia Manhattan. Independiente de la cantidad de puntos y de la distribución, este algoritmo no supera los 5 milisegundos del tiempo de respuesta.

Skyline X es el algoritmo que más se ve afectado por la distribución de los datos. Mientras que para datos uniformes puede llegar a demorar hasta los 900 milisegundos, para los datos clusterizados apenas alcanza a superar los 20 milisegundos. BBSk con la distancia Euclideana también se ve afectado por la distribución de los datos, pero en menor medida. Mientras que con datos uniformes el tiempo no supera los 700 milisegundos, para datos clusterizados llega a los 200 milisegundos, un 70 % menos.

En la Figura 9.4(a) y la Figura 9.4(b) se puede apreciar el tiempo utilizado por los algoritmos ejecutados en estas pruebas en función de la cantidad de puntos. A diferencia de lo anteriormente expuesto, se redujo la cantidad de puntos presentados. Se puede observar que para datos uniformes, *Skyline X* demora menos tiempo que BBSk con distancia Euclideana, pero a partir de los 800 mil puntos, este resultado varía, siendo BBSk el algoritmo que menos demora. De forma diferente funciona con los datos clusterizados, en donde *Skyline X* mentiene una amplia ventaja frente a BBSK con distancia Euclideana.

En la Figura 9.5(a) y la Figura 9.5(b) se puede apreciar el tiempo utilizado por los algoritmos ejecutados en estas pruebas en función de la densidad. Nuevamente Skyline Ingenuo demora demasiado y nuevamente la clusterización de los datos afecta negativamente al algoritmo. Mientras que para un 10 % de densidad con los datos uniformes el algoritmo demora 2 segundos, para un 5 % de densidad con los datos clusterizados, el algoritmo demora 45 minutos. Este último dato es importante, debido a que aunque el rango es más pequeño y la cantidad de puntos indexados es menor, demora tres veces más que para un rango 2000 veces más grande y 2 millones de puntos indexados (pero con una densidad mucho menor).

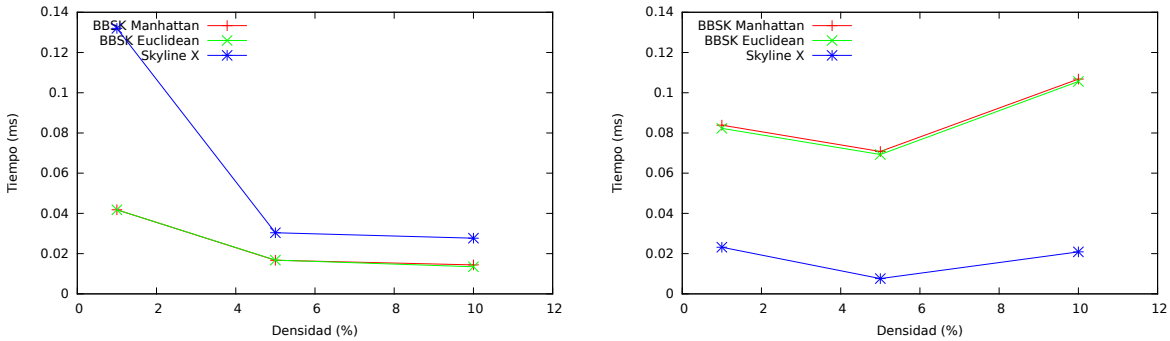
En la Figura 9.6(a) y la Figura 9.6(b) se puede observar el tiempo utilizado por los algoritmos ejecutados en estas pruebas en función de la densidad. Skyline Ingenuo no fue considerado en



(a) Datos uniformes.

(b) Datos clusterizados.

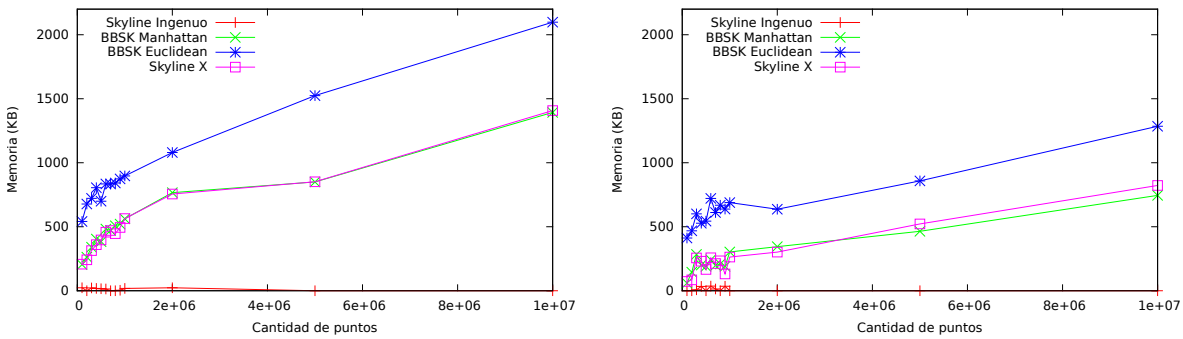
Figura 9.5: Tiempo utilizado por densidad.



(a) Datos uniformes.

(b) Datos clusterizados.

Figura 9.6: Tiempo utilizado por densidad (excluyendo Skyline Ingenuo).



(a) Datos uniformes.

(b) Datos clusterizados.

Figura 9.7: Memoria utilizada por cantidad de puntos.

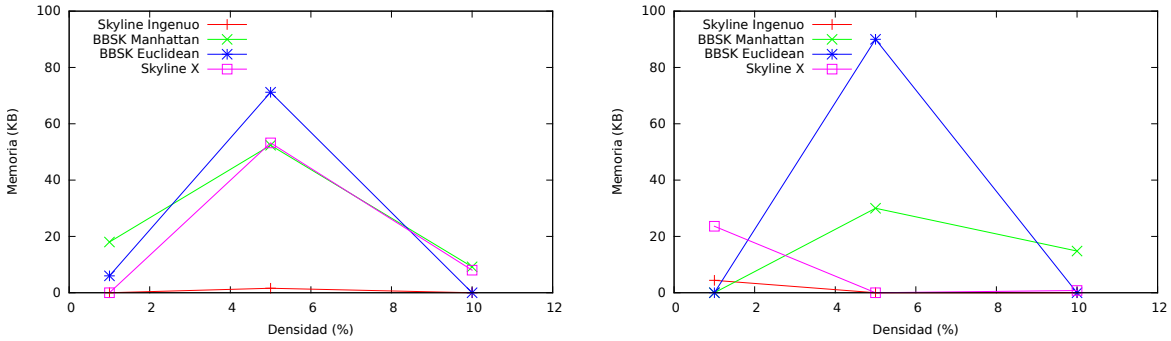
estas gráficas para poder observar con mayor detalle los resultados de los otros algoritmos. Se puede observar que el tiempo de respuesta es mucho menor en comparación con los resultados presentados en la Figura 9.3(a) y la Figura 9.3(b). El tamaño de la matriz utilizada en este tipo de experimentos, y por ende la altura del k^2-tree, puede ser la causa que explique el motivo del poco tiempo que demoran estos algoritmos.

9.5.2. Resultados experimentales: memoria requerida.

En la Figura 9.7(a) y la Figura 9.7(b) se puede observar la memoria utilizada por los algoritmos en función de la cantidad de puntos. Primero que todo, llama la atención el bajo consumo de memoria de Skyline Ingenuo. Esto ocurre porque dicho algoritmo no utiliza una estructura adicional, por lo tanto el uso de la memoria es bajo. Por otro lado, BBSk y *Skyline X* deben mantener el heap que ordena las submatrices en memoria principal, aumentando el uso de este recurso. A pesar de todo, los algoritmos que trabajan sobre el k^2-tree compensan el uso de memoria con la compactación de los datos frente a los datos descompactados con que trabaja el Skyline Ingenuo. Recordando que el espacio utilizado por los datos en memoria principal no ha sido medido en esta evaluación, se puede considerar como un punto clave a la hora de definir el uso total de memoria por cada algoritmo. Los datos descompactados que utiliza Skyline Ingenuo ocupan una cantidad de espacio mayor que los datos compactados más el heap de cada algoritmo.

De los algoritmos que utilizan los datos compactados en un k^2-tree, *Skyline X* y BBSk con la distancia Manhattan trabajan de forma igualmente eficiente ocupando menos memoria, mientras que BBSk con la distancia Euclideana utiliza un poco más de memoria. Recordando que la principal diferencia entre los tres algoritmos es la distancia utilizada para ordenar las submatrices en el heap, se puede concluir que la distancia Euclideana genera un costo que aumenta el uso de la memoria en el heap.

También se puede observar que todos los algoritmos se ven favorecidos frente a los datos clusterizados. Para los datos uniformes, el uso de memoria no supera 2 MB, mientras que para los datos clusterizados el uso de memoria máximo disminuye a 1,4 MB. Este es un interesante resultado tomando en cuenta que para utilizar esa cantidad de memoria se deben procesar 50



(a) Datos uniformes.

(b) Datos clusterizados.

Figura 9.8: Memoria utilizada por densidad.

millones de puntos.

En la Figura 9.8(a) y la Figura 9.8(b) se puede observar la memoria utilizada por los algoritmos en función de la densidad. Al igual que en la medición temporal, la memoria utilizada es muchisimo menor en comparación con lo presentado en la Figura 9.7(a) y la Figura 9.7(b). Se puede observar en todo momento que Skyline Ingenuo permanece con un bajo uso de memoria, mientras que el resto presenta un mayor uso de memoria con una densidad de un 5 % que con una densidad de un 10 %. Se puede observar que *Skyline X* se comporta de una mejor forma al aumentar la densidad sobre datos clusterizados. Esto es una buena señal para *Skyline X*, que aparentemente se ve favorecido frente a una alta densidad.

9.6. Conclusiones.

En este capítulo se presentó el proceso y los resultados de los experimentos sobre los algoritmos que responden la consulta Skyline. Se experimentó con un algoritmo Skyline Ingenuo sobre datos sin indexar en un k^2-tree, y con los dos algoritmos presentados en esta tesis: BBSk y *Skyline X*. BBSk fue evaluado utilizando dos distancias: la distancia Manhattan (o la suma de las coordenadas), y la distancia Euclideana (o la suma de los cuadrados de las coordenadas).

Con respecto a los resultados del tiempo de respuesta, Skyline Ingenuo es el algoritmo que más demora, llegando a tardar minutos, mientras que los demás algoritmos no demoran más de un segundo. Por el lado opuesto, BBSk con la distancia Manhattan resultó ser el algoritmo más eficiente en cuanto al uso del tiempo, demorando una pequeña porción de tiempo en comparación con *Skyline X* y que incluso el mismo BBSk con la distancia Euclideana.

La clusterización de los datos también influye de manera significativa en cuanto al tiempo que demoran los algoritmos. Skyline Ingenuo es el más afectado, aumentando su tiempo de respuesta desde aproximadamente 1 segundo hasta 45 minutos. El resto de los algoritmos se ven favorecidos frente a este tipo de distribución, siendo *Skylne X* el que presenta la mayor variación.

En cuanto al uso de memoria, Skyline Ingenuo es el que utiliza una menor cantidad de memo-

ria, mientras que BBSk con la distancia Euclideana es el que utiliza mayor cantidad de memoria. *Skyline X* y BBSk con la distancia de Manhattan utilizan una cantidad de memoria muy similar entre si. La clusterización también favorece a todos los algoritmos, reduciendo la cantidad de memoria utilizada de forma significativa.

Recordando lo presentado en el análisis teórico, un factor clave en la eficiencia de los algoritmos presentados radica en la cantidad de puntos Skyline presentes en el conjunto de datos. Por otro lado, como ya se ha explicado, los datos clusterizados presentan una cantidad de puntos Skyline menor en comparación con la distribución uniforme para la misma cantidad de punto. Esto se ve reflejado en la ventaja que obtienen todos los algoritmos en cuanto al uso de recursos al trabajar sobre datos clusterizados.

Aunque el algoritmo ingenuo utiliza una cantidad mínima de memoria, el tiempo que demora para responder la consulta Skyline es altísimo, y se vuelve imposible de utilizar para un conjunto de datos muy grande. Por otra parte, los algoritmos que trabajan sobre datos indexados en un k^2-tree, a pesar de utilizar una cantidad mayor de memoria, se ve compensado por un tiempo de demora mucho menor. Todo sin considerar que Skyline Ingenuo trabaja sobre datos sin compactar, perdiendo su ventaja en cuanto al uso de memoria frente a los demás algoritmos que trabajan sobre datos compactados.

BBSk y *Skyline X* funcionan de forma similar, con la única diferencia en cuanto a la distancia utilizada para ordenar los elementos del heap. En este sentido, en general BBSk con la distancia Manhattan es más eficiente de los tres, debido a su bajo costo temporal y su buen rendimiento en cuanto al uso de memoria. A pesar de todo, *Skyline X* presenta resultados prometedores en cuanto al uso de memoria y tiempo frente a datos clusterizados. En tercer lugar queda BBSk con la distancia Euclidena. El costo de elevar al cuadrado las coordenadas se ve reflejado tanto en el costo espacial como el temporal. El uso de la distancia correcta para operar con BBSk ha demostrado ser un punto clave a la hora de responder de forma eficiente la consulta Skyline.

Finalmente, BBSk con la distancia Manhattan es el algoritmo más eficiente, puesto que utiliza una baja cantidad de memoria respondiendo la consulta en un tiempo más que satisfactorio.

Capítulo 10

Compact Count.

En este capítulo se revisará el proceso de experimentación para medir la eficiencia de *Compact Count*. El objetivo es medir el tiempo y la memoria utilizada en su ejecución y analizar los resultados obtenidos.

En la Sección 10.1 se presenta el proceso de implementación de los algoritmos que serán evaluados. En la Sección 10.2 se presentan los datos utilizados en la experimentanción, tanto su formato, características, configuraciones y transformaciones. En la Sección 10.3 se presentan los algoritmos que serán evaluados. En la Sección 10.4 se presenta el entorno en donde se realizaron las pruebas. En la Sección 10.5 se presentan los resultados experimentales. Finalmente, en la Sección 10.6 se presentan algunas conclusiones de los resultados encontrados.

10.1. Implementación.

La librería utilizada que tiene la implementación del k^2-tree es la misma que la utilizada en la Sección 9.1. Cabe destacar que se debió modificar la estructura que implementa los bitmaps T y L pueda calcular la consulta *rank* sobre el bitmap L en un tiempo constante. Lo anterior es debido a que es necesario para la implementación de *Compact Count*.

La misma librería presenta la implementación de *Range*, algoritmo presentado en [27] y que rescata todos aquellos puntos pertenecientes a un rango de consulta. De este resultado solo se consideró la cantidad de puntos, puesto que es el resultado que también entrega *Compact Count*.

La implementació de *Compact Count* fue realizada usando el lenguaje de programación C++, con condiciones similares a las presentadas en la Sección 9.1.

10.2. Datos experimentales.

Fueron utilizados los mismos datos presentados en la Sección 9.2. Para la generación de los rangos de consultas, en las mediciones del tiempo de respuesta se generaron rangos aleatorios por cada archivo de pruebas. En las pruebas de uso de memoria se utilizó un rango de consulta fijo. Para los datos que miden el uso de memoria en función de la cantidad de puntos, el rango de consulta fue de $[1000000, 9000000] \times [1000000, 9000000]$. Para los datos que miden el uso de

memoria en función de la densidad, el rango de consulta fue de $[1000, 4000] \times [1000, 4000]$.

10.3. Línea de comparación (Baseline).

En esta experimentación, se realizará una comparación de los siguientes algoritmos:

- `Range`: Es el algoritmo presentado en [27], y que rescata todos los puntos al interior del rango de consulta.
- `Compact Count`: Es la propuesta presentada en esta tesis en el Capítulo 6.

Cabe destacar que ambos algoritmos utilizan datos ya indexados en un k^2-tree, por lo tanto el proceso de compresión y/o descompresión de los datos queda fuera del análisis de la experimentación.

10.4. Entorno de pruebas.

El entorno de pruebas es exactamente el mismo que el presentado en la Sección 9.4 utilizado para realizar los experimentos sobre la consulta Skyline. Se consideró el mismo servidor con las mismas características, con el mismo Sistema Operativo, las mismas instrucciones de medición y las mismas repeticiones sobre cada operación.

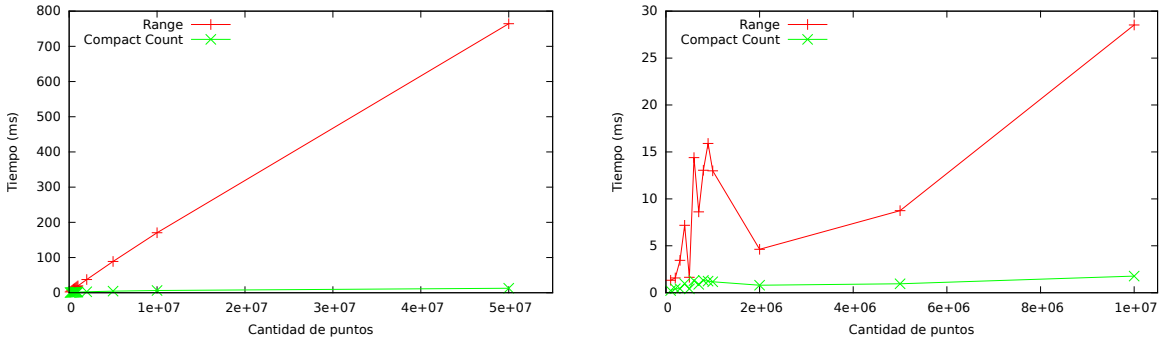
10.5. Resultados experimentales.

A continuación se presentan los resultados experimentales sobre los algoritmos que responden con la cantidad de puntos encontrados al interior de un rango de consulta. Estos resultados muestran tanto el uso del tiempo como el uso de la memoria, como también el comportamiento de los algoritmos con respecto a la cantidad de puntos indexados como de la densidad. Además se puede observar cómo la distribución de los datos afecta al comportamiento de los algoritmos.

10.5.1. Resultados experimentales: tiempo de ejecución.

En la Figura 10.1(a) y la Figura 10.1(b) se presenta el tiempo utilizado por los algoritmos ejecutados en estas pruebas en función de la cantidad de puntos. Lo primero que se puede observar que sin importar la distribución de los datos, *Compact Count* es el algoritmo más eficiente, demorando no más de 10 milisegundos para 50 millones de datos.

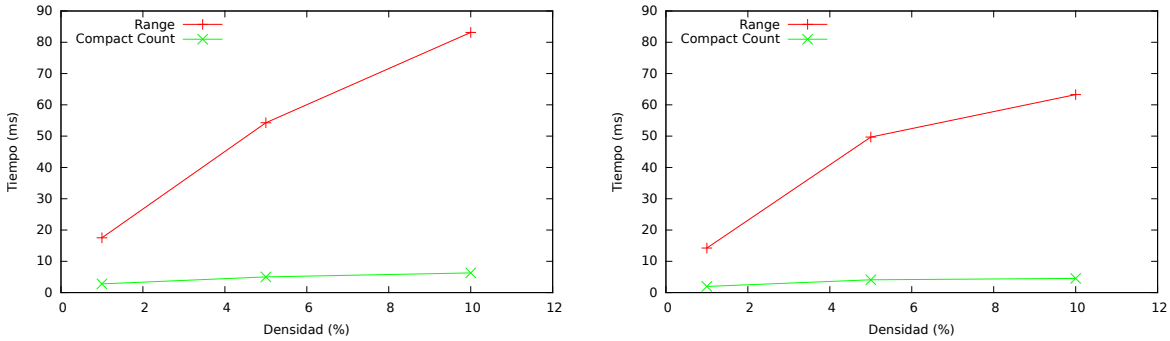
También se puede observar que *Range* se beneficia de la clusterización de los datos. Mientras que para datos uniformes, *Range* demora poco menos de 200 milisegundos para 10 millones de datos indexados, para datos clusterizados demora un 85 % menos. También se logra destacar que para los primeros 2 millones de puntos, el tiempo de ejecución sobre datos clusterizados es más inestable para *Range* que para datos uniformes. Esto debido a que la densidad de datos presente en la distribución uniforme es la misma en cualquier lugar dentro del espacio, mientras que para datos clusterizados no. Si un rango de consulta presenta pocos datos, *Range* demorará mucho



(a) Datos uniformes.

(b) Datos clusterizados.

Figura 10.1: Tiempo utilizado por cantidad de puntos.



(a) Datos uniformes.

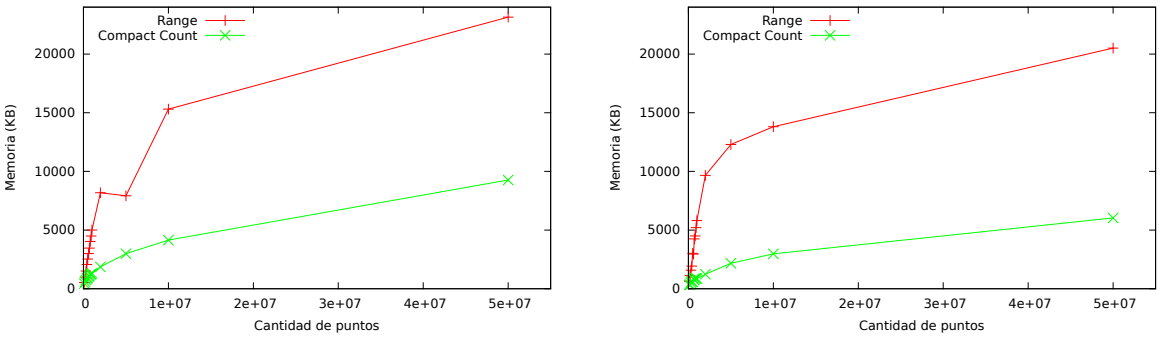
(b) Datos clusterizados.

Figura 10.2: Tiempo utilizado por densidad.

menos que si el rango tiene una alta cantidad de puntos. En el caso de los datos uniformes, el rango va a tener la misma cantidad de puntos independiente del lugar en donde se encuentre, en cambio para los datos clusterizados, esto no se cumple.

Compact Count no se ve afectado por este problema, debido a que su eficiencia radica en la cantidad de Cuadrantes Maximales y no en la cantidad de puntos al interior del rango. Es por ello que independiente del lugar en donde se encuentre el rango de consulta al interior del un conjunto con distribución clusterizada, *Compact Count* demorará lo mismo.

En la Figura 10.2(a) y la Figura 10.2(b) se pueden observar el tiempo utilizado por los algoritmos ejecutados en estas pruebas en función de la densidad. Se puede apreciar que nuevamente *Compact Count* obtiene una amplia ventaja, demorando menos de 10 milisegundos frente a los entre 70 y 90 que demora *Range*. Al igual que en el caso anterior, la clusterización de los datos afecta a *Range*, demorando aproximadamente un 25 % menos sobre datos clusterizados. Al igual que en el la medición del tiempo de respuesta con respecto a la cantidad de puntos, *Range* uti-



(a) Datos uniformes.

(b) Datos clusterizados.

Figura 10.3: Memoria utilizada por cantidad de puntos.

liza un mayor tiempo en la medida en que aumenta la densidad, mientras que *Compact Count* mantiene un tiempo constante. Esto demuestra que *Range* es suceptible a la cantidad de puntos indexados al interior del rango, y *Compact Count* no.

10.5.2. Resultados experimentales: memoria requerida.

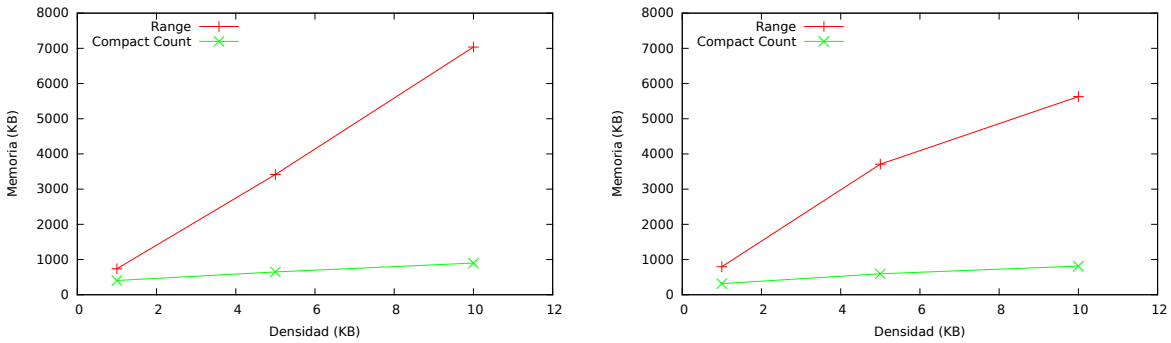
En la Figura 10.3(a) y la Figura 10.3(b) se puede apreciar la memoria utilizada por los algoritmos ejecutados en estas pruebas en función de la cantidad de puntos. Se puede apreciar que en todos los casos, la memoria utilizada aumenta sostenidamente en la medida en que aumenta la cantidad de puntos. Al igual que en el tiempo de respuesta, *Compact Count* utiliza mucho menos memoria que *Range*, con alrededor de un 50 % menos. A pesar de esta importante diferencia, el uso de memoria es mayor que el utilizado en los algoritmos que responden la consulta Skyline, los cuales no superaban los 3 MB.

La clusterización también afecta a ambos algoritmos de forma positiva, siendo *Compact Count* el más favorecido, con una reducción de un 35 % frente a un 10 % de reducción del uso de memoria por parte de *Range*.

En la Figura 10.4(a) y la Figura 10.4(b) se puede apreciar la memoria utilizada por los algoritmos ejecutados en estas pruebas en función de la densidad. De forma similar a los casos anteriores, *Compact Count* es bastante más eficiente que *Range*, mientras que la clusterización de los datos disminuye la cantidad de recursos utilizados.

10.6. Conclusiones.

En este capítulo se expermentó con dos algoritmos que permiten responder la cantidad de puntos al interior de un rango: *Range*, que no solo cuenta los puntos sino que identifica sus coordenadas, y *Compact Count*, que identifica los Cuadrantes Maximales que conforman el rango y luego cuenta los puntos por cada Cuadrante.



(a) Datos uniformes.

(b) Datos clusterizados.

Figura 10.4: Memoria utilizada por densidad.

Como principal resultado se tiene que *Compact Count* es más eficiente que *Range*, tanto en el tiempo de respuesta como en la memoria utilizada. La diferencia en cuanto al tiempo de respuesta comparado con *Range* es mucho mayor que la diferencia con respecto a la memoria utilizada.

La clusterización de los datos afecta a *Range*, reduciendo su tiempo de respuesta en un 85 % y el uso de memoria en un 10 %. *Compact Count* por su parte se ve afectado en menor medida, en donde solo se ve reducido el uso de memoria en un 35 %. Este cambio es menor en comparación con *Range* debido a que de por si, el algoritmo utiliza una baja cantidad de recursos.

Finalmente, los resultados experimentales demuestran que en caso de solo requerir la cantidad de puntos al interior de un rango y no cuáles son estos puntos, *Compact Count* es una alternativa eficiente.

Capítulo 11

Enumerating Compact.

En este capítulo se revisará el proceso de experimentación para medir la eficiencia de aquellos algoritmos que responde la consulta *Enumerating Skyline*. El objetivo es medir el tiempo y la memoria utilizada en su ejecución y analizar los resultados obtenidos.

En la Sección 11.1 se presenta el proceso de implementación de los algoritmos que serán evaluados. En la Sección 11.2 se presentan los datos utilizados en la experimentanción y el entorno en donde se realizaron las pruebas. En la Sección 11.3 se presentan los algoritmos que serán evaluados en esta experimentación. En la Sección 11.4 se presentan los resultados experimentales tras realizar las pruebas. Finalmente, en la Sección 11.5 se presentan algunas conclusiones de los resultados encontrados.

11.1. Implementación.

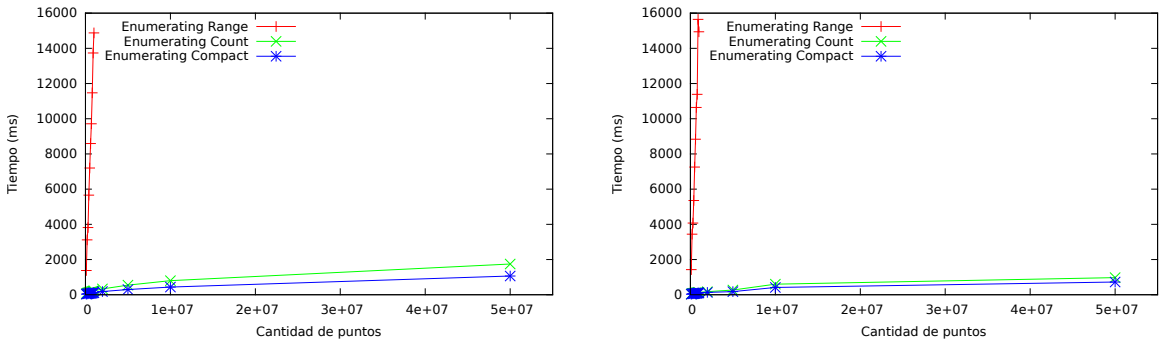
La librería utilizada que contiene la implementación del k^2-tree es la misma que la presentada en la Sección 9.1 y la Sección 10.1. De forma similar a lo presentado en la Sección 10.1, se debió intervenir en dicha librería para permitir calcular la consulta *Rank* sobre el bitmap L.

La implementación de los algoritmos fue desarrollada de forma similar a lo expuesto en la Sección 9.1 y la Sección 10.1. Los algoritmos utilizados en las pruebas utilizan de manera frecuente el algoritmo *Compact Count*, que según los resultados presentados en el Capítulo 10, ha demostrado ser muy eficiente en cuanto al uso de recursos.

11.2. Datos experimentales y entorno de pruebas.

Los datos experimentales que fueron usados son los mismos archivos usados en los experimentos de los capítulos anteriores, y que fueron presentados en la Sección 9.2.

El entorno de pruebas es el mismo que el utilizado en los experimentos anteriores. Se trabajó en las mismas condiciones, usando el mismo servidor con las mismas características, con el mismo Sistema Operativo, las mismas instrucciones de medición y las mismas repeticiones sobre cada operación.



(a) Datos uniformes.

(b) Datos clusterizados.

Figura 11.1: Tiempo utilizado por cantidad de puntos.

11.3. Línea de comparación (Baseline).

En esta experimentación, se realizará una comparación de los siguientes algoritmos:

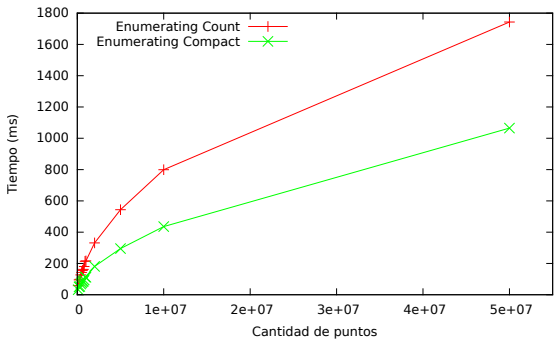
- `Enumerating Range`: Es similar al Algoritmo 7.2.1, con la diferencia que a la hora de contar la cantidad de puntos dominados por cada punto Skyline, se utiliza *Range* en vez de *Compact Count*.
- `Enumerating Count`: Es el mismo algoritmo presentado en el Algoritmo 7.2.1.
- `Enumerating Compact`: Es la propuesta de esta tesis para responder esta variante del Skyline y que fue descrito en la Sección 7.2.

11.4. Resultados experimentales.

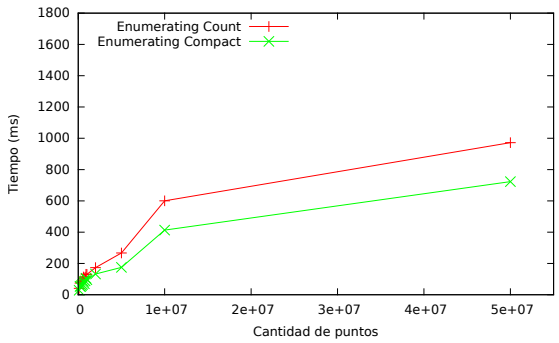
A continuación se presentan los resultados experimentales sobre los algoritmos que responden la consulta *Enumerating Skyline*. Estos resultados muestran tanto el uso del tiempo como el uso de la memoria, como también el comportamiento de los algoritmos con respecto a la cantidad de puntos indexados como de la densidad. Además se puede observar cómo la distribución de los datos afecta al comportamiento de los algoritmos.

11.4.1. Resultados experimentales: tiempo de ejecución.

En la Figura 11.1(a) y la Figura 11.1(b) se puede observar el tiempo utilizado por los algoritmos ejecutados en estas pruebas en función de la cantidad de puntos. Se puede observar que *Enumerating Range* aumenta su tiempo de respuesta rápidamente, llegando a los 14 segundos de duración para responder la consulta sobre 1 millón de puntos, frente a los otros algoritmos que llegan apenas a los 200 milisegundos para la misma cantidad de datos.

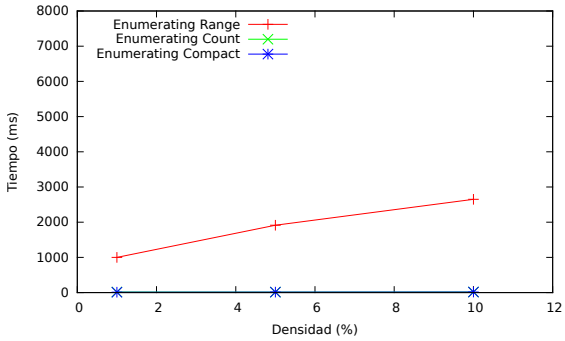


(a) Datos uniformes.

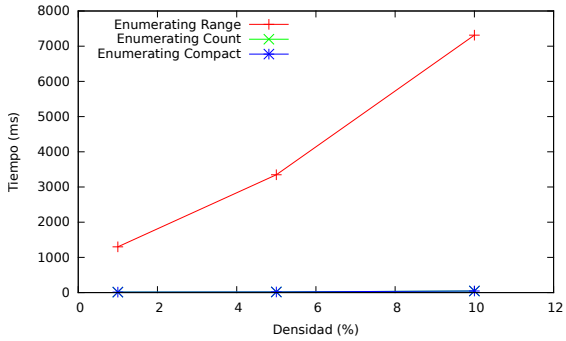


(b) Datos clusterizados.

Figura 11.2: Tiempo utilizado por cantidad de puntos (excluyendo Enumerating Range).



(a) Datos uniformes.



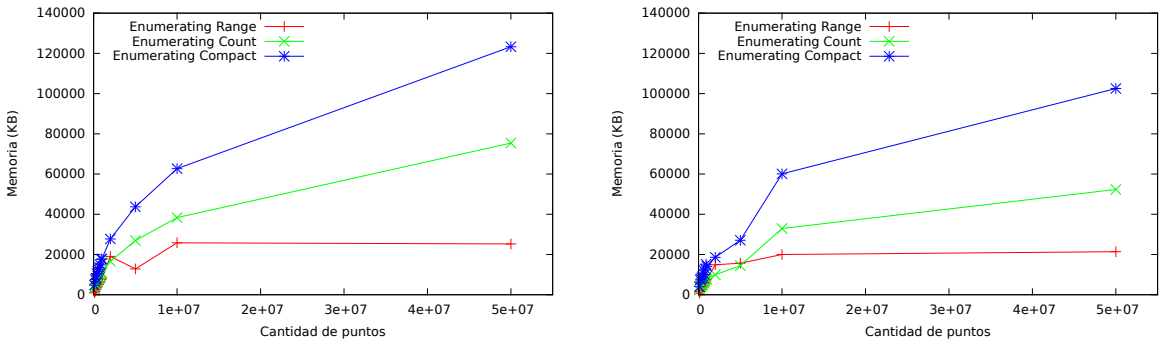
(b) Datos clusterizados.

Figura 11.3: Tiempo utilizado por densidad.

En la Figura 11.2(a) y la Figura 11.2(b) se puede observar el tiempo utilizado por *Enumerating Compact* y *Enumerating Count* en función de la cantidad de puntos. *Enumerating Range* no fue considerado para poder observar el comportamiento de los otros dos algoritmos con más detalle. Se puede apreciar que *Enumerating Compact* es más eficiente *Enumerating Count*, debido a que su tiempo de respuesta es menor. Esto demuestra que el uso del arreglo N en *Enumerating Compact* para reducir la cantidad de Cuadrantes Maximales contados a lo largo de proceso permite reducir significativamente el tiempo de respuesta. La reducción es de un 40 % para datos uniformes y de un 25 % para datos clusterizados.

También se puede observar que la clusterización de los datos favorece a ambos algoritmos. Mientras *Enumerating Count* reduce su tiempo de respuesta en un 45 %, *Enumerating Compact* lo reduce en un 32 %. Por lo tanto, *Enumerating Compact* se ve afectado en menor medida por la clusterización de los datos en comparación con *Enumerating Count*.

En la Figura 11.3(a) y la Figura 11.3(b) se puede observar el tiempo utilizado por los algorit-



(a) Datos uniformes.

(b) Datos clusterizados.

Figura 11.4: Memoria utilizada por cantidad de puntos.

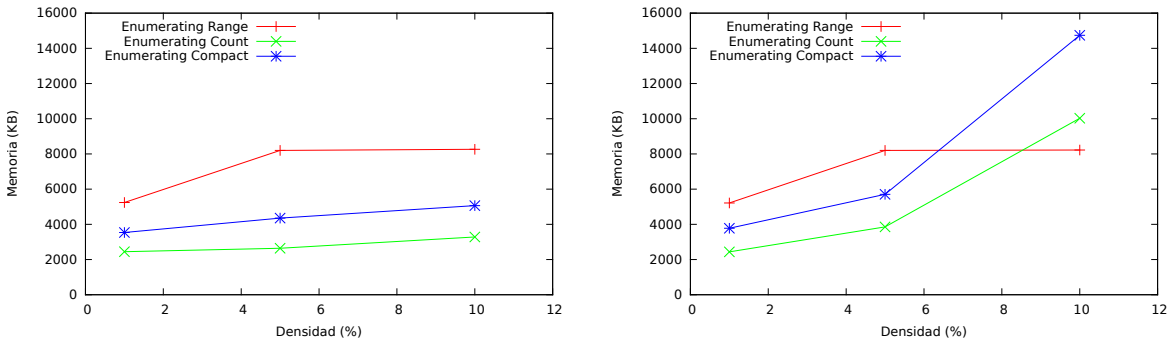
mos ejecutados en estas pruebas en función de la densidad. Se puede observar que *Enumerating Range* es el algoritmo que más tiempo demora en responder la consulta, con hasta 3 segundos para datos uniformes y 8 para datos clusterizados. Los otros dos algoritmos demoran bastante menos, sin siquiera llegar a los 50 milisegundos. Esto demuestra que la aplicación de *Compact Count* ayuda a responder la variante *Enumerating Skyline* eficientemente, reduciendo en gran medida el tiempo de respuesta.

11.4.2. Resultados experimentales: memoria requerida.

En la Figura 11.4(a) y la Figura 11.4(b) se puede apreciar la memoria utilizada por los algoritmos ejecutados en estas pruebas en función de la cantidad de puntos. A diferencia de los casos anteriores, *Enumerating Range* es el algoritmo que menos memoria utiliza, seguido de *Enumerating Count* y finalmente *Enumerating Compact*. El uso del heap sumado al arreglo N por un lado mejora enormemente el tiempo utilizado, a cambio de un importante costo en cuanto al uso de memoria. La clusterización también favorece a los tres algoritmos, pero en menor medida en comparación con la reducción del tiempo de respuesta. De los tres, *Enumerating Count* es aquel que reduce en mayor medida la memoria utilizada usando datos clusterizados, con un 30 % de reducción, frente a un 16 % de reducción de los otros dos algoritmos.

También se debe destacar que el uso de memoria es elevado en comparación con los algoritmos presentados en el capítulo anterior, los cuales no superaban los 24 MB de memoria. En este caso, la memoria usada por los algoritmos puede ascender a los 124 MB.

En la Figura 11.5(a) y la Figura 11.5(b) se puede apreciar la memoria utilizada por los algoritmos ejecutados en estas pruebas en función de la cantidad de puntos. Se puede observar que *Enumerating Range* no sufre variación con respecto a la distribución de los datos, mientras se estabiliza al aumentar la densidad. Por otro lado, *Enumerating Count* y *Enumerating Compact* aumentan el uso de memoria de forma significativa al aumentar la densidad frente a datos clusterizados, superando incluso a *Enumerating Range*. A pesar de ello, el uso de la memoria es menor que la encontrada en la Figura 11.4(a) y la Figura 11.4(b), debido al tamaño de la matriz



(a) Datos uniformes.

(b) Datos clusterizados.

Figura 11.5: Memoria utilizada por densidad.

utilizada en este caso en comparación a los experimentos realizados en función de la cantidad de puntos.

11.5. Conclusiones.

En este capítulo se presentaron los resultados experimentales de los algoritmos que responde la variante *Enumerating Skyline* sobre datos indexados en un k^2-tree. Esta consulta permite encontrar aquellos puntos que son Skyline, y por cada uno de ellos entregar la cantidad de puntos que dominan.

De entre los tres algoritmos evaluados, *Enumerating Range* aumenta rápidamente el tiempo utilizado en la medida en que aumentan la cantidad de puntos indexados, mientras que *Enumerating Count* y *Enumerating Compact* demoran menos de un segundo para una alta cantidad de datos, siendo la estrategia de *Enumerating Compact* la que menos tiempo utiliza. Por otro lado, en lo que respecta el uso de memoria, *Enumerating Compact* es el que más memoria utiliza, seguido de *Enumerating Count*. En este sentido, *Enumerating Range* es el algoritmo que menos memoria utiliza. Por otro lado, la clusterización si afecta de forma positiva a todos los algoritmos, reduciendo los recursos utilizados.

Aquellos algoritmos que utilizan *Compact Count* en su funcionamiento, reducen significativamente el tiempo utilizado para responder con la consulta, pero por otro lado se ven perjudicado en el uso de la memoria. El uso del arreglo N para evitar contar dos veces los mismos Cuadrantes Maximales si marca la diferencia en el tiempo utilizado, pero también genera un costo adicional en la memoria.

Finalmente, *Enumerating Compact* y *Enumerating Count* son los algoritmos que han tenido un mejor comportamiento en cuanto al uso de recursos. *Enumerating Range* presenta un tiempo de respuesta demasiado alto en comparación con los otros dos algoritmos, tiempo que no es compensado por el bajo costo en cuanto a memoria. De los otros dos algoritmos, *Enumerating Compact* utiliza una menor cantidad de tiempo a cambio de un uso mayor de memoria, mientras que *Enumerating Count* demora un poco más a cambio de un uso menor de memoria. Dependiendo

de los requisitos y del tipo de recurso más importante, se puede escoger entre uno u otro algoritmo.

Capítulo 12

Conclusiones de los resultados experimentales.

En esta parte se presentaron los resultados experimentales de los algoritmos propuestos en esta tesis y que consideran datos indexados en un k^2-tree. Se experimentó con *BBSk^2-tree* (BBSk) y *Skyline X* para responder la consulta Skyline, con *Compact Count* para contar los puntos al interior de un rango de consulta, y *Enumerating Compact* para responder la variante *Enumerating Skyline*. Se experimentó en función de la cantidad de puntos y la densidad, y con dos tipos de distribuciones: uniforme y clusterizada.

En lo que respecta a la consulta Skyline, BBSk resultó ser el algoritmo más eficiente, siempre y cuando se utilizara la distancia Manhattan para ordenar las submatrices al interior del heap. Es un algoritmo que no superó los 5 milisegundos en responder la consulta Skyline para una cantidad de 50 millones de datos independiente de la distribución de los mismos. Por otro lado, es el segundo algoritmo que menos memoria ocupa de los probados (Skyline Ingenuo es el que ocupa menos, pero el tiempo que utiliza es demasiado alto). También se demostró que al usar otro tipo de distancia, como es el caso de la distancia Euclideana, el comportamiento de BBSk es menos eficiente en comparación con el uso de la distancia Manhattan. La elección de la distancia a utilizar es un factor clave a la hora de mantener la eficiencia del algoritmo para responder la consulta Skyline.

Por otro lado, *Compact Count* resultó ser un algoritmo muy eficiente a la hora de contar los puntos al interior de un rango, obteniendo una importante ventaja frente a *Range*. Como *Compact Count* no depende de la cantidad de puntos indexados en el rango, sino en la cantidad de Cuadrantes Maximales que conforman dicho rango, la ventaja que logra conseguir frente a *Range* es mayor en la medida en que el rango de consulta contenga más puntos.

En lo que respecta a la consulta *Enumerating Skyline*, *Enumerating Compact* demostró ser una buena estrategia para reducir el tiempo de ejecución de la consulta, a cambio de un costo elevado de memoria. Como una forma de reducir la memoria utilizada a cambio de un aumento insignificante del tiempo de respuesta se encuentra *Enumerating Count*.

El k^2-tree es una estructura que puede compactar de forma más eficiente aquellos datos que presentan una distribución clusterizada. La ventaja que obtiene esta estructura la comparten los

algoritmos que participaron en la experimentación, ya que todos los algoritmos (a excepción de Skyline Ingenuo) han logrado reducir o mantener el uso de recursos al utilizar datos clusterizados en comparación con datos uniformes.

En la gran mayoría de los casos, los resultados del uso de recursos en función de la densidad muestra un uso menor de los recursos en comparación de los resultados del uso de recursos en función de la cantidad de puntos. La principal causa radica en la altura del k^2-tree, el cual es un factor clave de la eficiencia de todos los algoritmos presentados, y que está directamente relacionado con el tamaño de la matriz. Para los experimentos en función de la densidad se utilizó una matriz de un tamaño menor que los experimentos en función de la cantidad de puntos.

En general, todos los algoritmos presentados en esta tesis obtuvieron resultados más que satisfactorios tras los experimentos realizados. El tiempo de respuesta ha sido reducido de forma considerable en comparación con lo expuesto en la literatura, aunque por otro lado el uso de la memoria principal es un *tradeoff* que se debe tener en cuenta a la hora de aplicarlos. En este sentido, de los algoritmos presentados en esta tesis, BBSk es el que utiliza la menor cantidad de memoria, con hasta 1.3 MB. *Compact Count* aumenta la memoria utilizada, llegando a requerir hasta 8 MB. *Enumerating Count* por su parte ocupa más memoria que *Compact Count*, con hasta 80 MB. Este es un costo ínfimo al lado de las ventajas obtenidas gracias al uso de estos algoritmos, como el tiempo de respuesta y la enorme cantidad de puntos que deben ser procesados para llegar a utilizar esta cantidad de memoria.

Parte V

Conclusiones y Trabajo Futuro.

Capítulo 13

Conclusiones y Trabajo Futuro.

13.1. Conclusiones.

En esta tesis se pudo comprobar que el k^2-tree es una Estructura de Datos Compactas adecuada para evaluar la consulta Skyline y sus variantes, por lo tanto se da por verdadera la hipótesis planteada. Los resultados de esta tesis permiten expandir las capacidades del k^2-tree, permitiendo responder nuevas consultas para datos que se encuentren indexados en esta estructura.

Se diseñaron, implementaron, analizaron y evaluaron cuatro algoritmos que responden tanto la consulta Skyline como sus variantes sobre datos representados en un k^2-tree: una Estructura de Datos Compactas diseñada para representar relaciones binarias de forma compacta.

Gracias a esto, se han cumplido todos los objetivos planteados en el Capítulo 1:

- Se logró adaptar la mejor estrategia encontrada en la literatura para conseguir responder la consulta Skyline sobre datos indexados en un k^2-tree: *Branch and Bound Skyline Algorithm* (BBS) presentado en [36, 37].
- Se diseñaron dos algoritmos que mejoraron la estrategia de BBS frente a datos indexados en un k^2-tree: *BBSk^2-tree* (BBSk) y *Skyline X*.
- Se evaluó, a través de pruebas experimentales el desempeño de ambos algoritmos tanto en rendimiento como en almacenamiento.
- Se exploraron algunas de las variantes de la consulta Skyline y se adaptaron sus principales algoritmos para trabajar sobre un k^2-tree. De entre las variantes se exploraron *Enumerating Skyline* y *Constrained Skyline Query*.

Los dos algoritmos propuestos para responder la consulta Skyline aplicaron una nueva propiedad de poda, permitiendo mejorar la eficiencia de BBS. Esta propiedad consiste en la poda submatrices hermanas dominadas entre si, propiedad aplicada gracias a la estructura inherente que presenta el k^2-tree. De forma teórica se pudo comprobar que BBSk presenta un costo temporal de orden $O(k|S|^2 \log_k n)$ y un costo espacial de orden $O(k|S| \log_k n)$, en donde $|S|$ es la cantidad de puntos Skyline encontrados y n es el tamaño de la matriz. Por otro lado, *Skyline X*

presenta un costo tanto temporal como espacial de orden $O(k|S|\log_k n)$. De forma experimental, se logró demostrar que BBSk utilizando la distancia de Manhattan es el algoritmo más eficiente en la práctica.

El tercer algoritmo es *Compact Count*, un algoritmo que permite contar la cantidad de puntos al interior de un rango de consulta para datos indexados en un k^2-tree, y que posteriormente fue utilizado en el último algoritmo. Este algoritmo identifica los Cuadrantes Maximales que conforman el rango de consulta, y cuenta con rapidez los puntos de cada Cuadrante Maximal. Un Cuadrante Maximal es una submatriz del k^2-tree que está circunscrita al rango de consulta, pero su padre no lo está. Debido a la estructura del k^2-tree, se puede contar con rapidez los puntos al interior de un Cuadrante Maximal utilizando los bitmaps que conforman dicha estructura. Tras el análisis teórico, *Compact Count* presenta una complejidad algorítmica de $O(m\log_k n)$, en donde m es la cantidad de Cuadrantes Maximales que contiene el rango, y n es el tamaño de la matriz. Tras la evaluación experimental, se pudo demostrar que *Compact Count* trabaja de forma eficiente tanto en el tiempo utilizado como en el uso de memoria, frente a la propuesta de la literatura.

Tras la revisión de las variantes, se exploraron dos de ellas: *Enumerating Skyline* y *Constrained Skyline Query*. De la primera surgió *Enumerating Compact*: un algoritmo que utilizando BBSk y *Compact Count* permite responder *Enumerating Skyline* de manera eficiente. Con BBSk identifica los puntos que son Skyline, mientras que con *Compact Count* cuenta los puntos que domina cada punto Skyline. Para evitar contar los puntos al interior de un cuadrante dos veces, utiliza un arreglo N que almacena los resultados para ser consultados posteriormente en caso de ser necesario. Tras el análisis teórico se logró determinar que el costo temporal es de $O((k|S|^2 + m_t)\log_k n)$, mientras que el costo espacial es de $O(k|S|\log_k n) + m_{max}$, en donde $|S|$ es la cantidad de puntos Skyline encontrados, m_t la cantidad de Cuadrantes Maximales de la región de dominancia del Skyline, m_{max} corresponde a la cantidad de Cuadrantes Maximales que posee la región de dominancia del punto Skyline con más Cuadrantes Maximales y n es el tamaño de la matriz. Tras la evaluación experimental, *Enumerating Compact* resultó ser un algoritmo bastante competitivo, en conjunto con *Enumerating Count*: un algoritmo que aplica BBSk y *Compact Count* por separado. *Enumerating Compact* utiliza una menor cantidad de tiempo a cambio de un uso mayor de memoria, mientras que *Enumerating Count* demora un poco más a cambio de un uso menor de memoria.

En general el costo teórico de todos los algoritmos depende de la cantidad de puntos Skyline, cantidad de Cuadrantes Maximales y el tamaño de la matriz. Esto es una ventaja para los algoritmos presentados, puesto que no depende de otras variables que suelen tener un valor mayor, como por ejemplo la cantidad de puntos indexados o la cantidad de puntos al interior del rango de consulta o de la región de dominancia. En este sentido, la cantidad de puntos Skyline es el factor más importante que influye en el costo de los algoritmos.

Tras la evaluación experimental, se pudo comprobar que la clusterización de los datos favorece a todos los algoritmos presentados. Los datos clusterizados, que ya mejoraban la compactación de los datos en el k^2-tree, también reduce tanto el tiempo utilizado como la memoria requerida de todos los algoritmos propuestos en esta tesis en comparación con los datos con distribución uniforme. También el uso de recursos se vio reducida tras los experimentos en función de la densidad. El uso de una matriz más pequeña fue una influencia a la hora de conseguir estos

resultados.

Con todo lo anterior, se puede concluir que todos los objetivos planteados en el Capítulo 1 se han cumplido según lo planificado.

13.2. Trabajo futuro.

Tras el trabajo realizado durante esta tesis, se plantea como posible trabajo futuro lo siguiente:

- Explorar otras variantes de la consulta Skyline sobre datos indexados en un k^2-tree. Específicamente variantes como *K-Dominating query* o *K-Skyband Query*, las cuales son las más cercanas a *Enumerating Skyline*.
- Diseñar e implementar algoritmos que respondan la consulta Skyline sobre Estructura de Datos Compactas en más de dos dimensiones. Se puede trabajar sobre alguna estructura variante del k^2-tree que soporte relaciones en más de dos dimensiones o utilizar alguna otra Estructura de Datos Compactas.
- Diseñar e implementar algoritmos que respondan la consulta Skyline sobre otras Estructuras de Datos Compactas que representen relaciones binarias. Un ejemplo es el *Wavelet tree*.

Bibliografía

[1] Alrifai, M., Skoutas, D., Risse, T.: Selecting skyline services for qos-based web service composition. In: Proceedings of the 19th international conference on World Wide Web. pp. 11–20. ACM (2010)

[2] Atallah, M.J., Qi, Y.: Computing all skyline probabilities for uncertain data. In: Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. pp. 279–287. ACM (2009)

[3] Bartolini, I., Ciaccia, P., Patella, M.: Salsa: computing the skyline without scanning the whole sky. In: Proceedings of the 15th ACM international conference on Information and knowledge management. pp. 405–414. ACM (2006)

[4] Bartolini, I., Ciaccia, P., Patella, M.: Efficient sort-based skyline evaluation. ACM Transactions on Database Systems (TODS) 33(4), 31 (2008)

[5] Blandford, D.K.: Compact data structures with fast queries. Ph.D. thesis, PhD thesis, School of Computer Science, Carnegie Mellon University, 2006 (2006)

[6] Borzsony, S., Kossmann, D., Stocker, K.: The skyline operator. In: Data Engineering, 2001. Proceedings. 17th International Conference on. pp. 421–430. IEEE (2001)

[7] Brisaboa, N.R., Ladra, S., Navarro, G.: k^2-trees for compact web graph representation. In: International Symposium on String Processing and Information Retrieval. pp. 18–30. Springer (2009)

[8] Chan, C.Y., Eng, P.K., Tan, K.L.: Stratified computation of skylines with partially-ordered domains. In: Proceedings of the 2005 ACM SIGMOD international conference on Management of data. pp. 203–214. ACM (2005)

[9] Chen, H., Zhou, S., Guan, J.: Towards energy-efficient skyline monitoring in wireless sensor networks. In: European Conference on Wireless Sensor Networks. pp. 101–116. Springer (2007)

[10] Choi, W., Liu, L., Yu, B.: Multi-criteria decision making with skyline computation. In: Information Reuse and Integration (IRI), 2012 IEEE 13th International Conference on. pp. 316–323. IEEE (2012)

[11] Chomicki, J., Godfrey, P., Gryz, J., Liang, D.: Skyline with presorting: Theory and optimizations. In: Intelligent Information Processing and Web Mining, pp. 595–604. Springer (2005)

[12] Clark, D.: Compact pat trees. University of Waterloo (1997)

[13] Claude, F., Navarro, G., Ordónez, A.: The wavelet matrix: An efficient wavelet tree for large alphabets. Information Systems 47, 15–32 (2015)

[14] Dellis, E., Seeger, B.: Efficient computation of reverse skyline queries. In: Proceedings of the 33rd international conference on Very large data bases. pp. 291–302. VLDB Endowment (2007)

[15] Eng, P.K., Ooi, B.C., Tan, K.L.: Indexing for progressive skyline computation. Data & Knowledge Engineering 46(2), 169–201 (2003)

[16] Faust, F.J.C., Navarro, G., Cardenas, B.B., Caihuan, M.M., Barbay, J.: Estructuras comprimidas para grafos en la web (2008)

[17] Gao, Y., Chen, G., Chen, L., Chen, C.: An i/o optimal and scalable skyline query algorithm. In: British National Conference on Databases. pp. 127–139. Springer (2006)

[18] Godfrey, P., Shipley, R., Gryz, J.: Maximal vector computation in large data sets. In: Proceedings of the 31st international conference on Very large data bases. pp. 229–240. VLDB Endowment (2005)

[19] González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA). pp. 27–38 (2005)

[20] Hose, K., Vlachou, A.: A survey of skyline processing in highly distributed environments. The VLDB Journal The International Journal on Very Large Data Bases 21(3), 359–384 (2012)

[21] Idrissi, A., Abourezq, M.: Skyline in cloud computing. Journal of Theoretical & Applied Information Technology 60(3) (2014)

[22] Jung, H., Han, H., Yeom, H.Y., Kang, S.: A fast and progressive algorithm for skyline queries with totally-and partially-ordered domains. Journal of Systems and Software 83(3), 429–445 (2010)

[23] Kalyvas, C., Tzouramanis, T.: A survey of skyline query processing. arXiv preprint arXiv:1704.01788 (2017)

[24] Khalefa, M.E., Mokbel, M.F., Levandoski, J.J.: Skyline query processing for incomplete data. In: Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on. pp. 556–565. IEEE (2008)

[25] Kossmann, D., Ramsak, F., Rost, S.: Shooting stars in the sky: An online algorithm for skyline queries. In: VLDB'02: Proceedings of the 28th International Conference on Very Large Databases. pp. 275–286. Elsevier (2002)

[26] Kung, H.T., Luccio, F., Preparata, F.P.: On finding the maxima of a set of vectors. Journal of the ACM (JACM) 22(4), 469–476 (1975)

[27] Ladra, S.: Algorithms and compressed data structures for information retrieval. Ph.D. thesis (2011)

[28] Lee, J., Hwang, S.w.: Bskytree: scalable skyline computation using a balanced pivot selection. In: Proceedings of the 13th International Conference on Extending Database Technology. pp. 195–206. ACM (2010)

[29] Lee, Y.W., Lee, K.Y., Kim, M.H.: Efficient processing of multiple continuous skyline queries over a data stream. Information Sciences 221, 316–337 (2013)

[30] Li, C., Ooi, B.C., Tung, A.K., Wang, S.: Dada: a data cube for dominant relationship analysis. In: Proceedings of the 2006 ACM SIGMOD international conference on Management of data. pp. 659–670. ACM (2006)

[31] Li, H., Tan, Q., Lee, W.C.: Efficient progressive processing of skyline queries in peer-to-peer systems. In: Proceedings of the 1st international conference on Scalable information systems. p. 26. ACM (2006)

[32] Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theoretical Computer Science 387(3), 332–347 (2007)

[33] Navarro, G.: Wavelet trees for all. In: Annual Symposium on Combinatorial Pattern Matching. pp. 2–26. Springer (2012)

[34] Navarro, G.: Compact data structures: A practical approach. Cambridge University Press (2016)

[35] Pagh, R.: Low redundancy in static dictionaries with o (1) worst case lookup time. In: International Colloquium on Automata, Languages, and Programming. pp. 595–604. Springer (1999)

[36] Papadias, D., Tao, Y., Fu, G., Seeger, B.: An optimal and progressive algorithm for skyline queries. In: Proceedings of the 2003 ACM SIGMOD international conference on Management of data. pp. 467–478. ACM (2003)

[37] Papadias, D., Tao, Y., Fu, G., Seeger, B.: Progressive skyline computation in database systems. ACM Transactions on Database Systems (TODS) 30(1), 41–82 (2005)

[38] Peng, P., Chi-Wing, R.: Skyline queries and pareto optimality

[39] Preparata, F.P., Shamos, M.I.: Computational geometry: an introduction. Springer-Verlag (1985)

[40] Quijada Fuentes, C.F.: Ampliación de las Capacidades de Estructuras de Datos Compactas. Ph.D. thesis (2017)

[41] Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms. pp. 233–242. Society for Industrial and Applied Mathematics (2002)

[42] Samet, H.: The quadtree and related hierarchical data structures. ACM Computing Surveys (CSUR) 16(2), 187–260 (1984)

[43] Sharifzadeh, M., Shahabi, C.: The spatial skyline queries. In: Proceedings of the 32nd international conference on Very large data bases. pp. 751–762. VLDB Endowment (2006)

[44] Sharifzadeh, M., Shahabi, C., Kazemi, L.: Processing spatial skyline queries in both vector spaces and spatial network databases. ACM Transactions on Database Systems (TODS) 34(3), 14 (2009)

[45] Son, W., Lee, M.W., Ahn, H.K., Hwang, S.W.: Spatial skyline queries: An efficient geometric algorithm. In: International Symposium on Spatial and Temporal Databases. pp. 247–264. Springer (2009)

[46] Su, I.F., Chung, Y.C., Lee, C., Lin, Y.Y.: Efficient skyline query processing in wireless sensor networks. Journal of Parallel and Distributed Computing 70(6), 680–698 (2010)

[47] Tan, K.L., Eng, P.K., Ooi, B.C., et al.: Efficient progressive skyline computation. In: VLDB. vol. 1, pp. 301–310 (2001)

[48] Vlachou, A., Doulkeridis, C., Kotidis, Y., Vazirgiannis, M.: Efficient routing of subspace skyline queries over highly distributed data. IEEE Transactions on Knowledge and Data Engineering 22(12), 1694–1708 (2010)

[49] Xia, T., Zhang, D., Fang, Z., Chen, C., Wang, J.: Online subspace skyline query processing using the compressed skycube. ACM Trans. Database Syst. 37(2), 15:1–15:36 (Jun 2012), http://doi.acm.org/10.1145/2188349.2188357

[50] Yu, Q., Bouguettaya, A.: Computing service skylines over sets of services. In: Web Services (ICWS), 2010 IEEE International Conference on. pp. 481–488. IEEE (2010)

[51] Zhang, M., Alhajj, R.: Skyline queries with constraints: Integrating skyline and traditional query operators. Data & Knowledge Engineering 69(1), 153–168 (2010)

[52] Zhang, S., Mamoulis, N., Cheung, D.W.: Scalable skyline computation using object-based space partitioning. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of data. pp. 483–494. ACM (2009)

[53] Zhao, M., Yu, J.: Cuttingplane: an efficient algorithm for three-dimensional spatial skyline queries based on dynamic scan theory. In: International Conference on Web Information Systems and Mining. pp. 72–78. Springer (2011)