Un índice espacio temporal para puntos móviles basado en estructuras de datos compactas

Autor: Miguel Esteban Romero Vásquez

Tesis doctoral UDC / 2017

Directores:
Miguel Ángel Rodríguez Luaces
Diego Seco Naveiras

Departamento de Computación

PhD thesis supervised by
Tesis doctoral dirigida por

Miguel Ángel Rodríguez Luaces
Departamento de Computación
Facultad de Informática
Universidade da Coruña
15071 A Coruña (España)
Tel: +34 981 167000 ext. 1254
Fax: +34 981 167160
luaces@udc.es

Diego Seco Naveiras
Departamento de Ingeniería Informática y
Ciencias de la Computación
Facultad de Ingeniería
Universidad de Concepción
Edmundo Larenas 219
4070409 Concepción (Chile)
Tel: +56 41 220 46 92
Tel: +56 41 222 17 70
dseco@udec.cl

Miguel Ángel Rodríguez Luaces y Diego Seco Naveiras, como directores, acreditamos que esta tesis cumple los requisitos para optar al título de doctor y autorizamos su depósito y defensa por parte de Miguel Esteban Romero Vásquez cuya firma también se incluye.

Dedicado a
A mi esposa, padres e hijos.

En memoria de
Gladys Villalobos Sanchez.

Agradecimientos

Gracias a todas las personas que de una u otra manera me han apoyado para que este proyecto llegara a su fin. Sin duda son muchas a las que debo agradecer, pero quisiera hacer una mención especial a alguna de ellas.

A Nieves R. Brisaboa coordinadora del Laboratorio de bases de datos y a mis directores de tesis Miguel R. Luaces y Diego Seco, por sus buenas ideas, paciencia y apoyo.

A Andrea Rodríguez, por su fundamental apoyo al inicio de este proyecto y sus valiosos consejos.

A Gilberto Gutiérrez, por su disposición para conversar y analizar ideas, además de su gran apoyo durante su etapa como Director del Departamento de Ciencias de la Computación al cual estoy adscrito.

A toda mi familia, por su comprensión y apoyo. En especial a mi esposa por acompañarme en esta larga etapa de estudio.

Finalmente dar gracias a Dios por todas estas personas maravillosas que ha puesto en mi vida. Gracias Señor, por todos ellos y por darme el ánimo y la fuerza necesaria para levantarme después de cada caída y mirar al futuro con esperanza y optimismo.

Abstract

Spatio-temporal databases were designed to manage very large datasets of spatial objects, whose location and/or shape evolves with time, and whose changes are relevant to the domain of application. Air traffic control systems, fleet management, migratory birds and other animals are just some examples of this type of systems.

Much research on spatio-temporal databases has been devoted to spatial access methods and efficient indexes for secondary memory. However, just a few works tackle the efficient access in main memory. In the field of Information Retrieval, some strategies have arisen to develop new data structures and efficient algorithms in terms of space usage that also keep efficient access times. These have been called Compact Data Structures. These data structures are very space-efficient (getting good compression ratios), while supporting access queries to the data efficiently without decompressing them.

Similarly, several data structures that use compression techniques have been developed in the context of spatial access methods. However, to the best of our knowledge, there do not exist previous works on the use of compact data structures for spatio-temporal databases.

Therefore, this thesis proposes the use of compact data structures in the context of spatio-temporal databases and, in particular, to index moving objects (represented by spatial coordinates). As a result, a compact self-index is proposed to solve time-slice, time-interval, trajectory and k-nearest neighbor queries. An experimental evaluation shows that our proposal is efficient to support such queries, while using few space.

Resumen

Los sistemas de bases de datos espacio-temporales nacen con el objetivo de manipular grandes volúmenes de objetos espaciales cuya posición y/o forma cambia en el tiempo y donde dichos cambios son relevantes en el dominio de aplicación. Algunos ejemplos son los sistemas de control de tráfico aéreo, los sistemas de control de flotas de vehículos, de aves migratorias y de otros animales.

Se ha investigado mucho en el campo de las bases de datos espacio-temporales en relación a los métodos de acceso e indexación eficientes para memoria secundaria, pero poco para memoria principal.

En el ámbito de los sistemas de recuperación de información han surgido nuevas estrategias para desarrollar estructuras de datos y algoritmos eficientes en el uso de la memoria y que no penalizan los tiempos de acceso, las denominadas Estructuras de Datos Compactas. Estas estructuras de datos son muy eficientes en el uso de la memoria, incluso logrando altos ratios de compresión en algunos casos, a la vez que permiten un acceso eficiente a los datos contenidos sin la necesidad de descomprimir la estructura.

En el campo de la indexación espacial se han desarrollado diversas estructuras de datos que utilizan técnicas de compactación. Sin embargo, no existen trabajos previos de estructuras de datos compactas en el campo de las bases de datos espacio-temporales.

Por lo anterior en este tesis se abordó la temática de las estructuras de datos compactas en el contexto de las bases de datos espacio temporales y en particular, la indexación de objetos móviles, representados como un punto espacial. Como resultado, se ha definido un auto-índice compacto que permite responder a consultas de *time slice*, *time interval*, trayectoria de un objeto y los k vecinos más cercanos. En los experimentos nuestra propuesta demuestra minimizar el espacio utilizado a la vez que es eficiente al responder las consultas dadas en algunos escenarios.

Resumo

Os sistemas de bases de datos espazo-temporais naceron co obxectivo de manipular grandes volumes de obxectos espaciais cuxa posición e/ou forma cambia co tempo e onde ditos cambios son relevantes no dominio da aplicación. Algúns exemplos son os sistemas de control de tráfico aéreo, os sistemas de control de flotas de vehículos, de aves migratorias e doutros animais.

Existe moita investigación no campo das bases de datos espazo-temporais en relación cos métodos de acceso e indexación eficientes para memoria secundaria, pero moi pouca para memoria principal. No ámbito dos sistemas de recuperación da información xurdiron novas estratexias para propor estruturas de datos e algoritmos eficientes no uso de memoria e que non perxudican os tempos de acceso, as denominadas estruturas de datos compactas. Estas estruturas de datos son moi eficientes no uso de memoria, obtendo boas razóns de compresión nalgúns casos, á vez que permiten un acceso eficiente aos datos contidos sen necesidade de descomprimir a estrutura.

No campo da indexación espacial existen diversas estruturas de datos que utilizan técnicas de compactación. A pesar diso, non existen traballos previos de estruturas de datos compactas no campo das bases de datos espazo-temporais.

Debido a todo o exposto, nesta tese abordouse a temática das estruturas de datos compactas no contexto das bases de datos espazo-temporais e, en particular, a indexación de obxectos móbiles, representados como un punto no espazo. Como resultado, definiuse un auto-índice compacto que permite responder consultas de tipo *time-slice*, *time-interval*, traxectoria dun obxecto e os k veciños más próximos. A avaliación experimental amosa que é posible minimizar o espazo empregado á vez que se poden responder as consultas descritas de xeito eficiente.

Índice general

Índice de figuras

Índice de tablas

Lista de algoritmos

Capítulo 1

Introducción

1.1. Contextualización y motivación

El avance científico y tecnológico en el ámbito de las redes y de los dispositivos móviles ha facilitado la generación y recolección de datos que ocurren en una ubicación geográfica y en un instante de tiempo particular. Estos datos, llamados espacio temporales, tienen múltiples aplicaciones. Un caso particular de datos espacio-temporales son los generados por los objetos móviles que no poseen una extensión o área. Dichos objetos son modelados como un punto en el espacio que cambia de ubicación a lo largo del tiempo. Algunos ejemplos de objetos móviles son los vehículos que se mueven en una ciudad, una flota de barcos en el mar o las aves migratorias, entre otros.

Para el tratamiento de estos datos y sus particulares características nacen las llamadas bases de datos espacio-temporales. Estos sistemas de bases de datos son capaces de responder a diferente tipos de consultas, de las cuales las consultas del tipo *timestamp* e *intervalo* son las más estudiadas [TPZ02, GN07].

Una consulta de *timestamp* permite recuperar todos los objetos que se encuentran dentro de un rango espacial y un instante dado. Por ejemplo, en una base de datos de tráfico de vehículos se podría responder a la siguiente pregunta: *«¿cuáles o cuántos vehículos han pasado a menos de 100 metros de las coordenadas (x,y) el 20 de agosto de 2014?»*. La consulta de intervalo extiende el resultado de una consulta de timestamp considerando los instantes consecutivos dentro del intervalo temporal.

Dado el gran volumen de datos que pueden ser almacenados en un sistema de bases de datos espacio temporal es muy importante contar con sistemas de indexación que faciliten la resolución de consultas en forma eficiente, evitando así el tener que

revisar en forma exhaustiva toda la base de datos.

Se ha investigado mucho en el campo de las bases de datos espaciales y espacio-temporales en relación a los métodos de acceso e indexación eficientes lo que ha generado varias decenas de sistemas de indexación en los últimos 30 años [TPZ02, XHL90, NST98, NST99, NST98, TP01b, TP01a, TPZ02, GNR$^+$05].

A pesar del gran número de métodos de acceso espacio-temporal ninguno de ellos ha tenido en consideración el diseño de una estructura de datos que almacene los datos y el índice de un modo compacto, ya que el objetivo de diseño predominante ha sido mejorar los tiempos de respuesta pagando un alto coste en el almacenamiento.

Hasta ahora la mayoría de los esfuerzos para minimizar el coste del almacenamiento no se ha puesto en los sistemas de indexación, sino en los datos. Las soluciones presentadas en la literatura están basadas principalmente en la utilización de técnicas de compresión de datos, la simplificación de trayectorias y compresión semántica de trayectorias. Recientemente se ha demostrado que es posible lograr una disminución significativamente mayor del coste de almacenamiento usando técnicas de compresión de códigos aritméticos que al utilizar técnicas de simplificación de trayectorias con un bajo nivel de error [Koe13]. Sin embargo las técnicas de compresión clásicas y en particular los códigos aritméticos no permiten responder a las consultas espacio temporales sin descomprimir los datos previamente. Dicha situación hace que el uso de técnicas de compresión tradicionales no sean factibles de aplicar en un sistema de indexación espacio-temporal directamente debido al sobrecoste que supone la descompresión de los datos.

En el ámbito de los sistemas de recuperación de información han surgido nuevas estrategias para desarrollar estructuras de datos y algoritmos eficientes en el uso de la memoria y que no penalizan los tiempos de acceso, las denominadas estructuras de datos compactas. Las estructuras de datos compactas permiten minimizar el uso de la memoria logrando altos ratios de compresión en algunos casos y a la vez que permiten un acceso eficiente a los datos contenidos sin la necesidad de descomprimir la estructura. Al minimizar el espacio utilizado es posible, en algunos casos, contener toda la base de datos en memoria principal y con ello disminuir los tiempos de acceso en varios ordenes de magnitud simplemente por no estar en disco. Las estructuras de datos compactas han sido ampliamente estudiadas en el ámbito de la indexación de textos, grafos, entre otras.

En el campo de la indexación espacial se han desarrollado diversas estructuras de datos que utilizan técnicas de compactación. Como por ejemplo: Compact quadtree [AS99], compact kdtree [AMRP12], SpatialWT [Sec09, SNL09], k^2-tree [Lad11, dB14]. Sin embargo no existen trabajos previos de estructuras de datos compactas en el campo de las bases de datos espacio-temporales.

Por lo anterior en este trabajo de tesis se abordó la temática de las estructuras

de datos compactas en el contexto de las bases de datos espacio temporales con la finalidad de confeccionar un auto-índice para puntos móviles que fuera capaz de responder a las preguntas típicas espacio-temporales en forma eficiente.

1.2. Hipótesis y objetivos

En concreto, la hipótesis que sustenta la investigación es la siguiente:

Hipótesis. Es factible definir estructuras de datos compactas y algoritmos para el tratamiento de datos espacio-temporales que permitan hacer un uso eficiente de la memoria y que permitan realizar las operaciones espaciales y/o espacio-temporales típicas con un buen rendimiento.

Para demostrar la hipótesis propuesta, es necesario alcanzar los siguientes objetivos específicos:

- Estudiar los diferentes tipos de estructuras de datos compactas que puedan ser útiles en el contexto de las bases de datos espaciales y/o espacio-temporales, en especial el k^2-tree
- Desarrollar estructuras compactas y algoritmos para bases de datos espacio-temporales, usando un k^2-tree como base para la indexación espacial de puntos
- Desarrollar un prototipo que implemente las estructuras de datos propuestas y sus algoritmos
- Validar la propuesta con una serie de experimentos

1.3. Contribuciones de la tesis

Auto-índice para objetos móviles. La principal contribución de esta tesis es un nuevo índice que permite almacenar e indexar objetos móviles cuya ubicación en el espacio es modelada como un punto. Este índice se basa en un modelo de *Snapshots* + bitácoras de cambio, donde las estructuras de datos para los *snapshots* y para las bitácoras son estructuras de datos compactas. Para el caso de los *snapshots* se utiliza el k^2-tree extendido para indexar objetos, descrito anteriormente. En el caso de las bitácoras, como se presentará más adelante, se plantean tres propuestas:

una para objetos que se mueven a una velocidad máxima de una celda[1] por instante, y dos para objetos que se mueven sin una restricción de velocidad. Se presentan algoritmos que permiten responder consultas de *time slice*, *time interval*, recuperar la trayectoria de un objeto en un intervalo, obtener la ubicación de un objeto en un instante y obtener los k objetos más cercanos a un punto dado en un cierto instante.

Ampliación de las capacidades del k^2-tree para indexar objetos. Esta tesis propone una manera de vincular los puntos que son indexados en un k^2-tree con los objetos que se encuentran en dichas ubicaciones, utilizando para ello, una estructura de datos compacta adicional para indexar los identificadores de objetos y vincularlos al punto geográfico que lo representa. Se presentan dos versiones: una cuando hay solamente un objeto por celda y otra para cuando existen varios objetos por celda. Además, se propone una modificación al algoritmo de consulta por rango de un k^2-tree, para que la respuesta sean todos los objetos y sus ubicaciones vinculadas al rango respectivo. Además, se presenta un algoritmo nuevo que permite obtener la ubicación de un objeto dado su identificador.

Algoritmo de los k vecinos más cercanos en un k^2-tree que indexa objetos. Se presenta un algoritmo para responder a la consultas de los k vecinos más cercanos en un k^2-tree que indexa objetos.

1.4. Organización de la tesis

El resto del documento está organizado de la siguiente manera:

Capítulo 2: Conceptos previos. Presenta una descripción de los conceptos claves presentes en esta tesis. Primero, presenta los conceptos básicos relacionados con las bases de datos espacio-temporales, junto con la descripción de las principales consultas que se dan en estas bases de datos. Posteriormente, se presentan las estructuras de datos eficientes en en espacio, explicando qué son y cómo se clasifican. Además, se presentan las principales estructuras de datos eficientes en espacio.

Capítulo 3: Estado del arte. En este capítulo se presenta el estado del arte que abarca, en sus primeras secciones, la descripción de los principales índices espacio-temporales que existen en la literatura. Luego, se presentan las estrategias usadas

[1] Teniendo en cuenta que el espacio se ha dividido en un grid discreto, una ***celda*** es la porción más pequeña del espacio para la cual se ha definido una ubicación (x, y). Dicha porción puede representar un cm^2, un m^2, un km^2, etc. lo que depende de la decisión del usuario respecto de la granularidad utilizada para representar la ubicación de un objetos.

para minimizar el almacenamiento en dichos índices, que consisten principalmente en la simplificación de trayectoria y las técnicas de compresión clásica. Finalmente, se presentan aquellas estructuras de datos eficientes en espacio que se han utilizado para la indexación espacial. Si bien no están diseñadas para la indexación espacio-temporal, son útiles para definirlos.

Capítulo 4: Índice para objetos móviles con restricciones en el movimiento. Aquí se presenta la primera propuesta de esta tesis, que es un índice espacio-temporal para puntos, el cual hace un particionado del espacio en celdas y el tiempo en instantes discretos con dos restricciones importantes en el modelado de los datos: los objetos se deben mover a celdas adyacentes y no puede existir más de un objeto por celda. Ambas restricciones juntas son un problema, dado que el tamaño de la celda debe aumentar para garantizar que los movimientos sean a celdas adyacentes, pero si se aumenta el tamaño de la celda, es difícil garantizar que exista un punto por celda. Al final del capítulo se realiza un estudio experimental y se discuten los resultados.

Capítulo 5: Índice para objetos móviles sin restricción de movimientos. Debido a las restricciones del índice anterior, se propone una serie de cambios al índice presentado en el capítulo 4 para permitir que más de un objeto pueda existir por celda y que los objetos puedan moverse a una velocidad mayor que una celda por instante de tiempo. Con estos cambios se logra un índice nuevo que no pone restricciones al movimiento de los objetos. Además, se presentan algoritmos que permiten responder a la consulta de los k vecinos más cercanos (kNN). Después, se muestra un estudio experimental donde se evalúa esta propuesta, se compara con el MVR-Tree y se discuten los resultados.

Capítulo 6: Conclusiones. Finalmente se presentan las conclusiones de la tesis, destacando el trabajo realizado y la aportación que hace esta tesis al estado del arte. Junto con ello se presentan algunas ideas que podrían mejorar el rendimiento del índice actual, así como, líneas de investigación que se pueden explorar en un trabajo futuro.

Capítulo 2

Conceptos previos

En este capítulo se presentan los conceptos claves presentes en el estado del arte en relación con esta tesis y que, por lo tanto, son fundamentales para comprender este trabajo. Los tópicos cubiertos en este capítulo están agrupados en dos grandes secciones, por un lado las bases de datos espacio-temporales y por otro, las estructuras de datos eficientes en espacio.

2.1. Bases de datos espacio-temporales y de objetos móviles

Las bases de datos espacio-temporales y las bases de datos de objetos móviles tienen su fundamento en dos áreas diferentes: las bases de datos temporales y las bases de datos espaciales. Ambas áreas se desarrollaron de manera independiente tanto en investigación como en la industria hasta la aparición de las bases de datos espacio-temporales y de objetos móviles, las cuales emergen como una mirada integradora de ambos campos [PTKT04].

Por un lado, las bases de datos temporales nacen con la finalidad de responder a consultas referentes a la evolución de los datos y poder consultar el estado de la base de datos en un instante o un intervalo de tiempo en particular. Esto contrasta con las bases de datos tradicionales, en las cuales, al cambiar el valor de un atributo, el valor anterior se pierde y el nuevo valor ocupa su lugar. Un ejemplo de base de datos temporal sería, por ejemplo, un sistema de cotizaciones de valores donde es necesario conocer la serie de valores que ha tomado el instrumento bursátil durante un intervalo de tiempo, para realizar un análisis de la tendencia de los precios y poder así estimar su comportamiento futuro.

Por otro lado, las bases de datos espaciales surgen por la necesidad de manipular eficientemente objetos espaciales que poseen atributos que pertenecen a un tipo de dato geométrico [GS05]. Estos tipos de datos geométricos junto con los algoritmos que los manipulan son extensiones a los modelos de bases de datos tradicionales.

Los sistemas de bases de datos espacio-temporales y las bases de datos de objetos móviles nacen con el objetivo de manipular grandes volúmenes de objetos espaciales cuya posición y/o forma cambia en el tiempo y donde dichos cambios son relevantes en el dominio de aplicación. Existen innumerables ejemplos de aplicación donde se manipulan datos que poseen atributos espaciales y temporales. Algunos ejemplos son: los sistemas de control de tráfico aéreo, los sistemas de control de flotas de vehículos, o los sistemas de control de aves migratorias y otros animales. Para este tipo de aplicaciones una pregunta relevante que se puede responder con la base de datos es: ¿dónde está un objeto en particular (persona, avión, automóvil, ave, etc.) en un instante dado (a las 4:10 el 27/02/2010)?.

Existen dos enfoques predominantes para gestionar los cambios de la posición de de un objeto. Uno de ellos es tratar el cambio de ubicación de manera discreta, así cada vez que un objeto cambia de posición ésta se registra en la base de datos asociando el tiempo en que ocurre dicho cambio. De este modo un objeto o que en el instante t_1 se encuentra en la ubicación p y luego en el instante t_2 se ha movido a otro lugar, se puede afirmar que o se encuentra en la posición p durante el intervalo $[t_1, t_2)$. Este enfoque es útil cuando los cambios no son muy frecuentes, pues cada cambio requiere una actualización de la base de datos.

El segundo enfoque trata de modelar el movimiento de un objeto en función del tiempo, de este modo se define una función $f : objetos \times tiempo \rightarrow posición$ de forma que la función toma como argumentos un objeto y un instante de tiempo y devuelve la posición que ocupa el objeto en ese instante de tiempo, por ejemplo, $f(o, t_1) \rightarrow p$. Al definir la función no es necesario registrar en la base de datos cada ocurrencia de un cambio en la posición del objeto, sino que se registra cuando cambia la función que describe el movimiento [AAE03]. Algunos sistemas que utilizan modelos a veces definen un cierto umbral de error tolerable de modo que si la diferencia entre la posición observada del objeto en comparación con la posición determinada por la función es menor que el umbral no es necesario registrar el cambio de función.

En la literatura, la mayoría de los trabajos que utilizan la aproximación discreta para tratar el movimiento de los objetos espaciales utilizan la denominación de base de datos espacio-temporal para hacer referencia a su campo de investigación. En cambio, aquellos trabajos que tratan los movimientos de manera continua utilizan la denominación de bases de datos de objetos móviles. Desde la perspectiva de las consultas a la base de datos en general las bases de datos espacio-temporales se concentran en obtener la ubicación de los objetos en un determinado instante o en un determinado intervalo. En cambio en las de objetos móviles el foco de atención

está en la trayectoria de los objetos.

En las bases de datos espacio-temporales y de objetos móviles existen diferentes tipos de datos, los cuales nacen de la incorporación de la dimensión temporal a los tipos de datos espaciales. Los tipos de datos espacio temporales básicos son el punto móvil y la región móvil.

Un punto móvil es un objeto móvil que no posee extensión o área y sirve para modelar diferentes objetos del mundo real como por ejemplo: personas, animales, satélites, aviones, vehículos, paquetes, vehículos militares (cohetes, misiles, tanques, submarinos, etc.) [GS05].

Una región móvil es un objeto móvil que posee una extensión o área y permite modelar diversos fenómenos naturales como artificiales donde la cobertura espacial del objeto y su evolución en el tiempo son relevante en el universo del discurso. Ejemplo de este tipo de objetos son: bosques, lagos, glaciares, tropas, manadas, incendio, tornados, derrame de petróleo, límites políticos como países, condados, regiones, provincias, entre otros [GS05].

Aún existiendo en la literatura la distinción entre bases de datos espacio-temporales y de objetos móviles, también encontramos que es de uso común la utilización de la denominación de bases de datos espacio-temporales para referirse a ambos enfoques. Para simplificar la redacción se utilizará el término base de datos espacio-temporal para referirse a los dos tipos de bases de datos en el resto de este texto.

Al contar con una base de datos espacio-temporal es posible realizar diferentes tipos de consulta las cuales involucran información de distinta índole como, por ejemplo, la información temporal, espacial, propia del objeto y relacionada al movimiento. Considere por ejemplo una base de datos de vehículos que se desplazan en una ciudad donde se han definido diferentes tipos de vehículo como taxis, ambulancias, camionetas, camiones, etc., se podrían responder consultas como las siguientes: obtener la ubicación de todos los taxis de la ciudad, obtener la ubicación de todos los vehículos que sean de la marca Seat modelo León de color blanco, obtener la lista de vehículos que han pasado por una esquina determinada a una cierta hora, entre otras.

Dependiendo de que tipo de información es conocida y que tipo de información deseamos conocer, es posible clasificar a las consultas espacio temporales en perspectivas. Sultan Alamri y otros [ATS14] identifican cinco perspectivas:

- **Perspectiva de ubicación**. En esta perspectiva el elemento clave de la consulta es la ubicación, de modo que el resultado de la consulta está determinado por un punto o una región determinada. Por ejemplo, obtener todos los vehículos que se encuentran a 3 km del punto p.

- **Perspectiva del movimiento**. En esta perspectiva las consultas tienen en cuenta la velocidad, dirección, distancia y/o desplazamiento de un objeto. Por ejemplo: obtener todos los vehículos que se desplazan a más de 60km/h.

- **Perspectiva del objeto.** Los objetos poseen características propias y además son de algún tipo, por ende se pueden distinguir dos tipos de consulta: consultas que dependen del tipo del objeto, como por ejemplo, obtener la ubicación actual de todos los vehículos del tipo ambulancias, y consultas que dependen de las características propias del objeto, por ejemplo, obtener la ubicación actual de todos los vehículos del año 2010 y de color rojo.

- **Perspectiva temporal.** En esta perspectiva la información clave a la hora de responder las consultas es el tiempo. En este contexto se distingue dos abstracciones claves: **instante**, que señala un punto en la línea temporal que es atómico y sin duración; e **intervalos**, los cuales expresan períodos de tiempo que tienen una duración y están determinados por un instante inicial y un instante final. Por ejemplo, obtener la ubicación de todos los vehículos el día 12 de enero de 2012 a las 15:20 hrs.

- **Perspectiva de patrón.** Cuando los objetos se desplazan por el espacio o bien por una red, es posible identificar la forma del desplazamiento y descubrir en ellos un patrón. En esta perspectiva la información clave que determina los resultado es un determinado patrón, y el resultado serán aquellos objetos cuyos desplazamientos calcen con dicho patrón. Los patrones por su parte pueden ser de tres tipos: Patrones espaciales, patrones temporales y patrones espacio-temporales. Por ejemplo, obtener todos los vehículos que hayan efectuado el siguiente recorrido: transita por calle libertad, dobla a la izquierda en calle 5 de abril, dobla a la derecha por calle gamero, y se estaciona.

A la hora de hacer una consulta en particular se pueden mezclar las perspectivas y eso hace que el sistema de consultas de una base de datos espacio temporal sea muy rico en semántica, a la vez que impone un desafío importante a los sistemas de indexación.

Un ejemplo de tipo de consulta que mezcla dos perspectivas son las de rango espacio-temporal. Este tipo de consulta permite obtener a todos los objetos que se encuentran dentro de un rectángulo paralelo a los ejes del plano (perspectiva de ubicación) en un instante o intervalo de tiempo en particular (perspectiva temporal).

Otro ejemplo de consulta son las de tipo punto (*point queries*). Este tipo de consulta permite obtener la ubicación de un objeto móvil en particular (perspectiva del objeto) en un instante de tiempo en particular(perspectiva temporal). Si en vez de un instante de tiempo se da un intervalo temporal, el resultado es la trayectoria del objeto durante el intervalo dado.

Un tercer tipo de consulta son las de los k vecinos más cercanos (*k-nearest neighbor queries*). En este tipo de consulta se obtienen los k objetos que se encuentran más cercanos a una ubicación en el espacio, que puede ser dada por un objeto espacial o móvil. Por ejemplo encontrar los 5 hoteles más cercanos a un punto de interés turístico.

Los tipos de consultas de objetos móviles nombrados anteriormente (*point queries*, *range queries* y *k-nearest neighbor queries*) son los más comunes en sistemas de bases de datos de objetos móviles [Dan06].

Basado en [AAE03] a continuación se presenta una definición formal de las consultas de rango y los k-vecinos más cercanos para una base de datos de puntos móviles, dado que son el foco de interés en esta tesis:

Definición 2.1 *Sea $S = \{p_1, p_2, \ldots, p_n\}$ un conjunto de puntos móviles en el plano xy. Para cualquier tiempo t, $p_i(t)$ denota la posición de p_i al tiempo t, y $S(t) = \{p_1(t), p_2(t), \ldots .p_n(t)\}$. Dado lo anterior se definen las siguientes consultas:*

- ***Rango-instante**. Dado un rectángulo R paralelo a los ejes y un instante de tiempo t, esta consulta da como resultado todos los puntos de S que se encuentran dentro de R en el tiempo t, es decir $S(t) \cap R$.*
- ***Rango-intervalo**. Dado un rectángulo R paralelo a los ejes y dos instantes de tiempo t_1 y t_2, esta consulta da como resultado todos los puntos de S que se encuentran dentro de R en cualquier instante entre t_1 y t_2, es decir $\bigcup_{t=t1}^{t2} (S(t) \cap R)$.*
- ***El vecino más cercano**. Dado un punto de consulta $\sigma \in R^2$ y un instante de tiempo t, esta consulta da como resultado el punto $p \in S$ tal que $d(\sigma, p(t)) \leq d(\sigma, r(t))$ para todo $r \in S$, donde $d(\cdot, \cdot)$ es la distancia euclidiana.*

2.2. Entropía en Teoría de la Información

En esta sección se presentan algunos conceptos básicos de la teoría de la información, los cuales son fundamentales para comprender las estructuras de datos eficientes en espacio que se describen más adelante.

Una de las aportaciones centrales del trabajo de Shannon [Sha48], en su teoría matemática de la comunicación fue la de obtener una manera de medir la cantidad de información. Para ello plantea el concepto de incertidumbre o de elección y lo asocia con el de información. El plantea la siguiente pregunta: «Supongamos que tenemos un conjunto de posibles eventos cuya probabilidad de ocurrencia son $P_1, P_2, \cdots, P_n$. Estas probabilidades son conocidas, pero eso es todo lo que sabemos acerca de cual

evento se producirá. ¿Podemos encontrar una medida de la cantidad de "elección" que está involucrada en la selección del evento o de la incertidumbre que tenemos de los resultados?»

Como la probabilidad de los eventos es conocida, lo que hace la diferencia entre los resultados esperados y los resultados reales es la información, es decir, la incertidumbre respecto de la elección que hará el emisor del mensaje.

Formalmente se define la *cantidad de información* (en bits) de un símbolo fuente x como

$$h(x) = log_2 \frac{1}{P(x)}$$

donde $P(x)$ es la probabilidad de ocurrencia del símbolo x.

Dada $h(x)$ se puede observar que si la probabilidad es alta ($P(x) = 1$) la cantidad de información es 0, dado que x da muy poca información. Por ejemplo, si tenemos un dado donde todas sus caras son 5, lanzar el dado no aportará información, pues de antemano sabemos el resultado. Por otro lado, si la probabilidad es muy baja ($P(x)$ tiende a 0) la ocurrencia de x posee una alta cantidad de información.

Junto con el concepto de contenido de información de un símbolo está el de *contenido de información promedio* de un vocabulario que se calcula ponderando la probabilidad de cada símbolo con su contenido de información $h(x)$, lo que se denomina *Entropía*.

Dado un vocabulario fuente $X = \{x_1, x_2, x_3, \cdots, x_n\}$ la entropía del vocabulario $H(X)$ se define como:

$$H(X) = \sum_{i=1}^{n} P(x_i) \log_2 \frac{1}{P(x_i)}$$

Por propiedades de los logaritmos, la entropía también se puede expresar como:

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log_2 P(x_i)$$

Un caso particular de la entropía ocurre cuando todos los símbolos del vocabulario son independientes y tienen la misma probabilidad. En este caso la entropía corresponde a $H(X) = log_2 n$.

Un concepto relacionado con la entropía es el de redundancia. La redundancia se define como la diferencia entre la cantidad de bits promedio que se usan para codificar los símbolo del vocabulario y la entropía del vocabulario [FBN05]:

$$R = \sum_{i=1}^{n} P(x_i) l(x_i) - H(X),$$

donde $l(x_i)$ es el largo del código asignado al símbolo x_i de X.

2.2.1. Entropía en mensajes dependientes del contexto

El *contexto de un símbolo fuente* x_i es definido como una secuencia de largo fijo de símbolos fuente que le precede. Dependiendo del largo del contexto se definen varios modelos del texto que se pueden usar, de modo que si el largo del contexto es k se dice que el modelo es de orden k.

Dependiendo del modelo, la manera de calcular la entropía varía [FBN05]:

- *Modelo de orden base*. En este caso se considera que todos los símbolos son igual de probables e independientes: $H_{-1}(X) = log_2 n$
- *Modelo de orden cero*. En este caso, todos los símbolos son independientes, pero unos símbolos son más frecuentes que otros: $H_0(X) = -\sum_{i=1}^{n} P(x_i) \log_2 P(x_i)$
- *Modelo de primer orden*: La probabilidad de ocurrencia de un símbolo x_j está condicionada por la ocurrencia previa de x_i:

$$H_1(X) = -\sum_{i=1}^{n} P(x_i) \sum_{j=1}^{n} P(x_j|x_i) \log_2 P(x_j|x_i)$$

- *Modelo de segundo orden*: La probabilidad de ocurrencia de un símbolo x_k está condicionada por la ocurrencia previa de la secuencia $x_i x_j$:

$$H_2(X) = -\sum_{i=1}^{n} P(x_i) \sum_{j=1}^{n} P(x_j|x_i) \sum_{k=1}^{n} P(x_k|x_j, x_i) \log_2 P(x_k|x_j, x_i)$$

- Los modelos de ordenes superiores siguen la misma idea.

2.2.2. Cota inferior de la Teoría de la Información

La importancia de la entropía radica en que ella nos provee de una cota inferior para el número de bits por símbolo que serán necesarios para codificar todo el mensaje de origen.

El concepto de entropía no solo se aplica a la comunicación de datos como un flujo de símbolos que pertenecen a un vocabulario, también se puede aplicar a las estructuras de datos y con ello conseguir una cota inferior del espacio necesario para que dicha estructura almacene una cierta cantidad de información.

Por ejemplo si tenemos un dominio D con n objetos y todos tienen igual probabilidad de ocurrir, la entropía de D sería $H(D) = log_2 n$, de este modo se necesitan como mínimo $\lceil \log_2 n \rceil$ bits para identificar de forma única a un objeto y distinguirlo de cualquier otro objeto perteneciente a D.

Otro ejemplo. Considere una cuadricula de $n \times n$ que contiene m puntos, es posible calcular la entropía si consideramos que el vocabulario está constituido por todas las posibles cuadrículas, cuyo tamaño sería $\binom{n^2}{m}$ y la entropía sería [FGN13]:

$$\begin{aligned} H &= \log_2 \binom{n^2}{m} \\ &= m \log_2 \frac{n^2}{m} + (n^2 - m) \log_2 \frac{n^2}{n^2 - m} + \mathcal{O}(\log_2 n) \\ &= m \log_2 \frac{n^2}{m} + (n^2 - m) \log_2 (1 + \frac{m}{n^2 - m}) + \mathcal{O}(\log_2 n) \\ &= m \log_2 \frac{n^2}{m} + (n^2 - m) + \mathcal{O}(m + \log_2 n) \end{aligned}$$

y por lo tanto se requieren como mínimo $\lceil m \log_2 \frac{n^2}{m} + (n^2 - m) + \mathcal{O}(m + \log_2 n) \rceil$ bits para codificar cada una de estas cuadrículas.

En los ejemplos presentados aquí se ha usado la entropía para el modelo de orden base, pero dependiendo de la naturaleza de los datos, es decir, su distribución de probabilidad y si son independientes o dependientes del contexto, la cota inferior estará definida por la entropía para el modelo de orden que más se ajuste a la naturaleza de los datos.

2.3. Estructuras de datos eficientes en espacio

Contar con estructuras de datos eficientes en espacio es un objetivo importante en aquellos contextos donde se requiera que todos los datos se encuentren en memoria principal, como por ejemplo en los sistemas de indexación de motores de búsqueda [SG06].

Dado el gran volumen de datos que algunos sistemas pueden contener, es necesario utilizar alguna estrategia que optimice el espacio en memoria principal que estos utilizan. Una idea podría ser la compresión de los datos, y con ello lograr poner más información en la memoria principal. Sin embargo, si lo único que hacemos es comprimir los datos, éstos no se podrán usar sin antes descomprimirlos, lo que implica un acceso muy lento a la hora de recuperar información.

Como una solución al problema señalado se han desarrollado estructuras de datos que son eficientes tanto en el tiempo de acceso a la estructura como en el espacio utilizado, esto último significa que almacenan los datos usando una cantidad de espacio cercana a la cota inferior de la teoría de la información [JSS12].

Alcanzar esta cota inferior es todo un desafío, dado que las estruturas de datos en general necesitan de una considerable cantidad de espacio adicional para la organización de los datos. Por ejemplo, en un árbol binario se necesitan dos punteros para enlazar un nodo con sus hijos. Si cada puntero ocupa la misma cantidad de bits que el dato almacenado en el nodo, el espacio adicional equivale al 200 % del espacio ocupado por los datos. A este espacio adicional se le denomina «redundancia de la estructura de datos» [BF10].

Dependiendo de la redundancia de la estructura y del espacio utilizado por la representación de los datos, en la literatura se distinguen varios tipos de estructuras de datos eficientes en espacio, las cuales se presentan a continuación.

Estructuras de datos compactas (*Compact Data Structure)*). Las estructuras compactas son aquellas que usan un número de bits que está dentro de una factor constante de la cota inferior($\mathcal{O}(nH_{-1})$ bits) [Bla06].

Estructuras de datos sucintas (*Succinct Data Structure*). Las estructuras sucintas son estructuras de datos cuya redundancia es asintóticamente insignificante ($o(nH_{-1})$ bits) en relación al espacio requerido para codificar los datos [BF10]. Un ejemplo de estructura de datos sucinta es el *Dynamic bit vector* [HSS11], el cual necesita de $n + o(n)$ bits de espacio para codificar una secuencia de n bits y permitir operaciones en tiempo constante sobre él.

Estructuras de datos implícitas (*Implicit Data Structure*). Las estructuras de datos implícitas son aquellas que no necesitan información adicional a los datos para describir su estructura pues ésta se encuentra implícita en la forma en que los datos son almacenados [MS80]. Ejemplos clásicos de estas estructuras son el *Heap* y un *array* ordenado visto como un árbol binario de búsqueda implícito [MS80] accediendo a él mediante búsqueda binaria.

Estructura de datos comprimida(*Compressed Data Structures*). En una *estructura de datos comprimida* que codifica n objetos el espacio total usado por los datos está acotado por nH_0, o bien por una entropía de orden superior. A pesar de que los datos estan comprimidos, las operaciones sobre la estructura son eficientes en tiempo. Esto se logra al manipular directamente los datos comprimidos sin la necesidad de descomprimirlos o bien descomprimiendo una pequeña porción de los

datos, porción asintóticamente insignificante, garantizando así que las operaciones sean resueltas de manera eficiente [SG06, GHSV06, OS06].

Dependiendo de la redundancia de la estructura comprimida, en la literatura se pueden distinguir dos tipos de estructuras de datos comprimidas [BF10],estas son:

Estructuras de datos ultra sucintas (*Ultra-Succinct Data Structure*). Son estructuras de datos comprimidas cuya redundancia es asintóticamente insignificante en relación al espacio requerido para codificar los datos sin comprimir (o(nH_{-1}) bits). Un ejemplo de este tipo de estructura lo encontramos en [JSS12] donde se presenta una estructura de datos ultra sucinta para representar árboles cardinales usando $nH^*(T) + o(n)$ bits, donde $H^*(T)$ es el grado de la entropía del árbol T y n el número de nodos.

Estructuras de datos sucintas comprimidas (*Compressed Succinct Data Structure* o *Fully Compressed Representation*). Son estructuras de datos comprimidas cuya redundancia es asintóticamente insignificante en relación al espacio requerido para codificar los datos de manera comprimida, en otras palabras la redundancia es $o(nH_0)$ bits. Por ejemplo, en [FGN13] presentan una representación de un conjunto de m puntos en un *grid* $n \times n$, utilizando $H + o(H)$ bits de espacio, donde $H = \log_2 \binom{n^2}{m}$.

Las estructuras de datos eficientes en espacio pueden ser utilizadas en la elaboración de un índice. Un índice es una estructura de datos que permite el acceso eficiente a otra estructura de datos. El índice, desde el punto de vista del espacio es redundancia, respecto de los datos indexados. Un *Índice sucinto* (también llamado *Estructura de datos sistemática*) es un índice que representa una redundancia asintóticamente insignificante en relación a los datos de la estructura indexada [BF10].

Cuando el índice y los datos se encuentran combinados de manera tal que forman una única estructura de datos, esta se conoce como *auto índice* [BF10]. Otros nombres que recibe el auto índice es el de *Estructura de datos no sistemática*, *encoding data structure* e *integrated encoding* [BF10].

A continuación se presentan varias estructuras eficientes en espacio, de estas, las secuencias binarias, permutaciones y k^2-tree son la base de nuestra propuesta.

2.3.1. Secuencias binarias

Una secuencia binaria (en inglés: *bitmap*, *bit array*, *bitset*, *bit string*, o *bit vector*) es una estructura de datos básica usada para compactar varias estructuras de datos tradicionales como: árboles binarios, indexación de texto, árboles etiquetados, grafos, y muchas otras. Un *bitmap* $B_{1...n}$ es una secuencia arbitraria de n bits que posee las siguientes operaciones básicas:

- ACCESS(B, i), permite obtener el valor del i-ésimo bit de la secuencia ($B[i]$).
- RANK(B, i) que devuelve la cantidad de bits en 1 que aparecen en la subsecuencia B desde el primer bits al i-ésimo.
- SELECT(B, i) que devuelve la posición del i-ésimo 1 en B.

Existen diversas estructuras de datos que implementan estas operaciones eficientemente sobre una secuencia binaria.

Una de las primeras soluciones fue presentada por Jacobson, quien define una estructura que utiliza $n + o(n)$ bits de espacio y requiere $o(\log n)$ accesos a bits para las operacioens de rank y select [Jac89].

Este trabajo fue extendido por Clark [Cla96] y Munro [Mun96] quienes utilizando $n + o(n)$ bits de espacio obtienen rank y select en tiempo constante.

Más tarde Pagh [Pag99] presenta una estructura de datos comprimida que permite responder rank en tiempo constante usando $m \lg \frac{n}{m} + \mathcal{O}(m + \frac{n \lg \lg n}{\lg n})$ [FGN13].

Una estructura que da soporte tanto para rank como para select es presentada por Raman y otros [RRR07] utilizando también $m \lg \frac{n}{m} + \mathcal{O}(\frac{n \lg \lg n}{\lg n})$.

Okanohara y Sadakane [OS06] proponen cuatro estructuras cada una con diferentes ventajas y desventajas. Una de ellas es *Sdarray* que usa dos diferentes técnicas *Darray* para bitmaps densos y *Sarray* para bitmaps poco densos. Donde *Darray* requiere $n + o(n)$ bits y *Sarray* $m\lceil lg \frac{n}{m} \rceil + 2m + o(m)$ y responde select en tiempo constante y rank en tiempo $\mathcal{O}(\frac{n}{m})$.

2.3.2. Wavelet trees

Un *Wavelet Tree* es una estructura de datos presentada originalmente para representar secuencias de símbolos [GGV03], pero que se puede utilizar también en otros dominios como la representación de permutaciones y un *grid* de puntos [Nav14].

Sea $S[1, n] = s_1 s_2 \dots s_n$ una secuencia de símbolos $s_i \in \Sigma$ donde $\Sigma = [1..\sigma]$ es llamado alfabeto. Esta secuencia puede ser representada de manera plana utilizando $n\lceil log_2 \sigma \rceil = n \log_2 \sigma + \mathcal{O}(n)$ bits [Nav14].

Un wavelet tree es un árbol binario que codifica S como una partición recursiva del alfabeto. Cada nodo interno del wavelet tree representa a un rango $[a, b] \subset [1..\sigma]$ y la subsecuencia de símbolos de S que pertenecen a dicho rango.

Cada nodo particiona el rango del alfabeto que le corresponde en dos, asignando la primera mitad al hijo izquierdo y la segunda al derecho, lo que corresponde a los rangos $[a, m]$ y $[m + 1, b]$ respectivamente, donde $m = (a + b)/2$. Cuando $a = b$

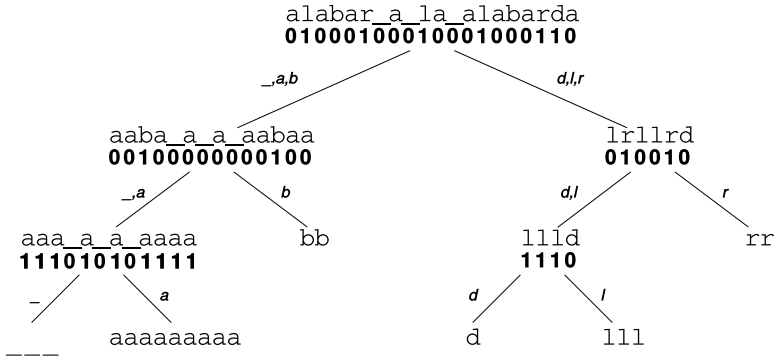


Figura 2.1: Ejemplo wavelet tree para la secuencia `alabar_a_la_alabarda` [Nav14]. el símbolo `_` indica un espacio en blanco

significa que el rango del alfabeto es solo un símbolo, y por lo tanto el nodo es una hoja.

La subsecuencia de S asociada a un nodo interno v es codificada con un bitmap B_v, de modo que si el símbolo $S[i] \leq m$, entonces $B_v[i] = 0$ y en caso contrario $B_v[i] = 1$.

Así como el rango del alfabeto fue particionado en dos mitades, también se hace lo mismo con la subsecuencia asociada al nodo, de modo que todo símbolo de $S[i] \leq m$ formará parte de la subsecuencia S_0 que es asociada al hijo izquierdo y los símbolos $S[i] > m$ forman la secuencia S_1 que se asocia al hijo derecho.

En la figura 2.1 se aprecia un wavelet tree para la secuencia $S =$`alabar_a_la_alabarda` con $\Sigma =$`{_,a,b,d,l,r}` $n = 19$ y $\sigma = 6$ [Nav14]. Como se puede observar en la figura, el nodo raíz tiene asociado el alfabeto y la secuencia completa. Al particionar el alfabeto en dos, el hijo izquierdo representa la subsecuencia que contiene las letras `_,a,b` y el hijo derecho tiene la subsecuencia para el resto del alfabeto. Los hijos se siguen subdividiendo recursivamente hasta que los rangos son letras individuales.

Es importante señalar, que únicamente se almacenan los *bitmaps* de los nodos, pero no se almacenan, ni las subsecuencias ni los rangos asociados a éstos.

Así, para conocer $S[i]$ es necesario recorrer el árbol desde la raíz hasta las hojas, utilizando para navegar el árbol la información contenida en los *bitmpas*. Para explicar el proceso se dará un ejemplo que usa la figura 2.1. Para referirnos a los nodos se han enumerando desde arriba a bajo y de izquierda a derecha $(1 \ldots 5)$.

Suponga que desea conocer el valor de $S[5]$ ($\mathsf{access}(S, 5) = a$), lo primero que

hay que hacer es visitar la raíz (nodo 1) y consultar por el valor de $B_1[5]$ el cual es 0 ($access(B_1, 5)$), lo que significa que el caracter buscado está en el sub-árbol izquierdo y es uno entre $\{\texttt{\_}, \texttt{a}, \texttt{b}\}$. Luego se continúa en el hijo izquierdo de la raíz, pero en vez de buscar el quinto bit, se busca el cuarto, dado que el quinto caracter en S le corresponde el cuarto cero ($\mathsf{rank}_0(B_1, 5)$) en el *bitmap* y por lo tanto será el cuarto símbolo de S_0. Al examinar el cuarto bit del segundo nodo, nos encontramos el valor 0 nuevamente, por lo tanto hay que buscar el $\mathsf{rank}_0(B_2, 4) = 3$ y por lo tanto descendemos por el hijo izquierdo buscando el tercer bit. En este punto los posibles caracteres se reducen a dos $\{\texttt{\_}, \texttt{a}\}$. Al revisar el tercer bits del nodo 4 encontramos un 1 y por lo tanto el caracter buscado está en la partición del alfabeto asociada al hijo derecho y, por lo tanto, hay que descender esta vez buscando el $\mathsf{rank}_1(B_4, 3) = 3$. En esta última llamada se llega al caso base de la recursión, dado que el rango del alfabeto $[a, b]$ contiene un único caracter ($a = b$) el cual es la respuesta ($\texttt{a}$).

Para responder a la consulta rank_c se procede de manera muy similar. Por ejemplo si se quisiera conocer cuantas letras l hay hasta la posición 11 de S, es decir $\mathsf{rank}_\mathsf{l}(S, 11) = 2$, el proceso comienza en la raíz y avanza hacia las hojas.

Al visitar la raíz, lo primero que hay que revisar es si el símbolo consultado se encuentra en la primera mitad o en la segunda del alfabeto. En nuestro ejemplo el símbolo l se encuentra en la segunda mitad, entonces, se debe proseguir la búsqueda en el hijo derecho, el cual representa el subconjunto de símbolos $\{\texttt{d}, \texttt{l}, \texttt{r}\}$. Al igual como sucede con la operación *access*, es necesario ir ajustando la posición de la consulta al descender por el árbol utilizando la operación de *rank* correspondiente. En efecto, se calcula $\mathsf{rank}_1(B_1, 11) = 3$ y se continua explorando el tercer bits del hijo derecho de la raíz (el tercer nodo), repitiendo el proceso. Al revisar el tercer bit del tercer nodo encontramos un 0, y por lo tanto hay que seguir por el hijo izquierdo buscando el bit número $\mathsf{rank}_0(B_3, 3) = 2$. Como el segundo bits del nodo 5 es un 1, se continúa buscando en el hijo derecho el $\mathsf{rank}_1(B_5, 2) = 2$. En esta última llamada se alcanza la condición de término de la recursión, porque el rango del alfabeto a buscar contiene un solo símbolo que es la letra $\texttt{l}$. En este punto y a diferencia de la operación *access* la respuesta es el valor que se debería buscar, es decir, el resultado de la última operación de *rank* calculada que fue 2.

Para la operación de $\mathsf{select}_\mathsf{c}(S, i)$ se sigue la misma estrategia que se usó para la operación de *access*, pero con dos diferencias importantes: se debe comenzar desde la hoja correspondiente al símbolo c hasta la raíz y al subir por el árbol, ajustamos la posición de búsqueda en los *bitmap* usando una operación de *select* en vez de *rank*.

Por ejemplo, si se desea conocer la posición de la segunda ocurrencia del símbolo $\texttt{r}$, es decir $\mathsf{select}_\mathsf{r}(S, 2) = 18$, se debe comenzar buscando desde el tercer nodo. Como la letra r corresponde a la segunda mitad, se busca la posición del segundo 1 en B_3, el cual está en la posición 5 ($\mathsf{select}_1(B_3, 2) = 5$). A continuación, como el tercer nodo es el hijo derecho de su padre, se debe buscar el quinto 1 del primer nodo, es

decir $\mathsf{select}_1(B_1, 5) = 18$ y como se ha llegado a la raíz del árbol la respuesta es el último *select* calculado, es decir 18.

Un Wavelet tree permite codificar S de un modo sucinto y permite responder a consultas de tipo access, rank_c y $\mathsf{select}_\mathsf{c}$ en tiempo $\log_2 \sigma$ utilizando un espacio de $n \log_2 \sigma + o(n \log_2 \sigma)$ [Cla13].

El wavelet tree es una estructura muy estudiada, para la cual se han propuesto varias variantes que mejoran los tiempos y/o el espacio utilizado. Si se desea profundizar en estas variantes se sugiere comenzar revisando [Nav16] .

2.3.3. Permutaciones

Una permutación π es un ordenamiento de los valores de un conjunto $\{1, 2, \ldots, n\}$, el cual puede ser representada por medio de un *array* $\pi[1, n]$ donde cada valor $1 \leq i \leq n$ aparece exactamente una vez [Nav16].

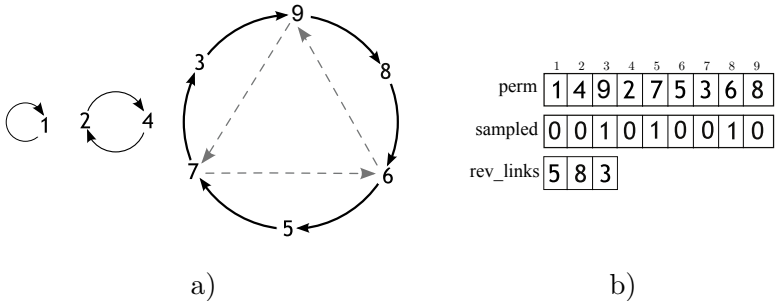
Existen dos operaciones básicas sobre una permutación: $\pi(i)$ que permite obtener el valor que ocupa la i-ésima posición en el ordenamiento, y $\pi^{-1}(i)$ que es la permutación inversa de i, es decir, el número j para el cual $\pi(j) = i$.

Por medio del *array* $\pi[1, n]$ es posible responder $\pi(i)$ para cualquier valor de i en tiempo constante, pero $\pi^{-1}(i)$ requiere $\mathcal{O}(n)$. Si el espacio no es importante, sería fácil responder $\pi^{-1}(i)$ en tiempo constante por medio de otro *array*, lo cual duplica el espacio.

En el estado del arte existen varias estructuras de datos compactas para representar permutaciones [MRRR03, BF10, MRRR12, BN13, Nav14]. Una de ellas es el wavelet tree presentado en la sección 2.3.2 el cual permite responder π y π^{-1} en tiempo $\mathcal{O}(\log_2 n)$, utilizando un espacio de $n \log_2 n + o(n \log_2 n)$, dado que $\sigma = n$.

Otra estructura es la presentadada por Munro en [MRRR03] la cual permite obtener π en tiempo constante y π^{-1} en tiempo $\mathcal{O}(t)$ usando tan solo $(1 + 1/t)n \log_2(n) + o(n)$ bits de espacio. La constante t permite ajustar el trade-off espacio tiempo según convenga. Si se requiere responder más rápido π^{-1} se disminuye t, pero aumenta el espacio. Por ejemplo, si $t = 1$, π^{-1} se responde en tiempo constante pero con el coste de almacenamiento más alto $2n \log(n) + o(n)$ bits que es equivalente a tener dos arrays uno para responder π y otro para π^{-1}. Si $t = \log(n)$ entonces se requiere $n \log(n) + o(n)$ bits [MRRR03] y se responde π^{-1} en tiempo $O(\log(n))$, que es el mismo que se consigue con un wavelet tree.

La estructura de datos para la permutación utiliza dos *arrays* y un *bitmap*. El *array perm* contiene la permutación de los valores. Por ejemplo, en la figura 2.2 b) la posición 5 de *perm* contiene un 7, es decir $\pi(5) = 7$. Así, por medio de *perm* se



a) b)

Figura 2.2: Ejemplo de permutación $\pi = \{1, 4, 9, 2, 7, 5, 3, 6, 8\}$. En a) se muestran los ciclos de la permutación y en b) su representación por medio de la estructura de datos sucinta de [MRRR03] con $t = 2$. Las flechas con línea punteada representan los punteros reversos que acortan el recorrido de los ciclos.

obtiene π es $O(1)$. El *bitmap sampled* y el *array* de enteros *rev_links* completan la estructura, y se utilizan para responder π^{-1}.

El concepto matemático de ciclo de una permutación es clave para entender como funciona la estructura y permite omitir la creación de un *array* que almacene directamente π^{-1}.

En una permutación puede existir uno o más ciclos, los cuales se obtienen al ir visitando de forma iterativa los valores de la permutación comenzando desde una cierta posición i, saltando a la posición $\pi(i)$ y repitiendo el proceso hasta llegar a la posición inicial. Gracias a los ciclos es posible encontrar $\pi^{-1}(i)$. Por ejemplo, en el *array perm* de la figura 2.2 b) existen tres ciclos (1), (2 4) y (9 8 6 5 7 3), los cuales se han graficado en 2.2 a). Así, por ejemplo si queremos saber $\pi^{-1}(3)$ recorremos el ciclo de la permutación partiendo de la posición 3 del *array perm*, como en la posición 3 hay un 9 entonces se visita la posición 9, la cual tiene el valor 8 y por lo tanto se continua en la posición 8 del *array* y así sucesivamente se recorren las posiciones restantes del ciclo: 6, 5 y 7. Dado que en la posición 7 se encuentra el 3, la respuesta de $\pi^{-1}(3) = 7$.

Una permutación puede tener, en el peor de los casos, un único ciclo de largo n y por lo tanto el coste de encontrar π^{-1} en ese caso será $\mathcal{O}(n)$ tiempo.

Con la finalidad de acortar el recorrido en los ciclos largos, se definen punteros reversos, los cuales serán un atajo para que todo recorrido se realice en a lo más t saltos, independiente de la posición inicial. En la figura 2.2 a), existen 3 punteros reversos, los cuales se representan como flechas con líneas punteadas.

Un puntero reverso está asociado a una cierta posición de la permutación. Para indicar dicha asociación se utiliza el *bitmap* llamado *sampled*, el cual tiene un 1 en el bit i si el término i de la permutación tiene asociado un puntero reverso, en caso contrario un 0.

Los punteros reversos son almacenados en el *array* de enteros *rev_links* siguiendo el orden de la subsecuencia de unos de *sampled*. De este modo, el puntero reverso asociado a la posición i de la permutación se encuentra en $rev\_links[\mathsf{rank}_1(sampled, i)]$.

El proceso para calcular $\pi^{-1}(i)$ cambia respecto del recorrido natural del ciclo, de modo que al visitar una posición j, si ésta tiene asociado un puntero reverso, en vez de continuar el recorrido en $perm[j]$ se hace en la posición del puntero reverso. Durante el recorrido del ciclo se debe utilizar solo el primer puntero reverso alcanzado, de lo contrario se entrará en un ciclo infinito.

Por ejemplo, para el caso visto anteriormente $\pi^{-1}(3) = 7$ el recorrido para encontrar la repuesta fue $perm[3] = 9$, $perm[9] = 8$, $perm[8] = 6$, $perm[6] = 5$, $perm[5] = 7$. Usando *sampled* y *rev_links* el recorrido sería así: primero se revisa si existe un puntero reverso para 3, es decir, si el tercer bits de *sampled* es 1. Como éste es el caso se recupera el puntero reverso asociado que corresponde a $rev\_links[\mathsf{rank}_1(sampled, 3)] = 5$; luego se continua con $perm[5] \rightarrow 7$ y como en la posición 7 de *perm* hay un 3, entonces 7 es la respuesta y se obtuvo en 2 pasos en vez de 5.

No siempre el puntero reverso estará al principio del ciclo. Por ejemplo, para encontrar $\pi^{-1}(6)$, como asociado a la posición 6 no existe puntero reverso, se continúa en la posición $perm[6] = 5$. Como la posición 5 tiene un puntero reverso asociado ($\mathsf{access}(sampled, 5) = 1$) que es la posición $rev\_links[\mathsf{rank}_1(sampled, 3)] = 8$, se utiliza 8 en el siguiente paso, acortando el ciclo. Finalmente como $perm[8] = 6$, 8 es la respuesta.

2.3.4. k^2-tree

Un k^2-tree [BLNS09] es un árbol de k^2 hijos que permite representar relaciones binarias dispersas a través de una matriz de adyacencia binaria de una manera muy compacta. Una celda $M[i, j]$ de la matriz M tiene un valor 1 si existe una relación entre la fila i y la columna j, y un valor 0 en caso contrario. Originalmente fue diseñada para representar grafos web, el k^2-tree puede ser usado en cualquier dominio que requiera representar una relación binaria como por ejemplo: red social [CL11], grafos de propósito general [ÁGBLP10] o datos RDF [ÁGBFMP11]

La representación del k^2-tree aprovecha la existencia de grandes áreas con ceros de la matriz dispersa para disminuir el espacio de memoria consumido pero, manteniendo

la posibilidad de una navegación eficiente sobre la estructura comprimida. La construcción del k^2-tree requiere que el tamaño de la matriz de adyacencia a representar sea de k^n (para algún entero n $n \geq 2$). Sin embargo, esto no es una restricción fuerte porque si el tamaño de la matriz es D con $k^{n-1} < D < k^n$, es posible agregar a la matriz columnas y filas con ceros con el fin de alcanzar el tamaño apropiado k^n. Estas filas y columnas adicionales no se oponen a una compresión satisfactoria, debido a la capacidad del k^2-tree para almacenar las zonas vacías de la matriz con pocos bits. La construcción del k^2-tree comienza con la subdivisión de la matriz de adyacencia en k^2 submatrices de igual tamaño, siguiendo la estrategia de un MAX-Quadtree [Sam06]. Los nodos del árbol son tuplas de k^2 bits. Cada bit del nodo representa a una de las k^2 submatrices, enumerándolas de izquierda a derecha y de arriba hacia abajo. El valor de cada bit depende del contenido de la submatriz, de modo que será un 1, si la submatriz contiene al menos una celda con un valor 1. En caso contrario, si todas las celdas son 0, es decir, es un área vacía, el valor del bit será un 0. Este proceso es aplicado recursivamente a todas las submatrices no vacías, (representadas por un bit en 1), y la tupla resultante es enlazada como hijo del nodo correspondiente. El proceso termina cuando la submatriz no se puede dividir más porque es una celda y no un área, en este caso la tupla correspondiente es una hoja de último nivel donde cada bit representa el valor de una celda de la matriz.

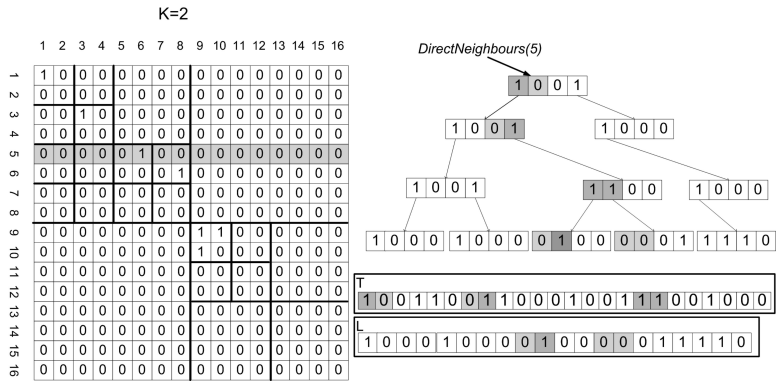


Figura 2.3: Ejemplo de relación binaria representada con un K^2-tree.

En la figura 2.3 se puede observar un ejemplo de la creación de uno de estos árboles para $k = 2$. El tamaño de la matriz es de k^4 (16×16). A la izquierda de la figura se observa la matriz dispersa y a la derecha su correspondiente k^2-tree. El primer 1 en la tupla raíz representa la submatriz superior izquierda de tamaño 8×8, donde se puede observar que al menos una de sus celdas tiene un valor de 1. El segundo bit es un 0, esto significa que la submatriz superior derecha no contiene ninguna celda con un valor de 1; lo mismo ocurre con los siguientes bits de la tupla

raíz. Mediante un recorrido de la estructura, desde la raíz, se puede comprobar que el tercer bit más a la derecha de la hoja es 1 y corresponde al valor que aparece en la coordenada $(10, 9)$ de la matriz dispersa. Note que un k^2-tree probablemente no es un árbol balanceado.

El árbol del k^2-tree con punteros es una representación conceptual. En efecto, este árbol es almacenado de una manera muy compacta usando únicamente dos *bitmaps* llamados T y L. T es un *bitmap* que almacena los bits que no son hojas de último nivel en el k^2-tree, siguiendo un recorrido del árbol por nivel (de izquierda a derecha). L almacena los bits de las hojas de último nivel k^2-tree, de izquierda a derecha también. Como se puede observar en la figura 2.3, T y L almacenan el k^2-tree conceptual completamente.

Un k^2-tree comprime las relaciones binarias de modo compacto usando menos de 4 bits por arco. Lo anterior, permite que grandes bases de datos puedan ser contenidas completamente en la memoria principal, ofreciendo una navegación eficaz a través de ello.

2.3.4.1. Operaciones básicas sobre un k^2-tree

Vecinos directos y reversos La recuperación de los vecinos directos y reversos es una de las operaciones más comunes de una matriz de adyacencia que representa un grafo. Para responder estas consultas se requiere obtener las celdas con un valor de 1 para una fila o columna dada. Estas operaciones son realizadas en un k^2-tree mediante un recorrido en profundidad del árbol.

En la figura 2.3, los bits sombreados en el árbol son aquellos involucrados en la obtención de los vecinos directos en el grafo para el elemento 5 (que corresponde a los valores en 1 que aparecen en la quinta columna de la matriz de adyacencia, la columna sombreada).

En el primer nivel, es decir en el nodo raíz, las dos primeras submatrices están involucradas, dado que ellas contienen celdas que pertenecen a la quinta fila. La primera submatriz es representada por un 1, y por lo tanto sus hijos en el siguiente nivel serán analizados. Por otro lado, el segundo bit que tiene un valor de 0 significa que el elemento 5 no posee vecinos entre 9 y 16. Este proceso continúa en los siguientes nodos internos del árbol hasta llegar a una hoja.

Como se puede observar en la figura 2.3, el décimo bit del último nivel con valor 1, corresponde a la celda $(5, 6)$ de la matriz. La navegación descrita sobre el k^2-tree es eficientemente desarrollada sobre los *bitmaps* T y L utilizando las operaciones de *rank* y *select* lo que requiere contar con una estructura de contadores sobre el bitmap T [BLNS09, CL11].

Como la búsqueda de los vecinos directos consiste en la búsqueda de una columna,

en el peor de los casos, por cada nodo será necesario visitar k hijos de los k^2, esto significa que el tiempo necesario para esta operación es $\mathcal{O}(n)$ [Lad11] . Sin embargo, en el caso promedio como algunos de los k hijos de un nodo no serán visitados por representar áreas completamente en 0, el coste en este caso es de $\mathcal{O}(\sqrt{m})$ tiempo.

Los vecinos reversos sobre el k^2-tree son implementadas siguiendo la misma estrategia de recorrido en profundidad descrita anteriormente pero orientados a obtener una columna. Los tiempos para los vecinos reversos son los mismos que para los vecinos directos.

Recuperar una celda Para determinar si una celda $M[i, j] = 1$ es necesario realizar un recorrido en profundidad, desde la raíz a la hoja. En cada paso se visita solo uno de los k^2 hijos que tiene el nodo visitado y por lo tanto esta operación tiene un coste temporal del orden de la altura del árbol $\mathcal{O}(log_{k^2} n^2) = \mathcal{O}(log_k n)$. Determinar el hijo a visitar es sencillo, será aquel donde su rango en filas contenga a i y su rango en columnas contenga a j.

Consultas por rango Dado un rango de filas $[p1, p2]$ y un rango de columnas $[q1, q2]$, una consulta por rango entrega todos los pares $(i, j) \in [p1, p2] \times [q1, q2]$ donde $M[i, j] = 1$. Esto se podría resolver consultando todos los pares en $[p1, p2] \times [q1, q2]$, sin embargo se puede resolver de un modo más eficiente con un solo recorrido del árbol.

El algoritmo 2.1 corresponde a las consultas por rango. El proceso comienza en la raíz del árbol y termina cuando se han visitado todos los nodos que intersectan con el rango de la consulta. En cada paso se determina cuales de los k^2 hijos que no están en 0 intersectan con la consulta y por cada uno de ellos se hace una llamada recursiva a la consulta por rango, ajustando el rango de la consulta a la intersección del área cubierta por el hijo y el área de la consulta, de modo que se repite el proceso pero con un sub-problema. Al llegar a una hoja de último nivel, se revisa si esta se encuentra en 1 (línea 3), si esto es así, significa que el punto (d_p, d_q) es parte de la salida.

2.3.4.2. Variantes

Existen varias variantes de esta estructura de datos permitiendo la misma funcionalidad antes descrita.

Enfoque Híbrido El k^2-tree explicado anteriormente mantiene un número fijo de hijos para todos los niveles de árbol. Si aumentamos el número de hijos la altura del árbol será menor, mejorando los tiempos de consulta, pero a costa del espacio, dado

Algorithm 2.1 (**Range**)$(n, p_1, p_2, q_1, q_2, d_p, d_q, z)$

```
1: T = k²-tree.T; L = k²-tree.L
2: if z ≥ |T| then
3:     if L[z − |T|] = 1 then
4:         OUTPUT(d_p, d_q)
5:     end if
6: else
7:     if z = −1 ∨ T[z] = 1 then
8:         y = rank(T, z) · k²
9:         for i = ⌊p1/(n/k)⌋ ... ⌊p2/(n/k)⌋ do
10:            p'_1 = 0
11:            if i = ⌊p1/(n/k)⌋ then
12:                p'_1 = p1 mód (n/k)
13:            end if
14:            p'_2 = (n/k) − 1
15:            if i = ⌊p2/(n/k)⌋ then
16:                p'_2 = p2 mód (n/k)
17:            end if
18:            for j = ⌊q1/(n/k)⌋ ... ⌊q2/(n/k)⌋ do
19:                q'_1 = 0
20:                if j = ⌊q1/(n/k)⌋ then
21:                    q'_1 = q1 mód (n/k)
22:                end if
23:                q'_2 = (n/k) − 1
24:                if j = ⌊p2/(n/k)⌋ then
25:                    p'_2 = p2 mód (n/k)
26:                end if
27:                d'_p = d_p + (n/k) · i
28:                d'_q = d_q + (n/k) · j
29:                z' = y + k · i + j
30:                RANGE(n/k, p'_1, p'_2, q'_1, q'_2, d'_p, d'_q, z', output)
31:            end for
32:        end for
33:    end if
34: end if
```

que las hojas de último nivel almacenan todos los bits, tanto 0 como 1, al tener más hijos son más los valores en 0 que se deben almacenar perjudicando la compresión. Una variación al índice consiste en mantener dos valores de K, uno más grande para los niveles superiores, y uno menor para los inferiores, así se consigue disminuir la altura del árbol, y no almacenar tantos unos en las hojas.

Hojas comprimidas Con la idea de mejorar el espacio utilizado por el k^2-tree, otra variante consiste en construir un vocabulario de matrices con las de último nivel. De modo que en vez de almacenar dichas matrices se almacena su código asociado. La compresión se logra cuando existen matrices del vocabulario que tienen una mayor frecuencia que otras.

Compresión de áreas completamente en 1 En un k^2-tree cuando el área de la matriz cubierta por un nodo está completamente en 0, ésta no se sigue subdividiendo. Siguiendo esta misma idea, se ha desarrollado una variante que deja de subdividir aquellos nodos cuya área está enteramente en 1. Como ahora existen dos casos distintos para los cuales la celda no se sigue subdividiendo, es necesario guardar un *bitmap* adicional que contendrá un bit por cada 0 en T para distinguir si la celda está llena de 0 o de 1.

Dinámico Otra varienta es el DK^2-tree [BdBN12] que es una versión dinámica del k^2-tree que alcanza resultados competitivos. En esta estructura es posible cambiar el estado de una celda de la matriz de adyacencia, y dinámicamente se ajustará el árbol. Esto permite agregar o eliminar elementos luego de haber construido el árbol.

Capítulo 3

Estado Del Arte

En todo sistema de base de datos es fundamental contar con un sistema de indexación que permita responder a un variado conjunto de consultas eficientemente, y las bases de datos espacio-temporales no son la excepción.

Desde la aparición de los primeros modelos de bases de datos espacio-temporales a finales de la década de los 80 se han presentado varios sistemas de indexación.

Dado que en esos años, la memoria principal era muy pequeña, las estrategias de diseño para abordar grandes conjuntos de datos necesariamente pasaban por indexación en memoria secundaria. Por esta razón la gran cantidad de índices que se han desarrollado, en su gran mayoría fueron construidos para minimizar los accesos a disco.

La situación actual ha cambiado notablemente. Pasamos de *kilobytes* de memoria principal en la década de los 80 a *gigabytes* en el día de hoy. Esta situación hace posible contener bases de datos completas o bien una parte importante de éstas en memoria principal la cual es varios órdenes de magnitud más rápida que la memoria secundaria. Por esta razón el foco de este capítulo está puesto en describir aquellas estructuras de datos eficientes en espacio que permitan indexar objetos espaciales en memoria principal.

Sin embargo, observar la evolución histórica del campo aporta estrategias de modelado útiles para el diseño de estructuras de datos eficientes en espacio en el ámbito espacio-temporal. Es por ello que se ha considerado oportuno presentar primero (sección 3.1) una breve síntesis de los principales métodos de acceso espacio-temporales destacando la manera de modelar los datos y las ventajas y desventajas de cada método presentado. Seguido de ello se presentan las estructuras de datos compactas, sucintas y/o implícitas que permiten indexar datos espacio-temporales.

3.1. Indexación espacio-temporal clásica

En la literatura encontramos una gran cantidad de propuestas para indexación de datos espacio-temporales. En función del tiempo, es posible distinguir tres tipos de consultas: Consultas sobre el pasado, consultas sobre el momento actual, y consultas sobre el futuro cercano.

Las consultas sobre el pasado o históricas permiten recuperar el estado de la base de datos o de un objeto en algún punto o intervalo temporal pasado. Por otro lado las consultas del momento actual permiten obtener la información de la ubicación actual de un objeto, o de los objetos que se encuentran en la base de datos o en una región del espacio. Por último, las consultas sobre el futuro hacen una predicción respecto de la ubicación que tendrá un objeto en base a la información contenida en la trayectoria actual del objeto (la velocidad, dirección, posición actual del objeto, etc.).

Dependiendo del espacio donde se muevan los objetos podemos distinguir dos tipos de índices: aquellos que indexan objetos que se mueven sobre una infraestructura o red y los que indexan objetos que se mueven por el espacio sin una restricción. Los trenes que se mueven sobre una red ferroviaria, o camiones sobre una red de carreteras son ejemplos de objetos que se mueven sobre una infraestructura; y las aves migratorias son un ejemplo de objetos que se mueven sin una restricción espacial.

En esta tesis, el foco está puesto en las consultas históricas y en aquellos objetos que se mueven sin estar restringidos por una red o infraestructura. Por esta razón la revisión estará centrada en los índices que no restringen el movimiento a un red y que responden consultas históricas omitiendo intencionalmente los otros.

Otra manera de clasificar a los índices espacio-temporales, es dependiendo de la manera que modelan los datos. De esta manera podemos distinguir varios modelos e índices que los implementan, los cuales son descritos en las secciones siguientes, destacando los principales exponentes en cada modelo.

3.1.1. Modelo *Snapshot*

La primera estrategia de indexación espacio-temporal es el modelo *Snapshot* [PTKT04]. En este modelo se captura la evolución de los datos espaciales como si se tratase de una película, que consiste en una secuencia temporal de fotografías del espacio. Cada fotografía consiste en el estado completo de cada objeto espacial en un instante dado, representando en algún índice espacial, como por ejemplo el R-tree, *k*d-tree, QuadTrees, *k*-d-b-Tree, entre otros.

Un índice espacial del tipo *Snapshot*, puede resolver eficientemente las consultas de tipo *time slice*, pero es ineficiente en las consultas de tipo *time interval*. Otra

desventaja de este tipo de indexación es la alta redundancia de datos almacenados cuando los objetos presentan una baja movilidad o evolución temporal. En efecto, para cada objetos que no se ha movido por un largo período de tiempo, en cada instante de dicho período, se copiará la información de su estado en cada uno de los snapshot, lo cual es claramente redundante.

3.1.2. Modelo de *Snapshot* con sobreposición

El HR-Tree [NST99, NST98] y MR-Tree [XHL90] utilizan un R-tree para cada instante de tiempo, pero que son almacenados teniendo en cuenta la sobreposición entre cada uno de ellos con la finalidad de ahorrar espacio. La idea básica es que dado dos árboles el más reciente de ellos corresponde a una evolución del más antiguo y los sub-árboles pueden ser compartidos entre ambos árboles.

Una consulta de tipo *time slice* se realiza ubicando primero el R-tree correspondiente al instante de la consulta, y ejecutando la consulta por rango espacial sobre ese árbol. Este tipo de consulta puede ser resuelta muy eficientemente. Por el contrario, las consultas de tipo *time interval* no lo son, dado que requiere ejecutar una consulta por rango espacial sobre cada R-tree asociado al intervalo de la consulta.

Este tipo de estructura puede degenerar fácilmente en un R-tree por instante dado que si varios objetos se mueven de un instante a otro, no existirán sub-árboles comunes que compartir. Cabe recordar que en un R-tree los objetos espaciales se almacenan en las hojas y, por lo tanto, un cambio afecta a todos los nodos que estén en el camino desde la raíz hasta la hoja que almacena al objeto modificado.

Una versión mejorada del HR-Tree es el HR$^+$-Tree [TP01c], el cual permite que en un mismo nodo se almacenen entradas ocurridas en distintos instantes de tiempo. Este cambio permite un ahorro de espacio. Además, las consultas de tipo *time interval* son procesadas de manera más eficiente, al evitar la visita de sub-árboles que han sido visitados en instantes previos.

3.1.3. Modelo de *Snapshot* + Eventos

En un modelo de datos espacio-temporales, los cambios de estado pueden ocurrir por diversas razones. Para modelar las razones por las que ocurren los cambios se utilza el concepto de Eventos [Wor05, GW05].

El SEST-Index [GNR$^+$05] es una estructura que mantiene, para algunos instantes de tiempo, un *snapshot* junto con bitácoras de eventos entre los *snapshot*. Los eventos aportan más semántica que un simple cambio de posición del objeto, por ejemplo pueden indicar cuando un objeto ha entrado o salido de un determinado lugar, o

si un objeto ha colisionado con otro, etc., y por ende esta estructura es también adecuada para responder consultas respecto de los eventos.

3.1.4. Modelado del tiempo como otra dimensión

Otro modelo o estrategia para indexar datos espacio-temporal consiste en tratar al tiempo como una dimensión adicional a las dimensiones necesarias para representar la ubicación del objeto en el espacio y utilizar un índice multi-dimensional para indexar estos datos.

3DR-Tree [TVS96] es un ejemplo de dicha estrategia, donde el índice multi-dimensional utilizado es un R-tree. En esta estructura un objeto en una ubicación determinada durante un intervalo temporal es representado por un segmento de línea. Por ejemplo si un objeto se encuentra en la posición (x, y) durante el intervalo $[t_1, t_2)$, el segmento almacenado en el R-tree corresponde a $[\,\overline{(x, y, t_1), (x, y, t_2)}\,)$.

Una consulta espacio-temporal en este contexto se modela como un cubo (o hipercubo), definido por un rango de valores para cada dimensión. Luego se realiza una consulta por rango en el R-tree con el cubo que define la consulta. La respuesta serán todos los segmentos que intersectan con la consulta.

El 3DR-Tree es eficiente en el procesamiento de consultas de intervalo. Sin embargo, ineficiente para el procesamiento de consultas del tipo timestamp [TPZ02]. Otra desventaja de este índice es que solo permite indexación de información histórica, debido a que es necesario conocer de antemano los intervalos en los cuales un objeto a estado en una determinada posición.

Este último problema es abordado por el 2+3 R-tree, el cual consiste en dos índices R-tree, uno 2D para mantener la ubicación actual de los objetos un 3DR-Tree para almacenar la información histórica de los movimientos. Cuando un objeto se mueve, se debe realizar primero una búsqueda de la ubicación que se encuentra registrada en el 2D R-tree, rescatando con ello también el instante cuando ocurrió. Luego con esta información, más el instante de la nueva posición del objeto se conforma el segmento correspondiente y se inserta en el 3DR-Tree. Luego se actualiza el 2D R-tree con los nuevos valores.

La principal desventaja del 2+3 R-tree es que las consultas se deben hacer sobre dos índices, lo que impacta en el rendimiento si lo comparamos con la versión histórica.

3.1.5. Modificación de los nodos del R-tree para incorporar el tiempo

El RT-Tree [XHL90] es una estructura con un enfoque diferente al anterior. Toma como base un R-tree, al cual se modifica la estructura del nodo para almacenar junto con la información espacial, el intervalo temporal en el cual dicha información es válida. Al momento de consultar sobre el RT-Tree, se va descendiendo por aquellos nodos que intersectan tanto espacial como temporalmente. En este tipo de estructura la información temporal juega un rol secundario debido a que la consulta es dirigida por la información espacial. Por lo anterior, las consultas con condiciones temporales no son eficientemente procesadas [NST98].

3.1.6. Basado en Multiversión

El MVR-Tree [TP01b,TP01a,TPZ02] es una estructura basada en la manipulación de múltiples versiones. Este es una extensión del MVB-Tree [BGO+96], donde el atributo que varía en el tiempo corresponde al espacial.

Un MVR-Tree al igual que un MVB-Tree posee varios nodos raíz, cada uno de los cuales corresponde a una versión del índice en un intervalo temporal dado. Así, una consulta que busque en una versión en particular del índice es respondida recorriendo el árbol cuyo nodo raíz está temporalmente contenido en el intervalo de la consulta [RVM06].

En un MVR-Tree cada entrada tiene la forma $\langle S, ti, te, ref \rangle$, donde ti indica el instante en que el registro fue insertado y te el instante de tiempo en que el objeto se eliminó. Para los nodos hojas, S se refiere al MBR (*minimum bounding rectangle*) de un objeto que cambia en el tiempo y para los nodos intermedios, la entrada S determina el MBR que incluye espacialmente a todas las entradas entre $[ti, te)$ en el sub-árbol apuntado por ref . El atributo ref, en el caso de los nodos hojas, es un puntero que apunta a un registro o tupla con la información del objeto [GNR+05].

Cuando se inserta una nueva entrada en el instante de tiempo t, el atributo ti se fija al valor t y el atributo te a "*" (para indicar el valor del tiempo actual - NOW). Cuando una entrada se elimina lógicamente en un instante t, el valor de te se actualiza con t. De esta forma las entradas vigentes tienen "*" como valor en te y en otro caso se consideran como entradas no vigentes ("muertas") [GNR+05].

Otro índice mulversión es el *Multiversion Linear Quadtree* (MVLQ) [TVM01]. Este índice también toma como base un MVB-Tree, pero en vez de almacenar como clave un MBR almacena un código de localización el cual corresponde al camino que se debe recorrer desde la raíz a una hoja del Quadtree que particiona el espacio. Note que el Quadtree queda implícito en los códigos de localización y no es necesario

almacenarlo.

De las estructuras presentadas, el MVR-Tree y su variante mejorada MV3R-Tree tiene un mejor desempeño que los demás métodos de acceso espacio-temporal previamente creados en términos de coste temporal para las consultas de tipo *timestamp* e intervalo.

3.1.7. Modelo basado en Trayectorias

Hasta ahora los índices presentados se concentran en responder consultas por rango espacial, tanto del tipo *Time Slice* como *Time Interval*. Pero existe otro grupo de índices donde el foco está en responder consultas orientada a la trayectoria de los objetos y por lo tanto la cercanía de los objetos es de menor relevancia, lo que impacta en el diseño del índice.

Las consultas de tipo trayectoria, son aquellas que requieren evaluar toda la trayectoria de un objeto o bien una parte de ella para poder ser respondida [Fre08]. Estas consultas pueden considerar las relaciones topológicas entre los objetos espacio-temporales como por ejemplo: entra, sale, toca, etc, así como aspectos navegacionales del objeto: velocidad actual, velocidad promedio, distancia recorrida, etc.

El *Trajectory-Bundle Tree* (TB-tree) es una extensión del R-tree que permite almacenar trayectorias. Un nodo hoja del árbol únicamente contiene segmentos que pertenecen a una misma trayectoria. Esto trae como desventaja que dos segmentos de diferentes trayectorias que son espacialmente cercanos se deben almacenar en diferentes nodos.

SETI [CEP03] es un índice de trayectoria que tiene una estructura híbrida, que separa la parte espacial de la temporal.

El índice asume que los objetos se mueven muy frecuentemente sobre un espacio fijo. De este modo, la componente espacial corresponde a un conjunto de particiones estáticas, y para cada una de ellas existe un índice de segmentos temporales que representan la permanencia de los objetos en las diferentes posiciones a lo largo de sus trayectorias. Si el segmento de trayectoria intersecta dos o más particiones, este se divide en varios segmentos y se insertan en el índice correspondiente.

Como señala Gutierrez [GNR$^+$05], la gran limitación de SETI es que las particiones espaciales son fijas y por lo tanto, si son muy grandes, la componente espacial discrimina muy poco lo que afecta el rendimiento de las consultas. Por otro lado, si las particiones son muy pequeñas habrán muchos segmentos que se intersectarán con varias particiones, lo que aumenta el almacenamiento y el rendimiento de las consultas. Por otro lado, si los segmentos se concentran en una zona específica del espacio, habrá un desbalance en la carga de los índices temporales.

Otro índice para trayectorias es el PA-tree [NR05]. Este índice, utiliza una aproximación de la trayectoria de un objeto basada en polinomio. Los autores a través de experimentos demuestran que su método basado en polinomio es mejor que una aproximación basada en MBR.

El índice PA-tree, posee una estructura de dos niveles. En el primer nivel utiliza una estructura similar a un R*-tree la cual indexa los 2 coeficientes principales del polinomio que describe el movimiento del objeto en cada dimensión y el valor máximo del error de desviación. En el segundo nivel, se almacenan más coeficientes, con la finalidad de proveer una mejor aproximación en la etapa de filtrado.

3.2. Compresión de trayectorias

Debido a que los actuales sistemas de posicionamiento permiten la captura de grandes volúmenes de información espacio-temporal se hace evidente la necesidad de contar con alguna estrategia para minimizar el coste de almacenamiento en las bases de datos espacio-temporales.

Hasta ahora la mayoría de los esfuerzos para minimizar el coste del almacenamiento no se ha puesto en los sistemas de indexación, sino en la compresión de trayectorias.

Las soluciones presentadas en la literatura se pueden categorizar como:

- Utilización de técnicas de compresión de datos: LZW, DEFLATE, LZMA y códigos aritméticos usando PPM [Koe13]
- Simplificación de trayectorias mediante: algoritmo de Douglas-Peucker, OPW, OPW-TR y Dead Reckoning [MOH+13] entre otros.
- La compresión semántica de trayectorias [SRL09] usando para ello la información espacial contenida en la infraestructura o red por donde se mueven los objetos y modelando las trayectorias como un camino dentro del grafo que modela la red.

Una completa revisión de las técnicas y algoritmos para la compresión de trayectorias la encontramos en el trabajo de Sun [SXYL16]

Un trabajo interesante es el de Koegel [Koe13, KKKM10, KBMS11, KRHM12, KM12]. Él plantea una técnica de compresión de trayectorias donde utiliza un modelo predictivo basado en las dos últimas posiciones del objeto, luego codifica mediante códigos aritméticos la diferencia entre la posición real del objeto y la que fue predecida por el modelo. Por medio de esta técnica es posible lograr una

disminución significativamente mayor del coste de almacenamiento que al utilizar técnicas de simplificación de trayectorias con un bajo nivel de error [Koe13].

Sin embargo las técnicas de compresión clásicas y en particular los códigos aritméticos no permiten responder a las consultas espacio temporales sin descomprimir los datos previamente. Dicha situación hace que el uso de técnicas de compresión tradicionales como las presentadas por Koegel no sean factibles de aplicar en un sistema de indexación espacio-temporal directamente debido al sobrecoste que supone la descompresión de los datos.

Por otro lado, las técnicas de simplificación de trayectorias pueden ser aplicadas a cualquier sistema de indexación al pre-procesar los datos antes de ser agregados a la base de datos.

3.3. Estructuras de datos eficientes en espacio para datos espacio-temporales

En la literatura encontramos una única propuesta para indexar datos de objetos móviles de forma compacta denominada *Compact Trip Representation*(CTR). Este índice permite la indexación de trayectorias de objetos que se mueven sobre una red [BFGR16] y puede responder eficientemente a consultas de rango a la vez que es eficiente en el espacio utilizado. Como la indexación de objetos que se mueven sobre una red no es el foco de esta tesis, el índice CTR no será estudiado en mayor profundidad.

En el ámbito de la indexación Espacio-Temporal, en el mejor de nuestro conocimiento, no existe ninguna estructura de datos que aborde el tema de indexar datos espacio-temporales de objetos que se mueven en un espacio sin restricciones, que es el foco de la tesis.

No obstante, sí existen estructuras de datos eficientes en espacio para indexar objetos espaciales.

Como se presentó en la sección 3.1 existen dos maneras clásicas de modelar datos espacio-temporales utilizando índices espaciales: el modelo de *snapshot* y modelar el tiempo como una dimensión más. Ambas técnicas pueden ser empleadas perfectamente con los índices espaciales eficientes en espacio existentes.

Por esta razón en las siguientes secciones se presentan aquellos índices espaciales presentes en la literatura que son eficientes en espacio como parte del estado del arte.

3.3.1. kd-tree implícito

Un kd-tree es una estructura de datos que permite la indexación de puntos de un espacio de k dimensiones. Un kd-tree es un árbol binario donde cada nodo guarda un punto el cual se utiliza para particionar el espacio en dos, mediante un hiperplano perpendicular a uno de los ejes (dimensión). Por ejemplo, si el punto almacenado en un nodo es (3,5) y el hiperplano que divide el espacio es perpendicular al eje y, entonces en el sub-árbol izquierdo estarán todos los puntos cuya coordenada y es menores que 5 y en el sub-árbol derecho los demás puntos.

En cada nivel del árbol se particionará una dimensión diferente repitiendo cíclicamente las dimensiones a medida que se desciende por el árbol. Por ejemplo, para un espacio de tres dimensiones, en la raíz el punto almacenado particionará el espacio en dos planos perpendiculares al eje x, en sus hijos al eje y, en sus nietos al eje z, y luego volverá a particionar el eje x, repitiendo el ciclo hasta llegar a las hojas.

Un kd-tree es sensible al orden en que se se insertan los puntos. Sin embargo, si de antemano conocemos cuales son todos los puntos que contendrá el índice es posible construir un kd-tree balanceado.

Como señala [He13] la representación implícita de un kd-tree por Munro [Mun79] es la primera estructura de datos eficiente en espacio que es útil para la indexación de puntos.

La construcción de un kd-tree implícito, comienza con una *array* $C[0..n-1]$ de n puntos, los cuales son re-ordenados con la finalidad interpretar el arreglo como un árbol binario balanceado (kd-tree).

El algoritmo de construcción sigue la estrategia de dividir para reinar. Inicialmente el arreglo C es considerado como un solo segmento S, el cual es la entrada al algoritmo de construcción.

En cada etapa se divide S en dos mitades, las cuales son re-ordenadas convenientemente siguiendo los siguientes pasos.

1. Buscar el punto m cuya dimensión j con $j = i \mod d$ corresponde a la mediana de todos los puntos en dicha dimensión.

2. Intercambiar el punto m con el elemento que está en la mitad de S.

3. Mover todos los puntos de S cuya coordenada en la dimensión j son menores que la coordenada correspondiente en m a la primera mitad de S y los mayores a la segunda mitad.

4. Tanto la primera mitad como la segunda pasan a ser un segmento a procesar.

El proceso de particionado de los segmentos termina luego de $\lceil \lg n \rceil$ pasos cuando todos los segmentos restantes no se pueden seguir dividiendo porque tienen un único punto.

Al terminar este proceso el arreglo C codifica el kd-tree de manera implícita.

Un kd-tree implícito se puede construir en tiempo $\mathcal{O}(n \log n)$ [Mun79].

Una consulta por rango en un kd-tree implícito comienza con el segmento que en la primera dimensión contiene todos los puntos y el nodo raíz que se encuentra en la mitad del arreglo C. Cada vez que se visita un nodo se revisa si el segmento que representa ese nodo se encuentra completamente contenido en el rango de la consulta. Si es así, entonces se reportan todos los puntos contenidos en ese segmento. En caso contrario, es necesario explorar los sub-árboles izquierdo y derecho recursivamente, si sus respectivos segmentos intersectan con el rango de la consulta.

La consulta por rango sobre un conjunto de n puntos puede ser resuelta en un tiempo $\mathcal{O}(n^{1-1/d} + k)$, donde k es el número de puntos reportados [Mun79] .

Como un kd-tree es un índice multidimensional, es posible utilizarlo para indexar puntos móviles siguiendo tanto el modelo de *snapshot* como modelando el tiempo como una dimensión adicional.

3.3.2. Wavelet Tree

Un *grid* de dos dimensiones es una matriz de c columnas y f filas, donde cada celda puede o no contener un punto.

Un Wavelet Tree permite indexar n puntos que se encuentran dentro de un *grid*, pero con una restricción importante, debe existir exactamente un punto por columna.

Por lo tanto es necesario contar con una estrategia para mapear los puntos desde su espacio original al espacio que puede ser indexado por el wavelet tree. En [Nav16] encontramos una propuesta para realizar dicho mapeo, la cual se explica a continuación:

Primero hay que ordenar los puntos por el valor de la coordenada x de modo que se obtenga una secuencia de puntos $P = \langle (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \rangle$, donde se cumple que para todo i, $x_i \leq x_{i+1}$. Usando P se crea un nuevo *grid* donde un punto (x_i, y_i) del *grid* original es representado por (i, y_i) en el nuevo *grid*, note que i es la posición del punto en P.

Una vez que se han convertido todos los puntos a el nuevo *grid*, es posible interpretar P como una secuencia $S[1, n]$, donde $S[i] = y_i$. La secuencia S se indexa

con un Wavelet Tree, donde el alfabeto de la secuencia son los valores que están en el rango $[1..f]$.

Para modelar el mapeo desde el *grid* original al nuevo *grid* se utiliza un *bitmap* B, el cual se construye poniendo un 1 por cada columna del *grid* original seguido por tantos ceros como puntos se encuentren en ella, formalmente $B[1, c+n] = 10^{n_1}10^{n_2}10^{n_3}\ldots10^{n_c}$, donde cada n_v es el número de puntos con $x_i = v$. Cabe señalar que en aquellas columnas donde no exista un punto, existirá un 1 en B pero que no es seguido por 0 alguno.

Por ejemplo, en la figura 3.1 se muestra el *bitmap* B resultante de mapear el conjunto de puntos desde el *grid* original al nuevo *grid*.

Gracias al *bitmap* B es posible devolver un punto (x, y) desde el nuevo *grid* al *grid* original, para ello se calcula la coordenada x original como: $\mathsf{select}_0(B, x) - x$, manteniendo el valor de la coordenada y, puesto que es el mismo en ambos *grid*.

De este modo la estructura que indexa el conjunto de puntos original está compuesta por B y S indexada con un Wavelet Tree. Si B es un *bitmap* comprimido, el espacio de S y B suman hasta $n \log f + o(n \log f) + n \log \frac{c+n}{n} + o(c+n) = n \log \frac{cf}{n} + \mathcal{O}(n) + o(c + n \log f)$ [Nav16].

Otra estrategia de mapeo encontramos en [Sec09,SNL09], la cual permite mapear un conjunto de puntos en $\mathbb{R}^2$ con un wavelet tree. Pero a diferencia de la estrategia propusta en [Nav16] y aquí descrita, requiere almacenar los puntos, lo que supone un coste adicional. Sin embargo, permite almacenar puntos decimales en coma flotante, lo cual puede ser importante en algunos dominios.

Independiente de la estrategia usada para mapear los puntos, un wavelet tree permite la indexación espacial de puntos en dos dimensiones y por lo tanto se podría aplicar en una indexación espacio-temporal siguiendo el modelo de *snapshot*. Pero como no es multidimensional, no es posible utilizar la estrategia de modelar el tiempo como otra dimensión.

Consulta por rango

Una vez que se tiene construida la estructura es posible realizar consultas por rango directamente sobre el wavelet tree, pero antes hay que mapear el rango original a su equivalente en el nuevo *grid*. Un rango de consulta $[x_1, x_2] \times [y_1, y_2]$ se mapea a la nueva grid como $[\mathsf{select}_1(B, x_1) - x_1 + 1, \mathsf{select}_1(B, x_2 + 1) - (x_2 + 1)] \times [y_1, y_2]$ [Nav16].

Siguiendo con el ejemplo de la figura 3.1, en ella podemos ver que el rango original $[x_1, x_2] \times [y_1, y_2] = [4, 9] \times [8, 12]$ corresponde al rango $[8, 14] \times [8, 12]$ en la nueva grid. Para llevar el rango original al nuevo *grid* se calculó: $[\mathsf{select}_1(B, x_1) -$

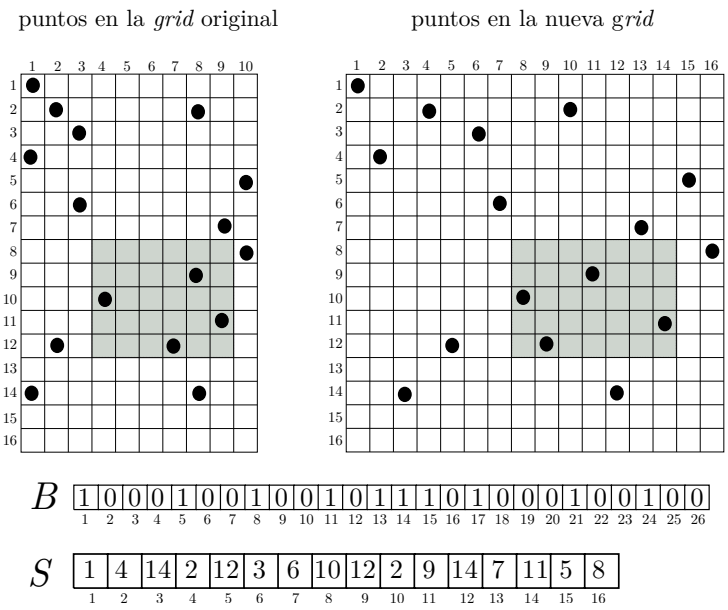


Figura 3.1: Ejemplo de mapeo de puntos desde la grid original a la grid que realmente se indexa con el wavelet tree, por medio del *bitmap* B.

$x_1 + 1, \mathsf{select}_1(B, x_2 + 1) - (x_2 + 1)] \times [y_1, y_2] = [\mathsf{select}_1(B, 4) - 4 + 1, \mathsf{select}_1(B, 9 + 1) - (9 + 1)] \times [8, 12] = [11 - 4 + 1, 24 - 10] \times [8, 12] = [8, 14] \times [8, 12]$.

Luego que se ha convertido el rango al espacio del nuevo *grid*, se resuelve la consulta por rango partiendo desde el nodo raíz.

El algoritmo que describe el procedimiento para la consulta por rango aplicado sobre un wavelet matrix, que es un tipo de wavelet tree, se encuentra en [Nav16], junto con el análisis y la explicación de los algoritmos. A continuación se describe dicho procedimiento de manera general siguiendo el ejemplo de la figura 3.2.

Conceptualmente, todo nodo v de un wavelet tree tiene asociado un *bitmap* con un bit por cada valor de S que se encuentra en el rango $[a, b]$ del eje y manteniendo el orden en S, es decir, el primer bit del nodo v corresponde al primer valor en S que se encuentra en el rango $[a, b]$, el segundo bit al segundo valor en S y así sucesivamente. Si un bit está en 0 significa que el valor asociado en S se encuentra en la primera mitad del rango $[a, b]$, si está en 1 significa que está en la segunda mitad. En nuestro ejemplo, el rango asociado para la raíz es $[1, 16]$ que es el rango total de los valores del eje x, y por lo tanto el *bitmap* de la raíz posee un bit para cada valor de S, donde los bits en 0 indican que el valor asociado está en el rango

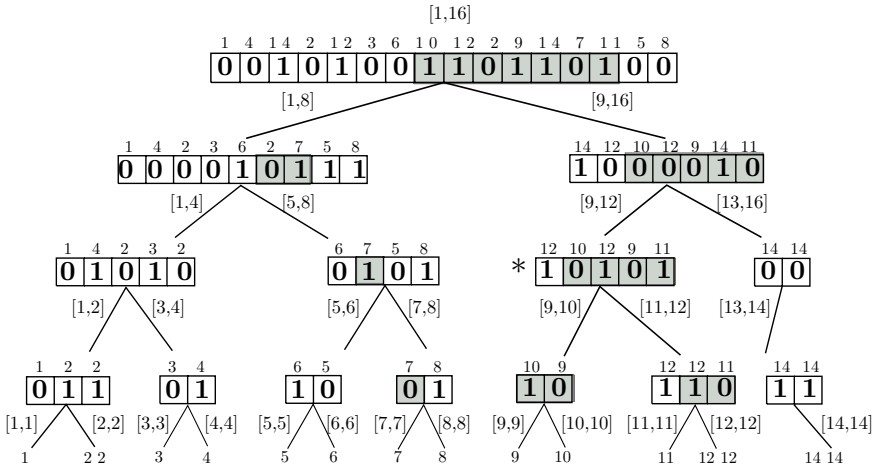


Figura 3.2: Wavelet Tree que codifica S (ver figura 3.1). En plomo aparece la proyección del rango en el eje x sobre los nodos visitados al resolver una consulta por rango.

$[1, 8]$ (primera mitad) y los bit en 1 que el valor asociado está en el rango $[9, 16]$ (segunda mitad).

En general, en cada paso se desea saber cuáles son los puntos que se encuentran en la sub-secuencia $S_v[x_1, x_2]$ cuyo valor para la coordenada y se encuentra en el rango $[y_1, y_2]$. Dado lo anterior, existen 4 casos: 1) que $x_1 > x_2$, y por lo tanto no existen puntos que recuperar. 2) $[a, b] \cap [y_1, y_2] = \emptyset$ tampoco hay puntos para recuperar. 3) $[a, b] \subset [y_1, y_2]$, en este caso por cada bit en el rango $[x_1, x_2]$ se debe recuperar tanto su coordenada x asociada (subiendo desde el nodo v hasta la raíz); como la coordenada y (bajando hasta la hoja correspondiente). 4) ninguno de los anteriores, en cuyo caso se debe procesar recursivamente ambos hijos, ajustando correctamente el rango en x.

Siguiendo con el ejemplo, partimos desde la raíz, consultando por el rango $[8, 14] \times [8, 12]$, como el rango del nodo raiz es $[a, b] = [1, 16]$, el caso corresponde al 4, y por lo tanto se debe continuar recursivamente con ambos hijos. Como en el hijo izquierdo van todos los valores cuyo bit está en 0, el rango en x ajustado para el hijo izquierdo corresponde a $[6, 7]$, don 6 significa que el primer 0 que aparece en el rango $[8, 14]$ es el sexto del *bitmap* del nodo raíz; y 7 porque el último cero del rango original es el séptimo. El rango en x del hijo izquierdo se calcula así: $[\mathsf{rank}_0(B_l, x_1 - 1) + 1, \mathsf{rank}_0(B_l, x_2)]$ [Nav16]

Luego se visita el hijo izquierdo, consultando el rango $[6, 7] \times [8, 12]$, como

$[a, b] = [1, 8]$ en este nodo, nuevamente estamos en el caso 4. Calculamos el rango en x para decender por el hijo izquierdo y nos da $[5, 5]$. Al visitar el nodo podemos observar que la intersección entre $[a, b] = [1, 4]$ y el rango en y de la consulta es $\emptyset$, por lo tanto, no hay resultados en esta rama y se termina la llamada, continuando con el hijo derecho del nodo con $[a, b] = [1, 8]$.

Nuevamente hay que ajustar el intervalo en x pero esta vez se hace con los valores en 1 (usando rank_1 en vez de rank_0), quedando el intervalo en $[2, 2]$. Esta rama se procesa hasta la hoja, y como el valor encontrado es un 7 se descarta, porque no se encuentra en el rango de $[y_1, y_2] = [8, 12]$.

Después, se vuelve al nodo raíz para procesar el hijo derecho que estaba pendiente. Se calcula el rango en x para el hijo derecho, el cual corresponde a $[3, 7]$, como para dicho nodo $[a, b] = [9, 16]$ se deben procesar ambos hijos.

Para el caso del hijo izquierdo (marcado con *) $[a, b] = [9, 12]$ y como $[9, 12] \subset [8, 12]$, se aplica el caso 3, es decir hay que recuperar el par (x, y) asociado a cada bit en el rango $[x_1, x_2] = [2, 5]$, es decir los pares $(8, 10), (9, 12), (11, 9)$ y $(14, 11)$, que son los puntos asociados a los bits 2,3,4,5 del nodo visitado. Para encontrar la coordenada x del bit 2 hay que buscar la posición del segundo 0 en el *bitmap* del padre, dado que el nodo visitado es hijo izquierdo. Esto se hace con una operación de select sobre el *bitmap* del padre B_p: $3 = \mathsf{select}_0(B_p, 2)$. Se sube al padre y como este nodo es hijo derecho de su padre, se busca la posición del tercer 1 en el nodo raíz, la cual es 8. Como llegamos a la raíz, significa que 8 es el valor de la coordenada x del punto asociado al bit 2 del nodo marcado con *.

Para encontrar la coordenada y se sigue la misma idea, pero descendiendo hacia las hojas usando la operación de rank_0 si se baja por el hijo izquierdo y rank_1 si se baja por el derecho.

Siguiendo con el ejemplo, para encontrar la coordenada y del bit 2, como $B_v[2] = 0$, hay que bajar por el hijo izquierdo, entones calculamos $i = \mathsf{rank}_0(B_v, 2) = 1$. Luego en el hijo izquierdo hay que revisar el valor asociado al primer bit. Como el valor asociado al primer bit es 1, significa que se debe bajar por el hijo derecho, al que le corresponde $[a, b] = [10, 10]$, como $a = b$, se ha obtenido el valor para la coordenada y que es 10.

El proceso antes descrito para el segundo bit del nodo * se repite para el tercero, cuarto y quinto.

Terminado lo anterior se continúa con la última llamada pendiente que es el hijo derecho del nodo con $[a, b] = [9, 16]$, llamada que termina sin aportar más resultados.

El coste de esta operación es de $\mathcal{O}(log_2 f)$ tiempo por cada punto reportado [Nav16].

3.3.3. Basado en k^2-tree

Como se describió en la sección 2.3.4, el k^2-tree es una estructura de tipo árbol que representa de manera compacta una matriz de adyacencia binaria, lo cual se puede aplicar a en la indexación espacial de puntos. En efecto, el espacio $\mathbb{N}^2$ puede ser visto como una matriz de adyacencia binaria donde las columnas corresponden al eje x, las filas al eje y y cada celda (x, y) tiene un valor 1 si existe uno o más objetos en esa posición, o un 0 si no los hay. Si el rango de la matriz de adyacencia que representa el espacio es de $n \times n$, el k^2-tree será un árbol de k^2 hijos y de altura $\lceil log_{k^2} n^2 \rceil$.

Cada nivel del árbol puede ser visto como un nivel de granularidad de la partición del espacio. En la raíz, el nodo representa todo el espacio. Los nodos intermedios representan una porción del espacio de su padre el que es particionado en k^2 celdas cuadradas e iguales, una por cada hijo. A cada nodo se le asigna un valor binario que es 1 si en la celda asociada hay uno o más objetos y 0 si la celda está vacía. En el k^2-tree los nodos en 0 no se siguen subdividiendo lo que permite ahorrar espacio. En las hojas de último nivel el espacio representado es tan pequeño como se requiera según la precisión de los datos (kilómetros, metros, centímetros, etc.). Cabe destacar que el tamaño de la celda más pequeña depende de la granularidad de los datos y no de una limitación del sistema, pues el k^2-tree se puede ajustar a distintos niveles de granularidad ajustando la altura del árbol.

Como se comentó en la sección 2.3.4, es posible responder a consultas por rango, requisito fundamental para poder usar la estructura en el ámbito espacial.

Al igual que pasa con el wavelet tree, como la estructura es para 2 dimensiones, en el caso de datos espacio-temporales es posible utilizarla con un modelo basado en *snapshot*.

Sin embargo se han definido variantes del k^2-tree para manejar datos multidimensionales. A continuación se describen dichas variantes.

Interleaved k^2-tree

El Interleaved k^2-tree (ik^2-tree) es una extensión del k^2-tree diseñada para manipular datos en 3 dimensiones donde una de ellas presenta una distribución *skewed* [Alv14, dB14, BBN14].

La idea general de esta estructura es dividir la dimensión *skewed* en m planos, uno para cada valor posible en dicha dimensión representados conceptualmente por m k^2-tree. Pero, en vez de representar estos m k^2-tree de manera separada se mezclan en uno solo, de modo que cada partición del espacio se representan con m bits en lugar de 1.

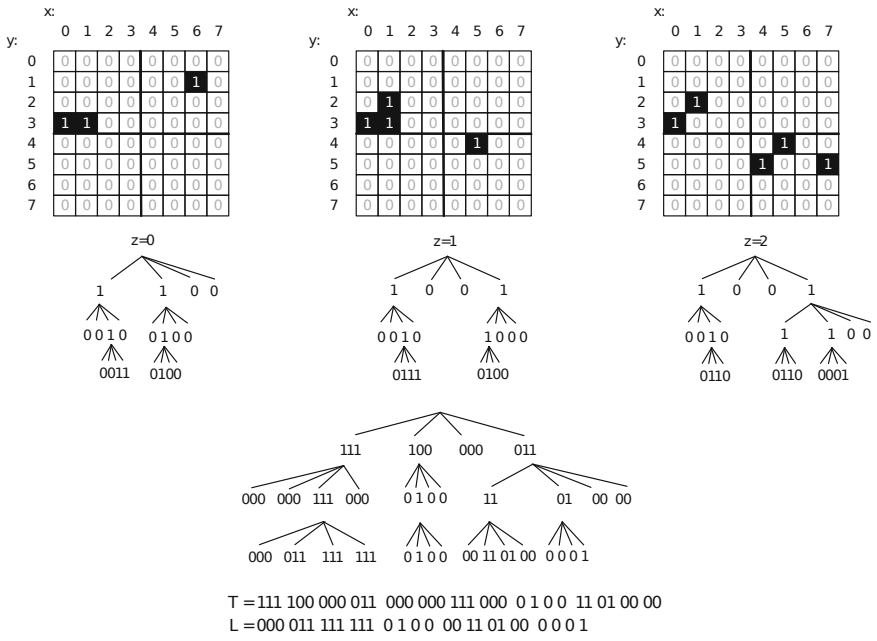


Figura 3.3: Ejemplo de un ik^2-tree para un espacio donde la dimensión Z está en el rango [0.,2] [CRBF15]

Por ejemplo en la figura 3.3 se presenta un espacio 3D $X \times Y \times Z$, donde Z es la dimensión *skewed* con valores en el rango $[0, 2]$. En ella se pueden apreciar los tres planos que se obtienen para los valores $z = 0$, $z = 1$, $z = 2$. Debajo de cada matriz se ve el k^2-tree que los representa. En la parte inferior de la imagen aparece el ik^2-tree que combina a los tres k^2-tree anteriores en uno solo.

Esta estructura no es útil en un dominio espacio temporal, dado que en general, no existe una dimensión *skewed*, por ejemplo para una hora de puntos móviles que se mueven a cada segundo, se requieren 3.600 bits por cada nodo del ik^2-tree.

Una estructura que no tiene este problema es la presentada a continuación.

k^d-tree

El k^d-tree (o k^n-tree) es una generalización del k^2-tree para abordar problemas en d dimensiones [dB14]. Al igual que en un k^2-tree, en un k^d-tree cada una de las d dimensión del espacio es dividida en $k - 1$ hiperplanos los que forman k^d particiones

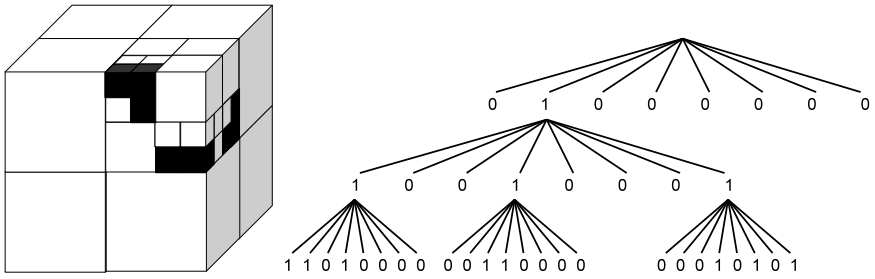


Figura 3.4: Ejemplo de un k^d-tree con $d = 3$ y $k = 2$ [dB14]

de igual tamaño. Cada partición es representada en el árbol con un bit, el cual es 0 cuando la partición está vacía y un 1 si en ella hay por lo menos un punto. Recursivamente, por cada partición no vacía, se repite el proceso subdividiendo la partición del espacio hasta llegar a particiones que no se pueden dividir más. En el árbol, un nodo representa a una partición en particular y sus hijos a las k^d sub-particiones que le corresponden.

En la figura 3.4 se presenta un ejemplo para un espacio de 3 dimensiones ($d = 3$) con $k = 2$, esto significa que el espacio será dividido por tres hiperplanos, uno para cada dimensión obteniendo 8 particiones del espacio ($k^d = 2^3 = 8$). Cada cubo de color blanco es un cubo vacío, en cambio los de color negro representan aquellos cubos que por lo menos tiene un punto. A la derecha de la figura se puede observar el k^3-tree (conceptual) que representa el espacio. En la raíz del nodo observamos 8 hijos con los valores 0 1 0 0 0 0 0 0 . Como el único cubo que tiene datos es el segundo, es el único de los cubos del primer nivel que se subdivide. Al subdividir se vuelven a obtener 8 hijos con valores 1 0 0 1 0 0 0 1, y por lo tanto se subdividen tres de estos 8 cubos. Finalmente se muestra el valor asociado a cada una de estas sub-divisiones.

Como se puede observar en la figura, el efecto de aumentar la cantidad de dimensiones, impacta en la cantidad de hijos por nodo nada más. Por esta razón, la estructura de datos para representar un k^d-tree es la misma que para el k^2-tree, es decir, se mantiene dos *bitmaps* T y L para almacenar los bits contenidos en los nodos del árbol conceptual luego de recorrerlo en anchura, separando los bits contenidos en los nodos internos, que se almacenan en T, de los contenidos en las hojas, los cuales se almacenan en L. Para el caso de la figura 3.4, el k^d-tree resultante es:

```
T:  01000000 10010001
L:  11010000 00110000 00010101
```

Las operaciones que se pueden hacer sobre esta estructura, son las mismas que se

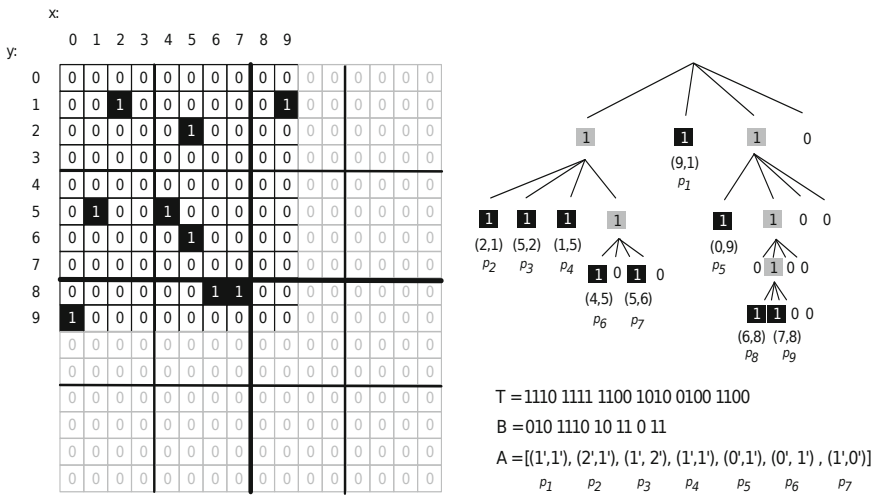


Figura 3.5: Ejemplo de un $c\mathsf{k}^\mathsf{d}$-tree con $d = 2$ y $k = 2$ [CRBF15]

pueden hacer sobre el k^2-tree, pero extendidas a d dimensiones, las cuales se pueden obtener por medio de consultas por rango, donde se impone una restricción de rango respecto de una, varias o todas las dimensiones. Por ejemplo, si tenemos un conjunto de puntos móviles, se podría obtener todos aquellos puntos que se encuentran en una región del espacio $[x_1, x_2] \times [y_1, y_2]$ en los instantes $[t_1, t_2]$ con el rango tridimensional $[x_1, x_2] \times [y_1, y_2] \times [t_1, t_2]$.

El algoritmo para las consultas por rango espacial en un k^d-tree generaliza el algoritmo presentado en 2.3.4 para k^2-tree, teniendo en cuenta las demás dimensiones, pero en términos generales es la misma idea básica: Una consulta por rango comienza desde la raíz y va descendiendo hacia las hojas únicamente por los hijos que se encuentren dentro del rango multidimensional que define la consulta y que por supuesto no sean 0. Cuando se llega a las hojas, todas aquellas que se encuentran en 1 son reportadas por la consulta por rango. Dado que el particionado del espacio es fijo, al igual que en k^2-tree, durante el descenso por el árbol es fácil determinar el rango espacial asociado a una partición y, al llegar a una hoja, conocer las coordenadas de la celda asociada.

k^d-tree multidimensional comprimido

Cuando se representan espacios que son poco densos con un k^d-tree existirán muchos ceros en los nodos que podrán ser omitidos dada la alta aridad de este tipo de árbol. Para ilustrar este problema suponga un k^d-tree, donde $k = 2$ y $d = 2$,

el espacio es de tamaño $n \times n = 16 \times 16$ y donde hay un solo punto en el árbol. Para representar dicho punto se necesitaría subdividir el espacio $log_k n = 4$ veces. por cada partición se deberán almacenar $k^2 = 4$ bits, en total se necesitarían 16 bits para almacenar dicho punto, lo cual es un exceso, dado que con 4 bits por coordenada, es decir con 8 bits se podría representar dicho punto. En general, el coste de un punto aislado es de $K^d \cdot \log_k n$ bits, suponiendo que todas las dimensiones son de igual rango, es decir, el coste de almacenamiento de puntos aislados crece exponencialmente al aumentar el número de dimensiones. En cambio el coste de almacenar dicho punto con los bits justos es de $d * log_k n$ bits, lo que muestra un crecimiento lineal.

Una manera de abordar este problema es el k^d-tree comprimido (ck^d-tree) [CRBF15].

Un ck^d-tree es una variante de un k^d-tree donde se definen tres tipos de nodos, blancos, negros y grises. Los blancos son aquellos nodos que representan particiones del espacio que están vacías, los negros, son aquellas particiones que tienen un único objeto y los grises que son aquellos nodos que tienen más de un objeto. Los nodos grises son internos y los blancos y negros son hojas.

Los puntos que se encuentran aislados en nodos negros, son almacenados en un arreglo separado (A), indicando la posición de manera relativa a la celda en que se encuentra y no al espacio original, lo que permite ahorrar espacio.

Para codificar el árbol se mantiene la idea básica del k^d-tree respecto de la división del espacio, asignando un 1 en T a las celdas que contienen uno o más puntos (negras o grises) y un 0 para las vacías (blancas). Para poder distinguir entre celdas negras o grises se necesita utilizar un *bitmap* (B) adicional con un bit por cada 1 en T .

Las operaciones sobre un ck^d-tree se resuelven de la misma manera que en un k^d-tree sin comprimir, pero teniendo en cuenta que existen tres clases de nodos, y que es necesario recuperando desde el arreglo A el valor de los puntos aislados cuando corresponda.

En la figura 3.5 se muestra un ejemplo de espacio con $d = 2$ y $k = 2$ codificados con un ck^d-tree. En esta figura se pueden ver las diferentes estructuras que codifican el árbol, es decir T, B y A.

El coste en espacio de esta estructura es asintótico con el espacio mínimo teórico necesario para representar m puntos en un espacio de n^d celdas [CRBF15] .

Capítulo 4

Índice para objetos móviles con restricciones en el movimiento

Nuestra propuesta para representar objetos móviles toma ideas del método de acceso espacio-temporal SEST-Index [GNR+05] y de la estructura de datos compacta k^2-tree [BLNS09]. El SEST-Index, como se comentó en el estado del arte, es una estructura que almacena, por un lado, secuencias de instantáneas, las cuales almacenan la ubicación de los objetos en un instante determinado; y por otro, una bitácora con los eventos ocurridos entre dos instantáneas. Dichos eventos encapsulan los cambios de estado que tienen los objetos en el espacio en un determinado instante, como por ejemplo, el cambio de la ubicación de un objeto.

En esta primera propuesta se tendrá en cuenta que el espacio es dividido en celdas y que en cada celda no puede haber más de un objeto. Respecto del movimiento de los objetos se asume como una restricción del modelo que los objetos cuando se muevan lo harán a celdas inmediatamente adyacentes. Ambas restricciones, que pueden ser fuertes en algunos dominios, se eliminaron en el capítulo 5

4.1. Esquema General

Existen dos abstracciones clave en nuestro modelo. La primera es el *snapshot*, el cual almacena la ubicación de todos los objetos en un instante particular. La segunda es la *bitácora*, la cual es una secuencia ordenada temporalmente de los movimientos

que ha hecho un objeto durante un intervalo de tiempo.

El método de acceso espacio temporal propuesto combina *snapshot* y *bitácoras*, con la finalidad de responder a las consultas espacio-temporales soportadas.

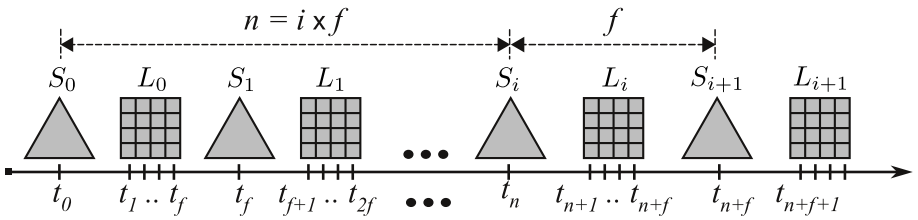


Figura 4.1: componentes del índice propuesto.

En la figura 4.1 los triángulos representan los *snapshot* y los rectángulos representan las colecciones de *bitácoras*. El primer *snapshot* S_0 captura la posición de todos los objetos en el instante t_0 y el segundo *snapshot* S_1, la posición de todos los objetos en el instante t_f. La constante f representa la frecuencia con que los *snapshot* son creados, por ende el i-ésimo *snapshot* ocurre $\lfloor i \times f \rfloor$ instantes después de t_0 (t_n en la Figura 4.1). La colección de *bitácoras* L_0 agrupa a todas las bitácoras de los objetos en el intervalo $[t_1 \ldots t_f]$ que son los instantes entre (s_0 y s_1]. Note que la colección L_0 termina en t_f y que s_1 ocurre en t_f, lo mismo ocurre para todo l_i y s_{i+1} en el índice. Esto es una redundancia necesaria para poder recuperar la ubicación de un objeto en algún instante $t' \in [t_{n+1} \ldots t_{n+f}]$, avanzando tanto desde s_i hasta t' como retrocediendo desde s_{i+1} hasta t'.

Cada *bitácora* l_i almacena los movimientos de los objetos de una manera compacta. La idea clave para lograr esta compresión se basa en el hecho de que la velocidad máxima a la que un objeto se mueve está acotada según su naturaleza, así es posible logar una compresión si en vez de almacenar la posición absoluta del objeto en el espacio en cada movimiento, se almacenan las unidades en que se ha movido tanto en el eje x como en el eje y respecto a su posición anterior. La nueva posición es registrada en la *bitácora* como una posición relativa a la anterior.

Considerando que para este modelo los objetos solo se mueven a posiciones adyacentes veamos un ejemplo sencillo donde los *snapshot* son tomados cada 8 instantes y existen dos objeto o cuyas trayectorias durante el intervalo $[0 \ldots 8]$ son:

$$\begin{aligned} o_1 &= \{(9,6),(9,7),(8,6),(7,6),(7,5),(7,4),(8,3),(9,3),(10,3)\} \\ o_2 &= \{(5,6),(5,5),(4,5),(4,4),(5,4),(5,5),(5,6),(5,7),(6,7)\} \end{aligned}$$

En este ejemplo, en el instante 0 se tomaría el primer *snapshot* y en él se almacenaría la posición absoluta de los objetos en ese instante, es decir $S_0 = \{o_1(9,6), o_2(5,6)\}$. El siguiente *snapshot* ocurre en el instante 8 y almacenaría la

posición de los objetos en ese instante y, por lo tanto, sería: $S_1 = \{o_1(10,3), o_2(6,7)\}$. En relación a las colecciones de bitácoras, en este ejemplo solo existe una que es L_0 la cual abarca el intervalo $[1 \ldots 8]$. Esta colección está compuesta por la bitácora de o_1 y la de o_2 (l_1 y l_2 respectivamente) que a continuación se detallan:

$$\begin{aligned} L_0 &= \{l_1, l_2\} \\ l_1 &= \{(0,1), (-1,0), (-1,0), (0,-1), (0,-1), (1,-1), (1,0), (1,0)\} \\ l_2 &= \{(0,-1), (-1,0), (0,-1), (1,0), (0,1), (0,1), (0,1), (1,0)\} \end{aligned}$$

En la sección 4.2 se describen las posiciones relativas y los cálculos necesarios para pasar de posiciones absolutas a relativas y viceversa.

4.2. Posiciones relativas y su procesamiento

Una posición relativa puede ser vista como el resultado de una traslación de eje. Para ello se considera a la posición anterior como el origen de un sistema de coordenadas paralelo al original y la posición actual es llevada a este nuevo sistema obteniendo así la posición relativa. El nuevo sistema representa una porción del espacio original y, por lo tanto, las posiciones en este nuevo sistema se pueden codificar usando menos bits. También puede ser vista como un vector que representa el movimiento desde la posición anterior a la posición actual.

En la siguiente sección se explica con más detalle cómo calcular posiciones relativas y cómo obtener las posiciones reales a partir de ellas.

4.2.1. Cálculo de posiciones relativas

Dada la posición p de un objeto en el espacio en el instante i y su posición en el instante anterior, se puede calcular la posición relativa r en el instante i como la diferencia entre la posición actual (p_i) con la anterior (p_{i-1}), es decir:

$$r_i = p_i - p_{i-1} \tag{4.1}$$

A partir de esta ecuación se deducen las ecuaciones para realizar la operación inversa es decir, llevar una posición relativa en el instante i a la posición absoluta en el instante i o en el instante $i-1$:

$$p_i = r_i + p_{i-1} \tag{4.2}$$

$$p_{i-1} = p_i - r_i \tag{4.3}$$

Ejemplo 4.1 *considere que un objeto para el instante 4 y 5 tiene las posiciones absolutas* $(20, 45)$ *y* $(19, 46)$*, respectivamente. Así:*

La posición relativa en el instante 5 *usando la ecuación 4.1 será:*

$$r_5 = p_5 - p_4 = (19, 46) - (20, 45) = (19 - 20, 46 - 45) = (-1, +1)$$

La posición absoluta en el instante 5 usando la ecuación 4.2 será:

$$p_5 = r_5 + p_4 = (-1, +1) + (20, 45) = (-1 + 20, 1 + 45) = (19, 46)$$

Y la posición absoluta en el instante 4 usando la ecuación 4.3 será:

$$p_4 = p_5 - r_5 = (19, 46) - (-1, +1) = (19 + 1, 46 - 1) = (20, 45)$$

□

4.2.2. Cálculo con secuencias de posiciones relativas

Las *bitácoras* registran secuencias de posiciones relativas de los objetos construidas a partir de la secuencia de sus posiciones absolutas. A partir de los cálculos básicos de la sección anterior, en esta sección se describen los cálculos con las secuencias de posiciones relativas.

Definición 4.1 (Secuencia de posiciones relativas) *Si* $P = \{p_0, p_1, p_2, \ldots, p_n\}$ *es la secuencia de posiciones absolutas del objeto o en el intervalo de tiempo* $[0 \ldots n]$*, entonces* $R = \{r_1, r_2, r_3, \ldots, r_n\}$ *es la secuencia de posiciones relativas de o en el intervalo*$[1 \ldots n]$ *sí y solo sí* $\forall i \in [1 \ldots n], \exists r_i \in R \wedge \exists p_{i-1}, p_i \in P \mid r_i = p_i - p_{i-1}$ □

En la definición 4.1, note que R comienza en 1 y no en 0 como en P, esto ocurre por dos razones: primero, porque r_0 no se puede calcular dado que no existe p_{-1} y segundo, es fundamental contar al menos con p_0 para poder recuperar las posiciones absolutas a partir de R.

De este modo R no reemplaza a P, sino a la sub-secuencia $P_{1\ldots n}$ y, por lo tanto, se necesita el par $\langle p_0, R\rangle$ para reemplazar a P. En nuestro modelo p_0 es almacenado en el *snapshot* y R en la *bitácora*.

Ejemplo 4.2 *La siguiente secuencia* P *contiene las posiciones absolutas de un objeto en el intervalo* $[0 \ldots 9]$ *y la secuencia* R *son las posiciones relativas correspondientes en el intervalo* $[1 \ldots 9]$*:*

$$\begin{aligned} P_{0\ldots9} &= \{(9,6),(9,7),(7,7),(4,7),(4,6),(4,3),(4,2),(5,1),(7,1),(10,1)\} \\ R_{1\ldots9} &= \{(0,1),(-2,0),(-3,0),(0,-1),(0,-3),(0,-1),(1,-1),(2,0),(3,0)\} \end{aligned}$$

□

Dada la propiedad asociativa de la adición es posible recuperar la posición absoluta del objeto en cualquier instante t del intervalo $[1 \dots n]$ sumando a la posición inicial el total de desplazamiento en cada eje hasta t:

$$p_t = p_0 + \sum_{i=1}^{t} r_i \tag{4.4}$$

Ejemplo 4.3 *Considere que para un objeto tenemos su posición absoluta en el instante inicial* $p_0 = (9, 6)$ *y su secuencia de posiciones relativas* R *en el intervalo* $[1 \dots 9]$ *con los siguientes valores:*

$$R_{1\dots9} = \{(0,1), (-2,0), (-3,0), (0,-1), (0,-3), (0,-1), (1,-1), (2,0), (3,0)\}$$

Podemos obtener la posición absoluta en el instante 7 usando la ecuación 4.4

$$\begin{aligned} p_7 &= p_0 + \sum_{i=1}^{7} r_i \\ &= (9,6) + \{(0,1) + (-2,0) + (-3,0) + (0,-1) + (0,-3) + (0,-1) + (1,-1)\} \\ &= (9,6) + (-4,-5) \\ &= (9-4, 6-5) \\ &= (5,1) \end{aligned}$$

□

Si además de almacenar la posición absoluta inicial del objeto (p_0) almacenamos también la última (p_n), entonces podemos obtener la posición absoluta al tiempo t restando a p_n el total de desplazamientos en cada eje desde t a n:

$$p_t = p_n - \sum_{i=t+1}^{n} r_i \tag{4.5}$$

Ejemplo 4.4 *Considere que para un objeto tenemos su posición absoluta en el instante final* $p_9 = (10, 1)$ *y su secuencia de posiciones relativas* R *en el intervalo* $[1 \dots 9]$ *con los siguientes valores:*

$$R_{1\dots9} = \{(0,1), (-2,0), (-3,0), (0,-1), (0,-3), (0,-1), (1,-1), (2,0), (3,0)\}$$

Podemos obtener la posición absoluta en el instante 7 usando la ecuación 4.5

$$\begin{aligned} p_6 &= p_9 - \sum_{i=7}^{9} r_i \\ &= (10,1) - \{(1,-1) + (2,0) + (3,0)\} \\ &= (10,1) - (6,-1) \\ &= (10-6, 1+1) \\ &= (4,2) \end{aligned}$$

□

4.3. Estructuras de datos

En esta sección se describen en detalle las estructuras de datos propuestas para *snapshot*, bitácoras, colecciones de bitácoras y finalmente el índice.

4.3.1. Snapshot

Un *snapshot* permite indexar a m objetos que se encuentran ubicados en un espacio discreto de dos dimensiones de manera compacta.

Cada *snapshot* usa un k^2-tree para representar de forma compacta las posiciones (x, y) de cada uno de los objetos indexados y una permutación que permite vincular el ID del objeto con la celda correspondiente y así, dada una celda, poder identificar cuál es el objeto asociado, o bien dado un objeto saber en qué celda se encuentra.

El k^2-tree, como se mencionó en la sección 2.3.4 es almacenado en dos bitarrays T y L. Por un lado, T contiene el valor binario de los nodos en el orden resultante de recorrer en anchura el k^2-tree desde la raíz hasta el penúltimo nivel, por otro lado L se obtienen de recorre en anchura las hojas del último nivel.

Note que el k^2-tree permite conocer cuáles son las celdas que contienen objetos, pero no permite saber cuáles son los objetos que están contenidos en dichas celdas. Para ello se usa una permutación entre el ID del objeto y el identificador de la celda que corresponde al número resultante de contar los unos en el bitarray L del k^2-tree, es decir, para el primer uno de L el ID de la celda es 1, para el segundo es 2, ..., para el n-ésimo es n, lo que se obtiene mediante la operación de rank sobre la posición de la celda en L ($ID = rank(L, i)$). únicamente los bits a 1 en L tienen un ID de celda, el resto no.

La estructura de datos para permutaciones permite dos operaciones principales π y π^{-1}. La primera permite obtener el ID del objeto asociado a una celda en

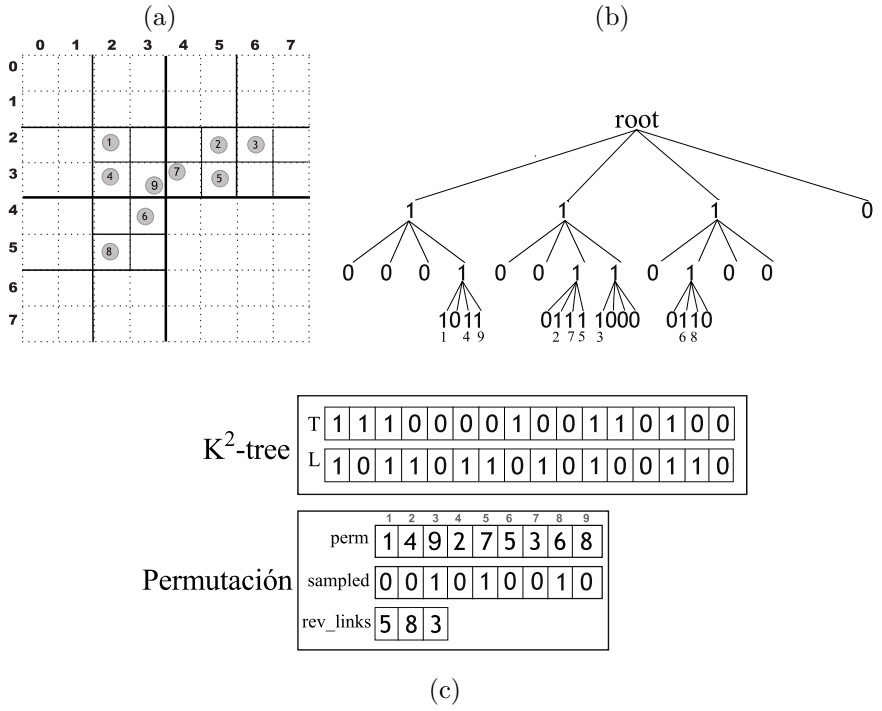


Figura 4.2: Un ejemplo de *snapshot* con $k = 2$: (a) espacio geográfico, (b) k^2-tree conceptual, (c) *snapshot* almacenado con $t = 2$.

particular (un uno a nivel de las hojas) y la operación π^{-1} permite identificar la celda a nivel de hoja que está asociada al ID de un objeto.

Como se mensionó en la sección 2.3.3, en el estado del arte existen varias estructuras de datos compactas para representar permutaciones. Se ha optado por la estructura de [MRRR03] porque permite obtener π en tiempo constante, requisito indispensable para responder consultas por rango espacial de manera eficiciente, y π^{-1} en tiempo $O(\log_2 m)$, donde m es el número de objetos.

Como se comentó en la sección 2.3.3, la estructura de datos para la permutación utiliza 3 arreglos, el primero de ellos (*perm*) representa la permutación entre el ID de la celda con el ID del objeto, por ejemplo, la posición 5 del arreglo perm contiene un 7, esto significa que en la celda 5 del k^2-tree se encuentra el objeto $ID = 7$, Así saber cual es el objeto que se encuentra en una celda es directo ($\mathcal{O}(1)$).

4.3.2. Bitácoras

Como se comentó anteriormente en el esquema general, en este modelo se ha considerado que los objetos cuando se mueven lo hacen a las celdas adyacentes. En la figura 4.3 se puede observar que existen ocho posibles movimientos considerando que un objeto se encuentra en la celda central (la oscura), lo que sumado al hecho que el objeto no se mueva existen ocho posibles estados de un objeto en un instante determinado.

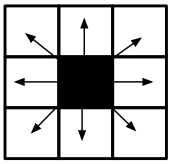


Figura 4.3: 8 posibles movimientos de un objeto para cambiar de posición.

Para codificar el estado de un objeto en la bitácora se utilizan 4 bits, uno por cada dirección del movimiento, izquierda, derecha, arriba y abajo respectivamente, los cuales en conjunto determinan el estado del objeto en un instante dado. Para ello se utilizan 4 *bitmaps* uno para cada tipo de desplazamiento. El largo de estos *bitmaps* corresponde a la cantidad de instantes que son codificados en la bitácora (parámetro f de la figura 4.1) de modo que no es necesario codificar el tiempo de modo explícito.

Considerando como ejemplo la bitácora presentada en el esquema general (ver sección 4.1): $l_2 = \{(0,-1),(-1,0),(0,-1),(1,0),(0,1),(0,1),(0,1),(1,0)\}$, Se presenta el siguiente ejemplo de bitácora codificada con cuatro *bitmaps*:

	1	2	3	4	5	6	7	8
left	0	1	0	0	0	0	0	0
right	0	0	0	1	0	0	0	1
up	1	0	1	0	0	0	0	0
down	0	0	0	0	1	1	1	0

Figura 4.4: Ejemplo de codificación de una bitácora utilizando cuatro *bitmaps*

Usando este sistema de codificación de bitácoras es posible recuperar la posición

absoluta de un objeto en cualquier punto conocido a partir de la información contenida en el *snapshot* y la bitácora siguiendo unas sencillas operaciones. Continuando con el ejemplo anterior asuma que desea obtener la posición absoluta del objeto o_2 en el instante 5, cuyo valor original ($p_5 = (5, 5)$) fue codificado como un desplazamiento relativo dentro de la bitácora. Para recuperar p_5 se necesita la posición absoluta del objeto en el instante 0 que se encuentra almacenada en el *snapshot* cuyo valor es $(5, 6)$ y el desplazamiento realizado por el objeto desde el instante 1 al 5, lo que se calcula utilizando la bitácora de la siguiente manera: utilizando la operación de rank_1 sobre los *bitmaps* calculamos la suma de los desplazamientos para cada *bitmap* hasta el instante 5, luego se calcula el desplazamiento en el eje x (δ_x) como la diferencia entre los desplazamientos hacia la derecha y la izquierda ($\delta_x = rank(right, 5) - rank(left, 5) = 1 - 1 = 0$) y el desplazamiento en el eje y (δ_y) como la diferencia entre los desplazamientos hacia abajo menos los desplazamientos hacia arriba ($\delta_y = rank(down, 5) - rank(up, 5) = 1 - 2 = -1$), de este modo el desplazamiento del objeto o_2 entre el instante 1 y 5 es $(0, -1)$. Una vez conocida la posición del objeto o_2 en el *snapshot* ($p_0 = (5, 6)$) y su desplazamiento ($(\delta_x, \delta_y) = (0, -1)$) se calcula la posición absoluta como la suma vectorial entre ambos vectores, es decir:

$$\begin{aligned} p_5 &= p_0 + (\delta_x, \delta_y) \\ p_5 &= (5, 6) + (0, -1) \\ p_5 &= (5 + 0, 6 - 1) \\ p_5 &= (5, 5) \end{aligned}$$

4.3.3. Colección de bitácoras

Como se comentó en el esquema general existe una **colección de bitácoras** la cual agrupa las bitácoras de cada objeto en en un mismo rango de tiempo en una única estructura. La estructura de datos utilizada para la colección de bitácoras es un arreglo de bitácoras, donde cada posición corresponde de manera directa al id de cada objeto. Así el objeto de ID 50, está en la posición 50. Se asume que si en el sistema existen 500 objetos hay 500 ID que van de 1 a 500.

4.3.4. Índice

Ya se han descrito las partes principales del índice, solo resta por comentar la estructura de datos principal que consiste en dos vectores, uno para almacenar la secuencia de *snapshots* (S) y el segundo para almacenar la secuencia de colecciones de bitácoras (CB). Como este índice está pensado para manejar información histórica, es posible determinar a priori el tamaño de estos vectores evitando la utilización de una estructura de datos más compleja que permita el crecimiento dinámico de ella.

En la figura 4.5 se puede observar un ejemplo del índice, con todas las estructuras de datos que lo componen. Para el *snapshot* S_3 se presenta la estructura con datos concretos. En el caso de las colecciones de bitácoras se presenta CB_1 que tiene m bitácoras y de las cuales se da un ejemplo concreto para B_n.

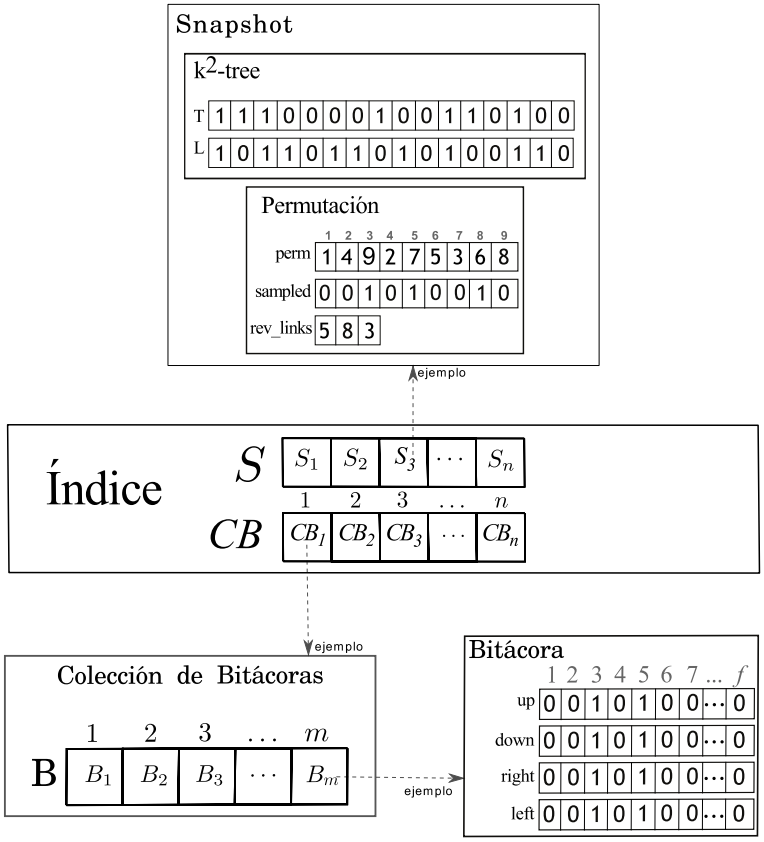


Figura 4.5: Ejemplo del índice propuesto

4.4. Algoritmos

4.4.1. Obtener la ubicación de un objeto

Cuando se desea obtener la posición de un objeto en un determinado instante de tiempo puede ocurrir que el instante de la consulta corresponda con el de un

snapshot . Este caso particular se describirá primero, debido a que se utiliza en el algoritmo que resuelve el caso general.

Obtener la ubicación de un objeto en un Snapshot

Cuando se almacenan las posiciones de los objetos en el k^2-tree, éstas son guardadas implícitamente y están determinadas por el camino que es necesario recorrer desde la raíz a la hoja que representa la celda del espacio en la que está el objeto. De modo que en cada paso se visitará un nodo el cual representa una celda a cierto nivel de granularidad o partición del espacio. Como el tamaño del espacio se conoce a nivel de la raíz es posible determinar el tamaño de cada partición representada por un hijo en particular. Al ir descendiendo por el árbol, en cada iteración se lleva la cuenta los límites del área del padre de modo de poder determinar exactamente los límites de las particiones de los hijos. Al llegar a la hoja, la partición es una celda que no se puede seguir dividiendo y, por lo tanto, se obtiene la posición absoluta del objeto que se encuentra en dicha celda.

Dado lo anterior, el proceso necesario para obtener la posición asociada a un objeto o en un *snapshot* s tiene tres pasos principales, los cuales están separados en dos algoritmos.

Algorithm 4.1 (**FindPath**), Obtiene el camino desde la raíz a la hoja en el *snapshot* que contiene un objeto especificado como parámetro

Entrada: Snapshot s, OID o.
Salida: Stack.
1: $path \leftarrow$ New($Stack$)
2: $p \leftarrow s.permutacion.\pi^{-1}(o)$
3: $x =$ Select($s.L, p$)
4: $\quad +$LengthInBits($s.T$)
5: **while** $x \geqslant 0$ **do**
6: $\quad i = \mathrm{I}(x)$
7: $\quad path =$ Push($path, i$)
8: $\quad x =$ Padre(x)
9: **end while**
10: **return** $path$

Algorithm 4.2 (**getObjectPos**), encuentra en un *snapshot* la posición absoluta de un objeto pasado como parámetro

Entrada: Snapshot s, OID o.
Salida: Punto.
1: $x_1 = 0$, $y_1 = 0$
2: $x_2 = s.N - 1$, $y_2 = s.N - 1$
3: $path \leftarrow$ FindPath(s, o)
4: **while** Not Empty($path$) **do**
5: $\quad i =$ Pop($path$)
6: $\quad colum = i \mod k$
7: $\quad row = \lfloor i \div k \rfloor$
8: $\quad piv_x = (x_2 - x_1 + 1) \div k$
9: $\quad piv_y = (y_2 - y_1 + 1) \div k$
10: $\quad x_2 = piv_x(colum + 1) + x_1 - 1$
11: $\quad x_1 = x_2 - piv_x + 1$
12: $\quad y_2 = piv_y \cdot (row + 1) + y_1 - 1$
13: $\quad y_1 = y_2 - piv_y + 1$
14: **end while**
15: **return** NewPoint(x_1, y_1)

El primer paso de este proceso ocurre en el algoritmo 4.1 FindPath() línea 2 y 3 y consiste en determinar la celda donde está contenido el objeto buscado. Para ello, primero se obtiene el ID de la celda (i) en la que está el objeto buscado (o) a través de la permutación y luego se obtiene la posición que ocupa la i-ésima celda en el arreglo L del k^2-tree (x) mediante una operación de select_1.

El segundo paso es obtener el camino que es necesario recorrer desde la raíz hasta la celda donde está el objeto. Para ello se toma como punto de partida x y se hace un recorrido ascendente del k^2-tree. En cada paso almacena en una pila el orden que le corresponde a la celda dentro de sus hermanos ($I(x)$). Finalmente en la pila está el camino de la raíz a la hoja del objeto buscado. Este segundo paso también lo encontramos en el algoritmo 4.1 desde la línea 3 en adelante.

Como tercer y último paso hay que recorrer el camino obtenido en el paso 2 llevando cuenta en cada paso de los límites de la partición. Esto se realiza en el el algoritmo 4.2 getObjectPos().

Para una mejor comprensión de estos algoritmos es necesario conocer las operaciones básicas para obtener el i-ésimo hijo de un nodo, la posición del padre de un nodo y el orden de hijo que le corresponde a un nodo; las cuales se describen a continuación.

Definición 4.2 (i-ésimo hijo de x) *Dada la posición (x) de un nodo con hijos, $\forall i \in [0 \dots k^2 - 1]$ el i-ésimo hijo de x es [Lad11]: $hijo_i(x) = Rank(T, x) * k^2 + i$.* □

Definición 4.3 (Padre de x) *Dada la posición x de un nodo la posición del padre de x es: $Select(T : L, \lfloor x \div k^2 \rfloor)$, si $x \geq k^2$, de lo contrario es -1 (la posición ficticia de la raíz), donde $T : L$ es la concatenación de T y L.*

$$padre(x) = \begin{cases} -1 & si\ x < k^2 \\ Select(T : L, \lfloor x/k^2 \rfloor) & si\ x \geq k^2 \end{cases}$$

□

Como cada padre aporta k^2 bits al arreglo, si se calcula $p = \lfloor x/k^2 \rfloor$ se determina cuantos bloques de k^2 bits hay hasta x. Si enumeramos los bloques de 0 a p, la numeración coincide con la numeración de los padres de los nodos, que son los unos del *bitmap* $T : L$. Notar que los primeros k^2 bits representan a los hijos de la raíz, y estos están en el bloque 0. El primer 1 del bloque 0, será el primer padre y sus hijos estarán en el bloque 1, el segundo 1 que puede o no estar en el bloque 0, será el segundo padre y le corresponderá el bloque 2, y así sucesivamente. De este modo, conocer la posicón de el padre de un nodo x, se reduce a encontrar el p-ésimo uno del *bitmap* $T : L$ lo que se obtiene mediante la operación de select_1.

Definición 4.4 (Posición de x como hijo) *Dada la posición x de un nodo, la posición de hijo que ocupa x es $I(x) = x \mod k^2$* □

Como cada nodo aporta k^2 bits a $T : L$ su largo es un múltiplo de k^2. Si aplicamos aritmética modular con k^2 sobre la posición x de un nodo en particular obtendremos un número entre 0 y $k^2 - 1$, que corresponderá a la posición que ocupa el nodo en el bloque que comparte con sus hermanos.

Obtener la ubicación de un objeto en un instante cualquiera

El algoritmo 4.3 describe en detalle los pasos para obtener la posición de un objeto *oid* en un instante t_q en particular.

Lo primero que se debe hacer para responder la consulta es obtener la posición del objeto *pos* en el *snapshot* (s_i) inmediatamente anterior al tiempo de la consulta (líneas 1 y 2).

Si el instante de la consulta (t_q) coincide con el instante del *snapshot* s_i (t_0) la respuesta es *pos*. Esta condición es revisada en la línea 5, dado que t corresponde a la cantidad de instantes transcurridos entre t_0 y t_q.

En caso contrario, hay que obtener el desplazamiento del objeto desde t_0 hasta t_q, utilizando para ello la bitácora del objeto que contiene sus desplazamientos hasta t_q (líneas 6-8). Con el desplazamiento obtenido (δ_x, δ_y) se actualiza la posición del objeto hasta el instante de la consulta (línea 9). La respuesta en este caso será la posición actualizada del objeto.

4.4.2. Time Slice

Una consulta de *time slice* permite obtener todos los objetos (id) que se encuentran en una región del espacio en un instante determinado. Una variación a esta consulta es obtener, no solo el identificador del objeto, sino que además su posición.

Para responder a una consulta de *time slice* sobre esta propuesta se consideran dos casos. El primer caso ocurre cuando el instante de la consulta t_q coincide con el instante en que se ha tomado un *snapshot* , entonces la consulta de time slice se resuelve mediante una consulta por rango sobre el *snapshot* (ver 4.4.2). El segundo ocurre cuando t_q se encuentra entre dos *snapshot* s_i y s_{i+1} (ver 4.4.2).

A continuación se presenta el primer caso.

Algorithm 4.3 (**getPosition)** obtiene la posición registrada en el índice de un objeto en un instante concreto

Input: *cst*: una instancia del índice (CST), *oid*: identificador del objeto buscado, t_q: instante consultado.
Output: Punto : Punto con la ubicación del objeto *oid* en el instante t_q indexado en *cst*.

1: $s_i = \text{GETSNAPSHOT}(cst, t_q)$
2: $pos = \text{GETOBJECTPOS}(s_i, oid)$
3: $t_0 = \text{TIMESTAMP}(s_i)$
4: $t = t_q - t_0$
5: **if** $(t > 0)$ **then**
6: $\quad L_i = \text{GETLOGCOLECTION}(cst, t_q)$
7: $\quad \delta_x = rank(L_i[oid].right, t) - rank(L_i[oid].left, t)$
8: $\quad \delta_y = rank(L_i[oid].down, t) - rank(up, t)$
9: $\quad pos = pos + (\delta_x, \delta_y)$
10: **end if**
11: **return** *pos*

Consultas por rango espacial en un Snapshot

La consulta por rango espacial en el *snapshot* se responde con una llamada a la operación de *Rango* del k^2-tree a la cual se le ha hecho una modificación al llegar a un nodo hoja de último nivel de modo que ahora se recuperarán los identificadores de objetos que están asociados a esa celda a los cuales se le asocia como punto los valores (x, y) de la celda asociada. El algoritmo 4.4 muestra las modificaiones comentadas al algoritmo presentado en [Lad11]. El parámetro n corresponde al número de columnas y filas, luego p_1, p_2, q_1, q_2 es la región espacial de la consulta, d_p, d_q, la fila y la columna en la que comienza la región espacial del primer hijo de la raíz z, y *output* es el conjunto de salida. La llamada incial se hace dentro de la operación RANGEQUERY la cual es un envoltorio de la llamada $\text{RANGE}(n, r.p1, r.p2, r.q1, r.q2, 0, 0, -1)$

Time slice entre dos snapshots

En este caso no es posible hacer la consulta por rango en t_q, dado que sobre las bitácoras el acceso es por objeto y no espacial. En términos generales, el proceso de resolver la consulta en este caso consiste en hacer la consulta por rango en el *snapshot* que más convenga (s_i o s_{i+1}) y luego, utilizando las bitácoras, actualizar los objetos hasta el instante de la consulta. Aquellos objetos que queden dentro de la región de la consulta r en t_q serán el resultado y se descartará el resto de los objetos.

Es importante destacar que al consulta r sobre un *snapshot* (t_0 o t_f), el resultado puede excluir a uno o más de los objetos que deben forman parte del resultado. Por

Algorithm 4.4 (**Range**)$(n, p_1, p_2, q_1, q_2, d_p, d_q, z, output)$

$T = \mathsf{k}^2\text{-tree}.T; L = \mathsf{k}^2\text{-tree}.L$
if $z \geq |T|$ **then**
  if $L[z - |T|] = 1$ **then**
    $O = s.permutacion.\pi(Rank(T, z - |T|))$ ▷ modificado
    $output = output \cup (O \times \{(d_p, d_q)\})$ ▷ modificado
  end if
else
  if $z = -1 \vee T[z] = 1$ **then**
    $y = rank(T, z) \cdot k^2$
    for $i = \lfloor p_1/(n/k) \rfloor \ldots \lfloor p_2/(n/k) \rfloor$ **do**
      $p'_1 = 0$
      if $i = \lfloor p_1/(n/k) \rfloor$ **then**
        $p'_1 = p_1 \mod (n/k)$
      end if
      $p'_2 = (n/k) - 1$
      if $i = \lfloor p_2/(n/k) \rfloor$ **then**
        $p'_2 = p_2 \mod (n/k)$
      end if
      for $j = \lfloor q_1/(n/k) \rfloor \ldots \lfloor q_2/(n/k) \rfloor$ **do**
        $q'_1 = 0$
        if $j = \lfloor q_1/(n/k) \rfloor$ **then**
          $q'_1 = q_1 \mod (n/k)$
        end if
        $q'_2 = (n/k) - 1$
        if $j = \lfloor p_2/(n/k) \rfloor$ **then**
          $p'_2 = p_2 \mod (n/k)$
        end if
        $d'_p = d_p + (n/k) \cdot i$
        $d'_q = d_q + (n/k) \cdot j$
        $z' = y + k \cdot i + j$
        RANGE$(n/k, p'_1, p'_2, q'_1, q'_2, d'_p, d'_q, z', output)$
      end for
    end for
  end if
end if

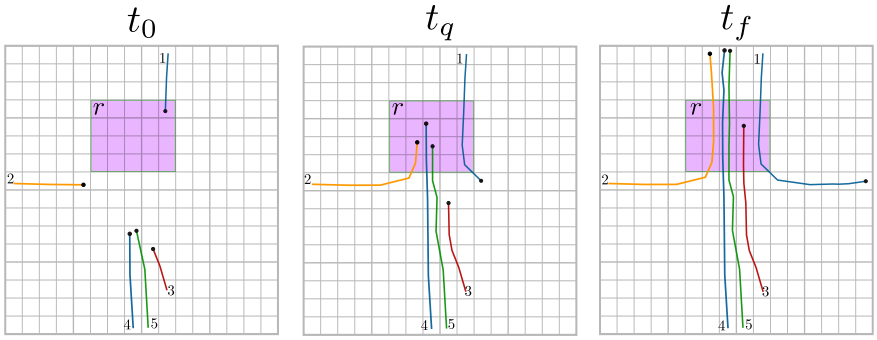


Figura 4.6: Consulta con el rango r en el *snapshot* s_i (t_0) y $s_{i+1}(t_f)$

el ejemplo en la figura 4.6), cuando se escoge el *snapshot* s_i que es previo a t_q los objetos 2, 4 y 5 no están en r, pero dado sus desplazamientos posteriores, en t_q sí lo están y, por lo tanto, son parte de los resultados de la consulta. Por otro lado, si se escoge el *snapshot* s_{i+1} que ocurre en el instante t_f, pueden existir objetos que están fuera de r en s_{i+1} pero que en t_q sí lo estaban (objetos 2, 4 y 5 en la figura 4.6).

Por esta razón al realizar la consulta por rango en uno de los *snapshot* (s_i o s_{i+1}) se debe usar una región que sea lo suficientemente grande para contener a todos los objetos que se encuentren en r en el instante t_q, aún cuando no lo estén en el *snapshot* consultado. Mientras más grande sea el área de la consulta sobre un *snapshot*, mayor es el tiempo que tomará la consulta por rango en el *snapshot* y, por lo tanto, se debe escoger el *snapshot* para el cual la ampliación del área de la consulta sea más pequeña.

Para determinar el área de la consulta ampliada r' sobre s_i es necesario conocer cuál es el máximo desplazamiento realizado en la colección de bitácoras desde t_0 al instante t_q $(izMD)$, y para determinar r'' sobre s_{i+1} se necesita conocer cuál es el máximo desplazamiento desde t_q a t_f $(derMD)$. Como los movimientos están restringidos a casillas adyacentes, el desplazamiento máximo será igual a la cantidad de instantes transcurridos desde el *snapshot* de referencia (s_i o s_{i+1}) al instante de la consulta. Así $izMD = t_q - t_0$ y $derMD = t_f - t_q$. Con estos valor se extiende r por cada uno de sus lados de modo que $r' = (r.x1 - izMD, r.x2 + izMD, r.y1 - izMD, r.y2 + izMD)$ y $r'' = (r.x1 - derMD, r.x2 + derMD, r.y1 - derMD, r.y2 + derMD)$

En la figura 4.7 se pueden observar las áreas r' y r''. Éstas se han creado dado los máximos desplazamientos: $izMD = 6$ y $derMD = 2$. Tanto r' como r'' capturan a los objetos 2,3,4 que es lo buscado, pero en este caso conviene hacer la consulta por rango en s_{i+1} con r'', dado que el área de r'' es menor que r'.

Luego de realizar la consulta por rango sobre el *snapshot* escogido tenemos una

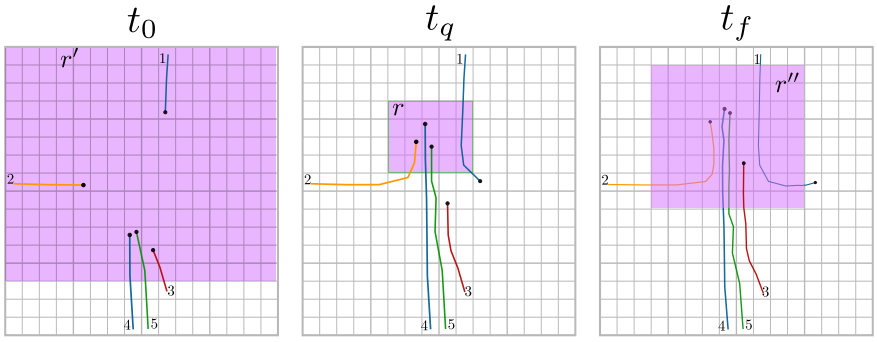


Figura 4.7: Ejemplo de ampliación de área r en r' y r''

lista de candidatos a evaluar, que en el caso del ejemplo de la figura 4.7 son los objetos 2,3,4,5. Dicha lista indica el OID del objeto y la posición almacenada en el *snapshot*.

Luego se evalúan los candidatos restantes. Por cada uno de los objetos, se obtiene su bitácora. En el caso de haber escogido el *snapshot* s_i se avanza[1] el objeto usando la suma de los desplazamientos desde el instante $t_0 + 1$ a t_q; si es el caso de haber escogido s_{i+1} se retrocede el objeto usando la suma de los desplazamientos desde $t_q + 1$ hasta t_f. Si el objeto está en r después de su actualización, éste se incluye en los resultados y en caso contrario se descarta.

Cuando no interesa la posición de cada objeto en r en el instante t_q es posible evitar la actualización de aquellos objetos que, aún desplazándose hacia afuera de r a máxima velocidad, se encontrarán en r en el instante t_q y, por lo tanto, son parte de los resultados.

Estos objetos son aquellos que se encuentran en el área r'_i para las consultas sobre s_i (ver línea 25 del algoritmo 4.5) y r''_i para las consultas sobre s_{i+1} (ver línea 11 del algoritmo 4.5).

En el algoritmo 4.5dicha condición es revisada en las líneas 13 y 27, para s_{i+1} y s_i respectivamente.

4.4.3. Time Interval

Las consultas de Time Interval se caracterizan porque la componente temporal es un intervalo y no un instante.

[1]para más detalle ver sección 4.2

Algorithm 4.5 (**Time Slice**) obtiene todos los objetos que se encuentran dentro de una región del espacio en particular en un instante de tiempo dado.

Input: *cst*: una instancia de CST-index, r: regiñ de la consulta, t_q: instante
Output: conjunto de objetos en *cst* que intersecan r en t

1: $result = \varnothing$; $s_i = \text{GETSNAPSHOT}(cst, t_q)$; $t_0 = \text{TIMESTAMP}(s_i)$
2: **if** $t_0 = t_q$ **then**
3: **return** $\text{RANGEQUERY}(s_i, r)$
4: **end if**
5: $t_f = t_0 + f$ $s_{i+1} = \text{GETSNAPSHOT}(cst, t_f)$ $L = \text{GETLOG}(cst, t_q)$
6: $izMD = t_q - t_0$; $derMD = t_f - t_q$
7: $r' = (r.x1 - izMD, r.x2 + izMD, r.y1 - izMD, r.y2 + izMD)$
8: $r'' = (r.x1 - derMD, r.x2 + derMD, r.y1 - derMD, r.y2 + derMD)$
9: **if** $(\text{AREA}(r'') < \text{AREA}(r')) \wedge (s_{i+1} \neq NULL)$ **then**
10: $cand = \text{RANGEQUERY}(s_{i+1}, r'')$
11: $r_i'' = (r.x1 + derMD, r.x2 - derMD, r.y1 + derMD, r.y2 - derMD)$
12: **for all** c such that $c \in cand$ **do**
13: **if** c in r_i'' **then**
14: $result = result \cup \{c\}$
15: **else**
16: $l = L[c]$
17: $c.pos = c.pos - \text{GETSUMMOV}(l, t_q + 1, t_f)$
18: **if** c in r **then**
19: $result = result \cup \{c\}$
20: **end if**
21: **end if**
22: **end for**
23: **else**
24: $cand = \text{RANGEQUERY}(s_i, r')$
25: $r_i' = (r.x1 + izMD, r.x2 - izMD, r.y1 + izMD, r.y2 - izMD)$
26: **for all** c such that $c \in cand$ **do**
27: **if** c in r_i' **then**
28: $result = result \cup \{c\}$
29: **else**
30: $l = L[c]$
31: $c.pos = c.pos + \text{GETSUMMOV}(l, t_0 + 1, t_q)$
32: **if** c in r **then**
33: $result = result \cup \{c\}$
34: **end if**
35: **end if**
36: **end for**
37: **end if**
38: **return** *result*

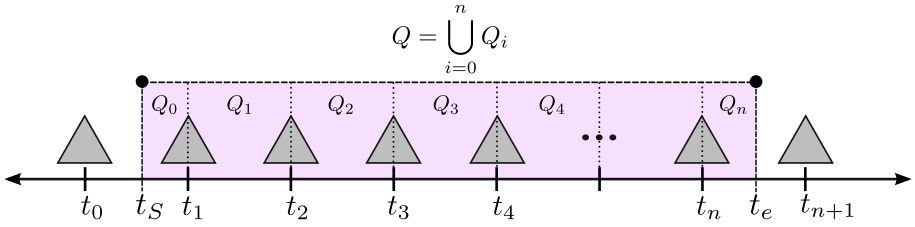


Figura 4.8: División de una consulta de Time Interval de intervalo grande

Como se ejemplifica en la figura 4.8 para las consultas de time interval, el intervalo temporal de la consulta puede ser muy grande e intersectar con n *snapshot*. Si bien es posible resolver la consulta únicamente con el primer *snapshot* y utilizar las n bitácoras, esto tiene un problema práctico que es la necesidad de ampliar la consulta a un tamaño tan grande como todo el espacio. Por lo anterior la consulta es descompuesta en $n+1$ subconsultas de time interval cuyos resultados $(Q_0, Q_1, \cdots, Qn)$ son unidos para formar la respuesta global $(Q = \bigcup_{i=0}^{n} Q_i)$.

Si $[t_s, t_e]$ es el intervalo de la consulta, la primera subconsulta tendrá un intervalo temporal de $[t_s, t_1 - 1]$, donde t_1 es el instante del primer *snapshot* que ocurre después de t_s. La segunda subconsulta tendrá el intervalo $[t_1, t_2 - 1]$ donde t_2 es el instante del snapshop que ocurre después de t_1 y así se continúa subdividiendo hasta completar todo el intervalo de la consulta con la subconsulta de intervalo $[t_n, t_e]$.

El algoritmo 4.6 crea las subconsultas y las ejecuta. Como los objetos que son parte de los resultados de una subconsulta, también lo serán de la respuesta global no es necesario volver a evaluarlos en las siguientes consultas. Por ello las respuestas parciales son pasadas como parámetro a las siguientes subconsultas. Después de evaluar todas las subconsultas, el resultado entregado por la última será el resultado global. El algoritmo 4.6 llama al algoritmo 4.7 (lines 4, 8, and 12) para resolver las subconsultas.

El algoritmo que resuelve las subconsultas es muy similar al de *time slice*, con la diferencia que al escoger el *snapshot* se deben considerar los desplazamientos máximos al instante más lejano del intervalo de la consulta. La otra diferencia es que al evaluar un candidato se deberá actualizar el objeto más de una vez si éste se ha movido durante el intervalo de la consulta. Lo que hace que la consulta de time interval sea más costosa que una consulta de time slice.

Durante la evaluación del candidato hay un par de estrategias que se usan para evitar, en lo posible, tener que procesar toda la trayectoria del objeto durante el intervalo de la consulta.

Lo primero que se evalúa es si el candidato forma parte de los resultados parciales. Si es el caso, entonces se procede a evaluar el siguiente objeto. Si no, se continúa

Algorithm 4.6 (**Time Interval**) obtiene todos los objetos que se encuentran dentro de una región del espacio en particular en un intervalo de tiempo dado.

input: *cst*: una instancia de CST-index, r: región de la consulta, $[t_s, t_e]$: intervalo temporal.
Output: conjunto de objetos de *cst* que intersectan con r en algun instante de $[t_s, t_e]$

1: $result = \varnothing$, $start = t_s$, $end = t_e$
2: $S = \{$subsecuencia de snapshot en *cst* que ocurren en el intervalo $[t_s, t_e]\}$
3: **if** $|S| \leq 1$ **then**
4: LIMITEDTI$(cst, r, t, result)$
5: **else**
6: **for** $i = 1$ **To** $|S| - 1$ **do**
7: $t' = [start,$ TIMESTAMP$(S_i) + f]$
8: LIMITEDTI$(cst, r, t', result)$
9: $start =$ TIMESTAMP$(S_{i+1}) + 1$
10: **end for**
11: $t' = [start, t.end]$
12: LIMITEDTI$(cst, r, t', result)$
13: **end if**
14: **return** *result*

revisando si el objeto está en la región interior. Si se cumple con esta condición el objeto es incluido en los resultados y se procede a evaluar el siguiente objeto candidato. Si no está en la región interior se actualiza el objeto al inicio del intervalo de la consulta y se revisa en qué región quedó. En este punto el objeto puede estar en la región universo, que corresponde al área del espacio, que dado el máximo desplazamiento, es imposible que entre en la región de la consulta r. Si este es el caso, el objeto se deja de evaluar pues no forma parte de los resultados. Si no está en la región universo, puede ocurrir que se encuentre en r, en este caso el objeto es parte de los resultados y, por lo tanto, se agrega al conjunto de resultados (*result*) y se termina la evaluación del objeto. Si el objeto aún es candidato se procede a actualizar el objeto en cada una de las posiciones que se han registrado para él durante el intervalo de la consulta. Durante esta evaluación, en cada iteración se ajusta el máximo desplazamiento posible, restando el desplazamiento actual del objeto lo que permite afinar la evaluación de la región en la que ha queda el objeto en la iteración actual. Nuevamente se evalúa si el objeto entra a la región de la consulta para incluirlo en los resultados o si se mueve a la región universo, porque si se dan alguno de estos casos ya no es necesario evaluar más el objeto y se continúa con el siguiente. La trayectoria del objeto en el intervalo de la consulta se evaluará completamente sólo en aquellos casos en que el objeto no entre en r ni en la región universo durante el intervalo de la consulta.

Algorithm 4.7 (**Limited Time Interval**) consulta de time interval donde el intervalo temporal insersecta con una única bitácora

Input: *cst*: una instancia de CST-index, r: región de la consulta, $[t_s, t_e]$: intervalo temporal. *Result*: conjunto de objetos con los resultados parciales previos
Output: *Result*: conjunto de objetos en *cst* que intersectan con r en algún instante de $[t_s, t_e]$ unido con los resultados parciales inciales.

1: **if** $t.start = t.end$ **then**
2: $Result = Result \cup \text{TIMESLICE}(cst, r, t.start)$
3: **end if**
4: $s_i = \text{GETSNAPSHOT}(cst, t_s)$; $t_0 = \text{TIMESTAMP}(s_i)$
5: $t_f = t_0 + f$ $s_{i+1} = \text{GETSNAPSHOT}(cst, t_f)$ $L = \text{GETLOG}(cst, t_s)$
6: $izMD = t_e - t_0$; $derMD = t_f - t_s$
7: $r' = (r.x1 - izMD, r.x2 + izMD, r.y1 - izMD, r.y2 + izMD)$
8: $r'' = (r.x1 - derMD, r.x2 + derMD, r.y1 - derMD, r.y2 + derMD)$
9: **if** $(\text{AREA}(r'') < \text{AREA}(r')) \wedge (s_{i+1} \neq NULL)$ **then**
10: *procesar lado derecho (ver alg. 4.8)*
11: **else**
12: *procesar lado izquierdo (ver alg. 4.9)*
13: **end if**

4.4.4. Obtener la trayectoria de un objeto

En el problema de la trayectoria de un objeto *oid* interesa obtener una secuencia ordenada por el tiempo del par (Punto, Instante) que ha sido registrado para el objeto en el índice *cst* durante un intervalo temporal dado $[t_s, t_e]$.

De modo similar a una consulta de *time interval* (ver 4.4.3), en la consulta de trayectoria el intervalo temporal de la consulta puede intersectar n *snapshots* y se resuelve ejecutando $n + 1$ subconsultas $(Q_0, Q_1, Q_2, \cdots, Qn)$ cuyo resultado es concatenado para formar el resultado global. La primera subconsulta tendrá un intervalo temporal de $[t_s, t_1]$, donde t_1 es el instante del primer *snapshot* que ocurre después de t_s. La segunda subconsulta tendrá el intervalo $[t_1 + 1, t_2]$ donde t_2 es el instante del *snapshot* que ocurre después de t_1 y así se continúa subdividiendo hasta completar todo el intervalo de la consulta con la subconsulta de intervalo $[t_n + 1, t_e]$. Note que los intervalos de las subconsultas parten en el instante siguiente a un *snapshot* y terminan en un *snapshot*, justo lo contrario de las consultas de time interval. Esto se debe a que las bitácoras abarcan dichos intervalos y son las bitácoras las que registran el instante donde ocurre un punto. Así, un punto que está en el *snapshot* t_n, puede haber ocurrido varios instantes antes, pero como es válido aún en t_n ha quedado registrado en el *snapshot*.

Otra diferencia con time interval es que para recuperar la trayectoria de un objeto basta con visitar el último *snapshot* y no los n que intersectan la consulta. Esto es

Algorithm 4.8 Procesar lado Derecho (continuación del alg. 4.7)

```
1: cand = RangeQuery(s_{i+1}, r'')
2: for all c ∈ cand do
3:     if ¬(c ∈ Result) then
4:         particion = getPartitionIz(c, r, izMD)
5:         if particion = INTERIOR then
6:             result = result ∪ {c}
7:         else
8:             l = L[c]
9:             desp = getSumMov(l, t_e − t_0 + 1, f)
10:            esCandidato = 1
11:            t = t_e − t_0 + 1
12:            newMD = izMD
13:            repeat
14:                retrocederObjeto(c, desp)
15:                newMD = newMD − 1
16:                particion = getParticionIz(c, r, newMD)
17:                if  then(particion = UNIVERSO)
18:                    esCandidato = 0
19:                else
20:                    if (particion = INTERIOR) then
21:                        result = result ∪ {c}; esCandidato = 0
22:                    else
23:                        t − −
24:                        d_x = acess(l.right, t) − access(l.left, t)
25:                        d_y = acess(l.down, t) − access(l.up, t)
26:                        desp = (d_x, d_y)
27:                    end if
28:                end if
29:            until ((t == t_s) ∨ (¬esCandidato))
30:        end if
31:    end if
32: end for
```

Algorithm 4.9 Procesar lado Izquierdo(continuación del alg. 4.7)

```
1: cand = RangeQuery(s_i, r')
2: for all c ∈ cand do
3:     if ¬(c ∈ Result) then
4:         particion = getPartitionDer(c, r, derMD)
5:         if particion = INTERIOR then
6:             result = result ∪ {c}
7:         else
8:             l = L[c]
9:             desp = getSumMov(l, 1, ts − t_0)
10:            esCandidato = 1
11:            t = t_s − t_0;
12:            newMD = derMD
13:            repeat
14:                avanzarObjeto(c, desp)
15:                newMD = newMD − 1
16:                particion = getPartitionDer(c, r, newMD)
17:                if  then(particion = UNIVERSO)
18:                    esCandidato = 0
19:                else
20:                    if (particion = INTERIOR) then
21:                        result = result ∪ {c}; esCandidato = 0
22:                    else
23:                        t + +
24:                        d_x = acess(l.right, t) − access(l.left, t)
25:                        d_y = acess(l.down, t) − access(l.up, t)
26:                        desp = (d_x, d_y)
27:                    end if
28:                end if
29:            until ((t == t_e + 1) ∨ (¬esCandidato))
30:        end if
31:    end if
32: end for
```

posible dado que las subconsultas se resuelven visitando la bitácora del objeto sin tener en cuenta los *snapshot*.

En el caso general, la consulta de trayectoria se resuelve llamando a las subconsultas en orden reverso desde Q_n hasta Q_0. En el caso de la subconsulta Q_n, ésta se resuelve de izquierda a derecha, dado que en t_n está la posición conocida más cercana a Q_n. En el resto de las subconsultas se resuelven de derecha a izquierda usando la última posición retornada por las consultas precedentes. Los pasos para obtener la trayectoria en éste se muestran en 4.10.

Algorithm 4.10 obtención de la trayectoria de un objeto

```
1: resp = una trayectoria vacía
2: base = t_n
3: l = getLog(cst, t_q)
4: logCol = getCompactLog(l, oid);
5: snap = getSnapshot(cst, t_n)
6: pos = getObjectPos(snap, oid)
7: if (t_e > t_n) then
8:     getTrajectoryL2R(logCol, oid, pos, base, 1, tn, resp)
9: end if
10: for i = n downto 1 do
11:     base = t_i
12:     l = getLog(cst, t_i)
13:     logCol = getCompactLog(l, oid);
14:     pos = getTrajectoryR2L(logCol, oid, pos, base, 1, f, resp)
15: end for
16: base = t_0
17: l = getLog(cst, t_0)
18: logCol = getCompactLog(l, c);
19: getTrajectoryL2R(logCol, oid, pos, base, t_s, f, resp)
20: return resp
```

Existe un caso especial de consulta de trayectoria, y es aquella que está contenida entre dos *snapshot*. En este caso es posible encontrar la trayectoria deseada recorriendo la estructura de izquierda a derecha (comenzar con s_i) o de derecha a izquierda (comenzando con s_{i+1}). Para tomar la desición se determina el menor número de muestras que hay desde s_i a t_s y desde t_e a s_{i+1}, ver algoritmo 4.11.

Algorithm 4.11 obtención de la trayectoria de un objeto contenida entre dos *snapshot*

1: $resp$ = una trayectoria vacía
2: s_i = snapshot que ocurre antes que t_s
3: s_{i+1} = snapshot que ocurre después que t_e
4: t_0 = instante en que ocurre s_i
5: $logCol$ =bitácora del objeto oid que contiene $[t_s, t_e]$
6: $d1$ = countSamples($logCol, 1, t1$)
7: $d2$ = countSamples($logCol, tn + 1, CST- > f$)
8: **if** $((d2 < d1) \wedge (s_{i+1} \neq NULL)$ **then**
9: pos = getObjectPos(s_{i+1}, oid)
10: $desp$ = getSumMov($logCol, t_e + 1, f$)
11: $pos = pos - desp$
12: getTrajectoryR2L($logCol, oid, pos, t_0, t_s, t_e, resp$)
13: **else**
14: pos = getObjectPos(s_i, oid)
15: $desp$ = getSumMov($logCol, 1, (t_s - t_0) - 1$)
16: $pos = pos - desp$
17: getTrajectoryL2R($logCol, oid, pos, t_0, t_s, t_e, resp$)
18: **end if**
19: **return** $resp$

4.5. Evaluación experimental

En esta sección, se presenta un estudio experimental comparativo de nuestra estructura y el MVR-Tree, tanto desde la perspectiva del almacenamiento como de los tiempos de consulta para time slice y time interval. Es importante señalar que esta comparación se hace a modo de linea base porque el MVR-Tree fue desarrollado con la intención de minimizar los accesos a disco y no pensada como una estructura para memoria principal. Por lo anterior, los datos, el análisis y las conclusiones deben ser valoradas en este contexto.

Para la experimentación se ha utilizado el conjunto de datos reales llamado *imis1month* obtenidos desde el portal chorochronos (`http://chorochronos.datastories.org/?q=node/81`). Estos datos corresponden a 58.691.821 ubicaciones de 4.824 barcos recolectadas en un período de un mes con muestras a cada segundo. Esta colección fue preprocesada para limpiarla de aquellos movimientos que, por la naturaleza de los barcos, no podían ser posibles, corrigiendo estos errores mediante interpolación. Además se realizó una división del espacio en celdas fijas de un tamaño de 7x7 metros aproximadamente, transformando las posiciones desde el formato latitud longitud a las celdas correspondientes en el nuevo espacio.

La evaluación considera también diferentes largos de bitácoras (parámetro f)

que es equivalente al número de instantes entre *snapshots* consecutivos. En el caso del MVR-Tree se han considerado diferentes valores para el tamaño del nodo porque este afecta el tamaño de la estructura y los tiempos de respuesta.

Las consultas fueron tomadas en forma aleatoria de los mismos datos, así se garantiza que al menos exista un resultado por cada una de ellas. Se agruparon las consultas en grupos de 2.000. Cada grupo está determinado por el tamaño del área y el tamaño del intervalo temporal (0 en el caso de time slice). El resultado experimental es el promedio de ejecutar estas 2.000 consultas. Ambos índices fueron ejecutados en memoria principal.

La implementación de MVR-Tree ha sido obtenida de SaIL [HHT05], *a spatial index library*[2]. Tanto el MVR-Tree como nuestra propuesta se compilaron usando la misma versión del compilador de C++ y los experimentos fueron ejecutados en un servidor con 8 procesadores Intel(r) Core(tm) i7-3820 @ 3.60 GHZ con memoria cache L1 de 32 KB, L2 de 256 KB, L3 de 10MB y 32 GB de RAM.

Se analizó la sensibilidad de la estructura para diferentes largos de bitácora y se comparó con el MVR-Tree con diferentes capacidades de nodo para contener claves. En la tabla 4.1 se muestran los diferentes tamaños expresados en MB para ambos índices. Los resultados muestran que para el MVR-Tree, si se aumenta la capacidad del nodo, disminuye el espacio de almacenamiento, pero como se puede observar en la tabla 4.2, el tiempo de respuesta empeora. Por otro lado, en nuestra propuesta se observa que al disminuir el largo de la bitácora el tamaño del índice aumenta. Si bien una bitácora corta ocupa poco espacio, esto hace necesario contar con más *snapshots* que son más costosos si los comparamos con las bitácoras dado que representan tan solo un instante. Al aumentar el largo de las bitácoras, éstas empiezan a ocupar más espacio, pero al requerir menos *snapshots* disminuye el tamaño total del índice. Sin embargo, como se puede observar para el caso de la frecuencia 2.048, el tamaño del índice comienza a incrementarse, lo que se debe al aumento del tamaño de las bitácoras, lo que no es compensado con la disminución del tamaño del conjunto de *snapshots*. Este aumento de tamaño obedece al mayor número de instantes que se representan. Como se comentó al describir las bitácoras, cuando un objeto no se mueve en alguna de las direcciones (arriba, abajo, izquierda o derecha) el *bitmap* no se almacena lleno de ceros, sino que solamente un puntero a nulo. Este ahorro de espacio es muy frecuente cuando la bitácora es corta pero, en el caso de bitácoras largas, es menos probable que el objeto no se mueva y, por lo tanto, no se puede ahorrar el espacio.

Al comparar ambos índices se puede observar que nuestra propuesta consume aproximadamente 5 veces menos espacio que el MVR-Tree para todas las configuraciones representadas y que, además, es menos costosa su construcción.

La tabla 4.2 se presentan los resultados de la ejecución de las consultas sobre los

[2]El código se puede obtener desde `http://libspatialindex.github.com/`

Tabla 4.1: Comparación del tamaño en MB y del tiempo de construcción en minutos considerando diferentes valores para el tamaño de los nodos en el MVR-Tree y diferentes largos de bitácoras para nuestra propuesta.

MVR-Tree			Índice Propuesto		
Capacidad	**Tamaño**	**T. de Carga**	**Frecuencia**	**Tamaño**	**T. de Carga**
15	6.043	25	128	1.177	12
30	5.533	33	256	853	6
45	5.779	37	512	752	3
60	5.597	45	1024	779	2
90	5.093	67	2048	855	2

índices considerando diferentes valores para sus parámetros.

Como se puede observar, en el caso del MVR-Tree en la medida que aumenta la capacidad del nodo también aumenta el tiempo de respuesta del índice.

Algo similar ocurre en el caso de nuestra propuesta. A medida que se aumenta el largo de la bitácora, aumenta también el tiempo de respuesta del índice. Esto se debe a que a medida que aumenta el largo del la bitácora, el área de la consulta que se realiza en el k^2-tree también se debe ampliar, siendo en el peor de los caso del mismo tamaño que el espacio total y, por lo tanto, teniendo que revisar todos los objetos indexados.

Si comparamos el tiempo de procesamiento de las consultas del tipo *time slice* podemos observar que nuestra estructura es mejor al compararla con el índice MVR-Tree cuando el largo de las bitácoras es de 1.024 instantes o menos. Para el caso de las consultas de *time interval* se puede observar que nuestra estructura es mejor si la duración del intervalo temporal es menor a 1.024 y el largo de las bitácoras es de 1.024 instantes o menos.

Para el caso de intervalos de duración 1.024 instantes ocurre que con un área de consulta de 100^2 celdas o menos nuestra propuesta supera al MVR-Tree cuando el largo de las bitácoras es de 512 instantes o menos. Pero cuando el área de la consulta es de $1{,}000^2$ celdas El MVR-Tree con una capacidad de nodo de 15 claves es superior. Esto se debe a que en nuestro índice es necesario realizar y combinar múltiples consultas sobre los *snapshots* que intersectan el intervalo temporal de la consulta, siendo la consulta sobre un *snapshot* una operación costosa.

Aumentar el largo de la bitácora tiene como efecto que se disminuye la cantidad de *snapshot* consultados para responder una consulta con una duración del intervalo grande, pero requiere que se realice una consulta por rango mucho más grande para

Tabla 4.2: Comparación del tiempo de CPU promedio en μ segundos para consultas de *time slice* y *time interval* para distintos tamaños de nodo en el MVR-Tree y largos de bitácoras en nuestra propuesta.

Tipo Consulta		MVR-Tree				**Índice Propuesto**			
Duración	**Área**	**15**	**30**	**60**	**90**	**256**	**512**	**1024**	**2048**
0	$\mathbf{10^2}$	40	61	54	59	**27**	28	31	45
	$\mathbf{100^2}$	41	62	54	59	**27**	28	32	47
	$\mathbf{1000^2}$	46	69	60	64	**34**	38	46	68
64	$\mathbf{10^2}$	61	84	86	107	33	**32**	35	51
	$\mathbf{100^2}$	65	88	89	113	34	**33**	37	53
	$\mathbf{1000^2}$	70	93	94	119	**47**	48	56	80
128	$\mathbf{10^2}$	80	106	117	153	41	**37**	41	57
	$\mathbf{100^2}$	88	113	122	166	42	**39**	43	61
	$\mathbf{1000^2}$	93	117	128	174	67	**66**	74	99
256	$\mathbf{10^2}$	118	149	178	249	60	**54**	60	81
	$\mathbf{100^2}$	134	163	189	274	63	**56**	65	86
	$\mathbf{1000^2}$	139	164	193	283	**118**	121	135	165
512	$\mathbf{10^2}$	195	236	300	438	**100**	110	142	184
	$\mathbf{100^2}$	227	264	322	492	**108**	119	150	193
	$\mathbf{1000^2}$	232	257	324	504	**227**	294	361	429
1.024	$\mathbf{10^2}$	349	409	544	825	**183**	242	471	663
	$\mathbf{100^2}$	410	464	588	929	**198**	262	481	679
	$\mathbf{1000^2}$	**417**	444	585	956	440	665	1.116	1.487

sean incluidos todos los objetos que se deben evaluar. Por esta razón, al ver los datos, un largo de bitácora igual o inferior a 512 instantes ofrece el mejor tiempo.

4.6. Discusión

En este capítulo se ha presentado un índice espacio-temporal que, utilizando estructuras de datos compactas para su implementación, permite responder a las principales preguntas requeridas para una base de datos espacio-temporal.

La evaluación experimental muestra que nuestra propuesta mejora en tiempo y espacio al índice MVR-Tree ejecutado en memoria principal para consultas con una duración inferior a 1.024 instantes.

Sin embargo, las dos restricciones que presenta el índice: 1) no puede haber más de un objeto por celda y 2) que un objeto, si se mueve, lo hace a una celda adyacente, hace que el rango de aplicabilidad del índice se vea reducido a unos contextos donde se puedan garantizar dichas restricciones de manera natural, como por ejemplo los barcos que se mueven en el mar.

Por un lado, la restricción de que los objetos no se puedan mover más allá de una casilla implica que los objetos o bien se mueven muy lento o que exista un sistema capaz de capturar el cambio de posición de los objetos de forma muy rápida, tanto como para garantizar dicha restricción.

Por otro lado, la granularidad espacial debe ser lo suficientemente fina de modo que se garantice la existencia de un único objeto por celda. Si bien esto no es un problema muy grande, dificulta aún más cumplir con la restricción anterior. Además, en algunos contextos esto puede ser innecesario y perfectamente válido contar con celdas lo suficientemente grandes como para abarcar varios objetos, por ejemplo en los sistemas celulares de la telefonía móvil.

Considerando que las limitaciones antes expuestas son un problema para que el índice tenga una aplicabilidad más general, en el siguiente capítulo se presenta una versión del índice que permite responder a las mismas consultas pero que no se ve limitado por el tamaño de las casillas ni por los movimientos de los objetos a celdas adyacentes.

Capítulo 5

Índice para objetos móviles sin restricción de movimiento

5.1. Esquema General

El índice que se presenta en este capítulo es una generalización del índice anterior que no presenta sus restricciones de que exista un único objeto por celda y tampoco que los objetos en dos instantes sucesivos se desplacen a la celda inmediatamente adyacente.

Esta versión del índice implementa las mismas operaciones de la versión precedente: ubicación de un objeto, *time slice*, *time interval*, trayectoria y, adicionalmente, soporta la operación de los k-vecinos más cercanos.

A nivel de esquema general, el índice que a continuación se detalla no presenta ninguna variación. Se mantiene el esquema de llevar un Snapshot cada f instantes y mantener los cambios ocurridos en la ubicación de los objetos en un conjunto o colección de bitácoras como posiciones relativas o desplazamientos respecto a su posición anterior.

Sin embargo, como es necesario que este nuevo índice no esté limitado por las restricciones de la versión anterior, sí existen cambios a nivel de estructuras de datos y algoritmos.

A continuación se explican los cambios tanto a nivel de estructuras de datos como de algoritmos.

5.2. Estructuras de datos

Esta versión del índice mantiene las cuatro estructuras de datos presentes en la versión anterior:

1. ***Snapshot***. Estructura que almacena de manera compacta e indexada la ubicación de cada objeto en un instante de tiempo.
2. **Colección de bitácoras.** Esta estructura agrupa a todas las bitácoras de objetos que estan contenidas en un mismo intervalo temporal.
3. **Bitácora.** Almacena de manera eficiente todas las posiciones que se han registrado para un objeto en un intervalo de tiempo dado.
4. **Índice.** Esta estructura tiene dos secuencias, una para los snapshot y otra para las colecciones de bitácoras.

De estas cuatro estructuras de datos, el índice es el único que se mantiene sin cambios y por esta razón no se presenta en este capítulo.

A continuación se explica esta versión del índice destacando los cambios que se han producido respecto de la versión anterior, tanto en sus estructuras de datos como en los algoritmos.

5.2.1. Snapshot

Al igual que en la versión anterior se utiliza un k^2-tree como estructura de datos principal para indexar la ubicación de los objetos en el espacio. Esto funciona igual que el caso anterior sin cambio.

Como en esta versión es importante permitir que existan en cada celda más de un objeto, es necesario implementar una estrategia para mantener listas de objetos asociados a celdas. Esta estructura, a su vez, debe dar respuesta a la pregunta: dado un objeto, ¿en qué celda se encuentra?.

La estructura de permutación utilizada en la versión anterior permitía asociar un único objeto con una celda. Con la finalidad de implementar la asociación de una lista de objetos a una celda se ha introducido un *bitmap* adicional sobre la permutación para controlar dónde comienza y termina una lista de objetos perteneciente a una celda en particular.

Para ello se concatenan las listas de identificadores de objetos ordenadas por el ID de la celda asociada en un único arreglo con el que se construirá la permutación y se utiliza el *bitmap* adicional, denominado $listMark$, donde se pondrá en 1 el i-ésimo

bit si en el arreglo de la permutación el identificador del objeto de la posición i es el primero de una lista.

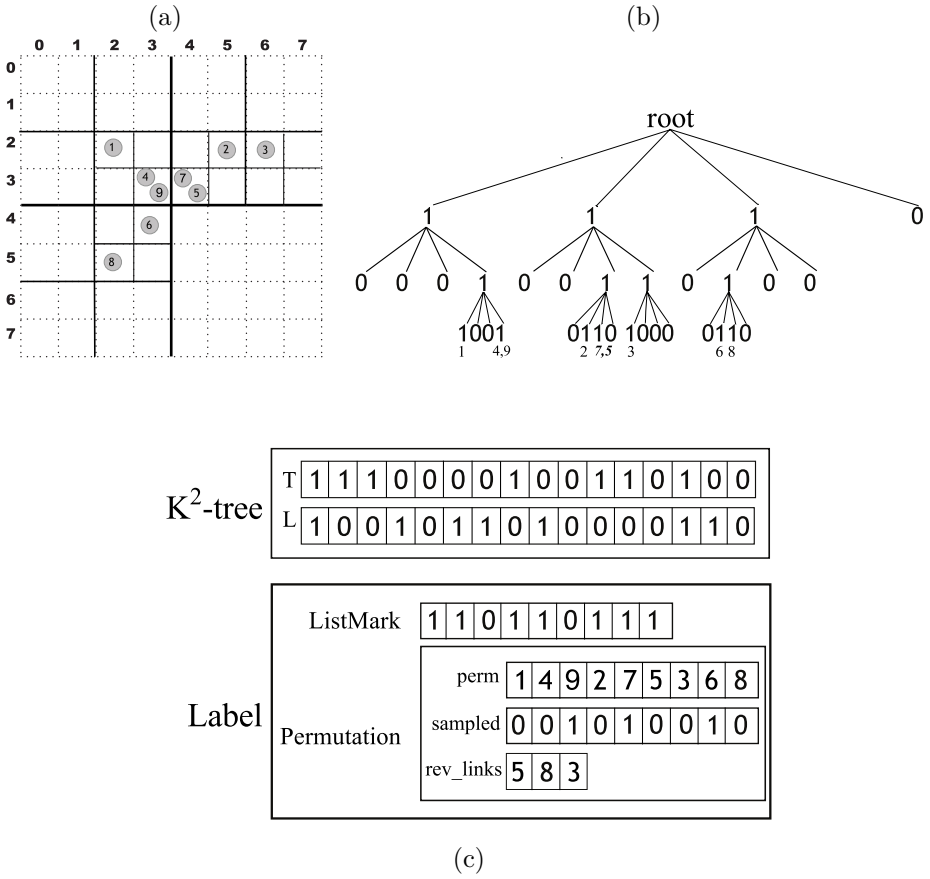


Figura 5.1: Un ejemplo de *snapshot* con $k = 2$: (a) espacio geográfico, (b) k^2-tree conceptual, (c) *snapshot* almacenado con $t = 2$.

Para comprender mejor los cambios en la estructura se presenta el ejemplo de la la Figura 5.1. La parte (a) presenta un espacio de $n = 7$ y $k = 2$ con 9 objetos, dos de los cuales, 7 y 9, comparten celda con 5 y 4 respectivamente. Las líneas gruesas muestran las divisiones del espacio que se hacen en cada nivel del k^2-tree cuando el área no está vacía. En la parte (b) se muestra el k^2-tree conceptual donde se ha agregado a nivel de las hojas una lista con el ID de los objetos que están en la celda representada. Allí se puede observar como las celdas vacías son marcadas con un 0 y luego no se ramifican hasta las hojas, lo que permite un ahorro significativo de espacio.

Finalmente la parte (c) de la imagen muestra cómo es el *snapshot* almacenado para el espacio (a).

Usando esta estructura adicional, es posible obtener la lista de identificadores de objetos contenidos en aquellas hojas de último nivel con valor 1. Para ello se necesita saber $c = \mathsf{rank}_1(L, x)$, porque se usará c como identificador de la ceda con información en la estructura adicional. Luego, calculando $p = \mathsf{select}_1(ListMark, c)$, se obtiene la posición en la permutación ($perm[p]$) donde se encuentra el primer identificador de objeto contenido en c. Si el siguiente bit de $ListMark$ está en 0 significa que $perm[p+1]$ también se encuentra en la celda c. La recuperación de todos los objetos se acaba cuando el siguiente bit en $ListMark$ está en 1, lo que indica que comienza una nueva lista o bien si alcanzamos el final del array $ListMark$. Otro modo de implementar el recorrido sería encontrar la posición de inicio p y término u de la lista y luego recorrer el arreglo $perm$ desde p hasta u así:

$$\begin{aligned} p &= \mathsf{select}_1(ListMark, c) \\ u &= \mathsf{selectNext}_1(ListMark, +1) - 1 \\ output &= \bigcup_{i=p}^{u} \{\pi(i)\} \end{aligned}$$

Esta segunda opción es una buena alternativa para listas largas de objetos, porque evita tener que preguntar cada vez si hemos alcanzado el final de la lista, pero no lo es para listas cortas, pues la operación access es más rápida que la de select. La operación $\mathsf{selectNext}_1$, busca el siguiente 1 desde una posición i en adelante lo que se puede implentar de manera más eficiente.

Para obtener en qué celda se encuentra un determinado objeto dado su identificador (o) se calcula:

$$cellID = \mathsf{rank}_1(ListMark, permutation.\pi^{-1}(o))$$

5.2.2. Colecciones de bitácoras

En la versión del índice para movimientos adyacentes (ver 4.3.2) se utiliza para la colección de bitácoras un vector de largo m, donde m es el número de objetos indexados y en la i-ésima posición del arreglo almacena la bitácora del objeto con id i.

Esta estrategia funciona bien en el caso de movilidades altas, lo que ocurre cuando todos los objetos o la gran mayoría se mueva al menos una vez durante el intervalo cubierto por las bitácoras de la colección, pero a baja movilidad de objetos no es eficiente en el uso del espacio.

Por ejemplo, si se están moviendo p objetos de un total de m, lo óptimo según la teoría de la información sería utilizar un espacio de $p \cdot log_2(m)$ bits. Sin embargo, en el caso de baja movilidad se emplearán $m \cdot log_2(m)$ bits.

Usando la técnica de *hashing* perfecto es posible lograr la misma funcionalidad pero utilizando $p \cdot log_2(m) + m + o(m)$ bits, lo que es muy conveniente en casos de baja movilidad.

Para implementar la técnica de *hashing* perfecto en la colección de bitácoras se necesita agregar a la estructura de datos un *bitmap* h de m bits, donde el i-ésimo bits está en 1 si la bitácora del objeto con id i se encuentra disponible y 0 en caso contrario. Junto con h es necesario mantener el arreglo B que guarda las bitácoras, pero ahora no es necesario que el largo sea m, sino que será de tamaño p, que son las bitácoras que realmente existen.

Para recuperar la bitácora de un objeto con id i primero se revisa si este objeto tiene una bitácora asociada ($access(h, i) == 1$), y en el caso de tenerla, su bitácora se encontrará en la posición $\mathsf{rank}_1(h, i)$ del vector B.

Como se comentó en la versión anterior del índice, al momento de hacer consultas es necesario ampliar el rango de la consulta al momento de realizar sobre un *snapshot* que ocurre i instantes antes que el instante de la consulta real. En el caso anterior para ampliar la consulta bastaba con ampliarla por el número de instantes porque coincidía con el mayor desplazamiento que un objeto pudiera alcanzar.

Para esta nueva versión, como los objetos se pueden mover a celdas adyacentes, es necesario contar con una estrategia diferente. Una posibilidad es almacenar el máximo desplazamiento posible para los objetos como un valor único y global y este valor multiplicarlo por los instantes y con esto obtener el número de celdas para ampliar el rango de la consulta. Sin embargo, esta estrategia si bien es correcta, no consigue el mejor acotamiento en todos los casos. Por ejemplo, en una ciudad la velocidad máxima de desplazamiento de los vehículos no es igual en hora punta que en horario normal o en la noche.

Con el fin de alcanzar un mejor acotamiento se ha optado aquí por una estrategia que consiste en guardar los valores máximos de desplazamiento reales por cada instante cubierto por la colección de bitácoras. Para ello se agregan a la estructura de datos cuatro arreglos de enteros de largo f uno para cada sentido de movimientos: izquierda, derecha, arriba y abajo. Los desplazamientos máximos guardados son valores medidos desde el último *snapshot* anterior a la colección de bitácoras hasta el instante respectivo.

El coste adicional no depende del número de objetos, sólo del número de instantes, por lo que no supone un gran coste si tenemos muchos objetos. Por ejemplo, si por cada bitácora se gastan 4 bits por instante en promedio, el coste de estos cuatro arreglos sería equivalente al de 16 bitácoras, lo que sería un 1 % adicional del coste

de una colección de 1.600 objetos y un 0,1 % en el caso de 16.000 objetos.

5.2.3. Bitácora

Para las bitácoras se presentan dos estructuras de datos distintas, las cuales permiten movimientos más grandes que una celda entre cada movimiento reportado.

En ambos casos las bitácoras codifican movimientos relativos mediante dos técnicas diferentes: códigos unarios y códigos Elías γ.

A continuación se explicará primero la estrategia mediante Códigos Unarios y posteriormente con Elías.

5.2.3.1. Bitácoras en unario

La estructura de datos de las bitácoras en unario mantiene la idea de contar los desplazamientos a celdas adyacentes por cada uno de los sentidos del movimiento (izquierda, derecha, arriba y abajo), por cada movimiento reportado por el objeto. Como en cada movimiento reportado un objeto puede desplazarse más de una celda, se utilizan los códigos unarios para codificar el movimiento.

Los códigos unarios son códigos de largo variables que permiten representar un número entero positivo v usando v unos seguidos de un cero [GHSV06]. Por ejemplo, para codificar el número 5, este en unario sería 111110. Los códigos unarios son óptimos en el caso que la probabilidad de ocurrencia de cada número a codificar sea $P(x) = 2^{-x}$, donde x es el número natural que se está codificando.

Con la idea de codificar eficientemente colecciones de objetos a baja movilidad se agrega un *bitmap* llamado *mov* para codificar el tiempo en el cual se ha reportado un movimiento. El *bitmap mov* tiene f bits donde el i-ésimo bit está en 1 cuando en el instante i tenemos un movimiento codificado en la bitácora, de otro modo es 0. Note que el instante será un valor entre 1 y f, este instante es relativo al instante en que ocurre el último snapshot. Desde el punto de vista del almacenamiento esto permite codificar con 1 bit aquellos instantes en los cuales el objeto no se ha movido en vez de 4 bits, para indicar 0 en cada uno de los sentidos del movimiento.

Para explicar cómo se crea una bitácora y cómo recuperar una posición relativa se presenta el siguiente ejemplo:

$$\begin{aligned} t &= \{3, 9, 15, 21, 27\} \\ \Delta &= \{(1, -5), (4, 3), (-1, -10), (-8, -15), (-1, -20)\} \end{aligned}$$

Donde t es la secuencia de instantes en los cuales se ha reportado un movimiento y Δ es la secuencia de posiciones relativas del objeto. En este ejemplo $f = 30$.

La bitácora codificada en unario para el ejemplo anterior sería la siguiente:

$$\begin{aligned} mov &= \{00100000100000100000100000001000\} \\ derecha &= \{1011110000\} \\ izquierda &= \{001011111111010\} \\ arriba &= \{01110000\} \\ abajo &= \{11111001111111111011111111111111101111111111111111111110\} \end{aligned}$$

Como se puede observar en *mov* están en 1 los bits 3°, 9°, 15°, 21° y 27°, porque en esos instantes es en los que existen movimientos reportados.

Al igual que en el caso de las bitácoras con movimientos adyacentes, cada posición relativa se codifica según lo que se ha desplazado en cada sentido del movimiento. Para la primera posición $(1, -5)$, significa que el objeto se ha movido 1 casilla a la derecha, 0 a la izquierda, 0 arriba y 5 a la derecha. Lo que se codifica como 10 en derecha, 0 en izquierda, 0 arriba y 111110 abajo. Cada posición relativa es codificada concatenando el resultado al *bitmap* respectivo siguiendo el mismo orden de ocurrencia temporal del movimiento.

La principal pregunta que se realiza sobre la bitácora es poder conocer cuánto se ha desplazado un objeto hasta un instante t_q dado, lo cual es necesario para poder recuperar la posición real del objeto. Esto significa obtener la suma de las posiciones relativas(Δ_x, Δ_y) que ocurren hasta el instante de la consulta. Δ_x se calcula como la diferencia entre el total de celdas que el objeto se desplazó hacia la derecha y el total de celdas hacia la izquierda desde el inicio del log hasta t_q; y Δ_y como la diferencia entre el total de celdas que se desplazó hacia arriba y el total de celdas que se desplazó hacia abajo desde el inicio del log hasta t_q. Por lo tanto el problema se reduce a encontrar la cantidad de celdas que el objeto se desplazó hasta el instante de la consulta en cada uno de los sentidos del movimiento.

Para obtener lo anterior primero hay que conocer cuántas posiciones relativas se tiene hasta el instante de la consulta, con la finalidad de saber hasta donde hay que sumar. Esto se hace contando la cantidad de unos que hay en *mov* hasta el instante de la consulta t_q (una operación de *rank*). Siguiendo con el ejemplo anterior, para conocer el desplazamiento del objeto hasta el instante 20 tendríamos que calcular $\mathsf{rank}_1(mov, 20)$ lo que nos da 3.

Una vez conocida la cantidad de posiciones relativas que se deben sumar, calculando $\mathsf{select}_0(b, i) - i$, se obtiene la cantidad de celdas que el objeto se desplazó en el sentido del movimiento b hasta la i-ésima posición relativa o lo que es lo mismo el instante t_q. Lo anterior funciona porque los códigos unarios codifican con 1 bit

cada celda que el objeto se ha desplazado, de esta manera $\mathsf{select}_0(b, i)$ entregará la posición p del *bitmap* donde ocurre el i-ésimo 0, como sabemos que de p bits i son 0, la diferencia es la cantidad de unos.

Siguiendo con el ejemplo anterior, si $i = 3$, entonces la cantidad de celdas desplazadas hacia arriba (cantidad de unos) serán $\mathsf{select}_0(arriba, 3) - 3 = 6 - 3 = 3$.

Usar unarios es la opción más natural para transitar desde las bitácoras que codifican movimientos adyacentes a este modelo en que los movimientos no lo son, pero puede ser ineficiente en el uso del espacio si los movimientos a casillas adyacentes son valores muy altos. En estos casos se utilizan *bitmaps* comprimidos con soporte de *rank* y *select*, específicamente *RRR* o *SDArray*. Así, algunas bitácoras en el índice pueden utilizar *bitmaps* sin comprimir, otras *bitmaps* comprimidos con *RRR* y otras con *SDArray* según convenga.

En los experimentos se compara la diferencia entre usar *bitmap* sin comprimir en todas las bitácoras o bien seleccionar el tipo de *bitmap* que más convenga.

Análisis del espacio Una bitácora almacena una secuencia de t posiciones relativas y su instante relativo. Por cada coordenada de una posición relativa (x, y) será codificada en unarios y puesta en el *bitmap* correspondiente a la dirección del movimiento y en los otros dos *bitmaps* que no fueron utilizados se pondrá un bit 0 en cada uno indicando que en esa dirección no se ha movido. Como por cada número p codificado con unario se requieren $p + 1$ bits, por cada posición relativa se necesitan $(x + 1) + (y + 1) + 2 = x + y + 4$ bits. Para codificar la secuencia $S = (x_1, y_1), (x_2, y_2), (x_3, y_3) \dots (x_t, y_t)$ de posiciones relativas se necesitan $\sum_{i=1}^{t}(x_i + y_i + 4) = \sum_{i=1}^{t} x_i + \sum_{i=1}^{t} y_i + 4t$, con el fin de simplificar la expresión se define $n_x = \sum_{i=1}^{t} x_i$ y $n_y = \sum_{i=1}^{t} y_i$. Con la sustitución llegamos a $n_x + n_y + 4t$ bits.

Como necesitamos manipular eficientemente los 4 *bitmaps* que codifican las coordenadas, necesitamos contar con soporte para las operaciones de *rank* y *select* en tiempo constante se requiere un coste adicional de $o(n)$ bits por *bitmap* siendo n el largo de la secuencia de bits.

Junto con lo anterior se requieren f bits para el *bitmap mov* porque necesita un bit por cada instantes relativo. Como en *mov* necesitamos soporte para *rank* y *select* se necesita $o(f)$ bits adicionales.

En total la estructura requiere $n_x + n_y + 4t + f + o(n_x + n_y + t + f)$ bits de espacio para codificar t posiciones relativas.

Si asumimos que existe un valor máximo u para las coordenadas, y que en el peor de los casos existen f posiciones relativas que son el máximo desplazamiento (u, u), entonces la bitácora requiere: $(2u + 5)f + o(fu)$ bits.

5.2.3.2. Bitácoras con Códigos Elías

Para la codificación de las posiciones relativas en esta estrategia se han utilizado los códigos Elías, en particular los códigos gamma (γ), dado que no sabemos cuántos movimientos serán posibles como máximo, pero sí sabemos que en su mayoría son cortos como se puede observar en la tabla 5.1.

Los códigos γ pueden codificar cualquier número entero mayor a 0 ($\mathbb{N}$). Estos códigos no requieren tablas de frecuencia o probabilidades, simplemente asumen que los más pequeños son más frecuentes y, por lo tanto, le asignan menos bits.

Para codificar un número $x \geq 1$ lo primero es determinar la mayor potencia de 2 que es menor o igual a x, es decir, $n = \lfloor log_2(x) \rfloor$, de modo que $2^n \leq x < 2^n + 1$. Luego se escriben n bits en cero seguido del valor binario de x que es un número de $n + 1$ bits. Por ejemplo, si queremos codificar el número 15, calculamos primero $n = \lfloor log_2(15) \rfloor = 3$, luego anteponemos 3 ceros al valor en binario de 15 que es 1111 y, por lo tanto, el código γ resultante es 0001111. En la tabla 5.1 se pueden observar como se codifican los números del 1 al 17 usando códigos γ.

Como se puede observar en las columnas Δ_x y Δ_y de la tabla 5.1 las coordenadas de las posiciones relativas son valores tanto positivos como negativos ($\mathbb{Z}$), pero los códigos γ no permiten números negativos y tampoco el cero. Esta falta de correspondencia requiere encontrar una manera de convertir un valor de $\mathbb{Z}$ a un valor de $\mathbb{N}$ y que sea reversible.

Para lograr lo anterior, primero se utilizará una función biyectiva $f(x)$ para mapear un valor de $\mathbb{Z}$ a uno de $\mathbb{N}_0$ y la función $f^{-1}(y)$ para lo contrario. Para ello se reordena $\mathbb{Z}$ intercalando los valores positivos y negativos generando la secuencia $\{0, -1, 1, -2, 2, -3, 3, \dots\}$ la cual se mapea con la secuencia de $\mathbb{N}_0 = \{0, 1, 2, 3, 4, 5, 6, 7, \dots\}$ uno a uno en orden, $(0 \mapsto 0, -1 \mapsto 1, 1 \mapsto 2, 2 \mapsto 3 \dots)$.

$$f(x) = \begin{cases} -2x - 1 & \text{si } x < 0 \\ 2x & \text{si } x \geq 0 \end{cases} \qquad f^{-1}(y) = \begin{cases} (-y-1)/2 & \text{si } y \text{ es impar} \\ y/2 & \text{si } y \text{ es par} \end{cases} \tag{5.1}$$

Esta técnica conocida como *ZigZag Encoding*, la podemos ver como un ordenamiento de pares e impares, de modo que el signo indica si es par(+) o impar(-) y luego el número en valor absoluto representa la posición que ocupa en la secuencia de pares o impares.

Por ejemplo, para -2, su correspondencia es el 2° impar es decir 3 ($f(-2) = -2 \cdot -2 - 1 = 3$) y para el 2 sería el 2° par, es decir 4 (recuerde que por definición el 0 no es par). Para invertir el proceso hay que tener en cuenta si el número es par o impar, por ejemplo, con 4 calculamos $f^{-1}(4) = 4/2 = 2$ y con 3 $f^{-1}(3) = (-3 - 1)/2 = -2$.

Tabla 5.1: Trayectoria de un objeto indicando instante (t), posición absoluta (x, y) y posición relativa (Δ_x, Δ_y)

t	x	y	Δ_x	Δ_y
0	486253	4207588	0	0
3	486261	4207543	8	-45
6	486292	4207562	31	19
9	486289	4207473	-3	-89
12	486226	4207270	-63	-203
15	486225	4207094	-1	-176
18	486233	4206933	8	-161
21	486276	4206825	43	-108
24	486292	4206720	16	-105
27	486253	4206701	-39	-19
30	486253	4206701	0	0
33	486212	4206704	-41	3
36	486203	4206706	-9	2
39	486203	4206706	0	0
42	486165	4206710	-38	4
45	486155	4206712	-10	2
48	486151	4206715	-4	3
51	486141	4206703	-10	-12
54	486186	4206573	45	-130
57	486210	4206519	24	-54

Tabla 5.2: Ejemplo de codificación de números del 1 al 19 con códigos γ

Número	binario	γ
$1 = 2^0 + 0$	1	1
$2 = 2^1 + 0$	1 0	0 1 0
$3 = 2^1 + 1$	1 1	0 1 1
$4 = 2^2 + 0$	1 00	00 1 00
$5 = 2^2 + 1$	1 01	00 1 01
$6 = 2^2 + 2$	1 10	00 1 10
$7 = 2^2 + 3$	1 11	00 1 11
$8 = 2^3 + 0$	1 000	000 1 000
$9 = 2^3 + 1$	1 001	000 1 001
$10 = 2^3 + 2$	1 010	000 1 010
$11 = 2^3 + 3$	1 011	000 1 011
$12 = 2^3 + 4$	1 100	000 1 100
$13 = 2^3 + 5$	1 101	000 1 101
$14 = 2^3 + 6$	1 110	000 1 110
$15 = 2^3 + 7$	1 111	000 1 111
$16 = 2^4 + 0$	1 0000	0000 1 0000
$17 = 2^4 + 1$	1 0001	0000 1 0001
$18 = 2^4 + 2$	1 0010	0000 1 0010
$19 = 2^4 + 3$	1 0011	0000 1 0011

Luego de llevar el valor $\mathbb{Z}$ a $\mathbb{N}_0$ es necesario llevarlo a $\mathbb{N}$ antes de codificar con Elías γ, pues este no codifica el 0. Para ello, simplemente se suma 1 al valor a codificar, el cual se restará al decodificar.

En síntesis, para codificar el valor de una de las coordenadas c de una posición relativa con códigos Elías γ se calcula $c' = cod(f(c) + 1)$ y se decodifica con $c = f^{-1}(decod(c') - 1)$.

Las funciones f y f^{-1} se pueden implementar fácilmente utilizando una estructura de control `if-then-else` para chequear la condición y luego realizar el cálculo adecuado, sin embargo, utilizando operaciones de bits se puede definir una fórmula que calcule lo mismo sin la necesidad de realizar comparación alguna lo que será mucho más eficiente en las actuales arquitecturas [Lem12] [1].

En esta versión la estructura de datos de la bitácora está compuesta por tres *bitmaps*: uno para codificar mediante códigos γ la primera coordenada de las posiciones relativas (Δ_x), otro para la segunda coordenada (Δ_y) y uno para marcar en qué instante ocurre un movimiento (mov). mov, al igual que en caso de las bitácoras con codificadas en unario, tiene dos propósitos, el primero es disminuir el almacenamiento necesario para codificar aquellos instantes en los que el objeto no se mueve, gastando un bits en vez de dos; y el segundo, disminuir el tiempo de procesamiento al procesar menos posiciones relativas.

Por ejemplo, consideremos el intervalo $[1, 30]$ de la tabla 5.1 y las columnas t, Δ_x y Δ_y cuyos valores son:

$$\begin{aligned} t &= \{3, 6, 9, 12, 15, 18, 21, 24, 27, 30\} \\ \Delta_x &= \{8, 31, -3, -63, -1, 8, 43, 16, -39, 0\} \\ \Delta_y &= \{-45, 19, -89, -203, -176, -161, -108, -105, -19, 0\} \end{aligned}$$

Si consideramos que el largo de la bitácora son 30 instantes, se codificaría el intervalo $[1, 30]$ en la bitácora con el *bitmap* mov, poniendo en 1 el i-ésimo bit para todo i en t. Luego se codifica cada entero x en $\Delta_x[1, 30]$ usando códigos γ sobre $f(x)$ como se explicó más arriba. Finalmente se realiza lo mismo para Δ_y. El resultado

[1] Para un entero x de 32 bits, el código en C++ que calcula $y = f(x)$ y $x = f^{-1}(y)$ es:

```
y=(x << 1) ^ (x >> 31)
x=((unsigned int)y >> 1) ^ ( (y << 31) >> 31 )
```

final sería:

$$
\begin{aligned}
mov &= 00100100100100100100100100100 1 \\
\gamma_x &= \{0000100010000011111100110000000111111001 0 \\
&\quad 0000100010000001010111000001000010000001001110 1\} \\
\gamma_y &= \{0000001011010000001001110000000101100 1 0 \\
&\quad 0000000011001011000000000101100000 \\
&\quad 0000000010100001000000001101100000000001101001000000100110 1\}
\end{aligned}
$$

Una operación básica con las bitácoras es conocer el desplazamiento de un objeto hasta un cierto instante t (δ_x, δ_y) con la finalidad de poder recuperar la posición del objeto a t conociendo la posición del objeto ocurrida en el snapshot inmediatamente anterior a t.

Por ejemplo, si queremos obtener la posición del objeto al instante $t = 10$, primero calculamos el número de posiciones relativas que tiene el objeto hasta t lo que corresponde a la cantidad de unos que hay en mov hasta 10 lo que calculamos: $\mathsf{rank}_1(mov, 10) = 3$. Eso significa que debemos decodificar y sumar los 3 primeros números de la secuencia comprimida, tanto en γ_x como en γ_y.

Para decodificar un código γ primero hay que leer y contar los ceros (0) hasta encontrar el primer 1. Se considera el 1 que se acaba de leer como el 1 bit del entero que se está recuperando y se procede a leer los n dígitos restantes, siendo n la cantidad de ceros que acabamos de contar. Luego, los $n+1$ bits últimos recuperados se convierten a binario. Por ejemplo, en el código γ_x contamos los ceros que hay hasta leer el primer 1, esto es 4, luego leemos los cuatro siguientes bits que son 0001, por lo tanto el entero en binario que hemos recuperado es 10001 que corresponde al número 17.

Una vez que hemos decodificado el número, hay que calcular $f^{-1}(decod(c') - 1)$ para obtener el número original. Siguiendo con el ejemplo, para el valor decodificado 17 su valor original es $f^{-1}(17-1) = 16/2 = 8$. Repetimos el proceso de decodificación desde el bit siguiente al último que hemos leído, en este caso desde el bit 10. Leemos 5 ceros, obtenemos el número en binario 111111, que es el decimal 63 y que corresponde al número 31 luego de calcular $f^{-1}(63 - 1)$. Repetimos el proceso por última vez, leemos 3 ceros, obtenemos el número en binario 110, que es el decimal 6 y que corresponde a -3 ($f^{-1}(6 - 1) = (-5 - 1)/2 = -6/2 = -3$). Finalmente obtenemos la suma de los tres números la cual se fue acumulando con los resultados de cada iteración, que en este caso sería $\delta_x = 36$. Siguiendo los mismos pasos se obtiene $\delta_y = -115$.

Análisis del espacio Para almacenar t posiciones relativas y sus respectivos instantes, esta versión de las bitácoras requiere dos secuencias de bits γ_x y γ_y que no requieren soporte de rank y select y un *bitmap mov* que al igual que en las bitácoras en unario requiere $f + o(f)$ bits porque requiere soporte para *rank* y *select*.

Un número x codificado con Elías γ requiere $2\lfloor \log_2(x) \rfloor + 1$ bits y es óptimo cuando x tiene la probabilidad $P(x) = 1/2x^2$.

Por lo anterior, para codificar las secuencia $S = (x_1, y_1), (x_2, y_2), (x_3, y_3) \dots (x_t, y_t)$ posiciones relativas se necesitan:

$$2\sum_{i=1}^{t} (\lfloor \log_2(x_i) \rfloor) + 2\sum_{i=1}^{t} (\lfloor \log_2(y_i) \rfloor) + 2t$$

y el espacio total para esta bitácora sería:

$$2\sum_{i=1}^{t} (\lfloor \log_2(x_i) \rfloor) + 2\sum_{i=1}^{t} (\lfloor \log_2(y_i) \rfloor) + 2t + f + o(f)$$

Si asumimos que existe un valor máximo u para una coordenada, y que en el peor de los casos existen f posiciones relativas que son el máximo desplazamiento (u, u), entonces la bitácora requiere: $(4\lfloor \log_2(u) \rfloor + 3)f + o(f)$ bits.

5.3. Algoritmos

En esta sección de la tesis presentamos las modificaciones a los algoritmos anteriores, que en términos generales se mantienen de forma muy similar, dado que el índice sigue el mismo esquema, pero con algunos cambios pequeños en las operaciones que recuperan dados de los *snapshots* y las bitácoras debido a los cambios en las estructuras de datos antes descritos.

Por esta razón, se mantiene la estructura del capítulo anterior presentando únicamente los algoritmos que cambian.

5.3.1. Obtener la ubicación de un objeto

Al igual que en la versión anterior, para obtener la posición de un objeto en un instante en particular se pueden dar dos situaciones o casos: 1) cuando el instante de la consulta ocurre en un snapshot, 2) cuando el instante de la consulta intersecta una bitácora.

5.3.1.1. Obtener la ubicación de un objeto cuando el instante de la consulta ocurre en un Snapshot

Al igual que en la versión anterior, el proceso necesario para obtener la posición asociada a un objeto o en un snapshot s se obtiene mediante dos algoritmos: FindPath() (algoritmo 5.1) y getObjectPos() (algoritmo 4.2). Como se comentó en el capítulo anterior, FindPath() primero encuentra la posición en el *bitmap* de las hojas del k^2-tree la celda donde se encuentra el objeto, luego a partir de esta posición asciende por el árbol hasta la raíz visitando el padre de la celda y devolviendo en una pila el recorrido. De este modo, FindPath() entrega el camino desde la raíz a la celda. El segundo algoritmo getObjectPos() toma este camino y lo recorre poniendo especial atención en los límites del espacio que representa cada celda, de modo que comenzamos con los límites del espacio completo en la raíz y llegamos a una celda atómica en la hoja, que representa la posición del objeto en el espacio.

En este proceso solo cambia la manera en que se determina la celda donde está contenido el objeto que se está buscando, que es un paso del algoritmo 5.1 FindPath(). El cambio se debe a que en esta versión del índice hay más de un objeto por celda, lo que lleva a tener listas de objetos y por ende la permutación no mapea directamente la posición de la celda, sino que indica la posición que ocupa el objeto en la concatenación de las listas.

Para saber a qué celda (o lista) pertenece un objeto se debe usar el *bitmap* $ListMark$. La línea 2 es donde se realiza este paso y, por lo tanto,

fue necesario modificarla. Como $ListMark$ indica en qué posición comienza una lista de id de objetos en la permutación al realizar la operación $p \leftarrow rank_1(ListMark, s.permutation.\pi^{-1}(o))$, se obtiene la celda a la que pertenece el objeto o.

El algoritmo 4.2 GETOBJECTPOS() no tiene ningún cambio. Así mismo, las operaciones básicas que dado un nodo permiten obtener su i-ésimo hijo, la posición del padre y el orden de hijo que le corresponde tampoco cambian. Estas están descritas en la sección 4.4.1.

Algorithm 5.1 (**FindPath**), Obtiene el camino desde la raíz a la hoja en el *snapshot* que contiene un objeto especificado como parámetro

Entrada: Snapshot s, OID o.
Salida: Stack.
1: $path \leftarrow$ NEW($Stack$)
2: $p \leftarrow rank_1(ListMark, s.permutation.\pi^{-1}(o))$
3: $x =$ SELECT($s.L, p$) + LENGTHINBITS($s.T$)
4: **while** $x > 0$ **do**
5: $\quad i =$ I(x)
6: $\quad path =$ PUSH($path, i$)
7: $\quad x =$ PADRE(x)
8: **end while**
9: **return** $path$

5.3.1.2. Obtener la ubicación de un objeto cuando el instante de la consulta intersecta una bitácora

Para obtener la posición de un objeto *oid* en un instante t_q que intersecta con una bitácora l en particular se mantiene la estrategia general de la versión anterior, pero como tenemos más de una manera de implementar las bitácoras, se ha simplificado el algoritmo delegando la responsabilidad de calcular el desplazamiento de un objeto desde su posición en el snapshot hasta t instantes más adelante con la función GETSUMMOV(l, t) → Punto que es implementada de manera adecuada según el tipo de bitácora.

El algoritmo 5.2 es la versión simplificada. En este algoritmo lo primero que se hace es encontrar el snapshot s_i, el cual posee el instante mayor de los menores o iguales que t_q (línea 1). Con s_i se obtiene la posición absoluta *pos* del objeto *oid* en el instante en que ocurre el snapshot t_0(línea 5).

Luego se obtiene la colección de bitácoras l_i cuyo intervalo temporal contiene al instante t_q. En esta colección se encuentra la bitácora del objeto *oid* que contiene

los movimientos relativos ocurridos desde el instante t_0 al instante t_q. Más adelante se necesitará esta bitácora.

Como en la colección de bitácoras los instantes son relativos al snapshot anterior, se obtiene t que es el instante t_q relativo a t_0 (línea 3).

Si $t = 0$, significa que el instante de la consulta coincide con el instante del snapshot s_i y, por lo tanto, no es necesario actualizar el punto y *pos* es la respuesta. Por el contrario si $t > 0$, es necesario actualizar el punto *pos* lo que se realiza en la línea 7 sumando su desplazamiento hasta t_q. Para obtener el desplazamiento del objeto se utiliza la función GETSUMMOV(), donde el primer parámetro corresponde a la bitácora del objeto *oid* contenida en la colección de bitácoras l_i y el segundo es el instante t.

Algorithm 5.2 getPosition

Input: *cst*: una instancia del índice (CST), *oid*: identificador del objeto buscado, t_q: instante consultado.
Output: Punto : Punto con la ubicación del objeto *oid* en el instante t_q indexado en *cst*.

1: s_i = GETSNAPSHOT(cst, t_q)
2: t_0 = TIMESTAMP(s_i)
3: $t = t_q - t_0$
4: l_i = GETLOGCOLECTION(cst, t_q)
5: pos = GETOBJECTPOS(s_i, oid)
6: **if** $(t > 0)$ **then**
7: $\quad pos = pos +$ GETSUMMOV($L_i.getLog(oid), t$)
8: **end if**
9: **return** *pos*

La operación GETSUMMOV() devuelve $(0, 0)$ en el caso que el objeto no se haya movido hasta el instante t_q, de otro modo se debe recuperar el desplazamiento del objeto, lo que corresponde a la suma de todas las posiciones relativas registradas hasta el instante de la consulta. El cómo obtener el desplazamiento del objeto varía según como esté codificada la bitácora.

En el caso de la codificación con Elías γ hay que decodificar y sumar cada posición relativa que tenga la bitácora hasta el instante de la consulta y, por lo tanto, el tiempo es directamente proporcional a la cantidad de posiciones relativas decodificadas.

En cambio, en el caso de la codificación en unario no es necesario decodificar todas las posiciones relativas, dado que es posible obtener la suma de todas ellas hasta el instante de la consulta en tiempo constante con cuatro operaciones de *select* y una de *rank*, siguiendo los pasos descritos y ejemplificados en la sección 5.2.3.1.

5.3.2. Time Slice

Para responder a una consulta de time slice se consideran dos casos, al igual que en la versión anterior: 1) cuando el instante de la consulta t_q coincide con el instante de un *snapshot* y 2) cuando t_q se encuentra entre dos snapshot s_i y s_{i+1} (ver 5.3.2.2).

A continuación se presenta el primer caso.

5.3.2.1. Consultas por rango espacial en un Snapshot

La consulta por rango espacial en el *snapshot* se responde con una llamada a la operación de *Rango* del k^2-tree a la cual se le ha hecho una pequeña modificación al llegar a un nodo hoja de último nivel de modo que ahora se recuperará la lista de identificadores de objetos que están asociados a esa celda y a cada uno de ellos se le asocia como punto los valores (x, y) de la celda asociada. El algoritmo 5.3 es la versión con las modificaciones (que se encuentran desde la línea 4 a la 7) al algoritmo presentado en [Lad11].

5.3.2.2. Time slice entre dos snapshots

En esta versión del algoritmo hay dos cambios fundamentales. El primero está relacionado con la ampliación de rango pues ahora es necesario ampliar la consulta un número mayor de celdas por instantes, pues los desplazamientos no son adyacentes. Y el segundo afecta a las bitácoras codificadas con Elías γ, que dada su naturaleza, no es posible leer la secuencia comprimida en forma reversa, lo que impide procesar la bitácora en ambas direcciones.

Si bien las bitácoras en unario pueden ser decodificadas en sentido reverso, por simplicidad, se presenta aquí una versión del time slice en un solo sentido que sirve tanto para codificación con Elías como en unario.

El algoritmo 5.4 presenta los cambios que se han descrito.

Algorithm 5.3 (**Range**)$(n, p_1, p_2, q_1, q_2, d_p, d_q, z, output)$

```
1: T = k²-tree.T; L = k²-tree.L
2: if z ≥ |T| then
3:     if L[z − |T|] = 1 then
4:         cell = Rank(T, z − |T|)
5:         p = select_1(ListMark, cell)
6:         u = select_1(ListMark, cell + 1) − 1
7:         output = output ∪ ⋃_{i=p}^{u} {π(i) × {(d_p, d_q)})}
8:     end if
9: else
10:    if z = −1 ∨ T[z] = 1 then
11:        y = rank(T, z) · k²
12:        for i = ⌊p_1/(n/k)⌋ ... ⌊p_2/(n/k)⌋ do
13:            p'_1 = 0
14:            if i = ⌊p_1/(n/k)⌋ then
15:                p'_1 = p_1 mód (n/k)
16:            end if
17:            p'_2 = (n/k) − 1
18:            if i = ⌊p_2/(n/k)⌋ then
19:                p'_2 = p_2 mód (n/k)
20:            end if
21:            for j = ⌊q_1/(n/k)⌋ ... ⌊q_2/(n/k)⌋ do
22:                q'_1 = 0
23:                if j = ⌊q_1/(n/k)⌋ then
24:                    q'_1 = q_1 mód (n/k)
25:                end if
26:                q'_2 = (n/k) − 1
27:                if j = ⌊p_2/(n/k)⌋ then
28:                    p'_2 = p_2 mód (n/k)
29:                end if
30:                d'_p = d_p + (n/k) · i
31:                d'_q = d_q + (n/k) · j
32:                z' = y + k · i + j
33:                RANGE(n/k, p'_1, p'_2, q'_1, q'_2, d'_p, d'_q, z', output)
34:            end for
35:        end for
36:    end if
37: end if
```

Algorithm 5.4 Time Slice

Input: *cst*: una instancia de CST-index, r: región de la consulta, t_q: instante
Output: conjunto de objetos en *cst* que intersectan r en t

```
1: result = ∅
2: s_i = getSnapshot(cst, t_q)
3: t_0 = timeStamp(s_i)
4: if t_0 = t_q then
5:     return RangeQuery(s_i, r)
6: end if
7: L = getLog(cst, t_q)
8: MD = getMaxDesp(L, t_q)
9: r' = RegionExterior(r, MD)
10: r'_i = RegionInterior(r, MD)
11: cand = RangeQuery(s_i, r')
12: for all c ∈ cand do
13:     if c in r'_i then
14:         result = result ∪ {c}
15:     else
16:         l = L.getLog(c)
17:         if timeSliceTest(l, c.pos, t_q, r, MD) then
18:             result = result ∪ {c}
19:         end if
20:     end if
21: end for
22: return result
```

Este algoritmo comienza por obtener el Snapshot s_i, que es el primer snapshot antecesor del instante t_q, o bien ocurre en t_q. Si s_i ocurre en t_q, la consulta se resuelve directamente en el snapshot (linea 5). Si no es así, entonces hay que realizar una consulta de rango ampliada en s_i.

Para ampliar la consulta por rango es necesario conocer el máximo desplazamiento que algún objeto realizó desde s_i hasta t_q. Este desplazamiento máximo se encuentra en la colección de bitácoras L que es la única colección que contiene al instante de la consulta t_q.

Para recuperar el desplazamiento máximo se utiliza la operación getMaxDesp(L, tq) (línea 8), la cual devuelve un *array* con 4 enteros que representan al desplazamiento máximo hacia la izquierda, derecha, abajo y arriba.

Con estos desplazamientos se definen dos regiones: la región exterior r' (línea 9) y la región interior r'_i (línea 10).

La región exterior r' corresponde al área del espacio donde podría existir un objeto en s_i que dado MD se pueda desplazar en los instantes sucesivos ($[t_0 \dots t_q]$)

a la región de la consulta r. La región exterior se define como $(r.x1 - MD[1], r.x2 + MD[0], r.y1 - MD[3], r.y2 + MD[2])$.

La región interior r'_i corresponde al área del espacio donde un objeto en los instantes sucesivos dado MD no podría desplazarse fuera del área de la consulta. La región interior se define como $(r.x1 + MD[0], r.x2 - MD[1], r.y1 + MD[2], r.y2 - MD[3])$.

Utilizando r' como región se realiza la consulta por rango sobre el snapshot s_i (línea 11). La correcta definición de r' garantiza que ningún objeto que tenga posibilidades de estar en r en el instante t_q sea excluido de los resultados, pero agrega a otros objetos que en t_q no lo serán.

Por lo anterior es necesario revisar a cada candidato para determinar si son efectivamente parte de los resultados o no. Lo primero que se evalúa por cada candidato es determinar si se encuentra en r'_i (línea 13), lo que quiere decir, que el objeto será parte de los resultados independientemente de su ubicación en el instante t_q. Por el contrario, el objeto puede ser parte de los resultados o no lo que obliga a revisar su ubicación real al instante t_q.

Para saber si el objeto será parte o no de los resultados se utiliza la función booleana TIMESLICETEST() que dará verdadero si el objeto se encuentra en r en el instante t_q o falso en caso contrario. Como se observa en la línea 17, la función recibe como parámetro la bitácora l del objeto candidato, la posición del objeto candidato *c.pos*, el instante de la consulta t_q, la región de la consulta r, y el máximo desplazamiento MD.

La función TIMESLICETEST() varía según el tipo de implementación de la bitácora.

En el caso de las bitácoras en unario se actualiza la ubicación del objeto hasta el instante t_q ($p = c.pos$+GETSUMMOV(l, tq)), y luego se retorna verdadero o falso dependiendo si p está dentro de r o no. Esta operación es eficiente, dado que la actualización del punto ocurre en tiempo constante.

En el caso de las bitácoras codificadas con Elías γ se podría hacer lo mismo, pero hay que tener en cuenta que es necesario decodificar todas las posiciones relativas hasta t_q para obtener el desplazamiento. Lo anterior está bien para el caso de los objetos candidatos que son parte de los resultados, pero es ineficiente para el caso de los objetos que no lo son.

Para evitar la decodificación de toda la bitácora hasta t_q se puede ir decodificando una posición relativa a la vez, actualizar su posición y verificar si el objeto sale de r', en cuyo caso podemos concluir que el objeto nunca entrará a r y, por lo tanto, se puede descartar. Algo similar ocurre si la posición del objeto se encuentra dentro de r'_i, en cuyo caso se puede concluir que el objeto será parte de los resultados independiente de los movimientos que falten por procesar.

En la medida que se va explorando la bitácora de un objeto, es posible acotar el máximo desplazamiento posible (MD) al descontar de MD los movimientos que el objeto a dado. Con el nuevo MD, más ajustado, se recalculan las regiones r' y r'_i, las cuales serán de un menor tamaño, acotando los análisis posteriores.

5.3.3. Time Interval

Las consultas de time interval siguen la misma estrategia general de la versión anterior. Por un lado está el algoritmo 4.6 *Time Interval Query* que toma el intervalo de la consulta y lo divide, en el caso de intersectar más de una bitácora, en tantas subconsultas como bitácoras intersectadas hayan. Luego para cada subconsulta llama al algoritmo 4.7 *Limited Time Interval* que resuelve el problema del time interval limitado a una bitácora.

El segundo algoritmo es el que varía dada las características de la bitácora, en cambio el primero se mantiene sin cambios por lo cual no es presentado aquí nuevamente.

El algoritmo 5.5 presenta la versión del algoritmo *Limited Time Interval* para bitácoras con movimientos largos. Al igual que en caso del *time slice* cambia en tres aspectos que son: la manera de determinar el máximo desplazamiento posible, las bitácoras se decodifican siempre utilizando el snapshot anterior y se delega la responsabilidad de evaluar la bitácora de un candidato a una función con la finalidad de encapsular las particularidades asociadas a un tipo específico de bitácora.

Este algoritmo asume que el intervalo está bien formado, es decir, el limite superior del intervalo temporal es mayor que el limite inferior. En el caso que sean iguales en realidad la consulta es de tipo *time slice* y, por lo tanto, se resuelve con una llamada al algoritmo respectivo (líneas 1-3).

A continuación se obtiene el *snapshot* s_i el cual es el antecesor de la colección de bitácoras L la cual es la que intersecta con el intervalo t.

Al igual que en la consulta de time slice, aquí hay que realizar una consulta por rango en s_i para obtener un conjunto de candidatos (línea 9). Previo a ello hay que determinar r' (línea 7), que es la región de la consulta ampliada según el desplazamiento máximo que algún objeto realizó desde el snapshot s_i hasta el instante $t.end$, obtenido en la línea 6. Se utiliza el instante final del intervalo, porque un objeto móvil puede entrar a la región de la consulta hasta el último instante de intervalo de la consulta.

Luego se revisan todos los candidatos, para determinar cuales son parte de los resultados y cuales no. Para ello se revisa si el candidato se encuentra en la región r'_i que corresponde a la región espacial de aquellos objetos que aún moviéndose quedarán dentro del área de la consulta al finalizar el intervalo t y, por lo tanto, son parte de

Algorithm 5.5 Limited Time Interval

Input: *cst*: una instancia de CST-index, r: región de la consulta, $[t_s, t_e]$: intervalo temporal. *Result*: conjunto de objetos con los resultados parciales previos
Output: *Result*: conjunto de objetos en *cst* que intersectan con r en algún instante de $[t_s, t_e]$ unido con los resultados parciales iniciales.

1: **if** $t.start = t.end$ **then**
2: $Result = Result \cup$ TimeSlice$(cst, r, t.start)$
3: **end if**
4: $s_i =$ getSnapshot$(cst, t.start)$
5: $L =$ getLog$(cst, t.start)$
6: $MD =$ getMaxDesp$(L, t.end)$
7: $r' =$ RegionExterior(r, MD)
8: $r'_i =$ RegionInterior(r, MD)
9: $cand =$ RangeQuery(s_i, r')
10: **for all** $c \in cand$ **do**
11: **if** c in r'_i **then**
12: $result = result \cup \{c\}$
13: **else if** $c \notin result$ **then**
14: $l = L.getLog(c)$
15: **if** timeIntervalTest$(l, c.pos, t, r, MD)$ **then**
16: $result = result \cup \{c\}$
17: **end if**
18: **end if**
19: **end for**
20: **return** *result*

los resultados. En caso contrario, si el objeto candidato c no es parte de los resultados previos, se evalúa mediante la función timeIntervalTest$(l, c.pos, t, r, MD)$ si el objeto en algún instante del intervalo t se encuentra en la región de la consulta r. Si es así la función retornará verdadero y se incluirá c en los resultados.

La función timeIntervalTest() varía según el tipo de bitácora, pero la estrategia general es la siguiente: Primero, se actualiza la posición del objeto al inicio el intervalo ($t.start$). Con la posición actualizada se revisan dos condiciones de salida: 1) si el objeto se encuentra fuera de r' lo que significa que el objeto no podrá entrar a la región de la consulta r y, por lo tanto, no forma parte de los resultados (se retorna falso); y 2) si el objeto se encuentra en r significa que el objeto es parte de los resultados y se retorna verdadero. Si no se cumplen las condiciones 1 y 2 se actualiza el objeto al siguiente instante dentro del intervalo t donde haya una posición relativa conocida y se vuelve a repetir el proceso mientras no se de una condición de salida o bien no queden posiciones relativas dentro del intervalo de la consulta t que revisar, retornando falso en este último caso.

Con cada desplazamiento parcial del objeto se redefine MD, y se ajusta la región

r', la cual se va achicando en la medida que avanzamos en el análisis del intervalo de la consulta.

Hay dos casos especiales que se revisan al principio y es que el objeto no se haya movido durante el intervalo de la bitácora y, por lo tanto, no hay nada que actualizar y se debe concluir basado en la posición del objeto en el snapshot.

El otro caso es muy parecido, y ocurre cuando el objeto no se ha movido hasta el inicio del intervalo, eso significa que la posición del objeto en el snapshot es válida durante t y, por lo tanto, hay que revisarla antes de decodificar las posiciones relativas que están dentro del intervalo, de modo que se retornando verdadero si $pos \in r$.

Como las bitácoras codificadas con Elías siempre se deben decodificar desde el principio, es posible terminar el proceso de evaluación antes de llegar al intervalo temporal en dos casos. Si el objeto se mueve fuera de r', no podrá entrar en r durante t y, por lo tanto, se retorna falso; o bien si entra a r'_i, el objeto estará en r al menos una vez durante t y se retorna verdadero.

El algoritmo 5.6 detalla la función TIMEINTERVALTEST() para bitácoras codificadas en unario y el algoritmo 5.7 para bitácoras con Elías γ.

Algorithm 5.6 Time Interval Test para bitácora codificada en unario

Input: l: bitácora, pos: posición del objeto en el snapshot, t:intervalo de la consulta, r: región de la consulta, MD: desplazamiento máximo posible desde el snapshot hasta $t.end$

Output: verdadero si el objeto ha entrado al menos una vez a r durante t, falso si no lo hace.

1: **if** $(l = null)$ **then**
2: **return** $(pos \in r)$?true:false
3: **end if**
4: $start = rank_1(l.mov, t.start)$
5: $end = rank_1(l.mov, t.end)$
6: **if** $(end = 0)$ **then**
7: **return** $(pos \in r)$?true:false
8: **end if**
9: **if** $(start = 0) \wedge (pos \in r)$ **then**
10: **return** true
11: **end if**
12: $i = start$
13: $p_r = \text{GETSUMMOV}(l, t.start)$
14: **while** $(i \leq end)$ **do**
15: $MD = \text{RESIZEMD}(MD, p_r)$
16: $pos = pos + p_r$
17: $r' = \text{REGIONEXTERIOR}(r, MD)$
18: $r'_i = \text{REGIONINTERIOR}(r, MD)$
19: **if** $(pos \notin r')$ **then**
20: **return** false
21: **else if** $(pos \in r)$ **then**
22: **return** true
23: **end if**
24: $i = i + 1$
25: sea p_r la i-ésima posición relativa del log decodificada
26: **end while**
27: **return** false

Algorithm 5.7 Time Interval Test para bitácora codificada con Elías γ

Input: l: bitácora, pos: posición del objeto en el snapshot, t:intervalo de la consulta, r: región de la consulta, MD: desplazamiento máximo posible desde el snapshot hasta $t.end$

Output: verdadero si el objeto ha entrado al menos una vez a r durante t, falso si no lo hace.

1: **if** $(l = null)$ **then**
2: **return** $(pos \in r)$?true:false
3: **end if**
4: $start = rank_1(l.mov, t.start)$
5: $end = rank_1(l.mov, t.end)$
6: **if** $(start = 0) \wedge (pos \in r)$ **then**
7: **return** true
8: **end if**
9: $i = 1$
10: **while** $(i \leq end)$ **do**
11: sea p_r la i-ésima posición relativa del log decodificada
12: $MD = \text{RESIZEMD}(MD, p_r)$
13: $pos = pos + p_r$
14: $r' = \text{REGIONEXTERIOR}(r, MD)$
15: $r'_i = \text{REGIONINTERIOR}(r, MD)$
16: **if** $(i < start)$ **then**
17: **if** $(pos \notin r')$ **then**
18: **return** false
19: **else if** $(pos \in r'_i)$ **then**
20: **return** true
21: **end if**
22: **else**
23: **if** $(pos \notin r')$ **then**
24: **return** false
25: **else if** $(pos \in r)$ **then**
26: **return** true
27: **end if**
28: **end if**
29: $i = i + 1$
30: **end while**
31: **return** false

5.3.4. Obtener la trayectoria de un objeto

Como se comentó en el capítulo anterior, una de las características de nuestro índice es que permite la obtención de la trayectoria de un objeto en un intervalo de tiempo dado. La trayectoria resultante de esta operación son las posiciones que se han registrado para un objeto en particular junto con el instante asociado a cada posición en el intervalo de la consulta. Esta trayectoria no es interpolada de manera alguna, simplemente recupera los valores insertados originalmente en el índice dejando al usuario la decisión de interpolar o no.

Dada las características de esta versión del índice, la obtención de la trayectoria se puede realizar únicamente leyendo las bitácoras de inicio a fin y no en sentido reverso, lo que representa un cambio importante respecto a la versión presentada en el capítulo anterior (4.10).

El algoritmo 5.8 getTrajectory() presenta los pasos para obtener la trayectoria de un objeto *oid* sobre el índice *cst* en el intervalo $[t_s, t_e]$. La trayectoria resultante es una secuencia o lista de pares ⟨Punto, Instante⟩ ordenada por tiempo de menor a mayor.

El algoritmo comienza creando una trayectoria vacía a la cual se añadirán más adelante los pares ⟨Punto, Instante⟩ que se vayan encontrando.

Luego se determina la secuencia de colecciones de bitácoras L que intersectan con el intervalo de la consulta (línea 2). L tendrá una o muchas colecciones de bitácoras lo que determina dos casos dentro de este algoritmo dependiendo del tamaño del intervalo de la consulta.

En este punto hay dos casos posibles que son tratados en forma separada dependiendo de la cantidad de colecciones de bitácoras en L.

Si solo hay una colección de bitácoras que intersecte con el intervalo de la consulta, la información necesaria para obtener la trayectoria se encontrará en la bitácora (*log*) del objeto de la consulta (*oid*), la cual se obtiene desde la primera y única colección de bitácoras (l_1) de L (línea 4).

Como la bitácora *log* tiene almacenada las posiciones y sus instantes de manera relativas al *snapshot* S_1, para obtener las pociones absolutas es necesario obtener previamente la posición (*pBase*) del objeto de la consulta en el *snapshot* S_1 (línea 6); y para obtener el instante absoluto se necesita el instante (*tBase*) de S_1 (línea 7).

Una vez obtenido *pBase* se revisa si forma parte de la trayectoria lo que ocurre cuando en la bitácora *log* no existe ninguna posición relativa hasta *ts* (línea 8). En este caso hay que agregar el punto *pBase* junto con su instante t en la trayectoria resultante (línea 12). Es importante señalar que *tBase* no necesariamente igual a

t. Para obtener t (línea 11) hay que encontrar el instante del último punto en l, donde l es la bitácora del objeto que precede a *log* (obtenida en la línea 9). En el caso que el objeto no se haya movido durante l ($l = null$) se asume que que este hueco temporal de f instantes en la historia del objeto implica el término de una trayectoria y comienzo de otra y, por lo tanto, $pBase$ se descarta.

Ahora que ya tenemos lo necesario para explorar *log* el problema se resuelve con una llamada al algoritmo LIMITEDTRAJECTORY() (línea 15). Este algoritmo explorará el *log* para agregar a la trayectoria *result* todas las posiciones y sus respectivos instantes que encuentre dentro del rango temporal relativo $[t_s - tBase$, $t_s - tBase]$, utilizando $pBase$ y $tBase$ para pasar desde las posiciones e instantes relativos a los absolutos.

El segundo ocurre cuando L tiene más de una colección de bitácoras.

La estrategia para abordar este caso consiste en particionar la consulta en tantas sub-consultas como colecciones de bitácoras hayan en L. La primera sub-consulta tendrá el rango temporal relativo de consulta $[ts-tBase, f]$ y la última $[1, te-tBase]$. Las consultas intermedias (de la 2 a la $|L| - 1$), si las hubiere, tendrán el rango temporal relativo igual al de la bitácora $[1, f]$.

Cada sub-consulta se ejecutará en orden, desde la primera a la última, usando LIMITEDTRAJECTORY().

Al igual que en el caso anterior, Antes de ejecutar la primera sub-consulta se obtiene *log*, $pBase$, $tBase$ y se verifica si el punto base es válido dentro del intervalo de la consulta, en cuyo caso se agrega a la trayectoria resultante como primer punto (líneas 21-27).

La ejecución de cada sub-consulta dará como resultado la última posición absoluta que ha insertado en *result*. Esta posición será $pBase$ de la siguiente sub-consulta.

Luego de ejecutar la primera consulta, se ejecutan las intermedias (líneas 29-33). En cada iteración se ajustan adecuadamente $tBase$, *log* y $pBase$ de modo que al ejecutar la sub-consulta los datos de entrada sean los correctos.

Finalmente, la última sub-consulta es ejecutada si t_e ocurre después que $tBase$, porque en caso contrario el instante t_e ya fue procesado.

Los pasos detallados de LIMITEDTRAJECTORY() son presentados tanto para las bitácoras codificadas en unario como para las codificadas con Elías γ en Algoritmo 5.9.

Algorithm 5.8 (**getTrajectory**)obtención de la trayectoria de un objeto sobre el índice

Input: *cst*: un índice, *oid*: identificador del objeto, $[t_s, t_e]$:intervalo temporal de la trayectoria.

Output: Secuencia de pares ⟨Punto, Instante⟩.

1: $result$ = *una trayectoria vacía*
2: L =*subsecuencia de colecciones de bitácoras en cst que ocurren en el intervalo* $[t_s, t_e]$
3: **if** $|L| = 1$ **then**
4: $log = L_1.getLog(oid)$
5: $S_1 = \text{GETSNAPSHOT}(cst, t_s)$
6: $pBase = \text{GETOBJECTPOS}(S_1, oid)$
7: $tBase = \text{TIMESTAMP}(S_1)$
8: **if** $\text{RANK1}(log.mov, t_s - tBase) == 0$ **then**
9: $l = \text{GETLOG}(cst, ts - 1).getLog(oid)$
10: **if** $l.mov \neq null$ **then**
11: $t = tBase - f$ + *posición del último 1 en l.mov*
12: $\text{ADD}(result, \langle pBase, t \rangle)$
13: **end if**
14: **end if**
15: $\text{LIMITEDTRAJECTORY}(log, pBase, tBase, t_s - tBase, t_e - tBase, result)$
16: **else**
17: $log = L_1.getLog(oid)$
18: $S_1 = \text{GETSNAPSHOT}(cst, t_s)$
19: $pBase = \text{GETOBJECTPOS}(S_1, oid)$
20: $tBase = \text{TIMESTAMP}(S_1)$
21: **if** $\text{RANK1}(log.mov, t_s - tBase) == 0$ **then**
22: $l = \text{GETLOG}(cst, ts - 1).getLog(oid)$
23: **if** $l.mov \neq null$ **then**
24: $t = tBase - f$ + *posición del último 1 en l.mov*
25: $\text{ADD}(result, \langle pBase, t \rangle)$
26: **end if**
27: **end if**
28: $pBase = \text{LIMITEDTRAJECTORY}(log, pBase, tBase, t_s - tBase, f, result)$
29: **for** $i = 2$ **To** $|L| - 1$ **do**
30: $tBase = tBase + f$
31: $log = L_i.getLog(oid)$
32: $pBase = \text{LIMITEDTRAJECTORY}(log, pBase, tBase, 1, f, result)$
33: **end for**
34: $tBase = tBase + f$
35: **if** $t_e > tBase$ **then**
36: $log = L_{|L|}.getLog(oid)$
37: $\text{LIMITEDTRAJECTORY}(log, pBase, tBase, 1, t_e - tBase, result)$
38: **end if**
39: **end if**
40: **return** $result$

Algorithm 5.9 (**limitedTrajectory**)

Input: l: bitácora, $pBase$: posición del objeto en el snapshot, $tBase$:instante del snapshot, t_s: instante relativo inicial, t_e: instante relativo final, $result$ trayectoria a la que se agregarán nuevos pares ⟨Punto, Instante⟩

Output: Punto que será base para las siguientes consultas.

Versión en unario

```
1: if (l = null) then
2:     return pBase
3: end if
4: start = RANK1(l.mov, t_s)
5: end = RANK1(l.mov, t_e)
6: if (end = 0) then
7:     return pBase
8: end if
9: if (start = 0) then
10:    start = 1
11: end if
12: t = SELECT1(l.mov, start)
13: i = start
14: while (i ⩽ end) do
15:    p_r = GETSUMMOV(l, t)
16:    pos = pBase + p_r
17:    ADD(result, ⟨pos, t⟩)
18:    i = i + 1; t = t + 1
19:    t = SELECTNEXT1(l.mov, t)
20: end while
21: return pos
```

Versión Elías γ

```
1: if (l = null) then
2:     return pBase
3: end if
4: start = RANK1(l.mov, t_s)
5: end = RANK1(l.mov, t_e)
6: if (end = 0) then
7:     return pBase
8: end if
9: if (start = 0) then
10:    start = 1
11: end if
12: t = SELECT1(l.mov, start)
13: i = 1
14: while (i ⩽ end) do
15:    sea p_r la i-esima posición relativa de
       l decodificada
16:    pos = pos + p_r
17:    if (i ⩾ start) then
18:        ADD(result, ⟨pos, t⟩)
19:        t = t + 1
20:        t = SELECTNEXT1(l.mov, t)
21:    end if
22:    i = i + 1
23: end while
24: return pos
```

Las primeras 11 líneas de ambas versiones de LIMITEDTRAJECTORY() son iguales, y se encargan de revisar tres casos especiales, además de obtener *start* y *end* que corresponden a la ubicación, en la secuencia codificada, de la primera y la última posición relativa que se encuentra dentro del intervalo de la consulta.

El primero de los casos especiales ocurre cuando la bitácora l no contiene movimiento alguno, terminando sin modificar *result* y devolviendo $pBase$ para la siguiente iteración.

El segundo caso se resuelve igual que el anterior, y ocurre cuando el objeto no se ha movido hasta el instante t_e y, por lo tanto, no hay movimientos que recuperar.

El tercer caso especial ocurre cuando el objeto no se ha movido hasta el inicio del intervalo t_s ($start = 0$), lo que requiere ajustar *start* a 1.

Sabiendo que existen posiciones relativas que procesar se recorre la bitácora para recuperar las posiciones relativas desde *start* hasta *end* y se llevan a posiciones absolutas usando *pBase* y *tBase*.

En el caso de las bitácoras codificadas en unario, es posible comenzar la decodificación desde *start*. Pero en el caso de las bitácoras con *Elías*, hay que comenzar siempre de la primera posición en la secuencia comprimida.

En ambos casos en cada iteración se obtiene una posición relativa y se lleva a una absoluta (líneas 15 y 16). Luego en el caso de las bitácoras en unario esta posición, junto con su instante t es insertado en *result*. En cambio en el caso de las bitácoras con *Elías*, solo se insertará el punto si se ha alcanzado la posición relativa inicial (línea 17).

Finalmente ambas versiones devuelven *pos* que es el último punto insertado en la trayectoria resultante *result*.

5.4. Los k-vecinos más cercanos

El problema de los k-vecinos más cercanos (kNN) en una base de datos espacio-temporal U consiste en encontrar los k objetos más cercanos a un punto dado p en el instante t de la consulta. Formalmente lo que se desea es encontrar un conjunto de objetos $A \subset U$ tal que $|A| = k$ y para todo objeto $a \in A$ y para todo objeto $b \in (U - A)$ se satisface la inecuación $dist(p, a_t) \leqslant dist(p, b_t)$ donde la función $dist(\cdots)$ es la función de distancia euclidiana entre dos puntos y a_t denota la posición del objeto a en el instante t, del mismo modo b_t es la posición de b en t.

Para abordar el problema de los k-vecinos más cercanos en nuestra estructura se han identificado dos casos. El primero caso ocurre cuando el instante de la consulta coincide con el instante en el cual se ha generado un snapshot. El segundo caso ocurre cuando el instante de la consulta está entre dos snapshot. En el primer caso la búsqueda se resuelve únicamente con la información contenida en el snapshot. En el segundo caso se deben realizar tres pasos, el primero de filtrado sobre el snapshot más cercano al instante de la consulta, el segundo de actualización de los candidatos al tiempo de la consulta y el tercero de selección de los k mejores candidatos.

La solución que a continuación se presenta consta de tres algoritmos:

1. **kNN en un índice CST.** Este algoritmo es el principal, recibe como entrada

el índice CST sobre el cual se consulta, el punto p de la consulta, la cantidad k de vecinos a recuperar y el instante t de la consulta. Este algoritmo identificará los casos y los resolverá apoyándose en los dos siguientes.

2. **kNN sobre un snapshot con resultado exacto.** Este algoritmo encuentra los k vecinos más cercanos a un punto p en un snapshot dado. Se utiliza para resolver el problema para el primer caso.

3. **kNN sobre un snapshot con resultado aproximado.** Este algoritmo encuentra k' vecinos más cercanos a p considerando que existe un distancia máxima δ en la que puede diferir la posición del objeto contenida en el snapshot respecto de la posición al momento de la consulta (t), y, por lo tanto, $|U| \geqslant k' \geqslant k$. Este algoritmo se utiliza para filtrar k' candidatos como primer paso del segundo caso.

A continuación se explicará cada uno detalladamente.

5.4.1. Consultas de kNN en un índice CST

Para resolver la consulta de los k vecinos más cercanos es necesario identificar si el instante t de la consulta ocurre en el mismo instante en el cual hay un snapshot o bien entre dos snapshot. Para ello, primero se necesita encontrar el snapshot S, el cual posee el instante mayor de los menores o iguales que t (línea 1) y el instante t_s en el que ocurre tal snapshot (línea 2). Si $t_s = t$ entonces estamos en el primero de los casos y la consulta se resuelve mediante el algoritmo 5.11.

En caso contrario, la consulta ocurre entre dos snapshot y, por lo tanto, la información contenida en el snapshot S indicará la ubicación aproximada de un objeto en t con un error máximo δ, que corresponde a la mayor distancia que un objeto se ha podido mover durante el intervalo $(t_s \dots t]$.

Para calcular δ primero se recupera la colección de bitácoras L que intersecta con t y de ella se obtiene el vector MD con los máximos desplazamientos desde el instante por cada uno de los sentidos del movimiento desde el instante t_s a t. Con MD se obtiene la distancia entre $(0, 0)$ y $(MD_1 - MD_0, MD_3 - MD_2)$ que es δ.

Una vez obtenido S y δ se realiza la búsqueda de los k vecinos más cercanos a p aproximada sobre el snapshot S considerando el error δ usando el algoritmo 5.12. El resultado de esta consulta será una lista de k' candidatos c, donde $k' \geqslant k$.

A continuación se procesa c para actualizar la posición de cada uno de los objetos candidatos al instante t obteniendo una lista de candidatos actualizada c' (líneas 10-15). Para cada objeto candidato o se obtiene su desplazamiento desde t_s a t, información contenida en su respectiva bitácora. Una vez encontrado este

desplazamiento se actualiza la posición del objeto, se corrige la distancia al punto de la consulta q y se agrega el objeto en la cola de salida utilizando como prioridad la distancia corregida.

Finalmente se obtienen los k mejores candidatos de la lista c'

El algoritmo 5.10 corresponde al proceso antes descrito.

Algorithm 5.10 (**kNNQuery**)búsqueda de los k vecinos más cercanos sobre un índice CST

input: el índice *cst* sobre el cual se ejecuta la consulta, el punto q de la consulta, la cantidad k de vecinos buscados, el instante t de la consulta.

output: una lista los k vecinos más cercanos ordenada del más cercano al más lejano.

```
1: S = getSnapshot(cst, t)
2: t_s = TimeStamp(S)
3: if (t_s = t) then return knn(s, p, k)
4: end if
5: L = getLog(cst, t)
6: MD = getMaxDesp(L, t)
7: δ = mdToDist(MD)
8: c = knnAprox(s, p, k, δ)
9: c' = createMin-heap()
10: for all objeto o ∈ c do
11:     d = getSumMov(L.getLog(o.oid), t)
12:     o.x = o.x + d.x
13:     o.y = o.y + d.y
14:     o.prioridad = dist(o, q)
15:     c'.push(o)
16: end for
17: return los k primeros objetos de c'
```

5.4.2. kNN sobre un snapshot con resultado exacto

El algoritmo 5.11 presenta la estrategia utilizada para resolver el problema de los kNN sobre un snapshot con un resultado exacto. Este algoritmo toma como entrada el snapshot que contiene los objetos, el punto de referencia para la consulta y la cantidad k de vecinos más cercanos a encontrar.

La estrategia general consiste en recorrer el k^2-tree visitando primero a los subárboles que están a una menor distancia al punto de la consulta. Para ello se utiliza la cola de prioridad *pQueue* que es un min-Heap de subárboles usando como prioridad

Algorithm 5.11 (**kNN**)sobre un snapshot con resultado exacto

input: un snapshot *snap*, el punto q de la consulta, la cantidad k de vecinos buscados.
output: una lista ordenada por distancia de identificadores de objetos y su ubicación en el espacio.

```
1: c = createMax-heap()
2: pQueue = createMin-heap()
3: e = createElementQueue(0, snap.AreaDelEspacio(), maxDist(q, e.area))
4: pQueue.push(e)
5: while (pQueue ≠ ∅) do
6:     e = pQueue.top()
7:     pQueue.pop()
8:     if (e es hoja de último nivel) then
9:         n =cantidad de objetos en e
10:        sea O los m = min(k, n) primeros objetos de la celda e
11:        for all objeto o en O do
12:            entry = createEntry(o, e.area.x1, e.area.y1, e.prioridad)
13:            if (|c| < k) then
14:                c.push(entry)
15:            else if (e.prioridad < c.top().priority()) then
16:                c.pop()
17:                c.push(entry)
18:            end if
19:        end for
20:    else
21:        if (e tiene hijos) then
22:            for all hijo h de e do
23:                obtener la subArea de h
24:                menorDis = minDist(q, subArea)
25:                if ((|c| < k) ∨ (menorDis < c.top.priority())) then
26:                    mayorDis = maxDis(q, subArea)
27:                    entry = createElementQueue(child, subArea, mayorDis)
28:                    pQueue.push(entry)
29:                end if
30:            end for
31:        end if
32:    end if
33: end while
34: return c
```

la mayor distancia del área de cobertura del sub-árbol al punto q de modo que la menor de las mayores distancias queda primero en el Heap.

Junto con *pQueue* se utiliza otra cola de prioridad para los candidatos llamada c. Esta cola es un max-Heap que usa como prioridad la distancia del objeto candidato

al punto de la consulta q de modo que el peor candidato queda al tope. Por cada candidato se cuenta con su identificador de objeto *oid*, la posición (x, y) donde está el objeto y su prioridad.

Inicialmente *pQueue* contiene a la raíz como único sub-árbol a visitar (línea 4). Cada vez que se extrae un sub-árbol de la cola puede ocurrir que este sea una hoja de último nivel, un nodo intermedio que tenga hijos (nodo en 1) y un nodo intermedio sin hijos (nodo en 0), los cuales se ignoran.

En el caso que el sub-árbol sea una hoja de último nivel, ésta representa a una celda que contiene n objetos. De los n objetos se necesitarán como máximo k, como n puede ser menor a k el total de objetos que se necesita recupera de la celda es $m = \text{mín}(k, n)$. De los m objetos alguno de ellos puede entrar en los candidatos. Un objeto o entra en los candidatos si aún no se han encontrado los k candidatos iniciales (líneas 13-14). Si ya se cuenta con k candidatos en la cola c, la única posibilidad de que un objeto o entre en la cola es que sea mejor candidato que el peor de los candidatos en la cola, es decir, que la distancia del objeto o a q sea menor que la distancia que tiene el candidato del tope de la cola c (línea 15). En este caso se elimina el elemento del tope y se agrega o a la cola de candidatos c (líneas 16 y 17).

En el caso de que el nodo extraído de la cola *pQueue* sea un nodo intermedio con hijos (linea 21), cada sub-árbol del nodo se pondrá en la cola *pQueue* si aún no se han encontrado k objetos candidatos; o bien, si existe la posibilidad de encontrar un mejor candidato en éste sub-árbol que el peor de los candidatos actuales (línea 25). Para verificar esta última condición se calcula la menor de las distancias del punto de la consulta al área del sub-árbol en cuestión (recordar que cada sub-árbol de un K^2-tree representa una sub-área del espacio de su padre) dicha distancia será el mejor caso para cualquiera de los objetos contenidos en el sub-árbol visitado. En el caso que esta distancia sea mayor o igual que la distancia del candidato más lejano al punto de la consulta se puede descartar el sub-árbol dado que ningún objeto contenido en él podrá ser mejor que el candidato actual, en caso contrario existiría la posibilidad de encontrar un mejor candidato en el sub-árbol y, por lo tanto, es necesario incluir el sub-árbol en la cola *pQueue* para su posterior revisión.

El algoritmo termina cuando se han visitado todos los sub-árboles de la cola *pQueue* entregando como resultado los k candidatos contenidos en la cola c.

5.4.3. kNN sobre un snapshot con resultado aproximado

Esta consulta es muy similar a la consulta de kNN con resultado exacto manteniendo la misma estrategia de recorrido del K^2-Tree usando una cola de prioridad *pQueue* pero añadiendo un parámetro más que es el error δ el cual es usado en las condiciones que permiten decidir si se incluye o no un sub-árbol y en las condiciones para determinar si se incluye o no un nuevo candidato. El resultado final puede ser mayor

a k dado que pueden existir candidatos para los cuales no podamos discriminar cual es el mejor dado el error existente. En el algoritmo 5.12 se presentan en detalle los pasos a seguir, pero en esta explicación estará centrada en las diferencias con respecto a la consulta kNN con resultado exacto.

Algorithm 5.12 Knn sobre un snapshot con resultado aproximado

input: un snapshot *snap*, el punto q de la consulta, la cantidad kn de vecinos buscados, la distancia máxima δ que un objeto se puede haber movido desde la posición en el snapshot hasta el momento de la consulta.

output: una lista ordenada por distancia de identificadores de objetos y su ubicación en el espacio.

```
1: c = createMax-heap()
2: pQueue = createMin-heap()
3: e = createElementQueue(0, snap.AreaDelEspacio(), maxDist(q, e.area))
4: pQueue.push(e)
5: while (pQueue ≠ ∅) do
6:     e = pQueue.top()
7:     pQueue.pop()
8:     if (e es hoja de último nivel) then
9:         if (|c| < k) then
10:            poner todos los objeto de la celda e en la cola de candidatos
11:        else if ((e.prioridad + δ) = (c.top().priority() − δ)) then
12:            poner todos los objeto de la celda e en la cola de candidatos
13:        else if ((e.prioridad + δ) < (c.top().priority() − δ)) then
14:            poner todos los objeto de la celda e en la cola de candidatos
15:            prioK = prioridad del k-ésimo mejor candidato en c
16:            eliminar de c todos los objetos con prioridad mayor a prioK.
17:        end if
18:    else
19:        if (e tiene hijos) then
20:            for all hijo h de e do
21:                obtener la subArea de h
22:                mayorDis = maxDis(q, subArea)
23:                menorDis = minDist(q, subArea)
24:                if ((|c| < k) ∨ (menorDis − δ < c.top().priority() + δ)) then
25:                    entry = createElementQueue(child, subArea, mayorDis)
26:                    pQueue.push(entry)
27:                end if
28:            end for
29:        end if
30:    end if
31: end while
32: return c
```

La primera diferencia la encontramos cuando llegamos a la hoja, como la ubicación del objeto es aproximada no podemos simplemente tomar k objetos, puesto que los escogidos podrían ser los peores al momento de actualizarlo. De este modo el criterio para incluir o descartar candidatos no se puede aplicar a uno en particular sino a toda las celda.

La segunda diferencia está en el caso de un nodo hoja y se ha completado la lista de candidatos (lineas 16-18). Aquí la diferencia corresponde a la estrategia con la cual se incorporan a la cola nuevos candidatos y se descartan los más lejanos. Al visitar una celda no se incluirá ninguno de los objetos si la distancia más cercana posible al punto de la consulta que es $e.priority() - \delta$, (si esta diferencia es <0 entonces la distancia mínima es 0) es mayor que la distancia más lejana del candidato del tope $(c.top().priority + \delta)$ de la cola c debido a que no existe la posibilidad de que exista un mejor candidato que los actuales que hay en la cola. Si la mejor de la distancia es igual a la peor de las distancias del candidato existiría la posibilidad que los objetos de la celda sean mejores candidatos que el del tope por lo que se incorporan todos. En el caso que la mayor distancia del objeto de la celda tenga una distancia menor que la menor distancias del objeto del tope de la cola, se insertan todos los objetos de la celda y se poda la cola de candidatos. La poda consiste en eliminar todos los objetos cuya prioridad es mayor que la prioridad del k-ésimo mejor candidato.

Por último la estrategia para incorporar un nuevos sub-arboles a la cola *pQueue* es similar a la de incorporar un candidato individual. Se descartará un sub-árbol si la menor de las distancias del sub-árbol posible es mayor que la mayor de las distancias posibles del candidato del tope de c.

5.5. Evaluación experimental

En esta sección, se presenta un estudio experimental comparativo de nuestra estructura tanto en espacio como en tiempo de ejecución.

Para la experimentación se ha utilizado el mismo conjunto de datos reales utilizado en la versión anterior. Estos datos corresponden a 58.691.821 ubicaciones de 4.824 barcos recolectadas en un período de un mes con muestras a cada segundo. Esta colección fue preprocesada para limpiarla de aquellos movimientos que, por la naturaleza de los barcos, no podían ser posibles, corrigiendo estos errores mediante interpolación.

En la tabla 5.3 se presenta un resumen estadístico de la colección, donde la columna Δ_t corresponde a la variación del tiempo en segundos entre dos muestras consecutivas de una misma trayectoria. La columna distancia corresponde a la distancia recorrida en metros entre dos muestras consecutivas de una misma trayectoria y la columna velocidad es la velocidad en metros por segundo entre dos

Tabla 5.3: Estadísticas de los movimientos de los barcos en la colección original. Como se puede apreciar, los movimientos máximos no son razonables para un barco. Estos datos fueron limpiados para los experimentos.

Estadísticos	Δ_t	**distancia (m)**	**velocidad (m/s)**
mínimo	0,00	0,00	0,00
1° cuartil	3,00	0,87	0,05
mediana	8,00	37,67	4,96
3° cuartil	11,00	62,74	9,27
media	84,18	145,60	19,67
Máximo	2.593.000	990.200	489.900

muestras consecutivas de una misma trayectoria.

Como en esta versión del índice es posible contar con movimientos a celdas no adyacentes y además es posible tener más de un punto por celda, se ha realizado una división del espacio en celdas fijas de un tamaño de 30x30 metros aproximadamente, transformando las posiciones desde el formato latitud longitud a las celdas correspondientes en el nuevo espacio, tomando muestras de los puntos a cada minuto.

Esta configuración disminuye el tamaño de la colección, pasando de 58.691.821 ubicaciones a 5.338.598, dado que al usar una celda de $30m^2$ todos los movimientos continuos de un objeto que ocurren en una misma celda son ignorados, salvo el primero de ellos.

La evaluación considera también diferentes largos de bitácoras (parámetro f) que es equivalente al número de instantes entre snapshots consecutivos.

Los experimentos fueron ejecutados en un servidor con 8 procesadores Intel(r) Core(tm) i7-3820 @ 3.60 GHZ con memoria cache L1 de 32 KB, L2 de 256 KB, L3 de 10MB y 32 GB de RAM.

Análisis del espacio utilizado por el índice Se analizó la sensibilidad de la estructura para diferentes largos de bitácora, teniendo en cuenta tres maneras de codificar las bitácoras: Codificación en unario usando *bitmaps* sin comprimir, codificación en unario usando *bitmaps* comprimidos utilizando [RRR07] y codificación con códigos Elías γ.

Tabla 5.4: Comparación del tamaño del índice en MB considerando diferentes largos de bitácoras para las tres versiones de bitácoras.

f	**S**	**Unario**		**Comprimido**		**Elías γ**	
		B	**Total**	**B**	**Total**	**B**	**Total**
15	125	750	876	930	1.056	**609**	735
256	7	56	64	82	89	**42**	49
512	4	35	39	51	55	**25**	29
1024	2	25	27	35	37	**17**	19
2048	1	20	22	27	28	**14**	15

En la tabla 5.4 se muestran los diferentes tamaños expresados en MB para las tres versiones de bitácoras del índices: en unario (columna **Unario**), en unario comprimido (columna **Comprimido**) y en **Elías** γ. La columna f indica el largo de las bitácoras, la columna **S** corresponde al tamaño en MB ocupado por los *snapshots* el cual es el mismo para las distintas versiones de bitácoras. La subcolumna **B** corresponde al tamaño de las bitácoras en MB y la subcolumna **Total** es el tamaño total de la versión del índice, que corresponde a la suma de las columnas **S** y **B**.

Como se puede observar en la tabla 5.4 la versión con *bitmaps* comprimidos es más costosa que la versión con los *bitmaps* sin comprimir, en todos los largos de bitácoras estudiados. Esto se debe a que las secuencias comprimidas son útiles en secuencias largas dado que el coste fijo de las estructuras adicionales domina el espacio en las secuencias cortas. Aquí es importante señalar que se utilizaron los *bitmaps* comprimidos a nivel de bitácoras, pero no se ha estudiado la posibilidad de comprimir toda la colección de bitácoras considerándolas como una única secuencia. Esta última opción permitiría hacer un uso efectivo de la compresión de *bitmaps* al compartir el coste fijo de la representación comprimida entre las bitácoras. Por otro lado, en todos los casos la versión con Elías γ produce un índice de menor tamaño, esto se debe a que la distribución de probabilidad de la colección se ajusta mejor a los códigos Elías γ que a los códigos en unario.

Si se considera que por cada ubicación de un objeto se necesita almacenar 4 enteros de 32 bits, y que en total hay 5.338.598 ubicaciones, los datos sin indexar ocupan 85,4 MB. Con el fin de tener una cota más ajustada respecto del tamaño mínimo para estos datos, se ha calculado la entropía empírica para el modelo de

Tabla 5.5: Comparación del tamaño del índice en respecto de $nH_{-1} = 42,68$ MB y $nH_0 = 36,20$ MB considerando diferentes largos de bitácoras para las versiones del índice con bitácoras en unario, unario comprimido y bitácoras con Elías γ.

f	**Unario**		**Comprimido**		**Elías γ**	
	%nH$_{-1}$	**%nH$_0$**	**%nH$_{-1}$**	**%nH$_0$**	**%nH$_{-1}$**	**%nH$_0$**
15	2.052,48 %	2.419,89 %	2.474,23 %	2.917,13 %	1.722,12 %	2.030,39 %
256	149,95 %	176,80 %	208,53 %	245,86 %	114,81 %	135,36 %
512	**91,38 %**	107,73 %	128,87 %	151,93 %	**67,95 %**	**80,11 %**
1024	63,26 %	**74,59 %**	**86,69 %**	102,21 %	44,52 %	52,49 %
2048	51,55 %	60,77 %	65,60 %	**77,35 %**	35,15 %	41,44 %

orden base $H_{-1} = 63,97$ [2] y para el modelo de orden cero $H_0 = 54,25$ bits [3]. Por lo tanto como mínimo se requieren $nH_{-1} = 42,68$ MB según modelo de orden base y $nH_0 = 36,20$ MB según modelo de orden cero, para representar la colección.

Como se puede observar en la tabla 5.5, cuando $f \leq 256$ el tamaño total del índice supera a nH_{-1} y a nH_0 y, por lo tanto, desde la perspectiva del almacenamiento no es conveniente utilizar dichos valores para este conjunto de datos. Sin embargo, cuando $f \geq 512$ el tamaño del índice es menor que nH_{-1} tanto cuando las bitácoras están codificadas en unario como con Elías γ. Para el caso de las bitácoras codificadas con Elías γ, cuando $f \geq 512$ el tamaño del índice es incluso menor que nH_0 casi en un 20 %. Esto significa que, ajustando adecuadamente el valor de f, es posible utilizar un espacio asintóticamente óptimo al mínimo dado por la teoría de la información.

Análisis de los tiempos de consulta Para comparar el tiempo de ejecución de las consultas de *time slices* y de *time interval* se generaron en forma aleatoria 50 consultas por grupo, tomando como base los mismos datos para garantizar que al menos exista un resultado por cada consulta. Cada grupo está determinado por el tamaño del área y el tamaño del intervalo temporal (0 en el caso de time slice). El resultado experimental es el tiempo de CPU promedio de ejecutar estas 50 consultas expresado en μ segundos.

Para la comparación de los tiempos se han excluido del análisis, los *bitmaps* comprimidos, pues son más lentos que los *bitmaps* planos y como se comentó en la

[2] Esto se calculó como la suma de H_{-1} obtenido para cada columna de la colección, es decir, $H_{-1}(Tiempo) = \log_2(44{,}640) = 15,44 + H_{-1}(Oid) = \log_2(4856) = 12,24 + H_{-1}(X) = \log_2(333{,}200) = 18,34 + H_{-1}(Y) = \log_2(250{,}600) = 17,93$.

[3] Esto se calculo con ayuda del paquete estadístico R y la biblioteca *entropy*, como la suma de la entropía empírica para cada columna: $H_0(Tiempo) = 15{,}05$, $H_0(Oid) = 11,03$, $H_{-1}(X) = 14,57$, $H_0(Y) = 14,08$. En el apéndice A se explica como usar R para calcular la entropía empírica.

sección anterior, no comprimen para los largos de bitácoras estudiados.

Tabla 5.6: Comparación del tiempo de CPU promedio en μ segundos para consultas de *time slice* y *time interval* para distintos largos de bitácoras en nuestra propuesta.

Tipo Consulta		**Bitácora en unario**				**Bitácora con Elías**			
Duración	**Área**	**256**	**512**	**1024**	**2048**	**256**	**512**	**1024**	**2048**
0	$\mathbf{10^2}$	470	1.162	1.611	2.701	500	1.345	2.083	4.947
	$\mathbf{100^2}$	654	1.185	1.638	1.910	693	1.370	2.331	3.146
	$\mathbf{1000^2}$	534	772	891	2.057	593	942	1.271	3.880
10	$\mathbf{10^2}$	818	1.376	1.865	3.038	841	1.447	2.302	5.098
	$\mathbf{100^2}$	689	930	1.332	1.501	691	1.014	1.693	2.217
	$\mathbf{1000^2}$	674	1.104	1.453	2.354	694	1.266	1.876	3.651
20	**10**	1.080	1.634	2.112	3.548	1.042	1.677	2.304	5.172
	$\mathbf{100^2}$	777	1.274	1.860	2.554	**758**	1.323	2.233	3.929
	$\mathbf{1000^2}$	905	1.162	1.521	2.298	911	1.216	1.794	3.090
30	$\mathbf{10^2}$	1.095	1.634	1.938	3.368	**1.054**	1.603	2.044	4.609
	$\mathbf{100^2}$	768	1.163	1.829	2.432	**741**	1.171	2.187	3.748
	$\mathbf{1000^2}$	875	1.230	1.686	2.212	**872**	1.269	2.039	2.844

Como se puede observar en la tabla 5.6 , para ambos índices, en la medida que aumenta el número de instantes entre snapshot, los tiempos también aumentan. Esto se explica por dos razones, la primera está relacionada con las consultas por rango en los snapshot, las cuales se deben realizar de manera ampliada. Esta ampliación de rango aumenta la cantidad de candidatos a evaluar y mientras mayor sea la distancia entre dos snapshot, mayor es la ampliación que se debe hacer de la consulta. La segunda razón tiene que ver con la manera en que se decodifica una posición relativa en una bitácora. Mientras las bitácoras en unario decodifican una posición en tiempo constante, las bitácoras con Elías γ requieren decodificar todas las posiciones relativas anteriores.

Lo anterior explica el porqué, mientras más largo sea el tiempo entre dos snapshot, mayor es la diferencia entre las bitácoras en unario y las bitácoras con Elías γ.

Como se puede observar, las bitácoras en unario son más rápidas en todos los casos cuando la distancia entre dos snapshot es de 512 o más instantes. Por ejemplo, si se observan los tiempos para la columna 2048, la diferencia es casi el doble.

Por otro lado, mientras más pequeña sea la separación de dos snapshot, también disminuyen estas diferencias, incluso hay casos (resaltados con negrita) en que las

bitácoras con Elías γ son mejores.

Estas diferencias se producen debido a que en bitácoras pequeñas, existen pocos puntos que decodificar con Elías γ, incluso puede que sólo sea 1 y, por lo tanto, la evaluación de una bitácora es más rápida que en unario, la que requiere decodificar por separado los movimientos de arriba y abajo, izquierda y derecha y luego obtener la posición relativa. En el caso de valores altos de f, la versión codificada en unario tiene un mejor desempeño, porque el coste de decodificar hasta una cierta posición del log es constante, en cambio en el caso de la versión codificada con Elías γ depende del número de posiciones relativas que se hayan codificado y si f es grande, pueden existir muchas posiciones relativas que sean necesarias decodificar.

Como se puede observar en la figura 5.2 existe un trade-off espacio tiempo importante en nuestra estructura, que depende del largo de la bitácora. A mayor separación entre snapshot, la estructura tiene un menor tamaño, pero aumentan los tiempos. En cambio, a una menor distancia entre snapshot, aumenta el tamaño de la estructura y disminuyen los tiempos. Esto ocurre con independencia de la manera de codificar la bitácora y del tipo de consulta realizada.

5.5.1. Comparación con MVR-Tree

En esta sección se presenta un estudio comparativo con el índice MVR-Tree y la bitácora codificada con Elías. La colección de datos y la configuración experimental es la misma descrita anteriormente.

Análisis comparativo del espacio del índice En la tabla 5.7 se presenta el tamaño y el tiempo de carga tanto para MVR-Tree como para nuestra propuesta codificada con Elías γ, considerando distintos valores de capacidad en los nodos del MVR-Tree y distintos valores para la frecuencia entre snapshot.

Respecto de los tamaños para MVR-Tree se ha realizado un ajuste, dado que la estructura original indexa rectángulos, pero aquí se desean indexar puntos. Esto permite que a nivel de las hojas, no sea necesario guardar el MBR de los datos, sino que basta con la posición del objeto, descontando este espacio adicional que en realidad no es necesario almacenar.

Al analizar el espacio que ocupa el MVR-Tree se observa que es mayor en un 246.02 % respecto de nH_{-1}, mientras que nuestra estructura, como se comentó anteriormente, es menor. Por ejemplo, si se compara nuestra propuesta con el MVR-Tree con capacidad de 60 (mejor caso), en el peor de los casos, nuestro índice necesita un 20 % del espacio ocupado por el MVR-Tree y, en el mejor, tan sólo un 6,27 %. Esta es una diferencia importante si lo que buscamos es contar con un índice en memoria principal.

Tabla 5.7: Comparación entre MVR-Tree y nuestra propuesta. Tamaño en MB. y tiempo de carga en segundos

MVR-Tree				Índice con Elías γ			
Capacidad	**Tamaño**	**T. de Carga**	**%nH$_{-1}$**	**Frecuencia**	**Tamaño**	**T. de Carga**	**%nH$_{-1}$**
15	259	60,6164	606,85 %	256	49	9,5	114,81 %
90	246	158,807	576,38 %	512	29	7,9	67,95 %
30	241	77,2326	564,66 %	1024	19	7,3	44,52 %
60	239	105,856	246.02 %	2048	15	7,7	35,15 %

Tiempo de consulta A diferencia de lo ocurrido cuando se experimentó con los datos modelados como movimientos adyacentes, aquí, el MVR-Tree, supera en todos los casos a nuestra propuesta. Esto se puede observar en la tabla 5.8.

Esta diferencia de comportamiento se debe, en parte, a que la resolución temporal se ha cambiado a minutos y la movilidad ha bajado bastante.

Los otros factores ya se han explicado, y tienen relación, con el funcionamiento del índice con una separación significativa entre snapshot.

Tabla 5.8: Comparación del tiempo de CPU promedio en μ segundos para consultas de *time slice* y *time interval* para distintos largos de bitácoras en nuestra propuesta.

Tipo consulta		MVR-Tree				Bitácora con Elías			
Duración	**Área**	**15**	**30**	**60**	**90**	**256**	**512**	**1024**	**2048**
0	10	9,4	45,4	31,46	79,9	921,0	1.526,3	2.587,8	5.446,9
	100	37,1	25,6	34,62	36,9	663,1	1.056,5	1.707,1	2.184,9
	1.000	131,8	137,	112,8	126,36	651,7	1.004,9	1.445,6	3.805,7
10	10	39,4	24,8	40,0	52,2	1.112,9	1.866,8	2.433,9	4.950,6
	100	46,7	79,8	45,8	62,3	605,6	948,7	1.563,5	2.541,9
	1.000	179,3	179,8	152,4	178,8	833,6	1.477,5	2.255,5	3.468,0
100	10	97,3	126,8	99,9	167,9	1.255,0	1.794,8	2.512,3	5.357,3
	100	100,0	141,4	131,8	203,0	913,5	1.213,6	2.046,4	3.169,9
	1000	311,2	332,3	292,6	367,2	930,3	1.360,2	1.813,0	3.530,1
1000	10	586,2	764,6	757,8	1.114,3	4.786,1	4.760,9	5.022,6	8.301,4
	100	852,9	987,08	1.042,3	1.479,1	3.086,7	3.198,8	4.156,1	4.839,3
	1000	2.342,3	2.127,5	2.118,7	2.709,7	6.357,0	5.923,6	6.078,2	6.656,8

5.6. Experimentos con datos sintéticos

En esta sección se presenta un estudio de nuestra estructura utilizando datos sintéticos. La idea es poder probar la estructura en diversos escenarios definidos por la cantidad de objetos totales, el porcentaje de objetos que se están moviendo en un instante (movilidad), el tamaño del *gird* ($N \times N$) y tamaño de la celda más pequeña (en m^2).

Para la generación de las colecciones se ha creado un programa en java que permite generar colecciones sintéticas de puntos que se mueven sobre un *grid* indicando la cantidad de objetos, el tamaño de la grid, la movilidad, la cantidad de instantes, la distribución de la velocidad del objeto (en celdas por instantes), duración mínima y máxima de un viaje y un ángulo de giro máximo. Este generador está disponible en `https://gitlab.com/miguelubb/MOGen`.

Tabla 5.9: Descripción estadística de la velocidad de distintos tipos de objetos móviles, en metros por segundo [DWF09].

Categoría	Mínimo	Máximo	Promedio	Mediana	σ	Asimetría
Motocicletas	0	35.13	31.12	32.8	4.94	-3.11
Automóvil	0	33.49	33.03	31.04	3.13	-3.04
Bicicletas	0	15	5.29	5.18	2.29	0.5
Peatones	0	2.5	1.65	1.68	0.29	-1.97

Para la generación de las colecciones se ha considerado la descripción del movimiento que hacen en [DWF09] para bicicletas (ver cuadro 5.9). Además se han considerado los siguientes valores para los parámetros de generación:

- Total de objetos=$\{12.000, 24.000\}$
- Movilidad = $\{ 80\,\%, 20\,\%, 10\,\%\}$
- tamaño mínimo de una celda = $\{1m^2, 10m^2\}$
- un instante es un minuto, y a cada minuto se toman muestras del espacio.

En la tabla 5.10 se describe las 36 colecciones sintéticas resultantes de combinar los diferentes valores de los parámetros.

A continuación se presenta el análisis de los resultados experimentales con dichas colecciones.

Tabla 5.10: Descripción de las colecciones sintéticas utilizadas.

Nombre	**Objetos**	**Movilidad**	**N**	m^2 **celda**	**m**	$\mathbf{H_{-1}}$	$\mathbf{H_0}$	$\mathbf{mH_0}$ en MB
Bici_1	12.000	80 %	100.000	1	82.009.155	60,06	60,04	615,54
Bici_2	12.000	80 %	100.000	10	95.022.818	60,06	60,04	713,25
Bici_3	12.000	80 %	10.000	1	20.713.892	53,42	53,17	137,68
Bici_4	12.000	80 %	10.000	10	79.750.304	53,42	53,32	531,61
Bici_5	12.000	80 %	1.000	1	1.160.333	46,78	45,65	6,62
Bici_6	12.000	80 %	1.000	10	18.698.912	46,78	46,47	108,63
Bici_7	12.000	20 %	100.000	1	20.551.751	60,06	60,04	154,26
Bici_8	12.000	20 %	100.000	10	23.795.265	60,06	60,04	178,59
Bici_9	12.000	20 %	10.000	1	5.196.290	53,42	53,24	34,58
Bici_10	12.000	20 %	10.000	10	20.266.855	53,42	53,38	135,23
Bici_11	12.000	20 %	1.000	1	301.104	46,78	45,41	1,70
Bici_12	12.000	20 %	1.000	10	4688375	46,78	46,52	27,26
Bici_13	12.000	1 %	100.000	1	1.035.400	60,06	59,80 %	7,74
Bici_14	12.000	1 %	100.000	10	1.205.442	60,06	59,48	8,96
Bici_15	12.000	1 %	10.000	1	285.623	53,42	52,74	1,88
Bici_16	12.000	1 %	10.000	10	1.025.489	53,42	52,88	6,77
Bici_17	12.000	1 %	1.000	1	25.568	46,30	39,93	0,12
Bici_18	12.000	1 %	1.000	10	241.335	46,78	44,77	1,35
Bici_19	24.000	80 %	100.000	1	164.173.042	61,06	61,04	1.252,8
Bici_20	24.000	80 %	100.000	10	189.846.885	61,06	61,04	1.448,77
Bici_21	24.000	80 %	10.000	1	41.629.851	54,42	54,17	281,93
Bici_22	24.000	80 %	10.000	10	15.983.8918	54,42	54,33	1.085,51
Bici_23	24.000	80 %	1.000	1	2.323.324	47,78	46,65	13,55
Bici_24	24.000	80 %	1.000	10	37.615.744	47,78	47,48	223,25
Bici_25	24.000	20 %	100.000	1	41.010.826	61,06	61,05	312,97
Bici_26	24.000	20 %	100.000	10	47.600.809	61,06	61,04	363,25
Bici_27	24.000	20 %	10.000	1	10.387.673	54,42	54,25	70,44
Bici_28	24.000	20 %	10.000	10	40.432.284	54,42	54,38	274,84
Bici_29	24.000	20 %	1.000	1	596.296	47,78	46,43	3,46
Bici_30	24.000	20 %	1.000	10	9.359.532	47,78	47,52	55,59
Bici_31	24.000	1 %	100.000	1	2.069.494	61,06	60,87	15,74
Bici_32	24.000	1 %	100.000	10	2.400.605	61,06	60,55	18,17
Bici_33	24.000	1 %	10.000	1	589.367	54,42	53,80	3,96
Bici_34	24.000	1 %	10.000	10	2.067.056	54,42	53,89	13,92
Bici_35	24.000	1 %	1.000	1	55.136	47,68	41,59	0,28
Bici_36	24.000	1 %	1.000	10	499.269	47,78	45,82	2,86

5.6.1. Análisis del espacio utilizado

Para el análisis del espacio se compararán las dos versiones del índice con el MVR-Tree. En el caso del MVR-Tree se ha considerado una capacidad del nodo de 60, pues es la que da un mejor trade-off espacio-tiempo. En el caso de nuestra propuesta, se han considerado diferentes valores para la frecuencia entre snapshot, con valores entre 30 y 240, dado que dicha frecuencia afecta el tamaño del índice. Junto con la frecuencia, los otros factores que afectan el tamaño del índice son: el tamaño del *grid*, el número de objetos, el tamaño de la celda y la cantidad de objetos.

En la figura 5.3 se presenta un gráfico por cada escenario, dado por el ancho del *grid* (filas) y la movilidad de la colección (columnas), considerando una celda de $10m^2$ y una colección de 12.000 objetos. El eje x de cada gráfico representa la frecuencia entre snapshot, donde el valor 0 corresponde al MVR-Tree y los valores desde el 30 al 240 a nuestra estructura.

Al observar la figura 5.3 se puede observar que la movilidad de los objetos tiene un gran impacto en el tamaño de los índices. Cuando existe una baja movilidad de los objetos (columna 1), el MVR-Tree tiene un tamaño menor que nuestra estructura, independientemente del tamaño del *grid*. En cambio, en la medida que aumenta la movilidad, el MVR-Tree se ve obligado a crear muchos nodos con intervalos pequeños aumentando significativamente de tamaño. En cambio nuestra estructura, en la medida que aumenta la movilidad, el aumento de tamaño no es tan brusco como el MVR. Al comparar ambas versiones de nuestra estructura se puede observar que la movilidad afecta mucho más a la versión codificada en unario que la codificada con Elías γ, esto se debe a que los desplazamientos tienen una distribución de probabilidad que se ajusta mejor a a Elías γ que a unarios.

Al observar el tamaño de las estructuras cuando aumenta el tamaño del *grid* se puede observar que en el caso de baja movilidad, el MVR-Tree no se ve afectado por el tamaño del *grid*, pero cuando la movilidad es del 20 % o más, si se observa un impacto en el tamaño. En cambio en nuestra estructura el impacto es mucho menor. Si bien el tamaño del *grid* afecta a la altura del k^2-tree, y por ende al tamaño de los *snapshots*, esto no afecta a las bitácoras y por lo tanto el tamaño total del índice no se vea muy afectado por un cambio en el tamaño del *grid*.

El análisis anterior también es válido, si aumentamos el número de objetos, es decir, se mantiene las mismas diferencias, y lo único que ocurre es un escalamiento del tamaño de las estructuras proporcional al número de objetos. Esto se puede apreciar al observar la figura 5.4 en la cual se a duplicado el número de objetos.

Otro factor importante a analizar es el tamaño de la celda más pequeña. Si se disminuye el tamaño de la celda a $1m^2$, esto no tiene mayor impacto en el MVR-Tree, pero sí en nuestro índice, en especial cuando se codifica en unario. Esto se debe a que al disminuir la celda, los valores a codificar serán mucho más grandes y por lo

tanto más costosos. Si se observa la figura 5.5, donde el tamaño de la celda es de $1m^2$ y existen 24.000 objetos, se puede ver que las bitácoras codificadas en unario son mucho más costosas que el MVR-Tree en este escenario. Por otro lado, las bitácoras codificadas con Elías γ, en este escenario, no muestran dicho problema.

En cuanto a la frecuencia entre *snapshots* , se mantiene lo observado con anterioridad, que en la medida que sea mayor la distancia, el tamaño del índice disminuye.

Luego de analizar las diferentes variables que determinan el tamaño del índice es posible concluir que nuestra versión del índice, desde la perspectiva del espacio, es superior al MVR-Tree en un entorno de alta movilidad (20 % o más).

5.6.2. Análisis del tiempo de consulta

Para el análisis del tiempo de consulta se comparó el desempeño de nuestra estructura contra el MVR-Tree, en distintos escenarios.

El primero de ellos está dado por una colección de 24.000 objetos, con 80 % de movilidad y un tamaño de celda de $10m^2$. Se escogió este escenario primero porque es donde encontramos la mayor diferencia a favor de nuestra estructura desde la perspectiva del almacenamiento, es decir es nuestro mejor caso frente al MVR-Tree. Si se observa la tabla 5.11 en la consulta de *time slice* con área 1.000 es la única consulta donde el MVR-Tree supera a nuestra estructura, pero por un estrecho margen (5μs). En cambio, en el resto de las consultas, nuestra estructura tiene un mejor desempeño, llegando a ser 10 veces más rápida en algunas consultas. Si se observan las consultas de *time slice*, la diferencia es muy poca, salvo cuando el área de la consulta es una celda, donde nuestra estructura es 16 veces más rápida. En el caso de las consultas de *time interval* (duración de 10 en adelante), nuestra estructura supera ampliamente al MVR-Tree.

Si la movilidad baja a un 20 % y se mantiene el mismo número de objetos y el tamaño de la celda, nuestra estructura sigue siendo mejor que le MVR-Tree en las consultas de *time interval*, pero pierde en la mayoría de los casos para las consultas de *time slice*. Aun así, nuestra estructura es mejor que el MVR-Tree cuando el área de la consulta es del tamaño de una celda, por una amplia ventaja. Estos resultados se pueden ver en la tabla 5.12, donde se han puesto en negrita los mejores tiempos.

En el caso de una movilidad de un 1 %, mantieniendo los demás parámetros iguales, el MVR-Tree supera a nuestra estructura en la mayoría de los casos. En la tabla 5.13 se muestran los tiempos para cada consulta.

Tabla 5.11: Comparación del tiempo de CPU promedio en μ segundos para consultas de *time slice* y *time interval* para distintos largos de bitácoras en nuestra propuesta. Considerando una colección con 24.000 objetos, **80 % de movilidad** y un tamaño de celda de $10m^2$

Tipo consulta		MVR-Tree	**Bitácora con Elías**		
Duración	**Área**	**60**	**64**	**128**	**256**
0	1	4.825,86	3.885,54	294,96	**288,82**
	10	50,28	34,88	**29,92**	31,14
	100	49,78	36,58	**35,18**	36,34
	1.000	**54,78**	60,84	59,52	61,18
10	1	222,52	35,46	**34,32**	35,36
	10	233,74	37,08	**36,62**	37,36
	100	229,02	46,3	**45,5**	46,72
	1.000	259,3	81,62	**80,14**	83,6
100	1	1.725,52	**197,16**	327,98	336,04
	10	1.671,9	**209**	362,18	370,44
	100	1.757,22	**231,08**	415,72	425,34
	1000	2.099,66	**358,96**	582,7	595,92
1.000	1	1.9809,88	**1.940,56**	3.511,44	7.458,62
	10	20.371,44	**1.966,94**	3.574,12	7.765,28
	100	20.205,46	**1.906,92**	3.430,78	7.598,8
	1000	24.304,32	**2.663,82**	4.408,16	9.040,96

Tabla 5.12: Comparación del tiempo de CPU promedio en μ segundos para consultas de *time slice* y *time interval* para distintos largos de bitácoras en nuestra propuesta. Considerando una colección con 24.000 objetos, **20 % de movilidad** y un tamaño de celda de $10m^2$

Tipo consulta		MVR-Tree	**Bitácora con Elías**		
Duración	**Área**	**60**	**64**	**128**	**256**
0	1	1.416,38	1.205,12	415,34	**405,92**
	10	**41,04**	52,66	52,82	53,44
	100	**42,26**	71,26	88,26	91,48
	1.000	**47,74**	50,9	153,34	160,16
10	1	76,44	36,82	37,1	**36,6**
	10	105,64	**53,58**	55,08	55,12
	100	104,06	**78,68**	79,94	82,92
	1.000	117,02	**93,4**	151,98	159,36
100	1	668,44	*145,44*	162,82	170,76
	10	625,8	**185,3**	210,9	214,28
	100	674,1	**194,8**	236,02	259,92
	1.000	798,64	**230,48**	365,9	397,48
1.000	1	6.948,54	**1.197,54**	1.413,64	1.737,42
	10	6.818,98	**1.191,86**	1.395,98	1.725,46
	100	6.939,4	**1.233,94**	1.471,42	1.816,3
	1.000	7.605,7	**1.815,36**	1.970,6	2.253,64

Tabla 5.13: Comparación del tiempo de CPU promedio en μ segundos para consultas de *time slice* y *time interval* para distintos largos de bitácoras en nuestra propuesta. Considerando una colección con 24.000 objetos, **1 % de movilidad** y un tamaño de celda de $10m^2$

Tipo consulta		MVR-Tree	**Bitácora con Elías**		
Duración	**Área**	**60**	**64**	**128**	**256**
0	1	**69,42**	386,1	414,5	310
	10	**16,24**	42,18	57,26	79,28
	100	**41,88**	45,88	61,34	80,58
	1.000	79,88	**68,22**	85,16	104,6
10	1	44,94	**42,88**	54,18	64,72
	10	**44,9**	52,88	62,06	78,08
	100	**46,3**	48,06	65,28	79,9
	1.000	**52,04**	76,04	90,2	100,12
100	1	**74,32**	108,62	93,7	88,98
	10	**51,54**	122,74	116,7	121,94
	100	**79,94**	117,94	114,64	98,82
	1.000	**91,86**	173,88	158,28	142,08
1.000	1	**379,6**	743,12	515,18	371,48
	10	**358,54**	763,2	544,62	385,96
	100	**350,12**	769,44	542,9	368,62
	1.000	**377,06**	1.151,4	791,62	502

5.7. Discusión

En este capítulo se ha presentado una versión mejorada del índice propuesto en el capítulo anterior, el cual permite la indexación de objetos sin restricción de movimientos.

Como se puede ver en la sección experimental, al comparar el índice con la colección de datos reales, el índice permite la indexación sucinta de los puntos, alcanzando para algunas configuración un tamaño inferior a nH_0. Si bien los tiempos son bajos (del orden de los milisegundos en el peor de los casos), el índice no se comporta bien si lo comparamos con el MVR-Tree. Esto se debe principalmente a que la colección de barcos, presenta una baja movilidad, además que son pocos objetos.

Complementando lo anterior se ha presentado un estudio con dato sintéticos para evaluar el desempeño de nuestra estructura bajo diferentes configuraciones de los datos.

Los experimentos demuestran que, para una colección de 24.000 objetos y un tamaño de celda de $10m^2$, nuestra estructura supera en tiempo y espacio al MVR-Tree cuando la movilidad es de un 80 %, en la gran mayoría de las consultas de *time slice* como de *time interval*. Por otro lado, cuando la movilidad es de un 20 % nuestra estructura es mejor en todas las consultas de *time interval* y competitiva en las de *time slice*, a la vez que utiliza menos espacio. Pero, en colecciones de baja movilidad (1 %) nuestro índice no presenta mejores resultados que el MVR-Tree, ni en espacio ni en tiempo. Lo anterior escala en el número de objetos.

Además, nuestra estructura permite recuperar la trayectoria de un objeto de manera eficiente, consulta que en un MVR-Tree implicaría recorrer todos los nodos para poder responderla.

Si bien este índice supera los problemas presentados en la versión anterior, y es mejor que el MVR-Tree en colecciones con una alta movilidad es necesario estudiar una mejor manera de indexar los datos, con el fin de minimizar el impacto que tiene, sobre las consultas de *time slice* y *time interval*, y así poder ser más competitivo cuando la movilidad es más baja.

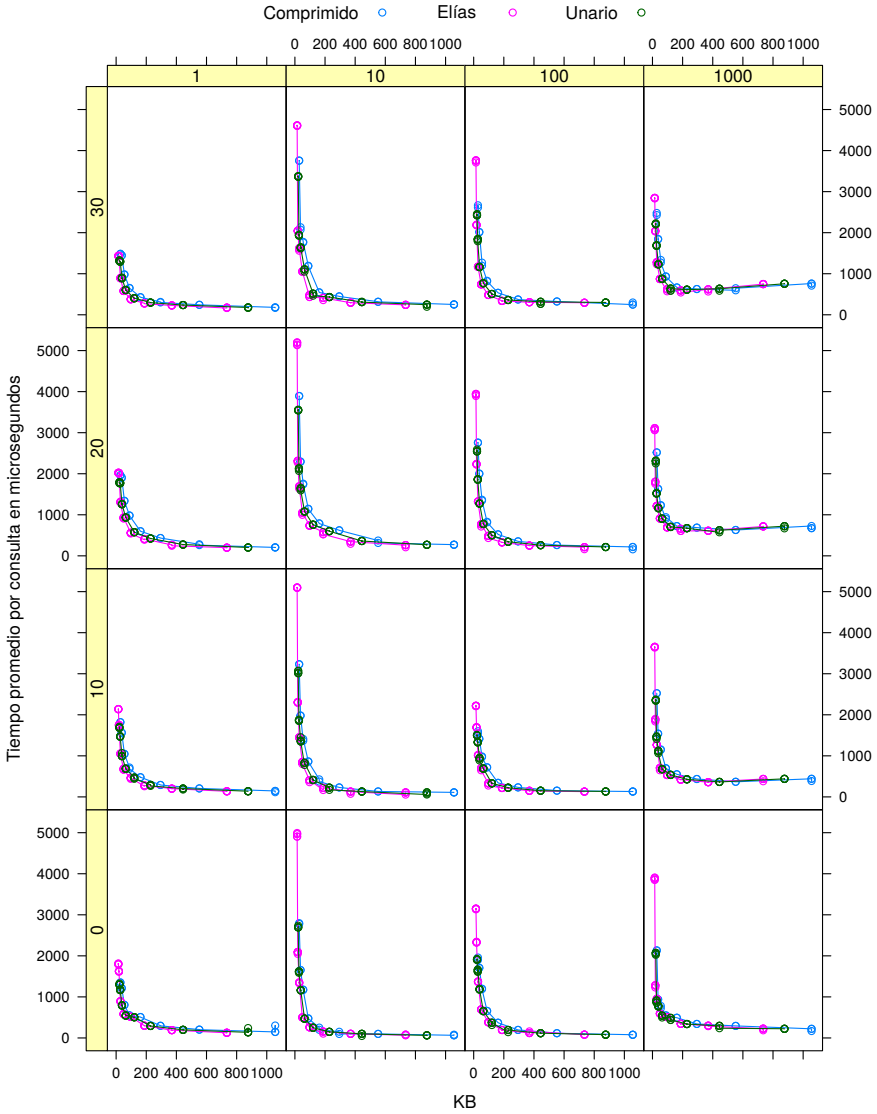


Figura 5.2: Gráfico del trade-off espacio-tiempo con las tres versiones de bitácoras para cada escenario, configurado por la duración de la consulta (filas en el rango [0;30]) y el tamaño del área consultada (columnas en el rango[1;1.000]).

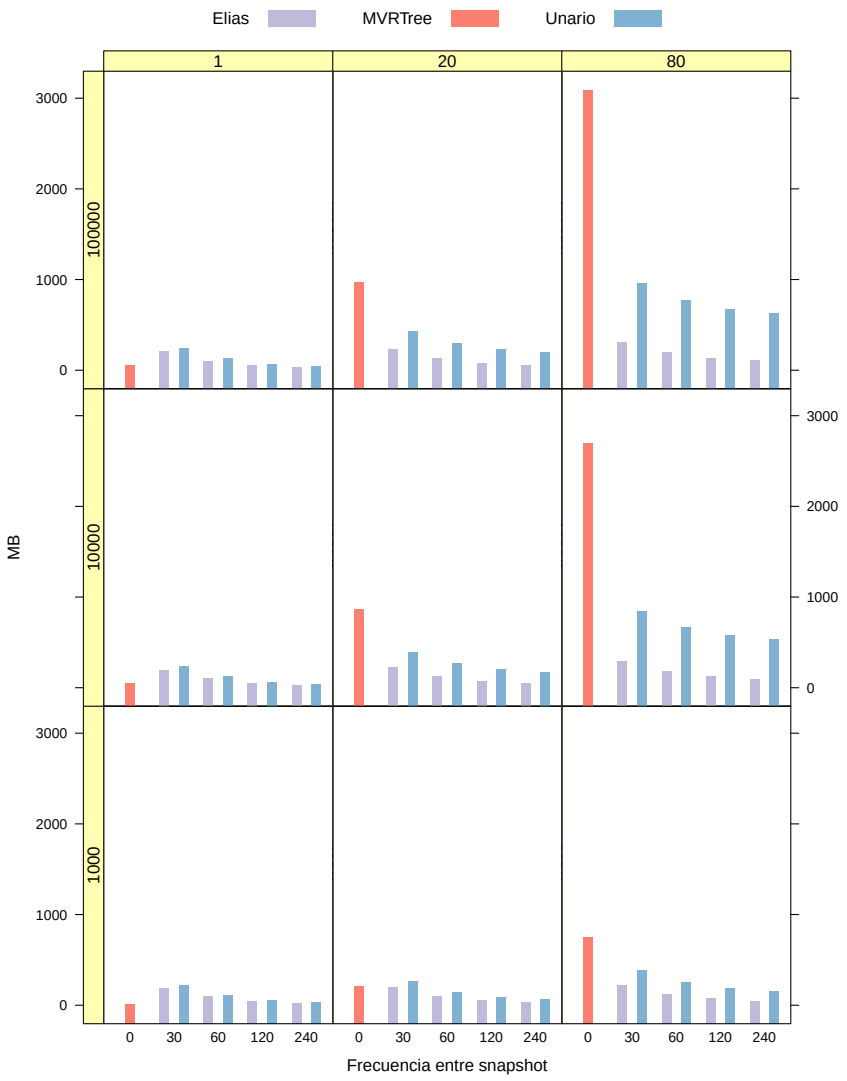


Figura 5.3: Tamaño del índice MVR-Tree, comparado con nuestra propuesta (Elías y Unarios) para celda de $10m^2$ y 12.000 objetos. Las filas de la matriz de gráficos corresponde al ancho del grid espacial(en el rango [1.000;100.000] y las columnas representan la movilidad como un porcentaje del total de objetos (1 %, 20 % y 80 %).

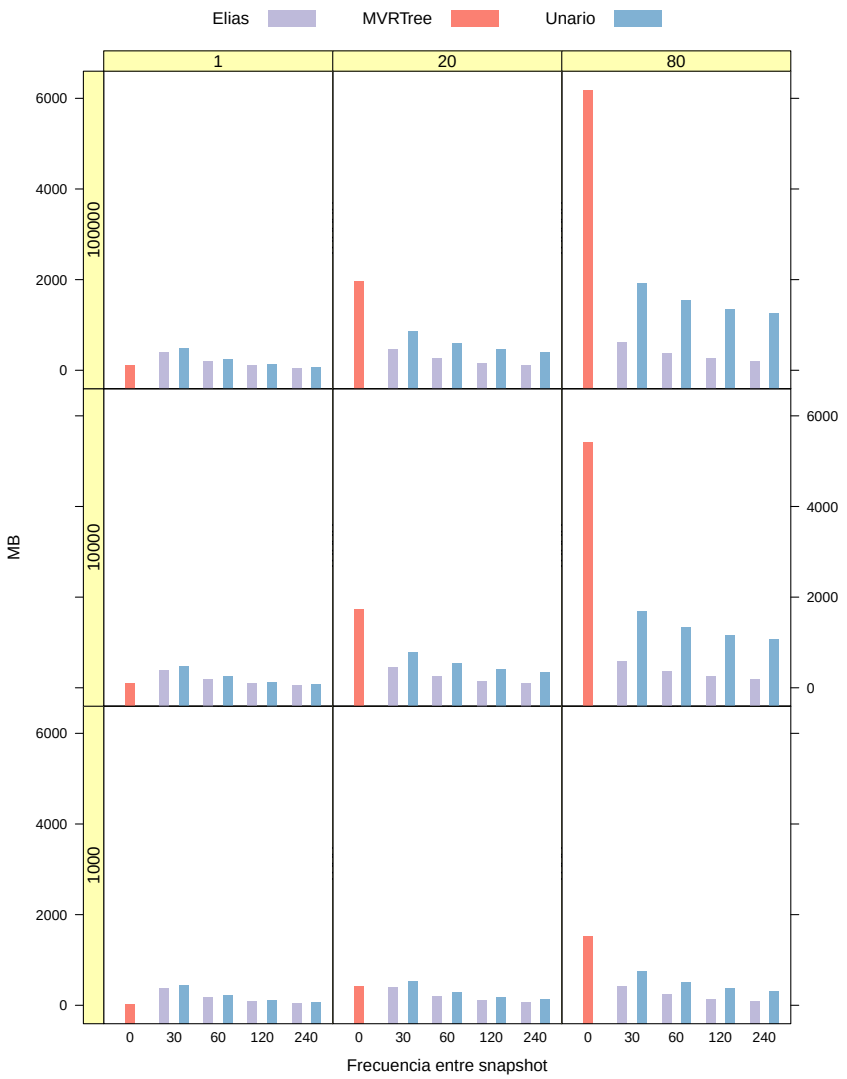


Figura 5.4: Tamaño del índice MVR-Tree, comparado con nuestra propuesta (Elías y Unarios) para celda de $10m^2$ y 24.000 objetos. Las filas de la matriz de gráficos corresponde al ancho del grid espacial(en el rango [1.000;100.000] y las columnas representan la movilidad como un porcentaje del total de objetos (1 %, 20 % y 80 %).

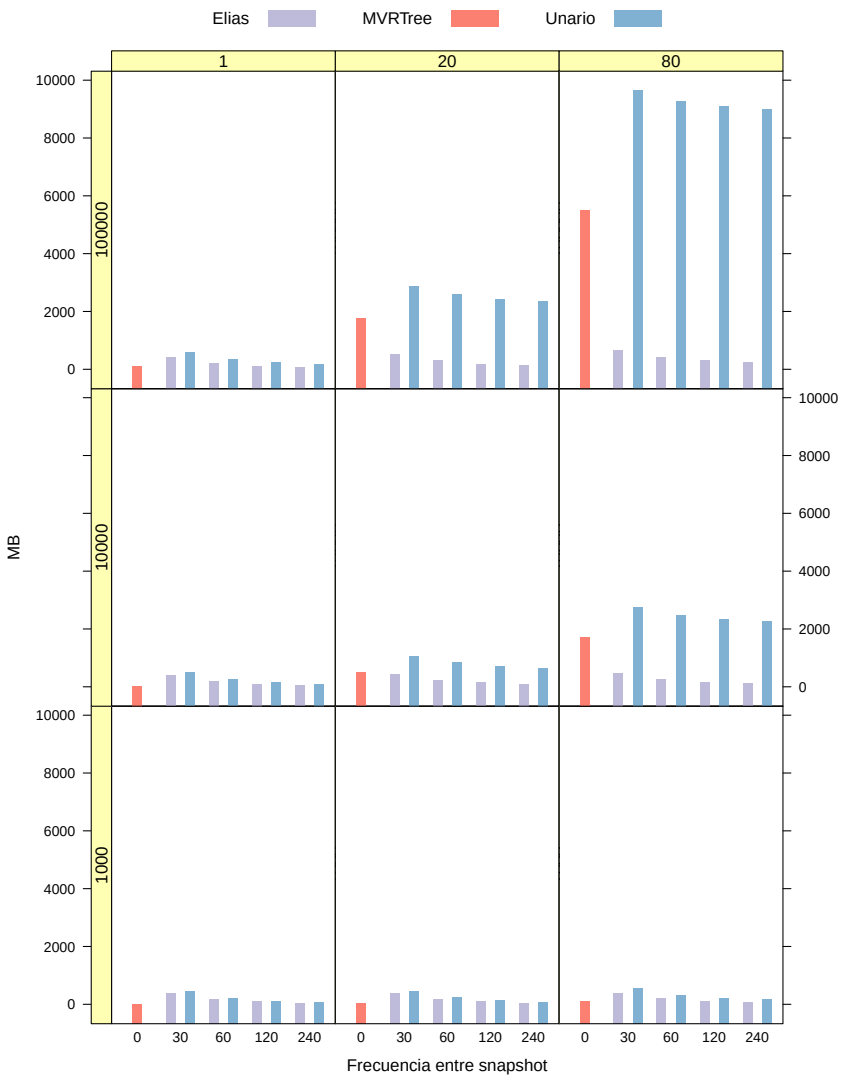


Figura 5.5: Tamaño del índice MVR-Tree, comparado con nuestra propuesta (Elías y Unarios) para celda de $1m^2$ y 24.000 objetos. Las filas de la matriz de gráficos corresponde al ancho del grid espacial(en el rango [1.000;100.000] y las columnas representan la movilidad como un porcentaje del total de objetos (1 %, 20 % y 80 %).

Capítulo 6

Conclusiones

La indexación de objetos móviles ha sido un área de investigación y desarrollo muy activa durante los últimos 30 años. Actualmente, dado el avance tecnológico en el ámbito de los dispositivos móviles, sensores y redes de comunicación a propiciado un aumento en la producción de datos espacio-temporales, y por consiguiente, en la necesidad de contar con nuevos y más eficientes sistemas de indexación.

En esta tesis se abordó la problemática de indexar objetos móviles desde una nueva perspectiva y que es el desarrollo de un auto-índice para memoria principal utilizando estructuras de datos compactas.

Una primera aportación de esta tesis fue la ampliación de las capacidades del k^2-tree para indexar objetos y sus ubicaciones, utilizando para ello, una estructura de datos compacta adicional para indexar los identificadores de objetos y vincularlos sus puntos. Además, se modificó el algoritmos de consultas por rango de un k^2-tree, para que la respuesta fueran todos los objetos y sus ubicaciones vinculadas al rango respectivo. Junto con esto se presenta un algoritmo nuevo que permite obtener la ubicación de un objeto dado su identificador.

La principal contribución de esta tesis es un nuevo índice compacto que permite almacenar e indexar objetos móviles cuya ubicación en el espacio es modelada como un punto. Con los datos experimentales se ha podido observar que la estructura propuesta es eficiente en el uso del espacio, logrando, para algunas configuraciones consumir un espacio inferior a nH_0, lo que es su principal virtud.

Se presentan algoritmos que permiten responder consultas de *time slice*, *time interval*, recuperar la trayectoria de un objeto en un intervalo, obtener la ubicación de un objeto en un instante y obtener los k objetos más cercanos a un punto dado en un cierto instante. Cabe destacar que la mayoría de los sistemas de indexación

responden consultas de tipo trayectoria o bien de tipo *time slice, time interval*, en cambio el índice propuesto puede responder a ámbos tipos de consulta.

De estas consultas se estudiaron experimentalmente las consultas de *time slice* y *time interval* y se comparó el rendimiento con el MVR-Tree.

El resultado de esta comparación indica que nuestro índice que restringe los movimientos de los objetos a casillas adyacentes, supera al MVR-Tree tanto en espacio como en tiempo de respuesta la las consultas estudiadas.

En el caso de la versión sin restricción de los movimientos, se evaluó tanto con datos reales como sintéticos. En la coleción real, el MVR-Tree supera ampliamente a nuestra estructura en los tiempos de respuesta, sin embargo, nuestra estructura aún mantiene tiempos razonables, ocupando 6 milisegundos para la consulta más costosa. En cambio al comparar el almacenamiento, nuestra propuesta ocupa entre el 6 % y el 20 % del espacio que necesita el MVR-Tree para los mismos datos.

Esta importante diferencia en el coste de almacenamiento hace que el índice propuesto sea útil en bases de datos grandes, donde el MVR-Tree se vería obligado a trabajar con los datos en el disco, en cambio nuestra estructura podría trabajar en memoria principal, sacando ventaja de la jerarquía de memoria.

Al estudiar la estructura con datos sintéticos, utilizando una colección más grande, de 24.000 objetos, los experimentos demuestran que nuestra estructura es superior al MVR-Tree cuando la movilidad de los objetos es alta, tanto en las consultas de *time slice* como de *time interval*, así como tambien en el espacio almacenado. Al bajar la movilidad a un 20 %, nuestra estructura supera al MVR-Tree en todas las consultas de *time interval*, pero no así en las de *time slice*. Aun así, las diferencias son pequeñas y junto con ello, el espacio utilizado por nuestra estructura es mucho menor y por lo tanto muy competitiva en la práctica.

El peor escenario para nuestra estructura es uno de baja movilidad (1 %). En el, los experimentos muestran que el MVR-Tree supera a nuestra estructura tanto en tiempo como en espacio.

El principal problema del enfoque utilizado para el diseño de la estructura, está en el marcado *trade-off* espacio-tiempo definido por la distancia entre dos *snapshot*. Esto hace que alcanzar una buena compresión y a la vez de un rápida respuesta sea muy difícil. La causa de ésto está en la necesidad de ampliar el rango espacial de la consulta, la cual puede incluir a todos los objetos indexados si la distancia entre *snapshot* es muy grande. Esto se da independiente del tipo de bitácora que se emplee, dado que es un problema del modelo subyacente.

6.1. Trabajo futuro

Como trabajo futuro se proponen algunas mejoras al índice propuesto:

Índice con múltiples tipos de bitácora. Una primera mejora es crear un índice que, al momento de la creación de las bitácoras escoja la más adecuada al tipo de movimiento, por ejemplo para aquellos objetos que se mueven a lo más una celda por instante son más convenientes las bitácoras presentadas en el capítulo 4. Entonces, la idea es contar con una superclase de bitácoras que tenga varias sub-clases, una por cada propuesta presentada en esta tesis, posibilitando contar con más bitácoras. las cuales serán intanciadas según las características del movimiento particular que ha tenido un objeto entre dos *snapshot*.

Utilizar 2 o más k^2-tree en los *snapshots*. Definir un tipo de snapshot que mantenga separadamente, en 2 k^2-tree los objetos que no se han movido hasta el próximo *snapshot*, de los que si lo han hecho. Esto debería mejorar los tiempos de respuesta en las consultas del tipo *time slices* y *time interval*, dado que en colecciones de baja movilidad, la mayoría de los objetos no se habrán movido y por lo tanto, bastará con una consulta por rango utilizando el rango original de la consulta, sin la necesidad de ampliarla sobre estos datos y el resultado sería definitivo. Por otro lado, con los objetos que si se han movido habría que hacer la consulta ampliada, pero sería más rápida al incluir menos puntos. Así la suma de los tiempos debería ser mejor. Se puede extender la idea a m k^2-tree con la finalidad de agrupar objetos que se muevan a velocidades máximas acotadas, por ejemplo, al primer cuartil, a la mediana, al tercer cuartil y el resto de las velocidades máximas, donde las consultas ampliadas serían de diferentes tamaños, pero las más grandes operarían sobre menos objetos comparado con usar un único k^2-tree. Este particionamiento se podría aprovechar para realizar las consultas en paralelo, disminuyendo aún más los tiempos.

Paralelizar la estructura. El índice propuesto se puede paralelizar con mucha facilidad. Esto por una parte, dada la separación entre snapshot y bitácoras, y por otro lado la independencia entre las bitácoras. Aunque dicha característica no fue explotada en la versión del índice propuesto, se espera poder hacerlo como trabajo futuro. Para ello sería necesario adaptar el resultado de las consultas por rango para que el resultado se de como un flujo, de modo que se evalúen los objetos a medida que se obtienen resultados y no sea necesario esperar hasta el último candidato para comenzar a evaluar el primero. Dado que las bitácoras del índice son estructuras independientes entre si, es muy sencillo paralelizar la etapa de actualización y evaluación de los objetos candidatos, luego de una consulta por rango.

Ampliar las capacidades del índice con otro tipo de consultas. Por ejemplo se podrían incluir consultas como, el par de vecinos más cercanos, la cerradura convexa, consultas de agregación espacial como por ejemplo, contar el flujo de objetos en una determinada área del espacio en un determinado intervalo de tiempo, entre otras.

Utilizar alguna alternativa al k^2-tree. Evaluar una estructura de datos diferente para la indexación espacial de objetos, en particular, el *Wavelet Tree*.

Indexación de rectángulos. Incorporar la posibilidad de indexar objetos cuya ubicación sea modelada como un rectángulo en vez de un punto para ampliar el ámbito de aplicación del índice.

Indexación de objetos sobre redes . Siguiendo con el esquema de *snapshot* y bitácoras, es posible indexar puntos sobre redes si se codifican las bitácoras utilizando el índice *Compact Trip Representation*(CTR) [BFGR16]. Esto permitiría incorporar consultas de *time slice* y *time interval* en CRT, consultas que actualmente no tiene.

Apéndice A

Cálculo de la entropía empírica utilizando el software R

R es un software para análisis de datos, cálculos y gráficas que además es programable. Existen varios paquetes que extiende las capacides básicas de R. Uno de estos es la biblioteca de funciones *entropy* que permite calcular la entropía empírica de un conjunto de datos, entre otras funciones que calculan la entropía. La documentación oficial se encuentra en `https://cran.r-project.org/web/packages/entropy/entropy.pdf`.

Este apéndice tiene como finalidad describe de manera sencilla como, a partir de un archivo con datos en formato tabular, calcular la entropía empírica para cada columna.

Pensando en aquellas personas que no conozcan R, se comenzará con una breve descripción de como instalar R en linux y como instalar el paquete *entropy* en R.

A.1. Instalar R (en shell)

Para la instalación de R en linux hay que utilizar las siguientes instrucciones:

```
1  :~$ sudo apt-get update
2  :~$ sudo apt-get install r-base
```

para abrir un terminar de R y comenzar a trabajar:

```
:~$ R
```

si R está bien instalado debería salir el siguiente mensaje:

```
R version 3.0.2 (2013-09-25) -- "Frisbee Sailing"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R es un software libre y viene sin GARANTIA ALGUNA.
Usted puede redistribuirlo bajo ciertas circunstancias.
Escriba 'license()' o 'licence()' para detalles de distribucion.

R es un proyecto colaborativo con muchos contribuyentes.
Escriba 'contributors()' para obtener más información y
'citation()' para saber cómo citar R o paquetes de R en publicaciones.

Escriba 'demo()' para demostraciones, 'help()' para el sistema on-line de ayuda,
o 'help.start()' para abrir el sistema de ayuda HTML con su navegador.
Escriba 'q()' para salir de R.
```

A.2. Uso de R para cálculo de la entropía

A.2.1. Carga de la biblioteca entropy

Para calcular la entropía con R, lo primero que hay que hacer es cargar la biblioteca «entropy»:

```
> library("entropy")
```

si el paquete no está instalado nos dará el siguiente error:

```
Error en library("entropy") : there is no package called 'entropy'
```

Para instalar el paquete «entropy» hay que hacer lo siguiente:

```
> install.packages("entropy")
```

Al instalar el paquete «entropy» puede ocurrir que no tengamos los permisos necesarios para instarlo en las carpetas por omisión, si ese es el caso, durante la instalación se le preguntará si se instala de forma local; hay que contestar que sí («y») a las dos preguntas que hace para instalar el paquete en este caso.

```
Installing package into '/usr/local/lib/R/site-library'
(as 'lib' is unspecified)
Aviso en install.packages("entropy") :
 'lib = "/usr/local/lib/R/site-library"' is not writable
Would you like to use a personal library instead?  (y/n) y
Would you like to create a personal library
~/R/x86_64-pc-linux-gnu-library/3.0
to install packages into?  (y/n) y
--- Please select a CRAN mirror for use in this session ---
probando la URL 'http://dirichlet.mat.puc.cl/src/contrib/entropy_1.2.1.tar.gz'
Content type 'application/x-gzip' length 13572 bytes (13 Kb)
URL abierta
==================================================
downloaded 13 Kb

* installing *source* package 'entropy' ...
** package 'entropy' successfully unpacked and MD5 sums checked
** R
** preparing package for lazy loading
** help
*** installing help indices
** building package indices
** testing if installed package can be loaded
* DONE (entropy)

The downloaded source packages are in
  '/tmp/Rtmpx6qp5R/downloaded_packages'
```

Luego volvemos a cargar la bibliteca «entropy»

```
> library("entropy")
>
```

Para calcular la entropía es necesario contar con la frecuencia de cada símbolo del alfabeto fuente en el mensaje. Para ello, primero tenemos que cargar los datos en memoria y tenemos dos maneras de hacerlo: la primera es poner los datos a mano y la segunda es cargar un archivo.

A.2.2. Digitar los datos y calcular la entropía

Si contamos con las frecuencias absolutas para cada símbolo, y queremos digitarlas en R, esto lo hacemos de la siguiente manera:

```
> frecuencias=c(1, 13, 1, 8, 1, 3, 4, 5, 2, 2, 3, 3, 3, 6)
```

«c(...)» es el constructor que nos permite crear una secuencia de valores en R, dicha secuencia es almacenada en la variable «frecuencias» mediante el operador de asignación. Para ver el contenido de la variable que hemos creado digitamos su nombre:

```
> frecuencias
 [1]  1 13  1  8  1  3  4  5  2  2  3  3  3  6
```

La función para el cálculo de la entropía empírica se denomina «entropy.empirical», y tiene dos parámetros importantes, el primero es la frecuencia absoluta de cada símbolo y el segundo es la base del logaritmo que utilizaremos, que en nuestro caso son logaritmos en base 2.

En el siguiente ejemplo calculamos la entropía empírica para la variable frecuencias usando logaritmo en base 2:

```
> entropy.empirical(frecuencias, unit="log2")
[1] 3.413275
```

A.2.3. Subir un archivo y calcular entropía

Si tenemos un archivo llamado «encuesta.dat» donde la primera línea es tiene los nombres de las columnas, lo podemos cargar en R de la siguiente manera:

```
datos<-read.table("encuesta.dat", header=T)
```

datos es el nombre de la variable que guardará el archivo en R. read.table es una función que nos permite leer datos desde el disco. Los parámetros de la función son: 1) el nombre del archivo a leer, que en el ejemplo anterior se llama «encuesta.dat» y 2) indicar si el archivo tiene una cabecera con nombres o no, en el ejemplo si lo tiene y por lo tanto hay que poner «header=t». Para ver el contenido de la variable datos y comprobar que se ha cargado correctamente el archivo, ponemos su nombre así:

```
> datos
   Row edad sexo escuela programa creditos  gpa familia hestud htv
1    1   21    f    públ     biol      119 3.60       3     35  10
2    2   18    f    priv     mbio       15 3.60       3     30  10
3    3   19    f    priv     biot       73 3.61       5      5   7
4    4   20    f    priv     mbio        * 2.38       3     14   3
5    5   21    m    públ     pmed      114 3.15       2     25  25
6    6   20    m    públ     mbio       93 3.17       3     17   6
7    7   22    m    públ     pmed      120 2.15       5     20  10
8    8   20    m    priv     pmed        * 3.86       5     15   5
9    9   20    m    priv     pmed       94 3.19       4     10   2
10  10   20    f    públ     pmed      130 3.66       6     20  33
11  11   21    f    priv     mbio       97 3.35       1     15  20
12  12   20    m    priv     mbio       64 3.17       4     30   2
13  13   20    f    públ     mbio        * 3.23       2      5   3
14  14   21    f    públ     mbio       98 3.36       4     15  10
15  15   21    f    priv     biol      113 2.88       5     15   3
16  16   21    f    priv     pmed      124 2.80       5     20  10
```

```
19  17  17   20    f    públ     eagr        * 2.50       4     10   5
20  18  18   20    f    priv     mbio        * 3.46       4     18   5
21  19  19   22    f    priv     pmed      120 2.74       2     10  15
22  20  20   20    f    priv     mbio       95 3.07       3     15  12
23  21  21   22    f    priv     biol      125 2.20       3     20  10
24  22  22   23    m    públ     eagr       13 2.39       3     10   8
25  23  23   21    m    priv     pmed      118 3.05       4     10  10
26  24  24   20    f    públ     mbio      118 3.55       5     38  10
27  25  25   21    f    públ     mbio      106 3.03       5     36  35
28  26  26   20    f    priv     mbio      108 3.61       3     20  10
29  27  27   22    f    públ     mbio      130 2.73       5     15   2
30  28  28   21    f    priv     pmed      128 3.54       3     18   5
```

Lo que desplegará el contenido del archivo.

Luego, para poder acceder a cada columna del archivo por su nombre es necesario indicar :

```
1  > attach(datos)
```

Un comando útil de R es «table» el cual calcula la tabla de frecuencias absolutas de una variable. Por ejemplo si queremos calcular la tabla de frecuencias para la columna familia sería:

```
1  > table(familia)
2  familia
3  1 2 3 4 5 6
4  1 3 9 6 8 1
```

y para la columna edad sería:

```
1  > table(edad)
2  edad
3  18 19 20 21 22 23
4   1  1 12  9  4  1
```

Con los datos cargados en memoria y con la ayuda de la función «table» que calcula las frecuencias absolutas es posible calcular la entropía empírica de los datos cargados sin mayores dificultades. En el siguiente ejemplo se calcula la entropía empírica para la variable familia usando logaritmo en base 2:

```
1  > entropy.empirical(table(familia), unit="log2")
2  [1] 2.20757
```

Otro ejemplo:

```
1  > entropy.empirical(table(edad), unit="log2")
```

```
2 [1] 1.966324
```

A.2.4. Salir de R

Finalmente para salir de R ponemos:

```
1 >q()
2 Save workspace image? [y/n/c]: n
```

Se nos preguntará si queremos guardar las variables que hemos creado con sus respectivos datos, si pones «y» serán guardados, si ponemos «n», no lo será y si ponemos «c» se cancela el comando de salir.

Bibliografía

[AAE03] Pankaj K Agarwal, Lars Arge, and Jeff Erickson. Indexing Moving Points. *Journal of Computer and System Sciences*, 66(1):207–243, 2003.

[ÁGBFMP11] Sandra Álvarez-García, Nieves R Brisaboa, Javier D Fernández, and Miguel A Martínez-Prieto. Compressed k2-Triples for Full-In-Memory RDF Engines. In *17th Americas Conference on Information Systems, AMCIS*. Association for Information Systems, 2011.

[ÁGBLP10] Sandra Álvarez-García, Nieves R Brisaboa, Susana Ladra, and Oscar Pedreira. Almacenamiento y explotacion de grandes bases de datos orientadas a grafos. In *XV Jornadas de Ingeniería del Software y Bases de Datos, JISBD*, pages 187–197. IBERGARCETA Pub. S.L., 2010.

[Alv14] Sandra Alvares. *Compact and Efficient Representations of Graphs*. Phd. thesis, Universidade da Coruña, 2014.

[AMRP12] Mohamed Aly, Mario Munich, Evolution Robotics, and Pietro Perona. CompactKdt: Compact Signatures for Accurate Large Scale Object Recognition. In *IEEE Workshop on Applications of Computer Vision (WACV)*, pages 505–512, Colorado, 2012.

[AS99] Srinivas Aluru and FatihE. Sevilgen. Dynamic Compressed Hyperoctrees with Application to the N-body Problem. In C.Pandu Rangan, V Raman, and R Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science SE - 2*, volume 1738 of *Lecture Notes in Computer Science*, pages 21–33. Springer Berlin Heidelberg, 1999.

[ATS14] Sultan Alamri, David Taniar, and Maytham Safar. A taxonomy for moving object queries in spatial databases. *Future Generation Computer Systems*, 37:232–242, jul 2014.

[BBN14] Nieves R Brisaboa, Guillermo De Bernardo, and Gonzalo Navarro. Interleaved K 2 -tree : Indexing and Navigating Ternary Relations. In *Data Compression Conference (DCC)*, pages 342–351, 2014.

[BdBN12] Nieves R Brisaboa, Guillermo de Bernardo, and Gonzalo Navarro. Compressed Dynamic Binary Relations. In *Data Compression Conference, DCC*, pages 52–61. IEEE Computer Society, 2012.

[BF10] Jérémy Barbay and Johannes Fischer. LRM-Trees: Compressed Indices, Adaptive Sorting, and Compressed Permutations. *arXiv cs.DS*, Sep(29):1–13, sep 2010.

[BFGR16] Nieves R. Brisaboa, Antonio Fariña, Daniil Galaktionov, and M. Andrea Rodríguez. Compact Trip Representation over Networks. pages 240–253. 2016.

[BGO+96] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion {B-tree}. *The VLDB Journal*, 5(4):264–275, 1996.

[Bla06] Daniel K Blandford. *Compact Data Structures with Fast Queries*. Phd thesis, Carnegie Mellon University, 2006.

[BLNS09] Nieves R. Brisaboa, Susana Ladra, Gonzalo Navarro, and Ladra Susana. K^2-tree for compact web graph Representation. In *SPIRE*, volume 5721 of *Lecture Notes in Computer Science*, pages 18–30. Springer, 2009.

[BN13] Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science*, 513:109–123, 2013.

[CEP03] Prasad Chakka, Adam Everspaug, and Jignesh M Patel. Indexing Large Trajectory Data Sets with {SETI}. In *Proceedings of the Intl. Conf. on Management of Innovative Data Systems Research, CIDR*, Asilomar, CA, 2003.

[CL11] Francisco Claude and Susana Ladra. Practical representations for web and social graphs. In *20th ACM Conference on Information and Knowledge Management, CIKM*, pages 1185–1190. ACM, 2011.

[Cla96] David Clark. *Compact Pat Trees*. Phd thesis, University of Waterloo, 1996.

[Cla13] Francisco Claude. *Space-Efficient Data Structures for Information Retrieval*. Phd thesis, University of Warerloo, 2013.

[CRBF15] Diego Caro, M. Andrea Rodríguez, Nieves R. Brisaboa, and Antonio Fariña. Compressed $k^d - tree$ for temporal graphs. *Knowledge and Information Systems*, 2015.

[Dan06] Lin Dan. *Indexing and Querying Moving Objects Databases.* Phd thesis, National University of Singapore, 2006.

[dB14] Gillermo de Bernardo. *New Data Structures and Algorithms for the Efficient Management of Large Spatial Datasets.* Phd., Universidade da Coruna, 2014.

[DWF09] Somayeh Dodge, Robert Weibel, and Ehsan Forootan. Revealing the physics of movement: Comparing the similarity of movement characteristics of different types of moving objects. *Computers, Environment and Urban Systems*, 33(6):419–434, 2009.

[FBN05] Antonio Fariña Martínez, Nieves Brisaboa, and Gonzalo Navarro. *New compression codes for text databases.* Phd thesis, Universidade da Coruña, 2005.

[FGN13] Arash Farzan, Travis Gagie, and Gonzalo Navarro. Entropy-bounded representation of point grids. *Computational Geometry: Theory and Applications*, 47(1):1–14, 2013.

[Fre08] E Frentzos. *Trajectory Data Management in Moving Object Databases.* Ph. d. thesis, University of Piraeus, Athens, 2008.

[GGV03] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-Order Entropy-Compressed Text Indexes. *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, 2068:841–850, 2003.

[GHSV06] Ankur Gupta, Wing Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Compressed data structures: Dictionaries and data-aware measures. *Data Compression Conference Proceedings*, 387:213–222, 2006.

[GN07] Gilberto Gutiérrez and Gonzalo Navarro. *Métodos de Acceso y Procesamiento de Consultas Espacio-Temporales.* Phd, Universidad de Chile, 2007.

[GNR+05] Gilberto Gutiérrez, Gonzalo Navarro, Andrea Rodríguez, Alejandro González, and José Orellana. A Spatio-Temporal Access Method based on Snapshots and Events. In *Proceedings of the 13th {ACM} International Symposium on Advances in Geographic Information Systems (GIS'05)*, pages 115–124. ACM Press, 2005.

[GS05] Ralf Hartmut Güting and Markus Schneider. *Moving Objects Databases*. Morgan Kufmann, 1st edition, 2005.

[GW05] Antony Galton and Michael Worboys. Processes and events in dynamic geo-networks. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3799 LNCS, pages 45–59, 2005.

[He13] M He. Succinct and implicit data structures for computational geometry. In *Conference on Space-Efficient Data Structures, Streams, and Algorithms*, volume 8066 LNCS, pages 216–235, 2013.

[HHT05] Marios Hadjieleftheriou, Erik Hoel, and Vassilis J. Tsotras. SaIL: A Spatial Index Library for Efficient Application Integration. *GeoInformatica*, 9(4):367–389, nov 2005.

[HSS11] Wing Kai Hon, Kunihiko Sadakane, and Wing Kin Sung. Succinct data structures for searchable partial sums with optimal worst-case performance. *Theoretical Computer Science*, 412(39):5176–5186, 2011.

[Jac89] G. Jacobson. Space-efficient static trees and graphs, 1989.

[JSS12] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Ultra-succinct representation of ordered trees with applications. *Journal of Computer and System Sciences*, 78(2):619–631, mar 2012.

[KBMS11] Markus Koegel, Daniel Baselt, Martin Mauve, and Bjorn Scheuermann. A comparison of vehicular trajectory encoding techniques. In *2011 The 10th IFIP Annual Mediterranean Ad Hoc Networking Workshop*, pages 87–94. IEEE, jun 2011.

[KKKM10] Markus Koegel, Wolfgang Kiess, Markus Kerper, and Martin Mauve. Compact Vehicular Trajectory Encoding (extended version). Technical report, Computer Science Department, Heinrich Heine University„ Düsseldorf, Germany, 2010.

[KM12] M Koegel and M Mauve. On the spatio-temporal information content and arithmetic coding of discrete trajectories. In *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering*, volume 104 LNICST, pages 13–24. Department of Computer Science, University of Düsseldorf, Germany, 2012.

[Koe13] Markus Koegel. *A Long Movement Story Cut Short — On the Compression of Trajectory Data*. Ph. d. thesis, Heinrich-Heine-Universität Düsseldorf, 2013.

[KRHM12] Markus Koegel, Matthias Radig, Erzen Hyko, and Martin Mauve. A Detailed View on the Spatio-Temporal Information Content and the Arithmetic Coding of Discrete Trajectories. *Mobile Networks and Applications*, (October):1–15, oct 2012.

[Lad11] Susana Ladra. *Algorithms and Compressed Data Structures for Information Retrieval*. Phd thesis, Universidade da Coruña, 2011.

[Lem12] Daniel Lemire. Effective compression using frame-of-reference and delta coding, 2012.

[MOH+13] Jonathan Muckell, Paul W. Olsen, Jeong-Hyon Hwang, Catherine T. Lawson, and S. S. Ravi. Compression of trajectory data: a comprehensive evaluation and new approach. *GeoInformatica*, 18(3):435–460, jul 2013.

[MRRR03] J.Ian Munro, Rajeev Raman, Venkatesh Raman, and SattiSrinivasa Rao. Succinct Representations of Permutations. In JosC.M. Baeten, JanKarel Lenstra, Joachim Parrow, and GerhardJ. Woeginger, editors, *Automata, Languages and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 345–356. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[MRRR12] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and Srinivasa S. Rao. Succinct representations of permutations and functions. *Theoretical Computer Science*, 438:74–88, 2012.

[MS80] J.Ian Munro and Hendra Suwanda. Implicit data structures for fast search and update. *Journal of Computer and System Sciences*, 21(2):236–250, 1980.

[Mun79] J. Ian Munro. A multikey search problem. In *17th Allerton Conference on Communication, control, and Computing*, pages 241–244, 1979.

[Mun96] J. Ian Munro. Tables. *Foundations of Software Technology and Theoretical Computer Science*, LNCS 1180:37–42, 1996.

[Nav14] Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, mar 2014.

[Nav16] Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. University of Cambridge, [2016], New York, NY, 2016.

[NR05] Jinfeng Ni and C Ravishankar. PA-tree: A parametric indexing scheme for spatio-temporal trajectories. *Advances in Spatial and Temporal Databases*, pages 254–272, 2005.

[NST98] Mario A Nascimento, Jefferson R O Silva, and Yannis Theodoridis. Access Structures for Moving Points. Technical Report TR–33, TIME CENTER, 1998.

[NST99] Mario A Nascimento, Jefferson R O Silva, and Yannis Theodoridis. Evaluation of Access Structures for Discretely Moving Points. In *Proceedings of the International Workshop on Spatio-Temporal Database Management (STDBM '99)*, pages 171–188, London, UK, 1999. Springer-Verlag.

[OS06] Daisuke Okanohara and Kunihiko Sadakane. Practical Entropy-Compressed Rank/Select Dictionary. pages 1–11, 2006.

[Pag99] Rasmus Pagh. Low Redundancy in Static Dictionaries with O(1) Worst Case Lookup Time. *Proceedings of ICALP*, LNCS 1644(20244):595–604, 1999.

[PTKT04] NIKOS PELEKIS, BABIS THEODOULIDIS, IOANNIS KOPANAKIS, and YANNIS THEODORIDIS. Literature review of spatio-temporal database models, 2004.

[RRR07] Rajeev Raman, Venkatesh Raman, and Srinivasa S. Rao. Succinct indexable dictionaries with applications to encoding k -ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):43–es, nov 2007.

[RVM06] Katerina Raptopoulou, Michael Vassilakopoulos, and Yannis Manolopoulos. On past-time indexing of moving objects. *Journal of Systems and Software*, 79(8):1079–1091, aug 2006.

[Sam06] Hanan Samet. *Foundations of Multidimensional And Metric Data Structures*. The Morgan Kaufmann series in computer graphics and geometric modeling. Elsevier/Morgan Kaufmann, 2006.

[Sec09] Diego Seco Naveiras. *Técnicas de indexación y recuperación de documentos utilizando referencias geográficas y textuales*. Phd, Universidade Da Coruña, 2009.

[SG06] Kunihiko Sadakane and Roberto Grossi. Squeezing succinct data structures into entropy bounds. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm - SODA '06*, pages 1230–1239, New York, New York, USA, 2006. ACM Press.

[Sha48] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27(3):379–423, 1948.

[SNL09] Diego Seco-Naveiras and Miguel Ángel Rodríguez Luaces. Técnicas De Indexación Y Recuperación De Documentos Utilizando Referencias Geográficas Y Textuales. *Departamento de Computación*, page 147, 2009.

[SRL09] Falko Schmid, KF Kai-Florian Richter, and Patrick Laube. Semantic Trajectory Compression. In *Advances in Spatial and Temporal Databases*, volume 5644, pages 411–416. Springer Berlin Heidelberg, 2009.

[SXYL16] Penghui Sun, Shixiong Xia, Guan Yuan, and Daxing Li. An Overview of Moving Object Trajectory Compression Algorithms. *Mathematical Problems in Engineering*, 2016:6587309, 2016.

[TP01a] Yufei Tao and Dimitris Papadias. Efficient Historical {R-tree}. In *{SSDBM} International Conference on Scientific and Statical Database Management*, pages 223–232, 2001.

[TP01b] Yufei Tao and Dimitris Papadias. {MV3R-tree}: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 431–440, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[TP01c] Yufei Tao Yufei Tao and D. Papadias. Efficient historical R-trees. *Proceedings Thirteenth International Conference on Scientific and Statistical Database Management. SSDBM 2001*, pages 223–232, 2001.

[TPZ02] Yufei Tao, Dimitris Papadias, and Jun Zhang. Cost models for overlapping and multiversion structures. *ACM Trans. Database Syst.*, 27(3):299–342, 2002.

[TVM01] Theodoros Tzouramanis, Michael Vassilakopoulos, and Yannis Manolopoulos. Multiversion Linear Quadtree for Spatio-Temporal Data. volume 0056, pages 279–292. 2001.

[TVS96] Yannis Theodoridis, Michalis Vazirgiannis, and Timos K Sellis. Spatio-Temporal Indexing for Large Multimedia Applications. In *Proceedings of the 1996 International Conference on Multimedia Computing and Systems (ICMCS '96)*, pages 441–448, Washington, DC, USA, 1996. IEEE Computer Society.

[Wor05] Michael Worboys. Event-oriented approaches to geographic phenomena. *International Journal of Geographical Information Science*, 19(1):1–28, 2005.

[XHL90] X Xu, J Han, and W Lu. {RT-tree}: An Improved {R-tree} Index Structure for Spatio-temporal Database. In *4th International Symposium on Spatial Data Handling*, pages 1040–1049, 1990.