

Un índice espacio-temporal compacto para consultas *time-slice* y *time-interval*

Nieves R. Brisaboa¹, Ramón Casares¹, Andrea Rodríguez², Miguel Romero³,
and Diego Seco²

¹ Laboratorio de Bases de datos, Universidade da Coruña
Campus de Elviña, 15071 A Coruña, España
`brisaboa@udc.es`, `rcasares@enxenio.es`

² Universidad de Concepción, Departamento de Ingeniería Informática y Ciencias de
la Computación, Edmundo Larenas 215, 4070409 Concepción, Chile
`{andrea|dseco}@udec.cl`

³ Universidad del Bío-Bío, Departamento de Ciencias de la Computación y
Tecnologías de la Información
Andrés Bello s/n, 3800708 Chillán, Chile
`miguel.romero@ubiobio.cl`

Resumen La indexación de datos espacio-temporales es fundamental para responder de manera eficiente consultas acerca de objetos móviles que se encuentran en una región durante un instante o intervalo temporal determinado. Esto es de interés, por ejemplo, para detectar embarcaciones que invaden zonas de navegación prohibidas. Este artículo presenta un índice espacio-temporal compacto (eficiente en espacio) que permite responder ambos tipos de consultas. La evaluación experimental muestra que el índice propuesto es competitivo en tiempo al compararlo con el MVR-Tree usando significativamente menos espacio.

Keywords: Objeto móvil, espacio-temporal, estructura de datos compacta, indexación

1. Introducción

El avance científico y tecnológico en el ámbito de las redes y de los dispositivos móviles ha facilitado la generación y recolección de datos espacio temporales. Un caso particular de estos datos son los generados por 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, aves migratorias, entre otros.

Para el tratamiento de estos datos y sus características particulares nacen las llamadas bases de datos espacio-temporales. Estos sistemas son capaces de responder a diferentes tipos de consultas, de las cuales las consultas de tipo instante temporal (*time-slice*) e intervalo temporal (*time-interval*) son las más estudiadas [4].

Una consulta de tipo instante temporal permite recuperar todos los objetos que se encuentran dentro de un rango espacial en un instante dado. Por ejemplo, en una base de datos de tráfico se podría responder la siguiente pregunta: «¿qué vehículos han pasado a menos de 100 metros de las coordenadas (x,y) el 20 de agosto de 2014?». Las consultas de tipo intervalo extienden el resultado de las anteriores considerando los instantes consecutivos dentro del intervalo temporal. En una base de datos de barcos pesqueros, se podría consultar «¿qué barcos invadieron una zona determinada durante la temporada de veda?».

El campo de investigación de las bases de datos espacio-temporales ha sido ampliamente estudiado, con trabajos relevantes tanto a nivel científico como en su aplicación industrial. En este sentido, el presente trabajo aporta una mirada nueva al campo de las bases de datos espacio-temporales presentando un índice para objetos móviles que, utilizando estructuras de datos compactas, es capaz de responder a las consultas espacio-temporales más habituales de forma eficiente⁴.

2. Estado del arte

Se ha investigado mucho en el campo de las bases de datos espacio-temporales en relación a métodos de acceso e indexación eficientes. Por ejemplo, el 3DR-Tree [17] es un método de acceso espacio-temporal que considera al tiempo como otro eje dentro de las coordenadas espaciales. En esta estructura, un segmento de línea representa a un objeto en una ubicación en un intervalo temporal. El 3DR-Tree es eficiente en el procesamiento de consultas de intervalo pero ineficiente para el procesamiento de consultas del tipo instante temporal.

El RT-Tree [18] es una estructura donde la información temporal es mantenida en los nodos del tradicional R-tree siendo, por tanto, una extensión de dicha estructura de datos ubicada en bases de datos espaciales. 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 [11].

El HR-Tree [12] utiliza un R-tree para cada instante de tiempo, pero los árboles son almacenados teniendo en cuenta la sobreposición entre ellos. La idea básica es que, dados dos árboles, el más reciente de ellos corresponde a una evolución del más antiguo y los sub-árboles que los componen pueden ser compartidos entre ambos árboles. La mayor ventaja del HR-Tree es su eficiente procesamiento de las consultas de tipo instante, mientras que su mayor desventaja es el excesivo espacio que requiere para almacenar la estructura.

El MVR-Tree [16] es una estructura basada en la manipulación de múltiples versiones. Éste es una extensión del MVB-Tree [2], donde el atributo que varía en el tiempo corresponde al espacial.

El SEST-Index [5] mantiene instantáneas del estado de los objetos (*snapshots*) para algunos instantes de tiempo junto con bitácoras de eventos entre dichas instantáneas. Los eventos aportan más semántica que un cambio de posición del

⁴ Una versión preliminar de este trabajo fue presentada en [13].

objeto. Por ejemplo, pueden indicar cuando un objeto ha entrado o salido de un determinado lugar o si un objeto ha colisionado con otro. Por tanto, esta estructura es adecuada para responder consultas respecto de los eventos.

Hasta donde nosotros conocemos, de las estructuras presentadas el MVR-Tree, y su variante mejorada MV3R-Tree, tienen un mejor desempeño que los demás métodos de acceso espacio-temporal previamente desarrollados en términos de coste temporal para las consultas de tipo instante e intervalo temporal.

A pesar del gran número de métodos de acceso espacio-temporal existentes, ninguno de ellos ha tenido en consideración el diseño de una estructura de datos que almacene tanto los datos como el índice de un modo compacto. Posiblemente esto se deba a que el objetivo de diseño principal ha sido mejorar los tiempos de respuesta a costa de pagar 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 se pueden categorizar como:

- Utilización de técnicas de compresión de datos como LZW, DEFLATE, LZMA y códigos aritméticos usando PPM [7].
- Simplificación de trayectorias mediante algoritmos de Douglas-Peucker, OPW, OPW-TR y Dead Reckoning [9].
- Compresión semántica de trayectorias [14] empleando la información espacial contenida en la infraestructura o red por donde se mueven los objetos y modelando las trayectorias como caminos en el grafo que modela la red.

Recientemente [7], se ha demostrado que es posible disminuir significativamente el coste de almacenamiento de los datos empleando técnicas de compresión basadas en códigos aritméticos (frente a utilizar técnicas de simplificación de trayectorias con un bajo nivel de error). Sin embargo, estas técnicas no permiten responder a las consultas espacio-temporales sin descomprimir los datos previamente. Esto provoca que no sea factible el utilizar dichas técnicas directamente en sistemas de indexación espacio-temporal, debido al sobre coste 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 castigan los tiempos de acceso, las denominadas estructuras de datos compactas. En muchos casos, estas estructuras logran altos ratios de compresión a la vez que permiten un acceso eficiente a los datos contenidos sin necesidad de descomprimirlos. 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 evitar el acceso a disco (que es órdenes de magnitud más lento).

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. Por ejemplo, el Compact Quadtree [3], el Compact Kd-tree[1] o el SpatialWT[15]. Sin embargo, no existen trabajos previos de estructuras de datos compactas en el campo de las bases de datos espacio-temporales.

3. Nuestra propuesta

Existen dos abstracciones claves en nuestro modelo. La primera es el *snapshot*, el cual almacenan la ubicación de todos los objetos en un instante particular y 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 *snapshots* y *bitácoras*, con la finalidad de responder a las consultas espacio-temporales soportadas.

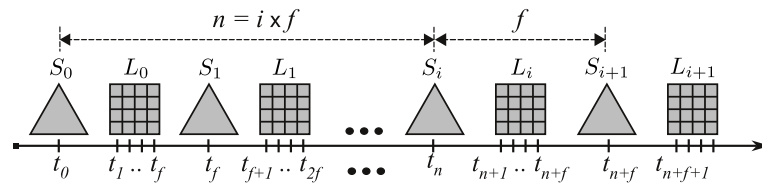


Figura 1. Componentes del índice propuesto.

En la figura 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 $[i \times f]$ instantes después de t_0 (t_n en la figura 1). La colección de *bitácoras* L_0 agrupa a todas las *bitácoras* de los objetos en el intervalo $[t_1 \dots t_f]$ 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 $t' \in [t_{n+1} \dots t_{n+f}]$, tanto avanzando desde s_i hasta t' , como retrocediendo desde s_{i+1} hasta t' .

Cada *bitácora* l_i almacena los movimientos de los objetos de manera compacta. La clave para lograr esta compactación 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 lograr 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 se registra en la *bitácora* como una posición relativa a la anterior.

Considere el siguiente ejemplo con un *snapshot* cada 8 instantes y en el que existen dos objetos o_1 y o_2 cuyas ubicaciones durante el intervalo $[0 \dots 8]$ son:

$$o_1 = \{(9, 6), (9, 7), (8, 6), (7, 6), (7, 5), (7, 4), (8, 3), (9, 3), (10, 3)\}$$

$$o_2 = \{(5, 6), (5, 5), (4, 5), (4, 4), (5, 4), (5, 5), (5, 6), (5, 7), (6, 7)\}$$

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* se crearía en el instante 8 y almacenaría

la posición de los objetos en ese instante (es decir, $S_1 = \{o_1(10, 3), o_2(6, 7)\}$). En relación a las colecciones de bitácoras, en este ejemplo solo existe una que es L_0 la cual abarca el intervalo $[1 \dots 8]$. 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:

$$L_0 = \{l_1, l_2\}$$

$$l_1 = \{(0, 1), (-1, 0), (-1, 0), (0, -1), (0, -1), (1, -1), (1, 0), (1, 0)\}$$

$$l_2 = \{(0, -1), (-1, 0), (0, -1), (1, 0), (0, 1), (0, 1), (0, 1), (1, 0)\}$$

Se han considerado dos restricciones en este modelo. La primera es que no pueden haber dos objetos en una misma celda y la segunda es que los objetos se mueven a celdas adyacentes (en una unidad de tiempo). A continuación se describen las estructuras de datos que implementan este modelo general.

3.1. Estructuras de datos

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

Cada *snapshot* usa un K^2 -tree [8] para representar de forma compacta las posiciones (x, y) de cada uno de los objetos indexados junto con una permutación que permite vincular el identificador (ID) del objeto con la celda correspondiente. De este modo, dada una celda, es posible identificar cuál es el objeto asociado y también, dado un objeto, es posible saber en qué celda se encuentra.

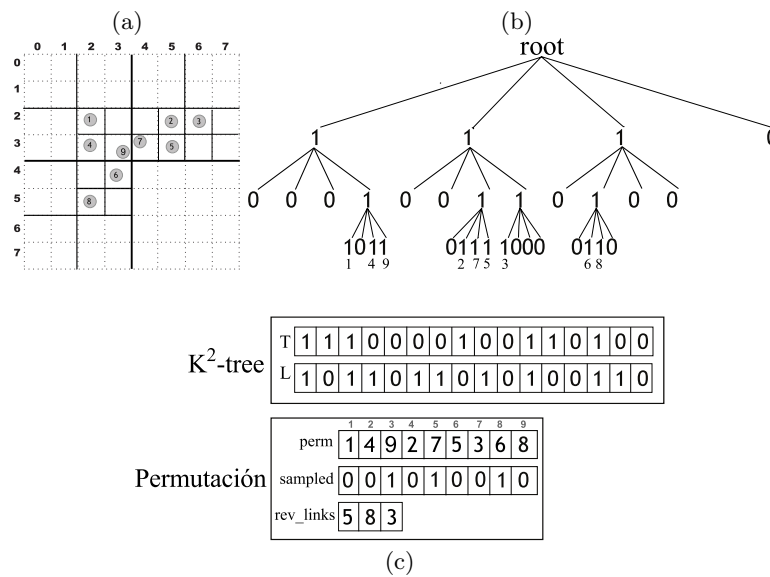


Figura 2. Un ejemplo de *snapshot* con $k = 2$: (a) espacio geográfico, (b) K^2 -tree conceptual, (c) *snapshot* almacenado con $t = 2$.

El K^2 -tree es una estructura de tipo árbol que representa de manera compacta una matriz de adyacencia binaria. En efecto, el espacio \mathbb{N}^2 (figura 2.a) 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 $\lceil \log_{k^2} n^2 \rceil$ (figura 2.b).

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 cual se particiona 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 o 0 en caso contrario. En el K^2 -tree los nodos en 0 no se siguen subdividiendo, lo que permite ahorrar espacio. En las hojas del 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.

El K^2 -tree es almacenado en dos *bitarrays* T y L (figura 2.c). 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 obtiene de recorrer en anchura las hojas del último nivel. Esta separación se debe a motivos de implementación, ya que el *bitarray* T requiere estructuras adicionales para soportar ciertas operaciones a nivel de bits, que no son necesarias en L .

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 . Esto 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.

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 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.

En el estado del arte existen varias estructuras de datos compactas para representar permutaciones. Se ha optado por la estructura presentada en [10] porque permite obtener π en tiempo constante, requisito indispensable para responder consultas por rango espacial de manera eficiente, y π^{-1} en tiempo $O(t)$ 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 un alto coste de almacenamiento ($2n \log(n) + o(n)$ 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 y se responde π^{-1} en tiempo $O(\log(n))$.

La estructura de datos para la permutación utiliza 3 *arrays*. El primero de ellos (*perm*) contiene la permutación entre el ID de la celda con el ID del objeto. Por ejemplo, la posición 5 del *array perm* contiene un 7 y esto significa que en la celda 5 del K^2 -tree se encuentra el objeto $ID = 7$. Así, saber cuál es el objeto que se encuentra en una celda es directo (es decir, toma tiempo $O(1)$).

Para responder π^{-1} se utiliza el concepto matemático de ciclo de una permutación, lo cual permite omitir la creación de un *array* que almacene directamente π^{-1} . En una permutación puede haber más de un ciclo, y cada ciclo se obtiene al ir visitando las celdas según su contenido a partir de una en particular. Por ejemplo, en la figura 2.c existen dos ciclos (2 4) y (3 9 8 6 5 7). El elemento 1 no configura un ciclo pues no está permutado. 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 vamos a revisar el contenido de la celda 9. En la celda 9 hay un 8, entonces vamos a la celda 8 a revisar su valor, y así sucesivamente se recorren las celdas 6, 5 y 7. Dado que la celda 7 contiene el número 3 la respuesta de $\pi^{-1}(3) = 7$.

Como los ciclos de una permutación en el peor caso podrían ser de largo n , se utilizan otros dos *arrays sampled* y *rev.links* los cuales permiten acortar el recorrido sobre el ciclo de la permutación de manera que se garantiza el comenzar el ciclo desde t elementos anteriores al elemento buscado. De este modo se garantiza que todos los ciclos se recorren en a lo más t iteraciones. La estrategia es bastante sencilla y consiste en marcar con un 1 en el *bitmap* llamado *sampled* aquellos ciclos que tengan un largo mayor a t y luego anotar la posición del *array perm* donde se debe comenzar el ciclo de la permutación para encontrar la respuesta en t accesos. Este puntero reverso se guarda en el *array* de enteros *rev.links*.

Por ejemplo, para el caso visto anteriormente $\pi^{-1}(3) = 7$, el recorrido para encontrar la respuesta fue (3 \rightarrow 9 \rightarrow 8 \rightarrow 6 \rightarrow 5 \rightarrow 7). Usando *sampled* y *rev.links* el recorrido comenzaría revisando si existe un puntero reverso para 3, es decir, si el tercer bit de *sampled* es 1. Como éste es el caso, se recupera el puntero reverso asociado que corresponde a 5, el cual está almacenado en el *array rev.links* en la primera posición (posición obtenida mediante la operación $rank_1(\text{sampled}, 3)$); entonces comenzamos el recorrido de la permutación desde la posición 5 (en vez de la 9) lo que acorta el ciclo a (3 \rightarrow 5 \rightarrow 7). Luego se sigue la permutación y el 5 nos lleva al 7, siendo ésta la respuesta.

Bitácoras. Como los objetos cuando se mueven lo hacen a las celdas adyacentes, 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 1) de modo que no es necesario codificar el tiempo de modo explícito.

Para la bitácora $\{(0, -1), (-1, 0), (0, -1), (1, 0), (0, 1), (0, 1), (0, 1), (1, 0)\}$, 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 3. Ejemplo de codificación de una bitácora utilizando cuatro *bitmaps*.

Al construir una bitácora puede ocurrir que un objeto no se haya movido o lo haga en una dirección de forma constante, dejando uno o más *bitmaps* vacíos. En estos casos es posible ahorrar espacio al marcar el *bitmap* como nulo.

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. Por ejemplo, para 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 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.

3.2. Algoritmos

Time Slice. Para responder a una consulta de *time slice* consideramos 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 recorriendo el K^2 -tree desde la raíz a las hojas visitando aquellos nodos que intersecten con el área de la consulta. Al llegar a las hojas se recuperan los ID contenidos en la permutación y éstos son la respuesta buscada.

El segundo caso ocurre cuando t_q se encuentra entre dos *snapshot* s_i y s_{i+1} . 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 general, 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 consulta. Aquellos objetos que queden dentro de la región de consulta r en t_q serán el resultado, descartando el resto de los objetos.

Al realizar la consulta por rango en un *snapshot* que ocurre en un instante distinto de t_q se debe usar una región de consulta lo suficientemente grande para contener a aquellos objetos que no están en r en el *snapshot* consultado, pero que sí lo están en t_q (r' en s_i y r'' en s_{i+1}). A mayor área de consulta, mayor es el tiempo que tomará la consulta por rango en el *snapshot*. Por esta razón se debe escoger el *snapshot* para el cual su área ampliada sea más pequeña.

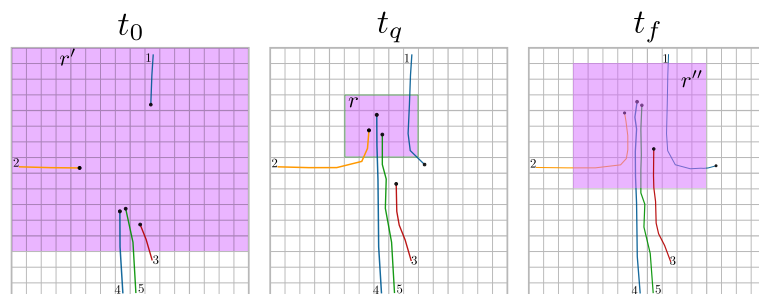


Figura 4. Ejemplo de ampliación de área r en r' y r''

En la figura 4 se pueden observar las áreas r' y r'' . Éstas se han creado dados los máximos desplazamientos, 6 y 2 respectivamente. 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 en el *snapshot* escogido tenemos una lista de candidatos a evaluar: 2, 3, 4, 5, junto con sus posiciones, y se excluyen aquellos objetos que, dado el máximo desplazamiento conocido, no es posible que se encuentren en la región de consulta en t_q (es decir, que sean parte del resultado).

Para cada uno de los objetos restantes, se obtiene su bitácora y se actualiza. En el caso de haber escogido el *snapshot* s_i se avanza el objeto sumando el resultado de 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 restando la sumatoria 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.

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

Para las consultas de *time interval*, el intervalo temporal de la consulta puede ser muy grande e intersectar con n *snapshots*. 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, \dots, Q_n) 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 *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 $[t_n, t_e]$.

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 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 este tipo de consulta sea más costoso que el anterior.

4. 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 para tener un punto de referencia ya que el MVR-Tree fue desarrollado con el objetivo de minimizar accesos a disco y no pensado como una estructura para memoria principal. Es decir, a pesar de que en nuestros experimentos ejecutamos el MVR-Tree en memoria principal, se debe tener en cuenta lo anterior en las conclusiones de los mismos.

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 corregir mediante interpolación aquellos movimientos que, por la naturaleza de los barcos, no eran posibles. 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 tamaños 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 éste afecta el tamaño de la estructura y los tiempos de respuesta.

Las consultas fueron tomadas de forma aleatoria de los mismos datos, de manera que se garantiza el que al menos exista un resultado en 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 [6], *a spatial index library*⁵. 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 tamaños de bitácora y se comparó con el MVR-Tree con diferentes capacidades de nodo para contener claves. En el cuadro 1 se muestran los diferentes tamaños expresados en MB para ambos índices.

Cuadro 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 tamaños de bitácoras para nuestra propuesta.

Capacidad	MVR-Tree		Índice Propuesto		
	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	1.024	779	2
90	5.093	67	2.048	855	2

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 el cuadro 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

⁵ El código se puede obtener desde <http://libspatialindex.github.com/>

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.

En el cuadro 2 se presentan los resultados de la ejecución de las consultas sobre los índices considerando diferentes valores para sus parámetros.

Cuadro 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 tamaños de bitácoras en nuestra propuesta.

Duración	Área	MVR-Tree				Índice Propuesto			
		15	30	60	90	256	512	1024	2048
0	10	40	61	54	59	27	28	31	45
	100	41	62	54	59	27	28	32	47
	1.000	46	69	60	64	34	38	46	68
256	10	118	149	178	249	60	54	60	81
	100	134	163	189	274	63	56	65	86
	1.000	139	164	193	283	118	121	135	165
512	10	195	236	300	438	100	110	142	184
	100	227	264	322	492	108	119	150	193
	1.000	232	257	324	504	227	294	361	429
1024	10	349	409	544	825	183	242	471	663
	100	410	464	588	929	198	262	481	679
	1.000	417	444	585	956	440	665	1.116	1.487

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 de la bitácora, el área de la consulta que se realiza en el K²-tree también se debe ampliar, siendo en el peor de los casos 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. Para el caso de las consultas de *time interval* se puede observar

que nuestra estructura es mejor si el intervalo temporal es pequeño (menor a 1.024), pero en el caso de intervalos largos el MVR-Tree es superior. Esto se debe a que 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.

5. Conclusiones

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, en la mayoría de los casos, mejora en tiempo y espacio al índice MVR-Tree ejecutado en memoria principal. Sin embargo, las dos restricciones que presenta el índice (no pueden haber dos objetos en una misma celda y los mismos solo se mueven a celdas adyacentes en una unidad de tiempo), hace que el rango de aplicabilidad se vea reducido a dominios donde se puedan garantizar dichas restricciones de manera natural, como por ejemplo los barcos que se mueven en el mar. Como trabajo futuro se está desarrollando un índice que sea capaz de indexar objetos móviles sin dichas restricciones.

Agradecimientos. Este trabajo está financiado en parte por CONICYT (Chile), programa Fondecyt ref. 1140428 (A.R.) y 11130377 (D.S.), por Ministerio de Economía y Competitividad (España) ref. TIN2013-46238-C4-3-R (N.B.) y ref. TIN2013-46801-C4-3-R (A.R. y D.S.), por Xunta de Galicia ref. RC2013/053 (N.B.), y por grupo Bases de Datos ref. 132019 GI/EF Universidad del Bío-Bío (GG, BT) (M.R.).

Referencias

1. Aly, M., Munich, M., Robotics, E., Perona, P.: CompactKdt: Compact Signatures for Accurate Large Scale Object Recognition. In: IEEE Workshop on Applications of Computer Vision (WACV). pp. 505–512. Colorado (2012)
2. Becker, B., Gschwind, S., Ohler, T., Seeger, B., Widmayer, P.: An asymptotically optimal multiversion {B-tree}. The VLDB Journal 5(4), 264–275 (1996)
3. Duffin, K.L.: Logical Operations in Compact Geospatial Quadtrees. In: Family History Tehnology workshop (2005)
4. Gutiérrez, G., Navarro, G.: Métodos de Acceso y Procesamiento de Consultas Espacio-Temporales. Phd, Universidad de Chile (2007)
5. Gutiérrez, G., Navarro, G., Rodríguez, A., González, A., Orellana, J.: 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). pp. 115–124. ACM Press (2005)
6. Hadjieleftheriou, M., Hoel, E., Tsotras, V.J.: SaIL: A Spatial Index Library for Efficient Application Integration. GeoInformatica 9(4), 367–389 (Nov 2005), <http://dl.acm.org/citation.cfm?id=1100986.1100992><http://www.springerlink.com/index/10.1007/s10707-005-4577-6>

7. Koegel, M.: A Long Movement Story Cut Short — On the Compression of Trajectory Data. Ph. d. thesis, Heinrich-Heine-Universität Düsseldorf (2013), <http://www.cn.uni-duesseldorf.de/publications/details/Koegel2013a.html>
8. Ladra, S.: Algorithms and Compressed Data Structures for Information Retrieval. Phd thesis, Universidade da Coruña (2011), http://lbd.udc.es/ShowThesisInformation.do?lang=es_ES&id=15
9. Muckell, J., Olsen, P.W., Hwang, J.H., Lawson, C.T., Ravi, S.S.: Compression of trajectory data: a comprehensive evaluation and new approach. *GeoInformatica* 18(3), 435–460 (Jul 2013), <http://link.springer.com/10.1007/s10707-013-0184-0>
10. Munro, J., Raman, R., Raman, V., Rao, S.: Succinct Representations of Permutations. In: Baeten, J., Lenstra, J., Parrow, J., Woeginger, G. (eds.) *Automata, Languages and Programming, Lecture Notes in Computer Science*, vol. 2719, pp. 345–356. Springer Berlin Heidelberg, Berlin, Heidelberg (2003), http://dx.doi.org/10.1007/3-540-45061-0_29http://link.springer.com/chapter/10.1007/3-540-45061-0_29
11. Nascimento, M.A., Silva, J.R.O., Theodoridis, Y.: Access Structures for Moving Points. *Tech. Rep. TR-33, TIME CENTER* (1998), citeseer.nj.nec.com/article/nascimento98access.html
12. Nascimento, M.A., Silva, J.R.O., Theodoridis, Y.: Evaluation of Access Structures for Discretely Moving Points. In: *Proceedings of the International Workshop on Spatio-Temporal Database Management (STDBM '99)*. pp. 171–188. Springer-Verlag, London, UK (1999)
13. Romero, M., Brisaboa, N., Rodríguez, M.A.: The SMO-index: a succinct moving object structure for timestamp and interval queries. In: *Proceedings of the 20th International Conference on Advances in Geographic Information Systems - SIGSPATIAL '12*. p. 498. ACM Press, New York, New York, USA (2012), <http://dl.acm.org/citation.cfm?doid=2424321.2424399>
14. Schmid, F., Richter, K.K.F., Laube, P.: Semantic Trajectory Compression. In: *Advances in Spatial and Temporal Databases*, vol. 5644, pp. 411–416. Springer Berlin Heidelberg (2009), http://link.springer.com/chapter/10.1007/978-3-642-02982-0_30
15. Seco Naveiras, D.: Técnicas de indexación y recuperación de documentos utilizando referencias geográficas y textuales. Phd, Universidade Da Coruña (2009)
16. Tao, Y., Papadias, D.: {MV3R-tree}: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In: *Proceedings of the 27th International Conference on Very Large Data Bases*. pp. 431–440. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)
17. Theodoridis, Y., Vazirgiannis, M., Sellis, T.K.: Spatio-Temporal Indexing for Large Multimedia Applications. In: *Proceedings of the 1996 International Conference on Multimedia Computing and Systems (ICMCS '96)*. pp. 441–448. IEEE Computer Society, Washington, DC, USA (1996)
18. Xu, X., Han, J., Lu, W.: {RT-tree}: An Improved {R-tree} Index Structure for Spatio-temporal Database. In: *4th International Symposium on Spatial Data Handling*. pp. 1040–1049 (1990)