UNIVERSIDAD DEL BÍO-BÍO

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

Consultas Espacio-Textuales sobre Estructuras de Datos Compactas

Por
Carlos Eduardo San Juan Contreras

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

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

Co-Dirigido por:
Dr. Miguel A. Martínez-Prieto
Universidad de Valladolid, Segovia, España

2018 - II

A mí abuelita Cocoy,
que siempre creyó en mí y lamentablemente
no alcancé a contarle este último logro.

Agradecimientos

A mi familia, padres y hermana, quienes siempre me han brindado su apoyo en todos los proyectos que se me ha ocurrido realizar durante la vida.

A Natanael Guerrero, Jefe del Departamento de Servicios Tecnológicos de la Universidad del Bío-Bío al cual pertenezco, por todas las facilidades laborales que me dió para ir a las clases y poder avanzar de la mejor forma en mí investigación y por supuesto, por siempre preguntarme cómo me iba en el desarrollo de la tesis.

A mis colegas de departamento, que siempre me ayudaron a bajar los niveles de stress con su cuota de humor diaria. A mis amigos de escalada y de Universidad, a los alumnos que les hice clases y a los distintos funcionarios que siempre me preguntaron amablemente como iba.

Al profesor Alfonso Rodríguez y a la profesora María Angélica Caro, que independiente de sus roles universitarios, siempre me orientaron y transmitieron sus experiencias como investigadores.

Al profesor Luis Gajardo, Director del Departamento de Ciencias de la Computación y Tecnologías de la Información, por confiar en mí y darme la oportunidad de realizar docencia, experiencia invaluable y que ha avivado mi pasión por la enseñanza.

A mi Director de Tesis, Dr. Gilberto Gutiérrez por ser antes que todo una excelente persona y ayudarme en todo momento, por sus constantes correcciones y sugerencias en el desarrollo de la investigación y por supuesto, por compartirme su basta sabiduría.

A mi Co-Director de Tesis, Dr. Miguel Martínez-Prieto por recibirme con los brazos abiertos durante mí pasantía en España, dedicando parte de su poco tiempo para guíarme y apoyarme en etapas fundamentales de la investigación.

Finalmente a quien estoy más agredecido, a mi pareja Catalina Espinoza, por su apoyo incondicional en todo este proceso y lograr animarme (y desconcentrarme) de alguna u otra forma siempre que podía, por su paciencia infinita para tolerar mis cambios de humor. Por simplemente ser ella y hacer que me enamore nuevamente cada día.

Resumen

Hoy en día, uno de los objetivo tecnológicos más importantes del mundo es capturar todo tipo de información y utilizarlos en beneficio de las personas. Por lo anterior, grandes cantidades de datos son recolectados en las diferentes áreas del mundo, ya sean generados por la naturaleza (biológico, astronómico, geográfico) o producidos por el hombre (imágenes, música, videos, libros). Sumado a lo anterior y como consecuencia del uso masivo de Internet, la popularización de los diferentes dispositivos móviles que pueden reportar su ubicación geográfica junto con las múltiples aplicaciones basadas en servicos Web que permiten etiquetar con palabras claves dichas posiciones, es que constantemente se están generando enormes cantidades de datos espacio-textuales con una tendencia de crecimiento exponencial. Todos estos datos son de utilidad si logran ser procesados eficientemente, por lo que la estructura de datos utilizada y el tamaño del conjunto de datos a evaluar juega un papel condicional en su tratamiento, principalmente cuando se habla de almacenamiento requerido y los tiempos de ejecución al realizar alguna consulta sobre ellos.

Las bases de datos espacio-textuales no se escapan de las condicionantes mencionadas, por lo que a lo largo de los años los investigadores se han enfocado en mejorar constantemente los tiempos de respuesta de las consultas espacio-textuales, proponiendo diferentes estructuras de datos que residen principalmente en memoria secundaria. Las consultas mencionadas se utilizan masivamente para proporcionar servicios de búsquedas innovadoras, como por ejemplo, recuperar "*el restaurante más cercano a una posición determinada y que ofrecezca algún tipo de servicio deseado*".

En el último periodo han aparecido las llamadas Estructuras de Datos Compactas que permiten representar los datos en forma comprimida directamente en memoria principal, logrando que su rendimiento no se vea afectado por los costos de Entrada/Salida y que el almacenamiento requerido sea considerablemente menor comparado con una estructura de datos tradicional, a la vez, posibilitan consultas de interés sin necesidad de descomprimir la información almacenada.

En esta tesis, se estudia el diseño e implementación de una nueva estructura de datos compacta basada en memoria principal, para representar grandes volúmenes de objetos espacio-textuales que clásicamente no cabrían en memoria RAM y que llamamos *c*B*i*K (Compact Bitmaps and Implicit KD-Tree). También se diseñan e implementan algoritmos para realizar los tres tipos de consultas más populares del área. Finalmente, mediante una serie de experimentos se evalúan las implementaciones mencionadas usando conjuntos de datos reales sobre *c*B*i*K y también sobre S2I, que según la literatura [12], es en promedio una de las mejores estructuras de datos a la fecha. Los resultados muestran una reducción del 38 % del almacenamiento de *c*B*i*K respecto a lo utilizado por S2I. En términos de tiempo de ejecución, *c*B*i*K supera a S2I en todas las consultas llegando

a diferenciarse hasta en tres órdenes de magnitud, demostrando las ventajas de la estructura de datos compacta propuesta.

De acuerdo a la revisón de la literatura, este trabajo de investigación propone la primera estructura de datos compacta para representar conjuntos de datos espacio-textuales y procesar eficientemente consultas sobre ella.

Índice

Parte I

Motivación

Capítulo 1

Introducción

1.1. Motivación

Las Bases de Datos Espaciales (BDE) realizan representaciones computacionales de la realidad geográfica utilizando datos que poseen una ubicación sobre el planeta (coordenadas) y que se ven reflejados como puntos, líneas o polígonos. Por medio del modelado de esos datos, se puede plasmar grandes espacios como montañas, ríos y países, así como también trayectorias de misiles, rutas aéreas o la red de transporte público de una ciudad, gracias a ello, es posible determinar qué es cierto objeto espacial o en qué parte del planeta se ubica. A través de diferentes estructuras de datos y algoritmos, se han enfrentado diversos problemas del área tales como el refinamiento de mallas [57], la búsqueda de los k vecinos más cercanos (kNN) [47], la separabilidad bricromática [56], el cálculo de la cerradura convexa (Convex Hull) [28], entre muchos otros.

Luego como una extensión de las BDE nacen las bases de datos espacio-textuales, las cuales almacenan una colección de objetos en donde cada uno de ellos está definido por su ubicación y una serie de palabras clave que lo describen. Los Puntos de Interés (POI) tales como hoteles, pubs, restaurantes, museos, cafés, parques, monumentos y atracciones turísticas, etc. pueden ser considerados objetos espacio-textuales, ya que cada uno de ellos consiste en un ítem georeferenciado cuya descripción puede comprender información espacial o textual. El POI de un hotel es un claro ejemplo de aquello, la información de la ubicación del hotel, los horarios de atención, los tipos de habitación que posee y los servicios como lavandería, wi-fi, desayuno, almuerzo, es la información espacio-textual que representa a ese hotel.

Otra área popular que utiliza información espacio-textual son las redes sociales. Generalmente los usuarios que suben fotos a Instagram o Facebook suelen agregar comentarios descriptivos a sus imágenes y las etiquetan con su posición geográfica. De similar forma, la utilización de hashtags en Twitter facilitan a los usuarios encontrar mensajes con un tema en específico.

Una consulta espacio-textual o *Spatial Keyword Query* (SKQ) se compone de una ubicación espacial y un conjunto de palabras clave requeridas. El resultado de la consulta se obtiene al recuperar los objetos que cumplen con los requisitos espacio-textuales especificados por la consulta.

En la literatura, se han presentado variados tipos de consultas y que responden a diferentes enfoques de resolución. Una de las más conocida es la consulta Top-k Spatial Keyword Query

(TkSKQ) y es posible dividirla en dos tipos: (i) La consulta Boolean Top-k Spatial Keyword Query (BkSKQ) recupera los k objetos más cercanos a una ubicación dada y que las palabras clave que definen a los objetos contengan las palabras clave especificadas en la consulta, y (ii) Ranked Top-k Spatial Keyword Query (RkSKQ), combina al mismo tiempo en una nueva métrica de similitud las posiciones y las palabras clave de los objetos, de tal forma, es posible determinar los k vecinos más cercanos con cierto grado de tolerancia a la inexistencia de algunas palabras clave requeridas. Otro tipo común de consulta SKQ, es la consulta por rango o Query Range Searching (bRS-SKQ), que entrega todos los objetos espacio-textuales que cumplan las palabras clave solicitadas y que además se encuentren dentro del rango espacial especificado.

En estos últimos años y como consecuencia de su gran utilidad, las aplicaciones que consumen consultas SKQ se han popularizado debido al uso exponencial que han experimentado los dispositivos móviles (smartphones, tablet, etc.), los que pueden reportar su ubicación geográfica, en conjunto con los servicios basados en la web que permiten etiquetar con palabras claves dichas posiciones (geotagging). Lo anterior, ha provocado que constantentemente se estén generando grandes cantidades de información espacio-textual y que requieren ser procesados eficientemente [32].

Existen varias propuestas de estructuras de datos (índices) que resuelven diferentes consultas espacio-textuales, sin embargo, todas ellas están diseñadas para representar índices espacio-textuales en memoria secundaria (principalmente disco). Si bien la tecnología avanza constantemente y mejora las capacidades de almacenamiento de los datos en memoria secundaria, realizar el procesamiento eficiente de esos datos en memoria principal (RAM) es cada vez más complejo debido al alto costo en operaciones de entrada/salida que se requiere, y ya que acceder a un dato en la RAM es aproximadamente 100.000 veces más rápido que en el disco, trabajar en memoria principal es crucial en el procesamiento de datos [45].

Recientemente, las estructuras de datos compactas (EDC) se han establecido como una alternativa para representar directamente en memoria principal, la mayor cantidad de datos en el menor espacio posible (logrando representar información que tradicionalmente no cabría en RAM) y a la vez, permiten procesarla directamente (sin descompactar) agregando un pequeño costo adicional para acceder a los elementos del conjunto [45]. Por lo anterior, las EDC evitan el problema del alto costo de operaciones de entrada/salida de las estructuras de datos tradicionales.

Debido a sus ventajas, las EDC han sido utilizadas en variados campos, por ejemplo, representación de imágenes [39], datos ráster (Mosaicos de datos) [3], relaciones en redes sociales [4], secuencias de ADN [6], árboles binarios de decisión [20], permutaciones de texto [36], grafos web [3, 17, 16], entre muchas otras aplicaciones.

Por supuesto, las ventajas de las EDC podrían extrapolarse a los conjuntos de datos espacio-textuales para lograr reducir su almacenamiento y tiempo de procesamiento total, en beneficio de los distintos tipos de dispositivos que tienen una memoria principal limitada (donde las técnicas actuales no funcionan) y que además, no tienen factibilidad de una conexión constante a una memoria secundaria (smartphones, smartwaches, routers, sensores o múltiples dispositivos embebidos utilizados por el Internet de las Cosas).

Por lo anterior, en esta tesis se describe el diseño e implementación de la primera EDC para representar objetos espacio-textuales en memoria principal. También, se implementan y evalúan algoritmos para las tres consultas espacio-textuales más populares existentes, respondiendo efi-

cientemente en términos de tiempo de ejecución y de almacenamiento requerido.

1.2. Hipótesis y Objetivos

Hipótesis

Es factible aprovechar las propiedades de las EDC para representar objetos espacio-textuales y permitir procesar consultas SKQ de manera más eficiente (referente al almacenamiento y tiempo) que aquellas técnicas que consideran que los datos se encuentran representados en estructuras de datos en memoria secundaria.

Objetivo General

El objetivo general es lograr representar conjuntos de objetos espacio-textuales sobre una EDC, aprovechando sus propiedades para permitir procesar eficientemente, y directamente en memoria principal consultas SKQ sobre ella.

Objetivos Específicos

Los objetivos específicos que se plantean alcanzar en esta tesis son:

1. Diseñar e implementar una EDC, utilizando bitmaps y un KD-Tree implícito para representar objetos espacio-textuales.
2. Diseñar e implementar algoritmos que permitan responder las siguientes consultas SKQ; los k vecinos más cercano booleano, los k vecinos más cercano con ponderación y la consulta por rango.
3. Evaluar el desempeño con datos de pruebas reales, tanto en rendimiento como también en almacenamiento, de los algoritmos implementados frente a técnicas similares propuestas en la literatura.

1.3. Alcance de la Investigación

Se contempla la implementación de los algoritmos bajo el lenguaje de programación C++ en conjunto con la biblioteca de estructuras de datos sucintas SDSL [26], resultante del trabajo de 40 publicaciones de investigación.

Es importante destacar que esta implementación será la primera en realizar consultas SKQ sobre una EDC que logre representar objetos espacio-textuales.

1.4. Metodología de Trabajo

La metodología de trabajo consta de las siguientes etapas secuenciales:

1. Revisión de la literatura.
2. Diseño de la EDC para manejar conjuntos de datos espacio-textuales.
3. Implementación de la EDC mencionada anteriormente.
4. Diseño de los algoritmos para las consultas espacio-textuales de *i*) los k vecinos más cercanos con coincidencia exacta, *ii*) los k vecinos más cercanos con ponderación de exactitud y *iii*) la búsqueda por rango sobre la EDC propuesta.
5. Implementación de los algoritmos anteriormente mencionados.
6. Realización del análisis experimental con conjuntos de datos reales.

1.5. Contribución

En el desarrollo del presente trabajo se realizan las siguientes contribuciones:

- Se propone una EDC para tratar conjuntos de objetos espacio-textuales.
- Se proponen algoritmos para obtener las tres consultas de interés más importantes: *i*) Boolean Top-k Spatial Keyword Query (BkSKQ), *ii*) Ranked Top-k Spatial Keyword Query (RkSKQ) y *iii*) Boolean Range Searching Spatial Keyword Query (bRS-SKQ).
- Se realizan pruebas de rendimiento (tiempo y almacenamiento) de las implementaciones mencionadas sobre la EDC usando conjuntos de datos reales.

De lo anterior se ha logrado la difusión de la investigación por diferentes medios que se detallan a continuación.

Presentación de artículos a conferencias, encuentros o workshop

- San Juan, C., Gutiérrez, G., Martínez-Prieto, M.: Estructuras de Datos Compactas para la Búsqueda por Palabras Claves en Conjuntos de Datos Espacio-Textuales. I Workshop de Tesistas del Magíster en Ciencias de la Computación. Universidad del Bío-Bío, Chillán - Chile (2017).
- San Juan, C., Gutiérrez, G., Martínez-Prieto, M.: A Compact Memory-based Index for Spatial Keyword Query Resolution. 37th International Conference of the Chilean Computer Science Society. University Andrés Bello, Santiago - Chile (2018) [55].
- San Juan, C., Gutiérrez, G., Martínez-Prieto, M.: cBiK - Un índice compacto para consultas espacio-textuales. VII Encuentro de Investigación de Estudiantes de Postgrado. Universidad del Bío-Bío, Concepción - Chile (2018).

Envío de artículos a revistas

- Así también se ha enviado un artículo a la revista GeoInformatica[1] de la editorial Springer, que contiene los principales resultados de investigación de esta tesis.

1.6. Organización de la Tesis

Este documento de tesis se organiza en tres partes principales. La primera parte dividida en 3 capítulos, considera el estado del arte de las consultas espacio-textuales, sus diferentes métodos de indexación y la descripción de las principales estructuras de datos compactas.

En consecuencia, se inicia con el Capítulo 2 describiendo los diferentes tipos de consultas espacio-textuales. Se detallan las tres consultas más importantes del área (k vecinos más cercanos booleano, con ponderación y la consulta por rango) y que se estudian en esta tesis. Además, se mencionan y explican brevemente otras consultas existentes en la literatura, como por ejemplo, consultas espacio-textales en movimiento [61] o por una red de caminos [51].

Luego en el Capítulo 3 se revisan los diferentes métodos de indexación espacio-textual que resuelven dichas consultas. Dado que las bases de datos espacio-textuales poseen dos componentes, este capítulo comienza haciendo mención a los tipos de indexación de las palabras claves, donde predominan los archivos invertidos y la utilización de bitmaps. Luego, se describen los tipos de indexación espacial y que están basados principalmente en los R-Tree y Quad-Tree.

Se da término al estado del arte con el Capítulo 4, describiendo brevemente las estructuras de datos compactas más utilizadas a la fecha. Se da especial énfasis a la estructura de Bitmaps ya que son la base para el diseño de las estructuras propuestas en este trabajo de investigación.

La segunda parte de esta tesis, se compone de tres capítulos y explican en detalle una nueva estructura de datos compacta, llamada cBiK, para procesar objetos espacio-textuales y lograr responder eficientemente consultas sobre ella.

En el Capítulo 5 se detalla la forma de representar los objetos espacio-textuales en cBiK, mediante la utilización de un KD-Tree implícito para la componente espacial y bitmaps compactos para la componente textual. Se describe la construcción, implementación y navegación de la estructura de datos compacta.

A continuación, en el Capítulo 6, se explican los algoritmos diseñados para resolver bajo cBiK, las tres consultas antes mencionadas.

El Capítulo 7 cierra el apartado con los resultados de un conjunto de experimentos, considerando diferentes escenarios y criterios para lograr evaluar el rendimiento de los algoritmos desarrollados en esta investigación frente a propuestas similares disponibles en la literatura.

Para terminar, la tercera parte de esta tesis considera el Capítulo 8 y se entregan algunas conclusiones generales del trabajo de investigación realizado, junto con algunas propuestas de mejora para ser consideradas en un trabajo futuro.

[1] `https://link.springer.com/journal/10707`

Parte II

Estado del Arte

Capítulo 2

Consultas Espacio-Textuales

Si se tiene un conjunto de datos espacio-textuales, por ejemplo, un conjunto de hoteles georeferenciados que poseen palablas claves que describen los servicios que ofrecen, es posible querer buscar el hotel más cercano a una posición en particular y que además, el hotel posea los servicios que se hayan seleccionado.

La anterior y otros tipos de consultas espacio-textuales han recibido mucha atención en el último tiempo, debido en gran parte a la variedad de aplicaciones informáticas desarrolladas en distintos sectores de la industria, junto con la masificación de los dispositivos móviles que cuentan con aplicaciones que utilizan dichas consultas constantemente.

En este capítulo se presentan los tres tipos de consultas espacio-textuales más estudiadas en la literatura y que se abordan en este trabajo de tesis. Además, se indican otros tipos de consultas propuestas por diferentes autores y que se aplican a situaciones específicas.

2.1. Información Preliminar

Para todas las consultas, se define D como el conjunto de objetos de una base de datos espacio-textual y T un conjunto de palabras claves, con $n = |D|$ la cantidad de objetos y $m = |T|$ el número total de palabras diferentes. Cada objeto $o \in D$ se define como un par $\langle l, t \rangle$, dónde $o.l$ describe la posición espacial (latitud y longitud) de un punto en 2 dimensiones y $o.t \subseteq T$ una lista de palabras claves que describen al punto.

Por ejemplo, si se tiene un conjunto de hoteles georeferenciados en un espacio bidimensional y las palabras claves que los describen corresponden a los servicios que ellos ofrecen, se tendría un escenario como el de la Figura 2.1, dónde $n = 7$ es el total de hoteles en D y $m = 10$ es la cantidad de palabras claves diferentes T. Este escenario se ocupará para ejemplificar las consultas principales del presente capítulo.

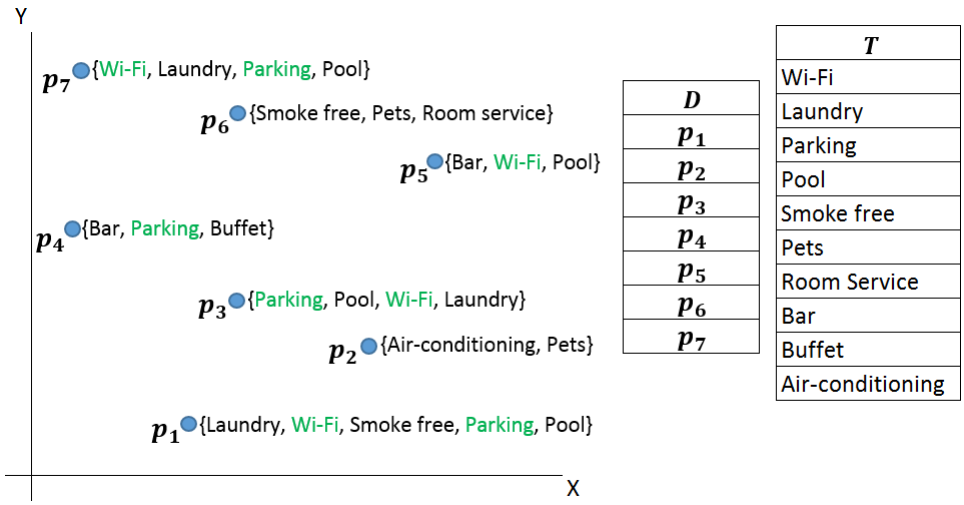


Figura 2.1: Espacio bidimensional con hoteles georefenciados y palabras claves que los describen.

2.2. Top-k Spatial Keyword Query

La consulta Top-k Spatial Keyword Query (TkSQK) devuelve un conjunto de objetos espacio-textuales referente a un punto q dado y a una secuencia de palabras claves. Dependiendo del tipo de consulta, la métrica de medición con respecto a q ordenará los objetos candidatos para entregar los mejores resultados, siendo $k = 1$ el mejor y $k = n$ el peor dentro de los objetos candidatos que cumplen las restricciones.

En el área de las bases de datos espacio-textuales, es posible mencionar dos tipos de consultas TkSQK.

- Boolean Top-k Spatial Keyword Query (BkSKQ)
- Ranked Top-k Spatial Keyword Query (RkSKQ)

A continuación se describen formalmente ambas consultas y se explica como trabaja cada una de ellas.

2.2.1. Boolean Top-k Spatial Keyword Query (BkSKQ)

Estudiada en [10, 19, 32, 60, 63], la consulta BkSKQ combina restricciones espaciales y textuales para obtener un resultado. Corresponde a una extensión del problema de los k vecinos más cercanos (kNN), donde los objetos más cercanos a un punto q dado, además satisfacen todas las palabras claves especificadas en la consulta.

Formalmente BkSKQ se define como una consulta $q =< l, t, k >$ dónde $q.l$ es la ubicación espacial del punto a consultar (latitud y longitud), $q.t$ es una lista de palabras claves y $1 \leq q.k \leq n$ es la cantidad de objetos resultantes. La respuesta de BkSKQ es un subconjunto de $q.k$ objetos de

D $\{o_1, o_2, ...o_{q.k}\}$ y que corresponde a los primeros $q.k$ objetos ordenados por la distancia euclídea (de menor a mayor) a $q.l$ y además $q.t \subseteq o_i.t$, con $1 \leq i \leq q.k$.

Un ejemplo de BkSKQ se puede ilustrar en la Figura 2.2, dónde se ha agregado un punto de consulta q con las palabras claves "Parking" y "Wi-Fi". Los objetos candidatos a solución son $C = \{p_1, p_3, p_7\}$ ya que ellos tienen la coincidencia exacta de las palabras claves solicitadas. Si se desean los $k = 2$ resultados, entonces se seleccionan por orden de cercanía al punto q y se tiene que $R = \{p_3, p_1\}$ dejando fuera al punto p_7.

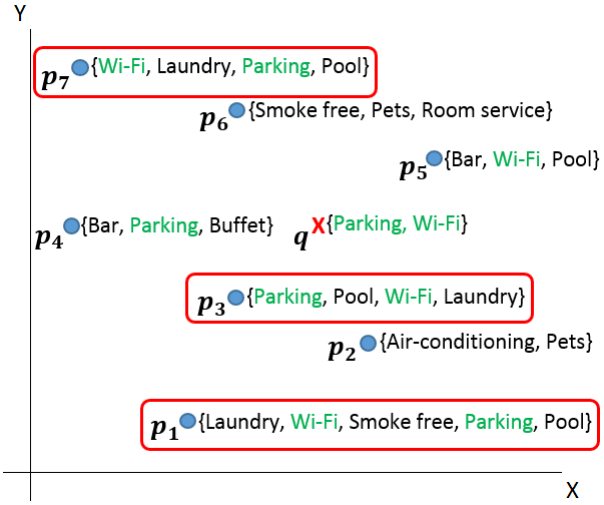


Figura 2.2: Ejemplo de BkSKQ con una consulta q con las palabras claves "Parking" y "Wi-Fi". Los puntos candidatos de solución son $C = \{p_1, p_3, p_7\}$

2.2.2. Ranked Top-k Spatial Keyword Query (RkSKQ)

En lugar de solo restringir los resultados en base a la coincidencia exacta de las palabras claves buscadas, en [18, 33, 38, 50, 52, 59, 65] se define la consulta *Ranked Top-k Spatial Keyword Query* (RkSKQ) que clasifica los objetos espacio-textuales mediante una función de puntuación que considera tanto la distancia a la ubicación de la consulta, como la relevancia textual de las palabras clave de la consulta. Específicamente, la función para obtener el puntaje del objeto espacio-textual o en base a la consulta q, se define en la ecuación 2.1.

$$\tau(o, q) = \alpha \cdot \delta(o.l, q.l) + (1 - \alpha) \cdot \theta(o.t, q.t) \tag{2.1}$$

Notar que $\delta(o.l, q.l)$ corresponde a la proximidad espacial entre o y q, $\theta(o.t, q.t)$ resuelve la relevancia textual entre o y q. Finalmente, el componente $\alpha \in [0, 1]$ es el parámetro de preferencia de la consulta, que pondera la proximidad espacial y la relevancia textual en el cálculo de su puntaje final. Si $\alpha = 1$, la función calculará los puntajes usando solo la distancia euclídea entre los puntos.

La proximidad espacial corresponde a la distancia euclídea normalizada y que está definida por la ecuación 2.2, dónde $dist(o.l, q.l)$ indica la distancia euclídea entre o y q y $dist_{max}$ es el

valor de la distancia máxima entre dos objetos en D.

$$\delta(o.l, q.l) = \frac{dist(o.l, q.l)}{dist_{max}} \tag{2.2}$$

Por otro lado, para obtener la relevancia textual es posible usar distintos modelos de recuperación de la información tales como “Language Model"[18], “Cosine Similiraty"[50] o “BM25"[14]. En esta investigación se utiliza un modelo simple que asigna a todas las palabras claves el mismo valor y que está definido por la ecuación 2.3. Notar que dependiendo de la cantidad de palabras claves que posea q, cada término tendrá el valor $\frac{1}{|q.t|}$. El resultado de la relevancia textual de o corresponderá a la sumatoria de los valores de las palabras claves coincidentes con q.

$$\theta(o.t, q.t) = \sum_{i=1}^{|q.t|} \frac{1}{|q.t_i|} \qquad , si \qquad q.t_i \in o.t \tag{2.3}$$

Por lo anterior, RkSKQ se ejemplifica en la Figura 2.3 descartando los objetos que no poseen al menos una palabra clave coincidente de las consultadas y dejando todos los demás como candidatos de solución. Si $k = 2$ resultados con un $\alpha = 0{,}5$ (la importancia espacial es igual a la textual, 50 % cada una) entonces se tiene $R = \{p_3, p_5\}$ debido a que p_3 es el objeto más cercano al punto q y tiene coincidencia total de las palabras claves consultadas y luego p_5, que sigue en cercanía y tiene la mitad de las coincidencias. Los demás objetos, tienen puntajes de clasificación inferiores.

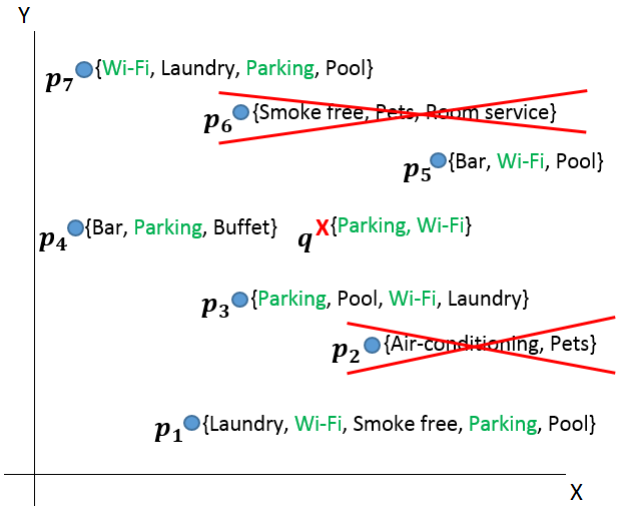


Figura 2.3: Ejemplo de RkSKQ con una consulta q con las palabras claves “Parking” y “Wi-Fi”.

2.3. Boolean Range Searching Spatial Keyword Query (bRS-SKQ)

La consulta bRS-SKQ ha sido estudiada en [13, 14, 31, 35, 58, 66] y especifican un $q =< r, t >$, dónde $q.r$ indica una región espacial de consulta que es determinada por dos puntos y $q.t$ una

lista de palabras claves buscadas. Entonces, bRS-SKQ recupera todos los objetos contenidos en la región espacial y que además posean la totalidad de las palabras claves consultadas. Formalmente, se decribe el resultado de q como un subconjunto de objetos o en D tal que $\forall\ o$ se cumple que $o.l \cap q.r \neq \emptyset \wedge q.t \subseteq o.t$. Notar que el resultado q no tiene un puntaje de clasificación de los objetos.

Para ejemplificar este tipo de consulta, se muestra en la Figura 2.4 una región de consulta y "Parking" y "Wi-Fi" como las palabras claves solicitadas. Los puntos que están fuera de la región son descartados inmediatamente y debido a que la coincidencia de las palabras claves debe ser exacta, el resultado es $R = \{p_3\}$, los demás objetos a pesar de estar contenidos en la región espacial (p_2, p_5 y p_6) no cumplen con la restricción textual.

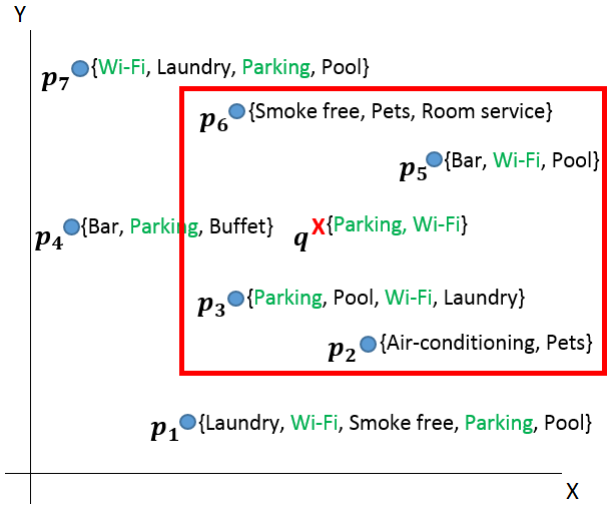


Figura 2.4: Ejemplo de consulta bRS-SKQ con una región q y con las palabras claves "Parking" y "Wi-Fi".

2.4. Otros tipos de consultas

Además de las anteriores tres consultas de interés, existen otras propuestas formalizadas por distintos investigadores a través de los años y que abarcan diferentes y variados enfoques. A continuación, se describe de forma breve su funcionamiento y el índice que utilizan:

M-closest keywords query: (2009) Zhang et al. en [64] proponen la consulta *m-closest keywords (mCK)* que utiliza un nuevo índice llamado bR*-Tree, este corresponde a una extensión del R*-Tree. La consulta recibe como argumento solo las palabras claves para recuperar los objetos espaciales con un mínimo diámetro entre ellos y que cada objeto está asociado con una sola palabra clave.

Colective spatial keyword query: (2011) Cao et al. en [9] utiliza el índice IR-Tree para recuperar un grupo de objetos espaciales de modo que las palabras clave del grupo, cubran las

palabras clave buscadas y que los objetos estén más cerca del punto de consulta a la vez que tengan las distancias más bajas entre ellos. Para ello propone dos consultas: (1) encontrar un grupo de objetos que cubra todas las palabras clave de consulta de manera que se minimice la suma de sus distancias al punto de consulta; y (2) encontrar un grupo de objetos que cubran todas las palabras clave de consulta, de modo que se minimice la suma de la distancia máxima desde un objeto en el grupo al punto de consulta y la distancia máxima entre dos objetos en el grupo.

Location-aware type ahead search (2011) Roy et. al. propone en [52] una búsqueda a medida que se escribe la palabra clave y que entrega como resultado, los k objetos espaciales cuyos nombres (o descripciones) sean coincidencias válidas de la cadena de consulta tipeada hasta el momento y que en términos de proximidad espacial, sus ubicaciones sean las más cercanas respecto a la posición geográfica del usuario.

Moving top-k spatial keyword query (MkSK) (2011) Wu et al. [61] y (2012) Huang et al. [33] utilizan un índice IR-tree [18], que corresponde a un R-tree [30] extendido con un archivo invertido [67] para ejecutar su consulta. MkSK toma como argumentos un punto espacial en movimiento y un conjunto de palabras clave. La consulta continuamente encuentra los k objetos que coinciden mejor con respecto a la proximidad espacial y la relevancia del texto.

Direction-aware spatial keyword query (DESKS) (2012) Li et al. [37] toma como argumentos un punto espacial, una dirección y un conjunto de palabras clave. La consulta encuentra los k vecinos más cercanos al punto espacial y a la dirección de consulta dada y que contengan todas las palabras clave requeridas.

Top-k Spatial Keyword Queries on Road Networks (2012) Rocha-Junior et al. [51] y (2014) Lin et al. [40] toman como argumento una ubicación espacial y un conjunto de palabras clave. La consulta devuelve los k mejores objetos clasificados en términos de su similitud textual con las palabras clave y la ruta más corta a la ubicación de la consulta, ésta última se apega más a las problemática de la vida real para llegar a los objetivos por una determinada ruta, ya que las propuestas que utilizan la distancia Euclídea como métrica espacial realizan un cálculo directo sin tomar en cuenta los obstáculos del camino.

Reverse Spatial-Keyword k-Nearest Neighbor (RSKkNN) query: (2014) Lu et al. propone en [41] un índice híbrido llamado IUR-Tree (Intersection-Union R-tree) que combina de forme eficiente la cercanía espacial junto con la similitud textual. La consulta toma un objeto con un punto espacial y un conjunto de palabras clave como argumentos y como resultado recupera los objetos que tienen el objeto de consulta como uno de sus k objetos más similares con respecto a la proximidad espacial y la relevancia del texto.

Estas consultas a pesar de ser notoriamente interesantes, no son parte de esta investigación debido a la envergadura que poseen y perfectamente pueden abarcar un nuevo trabajo de tesis. Por lo anterior, para nuestro estudio se han seleccionado las tres consultas principales del área las que ya fueron explicadas al inicio de este capítulo.

Capítulo 3

Indexación Espacio-Textual

Las consultas espacio-textuales se caracterizan por poseer dos partes, por un lado se tiene la componente espacial que organiza las coordenadas espaciales del punto y por otro, la componente textual que asocia las palabras claves a dicho punto. Por tal motivo, los índices utilizados en los diferentes trabajos de investigación existentes en la literatura, se pueden categorizar según las estructuras que utilizan para indexar cada tipo de información.

Es importante destacar que la mayoría de las propuestas de índices discutidas en este capítulo, se focalizan en procesar de manera eficiente las consultas, descuidando el almacenamiento que ellas requieren y que en algunos casos puede llegar a ser muy alto. Notar que todos ellos están representados por estructuras de datos que residen en memoria secundaria.

3.1. Indexación Textual

Para la componente textual, principalmente se ocupan dos tipos de estructuras para realizar la indexación de las palabras claves y que se describen a continuación.

3.1.1. Basados en Archivos Invertidos (Inverted File)

Algunos de los índices espacio-textuales como [14, 18, 25, 31, 35, 38, 50, 58, 66] ocupan el Inverted File para realizar la indexación textual.

Un Inverted File [67] consiste simplemente en un vocabulario con las palabras claves únicas, y cada palabra clave está asociada a una lista invertida. Esta a su vez, comprende una secuencia de objetos o de los cuales contiene normalmente un identificador y la frecuencia de la palabra clave. Un ejemplo se muestra en la Figura 3.1.

1	The old night keeper keeps the keep in the town
2	In the big old house in the big old gown.
3	The house in the town had the big old keep
4	Where the old night keeper never did sleep.
5	The night keeper keeps the keep in the night
6	And keeps in the dark and sleeps in the light.

(a) La base de datos de "Keeper", conformada por seis documentos de una sola línea.

term t	f_t	Inverted list for t
and	1	$\langle 6,2\rangle$
big	2	$\langle 2,2\rangle$ $\langle 3,1\rangle$
dark	1	$\langle 6,1\rangle$
did	1	$\langle 4,1\rangle$
gown	1	$\langle 2,1\rangle$
had	1	$\langle 3,1\rangle$
house	2	$\langle 2,1\rangle$ $\langle 3,1\rangle$
in	5	$\langle 1,1\rangle$ $\langle 2,2\rangle$ $\langle 3,1\rangle$ $\langle 5,1\rangle$ $\langle 6,2\rangle$
keep	3	$\langle 1,1\rangle$ $\langle 3,1\rangle$ $\langle 5,1\rangle$

(b) Parte del Inverted File para la base de datos de "Keeper".

Figura 3.1: Ejemplo del Inverted File dado por Zobel en [67]. Notar que en b), cada término t está compuesto por su frecuencia f_t y una lista de pares, siendo estos el Id del documento y la frecuencia del término en él.

En general, los objetos en cada Inverted File se ordenan por su Id de objeto. Sin embargo, algunos índices espacio-textuales los ordenan de manera diferente, como por los cuadrantes del quad-tree [35] o según la curva de relleno de espacio (Space filling curve) [14].

3.1.2. Basados en Bitmap

Por otro lado, algunos índices que usan R-tree [10, 19, 60] utilizan bitmaps [21] para lograr indexar la información de texto en los subárboles. Su funcionamiento es sencillo, cada bit en un bitmap representa la presencia o ausencia de una palabra clave en el objeto.

Ejemplo de aquello puede verificarse en el IR^2-Tree [19], que mejora cada nodo del R-tree con un bitmap para determinar la información de texto de los objetos en el subárbol del nodo. Por otro lado, algunos índices espacio-textuales como [60] usan bitmaps invertidos, en los que cada palabra clave corresponde a un bitmap y cada bit determina si un objeto contiene la palabra clave.

3.2. Indexación Espacial

Al igual que los índices textuales, la indexación espacial puede categorizarce en base a su estructura utilizada. A la fecha, se pueden mencionar dos grupos bien definidos, a saber, índices basados en un R-tree y los índices basados en Quad-tree. La explicación de cada índice se revisa a continuación.

3.2.1. Basados en R-tree

Esta categoría de índices usa el R-tree [30] o una variación (por ejemplo, el árbol R*-tree [1]). La mayoría de los índices espacio-textuales pertenecen a esta categoría y usan el archivo invertido para la indexación de texto. En los primeros trabajos [66], los índices basados en R-tree combinan libremente los R-tree y los archivos invertidos para organizar los datos espaciales y de texto por separado. En contraste, los índices recientes combinan fuertemente el R-tree con un índice de texto.

ID-IR*, R*-IF y IF-R* Zhou et al. en [66] propone tres índices, el Doble Índice Inverted File R*-Tree (ID-IR*), el Inverted File R*-tree (IF-R*) y el R*-tree Inverted File (R*-IF). Todos ellos abordan el problema de recuperar objetos espacio-textuales para una consulta de palabra clave dentro de una región espacial, es decir, la bRS-SKQ.

Primero el ID-IR*, indexa por separado dos veces los objetos espacio-textuales, una por un R*-tree y la segunda por Inverted List's. La estructura del ID-IR* se puede ver en la Figura 3.2. La consulta recupera las palabras claves consultadas de manera similar a un Inverted File convencional, mientras que la región de consulta se pasa por un R*-tree. Los resultados finales son la combinación de las listas de páginas de ambos índices.

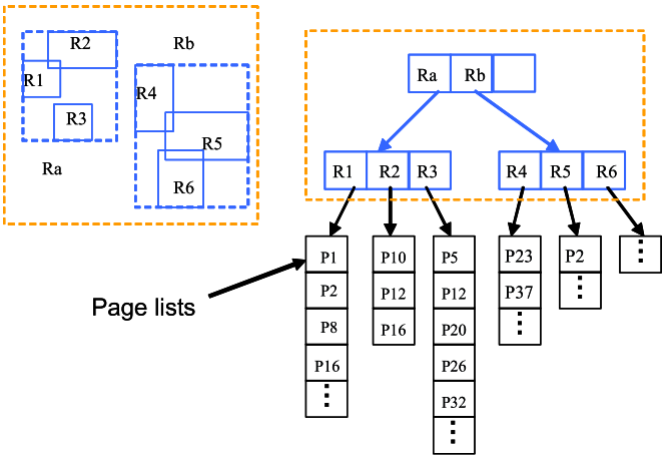
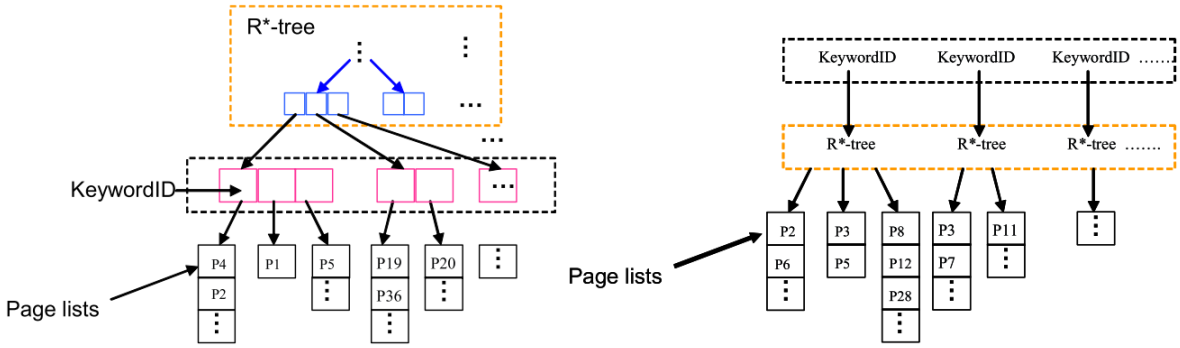


Figura 3.2: Esquema del ID-FR* dado por Zhou en [66].

Como segundo índice, el R*-IF primero filtra la componente espacial. Para ello crea un R*-tree para indexar todos los objetos en D sin considerar su componente textual y luego, para cada

nodo hoja del R*-tree se crea un Inverted File para indexar las palabras claves a los objetos contenidos en el nodo hoja (ver Figura 3.3-a).

Por último, el IF-R* es un índice que primero filtra la componente textual, al contrario del índice R*-IF. Para cada término distinto t en D, se construye un R*-tree para los objetos espacio-textuales en D (ver Figura 3.3-b).



(a) R*-IF (b) IF-R*

Figura 3.3: Esquema de las estructuras R*-IF y IF-R* dado por Zhou en [66]

Cuando se realiza una consulta, R*-IF primero recupera un conjunto de nodos hoja que están contenidos en la región consultada. Luego, los objetos que poseen las palabras clave especificadas se devuelven como resultado. Por otro lado, IF-R* revisa todos los R*-trees correspondientes a las palabras clave consultadas para formular su solución.

El trabajo experimental de [66] indica que de los tres índices expuestos, IF-R* es el mejor para responder consultas bRS-SKQ.

KR*-tree Hariharan et al. [31] propuso el KR*-tree (Keyword R*-tree) para procesar consultas bRS-SKQ. Cada nodo del KR*-tree está mejorado con el conjunto de palabras clave y están organizados en listas invertidas al igual que los objetos. Lo anterior ayuda a podar los nodos del árbol bajo los cuales los objetos no contienen las palabras clave consultadas.

La consulta basada en el KR*-tree primero encuentra el conjunto de nodos que contienen las palabras clave consultadas. Luego el conjunto resultante se transforma en el conjunto de candidatos para realizar el filtrado espacial y devolver un resultado.

IR^2-Tree De Felipe et al. [19] propuso uno de los primeros índices para evaluar consultas BkSKQ, denominado IR^2-Tree. Este integra un Signature File [21] en cada nodo del R-tree [30] (ver Figura 3.4).

El Signature File se representa mediante un bitmap que es almacenado en cada nodo del IR^2-tree, por lo tanto, la distribución del árbol dependerá de la longitud del Signature File. Cada Signature File del nodo se compone de la unión de todos los bitmaps de sus entradas, cada una de las cuales representa un nodo secundario y resume la presencia/ausencia de las palabras clave en los objetos hijos del nodo. Lo anterior facilita el proceso de búsqueda podando nodos que no

cumplen con el bitmap de la consulta.

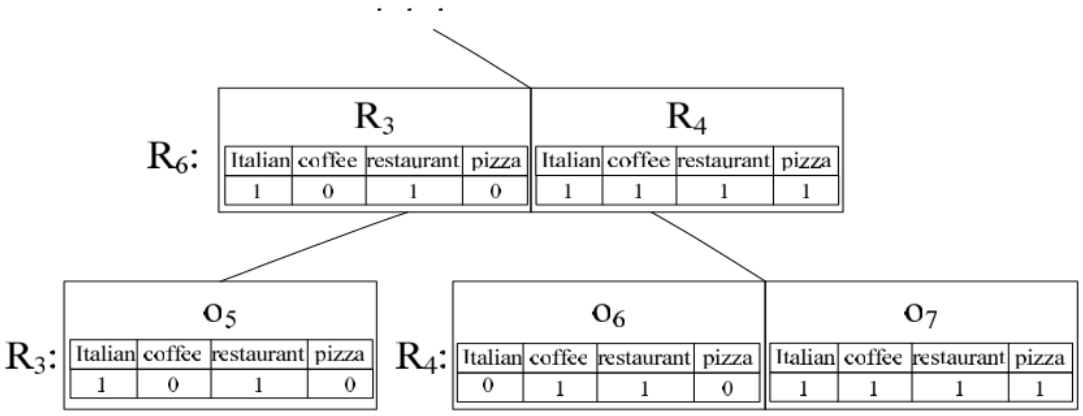


Figura 3.4: Esquema de un IR^2-Tree con bitmaps ilustrado por [12]

Hybrid Spatial-Keyword Index (SKI) Cary et al. [10] propuso SKI para resolver consultas BkSKQ usando el R-tree para la componente espacial y los bitmaps para almacenar la información textual.

Al nodo padre de un nodo hoja se le denomina supernodo. Cada nodo no hoja posee la información del rango de los Id de los supernodos bajo el nodo no hoja. Luego, cada supernodo tiene asociado un bitmap invertido. Específicamente, cada palabra clave tiene un bitmap cuyo bit corresponde a un objeto bajo el supernodo y se establece en 1 si el objeto contiene la palabra clave (ver Figura 3.5).

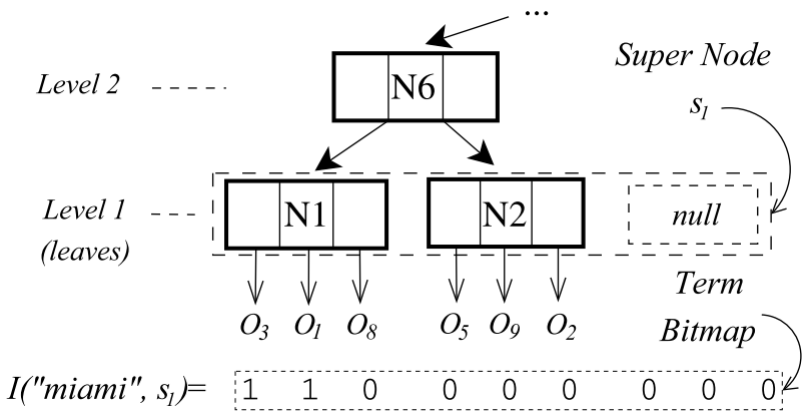


Figura 3.5: Supernodo S_1 compuesto de los nodos hoja [N_1, N_2] y el bitmap para la palabra clave "miami", ilustrado por Cary [10].

IR-tree El IR-tree [18, 59] mejora cada nodo del R-tree con un resumen de la información textual de los objetos en el subárbol correspondiente para procesar los tres tipos de consultas (bRS-SKQ, BkSKQ y RkSKQ).

La estructura del IR-tree se presenta en la Figura 3.6. Cada nodo almacena un puntero a un Inverted File que describe los objetos de los subárboles hijos de un nodo y dicho Inverted File contiene en un nodo en particular lo siguiente:

1. Un vocabulario con todas las palabras claves distintas de los objetos de los subárboles hijos del nodo.

2. Un conjunto de listas de publicación con la información de puntuación de la relevancia textual y cada una de ellas se relaciona con la palabra clave t.

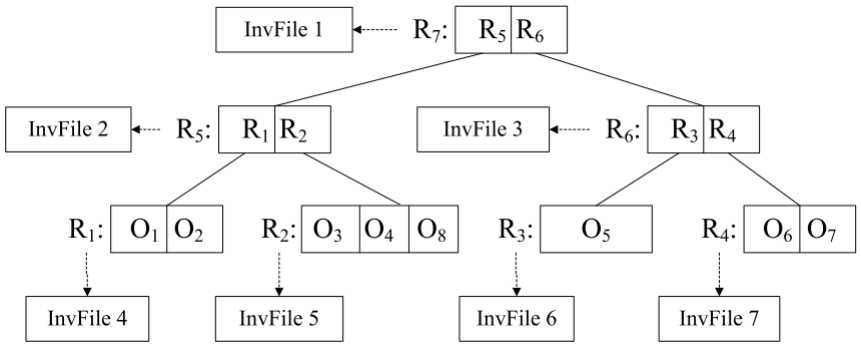


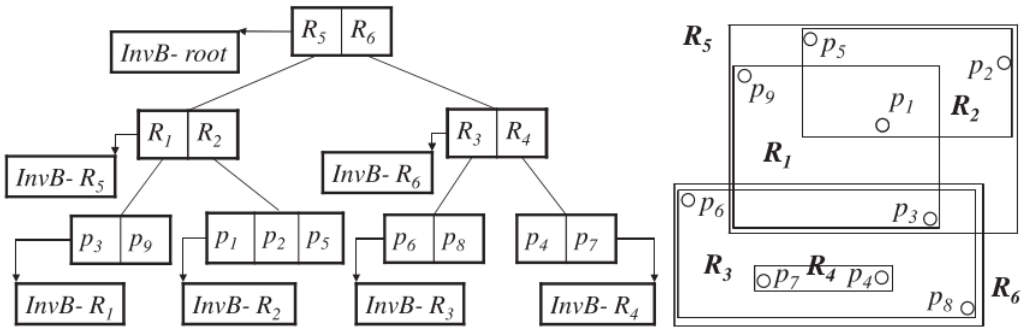
Figura 3.6: Estructura del IR-tree, ilustrado por Cong en [18].

De manera similar, Li et al. [38] presenta una estructura de índice que también la denomina IR-tree pero se diferencia de [18, 59], ya que ellos almacenan los archivos invertidos de cada nodo por separado, mientras que [38] almacena un archivo invertido integrado para todos los nodos, por lo tanto, la lista de publicación para cada palabra clave corresponde a la concatenación de las listas de publicación de todos los nodos del IR-tree.

WIR-Tree Wu et al. [60] propone el índice WIR-tree para realizar consultas BkSKQ. Su estructura utiliza una variante del IR-tree (Inverted File y R-tree) y se construye en base a la agrupación de las palabras claves de los objetos afines.

Su objetivo es lograr que cada grupo comparta la menor cantidad de palabras clave posible. Para ello, los objetos en D se dividen primero en dos grupos usando la palabra más frecuente w_1. Se genera un grupo cuyos objetos contienen a w_1 y el otro grupo cuyos objetos no lo tienen. Luego se divide cada uno de estos dos grupos por la siguiente palabra más frecuente w_2. El proceso se repite hasta que cada partición contenga una cierta cantidad de objetos y cada grupo de objetos se convierten en el nodo hoja del WIR-tree (ver Figura 3.7).

El WIR-tree [60] puede ser modificado para usar el bitmap invertido en reemplazo del archivo invertido y que Wu denomina como WIBR-tree, donde una posición de bitmap corresponde a la posición relativa de una entrada en el nodo del WIR-tree. En consecuencia, se logra reducir el espacio de almacenamiento del WIR-tree y también reducir operaciones de E/S durante el procesamiento de consultas.



(a) Estructura del árbol (b) Localización de los objetos

Figura 3.7: Esquema de las estructuras WIR-Tree dado por Wu en [60].

Spatial Inverted Index (S2I) Rocha-Junior et al. [51] propone su índice S2I para la resolución de consultas RkSKQ utilizando un R-tree y un Inverted File. S2I utiliza dos estrategias diferentes para realizar el indexado de objetos con palabras claves frecuentes e infrecuentes. Para ello, S2I mapea cada palabra clave t a un R-tree agregado (aR-tree) o a un bloque que almacena los objetos espacio-textuales o que contienen t. Las palabras claves más frecuentes se almacenan en el aR-tree, un árbol por palabra clave. Las menos frecuentes se almacenan en bloques en un archivo, un bloque por término de forma similar a los Inverted File. En el aR-Tree, cada nodo almacena un valor agregado que indica el impacto máximo (en términos de la puntuación de la relevancia textual) de la palabra clave en los objetos del árbol. El aR-Tree se puede considerar como el IR-tree [18, 59] para una sola palabra clave.

Para consultas con una sola palabra clave, solo se debe acceder a un árbol pequeño (en general) o a un bloque. Para consultas con varias palabras clave, se accede a algunos nodos de un conjunto pequeño de árboles o bloques conllevando a una ejecución eficiente de las consultas.

MFUKCache Gopinath et al. [27] presenta la estructura MFUKCache, que utiliza un esquema de almacenamiento similar a un caché para lograr reducir los accesos a disco y acelerar el proceso de revisión y poda en las consultas RkSKQ. Para ello utiliza un IR-tree que asocia índices invertidos en los nodos hoja de un R-tree y para cada nodo no hoja, se agrega un documento resumen que proporciona información de las palabras claves de los objetos en los subárboles dependientes.

En general, cuando se realiza una consulta SKQ tradicional se debe conocer la información de las palabras claves de un objeto para estimar la relevancia de cada nodo del IR-tree. Los bloques de valor t_f (frecuencia de la palabra clave en el objeto) y d_f (la frecuencia de los objetos que tienen la palabra clave) de las palabras clave presentes en la consulta se cargan desde el disco a la memoria. Luego se usan como resumen del documento para verificar la relevancia de cada nodo del IR-tree.

Por otro lado, MFUKCache posee un esquema de almacenamiento para mantener la información en memoria y acceder fácilmente a ella, ahorrándose todas las operaciones innecesarias de E/S. La estructura almacena una lista de palabras y su información t_f y d_f para cada nodo del IR-tree. Con ello, la información de las palabras claves que aparecen con frecuencia en las

reiteradas consultas es precargada en MFUKCache.

3.2.2. Basados en Quad-Tree

Esta categoría combina un índice quad-tree con un índice textual (por ejemplo, el Inverted File). Los índices quad-tree dividen el espacio en un número predefinido de celdas cuadradas o rectangulares de igual tamaño. El quad-tree y el índice textual pueden organizarse por separado [58] o combinados fuertemente [35].

ST y TS Vail et al. [58] propuso dos esquemas de indexación espacio-textual para resolver la consulta bRS-SKQ, los primeros basados en un quad-tree. Por un lado, el Índice Primario Espacial (ST) que realiza un primer filtrado de la componente espacial y luego con los candidatos entregados realiza una segunda revisión para restringir la componenete textual. El segundo índice, denominado Índice Primario Textual (TS) realiza una estrategia similar pero primero realiza el filtrado textual y luego el espacial. La estructura de TS se ilustra en la Figura 3.8, y corresponde a un índice invertido modificado del cual cada lista de publicación está asociada a una celda del quad-tree según su ubicación.

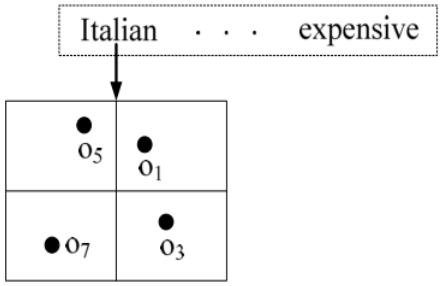


Figura 3.8: Estructura del quad-tree bajo la palabra "Italian", ilustrado por Chen [12].

Spatial-Keyword Inverted File (SKIF) Khodaei et al. propone la utilización de SKIF [35], una estructura similar a un Inverted File que almacena información espacial y textual de los objetos, de modo que ambas componentes se puedan manejar simultáneamente para responder consultas similares a las bRS-SKQ.

Para su componente textual, SKIF utiliza un Inverted File donde cada palabra clave está asociado a una lista invertida. De forma similar, la componente espacial utiliza un quad-tree y cada una de sus celdas distintas están representada por un Inverted File que comprende una lista de contabilizaciones, de las cuales consiste en un ID de objeto y su valor *idf* (frecuencia inversa del objeto) espacial, donde el objeto se solapa con la celda del quad-tree y el *idf* se mide por el grado de superposición. Importante es destacar que SKIF asume la ubicación de cada objeto como una región en lugar de un solo punto.

SFC-QUAD Christoforaki et al. [14] propuso el índice híbrido SFC-QUAD que combina el Space Filling Curve [5, 23, 53] y el Inverted File para resolver la consulta bRS-SKQ.

El índice SFC-QUAD es esencialmente un Inverted File en el que los ID y las frecuencias de los objetos se comprimen utilizando el algoritmo OPT-PFD [62]. Los ID en cada Inverted File se asignan y ordenan en base a sus posiciones espaciales en la curva Z [53]. SFC-QUAD también construye en memoria un Quad-Tree superficial para que el grado de la curva Z pueda ser calculado fácilmente atravesando el árbol del Quad-Tree en profundidad.

Para procesar una consulta bRS-SKQ, se adquiere un conjunto de rangos de ID pertenecientes a los objetos contenidos en la región de consulta al atravesar el Quad-Tree. Luego, para cada palabra clave t cuya lista invertida no se encuentra en la memoria principal, se determina un $k = 1$, 2 ó 3 rangos de ID con el objetivo que cubran los rangos de los ID anteriores y minimizar el costo total de E/S al disco. Posteriormente, para cada una de las palabras claves, se realizan k búsquedas en el disco para recuperar las partes necesarias de la lista invertida. Finalmente, los objetos que contienen todas las palabras clave consultadas son obtenidas.

Integrated Inverted Index (I^3) Zhang et al. propone el índice I^3 [65] para responder las consultas BkSKQ y RkSKQ. Su estructura utiliza el Quadtree [22, 54] para descomponer la información espacial en una jerarquía de celdas. El índice almacena la celda de palabras clave como la unidad básica y es denotada por $<w_i, C_j>$, ella corresponde a una lista de objetos que contienen la palabra clave w_i y que tienen sus ubicaciones asociadas en la celda C_j. Además, I^3 almacena información resumen de la palabra clave de celda para una poda efectiva. La información del resumen incluye un archivo de firma [21] que agrega en la celda de palabra clave, el ID del objeto y su puntuación de relevancia.

El índice posee tres componentes principales (ver Figura 3.9). Una tabla de búsqueda que sirve como portal, un archivo principal que contiene información resumida de celdas de palabras clave densas y un archivo de datos que almacena las tuplas de celdas de palabras clave en toda la lista invertida.

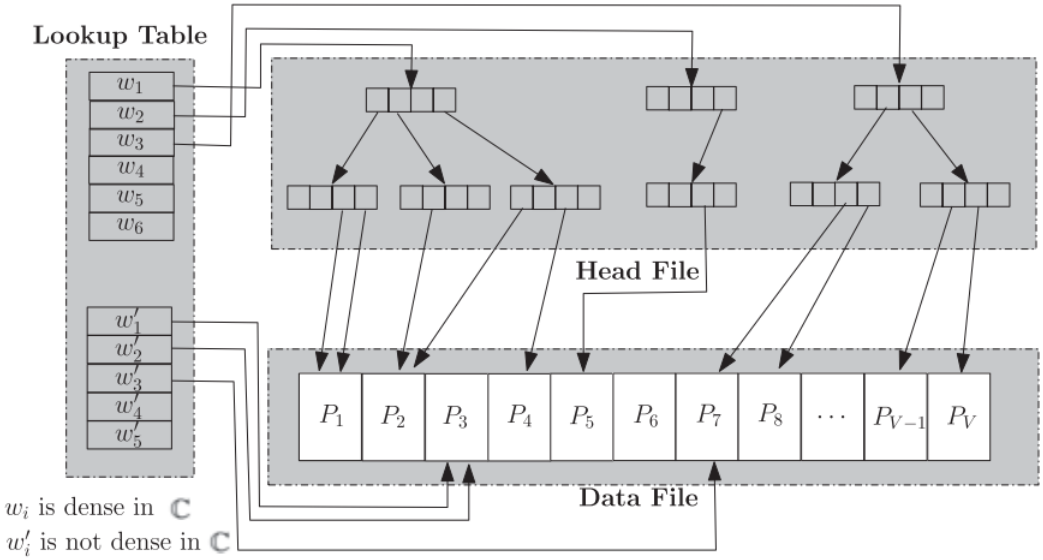


Figura 3.9: Estructura del índice I^3, ilustrado por Zhang [65].

Inverted Linear Quadtree (IL-Quadtree) Zhang et al. propone el índice IL-Quadtree [63] para procesar la consulta BkSKQ mediante el uso del Quadtree lineal [24].

La idea general es que para cada palabra clave $t_i \in V$ se construye un quadtree lineal LQ_i para los objetos que contienen la palabra clave t_i. Dadas dos palabras clave t_1 y t_2, y un conjunto de puntos, la Figura 3.10 ilustra los quadtrees lineales LQ_1 y LQ_2 construido para las palabras clave t_1 y t_2 respectivamente. Notar que los nodos hoja están etiquetados por sus códigos Morton [42], es decir, cada cuadrante está definido por a lo más dos bits. Los nodos de hoja negra (los cuadrantes que tienen un objeto) se denotan por 11, 10 para los nodos sin hoja, y 0 en caso contrario. Obviamente, un nodo en LQ_i está vacío (no contiene ningún objeto con la palabra clave t_i) si el bit se establece en 0.

La estrategia de consulta es la misma que la del R-tree y el R^2-tree. Los objetos que no aparecen en los LQ_i se eliminan inmediatamente logrando una poda eficiente.

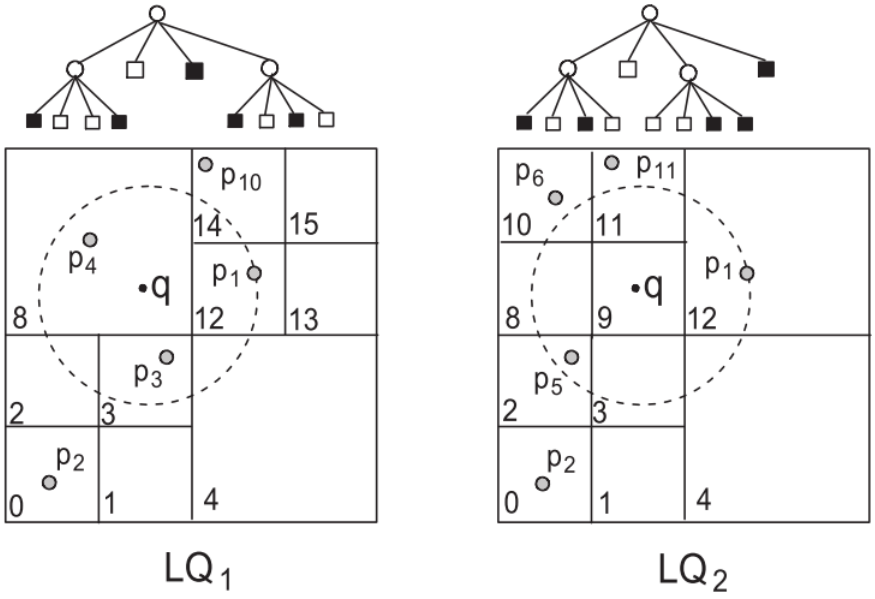


Figura 3.10: Ejemplo de IL-Quadtree, ilustrado por Zhang [63].

Posteriormente, Hong et al. [32] plantea una mejora al IL-Quadtree agregando una lista de palabras claves a cada objeto en los Quadtrees, logrando que el algoritmo solo necesite revisar un único quad-tree para responder la consulta BkSKQ.

Capítulo 4

Estructuras de Datos Compactas

Hoy en día las distintas aplicaciones como Facebook, Twitter, Instagram, Google Maps, entre muchas otras de uso masivo por parte de las personas, están generando constantemente grandes volúmenes de datos que son utilizados posteriormente para realizar múltiples tipos de consultas de interés. Dicha información es tradicionalmente guardada en memoria secundaria y parece no existir un problema de espacio gracias a la constante mejora de los dispositivos de almacenamiento que cada año incrementan sus capacidades. Lamentablemente, los accesos desde estos dispositivos a memoria principal son costosos y conseguir minimizar estos accesos se traduce en mejores tiempos de respuesta para los usuarios.

Por lo anterior, durante los últimos años se han presentados estructuras de datos que preservan esa gran cantidad de información pero de una manera compacta, logrando reducir el espacio utilizado por una representación común. A diferencia de la compresión de datos, las estructuras compactas (EDC) reducen su espacio de almacenamiento y a la vez mantienen las propiedades de accesibilidad a sus datos, de tal forma, se intenta aprovechar al máximo la jerarquía de memoria.

4.1. Estructuras de Datos Compactas

Antes de comentar los tipos de EDC, se hace necesario explicar en que consisten y como trabajan los llamados mapas de bits o bitmaps, ya que es uno de los elementos fundamentales para construirlas. Los bitmaps serán los encargados de definir el cómo se almacenará la información y posteriormente, dónde se realizarán los diferentes tipos de consultas.

4.1.1. Bitmap

Propuesto por Jacobson [34], el bitmap corresponde a un arreglo unidimensional de bits, o secuencia de bits, que posee estructuras adicionales para lograr ejecutar eficientemente consultas de Rank, Select y Access, las que se explican a continuación:

$\mathbf{Rank}_1(B, i)$: Determina el total de bits encendidos (con valor 1) desde el inicio hasta una i-ésima posición dada.

$\mathbf{Select}(B, i)$: Indica cuál es la posición del i-ésimo bit encendido.

Access(B, i)**:** Devuelve el valor del i-ésimo bit.

En la Figura 4.1 se puede ver un ejemplo de un bitmap y de sus operaciones rank y select.

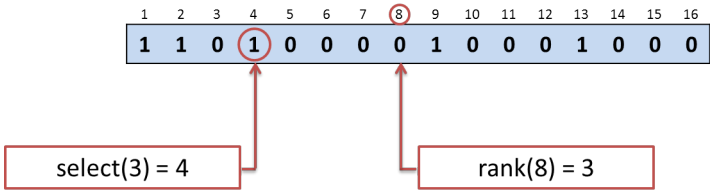


Figura 4.1: Ejemplo de bitmap y sus operaciones, ilustrado por Quijada [48].

Para lograr responder estas consultas de *rank* y *select*, la estructura segmenta el arreglo de bits en trozos de tamaño constante, y por cada trozo se almacena la suma de bits en 1 que contiene hasta su última casilla. Por ejemplo, si para el bitmap de la Figura 4.1 se divide en segmentos de 4 elementos, se requerirá de un arreglo adicional que almacene 4 valores, los cuales serían 3, 3, 4 y 5. De tal manera se responde la operación *rank* de manera constante, donde se revisarán 4 casillas del bitmap en el peor caso.

El bitmap como estructura de datos tiene variadas implementaciones, entre las que destacan la de Munro [43]; la de Raman, Raman y Rao [49]; y la de Okanohara y Sadakane [46] (ver Tabla 4.1), donde cada uno presenta diferentes estructuras auxiliares para dar solución a las operaciones *rank* y *select* eficientemente. Estas estructuras auxiliares son tratadas a través del problema de indexación de diccionarios.

Método	**Tamaño (bits)**	**rank**	**select**
Munro [43]	$n + O(\frac{n \log\log n}{\log n})$	$O(1)$	$O(1)$
Raman et al. [49]	$nH_0(B) + O(\frac{n \log\log n}{\log n})$	$O(1)$	$O(1)$
Okanohara y Sadakane [46]	$m \log \frac{n}{m} + 1{,}92m + o(m)$	$O(\log \frac{n}{m} + \frac{\log^4 m}{\log n})$	$O(\frac{\log^4 m}{\log n})$

Tabla 4.1: Representaciones de bitmaps [15]. Siendo n la cantidad total de elementos en un bitmap y m la cantidad de 1's presentes en el.

4.1.2. Wavelet Tree

El wavelet tree es una EDC propuesta por Grossi, Gupta y Vitter [29] de tipo árbol binario en que cada nodo es un bitmap.

Dada una cadena de texto S del alfabeto Σ. El wavelet tree representará la secuencia S en su nodo raíz a través de un bitmap de longitud n (tamaño del alfabeto) en donde cada s_i es representado por un 1 o por un 0 dependiendo de su posición en el alfabeto (lexicográficamente ordenada), es decir, si s_i está dentro de la primera mitad $\sigma/2$ del alfabeto Σ será un 1, en caso contrario será un 0. Recursivamente se aplica el mismo criterio para las mitades de los alfabetos correspondientes hasta llegar a las hojas que corresponden a cada elemento del alfabeto. La altura

del árbol es $\lceil \log \sigma \rceil$ y corresponde al tamaño del alfabeto. El tamaño total de almacenamiento de la estructura es de $n\lceil \log \sigma \rceil$.

Para un mejor entendimiento, se ejemplifica el siguiente caso con la Figura 4.2.

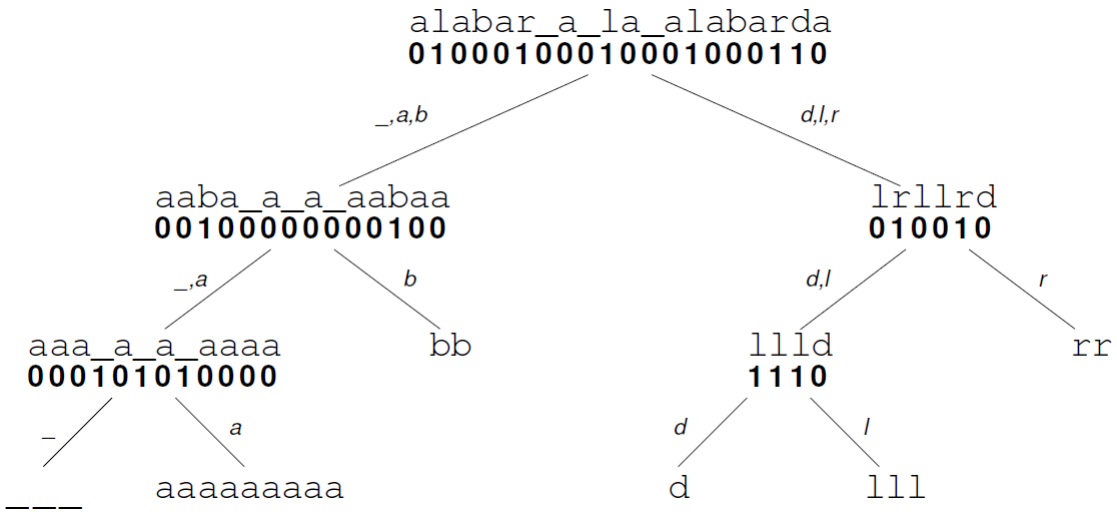


Figura 4.2: Ejemplo de wavelet tree para la secuencia “alabar a la alabarda"[44].

Las características de wavelet tree son:

- $S = \{alabar\ a\ la\ alabarda\}$, de tamaño $n = 20$
- $\Sigma = \{\text{''},a,b,d,l,r\}$, de tamaño $\sigma = 6$
- Altura del wavelet tree $\lceil \log \sigma \rceil = 3$
- Tamaño de la estructura $n\lceil \log \sigma \rceil = 120$ bits.
- Tamaño de la secuencia considerando tipo de dato char $n * 8\ bits = 160$ bits

Comenzando con la raíz, todos los caracteres que estén presentes en la primera mitad del alfabeto se representan a través de un 0, caso contrario con un 1. Luego, siguiendo el orden en que aparecen en la raíz, todos los $0's$ se irán por una rama hacia la izquierda y los $1's$ hacia la derecha. En consecuencia, cada nodo hijo tendrá como alfabeto las letras que siguen en la rama correspondiente, y en base a este nuevo alfabeto se representarán con un 0 los caracteres que correspondan a la primera mitad del nuevo alfabeto, y con un 1 los que no. La estrategia se sigue recursivamente hasta las hojas que corresponden a cada carácter.

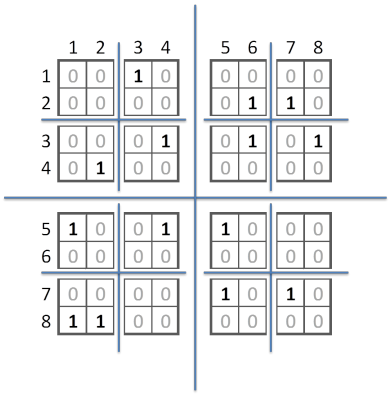
Para recuperar cualquier elemento de la secuencia S se recorre el árbol desde la raíz hasta la hoja que corresponde el índice accesado. El acceso a las distintas posiciones del bitmap en cada nodo se hace a través de las funciones *rank* y *select*.

4.1.3. k^2-tree

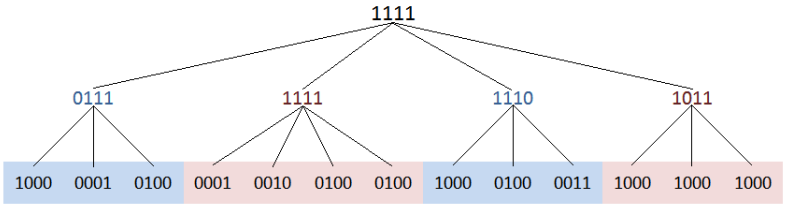
El k^2-tree es una EDC utilizada para representar matrices binarias tomando ventaja de las grandes áreas que contienen $0s$ en la matriz de adyacencia de la relación binaria. Esta EDC fue propuesta originalmente para la representación compacta de grafos web pero posteriormente se ha aplicado a variados campos.

Similar al quadtree [54], el k^2-tree [7] divide la matriz de adyacencia en k^2 cuadrantes de igual tamaño cada vez, es decir, si se define un $k = 2$ la matriz será dividida en cuatro cuadrantes en el orden NorOeste, NorEste, SurOeste y SurEste, y para cada cuadrante resultante, se dividen en k^2 sub-cuadrantes. Esto recursivamente hasta llegar a los valores atómicos de la matriz representada. Cada nodo del árbol representará un cuadrante de la matriz de adyacencia, donde las dimensiones del cuadrante representado dependerán del número del nivel en que se encuentre el nodo.

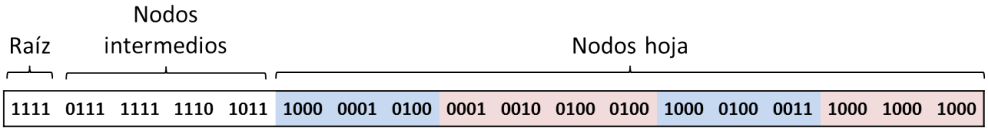
Si un cuadrante posee $1s$ ó $1s$ y $0s$, el bit que lo representa en el nodo es 1 y la división recursiva continua realizándose hasta que se obtiene un cuadrante de k^2 celdas o uno que solo posea $0s$, por otro lado cada vez que se encuentre un cuadrante completo de 0's pasará a ser una hoja con valor 0 (no se continúa realizando la división de ese cuadrante). En la Figura 4.3(b) se muestra la estructura del árbol compuesto de nodos que poseen k^2 bits para la matriz de adyacencia de la Figura 4.3(a), donde los $1's$ representan un nodo hijo, y los $0s$ una rama vacía. Si son nodos hoja, los $1's$ indican la existencia de una relación entre dos objetos, y el valor 0 lo contrario.



(a) Descomposición en cuadrantes de la matriz de adyacencia.



(b) k^2-tree desde Figura 4.3(a)



(c) Bitmap resultante para representar k^2-tree

Figura 4.3: Descomposición de matriz de adyacencia para generar k^2-tree con $k = 2$, ilustrado por Quijada [48].

Importante es mencionar que si una matriz de adyacencia no tiene los k^i nodos exactos, con $i \in \mathbb{N}$, el k^2-tree se representará con una matriz de adyacencia con $i = \lceil log_k n \rceil$, es decir, se tomará la siguiente potencia de k y el contenido de las celdas adicionales se definen con $0's$. Los cuadrantes adicionales generados serán compactados por la estructura ya que solo poseen $0s$. En cualquier caso la altura del árbol resultante corresponde a $\lceil log_k n \rceil$.

Finalmente, una vez construida la estructura no es necesario almacenar el árbol resultante con todos sus punteros. El k^2-tree corresponderá a un bitmap con la concatenación de todos los niveles como se aprecia en la Figura 4.3(c). El bitmap servirá para conocer la información del árbol mediante las operaciones de rank y select.

También existe una variante del k^2-tree, que comprime 1's [2] de la misma forma que el quadtree para evitar la redundancia de datos.

4.2. Objetos espacio-textuales sobre estructuras de datos compactas

En el estado del arte realizado, no se han encontrado trabajos de investigación referente a la representación de conjuntos de objetos espacio-textuales bajo una estructura de datos compacta, por lo que es un campo abierto de exploración.

Por lo anterior, es que en esta investigación se propone una nueva estructura de datos compacta para lograr aprovechar sus propiedades, reducir el espacio de almacenamiento para representar conjuntos espacio-textuales y consecuentemente, mejorar el tiempo de procesamiento de los algoritmos al ejecutarlos directamente en memoria principal.

Lo anterior se justifica pues como se menciona en la Sección 3, todos los métodos actuales trabajan en memoria secundaria y por lo tanto, se focalizan principalmente en reducir las operaciones de E/S y no explotan la opción de reducir el tamaño del almacenamiento ocupado por los conjunto de datos y que estos sean procesados directamente en memoria principal.

Parte III

Estructura de Datos Compacta cBiK para SKQ

Capítulo 5

Representación Espacio-Textual en cBiK

En este capítulo se presenta una nueva EDC llamada cBiK (compact Bitmaps and implicit KD-Tree), para representar conjuntos de datos espacio-textuales y lograr procesar consultas de interés sobre ella.

La estructura cBiK está compuesta principalmente por: i) un KD-Tree implícito balanceado como componente espacial, llamado iKD-Tree y que almacenará las coordenadas geográficas de latitud y longitud y ii) dos bitmaps compactados para representar la componente textual; uno para indicar las palabras claves de los propios puntos y otro para indicar las palabras claves que se encuentran en el subespacio izquierdo y derecho de cada nodo interno del iKD-tree. Adicionalmente, se utiliza un esquema Hashing para representar las palabras clave del conjunto espacio-textual.

Para la compactación de los bitmaps se utiliza la biblioteca *Succinct Data Structure Library* (SDSL) [26]. A continuación, se describe la biblioteca SDSL y luego, cada una de las estructuras de datos mencionadas en detalle.

5.1. SDSL Lite y la compactación de bitmaps

Antes de comenzar a explicar la estructura y construcción de *c*B*i*K, se describe la biblioteca *Succinct Data Structure Library* (SDSL) que es utilizada para varias funcionalidades de este trabajo. Principalmente, SDSL implementa estructuras de datos sucintos y ha sido desarrollada en el lenguaje $C++11$.

Las estructuras de datos sucintas pueden representar un objeto, como un bitvector (bitmap) o un árbol, en el espacio cercano al límite inferior teórico de la información del objeto a la vez que admite las operaciones del objeto original de manera eficiente. La complejidad de tiempo teórica de una operación realizada en la estructura de datos clásica y la estructura de datos sucinta equivalente es idéntica en la mayoría de los casos.

La biblioteca SDSL contiene varias estructuras de datos sucintas agrupadas en las siguientes categorías:

- Bitvectors supporting Rank and Select
- Integer Vectors
- Wavelet Trees
- Compressed Suffix Arrays (CSA)
- Balanced Parentheses Representations
- Longest Common Prefix (LCP) Arrays
- Compressed Suffix Trees (CST)
- Range Minimum/Maximum Query (RMQ) Structures

Dado que los bitmaps son parte del núcleo estructural de la propuesta *c*B*i*K, es que a continuación en la Tabla 5.1 se detallan las clases principales de la categoría Bitvector.

Tabla 5.1: Clases para un bitvector de longitud n con m bits encendidos.

Clase	Descripción	Espacio
bit_vector	bitmap plano	$64[\frac{n}{64}+1]$
bit_vector_il	bitmap intercalado	$\approx n(1+\frac{64}{K})$
rrr_vector	bitmap H_0 compimido	$\approx [log\binom{n}{m}]$
sd_vector	sparse bitmap	$\approx m \cdot (2+log\frac{n}{m})$
hyb_vector	bitmap híbrido	

Para esta investigación, se han utilizado principalmente dos estructuras de datos; el bitmap plano (*bit_vector*) y el sparse bitmap (*sd_vector*). Para este último, SDSL ofrece dos métodos diferentes para su construcción:

1. **Crear el bitvector plano y luego compactarlo:** Este método al crear un bitvector plano de tamaño n tiene la flexibilidad de ir encendiendo los bit en un orden aleatorio y una vez que se termine su llenado, se realiza su compactación. La desventaja es que para un bitvector demasiado largo, el espacio de almacenamiento utilizado es una limitante previo a la compactación.

2. **Crear directamente el bitvector compactado:** Esta estrategia puede crear un bitmap compactado de tamaño n sabiendo de antemano cual es el valor de los m bits que tienen valor 1. La limitante principal es su construcción, ya que las posiciones de los bits a encender deben ingresarse de forma ordenada creciente. La ventaja es que la preocupación por el almacenamiento utilizado para bitvector demasiado largos es mínima.

Dado que los bitmaps que se utilizan para la construcción de *c*B*i*K son enormes, estos se crean directamente de forma compacta utilizando la segunda estrategia mencionada.

Para aprovechar todas las propiedades de las estructuras de datos compactas, a los bitmaps se les puede agregar estructuras para resolver las operaciones de *Rank* y *Select*.

Las tablas 5.2 y 5.3 muestran las estructuras soportadas para cada una de las operaciones. Para *c*B*i*K, se utiliza en gran parte de las consultas la operación $Rank(i)$ de un bitvector plano y $Access(i)$ de un *sd_vector*.

Tabla 5.2: Clases de ayuda para agregar funcionalidad de *Rank* a las estructuras de la Tabla 5.1.

Clase	Compatibilidad	+Bits	Tiempo
rank_support_v	*bit_vector*	0,25n	$O(1)$
rank_support_v5	*bit_vector*	0,0625n	$O(1)$
rank_support_scan	*bit_vector*	64	$O(n)$
rank_support_il	*bit_vector_il*	128	$O(1)$
rank_support_rrr	*rrr_vector*	80	$O(k)$
rank_support_sd	*sd_vector*	64	$O(log\frac{n}{m})$
rank_support_hyb	*hyb_vector*	64	-

Tabla 5.3: Clases de ayuda para agregar funcionalidad de *Select* a las estructuras de la Tabla 5.1.

Clase	Compatibilidad	+Bits	Tiempo
select_support_mcl	*bit_vector*	$\leq$ 0,2n	$O(1)$
select_support_scan	*bit_vector*	64	$O(n)$
select_support_il	*bit_vector_il*	64	$O(log\, n)$
select_support_rrr	*rrr_vector*	64	$O(log\, n)$
select_support_sd	*sd_vector*	64	$O(1)$

5.2. Esquema hashing

Primeramente, para asociar un índice a cada palabra clave de T, se ha utilizado un esquema hashing de tipo $\langle keyword, index \rangle$, dónde la clave del esquema hashing es $keyword \in T$ y su valor es $index$ con $0 \leq index \leq m-1$. En la Figura 5.1 se presenta una instancia de un esquema con cuatro palabras claves ($m = 4$). Como se explicará más adelante, el valor $index$ permite acceder al bit destinado a representar la palabra clave en los bitmaps utilizado en la estructura.

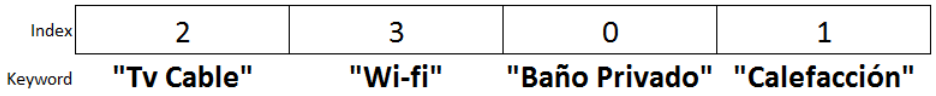


Figura 5.1: Palabras claves asociadas a un índice almacenados en un esquema Hashing.

5.3. *i*KD-Tree para representar la ubicación espacial

El *i*KD-Tree utilizado en *c*B*i*K, se construye usando la misma estrategia del KD-Tree propuesto en [8], el que consigue construir un árbol balanceado independiente de la cantidad de elementos existentes y que difiere en a lo más en un elemento con respecto al subárbol izquierdo y derecho. La Figura 5.2 ejemplifica el resultado final del árbol variando entre uno y siete elementos contenidos. Es importante destacar que la estrategia para balancear el árbol es siempre ir colocando los nodos hijos a la derecha del nodo padre y luego, ir rellenando con los nodos hijos a la izquierda.

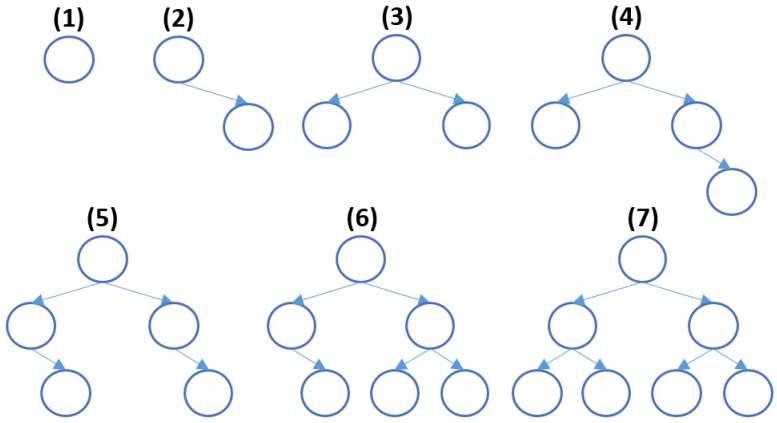


Figura 5.2: Árbol conceptual de diferentes KD-Tree balanceados variando entre uno y siete nodos contenidos.

La diferencia más importante del KD-Tree de [8] con respecto al *i*KD-Tree que se plantea en esta investigación, es la omisión en la utilización de punteros en la construcción del árbol, a cambio, los puntos son almacenados directamente en un arreglo de tamaño n y que se le ha dado el nombre de *nodes*. Como resultado del proceso, se obtiene un KD-tree implícito (*i*KD-Tree), es decir un KD-tree que no utiliza punteros en sus nodos internos para representar sus nodos hijos.

Notar que el *i*KD-Tree conseguido mediante este método también es balanceado y por lo tanto su altura está acotada por $O(\log_2 n)$.

Para construir el *i*KD-Tree, primero se ordenan (de menor a mayor) los objetos espaciales por cada una de las dimensiones (dos en este caso). Luego, y de acuerdo a la coordenada seleccionada para realizar la partición, el elemento de la mitad del arreglo es la raíz del árbol. Por ejemplo, en la Figura 5.3-a) si se considera a x como la coordenada para la orientación de la partición, el punto $(5, 3)$ se almacena en la mitad del arreglo (ver Figura 5.3-b)). Luego se procede de manera recursiva con cada subarreglo alternando la coordenada que orienta la partición del subespacio. De esta forma la coordenada y del punto $(2, 4)$ particiona el subespacio izquierdo y se almacena en la cuarta posición del arreglo *nodes*.

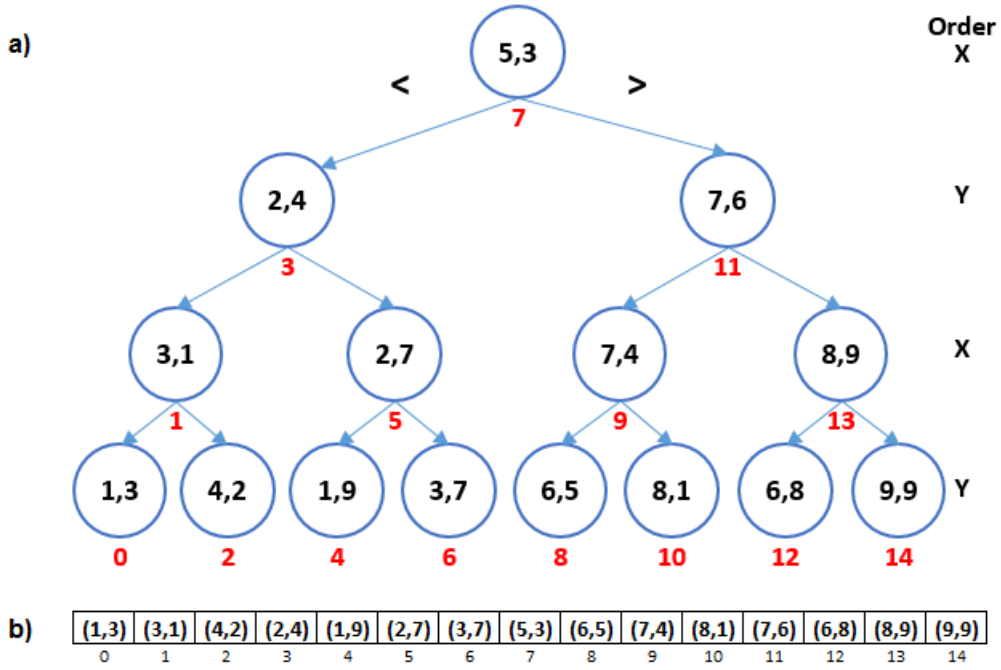


Figura 5.3: a) Árbol conceptual de un *i*KD-Tree balanceado de 15 puntos junto con sus índices implícitos. b) Arreglo *nodes* utilizado para almacenar el *i*KD-Tree.

La Figura 5.4 muestra las particiones del espacio generadas por el *i*KD-Tree de la Figura 5.3-a). La construcción del *i*KD-Tree toma tiempo $O(dn \log n)$, con d el número de dimensiones [8].

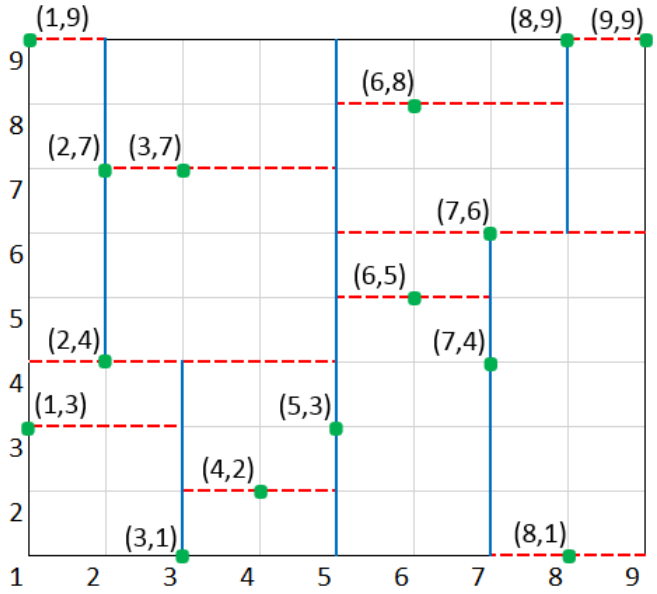


Figura 5.4: Representación espacial del *i*KD-Tree.

Asociado al *i*KD-Tree se usa el Bitmap Mapa (BM) de tamaño $2 \cdot n$. Su primera mitad indica si un nodo del *i*KD-Tree es interno (1) u hoja (0) asociando su índice con el índice implícito del árbol. Por ejemplo, en la Figura 5.3-a) se tienen los índices 0, 2, 4, 6, 8, 10, 12 y 14 todos nodos hojas, correspondientemente en la Figura 5.5 se aprecian los mismos índices mencionados con un valor cero. La segunda mitad del BM, indica si el nodo interno posee un resumen explícito (se explicará en detalle en qué consisten en la Sección 5.4.2). Notar que un nodo tiene un resumen explícito si su bit está encendido en la primera y segundo mitad del BM, por ejemplo, sabiendo que la cantidad de objetos almacenados es $n = 15$ los nodos internos con $i = 3$, $i = 7$ e $i = 11$ poseen un resumen explícito ya que los bits $i = (3+n) = 18$, $i = (7+n) = 22$ e $i = (11+n) = 26$ también están encendidos.

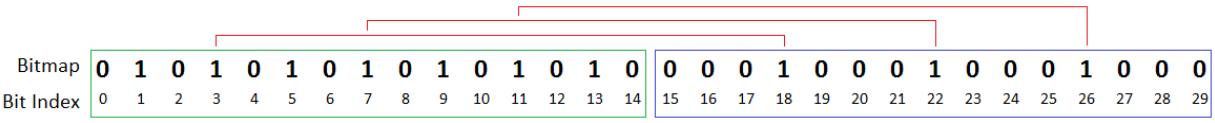


Figura 5.5: Bitmap Mapa correspondiente al *i*KD-Tree de la Figura 5.3. La primera mitad, delimitada por el rectángulo verde, identifica los nodos hojas (0) y nodos internos (1) del árbol. La segunda mitad, en azul, indica todos los nodos internos que poseen un resumen explícito.

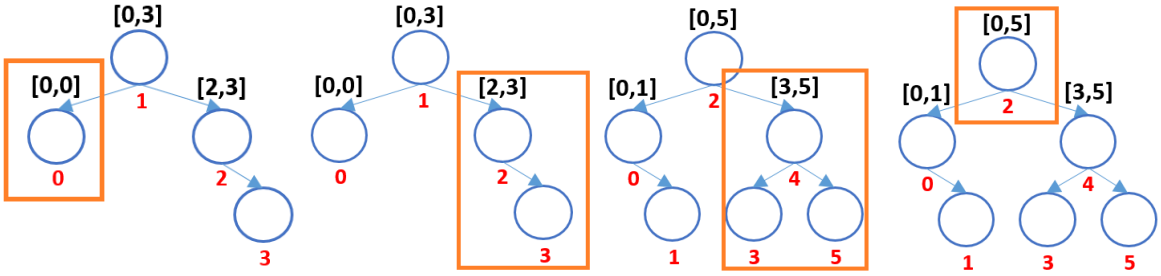
El apoyo funcional del BM es fundamental ya que ayuda a construir parte de la estructura de *c*B*i*K y también facilita muchos cálculos utilizados en los algoritmos de las consultas de interés. Por ejemplo, con BM es posible conocer (mediante la operación $rank_1(BM, i)$) cuántos nodos internos del *i*KD-Tree existen hasta una determinada posición i. En cuanto al espacio requerido, BM utiliza una porción mínima dentro de la estructura de *c*B*i*K, en contraste con los otros bitmaps utilizados para representar las palabras clave asociadas a los objetos.

5.3.1. Recorrido del *i*KD-Tree

Dado que el *i*KD-Tree utilizado es implícito, su recorrido debe ser realizado utilizando auxiliarmente las variables de inicio (*start*) y fin (*end*) del subespacio que se está evaluando, dichas variables indican el índice de posición del subarreglo *nodes*.

En general, el acceso a un nodo i se realiza calculando la mediana de las posiciones de inicio y fin $i = \left\lfloor \frac{start+end}{2} \right\rfloor$. El hijo izquierdo del nodo en la posición i se encuentra en la posición $hi_i = \left\lfloor \frac{start+(i-1)}{2} \right\rfloor$ y el hijo derecho en $hd_i = \left\lfloor \frac{(i+1)+end}{2} \right\rfloor$. Por ejemplo, si se quiere recorrer el árbol de la Figura 5.3-a), se comienza con $start = 0$ y $end = 14$, calculando su mediana se obtiene el nodo raíz con $i = 7$, por lo tanto, su hijo izquierdo es $hi_7 = \left\lfloor \frac{0+(7-1)}{2} \right\rfloor = 3$ y su hijo derecho corresponde a $hd_7 = \left\lfloor \frac{(7+1)+14}{2} \right\rfloor = 11$. Para continuar el acceso a los nodos del siguiente nivel el procedimiento es similar.

Muy importante es mencionar la estrategia utilizada para lograr reconocer cuando se está visitando un nodo hoja o un nodo interno en el recorrido. Dependiendo del intervalo del arreglo *nodes* evaluado y definido por [*start*, *end*], se explican los cuatro casos siguientes:



(a) Caso 1 (b) Caso 2 (c) Caso 3 (d) Caso 4

Figura 5.6: Ejemplo de los cuatro casos en el recorrido por mediana

- **Caso 1** ($end == start$)**:** Dado que sus valores son iguales, la posición de *start* (o *end*) siempre corresponderá a un nodo hoja. El ejemplo más claro es cuando el árbol solo tiene un elemento. Otro ejemplo, es cuando el árbol posee cuatro nodos y se evalúa el intervalo del hijo izquierdo de la raíz (ver Figura 5.6-a).

- **Caso 2** ($end == start + 1$)**:** Indica que la posición *start* siempre será un nodo interno, y *end* su hijo derecho de tipo hoja. Puede verificarse con el árbol de la Figura 5.6-b, cuando se quiere evaluar el intervalo del hijo derecho de la raíz.

- **Caso 3** ($end == start + 2$**:**) Corresponde a un intervalo con tres nodos. Primero $start + 1$ representa el nodo interno que tiene sus dos hijos hoja, *start* como el izquierdo y *end* el derecho. El ejemplo de este caso se retrata en la Figura 5.6-c al revisar el intervalo del hijo derecho de la raíz.

- **Caso 4** ($end > start + 2$)**:** Este caso se encuentra en los niveles superiores e intermedios del árbol, contrario a los otros casos que ocurren en los niveles más bajos. En este punto

se realiza el recorrido recursivo en profundidad para lograr llegar a uno de los tres casos anteriores. Por ejemplo en la Figura 5.6-d la raíz tiene el intervalo $start = 0$ y $end = 5$ indicando que su mediana ($i = 2$) es un nodo interno, a su vez, el intervalo de su hijo izquierdo y derecho están caracterizados por el caso 2 y 3 respectivamente.

5.4. Bitmaps para representar las palabras claves

Como se mencionó al inicio de este capítulo, la estructura de datos *cBiK* utiliza dos bitmaps para indicar tanto las palabras claves de cada punto como las presentes en cada uno de los subespacios generados por el *i*KD-tree. A continuación se describe cada uno de estos bitmaps.

5.4.1. Bitmap Keywords (BK)

La idea base es que al tener un bitmap de tamaño m en cada objeto espacial es posible representar sus palabras claves asociadas. Si la i-ésima palabra clave de T se encuentra presente en el punto p, entonces el i-ésimo bit de p está puesto a 1 y a 0 en caso contrario. En la Figura 5.7-a se muestra el árbol conceptual del *i*KD-Tree dónde cada nodo posee sus coordenadas y su bitmap asociado para representar las palabras claves que lo describen. Por ejemplo, el bitmap 1010 del punto (8,1) ubicado en el índice 10, indica que tiene las palabras claves "Baño Privado" y "Tv Cable" del conjunto definido en la Figura 5.1.

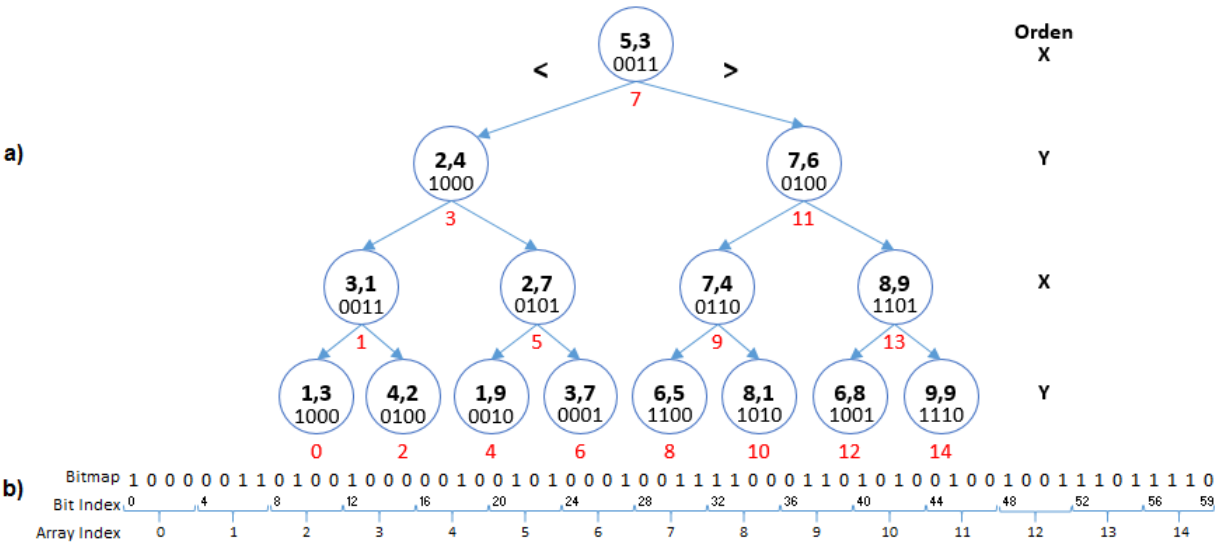


Figura 5.7: Árbol conceptual del *i*KD-tree y su bitmap Keywords.

Dado que m y n pueden llegar a ser muy grandes, almacenar un bitmap en cada objeto espacial supondrá un almacenamiento en memoria muy costoso debido a que la compactación por separado de cada bitmap sería mínimo, en consecuencia, el BK, de tamaño $m \cdot n$, se construye

directamente de forma compacta con la unión de todos los bitmaps y en el mismo orden en que se almacenan los puntos en el arreglo *nodes* (Figura 5.7-b). Por ejemplo, los primeros cuatro bits de BK corresponden al bitmap del punto (1,3) almacenado en la posición 0 de *nodes*.

La biblioteca $SDSL$, explicada en la Sección 5.1, se utiliza para la compactación de BK de forma directa. Notar que para construir los sparse bitmaps, $SDSL$ pide como requisito indicar las posiciones de los bits a encender de forma ordenada ascendente. El Algoritmo 5.4.1 muestra como se construye el BK al recibir el arreglo *nodes*. Este último se recorre para ir obteniendo sus palabras claves, siendo i el índice del arreglo y $newIndex$ la variable que indica la posición del primer bit del nodo i en el BK y que está dado por $m \cdot i$ en la línea 3. Entre las líneas 4-6 se realizan tantas iteraciones como palabras clave existan en el objeto, donde $posKey$ representa la posición del bit de la palabra clave según el esquema hashing. Finalmente en la línea 5, dicho valor se suma al de la variable $newIndex$ para obtener la posición real del bit a encender en el bitmap BK.

Algorithm 5.4.1 makeCompactKeywords(*nodes*)

```
1: newIndex ← 0
2: for i = 0 to i < nodes.size() do
3:     newIndex ← m · i
4:     for all posKey ∈ nodes[i] do
5:         BK.set(newIndex + posKey)          ▷ BK es una variable global
6:     end for
7: end for
```

En general, el acceso a los bits asociados a un punto ubicado en la posición i de *nodes* se encuentran en el rango de posiciones $[i \cdot m, (i+1) \cdot m - 1]$ de BK, con $0 \leq i < n$. Por ejemplo, si se quieren conocer el rango de las palabras claves asociadas al punto (7,4) que posee índice 9 en el arreglo *nodes*, se calcularía de la siguiente forma: $[9 \cdot 4, (9+1) \cdot 4 - 1]$. Lo anterior entrega el rango $[36, 39]$ de BK.

5.4.2. Bitmap Resumen (BR)

Este bitmap se utiliza para representar las palabras claves que se encuentran en los puntos de cada uno de los subespacios del iKD-Tree. La idea inicial es que por cada nodo interno p se asocie un bitmap de tamaño $2 \cdot m$ que denominamos resumen local (RL). Los primeros m bits representan las palabras claves que se encuentran en alguno de los puntos ubicados en el subespacio de la izquierda de p y los m restantes a palabras claves de puntos ubicados en el subespacio de la derecha de p.

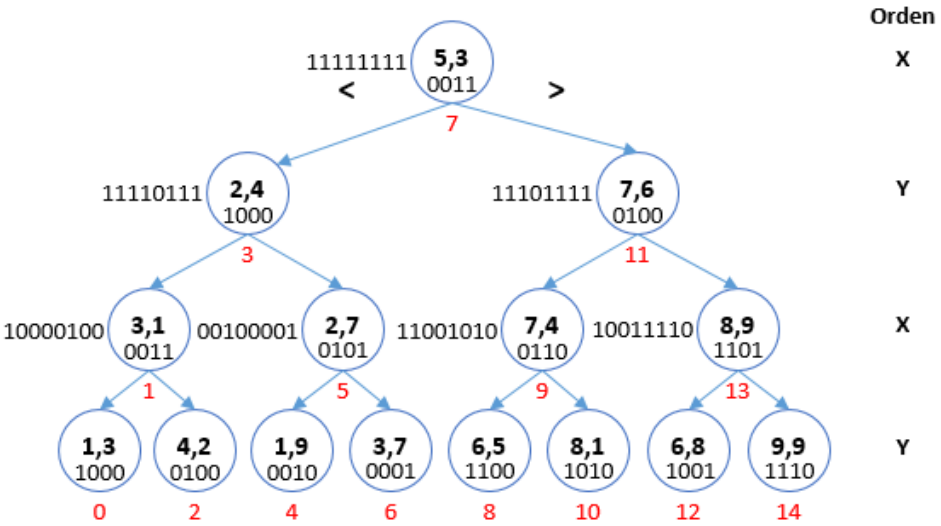


Figura 5.8: Árbol conceptual del *i*KD-tree junto con la idea básica de sus bitmaps RL's.

En la Figura 5.8 es posible ejemplificar tal idea y se puede visualizar que los correspondientes RL's se ubican a la izquierda de cada nodo interno. Si los hijos de un nodo son hojas, entonces su RL se forma concatenando sus correspondientes BK's, primero el hijo izquierdo y luego el derecho. Por ejemplo, el RL del nodo $i = 1$ de la Figura 5.8 corresponde a la concatenación de los BK's de los nodos $i = 0$ y $i = 2$ (puntos $(1, 3)$ y $(4, 2)$).

Para el caso en que nodo interno tenga a su vez hijos nodos internos, su RL se obtiene de la siguiente manera: los primeros m bits se obtienen realizando la operación bitwise *or* ($\vee$) entre los primeros m bits con los m bits restantes del RL del hijo izquierdo. Dicho resultado es operado nuevamente con el operador *or* con el BK del hijo izquierdo. Por ejemplo, los $m = 4$ primeros bits del RL asociado al nodo $i = 3$ (punto (2,4)) de la Figura 5.8 se obtienen de la siguiente manera: $1111 = (1000 \vee 0100) \vee 0011$. De manera similar se obtienen los m bits para representar las palabras claves del subespacio derecho.

A modo de ejemplo, para el nodo con $i = 3$ del *i*KD-Tree (punto (2,4)) su RL es 11110111. Esto significa que todas las palabras claves de la Figura 5.1 aparecen en los puntos ubicados en el subespacio izquierdo, definido por el rectángulo cuyos puntos extremos son (1,1) y (5,4) (ver Figura 5.4). A su vez los siguientes 4 bits indican que las palabras claves *"Calefacción"*, *"Tv Cable"* y *"Wi-fi"* aparecen en el subespacio derecho, definido por el rectángulo con puntos extremos (1,4) y (5,9).

Mirando con atención, la propuesta básica de tal estrategia hace que la información contenida en los $RL's$ de los nodos internos inferiores, los que como hijo solo tienen uno o dos nodos hojas, redunde con información contenida y fácilmente accesible desde el BK. Por ejemplo, el RL del punto $(3, 1)$ antes mencionado puede obtenerse de forma inmediata accediendo a los BK de sus hijos y no valdría la pena almacenarlos, a diferencia del RL del punto $(2, 4)$ que para recuperarlos se requiere de un cálculo más complejo. Por lo anterior, los $RL's$ se clasificarán de la siguiente manera:

- **Resumen Local Explícito** (eRL)**:** Los que su RL no es de acceso directo y necesita ser

precalculado, por lo tanto, se debe almacenar para su posterior acceso inmediato.

- **Resumen Local Implícito (iRL):** Los que su RL se obtiene mediante el acceso a los BK de sus hijos y por lo tanto, no necesitan almacenarse.

Al igual que el bitmap BK, para evitar el exagerado costo de almacenamiento los eRL's se guardan directamente en un sparse bitmaps compacto llamado Bitmap Resumen (BR), este se refleja en la Figura 5.9.

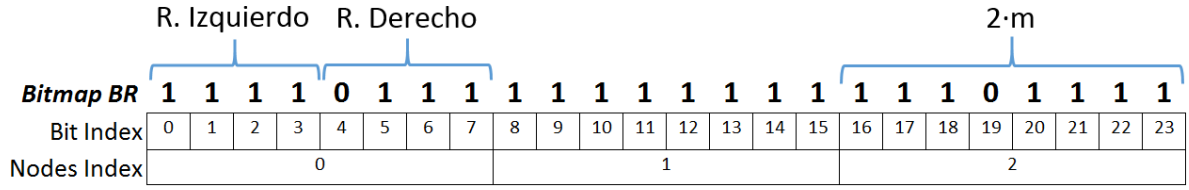


Figura 5.9: Bitmap Resumen correspondiente a los nodos internos con i igual a 3, 7 y 11 del iKD-Tree de la Figura 5.8.

Según la estrategia antes mencionada, el Algoritmo 5.4.2 muestra el proceso de construcción de los eRL's. La idea general es ir acumulando de forma recursiva las posiciones de las palabras claves desde las hojas hacia la raíz con un recorrido en In-order. Dichos resúmenes se guardan en una estructura que no acepta valores repetidos y que luego pueda ser recorrida por sus valores en orden ascendente debido a la restricción de la bibloteca $SDSL$ para crear el bitmap compacto (explicado en la Sección 5.1). En este caso se ha utilizado un Árbol Binario de Búsqueda llamado $termSum$.

Algorithm 5.4.2 crearResumen($start$, end)

1: $termSum$ ▷ Es un ABB
2: **if** $(end == start) \vee (end == start + 1)$ **then**
3: $addTerm(termSum, end)$
4: **else if** $(end == start + 2)$ **then**
5: $addTerm(termSum, start)$
6: $addTerm(termSum, end)$
7: **else if** $(end > start + 2)$ **then**
8: $median \leftarrow \frac{(start+end)}{2}$
9: $pos = getBRIndex(median)$
10: $resumenI = crearResumen(start, median - 1)$ ▷ Recibe un ABB
11: $addTerm(resumenI, \frac{start+(median-1)}{2})$
12: $addTerm2(resumenI, termSum, pos)$
13: $resumenD = crearResumen(median + 1, end)$ ▷ Recibe un ABB
14: $addTerm(resumenD, \frac{(median+1)+end}{2})$
15: $addTerm2(resumenD, termSum, (pos + m))$
16: **end if**
17: **return** $termSum$

Notar que el algoritmo recibe los índices *start* y *end* del arreglo *nodes* para conseguir evaluar los nodos de los subarreglos mediante su mediana. Se consideran tres casos distintos según sean sus valores:

Líneas 2-4: Caso cuando es una hoja ($end = start$) o, cuando $end = start + 1$ corresponde a un nodo interno (*start*) junto con su hijo derecho hoja (*end*). Al ser una hoja o un hijo derecho, se envía la posición de *end* a la función $addTerm(.,.)$ (ver Algoritmo 5.4.3) para agregar la posición de palabras claves que existen en $nodes[end]$ (línea 3). Como el nodo no posee un *eRL* solo se agregan a *termSum* para ir acumulando los resúmenes.

Algorithm 5.4.3 addTerm(*termSum*, *index*)

```
1: for all term ∈ nodes[index] do
2:     termSum.insert(term)                    ▷ termSum es un ABB
3: end for
```

Líneas 4-7: Caso cuando es un nodo interno ($start+1$) y sus dos hijos son hojas (hijo izquierdo *start* e hijo derecho *end*). Aquí existen ambos hijos hojas, pero no posee un *eRL*, por lo que se agregan el hijo izquierdo (línea 5) y el hijo derecho (línea 6) a *termSum*.

Líneas 7-17: Caso cuando es un nodo interno (*median*) y sus hijos también lo son. Aquí si existen $eRL's$ por lo que el proceso es más complejo.

En este punto es necesario explicar la función auxiliar $getBRIndex(.)$ mostrado en el Algoritmo 5.4.4. Sabiendo que el nodo i recibido posee un *eRL* entonces existe un rango de posiciones de largo $2 \cdot m$ destinado a albergar los resumenes izquierdo y derecho del propio punto, por lo tanto, la función devuelve la primera posición de ese rango con el propósito de usarlo como base para el encendido correcto de los bits resumenes del punto. Para ello, es necesario conocer cuál es el índice del segmento de *BR* que corresponde a i. Utilizando el bitmap *BM* es fácilmente identificable usando $Rank_1$. Primero se obtiene cuántos nodos internos tiene el iKD-Tree (línea 1) y luego los nodos internos más los *eRL* hasta la posición $n + i$. Finalmente se devuelve la diferencia de estos multiplicado por el largo $2 \cdot m$ de cada segmento (línea 3).

Algorithm 5.4.4 getBRIndex(i)

```
1: n1 ← Rank_1(BM, n)          ▷ Cantidad de nodos internos hasta n
2: n2 ← Rank_1(BM, n + 1)      ▷ Cantidad de nodos internos y eRL's hasta i + n
3: return 2 · m · (n2 − n1)
```

Detallada la función anterior se continúa con la explicación de las líneas 7-17. Lo primero es obtener mediante la función $getBRIndex(.)$ el menor valor del rango de posiciones del nodo evaluado por *median* (línea 9). Como se explicó dicha posición es el comienzo del segmento destinado a ese eRL, en consecuencia, se sumará con las posiciones de cada palabra clave existente para lograr encender el valor correcto en el bitmap *BR*. Luego se hace la llamada recursiva para obtener los resumenes del hijo izquierdo de *median* (línea 10). Lo anterior

retorna todas la palabras claves que están presentes en el subespacio izquierdo de *median* y se guardan en $resumenI$. Debido a que el resumen de un punto no posee las palabras claves propias del punto, se hace necesario agregar a $resumenI$ las palabras claves del hijo izquierdo de *median* (línea 11). De esta forma se logra conformar todo el resumen izquierdo de *median* que finalmente en la línea 12 se seguirán acumulando en $termSum$ (para ayudar en un posterior regreso de la recursividad) y se agregan al resumen final (que es una variable global llamada $resumenFinal$) con la función $addTerm2(.,.,.)$ (ver Algoritmo 5.4.5). El proceso es análogo para el lado derecho (líneas 13-15).

Algorithm 5.4.5 addTerm2($resumen$, $termSum$, pos)

```
1: for all term ∈ resumen do
2:     termSum.insert(term)
3:     resumenFinal.insert(pos + term)        ▷ resumenFinal es un ABB y variable global
4: end for
```

Notar que $resumenFinal$ almacenará todas las posiciones a encender en el BR, por lo que terminado todo el proceso de generación de resumenes, se crea el bitmaps compacto BR recorriendo de forma ascendente los valores contenidos en la variable $resumenFinal$ e ir asignado los 1's correspondientes (similar al del Algoritmo 5.4.1).

Para recuperar el RL de un nodo interno indexado por j en el arreglo *nodes* se procede de la siguiente manera. En primer lugar se necesita saber si el RL es un iRL o un eRL, para ello, se confirma que sea un nodo interno con la primera mitad del BM accediendo a la posición j y que su valor sea 1. Luego, basta con revisar la segunda mitad del BM en la posición $n + j$, si su valor es 0, indica que j posee un iRL y por ende su RL corresponde a los hijos hoja de j, el acceso a cada hijo es análogo al del BK. En caso contrario, si el bit está encendido j posee un eRL y por lo tanto se debe acceder al BR directamente. El proceso anterior como se sabe no es tan directo, primero se debe conocer cuántos nodos internos existen en total revisando la primera mitad del BM mediante la operación $n_i = rank_1(BM, n)$, luego se obtiene la cantidad de nodos internos que poseen un eRL hasta la posición j con la operación $i = (rank_1(BM, n + j) - n_i)$ y finalmente, el RL del nodo j se encuentra en el rango de posiciones $[2 \cdot i \cdot m, 2 \cdot m \cdot (i + 1) - 1]$, con $i \geq 0$ del bitmap compacto BR.

Un ejemplo de lo anterior, puede ser conocer el RL del punto $(2, 4)$ almacenado en $j = 3$ de la Figura 5.8. Para hacer la confirmación de que es un nodo interno, se revisa el BM de la Figura 5.5 accediendo al bit con $j = 3$. Como sí lo es ya que el bit está encendido, se pasa a revisar la posición $n + j$ de BM para determinar de qué tipo es su RL. Dado que el valor de la posición $15 + 3 = 18$ es 1, el nodo es un eRL y por lo tanto se debe acceder al BR directamente. Para ello, se calcula el total de nodos internos existentes en el iKD-Tree con $n_i = rank_1(BM, 15) = 7$ (los unos de la primera mitad del BM), luego se calcula el índice del BR con la diferencia de los nodos internos hasta $n + j$ y n_i usando $i = (rank_1(BM, 15 + 3) - 7) = 0$, finalmente se sabe que el resumen local está en el rango $[2 \cdot 0 \cdot 4, 2 \cdot 4 \cdot (0 + 1) - 1]$ o lo que es igual a $[0, 7]$, se puede corroborar observando la Figura 5.9.

Notar que el bitmap BR jugará un rol muy importante en el procesamiento de las consultas espacio-textuales pues permite que el índice cBiK logre podar ramas completas usando al mismo

tiempo la dimensión espacial y textual.

Capítulo 6

Algoritmos para Procesar Consultas SKQ

Una vez construido el esquema Hashing, el iKD-Tree y los tres Bitmaps (BM, BK, BR) es posible evaluar consultas SKQ sobre la estructura cBiK.

En esta investigación se exploran los tres tipos de consultas más estudiadas en la literatura, a saber:

- Boolean Top-k Spatial Keyword Query (BkSKQ)
- Ranked Top-k Spatial Keyword Query (RkSKQ)
- Boolean Range Searching Spatial Keyword Query (bRS-SKQ)

Importante es destacar que antes de realizar la búsqueda propiamente tal, en todos los algoritmos se realiza previamente la recuperación de la claves numéricas de cada una de las palabras claves mediante el uso del esquema Hashing. Dichas claves se usan para acceder a los bitmaps BK y BR con el propósito de determinar si la palabra existe tanto en un punto o en una subespacio del iKD-Tree.

Se asume que tanto el vector que representa el iKD-Tree ($nodes$) como los tres bitvector BM, BK y BR son variables globales y por lo tanto no se indican como parámetros en las diferentes funciones. El detalle y la explicación de cada algoritmo se presenta a continuación.

6.1. Evaluación de consultas BkSKQ con cBiK

Como se detalló en el Capítulo 2, la consulta BkSKQ recupera los k objetos espacio-textuales que están más cercanos espacialmente a un punto dado y que además, satisfagan completamente un conjunto de palabras claves buscadas.

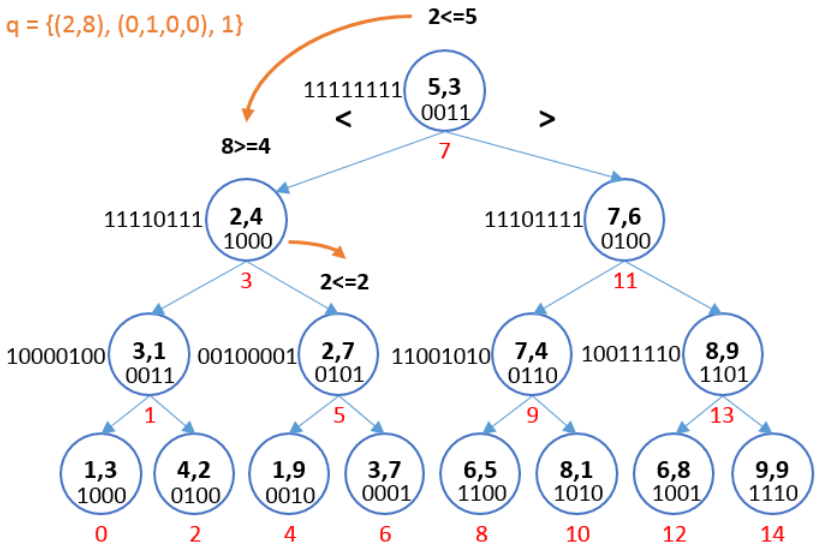


Figura 6.1: Ejemplo de consulta BkSKQ dado un q y el camino de búsqueda sobre el árbol conceptual de la estructura cBiK. Cada nodo posee la información espacial, las palabras claves del punto y el bitmap resumen para una poda eficiente por izquierda o derecha.

Para evaluar las consultas BkSKQ con cBiK se sigue, en general, la misma idea detrás del algoritmo clásico para encontrar los k vecinos más cercanos (kNN) por medio del KD-Tree tradicional. Es decir, desde la raíz del árbol se desciende hasta las hojas alternando el eje o dirección utilizada para la partición de los subespacios. Tanto en el descenso (nodos hojas) como en el ascenso o regreso de la recursividad (nodos internos) los puntos de los nodos se utilizan para mejorar la solución. Esta estrategia puede ser utilizada para la consulta BkSKQ ya que se prioriza encontrar los puntos más cercanos al punto dado en función de la distancia euclídea y luego, basta determinar si el punto candidato presenta la totalidad de las palabras claves buscadas para agregarlas a la solución.

Este tipo de consulta q está compuesta por tres parámetros, que se ilustran en la Figura 6.1. Primero el punto a consultar $(2, 8)$, luego las palabras claves buscadas "Calefacción" (representado por su bitmap) y finalmente el valor de $k = 1$. También se aprecia el árbol conceptual del iKD-Tree, dónde cada nodo posee los bitmaps de sus palabras claves asociadas y los nodos internos tienen su bitmap resumen correspondiente. Dicha consulta comienza desde la raíz evaluando la partición del eje x. Como el valor de la coordenada del punto de consulta es menor al del punto evaluado ($2 \leq 5$) entonces se sigue el recorrido por el lado izquierdo ya que su resumen (la primera mitad de bits) indica que la palabra clave buscada está presente en ese subespacio. Luego el eje a revisar es el y, el valor de la coordenada del punto de consulta es mayor al del punto ($8 \geq 4$), su

recorrido continúa por el subárbol derecho al verificar que existe presencia de la palabra clave en el resumen derecho del punto (la segunda mitad de bits). Luego, nuevamente se revisa la partición del eje x y como el valor de la coordenada de la consulta es menor o igual al del punto ($2 \leq 2$), corresponde seguir por el lado izquierdo pero el resumen del punto indica que no existe la palabra clave en ningún subespacio siguiente, por lo tanto, se termina el recorrido en profundidad y se evalúa al punto como un candidato de solución. En el regreso de la recursividad se sigue evaluando los puntos visitados para mejorar la solución si corresponde.

Algorithm 6.1.1 searchBkSKQ(q, $start$, end, $depth$, $heap$)

```
1:  mid ← (start+end)/2
2:  p ← nodes[mid]
3:  if (start ≠ end) then                                   ▷ Es un nodo interno
4:      c_q ← getCoordinate(depth, q.l)
5:      c_p ← getCoordinate(depth, p)
6:      if (c_q ≤ c_p) then                                 ▷ Busca por la izquierda
7:          if (checkLS(q.t, mid, start, end)) then
8:              searchBkSKQ(q, start, mid − 1, depth + 1, heap)
9:              if existsRight(heap, q.l) ∧ checkRS(q.t, mid, start, end) then
10:                 searchBkSKQ(q, mid + 1, end, depth + 1, heap)
11:             end if
12:         else if (checkRS(q.t, mid, start, end)) then
13:             searchBkSKQ(q, mid + 1, end, depth + 1, heap)
14:         end if
15:     end if
16:     if (c_q ≥ c_p) then                                 ▷ Busca por la derecha
17:         if (checkRS(q.t, mid, start, end)) then
18:             searchBkSKQ(q, mid + 1, end, depth + 1, heap
19:             if existsLeft(heap, q.l) ∧ checkLS(q.t, mid, start, end) then
20:                 searchBkSKQ(q, start, mid − 1, depth + 1, heap)
21:             end if
22:         else if (checkLS(queryKey, mid, start, end)) then
23:             searchBkSKQ(q, start, mid − 1, depth + 1, heap)
24:         end if
25:     end if
26: end if
27: if checkKeywords(q.t, mid) then
28:     updateCandidate(heap, p, q.l, q.k)
29: end if
```

El algoritmo se muestra en Alg. 6.1.1 y recibe cinco parámetros, q representa a la variable de consulta y posee un atributo espacial $q.l$, uno textual $q.t$ y un atributo $q.k$ para determinar cuantos resultados se esperan obtener; *start* y *end* definen la zona del arreglo *nodes* donde continuar el recorrido en el iKD-Tree; inicialmente $start = 0$ y $end = n - 1$. La variable *depth* se utiliza para

indicar la profundidad del iKD-Tree y con su valor se decide la dirección de las particiones de los subespacios. Finalmente, la variable *heap* corresponde a un Max-Heap de tamaño k utilizado para mantener los puntos (objetos espacio-textuales) candidatos de la solución. Debido a que todos los puntos que contiene el *heap* poseen la totalidad de las palabras claves, estos se encuentran organizados por la mayor distancia al punto de la consulta $q.l$. La respuesta de la consulta queda finalmente en *heap*. Notar que puede ocurrir que no sea posible conseguir de D los k objetos que satisfagan todas las palabras claves definidas en $q.t$ y por lo tanto, el número de objetos contenidos en el *heap* puede ser menor que k.

El algoritmo comienza obteniendo la mediana de las posiciones recibidas *start* y *end* (línea 1), dicho valor corresponde a la posición del punto accedido en el arreglo *nodes* que almacena el iKD-Tree. Con dicho valor se accede al punto y se guarda en la variable p.

En las líneas 3-26 se resuelve el caso de los nodos internos para lograr recorrer recursivamente el árbol en profundidad. Las líneas 4 y 5 recuperan los valores de coordenada que corresponde a la dirección de la partición de acuerdo a la profundidad del iKD-Tree (*depth*), tanto del punto $q.l$ de la consulta como del punto p del nodo.

En las líneas 6-15 se procesa el caso en que el punto $q.l$ se encuentra a la izquierda del punto p ya que el valor de c_q es menor o igual al de c_p. Luego, se verifica mediante la función $checkLS(.,.,.,.)$ (línea 7) si el bitmap BR asociado al subespacio izquierdo del nodo contiene todas las palabras claves indicadas en $q.t$, es decir, si $q.t$ es un subconjunto de la unión de las palabras claves de todos los puntos ubicados en el subespacio izquierdo. De ser así, se continúa el recorrido accediendo recursivamente a la primera mitad (subárbol izquierdo) del subarreglo de *nodes* entre *start* y *end* (línea 8). Si no las tuviera se revisa la otra rama del iKD-Tree para conocer posibles candidatos y acceder a ella (línea 12-14). A continuación, mediante la función $existsRight(.,.)$ (se explica más adelante), se verifica si es necesario explorar los puntos del subespacio de la derecha de p con el propósito de averiguar si mejoran la solución conseguida hasta el momento. Análogamente, en las líneas 16-25 se procesa el caso cuando el recorrido debe continuar por el lado derecho de p.

En las líneas 27-29 se resuelve el caso en el que el recorrido en profundidad ha alcanzado un nodo hoja y debe ser revisado, o cuando se regresa de la recursividad y la solución debe ser mejorada. En tales casos, se verifica mediante la función $checkKeywords(.,.)$ (se explica más adelante) si el punto contiene todas las palabras claves especificadas en la lista $q.t$, si corresponde, el punto es considerado un candidato y se utiliza para actualizar el *heap* mediante la función $updateCandidate(.,.,.,.)$.

El algoritmo usa las siguientes funciones (no triviales) que se explican a continuación:

checkKeywords($q.t, pos$): Esta función revisa en el punto visitado si en el bitmap BK existen las palabras claves de la consulta $q.t$. Como se aprecia en el Algoritmo 6.1.2 se reciben dos parámetros, $q.t$ representa un arreglo con las posiciones de las palabras claves de la consulta determinado por el esquema Hashing. La variable *pos* indica el índice del punto en el arreglo *nodes* que se quiere revisar.

En la línea 2, se obtiene el menor índice del rango de posiciones en BK de las palabras claves del objeto en *pos*. En las líneas 3-8 se evalúa si existen todas las palabras claves buscadas, iterando por tantas palabras tenga la consulta, si no existe una palabra se retorna $false$. En las líneas

Algorithm 6.1.2 checkKeywords($q.t$, pos)

```
1: i ← 0
2: startIndexK ← pos * m
3: while (i < q.t.size()) do
4:     if (BK[(startIndexK + q.t[i])] = 0) then
5:         return false
6:     end if
7:     i++
8: end while
9: return true
```

4-6 se revisa si el bit en BK está apagado para determinar si es necesario seguir chequeando las siguientes palabras claves. Finalmente en la línea 9, se retorna *true* si el chequeo ha sido satisfactorio.

checkLS($q.t, pos, start, end$) y checkRS($q.t, pos, start, end$): Estas funciones se utilizan para revisar si existen las palabras claves de la consulta $q.t$, en el subespacio izquierdo o derecho del punto visitado en pos. El algoritmo 6.1.3 detalla el proceso de revisión del subespacio derecho.

Algorithm 6.1.3 checkRS($q.t$, pos, $start$, end)

```
1: if (BM[pos] = 0) then                                   ▷ Es una hoja
2:     return false
3: else                                                    ▷ Es un nodo interno
4:     i ← 0
5:     if (end = start + 1) ∨ (end = start + 2) then
6:         resp ← checkKeywords(q.t, end)
7:     else if (end > start + 2) then
8:         startIndexR ← getBRIndex(pos)
9:         while (i < q.t.size()) do
10:            if (BR[(startIndexR + m + q.t[i])] = 0) then
11:                return false
12:            end if
13:            i++
14:        end while
15:    else                          ▷ Es una hoja, no tiene resumen (end = start)
16:        return false
17:    end if
18: end if
19: return true
```

El algoritmo, en la línea 1, comprueba que el nodo revisado sea realmente un nodo interno, si no lo fuera, se retorna $false$ y se termina el proceso de revisión ya que los nodos hojas no tienen

bitmaps resumenes. En las líneas 5-7 se evalúa el caso cuando se tiene un resumen local implicito (iRL), tal situación se da cuando el nodo interno tiene un único nodo hijo derecho de tipo hoja (end), o cuando el nodo interno ($start + 1$) tiene ambos hijos, el izquierdo ($start$) y el derecho (end) de tipo hoja. Ya que para hacer la comprobación de un iRL se debe acceder a las palabras claves de BK y no en el bitmap BR, el proceso es similar al del Algoritmo 6.1.2.

En las líneas 7-15 se evalúa el caso cuando se tiene un nodo interno con un resumen local explícito (eRL), por lo tanto, la revisión de las palabras claves se hace directamente en el bitmap BR. Para ello, se obtiene mediante la función $getBRIndex(.)$ (vista en el Algoritmo 5.4.4) el menor índice del rango de posiciones del punto pos en BR (línea 8). Luego se revisa la presencia de las palabras claves consultadas iterando cuanto fuese necesario. Notar que $checkRS$ intenta revisar el subespacio derecho del punto y ya que estos poseen un RL de largo $2 \cdot m$, los accesos a sus bits (línea 10) deben ser realizados con m posiciones extras (las primeras m posiciones corresponden al subespacio izquierdo).

La revisión del subespacio izquierdo con la función $checkLS(.,.,.,.)$ es similar por lo que su algoritmo se omitirá.

updateCandidate($heap, p, q.l, q.k$): Esta función se utiliza para actualizar con el punto p los candidatos del $heap$. Se asume que p es también un candidato, es decir contiene todas las palabras claves de la consulta. La función básicamente decide si insertar p al $heap$. Para ello verifica si el tamaño del heap es menor que k: Si es así, p es insertado. En caso contrario, se verifica si la distancia euclídea entre p y $q.l$ es menor que la distancia entre $q.l$ y el punto de la raíz de $heap$. Si es así, se elimina el punto de la raíz y se inserta p.

existsLeft($heap, q.l$) y existRight($heap, q.l$): Se utilizan para decidir, al regresar desde la recursividad en los nodos internos del iKD-Tree, si es necesario continuar el recorrido, a partir del punto, por el subespacio izquierdo o derecho dependiendo si previamente el recorrido se realizó por el subespacio derecho o izquierdo respectivamente. Si el tamaño del heap es menor que k el recorrido se realiza de todas maneras. En caso contrario se verifica si la circunsferencia con centro en el punto c de la raíz de $heap$ y radio igual a la distancia entre $q.l$ y c intersecta el subespacio no explorado por la recursividad.

6.2. Evaluación de consultas RkSKQ con cBiK

A diferencia de BkSQK, la consulta RkSKQ no busca los objetos espacio-textuales que posean una coincidencia exacta con todas las palabras claves consultadas, sino que necesita que exista al menos un término coincidente para que el objeto espacio-textual sea candidato.

La estrategia se basa en buscar los objetos espacio-textuales que están más cercanos al punto de consulta, clasificando sus resultados mediante la suma del puntaje de la proximidad espacial (distancia euclídea entre ellos) y el puntaje asignado por la relevancia textual (cantidad de las palabras claves que posee). La idea general es recorrer en profundidad el árbol, partiendo desde la raíz e ir decidiendo según el puntaje total de clasificación de los resumenes, por cuál subespacio continuar hasta llegar a un nodo hoja. En el regreso de la recursividad se evalúan los puntos visitados para construir e ir mejorando la solución.

Recordar que la consulta BkSKQ utiliza las particiones de los subespacios como guía de navegación para recorrer el árbol y además, se ayuda de los resumenes de los nodos internos chequeando si existe coincidencia exacta de los términos buscados en sus subespacios, por lo tanto, dicha estrategia se basa en la priorización de minimizar la distancia euclídea entre el punto q y los puntos del iKD-Tree. Por otro lado, RkSKQ debe maximizar el puntaje de clasificación de los puntos y las palabras claves, por lo tanto, no es posible usar la misma estrategia que BkSKQ. Para decidir por dónde recorrer el árbol se utiliza la comparación del puntaje total de clasificación de sus subespacios.

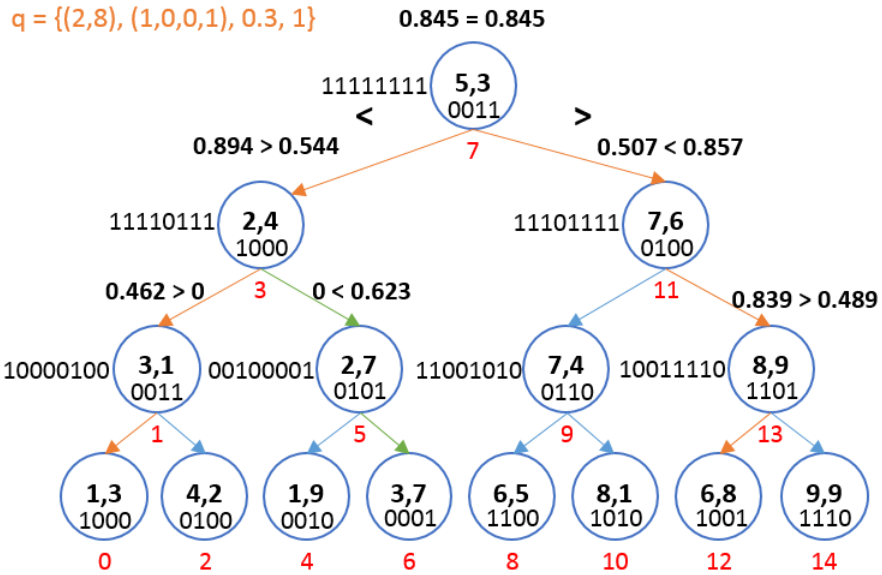


Figura 6.2: Ejemplo de consulta RkSKQ dado un q. Se muestra el recorrido de búsqueda sobre el árbol conceptual de la estructura cBiK. Cada nodo posee la información espacial, las palabras claves del punto y el bitmap resumen para una poda eficiente por izquierda o derecha.

Un ejemplo del funcionamiento de RkSKQ se puede ver en el árbol conceptual de la Figura 6.2. Para un mejor entendimiento de los cálculos se utilizará la Tabla 6.1 como apoyo. La figura

Tabla 6.1: Paso a paso en los cálculos de ejecución para la consulta RkSKQ de la Figura 6.2.

Paso	i	Pto.	Ptje. Espacial	Ptje. Textual	Ptje Total	Ptje. RI	Ptje. RD
1	7	(5,3)	0.484	0.5	0.495	0.845	0.845
2	3	(2,4)	0.646	0.5	0.544	0.894	0.544
3	1	(3,1)	0.374	0.5	0.462	0.462	0.000
4	0	(1,3)	0.549	0.5	0.515		
5	5	(2,7)	0.912	0.5	0.623	0.000	0.623
6	6	(3,7)	0.875	0.5	0.612		
7	11	(7,6)	0.523	0	0.000	0.507	0.857
8	13	(8,9)	0.462	1	0.839	0.839	0.489
9	12	(6,8)	0.646	1	0.894		

muestra una consulta q con cuatro parámetros, primero el punto a consultar $(2, 8)$, luego las palabras claves buscadas "Baño Privado" y "Wi-fi" (representados por su bitmap), el parámetro de preferencia $\alpha = 0{,}3$ (indica que el puntaje total será 30 % proximidad espacial y 70 % relevancia textual) y finalmente el valor de $k = 1$. Iniciando en el nodo raíz, se evalúa el puntaje de clasificación de sus resumenes para determinar el subespacio que posee mayor concentración de palabras claves y que está más cercano al punto de consulta. Como tanto el resumen izquierdo (RI) y el resumen derecho (RD) tienen el mismo puntaje, corresponderá revisar ambos subespacios, primero el izquierdo y luego el derecho. El resumen del punto $(2, 4)$ indica que el RI tiene mayor puntaje que el RD, en consecuencia se continúa con el punto $(3, 1)$. El RD del punto actual tiene ausencia de las palabras claves consultadas y por lo tanto su puntaje es cero y dado que el puntaje del RI es 0,462 se continúa el recorrido accediendo al punto $(1, 3)$. Este punto corresponde a un nodo hoja (no tiene resumen) por lo tanto se termina el recorrido y considera al punto como un candidato de solución.

Luego por cada punto al regreso de la recursividad se intenta mejorar la solución. Notar que hasta llegar al punto $(2, 4)$, la mejor solución corresponde al punto $(1, 3)$ con un puntaje de 0,515 y ya que el RD estima que hay mejores candidatos al tener un puntaje mayor al del mejor actual, inicia el recorrido en profundidad por ese lado, y en efecto, la mejor solución pasa a ser el punto $(2, 7)$. Una vez que se regresa al nodo raíz se continúa el recorrido por el subespacio derecho que había quedado pendiente. Finalmente, después de decidir qué subespacios elegir utilizando la estrategia explicada, se llega a la solución final con el punto $(6, 8)$ que posee un puntaje total de 0,894. Notar que el punto $(2, 7)$ del iKD-Tree es el más cercano al punto de consulta q, pero como se le ha dado una mayor importancia al componente textual ($\alpha = 0{,}3$), la solución final corresponde a un punto un poco más lejano, pero contiene una coincidencia mayor de las palabras claves buscadas.

Todo el proceso anteriormente visto para evaluar consultas RkSKQ se traduce en el Algoritmo 6.2.1. Los parámetros recibidos para el método recursivo son cinco, la variable q que posee la información de consulta con un atributo espacial $q.l$, uno textual $q.t$ y un atributo $q.k$ para determinar cuantos resultados se esperan obtener, *start* y *end* definen la zona del arreglo *nodes*

donde continuar el recorrido en el iKD-Tree; inicialmente $start = 0$ y $end = n - 1$. La variable h corresponde a un Min-Heap de tamaño k utilizado para mantener los puntos candidatos de solución, dichos puntos se encuentran organizados por el menor puntaje obtenido según la clasificación al punto de consulta. Después de la revisión del iKD-Tree la respuesta queda en el heap h. Notar que puede ocurrir que no sea posible conseguir de D los k objetos que satisfagan las palabras claves definidas en $q.t$ y por lo tanto, el número de objetos contenidos en h puede ser menor que k. Por último, la variable α corresponde al parámetro de preferencia entre el puntaje espacial y el textual, el que definirá la importancia que se le dará entre una medida y otra para obtener el puntaje final.

Algorithm 6.2.1 searchRkSQK(q, $start$, end, h, α)

```
1:  mid ← (start+end)/2
2:  spatialS ← spatialScore(q.l, mid)
3:  textualS ← textualScore(q.t, mid)
4:  totalScore ← totalScore(spatialS, textualS, α)
5:  if (start ≠ end) then
6:      textualScores ← summaryTextualScores(q.t, mid, start, end)
7:      score_LS ← totalScore(spatialS, textualScores_SL, α)
8:      score_RS ← totalScore(spatialS, textualScores_SR, α)
9:      if (score_LS > score_RS) then                          ▷ Busca por izquierda
10:         searchRkSQK(q, start, mid − 1, h, α)
11:         if goToR(h, score_RS, q.k) then
12:             searchRkSQK(q, mid + 1, end, h, α)
13:         end if
14:     else if (score_RS > score_LS) then                     ▷ Busca por derecha
15:         searchRkSQK(q, mid + 1, end, h, α)
16:         if goToL(h, score_LS, q.k) then
17:             searchRkSQK(q, start, mid − 1, h, α)
18:         end if
19:     else if (score_RS ≠ 0) then   ▷ Score igual pero ≠ a cero, se busca en ambos lados
20:         searchRkSQK(q, start, mid − 1, h, α)
21:         searchRkSQK(q, mid + 1, end, h, α)
22:     end if
23: end if
24: if (totalScore ≠ 0) then                  ▷ Existe al menos un término buscado
25:     updateCandidate(h, p, totalScore, q.k)
26: end if
```

El algoritmo inicia obteniendo la mediana de los valores de *start* y *end* (línea 6), correspondiente a la posición del punto a evaluar en el arreglo *nodes*. Luego en las líneas 2-4, se obtiene el puntaje espacial (*spatialScore*), el textual (*textalScore*) y el total (*totalScore*) del punto de consulta con respecto al punto accedido. El detalle de las funciones utilizadas en dicho proceso se explicarán más adelante.

Luego en las líneas 5-23 se evalúa el caso cuando el punto accedido corresponde a un nodo interno del árbol y es necesario determinar por cual subespacio se debe seguir el recorrido en profundidad. La línea 6 obtiene los puntajes textuales del resumen izquierdo ($textualScores_{LS}$) y derecho ($textualScores_{RS}$) para luego obtener los puntajes totales, del resumen izquierdo $score_{LS}$ (línea 7) y del resumen derecho $score_{RS}$ (línea 8) con respecto al punto visitado, lo que ayudará a decidir por dónde continuar el recorrido en base al puntaje que tiene cada subespacio.

En la línea 9-14, se resuelve el caso cuando en el subespacio izquierdo existe mayor concentración de coincidencias de palabras claves que en el subespacio derecho ($score_{LS} > score_{RS}$), por lo que se accede recursivamente a dicho subespacio en la línea 10. Al regreso de la recursividad, la función $goToR(.,.,.)$ determina si es necesario acceder al subespacio contrario para mejorar la solución de h. De resultar necesario este se accede en la línea 12. Análogamente, en las líneas 14-19 se revisa el caso cuando el subespacio derecho tiene un puntaje mayor al del izquierdo. En las líneas 19-22, se da el caso cuando los puntajes de ambos subespacios son iguales y por lo tanto, se deben recorrer recursivamente primero el izquierdo y luego el derecho.

Finalmente en las líneas 24-26 se resuelve el caso cuando el recorrido ha alcanzado un punto correspondiente a un nodo hoja, o cuando se está regresando de la recursividad y se debe considerar al punto accedido como un candidato para mejorar la solución.

Las funciones utilizadas en el algoritmo 6.2.1 y que no son triviales se explican a continuación:

spatialScore($q.l, pos$): La función devuelve el puntaje espacial de dos puntos. Los parámetros que se reciben son la ubicación del punto de consulta $q.l$ (latitud y longitud) junto con la posición del punto pos del arreglo $nodes$ (iKD-Tree) con el que se desea medir. La función retorna el resultado de la Fórmula 6.1.

$$\left(1 - \frac{euclideanDistance(q.l, nodes[pos].l)}{dmax}\right) \tag{6.1}$$

Siendo $euclideanDistance(,.,)$ una función básica que devuelve la distancia euclídea entre ambos puntos y $dmax$ una constante global con el valor de la máxima distancia entre dos puntos de D.

textualScore($q.t, pos$): La función devuelve el puntaje textual entre dos puntos. El proceso se visualiza en el Algoritmo 6.2.2 . Los parámetros recibidos corresponden a las palabras claves $q.t$ del punto de consulta y la posición pos del punto en el arreglo $nodes$ (iKD-Tree) con el que se desea evaluar.

Dado que el modelo utilizado en ésta investigación asigna a todos los términos la misma importancia, basta con dividir en partes iguales el puntaje total según la cantidad de palabras claves que se consultan y luego, sumar las fracciones de los términos coincidentes. Lo anterior se verifica en el ciclo de la línea 4, que realiza tantas iteraciones como palabras claves posea $q.t$, luego en la línea 5 se revisa si la palabra clave consultada existe en el punto, si es así, se acumulan las coincidencias en la variable s para finalmente retornar el puntaje textual en la línea 10 .

Algorithm 6.2.2 textualScore($q.t$, pos)

```
1: startIndexK ← pos * m
2: s ← 0
3: i ← 0
4: while (i < q.t.size()) do
5:     if (BK[(startIndexK + q.t[i])] = 1) then
6:         s++
7:     end if
8:     i++
9: end while
10: return s / q.t.size()
```

totalScore($spatialScore, textualScore, \alpha$)**:** El Algoritmo 6.2.3 muestra la función que retorna el puntaje total de un punto en base a los puntajes parciales recibidos (el espacial y el textual) y el parámetro de preferencia para determinar la proporción de cálculo a utilizar. Notar que si el puntaje textual recibido es cero, su puntaje total también lo será. Lo anterior es consecuencia de que se necesita al menos una palabra clave coincidente para que el punto sea candidato de solución, si no la tiene, entonces su puntaje se descartará totalmente.

Algorithm 6.2.3 totalScore($spatialScore$, $textualScore$, α)

```
1: return textualScore == 0 ? 0 : ((α · spatialScore) + (1 − α) · (textualScore))
```

summaryTextScores($q.t, pos, start, end$) El Algoritmo 6.2.4 muestra la función que retorna un objeto con dos atributos, el puntaje textual del resumen izquierdo $textualScores_{LS}$ y el del resumen derecho $textualScores_{RS}$. La idea general es asignárles un puntaje de clasificación, revisando la existencia de las palabras claves consultadas $q.t$ en los diferentes subespacios haciendo uso de los resumenes de cada nodo interno. En la línea 3 del algoritmo, se verifica mediante el acceso a BM que el punto evaluado (dado por la variable *pos*) sea un nodo interno del iKD-Tree. En las líneas 4-22 se revisan los 3 casos existentes cuando el subarreglo recibido dado por *start* y *end* es:

- **Nodo interno con hijo derecho hoja:** Caso cuando el nodo tiene un iRL (líneas 4-6), por lo tanto, su resumen local debe ser obtenido mediante el BK y calcular el puntaje mediante la función $textualScore(q.t, end)$ explicada anteriormente. Cómo no tiene hijo izquierdo, solo se obtiene el puntaje del hijo derecho (línea 5).
- **Nodo interno con ambos hijos hoja:** En este caso el nodo tiene ambos hijos (líneas 6-9) e igual que el caso anterior los resumenes corresponden a iRL's, en consecuencia, se obtiene el puntaje mediante la función $textualScore(.,.)$ calculando el puntaje textual del resumen izquierdo (línea 7) y el resumen derecho (línea 8).

- **Nodo interno con al menos un hijo nodo interno:** En este caso los resúmenes del nodo evaluado son eRL's (líneas 9-22) y para recuperar los resumenes se deberá acceder directamente al bitmap BR. El proceso para revisar las palabras claves existentes en cada resumen es similar al proceso de la función $textualScore(.,.)$ del Algortimo 6.2.3. Primero en la línea 10, se obtiene mediante la función $getBRIndex(.)$ (ver Algoritmo 5.4.4) la posición del primer bit del rango de posiciones de BR correspondientes a pos. En el ciclo de las líneas 13-21 se realizan tantas iteraciones como palabras claves tenga la consulta, en cada ciclo se revisa si existe coincidencia de las palabras claves consultadas en el subespacio izquierdo (líneas 14-16) y derecho (líneas 17-19) a la vez, asignándoles el puntaje correspondiente. Finalmente el puntaje textual quedará en el objeto $textualScores$ para ser retornado.

Algorithm 6.2.4 summaryTextualScores($q.t$, pos, $start$, end)

```
1: textualScores_LS ← 0
2: textualScores_RS ← 0
3: if (BM[pos] = 1) then
4:     if (end = start + 1) then                          ▷ Es un iRL
5:         textualScores_RS ← textualScore(q.t, end)
6:     else if (end = start + 2) then                     ▷ Ambos son un iRL
7:         textualScores_LS ← textualScore(q.t, start)
8:         textualScores_RS ← textualScore(q.t, end)
9:     else if (end > start + 2) then                     ▷ Ambos son eRL
10:        posInicial ← getBRIndex(pos)
11:        n_keys ← q.t.size()
12:        i ← 0
13:        while (i < n_keys) do
14:            if (BR[(posInicial + q.t[i])] = 1) then    ▷ Resumen izquierdo
15:                textualScores_LS+ = 1/n_keys
16:            end if
17:            if (BR[(posInicial + m + q.t[i])] = 1) then   ▷ Resumen derecho
18:                textualScores_RS+ = 1/n_keys
19:            end if
20:            i++
21:        end while
22:    end if
23: end if
24: return textualScores
```

updateCandidate($heap, p, totalScore, q.k$) Esta función se utiliza para actualizar con el punto p los candidatos del $heap$. Se asume que p es también un candidato, es decir contiene al menos una de las palabras claves de la consulta. La función básicamente decide si insertar p al $heap$. Para ello verifica si el tamaño del heap es menor que k, si es así, p es insertado. En caso contrario,

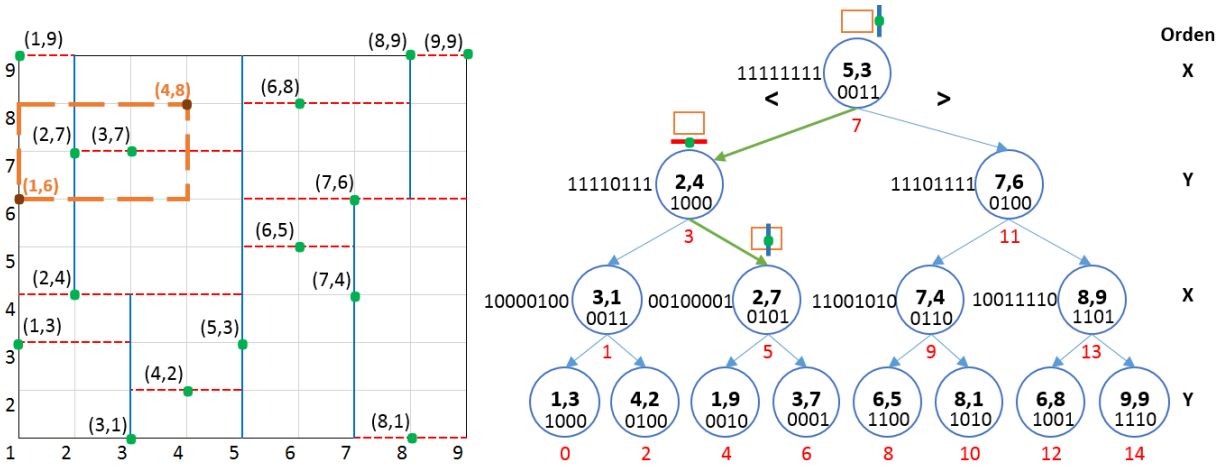
se verifica si el puntaje total de p es mayor que el puntaje total del punto de la raíz de *heap*. Si es así, se elimina el punto de la raíz y se inserta p. Debido a que el heap corresponde a un Min-Heap, los puntos con mayores puntajes irán quedando dentro del heap.

goToL$(heap, score_{LS}, q.k)$ **y goToR**$(heap, score_{RS}, q.k)$**:** Se utilizan para decidir, al regresar desde la recursividad en los nodos internos del iKD-Tree, si es necesario continuar el recorrido a partir del punto visitado por el subespacio contrario al ya accedido. Si el tamaño del heap es menor que k y el puntaje del resumen recibido es mayor a cero (si es cero significa que no posee ningún término buscado), el recorrido se realiza de todas maneras. O también, cuando el puntaje del resumen recibido es mayor al puntaje total del punto de la raíz del heap, ya que se estima que existan candidatos con mejores puntajes en el subespacio no explorado por la recursividad.

6.3. Evaluación de consultas bRS-SKQ con cBiK

Distinta a las consultas BkSKQ y RkSKQ que buscan los k objetos más cercanos dado un punto de consulta $q.l$, la consulta bRS-SKQ recibe dos puntos $q.l_1$ y $q.l_2$ para formar una región de consulta, por lo tanto, el resultado estará dado por todos los puntos que posean la coincidencia total de las palabras claves buscadas y que estén dentro del área mencionada.

La idea general es recorrer en profundidad el árbol iniciando desde el nodo raíz y en base al valor de la coordenada del eje de partición del punto, se determina si el recorrido debe ser realizado por izquierda o derecha para lograr encontrar el rango especificado. El acceso al hijo correspondiente se realiza siempre y cuando el RL informe que sí existe la totalidad de las palabras claves buscadas en dicho subespacio, en caso contrario no se continúa el recorrido. La navegación termina cuando se llega a un nodo hoja, luego en el regreso de la recursividad se revisa si los nodos visitados están contenidos en dicho región y si contienen la coincidencia exacta de las palabras claves solicitadas, de ser así, se agrega el punto a una lista que guardará todos los puntos de solución.



(a) Representación espacial.

(b) Recorrido en el iKD-Tree.

Figura 6.3: Ejemplo de consulta bRS-SKQ con $q = \{(1,6),(4,8),(0100)\}$.

Para mayor comprensión, se ejemplifica la consulta bRS-SKQ usando la Figura 6.3. Notar que se indica una consulta q con tres parámetros, siendo los dos primeros los puntos extremos de la región espacial que puede visualizarse en la Figura 6.3-a), el último parámetro corresponde a la palabra clave "Calefacción" que es representado por su bitmap 0100. El recorrido se representa en la Figura 6.3-b) y se comienza evaluando el nodo raíz. Debido a que el punto se encuentra a la derecha de la región buscada, se debe acceder a su subespacio izquierdo (el RL indica presencia de todas las palabras claves). De forma similar, el punto $(2,4)$ se ubica bajo la región y por lo tanto se accede al subespacio superior. Luego el eje de partición del punto $(2,7)$ intersecta la región buscada por lo que pueden existir puntos en ambos subespacios y deberían ser accedidos. Lo anterior no ocurre ya que el RL indica que no hay presencia de la palabra buscada en ninguno

de los dos subespacios hijos, se termina el recorrido y comienza el regreso de la recursividad revisando si los puntos visitados están contenidos por la región especificada. El resultado final solo comprenderá el punto $(2, 7)$. Notar que el punto $(3, 7)$ está dentro del rango pero no posee la palabra clave buscada.

El algoritmo 6.3.1 muestra el procedimiento para buscar el bRS-SKQ. La función recibe 4 parámetros, q contiene la información de consulta con atributos $q.l_1$ y $q.l_2$ los puntos de consulta y $q.t$ las palabras claves buscadas. Antes de realizar la consulta, las coordenadas son ordenadas para que $q.l_1$ sea el punto inferior izquierdo y $q.l_2$ el punto superior derecho de la región determinada. Las variables *start* y *end* definen la zona del arreglo *nodes* donde continuar el recorrido en el iKD-Tree; inicialmente $start = 0$ y $end = n - 1$. Finalmente *depth* indica la profundidad del iKD-Tree y con su valor se decide la dirección de las particiones de los subespacios. Notar que existe una lista *result* como variable global y es donde se van almacenando los objetos que cumple las condiciones espaciales y textuales solicitadas.

Algorithm 6.3.1 rangeSearching(q, *start*, *end*, *depth*)

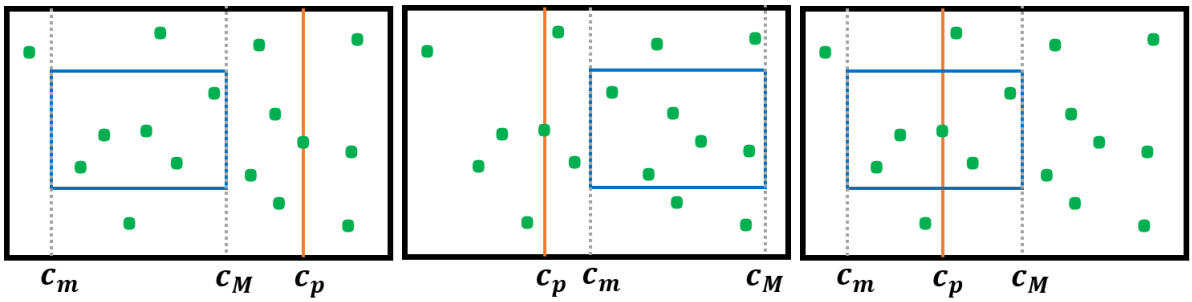
```
1:  mid ← ⌊(start+end)/2⌋
2:  p ← nodes[mid]
3:  if (start ≠ end) then
4:      c_M ← getCoordinate(depth, q.l_2)          ▷ Coordenada superior derecha
5:      c_m ← getCoordinate(depth, q.l_1)          ▷ Coordenada inferior izquierda
6:      c_p ← getCoordinate(depth, p)
7:      if (c_M ≤ c_p)) then                       ▷ Busca hacia la izquierda
8:          if checkLS(q.t, mid) then
9:              rangeSearching(q, start, mid − 1, depth + 1)
10:         end if
11:     else if (c_m > c_p) then                   ▷ Busca hacia la derecha
12:         if checkRS(q.t, mid) then
13:             rangeSearching(q, mid + 1, end, depth + 1)
14:         end if
15:     else                                       ▷ Busca hacia ambos lados
16:         if checkLS(q.t, mid) then
17:             rangeSearching(q, start, mid − 1, depth + 1)
18:         end if
19:         if checkRS(q.t, mid) then
20:             rangeSearching(q, mid + 1, end, depth + 1)
21:         end if
22:     end if
23: end if
24: if containsPoint(q, p) ∧ checkKeywords(q.t, mid) then
25:     result.insert(p)
26: end if
```

El algoritmo comienza obteniendo el valor de la mediana de los valores de *start* y *end* (línea

1), dicho valor corresponde a la posición del punto accedido en el arreglo *nodes* (*i*KD-Tree). Luego se recupera al punto y se guarda en la variable p.

En las líneas 3-23 se evalúa el caso cuando el punto revisado es un nodo interno y por lo tanto, se recorre recursivamente en profundidad. Primero, en la líneas 4-6 se obtiene el valor de la coordenada según sea la partición a revisar, c_m y c_M corresponden a los valores del punto inferior izquierdo y superior derecho de la región y c_p el valor propio del punto evaluado. Para determinar por cual subespacio continuar la búsqueda, se revisan los tres casos siguientes:

- **Caso 1:** Cuando $c_M \leq c_p$, la partición del punto se ubica a la derecha (o arriba) de la región de consulta (línea 7), la búsqueda debe seguir por el subespacio izquierdo del punto.
- **Caso 2:** Cuando $c_m > c_p$, la partición del punto se ubica a la izquierda (o abajo) de la región de consulta (línea 11), la búsqueda debe seguir por el subespacio derecho del punto.
- **Caso 3:** En otro caso (línea 15), la partición del punto se encuentra intersectando la región de consulta, por lo tanto, la búsqueda debe continuar por ambos subespacios.



(a) Caso 1. (b) Caso 2. (c) Caso 3.

Figura 6.4: Ejemplo de los diferentes casos en la evaluación del punto con respecto al rango espacial en la consulta *b*RS-SKQ.

Notar que en todos los casos, para permitir hacer la llamada recursiva se comprueba previamente que el subespacio posea las palabras claves solicitadas haciendo uso de las funciones $checkLS(.,.)$ y $checkRS(.,.)$ según sea el caso, si las contiene, se continúa el recorrido en profundidad ya que existen puntos candidatos dentro del rango de posiciones consultado.

Finalmente, en las líneas 24-26 se evalúa si el punto está contenido en la región de consulta con la función $containsPoint(q, p)$, además con la función $checkKeywords(q.t, mid)$ se determina si las palabras claves buscadas coinciden en su totalidad con las palabras claves del punto, si es así, el punto es agregado a la lista $result$.

Capítulo 7

Experimentación

En este capítulo se describe la evaluación experimental realizada sobre los tres algoritmos implementados en la estructura cBiK, detallados anteriormente en el Capítulo 6 y que corresponden a las consultas BkSKQ, RkSKQ y bRS-SKQ. Por lo anterior, se describe el entorno de pruebas (lenguaje de programación, hardware, sistema operativo, etc.), las métricas y los conjuntos de datos utilizados para realizar los diferentes experimentos.

Finalmente, se muestra una serie de resultados que miden el rendimiento de cBiK y de las consultas ya mencionadas en cuanto al espacio de almacenamiento requerido por los índices revisados y sus tiempos de ejecución.

7.1. Entorno de Pruebas

La propuesta de esta investigación se compara con la estructura S2I [50]. A pesar de que S2I fue originalmente diseñado para responder consultas RkSKQ, es posible modificar fácilmente su algoritmo para lograr responder consultas BkSKQ y bRS-SKQ. Su elección ha sido fundamentalmente por el trabajo experimental de [12], que realiza una comparativa de 12 índices espacio-textuales y que de acuerdo a los resultados, S2I es uno de las mejores en general para responder los tres tipos de consulta. Por lo anterior, si cBiK logra mejorar los resultados de S2I, por transitividad es posile decir que nuestra estructura también es mejor que las otros índices revisados.

En la evaluación se asume que el índice S2I, para los diferentes conjuntos de datos espacio-textuales, reside en memoria secundaria y por lo tanto, el procesamiento de las consultas involucra varias operaciones de entrada/salida. Este escenario permite comparar cuánto mejoran los tiempos de respuestas, almacenando los conjuntos de datos espacio-textuales directamente en memoria de manera compacta versus almacenarlos en memoria secundaria y procesarlos desde allí.

Dado que se usa una implementación en Java de S2I[1] y esta propuesta fue implementada en C++, existe una diferencia favorable a cBiK en el tiempo de ejecución producto del lenguaje de programación pero se considera que es relativamente baja en comparación a los costos de E/S y por lo tanto no afectará las conclusiones de la investigación. Ambos índices se ejecutaron bajo el sistema operativo Ubuntu 16.04 LTS de 64 bits en un servidor con procesador Intel Core i5, velocidad de 3.4 Mhz, memoria RAM de 8GB y un disco duro SATA de 1TB.

7.2. Conjuntos de Datos

En los experimentos se usaron dos conjuntos de datos espacio-textuales[2] (utilizados en los experimentos en [11, 12, 63] y que considera las coordenadas y luego las palabras clave, todo separado por espacios en blanco). El primero considera $1,1$ millones de puntos de la vida real recopilados en Foursquare y que corresponden a Puntos de Interés (POI) de todo el mundo que poseen una ubicación y un texto descriptivo. El segundo conjunto de datos pertenece a datos reales generados por Twitter y que posee 20 millones de tweets geolocalizados. A consecuencia de no poder procesar más de 5 millones de objetos con el índice S2I debido a la gran cantidad de espacio utilizado en su construcción, este último conjunto de datos se subdivide en 3 dataset más pequeños para realizar los experimentos, especificamente con el primer millón, luego los primeros 3 millones y finalmente los primeros 5 millones de objetos del dataset de Twitter.

Las características de cada conjunto de datos se muestran en la Tabla 7.1.

7.3. Generación de las consultas

Las consultas BkSKQ y RkSKQ se generaron de la siguiente manera. Para cada conjunto de datos espacio-textual de la Tabla 7.1 se generaron 5 conjuntos con 1.000 consultas. Todas las

[1] El código del índice S2I fue facilitado por su autor.
[2] Disponibles en http://www.ntu.edu.sg/home/gaocong/datacode.htm

Tabla 7.1: Conjuntos de datos espacio-textuales utilizados en los experimentos

Datasets	$n = \|D\|$ **Millones**	$\bar{x}$ **palabras por objeto**	$m =$ **palabras diferentes**	**Total palabras**
POI's	1,1	4,00	261.212	4.389.929
Twitter1M	1,0	4,34	283.806	4.342.238
Twitter3M	3,0	4,28	585.624	12.848.921
Twitter5M	5,0	4,27	825.971	21.338.978

consultas de un mismo conjunto consideran el mismo número de palabras claves y que corresponde a 1, 2, 3, 4 y 5, esta cantidad obedece en promedio al número de palabras claves que se consultan en la vida real [11, 12, 32, 63, 65] y por lo tanto, indicar más de 5 términos en una consulta no se justifica. Las coordenadas de los puntos se generan de manera aleatoria, para x en un rango $[-90, 90]$ y para y en $[-180, 180]$. Las palabras claves de la consulta se obtienen eligiendo de manera aleatoria un objeto del conjunto que contenga igual o mayor número de palabras claves que la cantidad l definida para la consulta. Luego l palabras del objeto escogido, seleccionadas al azar, configuran la lista de palabras claves de la consulta. Esta forma de generar las consultas asegura que siempre habrá al menos un objeto que la satisfaga.

Para bRS-SKQ las consultas se generaron siguiendo la misma estrategia anterior en cuanto a la componente textual se refiere. Para la componente espacial, en vez de generar un punto es necesario conformar una región espacial de consulta. Para ello, se selecciona aleatoriamente un objeto espacial del conjunto y se generan dos puntos equidistantes con respecto a su coordenada geográfica, uno inferior izquierdo q_1 y otro superior derecho q_2. Lo anterior sugiere una región espacial cuadrada y que como centro tiene al punto seleccionado. Se considera la distancia de la región espacial como la distancia entre los puntos q_1 y q_2 (hipotenusa del cuadrado). Para cada conjunto de datos, se contempla generar distancias de 1, 2, 5, 10 y 20 Kilómetros.

La Tabla 7.2 muestra información de las regiones de consultas generadas para cada conjunto de datos, por ejemplo para POI's, se indica que la distancia entre los dos puntos más lejanos existentes es de 18.909,6 Km. (región total) lo que hace referencia a que los puntos de interés están repartidos por casi la mitad del planeta. También se muestra el porcentaje de la región espacial generada con respecto a la región total del conjunto de datos, siendo solo de un $0,005\,\%$ la región de consulta de 1Km.

7.4. Resultados experimentales

A continuación se muestran los resultados de almacenamiento requerido y tiempos de ejecución obtenidos en la etapa de experimentación.

En cuanto al almacenamiento, se reflejan los resultados de realizar la compactación de los bitmaps para representar las palabras claves y como consecuencia, la reducción notoria en la totalidad del almacenamiento utilizado por cBiK.

Tabla 7.2: Características de las regiones de consultas generadas para cada conjuntos de datos de prueba.

Dataset	**Máx. distancia en Km entre dos puntos**	**% de la región respecto del área total**				
		1Km	**2Km**	**5Km**	**10Km**	**20Km**
POI's	18.909,6	0,005	0,011	0,026	0,053	0,106
Twitter1M	8.867,5	0,011	0,023	0,056	0,113	0,226
Twitter3M	8.931,26	0,011	0,022	0,056	0,112	0,224
Twitter5M	10.563,1	0,009	0,019	0,047	0,095	0,189

Referente a los tiempos de ejecución, los tres algoritmos de cBiK presentan mejoras notables, superando entre 1 y 3 ordenes de magnitud al índice S2I. Tales resultados muestran los beneficios de evitar las operaciones de entrada/salida al usar una estructura de datos compacta y trabajar directamente en memoria principal.

7.4.1. Almacenamiento cBiK

En la Tabla 7.3, se muestra el almacenamiento requerido por el conjunto original y por los índices S2I y cBiK. Es posible apreciar que esta propuesta requiere en promedio de solo un 38 % del almacenamiento requerido por el índice S2I.

Tabla 7.3: Almacenamiento requerido por ambas estructuras en MB.

Dataset	**Original (Mb)**	**S2I (Mb)**	**cBiK (Mb)**					
			Hashing	**Points**	**BM**	**BK**	**BR**	**Total**
POI's	81,9	246	3,09	36,2	0,3	8,6	48,5	96,7
Twitter1M	62,8	281	3,45	24,6	0,2	9,4	63,2	100,9
Twitter3M	191,0	845	7,58	73,9	0,7	29,6	208,0	319,8
Twitter5M	320,0	1.403	10,9	123,0	1,2	50,0	376,0	561,1

7.4.2. Tiempo de ejecución BkSKQ

En esta sección se presentan los resultados de cBiK y S2I con diferentes configuraciones para medir el rendimiento de procesamiento de las consultas BkSKQ.

Variando el número de palabras claves: La Figura 7.1, ilustra el efecto de variar el número de palabras claves consultadas entre 1 y 5 para los 4 conjuntos de datos, (a) POI's, (b) Tweets

1M, (*c*) Tweets 3M y (*d*) Tweets 5M respectivamente. En este grupo de experimentos, se ha establecido $k = 5$ para todas las consultas.

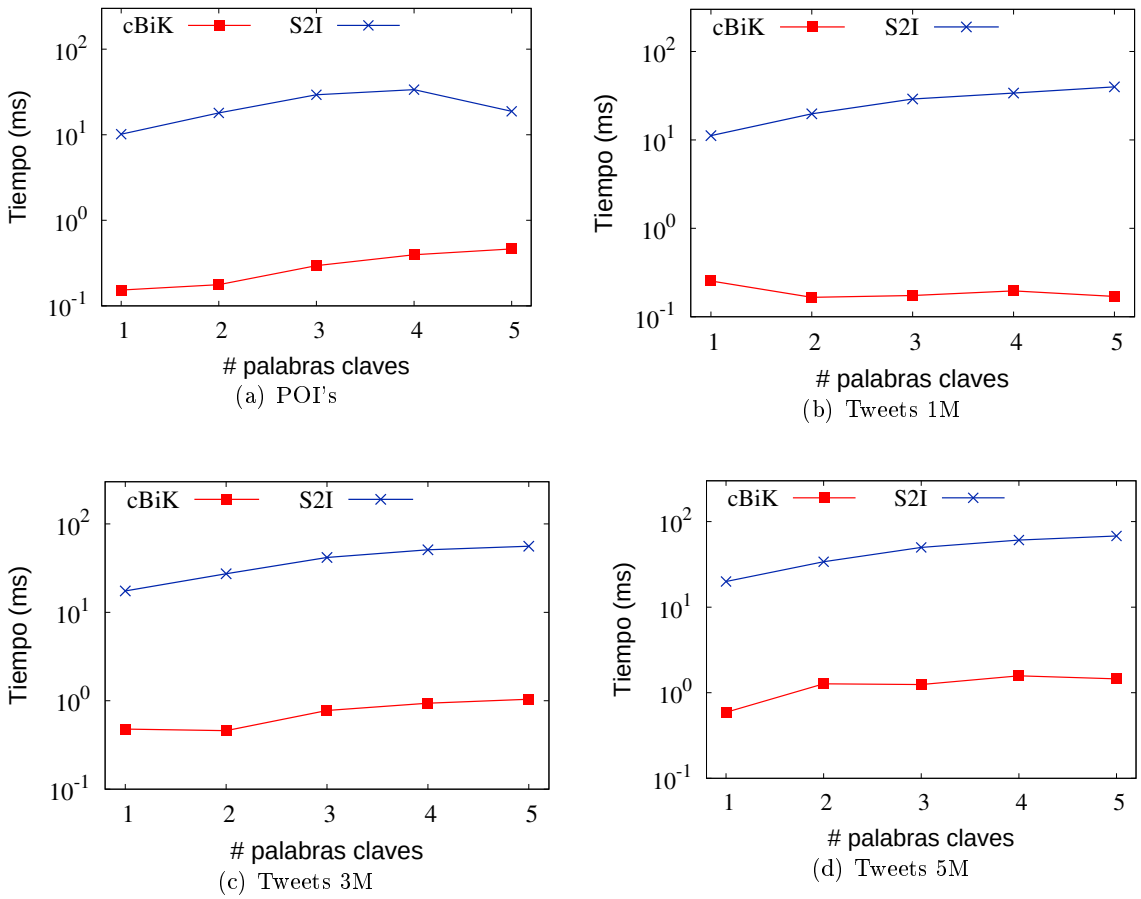


Figura 7.1: Resultados variando el número de palabras claves solicitadas, con $k = 5$ para la consulta BkSKQ.

Es posible apreciar que *c*B*i*K supera a S2I en los cuatro conjuntos de datos estudiados. En promedio *c*B*i*K requiere aproximadamente de un 1,7 % del tiempo requerido por S2I. Se refleja además, que los tiempos de ejecución de ambas estructuras tienden a aumentar en los cuatro conjuntos de datos a medida que el número de palabras claves de la consulta aumenta. De forma similar, los tiempos de ejecución de las consultas para un número de palabras clave es cada vez mayor al aumentar el tamaño de los conjuntos de datos.

Para la Figura 7.1-a, el tiempo de ejecución del índice S2I tiene una baja considerable en la consulta con 5 palabras claves. Lo anterior es consecuencia de la poca cantidad de palabras claves que contiene cada objeto en el conjunto de datos, es decir, los objetos con 5 o más palabras claves son mínimos y por lo tanto, S2I no necesita evaluar tantos objetos para entregar su resultado.

Variando k: La Figura 7.2 muestra el efecto de variar los k resultados solicitados entre 1 y 20, para los conjuntos de datos (a) POI's, (b) Tweets 1M, (c) Tweets 3M y (d) Tweets 5M respectivamente. Para este grupo de experimentos se ha fijado a 3 la cantidad de palabras claves en todas las consultas.

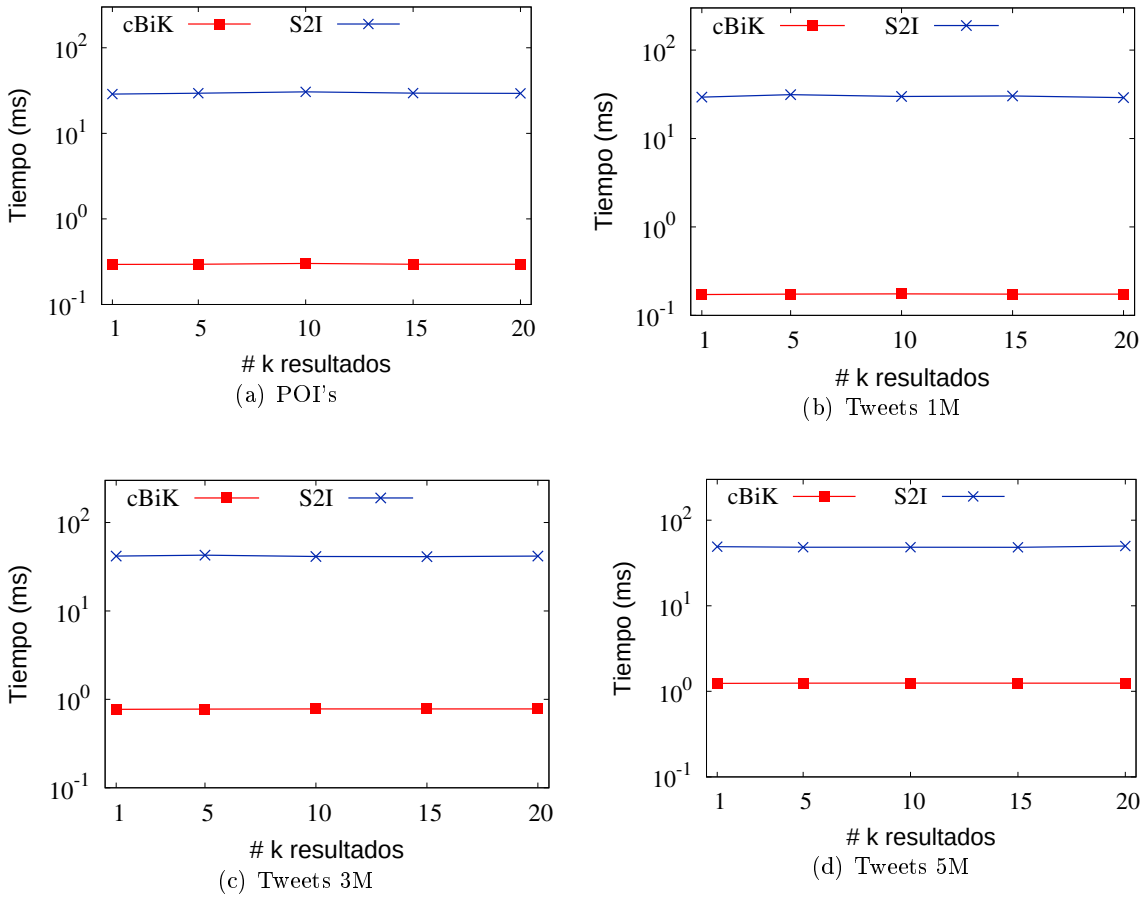


Figura 7.2: Resultados variando el número de k, con 3 palabras claves consultadas.

Es posible determinar que nuevamente cBiK supera a S2I. Además, cBiK requiere solo de un 1,5 % en promedio, del tiempo requerido por S2I para los conjuntos de datos evaluados. Notar que ambas propuestas no se ven mayormente afectadas en los tiempos de ejecución, entregando valores similares conforme varía el valor de k. También se muestra que al aumentar la cardinalidad de los conjuntos de datos, el tiempo de ejecución para entregar el mismo valor de k aumenta.

7.4.3. Tiempo de ejecución RkSKQ

En esta sección se evalúan los algoritmos para la consulta RkSKQ. A continuación se muestran tres grupos de experimentos con diferentes configuraciones con el propósito de conocer sus

rendimientos en tiempos de ejecución.

Como la nueva métrica de evaluación de RkSKQ combina simultaneamente el puntaje de clasificación espacial y textual, se agrega el atributo α al grupo de experimentos realizados.

Variando el número de palabras claves: La Figura 7.3, ilustra el efecto de variar el número de palabras claves consultadas entre 1 y 5 para los 4 conjuntos de datos, (a) POI's, (b) Tweets 1M, (c) Tweets 3M y (d) Tweets 5M respectivamente. En este grupo de experimentos, se ha establecido $k = 5$ y $\alpha = 0{,}3$ para todas las consultas.

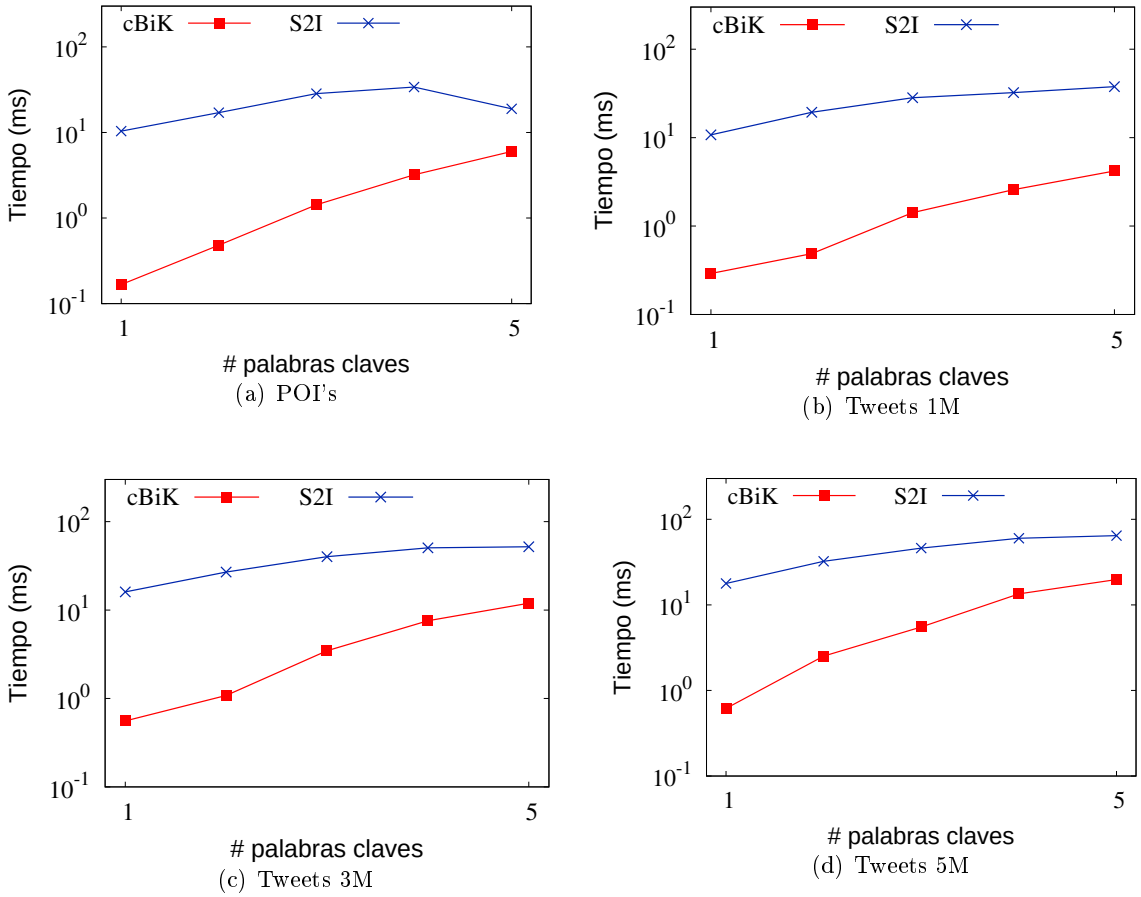


Figura 7.3: Resultados variando el número de palabras claves, con $k = 5$ y $\alpha = 0,3$.

En este experiemento, cBiK requiere solo de un 12,4 % en promedio, del tiempo requerido por S2I. Pero existen varias consideraciones a mencionar con respecto a los resultados:

- Notar que los tiempos de S2I para este tipo de experimento en las consultas BkSKQ y RkSKQ son similares e incluso para este último, levemente inferior dado que no es necesario revisar si existen todas las palabras claves solicitadas del conjunto candidato.

- Los tiempos de ejecución de la consulta RkSKQ para el índice cBiK son muy superiores con respecto al mismo experimento en BkSKQ. Aquello es provocado por la flexibilidad de permitir ausencias de palabras claves requeridas en los resultados, abriendo un abanico enorme de posibilidades (las podas no son tan efectivas) y los cálculos para determinar los k mejores objetos aumentan.
- El comportamiento de la consulta con 5 palabras claves en la Figura 7.3-a es idéntico al de la Figura 7.1-a, por lo que su explicación es la misma.

Variando k: La Figura 7.4 muestra el efecto de variar k resultados entre 1 y 20, para los conjuntos de datos (a) POI's, (b) Tweets 1M, (c) Tweets 3M y (d) Tweets 5M respectivamente. Para este grupo de experimentos se ha fijado a 3 la cantidad de palabras claves y $\alpha = 0{,}3$ en todas las consultas.

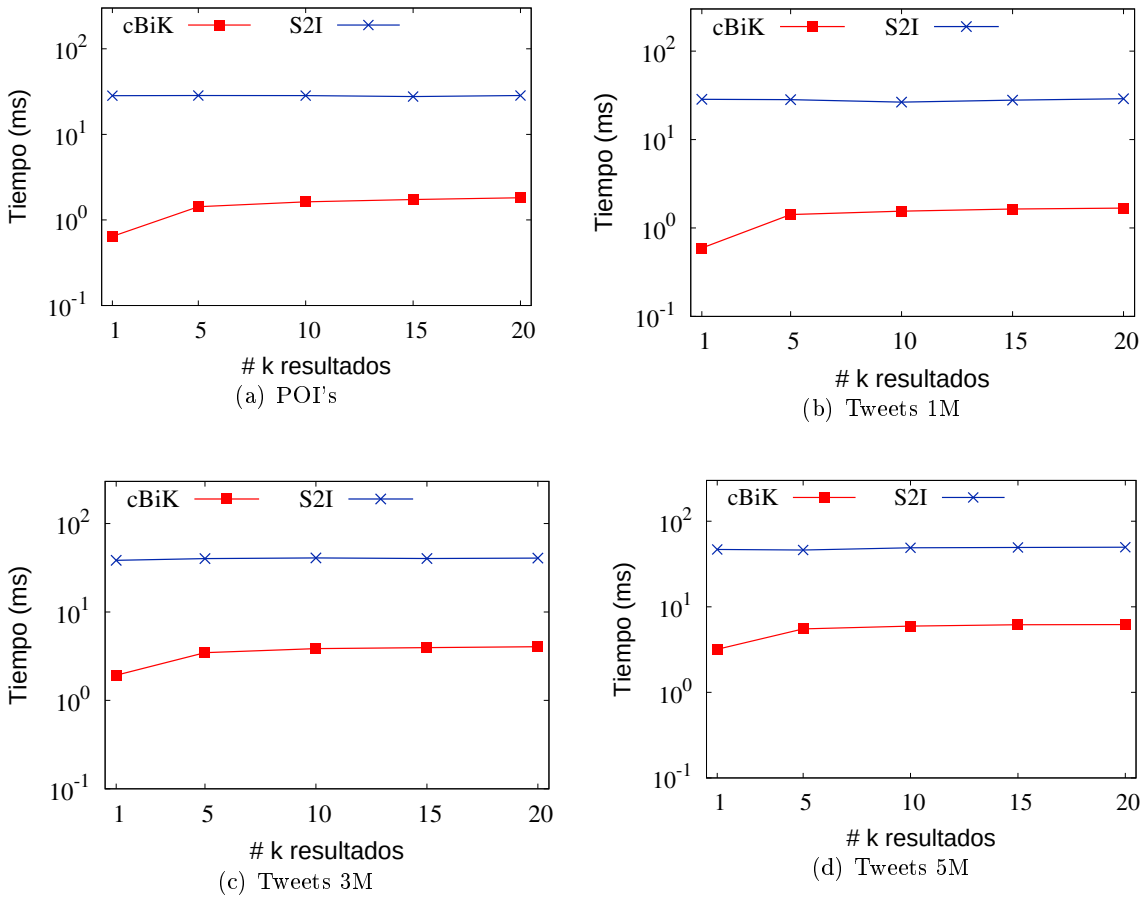


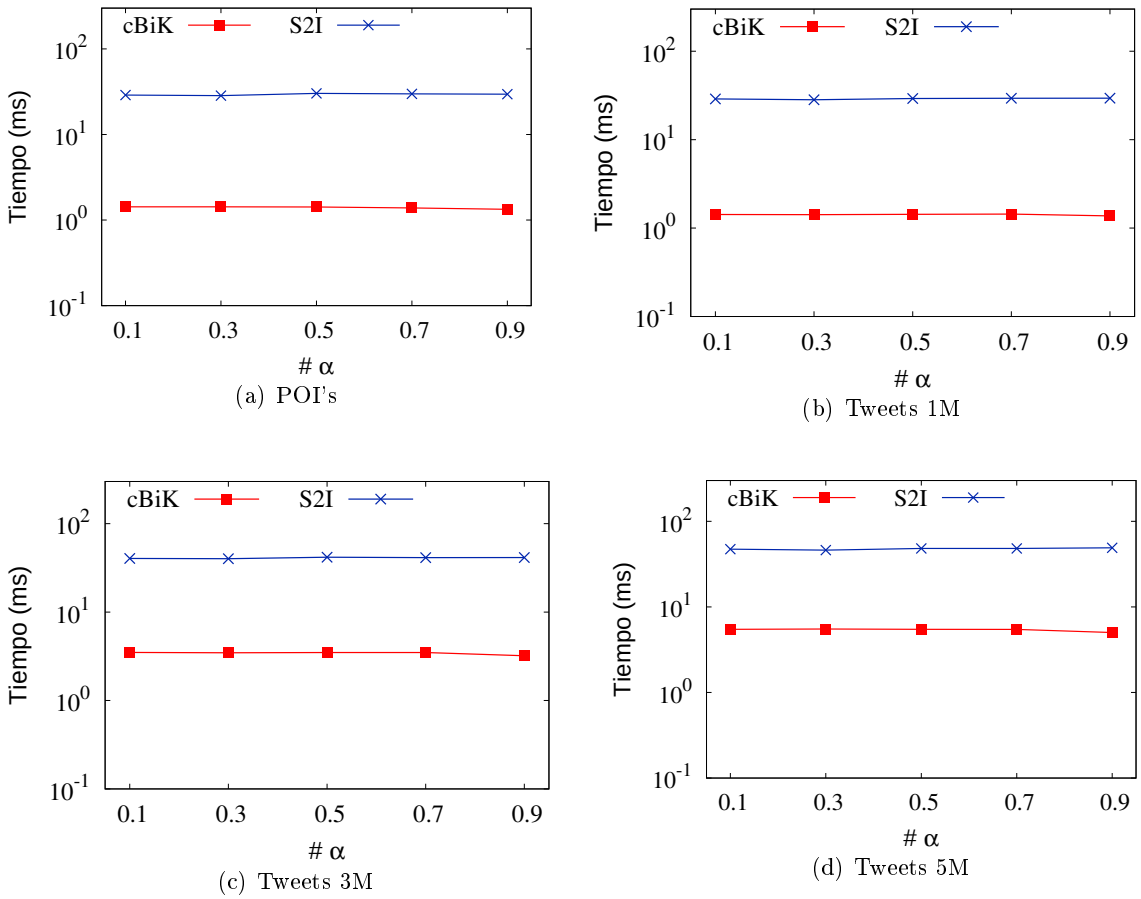
Figura 7.4: Resultados variando el número de k, con $\alpha = 0,3$ y 3 palabras claves consultadas.

En este experimento, al igual que el realizado para la consulta BkSKQ, los tiempos de ejecución

se mantienen relativamente constantes a medida que varía el valor de los k resultados esperados. Por otro lado, se refleja que los tiempos de ejecución aumentan al ejecutar la consulta en conjuntos de datos cada vez más grandes.

Finalmente, mediante la comparativa de los resultados, es posible mencionar que cBiK requiere solo de un 7,5 % en promedio, del tiempo requerido por S2I para este tipo de experimento.

Variando α: La Figura 7.5 muestra el efecto de variar el parámetro de preferencia α entre 0,1 y 0,9, para los conjuntos de datos (a) POI's, (b) Tweets 1M, (c) Tweets 3M y (d) Tweets 5M respectivamente. Para este grupo de experimentos se ha fijado a 3 la cantidad de palabras claves y a $k = 5$ la cantidad de resultados esperados en todas las consultas.



(a) POI's (b) Tweets 1M

(c) Tweets 3M (d) Tweets 5M

Figura 7.5: Resultados variando el valor de α, con $k = 5$ y 3 palabras claves consultadas.

Los resultados experimentales muestran que los tiempos de ejecución se mantienen relativamente constantes a medida que el valor de α cambia. Notar que el índice S2I tiene una insinuación leve de incremento a medida que sube el valor de α, caso contrario ocurre con cBiK que manifiesta una leve tendencia a la disminuición en su tiempo de ejecución. En este experimento, cBiK

requiere solo de un 7,3 % en promedio, del tiempo requerido por S2I.

7.4.4. Tiempo de ejecución bRS-SKQ

En esta sección se evalúan los algoritmos para la consulta bRS-SKQ. A continuación se muestran dos grupos de experimentos con diferentes configuraciones con el propósito de conocer su rendimiento en tiempos de ejecución.

Variando el número de palabras claves: La Figura 7.6, ilustra el efecto de variar el número de palabras claves consultadas entre 1 y 5 para los 4 conjuntos de datos, (a) POI's, (b) Tweets 1M, (c) Tweets 3M y (d) Tweets 5M respectivamente. En este grupo de experimentos, se ha establecido la distancia de la región en $d = 10$ Km. para todas las consultas.

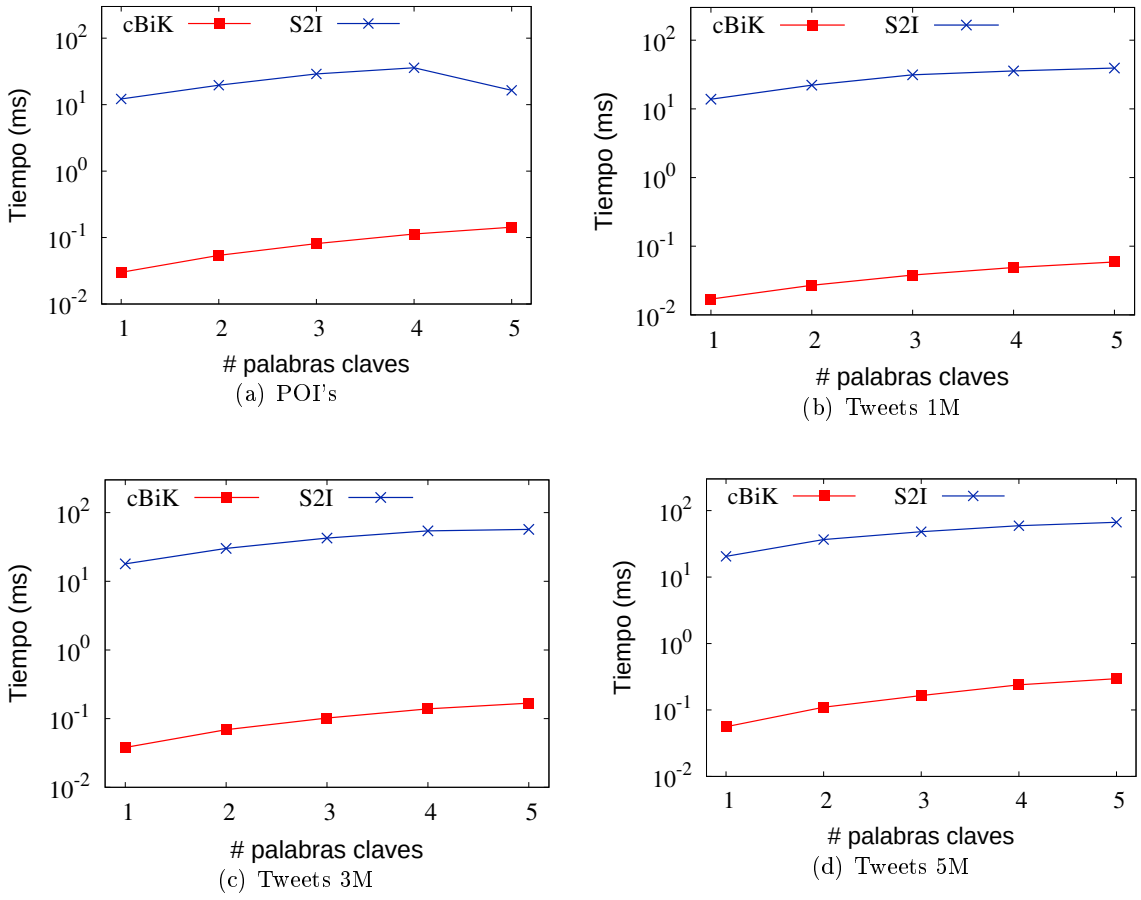


Figura 7.6: Resultados variando el número de palabras claves, con la distancia de la región de consulta en $d = 10$ Km.

Los resultados indican que cBiK requiere solo de un 0,3 % en promedio, del tiempo requerido

por S2I. Notar la tendencia incremental del índice S2I a medida que las palabras claves consultadas aumentan en un mismo conjunto de datos. Análogamente, *c*B*i*K muestra un patrón similar pero en comparación a S2I sus incrementos son prácticamente insignificantes. Como en todos los experimentos anteriores, al aumentar el tamaño del conjunto de datos para una consulta con un mismo número de palabras claves los tiempos de ejecución aumentan.

Finalmente, en la Figura 7.6-a se reproduce el mismo patrón, ya explicado, que las Figuras 7.1-a y 7.3-a, al disminuir considerablemente el tiempo de ejecución al consultar por 5 palabras claves.

Variando la distancia del rango de consulta: La Figura 7.7, ilustra el efecto de variar la distancia en Km. del rango espacial consultado entre 1 y 20 para los 4 conjuntos de datos, (*a*) POI's, (*b*) Tweets 1M, (*c*) Tweets 3M y (*d*) Tweets 5M respectivamente. En este grupo de experimentos, se ha establecido en 3 la cantidad de palabras claves para todas las consultas.

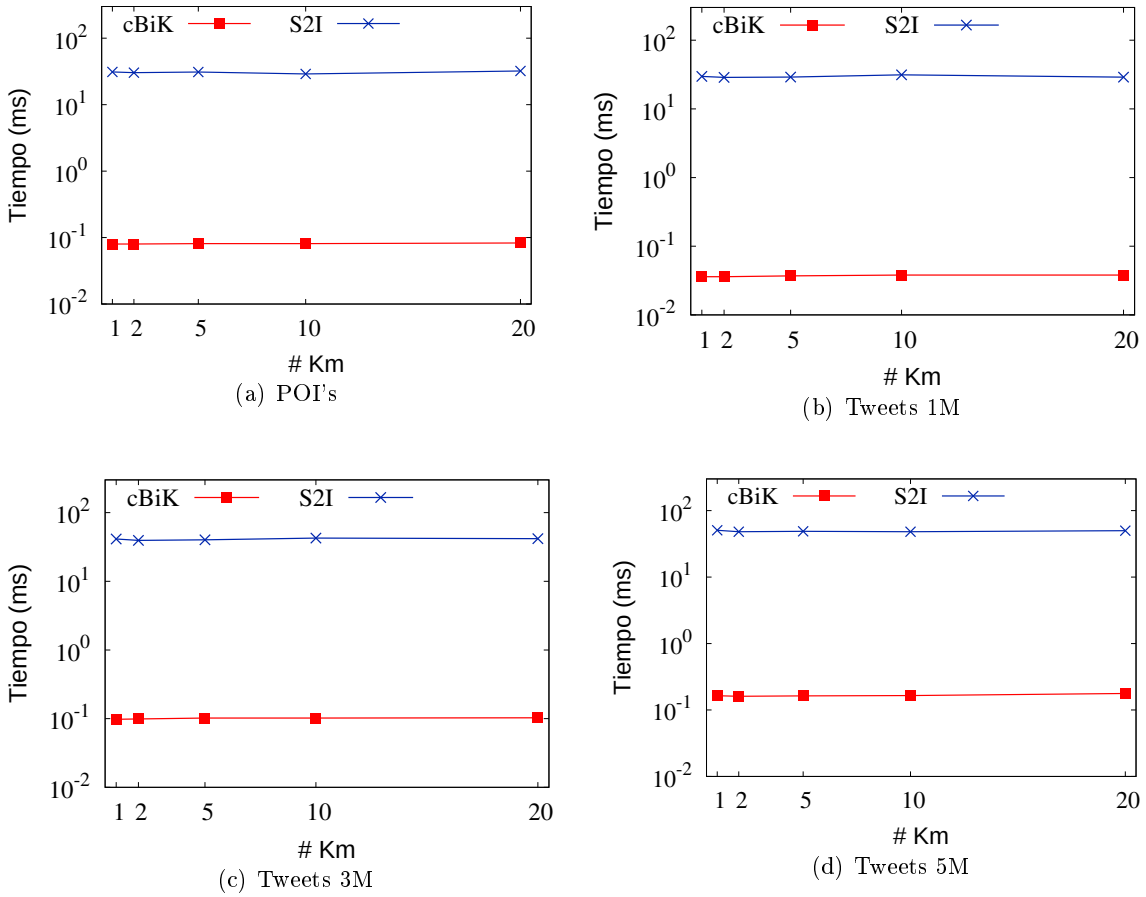


Figura 7.7: Resultados variando el largo de la diagonal (en Km) de la región de consulta.

Los resultados de *c*B*i*K para este experimento indican un leve incremento constante a medida

que la distancia de la región consultada era mayor. Muy similar para el índice S2I los tiempos muestran una relativa constancia en sus resultados ya que independiente del rango consultado, S2I elige la misma cantidad de objetos candidatos en base al número de palabras claves consultadas.

Por lo anterior, se muestra que *c*B*i*K requiere sólo de un 0,2 % en promedio, del tiempo requerido por S2I.

Parte IV

Conclusiones

Capítulo 8

Conclusiones y Trabajo Futuro

8.1. Conclusiones

En esta tesis se propone la estructura de datos compacta cBiK para representar conjuntos de objetos espacio-textuales y procesar eficientemente tres tipos de consultas SKQ. La estructura permite representar y procesar conjuntos de datos directamente en memoria principal, evitando realizar las operaciones de entrada/salida que ocurren en los modelos de memoria secundaria que tradicionalmente se han utilizado y propuesto hasta la fecha.

También se han presentado una serie de experimentos para evaluar el rendimiento de la implementación de los algoritmos de consulta SKQ para cBiK, comparandolas con el índice S2I, una de las mejores estructuras de datos para construir índices espacio-textuales en memoria secundaria según literatura [12]. La experimentación se realiza sobre dos conjuntos de datos de la vida real, el primero correspondiente a Puntos de Interés (POI's) y el segundo a Tweets geolocalizados obtenidos desde Twitter.

La experimentación consideró la evaluación de los tiempos de ejecución y el espacio de almacenamiento utilizado por cada estructura. Los resultados muestran que cBiK supera a S2I en almacenamiento, concretamente cBiK requirió de aproximadamente un 38 % del almacenamiento de S2I para los mismos conjuntos espacio-textuales. Respecto al tiempo de ejecución de las consultas, cBiK supera a S2I en 1, 2 y 3 órdenes de magnitud para responder las consultas RkSKQ, BkSKQ y bRS-SKQ respectivamente.

De acuerdo a la revisión de la literatura, este trabajo de investigación propone la primera estructura de datos compacta para representar conjuntos de datos espacio-textuales. Por todo lo anteriormente expuesto, se puede concluir que es posible implementar algoritmos eficientes para realizar consultas espacio-textuales mediante una EDC directamente en memoria principal.

8.2. Trabajo Futuro

Considerando el trabajo realizado en esta investigación, se plantea como trabajo futuro:

- Compactar el esquema hashing de las palabras claves y los valores decimales de las coordenadas geográficas, para reducir el espacio de almacenamiento de la estructura.

- Mejorar la construcción de la estructura cBiK, realizando alguna estrategia de segmentación de los bitmaps para no utilizar números de índices tan grandes.
- Implementar nuevos algoritmos para resolver otras consultas espacio-textuales de interés sobre cBiK.
- Paralelizar los algoritmos para mejorar los tiempos de ejecución.
- Extender cBiK a un modelo que maneje las frecuencias de las palabras claves.
- Extender cBiK para que sea un índice que soporte puntos espaciales en movimiento.
- Extender cBiK para que sea un índice dinámico, que pueda insertar y eliminar nuevos objetos espacio-textuales en su estructura.
- Estudiar otras EDC para lograr representar conjuntos de datos espacio-textuales.

Bibliografía

[1] Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. ACM SIGMOD Record 19(2), 322–331 (1990)

[2] de Bernardo, G.: New data structures and algorithms for the efficient management of large spatial datasets. Ph.D. thesis, Universidade da Coruña (2014)

[3] de Bernardo, G., Gutierrez, G., Ladra, S., Troncoso, B.A., Penabad, M.R., Brisaboa, N.R.: Efficient Set Operations over k2-Trees 053, 373–382 (2015)

[4] Boldi, P., Rosa, M., Santini, M., Vigna, S.: Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks p. 587 (2011)

[5] BÖxhm, C., Klump, G., Kriegel, H.P.: XZ-Ordering: A Space-Filling Curve for Objects with Spatial Extension. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 1651, pp. 75–90 (1999), http://link.springer.com/10.1007/3-540-48482-5{_}7

[6] Brisaboa, N.R., Cánovas, R., Martínez-Prieto, M.A., Navarro, G.: Compressed String Dictionaries. Search (jan 2011), http://arxiv.org/abs/1101.5506

[7] Brisaboa, N.R., Ladra, S.: K 2 -trees for Compact Web Graph Representation. Lncs (2009)

[8] Brown, R.A.: Building a Balanced k-d Tree in O(kn log n) Time. Journal of Computer Graphics Techniques (JCGT) 4(1), 50–68 (oct 2014), http://arxiv.org/abs/1410.5420

[9] Cao, X., Cong, G., Jensen, C.S., Ooi, B.C.: Collective spatial keyword querying. Proceedings of the 2011 international conference on Management of data - SIGMOD '11 p. 373 (2011), http://portal.acm.org/citation.cfm?doid=1989323.1989363

[10] Cary, A., Wolfson, O., Rishe, N.: Efficient and scalable method for processing top-k spatial Boolean queries. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). vol. 6187 LNCS, pp. 87–95 (2010)

[11] Chen, L., Cong, G., Cao, X., Tan, K.L.: Temporal Spatial-Keyword Top-k publish/subscribe. Proceedings - International Conference on Data Engineering 2015-May, 255–266 (2015)

[12] Chen, L., Cong, G., Jensen, C.S., Wu, D.: Spatial keyword Query Processing: An Experimental Evaluation. Proceedings of the VLDB Endowment 6(3), 217–228 (2013), http://dl.acm.org/citation.cfm?doid=2535569.2448955

[13] Chen, Y., Chen, Y., Suel, T., Suel, T., Markowetz, a., Markowetz, a.: Efficient query processing in geographic web search engines. Proceedings of the 2006 ACM SIGMOD international conference on Management of data pp. 277–288 (2006), http://portal.acm.org/citation.cfm?id=1142505

[14] Christoforaki, M., He, J., Dimopoulos, C., Markowetz, A., Suel, T.: Text vs. Space: Efficient Geo-Search Query Processing pp. 423–432 (2011)

[15] Claude, F.: Space-Efficient Data Structures for Information Retrieval. Ph.D. thesis (2013)

[16] Claude, F., Ladra, S.: Practical representations for web and social graphs p. 1185 (2011)

[17] Claude, F., Navarro, G.: Extended compact Web graph representations. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 6060 LNCS, 77–91 (2010)

[18] Cong, G., Jensen, C.S., Wu, D.: Efficient retrieval of the top-k most relevant spatial web objects. Proceedings of the VLDB Endowment 2(1), 337–348 (2009), http://portal.acm.org/citation.cfm?id=1687666

[19] De Felipe, I., Hristidis, V., Rishe, N.: Keyword Search on Spatial Databases. 2008 IEEE 24th International Conference on Data Engineering pp. 656–665 (2008), http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4497474

[20] Denzumi, S., Kawahara, J., Tsuda, K., Arimura, H., Minato, S.I., Sadakane, K.: DenseZDD: A compact and fast index for families of sets. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 8504 LNCS, 187–198 (2014)

[21] Faloutsos, C., Christodoulakis, S.: Signature files: an access method for documents and its analytical performance evaluation. ACM Transactions on Information Systems 2(4), 267–288 (1984)

[22] Finkel, R.A., Bentley, J.L.: Quad trees a data structure for retrieval on composite keys. Acta Informatica 4(1), 1–9 (1974)

[23] Gaede, V., Günther, O.: Multidimensional access methods. ACM Computing Surveys 30(2), 170–231 (1998), http://portal.acm.org/citation.cfm?doid=280277.280279

[24] Gargantini, I.: An effective way to represent quadtrees. Communications of the ACM 25(12), 905–910 (1982)

[25] Göbel, R., Henrich, A., Niemann, R., Blank, D.: A hybrid index structure for geo-textual searches. Cikm p. 1625 (2009), http://portal.acm.org/citation.cfm?doid=1645953.1646188

[26] Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: Plug and play with succinct data structures. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 8504 LNCS, 326–337 (2014)

[27] Gopinath, A.P., Salim, A.: An Approach for Faster Processing of Top-k Spatial Keyword Queries. International Conference on Control Communication & Computing India (ICCC) (November), 622–627 (2015)

[28] Graham, R.L.: On Finding the Convex Hull of a Simple Polygon. Journal of Algorithms 331(80-03-FC-01), 324–331 (1980)

[29] Grossi, R., Gupta, A., Vitter, J.S.: High-Order Entropy-Compressed Text Indexes. Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms 2068, 841–850 (2003)

[30] Guttman, A.: R-trees: a dynamic index structure for spatial searching. ACM 14 (1984)

[31] Hariharan, R., Hore, B., Chen, L., Mehrotra, S.: Processing spatial-keyword (SK) queries in Geographic Information Retrieval (GIR) systems. Proceedings of the International Conference on Scientific and Statistical Database Management, SSDBM (Ssdbm) (2007)

[32] Hong, H.J., Chiu, G.M., Tsai, W.Y.: A single quadtree-based algorithm for top-k spatial keyword query. Pervasive and Mobile Computing 42, 93–107 (2017), https://doi.org/10.1016/j.pmcj.2017.09.009

[33] Huang, W., Li, G., Tan, K.l., Feng, J.: Efficient safe-region construction for moving top-K spatial keyword queries. Proceedings of the 21st ACM international conference on Information and knowledge management - CIKM '12 p. 932 (2012), http://dl.acm.org/citation.cfm?doid=2396761.2396879

[34] Jacobson, G.: Space-efficient static trees and graphs. Proc. 30th FOCS pp. 549–554 (1989)

[35] Khodaei, A., Shahabi, C., Li, C.: Hybrid indexing and seamless ranking of spatial and textual features of web documents. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 6261 LNCS(PART 1), 450–466 (2010)

[36] Larsson, N.J., Moffat, A.: Offline Dictionary-Based Compression pp. 1–10

[37] Li, G., Feng, J., Xu, J.: DESKS: Direction-aware spatial keyword search. In: Proceedings - International Conference on Data Engineering. pp. 474–485 (2012)

[38] Li, Z., Lee, K.C.K., Zheng, B., Lee, W.c., Lee, D.L., Wang, X.: IR-Tree : An Efficient Index for Geographic Document Search 23(4), 585–599 (2011)

[39] Lin, T.W.: Set operations on constant bit-length linear quadtrees. Pattern Recognition 30(7), 1239–1249 (1997)

[40] Lin, X., Cheema, M.A., Wang, X., Zhang, C., Zhang, Y., Zhang, W.: Diversified Spatial Keyword Search On Road Networks (2014)

[41] Lu, Y., Lu, J., Cong, G., Wu, W., Shahabi, C.: Efficient Algorithms and Cost Models for Reverse Spatial-Keyword k -Nearest Neighbor Search. ACM Transactions on Database Systems 39(2), 1–46 (2014), http://dl.acm.org/citation.cfm?id=2627748.2576232

[42] Morton, G.M.: A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing. International Business Machines Company (1966), https://books.google.cl/books?id=9FFdHAAACAAJ

[43] Munro, J.I.: Tables. Foundations of Software Technology and Theoretical Computer Science LNCS 1180, 37–42 (1996), http://link.springer.com/10.1007/3-540-62034-6{_}35

[44] Navarro, G.: Wavelet Trees for All. In: Kärkkäinen, J., Stoye, J. (eds.) Combinatorial Pattern Matching. pp. 2–26. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

[45] Navarro, G.: Compact Data Structures – A practical approach. Cambridge University Press (2016)

[46] Okanohara, D., Sadakane, K.: Practical Entropy-Compressed Rank/Select Dictionary (April) (2006), http://arxiv.org/abs/cs/0610001

[47] OÑeil, D.: Nearest neighbors problem. Encyclopedia of GIS pp. 783–787 (2008)

[48] Quijada, C.: Ampliación de las Capacidades de Estructuras de Datos Compactas (2017)

[49] Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k -ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3(4), 43–es (nov 2007), http://arxiv.org/abs/0705.0552{%}0Ahttp://dx.doi.org/10.1145/1290672.1290680http://portal.acm.org/citation.cfm?doid=1290672.1290680

[50] Rocha-Junior, J.B., Gkorgkas, O., Jonassen, S., Nørvåg, K.: Efficient Processing of Top-k Spatial Keyword Queries. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 6849 LNCS, pp. 205–222 (2011), http://link.springer.com/10.1007/978-3-642-22922-0{_}13

[51] Rocha-junior, J.B., Nørvåg, K.: Top-k Spatial Keyword Queries on Road Networks pp. 168–179 (2012)

[52] Roy, S.B., Chakrabarti, K.: Location-Aware Type Ahead Search on Spatial Databases : Semantics and Efficiency pp. 361–372 (2011)

[53] Samet, H.: Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling) (2005)

[54] Samet, H.: The Quadtree and Related Hierarchical Data Structures. ACM Computing Surveys 16(2), 187–260 (1984), http://portal.acm.org/citation.cfm?doid=356924.356930

[55] San Juan, C., Gutiérrez, G., Martínez-Prieto, M.A.: A Compact Memory-based Index for Spatial Keyword Query Resolution. In: 37th International Conference of the Chilean Computer Science Society (SCCC) (2018)

[56] Seara, C.: On geometric separability. Ph.D. thesis, Univ. Politecnica de Catalunya (2002)

[57] Shewchuk, J.R.: What is a Good Linear Element. Eleventh International Meshing Roundtable pp. 115–126 (2002)

[58] Vaid, S., Jones, C.B., Joho, H., Sanderson, M.: Spatio-Textual Indexing for Geographical Search on the Web. Advances in Spatial and Temporal Databases 9th International Symposium SSTD 2005 3633, 218–235 (2005), http://orca.cf.ac.uk/1840/

[59] Wu, D., Cong, G., Jensen, C.S.: A framework for efficient spatial web object retrieval. VLDB Journal 21(6), 797–822 (2012)

[60] Wu, D., Yiu, M.L., Cong, G., Jensen, C.S.: Joint Top-K Spatial Keyword Query Processing. IEEE Transactions on Knowledge and Data Engineering 24(10), 1889–1903 (2012)

[61] Wu, D., Yiu, M.L., Jensen, C.S., Cong, G.: Efficient Continuously Moving Top- K Spatial Keyword Query Processing pp. 541–552 (2011)

[62] Yan, H., Ding, S., Suel, T.: Inverted index compression and query processing with optimized document ordering. Proceedings of the 18th international conference on World wide web - WWW '09 p. 401 (2009), http://portal.acm.org/citation.cfm?doid=1526709.1526764

[63] Zhang, C., Zhang, Y., Zhang, W., Lin, X.: Inverted Linear Quadtree: Efficient Top K Spatial Keyword Search. IEEE Transactions on Knowledge and Data Engineering 28(7), 1706–1721 (2016)

[64] Zhang, D., Chee, Y.M., Mondal, A., Tung, A.K.H., Kitsuregawa, M.: Keyword search in spatial databases: Towards searching by document. In: Proceedings - International Conference on Data Engineering. pp. 688–699 (2009)

[65] Zhang, D., Tan, K.l., Tung, A.K.H.: Scalable top-k spatial keyword search. Proceedings of the 16th International Conference on Extending Database Technology - EDBT '13 p. 359 (2013), http://www.scopus.com/inward/record.url?eid=2-s2.0-84876799315{&}partnerID=tZOtx3y1

[66] Zhou, Y., Xie, X., Wang, C., Gong, Y., Ma, W.Y.: Hybrid index structures for location-based web search. Proceedings of the 14th ACM international conference on Information and knowledge management - CIKM '05 (49), 155 (2005), http://portal.acm.org/citation.cfm?doid=1099554.1099584

[67] Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Computing Surveys 38(2), 6–es (2006), http://portal.acm.org/citation.cfm?doid=1132956.1132959