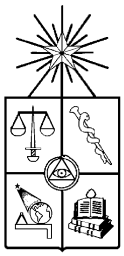


Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencia de la Computación

Métodos de Acceso y Procesamiento de Consultas Espacio-Temporales

Por

Gilberto Antonio Gutiérrez Retamal

Tesis para optar al grado de Doctor en Ciencias
mención Ciencias de la Computación

Profesores guías	:	**Gonzalo Navarro Badino**
	:	**María Andrea Rodríguez Tastets**
Comité	:	María Cecilia Rivara Zúñiga
	:	Claudio Domingo Gutiérrez Gallardo
	:	Claudia Bauzer Medeiros (profesor externo, Universidad de Campinas, SP - Brasil)

Santiago - Chile
Abril de 2007

a mi esposa Susana,
a mis hijos Susana y Felipe,
a mi padre Francisco,
a mi madre Concepción,
a mis tios Elvia y Gerardo y
a mi nietecita Sofía.

Agradecimientos

Uno de los riesgos que normalmente se corre al dar los agradecimientos es que se omitan personas que han aportado de alguna manera a lograr un objetivo. Trataré de mimimizar este riesgo. Sin embargo, de todas formas pido disculpas por aquellas personas que, pese a mi esfuerzo, se me han escapado y que de alguna u otra forma me han ayudado a alcanzar esta importante meta.

Quisiera partir agradeciendo a la Universidad del Bío Bío, mi casa, por depositar en mi la confianza y darme el apoyo para asumir este desafio y a la Universidad de Chile por aceptarme en su programa de Doctorado.

También quisiera agradecer a mi colega y amigo Benito Umaña, quien como Director del Departamento de Auditoría tramitó ante las autoridades de mi Universidad mi liberación de jornada para dedicarme de manera exclusiva a mi doctorado. Benito, te debo otro favor. También al Decano de la Facultad de Ciencias Empresariales de la Universidad del Bío-Bío Sr. Alex Medina G. quien respaldó en todo momento mi postgrado.

Mis más sinceros agradecimientos al Profesor José Pino del Departamento de Ciencias de la Computación de la Universidad de Chile quien, en su función de Director del programa de Doctorado, me orientó y motivó para completar mi postgrado. De la misma manera quisiera agradecer a Angélica Aguirre quien me ayudó de manera muy cordial con todos mis trámites durante mi permanencia en el programa.

De forma especial agradezco a mis estudiantes de pregrado de la Universidad del Bío Bío Alejandro González, José Orellana quienes realizaron la programación del SEST-Index y a Chun-Hau Lai y Yerko Bravo quienes hicieron lo mismo con la variante del SEST-Index. Sus observaciones durante la implementación contribuyeron de manera importante al mejoramiento de los métodos de accesos propuestos en esta tesis.

Deseo agradecer también a la profesora de la Universidad Nacional de San Luis (Argentina) Verónica Gil Costa quien, de manera muy decidida, me asesoró con la implementación disponible del MVR-tree. Su ayuda me permitió utilizar el MVR-tree en varios de los experimentos realizados en este trabajo.

A mis compañeros y amigos del programa de Doctorado Karina Figueroa, Rodrigo Paredes, Rodrigo González, Olivier Motolet, Diego Arroyuelo, Pedro Rossel, Hugo Andrés Neyen y Renzo Angels. Les agradezco de manera muy especial por la forma en que me recibieron, me apoyaron e integraron. Quisiera destacar de entre mis compañeros a Karina, con quien compartimos momentos de mucha alegría (cuando las cosas resultaban) y también de decepciones (cuando las cosas no resultaban). Su apoyo y consejos me fueron de muchísima ayuda, ni siquiera te imaginas Karina.

También quiero agradecer a los profesores de mi comisión: María Cecilia Rivara, Claudia Medeiros y Claudio Gutiérrez quienes revisaron esta tesis y realizaron importantes observaciones que ayudaron a mejorar el trabajo.

De manera muy especial deseo dar mis más sinceros agradecimientos a mis profesores supervisores: Andrea Rodríguez y Gonzalo Navarro. Ellos tuvieron que lidiar con mis dudas y confusiones. También tuvieron la paciencia para escucharme y la sabiduria para dejarme sólo cuando era necesario explorar nuevos caminos. Si bién su ayuda profesional ha sido muy valiosa para mi, su motivación, consideración y confianza fueron imprescindibles para completar mi tesis. Les estaré eternamente agradecido.

He dejado para el final a una persona de quien siempre estaré muy agradecido, me refiero a mi esposa Susana. En primer lugar por aceptar compartir con ella todos estos años los que han

sido, sin duda alguna, los mejores de mi vida. En segundo lugar por su apoyo, comprensión y confianza para emprender y completar mi perfeccionamiento. Acudiendo al dicho "*Destrás de un gran hombre hay una gran mujer "*, creo que con Susana tengo muy buenas posibilidades de llegar a ser un gran hombre, a pesar de que le queda mucho trabajo para lograrlo.

RESUMEN DE LA TESIS
PARA OPTAR AL GRADO DE DOCTOR
EN CIENCIAS, MENCION COMPUTACION
POR: GILBERTO GUTIERREZ R.
FECHA: 17-04-2007
PROF. GUIAS : Sr. GONZALO NAVARRO
: Sra. ANDREA RODRIGUEZ

Métodos de Acceso y Procesamiento de Consultas Espacio-Temporales

Existe una necesidad creciente por contar con aplicaciones espacio-temporales que necesitan modelar la naturaleza dinámica de los objetos espaciales. Las bases de datos espacio-temporales intentan proporcionar facilidades que permitan apoyar la implementación de este tipo de aplicaciones. Una de estas facilidades corresponde a los métodos de acceso, que tienen por objetivo construir índices para permitir el procesamiento eficiente de las consultas espacio-temporales.

En esta tesis se describen nuevos métodos de acceso basados en un enfoque que combina dos visiones para modelar información espacio-temporal: snapshots y eventos. Los snapshots se implementan por medio de un índice espacial y los eventos que ocurren entre snapshots consecutivos, se registran en una bitácora. Se estudió el comportamiento de nuestro enfoque considerando diferentes granularidades del espacio. Nuestro primer método de acceso espacio-temporal (SEST-Index) se obtuvo teniendo en cuenta el espacio completo y el segundo (SEST_L) considerando las divisiones más finas del espacio producidas por el índice espacial.

En esta tesis se realizaron varios estudios comparativos entre nuestros métodos de acceso y otros métodos propuestos en la literatura (HR-tree y MVR-tree) para evaluar las consultas espacio-temporales tradicionales (*time-slice* y *time-interval*). Los estudios muestran la superioridad de nuestras estructuras de datos en términos de almacenamiento y eficiencia para procesar tales consultas en un amplio rango de situaciones. Para nuestros dos métodos de acceso se definieron modelos de costos que permiten estimar tanto el almacenamiento como el tiempo de las consultas. Estos modelos se validaron experimentalmente presentando una buena capacidad de estimación.

Basándonos en nuestros métodos propusimos algoritmos para procesar otros tipos de consultas espacio-temporales, más allá de *time-slice* y *time-interval*. Específicamente diseñamos algoritmos para evaluar la operación de reunión espacio-temporal, consultas sobre eventos y sobre patrones espacio-temporales. Se realizaron varios experimentos con el propósito de comparar el desempeño de nuestros métodos frente a otros propuestos en la literatura (3D R-tree, MVR-tree, HR-tree y CellList) para procesar estos tipos de consultas. Los resultados muestran un rendimiento, en general, favorable a nuestros métodos.

En resumen, nuestros métodos son los primeros que resuelven de manera eficiente no sólo las consultas de tipo *time-slice* y *time-interval*, sino también varias otras de interés en aplicaciones espacio-temporales.

Contenido

Lista de Figuras

Lista de Tablas

Lista de Algoritmos

Capítulo 1

Introducción

1.1 Motivación y objetivos de la tesis

Los Sistemas de Administración de Bases de Datos (SABDs) han sido por excelencia la tecnología informática que ofrece las herramientas adecuadas para manejar grandes volúmenes de datos interrelacionados.

Inicialmente esta tecnología sólo permitía modelar bases de datos cuyos objetos se representaban con tipos de datos básicos o primitivos, tales como enteros y cadenas de caracteres (strings), entre otros. Con la aparición de nuevas aplicaciones que necesitan modelar objetos más complejos (sonido, imágenes, texto y video, entre otros), los SABDs se fueron extendiendo para soportar nuevos tipos de datos. Algunos ejemplos de tales extensiones corresponden a la incorporación de facilidades para el tratamiento de texto (bases de datos de texto o textuales) o de datos espaciales (bases de datos espaciales).

La mayoría de los modelos de bases de datos, sin embargo, solamente tienen la capacidad de representar un único estado de la realidad modelada, que normalmente corresponde al estado más reciente. De esta forma, cada vez que un objeto cambia su estado en el mundo real, su estado actual reemplaza el estado mantenido en la base de datos, con lo cual la historia del objeto se destruye.

Las bases de datos temporales, que se centran en aplicaciones que requieren manejar algún aspecto del tiempo para estructurar u organizar la información, proporcionan un conjunto de herramientas para administrar los diferentes estados de los objetos a través del tiempo. En las bases de datos temporales se consideran dos dimensiones del tiempo [ST99], a saber: (i) *Tiempo válido* que corresponde al tiempo en el cual un hecho se considera verdadero para la realidad modelada. Una relación que interpreta el tiempo de esta forma se denomina *relación de tiempo válido*. (ii) *Tiempo de transacción*, que hace referencia al tiempo en el cual el hecho se registró en la base de datos. Las relaciones que consideran esta dimensión del tiempo se denominan *relaciones de tiempo de transacción*. Existen aplicaciones en la cuales se requiere mantener, en una misma relación, las dos dimensiones del tiempo. A estas relaciones se les conoce como *relaciones bitemporales*.

De la combinación de la dimensión temporal con la dimensión espacial de los objetos surgen las bases de datos espacio-temporales, las que en un sentido amplio modelan objetos espaciales cuya identidad, posición o forma cambian a través del tiempo [TSPM98]. Con tales bases de datos es posible modelar la naturaleza dinámica de los atributos espaciales de los objetos permitiendo contar con aplicaciones más cercanas al mundo real, el cual es esencialmente cambiante [NST98]. Es decir, con este tipo de bases de datos es posible modelar no solamente el estado espacial actual

de los objetos, sino también los estados alcanzados en el pasado e incluso predecir la posición que un objeto podría llegar a tener en un tiempo futuro.

Actualmente están apareciendo muchas aplicaciones en el dominio espacio-temporal. Un primer ejemplo podemos tomarlo de la telefonía móvil. Ya en el año 2002 existían alrededor de 500 millones de teléfonos móviles y se espera que para mediados de la década actual este número alcance a 1 billón (mil millones) de estos dispositivos [TP01b]. Aunque muchos de los sistemas manejan sólo la posición actual de los dispositivos móviles, es conveniente mantener la información histórica de localización de estos dispositivos con el propósito de proveer mejores servicios de comunicación, por ejemplo.

Otras aplicaciones espacio-temporales se encuentran en el ámbito del transporte. Por ejemplo, sistemas de supervisión de tráfico que monitorean las posiciones de los vehículos y patrones de desplazamiento con el propósito de proveer servicios tales como prevención de congestión, rutas despejadas, límites de velocidad, etc. Otro ejemplo en este contexto lo podemos encontrar en los servicios de control de tráfico aéreo. En este caso se utiliza el *principio de la pirámide de Heinrich* para preveer la presencia de siniestros aéreos. El principio de la pirámide de Heinrich predice que por cada accidente fatal, existen entre tres a cinco accidentes no fatales, diez a quince "incidentes" y cientos de hechos no reportados. Bajo los estándares de la aviación, el caso en que dos aviones vuelen a una distancia menor que 5 millas a la misma altitud se considera un incidente, pero el caso en que la distancia es mayor que 5 millas constituye un hecho no reportado. Si se mantiene la información de las trayectorias de los vuelos es posible evaluar consultas como la siguiente *"cuántos pares de aviones estuvieron a menos de 10 millas el mes pasado en la zona aérea de Santiago"*, obtener la cantidad de hechos no reportados y, de esta manera, poder tomar acciones para reducir tanto los incidentes como los accidentes.

La planificación urbana también constituye un dominio de aplicaciones espacio-temporales. En este caso puede ser de interés mantener la aparición y desaparición de objetos (por ejemplo, edificios) en lugares e instantes de tiempo específicos. También puede ser conveniente mantener, a través del tiempo, los límites comunales, el trazado de redes de energía eléctrica, redes de agua o redes de telefonía fija, entre otros. El propósito de estas aplicaciones es poder recuperar estados de las ciudades en algún punto del tiempo en el pasado.

En general y de acuerdo con [PT98], es posible clasificar las aplicaciones espacio-temporales en tres categorías dependiendo del tipo de datos que necesitan manejar.

i. Aplicaciones que tratan con cambios continuos de los objetos, tales como el movimiento de un auto cuando se encuentra desplazándose por una carretera.

ii. Aplicaciones que incluyen objetos ubicados en el espacio y que pueden cambiar su posición por medio de la modificación de su forma geométrica o atributos espaciales. Por ejemplo, los cambios que se pueden producir en los límites administrativos de una comuna o ciudad a través del tiempo. En este tipo de aplicaciones, los cambios espaciales sobre los objetos ocurren de manera discreta.

iii. Aplicaciones que integran los dos comportamientos anteriores. Este tipo de aplicaciones se puede encontrar en el área medioambiental, donde es necesario modelar tanto los movimientos de los objetos, como las distintas formas geométricas que éstos alcanzan a lo largo del tiempo.

Notar que las dimensiones temporales en algunos tipos de aplicaciones no se distinguen claramente. Por ejemplo, para las aplicaciones del tipo (i), si cada auto está equipado con un GPS y envía su localización a un servidor central con un retardo mínimo, el tiempo válido y el de transacción son similares y, por lo tanto, la base de datos se puede considerar de tiempo válido, de tiempo de transacción y bitemporal a la vez. Distinto es el caso de las aplicaciones del tipo (ii). En este tipo de aplicaciones es más clara la diferencia entre las dos dimensiones del tiempo ya que, por ejemplo, los límites de una ciudad son válidos dentro de un periodo de tiempo, el cual puede ser distinto al tiempo en el cual los límites se registran en la base de datos. Aclarado lo anterior, en este trabajo la interpretación del tiempo corresponde a la de tiempo válido.

Las consultas espacio-temporales que han recibido mayor atención son dos: (i) *time-slice* y (ii) *time-interval* [AAE00, TPZ02, TP01a, MGA03, GS05, TSPM98, PJT00]. Una consulta de tipo *time-slice* permite recuperar todos los objetos que intersectan un rango espacial (window) en un instante de tiempo específico. A su vez, una consulta de tipo *time-interval* extiende la misma idea a un intervalo de tiempo.

Debido al dinamismo de los objetos espaciales, las bases de datos espacio-temporales necesitan almacenar grandes volúmenes de información por periodos largos de tiempo. Estas grandes cantidades de información hacen imposible pensar en un procesamiento secuencial de las consultas. Una mejor solución es construir un índice con algún método de acceso espacio-temporal que permita accesar sólo una pequeña parte de la base de datos.

En la literatura se han propuesto varios métodos de acceso espacio-temporales. Por ejemplo, HR-tree [NST99, NST98], MR-tree [XHL90], MVR-tree [TP01b, TP01a, TPZ02], 3D R-tree [TVS96, PLM01] y RT-tree [XHL90], entre otros. Tales métodos apuntan principalmente a resolver las consultas de tipo *time-slice* y *time-interval*. Sin embargo, existen otros tipos de consultas, tales como la reunión espacio-temporal [Kol00, TP01a, SHPFT05], consultas sobre el vecino más cercano [AAE00], consultas sobre patrones espacio-temporales [HKBT05, Sch05, ES02], y sobre eventos espacio-temporales [Wor05, dMR04, dMR05], entre otras, que no han sido abordadas por los actuales métodos de acceso y para las cuales existe mucho interés en contar con algoritmos eficientes para resolverlas. De estos tipos de consultas, las consultas sobre patrones espacio-temporales y sobre eventos constituyen tipos de consultas novedosas y de mucha utilidad. Las consultas sobre patrones espacio-temporales permiten especificar secuencia de predicados en la consulta. Por ejemplo, "*encontrar los autos que primero entraron área del centro de la ciudad de Santiago, luego se desplazaron a la zona de la comuna de Melipilla y finalmente aparecieron en la ciudad Rancagua*". A su vez las consultas sobre eventos permiten recuperar los hechos espaciales que en un determinado momento han ocurrido en un área determinada. Por ejemplo, "*¿ cuántos autos entraron al área del centro de la ciudad de Santiago a las 15 horas de ayer* ?".

El objetivo de esta tesis es diseñar, implementar y evaluar métodos de acceso espacio-temporales basados en snapshots y eventos de tal manera de representar de manera explícita los eventos de los objetos en el tiempo. Estos métodos de acceso, a diferencia de los métodos de acceso actuales quienes utilizan los datos de los eventos para actualizar las estructuras de datos subyacente, deben permitir la construcción de índices para procesar de manera eficiente consultas espacio-temporales, no sólo del tipo *time-slice* y *time-interval*, sino que además consultas como las enumeradas en el párrafo anterior.

1.2 Contribuciones de la tesis

Las principales contribuciones de esta tesis son las siguientes:

i. Se presentaron dos nuevos métodos de acceso espacio-temporal, SEST-Index y SEST_L, basados en snapshots y eventos. Con estos métodos es posible procesar de manera eficiente las consultas de tipo *time-slice* y *time-interval*. El SEST_L es una evolución del SEST-Index, al que supera en casi todos los aspectos.

ii. Se comparó SEST_L con el método de acceso MVR-tree [TP01b, TP01a, TPZ02] en términos de almacenamiento y tiempo para procesar las consultas de tipo *time-slice* y *time-interval*, resultando el SEST_L superior en almacenamiento y en consultas de tipo *time-interval*. El MVR-tree es considerado el mejor método de acceso espacio-temporal.

iii. Para los dos métodos de acceso creados se definieron modelos de costo que estiman el almacenamiento y la eficiencia de las consultas. Las evaluaciones experimentales realizadas demostraron que los modelos presentan una buena capacidad de estimación. Los modelos de costo se generalizaron de tal forma de modelar la asignación de las bitácoras con los eventos en diferentes niveles de las jerarquías de las subdivisiones de las regiones, resultando que SEST_L representa efectivamente la mejor elección.

iv. Se diseñaron algoritmos para procesar consultas sobre eventos. Nuestros métodos de acceso presentaron un buen comportamiento para este tipo de consultas, pues se basan precisamente en mantener de manera explicíta la información de dichos eventos.

v. Se diseñó un algoritmo eficiente para procesar la reunión espacio-temporal utilizando SEST_L. Este algoritmo se comparó con el 3D R-tree [TVS96, PLM01] resultando ser cinco veces más rápido.

vi. También se diseñó y evaluó un algoritmo (sobre el SEST_L) para procesar consultas sobre patrones espacio-temporales. Este algorimo tuvo un rendimiento levemente inferior a un algoritmo basado en una estructura ad-hoc, la cual es muy ineficiente para evaluar consultas de tipo *time-slice* y *time-interval*.

vii. Finalmente, se propusieron una serie de nuevos tipos de consultas espacio-temporales y algoritmos preliminares basados en SEST_L para abordarlas.

Como resultado de esta tesis se han generado las siguientes publicaciones en revistas y en conferencias regionales e internacionales.

- Gilberto Gutiérrez. Propuesta de un método de acceso espacio-temporal. II WorkShop de Bases de datos, Chillán (Chile), 4 de Noviembre de 2003.

- Gilberto Gutiérrez. Propuesta de un método de acceso espacio-temporal. Revista de la Sociedad Chilena de Ciencias de la Computación, vol. 5(1), 2003 (trabajo seleccionado como uno de los dos mejores presentados en el II Workshop de Bases de datos, Chillán (Chile), 4 Noviembre de 2003).

- Gilberto Gutiérrez, Gonzalo Navarro y Andrea Rodríguez. An Access Method for Objects Moving among Fixed Regions. III Workshop de Bases de Datos, Arica (Chile), 9 de Noviembre de 2004.

- Edilma Gagliardi, María Dorzán, Fernando Carrasco, J. Camilo García, Juan Gómez y Gilberto Gutiérrez. Métodos de Acceso espacio-temporales: nuevas propuestas basadas en métodos existentes. VII Workshop de Investigadores en Ciencias de la Computación 2005. Río Cuarto (Argentina), 13–14 de Mayo de 2005.

- Anibal Díaz, Edilma Gagliardi, Gilberto Gutiérrez y Norma Herrera. Procesamiento de join espacio-temporal. VII Workshop de Investigadores en Ciencias de la Computación, Río Cuarto (Argentina), 13–14 de Mayo 2005.

- Edilma Gagliardi, Gilberto Gutiérrez, María Dorzán y Juan Gómez. D*R-Tree: un método de acceso espacio-temporal. XI Congreso Argentino de Ciencias de la Computación 2005. Santa Fe (Argentina), 5–10 de Octubre de 2005.

- Gilberto Gutiérrez, Gonzalo Navarro, Andrea Rodríguez, Alejandro González y José Orellana. A Spatio-temporal Access Method based on Snapshots and Events. XIII ACM international Workshop on Geographic Information Systems, Bremen (Alemania), Noviembre 4–5, 2005.

- Edilma Gagliardi, María Dorzán, Juan Gómez y Gilberto Gutiérrez. Procesamiento de consultas espacio-temporales sobre un índice eficiente. VII Workshop de Investigadores en Ciencias de la Computación 2006, La Matanza (Argentina), Mayo de 2006.

- Andrés Pascal, Anabella de Battista, Gilberto Gutiérrez y Norma Herrera. Búsqueda en Bases de datos Métricas-temporales. VIII Workshop de Investigadores en Ciencias de la Computación, La Matanza (Argentina), Mayo de 2006.

- María Dorzán, Edilma Gagliardi, Juan Gómez Barroso y Gilberto Gutiérrez. Un nuevo índice eficiente para resolver diversas consultas espacio-temporales. XXXII Conferencia Latinoamericana de Informática, Santiago (Chile), Agosto de 2006.

- Anibal Díaz, Edilma Gagliardi y Gilberto Gutiérrez. Algoritmo de reunión espacio-temporal usando MVR-tree. II Congreso de Ingeniería e Investigación Científica y V Concurso de Proyectos de Investigación y Desarrollo. Universidad Tecnológica del Perú, Lima (Perú), 8–10 de Noviembre de 2006.

- Gilberto Gutiérrez, Gonzalo Navarro y Andrea Rodríguez. A Spatio-temporal Access Methods based on Snapshots and Events. Artículo de revista (journal) sometido a revisión a ACM Transactions on Database Systems.

1.3 Descripción de la estructura y contenidos de la tesis

Esta tesis contempla 3 partes y 9 capítulos. En la primera parte se presenta una revisión de las bases de datos espaciales y espacio-temporales. El Capítulo 2 se centra en describir las bases de datos espaciales desde el punto de vista de los tipos de datos y tipos de consultas más comunes.

En este capítulo también se describen de manera detallada los métodos de acceso espacial R-tree [Gut84] (y sus variantes) y K-D-B-tree [Rob81] los que se utilizan en nuestras propuestas. En el caso específico de R-tree se describen los algoritmos para las principales operaciones (inserción, búsqueda, eliminación, etc) y los modelos de costos asociados que permiten predecir el espacio y el rendimiento de las consultas.

El Capítulo 3 está destinado a describir las bases de datos espacio-temporales. Este capítulo, sigue la misma estructura que el anterior, es decir, se especifican los tipos de datos espacio-temporales, las aplicaciones más comunes, los tipos de consultas más estudiadas y los métodos de acceso espacio-temporales que se han propuesto hasta ahora.

La segunda parte describe los métodos de acceso espacio-temporal propuestos como resultado de esta tesis y que fueron obtenidos a partir de nuestro enfoque (snapshot y eventos). En el Capítulo 4 se presenta nuestro primer método de acceso espacio-temporal, denominado SEST-Index. Se describe el enfoque seguido, las estructuras de datos y algoritmos. También se muestran los resultados experimentales en que se compara con el HR-tree y se discute una variante. Finalmente, se propone un modelo de costo que permite estimar el almacenamiento y la eficencia de las consultas.

En el Capítulo 5 se plantea nuestro segundo método de acceso espacio-temporal que se denomina SEST_L. Este método es una evolución del SEST-Index. Se describen sus estructuras de datos y algoritmos. También se presentan resultados experimentales que lo comparan con el MVR-tree y el SEST-Index. Al igual que el SEST-Index, también se establece un modelo de costo para el SEST_L que predice su comportamiento en términos de tiempo y espacio. Uno de los problemas del SEST_L es el empeoramiento de su rendimiento a lo largo del tiempo. La solución a este problema consistió en generar snapshots globales cada ciertos instantes de tiempo. Dicha solución también se discute en este capítulo junto con resultados de nuevos experimentos y modelos de costos considerando a SEST_L con snapshots globales. El SEST-Index asigna una bitácora a la raíz del R-tree y el SEST_L las asigna a las subregiones formadas por las hojas del árbol. La pregunta natural es *¿ a regiones de qué nivel del R-tree es más conveniente asignar las bitácoras?*. El Capítulo 6 trata de despejar esta interrogante. Para ello se generalizó el modelo de costo para estimar el espacio y la eficiencia de las consultas considerando, como un parámetro del modelo, el nivel del R-tree donde asignar las bitácoras. Con el modelo general se discute cuál es el nivel óptimo del R-tree a cuyas subregiones asignar las bitácoras.

En la tercera parte de esta tesis se explica como utilizar nuestro método de acceso SEST_L para procesar eficientemente otros tipos de consultas. El Capítulo 7 se concentra en resolver la operación de reunión (join) espacio-temporal utilizando el SEST_L. Se describen los algoritmos y se discute una evaluación experimental en la que se compara con el 3D R-tree. También se presenta un modelo de costo para predecir el rendimiento de la reunión espacio-temporal.

En el Capítulo 8 se hace una exploración de la factibilidad de evaluar con el SEST_L consultas espacio-temporales que se han propuesto en la literatura reciente y por las cuales existe gran interés en contar con algoritmos eficientes en ámbitos espacio-temporales. Por ejemplo, se discute la factibilidad de procesar con el SEST_L el problema de los K-vecinos más cercanos y el par de vecinos más cercanos, evaluación de patrones espacio-temporales, entre otros. También se presenta una solución para uno de estos tipos de consultas, la cual se compara experimentalmente con otras soluciones presentes en la literatura.

Finalmente, en el Capítulo 9 se discuten las conclusiones de esta tesis.

Parte I

Bases de datos espaciales y espacio-temporales

Capítulo 2

Bases de Datos Espaciales

2.1 Introducción

Los Sistemas de Administración de Bases de Datos (SABD) actuales se han extendido con el propósito de representar y consultar objetos espaciales en forma natural. Esta extensión contempla el diseño de nuevas estructuras de datos para almacenar datos espaciales, la creación de métodos de acceso espaciales o multidimensionales y el diseño de lenguajes de consultas, entre otros.

Las principales aplicaciones que utilizan estos tipos de datos se encuentran en el ámbito de los Sistemas de Información Geográfica (GIS). En una primera etapa de los GISs, los SABDs se utilizaban básicamente para manejar los atributos no espaciales de los objetos. Por su parte, para administrar los atributos espaciales se crearon estructuras de datos ad-hoc manejadas por software específico. Sin embargo, actualmente varios productos de SABDs, tales como Oracle y Postgres, han incorporado los tipos de datos espaciales (o geométricos) encontrándose disponibles para la implementación de aplicaciones en diferentes dominios. Si bien los GISs han sido las aplicaciones más destacadas en el contexto espacial, también existen otras que han ido apareciendo, por ejemplo, en el área de Diseño Asistido por Computador (CAD), diseño VLSI, visión por computador y robótica [LJF94, Gut94, GS05].

De acuerdo a [GG98], los datos espaciales son muy particulares ya que cuentan con las siguientes propiedades:

i. Tienen una estructura compleja. Un dato espacial puede estar compuesto de un sólo punto o de varios cientos de polígonos arbitrariamente distribuidos en el espacio y, por lo tanto, no se pueden almacenar en una sola tupla.

ii. Las bases de datos espaciales tienden a ser muy grandes. Por ejemplo, un mapa geográfico puede ocupar varios gigabytes de almacenamiento.

iii. No existe un álgebra estándar definida sobre los datos espaciales y, por lo tanto, no existe un conjunto de operadores estandarizados. El conjunto de operadores es altamente dependiente del dominio de la aplicación, aunque algunos operadores son más comunes que otros.

iv. La mayoría de los operadores no son cerrados. La intersección de dos polígonos, por ejemplo, no necesariamente entrega como resultado otro polígono. Esta propiedad es particularmente relevante cuando se quieren componer operadores tal como se hace con los operadores relacionales.

v. El costo computacional de implementar los operadores espaciales es mucho mayor que los operadores relacionales.

Un Sistema de Administración de Bases de Datos Espaciales (SABDE) debería ofrecer tipos de datos apropiados y un lenguaje de consulta para soportar datos espaciales, además debe contar con métodos eficientes de acceso (índices) y modelos de costos que estimen el rendimiento de las consultas con propósitos de optimización [TSS98a].

El objetivo de este capítulo es describir los operadores espaciales y los tipos de consultas espaciales típicas. También se describen los métodos de acceso multidimensional o espacial más destacados (K-D-B-tree, R-tree y algunas de sus variantes tales como R^+ -tree y R^*-tree) y en los cuales descansan los métodos de acceso espacio-temporal propuestos en esta tesis. El K-D-B-tree corresponde a un método de acceso multidimensional o espacial que pertenece a la categoría denominada como PAM (Point Access Methods) [GG98], que considera sólo objetos de tipo punto. Por su parte, el R-tree pertenece a la clase de métodos de acceso denominada SAM (Spatial Access Methods) [GG98] los cuales pueden manejar tanto objetos de tipo punto como objetos cuyos atributos espaciales definen figuras geométricas más complejas (polilíneas, polígonos, etc.). Debido a que en esta tesis se utiliza el R-tree de manera frecuente, en este capítulo se realiza una descripción detallada de dicho método, considerando su estructura de datos, algoritmos, modelos de costos y algunas de sus variantes más conocidas.

2.2 Tipos de datos y operadores espaciales

Uno de los principales problemas de los tipos de datos espaciales es que es difícil definir un conjunto de operadores (relaciones) lo suficientemente general para cubrir todas las aplicaciones. En otras palabras, los operadores espaciales son altamente dependiente de la aplicación. Sin embargo, es posible definir algunos operadores espaciales (por ejemplo la intersección) que son más comunes que otros [GG98]. En esta sección se describen los operadores espaciales según [SC03].

Un objeto *o* en una base de datos espacial se define, normalmente, mediante varios atributos no-espaciales y un atributo espacial de algún tipo.

Para modelar los atributos espaciales de los objetos se utilizan tres abstracciones fundamentales y que son *punto*, *línea* y *región* o *polígono* [GS05] (ver Figura 2.1). Un punto representa (el aspecto geométrico) un objeto para el cual sólo interesa su posición en el espacio. Ejemplos de objetos puntos son hospitales, edificios, estaciones de trenes, etc. Una línea es una abstración que se utiliza para modelar objetos tales como ríos, líneas de trenes, carreteras, líneas telefónicas, etc. Finalmente, una región permite modelar objetos con cobertura espacial (spatial extent). Por ejemplo, países, predios agrícolas, etc.

Figura 2.1: Tres tipos de datos espaciales: *punto*, *línea* y *región*.

Existen varios tipos de operaciones sobre objetos espaciales siendo las más comunes las *topológicas*, *métricas* y de *orientación* [SC03]. Las operaciones topológicas, por ejemplo *overlap* en la Figura 2.2, establecen relaciones entre objetos espaciales las cuales se mantienen ante operaciones de rotación y traslación. Las relaciones topológicas binarias entre dos objetos espaciales, A y B, en un espacio de dos dimensiones están basadas en la intersección del interior de A (A°), borde de A (∂A) y exterior de A (A^-) con el interior (B°), borde (∂B) y exterior (B^-) de B [Ege94]. Las relaciones entre estas seis partes de los objetos se pueden resumir en una matriz (denominada matriz de nueve intersecciones) y que describe las relaciones topológicas entre dos objetos espaciales. La matriz es la siguiente:

$$\mathbf{T_9}(\mathbf{A},\mathbf{B}) = \begin{pmatrix} A^\circ \cap B^\circ & A^\circ \cap \partial B & A^\circ \cap B^- \\ \partial A \cap B^\circ & \partial A \cap \partial B & \partial A \cap B^- \\ A^- \cap B^\circ & A^- \cap \partial B & A^- \cap B^- \end{pmatrix}$$

Considerando que cada intersección de la matriz T_9 puede ser vacía (0) o no (1) es posible distinguir $2^9 = 512$ posibilidades de relaciones binarias entre dos objetos espaciales. Sin embargo, en un espacio de 2 dimensiones y para objetos de tipo *región*, sólo tienen sentido ocho de estas posibilidades. Estas relaciones son: *disjoint*, *meet*, *overlap*, *equal*, *contains*, *inside*, *covered* y *coverBy*. Estas ocho relaciones se pueden representar por medio de la matriz T_9 según se muestra en la Figura 2.2. Notar que para la relación *equal* la matriz tiene 1s en la diagonal principal (en el resto son 0s), es decir, dos objetos espaciales de tipo *región* son iguales si y sólo si las intersecciones sólo ocurren entre los interiores, bordes y exteriores de los objetos. También es posible notar la simetría que existe entre las relaciones *contains* y *inside* y entre *cover* y *coverBy* por medio de las matrices.

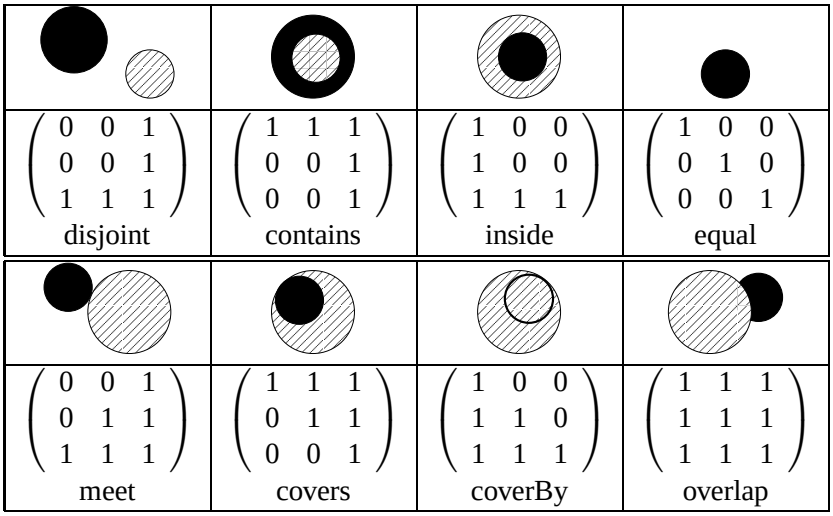


Figura 2.2: Relaciones topológicas binarias entre objetos de tipo *región*.

Las relaciones topológicas sobre otros pares de tipos de datos espaciales, por ejemplo entre (*punto, línea)*, (*punto, región*) se pueden definir de manera similar. En [ME94] se presentan 19 relaciones topológicas entre pares de objetos (*punto, región*). Algunos ejemplos de estas relaciones

topológicas pueden ser *crosses(línea, región)* (verifica si una línea cruza una región), *touches(línea, región)* (verifica si una línea toca una región), etc.

Las operaciones o relaciones métricas permiten obtener atributos numéricos de un objeto espacial o valores entre ellos que los relacionan. Por ejemplo, el área de una región, la longitud de una línea o la distancia entre dos puntos o entre una línea y un punto.

Las relaciones de orientación pueden ser básicamente de dos tipos, a saber: absoluta o relativa a un objeto. Las relaciones del primer tipo se definen en el contexto de un sistema de referencia global. Algunas de estas relaciones son: *norte, sur, este, oeste, sur-este, sur-oeste*, etc. las relaciones en el segundo caso (relativa a un objeto), se definen en base a un objeto dado. Por ejemplo *izquierda, derecha, arriba, abajo*, etc.

2.3 Consultas espaciales

En esta sección se discuten las consultas espaciales que, a juicio de [GG98], constituyen el núcleo de consultas espaciales fundamentales. Se asume que el atributo espacial (geometría) de los objetos se encuentra definido en un espacio Euclidiano n–dimensional. Denotamos el atributo espacial de un objeto o por $o.G \subseteq \mathbb{R}^n$.

La semántica y, por lo tanto, el procedimiento usado para verificar algunos predicados espaciales utilizados en la definición de las consultas, va a depender de los tipos de objetos involucrados. A modo de ejemplo consideremos el predicado espacial $o'.G = o.G$. Si el tipo de los objetos es *punto* basta con verificar las correspondientes coordenadas. En cambio si el tipo de los objetos es *región* la verificación del predicado se puede realizar mediante la matriz definida para la relación topológica *equals* en la Figura 2.2.

i. EMQ (Exact Match Query). Dado un objeto o' encontrar todo objeto o cuyo atributo espacial es igual al de o':

$$EMQ(o') = \{o \mid o'.G = o.G\}$$

ii. PQ (Point Query). Dado un punto $p \in \mathbb{R}^n$, encontrar todos los objetos o que cubren a p:

$$PQ(p) = \{o \mid p \cap o.G = p\}$$

iii. WQ/RQ (Window Query / Range Query). Dado un rango $q \subseteq \mathbb{R}^n$, encontrar todos los objetos o que tienen al menos un punto en común con q:

$$WQ(q) = \{o \mid q \cap o.G \neq \emptyset\}$$

En este tipo de consultas, el rango q es iso–orientado, es decir, las caras que definen a q son paralelas a cada uno de los ejes.

iv. IQ (Intersection Query). Dado un objeto o' con atributo espacial $o'.G \subseteq \mathbb{R}^n$, encontrar todos los objetos o que tienen al menos un punto en común con o' :

$$IQ(o') = \{o \mid o'.G \cap o.G \neq \emptyset\}$$

v. EQ (Enclosure Query). Dado un objeto o' con atributo espacial $o'.G \subseteq \mathbb{R}^n$, encontrar todos los objetos o que lo encierran o contienen completamente:

$$EQ(o') = \{o \mid o'.G \cap o.G = o'.G\}$$

vi. CQ (Containment Query). Dado un objeto o' con atributo espacial $o'.G \subseteq \mathbb{R}^n$, encontrar todos los objetos o encerrados por o':

$$CQ(o') = \{o \mid o'.G \cap o.G = o.G\}$$

vii. AQ (Adjacency Query). Dado un objeto o' con atributo espacial $o'.G \subseteq \mathbb{R}^n$, encontrar todos los objetos o adyacentes a o':

$$AQ(o') = \{o \mid o.G \cap o'.G \neq \emptyset \wedge o'.G^o \cap o.G^o = \emptyset\}$$

Aquí $o.G^o$ y $o'.G^o$ denotan el interior de las figuras geométricas formadas por los atributos espaciales $o.G$ y $o'.G$, respectivamente.

viii. NNQ (Nearest-Neighbor Query). Dado un objeto o' con atributo espacial $o'.G \subseteq \mathbb{R}^n$, encontrar todos los objetos o que tienen la mínima distancia a o':

$$NNQ(o') = \{o \mid \forall o'' : dist(o'.G, o.G) \leq dist(o'.G, o''.G)\}$$

En este caso se define $dist()$ entre atributos espaciales como la distancia (Euclidiana o de Manhattan) entre sus puntos más cercanos.

ix. SJ (Spatial Join). Dadas dos colecciones de objetos espaciales R y S y un predicado binario espacial θ, encontrar todos los pares de objetos $(o, o') \in R \times S$ donde $\theta(o.G, o'.G)$ es verdadero:

$$R \bowtie_\theta S = \{(o, o') \mid o \in R \wedge o' \in S \wedge \theta(o.G, o'.G)\}$$

Existe una amplia variedad de predicados θ. Sin embargo, según [GG98] el predicado de *intersección* tiene un rol muy importante pues con él es posible obtener casi todos los restantes.

2.3.1 Procesamiento de consultas espaciales apoyado en un índice espacial

La Figura 2.3 ilustra los pasos que se deben seguir para procesar consultas espaciales, donde se supone el uso de un índice espacial. Sin antes haber descrito un índice espacial, brevemente se explican aquí los elementos básicos para comprender el procesamiento de las consultas. En primer lugar, los objetos que se almacenan en el índice corresponden a una aproximación geométrica de los objetos reales. Usualmente se utiliza el mínimo rectángulo (MBR - Minimun Bounding Rectangle) que es capaz de contener completamente la geometría o atributos espaciales de los objetos (ver Figura 2.7). Normalmente estos MBRs se organizan en estructuras jerárquicas que evitan hacer un recorrido por todos los objetos verificando los predicados espaciales de las consultas.

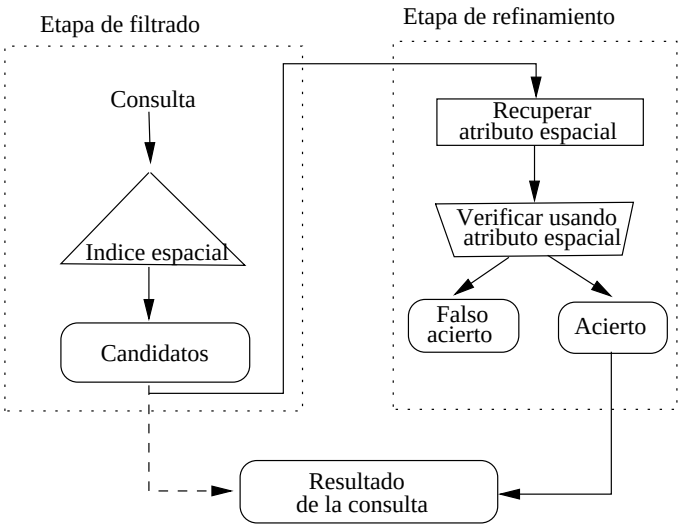


Figura 2.3: Procesamiento de consultas espaciales usando un índice muldimensional o espacial [BKSS94].

Para explicar el procesamiento de las consultas se utiliza como ejemplo una consulta espacial de tipo WQ (window query), definida en esta sección, y que se denomina *q*.

El procesamiento o evaluación de las consultas espaciales se hace siguiendo dos etapas: (i) etapa de filtrado y (ii) etapa de refinamiento. En la etapa de filtrado se utiliza el índice espacial para seleccionar un conjunto de objetos (*candidatos*) cuyos MBRs se intersectan con *q*. En el caso de las consultas WQ (window query), la intersección del MBR de un objeto con *q* no asegura que el objeto pertenezca a la respuesta, ya que no es garantía de que la intersección también ocurra con la componente espacial del objeto. En este caso todos los candidatos deben pasar a la etapa de refinamiento, que se describe más adelante, para decidir si definitivamente el objeto es parte de la respuesta. Sin embargo, existen tipos de consultas en las cuales la pertenencia del objeto a la respuesta se puede decidir en la etapa de filtrado. Por ejemplo, *"encontrar todos los objetos disjuntos a q"*. En este caso si el MBR de un objeto no se intersecta con *q* el objeto pasará directamente a formar parte del resultado de la consulta, pero si el MBR del objeto se intersecta con *q*, entonces se debe verificar en la etapa de refinamiento.

Tal como se anticipó, en la etapa de refinamiento se examina si los objetos *candidatos* efectivamente pertenecen a la respuesta. Para lograr esto último es necesario recuperar el atributo espacial (geometría exacta) de cada objeto perteneciente al conjunto *candidato* y verificar con algún algoritmo geométrico el predicado espacial. Debido a que los datos espaciales tienen una estructura compleja y son muy voluminosos, la etapa de refinamiento puede llegar a incurrir en un costo muy alto de CPU (Unidad Central de Procesos) siendo una componente no despreciable del costo total del procesamiento de la consulta. Por esta razón se ha destinado bastante esfuerzo en tratar de que en la etapa de filtrado se logre un conjunto *candidatos* que sea lo más cercano posible a la respuesta definitiva, evitando al máximo la etapa de refinamiento.

2.4 Métodos de acceso espaciales

Un aspecto importante de los datos espaciales es que la recuperación y actualización de ellos no se hace solamente por medio de atributos descriptivos (atributos alfanuméricos) sino también por medio de los atributos espaciales de los objetos. De esta forma, el procesamiento de las consultas espaciales, tales como las de tipo PQ (point query) o WQ (window query), requieren del acceso rápido a los objetos que satisfacen el predicado espacial sin necesidad de recorrer de manera secuencial todos los objetos de la base de datos. Este requerimiento obliga a contar con métodos de acceso espaciales (multidimensionales) que permitan mantener un índice organizado por el atributo espacial de los objetos [Gut94].

El problema principal de diseño de estos métodos es la ausencia de un orden total entre los objetos espaciales que conserve la proximidad espacial entre ellos [GG98, KGK93]. En otras palabras, no existe una forma de relacionar objetos de un espacio de dos o más dimensiones con objetos en un espacio de una dimensión de manera que, si dos objetos cualquiera se encuentran espacialmente cercanos en el espacio de mayor dimensión, se encuentran también cercanos en el espacio de una dimensión [GG98, NHS84, Sam95].

Se han propuesto varias estructuras de índice. Algunas se han diseñado inicialmente para indexar conjuntos de puntos y luego se han adaptado para manejar objetos espaciales más complejos (líneas y regiones, por ejemplo); en cambio otras se han diseñado ad-hoc para acceder a objetos complejos. Ejemplos de las primeras estructuras son Grid Files [NHS84], K-D-B-tree [Rob81], hB-trees [LS89, LS90] y K-D-tree [Ben75], entre otras. Para el segundo caso, se han propuesto varias estructuras, siendo la más popular R-tree [Gut84], R^+-tree [SRF87], R^*-tree [BKSS90] y Buddy-tree [SK90], entre otras. El lector interesado puede encontrar en [GG98] y en [BBK01] una revisión detallada y completa de los métodos de acceso espaciales. En esta sección explicamos, con cierto nivel de detalle, sólo dos métodos de acceso K-D-B-tree y R-tree (y sus variantes). La razón de esta elección es que los métodos de acceso espacio-temporales que se proponen en esta tesis se basan principalmente en estas dos estructuras.

2.4.1 K-D-B-Tree

Un K-D-B-tree [Rob81] es una estructura jerárquica que corresponde a una combinación de las estructuras K-D-tree [Ben75] y B-tree [BM72]. La estructura B-tree es muy conocida, pero no asi K-D-tree, razón por la cual se explica con más detalle a continuación. K-D-tree es una estructura que reside en memoria principal, utilizada para indexar conjuntos de objetos de tipo *punto*. Un K-D-tree es un árbol binario que representa una subdivisión recursiva del espacio (universo) en subespacios utilizando hiperplanos de *(n-1)*–dimensiones. Los hiperplanos son iso–orientados y sus direcciones alternan entre las *n* posibilidades. Por ejemplo, para $n = 3$, las direcciones de los hiperplanos alternan perpendicularmente a los ejes *x*, *y* y *z*. Cada hiperplano debe contener al menos un punto, el cual se usa para representarlo en el árbol. En la Figura 2.4 se muestra un K-D-tree en un espacio de dos dimensiones. El primer punto que se inserta, p_1, divide el universo en dos subespacios los cuales se encuentran a la izquierda y derecha de la recta (hiperplano) perpendicular al eje *x* y que pasa por la coordenada *y* de p_1. Este punto pasa a formar la raíz del K-D-tree. Luego se inserta el punto p_2 en el subespacio de la derecha definido por p_1. En este caso la dirección del hiperplano de división es perpendicular al eje *y*. Buscar un objeto en un K-D-tree es bastante simple. Se comienza desde la raíz del árbol y se decide por medio de las coordenadas del objeto y

el hiperplano, en que subregión continuar la búsqueda.

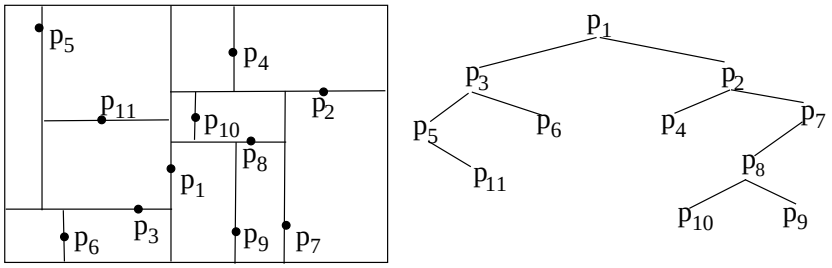


Figura 2.4: Ejemplo de un K-D-tree.

Una de las desventajas del K-D-tree es que la estructura es sensible al orden en que se insertan los objetos. Otra de sus desventajas es que todos los objetos se almacenan en el árbol. Una variante de K-D-tree, conocida como K-D-tree *adapatativo* [BF79], trata de eliminar estas desventajas mediante una elección adecuada de los hiperplanos de tal manera que cada subespacio contenga aproximadamente la misma cantidad de objetos. En esta variante, a pesar que los hiperplanos siguen siendo ortogonales, las direcciones de estos no necesariamente deben alternarse en los diferentes ejes y tampoco contiene un punto. Como resultado de esto los puntos utilizados para definir el hiperplano no forman parte del conjunto de objetos de entrada, los cuales se almacenan en las hojas.

Un K-D-B-tree es una estructura de datos diseñada para residir en memoria secundaria, particiona el universo de manera similar a como procede un K-D-tree *adaptativo*, y los subespacios resultantes se asocian a los nodos del árbol (ver Figura 2.5). Un K-D-B-tree es un árbol perfectamente balanceado y un nodo se almacena en una página o bloque de disco. Al igual que en un B-tree, un K-D-B-tree considera dos tipos de nodos: nodos internos y nodos hojas. Cada nodo interno del árbol representa una región del espacio. Los nodos del mismo nivel corresponden a regiones que son mutuamente disjuntas y cuya unión constituye el universo original. Los nodos hojas almacenan los datos (puntos) que están ubicados en la misma región. En la Figura 2.5 podemos ver que el nodo raíz tiene tres entradas, que dividen el espacio total en igual número de subespacios o subregiones (las subregiones achuradas en el segundo nivel no se encuentran en el nodo). A su vez, de manera recursiva, cada subregión se vuelve a dividir hasta alcanzar los nodos hojas donde se almacenan los objetos.

Cada entrada de un nodo interno almacena tuplas de la forma $\langle$*región*,*ref*$\rangle$ donde *región* representa el subespacio y ref es una referencia al nodo hijo que contiene todas las subregiones de la región definida en la entrada. Por su parte los nodos hojas almacenan colecciones de tuplas del tipo $\langle punto, oid \rangle$. Aquí *oid* se refiere a la dirección de la tupla o registro dentro de la base de datos.

2.4.1.1 Consultas con K-D-B-tree

Suponemos que nuestra consulta corresponde a una del tipo WQ (window query) y, por lo tanto, lo que se quiere es recuperar todos los objetos (puntos) que están contenidos en el rango espacial q. El procedimiento parte desde la raíz verificando las entradas cuya *región* se intersecta espacialmente con q. El algoritmo, recursivamente, desciende por todas aquellas ramas del árbol en que *región* se

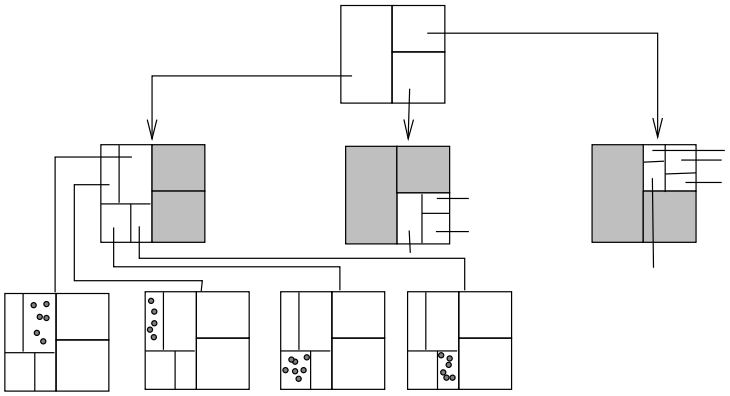


Figura 2.5: Ejemplo de un K-D-B-tree (tomado de [Rob81]).

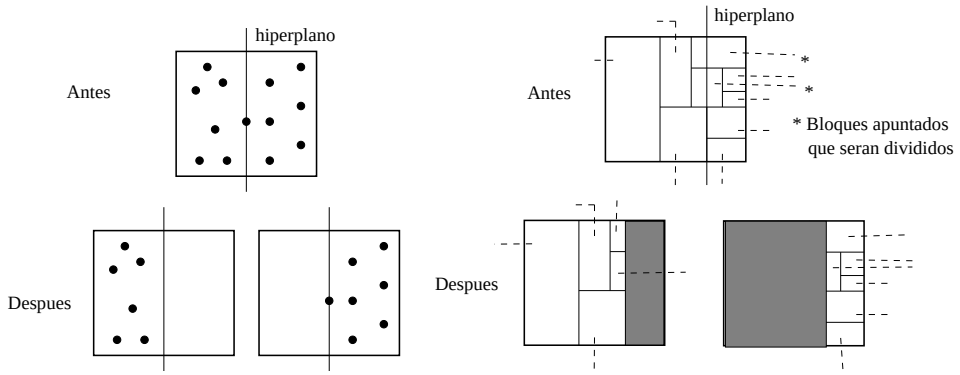
intersecta espacialmente con *q*. Cuando se alcanza un nodo hoja, se recuperan todos los objetos que se intersectan con *q* constituyéndose en parte de la respuesta. Como es posible ver, para procesar una consulta puede ser necesario seguir varios caminos o rutas del K-D-B-tree, cuestión que no sucede cuando la consulta, es decir *q*, es de tipo PQ (point query).

2.4.1.2 Inserción en un K-D-B-tree

Para insertar un nuevo punto, primero se hace una búsqueda del punto y se determina la hoja donde debería quedar almacenado; si existe espacio suficiente, entonces el punto se inserta en la hoja, en caso contrario la hoja se divide en dos quedando en cada nodo aproximadamente la mitad de los puntos, ver Figura 2.6 (a). Si el nodo padre (nodo interno) no tiene suficiente espacio para alojar una nueva entrada que define una región, entonces se obtiene un nuevo bloque y el nodo padre se divide mediante un hiperplano (ver Figura 2.6 (b)). Las entradas son distribuidas entre las dos páginas y la partición o división es propagada hacia la raíz del árbol. La división de un nodo interno también puede afectar regiones en los niveles inferiores del árbol, las cuales tienen que ser divididas. En la Figura 2.6 (b) los nodos cuyas referencias aparecen marcadas con * también se tienen que particionar como consecuencia de la partición del nodo interno. Esta estrategia de partición no permite garantizar una utilización mínima del almacenamiento, convirtiéndose esta caracterśtica en la principal desventaja de K-D-B-tree.

2.4.1.3 Eliminación en un K-D-B-tree

La eliminación de un objeto (punto) también se lleva a cabo de forma similar a como se realiza en un B-tree, es decir, primero se busca el nodo hoja donde se encuentra el punto, y luego se elimina. En el caso de que la hoja quede con menos objetos que un valor de umbral establecido para K-D-B-tree, entonces se pueden mezclar varios nodos hojas. La unión de varios nodos hojas provoca la eliminación de, al menos, un hiperplano en el nodo padre. Si en éste último ocurre que el número de entradas es inferior a un número mínimo establecido, la eliminación se propaga hacia la raíz del árbol.



(a) Partición de un nodo hoja (b) Partición de un nodo interno

Figura 2.6: Ejemplo de partición de los nodos de un K-D-B-tree.

2.4.2 R-tree

El R-tree es uno de los métodos de acceso multidimensional o espacial más estudiados y, por ende, uno de los más conocidos y que ha sido adoptado por productos de SABD, tales como Oracle y Postgres.

Un R-tree es una extensión natural de un B-tree [BM72] para objetos espaciales (puntos y regiones) [Gut84, SRF97]. Cada nodo corresponde a una página o bloque de disco. Los nodos hojas de un R-tree contienen entradas de la forma $\langle MBR, oid \rangle$ donde *oid* es el identificador del objeto espacial en la base de datos y MBR (Minimun Bounding Rectangle) es un rectángulo multidimensional que corresponde al mínimo rectángulo que encierra al objeto espacial. Los nodos internos (nodos no-hojas) contienen entradas de la forma $\langle MBR, ref \rangle$, donde ref es la dirección del correspondiente nodo hijo en el R-tree y MBR es el rectángulo mínimo que contiene a todos los rectángulos definidos en las entradas del nodo hijo.

Sea M el número máximo de entradas que se pueden almacenar en un nodo y sea $m \leq \frac{M}{2}$ un parámetro especificando el número mínimo de entradas en un nodo. Un R-tree satisface las siguientes propiedades [Gut84]:

1. Cada nodo contiene entre m y M entradas a menos que corresponda a la raíz.
2. Para cada entrada $\langle MBR, oid \rangle$ en un nodo hoja, MBR es el mínimo rectángulo que contiene (espacialmente) al objeto.
3. Cada nodo interno tiene entre m y M hijos, a menos que sea la raíz.
4. Para cada entrada de la forma $\langle MBR, ref \rangle$ de un nodo interno, MBR es el rectángulo más pequeño que contiene espacialmente los rectángulos definidos en el nodo hijo.
5. El nodo raíz tiene al menos dos hijos, a menos que sea una hoja.
6. Todas las hojas se encuentran al mismo nivel.

La Figura 2.7 muestra las particiones del espacio a diferentes niveles. Por ejemplo los MBRs $R1$ y $R2$ forman una partición en un primer nivel. A su vez los MBRs $R6$ y $R7$ conforman una

partición del subespacio $R2$ en un segundo nivel y así sucesivamente. La Figura 2.8 muestra el conjunto de rectángulos de la Figura 2.7 organizados en un R-tree.

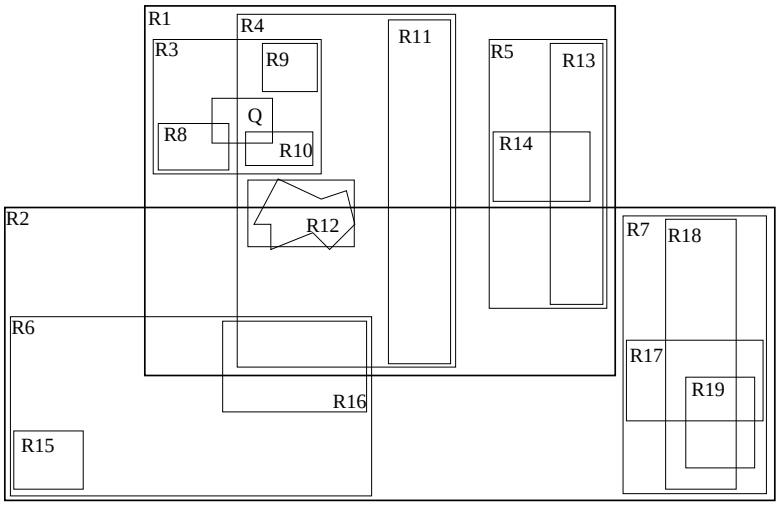


Figura 2.7: Agrupaciones de MBRs generadas por un R-tree. .

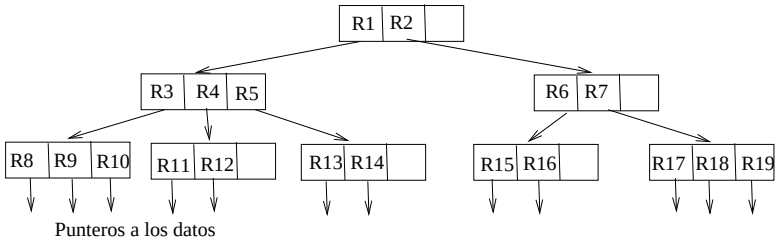


Figura 2.8: R-tree de los rectángulos de la Figura 2.7.

La altura de un R-tree que almacena N claves es a lo más $\lceil \log_m N \rceil - 1$, ya que el número de hijos es al menos m. El número máximo de nodos es $\lceil \frac{N}{m} \rceil + \lceil \frac{N}{m^2} \rceil + \cdots + 1$ [Gut84]. La utilización del almacenamiento (peor caso) para todos los nodos, excepto la raíz, es $\frac{m}{M}$ [Gut84]. Los nodos tienden a tener más de m entradas, lo que permite que la altura del árbol disminuya y mejore la utilización del almacenamiento. En la Sección 2.4.6 se presenta un análisis más detallado del rendimiento de un R-tree, tanto desde el punto de vista del almacenamiento ocupado como de la eficiencia de las consultas.

2.4.2.1 Búsqueda en un R-tree

El algoritmo de búsqueda[1] en un R-tree es similar al utilizado por un B-tree, es decir, la búsqueda se inicia en la raíz del árbol y se desciende por él hasta alcanzar un nodo hoja. En un nodo

[1] El algoritmo de búsqueda corresponde a la etapa de filtrado de la Figura 2.3 para una consulta de tipo WQ (window query)

interno, la decisión sobre cuáles hijos continuar la búsqueda, se toma verificando si el MBR de la correspondiente entrada del nodo y Q se intersectan. Producto de que las particiones en los diferentes niveles no son disjuntas, puede ser necesario recorrer varios caminos desde un nodo interno a las hojas en una consulta. Por ejemplo, en el primer nivel Q solamente se intersecta con el rectángulo $R1$ (ver Figura 2.7). Sin embargo, en el segundo nivel se intersecta con $R3$ y $R4$ (ver Figura 2.7) lo que implica que los punteros asociadas a las particiones $R3$ y $R4$ se deben explorar hasta llegar a las hojas. Cuando se alcanza un nodo hoja, el algoritmo verifica si el MBR del objeto se intersecta con Q, y de intersectarse el objeto pasa a ser un candidato del resultado de la consulta. El procedimiento de búsqueda se describe en más detalle en el Alg. 2.1.

```
1:  BuscarEnRtree(T, Q) {L es un conjunto de punteros a objetos cuyos MBRs se intersectan con Q. E representa
    una entrada de un nodo. E.MBR se usa para referirse al MBR de una entrada y E.p para indicar, ya sea una
    referencia de un nodo interno del R-tree, o una referencia donde se almacena el atributo espacial o geometría
    exacta del objeto}
2:  L = ∅
3:  for cada entrada E ∈ T do
4:     if E.MBR ∩ Q ≠ ∅ then
5:        if T es una hoja then
6:           L = L ∪ {E.p}
7:        end if
8:     else
9:        L = L ∪ BuscarEnRtree(E.p, Q)
10:    end if
11: end for
12: return L
```

Alg. 2.1: Algoritmo de búsqueda en un R-tree.

2.4.2.2 Inserción en un R-tree

La inserción en un R-tree es similar a la inserción en un B-tree en el sentido que el elemento se inserta en las hojas, produciendo eventualmente una división del nodo hoja y propagando un elemento hacia el padre, el que a su vez también se puede dividir y propagar un elemento a su correspondiente padre. El proceso puede continuar recursivamente hasta llegar a la raíz, la cual también se puede dividir y generar una nueva raíz. El Alg. 2.2 describe el procedimiento de inserción.

```
1:  InsertarEnRtree(T, E)
2:  Sea L = SeleccionarHoja(T, E) {Alg. 2.3} { L contiene el nodo hoja donde debería ser insertada la entrada E}
3:  if L tiene espacio suficiente para contener a E then
4:     Almacenar la entrada E en L y terminar
5:  else
6:     LL = DividirNodo(L) {Se obtienen dos nodos hojas L y LL que contienen todas las entradas de L y la entrada
       E}
7:     AjustarArbol(L, LL) {Alg. 2.4}
8:     if la propagación provoca dividir la raíz then
9:        Crear una nueva raíz cuyos hijos son los nodos resultantes
10:    end if
11: end if
```

Alg. 2.2: Algoritmo para insertar un MBR en un R-tree.

```
1: SeleccionarHoja(N, E)
2: if N es hoja then
3:    return N
4: else
5:    Buscar las entradas en N cuyos rectángulos necesitan crecer menos para contener E.MBR. Si hay varios, elegir
      uno entre aquellos cuyos rectángulos que tengan la menor área. Sea F dicha entrada
6:    return SeleccionarHoja(F.p, E)
7: end if
```

Alg. 2.3: Algoritmo para seleccionar el nodo hoja del R-tree sobre el cual hacer la inserción.

```
1: AjustarRtree(N, NN)
2: if N no es la raíz then
3:    Sea P el nodo padre de N y E_N la entrada de N en P
4:    Ajustar E_N.MBR de tal manera que cubra todos los rectángulos de las entradas de N.
5:    Sea E_NN una nueva entrada con E_NN.p apuntando a NN y E_NN.MBR los limites del rectángulo que cubre todos
      los rectángulos en NN.
6:    if P tiene espacio then
7:       Agregar E_NN a P y terminar
8:    else
9:       PP = DividirNodo(P, E_NN) { Las entradas de P, más E_NN se reparten entre los nodos P y PP}
10:      AjustarRtree(P, PP)
11:   end if
12: end if
```

Alg. 2.4: Algoritmo para ajustar las áreas de los rectángulos de un R-tree, propagando hacia arriba una entrada cuando sea necesario.

2.4.2.3 Eliminación en un R-tree

El proceso de eliminación de una entrada E desde un R-tree comienza buscando (Alg. 2.6) el nodo hoja en el cual se encuentra dicha entrada E (Alg. 2.5). Luego se elimina del nodo hoja y, si dicho nodo queda con menos entradas que las permitidas, éstas se reubican. La eliminación se propaga hacia la raíz del árbol, si es necesario ajustando las áreas cubiertas por los rectángulos a partir de la raíz hacia la hoja (Alg. 2.7).

```
1: EliminarEnRtree(T, E)
2: L =EncontrarHoja(T, E) {L apunta al nodo hoja del R-tree que contiene la entrada del objeto a eliminar}
3: if E no se encuentra then
4:    return "ERROR"
5: else
6:    Remover E de L
7:    CondensarRtree(L) {Alg. 2.7}
8:    if después de condensar el R-tree, la raíz tiene un sólo hijo then
9:       Dejar como nueva raíz del R-tree al hijo
10:   end if
11: end if
```

Alg. 2.5: Algoritmo para eliminar un objeto espacial del R-tree.

```
1:  EncontrarHoja(T, E)
2:  if T no es un nodo hoja then
3:     for cada entrada F ∈ T do
4:        if F.MBR ∩ E.MBR ≠ ∅ then
5:           K = EncontrarHoja(F.p, E)
6:           if K ≠ " NO ENCONTRADO" then
7:              return K
8:           end if
9:        end if
10:    end for
11: else
12:    for cada entrada F ∈ T do
13:       if F coincide con E then
14:          return T
15:       end if
16:    end for
17: end if
18: return "NO ENCONTRADO"
```

Alg. 2.6: Algoritmo para encontrar el nodo hoja que contiene la entrada E del objeto a eliminar.

2.4.2.4 Actualización y otras operaciones

Si los límites de un objeto espacial se modifican de tal forma que su MBR cambie, su entrada debe ser eliminada, actualizada y reinsertada en el árbol. En un R-tree también es posible realizar otros tipos de búsquedas útiles. Por ejemplo *"encontrar todos los objetos espaciales contenidos completamente en un área de búsqueda"* o *"todos los objetos que contienen una determinada área"*. Estas operaciones se pueden implementar mediante simples variaciones de los algoritmos vistos anteriormente.

2.4.2.5 División de un nodo en un R-tree

Cuando se agrega una nueva entrada en un nodo que se encuentra lleno, es decir, contiene M entradas, es necesario repartir las $M+1$ entradas entre dos nodos. La división debería realizarse de tal forma que la posibilidad de accesar ambos nodos en búsquedas futuras sea lo más pequeña posible. Ya que la decisión de visitar un nodo depende de si el área cubierta por su rectángulo se sobrepone con el área de búsqueda, el área total cubierta por los dos rectángulos después de dividir un nodo debería ser minimizada. En la Figura 2.9 (a) se considera una mala división pues el área cubierta por los rectángulos es mucho mayor que en aquella considerada una buena división (Figura 2.9 (b)).

El mismo criterio se usa en el procedimiento *SeleccionarHoja()* (Alg. 2.3) para decidir dónde insertar una nueva entrada de índice, es decir, en cada nivel del árbol, el subárbol elegido es aquel cuyo rectángulo necesita expandirse menos para contener una nueva entrada. En [Gut84] se proponen varias heurísticas para repartir las $M+1$ entre dos nodos.

Algoritmo exhaustivo. La forma más simple de encontrar una división del nodo de tal forma que las áreas de las particiones sean mínimas, es generar todos los posibles grupos y elegir el mejor. Sin embargo el número de posibilidades es aproximadamente 2^{M+1} y un valor razonable de M es

```
1:  CondensarRtree(N)
2:  Q = ∅
3:  while N no es la raíz do
4:      Sea P el padre de N y E_N la entrada de N en P
5:      if el número de entradas en N es menor que m then
6:          Eliminar E_N de P
7:          Q = Q ∪ {E_N}
8:      else
9:          Ajustar E_N.MBR para que contenga todas las entradas de N
10:     end if
11:     N = P
12: end while
13: Reinsertar todas las entradas de los nodos en Q. Las entradas que provienen de nodos hojas eliminados son
    reinsertadas en las hojas del árbol, tal como se describió en el algoritmo InsertarEnRtree (Alg. 2.2). Las entradas
    que provienen de nodos eliminados de mayor nivel (cercanos a la raíz) deben ser ubicados más arriba en el árbol,
    de tal manera que las hojas de sus subárboles queden al mismo nivel que las restantes hojas del árbol
```

Alg. 2.7: Condensar R-tree después de haber eliminado un entrada E.

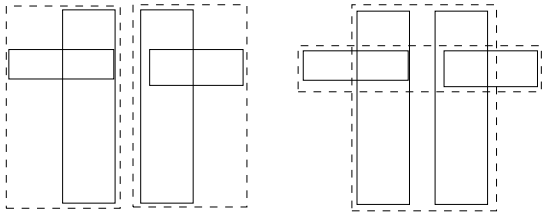


Figura 2.9: Distintas posibilidades de división de un nodo. (a) Mala división, (b) Buena división.

50 [Gut84][2], de forma que el número de posibilidades es demasiado grande.

Algoritmo de complejidad cuadrática. Este algoritmo intenta encontrar una división tal que las áreas de los MBRs que contienen a los MBRs de cada grupo sean mínimas. Este algoritmo no garantiza encontrar una división óptima, es decir, una en que el área sea la menor posible. El costo es cuadrático en M y lineal en el número de dimensiones. El algoritmo, mediante el procedimiento *SeleccionarSemilla()* (ver Alg. 2.10), elige dos de las $M+1$ entradas, las que forman los primeros elementos (semillas) de los dos nuevos grupos. Sean $E1$ y $E2$ estas entradas y r el MBR que contiene a los rectángulos $E1.MBR$ y $E2.MBR$. El par de entradas $\langle E1, E2\rangle$ se elige de entre todas aquellas tal que el área de r presenta la mayor área inutilizada, es decir, el área de r menos el área de $E1.MBR$ y de $E2.MBR$ es la mayor. Luego, el resto de las entradas se agregan una a una a los grupos que correspondan. En cada paso se calcula el área de expansión requerida para agregar cada entrada restante a cada grupo, y se elige la entrada que presenta la mayor diferencia entre los dos grupos. El procedimiento se describe en los Algs. 2.8, 2.9 y 2.10.

[2]Considerando un tamaño de página de 1.024 bytes y que cada entrada necesita de 20 bytes (16 para almacenar los límites del rectángulo y 4 bytes para almacenar un puntero a un bloque), entonces en un bloque se pueden almacenar 50 entradas.

```
1:  DivisiónCuadrática()
2:  Sea (E1, E2) =SeleccionarSemilla()
3:  G1 = {E1} {Elementos del grupo 1}
4:  G2 = {E2} {Elementos del grupo 2}
5:  while queden entradas por asignar do
6:     if la cantidad de entradas en un grupo + la cantidad de entradas sin asignar ≤ m then
7:        Asignar todas las entradas sin asignar al grupo y terminar
8:     end if
9:     Sea E =SeleccionarSiguienteEntrada()
10:    Sea d1 =Area((G1 ∪ {E}).MBR) − Area(G1.MBR)
11:    Sea d2 =Area((G2 ∪ {E}).MBR) − Area(G2.MBR)
12:    if d1 = d2 then
13:       Seleccionar el grupo de acuerdo a los siguientes criterios: El grupo con menor área. Si el empate persiste,
          entonces seleccionar el grupo con menos entradas. Si los grupos tienen el mismo número de entradas , elegir
          cualquiera. Sea i el grupo elegido
14:       Gi = Gi ∪ {E}
15:    else if d1 < d2 then
16:       G1 = G1 ∪ {E}
17:    else
18:       G2 = G2 ∪ {E}
19:    end if
20: end while
```

Alg. 2.8: Algoritmo de orden cuadrático para dividir un nodo de un R-tree.

```
1: SeleccionarSemilla()
2: Para cada par de entradas E1, E2 obtener el rectángulo r el cual incluye a E1.MBR y a E2.MBR
3: d =Area(r) − Area(E1.MBR) − Area(E2.MBR)
4: Elegir y retornar el par con mayor valor de d
```

Alg. 2.9: Algoritmo para seleccionar los rectángulos semillas.

Algoritmo de complejidad lineal. Este algoritmo es lineal con respecto a M y al número de dimensiones. Este algoritmo es similar al de costo cuadrático sólo que usa una forma diferente de seleccionar las entradas semillas. Este procedimiento obtiene los pares de objetos más distantes a lo largo de cada dimensión, y luego elige como par semilla a aquel par cuyos objetos se encuentran más alejados. En cuanto al procedimiento *SeleccionarSiguienteEntrada()*, este algoritmo selecciona cualquiera de las entradas restantes que aún no se han asignado.

2.4.3 R$^+$-tree

Uno de los problemas de R-tree es que para procesar consultas del tipo WQ (window query) es probable que sea necesario seguir más de un camino desde nodos internos a las hojas del árbol lo

```
1: SeleccionarSiguienteEntrada()
2: for cada entradas E sin asignar a un grupo do
3:    Sea d1 el área que es necesario aumentar en el grupo 1 de tal manera de cubrir la entrada E
4:    Sea d2 el área que es necesario aumentar en el grupo 2 de tal manera de cubrir la entrada E
5: end for
6: return la entrada cuya diferencia entre d1 y d2 sea la máxima. Si hay más de una elegir cualquiera
```

Alg. 2.10: Algoritmo para seleccionar siguiente entrada.

que se explica por las intersecciones de los MBR's de los nodos. Por ejemplo una consulta cuyo rectángulo de búsqueda WQ (window query) cae dentro de un área con *k* nodos que se sobreponen en un cierto nivel *h* implica seguir, por lo menos, *k* caminos desde el nivel *h* hasta las hojas. En la Figura 2.7 el rectángulo *Q* intersecta un área de sobreposición de *R*3 y *R*4 y, por lo tanto, la búsqueda se debe realizar en el subárbol apuntado por la entrada *R*3 y en el subárbol apuntado por la entrada *R*4. El R^+-tree [SRF87] trata de aminorar este problema evitando las áreas de sobreposición en los nodos intermedios insertando un objeto en múltiples hojas si es necesario. La Figura 2.10 muestra un agrupamiento bajo la estructura R^+-tree. Se puede observar en la Figura 2.10 que el rectángulo *G* ha sido dividido en dos subrectángulos de los cuales el primero está contenido en el nodo *A* y el segundo en el nodo *P*. De esta forma los objetos cuyos MBR se intersectan con más de un rectángulo se insertan en varios nodos.

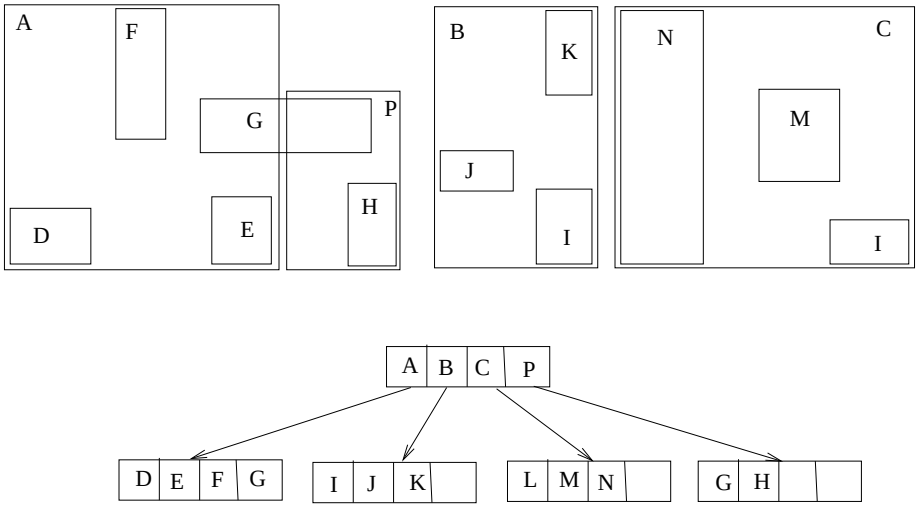


Figura 2.10: Rectángulos agrupados bajo la forma de un R^+tree.

Dado que no existen áreas de sobreposición, una búsqueda de un punto, es decir, una consultas de tipo PQ (point query) requiere seguir un sólo camino desde la raíz hasta una hoja. En este tipo de consultas el R^+-tree tiende a superar al R-tree. Sin embargo, las consultas de tipo WQ (window query) deben recorrer varios caminos de todas maneras (similar a R-tree).

La inserción de un nuevo objeto (rectángulo) en un R^+-tree se realiza buscando el rectángulo en el árbol y agregándolo en una hoja. La diferencia más importante con el correspondiente algoritmo de R-tree es que el ractángulo a insertar puede ser agregado en más de un nodo hoja. Este es el caso del objeto *G* de la Figura 2.10, el cual es insertado en los subárboles *A* y *P* ya que *G* se sobrepone con ambas regiones. La inserción puede requerir de mucho esfuerzo, ya que al insertar un fragmento del objeto, se puede provocar que haya sobreposición de MBRs en los nodos internos y, con el propósito de eliminarla, algunos MBRs tienen que ser divididos y reinsertados en el árbol. Al igual que en un R-tree, si un nodo hoja se encuentra lleno al momento de insertar un nuevo objeto, entonces será necesario dividirlo. Sin embargo, una diferencia importante con R-tree es que la división no sólo puede propagarse hacia la raíz del árbol, sino que eventualmente hacia las hojas de éste. Esto conlleva a que el proceso de inserción puede llegar a ser muy complicado provocando mucha fragmentación de los MBRs de los datos.

La operación de eliminación en un R^+-tree es mucho más complicada que la de un R-tree. En este tipo de operación se requiere ubicar todos los nodos hojas en las cuales se han almacenado fragmentos de los objetos y eliminarlos. Si la utilización del almacenamiento de un nodo cae bajo un valor de umbral dado, es necesario combinarlo con sus nodos hermanos o reorganizar el árbol. Lamentablemente esto no siempre se puede realizar, y esta es la razón por la cual R^+-tree no puede garantizar una utilización mínima del almacenamiento.

2.4.4 R*-tree

Basado en un cuidadoso estudio discutido en [BKSS90] se demostró que bajo diferentes distribuciones de datos el R-tree presenta comportamientos distintos. En particular, se confirmó que la fase de inserción es crítica en el rendimiento de la etapa de búsqueda. El R^*-tree, por lo tanto, introduce una política de inserción llamada reinserción forzada, la cual consiste en no dividir un nodo en forma inmediata cuando éste se llena. En lugar de ello se propone eliminar primero p entradas del nodo y reinsertarlas en el árbol. Por otra parte, los algoritmos para la división de los nodos propuestos en R-tree sólo tratan de minimizar el área que es cubierta por una región [Gut84], en cambio los algoritmos propuestos para R^*-tree toman en cuenta la optimización de los siguientes parámetros:

1. El área de sobreposición entre regiones en el mismo nivel del árbol debería ser minimizada. Mientras menor sea esta área, menor es la probabilidad de seguir múltiples caminos para una búsqueda.

2. Los perímetros de las regiones deberían ser minimizados. La forma rectangular preferida es el cuadrado, ya que es la representación rectangular más compacta.

3. La utilización del almacenamiento debiera ser maximizada. Mayor utilización del almacenamiento generalmente reduce el costo en una operación de búsqueda, dado que la altura del árbol se reduce en la medida que los nodos tienden a llenarse.

Con estos cambios, en [BKSS90] se reportaron resultados de rendimiento favorables al R^*-tree en hasta 50% comparado con el R-tree básico. Los resultados también mostraron que la política de reinserción forzada puede mejorar la utilización del almacenamiento. Sin embargo, esta política puede incurrir en costos de CPU importantes, las cuales aumentan en la medida que la capacidad de los nodos también lo hace. Las operaciones de búsqueda y eliminación de un R^*-tree no cambian con respecto a las correspondientes de un R-tree.

2.4.5 Reunión espacial usando R-tree

La operación de reunión (join) es una de las operaciones fundamentales y a la vez una de las más costosas de evaluar en SABD relacionales [ME92]. Estas características se agudizan cuando la operación de reunión se debe realizar sobre conjuntos de objetos espaciales [BKS93, TSS98a]. Una reunión espacial se puede definir de la siguiente manera: dadas dos colecciones de objetos espaciales R y S y un predicado espacial θ, encontrar todos los pares de objetos $(o,o') \in R \times S$ donde $\theta(o.G, o'.G)$ es verdadero, es decir,

$$R \bowtie_\theta S = \{(o,o') \mid o \in R \wedge o' \in S \wedge \theta(o.G, o'.G)\}$$

Tal como comentamos en la sección 2.3 existen varios predicados θ posibles, siendo el más importante la intersección espacial la cual se discute en esta sección. En [BKS93] se definen varios tipos de reunión espacial:

1. `MBR-Reunión-Espacial:` Obtiene todos los pares $\langle r_i.Oid, s_i.Oid \rangle$ tal que $r_i.MBR \cap s_j.MBR \neq \emptyset$
2. `ID-Reunión-Espacial:` Obtiene todos los pares de objetos $\langle r_i.Oid, s_i.Oid \rangle$ tal que $r_i.G \cap s_j.G \neq \emptyset$
3. `Objeto-Reunión-Espacial:` Obtiene $r_i.G \cap s_j.G$ con $r_i.G \cap s_j.G \neq \emptyset$

Notar que `MBR-Reunión-Espacial` puede ser utilizado para implementar la etapa de filtrado de los otros dos tipos de reunión, es decir, `ID-Reunión-Espacial` y `Objeto-Reunión-Espacial`. Esto explica la importancia de `MBR-Reunión-Espacial` y el por qué nos concentramos en este tipo de reunión espacial.

Se asume que tanto los elementos de R como de S se encuentran indexados mediante un R-tree (o alguna de sus variantes), sean $R1$ y $R2$ estos índices. Para una primera aproximación consideremos que $R1$ y $R2$ tienen la misma altura. En primer lugar se recuperan las raíces de ambos R-trees. Luego se revisan todos los pares de entradas verificando si los MBRs de ambas entradas se intersectan. Para aquellos pares de entradas en donde existe intersección espacial se desciende recursivamente en ambos subárboles hasta alcanzar los nodos hojas. En los nodos hojas se verifican los pares de entradas reportando los identificadores de los objetos cuando se produce una intersección de los MBRs. Este primer procedimiento se describe en el Alg. 2.11.

```
 1: SJ1(R1, R2) {R1 y R2 corresponden a nodos de R-tree}
 2: for all E1 ∈ R1 do
 3:    for all E2 ∈ R2 con E1.MBR ∩ E2.MBR ≠ ∅ do
 4:       if R1 es hoja then
 5:          output(E1.Oid, E2.Oid)
 6:       else
 7:          ReadPage(E1.ref)
 8:          ReadPage(E2.ref)
 9:          SJ1(E1.ref, E2.ref)
10:       end if
11:    end for
12: end for
```

Alg. 2.11: Algoritmo para procesar la reunión espacial utilizando R-trees de la misma altura.

Ahora se discute el caso más general, el cual considera R-trees de diferente altura. En este caso el algoritmo no presenta diferencias para los nodos internos del R-tree, es decir, se desciende coordinadamente por las entradas en las cuales existe intersección entre los correspondientes MBRs. Cuando se produce el caso en que en un R-tree se alcanzó un nodo interno (N_{R1}) y en el otro un nodo hoja (N_{R2}), entonces se realiza una reunión espacial entre N_{R1} y N_{R2}. La idea ahora es ejecutar consultas de tipo WQ (window query) sobre el subárbol apuntado por N_{R1} usando como consultas los MBRs de las entradas de N_{R2}. Estas ideas se resumen en el Alg. 2.12.

```
 1: SJ(R1, R2) {R1 y R2 corresponden a nodos de R-tree}
 2: for all E1 in R1 do
 3:    for all E2 in R2 do
 4:       if E1.MBR ∩ E2.MBR ≠ ∅ then
 5:          if R1 y R2 son hojas then
 6:             output(E1.Oid, E2.Oid)
 7:          else if R1 es hoja then
 8:             ReadPage(E2.ref)
 9:             SJ(R1, E2.ref)
10:          else if R2 es hoja then
11:             ReadPage(E1.ref)
12:             SJ(E1.ref, R2)
13:          else
14:             ReadPage(E1.ref)
15:             ReadPage(E2.ref)
16:             SJ(E1.ref, E2.ref)
17:          end if
18:       end if
19:    end for
20: end for
```

Alg. 2.12: Algoritmo para procesar la reunión espacial utilizando R-trees de diferente altura.

2.4.6 Modelos de costo de R-tree

En esta sección se describen con bastante detalle los modelos de costos establecidos para el R-tree, tanto para determinar el espacio requerido como para predecir el rendimiento de las consultas de tipo WQ (window query) y de reunión espacial. Los resultados de estos modelos son utilizados más adelante para definir los modelos de costos de nuestros métodos de accesos SEST-Index, SEST_L y establecer un modelo general para métodos de acceso espacio-temporales basados en snapshots y eventos.

2.4.6.1 Estimación del espacio ocupado por un R-tree

Sea N el número de objetos que se van a almacenar. De acuerdo con [FSR87], la altura (h) de un R-tree se define por la Ec. (2.1), donde $f = cc \cdot M$ representa el número promedio de entradas en un R-tree cuya capacidad máxima es M y su porcentaje de utilización es cc.

$$h = 1 + \left\lceil \log_f \frac{N}{f} \right\rceil \tag{2.1}$$

Ya que el número de entradas en un nodo (bloque o página de disco) es aproximadamente f, es posible asumir que el número promedio de hojas es $N_1 = \left\lceil \frac{N}{f} \right\rceil$ y que el número de nodos en el nivel inmediatamente superior será $N_2 = \left\lceil \frac{N_1}{f} \right\rceil$. Considerando que el nivel h se encuentra en la raíz y que el nivel 1 corresponde a las hojas, el número promedio de nodos de un R-tree en el nivel j está dado por la ecuación $N_j = \left\lceil \frac{N}{f^j} \right\rceil$ [TS96]. De esta forma, el número promedio de nodos de un R-tree está dado por la Ec. (2.2)

$$TN = \sum_{j=1}^{h} \left\lceil \frac{N}{f^j} \right\rceil \leq h + \left\lfloor N \cdot \frac{(f^h - 1)}{f^h \cdot (f-1)} \right\rfloor \approx \log_f N + \frac{N}{f} \tag{2.2}$$

2.4.6.2 Estimación del rendimiento de las consultas usando R-tree

Existen varias propuestas para estimar el rendimiento de una consulta espacial de tipo WQ (window query) sobre un R-tree [FSR87, KF93, PSTW93, TS96, TSS00]. Uno de los modelos más conocidos es el descrito en [TS96, TSS00] cuya característica más destacada es su capacidad de estimar el rendimiento del R-tree sin necesidad de construirlo ya que sólo depende de la cantidad de objetos, de la densidad inicial de los mismos y de la capacidad promedio de un nodo de un R-tree (f). En este modelo, la densidad D de un conjunto de reactángulos se define como la razón entre el área total ocupada por los rectángulos y el espacio de trabajo (work space). Si se considera como espacio de trabajo un hiper-rectángulo $[0,1)^n$, la densidad $D(N,s)$ está dada por la Ec. (2.3) donde s_i es la longitud promedio de los hiper-rectángulos (cuadrados) en la dimensión i.

$$D(N,s) = \sum_{N} \prod_{i=1}^{n} s_i = N \cdot \prod_{i=1}^{n} s_i \tag{2.3}$$

De acuerdo a [TS96], la cantidad de nodos accesados de un R-tree por una consulta espacial de tipo WQ de n dimensiones está dada por la Ec. (2.4) .

$$DA_n(q) = \sum_{j=1}^{h-1} \left\{ N_j \cdot \prod_{k=1}^{n} (s_{j,k} + q_k) \right\} \tag{2.4}$$

El número N_j de nodos en el nivel j del R-tree queda establecido por:

$$N_j = \frac{N}{f^j} \tag{2.5}$$

y la longitud promedio en cada dimensión de cada MBR de cada nodo (asumiendo rectángulos cuadrados) en el nivel j se obtiene por medio de la siguiente fórmula [TS96]:

$$S_{j,k} = \left(\frac{D_j}{N_j} \right)^{\frac{1}{n}} \tag{2.6}$$

Donde $D_0 = D$ (ver Ec. (2.3)), es decir la densidad de los rectángulos de los objetos espaciales y D_j denota la densidad de los MBRs(rectángulos) en el nivel j y es obtenido como una función de la densidad en el nivel inmediatamente inferior (D_{j-1}), es decir,

$$D_j = \left(1 + \frac{D_{j-1}^{\frac{1}{n}} - 1}{f^{\frac{1}{n}}} \right)^n \tag{2.7}$$

2.4.6.3 Modelo de costo para la reunión espacial usando R-tree

En [TSS98b] se propone un modelo que estima el costo de un operación de reunión espacial. Dicho modelo asume un espacio n-dimensional y que existen dos conjuntos espaciales de cardinalidad N_{R1} y N_{R2} respectivamente, y que cada uno de ellos cuenta con un R-tree, R_1 y R_2. El objetivo del modelo es obtener el número promedio de nodos (NA) accesados o número promedio de bloques de disco leídos (DA) por una operación de reunión espacial. La diferencia entre NA y DA se debe a que en el cálculo de NA no se considera un sistema de buffer mientras que si se considera para DA. De esta forma la desigualdad $DA \leq NA$ siempre se cumple.

Suponiendo que R_1 y R_2 tienen la misma altura h y que las dos raíces se almacenan en memoria, el valor de NA se define por la siguiente ecuación [TSS98b]:

$$NA_{total}(R_1,R_2) = \sum_{j=1}^{h-1}\{NA(R_1,j) + NA(R2,j)\} \tag{2.8}$$

donde $NA(R_i,j)$ se define por la Ec. (2.9), j representa el nivel del R-tree ($1 \leq j \leq h-1$), $N_{R_i,j}$ el número promedio de nodos en el nivel j del R-tree i, y $s_{R_i,j}$ la longitud promedio en cada dimensión de los MBR del nivel j del R-tree i. Los valores de $N_{R_i,j}$ y $s_{R_i,j}$ se obtienen con la Ec. (2.5) y (2.6), respectivamente.

$$NA(R_i,j) = N_{R_2,j} \cdot N_{R_1,j} \cdot \prod_{k=1}^{n} min\{1,(s_{R_1,j,k} + s_{R_2,j,k})\} \tag{2.9}$$

Al considerar la utilización de un buffer por cada R-tree para almacenar los nodos más recientemente accesados del árbol, el número de lecturas de bloques de disco disminuye y se obtienen por medio de la Ec. (2.10).

$$DA_{total}(R_1,R_2) = \sum_{j=1}^{h-1}\{DA(R_1,j) + DA(R_2,j)\} \tag{2.10}$$

donde $DA(R_i,j)$ se define por la ecuación 2.11.

$$DA(R_2,j) = N_{R_2,j} \cdot \sum intersect(N_{R_1,j}, s_{R_1,j+1}, s_{R_2,j}) \tag{2.11}$$

y $DA(R_1,j) \approx NA(R_1,j)$

En [TSS98b] también se proponen modelos para NA y DA teniendo en cuenta que las alturas de ambos R-tree son diferentes. Asumiendo que h_{R_1} y h_{R_2} representan las alturas de R_1 y R_2 respectivamente y que $h_{R_1} > h_{R_2}$, la Ec. (2.12) permite obtener el valor de NA y la Ec. (2.13) el valor de DA.

$$NA'_{total}(R_1,R_2) = \sum_{j=1}^{h-1}\{NA(R_1,j) + NA(R_2,j')\} \tag{2.12}$$

donde

$$j' = \begin{cases} j-(h_{R_1}-h_{R_2}), & h_{R_1}-h_{R_2}+1 \leq j \leq h_{R_1}-1 \\ 1, & 1 \leq j \leq h_{R_1}-h_{R_2} \end{cases}$$

$$DA'_{total}(R_1,R_2) = \sum_{j=abs|h_{R_1}-h_{R_2}|+1}^{max(h_{R_1},h_{R_2})-1} E_1 + \sum_{j=1}^{abs|h_{R_1}-h_{R_2}|} E_2 \tag{2.13}$$

donde $E_1 = \begin{cases} DA(R_1,j) + DA(R_2,j'), & \text{si } h_{R_1} > h_{R_2} \\ DA(R_1,j') + DA(R_2,j), & \text{si } h_{R_1} < h_{R_2} \end{cases}$ y

$E_2 = \begin{cases} DA(R_1,j), & \text{si } h_{R_1} > h_{R_2} \\ 2 \cdot DA(R_2,j), & \text{si } h_{R_1} < h_{R_2} \end{cases}$

En resumen, en este capítulo se discuten varios aspectos de las bases de datos espaciales destacando los tipos de datos, operadores, tipos de consultas y métodos de acceso espacial. Se han descrito con bastante detalle los métodos espaciales K-D-B-tree y R-tree (junto a algunas de sus variantes) pues estos se utilizan en nuestras propuestas. Para el caso específico del R-tree también

se ha descrito de manera exhaustiva los modelos de costos asociados que permiten predecir el rendimiento de la estructura de datos para procesar las consultas por rango espacial (Window Query) y la reunión espacial. El próximo capítulo continua con la revisón de las bases de datos espacio-temporales y sigue una estructura similar a la dada a este capítulo, es decir, se analizan los tipos de datos, operadores y principales métodos de acceso espacio-temporales propuestos en la literatura.

Capítulo 3

Bases de Datos Espacio-temporales

3.1 Introducción

Las Bases de Datos Espacio-temporales están compuestas de objetos espaciales que cambian su posición y/o forma a lo largo del tiempo [TSPM98]. El objetivo es modelar y representar la naturaleza dinámica de las aplicaciones en el mundo real [NST98]. Algunos ejemplos de estas aplicaciones se pueden encontrar en el ámbito del transporte, medioambiental y en aplicaciones multimedia.

Al igual que un SABDE, un Sistema de Administración de Bases de Datos Espacio-temporales (SABDET) también debería proporcionar, entre otras facilidades, tipos de datos apropiados, un lenguaje de consulta, métodos de acceso y modelos que permitan establecer planes óptimos de ejecución de las consultas.

El objetivo de este capítulo es describir los tipos de datos espacio-temporales, los operadores y predicados sobre estos tipos de datos. Dado que los aportes de esta tesis se concentran en aplicaciones del segundo grupo (ii) descritas en el Capítulo 1 (Sección 1.1), en este capítulo se describen tanto las consultas típicas como los métodos de acceso espacio-temporales que tienen en cuenta las restricciones planteadas para este tipo de aplicaciones.

3.2 Tipos de datos, operadores y predicados espacio-temporales

Existe poca claridad en la literarura sobre los tipos de datos, operadores y predicados espacio-temporales. No está claro cuáles deben ser los tipos de datos, los operadores y su semántica. En los trabajos [ES02, Sch05, GBE+00] se ha abordado el tema y propuesto tipos de datos y predicados espacio-temporales. Los tipos de datos se consiguen extendiendo los tipos de datos espaciales (*punto*, *línea* y *región*) los cuales consideren la dimensión temporal. De esta forma, a partir del tipo de dato *punto* y *región* es posible derivar el tipo *mpunto* o *mregión* [GS05]. Estos tipos de datos se pueden modelar como funciones con dominio temporal y recorrido espacial, es decir,

$$mpunto : tiempo \rightarrow punto$$
$$mregión : tiempo \rightarrow región$$

Un valor de tipo *mpunto* describe una posición como función del tiempo y puede ser representado como una curva en un espacio de tres dimensiones (x,y,t) tal como se muestra en

la Figura 3.1.

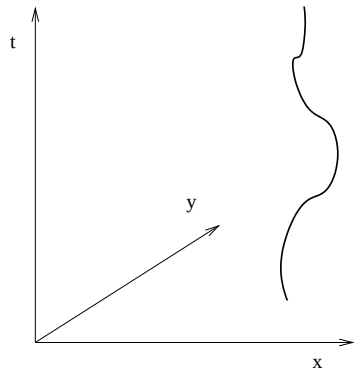


Figura 3.1: Un punto en movimiento.

Un valor de tipo *mregión* es un conjunto de volúmenes en un espacio de tres dimensiones (x,y,t). La intersección de este conjunto de volúmenes con un plano $t = t_0$ produce un objeto de tipo *región* y que describe la componente espacial del objeto espacio-temporal en t_0.

Es posible establecer algunas operaciones sobre estos tipos de datos, por ejemplo,

$$mpunto \times mpunto : \quad \rightarrow mreal \quad \textbf{distancia}$$

el cual calcula la distancia en todos los instantes de tiempo entre dos objetos de tipo *mpunto*. El operador retorna un objeto de tipo *mreal*. Otra operación podría ser

$$mpunto \times mregión : \quad \rightarrow mpunto \quad \textbf{visitas}$$

Este operador obtiene las posiciones de todas las veces que el punto se mantuvo dentro de la región.

Otras operaciones pueden involucrar parámetros espaciales o temporales y otros tipos auxiliares. Por ejemplo,

$mpunto$	$\rightarrow línea$	**trayectoria**
$línea$	$\rightarrow real$	**longitud**
$mreal$	$\rightarrow real$	**mínimo, máximo**
$mreal$	$\rightarrow tiempo$	**tiempo mínimo, tiempo máximo**

Los predicados espacio-temporales se pueden conceptualizar como una extensión de los tipos de datos que permiten modelar los cambios en las relaciones topológicas, métricas o de orientación entre objetos espaciales a lo largo del tiempo. Por ejemplo, dos objetos pueden encontrarse a 10 kilómetros de distancia uno del otro, luego pueden intersectarse para posteriormente separarse. Otro ejemplo de predicado espacio-temporal es aquel que verifica si un objeto cruza (Cross) a otro. Usando las relaciones topológicas definidas en la Sección 2.2 (Figura 2.2) es posible definir el predicado *Cross* de la siguiente manera:

$$\begin{aligned} Cross(A,B) :- \quad & Disjoint(A,B,[t_1,t_2)), Meet(A,B,t_2), Inside(A,B,(t_2,t_3)), \\ & Meet(A,B,t_3), Disjoint(A,B,(t_3,t_4]) \end{aligned}$$

es decir, el objeto A cruza al objeto B si primero ambos permanecen disjuntos, luego se "tocan", posteriormente A permanece dentro de B por un periodo, a continuación se vuelven a "tocar" para finalmente separarse. Notar la importancia del orden en el tiempo en que las relaciones topológicas suceden.

3.3 Consultas espacio-temporales comunes

De manera similar al tipo de consulta WQ (window query) en las bases de datos espaciales, los tipos de consultas espacio-temporales que han recibido mayor atención son *time-slice* y *time-interval* [AAE00, TPZ02, TP01a, MGA03, GS05, TSPM98, PJT00]. Una consulta de tipo *time-slice* recupera todos los objetos que intersectan un rango espacial (window) en un instante de tiempo específico. Una consulta de tipo *time-interval* extiende la misma idea, pero a un intervalo de tiempo. Estas y otras consultas están basadas en las coordenadas de las posiciones que los objetos van alcanzando a lo largo del tiempo.

En esta sección se describen formalmente las consultas anteriores, además de otras que también se consideran importantes en la literatura y sobre las cuales se basan consultas espacio-temporales de mayor complejidad y contenido semántico.

Sea $S = \{o_1, o_2, \ldots, o_m\}$ un conjunto de objetos espaciales. Se asume que los atributos espaciales (G) de los objetos se definen utilizando elementos de un espacio n–dimensional ($\mathbb{R}^n$). En cualquier instante de tiempo t, $o_i(t).G$ define el valor de los atributos espaciales del objeto o_i en el instante t, entonces $S(t) = \{o_1(t).G, o_2(t).G, \ldots, o_m(t).G\}$ define las posiciones y/o forma geométricas de todos los objetos en el instante t. Tal como ya se comentó, los dos tipos de consultas espacio-temporales que más se han estudiado son las siguientes:

i. TS (*time-slice*). Dado un predicado espacial definido por un rectángulo $Q \subseteq \mathbb{R}^n$ y un instante de tiempo, t, reportar todos los objetos que se intersectan espacialmente con Q en el instante de tiempo t, es decir,
$$TS(Q,t) = \{o \mid o(t).G \cap Q \neq \emptyset\}$$

ii. TI (*time-interval*). Dado un rectángulo $Q \subseteq \mathbb{R}^n$ y dos valores de instantes de tiempo t_1 y t_2, seleccionar todos los objetos que se intersectan espacialmente con Q en algún instante de tiempo entre t_1 y t_2, es decir,

$$TI(Q,[t_1,t_2]) = \bigcup_{t_1 \leq t \leq t_2} \{o \mid o(t).G \cap Q \neq \emptyset\}$$

Se han propuesto varios métodos de acceso espacio-temporal para evaluar eficientemente los dos tipos de consultas anteriores (ver Sección 3.4). La evaluación eficiente de estas consultas es de mucho interés, ya que a partir de ellas se pueden componer otras con mayor contenido semántico y eventualmente de mayor costo de evaluación.

Si bien las consultas TS y TI son las que han recibido la mayor atención, existen otras consultas que han ganado interés recientemente. Por ejemplo la reunión espacio-temporal [SHPFT05], el vecino más cercano [AAE00], consultas que permiten verificar un *patrón* espacio-temporal [HKBT05], o aquellas que permiten consultar por los eventos ocurridos en algún instante o periodo de tiempo [Wor05], entre otras. Se definirán a continuación algunas de estas consultas, las que serán abordadas con mayor profundidad en los Capítulos 4, 5, 7 y 8.

i. RST (Reunión espacio-temporal). Dados dos conjuntos de objetos espacio-temporales R y S y un predicado espacial θ, definiremos la reunión espacio-temporal de la siguiente manera:

$$R \bowtie_\theta S = \{\langle (o,o'),[t_i,t_f]\rangle \mid t_i,t_f \in T \wedge t_i \leq t_f \wedge (\forall t \in [t_i,t_f]\ (o,o') \in R(t) \bowtie_\theta S(t))\}$$

con T el conjunto de todos los instantes de tiempo almacenados en la base de datos, $[t_i,t_f]$ es maximal y $R(t) \bowtie_\theta S(t) = \{(o,o') \mid o \in R(t) \wedge o' \in S(t) \wedge \theta(o.G,o'.G)\}$

ii. *K-NN* (Los k-vecinos más cercanos). Dado un objeto o' con $o'.G \subseteq \mathbb{R}^n$, encontrar los k objetos más cercanos a o' en un instante de tiempo dado t.

$$K\text{-}NN(o',t) = \{o \mid \forall o'' : dist(o'(t).G,o(t).G) \leq dist(o'(t).G,o''(t).G)\}$$

iii. Eventos (Consultas sobre eventos). Este tipo de consultas permite verificar si ciertos eventos se han producido en un instante o periodo de tiempo. Los eventos pueden ser, por ejemplo el ingreso de un objeto a una región, la salida de un objeto desde una región o sencillamente la permanencia de un objeto en una región en un determinado instante o periodo de tiempo. A modo de ejemplo se define una consulta orientada a eventos considerando el evento de ingreso de un objeto a una región. Para ello primero se define la ocurrencia del evento (ingreso) de la siguiente manera: $ingreso(o,Q,t_i) = o(t_i).G \cap Q \neq \emptyset \wedge o(t_{i-1}).G \cap Q = \emptyset$. De manera similar se pueden establecer los predicados para los restantes eventos. Los predicados de los eventos se pueden verificar de la siguiente manera:

$$Evento(Q,t,e) = \parallel \{o \mid e(o(t),Q,t)\} \parallel$$

donde $\parallel X \parallel$ representa la cardinalidad el conjunto X.

iv. STP (Consultas sobre patrones espacio-temporales). Una consulta STP (Spatio-Temporal Pattern) se expresa como una secuencia Ω de longitud arbitraria m de predicados espacio-temporales de la forma

$$\Omega = \{(Q_1,T_1),(Q_2,T_2),\ldots,(Q_m,T_m)\},$$

donde en cada par (Q,T), Q representa un predicado espacial y T una restricción temporal. En esta definición Q puede representar ya sea una rango espacial (window query) o una consulta por el vecino más cercano (*K-NN*). Sin embargo, el modelo puede ser fácilmente ampliado a otros predicados espaciales [HKBT05]. T puede ser un instante de tiempo (t), un intervalo de tiempo (Δt) o vacío ($\emptyset$). La evaluación de esta consulta permite recuperar todos los objetos que satisfacen todos los predicados espacio-temporales especificados en Ω, es decir,

$$STP(\{(Q_1,T_1),(Q_2,T_2),\ldots,(Q_m,T_m)\}) = \{o \mid o\ satisface(Q_i,T_i)\ \forall\ 1 \leq i \leq m\}$$

3.4 Métodos de acceso espacio-temporales

Un SABDET debe proveer, entre otros servicios, métodos de acceso que permitan procesar en forma eficiente consultas cuyos predicados contemplan relaciones espacio-temporales sobre

los objetos. En esta sección se describen los métodos de acceso espacio-temporales conocidos hasta ahora. Consideraremos solamente las técnicas que están orientadas, principalmente, a procesar consultas de tipo *time-slice* y *time-interval* sobre información histórica de aplicaciones (ii) descritas en el Capítulo 1 (Sección 1.1), dado que los aportes de esta tesis se encuentran principalmente orientados a ese nicho. Existen varios métodos de acceso espacio-temporales que son adecuados para aplicaciones que consideran cambios espaciales de manera discreta. Una clasificación de ellos es la siguiente:

i. Métodos que tratan el tiempo como otra dimensión.

ii. Métodos que incorporan la información temporal dentro de los nodos de la estructura sin considerar el tiempo como otra dimensión.

iii. Métodos que usan sobreposición de la estructura de datos subyacente.

iv. Métodos basados en versiones de la estructura.

v. Métodos orientados a la trayectoria.

La Figura 3.2 muestra un ejemplo de la evolución de un conjunto de objetos espacio-temporales en distintos instantes de tiempo[1]. En tal figura se puede observar que la geometría del objeto A se aproxima con un MBR. En adelante sólo se considerarán los MBRs de los objetos y el identificador usado ($A, B, \ldots, I, A', B'$) se referirá indistintamente al objeto propiamente tal o a su MBR. Los MBRs identificados con $R1, R2, \ldots, R13$ y con línea punteada se utilizan para agrupar MBRs y no forman parte del conjunto de objetos del problema; se utilizarán para explicar el método de acceso espacio-temporal HR-tree. El rectángulo Q modela la componente espacial de una consulta dada en $t = 1$.

En el instante $t = 0$ de la Figura 3.2, permanecen en el espacio los objetos identificados con $A, B, \ldots, H$. En el instante $t = 1$ el objeto A se desplaza a la vez que cambian los límites de su geometría transformándose en el objeto A'; algo similar ocurre con el objeto B. Finalmente, en el instante $t = 2$, G desaparece y aparece el objeto I. En el instante $t = 1$ se realiza una consulta espacio-temporal cuya restricción espacial se encuentra especificada por el rectángulo Q. La respuesta de esta consulta incluye los MBRs D, E y F.

3.4.1 3D R-tree

El 3D R-tree, propuesto en [TVS96, PLM01] es un método espacio-temporal basado en el primer enfoque, ya que considera el tiempo como una dimensión adicional a las coordenadas espaciales. Usando este razonamiento un objeto espacial, modelado como un punto, el cual permanece inicialmente en (x_i, y_i) durante el tiempo $[t_i, t_j)$ y luego en (x_j, y_j) durante el tiempo $[t_j, t_k)$, puede ser descrito por dos segmentos de línea en el espacio tridimensional identificados por $\overline{[(x_i, y_i, t_i), (x_i, y_i, t_j))}$ y $\overline{[(x_j, y_j, t_j), (x_j, y_j, t_k))}$, y luego ser indexados por un R-tree [Gut84]. En el caso de representar los objetos espaciales con rectángulos, como los de la Figura 3.2, las diferentes versiones de un objeto se modelan como cubos, como se puede apreciar en la Figura 3.3, donde los objetos A y B tienen distintas versiones A' y B' respectivamente y se encuentran modeladas por cuatro cubos diferentes. Los distintos cubos ($A, B, \ldots, H, I, A'$ y B') se almacenan en un R-tree de tres dimensiones (3D R-treee), tal como se muestra en la Figura 3.4.

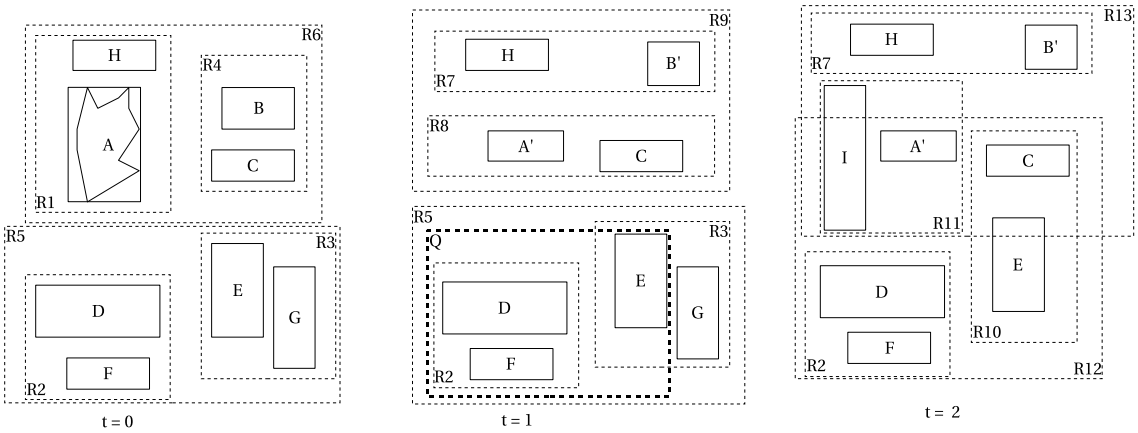


Figura 3.2: Ejemplo de la evolución de un conjunto de objetos.

Una consulta espacio-temporal es modelada como un cubo, donde la proyección *x–y* representa el subespacio (rectángulo) de la consulta y la proyección del tiempo corresponde al intervalo temporal. Usando el cubo especificado en la consulta, se busca en el 3D R-tree todos los objetos que se intersectan con él usando el mismo procedimiento de búsqueda especificado en la propuesta original de R-tree [Gut84]. De esta manera, para procesar la consulta especificada en el ejemplo de la Figura 3.2, se comienza desde la raíz del árbol de la Figura 3.4 verificando si el ractángulo Q se intersecta con los cubos $S1, S2, S3$ o $S4$. En este caso, sólo los cubos $S2$ y $S4$ cumplen la condición. Seguidamente se verifica si los cubos contenidos por $S2$ y $S4$, es decir, A', D, F, E y G se intersectan con Q resultando como respuesta de la consulta los objetos D, E y F.

Las principales ventajas de 3D R-tree son su buena utilización del almacenamiento y su eficiencia en el procesamiento de consultas de tipo *time-interval* dado que simplemente necesita recuperar del 3D R-tree sólo los objetos (cubos) que se intersectan con el cubo de la consulta. Sin embargo, una de sus desventajas más importantes es su ineficiencia para procesar consultas del tipo *time-slice* ya que la altura del 3D R-tree puede llegar a ser muy grande producto de la gran cantidad de objetos que es necesario almacenar. Otra desventaja es la necesidad de conocer por anticipado los intervalos de tiempo, lo que impide mezclar operaciones que actualizan la estructura de datos con operaciones de consultas.

Una manera de resolver esta última limitación del 3D R-tree es considerando dos R-trees (2+3 R-tree) [NST99, NST98]. El primero, un R-tree de dos dimensiones, se utiliza para mantener aquellos objetos cuyo límite final del intervalo de tiempo aún es desconocido. El segundo corresponde a un 3D R-tree en el cual se almacenan los objetos cuyos intervalos espaciales y temporales son conocidos. De esta forma cuando un objeto se desplaza a una nueva posición o cambia su forma geométrica, se busca dicho objeto en el R-tree de dos dimensiones, se completa su intervalo temporal y se inserta un objeto de tres dimensiones (un cubo) en el 3D R-tree. Seguidamente, se actualiza el R-tree de dos dimensiones con los nuevos valores de los atributos espaciales del objeto y con el instante de tiempo en que alcanzó dicha posición o forma. Una de las principales desventajas de esta variante es que las operaciones de búsqueda requieren revisar dos

[1]Este ejemplo será utilizado posteriomente para explicar algunos de los métodos de acceso espacio-temporales

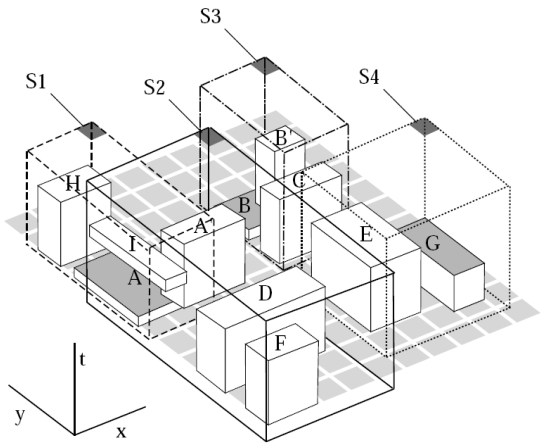


Figura 3.3: Objetos espacio-temporales de la Figura 3.2 modelados en tres dimensiones (x,y,t). Los cubos con su cara superior de color gris indican cubos completamente delimitados (A, B y G); los restantes cubos no se encuentran delimitados totalmente, ya que aún no se conoce el tiempo final de permanencia en su última posición.

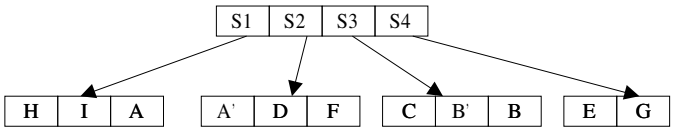


Figura 3.4: 3D R-tree de los objetos de la Figura 3.3.

árboles, dependiendo del intervalo de tiempo dado en la consulta. Sin embargo, puede ser muy útil en aplicaciones en las cuales existen pocos objetos pero con una frecuencia muy alta de cambios, ya que gran parte del R-tree de dos dimensiones podría mantenerse en memoria.

3.4.2 RT-tree

El RT-tree [XHL90] es un método que se basa en el segundo enfoque. En este método la estructura de datos original de los nodos del R-tree [Gut84] se ha modificado con el propósito de almacenar en los nodos la información espacial junto a la información temporal. Esto se puede apreciar en el RT-tree de la Figura 3.5, el que agrupa los objetos según muestran los rectángulos $S1, S2, S3, S4, S5$ y $S6$ de líneas punteadas de la Figura 3.6.

Un RT-tree de orden *m* es un árbol balanceado por la altura en que los nodos hojas contienen entradas de la forma $\langle MBR, T, ref \rangle$, donde ref es un puntero al objeto, y T representa el intervalo de tiempo $[t_i, t_j]$ que indica el periodo de tiempo en el cual el objeto ha permanecido en una determinada posición.

Por ejemplo, el objeto H (ver Figura 3.2) ha permanecido en la misma posición en el intervalo de tiempo $[0,2]$, lo que se indica en el RT-tree de la Figura 3.5. Dado que para el objeto A su forma

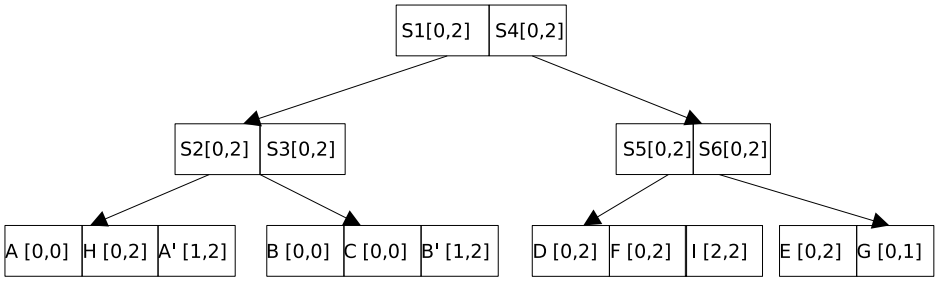


Figura 3.5: Objetos de la Figura 3.6 almacenados en un RT-tree.

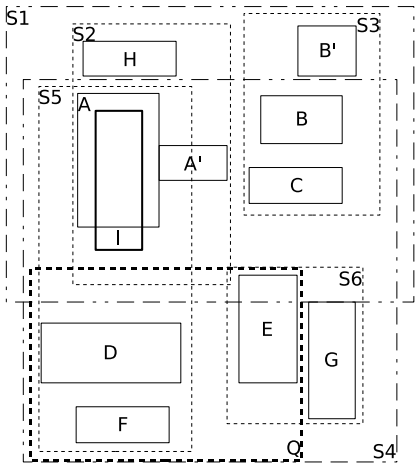


Figura 3.6: Agrupación de los objetos de la Figura 3.2 para formar un RT-tree.

geométrica y su posición han cambiado, es necesario representarlo por un MBR distinto en el instante $t = 1$, A' en este caso, el cual se inserta en el RT-tree. En el momento de la inserción de A' su intervalo es $[1,1]$ el que se actualiza en el instante $t = 2$ a $[1,2]$.

Cada entrada en un nodo interno tiene la misma estructura que un nodo hoja, es decir, $\langle MBR, T, ref \rangle$. Pero en este tipo de entradas, ref apunta a un subárbol cuyas hojas tienen entradas de la forma $\langle MBR_i, T_i, ref_i \rangle$, tal que cada MBR_i es cubierto por MBR.

En este método de acceso, una consulta espacio-temporal se procesa recuperando todos los objetos cuya geometría (atributos espaciales) se intersecta espacialmente con el rectángulo dado en la consulta y su intervalo de tiempo está contenido en el especificado en la consulta. Por ejemplo, la consulta de la Figura 3.6 se procesaría partiendo desde la raíz del árbol de la Figura 3.5 y seleccionando los rectángulos que se intersectan con Q, en este caso, $S1$ y $S4$. De la misma manera, a continuación se seleccionan los rectángulos contenidos por $S1$ y $S4$ que se intersectan con Q, resultando ser $S2, S5$ y $S6$. Finalmente, el proceso anterior se repite para $S1, S5$ y $S6$ y se obtiene como respuesta de la consulta los objetos D, E y F.

Las desventajas del RT-tree son su alto costo para procesar consultas del tipo *time-slice*, el cual proviene de la gran altura que puede llegar a alcanzar el R-tree y de su ineficiencia

para procesar consultas con predicados temporales, ya que la dimensión temporal juega un rol secundario. No obstante lo anterior, el RT-tree es uno de los métodos más eficientes en la utilización del almacenamiento.

3.4.3 HR-tree

Los métodos HR-tree [NST99, NST98], MR-tree [XHL90] y OLQ (Overlapping Linear Quadtree) se basan en el concepto de sobreposición y tratan de impedir el exceso de almacenamiento producido al contar con un R-tree por cada instante de tiempo. En la Figura 3.7 se muestra un HR-tree para los objetos de la Figura 3.2 y considerando los instantes de tiempo $t = 0$, $t = 1$ y $t = 2$. El posible ahorro se logra evitando almacenar objetos comunes entre R-trees para instantes de tiempo consecutivos. Esto se puede apreciar en la Figura 3.7, donde el subárbol $n1$ se reutiliza en el instante $t = 1$ y los subárboles $n2$ y $n3$ se reutilizan en en el insante $t = 2$.

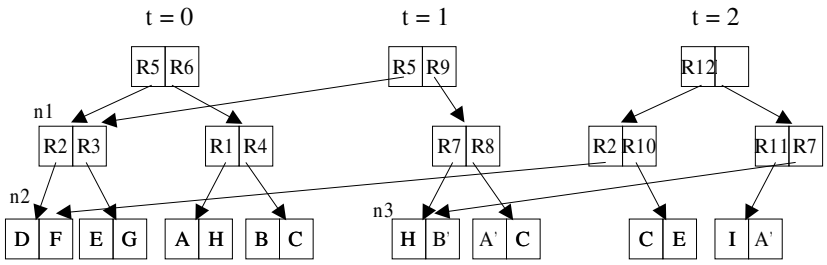


Figura 3.7: Objetos de la Figura 3.2 almacenados en un HR-tree.

Para procesar una consulta de tipo *time-slice* por ejemplo, primero se ubica el R-tree adecuado de acuerdo al tiempo especificado en la consulta. Posteriormente, y usando el árbol seleccionado, se recuperan todos los objetos cuyos atributos espaciales se intersectan con el rectángulo dado en la consulta. Una consulta del tipo *time-interval* se procesa de manera muy similar, sólo que es necesario recorrer varios R-trees. Para procesar la consulta especificada en la Figura 3.2, en primer lugar se selecciona el R-tree del instante $t = 1$. Luego, usando el rectángulo Q, se recupera el rectángulo $R5$ ya que se intersecta con Q. A continuación se verifican si los rectángulos contenidos por $R5$, es decir, $R2$ y $R3$ se intersectan con Q, resultando verdadero en ambos casos. Finalmente se exploran $R2$ y $R5$ verificando si los MBRs contenidos en ellos se intersectan con Q, recuperando así los objetos D, E y F los que satisfacen la consulta espacio-temporal.

El HR-tree es uno de los métodos más estudiado, para el cual existen implementaciones, se han realizado evaluaciones y ha servido de base para otros métodos. La gran ventaja del HR-tree es su eficiencia para procesar consultas del tipo *time-slice*, de hecho es uno de los métodos más eficientes. Sin embargo, el HR-tree es muy ineficiente para procesar consultas de tipo *time-interval*, ya que necesita recorrer cada uno de los árboles creados en cada instante de tiempo comprendido en el intervalo de tiempo dado en la consulta. Su otra gran desventaja es la excesiva cantidad de almacenamiento que requiere para el índice, lo que lo hace poco práctico. Según [TP01a], eventualmente bastaría con que sólo un 1% de los objetos sufriera un cambio en su geometría entre instantes de tiempo consecutivos, para que el HR-tree degenere en un R-tree

(físico) por cada instante de tiempo almacenado en la base de datos (fenómeno de la redundancia de versión).

Con el propósito de eliminar las desventajas del HR-tree, en [TP01a] se propone una variante denominada HR$^+$-tree. Distinto a un HR-tree, este método permite almacenar entradas pertenecientes a instantes de tiempo diferentes en un mismo nodo. Este arreglo elimina en parte el fenómeno de la redundancia de versión producida en el HR-tree, permitiendo un ahorro significativo de almacenamiento. Por otra parte, el HR$^+$-tree también mejora la eficiencia de las consultas de tipo *time-interval* evitando recorrer subárboles que ya han sido recorridos en instantes de tiempo previos. Por ejemplo, si la consulta espacio-temporal realizada sobre el HR-tree de la Figura 3.7 tiene como intervalo $[0,2]$ y Q obliga a recorrer el subárbol $n1$, entonces éste es marcado en $t=0$ como recorrido, y en $t=1$ no se vuelve a recorrer.

3.4.4 MVR-tree y MV3R-tree

El MVR-tree [TP01b, TP01a, TPZ02] es un método de acceso multiversión y corresponde a una extensión de MVB-tree [BGO$^+$96], donde el atributo que cambia a través del tiempo es la componente espacial. Dado que MVB-tree y MVR-tree siguen las mismas ideas básicas, y teniendo en cuenta que la explicación es más simple realizándola sobre un MVB-tree, en esta sección se describe con bastante detalle el MVB-tree. También se utilizan ejemplos con datos diferentes a los que se han venido considerando con el objeto de presentar con mayor claridad el comportamiento del MVB-tree y, por lo tanto, del MVR-tree. También se justifica describir con mayor detalle el MVB-tree (MVR-tree), ya que el MVR-tree se utiliza para comparar los resultados de los métodos de acceso espacio-temporales propuestos en esta tesis.

En general, en una estructura multiversión (MVB-tree o MVR-tree) cada entrada tiene la forma $\langle S, t_i, t_e, ref \rangle$, donde t_i indica el instante en que el registro fue insertado y t_e el instante de tiempo en que el objeto se eliminó. Para los nodos hojas, S se refiere al valor de la variable de un objeto que cambia en el tiempo (por ejemplo, los salarios de un empleado o posiciones de un auto para el caso del MVR-tree). Para los nodos intermedios, la entrada S determina el mínimo rango que incluye a las entradas vigentes ("vivas") en el intervalo de tiempo $[t_i, t_e)$. Por ejemplo, en el caso del MVR-tree será el MBR que incluye espacialmente a todas las entradas vigente entre $[t_i, t_e)$ en el subárbol apuntado por ref. En el caso del MVB-tree S indica el valor mínimo de la variable cambiante en el tiempo en el subárbol. El atributo ref es un puntero, que dependiendo de si el nodo es una hoja o un nodo interno, apunta a un registro o tupla con la información del objeto, o a un nodo del siguiente nivel del árbol, respectivamente. Cuando se inserta una nueva entrada en el instante de tiempo t, el atributo t_i se fija al valor t y el atributo t_e a "*" (para indicar el valor del tiempo actual - NOW). Al revés, cuando una entrada se elimina lógicamente en un instante t, el valor de t_e se actualiza con t. De esta forma las entradas vigentes tienen "*" como valor en t_e y en otro caso se consideran como entradas no vigentes ("muertas"). La Figura 3.8 muestra una instancia de un MVB-tree.

En un MVB-tree (MVR-tree), para cada instante de tiempo t y para cada nodo, excepto la raíz (R en la Figura 3.8), se asegura que existan, o bien al menos $b \cdot P_U$, o bien ninguna entrada "viva" en el instante t, donde b es la capacidad del nodo (cantidad de entradas) y P_U es un parámetro del árbol (en los ejemplos que se discuten a continuación $P_U = \frac{1}{3}$ y $b=6$). Esta condición se conoce como *condicón de versión débil* (weak version condition) y asegura que las entradas "vivas" en el mismo instante de tiempo queden juntas con el propósito de facilitar consultas de tipo *time-slice*.

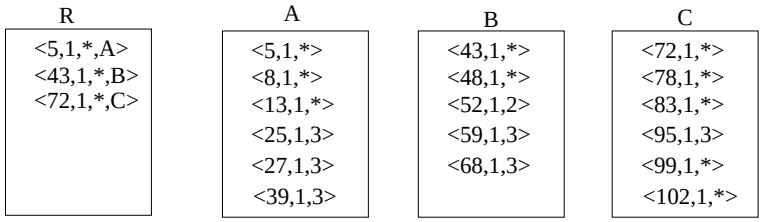


Figura 3.8: Instancia de un MVB-tree.

Esta condición se puede violar al momento de hacer una eliminación lógica de una entrada, y entonces debe reestablecerse.

Las inserciones y eliminaciones se llevan a cabo de manera diferente a como se hace en un B-tree o R-tree. Estas diferencias se producen en los casos en que es necesario dividir (inserción) o mezclar (eliminación) nodos. Por ejemplo, cuando se inserta una nueva entrada y el nodo donde corresponde almacenar esta nueva entrada se encuentra lleno (overflow), lo que se hace en primer lugar es una partición por versión (version split), la que consiste en que todas las entradas "vivas" del nodo en cuestión se copian a un nodo nuevo con sus atributos t_i inicializado al tiempo actual (el instante de tiempo en que se produjo el overflow). El valor de los atributos t_e (en el nodo en que se produjo el overflow) se actualizan al valor del tiempo actual. Esto lo podemos ver en la Figura 3.9 cuando se inserta la entrada $\langle 28, 4, * \rangle$ en el nodo *A* en el instante de tiempo 4 en el MVB-tree de la Figura 3.8. Es posible observar que al tratar de hacer la inserción en el nodo *A*, este se encuentra lleno y, por lo tanto, se crea un nuevo nodo *D* al cual se transfieren todas las entradas que aún están "vivas" en el instante 4 en *A*. La nueva entrada se inserta en el nodo *D*, se agrega una nueva entrada apuntando a *D* en el nodo raíz *R* y la entrada en *R* que apunta a *A* se actualiza poniendo 4 como valor de t_e. Es posible que continuas inserciones produzcan que un nodo raíz también se tenga que dividir por medio de una partición de versión en cuyo caso el nuevo nodo de la división corresponde a la raíz de otro árbol lógico. Es posible observar que las particiones por versiones provocan duplicación de entradas. Por ejemplo, en la Figura 3.9 los objetos con claves 5, 8 y 13 se encuentran duplicados en el nodo *D*.

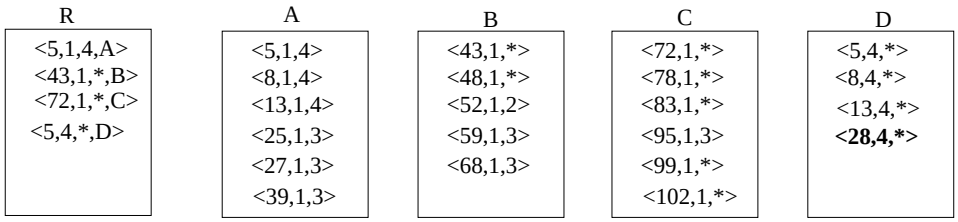


Figura 3.9: Ejemplo de una partición por versión de un MVB-tree.

En algunas situaciones puede ocurrir que al realizar una partición por versión, el nuevo nodo quede casi lleno de tal manera que basta con unas pocas inserciones para que sea necesario hacer una nueva partición por versión. También puede ocurrir que el nuevo nodo quede con pocas entradas "vivas" de tal manera que con unas pocas eliminaciones se puede violar la condición de versión débil. Con el propósito de evitar estas situaciones, es necesario que el número de

entradas en el nuevo nodo se encuentre en el rango de $[b \cdot P_{SVU}, b \cdot P_{SVO}]$ después de una partición por versión. Cuando el número de entradas supera a $b \cdot P_{SVO}$ entonces, el nuevo nodo se debe dividir nuevamente, pero ahora basado en los valores de la clave (key split), es decir, se divide un nodo exactamente igual a como se hace en un B-tree o R-tree cuando se produce la misma situación. Por ejemplo, si se considera $P_{SVU} = \frac{1}{3}$ y $P_{SVO} = \frac{5}{6}$ e insertamos la entrada $\langle 105, 5, * \rangle$ en el nodo *C* (Figura 3.9). En este caso podemos observar que el nodo *C* se debe particionar por versión seguido de una partición por clave (key split) dando origen a los nodos *E* y *F* de la Figura 3.10.

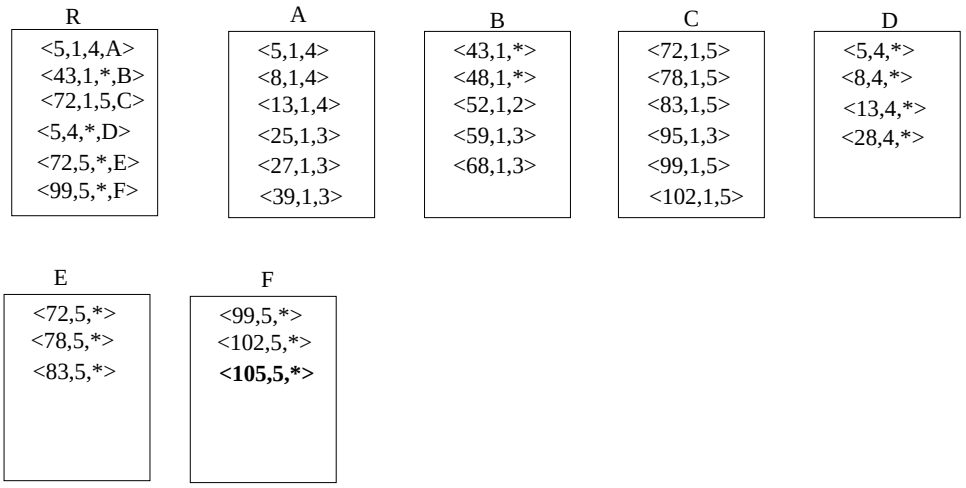


Figura 3.10: Ejemplo de una partición por versión seguida de una partición por clave en un MVB-tree.

Existen dos situaciones en las cuales un nodo puede quedar con pocas entradas "vivas". La primera es debido a la eliminación lógica de entradas (se puede violar la condición de versión débil), y la otra es la que se puede producir al realizar una partición por versión y donde la cantidad de entradas en el nuevo nodo esté por debajo de $b \cdot P_{SVU}$. En ambos casos la solución es la misma y consiste en mezclar las entradas del nodo con las entradas "vivas" de un nodo hermano. Por ejemplo, si se elimina la entrada $\langle 48, 1, * \rangle$ del nodo *B* de la Figura 3.8, dicho nodo queda con pocas entradas "vivas". Lo que se hace luego es elegir un nodo hermano, en este caso *C*, y todas sus entradas "vivas" se copian a un nuevo nodo junto con la entrada $\langle 43, 4, * \rangle$ siendo necesario hacer una partición por clave del nuevo nodo generando los nodos *D* y *E* (Figura 3.11).

Cada raíz abarca un intervalo temporal conocido como intervalo de jurisdicción y que corresponde al intervalo temporal mínimo que incluye a todos los intervalos temporales de las entradas en la raíz. Los intervalos de jurisdicción son mutuamente disjuntos. El procesamiento de un *time-slice* o *time-interval* comienza recuperando aquellas raíces cuyos intervalos de jurisdicción se intersectan con el intervalo de la consulta. A continuación la búsqueda continúa guiada por *S*, t_i y t_e hasta alcanzar las hojas.

La Figura 3.12 ilustra un ejemplo de un MVR-tree (no se utilizan los mismos datos de los ejemplos que se han usado para los restante métodos de acceso) visualizado en tres dimensiones. En el ejemplo (tomado de [TP01b]) el valor de $b = 3$ y $P_U = \frac{1}{3}$. Los objetos *A* a *G* (líneas delgadas) fueron insertados en orden alfabético en el tiempo. Podemos observar que en el instante de tiempo

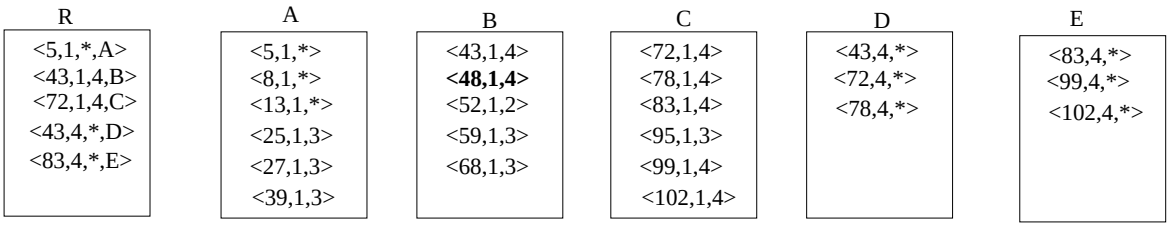


Figura 3.11: Ejemplo de una versión débil en un MVB-tree.

t al insertar el objecto C, el nodo H se llena provocando la creación del nodo I el cual almacena una copia del objeto C (C era el único objeto de H vigente en t). Junto con lo anterior, en la entrada correspondiente a H, el valor de t_e se cambia a t cerrando este nodo. En la Figura 3.12 los nodos I, C y K están vigentes en el instante de tiempo actual (NOW) y A, B, D, E, G y F se encuentran no vigentes.

Similar al MVB-tree, un MVR-tree tiene múltiples R-trees (árboles lógicos) que organizan la información espacial en intervalos de tiempo que no se sobreponen. El MVR-tree necesita de menos almacenamiento y de menos tiempo de procesamiento para consultas espacio-temporales que el HR-tree para intervalos cortos. Existe una variante del MVR-tree conocida como MV3R-tree [TP01b] la cual mejora el rendimiento del MVR-tree para consultas cuyo intervalo temporal es largo (sobre un 5% del largo del intervalo total de tiempo almacenado en la base de datos) por medio de la mezcla de dos estructuras: un MVR-tree para procesar consultas con longitud de intervalo corto y un 3D R-tree (construído sobre las hojas del MVR-tree) para procesar consultas con intervalos temporales largos.

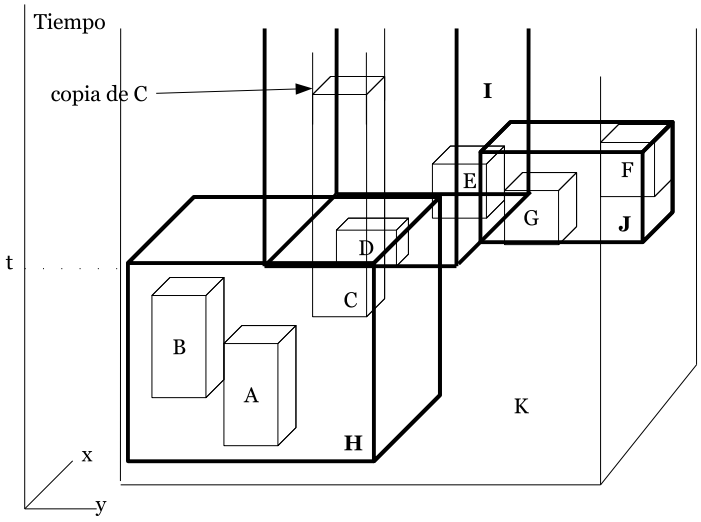


Figura 3.12: Ejemplo de un MVR-tree.

3.4.5 STR-tree, TB-tree y SETI

Estos métodos de acceso se basan en la trayectoria que siguen los objetos. Tales métodos de acceso tienden a privilegiar las consultas sobre las trayectorias de los objetos más que las consultas de tipo *time-slice* y *time-interval*, es decir, el concepto de cercanía espacial de los objetos utilizado para mantener el índice espacial es de segunda importancia.

En [PJT00] se proponen dos métodos de acceso espacio-temporal basados en las trayectorias seguidas por los objetos espaciales de tipo punto. El primero de ellos, conocido como STR-tree, corresponde a una extensión del R-tree donde se ha modificado de manera considerable el algoritmo de inserción. Los nodos hojas tienen la siguiente estructura $\langle oid, ty_i, MBR, o \rangle$ donde ty_i es un identificador de la trayectoria y o es la orientación de la trayectoria dentro del MBR. La idea fundamental del STR-tree es tratar de almacenar juntos los segmentos pertenecientes a la trayectoria de un mismo objeto manteniendo en lo posible la cercanía espacial de tal manera de no deteriorar el rendimiento del R-tree. El algoritmo de inserción no selecciona el MBR donde insertar un objeto de acuerdo a un criterio de minimizar el crecimiento del área (como se hace en el R-tree), si no que en la elección se privilegia que los segmentos que corresponden a la trayectoria de un mismo objeto se almacenen lo más cerca posible. De esta forma cuando se inserta un nuevo segmento de línea el objetivo es hacerlo lo más cerca posible del segmento predecesor en la trayectoria del objeto. El algoritmo primero busca el nodo hoja que contiene el segmento predecesor en la trayectoria del objeto. Si el nodo contiene suficiente espacio, entonces el segmento se inserta, se ajustan los MBRs de los nodos ancestros y el algoritmo de inserción termina. Cuando el nodo hoja seleccionado está lleno, es necesario hacer una partición del mismo. El algoritmo de partición debe considerar varios escenarios entre los segmentos del nodo hoja al momento de particionarlo. Por ejemplo, que los dos segmentos pertenezcan a la misma trayectoria o no, y si pertenecen a la misma trayectoria, que éstos tengan puntos en común o no.

El segundo método de acceso propuesto en [PJT00], conocido como TB-tree, es radicalmente diferente al STR-tree. El STR-tree intenta que los segmentos de una misma trayectoria queden almacenados juntos, en cambio el TB-tree considera una estructura de datos que permite acceder de manera directa a la trayectoria completa de un objeto. Un nodo hoja de un TB-tree contiene segmentos que pertenecen a la trayectoria de un mismo objeto. Una de las desventajas del TB-tree es que los segmentos de trayectorias de distintos objetos quedan almacenadas en bloques diferentes a pesar de estar espacialmente cercanas, lo que tiene como consecuencia una baja capacidad de discriminación espacial y esto a su vez tiene un efecto negativo sobre el rendimiento de las consultas de tipo *time-slice* y *time-interval*. En [PJT00] se propone relajar esta restricción particionando la trayectoria total en tramos y manteniendo la trayectoria total por medio de una lista doblemente encadenada de los tramos.

En [CEP03] se propone otro método enfocado a la trayectoria, conocido como SETI. El método particiona el espacio en un conjunto de zonas o regiones disjuntas y estáticas. Este método supone que la dimensión espacial es estática y que existen una alta evolución de los objetos en el tiempo, es decir, los objetos se mueven muy frecuentemente sobre un espacio fijo. En cada partición existe un índice escaso o ralo que indexa segmentos temporales que representan la permanencia de los objetos en las diferentes posiciones a lo largo de sus trayectorias. Si el segmento de trayectoria intersecta dos o más particiones, éste se divide en varios segmentos los que se insertan en el índice. La gran limitante de SETI es que las particiones espaciales son fijas. Si las áreas de las particiones son muy grandes, la componente espacial discriminará muy poco lo

que afectará el rendimiento de las consultas. Por otro lado, si las áreas son muy pequeñas habrá muchos segmentos que se intersectarán con varias particiones, lo que aumenta el almacenamiento y también afectará el rendimiento de las consultas. Una partición estática se puede volver ineficiente en la medida que los segmentos, a lo largo del tiempo, se concentren en zonas específicas del espacio saturando los índices temporales asignados a esas zonas.

En la Figura 3.13 es posible ver la evolución que han tenido los métodos de acceso espacio-temporales y también se muestran los métodos de acceso espaciales y temporales sobres los cuales se han basado, y donde es posible apreciar la influencia del R-tree en la mayoría de los métodos de acceso espacio-temporales.

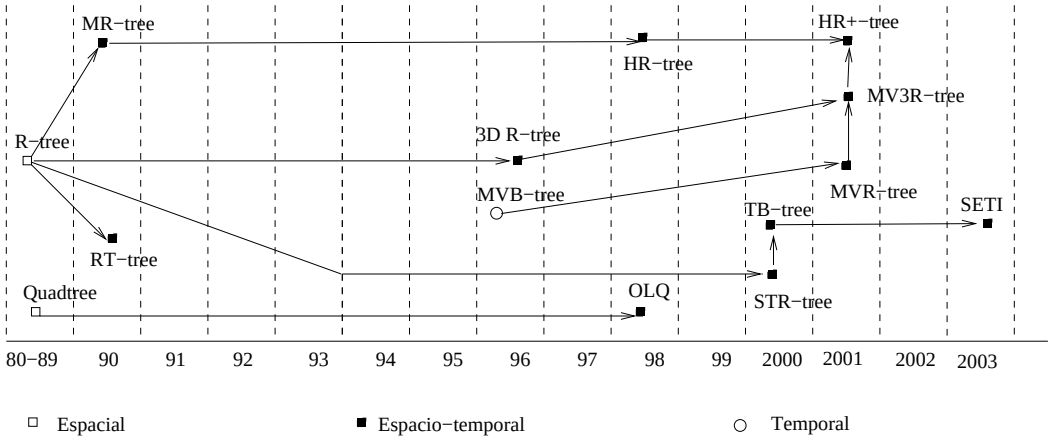


Figura 3.13: Evolución de los métodos de acceso espacio-temporales utilizados para indexar la historia de los objetos espaciales.

Este capítulo se concentra en describir las bases de datos espacio-temporales. Se discuten los tipos de datos y operadores espacio-temporales. Se hace una descripción muy detallada de los tipos de consultas más comunes y de los principales métodos de acceso abordados en la literatura para procesarlas. El próximo capítulo, perteneciente a la segunda parte de esta tesis, discute nuestro primer método de acceso espacio-temporal el que sigue el enfoque basado en snapshots y eventos.

Parte II

Métodos de acceso espacio-temporales basados en snapshots y eventos

Capítulo 4

SEST-Index

4.1 Introducción

En este capítulo y en el siguiente se describen los métodos de acceso espacio-temporales que proponemos como parte de esta tesis. La idea general detrás de ambos métodos consiste en mantener snapshots[1] de la base de datos para ciertos instantes de tiempo y una bitácora o *log* para almacenar los eventos ocurridos entre snapshots consecutivos. Este enfoque es consistente con el modelo propuesto en [Wor05] para la representación de fenómenos geográficos y que consiste de (i) *snapshot temporales* y (ii) *eventos sobre objetos*. Nuestro objetivo es contar con métodos de acceso espacio-temporal que no solamente puedan procesar consultas de tipo *time-slice* y *time-interval*, sino que también consultas sobre eventos. A pesar de que varios métodos de acceso espacio-temporal también manejan los cambios espaciales de los objetos, ellos los utilizan sólo con el propósito de actualizar la estructura de datos subyacente. En el contexto de este trabajo un cambio corresponde a una modificación producida en la geometría o componente espacial de un objeto en un instante determinado. Por ejemplo, el desplazamiento espacial de un objeto se modela mediante un cambio. Un cambio se considera formado por o equivalente a dos eventos, a saber, *move_out* (un objeto deja de permanecer en su última posición) y *move_in* (un objeto se mueve a una nueva posición). En este trabajo en oportunidades consideraremos cambios y eventos como sinónimos.

Los métodos de acceso espacio-temporal actuales, no mantienen los datos sobre los cambios como registros de eventos ocurridos sobre los objetos y, por lo tanto, no es posible responder eficientemente consultas sobre los eventos ocurridos en un instante o intervalo de tiempo.

Nuestro enfoque ha sido discutido en otros trabajos [Kol00, KTG+01], donde fue descartado a priori argumentando que no es fácil determinar cada cuántos eventos realizar un nuevo snapshot y cuál es el tiempo extra para procesar las consultas. Nuestros métodos de acceso muestran que esto no es una desventaja muy importante. El número de snapshots representa un trade-off entre almacenamiento y tiempo, ya que en la medida que el número de snapshots aumenta, disminuye el tiempo de una consulta mientras que la cantidad de almacenamiento aumenta. A la inversa, con un número pequeño de snapshots, el almacenamiento disminuye mientras que el tiempo de respuesta de las consultas aumenta y, por lo tanto, la frecuencia de los snapshots puede ser ajustada

[1] un snapshot corresponde a un estado de la base de datos en un instante determinado, es decir, las posiciones espaciales de todos los objetos en dicho instante

dependiendo del tipo de aplicaciones y del porcentaje de movilidad de los objetos. Por ejemplo, existen muchas aplicaciones donde no es de interés consultar sobre el estado de los objetos en algún periodo de tiempo. Nuestras estructuras de datos para snapshots y eventos son independientes, y en consecuencia nuestros métodos de acceso se pueden beneficiar de las mejoras que tenga cada estructura. Además, en nuestro enfoque es muy fácil integrar otros métodos de acceso espaciales para manejar los snapshots.

La idea de snapshot y bitácora también se ha utilizado con el propósito de mantener vistas concretas (materialized views) [GM95]. En este contexto, se crea una bitácora sobre una tabla denominada tabla maestra, a partir de la cual se obtiene la vista inicial. Luego, la vista se actualiza (se refresca) con los eventos almacenados en la bitácora, los que se eliminan una vez que la vista se pone al día. En nuestros métodos de acceso, por el contrario, las bitácoras son parte de una estructura de datos que permanece en el tiempo y, a partir de la cual se pueden obtener diferentes estados de la base de datos.

Este capítulo se concentra en definir nuestro primer método (SEST-Index), el cual considera la creación de los snapshots teniendo en cuenta la posición de todos los objetos en el instante de tiempo en que se decide crear los snapshots; y las bitácoras mantienen la información de los eventos que se van produciendo en el espacio completo (ver Figura 4.1). Las entradas en las bitácoras se mantienen ordenadas por el tiempo y permiten reconstruir el estado de la base de datos entre dos snapshots consecutivos. El SEST-Index utiliza un R-tree para indexar los objetos considerando sus posiciones espaciales en el instante de tiempo en que se decide crear un nuevo snapshot. Por ejemplo, en la Figura 4.1, los objetos considerados en el snapshot creado en el instante de tiempo t_0 se almacenan en el R-tree R_0, y los eventos de los objetos producidos en el intervalo temporal (t_0, t_1) se almacenan en la bitácora L_0. De esta forma, para recuperar el estado de la base de datos en un instante t con $t_0 < t < t_1$, se empieza desde el R-tree en el instante t_0 y se actualiza con la información contenida en la bitácora L_0.

La organización de este capítulo es como sigue. En la Sección 4.2 se describen las estructuras de datos y algoritmos que evalúan las consultas espacio-temporales y operaciones fundamentales del SEST-Index. En la Sección 4.3 se presentan y discuten los resultados de una evaluación experimental en la que se compara el SEST-Index con el HR-tree. En la Sección 4.4 se presenta una variante del SEST-Index, la que también se compara experimentalmente con el HR-tree al cual supera en espacio y tiempo de procesamiento de las consultas. En la Sección 4.5 se especifica un modelo de costo para estimar el espacio y la eficiencia de las consultas espacio-temporales procesadas con el SEST-Index. Finalmente, en la Sección 4.6 se presentan las conclusiones de este capítulo.

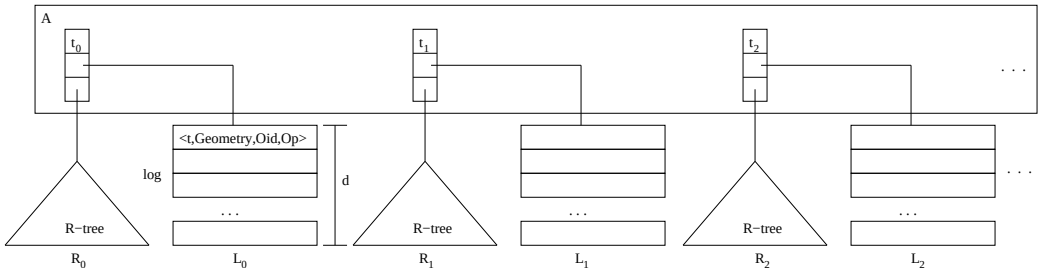


Figura 4.1: Esquema general del SEST-Index.

4.2 Estructura de datos

La estructura de datos para el R-tree es básicamente la misma que la propuesta en [Gut84]. La bitácora se implementa por medio de una lista encadenada simple de bloques o páginas (nodos). Las entradas en los nodos de una bitácora tienen la estructura $\langle t, Geom, Oid, Op \rangle$, donde t corresponde al tiempo en el cual la modificación (por ejemplo, la inserción de un nuevo elemento) sucedió, *Oid* es el identificador del objeto, *Geom* corresponde a los valores de la componente espacial y que depende del tipo de objeto espacial que se requiere almacenar, y Op indica el tipo de operación o evento (es decir, inserción o eliminación). Si los objetos son puntos en un espacio de dos dimensiones, entonces *Geom* será un par de coordenadas (x,y). Si los objetos son polígonos, entonces *Geom* será un conjunto de puntos que lo definen o su aproximación espacial, por ejemplo un MBR.

En el SEST-Index se consideran dos tipos de operaciones o eventos: *move_in* (un objeto se mueve a una nueva posición) y *move_out* (un objeto deja de permanecer en su última posición). De esta forma, la creación de un objeto se modela como un *move_in*, una eliminación como un *move_out*, y un objeto que cambió su posición y/o forma se modela como un *move_out* seguido de una operación *move_in*. Los snapshots se encuentran indexados por el tiempo. Para ello se utiliza un arreglo A, tal que $A_i.t$ indica el tiempo en que el $i-$ésimo snapshot fue creado, $A_i.R$ es el R-tree del snapshot, y $A_i.L$ es la bitácora de eventos producidos entre $A_i.t+1$ y $A_{i+1}.t$. En el SEST-Index también se considera un parámetro d (medido en cantidad de bloques de disco) que define el largo máximo de las bitácoras.

4.2.1 Operaciones

4.2.1.1 Consultas de tipo time-slice

Para procesar una consulta de tipo *time-slice* (Q,t), el primer paso es encontrar el correspondiente snapshot de acuerdo al instante de tiempo t especificado en la consulta. Este snapshot corresponde al creado en el último instante de tiempo dentro del intervalo $[0,t]$. Luego, se hace una búsqueda espacial usando el mismo método definido por [Gut84] (descrito en el Alg. 2.1) sobre el R-tree seleccionado en el paso anterior. El conjunto de objetos obtenidos por la búsqueda anterior se actualiza con las entradas de la correspondiente bitácora (ver Alg. 4.1).

```
 1: TimeSliceQuery(Rectangle Q, Time t)
 2: {S es un conjunto para almacenar la respuesta}
 3: Encontrar la última entrada i en A tal que A_i.t ≤ t
 4: S = SearchRtree(A_i.R, q) {Se recuperan todos los objetos en A_i.R que se intersectan con Q en el instante A_i.t }
 5: for cada entrada e ∈ A_i.L tal que e.t ≤ t do
 6:    if e.Geom Intersect(Q) then
 7:       if e.Op = move_in then
 8:          S = S ∪ {e.Oid}
 9:       else
10:          S = S − {e.Oid}
11:       end if
12:    end if
13: end for
14: return S
```

Alg. 4.1: Algoritmo para procesar una consulta de tipo *time-slice*.

4.2.1.2 Consultas de tipo time-interval

El procedimiento para procesar este tipo de consultas es similar al seguido para las consultas de tipo *time-slice*. Así, para evaluar o procesar una consulta de tipo *time-interval*, $(Q,[t_i,t_f])$, primero se ubica el último snapshot creado en el intervalo $[0,t_i]$, sea $t \leq t_i$ este instante de tiempo. Luego, haciendo una búsqueda en el R-tree seleccionado, se recupera el conjunto de los objetos que se intersectan espacialmente con Q. Posteriormente este conjunto se actualiza con los eventos ocurridos entre $(t,t_i]$, produciendo la respuesta para el instante t_i. Seguidamente, el conjunto se actualiza con los eventos ocurridos en el intervalo $(t_i,t_f]$, obteniendo los objetos "vivos" en cada instante de tiempo del intervalo $(t_i,t_f]$ (ver Alg. 4.2).

```
 1: IntervalQuery(Rectangle Q, Time t_i, Time t_f)
 2: G = ∅ {G es un conjunto para almacenar la respuesta}
 3: Encontrar la última entrada i en A tal que A_i.t ≤ t_i
 4: S = SearchRtree(A_i.R, q) {Se obtienen todos objetos de A_i.R que intersectan a Q en el instante A_i.t}
 5: L = A_i.L
 6: k = i
 7: Se actualiza S con los eventos de la bitácora L la cual se encuentra en el intervalo [t,t_i] (similar al procedimiento
    seguido para evaluar una consulta de tipo time-slice)
 8: G = S
 9: if todas las entradas en la bitácora L fueron procesadas then
10:    k = k+1
11:    L = A_k.L
12: end if
13: Sea t_s el siguiente instante de tiempo después de t_i almacenado en L
14: while t_s ≤ t_f y aún quedan entradas que procesar en L do
15:    Actualizar S con los eventos de L producidos en t_s
16:    G = G ∪ S
17:    if todas las entradas en L fueron procesadas then
18:       k = k+1
19:       L = A_k.L
20:    end if
21:    Sea t_s el siguiente instante de tiempo almacenado en L
22: end while
23: return G
```

Alg. 4.2: Algoritmo para procesar una consulta de tipo *time-interval*.

4.2.1.3 Consultas sobre eventos

Una de las novedades del SEST-Index, y también del SEST_L (detallado en el Capítulo 5), es la capacidad para procesar, no solamente consultas de tipo *time-slice* y *time-interval*, si no que además consultas sobre los eventos almacenados en las bitácoras. Por ejemplo, dada una región Q y un instante de tiempo t, una consulta de este tipo permite encontrar el total de objetos que salió del rectángulo Q o ingresó al mismo en el instante t. Consultas de este tipo son posible y útil en aplicaciones que requieren analizar patrones de movimiento de los objetos [Wor05, GW05].

Una implementación simple de este tipo de consultas con el SEST-Index consiste en usar un arreglo B para ubicar el primer bloque de la bitácora *Bid* que almacena el primer evento ocurrido en t. Cada entrada de B ocupa poca memoria (8 bytes), por lo tanto, es posible guardar varias entradas en un bloque de disco. Por ejemplo, para bloques de tamaño 1.024 bytes es posible

almacenar aproximadamente 128 entradas. Podemos notar que para evaluar este tipo de consultas no se requiere accesar el R-tree (Alg. 4.3).

```
 1: EventQuery(Rectangle Q, Time t, Array B)
 2: Encontrar la entrada i en el arreglo B tal que B_i.t = t
 3: L = B_i.Bid
 4: terminado = false
 5: oi = 0 {cantidad de objetos que entran Q en t }
 6: os = 0 {cantidad de objetos que dejan Q en t }
 7: while no terminado do
 8:    for por cada entrada e ∈ L tal que e.t = t do
 9:       if e.Geom Intersect(q) then
10:          if e.Op = Move_in then
11:             oi = oi + 1
12:          else
13:             os = os + 1
14:          end if
15:       end if
16:    end for
17:    if e.t > t then
18:       terminado = true
19:    else
20:       L = siguiente bloque de bitácora.
21:    end if
22: end while
23: return (oi, os)
```

Alg. 4.3: Algoritmo para procesar consultas sobre *eventos*.

4.2.1.4 Actualización de la estructura de datos

Este método se encarga de actualizar la estructura de datos conforme ocurren los eventos. Se asume que se cuenta con una lista con todos los eventos que han ocurrido en un instante de tiempo t y con todos los objetos "vivos" en la base de datos justo antes de t. Lo que hace el algoritmo es actualizar los objetos "vivos" con los eventos producidos en t. Luego se verifica si el parámetro d satisface el largo asignado a las bitácoras considerando los nuevos eventos. Si el largo de la bitácora es mayor que d, entonces se crea un nuevo snapshot, lo que implica crear un nuevo R-tree y una nueva entrada en el arreglo A. Si el valor de d aún no se alcanza, entonces los eventos simplemente se almacenan en la bitácora (Alg. 4.4).

4.3 Evaluación Experimental

Con el objetivo de evaluar el rendimiento del SEST-Index, éste fue comparado con el HR-tree en términos de almacenamiento y eficiencia de las consultas (*time-slice*, *time-interval* y sobre eventos). Se eligió comparar con HR-tree ya que éste es el método conocido hasta ahora que resuelve de manera más eficiente las consultas de tipo *time-slice*. Con HR-tree es posible procesar consultas sobre eventos utilizando las respuestas de las consultas *time-slice*. Por ejemplo, para obtener cuántos objetos entraron a un área determinada en un instante t, se resuelve con HR-tree realizando una consulta *time-slice* en el instante t y otra en el instante inmediatamente anterior

```
1: InsertChanges(Time t, Changes c, SnapShot F) {t es el instante de tiempo en el cual ocurren los eventos, c es
   una lista con los eventos ocurridos en t y F corresponde a los objetos con sus posiciones espaciales en el instante
   de tiempo que precede inmediatamente a t}
2: Sea i la última entrada de A.
3: Actualizar F con los eventos de c
4: Insertar los objetos de c al final de la bitácora A_i.L
5: if la cantidad de eventos en c más los eventos almacenados en la bitácora A_i.L es mayor que d then
6:    {Crear un nuevo snapshot en el SEST-Index}
7:    A_{i+1}.t = t
8:    A_{i+1}.R = R-tree con los objetos "vivos" en F
9:    A_{i+1}.L = ∅
10: end if
```

Alg. 4.4: Algoritmo para actualizar la estructura del SEST-Index.

para luego verificar que objetos se encuentra en el conjunto respuesta de la primera consulta pero no en el conjunto respuesta de la segunda. De esta forma podemos comparar la eficiencia de las consultas sobre eventos al procesarlas con SEST-Index y con HR-tree y extender las conclusiones a otros métodos de acceso propuestos en la literatura y que evaluan dichas consultas utilizando los resultados de dos o más consultas de tipo *time-slice*.

Para obtener los conjuntos de datos espacio-temporales se utilizó el generador GSTD [TSN99]. Se generaron conjuntos con 1.000, 3.000 y 5.000 objetos (puntos) distribuidos uniformemente dentro del espacio en el instante 0. Posteriormente los objetos se movieron al azar durante 50 instantes de tiempo hasta alcanzar el instante 1,0. Se consideraron cuatro porcentajes de movilidad (1%, 3%, 5% y 7%), es decir, porcentaje de objetos que cambian su posición de un instante al siguiente. También se consideraron 3 valores para el parámetro d que correspondieron a 4, 8 y 12 bloques de 1.024 bytes. Para medir el almacenamiento ocupado por el SEST-Index y el HR-tree se usó el número de bloques utilizados por la estructura después de insertar todos los objetos y los eventos. Por otra parte, la eficiencia de las consultas fue medida como el número de bloques accesados, promediando 100 consultas del mismo tipo.

4.3.1 Almacenamiento utilizado por el SEST-Index

La Figura 4.2 muestra que el SEST-Index usa menos espacio que el HR-tree. Esta diferencia aumenta en la medida que el porcentaje de movilidad o el número de objetos disminuye. También es posible observar que conforme el tamaño del parámetro d disminuye, el espacio requerido por el SEST-Index aumenta, llegando al caso extremo en que necesita casi el mismo almacenamiento que el HR-tree para $d = 4$ y un porcentaje de movilidad de un 7%. Esta debilidad del SEST-Index se explica por el hecho de que con d pequeño es muy probable que quepan en cada bitácora los eventos correspondiente a un sólo instante de tiempo. En tal situación será necesario crear un snapshot por cada instante de tiempo y, por lo tanto, la estructura degenera en muchos R-trees individuales (uno por cada instante de tiempo almacenado en la base de datos).

Una idea simple para reducir el espacio usado por el SEST-Index es considerar la estrategia básica del HR-tree, esto es, al momento de crear un nuevo R-tree en un snapshot, reutilizar parte del R-tree previo. Esta idea se evaluó resultando que el ahorro es muy pequeño, entre 7% y 8%, dependiendo del número de objetos, del porcentaje de movilidad, y del parámetro d (ver Tabla 4.1). El escaso ahorro se explica por el hecho que los eventos producidos entre snapshots provocan que la reutilización de los R-tree, en promedio, sea baja y, de esta forma, mientras mayor sea la cantidad

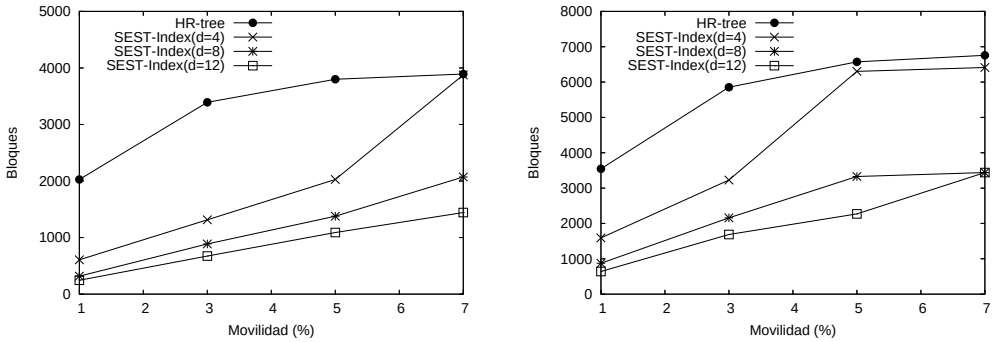


Figura 4.2: Espacio utilizado por el SEST-Index.

de eventos que pueda almacenar la bitácora (parámetro d en la Tabla 4.1), menor será el ahorro de espacio.

Número de objetos	Porcentajes de ahorro		
	$d=4$	$d=8$	$d=12$
1.500	7,3	8,6	9,9
2.500	8,6	7,8	8,1
3.500	10,0	8,3	8,1
4.500	9,6	9,0	7,0
7.500	10,8	9,4	8,5
10.500	7,2	8,6	6,3
13.500	10,0	7,0	7,5
12.500	7,7	4,5	5,7
17.500	8,1	5,2	5,2
22.500	5,6	5,6	5,5
Promedios	8.4	7.4	7.1

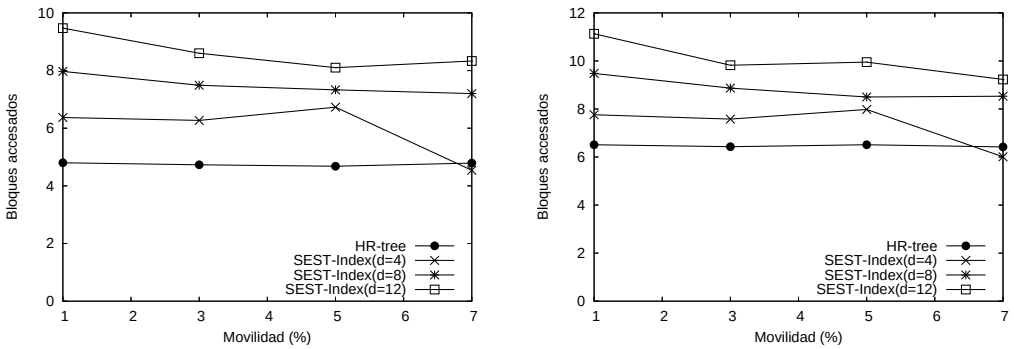
Tabla 4.1: Porcentajes de ahorro de almacenamiento del SEST-Index conseguidos por medio de la estrategia de reutilización del espacio del HR-tree.

4.3.2 Consultas *time-slice*.

Con el propósito de comparar ambos métodos de acceso se ejecutaron 100 consultas de tipo *time-slice* (Q,t). Q correspondió a rectángulos formados por el 5% y 10% de cada dimensión. Tanto la posición de Q en el espacio como el valor de t fueron elegidos de manera aleatoria. Sólo se muestran los resultados correspondiente a 3.000 objetos ya que para 1.000 y 5.000 se llegó a conclusiones similares.

Es posible observar en la Figura 4.3 que el SEST-Index necesita leer más bloques que el HR-tree para este tipo de consultas. Para la mayoría de los casos el SEST-Index, en promedio, necesitará realizar $\frac{d}{2}$ accesos más que el HR-tree, que corresponden al procesamiento adicional de bloques de bitácora. También es posible observar en la Figura 4.3 que el rendimiento del

HR-tree se mantiene constante frente a las variaciones de la movilidad de los objetos. Esto se debe a que el HR-tree necesita recorrer siempre un sólo R-tree del conjunto de R-trees virtuales, que en promedio tienen la misma altura. La Figura 4.3 también confirma nuestros pronósticos respecto del comportamiento del SEST-Index en función del parámetro d. Es posible observar que el rendimiento del SEST-Index mejora en la medida que el parámetro d disminuye.



a) $q = 5\%$ b) $q = 10\%$

Figura 4.3: Número de bloques accesados por una consulta *time-slice* utilizando el SEST-Index (rango espacial formado por el q% de la longitud de cada dimensión).

4.3.3 Consultas *time-interval*.

Similar a como se hizo con las consultas de tipo *time-slice*, también se promediaron 100 consultas de tipo *time-interval* $(Q, [t_i, t_f])$. Q correspondió a rectángulos formados por el 5% y 10% de cada dimensión. Por otra parte, las longitudes de los intervalos temporales fueron de 5 y 10 unidades de tiempo. Tanto la posición de Q en el espacio como los límites de los intervalos temporales fueron elegidos de manera aleatoria. Sólo se muestran los resultados correspondiente a 3.000 objetos ya que para 1.000 y 5.000 se llegó a conclusiones similares.

Es posible observar en las Figuras 4.4 y 4.5 que el SEST-Index requiere accesar menos bloques que el HR-tree cuando el porcentaje de movilidad es bajo (menor que 5% si el intervalo temporal es de 5 unidades). En la medida que la longitud del intervalo temporal aumenta, la ventaja del SEST-Index con respecto al HR-tree también crece. Esta ventaja se debe al hecho de que el SEST-Index sólo necesita leer un R-tree y luego secuencialmente los bloques de las bitácoras. El HR-tree, en cambio, debe leer un R-tree creado en cada instante de tiempo dentro del intervalo $[t_i, t_f]$. También es posible observar que el rendimiento del HR-tree permanece constante frente a las fluctuaciones de las movilidades de los objetos.

4.3.4 Consultas sobre eventos

De la misma manera que en las evaluaciones realizadas para las consultas anteriores, se consideraron 100 consultas con rectángulos (Q) formados por el 5% y 10% de cada dimensión. Tanto las coordenadas de Q, como los instantes de tiempo se eligieron de manera aleatoria. En esta sección sólo se discuten los resultados para 3.000 objetos, pues para 1.000 y 5.000 objetos las

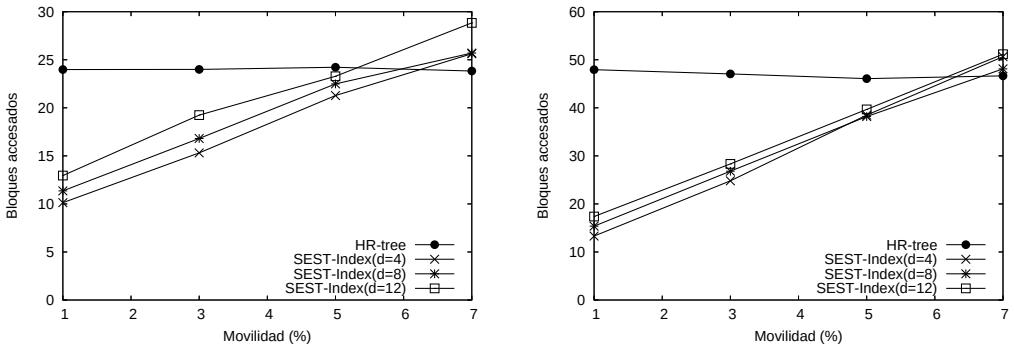


Figura 4.4: Número de bloques accesados por una consulta *time-interval* utilizando el SEST-Index (rango espacial formado por el q% de la longitud de cada dimensión y longitud del intervalo temporal igual a 5 unidades de tiempo).

conclusiones fueron similares. Las consultas fueron procesadas utilizando el Alg. 4.3 en el cual el número de bloques accesados sólo depende del porcentaje de movilidad y no del parámetro d o del tamaño del área de la consulta. En la Figura 4.6 es posible apreciar que el SEST-index mantiene un rendimiento más o menos constante frente a las variaciones del parámetro d.

La Figura 4.7 muestra el número de bloques de disco que fue necesario leer para procesar consultas sobre eventos para el HR-tree y el SEST-Index, considerando 3.000 objetos y $d = 8$ (se obtuvieron resultados similares para otros valores de d). Es posible observar que el SEST-Index necesita accesar muchos menos bloques que el HR-tree. Esto se debe al hecho de que el HR-tree requiere leer dos R-trees consecutivos para descubrir los eventos, mientras que con el SEST-Index es posible recuperar los eventos de manera directa desde las bitácoras, sin necesidad de recorrer ningún R-tree.

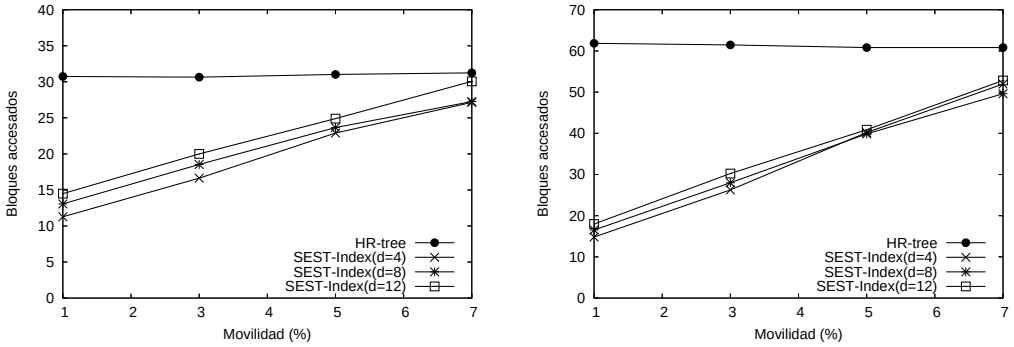


Figura 4.5: Número de bloques accesados por una consulta *time-interval* utilizando el SEST-Index (rango espacial formado por el q% de la longitud de cada dimensión y longitud del intervalo temporal igual a 10 unidades de tiempo).

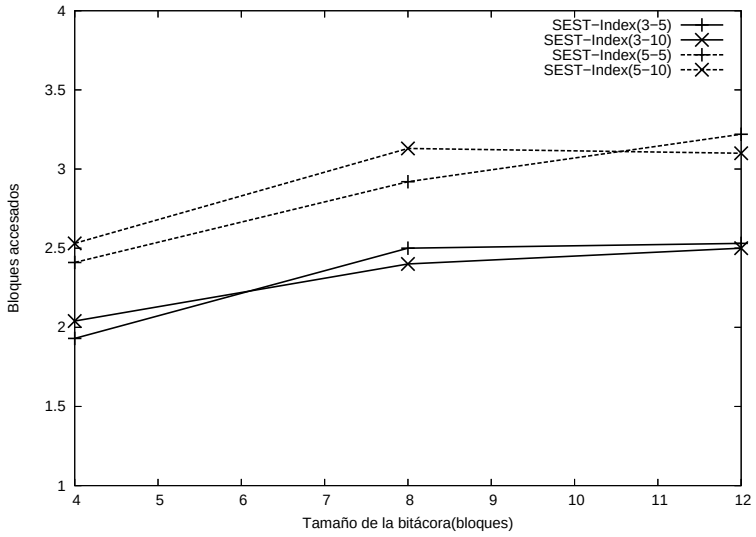


Figura 4.6: Bloques accesados por el SEST-Index para procesar consultas sobre eventos para 3.000 objetos. *SEST-Index(i-j)* indica consultas para conjunto de objetos con una movilidad de i% y el rango espacial de la consulta formado por j% de cada dimensión.

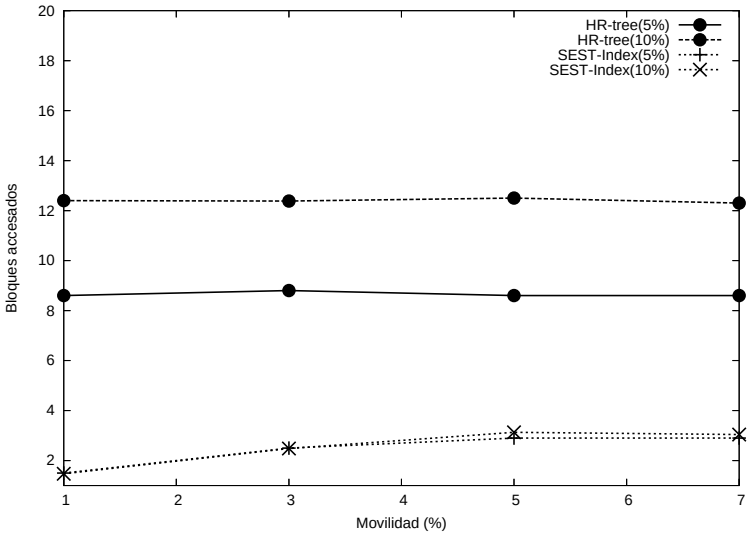


Figura 4.7: Bloques accesados por el HR-tree y el SEST-Index para procesar consultas sobre eventos para 3.000 objetos. *XX(i%)* indica el SEST-Index o el HR-tree para consultas con rango espacial formado por $i\%$ de cada dimensión.

4.4 Variante de SEST-Index

A pesar de que el SEST-Index hace un uso eficiente del almacenamiento para movilidades bajas (entre 1% y 13%) y supera al HR-tree para consultas de tipo *time-interval* y sobre eventos para movilidades entre 1% y 7%, es posible apreciar que una de sus principales desventajas es el rápido crecimiento del almacenamiento (ver Figura 4.8) el cual se explica por: (i) El mayor número de snapshots (R-trees) que es necesario crear en la medida que el porcentaje de movilidad y/o la cantidad de objetos aumenta y (ii) cada vez que se crea un nuevo R-tree (nuevo snapshot), todos los objetos se duplican incluyendo aquellos que no han sufrido modificaciones entre snapshots consecutivos.

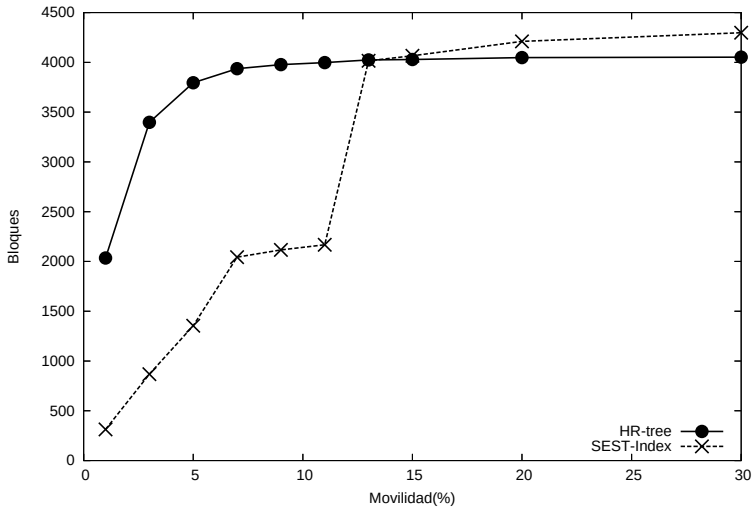


Figura 4.8: Crecimiento del tamaño del índice versus el porcentaje de movilidad (3.000 objetos).

Una forma simple de resolver el último problema es particionando el espacio original en un conjunto de subespacios o regiones disjuntas y aplicando los conceptos de snapshots y bitácoras sobre tales regiones. Para obtener las regiones se utilizó un K-D-B-tree (se eligió esta estructura porque precisamente divide el espacio en un conjunto disjunto de subregiones y la unión de las subregiones permite obtener el espacio original). Luego, las bitácoras se asignaron a las subregiones del nivel más bajo, es decir, a las subregiones formadas por los elementos de los nodos hoja del K-D-B-tree (en adelante a esta idea le llamaremos simplemente variante). Con esta idea se evita duplicar en cada nuevo snapshot los objetos que no han sufrido modificación alguna en sus atributos espaciales. La Figura 4.9 muestra un esquema general de la variante del SEST-Index.

El índice temporal I se usa para ubicar rápidamente el snapshot apropiado de acuerdo al tiempo dado en la consulta. Para procesar una consulta de tipo *time-slice*(Q,t) se procede de la siguiente manera. En primer lugar se seleccionan todas las regiones que se intersectan con Q. Luego, por cada una de estas regiones, y usando el índice I, se ubica el correspondiente snapshot. El conjunto de objetos almacenados en el snapshot se actualiza con los eventos almacenados en la bitácora y que se han producido hasta el instante de tiempo t. El conjunto resultante es parte de la

respuesta y la respuesta definitiva se logra procesando cada una de las regiones intersectadas por Q.

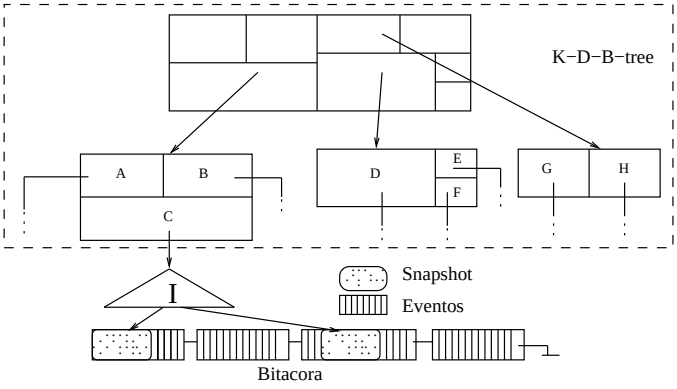


Figura 4.9: Esquema general de la variante del SEST-Index.

Se realizó un experimento preliminar para estudiar el comportamiento de esta variante, que consistió en compararla con el HR-tree y el SEST-Index usando 3.000 objetos (puntos), con 4 porcentajes de movilidad (1%, 3%, 5% y 7%) y tres tamaños diferentes de bitácora (4, 8 y 12 bloques). Los resultados muestran que la variante, en promedio, no necesita más de un 10% (ver Figura 4.10) del almacenamiento requerido por el HR-tree y de un 35% del ocupado por SEST-Index para el mismo conjunto de objetos y el mismo número de eventos.

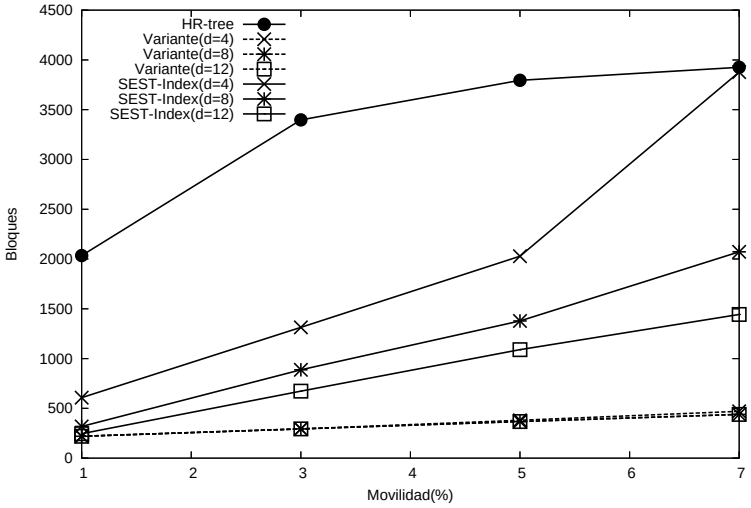


Figura 4.10: Espacio utilizado por la variante, SEST-Index y HR-tree.

En cuanto a las consultas de tipo *time-slice*, el HR-tree supera a la variante por poco más del doble, es decir, la variante necesita accesar casi el doble de nodos para procesar este tipo de consultas (ver Figura 4.11). SEST-Index también supera a la variante para porcentajes de

movilidad mayores que 3%. Es interesante hacer notar que el HR-tree es el método de acceso espacio-temporal conocido que procesa de manera más eficiente este tipo de consultas, ya que es equivalente a ejecutar una consulta espacial de tipo WQ (window query) sobre un R-tree. Sin embargo, la variante superó por lejos al HR-tree y al SEST-Index en consultas de tipo *time-interval* tal como se puede apreciar en la Figura 4.12. De hecho la variante puede procesar la misma consulta en sólo un 25% del tiempo consumido por el HR-tree.

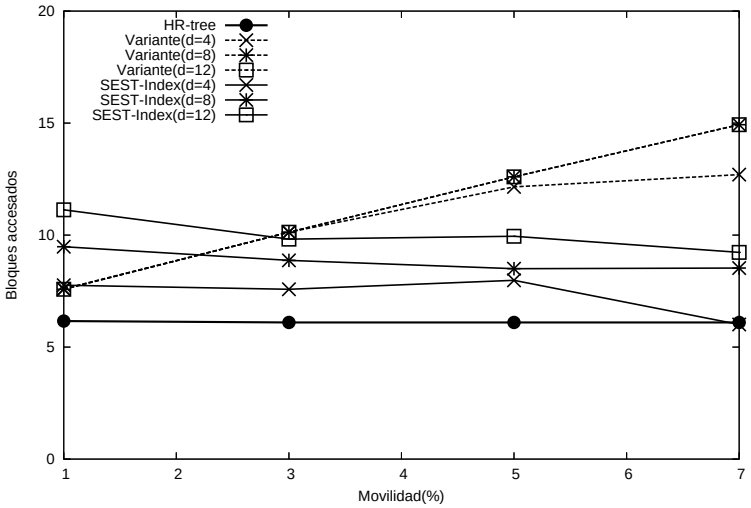


Figura 4.11: Bloques accesados por una consulta de tipo *time-slice* con rango espacial de la consulta formado por el 10% de cada dimensión (HR-tree, SEST-Index y la variante).

La variante discutida en esta sección permite vislumbrar que la asignación de las bitácoras a regiones de tamaño adecuado garantiza un ahorro de espacio y un procesamiento eficientemente de las consultas de tipo *time-interval*, lo que la transforma en una idea muy promisoria a la luz de estos resultados preliminares. Debido a que se utilizó un K-D-B-tree para generar las particiones a las cuales asignar las bitácoras, la variante presenta dos limitaciones importantes: (i) el espacio de trabajo debe ser fijo y (ii) los objetos deben ser de tipo punto. Estas limitaciones obligan a explorar otros métodos de acceso y a diseñar nuevas estructuras de datos que permitan eliminar estas restricciones y que presenten rendimientos, al menos, similares a los de la variante.

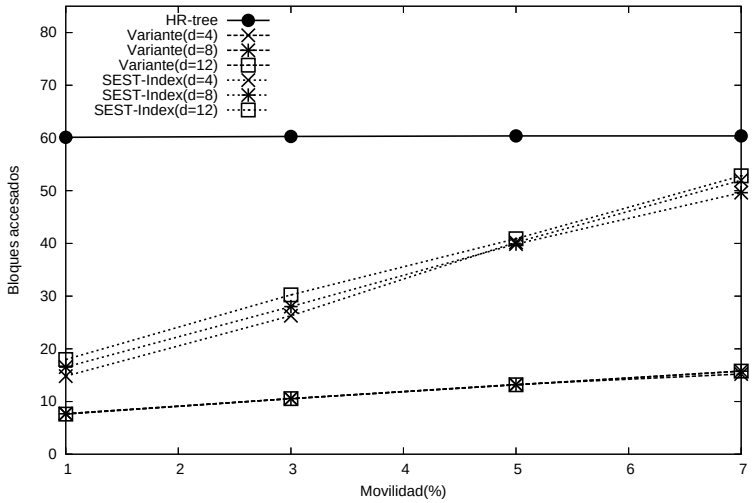


Figura 4.12: Bloques accesados por una consulta de tipo *time-interval* con rango espacial de la consulta está formado por el 10% de cada dimensión y longitud temporal de 10 unidades de tiempo (HR-tree, SEST-Index y la variante).

4.5 Modelo de costo de SEST-Index

En esta sección se describe un modelo de costo para el SEST-Index que permite estimar el almacenamiento ocupado por la estructura y el tiempo requerido para procesar consultas espacio-temporales. El modelo se evaluó experimentalmente para demostrar su capacidad de predicción. El modelo asume que, tanto la posición inicial de los objetos, como las posiciones alcanzadas por ellos en los subsiguientes instantes de tiempo, siguen una distribución uniforme. Por otro lado, se consideró un espacio de dos dimensiones para definir los atributos espaciales de los objetos. En la Tabla 4.2[2] se describen las variables usadas en la definición del modelo.

Símbolo	Definición
ai	Amplitud del intervalo temporal dado en una consulta de tipo *time-Interval*
c	Cantidad de cambios que se pueden almacenar en un bloque (un cambio es equivalente a dos operaciones o eventos, un *move_in* más un *move_out*).
cc	Porcentaje de utilización de un nodo del R-tree
d	Tamaño de la bitácora en bloques de disco
D_0	Densidad inicial del conjunto de objetos espaciales
DA	Cantidad de bloques o nodos accesados en una consulta espacial
$DA^{SEST-Index}_{time-Interval}$	Cantidad de bloques accesados en una consulta *time-interval* usando el SEST-Index
$DA^{SEST-Index}_{time-slice}$	Cantidad de bloques accesados en una consulta *time-slice* usando el SEST-Index
$DA^{SEST_L}_{time-Interval}$	Cantidad de bloques accesados en una consulta *time-interval* usando el SEST$_L$
$DA^{SEST_L}_{time-slice}$	Cantidad de bloques accesados en una consulta *time-slice* usando el SEST$_L$
$DASG^{SEST_L}_{time-interval}$	Cantidad de bloques accesados en una consulta *time-interval* usando el SEST$_L$ con snapshots globales
f	Capacidad promedio (fanout) de un nodo del R-tree, $f = cc \cdot M$
h	Altura del R-tree
il	Cantidad de instantes de tiempo posibles de almacenar por bitácora entre snapshots consecutivos
l	Cantidad de cambios que se pueden almacenar entre dos snapshots consecutivos, $l = c \cdot d$
igs	Cantidad de instantes de tiempo entre snapshots globales (mide cada cuantos instantes de tiempo se crea un snapshot global)
M	Máxima capacidad de un nodo del R-tree (máximo número de entradas)
M_1	Máxima capacidad de un nodo del R-tree para alcanzar una densidad dada
N	Cantidad de objetos del conjunto
nl	Capacidad de la bitácora (número de cambios) ajustada a un número entero de instantes de tiempo
nt	Cantidad de instantes de tiempo almacenados en la estructura
p	Porcentaje de movilidad
q	Ancho del rango espacial (rectángulo) de la consulta a lo largo de cada dimensión, expresado como una fracción del espacio total
ls	Valor umbral que determina cuándo crear un nuevo snapshot global
NB	Número de bitácoras
TN	Total de bloques utilizados por el R-tree
$TB^{SEST-Index}$	Tamaño (en bloques) del SEST-Index
TB^{SEST_L}	Tamaño (en bloques) del SEST$_L$
$TBSG^{SEST_L}$	Tamaño (en bloques) del SEST$_L$ con snapshots globales

Tabla 4.2: Variables utilizadas en la definición de los modelos de costos.

[2]En esta tabla se encuentran descritas todas las variables que se utilizan en la especificación de todos los modelos definidos en esta tesis.

4.5.1 Modelo de costo para estimar el espacio utilizado por el SEST-Index

El almacenamiento ocupado por el SEST-Index se puede estimar considerando el espacio destinado a almacenar los R-trees de los snapshots, más el espacio ocupado por las bitácoras. Con la Ec. (4.1) se puede calcular la cantidad de instantes de tiempo cuyos cambios es posible almacenar en una bitácora.

$$il = \left\lceil \frac{l}{N \cdot p} \right\rceil \tag{4.1}$$

Utilizando *il* se puede obtener la cantidad de R-trees (snapshots) por medio de la expresión $\left\lceil \frac{nt}{il} \right\rceil$. De manera similar, con la expresión $\left(\left\lceil \frac{nt}{il} \right\rceil - 1\right)$ se obtiene el número de bitácoras completamente llenas.

En el SEST-Index, si en el instante t se intenta crear un nuevo snapshot, pero aún quedan cambios producidos en t que almacenar, tal decisión se posterga hasta que todos los cambios se almacenen en la bitácora. En otras palabras, no se acepta que los cambios ocurridos en un instante de tiempo queden almacenados en bitácoras distintas. Para modelar este comportamiento se necesita redefinir la capacidad de las bitácoras (l) con la Ec. (4.2).

$$nl = N \cdot p \cdot il \tag{4.2}$$

De esta forma, el número de bloques usados por el SEST-Index está dado por la Ec. (4.3). En esta ecuación, el primer término representa el almacenamiento (número de bloques promedio) ocupado por los R-trees. El segundo estima la cantidad promedio de bloques destinados a bitácoras completamente llenas, es decir, conteniendo *nl* cambios. Finalmente, el tercer término estima el número de bloques ocupados, en promedio, para almacenar los eventos ocurridos después del último snapshot.

$$TB^{SEST-Index} = \left\lceil \frac{nt}{il} \right\rceil \cdot TN + \left(\left\lceil \frac{nt}{il} \right\rceil - 1\right) \cdot \left\lceil \frac{nl}{c} \right\rceil + \left\lceil \frac{nt - \lfloor \frac{nt}{il} \rfloor \cdot il}{il} \cdot \frac{nl}{c} \right\rceil \tag{4.3}$$

Utilizando la simplificación dada para TN en la Ec. (2.2), asumiendo que la última bitácora almacenará en promedio $\frac{l}{2}$ cambios y eliminando algunos de los redondeos es posible obtener una expresión aproximada para la Ec. (4.3) (ver Ec. (4.4)) que muestra de manera más intutiva la influencia de los parámetros de la estructura sobre el tamaño del índice.

$$TB^{SEST-Index} \approx \frac{nt \cdot N \cdot p}{l} \cdot \left(\log_f N + \frac{N}{f} + \left\lceil \frac{l}{c} \right\rceil \right) \tag{4.4}$$

En efecto, utilizando la Ec. (4.4), podemos analizar el impacto del parámetro $l = d \cdot c$ (capacidad de la bitácora) sobre el almacenamiento. Es fácil ver que en la medida que el valor de l crece disminuye el almacenamiento para los snapshots. Esto en cambio, no tiene efecto sobre el almacenamiento de las bitácoras, que se estabiliza en $\frac{nt \cdot N \cdot p}{c}$. Aclaramos que en las evaluaciones realizadas sobre el modelo se usó la Ec. (4.3).

4.5.2 Modelo de costo para estimar la eficiencia de las consultas con el SEST-Index

Dado que se consideró un espacio de dos dimensiones, se derivó la Ec. (4.5) a partir de la Ec. (2.4) para el caso particular $n = 2$ y $q = q_1 = q_2$. En la Ec. (4.5), D_j se obtiene con la Ec. (2.7).

$$DA = 1 + \sum_{j=1}^{h} \left(\sqrt{D_j} + q \cdot \sqrt{\frac{N}{f^j}} \right)^2 \quad (4.5)$$

El costo temporal del SEST-Index (cantidad de accesos a bloques o nodos) se puede estimar como la suma de los accesos a nodos del R-tree (snapshot) y los accesos a bloques de las bitácoras. La Ec. (4.6) estima la cantidad de accesos que, en promedio, se requieren para evaluar una consulta *time-slice*, (Q,t). El primer término de la Ec. (4.6) se obtiene con la Ec. (4.5) utilizando Q como rango espacial. El segundo término estima la cantidad de bloques o nodos de la bitácora que, en promedio, se necesitan accesar para procesar los cambios producidos en el intervalo $(ts,t]$, donde t_s es el instante de tiempo más cercano a t, con $t_s \leq t$ en el cual se creó un snapshot.

$$DA^{SEST-Index}_{time-slice} = DA + \left\lceil \frac{nl}{2 \cdot c} \right\rceil \quad (4.6)$$

Se puede estimar el número de accesos realizados por una consulta *time-interval* $(Q,[t_i,t_f])$ considerando que el costo de una consulta de este tipo corresponde al costo de una consulta *time-slice* (Q,t_i) más el costo de procesar todos los cambios ocurridos en el intervalo $(t_i,t_f]$. La Ec. (4.7) estima el promedio de bloques accesados por una consulta de tipo *time-interval*.

$$DA^{SEST-Index}_{time-interval} = DA^{SEST-Index}_{time-slice} + \left\lceil (ai-1) \cdot \frac{N \cdot p}{c} \right\rceil \quad (4.7)$$

Al igual que para la ecuación definida para el almacenamiento ($TB^{SEST-Index}$), también es interesante obtener una aproximación para la Ec. (4.7) que estime el costo de las consultas. Para ello se descartan los redondeos y se reemplaza *nl* por l en las Ecs. (4.6) y (4.7) con lo cual se obtiene una aproximación para $DA^{SEST-Index}_{time-interval}$ definida en la Ec. (4.8).

$$DA^{SEST-Index}_{time-interval} \approx DA + \frac{l}{2 \cdot c} + \frac{(ai-1) \cdot N \cdot p}{c} + 1 \quad (4.8)$$

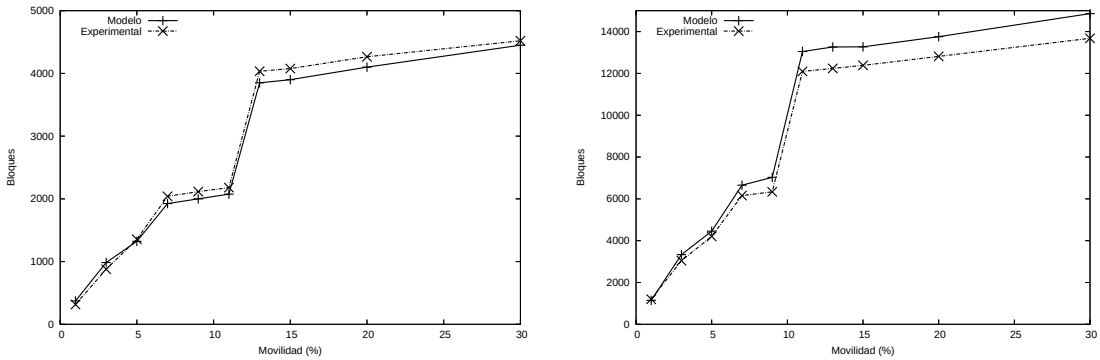
A partir de la Ec. (4.8) es fácil intuir el efecto de los parámetros en las consultas. Por ejemplo, si se disminuye el valor de l, entonces también disminuirá el tiempo de la consulta. Algo similar ocurre con el parámetro c. Con la Ec. (4.8) es posible fundamentar la afirmación realizada en la Sección 4.3.3 respecto de la explicación de la ventaja del SEST-Index sobre el HR-tree; en efecto, el costo total de una consulta de tipo *time-interval* está compuesto de un costo inicial ($DA + \frac{l}{2 \cdot c}$), más un costo adicional que depende de *ai* para valores constantes de N, p y c. En la validación del modelo se utilizó la Ec. (4.7).

4.5.3 Evaluación experimental del modelo de costo

El modelo de costo se evaluó experimentalmente considerando conjuntos de datos generados con GSTD [TSN99]. Estos experimentos utilizaron 3.000 y 5.000 objetos (puntos) con 50 y 100 instantes de tiempo. Se consideró una capacidad por nodo (bloque) de un R-tree de 50 entradas y que cada nodo mantiene en promedio 34 entradas (68% de la capacidad de un nodo).

4.5.3.1 Estimación del almacenamiento

Se experimentó con movilidades entre 1% y 30% y un tamaño de bitácora de 8 y 12 bloques para los conjuntos de 3.000 y 5.000 objetos respectivamente. La Figura 4.13 muestra la capacidad de predicción del modelo del SEST-Index, el cual presentó un error relativo promedio de un 6%.



(a) $d = 8,\ N = 3.000,\ nt = 50$ (b) $d = 12,\ N = 5.000,\ nt = 100$

Figura 4.13: Estimación del almacenamiento ocupado por el SEST-Index

4.5.3.2 Estimación del rendimiento de las consultas *time-slice* y *time-interval*.

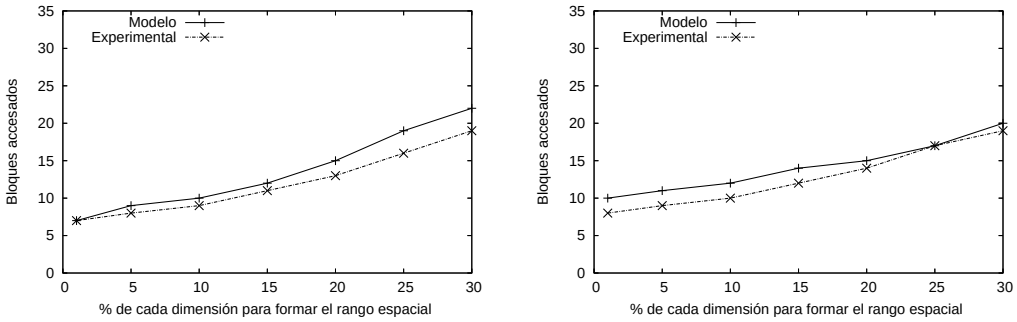
Para el caso de las consultas, también se consideró un tamaño de bitácora igual a 8 bloques. Los porcentajes de movilidad fueron de 5% y 11%. Las consultas consideraron áreas formadas por porcentajes de cada dimensión entre 1% y 30%, y la longitud del intervalo temporal se fijó a 10 instantes de tiempo. La Figuras 4.14 y 4.15 muestran la estimación del modelo para ambos tipos de consultas. Es posible observar que el modelo estima bastante bien el rendimiento de las consultas. Específicamente, para los experimentos realizados, el error relativo promedio alcanzó sólo a un 7%.

En la tabla 4.3 se muestra un resumen con los resultados más destacados de SEST-Index y la variante.

4.6 Conclusiones

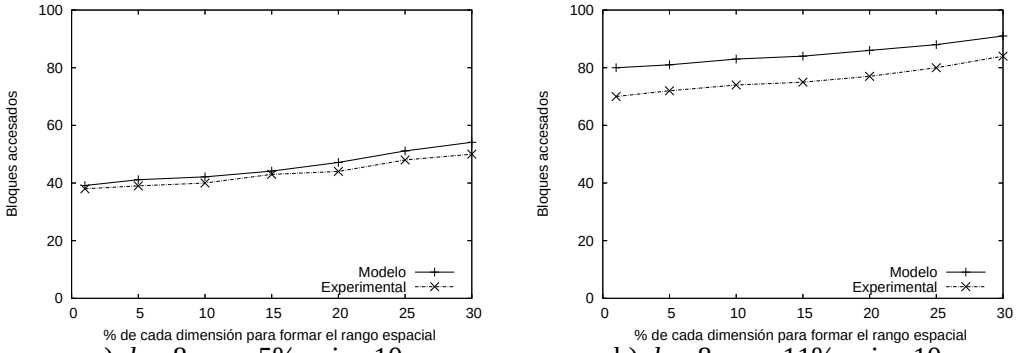
En este capítulo se ha propuesto un nuevo método de acceso espacio-temporal que combina la creación de snapshots y eventos. El método trata de establecer un compromiso entre el tiempo requerido por las consultas y el espacio para almacenar el índice.

Los experimentos realizados muestran que el SEST-Index necesita de menos espacio que el HR-tree para porcentajes de movilidad bajos (entre 1% y 13%). El SEST-Index también superó al HR-tree para consultas de tipo *time-interval* y sobre eventos. Las ventajas del SEST-Index sobre el HR-tree para procesar consultas de tipo *time-interval* aumentan en la medida que tanto el área como la longitud del intervalo temporal de la consulta también lo hacen. En el caso de las consultas de tipo *time-slice,* el SEST-Index necesitó de más accesos a bloques de disco que el HR-tree. Aquí



a) $d = 8,\ p = 5\%$ b) $d = 8,\ p = 11\%$

Figura 4.14: Estimación del rendimiento de las consultas *time-slice* por parte del modelo costo del SEST-Index



a) $d = 8,\ p = 5\%,\ ai = 10$ b) $d = 8,\ p = 11\%,\ ai = 10$

Figura 4.15: Estimación del rendimiento de las consultas *time-interval* por parte del modelo costo del SEST-Index

hay que hacer notar que el HR-tree es el método de acceso espacio-temporal, conocido hasta ahora, más eficiente para procesar este tipo de consultas.

Por otra parte, la variante del SEST-Index, descrita en este capítulo, presentó un rendimiento muy promisorio frente al HR-tree. Concretamente, la variante sólo necesitó de un 10% del almacenamiento ocupado por el HR-tree y de un 25% del tiempo requerido por HR-tree para procesar consultas de tipo *time-interval*. Sin embargo, y como ya comentamos, la variante presenta dos limitaciones importantes. Pese a ello las ideas detrás de la variante son las que se usan en el método de acceso espacio-temporal SEST_L discutido en el Capítulo 5. El SEST_L elimina las dos limitaciones de la variante y supera al MVR-tree en términos de almacenamiento y tiempo y constituye una parte medular de los aportes de esta tesis.

En este capítulo también se definió y validó un modelo de costo para estimar el almacenamiento del SEST-Index y la eficiencia de las consultas. El modelo presentó un error relativo promedio de un 6% en la estimación del almacenamiento y de 7% para las consultas.

I T E M	Evaluación
SEST-Index	
1) Almacenamiento	SEST-Index requiere de menos espacio que HR-tree para porcentajes de movilidad entre 1% y 13%.
2) Consultas	
a) *time-interval*	SEST-Index supera al HR-tree en la medida que el área y/o el largo del intervalo aumenta.
b) *sobre eventos*	SEST-Index resultó ser más eficiente que el HR-tree.
c) *time-slice*	HR-tree supera al SEST-Index.
3) Modelo de costo	
a) Almacenamiento	Error relativo promedio de 6%.
b) Consultas	Error relativo promedio de 7%.
Variante	
1) Limitaciones	Espacio de trabajo estático y sólo objetos de tipo punto.
2) Almacenamiento	Sólo requiere del 10% de HR-tree.
3) Consultas	25% de los accesos requeridos por HR-tree para consultas de tipo *time-interval.*

Tabla 4.3: Resultados destacados de SEST-Index y la variante.

Capítulo 5

SEST$_L$

5.1 Introducción

Tal como se demostró en el capítulo anterior, una de las principales desventajas del SEST-Index es el rápido crecimiento del almacenamiento ocupado por el índice en la medida que el porcentaje de movilidad aumenta entre instantes de tiempo. Esta desventaja se explica por el hecho de que en cada snapshot los objetos se duplican incluyendo aquellos que no han sufrido modificaciones. En el Capítulo 4 (Sección 4.4) se discute una variante del SEST-Index que precisamente apunta a resolver este problema. Sin embargo, dicha variante presenta dos limitaciones importantes: (i) los objetos a indexar deben ser puntos y (ii) la región o espacio donde ocurren los cambios debe ser fija. El SEST$_L$ supera estas dos limitaciones manteniendo un rendimiento muy similar al de la variante.

La idea del SEST$_L$ consiste en dividir, mediante un único R-tree, el espacio original en regiones al nivel de las hojas y asignarle una bitácora a cada una de estas regiones. Aquí una bitácora es una estructura que almacena (en las hojas) snapshots y eventos. La Figura 5.1 presenta un esquema general del SEST$_L$. Los rectángulos $R1, R2$ y $R3$ corresponden a una partición del espacio original, en un primer nivel, generadas por el algoritmo de inserción del R-tree en el instante t_0. Recursivamente los rectángulos A, B y C corresponden a una partición del subespacio $R1$ en el mismo instante de tiempo t_0. A su vez cada uno de estos rectángulos contiene las aproximaciones espaciales de los objetos (MBR) o su componente espacial (geometría) directamente. Tanto los nodos internos como los nodos hojas corresponden a bloques o páginas de disco. La Figura 5.1 también muestra la evolución del nodo A y que se define por los eventos que se producen en la región asociada a dicho nodo. En el instante t_0, A mantiene tres objetos los que conforman el primer snapshot de A. Conforme avanza el tiempo se van generando eventos en la región definida por el nodo A. Por ejemplo es posible que nuevos objetos alcancen dicha zona y otros salgan de ella, producto de los desplazamientos de los objetos en el espacio y que provocan modificaciones en el R-tree. Todos estos eventos se van registrando en la bitácora (como eventos) hasta que en el instante $t_1 > t_0$ se decide crear un nuevo snapshot con los objetos que permanecen dentro del área definida por el MBR del nodo A la cual ha crecido en t_1. El snapshot creado en el instante t_1 almacena cuatro objetos y representa el estado del nodo A en dicho instante de tiempo. Con SEST$_L$ es posible obtener cualquier estado intermedio del nodo A en el intervalo (t_0, t_1) lo cual se explica más adelante. De manera similar los eventos que se producen en el intervalo (t_1, t_2) se almacenan en la bitácora asociada a A para posteriormente, en el instante t_2, crear un nuevo

snapshot del nodo *A*, esta vez con cinco objetos. Podemos observar que los eventos producidos en el área de la región definida por el nodo *A* se ven reflejados en el nodo interno del cual depende, es decir, *R*1 en este caso.

En el SEST$_L$, tanto las áreas de las regiones asignadas a las hojas y los MBRs de los nodos internos del R-tree son siempre crecientes a lo largo del tiempo. Debido a esta situación el área solapada producida en los nodos internos del R-tree tiende a crecer produciendo un impacto negativo sobre el rendimiento de las consultas, ya que es necesario procesar muchas bitácoras para poder evaluarlas. Para resolver esta situación, la estructura usa snapshots globales (ver Sección 5.3), con lo cual periódicamente se reparticiona el espacio y se asignan bitácoras a estas nuevas regiones o particiones.

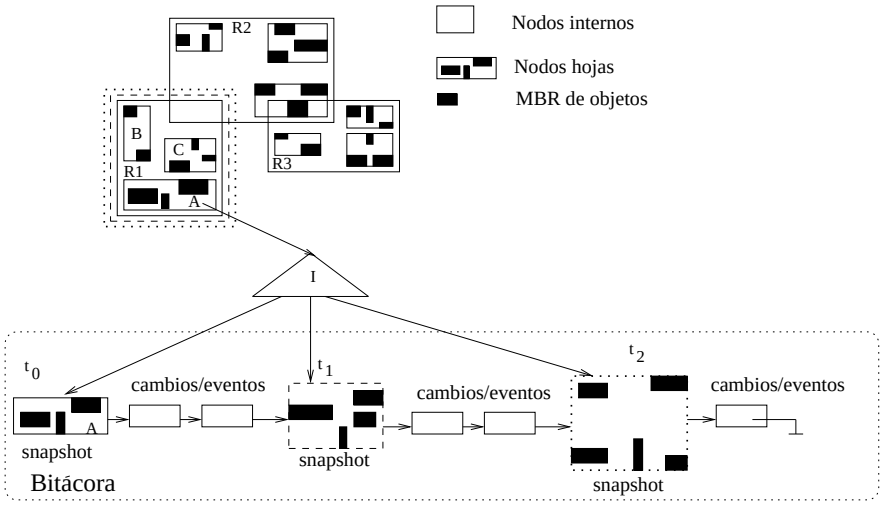


Figura 5.1: Esquema general del SEST$_L$.

La organización de este capítulo es la siguiente. En la Sección 5.2 se definen las estructuras de datos y los algoritmos utilizados por el SEST$_L$ para implementar las consultas espacio-temporales y las operaciones para mantener actualizadas las estructuras de datos con los eventos que suceden a lo largo del tiempo. La Sección 5.3 está destinada a describir la incorporación de snapshots globales al SEST$_L$ con el objeto de contrarestar el impacto del crecimiento de los MBRs en la eficiencia de las consultas. En la Sección 5.4 se presentan y discuten los resultados de una evaluación experimental en la que se compara el rendimiento del SEST$_L$ con el SEST-Index y el MVR-tree en diferentes escenarios. La Sección 5.5 está dedicada a definir y validar experimentalmente un modelo de costo para el SEST$_L$ que estima el almacenamiento ocupado por el índice y el costo de procesamiento de las consultas. Finalmente, en la Sección 5.6 se discuten las conclusiones de este capítulo.

5.2 Estructura de datos

Al igual que el SEST-Index, el SEST$_L$ también considera un R-tree con la misma estructura para las entradas de los nodos. Sin embargo, las bitácoras son diferentes ya que consideran dos tipos

de entradas: eventos y entradas de snapshot. El primer tipo de entradas (eventos) tiene la misma estructura y significado que las definidas en el SEST-Index. El segundo tipo de entradas (snapshot) almacena el snapshot de un nodo hoja con una entrada por cada objeto vigente o "vivo" en el instante en que el snapshot fue creado. Las entradas en la bitácora se almacenan ordenadas por el tiempo. Similarmente al SEST-Index, el SEST_L también considera un parámetro d, igual para todas las bitácoras en la estructura, que define el número máximo de eventos que es posible almacenar entre snapshots consecutivos.

5.2.1 Operaciones

5.2.1.1 Consultas de tipo *time-slice*

Para procesar una consulta de tipo *time-slice* (Q,t) el primer paso consiste en encontrar todas las hojas que intersectan el rectángulo Q. A continuación, para cada bitácora de cada hoja, se obtiene el snapshot adecuado de acuerdo al tiempo t. Este snapshot corresponde al creado en el último instante de tiempo tr tal que $tr < t$. Los objetos espaciales que están almacenados en el snapshot seleccionado y que se intersectan espacialmente con Q, forman el conjunto inicial de la respuesta. Este conjunto se actualiza con los eventos almacenados como entradas de tipo evento de la bitácora dentro del intervalo $(tr,t]$ (Alg. 5.1). El proceso anterior se repite para cada una de las bitácoras cuya subregión asignada se intersecta con Q. La respuesta de la consulta se consigue por medio de la unión de los conjuntos obtenidos a partir de cada una de las bitácoras.

```
 1: time-sliceQuery(Rectangle Q, Time t, R-tree R)
 2: S = SearchRtree(Q,R) {S es el conjunto de bitácoras que se intersectan espacialmente con Q}
 3: G = ∅ {G almacena el conjunto de objetos que pertenecen a la respuesta}
 4: for cada bitacora b ∈ S do
 5:    sea tr el instante del último instante de tiempo de b tal que tr ≤ t.
 6:    sea A el conjunto de todos los objetos vigentes en el snapshot creado en el instante tr en la bitácora b y que se
       intersectan espacialmente con Q
 7:    for cada entrada de tipo evento c ∈ b tal que tr < c.t ≤ t do
 8:       if c.Geom Intersect(Q) then
 9:          if c.Op = move_in then
10:             A = A ∪ {c.Oid}
11:          else
12:             A = A − {c.Oid}
13:          end if
14:       end if
15:    end for
16:    G = G ∪ A
17: end for
18: return G
```

Alg. 5.1: Algoritmo para procesar una consulta de tipo *time-slice*.

5.2.1.2 Consultas de tipo *time-interval*

Para procesar una consulta de tipo *time-interval* $(Q,[t_i,t_f])$ se requiere encontrar el conjunto de objetos que intersecta espacialmente a Q en el instante de tiempo t_i. Esto es equivalente a ejecutar una consulta *time-slice* en el instante t_i. Posteriormente, el conjunto de objetos se actualiza con los eventos ocurridos en el intervalo $(t_i,t_f]$ (Alg. 5.2).

```
1:  IntervalQuery(Rectangle Q, Time t_i,t_f, R-tree R)
2:  S = SearchRtree(Q,R) {S es el conjunto de bitácoras que intersecta espacialmente a Q}
3:  G = ∅ {G es el conjunto de objetos que pertenecen a la respuesta}
4:  for each bitacora b ∈ S do
5:     sea tr el instante de tiempo del último snapshot en b tal que tr ≤ t_i.
6:     sea A el conjunto de todos los objetos vigentes en el snapshot creado en el instante de tiempo tr en la bitácora b
       y que se intersectan espacialmente con Q
7:     actualizar A con los eventos almacenados en b ocurridos entre (tr,t_i]. {similar al procedimiento seguido para
       una consulta de tipo time-slice}
8:     ts = Next(t_i) {Next(x) obtiene el instante de tiempo que sigue a x de los eventos almacenados en b}
9:     while t_s ≤ t_f ∧ existan entradas de eventos en b do
10:       for cada entrada c ∈ b tal que t_s = c.t do
11:          if c.Geom intersect(Q) then
12:             if c.Op = move_in then
13:                A = A ∪ {c.Oid}
14:             else
15:                A = A − {c.Oid}
16:             end if
17:          end if
18:       end for
19:       G = G ∪ {⟨o,ts⟩, o ∈ A}
20:       ts = Next(ts)
21:    end while
22: end for
23: return G
```

Alg. 5.2: Algoritmo para procesar una consulta de tipo *time-interval*.

5.2.1.3 Consultas sobre los eventos

Procesar consultas sobre eventos con el SEST_L (ver Alg. 5.3) es muy simple y a la vez eficiente, pues la estructura almacena de manera explícita los eventos que ocurren sobre el atributo espacial de los objetos. Los algoritmos para este tipo de consultas son similares a aquellos definidos para consultas de tipo *time-slice* o *time-interval* utilizando el SEST_L.

5.2.1.4 Actualización de la estructura de datos

Esta operación permite actualizar la estructura con los eventos que ocurren en cada instante de tiempo. Se asume que los eventos a ser procesados se almacenan en una lista. El movimiento de un objeto genera dos eventos, *move_out* y *move_in*, los que se deben almacenar en las bitácoras. Un evento de tipo *move_in* incluye todos los valores de los atributos *t*, *Geom* y *Oid* para el objeto a insertar. Por cada evento de tipo *move_in*, se elige la correspondiente bitácora donde el objeto debería ser insertado de acuerdo a la clásica política de inserción de un R-tree [Gut84] (llamado *chooseleaf()* en el Alg. 5.4).

Un evento de tipo *move_out*, por el contrario, sólo contiene los valores de los atributos *t* y *Oid*. Esto hace más complicado el procesamiento de los eventos *move_out* ya que: (i) se debe recuperar el valor del atributo *Geom* a partir del valor del atributo *Oid* del evento, y almacenar este valor como una entrada de un evento de tipo *move_out* en una bitácora y (ii) se debe asegurar que los eventos se almacenen en la misma bitácora de su correspondiente evento *move_in*. Debido a que la forma del R-tree cambia a lo largo del tiempo del evento *move_in*, el procedimiento $chooseleaf()$ no garantiza encontrarlo en la misma bitácora nuevamente. Una solución para ambos problemas es

```
1:  EventQuery(Rectangle Q, Time t, R-tree R)
2:  S = SearchRtree(Q,R) {S es el conjunto de bitácoras que se intersectan espacialmente con Q}
3:  oi = 0 {cantidad de objetos que entran a una región Q en el instante de tiempo t}
4:  os = 0 {cantidad de objetos que salen de una región Q en el instante de tiempo t}
5:  for cada bitacora b ∈ S do
6:     sea tr el instante de tiempo de creación del último snapshot b tal que tr ≤ t.
7:     encontrar la primera entrada de c ∈ b tal que c.t = t.
8:     while c.t = t do
9:        if c.Geom intersect(Q) then
10:          if c.Op = move_in then
11:             oi = oi + 1
12:          else
13:             os = os + 1
14:          end if
15:       end if
16:       c = NextChange(c) { la función NextChange(x) obtiene la entrada de tipo evento siguiente a x en la bitácora
          b}
17:    end while
18: end for
19: return (oi, os)
```

Alg. 5.3: Algoritmo para procesar consultas sobre eventos.

mantener una tabla hashing ($\langle Oid, bid \rangle$) donde *bid* es una referencia a un bloque de bitácora donde se encuentra el objeto justo antes que ocurra el evento *move_out*. El procedimiento para encontrar la hoja se llama $choosepreviousleaf()$ en el Alg. 5.4.

Tanto los eventos *move_in* como *move_out* se insertan como una entrada de tipo evento después del último snapshot que fue almacenado en la correspondiente bitácora. Si en el instante de inserción el número de eventos excede el valor definido para el parámetro d, entonces se crea un nuevo snapshot. Notar que a pesar que la cantidad de eventos pueda sobrepasar la capacidad de la bitácora (parámetro d), todos los eventos ocurridos en el mismo instante de tiempo en la región asignada a la bitácora se almacenan como entradas de tipo evento entre dos snapshot consecutivos. La inserción de un evento *move_in* puede requerir actualizar los MBRs de las hojas, como también nodos ancestros cuyos MBRs deberían ahora incluir el atributo *Geom* del objeto.

5.3 El SEST$_L$ con snapshots globales

Un problema potencial del SEST$_L$ es que el rendimiento de las consultas se puede deteriorar debido al crecimiento de las áreas de las regiones, que es una consecuencia de las inserciones de los eventos en las bitácoras a través del tiempo. Recordemos que la inserción de un entrada de tipo *move_in* puede expandir las áreas de los MBRs de la raíz a las hojas (bitácoras) del árbol, mientras que un evento de tipo *move_out* no reduce las áreas de sus correspondientes MBRs. Como una consecuencia de este crecimiento de las áreas de los MBRs, el tiempo que le toma el procesamiento de una consulta espacio-temporal (Q,t) ejecutada en el instante de tiempo t_1 será menor que el tiempo que le toma a la misma consulta ejecutada en el instante t_2, con $t_2 > t_1$.

Para medir este efecto, se definió una *densidad* asociada con el R-tree. Esta medida, *denReal*, está definida por la Ec. (5.1), donde M es el conjunto de MBRs ubicados al nivel de las hojas del R-tree, y *TotalArea* es el área del mínimo rectángulo que contiene a todos los objetos. En otras

```
1:  InsertChanges(Changes C, R-tree R, Integer d) {C es la lista de eventos ocurridos de manera creciente en el
    tiempo y d es la capacidad de la bitácora para almacenar entradas de tipo evento entre dos snapshots consecutivos}
2:  for cada c ∈ C do
3:     if c.Op = move_in then
4:        b = chooseleaf(R, c.Geom)
5:     else
6:        b = choosepreviousleaf(R, c.Oid).
7:     end if
8:     sea L la lista de eventos ocurridos en b después del último snapshot
9:     sea l el total de eventos almacenados en L
10:    if l > d ∧ c.t ≠ L[l].t then
11:       crear un nuevo snapshot S en la bitácora b
12:       crear una nueva lista vacía L y asignarla a S
13:    end if
14:    insertar c al final de L
15:    if c.Op = move_in then
16:       actualizar todos los MBRs que fueron seguidos por el procedimiento chooseleaf() para alcanzar b.
17:    end if
18: end for
```

Alg. 5.4: Algoritmo para actualizar la estructura del SEST_L.

palabras, *denReal* describe cuán compactos se encuentran los MBRs de las hojas, de esta forma un valor bajo de *denReal* implica que los objetos de la base de datos se encuentran cubiertos por pequeños MBRs. El valor de *denReal* afecta el rendimiento de las consultas, ya que entre mayores son los MBRs, mayor será la cantidad de bitácoras que deben ser procesadas por una consulta. El valor de *denReal* normalmente crece con las inserciones de los eventos de tipo *move_in*, debido a que los MBRs crecen y el espacio que contiene a todos los objetos tiende a permanecer constante. Esto explica por qué una consulta ejecutada en el instante t_1 cuesta menos que ejecutarla en el instante t_2.

$$denReal = \frac{\sum_{i \in M} Area_i}{TotalArea} \tag{5.1}$$

Una solución a este problema es definir snapshots globales sobre el SEST_L. Un snapshot global es un R-tree construido con todos los objetos vigentes en un instante particular. Tal R-tree permite redefinir las regiones asociadas con las bitácoras y, por lo tanto, eventualmente, disminuir el valor de la densidad (*denReal*) (Ec. (5.1)).

Se usó el valor de la densidad *denReal* para determinar el instante de tiempo en que se debe crear un nuevo snapshot global. El proceso para crear un snapshot global se describe en el Alg. 5.5. Distinto al algoritmo *InsertChanges()*, *UpdateSEST$_L$()* requiere procesar de manera conjunta todos los eventos producidos en un mismo instante de tiempo. Esto asegura que todos estos eventos queden almacenados dentro del ámbito de influencia de un mismo snapshot global. Sin embargo, es fácil modificar el algoritmo para manejar conjuntos con eventos pertenecientes a intervalos consecutivos diferentes y conservar la propiedad de que los eventos producidos en un mismo instante de tiempo queden contenidos en el ámbito de un snapshot global. El algoritmo tiene en cuenta dos R-tree's. El primero es aquel establecido en el último snapshot creado (en Alg. 5.5 corresponde a $E.R$) y el segundo corresponde a uno que mantiene las posiciones de cada objeto en el último instante de tiempo (en Alg. 5.5 se denomina *nuevoRtree*). Los eventos que se van

produciendo en cada momento actualizan a ambos R-tree's pero de manera diferente. El primer R-tree se actualiza tal como se explica en el algoritmo *InsertChanges()* y el segundo utilizando los algoritmos típicos de inserción y eliminación definidos en el R-tree original.

El Alg. 5.5 verifica que el valor de *denReal*, después de insertar los nuevos eventos, no exceda el valor de umbral $ls \cdot ultimaDensidad$, donde *ultimaDensidad* es la densidad del último snapshot global y $ls \geq 1$ es un parámetro de la estructura. Cuando el valor de *denReal* sobrepasa al del umbral, entonces se verifica si es conveniente crear un nuevo snapshot global. Sea *nuevaDensidad* la densidad del R-tree *nuevoRtree*. El algoritmo se asegura de que $nuevaDensidad < li \cdot denReal$ antes de que se produzca la creación definitiva del nuevo snapshot global. Si la condición anterior no se cumple, entonces el snapshot global sencillamente no se crea, pues el almacenamiento invertido en crearlo no mejorará la eficiencia de las consultas. En este algoritmo $li \leq 1$ es un parámetro del procedimiento.

1: **UpdateSEST$_L$(entradaSnapshotGlobal** E**, float** *ultimaDensidad*, **Eventos** C, **float** ls, **float** li, **Integer** d, **R-tree** *nuevoRtree*) { $E(\langle$ R-tree R, Tiempo t_s, float *ultimaDensidad*$\rangle)$ es la entrada con el último snapshot global (creado en t_s), $E.ulimaDensidad$ corresponde al valor de *denReal* para $E.R$ al momento de crear E, C es una lista con los eventos a insertar en el SEST$_L$ que ocurrieron en el último instante de tiempo t, ls y li son fracciones de *ultimaDensidad* }
2: *InsertChanges*(C,R,d) {Inserta los eventos en el SEST$_L$ usando el Alg. 5.4}
3: *ActualizarNuevoRtree*$(C,$ *nuevoRtree*$)$ {Actualiza *nuevoRtree* con los evntos}
4: Sea *nuevaDensidad* el nuevo valor de *denReal* para $E.R$ después de insertar los eventos
5: **if** $nuevaDensidad > ls \cdot E.ultimaDensidad$ **then**
6: Sea $tmpDensidad$ el valor de *denReal* de *nuevoRtree*
7: **if** $tmpDensidad < li \cdot nuevaDensidad$ **then**
8: entradaSnapshotGlobal $E1 = \langle nuevoRtree, t, tmpDensidad \rangle$
9: *nuevoRtree* = Duplicar(*nuevoRtree*)
10: Agregar $E1$ a la secuencia de snapshots globales
11: **end if**
12: **end if**

Alg. 5.5: Algoritmo que decide cuando crear un nuevo snapshot global.

Los algoritmos para procesar las consultas de tipo *time-slice* y *time-interval* con el SEST$_L$ con snapshots globales no diefieren mucho de los definidos para las mismas consultas utilizando el SEST$_L$ original. Para una consulta de tipo *time-slice* (Q,t) sólo se necesita localizar el snapshot global apropiado utilizando el parámetro t y, luego, se sigue el mismo procedimiento definido en el Alg. 5.1. Para procesar una consulta de tipo *time-interval* $(Q,[t_i,t_f])$ se usa el procedimiento descrito en el Alg. 5.2, pero primero es necesario transformar la consulta $(Q,[t_i,t_f])$ en un conjunto de subconsultas de tipo *time-interval* (Q,T_j), cuyos intervalos de tiempo T_j forman una partición del intervalo temporal $[t_i,t_j]$. Los límites de un intervalo T_j se definen de tal manera que la subconsulta cubra el intervalo de tiempo entre snapshots globales consecutivos, con la excepción de que el instante inicial del primer intervalo debe ser t_i, y el instante final del último intervalo debe ser t_f. La respuesta de una consulta $(Q,[t_i,t_f])$ se obtiene como la unión de las respuestas de cada subconsulta. Notar que las subconsultas pueden ser procesadas de manera independiente.

5.4 Evaluación Experimental

5.4.1 Comparación de SEST$_L$ con SEST-Index

Se hizo una primera evaluación experimental en la que se comparó el SEST$_L$ (sin snapshots globales) contra el SEST-Index. Se compararon ambos métodos en términos de almacenamiento y tiempo necesario para procesar las consultas con 23.268 objetos (puntos) distribuidos de manera uniforme dentro del espacio en el instante 0 cuyas posiciones cambian a lo largo de 200 instantes de tiempo también de manera uniforme. Los conjuntos fueron obtenidos con el generador de objetos espacio-temporales GSTD [TSN99].

Una primera evaluación consistió en medir la eficiencia de las consultas de tipo *time-interval* al almacenar la misma cantidad de eventos por bitácora. Las diferentes capacidades de las bitácoras se definieron de tal manera que almacenen todos los eventos producidos en $1, 2, \ldots, k$ instantes de tiempo, con $k = 20$. De esta forma el tamaño de la bitácora para SEST-Index se obtuvo con la ecuación (cantidad de bloques) $d = \frac{p \cdot N \cdot k}{c}$ y la de SEST$_L$ con la ecuación $d = \frac{p \cdot f \cdot k}{c}$. Las Figuras 5.2 y 5.3 muestran el rendimiento de ambas estructuras en las cuales se puede apreciar que SEST$_L$ supera largamente a SEST-Index tanto en espacio como en eficiencia para evaluar las consultas. Por ejemplo, en la Figura 5.3 podemos ver que si fijamos el espacio a 40 megabytes, SEST$_L$ deberá accesar alrededor de 300 bloques. En cambio SEST-Index requiere accesar más de 3.000 bloques para la misma consulta.

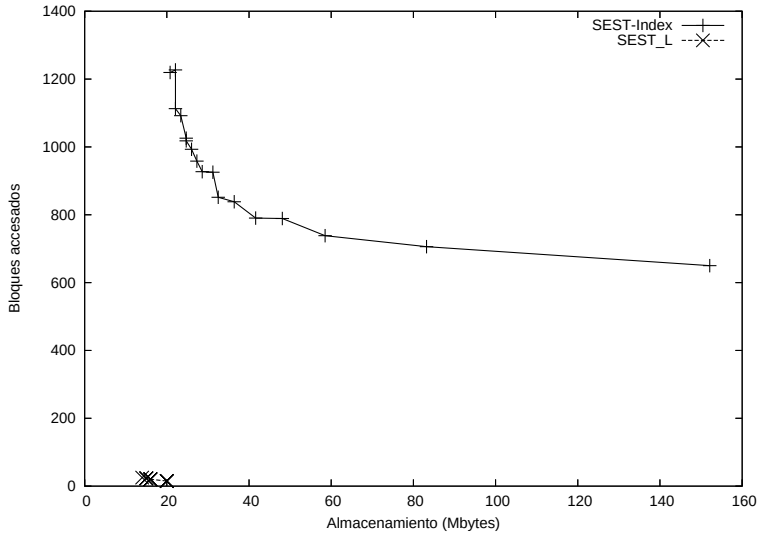


Figura 5.2: Rendimiento del SEST-Index versus el SEST$_L$ (movilidad = 10%, longitud intervalo temporal = 10 y área formada por el 6% de cada dimensión).

El rendimiento desfavorable del SEST-Index, con respecto al SEST$_L$, se explica por dos factores:

a) El SEST-Index duplica en cada snapshot todos los objetos, incluyendo aquellos que no sufren modificaciones en sus atributos espaciales. El problema de la duplicación, en principio, podría ser resuelto utilizando la estrategia de sobreposición utilizada por el HR-tree, pero los resultados experimentales (ver Sección 4.3.1) muestran que el ahorro en

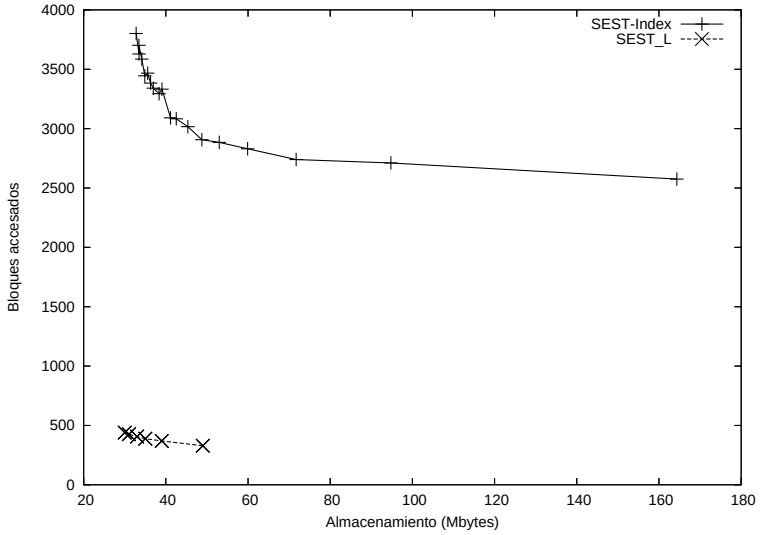


Figura 5.3: Rendimiento del SEST-Index versus el SEST_L (movilidad = 20%, longitud intervalo temporal = 20 y área formada por el 10% de cada dimensión).

espacio es bajo, alrededor de un 7% – 8%. Por el contrario, el SEST_L construye un nuevo snapshot sólo para la hojas que han sufrido cambios.

b) El SEST-Index agrupa los eventos sólo por tiempo y no por tiempo y espacio. Todos los eventos que ocurren entre snapshot consecutivos se almacenan en una sola bitácora, los cuales pueden requerir de varios bloques de disco. En una consulta (Q,t), Q se usa sólo para obtener el conjunto inicial de los objetos y no filtra los bloques de bitácora durante el procesamiento. De esta manera, todos los eventos entre el tiempo de creación del snapshot y el tiempo dado en la consulta t se deben procesar. Este problema se puede aminorar disminuyendo el valor del parámetro d, pero lamentablemente esto agudiza el problema a).

Posteriormente se comparó el rendimiento de ambas estructuras para evaluar consultas de tipo *time-slice* y sobre eventos. Para evaluar los dos tipos de consultas se consideró una movilidad de un 10% de los objetos y que ambas estructuras usan aproximadamente la misma cantidad de almacenamiento para almacenar los objetos y sus eventos. Para cumplir con esta última restricción se estaleció $d = 816$ bloques para el SEST-Index y $d = 4$ para el SEST_L. En la Figura 5.4 se puede observar que SEST-Index supera a SEST_L para consultas de tipo *time-slice* cuando el rango espacial está formado por un porcentaje superior a 60% en cada dimensión. El rendimiento favorable del SEST-Index, con respecto a SEST_L, se explica por el hecho de que todos los eventos producidos en un mismo instante de tiempo se encuentran en bloques contiguos en la bitácora del SEST-Index. En cambio en el SEST_L se encuentran repartidos en varias bitácoras, y en la medida que el rango espacial aumenta, es necesario procesar una mayor cantidad de bitácoras para obtener la respuesta.

La Figura 5.5 muestra el rendimiento de ambas estructuras para procesar consultas orientadas a eventos. Podemos observar que el SEST-Index supera a SEST_L para este tipo de consultas cuando el rango espacial está formado por un porcentaje de cada dimensión mayor o igual a 11%

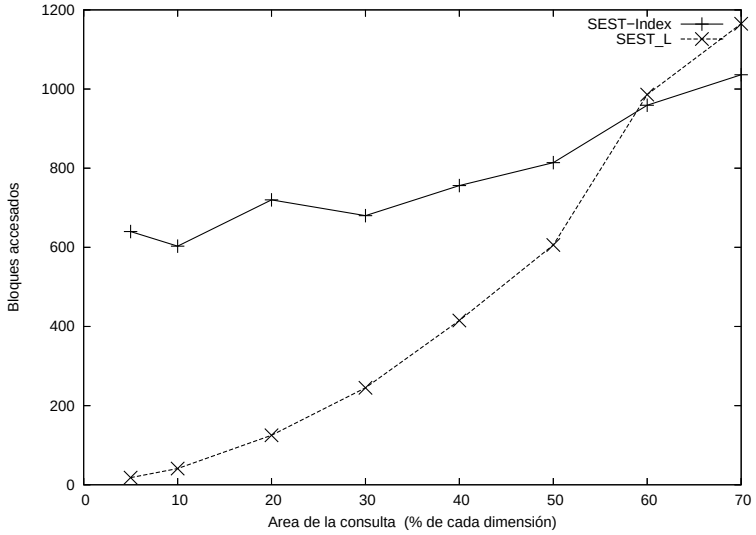


Figura 5.4: Rendimiento del SEST-Index versus el SEST_L (consultas *time-slice*, ambas estructuras usan la misma cantidad de almacenamiento).

aproximadamente. La cantidad constante de bloques accesados por SEST-Index se explica por el hecho de que es posible acceder de manera directa al primer bloque de la bitácora donde se comienzan a almacenar los eventos de cada instante de tiempo (ver Sección 4.2.1.3). Notar que la cantidad de bloques es de 50 y que corresponde aproximadamente a la cantidad de bloques requeridos para almacenar los 2.327 cambios producidos por los 23.268 objetos en cada instante de tiempo.

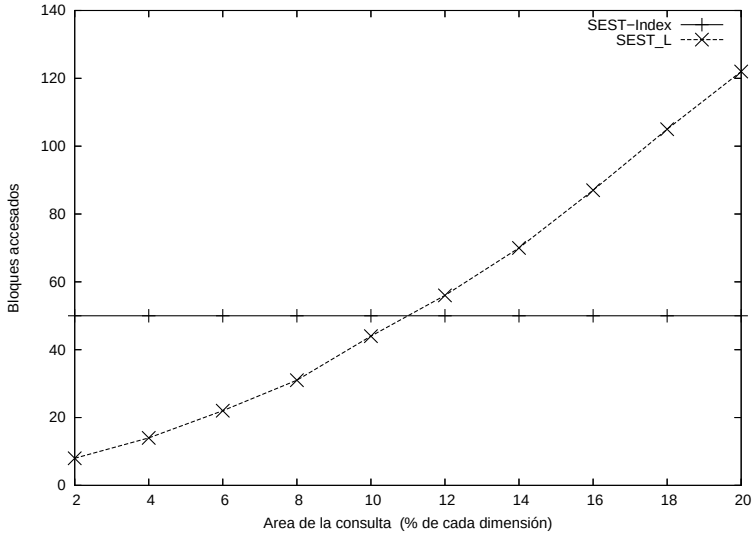


Figura 5.5: Rendimiento del SEST-Index versus el SEST$_L$ (consultas sobre eventos).

5.4.2 Ajustes de la estructura de datos de los eventos

El SEST$_L$ es un método de acceso orientado a eventos que puede procesar de manera eficiente consultas de tipo *time-slice* y *time-interval* y, además, consultas sobre los eventos. Para garantizar un rendimiento adecuado de todos los tipos de consultas, la estructura mantiene algunos atributos que pueden ser eliminados si solamente interesan las consultas de tipo *time-slice* y *time-interval*. Específicamente es posible eliminar el atributo *Geom* en los eventos *move_out* si solamente interesan estos dos últimos tipos de consultas.

5.4.3 Comparación de SEST$_L$ con MVR-tree

Una segunda evaluación experimental consistió en comparar el SEST$_L$ (sin snapshots globales) con el MVR-tree en diferentes escenarios. El MVR-tree se considera el mejor método de acceso espacio-temporal para responder consultas de tipo *time-slice* y de tipo *time-interval* [TP01b, TPZ02, TP01a], superando al HR-tree [NST99]. Se compararon ambos métodos en términos de almacenamiento y tiempo necesario para procesar las consultas con 23.268 objetos (puntos) distribuidos de manera uniforme dentro del espacio en el instante 0 movidos a lo largo de 200 instantes de tiempo también de manera uniforme. Se estudiaron 6 valores para el porcentaje de movilidad (frecuencia de cambios) que fueron 1%, 5%, 10%, 15%, 20% y 25%. Para el caso del parámetro d se consideraron los valores de 2, 4 y 8 bloques de disco con un tamaño de bloque igual a 1.024 bytes. El almacenamiento se midió por el número de bloques o megabytes (Mbytes) utilizados después de realizadas todas las inserciones de los objetos y sus eventos. La eficiencia de las consultas se midió como el promedio de bloques accesados o leídos al ejecutar 100 consultas al azar del mismo tipo. Los conjuntos fueron obtenidos con el generador de objetos espacio-temporales GSTD [TSN99] siguiendo una distribución uniforme. En los experimentos se utlizaron las implementaciones de ambas estructuras con las mismas entradas y parámetros.

5.4.3.1 Almacenamiento utilizado

La Figura 5.6 muestra que el SEST$_L$ necesita de menos espacio que el MVR-tree lo que se ve más claramente en la medida que el porcentaje de movilidad aumenta. En este escenario, el SEST$_L$ (estructura de datos original) requiere sólo entre 50% y 65% del almacenamiento ocupado por el MVR-tree. Se puede observar que en la medida que el valor del parámetro d aumenta, el SEST$_L$ necesita de menos almacenamiento que el MVR-tree, lo que se explica por la menor cantidad de snapshots que se deben almacenar. También podemos ver que la diferencia favorable del SEST$_L$ sobre el MVR-tree aumenta de manera importante cuando se considera la estructura de datos ajustada.

5.4.3.2 Rendimiento de las consultas *time-slice* y *time-interval*

Las Figuras 5.7 y 5.8 muestran el rendimiento de las consultas espacio-temporales utilizando el SEST$_L$ y el MVR-tree. Estas consultas consideraron diferentes longitudes del intervalo temporal y un rango espacial formado por el 6% de cada dimensión y se ejecutaron sobre conjuntos con movilidades de 5% y 10%. En las Figuras 5.7 y 5.8 podemos observar que en la medida que la longitud del intervalo temporal aumenta SEST$_L$ supera mas ampliamente a MVR-tree. También podemos observar que la curva de SEST$_L$ para $d = 8$ corta a la de MVR-tree cuando la longitud

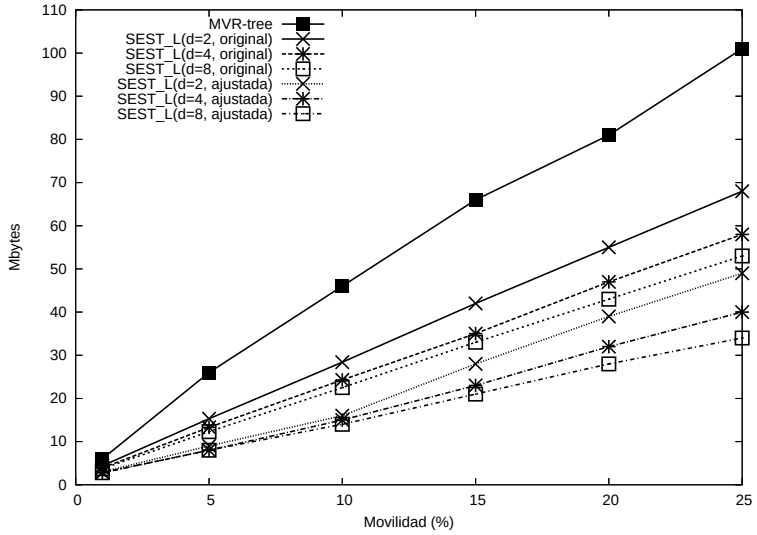


Figura 5.6: Almacenamiento utilizado por el $SEST_L$ (estructura de datos original y ajustada) y el MVR-tree.

del intervalo temporal es aproximadamente 3. Cabe hacer notar que para este valor de d ($d = 8$) es cuando $SEST_L$ requiere de la menor cantidad de almacenamiento (ver Figura 5.6).

Al igual que para el almacenamiento, las ventajas del $SEST_L$ sobre MVR-tree se destacan aún más cuando se considera la estructura de datos ajustada. Por ejemplo, en la Figura 5.7 es posible apreciar que $SEST_L$ supera a MVR-tree para todas las longitudes del intervalo temporal estudiadas. Para el resto de las evaluaciones sobre el $SEST_L$, contempladas en este tesis (excepto las mediciones de los experimentos realizados en el Capítulo 8), sólo se muestran las que consideran la estructura de datos original.

Las Figuras 5.9 y 5.10 muestran el comportamiento del MVR-tree y el $SEST_L$ para consultas de tipo *time-interval* fijando la longitud del intervalo temporal en 1 y 4 y rango espacial de las consultas formados por 2% – 20% de cada dimensión. En la Figura 5.9 es posible observar que el MVR-tree supera al $SEST_L$ en la medida que el área de la consulta aumenta. Por ejemplo, en la Figura 5.9 el MVR-tree empieza a superar al $SEST_L$ a partir de un área formada por el 4% de cada dimensión para $d = 8$ y a partir de 8% y 14% para $d = 4$ y $d = 2$, respectivamente. También es posible notar que el área de las consultas donde el MVR-tree supera al $SEST_L$, aumenta en la medida que crece la longitud del intervalo temporal de las consultas. Esto último se puede ver en la Figura 5.10 donde la curva del MVR-tree corta a las del $SEST_L$ para $d = 8$ y d $= 4$ en consultas cuya área está formada por 8% y 16%, respectivamente. Notar que el $SEST_L$ siempre supera al MVR-tree para las diferentes áreas de las consultas al considerar $d = 2$ y un largo de intervalo igual a 4. Cabe destacar que el $SEST_L$, para $d = 2$, requiere de menos espacio que el MVR-tree para las movilidades de 5% y 10%.

Las Figuras 5.11 y 5.12 muestran las zonas donde cada uno de los métodos de acceso domina y donde se puede observar que el ámbito de dominio del $SEST_L$ es superior al del MVR-tree. En ambas figuras es posible observar que el MVR-tree supera al $SEST_L$ cuando el área de las consultas está formada por un porcentaje de cada dimensión mayor a 12% y el intervalo temporal de la misma

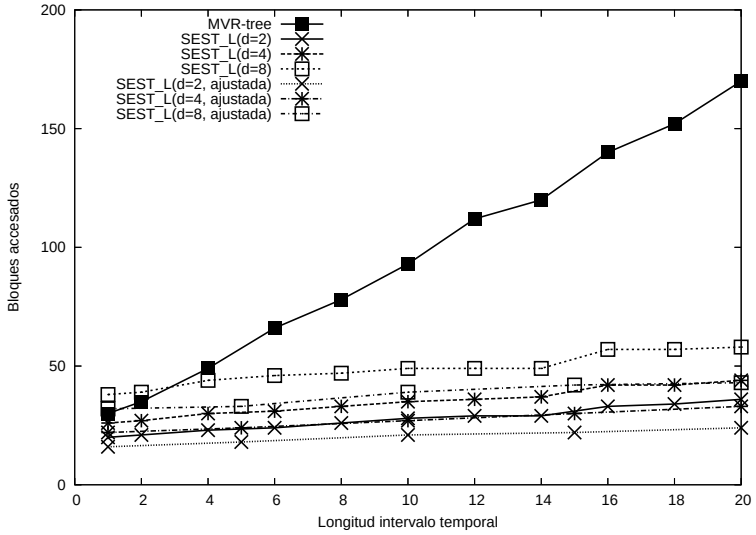


Figura 5.7: Bloques accesados por el SEST_L (estructura de datos original y ajustada) y el MVR-tree para consultas con diferentes longitudes del intervalo temporal (10% de movilidad y rango espacial formado por el 6% de la longitud de cada dimensión).

es inferior a 6. Sin embargo, el SEST_L supera al MVR-tree en la medida que el largo del intervalo temporal aumenta. Específicamente los datos de nuestros experimentos muestran que para el caso de una movilidad de 5% (Figura 5.11), el SEST_L supera al MVR-tree para consultas cuya longitud es superior a 16 unidades de tiempo y cuyas áreas están formadas por cualquiera de los porcentajes de cada dimensión estudiados (2%–20%). La amplitud del intervalo temporal disminuye a 12 cuando se consideró un 10% de movilidad (Figura 5.12). La razón por la cual el rendimiento del SEST_L mejora, con respecto al MVR-tree, en la medida que aumenta el intervalo temporal se debe a que el SEST_L paga un costo inicial importante para recuperar los objetos hasta el instante inicial del intervalo temporal, el cual incluye recorrer el R-tree y procesar los eventos hasta alcanzar el instante inicial del intervalo temporal. De ahí en adelante, el costo total de la consulta es bajo, ya que sólo se requiere procesar los eventos hasta alcanzar el instante final del intervalo temporal de la consulta. En el caso del MVR-tree, la cantidad de R-trees que se necesitan revisar aumenta con la longitud del intervalo temporal de la consulta. Esto explica por qué el MVR-tree es mejor que el SEST_L en intervalos cortos mientras que el SEST_L lo supera en intervalos largos.

Existe un método de acceso espacio-temporal conocido como MV3R-tree (comentado en el Capítulo 3) y que corresponde a una variante del MVR-tree. El MV3R-tree combina un MVR-tree y un pequeño 3D R-tree[1][TP01b]. Las consultas con intervalos temporales cortos se resuelven utilizando el MVR-tree y las consultas con intervalos temporales largos se procesan con el 3D R-tree. En [TP01b] se compara el MV3R-tree con el HR-tree [NST98, NST99] y el 3D R-tree original[2][TVS96]. Por una parte, los resultados muestran que el 3D R-tree usado por el MV3R-tree emplea un 3% del almacenamiento ocupado por el MVR-tree (así el MV3R-tree es sólo un

[1]Este 3D R-tree se construye considerando las hojas del MVR-tree. Esto es, cada hoja del MVR-tree es un objeto en el 3D R-tree

[2]Este 3D R-tree, en cambio, almacena cada objeto, en lugar de cada hoja del MVR-tree

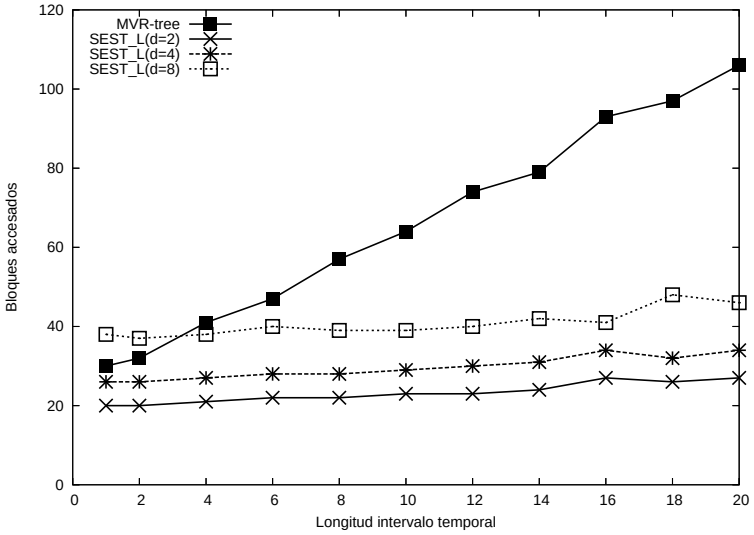


Figura 5.8: Bloques accesados por el $SEST_L$ (estructura de datos original) y el MVR-tree para consultas con diferentes longitudes del intervalo temporal (5% de movilidad y rango espacial formado por el 6% de la longitud de cada dimensión).

3% mayor que el MVR-tree); el MV3R-tree utiliza 1.5 veces el almacenamiento requerido por el 3D R-tree original; y el MV3R-tree necesita una mínima fracción del almacenamiento del HR-tree. Por otra parte, nuestros experimentos muestran que el $SEST_L$ requiere, para $d = 4$, alrededor de 55% del almacenamiento utilizado por el MVR-tree. De esta forma es posible estimar que el $SEST_L$ necesita de alrededor del 85% del almacenamiento utilizado por el 3D R-tree original.

Con respecto a las consultas con intervalos temporales cortos (menores que el 5% del total del intervalo temporal de la base de datos), el MV3R-tree utiliza el MVR-tree para evaluarlas. En nuestros experimentos consideramos 200 snapshots y por lo tanto $SEST_L$ superará al MV3R-tree en los mismos escenarios donde se consideran intervalos temporales menor o igual a 10 unidades de tiempo. Para consultas que consideran un intervalo temporal superior al 5% el MV3R-tree realiza aproximadamente entre un 75% y 80% de los accesos requeridos por el MVR-tree según se muestra en [TP00]. Asumiendo que el costo (tiempo) del MV3R-tree será siempre de 75% del MVR-tree, es posible afirmar que el $SEST_L$ es de todas maneras mejor que el MV3R-tree para consultas con intervalos temporales mayores que 2 unidades de tiempo (para $d = 4$ y una movilidad promedio de un 10%). Las diferencias favorables al $SEST_L$, tanto en espacio como en tiempo, son aún más evidentes al realizar la comparación considerando la estructura ajustada del $SEST_L$ descrita en la Sección 5.4.2.

También es posible concluir que el $SEST_L$ es mejor que el método de acceso espacio-temporal que use el 3D R-tree original, ya que los resultados en [TP01b] muestran que el 3D R-tree no supera al MV3R-tree en ninguno de los escenarios analizados (en consultas con intervalos temporales largos el 3D R-tree experimentó un rendimiento muy similar al del MV3R-tree).

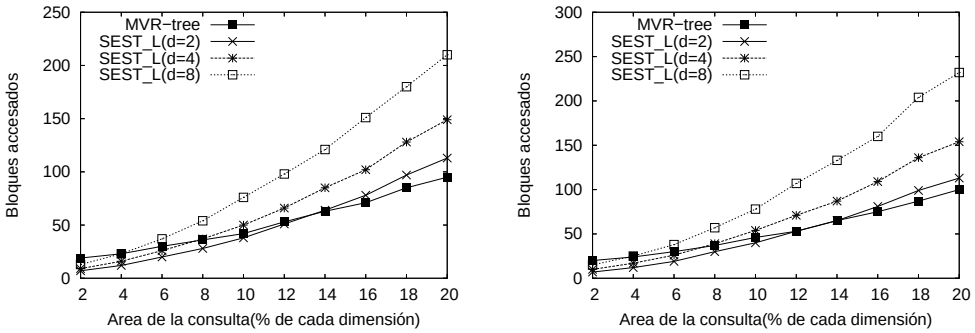


Figura 5.9: Bloques leídos por consultas *time-slice* (1 unidad del intervalo temporal) para diferentes rangos espaciales.

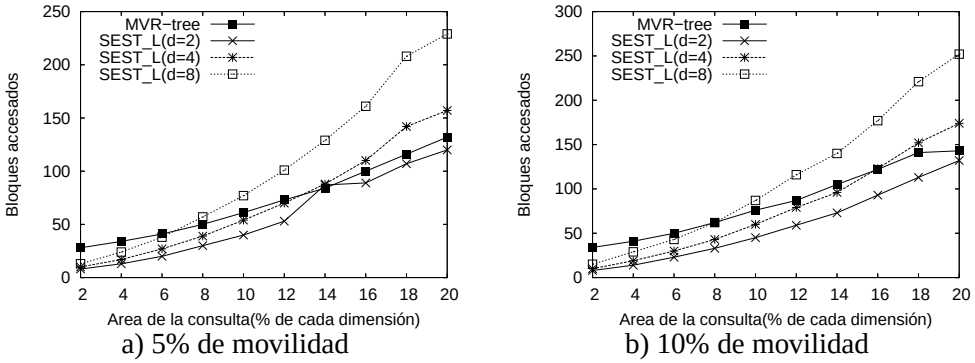


Figura 5.10: Bloques leídos por consultas para diferentes rangos espaciales (4 unidades del intervalo temporal).

5.4.3.3 Consultas sobre eventos

Tal como explicamos anteriormente, el $SEST_L$ también permite procesar consultas sobre los eventos que ocurrieron en un instante o periodo de tiempo. El costo de procesamiento de este tipo de consultas es el mismo que se necesita para las consultas de tipo *time-slice* y *time-interval*. Por otro lado, como el MVR-tree es un método orientado a resolver consultas de tipo *time-slice* y *time-interval*, éste no provee un algoritmo para responder consultas sobre los eventos. Sin embargo, es posible procesar consultas sobre eventos utilizando el MVR-tree. Por ejemplo, supongamos una consulta para recuperar cuántos objetos entraron a una área determinada (Q), en un instante de tiempo t. Para procesar esta consulta con MVR-tree se necesitan realizar dos consultas de tipo *time-slice*: una en el instante t y la otra en el instante inmediatamente anterior a t (t'). Los objetos que entraron a Q en t corresponde a aquellos que no se encontraban en Q en t' pero si se encuentran en Q en t. El costo de procesar esta consulta con MVR-tree tiene un costo igual doble del costo de procesar una consulta de tipo *time-slice* usando la misma estructura. De esta forma, para una

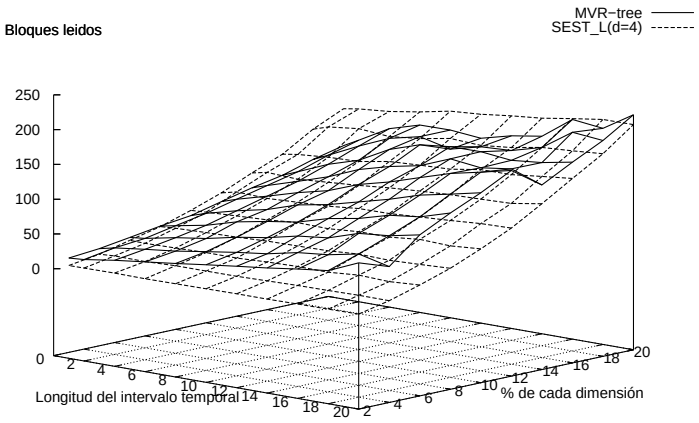


Figura 5.11: Bloques leídos por el SEST$_L$ y el MVR-tree (5% de movilidad).

consulta sobre eventos como la anterior considerando el área de Q formada por un 6% de cada dimensión con un 10% de movilidad y para un valor de $d = 4$, se requieren de 60 accesos con MVR-tree en lugar de 26 necesitados con el SETS$_L$, es decir, el SEST$_L$ requiere de menos de la mitad de los accesos realizados por el MVR-tree.

5.4.3.4 Evaluación del SEST$_L$ con snapshots globales

En esta sección se evalua el rendimiento del SEST$_L$ considerando snapshots globales. Los experimentos se llevaron a cabo utilizando el conjunto de datos usados en [TPZ02]. Dicho conjunto considera 23.268 puntos con una distribución inicial no uniforme (Figura 5.13 (a)) y una movilidad de un 10%. Los objetos se "mueven" a lo largo de 199 instantes de tiempo para alcanzar la distribución que se aprecia en la Figura 5.13. Para estos experimentos consideramos 4 bloques de disco como valor para el parámetro d.

La Figura 5.14 muestra los valores de la densidad *denReal* en cinco escenarios diferentes: (1) densidad del SEST$_L$ con 23.268 objetos (puntos), distribución uniforme inicial y 10% de movilidad, (2) la densidad del SEST$_L$ con el conjunto de objetos *NUD* sin snapshots globales; (3), (4) y (5) densidades del SEST$_L$ con los objetos del conjunto *NUD*, $li = 1$, y los valores de $ls = 1,3$, $ls = 1,6$ y $ls = 1,8$, respectivamente.

El almacenamiento ocupado por el SEST$_L$ para los valores $ls = 1,3$, $ls = 1,6$ y $ls = 1,8$ fue 33 Mbytes, 29 Mbytes y 28 Mbytes, respectivamente, contra 24 Mbytes del SEST$_L$ sin snapshots globales. La Figura 5.14 indica que cuando se considera $ls = 1,6$, se crean dos snapshots globales (aproximadamente en los instantes de tiempo 10 y 50). Esto provoca una mejora importante en la densidad, alcanzando valores muy similares a los del escenario (1). Utilizando valores de $ls = 1,3$ se pueden obtener resultados muy similares a los anteriores. Sin embargo, con un valor de $ls = 1,8$ (escenario (5)) la densidad es mayor que la del escenario (1), afectando negativamente la eficiencia de SEST$_L$ para procesar las consultas.

La Figura 5.15 muestra el rendimiento de las consultas con el SEST$_L$ original y el SEST$_L$ con

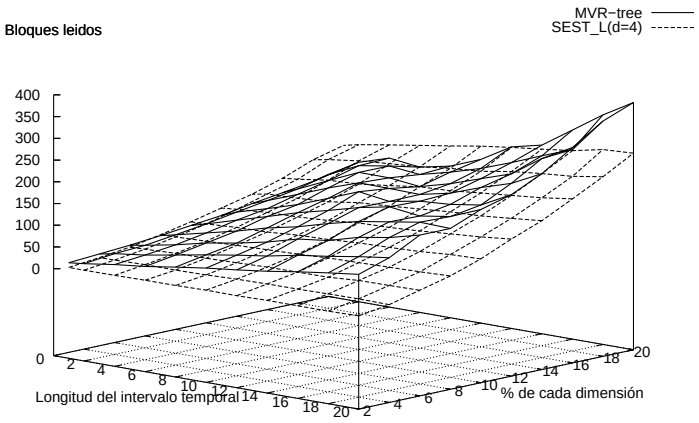
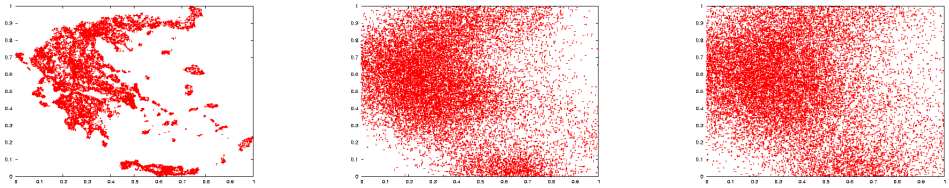


Figura 5.12: Bloques leídos por el SEST$_L$ y el MVR-tree (10% de movilidad).



(a) (b) (c)

Figura 5.13: Evolución de un conjunto de objetos: (a) instante 1, (b) instante 100 and (c) instante 200.

snapshots globales en los escenarios (2), (3), (4) y (5). En tal figura es posible apreciar que con los valores de $ls=1,3$ ó $ls=1,6$, el SEST$_L$ con snapshot globales presenta un rendimiento muy similar al SEST$_L$ original considerando objetos (23.268 puntos) que presentan una distribución uniforme.

5.5 Modelo de costo para el SEST$_L$

Esta sección describe un modelo de costo para el SEST$_L$. El modelo de costo fue evaluado experimentalmente para demostrar su capacidad de predicción.

El modelo de costo asume que las posiciones iniciales y subsiguientes de los objetos se distribuyen uniformemente y que los eventos consisten de cambios sobre la posición de los objetos producidos al azar. También supone que el número de objetos permanece constante a lo largo del tiempo. Las variables utilizadas en la definición del modelo de costo se encuentran descritas en la Tabla 4.2.

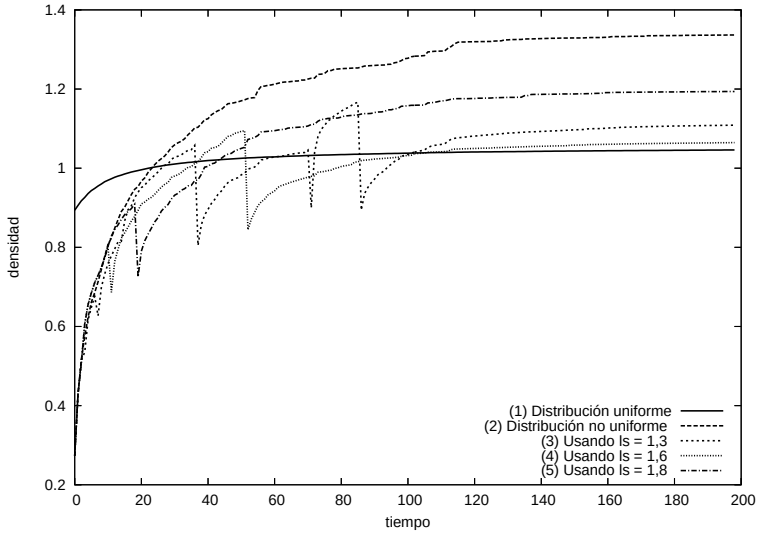


Figura 5.14: Densidad de los MBRs de las bitácoras del SEST$_L$.

5.5.1 Costo de almacenamiento del SEST$_L$

El almacenamiento del SEST$_L$ está principalmente determinado por el número de bitácoras y por el tamaño promedio de cada bitácora. El número de bitácoras es igual al número de hojas del R-tree y se obtiene con la ecuación $N_1 = NB = \left\lceil \frac{N}{f} \right\rceil$. El número de instantes de tiempo (il) que se pueden almacenar entre snapshot consecutivos se determina por la Ec. (5.2).

$$il = \left\lceil \frac{l}{p \cdot f} \right\rceil \tag{5.2}$$

Utilizando la Ec. (5.2) es posible obtener el valor de nl (Ec. (5.3)) de tal forma que todos los cambios ocurridos en el mismo instante de tiempo queden almacenados en la bitácora entre dos snapshots consecutivos.

$$nl = p \cdot f \cdot il \tag{5.3}$$

Con il y nl es posible obtener con la Ec. (5.4) el número de bloques de disco que se necesitan para almacenar los cambios que se producen en una bitácora. En esta ecuación, el primer término de la suma corresponde al total de bloques destinados a almacenar los snapshots en cada bitácora asumiendo que, en promedio, se requiere de un bloque para almacenar los objetos vigentes en cada snapshot. El segundo término representa el número de bloques que se necesitan, en promedio, para almacenar todos los cambios que se producen en el ámbito de una bitácora y que ocurren durante todos los instantes de tiempo almacenados en la base de datos. El último término se usa para obtener el número de bloques para almacenar los cambios ocurridos después del último snapshot de la bitácora.

$$TB_{bitacora} = \left\lceil \frac{nt}{il} \right\rceil + \left(\left\lceil \frac{nt}{il} \right\rceil - 1 \right) \cdot \left\lceil \frac{nl}{c} \right\rceil + \left\lceil \left(\frac{nt}{il} - \left\lfloor \frac{nt}{il} \right\rfloor \right) \cdot \frac{nl}{c} \right\rceil \tag{5.4}$$

Finalmente, la cantidad de bloques utilizados por el SEST$_L$ está dada por la Ec. (5.5)

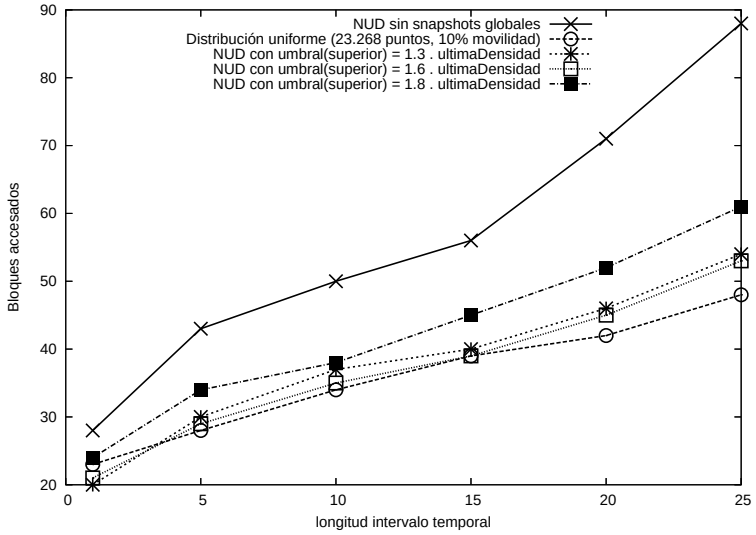


Figura 5.15: Rendimiento de las consultas con el SEST_L utilizando los objetos del conjunto NUD.

$$TB^{SEST_L} \;=\; (TN - NB) + NB \cdot TB_{bitacora} \;\approx\; \log_f N + N \cdot nt \cdot p \left(\frac{1}{l} + \frac{1}{c} \right) \tag{5.5}$$

La última expresión representa una aproximación más intuitiva del resultado y no reemplaza la fórmula exacta. Podemos analizar con la expresión aproximada del almacenamiento el efecto del parámetro l. En la medida que l disminuye el almacenamiento aumenta; a la inversa, cuando l crece el almacenamiento disminuye. Este comportamiento está explicado por la mayor o menor cantidad de snapshots que hay que considerar dependiendo del valor de l. Notar que el espacio destinado a almacenar cambios no depende del tamaño de l. En la validación experimental del modelo se usó la fórmula exacta de TB^{SEST_L}, y no la aproximación.

5.5.2 Estimación del costo de las consultas espacio-temporales con el SEST_L

El costo (cantidad de bloques accesados) del SEST_L puede ser estimado sumando el tiempo de acceso a los nodos del R-tree, sin considerar las hojas, y el tiempo para procesar todas las bitácoras que intersecta el rango espacial de la consulta. En lo que sigue, denotaremos a un R-tree sin las hojas como Rp-tree (R-tree podado). Los nodos hojas del Rp-tree contienen los MBRs de todos los nodos hojas del R-tree.

El número de objetos que se almacenan en el Rp-tree es $N_p = \frac{N}{f}$. Sea $h_p = \left\lceil \log_f \frac{N_p}{f} \right\rceil = h - 1$ la altura del Rp-tree. El número de nodos accesados en el Rp-tree está dado por la Ec. (5.6) la cual se obtiene a partir de la Ec. (2.4) para el caso particular $n = 2$ y $q = q_1 = q_2$.

$$DA_{Rp-tree} \;=\; 1 + \sum_{j=1}^{h_p} \left(\sqrt{D_{j+1}} + q \cdot \sqrt{\frac{N_p}{f^j}} \right)^2 \;=\; DA - \left(\sqrt{D_1} + q \cdot \sqrt{\frac{N}{f}} \right)^2 \tag{5.6}$$

En la Ec. (5.6), se usó D_{j+1} ya que los objetos en el Rp-tree corresponden a las hojas del R-tree. El número de bitácoras a ser procesadas se determina por medio de la densidad de los MBRs

de las hojas y del rango espacial dado en la consulta. Este número de bitácoras queda definido por la Ec. (5.7), donde $D_1 = \left(1 + \frac{\sqrt{D_0}-1}{\sqrt{f}}\right)^2$. Notar que la Ec. (5.7) representa la cantidad de hojas del R-tree que son accesadas por la componente espacial de la consulta (q).

$$NL = \left(\sqrt{D_1} + q \cdot \sqrt{\frac{N}{f}}\right)^2 \tag{5.7}$$

Por lo tanto, la cantidad de bloques accesados por una consulta de tipo *time-slice* queda definido por Ec. (5.8).

$$DA^{SEST_L}_{time-slice} = DA_{Rp-tree} + NL \cdot \left(1 + \left\lceil \frac{nl}{2 \cdot c} \right\rceil\right) \approx q^2 \cdot \frac{N}{f} \cdot \left(1 + \frac{l}{2 \cdot c}\right) \tag{5.8}$$

De la misma manera, la cantidad promedio de bloques accesados por una consulta de tipo *time-interval* queda determinada por la Ec. (5.9).

$$DA^{SEST_L}_{time-interval} = DA^{SEST_L}_{time-slice} + NL \cdot \left\lceil \frac{(ai-1) \cdot p \cdot f}{c} \right\rceil \approx q^2 \cdot \frac{N}{f} \cdot \left(1 + \frac{\frac{l}{2} + ai \cdot p \cdot f}{c}\right) \tag{5.9}$$

Nuevamente, las aproximaciones de las ecuaciones para calcular $DA^{SEST_L}_{time-slice}$ y $DA^{SEST_L}_{time-interval}$ sólo sirven para dar una intuición del resultado y no reemplazan a las fórmulas exactas.

Debido a que el procesamiento de las consultas sobre eventos se lleva a cabo de manera muy similar a una consulta de tipo *time-slice* (ver el Alg. 5.3), el costo de una consulta de este tipo se puede estimar con el modelo de costo definido para una consulta de tipo *time-slice*.

5.5.3 Evaluación experimental del modelo de costo

Con el propósito de validar el modelo de costo se realizaron varios experimentos utilizando datos artificiales (sintéticos) obtenidos con el generador GSTD [TSN99]. En los experimentos se utilizaron 23.268 objetos (puntos) y 200 instantes de tiempo con movilidades de 1%, 5%, 10%, 15%, 20% y 25%. Se consideraron 3 valores para el parámetro d, a saber, 2, 4 y 8 bloques de disco de 1.024 bytes, donde $l = d \cdot c$ (en los experimentos se consideró $c = 21$ cambios[3] por bloque). El valor utilizado para f fue de 34 (68% de la capacidad de un nodo de un R-tree que almacena como máximo 50 entradas [TS96]).

5.5.3.1 Estimación del almacenamiento

La Figura 5.16 indica que el costo en almacenamiento que predice el modelo y el obtenido por medio de los experimentos son muy similares para los valores de d analizados, alcanzando un error relativo promedio de un 15%.

[3] Cada cambio considera una operación *move_out* más una operación *move_in* y cada operación requiere de 24 bytes (4 bytes para almacenar el *Oid* de un objeto, 4 bytes para almacenar un instante de tiempo y 16 bytes almacenar un MBR).

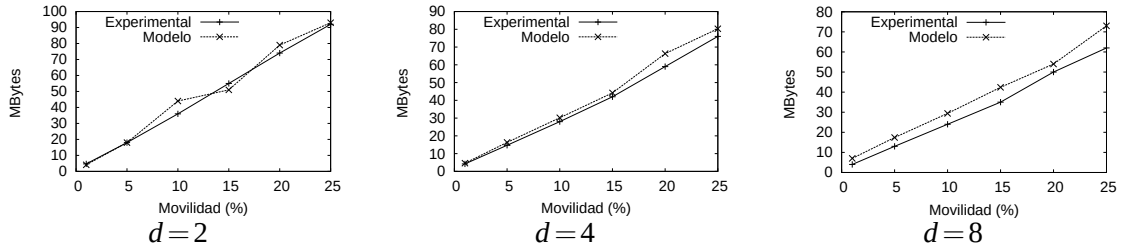


Figura 5.16: Estimación del almacenamiento utilizado por el SEST$_L$.

5.5.3.2 Estimación del rendimiento de las consultas *time-slice* y *time-interval*.

Probamos la capacidad del modelo para predecir el costo de las consultas de tipo *time-slice* y *time-interval* considerando rangos espaciales de las consultas formados por 2% – 12% de cada dimensión y diferentes longitudes del intervalo temporal que van desde 1 hasta 80 instantes de tiempo. Las Figuras 5.17, 5.18, 5.19 y 5.20 muestran que la predicción del rendimiento de las consultas por el modelo es muy buena, alcanzando sólo a un 8% de error relativo promedio.

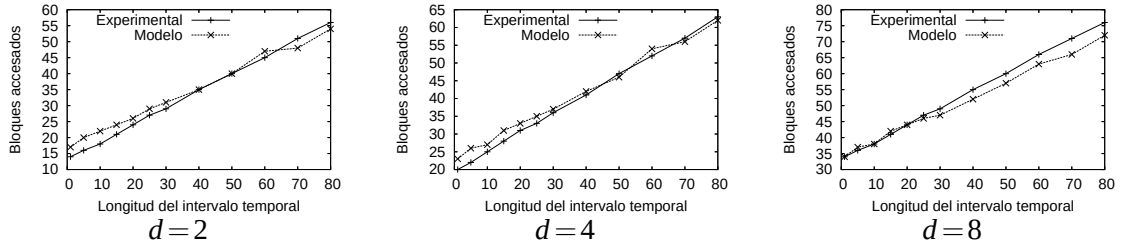


Figura 5.17: Estimación del tiempo de las consultas (5% de movilidad y rango espacial de la consulta formado por el 6% de cada dimensión).

5.5.4 Modelo de costo del SEST$_L$ con snapshots globales

En esta sección se describe un modelo de costo para estimar el tiempo y almacenamiento del SEST$_L$ con snapshots globales. La idea de los snapshots globales es mantener un rendimiento estable de la estructura a lo largo del tiempo, ya que este rendimiento se deteriora como consecuencia del crecimiento del valor de la densidad del R-tree inicial. El modelo de costo asume que la cantidad de objetos es la misma a lo largo del tiempo y que estos siguen una distribución uniforme en el espacio. También supone que la densidad de los R-tree es la misma para todos los snapshots globales.

Para incorporar el efecto de los snapshots globales en el modelo, es necesario estimar cada cuántos instantes de tiempo (*igs*) se necesita crear un nuevo R-tree. Para lograr esto, se estimó la cantidad de objetos (M_1) que debe contener cada nodo de tal manera de alcanzar el valor umbral

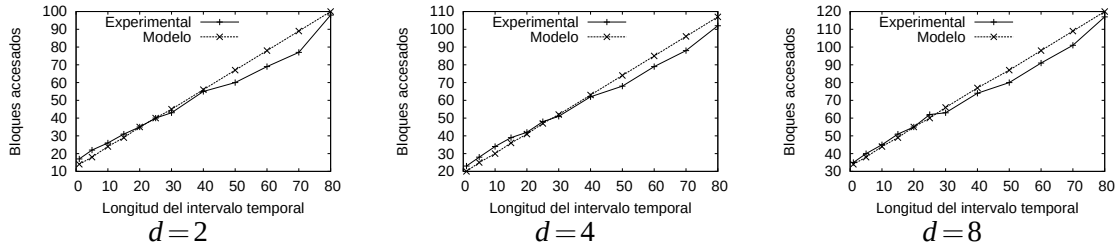


Figura 5.18: Estimación del tiempo de las consultas (10% de movilidad y rango espacial de la consulta formado por el 6% de cada dimensión).

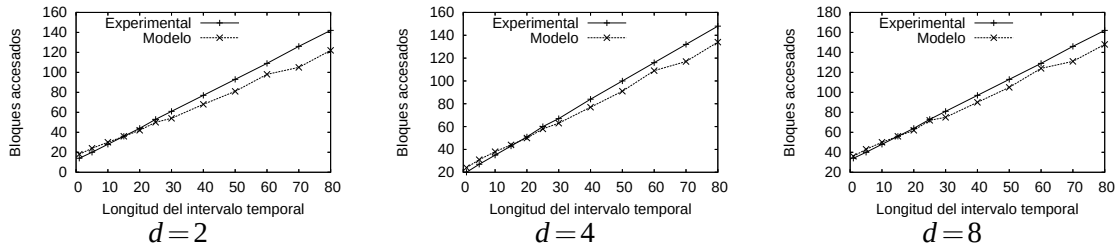


Figura 5.19: Estimación del tiempo de las consultas (15% de movilidad y rango espacial de la consulta formado por el 6% de cada dimensión).

establecido para la densidad, es decir, $D_1' = ls \cdot D_1$, donde D_1 es la densidad inicial del R-tree y D_1' es la densidad que al ser alcanzada implica crear un nuevo snapshot global. La densidad D_1' se puede alcanzar modelando las operaciones *move_in* como inserciones en el R-tree ya que éstas tienen un efecto sobre la densidad del R-tree a pesar de que se almacenen como eventos en las bitácoras. En otras palabras, la inserción de un *move_in* puede provocar que aumente el área del MBR de la bitácora elegida para contenerlo.

Al considerar $f = cc \cdot M$ y la Ec. (2.7) para $n = 2$, podemos escribir la Ec. (5.10) para estimar la densidad que forman los MBRs de las hojas del R-tree.

$$D_1 = \left(1 + \frac{\sqrt{D_0} - 1}{\sqrt{cc \cdot M}}\right)^2 \tag{5.10}$$

De esta forma, es posible obtener M_1 por medio de Ec. (5.11).

$$\left(1 + \frac{\sqrt{D_0} - 1}{\sqrt{cc \cdot M1}}\right)^2 = ls \cdot \left(1 + \frac{\sqrt{D_0} - 1}{\sqrt{cc \cdot M}}\right)^2 \tag{5.11}$$

A partir de la Ec. (5.11), M_1 se puede expresar como se indica en la Ec. (5.12).

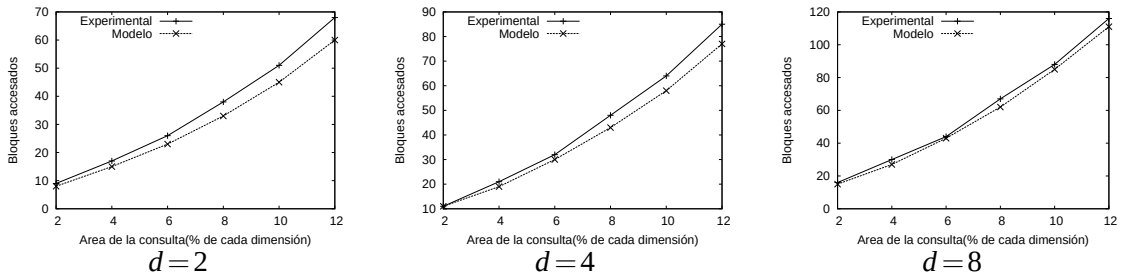


Figura 5.20: Estimación del tiempo de las consultas (10% de movilidad y longitud del intervalo temporal igual a 10).

$$M_1 = \frac{\left(\sqrt{D_0}-1\right)^2}{cc\cdot\left(\sqrt{ls}\cdot\left(1+\frac{\sqrt{D_0}-1}{\sqrt{cc\cdot M}}\right)-1\right)^2} \tag{5.12}$$

Tal como se comentó, el valor de M_1 indica cuántos objetos es necesario insertar para lograr la densidad D_1'. Con M_1 se puede obtener cada cuántos instantes de tiempo (*igs*) se necesita crear un nuevo snapshot global. Para ello, se obtiene la cantidad total de objetos almacenados en el R-tree al considerar la capacidad máxima de un nodo del R-tree igual a M_1. Esto es $M_1 \cdot \frac{N}{cc\cdot M}$, donde $\frac{N}{cc\cdot M}$ es la cantidad de nodos hoja (o cantidad de bitácoras) del R-tree. De esta forma el total de cambios que es necesario insertar en el R-tree para alcanzar D_1' es $M_1 \cdot \frac{N}{cc\cdot M} - N$. Por otro lado, la cantidad de objetos a insertar en el R-tree en cada instante de tiempo es $N \cdot p$ y, por lo tanto, $igs = \left\lceil \frac{M1\cdot\frac{N}{cc\cdot M}-N}{N\cdot p} \right\rceil$ es el número de instantes de tiempo que se pueden almacenar en las bitácoras antes que un nuevo snapshot global se cree. Al simplificar esta ecuación se obtiene la Ec. (5.13).

$$igs = \left\lceil \frac{\left(\frac{M_1}{cc\cdot M}-1\right)}{p} \right\rceil \tag{5.13}$$

Por ejemplo si se considera las siguientes asignaciones a la variables: $cc = 0,68$, $N = 20.000$, $M = 50$; $ls = 1,3$, $p = 0,10$ y $D_0 = 0$, el valor para M_1 es de aproximadamente 480 y el de *igs* es de 132, lo que significa que será necesario crear un nuevo snapshot global cada 132 instantes de tiempo.

5.5.4.1 Estimación del almacenamiento del SEST_L con snapshots globales

La estimación del almacenamiento ocupado por el SEST_L se torna bastante simple una vez que se conoce *igs*, ya que basta con considerar que habrá $ngs = \lceil \frac{nt}{igs} \rceil$ snapshots globales. En cada uno de los $ngs - 1$ snapshots globales habrá que almacenar los cambios producidos en *igs* instantes de tiempo. Además, existirá un snapshot (el último) en el cual sólo se almacenarán los cambios producidos en $lt = nt - igs \cdot \lfloor \frac{nt}{igs} \rfloor$ instantes de tiempo. Con la Ec. (5.4) y reemplazando adecuadamente *nt* por *igs* ($TB_{bitacora}\langle igs \rangle$) ó *lt* ($TB_{bitacora}\langle lt \rangle$) se obtiene el almacenamiento ocupado por las bitácoras de cada snapshot global. De esta forma el almacenamiento ocupado por el SEST_L con snapshot globales queda determinado por la Ec. (5.14).

$$TBSG^{SEST_L} = \left(\left\lceil \frac{nt}{igs} \right\rceil - 1 \right) \cdot ((TN - NB) + NB \cdot TB_{bitacora}\langle igs \rangle) + (TN - NB) + NB \cdot TB_{bitacora}\langle lt \rangle \quad (5.14)$$

5.5.4.2 Estimación de la eficiencia de las consultas con el SEST$_L$ considerando snapshots globales

Es claro que el modelo de costo definido por la Ec. (5.8) para las consultas *time-slice* no cambia, ya que estas consultas son procesadas de la misma manera ya sea con el SEST$_L$ o con el SEST$_L$ considerando snapshots globales. Para las consultas *time-interval*, lo primero que es necesario conocer es la cantidad de snapshots globales que el intervalo temporal (de longitud *ai*) de la consulta accesará y que se define por la Ec. (5.15).

$$ngsi = \left\lceil \frac{ai}{igs} \right\rceil \quad (5.15)$$

La longitud del intervalo temporal *ai* se transforma en un conjunto de $ni = \left\lfloor \frac{ai}{igs} \right\rfloor$ intervalos de longitud *igs* y un intervalo adicional de longitud $lr = ai - ni \cdot igs$. De esta forma, para determinar el costo del SEST$_L$, se ejecutan *ni* consultas con intervalo temporal *igs*, más una consulta adicional de longitud lr. Para cada R-tree (snapshot global), la densidad de los MBRs de las hojas es a lo más $D'_1 = ls \cdot D_1$ y, por lo tanto, el número de bitácoras que el rango espacial de la consulta intersectará queda determinado por la Ec. (5.16).

$$NB = \left(\sqrt{D'_1} + q \cdot \sqrt{\frac{N}{cc \cdot M}} \right)^2 \quad (5.16)$$

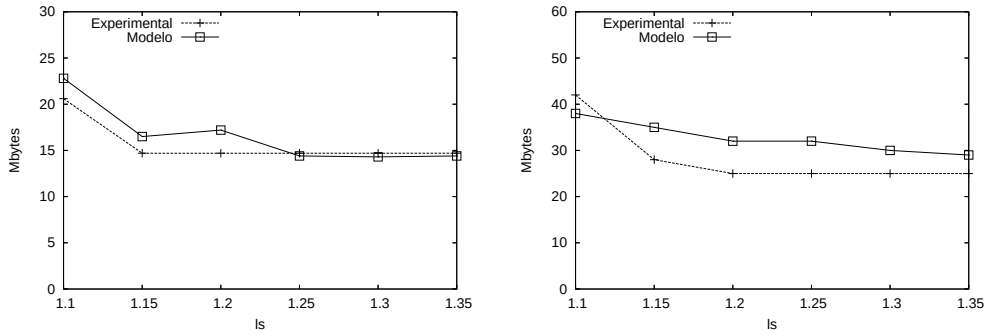
Finalmente, reemplazando *ai* por *igs* ($DA^{SEST_L}_{time-interval}\langle igs \rangle$) y lr ($DA^{SEST_L}_{time-interval}\langle lr \rangle$) en la ecuación Ec. (5.9) podemos obtener el número de bloques accesados por el SEST$_L$ en una consulta espacio-temporal (Ec. (5.17)).

$$DASG^{SEST_L}_{time-interval} = ni \cdot DA^{SEST_L}_{time-interval}\langle igs \rangle + DA^{SEST_L}_{time-interval}\langle lr \rangle \quad (5.17)$$

5.5.4.3 Evaluación experimental del modelo de costo del SEST$_L$ con snapshots globales

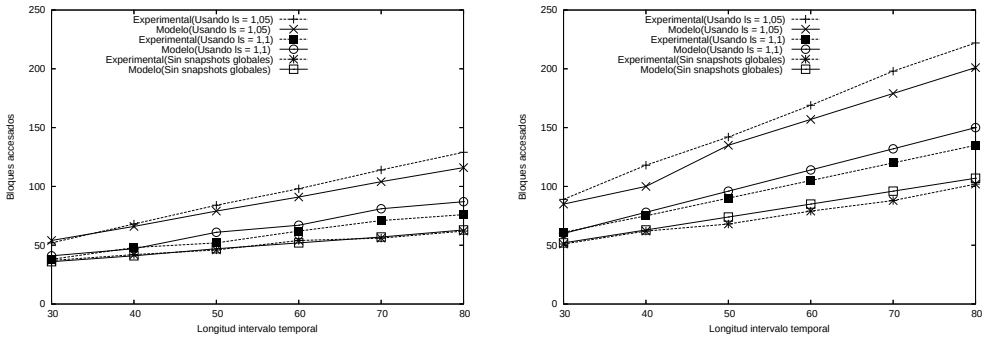
Con el propósito de validar el modelo de costo del SEST$_L$ con snapshots globales, se realizaron varios experimentos utilizando datos obtenidos con el generador de objetos espacio-temporales GSTD [TSN99]. Los experimentos utilizaron 23.268 objetos (puntos) y 200 instantes de tiempo con movilidades de 5% y 10% y 4 bloques para el parámetro d. También se consideraron los valores entre 1,1 y 1,35 para *ls*. En la Figura 5.21 podemos ver que el modelo predice con bastante exactitud el espacio ocupado por el SEST$_L$ con snapshots globales, alcanzando un error promedio de un 14%. El modelo estima el rendimiento de las consultas con un error promedio de un 14% (ver Figuras 5.22 y 5.23). Notar que en la Figura 5.22, el rendimiento de las consultas aparece muy bajo comparado con el SEST$_L$ original. Esto sucede por que se eligieron valores para *ls* muy estrictos, lo cual se hizo con el propósito de mostrar una clara diferencia con la estructura original. Al considerar una distribución uniforme, una parametrización correcta ($ls = 1,20$) debería

recomendar muy pocos snapshots globales y prácticamente no existirían diferencias con el SEST$_L$ original. La ventaja de usar snapshots globales se advierte al momento de usar datos que no tienen una distribución uniforme tal como se muestra en la Sección 5.4.3.4.



a) 5% de movilidad b) 10% de movilidad

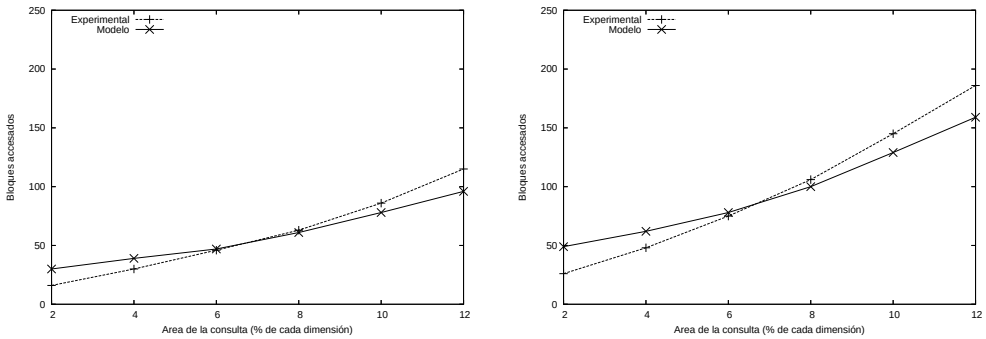
Figura 5.21: Estimación del almacenamiento utilizado por el SEST$_L$ considerando snapshots globales.



a) 5% de movilidad b) 10% de movilidad

Figura 5.22: Estimación de la eficiencia de las consultas procesadas con el SEST$_L$ considerando snapshots globales y distribución uniforme (rango espacial de las consultas formado por el 6% de cada dimensión).

El modelo del SEST$_L$ con snapshots globales también puede ser útil sobre conjuntos de datos con distribución no uniforme. La razón es que la estrategia de los snapshots globales, en cierto sentido, hace que la estructura con datos no uniformes se comporte de manera similar al caso de datos uniformemente distribuidos. Para adaptar el modelo a conjuntos de datos con distribución arbitraria, sólo es necesario calcular la densidad inicial y cada cuántos instantes de tiempo (*igs*) crear un nuevo snapshot global, de acuerdo a los datos reales. Para la densidad, en lugar de usar la Ec. (5.10), se construye el R-tree con los datos espaciales y se calcula *denReal* por medio de la Ec. (5.1). Esta densidad se considera como D_1. Para el valor de *igs*, se necesita calcular M_1 para distribuciones arbitrarias. Para ello se reemplaza la Ec. (5.12) por el procedimiento que construye

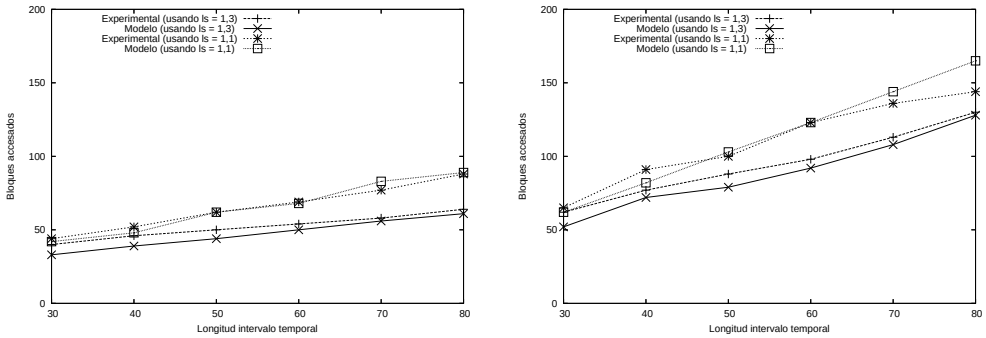


a) 5% de movilidad b) 10% de movilidad

Figura 5.23: Estimación de la eficiencia de las consultas procesadas con el SEST$_L$ considerando snapshots globales y distribución uniforme (longitud del intervalo temporal igual a 40 unidades y $ls = 1,1$).

un R-tree a partir de los datos iniciales y luego se aumentan las capacidades de los nodos hasta alcanzar la densidad $D'_1 = ls \cdot D_1$. Esto es lo único que se necesita calcular sobre los datos reales (no se necesita procesar eventos ni evaluar consultas, por ejemplo). Todas las restantes ecuaciones del modelo se pueden utilizar tal cual se establecieron, una vez que se han obtenido los valores de D_1 y M_1.

En la Figura 5.24 se compara experimentalmente el modelo, considerando los datos del conjunto *NUD*. Se puede ver que el modelo, a pesar de ser definido para datos con distribución uniforme, estima bastante bien el comportamiento de la estructura sobre datos con distribución no uniforme. El almacenamiento también se estima muy bien: el error es normalmente muy bajo (por ejemplo, 3% para $ls = 1,3$). Para un valor muy bajo de ls el error puede ser mayor (por ejemplo, 20% para $ls = 1,10$), pero en la práctica estos valores bajos de ls no se usan.



a) 5% de movilidad b) 10% de movilidad

Figura 5.24: Estimación de la eficiencia de las consultas procesadas con el SEST$_L$ considerando snapshots globales para el conjunto de datos *NUD* (5% y 10% de movilidad y un rango espacial formado por el 6% en cada dimensión).

En la tabla 5.1 se pueden apreciar las principales ventajas de SEST$_L$.

I T E M	Evaluación
SEST-Index (Distribución Uniforme y no Uniforme)	
1) Almacenamiento	Requiere de menos almacenamiento que el MVR-tree (entre 50% y 65% del utilizado por MVR-tree) y que el SEST-Index.
2) Consultas	
a) *time-interval*	Superior que el MVR-tree y el MV3R-tree.
b) *sobre eventos*	Superior al MVR-tree pero inferior al SEST-Index.
c) *time-slice*	Similar al MVR-tree y al MVR-tree.
3) Modelo de costo	
a) Almacenamiento	Error relativo promedio de 14%.
b) Consultas	Error relativo promedio de 11%.

Tabla 5.1: Resumen de las principales ventajas de SEST$_L$.

5.6 Conclusiones

En este capítulo se propone un nuevo método de acceso espacio-temporal, denominado SEST$_L$, que maneja eventos y snapshots asociados con particiones del espacio. Basado en resultados experimentales, el SEST$_L$ (con parámetro $d = 4$) requiere de un 58% del almacenamiento utilizado por el MVR-tree, el mejor método de acceso conocido hasta ahora. Por otra parte, el SEST$_L$ supera al MVR-tree para consultas de tipo *time-interval*. Distinto a otros métodos de acceso espacio-temporales propuestos, con el SEST$_L$ también es posible procesar consultas sobre eventos con una eficiencia similar a los algoritmos utlizados para procesar consultas de tipo *time-slice* y de tipo *time-interval*.

En este capítulo también se describió y validó un modelo de costo que permite estimar el almacenamiento y la eficiencia de las consultas procesadas con el SEST$_L$ (con y sin snapshots globales) con un error relativo promedio de un 15% para el almacenamiento y de un 11% para las consultas.

Capítulo 6

Generalización del enfoque

6.1 Introducción

En los Capítulos 4 y 5 se proponen dos métodos de acceso espacio-temporales los que se basan en el enfoque de snapshots y eventos. En el primero de ellos (SEST-Index) se asigna una sola bitácora al espacio completo. En el segundo caso (SEST$_L$) las bitácoras se asignan a las subregiones formadas por las hojas del R-tree. Las evaluaciones realizadas en el Capítulo 5 indican que, en general, para las consultas de tipo *time-slice* y *time-interval* el SEST$_L$ supera al SEST-Index. A partir de estos resultados es posible suponer que con el SEST$_L$ se logra el mejor rendimiento de nuestro enfoque para este tipo de consultas, sin embargo, no es posible garantizar que sea lo óptimo. Es decir, podría ser que asignando las bitácoras a subregiones de los nodos internos (entre la raíz y las hojas) se logren mejores resultados. En este capítulo se trata de despejar esta interrogante generalizando los modelos de costos descritos para el SEST-Index y el SEST$_L$ para estimar el almacenamiento y la eficiencia de las consultas de tipo *time-slice* y *time-interval*.

La estructura de este capítulo es como sigue. En la Sección 6.2 se propone el modelo de costo general de nuestro enfoque que considera la asignación de las bitácoras a subregiones de cualquier nivel del R-tree, tornándose los modelos del SEST-Index y del SEST$_L$ en casos particulares del modelo general. La Sección 6.3 está dedicada a comparar, utilizando el modelo general, el rendimiento de nuestro enfoque al asignar las bitácoras en los diferentes niveles del R-tree. En la Sección 6.4 se hace un análisis teórico del enfoque utilizando aproximaciones del modelo de costo. Finalmente, en la Sección 6.5 se discuten las conclusiones de este capítulo.

6.2 Generalización de los modelos de costo del SEST-Index y del SEST$_L$

La Figura 6.1 muestra una generalización del enfoque basado en snapshots y eventos donde es posible observar que las bitácoras se han asignado al nivel *i* (sólo se muestra la bitácora del subárbol *A*). Similarmente al SEST$_L$, en el esquema general se considera que una bitácora está compuesta de dos partes: (i) snapshots y (ii) eventos. El modelo supone el uso de un R-tree para almacenar los objetos en los snapshots al que denominaremos Rs-tree. Los eventos se almacenan en una lista simple de nodos ordenados por el tiempo. También se define un R-tree podado (Rp-tree en la Figura 6.1) como un R-tree sin los subárboles Rs-tree. Por otra parte, el modelo considera objetos

cuya componente espacial se define en un espacio de dos dimensiones ($n = 2$) y que los objetos siguen una distribución uniforme en el espacio. Aquí se utilizan las mismas variables descritas en la Tabla 4.2, como también otras definidas localmente en este capítulo. En ocasiones se utilizan variables subindizadas/superindizadas por el nivel del R-tree a cuyas subregiones se ha decidido asignar las bitácoras. Por ejemplo, NB_i significa el número de bitácoras que se necesitan si se decide asignarlas a las subregiones del nivel i.

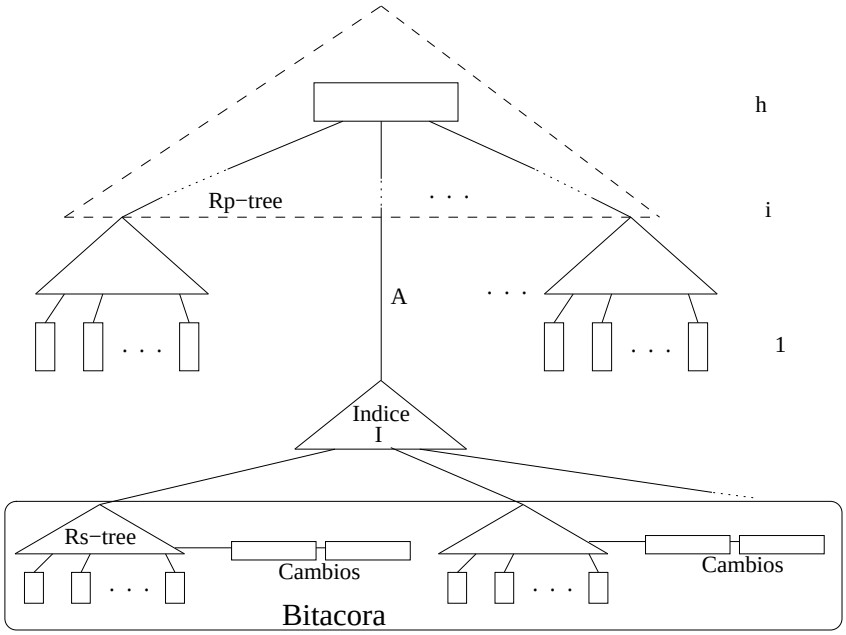


Figura 6.1: Generalización del enfoque de snapshots y eventos.

6.2.1 Modelo general para determinar el almacenamiento

En la definición del modelo general no se consideran la existencia de snapshots globales, pues hemos asumido una distribución uniforme de los objetos en el espacio y bajo este supuesto la densidad del R-tree tiende a mantenerse en el tiempo.

Para el caso del SEST_L (bitácoras asignadas a subregiones del nivel 1), el número de bitácoras que se necesitan se obtiene mediante $\frac{N}{f}$ y el número de objetos a almacenar en cada snapshot es aproximadamente igual a f, es decir, se necesita de aproximadamente 1 bloque por cada snapshot (un Rs-tree con sólo un nodo). En el otro extremo se encuentra el SEST-Index, donde existe una sola bitácora y se requiere, por cada snapshot, una cantidad de bloques igual a la cantidad utilizada para almacenar todos los objetos en un R-tree (en este caso los Rp-trees son vacíos y, por lo tanto, existen solamente Rs-trees). Del mismo modo, si se asignan las bitácoras a las subregiones del nivel 2, se necesitan $\frac{N}{f^2}$ bitácoras y la cantidad de objetos a almacenar en cada snapshot está dada por f^2. En general, en el nivel i, con $1 \leq i \leq h$, la cantidad de bitácoras está dada por la ecuación

$$NB_i = \frac{N}{f^i} \tag{6.1}$$

y la cantidad de objetos a almacenar en cada snapshot (N_i) se obtiene con la expresión

$$N_i = \begin{cases} f^i & \text{si } 1 \leq i < h \\ N & \text{si } i = h \end{cases} \tag{6.2}$$

La altura del árbol Rs-tree$_i$ es i y el número de nodos del árbol Rs-tree$_i$ queda determinado por la Ec. (6.3)

$$TN_i = \begin{cases} 1 & \text{si } i = 1 \\ i + \left\lfloor \frac{(f^i - 1)}{(f-1)} \right\rfloor & \text{si } 1 < i \leq h \end{cases} \tag{6.3}$$

Ahora se puede calcular el número de instantes de tiempo que es posible almacenar por bitácora (il_i) de acuerdo con la capacidad establecida (l_i) (Ec. (6.4)).

$$il_i = \frac{l_i}{f^i \cdot p} \tag{6.4}$$

En el caso general (bitácoras en diferentes niveles) también se considera que los eventos producidos en un instante de tiempo se almacenan completamente en la bitácora antes de crear un nuevo snapshot. Esta consideración exige redefinir un nuevo tamaño de bitácora (nl_i), el cual queda definido por la Ec. (6.5).

$$nl_i = p \cdot N_i \cdot il_i \tag{6.5}$$

De esta forma, el tamaño (bloques) de cada bitácora ubicada en el nivel i (TB_i) se obtiene por la Ec. (6.6). El primer término de la Ec. (6.6) calcula la cantidad de almacenamiento que ocupan todos los Rs-trees. El segundo término obtiene el almacenamiento que ocupan los cambios producidos hasta justo antes del último snapshot creado. El último término corresponde al almacenamiento que se necesita para almacenar los cambios que han ocurrido después del último snapshot.

$$TB_i = \left\lceil \frac{nt}{il_i} \right\rceil \cdot TN_i + \left(\left\lceil \frac{nt}{il_i} \right\rceil - 1 \right) \cdot \left\lceil \frac{nl_i}{c} \right\rceil + \left\lceil \left(\frac{nt}{il_i} - \left\lfloor \cdot \frac{nt}{il_i} \right\rfloor \right) \cdot \frac{nl_i}{c} \right\rceil \tag{6.6}$$

Finalmente, el espacio total ocupado por el índice (TB^i), al fijar las bitácoras a subregiones del nivel i, queda definido por la Ec. (6.7).

$$TB^i = (TN - TN_i \cdot NB_i) + NB_i \cdot TB_i \tag{6.7}$$

Se puede observar que si se considera que $i = h$, en la Ec. (6.7), se puede obtener la Ec. (4.3) que estima el espacio ocupado por el SEST-Index. De la misma manera, al considerar ($i = 1$) obtenemos la ecuación para estimar el espacio del SEST$_L$, es decir, la Ec. (5.5). En efecto, para el SEST-Index y aplicando la Ec. (6.7) al caso particular de $i = h$, se tiene

$$TB^{SEST-Index} = TB^h = (TN - TN_h \cdot NB_h) + NB_h \cdot TB_h \tag{6.8}$$

Pero como en la Ec. (6.8) $i = h$, entonces $TN_h = TN$ ya que TN_h es el número de bloques utilizados por los Rs-trees al considerar $i = h$. Por otra parte, el término TB_h de la Ec. (6.8) se puede expresar

como $\frac{N}{f^h}$, pero $f^h = N$ y, por lo tanto, la Ec. (6.8) queda como sigue

$$\begin{aligned} TB^h &= \left(TN - TN \cdot \frac{N}{f^h}\right) + NB_h \cdot TB_h \\ TB^h &= NB_h \cdot TB_h, \ pero\ NB_h = \frac{N}{f^h} \\ TB^h &= TB_h \\ TB^h &= \left\lceil \frac{nt}{il_h} \right\rceil \cdot TN_h + \left(\left\lceil \frac{nt}{il_h} \right\rceil - 1 \right) \cdot \left\lceil \frac{nl_h}{c} \right\rceil + \left\lceil \left(\frac{nt}{il_i} - \left\lfloor \cdot \frac{nt}{il_i} \right\rfloor \right) \cdot \frac{nl_i}{c} \right\rceil \end{aligned} \tag{6.9}$$

Pero $il_h = \frac{l \cdot f^h}{N^2 \cdot p}$ y $f^h = N$, por lo tanto, $il_h = \frac{l}{N \cdot p}$ que corresponde a la Ec. (4.1) definida para el SEST-Index. De la misma manera $nl_h = p \cdot f^h \cdot il$, y teniendo que $f^h = N$, entonces $nl_h = p \cdot N \cdot il$ que corresponde a la Ec. (4.2). De esta forma, la Ec. (6.9) se puede expresar como

$$TB^h = \left\lceil \frac{nt}{il} \right\rceil \cdot TN + \left(\left\lceil \frac{nt}{il} \right\rceil - 1 \right) \cdot \left\lceil \frac{nl}{c} \right\rceil + \left\lceil \left(\frac{nt}{il} - \left\lfloor \cdot \frac{nt}{il} \right\rfloor \right) \cdot \frac{nl}{c} \right\rceil \tag{6.10}$$

lo que es igual a la Ec. (4.3).

De manera similar se puede derivar, a partir de la Ec. (6.7), la Ec. (5.5) la cual estima el almacenamiento requerido por el SEST$_L$. Para este caso consideramos $i = 1$ en la Ec. (6.7) con lo que obtenemos

$$TB^{SEST_L} = TB^1 = (TN - TN_1 \cdot NB_1) + NB_1 \cdot TB_1 \tag{6.11}$$

Si consideramos la Ec. (6.3) obtenemos $TN_1 = 1$ y que $NB_1 = NB = \frac{N}{f}$, entonces la Ec.(6.11) se transforma en la Ec. (6.12)

$$TB^{SEST_L} = TB^1 = (TN - NB) + NB \cdot TB_1 \tag{6.12}$$

Pero al considerar $i = 1$ y $l_1 = l$, entonces $TB_1 = TB_{bitacora}$, ya que $il_1 = il = \frac{l}{p \cdot f}$ y $nl_1 = nl$. Por lo tanto, las Ecs. (6.12) y (5.5) son equivalentes. De esta forma, es posible deducir, a partir del modelo general, los modelos que estiman el almacenamiento tanto para SEST-Index como para SEST$_L$.

6.2.2 Modelo para determinar la eficiencia de las consultas

En esta sección establecemos un modelo general para predecir el rendimiento de nuestro enfoque para procesar consultas de tipo *time-slice* (Q,t) y *time-interval* $(Q,[t_i,t_f])$. En este capítulo suponemos que el predicado espacial Q se define mediante un cuadrado de lado q. Utilizaremos la variable i, $1 \leq i \leq h$, para denotar el nivel del R-tree a cuyas subregiones se le asignarán las bitácoras.

En primer lugar estableceremos una expresión para estimar los bloques accesados por una consulta de tipo *time-slice*, que luego extenderemos para las consultas de tipo *time-interval*. El procesamiento de una consulta de tipo *time-slice*, por medio de nuestro enfoque, se puede llevar a cabo en dos fases. En la primera, utilizando el Rp-tree se seleccionan todas las bitácoras que se intersectan con Q. Luego, en la segunda fase, por cada bitácora seleccionada en la fase anterior, se elige, de acuerdo a t, el Rs-tree creado en el último instante de tiempo tr tal que $tr < t$. Sobre este Rs-tree se ejecuta la consulta espacial Q obteniendo los objetos iniciales de la respuesta los que se

actualizan con los eventos producidos en el intervalo $(tr,t]$. La respuesta final se forma por medio de la unión de los objetos obtenidos de cada bitácora procesada.

La primera fase se resuelve principalmente accesando el Rp-tree y, por lo tanto, se necesita estimar cuántos nodos serán accesados del Rp-tree y cuántas bitácoras son intersectadas por Q. La densidad de los objetos que se almacenarán en el Rp-tree se puede obtener a partir de la densidad inicial de los objetos. De esta forma, la Ec. (6.13) (la cual se obtiene a partir de la Ec. (2.7)) define la densidad de los MBRs del nivel i a partir de los cuales se construye el Rp-tree, donde D_0 es la densidad inicial de los N objetos.

$$D_i = \left(1 + \frac{\sqrt{D_{i-1}} - 1}{\sqrt{f}}\right)^2 \tag{6.13}$$

La altura (h_i) de este árbol podado queda determinado por la Ec. (6.14).

$$h_i = 1 + \left\lceil \log_f \frac{NB_i}{f} \right\rceil = \lceil \log_f NB_i \rceil \tag{6.14}$$

De esta forma, el número de nodos del Rp-tree que, en promedio, se necesitan accesar para procesar la consulta espacial que considera a Q como predicado se obtiene de la Ec. (6.15)

$$DAP_i = 1 + \sum_{j=1}^{h_i} \left(\sqrt{D_{j+i}} + q \cdot \sqrt{\frac{NB_i}{f^j}}\right)^2 \tag{6.15}$$

Observe que en la Ec. (6.15) para el caso $j = i+1$, D_j se obtiene a partir de D_{j-1} utilizando la Ec. (2.7.).

El número de bitácoras que se intersectarán con Q en el nivel i (NL_i) del R-tree es equivalente a determinar cuántos nodos del nivel i se intersectarán con Q, es decir,

$$NL_i = \left(\sqrt{D_i} + q \cdot \sqrt{\frac{N}{f^i}}\right)^2. \tag{6.16}$$

La segunda fase, para evaluar una consulta de tipo *time-slice*, consiste en procesar las NL_i bitácoras seleccionadas en la primera fase. Para ello se debe accesar un Rs-tree y luego procesar los bloques con los eventos ocurridos hasta t. La altura de un Rs-tree que almacena N_i objetos es i y el número promedio de nodos accesados por Q queda definido por la Ec. (6.17), donde D_j se obtiene a partir de D_{j-1} utilizando la Ec. (2.7).

$$DAS_i = 1 + \sum_{j=1}^{i} \left(\sqrt{D_j} + q \cdot \sqrt{\frac{N_i}{f^j}}\right)^2 \tag{6.17}$$

Con la Ec. (6.17) es posible obtener una expresión para determinar la cantidad de bloques que, en promedio, se requerirán accesar por cada bitácora seleccionada (bl) (ver Ec. (6.18)).

$$bl = DAS_i + \left\lceil \frac{nl_i}{2 \cdot c} \right\rceil \tag{6.18}$$

Finalmente, con las ecuaciones Ecs. (6.17) y (6.18) se puede definir una expresión para estimar el total de bloques que, en promedio, necesita accesar una consulta de tipo *time-slice* utilizando nuestro enfoque (ver Ec. (6.19)).

$$DA^{i}_{time-slice} = DAP_i + NL_i \cdot bl = DA + \frac{nl_i}{2 \cdot c} \tag{6.19}$$

A partir de la Ec. (6.19) es muy simple obtener una expresión para las consultas de tipo *time-interval* $(Q, [t_1, t_2])$, ya que el costo total se puede calcular considerando el costo de procesar una consulta *time-slice* (Q, t_1), más el costo de procesar los bloques de las bitácoras que almacenan cambios ocurridos en el intervalo $(t, t_2]$. La Ec. (6.20) permite predecir el número de bloques accesados por una consulta de tipo *time-interval*.

$$DA^{i}_{time-interval} = DA^{i}_{time-slice} + NL_i \cdot \left\lceil \frac{(ai-1) \cdot \frac{p \cdot N_i}{NB_i}}{c} \right\rceil \tag{6.20}$$

Con las Ecs. (6.19) y (6.20) es fácil deducir las ecuaciones del SEST-Index y del SEST_L que determinan el costo de procesamiento de las consultas *time-slice* y *time-interval*, respectivamente. Por ejemplo, consideremos las consultas de tipo *time-slice* procesadas con el SEST-Index. En este caso $i = h$ y los Rp-trees son vacíos, de tal manera que sólo tenemos Rs-trees que tienen una altura $h = i$, pues cada Rs-tree almacena todos los objetos vigentes o "vivos" cuando se crea el snapshot. Por otro lado, en la Ec. (6.17), $N_i = N$ y, por lo tanto, la Ec. (6.17) se transforma en la Ec. (4.5). Al considerar $i = h$, entonces $NL_i = 1$ y $nl_i = nl$ (existe una sola bitácora) y, por lo tanto, las Ecs. (4.6) y (6.19) son equivalentes.

6.3 Evaluación de diferentes escenarios con el modelo general

En esta sección se discute un análisis del enfoque de snapshots y eventos utilizando el modelo general previamente definido. Para ello se evaluaron varios escenarios considerando 20.000 objetos (puntos), los que siguieron una distribución uniforme en el espacio de trabajo y se desplazaron a lo largo de 200 instantes de tiempo. Se consideraró el tamaño de un bloque igual a 1.024 bytes, la capacidad máxima de un nodo de un R-tree igual a 50 entradas y una capacidad promedio de 34 entradas. Para estos parámetros la altura (cantidad de niveles) de los R-trees fue de 3.

6.3.1 Escenario 1

En este escenario se definieron los tamaños de las bitácoras (valor del parámetro d) de cada nivel de tal manera que solamente los eventos ocurridos en un instante de tiempo se almacenen entre snapshots consecutivos. Este escenario permite que nuestro enfoque alcance la máxima eficiencia en la evaluación de las consultas sacrificando la utilización del almacenamiento. En la Figura 6.2 se puede observar que al asignar las bitácoras a las subregiones del nivel 1 (SEST_L) se obtiene el mejor rendimiento tanto para el almacenamiento utilizado como en la eficiencia para evaluar las consultas. En el caso particular de las consultas es posible concluir que la asignación de las bitácoras al nivel 1 supera por 2 y 4 órdenes de magnitud a la alternativa de ubicarlas en el nivel 2 y 3, respectivamente, utilizando aproximadamente la misma cantidad de almacenamiento.

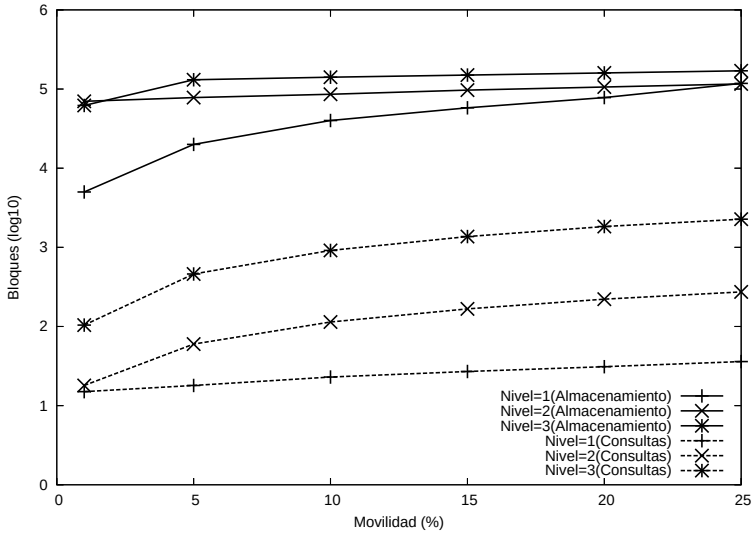


Figura 6.2: Escenario 1: Longitud del intervalo temporal 10 y área formada por el 6% de cada dimensión.

6.3.2 Escenario 2

En este escenario se privilegió la utilización del almacenamiento en lugar de la eficiencia de las consultas. El menor almacenamiento se logra teniendo un sólo snapshot por subregión y un valor para *d* tal que quepan todos los eventos producidos en todos los instantes de tiempo almacenados en la base de datos (200 para estos experimentos). En la Figura 6.3 se muestran los resultados de este experimento. Tal como se esperaba, el almacenamiento ocupado es prácticamente el mismo independientemente del nivel al cual se asignan las bitácoras. Sin embargo, nuevamente la alternativa de ubicar las bitácoras en el nivel 1 supera por varios ordenes de magnitud la asignación a los niveles 2 ó 3.

6.3.3 Escenario 3

El objetivo del experimento realizado en este escenario es analizar el efecto de la longitud del intervalo temporal, el área, y el nivel donde se ubican las bitácoras, en el rendimiento de las consultas. En este experimento se consideró el almacenamiento ocupado es el mismo independiente del nivel al cual se asignan las bitácoras. De esta forma el parámetro *d* fue de 1, 5 y 66 al asignar las bitácoras al nivel 1, 2 y 3 respectivamente. Las consultas se formaron considerando dos longitudes para el intervalo temporal (1, 10 y 40 unidades de tiempo) y 16 diferentes rangos espaciales formados por 5% – 80% de cada dimensión. La Figura 6.4 muestra los resultados de este experimento en la cual podemos observar que la asignación de las bitácoras al nivel 1 tiende a empeorar conforme crece el área de las consultas. Por ejemplo, para el caso *time-slice*, es decir $ai = 1$, la curva del nivel 1 cruza a las curvas del nivel 2 y 3 en áreas formadas por el 20% y 40% de cada dimensión, respectivamente. De la misma manera la curva del nivel 2 cruza a la curva del nivel 3 cuando el área corresponde a una formada por el 60% de cada dimensión aproximadamente. Este comportamiento también lo podemos observar para consultas

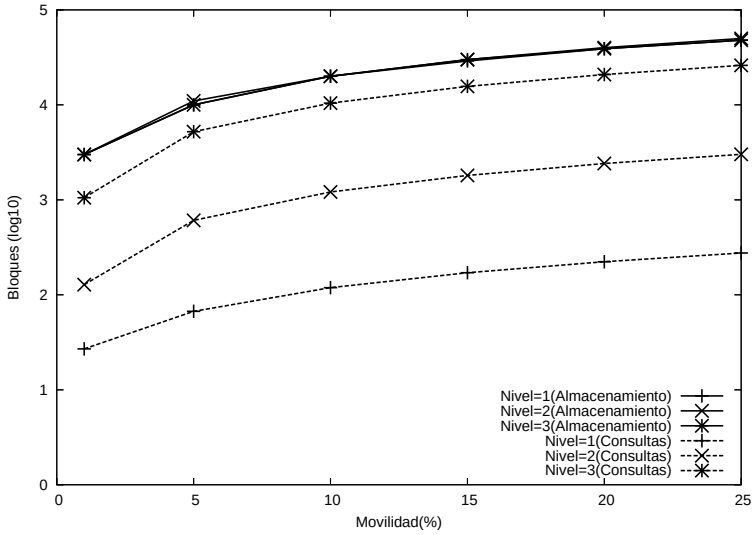


Figura 6.3: Escenario 2: Longitud del intervalo temporal 10 y área formada por el 6% de cada dimensión.

con longitud temporal igual a 10 y 40 unidades de tiempo, sólo que el cruce entre las curvas se va produciendo con áreas cada vez mayores.

El comportamiento descrito anteriormente se explica por la cantidad de bloques de tipo fb e *ib* que se deben accesar para evaluar las consultas. Los bloques fb se definen como aquellos bloques de bitácora destinados a almacenar eventos producidos en el instante de tiempo t y $t+i$ con $i \geq 1$, donde t es el instante de tiempo final especificado en el intervalo temporal de la consulta (si la consulta es de tipo *time-slice* el tiempo inicial y final del intervalo temporal son iguales). En un bloque fb pueden haber eventos de varios instantes de tiempo en los que se incluye los ocurridos en t. Los bloques *ib* corresponden a aquellos bloques de bitácora (también destinados a almacenar eventos) que es necesario leer antes de un bloque fb al procesar una consulta. Supongamos que ejecutamos una consulta sobre un rango espacial de área pequeña y que hemos asignado las bitácoras al nivel h del R-tree. En este caso habrá que accesar muchos bloques de tipo *ib* y sólo uno de tipo fb. Si aumentamos el área de la consulta la cantidad de bloques *ib* será aproximadamente la misma y nuevamente tendremos que accesar un sólo bloque de tipo fb. Notar que en la medida que crece el área de la consulta la información de los *ib* es más util para formar la respuesta. Si asignamos las bitácoras al nivel 1 y consideramos un rango espacial de área pequeña, habrá menos bloques de tipo *ib* que accesar, pero más de tipo fb. Sin embargo, al considerar un rango espacial con área grande, será necesario procesar muchas bitácoras y por lo tanto la cantidad de bloques de tipo fb que hay que accesar aumenta demasiado. En otras palabras, entre mayor es el área del rango espacial de la consulta y menor es el nivel del R-tree al cual se asignan las bitácoras, mayor es la cantidad de bloques de tipo fb que se deben accesar.

Notar que en la medida que la longitud del intervalo temporal de las consultas aumenta, el área donde las curvas se cruzan también aumenta. Esto se explica por que la proporción de bloques de tipo fb con respecto a los bloques de tipo *ib* disminuye en la medida que la longitud del intervalo temporal de la consulta aumenta.

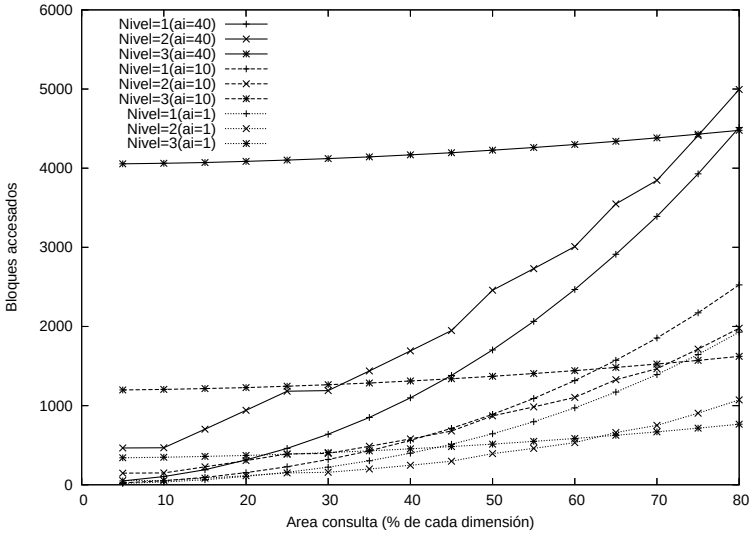


Figura 6.4: Escenario 3: Movilidad de 10% (*ai* representa la longitud del intervalo temporal).

6.4 Análisis teórico del enfoque utilizando el modelo general de costo

En esta sección se hace un análisis de nuestro enfoque utilizando el modelo de costo. Específicamente se ilustra de manera teórica el efecto que, sobre el almacenamiento y la eficiencia de las consultas, tiene el nivel del R-tree a cuyas regiones se asignan las bitácoras. Dado lo engorroso de algunas expresiones del modelo y que la incidencia de algunos términos de las ecuaciones no tienen un gran impacto sobre el resultado, se hicieron aproximaciones que permiten vislumbrar de manera más intuitiva el efecto del nivel del R-tree sobre el almacenamiento y el rendimiento de las consultas.

6.4.1 Análisis del comportamiento del enfoque sobre el almacenamiento

Tal como anticipamos, en primer lugar haremos una simplificación que permite obtener el total de almacenamiento ocupado por el índice, es decir, la Ec. (6.7). La primera aproximación es sobre la cantidad de nodos de un Rs-tree al nivel i, es decir, $TN_i \approx f^{i-1}$. Luego se aproximó el último sumando de la Ec. (6.6), es decir, $\left\lceil \frac{nt - \lfloor \frac{nt}{il_i} \rfloor \cdot il_i}{il_i} \cdot \frac{nl_i}{c} \right\rceil$. Este término representa la cantidad de bloques destinado a almacenar cambios después del último snapshot realizado en la bitácora. Para efectos de simplicación se supone que, en promedio, esta sección de la bitácora se encuentra llena hasta la mitad y, por lo tanto, el número de bloques queda determinado por $\frac{l_i}{2 \cdot c}$. De esta forma, al redondear las aproximaciones (entero superior e inferior) y reemplazando il_i por $\frac{l_i}{f^i \cdot p}$ podemos obtener una aproximación para la Ec. (6.6) dada por la Ec. (6.21)[1].

$$TB_i \approx \frac{nt \cdot f^i \cdot p \cdot f^{i-1}}{l_i} + \frac{nt \cdot p \cdot f^i}{c} + \frac{l_i}{2 \cdot c} = nt \cdot f^i \cdot p \cdot \left(\frac{f^{i-1}}{l_i} + \frac{1}{c} \right) + \frac{l_i}{2 \cdot c} \qquad (6.21)$$

[1] A pesar de que esta expresión, y otras más adelante en este capítulo, representan aproximaciones, nosotros las referenciaremos como Ec. para mantener la consistencia de la nomenclatura.

Con el resultado de la Ec. (6.21) y reagrupando la Ec. (6.7) podemos conseguir una aproximación para calcular la cantidad de almacenamiento ocupado por el índice por medio de la Ec. (6.22).

$$\begin{aligned} TB^i &= TN + NB_i \cdot (TB_i - TN_i) \\ TB^i &\approx \frac{N}{f} + \frac{N}{f^i} \cdot \left(f^i \cdot nt \cdot p \cdot \left(\frac{f^{i-1}}{l_i} + \frac{1}{c} \right) + \frac{l_i}{2 \cdot c} - f^{i-1} \right) \\ TB^i &\approx N \cdot \left(nt \cdot p \cdot \left(\frac{f^{i-1}}{l_i} + \frac{1}{c} \right) + \frac{l_i}{2 \cdot c \cdot f^i} \right) \end{aligned} \tag{6.22}$$

Los términos $\frac{l_i}{2 \cdot c \cdot f^i}$ y $\frac{1}{c}$ se pueden eliminar de la Ec. (6.22) con lo que se consigue la Ec. (6.23) y que es una aproximación para TB^i.

$$TB^i \approx \frac{N \cdot nt \cdot p \cdot f^{i-1}}{l_i} \tag{6.23}$$

Dado que nuestro interés es analizar cómo afecta el nivel del R-tree al cual se asignan las bitácoras en la cantidad de almacenamiento de la estructura, supongamos que mantenemos constante todos los parámetros de la Ec. (6.23), excepto i. Bajo estas condiciones es claro que el valor obtenido por la Ec. (6.23) aumenta en la medida que el valor de i también ya que éste incide en el espacio que se necesita para almacenar los snapshots (notar que el nivel i al cual se asignan las bitácoras no influye en el almacenamiento requerido para almacenar los eventos). En todo caso es posible conseguir un almacenamiento similar asignando las bitácoras a los diferentes niveles del R-tree considerando un l_i proporcional a f^i.

6.4.2 Análisis del comportamiento de la eficiencia de las consultas

Una observación sobre el procesamiento de las consultas espacio-temporales basadas en nuestro enfoque es que la cantidad de nodos accesados del R-tree por la componente espacial de una consulta es el mismo, independientemente del nivel del R-tree al cual se asignan las bitácoras. Esto se puede apreciar en la Figura 6.5, donde se consideran 16 eventos (e_{ij} representa el evento del objeto j en el instante i). Notar que si bien en la figura se muestran la configuración de las bitácoras en los diferentes, al momento de elegir el nivel adecuado las bitácoras quedarán asignadas a ese nivel solamente y no repartidas en todos los niveles. En la medida que desplazamos las bitácoras desde la raíz hacia las hojas, el Rs-treee disminuye su altura en la misma cantidad que el Rp-tree la aumenta. Por ejemplo, en la Figura 6.5, (i) si asignamos las bitácoras al nivel 2, los Rs-trees tendrán altura 2 y el Rp-tree altura 1 y (ii) si asignamos las bitácoras al nivel 1, los Rs-trees tendrán altura 1 y el Rp-tree altura 2. Consideremos que ejecutamos la misma consulta de tipo *time-slice* en los escenarios (i) y (ii). En el escenario (i) debemos recorrer un Rp-tree de altura 1 para obtener la cantidad de bitácoras que hay que procesar y varios Rs-trees de altura 2 (1 por cada bitácora seleccionada) para obtener el conjunto de objetos iniciales (que posteriormente se actualiza con los eventos). En el escenario (ii) debemos accesar un Rp-tree más alto para determinar la cantidad de bitácoras, pero ahora los Rs-trees tienen altura 1. Notar que en ambos escenarios se accesan la misma cantidad de nodos del R-tree.

Teniendo en cuenta la observación descrita arriba, la eficiencia de las consultas depende principalmente del largo de la bitácora, es decir, del parámetro l_i. Esto lo podemos ver en la Figura 6.5, donde $l_3 = 16$, $l_2 = 8$ y $l_1 = 4$ y, por lo tanto, la cantidad de bloques de bitácora que

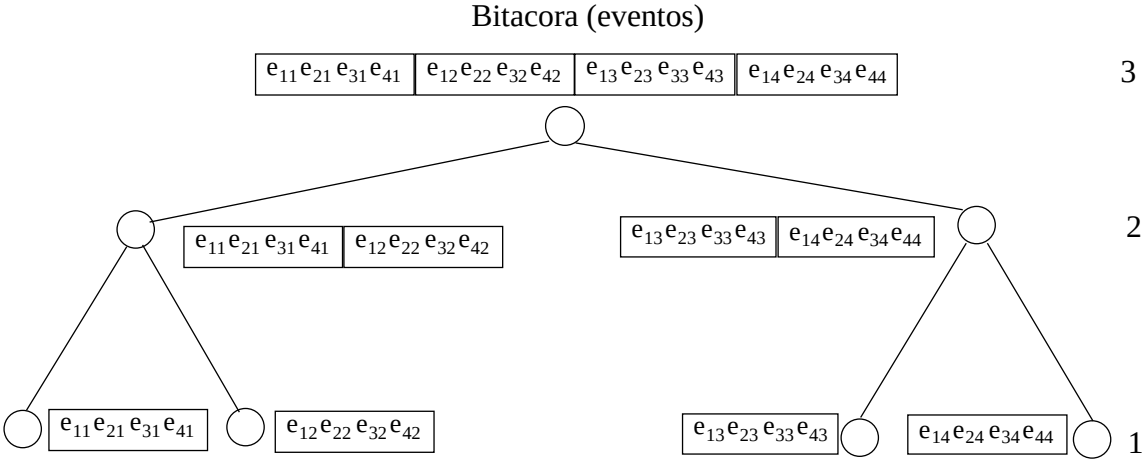


Figura 6.5: Un R-tree con las bitácoras en diferentes niveles.

hay que procesar es mayor si asignamos las bitácoras en el nivel 3 que si lo hacemos en el nivel 2. Si asumimos que tenemos que almacenar la misma cantidad de eventos, independientemente del nivel al cual se asignan las bitácoras, entonces el tamaño de l_i se puede obtener mediante la expresión de la Ec. (6.24), donde *il* representa la cantidad de instantes de tiempos cuyos eventos se almacenan en las bitácoras entre snapshots consecutivos (en el ejemplo de la Figura 6.5, $il = 4$). Con esto se logra que el almacenamiento es aproximadamente el mismo independientemente del nivel del R-tree al cual se asignan las bitácoras.

$$l_i = il \cdot p \cdot f^i \tag{6.24}$$

Recordemos que la cantidad de bitácoras que hay que procesar se obtiene con la Ec. (6.16). En este análisis se aproximó esta ecuación mediante la expresión de la Ec. (6.25), ya que D_i (Ec. (6.13)) es muy cercano a 1.

$$NL_i \approx cte \cdot q \cdot \sqrt{\frac{N}{f^i}} + \frac{q^2 \cdot N}{f^i} \tag{6.25}$$

Dado que en una consulta de tipo *time-slice*, la cantidad de bloques que se accesan del R-tree es la misma cualquiera sea el nivel al cual se asignan las bitácoras, y teniendo presente las Ecs. (6.24) y (6.25), la expresión de la Ec. (6.26) modela el efecto del nivel al cual se asignan las bitácoras sobre una consulta.

$$DA^i_{time-slice} \approx DA + \frac{il \cdot p \cdot f^i}{2 \cdot c} \cdot \left(cte \cdot q \cdot \sqrt{\frac{N}{f^i}} + \frac{q^2 \cdot N}{f^i} \right) \tag{6.26}$$

Es decir,

$$DA^i_{time-slice} \approx DA + \frac{cte \cdot q \cdot \sqrt{N \cdot f^i} \cdot p \cdot il}{2 \cdot c} + \frac{q^2 \cdot N \cdot il \cdot p}{2 \cdot c} \tag{6.27}$$

Al analizar la Ec. (6.27) podemos ver que la cantidad de bloques de bitácoras destinados a almacenar eventos se obtiene la expresión $\frac{cte \cdot q \cdot \sqrt{N \cdot f^i} \cdot p \cdot il}{2 \cdot c} + \frac{q^2 \cdot N \cdot il \cdot p}{2 \cdot c}$, donde sólo el primer término

depende de i. Este término, es decir, $\frac{cte \cdot q \cdot \sqrt{N \cdot f^i} \cdot p \cdot il}{2 \cdot c}$ es creciente en función de i. De esta forma, es claro que la cantidad de nodos accesados aumenta en la medida que i crece y, por lo tanto, se logra un mejor rendimiento de las consultas de tipo *time-slice* asignando las bitácoras en las hojas del R-tree.

De manera similar podemos obtener una aproximación para las consultas de tipo *time-interval*. Para ello obtenemos una aproximación de la Ec. (6.20) utilizando la expresión de la Ec. (6.27) para las consultas de tipo *time-slice*, la Ec. (6.25) y Ec.(6.1). Esta última se utiliza para transformar el término $\frac{p \cdot N_i}{NB_i}$ de la Ec. (6.20) en la expresión $p \cdot f^i$. Teniendo en cuenta que lo que queremos es conseguir una expresión aproximada que modele de manera más intuitiva el comportamiento de las consultas, el término $\left\lceil \frac{(ai-1) \cdot \frac{p \cdot N_i}{NB_i}}{c} \right\rceil$ de la Ec. (6.20) se transformó en la la expresión $p \cdot f^i \cdot ai$. Teniendo presente estas consideraciones, la Ec. (6.28) modela el comportamiento aproximado de las consultas de tipo *time-interval*.

$$DA^i_{time-interval} \approx DA^i_{time-slice} + \left(cte \cdot q \cdot \sqrt{\frac{N}{f^i}} + \frac{q^2 \cdot N}{f^i} \right) \cdot p \cdot ai \cdot f^i \tag{6.28}$$

Es decir,

$$DA^i_{time-interval} \approx DA^i_{time-slice} + cte \cdot p \cdot q \cdot ai \cdot \sqrt{N \cdot f^i} + q^2 \cdot N \cdot p \cdot ai \tag{6.29}$$

Al analizar la Ec. (6.29) podemos ver que el largo de la bitácora a recorrer aumenta en la medida que i también lo hace y, por lo tanto al igual que para las consultas de tipo *time-slice*, también se logra un mejor rendimiento de las consultas de tipo *time-interval* asignando las bitácoras al nivel 1 del R-tree.

Las Ec.s (6.23), (6.27), y (6.29) claramente muestran que el mejor rendimiento de nuestro enfoque, en términos de almacenamiento y eficiencia de las consultas, se logra ubicando las bitácoras en el nivel 1. Notar que la instancia del enfoque que considera las bitácoras en el nivel 1 corresponde al SEST$_L$.

Con el objeto de verificar el desempeño que predice el modelo aproximado del enfoque, se realizaron experimentos adicionales. Estos experimentos consideraron 20.000 objetos (puntos), los que siguieron una distribución uniforme en el espacio de trabajo y se desplazaron a lo largo de 200 instantes de tiempo. El porcentaje de movilidad de los objetos se estableció en 10%. Utilizando las ecuaciones exactas Ecs. (6.7) y (6.20) del modelo se obtuvo el almacenamiento y eficiencia de las consultas, respectivamente, para diferentes valores del largo de la bitácora (l_i). Los diferentes largos de l_i se definieron de tal manera que almacenen todos los cambios producidos en $1, 2, \cdots, k$ instantes de tiempo, con $k = 20$. De esta forma, los diferentes largos de bitácoras se obtuvieron con la ecuación $l_i = p \cdot f^i \cdot k$. Los resultados de estos experimentos se resumen en la Figura 6.6, donde se puede apreciar que para los diferentes largos de bitácora siempre es más conveniente, en términos de almacenamiento y eficiencia de las consultas, asignar las bitácoras al nivel 1 del R-tree.

6.5 Conclusiones

En este capítulo se presentó un modelo general para determinar el rendimiento, en términos de almacenamiento y eficiencia de las consultas, del enfoque basado en snapshots y eventos en los cuales se apoyan los métodos de acceso espacio-temporales SEST-Index y SEST$_L$.

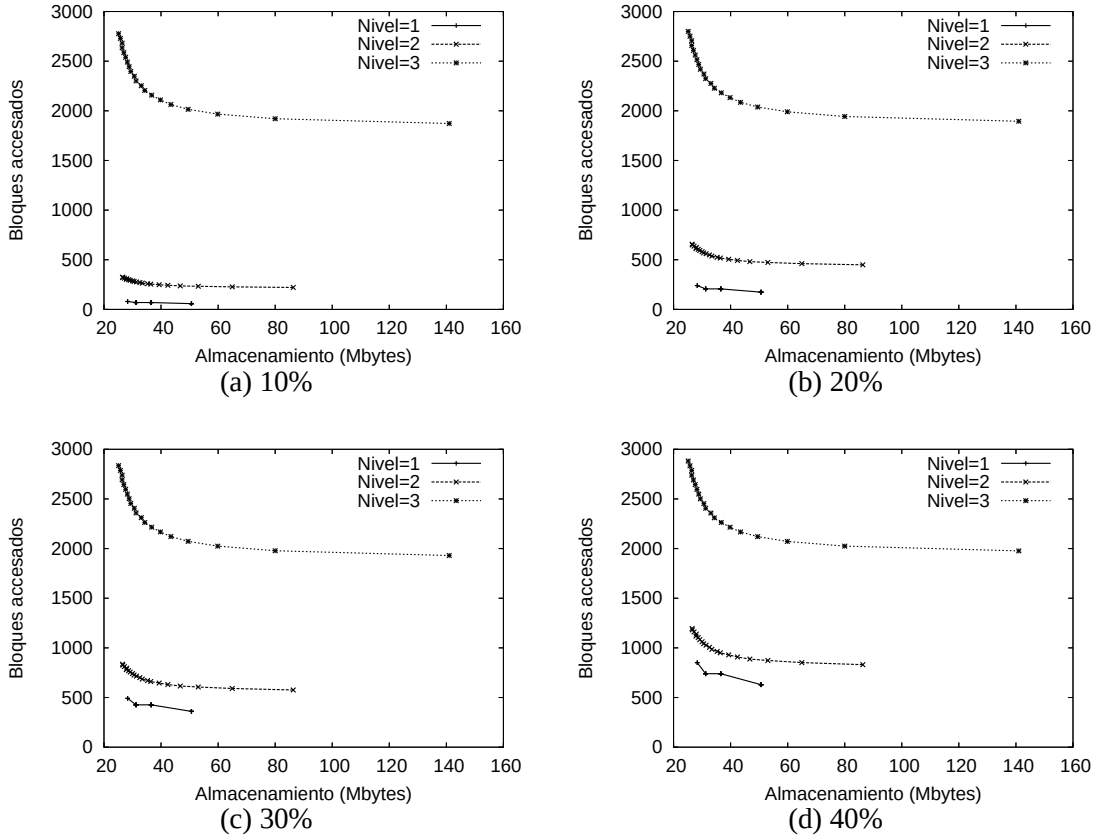


Figura 6.6: Trade-off almacenamiento/eficiencia de las consultas (área de la consulta formada por el 10%, 20%, 30% y 40% de cada dimensión y longitud del intervalo temporal igual a 20 unidades de tiempo).

A partir de los resultados de las pruebas realizadas con el modelo general podemos concluir que la asignación de las bitácoras en las hojas (SEST_L) es lo más conveniente, tanto desde el punto de vista del almacenamiento como de la eficiencia de las consultas. Esto se explica por que en la medida en que las bitácoras se asignan a subregiones mayores, mayor será el fenómeno de la duplicación –objetos que no se han modificado entre instantes de tiempo y que requieren ser almacenados nuevamente– lo que tiene un efecto negativo en el almacenamiento. Por otra parte, en la medida que las bitácoras se asignan a subregiones de mayor área, la cantidad de bloques que se requieren para almacenar los cambios también es mayor, lo que afecta el rendimiento de las consultas ya que será necesario procesar demasiados bloques. Esto último también se puede explicar pensando que en la medida en que las áreas de las subregiones es mayor, la capacidad de discriminación del proceso de selección de las bitácoras disminuye, debiéndose procesar muchos eventos irrelevantes para la consulta. Esto se nota más cuanto más selectiva es la consulta.

Parte III

Procesamiento de consultas espacio-temporales complejas con $\mathbf{SEST}_L$

Capítulo 7

Reunión espacio-temporal con el SEST$_L$

7.1 Introdución

La operación de reunión (join) es una de las más fundamentales y a la vez más costosa de evaluar por SABD relacionales [ME92]. En la medida que los SABDs modernos han incorporado nuevos tipos de datos (temporal, espacial, espacio-temporal entre otros), se ha hecho necesario contar con nuevos algoritmos para procesar la operación de reunión en estos dominios.

En el ámbito espacial una operación de reunión permite recuperar desde dos conjuntos de objetos aquellos pares que satisfacen un predicado espacial, siendo el más frecuente la intersección espacial (también conocida como solapamiento) [MP03, GG98]. Un ejemplo de reunión espacial es "*encontrar todas la ciudades cruzadas por un río*". Se han propuesto varios algoritmos para la reunión espacial. Algunos de ellos tienen en cuenta que los dos conjuntos disponen de un índice espacial (normalmente considerando un R-tree o algunas de sus variantes), como es el caso del [BKS93]. Otros algoritmos consideran que ninguno de los conjuntos cuenta con un índice. En este caso se particiona el espacio de manera regular o irregular y se distribuyen los objetos en las particiones, y, la reunión espacial se lleva a cabo por medio de algún esquema de hashing. Algunas propuestas en este sentido se pueden encontrar en [PD96] y [LR96], entre otros. Existen otros algoritmos que consideran que uno de los conjuntos tiene un índice y el otro no (por ejemplo, uno de los conjuntos es el resultado de una selección). Estos algoritmos construyen un índice para el conjunto que no lo tiene a partir del índice existente, y luego, aplican uno de los algoritmos propuestos para el caso en que ambos conjuntos disponen de índices espaciales; en [LR94] y [MP03] se discuten propuestas para estos casos.

En el dominio espacio-temporal la operación de reunión permite recuperar desde dos conjuntos espacio-temporales aquellos pares de objetos que cumplen un predicado espacial y temporal a la vez. Por ejemplo "*cuáles eran las ciudades de Chile que eran cruzadas por alguna carretera en la década de los noventa*".

A diferencia del dominio espacial, la reunión espacio-temporal no ha sido tan estudiada, a pesar de ser reconocida por la comunidad científica la necesidad de contar con algoritmos para esta operación [Kol00, TP01a, TP01b, SHPFT05, GS05]. Los métodos de acceso espacio-temporales propuestos hasta ahora están orientados a procesar principalmente consultas de tipo *time-slice* y de tipo *time-interval* y no proponen algoritmos para procesar la reunión sobre sus estructuras de datos. En [SHPFT05] se hace una evaluación sobre la conveniencia de la utilización de índices espacio-temporales en lugar de temporales y espaciales, separadamente, para resolver el problema

de la reunión. Las conclusiones de este trabajo aseguran que es más conveniente utilizar un índice espacio-temporal que un índice espacial junto a un índice temporal para evaluar una operación de reunión espacio-temporal. El único método de acceso espacio-temporal utilizado en la evaluación realizada en [SHPFT05] fue el 3D R-tree y, por lo tanto, las conclusiones del trabajo no contemplan comparaciones entre algoritmos de reunión espacio-temporal. Cabe destacar que al utilizar un 3D R-tree como método de acceso, el algoritmo de reunión espacio-temporal equivale a una reunión espacial en un espacio de tres dimensiones y, para tal operación, existen algoritmos muy eficientes como el propuesto en [BKS93] que resumimos en el Capítulo 2 y en el Alg. 2.12. Recordemos que el algoritmo supone que los dos conjuntos de objetos tienen disponible un índice espacial basado en un R-tree y que el tipo de reunión espacial corresponde a la denominada `MBR-Reunión-Espacial` en [BKS93]. Este tipo de reunión se utiliza en la etapa de filtrado del modelo de procesamiento de consultas espaciales descrito en la Figura 2.3. Por otra parte, es posible aplicar directamente los modelos de costo para estimar el rendimiento de la operación de reunión espacial definidos en [TSS98b] y que se resumen en la Sección 2.4.6.

En este capítulo se presenta un algoritmo para evaluar la operación de reunión espacio-temporal que está basado en nuestro método de acceso espacio-temporal SEST_L. En la Sección 7.2 se hace una descripción formal del problema y en la Sección 7.3 se describe nuestro algoritmo de reunión. En la Sección 7.4 realizamos una evaluación experimental del algoritmo y lo comparamos con el 3D R-tree. En la Sección 7.5 se describe un modelo de costo para estimar el rendimiento de nuestro algoritmo para procesar la operación de reunión y, finalmente, en la Sección 7.6 se comentan las conclusiones de este capítulo.

7.2 Definición del problema

En la literatura no existe una definición formal sobre el significado de la reunión espacio-temporal. Es así que para dos conjuntos de objetos espacio-temporales R y S y un predicado espacial θ, la reunión espacial podría significar:

i. Recuperar todos los pares de objetos que cumplen θ en un mismo instante. Esta opción podría entregar, además, el intervalo de tiempo de longitud máxima donde θ se cumple.

ii. Recuperar los pares de objetos que cumplen θ para un instante o intervalo de tiempo especificado en la consulta.

iii. Recuperar los pares de objetos que cumplen θ para un intervalo de tiempo y un rango espacial especificados en la consulta.

En este trabajo se adoptó el significado de la primera opción, ya que es el caso más general y corresponde a una extensión natural de la reunión espacial que considera el tiempo. Las restantes opciones se pueden entender como situaciones especiales del primer caso, las que son posibles de resolver combinando consultas de tipo *time-slice* o *time-interval* con operaciones de reunión. De esta forma, en este capítulo adoptamos la definición dada en la Sección 3.3 (Capítulo 3) para la operación de reunión espacio-temporal. Asi podemos responder consultas tales como: *"encontrar todos los autos que tuvieron una colisión"*. Suponiendo que el conjunto contiene todas las trayectorias de los autos, la consulta anterior implica buscar todos los autos cuyas trayectorias se intersectaron en el mismo instante de tiempo. Claramente, esta consulta se resuelve por medio

de una reunión espacio-temporal. Los algoritmos que a continuación se detallan suponen como predicado espacial la *intersección*. Sin embargo, es posible (sin pérdida de generalidad) precisar otros tipos de relaciones topológicas (por ejemplo, disjoint), de dirección (por ejemplo, Norte) o de distancia (absoluta o relativa). En este último caso, podemos especificar consultas como la siguiente: *"encontrar los autos que permanecieron a menos de una distancia n en un mismo instante o intervalo de tiempo"*.

7.3 Algoritmo de reunión espacio-temporal basado en el SEST$_L$ (RET)

Suponemos que tenemos dos conjuntos de objetos espacio-temporales S_1 y S_2, y que cada uno de ellos cuenta con un índice SEST$_L$, SI_1 y SI_2, respectivamente. Una forma directa y simple de implementar RET es ejecutar el Alg. 2.12 sobre los Rp-trees (R-trees podados) de los índices SI_1 y SI_2 cambiando la instrucción de la línea 6 por $ReunionBitacoras(E1.ref, E2.ref)$, tal como se muestra en el Alg. 7.1. Sin embargo, es muy probable que existan bitácoras de un índice (SI_i) que se intersectan con varias del otro índice (SI_j). De ocurrir lo anterior, y utilizando la primera idea para RET, será necesario leer todos los bloques de cada bitácora de SI_i tantas veces como bitácoras intersecte en SI_j. Una mejor solución, que permite planificar y optimizar los accesos a disco, consiste en mantener un conjunto E con todas las tuplas de la forma $\langle L_{SI_1}, L_{SI_2}\rangle$ con L_{SI_1} y L_{SI_2} bitácoras de SI_1 y SI_2, respectivamente, tal que $L_{SI_1}.MBR \cap L_{SI_2}.MBR \neq \emptyset$, donde $L_{SI_i}.MBR$ es la región o área asignada a la bitácora L_{SI_i} mediante la partición generada por el R-tree. De esta forma, en lugar de ejecutar directamente $ReunionBitacoras(E1.ref, E2.ref)$ se almacena en el conjunto E el elemento $\langle E1.ref, E2.ref\rangle$, es decir, se reemplaza la línea de 6 de Algoritmo 7.1 por la instrucción $Insertar(\langle E1.ref, E2.ref\rangle, E)$. El conjunto E se puede implementar mediante un arreglo o un esquema de hashing, el cual ocupa muy poco almacenamiento siendo posible mantenerlo completamente en memoria principal. Por ejemplo, si por cada conjunto S_1 y S_2 consideramos que en el instante inicial (t_0) existen 100.000 objetos distribuidos de manera uniforme en el espacio, el tamaño promedio de E no sobrepasa los 25 Kbytes.

El conjunto E modela un grafo bipartito, donde los vértices del grafo corresponden a las bitácoras y los arcos a los elementos de E, es decir, representan los pares de bitácoras que se deben reunir. En la Figura 7.1 se puede apreciar una instancia de E donde la bitácora b_1, del conjunto SI_1, se debe reunir con las bitácoras c_1 y c_3 del conjunto SI_2.

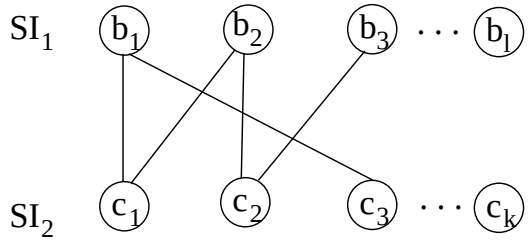


Figura 7.1: Grafo bipartito representando la reunión de bitácoras

Para procesar de manera óptima la reunión espacio-temporal, a partir de E, necesitamos

encontrar la secuencia de accesos a las bitácoras de tal manera que la cantidad de accesos a las páginas sea mínima utilizando dos buffers para almacenar bitácoras. Este problema de planificación ha sido estudiado en [MKY81, PI85, Val87] para bases de datos relacionales. En [MKY81] se demostró que el problema puede ser representado como un caso especial de un camino Hamiltoniano demostrando de esta forma que el problema es NP-completo cuando se consideran dos buffers para almacenar páginas de disco. A igual conclusión se llega en [NW97] para la reunión espacial. En nuestro caso, también podemos afirmar que el problema es NP-completo, pués la selección de los elementos de E se obtienen teniendo en cuenta solamente los Rp-trees podados de ambos conjuntos los que corresponden a índices espaciales de los correspondientes conjuntos S_1 y S_2. Por lo tanto, la única forma de abordar el problema de planificación descrito anteriormente es por medio de heurísticas.

Nosotros utilizamos una heurística simple para establecer un plan de acceso a las bitácoras. La heurística consiste en seleccionar la bitácora *b*, de cualquiera de los dos conjuntos, que se relaciona con el mayor número de bitácoras del conjunto restante. La bitácora *b* se lleva a memoria (ocupando un buffers) y se procede a realizar la reunión con todas las bitácoras con las cuales *b* se relaciona. Con esta heurística es posible reducir en un 30% la cantidad de bloques accesados con respecto a procesar cada par de bitácoras de manera inmediata, es decir, sin ninguna planificación. En el algoritmo Alg. 7.2 se describe la implementación de nuestra heurística para planificar la secuencia de acceso a las bitácoras involucradas en la reunión espacio-temporal.

```
 1: RE(R1, R2) {R1 y R2 corresponden a nodos de los Rp-trees}
 2: for all E1 in R1 do
 3:    for all E2 in R2 do
 4:       if E1.MBR ∩ E2.MBR ≠ ∅ then
 5:          if R1 y R2 son hojas then
 6:             ReunionBitacoras(E1.Oid, E2.Oid)
 7:          else if R1 es hoja then
 8:             ReadPage(E2.ref)
 9:             RE(R1, E2.ref)
10:          else if R2 es hoja then
11:             ReadPage(E1.ref)
12:             RE(E1.ref, E2.ref)
13:          else
14:             ReadPage(E1.ref)
15:             ReadPage(E2.ref)
16:             RE(E1.ref, R2)
17:          end if
18:       end if
19:    end for
20: end for
```

Alg. 7.1: Algoritmo para realizar reunión espacial de los Rp-trees.

7.3.1 Algoritmo para realizar la reunión espacio-temporal de dos bitácoras

Para llevar a cabo la reunión de dos bitácoras, se parte obteniendo dos conjuntos $T1$ y $T2$, uno por cada bitácora, con el procedimiento *ObtenerLista()* definido en el Alg. 7.4. Ambos conjuntos están formados por tuplas con la siguiente estructura $\langle Oid, Geom, t_i, t_f \rangle$, donde *Oid* es el identificador del objeto, *Geom* corresponde a la aproximación espacial (MBR) del objeto, t_i representa el momento

```
1:  RET(SI_1, SI_2) {SI_1 y SI_2 corresponden a índices espacio-temporales formados por el SEST_L}
2:  E = RE(SI_{1_{Rp-tree}}, SI_{2_{Rp-tree}})
3:  while E ≠ ∅ do
4:     L = Ranking(E)
5:     Leer todos los bloques de la bitácora L y ponerlos en un buffer B en memoria
6:     for cada tupla e = ⟨L1, L2⟩ ∈ E do
7:        if L = e.L1 then
8:           ReunionBitacoras(L, L2)
9:           Eliminar tupla e del conjunto E
10:       end if
11:       if L = e.L2 then
12:          ReunionBitacoras(L1, L)
13:          Eliminar tupla e del conjunto E
14:       end if
15:    end for
16: end while
```

Alg. 7.2: Algoritmo para procesar la reunión espacio-temporal con el SEST_L.

en que el objeto alcanzó la posición o forma *Geom*, y t_f el momento en que dejó la posición o forma *Geom*. De esta manera, cada entrada indica la posición alcanzada por un objeto, y el tiempo que permaneció en tal posición. Por ejemplo la tupla $\langle 3, \langle (3,4),(9,20)\rangle, 5, 18\rangle$, significa que el objeto cuyo *Oid* es 3, permaneció en la posición, cuya aproximación está definida por el MBR definido por los puntos $(3,4)$ y $(9,20)$, durante el intervalo de tiempo $[5,18)$.

Luego el algoritmo necesita obtener los pares de objetos $(o1,o2)$ tal que $o1 \in T1$ y $o2 \in T2$ y que se hayan intersectado tanto sus intervalos temporales como sus MBRs. Este último paso es posible llevarlo a cabo por medio del algoritmo plane sweep definido en [PS85, BO79], el que permite obtener los pares de segmentos que se intersectan en un espacio de una o dos dimensiones. Plane sweep reporta los K pares de intersecciones de N segmentos en tiempo $O((N+K)\log N)$ y requiere almacenamiento $O(N)$. El algoritmo consiste en aplicar plane sweep sobre los intervalos temporales y en el momento en que se descubre que dos intervalos (pertenecientes a conjuntos diferentes) se intersectan, se verifica si los MBRs también lo hacen. Un segundo algoritmo consiste en verificar primero si los MBRs se intersectan y posteriormente verificar si los intervalos temporales también lo hacen. La verificación de la intersección de los MBRs se puede realizar por medio del algorimo propuesto en [BKS93]. Dicho algoritmo considera dos secuencias de rectángulos, $R_{sec} = \langle r_1, r_2, \ldots r_n\rangle$ y $S_{sec} = \langle s_1, s_2, \ldots s_m\rangle$ ordenadas por la coordenada x del punto extremo inferior (x_l) que define a cada MBR. Posteriormente se mueve una línea de barrido al MBR, digamos t, en $R_{sec} \cup S_{sec}$ con el menor valor en x_l. Si el MBR t se encuentra en R_{sec}, entonces se recorre (desde el comienzo) secuencialmente la secuencia S_{sec} hasta alcanzar un MBR, digamos s_h, cuyo valor en x_l es mayor que el valor de la coordenada x del punto extremo superior (x_u) de t. En este punto se sabe que el segmento $\overline{t.x_l, t.x_u}$ se intersectan con los segmentos $\overline{s_j.x_l, s_j.x_u}$ con $1 \leq i \leq h$. Si el segmento $\overline{t.y_l, t.y_u}$ se intersecta con el segmento $\overline{s_j.x_l, s_j.x_u}$, entonces los MBRs t y s_j se intersectan. Si el MBR t se encuentra en S_{sec}, entonces R_{sec} se recorre de manera análoga a como se explicó para S_{sec}. A continuación el MBR t se marca como procesado. Luego la línea de barrido se desplaza al siguiente MBR en $R_{sec} \cup S_{sec}$ que no se encuentra marcado como procesado y cuyo $x.l$ es el menor y el paso descrito anteriormente se repite para todos los MBRs que no han sido marcados. El proceso continua hasta que todos los MBRs en R_{sec} y S_{sec} se han procesados. El algoritmo puede ser ejecutado en tiempo $O(\|R_{sec}\| + \|S_{sec}\| + k_X)$, donde k_X denota

el número de intersecciones de segmentos entre pares de segmentos cuyos puntos extremos están definidos por x_l y x_u de los MBRs de R_{sec} y S_{sec}.

El algoritmo que realiza la reunión espacio-temporal de dos bitácoras se describe en el Alg. 7.3. El procedimiento *IntersecciónObjetos(*$L1$,$L2$*)* se puede implementar considerando cualquiera de los dos algoritmos descritos anteriormente.

```
1: ReunionBitacoras(L1, L2) {L1 y L2 referencian a las bitácoras que se procesaran}
2: T1 = ObtenerLista(L1)
3: T2 = ObtenerLista(L2)
4: IntersecciónObjetos(T1, T2)
```

Alg. 7.3: Algoritmo para obtener los pares de objetos e intervalo de tiempo de dos bitácoras que se intersectan tanto temporal como espacialmente.

El procedimiento $ObtenerLista(L1)$ (ver Alg. 7.4), permite generar las tuplas con las diferentes posiciones y/o forma que los objetos (cuyos eventos se encuentran en la bitácora) han alcanzado. Para ello, se utiliza una tabla de hashing H, donde cada entrada o tupla tiene la estructura $\langle Oid, Geom, t\rangle$. El objetivo de H es mantener todos los objetos cuyo tiempo final t_f de permanencia en su última posición se encuentran indeterminados. H se inicializa con los objetos del primer snapshot de la bitácora (líneas 4 a 6 del Alg. 7.4). Cuando un objeto se desplaza a una posición (operación *move_in*) se inserta una tupla en H. Cuando el objeto se desplaza nuevamente (recordemos que en este caso se generan dos eventos u operaciones; un *move_out* seguido de un *move_in*), entonces la correspondiente tupla de H se recupera utilizando el *Oid* de la entrada de *move_out*. Con los datos de la tupla recuperada y los datos de *move_out* se forma una tupla con la estructura definida para los conjuntos $T1$ y $T2$, es decir, $\langle Oid, Geom, t_i, t_f\rangle$, la cual se inserta en el conjunto T. Posteriormente, se elimina la tupla con clave *Oid* de la tabla H.

El almacenamiento de H es proporcional al mayor fanout (f) de los Rp-trees. Por ejemplo, para bloques de 1 Kbyte, el valor de f es de 34, lo que implica un tamaño promedio de H inferior a 1 Kbyte. El tamaño de los conjuntos $T1$ y $T2$ es relativamente pequeño y se encuentra acotado por $O(f \cdot p \cdot nt \cdot bt)$, donde p es el porcentaje de movilidad, nt el número de instantes de tiempo almacenados en la base de datos, y bt la cantidad de bytes ocupados por una tupla de $T1$ ó $T2$. Por ejemplo, si consideramos $f = 34$, $nt = 200$, $p = 0.10$ y $bt = 28$, el tamaño de $T1$ ó $T2$ no supera los 20 Kbytes en promedio.

7.4 Evaluación Experimental

En esta sección se presenta una evaluación experimental en la cual el algoritmo (RET), se compara con el basado en el 3D R-trees [TVS96]. La razón de realizar la comparación con el 3D R-tree es que éste requiere de menos almacenamiento que el MV3R-tree [TP01b, TPZ02] y presenta un rendimiento levemente inferior para consultas de tipo *time-interval* tal como se muestra en [TP01b, TPZ02]. Además, existen algoritmos de reunión espacial con sus correspondientes modelos de costos bien establecidos que se basan en el 3D R-tree.

Nuestros experimentos consideraron conjuntos de 20.000 objetos de tipo punto, los cuales se distribuyeron uniformemente en el espacio. También se consideraron 5 valores de movilidad (2%, 4%, 6%, 8% y 10%) y 200 instantes de tiempo. Para ambos conjuntos se estableció el mismo dominio espacial y temporal. La unidad de medida utilizada correspondió a la cantidad de bloques

```
1:  ObtenerLista(L) {Permite obtener, a partir de una bitácora, las diferentes posiciones y períodos de tiempo de los
    objetos}
2:  H = ∅ { H es una tabla de hashing con clave Oid del objeto. Una entrada en esta tabla tiene la siguiente estructura:
    ⟨Oid, Geom, t⟩. }
3:  T = ∅ {T es un conjunto que almacenará tuplas con la siguiente estructura ⟨Oid, Geom, t_i, t_f⟩}
4:  for cada entrada e almacenado en el snapshot creado en t_0 do
5:     Insertar en H la tupla ⟨e.Oid, e.Geom, t_0⟩
6:  end for
7:  for cada evento c ∈ L do
8:     if c.Op = move_in then
9:        Insertar en H la tupla ⟨c.Oid, c.Geom, c.t⟩
10:    else
11:       Sea r = H(c.Oid) {r es una tupla, r.Oid = c.Oid}
12:       Insertar en T la tupla ⟨c.Oid, r.Geom, r.t, c.t⟩
13:       Eliminar de H la entrada con clave c.Oid.
14:    end if
15: end for
16: Sea t_e el último instante de tiempo almacenado en la base de datos.
17: for cada tupla r ∈ H do
18:    Insertar en T la tupla ⟨r.Oid, r.Geom, r.t, t_e⟩
19: end for
20: return T
```

Alg. 7.4: Algoritmo para obtener las diferentes posiciones alcanzadas por los objetos y los intervalos de tiempo en que se mantuvieron en cada posición.

de disco de 1.024 bytes accesados al ejecutar la operación de reunión.

El rendimiento de nuestro algoritmo (RET) se obtuvo directamente utilizando la implementación descrita en los Alg. 7.1 a Alg 7.4. En cambio el rendimiento del 3D R-tree se obtuvo mediante el modelo de costo para la reunión espacial usando R-tree definido en [TSS98b] y resumido en el Capítulo 2. El total de objetos (MBRs de tres dimensiones) a insertar en el 3D R-tree se consiguió con la ecuación $to = N \cdot p \cdot nt + (N - N \cdot p)$, donde N es el número inicial de objetos, p es la movilidad (%) y nt el total de instantes de tiempo almacenado en la base de datos. Por otra parte, la densidad inicial (D_0) de los to objetos se calculó por medio de la ecuación $D_0 = \frac{\sum_{i=1}^{to} MBR_i}{TA}$, donde TA es el área del espacio que contiene los to MBRs. Nuestra comparación consideró la existencia de un buffer para el algoritmo SJ (reunión espacial), es decir, los valores obtenidos corresponden a DA_{total} o DA'_{total} calculados con la Ec. (2.10) ó Ec. (2.13), respectivamente, del Capítulo 2. También se consideró una capacidad máxima de 36 entradas para un nodo de un 3D R-tree (dos puntos de tres coordenadas (x,y,t)) y de 50 para un nodo de un 2D R-tree (2 puntos de dos coordenadas (x,y)) con una capacidad promedio igual a 68% de la capacidad máxima para ambos tipos de nodos.

En la Figura 7.2 es posible observar que RET requiere solamente de un 20% del tiempo requerido por SJ para procesar la reunión espacio-temporal. El pobre rendimiento de SJ (3D R-tree) se debe a la gran cantidad de intersecciones producidas por los MBR de los R-trees ya que estos se encuentran en 3 dimensiones (en general, el R-tree disminuye su eficiencia en la medida que aumenta el número de dimensiones) y además, a la existencia de muchos objetos que permanecen sin moverse por intervalos de tiempo demasiado largos generando MBRs con áreas muy grandes. Para resolver este último problema se han propuesto soluciones que crean MBRs artificiales [HKTG02], es decir, en lugar de insertar un MBR de una determinada área, se

insertan varios MBRs (artificiales) de áreas más pequeñas disminuyendo de esta forma las áreas de intersecciones del 3D R-tree.

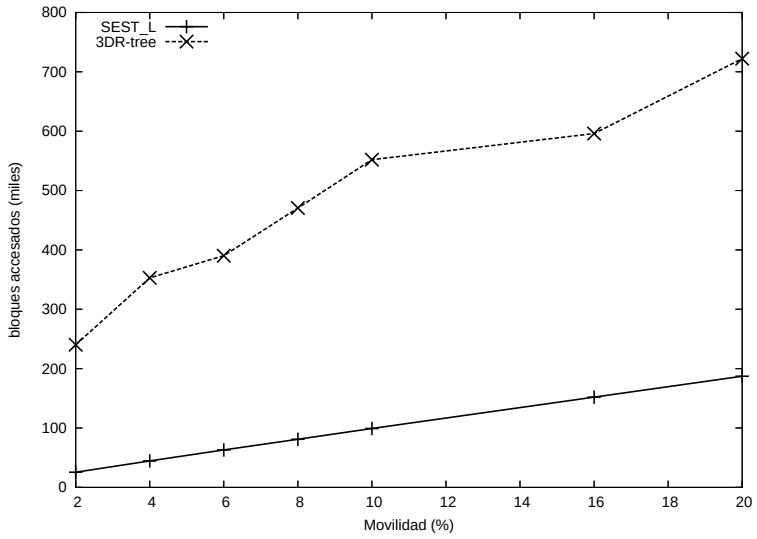


Figura 7.2: Bloques o nodos accesados por una operación de reunión espacio-temporal considerando diferentes porcentajes de movilidad y utilizando el 3D R-tree (SJ) y el SEST_L (RET).

También se estudió el efecto de la distancia de los desplazamientos (Δd) de los objetos en cada cambio de posición o evento. Se estudiaron 5 valores (porcentajes respecto de la longitud total de cada dimensión) que correspondieron a 2%, 4%, 6%, 8% y 10% y con una movilidad de 6%. Por ejemplo, al considerar $\Delta d = 8\%$ significa que los objetos pueden desplazarse como máximo un 8% de la longitud en cada dimensión espacial cada vez que lo hagan. La Figura 7.3 muestra los resultados del experimento. Podemos observar que en la medida que el valor para Δd aumenta, SJ (3D R-tree) se ve fuertemente afectado. Esto se explica nuevamente por el crecimiento de las áreas de intersección del 3D R-tree, la cual aumenta en la medida que el valor de Δd también lo hace. Sin embargo, el rendimiento de RET (SEST_L) permanece casi constante para los diferentes valores de Δd. La explicación de este comportamiento es que el espacio ocupado por las bitácoras sólo depende de la cantidad de cambios que hay que almacenar en cada instante de tiempo y no de las distancias alcanzadas por los desplazamientos de los objetos.

7.5 Modelo de costo para RET

En esta sección presentamos un modelo de costo para nuestro algoritmo de reunión espacio-temporal (RET) basado en el método de acceso SEST_L. El modelo permite predecir el costo (total de bloques/nodos accesados) por la operación de reunión. Con el objeto de evaluar la capacidad de estimación de nuestro modelo, lo comparamos con valores obtenidos por medio de experimentación. La Figura 7.4 describe las variables utilizado en la definición del modelo. Nuestro modelo se basa en los resultados presentados en la Sección 2.4.6 y que permiten predecir el rendimiento del R-tree para consultas WQ (window query) y para la reunión espacial. En la Sección 7.5.1 se define el modelo y en la Sección 7.5.2 mostramos su capacidad de predicción.

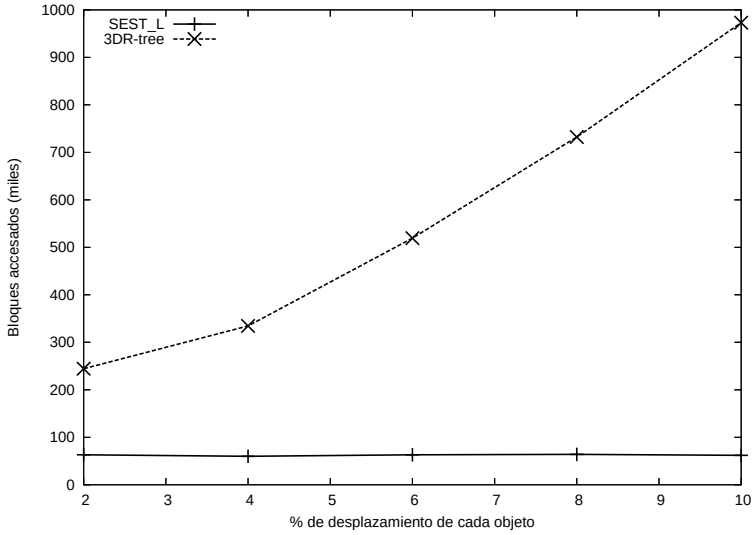


Figura 7.3: Bloques accesados por una operación de reunión espacio-temporal usando el 3D R-tree (SJ) y el SEST_L (RET) al considerar diferentes valores para Δd.

7.5.1 Definición del modelo de costo para RET

Podemos estimar el número promedio de bloques de disco (nodos) que el algoritmo RET necesita accesar por medio de la siguiente fórmula:

$$NA_{RET} = NA_{Rp-tree} + LP \tag{7.1}$$

En la Ec. (7.1) $NA_{RP-tree}$ representa el total de bloques accesados al realizar la operación de reunión espacial entre los R-trees de SI_1 y SI_2. El segundo término, LP, estima la cantidad de bloques accesados al procesar las bitácoras seleccionadas por la operación de reunión espacio-temporal. La cantidad de MBRs a partir de los cuales se construye el Rp-tree se obtiene con la Ec. (2.5) que equivale a la cantidad de bitácoras (NL_i) formadas por los N_i MBRs La densidad de estos NL_i objetos se puede obtener directamente con la Ec. (2.7), considerando $j = 1$ y D_0 como la densidad inicial de los N_i objetos. Sin embargo, la Ec. (2.7) no tiene en cuenta la evolución de los objetos a través del tiempo, la que incide en la densidad de los NL_i MBRs. Para capturar este efecto, se definió una nueva expresión para el parámetro c_i de la Ec. (2.7) y que denominamos nc_i (Ec. (7.2)).

$$nc_i = (p_i \cdot nt_i + 1) \cdot c \cdot M_i \tag{7.2}$$

El valor de nc_i se obtiene considerando un evento como una inserción de un nuevo objeto, sólo que realmente no se inserta en el nodo hoja si no que en su bitácora. Aunque la inserción se haga en las bitácora, sí se tiene en cuenta el efecto que el evento tiene sobre la densidad de los NL_i MBRs. Con nc_i podemos obtener el valor real de la densidad de los MBRs de las bitácoras (Ec. (8.8)) y, por lo tanto, calcular el valor de $NA_{RP-tree}$ utilizando las fórmulas definidas en Ec. (2.10) o Ec. (2.13).

$$D_{1_i} = \left(1 + \frac{\sqrt{D_{0_i}} - 1}{\sqrt{(nc_i \cdot M_i)}}\right)^2 \tag{7.3}$$

Símbolo	Descripción
N_i	Número de objetos iniciales del conjunto i
SI_i	Indice SEST_L del conjunto i
p_i	Movilidad del conjunto i
nt_i	Cantidad de instantes de tiempo almacenados en el índice i
M_i	Capacidad máxima del R-tree formado con los objetos del conjunto i
c_i	Número de cambios del conjunto i posible de almacenar en un bloque
nc_i	Capacidad promedio de los nodos hojas del conjunto espacio-temporal i considerando los objetos iniciales más los cambios
q_i	Longitud promedio en cada dirección de los MBRs de las bitácoras del conjunto i
tb_i	Promedio de bloques requeridos por una bitácora del conjunto i
NL_i	Promedio de bitácoras del conjunto i
LI_i	Cantidad de bitácoras del conjunto i que son accesados por cada MBR de las bitácoras del conjunto j, con $j \neq i$
D_{0_i}	Densidad inicial del conjunto i
D_{1_i}	Densidad de los MBRs de las bitácorass del conjunto i
NA_{RET}	Promedio de nodos o bloques accesados por la la operación de reunión espacio-temporal
$NA_{Rp-tree}$	Promedio de nodos accesados por la operación de reunión espacial considerando los árboles podados del SEST_L
LP	Promedio de nodos de cambios accesados al procesar las bitácoras

Figura 7.4: Definición de variables utilizadas por el modelo de costo del algoritmo RET.

Para obtener el valor de LP se necesita primero obtener la cantidad de pares de bitácoras que se requieren procesar. Para ello, se procede de la siguiente forma: por cada bitácora $l \in SI_1$ se estima la cantidad promedio de bitácoras que L intersectará en SI_2 (LI_2) (siendo los roles de SI_1 y SI_2 intercambiables). Para obtener el valor de LI_2 se considera la región (MBR) asignada a L como el predicado de una consulta espacial WQ (window query), la cual se ejecuta sobre el R-tree de SI_2. Para ello se necesita obtener la longitud promedio en cada dimensión del área de L, es decir, de los MBRs asignados a las bitácoras de SI_1. Dicha longitud se puede obtener mediante la Ec. (7.4) y Ec. (7.3), las que se pueden deducir a partir de la Ec. (2.6) y Ec. (2.7), respectivamente, y considerando $c = nc_i$ y $N = NL_i$.

$$q_i = \sqrt{\frac{D_{1_i}}{NL_i}} \tag{7.4}$$

Usando las Ecs. (7.3) y (7.4), es posible calcular la cantidad de bitácoras de SI_2 que, en promedio, cada bitácora de SI_1 intersectará (Ec. (7.5)).

$$LI_i = \left(\sqrt{D_{1_i}} + q_i \cdot \sqrt{\frac{N_i}{c \cdot M_i}} \right)^2 \tag{7.5}$$

El número promedio de bloques por bitácora para almacenar los eventos producidos a través del tiempo se puede calcular con la Ec. (7.6).

$$tb_i = \left(\frac{N_i \cdot p_i \cdot nt_i}{NL_i \cdot c_i} \right) \tag{7.6}$$

De esta forma, es posible expresar LP por medio de la Ec. (7.7).

$$LP = NL_1 \cdot (tb_1 + 1) + NL_1 \cdot LI_2 \cdot (tb_2 + 1). \tag{7.7}$$

En la Ec. (7.7), el valor 1 en la expresión $(tb_1 + 1)$ y $(tb_2 + 1)$ indica el primer snapshot que hay que procesar en cada bitácora.

Finalmente, el número promedio de nodos accesados por el algoritmo RET se resume en la Ec. (7.8).

$$NA_{stj} = NA_{Rp-tree} + NL_1 \cdot (tb_1 + 1) + NL_1 \cdot LI_2 \cdot (tb_2 + 1) \tag{7.8}$$

7.5.2 Evaluación del modelo

Con el objeto de comprobar la capacidad de predicción del modelo, se realizaron experimentos utilizando los mismos conjuntos de objetos y parámetros definidos en la Sección 7.4. Podemos observar en la Figura 7.5 que el modelo estima el rendimiento de una operación de reunión espacio-temporal basada en el SEST_L con un error relativo promedio de un 6%.

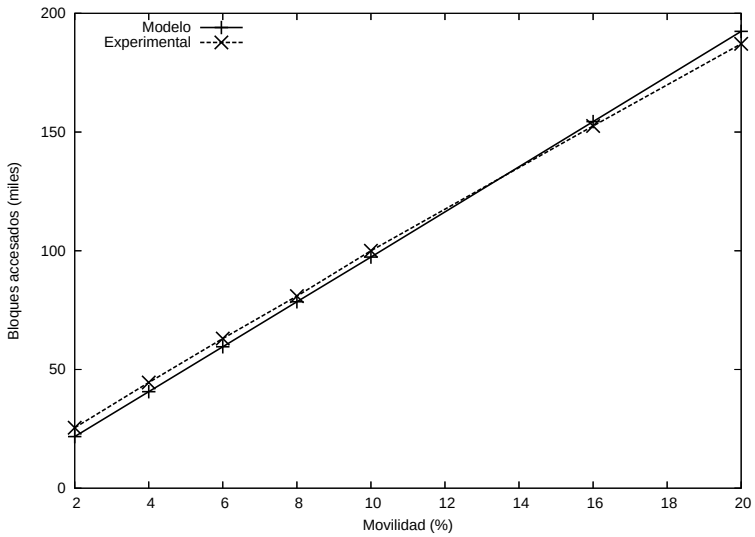


Figura 7.5: Capacidad de estimación del modelo de la eficiencia de RET.

7.6 Conclusiones

En este capítulo se presentó un algoritmo novedoso para procesar la operación de reunión espacio-temporal utilizando al SEST_L como estructura de datos subyacente. Se realizaron experimentos en los cuales se comparó nuestro algoritmo contra la reunión espacio-temporal llevada a cabo con un 3D R-tree. Los resultados experimentales muestran que nuestra propuesta requiere de sólo un 20% del tiempo consumido por el 3D R-tree. También se estableció un modelo de costo que mide la eficiencia de nuestro algoritmo el que presentó un error relativo promedio de un 6%.

Capítulo 8

Evaluación de consultas complejas con SEST$_L$

8.1 Introducción

Existen varias consultas espaciales, más allá de las que hemos estudiado, tales como encontrar los *k*-vecinos más cercanos a un objeto dado (*K*-NN) o encontrar los pares de vecinos más cercanos entre dos conjuntos de objetos espaciales. Estas consultas han sido muy estudiadas en el modelo espacial. Sin embargo, para éstas y otros tipos de consultas espaciales, no existen algoritmos para evaluarlas considerando objetos espacio-temporales. Del mismo modo en trabajos recientes [HKBT05, Sch05, ES02, dMR04, dMR05, dWNCdTdM06, SHPFT05] dentro del ámbito de las bases de datos espacio-temporales se proponen nuevos tipos de consultas de interés en áreas muy diversas y para las cuales se requiere de algoritmos eficientes para procesarlas. En este capítulo se describen estos nuevos tipos de consultas espacio-temporales y se analiza la factibilidad de evaluarlas utilizando nuestro método de acceso SEST$_L$ como estructura de datos subyacente. También se describe con detalle la implementación de un tipo de consulta, específicamente consultas sobre patrones espacio-temporales, por medio de un algoritmo basado en el SEST$_L$. Dicho algoritmo se comparó experimentalmente con el MVR-tree [TP01b] y el CellList [HKBT05] (un algoritmo ad-hoc para este tipo de consultas) mostrando un rendimiento muy superior al MVR-tree y levemente inferior al CellList.

Este capítulo se organiza de la siguiente manera. En la Sección 8.2 se describen varios tipos de consultas espacio-temporales. Se empieza describiendo consultas espaciales del vecino más cercano y de los pares de vecinos más cercanos formalizando su extensión al ámbito espacio-temporal, delineando algoritmos que pueden procesar tales consultas con el SEST$_L$. Luego se analizan consultas sobre patrones espacio-temporales y se exploran extensiones de éstas para permitir especificar patrones más complejos. Junto a lo anterior se discuten algoritmos para evaluar casos especiales de este tipo de consultas con el proósito de mostrar la factibilidad de procesarlas con el SEST$_L$. En la Sección 8.3 se presentan los detalles de la implementación de un algoritmo para evaluar consultas sobre patrones espacio-temporales y basado en nuestro método de acceso SEST$_L$. También se discuten resultados experimentales que comparan nuestro algoritmo con el MVR-tree y el CellList, y un modelo de costo que predice su rendimiento. En la Sección 8.4 se presentan las conclusiones de este capítulo.

8.2 Nuevos tipos de consultas espacio-temporales

En esta sección se describen un conjunto de consultas espacio-temporales, las cuales han sido seleccionadas en base a su relevancia en la literatura relacionada y porque se consideran fundamentales para implementar otros tipos de consultas. Para cada una de ellas se analiza la factibilidad de implementarlas por medio del SEST_L.

8.2.1 Evaluación del vecino más cercano con el SEST_L

Un tipo de consulta muy frecuente sobre un conjunto de objetos espaciales es encontrar los K vecinos más cercanos (*K-NN*) a un punto dado en el espacio. El procesamiento de este tipo de consultas requiere de algoritmos muy diferentes a los utilizados para evaluar consultas de tipo WQ (window query) [RKV95]. En [RKV95] se propone un algoritmo para determinar *K-NN* que supone que los objetos se encuentran almacenados en un R-tree. En esta sección extendemos dicho algoritmo para permitir encontrar los *K-NN* en un conjunto de objetos espacio-temporales almacenados en un índice espacio-temporal de tipo SEST_L. En nuestro contexto suponemos que las consultas del vecino más cercano tienen la forma *K-NN(P, t)*, con P el punto y t un instante de tiempo. Esta definición corresponde a la que diéramos en el Capítulo 3. Si bien en esta sección se esboza una solución para el problema particular *1-NN(P, t)*, la extensión para solucionar el caso más general *K-NN(P,T)*, con $k > 1$ y T un intervalo de tiempo, es factible y sencilla de realizar con el SEST_L.

8.2.1.1 *K-NN* en bases de datos espaciales

En esta sección se resumen tanto las métricas como el algoritmo propiamente tal para resolver *K-NN* que se propone en [RKV95]. El algoritmo usa la estrategia de ramificación y poda y supone que los objetos espaciales se encuentran almacenados en un R-tree. Para podar el R-tree se usan dos métricas MINDIST y MINMAXDIST, la cuales se explican a continuación.

Métricas. Dado un punto P y un objeto O contenido en su MBR se cuenta con dos métricas para el algoritmo *K-NN*. La primera está basada en la mínima distancia (MINDIST) a O desde P. La segunda métrica está basada en la mínima de las máximas distancias (MINMAXDIST) de P a los vértices del MBR que contiene a O. MINDIST y MINMAXDIST corresponden a una cota inferior y superior, respectivamente, de la distancia real de P a O.

MINDIST Supongamos un rectángulo R en un espacio n–dimensional definido por dos puntos $S(s_1, s_2, \ldots, s_n)$ y $T(t_1, t_2, \ldots, t_n)$, con $s_i \leq t_i$ para $1 \leq i \leq n$ y un punto $P(p_1, p_2, \ldots, p_n)$ en el mismo espacio. Si P está dentro de R, entonces MINDIST es cero. Si P se encuentra fuera de R, entonces MINDIST se define como el cuadrado de la distancia Euclidiana entre P y el lado más cercano del rectángulo R. La Ec. (8.1) define MINDIST.

$$MINDIST(P,R) = \sum_{i=1}^{n} |p_i - r_i|^2 \tag{8.1}$$

donde

$$r_i = \begin{cases} s_i & \text{si } p_i < s_i \\ t_i & \text{si } p_i > t_i \\ p_i & \text{en otro caso} \end{cases}$$

La distancia mínima (también definida como el cuadrado de la distancia Euclidiana) de un punto P a un objeto espacial o, denotada por $\|(P,o)\|$ es:

$$\|(P,o)\| = \min\left(\sum_{i=1}^{n} |p_i - x_i|^2, \forall X = (x_1, x_2, \ldots, x_n) \in o\right) \tag{8.2}$$

Teorema 8.2.1. *Dado un punto P y un MBR R el cual encierra un conjunto de objetos* $O = \{o_i, 1 \leq i \leq m\}$*, la siguiente desigualdad es siempre verdadera.*

$$\forall o \in O, \mathit{MINDIST}(P,R) \leq \|(P,o)\|.$$

MINDIST se usa para determinar el objeto más cercano a P de todos los contenidos en R. La igualdad en el teorema anterior se produce cuando un objeto de R toca el círculo con centro en P y radio la raíz cuadrada de MINDIST. Cuando se recorre el R-tree para encontrar el *NN* al punto P, en cada nodo visitado debemos decidir qué MBR seguir primero. Esta decisión la podemos tomar usando MINDIST (se elige el que tiene menor MINDIST). Sin embargo, es posible que en muchos casos esta estrategia nos lleve a visitar nodos innecesarios del R-tree. Esto lo podemos apreciar en la Figura 8.1, donde la estrategia elige equivocadamente el MBR B como el más promisorio.

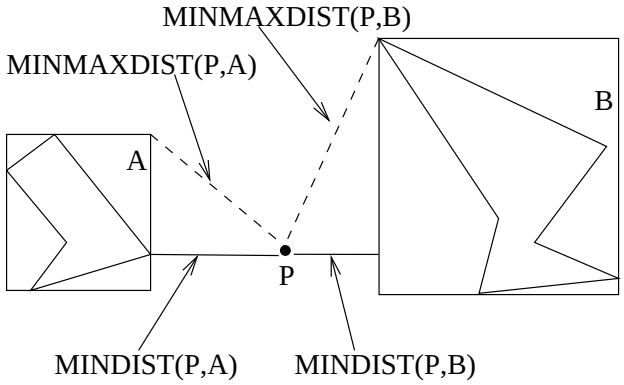


Figura 8.1: Dos MBRs y sus correspondientes MINDIST y MINMAXDIST al punto P.

Con el propósito de evitar visitas infructuosas de nodos se define MINMAXDIST. Antes de definir MINMAXDIST, es necesario observar la siguiente propiedad de un MBR: cada lado de cualquier MBR (en cualquier nivel del R-tree) contiene al menos un punto de algún objeto en la base de datos. Esta propiedad la podemos ver en la Figura 8.2 donde los MBRs A y B se encuentran en un nivel más profundo en el R-tree que el nodo C, sin embargo, cada lado de A, B y C tocan un objeto de la base de datos.

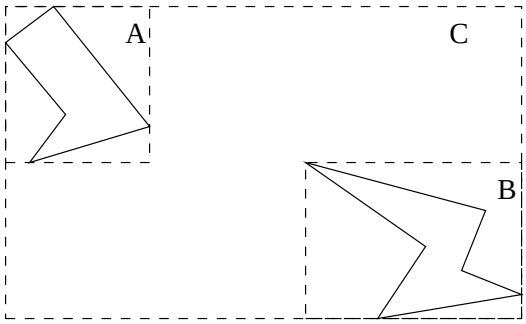


Figura 8.2: Propiedad de los objetos espaciales y sus MBRs.

MINMAXDIST El objetivo de esta métrica es evitar visitas innecesarias de MBRs de tal manera de podar adecuadamente el R-tree. MINMAXDIST es la mínima distancia (considerada como el cuadrado de la distancia Euclidiana) que garantiza que un objeto puede ser encontrado como parte de la solución. En otras palabras es la mínima distancia entre P y el punto más lejano del segmento o hiperplano más cercano R (ver Figura 8.1). MINMAXDIST garantiza que existe un objeto dentro del MBR a una distancia menor o igual al valor de MINMAXDIST, el cual se obtiene con la siguiente ecuación:

$$MINMAXDIST(P,R) = \min_{1\leq k\leq n}(|p_k - rm_k|^2 + \sum_{i\neq k, 1\leq i\leq n}|p_i - rM_i|^2), \tag{8.3}$$

donde

$$rm_k = \begin{cases} s_k & \text{si } p_k \leq \frac{s_k+t_k}{2} \\ t_k & \text{en otro caso} \end{cases}$$

y

$$rM_i = \begin{cases} s_i & \text{si } p_i \geq \frac{s_i+t_i}{2} \\ t_i & \text{en otro caso} \end{cases}$$

Teorema 8.2.2. *Dado un punto P y un MBR R el cual contiene a un conjunto de objetos* $O = \{o_i, 1 \leq i \leq m\}$*, la siguiente propiedad siempre se mantiene.*

$$\exists o \in O, \|(P,o)\| \leq MINMAXDIST(P,R)$$

El teorema anterior dice que MINMAXDIST es la distancia mínima que garantiza la presencia de un objeto O en R cuya distancia desde P se encuentra dentro de esta distancia.

Los dos teoremas anteriores se pueden utilizar para definir las siguientes estrategias de poda durante la búsqueda en el R-tree:

a) Un MBR M con $MINDIST(P,M)$ mayor que $MINMAXDIST(P,M')$ de otro MBR M' se puede descartar, ya que no contendrá la respuesta.

b) Un objeto O cuya distancia a P es mayor que $MINMAXDIST(P,M)$ para un MBR M se puede descartar ya que M contiene un objeto O' el cual es más cercano a P.

c) Cada MBR M con $MINDIST(P,M)$ mayor que la distancia desde P a un objeto O dado se puede descartar.

El algoritmo para *K-NN* propuesto en [RKV95] utiliza las estrategias definidas anteriormente para descartar ramas o subárboles en el recorrido del R-tree. El algoritmo recorre el R-tree en profundidad partiendo desde la raíz del árbol. Inicialmente el algoritmo supone que la distancia (*dist*) entre P y el vecino más cercano es infinita, es decir, $dist = \infty$. Durante el recorrido del árbol y para cada nodo interno visitado, se calcula MINDIST para cada uno de los MBRs del nodo. Las entradas del nodo se ordenan de acuerdo a MINDIST en una lista denominada ABL (Active Branch List). Luego se aplican las estragias de poda a y b sobre la lista ABL con el propósito de remover entradas que contienen subárboles que no es necesario recorrer. El algoritmo itera sobre la lista ABL hasta que esta se encuentre vacía. En cada iteración, el algoritmo selecciona el siguiente subárbol recorriendolo recursivamente. En los nodos hojas, para cada objeto se calcula la distancia con una función de distancia específica (que depende del tipo de objeto espacial almacenado en la base de datos) a P, sea $D = \{d_1, d_2, \ldots, d_h\}$ el conjunto de todas estas distancias. Luego se actualiza $dist = \min\{D \cup \{dist\}\}$ y el vecino más cercano (*NN*). Al retornar de la recursividad, se utiliza el valor de *dist* para podar el R-tree utilizando la regla c, es decir, se remueven todas entradas en ABL con MINDIST$(P,M) > dist$, donde M representa el MBR de la entrada.

El algoritmo anterior se puede extender fácilmente para obtener los K vecinos más cercano para cualquier $K \geq 0$. Las modificaciones que hay que realizar son las siguientes:

- Mantener en un arreglo (ordenado por la distancias a P a cada uno de los K vecinos más cercanos), los k vecinos cercanos más recientes.
- Para podar el R-tree se utiliza como distancia (*dist*) la mayor de las distancias entre P y los vecinos más cercanos almacenados en el arreglo.

8.2.1.2 El vecino más cercano con el SEST$_L$

Para implementar *K-NN(P,t)* con el SEST$_L$ podemos proceder como sigue:

a) Seleccionar las bitácoras utilizando el algoritmo definido en la sección anterior y considerando el R-tree podado (Rp-tree) del SEST$_L$.

b) Por cada bitácora seleccionada, obtenemos los objetos vigentes o "vivos" en el instante de tiempo dado en la consulta y actualizamos el *NN*.

La primera parte del procedimiento anterior se resuelve aplicando el algoritmo *K-NN(P)* sobre el Rp-tree pero con las modificaciones que se detallan a continuación. En primer lugar la propiedad sobre la cual se basa la métrica MINMAXDIST no siempre se cumple en el SEST$_L$, debido a que el Rp-tree debe contener todos los instantes de tiempo lo que provoca que no se pueda garantizar que exista un objeto en cada lado del MBR. Este fenómeno lo podemos ver en la Figura 8.3 donde se muestran dos estados de un MBR y los objetos que contiene. En el instante t_i se puede apreciar que se cumple la propiedad para el MBR R, pues cada lado de R contiene un objeto. En cambio, en el instante t_j la propiedad no se cumple, ya que el objeto q se movió a otro lugar y, por lo tanto, el lado que lo contenía queda sin ningún objeto. Esta propiedad del SEST$_L$ imposibilita que se pueda usar MINMAXDIST y, por lo tanto, las estrategias a y b no se pueden utilizar.

La segunda parte del procedimiento (para las bitácoras) se resuelve obteniendo en primer lugar los objetos vigentes en el instante t. Luego se utilizan estos objetos para actualizar *NN* y el valor de la distancia. El procedimiento se describe en detalle en el Alg. 8.1.

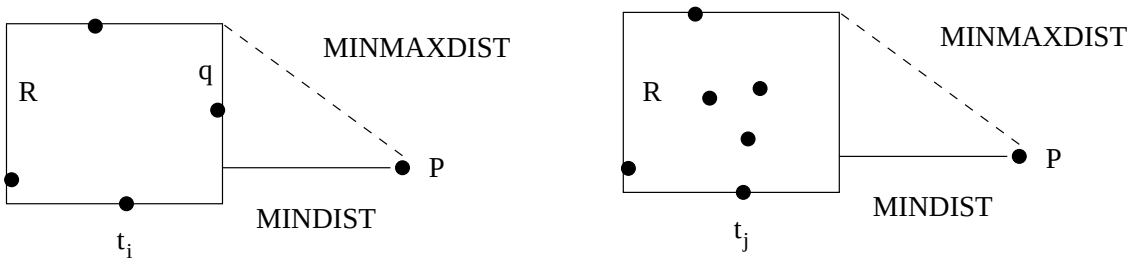


Figura 8.3: Dos instantes de tiempo que muestran cómo el teorema 8.2.2 no se cumple en el SEST_L.

```
 1: 1-NN(Nodo L, Objeto P, Tiempo t, Distancia d, Punto NN) {inicialmente d = ∞, NN es una variable global}
 2: if L es una bitácora then
 3:    Sea tr el instante de tiempo del último snapshot S en L tal que tr ≤ t
 4:    Sea SS = S
 5:    tr = next(tr) {next(tr), retorna el instante de tiempo siguiente a tr almacenado en la bitácora}
 6:    Actualizar el conjunto SS con todos los eventos de L producidos en el intervalo [tr,t]
 7:    for cada Objeto o ∈ SS do
 8:       nd = Distancia(P, o.Geom)
 9:       if nd < d then
10:          d = nd
11:          NN = o.Oid
12:       end if
13:    end for
14: else
15:    {Nodos internos del Rp-tree}
16:    Seleccionar el nodo siguiente a visitar utilizando la estrategia c de descarte definida para K-NN(P)
17: end if
```

Alg. 8.1: Algoritmo para encontrar el vecino más cercano a un objeto dado utilizando el SEST_L.

El algoritmo 8.1 se puede extender a un intervalo de tiempo T (*K-NN(P,T)*). Entenderemos la semántica de *K-NN(P,T)* como encontrar los K objetos que estuvieron más cercanos a P en algún instante de tiempo dentro del intervalo T. Para implementar *K-NN(P,T)* se requiere iterar los pasos 6 al 13 del Alg. 8.1 por cada instante $t \in T$ almacenado en la base de datos y asignando adecuadamente los valores a t y tr.

8.2.2 Los pares de vecinos más cercanos

En [CMTV00, Cor02] se discute una definición del par de vecinos más cercano (K closest pairs query, *K-CPQ*) considerando objetos de tipo punto. Sea $P = \{p_1, p_2, \ldots, p_n\}$ y $Q = \{q_1, q_2, \ldots, q_m\}$ dos conjuntos de objetos de tipo punto que se almacenan en dos R-trees, R_P y R_Q, respectivamente. El par de vecinos más cercano de los dos conjuntos corresponde al par

$$(p_z, q_l), p_z \in P \wedge q_l \in Q,$$

tal que

$$dist(p_i, q_j) \geq dist(p_z, q_l), \forall p_i \in P \wedge \forall q_j \in Q.$$

En otras palabras, el par de vecinos más cercanos de P y Q es el par que tiene la menor distancia entre todos los pares de objetos que pueden ser formados eligiendo un objeto de P y otro de Q.

Los algoritmos existentes en el ámbito espacial para resolver el problema propuesto en [CMTV00, Cor02] extienden las métricas definidas para *K-NN*, considerando que los objetos sobre los cuales se aplican son rectángulos (MBRs). Tales métricas se pueden observar en la Figura 8.4.

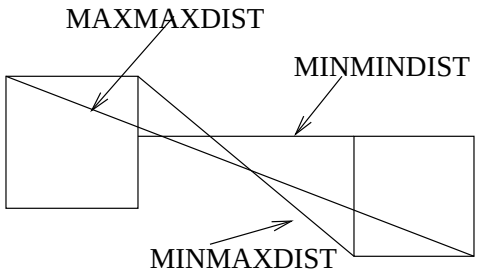


Figura 8.4: Dos MBRs y las métricas definidas entre ellos.

Las definiciones de estas nuevas métricas [CMTV00, Cor02] son las siguientes. Sean N_P y N_Q dos nodos internos de los R-trees R_P y R_Q, respectivamente. El MBR de cada nodo contiene los puntos que se encuentran dentro de su subárbol. Por otra parte, al menos un objeto está ubicado en cada lado del rectángulo. Sean M_P y M_Q los MBRs de N_P y N_Q, respectivamente. Sean r_1, r_2, r_3 y r_4 los cuatro lados de M_P, y s_1, s_2, s_3 y s_4 los cuatro lados de M_Q. Se define MINDIST(r_i, s_i) como la distancia mínima entre dos puntos que caen sobre r_i y s_i. De la misma manera se define MAXDIST(r_i, s_i) como la máxima distancia entre dos puntos que caen sobre r_i y s_i. Basándose en MINDIST y MAXDIST, las métricas de la Figura 8.4 se definen de la siguiente manera:

$$MINMINDIST(M_P, M_Q) = \min_{i,j}\{MINDIST(r_i, s_j)\}.$$

En el caso de que los MBRs se intersecten, MINMINDIST(M_P, M_q) es cero. Las siguientes dos métricas, sin embargo, se pueden definir tanto sea que los MBRs se intersectan o no.

$$MINMAXDIST(M_P, M_Q) = \min_{i,j}\{MAXDIST(r_i, s_j)\}$$

y

$$MAXMAXDIST(M_P, M_Q) = \max_{i,j}\{MAXDIST(r_i, s_j)\}$$

Para cada par de puntos (p_i, q_j), $p_i \in M_P$ y $q_j \in M_Q$, se cumple la siguiente relación:

$$\begin{aligned} &MINMINDIST(M_P, M_Q) \leq dist(p_i, q_j) \\ &\leq MAXMAXDIST(M_P, M_Q) \end{aligned} \tag{8.4}$$

Además, existe al menos un par de puntos (p_i, q_j), con $p_i \in M_P$ y $q_j \in M_Q$, tal que

$$dist(p_i, q_j) \leq MINMAXDIST(M_P, M_Q). \tag{8.5}$$

Uno de los algoritmos propuestos en [CMTV00, Cor02] (ver Alg. 8.2) es el descrito a continuación, el cual considera encontrar el par más cercano entre dos conjuntos de objetos espaciales indexados por un R-tree de la misma altura.

```
 1: 1-CPQ(R_P, R_Q, D) {D es la mínima distancia, inicialmente D = ∞ }
 2: if R_P y R_Q son nodos internos then
 3:    m = min_{i,j}{MINMAXDIST(E_i.MBR ∈ R_P, E_j.MBR ∈ R_Q)} {E_i y E_j corresponden a cada una de las entradas en
       los nodos R_P y R_Q, respectivamente}
 4:    if m ≤ D then
 5:       D = m
 6:    end if
 7:    for cada entrada E ∈ R_P do
 8:       for cada entrada F ∈ R_Q do
 9:          if MINMINDIST(E.MBR, F.MBR) < D then
10:             1-CPQ(E.ref, F.ref, D) {E.ref y F.ref corresponden a las referencias de los nodos hijos}
11:          end if
12:       end for
13:    end for
14: else
15:    {nodos hojas}
16:    for cada entrada E ∈ R_P do
17:       for cada entrada F ∈ R_Q do
18:          m = dist(E.p, F.p) {E.p y F.p corresponden a los puntos almacenados en las hojas de los R-trees }
19:          if m < D then
20:             D = m
21:             cpn = ⟨E.oid, F.oid⟩ {cpn es una variable que mantiene el par de vecinos más cercanos. E.oid y F.oid
                corresponden a los identificadores de los objetos (puntos)}
22:          end if
23:       end for
24:    end for
25: end if
26: return cpn
```

Alg. 8.2: algoritmo para encontrar el par de vecinos más cercanos entre dos conjuntos de objetos espaciales indexados por un R-tree.

A partir de la introducción de modificaciones en los pasos 7–13 del Alg. 8.2, se proponen variantes que mejoran la idea original. Por ejemplo, una forma de mejorar el rendimiento del algoritmo, es ordenando los pares de MBRs de los nodos de manera ascendente por MINMINDIST y siguiendo este orden se desciende por los R-trees. También se extiende para árboles de diferentes alturas y para obtener los K pares de vecinos más cercanos (*K-CPQ*) para $K > 1$.

8.2.2.1 El par de vecinos más cercanos con el SEST$_L$

Al igual que para el caso del vecino más cercano, se necesita definir el problema en el contexto de las bases de datos espacio-temporales. En este caso, se considerará *K-CPQ(R_P, R_Q, t)* como los objetos que se encuentran más cercanos en el instante t.

El paso 3 del Alg. 8.2 utiliza MINMAXDIST para descartar. Podemos observar que primero se obtiene el mínimo valor de MINMAXDIST con el cual se actualiza *D*, el que posteriormente se utiliza para decidir los subárboles a seguir. Lamentablemente esta estrategia no es posible aplicarla utilizando el SEST$_L$ ya que, por la misma razón dada para *K-NN(P,t)*, pueden existir MBRs que al momento de la consulta se encuentren vacios producto de los eventos en las bitácoras. En

otras palabras, puede suceder que el valor mínimo de MINMAXDIST ocurra entre pares de MBRs que se encuentren vacios en el instante de tiempo dado en la consulta. Si además este valor se asigna a D y el par es el único que cumple la condición MINMINDIST $\leq D$, entonces se cambia equivocadamente el valor de D y por lo tanto el algoritmo falla. De esta forma un procedimiento para buscar el par de vecinos más cercanos sólo puede utilizar MINMINDIST para descartar (ver Alg. 8.3).

```
 1: 1-CPQ(R_P, R_Q, D, t) {D es la mínima distancia, inicialmente D = ∞ }
 2: if R_P y R_Q son nodos internos then
 3:    for cada entrada E ∈ R_P do
 4:       for cada entrada F ∈ R_Q do
 5:          if MINMINDIST(E.MBR, F.MBR) < D then
 6:             1-CPQ(E.ref, F.ref, D) {E.ref y F.ref corresponden a las referencias de los nodos hijos}
 7:          end if
 8:       end for
 9:    end for
10: else
11:    {nodos hojas (bitácoras)}
12:    Sea SS1 los objetos vigentes en el instante t en la bitácora R_P
13:    Sea SS2 los objetos vigentes en el instante t en la bitácora R_Q
14:    for cada entrada E ∈ SS1 do
15:       for cada entrada F ∈ SS do
16:          m = dist(E.p, F.p) {E.p y F.p corresponden a los puntos almacenados los conjuntos SS1 y SS2
             respectivamente }
17:          if m < D then
18:             D = m
19:             cpn = ⟨E.oid, F.oid⟩
20:          end if
21:       end for
22:    end for
23: end if
24: return cpn
```

Alg. 8.3: algoritmo para encontrar los pares de vecinos más cercanos entre dos conjuntos de objetos espacio-temporales indexados por un SEST$_L$.

El algoritmo 8.3 se puede extender a un intervalo de tiempo T y permitir evaluar consultas del tipo *K-CPQ(P,T)*. De manera similar a la semántica de *K-NN(P,T)*, nosotros definimos la de una consulta *K-CPQ(P,T)* como encontrar los K pares de objetos que estuvieron más cercanos en algún instante de tiempo dentro del intervalo T. Las modificaciones principales se concentran en las líneas 12 y 13 del Alg. 8.3 y que consisten en ir generando coordinadamente los conjuntos $SS1$ y $SS2$ para todos los instantes $t \in T$ que se encuentran almacenados en las bitácoras R_P y R_Q. Inicialmente los conjuntos $SS1$ y $SS2$ mantienen los objetos vigentes de las correspondientes bitácoras en el instante inicial del intervalo T. Luego se ejecutan las pasos 14 al 22 del Alg. 8.3. A continuación se actualizan los conjuntos $SS1$ y $SS2$ con los objetos vigentes en el instante $tn = \min\{t_1, t_2\}$, donde $t_1, t_2 \in T$ y corresponden a los diferentes instantes de tiempo almacenados en las bitácoras R_P y R_Q, respectivamente. Los dos últimos pasos se repiten hasta alcanzar el límite superior de T.

8.2.3 Consultas sobre patrones de movimiento

En [dMR04, dMR05] se propone un modelo discreto (en el espacio y el tiempo) para procesar consultas sobre los patrones de desplazamiento que siguen los objetos dentro de un conjunto de regiones previamente definidas y fijas, tal como podemos ver en la Figura 8.5.

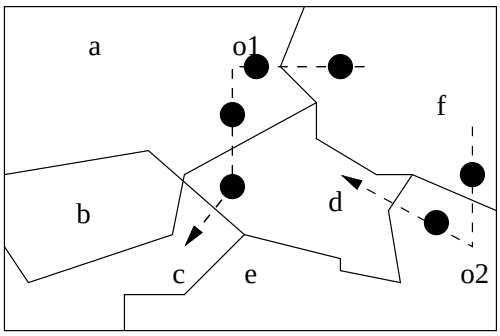


Figura 8.5: Objetos moviéndose sobre un mapa particionado.

En esta propuesta las trayectorias de los objetos se consideran como un string formado por los identificadores de las regiones, y las consultas como expresiones regulares formadas a partir de un alfabeto que incluye los identificadores de las regiones y un alfabeto para expresar variables. Una trayectoria de un objeto se representa como una secuencia que tiene la siguiente forma: $l_1\{T_1\}.l_2\{T_2\}\ldots.l_n\{T_n\}$, caracterizando las sucesivas regiones por donde un objeto pasó y el tiempo en que se mantuvo en cada una de ellas. Por ejemplo, la secuencia $a\{2\}.b\{4\}.c\{8\}.d\{2\}$, indica que el objeto primero pasó por la región *a* donde permaneció 2 unidades de tiempo, luego se trasladó a la región *b* donde se mantuvo por 4 unidades de tiempo, etc. Las consultas se expresan como una expresión regular. Por ejemplo, la consulta $(a|b).@x^+.e^+.@x^+.a$ permite recuperar los objetos que partieron en *a* ó *b*, luego llegaron a *e* por un camino seguido a través de otras regiones (distintas de *a*, *b* ó *e*) y luego desde *e*, siguiendo el mismo camino, regresan a *a*. Esta técnica no permite expresar restricciones sobre el tiempo y tampoco sobre la distancia. Por otra parte, las consultas sólo se pueden ejecutar sobre el sufijo de la trayectoria de un objeto y no sobre su historia completa.

8.2.4 Consultas sobre patrones espacio-temporales

En [HKBT05] se abordan las consultas sobre patrones espacio-temporales. Dicho trabajo amplía la expresividad del modelo descrito en la Sección 8.2.3. Una consulta STP (Spatio-Temporal Pattern) se define como una secuencia Ω de longitud arbitraria *m* de predicados espacio-temporales de la forma:

$$\Omega = \{(Q_1, T_1), (Q_2, T_2), \ldots, (Q_m, T_m)\}$$

donde en cada par (Q, T), Q representa un predicado espacial y T una restricción temporal. Q puede representar ya sea una rango espacial (window) o una consulta del vecino más cercano (*NN*). Sin embargo, el modelo puede ser fácilmente ampliado a otros predicados espaciales [HKBT05]. Por otra, parte T puede ser un instante de tiempo (t), un intervalo de tiempo (Δt) o vacío ($\emptyset$). Una consulta STP puede estar formada por predicados espaciales de cualquier tipo (rango, el vecino

más cercano, entre otros), donde cada predicado puede estar asociado con un instante de tiempo o intervalo de tiempo (STP With Time - STPWT) o, más generalmente, con un orden relativo con los otros predicados (STP With Order - STPWO). Un ejemplo de una consulta STPWT es la siguiente: "*encontrar los objetos que cruzaron la región A en t_1, llegaron a estar lo más cerca posible del objeto B en t_2 y luego se detuvieron dentro del círculo C en algún instante del intervalo $[t_3, t_4]$*". Un ejemplo de una consulta STPWO es la siguiente: *"encontrar los objetos que primero cruzaron la región A, luego pasaron lo más cerca posible del objeto B y finalmente se detuvieron dentro del círculo C"*. En este último tipo de consultas sólo interesa el orden relativo de los predicados espaciales independientemente de cuándo ocurrieron exactamente.

Una forma simple de evaluar las consultas STPWT es utilizando algunos de los métodos de acceso espacio-temporal propuestos hasta ahora (RT-tree, HR-tree, 3DR-tree, MVR-tree, entre otros). Cada predicado es evaluado individualmente y luego las respuestas parciales se combinan para obtener la respuesta definitiva. Este enfoque puede ser útil sólo cuando los predicados espaciales son rangos (window). Sin embargo, el problema se complica cuando los predicados espaciales necesitan evaluarse en forma conjunta. Por ejemplo "*encontrar los objetos que pasaron lo más cerca posible del punto A en t_1 y luego lo más cerca posible del punto B en t_2"*. En este caso no sirve evaluar individualmente cada predicado y luego combinar los resultados para obtener la respuesta, ya que la evaluación individual de cada predicado no minimiza la distancia entre ambos puntos. El problema se complica más aún cuando se consideran consultas STPWO, ya que ningún método de acceso espacio-temporal tradicional orientado a resolver consultas de tipo *time-slice* y *time-interval* (MVR-tree, HR-tree, RT-tree, MV3R-tree, entre otros) puede resolver eficientemente este tipo de consultas. Esto se debe a que sólo interesa el orden relativo de los predicados (la especificación del tiempo es relativa: antes, después, etc.) y, por lo tanto, la dimensión temporal no ayuda a discriminar. Por ejemplo la consulta "*encontrar los objetos que primero cruzaron la región A y posteriormente alcanzaron la región B"*, al intentar procesarla con HR-tree, requiere de una revisión que tome tiempo cuadrático en el número de R-trees virtuales.

En [HKBT05] se plantean varios algoritmos para procesar las consultas STPWT y STPWO. Para las primeras los algoritmos se basan en estrategias especializadas de evaluación sobre métodos de acceso espacio-temporal similares al R-tree o al MVR-tree y tratan de minimizar la cantidad de bloques accesados. Asumiendo que todos los predicados son de tipo rango espacial, un primer enfoque consiste en evaluar cada predicado separadamente usando el índice subyacente y luego, a partir de estos resultados parciales, obtener los objetos que pertenecen a la interseción de todos los conjuntos generados por cada predicado espacial. Una segunda estrategia consiste en evaluar primero el predicado más selectivo (suponiendo que la información de selectividad se encuentra disponible). El resultado de esta evaluación se carga en memoria y se utiliza para procesar los restantes predicados. Para consultas STPWO en [HKBT05] se propone una estructura de datos llamada CellList que corresponde a una partición del espacio, tal como se puede apreciar en la Figura 8.6. En esta estructura, cada celda se representa por un único identificador. A cada celda se le asocia una lista ordenada de entradas del tipo (P_l, t_k), donde cada entrada indica el instante de tiempo en que un objeto alcanzó la celda. Por ejemplo, el objeto P_3 cruzó la celda B en el instante de tiempo 3, y en el instante 4 el objeto P_1 entró a la celda B. P_2 entró en el instante 9 a B y en el instante 13 lo hizo de nuevo. Por lo tanto, la lista para la celda B es $\{(P_1,4),(P_2,9),(P_2,13),(P_3,3)\}$. La lista se mantiene ordenada por objeto e instante de tiempo.

Una consulta STPWO $\Omega = \{R(r_1), R(r_2), \ldots, R(r_n)\}$, con $R(r_i)$ una consulta de tipo WQ con rango espacial dado por r_i, puede ser evaluada usando una operación similar a "merge-join" con

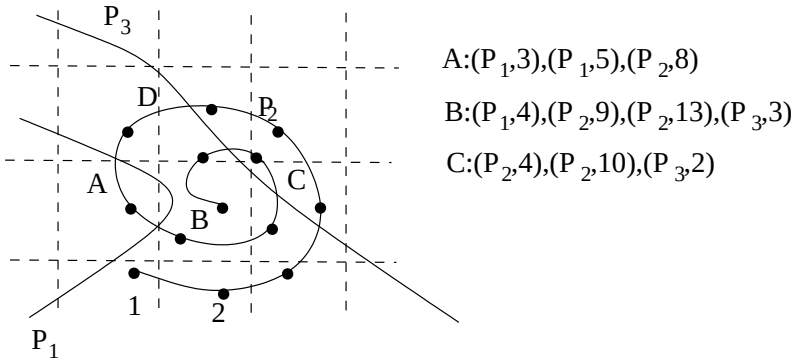


Figura 8.6: Una partición uniforme y la representación de celdas como listas ordenadas de identificadores de trayectorias.

las *n* listas asociadas con estos predicados. Por ejemplo si $A1$ está contenido por A y $B1$ por B, la consulta *"los objetos que cruzaron* $A1$ *en* $t = [5,8]$ *y luego pasaron por* $B1$ *en* $t = [9,13]$*"* conlleva procesar coordinadamente las listas de las celdas A y B. De manera similar, también es posible procesar con el CellList consultas STPWO que contemplan predicados espaciales sobre el vecino más cercano.

8.2.4.1 Evaluación de consultas STP con el SEST$_L$

Tal como comentamos anteriormente, la estructura del CellList considera una partición del espacio. Esta estrategia también es utilizada por el SEST$_L$, ya que también divide el espacio pero, en este caso, las celdas o subregiones se obtienen a partir del R-tree y, por lo tanto no son necesariamente del mismo tamaño ni disjuntas. Cada entrada de tipo evento en una bitácora tiene la estructura $\langle Oid, t, Geom, Op \rangle$ que es un superconjunto de los atributos considerados por el CellList ($\langle Oid, t \rangle$) y, en consecuencia, se pueden asimilar a las bitácoras del SEST$_L$ con las listas de las celdas del CellList (tienen exactamente la misma función). Con estas consideraciones es posible aplicar, casi directamente, los algoritmos definidos para evaluar consultas STPWO utilizando el CellList, considerando como estructura de datos subyacente la del SEST$_L$.

Según [HKBT05], las consultas definidas para patrones de desplazamiento, descritos en la Sección 8.2.3, forman un subconjunto de Ω y es factible evaluarlas con el CellList y, por lo tanto, también es posible de hacerlo con el SEST$_L$. El rendimiento esperado del SEST$_L$ para procesar estas consultas debería ser un poco peor que con el CellList (producto de las intersecciones de los MBRs de las bitácoras y del mayor espacio que necesita cada entrada), pero se debe considerar que con el SEST$_L$ se procesan eficientemente consultas de tipo *time-slice*, *time-interval*, sobre eventos y de reunión espacio-temporal. El CellList, sin embargo, es una estructura ad-hoc para consultas STPWO (no se reportaron resultados comparando otros tipos de consultas) y su utilidad para evaluar otros tipos de consultas espacio-temporales no es tan clara. Por ejemplo, una consulta *time-slice* implicaría procesar, en el peor caso, todas las entradas de cada lista que intersecta la componente espacial de la consulta. Estos pueden llegar a ser muchos bloques de disco. Sin embargo, esta consulta se ejecuta eficientemente en el SEST$_L$, ya que existen snapshots a nivel de las subregiones (entradas de tipo snapshot en las bitácoras). Además, la partición del espacio en

el CellList es arbitraria y, por lo tanto, pueden haber celdas vacías y otras con listas muy largas, lo que afectará en forma importante el rendimiento de consultas que intersecten estas últimas celdas.

8.2.4.2 Extensiones de las consultas STP

Considerando que las entradas de eventos en las bitácoras del SEST_L contemplan la geometría del objeto, es posible ampliar las consultas STP (Ω) agregándoles predicados que consideran tiempo relativo, restricciones sobre las distancia de los objetos en el tiempo y considerando explícitamente predicados sobre eventos. De esta forma podemos redefinir Ω de la siguiente manera:

$$\Omega = \{\langle S_1, E_1, T_1\rangle, \langle S_2, E_2, T_2\rangle \ldots \langle S_n, E_n, T_n\rangle\}$$

donde en cada tupla $\langle S, E, T\rangle$, S representa un predicado espacial (un rango espacial o una consulta del vecino más cercano (NN)) o un predicado basado en la distancia (Euclidiana) que se especifica mediante D. Este predicado permite acotar la distancia de desplazamiento de los objetos a D como máximo en el patrón. Por su parte E puede ser vacío ($\emptyset$) o uno de los eventos a (aparición de un objeto en región), s (permanencia de un objeto en una región) o l (un objeto deja una región). Finalmente, T puede ser un instante de tiempo (t), un tiempo relativo ($\mu t > 0$), un intervalo de tiempo (Δt) o vacío ($\emptyset$).

Este tipo de consultas incluyen las consultas STP. Por ejemplo, para especificar una consulta STPWO (con rangos) debemos considerar rangos espaciales como valores de S_i y $\emptyset$ como valores de E_i y T_i.

Con este tipo de consultas se pueden recuperar los objetos que siguen un patrón en el cual se pueden especificar restricciones sobre sus propios atributos espaciales y temporales, además de los especificaciones de tiempo y espacio dadas en los patrones de las consultas STPs originales. También es posible ampliar la capacidad del las STPs originales considerando los eventos que han generado los objetos.

Una consulta espacio-temporal de tipo Ω permite recuperar todos los objetos que satisfacen todos los predicados espacio-temporales del patrón, es decir,

$$STP(\Omega) = \{o | P_1(o) \wedge P_2(o) \wedge \ldots \wedge P_n(o) \textit{ es verdadero}\}$$

, donde $P_i(o)$ corresponde a la evaluación del objeto o en el i–ésimo predicado de Ω.

Un ejemplo de una consulta de tipo Ω sería la siguiente: *"los objetos que ingresaron a la región R1, en algún momento entre las 10:00 y 12:00 de ayer, permaneciendo allí por 20 minutos, luego, en algún instante entre las 13:00 y 15:00 horas se desplazaron a no más de 3 kilómetros y regresaron a R1"*. La especificación de la consulta sería la siguiente:

$$\{\langle R1, a, [10,12]\rangle, \langle R1, s, 20\rangle, \langle 3, [13,15]\rangle, \langle R1, a\rangle\}$$

Podemos observar que no es posible evaluar eficientemente la consulta anterior con el CellList, ya que la restricción "*permaneciendo allí por 20 minutos*" es muy costosa de verificar, pues implicaría, por cada objeto que primero entró en $R1$, buscar en las listas de todas las celdas los instantes de tiempo que ingresó con el propósito de verificar si permaneció en $R1$ por 20 minutos. La restricción "*... se desplazaron a no más de 3 kilómetros*" simplemente no es posible evaluarla con el CellList, ya que para hacerlo es necesario conocer las posiciones de los objetos que ingresaron a la región $R1$, atributo que no es considerado por las entradas del CellList.

Con el propósito de mostrar la factibilidad de procesar casos particulares de Ω con el SEST$_L$, a continuación se describen dos algoritmos. El primero (Alg. 8.4) evalúa una secuencia de predicados QE. Cada elemento de QE es una tupla de la forma $\langle R, te, T \rangle$, donde R es un rango espacial, te es el tipo de evento y T la especificación del tiempo. Una consulta espacio-temporal con predicados QE permite recuperar todos los objetos que satisfacen cada uno de los predicados de QE. Para simplificar el algoritmo suponemos que los valores para T pueden ser t ó μt, es decir, un instante de tiempo o un desplazamiento. También suponemos que si en la primera entrada del patrón el tiempo especificado es relativo (μt), entonces se considera como base el tiempo inicial de la base de datos (t_0). Finalmente, suponemos que para dos entradas $e_i \in QE$ y $e_j \in QE$, con $1 \leq i < j \leq |QE|, e_i.T \leq e_j.T$, es decir, los valores absolutos del tiempo especificado en los predicados deben ser crecientes. El algoritmo descrito en el Alg. 8.4 se puede mejorar de tal manera de evitar accesar más de una vez bitácoras que se intersecten con varios rangos espaciales. Para ello, se pueden diseñar procedimientos similares a los utilizados para implementar la operación de reunión espacio-temporal descrita en el Capítulo 7. También es posible extenderlo para considerar tuplas de la secuencia con valores para $T = \emptyset$. Por ejemplo, la consulta $\{\langle R1, a \rangle, \langle R2, a \rangle\}$ significa recuperar los objetos que primero entraron a la región $R1$ y después a la región $R2$. Para este tipo de consultas se puede considerar un algoritmo similar al utilizado para procesar las consultas de tipo STPWOR especificado en la Sección 8.3.

1: **STPE1(**$QE = \{\langle R_1, te_1, T_1 \rangle, \langle R_2, te_2, T_2 \rangle, \ldots, \langle R_n, te_n, T_n \rangle\}$**)**
2: $nt = 0$
3: t= $Tiempo(t_0, e_1.T)$ {El procedimiento $Tiempo(t1, t2)$ obtiene el instante de tiempo en el cual se verificará la ocurrencia del evento. Si $t2$ corresponde a un instante de tiempo, *Tiempo()* devolverá $t2$ y en caso contrario retornará el resultado de la expresión $t1 + t2$ }
4: **for** cada tupla $e_i \in QE$ con $1 \leq i \leq |QE|$ **do**
5: $S_i \leftarrow$ Event$(e_i.R, t, e_i.te)$ {El procedimiento $Event(R, t, te)$ obtiene todos los objetos que presentan un evento de tipo te, dentro del rango espacial R y en el instante de tiempo t.}
6: **if** $S_i = \emptyset$ **then**
7: $rsp = false$
8: **break**
9: **end if**
10: $t = Tiempo(t, e_i.T)$
11: **end for**
12: **if** rsp **then**
13: reportar $\{o \ / \ o \in S_i, \forall\, i,\ 1 \leq i \leq |QE|\}$
14: **end if**

Alg. 8.4: Algoritmo para evaluar consultas sobre patrones de eventos.

Un segundo algoritmo (Alg. 8.5) muestra cómo se pueden evaluar consultas Ω en las cuales se especifican restricciones sobre las distancias alcanzadas por los objetos. La consulta Q es un caso particular y sencillo, pero permite vislumbrar la factibilidad de procesarla con el SEST$_L$ y proyectar la evaluación de consultas tipo Ω más complejas.

$$Q = \{\langle R, a, t_1 \rangle, \langle D, \mu t_2 \rangle\}$$

Una consulta de tipo Q permite recuperar todos los objetos que ingresaron a la región R en el instante de tiempo t_1 y luego, en el instante $t_1 + \mu t_2$, se encuentran (a partir de su posición actual) a una distancia no mayor a D. Un ejemplo de una consulta de este tipo puede ser la siguiente

"los objetos que entraron a la región R_1 en el instante t_1 y luego se encontraban a no más de 10 unidades de distancia en el instante $t_1 + 30$".

La estrategia seguida para resolver este tipo de consultas es la siguiente. En primer lugar obtenemos todos los objetos que entraron a R_1 en el instante t_1. En la Figura 8.7 los objetos seleccionados corresponden al conjunto $\{o_1, o_2, o_3, o_4, o_5\}$, donde el rectángulo con línea punteada (R) es el rango de la consulta. Posteriormente, y a partir de la posición actual de los objetos y de D, se obtiene un segundo rango espacial (R' en la Figura 8.7). A continuación se ejecuta una consulta *time-slice* con parámetros R' y tiempo $t_1 + 30$. Finalmente, se reportan los objetos que se encuentran en ambos conjuntos.

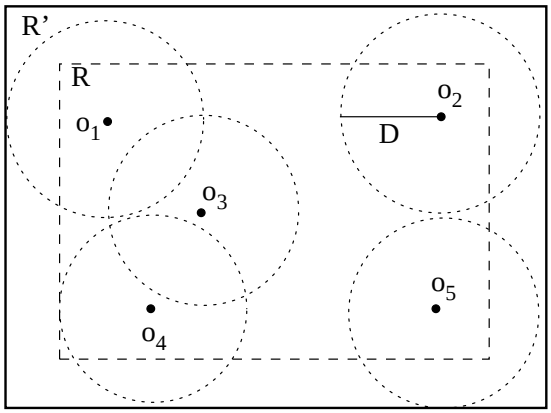


Figura 8.7: Estrategia para evaluar consultas con restricciones sobre los desplazamientos de los objetos.

1: **STPE2(**$R, t_1, D, \mu t_2$**)**
2: $A \leftarrow Event(R, t_1, a)$
3: $R' \leftarrow NewRangeQuery(A, D)$
4: $B \leftarrow timeSliceQuery(R', t_1 + \mu t_2)$
5: Reportar $\{o \ / \ o \in A \cap B\}$

Alg. 8.5: Consultas que consideran desplazamiento de objetos en el tiempo.

Es posible también incorporar más opciones a las consultas sobre los desplazamientos (Q). Por ejemplo, agregar la dirección del desplazamiento. Así sería posible evaluar consultas como las siguientes: *"los objetos que entraron a la región R_1 en el instante t y luego se desplazaron hacia el norte a no más de 30 Km en el intervalo de tiempo $[t+1, t+30]$"*.

La Tabla 8.1 resume los diferentes tipos de consultas analizados, su factibilidad de implementarlas con el SEST_L y el probable rendimiento o costo de su evaluación

8.3 Algoritmo para evaluación de consultas STPWOR

En esta sección se describe en detalle un algoritmo para procesar consultas STPWOR (Spatio-temporal Pattern With Order Range) un caso particular de las consultas STP y que se discutieron en la Sección 8.2.4) utilizando el método de acceso espacio-temporal SEST_L. Se realizaron varios

Tipo de consultas	Eficiencia al procesar con el $SEST_L$
Vecino más cercano	Rendimiento similar al *K-NN* espacial. Agregando el efecto de los eventos sobre el Rp-tree (crecimiento de la densidad) y el procesamiento de los bloques de las bitácoras.
Pares de vecinos más cercanos	Rendimiento muy similar *K-CPQ* espacial. Con las mismas consideraciones que para el *K-NN*.
Patrones espacio-temporales (STP)	Rendimiento similar al del CellList (costo entre un 10% y 15% más que con CellList, ver Sección 8.3.1)
Patrones en movimiento	Estas consultas están incluídas en las consultas de tipo STP
Extensiones de las STP	Rendimiento similar a las consultas de tipo STP

Tabla 8.1: Tipos de consultas espacio-temporales posibles de procesar con el $SEST_L$.

experimentos que miden el rendimiento del algoritmo frente al CellList, MVR-tree y R-tree para procesar las mismas consultas. También se elaboró un modelo de costo que predice el rendimiento del algoritmo.

8.3.1 Algoritmo basado en el $SEST_L$

Tal como ya comentamos, al comparar las ideas subyacentes detrás del CellList y el $SEST_L$ es posible ver que existe una gran similitud en ambos enfoques. Ambos consideran una partición del espacio en subregiones. En el caso del CellList la partición se hace independientemente de las localizaciones o posiciones de los objetos; en cambio en el $SEST_L$ las particiones están dirigidas por las posiciones iniciales de los objetos y en base a las particiones generadas por el R-tree. Además, las estructuras de datos de las listas de cada estructura son diferentes. Las entradas de las listas del CellList son tuplas de la forma $\langle Oid, t\rangle$, en cambio las listas del $SEST_L$ consideran entradas para los eventos y para mantener snapshots a nivel de cada subregión. La similitud de ambos enfoques permiten resolver las consultas STPWOR prácticamente de la misma manera.

La Figura 8.8 muestra un escenario de una consulta STPWOR con tres predicados espaciales (R_1, R_2 y R_3) y un $SEST_L$ formado por 4 bitácoras (L_1, L_2, L_3 y L_4). Podemos ver que el primer predicado espacial R_1 se intersecta con las bitácoras L_1 y L_2, a su vez R_2 se intersecta con las bitácoras L_2 y L_3 y R_3 con L_4 y L_3, respectivamente.

En el Alg. 8.6 se muestra el procedimiento general seguido para procesar las consultas de tipo STPWOR. El procedimiento *CrearLista(R_i)* se utiliza para formar las listas asociadas a cada predicado espacial. Las entradas de estas listas tienen la misma estructura que las del CellList, es decir, $\langle Oid, t\rangle$. Para seleccionar las bitácoras se utiliza el Rp-tree del $SEST_L$, fijando como parámetro de búsqueda espacial cada uno de los predicados dados en la consulta STPWOR. El procedimiento *CrearLista(R_i)* obtiene, en una sola lista ordenada por *Oid* y *t*, todas las entradas de todas las bitácoras intersectadas por un predicado espacial. Las listas obtenidas con *CrearLista(R_i)* se procesan con el algoritmo descrito en el Alg. 8.7, el cual es similar al propuesto en [HKBT05].

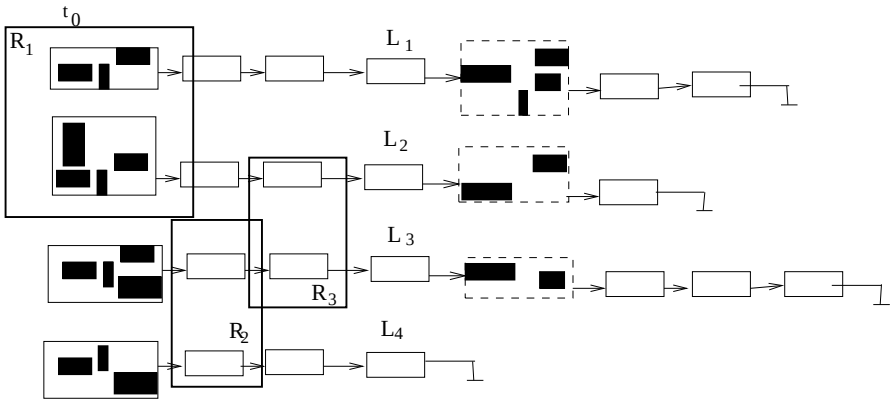


Figura 8.8: Ejemplo de una consulta STPWOR con tres predicados espaciales.

1: **STPWOR(**$R_1, R_2, \ldots, R_m$**)**
2: **for** $i = 1$ to m **do**
3: $L_i \leftarrow$ CrearLista(R_i)
4: **end for**
5: MezclarListas($L_1, L_2, \ldots, L_m$)

Alg. 8.6: Algoritmo general para resolver STPWOR con el SEST$_L$.

8.3.2 Modelo de costo para el algoritmo STPWOR

En esta sección se propone un modelo de costo que permite estimar el rendimiento del algoritmo para procesar las consultas de tipo STPWOR. En la definición del modelo utilizamos las mismas variables definidas en la Tabla 4.2 del Capítulo 2.

Sean $R_1, R_2, \ldots, R_m$ los predicados espaciales de una consulta STPWOR. Suponemos que todos los R_i son cuadrados de la misma área. Sea *nl* la cantidad de bitácoras que, en promedio, un R_i intersecta, y *bl* la cantidad de bloques que en promedio se necesitan para almacenar los eventos producidos en cada bitácora a lo largo de todos los snapshots y almacenados en la base de datos. El total (TB) de bloques que se requieren procesar al evaluar una consulta STPWOR utilizando el SEST$_L$ queda determinado por la siguiente ecuación.

$$TB = m \cdot nl \cdot bl \tag{8.6}$$

8.3.2.1 Calculando el número (*nl*) y el tamaño (*bl*) de las bitácoras

Para estimar *nl*, se utiliza el modelo de costo que predice el rendimiento de una consulta espacial de tipo WQ (window query), considerando que los objetos espaciales se almacenan en un R-tree (definido en [TS96]) y que resumimos en el Capítulo 2.

Dado que los MBRs del Rp-tree son siempre crecientes, es necesario reflejar el efecto de los eventos en el Rp-tree. Para capturar este efecto, se definió una nueva expresión para el parámetro *c* de la Ec. (2.7) y que denominamos *nc* (Ec. (8.7)) el cual es similar a la que fue establecida para la operación de reunión espacio-temporal.

$$nc = (p \cdot nt + 1) \cdot c \cdot M \tag{8.7}$$

```
1:  MezclarListas(L_1, L_2 ..., L_m)
2:  U ← ∅ {Conjuntos de candidatos}
3:  for i = 1 to m do
4:     for j = 1 to m do
5:        if L_1[i].Oid ∉ L_j then
6:           break
7:        else
8:           Sea k la primera entrada de L_1[i].Oid en L_j
9:           while L_1[i].Oid = L_j[k].Oid ∧ L_1[i].t > L_j[k].t do
10:             k = k + 1
11:          end while
12:          if L_1[i].Oid ≠ L_j.Oid then
13:             break
14:          end if
15:          U = U ∪ L_1[i].Oid
16:       end if
17:    end for
18: end for
```

Alg. 8.7: Algoritmo para mezclar listas.

Con *nc* se puede obtener el valor real de la densidad de los MBRs de las bitácoras (Ec. (8.8)) y, por lo tanto, calcular el valor *nl*.

$$D'_1 = \left(1 + \frac{\sqrt{D_0} - 1}{\sqrt{(nc \cdot M)}}\right)^2 \tag{8.8}$$

$$nl = \left(\sqrt{D'_1} + q \cdot \sqrt{\frac{N}{c \cdot M}}\right)^2 \tag{8.9}$$

La cantidad de bloques que, en promedio, cada bitácora requiere para almacenar los cambios se obtiene con la siguiente ecuación.

$$bl = \left(\frac{N \cdot p \cdot nt}{cb \cdot N_1}\right) \tag{8.10}$$

donde N_1 corresponde a la cantidad de bitácoras o nodos hojas del R-tree la cual se obtiene con la Eq.(2.5) $j = 1$.

De esta forma, la cantidad de bloques accesados por el SEST_L al evaluar una consulta STPWOR queda determinado por la Ec. (8.11).

$$TB = m \cdot \left(\sqrt{D'_1} + q \cdot \sqrt{\frac{N}{c \cdot M}}\right)^2 \cdot \left(\frac{N \cdot p \cdot nt}{cb \cdot N_1}\right) \tag{8.11}$$

8.3.2.2 Evaluación del modelo

Se realizaron varios experimentos con el propósito de evaluar la capacidad de predicción del modelo. Se consideraron 23.268 objetos (puntos) con distribución uniforme con una movilidad de 10% y 200 instantes de tiempo. El valor de *d* fue 4 y el tamaño de los bloques de disco fue de 1.024 bytes. Los largos de las secuencias de predicados de las consultas STPWOR fueron 5, 10, 15,

20, 25 y 30 predicados. Para cada tipo de consulta, con una determinada cantidad de predicados, se generaron al azar 100 consultas STPWOR. Los predicados de las consultas fueron formados con rectángulos de lado igual a 1.5% y 3% de cada dimensión del espacio total. La decisión de medir predicados de tamaño 1.5% $\times$ 1.5% se debió a que este tamaño fue utilizado en [HKBT05] para medir la eficiencia del CellList. Las mediciones para predicados de 3% $\times$ 3% se hicieron para confirmar la capacidad de predición del modelo.

Podemos ver en la Figura 8.9 que el modelo estima muy bien el rendimiento de las consultas STPWOR. El error relativo promedio es de un 10%.

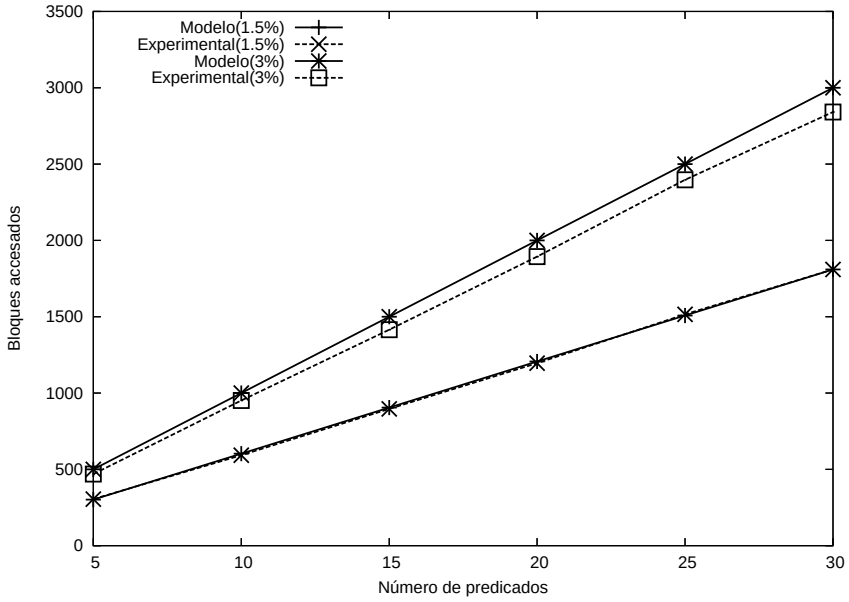


Figura 8.9: Rendimiento de las consultas STPWOR con el SEST_L según el modelo de costo y experimental.

8.3.3 Evaluación del algoritmo STPWOR

Con el objeto de comparar la eficiencia del SEST_L para procesar consultas de tipo STPWOR, lo comparamos con el CellList, R-tree (2-dimensiones) y el MVR-tree. La alternativa del R-tree (2-dimensiones) se propone en [HKBT05], donde se utilizó como testigo para comparar al CellList. Dicha alternativa consiste en desacoplar la componente temporal de los objetos, es decir, se consideran solamente las diferentes posiciones de los objetos a través del tiempo. Estas posiciones se indexan por un R-tree de dos dimensiones. De esta forma un predicado espacial puede fácilmente recuperar todas las trayectorias que lo satisfacen, independientemente del tiempo en que ocurrió. Así, para saber si una trayectoria cumple con el predicado de una consulta STPWOR se deben evaluar, primero que nada, todos los predicados de la consulta en forma independiente, luego obtener la intersección de los conjuntos y, finalmente, las trayectorias resultantes deben ser verificadas para asegurarse que satisfagan todos los predicados en el orden dado por la consulta STPWOR.

Para estimar la eficiencia de un MVR-tree para evaluar consultas STPWOR, se procedió de la siguiente manera: se evalúan los *m* predicados de la consulta a partir del primer instante de tiempo almacenado en la base de datos, es decir, se ejecuta una consulta *time-slice* considerando el primer predicado y, como instante de tiempo, el primero almacenado en la base de datos. Luego se realiza un segundo *time-slice* considerando el siguiente predicado espacial y el segundo instante de tiempo. Esto continúa para todos los predicados espaciales de la consulta STPWOR. Luego, los objetos que se encuentran en la intersección de todos los resultados de las consultas *time-slice* se retornan como parte de la respuesta. El procedimiento sigue, ahora, emparejando el primer predicado de la consulta STPWOR con el segundo instante de tiempo almacenado en la base de datos, repitiendo todo de nuevo. Para estimar la cantidad de bloques accesados por una consulta *time-slice* considerando un MVR-tree se utilizó el modelo de costo descrito en [TPZ02].

En la evaluación tratamos de establecer los mismos parámetros del conjunto de datos que los utilizados en [HKBT05]. Específicamente, consideramos 300.000 objetos (puntos) con una movilidad de 10% y 200 instantes de tiempo. También consideramos bloques de tamaño 4 Kbytes y un tamaño de bitácora igual a 4 bloques. Dado que el generador de datos spatio-temporales GSTD [TSN99] considera como máximo 100.000 objetos, utilizamos el modelo definido en la sección 8.3.2 para estimar la eficiencia de nuestro algoritmo (SEST$_L$). Los datos del CellList y R-tree (2 dimensiones) se obtuvieron directamente de los experimentos realizados en [HKBT05]. Para el caso del SEST$_L$ evaluamos dos situaciones con respecto a la estructura de las entradas de los eventos del SEST$_L$. El primer caso considera que cada movimiento genera dos entradas de bitácora del tipo $\langle Oid, t, Geom \rangle$. A esta situación le denominamos SEST$_L$ (original) en las Figuras 8.10 y 8.11. El segundo caso considera los cambios en la estructura de las entradas explicados en la Sección 5.4.2 (Capítulo 5). A esta última situación le llamamos SEST$_L$ (ajustada).

Las Figuras 8.10 y 8.11 permiten comparar el algoritmo STPWOR con el CellList, R-tree (2 dimensiones) y el MVR-tree. Podemos observar que el SEST$_L$ (original) requiere de aproximadamente un 35% más de accesos que el CellList y que el SEST$_L$ (ajustada) requiere de sólo un 15% más de accesos que el CellList. Estas diferencias se explican porque las entradas de las bitácoras del SEST$_L$ mantienen la información espacial lo que obliga a ocupar más bloques de disco. Por otra parte, el SEST$_L$ (original) y el SEST$_L$ (ajustada) superan ampliamente al R-tree (2 dimensiones) y al MVR-tree en este experimento. En nuestra opinión, la comparación del algoritmo STPWOR debe hacerse con el MVR-tree, ya que sólo éste permite procesar el mismo tipo de consultas (*time-slice* y *time-interval*) que el SEST$_L$ con una eficiencia comparable. Por otro lado, el CellList es una estructura ad-hoc para procesar consultas de tipo STPWOR y en [HKBT05] no se presentan algoritmos para procesar consultas de tipo *time-slice* o *time-interval* y tal como comentamos en la Sección 8.2.4.1 el procesamiento de una consulta de este tipo se torna muy ineficiente con el CellList.

8.4 Conclusiones

Como podemos ver en la Tabla 8.1 es posible procesar varios tipos de consultas con el SEST$_L$. Algunos de ellos han sido bastante estudiados en el ámbito de las bases de datos espaciales (*K-NN* y *K-CPQ*) y son de mucho interés en el contexto espacio-temporal. Estas consultas se pueden procesar con el SEST$_L$, ya que se basa fundamentalmente en un R-tree. Las consultas sobre patrones (STP y otras) también es factible evaluarlas con el SEST$_L$, ya que la estructura de datos diseñada ad-hoc (CellList) para evaluarlas es muy similar a la estructrura del SEST$_L$; en ambos

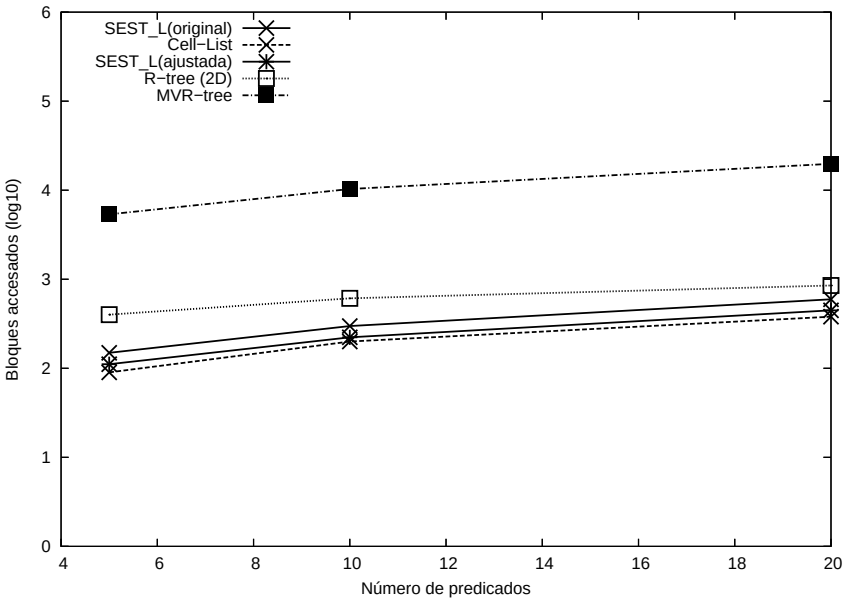


Figura 8.10: Rendimiento de las consultas de tipo STPWOR considerando diferentes número de predicados y cada predicado espacial formado con 1.5% de cada dimensión.

casos se hace una partición del espacio y se asignan listas para mantener información sobre los eventos de los objetos.

El procesamiento de consultas STPWOR con el SEST_L se puede hacer de manera casi tan eficiente como se hace con el CellList. Sin embargo, el CellList no permite procesar consultas de tipo *time-slice* o *time-interval* eficientemente. Por otro lado, nuestra propuesta (algoritmo STPWOR) superó ampliamente al MVR-tree (el que sí puede evaluar consultas *time-slice* y *time-interval*). Estos resultados amplían las capacidades del método de acceso espacio-temporal SEST_L.

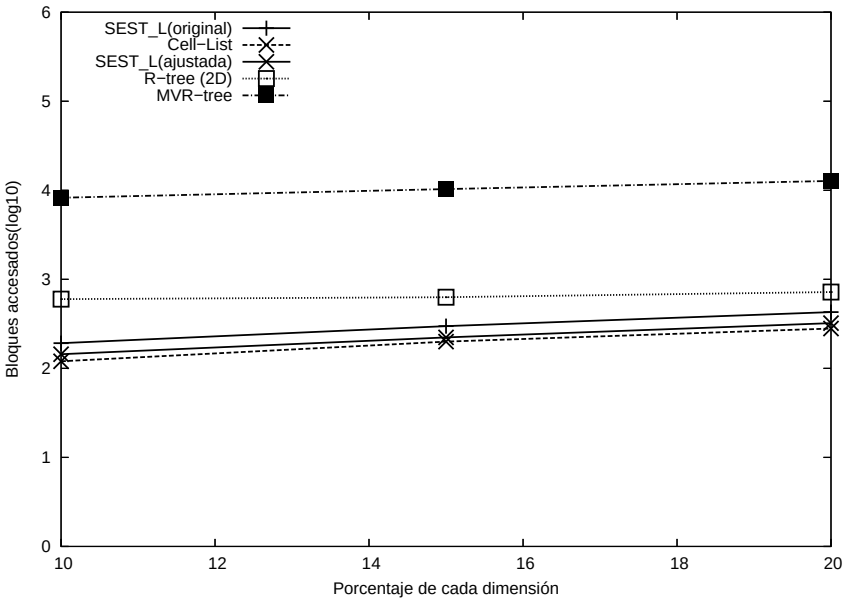


Figura 8.11: Rendimiento de las consultas STPWOR considerando rangos espaciales formados por diferentes porcentajes de cada dimensión y 10 predicados por consultas.

Capítulo 9

Conclusiones y trabajo futuro

Actualmente existe una necesidad creciente por el almacenamiento y procesamiento de información espacio-temporal. Aplicaciones en el ámbito de los Sistemas de Información Geográfica, pasando por aplicaciones multimedia y telefonía celular, por citar algunas, están considerando la dimensión temporal de los objetos. Uno de los problemas comunes a estas aplicaciones es el procesamiento de las consultas que tienen en cuenta restricciones espacio-temporales. Esta tesis se concentró en el diseño de nuevos métodos de acceso y de algoritmos para el procesamiento de consultas espacio-temporales utilizando tales métodos de acceso. En este capítulo se detallan las conclusiones y aportes de esta tesis, como también los principales problemas a abordar a futuro en este ámbito.

9.1 Conclusiones

Debido a la necesidad cada vez más frecuente por contar con aplicaciones espacio-temporales, existe mucho interés en incorporar a los SABDs tipos de datos espacio-temporales con el propósito de apoyar la implementación de tales aplicaciones. Dicho interés ha abierto un campo amplio de investigación sobre tópicos que se refieren a lenguajes de consulta, métodos de acceso, algoritmos de reunión, entre otros para los tipos de datos espacio-temporales.

Los métodos de acceso espacio-temporal propuesto en esta tesis se basan en el enfoque de snapshots y eventos. Este enfoque permite que se representen de manera explícita los eventos espaciales de los objetos en la estructura de datos subyacente. Esta capacidad constituye una diferencia fundamental con los actuales métodos de acceso quienes manejan los datos de los eventos de los objeto, sólo con el propósito de actualizar sus estructuras de datos y no los mantienen a lo largo del tiempo. Nuestro enfoque permite explotar la información de los eventos con el propósito de evaluar eficientemente otros tipos de consultas, diferentes a las clásicas *time-slice* y *time-interval*, como por ejemplo consultas sobre eventos. Las principales conclusiones de nuestros métodos de acceso se resumen a continuación.

- Nuestro primer método de acceso, SEST-Index necesitó de menos almacenamiento que el HR-tree (método más eficiente para evaluar consultas *time-slice* según la literatura y por lo tanto el más eficiente para evaluar consultas sobre eventos por medio de consultas *time-slice*) para porcentajes de movilidad entre un 1% y 13%. El SEST-Index también superó al HR-tree para consultas de tipo *time-interval* y sobre eventos. Para las consultas *time-interval*

el SEST-Index aumenta su ventaja sobre el HR-tree en la medida que aumentan tanto el intervalo temporal de la consulta como el área de la misma. Sin embargo, el SEST-Index es superado por el HR-tree para procesar consultas de tipo *time-slice*. Un estudio preliminar de una variante del SEST-Index (que denominamos variante) demostró que es posible almacenar todos los objetos y sus eventos con sólo un 10% del almacenamiento requerido por el HR-tree y procesar consultas de tipo *time-interval* en sólo un 25% del tiempo que esta misma estructura necesita. Sin embargo, la variante presenta dos limitaciones: (1) el espacio de trabajo debe ser fijo a lo largo del tiempo y (2) los objetos espaciales deben ser puntos. También se definieron modelos de costos que predicen el almacenamiento ocupado y la eficiencia de las consultas utilizando el SEST-Index. Los modelos presentaron un error relativo promedio de un 7%.

- Nuestro segundo método de acceso espacio-temporal, denominado SEST_L, fue comparado con el MVR-tree (uno de los que presenta el mejor rendimiento de acuerdo con la literatura) al que en general superó tanto en almacenamiento como en eficiencia para evaluar las consultas. El SEST_L sólo necesita de aproximadamente un 58% del almacenamiento requerido por el MVR-tree y presenta, en general, un mejor rendimiento que el MVR-tree para consultas *time-interval*. Con el propósito de mantener un rendimiento más o menos constante de las consultas a lo largo del tiempo, se implementó el SEST_L con snapshots globales. Esta variante resultó ser una buena solución para aplicaciones en las que los objetos no se distribuyen de manera uniforme, bastando sólo un poco de almacenamiento adicional (16% del almacenamiento ocupado por el SEST_L sin snapshots globales) para alcanzar un rendimiento similar a los casos con distribución uniforme. También se comparó el SEST_L con el SEST-Index, al que superó llegando a necesitar sólo un 15% del almacenamiento y 20% del tiempo requerido por el SEST-Index para procesar las consultas. Para el SEST_L (con y sin snapshots globales) se definieron y validaron modelos de costos que estiman el almacenamiento y eficiencia de las consultas. Los modelos presentaron una buena capacidad de predicción alcanzando un error relativo promedio de un 15% para el almacenamiento y de un 11% para las consultas.

- Por medio de un modelo de costo generalizado se demostró que la asignación de las bitácoras en las hojas del R-tree (SEST_L) es la más conveniente tanto desde el punto de vista del almacenamiento como de la eficiencia de las consultas.

- Utilizando SEST_L, se elaboró un algoritmo para procesar la operación de reunión espacio-temporal. Se comparó dicho algoritmo con el basado en el 3D R-tree al cual superó ya que necesitó a lo sumo un 20% del tiempo requerido por el 3D R-tree. También se definió un modelo de costo para el algoritmo de reunión el que presentó un error relativo promedio de 6%.

- Se analizó la factibilidad de procesar con el SEST_L consultas definidas en el ámbito espacial (el vecino más cercano y los pares de vecinos más cercanos) y que es de interés extenderlas al dominio espacio-temporal. Se demostró que la evaluación de tales consultas con el SEST_L es bastante directa y simple de implementar. También se demostró que es posible implementar algoritmos basados en SEST_L para procesar consultas sobre patrones espacio-temporales y patrones de movimiento. Se implementó un tipo de consultas específico (STPWOR) por medio del SEST_L y se comparó con el CellList (una estructura de datos had-hoc para este

tipo de consultas), el MVR-tree y el R-tree (2D). Los resultados mostraron que las consultas STPWOR se pueden resolver con el SEST$_L$ tan eficientemente como con el CellList y que el SEST$_L$ supera por lejos al MVR-tree y al R-tree (2D) en este tipo de consultas.

Los resultados experimentales contenidos en esta tesis permiten concluir que el SEST$_L$ es el método de acceso espacio-temporal que presenta un mejor rendimiento en almacenamiento/tiempo para las consultas de tipo *time-interval* y sobre eventos. Además resultó ser muy eficiente para evaluar la reunión espacio-temporal y consultas STPWOR, lo que lo transforma en el único método conocido hasta ahora capaz de procesar estos tipos de consultas de manera eficiente y con un costo de almacenamiento moderado.

9.2 Trabajo futuro

En esta sección delineamos una serie de problemas interesantes que pueden ampliar las capacidades de nuestros métodos de acceso.

- La estructura de datos del SEST$_L$ mantiene información que puede ser explotada para procesar consultas espacio-temporales que han aparecido recientemente. En este grupo cobran espacial interés las consultas de tipo STP (Spatio-Temporal Pattern) las que constituyen un superconjunto de las consultas STPWOR y de las consultas sobre patrones en movimiento. Considerando la información de las entradas en las bitácoras, es posible ampliar las consultas de tipo STP en las cuales se consideren tiempos relativos, restricciones sobre las distancia de desplazamiento de los objetos y predicados sobre eventos, es decir, consultas como la siguiente: "*los objetos que ingresaron a la región R*1 *en algún momento entre las 10:00 y 12:00 de ayer, permaneciendo allí por 20 minutos, luego y en algún instante entre las 13:00 y 15:00 horas, se desplazaron a no más de 3 kilómetros y regresaron a R*1".

- Nuestro algoritmo para procesar la operación de reunión considera que los dos conjuntos de objetos cuentan con un índice espacio-temporal (SEST$_L$). Sin embargo, existen casos o situaciones en las cuales uno o ninguno de los conjuntos disponen de un índice. Estos casos son muy frecuentes, ya que es muy común realizar operaciones de reunión entre conjuntos de objetos espacio-temporales obtenidos previamente mediante operaciones de selección.

- La información que se almacena sobre los eventos en las bitácoras se puede ver como transacciones ocurridas sobre los objetos espaciales. De esta forma es posible aplicar métodos de minería de datos para descubrir patrones inmersos en la base de datos por medio de reglas de asociación que pongan de relieve los movimientos o cambios incidentales de objetos sobre otros. Por ejemplo, "*siempre que el objeto* o_1 *se mueve, también lo hacen los objetos* o_2 *y* o_3 *pero tres minutos más tarde*". También es interesante contar con algoritmos de minería de datos para descubrir patrones de movimiento de los objetos. Por ejemplo, "*en qué sentido (norte, sur, etc.) se desplazan los objetos en ciertos periodos de tiempo*" o si los objetos son automóviles, "*entre las 8 y 10 de la mañana los autos se desplazan hacia el centro de la ciudad* " o que "*entre las 12:00 y 13:00 la mayoría de los autos no se mueven*".

- La mayoría de las consultas evaluadas con el SEST$_L$ requieren procesar de manera independiente varias bitácoras lo que permite vislumbrar que el SEST$_L$ puede tomar muchas

ventajas de la paralelización de los algoritmos que procesan tales consultas. Una forma de abordar la paralelización sería asignando grupos de bitácoras a distintos procesadores manteniendo el Rp-tree en un procesador. De esta manera, para procesar una consulta espacio-temporal de tipo *time-interval* por ejemplo, la consulta se dirige hacia el procesador que contiene el Rp-tree, donde se seleccionan las bitácoras involucradas por la consulta. Claramente estas bitácoras se pueden procesar en forma paralela y formar la respuesta como la unión de los resultados obtenidos a partir de cada bitácora.

- En esta tesis se estudiaron los casos en los cuales las bitácoras se asignan a nivel de las hojas o en la raíz del R-tree. Tales alternativas fueron implementadas y evaluadas. La asignación a los restantes niveles fueron estudiados sólo con el modelo general de costo. En este punto sería muy conveniente estudiar dos aspectos: (i) evaluar experimentalmente las asignaciones de las bitácoras en los niveles intermedios y (ii) diseñar algoritmos que asignen dinámicamente las bitácoras a diferentes niveles del R-tree para un mismo conjunto de objetos espacio-temporales. Con esta última idea es posible que en un mismo índice, algunas bitácoras se asignen a áreas del nivel 1, otras a áreas del nivel 2, etc.

- Un línea de trabajo interesante es el diseño de algoritmos de concurrencia, de recuperación, entre otros, para el SEST_L para permitir su integración en un contexto de bases de datos.

Referencias

[AAE00] Pankaj K. Agarwal, Lars Arge, and Jeff Erickson. Indexing moving points. In *Symposium on Principles of Database Systems*, pages 175–186, 2000.

[BBK01] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, 2001.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[BF79] Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11(4):397–409, 1979.

[BGO+96] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion B-tree. *The VLDB Journal*, 5(4):264–275, 1996.

[BKS93] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using R-trees. In *ACM SIGMOD Conference on Management of Data*, pages 237–246, Washington, DC, USA, 1993. ACM.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-Tree: An efficient and robust access method for points and rectangles. In *ACM SIGMOD Conference on Management of Data*. ACM, 1990.

[BKSS94] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. Multi-step processing of spatial joins. In *ACM SIGMOD Conference on Management of Data*, pages 197–208, Minnesota, USA, 1994.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

[BO79] Jon Louis Bentley and Thomas Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Computers*, 28(9):643–647, 1979.

[CEP03] Prasad Chakka, Adam Everspaug, and Jignesh M. Patel. Indexing large trajectory data sets with SETI. In *Proceedings of the Intl. Conf. on Management of Innovative Data Systems Research, CIDR*, Asilomar, CA, 2003.

[CMTV00] Antonio Corral, Yannis Manolopoulos, Yannis Theodoridis, and Michael Vassilakopoulos. Closest pair queries in spatial databases. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 189–200, New York, NY, USA, 2000. ACM Press.

[Cor02] Antonio Corral. *Algoritmos para el Procesamiento de Consultas Espaciales utilizando R-trees. La Consulta de los Pares Más Cercanos y su Aplicación en Bases de Datos Espaciales*. PhD thesis, Universidad de Almería, Escuela Politécnica Superior, España, Enero 2002.

[dMR04] Cédric du Mouza and Philippe Rigaux. Mobility patterns. In *Proceedings STDBM*, pages 1–8, 2004.

[dMR05] Cédric du Mouza and Philippe Rigaux. Mobility patterns. *GeoInformatica*, 9(4):297–319, 2005.

[dWNCdTdM06] Van de Weghe Nico, Anthony G. Cohn, Guy de Tre, and Philippe de Maeyer. A qualitative trajectory calculus as a basis for representing moving objects in geographical information systems. *Cybernetics and Control*, 35(1):97–120, 2006.

[Ege94] M. J. Egenhofer. Spatial SQL: A query and presentation language. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):86–95, 1994.

[ES02] Martin Erwig and Markus Schneider. Spatio-temporal predicates. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):881–901, 2002.

[FSR87] Christos Faloutsos, Timos Sellis, and Nick Roussopoulos. Analysis of object oriented spatial access methods. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 426–439, New York, NY, USA, 1987. ACM Press.

[GBE+00] Ralf Hartmut Güting, Michael H. Böhlen, Martin Erwig, Christian S. Jensen, Nikos A. Lorentzos, Markus Schneider, and Michalis Vazirgiannis. A foundation for representing and querying moving objects. *ACM Trans. Database Syst.*, 25(1):1–42, 2000.

[GG98] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[GM95] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques and applications. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing*, 18(2):3–18, 1995.

[GS05] Ralf Hartmut Güting and Markus Schneider. *Moving Objects Databases*. Morgan Kufmann, 1st edition, 2005.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD Conference on Management of Data*, pages 47–57. ACM, 1984.

[Gut94] Ralf Guting. An introduction to spatial database systems. In *VLDB Journal*, volume 3, pages 357–399, 1994.

[GW05] Antony Galton and Michael F. Worboys. Processes and events in dynamic geo-networks. In M. Andrea Rodríguez, Isabel F. Cruz, Max J. Egenhofer, and Sergei Levashkin, editors, *GeoSpatial Semantics, First International Conference, GeoS, 2005, Mexico City, Mexico, November 29-30, 2005, Proceedings*, volume 3799 of *Lecture Notes in Computer Science*, pages 45–59. Springer, 2005.

[HKBT05] Marios Hadjieleftheriou, George Kollios, Petko Bakalov, and Vassilis J. Tsotras. Complex spatio-temporal pattern queries. In *Proceedings of the 31st international conference on Very large data bases (VLDB '05)*, pages 877–888. VLDB Endowment, 2005.

[HKTG02] Marios Hadjieleftheriou, George Kollios, Vassilis J. Tsotras, and Dimitrios Gunopulos. Efficient indexing of spatio-temporal objects. In *Extending Database Technology*, pages 251–268, 2002.

[KF93] Ibrahim Kamel and Christos Faloutsos. On packing R-trees. In *CIKM*, pages 490–499, 1993.

[KGK93] Won Kim, J Garza, and A Keskin. Spatial data management in database systems: research directions. In *Advances in Spatial Databases, 3er Symposium, SSD'93*, volume 692 of *Lecture Notes in Computer Science*, pages 1–13, Singapore, 1993.

[Kol00] George N Kollios. *Indexing Problems in SpatioTemporal Databases*. PhD thesis, Polytechnic University, New York, June 2000.

[KTG+01] George Kollios, Vassilis J. Tsotras, Dimitrios Gunopulos, Alex Delis, and Marios Hadjieleftheriou. Indexing animated objects using spatio-temporal access methods. *Knowledge and Data Engineering*, 13(5):758–777, 2001.

[LJF94] King-Ip Lin, H. V. Jagadish, and Christos Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB Journal: Very Large Data Bases*, 3(4):517–542, 1994.

[LR94] Ming-Ling Lo and Chinya V. Ravishankar. Spatial joins using seeded trees. In *ACM SIGMOD Conference on Management of Data*, pages 209–220, Minneapolis, Minnesota, USA, 1994.

[LR96] Ming-Ling Lo and Chinya Ravishankar. Spatial hash-joins. In *ACM SIGMOD Conference on Management of Data*, pages 247–258, Montreal, Canada, 1996.

[LS89] D B Lomet and B Salzberg. The hB-tree: A robust multiattribute search structure. In *5th IEEE Conference on Data Engeneering*, pages 296–304, 1989.

[LS90] David B. Lomet and Betty Salzberg. The hB-tree: a multiattribute indexing method with good guaranteed performance. *ACM Trans. Database Syst.*, 15(4):625–658, 1990.

[ME92] Priti Mishra and Margaret H. Eich. Join processing in relational databases. *ACM Comput. Surv.*, 24(1):63–113, 1992.

[ME94] David M. Mark and Max J. Egenhofer. Modeling spatial relations between lines and regions: Combining formal mathematical models and human subjects testing. *Cartography and Geographical Information Systems*, 21(3):195–212, 1994.

[MGA03] Mohamed F. Mokbel, Thanaa M. Ghanem, and Walid G. Aref. Spatio-temporal access methods. *IEEE Data Engineering Bulletin*, 26(2):40–49, 2003.

[MKY81] T. H. Merrett, Yahiko Kambayashi, and H. Yasuura. Scheduling of page-fetches in join operations. In *Proceedings of the 7th international conference on Very large data bases*, pages 488–498, 1981.

[MP03] Nikos Mamoulis and Dimitris Papadias. Slot index spatial join. *IEEE Transactions on Knowledge and Data Engineering*, 15(1):211–231, 2003.

[NHS84] J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38–71, 1984.

[NST98] Mario A. Nascimento, Jefferson R. O. Silva, and Yannis Theodoridis. Access structures for moving points. Technical Report TR–33, TIME CENTER, 1998.

[NST99] Mario A. Nascimento, Jefferson R. O. Silva, and Yannis Theodoridis. Evaluation of access structures for discretely moving points. In *Proceedings of the International Workshop on Spatio-Temporal Database Management (STDBM '99)*, pages 171–188, London, UK, 1999. Springer-Verlag.

[NW97] Gabriele Neyer and Peter Widmayer. Singularities make spatial join scheduling hard. In *ISAAC '97: Proceedings of the 8th International Symposium on Algorithms and Computation*, pages 293–302, London, UK, 1997. Springer-Verlag.

[PD96] Jignesh M. Patel and David J. DeWitt. Partition based spatial-merge join. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 259–270, New York, NY, USA, 1996. ACM Press.

[PI85] Sakti Pramanik and David Ittner. Use of graph-theoretic models for optimal relational database accesses to perform join. *ACM Trans. Database Syst.*, 10(1):57–74, 1985.

[PJT00] Dieter Pfoser, Christian S. Jensen, and Yannis Theodoridis. Novel approaches in query processing for moving object trajectories. In *The VLDB Journal*, pages 395–406, 2000.

[PLM01] Kriengkrai Porkaew, Iosif Lazaridis, and Sharad Mehrotra. Querying mobile objects in spatio-temporal databases. In *SSTD '01: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pages 59–78, London, UK, 2001. Springer-Verlag.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[PSTW93] Bernd-Uwe Pagel, Hans-Werner Six, Heinrich Toben, and Peter Widmayer. Towards an analysis of range query performance in spatial data structures. In *PODS '93: Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 214–221, New York, NY, USA, 1993. ACM Press.

[PT98] Dieter Pfoser and Nectaria Tryfona. Requirements, definitions, and notations for spatio-temporal application environments. In *Proceedings of the 6th ACM International Symposium on Advances in Geographic Information Systems (GIS'98)*, pages 124–130. ACM Press, 1998.

[RKV95] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. Nearest neighbor queries. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 71–79, New York, NY, USA, 1995. ACM Press.

[Rob81] J. T. Robinson. The K-D-B-tree: A search structure for large multidimensional dynamics indexes. In *ACM SIGMOD Conference on Management of Data*, pages 10–18. ACM, 1981.

[Sam95] Hanan Samet. Spatial data structures. In Won Kim, editor, *Modern Database Systems: The Object Model, Interoperability*, pages 361–385. Addison Wesley/ACM Press, Reading MA, 1995.

[SC03] Shashi Shekhar and Sanjay Chawla. *Spatial Databases: A Tour*. Prentice Hall, 2003.

[Sch05] Markus Schneider. Evaluation of spatio-temporal predicates on moving objects. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 516–517, Washington, DC, USA, 2005. IEEE Computer Society.

[SHPFT05] Jeong Seung-Hyun, W. Paton, A. A. Fernandes, and Griffiths Tony. An experimental performance evaluation of spatio-temporal join strategies. *Transactions in GIS*, pages 129–156, March 2005.

[SK90] Bernhard Seeger and Hans-Peter Kriegel. The buddy-tree: An efficient and robust access method for spatial data base systems. In *16th Conference on Very Large Data Bases*, pages 590–601, Brisbane, Australia, 1990.

[SRF87] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The r^+-tree: A dynamic index for multi-dimensional objects. In *13th Conference on Very Large Data Bases*, pages 507–518, Brighton, England, 1987.

[SRF97] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. Multidimensional access methods. trees have grown everywhere. In *23rd Conference on Very Large Data Bases*, pages 13–14, Athens, Greece, 1997.

[ST99] Betty Salzberg and Vassilis J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2):158–221, 1999.

[TP00] Yufei Tao and Dimitris Papadias. MV3R-tree: A spatio-temporal access method for timestamp and interval queries. In *Technical Report HKUST-CS00-06*. Department of Computer Science, Hon Kong University of Science Technology, Hon Kong, 2000.

[TP01a] Yufei Tao and Dimitris Papadias. Efficient historical R-tree. In *SSDBM International Conference on Scientific and Statical Database Management*, pages 223–232, 2001.

[TP01b] Yufei Tao and Dimitris Papadias. MV3R-tree: A spatio-temporal access method for timestamp and interval queries. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 431–440, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[TPZ02] Yufei Tao, Dimitris Papadias, and Jun Zhang. Cost models for overlapping and multiversion structures. *ACM Trans. Database Syst.*, 27(3):299–342, 2002.

[TS96] Yannis Theodoridis and Timos Sellis. A model for the prediction of R-tree performance. In *Proceedings of the fifteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (PODS '96)*, pages 161–171, New York, NY, USA, 1996. ACM Press.

[TSN99] Yannis Theodoridis, Jefferson R. O. Silva, and Mario A. Nascimento. On the generation of spatiotemporal datasets. In *Proceedings of the 6th International Symposium on Advances in Spatial Databases (SSD '99)*, pages 147–164. Springer-Verlag, 1999.

[TSPM98] Yannis Theodoridis, Timos K. Sellis, Apostolos Papadopoulos, and Yannis Manolopoulos. Specifications for efficient indexing in spatiotemporal databases. In *IEEE Proceedings of the 10th International Conference on Scientific and Statistical Database Management*, pages 123–132, 1998.

[TSS98a] Yannis Theodoridis, E Stefanakis, and Timos Sellis. Cost models for join queries in spatial databases. In *14th IEEE Conference on Data Engineering (ICDE)*, 1998.

[TSS98b] Yannis Theodoridis, Emmanuel Stefanakis, and Timos K. Sellis. Cost models for join queries in spatial databases. In *ICDE*, pages 476–483, 1998.

[TSS00] Yannis Theodoridis, Emmanuel Stefanakis, and Timos Sellis. Efficient cost models for spatial queries using R-Trees. *IEEE Transactions on Knowledge and Data Engineering*, 12(1):19–32, 2000.

[TVS96] Yannis Theodoridis, Michalis Vazirgiannis, and Timos K. Sellis. Spatio-temporal indexing for large multimedia applications. In *Proceedings of the 1996 International Conference on Multimedia Computing and Systems (ICMCS '96)*, pages 441–448, Washington, DC, USA, 1996. IEEE Computer Society.

[Val87] Patrick Valduriez. Join indices. *ACM Trans. Database Syst.*, 12(2):218–246, 1987.

[Wor05] Michael Worbys. Event-oriented approaches to geographic phenomena. *International Journal of Geographical Information Science*, 19(1):1–28, 2005.

[XHL90] X Xu, J Han, and W Lu. RT-tree: An improved R-tree index structure for spatio-temporal database. In *4th International Symposium on Spatial Data Handling*, pages 1040–1049, 1990.