Erweiterungen des R-Baums für räumliche Datenbankanfragen

Der R*-Baum

Patrick Schulz & Simon Hötten

Seminar Geodatenbanken
Dozent: Prof. Dr.-Ing. Jan-Henrik Haunert
Institut für Geoinformatik und Fernerkundung
Universität Osnabrück
Sommersemester 2015

Schlüsselwörter: Geodatenbanken, R*, Spatial Access, Index, R-Tree

1 Motivation

Herkömmliche eindimensionale Indexstrukturen bieten keine Möglichkeiten, mehrdimensionale räumliche Daten effizient zu durchsuchen. Die Reduzierung auf Punkte, um Objekte mit Point access methods (PAM) abzufragen, ist mit gewissen Einbußen möglich, aber insbesondere für komplexere Anfragen unzureichend. Der 1984 von Guttman entwickelte R-Baum (Guttman, 1984) versucht dieses Problem zu lösen, in dem der Index direkt auf den räumlichen Eigenschaften basiert. Mittlerweile existieren unzählige Varianten und Verwandte des R-Baums, dessen Einsatzgebiet weit über die klassische Geoinformatik hinaus geht.

Eine dieser Varianten ist der R*-Baum, welcher die (teils unbegründeten) Annahmen in der ursprünglichen Veröffentlichung hinterfragt und so die Datenstruktur weiter optimiert. Im Folgenden gehen wir auf die Verfahren und Eigenheiten des regulären R-, als auch des R*-Baums ein, stellen allgemeine Optimierungskriterien auf und schließen mit einem Vergleich.

Im Prinzip ist der R*-Baum für n-dimensionale Daten geeignet. Diese Arbeit beschränkt sich allerdings auf Geodaten, insbesondere sind alle Beispiele im zwei dimensionalen Raum. Hier liegt auch das Haupteinsatzgebiet von R*-Bäumen. Für höher-dimensionale Daten sind andere Indexstrukturen, wie der X-Baum, zu bevorzugen (vgl. Berchtold u. a., 1996, S. 28-29).

2 Prinzipien eines R-Baums

Der R-Baum ist eine räumliche Indexstruktur für effiziente Bereichsabfragen von Geodaten im n-dimensionalen Raum. Der R-Baum teilt in jeder Ebene seiner Struktur die beinhalteten Geometrien der Geodaten in Partitionen auf. Die Gesamtheit der Geometrien einer Partitionen werden durch minimal umschließende Rechtecke (kurz MBR) repräsentiert. Innerhalb einer Partition werden

die Geoemtrien in weitere Partitionen unterteilt, bis die Anzahl der Geometrien den Schwellwert M innerhalb einer Partition nicht mehr überschreitet. Der Vorteil dieser Strukturierung sorgt dafür, dass bei Bereichabfragen - zB. Was befindet sich in der Umgebung von Polygon p? - nicht zwingend alle Geometrien in der Ebene betrachtet werden müssen: Die Abbildung 1 stellt einen solchen R-Baum dar, wo bei einer Bereichsabfrage im besten Fall nach der ersten Entscheidungsebene die Hälfte des Geometriebestandes nicht mehr betrachtet werden muss. Wenn die MBR einer Partition sich nicht in der Umgebung von Polygon p befindet, so werden auch alle darunterliegenden Ebenen der Partition nicht mehr behandelt. Die Verwendung von achsparallelen MBR sorgt für effiziente Verschneidungs-Abfragen. Durch das alleinige Vergleichen der Koordinaten kann ermittelt werden, ob zwei MBR sich schneiden oder nicht. Die Geometrien selbst werden ebenfalls durch MBR repräsentiert, sind allerdings zur Bewahrung der Übersichtlichkeit in der Abbildung 1 nicht dargestellt.

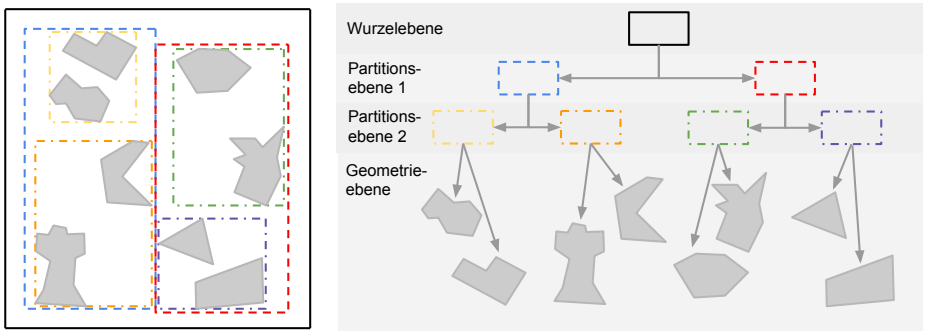

Abb. 1. Beispiel für einen R-Baum. Links: Verteilung der Polygone und Partitionen im zweidimensionalen Raum. Rechts: Die dazugehörige Baumstruktur. (M=3, m=2)

Der Befüllen eines R-Baums verläuft iterativ, wo jedes einzelne Polygon separat nach einem Schema eingefügt wird. Der komplette Prozess des Einfügens lässt sich in zwei Methoden beschreiben, der Author bezeichnet diese Methoden als ChooseSubtree() und QuadraticSplit(). Beide Methoden werden in den nächsten Absätzen beschrieben. Abhängig in welcher Reihenfolge die Polygone in den R-Baum eingesetzt werden kommt es zu verschiedenen Verteilungen innerhalb des R-Baumes. Diese unterschiedlichen Verteilungsmöglichkeiten sind nicht immer ideal und sorgen für weniger effizientere Abfragen. Eie Neuverteilung der Polygone würde zu einer potenziell besseren Performance führen. Die Polygone in der Abbildung 1 sind im R-Baum zur Veranschaulichung ideal verteilt. Eine Lösung für dieses Problem wird in Kapitel Abschnitt 4.3 behandelt und ist Teil der Verbesserungen im R*-Baum.

2.1 ChooseSubTree()

ChooseSubTree() ist eine rekursive Methode, die zu Beginn des Einfügevorgangs aufgerufen wird. Für die einzufügende Geometrie ist in der aktuellen Baum-Ebene die passende Partition zu finden, in dem die Goemetrie daraufhin zugeordnet werden soll. Der Vorgang wird in der darunterliegenden Ebene wiederholt bis die eigentliche Geometrie-Ebene erreicht wird. Bei der Auswahl der geeigneten Partition wird im R-Baum nur der potenzielle Flächenzuwachs des MBR betrachtet. Die Geometrie wird der Partition zugewiesen, dessen MBR den geringsten Flächenzuwachs erfährt. Lässt sich durch das erwähnte Kriterium kein eindeutiger Favorit ermitteln, wird danach entschieden, welches MBR gesamtflächenmäßig kleiner ist. Abbildung 2 zeigt eine Situation in der untersten Baumebene mit zwei Partitionen, wo verglichen werden muss, welches der beiden Partitionen (rot oder blau) für das hellgraue Polygone am geeignetesten ist. Sobald die Geometrieebene erreicht wird, endet an dieser Stelle die ChooseSubTree()-Methode und die Geometrie wird in die erreichte Partition abgelegt. Abschließend wird überprüft, ob die neue Anzahl an Geometrien innerhalb der Partition den Wert M überschritten hat. Wenn jenes der Fall ist, wird die QuadraticSplit()-Methode aufgerufen, ansonsten ist der Einfüge-Prozess abgeschlossen.

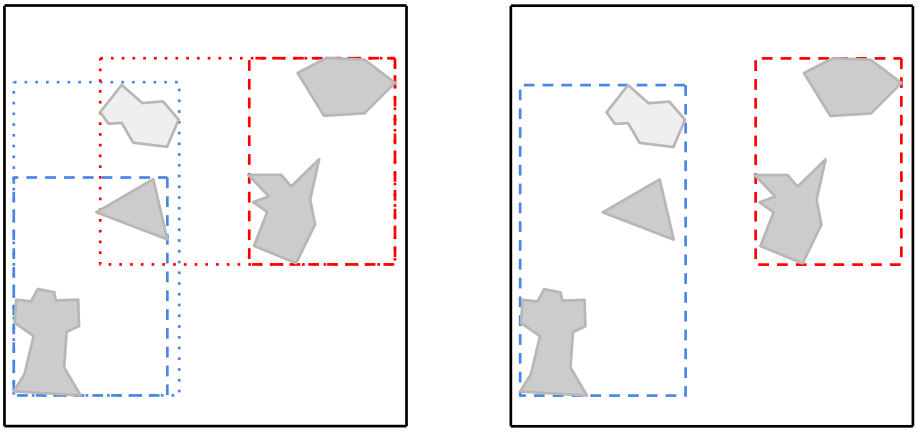

Abb. 2. ChooseSubTree()-Szenario. Links: Vergleich des Flächenzuwachses der Partitionen bei Aufnahme des Polygons. Rechts: hellgraues Polygon wird der blauen Partition zugeordnet, die den geringeren Flächenzuwachs erfährt. (M=4, m=2)

2.2 QuadraticSplit()

Die QuadraticSplit() kümmert sich darum, dass überfüllte Partitionen nach dem Einfügen von Geometrien in zwei neue Partitionen aufgeteilt wird. Zuerst

werden zwei Geometrien aus der Menge der Partition ermittelt, dessen MBR den größten eindimensionalen Abstand voneinander aufweisen. In Abbildung 3 links ist das gesuchte Polygonpaar blau und grün markiert. Beide Polygone repräsentieren nun jeweils eine neue Partition. Abschließend werden die restlichen Polygone auf die beiden neuen Partitionen verteilt. Dabei kommt es zum selben Einfügeprozess, wie man es bereits von der ChooseSubTree()-Methode kennt. Die Reihenfolge, welches Polygon zuerst wieder eingefügt wird, wird durch die Differenz des potenziellen Flächenzuwachs der beiden neuen Partitionen ermittelt. Je höher die Differenz, desto früher wird das betroffene Polygon eingefügt. Da nach jedem Einfügevorgang werden die verbliebenen Polygone neu sortiert, da nach jedem Einfügen die erwähnte Differenz sich verändern kann. Wenn die Summe der verbliebenen Polygone und die Anzahl Polygone einer Partiotion den Wert m ergibt, dann werden alle verbliebene Polygone der betroffenen Partition zugeordnet, um die Anforderung der Mindestpolygonanzahl m zu erhalten und sich damit eine ungleichgewichtete Verteilung zwischen den beiden neuen Partitionen vermeiden lässt. Aus der Logik kann und darf die Variable m nie größer M/2 sein. Die beschriebene Zwangszuordnung ist ebenfalls eine Problemquelle für ungeschickte Zuordnungen. Nach der Verteilung der Geometrien auf die beiden neuen Partitionen wird abschließend in der nächsthöheren Partitionsebene auf eine Überfüllung überprüft. Der Splitvorgang propagiert sich nach oben und wird in der nächsthöhern Ebene ebenfalls durchgeführt, wenn die Partitionsauteilung in der vorherigen Ebene zu einer Überfüllung in der jetzigen Ebene geführt hat.

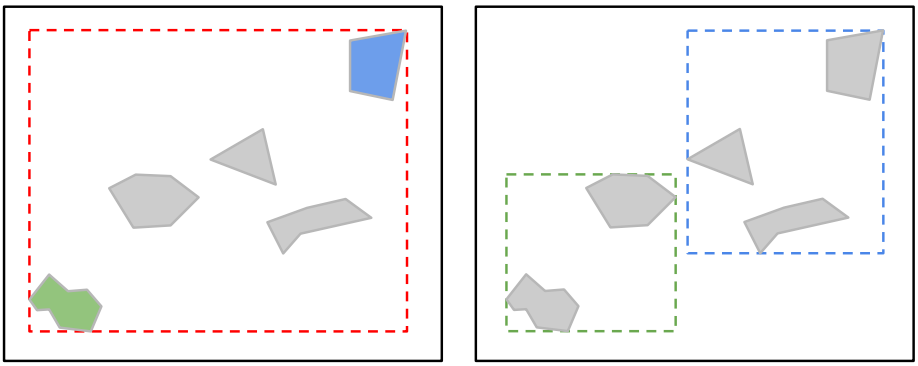

Abb. 3. Split()-Szenario. Links: Überfüllte Partition und gewähltes Polygonpaar, welche eine neue Partition repräsentieren. Rechts: Aufteilung der Polygone in die zwei neuen Partitionen. (M=4, m=2)

Der Grund, wieso die Methode QuadraticSplit() heißt und nicht einfach Split(), liegt daran, dass verschiedene Umsetzungen für die Aufteilung existieren, die eine Unterschiedliche Performance aufweisen. QuadraticSplit() deutet darauf hin, dass

die Methode abhängig von der Anzahl Polygone eine quadratische Laufzeit besitzt. Neben dem QuadraticSplit() existieren noch der LinearSplit(), CubicSplit() und der GreenesSplit(). Der LinearSplit() unterscheidet sich vom QuadraticSplit() nur darin, dass dieser die Sortierung der ausstehenden Polygone nur einmal am Anfang durchführt. Dies sorgt zwar für eine bessere Laufzeit, die daraus resultierenden Aufteilungen werden im Vergleich zum QuadraticSplit() allerdings nie besser sein. (Aussage erwartet wahrscheinlich Beweis) Der CubicSplit hingegen verspricht, dass nach dem Flächenkriterium die besten Aufteilungen generiert wird auf Kosten der längeren Bearbeitungszeit. Der QuadraticSplit() ist somit ein Kompromiss zwischen Effizienz und Qualität. Dennoch gibt es Situationen, die zu ungünstigen Verteilungen führen. Ein markantes Beispiel ist in Abbildung 4 dargestellt: Die Kombination von kleinen bzw. schmalen Geometrien, die zwar weit entfernt von einanderliegen und doch in ihrer Lage auf identischer Anhöhe liegen, sorgen für sehr schmale, nadelartige Partitionen, dessen MBR mit einer sehr hohen Wahrscheilichkeit mit der anderen Partition überlappen.

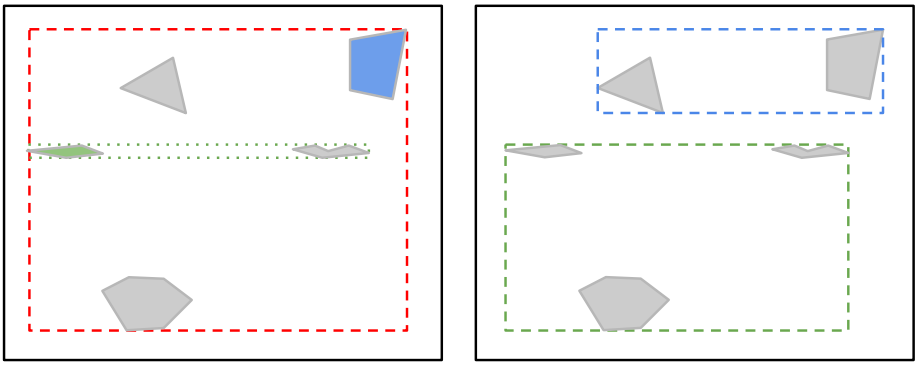

Abb. 4. Problem der kleinen Objekte beim QuadraticSplit. Links: Durch die selbe Y-Achsen-Anhöhe zwischen den beiden schmalen Objekten entsteht ein sehr schmales MBR. Rechts: Ergebnis des Splits auf Basis des initalen, schmalen MBRs (M=3, m=2)

2.3 GreenesSplit()

In einer der wissenschaftlichen Veröffentlichungen von "Greene"wird ein alternatives Verfahren für die bisher bekannten Split-Methoden vorgestellt. Die Idee besteht darin, die Polygone nach der jeweiligen Achse zu sortieren, in der die größte Ausbreitung ermittelt wird. Auf Basis dieser Sortierung hat man nun eine Reihe von möglichen Trenn-Lösungen, aus denen dann der Algorithmus den Favoriten ermittelt und übernimmt. Das Ergebniss in Abbildung 5 rechts stellt den Favoriten dar mit der gerinsten Flächensumme. Man beachte, dass es zwei Möglichkeiten gibt, die Polygone nach einer Achse zu sortieren, nämlich

nach ihrem Minimal- oder ihrem Maximalwert. In der Abbildung 5 wird nur die Sortierung nach dem Minimalwert dargestellt. Bei der Wahl des Favoriten müssen die Trenn-Lösungen beider Sortierungen betrachtet werden. Neben dem besagten Flächenkriterium besitzt der GreenesSplit() im Vergleich zu den anderen Split-Algorithmen ein zusätzliches, geometrisches Kriterium: die Verteilung der Geometrien im Raum und die daraus ermittelte Trenn-Achse - wobei es auch hier zu Situationen kommen kann, in der sich der GreenesSplit() für die "falscheÄchse entscheidet. Es existieren allerdings noch weitere Kriterien, die im nächsten Kapitel aufgezählt werden, womit man die Verteilung im R-Baum optimieren könnte.

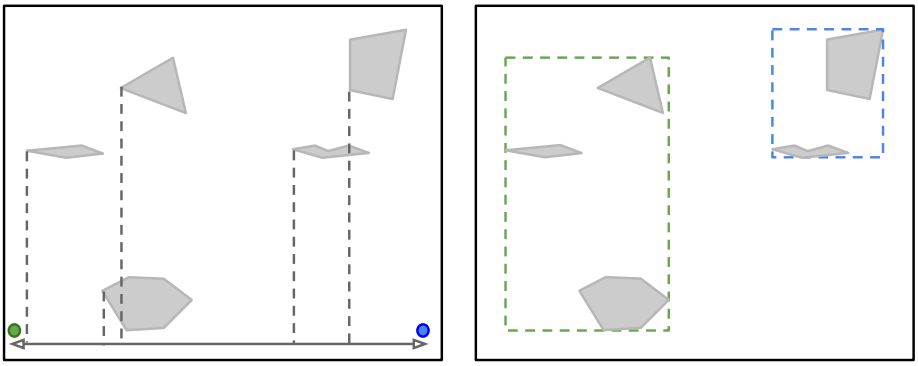

Abb. 5. Beispiel für GreenesSplit. Links: Sortierung der Geometrien nach der gewählten Trenn-Achse. Rechts: Resultierende Ergebnis aus der Sortierung. (M=3, m=2)

3 Optimierungskriterien

Bei dem herkömmlichen R-Baum wird, sowohl beim Hinzufügen neuer Elemente als auch beim Split, lediglich die Fläche der umschließenden Rechtecke minimiert (vgl. Guttman, 1984, S. 50-51). Einige der daraus resultierenden Probleme wurden bereits im vorherigen Abschnitt dargelegt. Im Folgenden werden weitere mögliche Optimierungen und ihre Wechselwirkungen aufgeführt. Ein R-Baum muss mit unterschiedlichen Geometrien und Anfragen umgehen können, daher wirken sich einige Kriterien in einigen Situationen stärker aus als andere.

Flächenausnutzung maximieren

Die Fläche, welche von dem umschließenden Rechteck, aber nicht von den in ihm enthaltenen Rechtecken, überdeckt wird, soll minimiert werden. Es soll also möglichst wenig Platz „verschwendet“ werden. (vgl. Beckmann u. a., 1990, S. 323)

Überlappung minimieren

Die Überlappung der umschließenden Rechtecke soll minimiert werden. Dadurch müssen ebenfalls weniger Pfade im Baum traversiert werden. Liegt ein angefragter Punkt beispielsweise in einer Region, in der sich viele Rechtecke Überschneiden, müssen alle Möglichkeiten weiter verfolgt werden, was zu erhöhtem Rechenaufwand führt.

Summe der Kantenlänge minimieren

Die Summe der Kantenlänge der Verzeichnisrechtecke (*„margin“*) soll möglichst klein sein. Quadrate werden also bevorzugt. Da Quadrate auf den jeweils höheren Ebenen im Baum besser zusammengefasst werden können, reduziert sich so die benötigte Fläche. Außerdem profitieren Anfragen mit großen, quadratischen Elementen von dieser Optimierung. (vgl. ebd., S. 323)

Speichernutzung maximieren

Eine geringe Höhe des Baumes wirkt sich positiv auf die Kosten einer Abfrage aus. Das kann durch eine möglichst gleichmäßige Verteilung der Blattknoten erreicht werden. Insbesondere für große Abfragerechtecke ist dies relevant, da hier, auch abgesehen von den ersten drei genannten Optimierungen, mehrere Pfade traversiert werden müssen. (vgl. ebd., S. 323-324)

Wechselwirkungen

Um die Flächenausnutzung zu maximieren und die Überlappung zu minimieren, bedarf es einer größeren Freiheit bei der Wahl der Formen und der Anzahl an Rechtecken pro Knoten. Die Kriterien stehen also in Konkurrenz mit einer geringen Kantenlänge und hohen Speicherausnutzung. Auf der anderen Seite können quadratischere Rechtecke besser zusammengefasst werden, was sich wiederum positiv auf die Speichernutzung auswirkt. (vgl. ebd., S. 323-324)

4 Der R*-Baum

In der bisher vorgestellten R-Baum-Variante war das einzigste Optimierungskriterium die Minierung der Flächeninhalte innerhalb der Partitionen. In Abschnitt 3 sind weitere potenzielle Kriterien gelistet, die während des Einfüge-Verfahrens berücksichtigt werden könnten. Diese besagten Kriterien wurden von den Authoren auf verschiedenste Weise kombiniert, untersucht und evaluiert. Dabei entstand aus den untersuchten Kombinationen die Methoden für den verbesserten R*-Baum, die auf Hinblick der Query-Performance die besten Ergebnisse liefern. (vgl. Kriegel u. a., 2008, S. 325)

4.1 ChooseSubTree()

Die ChooseSubTree()-Methode vom R*-Baum unterscheidet sich nicht erheblich von der bisher bekannten R-Baum-Variante. Der Unterschied zwischen den beiden rekursiven Methoden befindet sich im Entscheidungskriterium in der letzten Partitionsebene. Hier wird anstelle des Flächenkriteriums zuerst das Überlappungskriterium berücksichtigt. Der Grund, wieso nicht in allen Partitionsebenen nach dem Überlappungskriterium berücksichtigt werden, ist der Rechenaufwand, der für das Überpfüfen von Überlappungen benötigt wird. Der Aufwand ist quadratisch in Relation zu den vorhandenen Partitionen, zwischen denen gewählt werden muss. Im Falle einer hohen Anzahl von zu vergleichenden Partitionen steigt der Rechenaufwand erheblich. Für dieses Problem schlagen die Autoren vor, dass eine Vorauswahl der Partitionen stattfindet, in der nur die n-nächsten Partition als Kandidaten für die einzufügende Geometrie in Frage kommen. Determiniert werden diese Kandidaten durch das bisher bekannte Flächenkriterium. Anschließend wird nur ungefähr"das Überlappungskriterium ermittelt, da hierfür nur die Überlappung zwischen den Kandidaten betrachtet wird.

4.2 Split()

Die Split()-Methode besitzt die selbe Herangehensweise, wie der bereits erwähnte GreenesSplit() in Abschnitt 2.3. In beiden Methoden wird eine Trenn-Achse gewählt, die Partitionen sortiert und anschließend ein passender Split aus den Sortierungen gewählt. Der Unterschied liegt hierbei wieder in den Entscheidungskriterien. Bei der Wahl der Trenn-Achse wird für jede Achse die Menge aller Trenn-Lösungen aus den beiden bekannten Sortierungsmöglichkeiten erzeugt (siehe Abbildung 6). Aus der Menge wird dann die Summe der Kantenlängen der Partitionen ermittelt. Als Trenn-Achse gewählt wird dann die Achse mit der geringsten Summe. Anschließend wird aus der gegebenen Lösungsmenge der Favorit ausgewählt, welche die geringste Überlappung zwischen den neuen Partitionen erzeugt. Ergibt sich durch die Betrachtung der Überlappung kein Favorit, wird bei einem Gleichstand die Lösung mit der geringeren Flächensumme ausgewählt. Das Problem der Überfüllung kann wie im R-Baum in die nächsthöhere Partitionsebene hochpropagieren, wo die Split()-Methode dann ggf. erneut ausgeführt wird.
Die Autoren haben neben den geometrischen Entscheidungskriterien auch mit Parametrisierung der Variablen M und m experimentiert und sind zu dem Ergebnis gekommen, dass die besten Splits entstehen, wenn m ungefähr 40 Prozent von M entspricht. (vgl. Kriegel u. a., 2008, S. 325)

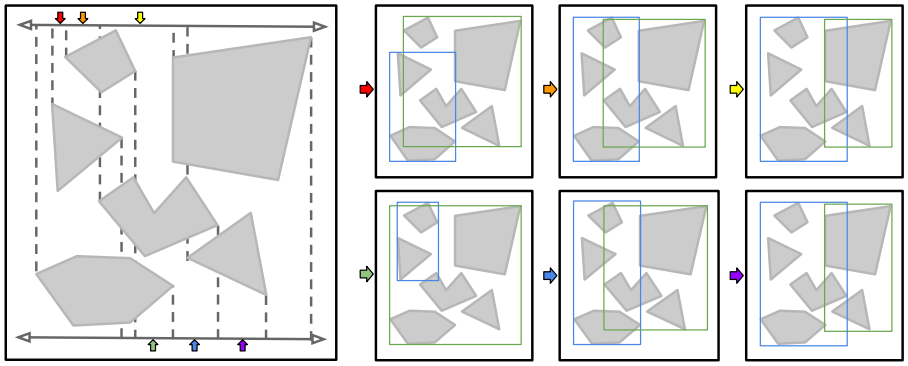

Abb. 6. Aufzählung der Trenn-Lösungen für die waagerechte Achse. Die bunten Pfeile auf den grauen Sortierebene (links) verweisen auf die darausentstehenden Trenn-Lösungen. (M=5, m=2)

4.3 ForcedReinsert()

Im Abschnitt 2 ist das Problem des iterativen Einsetzverfahrens bereits angesprochen worden. Für dieses Problem ist der Ansatz einer Neu-Verteilung von gewählten Geometrien in die Baumstruktur eine natürliche Lösung um ungeschickte Verteilungen zu korrigieren, die durch eine ungünstige Einfügereihenfolge entstanden sind. Die Autoren bestätigen ihre Annahme durch einen Test, indem sie aus einem bestehenden, befüllten R-Baum die hälfte der Datensätze löschen und wieder in den Baum einfügen. Die daraus entstandene Struktur innerhalb vom R-Baum führte zu einer 50 Prozent verbesserten Query-Performance im Vergleich zur vorherigen Struktur. Die ReInsert()-Methode wird nach jedem Einfügeprozess aufgerufen, sobald eine Partition überfüllt ist. Wenn die Methode das Überfüllungs-Problem nicht lösen konnte, wird erst dann die Split()-Funktion aufgerufen.
letzter Absatz mit passender Beschreibung + Abbildung

5 Fazit

Zunächst lässt sich festhalten, dass der R*-Baum alle vorgestellten Optimierungskriterien berücksichtigt. Das hat einen etwas erhöhten Implementierungsaufwand gegenüber herkömmlichen R-Bäumen zur Folge.

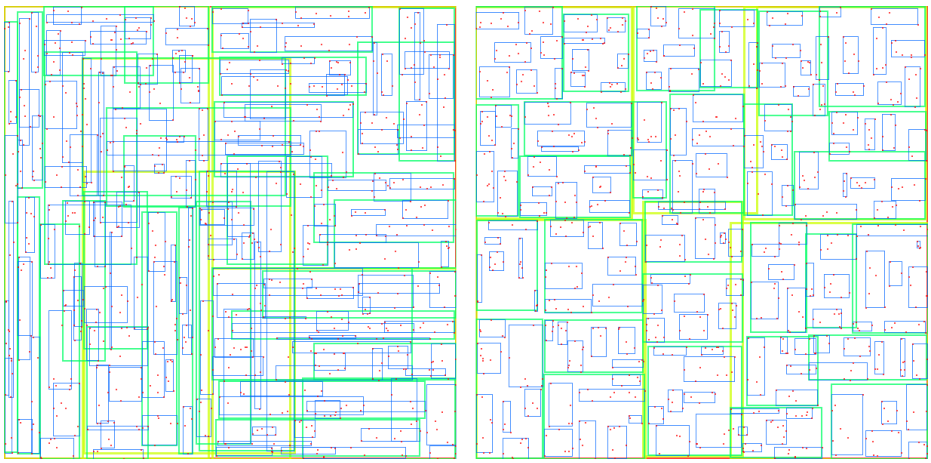

Abb. 7. Quadratischer- (links) und R*-Split (rechts) im Vergleich. (Bildquelle: `https://github.com/davidmoten/rtree`)

Der R*-Baum betreibt sehr hohen Aufwand beim Hinzufügen und Löschen von Elementen, um eine gute Struktur zu bewahren. Abbildung 7 zeigt den gleichen Datensatz (mit gleicher Einfügereihenfolge), aber unterschiedlichen Split-Verfahren. Das Beispiel zeigt, wie positiv sich der Mehraufwand auswirkt. Mit R*-Split überschneiden sich die Verzeichnisrechtecke deutlich weniger, was schnellere Abfragen ermöglichen sollte und in den folgenden Benchmarks bestätigt wird.

Benchmarks

Die folgenden Benchmarks wurden von Beckmann, et.al. im Rahmen der ursprünglichen R*-Baum Veröffentlichung durchgeführt. Verglichen werden der originale R-Baum mit linearem, quadratischem und Greene-Split und der hier beschriebene R*-Baum. Die Parameter sind die für die jeweiligen Implementationen optimal gewählt[1] (vgl. Beckmann u. a., 1990, S. 328).

Als Testdaten wurden fünf über verschiedene Verteilungen automatisch generierte Datensätze und ein realer Datensatz aus Höhenlinien gewählt (für eine genaue Beschreibung siehe ebd., S. 328). Die Daten wurden mit jeweils sieben verschiedenen Anfragen durchsucht und die Speicherzugriffe gemessen. Die Anfragen bestanden aus:

Point Punkt-Anfrage (1000×, gleich-verteilt im Raum)

Intersection Suche nach allen Daten, welche die unterschiedlich großen Anfragerechtecke schneiden (je 100×, gleich-verteilt) (Größenangaben relativ zu der Größe des Testdatensatzes)

[1] $m = 20\%$ (von M maximalen Einträgen) für den linearen Split, quadratischer Split: $m = 40\%$, Greene's-Split: $m = 40\%$ und R*-Split: $m = 40\%$

Contains Suche nach allen Daten, die in den Anfragerechtecken komplett enthalten sind (ebenfalls je 100×, gleich-verteilt) (Größenangaben relativ zu der Größe des Testdatensatzes)

Tabelle 1. Vergleich der Bäume bei den verschiedenen Anfragen, gemittelt über die sechs Testdatensätze. Alle Angaben (bis auf Insert) sind relativ zu den Speicherzugriffen des R*-Baums. Insert zeigt die durchschnittlichen Speicherzugriffe beim Einfügen.

	Point	Intersection				Contains		Insert
		0.001%	0.01%	0.1%	1%	0.001%	0.01%	
linear	251.9	242.2	231.1	189.8	152.1	256.5	274.1	12.63
quad.	135.3	132.4	132.8	126.4	117.6	131.3	137.0	7.76
Greene	148.7	143.9	148.0	137.7	121.3	145.0	155.2	7.67
R*	100	100	100	100	100	100	100	6.13

Tabelle 2. Performance der Bäume im Vergleich, hier gemittelt über alle sieben Anfragetypen. Alle Angaben sind relativ zu den Speicherzugriffen des R*-Baums.

	Gauß-Vert.	Cluster	Mix-Gleichvert.	Parzelliert	Höhenlinien	Gleichvert.
linear	164.3	216.0	308.1	247.2	227.2	206.6
quad.	112.9	153.9	121.8	128.1	144.5	121.1
Greene	123.1	147.1	115.5	192.4	144.2	134.8
R*	100	100	100	100	100	100

Tabelle 1 und Tabelle 2 zeigen die, auf unterschiedliche Art aggregierten, Testergebnisse. Die bessere Performance des R*-Baums ist für alle verschiedenen Anfragen und Testdaten klar ersichtlich. Anzumerken ist, dass lediglich zwei dimensionale Daten verwendet wurden (328 Beckmann u. a., 1990, vgl.). Das entspricht in den meisten Fällen den Anforderungen an Geodatenbanken. Für andere Anwendungsfälle wären weitere Tests notwendig.

Interessant ist, dass trotz der Verwendung von Forced Reinsert der R*-Baum auch beim Einfügen von neuen Elementen schneller ist als herkömmliche Varianten.

Weiterentwicklungen

Die Effizienz des R*-Baums nimmt ab fünf Dimensionen rapide ab (vgl. Berchtold u. a., 1996, S. 29). Um auch höher dimensionalen Daten gerecht zu werden, existieren daher zahlreiche Weiterentwicklungen. Dazu gehört, wie eingangs erwähnt, der X-Baum, welcher darauf ausgelegt ist auch in höheren Dimensionen Überlappungen zu vermeiden (vgl. ebd.). Andere Indizes bilden Näherungen

der tatsächlichen Daten und führen Anfragen zunächst auf diesen aus („*Vector Approximation*“, siehe Gibas und Ferhatosmanoglu, 2008 oder Daoudi u. a., 2008). So wird der Zugriff auf Daten mit über 100 Dimensionen vergleichsweise effizient ermöglicht (vgl. ebd.).

Anhang

Abkürzungsverzeichnis

m	Variable: minimale Anzahl der Geometrien in einer Partition
M	Variable: maximale Anzahl der Geometrien in einer Partition
MBR	Minimum bounding Rectangle
SAM	Spatial access methods
PAM	Point access methods

Literatur

Beckmann, Norbert, Hans-Peter Kriegel, Ralf Schneider und Bernhard Seeger (1990). „The R*-tree: An Efficient and Robust Access Method for Points and Rectangles“. In: *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*. SIGMOD '90. Atlantic City, New Jersey, USA: ACM, S. 322–331. DOI: 10.1145/93597.98741 (siehe S. 11, 15 f.).

Berchtold, Stefan, Daniel A. Keim und Hans-Peter Kriegel (1996). „The X-tree: An Index Structure for High-Dimensional Data“. In: *Proceedings of the 22nd VLDB Conference*. Mumbai, India, S. 28–39 (siehe S. 1, 16).

Daoudi, I., S.E. Ouatik, A. El Kharraz, K. Idrissi und D. Aboutajdine (2008). „Vector Approximation based Indexing for High-Dimensional Multimedia Databases“. In: *Engineering Letters* 16.2 (siehe S. 16).

Gibas, MICHAEL und Hakan Ferhatosmanoglu (2008). „High Dimensional Indexing“. In: *Encyclopedia of GIS*. Springer US, S. 502–507. DOI: 10.1007/978-0-387-35973-1_602 (siehe S. 16).

Guttman, Antonin (1984). „R-trees: a dynamic index structure for spatial searching“. In: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. SIGMOD '84. New York, NY, USA: ACM, S. 47–57. DOI: 10.1145/602259.602266 (siehe S. 1, 10).

Kriegel, Hans-Peter, Peter Kunath und Matthias Renz (2008). „R*-Tree“. In: *Encyclopedia of GIS*. Springer US, S. 987–992. DOI: 10.1007/978-0-387-35973-1_1148 (siehe S. 11 f.).