Westfälische Wilhelms-Universität
Münster
Fachbereich Mathematik und Informatik

Effiziente Konstruktion und Aktualisierung von Datenstrukturen für Trajektorien

Abschlussarbeit zur Erlangung des akademischen Grades
Master of Science (M.Sc.)

Author: Michael Josef Paul Beckemeyer
Steinbreede 12
49497 Mettingen
m.beckemeyer@uni-muenster.de

Matrikelnummer: 382 861

Gutachter: Prof. Dr. Jan Vahrenhold

Zweitgutachter: Prof. Dr. Klaus Hinrichs

Münster, 5. Juni 2017

Plagiatserklärung

Hiermit versichere ich, dass die vorliegende Arbeit über *Effiziente Konstruktion und Aktualisierung von Datenstrukturen für Trajektorien* selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

(Ort, Datum) (Unterschrift)

Ich erkläre mich mit einem Abgleich der Arbeit mit anderen Texten zwecks Auffindung von Übereinstimmungen sowie mit einer zu diesem Zweck vorzunehmenden Speicherung der Arbeit in eine Datenbank einverstanden.

(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

1. Einleitung

Zusammen mit der hohen Verfügbarkeit GPS-fähiger Endgeräte und der daraus folgenden großen Menge erfassbarer Daten wächst auch die Notwendigkeit, diese mit effizienten Datenstrukturen und Algorithmen zu verwalten. Räumliche Indexstrukturen, wie z.B. der R-Baum [Gut84], sind in der Lage, Punkte und andere mehrdimensionale Objekte zu speichern und sie so zu organisieren, dass die Ergebnisse für eine Vielzahl verschiedener, räumlicher Anfragetypen schnell gefunden werden können. *Räumliche Trajektorien* sind Folgen $(p_1, t_1), ...(p_n, t_n)$ von räumlichen Punkten und Zeitpunkten, die die Bewegungen eines Objekts durch den Raum angeben. Herkömmliche räumlichen Indexstrukturen können so angepasst werden, dass sie effizienten Zugriff auf solche Trajektorien bieten.

Räumliche Daten werden häufig mit Informationen aus anderen Quellen zusammenlegt, um sie mit einer anwendungsspezifischen Bedeutung zu versehen. Dies geschieht beispielsweise bei Datenbanken für *Points of Interest* (z.B. Sehenswürdigkeiten, lokale Geschäfte), die einem Benutzer Suchen nach Objekten ermöglichen, die sich in seiner Nähe befinden. *Textuelle Trajektorien* [ID16] (auch *symbolische Trajektorien* [GVD15]) sind Folgen von Zeitintervallen, die mit symbolischen Informationen annotiert wurden. Eine textuelle Trajektorie $(I_1, l_1), ..., (I_n, l_n)$ enthält zu jedem Zeitintervall I ein *Label* l, welches die zu diesem Zeitraum aktive Information repräsentiert. Mithilfe von textuellen Trajektorien können räumliche Trajektorien für eine Vielzahl von Anwendungen mit Bedeutung versehen werden. Ein Label kann zum Beispiel die Aktivität eines Benutzers, dessen aktuelles Transportmittel, den Namen der befahrenen Straße oder die Nähe zu einem Point of Interest zu einem gewissen Zeitabschnitt enthalten. Trajektorien, die sowohl aus räumlichen wie auch aus textuellen Trajektorien zusammengesetzt sind, werden *spatio-textuelle Trajektorien* genannt [ID16].

Die Arbeit mit solchen Trajektorien erfordert neue Datenstrukturen, um sowohl den textuellen wie auch den räumlichen Informationsaspekt zu erfassen und effizienten Zugriff bei der Suche zu ermöglichen. Issa und Damiani [ID16] entwickelten zu diesem Zweck die

IRWI-Datenstruktur, einen modifizierten R-Baum, dessen Knoten den textuellen Inhalt der Trajektorien in ihren Teilbäumen indizieren. Die Datenstruktur kann mit komplexen, aus räumlichen und textuellen Kriterien zusammengesetzten Anfragen durchsucht werden um entsprechende Trajektorien effizient zu finden. Der Aufbau eines IRWI-Baums erwies sich allerdings als ineffizient: die Konstruktionszeit für nur wenige Millionen Einträge (einzelne Abschnitte der Trajektorien) betrug schon mehrere Stunden.

In dieser Arbeit werden mehrere Verfahren zur effizienten Konstruktion des IRWI-Baums entwickelt, damit auch sehr große Datenmengen in annehmbarer Zeit unterstützt werden können. Dazu werden existierende, schon von R-Bäumen bekannte, *bulk-loading*-Verfahren beschrieben und für die Datenstruktur angepasst. Mithilfe dieser Algorithmen lässt sich die Konstruktionszeit um mehrere Größenordnungen verbessern, wobei die Qualität der so erhaltenen Datenstruktur nicht unbedingt besser sein muss.

Der Rest dieser Arbeit ist wie folgt gegliedert: Kapitel 2 enthält die Definition der Trajektorien und verwandter Konzepte. Danach (in Kapitel 3) wird eine Übersicht über die für diese Arbeit relevanten, grundlegenden Indexstrukturen gegeben. In Kapitel 4 werden die verschiedenen Algorithmen zur Konstruktion eines R-Baums beschrieben. Kapitel 5 enthält eine ausführliche Erklärung des IRWI-Baums und erweitert die Algorithmen aus Kapitel 4 für diese Datenstruktur. Die Implementierung des IRWI-Baums und der *bulk-loading*-Verfahren wird in Kapitel 6 präsentiert. Kapitel 7 enthält die Beschreibung der durchgeführten Experimente und deren Ergebnisse. Die Arbeit endet mit der Zusammenfassung in Kapitel 8.

2. Begriffe

Dieses Kapitel enthält die Definition von spatio-textuellen Trajektorien und verwandten Konzepten. Das Vokabular ist – mit Ausnahme von Übersetzungen – identisch mit dem von Issa und Damiani [ID16].

Es sei $\mathcal{T}$ die Menge der Zeitpunkte, $\mathcal{S}$ ein Raum und $\mathcal{L}$ die Menge der Labels. Labels sind beliebige semantische Informationen, z.B. repräsentiert als Zeichenketten. Eine *spatio-textuelle Trajektorie* repräsentiert die Bewegung eines Objekts als eine Funktion $tr : \mathcal{T} \to \mathcal{S} \times \mathcal{L}$, die einem Zeitpunkt eine räumliche Koordinate und eine semantische Annotation zuordnet. Für $t \in \mathcal{T}$ ist $tr(t) = (p, l)$, wobei der Punkt $p \in \mathcal{S}$ die Position eines Objekts zum Zeitpunkt t und $l \in \mathcal{L}$ das zu diesem Zeitpunkt aktive Label ist.

Trajektorien werden hier als eine Folge von *Units* $u_1, ..., u_n$ repräsentiert. Zur Trajektorie tr erhält man n Units mit einer gewissen Präzision, indem man die Funktionswerte von tr zu n diskreten Zeitpunkten misst und zwischen diesen Zeitpunkten kontinuierliche Bewegung im Raum (bei konstanter Geschwindigkeit) und ein konstantes Label voraussetzt. Für $1 \leq i \leq n$ ist $u_i = (I_i, l_i, seg_i)$. I_i ist ein Intervall in $\mathcal{T}$, $l_i \in \mathcal{L}$ ist das in diesem Intervall aktive Label und seg_i ist ein Liniensegment aus zwei Punkten in $\mathcal{S}$, welches die Position des Objekts zu Beginn und zum Ende des Zeitintervalls repräsentiert.

Eine im Kontext dieser Arbeit als *einfache Suchanfrage* bezeichnete Suchanfrage ist die Erweiterung einer gewöhnlichen räumlichen Bereichsanfrage auf die semantischen Informationen der Trajektorien. Zur Ergebnismenge einer einfachen Anfrage $q = (I, L, R)$ gehören alle Trajektorien, welche zu einem beliebigen Zeitpunkt im Intervall I im räumlichen Bereich $R \subseteq \mathcal{S}$ liegen und deren Label in der Menge $L \subseteq \mathcal{L}$ enthalten ist. Mit anderen Worten gehört die Trajektorie tr genau dann zum Ergebnis, wenn ein $t \in I$ existiert, sodass für $tr(t) = (p, l)$ die Bedingungen $p \in R$ und $l \in L$ zutreffen. Mit diesen Suchanfragen kann nach Trajektorien gesucht werden, die sich in einem bestimmten Bereich bewegen und dabei eine gewisse Annotation besitzen (z.B. eine Aktivität ausführen).

Eine *sequentielle Suchanfrage* $q = q_1, ..., q_n$ ist eine Folge einfacher Suchanfragen, die von Trajektorien der Reihe nach erfüllt werden müssen. Eine Trajektorie tr erfüllt die Anfrage q genau dann, wenn es n verschiedene Zeitpunkte $t_1 < t_2 < ... < t_n$ gibt, sodass tr zum Zeitpunkt t_i die Anfrage q_i erfüllt. Es handelt es sich also um das logische UND aller einzelnen einfachen Anfragen mit einer zusätzlichen Bedingung an die Reihenfolge. In einer Anwendung, bei der die räumlichen Trajektorien von Kraftfahrzeugen zusammen mit der zu einem Zeitpunkt befahrenen Straße behandelt werden. Die Suche nach

$$([10{:}00\ 11{:}00], \mathcal{L}, \text{Berlin}), ([11{:}00\ 12{:}00], \{\text{A2}, \text{A9}\}, \text{Brandenburg})$$

liefert alle Trajektorien, welche zwischen 10 und 11 Uhr in Berlin verlaufen (mit beliebigem Straßennamen) und im Anschluss entweder die A2 oder die A9 innerhalb Brandenburgs befahren. Die von Issa und Damiani [ID16] entwickelte IRWI-Datenstruktur ist in der Lage, diese Suchen effizient auszuführen, indem die q_i parallel ausgewertet werden.

3. Datenstrukturen

Für das effiziente Suchen nach beliebigen Daten werden Indexstrukturen benötigt, die die Daten effektiv organisieren. Die Konstruktion eines Index soll, in Abhängigkeit von der Anzahl der zu speichernden Objekte, in annehmbarer Zeit geschehen. Nach der Konstruktion soll ein Index schnellen Zugriff auf seine Elemente ermöglichen. Häufig sollen Indizes außerdem dynamisch sein, d.h. auch nachträgliches Einfügen oder Löschen effizient ermöglichen. Viele der in dieser Arbeit betrachteten Indexstrukturen sind Bäume, in denen Daten hierarchisch angeordnet werden. Deshalb soll im Folgenden eine einheitliche Terminologie eingeführt werden.

Ein Baum besteht aus *Knoten*, die durch *Zeiger* (*pointer*) miteinander verbunden sind. Die Knoten, auf die von einem *Elternknoten* (*parent*) verwiesen wird, sind dessen *Kinder* (*children*). In jedem nicht-leeren Baum existiert genau ein Knoten ohne Elternknoten, dies ist die *Wurzel* (*root*); alle anderen Knoten besitzen genau einen Elternknoten. Ein Knoten, der keine Kinder besitzt, ist ein *Blatt*. Alle anderen Knoten werden als *interne Knoten* bezeichnet.

Zu einem Knoten v ist dessen *Tiefe* $depth(v)$ Abstand zur Wurzel des Baums. Die Tiefe der Wurzel ist 0; für jeden anderen Knoten ist $depth(v) = depth(parent(v)) + 1$. Die *Höhe* eines Baumes $height(t)$ ist bestimmt durch die maximale Tiefe seiner Knoten plus 1. Der in v gewurzelte *Teilbaum* besteht aus v selbst und allen Knoten, die über Kindzeiger direkt oder indirekt von v aus erreichbar sind. Die Höhe eines Knotens ist die Höhe des in ihm gewurzelten Teilbaums. Ein Baum mit n Knoten ist *balanciert*, wenn seine Höhe in $\mathcal{O}(\log n)$ liegt. Mithilfe von balancierten Bäumen können Such-, Einfüge- und Löschoperationen für selbst große Datenmengen mit geringer asymptotischer Laufzeit umgesetzt werden.

Bäume werden häufig im internen Speicher als Knoten geringer Größe repräsentiert: in der Regel umfasst ein Knoten nicht mehr als zwei Kindzeiger und nur ein Datenobjekt.

Für Implementierungen im Hauptspeicher moderner Computer sind diese Lösungen ausreichend: die Größe eines Knotens entspricht ungefähr eines vom Prozessor auf einmal gelesenen Datenblocks.[1]

Indexstrukturen für große Datenmengen müssen allerdings aufgrund von zu geringem zur Verfügung stehenden internem Speicherplatz in den externen Speicher (z.B. Festplatten oder netzwerkbasierte Speicherlösungen) ausgelagert werden. Der Zugriff auf einen beliebigen Speicherblock ist in diesen Fällen um viele Größenordnungen langsamer als das Lesen oder Schreiben im internen Speicher. Gleichzeitig sind die hier zum Einsatz kommenden Speicherblöcke um ein Vielfaches größer.[2] Die naive Portierung eines klassischen Binärbaums würde Großteile der zur Verfügung stehenden Kapazitäten ungenutzt lassen. Die Anforderungen an effiziente Datenstrukturen im Sekundärspeicher sind also eine möglichst geringe Anzahl an Speicherzugriffen und eine möglichst gute Ausnutzung der Speicherblöcke.

3.1. Das I/O-Modell

Die Komplexität von Algorithmen wird für gewöhnlich ausgedrückt, indem die Anzahl der Rechenoperationen auf einer Maschine mit *random access memory* betrachtet wird. In diesem Modell kann auf eine beliebige Speicherzelle in geringer konstanter Zeit zugegriffen werden. Zur Bewertung der asymptotischen Laufzeit wird dann die Anzahl der insgesamt ausgeführten Instruktionen herangezogen, wobei Speicherzugriffe und Rechenoperationen in gleicher Zeit ausgeführt werden.[3]

Für große Datenmengen, die nicht in den Hauptspeicher eines Computers passen, muss ein externer Speicher (z.B. eine Festplatte) verwendet werden. Der Lese- und Schreibzugriff auf ein solches Gerät ist um Größenordnungen langsamer als der auf den internen

[1] Die *cache line* des L1-Zwischenspeichers eines Intel Prozessors umfasst gegenwärtig 64 Byte. Eine 64-Bit Prozessorarchitektur benötigt 8 Byte zur Repräsentation eines Zeigers, damit liegt die Mindestgröße eines Binärbaums bei 24 Byte (2 Kindknoten und ein Zeiger auf den *parent*), zuzüglich der Größe eines Datenobjekts. Je nach Natur der Daten wird daher nur wenig Kapazität verschwendet.

[2] Speichersektoren moderner Festplattenlaufwerke sind oft 4 Kilobyte groß. Dies entspricht der Größe einer Speicherseite in gängigen Betriebssystemen wie Linux, Mac OS X oder Windows.

[3] Selbst für geringe Datenmengen ist dieses Modell inzwischen nicht mehr angemessen. Speicherzugriffe, insbesondere solche, die nicht von einem der Zwischenspeicher des Prozessors abgefangen werden, sind um Größenordnungen langsamer als primitive Rechenoperationen.

Hauptspeicher. Im I/O-Modell werden daher nur die Anzahl der Zugriffe auf den externen Speicher gezählt. In dieser Arbeit wird das Modell von Aggarwal und Vitter [AV88] verwendet.

Das Modell umfasst eine Maschine mit einem Hauptspeicher (dem internen Speicher) und einem Sekundärspeicher (dem externen Speicher). Der Transfer von Daten vom Sekundärspeicher in den Hauptspeicher und umgekehrt geschieht in einzelnen Speicherblöcken der Größe B: es werden immer B Einträge auf einmal gelesen oder geschrieben. Jeder dieser Zugriffe geschieht in $\mathcal{O}(1)$ Zeit, unabhängig von der Position des Speicherblocks. Der Hauptspeicher wiederum fasst M Einträge; es können nur Rechenoperationen auf Einträgen durchgeführt werden, die vorher in diesen Speicher geladen wurden. Die Anzahl der Rechenoperationen des Prozessors oder der Speicherzugriffe auf den Hauptspeicher spielen in diesem Modell wegen der dominierenden I/O-Kosten keine Rolle. Die asymptotische Laufzeit von Algorithmen wird als Anzahl der durchgeführten I/O-Operationen angegeben.

Es wird im Folgenden immer davon ausgegangen, dass die Kapazität des Hauptspeichers M wesentlich kleiner als jede Problemgröße N ist. Wichtige Werte sind die Anzahl der Speicherblöcke $\frac{M}{B}$, die sich zur gleichen Zeit im Hauptspeicher befinden können und die Anzahl der Blöcke $\frac{N}{B}$, die für eine Problemgröße N erforderlich sind. Die Werte sind in Tabelle 3.1 zusammengefasst.

Tabelle 3.1.: Parameter des I/O-Modells.

Wert	Semantik
N	Die Problemgröße als Anzahl von Einträgen.
M	Die Kapazität des Hauptspeichers (in Einträgen).
B	Die Anzahl der Einträge in einem Speicherblock.
$\frac{N}{B}$	Die Problemgröße (in Blöcken).
$\frac{M}{B}$	Die Kapazität des Hauptspeichers (in Blöcken).

3.2. Der B-Baum für eindimensionale Werte

Ein B-Baum ist ein Suchbaum, der im Vergleich zu den vorher erwähnten Bäumen den Gegebenheiten des Sekundärspeichers besser angepasst ist. In den Knoten des Baumes werden Schlüssel gespeichert, die mit Datenobjekten assoziiert sind. Dabei umfasst jeder Knoten genau einen Block im Sekundärspeicher. Wie ein binärer Suchbaum partitioniert der B-Baum den gespeicherten Datenbereich, sodass in jedem internen Knoten zur einem Suchschlüssel x immer nur höchstens ein Kindknoten zur Fortführung der Suche betrachtet werden muss. Um aber die gegebene Blockgröße gut auszunutzen ist der Verzweigungsgrad, oder *Fan-out* b, deutlich höher: $b \leq B$ wird so gewählt, dass zusammen mit dem Overhead der Datenstruktur (Zeiger und sonstige Metainformationen) der ungenutzte Speicherplatz in einem Block minimiert wird. So kann pro I/O eine deutlich größere Menge an Informationen übertragen werden.

Die wesentlichen Eigenschaften eines B-Baumes lauten wie folgt [nach Knu98]:

1. Ein Knoten v mit Grad $k \leq b$ besitzt k Kindzeiger und $k-1$ Schlüssel. Er hat die Form $\langle p_1, e_1, ..., p_{k-1}, e_{k-1}, p_k \rangle$. Für $1 \leq i \leq k$ zeigt p_i auf das k-te Kind des Knotens (oder *null*). Für $1 \leq i < k$ ist e_i der i-te Eintrag von v, in dem Suchschlüssel und Datenobjekt gemeinsam gespeichert sind.
2. Jeder Knoten, mit Ausnahme der Wurzel, ist mindestens halb voll: $k \geq \lceil \frac{b}{2} \rceil$.
3. Für die Wurzel gilt $k \geq 2$, es sei denn, sie ist ein Blatt.
4. Alle Blätter haben die gleiche Tiefe. Ihre Kindzeiger sind alle *null*.
5. Die $k \leq b$ Kindzeiger eines internen Knoten zeigen nicht auf *null*.
6. Die Schlüssel innerhalb eines Knotens sind geordnet: es gilt $e_1 < e_2 < ... < e_{k-1}$.
7. Ein interner Knoten partitioniert den Inhalt seines Teilbaums. Es sei x ein beliebiger Schlüssel im Teilbaum von p_i. Für $i > 1$ folgt $x > e_{i-1}$, für $i < k$ folgt $x < e_{i+1}$.

Zusammengenommen ergeben diese Eigenschaften einen balancierten Suchbaum:

Satz 1. Die Höhe eines B-Baums mit N Schlüsseln liegt in $\mathcal{O}(\log_b N)$.

Beweis (nach [Cor+09]). Im Folgenden sei $k := \frac{b}{2} - 1$.

Ein nicht-leerer Baum der Höhe 1 besteht nur aus einem Blatt und besitzt $N \geq 1$ Schlüssel. Ein Baum der Höhe $h \geq 2$ besitzt als Wurzel einen internen Knoten mit mindestens einem Schlüssel und zwei Kindern. Die Knoten mit Tiefe 1 haben mindestens k Schlüssel, auf dieser Ebene gibt es also $2k$ Schlüssel. Jeder dieser Knoten besitzt weitere $k+1$ Kinder, sodass es wenigstens $2(k+1)$ Knoten und $2(k+1)k$ Schlüssel in Tiefe 2 gibt, usw., bis die Blattebene erreicht wurde. In Tiefe $d \geq 1$ gibt es also mindestens $2k(k+1)^{d-1}$ Schlüssel. Insgesamt gilt

$$\begin{aligned} N \geq 1 + \sum_{d=1}^{h-1} 2k(k+1)^{d-1} &= 1 + 2k \sum_{d=1}^{h-1} (k+1)^{d-1} \\ &= 1 + 2k \cdot \frac{(k+1)^{h-1} - 1}{k} \\ &= 2(k+1)^{h-1} - 1 \\ \Rightarrow \frac{N+1}{2} &\geq (k+1)^{h-1} \\ \Rightarrow h &\leq \log_{k+1}\left(\frac{N+1}{2}\right) + 1. \end{aligned} \tag{3.1}$$

□

Um nach einem Objekt zu suchen, wird mit der Wurzel begonnen. Mit binärer Suche wird der Knoten nach dem Objekt durchsucht. Ist das Objekt im aktuellen Knoten enthalten, ist die Suche beendet. Ist der aktuelle Knoten ein Blatt, so wurde das Objekt nicht gefunden. Ansonsten wird die Suche im passenden Teilbaum (Punkte 6. und 7.) fortgeführt. Wegen der Höhe des Baumes benötigt eine Suchoperation also im schlechtesten Fall $\mathcal{O}(\log_{b/2} N)$ Leseoperationen.

Ein B-Baum stellt keine besonderen Anforderungen an die in ihm gespeicherten Objekte, mit Ausnahme der Existenz einer wohldefinierten „<"-Relation auf den Suchschlüsseln. In der Praxis werden häufig Suchschlüssel zusammen mit zeigerähnlichen Objekten gespeichert, die auf den tatsächlichen Ort des Objekts im Sekundärspeicher verweisen (z.B. ein Blockindex zusammen mit einem Offset).[4] Ein Objekt kann allerdings auch direkt im Baum gespeichert werden: die Wahl der Strategie ist anwendungsspezifisch. Große Objekte verringern die Kapazität eines Baumes und wirken sich deshalb negativ auf dessen Höhe

[4] Dieser Ansatz wird von Datenbankmanagementsystemen wie MySQL verfolgt, um die Zeilen einer Tabelle zu indizieren: Ein B-Baum-Index speichert Suchschlüssel zusammen mit einem *row index*, welcher wiederum auf die tatsächliche Zeile in der Tabelle zeigt.

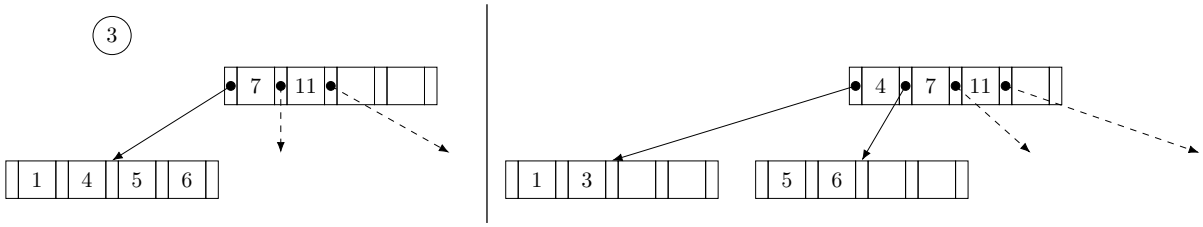


Abbildung 3.1.: Ein neuer Schlüssel wird in ein volles Blatt eines B-Baumes eingefügt.

aus, während ein Zeiger eine Indirektion darstellt und damit die Zeit bis zum Auffinden des Objekts potentiell verlängert.

Einfügen und Löschen

Wenn ein Schlüssel x in einen schon existierenden Baum eingefügt wird, müssen die Invarianten eines B-Baumes erhalten bleiben. Da ein leeres Blatt ein gültiger B-Baum ist, kann das folgende Verfahren benutzt werden, um einen B-Baum schrittweise aufzubauen. Zunächst wird das Blatt bestimmt, welches den Schlüssel x speichern kann (wie bei der Suche ist dieses Blatt wegen der Eigenschaften 6. und 7. eindeutig). Gibt es in dem Blatt noch Platz für ein weiteres Element, so wird x an der durch die Ordnung „$<$“ vorgeschriebenen Stelle einsortiert und wir sind fertig. Speichert das Blatt allerdings schon $b-1$ Schlüssel, muss ein *node split* durchgeführt werden: Die Liste der Schlüssel $e_1, ..., e_b$ wird in zwei Hälften geteilt: die Elemente $e_1, ..., e_{\lceil \frac{b}{2} \rceil - 1}$ werden in ein neues Blatt verschoben, $e_{\lceil \frac{b}{2} \rceil + 1}$, ..., e_b verbleiben in ihrem Blatt und $e_{\lceil \frac{b}{2} \rceil}$ wird zusammen mit einem Zeiger auf das neue Blatt in den Elternknoten eingefügt. Dies kann dazu führen, dass auch der Elternknoten überläuft; das Verfahren muss also bis zur Wurzel fortgesetzt werden. Muss die Wurzel geteilt werden, so wird ein neuer interner Knoten mit genau zwei Kindern (der alten Wurzel und dem beim Split neu erzeugten Knoten) erstellt und der Baum wächst um eine Ebene.

Abbildung 3.1 illustiert den Algorithmus für einen B-Baum mit $b = 5$. Der Schlüssel 3 soll in ein bereits volles Blatt eingefügt werden. Man erhält für das überlaufende Blatt die Liste $1, 3, 4, 5, 6$. Der mittlere Eintrag wird in den Elternknoten eingefügt, die linke und rechte Hälfte bilden jeweils einen Knoten.

Der Algorithmus erhält die Eigenschaften des B-Baums: Da überlaufende Knoten immer in zwei etwa gleich große neue Knoten aufgeteilt werden, sind sie mindestens halb voll. Da das Ergebnis eines Splits immer zwei Teilbäume mit gleicher Höhe sind, bleibt die

Tiefe aller Blätter gleich. Im schlimmsten Fall muss der Algorithmus beim Einfügen von x in einen Baum mit N Schlüsseln zunächst ein passendes Blatt in $\mathcal{O}(\log_{b/2} N)$ I/Os finden. Anschließend müssen gegebenenfalls genauso viele Splits durchgeführt werden, welche als Ergebnis jeweils einen neuen Knoten erstellen und einen alten modifizieren. Knuth [Knu98, Kapitel 6.2.4] zeigt allerdings, dass beim Erstellen eines Baumes aus N Schlüsseln pro eingefügtem Schlüssel im Mittel weniger als $\frac{1}{\lceil b/2 \rceil - 1}$ Splits durchgeführt werden müssen.

Bei der Löschung eines Schlüssels muss umgekehrt vorgegangen werden: enthält ein Knoten weniger als $\lceil \frac{b}{2} \rceil$ Einträge, so wird er mit seinen Nachbarn verschmolzen. Entweder werden von benachbarten Knoten Einträge überführt oder der Inhalt des zu leeren Knotens wird auf die Nachbarn verteilt; der Knoten wir anschließend gelöscht. Ist als Konsequenz einer Löschung die Wurzel ein interner Knoten mit nur einem Kind, so kann sie gelöscht und das Kind als neue Wurzel verwendet werden. Auf diese Art und Weise reduziert sich die Höhe um eins.

Der B$^+$-Baum

Eine wichtige Variante des B-Baums ist der B$^+$-Baum (beschrieben in [EN10]). Während der B-Baum in jedem seiner Knoten sowohl Suchschlüssel wie auch Datenobjekte speichert, enthalten interne Knoten eines B$^+$-Baumes nur Suchschlüssel und Kindzeiger; die Daten befinden sich ausschließlich in den Blattknoten. In den internen Knoten steht durch das Fehlen der Datenobjekte mehr Speicherplatz für Schlüssel und Zeiger zur Verfügung. Der Fan-out in internen Knoten lässt sich daher abhängig von der Größe der Objekte oft drastisch steigern, was die Höhe des Baumes reduziert und die *worst-case*-Laufzeit verbessert. Als Konsequenz haben interne Knoten und Blätter eine unterschiedliche Struktur, was die Implementierung komplizierter gestaltet. Der eine Nachteil eines B$^+$-Baums ist, das nun *alle* Suchen das Blattlevel erreichen, vorher konnte eine Suche gegebenenfalls schon in einem internen Knoten erfolgreich beendet werden.

B$^+$-Bäume können genutzt werden, um eine sehr effiziente *in-order*-Traversierung der Objekte in einem Baum zu ermöglichen: indem alle benachbarten Blätter mit Zeigern verlinkt werden, erhält man eine Liste aller Objekte in der durch die verwendete „<"-Relation induzierten Ordnung. So können Intervallabfragen realisiert werden: Das Finden aller Schlüssel im Intervall $[a, b]$ beinhaltet nur das Suchen nach dem Schlüssel a und ein anschließendes Iterieren durch die geordnete Liste, bis alle Schlüssel im Intervall besucht

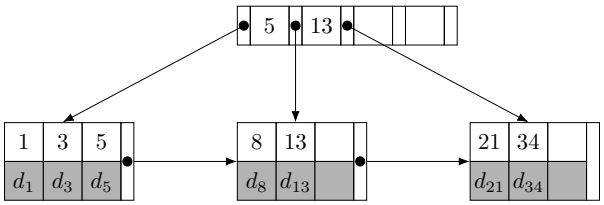


Abbildung 3.2.: Ein B$^+$-Baum der Höhe 2. Interne Knoten enthalten nur duplizierte Suchschlüssel. Die mit den Schlüsseln assoziierten Daten werden in den Blättern abgelegt. Knoten werden immer durch ihren maximalen Schlüssel im Elternknoten repräsentiert.

worden sind. Es werden also nur $\mathcal{O}(\log_{b/2} N + \frac{K}{B})$ I/Os benötigt, wobei K die Größe der Ergebnismenge ist. Ein B$^+$-Baum ist in Abbildung 3.2 dargestellt. Die Suche nach dem Intervall $[3, 9]$ findet zuerst im linken Blatt den Schlüssel 3 und iteriert von dort aus in aufsteigender Reihenfolge weiter, bis beim Erreichen des Schlüssels 13 die Ergebnismenge $\{(3, d_3), (5, d_5), (8, d_8)\}$ bestimmt worden ist.

Caching

Sowohl für den B-Baum wie auch für die im Folgenden besprochenen Datenstrukturen für mehrdimensionale Objekte ist es sinnvoll, einen Cache (etwa nach dem *least-recently-used*-Prinzip) einzusetzen. Da die Höhe der hier verwendeten Bäume selbst für sehr große Datenmengen aufgrund des hohen Fan-Outs niedrig bleibt, kann schon ein Cache mit einer geringen Anzahl von Blöcken viele Operationen auf häufig benutzen Knoten wie der Wurzel signifikant reduzieren. Beim Durchsuchen eines B-Baums mit N Schlüsseln reicht zum Beispiel bereits ein Zwischenspeicher mit der Kapazität $C = \lceil \log_{b/2} N \rceil$ aus, um zu garantieren, dass die Wurzel immer im Cache verfügbar ist.

3.3. Repräsentation mehrdimensionaler Objekte

Das effiziente Arbeiten mit mehrdimensionalen Daten gestaltet sich als deutlich schwieriger als der eindimensionale Fall. Es gibt im Allgemeinen keine offensichtliche Ordnung, um n-dimensionale Objekte so linear anzuordnen, dass sie sich direkt in einem B-Baum oder einer ähnlichen eindimensionalen Datenstruktur speichern lassen. Neben dem bloßen Auffinden eines Objekts müssen auch komplexere räumliche Abfragen unterstützt werden. Beispielsweise kann in einer *Bereichsabfrage* verlangt werden, alle Objekte zu finden,

die einen gewissen Bereich überlappen oder in diesem vollständig enthalten sind. Da es sich nicht zwingend nur um n-dimensionale Punkte, sondern auch um komplexe Figuren wie etwa Polygone handeln kann, muss auch untersucht werden, wie diese in einer Datenstruktur effektiv repräsentiert werden können: Tests auf Überlappungen o.Ä. sollen in akzeptabler Zeit durchgeführt werden.

Definition 2. Es sei $O \subset \mathbb{R}^n$ ein durch eine Punktmenge beschriebenes, n-dimensionales Objekt mit endlicher Ausdehnung in allen Dimensionen. Die *minimum bounding box* (im Folgenden immer abgekürzt als *MBB*) ist das kleinste n-dimensionale, achsenparallele Hyperrechteck, das alle $p \in O$ enthält.
Sie hat die Gestalt $mbb(O) := (I_1, ..., I_n)$, wobei I_i für $1 \leq i \leq n$ ein Intervall in $\mathbb{R}$ ist, welches den minimalen und den maximalen Wert für Dimension i als Grenzen besitzt.

Die Bounding Box von Objekten kann nun dazu dienen, eine Vielzahl von Vergleichen zu eliminieren: Gilt es, die Objekte O_1 und O_2 auf Überlappung zu überprüfen, werden stattdessen zunächst deren MBBs betrachtet. Diese werden in der Regel anstatt der tatsächlichen geometrischen Struktur der Objekte in der Indexstruktur gespeichert und müssen daher nicht immer neu berechnet werden. Falls die MBBs der beiden Objekte sich nicht schneiden, so kann eine Überlappung in jedem Fall ausgeschlossen werden. Nur wenn dies nicht der Fall ist, muss ein (ggf. sehr teurer) Vergleich von O_1 und O_2 durchgeführt werden.

Die Leistung dieser Heuristik hängt stark von den verwendeten Daten ab. Je nach geometrischer Form der Objekte wird durch die grobe Abschätzung (das Minimum und Maximum aller Komponenten) gegebenenfalls viel Volumen verschwendet. Dies kann zu vielen *false positives* bei der Prüfung auf Überschneidungen führen. Als Beispiel betrachte man ein zweidimensionales Liniensegment zwischen den Punkten $(0, 0)$ und $(1, 1)$, dessen MBB ebenfalls durch genau diese Punkte bestimmt wird. Obwohl das Liniensegment nur über die Hauptdiagonale des Rechtecks verläuft, müssen alle Objekte, die die im Vergleich sehr große MBB überlappen, genauer betrachtet werden.

Es existieren verschiedene Strategien, um räumliche Objekte besser zu repräsentieren. Eine *oriented minimum bounding box* (OBB) ist im Vergleich zur MBB nicht unbedingt achsenparallel; es handelt sich um ein um einen beliebigen Winkel rotiertes Hyperrechteck [ORo85]. Gerade für das Beispiel der Liniensegmente ist diese Lösung optimal. Eine weitere Alternative ist die *bounding sphere*, also die kleinste n-dimensionale Kugel, die ein Objekt vollständig enthält. Aufgrund der Einfachheit der Berechnung des Rechtecks selbst

und der Effizienz bei der Bestimmung des Schnittes von Rechtecken sind MBBs trotzdem eine beliebte Art und Weise, um höherdimensionale Objekte zu repräsentieren.

3.4. Der R-Baum

Der R-Baum ist eine Indexstruktur für mehrdimensionale räumliche Objekte und wurde erstmals von Guttman [Gut84] vorgeschlagen. Er ist in seiner Struktur dem B-Baum sehr ähnlich – mit dem wichtigen Unterschied, dass ein R-Baum die in ihm gespeicherten Datensätze *nicht* in vollständig disjunkte Mengen partitionieren kann.

Die wesentlichen Eigenschaften des Baums lauten wie folgt [nach GS05, Kapitel 7]:

1. Jeder Knoten v, mit Ausnahme der Wurzel, besitzt zwischen a und b Kindern, mit $1 \leq a \leq \frac{b}{2}$. $mbb(v)$ ist die MBB aller Einträge von v.
2. Die Wurzel ist entweder ein Blatt oder besitzt mindestens zwei Einträge.
3. Die Einträge eines Blatts haben die Form $(mbb(d), d)$. d repräsentiert ein n-dimensionales Objekt.
4. Die Einträge eines internen Knoten haben die Form $(mbb(c), c)$, wobei c ein Zeiger auf ein Kind des Knotens ist. Kindknoten werden in ihrem Elternknoten durch ihre MBB repräsentiert.
5. Alle Blätter haben die gleiche Tiefe.

Mit diesen Bedingungen erhält man einen balancierten Baum, dessen Knoten – mit Ausnahme der Wurzel – mindestens zu $\frac{a}{b}$ gefüllt sind und dessen Höhe in $\mathcal{O}(\log_b N)$ liegt [GS05].

Abbildung 5.3 (auf Seite 39) zeigt einen R-Baum mit $b = 4$, in dem dreidimensionale Liniensegmente gespeichert werden. In diesem Fall liegt die MBB eines Segments nicht explizit vor, sondern ergibt sich indirekt aus den beiden Endpunkten. $R_1, ..., R_4$ sind die MBBs der vier Blätter des Baums. Die Wurzel (nicht abgebildet) hat den Grad 4 und speichert diese MBBs zusammen mit einem Zeiger auf das jeweilige Blatt.

Um in einem R-Baum nach Objekten zu suchen, die einen Punkt enthalten oder ein n-dimensionales Hyperrechteck überlappen, wird der Baum von der Wurzel aus durchlaufen. In jedem internen Knoten werden die MBBs aller Kindknoten darauf überprüft, ob sie sich

mit dem Suchbereich überschneiden. Ist dies der Fall, so werden ihre Teilbäume rekursiv durchsucht. Die Rekursion endet in der Blattebene: die Datenobjekte werden geladen und auf tatsächliche Überlappung getestet. Da sich allerdings in jedem Knoten die MBBs der Kinder überlappen können, müssen im Allgemeinen mehrere Teilbäume durchsucht werden. Im schlimmsten Fall muss selbst für einen kleinen Suchbereich der gesamte Baum durchsucht werden, weshalb für die Suche keine bessere obere Schranke als $\mathcal{O}(N)$ I/O-Operationen angegeben werden kann. Umso wichtiger ist es, bei der Konstruktion eines Baumes Verfahren anzuwenden, um diesen *Overlap* zu reduzieren.

Einfügen

Um ein neues Objekt in einen R-Baum einzufügen, wird beginnend mit der Wurzel in jedem Knoten das Kind ausgewählt, für welches das Einfügen am günstigsten ist. Als Maß für die Kosten verwendet man die Vergrößerung der MBB des Kindknotens, die stattfinden muss, falls das neue Objekt in dieses Kind eingefügt wird. Ist ein Objekt schon vollständig in der MBB eines Kindes enthalten, so sind die Kosten 0. Indem die MBBs durch diesen Ansatz möglichst klein gehalten werden, soll die Wahrscheinlichkeit verringert werden, dass mehrere Kinder einen Suchbereich überlappen.

Nachdem die MBB des ausgewählten Kindknotens so aktualisiert wurde, dass sie das neue Objekt enthält, wird der Algorithmus rekursiv im Teilbaum des Kindes fortgesetzt, bis ein Blatt erreicht wird, in den das neue Objekt schließlich eingefügt wird. Wie beim B-Baum muss eine Knotenteilung durchgeführt werden, falls die Kapazität b des Blattes überschritten wird. Das so neu hinzukommende Blatt wird im Elternknoten eingefügt, welcher wiederum selbst gegebenfalls geteilt werden muss, bis schließlich entweder das Einfügen gelingt oder die Wurzel geteilt wird und der Baum in der Höhe wächst.

Nicht jede der möglichen Aufteilungen eines Knotens ist gut: die beiden neuen Knoten sollten eine möglichst geringe Fläche abdecken und sich so wenig wie möglich überlappen. Guttman [Gut84] beschreibt für das Teilen von Knoten verschiedene Algorithmen. An dieser Stelle wird nur der (häufig benutzte) QuadradicSplit näher beschrieben.[5] Aus der Menge der $b+1$ Einträge $\{e_1, ..., e_{b+1}\}$ werden zwei *seeds* gewählt – dies sind die

[5] Guttmans LinearSplit benötigt weniger Rechenzeit, erzielt aber schlechtere Ergebnisse. Die Bestimmung aller möglichen Partitionen würde hingegen den besten Split zur Folge haben, die exponentielle Laufzeit macht diesen Ansatz allerdings in der Praxis unbrauchbar.

ersten Einträge der beiden Gruppen. Es werden genau die Einträge e_i, e_j berechnet, deren gemeinsame MMB am meisten Volumen *vol* verschwenden würde, deren gemeinsame Unterbringung in einem Blatt also am wenigsten Sinn ergeben würde:

$$(i, j) := \underset{1 \leq i < j \leq b+1}{\operatorname{argmin}} \; vol(mbb(e_i, e_j)) - vol(mbb(e_i)) - vol(mbb(e_j)). \tag{3.2}$$

Im Anschluss wird solange iteriert, bis alle anderen Einträge einer der Gruppen zugeordnet wurden. In jeder Iteration werden für jeden verbleibenden Eintrag die Kosten[6] für beide Gruppen berechnet. Der Eintrag, dessen beiden Kostenwerte die größte absolute Differenz aufweisen, wird der günstigeren Gruppe zugeordnet. Zusammen mit der Auswahl der beiden *seeds* soll diese Heuristik möglichst kleine und wenig überlappende Gruppen erzeugen. Die Iteration wird beendet, wenn alle Einträge zugeordnet wurden oder falls eine der Gruppen zu groß wird. Jede Gruppe muss aufgrund der Eigenschaften des R-Baums mindestens a Einträge umfassen; in diesem Fall werden die verbleibenden Einträge der kleineren Gruppe zugeteilt. Die Laufzeit liegt wegen der Auswahl aus Paaren in Gleichung 3.2 und der gerade beschriebenen Iteration in $\mathcal{O}(b^2)$. In der Praxis ist die quadratische Laufzeit jedoch für R-Bäume im Sekundärspeicher wenig problematisch, da die I/O-Kosten dominieren.

3.5. Datenstrukturen für Trajektorien

R-Bäume können genutzt werden, um raumzeitliche Trajektorien zu indizieren. Indem man die räumliche Position innerhalb einer Trajektorie zu diskreten Zeitpunkten festhält, erhält man eine Folge von Punkten, die sich wiederum zu einer *Polyline* (eine Reihe direkt aufeinanderfolgender Liniensegmente) verbinden lassen. Diese Linie ist eine Approximation der ursprünglichen Trajektorie, deren Präzision von den gewählten Zeitpunkten abhängt. In diesem Kontext wird die Zeit als gewöhnliche Dimension interpretiert: Aus der Trajektorie eines sich im zweidimensionalen Raum bewegenden Objekts erhält man eine dreidimensionale Polyline, deren Punkte die Gestalt (x, y, t) besitzen.

Indem man die einzelnen Liniensegmente in den R-Baum einfügt und zusätzlich mit der eindeutigen ID *tid* der ursprünglichen Trajektorie annotiert, lässt sich nun eine Menge von

[6] Wie beim Einfügen wird hier als Maß für die Kosten die notwendige Vergrößerung der MBB der Gruppe verwendet.

Trajektorien nach räumlichen Kriterien durchsuchen. Eine Anfrage nach einem bestimmten raumzeitlichen Suchbereich liefert dann eine Menge von $(line, tid)$ Tupeln, anhand derer sich feststellen lässt, welche Trajektorien durch den Suchbereich verlaufen. Da ein R-Baum aber die in ihm indizierten Objekte unabhängig voneinander nur anhand der räumlichen Nähe speichert, lassen sich die Trajektorien anhand der Liniensegmente nicht effizient rekonstruieren. Ein so konstruierter R-Baum eignet sich also nicht als primäre Indexstruktur für Trajektorien, welche sowohl raumzeitliche Anfragen wie auch schnellen Zugriff auf vorherige oder nachfolgende Liniensegmente ermöglichen muss.

3.5.1. Der STR-Baum

Der *Spatio-Temporal R-tree* (STR-Baum) ist eine Erweiterung des R-Baums mit dem Ziel, die einzelnen Segmente von Trajektorien nahe beieinander zu speichern. Wird ein neues Liniensegment eingefügt, so wird zunächst dessen Vorgänger in derselben Trajektorie gesucht. Falls möglich, wird das neue Liniensegment dann im selben Blatt wie sein Vorgänger abgelegt. Ansonsten wird ein *node split* durchgeführt, welcher wiederum Vorkehrungen trifft, um zusammengehörige Segmente im gleichen Knoten anzusiedeln.

Dieses Verhalten lässt sich durch den *preservation parameter* $p \geq 1$ steuern: Sind sowohl das Blatt des Vorgängersegments wie auch seine $p-1$ Elternknoten voll, so wird *kein* Split, sondern der „gewöhnliche" R-Baum Einfügealgorithmus durchgeführt. In diesem Fall geht der Zusammenhang der Segmente verloren. Das Suchen und Löschen funktionieren genauso wie beim R-Baum. Zusammengefasst ist der STR-Baum weiterhin als räumlicher Index nutzbar, erzielt aber einen besseren Zusammenhang der Trajektoriensegmente als der R-Baum [GS05, Kapitel 7].

3.5.2. Der TB-Baum

Sowohl beim R- wie auch beim STR-Baum können in einem Blatt Liniensegmente verschiedener Trajektorien auftreten. Dadurch wird es nötig, jedes Segment mit seiner *tid* zu annotieren. Der *Trajectory-Bundle tree* (TB-Baum) wählt einen anderen Ansatz: Jedes Blatt speichert Segmente zu genau einer Trajektorie, deren ID daher nur einmal in dem Blatt festgehalten werden muss. Die raumzeitlichen Aspekte von Blättern und internen Knoten werden wie beim R-Baum durch MBBs repräsentiert. Der TB-Baum ist eine *append-only* Datenstruktur: die Liniensegmente einer Trajektorie können nur in der

Reihenfolge eingefügt werden, in der sie in der Trajektorie selbst auftreten (also zeitlich aufsteigend). Die Blätter des Baums, die Segmente einer bestimmten Trajektorie speichern, sind in beide Richtungen verlinkt: wird bei einer Suche ein einzelnes Blatt besucht, kann die gesamte Trajektorie durch Folgen der Zeiger in beide Richtungen rekonstruiert werden.

Um ein Liniensegment in den TB-Baum einzufügen, wird zunächst dessen unmittelbarer Vorgänger gesucht. Da es sich um Segmente einer Polyline handelt, muss nur das Liniensegment gefunden werden, welches den aktuellen Startpunkt als Endpunkt besitzt. Es wird versucht, das Segment in das Blatt b des Vorgängers einzufügen; ist dies aufgrund der Kapazität des Blattes nicht möglich, so wird *kein* Split durchgeführt, sondern ein neues Blatt b' für das neue Segment erstellt. Im Anschluss wird ein nicht-voller Elternknoten von b bestimmt, um b' in seinem Teilbaum zu speichern. Da sich alle Blätter eines TB-Baums in der gleichen Tiefe befinden müssen, wird b' das Kind des internen Knoten der Höhe 1, der sich in diesem Teilbaum am weitesten rechts befindet. Falls dieser Knoten voll ist, muss an dieser Stelle ein Split durchgeführt werden, welcher gegebenenfalls wie beim R-Baum bis zur Wurzel fortgesetzt wird.

Da die Blätter einer Trajektorie im Baum verlinkt sind, eignet sich dieser gut, um die gesamte Bewegungshistorie eines gefundenen Objekts zu rekonstruieren. Als *append-only*-Datenstruktur kann ein TB-Baum genutzt werden, um Bewegungsmuster in Echtzeit aufzuzeichnen. Der Baum unterstützt die Suche nach räumlichen Bereichen, die aufgrund der Besonderheiten beim Einfügen in der Effizienz allerdings schlechter ausfällt als beim R-Baum.

3.6. Indizierung von Texten

3.6.1. Der invertierte Index

Es sei W eine Menge möglicher Worte und $D = (d_1, ..., d_n) \in W^*$ ein Dokument, welches aus diesen Worten besteht. Um zu einem bestimmten Wort alle dessen Positionen zu finden, ist ohne eine geeignete Indexstruktur eine lineare Suche über die gesamte Datei notwendig. Ein *invertierter Index* [ZMR98] (oder eine *invertierte Datei* [Knu98, Kapitel 6.5]) dient der effizienten Suche von Worten in D und liefert zu einem Wort $w \in W$ die

$$D = (w_1, w_2, w_1, w_3, w_2)$$
$$W = \{w_1, w_2, w_3, w_4\}$$

w	$inv(w)$
w_1	$\{1, 3\}$
w_2	$\{2, 5\}$
w_3	$\{4\}$
w_4	$\emptyset$

Abbildung 3.3.: Ein Dokument und sein invertierter Index.

Positionen, an denen es sich in D befindet:

$$inv(w) := \{i \mid 1 \leq i \leq n \wedge w = d_i\}. \tag{3.3}$$

Abbildung 3.3 enthält ein Beispiel für den invertierten Index eines Dokuments.

Eine invertierte Datei wird häufig durch 2 Datenstrukturen implementiert. Die *Suchstruktur* unterstützt die Suche nach einem Wort w und bildet es auf den Zeiger zu einer *Posting List* (oder einer *invertierten Liste*) ab. Eine dafür gut geeignete Datenstruktur ist der B^+-Baum, der aufgrund seines hohen Fan-outs eine hohe Anzahl von Einträgen mit geringem Overhead sowohl bei der Speicherung wie auch bei der Suche unterstützen kann. Jede Posting List zu einem Wort w enthält den Inhalt der Menge $inv(w)$ und kann als lineare Liste implementiert werden. Wird in d eine Suche nach w ausgeführt, so muss nur mithilfe weniger I/O-Operationen in der Suchstruktur die Posting List zu w lokalisiert werden und diese danach iteriert werden.

Das Konzept des invertierten Index kann leicht auf die Textsuche in einer Menge von Dokumenten $\mathcal{D} = \{D_1, ..., D_n\}$ erweitert werden, wobei die D_i für $1 \leq i \leq n$ wieder wie oben aus Worten bestehen. In Suchmaschinen werden invertierte Listen eingesetzt, um zu einem Wort $w \in W$ alle Dokumente in $\mathcal{D}$ zu finden, in denen dieses vorkommt. Dazu werden in der Posting List zu w Zeiger auf genau die Dokumente abgelegt, die w mindestens einmal enthalten – die exakten Positionen von w innerhalb der Dokumente ist nicht unbedingt relevant. Zusätzlich zu diesen Zeigern können weitere Informationen gespeichert werden, wie z.B. die absolute oder relative Häufigkeit des Worts im Dokument. Mithilfe der Häufigkeiten kann eine rudimentäre Anordnung der Ergebnisse geschehen, indem man von einer großen Häufigkeit auf eine hohe Relevanz eines Dokuments schließt.

In dieser Arbeit werden invertierte Dateien für die Indizierung von Knoten und deren Kindern verwendet. Im IRWI-Baum (Kapitel 5) kann zu einem Knoten und einem *La-*

bel die Menge der Kinder des Knotens bestimmt werden, deren Teilbäume Einträge mit diesem Label enthalten.

3.6.2. Signaturdateien

Signaturdateien sind als Alternative zu invertierten Dateien ebenfalls zur Volltextsuche in Dokumenten (oder in Mengen von Dokumenten) verwendbar. Zu einem Dokument $D = (d_1, ..., d_n) \in W^*$ erhält man dessen *Signaturen*, indem man es zunächst in Textblöcke der Größe k unterteilt: jeweils k eindeutige, aufeinanderfolgende Worte bilden einen Block. Die Signatur eines Blocks wird berechnet, indem zu jedem der Worte dessen Hashwert gebildet wird, welcher aus m-Bits besteht; die k Hashwerte werden dann mithilfe des binären ODERs zur Signatur zusammengefasst. In der Literatur heißt dieser Prozess *superimposed coding* [LKP95]. Soll eine Suche nach einem Wort $w \in W$ durchgeführt werden, so wird der Hashwert von w mit jeder Blocksignatur verglichen: ist das binäre UND von Signatur und Hash nicht 0, so ist w *wahrscheinlich* im Block enthalten. Wegen möglicher Kollisionen kann es hier zu *false positives* (auch: *false drops*) kommen; um das Vorhandensein eines Worts mit Sicherheit festzustellen müssen diese Blöcke geladen werden.

Mithilfe der Signaturen können Textblöcke häufig von der Suche ausgeschlossen werden, ohne ihren Inhalt zu laden. Die Qualität von Signaturdateien und deren Speicherplatzverbrauch hängt von den Parametern m und k ab: mehr Bits (oder weniger Worte) pro Signatur verringern die Wahrscheinlichkeit von Kollisionen, benötigen aber auch mehr Speicherplatz. Da die Anzahl der Textblöcke aber nur von der Größe des Dokuments abhängt, ist der Overhead in jedem Fall in $\mathcal{O}(n)$. Laut Lee, Kim und Patel [LKP95] fällt dieser allerdings deutlich geringer aus als bei invertierten Dateien. Um selbst viele Textblöcke effizient zu durchsuchen, schlagen sie die Verwendung einer *multi-level signature file* vor: es wird eine Baumstruktur angelegt, in der ein Knoten die Signatur aller Textblöcke enthält, die in seinem Teilbaum gespeichert sind (es wird auch hier *superimposed coding* verwendet). Knoten höherer Ebenen besitzen dabei längere Signaturen, um die aus der größeren Datenmenge folgende, ansonsten höhere Wahrscheinlichkeit von *false positives* auszugleichen.

Zobel, Moffat und Ramamohanarao [ZMR98] haben festgestellt, dass Signaturdateien den invertierten Dateien in ihrer Leistung insgesamt unterlegen sind. Trotzdem können sie gegebenenfalls genutzt werden, um den invertierten Index des IRWI-Baums (Kapitel 5) zu ersetzten. Dieser belegt unter gewissen Umständen besonders viel Speicherplatz und

könnte daher von dem verringerten Overhead der Signaturen profitieren. Die *multi-level signature file* sollte leicht in die ohnehin schon hierarchische Struktur des Baums integrierbar sein. Eine Untersuchung dieses Punkts findet in dieser Arbeit nicht statt, stellt aber eine mögliche Richtung für zukünftige Arbeiten dar.

4. Gebündelte Operationen

Die bisher betrachteten Einfügeoperationen unterstützen nur jeweils das effiziente Einfügen eines einzelnen Elements. Vor und nach jeder Operation erhält man einen gültigen Zustand der betreffenden Datenstruktur. Ein neuer B$^+$-Baum kann beispielsweise aus N Schlüssel/Wert-Paaren konstruiert werden, indem alle Paare nacheinander mithilfe des Einfügealgorithmus im Baum abgelegt werden. Diese Herangehensweise wird im Folgenden *one-by-one insertion* (OBO) genannt. Obwohl das Einfügen eines einzelnen Elements effizient ist, ist die Leistung dieses Verfahrens für große Datenmengen in der Praxis inakzeptabel: das Einfügen von N Einträgen kostet $\mathcal{O}(N \log_b N)$ I/Os; eine Anzahl, die durch verschiedene *bulk-loading*-Algorithmen zumindest um einen konstanten Faktor verringert werden kann. Dabei wird daraus Vorteil gezogen, dass alle Elemente schon im Voraus bekannt sind, es können also Wege gefunden werden, um mehr als nur ein einziges Element auf einmal in den Baum einzufügen.

Mehrere der nachfolgenden Algorithmen basieren auf einem effizienten Sortierverfahren für selbst sehr große Datenmengen. ExternalMergeSort ist eine des Variante *Merge-Sort* Algorithmus, die die gesamte Datenmenge in *Chunks* der Länge C unterteilt und diese einzeln intern sortiert. Die Chunks werden in den externen Speicher ausgelagert und schließlich, nachdem sie alle sortiert worden sind, zu einer global sortierten Sequenz zusammengeführt. Die asymptotische Laufzeit (die Anzahl der I/Os) ist von den Parametern B, N und C abhängig und liegt in $\Theta(\frac{N}{B} \log_{\frac{C}{B}} \frac{N}{B})$ [AV88].

4.1. Der eindimensionale Fall

Aufgrund seiner Struktur kann ein B$^+$-Baum besonders einfach mit N Schlüssel/Wert-Paaren geladen werden. Es kann die Tatsache ausgenutzt werden, dass die Blätter eines B$^+$-Baums von „links" nach „rechts" genau das gesamte Datenset in sortierter Reihenfolge enthalten. Die Idee des Algorithmus kann in 4 Schritten zusammengefasst werden:

1. Sortiere die N Elemente bezüglich ihres Schlüssels mit ExternalMergeSort.
2. Durchlaufe die sortierte Sequenz und gruppiere jeweils b_1 aufeinanderfolgende Einträge zu einem Blatt. b_1 ist der Fan-out der Blätter des Baums.
3. Falls im letzten Schritt nur ein einziger Knoten konstruiert wurde, so mache diesen zur Wurzel des Baums und terminiere.
4. Durchlaufe die Sequenz der im letzten Schritt konstruierten Knoten und gruppiere jeweils b_2 aufeinanderfolgende zu Kindern eines neuen internen Knoten. Ein Knoten wird in seinem neuen Elternknoten durch den maximalen Schlüssel seines Teilbaums repräsentiert. b_2 ist der Fan-out der internen Knoten des Baums. Fahre anschließend mit Schritt 3 fort.

Der Algorithmus ist ein *bottom-up*-Verfahren: die Ebenen des Baumes werden von den Blättern aus aufsteigend konstruiert. Die Knoten einer Ebene werden solange zu Knoten der nächsten Ebene kombiniert, bis nur noch ein Knoten übrig bleibt. Die in der Praxis festzustellende Leistungssteigerung im Vergleich zum OBO-Algorithmus wird dadurch erzielt, dass der Inhalt jedes Knotens bei seiner Erzeugung schon vollständig feststeht: es muss niemals ein Split durchgeführt werden; pro fertigem Knoten wird genau ein Speicherblock geschrieben. Nach der anfänglichen Sortierung in Schritt 1 werden also nur $\mathcal{O}(\frac{N}{B})$ zusätzliche I/Os durchgeführt, sodass die asymptotische Laufzeit wegen ExternalMergeSort weiterhin in $\Theta(\frac{N}{B} \log_{\frac{C}{B}} \frac{N}{B})$ I/Os liegt.

Das Ergebnis ist ein in (fast) allen Belangen gültiger B^+-Baum. Der Baum ist balanciert, die Schlüssel werden aufgrund der Sortierung korrekt partitioniert und alle Knoten enthalten höchstes b_1 bzw. b_2 Einträge. Allerdings wird von diesem Algorithmus die Bedingung der *minimalen* Anzahl der Einträge verletzt: der letzte Knoten einer jeden Ebene ist gegebenenfalls weniger als halb voll. Da dieses Problem aber nur in höchstens einem Knoten pro Ebene, also im schlimmsten Fall $\mathcal{O}(\log N)$ mal auftritt, wird die Sucheffizienz des Baumes dadurch nicht beeinträchtigt.

Ein durch *bulk loading* konstruierter Index unterstützt alle Operationen des B^+-Baums mit derselben asymptotischen Komplexität. In der Praxis ist die Suchperformance sogar besser, da die Blätter des Baumes im Vergleich zu den Blättern eines mit OBO erzeugten Baumes maximal gefüllt sind: die Anzahl der zu durchsuchenden Blätter ist also insgesamt geringer.

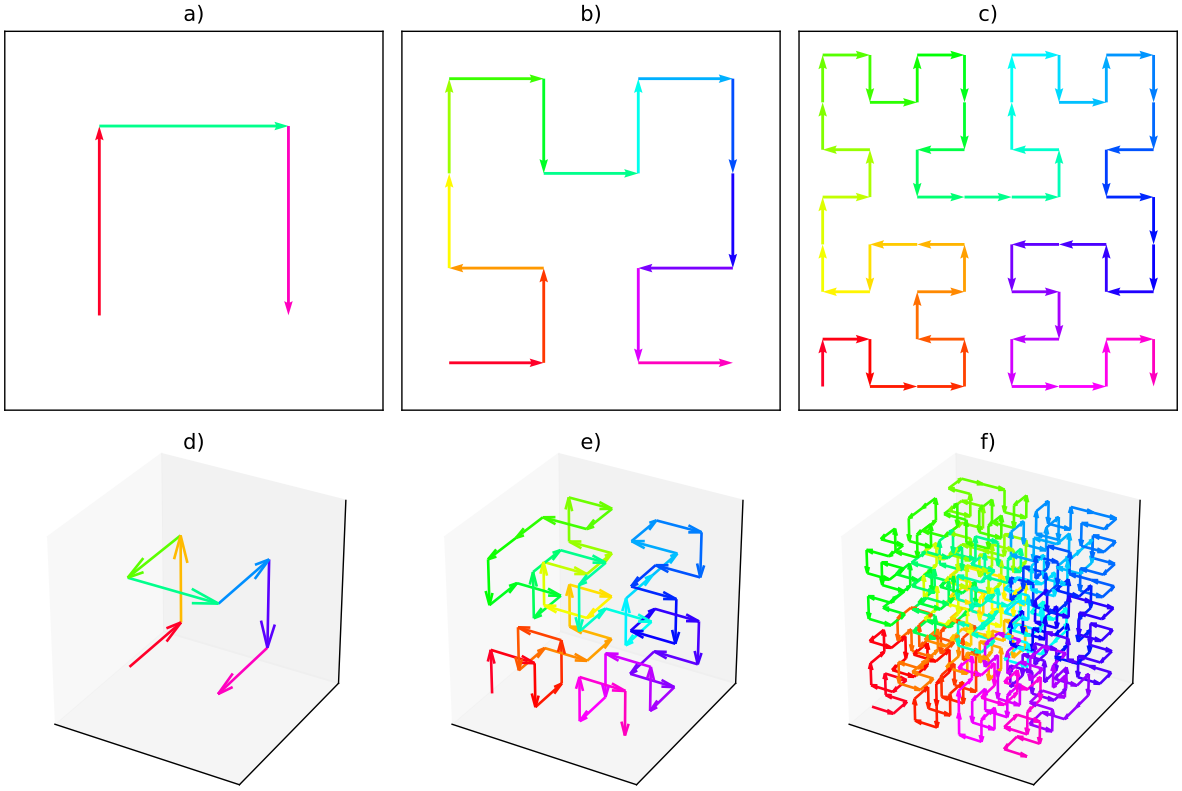


Abbildung 4.1.: Die Hilbertkurve steigender Ordnung in zwei und drei Dimensionen.

4.2. Hilbert-Packing

Im Unterschied zu eindimensionalen Objekten ist es für den mehrdimensionalen Fall nicht einfach möglich, eine gute Ordnung zu finden, sodass ein Verfahren wie in Sektion 4.1 für einen R-Baum angewendet werden kann. Würde eine gute lineare Ordnung vorliegen, so könnte man die so entstehende Sequenz den Blättern eines R-Baums zuweisen und diesen dann *bottom-up* aufbauen. Allerdings eignen sich zum Beispiel im Zweidimensionalen weder die naive Sortierung nach x oder y noch der lexikographische Vergleich dazu, eine gute Ordnung anzugeben: es ist in allen Fällen sehr einfach Situationen herzustellen, die in einem inakzeptabel hohem Overlap des resultierenden R-Baums enden [KF93].

Kamel und Faloutsos [KF93] haben die *Hilbert-Kurve* angewendet, um eine lineare Ordnung für beliebige n-dimensionale Punkte zu erhalten. Die Hilbert-Kurve ist eine von Hilbert [Hil91] entdeckte fraktale, raumfüllende Kurve. Bei unendlicher Fortführung wird das zweidimensionale Einheitsrechteck $[0, 1]^2$ vollständig abdeckt, wobei jeder Punkt in diesem Rechteck genau einmal besucht wird. Man erhält für jeden Punkt einen eindeuti-

gen Index, indem man dessen Position auf der Hilbert-Kurve betrachtet und die von der Kurve bis zu diesem Punkt zurückgelegte Distanz misst. Objekte mit einer Ausdehnung können unterstützt werden, indem man zum Beispiel als Punkt das Zentrum ihrer MBB wählt. Die Hilbert-Kurve kann ohne Weiteres auf das Hyperrechteck $[0, 1]^n$ verallgemeinert werden.

Da die Hilbert-Kurve unendlich viele Punkte besucht, ist für praktische Berechnungen nur H_i, die Hilbert-Kurve i-ter Ordnung, relevant. H_i besucht in jeder Dimension des Rechtecks 2^i verschiedene Koordinaten, sodass in $[0, 1]^n$ insgesamt 2^{ni} verschiedene Punkte erreicht werden. Zu jedem dieser endlich vielen Punkte gibt es genau einen *Hilbert-Index*: den Rang des Punktes auf der Kurve. Zu einem beliebigen Punkt in $[0, 1]^n$ erhält man dessen Hilbert-Index, indem man den nächsten von H_i besuchten Punkt lokalisiert und dessen Index berechnet. Offensichtlich ist dieser Index nun nicht mehr eindeutig. Sortiert man Punkte nach ihrem Hilbert-Index, erhält man eine lineare Ordnung, bei der aufeinanderfolgende Punkte auch räumlich relativ nahe beieinander liegen. Die Qualität dieser Ordnung ist von i abhängig: eine höhere Anzahl verfügbarer Hilbert-Indizes ergibt eine feinere Unterteilung des Raums und somit auch wahrscheinlich ein besseres Ergebnis des folgenden Algorithmus.

Abbildung 4.1 zeigt die Kurven H_1, H_2 und H_3 in zwei (a-c) und drei (d-f) Dimensionen. Gezeigt wird jeweils das Einheitsrechteck $[0, 1]^2$ bzw. $[0, 1]^3$. Die Pfeile geben die Reihenfolge an, in der die Kurven die verschiedenen Punkte besuchen. Die Farbe dient dabei als zusätzliche Orientierung und verläuft in den Regenbogenfarben von Rot (Anfang der Kurve) bis Violett (Ende der Kurve). Die Anzahl der verschiedenen Indizes steigt rasant: schon H_3 besucht im dreidimensionalen Raum $2^9 = 512$ Punkte.

Der von Kamel und Faloutsos [KF93] entwickelte Algorithmus HilbertPack zum Packen eines R-Baums mit Fan-out b mit einer Menge von N Objekten lautet schließlich wie folgt:

1. Berechne für jedes Objekt dessen Hilbert-Index.
2. Sortiere die Objekte bezüglich ihres Hilbert-Index.
3. Traversiere die sortierte Folge und fasse jeweils b aufeinanderfolgende Objekte zu einem Blatt zusammen.
4. Solange in der letzten Ebene mehr als ein Knoten konstruiert wurde: Erstelle eine neue Ebene interner Knoten, indem jeweils b Knoten der letzten Ebene zu Kindern

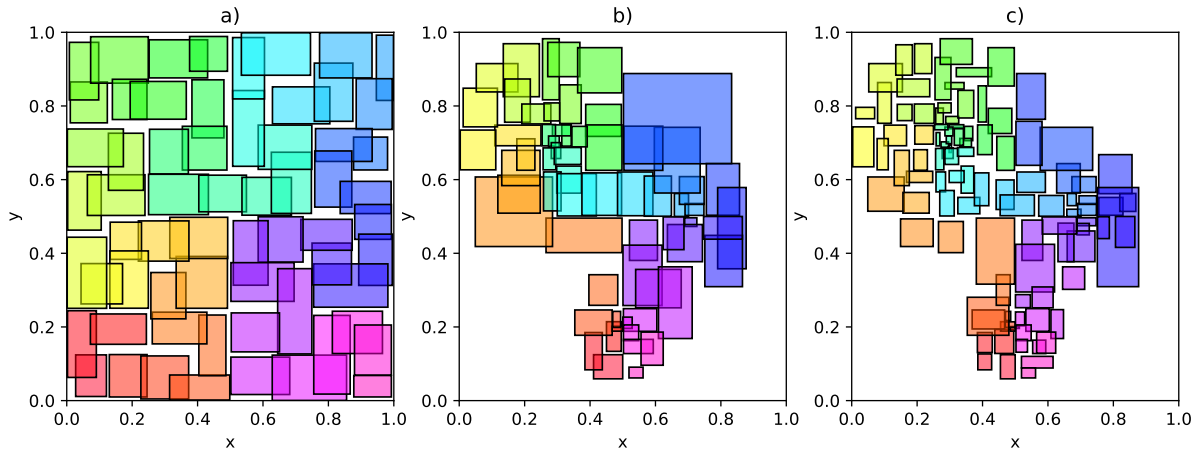


Abbildung 4.2.: Durch Hilbert-Packing entstandene Blätter eines R-Baums.

eines neuen internen Knotens werden. Gehe dabei in der Reihenfolge vor, in der die Knoten des letzten Levels erzeugt wurden.

Wie im Falle des *bulk-loadings* von B$^+$-Bäumen wird hier jeder Knoten nur genau einmal bei dessen Konstruktion geschrieben, man erhält daher wieder einen Leistungsgewinn aus dem gebündelten Schreiben und der Vermeidung von Splits. Hamilton und Rau-Chaplin beschreiben effiziente Algorithmen für die Berechnung von Hilbert-Indizes zu beliebigen Punkten und deren Umkehrung, sodass Schritt 1 in $\mathcal{O}(\frac{N}{B})$ I/Os (und nur $\mathcal{O}(N)$ Rechenoperationen) durchführbar ist [HR07], [Ham06].[1] Für die asymptotische Laufzeit von HilbertPack dominiert also wieder der Sortierschritt.

In Abbildung 4.2 sind die mit diesem Verfahren erzeugten Blattknoten dargestellt. In allen Fällen wurden Blätter mit maximaler Größe $b = 16$ aus $N = 1000$ über das Intervall $[0, 1]^2$ verteilten Punkten konstruiert. In Grafik a) sind die Punkte gleichmäßig auf das Rechteck verteilt worden. Der Overlap zwischen den Blättern ist gering und der Raum ist gut partitioniert. Anhand der Farben lässt sich der Verlauf der Hilbert-Kurve gut nachvollziehen, siehe Abbildung 4.1. In Grafik b) sind die Punkte zufällig über drei Kreise mit den Zentren (0.3, 0.7), (0.5, 0.2) und (0.7, 0.5) verteilt, wobei sich die Punkte eines Kreises zum Mittelpunkt hin häufen. Dieser Fall veranschaulicht einen Nachteil des Algorithmus: Wenn sich die Hilbert-Indizes zweier aufeinanderfolgender Objekte stark unterscheiden, so werden sie trotzdem in einem Blatt zusammengefasst, welches als Folge dessen sehr groß werden kann.

[1] Die Dimensionalität n und die gewählte Ordnung der Hilbert-Kurve i werden als konstant vorausgesetzt.

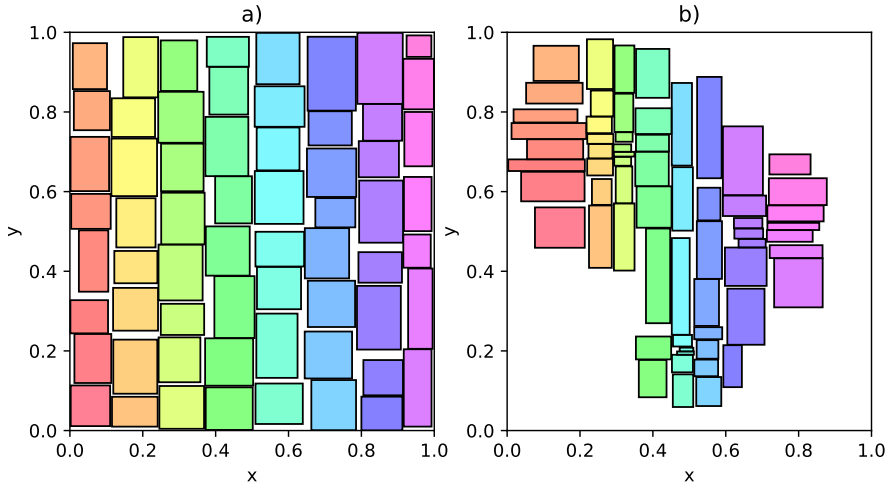


Abbildung 4.3.: Durch die Anwendung von STR gepackte Blätter eines R-Baums.

Eine von DeWitt u. a. [DeW+94] vorgeschlagene Heuristik kann in dieser Situation Abhilfe schaffen: anstatt jedes der Blätter maximal zu füllen, wird jedes Blatt zunächst nur bis zu einem bestimmten Anteil mit Einträgen besetzt. Weitere Einträge werden nur dann hinzugenommen, wenn sie die MBB des Blattes nicht zu sehr vergrößern. Es handelt sich hierbei um ein gieriges Vorgehen – es gibt keine Garantie, dass das lokale Kriterium der Vergrößerung in global optimalen (oder auch nur guten) Blättern resultiert. Das Ergebnis dieses Verfahrens, angewendet auf das Datenset von Abbildung 4.2 b), ist in Abbildung 4.2 c) zu sehen. Die Blätter wurden zunächst zu 50 % gefüllt. Weitere Einträge kamen hinzu, wenn sie die ursprüngliche Bounding Box des Blattes um nicht mehr als 20 % vergrößerten. Die dargestellten Blätter sind daher nur zu etwa 60 % voll; ihre Anzahl ist deshalb deutlich höher. Die MBBs der Blätter sind im Mittel wesentlich kleiner und der Overlap konnte an vielen Stellen reduziert werden.

4.3. Sort-Tile-Recursive

Der Algorithmus SortTileRecursive (STR) von Leutenegger, Edgington und Lopez ist ein ebenfalls auf Sortierung basierendes Verfahren zum gebündelten Laden einer Datenmenge in einen R-Baum. Mehrdimensionale Objekte werden nach verschiedenen Kriterien sortiert, wobei nach jedem Sortiervorgang die Datenmenge rekursiv in kleinere Kacheln zerlegt wird [LEL97]. Die Rekursion stoppt, wenn nach jedem Kriterium sortiert wurde. Der Algorithmus ist so konzipiert, das die Kacheln zu genau diesem Zeitpunkt die Größe

eines Blattes besitzen, sodass sie direkt in den Baum geladen werden können.

Algorithmus 1 Der STR-Algorithmus.

Input:

$[e_1, ..., e_N]$ Eine Liste von Einträgen, aus denen Blätter erzeugt werden sollen.
b Die Größe eines Blattes.
k Die Dimension der Objekte.
$[c_1, ..., c_k]$ Eine Liste von Vergleichsfunktionen (eine pro Dimension).

Output:

Die Einträge sind so geordnet, dass sie in linearer Reihenfolge gute Blätter ergeben.

```
1:  function SortTileRecursive([e_1, ..., e_N], b, k, [c_1, ..., c_k])
2:      Sort([e_1, ..., e_N], c_1)                    ▷ Sortiere bezüglich c_1.
3:      if k > 1 then
4:          P := ⌈N/b⌉                                ▷ Anzahl der Blätter.
5:          S := ⌈P^(1/k)⌉                            ▷ Anzahl der Kacheln.
6:          L := b · ⌈P^((k-1)/k)⌉                    ▷ Anzahl der Einträge pro Kachel.
7:          i := 1
8:          while i ≤ N do                            ▷ Erzeuge Kacheln der Länge L.
9:              j := min(N, i + L)
10:             SortTileRecursive([e_i, ..., e_j], b, k − 1, [c_2, ..., c_k])
11:             i := j + 1
```

Die Herangehensweise von STR ist in Algorithmus 1 zu sehen. Nach der Sortierung in Zeile 2 wird die Liste der Einträge rekursiv zerlegt, sofern ihre Dimension größer als eins ist. Die Variablen in den darauffolgenden Zeilen werden so bestimmt, dass jede Tiefe der Rekursion eine ungefähr gleiche Anzahl an Kacheln produziert: gibt es insgesamt N Einträge der Dimension k, so werden jeweils ca. $\lceil \sqrt[k]{\frac{N}{b}} \rceil$ Kacheln erzeugt. Durch den rekursiven Ausführungsschritt in Zeile 10 wird jede Kachel als ein alleinstehendes Datenset mit geringerer Dimension behandelt. Nachdem die Einträge mit SortTileRecursive vorbereitet wurden, wird der Baum wie beim Algorithmus HilbertPack *bottom-up* konstruiert, indem die Knoten einer Ebene rekursiv solange zu Knoten höherer Ebene gruppiert werden, bis nur noch ein Knoten, die Wurzel, verbleibt. Wegen der mehrfachen Sortierschritte ist STR im Vergleich zu HilbertPack langsamer.

Abbildung 4.3 zeigt die vom STR-Algorithmus erzeugten Blätter, angewendet auf dasselbe Datenset wie in Abbildung 4.2. Die zweidimensionalen Punkte wurden zuerst nach $<_x$ und dann bezüglich $<_y$ sortiert und schließlich zu Blättern mit $b = 16$ Einträgen zusammengefasst. In x-Richtung entstanden so 8 Kacheln, welche in y-Richtung wiederum in 8 Blätter

zerlegt wurden. Die Farben geben wieder die Reihenfolge an, in der die Blätter erstellt wurden. Da es sich hier um Punktdaten handelt, kann STR Blätter ohne jeden Overlap erzeugen; für Objekte mit räumlicher Ausdehnung, z.B. nach dem Zentrum ihrer MBBs sortiert, wäre das Resultat im Zweifel nicht so gut. Da der Algorithmus nur unter der Verwendung beliebiger Vergleichsfunktionen definiert ist, kann er ohne Schwierigkeiten um weitere Kriterien erweitert werden.

4.4. Quickload

Der Quickload-Algorithmus ist ein von Bercken und Seeger [BS01, Algorithmus 2] entwickeltes *bottom-up*-Verfahren, um einen R-Baum (oder ähnliche Indizes) effizient zu laden. Im Unterschied zu den vorher beschriebenen Algorithmen basiert Quickload nicht auf der Sortierung der Datenmenge, sondern verwendet die schon vorhandenen Funktionen des Baums. Der einzige zusätzliche Parameter, der den Ressourcenverbrauch des Algorithmus steuert, ist $C \leq \frac{M}{B}$, die Anzahl der Blätter, die maximal im internen Speicher gehalten werden.[2]

Die zugrundeliegende Idee von Quickload ist das rekursive Zerlegen der Datenmenge in kleinere Teilprobleme. Dabei werden für die räumliche Partitionierung der Daten die klassischen Algorithmen des R-Baums wiederverwendet. Im Hauptspeicher wird ein temporärer R-Baum schrittweise mittels *one-by-one insertion* mit Einträgen gefüllt, bis dessen Blattanzahl den Wert C erreicht. Da alle Operationen im Hauptspeicher stattfinden, lässt sich hier eine akzeptable Leistung erreichen. Alle übrig bleibenden Einträge werden nicht mehr direkt in ein Blatt eingefügt. Stattdessen wird für jedes Blatt ein Puffer erzeugt, welcher überschüssige Einträge auffangen soll. Der Puffer hält den aktuellen zu beschreibenden Block im Speicher; um dafür Speicherplatz zu schaffen, werden die Blätter in den Sekundärspeicher ausgelagert (dies ist ohne Weiteres möglich, da sie nicht mehr modifiziert werden). Im Anschluss wird ein modifizierter Einfügealgorithmus eingesetzt: Es wird weiterhin der Baum traversiert, um ein geeignetes Blatt zu finden. Die MBBs der Vorgänger dieses Blattes werden angepasst, um auch den neuen Eintrag vollständig zu enthalten. Der Eintrag wird aber nicht in das Blatt eingefügt, sondern an dessen Puffer angehängt.

[2] Ein Knoten entspricht einem Speicherblock. Die Anzahl der für die Blätter benötigten internen Knoten sollte beim Bestimmen eines guten Wertes für C miteinbezogen werden.

Nachdem alle Einträge so verarbeitet wurden, wird für jedes $\langle blatt, puffer \rangle$-Paar zwischen zwei Fällen unterschieden:

- Der Puffer ist leer. Das Blatt wird unverändert übernommen und in einer I/O-Operation in den zu konstruierenden R-Baum im Sekundärspeicher eingefügt.
- Ein Blatt verfügt über einen nicht-leeren Puffer. Der Inhalt des Blattes und des Puffers wird als eine alleinstehende Datenmenge betrachtet und rekursiv von Quickload bearbeitet. Dazu werden die Adresse des Blattes und des Puffers in eine Warteschlange namens *todo* eingefügt.

Nach Abschluss dieser Operationen werden die internen Knoten zerstört: sie dienen nur dem *routing* der Einträge und können im fertigen Baum nicht verwendet werden. Ist *todo* nicht leer, so wird ein $\langle blatt, puffer \rangle$-Tupel aus der Warteschlange entfernt und der Algorithmus wird für diese Datenmenge wiederholt.

Man erhält auf diese Art und Weise zunächst nur die Blattknoten eines Baums. Um auch höhere Ebenen zu konstruieren, wird für jeden Knoten einer Ebene ein $\langle mbb, ptr \rangle$-Tupel konstruiert. Dabei ist *mbb* die MBB des Knotens und *ptr* dessen Adresse im externen Speicher. Wendet man Quickload nun auf die so entstandene Liste von Tupeln an, erhält man wieder eine Reihe von Knoten, die sich diesmal direkt als interne Knoten einer höheren Ebene in den R-Baum einfügen lassen. Nach diesem Schema wird fortgefahren, bis nur noch ein Knoten übrig bleibt. Der zur Verfügung stehende Platz im Hauptspeicher wird mit diesem Schema nicht optimal ausgenutzt. Ein R-Baum garantiert als minimalen Fan-out nur weniger als die Hälfte des maximalen Fan-outs[3]; es wird also gegebenenfalls ein großer Anteil der Speicherkapazität verschwendet.

Die Laufzeit des Algorithmus ist stark von der Datenmenge abhängig: die Qualität der räumlichen Unterteilung, die vom internen R-Baum geleistet wird, ist vom initialen *sample* abhängig. Sind die Einträge, mit denen die C Blätter konstruiert werden, nicht repräsentativ für die gesamte Datenmenge, so wird die Leistung von Quickload schlecht sein. Im schlimmsten Fall werden nach der Erstellung der C Blätter im Hauptspeicher alle weiteren Einträge in den Puffer von nur einem einzigen Blatt geschrieben, sodass der Algorithmus rekursiv auf eine kaum reduzierte Problemgröße angewendet werden muss. In diesem Fall liegt mit $n := \frac{N}{B}$ und $m := \frac{M}{B}$ die Anzahl der I/Os in $\mathcal{O}(\frac{n^2}{m})$. Die im Mittel erwartete Anzahl der I/Os liegt allerdings laut Bercken und Seeger wie bei der externen Sortierung in $\mathcal{O}(n \log_m n)$ [BS01].

[3] In dieser Arbeit wird $\frac{1}{3}$ des max. Fan-outs verwendet.

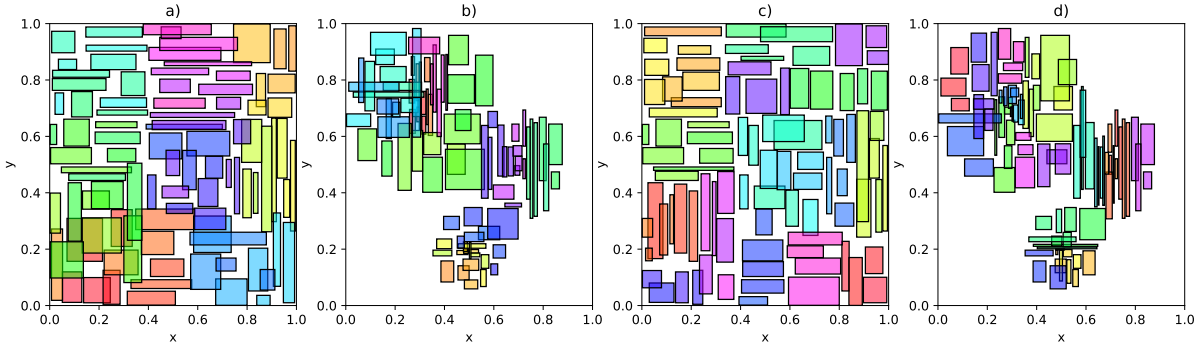


Abbildung 4.4.: Ein Vergleich der Ergebnisse von OBO (a, b) und Quickload (c, d).

Trotz der im schlimmsten Fall quadratischen Laufzeit kann der Algorithmus so umgesetzt werden, dass er nur $\mathcal{O}(n)$ Speicherplatz im externen Speicher belegt. Nach jedem Durchlauf des Algorithmus befindet sich jeder Eintrag der Eingabe entweder schon in einem fertigen Blatt des zur konstruierenden Baums oder in der Warteschlange für weitere rekursive Aufrufe. Es kann daher der Speicherplatz für die Eingabe[4] nach einem Aufruf freigegeben werden. Auf diese Art und Weise befindet sich jeder Eintrag nur an genau einer Stelle und es werden nur $\mathcal{O}(n)$ Blöcke zur Speicherung benötigt.

Die Implementierung von Quickload ist einfach: es können die herkömmliche Algorithmen des R-Baums wiederverwendet werden. Der Baum muss nur um die Funktionalität erweitert werden, C Blätter und die dazu notwendigen internen Knoten gleichzeitig im Hauptspeicher zu behalten. Neue Komplexität entsteht durch die Verwaltung der Puffer und die Anforderung, nicht mehr verwendete Blätter in den Sekundärspeicher auszulagern. Wegen der Wiederverwendung des klassischen Einfügealgorithmus besitzt der resultierende Baum eine mit dem Ergebnis des OBO-Algorithmus vergleichbare Struktur. Das ist für diese Arbeit von Vorteil, da sich die in Kapitel 5 speziell für den IRWI-Baum angepasste hybride Kostenfunktion direkt wiederverwenden lässt. Mit Quickload kann ein IRWI-Baum deutlich schneller als mit OBO konstruiert werden, der trotzdem ähnlich gute Antwortzeiten aufweist (siehe Kapitel 7).

In Abbildung 4.4 ist das Ergebnis von OBO (a und b) bzw. von Quickload (c und d) zu sehen. Es wurden wieder dieselben Punktdaten wie in Abbildung 4.2 und 4.3 verwendet. Quickload wurde mit $C = 16$ Blättern konfiguriert; es passte also nur ein Bruchteil der Daten gleichzeitig in den Hauptspeicher, sodass mehrere rekursive Schritte unternommen werden mussten. Die Resultate der beiden Algorithmen sind wegen der Verwendung

[4] Dies ist entweder die ursprüngliche Eingabe, also die Folge aller N Elemente, oder ein Puffer aus einem früheren Aufruf.

desselben Algorithmus für das Einfügen einzelner Einträge sehr ähnlich.

4.5. TPA

Der *Two-Phase Partition Algorithm* von Li u. a. [Li+13] verwendet im Unterschied zu den bisher beschriebenen Verfahren einen *top-down*-Ansatz. TPA unterstützt nicht nur rein räumliche Daten, sondern ist zur Konstruktion eines IR-Baums [CJW09], [Li+11] geeignet. Ein IR-Baum ist ein R-Baum, in die Knoten um einen *invertierten Index* erweitert wurden, der den textuellen Inhalt der Kindknoten indiziert und somit nicht nur die Suche nach räumlichen Bereichen, sondern auch nach Schlüsselwörtern ermöglicht. Häufig werden bei Suchen in einem IR-Baum beide Kriterien miteinander kombiniert, um z.B. alle Dokumente in einem räumlichen Bereich zu finden, die ein gewisses Schlüsselwort enthalten.

Der Algorithmus geht wie folgt vor: der Datensatz wird rekursiv immer zuerst bezüglich der x- und dann bezüglich der y-Koordinaten partitioniert. In einem Folgeschritt werden die so entstandenen Gruppen weiter bezüglich der in ihnen enthaltenen Wörter aufgeteilt. Dieses Verfahren wird rekursiv solange fortgeführt, bis die Datenmenge klein genug ist, um in ein Blatt des Baums zu passen. Auf diese Art und Weise soll einerseits der Overlap zwischen den MBBs der Kinder eines Knotens gering gehalten werden und andererseits durch eine möglichst geringe Anzahl von Schlüsselwörtern pro Kindknoten eine effektive Suche nach diesen ermöglicht werden.

Wegen der Miteinbeziehung der Schlüsselwörter bei der Konstruktion des Baums ist TPA ein guter Kandidat für die Anwendung auf den in Kapitel 5 beschriebenen IRWI-Baum, welcher wiederum ein erweiterter IR-Baum ist. Da sich der Ansatz des Algorithmus allerdings stark von den anderen Verfahren unterscheidet, wurde eine Implementierung in dieser Arbeit aus Zeitgründen nicht versucht. Es kann außerdem argumentiert werden, dass die in dieser Arbeit verwendete Variante von Quickload (siehe Seite 58) ein ähnliches Ergebnis produziert, da die Verwendung der hybriden Kostenfunktion auch einen Kompromiss zwischen der Anzahl der Schlüsselwörter pro Kindknoten und der räumlichen Trennung dieser herstellt.

5. Der IRWI-Baum

Der von Issa und Damiani [ID16] beschriebene IRWI-Baum (*IR-With-Identifiers*) ist ein R-Baum, der in seinen Blättern die Abschnitte von spatio-textuellen Trajektorien speichert. Eine einzelne Trajektorie wird abgelegt, indem alle ihre *Units* separat in den Baum eingefügt werden. Dabei wird zusätzlich für jede Unit festgehalten, zu welcher Trajektorie sie gehört und an welcher Position sie sich innerhalb der Trajektorie befindet. Wie ein IR-Baum [CJW09; Li+11] erfassen Knoten die textuellen Inhalte ihrer Teilbäume in einem *invertierten Index*. Jeder interne Knoten des IRWI-Baums verfügt über eine Reihe invertierter Listen, die zu einem bestimmten *Label* diejenigen Kinder aufführen, die über dieses Label verfügen. Im Unterschied zum IR-Baum enthalten die invertierten Listen zusätzlich pro Kind eine komprimierte Darstellung der Trajektorien-IDs, die in dem Teilbaum dieses Kindes enthalten sind. Diese Datenstrukturen werden verwendet, um die parallele Auswertung von sequentiellen Anfragen effizienter zu gestalten.

5.1. Die Struktur des Baums

Jedes Blatt eines IRWI-Baums enthält eine Reihe von Einträgen der Form $\langle tid, index, unit \rangle$. *unit* ist die spatio-textuelle Unit und besteht aus den Attributen $\langle I, label, p, q \rangle$. I ist das von *unit* repräsentierte Zeitintervall und p bzw. q sind die zweidimensionalen Punkte der räumlichen Trajektorie zu Beginn bzw. am Ende des Zeitintervalls. $label \in \mathcal{L}$ ist der Wert der textuellen Trajektorie während des Zeitintervalls I und $\mathcal{L}$ ist die Menge, in der alle Label enthalten sind. Durch *tid* und *index* wird ein Blatteintrag einer Trajektorie zugeordnet: *tid* ist deren eindeutiger Identifikator und *index* ist der Index der Unit innerhalb der Trajektorie. Die dreidimensionale MBB eines Blatteintrags ist implizit durch das Zeitintervall I und die beiden Punkte p, q gegeben: die Zeit wird als dritte Dimension behandelt. Im Gegensatz zu internen Knoten verfügen Blätter über keinen invertierten Index.

	Unit			
Trajektorie	1	2	3	4
1	train	train	bike	
2	car	car	car	car
3	car	walk	walk	walk

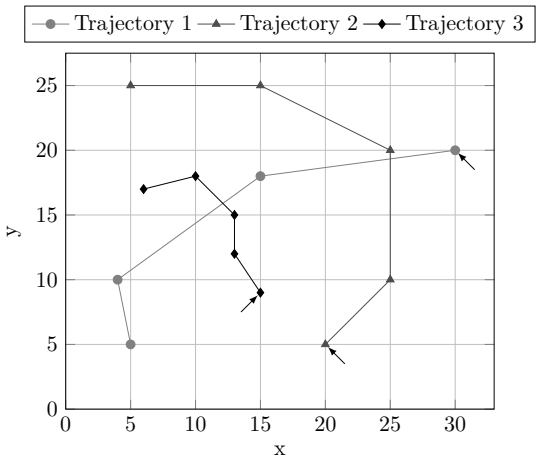


Abbildung 5.1.: Der Verlauf von drei räumlich-textuellen Trajektorien.

Die Einträge eines internen Knotens bestehen wie bei einem R-Baum aus den Werten $\langle mbb, ptr \rangle$, wobei *ptr* auf einen Kindknoten zeigt und *mbb* dessen Minimum Bounding Box repräsentiert. Zusätzlich verfügt jeder interne Knoten über eine Referenz auf seinen *invertierten Index*. Der invertierte Index enthält für jedes Label l, welches in mindestens einer Unit innerhalb des Teilbaums genannt wird, eine *Posting List*. Diese Liste speichert *Postings* der Form $\langle child, count, ids \rangle$ für jedes Kind des Knotens, in dessen Teilbaum l auftritt. Dieses Kind des Knotens wird eindeutig durch den Index *child* identifiziert. Der Zähler *count* hält die Gesamtanzahl der Units im Teilbaum vom *child* fest, die l als Label besitzen; *ids* ist eine komprimierte Darstellung der Menge der Identifikatoren aller dieser Units (siehe Sektion 5.2).

Zusätzlich enthält der invertierte Index einen Eintrag für das virtuelle Label „Total". Die dazugehörige Liste speichert für jedes Kind in genau einem Posting die Summe bzw. die Vereinigung aller *count*- bzw. *ids*-Felder der anderen Listen. Man erhält so für jedes Kind die Gesamtanzahl von Units in dessen Teilbaum und eine Menge, die die IDs aller in diesem Teilbaum gespeicherten Trajektorien enthält. Der Zweck dieser Liste ist das Unterstützen von Anfragen mit beliebiger textuellen Komponente und der effiziente Zugriff auf die Größe eines Teilbaums, welche vom Einfügealgorithmus benötigt wird (Sektion 5.3). Issa und Damiani verwenden für diese Liste stattdessen das Symbol „_".

Abbildung 5.1 zeigt als Beispiel den Verlauf von drei räumlich-textuellen Trajektorien. Die Grafik auf der rechten Seite zeigt die räumlichen Entwicklung – um die Präsentation übersichtlich zu halten, ist die Zeitkomponente nicht dargestellt. Die Startpunkte der Trajektorien sind durch Pfeile markiert. Jede Trajektorie beginnt zum Zeitpunkt

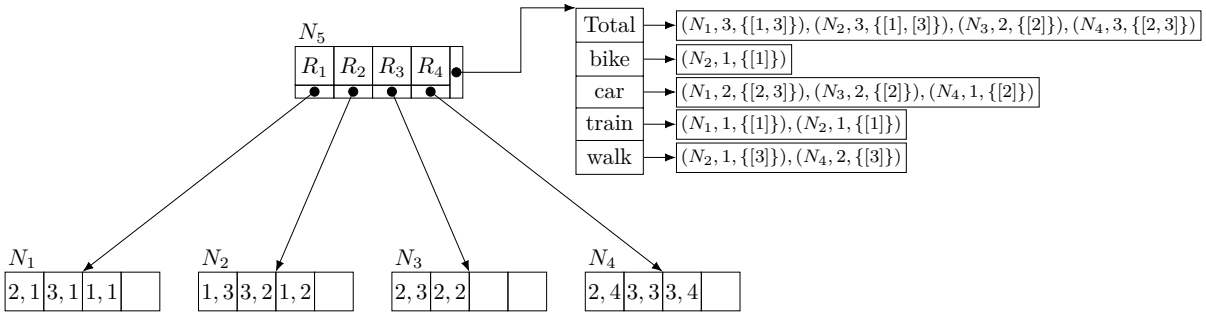


Abbildung 5.2.: Schematische Darstellung eines IRWI-Baums.

0, die Zeitspanne zwischen zwei aufeinanderfolgenden Punkten ist genau 5 Sekunden. Die Tabelle auf der linken Seite zeigt die textuelle Komponente der Trajektorien. Jeder Unit einer Trajektorie ist ein Label zugeordnet worden (Unit i verläuft zwischen den räumlichen Punkten mit Index i und $i + 1$), in diesem Fall handelt es sich um das verwendete Transportmittel. Die vollständige Liste der Units von Trajektorie 1 lautet damit $\langle [0, 5), (30, 20), (15, 18), \text{train} \rangle$, $\langle [5, 10), (15, 18), (4, 10), \text{train} \rangle$, $\langle [10, 15), (4, 10), (5, 5), \text{bike} \rangle$.

In Abbildung 5.2 ist ein IRWI-Baum dargestellt, der durch aufeinanderfolgendes Einfügen der drei Trajektorien entstanden ist. Die Blatteinträge sind abgekürzt als Paare *tid*, *index*. Der invertierte Index des Knotens N_5 enthält für jedes der vier Label eine Posting List. Der einzige Eintrag $(N_2, 1, \{[1]\})$ zum Label „bike" besagt, dass im Teilbaum von N_2 genau eine Unit über dieses Label verfügt und dass die einzige Trajektorie die ID 1 besitzt. Die räumliche Struktur des Baums ist in Abbildung 5.3 zu sehen. Gezeigt werden die MBBs $R_1, ..., R_4$, die im Knoten N_5 zu den Blättern gespeichert werden und die einzelnen, dreidimensionalen Liniensegmente, die in diesen Blättern abgelegt wurden. Mithilfe des invertierten Index und des R-Baums kann der IRWI-Baum gleichzeitig nach einem räumlichen Bereich R und nach einem Label l durchsucht werden: Bei dem Durchlaufen des Baums werden immer nur solche Knoten besucht, welche laut Index das Label l enthalten und deren MBB R überlappt. Die im invertierten Index abgelegten ID-Mengen werden an dieser Stelle noch nicht genutzt (siehe Sektion 5.4).

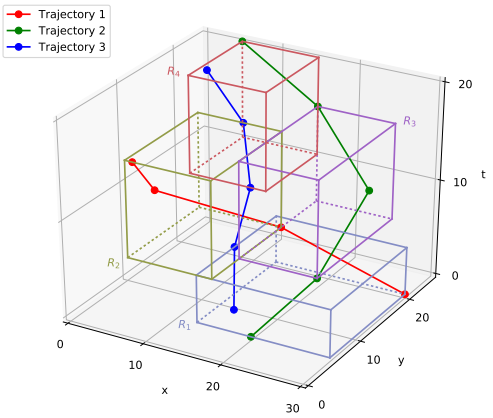


Abbildung 5.3.: Die räumliche Struktur des IRWI-Baums.

5.2. Repräsentation der Identifikatoren

Issa und Damiani [ID16] beschreiben zur Repräsentation der *Identifikatoren* (IDs) von Trajektorien im invertierten Index der internen Knoten eine kompakte, auf Intervallen basierende Datenstruktur. Sie geben allerdings keine Algorithmen für die Arbeit mit dieser Datenstruktur an; diese werden im Folgenden detailliert. Es wird davon ausgegangen, dass die ID einer Trajektorie durch eine natürliche Zahl dargestellt werden kann. Die Speicherung von Identifikatoren findet in Mengen statt, die als Folge von Intervallen repräsentiert werden. Die Algorithmen dieser Sektion beschäftigen sich nicht mit I/O-Operationen; die hier angegebenen Laufzeiten beziehen sich daher nur auf Anzahl der durchgeführten Rechenoperationen.

Definition 3. Ein *Intervall-Set* $\mathcal{I} = I_1, ..., I_n$ ist eine geordnete Folge endlicher, nicht-überlappender Intervalle. Jedes Intervall hat die Form $I_i = [a_i, b_i] \subset \mathbb{N}$ mit $a_i \leq b_i$ für $1 \leq i \leq n$. Die Intervalle folgen aufeinander: für alle $1 < i \leq n$ gilt $a_i > b_{i-1}$.

Ein Intervall-Set modelliert eine gewöhnliche, endliche Menge natürlicher Zahlen: $x \in \mathbb{N}$ ist genau dann ein Element von $\mathcal{I}$, wenn es in der Vereinigung der n Intervalle enthalten ist, also $x \in \mathcal{I} \Leftrightarrow \exists i$ mit $1 \leq i \leq n : x \in [a_i, b_i]$. Mithilfe von binärer Suche findet man ein solches Intervall in $\mathcal{O}(\log n)$ durchgeführten Vergleichen oder schließt dessen Existenz aus. Dies ist möglich, da die Intervalle sich nicht überlappen und sortiert sind.

Das Einfügen eines einzelnen Wertes x ist trivial: Eine binäre Suche nach x wird durchgeführt. Wird ein Intervall gefunden, welches x schon enthält, ist nichts zu tun. Ansonsten wird an dieser Stelle ein neues Intervall $[x, x]$ eingefügt. Grenzt das neue Intervall an einen oder an beide seiner direkten Nachbarn an, so können die Intervalle ohne Informationsverlust miteinander vereinigt werden.

Intervall-Sets werden in dieser Arbeit als sortierte Arrays im Speicher repräsentiert. Diese Darstellungsform bietet sich aufgrund der üblicherweise geringen Größe und wegen der Vermeidung von Overhead bei der häufig stattfindenden (De-) Serialisierung an: die Formen der Intervall-Sets im Haupt- und Sekundärspeicher sind identisch. Die Organisation in einem klassischen binären Suchbaum ist eine mögliche Alternative, die für größere Sets zu empfehlen ist, da das Einfügen in ein sortiertes Array $\mathcal{O}(n)$ Operationen benötigt, was es für große Datenmengen inakzeptabel macht. Beide Fälle sind sehr einfach zu implementieren, da sich die Intervalle eines Sets nicht überlappen und daher eine natürliche totale Ordnung existiert.

5.2.1. Schnitt und Vereinigung

An vielen Stellen in dieser Arbeit müssen die Schnitte oder die Vereinigungen einer großen Anzahl von Intervall-Sets gebildet werden. Die dazu nötigen effizienten Algorithmen basieren auf einer Kombination des *Sweep-Line*-Paradigmas mit dem *k-way merge*, welcher ansonsten unter anderem bei der Sortierung großer Datenmengen im externen Speicher Anwendung findet.

Definition 4. Für ein Intervall-Set $\mathcal{I} = I_1, ..., I_n$ sind dessen *Intervall-Events* die geordnete Folge von Punkten, an denen eine Intervallgrenze (Öffnung oder Schließung) liegt.

$$\begin{aligned} \mathit{events}(\mathcal{I}) &:= \mathit{events}([a_1, b_1], ..., [a_n, b_n]) \\ &:= (\text{„open”}, a_1), (\text{„close”}, b_1), ..., (\text{„open”}, a_n), (\text{„close”}, b_n) \end{aligned} \tag{5.1}$$

Die Berechnung von $\mathit{events}(\mathcal{I})$ kann in $\mathcal{O}(n)$ Zeit stattfinden, da die Intervalle schon per Definition aufsteigend nach ihren Intervallgrenzen sortiert sind.

Algorithmus 2 Berechnung der gemeinsamen Events für beliebig viele Intervall-Sets.

Input:

$[\mathcal{I}_1, ..., \mathcal{I}_m]$ Eine Liste von m Intervall-Sets.

Output:

result Die vereinigte Liste aller Intervallgrenzen, in aufsteigender Reihenfolge.

```
1:  function MergedEvents([I_1, ..., I_m])
2:      result := Erzeuge leere Liste.
3:      heap := Erzeuge leeren Heap.          ▷ Siehe Text für die Priorität der Elemente.
4:      for i := 1, ..., m do                 ▷ Repräsentiere die einzelnen Sets im Heap.
5:          iter := Iterator für events(I_i).
6:          if ¬atEnd(iter) then
7:              push(heap, iter)
8:      while ¬empty(heap) do                 ▷ Besuche die geordneten Events.
9:          iter := pop(heap)
10:         event := getCurrent(iter)
11:         append(result, event)             ▷ Speichere das aktuelle Event und
12:         moveNext(iter)                        bewege den Iterator nach vorn.
13:         if ¬atEnd(iter) then
14:             push(heap, iter)
15:     return result
```

Algorithmus 2 zeigt die Funktion MergedEvents, welche als Grundlage für die beiden späteren Sweep-Line Algorithmen dient. Für eine Liste von beliebig vielen Intervall-Sets

liefert MergedEvents eine einzelne Liste *aller* Events in aufsteigender Reihenfolge. Dazu bedient sich die Funktion der beiden Datentypen *Heap* und *Iterator*. Ein binärer (Min-)Heap ordnet die in ihm gespeicherten m Elemente so an, dass das Entfernen des Elements mit geringster Priorität (Operation *pop*) und das Hinzufügen eines neuen Elementes (Operation *push*) nur $\mathcal{O}(\log m)$ Vergleichs- und Tauschoperationen benötigt. Ein Iterator wird benutzt, um alle Elemente einer Folge zu besuchen, und unterstützt die Operationen *atEnd*, *getCurrent* und *moveNext*. *atEnd* ist genau dann wahr, wenn der Iterator alle Elemente der Folge besucht hat. *getCurrent* liefert das aktuelle Element und kann nur dann benutzt werden, wenn der Iterator noch nicht auf das Ende der Folge zeigt. *moveNext* hat dieselbe Voraussetzung und bewegt den Iterator auf das nächste Element. Ein neu erzeugter Iterator zeigt für eine nicht-leere Folge anfangs auf der erste Element und ansonsten auf das Ende.

MergedEvents beginnt, indem es jedes der nicht-leeren Intervall-Sets durch einen Iterator im Heap repräsentiert. Jeder der Iteratoren besucht Schrittweise jedes Intervall-Event seines Sets. Da die Events jedes individuellen Sets bereits in sortierter Reihenfolge vorliegen, muss an dieser Stelle nur noch ein *merge*-Schritt durchgeführt werden, d.h. die Events der verschiedenen Sets müssen in der korrekten Reihenfolge besucht werden. Für diesen Punkt ist die Anordnung der Iteratoren im Heap kritisch: Ein Iterator a besitzt dann eine geringere Priorität als ein Iterator b, wenn die Koordinate des Events getCurrent(a) kleiner ist als die von getCurrent(b). Sind ihre Koordinaten gleich, so verfügen Events mit Typ „open" über eine geringere Priorität als solche mit Typ „close".[1]

Anschließend wird in Zeile 8 ff. solange iteriert, bis der Heap leer ist. In jedem Schleifendurchlauf wird der Iterator mit geringster Priorität aus dem Heap entfernt, sein aktuelles Event in einer Liste gespeichert und schließlich auf das nächste Element bewegt. Ist der Iterator noch nicht am Ende seiner Folge angelangt, so wird er – mit nun veränderter Priorität – wieder in den Heap eingefügt. Die Schleife terminiert, wenn jeder Iterator seine gesamte Folge besucht hat.

Korrektheit: Die nach Koordinaten aufsteigende Reihenfolge der Events in der Variable *result* folgt sofort aus der Definition der Prioritätsfunktion des Min-Heaps. Da in jedem Durchlauf der while-Schleife genau ein Event verarbeitet wird und jeder Iterator

[1] So wird sichergestellt, dass pro Koordinate immer zuerst alle Öffnungen gesehen werden, bevor ein Intervall geschlossen wird. Die Algorithmen zur Berechnung der Vereinigung und des Schnitts zählen die Anzahl der offenen Intervalle, daher darf es hier zu keiner Ungenauigkeit kommen.

bis zum Ende besucht wird, werden tatsächlich die Events aller Intervall-Sets in korrekter Reihenfolge in *result* gespeichert.

Laufzeit: Es sei $n := \sum_{i=1}^{m} |\mathcal{I}_i|$ die Summe der Anzahl der Intervalle aller Intervall-Sets. In jeder Iteration der for-Schleife in Zeile 4 ff. wird zunächst die Folge der Events für eines der Sets berechnet, um schließlich referenziert durch einen Iterator in den Heap eingefügt zu werden. Über alle m Schleifendurchläufe werden insgesamt $\mathcal{O}(n)$ Intervall-Events in ebenso viel benötigten Operationen berechnet. Das Einfügen in den Heap benötigt hier insgesamt $\mathcal{O}(m \log m)$ Zeit.[2] Die while-Schleife in Zeile 8 ff. wird genau n mal durchlaufen. Jede Iteration benötigt wegen *push* und *pop* $\mathcal{O}(\log m)$ Zeit, sodass die Laufzeit der gesamten Schleife in $\mathcal{O}(n \log m)$ liegt. Mit der (in der Praxis vernünftigen) Annahme, dass $n > m$ gilt, dominiert die while-Schleife und man erreicht eine zusammengenommene asymptotische Laufzeit von $\mathcal{O}(n \log m)$.

Anmerkung: Der Algorithmus kann so implementiert werden, dass er mit nur $\mathcal{O}(m)$ zusätzlichem Speicher für den Heap auskommt. Dies kann erreicht werden, indem man, anstatt die Folgen $events(\mathcal{I}_i)$ explizit zu berechnen, jeden der Grenzpunkte eines Intervalls in den schon vorliegenden Interval-Sets abwechselnd besucht und so implizit dieselbe Folge erhält. Ohne diese Optimierung liegt der asymptotische Speicherverbrauch in $\mathcal{O}(m+n)$.

Auf Grundlage von MergedEvents können nun die Funktionen SetUnion und SetIntersection als Sweep-Line-Algorithmen formuliert werden. Die Vereinigung von m Intervall-Sets $\mathcal{I}_1, ..., \mathcal{I}_m$ ist wiederum ein Intervall-Set, welches jedes Element enthält, das in mindestens einem der Eingabe-Intervall-Sets enthalten ist. Mit anderen Worten wird die Vereinigung alle Bereiche von $\mathbb{N}_0$ abdecken, in denen mindestens ein Intervall eines der Sets „aktiv" ist. Algorithmus 3 berechnet die Vereinigung, indem er die überlappenden Intervalle aller Sets vereinigt. Dazu bedient er sich der Funktion MergedEvents, um alle Intervallgrenzen in aufsteigender Reihenfolge zu beobachten. Ein Intervall der Vereinigung beginnt dort, wo der Zähler *count* den Wert 1 erreicht (Zeile 8), wo also mindestens ein Intervall aktiv ist. Ein Intervall endet mit der Schließung des letzten aktiven Intervalls (Zeile 12). Die Korrektheit des Algorithmus folgt aus den Eigenschaften von MergedEvents und der korrekten Verwaltung des Zählers. Die asymptotische Laufzeit liegt, da pro Event nur eine konstante Anzahl von Operationen durchgeführt wird, wie bei MergedEvents in $\mathcal{O}(n \log m)$.

[2] Das Bauen eines Heaps aus einem unsortierten Array ist in $\mathcal{O}(m)$ Zeit möglich; der Übersichtlichkeit halber wurde dieser Weg nicht gewählt. Die asymptotische Laufzeit des gesamten Algorithmus ändert sich dadurch nicht.

Algorithmus 3 Berechnung der Vereinigung von mehreren Intervall-Sets.

Input:

$[\mathcal{I}_1, ..., \mathcal{I}_m]$ Eine Liste von m Intervall-Sets.

Output:

union Ein Intervall-Set, welches die Vereinigung der m Sets enthält.

```
1: function SetUnion([I_1, ..., I_m])
2:     union := Erzeuge leere Liste.
3:     begin := 0                          ▷ Der Startpunkt des aktuellen Intervalls.
4:     count := 0                          ▷ Die Anzahl der gegenwärtig geöffneten Intervalle.
5:     for all (type, point) in MergedEvents([I_1, ..., I_m]) do
6:         if type = „open" then
7:             count := count + 1
8:             if count = 1 then           ▷ Es beginnt ein neues Intervall der Vereinigung.
9:                 begin := point
10:        else if type = „closed" then
11:            count := count - 1
12:            if count = 0 then           ▷ Das aktuelle Intervall ist beendet.
13:                append(union, [begin, point])
14:    return union
```

Algorithmus 4 Berechnung des Schnitts von mehreren Intervall-Sets.

Input:

$[\mathcal{I}_1, ..., \mathcal{I}_m]$ Eine Liste von m Intervall-Sets.

Output:

intersection Ein Intervall-Set, welches den Schnitt der m Sets enthält.

```
1: function SetIntersection([I_1, ..., I_m])
2:     intersection := Erzeuge leere Liste.
3:     begin := 0                          ▷ Der Startpunkt des aktuellen Intervalls.
4:     count := 0                          ▷ Die Anzahl der gegenwärtig geöffneten Intervalle.
5:     for all (type, point) in MergedEvents([I_1, ..., I_m]) do
6:         if type = „open" then
7:             count := count + 1
8:             if count = m then           ▷ Es beginnt ein neues Intervall der Schnittmenge.
9:                 begin := point
10:        else if type = „closed" then
11:            count := count - 1
12:            if count = m − 1 then       ▷ Das aktuelle Intervall ist beendet.
13:                append(intersection, [begin, point])
14:    return intersection
```

Das Vorgehen von Algorithmus 4 ist nahezu identisch; nur die Verwaltung des Zählers unterscheidet sich von der in Algorithmus 3. Ein Intervall der Schnittmenge beginnt genau zu dem Zeitpunkt, an dem die Anzahl der aktiven Intervalle m erreicht (Zeile 8). Da sich die Intervalle einzelner Intervall-Sets untereinander nicht überlappen, besteht hier die Gewissheit, dass die m aktiven Intervalle zu paarweise verschiedenen Intervall-Sets gehören. Es wird also tatsächlich der Beginn eines Intervalls berechnet, dessen Elemente in allen Eingabe-Sets enthalten sind. Das Intervall endet, wenn die Anzahl der aktiven Intervalle weniger als m wird, also wenn der Zähler *count* den Wert $m - 1$ erreicht (Zeile 12). Für das Funktionieren dieses Algorithmus wird die von MergedEvents garantierte Eigenschaft benötigt, dass bei Intervallöffnungen und -schließungen mit gleicher Koordinate die Öffnungen zuerst beobachtet werden: ansonsten würde der Zähler in manchen Situationen fälschlicherweise niemals den Wert m erreichen. Die asymptotische Laufzeit liegt wie bei Algorithmus 3 in $\mathcal{O}(n \log m)$.

5.2.2. Beschränkung des Speicherplatzes

Bei der Indizierung eines Teilbaumes im IRWI-Baum muss zwischen gewünschter Präzision und dem dafür benötigten Speicherplatz abgewogen werden. Da der Teilbaum eines internen Knotens in Abhängigkeit von dessen Höhe beliebig viele Units enthalten kann, ist auch die Anzahl der verschiedenen Trajektorien innerhalb eines Teilbaumes nicht nach oben beschränkt. Eine exakte Repräsentation aller Identifikatoren dieser Trajektorien ist also aus Speicherplatzgründen im Allgemeinen nicht wünschenswert. Stattdessen führen Issa und Damiani für den IRWI-Baum den Parameter $\lambda \in \mathbb{N}$ ein: keines der sich im Index befindenden Intervall-Sets soll mehr als λ Intervalle enthalten [ID16].

Ziel dieser Sektion ist die Entwicklung des Algorithmus $\textsc{Trim}(\mathcal{I}, \lambda)$, der für ein beliebiges Intervall-Set $\mathcal{I} := I_1, ..., I_n$ ein neues Set $\mathcal{J} := J_1, ..., J_m$ mit $m \leq \lambda$ Intervallen konstruiert. Als weitere Bedingung muss $\mathcal{J}$ jedes in $\mathcal{I}$ enthaltene Element auch weiterhin repräsentieren; es muss also $\mathcal{J} \supseteq \mathcal{I}$ gelten. Die Idee zur Lösung dieses Problems ist die Verschmelzung von benachbarten Intervallen aus $\mathcal{I}$. In $\mathcal{I}$ aufeinanderfolgende Intervalle werden solange miteinander verschmolzen, bis nur noch höchstens λ Intervalle übrig bleiben.

Definition 5. Die *Verschmelzung* von zwei aufeinanderfolgenden, sich nicht überlappen-

den Intervallen $[a_1, b_1], [a_2, b_2]$ ist definiert als

$$merge([a_1, b_1], [a_2, b_2]) := [a_1, b_2]. \tag{5.2}$$

Um die Beschreibung des Algorithmus zu vereinfachen ist die Verschmelzung eines einzelnen Intervalls die Identitätsfunktion: $merge([a, b]) := [a, b]$. Die Verschmelzung von mehr als zwei Intervallen ist rekursiv definiert:

$$\begin{aligned} merge([a_1, b_1], [a_2, b_2], ..., [a_k, b_k]) :=& merge(merge([a_1, b_1], [a_2, b_2]), ..., [a_k, b_k]) \\ =& ... \\ =& [a_1, b_k]. \end{aligned} \tag{5.3}$$

Definition 6. Bei der Verschmelzung von Intervallen entsteht ein Fehler durch nun fälschlicherweise im daraus resultierenden Intervall enthaltene Elemente: für alle natürlichen Zahlen x im offenen Intervall (b_1, a_2) gilt $x \in merge([a_1, b_1], [a_2, b_2])$ aber $x \notin [a_1, b_1] \cup [a_2, b_2]$. Der absolute Fehler ist gegeben durch die Anzahl dieser *false positives*:

$$error([a_1, b_1], [a_2, b_2]) := a_2 - b_1 - 1. \tag{5.4}$$

Er ist nur dann 0, wenn $a_2 = b_1 + 1$ gilt, also wenn die beiden Intervalle „lückenlos" aufeinander folgen. Analog zu Definition 5 sei $error([a, b]) := 0$ und für $k > 2$ Intervalle sei

$$\begin{aligned} error([a_1, b_1], ..., [a_k, b_k]) :=& error([a_1, b_1], [a_2, b_2]) + error([a_2, b_2], ..., [a_k, b_k]) \\ =& ... \\ =& \sum_{i=1}^{k-1} error([a_i, b_i], [a_{i+1}, b_{i+1}]). \end{aligned} \tag{5.5}$$

Die Intervall-Indizes $[1, n]$ sollen nun so in $m \leq \lambda$ nicht-leere Teilintervalle $p_1, ..., p_m$ zerlegt werden, dass sich daraus das „gestutzte" Intervall-Set $\mathcal{J}$ ableiten lässt: es sei

$$\mathcal{J} = J_1, ..., J_m = \underset{i \in p_1}{merge}(I_i), ..., \underset{i \in p_m}{merge}(I_i). \tag{5.6}$$

Jede so gestaltete Partition der Indizes erfüllt die Bedingungen an TRIM: wegen der Nutzung von *merge* gilt offensichtlich $\mathcal{J} \supseteq \mathcal{I}$, denn der repräsentierte Wertebereich wird immer nur ausgedehnt. Der bei der Konstruktion von $\mathcal{J}$ insgesamt anfallende Fehler ist

gegeben durch die Summe der in den einzelnen *merge*-Schritten auftretenden Fehler:

$$error(\mathcal{J}) := \sum_{j=1}^{m} \underset{i \in p_j}{error}(I_i). \tag{5.7}$$

Um die Qualität des „gestutzten" Sets $\mathcal{J}$ möglichst gut zu halten, soll der Algorithmus TRIM für beliebige $\mathcal{I}$ eine optimale Lösung, also eine Lösung mit minimalem *error*, berechnen.

Ein optimaler Algorithmus

Algorithmus 5 Berechnung eines gestutzten Intervall-Sets.

Input:

$\mathcal{I} = [I_1, ..., I_n]$ Ein Intervall-Set, repräsentiert durch eine Liste von Intervallen.

λ Die geforderte Kapazität.

Output:

$\mathcal{J}$ Ein Intervall-Set der Größe $m \leq \lambda$ mit $\mathcal{J} \supseteq \mathcal{I}$.

```
1: function TRIM(I, λ)
2:     if n ≤ λ then
3:         return I
4:     J := Erzeuge leere Liste.
5:     gaps := k-argmax_{1≤i≤n−1}(λ − 1, error(I_i, I_{i+1}))   ▷ Berechne die λ − 1 größten Lücken.
6:     SORT(gaps)                                               ▷ Sortiere Indizes aufsteigend.
7:     i := 1
8:     for all j in gaps do          ▷ Verschmelze die Intervalle zwischen allen Lücken.
9:         append(J, merge(I_i, ..., I_j))
10:        i := j + 1
11:    append(J, merge(I_i, ..., I_n))
12:    return J
```

Innerhalb eines Intervall-Sets $\mathcal{I} = I_1, ..., I_n$ ist die *Lücke* (*gap*) mit dem Index $1 \leq j \leq n-1$ das offene Intervall zwischen I_j und I_{j+1}. Der mit der Lücke j assoziierte Fehler ist $error(I_j, I_{j+1})$. Algorithmus 5 berechnet für Intervall-Sets $\mathcal{I} = I_1, ..., I_n$ mit $n > \lambda$ das geforderte Set $\mathcal{J}$, indem es zunächst die $\lambda - 1$ größten Lücken (gemessen durch *error*) identifiziert. Dazu dient die Funktion k-argmax(k, f), welche für die Elemente einer beliebigen Menge den Ausdruck f auswertet und die k Elemente zurückliefert, für

die der Ausdruck die größten Werte erreicht.[3] Die Funktion kann durch eine einmalige Iteration über alle Elemente der Menge implementiert werden, wobei ein Heap die bis zur aktuellen Iteration gesehenen k Elemente festhält, für die die größten Werte erreicht wurden. Am Ende der Iteration ist die Menge der k Mengenelemente bekannt. Eine solche Implementierung benötigt $\mathcal{O}(n \log k)$ Zeit und $\mathcal{O}(k)$ zusätzlichen Speicherplatz.

Korrektheit: TRIM berechnet nicht explizit die Mengen der zu verschmelzenden Intervalle, sondern genau die Stellen, an denen eine solche Operation *nicht* durchgeführt werden soll. In diesem Fall werden in Zeile 5 diejenigen Indizes in $[1, n-1]$ gefunden, deren Lücken über den größten assoziierten Fehler verfügen. Im Anschluss werden alle benachbarten Intervalle in $\mathcal{I}$ miteinander verschmolzen, um λ neue Intervalle für $\mathcal{J}$ zu erhalten, wobei jede Lücke das Ende einer solchen Operation und den Beginn eines neuen Intervalls markiert (Zeilen 7 bis 11). Das Sortieren in Zeile 6 stellt sicher, dass die Indizes in *gaps* in der korrekten, aufsteigenden Reihenfolge betrachtet werden. Da *gaps* genau $\lambda - 1$ Lücken beinhaltet, erhält man ein neues Intervall-Set aus genau λ Intervallen. Da nur jeweils ein *merge* von aufeinanderfolgenden Intervallen durchgeführt worden ist, gilt auch $\mathcal{J} \supseteq \mathcal{I}$. Falls $n \leq \lambda$ gilt, so muss keine zusätzliche Arbeit verrichtet werden (Zeile 2 f.).

Laufzeit: Ist $n \leq \lambda$, so terminiert der Algorithmus sofort. Ansonsten benötigt er $\mathcal{O}(n \log \lambda)$ Zeit, um die $\lambda - 1$ größten Lücken zu finden. Diese werden in $\mathcal{O}(\lambda \log \lambda)$ Zeit sortiert. Die for-Schleife in Zeile 8 terminiert nach genau $\lambda - 1$ Iterationen, von denen jede nur eine konstante Anzahl von Operationen ausführt. Insgesamt liegt die asymptotische Laufzeit von TRIM damit in

$$\begin{aligned} \mathcal{O}(n \log \lambda + \lambda \log \lambda + \lambda) &= \mathcal{O}((n + \lambda) \log \lambda + \lambda) \\ &= \mathcal{O}(n \log \lambda), \text{ da } \lambda < n. \end{aligned} \tag{5.8}$$

Wegen k-argmin und der Konstruktion von $\mathcal{J}$ aus λ neuen Intervallen wird $\mathcal{O}(\lambda)$ zusätzlicher Speicherplatz benötigt.

Satz 7. Die von TRIM gewählten Lücken sind optimal, d.h. sie resultieren in einem Intervall-Set $\mathcal{J}$ mit minimalem $error(\mathcal{J})$.

Beweis. Durch Widerspruch. Es sei $\mathcal{I} := I_1, ..., I_n$ ein Intervall-Set. Die Kapazität sei

[3] Die so definierte Menge ist nicht unbedingt eindeutig, da der Ausdruck denselben Wert für viele Argumente annehmen kann. Eindeutigkeit ist für das Zeigen der Optimalität aber auch nicht erforderlich; jede solche Menge genügt den Anforderungen.

$\lambda \geq 1$ und es gelte zunächst $n > \lambda$. $\mathcal{J} := J_1, ..., J_m$ sei das Ergebnis des Algorithmus $\textsc{Trim}(\mathcal{I}, \lambda)$. $g_1, ..., g_{\lambda-1}$ sei die in Trim verwendete Folge von Lückenindizes in der Variable *gaps*.

Des Weiteren sei $\mathcal{J}' = J'_1, ..., J'_{m'}$ mit $m' \leq \lambda$ das von einem optimalen Algorithmus $\textsc{TrimOpt}(\mathcal{I}, \lambda)$ berechnete Intervall-Set und $g'_1, ..., g'_{m-1}$ sei die dazugehörige Folge von Lückenindizes.[4]

Annahme: Es gilt $error(\mathcal{J}) > error(\mathcal{J}')$.

Der insgesamt verursachte Fehler ist definiert durch die Summe der Fehler, die bei den einzelnen Verschmelzungen aufgetreten sind. Es wurden *alle* Intervalle mit Ausnahme derer verschmolzen, bei denen eine Lücke gesetzt wurde. Also:

$$\begin{aligned} error(\mathcal{J}) &= \sum_{i=1}^{n-1} error(I_i, I_{i+1}) - \sum_{i=1}^{\lambda-1} error(I_{g_i}, I_{g_{i+1}}), \\ error(\mathcal{J}') &= \sum_{i=1}^{n-1} error(I_i, I_{i+1}) - \sum_{i=1}^{m'-1} error(I_{g'_i}, I_{g'_{i+1}}). \end{aligned} \tag{5.9}$$

Wegen der Ungleichung in der Annahme folgt daher sofort

$$\sum_{i=1}^{\lambda-1} error(I_{g_i}, I_{g_{i+1}}) < \sum_{i=1}^{m'-1} error(I_{g'_i}, I_{g'_{i+1}}). \; \lightning \tag{5.10}$$

Da mit den g_i aber gerade die $\lambda - 1$ größten Lücken gewählt worden sind und da $m' \leq \lambda$ ist, kann die rechte Seite der Ungleichung nicht größer sein; die Annahme ist falsch und es folgt die Behauptung.

Für den Fall, dass $n \leq \lambda$ gilt, ist das Resultat von Trim trivialerweise optimal, da die Eingabe unverändert zurückgegeben wird und somit auch kein Fehler eingeführt werden kann. □

Eine einfache Optimierung von Trim liegt in der *in-place* Berechnung von $\mathcal{J}$. Der schon für $\mathcal{I}$ allozierte Speicherplatz kann wiederverwendet werden, indem die Intervalle in Zeile 8-11 schrittweise überschrieben werden und im Anschluss die Größe der Liste auf λ reduziert wird. Dies würde eine einzelne Allokation von $\mathcal{O}(\lambda)$ Speicherplatz ersparen, aber nicht den asymptotischen Speicherplatzverbrauch reduzieren.

[4] Zu jeder Zerlegung gemäß Gleichung 5.6 existiert offensichtlich eine solche Lückenfolge.

5.3. Konstruktion

Die Konstruktion eines Baumes durch das Einfügen einzelner Units u ist dem normalen R-Tree-Einfügealgorithmus sehr ähnlich. Algorithmus 6 zeigt grob das Einfügen in den IRWI-Baum. Angefangen mit der Wurzel wird in jedem internen Knoten der Teilbaum eines Kindes bestimmt, in dem das Einfügen von u die geringsten Kosten verursacht. Wird ein Blatt erreicht, so wird u dort abgelegt. Falls das Blatt voll ist, so wird ein *node split* durchgeführt und gegebenenfalls bis zur Wurzel fortgesetzt.

Algorithmus 6 Das Einfügen in den IRWI-Baum.

Input:

tree Der IRWI-Baum.
tid Die ID der Trajektorie.
index Der Index der Unit in der Trajektorie.
unit Die Unit $\langle I, label, p, q \rangle$.

```
1:  function IRWI-Insert(tree, tid, index, unit)
2:      node := Die Wurzel des Baums.
3:      level := 1
4:      while level < height(tree) do          ▷ Für jede Ebene interner Knoten.
5:          Finde das Kind c mit dem geringsten Wert für cost(c, mbb(unit), label).
6:          Aktualisiere die MBB von c, sodass sie unit vollständig enthält.
7:          Aktualisiere die Einträge zu c im invertierten Index von node mit unit.
8:          node := Der von c referenzierte Kindknoten.
9:          level := level + 1
10:     if node ist nicht voll then
11:         Füge ⟨tid, index, unit⟩ in das Blatt ein.
12:     else
13:         Führe IRWI-Split(node, tid, index, unit) aus.
```

Das Aktualisieren des invertierten Index in Zeile 7 beinhaltet das Finden der Posting List zu *label*. Existiert diese nicht, weil der Teilbaum von c vorher keine Unit mit diesem Label enthielt, so muss die Liste zuerst angelegt und in den Index eingefügt werden. Dann wird das Posting p zu c in dieser Liste gesucht und wieder gegebenenfalls angelegt. Der Zähler $p.count$ wird um eins inkrementiert und *tid* wird in die Menge $p.ids$ eingefügt. Im Anschluss wird derselbe Prozess für die Liste zu „Total" wiederholt.

Beim Teilen von internen Knoten muss neben der Partitionierung der Einträge auch eine Aufteilung des invertierten Index stattfinden. Bei der Erzeugung neuer Knoten wird

gleichzeitig ein neuer invertierter Index für diese Knoten angelegt. Werden einem Knoten nun Einträge zugewiesen, so muss gleichzeitig der invertierte Index gefüllt werden. Jedes Posting einer jeden Posting List des zu teilenden Knotens muss in den korrekten Index verschoben werden, sodass die Invariante, dass ein interner Knoten den Inhalt seiner Kind-Teilbäume indiziert, aufrechterhalten bleiben. Es gibt mehrere Möglichkeiten, den Algorithmus IRWI-Split umzusetzen. In dieser Arbeit werden die $b+1$ Einträge mithilfe von Guttmans QuadraticSplit [Gut84] auf zwei Knoten verteilt. Als Kostenfunktion sowohl für den Split wie auch für das normale Einfügen dient dabei die von Issa und Damiani definierte hybride Kostenfunktion.

Laufzeit: Es sei $\langle I, p, q, \mathit{label} \rangle$ eine Unit, die in einen bestehenden Baum mit Fan-out b und N Einträgen eingefügt werden soll. Der Parameter λ (siehe Sektion 5.2) gebe die Kapazität der Postings an. Das Traversieren des Baums zum Finden eines Blattes beinhaltet den Besuch von $k \in \mathcal{O}(\log_b N)$ internen Knoten $v_1, ..., v_k$. v_1 ist die Wurzel des IRWI-Baums und v_k ist das Blatt, in das u eingefügt werden soll.

Der Zugriff auf jeden internen Knoten v_i mit $1 \leq i \leq k-1$ benötigt genau eine I/O-Operation. Für die Auswertung der Kostenfunktion in Zeile 5 müssen die MBBs aller Kinder der Knoten und jeweiligen Posting Lists zu den Labels *label* und „Total" vorliegen. Jede dieser Listen muss vollständig geladen werden, damit die Postings jedes der Kinder zur Verfügung stehen. Es sei l_i die Anzahl der im Teilbaum von p_i verschiedenen gespeicherten Labels. Die Anzahl der Posting Lists im invertierten Index von v_i ist folglich $l_i + 1$ (vgl. Abbildung 5.2). Das Auffinden des zu einem Label gehörenden Eintrags im invertierten Index kostet also pro Knoten $\mathcal{O}(\log l_i)$ weitere I/O-Operationen, sofern dieser Teil des invertierten Index durch eine effiziente Suchstruktur, zum Beispiel durch einen B-Baum, umgesetzt ist. Es ist nicht möglich, für l_i eine gute obere Schranke anzugeben: im schlimmsten Fall enthält ein Teilbaum Einträge mit jedem Label aus $\mathcal{L}$, sodass nur von $l_i \leq |\mathcal{L}|$ ausgegangen werden kann. Die Länge einer einzelnen Posting List ist höchstens b, also erfordert das Laden der beiden benötigten Listen das Lesen von $\mathcal{O}(b \cdot \lambda)$ Speicherblöcken. Die Aktualisierung der Strukturen in Zeile 6 f. kostet pro Knoten und Liste jeweils $\mathcal{O}(1)$ Schreiboperationen.

In dem Fall, dass in v_k Platz für ein weiteres Element vorhanden ist (Zeile 11), liegt die Gesamtanzahl der I/O-Operationen also in $\mathcal{O}((\log |\mathcal{L}| + b\lambda) \log_b N)$. Ansonsten wird in Zeile 13 ein Split von v_k und gegebenenfalls von jedem der anderen v_i ausgeführt. Das Teilen eines internen Knotens beinhaltet neben dem Schreiben der neuen Knoten selbst auch das Schreiben ihrer neu zusammengestellten invertierten Indizes aller ihrer Posting

Lists. Pro Knoten muss also allein auf $\Omega(|\mathcal{L}| \cdot b \cdot \lambda)$ Speicherblöcke zugegriffen werden, um nur das Ergebnis des Splits zu persistieren. Die tatsächliche asymptotische Laufzeit ist von dem gewählten Split-Algorithmus abhängig, vgl. dazu die Definition 12 auf Seite 54.

5.3.1. Die hybride Kostenfunktion

Bei der Betrachtung der Laufzeit des Algorithmus IRWI-Insert ist schon aufgefallen, dass sich die Anzahl der verschiedenen Label in einem Teilbaum der Datenstruktur drastisch auf die Leistung auswirken kann. Dasselbe ist für den benötigten Speicherplatz wahr, denn im *worst case* enthält jeder interne Knoten in seinem invertierten Index zu jedem Label eine Posting List der Länge b und hat somit eine tatsächliche Größe von $\mathcal{O}(|\mathcal{L}| \cdot b \cdot \lambda)$ Speicherblöcken. Wird ein IRWI-Baum sowohl nach räumlichen wie auch nach textuellen Kriterien durchsucht, so ist es allerdings von Vorteil, innerhalb der Knoten eine gewisse textuelle Homogenität herzustellen. Enthält ein interner Knoten oder ein Blatt nur eine geringe Anzahl verschiedener Label, so kann er mit höherer Wahrscheinlichkeit bei einer für ihn nicht relevanten Suche übergangen werden. Speichert hingegen jeder Knoten die größtmögliche Anzahl verschiedener Label, dann ergibt sich aus dem invertierten Index kein Performancevorteil: es können nur durch Nicht-Überlappen der MBBs Knoten von der Suche ausgeschlossen werden.

Die Forderung nach Homogenität steht im Widerspruch zur typischen Struktur eines R-Baums, da dort die Einträge nach räumlichen Kriterien möglichst gut partitioniert werden sollen. In dem Versuch, beide Kriterien gleichzeitig zu respektieren, wird von Issa und Damiani [ID16] die hybride Kostenfunktion $cost_{ST}$ eingesetzt, welche sowohl die geometrischen wie auch die textuellen Kosten beinhaltet.

Definition 8. Die Funktion $cost_G(u, c)$ liefert für einen internen Knoten v die *normalisierten geometrischen Kosten*, um die dreidimensionale raum-zeitliche MBB der Unit u in die MBB des Eintrags c einzufügen:

$$cost_G(u, c) := \frac{enlargement(c.mbb, u.mbb)}{\max\limits_{c' \text{ Kind von } v} enlargement(c'.mbb, u.mbb)}. \tag{5.11}$$

$enlargement(c.mbb, u.mbb)$ liefert die absolute Vergrößerung, gemessen am Volumen der Bounding Box, die nötig ist, damit die MBB von c die MBB von u vollständig enthält. $cost_G$ ist also die notwendige Vergrößerung für den Kindeintrag c relativ zur maximal in v auftretenden Vergrößerung.

Definition 9. Die *normalisierten textuellen Kosten* sind aus der relativen Häufigkeit des Labels von u im Teilbaum von c abgeleitet:

$$cost_T(u, c) := 1 - \frac{count(c, u.label)}{total(c)}. \tag{5.12}$$

$count(c, u.label)$ liefert die Anzahl der Units im Teilbaum von c, welche das Label $u.label$ besitzen. $total(c)$ ist die Anzahl aller Units im Teilbaum von c. Beide Zahlen lassen sich direkt aus den zu v gehörenden Posting Lists abrufen. Die textuellen Kosten, um eine Unit in einen Teilbaum einzufügen, sind umso geringer, je häufiger das Label der Unit in diesem Teilbaum schon vorhanden ist.

Definition 10. Es sei $\beta \in (0, 1]$. Die *spatio-textuelle Kostenfunktion* lautet

$$cost_{ST}(u, c) := \beta \cdot cost_G(u, c) + (1 - \beta) \cdot cost_T(u, c). \tag{5.13}$$

Durch den Parameter β lässt sich die Gewichtung von räumlichen und textuellen Kosten beeinflussen. Setzt man $\beta = 1$, so vernachlässigt man die textuellen Kosten und erhält einen gewöhnlichen R-Baum. Eine Veränderung von β hat eine unmittelbare Änderung der Zusammensetzung der Teilbäume zur Folge, siehe dazu die in Kapitel 7 durchgeführten Experimente.

Das Einfügen in den IRWI-Baum erfordert nicht nur die Kostenberechnung für einzelne Blatteinträge, sondern auch für gesamte Teilbäume. Eine solche Berechnung ist bei der Ausführung des QuadraticSplit-Algorithmus nötig, da hier sowohl Blätter wie auch interne Knoten neu partitioniert werden. In einem internen Knoten repräsentiert jeder Eintrag einen ganzen Teilbaum mit einer gegebenenfalls großen Anzahl von Labels. Im Vergleich zum R-Baum, der für einen Split nur die MBBs der jeweiligen Einträge betrachten muss, wird damit die Berechnung der Kosten deutlich komplexer.

Definition 11. Die *generalisierte textuelle Kostenfunction* $cost_{\widehat{T}}(c_1, c_2)$ liefert für zwei Teilbäume c_1 und c_2 die textuellen Kosten, um beide Teilbäume in einem Knoten abzulegen:

$$cost_{\widehat{T}} := 1 - \max_{l \in SL(c_1, c_2)} \frac{count(c_1, l) + count(c_2, l)}{total(c_1) + total(c_2)}. \tag{5.14}$$

$SL(c_1, c_2)$ ist die Menge aller Labels, die sowohl im Teilbaum von c_1 wie auch im Teilbaum von c_2 vertreten sind. Sie kann berechnet werden, indem man auf die invertierten Indizes von c_1 und c_2 zugreift. *count* und *total* haben die gleiche Bedeutung wie in Definition 9.

Definition 12. Die *generalisierte spatio-textuelle Kostenfunktion* für zwei Teilbäume wird analog zu Definition 10 durch ein gewichtetes Mittel von geometrischen und textuellen Kosten erhalten:

$$cost_{\widehat{ST}}(c_1, c_2) := \beta \cdot cost_{\widehat{G}}(c_1, c_2) + (1 - \beta) \cdot cost_{\widehat{T}}(c_1, c_2). \tag{5.15}$$

Dabei ist $cost_{\widehat{G}}$ die beim herkömmlichen QuadaticSplit übliche (aber hier wieder normalisierte) Kostenfunktion für das Gruppieren zweier MBBs im selben Knoten [Gut84].

Zusammengefasst kann also das Einfügen von Blatteinträgen und das Splitten von Knoten unter der Verwendung der hybriden Kostenfunktion stattfinden. Abhängig von der Wahl des Parameters β erwartet man homogenere Teilbäume (bezüglich der Verteilung der Labels) oder eine bessere räumliche Verteilung der Units.

5.4. Suche im Baum

Die Suche im IRWI-Baum kann sowohl anhand von räumlichen wie auch von textuellen Kriterien geschehen. Um dies zu unterstützten, werden bei der Traversierung des Baumes immer sowohl die MBBs der Einträge wie auch deren invertierte Indizes konsultiert. Eine einfache Suchanfrage $\langle I, L, R\rangle$ kann schon durch einen nur um den invertierten Index erweiterten R-Baum-Suchalgorithmus beantwortet werden:

- Traversiere den Baum, beginnend mit der Wurzel.
- Ist der aktuelle Knoten ein interner Knoten, so fahre rekursiv mit allen Kindern des Knotens fort, sofern ihre raum-zeitlichen MBBs das Rechteck $\langle I, R\rangle$ schneiden und ihre Teilbäume mindestens eines der Labels in L enthalten.
- Ist der aktuelle Knoten ein Blatt, so gib alle Einträge $\langle tid, index, unit\rangle$ zurück, für die $unit.label \in L$ gilt und deren raumzeitliches Liniensegment sich mit der durch I und R aufgespannten dreidimensionalen Box schneidet.

Man erhält als Resultat also sowohl die Menge der zutreffenden Trajektorien (durch die verschiedenen Werte von *tid*) wie auch die Abschnitte, an denen sie die Suchanfrage erfüllen (*index* und *unit*).

Bei der Durchführung einer sequentiellen Suchanfrage $q = q_1, ..., q_n$, sollen alle Trajektorien gefunden werden, die die einfachen Suchanfragen $q_1, ..., q_n$ der Reihe nach erfüllen. Der Suchalgorithmus von Issa und Damiani [ID16] wertet alle q_i parallel zueinander aus, während er den Baum durchsucht. Wegen der Nutzung der im invertierten Index abgelegten ID-Mengen können gegebenenfalls frühzeitig gesamte Teilbäume von der Suche ausgeschlossen werden.

Algorithmus 7 Die Suche im IRWI-Baum.

Input:

tree Der IRWI-Baum.

$q = q_1, ..., q_n$ Eine sequentielle Suchanfrage.

Output:

result Die Menge aller Trajektorien, die q erfüllen.

```
1:  function IRWI-Search(tree, q = q_1, ..., q_n)
2:      nodes, entries, timeWindows, ids := Arrays der Größe n.
3:      for i := 1, ..., n do
4:          nodes[i] := {root(tree)}                    ▷ Jede Suche beginnt an der Wurzel.
5:      for level := 1, ..., height(tree) − 1 do        ▷ Für jede Ebene aus internen Knoten.
6:          for i := 1, ..., n do                       ▷ Für alle einfachen Suchanfragen.
7:              entries[i] := MatchingEntries(nodes[i], q_i)
8:              timeWindows[i] := Zeitintervall [t_min, t_max] für entries[i].
9:              ids[i] := Alle Trajektorien-IDs, die in entries[i] enthalten sind.
10:         TrimTimeWindows(timeWindows)
11:         sharedIds := ⋂_{1≤i≤n} ids[i].
12:         for i := 1, ..., n do                       ▷ Schließe Teilbäume von der Suche aus.
13:             FilterByTime(entries[i], timeWindows[i])
14:             FilterByIds(entries[i], sharedIds)
15:             nodes[i] := Lade alle von entries[i] referenzierten Knoten.
16:             if nodes[i] ist leer then
17:                 return {}                           ▷ Es gibt keine passenden Trajektorien.
18:     trunits := Array der Größe n.
19:     for i := 1, ..., n do
20:         trunits[i] := Lade alle Blatteinträge aus nodes[i], die q_i erfüllen.
21:         Gruppiere trunits[i] nach der tid dieser Einträge.
22:     result := CheckOrdering(trunits)
23:     return result
```

Eine genaue Beschreibung des Algorithmus, zusammen mit einem Beispiel, ist bei Issa und

Damiani [ID16] nachzulesen. Es folgt eine kurze Übersicht über die wichtigsten Schritte von IRWI-Search. Um Konsistenz zu wahren, wurde die Benennung der Variablen und Funktionen beibehalten.

nodes[i] speichert vor jeder Iteration der äußersten Schleife die Menge der Knoten, die in dieser Iteration für die Suchanfrage q_i zu durchsuchen sind. Für jede Ebene des Baums und für jedes der q_i werden zuerst die zu durchsuchenden Teilbäume der nächsten Ebene bestimmt (Zeile 6-9). Neben den Kindeinträgen in *entries* wird außerdem jeweils ein alle Einträge abdeckendes Zeitintervall und die gemeinsame Menge aller Trajektorien-IDs berechnet.[5]

Der Aufruf von TrimTimeWindows in Zeile 10 „bereinigt" die vorher berechneten Zeitfenster: da die q_i alle der Reihenfolge nach von einer Trajektorie erfüllt werden müssen, können bestimmte Bereiche dieser Zeitfenster gegebenenfalls ausgeschlossen werden. Es sei $1 \leq i < n$ und $[b_1, e_1]$ bzw. $[b_2, e_2]$ seien die Zeitfenster zu Suchanfragen q_i bzw. q_{i+1}. Ist $b_2 < b_1$, so können die Units im Zeitintervall $[b_2, b_1]$ nicht zum Ergebnis von q_{i+1} gehören, da es für sie nicht möglich ist, dass eine Unit vorher die Anfrage q_i erfüllt hat. Das Zeitintervall von q_{i+1} kann also in diesem Fall auf $[b_1, e_2]$ verkürzt werden. TrimTimeWindows wendet dieses Verfahren und eine dazu analoge Operation für die Endpunkte e_1, e_2 auf alle Endpunkte der Intervalle in *timeWindows* an.

Nach der Berechnung von *sharedIds* werden in Zeile 12 ff. alle Einträge gefiltert, deren Einträge die Suchanfrage q nicht erfüllen können. Ein Teilbaum kann von der Suche zur Anfrage i ausgeschlossen werden, wenn das Zeitintervall seiner raum-zeitlichen MBB nun nicht mehr *timeWindow*[i] überlappt oder wenn er keine der Trajektorien in *sharedIds* enthält. Nach der Terminierung der for-Schleife werden aus allen berechneten Blättern des Baumes für jede der einfachen Suchanfragen die passenden Einträge geladen (Zeile 20). Es werden schließlich die Trajektorien-IDs zurückgegeben, die alle q_i in der vorgegebenen Reihenfolge zufriedenstellen und somit auch die sequentielle Suchanfrage q erfüllen.

Für eine effiziente Implementierung von IRWI-Search ist es sinnvoll, weitere Vorteile aus der parallelen Auswertung der Suchanfragen zu ziehen. Abhängig von $q_1, ..., q_n$ ist es möglich, dass ein einzelner Knoten für viele verschiedene q_i betrachtet wird und somit bei naiver Umsetzung auch mehrfach geladen wird. Im schlimmsten Fall selektiert jedes der q_i alle Units im Baum, sodass jeder Knoten genau n mal gelesen wird. Stattdessen sollten

[5] Tatsächlich handelt es sich um eine als Intervall-Set repräsentierte Obermenge davon, wie in Sektion 5.2.2 beschrieben.

Schritte unternommen werden, um diese I/O-Operationen zu deduplizieren, damit jeder Knoten des Baums höchstens einmal geladen wird.

5.5. Bulk-Loading

Da es sich beim IRWI-Baum um eine Variante des R-Baums handelt, können die in Kapitel 4 erwähnten Techniken zum *bulk-loading* angewendet werden. Neben der Gruppierung der Units in Blätter muss allerdings immer darauf geachtet werden, den invertierten Index von internen Knoten anzulegen.

Hilbert-Packing

Die Menge der Units wird aufsteigend nach ihrem Hilbert-Index sortiert. Für eine Unit $\langle I = [t_1, t_2], \mathit{label}, p = (p_x, p_y), q = (q_x, q_y)\rangle$ wird als Punkt zur Berechnung des Index der Mittelpunkt des dreidimensionalen, raumzeitlichen Liniensegments verwendet, welches zwischen den Punkten (p_x, p_y, t_1) und (q_x, q_y, t_2) verläuft. Es wird also die raumzeitliche Ausdehnung der Units ignoriert. Da diese Abschnitte von Trajektorien sind, wird davon ausgegangen, dass der Abstand zwischen den Punkten im Allgemeinen wegen häufiger Messungen gering ist und dass die Länge der Liniensegmente verschiedener Units zumindest vergleichbar ist.

Der Algorithmus kann leicht so erweitert werden, dass er das *label* einer Unit als eine weitere Dimension betrachtet und in die Berechnung des Index miteinbezieht; man erhielte so den Hilbert-Index eines vierdimensionalen Punkts. Diese Erweiterung wird in dieser Arbeit nicht implementiert, da sie schon vom STR-Algorithmus (siehe unten) unternommen wird. Stattdessen dient das dreidimensionale Hilbert-Packing unter anderem als Benchmark, da es aufgrund des einmaligen Sortiervorgangs eine besonders schnelle Konstruktionsgeschwindigkeit erwarten lässt, wegen der Vernachlässigung textueller Kritieren aber wahrscheinlich schlechte invertierte Indizes erstellt.

Sort-Tile-Recursive

Man erhält eine gute raumzeitliche Verteilung der Units auf Blätter, indem man den STR-Algorithmus bezüglich der Koordinaten x, y und t kacheln und sortieren lässt. Dabei

kann genauso wie beim Hilbert-Packing der Mittelpunkt des Liniensegments einer Unit zu Vergleichen herangezogen werden. Wendet man STR mit den Vergleichsfunktionen $<_{label}$, $<_x, <_y, <_t$ an,[6] so erhält man einen IRWI-Baum, dessen Knoten in den obersten Ebenen zunächst bezüglich des Labels und erst dann bezüglich der raumzeitlichen Dimensionen partitionieren. Mit dieser Reihenfolge lässt sich ein gutes Ergebnis erzielen, wenn der Baum nach nur einem Label von vielen durchsucht werden soll: in den oberen Ebenen des Baums lassen sich schon viele Knoten von der Suche ausschließen, da diese ein anderes Label speichern.

Diese Herangehensweise kann allerdings auch gravierende Nachteile besitzen: Knoten nahe der Wurzel können sich stark überlappen, da räumliche Kriterien bei ihrer Konstruktion vernachlässigt wurden. Wird der Baum nur nach räumlichen Kriterien durchsucht, so kann in der Nähe der Wurzel keinerlei Eliminierung von Einträgen stattfinden; es müssen die Teilbäume zu jedem Label durchsucht werden. Konzeptionell ist dieses Vorgehen dann vergleichbar mit dem Ansatz, für jedes vorliegende Label einen separaten R-Baum zu konstruieren. Das Phänomen kann nur dann vermieden werden, wenn ein starker Zusammenhang zwischen *label* und räumlicher Position der Units vorliegt. Dann würde die Sortierung bezüglich *label* auch implizit eine räumlichen Partitionierung vornehmen.

Quickload

Die Anwendung von Quickload ist im Vergleich zu den anderen Verfahren am einfachsten. Die IRWI-Datenstruktur wird wiederverwendet, um die Units mithilfe eines Baums im Hauptspeicher zu partitionieren, wobei die hybride Kostenfunktion für das Einfügen der Einträge in den temporären Baum verwendet wird. Trotzdem stellt das Design des IRWI-Index in diesem Fall auch eine Herausforderung dar: die besondere Schwierigkeit besteht darin, dass die internen Knoten des Baums – beziehungsweise ihre invertierten Indizes – beliebig groß werden können, sofern ihre Teilbäume viele verschiedene Label enthalten. Obwohl die von Quickload im Hauptspeicher erstellten Bäume nur eine geringe Größe (nicht mehr als C Blätter) besitzen, macht das Einfügen in den Puffer eines Blattes die Aktualisierung aller Vorfahren dieses Blattes nötig, was für den IRWI-Baum auch die Aktualisierung ihrer invertierten Indizes beinhaltet.

[6] Die Label werden als eindeutige Indizes repräsentiert, die Sortierung bezüglich *label* bildet also zu jedem Label einen Cluster mit allen Units, die dieses Label tragen. Die darauffolgende Sortierung bzgl. x, y und t sorgt dann für eine räumliche Organisation innerhalb dieser Cluster.

Es kann also nicht einfach ein IRWI-Baum mit C Blättern erstellt werden, da dieser unter Umständen wesentlich mehr als die verfügbaren $\frac{M}{B}$ Blöcke des Hauptspeichers belegen würde. Stattdessen sollte pro internen Knoten eine feste Anzahl von Blöcken für dessen invertierten Index reserviert werden; wird diese Kapazität bei der Konstruktion des Baums überschritten, so kann der überschüssige Teil des Index in den Sekundärspeicher ausgelagert werden. In Abhängigkeit von dieser Bedingung kann nun ein Wert für C bestimmt werden, sodass der Baum in den vorhandenen Speicherplatz passt.

Die internen Knoten der durch Quickload im Hauptspeicher erstellten Bäume werden nur temporär zur Verteilung der Eingabedaten verwendet; nach der Beendigung eines Schrittes des Algorithmus werden sie zerstört. Das bedeutet insbesondere, dass sie niemals durchsucht werden. Diese Tatsache macht die Erfassung der Trajektorien-IDs in den Posting Lists dieser Knoten unnötig: sie werden ausschließlich vom Algorithmus IRWI-Search zur Beantwortung sequentieller Anfragen betrachtet. Die Posting Lists eines Quickload-Baums besitzen daher nur Einträge der Form $\langle child, count \rangle$ (vgl. Seite 37); durch das Wegfallen der λ Intervalle des Attributs *ids* werden die Posting Lists damit sehr viel kleiner.[7] Dies macht es einfacher, viele Listen gleichzeitig im Hauptspeicher zu behalten, und macht gleichzeitig das Laden der Listen aus dem Sekundärspeicher effizienter. Da die Listen zur Berechnung der Einfügekosten (siehe Algorithmus 6) immer vollständig geladen werden müssen, ist dieser Punkt besonders wichtig.

Quickload wird nicht nur für die Konstruktion der Blattebene, sondern auch für die Konstruktion aller höheren Ebenen verwendet. Für einen R-Baum sind dafür die Unterschiede in der Implementierung nur gering: sowohl einzelne Einträge wie auch ganze Teilbäume können durch eine einzige MBB repräsentiert werden. Für den IRWI-Baum ist die Situation komplexer, da sowohl die MBB eines Teilbaums wie auch dessen textueller Inhalt, also die verschiedenen Label und deren Häufigkeit, berücksichtigt werden sollten. Wie schon oben erwähnt, kann die Anzahl der verschiedenen Label in einem Teilbaum beliebig groß sein. Für die höheren Ebenen werden Teilbäume daher im Quickload-Baum durch einen *Pseudo-Blatteintrag* repräsentiert, welcher neben der MBB des Teilbaums und einem Zeiger auf den Teilbaum selbst auch einen Zeiger auf eine Liste von $\langle label, count \rangle$-Strukturen besitzt. Zur Berechnung der Einfügekosten für einen dieser Pseudo-Blatteinträge wird die generalisierte hybride Kostenfunktion verwendet (Definition 12), da diese schon für Teilbäume konzipiert ist. Nach der Terminierung des Quickload-Aufrufs für eine Ebene

[7] Für die Standardkonfiguration ($\lambda = 40$) der Implementierung beträgt die Größe eines einzelnen Postings dann nur noch 12 Byte (vorher: 332 Byte).

werden dann die so gebauten Blätter zu internen Knoten des IRWI-Baums.[8]

5.5.1. Effiziente Konstruktion interner Knoten

Die Hilbert-Packing und STR basieren beide auf der Sortierung der Eingabedaten, um gute Einträge für die Blätter des Baums zu finden. Nachdem die Blattknoten erstellt wurden, werden nacheinander erstellte Knoten einer Ebene nur noch linear miteinander zu Knoten höherer Ebenen gruppiert. Quickload konstruiert ebenfalls zunächst geeignete Blätter, kombiniert dann aber in höheren Ebenen die Teilbäume der jeweils tieferen Ebene mithilfe der Kostenfunktion des IRWI-Baums. Die Besonderheit von Quickload ist also, dass die Kinder eines Knotens nicht unbedingt direkt nacheinander erstellt wurden. Alle drei Algorithmen haben gemeinsam, dass sie den Baum Ebene für Ebene *bottom-up* konstruieren. Daher stehen die Einträge für jeden internen Knoten des IRWI-Baums schon zu dem Zeitpunkt, an dem er erstellt wird, fest.

Ein interner Knoten speichert die folgenden Informationen über jeden Kindknoten c:

1. Einen Zeiger auf c.
2. Die MBB aller Einträge im Teilbaum von c.
3. Zu jedem Label $l \in \mathcal{L}$, welches im Teilbaum von c enthalten ist, jeweils ein Posting der Form $\langle c, \mathit{count}, \mathit{ids} \rangle$ in der Posting List zu l im invertierten Index.

Bei der *bottom-up*-Konstruktion der Ebenen des Baums müssen diese Informationen, die bei einer großen Anzahl von Labels eine beträchtliche Größe entwickeln können, effizient nach „oben" weitergegeben werden. Vereinfacht ausgedrückt sollen genau die Operationen, die in IRWI-Insert (Seite 50) bei der Aktualisierung eines internen Knotens stattfinden, als gebündelte Operation ausgeführt werden.

Definition 13. Es sei c ein Knoten eines IRWI-Baums. Die Zusammenfassung (oder *summary*) von c ist eine Struktur der Form $\langle \mathit{pointer}, \mathit{mbb}, \mathit{labels} \rangle$. *pointer* ist ein Zeiger auf c und *mbb* ist die MBB aller Units im Teilbaum von c. Zu jedem $\mathit{label} \in \mathcal{L}$, welches in mindestens einer Unit im Teilbaum von c vorkommt, enthält die Liste *labels* einen Eintrag

[8] Eine mögliche Verbesserung der Laufzeit dieses Schrittes ließe sich erreichen, indem man nur die häufigsten k Labels zu einem Pseudo-Blatteintrag speichert. Diese Optimierung wurde hier nicht versucht, da sie möglicherweise die Qualität des resultierenden IRWI-Baumes reduziert und da in allen Fällen die Konstruktion der Blätter deutlich länger dauerte als die Erzeugung der (in ihrer Anzahl viel geringeren) internen Knoten.

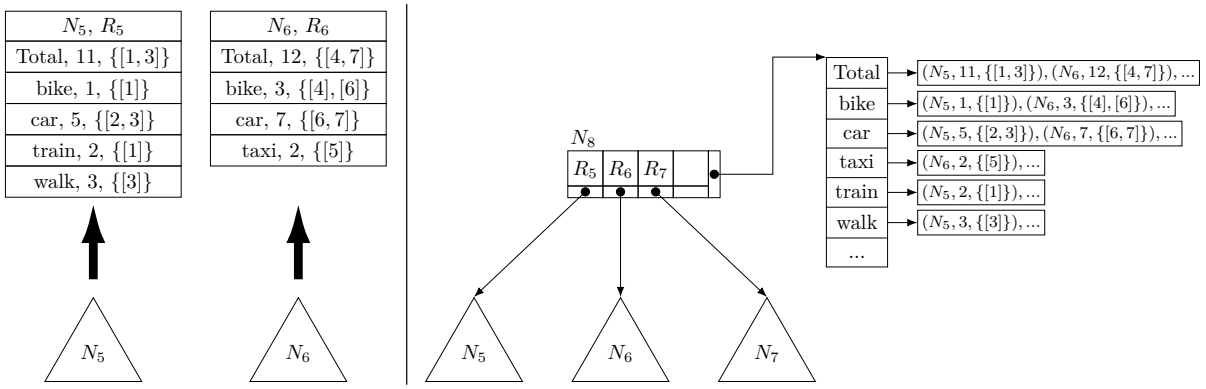


Abbildung 5.4.: Zusammenfassung der Knoten bei der *bottom-up*-Konstruktion.

der Form $\langle label, count, ids \rangle$, wobei *count* die absolute Häufigkeit des Labels ist und *ids* ein Intervall-Set mit höchstens λ Intervallen ist, welches die IDs der Trajektorien enthält, deren Units dieses Label tragen. Die Einträge in *labels* sind nach dem Index des Labels geordnet; als Konvention steht dabei der Eintrag zum virtuellen Label „Total" an erster Stelle.

Abbildung 5.4 zeigt auf der linken Seite die *summary* der Knoten N_5 und N_6. Der Teilbaum von N_5 war schon in Abbildung 5.2 zu sehen; die hier gezeigte Zusammenfassung von N_5 ergibt sich aus jener Grafik (die von N_6 ist fiktiv). Auf der rechten Seite wurde der interne Knoten N_8 angelegt, der unter anderem N_5 und N_6 als Kinder besitzt. Die Einträge aus den Zusammenfassungen der beiden Knoten wurden in den invertierten Index von N_8 kopiert und dabei in die zum jeweiligen Label gehörende Posting List eingefügt. Als Konsequenz aus der Definition der Summaries sind die im invertierten Index zu den Kindern gespeicherten Informationen korrekt.

Berechnung der Zusammenfassung

Ein Blatt kann zusammengefasst werden, indem einfach nur dessen Einträge betrachtet werden. Da die Units im Blatt selbst gespeichert sind, werden dafür keine weiteren I/Os nötig sein. Für jedes Label (einschließlich „Total") wird ein separater Zähler inkrementiert und ein Intervall-Set angelegt. Die MBB der Einträge wird ebenfalls durch eine einmalige Iteration über alle Units berechnet. Die Anzahl der Einträge in einem Blatt ist gering, daher ist mit dieser Operation kein besonderer Aufwand verbunden.

Algorithmus 8 Berechnung der Zusammenfassung eines internen Knotens.

Input:

c Ein Zeiger auf einen internen Knoten des IRWI-Baums.

Output:

Die Zusammenfassung des Knotens.

```
1:  function SUMMARIZE(c)
2:      mbb := MBB der Einträge von c.
3:      labels := Erzeuge leere Liste.
4:      for label im invertierten Index von c do        ▷ Iteriere geordnet nach label.
5:          list := Öffne die Posting List zu label.
6:          count := Σ_{p∈list} p.count
7:          ids := SETUNION({p.ids | p ∈ list})
8:          ids := TRIM(ids, λ)
9:          append(labels, ⟨label, count, ids⟩)
10:     return ⟨c, mbb, labels⟩
```

Algorithmus 8 zeigt die Berechnung der *summary* eines internen Knotens. Es muss *nicht* der gesamte Teilbaum des Knotens besucht werden. Stattdessen genügt es, nur auf den Knoten selbst und dessen invertierten Index zuzugreifen: die MBB des Knotens wird aus den Kindeinträgen berechnet und die Liste *labels* wird bestimmt, indem jede der Listen des invertierten Index zusammengefasst wird. Für jede Posting List werden alle ihre Einträge geladen. In Zeile 6 werden alle Zähler dieser Postings aufsummiert. In den Zeilen 7 und 8 wird die Vereinigung der Mengen $p.ids$ mithilfe von SETUNION (Algorithmus 3) und TRIM (Algorithmus 5) berechnet. Da diese Vereinigung letztendlich wiederum in einem invertierten Index abgelegt werden soll, wird an dieser Stelle auf die Kapazität λ gestutzt. Die Labels müssen in Zeile 4 in sortierter Reihenfolge besucht werden; dies kann implementiert werden, indem für die Speicherung des invertierten Index ein B-Baum benutzt wird, da dieser effiziente *in-order*-Traversierung ermöglicht.

Die in der Zusammenfassung enthaltenen Intervall-Sets sind von besserer Qualität als die, die bei der *one-by-one insertion* in Algorithmus 6 schrittweise erzeugt werden. Der Grund dafür ist, dass nach jeder Aktualisierung in IRWI-INSERT ein Aufruf von TRIM erfolgen muss, wenn ein Set nun $\lambda + 1$ Intervalle enthält. In Algorithmus 8 hingegen gibt es pro Label nur einen einzigen Aufruf von TRIM, der häufig deutlich mehr als nur $\lambda + 1$ Intervalle sieht: es wird die Vereinigung aller Intervall-Sets einer Liste gebildet, wobei jedes dieser Sets bis zu λ Intervalle enthält. Die Fähigkeit von TRIM, ein optimales Ergebnis zurückzugeben, fällt aufgrund der höheren Anzahl der Intervalle also stärker ins Gewicht.

Ein so erstelltes, gestutztes Intervall-Set ist in keinem Fall schlechter als eines, das bei der *one-by-one insertion* erstellt wurde.

Konstruktion des Knotens

Es soll bei der *bottom-up*-Konstruktion des Baums ein neuer Knoten erzeugt werden, dessen Menge an Kindknoten schon bekannt sind. Sei $C := \{c_1, ..., c_n\}$ die Menge der *summaries* dieser Kinder. Ein naiver Algorithmus zur Erzeugung des neuen internen Knotens p lautet wie folgt:

1. Alloziere einen neuen Speicherblock für den Knoten p und erzeuge einen neuen invertierten Index.
2. Für jedes $c \in C$: Füge einen Eintrag $\langle c.pointer, c.mbb \rangle$ zu p hinzu. Iteriere anschließend über die Liste $c.labels$. Für jeden Eintrag $\langle label, count, ids \rangle$ dieser Liste:
   a) Finde im invertieren Index den Zeiger auf die Posting List zu *label* oder erstelle sie, falls noch nicht vorhanden.
   b) Öffne die Posting List und hänge den Eintrag $\langle c.pointer, c.count, c.ids \rangle$ an sie an.

Bei diesem Vorgehen ist es nötig, für jeden einzelnen Label-Eintrag in der Zusammenfassung jedes Kindes eine Suche für das Label auszuführen. Außerdem muss jedes Mal von Neuem auf die richtige Posting List zugegriffen werden. Je nach Implementierung der Listen kann dies mit nicht zu vernachlässigendem Overhead verbunden sein. Liegen diese zum Beispiel als einzelne Dateien vor, so macht jede Ausführung von Schritt 2a) das Öffnen (und Schließen) einer solchen Datei nötig.

In Algorithmus 9 wird ein besser Ansatz gewählt. Es wird die Tatsache ausgenutzt, dass die Einträge in den Zusammenfassungen der Kindknoten schon in sortierter Reihenfolge bezüglich des Labels vorliegen; somit kann genauso wie in Algorithmus 2 ein k-Wege-Merge durchgeführt werden. In Funktion MakeNode werden zunächst wie oben der neue Knoten und dessen invertierter Index angelegt. Im Anschluss wird in Zeile 5 ff. einmal über alle Zusammenfassungen iteriert und jeder Kindknoten durch einen Zeiger und seine MBB in den Knoten eingefügt. Statt schon hier über die Liste *c.labels* zu iterieren, wird ein Iterator für diese Liste zusammen mit einer Referenz auf den Kindknoten in einen Min-Heap eingefügt. Für die Anordnung der Elemente im Heap wird das Label des nächsten Elements eines Iterators verwendet (vgl. Algorithmus 2). Da jede der Listen zumindest

Algorithmus 9 Effiziente Konstruktion eines internen Knotens.

Input:

children Eine Liste von Zusammenfassungen (im Sekundärspeicher) derjenigen Knoten, die Kinder des neuen Knotens werden sollen.

Output:

node Ein Zeiger auf den neuen internen Knoten.

```
1:  function MakeNode(children)
2:      node := Erzeuge einen neuen internen Knoten.
3:      index := Erzeuge den invertierten Index für node.
4:      heap := Erzeuge einen leeren Heap.                    ▷ Siehe Beschreibung.

5:      for all c in children do
6:          Füge Eintrag ⟨c.pointer, c.mbb⟩ zu node hinzu.
7:          iter := Iterator für c.labels.
8:          push(heap, ⟨c.pointer, iter⟩)

9:      currentLabel := null
10:     currentList := null
11:     while ¬empty(heap) do
12:         ⟨pointer, iter⟩ := pop(heap)
13:         ⟨label, count, ids⟩ := getCurrent(iter)
14:         if currentLabel ≠ label then          ▷ Es beginnt ein neuer Label-Cluster.
15:             currentLabel := label
16:             currentList := Erzeuge und öffne eine neue Posting List.
17:             push(index, ⟨label, Zeiger auf currentList⟩)   ▷ Registriere Liste im Index.
18:         Hänge den Eintrag ⟨pointer, count, ids⟩ an currentList an.
19:         moveNext(iter)
20:         if ¬atEnd(iter) then
21:             push(heap, ⟨pointer, iter⟩)
22:         return node
```

einen Eintrag für „Total" besitzt, zeigt der Iterator in Zeile 8 immer auf ein gültiges Element.

Im Anschluss wird in Zeile 11 ff. solange ein Iterator aus dem Heap entnommen, bis dieser leer ist. Das aktuelle Element des Iterators wird in Zeile 13 gelesen; am Ende des Schleifenkörpers wird er inkrementiert und, sofern er noch nicht am Ende seiner Sequenz angelangt ist, wieder in den Heap eingefügt. Die while-Schleife besucht also jeden Eintrag von jeder Zusammenfassung eines Kindes. Da mithilfe des Heaps ein Merge bezüglich des Labels durchgeführt wird, werden alle Einträge, die ein bestimmtes Label besitzen, als ein

zusammenhängender Cluster in der Iteration auftreten. Indem also in Zeile 14 ff. genau dann eine neue *currentList* erzeugt wird, wenn sich das aktuelle Label ändert, wird zu jedem Label genau einmal eine Posting List erzeugt und in den direkt darauffolgenden Iterationen auch in ihrer Gesamtheit durch das Anhängen in Zeile 18 geschrieben.

In Zeile 17 wird die neu erzeugte Posting List zusammen mit ihrem Label im invertierten Index registriert. Hier kann der *bulk-loading*-Algorithmus eines B-Baums (bzw. eines B^+-Baums) leicht angewendet werden: da die Label in sortierter Reihenfolge betrachtet werden, ist zum Einfügen keine Traversierung des Baums notwendig. Stattdessen können die Blätter direkt mit einer Gruppe aufeinanderfolgender Einträge erzeugt werden.

Der Vorteil des Algorithmus ist, dass der invertierte Index nicht durchsucht werden muss und dass jede Posting List nur einmal geöffnet wird, und zu dem Zeitpunkt, an dem eine neue Liste beginnt, schon finalisiert ist. Erkauft wird diese Verbesserung durch eine komplexere Logik für die Iteration über die *summaries*. Da die Anzahl der Labels in einem Teilbaum nicht nach oben beschränkt ist, kann auch die Anzahl der Einträge in einer Zusammenfassung beliebig groß werden; es ist also zu erwarten, dass die Summaries in einer Datei im Sekundärspeicher vorliegen. Um trotzdem eine effiziente Iteration zu gewährleisten, sollte pro Iterator immer der aktuelle Block im Hauptspeicher gehalten werden. Ansonsten würde das häufige Springen zwischen den einzelnen Iteratoren zu *trashing* führen. Da die Anzahl der Kinder und damit die Anzahl der Iteratoren durch den Fan-out des internen Knotens beschränkt ist, wird dafür keine große Speichermenge benötigt.

Laufzeit: Es sei $N := \sum_{i=1}^{n} |children[i].labels|$ die Anzahl der Einträge aller n Zusammenfassungen in *children*. Da die while-Schleife jeden Eintrag jeder *labels*-Liste genau einmal besucht, und da mit den vorhergehenden Argumentationen jede durchgeführte Operation nur eine konstante Anzahl I/Os benötigt, liegt die asymptotische Anzahl der für die Konstruktion des Knotens aufgewendeten I/Os insgesamt in $\mathcal{O}(\frac{N}{B})$.

6. Implementierung

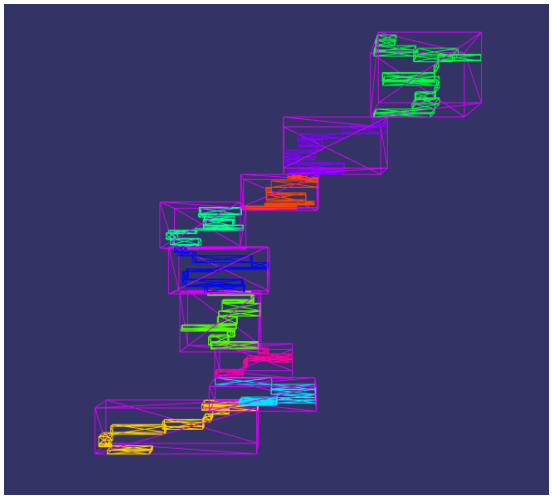
Im Zuge dieser Abschlussarbeit wurde zur Evaluationszwecken eine Implementierung der bisher beschriebenen Verfahren erstellt. Die Software wurde in der Programmiersprache C++ verfasst und besteht aus etwa 14000 Zeilen Quelltext (einschließlich Dokumentation und Tests). Als Grundlage für die Implementierung wird an vielen Stellen das *Templated Portable I/O Environment* (TPIE) [TP12], [APV02] benutzt. TPIE unterstützt die effiziente Umsetzung von I/O-Algorithmen und Datenstrukturen im Sekundärspeicher durch blockorientierte Klassen und Funktionen. Zu den für diese Arbeit wichtigsten Merkmalen der Bibliothek zählen eine Implementierung des externen Merge-Sort-Algorithmus (für Hilbert-Packing und STR), ein generischer externer B-Baum (für den invertierten Index des IRWI-Baums) und einige Hilfsdatenstrukturen wie zum Beispiel externe Stacks, Warteschlangen und Puffer.

Die Software ist in zwei Bereiche unterteilt. Die Kernfunktionalität (alle Algorithmen und Datenstrukturen) befinden sich in der Bibliothek `libgeodb` und ist fast ausschließlich in Headerdateien implementiert.[1] Mehrere Kommandozeilenprogramme benutzen `libgeodb`, um Werkzeuge für die Arbeit mit IRWI-Bäumen und die dazugehörigen Algorithmen zur Verfügung zu stellen. Die wichtigsten Werkzeuge sind:

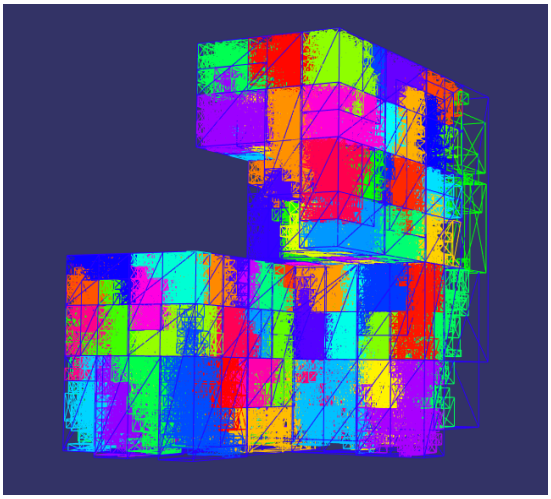
`loader`
: Fügt eine Reihe von Blatteinträgen in einen neuen oder schon existierenden IRWI-Baum ein. Der dazu verwendete Algorithmus ist vom Benutzer konfigurierbar. Es kann zwischen *one-by-one-insertion*, dem Hilbert-Pack-Algorithmus, mehreren STR-Varianten und Quickload gewählt werden. Für die *bulk-loading*-Verfahren kann außerdem der zur Verfügung stehende Speicherplatz spezifiziert werden.

`query`
: Durchsucht einen IRWI-Baum und liefert die Ergebnisse für eine sequentielle Anfra-

[1] Es wurden C++-Templates verwendet, um sowohl eine flexible wie auch eine typsichere Implementierung zu erhalten.



(a) Fan-out 16, OBO.



(b) Fan-out 127, Hilbert-Packing.

Abbildung 6.1.: Die Kinder und Enkel von internen Knoten.

ge als eine Liste von zutreffenden Trajektorien beziehungsweise Units. Die Anfrage wird vom Benutzer als eine Folge dreidimensionaler Bereiche und Mengen von Labelindizes spezifiziert (vgl. Definition der Anfragen in Kapitel 2).

Beide Werkzeuge können geskriptet werden und erfassen als Metriken sowohl die Anzahl der durchgeführten I/Os wie auch die benötigte Zeit und unterstützen so die in Kapitel 7 durchgeführten Experimente.

Beispiel: Anwendung der Kommandozeilenprogramme.

```
# Der Baum "osm" wird mithilfe von Quickload und bis zu 64 MB RAM aus
# den Einträgen in "osm.entries" geladen.
$ loader --entries osm.entries --tree osm --max-memory 64 --algorithm quickload

# Liefert alle Trajektorien (und die zutreffenden Units), die zuerst
# zu einem beliebigen Zeitpunkt durch Münster und danach über
# die A1 (Label 47) verlaufen.
$ query --tree osm
    --rect "51.927, 51.973, 7.579, 7.690, MIN, MAX" --label ""
    --rect "MIN, MAX, MIN, MAX, MIN, MAX" --label "47"
    --results output.txt
```

Ein weiteres Programm, `inspector`, ermöglicht die interaktive Inspizierung eines IRWI-Baums. Zur Programmierung wurden die Grafikbibliothek *OpenSceneGraph* [Osg] und das GUI-Toolkit *Qt* [Qt] verwendet. Zu einzelnen Knoten des Baums können sowohl die räumlichen Komponenten (z.B. die MBBs der Einträge eines Knotens) wie auch die textuellen

Komponenten (z.B. der invertierte Index eines Knotens) dargestellt werden. Abbildung 6.1 zeigt den grafischen Output des Programms für zwei interne Knoten verschiedener Bäume. In der linken Grafik ist der Inhalt eines Knotens der Höhe 2 (Blätter haben die Höhe 1) aus einem Baum, der zu Visualisierungszwecken nur mit maximalem Verzweigungsgrad 16 erstellt wurde. Das rechte Bild enthält einen Knoten der Höhe 3 zu einem Baum, der mithilfe von Hilbert-Packing und dem Datensatz *random-walk* erzeugt wurde (siehe Kapitel 7). Abgebildet sind nicht nur die Einträge des Knotens selbst, sondern auch die Einträge der direkten Kinder. Für den Inhalt verschiedener Kinder wurden verschiedene Farben gewählt; wegen der hohen Anzahl von Kindern kommt es zu Wiederholungen. Die xy-Ebene verläuft horizontal, die t-Achse vertikal.

Die Werkzeuge können durch verschiedene Parameter zur Compile-Zeit konfiguriert werden. Unter anderem können die verwendete Blockgröße (in Byte) und der Fan-out des Baums festgelegt werden. Standardmäßig werden 4 Kilobyte große Blöcke genutzt und der Fan-out wird so berechnet, dass diese maximal gefüllt werden können. Wegen der unterschiedlichen Größe der Einträge in internen Knoten und Blättern kommt es daher in der Regel zu einem verschiedenen maximalen Fan-out für die beiden Knotentypen. Außerdem kann der Wert λ – die Anzahl der Intervalle pro Posting im invertierten Index – verändert werden; als Standardwert wurde wie in [ID16] $\lambda = 40$ gewählt. In Kapitel 7 werden verschiedene Konfigurationen miteinander verglichen. Die Grafiken zum *bulk-loading* in Kapitel 4 wurden erstellt, indem die hier implementierten generischen Algorithmen auf die zweidimensionalen Punkte angewendet wurden.

Implementierung des IRWI-Baums

Die IRWI-Datenstruktur wird in der Bibliothek `libgeodb` durch die Templateklasse `tree` umgesetzt. Zur Compile-Zeit können sowohl der Parameter λ wie auch ein *Storage Backend* bestimmt werden. Dieses gibt an, wie der Baum auf seine Einträge zugreift:

Beispiel: Verschiedene Storage Backends.

```
using storage_type = tree_internal<16>;   // Fan-out == 16
using tree_type = tree<storage_type, 40>; // Lambda == 40

// Ein IRWI-Baum mit beta == 0.5,
// der sich im internen Speicher befindet.
tree_type t(storage_type(), 0.5);

// Oder:
using storage_type = tree_external<4096>; // Blockgröße == 4 KB, Fan-out automatisch.
using tree_type = tree<storage_type, 40>; // Lambda == 40

// Ein IRWI-Baum im externen Speicher,
// der im angegebenen Verzeichnis residiert.
tree_type t(storage_type("path/to/directory"), 0.5);
```

Die Algorithmen des Baums sind unabhängig von der gewählten Speicherart formuliert: sie verwenden deren gemeinsame Schnittstelle, die unter anderem den Zugriff auf den Inhalt einzelner Knoten und auf innere Datenstrukturen wie z.B. den invertierten Index eines Knotens ermöglicht. Die interne Variante des Baums ist sehr einfach gehalten: sie basiert nur auf elementaren Datenstrukturen wie etwa `std::map` zur Implementierung des invertierten Index und `std::vector` für dessen einzelne Posting Lists. Interne Bäume wurden häufig bei der Entwicklung der Algorithmen genutzt, da sie mithilfe eines Debuggers besser als ihre externen Gegenstücke analysiert werden können. Als Vorbild für dieses Design diente TPIEs Implementierung eines B-Baums. Eine modifizierter interner IRWI-Baum kommt bei der Umsetzung des Quickload-Algorithmus zum Einsatz.

Ein Baum im Sekundärspeicher belegt ein Verzeichnis im Dateisystem. Zum Zugriff auf die Posting Lists und auf die Knoten des Baums wird TPIEs *block collection* [APV02] verwendet. Diese implementiert Allokation und *random access* für individuelle Speicherblocks, beinhaltet einen LRU-Cache (*least recently used*) für zuletzt verwendete Blöcke und ist intern durch eine gewöhnliche Unix-Datei umgesetzt. Die Knoten des Baums entsprechen genau einem Speicherblock. Jeder interne Knoten enthält einen Zeiger auf

seinen invertierten Index. Ein invertierter Index ist ein `btree`[2], der den Index jedes im Teilbaum enthaltenen Labels zusammen mit einem Zeiger auf die dazugehörige Posting List abspeichert. Diese Listen sind wiederum als eine Reihe miteinander verketteter Knoten implementiert.

Eine erste Version der Bibliothek verwendete für jede Liste eine gesonderte, lineare Datei. Dieser Ansatz stellte sich als sehr ineffizient heraus, da das System einen nicht zu vernachlässigenden Anteil der Laufzeit damit verbringen musste, diese Dateien immer wieder zu öffnen und zu schließen. An dieser Stelle sei daran erinnert, dass bei der Benutzung von OBO für jede Unit in jedem internen Knoten immer zwei Listen geladen werden müssen (siehe Algorithmus 6). Indem sich alle Posting Lists eine gemeinsame *block collection* teilen, entfällt dieser Overhead vollständig: die Datei wird einmal zu Beginn geöffnet und erst zum Schluss wieder geschlossen. Unglücklicherweise kann dieser Ansatz nicht auch auf die B-Bäume der internen Knoten angewendet werden: TPIEs `btree` benötigt zur Zeit für jede Instanz eine eigene Datei. Der Anteil der B-Bäume an der Gesamtgröße der Datenstruktur ist nur sehr gering: bei den in Kapitel 7 getesteten Fällen lag dieser stets unter 4 %. Trotzdem würde die Leistungsfähigkeit der Software von der Möglichkeit profitieren, alle B-Bäume innerhalb einer gemeinsamen *block collection* ablegen zu können.

Repräsentation der Daten

Alle im Baum enthaltenen Daten werden durch primitive Objekte repräsentiert. Dies gilt sowohl für die Knoten des Baums wie auch für die einzelnen Einträge der „geschachtelten" Datenstrukturen (B-Bäume und Listen). Speicherblöcke, die aus dem Sekundärspeicher geladen wurden, lassen sich ohne Overhead durch das Casten von Zeigern als C++-Objekte reinterpretieren. Nachdem die Manipulation eines Objekts abgeschlossen ist, wird der entsprechende Speicherblock als *dirty* markiert und zu einem vom Cache der *block collection* bestimmten Zeitpunkt wieder in den externen Speicher geschrieben.

In den Blattknoten werden die Units zusammen mit der ID der Trajektorie und ihrem Index innerhalb dieser Trajektorie abgelegt. Abbildung 6.2 zeigt das exakte Layout der Blatteinträge im Speicher. Ein einzelner Blatteintrag besitzt damit eine Größe von 36 Byte; bei der üblichen Seitengröße von 4 KB lässt sich damit auf der Blattebene ein Fan-out von 113 erreichen. Interne Knoten des Baums speichern als Einträge Strukturen der Form $\langle mbb, ptr \rangle$, wobei die MBB durch 2 `vector3`-Instanzen spezifiziert wird. Bei derselben

[2] Aus der TPIE Bibliothek.

```
struct vector3 {
    float x, y;     // Beliebig.
    uint32_t t;     // Unix-Timestamp (Sekunden).
};

struct tree_entry {
    uint32_t trajectory_id; // ID der Trajektorie.
    uint32_t index;         // Index dieser Unit in der Trajektorie.
    vector3 begin;          // Beginn des Liniensegments.
    vector3 end;            // Ende des Liniensegments.
    uint32_t label;         // Index des Labels.
};
```

Abbildung 6.2.: Layout der Blatteinträge.

Blockgröße erreichen interne Knoten einen Fan-out von bis zu 127. Die Implementierung ist in der Lage, innerhalb eines Baums bis zu 2^{32} Trajektorien mit jeweils bis zu 2^{32} Units und genauso vielen verschiedenen Labels zu speichern.

Innerhalb der B-Bäume der invertierten Indizes werden nur kleine Strukturen der Form $\langle label, blockIndex \rangle$ abgelegt; *blockIndex* ist ein 64-Bit-Zeiger auf die Posting List zu *label*. Die Einträge der Listen speichern zu jedem relevanten Kind dessen Index, einen 64-Bit-Zähler für die Anzahl der Units mit dem betreffenden Label in dessen Teilbaum und bis zu λ Intervalle, deren Grenzen jeweils durch einen 4 Byte großen Wert repräsentiert werden. Im Unterschied zur Implementierung von Issa und Damiani [ID16] wurde kein Versuch unternommen, die Intervalle zu komprimieren. Komprimierung der Listen würde sich positiv in der absoluten Laufzeit der Experimente niederschlagen, da weniger I/Os für das Laden und Speichern der Listen aufgewendet werden müssten. Es ist allerdings davon auszugehen, dass alle Algorithmen gleichermaßen davon profitieren würden, sodass die Ergebnisse in Kapitel 7 trotzdem aussagekräftig sind.

Die Datenstruktur speichert die Label nicht als Zeichenketten. Grund dafür ist die sehr große Platzverschwendung, die auftreten würde, wenn man Kopien dieser Zeichenketten an allen relevanten Stellen (Blätter, invertierte Indizes) ablegen würde. Stattdessen sollte jeder der verwendeten Zeichenketten ein eindeutiger, ganzzahliger 32-Bit Index zugewiesen werden. Eine Datenstruktur, die ein solches Mapping effizient herstellt, ist in `libgeodb` mit enthalten.

Bulk-Inserts

Neben dem Einfügen einzelner Einträge und dem Suchalgorithmus unterstützt `tree` das Einfügen ganzer Teilbäume zur Umsetzung von *bulk inserts*. Um in einen bestehenden, nicht-leeren[3] IRWI-Baum t der Höhe h einen Teilbaum t' der Höhe h' einzufügen, werden die folgenden Schritte unternommen:

1. Ist $h' > h$, dann tausche t und t' und h und h' und fahre mit dem nächsten Schritt fort. Es wird in diesem Fall also t in t' eingefügt.
2. Ist $h = h'$, so erzeuge eine neue Wurzel mit den Kindern t und t' und terminiere.
3. Suche ansonsten mithilfe der generalisierten hybriden Kostenfunktion für t' den kostengünstigsten Elternknoten der Höhe $h' + 1$. Füge t' dort ein und führe, falls nötig, Knotenteilungen durch.

Sollen N neue Einträge in einen Baum eingefügt werden, so kann zunächst mithilfe eines der *bulk-loading*-Algorithmen ein neuer Teilbaum (in derselben *block collection*) erzeugt werden und dann in einem Schritt in den bestehenden Baum eingefügt werden, was nur eine einmalige Traversierung des Baumes zum Auffinden eines passenden Elternknotens erfordert. Dieser Ansatz wurde von Chen, Choubey und Rundensteiner [CCR02] unter dem Namen STLT (*small-tree-large-tree*) vorgestellt und wird vom Werkzeug `loader` unterstützt. Da in diesem Fall die Einträge der beiden Bäume vollkommen getrennt behandelt werden, kann der Baum seine Einträge unter Umständen nicht besonders gut partitionieren. Zum Beispiel würde das Einfügen eines Teilbaums der Höhe h in einen existierenden Baum der Höhe h eine Wurzel erzeugen, deren Einträge sich vollständig überlappen, falls die MBBs der beiden Bäume gleich sind. Es kann also eine bessere Strategie sein, zu den N Einträgen mehrere kleine Teilbäume, zum Beispiel nur $\mathcal{O}(\frac{N}{B})$ Blätter, zu erzeugen und diese dann einzeln in den Baum einzufügen. Im Vergleich zum normalen OBO-Einfügen wäre dies immer noch eine Leistungssteigerung, da anstatt einzelner Einträge ganze Blätter im Baum abgelegt würden, was die Anzahl der Einfügeoperationen ungefähr um einen Faktor in $\mathcal{O}(B)$ verringert. Zu dieser Thematik wurden keine Experimente unternommen, da sie den Rahmen dieser Arbeit überstiegen hätten.

[3] Das Einfügen eines Teilbaums in einen leeren Baum ist trivial: die Wurzel des Teilbaums wird zur Wurzel des Baums.

Hilbert-Packing

Um die Sortierung bezüglich der Hilbert-Indizes zu realisieren, werden diese zunächst unter Nutzung des von Hamilton [Ham06] beschriebenen Verfahrens berechnet. Mithilfe von nur einigen binären Rechenoperationen kann zu einem Punkt dessen Index effizient berechnet werden; insbesondere ist die Laufzeit unabhängig von dessen Position und von allen anderen Punkten. Ausschlaggebend für die Laufzeit sind nur die Dimensionalität und die Präzision der gewünschten Hilbert-Kurve. `libgeodb` verwendet für die Berechnung der Hilbert-Indizes der Units (bzw. deren raum-zeitlichen Mittelpunkte) 16 Bit für jede der drei Dimensionen, sodass die möglichen Koordinaten im ganzzahligen Intervall $[0, 2^{16} - 1]$ liegen und die Kurve insgesamt 2^{48} verschiedene Punkte besucht.

Um selbst für unregelmäßig verteilte Daten ein gutes Mapping zwischen Units und Punkten auf der Hilbert-Kurve zu erhalten, wird zunächst eine einzelne Iteration über das gesamte Datenset durchgeführt, wobei für jede der drei Dimensionen der maximale und der minimale Wert festgehalten werden. Im Anschluss werden die Koordinaten der Mittelpunkte der Units normalisiert und neu auf das ganzzahlige Intervall $[0, 2^{16} - 1]$ skaliert: der so erhaltene Punkt liegt auf der Hilbert-Kurve und besitzt somit einen eindeutigen Index. Dieser Index wird zusammen mit der Unit gespeichert. Für die darauffolgende Sortierung der Daten nach ihrem Hilbert-Index wird die externe Merge-Sort-Funktionalität von TPIE verwendet. Danach werden die Einträge zu Blättern und diese wiederum zu internen Knoten gruppiert. Die Konstruktion der internen Knoten findet hier, wie auch bei allen anderen *bulk-loading*-Verfahren, mithilfe einer Implementation von MakeNode (Algorithmus 9) statt. Ein interner Knoten eines so konstruierten Baums ist in Abbildung 6.1 (b) zu sehen.

Die Berechnung der Minima und Maxima kann in der Praxis übersprungen werden, wenn diese schon aus anderer Quelle bekannt sind. Die Hilbert-Indizes müssen nicht unbedingt im Voraus berechnet werden, sondern können jedes Mal aufs Neue beim Vergleich zweier Objekte kalkuliert werden. In der Experimentierumgebung war Letzteres allerdings etwa um den Faktor 4 langsamer, da alle Daten in den Dateicache des Betriebssystems passten.

Für das Füllen der Blätter wird eine Heuristik wie in Sektion 4.2 verwendet: Nachdem die Blätter zu 50 % gefüllt wurden, werden nur solange neue Einträge hinzugenommen, bis die MBB des Blattes mehr als das 1,2-fache des Ausgangsvolumens erreicht.

Sort-Tile-Recursive

Es werden drei verschiedene Varianten des STR-Algorithmus implementiert:

STR-PLAIN
: Die Units werden nach rein raum-zeitlichen Kriterien sortiert und in Kacheln unterteilt (x, y, dann t der Mittelpunkte).

STR-LF (*labels first*)
: Die Einträge werden zunächst nach ihrem Label und dann nach x, y und t sortiert. Man erhält eine gute Partitionierung bezüglich der Labels und eine schlechte Struktur für räumliche Bereichsanfragen.

STR-LL (*labels last*)
: Sortierung nach $x, y, t, label$. Die Struktur ist besser für räumliche Anfragen, aber eine Partitionierung bzgl. der Labels findet fast gar nicht statt. Trotzdem enthält ein Blatt unter guten Umständen nur eine geringe Anzahl verschiedener Labels.

Es wird der externe Merge-Sort-Algorithmus von TPIE in Verbindung mit einer Implementation von Algorithmus 1 verwendet. STR ist der einzige Algorithmus in dieser Arbeit, der die Blätter und die internen Knoten (mit Ausnahme des letzten Knotens eine Ebene) immer zu 100 % füllt. Die mit diesem Algorithmus konstruierten Bäume benötigen daher am wenigsten Speicherplatz. Dies kann die Anzahl der I/Os, die zur Beantwortung einer Anfrage nötig sind, reduzieren.

In den Experimenten wird nur die Variante STR-LF eingesetzt. STR-PLAIN besitzt wegen der rein räumlichen Kriterien dieselben Schwächen, die auch der Hilbert-Algorithms zeigt. STR-LL lieferte im Vergleich zu STR-LF schlechtere Ergebnisse.

Quickload

Wie schon in Sektion 5.5 festgestellt wurde, ist die Anwendung von Quickload auf einen IRWI-Baum im Vergleich zu einem R-Baum deutlich komplexer, da die invertierten Indizes, insbesondere deren Posting Lists, beliebig viel Speicher benötigen können. Um Quickload trotzdem effizient anwenden zu können, wurde ein eigenes, hybrides Speicherbackend implementiert, welches für die temporären Bäume verwendet wird und die folgenden Merkmale aufweist:

- Alle Knoten des Baums werden immer im Hauptspeicher gehalten.
- Teile der Suchstruktur des invertierten Index, die Label auf die Zeiger der dazugehörigen Posting Lists abbildet, werden in den externen Speicher ausgelagert, wenn sie zu groß wird.
- Posting Lists befinden sich im externen Speicher, versehen mit einem großen LRU-Cache: Für jeden internen Knoten werden k weitere Speicherblöcke zwischengespeichert. Da keine Intervall-Sets erfasst werden müssen, sind die Listen sehr kompakt: während sie bei der normalen IRWI-Datenstruktur mehrere Speicherblöcke in Anspruch nehmen, können hier mehrere Listen in einen gemeinsamen Block platziert werden.

Der Parameter, der den Speicherverbrauch von Quickload kontrolliert, ist C, die maximale Anzahl der Blätter eines temporären Baums. Mithilfe einer *worst-case*-Abschätzung wird aus dem konfigurierbaren Parameter M, dem verfügbaren Platz im Hauptspeicher, ein guter Wert C und der dafür im schlimmsten Fall benötigten internen Knoten berechnet, damit insgesamt höchstens M Speicherblöcke belegt werden. Interne Knoten werden dabei schwerer gewichtet, da sie neben dem einen Speicherblock für den Knoten selbst auch die k Blöcke für den Cache des invertierten Index mit sich bringen.

Wegen der Benutzung von TPIEs `btree` war es nicht möglich, diese Bedingung selbst für eine sehr große Anzahl von Labels zu garantieren. Da alle B-Bäume über eine eigene Datei verfügen, konnten sie nicht einfach in das Caching-Framework integriert werden. Dies hätte es ermöglicht, den von ihnen verwendeten Speicherplatz in den Wert k mit einfließen zu lassen. Auch dieser Teil der Software würde also von einer angepassten Implementierung eines B-Baums profitieren. Für die Experimente in dieser Arbeit spielt dieser Punkt kaum eine Rolle, da die B-Bäume nur sehr langsam mit der Anzahl der Labels wachsen (ein Eintrag entspricht 12 Byte); sie erreichen selbst für die größten getesteten Label-Mengen keine signifikante Speichergröße.

Das Speicherbackend musste außerdem um die für Quickload benötigte Operation, die Blätter eines Baums auszulagern, erweitert werden. Dieser Schritt wird unternommen, wenn C Blattknoten erstellt wurden und durch Puffer zum Auffangen weiterer Elemente ersetzt werden. Das direkte Auslagern der Speicherblöcke würde Zugriff auf die Funktionalität des Betriebssystems zur direkten Verwaltung der Speicherseiten erfordern, diese wird von TPIE aus Gründen der Portabilität aber nicht direkt zur Verfügung gestellt.

Daher wird zur Umsetzung der Operation zunächst der Inhalt der Blätter in einen externen Speicherbereich kopiert; danach wird ihr Speicher freigegeben. Die Zeiger auf diese Blätter werden allerdings als eindeutige IDs beibehalten und dienen dazu, ein Mapping zwischen Blatt-Zeiger und Blatt-Puffer herzustellen. Die Blätter werden nicht sofort in den zu konstruierenden Baum eingefügt, da sie gegebenenfalls (bei einem nicht leeren Puffer) in einem rekursiven Schritt weiterverarbeitet werden müssen. Ist der Puffer eines Blatts am Ende eines Schrittes des Algorithmus leer, so wird es, abhängig von dessen Fortschritt, entweder ebenfalls als ein Blatt (erster Aufruf) oder als interner Knoten (wiederholte Aufrufe mit Pseudo-Blatteinträgen) in den fertigen Baum eingefügt.

Die tatsächlichen Werte der Blattzeiger sind nicht deterministisch, da sie nach Belieben bei Speicherallokationen von der Laufzeitumgebung zur Verfügung gestellt werden. Bei der Iteration über alle $\langle \mathit{blatt}, \mathit{puffer} \rangle$-Paare, die nach dem Einfügen aller Einträge in einen temporären Quickload-Baum stattfindet, war es aber wünschenswert, dass diese für denselben Input in derselben Reihenfolge durchgeführt wird. Aus ihr geht sowohl die Reihenfolge der rekursiven Aufrufe von Quickload wie auch die Reihenfolge, in der die fertigen Knoten im externen Speicher konstruiert werden hervor; eine nicht-deterministische Iteration macht also die Ergebnisse nicht genau reproduzierbar. Die Iteration über $\langle \mathit{blatt}, \mathit{puffer} \rangle$-Paare erfolgt daher in der Reihenfolge, in der die *puffer* erzeugt wurden: da das Einfügen in den Baum selbst deterministisch ist, ist auch diese Reihenfolge deterministisch.

Für das Einfügen in den Quickload-Baum konnten alle schon existierenden Algorithmen mit dem modifizierten Speicherbackend wiederverwendet werden. Besteht der Baum aus weniger als C Blättern, so wird der gewöhnliche Einfüge-Algorithmus des IRWI-Baums ausgewählt. Ansonsten wird der Zeiger auf ein Blatt gefunden, an dem das Einfügen stattfinden sollte. Mithilfe des Zeigers wird stattdessen der Puffer des Blattes lokalisiert und der neue Eintrag dort abgelegt. Wegen der Wiederverwendung der Algorithmen kann Quickload mit dem Parameter β der hybriden Kostenfunktion konfiguriert werden. Da im Prinzip nur *one-by-one*-Insertion im Hauptspeicher ausgeführt wird, ist Quickload der einzige getestete Algorithmus, der einen signifikanten Anteil seiner Laufzeit mit Rechenoperationen verbringt.

7. Experimente

Im Folgenden wird die Implementation mit verschiedenen Konfigurationen getestet. Unter der Verwendung aller *bulk-loading*-Algorithmen werden IRWI-Bäume erstellt und ausgewertet. Dabei wird sowohl der Aufwand zur Konstruktion wie auch die Qualität des resultierenden Baums betrachtet. Alle Experimente wurden auf einem Computer mit einen Intel i5-Prozessor mit 3,4 GHz, 16 GB Hauptspeicher und dem Linux Betriebssystem durchgeführt. Als externer Speicher diente eine Festplatte der Marke Seagate mit Kapazität 2 TB und physischer Sektorgröße 4 KB. Wo nicht explizit anders angeben wurde die folgende Konfiguration der Software benutzt:

Tabelle 7.1.: Standardkonfiguration der Software.

Parameter	Wert	Kommentar
Blockgröße	4 KB	Menge der in einer I/O transportierten Daten.
Fan-out	113, 127	Maximale Anzahl der Einträge in Blättern bzw. internen Knoten. Der minimale Fan-out liegt immer bei $\frac{1}{3}$ dieses Werts.
Memory	64 MB	Beim *bulk-loading* verwendeter interner Speicherplatz.
β	0.5	Gewichtungsfaktor der hybriden Kostenfunktion. Nur relevant für OBO und Quickload.
λ	40	Anzahl der Intervalle pro Eintrag einer Posting List. Mit diesem Wert belegt ein einzelner Eintrag 320 Byte.

Zur Messung der durchgeführten I/Os wurde die Funktionalität von TPIE benutzt. Jede mit dieser Bibliothek durchgeführte Lese- oder Schreiboperation ruft intern die Posix-Systemfunktionen `read` oder `write` auf und inkrementiert einen Zähler um die Anzahl der transferierten Bytes. TPIE stellt sicher, dass jeweils nur ein ganzzahliges Vielfaches der

Blockgröße für I/O verwendet wird; man erhält also durch Division durch die Blockgröße eine annehmbar präzise Messung für die Anzahl der tatsächlich durchgeführten I/Os. Zu einigen Experimenten wird außerdem die vergangene Zeit (*wall time*) angegeben; diese wurde mithilfe der C++-Standardbibliothek (`steady_clock`) bestimmt. Im Vergleich zur Kapazität des Hauptspeichers sind die verwendeten Datenmengen häufig klein, sie befinden sich daher mit hoher Wahrscheinlichkeit im Dateicache des Betriebssystems. In der Anzahl der gemessenen I/Os schlägt sich diese Situation nicht wieder, da die Zähler von TPIE immer unabhängig davon inkrementiert werden (der Cache ist ein Implementierungsdetail von `read` bzw. `write`). Bei der Betrachtung der verstrichenen Zeit muss dieser Effekt allerdings berücksichtigt werden.

7.1. Beschreibung der Datensätze

Für die Durchführung der Experimente werden drei verschiedene Datensätze aus verschiedenen Quellen verwendet.

Tabelle 7.2.: Attribute der Datensätze.

Name	#Trajektorien	#Units	#Label
Geolife	10 656	5 400 000	11
OSM	4 653	4 000 000	1 532
Random-Walk*	9 980	10 000 000	100

* In einigen Experimenten werden die Parameter variiert.

Geolife

Das *GeoLife Trajectory Dataset* [Zhe+08], [Zhe+09], [ZXM10] enthält eine große Anzahl personenbezogener Trajektorien. Die Bewegungen von 182 Teilnehmern wurden über einen Zeitraum von etwa 5 Jahren aufgezeichnet und stehen als GPS-Trajektorien zur freien Verfügung. Der Großteil der Trajektorien verläuft in der Umgebung der Stadt Peking, China. Zusätzlich zu den Längen- und Breitengraden der Personen zu einem bestimmten Zeitpunkt steht außerdem in vielen Fällen eine Annotation mit dem aktuellen Transportmittel zur Verfügung, zum Beispiel „bus", „bike" oder „walk". Dieses wird bei der Konstruktion des IRWI-Baums als Label

verwendet. Insgesamt können 10 656 der insgesamt 17 621 Trajektorien des Datensatzes auf diese Art und Weise genutzt werden; diese resultieren insgesamt in einer Menge von 5 400 000 Units mit 11 verschiedenen Labels.

OSM

Dieser Datensatz wurde mithilfe von Rohdaten des *OpenStreetMap*-Projekts [Ope17] generiert. Das Projekt sammelt frei verfügbare Geodaten und stellt sie in öffentlichen Datenbanken zur Verfügung. Mithilfe den so erfassten Straßennetzen lassen sich kartografische Anwendungen oder auch Routenplaner realisieren. Für diese Arbeit wurde der lokale Datensatz für Deutschland mit der Open-Source Routenplanungsengine *OSRM* [LV11], [Osrm] aufbereitet. Zwischen Paaren der 100 größten Städte Deutschlands wurden Routen generiert und als Trajektorien interpretiert. Um GPS-Trajektorien besser zu simulieren, wurden ein maximaler zeitlicher Abstand zwischen zwei Wegpunkten festgelegt. Wird dieser überschritten, so werden auf der Strecke weitere Wegpunkte eingefügt, wobei von gleichmäßiger Bewegungsgeschwindigkeit ausgegangen wird. Die so erhaltenen Trajektorien verfügen mindestens alle 20 Sekunden über einen Positionswert. Units wurden erzeugt, indem die GPS-Koordinaten der Wegpunkte zusammen mit der für die bisherige Wegstrecke benötigten Zeit für die Werte x, y und t verwendet wurden; als Label dient der Name der zu einem Zeitpunkt befahrenen Straße. Wegen der Beschaffenheit des deutschen Straßennetzes treten damit Autobahnen und wichtige Bundesstraßen im Vergleich zu anderen Labels besonders häufig auf.

Random-Walk

Der *Random-Walk* Datensatz ist vollständig künstlicher Natur und wurde mithilfe eines Pseudozufallszahlengenerators erstellt. Innerhalb eines vordefinierten dreidimensionalen Bereichs wird ein zufälliger Ausgangspunkt ausgewählt, von dem aus in kleinen Schritten mit zufälliger Richtung – aber immer mit steigendem Wert für t – weitere Punkte erreicht werden. Mit einer Wahrscheinlichkeit von 20 % ändert sich zwischen Punkten das aktuelle Label und wird zufällig neu aus den 100 möglichen Werten ausgewählt. Nach (im Mittel) 1000 Wegpunkten wird der generierte Weg als Trajektorie ausgegeben. Auf diese Art und Weise wurden solange Trajektorien erzeugt, bis die Anzahl der generierten Units 10 Millionen erreichte. Wegen der zufälligen Generierung sind die Wegpunkte und Labels ungefähr gleichmäßig über die möglichen Wertebereiche verteilt.

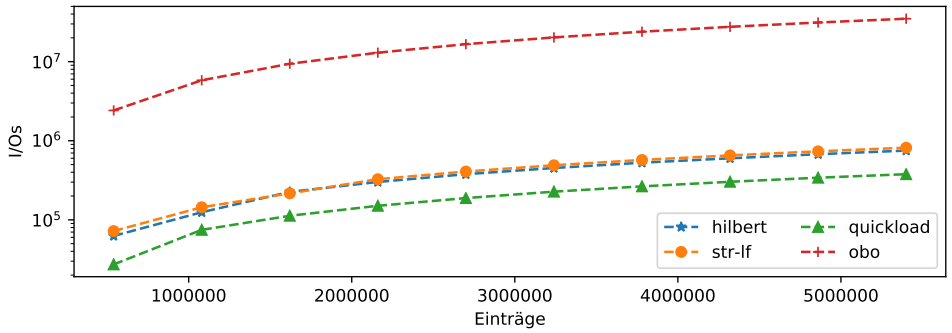


Abbildung 7.1.: Konstruktionskosten für einen IRWI-Baum (Geolife Datensatz, logarithmische Darstellung).

7.2. Konstruktionskosten

Aus der Literatur ist bereits hinlänglich bekannt, dass *one-by-one insertion* (OBO) im Vergleich zu vielen *bulk-loading*-Verfahren sehr schlechte Laufzeit bietet. Im Fall des IRWI-Baums ist die Situation noch schlimmer, da beim Einfügen eines einzelnen Eintrags deutlich mehr Arbeit (sowohl Rechenaufwand wie auch I/O) zu verrichten ist. Abbildung 7.1 zeigt die Konstruktionskosten eines IRWI-Baums für den Geolife-Datensatz für die verschiedenen Algorithmen. Als Maß wird hier die Anzahl der durchgeführten I/Os verwendet. Dabei ist zu beachten, dass aufgrund der besonders schlechten Leistung von OBO in dieser Grafik die y-Achse logarithmisch skaliert ist: OBO ist damit um über zwei Größenordnungen langsamer als die anderen Algorithmen. Für das Einfügen von 5,4 Millionen Einträgen benötigte OBO 236-mal so viele I/Os wie der Algorithmus STR-LF, welcher wiederum mehr I/Os durchführte als die anderen beiden.

Die Ursache für diesen gewaltigen Unterschied ist das häufige Lesen gesamter Posting Lists. Jedes mal, wenn in einem internen Knoten für einen Eintrag die hybride Kostenfunktion ausgewertet wird, müssen zwei Posting Lists vollständig gelesen werden, um für jeden der Einträge des Knotens einen Kostenwert zu bestimmen (siehe Algorithmus 6 und Definition 10). In der hier verwendeten Konfiguration besteht eine einzelne Liste im schlimmen Fall aus 127 Einträgen und ist damit ungefähr 40 KB groß. Es ist also auch nicht überraschend, dass eine weitere Untersuchung der durchgeführten I/Os in diesem Fall ergeben hat, dass 86 % der von OBO durchgeführten I/Os Leseoperationen waren. Dazu kommt noch die schon R-Baum bekannte Problematik der häufig durchgeführten Splits, welche das wiederholte Schreiben von Knoten nötig machen. Für die anderen beiden Datensätze wurden ähnliche Ergebnisse beobachtet, für praktische Anwendungen und

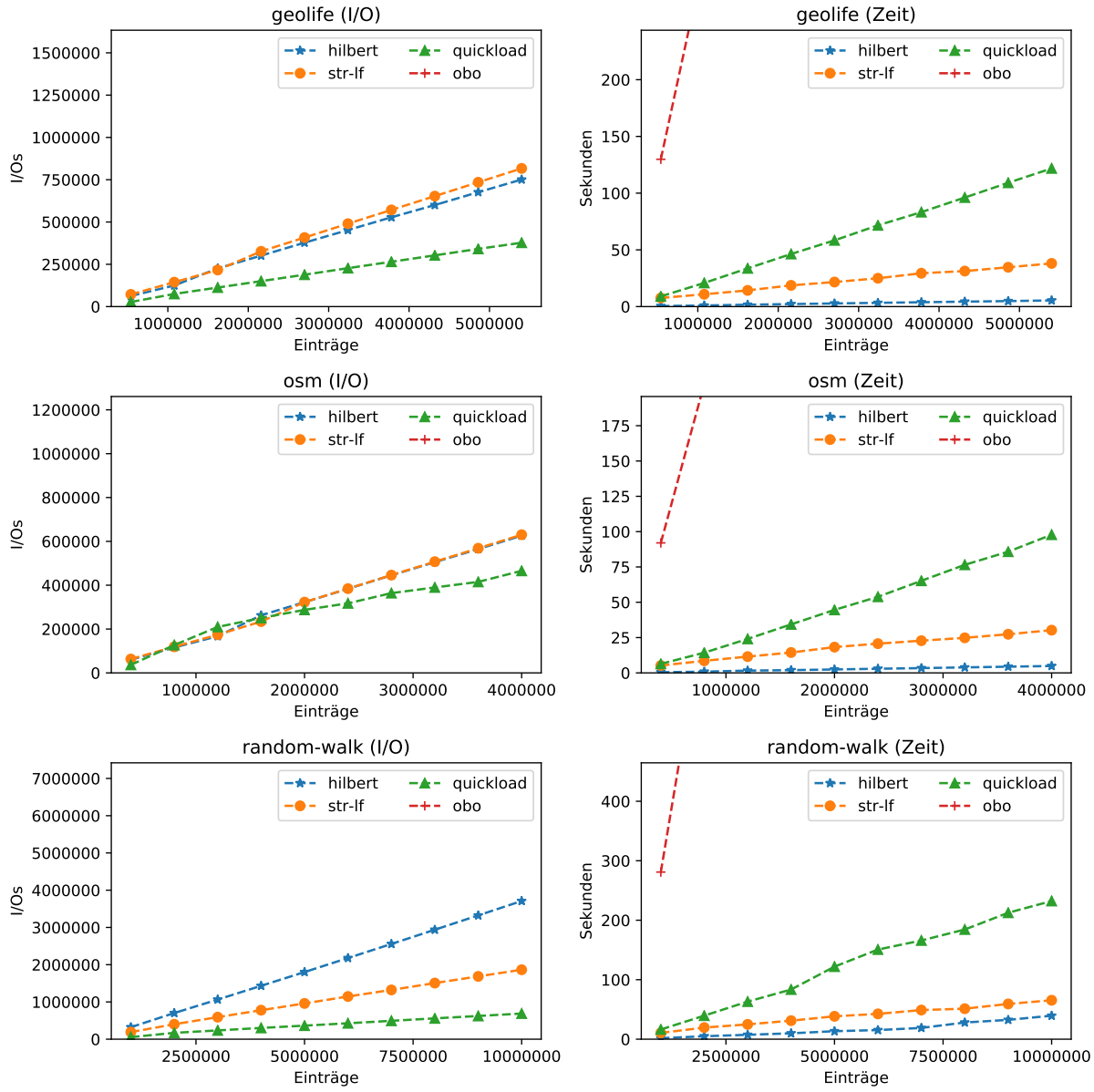


Abbildung 7.2.: Konstruktionskosten für einen IRWI-Baum (alle Datensätze, jeweils I/O und Zeit).

große Datenmengen ist das OBO-Einfügen also kaum verwendbar.

Im Folgenden werden die verschiedenen *bulk-loading*-Algorithmen miteinander verglichen. Die Präsentation der Ergebnisse erfolgt daher nur noch mit linearen Achsen. Abbildung 7.2 zeigt die Konstruktionskosten für IRWI-Bäume für alle Datensätze mit variierender Anzahl der Einträge. Es werden sowohl die durchgeführten I/Os wie auch die tatsächliche Laufzeit in Sekunden angegeben. Es wird die vollständige Konstruktion der Datenstruktur

betrachtet, es werden also neben den Knoten selbst auch die invertierten Indizes angelegt.

Mit Blick auf die benötigte Zeit verursacht das Bauen des Baums mit Hilbert-Packing („hilbert") am wenigsten Kosten. Für alle Datensets terminiert dieser Algorithmus schneller als alle anderen innerhalb von nur weniger Sekunden. Dies ist konsistent mit der Tatsache, dass das einmalige Sortieren des gesamten Datensatzes bei diesem Verfahren aus asymptotischer Sicht der dominierende Schritt ist. Die im Vergleich zum Algorithmus STR-LF ausgeführten I/Os stehen allerdings zunächst im Widerspruch dazu. Diese Variante von Sort-Tile-Recursive kachelt den Datensatz nach vier Kritieren (*label, x, y* und t) und sollte daher auch deutlich mehr I/Os benötigen. Tatsächlich ist die Anzahl der I/Os allerdings für die Datensätze Geolife und OSM nur unwesentlich höher als für Hilbert-Packing, für Random-Walk ist sie sogar deutlich geringer. Die Ursache für dieses Phänomen ist, dass in dieser Implementierung von Hilbert-Packing nur nach raum-zeitlichen Kriterien partitioniert wird. Da der Algorithmus die Label der Units vernachlässigt, hat er keinen Einfluss darauf, wie viele verschiedene Labels sich in dem Teilbaum eines internen Knotens befinden. STR-LF hingegen partitioniert zunächst bezüglich des Labels, sodass die Anzahl der Labels pro internem Knoten gering – in niedrigen Ebenen sogar nur eins – ist. Beim Datensatz Random-Walk fällt dies besonders ins Gewicht: wegen der gleichmäßigen Verteilung der Labels umfasst der invertierte Index jedes internen Knotens ca. 100 Labels. Die Kosten, die für das Erstellen eines internen Knotens anfallen, sind damit für STR-LF tendenziell geringer, was sich auch in der Größe der resultierenden Datenstruktur niederschlägt (siehe Tabelle 7.3).

Für Quickload trifft die umgekehrte Beobachtung zu: es werden in allen Fällen am wenigsten I/Os benötigt, aber es ist im Vergleich deutlich mehr Zeit erforderlich. Da Quickload den gewöhnlichen Einfügealgorithmus des IRWI-Baums im Hauptspeicher anwendet , ist der nötige Rechenaufwand nicht zu vernachlässigen: es wird ständig der Algorithmus zum Splitten eines Knotens ausgeführt, und dessen Laufzeit ist quadratisch. Wegen der Benutzung der hybriden Kostenfunktion besitzt der invertierte Index auch hier im Vergleich zum Hilbert-Packing eine geringere Größe.

In Tabelle 7.3 werden pro Datensatz und verwendetem Algorithmus die Eigenschaften der durch Einfügen aller Elemente konstruierten Bäume aufgelistet. Pro Baum wird bei der Größe der Datenstruktur (in Megabyte) zwischen dem Speicherbedarf für die Knoten des Baums und für den invertierten Index unterschieden. Da alle Bäume jeweils die gleiche Datenmenge speichern, unterscheidet sich der Speicherbedarf für die Knoten kaum.

Die relativ geringen Differenzen haben ihre Ursache in der Fülle der einzelnen Knoten. Nur STR-LF füllt die Knoten zu 100 %; Hilbert verwendet die in auf Seite 73 beschriebene Heuristik und garantiert nur mindestens 50 % volle Knoten. Quickload und OBO garantieren nur, dass kein Knoten weniger als $\frac{1}{3}$ voll ist.

Tabelle 7.3.: Charakteristiken der Bäume pro Datensatz und Algorithmus.

		Speicherplatz (MB)		Kosten	
Datensatz	Algorithmus	Knoten	inv. Index	I/Os (tsd.)	Dauer (s)
Geolife	Hilbert-Packing	270	125	751	5
	STR-LF	198	78	817	38
	Quickload	356	145	378	121
	OBO	363	148	34 933	1 843
OSM	Hilbert-Packing	204	208	626	5
	STR-LF	146	104	630	30
	Quickload	223	156	465	98
	OBO	222	170	59 736	1 434
Random-Walk	Hilbert-Packing	563	2 665	3 710	39
	STR-LF	366	486	1 864	65
	Quickload	550	248	688	232
	OBO	549	257	189 469	5 576

Der für den invertierten Index benötigte Speicherplatz variiert deutlich stärker. Für die Datensätze Geolife und OSM erzeugt STR-LF den kleinsten Index, da dieser zunächst eine Sortierung bezüglich des Labels durchführt und somit die Anzahl der verschiedenen Labels pro Kachel gering halten kann. Dies funktioniert für Random-Walk weniger gut: der Index ist größer als der von Quickload und OBO. Die Anzahl der verschiedenen Labels in OSM ist zwar deutlich höher. Eine Analyse der Bäume liefert im Mittel ähnlich viele Labels pro internem Knoten, aber die durchschnittliche Anzahl der Einträge in den dazugehörigen Posting Lists ist für Random-Walk sehr viel größer (OSM: 9, Random-Walk:

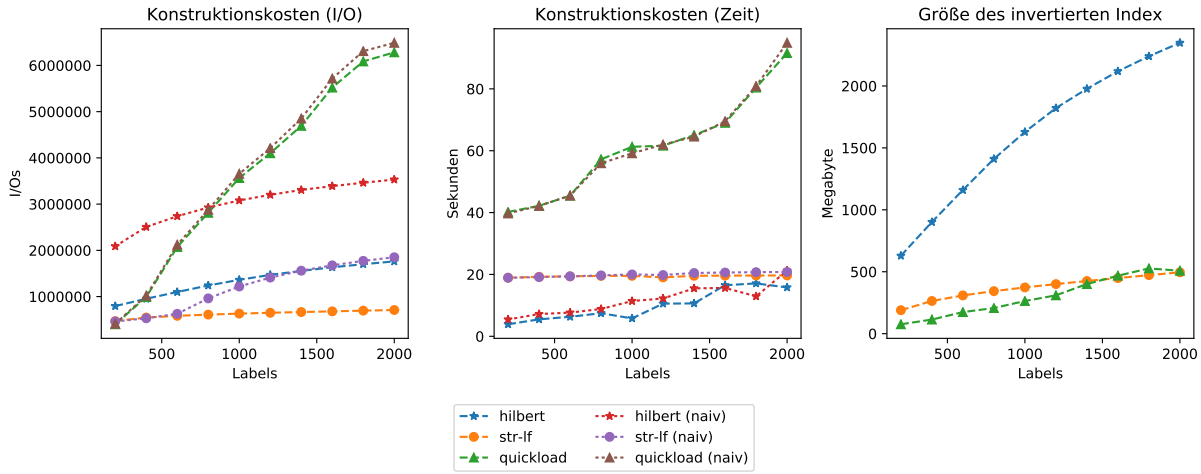


Abbildung 7.3.: Konstruktionskosten bei steigender Anzahl der Labels (Datensatz Random-Walk mit nur 2 000 000 Einträgen).

100).[1] Der Grund dafür ist die gleichmäßige räumliche Verteilung der Labels in Random-Walk. Das Ergebnis für Hilbert-Packing ist im Fall von Random-Walk besonders schlecht: der invertierte Index ist 10 mal größer als der von OBO, da aufgrund der gleichmäßigen Verteilung der Labels jeder interne Knoten jedes Label in seinem Teilbaum besitzt (der Durchschnittswert ist genau 100). Dies demonstriert, dass ein auf rein räumlichen Kriterien arbeitender Pack-Algorithmus nicht für einen allgemeinen Einsatz geeignet ist.

Obwohl Quickload für die Konstruktion eines Baumes die größten Kosten verursacht, ist die Verbesserung im Vergleich zu OBO immer noch enorm. Der Speicherplatzbedarf für Quickload und OBO unterscheidet sich hingegen in allen Fällen nur minimal. Quickload kann daher aus dieser Perspektive als guter Ersatz für OBO angesehen werden, während die anderen beiden Verfahren teilweise deutliche Schwächen aufweisen.

7.3. Konstruktionskosten für viele Labels

Bisher wurde nur die Leistung der Algorithmen bei steigender Anzahl der Einträge betrachtet. Die Anzahl der Labels stellt eine weitere Problemdimension dar, da sie einen direkten Einfluss auf den invertierten Index und somit auf die Konstruktionskosten eines Baums besitzt. Abbildung 7.3 zeigt die Konstruktionskosten eines IRWI-Baums bei

[1] Die Anzahl der Einträge in den Listen zu „Total" wurde nicht mitgezählt, da diese ohnehin immer voll besetzt sind.

gleichbleibender Anzahl der Einträge aber variierender Anzahl der Labels und die Größe des dabei entstehenden invertierten Index in Abhängigkeit vom verwendeten Algorithmus. Es wurde ein modifizierter Random-Walk Datensatz verwendet, da bei den anderen Datensätzen die Anzahl der Labels nicht einfach skaliert werden konnte. Die mit „naiv" gekennzeichneten Algorithmen verwenden den naiven Algorithmus von Seite 63, die anderen nutzen die effizientere Funktion MakeNode (Algorithmus 9).

An der linken Grafik lässt sich ablesen, wie viele I/Os sich durch die Nutzung des effizienteren Algorithmus einsparen lassen. Für Hilbert-Packing und STR-LF ist die Verbesserung immens, im Fall von Quickload ist sie dagegen fast überhaupt nicht vorhanden. Ein Grund dafür ist, dass nur ein kleiner Bruchteil der I/Os von Quickload für die Erzeugung interner Knoten verwendet wird. Bei 2000 Labels fallen für den naiven Algorithmus nur etwa 1 Million der insgesamt 6,5 Millionen I/Os auf diesen Schritt. Diese Zahl kann bei der Verwendung von MakeNode um etwa 200 000 I/Os reduziert werden. Problematisch ist auch, dass Quickload die Knoten höherer Ebenen nicht in linearer Reihenfolge erstellt, wie es bei STR und Hilbert-Packing der Fall ist. Aus diesem Grund müssen Speicherblöcke, die Teile der Zusammenfassungen von Knoten enthalten, möglicherweise nur wegen eines ihrer Einträge geladen werden. Die anderen beiden *bulk-loading*-Verfahren benutzen im Gegensatz dazu immer alle benachbarten Zusammenfassungen der Knoten. Daher, und wegen der ggf. größeren Anzahl von Einträgen in ihren inverierten Indizes, lässt sich deshalb dort die Anzahl der I/Os stärker verbessern.

Da der Algorithmus STR-LF zuerst nach dem Label sortiert und eine gewisse Anzahl von Kacheln erzeugt, kann die Anzahl der verschiedenen Labels in einem Knoten relativ klein gehalten werden; für wenige Label gibt es nur ein Label pro Kachel. Die Anzahl der Kacheln hängt allerdings von der Anzahl der Units in der Eingabemenge ab (siehe Algorithmus 1). Da in diesem Experiment die Anzahl der Units konstant bleibt, die Anzahl der Labels aber steigt, erreicht die Anzahl der verschiedenen Label pro Kachel an einem gewissen Punkt größere Werte als eins. Dies ist die Ursache für das Auseinanderdriften der I/O-Graphen zum Algorithmus STR-LF, das bei ca. 600 Labels beginnt: erst mit einer größeren Anzahl von Labels fällt die Verbesserung durch MakeNode stärker aus.

Die Verringerung der I/O-Kosten schlägt sich nicht in der tatsächlichen Laufzeit nieder (mittlere Grafik). Die vermutliche Ursache dafür ist, dass die zu lesenden Zusammenfassungen bzw. die zu manipulierenden Datenstrukturen sich jeweils mit sehr hoher Wahrscheinlichkeit noch im Cache des Betriebssystems befinden, die zusätzlich ausgeführten

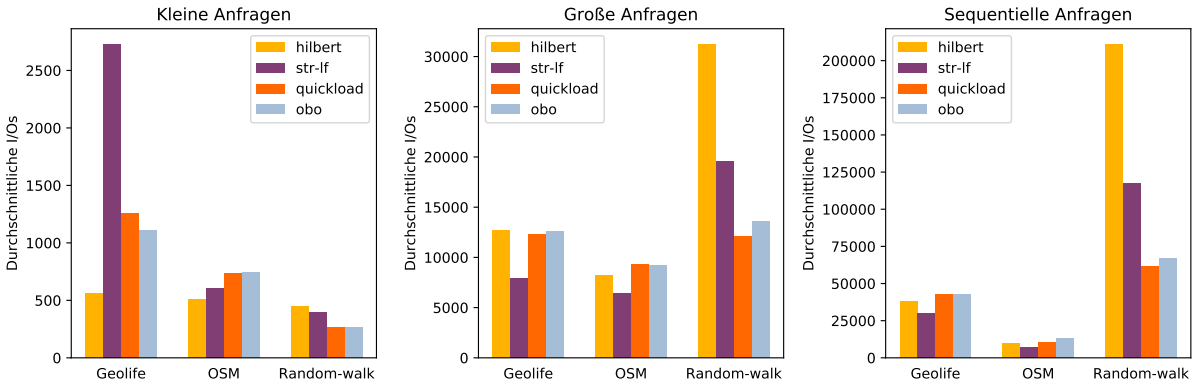


Abbildung 7.4.: Antwortzeiten für einen mit dem jeweiligen Algorithmus konstruierten IRWI-Baum (alle Datensätze).

I/Os sind daher aus praktischer Sicht kostenlos. Allerdings ist zu erwarten, dass dieser Effekt für größere Datensätze weniger stark auftritt.

In der rechten Grafik von Abbildung 7.3 ist die Größe des entstandenen invertierten Index dargestellt. Da beide Verfahren zur Erstellung interner Knoten denselben invertierten Index erzeugen, gibt es hier nur einen Graphen pro *bulk-loading*-Algorithmus. Wie schon vorher bemerkt, resultiert die Anwendung von Hilbert-Packing auf diesen Datensatz in einem besonders großen invertierten Index: er wächst ungefähr linear mit der Anzahl der Labels. Wegen der Sortierung nach dem Label einer Unit kann STR-LF den Index dagegen vergleichsweise klein halten. Quickload erreicht einen ähnlichen Effekt mit der Benutzung der hybriden Kostenfunktion.

7.4. Antwortzeiten

Neben den Konstruktionskosten für einen Baum darf dessen Qualität nicht vernachlässigt werden. Für jeden Algorithmus und jeden Datensatz wurde ein Baum erstellt und im Anschluss mithilfe von drei verschiedenen Anfragetypen bewertet. Für jeden Anfragetypen existiert eine Vielzahl von Anfragen, um mit der durchschnittlichen Anzahl von I/Os, die zur Beantwortung der Suchen notwendig sind, einen fairen Wert ermitteln zu können.

- Kleine Anfragen wählen nur wenige tausend Units. Es wird in einem kleinen Bereich und mit nur wenigen oder *allen* Labels gesucht.[2]

[2] Die Suche nach Units mit beliebigem Label ist eine gewöhnliche Bereichsabfrage im R-Baum und sollte effizient ausgeführt werden können.

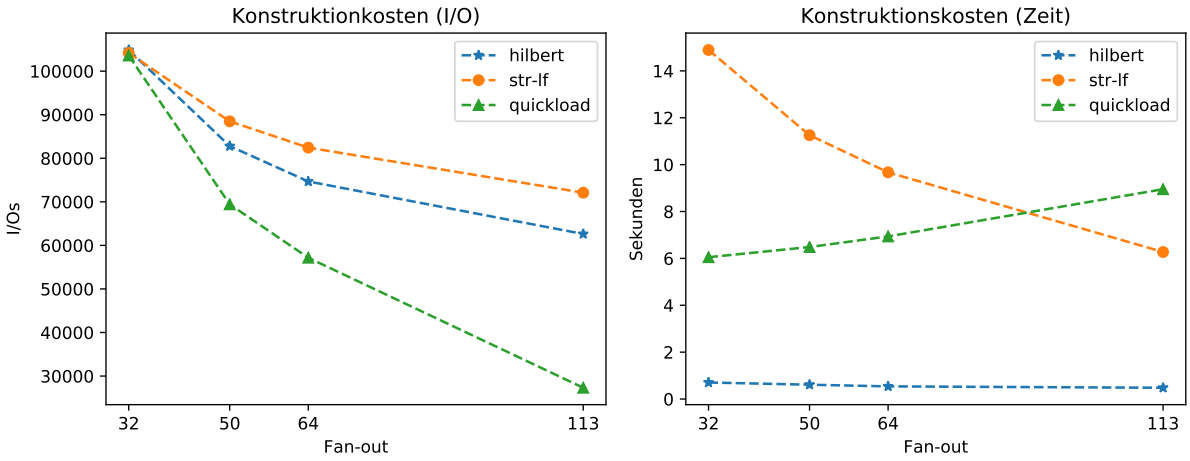
- Große Anfragen selektieren bis zu 20 % der Einträge des Baums mithilfe eines großen Suchbereichs und/oder einer großen Anzahl von Labels.
- Sequentielle Suchanfragen bestehen aus mehreren einfachen Anfragen, um die Leistung des Baums bei der parallelen Auswertung von Anfragen zu bewerten. Es werden die Trajektorien-IDs und Units zurückgegeben, die alle einfachen Anfragen erfüllen.

Die vergangene Zeit wird in diesem Experiment nicht berücksichtigt, da für alle Anfragen dasselbe Verfahren verwendet wird (Algorithmus 7); damit sind die I/O-Kosten allein schon aussagekräftig. Tatsächlich würde die Hinzunahme der Zeit das Ergebnis verfälschen, da sich einige Bereiche des Baums gegebenenfalls noch wegen einer vorhergehenden Anfrage im Cache des Betriebssystems befinden könnten.

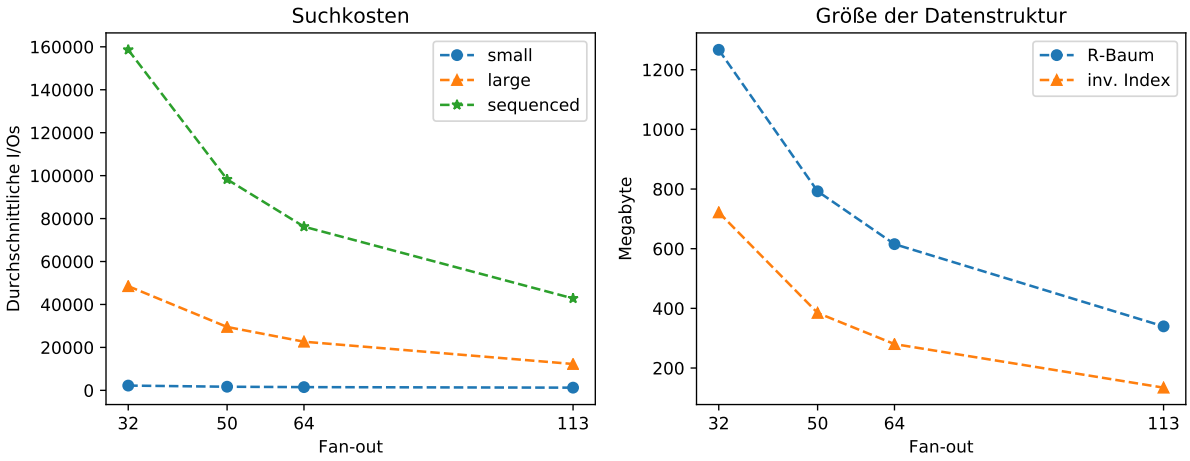
Abbildung 7.4 zeigt die Ergebnisse dieses Experiments. Da alle verwendeten *bulk-loading*-Verfahren im Wesentlichen auf Heuristiken basieren, ist es plausibel, dass es nicht immer einen eindeutigen, besten Algorithmus für jede der Situationen gibt. Trotzdem lassen sich an den Ergebnissen einige wichtige Tatsachen beobachten. Zunächst kann für STR-LF leicht eine Situation entstehen, in der dieser besonders schlecht abschneidet (Geolife, kleine Anfragen). Dies geschieht, wenn nach einem kleinen räumlichen Bereich und mehreren Labels gesucht wird. Da der Datensatz zuerst bezüglich des Labels gekachelt wurde, muss der Baum, wenn nach allen Labels in einem kleinen Bereich gesucht wird, jedes Kind der Wurzel durchsuchen. Weniger ungünstig, aber trotzdem ineffizient, ist die Suche nach mehreren Labels: auch hier muss jedes der betreffenden Kinder durchsucht werden; das Suchverhalten ähnelt damit der Suche in mehreren alleinstehenden R-Bäumen (ein R-Baum pro Label).

Ebenfalls auffallend schlecht ist die Leistung des Hilbert-Algorithmus bei sequentiellen Anfragen und dem Random-Walk Datensatz. Da die Labels in Random-Walk gleichmäßig verteilt sind, kann fast kein Teilbaum aufgrund des invertierten Index bei der Suche ausgeschlossen werden: jeder Teilbaum enthält jedes Label. Es können also nur die MBBs der Knoten bei der Suche verwendet werden, sodass die Leistung im Vergleich zu den anderen Verfahren deutlich schlechter ist.

Für den Quickload-Algorithmus lässt sich die gleiche Beobachtung wie beim Speicherplatzverbrauch (Tabelle 7.3) machen: es existiert fast kein Unterschied zu dem mit OBO erzeugten Baum. Für jeden Datensatz und Anfragetypen ist die Differenz minimal, damit ist Quickload ein guter Ersatz für die sehr langsame Konstruktion durch OBO.



(a) Konstruktionskosten für die verschiedenen Algorithmen.



(b) Eigenschaften der Bäume (Quickload).

Abbildung 7.5.: Konstruktion und Suche in einem Baum bei variierendem Fan-out (Datensatz Geolife, nur die ersten 10 %).

7.5. Auswirkung des Fan-outs

Für dieses Experiment wurde mithilfe des Datensatzes Geolife für die verschiedenen Werte 32, 50, 64, 113 des Fan-outs bei gleichbleibender Blockgröße 4 KB ein IRWI-Baum konstruiert. Aufgrund der für geringe Fan-outs erheblichen Größe der konstruierten Bäume wurden nur die ersten 10 % der Einträge des Datensatzes verwendet. Die Werte 32, 50 und 64 geben sowohl den maximalen Fan-out von internen Knoten wie auch von Blättern an. Der Wert 113 ist der mit dieser Implementierung größte mögliche Wert für den Fan-out eines Blatts. Bei diesem Datenpunkt werden auch die internen Knoten vollständig ausgefüllt (mit Fan-out 127). Der minimale Fan-out eines Knotens ist in allen Fällen $\frac{1}{3}$

des maximalen Fan-outs.

Abhängig vom verwendeten Fan-out werden die Speicherblöcke besser oder schlechter ausgenutzt. Eine geringere Kapazität der Knoten erhöht die Anzahl der zur Speicherung desselben Datensatzes nötigen Knoten und erhöht damit sowohl die Anzahl der nötigen I/Os wie auch die Höhe des fertigen Baums. Andererseits erhöht sich mit der Anzahl der Einträge eines Knotens auch der notwendige Rechenaufwand, um eine Suche oder, im Fall von OBO, eine Einfügung vorzunehmen. In diesem Experiment werden nur die verschiedenen *bulk-loading*-Verfahren betrachtet, trotzdem ist diese Überlegung relevant für den Quickload-Algorithmus, da dieser OBO im Hauptspeicher ausführt.

In Abbildung 7.5 sind die Ergebnisse des Experiments zu sehen. In (a) werden die Konstruktionskosten pro Fan-out und Algorithmus sowohl als I/O-Kosten wie auch als benötigte Zeit angegeben. In allen Fällen resultiert ein höherer Fan-out erwartungsgemäß in einer geringeren Anzahl durchgeführter I/Os, wobei diese Entwicklung für den Quickload-Algorithmus besonders stark ausfällt. Wegen der vergleichsweise geringen Größe des Datensatzes terminiert Quickload für höhere Fan-outs schon nach nur wenigen Schritten und ist daher besonders I/O effizient. Beim Hilbert-Packing und bei STR-LF ist wegen sinkender I/O Anzahl auch die benötigte Zeit geringer; beide Algorithmen sind nicht rechenintensiv. Mit steigendem Fan-out erhöht sich allerdings die Dauer des Quickload-Algorithmus: der Grund dafür ist die Verwendung des quadratischen Knotenteilungsalgorithmus und die mit dem Fan-out wachsende Länge der Posting Lists. Wie schon in Abbildung 7.2 zu sehen war, benötigt Quickload deutlich mehr Rechenzeit als die anderen Algorithmen, sodass sich der steigende Fan-out auch hier in der Laufzeit bemerkbar macht.

Abbildung 7.5 (b) zeigt die Eigenschaften der mit Quickload erstellten IRWI-Bäume. Auf der linken Seite wurde jeder Baum mit den schon in Sektion 7.4 genutzten Anfragen durchsucht; die dafür im Mittel nötigen I/Os sind in der Grafik eingezeichnet. In allen Fällen bewirkt eine Steigerung des Fan-outs geringe Suchkosten. In der rechten Grafik wird die Größe der resultierenden Datenstruktur dargestellt. Wie erwartet bewirkt eine Verringerung des Fan-outs einen gesteigerten Speicherplatzbedarf. Dabei ist zu beachten, dass ein Baum mit geringerem Fan-out wegen seiner größeren Anzahl interner Knoten auch einen größeren invertierten Index besitzt. Insgesamt wird also, trotz der höheren Konstruktionszeit im Falle Quickloads, mit vollständiger Ausnutzung der Speicherblöcke eine besseres Ergebnis erzielt.

7.6. Skalierbarkeit von Quickload

Bei gleichbleibendem Datensatz sollte die Laufzeit von Quickload sinken, sofern man den verfügbaren Hauptspeicher M vergrößert, da die erwartete Anzahl der I/Os in $\mathcal{O}(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ liegt [BS01]. Dies ließ sich in der Praxis mit der vorliegenden Implementierung allerdings nicht beobachten. Mit den Datensätzen Geolife, OSM und einer besonders großen Variante von Random-Walk (200 000 000 Einträge) wurde mit Quickload und variierender Hauptspeicherkapazität ein IRWI-Baum erstellt. Die nachfolgende Tabelle enthält die Ergebnisse dieser Versuche:

Tabelle 7.4.: Laufzeit des Quickload-Algorithmus.

			Konstruktionskosten	
Datensatz	#Units	Memory (MB)	I/Os (tsd.)	Dauer (s)
Geolife	5 400 000	16	379	158
		256	379	157
OSM	4 000 000	16	665	114
		256	235	110
Random-Walk	200 000 000	64	13 735	6 293
		128	13 748	6 436
		256	13 783	6 251

Es konnten keine Hauptspeichergrößen über 256 MB untersucht werden, da die Implementierung zur Zeit für den Puffer jedes Blattes eine neue Datei öffnet und das Limit des Betriebssystems bezüglich der gleichzeitig geöffneten Dateien so schnell erreicht wird. Ein anderer Ansatz wäre erforderlich, um diese Kapazität weiter zu erhöhen (z.B. die Implementierung als eine Reihe verlinkter Blocklisten in einer gemeinsamen Datei, wie es schon für die Posting Lists geschehen ist).

Weder für Geolife noch für Random-Walk lässt sich eine nennenswerte Verbesserung bei einem vergrößerten Hauptspeicher feststellen. Sowohl die Anzahl der I/Os wie auch die benötigte Zeit sind bis auf minimale Abweichungen identisch. Nur für OSM ließ sich die Anzahl der I/Os reduzieren; dies liegt aber nur daran, dass hier der Datensatz vollständig

in den Hauptspeicher passte und Quickload daher keine rekursiven Schritte ausführen musste. Die Problematik lässt sich beim Geolife-Datensatz gut beobachten: bei 16 MB Speicherkapazität verwendete Quickload 58,5 Sekunden für den Aufbau des ersten temporären Baums für die Blattebene und weitere 95,5 Sekunden für Abarbeitung der rekursiven Schritte. Die Vergrößerung auf 256 MB hatte zur Folge, dass 117,5 Sekunden für den Aufbau des nun größeren ersten Baums benötigt wurden; in den rekursiven Schritten wurden hier nur noch 33,5 Sekunden verbracht.[3] Die bei den rekursiven Aufrufen wegfallende Ausführungszeit wurde also durch einen teureren ersten Schritt wieder ausgeglichen.

7.7. Gewichtung der Kosten

In der hybriden Kostenfunktion dient $\beta \in (0, 1]$ der Gewichtung der räumlichen und textuellen Kosten. Um die Auswirkung dieses Parameters zu untersuchen, wurden zwei Experimente durchgeführt. Zum einen wurden IRWI-Bäume mit verschiedenen, festen Werten für β und den Algorithmen OSM und Quickload erstellt. $\beta = 0.5$ ist der in den anderen Experimenten verwendete Standardwert; mit $\beta = 1.0$ erhält man die Kostenfunktion eines normalen R-Baums. Ein größerer Wert für β lässt eine bessere räumliche Verteilung der Daten erwarten, während ein niedriger Wert in einer besseren Trennung der Labels resultieren sollte.

Um die räumliche Verteilung der Daten zu bewerten, wird der Overlap der MBBs in einem internen Knoten betrachtet. Im besten Fall überlappen sich die MBBs der Einträge des Knotens überhaupt nicht: ihr aufsummiertes Volumen V_{sum} ist gleich dem Volumen V_{union}, welches man erhält, indem man das Volumen der Vereinigung aller MBBs berechnet. In jedem Fall gilt $V_{\text{union}} \leq V_{\text{sum}}$. Es gibt also immer einen Faktor $d \in [0, 1]$ mit

$$d = \frac{V_{\text{union}}}{V_{\text{sum}}},$$

wobei ein größerer Wert für d auf weniger Overlap hindeutet. Für jeden internen Knoten wurde der Wert für d bestimmt, dabei wurde V_{union} mithilfe von Bentleys Algorithmus [Ben77] berechnet. Die folgende Tabelle enthält für jeden der mit Quickload erzeugten Bäume und für jede Nicht-Blattebene des Baums den Durchschnittswert von d. Des Weiteren wurde für jeden internen Knoten die Anzahl der verschiedenen Labels in seinem

[3] Die Ausführungszeit für die höheren Ebenen des Baums ist in beiden Fällen sehr gering und wird hier daher nicht betrachtet.

invertierten Index festgehalten, um ein Maß für die Verteilung der Labels auf die Knoten zu erhalten.

Tabelle 7.5.: Eigenschaften der Bäume in Abhängigkeit von β.

		d^*			#Labels*		
Datensatz	β	1.	2.	3.	1.	2.	3.
Geolife	0.25	0.54	0.89	0.97	11.0	2.67	1.02
	0.50	0.61	0.87	0.98	11.0	4.07	1.04
	0.75	0.73	0.90	0.98	11.0	4.07	1.05
	1.00	0.96	0.91	0.97	11.0	8.63	4.00
OSM	0.25	0.19	0.32	0.93	1 532.0	365.13	6.29
	0.50	0.22	0.32	0.93	1 532.0	379.88	6.60
	0.75	0.32	0.28	0.92	1 532.0	411.43	7.67
	1.00	0.48	0.46	0.90	1 532.0	424.00	20.99
Random-Walk	0.25	0.07	0.06	0.77	100.0	48.50	1.58
	0.50	0.08	0.06	0.77	100.0	70.87	2.22
	0.75	0.09	0.05	0.75	100.0	92.64	4.30
	1.00	0.37	0.45	0.78	100.0	100.0	99.99

* Durchschnittswert für jede Ebene, die aus internen Knoten besteht. Ebene 1 ist die Wurzel; alle Bäume haben die Höhe 4. Wegen des hohen dafür nötigen Rechenaufwands wurde die Blattebene nicht untersucht.

Mit steigendem β verringert sich in allen Fällen der Overlap in der Wurzel besonders stark. Für tiefere Ebenen lässt sich an einigen Stellen ebenfalls eine Verbesserung feststellen, wobei die verschiedenen Werte für d allerdings deutlich geringere Unterschiede aufweisen. Gleichzeitig erhöht sich aber auch die Anzahl der verschiedenen Label pro internen Knoten in den Ebenen 2 und 3. Die Anzahl in Ebene 1 verändert sich nicht, da spätestens in der Wurzel jedes Label im invertierten Index repräsentiert werden muss.

In einem zweiten Experiment wurden zwei neue, dynamische Strategien für die Bestimmung von β versucht. Es seien n ein beliebiger Knoten der Höhe h und $\hat{h} := \max(1, h-1)$. Der konkrete Wert für β wird nun von der Höhe des Knotens abhängig gemacht:

increasing

Es ist $\beta = 0.5^{1/\hat{h}}$, d.h. in höheren Ebenen des Baums werden die räumlichen Kosten zunehmend priorisiert.

decreasing

Es ist $\beta = 0.5^{\hat{h}}$, d.h. die textuellen Kosten werden bei steigender Höhe stärker gewichtet.

normal

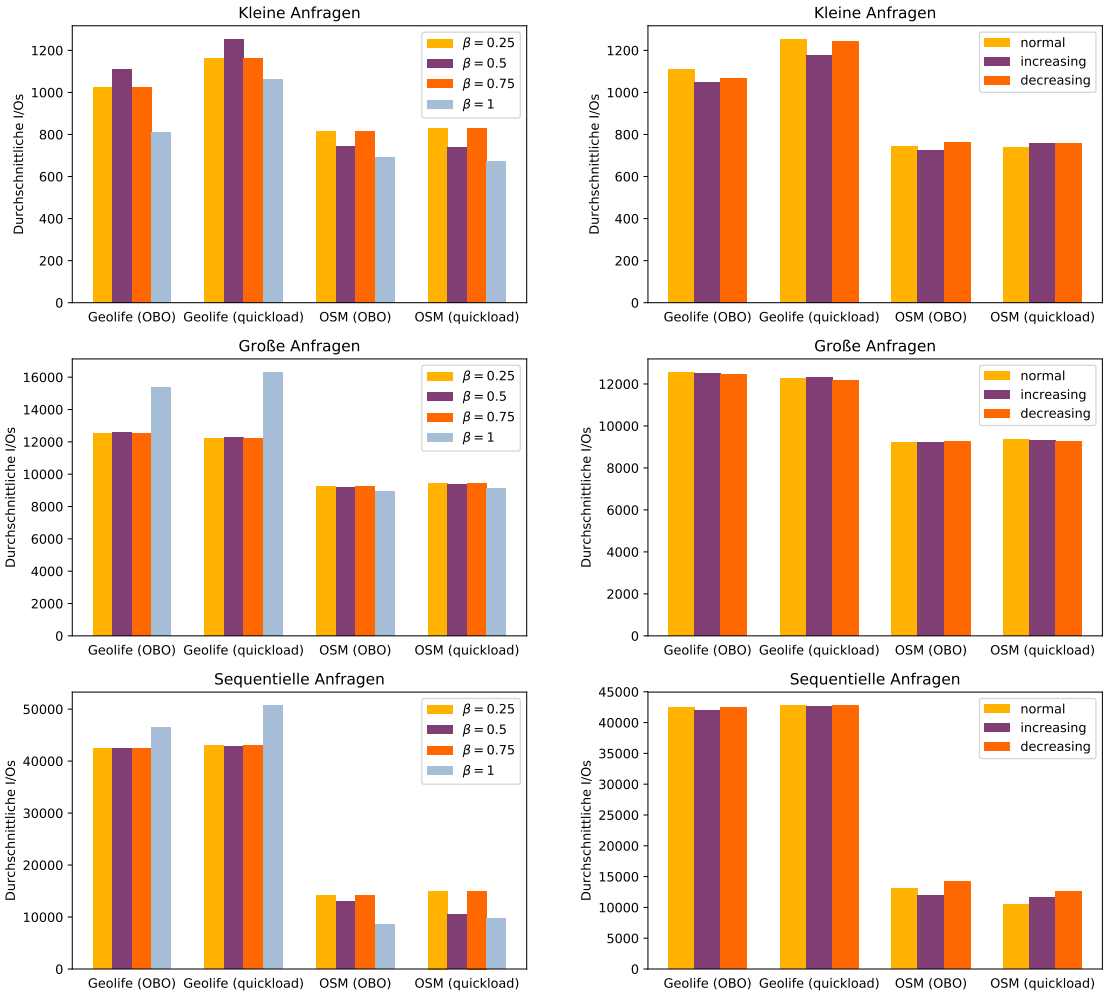
Das Standardverhalten: $\beta = 0.5$ unabhängig von der Höhe des Knotens.

Die Definition von $\hat{h}$ bewirkt, dass Blätter und die tiefste Ebene interner Knoten die gleiche Gewichtung der Kosten verwenden.

Abbildung 7.6 zeigt die Suchkosten für die in dieser Sektion erstellten Bäume. Grafik (a) enthält die durchschnittlichen Kosten für die Suche bei einem festen β. Für den Geolife-Datensatz sind die Kosten bei großen und sequentiellen Suchanfragen für $\beta = 1$ am größten; die Ergebnisse für kleine Werte unterscheiden sich hingegen kaum. Bei OSM wurden die besten Ergebnisse mit $\beta = 1$ erzielt. Die Ursache dafür wird sein, dass der Straßenname, der hier als Label dient, mit der räumlichen Position in Verbindung steht. Eine Priorisierung der räumlichen Kosten erreicht damit automatisch auch eine recht gute Verteilung der verschiedenen Label. Aus diesem Experiment kann kein allgemeingültiger, bester Wert für β abgeleitet werden; ein guter Wert ist von der Natur des Datensatzes abhängig. Es kann allerdings beobachtet werden, dass eine Veränderung des Parameters für Quickload und OBO die gleichen Auswirkungen zeigt. In Grafik (b) sind die Suchkosten für die verschiedenen Strategien dargestellt. Es ist keine besondere Veränderung bei den Ergebnissen zu beobachten.

7.8. Bloomfilter

In diesem Experiment wurden die im invertierten Index zum Einsatz kommenden Intervall-Sets durch Bloomfilter ausgetauscht. Ein Bloomfilter [Blo70] ist eine speichereffiziente, auf m Bits basierende Datenstruktur zur approximativen Darstellung von Mengen. Um einen Eintrag e in einen Bloomfilter einzufügen, wird unter der Verwendung k verschiedener Hashfunktionen $hash_1, ..., hash_k$ jeweils das Bit an der Stelle $hash_i(e)$ mod m auf eins gesetzt. Soll überprüft werden, ob ein Element enthalten ist, so werden dessen k Hashwerte



(a) Verschiedene feste Werte.

(b) β abhängig von der Knotenhöhe.

Abbildung 7.6.: Suchkosten bei verschiedenen Konfigurationen des Parameters β.

berechnet und genau dann *wahr* zurückgegeben, wenn alle k Bits gesetzt sind. Es ist offensichtlich möglich, dass es dabei für verschiedene Elemente zu Kollisionen kommt; ein Bloomfilter ist daher anfällig für *false-positives*. Wegen der Verwendung der Funktion Trim ist dies aber für Intervall-Sets genauso der Fall. Die in Algorithmus 7 und 8 benötigte Berechnung der Schnittmenge und der Vereinigung ist mit Bloomfiltern gleicher Größe und gleicher Hashfunktionen trivialerweise durch binäres UND oder ODER implementierbar.

Für die Resultate in Abbildung 7.7 wurde den Bloomfiltern die gleiche Speicherplatzgröße wie für $\lambda = 40$ Intervalle zugewiesen (320 Byte pro Eintrag in einer Posting List). Als Hashfunktion diente der Algorithmus Murmur3 [Mur16], welcher 128-Bit große Hash-

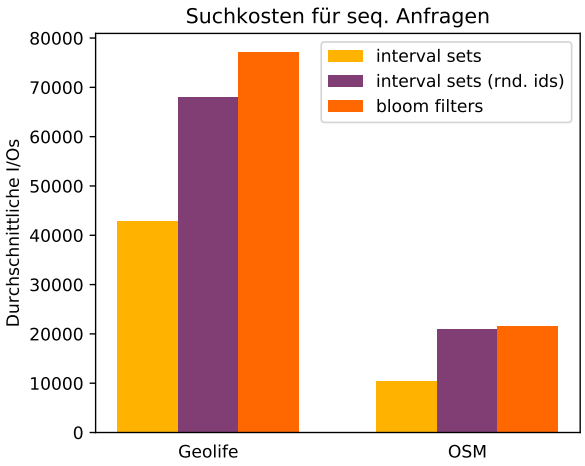


Abbildung 7.7.: Vergleich von Intervall-Sets und Bloomfiltern (Quickload).

werte liefert. Die ersten und letzten 64 Bits dieses Wertes werden als zwei unabhängige Hashfunktionen interpretiert und können mit dem von Kirsch und Mitzenmacher [KM08] beschriebenen Verfahren genutzt werden, um $k = 5$ Hashfunktionen für die Benutzung in den Bloomfiltern zu simulieren. Für die normalen Datensätze Geolife und OSM wurden die IDs der Trajektorien aufsteigend zugewiesen; die IDs von n Trajektorien stammen aus dem Intervall $[1, n]$. Für die mittleren Balken der Abbildung wurden die Trajektorien-IDs zufällig aus dem Intervall $[0, 2^{32} - 1]$ neu zugewiesen. In diesem Fall haben die Intervall-Sets eine schlechtere Qualität, da beim Reduzieren auf λ-Intervalle in TRIM sehr viel höhere Kosten entstehen und die Intervalle wegen des höheren Wertebereichs insgesamt deutlich größer sein müssen.

Trotzdem sind die Suchkosten für sequentielle Anfragen bei Bloomfiltern höher als bei beiden Varianten der Intervall-Sets, die Eliminierung von zu durchsuchenden Teilbäumen funktioniert also schlechter. Dafür kann es mehrere Ursachen geben:

- Der kompakte Wertebereich $[1, n]$ liefert Intervall-Sets mit vergleichsweise kleinem Fehler.
- Sowohl für Geolife wie auch für OSM gibt es häufig einen räumlichen Zusammenhang zwischen zwei Trajektorien mit aufeinanderfolgenden IDs. Bei Geolife gehören diese Trajektorien mit einiger Wahrscheinlichkeit zur selben Person (die Trajektorien einer Person werden nacheinander eingefügt); bei OSM können die Ausgangspunkte einer Route gleich sein (alle Routen, die von einer Stadt ausgehen, werden nacheinander erzeugt). Sowohl dieser wie auch der vorhergehende Punkt werden durch die

Randomisierung der IDs außer Kraft gesetzt.

- In Algorithmus 7 wird ein Teilbaum dann eliminiert, wenn die Schnittmenge mit den gemeinsamen IDs leer ist. Zwei Bloomfilter besitzen nur dann garantiert keine gemeinsamen Elemente, wenn ihr binäres UND kein einziges Bit mit dem Wert eins besitzt. Umgekehrt genügt ein einziges in beiden Filtern gesetztes Bit, damit ein Teilbaum durchsucht wird, obwohl dies eigentlich nicht nötig ist; eine Situation, die gegebenenfalls leicht entstehen kann.

Insgesamt ist daher für die benutzten Datenmengen das Intervall-Set die vorzuziehende Datenstruktur.

8. Zusammenfassung

Durch die Verwendung der verschiedenen *bulk-loading*-Verfahren ließen sich die Konstruktionskosten der IRWI-Datenstruktur um einen sehr großen Faktor verringern. Die Qualität des entstandenden Index hing dabei sowohl vom Verfahren wie auch vom verwendeten Datensatz ab; nur der Algorithmus Quickload lieferte im Vergleich zu OBO einen Baum mit konsistent ähnlichen Suchkosten. Bei den sortierbasierten Verfahren war es dagegen recht einfach, Situationen herzustellen, in denen ihre Performance besonders schlecht war. Obwohl Quickload in allen Experimenten das langsamste der getesteten Verfahren war, ist die Reduzierung in der Laufzeit im Vergleich zu OBO immer noch sehr groß. Wegen der im Vergleich besseren Struktur des resultierenden IRWI-Baums sollte Quickload daher gegenüber den anderen *bulk-loading*-Verfahren vorgezogen werden.

Die Experimente haben außerdem gezeigt, dass ein auf rein räumlichen Prinzipien arbeitendes Verfahren im Allgemeinen nicht angewendet werden sollte. Dies kann besonders klar bei der Anwendung von Hilbert-Packing auf den Random-Walk-Datensatz in den Sektionen 7.2 und 7.3 gesehen werden. Um für spatio-textuelle Daten eine gute Indexstruktur zu erhalten, sollten daher bei der Konstruktion sowohl die räumlichen Daten wie auch die textuellen Label berücksichtigt werden, wie es bei Quickload der Fall ist. Geschieht dies nicht, erhält man schnell einen unnötig großen invertierten Index.

Es gibt mehrere Punkte, die es in der Zukunft zu untersuchen lohnt. Einerseits offenbarte das Experiment in Sektion 7.6 gewisse Schwachstellen bei der Skalierbarkeit von Quickload, die gegebenenfalls behoben werden können. Es ist außerdem unklar, wie für einen Datensatz gute Werte oder Strategien für β gefunden werden können; die Ergebnisse in Sektion 7.7 ließen keine starken Folgerungen zu. Eine weitere Richtung ist die Erweiterung des *bulk-loadings* auf die Behandlung ganzer Trajektorien: wenn Trajektorien in ihrer Gesamtheit auf einmal eingefügt werden, könnte es von Vorteil sein, ähnlich wie beim STR-Baum oder TB-Baum vorzugehen. Indem man die Einträge einer Trajektorie näher beieinander speichert, lassen sich eventuell bessere Ergebnisse bei der Auswertung von

sequentiellen Suchanfragen erhalten, da die Anzahl der verschiedenen Trajektorien-IDs pro Knoten geringer ausfallen wird.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[APV02] Lars Arge, Octavian Procopiuc und Jeffrey Scott Vitter. „Implementing I/O-efficient Data Structures Using TPIE“. In: *Proceedings of the 10th Annual European Symposium on Algorithms*. ESA '02. London, UK, UK: Springer-Verlag, 2002, S. 88–100.

[AV88] Alok Aggarwal und S. Vitter Jeffrey. „The Input/Output Complexity of Sorting and Related Problems“. In: *Commun. ACM* 31.9 (Sep. 1988), S. 1116–1127.

[Ben77] Jon Louis Bentley. *Algorithms for Klee's rectangle problems*. Unpublished Notes. 1977.

[Blo70] Burton H. Bloom. „Space/Time Trade-offs in Hash Coding with Allowable Errors“. In: *Commun. ACM* 13.7 (Juli 1970), S. 422–426.

[BS01] Jochen Van den Bercken und Bernhard Seeger. „An Evaluation of Generic Bulk Loading Techniques“. In: *Proceedings of the 27th International Conference on Very Large Data Bases*. VLDB '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, S. 461–470.

[CCR02] Li Chen, Rupesh Choubey und Elke A. Rundensteiner. „Merging R-Trees: Efficient Strategies for Local Bulk Insertion“. In: *GeoInformatica* 6.1 (2002), S. 7–34.

[CJW09] Gao Cong, Christian S. Jensen und Dingming Wu. „Efficient Retrieval of the Top-k Most Relevant Spatial Web Objects“. In: *Proc. VLDB Endow.* 2.1 (Aug. 2009), S. 337–348.

[Cor+09] Thomas H. Cormen u. a. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009.

[DeW+94] David J. DeWitt u. a. „Client-Server Paradise“. In: *Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, S. 558–569.

[EN10] Ramez Elmasri und Shamkant Navathe. *Fundamentals of Database Systems*. 6th. USA: Addison-Wesley Publishing Company, 2010.

[GS05] Ralf Hartmut Güting und Markus Schneider. *Moving Objects Databases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[Gut84] Antonin Guttman. „R-trees: A Dynamic Index Structure for Spatial Searching“. In: *SIGMOD Rec.* 14.2 (Juni 1984), S. 47–57.

[GVD15] Ralf Hartmut Güting, Fabio Valdés und Maria Luisa Damiani. „Symbolic Trajectories“. In: *ACM Trans. Spatial Algorithms Syst.* 1.2 (Juli 2015), 7:1–7:51.

[Ham06] Chris H. Hamilton. *Compact Hilbert indices*. Techn. Ber. Dalhousie University, Faculty of Computer Science, Juli 2006.

[Hil91] Daniel Hilbert. „Über die stetige Abbildung einer Linie auf ein Flächenstück“. In: *Mathematische Annalen* 38 (1891), S. 459–460.

[HR07] C. H. Hamilton und A. Rau-Chaplin. „Compact Hilbert Indices for Multi-Dimensional Data“. In: *Complex, Intelligent and Software Intensive Systems, 2007. CISIS 2007. First International Conference on*. Apr. 2007, S. 139–146.

[ID16] H. Issa und M. L. Damiani. „Efficient Access to Temporally Overlaying Spatial and Textual Trajectories“. In: *2016 17th IEEE International Conference on Mobile Data Management (MDM)*. Bd. 1. Juni 2016, S. 262–271.

[KF93] Ibrahim Kamel und Christos Faloutsos. „On Packing R-trees“. In: *Proceedings of the Second International Conference on Information and Knowledge Management*. CIKM '93. Washington, D.C., USA: ACM, 1993, S. 490–499.

[KM08] Adam Kirsch und Michael Mitzenmacher. „Less Hashing, Same Performance: Building a Better Bloom Filter“. In: *Random Struct. Algorithms* 33.2 (Sep. 2008), S. 187–218.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[LEL97] Scott T. Leutenegger, Jeffrey M. Edgington und Mario A. Lopez. *STR: A Simple and Efficient Algorithm for R-tree Packing*. Techn. Ber. 1997.

[Li+11] Z. Li u. a. „IR-Tree: An Efficient Index for Geographic Document Search“. In: *IEEE Transactions on Knowledge and Data Engineering* 23.4 (Apr. 2011), S. 585–599.

[Li+13] D. Li u. a. „Efficient Bulk Loading to Accelerate Spatial Keyword Queries“. In: *2013 International Conference on Parallel and Distributed Systems*. Dez. 2013, S. 480–485.

[LKP95] Dik Lun Lee, Young Man Kim und Gaurav Patel. „Efficient Signature File Methods for Text Retrieval“. In: *IEEE Trans. on Knowl. and Data Eng.* 7.3 (Juni 1995), S. 423–435.

[LV11] Dennis Luxen und Christian Vetter. „Real-time routing with OpenStreetMap data“. In: *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. GIS '11. Chicago, Illinois: ACM, 2011, S. 513–516.

[Mur16] Austin Appleby. *Murmur3 Hash*. Frei verfügbar unter `https://github.com/aappleby/smhasher`. 2016.

[Ope17] OpenStreetMap contributors. `https://www.openstreetmap.org`. Datensatz abrufbar unter `https://planet.osm.org`. 2017.

[ORo85] Joseph O'Rourke. „Finding minimal enclosing boxes“. In: *International Journal of Computer & Information Sciences* 14.3 (1985), S. 183–199.

[Osg] *The OpenSceneGraph Project*. Software und Dokumentation verfügbar unter `http://www.openscenegraph.org/`.

[Osrm] OSRM contributors. *The OSRM Project*. Software und Dokumentation verfügbar unter `https://github.com/Project-OSRM`.

[Qt] *Qt Framework*. Software und Dokumentation verfügbar unter `https://www.qt.io/`.

[TP12] *TPIE – The Templated Portable I/O Environment*. Version 1.0 erschienen 14.12.2012. Software und Dokumentation verfügbar unter `http://www.madalgo.au.dk/tpie/`.

[Zhe+08] Yu Zheng u. a. „Understanding Mobility Based on GPS Data“. In: *Ubicomp 2008*. Ubicomp 2008, Sep. 2008.

[Zhe+09] Yu Zheng u. a. „Mining Interesting Locations and Travel Sequences From GPS Trajectories“. In: *WWW 2009*. WWW 2009, Apr. 2009.

[ZMR98] Justin Zobel, Alistair Moffat und Kotagiri Ramamohanarao. „Inverted Files Versus Signature Files for Text Indexing“. In: *ACM Trans. Database Syst.* 23.4 (Dez. 1998), S. 453–490.

[ZXM10] Yu Zheng, Xing Xie und Wei-Ying Ma. „GeoLife: A Collaborative Social Networking Service among User, location and trajectory“. In: *IEEE Data(base) Engineering Bulletin* (Juni 2010). URL: `https://www.microsoft.com/en-us/research/publication/geolife-a-collaborative-social-networking-service-among-user-location-and-trajectory/`.

A. Inhalt der DVD

```
DVD
|-- code                                Der Quellcode des C++-Projekts.
|   |-- cmd                             Quellcode der Werkzeuge.
|   |-- deps                            Gebündelte Dependencies (z.B. TPIE).
|   |-- geodb                           Quellcode der C++-Bibliothek.
|   \-- test                            Unit-Tests.
|-- data                                Datensätze zum Bauen von Bäumen (1).
|   |-- geolife.entries
|   |-- geolife-shuffled.entries
|   |-- geolife.strings
|   |-- geolife.trajectories
|   |-- osm.entries
|   |-- osm-shuffled.entries
|   |-- osm.strings
|   \-- osm.trajectories
|-- doc                                 Mit doxygen erzeugte API-Dokumentation.
|-- output                              Output der Experiemente (Bäume, Logs, etc.).
|-- results                             Ergebnisse (Rohdaten, Grafiken) (2).
|-- scripts                             Quellcode der Experimente.
|-- Makefile                            Führt Experimente aus (3).
|-- Masterarbeit.pdf                    Elektronische Version dieses Dokuments.
\-- README
```

(1) Die Rohdaten sind wegen ihrer Größe nicht mit enthalten (siehe nächste Seite). Die mitgelieferten „*.entries"-Dateien enthalten die aus den Rohdaten erzeugten Trajectory-Units und sind jeweils ein `tpie::file_stream` von `geodb::tree_entry`-Instanzen. Der Datensatz „Random-Walk" wird automatisch erzeugt. Die „*.strings"-Dateien liefern zu einem numerischen Label dessen lesbaren Text und können mit dem Tool `build/strings` ausgelesen werden.

(2) Enthält die zuletzt erziehlten Ergebnisse.

(3) Die Makefile dient zur Durchführung der Experimente und zur Erzeugung der Dateien in `results/`. Für eine Wiederholung der Experimente sollte der Inhalt von `results/` vorher geleert werden.

B. Quelle der Datensätze

Der Geolife-Datensatz wurde 2012 von Microsoft veröffentlicht und kann von der folgenden URL heruntergeladen werden:

`http://www.microsoft.com/en-us/download/details.aspx?id=52367`

Die Datensätze von OpenStreetMap können unter

`http://wiki.openstreetmap.org/wiki/Planet.osm`

heruntergeladen werden, die in dieser Arbeit verwendeten Daten sind auf den Bereich Deutschland beschränkt und stammen von

`http://download.geofabrik.de/europe.html`

(Unterpunkt „Germany", Datei `germany-latest.osm.bz`).

Für den Random-Walk-Datensatz sind keine externen Datenquellen erforderlich; er wird automatisch erstellt.