Fachpraktikum
Erweiterbare Datenbanksysteme
Wintersemester 2015/16
Aufgabe 3 der Phase 2

Indexe in Dateien

Ralf Hartmut Güting, Thomas Behr, Fabio Valdés, Holger Helmut Hennings

19.11.2015

Lehrgebiet Datenbanksysteme für neue Anwendungen
Fakultät für Mathematik und Informatik

FernUniversität in Hagen

1 Motivation

Indexe stellen Zugriffspfade dar, die ein schnelles Auffinden von Datensätzen innerhalb einer Relation ermöglichen. Dazu sind die zugrundeliegenden Datenstrukturen nach einem Attribut der Relation organisiert. Verschiedene Indexstrukturen können unterschiedliche Zugriffsarten unterstützen. Diese Aufgabe behandelt zwei sehr wichtige Indexstrukturen, nämlich den B^+-Baum sowie den R-Baum. Während der B^+-Baum vor allem der Indizierung von Standardattributen (*real*, *int*, *bool*, *string*) dient, werden R-Bäume bei der Indizierung geometrischer Objekte verwendet. Beide Indexstrukturen stehen in SECONDO bereits zur Verfügung. Diese werden in einer Datenbank gespeichert und sind an eine Relation, die in der gleichen Datenbank gespeichert ist, gebunden. Relationen sowie Indexe unterliegen dem Transaktionsmanagement, so dass die ACID-Eigenschaften von Transaktionen garantiert werden.

Neuerdings werden durch Datenbanken riesige Datenmengen verarbeitet, wobei eine Aufweichung der ACID-Eigenschaften zu Gunsten einer höheren Geschwindigkeit in Kauf genommen wird. Die Daten werden zur schnelleren Verarbeitung auf verschiedenen Computern bearbeitet. Auch in SECONDO gibt es Bestrebungen in diese Richtung. Als Beispiele seien die `HadoopAlgebra`, die `DistributedAlgebra` und die `Distributed2Algebra` genannt. Einige Aufgaben erfordern, dass Daten zwischen verschiedenen Computern ausgetauscht werden. Dazu werden die Daten in Dateien verpackt, die dann übertragen werden. Zur Geschwindigkeitssteigerung werden Operationen dann, wenn möglich, direkt auf den Dateien angewandt anstatt die Objekte wieder zeitraubend in die Datenbank zu integrieren. Während dies für Relationen bereits implementiert ist, fehlt eine solche Unterstützung für Indexe bislang. Somit muss derzeit für eine übertragene Relation ein neuer Index aufgebaut werden – eine langwierige Operation, die vermieden werden sollte.

Dem soll durch die Implementierung der `FileIndexAlgebra` Rechnung getragen werden. Diese Algebra soll die oben genannten Indexstrukturen in Dateien aufbauen können, Updates auf solchen Indexen erlauben und Indexabfragen gestatten.

2 B^+-Bäume

2.1 Struktur

B^+-Bäume sind der Klassiker schlechthin, wenn es um die Indizierung von Standarddaten geht. Es handelt sich dabei um Vielwegsuchbäume, bei denen die eigentlichen Datensatzverweise ausschließlich in den Blättern gespeichert sind. Wenn Sie den Kurs 1664 „Implementierungskonzepte für Datenbanksysteme“ belegt hatten, kennen Sie diese Struktur natürlich bereits. Für alle anderen gibt es innerhalb dieses Abschnitts eine kleine Zusammenfassung der Eigenschaften, des Aufbaus und der Algorithmen für solche Bäume.

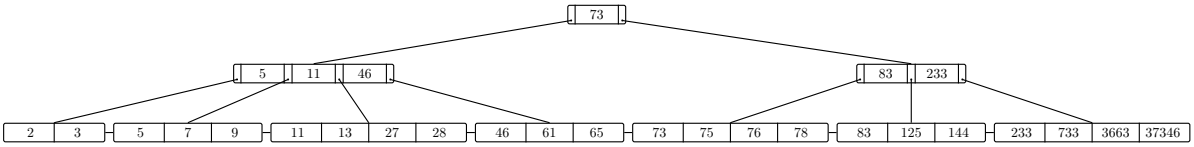
B^+-Bäume werden über einem Attribut einer Relation aufgebaut. Auf diesem Attribut muss eine Ordnung definiert sein. Dies ist in SECONDO grundsätzlich der Fall, da – wie Sie ja in der ersten Phase gelernt haben – jede Attributklasse eine `Compare` Funktion anbietet. Dennoch ist es nicht sinnvoll, über jedem möglichen Attributdatentyp einen B^+-Baum aufzubauen, da die Ordung häufig willkürlich gewählt wurde und/oder Vergleiche sehr teuer sein können. Bei den Attributen beschränkt sich diese Aufgabe daher auf *int*, *real*, *bool*, *string* sowie die Datentypen in der Kind `INDEXABLE`.

Innere Knoten eines B^+-Baums enthalten Attributwerte und Verweise auf Kindknoten. Dabei gibt es stets einen Verweis mehr, als Einträge im Knoten zu finden sind. Die maximale Anzahl an Einträgen wird so gewählt, dass die Inhalte eines solchen Knotens innerhalb einer einzelnen Seite im Dateisystem Platz finden. In der Datei `WinUnix.h` gibt es eine Funktion, um die Seitengröße des Systems zu ermitteln. Jeder Knoten im Baum ist mindestens zur Hälfte gefüllt. Eine Ausnahme bildet die Wurzel des Baums, die auch weniger Einträge besitzen kann. Die Einträge innerhalb eines Knotens sind sortiert. Die Teilbäume, die durch die Söhne gegeben sind, enthalten ausschließlich solche Attributwerte, die zwischen den beiden umschließenden Einträgen liegen, wobei Gleichheit erlaubt ist.

In den Blättern gibt es statt der Verweise auf Söhne sogenannte Datensatzidentifikatoren. Innerhalb von SECONDO wird dazu eine `TupleId` verwendet, die in der `TupleIdentifierAlgebra` definiert ist.

Weiterhin sind die Blattknoten eines solchen Baums doppelt miteinander verkettet, um einen geordneten sequentiellen Durchlauf über die Blattknoten zu ermöglichen.

Ein Beispiel eines B^+-Baums:



2.2 Einfügen

Beim Einfügen eines neuen Elements wird zunächst von der Wurzel dem Pfad bis zum Blatt gefolgt, bei dem der Eintrag hinzukommen wird. Dort wird der Eintrag hinzugefügt. Passt der neue Eintrag noch in das Blatt, ist das Einfügen beendet. Hat einer der Nachbarknoten (1 oder 2) noch freie Kapazitäten, so wird mit diesem ausgeglichen, indem die äußeren Einträge des übergelaufenden Knotens in diesen Knoten verschoben werden. Der Attributwert im Vaterknoten muss dabei ebenfalls angepasst werden. Die Anzahl der zu verschiebenden Einträge wird so festgelegt, dass anschließend beide Knoten etwa die gleiche Anzahl an Einträgen besitzen.

Falls der Knoten selbst überläuft und seine Nachbarknoten bereits vollständig gefüllt sind, so wird der Knoten gesplittet, d.h. zwei etwa halbvolle Knoten erzeugt. Die Verkettung der Blätter wird korrigiert. Der linkeste Eintrag des rechten Knotens wird in den Vaterknoten kopiert. Läuft der Vaterknoten über, so wird ebenfalls zunächst versucht, mit einem Nachbarn auszugleichen. Ist dies nicht möglich, so wird auch hier gesplittet. Im Unterschied zu den Blattknoten wird bei inneren Knoten der mittlere Eintrag in den Vatenknoten verschoben (nicht kopiert). Dies kann sich bis zur Wurzel fortsetzen. Läuft diese über, so wird eine neue Wurzel erzeugt, die nur einen Eintrag und zwei Söhne besitzt.

2.3 Löschen

Beim Löschen wird zunächst das entsprechende Blatt gesucht und der Eintrag dort gelöscht. Ist der Knoten noch zur Hälfte gefüllt, so ist das Löschen beendet. Entsteht ein Unterlauf, aber einer der Nachbarknoten hat noch mehr als die notwendige Anzahl an Einträgen, so wird mit diesem Nachbarn ausgeglichen. Dies funktioniert so wie der Ausgleich beim Einfügen. Gibt es einen Unterlauf und sind die Nachbarn minimal gefüllt, so wird mit einem der Nachbarn verschmolzen. Der entsprechende Eintrag im Vaterknoten wird gelöscht. Die Unterlaufbehandlung kann sich bis zur Wurzel fortsetzen. Wird der letzte Eintrag der Wurzel entfernt, so wird deren nun einziges Kind zur neuen Wurzel.

2.4 Bulkload

Werden Elemente nacheinander in einen B^+-Baum eingefügt, so gibt es in der Regel häufige Umstrukturierungen, was viele Dateioperationen nach sich zieht. Daher ist es häufig günstig, einen sogenannten Bulkload zu implementieren. Dabei werden die einzelnen Elemente nach dem zu indizierenden Attribut sortiert eingefügt. Somit kann zunächst das erste Blatt komplett gefüllt werden. Passt das nächste Element nicht mehr in dieses Blatt, so wird ein neues Blatt sowie ein Vaterknoten erzeugt. Dieses Blatt wird wieder gefüllt, bis ein neues Blatt generiert werden muss. Im Vaterknoten wird ein neuer Eintrag erstellt. Dies setzt sich fort, bis alle Elemente verarbeitet wurden. Läuft der Vaterknoten über, wird dieser nicht gesplittet, sondern auf die Platte geschrieben und ein neuer Nachbar des Vaterknotens wird erzeugt, der nur das übergelaufene Element enthält. Die beiden Knoten erhalten einen gemeinsamen Vaterknoten, der die neue Wurzel des Baums darstellt. In späteren Durchgängen kann sich dieses Vorgehen über mehrere Ebenen erstrecken. Am Ende kann es passieren, dass auf dem Pfad von der Wurzel zum rechtesten Blatt Knoten existieren, die keine ausreichende Füllung aufweisen. Da jedoch die linken Nachbarn dieser Knoten stets voll gefüllt sind, ist hier ein Ausgleich möglich. Während des Bulkloads werden nur solche Seiten in die Datei geschrieben, die nicht mehr betrachtet werden müssen, das sind solche Knoten, die nicht auf dem Pfad von der Wurzel zum aktuell rechtesten Blatt liegen. Die Knoten auf diesem Pfad werden im

Hauptspeicher gehalten. Da die Höhe des Baums logarithmisch zur Anzahl seiner Einträge ist, ist dies selbst bei einer größeren Anzahl an Einträgen möglich.

2.5 Suchen

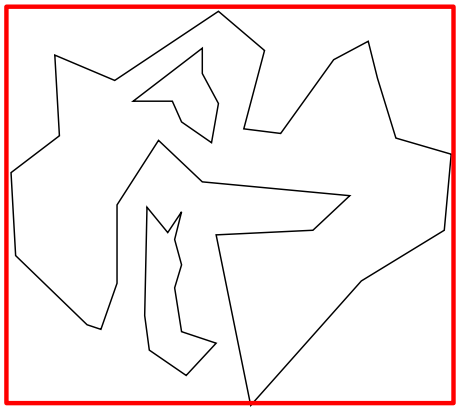
Es gibt zwei Arten von Anfragen auf solche Bäume, Punktanfragen, bei denen nach einem bestimmten Wert gesucht wird, und Bereichsanfragen, bei denen alle Datensatzidentifikatoren ausgegeben werden, deren dazugehörige Attributwerte in einem gegebenen Intervall liegen. Die Punktsuche kann auf die Intervallsuche abgebildet werden, indem man dem linken und dem rechten Intervallende den gleichen Wert zuweist. Auch bei der Punktsuche kann es mehrere Ergebnisse geben, da die Attribute in der Relation nicht eindeutig sein müssen. Bei der Bereichssuche wird von der Wurzel aus zu dem Blatt hinabgestiegen, das den kleinsten Wert, der größer oder gleich dem linken Intervallende ist, enthält. Von dort startet man einen Scan über die verketteten Blätter, bis ein gespeicherter Wert größer als das rechte Intervallende ist. Die auf diesem Weg gefundenen Tupel-IDs bilden das Ergebnis einer solchen Anfrage.

2.6 Implementierung der Struktur

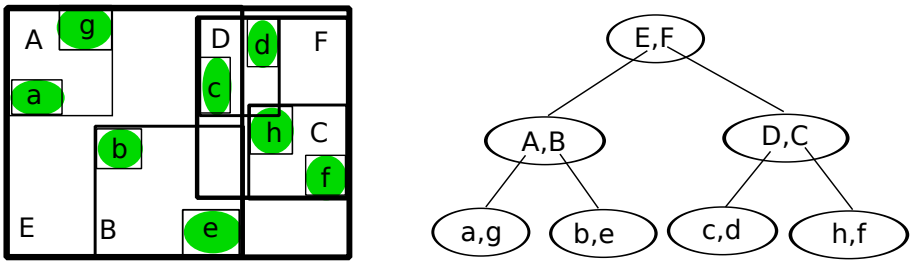
Der Index soll nicht, wie die in SECONDO existierenden Indexe, in der Datenbank, sondern innerhalb einer gewöhnlichen Datei gespeichert werden. Diese Datei hat am Anfang einige Header-Informationen. Diese enthalten mindestens einen Marker, den indizierten Typ, die verwendete Seitengröße und die Seitennummer der Seite mit der Wurzel. Weiterhin können dort auch Informationen wie Anzahl der Einträge usw. untergebracht werden. Um nicht ständig auf dem Dateisystem zu arbeiten, wird eine beschränkte Anzahl an Seiten im Hauptspeicher gehalten. Passen neue Seiten nicht mehr in diesen Cache, so werden lange nicht benutzte Seiten in die Datei geschrieben, um Platz für die neuen Seiten zu machen. Tatsächlich müssen nur geänderte Seiten in die Datei übertragen werden, andere Seiten können einfach gelöscht werden. Dazu bietet sich eine LRU-Strategie an, bei der die Seite verdrängt wird, deren letzte Benutzung am längsten her ist. Das Herausschreiben in die Datei ist nur dann erforderlich, wenn sich die Hauptspeicherseite und die Seite in der Datei unterscheiden. Über eine Funktion können alle Seiten im Cache, die geändert wurden, auf die Festplatte geschrieben werden. Diese Funktion wird beispielsweise am Ende eines SECONDO-Operators aufgerufen, um sicherzustellen, dass der Index nun vollständig in der Datei vorliegt. Wird eine Seite nicht mehr benötigt, so wird diese in eine Freispeicherliste übergeben. Die Nummer der ersten freien Seite wird im Header angegeben. Auf einer freien Seite steht die Nummer der nächsten freien Seite (oder 0 im Fall der letzten freien Seite), so dass sich eine einfach verkettete Liste freier Seiten ergibt. Einfügen und Löschen in dieser Liste werden stets am Listenanfang durchgeführt, so dass sich die Methoden wie die eines Stacks verhalten. Wird eine neue Seite benötigt, so wird zunächst eine Seite aus der Freispeicherliste verwendet. Steht dort keine Seite zur Verfügung, so wird eine neue Seite ans Ende der Datei gehängt.

3 R-Bäume

Mit R-Bäumen lassen sich geometrische Objekte indizieren. Da diese recht umfangreich werden können, werden die Geometrien nicht direkt als Einträge im R-Baum verwendet, sondern durch die sogenannte Bounding Box approximiert. Die Bounding Box ist das kleinste achsenparallele Rechteckt, das die gesamte Geometrie des Objekts einschließt.



Der Aufbau des R-Baums ähnelt dem des B^+ Baums. So wird jeder Knoten durch eine Dateiseite dargestellt, die Verweise auf die Datensätze befinden sich in den Blättern und es wird eine minimale Füllung der Knoten bis auf die Wurzel gefordert. Diese muss jedoch nicht zwingend die Hälfte der maximalen Einträge sein, wie es bei B^+-Bäumen der Fall ist. Zu jedem indizierten Objekt wird dessen Bounding-Box sowie die entsprechende Tupel-ID in einem Blattknoten gespeichert. Eine Ordnung der Elemente innerhalb eines Knotens gibt es im Gegensatz zu B^+-Bäumen nicht. In den inneren Knoten werden ebenfalls Rechtecke sowie Verweise auf Kindknoten gespeichert. Das Rechteck ist dabei das kleinste Rechteck, in das alle Rechtecke des Kinds passen. Es ist erlaubt, dass Rechtecke sich überlappen.



3.1 Einfügen

Das Einfügen in einen R-Baum verläuft nach folgendem Schema:

- starte an der Wurzel
- solange kein Blatt erreicht wurde, wähle besten Sohn
- füge den Eintrag in das Blatt ein
- solange ein Überlauf stattfindet, splitte Knoten und trage neuen Eintrag in Vaterknoten ein
- falls Wurzel gesplittet wird, erzeuge neue Wurzel

Offensichtlich entspricht der Einfügealgorithmus dem des B^+-Baums, wobei kein Ausgleich zwischen Nachbarknoten stattfindet. Es sind noch zwei Dinge zu klären. Wie wähle ich den *besten* Sohn? Wie splitte ich einen Knoten?

Der beste Sohn wird durch folgende Sequenz von Bedingungen ermittelt:

- wähle Sohn mit geringstem Flächenzuwachs
- wenn gleiche dabei: wähle Sohn mit geringster Fläche
- wenn gleiche dabei: wähle Sohn mit den wenigsten Einträgen

- wenn gleiche dabei: wähle willkürlich

Für das Splitten eines R-Baum Knotens gibt es unterschiedliche Strategien. Innerhalb dieser Aufgabe ist der sogenannte „Quadratic Split“ zu implementieren. Dieser funktioniert wie folgt:

- erzeuge zwei neue, leere Knoten
- füge diejenigen zwei Einträge, deren Vereinigung der Bounding Boxen maximal ist, in die beiden Knoten ein
- für alle anderen Elemente
  - wenn Minimalfüllung es erfordert, füge restliche Elemente einem Knoten hinzu
  - sonst
    * berechne für alle Elemente den Flächenzuwachs in beiden Knoten
    * wähle das Element mit der größten absoluten Differenz
    * füge in den Knoten mit geringerem Flächenzuwachs ein
    * falls gleich in den Knoten mit kleinerer Fläche
    * falls gleich in den Knoten mit weniger Einträgen
    * falls gleich wähle willkürlich

3.2 Suchen

Es gibt zwei Arten von Suchanfragen an einen R-Baum. Bei der Punktsuche werden alle Tupel-IDs gesucht, bei denen ein gegebener Punkt innerhalb der dieser ID zugeordneten Bounding Box liegt. Die Bereichssuche liefert hingegen die Tupel-IDs, deren zugehörige Rechtecke das gegebene Anfragerechteck schneiden. Die Punktsuche stellt somit einen Spezialfall der Bereichssuche dar, bei dem das Anfragerechteck zu einem Punkt degeneriert ist. Im Folgenden wird daher nur die Bereichssuche beschrieben.

Wie üblich, startet die Suche an der Wurzel des Baums. Vom aktuellen Knoten aus werden alle Söhne besucht, bei denen die Bounding Box das Anfragerechteck schneidet. Im Gegensatz zum B^+-Baum werden hier also möglicherweise mehrere Pfade verfolgt. Ist man bei einem Blatt angelangt, werden dort alle Einträge betrachtet und die Tupel-IDs der Einträge, deren Bounding Boxen das Anfragerechteck schneiden, ausgegeben.

3.3 Löschen

Das Löschen in einem R-Baum folgt dem Muster:

- suche Blatt, das das zu löschende Element enthält
- bei einem Unterlauf, merke Knoten zusammen mit seiner Ebene und lösche den gesamten Knoten da hierbei im Vaterknoten ein Element gelöscht wird, kann sich dies bis zur Wurzel hinziehen
- füge alle zwischengespeicherten Elemente auf ihren alten Ebenen wieder ein
  (gleiches Prinzip wie das Einfügen eines Eintrags)
- falls Wurzel nur noch einen Sohn hat, wird dieser zur neuen Wurzel

4 Bulkload

Auch für R-Bäume ist ein Bulkload möglich, falls die Rechtecke so eintreffen, dass nahe beieinanderliegende Rechtecke hintereinander im Tupelstrom auftreten. Hierbei kann es jedoch auch vorkommen, dass ein neuer Knoten entsteht, ohne dass der aktuelle Blattknoten voll ist. Dies ist der Fall, wenn das neu einzufügende Rechteck zu weit weg von der Bounding-Box des aktuellen Blatts ist. Eine Forderung nach

einer minimalen Anzahl an Einträgen wird daher im Bulkload-Modus ignoriert, d.h. es kann sogar Knoten mit nur einem Eintrag geben. Dennoch wird die minimale Anzahl an Elementen gespeichert, so dass nach entsprechend vielen Aktualisierungen des Baums die Minimalfüllung der Knoten wieder hergestellt werden kann.

5 Implementierung

Wie bereits bei den B^+-Bäumen soll der Baum in einer gewöhnlichen Datei dargestellt werden. Der Baum soll 1, 2, 3, 4 und 8-dimensionale Rechtecke verwalten können, daher ist die Klasse als Template-Klasse zu implementieren. Im Dateiheader werden ein Marker, die Seite des Wurzelknotens, die verwendete Seitengröße, die minimale Füllung eines Knoten sowie ein Verweis auf eine Liste gelöschter Seiten gespeichert. Weiterhin enthält der Header die Bounding Box aller im Baum enthaltenen Elemente. Um häufige Dateizugriffe zu vermeiden, wird ein Cache verwendet, der kürzlich verwendete Seiten vorhält.

Eine Besonderheit stellen Anfragen an einen R-Baum dar. Während bei B^+-Bäumen nur ein einzelner Pfad bis zu einem Blatt verfolgt werden muss und anschließend der Verkettung der Blätter gefolgt wird, ist dies bei einem R-Baum aufgrund einer fehlenden Ordung auf den Blättern nicht möglich. Da auch noch mehrere Pfade verfolgt werden müssen, scheint sich eine rekursive Lösung anzubieten. Da aus einer Rekursion heraus jedoch keine Elemente zurückgeliefert werden können und somit alle Ergebnisse während der Rekursion zunächst eingesammelt werden müssten, scheidet eine solche Lösung aus. Stattdessen wird ein Stack verwendet, dessen Elemente den jeweiligen Knoten sowie den letzten bearbeiteten Eintrag enthalten. Mit solch einer Struktur kann die Rekursion simuliert werden. Weiterhin ist es damit möglich, ein gefundenes Ergebnis sofort in den Ausgabestrom zu schreiben und so aufwändiges Zwischenspeichern von Ergebnissen zu vermeiden. Dies kann in einer Iterator-Klasse gekapselt werden, die als LocalInfo innerhab des ValueMappings verwendet wird.

6 Operationen der Algebra

Da sämtliche Strukturen ausschließlich in Dateien existieren oder sich temporär im Hauptspeicher aufhalten, bietet diese Algebra keine neuen Typen, sondern ausschließlich Operatoren an.

6.1 Operationen für B^+-Bäume

6.1.1 Erzeugen

`createfbtree`: *stream*(*tuple*(X)) × {*text*, *string*} × Ident × Ident → *stream*(*tuple*(X))

Dieser Operator erhält einen Tupelstrom, den Dateinamen für den Baum und zwei Attributnamen. Im Strom befinden sich die einzelnen Datensätze, die indiziert werden sollen. Der Dateiname kann als String oder als Text angegeben werden. Der erste Attributname spezifiziert dasjenige Attribut, nach dem der B^+-Baum organisiert werden soll. Dieses stammt entweder aus der `StandardAlgebra` oder gehört zur Kind `INDEXABLE`. Der zweite Attributname gibt ein Attribut im Tupelstrom vom Typ *tid* an. Dieses enthält den jeweiligen Datensatzidentifikator. Der eingehende Strom wird unverändert in die Ausgabe weitergegeben. Der Operator ist als Postfixoperator zu implementieren. Eine Beispielanfrage ist:

```
query strassen feed addid createfbtree["strassen_name_btree.bin', Name, TID] count
```

`bulkloadfbtree`: *stream*(*tuple*(X)) × {*text*, *string*} × Ident × Ident → *stream*(*tuple*(X))

Dieser Operator funktioniert sehr ähnlich zum `createfbtree`-Operator mit dem Unterschied, dass der eingehende Tupelstrom sortiert vorliegen muss. iZusätzlich wird der Baum wird im Bulkload-Modus aufgebaut. Elemente, die nicht entsprechend sortiert sind, werden ignoriert, jedoch an den Ausgabestrom weitergereicht.

```
query strassen feed addid sortby[Name]
        bulkloadfbtree["strassen_name_btree.bin", Name, TID] count
```

6.1.2 Updates

Eine besondere Schwierigkeit bei den nachfolgenden Operatoren ist, dass bereits im TypeMapping geprüft wird, ob es sich bei dieser Datei tatsächlich um einen B^+-Baum handelt und ob der dort indizierte Typ mit dem erwarteten Typ übereinstimmt. Hierzu wird bereits Zugriff auf den Dateinamen benötigt, da lediglich der Typ *string* oder *text* nicht aussagekräftig genug ist. Wie dies innerhalb von SECONDO implementiert werden kann, zeigt der Abschnitt „Accessing values in Type Mappings“ in der `GuideAlgebra`.

`insertfbtree`: *stream*(*tuple*(X)) × {*text*, *string*} × Ident × Ident → *stream*(*tuple*(X))
Dieser Operator erhält die gleichen Argumente wie die Operatoren zum Erzeugen von B^+-Bäumen. Dieser Operator erstellt jedoch keine neue Datei, sondern fügt neue Elemente in einen bestehenden dateibasierten B^+-Baum ein. Im TypeMapping wird geprüft, ob die angegebene Datei ein B^+-Baum ist, der den gleichen Typ indiziert wie das angegebene Attribut.

```
query neuestrassen feed addid "strassen_name_btree.bin" insertfbtree[Name, TID] count
```

`deletefbtree`: *stream*(*tuple*(X)) × {*text*, *string*} × Ident × Ident → *stream*(*tuple*(X))
Hiermit werden Elemente aus einem dateibasierten B^+-Baum gelöscht. Die Argumente entsprechen denen des Operators `insertfbtree`.

```
query neuestrassen feed addid "strassen_name_btree.bin" deletefbtree[Name, TID] count
```

6.1.3 Anfragen

`frange`: {*text*, *string*} × T × T → *stream*(*tid*)
Mit Hilfe dieses Operators werden alle Tupel-IDs aus dem Baum extrahiert, deren dazugehörige Werte im angegebenen Intervall liegen. Im ersten Argument wird der Dateiname des Baums angegeben. Die beiden anderen Argumente definieren das Anfrageintervall. Das Ergebnis ist ein Strom von Tupel-IDs.

```
query "strassen_name_btree.bin" frange["A","B"] strassen gettuples consume
```

`fleftrange`: {*text*, *string*} × T → *stream*(*tid*)
Auch dieser Operator beschreibt eine Intervallsuche, wobei in dieser Variante lediglich das rechte Intervallende angegeben wird. Hier wird also der Pfad zum äußersten linken Blatt gesucht und alle Elemente bis einschließlich dem gegebenen Wert abgearbeitet.

```
query "strassen_btree.bin" fleftrange["Syringenweg"] strassen gettuples consume
```

`frightrange`: {*text*, *string*} × T → *stream*(*tid*)
In dieser Variante der Bereichssuche werden alle Tupel-IDs ausgegeben, bei denen die dazugehörigen Objekte größer oder gleich dem übergebenen Element sind.

```
query "strassen_btree.bin" frightrange["Syringenweg"] strassen gettuples consume
```

`fexactmatch`: {*text*, *string*} × T → *stream*(*tid*)
Das Ergebnis dieses Operators entspricht einer Intervallsuche, bei der das linke und rechte Intervallende gleich sind. Es werden also genau diejenigen Elemente ausgewählt, bei denen der gespeicherte Attributwert dem Suchwert entspricht. Da die gespeicherten Werte nicht eindeutig sind, sind auch bei diesem Operator mehrere Ergebnisse möglich, so dass das Ergebnis ebenfalls ein Strom von Tupel-IDs ist.

```
query "strassen_btree.bin" fexactmatch["Syringenweg"] strassen gettuples consume
```

6.1.4 Restrukturierung

Durch häufiges Einfügen und Löschen können Dateien entstehen, bei denen aufeinanderfolgende Blätter innerhalb der Datei sehr weit voneinander entfernt sind. Weiterhin können viele unbenutzte Seiten innerhalb der Datei existieren. Während das erstgenannte Problem die Suche im Index selbst verlangsamt,

verlangsamen unbenutzte Seiten die Übertragung von einem Computer auf einen anderen. Aus diesen Gründen ist es wünschenswert, einen solchen Baum wieder in eine bessere Form zu bringen. Die Inhalte der Knoten werden, bis auf die Anpassung der Referenzen, nicht verändert. Dies soll der folgende Operator bewerkstelligen.

`rebuildfbtree`: {*string*, *text*} × {*string*, *text*} → *bool*

Dieser Operator nimmt einen Baum, der in einer Datei gespeichert ist, und erzeugt daraus einen Baum mit dem gleichen Aufbau, indem die Knoten jedoch besser innerhalb der Datei verteilt sind. So befinden sich in der neuen Datei sämtliche Blattknoten entsprechend ihrer natürlichen Reihenfolge in aufeinanderfolgenden Seiten der Datei und die Liste der freien Seiten ist leer.

6.2 Operationen für R-Bäume

6.2.1 Erstellen

`createfrtree`: *stream*(*tuple*(X)) × {*text*, *string*} × Ident × Ident × *int* → *stream*(*tuple*(X))

Die vorderen Argumente dieses Operators entsprechen denen des Operators `createfbtree`. Das letzte Argument vom Typ *int* gibt die minimale Anzahl an Elementen innerhalb eines Knotens an. Liegt der dort angegebene Werte außerhalb des erlaubten Bereichs, so wird dieser angepasst. Wird dort ein undefinierter Integer angegeben, so wird die Hälfte der Maximalanzahl an Einträgen angenommen. Der erste Attributname spezifiziert ein Attribut im Tupel, das entweder ein Rechteck oder in der Kind SPATIAL?D ist. Die zu unterstützenden Dimensionen wurden bereits genannt. Der zweite Attributname gibt ein Attribut vom Typ *tid* an, das als Datensatzidentifikator verwendet wird.

`bulkloadfrtree`: *stream*(*tuple*(X)) × {*text*, *string*} × Ident × Ident × *int* × *real* → *stream*(*tuple*(X))

Auch dieser Operator erzeugt einen R-Baum aus einem Tupelstrom, allerdings unter Verwendung eines Bulkload-Algorithmus. Das zum Operator `createfrtree` zusätzliche Argument vom Typ *real* gibt an, ab welcher Entfernung des neuen Rechtecks zur Bounding Box des aktuellen Blatts ein neuer Knoten zu beginnnen ist, auch wenn das Blatt noch nicht gefüllt ist. Ist dieser Wert undefiniert oder kleiner oder gleich Null, so wird nur dann ein neues Blatt angefangen, wenn das aktuelle Blatt voll ist.

6.2.2 Updates

`insertfrtree`: *stream*(*tuple*(X)) × {*text*, *string*} × Ident × Ident → *stream*(*tuple*(X))

Mit diesem Operator werden neue Einträge in einen bestehenden dateibasierten R-Baum eingefügt. Im Gegensatz zu den erzeugenden Operatoren, braucht hier keine Minimalanzahl von Einträgen angegeben zu werden, da diese aus dem existierenden Baum übernommen wird.

`deletefrtree`: *stream*(*tuple*(X)) × {*text*, *string*} × Ident × Ident → *stream*(*tuple*(X))

Durch Anwendung dieses Operators werden Elemente aus einem existierenden R-Baum gelöscht. Die Argumente entsprechen denen des Operators `insertfrtree`.

6.2.3 Anfragen

`fwindowintersects`: {*string*, *text*} × *R* → *stream*(*tid*)

Wobei *R* ∈{*rect*, *rect2*, *rect3*, *rect4*, *rect8*, *SPATIAL*, *SPATIAL3D* } gilt.

Das erste Argument gibt den Namen der Datei an, in der der R-Baum gespeichert ist. Wie auch bei den B^+-Bäumen muss hier bereits im TypeMapping auf diese Datei zugegriffen werden, um zu prüfen, dass diese existiert, es sich um einen R-Baum handelt und die Dimension korrekt ist. Das zweite Argument beschreibt das Rechteck, mit dem die Suche durchgeführt werden soll. Objekte in der Kind SPATIAL (SPATIAL3D) bieten eine Funktion `BoundingBox` an, die ein `Rectangle<2>` (`Rectangle<3>`) zurückliefert. Mit diesen Rechtecken wird die Suche dann durchgeführt.

6.2.4 Restrukturierung

Bei R-Bäumen können ähnliche Probleme wie bei den B^+-Bäumen auftreten. Da jedoch keine Durchlaufreihenfolge durch die Blätter festgelegt ist, können diese nicht optimal angeordnet werden. Daher

beschränkt sich die Restrukturierung bei den R-Bäumen auf das Löschen von leeren Seiten in der Datei.

`rebuildfrtree`: $\{\underline{text}, \underline{string}\} \times \{\underline{text}, \underline{string}\} \rightarrow \underline{bool}$