Philipps Universität Marburg

Fachbereich 12 Mathematik und Informatik

Diplomarbeit
im Studiengang Informatik

Effiziente Indexstrukturen für zeitabhängige RDF Daten

Autor

Daniar Achakeyev

geboren am 11.04.1982 in Kharkov, Ukraine

Themenstellung und Betreuung: Prof. Dr. B. Seeger

Marburg (Lahn), im Februar 2009

Zusammenfassung

Resource Description Framework (RDF) hat im Laufe der Zeit mehr und mehr an Bedeutung gewonnen. Durch ein einfaches Konzept und eine pragmatische Semantik kann RDF als ein alternatives Modell für die Datenverwaltung in heterogenen Umgebungen in Betracht gezogen werden. Insbesondere eignet sich RDF für Integration von Daten aus verschiedenen Quellen. Nicht zu vergessen, dass RDF das Rückgrat für das Semantische Web und die Basis für die Zukunftsvision vom Zusammenspiel von intelligenten Maschinen bildet.
Viele Applikationen aus verschiedenen Bereichen nutzen das Framework für Annotation von Daten und Bereitstellung von Meta-Daten für semantische Analyse. Dabei werden Datenmengen gesammelt, die in Größenordnung von mehreren Gigabytes sind. Diese Größenordnungen von typischen RDF Datenbanken fordern die effiziente Zugriffs- und Anfragefunktionalität. Die Verwaltung von großen Mengen von RDF Daten in Form von speziellen Tabellen in einer relationalen Datenbank hat sich in der Praxis etabliert. Im Laufe der Entwicklung der RDF Datenbanken wurde intensiv der Einsatz von konventionellen Indexstrukturen aus relationalen Datenbanken erforscht. Auch spezielle neue Strukturen, die funktionale Besonderheiten von RDF berücksichtigen, wie die Graphstruktur von RDF, wurden entworfen.
Viele Applikationen für RDF verwenden Daten mit temporalem Bezug. Die Arbeit beschäftigt sich mit der Verwaltung von großen Mengen von temporalen RDF Daten. Der Schwerpunkt der Forschung in dem Indexstrukturenbereich für RDF Daten lag bisher auf der Verwaltung von RDF Daten ohne zeitlichen Bezug. Im Zuge der Entwicklung von RDF Indexstrukturen wurde aber auch eine spezielle Indexstruktur für temporale RDF Graphen entworfen. Diese kann limitierte temporale Anfragen ausführen. Im Laufe der Arbeit wird die allgemeine Möglichkeit der Verwendung und Anpassung von temporalen Index- und Zugriffstrukturen für RDF Datenbanken untersucht. Eine der entscheidenden Rollen spielt dabei die effiziente Verarbeitung von komplexen temporalen Anfragen.
Dabei wird ähnlich dem relationalen Modell vorgegangen. Zuerst werden theoretische Grundlagen für die Einführung der Zeitdimension in RDF betrachtet. Danach wird anhand der abstrakten temporalen Modelle die Möglichkeit des Einsatzes von temporalen Strukturen aus der relationalen Welt untersucht.

Inhaltsverzeichnis

Einleitung

Diplomarbeitaufbau

Im ersten Kapitel geht es um der Resource Description Framework. Im Kapitel werden syntaktische und semantische Eigenschaften des RDF erläutert. Dabei wird auf die praktische Relevanz von RDF eingegangen. Im Verlauf des Kapitels wird das RDF Datenmodell mit dem Relationalen Modell verglichen. Im Anschluss des Kapitels werden die möglichen Verwaltungstechniken für RDF Daten der Reihe nach präsentiert.
Das zweite Kapitel beschäftigt sich mit der Zeitdimension in RDF. In der Einführung des Kapitels werden die Beispiele und Argumentationen für die Notwendigkeit der Verwaltung von zeitabhängigen Daten angegeben. Im ersten Abschnitt werden die formale Methoden und Begriffe für die Zeitdarstellung innerhalb des relationalen Modells vorgestellt. Im zweiten Abschnitt wird eine Möglichkeit, die Modelle aus der relationalen Welt auf das RDF zu übertragen, präsentiert.
Im dritten Kapitel geht es um die Zugriffstrukturen für RDF Daten. In der Einführung werden allgemeine Begriffe in Zusammenhang mit Zugriffstrukturen geklärt. Dabei wird für die Notwendigkeit von effizienten Indexstrukturen für RDF Daten argumentiert. Im ersten Abschnitt des Kapitels werden die Zugriffstrukturen für nicht-temporale RDF Daten vorgestellt. Der zweite Abschnitt beschäftigt sich mit temporalen Zugriffstrukturen und ihrer Klassifizierung. Dabei wird der Frage nachgegangen, wie tauglich die Strukturen für die Verwaltung von zeitabhängigen RDF Daten sind. Im letzten Abschnitt werden Indextechniken, die speziell für die Verwaltung von RDF Daten entwickelt wurden präsentiert.
Thema des vierten Kapitels ist die Implementierung der Indexstrukturen mit Hilfe der Java-Bibliothek XXL. Es wird unter anderem die in Rahmen der Diplomarbeit entwickelten und bestehenden Strukturen und Algorithmen beschrieben. Im letzten Abschnitt werden die Ergebnisse von Experimenten mit RDF Daten vorgestellt.

1. Resource Description Framework (RDF)

Der größte Traum in Semantik Web ist es, die Webdaten nicht nur dem Menschen sondern auch den Maschinen verständlich zu machen. In diesem Kapitel geht es zum einen um das Grundgerüst des semantischen Web RDF. Da der Fokus der Arbeit auf der Verwaltung von den RDF Daten liegt, wird es in dem Kapitel weniger auf die Applikationen aus dem Bereich Semantik Web, sondern vielmehr auf die Struktur von RDF eingegangen. Zuerst werden die Architektur des Frameworks vorgestellt. Die Serialiseirungsformate, die Möglichkeit die Daten in einem Relationalen Datenbank Management System (RDBMS) zu verwalten, verschiedene Architekturmodelle für RDF Verwaltung in RDBMS. Zusätzlich werden kurz die Anfragesprachen besprochen.

1.1. RDF

RDF ermöglicht Aussagen über Ressourcen zu treffen unter anderem Attribute zu zuordnen und die Beziehungen zwischen den Ressourcen zu beschreiben. Ressource Description Framework besteht aus zwei Komponenten. Die erste ermöglicht salopp ausgedrückt „etwas über etwas auszusagen“[4]. Die Zweite beschreibt die Beziehungen und Typen von Ressourcen.
Eines Ziele bei der Entwicklung des RDF waren die Bereitstellung eines einfachen Datenmodells und einfache Semantik mit beweisbaren Deduktionsregeln. Als Datenmodell für RDF wird ein simples Graphmodell mit dem Grundgerüst RDF Tripel *(Subjekt, Prädikat, Objekt)* kurz *(S,P,O)* verwendet. Die Menge von RDF Tripel wird ein RDF Graph genannt. Wobei die Knoten des Graphen sind die Subjekte und Objekte. Die Prädikate stellen die Kanten des Graphen dar. Strukturell ist ein RDF Graph ein direkte beschriftete Graph(vgl. Abbildung Einer der zentralen syntaktischen Bausteine des RDF Graphmodels ist Uniform Resource Identifier (URI) Konzept. Das Framework erweitert und verwendet dieses Konzept zur eindeutigen Beschreibung der Ressourcen. Weitere RDF Konstrukte sind die *Literale* und *leere Knoten*. Das *Literal* ist eine Zeichenkette mit zusätzlichen Information wie die Lokalisationssprache *plain Literal* oder der Datentyp *getypter Literal*. Aus einer praktische Perspektive kennt RDF nur einen Datentyp und zwar Zeichenketten, alle anderen Datentypen werden durch den *getypter Literal* kodiert. Die *leeren Knoten* sind die Graphknoten mit bezogen auf einen bestimmten Graph lokalen Indentifikatoren. Zusammengeffast hat RDF Tripel folgende syntaktische Struktur:

Definition 1.1.1 (RDF Tripel) *Ein RDF Tripel (S,P,O) ist eine Aussage oder Fakt mit folgende Aufbau:*

- *Subjekt S ist entweder ein URI oder ein leerer Knoten.*
- *Prädikat P ist ein URI.*
- *Objekt O ist entweder ein URI oder ein Literal oder ein leerer Knoten.*

Ein RDF Graph ist eine Mengen von RDF Tripel.

Mit dieser Organisation lassen sich beliebige Aussagen und Beziehungen zwischen denen erstellen. Ursprünglich wurde RDF für Beschreibung von Web-Ressourcen entwickelt. Das Ziel

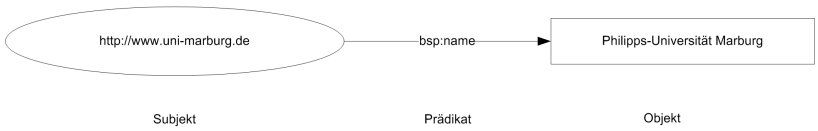

Abbildung 1.1.: Graphisch wird ein RDF Tripel wie folgt repräsentiert. Die Subjekt- und Objekt-Ressourcen werden als Ovale, Prädikate als gerichtete Kannten und Literale als Rechtecke dargestellt.

ist es, ein einfaches Konzept zu Darstellung von Meta-Information anzubieten. Zunächst betrachten wir das Tripel Beispiel auf der Abbildung 1.1, diese entspricht der Aussage *Die Ressource „http:www.uni-marburg.de“ hat den Namen „Philipps-Universität Marburg“*. Was uns fehlt ist die Interpretation von diesem Tripel z.B. dass die Ressource eine Universität ist und Name ein Attribut von diesem Typ ist, sogar wissen wir nicht ob der *Name* eine Beziehung oder ein Attribut ist. Aus diesem Grund sind die Prädikate als URI definiert und können als Knoten von RDF Graphen verwendet werden, damit lassen sich die Aussagen über die Kanten der Graphen erstellen.

Für die Interpretation von Prädikaten müssen die Meta-Informationen bzw. das Schema verfügbar sein. Diese Aufgaben übernimmt in RDF das RDF Schema Vokabular (RDF/S). Mit dessen Hilfe können die Attribute und Beziehungen zwischen Ressourcen beschrieben werden. Das ermöglicht maschinelle Interpretation von RDF Daten. Von Vorteil ist auch, dass die Metadaten über Daten auch als RDF Graphen verwaltet werden. An dieser Stelle greift das URI Konzept.

Ähnlich wie in relationalen Datenbanken die Metatabellen, die Information über die Tabellen und Attributen mit Datentypen verwalten, werden in RDF die Schemainformationen (wie Beziehungen zwischen den Ressourcen) mit Hilfe des RDF Schema Vokabular repräsentiert. Im Allgemeinen ist *RDF/S Vokabular* ist definiert durch die Menge von reservierten Wörtern, die für die Beschreibung von „Eigenschaften“ und Beziehungen von und zwischen Ressourcen dienen. Das RDF/S Vokabular definiert Klassen und Eigenschaften für die Gruppen Ressourcen. Für die Modellierung von Beziehungen zwischen den Ressourcen bietet RDF vordefinierte Konzepte wie Klasse (eine Menge von Ressourcen), Instanz eine Klasse und Attributen von Klassen. Hier sind die wichtigsten Klassendefinitionen von RDF/S `rdfs:Resource`, `rdf:Property`, `rdfs:Literal`, `rdfs:Class`. Durch die `rdf:Property` werden die binären Beziehungen zwischen Subjekten und Objekten beschrieben. Die eingebauten Attribute sind z.B. `rdf:Type`, `rdfs:SubClassOf` und `rdf:subPropertyOf`. Das eine Ressource zu einer Klasse gehört kann z.B. wie folgt dargestellt werden:

```
[http://www.uni-marburg.de, rdf:type, bsp:Universität]
[bsp:name, rdf:type, rdf:Property]
```

Die Modellierung von Beziehungen mit RDF/S stellt die Grundlage für die semanitschen Analyse. Durch die Zugehörigkeit zu einer Klasse kann z.B. semantische Gleichheit nachgewiesen werden, die transitiven Beziehungen zwischen den Klassen der Beziehungen, erlauben Herleitung von neuen Informationen.

Wohl eine der interessanten funktionalen Besonderheiten von RDF ist die Möglichkeit Aussagen über die Tripel zu erstellen. Damit ermöglicht RDF die Metainformation über die RDF Aussagen in RDF Syntax darzustellen. Betrachten wir folgendes Szenario, angenommen wir wollen in unserem RDF System zusätzlich simple Auditfunktionalität anbieten (vgl. Abbildung 1.2). Diese Technik die Aussagen über die Aussagen zu machen wird in RDF als *Reification* bezeichnet. RDF bietet verschiedene Möglichkeiten die Aussagen über die Aussagen zu machen. Eine Option wäre das Reifications Vokabular von Framework zu benutzten

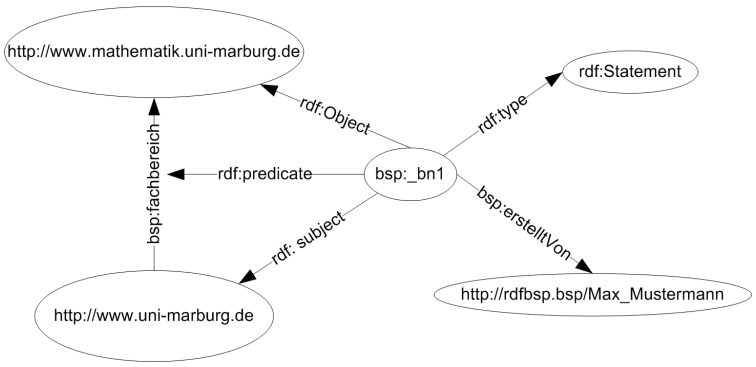

Abbildung 1.2.: Reification Beispiel. Der Beispielgraph zeigt, wie mit Hilfe von der Reification eine einfache Auditfunktionalität implementiert werden kann.

(`rdf:Statement`,`rdf:type`, `rdf:predicate`, `rdf:subject`, `rdf:object`). Die einfachste Methode die RDF Tripel über Tripel zu erstellen ist wie folgt:

```
[X, rdf:type, rdf:Statement]
[X, rdf:predicate, bsp:fachbereich]
[X, rdf:subject, http://www.uni-marburg.de]
[X, rdf:object, http://www.mathematik.uni-marburg.de]
```

Über den leeren Knoten `X` kann die Metainformation beschrieben werden. Die Flexibilität des RDFs erlaubt auch andere Darstellungsmöglichkeiten, es kann z.B. auch das eigene spezifische Vokabular zur Reifications Zwecken erstellt werden. Im nächsten Kapitel wird gezeigt wie dieses Konzept in Zusammenhang mit Zeitabhägigen RDF Daten benutzt werden kann. *Im Laufe der Arbeit werde ich folgendes fiktives Beispielsystem verwenden, die einer Wissensdatenbank über deutschen Hochschulen verwaltet. Information sind von jeweiligen Hochschulen als RDF Daten zu Verfügung gestellt. Auch Zugriff auf interne Datenbanken (Deep Web Datenbanken) wird bereitgestellt, neben dem Zugriff wird auch die Schemainformation von Deep Web Datenbanken RDF/S verfügbar gemacht. Die Daten werden dann mit Crawler gesammelt und zu RDF Datenbank hinzugefügt. Zusätzlich bietet das System Suchdienste und Interface für die Kopplung von anderen Diensten. Wissensdatenbank verwaltet unter anderem Informationen über die laufende Forschungsprojekte, Diplom- und Doktorarbeiten, wissenschaftliche Paper, Kooperationspartner etc.*
Es ist klar, dass die Daten für den Crawler als RDF Metainformationen bereitgestellt wurden. Eine interessante Frage in Zusammenhang mit unserer fiktiven RDF Datenbank ist die Organisation von der Backenddatenbank. Soll ein relationales Modell oder ein RDF für die Backend-Verwaltung verwendet werden? Warum sich die Mühe geben und die Daten mit RDF verwalten, wenn dieselbe Information kann auch in relationalen Modell dargestellt werden. In der Tat haben das RDF Graph Modell und das Relationale Modell viel gemeinsam. Die Beziehungen zwischen den Ressourcen oder die Informationen über Ressourcen können auch in relationalen Modell abgebildet werden. Es kann ein relationales Modell entworfen werden und ein Schema für die Hochschulen Informationsverwaltung für die relationale Datenbank angelegt werden. Das Modell ist aber sehr starr.
Die Daten für unser fiktives System kommen aus heterogener Web-Umgebung, mit stark variierenden Schemainformationen von einzelnen Hochschulen z.B. für die Daten, dieselbe Information ausdrücken, können verschieden Datentypen verwendet werden. Durch das Bereitstellen von RDF/S Dokumenten können Beziehungen analysiert werden und in unser System integriert werden. Außerdem ermöglichen wir auch die Semantische Analyse der Daten durch spezielle Dienste zu betreiben z.B. können neue Beziehungen entdeckt werden oder

das System kann präzise neue Forschungskooperationspartner finden. Hätten wir als Backend Modell ein relationales Modell verwendet, müssten wir kontinuierlich das Schema anpassen auf neue Datentypen eventuelle Konverter schreiben, neue Beziehungstabellen anlegen, neue Tabelle für neu entdeckte Konzepte. Auch die maschinelle Interpretation von diesen Daten ist schwieriger. Die Beziehungen - und Attributeninformationen von den Tabellen sind zwar auch in relationalen Modell verfügbar, müssen aber mühsam aus Metatabellen herausgezogen und durch Schlüssel- und Fremdschlüssel-Eigenschaften hergestellt werden, meistens sind diese Informationen unzureichend.
Betrachten wir die Tabelle, die das Konzept Hochschule darstellt und die Tabelle, die Fachbereiche verwaltet. Angenommen die beiden stehen durch Fremdschlüssel Eigenschaft in Parent-Child Beziehung zu einander. Ohne entsprechende Zusatzinformation kann nur die Information, dass die beiden Konzepte, die durch die Tabellen repräsentiert sind, miteinander in Beziehung stehen. Die Information, dass der Fachbereich die Einrichtung der jeweiligen Universität ist, ist nicht explizit verfügbar. Natürlich kann ein Beziehungstyp eingeführt werden, der dann diesen Sachverhalt beschreibt. Dann müssen über diesen Beziehungstyp wiederum Metainformationen zu Verfügung gestellt werden.
Es kann aber auch eine Hybrid Lösung implementiert werden, indem, vorausgesetzt, dass das Schema nicht so stark mit der Zeit variiert, Backend nach relationalen Modell zu organisieren. Für semantische Analyse der Daten können die Meta-Daten über die Relationalen Modell in RDF/S geschrieben werden, und über Abbildungsmodul für relationales und RDF Modell bereitgestellt werden.
Die Flexibilität, die uns RDF bietet, hat auch ihren Preis. Bei Verwendung von relationalem Modell für Backend-Datenbank können wir Normalisierung und anderen Techniken aus relationalen Datenbanken verwenden, das erlaubt ein Hochleistungsystem zu implementieren. Im Laufe der Arbeit wird aber auch gezeigt, wie die Performanz der RDF Datenbanken gesteigert werden kann.

RDF und XML

Ein weiteres Ziel bei der Entwicklung des RDF war die Verwendung des XML zur Austausch und maschinellen Interpretation von RDF Daten. Aus diesem Grund wird z.B. für *getypte Literale* das XMLLiteral Konzept verwendet. Die andere Schnittemenge zwischen der RDF und XML Welt ist Repräsentation von RDF Graphen mit Hilfe der XML Notation. Betrachten wir folgenden Beispiel.

Beispiel 1.1.1 [1] *Ein Auszug in XML Format aus unserer Universität Datenbank kann wie folgt serealisiert werden: „Phlipps-Universität Marburg hat Fachbereich Mathematik und Informatik. Der Fachbereich bietet zwei Fachrichtungen Mathematik und Informatik. “*

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:bsp="http://mathematik.uni-marburg.de/~achakeye/rdfbsp#">
<rdf:Description rdf:about="http://www.uni-marburg.de">
<bsp:name>Philipps-Universität Marburg</bsp:name>
<bsp:hatFachbereich rdf:resource = "http://www.mathematik.uni-marburg.de"/>
</rdf:Description>
 <rdf:Description rdf:about="http://www.mathematik.uni-marburg.de">
 <bsp:fachrichtung>
 <rdf:Bag>
 <rdf:li>Mathematik</rdf:li><rdf:li>Informatik</rdf:li>
```

[1] Mit Hilfe von RDF Validator http://www.w3.org/RDF/Validator/ können die RDF Dokumente online validiert werden. Dieser Dienst bietet auch die graphische Repräsentation von RDF XML Dokumenten

```
</rdf:Bag>
</bsp:fachrichtung>
</rdf:Description>
</rdf:RDF>
```

Neben der Möglichkeit die Daten mit Hilfe von XML darzustellen, bietet RDF die Option die Daten in einem *N-Tripel* Format zu repräsentieren.

```
...
<http://www.uni-marburg.de> <bsp:name> "Philipps-Universität Marburg".
...
```

1.1.1. Anfragesprachen

In vorigen Abschnitten haben wir kennen gelernt, wie die RDF Daten gespeichert und serialisert werden können. Wenden wir uns jetzt der Frage wie die gewünschte Information in RDF Datenbanken aufgesucht wird. Grundsätzlich gibt es verschiedene Möglichkeiten, z.B. unser Beispielsystem kann eine Programmierinterface anbieten. Das ermöglicht selbst einen Algorithmus zum Traversieren des RDF Graphen zu implementieren, vergleichbar mit dem Netzwerkmodell. Das wäre äußerst umständlich bei jeder Anfrage anzugeben, wie die gesuchten Daten berechnet werden können. Gewünscht ist ähnlich wie in relationalen Modell (Tupelkalkül, relationale Algebra[2]) oder XML (XQuery) eine deklarative Anfragesprache die unabhängig von der physikalischen Implementierung ist.
Nach der Definition des RDF Graphen als Menge von Tripel lässt sich ein Graph als eine Tabelle mit drei Attributen (S, P, O) darstellen. Aus diesem Grund können die Parallelen zu relationalen Modell hergestellt werden. In vorigen Abschnitt wurde in Zusammenhang mit RDF ein Graph Datenmodell als Datenmodell vom Framework angesprochen. Aus diesem Blickwinkel lassen sich auch die Konzepte der Anfragesprachen aus der relationalen Datenbanktheorie auf RDF übertragen. Würde diese Tabelle in einer Relationale Datenbank verwaltet, könnten schon bestimmte Anfragen wie Selektion nach einem Tripel und Projektion auf Attribute durchgeführt werden Z.B. $(?, P, ?)$ sind alle Tripel die ein bestimmtes Prädikat enthalten. Wir können auch mehr, durch relationale Joins und Tupelvariablen können Graphpfade berechneten werden z.B. $(?, P_1, x), (x, P_2, ?)$. In Zusammenhang mit dem Graphdatenmodell können diese Anfragen als Anfragegraphen aufgefasst werden. Allgemein können wir es als die Suche nach allen Teilgraphen im RDF Graphen, die dem Anfragegraphen matchen, betrachten. Dies wird im RDF als Graphmatching bezeichnet. Schon mit drei Operatoren der Relationalen Algebra wie Selektion, Projektion und Join kann die Graphmatching-Funktionalität implementiert werden. Diese stellen auch die theoretische Grundlage für die SPARQL[3] Anfragesprache für RDF dar. Die Sprachsyntax ist in Anlehnung an SQL entwickelt worden. So würde eine Anfrage in SPARQL aussehen, die nach Namen von Projekten, die am Fachbereich Mathematik und Informatik laufen, anfragt.

```
SELECT ?Name
WHERE { ?x  rdf:Type bsp:Projekt.
'http://www.mathematik.uni-marburg.de' bsp:hasProjekt ?x.
?x  bsp:Name ?Name. }
```

Ein Großteil von Anfragen aus der praktischen Perspektive kann schon mit SPARQL beantwortet werden. Die Ausdruckstärke der Sprache ist aber beschränkt auf SPJ Anfragen.

[2] die Sprache ist nicht ganz deklarativ
[3] seit 2008 ist SPARQL als Standard Anfragesprache für RDF Daten von W3C festgelegt

SPARQL kann aber nicht mit dem Graphmodell verbundenen Anfragen wie Erreichbarkeitsfrage zwischen zwei Ressourcen beantworten. Diese Art von Anfragen lassen sich nicht mit SPJ Anfragen ausdrücken, da in diesem Fall die Rekursion fehlt. In SPARQL lassen sich nur die einfache Pfad Ausdrücke berechnen, bei denen die Länge des Pfades vorab bekannt ist. Was gewünscht sind, nicht nur Matching- sondern auch Navigationanfargen ähnlich zu XML Anfragesprachen wie XPath und XQuery. Im Zuge der Entwicklung von RDF wurden auch andere Anfragesprachen entwickelt, die auch allgemeine Pfadausdrücke berücksichtigen. Die SPARQL hat sich aber durchgesetzt, ausschlaggebend war dabei die Einfachheit und die gewisse Ähnlichkeit zu SQL. Die typischen Graphanfragen, bei denen der Pfad zwischen Ressource a und b abgefragt wird, oder allgemeine Pfadausdrücke, die z.B. nach dem Pfad von Ressource a nach b suchen, bei dem das Prädikat p_1 nur einmal und p_2 mehrmals vorkommt, können als spezielle Operatoren implementiert werden und SPARQL anreichern. Ein gute Übersicht über die RDF Sprachen bieten folgende Arbeiten [6][7][9]. Eine kurze Einführung in SPARQL bietet [8].

1.2. RDF Datenbanken

Zwei mögliche Serialisierungsformate haben wir in diesem Abschnitt kennen gelernt. Eine Idee für die Verwaltung von RDF Daten ist die Verwendung von XML oder serialisierten N-Tripel Dateien, dabei gibt es ähnlich der Argumentation bei der Verwaltung von relationalen Daten, viele offensichtliche Nachteile. Durch Querverbindung zum relationalen Modell ist es von Vorteil die RDF Daten in einem RDBMS zu verwalten. Dafür spricht nicht nur die Ähnlichkeit von Modellen, sondern auch die praktischen Gründe, wie die Verwaltung von großen Datenmengen, Datenmanipulation mit SQL, Nebenläufigkeit- und Wiederherstellungs- Komponenten, die von den RDBMS angeboten werden. Die Anwendungen für komplexere Semantische Manipulation von RDF Daten können auf den relationalen Datenbanken aufgebaut werden und auf Applikationseebene die benötigte Daten effizient traveresieren. Zudem lässt sich die standard RDF Anfragesprache SPARQL in SQL übersetzten.
Die Tatsache dass die RDF Graph eine Menge von RDF Tripel ist, ermöglicht den RDF Graphen direkt die relationale Tabelle zuzuordnen. In der einfachsten Variante könnte eine Tabelle mit Schema *Subjekt, Prädikat, Objekt* mit einem RDF Graphen assoziiert werden:

Beispiel 1.2.1 [4]

```
CREATE TABLE TRIPLE_TABLE(
RDF_Subject TEXT,
RDF_Predicate TEXT,
RDF_Object TEXT
);
```

Einfach heißt nicht unbedingt, dass diese Abbildung schlecht ist. Diese Art der Verwaltung wird auch als *Tripletable* bezeichnet. Im Laufe der Arbeit wird auch gezeigt, dass diese Möglichkeit der Verwaltung von RDF Daten in RDBMS in Bezug auf Effizienz eine gute Wahl ist. Zusätzlich zur Verbesserung der Speicherplatzausnutzung kann eine Mapping Tabelle für Präfixe erstellt werden, die der Übersetzung von Namensräumen dienen kann. Die *Tripletable* kann auch normalisiert werden, indem eine Hauptabelle für die Graphstruktur und drei Tabellen für das Subjekt, Prädikat und Objekt verwaltet werden, ähnlich dem Star-Schema. Die komplette Normalisierung hält zwar die Konzepte getrennt, bei überwiegend

[4] Dabei wurde SQL Syntax von PostgreSQL verwendet. Der TEXT Datentyp steht für die variablen langen Zeichenketten

lesenden Zugriffen werden dann die Joins berechnet, um die Tripel darzustellen (die Effizienz von diesem Modell kann aber ähnlich dem Datawarehouse Ansatz durch die Vorberechnung der Joinindexe gesteigert werden).
Eine andere Normalisierung wäre neben der Haupttabelle die Tabelle mit Ressourcen und Tabelle mit Literalen zu verwalten. In vielen RDBMS Systemen sind die Datentypen für Zeichenketten begrenzt auf eine bestimmte Länge, lange Zeichenketten insbesondere Literale können nicht direkt verwaltet werden, dafür gibt es spezielle Datentypen wie CLOB. In [10] z.B. wird eine Hybrid-Lösung vorgeschlagen, neben der Tripletable werde zwei Tabellen mit Datentypen für die Verwaltung von langen Zeichenketten bereitgestellt, eine für Ressourcen und eine für langen Literalen, Falls der Spaltendatentyp die Zeichenkette nicht aufnehmen kann, wird diese in die Zusatztabelle ausgelagert. Zusätzlich, um die Geschwindigkeit von Anfragen zu steigern werden in [10][11] so genannte Property Tabellen eingeführt in diesen werden die Prädikatwerte, die tendieren häufig mit Subjekten aufzutreten und auf gewisse Weise die zu clustern, in eine Zusatztabelle ausgelagert. Diese Prädiaktenwerte werden dann als Attribute der Tabelle dienen. (Z.B. hätten wir die Prädikatenwerte wie *Name, istImProjekt, Geburtsdatum*, dann könnte die Tabelle mit folgendem Schema zusätzlich angelegt werden *(Subjekt, Name, istImProjekt , Geburtsdatum)*). Diese Erweiterung kann die Anzahl der Join Operationen erheblich reduzieren z.B. bei der Anfrage *„In welchem Jahr ist Max Mustermann geboren, der im Projekt „http:www.rdfbsp.bsp/SemanitkWeb“ arbeitet? “*[13]. Die Anfrage lässt sich in SQL mit zwei Subjekt-Subjekt Joins berechnen:

```
SELECT  C.RDF_Object
FROM    Triple_Table AS A, Triple_Table AS B, Triple_Table AS C
WHERE   A.RDF_Subject = B.RDF_Subject AND B.RDF_Subject = C.RDF_Subject AND
A.RDF_Predicate = 'Name' AND A.RDF_Object = 'Max Mustermann' AND
B.RDF_Predicate = 'istImProjekt' AND
B.RDF_Object = 'http:www.rdfbsp.bsp/SemanitkWeb' AND
C.RDF_Predicate = 'Geburtsdatum'
```

Mit Hilfe der Property Tabelle dagegen würde man nur eine Selektion benötigen.
In [13] wurde eine Besonderheit bei der RDF Daten in TripleTable beobachtet. Häufig ist in vielen RDF Graphen die Anzahl von Beziehungstypen (Prädikaten) im Vergleich zu Subjekten und Objekten nicht so groß. Damit ist die Selektivität des P Attributes vergleichsweise klein. Diese Beobachtung führte dazu, dass die Daten *vertikal* nach P Attributen partitioniert wurden. Die Idee basiert auch auf den oben beschriebenen Property Tabellen, nur ist diese Lösung viel flexibler. Bei der vertikalen Partitionierung werden pro Prädikatwert eine zweispaltige Tabelle mit führender Spalte für Subjekte und der zweiten Spalte für Objekte erzeugt(Hätten wir wie in unserem kleinerem Beispiel nur die drei Prädikatenwerte *Name, istImProjekt, Geburtsdatum*, könnten drei Tabellen erzeugt werden). Zwar wird die Anzahl der Joins nicht so stark reduziert wie bei der Property Tabelle Technik, die sind aber sehr effizient ausführbar.
Eine zusätzliche Effizienzverbesserung bringt die Verwaltung von zweispaltigen Tabellen in einem Spaltenorientierten RDBMS [13][12][5]. Im Allgemeinen speichern die Spaltenorientierten RDBMS die Daten als zweikomponentige Tupel *(Id, Wert zum Spalte-Attribut)*, damit werden die Tabellen vertikal zerlegt. In „normalen“ RDBMS werden Datenbanktupel reihenweise in Speicherseiten des Externenspeicher verwaltet. Beim Lesen von Daten, bei denen nur ein Teil der Attribute benötigt wird (Projektion), wird das ganze Tupel geladen. Bei der spaltenorineiterten Speicherung werden nur die benötigten Attribute geladen. Das erhöht den Durchsatz und senkt auch die CPU Kosten bei einer späteren Projektion. Allerdings

[5] Nähe zur Architektur von Spaltenorientierten RDBMS ist in [51] zu finden

wenn eine Teilmenge von Attributen gefragt wird, werden natürlich die Joins benötigt. In diesen Systemen werden aber diverse Optimierungen von Joins vorgenommen, z.B. es werden vorab die Join-Indexe für eventuellen Teilmengen von Attributen berechnet. Zudem sind die internen Spaltentabellen meistens als Indexe organisiert, und die Daten liegen sortiert vor. Das ermöglicht den direkten Einsatz von Sort-Merge-Join Algorthmen. Die spaltenorientierten Systeme sind primär aber für die Daten gedacht, bei denen die Lesezugriffe überwiegend sind, da bei häufigen Einfügen- und Update-Operationen alle internen zweispaltigen Tabellen und Joinindexe modifiziert werden müssen.
Bei der *Vertikalen Partitionierungstechnik* für RDF Daten liegen die Tabellen für Prädikate schon als zweispaltige Tabellen vor. In der Praxis zeigen die RDF Daten auch am häufigsten ein statisches Verhalten. Damit können die Daten in Batch-Modus geladen werden. In [13][12] wird experimentell gezeigt, dass diese Art der Verwaltung der statischen RDF Daten eine der effizientesten ist. Die als erste beschriebene Form der Verwaltung als TripleTable kann aber genau so effizient, ohne Einsatz der spaltenorientierten RDBMS, sein, wenn die Daten richtig indexiert sind[23]. Zudem wurden als Fazit in allen drei Arbeiten experimentelle Vergleiche mit RDF Systemen gemacht, die RDF Daten anders als RDBMS verwalten. Alle diese Systeme haben eine schlechtere Performanz gezeigt als hier angesprochene Verwaltungstechniken.

1.3. Zusammenfassung

Seit der Erfindung von RDF hat sich das Framework in dem Zielbereich Web nicht so richtig durchgesetzt. Es gibt aber Anwendungsgebiete bei, denen das RDF aktuell eine große Rolle spielt. In P2P Netzwerken wird zum Beispiel RDF verwendet, um einzelne Knoten einer P2P Netzwerk zu beschreiben. Das ermöglicht eine automatische Suche nach Netzwerknoten, die eine interessante Information, Dienst anbieten. In Biologie wird RDF als Annotationstechnik für Beschreibung von Beziehungen zwischen z.B. Proteinen, biologischen Arten etc. eingesetzt. Für die Abbildung von sozialen Netzwerken wird auch RDF eingesetzt. Unter anderem dient RDF als Basis für Ontologien Entwicklung und Ontology Web Language (OWL). Ganz andere Anwendung findet RDF im Bereitstellen von Metainformation für Multimedia Objekte z.B. mittlerweile besitzen viele digitale Kameras eine Funktionalität, die erlaubt automatisch die Metainformation in RDF mit geschossenen Bilder zu assoziieren, z.B. Ort und Zeit Angaben.
In diesem Kapitel wurde weniger auf das Semantik-Web eingegangen. RDF wurde eher aus der technischen Sicht präsentiert. Eine vertiefende Diskussion über Semantik Web wird in[4][3] gut beschrieben. Wir haben die syntaktische Struktur von RDF kennengelernt, die Möglichkeit der semantischen Analyse von Daten mit RDF/S. Die theoretischen Grundlagen von RDF sind in [2] [5] zu finden. Aus der Datenbanksicht bietet RDF ein alternatives Datenmodell für die Verwaltung von Daten. Besondere Vorteile vom Framework sind einfacher syntaktischer Aufbau, wohldefinierte Semantik, Domainunabhägigkeit.
Als Fazit lassen sich RDF Daten mit Hilfe eines RDBMS verwalten. Die Standardanfragesprache SPARQL basiert auf SPJ Anfragen, damit lässen sich die Anfragen in SQL übersetzen. Im nächsten Kapitel geht es um die Erweiterung des RDF um die Zeitdimension. Dabei wird versucht zu zeigen, wie die Techniken aus dem verwandten relationalen Modell auf RDF angewendet werden können.

2. Zeitdimension in RDF

In diesem Kapitel werden Grundbegriffe und die wichtigsten Konzepte aus dem Bereich temporaler Datenbaken erläutert. Die RDF-Tripel stellen ähnlich dem relationalem Modell Fakten und Aussagen über die modellierte Welt dar. Die Aussagen können auch explizit Zeitinformation enthalten, z.B. Information über die zeitliche Gültigkeit eines Faktes, die durch einen Tripel oder durch einen Subgraphen repräsentiert ist. RDF/S erlaubt die Vokabulare flexibel zu schreiben, so dass die Zeitkomponente von „intelligenten“ Maschinen interpretiert und weiterverarbeitet werden kann. Zunächst betrachten wir folgendes Beispiel:

Beispiel 2.0.1 *Folgende RDF XML Darstellung beschreibt die Information, dass Max Mustermann im Projekt „Semantic Web“ von Jahr 2000 bis Jahr 2006 gearbeitet hat*[1]*.*

```
...
<rdf:Description rdf:about="http://rdfbsp.bsp/Max_Mustermann">
<bsp:position>
<bsp:period>
<time:from rdf:nodeID="date2000-01-01"/>
<time:to   rdf:nodeID="date2006-01-01"/>
<bsp:projekt rdf:resource = "http://rdfbsp.bsp/databases/SymanticWeb"/>
</bsp:period>
</bsp:position>
</rdf:Description>
<rdf:Description rdf:nodeID="date2000-01-01">
    <time:at rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
     2000-01-01</time:at>
</rdf:Description>
<rdf:Description rdf:nodeID="date2006-01-01">
    <time:at rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
     2006-01-01</time:at>
</rdf:Description>
...
```

Im Beispiel wird explizit zeitliche Information verwaltet. Dafür wird ein Zeitvokabular definiert, mit dessen Hilfe der Internetdienst gezielt nach Zeitkomponenten suchen und sie verarbeiten kann. Eine ganz andere Sicht auf die Zeit vermittelt das nächste Beispiel 2.0.2.

Beispiel 2.0.2 *Betrachten wir folgendes Szenario (Abbildung 2.1): Angenommen nach der Einführung des neuen Content-Management-Systems an der Universität Marburg wurde die Webaddresse des Fachbereichs Mathematik und Informatik geändert. Sämtliche Änderungen wurden von Hochschulrechenzentrum der Uni Marburg an unsere Beispiel-RDF-Datenbank geschickt. Nach einer Revision der Seiten-Struktur wurden die Seiten von abgeschlossen Projekten als gelöscht markiert und nicht mehr verlinkt. Nach einiger Zeit wollte eine Arbeitsgruppe von der Universität „N“ wissen, ob es auch an dem Fachbereich Informatik an der Uni-Marburg Projekte in Bereich Semantik Web gaben. Leider war es durch die propagierten Änderungen unmöglich diese Information zu finden. Um die Änderungen verfolgen zu können wurde dann entschieden auch alte Zustände der Wissensdatenbank zu speichern. Jede*

[1] Bei der Erstellung von diesem Beispiel wurde folgende Quelle http://www.govtrack.us/source.xpd verwendet. Namensraum für time: http://pervasive.semanticweb.org/ont/2004/06/time#

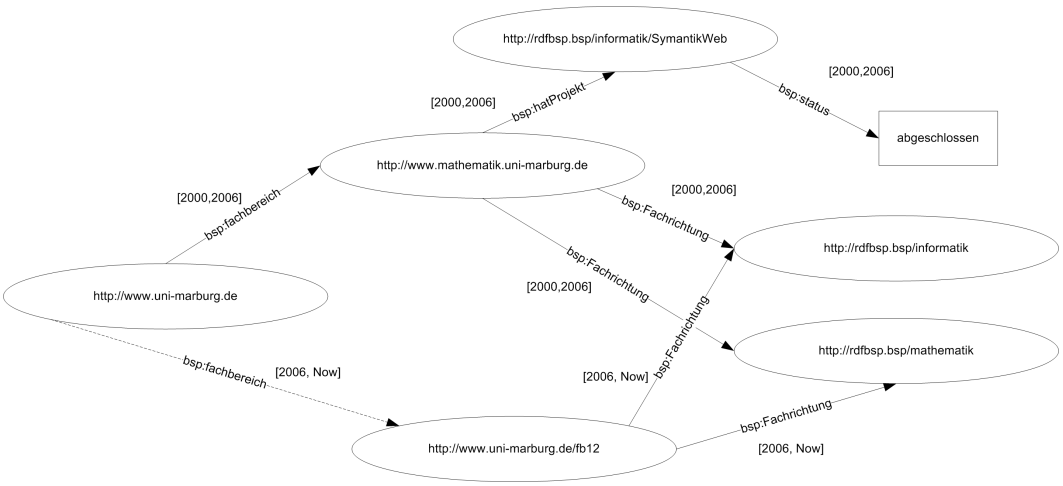

Abbildung 2.1.: Zeitliche Änderungen eines RDF Graphen.

Änderungsoperation wie Einfüge-, Lösch und Updateoperation waren mit einem Zeitstempel assoziiert, mit dieser Architektur wäre es möglich zu jeder Zeit vergangene Zustände zu traversieren.

Aus beiden Bespielen geht hervor, dass die Interpretation der Zeit in Zusammenhang mit Datenverwaltung sehr unterschiedlich sein kann. Diese Unterschiede werden im Laufe des Kapitels erläutert. Unter anderem werden das Modell zur Zeitrepräsentation vorgestellt, und die Frage, wie sich die unterschiedliche Semantik der Zeit auf Anfragen auswirkt, geklärt.
Als erstes werden Zeitbegriffe und formale Modelle aus dem relationalen Modell präsentiert. Danach wird gezeigt, wie diese Begriffe sich auf RDF zeitabhängige Daten übertragen lassen.

2.1. Temporale Modelle in relationalen Datenbanken

Dieses Unterkapitel beschäftigt sich mit der Frage, wie sich das relationale Modell um eine Zeitdimension erweitern lässt. Im weiteren Verlauf der Arbeit werden wir sehen, dass die vorgestellten Konzepte sich auch auf RDF zeitabhängige Daten anwenden lassen.
Formal können wir die Zeit als geordnete Menge über eine Domain von Elementen (Zeiteinheiten) interpretieren. Aus der theoretischen Perspektive spielt die Granularität von der Menge keine große Rolle. Zusammengefasst können wir die Zeit als folgende Struktur $(T, <)$ temporale Domain definieren:

Definition 2.1.1 (Temporale Domain) *Eine temporale Domain ist eine Struktur $T_d = (T, <)$, mit T als Trägermenge von Zeiteinheiten und linearer Ordnung $<$ auf T.*

Als Trägermenge können z.B. natürliche oder ganze Zahlen N,Z fungieren.
Nachdem T_d formal definiert ist, kann die temporale Datenbank D_t definiert werden. Das relationale Datenbankmodell lässt sich im Allgemeinen in verschiedener Weise mit der Zeitdimension erweitern. Dazu betrachten wir zunächst zwei unterschiedliche, aber von der Ausdruckstärke äquivalente abstrakte temporale Modelle. Die erste Möglichkeit formal die temporale Datenbank zu beschreiben ist eine Funktion zu definieren, die als Eingabe Elemente aus T erhält und als Ausgabe die relationale Datenbank, mit zu diesem Zeitpunkt „gültigen“ Daten, produziert. Die zweite Möglichkeit ist die Einführung von zusätzlichen temporalen Attributen. Im Folgenden werden beide Optionen formaler beschrieben:

- Beim *Snapshot Modell* werden einzelne *Snapshots*(Inhalt der Tabellen zu einem Zeitpunkt) der Datenbank, als Ausgabe eine Funktion von $T \rightarrow DB(D, \rho)$ betrachtet, wobei $DB(D, \rho)$ eine Datenbank über Datendomain D mit Schema $\rho = \{r_1 \dots r_n\}$ ist. Der Datentyp von Relationen bei diesem Modell hat dann folgende Signatur $T \rightarrow (D^n \rightarrow bool)$.
- *Timestamp Modell*: Bei diesem Modell werden Relationen (Tabellen) um ein temporales Attribut erweitert und haben somit folgende Typsignatur $T \times D^n \rightarrow bool$.

Sowohl aus theoretischen als auch aus praktischen Gründen hat das *Timestamp* Modell mehrere Vorteile. In der Praxis können relationalen Tabellen direkt durch Aufnahme von Zeitattributen auf temporale Relationen erweitert werden. Ein anderer Vorteil ist z.B., die dass die Anfragen nach allen Zeitpunkten fragen, bei denen die Bedingung φ erfüllt ist, von der Komplexität sich leichter beantworten lassen, als bei der Umsetzung des *Snapshot-Modells*.

$$\{t : \ DB \models \varphi(t)\}$$

Aus diesen Gründen hat sich das *Timestamp* Modell in relationalen Datenbanken durchgesetzt. Allerdings sind die beiden Modelle von Ausdruckstärke äquivalent.

2.1.1. Zeitsemantik

Die Zeitkomponente im temporalen Modell kann unterschiedliche Interpretation (Semantik) besitzen, z.B. (vgl. Bsp. 2.0.1, 2.0.2) ist t in der Relation ein Zeitpunkt des Einfügens eines Tupel oder es ist die Zeit, in der ein Tupel in der modellierten Welt gültig ist. Im folgenden werden die wichtigen Konzepte der temporalen Datenbanken hinsichtlich der Bedeutung der Zeitkomponente vorgestellt.
Snapshot Datenbanken sind die Datenbanken, die nur den aktuellen Zustand verwalten (diese werden voll von modernen DBMS unterstützt (Anfragesprachen, Integritätsbedingungen, Datenstrukturen, Indexstrukturen etc.). Dabei zeichnet sich ein Zustand einer relationalen Datenbank durch den aktuellen Inhalt seiner Tabellen. Datenmanipulierende Operationen einer Datenbank (z.B. UPDATE, INSERT oder DELETE SQL Befehle) erlauben den aktuellen Inhalt zu verändern, gegebenenfalls zu korrigieren und dem Modell anzupassen. Durch diese Operationen ändert sich der Zustand der Datenbank, wobei der vorhergehende Zustand verworfen(„vergessen“) wird.
In der Praxis gibt es eine Reihe von Anwendungen (z.B. Archivierungssysteme), in denen nicht nur der aktuelle Zustand einer Datenbank, sondern auch die vorhergehenden Zustände (Versionen) von Interesse (vgl. Bsp. 2.0.2) sind. Die Evaluation von Zuständen einer Datenbank über die Zeit wird auch als *Historie* bezeichnet. Wird dieser Begriff mit dem Snapshot-Modell verglichen, so stellt man fest, dass sie sehr viel gemeinsam haben. Die Datenbanken,die die Historie verwalten, werden als *Transaktionszeit-Datenbanken* definiert.
In dieser Art von Datenbanken ist es möglich deren Versionen zu jedem Zeitpunkt in der Vergangenheit aufzurufen und Anfragen auf diese Version zu stellen. In diesen Anwendungen ist die Zeit durch so genannte *Transaktionszeit* ausgedrückt, das ist die Zeit, die den Zeitpunkt einer datenmanipulierenden Operation mitprotokolliert. Diese Operationen, die unter anderem zur Korrektur von Daten dienen, operieren auf dem aktuellsten Zustand, und erzeugen somit einen neuen Zustand.
Dem Transaktionszeit-Begriff steht die *Validzeit* gegenüber. Dieser Zeitbegriff beschreibt die Gültigkeit eines Faktes in der modellierten Welt, der durch Datenbanktupel beschrieben ist. Die Schwierigkeit der Verwaltung von Validzeitdaten liegt in der Gewährleistung der Korrektheit der Daten bezüglich ihrer Gültigkeit in der modellierten Welt. In diesen Datenbanken wird stets angestrebt den aktuellen korrekten Zustand zu gewährleisten. Dadurch ist die Semantik von solchen Datenbanken viel komplexer als bei Snapshot-Datenbanken, z.B. in

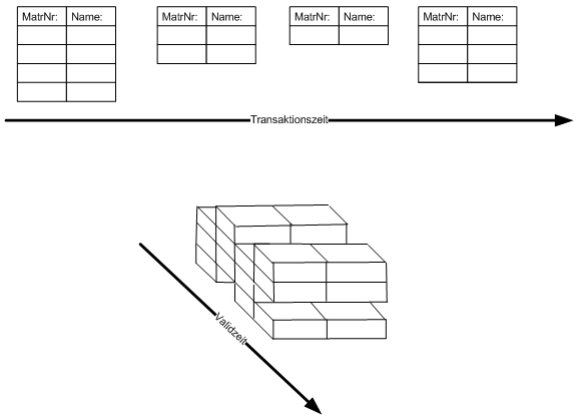

Abbildung 2.2.: Die Abbildung veranschaulicht die Zeitbegriffe in relationalen Datenbanken. Quelle [15]

Hinsicht auf Anfragen und Gewährleistung von Integritätsbedingungen. Bei der Überführung von einem konsistenten Zustand einer Validzeit-Datenbank in eien neuen Zustand werden alte Zustände (gegebenenfalls auch fehlerhafte Versionen) verworfen und nicht mitgespeichert. Würde man auch diese Zustande mitverwalten, spricht man dann auch von *Bitemporalen* Datenbanken.
Es gibt auch eine dritte Form der Zeit in Datenbanken, eine so genannte *userdefinierte Zeit*. Sie gilt, um temporale Aspekte von Daten zu beschreiben, die nicht durch oben genannte Zeitdefinitionen abgedeckt sind. Zum Beispiel: Geburtstag-Information von Mitarbeitern der Mitarbeiter-Relation. Diese Zeit ist zwar eng mit der Validzeit verwandt, wird aber ohne große Schwierigkeiten von modernen DBMS unterstützt, da nur Datentypen für interne Repräsentation und Ein-Ausgabe Funktionen für diese Datentypen definiert werden müssen.

2.1.2. Darstellung von Zeit

Ein sehr wichtiger Aspekt, der sehr starken Einfluss auf die praktische Umsetzung hat, ist die Zeitdarstellung. Es gibt drei fundamentale Datentypen für die Zeitdarstellung[16]:

- Ein einzelner Zeitpunkt. Der drückt z.B. das ein Ereignis zum Zeitpunkt t stattgefunden hat.
- Die Dauer eines Faktes oder Ereignisses z.B. drei Tage, acht Stunden.
- Ein Zeitintervall ist ein zu einem Zeitpunkt gebundene Zeitraum.

In der Praxis wird im Allgemeinen eine diskrete Domain für die Zeiteinheiten verwendet[2]
Ein interessante Fragestellung ist, wie die Zeitintervalle interpretiert werden, als Menge von Punkten eingeschlossen zwischen Start- und Endpunkt oder als ein eigenständige Objekte. Grundsätzlich gibt es zwei unterschiedliche Repräsentationens-Modelle in Datenbanken *Punkt*- und *Intervall*-Modell [19] [16]. Je nach Deutung, werden auch unterschiedliche Ergebnisse bei der Anfrageverarbeitung produziert. Das folgende Beispiel erklärt die Unterschiede von den beiden Interpretationen [19]:

[2]Seit SQL 92 Standard bietet SQL spezielle Datentypen und Operationen für Zeitdarstellung: DATE, TIME, TIMESTAMP, und TIMEINTERVAL. SQL 99 zusätzlich bietet den PERIOD Datentyp. Wobei der TIMEINTERVAL entspricht der Dauer, und PERIOD dem Zeitintervall in dieser Arbeit verwendete Konvention.

r1:	ID:	Interval
	pn42	$[20, 30]$
	pn42	$[31, 40]$
	pn42	$[41, 50]$

r2:	ID:	Interval
	pn42	$[26, 32]$

Im Beispiel werden auf zwei Tabellen $r1$ und $r2$ Differenzoperator angewendet, je nach definierte Semantik erhält man verschiedene Ausgabe. Die werden der Reihe nach besprochen:

R1:	ID:	Intervall
	pn42	$[20, 25]$
	pn42	$[33, 50]$

R2:	ID:	Interval
	pn42	$[20, 30]$
	pn42	$[31, 40]$
	pn42	$[41, 50]$

R3:	ID:	Interval
	pn42	$[20, 25]$
	pn42	$[33, 40]$
	pn42	$[41, 50]$

Bei der ersten Ergebnistabelle, die Semantik der Anfrage kann als Berechnung von allen Einträgen interpretiert werden, die gültig in den Zeitpunkten sind, die nicht in der Tabelle $r2$ vorhanden sind. Diese Interpretation entspricht dem Punktmodell. Beim genauen Betrachten, stellt man fest, dass bei der Ausgabetabelle zwei $[33, 40]$ und $[41, 50]$ Intervalle zusammengeschmolzen wurden. Das Prozess der Vereinigung von aufeinanderfolgenden oder überlappenden Intervallen von Tupeln mit den gleichem Schlüssel wird *Colescing* genannt.
Die Tabelle $R2$ ist auf ersten Blick äquivalent zum Ergebnis einer normalen relationalen Differenz. Das kommt daher, dass die Intervalle in diesem Fall, als atomare Werte betrachtet werden. Diese Semantik von Intervallen gehört zum Intervall-Modell.
Die Ausgabe-Tabelle $R3$ besitzt die selbe Semantik wie die erste Ausgabe, der Unterschied ist, dass der *Colescing*-Schritt ausgelassen wurde. Wird die Funktion, die einzelne Snapshots der Datenbank zu den Zeitpunkten liefert, analog dem Snapshot-Modell auf die Tabellen angewendet, so stell man fest, dass die beiden Ausgabetabellen die selbe Menge von Daten zu jedem Zeitpunkt besitzen. Diese semantische Gleichheit bezüglich allen Snapshots wird als *Snapshot-Äquivalenz* bezeichnet. Je nach Interpretationen müssen die speziellen Operatoren entwickelt werden, um die gewünschte Ausgabe zu erhalten.
Eine weitere Fragestellung ist die Darstellung von Intervallen in Punkt- und Interavall-Modell. In der Praxis werden Zeitintervalle entweder als $[x_s, x_e]$ oder $[\, x_s, x_e)$ dargestellt, wobei letztere Darstellung kleine Vorteile bringt. Betrachten wir das folgende Beispiel[16]. Angenommen wie wollen wissen, ob Intervall I_1 den Intervall I_2 schneidet, bei der Darstellung $I_1 = [x_s, x_e]$, $I_2 = [y_s, y_e]$ es ist genau dann der Fall, wenn folgende Bedingung erfüllt ist:

$$(x_s \leq y_e \wedge y_s \leq x_e)$$

bei Halb-offene Repräsentation gehört zwar der Endpunkt nicht zum Intervall, dafür wird aber $\leq$ Operator durch $<$ ersetzt, dann gilt für $I_1 = [\, x_s, x_e)$, $I_2 = [\, y_s, y_e)$

$$(x_s < y_e \wedge y_s < x_e)$$

Aus praktischen Gründen wird die Halb-offene Darstellung bevorzugt. Im Unterschied zu einzelnen Zeitpunkten, sind Zeitintervalle nur partiell geordnet. Dennoch können die Beziehungen auf Intervallen betrachtet werden z.B. Überlappung, liegt erste Intervall vor dem

zweiten etc. Insgesamt stehen die Intervalle in dreizehn verschiedenen Beziehungen zu einander.

2.2. Temporal RDF

Nachdem grundsätzliche Begriffe aus temporalen Datenbanken erläutert wurden, wird in diesem Unterkapitel kurz ein formales Modell zur Anreicherung von RDFs Frameworks mit Zeitsemantik präsentiert. Das Unterkapitel basiert hauptsächlich auf der Arbeit von [20]. In der Arbeit haben die Autoren die formalen Grundlagen aus der temporalen Erweiterung der relationalen Datenbanken für die formale Definition von temporalem RDF verwendet. Es gibt auch eine Reihe von anderen Forschungsarbeiten auf diesem Gebiet[21][22]. In [20] wird das Timestamp-Modell auf RDF angewendet.
Ähnlich wie bei der Einführung der Zeitdimension in relationalen Datenbanken können in RDF Framework zwei Möglichkeiten in Betracht gezogen werden. Bei der Verwendung des *Snapshots-Modells* würden die Zustände des RDF Graphen zu den jeweiligen Zeitpunkten gespeichert. Bei dem *Timestamp-Modell* wäre eine Möglichkeit die Tripel direkt mit Zeitinformation zu assoziieren. Das erste Modell, ähnlich der relationalen Welt, hat kleine Nachteile z.B. bei der Verwaltung von Validzeit-Daten und Anfrageverarbeitung. Es kann aber für Transaktionszeitdaten verwendet werden, da die Transaktionszeit viel Gemeinsam mit Versionierungskonzept hat (nähe zur Versionierung von RDF Daten wird im nächsten Kapitel beschrieben).
Ähnlich wird, wie in vorigen Abschnitt, in[20] argumentiert das *Timestamp-Modell* besser geeignet als temporales Modell für die Einführung der Zeitdimension in RDF. Das Grundkonzept des *Timestemp-Modell* in RDF ist *Labeling*, mit anderen Worten, explizite Assoziation von temporalen Information z.B. Gültigkeitsintervalls mit einem RDF Tripel. Mit diesem Konzept kann sowohl die Validzeit als auch die Transaktionszeit ausgedrückt werden. Für die Interpretation der Intervalle in [20] kann sowohl das Punkt- als auch Intervall Modell für die Anfrageverarbeitung verwendet werden . Durch die Verwendung des Timestamp-Modells lässt sich die Zeitdimension leicht durch die Erweiterung der Tripeltable mit Zeit-Attributen darstellen:

```
CREATE TABLE TRIPLE_TABLE(
RDF_Subject TEXT,
RDF_Predicate TEXT,
RDF_Object TEXT,
Start TIMESTAMP,
End TIMESTAMP
);
```

Es gibt aber auch eine andere Möglichkeit, nämlich die Zeitkomponenete über die RDF selbst auszudrücken. Dazu werden die Reification-Tripel verwendet. Mit diesem Konzept kann auf die Erweiterung der Tripletable um Zusatzattribute verzichtet werden.
Temporalen Annotationsdaten über temporale Tripel werden auch in RDF ausgedrückt und können mit RDF Daten verwaltet werden. Dabei könnten die Reification-Tripel direkt in der Tripeltable von dem temporalen Graphen gespeichert werden. Es kann auch eine Zusatztabelle für diesen Zwecken angelegt werden. Aus den praktischen Sicht gibt es sowohl Vor- und Nachteile bei beiden Techniken.
Bei der zweiten Lösung z.B. für die Anfargeverarbeitung müssen erst die Reifications-Tripel berechnet werden, bei der ersten dagegen, ist die Zeitinformation fester Bestandteil eines

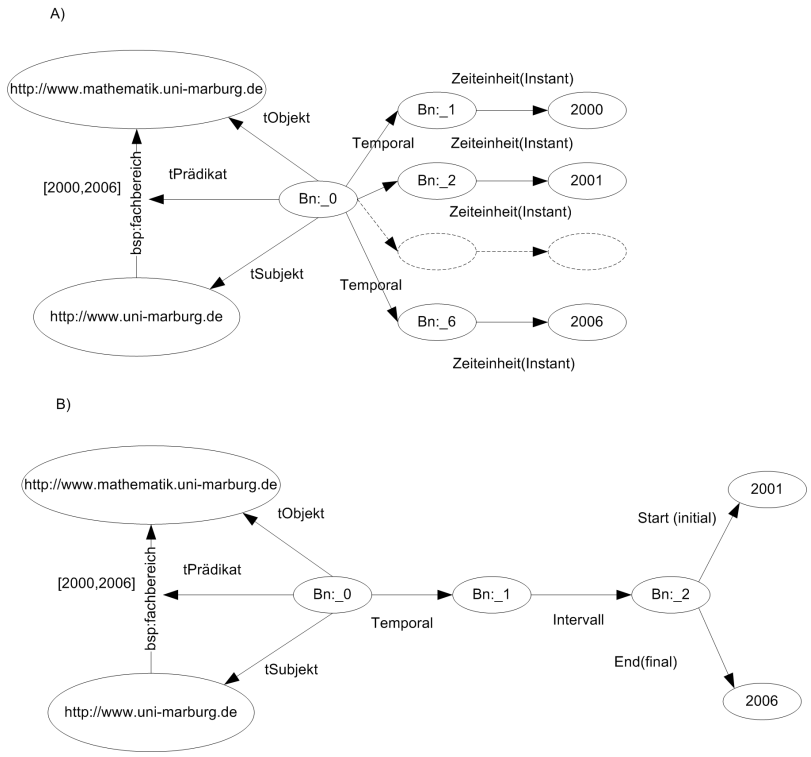

Abbildung 2.3.: A) Beschreibt die Reification mit Bezug auf Punktmodell; B) Temporale Reification mit Intervall-Modell

Datenbanktupel. Einer der Vorteilen der zweiten Lösung ist, dass auch Aussagen über die Reifications-Tripel gemacht und mitverwaltet werden können. Bei der ersten Möglichkeit müssen erst durch eine z.B. Mapfunktion die Zeitinteravallinformation auf Reificationsgraphen abgebildet werden. Trotz der offensichtlichen Einfachheit der ersten Technik, wird die zeitliche Vokabular und Reification eingeführt, um die Syntax von temporalen RDF Graphen zu definieren. Dadurch lassen sich Aussagen über die Zeitdimension erstellen, das wiederum ermöglicht semantische Besonderheiten des RDF beizubehalten. In Abbildung 2.3 kann der Beispiel-Reifikationsgraph veranschaulicht werden.

Der zentrale Begriff, der in [20] eingeführt wird, ist der *Temporalle RDF Graph*[3]. Im Folgenden wird eine semantische Sicht auf den temporalen RDF Graph präsentiert:

Definition 2.2.1 (Temporaler RDF Graph) *Ein temporale RDF Graph ist die Mengen von temporalen Tripel. Eine temporale RDF Tripel ist ein Triple mit (s,p,o)[t], wobei t ist ein temporales Label.*
Die Notation (s,p,o)$[t_1, t_2]$ *bezeichnet die Menge* $\{(s, p, o)\,[t] \mid t_1 \leq t \leq t_2\}$

Ein recht simples Konzept, das auf Grundlage des Timestamp-Modells basiert. Trotz dieser Einfachheit lassen sich andere Begriffe aus der formalen Semantik des Frameworks wie Teilgraph, Ableitung und Clousur auf Zeitdimension erweitern[20].
Da die temporale Erweiterung von RDF nach [20] nicht der Bestandteil von RDF ist, wird das Framework um spezielles temporales Vokabular erweitert. Mit diesem Vokabular können dann die Reification-Tripel definiert werden. Die Standardsyntax von RDF kann somit auf temporale Erweiterung angereichert werden(vgl. Abbildunng 2.3):

[3]Ein geschlossenes Intervall $[x_a, x_e]$ in dem Modell repräsentiert die linear geordnete Menge von Zeiteinheiten über einer diskreten Domain.

Definition 2.2.2 (Temporales Vokabular nach [20]) *Das temporale Vokabular zur Definition von Reification-Tripel besteht aus:* `temporal` *(Kurz. tmp),* `instant` *(Zeiteinheit),* `interval` *(Zeitinterval),* `initial` *(Startpunkt des Zeitintervalls) und* `final` *(Endzeitpunkt). Diese RDF Typen sind von dem Typ* `property` *(Prädikate). Zusätzlich wird der aktuelle Zeitpunkt durch* `now` *dargestellt(mit dem Typ plain Literal). Der Definitonsbreich von* `instant` *,* `initial` *und* `final` *sind die natürlichen Zahlen.*

Wohl bemerkt, haben die Autoren auf die Standard Reifikationsvokabular von RDF verzichtet, um die Unabhängigkeit der temporalen Erweiterung zu gewährleisten (Es wird z.B. statt `rdf:subject` `tsubj` verwendet). Syntaktisch wird der temporale RDF Graph nach [20] wie folgt definiert(vgl. Abbildung 2.3):

Definition 2.2.3 *Syntaktisch das temporale Tripel mit eingeführtem temporalen Vokabular hat eine der folgenden Strukturen:*

- *(s,p,o), reif(s,p,o,X)*[4]*, (X, tmp, Y), (Y,* `instant`*, n), mit n als natürliche Zahl. Als Abkürzung wird folgende Notation verwendet (s,p,o)*$[X, Y, n]$
- *(s,p,o), reif(s,p,o,X), (X, tmp, Y), (Y,* `interval`*, Z), (Z,* `initial`*, I), (Z,* `final`*, F), wobei F, I natürliche Zahlen sind. Abgekürzt wird mit (s,p,o)*$[X, Y, I, F]$

Temporaler RDF Graph ist definiert dann durch die Menge von temporalen RDF Tripel.

Mit Hilfe der Reification und temporalem Vokabular wird in [20] die Einführung des neun Begriffs *anonyme Zeit* ermöglicht. Durch das Konzept des leeren Knoten können die Zeitangaben von Labels nicht nur als Konstante, sondern auch als Variablen, behandelt werden. Z.B. wenn die Gültigkeit einer Aussage aktuell nicht bekannt ist, kann das durch $(s, p, o) : [X]$ ausgedrückt werden (mit X als anonyme Zeitvariable, die ausdrückt, dass das Tripel in einer unbekannte Zeit gültig ist). Eine der Anwendungen diese Technik ist Assoziierung von unvollständiger temporaler Information.
Anderer Vorteil von diesem Konzept, dass ein nicht-temporaler Graph in einen temporalen RDF Graph umgewandelt werden kann. Zusätzlich ermöglicht die Einführung der Zeitvariablen die Zeitangaben abzuleiten. Ein temporaler RDF Graph, der um die anonyme Zeitvariablen erweitert wird, wird als *generalisierter temporaler RDF Graph* definiert[5].
Die Einführung des generalisierten temporalen RDF Graphen mit Zeitvariablen erlaubt eine Anfragesprache für temporale RDF Daten zu definieren. Die Anfragesprache folgt dem SPARQL-Graph-Matching-Modell und basiert auf der Tableu-Anfragen für einen Graph-Datenmodell. Als kleines Beispiel, betrachten wir folgende Anfrage an unsere Beispiel-Datenbank: *„Finde alle Projekte in Bereich Semanitk-Web an der Fachbereich Mathematik und Informatik der Uni-Marburg zwischen 2000 und 2006 “*

$$
\begin{aligned}
(X?, \text{type}, \text{projekt}) \quad \longleftarrow \quad & (X?, \text{type}, \text{projekt})\,[T?]\,, \\
& (X?, \text{gebiet}, \text{Semanitk Web})\,[T?]\,, \\
& (\text{http://www.mathematik.uni-marburg.de}, \text{hatProjekt}, X?)\,[T?]\,, \\
& 2000 \leq T?, T? \leq 2006.
\end{aligned}
$$

Die Sprache muss natürlich eine eingebaute Arithmetik für die Zeitinetravalle bereitstellen. Zusätzlich wurden im Paper die optionale Zusatzoperatoren besprochen, wie Aggregation `Max` und `Min` z.B. die Anfragen wie *„Gibt das längste Zeitintervall in dem die Aussage (s,p,o) gültig war“*. Die Vertiefende Diskussion über Komplexität und Ausdruckstärke der Sprache und andere Besonderheiten der temporalen Sprachen für RDF sind in [20] beschrieben.

[4] reif ist Abkürzung für Reification
[5] In [20] vertieft die Semantik der generalisierte temporale RDF Graph präsentiert

2.3. Zusammenfassung

In diesem Kapitel wurden die wichtigsten Begriffe aus temporalen Datenbanken, wie Snapshot, Transaktionszeit, Validzeit, Bitemporale Zeit und Userdefinierte Zeit, präsentiert. Es wurde erläuteret, dass Timestamp-Modell als grundlegendes abstraktes temporales Modell in relationalen Datenbank verwendet wird. Aus dem praktischen Sicht erweitertet das Modell die Tabelle im einfachsten Fall um zwei Attribute, die den Anfangs- und Endpunkt des Zeitintervalls darstellen. Es wurde kurz beschrieben, dass das Modell auf ersten Blick sehr einfach scheint, verbirgt aber Schwierigkeiten, insbesondere in Hinsicht auf Validzeit (Erhalt der Konsistenz, Gewährleistung von temporalen Integritätsbedingungen).
Was in diesem Kapitel nicht besprochen wurde, ist die erweiterte Anfrageverarbeitung, die Erweiterung von und Einführung von speziellen Operatoren(z.B. der Operator das *Coalecing* durchführt) und Intervall Arithmetik. Die vertiefende Diskussionen zu diesen Themen sind in [29] zu finden.
Ein gutes Buch zur Implementierung von temporaler Datenbank in einem RDBMS ist [16]. Eine wichtige Schlussfolgerung des Kapitels ist, dass das Timestamp-Modell auch als abstraktes temporales Modell für RDF verwendet werden kann. Die Konzepte aus dem relationalen temporalen Modell lassen sich auf das RDF übertragen. Im Anführung wurde kurz angesprochen, dass es auch andere Forschungsarbeiten in theoretischen Bereich des temporalen RDF existieren [21][22]. Die sind aber von der generellen Natur und lassen nicht nur die Zeitdimension in RDF, sonder beliebig andere Dimensionen, einführen. [20] wurde nur die Zeiterweiterung betrachtet. Die wichtigsten Kontributionen von [20] ist die Untersuchung der Anwendung des Timestamp-Modells (explizite Assoziierung der Zeitinformation mit RDF Tripel), formale Definition von Zeit in RDF Erweiterung von RDF Semantik und Ausarbeitung einer möglichen temporalen Anfragesprache.
Im nächsten Kapitel geht es um die Indexstrukturen für die temporale RDF Daten, ähnlich wie in diesem Kapitel, wird zuerst temporale Strukturen aus relationalen Datenbanken präsentiert. Danach wird es erläutert, wie sie für die Verwaltung von zeitabhängigen RDF-Daten angepasst werden können. Unter andrem werden auch spezielle Indexstrukturen für RDF Daten vorgestellt.

3. Zugriffstrukturen für RDF Daten

Bei der Verwaltung von großen Mengen von Daten ist die Verwendung von effizienten Zugriffstrukturen unabdingbar. Die Größenordnung von typischen RDF Datenbanken liegt weit über Millionen von Tripel.
Aus diesem Grund, wird effiziente Verwaltung von RDF Daten unter Berücksichtigung des I/O Modell benötigt. Bei der Verwaltung der RDF Daten in einem RDMBS sind diese großen Mengen allein mit „Full Table Scans“ nicht effizient zu bewerkstelligen. Die modernen DBMS bieten verschiedene Indextechniken und Strukturen für verschiedene Art von Daten und Anfragemuster.
Die Indexstrukturen für RDF Daten lassen sich grob in zwei Typen unterteilen:

- Standard Indexstrukturen aus RDBMS und Information Retrieval wie z.B. B+Bäume, Externes Hashing, Invertierte Listen.
- Spezielle Indexstrukturen, für Indexierung von RDF Graphen. Diese sind primär für die Beantwortung von Graphanfragen entwickelt worden.

Für die Entwicklung und Anpassung der Indextechniken für RDF Daten, müssen unter anderem die Struktur und Speicher-Modell (z.B. ob die Datensätze in einer „normalen“ RDBMS oder Spaltenorientierneten RDBMS verwaltet werden etc.), so wie statistische Metadaten z.B. Werteverteilung berücksichtigt werden. Der Kernpunkt ist aber, dass RDF Daten eine Graphstruktur sind, das stellt eine zusätzliche Anforderung an Indexstrukturen. Zusätzlich ist es wünschenswert effizient auch die erweiterten Graphanfragen, wie die allgemeinen Pfadausdrücke, zu verarbeiten.
Werden zum Beispiel herkömmliche Indexstrukturen betrachtet, werden die Graphanfragen in RDBMS mit Joins verarbeitet. Das stellt auch einen zusätzlichen Aspekte bei der Planung von Index-Implementierung. Für temporale RDF Daten kommt natürlich auch ein zusätzlicher Grad an Schwierigkeit, nämlich Verarbeitung von zeitbezogenen RDF Daten.
Dieses Kapitel behandelt erst die konventionellen Techniken aus RDBMS und deren Einsatz bei der Verwaltung von RDF Daten. Es wird beschrieben, welche Index- und Zugriffstrukturen aktuell in der Praxis eingesetzt werden. Im anschließenden Unterkapitel werden die temporalen Indexstrukturen präsentiert. Dabei wird Klassifikation von temporalen Indexen in Abhängigkeit von der Zeitsemantik vorgenommen.
Es wird gezeigt, wie sie auf die Verwaltung von RDF Daten angepasst werden können. Danach wird die mögliche Anwendung des Transaktionszeitindexes für RDF Daten erläutert. Anschießendes Unterkapitel beschreibt Indexe von dem zweiten Typ. Am Ende des Kapitels wird eine Indexstruktur tiefer erläutert, da sie speziell für temporale RDF Graphen entwickelt wurde.
Im laufe der Entwicklung von RDF wurde auch viele effiziente Hauptspeicher-Indexe für relativ kleineren RDF Graphen vorgeschlagen. Der Akzent der Arbeit liegt aber auf die externspeicher-basierten Verfahren und die Manipulation von großen Mengen der RDF-Daten.
Die Abschätzung des Aufwands von Strukturen hinsichtlich der Zeit- und Platzkomplexität folgt in diesem und nächsten Kapitel dem I/O Modell. Im Kapitel werden folgende Notation für die Kosten, falls nicht andres angegeben wird, verwendet: n-Anzahl der Datensätzen, B

ist die Kapazität einer Speicherseite in externen Speicher (bei der Datensätzen mit fester Länge, gibt diese Zahl die Anzahl der Datensätzen, sonst Anzahl der Bytes). Durch b wird Mindestanzahl von Daten in einer Seite bezeichnet.

3.1. Konventionelle Zugriffstrukturen für RDF Daten

Die Verwaltung von RDF Daten in einem RDBMS hat sich gegenüber andre Lösungen etabliert. Die Tatsache, dass die RDF Daten in einfacher Form als Datenbanktupel mit drei Attributen gespeichert werden können, ermöglicht direkte Abbildung auf eine relationale Tabelle. Dazu kommt auch, dass die SPARQL Anfragen sich mit Hilfe von relationalen Operatoren ausdrücken lassen. Deshalb lassen sich die SPARQL Anfragen in einen SQL übersetzten, und für die Anfrageverarbeitung die relationale Operatoren von RDBMS verwenden. Es spricht viel dafür, auch die Index- und Zugriffstrukturen aus diesem Bereich einzusetzen.
In diesem Unterkapitel wird geklärt, inwiefern sich die Indexkonzepte aus relationalem Bereich für die RDF eignen und welche Indextechniken sind besonders gut geeignet. Um die Notwendigkeit einer Indexverwaltung zur begründen, betrachten wir das folgende Beispiel:

Beispiel 3.1.1 *Nehmen wir an, dass die RDF Daten[1] sind in einer einfachen Tripletable mit Namen „Personen“ gespeichert(mit folgende Schema (S, P, O)). Zusätzlich wird angenommen, dass es mindestens eine Million Tripel in der Tabelle vorhanden sind. Nehmen wir an, dass uns die Emailadressen und Namen von Personen die „Max Mustermann“ kennen, die auch „Max Mustermann“ kennt, interessieren. (Wir nehmen an, dass das eindeutige URI von Max Mustermann bekannt ist.)*

```
SELECT $name $email
WHERE {
?personA foaf:knows 'http:/beispielRelation/MaxMustermann'.
'http:/beispielRelation/MaxMustermann' foaf:knows ?personA.
?personA foaf:mbox $email.
?personA foaf:name $name.
}
```

Mit Beispiel Plan:

$$
\begin{aligned}
A = &\quad \sigma_{P=foaf:knows \wedge O='http:/beispielRelation/MaxMustermann'}(Personen) \\
B = &\quad \sigma_{P=foaf:knows \wedge S='http:/beispielRelation/MaxMustermann'}(Personen) \\
C = &\quad (A \bowtie_{\theta(S_A=O_S \wedge O_A=S_B)} B) \\
D = &\quad \pi_P(C \bowtie_{\theta(S_C=S_{Personen})} (\sigma_{P=foaf:email}(Personen))) \\
F = &\quad \pi_P(C \bowtie_{\theta(S_C=S_{Personen})} (\sigma_{P=foaf:name}(Personen))) \\
\ldots &\quad \ldots
\end{aligned}
$$

Bei der Verarbeitung im RDBMS könnte zuerst die Selektion nach den Tripel, in denen „Max Musterman“ als Objekt und als Subjekt fungiert, verarbeitet werden. Das Ergebnis könnte weiter an Join Operator geleitet werden. Im Anschluss können durch weiter Joins Emailadressen und Namen selektiert werden.

[1] Im Beispiel wir FOAF Schema verwendet, http://www.foaf-project.org/

In einem RDBMS lässt sich diese Anfrage ausschließlich unter Verwendung von relationalen Scans beantworten. Allerdings wäre es für die die vorausgesetzte Menge von Daten äußerst ineffizient. Ein erfahrener Datenbankadministrator würde sich fragen, über welchen Spalten (Spaltenkombinationen) unserer Beispielrelation Indexe angelegt werden sollen, um die Effizienz zumindest von Selektion zu verbessern. Zum Beispiel wäre es, für die Verarbeitung von den ersten beiden Matchings, Hilfreich den Index auf Spalten O oder sogar PO analog S oder SO anzulegen. Für die anschließende Selektion nach Namen und Emailadressen könnte ein Index auf alle drei Attributen angelegt werden.
Das Beispiel Zeigt, wie die Effizienz der Anfrageverarbeitung durch den Einsatz von Indexe gesteigert werden kann. Dabei werfen sich die Fragen, welche Art von Indexe angelegt werden sollen und was sind die Anforderungen an ihnen. Im Beispiel wäre eine Möglichkeit für die Indexe auf SP und PO B+Bäume und für den Index auf SPO ein externes Hashverfahren zu verwenden. Die ersten beiden Matchings würden dann durch effiziente Bereichsanfrage an B+Baum und anschließendes Matching durch Hashindex-Zugriffe berechneten. Anstelle des Hashindexes könnte auch der B+Baum auf SPO aufgebaut werden, so dass auf den Index auf SP verzichtet werden kann.
Im Allgemeinen ist aber nicht bekannt, über welche Attribute Selektion durchgeführt wird. Daher können Indexe über alle möglichen Spaltenkombination angelegt werden. Generell beim Anlegen eines Indexes in RDBMS sollten folgende Überlegungen herangezogen werden: die typischen Zugriffmuster, Verteilung von Daten, Typdomain der Attribute und Dynamik von Daten (z.B. Häufigkeit der Update-Operationen).
RDF stell auch durch seine funktionalen und strukturellen Besonderheiten zusätzliche Anforderungen auf:

- Die Zugriffstruktur soll die Verwaltung von zusammengesetzten Attributen unterstützen.
- Es sollen möglichst effizient die partielle Anfragen ausgeführt werden.
- Die Indexstrukturen sollen möglichst effizient die Zeichenkettendaten verwalten.
- Die Indexe sollen optional Clusterung der Daten unterstützen.

In der Praxis wird am häufigsten B+Bäume[23][26]-Verfahren für die Indexierung von RDF Daten eingesetzt, da sie die wesentliche Anforderungen erfüllen. Seltener werden auch die externe Hashverfahren verwendet. Zusätzlich können Hifsindexe aus dem Bereich Information-Retrieval aufgebaut werden(wie Signatur Bäume und invertierte Listen, falls die Volltext-Suche-Funktionalität erwünscht ist).
B+Bäume unterstützen die zusammengesetzten Attribute. Durch die Ausnutzung der Sortierreihefolge können die partiellen Anfragen direkt durch den Index beantwortet werden. Da die Typdomain des Tripel im Allgemeinen die Zeichekettendomain ist, können speziell für die Zeichenketteverwaltung entwickelten Präfix-B+Bäume verwendet werden. Zusätzlich lassen sich die Zeichenketten gut komprimieren [23].
Die einfachste Idee wäre, die allgemein bei zusammengesetzten Attributen angewendet werden kann, zum Beispiel in einem SPO-Index-Seite bei den Daten mit gleichem führenden Attribut nur den ersten Datensatz vollständig und die Nachfolger partiell zur speichern. Eine andere Möglichkeit ist die Δ-Komprimierungstechnik zu verwenden. In [23] wurden, basierend auf diesem Verfahren, die Daten mit speziell für RDF Daten entwickelten Algorithmus komprimiert. Außerdem, nicht zu vergessen, garantieren B+Bäume die logarithmische Zugriff- und Anfragekosten.
Ein geclusterter Index (z.B. Verwaltung der Datensätzen in Blätterknoten des B+Baums) kann sehr hilfreich bei der Join-Verarbeitung sein. Eine Besonderheit des RDFs ist eine Graphstruktur, die Graphanfragen, die im Allgemeinen einen Pfad in den Graph berechnen,

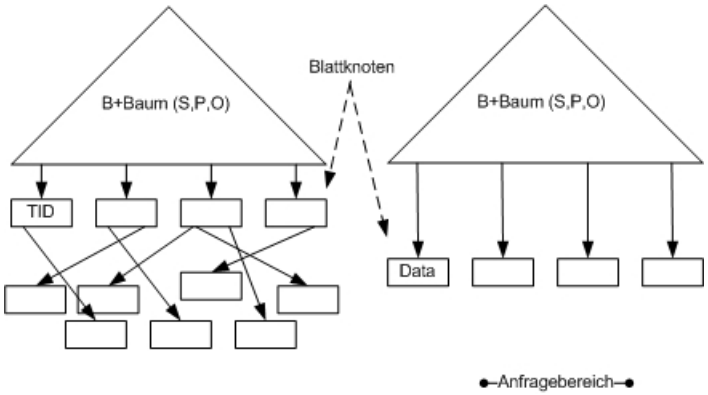

Abbildung 3.1.: Unterschied zwischen den geculsterten und nicht geclusterten B+Baum.

kommen nicht in einem relationalen System ohne effiziente Join-Verarbeitung aus. Die komplette Seite eines geclusterten B+Baum Index kann nach eine z.B. Bereichsanfrage direkt für die Joinverarbeitung genutzt werden, damit wird je nach Joinanlgorithmus bis zur B I/O Zugriffe pro Blattseite des B+Baums eingespart. Außerdem sind die Daten des B+Baums nach den indexierten Attributen sortiert, was den Einsatz von Sort-Merge-Join-Algorithmen sehr attraktiv macht[23].

Nach dem die Frage, welche Indexstruktur am besten sich für die Indexierung der RDF Tripel eignet, geklärt wurde, wenden wir uns der Frage, wie viele B+Baum Indexe und über welche Attribute angelegt werden sollen. Auf drei Attributen des RDF Triple können folgende Zugriffsmuster (Teilmengen von (S, P, O)) auftreten:

$$\{S, ?, ?\}, \{?, P, ?\}, \{?, ?, O\}, \{S, P, ?\}, \{?, P, O\}, \{S, ?, O\}, \{S, P, O\}$$

Auf ersten Blick scheint es, dass es Sieben verschiedenen B+Bäume angelegt werden sollen, allerdings erlauben die B+Bäume, durch die Sortierung, partielle Suche auf den Präfixattributen eines zusammengesetzten Tupel. Damit lassen sich die Zugriffmuster mit z.B. $(S, ?, ?)$, $(S, P, ?)$, (S, P, O) sich mit einem (S, P, O) B+Index beantworten. Durch die lexikographische Sortierung nach den indexierten Attributen lassen sich mit drei folgenden B+Indexen alle Zuggriffmuster beantworten:

$$\{S, P, O\}, \{P, O, S\}, \{O, S, P\}$$

In [47] wird die allgemeine Fragestellung bezüglich partiellen Anfragen auf Tupel mit n-Attributen beantwortet. Dabei ist die wichtigste Frage, wie viele Permutationen von n -Attributen benötigt werden, um alle Zugriffmuster abzudecken. Die Zugriffmuster sind äquivalent zur Teilmengen der n-Attributen.

Im [47] wird gezeigt, dass es eine allgemeine Schema für Konstruktion von $\binom{n}{k}$ kombinierten Attributen aus n Attributen mit $k \leq \frac{1}{2}n$ existiert, so dass Tupel mit Kombinationen der Attribute mit Anzahl $a \leq k$ oder $a \geq n - k$ Attributen, werden in einem der B+Bäumen sortiert vorkommen.

Unter Umständen reicht auch wenigere Anzahl der Kombinationen aus. Falls eins der Attributen, eine relativ kleine Selektivität bei der n Attributen besitzt, kann diese Wert als führende Attribut bei den $\binom{n-1}{k}$ Permutationen von n [47] gestellt werden. Die Suche kann dann mit *Skip* bzw. *Jump* Technik erfolgt werden. Diese Optimierung kann auch bei RDF Indexierung sehr hilfreich sein.

Im Einführungskapitel wurde die *Vertikale Partitionierungs-Technik* vorgestellt. Bei vielen

RDF Daten ist die Selektivität der P Prädikatwerten im Vergleich mit S und O Attributwerten sehr gering. Daher bietet sich, die Daten nach P Werte zu partitionieren[13]. Für das Schema (S, P, O) reichen dann $\{P, S, O\}$ und $\{P, O, S\}$ B+Indexe aus. Die P Werte könne auch durch eine Mapping Tabelle auf kleinere numerische Werte abgebildet werden und z.B. ähnliche Technik wie *Partition B-Tree*[46] eingesetzt werden kann.
In [23] werden jedoch über alle Permutation von S, P, O Schema B+Indexe angelegt. Die ausgeklügelte Komprimierungs-Technik der Indexeinträgen ermöglicht effiziente Platzausnutzung. Die Verwendung von alle Permutation ermöglicht dem Anfrageroptemierer bessere Entscheidungen für die Join-Verarbeitung zu treffen. Durch die Ausnutzung der Sortierung und Clusterung der Daten in Blattknoten kann sehr effizient der Sort-Merge-Join-Algorithmus angewendet werden.
Im Vergleich werden im System von [26] auch B+Bäume verwendet. Allerdings werden nicht die Tripel sondern die Tupel mit vier Komponenten $(S, P, O)C$ (das vierte Attribut steht für den Graphkontext) verwaltet. Nach dem Prinzip von [47] werden $\binom{4}{2} = 6$ Indexe Angelegt, um alle Zugriffmuster abzudecken. Einen guten Übersicht und Vergleich von B+Indexorganization in verschiedenen Systemen bieten [23][12].
Zusätzlich bei der RDF Daten können auch die Join-Indexe für die typischen Graphanfragen angelegt werden[12][23][11]. Bei dem Entwurf von RDF Index können auch die Idee wie Multidimensional B+Trees [44][45] in Betracht gezogen werden, denn die erweitern die B+Bäume auf die Verwaltung von Einträgen mit zusammengesetzten Attributen und erlauben die partielle Anfragen in einer Struktur. Beim Multidimensional B+Tree von [44] stellt eine allgemeine Struktur, deren Index- und Blattknoten selbst die B(+)Bäume zu jeweiligen Attributen sind, dar. Das heißt, es kann ein Multidimensional-B+Tree über RDF Tripel direkt aufgebaut werden (z.B. mit P Attributen als erste Schlüssel und zu jedem P Wert der Baum deren führende P Werte gleich sind und usw.) Diese Indexhierarchie ist dann auch horizontal zu jeweiligen Attributen verlinkt (z.B. P, S und O Level).
Am häufigsten zeigen die Daten in RDF ein statisches Verhalten, die Daten werden in periodischen Abständen geladen und es überwiegen die lesenden Zugriffe. Für das Laden von großen Mengen von Daten eigenen sich wiederum die B+Bäume, da für sie effiziente *Bulk-Loading* Verfahren existieren. In vielen Systemen werden die statischen Varianten der B+Bäumen eingesetzt (die Vorläufe der B+Bäume) die Index-Sequentielle Zugriffstrukturen. In Spaltenporientierten Lösungen für RDF sind die Spaltentabellen ohnehin als Index-Sequentielle Zugriffstrukturen organisiert[51]. Zudem sind die einzelnen Spaltentabellen mit Zusatz Join-Indexe verlinkt. Aus diesem Grund zeigen diese Systeme extrem gute Leistung beim lesenden Verhalten.
Zusammengefasst: RDF Tripel lässt sich auf ein relationales Datenbanktupel abbilden und mit einem B+Baum über Attribute S, P, O indexieren. Generelle Mindestanzahl der benötigten Bäumen für effiziente Verarbeitung ist drei mit $(S, P, O), (P, O, S), (O, S, P)$. Die Beste Performanz wird aber mit allen sechs Permutationen erzielt. Das erlaubt dem Anfrageoptimierer den günstigeren Plan zu wählen. Die B+Bäume sollten am besten ihre Daten direkt in Blattebene verwalten und nicht die TIDs, um die Join Effizienz zu steigern.

3.2. Temporale Zugriffstrukturen für RDF Daten

Das Unterkapitel beschäftigt sich mit dem Einsatz von Indexstrukturen bei der Verwaltung von Daten mit zeitlichem Bezug. Die Aufnahme der Zeitdimension fordert neue Überlegung für die Implementierung von Indexstrukturen. Die temporale Datenbanken lassen sich gemäß der Zeitsemanitk in Transaktionszeit- , Validzeit- und bitemporale Datenbanken unterteilen. Diese Unterteilung prägt auch die Zugriffstrukturen-Design. Gemäß dieser Klassifizierung

werden auch die Indexstruktren für temporale Daten unterscheiden:

- Transaktionszeit-Techniken
- Validzeit-Techniken.
- Bitemporale Techniken.

An dieser Stelle wird noch ein mal einer der Hauptunterschiede der Semantik der temporalen Daten beschrieben, der maßgeblich die Implementierung der Strukturen beeinflusst. Bei der Transaktionszeit Daten erfolgt kein physisches Löschen von Daten, deshalb die größte Herausforderung bei der Entwicklung von Indexstrukturen eine Balance zwischen den Zugriff/Anfragekosten und Platzeffizienz zu erstellen.
Bei den Validzeit Daten ist die Einfügereihenfolge unabhängig von der Zeitdimension, bei diesen Daten liegt die Betonung an die effiziente Verwaltung von assoziierten Zeitintervallen. Aus der Sicht der Zugriffstrukturen kann die Validzeit Datenbank als eine dynamische Kollektion von Intervallobjekten interpretiert werden. Einen guten Überblicken und Vergleich der Index-Techniken für temporale Datenbanken bieten [28]. Das Unterkapitel basiert zum größten Teil auf diese Arbeit.
In dem Kapitel werden nur Techniken für Transaktionszeit und Validzeit präsentiert. Die Bitemporale lassen sich mit Hilfe der vorgestellten Methoden entwickeln. Es gibt auch natürlich die spezielle Zugriffstrukturen für diese Zeitsemantik. Die vertiefende Ausarbeitung zu diesem Thema ist in [28] zu finden.
Im folgenden werden die wesentliche Anforderungen für die Implementierung von Zugriffstrukuren zusammengefasst[28].

- Transaktionszeit-Zugriffstrukturen:
  - Alle vergangene logische Zustände einer temporalen Datenbank sollen gespeichert werden.
  - Datenmanipulierende Operationen ändern *nur* den aktuellen logischen Zustand einer Datenbank.
  - Die Struktur sollte effizienten Zugriff auf alle Zustände der Datenbank unterstützen.
- Validzeit-Zugriffstrukturen:
  - Zugriffstruktur verwaltet den aktuellsten Zustand (Menge) von Intervallobjekten.
  - Unterstützt die datenmanipulierenden Operationen (Einfügen/Löschen/Updaten) an dieser Menge
  - und Implementiert effiziente Anfargeverarbeitung
- Bitemporale-Zugriffstrukturen:
  - Sollten wie bei der Transaktionszeit alle Zustände der Menge von Intervallobjekten verwaltet werden
  - Datenmanipulierende Operationen ändern *nur* die aktuelle Menge von Intervallobjekten
  - Effizienter Zugriff soll auf alle Zustände der Intervallobjekten-Menge implementiert werden

Aus der Sicht der Anfrageverarbeitung sind sowohl die Transaktionszeit- als auch die Validzeit-Indexstrukturen Mengen von Intervallobjekten.
Der Einsatz von Techniken für Temporale Daten aus den relationalen Datenbanken für RDF

Daten ist genau so berechtigt, wie der Einsatz von B+Bäumen für nicht temporale RDF Daten. Die RDF Tripel beim Verwalten in RDBMS wird auf Tupel (S, P, O) abgebildet. Mit Einführung der Zeitdimension auf Grundlage des Timestamp-Modells lässt sich dann das Tupel um die Zeitintervall erweitern $(S, P, O)\,[t_s, t_e]$.

3.2.1. Transaktionzeit-Zuggriffstrukturen

In einer Transaktionszeitumgebung agieren die datenmanipulierende Operationen ausschließlich auf den aktuellen Zustand einer Relation, durch die Veränderung des aktuellen Zustand, wird ein neuer Zustand erzeugt. Die Daten aus dem „alten“ aktuellen Zustand werden dann nur lesend über die Anfragealgorithmen zugegriffen.
Eine der größten Problematik beim Entwurf einer Zugriffstruktur in einer Transaktionszeitumgebung ist eine große Menge von historischen Daten (da die Daten nicht physikalisch gelöscht werden), daher müssen auch nicht nur die Methoden im Hinsicht auf effizienten Speicherplatzverbrauch entworfen werden, sondern es könnte auch der Zugriff auf den aktuellen Zustand möglichst effizient gestaltet werden, da im Allgemeinen auf „live“[2] Daten häufiger zugegriffen wird. Im Praxis werden die historischen Daten auch häufig auf tertiär Speichermedien verschoben und die aktuellen Daten weiter in dem Sekundärspeicher verwaltet. Beim Entwurf von Strukturen muss dann auch der Migrationprozess von der Daten auf tertiär Speichermedien berücksichtigt werden. Wobei, falls die „alten“ Daten zur der Struktur sequentiell hinzugefügt werden, ist auch denkbar, die Daten auf die so genannte WORMs (eng. write onse read miltiple) Speichermedien zu migrieren. Gleichzeitig muss mit großer Sorgfalt überlegt werden, welche Anfragen soll die Struktur unterstützen. Natürlich alle diese Überlegungen sind Anwendungsspezifisch.
Einen großen Einfluss auf den Entwurf einer Zugruffstruktur spielen die Anfragetypen, die von der Struktur unterstützt werden sollen. Für Tranksationszeit-Zugriffstrukuren sind die folgenden drei Typen von Anfragen von großer Bedeutung[28], im Großen und Ganzen, sind das die Anfragen auf zweidimensionalen Schlüssel-Zeit-Raum:

1. *Pur Zeit Anfragen*: Gegeben ist ein Intervall T, gib alle Datenobjekte, die während diesem Zeitintervall gültig sind. Nach der Transaktionszeit-Semantik, das sind die Daten, dessen Löschzeitpunkt nicht in dem Anfrageintervall liegt. Eine abgeschwächte Form ist die so genannte *transaction pure-timeslice Anfrage*, bei der T zu einem Punkt reduziert wird.

2. *Schlüssel-Zeit Breichsanfragen*: Gegeben ein Schüsselintervall K und ein Zeitintervall T, finde alle Datenobjekte, deren Schlüssel im K liegen und gültig in T sind. Auch hier sind die abgeschwächte Formen denkbar, eine in der Praxis häufig vorkommende Anfrage ist so genannte *transaction range-timeslice* bei der wiederum, das T zu einem Zeitpunkt reduziert ist.

3. *Pur Schlüssel Anfragen*: Gegeben ein Schüssel Intervall K gib eine Historie von Datenobjekten aus dem Bereich K aus. Eine besondere Form der Anfrage ist *transaction pure-key*, die eine Historie zu einem bestimmten Schlüssel berechnet.

Der Abschnitt ist so aufgebaut, dass es der Reihe nach die ausgewählten Techniken und Methoden für jeweiligen Anfrage Typen präsentiert werden. Als erstes werden zwei grundlegende Methoden für die Transkationszeit-Verwaltung vorgestellt.
Beide grundlegende Verfahren können zur Unterstützung von *transaction pure-timeslice Anfrage* und *transaction pure-key* herangezogen werden.

[2] „Live“ werden die Daten aus dem aktuellen Zustand bezeichnet (die Daten die noch nicht gelöscht sind)

- *Copy*-Technik: Wie der Name schon sagt, werden bei dieser Technik die Kopien der Zuständen $s(t)$ einer Datenbank für jeden Transaktionszeitpunkt t, bei dem mindestens eine Änderung des aktuellen Zustandes stattgefunden wurde, gespeichert. Der Zugriff auf den $s(t)$ könnte über einen Mehrwegbaum (z.B. B+Baum)auf t organisiert und erfolgt werden. Offensichtlicher Nachteil der Methode ist seiner Platzverbrauch, der im schlimmsten Fall $O(n^2/b)$ beträgt, obwohl eine ausbalancierte Anzahl von Lösch- und Updateoperationen den Plaztverbrach mindern können, im Falle, bei dem die Einfügeoperatioen überwiegen, bleibt der Aufwand sehr groß, da die „live“ Daten ständig mitkopiert werden sollen. Zudem bei Update-Operationen müssen im schlimmsten Fall alle indexierte Zustände linear mit $O(n/b)$ durchgesucht werden. Die einzelnen Versionen könnten wiederum in einem Index organisiert werden.

- *Log*- Technik: Die quadratischen Kosten der Copy-Technik lassen sich durch die Verwaltung der Protokolldatei, den so genannten *log*, verringern. Es werden lediglich nur die Δ von Zuständen mit der Log-Datei verwaltet. Zudem reduzieren sich die Update-Kosten auf die Konstante, da nur ein Eintrag zu die Logdatei hinzugefügt werden muss. Leider durch diese Implementierung steigen die Zugriffkosten auf die historischen Zustände. Im schlimmsten Fall für Rekonstruktion eines Zustandes wird $O(n/B)$ Kosten benötigt.

Auch die Kombination von diesen zwei Techniken möglich. Es können periodisch zu den bestimmten Zeitintervallen die Zustände zusammen gespeichert und die Differenz zwischen diesen mit Log-Datei verwaltet.
Im nächsten Abschnitt wenden wir uns der Frage der Organisation der Indexstruktur für die Unterstützung von Anfragen nach Schlüssel-Historie. Für die Anfragen von Typ drei ist die beste Methode, die Listen (Arrays) der Historien zu verwalten. Der Zugriff auf die Listen kann durch einen Hash oder B+Baum organisiert werden. Der Letztere wird gegenüber Hashing bevorzugt, da der B+Baum die logarithmischen Zugriffskosten garantieren kann. Im pathologischen Fällen kann das externe Hashverfahren entarten. Noch ein Vorteil ist die Unterstützung von Bereichsanfragen. Diese Organisation kann somit vollständig die Anfragen von Typ drei beantworten. Im nächsten Abschnitt wird exemplarisch ein Basisverfahren präsentiert, das diese Strategie verfolgt.

Reverse Chaining

Reverse Chaining Organisation [28] ist nach der beschriebenen Methode aufgebaut. Die Struktur besteht aus zwei getrennten Speicherstrukturen, eine für die aktuellen Daten und eine für die Verwaltung von Listen für vergangene Versionen, diese Trennung ist begründet dadurch, dass auf die aktuellen Data öfter zugegriffen wird. Die erste besteht aus zwei B+Bäumen (die Notwendigkeit von dem zweiten B+Baum wird im Laufe des Abschnitts erklärt). Die Einträge von dem Verfahren sind Tupel, die einen Schlüssel, Attributwerte und ein Zeitintervall enthalten, zusätzlich erweitert um einen Zeiger auf die Vorgänger-Version.
Wenn der Schlüssel zum ersten Mal in die Struktur eingefügt wird, wird der Zeiger auf den *null* gesetzt. Entsteht eine neue Version des Datensatzes, wird diese in der „frontend“-Struktur ersetzt und die veraltete Version in den „backend“-Index verschoben. Zugriff auf den Schlüssen in der „frontend“-Struktur erfolgt über einen B+Baum(Abbildung 3.2).
Beim Löschen eines Tupel wird sein Schlüssel komplett aus den aktuellen Index gelöscht und der Datensatz wird in den zweiten B+Baum verschoben. Somit erfolgt der Zugriff auf die gelöschten Versionen über den zweiten Index. Die Anfragen, die nach eine Historie eines Schlüssel abfragen, durchsuchen erst den Frontend, falls der Schlüssel in dem aktuellen Index vorhanden ist, wird die Liste, die chronologisch rückwärts Verlinkt ist, bis zum gesuchten

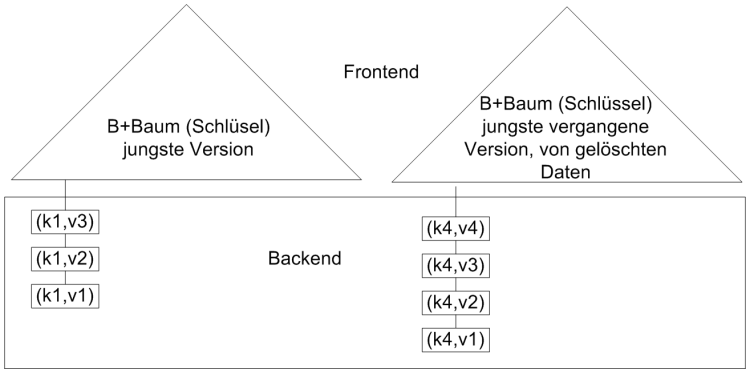

Abbildung 3.2.: *Reverse Chaining* Architektur. Linker Baum verwaltet die jüngsten Versionen. Der rechte Baum dient der Verwaltung von den gelöschten Objekten. Falls der Schlüssel in dem linken Baum nicht vorhanden ist, wird er über den rechten Baum aufgesucht.

Zeitpunkt durchgesucht, ansonsten wird der zweiter Index abgefragt. Die Kosten für die *transaction pure-key* betragen, falls a die Anzahl die Vergangenen Version ist und die Versionen in verschiedenen Seiten gespeichert sind, $O(log_b n + a)$ ansonsten $O(log_b n + a/B)$. Es gibt auch eine Reihe der Verbesserungen der Laufzeit des Verfahrens, insbesondere wird die Organisation der Listen so modifiziert, dass es möglich wird, die Listen in den „backend"-Struktur effizient durchzulaufen.

Für die Beantwortung von allgemeinen Bereichsanfragen im Schlüssel-Zeit Raum, eignen sich am besten die Techniken aus dem multidimensionalen Bereich (insbesondere Geo-Bereich). Um die Verarbeitung effizienter zu gestalten, ist die beste Praxis die Daten physikalisch mit einem Schlüssel und Zeitintervall zu clusteren. Dabei wird versucht den Seiten in externen Speicher die zweidimensionalen Regionen (Schlüssel, Zeit) zu zuordnen.
Für die Anfrageverarbeitung wird nicht-überlappende Zerlegung des Raums erwünscht, ideal wäre es, dass die nicht-überlappende Regionen sich eins-zu-eins auf die Seiten abbilden lassen. Leider lässt es sich nicht immer eine direkte Zuordnung von Regionen (meistens wird der Raum in Rechtecken zerlegt) auf die Seiten der externen Speicher realisieren. Dabei bereitet die Schwierigkeiten die unterschiedliche Verteilung der Zeitintervalllängen.
Es könnte sein, dass ein Region ein Paar „langlebige" Intervalle und viele kürzere Intervalle enthalten und von der Speicherplatzgrösse nicht in einer Speicherseite passen (vgl. Abbildung 3.3). Ein Ausweg dabei, ist die Verwaltung von überlappenden Regionen oder die Datenduplizierung, eine andere Möglichkeit wäre die Verwaltung in dreidimensionalen Raum mit Schlüssel, Startpunkt, Endpunkt und dei anschließende Indexierung durch mehrdimensionalen Zugriffstrukturen.
Im Vergleich zu Verwaltung der Objekte in Geo-Anwendungen haben die Objekte aus dem Raum (Schlüssel, Zeit) keinen direkten Überlappungsbegriff, da die Daten im Bezug auf die Schlüssel-Achse immer disjunkt sind, und deshalb besitzen sie keine Überlappung. Aus diesen Überlegungen wird die Datenduplizierung bei temporalen Indexen bevorzugt.
Die Strukturen, die wie R-Bäume auf dem hierarchischen Aufbau der überlappenden Regionen basieren, fordern für die Schlüsse-Zeit Anfragen das Backtracking der Baumstruktur, insbesondere auch für die Punktanfragen. Im Vergleich zur Verwaltung von Geo-Objekten in der temporalen Umgebung, die relativ langen Intervalle sind fast immer Bestandteil der Daten. Demzufolge hat es die starke Auswirkung auf die Zeitanfragen, es werden zum Beispiel die Seiten geholt, die eine kleinere Anzahl der qualifizierten Elementen enthalten, da nur die

langlebigen Intervalle in der Seite, die den Zeitpunkt schneiden(vgl. Abbildung 3.3), liegen.
Beim Mapping der Zeitdaten auf den höheren Dimensionen kann wiederum die R-Bäume benutzt werden. Unter Umständen kann das eine gute Wahl sein, der Grund dafür, dass die „kürzere“ Daten werden zusammengeclastert. Zu diesem Thema wird nähe im Kapitel über die Validzeit-Indexe beschrieben.
Im Folgenden werden Exemplarisch Strukturen, die zum Teil während der Arbeit implementiert wurden, präsentiert.
Grob besteht der nächste Abschnitt aus zwei Teilen. In ersten Teil werden die R-Baum-basierte Techniken, die auf den überalppenden Regionen basieren, präsentiert. Im zweiten Teil werden die B+Baum-Verfahren beschrieben, die sich zur nicht-überlappenden-Regionen Technik mit Datenduplizierung zuordnen lassen.

RTree Verfahren

Im Allgemeinen ist die beste Methode für die effiziente Anfrageverarbeitung von Bereichsanfragen auf den Schlüssel und Zeit, die Daten auf die beiden Dimensionen zu clustern. Prinzipiell werden zwei Grundstrukturen für den Entwurf genommen. Bei der Duplizierung von Daten werden B+Baum-basierte Zugriffstrukturen für die Basistruktur benutzt, und für die überlappenden Regionen die R-Baum-Methoden verwendet.
Für die Verwaltung von historischen Daten und insbesondere für die Anfrageverarbeitung hat diese Struktur folgenden Vorteile, die Anzahl der Daten in den Baum ist proportional zu n. Das Einfügen hat logarithmische Kosten $O(log_b n)$. Bei der Anfrageverarbeitung fallen im schlimmsten Fall zwar die linearen Kosten $O(n/b)$, durchschnittlich sind die aber logarithmisch. Aus der praktischen Sicht existieren für die R-Bäume, Nebenläufigkeits- und Wiederherstellungsalgorithmen, das macht den Einsatz von R-Bäumen in der Praxis attraktiv.
Einen Nachteil haben diese Strukturen, dass die Performanz von Anfragen stark von der Intervall-Längenverteilung abhängt. Falls die Datenmenge eine kleinere Menge von „langlebigen“ Intervallen und vergleichsweise große Menge von „kurzlebigen“ Intervallen besitzt, wirkt das auf die Überlapppungsgrad der Regionen, die durch die Knoten des Baums repräsentiert sind. Entsprechend steigt die Anzahl zu besuchenden Knoten (vgl. Abbildung 3.3).
Um das Problem der Verwaltung von „langlebigen“ Intervallen in einem R-Baum effiziente zur gestalten, wurde eine Indexstruktur *Segment R-Tree* (kurz SR-Tree) vorgeschlagen, diese kombiniert die Idee von den *segment Baum*[48] und R-Baum.
Der Segment Baum ist eine Hauptspeicherstruktur für die Verwaltung von eindimensionalen Intervallen und Beantwortung des Intervallschnitts-(Überlappungs)-Problem. In einer einfachen Version ist sie ein binärer Baum, dessen Blattknoten ,die Endpunkte der Intervallen aus der Intervallmenge und die Innerenknoten die Vereinigungsintervall der Endpunkte von den Kinderknoten, repräsentieren.
Ein Intervall I aus der Datenmenge wird in den höchsten Knoten des Baums gespeichert, so dass I vollständig, den von Knoten repräsentierten Intervall abdeckt. Damit wird ein Intervall unter Umständen an mehreren Stellen des Baums abgespeichert. Es kann aber auch gezeigt werden, dass ein Intervall von höchstens logarithmisch viele Knoten verwaltet werden kann. Das ist auch der Nachteil der Struktur, da die Speicherplatzkosten nicht mehr linear sind. Die Anfrage Kosten sind aber logarithmisch $O(\log n + r)$.
SR-Tree eignet sich sowohl für die Indexierung von Validzeit- als auch für die Transaktionszeitdaten. Die Struktur ist ein R-Baum, bei dem die Intervalldaten sowohl in Blattknoten als auch in den Indexknoten gespeichert werden können. Der Grundidee des Verfahrens ist die „langlebigen“ Intervalle in oberen Levels des Baums zu platzieren, um die Überlappung der Blattknoten zu verringern. Ähnlich dem Segment-Baum Prinzip wird ein Datensatz mit

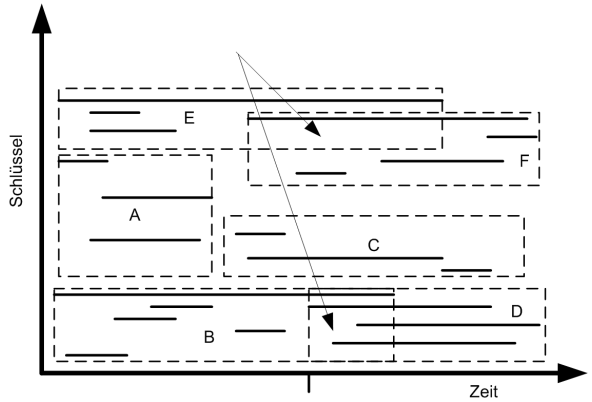

Abbildung 3.3.: Mögliche R-Baum-Regionen-Verteilung. Mit den Pfeilen sind die überlappenden Regionen gekennzeichnet.

Intervall I in den höchsten Knoten gespeichert, falls er den Indexknoten-Intervall vollständig abdeckt, falls es nicht der Fall ist, wird der Datenintervall aufgeteilt und in Kindknoten eingefügt.

Ein Problem bei der Struktur ist, dass durch Verwaltung der Datensätzen in Indexknoten

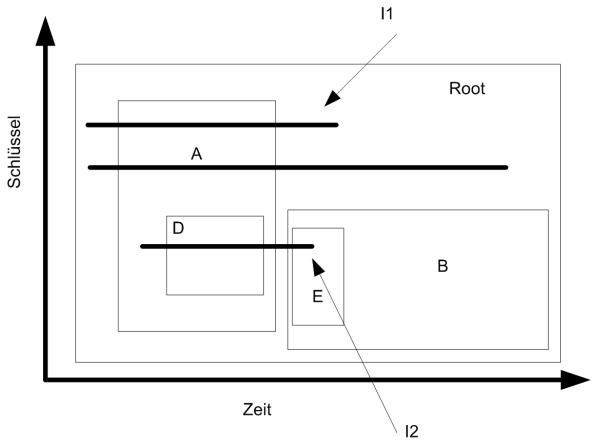

Abbildung 3.4.: SRTree Indexeinträge in dem Wurzelknoten, repräsentiert durch Rechteckregionen. Der Datensatz I1 wird in den Wurzelknoten mit dem Indexeintrag A platziert, der I2 dagegen, obwohl er vollständig den Knoten D abdeckt „aufgeteilt“ und ein Teil in den Knoten A zu dem Eintrag D und der zweiter Teil wird in den Knoten E platziert. Quelle [28]

wenig Platz für die Zeiger auf die Kinderknoten bleibt. Das verringert den Verzweigungsgrad eines Baums. Eine Möglichkeit wäre das Problem im Griff zu bekommen, an den inneren Knoten die Überlaufseiten anzuhängen. Dann aber bei Anfrageverarbeitung müssten diese Seiten zusätzlich in die Speicher geholt werden. Zusätzliche Schwierigkeit bereiten die Einfüge- Update- und Löschalgorithemen,die die Änderungen an den Regionen verursachen, demzufolge auch die Änderungen an Intervallen von inneren Knoten. Zum Beispiel nach einfügen eines Datensatzes in den Blaknotten muss unter Umständen die Region des Blattknoten angepasst und auf den ganzen Pfad propagiert werden, um die Intervallverhältnisse anzupassen. Auch die Split-Algorithmen verursachen die Änderungen an Intervallen.

Ähnlich zu Segment Baum kann ein Intervall an mehreren Stellen des Baum gespeichert wer-

den. Der Speicherplatzverbrauch entartet demzufolge zu $O((n/b)\log_b n)$. Trotz diesen Nachteile erreicht SR-Tree bessere Anfrage Performanz als gewöhnliche RTrees. Bei einer nicht so starken Varianz der Intervalllängen können auch die R-Bäume direkt eingesetzt werden.

B+Tree Verfahren

Die Hauptidee bei den B+Baum-Verfahren ist eine Assoziierung von Seiten mit den disjunkten Rechteck-Regionen des Schlüssel-Zeit-Raums. Die Indexeinträge von B+Baum werden um Zeitintervalle erweitert. Eine der Vorläufer von diesen Strukturen war *Write-Once B-Tree* [28] kurz WOBT, der Index wurde entwickelt für die Verwaltung von Daten, die komplett auf den WORM Speicher Medien verwaltet werden.
In diesem Abschnitt werden Strukturen präsentiert, deren Vorläufer der WOBT ist. In der Praxis haben sich die B+Baum-Verfahren mit partieller Duplizierung durchgesetzt. Die Gründe dafür werden dann in diesem Abschnitt geklärt. Zunächst wenden wir uns der WOBT Architektur.
So bald eine Seite vollständig auf den WORM Datenträger geschrieben wurde, könnte sie nicht mehr verändert werden. Baumknoten in WOBT enthalten die Seiten, die auf dem WORM gespeichert sind. Die Einträge der WOBT-Indexknoten enthalten die Startpunkte ihre Transaktionen, falls die neue Version des Objekts eingefügt wurde, ist der Startzeitstempel von neuen Version der Endzeitspempel der alten „aktuellen“ Version, gemäß der B+Baum einfüge Strategie werden die Versionen mit dem gleichen Schlüssel in den selben Knoten eingefügt.
Beim Überlauf des Knotens wird der Knoten mit aktuellem Transaktionszeitstempel gesplittet (diese Operation wird im weiteren als *Versionsplit* bezeichnet), dabei werden die Knoten entsprechend der WORM Charakteristik samt alle seinen Daten, da nur ein mal geschrieben werden kann, auf seinen Platz gelassen. Die Datensätze, die noch „live“ sind, werden aber in den neuen Knoten kopiert. In Abhängigkeit von der Anzahl der „live“ Versionen, um den wie bei B+Baum Splitstrategie, bestimmte Speicherplazausnutzung der Knoten zu gewährleisten bzw. damit der neue Knoten nicht wieder nach kleineren Anzahl der Einfügeoperationen überläuft, kann der neue Knoten zusätzlich mit dem Schlüssel gesplittet werden (*Schlüssel-Split*). Dabei wird je nach Konstellation der Daten entstehen entweder ein oder zwei neue Knoten. Die Wurzel nach den Aufteilungen können in einem variablen-langen Array verwaltet werden.
Das Löschen von Dateneinträgen erfolgt durch Einfügen eines Löscheversioneintrags des Datensatzes mit Löschzeitstempel, in den aktuellen Knoten, beim Splitten des Knoten werden diese Einträge nicht mehr für das Kopieren berücksichtigt. Auf dieser Weise erfolgt auch eine Clusterung von Daten auf den beiden Dimensionen. Dadurch, dass die Einträge des Baums mit den Zeitstempeln versehen sind und mit der Einführung des Versionssplits, ist die temporale Anfrageverarbeitung gewährleistet.
Die Anfragen können anhand der Schlüssel und Zeit oder nur mit Zeit effizient beantwortet werden. Allerdings für die effiziente Verarbeitung der Anfragen, die Historie eines Schlüssel ab bestimmten Zeizpunkt abfragen, kann die Rückwärtsverlinkung an den einzelnen Datensätzen oder an den Seiten effizient implementiert werden. Die Beantwortung von Bereichsnafragen erfolgt im Allgemeinen in $O(\log_b n + r/b)$. Die Platzausnutzung bei diesem Verfahren ist die $O(n/b)$ Seiten.
Falls der WOBT Technik auf den „normalen“ Datenträger verwaltet wird, werden die Daten manchmal unnötig kopiert. Dieser kleiner „Nachteil“ bei der Verwendung von WOBT in Magnetdatenträger ist durch die WORM Eigenschaft verursacht. Diese fordert immer den Versionssplit bevor den eventuellen Schlüssel-Splits, da WORM Eigenschaft keine andere Möglichkeit zulässt. Dadurch entstehen aus eine Seite drei Seiten, eine historisch und zwei

aktuellen. Dagegen bei den B+Baum entstehen nur zwei. Durch das Erlauben des normalen B+Baum-Schlüssel-Splits kann unter Umständen die Platzausnutzung der Struktur verbessert werden. Im Folgenden wird eine Zugriffstruktur *Time Split BTree*, die in ihre Basis Variante direkte Schlüssel Splits erlaubt, beschrieben. Es wird gezeigt welche Nachteile verbirgt diese Strategie, danach wird die Variante vorgestellt, die in einem Prototypsystem von Microsoft implementiert ist.
Die Basisvariante von TSB-Tree ist die Erweiterung von WOBT-Technik, bei der direkte Schlüssel-Splits erlaubt sind. Im Unterschied zur WOBT werden die Wurzel-Knoten nicht in einen separaten Struktur gespeichert, sondern ähnlich zu B+Baum, wird neue Wurzel Knoten erzeugt und die Höhe des Baum um eins erhöht. Zusätzlicher Unterschied von TSB-Tree zu WOBT und zu den Strukturen, die in den nächsten Abschnitt beschrieben werden, ist, dass bei den Versionsplits die historischen Daten aus dem Knoten kopiert werden, so dass die „live“ Daten verbleiben in dem durch den Überlaufbedingung betroffen Knoten. Das begründet dadurch, dass die historischen Daten direkt auf die langsamen tertiären Speichermedien oder sogar auf die WORMs migriert werden können.
Im Unterschied zu WOBT werden die Indexknoten- und Blattknotensplits unterschiedlich behandelt. Die Indexknoten der TSB-Tree enthalten Einträge, die die Rechteckregionen des Schlüssel-Zeit-Raums repräsentieren. Sowohl die Versionssplits als auch die Schlüsselsplits in Indexknoten können dazu führen, dass die Kinderknoten auf zwei Elternknoten verweisen. Die Splitalgorithmen in den Indexknoten werden so eingeschränkt, dass die „live“ Kinderknoten genau einen Elternknoten besitzen.
Bei dem Überlauf der Blattseiten, falls die Seite eine vorgegebene Schranke an der Anzahl der unterschiedlichen Schlüssel in „live“ Version nicht überschreitet, wird der Versionssplit vorgenommen, im Unterschied zu WOBT darf der Splitzeitstempel kleiner als der aktuelle Transaktionszeitstempel sein. Es kann zum Beispiel der Zeitpunkt des letzten Updates, nach dem nur Einfüge Operationen stattgefunden haben, gewählt werden. Dadurch kann die Trennung zwischen den historischen und die aktuellen Daten ermöglicht werden.
Die Seiten mit großem Anzahl „live“ Datensätzten mit verschiedenen Schlüssel können direkt gemäß B+Baum-Splitstrategie gesplittet werden. Allerdings gibt es ein Problem bei dieser Umsetzung, die Entscheidung über den Split des Knotens nach dem Schlüsselachse, bei dem Zeitpunkt des Überlaufs, wird allein auf der Anzahl der „live“ Einträgen getroffen, das heißt es wird nicht garantiert dass in früheren Zeitpunkten überhaupt die „live“ Daten vorhanden sind 3.5.
Diese Aufteilungsstrategie verschlechtert die temporale Anfrageverarbeitung, insbesondere, wenn viele Einfüge-Operationen stattfinden, werden sehr viele direkte Schlüsselsplits angewendet, so dass viele Knoten zwar „live“ sind, enthalten aber wenig oder sogar keine „live“ Daten zu bestimmten Zeitpunkten, dass erfordert unnötige Besuche der Knoten bei der Traversierungsalgorithmen. Daher der Aufwand für die temporale Suche mit einem Zeitintervall oder einem Zeitpunkt beträgt im schlimmsten Fall nicht die logarithmisch Kosten sondern Lineare $O(n/b)$.
Zu den Nachteilen der Basisvariante des TSB-Tree zählt auch sicherlich, dass es keine Schranken für Zugrirffkosten auf die vergangener Versionen der Daten existieren. Es wird zwar experimentell gezeigt, dass die Durchschnittskosten in logarithmischen Bereich liegen, die Struktur gibt aber keine Garantien für das Worst-Case-Verhalten. Im Folgenden wird eine allgemeine Technik für den Entwurf von Externspeicherzugriffstrukturen für Transaktionszeit-Umgebung vorgestellt, die auch asymptotischen Worst-Case-Schranken für den Zugriff auf jeder Version besitzt [31] und lineare Speicherplatz erfordert.
Im Weiteren wird die prinzipielle Technik mit kurze Beschreibung der Einfüge und Löschalgortihmen beschrieben, die detaillierte Beschreibung der Implementierung von MVBT folgt im nächsten Kapitel.

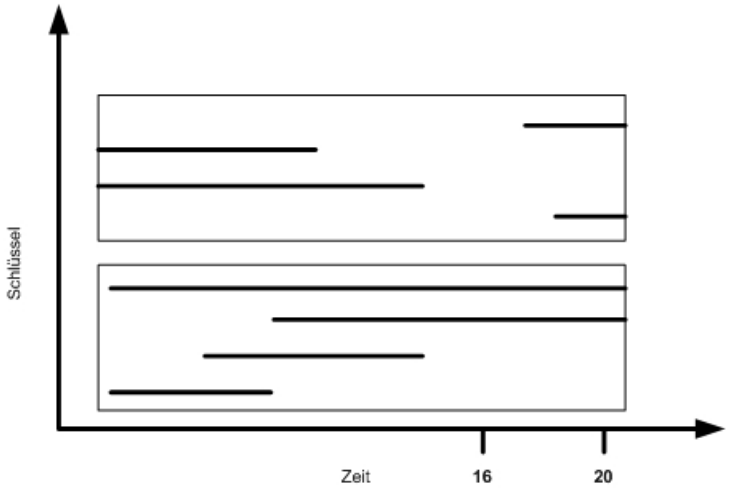

Abbildung 3.5.: Problem bei dem Schlüssel-Split in TSB-Tree. Der Split fand zum Zeitpunkt $t = 20$, in beiden Knoten sind zu diesem Zeitpunkt jeweils 2 „live“ Intervalle vorhanden. Dagegen zum Zeitpunkt $t = 16$ der obere Knoten enthält überhaupt keine „live“ Objekten. Quelle [28]

Multiversion BTree

Im [31] wurde generelle Technik für die Erweiterung von hierarchischen Zugriffstrukturen ohne Verlust der asymptotischen Zugriffschranken auf der Basis Struktur entwickelt. Dabei wurde das Verfahren für die Erweiterung von Haupspeicherstrutuen von [34] auf externen Speicherstrukturen als Grundlage verwendet.
Bei diesem Verfahren war es möglich die asymptotischen Schranken der zu erweiternde Struktur im multiversionen Verhalten beizubehalten. Exemplarisch wurde der B+Baum als Ausgangstruktur genommen, da B+Baum nicht nur das „Arbeitspferd“[46] in RDBMS ist, sondern, dass der gute asymptotische Worst-Case-Such- und Zugriffskosten besitzt.
Für die Entwicklung asymptotisch-optimaler Multiversion-Struktur auf der Basis des B+Baums werden folgende Anforderungen gestellt: Sei N die Anzahl der datenmanipulierenden (Einfüge- und Lösch) Operationen in Transaktionszeit-Umgebung, die auf eine leere Struktur angewendet wurden. Sei i Versionsnummer (z.B. Transaktionszeitstempel der Operation) mit $0 \leq i \leq N$. Sei m_i die Anzahl der Datenelemente in dem Index nach der i Operation. Dann wird es gefordert, dass der Multiversion BTree folgende Schranken erfüllt:

- für die ersten i Versionen wird im schlimmsten Fall $O(m_i/b)$ Platz verwendet;
- die $(i + 1)$ Operation verursacht im schlimmsten Fall $O(\log_b m_i)$ Kosten;
- die exakte Suche in Version i besitzt im schlimmsten Fall $O(\log_b m_i)$ Aufwand;
- die Bereichssuche in Version i erfordert im schlimmsten Fall den Aufwand von $O(\log_b m_i + r_i/b)$

Die Struktur soll das Gefühl vermitteln, dass die Daten in einzelnen Versionen in separaten B+Bäumen organisiert sind.
Die Organisation der Mutliversion BTree ähnelt der Organisation von WOBT und TSB-Tree. Datenelemente, die in MVBT in Blattknoten verwlatet werden, sind erweitert um den Zeitintervall, im Allgemeinen kann der Dateneintrag wie folgt präsentiert werden *(Schlüssel, Einfügeversion, Löschversion, Informationsteil)*. Solange der Datenlement noch nicht gelöscht

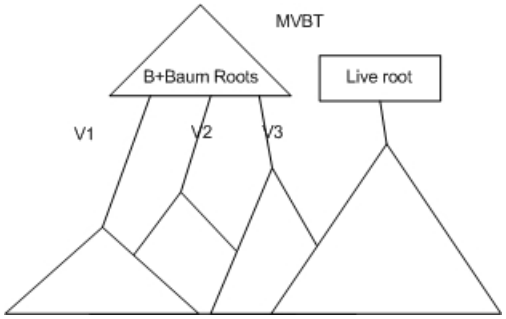

Abbildung 3.6.: MVBT Gesamt-Architektur.

ist, kann für die Löschversion eine spezielle Markierung verwendet werden. Die Indexeinträge der MVBT sind ähnlich den Dateneinträgen organisiert, mit dem Unterschied, dass der *Infromationsteil* durch den Zeiger auf den Kindknoten ersetzt ist. Es ist zu beachten, dass die Indexeinträge die disjunkten Rechteckregionen der Schlüssel-Zeit repräsentieren.
Die Zeitgrenze des Rechtecks von dem Indexeintrags ist explizit durch die Einfüge- und Löschversion gegeben, dagegen der Schlüssel-Bereich ähnlich dem B+Baum zusammen aus dem Nachbarschlüssel aufgebaut ist. Der Knoten dessen Intervall von dem Indexeintrag noch nicht durch das Löschzeitstempel abgeschlossen ist, wird als „live“ bezeichnet. Wie es in späteren Kapitel erklärt wird, Repräsentation der Schlüssel-Zeit Regionen durch Rechtecke spielt eine sehr wichtige Rolle bei der Anfrageverarbeitung.
Um ein asymptotischen Worst-Case-Schranken zu gewährleisten werden Bedingungen an der Mindestanzahl der gültigen („live“) Daten in eine Version i in den Knoten des Baums benötigt. Dabei wird folgendes Modell verwendet, sei B maximale Anzahl der Datensätzen, die in einer Speicherseite passen, dann wird es gefordert, dass mindestens d oder 0 Datensätzen von Version i in eine Seite vorliegen, wobei $B = k \cdot d$ für einen k ist. Diese Bedingung wird als *schwache Versions-Bedingung* (eng. weak version condition) bezeichnet. Sowohl die Einfüge- als auch die Löschvorgänge können in MVBT die strukturellen Veränderungen der Struktur bewirken.
Die Einfüge Operationen führen zur physikalischen Überlauf der Seiten. Dagegen die Löschvorgänge führen in einer Transaktionsumgebung zum keinen physikalischen Unterlauf der Seiten, allerdings können sie die Bedingung an der Mindestanzahl der „live“ Daten zur Version i verletzen. Die Konstellation bei der, nach einer Löschoperation, die schwache Versions-Bedingung verletzt wurde, wird als *schwache Versionsunterlauf* (eng. weak version underflow) bezeichnet.
Ähnlich zur WOBT werden auch in den MVBT Versionssplit mit anschließendem Kopiervorgang durchgeführt (im Unterschied zu TSB-Tree sind die direkten Schlüsselsplits nicht erlaubt). Dabei ist der entscheidende Unterschied zur WOBT und TSB-Tree ist die Einführung vom speziellen Kontrollparameter $e \cdot d + 1$, mit dem die Mindestanzahl der Einfüge- oder Löschoperationen voraussetzten lässt, die benötigt werden, um den Seitenüberlauf oder Verletzung der schwachen Versionsbedingung zu bewirken.
Die Begründung ist, durch die Tatsache, dass nach einem Versionssplit und anschließendem Kopiervorgang, entweder die neue Seite fast voll sein kann, so dass die anschließendes Einfügen wieder einen Versionssplit verursacht(im schlimmsten Fall würde das pro Einfügeoperation $O(1)$ Seitenplatzverbarauch-Kosten bewirken), oder die Seite fast d Einträge enthält, so dass die anschließende Löschoperation die Bedingung an der Mindestanzahl der „live“ Elementen verletzt.
Ausgehend von diesem Parameter wird nach jedem Versionssplit gefordert, dass die Anzahl der „live“ Daten in dem neuen Knoten zwischen $(1+e) \cdot d$ und $(k-e) \cdot d$ liegt. Diese Forderung

wird als *starke Versionsbedingung* (eng. strong version condition) genannt. Dabei wird die Verletzung der oberen Grenze durch die *starke Versionsüberlauf* (eng. strong version overflow) und die Unterschreitung der unteren Grenze durch *starke Versionunterlauf* (eng. strong version underflow) bezeichnet. Diese zwei Bedingungen steuern die strukturellen Veränderungen in MVBT.
Im Folgenden werden kurz die Algorithmen für das Einfügen und Löschen von Daten in von MVBT beschrieben, die detaillierte Beschreibung von Algorithmen folgt auch im nächsten Kapitel.
Das Einfügen eines „live“ Datensatzes s mit aktuelle Version i erfolgt ähnlich zu B+Tree Einfügealgorithmus, es wird zuerst der „live“ Blattknoten A anhand des Schlüssel aufgesucht, falls keine Überlauf der Seite stattfindet wird der Datensatz der Seite hinzugefügt.
Andernfalls wird ein Versionssplit an der Seite A vorgenommen, dabei wird ein neue Knoten B die kopierte „live“ erhält erzeugt. An dieser Stelle wird geprüft ob die Anzahl der „live“ Daten entweder die starke Versionsüberlauf verletzt, oder den starken Versionunterlauf.
Andernfalls wird der Indexintervall des Knoten A abgeschlossen, und eine Indexeintrag für den Knoten B mit dem „live“ Intervall mit Beginversion i erstellt. Anschließend werden die Änderungen an den Elternknoten angewendet und dass Einfügealgortihmus kann rekursiv nach oben propagiert werden.
Fall jedoch die Grenze von $(k-e)\cdot d$ überschritten wurde, wird auf den Knoten B Schlüsselsplit

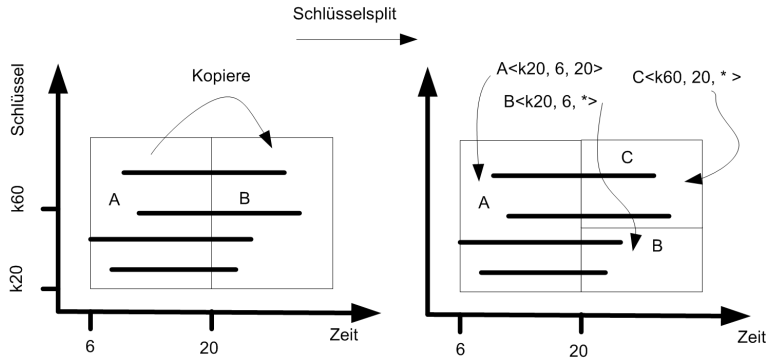

Abbildung 3.7.: Überlaufbehandlung. Schlüsselsplit nachdem die starke Versionsüberlauf-Bedingung verletzt wurde. Zuerst wurde der Versionssplit ausgeführt und die „live“ Daten in den neuen Knoten kopiert, nach dem Kopiervorgang war die Anzahl die Daten in dem Knoten B größer als $(k-e)\cdot d+1$. Aus diesem Grund wurde der B+Baum-Schlüssel-Split angewendet

angewendet. Die Operation erzeugt einen neuen Knoten C, durch die spezielle Wahl der Kontrollparameter ist es garantiert, dass die beiden Knoten die Mindestschranke an der Anzahl der „live“ Daten erfüllen. Im Anschluss werden ähnlich dem normalen Fall Änderungen in den Elternknoten propagiert.
Falls die Grenze unterschritten ist, wird ein Merge von dem Knoten B mit einem Nachbar-Knoten von A durchgeführt. Im Allgemeinen der Vorgang ähnelt dem Merge von dem B+Tree. Durch die Bedingung an der Mindestanzahl der „live“ Daten, besitzt der Nachbarknoten D mindestens $(1+e)\cdot d$ „live“ Einträge. Auf den Nachbarknoten D wird eine Versionssplit durchgeführt die „live“ Einträge werden in den Knoten E Kopiert und mit dem Knoten B verschmolzen. Durch das Verschmelzen der Daten in dem Knoten B kann die starke Versionsüberlauf-Bedingung verletzt werden, falls dieser Fall auftritt, wird der Schlüsselsplit angewendet.

Das Löschalgoritmus findet zuerst den Knoten, in dem sich zu löschendes Element befindet

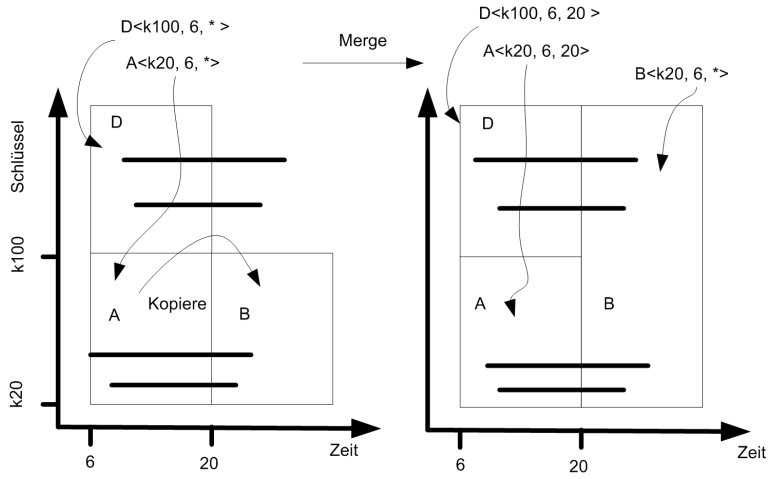

Abbildung 3.8.: Überlaufbehandlung. Merge falls die starke Versionsunterlauf Bedingung verletzt wurde. Nach dem Versiosplit (Kopiervorgang der „live“ Daten) enthält der Knoten B weniger als $(1 + e) \cdot e$ Elemente. Es wird nach dem Schlüssel-Nachbar D von dem Knoten A gesucht. Auf den Knoten D wird zuerst ein Versionsplit angewendet und der Knoten wird mit der gleicher Version wie Knoten A abgeschlossen. Die „live“ Daten werden in den Knoten B aus dem zwischen Knoten kopiert. An dieser stellt kann ein weitere Schlüsselsplit folgen, falls die Anzahl von „live“ Elemnten in B größer als $(k - e) \cdot d + 1$ ist.

und setzt den Endpunkt des assoziierten Intervalls auf die Löschverison. Dabei wird geprüft ob die starke Versionunterlauf Bedingung verletzt ist, falls dieser Fall auftritt, wird ein Merge ähnlich wie beim Einfügalgortihmus nach dem Versionssplit durchgeführt.
Ähnlich zu TSB-Tree und WOBT die Versionsplits mit Kopiervorgängen in den Indexknoten, führen dazu, dass die Kindknoten mehrere Elternknoten besitzen. Daher ist die Struktur des MVBT kein Baum mehr, sondern ein direkte azyklische Graph. Durch die Kopiermethoden entstehen auch die neuen Wurzelknoten für vergangene Versionen.
Im Unterschied zu WOBT, in dem für die einzelne Wurzelknoten ein Array bereitgestellt wurde, werden sie in MVBT in einem zusätzlichen B+Baum gespeichert. Im Weiteren wird dieser B+Baum als *Root-Tree* bezeichnet. Die Einträge der Root-Tree werden über die Zeitintervalle indexiert (Anfangspunkt des assoziierten Intervalls), dadurch wird effizienter Einstieg zu den vergangenen Versionen ermöglicht (vgl. Abbildung 3.6).
Es bleibt noch zu klären, wie sich die Parameter k und e berechnen lassen. In einem B+Baum der Füllungsfaktor c der z.B. im Bereich $2/B \leq c \leq 0.5$ liegt, garantiert, dass nach einer Splitoperation der neue Knoten $c \cdot B$ Dateneinträge enthält.
Daher um zu gewährleisten, dass nach einem Schlüsselsplit in MVBT die neue Knoten, ähnlich wie nach einem B+Baum-Split, mindestens $(1 + e) \cdot d$ „live“ Daten enthalten, unterliegt der Parameter k der folgenden Bedingung

$$(k - e) \cdot d + 1 \geq \frac{1}{c} \cdot (1 + e) \cdot d$$

(Vor dem Schlüsselsplit enthält der MVBT-Knoten, falls der die starke Versionsüberlauf Bedingung verletzt, mindestens $(k - e) \cdot d + 1$, nach einer Schlüsselsplitoperation wird gefordert

dass die beiden Knoten mindestens $(1+e)\cdot d$ Daten enthalten). Nach der Auflösung nach k muss die Ungleichung $k \geq \frac{1}{c} + (1+\frac{1}{c})\cdot e - \frac{1}{d}$ erfüllt sein.
Wird z.B. für Füllungsfaktor c Standardwert 0.5 angenommen, dann unterliegt k folgender Einschränkung $k \geq 2 + 3\cdot e - \frac{1}{d}$.
Der Parameter e lässt sich aus der folgende Überlegung berechnen. Nach einer Merge Operation wird gefordert, dass die Anzahl der „live“ Daten, in dem durch Merge entstandenen Knoten, die untere Grenze $(1+e)\cdot d$ nicht unterschreitet. Vor dem Mergevorgang sind es mindestens $2\cdot d-1$ „live“ Daten in den beiden in der Merge-Operation teilnehmenden Knoten. Daher muss folgende Ungelichung erfüllt sein:

$$(2\cdot d-1) \geq (1+e)\cdot d \quad \text{dann nach der Umformung} \quad e \leq 1 - \frac{1}{d}$$

In [31] wird gezeigt dass die MVBT Struktur erfüllt die gestellten Anforderungen an Worst-Case Schranken für die Anfrageverarbeitung in Version i:

- Anzahl der Seitenzugriffe bei der Suche nach einem Datenelement in Version i ist durch die Schranke von $O(\lceil \log_d m_i \rceil)$ im schlimmsten Fall beschränkt.
- Anzahl der Seitenzugriffe bei der Bereichssuche nach r - Datenelementen in Version i ist durch $O(\lceil \log_d m_i \rceil + r/d)$ im schlimmsten Fall.
- Anzahl der Seitenzugriffe für Tranksationsoperatrion für $(i+1)$ Version ist durch $O(5\cdot \lceil \log_d m_i \rceil)$ im Schlimmsten Fall beschränkt.

In [31] wurde die Kostenanalyse für den Speicherplatzverbrauch durchgeführt. Dabei wurden amortisierten Kosten der einzelnen Operation der Splitallgortimus berechnet, bei der Berechnung der Kosten wurden die Blattseiten und Indexseiten getrennt behandelt. Die Ergebnisse der Analyse sind im Folgenden zusammengefasst:

- Die Worst-Case amortisierte Speicherplatzverbrachkosten für einzelne Einfüge-oder Löschoperation sind für die ganze Struktur $O(1)$ falls $d \geq \frac{2}{e}$

Gesamt Ergebnisse der beiden Analysen werden in folgenden Satz zusammengefasst: *Multiversion BTree, der durch die Erweiterung von einem B+Tree, mit beschriebenen Verfahren konstruiert wurde, asymptotisch optimal in Zeit und Speicherplatzverbrauch für beschriebene Operationen.*
In [31] wurde die Technik erläutert wie die Externspeicherindexstrukturen auf multiversionen Verhalten erweitert werden können und zu gleich die asymptotischen Zugriffkosten Kosten beibehalten werden können. Mit vorgestelltem Verfahren lassen sich nicht nur B+Bäume sonder auch andere Strukturen umstellen.
Im folgendem wird eine Zugriffstrutur, die nach selbe Technik implementiert wurde, präsentiert. Allerdings verbessert die Struktur, die auch B+Baum als Basis Struktur verwendet, die theoretischen Schranken für Speicherplatz- und Zugriffkosten von MVBT.

MVAS

Ähnlich wie in MVBT werden für die Kontrolle über die Split- und Meregalgorithmen zwei Kontrollparameter eingeführt T_h und T_l diese Parameter sind vergleichbar mit Schranken $(1+e)\cdot d$ und $(k-e)\cdot d$ von MVBT. *Multi Version Access Strucure* kurz MVAS von [33] unterscheidet sich hauptsächlich von MVBT durch die unterschiedliche Behandlung der Kontrollparameter beim Split- und Mergeoperationen.
Zunächst wird der Fall betrachtet bei dem der physikalische Überlauf des Knoten A stattfindet. Als erstes wird, wie in MVBT, ein Versionsplit durchgeführt, dabei werden die „live“

Daten in den „zwischen“ Knoten B kopiert. Mit N wird die Anzahl der „live“ Daten einschließlich den Datensatz, der den Überlauf verursacht hat, gekennzeichnet.
Anhand der eingeführten Schranken wird entschieden, ob eine weitere strukturelle Operation (Merge- oder SchlüsselSplit) notwendig ist oder nicht. Die Entscheidung wird anhand von drei folgenden Bedingungen getroffen:

1. Falls $N \geq T_h$ wird ein Schlüssel-Split ausgeführt.

2. Falls $2 \cdot T_l \leq N \leq T_h$, wird keine weitere Operation benötigt, somit verbleiben die Daten in dem Knoten B.

3. Falls $N \leq 2 \cdot T_l$ wird ähnlich dem MVBT Verfahren wird der Knoten B mit Nachbarknoten gemergt, anschließend wird auf den Fall 1 und Fall 2 getestet.

Die Vorgehensweise beim Überlauf ist fast identisch mit MVBT Überlaufbehandlung, dagegen die Behandlung des starken Versionsunterlaufs bei der Löschoperation, erweist mehrere Unterschiede.
Sei A Knoten der nach eine Löschoperation die starke Vesionsunterlauf Bedingung verletzt. Das ist der Fall in MVAS, wenn die Anzahl der „live“ Daten in den Knoten kleine als T_l ist. Ähnlich zu MVBT wird ein Nachbar Knoten B gesucht mit N_l „live“ Daten. Sei B so ein Knoten, mit N_g wird die Gesamtanzahl der Daten in B bezeichnet.
Für das weitere Vorgehen werden drei folgende Fälle betrachtet:

1. Falls $N_g \geq 3 \cdot T_l$, erzeuge einen neuen Knoten C, kopiere alle „live“ Daten aus A in C, kopiere die T_l „live“ Einträge mit benachbarten Schlüssel aus B in C, markiere diese Schlüssel Bereich in B als Kopiert.

2. Falls $N_l < 3 \cdot T_l$ und $N_g > (T_h + 1)$, erzeuge einen neuen Knoten C, kopiere „live“ Daten aus A in C, führe den Versionsplit an Knoten B, wobei alle „live“ Daten werden in den Knoten C Kopiert. Dieser Strategie ist ähnlich dem MVBT-Merge.

3. Falls $N_l < 3 \cdot T_l$ und $N_g \leq (T_h + 1)$, kopiere T_l „live“ Daten von A nach B, markiere den Knoten A als gelöscht.

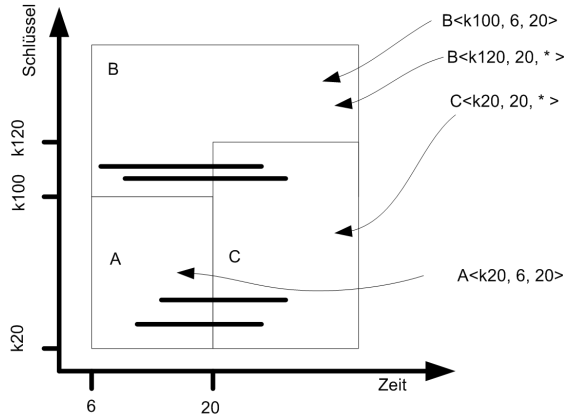

Abbildung 3.9.: MVAS: Die Abbildung veranschaulicht die Situation nach dem Merge-Schritt: Fall 1. Knoten B bekommt einen zusätzlichen Indexeintrag. Das Intervall des alten Eintrags wird abgeschlossen. Es wird keine weitere Seite für die „live“ Daten aus B benötigt.

An dieser Stelle soll betont werden, dass es kein Versionssplit an Knoten B im Fall 1 durchgeführt wird, die Daten werden lediglich nur kopiert, der Zeitintervall des Indexeintrags von den Knoten B bleibt zwar unverändert, allerdings durch das Kopieren eines Schlüssel-Bereichs kann sich die Schlüssel-Grenze des Knoten verschieben, falls dieser Fall auftritt muss der Indexeintrag angepasst werden. Um zu gewährleisten, dass die z.B. Bereichanfragen, die in von dem Löschzeitpunkt der Knoten A starten und die kopierte Elemente der Knoten B überlappen, in den Knoten C landen, wird der alte Indexeintrag auf B mit dem Löschzeitpunkt von A beendet und ein neuer Eintrag für denselben Knoten B eingeführt mit angepasste Schlüssel Grenze und einem Beginzeitpunkt als aktuelle Version.
Auch im Fall 3 muss die Grenze angepasst werden um die kopierte Elemente aus A wieder

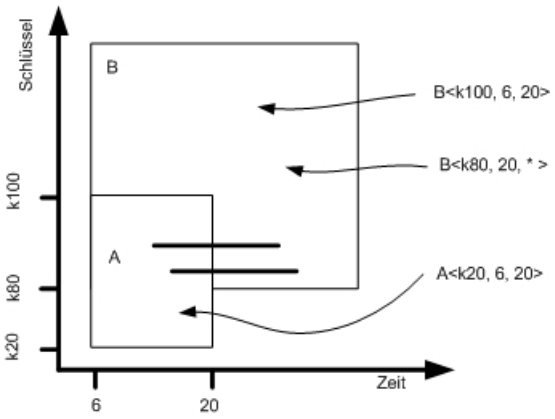

Abbildung 3.10.: Die Abbildung veranschaulicht die Situation nach dem Merge Schritt: Fall 3. In diesem Fall, wird überhaupt keine neuen Seite angelegt.

zu finden. Wie auf den Abbildung 3.2.1 und 3.2.1 zu sehen ist durch die diese Mergestrategie entstehen die Regionen der Schlüssel-Zeitraum die nicht mehr Rechteck sind, das wie im weiteren Verlauf der Arbeit beschrieben wird, erschwert die Implementierung der historischen Bereichsanfragen.
Im MVAS [33] werden für T_h und T_l folgende werte genommen. $T_h = 4 \cdot \frac{B}{5}$ und für $T_l = \frac{B}{5} - 1$. Aus diese Forderung resultiert, dass der Parameter d (Bedingung an Mindestanzahl der „live“ Daten in Version i) von MVBT ist in MVAS mindestens $d = \frac{B}{5}$. Mit der Wahl von diesen Parameterwerten und beschriebenen Merge- und Splitverfahren wird in MVAS bessere Speicherplatzausnutzung erreicht (es entstehen weniger Knoten beim Merge-Schritt). Die gesamt Speicherplatzkosten können bis zu 30% gesenkt werden, im Vergleich zu MVBT[33].
Im nächsten Abschnitt wird die Erweiterung der TSB-Tree beschrieben die zwar keine Garantien für die Laufzeit und Speicherplatzausnutzung gibt. Im Schnitt zeigt aber gute Performanz und ist implementiert in einem Prototyp System von Microsoft.

TSB-Tree in ImmortalDB

In der aktuellen Variante des TSB-Trees [30] wurde die Struktur gründlich überarbeitet. Im Folgendem werden die wesentlichen Veränderungen und Besonderheiten, die für den Entwurf von Transaktionszeitstrukturen auf Basis der B+Baums von Interesse sind, präsentiert. Die Gesamtstruktur basiert auf der B+Baum Implementierung von MS SQL Server.
Die Seiten in TSB-Tree wurden so entworfen, dass es die Rückwärtskompatibilität mit B+Baum von SQL Server ermöglicht wurde.
Eine Speicherseite, die den Blattknoten des TSB-Tree repräsentiert, enthält neben dem Sys-

teminformationteil einen Array von Zeiger, die auf die durch Zeigern verlinkte Datensätze zeigen. Die Datensätze sind rückwärts, wie bei der *Reverse Chaining* Methode miteinander verkettet, so dass die aktuellste Version eines Dateneintrags direkt über den Array erreichbar ist. Diese Organisation der Seiten ermöglicht Abschirmung der historischen Daten und effiziente Ausführung von *transaction pure-Key* Anfragen. Außerdem dieses Seiten-Layout ermöglicht dem B+Baum-Index die Seiten als Blattknoten ohne den Transaktionssemantik zu benutzten. Das Löschen der Datensätze erfolgt durch spezielle Markierung.
Die Speicherseiten von dem Indexknoten enthalten spezielle Einträge die neben dem Zeiger auf den Kindknoten auch die Beschreibung der Rechteckregionen der Schlüssel-Zeit Raum enthalten. Wobei die Rechtecke werden nur durch den minimalen Schlüssel und den minimalen Zeitstempel repräsentiert. Die Rekonstruktion des kompletten Regions erfolgt durch die geschickte Verkettung der Indexeinträgen.
Ähnlich zur der Datenseite enthält die Indexseite ein Array mit den Zeiger, die auf die aktuellen Indexeinträge zeigen, was wiederum die Rückwärtskompatibilität mit den B+Baum-Seiten von SQL Server ermöglicht(vgl. Abbildung 3.2.1).
Einer der Hauptunterschiede zu WOBT, MVBT und MVAS, der auch von Autoren aus

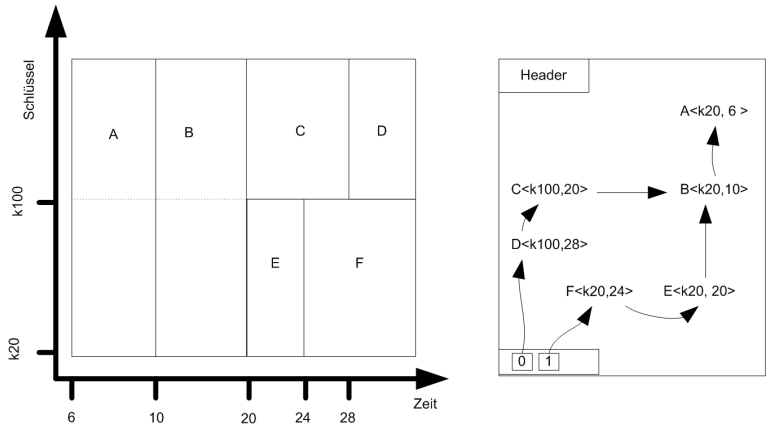

Abbildung 3.11.: Links ist sind die Rechteckigeregionen der Blattknoten, und rechts die zugehörige Indexseite dargestellt. Die punktierte Linie zeigt die mögliche Schlüssel-Splitgrenze von der Region. Dabei ist das Problem, dass die Indexenträge von Knoten A und B nach einem Schlüsselsplit in beiden neuen Knoten platziert werden müssen. Quelle [30]

[30] hervorgehoben wird, ist, dass schon in früheren Variante des TSB-Tree implementierte Verionssplit die historischen Daten in die neue Seite kopiert, so dass die "live" Daten in den aktuellen Knoten verbleiben (Nach der Argumentation der Autoren, diese Strategie ermöglicht bessere Migration der historischen Daten z.B. auf langsamere Speichermedien). Im Weiteren werden die Splitstrategien der neuen TSB-Tree vorgestellt.
Ähnlich zur der Basisvariante des TSB-Trees werden auf die Blattknoten und Indexknoten verschiedene Splitstrategien angewendet. Einer der Unterschiede zur der Basisvariante ist, dass auf den Blattknoten keinen direkten Schlüsselsplit mehr erlaubt ist. Ähnlich wie in WOBT, MVBT und MVAS erfolgt vor dem Schlüssel-Split immer der Versionssplit und zwar mit dem aktuellen Transaktionszeitstempel. Die Entscheidung über den weiteren Schlüsselsplits erfolgt anhand vorgegebener Schranke. Falls die Anzahl der „live" Daten einer Seite nach dem Kopiervorgang diese Grenze überschreitet, erfolgt der Schlüsselsplit.
Es wird in [30] ähnlich argumentiert, dass der Wert von der Schranke nicht nah am 100% der Seiteninhalt liegen darf. Experimentel wurde ermittel, dass der beste Wert für diese Parame-

ter etwa bei 0.67 liegen soll. Im Unterschied zu MVBT und MVAS wird keine Mergetechnik benutzt, demzufolge TSB stellt keine Garantien für den Mindestanzahl der „live“ Daten pro Knoten.
Die Überlaufbehandlung der Indexknoten unterscheidet sich im erste Linie durch direkten Schlüsselsplit und durch die Besonderheit, dass der Versionssplit nicht durch die aktuellen Zeitstempel der Einfüge- oder Löschopreation festgesetzt (vgl. MVBT, MVAS) wird, sondern frei wählbar ist. Genau wie in der Basisvariante der TSB-Tree wird es versucht die Indexseiten so aufzuteilen, dass die Datenseiten mit „live“ Daten genau einen Elterknoten bekommen. Diese Invariante ermöglicht die Implementierung von effizienten Nebenläufigkeitsalgorithmen für Transaktionen.
Die Wahl zwischen den Versionssplit und direkten Schlüsselsplit wird ähnlich den Datenseiten durch speziellen Kontrollparameter getroffen.
Der direkte Schlüsselsplit wird ähnlich dem B+Tree Split durchgeführt. Allerdings die Knotenregionen ,die den Schlüsselsplit Grenze schneiden, sollen in den beiden Knoten kopiert werden, da sonst die Daten nicht mehr auffindbar sein werden. Auf Abbildung 3.2.1 Schneiden die Rechtecke A und B den Splitschlüssel $k100$ daher müssen die Indexeinträge von diesen Regionen in den beiden Knoten vorhanden sein. Durch die unvermeidbare Datenduplizierung reduziert sich die Verzweigungsgrad des TSB-Trees.
Die Versionsplits, durch die Einschränkung, dass die „live“ Knoten nur einen Elternknoten haben dürfen, sind nicht immer möglich. In Abbildung 3.2.1, falls es mit einem Zeitpunkt $t = 28$ gesplittet wird, muss der „live“ Knoten F in den beiden resultierenden Knoten vorkommen, als Ergebnis müssten dann die historische Indexseite, die einen Kopierten Indexeintrag von F enthält, und die aktuelle Indexseite, eventuell bei einer späteren Updateoperation mit aktualisiert werden.
Bei der Umsetzung der Nebenläufigkeit auf dem TSB-Tree müsste dann die historische Indexsetite gesperrt werden, dass allerdings erschwert die praktische Umsetzung des Verfahrens. Deshalb wird in der neuen TSB-Tree folgende Strategie für den Versionssplit verwendet:
Zuerst wird versucht die Zeitgrenze zu finden, die keinen den Knoten schneidet, falls solche Aufteilung existiert, wird der Knoten in den historischen und den aktuellen gesplittet. Falls jedoch keine solche Trennung des Schlüssel-Zeit-Raums gibt, wird der direkte Schlüsselsplit benutzt.
Um den Verzweigungsgrad zu steigern wird in den TSB-Tree spezielle Kompressionstechnik verwendet. Die Elemente in den WOBT, MVBT, MVAS und TSB-Tree sind geclustert sowohl auf Schlüssel- als auch auf Zeitdimension. Durch die Verlinkung der vergangen Versionen eines Datensatzes in TSB-Tree Seiten ist es möglich die Daten effizient zu komprimieren, da im Allgemeinen die Unterschiede zwischen der Versionen eines Datensatzes geringfügig sind. Daher wird in TSB-Tree die *Delta-Kompressionstechnik* verwendet.
Im Großen und Ganzen stellt die TSB eine alternative Verwaltungstechnik zu MVBT und MVAS. Die RDF Daten können als Datenbanktupel mit Zusammengesetztenschlüssel über drei Attributen (S, P, O) von vorgestellten Verfahren indexiert werden. Die Tatsache, dass alle drei Verfahren erweitern den B+Baum spricht auch dafür. Ein kleinen praktischen Vorteil ist beim TSB festzustellen, die effiziente Unterstützung der Mehrtransaktionenbetrieb und Recovery, aber auf den Kosten der Anfrage und Platzeffizienz. Es gibt aber auch für MVBT einen Verfahren für Mehrtransaktionenbetrieb [32].

3.2.2. Validzeit-Indexstrukturen

Aus der Sicht der Verwaltung der temporalen Daten ist die Behandlung der Validzeit viel komplexer im Vergliech zu Transaktionszeit. Die Validtzeitdatenbanken fordern immer die Gewährleistung der temporalen Integritätsbedingungen, dementsprechend gestaltet sich die

Anfrageverarbeitung viel schwieriger.
Dagegen die Zugriffstrukturen für die Validzeit sind viel einfacher zu implementieren. Die Hauptanfoderung ist die effiziente Verwaltung von Intervall-Objekten und insbesondere Unterstützung der typischen Intervall Anfragen wie z.B. Schnitt.
Die Validzeitstrukturen können ähnlich den Transaktionszeitzugriffstrukturen nach der Unterstützung von bestimmten Anfragetypen klassifiziert werden. Dabei kann dieselben Anfragetypen verwendet werden wie in Transaktionszeitumgebung. Allerdings spielen die Zeitabfragen, die z.B. nach der Gültigkeit eines RDF Tripels zum Zeitpunkt t , so wie Schlüssel-Zeit Bereichsanfragen (z.B. ob ein RDF Teilgraph in einem bestimmten Zeitintervall gültig ist) eine große Rolle.
Bei der Verarbeitung von temporalen Anfragen, wie das Differenzberechnung-Beispiel aus vorigem Kapitel zeigt, erfordert, dass das Ergebnis der temporalen Operatoren auch eine temporal-gültige Relation ist. Bei dem Punkt-Modell für die Semantik von Intervallen werden bei den temporalen Operatoren oft viele Intervallschnitte berechnet. Durch die Verwendung der effizienten Indexstruktur für Intarvallobjekten kann unter anderem die Effizienz der Operatoren verbessert werden.
Die Validzeit Indexe lassen sich in zwei Gruppen unterteilen.

- Zugriffstrukturen die ausschließlich eindimensionale Intervalle verwalten.
- Strukturen für die Unterstützung von Schlüssel-Zeit-Bereichsanfragen.

Die ersten hauptsächlich basieren auf die Erweiterung der Hauptspeicherstrukturen wie *Segment Tree, Interval Tree, Priority-Search Tree* für die Verwendung in Externspeicherumgebung. Auch der R-Baum kann für die Verwaltung von eindimensionalen Intervallen eingesetzt werden, als ein eindimensionaler R-Baum.
Der Entwurf der Indexstrukturen für Validzeit Daten wird auf ähnlicher Weise, wie bei Transaktionszeitindexen durch unterschiedliche Längenverteilung der Zeitintervallen beeinflusst. Insbesondere wirkt das auf die Performanz von R-Baum-basierten Techniken, auf Grund des großen Überlappungsgrads der Knoten.
Noch besser eignet sich der R-Baum für die Indexierung, wenn Schlüssel-Zeit Bereichsnafragen dominieren. Sowohl bei der eindimensionale als auch bei zweidimensionalen Schlüssel-Zeit Indexen die unterschiedliche Intervalllängenverteilungen verschlechtern die Performanz der Anfragen und sind im Allgemeinen schwer zu handhaben. Eine grundsätzliche Idee für das Design von sowohl eindimensionalen als auch zweidimensionalen Strukturen, die die Auswirkung von unterschiedlich verteilten Intervalllängen auf die Anfrageverarbeitung mindert, besteht in der Verwaltung von Intervallen in verschiedenen Levels eine Indexstrukur, in Abhängigkeit von ihren Längen [29]. Je länger der zeitliche Intervall ist desto nähe an Wurzel der hierarchischen Indexstruktur wird dieser platziert. Für die Implementierung von dieser Idee müssen die meisten Indexstrukturen für Externspeicher überarbeitet und z.B. die Restriktion, dass nur die Blattknoten die Daten enthalten, aufgegeben werden. Wie schon beim *SR-Tree* verbirgt das die Probleme mit Platzverwaltung in Indexknoten und Datenduplizierung. Falls aber die Intervalllängen nicht so stark variieren, zeigt der R-Baum sehr gute Ergebnisse.
Eine der Erweiterung des *Segment-Tree*s auf Externenspeicher wurde schon in letzten Abschnitt vorgestellt (*SR-Tree*). Eins der Probleme bei der Segment-Tree ist die Platzeffizienz, die ist nämlich nicht linear. Eine effizientere Lösung des Intervallsproblems in Hauptspeicher ist der *Interval-Tree* [48].
Der Interval-Tree kommt im Unterschied zu Segment-Tree mit linearem Speicherplatzvebrauch aus. Der Interval-Tree in einer einfachen Variante ist eine halb-dynamische Struktur, die aus zwei Strukturen besteht, die erste stellt einen vollständigen Binären Suchbaum dar. Dieser wird aus Endpunkten der Intervallen konstruiert. Der Wert des Elternknoten repräsentiert etwa der Mittelpunkt, der durch den Blattknoten aufgespannten Intervalls.

Diese Konstruktion erlaubt ein Intervall genau einem Knoten zuzuweisen, anstatt wie in Segment Tree an mehreren Knoten. Die Intervalle werden in den Knoten dessen Mittelpunktwert sie enthalten (Bildlich werden sie von diesem Wert aufgespießt), anstatt wie in Segment-Tree an den höchsten Knoten, deren Intervalle vollständig durch den Einfügeintervall abgedeckt sind, gespeichert. Für die effiziente Anfrageverarbeitung werden die „aufgespießten" Intervalle in den Knoten des Binären Suchbaums in einer zweiten Struktur verwaltet. In einfachere Implementierung besteht sie aus zwei sortierten Listen, die entsprechend nach Anfangs- und Endpunkten der assoziierten Intervalle sortiert sind.

External Interval-Tree

Eine Erweiterung von Interval Tree ist von [35] entwickelte I/O optimale Struktur für die Intervallverwaltung. Die erste Idee zur „Externalisierung" von Intervall Tree ist die Ersetzung des binären Suchbaum durch die Struktur mit großem Verzweigungsgrad. Dazu eignet sich sehr gut die B-Baum-Strukturen.
Dem Knoten wird in Externenspeicherstruktur eine Speicherseite assoziiert. Um I/O Schranken von B-Baum Strukturen beizubehalten und gleichzeitig die Besonderheiten der Intervall Tree implementieren zu können, muss der Knotendesign und insbesondere Füllungsfaktor geschickt angepasst werden. Die Schwiriegkeit liegt darin, den verfügbaren Speicherplatz einer Seite zwischen den Zeiger auf den Kinderknoten (entscheiden für den Verzweigungsgard und damit die Höhe des Baums) und Intervallelisten, die mit diesen Knoten assoziiert sind, aufzuteilen.
Ähnlich dem Hauptspeicherprinzip wird mit dem Knoten assoziiertes Intervall in $\sqrt{B}$ Intervalle aufgeteilt. Somit stehen $\sqrt{B}+1$ Werte, auf denen die Intervalle „aufgespießt" werden können, in einer Seite zu Verfügung.
Dabei werden mit jeder interne Intervall grenze b_i folgende Strukturen assoziiert, ähnlich wie in Intervall-Tree ein Zeiger auf die Listen mit Intervallen, die den Grenzpunkt b_i Schneiden, und eine Liste mit höchstens $\sqrt{B}$ Zeigern auf die Listen die Intervalle repräsentieren, die von mehreren Grenzpunkten „aufgespießt" werden. Die Letztere enthält so viel Zeiger wie Anzahl der Grenzpunkten rechts von b_i. Einzelne diese Listen werden nach Endpunkten der Intervallen sortiert.
Etwa die Hälfte $\approx \frac{B}{2}$ der Speicherseite ist durch diese Organisation belegt. Der Verzweigungsgrad der Struktur ist dementsprechend $\sqrt{B}$. Etwa weniger als $\frac{B}{2}$ bleibt für die direkte Speicherung von Intervallen in den Knoten, ansonsten werden diese in eine Hilfstruktur ausgelagert.
Die Anfrageverarbeitung läuft ähnlich dem Hauptspeicherstruktur. Z.B. für die Beantwortung von eine Zeitanfrage für einen einzigen Punkt t, wird der Baum nach unten wie ganz normale Suchbaum bis zum Blattknoten, der den t enthält, traversiert. Dabei werden die listen in internen Knoten durchgesucht. Sortierung der Listen ermöglicht effiziente Suche. Z.B. bei der Liste die nach Anfangspunktensortiert ist, wird die Liste vom Kopf bis zum Intervall mit $t < x_a$ durchgelaufen.
In [35] wird gezeigt, dass die Punktanfrage asymptotische Kosten von $O(\log_{\sqrt{B}} n) = O(\log_B n)$ besitzt. In [35] wird auch die Erweiterung auf voll dynamische Struktur vorgeschlagen. Durch die Komplexität von dieser Struktur wird sie selten in der Praxis implementiert.

Noch eine Möglickeit ist, wie bei der *Priority-Search Tree* [49], Mapping in zweidimensionalen Raum $I = (x_a, x_e)$ mit z.B. Anfangspunkten als X Achse und Endpunkten als Y Achse. Die Daten werden dabei in oberen Dreiecksregion der Bildraum vorkommen. Das Überlappungsproblem mit einem Anfrageinterval kann direkt mit einer zweidimensionalen

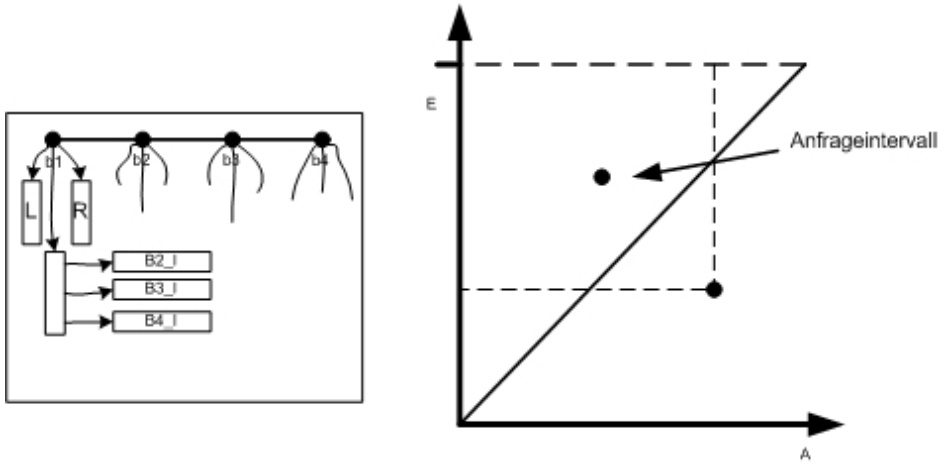

Abbildung 3.12.: Rechts ist ein Seitenlayout von externen Interval Tree dargstellt. Links ist das Mapping in zweidimensionalen Raum. Obere Dreieck ist die Bildregion von zweidimensionalen Punkten $I = (x_s = x, x_e = y)$. Das Anfrageintervall $I_q = (x_q, y_q)$ wird um die $y = x$ gespiegelt und der Anfragebereich wird als Rechteck mit Koordinaten $(0, x_q)$ und (y_q, max_e) dargestellt (für die Schnittberechnung $x_i < y_q \wedge x_q < y_i$).

Bereichsanfrage gelöst werden. Durch die Reduzierung des Problems auf Bereichsanfrage in zweidimensionalen Raum kann für die Verwaltung von diesen Punktdaten eine mehrdimensionale Indextechnik verwendet werden z.B. ein R-Baum oder ein B+Baum mit Einsatz von raumfüllenden Kurven. Das gleiche Prinzip kann auch auf die Schlüssel-Zeit Bereichsanfragen zu Nutze gemacht werden. Dabei werden die Intervallobjkete in dreidimensionalen Raum abgebildet. Eine der Vorteile dieser Technik, dass die relative kleineren Intervalle werden eher zusammengeclustert und getrennt von den längeren Intervallen in den Seiten vorkommen. Dabei ist lleiner Nachteil, dass die Anfragealgorithmus sehr großen Raumbereich traversieren muss.

MAP21

Eine andere Mapping-Technik ist *MAP21* [28] [37]. Bei dieser, werden die Intervalle $I = (x_a, x_e)$ auf einen Punkt $z = x_a \cdot 10^s + x_e$ abgebildet. Wobei s repräsentiert die maximale Anzahl der Ziffern, die notwendig sind, um jeden Zeitpunkt aus der Zeitdomain darzustellen. Die, auf dieser Weise abgebildeten Punkte, werden in einem B+Baum gespeichert.
Der offensichtliche Vorteil ist die volldynamische Struktur mit logarithmischen Einfüge- und Löschkosten. Um die Überlappungsanfrage effizient zu gestalten, wird in *MAP21* angenommen, dass Δ die Länge des längsten Intervalls aus der Datenmenge stets bekannt ist. Dieses Wert wird ausgenutzt um die Suche nach der Schnittintervallen einzuschränken, dieser garantiert, dass für einen Anfrageintervall $I_q = (x_{a_q}, x_{e_q})$ keine Intervalle außerhalb der $(x_{a_q} - \Delta, x_{e_q} + \Delta)$ liegen, die den Anfrageintervall schneiden.
Damit wird die Überlapungsproblem wie folgt gelöst, gegeben sei ein Intervall $I_q = (x_{a_q}, x_{e_q})$, berechne den Bild Punkt z_q für den erweiterten Intervall $\acute{I}_q = (x_{a_q} - \Delta, x_{e_q})$. Lokalisiere den Knoten, der den nächst kleinsten Punkt zu z enthält, mit Standard-B+Baum-Suchalgorithmus, laufe die Blattknoten sequentiell bis zum Knoten mit dem Punkt, der die rechte Suchgrenze repräsentiert $(x_{a_q}, x_{e_q} + \Delta)$. Dabei filtere die Intervalle die nicht das Anfrageintervall schneiden aus. Einer der Nachteile der Struktur, dass im Allgemeinen viele Intervalle, die nicht zum Ergebnis gehören, betrachtet werden.

Im nächsten Abschnitt wird eine effizientere Technik repräsentiert, die einerseits als Mappingtechnik andererseits als Externaliesierung des Intervall Tree angesehen werden kann.

IR-Tree

Eine Alternative zu R-Bäumen ist die Technik *IR-Tree* für die Verwaltung von eindimensionalen Intervallen von [36]. Die Technik besitzt einen großen Vorteil, sie ist sehr leicht zu implementieren. Die Gesamtstruktur folgt dem Paradigma von Indexstrukuren, die auf den Top-Level einer ORDBMS[3] implementiert werden können[36].
Der *IR-Tree* ist so wie *External Intervall Tree* von [35] basiert auf Hauptspeicherstruktur *Intervall Tree*. Anders als bei [35] wird die Aufteilung des Intervallraums „virtuell“ vorgenommen. Der Knotenwert („Aufspießwert“) des Intervall-Trees wird als Funktionswert berechnet. Vergleichbar mit Intervall Tree, besteht *IR-Tree* aus zwei Strukturen dem „virtuellen“ vollständigen Binärbaum und zwei B+Bäumen, die äquivalent den Links- und Rechtslisten der Knoten des Intervall Tree sind. Die Einträge, die in den beiden B+Bäumen verwaltet werden, besitzen folgende Struktur *(Interval Tree Knotenwert, Anfangspunkt, Zeiger)* bzw. *(Interval Tree Knotenwert, Endpunkt, Zeiger)*, der Zeiger zeigt auf den Speicherort des Intervallobjekts. Als Schlüssel dient der gesamte Eintrag mit lexikographische Ordnung. Der Interval-Tree-Knotenwert ist der Knotenwert des Knotens von den virtuellen Intervall Tree, von dem das Intervallobjekt „aufgespießt“ wird.
Der virtuelle Intervall Tree ist ein vollständiger Binärbaum von Höhe h. Somit repräsentieren die Knoten des Baums die Intervallendpunkte aus dem Bereich $\left[1, 2^h - 1\right]$. Mit dieser Aufbau lässt sich der „virtuelle“ *Intervall Tree* als eine Funktion implementieren mit dem Parameter h, die für einen Intervall den „Aufspießwert“ berechnet.
Aus diesem Grund das Positionieren des Intervalls in einem Intervall Tree erfordert nur die CPU Zeit ohne den Zugriff auf Externenspeicher.
Nach der Berechnung des Knotenwerte wird das Intervall in beiden B+Baumen eingefügt, das verursacht pro Datensatz nur die logarithmischen Kosten. Ähnlich gilt auch für das Löschen eines Intervalldatensatzes. Da ein Datensatz für ein Intervall nur einmal in beiden B+Bäumen vorkommt, besitzt der *IR-Tree* Speicherplatzaufwand von $O(n/b)$.
Die Anfrage, die zu einem gegebenen Anfrageintervall I_q alle Intervalle, die das I_q schneiden, berechnet, läuft in zwei schritten:

- Im ersten Schritt wird mit Hilfe des „virtuellen“ *Intervall Tree* entsprechend dem Hauptspeichervariante des *Intervall Tree* die Blattknoten, die die nächst kleinere oder größere Zahlwert zum Endpunkten des I_q repräsentieren, berechnet. Ähnlich dem Ausgangsvariante werden dann alle inneren Knoten des inneren Unterbaums eingeschlossen durch berechneten Werte, als potentieller Kandidaten betrachtet.
  Bei der Berechnung des Unterbaums werden drei Mengen von Knotenwerten ausgegeben. Beim Abstieg mit dem Anfangspunt des I_q werden „Aufspießwerte“ der Innerenknoten betrachtet, die außerhalb des zu I_q nächst liegenden Blattknoten sind, diese werden zu eine Menge L_q zusammengefasst. Entsprechend werden auch die Knoten die rechts von Endpunkt zu eine Menge R_q hinzugefügt. Die dritte Menge $Inner_q$ Repräsentiert den zusammenhängenden Bereich zwischen den berechneten Blattknoten, daher wird die Gesamtmenge, durch den linken und rechten Berechneten Blattkontenwerten dargestellt. Folglich verursacht der erste Schritt keine I/O Kosten.

- Im zweiten Schritt werden mit berechneten Mengen L_q, R_q und $Inner_q$, Anfragen an B+Bäumen gestellt. Für jeden Element aus L_q wird eine „Bereichsanfrage“ auf den

[3] Objekt-Relationales Datenbank Management System

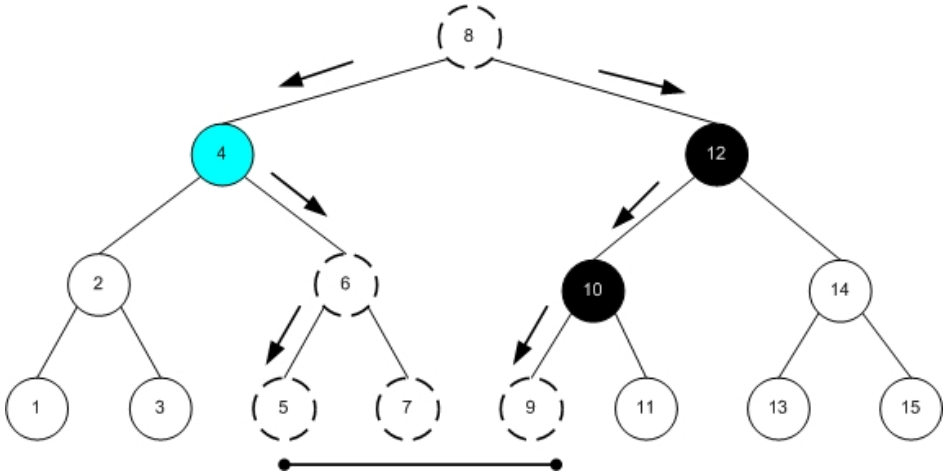

Abbildung 3.13.: Anfrage mit Intervall $I_q = (5, 9)$. Menge $L_q = \{4\}$ von den Knotenwerten die Links von dem Anfrageintervall sind, (Grau gefüllte Knoten). Menge $R_q = \{10, 12\}$ (Schwarz gefüllte Knoten) und $Inner_q = [5 - 9]$ Quelle[36]

B+Baum, in dem die Einträge mit Endpunkten von Intervallen verwaltet werden, gestellt. Die Anfrage gibt alle Intervalle, dessen Knotenwert gleich dem Anfrageknotenwert und der Endpunkt rechts von Anfangspunkt von I_q ist, aus. Analog werden auch die Intervall für die Menge R_q berechnet. Für die $Innerq$ kann sowohl B+Baum mit Anfangspunkten als auch B+Baum mit Endpunkten verwendet werden, dabei wird eine Berechsanfrage für den inneren Bereich durchgeführt. Anschließend werden die Ergebnisse von Bereichsanfragen zusammengefasst und ausgegeben.

Die I/O Kosten des Verfahren setzten sich ausschließlich aus den Bereichsanfragen an den B+Bäumen, da das Traversieren des „virtuellen“ *Intervall Tree* nur CPU Kosten erfordert. Der gesamt I/O Aufwand beträgt damit $O(h \cdot \log_b n + r/b)$, wobei h ist die Höhe des „virtuellen“ Baums und b ist der Mindeskapazität der B+Baum Knoten. Daher bietet das *IR-Tree* Verfahren auch optimale I/O Schranken. Der größte Vorteil ist allerdings die Einfachheit des Verfahrens. Im Allgemeinen lässt sich *IR-Tree* in fast jeden ORDBMS implementieren, die Technik benötigt ein Prozedur zur Berechnung des „virtuellen“ Baums und B+Bäume, die Anfragen könnten direkt durch die SQL gestellt werden [36].
Eine Frage die noch nicht besprochen wurde, und sowohl für die Transaktionszeitumgebung und insbesondere für Validzeitumgebung relevant ist, die Verwaltunng von Intervallen dessen Endzeitpunkt noch nicht bekannt ist. Beim Entwurf von Validzeitindexe ist die gängige Technik, die Verwaltung von solchen Objekten in einer separaten Struktur. Häufig wird für diesen Zwecken ein B+Baum eingesetzt. Die Intervalle werden dann nach den Anfangspunkten indexiert. Damit reduziert sich die Überlappungsproblem auf eine Standard-Bereichsanfrage vom B+Baum.
In dem Abschnitt wurden die Verfahren präsentiert für Validzeit. Dieser Zeitbegriff ist explizit in RDF Daten vorhanden dagegen für die Transkationszeit wurden keine klare Anwendungen präsentiert. Im nächsten Abschnitt werden Argumente für die Verwendung von Transaktionszeit-Techniken für RDF Daten vorgestellt.

3.2.3. Transaktionszeit und Versionierung in RDF

Dieser Abschnitt versucht die Frage zu klären, welche Rolle die Transaktionszeit in RDF spielt. In dem Abschnitt, in dem die formale Einführung der Zeitdimension in RDF vorge-

stellt wurde, war kurz angedeutet, dass die Modellierung der Zeit durch Timestamp-Modell sowohl die Validzeit als auch die Transaktionszeit in RDF darzustellen erlaubt. Der Anwendungsbereich von Transkationszeit in RDF wurde aber nicht genau erläutert. Die berechtigte Frage ist dabei, ob überhaupt, die praktische Relevanz der Transaktionszeit in Zusammenhang mit RDF Daten besteht.
Ein verwandter Begriff zur Transaktionszeit ist die Versionierung, die kann als eine praktische Umsetzung des temporalen Snapshot-Modells interpretiert werden. Mit dem Timestamp-Modell in relationalen Datenbanken wird die Versionierung durch die Transaktionszeitkonzept ausgedrückt. In meisten Fällen die kleinste zu versionierende Einheit ist ein relationales Tupel, der einen Fakt aus der modellierten Welt repräsentiert. Versionierung kann auch Dokumentenorientiert bzw. Objektorientiert implementiert werden, z.B. Source-Code und Textdokumenten Versionsverwaltungssysteme. In RDF ist vorstellbar sowohl die Fakten-Daten als auch insbesondere Schema-RDF-Graphen zur versionieren.
RDF Daten dienen zur Wissensrepräsentation und werden unter anderem aus verschiedenen Quellen zusammen geführt, um z.B. semantische Analyse von Daten durchzuführen. Da die Anreicherung und Integration von RDF Daten ein kontinuierliche Prozess ist, können neue Beziehungen und Aussagen hinzugefügt oder gelöscht werden, mit der Verwaltung von vergangenen Zuständen wäre es möglich die Qualität die RDF Daten zu verbessern und strukturelle und semantische Unterschiede festzustellen. Die historische Evolution von RDF Graphen kann sowohl semantisch als auch strukturell analysiert werden um neue Erkenntnisse zu gewinnen[4].
Bei der Versionierung von RDF Daten können verschiedene zu versionierende Einheiten verwendet werden. Die Wahl der Granularität ist applikationsabhängig. Grundsätzlich können die gesamten Graphen versioniert werden. Falls aber der RDF Graf die Wissensdatenbank darstellt, besitzt er im Allgemeinen eine große Anzahl der Tripel. Die Verwaltung von einzelnen Zuständen ist daher ineffizient (es könne natürlich auch die effiziente Δ - Kompressionstechnik verwendet werden, die temporale Anfragen wären trotzdem ineffizient, vergleiche die Diskussion über Copy- und Log-Technik).
Der Verwaltungsaufwand könnte verringert werden, indem der Graph geeignet partitioniert wird um die einzelne Subgraphen zu versionieren z.B. auf dem Niveau der einzelnen Knoten mit ausgehenden Kanten. Für die einzelnen Subgraphen könnten die Ids oder Signaturen berechnet werden(im einfachsten Fall die Subjekt URI) und es könnte eine der transaktionszeit Zugriffstrukturen aufgebaut werden. Dabei aber könnten auch die Techniken aus dem XML-Versionierung verwendet werden.
Bei dieser Lösung ist aber die Größe der einzelnen Subgraphen ist schwer zu kontrollieren. Daher ist die einfachste Lösung, ähnlich der Versionierung der relationalen Daten, Verwendung für die kleinste Einheit den Subgraphen in Form eines einzelnes RDF Tripel. Der Vorteil dabei ist die vollständige Verwendung der relationalen Transaktionszeittechniken, unter andrem die Verarbeitung von komplexen temporalen Anfragen.
In [39] wird diese Methode zur Versinonierung von RDF und RDF Schema verwendet. Im diesen Kontext die kleinste zu versionierende Einheit wird auch asl Atom bezeichnet, das kommt daher, dass das Tripel als kleinste nicht änderbare Einheit angesehen wird. Mit anderen Worten es werden nur die Einfüge- und logische Löschoperationen unterstützt (Abbildung 3.2.3). In dem Paper wird aber die allgemeine Verwaltungsschema vorgeschlagen auf dem höheren Ebene (es werden nach dem Techniken aus [16] die Tabellen auf dem SQL Niveau für Transaktionszeitverwaltung erstellt). Diese Lösung kann auch zur Versionierung von den beliebigen Graphen eingesetzt werden. Auf Grundlage diese Annahme können direkt die Techniken wie MVBT, MVAS und TSB eingesetzt werden. Die Gründe für die Verwendung von diesen

[4]http:blogs.sun.combblfishentrytemporal_relations beschreibt ein kleines Beispiel für temporale RDF

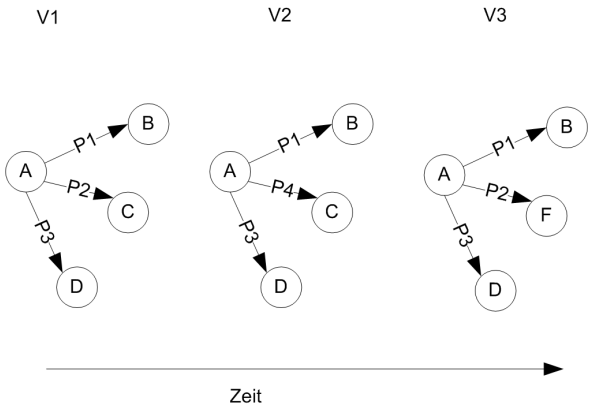

Abbildung 3.14.: Versionierung von einem RDF Subgraphen. In erste Version werden die Tripel $(A, p_1, B), (A, p_2, C), (A, p_3, D)$ eingeführt, in zweiten Version ändert sich das Tripel $(A, p_2, C) \rightarrow (A, p_3, C)$. Danach wird das Tripel (A, p_3, C) komplett gelöscht und durch das Tripel (A, p_2, F) ersetzt.

Strukturen wären: insbesondere MVBT und MVAS die lineare Speicherplatverbrauch und Unterstützung von komplexen temporalen Anfragen in Zusammenhang mit Transaktionszeit (z.B. Schlüssel-Zeit-Anfragen).
Die MVBT Techniken werden bereits in Bereich der Versionierung von XML Dokumenten eingesetzt um die temporale Anfragen zu unterstützen [42]. Eine der Probleme beim Einsatz MVBT in XML Bereich ist die Verwendung von eindeutigen Identifikatoren und die Frage nach den kleinsten Einheiten. Grundsätzlich wird dabei die Knoten des XML Baums als Atome verwendet. Die häufigste Technik für Identifikatorenvergabe in XML Dokumenten ist die Preorder-Nummerierung der Knoten in einem XML Dokument. Die Schwierigkeit bei der Versionierung ist, dass durch die Änderung der Struktur eines Dokuments die Ids neu berechnet werden müssen. Dafür in [42] wird die spezielle Identifikatoren-Technik verwendet, die erlaubt ohne neue Berechnung des Ids auszukommen. Diese Technik gibt jeden Knoten entsprechend großen Intervall im Voraus, das die eventuellen späteren Preorder Änderungen aufnehmen kann und bis zur n-Änderungen invariant bleibt, damit besteht der eindeutige Identifikator aus dem Intervall und dem Preroder-Nummer. Mit Hilfe von diesen speziellen Identifikatoren wird der MVBT aufgebaut.
Im Vergleich zu RDF die Verwendung von MVBT in XML viel komplizierter. Durch die Verwendung von Tripel als ein Atom reduziert sich die Aufgabe auf die Verwaltung von dreistelligen relationalen Tupel, in einem transaktionszeit Kontext. Trotz der simplen Idee erlaubt diese Ansatz die strukturelle Evolution eines Graphen mit zu verfolgen (Abbildung 3.2.3).
In der Praxis wird am häufigsten die Versionierung von RDF Daten bei der Ontologienverwaltung verwendet. Zum Beispiel im *Gen Ontology* Projekt werden Beziehungen zwischen den biologische Arten beschreiben. Diese werden aber ständig rekonsolidiert, überarbeitet und angereichert[38]. Die wichtigste Anforderung bei der Versionierung der Ontologien ist Berechnung der strukturellen und semantischen Differenz zwischen den Versionen. Da semantischer Teil der Ontologie Konzept mit OWL viel komplexer als RDF und RDF/S, die Berechnung der semantischen Differenz hat dabei eine große Stellung, es ist vergleichbar mit der Gewährleistung der Integritätsbedingungen in relationalen Datenbanken, z.B. bei der Änderungen von Beziehungen zwischen Klassen von Objekten können Konflikte auftreten, deshalb muss das System in der Lage sein, die semantischen Unterschiede festzustellen und

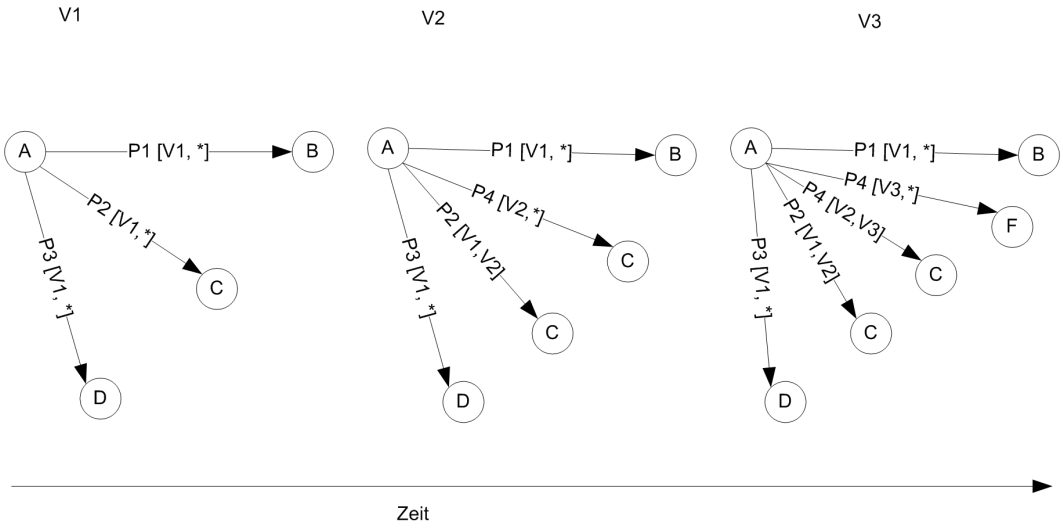

Abbildung 3.15.: Bei dieser Versionierung wird RDF Tripel als atomare Einheit interpretiert. Auf diese Weise würde das RDF Graph bei der Verwendung von Transkationszeit-Indexstruktur aussehen.

unter Umständen das Propagieren der Änderungen zu verbieten.
In aktuellen Versionierungssystemen für Ontologien aus der Datenbanksicht werden entweder *Log*-Technik oder *Copy*-Technik verwendet[41]. Viele Systeme haben ähnliche Architektur wie Source-Code-Verwaltungssystemen, die Speicherung der Versionen geschieht durch die Δ Verwaltung. Ein der Nachteilen dieser Methode ist, in Abhängigkeit der verwendete Δ-Verfahren, Wiederherstellung der älteren Versionen und ineffiziente Unterstützung von Anfragen auf vergangene Versionen. Ein andrer Extremem ist die Verwaltung von einzelnen Versionen (vgl. *Copy*-Texchnik) in einzelnen Tripletables.
In [41] wird versucht die Speicherplatzausnutzung von dem Δ-Verfahren zu verbessern, die einzelnen Δ werden als Mengen der Tripel betrachtet. Durch die Ausnutzung der partiellen Ordnung auf Mengen wird versucht die Anzahl zu speichernden Δ Mengen zu verringern. Dafür wird ein Graph über der partielle Ordnung aufgebaut. Einzelne Versionsmarken zeigen dann auf die Knoten der Δ-Graphen. Das Verfahren verbessert zwar die Speicherplatzausnutzung hat aber dieselben Nachteile in Bezug auf Anfrageverarbeitung wie die vorigen.
Es bietet sich an, auch bei der Versionierung von Ontolgie-Daten als Backend-Struktur eine von Transaktionszeittechniken wie MVAS, TSB undMVBT einzusetzen. Die Versionen die sich auf größere Einheiten als ein einziger RDF Tripel beziehen, können in Versionshierarchien zusammengefasst werden, und durch den Zusatzindex verwaltet werden[40]. Der offensichtliche Vorteil ist die Anfrageverabeitung, die direkt an allen Versionen arbeiten kann.
Im nächsten Abschnitt wenden wir uns ganz andere Kategorie von Indexen, bis jetzt wurden die Strukturen angepasst auf die Verwaltung von RDF Daten, demnächst werden die Indexe präsentiert, die speziell für RDF Daten entwickelt wurden.

3.3. Spezielle Indexe

In diesem Unterkapitel werden spezielle Indextechniken für RDF Daten präsentiert, die unter Berücksichtigung der Graphstruktur der RDF Daten entwickelt wurden. Viele Indextechniken für die Beantwortung von Graph-Anfragen sowohl im Bereich RDF als auch anderen Bereichen von Datenverwaltung, basieren auf einen gemeinsamen Prinzip, geschickte Parti-

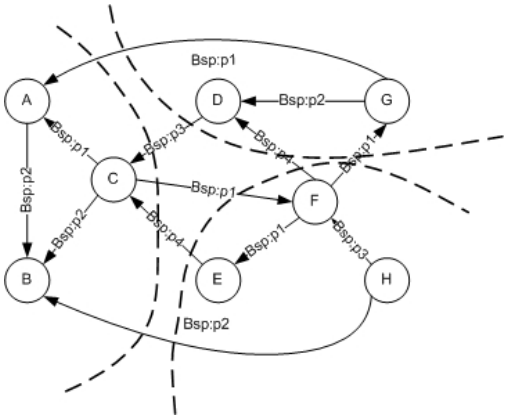

Abbildung 3.16.: Exemplarische Aufteilung eines RDF Graphen

tionierung des Graphen (z.B. in Geo-Anwendungen für Straßennetzwerke). Die Motivation ist, dass viele Graph Algorithmen relative hohe Berechnungskosten besitzen (divide and conquer Einsatz). Die Typischen SPARQL-Anfragen lassen sich in einer Graphenumgebung, als die Suche nach einem Subgraphen in einem Graphen interpretieren. Grundlage für die Algorithmen könnten unter anderem Subgraphisomorphismus-Algorithmen dienen, allgemein bekannt, dass diese Algorithmen hohe Berechnungskosten aufweisen, so dass die zum dominierenden Teil werden. Um den entsprechenden Aufwand zu verringern wird auf das Aufteilungsprinzip zugegriffen. Unter Umständen erlaubt die Partitionierung, bei der Anwendung von teuren Graphalgorithmen groß Teil der Partitionen anhand von bestimmten Eigenschaften auszuschließen.
Im Bereich der RDF Indexierung wird daher auch auf bewährte Techniken aus dem Bereich der Graphindexierung benutzt. Z.B. Aglomerative Hierarchische Cluster Algorithmen[25]. Es ist auch denkbar einen Algorithmen für starken Zusammenhangskommponenten für den Einsatz in externen Speicher Umgebung anzupassen, allerdings die Hierarchische Cluster Algorithmen haben den Vorzug, da sie bessere Kontrolle über die Größe der Partitionen erlauben. Bei der Umsetzung von erweiterten Operatoren, für allgemeine Pfadausdrücke, werden auch Techniken aus dem Bereich Graphindexierung benötigt. In diesem Unterkapitel werden zwei Techniken exemplarisch präsentiert, die erste wird nur kurz skizziert, die zweite wird tiefer Beschrieben, da sie, nicht nur die Graphstruktur von RDF berücksichtigt sondern die Anfragen auf temporalen RDF Graphen unterstützt, was ausschlaggeebend für meine Arbeit ist.
Ein interessantes Verfahren zur Indexierung RDF Graphen schlagen die Autoren [27]. Bei diesem Verfahren kommt zusätzlich Techniken aus dem Information-Retrieval-Bereich. Bei dieser Technik wird der RDF Graph in überlappenden Partitionen aufgeteilt, zum Partitionierung Algorithmus leider geben die Autoren keine Auskunft, anschließend werden Signaturen(textbasiert wie in Information Retrieval) zur jede Partition berechnet und eine Signatur Baum aufgebaut, mit dessen Hilfe wird versucht den Suchraum zu verringern.
Im nächsten Abschnitt wird eine Indexstrukur präsentiert, die auf einem ganz anderem Konzept basiert ist, und kann sowohl die Graphangfragen als auch temporale Graphanfragen beantworten. Die Struktur ist von [25] entwickelt worden.

3.3.1. TGRIN Indexstruktur für temporale RDF Graphen

Die Folgende *TGRIN* Indexstruktur für die Beantwortung von Graphanfargen an temporalen RDF Graph 2.2 wurde auf der Basis der Hauptspiecherstruktur [24] entwickelt. Diese Struktur unterscheidet sich maßgeblich von Strukturen die in diesem Kapitel schon präsentiert

worden. TGRIN lässt sich in die zweite Kategorie der Indexe für RDF Daten einordnen, und indexiert unter anderem temporalen RDF Graphen. Aus der Sicht der temporalen Indexstrukturen, lässt sich TGRIN für die Indexierung von Validzeit-Daten verwenden.
Die Autoren aus [25] setzten die temporale Semantik von RDF Daten von 2.2 voraus. Das heisst, dass die RDF Tripel sind mit einem Zeitintervall direkt annotiert sind.
Eine der Vorteile von dieser Technik, dass die Graphanfragen (unter Einschränkung, die später besprochen wird) werden direkt mit dem Index beantwortet. Mit andren Worten, die Anfragen werden nicht in relationalen Operatoren übersetzt, sondern als ein Subgraph dargestellt und es wird das Graphmatching direkt mit Hilfe der Subgraph-Isomorphismus Algorithmus ausgeführt. Von dem Konzept her, lässt sich TGRIN als eine statische, metrische Indexstruktur beschreiben.
Grund Idee von [24][25] den Graph so geschickt zu Partitionieren (aufteilen) um die kosten des teuren *Subgrpah-Ismorphismus* Algorithmus zu reduzieren. Für die Graphaufteilung bei diesem Verfahren sorgt von den Autoren[25] angepasste Hierarchische Aglomerative Cluster Algorithmus. Mit dem Algorithmus wird eine hierarchische Struktur im Form von einem Mehrwegbaum aufgebaut.
Verteilung der Graphknoten des RDF Graphen (Subjekte und Objekte) auf verschiedene Partitionen erfolgt anhand definierter metrischer Distanz. Ähnlich zum metrischen Indexstrukturen, wie zum Beispiel M-Baum, wird auf dieser Weise aufgebaute Hierarchie zur Beantwortung von Anfragen in Top-Down Manier traversiert. Es wird dabei versucht möglichst viele Unterhierarchien auszuschließen, um eine relativ kleinere Mengen von Graph-Konten zu bekommen. Auf diese Mengen wird anschließend der teure Graphisomorphismus-Algorithmus angewendet.
Vor dem Aufbau des Indexes wird zusätzlich der Datenbestand von temporalen RDF Tripel bereinigt. Diesen Prozess nennen die Autoren Normalisierung der temporalen RDF Datenbank. Eigentlich wird die *Colescing* an RDF Tripel durchgeführt, dadurch argumentieren die Autoren wird die Anzahl der Tripel reduziert. Die Diskussion über die temporale Normalisierung wird in [25] ausführlich beschrieben.
Der TGRIN Index wird nicht über die RDF Tripel Aufgebaut sonder nur über die Knoten des RDF Graphen, dass heißt nur über die Subjekten und Objekten. Im Folgenden werden die wichtigsten Konzepte des Verfahren nähe erläutert.

TGRIN Architektur

Der Kern der Indexstruktur ist die metrische Distanzfukntion. Um „geignet“ den RDF Graphen zu Partitionieren wird eine Metrische Distanzfunktion zwischen den Knoten der RDF Graphen verwendet. Dabei werden zwei unterschiedliche metrische Distanzen $d_t(n_i, n_j)$ und $d_g(n_i, n_j)$ definiert und miteinander mit Hilfe der k-Norm vereinigt.
Die erste ist eine temporale Distanz zwischen zwei Knoten der RDF Graphen, die zweite ist eine Graph Distanz eines RDF Graphen. Für diese Distanz müssen wir die üblichen Begriffe aus der Graphen-Theorie, wie Pfad, Länge des Pfades, Kannte und Knoten verwenden. Den *Pfad* zwischen zwei Knoten x und y definieren wir durch $p = (e_1 \dots e_n)$ Menge von Kanten. Die Länge ist definiert durch die Anzahl der Kanten eines Pfades.
Zwischen zwei Knoten in einem RDF Graphen können viele Verschiedene Pfade existieren, ähnlich wie in Netzwerkgraphen, daher wird ein ausgezeichneter Pfad gewählt z.B. mit maximale oder minimale Distanz. Für den Graph-Distanz kann z.B. die Länge des kürzesten Pfades zwischen zwei Knoten-Ressourcen verwendet werden. Um die temporale Distanz definieren zu können, werden mehrere Schritte gebraucht.
Im ersten Schritt wird die Interval-Distanz δ definiert, die die Distanz zwischen zwei aufein-

ander folgender Intervalle beschreibt. Dabei soll Distanz δ folgende Axiomen erfüllen:

1. $\delta(T_i, T_j) = 0$
2. $\delta(T_i, T_j) = \delta(T_j, T_i)$
3. $\delta(T_i, T_j) \leq \delta(\acute{T}_i, \acute{T}_j)$ wenn $\acute{T}_i \preceq T_i, T_i \preceq T_j$ und $T_j \preceq \acute{T}_j$ wobei $T \preceq \acute{T}$ ist definiert durch $T.e \leq \acute{T}.s$ mit $T.s$ als Startpunkt und $T.e$ Endpunkt.

Im folgenden sind eine Paar Distanzfunktionen aufgelistet die diese Axiomen erfüllen und können für die δ verwendet werden:

1. $\delta(T_i, T_j) = \left| \frac{T_i.e - T_i.s}{2} - \frac{T_j.e - T_j.s}{2} \right|$ absolute Differenz der Mittelpunkte.
2. $\delta(T_i, T_j) = |T_i.s - T_j.s|$ Absolute Differenz der Startpunkte
3. $\delta(T_i, T_j) = |T_i.e - T_j.e|$ analog ist auch absolute Differenz der Endpunkte.

Nachdem δ Definiert wurde, können wir die gesamte temporale Distanz zwischen zwei Ressourcen mit δ definieren $d_t(n_i, n_j)$.

Definition 3.3.1 (Temporale Distanz in TGRIN-Index) *Sei D eine temporale RDF Datenbank, $x, y \in (U \cup L) = R$, sei $p = (e_1 \dots e_n)$ ist ein Pfad zwischen den Ressourcen x und y in einem temporalen RDF Graphen, sei T_j ein Zeitintervall von der Kante e_j. Dann temporale Distanz eines Pfades $d_t^p(x, y)$ ist wie folgt definiert, Falls $n = 1$ dann wird $d_t^p(x, y) = 0$ gesetzt, andernfalls $d_t^p(x, y) = \sum_{j \in [2,n]} \delta(T_j, T_{j-1})$. Die temporale Distanz $d_t(x, y)$ ist definiert dann durch:*

$$d_t(x, y) = \min_p(d_t^p(x, y))$$

Nachdem beide Distanz definiert wurden lässt sich mit Hilfe der k-Norm die Gesamtdistanz definieren:

$$d(n_i, n_j) = \left\{ d_t(n_i, n_j)^k + d_g(n_i, n_j)^k \right\}^{\frac{1}{k}}$$

Mit Hilfe von dieser Distanz lässt sich TGRIN als (balancierte) Mehrwegbaum aufbauen. Damit wird der TGRIN wie folgt definiert:

- Jeder Blattknoten l enthält eine Menge N_l von Ressourcen und Literalen mit $N_l \in U \cup L = R$. Für jeden Blattknoten mit $\acute{l} \neq l$ gilt $N_{\acute{l}} \cap N_l = \emptyset$ zusätzlich gilt: sei M menge von allen Blattknoten von TGRIN dann $\cup_{l \in M} N_l = R$
- Jeder Indexknoten t enthält ein Paar (c, r) mit $c \in R$ und $r \in \mathcal{N}$ (Zentroid und Radius), und die Menge von Zeiger N_t mit $N_t = \{\acute{c} \in R \mid d(c, \acute{c}) \leq r\}$. Mit Anderen Worten mit dem Zentroid und Radius werden die darunter liegende Menge von Knoten die gemäß der Distanz (z.B. Temporallen Distanz) kleiner gleich dem r zu dem Zentroid haben, beschrieben.
- Für jeden Knoten x und y so dass x ein Elternknoten von y ist, gilt $N_y \subseteq N_x$.

Ein weitere wichtiger Parameter beim allen externspeicher basierten Mehrwegbäumen ist die Verzweigungsgrad M. Beim TGRIN je höher der M ist desto höher die Wahrscheinlichkeit kleinere Graphpartitionen zu matchen, andererseits der „buschiger“ Baum vergrößert den Suchraum.
Der Aufbau des Indexes erfolgt in drei Schritten. Im Schritt eins wird modifizierte HAC

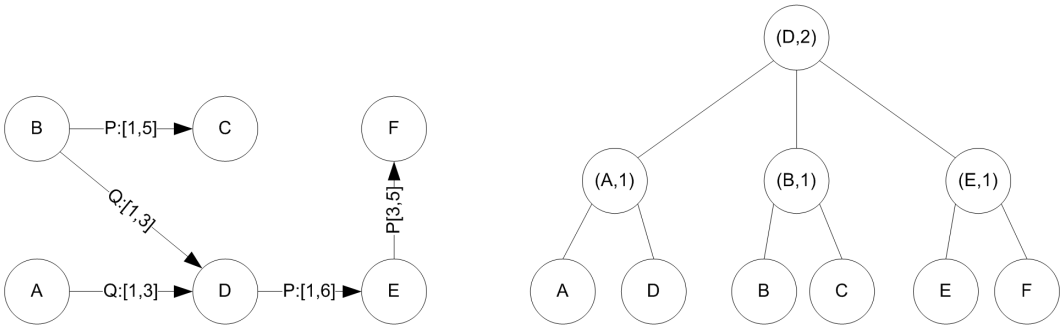

Abbildung 3.17.: Beispielaufbau eines TGRIN-Indexes. Quelle[25]

verwendet. Der Algorithmus fängt mit je einem Knoten pro Cluster und wird rekursiv aufgerufen. Für jeden Paar der RDF Ressourcen wird die definierte temporale Distanz berechnet. Der TGRIN Index an sich verwaltet nur die Zeiger auf die Tripel, die wiederum z.B. in einer Tripletable gespeichert sind. Somit lässt sich der RDF Tripel zu bestimmten Subjekt und Objekt durch den Look-Up in der Tripeltable berechnen.
Entscheidende Rolle spielt dabei die Zwischenclusterdistanz. Sie wird als minimale oder maximale Distanz zwischen zwei Cluster festgelegt. Anhand von dieser werden die Cluster verschmolzen, anschließend wird ein Dendrogramm als Output ausgegeben. Die Anzahl der Knoten pro Cluster in der Dendrogramm ist höchstens M.
Im zweiten Schritt für einzelne Cluster in der berechneten Dendrogramm werden *Zentroid* und *Radius* ausgewählt. Der Zenroid c für die Menge von Knoten (Cluster) S wird definiert durch die minimale durchschnittliche Distanz $c = \min_z(avg_{y \in S}(d(z, y)))$ und Radius als $r = max_{x \in S}\{d(c, x)\}$.
Im letzten Schritt wird der Baum ausbalanciert. Der Worst-Case-Kompexität für den Aufbau des Baums beträgt[25] $O(|R|^3 \log_M |R|)$.
Im wesentlich setzen sich die Kosten aus der Berechnung von Distanzen zwischen allen Subjekt- und Objekt-Knoten des RDF Graphen in ersten Schritt des Clusteralgorithmus. Der Aufwand für die Berechnung von allen minimalen Distanzen zwischen den Knoten beträgt $O(|R|^3)$. Dabei ist zu beachten, dass diese Distanzen müssen explizit verfügbar sein, für diesen Zweck kann ein Hash-Index aufgebaut werden, um bei der Anfrageverarbeitung die Distanz nicht mehr zu berechnen sondern als Konstante behandeln. Für die Aufstellung der Dendrogramm beträgt der Aufwand $O(|R|)$. Die Berechnung der Interclusterdistanz ist mit Aufwand von $O(|R|^2)$ verbunden. Die Höhe des Indexes beträgt $\log_B |R|$ mit maximal $O(|R| \log_M |R|)$ Knoten. Der weitere Faktor ist die Berechnung des Zentroids für jeden Knoten, diese Schritt hat Kosten von $O(|R|^2)$. Damit ergibt sich der Gesamtaufwand von $O(|R|^3 \log_M |R|)$.

TGRIN Anfrageverarbeitung

Um den Anfrageverarbeitungsalgorithmus zu beschreiben brauchen wir zunächst die Anfragegraphen in temporalen RDF-Umgebung formal zu definieren vgl. SPARQL:

Definition 3.3.2 (Temporale RDF-Anfrage) *Temporale RDF Anfrage ist ein Quintupel* $(N, E, V, \lambda_n, \lambda_t)$ *mit:*

- N *ist eine Menge von Knoten*
- V *ist eine Menge von Variablen*
- $E \subseteq N \times N \times (V \cup U)$ *ist eine Menge von Kanten*

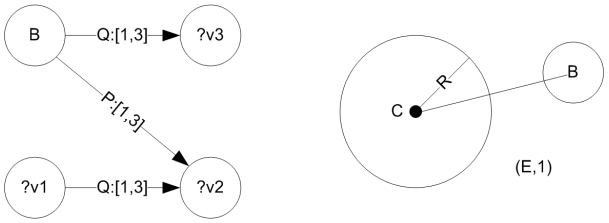

Abbildung 3.18.: Rechts Anfragegraph. Links ist die Situation, bei der der TGRIN Knoten nach der Bedingung 1 ausgeschlossen wird. Quelle [25]

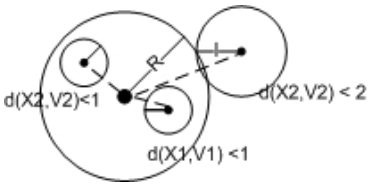

Abbildung 3.19.: Fall 2 bei dem ein innere Knoten von TGRIN verworfen wird, da zwei Bedingungen werden zwar erfüllt der Dritte liegt aber nicht vollständig in dem Kreis

- $\lambda_n : N \rightarrow (U \cup L) \cup V$ *ist eine Funktion, die das Label (Namen) den Knoten zuweist.*
- λ_t *ist eine Funktion, die den Kanten die temporale Labels zuweist.*

Die Anfrageverarbeitung läuft in mehreren Schritten ab.
Im ersten Schritt werden die temporale TGRIN Distanzen zwischen den Knoten, die die Variablen repräsentieren, und den Ressource-Knoten berechnet. Die berechneten Distanzen $d_q(n, v)$ werden in Form von Ungleichungen $d(n, v) \leq d_q(n, v)$ zu der Menge $cons(q)$ hinzugefügt. Die Menge $cons(q)$ repräsentiert die Menge der Bedingungen oder Einschränkungen (eng. Constraints) mit deren Hilfe die Entscheidung getroffen wird, ob der Knoten aus dem TGRIN Index definitiv den Anfragegraphen *nicht* enthält.
Ähnlich zu den metrischen Bäumen läuft die Anfragealgorithmus in Top-Down-Tiefensuche, dabei werden die Kreise der inneren Knoten mit der Menge $cons(q)$ untersucht. Zum Ausschluss der Knoten werden zwei Regeln angewendet:

1. Die erste Regel bezieht sich auf die Ressourceknoten aus q. Ein TGRIN Knoten mit (c, r) wird definitiv ausgeschlossen, falls für einen Ressourceknoten x aus q gibt mit $d(c, x) > r$ ist. Mit anderen Worten, falls der Resourceknoten aus der Anfragegraphen außerhalb der Kreises, der den TGRIN Knoten beschreibt, liegt.

2. Die zweite Regel nutzt die metrische Eigenschaft der Temporalen Distanz aus, und bezieht sich auf die Elemente der Menge $cons(q)$. Sei $d(x, v) \leq l \in cons(q)$ dann folgt aus der Dreiecksungleichung $d(c, v) \leq d(c, x) + d(x, v) \leq d(c, x) + l$ und mit $d(c, x) + l \leq r$, dass die Variable v liegt in den Kreis definiert durch (c, r). Um den Indexknoten auszuschließen genügt es, dass mindestens eine der Varibalen v aus q außerhalb der Kreises liegt, ansonsten wird der Knoten zu Weiterverarbeitung aufgenommen.

Ausgabe des Traverisierungsalgorithmus ist die Menge der Blattknoten.
Im letzten Schritt des Algorithmus wird der Subgraphisomorphismusalgorithmus auf die Ausgabemenge angewendet. Im [25] wird gezeigt dass die Worst-Case-Zeitkomplexität des Anfragealgorithmus in TGRIN beträgt $O(|R|!)$, wobei diese hohe Kosten werden hauptsächlich von dem Subgraphmathing-Algorithmus dominiert.

Der TGRIN Indexstrukutur wurde experimentell mit R+Baum[5] und MAP21 verglichen. Dem Ergebnissen zufolge für die Graphmathinganfragen in einer Validzeit-Umgebung zeigt der TGRIN wesentliche bessere Performanz.
Der Aufbau-Zeit des TGRIN Index leider lässt sich schwer mit in diesem Kapitel vorgestellten Strukturen vergleichen, da der Index Komplett in Hauptspeicher aufgebaut wurde. Die I/O Zugriffe und Zeit für das Laden der Daten und Schreiben des Indexes in externen Speicher wurde nicht berücksichtigt.
In den Experimenten wurden dann auch einer der wichtigsten Parameter für den TGRIN ermittelt, nämlich der Verzweigunsgrad M. Mit M zwischen 4 und 6 wurden die besten Antwortzeiten bei der Anfrageverarbeitung gemessen.
In Großen und Ganzen kann TGRIN als zusätzlicher Index für die Beantwortung komplexer Graphmathings in einer Validzeit-Umgebung verwenden. Die Vorteile von dem Index, dass der unabhängig von der relationalen Verwaltungschema für RDF Daten verwendet werden kann. Bei den Graphfragen werden die teuren Joins vermieden. Die SPARQL Anfragen müssen nicht in SQL übersetzt werden sondern können direkt als Subgraph dargestellt und mit TGRING verarbeitet werden.
Zu den Nachteilen zählt die Komplexe Architektur, Voraussetzung von dem statischen Verhalten der Daten (bei der Änderung der Datenmenge, muss unter Umständen der Index neu Aufgebaut werden), limitierte Unterstützung von Anfragen, es werden nämlich nur Graphpfadanfragen mit vorgegebenen Subjekten und Objekten effizient verarbeitet werden. Die Anfragen die ausschließlich nach der Beziehungen von den RDF Ressourcen anfragen, werden sehr ineffizient verarbeitet, da keine Einschränkungen aus der Anfragegraphen abgeleitet werden können und keine Ressourcen explizit in die Anfrage involviert sind.

3.4. Zusammenfassung

In diesem Kapitel wurden sowohl die Indexstrukturen für RDF Daten als auch für zeitabhängige RDF Daten vorgestellt. In der Praxis werden die B+Bäume für die Indexierung von nichttemporalen Daten eingesetzt. Der RDF Tripel lässt sich auf die dreikomponentigen Tupel abbilden. Es wurde gezeigt, dass es mindestens drei B+Bäume für die effiziente Anfrageverarbeitung benötigt werden. Für die bessere Performance werden aber die geclusterte Indexe über alle Permutationen von (S, P, O) eingesetzt. Trotz der Großen Anzahl der B+Indexe kann der Platzverbrauch klein gehalten werden, indem die effiziente Komprimierungstechnik für die B+Bäume in Zusammenhang mit der Verwaltung von zusammengesetzten Schlüssel verwendet wird. Die B+Bäume dienen auch direkt als Speicherstrukturen. Die Bereichsanfragen werden (Indexscans) als Knoten der Operatoren Bäume verwendet(Insbesondere bei Verarbeitung von Pfad-Anfragen mit Joins). Auf der Basis der dreikomponentigen Datenbanktupels und des Timestamp-Modells lassen sich auch die temporalen Techniken aus der RDBMS verwenden.
In dem Kapitel wurde die Klassifikation der Techniken beschrieben, insbesondere im Hinsicht auf die Unterstützung von Anfragen. Bei Transaktionszeit-Techniken war die Schwierigkeit die Balance zwischen den Speicherpaltzkosten und Anfrageverarbeitung zu finden. Die Techniken wurden in drei Typen in Bezug auf Anfragen unterteilt: die reinen Zeitanfragen, allgemeine Schlüssel-Zeit Anfragen, die reinen Schlüssel-Anfragen. Dabei wurden zwei Grundlegende Verfahren wie Copy- und Log-Techniken vorgestellt.
Danach wurden die Technik die Historie eines Schlüssels effizient berechnen kann (Reverse Chaning). Für die Beantwortung der allgemeinen Schlüssel-Zeit Anfragen wurde argumen-

[5] Ist ein R-Baum bei dem sich die Knotenregionen nicht überlappen, dafür werden aber die Duplikate in den Knoten verwaltet.

tiert, dass die beste Methode die Daten sowohl nach Schlüssel als auch nach Zeit zu clustern. Im Allgemeinen wurde angstrebt den Schlüssel-Zeit-Raum in disjunkte Regionen aufzuteilen und denen die Seiten der Externen-Speicher zuzuordnen. Dabei wurde die allgemeine Problematik mit unterschiedlichen Längenverteilungen der Intervalle angesprochen.
Die Strukturen für allgemeine Schlüssel-Zeit Anfragen lassen sich dann in drei Typen unterteilen: Überlappende Regionen (R-Baum ähnliche Verfahen), Disjunkte Rechteckregionen mit Datenduplizierung (B+Baum basierte Verfahren WOBT, MVBT,MVAS, TSB), dreidimensionale Mapping. Es wurde argumentiert, dass die B+Baum basierte Verfahren wie MVBT, MVAS, TSB sich gut für die Indexierung von Transaktionszeit RDF Daten eignen würden, denn die bieten den linearen Speicherplatzverbrauch und logarithmischen Zugriffs- und Anfragekosten. Dabei Spielt auch die Tatsache, dass sie auf den Basis von B+Baum entwickelt sind, das ermöglicht die nahtlose Integration mit den RDF B+Indexen.
Für Validzeit Indexe wurden wiederum die R-Bäume vorgestellt, die für RDF Daten mit nicht also stark variierenden Intervalllängen eingesetzt werden können und Techniken wie (External Interval Tree, Zwei bzw. dreidimensionale Mapping, MAP 21, IR-Tree). Auch Mapping-Techniken in zwei oder dreidimensionalen Raum bei gleichmäßigen Intervalldaten benutzt werden können. Eine andere Alternative stellt der IR-Baum, die Struktur garantiert die logarithmischen Zugriffs- und Anfragekosten und linearen Speicherplatzverbaruch, zudem ist sie leicht zu implementieren.
Im Anschluss wurde spezielle Indexstruktur TGRIN für temporalen RFD Graphen präsentiert. Die Struktur wird über die Ressourcen (Subjekte, Objekte) der temporalen RDF Graphen aufgebaut. Dabei wurde ganz anderes Indexierungskonzept verwendet. Die lässt sich für Graphmatching Anfragen, bei dem die Ressourcen als Konstante vorkommen, einsetzten. Die Struktur ist zwar für exetrnspecher geeignet, hat aber den kleineren Verzweigungsgrad und sie ist statisch, zeigt aber für eingeschränkte Anfragetypen für mittelgroße Datenmengen sehr gut Anfrageperformanz.
Die Besprechung von Bitemporalen Indexstrukturen wurde ausgelassen, da sie in Großen und Ganzem auf vorgestellten Verfahren basieren, die vertiefende Diskussion zu diesem Themen ist in [28][29] zu finden. Es wurden auch die Graphindexe ausgelassen die für die Unterstützung von speziellen Graphanfragen eingesetzt werden können. Wie zum Beispiel die Transitive Hülle, Erreichbarkeit, allgemeine Graphpfadasudrücke. Eine Übersicht über die Komplexität von den Strukturen, die benötigt werden um diesen Typ von Anfragen zu beantworten, bieten [9].
Im nächsten Kapitel geht es tiefer in der Implementierung von temporalen Strukturen mit Hilfe von XXL Java Bibliothek, unter andrem werden die B+Bäume, MVBT und IR-Tree mit R-Bäumen besprochen.

4. Implementierung von Indexstrukturen für zeitabhängige RDF Daten in der XXL Bibliothek

In diesem Kapitel wird die Implementierung der Indexstrukturen für temporale RDF Daten in der Java-Bibliothek XXL beschrieben. Die Bibliothek wurde von der Arbeitsgruppe Datenbanksysteme vom Fachbereich Mathematik und Informatik an der Universität Marburg entwickelt. Diese stellt Methoden und Werkzeuge für die Entwicklung von Datenabsystemkomponenten, unter anderem Indexstrukturen, physische Speicherkomponenten und eine reiche Bibliothek von Operatoren für die Anfrageverarbeitung etc. zur Verfügung.
Im ersten Kapitel werden grundsätzliche Techniken und Klassen für die Entwicklung, Anpassung und Verwendung von Indexstrukturen erläutert. In weiteren Abschnitten wird näher auf die Architektur von den in XXL entwickelten Indexstrukturen für RDF und zeitabhängigen RDF Daten, eingegangen.

4.1. Indexstrukturen in XXL

Neben zahlreichen Klassen und Methoden für die Datenverwaltung beherbergt die Bibliothek das Paket `xxl.core.indexStructures`. Dieses bietet sowohl bereits implementierte Externspeicherstrukturen als auch Werkzeuge und Techniken für die Entwicklung von eigenen Indexstrukukturen.
Die Kernklasse für die Implementierung von baumbasierten Indexstrukuren stellt die abstrakte Klasse `Tree` dar. Die Idee für die Entwicklung von einer gemeinsamen abstrakten Klasse für die baumbasierten Indexstrukturen, mit Hilfe der objektorientierten Architektur, basiert auf dem Konzept von externen Indexstrukturen mit gemeinsamer Funktionalität. Ganz grob lässt sich die Struktur von Bäumen wie folgt beschreiben: Die Blattknoten der Struktur enthalten die Daten, die Indexknoten (die auch Daten enthalten können) verwalten neben den Zeigern auf Kindknoten auch die „Beschreibungsschlüssel“(im R-Baum wäre es z.B. das minimal umgebende Rechteck und im B+Baum der eindimensionale Schüssel z.B. eine Integer-Zahl). Die Daten werden meistens Top-Down eingefügt und der Pfad wird mit Hilfe der „Beschreibungsschlüssel“(Wegweiser) in den meisten Fällen bis zu Blattknoten berechnet. Falls das Einfügen eines Elements den Überlauf einer Knotenseite bewirkt, wird der Knoten aufgeteilt. Diese Aufteilung geschieht durch geeignete Splitstrategie und dabei werden strukturelle Änderungen von unten nach oben propagiert. Die Klasse `Tree` stellt die Grundfunktionalität für die Entwicklung von Indexstrukturen, die nach dem beschriebenen Prinzip arbeiten, bereit[1].
Die Grundbausteine für die Entwicklung von eigenen baumbasierten Strukturen sind die Indexeinträge `Tree.IndexEntry`, Knoten des Baums `Tree.Node` und „Beschreibungsschlüssel“-Interface `Descriptor`. Die Klasse bietet zusätzlich Methoden zum Propagieren von strukturellen Änderungen, abstrakte Methoden zur Top-Down-Navigation (beim Einfügen und Löschen der Daten), abstrakte Methoden zur Behandlung vom Überlauf und Unterlauf der

[1] Paket IndexStructures ist vergleichbar mit GiST Bibliothek http://gist.cs.berkeley.edu/software.html

Knotenseiten und allgemeine Parameter. Zusätzlich sind auch die Initialisierungsmethoden vorhanden, um allgemeine Basisparameter zu initialisieren.
Die wichtigsten Parameter der `Tree`-Klasse, die für die Entwicklung von eigener Lösung auf Basis von der Klasse sind, sind: Methode (Funktion) zur Berechnung des „Beschreibungsschlüssels“ aus Datenobjekten, Methoden (boolesche Funktionen) für das Testen von Überlauf- bzw. Unterlauf-Bedingungen, Parameter für Splitgrenzen, Indexeintrag vom Wurzelknoten, „Beschreibungsschlüssel“ vom Indexeintrag des Wurzelknotens und Container zum Serialisieren der Knoten.

```
protected Function getDescriptor;
protected Predicate overflows;
...
protected Function getSplittMinRatio;
...
protected IndexEntry rootEntry;
protected Descriptor rootDescriptor;
```

Zunächst wenden wir uns kurz den Konzepten `Function`,`Predicate`, da die Parameter zum größten Teil von diesem Typ sind. Die beiden Konzepte wurden in Anlehnung an funktionale Sprachen implementiert. Das erste stellt die direkte Umsetzung einer Funktion, das zweite einer booleschen Funktion dar. Mit Hilfe von dieser Technik können Funktionen als Parameter von Methoden verarbeitet werden. Das Konzept ist vergleichbar mit Funktionen höherer Ordnung in funktionalen Sprachen(z.B. Map Funktion).
Die Funktion `getDescriptor` bildet die Objekte auf die ihren Schlüssel (Wegeweiser) ab. Als Ausgabe wird ein Objekt der Interface `Descriptor` erwartet. Für die Entwicklung von Objekten mit Wegweiser-Funktionalität wird in der Bibliothek ein Interface `Descriptor` bereitgestellt. Interface bietet fünf folgende Methoden für die Entwicklung eigener „Beschreibungschlüssel“: Methoden zum Testen von Überlappung und Enthaltensein mit einem anderen `Descriptor`, Vereinigungsmethode und Methoden für objektorientierte Umgebung wie Vergleichs- und Kopieremethode. Der `Descriptor` ist eines der wichtigsten Konzepte bei der XXL-Indexstrukturen. Vom Interface bereitgestellte Operationen bieten die Grundfunktionalität zur Navigation innerhalb des Knotens der Indexstruktur. Die Testmethoden dienen zum Beispiel dazu die Wahl zu treffen, in welchem Unterbaum die Berechnung fortgesetzt werden soll. Die Vereinigung ist nützlich unter anderem bei der Berechnung des Descriptors eines Knoten.
`IndexEntry` entsprechen den klassischen Indexeinträgen, die neben den Zeigern auf Kindknoten auch den „Beschreibungschlüssel“ enthalten können. Die Indexeinträge werden als innere Klasse implementiert. `Node` ist eine abstrakte innere Klasse eines Baums. Sie ist verantwortlich für die Logik der zu entwickelnden Indexstruktur. Die Überlauf- und Unterlaufbehandlung wird von dieser Klasse verarbeitet. Die Klasse bietet vordefinierte abstrakte Methoden und stellt die Basislogik für die Splitverarbeitung. Die allgemeine Verarbeitung eines Überlaufs lässt sich in folgenden Schritten zusammenfassen:

- Die Einfügemethode platziert den Datensatz in den jeweiligen Knoten. Bei der Einfügemethode wird der Pfad bis zum Zielknoten in einem Stack aufgebaut und aufbewahrt. Danach wird der aufgebaute Stack Schritt für Schritt abgearbeitet.

- Im nächsten Schritt wird geprüft, ob die Überlauf-Bedingung von dem obersten Knoten verletzt ist. Falls es der Fall ist, wird ein neuer Knoten erzeugt. Er kann als Empfänger-Knoten bezeichnet werden. Von dem neu erzeugten Knoten wird dann der Splitalgorithmus aufgerufen. Dieser bekommt als Parameter den Stack mit dem Pfad und im Allgemeinen wird der oberste Knoten (der Knoten, der übergelaufen ist) entnommen

und je nach Semantik der Indexstruktur werden die Daten zwischen dem neu erzeugten und dem alten Knoten verteilt. Die Information über den Split und insbesondere den Indexeintrag von dem neuen Knoten wird im speziellen Objekt `SplitInfo` festgehalten.

- Der neu erzeugte Indexeintrag wird dann in den Elternknoten eingefügt. Die Prozedur wiederholt sich, bis der Stack vollständig abgearbeitet wurde. Es kann auch früher abgebrochen werden, falls keine strukturellen Veränderungen stattgefunden haben.

Für diesen Verarbeitungsprozess stellen die Klasse `Tree` und ihre innere Klasse `Node` Grundfunktionalität bereit.
Die Knoten des Baums werden auf den Externspeicher mit Hilfe eines Objekts der Klasse `Converter` abgebildet. Die abstrakte Klasse `Converter` implementiert das Java-Interface `Serializable` und bietet abstrakte Methoden zum Lesen und Schreiben von Daten auf der Grundlage des Datenstrom-Konzepts (DataStream) von Java. Viele vorimplementierte Klassen existieren bereits in XXL für primitive Datentypen.
Die Speicherkomponente wird in XXL durch das `Container`-Konzept repräsentiert. Das Interface `Container` bietet die Funktionalität zum Einfügen von Objekten und zur Vergabe der Identifaktoren von den zu verwaltenden Objekten. Die Suche nach Objekten erfolgt im Container mit Hilfe von eindeutigen Identifikatoren. Zahlreiche Implementierungen des Interfaces existieren in XXL, angefangen von Hauptspeicher- bis zu Externspeicher-Containern. Natürlich sind die letzteren für die Entwicklung von Externspeicherstrukuren von großer Bedeutung.
Der `BlockFileContainer` verwaltet die Blöcke von festen Bytegrößen im externen Speicher. Die Blöke repräsentieren dann die Speichereinheit im externen Speicher (Speicherseiten). Der zweitwichtigste Container ist der `ConverterContainer`, der für die Abbildung von Objekten auf die Blöcke mit Hilfe der als Parameter übergebenen Datenconverter sorgt.
Die Containerhierarchie wird auf der Basis vom Decorator-Entwurfsmuster implementiert.
Die Blöcke wurden als Objekte mit internem Bytearray und Methoden zum Schreiben und Lesen von Bytes entwickelt.
Bei der Initialisierung eines Baums muss schon vorab bekannt sein, wie groß die Speicherseite ist. In Abhängigkeit von den Blöckgrößen (Speichergröße) werden dann die Überlauf- bzw. Unterlauf-Prädikate initialisiert. Zusätzlich, je nach Architektur, werden die Bytegrößen der zu speichernden Datensätze, Schlüssel und ihre Konverter benötigt. Der Knotenkonverter nutzt dann den übergebenen Datensatz- und Schlüsselkonverter zur Abbildung der Baumknoten auf den Externspeicher. Mit dieser Technik lässt sich ein effizientes Seitendesign gestalten.
Im Allgemeinen kann der Block strukturell in zwei Teile aufgeteilt werden: den Informations- (eng. Header) und den Datenabschnitt. Die Mindestgröße des Informationsteils setzt sich z.B. aus der Knoten-Level-Information und der Anzahl der gespeicherten Datensätze. In den B+Baum-Seiten können zusätzlich die Zeiger auf den Nachbarknoten mitverwaltet werden.
Sei die Seitengröße von dem `BlockFileContainer` 4096 Bytes (4 KB). Angenommen die Anzahl der Daten wird als Java Integer-Wert von 4 Byte und die Level-Information als Java Short-Wert von 2 Byte gespeichert. Dann stehen für die Daten 4090 Bytes zur Verfügung. Aus diesem Wert werden dann die Prädikate für Überlauf- bzw. Unterlaufbehandlung berechnet. Angenommen, dass sowohl die Daten als auch die Schlüssel eine feste Bytegröße besitzen, dann könnte der Knotenkonverter anhand der Information über die Anzahl der Elemente und Levelangaben mit den Daten- bzw. Schlüsselkonvertern die Daten iterativ aus dem gelesenen Block wiederherstellen. Dabei steht einem frei, in welche Hauptspeicher-Collection (Java HashMap, ArrayList) die Daten geladen werden können.
Es gibt auch eine weitere Spezialiseirung der Klasse `Tree` `ORTree`. Diese bietet Basisfunktionalität für die Baumstrukturen, die überlappende Regionen eines mehrdimensionalen Raums

verwalten. Z.B. stellt die Familie der implementierten R-Bäume eine weitere Spezialisierung dieser Klasse dar.
Die `IndexEntry` Klasse wird um die Variable `Descriptor` erweitert, die den Regionschlüssel verwaltet. Auch weitere Klassen wie Knoten, Knoten-Konverter und Basissuchlogik werden von der Klasse bereitgestellt, so dass bei der Entwicklung von den Strukturen wie R-Bäume, M-Bäume etc. ein Großteil des Impelementierungsaufwands abgenommen wird.
Für die Implementierung vom Anfragemodul einer Indexstruktur kann die Klasse `QueryCursor` benutzt werden. Diese ist als eine abstrakte Klasse implementiert, mit dem Ziel die Möglichkeit für die Entwicklung vom eigenen *Lazy*-Verarbeitungssemanitk-Anfrage-Cursor anzubieten. Die abstrakte Klasse enthält neben dem Verweis auf den zu traversierenden Baum auch den Stack für die Auf- und Abstiegmethoden und implementiert den Cursor aus dem Paket `xxl.core.cursors`. Dadurch lassen sich die Suchalgorithmen auf den Indexstrukturen, die durch Erweiterung des QueryCursors implementiert wurden, nahtlos mit den anderen Cursors benutzt werden. Da die Cursorsemanitk die Basis für die Operatoren-Entwicklung bereitstellt, lassen sich auch Indexcursors für die Anfrageverarbeitung als Knoten der Operatoren-Bäume verwenden.
Ein interessanter QueryCursor, der direkt in der Klasse `Tree` implementiert ist, wird durch die Klasse `Tree.Query` implemeniert. Dieser bietet die Möglichkeit den Baum nach zuvor definierter Ordnungsrelation zum Anfrageobjekt z.B. distanzbasierend zu traversieren. Als Parameter erhält der Cursor die Prioritätswarteschlange. Beim Abstieg werden die Daten des aktuell betrachteten Knoten gemäß der Ordungsrelation (Distanz zum Anfrageobjekt) in die Prioritätswarteschlange eingefügt und das zunächst zu betrachtende Element wird aus der Warteschlange entnommen.
Grundsätzlich können folgende Schritte für die Entwicklung von eigenen baumbasierten Lösungen mit Hilfe des `xxl.core.indexStructures` Pakets benutzt werden:

- Entwicklung einer Klasse, die Objekte mit Wegweiserfunktionalität auf der Basis des Interfaces `Descriptor` bereitstellt. Im Weiteren wird das Bereitstellen der Funktion `getDescriptor` benötigt. Dieser bildet Daten auf Wegweiser(Schlüssel) ab.
- Unter Umständen ist die Anpassung der Klasse `IndexEntry` auf die Verwaltung von eigenen Wegweiser(Schlüssel)-Objekten erforderlich (oder falls Zusatzfunktionalität benötigt wird). Ansonsten kann `ORTree.IndexEntry` für Strukturen mit überlappenden Regionen verwendet werden.
- Implementierung der Hauptlogik der Struktur z.B. für Unterlauf- bzw. Überlauf-Behandlung, Suche etc. in der Klasse `Node`, durch Erweiterung der Klasse `Tree.Node` oder `ORTree.Node` etc.
- Bereitstellen der Konverter für Schlüssel und Daten. Entwurf vom physischen Design der Seiten mit Hilfe der Knotenkonverter.
- Entwicklung von Anfrageverarbeitungskomponenten auf Basis von `QueryCursor`.

In den nächsten Abschnitten werden die bestehenden Indexstrukturen aus dem XXL-Paket präsentiert, die die Klasse `Tree` erweitern. Es werden unter anderem die Architektur und Algorithmen der Strukturen erläutert. Insbesondere werden die Anpassungen vertieft beschrieben, die mit der Verwaltung von RDF Daten verbunden sind.

4.1.1. B+Baum in XXL

Für die Indexierung von RDF Daten wurde bei der Arbeit die bestehende `BPlusTree` Klasse als Vorlage für die Entwicklung des B+Baums für RDF Daten verwendet. Der Grund,

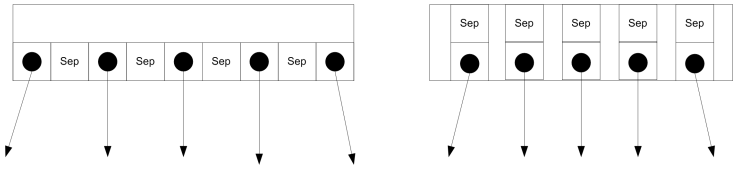

Abbildung 4.1.: Links klassisch, rechts implementiert

weshalb die Klasse `BPlusTree` nicht direkt verwendet wurde, ist, dass der Baum keine variablen langen Datensätze unterstützt. Zuerst wird in diesem Abschnitt kurz die Architektur der Vorlage-Klasse `BPlusTree` präsentiert. `BPlusTree` stellt die Erweiterung der abstrakten Klasse `Tree` dar. Die bereitgestellten Methoden vom Interface `Descriptor` reichen nicht aus für die B+Baum-„Beschreibungsschlüssel“, denn der Baum setzt die Ordnung über die Schlüssel voraus. Für diese Zwecke wurde die abstrakte Klasse `Separator` entwickelt, die zusätzlich neben dem Interface `Descriptor` auch das Interface `Comparable` implementiert. Die Sortierung der Daten erfolgt dann intern mit der `CompareTo()`-Methode. Die innere Klasse `IndexEntry` wurde entsprechend auf die Verwaltung von Separatoren-Objekten erweitert. Im Unterschied zur klassischen Definition vom B+Baum werden statt $n + 1$ Zeigern und n Separatoren n Zeiger und n Separatoren in den Indexknoten des BPlusTrees verwaltet. Zeiger und Separatoren werden gemeinsam durch das `IndexEntry`-Objekt gespeichert (vgl. Abbildung 4.1). Dieser Zeiger wird durch das Container-Id vom serialisierten Knoten dargestellt. Die Knoten des Baums erweitern die Knoten vom `Tree.Node`.
Die Indexeinträge und die Daten werden in einer von Java API bereitgestellten ArrayList-Kollektion verwaltet: Das ermöglicht die effiziente binäre Suche innerhalb der Knoten.
Der B+Baum kann sowohl für Primär-Schlüssel als auch für Sekundär-Schlüssel verwendet werden. Der Baum bietet die Möglichkeit die Duplikate im Bezug auf Schlüssel zu verwalten. Grundsätzlich gibt es verschiedene Methoden zur Verwaltung von Daten mit Duplikatenschlüsseln.
Eine Idee wäre, die Duplikate in die externen Listen auszulagern und darauf über die Zeiger in den Blattknoten des Baums zuzugreifen. Eine andere Möglichkeit ist Duplikate direkt in dem B+Baum zu verwalten, unter Voraussetzung, dass die Anzahl der Duplikate relativ klein im Vergleich zu der Gesamtanzahl der Daten ist. Dabei muss der interne Suchalgorithmus umgestellt werden, da die direkte Binärsuche nicht mehr möglich wäre. Diese Option ist auch im XXL B+Baum implementiert.
Die Duplikat-Option wird als boolesche Variable im Konstruktor eingestellt. Der interne Suchalgorithmus innerhalb der Knoten läuft dann in Abhängigkeit von diesem Parameter.
Bei der Indexierung der RDF Daten können Duplikate auftreten, falls der Index partiell nur über (S, O) oder (S, P) etc. aufgebaut werden muss. Mit der vorigen Argumentation könnten diese Daten vollständig durch einen B+Baum indexiert werden.
Eine ganz andere Möglichkeit für die Verwaltung von Duplikaten in einem B+Baum wäre Duplikate in den Indexknoten komplett zu verbieten und für die Blattknoten ähnlich der ersten Option Duplikate in Listen von Seiten auszulagern.
Die Grundidee dabei ist, die Duplikatdaten mit gleichem Schlüssel nicht sofort auszulagern, sondern die Daten so lange zu behalten, bis ihre Anzahl eine bestimmte Schranke d an Duplikaten von einem Schlüssel überschreitet. Der Parameter d kann so gewählt werden, dass er diese Ungleichung $d >= B - 2 \cdot b$ mit $b < B/2$ erfüllt(z.B. wäre etwa $b = B/3$ eine gute Wahl, die Speicherplatzausnutzung $b = B/3$ für die ausgelagerten Listen kann dann garantiert werden). Die Auslagerung kann dann verzögert geschehen und zwar erst dann, wenn der Überlauf der Seite behandelt werden muss. Dann könnte folgende Strategie angewendet werden: Es wird in einem Knoten nach einem zusammenhängenden Bereich von Daten mit

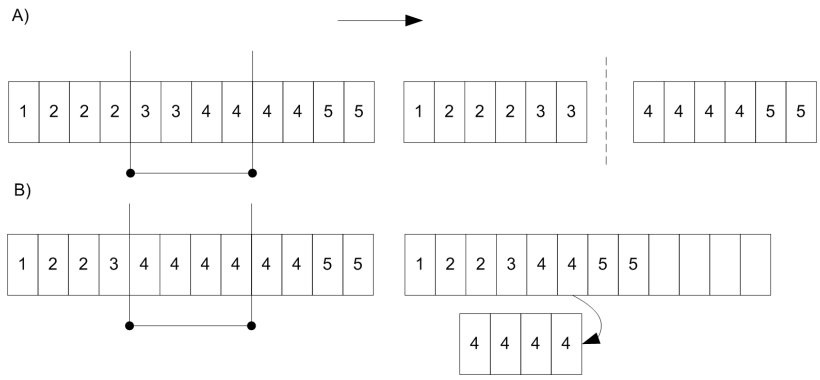

Abbildung 4.2.: Verwaltung von Daten mit Duplikatschlüssel mit der verzögerten Auslagerung. A) Falls kein zusammenhängender Bereich gefunden wird, führe normalen B+Split. B) Lagere den Duplikatenbereich aus.

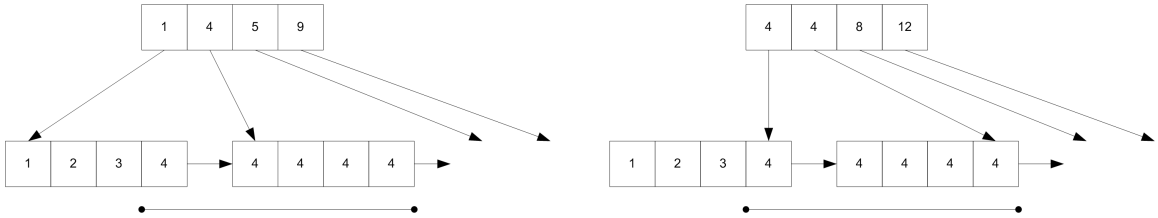

Abbildung 4.3.: Wahl der Schlüssel, links Minimum, rechts Maximum

gleichem Schlüssel gesucht, der größer als d ist. Falls es nicht der Fall ist, wird ein normaler Split durchgeführt. Die Wahl des Parameters d und der Mindestanzahl b garantieren, dass eine Aufteilung gefunden wird, bei der keine gleichen Schlüssel für die Separatoren verwendet werden(vgl. Abbildung 4.2). Falls ein Duplikatenbereich gefunden wird, kann er entweder komplett oder es können mindestens d Elemente aus dem gefundenen Bereich ausgelagert werden.

Zusätzlich sind die Knoten des B+Baums nicht nur in der Blattebene, sondern auch allgemein einseitig verkettet. Das ermöglicht nicht nur effiziente Bereichsanfragen, sondern auch die Implementierung von Nebenläufigkeitsalgorithmen in einer Transaktionsumgebung.

Ähnlich der klassischen Definition des B+Baums kann der Schlüssel eines Knotens entweder als maximales oder als minimales Element dienen. In der implementierten Version wird die erste Option benutzt. Diese Wahl ist begründet durch die Verwaltung von Duplikaten. Im Duplikaten-Modus wird jede Suche als Bereichssuche betrachtet, es wird stets nach dem linken Element aus dem Duplikatenbereich gesucht. Durch die Verwendung vom minimalen Schlüssel muss stets ein Knoten links von dem durch den internen Suchalgorithmus gelieferten Kindknoten betrachtet werden. Mit maximalem Element als Schlüssel kann der Kindknoten direkt benutzt werden (vgl. Abbildung 4.3).

Die Split- und Merge-Logik des B+Baums wird in der zugehörigen Klasse `Node` implementiert. Der verwendete Splitalgorithmus ist der Standard-Splitalgortihmus vom B+Baum mit Schlüsseln fester Länge. Die Aufteilung geschieht nach der Anzahl der Elemente in den Knoten. Der Split wird durch den Parameter gesteuert, der die Mindestanzahl der Elemente nach dem Split angibt. Per Default wird er mit 0.5 initialisiert.

Im Duplikaten-Modus wird eine andere Strategie benutzt, dabei wird versucht den möglichst größeren zusammenhängenden Bereich von Duplikaten zusammen zu erhalten. Diese Strategie ist dadurch begründet, dass die exakte Suche in Duplikaten-Umgebung sich in die

Bereichssuche entartet, dabei wird versucht eine möglichst kleine Anzahl von Knoten in der Blattebene zu besuchen.
Das Löschen von Elementen kann zum Unterlauf der Seiten führen. In vielen B+Bäumen in kommerziellen Systemen werden keine strukturellen Änderungen vorgenommen, sondern in periodischen Abständen in einem Batch-Modus die Speicherplatzausnutzung der Knoten geprüft und dabei wird der gesamte Baum reorganisiert. Dadurch kann die Performanz von nebenläufigen Transaktionen gesteigert werden, da durch die lokale Reorganization während des Mergealgorithmus der Pfad zum unterlaufenen Blattknoten gesperrt werden muss. Bei der kleineren Anzahl von Transaktionen kann auch die lokale Reorganisation stattfinden. Dabei wird die unterlaufende Seite mit der Nachbar-Seite gemerget. Im Duplikaten-Modus kann die Situation auftreten, bei der das zu löschende Element nicht in dem berechneten Knoten liegt. In diesem Fall wird das zu löschende Element mit seinem linken Duplikat-Partner aus dem Duplikatenbereich vertauscht. Dadurch kann der normale Mergealgorithmus auf dem berechneten linken Blattknoten durchgeführt werden.
Die Anfrageverarbeitungslogik wird durch den `QueryCursor` umgesetzt. Der Cursor wird nach der *Lazy*-Semantik implementiert und kann sowohl Punkt- als auch Bereichsanfragen beantworten. Der gesuchte Schlüssel-Bereich der Anfrage wird durch eine spezielle Erweiterung (`KeyRange`) der Wegweiser-Klasse `Separator` implementiert. Die Klasse enthält den Anfangs- und Endschlüssel des zu suchenden Bereichs. Der zuvor angesprochene allgemeine Parameter von `Tree RootDescriptor` wird in `BPlusTree` als `KeyRange` kodiert. Das ermöglicht sowohl Punkt- als auch Bereichsanfragen sofort zu beantworten, wenn der Anfragepunkt oder der Anfragebereich außerhalb des durch den Baum verwaltenden Schlüsselbereichs liegen.
Ansonsten wird der QueryCursor mit einem Anfragebereich initialisiert und durch die `hasNext`- und `next`-Methoden gesteuert. Die Bereichsanfrage läuft nach dem klassischen Prinzip, zuerst wird mit dem linken Anfangsschlüssel des Anfragebreichs der Blattknoten gesucht, danach wird die Verketunng der Knoten ausgenutzt. Mit Hilfe vom implementierten `QueryCursor` lässt sich auch der IndexScan-Operator implementieren, dabei könnte `KeyRange` von dem Wurzel-Knoten als Anfragebereich benutzt werden.

4.1.2. B+Baum für variabel lange Schlüssel in XXL

Für die Indexierung von RDF Daten wurde der B+Baum auf die Verwaltung von variabel langen Datensätzen erweitert, da die RDF Daten die Zeichenketten variabler Länge haben. Die Klasse `VariableLengthBPlusTree` erweitert die Klasse `Tree` und besitzt ähnliche Gesamtarchitektur wie `BPlusTree`. Die wesentlichen Unterschiede sind die Unterlauf- bzw. Überlauf-Behandlung in Abhängigkeit von der belegten Bytegröße und Umsetzung der Split- und Merge-Logik mit Strategie-Pattern.
Die Berechnung der physikalischen Bytegrößen von einem Datensatz und seinem Schlüssel erfolgt mit Parameter-Funktionen `getActualKeySize` bzw. `getActualEntrySize`. Das sind die generischen Funktionen, die als Eingabe ein Objekt erwarten und als Ausgabe eine Integerzahl zurückgeben, die die serialisierten Bytegrößen des Objekts angibt. Dieser Ansatz erlaubt die generische Implementierung der Berechnungsmethode. Z.B. falls die Bytelänge der serialisierten Objekte konstant ist, kann die Funktion als Konstante definiert werden.
Die Knoten des Baums sind um eine Variable erweitert, die die aktuelle belegte Bytegröße der Seiten angibt. Diese wird aber nicht explizit physikalisch gespeichert. Die Klasse `NodeConverter`, die Serialisierungslogik von Baumknoten implementiert, initialisiert die Variable aus den einzelnen Indexeinträgen beim Lesen der Knoten aus dem externen Speicher (bzw. aus dem `Container`).
Durch die Umstellung auf die Verwaltung von variabel langen Datensätzen wird die Entscheidung für den Split mit dem Parameter `minAllowedLoad` kontrolliert. Im Unterschied

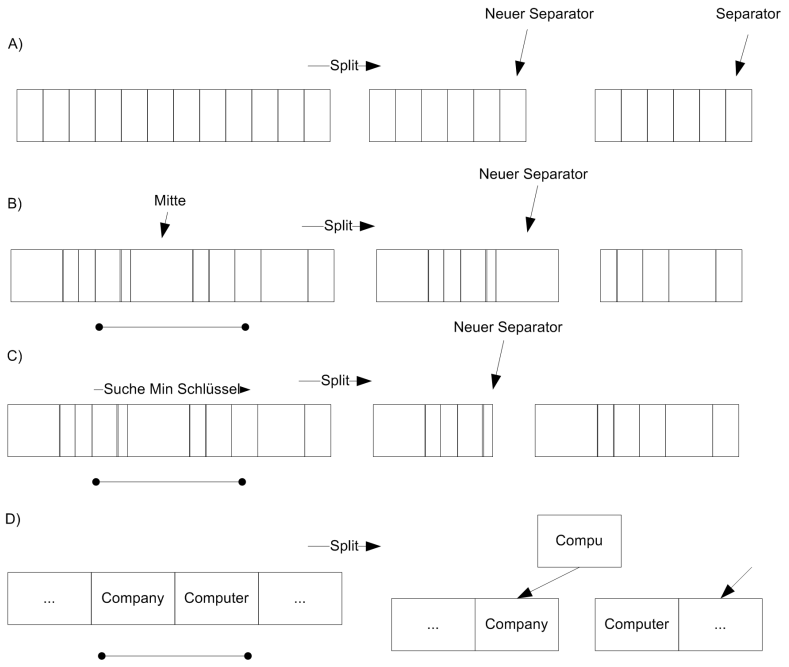

Abbildung 4.4.: Verschiedene Splittstrategien. A) B+Baum mit Datensätzen fester Länge. B) Problem bei der direkten Anwendung von B+Split für Datensätze fester Länge. C) Minimaler-Schlüssel-Splitstrategie D) Simple-Präfix-Splitstrategie

zu Datensätzen fester Länge kann der Standardwert von $b = B/2$ nicht eingehalten werden, da die Split-Entscheidung nicht anhand der Anzahl, sondern anhand der Bytelänge von Datensätzen getroffen wird. Aus diesem Grund mussten die Standard-Splitstrategien auf die neue Architektur umgestellt werden. Dabei bieten sich verschiedene Splitstrategien in Bezug auf die Verwaltung von variabel langen Datensätzen an. Einer der kleinen Nachteile bei dem implementierten `BPlusTree` ist, dass die Split- und Mergelogik fest in den Knoten der Indexstruktur verankert ist. Für die Umsetzung der neuen Split- oder Mergelogik musste die Unterklasse der Klasse `BPlusTree` erstellt werden. In `VariableLenghtBPlusTree` wurde sowohl Splitlogik als auch allgemein die Unterlaufbehandlung als Strategie-Pattern umgesetzt. Der Baum kann dann mit ausgewählter Strategie für Split- oder Unterlaufbehandlung initialisiert werden. Die folgenden Strategien wurden für den B+Baum entwickelt (vgl. Abbildung 4.4):

- Einfache Strategie, die dem Split vom B+Baum mit Datensätzen fester Länge entspricht. Dabei wird der Datensatz nach dem vorgegebenen Parameter `averageLoadRatio` ausgesucht, der die Durchschnittsanzahl der Bytes pro Knoten angibt. Bei den Datensätzen, die sich nicht sehr stark der Länge nach unterscheiden, kann diese Strategie verwendet werden.

- Die Verbesserung der oberen Strategie ist die Suche nach dem Schlüssel mit minimaler Länge. Der Splitalgorithmus sucht den Datensatz zwischen Grenzen, die durch `minAllowedLoad` (b) Parameter vorgegeben sind, nach dem Schlüssel mit der kleinsten Bytelänge (z.B. wird mit $b = B/3$ zwischen b und $B-b$ gesucht). Durch die Verwendung dieser einfachen Strategie können die Indexknoten mehr Schlüssel aufnehmen. Entsprechend steigt der Verzweigungsgrad des Baums und damit reduziert sich die Höhe des Baums (bzw. die Anzahl der I/O-Zugriffe bei den Algorithmen).

- Im Kapitel zuvor wurden die Präfix-B+Bäume besprochen, die sich gut für die Verwal-

tug von RDF Daten eignen würden. Eine der simplen Strategien dabei ist, die Suche nach dem kürzesten Präfix zwischen aufeinander folgenden Zeichenketten. Dieses Präfix erfüllt alle Eigenschaften des Separators (z.B. bei den Zeichenketten „Company“ und „Office“ reicht es, um die Wegweiserfunktion zu erfüllen, je nach Architektur „O“ oder „C“ als Wegweiser zu verwenden).

Das Strategie-Entwurfsmuster erlaubt sehr flexibel neue Splittechniken zu verwenden.
Für das Verwalten von Schlüsseln aus zusammengesetzten Attributen wurde eine spezielle Klasse bereitgestellt. Die Objekte dieser Klasse verwalten intern ein Objekt-Array und implementieren die Vergleichsmethode, die lexikographisch die Elemente des Arrays vergleicht. Die flexible Architektur der IndexStrukturen erlaubt, eine Funktion `getDescriptor` so zu schreiben, dass das zu speichernde Objekt auf den Schlüssel mit zusammengesetzten Attributen abgebildet werden kann. Auf diese Weise lassen sich unter anderem auch die Partitions-B+Trees implementieren. Im einfachsten Fall können Indexeinträge als Tupel mit führender Partitionsnummer als numerischem Datentyp und mit eigentlichem Schlüssel implementiert werden. In der RDF-Umgebung können die Partitions-B+Bäume beim kleineren Selektivitätswert der Prädikaten-Werte verwendet ((P, S, O), (P, O, S)) werden. Das Strategie-Pattern für Splitdurchführung erlaubt auch den Simple-Präfix-B+Baum-Algorithmus zu erweitern und mit Schlüsseln mit zusammengesetzten Attributen zu verwenden.
Eine andere Möglichkeit innerhalb des XXL-B+Baums die variabel langen Datensätze zu verwalten wäre sie direkt auf Bytearrays abzubilden. Dafür wird eine Funktion, die Datensätze auf Bytearrays abbildet, benötigt. Der Vorteil dieser Verwaltung sind schnelle Vergleichsoperationen auf Byte-Ebene.

4.1.3. MVBT in XXL

Im vorigen Kapitel wurde argumentiert, dass die Transaktionszeit-Indxstruktur MVBT sich gut für die Verwaltung von zeitbezogenen RDF-Daten in der Transaktionszeitumgebung eignet. Die zu versionierende Einheit wird als zusammengesetzter Schlüssel mit drei Komponenten (S, P, O) verwendet. Bereits im vorigen Kapitel wurden grob die Einfüge- und Lösch-Algorithmen vorgestellt. In diesem Abschnitt wird die Architektur von MVBT und ihre Erweiterung auf die Verwaltung von variabel langen Schlüsseln beschrieben
Die Klasse `MVBTree` erweitert die Klasse `BPlusTree`. Analog wurde `VariableLengthMVBT` implementiert als Erweiterung der Klasse `VaribaleLengthBPlusTree`. Entsprechend der Semantik der Transaktionszeit wurde der Wegweiser `Separator` auf die Verwaltung von Zeitintervallen erweitert. `MVSeparator` enthält neben dem Schlüssel auch ein halboffenes Zeitintervall, das durch die Klasse `Lifespan` repräsentiert wird. Auch die Klasse `KeyRange` wurde auf `MVRegion` umgestellt. Das Objekt von dieser Klasse wird auch in der Variable `rootDescriptor` verwaltet. Diese Variable beschreibt die Region des Schlüssel-Zeit-Raums, die durch den MVBT aktuell verwaltet wird, und dient unter anderem der Anfrageverarbeitung.
Die Bereichsanfragen werden intern auf Objekte dieser Klasse abgebildet. `QueryCursor` mit diesem Objekt parametrisiert. Die Index-Einträge wurden nicht erweitert, da die Gesamtlogik durch den `MVSeparator` implementiert ist.
Intern werden die Daten in einem speziellen Objekt `LeafEntry` gepackt. Das Objekt verwaltet neben dem eigentlichen Datensatz auch das Zeitintervall `LifeSpan`. Beim ersten Einfügen des Datensatzes erwartet die Einfügemethode neben den eigentlichen Daten auch ein Objekt des Interfaces `Version`. Der Datensatz und die Einfügeversion werden auf `LeafEntry` abgebildet und der Anfangspunkt des Zeitintervalls mit „Beginverison“ initialisiert. Die Löschmethode erwartet analog den Zeitpunkt als Parameter.

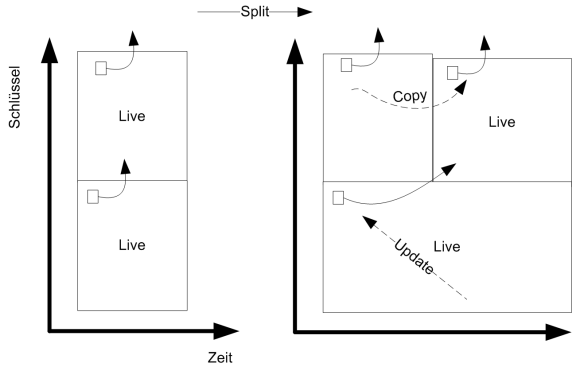

Abbildung 4.5.: Einseitige Verlinkung. Nach dem Versionssplit der oberen Seite muss der Zeiger der unteren Seite korrigiert werden. Dafür muss aber die Seite lokalisiert und aus dem Speicher geholt werden.

Das zu löschende Objekt wird in dem „live“-Baum gefunden und der Endpunkt des Zeitintervalls vom `LifeSpan` mit der Löschversion initialisiert. Ab diesem Punkt gilt der Datensatz als gelöscht, ansonsten ist er „live“ (`null`-Wert signalisiert, dass Objekt noch nicht gelöscht wurde).
Die historischen Wurzeln werden im speziellen B+Baum verwaltet. Die Wurzel-Einträge werden in den Objekten der Klasse `Root` gepackt. Das Objekt enthält den Verweis auf den historischen Wurzeleintrag und das Objekt, das die Region von dem Schlüssel-Zeit-Raum repräsentiert. Die `Root`-Objekte werden mit dem Anfangspunkt des Zeitintervalls indexiert. Der geerbte Parameter `RootEntry` verwaltet die Wurzel von den „live“-Daten. Bei historischen Anfragen wird zuerst die Wurzel des B+Baums abgefragt und die Einstiegsknoten für die Anfrageverarbeitung geliefert.
Die Knoten vom MVBT stellen neben der Logik für Unterlauf- und Überlaufbehandlung auch die Methoden für die Verwaltung von den Einträgen mit einem Zeitintervall. Im Unterschied zum implementierten B+Baum wird der Wegweiser für die Schlüssel-Dimension als minimaler Schlüssel genommen.
In der gleichen Version sind Schlüsselduplikate nicht erlaubt. Die Einträge innerhalb des Knotens sind lexikographisch nach dem Anfangspunkt des Zeitintervalls und dem Schlüssel sortiert (eine andere Option wäre umgekehrt zu sortieren) (vgl.Abbildung 4.6).
Im Gegensatz zum B+Baum bereitet die Implementierung der einseitigen Verlinkung der Knoten in der Schlüssel-Dimension Schwierigkeiten. Insbesondere die Verkettung von „live“-Knoten ist von großer Bedeutung. Wäre so eine Verlinkung möglich, könnten die Bereichsanfragen an „live“-Daten problemlos mit einem B+Baum-Algorithmus durchgeführt werden. Da die Seiten den zweidimensionalen Regionen des Schlüssel-Zeit-Raums entsprechen, kann die Verkettung in eine Richtung nicht ohne weiteres implementiert werden. Die Schwierigkeiten bereiten die Versionssplits. Da nach dem Split eine Seite mit kopierten „live“-Daten erzeugt wird, muss der Verweis auf die Vorgänger-Seite auf die neue Seite umgeleitet werden. Falls der Versionssplit die rechte Seite in Bezug auf die Schlüssel-Dimension betrifft, muss die linke Schlüssel-Nachbar-Seite aus dem Speicher geholt werden, um den Verweis zu korrigieren(vgl. Abbildung 4.5). Weitere Diskussion über die Verkettung auf der Blattknoten-Ebene wird im nächsten Abschnitt besprochen.
Allerdings sind die Knoten chronologisch verlinkt (vgl.*reverse chaining*). Diese Verlinkung spielt eine entscheidende Rolle bei der Verarbeitung von Schlüssel-Zeit-Bereichsanfragen und *Pure-Key*-Anfragen.
Im weiteren Verlauf des Abschnitts wird die Implementierung des Anfragealgorithmus näher

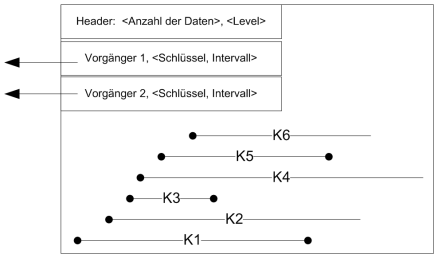

Abbildung 4.6.: Physisches Layout der Seiten in MVBT XXL

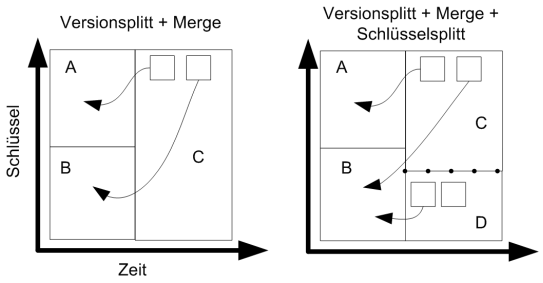

Abbildung 4.7.: Chronologische Verkettung der Knoten in MVBT

beschrieben. An dieser Stelle wenden wir uns der Organisation chronologischer Verkettung zu. In jedem Knoten ist ein Platz reserviert, um höchstens zwei Zeiger auf zeitliche Vorgänger-Knoten zu speichern (vgl. 4.6). Die Begründung, dass es höchstens zwei Vorgänger gibt, ist, dass nach strukturellen Änderungen durch Versionsplit genau ein temporaler Vorgänger entsteht. Falls aber der anschließende Merge-Schritt folgt, bekommt der aktuelle Knoten neben den kopierten „live“-Daten aus seinem direkten zeitlichen Vorgänger auch die „live“-Daten von dem Schlüssel-Nachbarn seines Vorgängers. Damit verwaltet der neue aktuelle Knoten die „live“-Daten aus zwei Vorgänger-Knoten.
Falls jedoch ein weiterer Schlüssel-Split folgt, kann es auch dazu führen, dass einer der neuen Knoten zwei Zeiger verwalten muss. Nach einem Merge-Schritt wird geprüft, ob die starke Überlauf-Bedingung verletzt ist. Falls ja, wird dann nach dem Schlüssel gesplittet. Die Schlüsselgrenze wird durch die verwendete Splitstrategie entweder angehoben oder nach unten gedrückt. Dabei wird einer der beiden Knoten jeweils die „live“-Daten von den beiden Knoten verwalten (vlg. Abbildung 4.7). Neben den Zeigern wird zusätzlich der Schlüssel und das Zeitintervall von den Vorgänger-Knoten verwaltet. Die gesamte Information über die zeitlichen Vorgänger wird in `IndexEntry`-Objekte eingepackt. Die Angaben über die Schlüssel-Zeit-Region, die durch diese Objekte gespeichert sind, helfen bei der Anfrageverarbeitung. Anhand dieser Information kann schnell entschieden werden, ob die Vorgänger-Seite verarbeitet werden soll. Im Weiteren werden die Anfragealgorithmen beschrieben.
Die Punktanfragen, die nur „live“-Daten betreffen, werden mit effizienter Implementierung vom B+Baum-Algortihmus ausgeführt. Die Berechsanfragen an „live“-Daten werden aber durch den allgemeinen `QueryCursor` implementiert. Die fehlende Verkettung in der Schlüssel-Dimension der Blattebene macht den Einsatz vom Standard-B+Baum-Algorithmus unmöglich, stattdessen wird der Tiefendurchlauf verwendet(vgl. Abbildung 4.8). Die interne Implementierung von der Klasse `QueryCursor` dient der Beantwortung von verschiedenen Bereichsanfragen. Das Objekt der Klasse wird mit dem zuvor besprochenen `MVRegion`-Objekt initialisiert. Das Objekt repräsentiert die Anfrageregion. Im Folgenden wird der Schlüssel-Zeit-Anfragealgorthmus exemplarisch kurz skizziert. Alle anderen Algorithmen wie reine zeitliche (Punkt)-Bereichsanfrage oder Schlüssel-Bereichsnafrage an „live“-Daten sowie his-

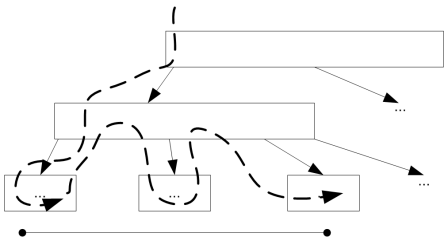

Abbildung 4.8.: Exemplarische Verarbeitung von einer Bereichsanfrage in MVBT in „live“-Daten. Durch fehlende Verkettung wird ein Tiefendurchlauf ausgeführt.

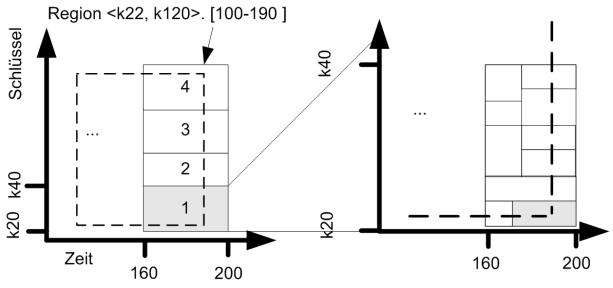

Abbildung 4.9.: Schlüssel-Zeit-Anfrageverarbeitung in MVBT

torische Schlüssel-(Punkt)-Bereichanfrage werden auf ähnliche Weise verarbeitet.

Bei der Verarbeitung von allgemeinen Schlüssel-Zeit-Anfragen $((x_{\min_k}, x_{\max_k}), (y_{\min_t}, y_{\max_t}))$ wird eine Art *Sweep-Line*-Verfahren verwendet. Allgemein werden zuerst mit der rechten Grenze der Schlüssel-Zeit-Anfrage-Region Startblattknoten gesucht. Nachdem die Blattknotten, die die Grenze schneiden, berechnet wurden, wird die Berechnung durch die Ausnutzung der Vorgängerverkettung in Richtung der linken Grenze (chronologisch rückwärts) fortgesetzt. Daher wird zuerst mit dem Endpunkt vom Zeitintervall $y_{\max_t}$ eine Abstiegs-Wurzel im Wurzelbaum gesucht. Ganz grob beschrieben wird danach im Tiefendurchlauf der Baum traversiert. Bildlich gesprochen wird eine Scheibe ausgeschnitten, die aus Knoten besteht (Wurzel der Unterbäume), die $y_{\max_t}$ in Zeit- und $(x_{\min_k}, x_{\max_k})$ Schlüsselachse schneiden. Danach wird die Links-Tiefen-Suche (bezogen auf Schlüsselachse) bis auf die Blattebene fortgesetzt. Wird der Blattknoten aus dem Bereich $y_{\max_t}$ in Zeit- und $(x_{\min_k}, x_{\max_k})$ Schlüsselachse betrachtet, wird die Suche chronologisch rückwärts auf der Blattebene bis zu den Knoten, die außerhalb der $y_{\min_t}$ liegen, fortgesetzt. Genau an dieser Stelle spielen die Vorgängereinträge ihre Rolle. Dadurch, dass neben dem Zeiger auch die Region des Vorgängers gespeichert ist, kann der Algorithmus ohne den Zugriff auf den externen Speicher über die Fortsetzung der Suche in Rückwärtsrichtung Zeitdimesion entscheiden (vgl. Abbildungen 4.8,4.9, 4.10). Durch das Kopieren der Daten durch Versionsplit kommen natürlich Duplikate vor. Die Eliminierung der Duplikate geschieht durch die *Reference-Punkt*-Methode(vgl. Abbildung 4.10). Als Reference-Punkt wird der rechte Endpunkt des Zeitintervalls verwendet. Bei live-Daten oder Intervallen, die die rechte Regiongrenze schneiden, wird der Schnittpunkt mit der Anfrageregion genommen. Liegt der Reference-Punkt in dem aktuell betrachteten Knoten, wird dieser ausgegeben, ansonsten von der Berechnung ausgelassen. Im Folgenden wird versucht den Aufwand für den Algorithmus abzuschätzen.

Der aufwendigste Teil der Berechnung, wie aus dem Verfahren abzulesen ist, ist der Tiefendurchlauf für die Berechnung der Schnittmenge von Blattknoten mit rechter Anfrageregiongrenze. Dabei werden alle Knoten des Unterbaums betrachtet, der durch die Schnittmenge

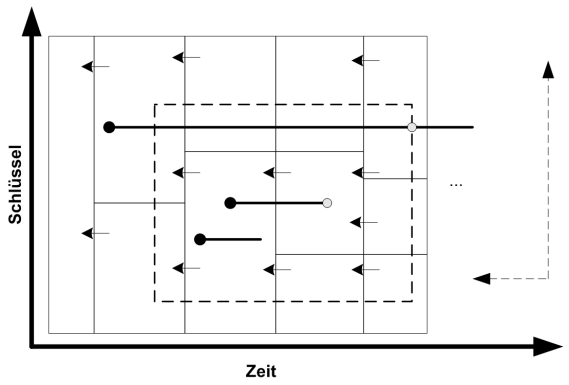

Abbildung 4.10.: Reference-Punkt-Methode. Die graue Punkte stellen die Reference-Punkte dar. Mit dieser Methode werden die Duplikate bei der Ausgabe vermieden.

der rechten Grenze aufgespannt worden ist. Der Gesamtaufwand setzt sich aus der Suche im Wurzelbaum, der Berechnung der Blattknoten für die rechte Grenze und der *Sweep-Line* in Rückwartsrichtung der Zeitachse. Sei $((x_{\min_k}, x_{\max_k}), (y_{\min_t}, y_{\max_t}))$ Anfrageregion und $m_{\max_t}$ Anzahl der Elemente, die "live" sind, zum Zeitpunkt, der dem Endpunkt von dem Zeitintervall der Anfrageregion entspricht. Zu dem sei j Gesamtanzahl der Versionen, die durch MVBT verwaltet werden. Sei r_k die Anzahl der Elemente, die durch den Schlüssel-Bereich der rechten Grenze aufgespannt sind, und r die Gesamtanzahl der Elemente, die die Anfrageregion schneiden. Der Parameter d steht für die Mindestanzahl der Elemente in den Knoten in Version i und b ist wie üblich der Füllungsparamter des B+Baums. Dann betragen die Kosten für die Suche nach dem Abstiegspunkt im Wurzelbaum $O(\log_b j)$. Die Kosten für die Berechnung der Wurzel des Unterbaums, der r_k Elemente aufspannt, setzen sich wie folgt zusammen: Die Höhe des Unterbaums ist $O(\log_d r_k)$. Dann beträgt der Aufwand für die Berechnung dieser Wurzel $O(\log_d m_{\max_t} - \log_d r_k) = O(\log_d m_{(\max_t}/r_k))$. Durch die fehlende Verlinkung in der Schlüsseldimension werden alle Knoten des Unterbaums mit der Tiefensuche betrachtet. Die Gesamtanzahl der Knoten im Unterbaum beträgt im schlimmsten Fall etwa

$$O(d + \frac{d^{(\log_d(r_k/d)+1)} - 1}{d-1}) = O(\frac{(r_k/d)\cdot d - 1}{d-1}) = O(\frac{r_k - 1}{d-1})$$

Damit ergibt sich der Gesamtaufwand asymptotisch geschätzt:

$$O(\log_b j + \log_d(m_{\max_t}/r_k) + \frac{r_k - 1}{d-1} + r/d)$$

Der Aufwand könnte verbessert werden, wenn es die Verlinkung in Schlüssel-Dimension geben würde. Die einseitige Verkettung in der Schlüsselachse kann aber nur in den „live"-Blattknoten implementiert werden (vgl. Abblidung 4.5). Ein kleines Problem gibt es auch dabei und zwar die Lokalisierung der Schlüssel-Nachbar-Blattseiten für die Korrektur des Zeigers. Problematisch wird es dann, wenn der Nachbar in den anderen Elternknoten liegt. Das erfordert die Traversierung des Baums bzw. zusätzliche I/O-Zugriffe. Im schlimmsten Fall (falls der kleinste Eintrag von der Seite in Bezug auf Schlüssel vom Versionssplit betroffen ist) müsste der Pfad von der Wurzel bis zum unteren Nachbarknoten berechnet werden. Eine andere Alternative wäre es die doppelseitige Verkettung zu implementieren. Wiederum kann diese Lösung nur für die „live"-Daten verwendet werden. Mit der doppelseitigen Verkettung kann aber garantiert werden, dass es höchstens zwei zusätzliche I/O-Zugriffe erforderlich sind, um die Verlinkung aufrecht zu erhalten. Mit beiden Lösungen wäre es möglich den Aufwand der Bereichsnafagen, die in „live"-Daten anfangen, zu reduzieren. Anders als bei historischen

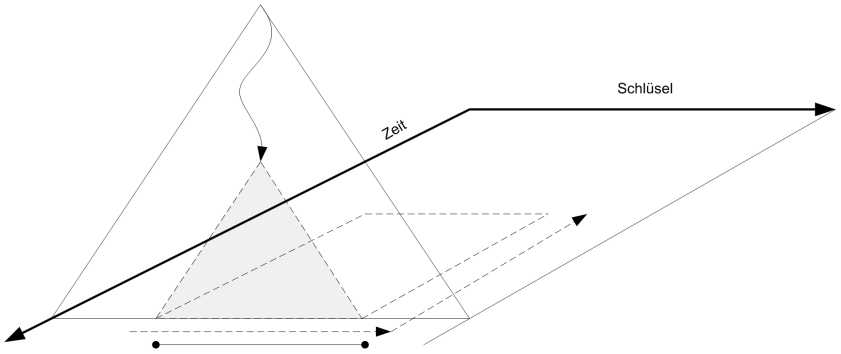

Abbildung 4.11.: Berechnungsablauf für Schlüssel-Zeit-Anfragen. In dem schattierten Dreieck müssen alle Knoten besucht werden.

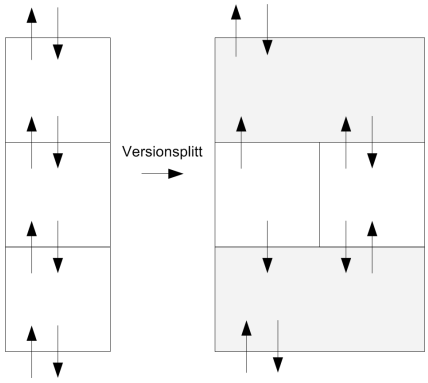

Abbildung 4.12.: Verwendung der doppelseitigen Verkettung erfordert höchstens zwei zusätzliche I/O-Zugriffe.

Zeigern auf Vorgänger-Knoten gibt es keine Notwendigkeit zusätzlich zu dem Zeiger auch die kompletten Seitenregionsinformation zu speichern. Damit wird der Infromationsteil der Seite nur minimal vergrößert. Angenommen sei m_l die Anzahl der „live“-Daten und r_k und j wie gehabt, dann reduziert sich der Gesamtaufwand der Bereichsanfragen, die im „live“-Baum anfangen, auf:

$$O(\log_b j + \log_d m_l + r/d)$$

Die Umstellung auf die Verwaltung von variabel langen Schlüsseln wurde auf ähnliche Weise wie beim B+Baum gemacht. Allerdings ist der frei verfügbare Raum einer Seite nicht konstant. Der Informationsteil einer Seite verwaltet die Vorgänger-Information samt der Schlsselzeit-Regionen. Schlüssel können wiederum variabel lang sein. Dementsprechend beziehen sich die Parameter wie e und d, mit denen die Bedingungen für starken Versions-Überlauf bzw. Unterlauf initialisiert werden, auf aktuell frei verfügbaren Raum. Die verfügbare Kapazität setzt sich zusammen aus Seitengröße in Bytes minus Level, Anzahl der Daten und Vorgängerinformation.
Die Parameterberechnung von MVBT für Schlüssel mit fester Länge wie d, e und k lässt sich auf ähnliche Weise durchführen. Allerdings wird der Parameter d (im Unterschied zu MVBT keine ganze Zahl) $0 < d < 0.5$ als Anteil von „live“-Daten (Bytes) von einer Seite betrachtet. Sei $F = 1 = k \cdot d$ der verfügbare Raum einer Seite. Dann sollte für den starken Versionsüberlauf gelten: $k \cdot d - e \cdot d = (1/d - e) \cdot d = (1 - e \cdot d)$. Für einen Füllungsfaktor von

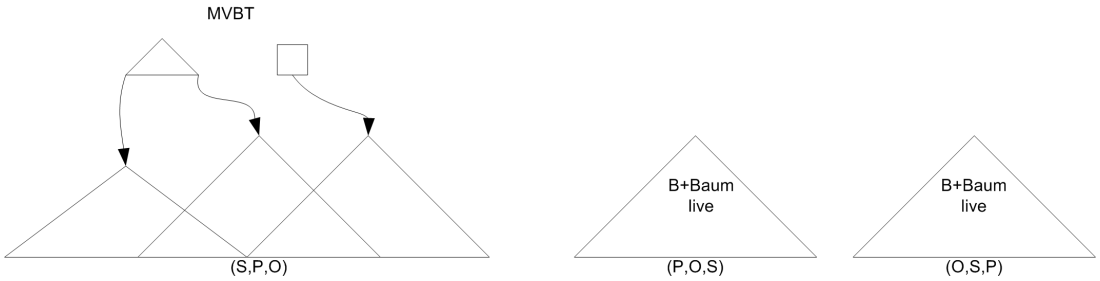

Abbildung 4.13.: Indexarchitektur für RDF Daten in Transaktionszeitumgebung

B+Baum c mit $0 < c \leq 0.5$ sollte dann für einen Schlüsselsplit gelten:

$$(1 - e \cdot d) > 1/c \cdot (1 + e) \cdot d$$

Ähnlich sollte die Bedingung $2 \cdot d > (1 + e) \cdot d = 1 > e$ für einen Merge-Schritt gelten.
Im nächsten Abschnitt geht es um die Verwendung von MVBT für RDF Daten in der Transaktionszeit-Umgebung.

4.1.4. Transaktionszeit-Indexe für RDF in XXL

Im Übersichtskapitel über die Indexstrukturen für RDF Daten wurde argumentiert, dass es im allgemeinen Fall mindestens drei B+Bäume benötigt((S, P, O), (P, O, S), (O, S, P)) werden, um effizient die Anfrageverarbeitung zu gestalten. Aus dieser Überlegung für die Transaktionszeit-Umgebung kann die folgende Basis-Architektur vorgeschlagen werden: Ein geclusterter MVBT Baum als Speicherstruktur über (S, P, O) und zwei geclusterte B+Bäume für die Unterstützung von Anfragen auf „live“-Daten über (P, O, S), (O, S, P) (vgl. Abbildung 4.13). Das Einfüge-Algortihmus erzeugt eine neue Version von RDF Tripeln und fügt dem MVBT und beiden B+Bäumen einen Datensatz hinzu. Der Lösch-Algortihmus löscht logisch den Datensatz mit Löschversion in MVBT und physikalisch aus den „live“-Indexen.
Der Vorteil dieser Architektur liegt darin, dass historische Daten abgeschirmt sind und die Anfragen an „live“-Daten voll funktionale Vorteile der B+Bäume nutzen können. Einen kleinen Effizienz-Verlust kann es bei den Bereichsanfragen auf den MVBT „live“-Baum durch die fehlende Verkettung der Knoten auf der Blattebene geben. Die Verkettung kann aber mit zuvor beschriebenen Methoden ohne Verlust der asymptotischen Schranken implementiert werden. Die Wiederherstellung der vergangenen Version erfolgt durch MVBT und die Ausgabe kann nahtlos an die Anfrageverarbeitung gebunden werden. Ein weiterer angesprochener Vorteil ist, dass die komplexeren temporalen Anfragen von den MVBT unterstützt werden.
Ein weitere interessante Besonderheit in Zusammenhang mit RDF Daten sind die allgemeine *Bulk*-Operationen. Im weiteren Verlauf des Abschnitts wird die Möglichkeit und Notwendigkeit dieser Operationen untersucht.
Insbesondere sind die Einfüge- und Löschoperationen eines Bereichs in RDF Daten (engl. Rangedeletion bzw. Rangeinsertion) interessant. Diese Operationen gewinnen allgemein an Bedeutung bei den Indexen, die Schlüssel über zusammengesetzte Attribute verwalten. Denn durch die Ausnutzung der lexikographischen Sortierung der Attribute können ganze Bereiche z.B. mit gleichen führenden Attributen gelöscht oder eingefügt werden. Der Vorteil ist z.B. beim Range-Einfüge-Algortihmus, dass der Unterbaum mit großerem Füllungsfaktor aufgebaut (ähnlich dem Bulk-Loading) und in den B+Baum effizient eingehängt werden kann. Beim Range-Löschalgortihmus kann die Wurzel des Unetrbaums, der den Bereich aufspannt, direkt an die Freispeicherverwaltungskomponente übergeben werden.

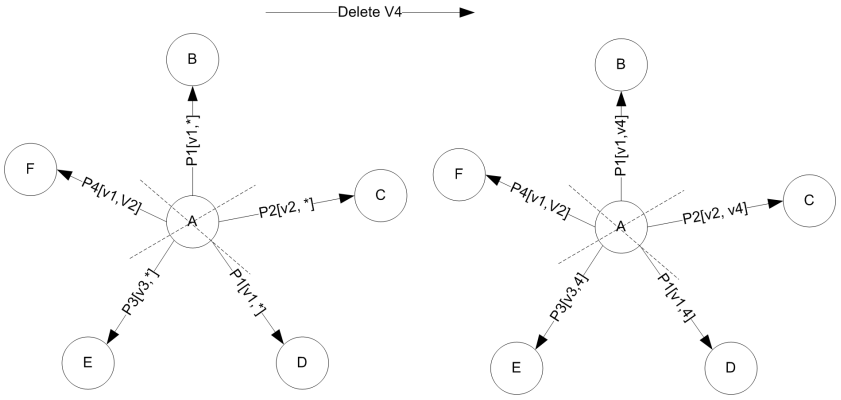

Abbildung 4.14.: Das Löschen des Bereich zum Zeitpunkt V4

Für B+Bäume existieren bereits effiziente Algorithmen für Bulk-Operationen. In der Snapshot-Umgebung können B+Bäume effizient mit *Bulk-Loading*-Algorithmen aufgestellt werden. In der Transaktionszeitumgebung lässt sich das *Bulk-Loading* nicht ohne weiteres anwenden. Anders ist es aber bei Bereichsoperationen.

Angenommen zum Zeitpunkt t wird die Ressource R samt ihren Beziehungen gelöscht. Das Löschen kann iterativ erfolgt werden. Dabei wird jeder Tripel einzeln aus dem „live“ ´-Teilbaum des MVBT mit Version t (vgl. Abbildung 4.14) gelöscht, dazu kommt noch das Löschen aus den beiden „live“-Bäumen. Dieser Prozess kann aber effizienter gestaltet werden. Das Bereichslöschen in MVBT wurde kurz in [42] in Zusammenhang mit XML-Versionierung bereits als Option besprochen. Dabei wurde die Idee vorgeschlagen, dass bei dem Löschen einer sortierten Menge von XML-Dokumenten ein effizienter Bereichlöschalgorithmus implementiert werden kann. Der Algortihmus in [42] wurde aber nur kurz skizziert, die Autoren haben auch keine Kostenuntersuchungen gemacht.
Die Idee für das Bereichslöschen basiert auf der Implemendierung des Range-Löschens in B+Baum. Im Folgenden werden Bereichseinfüge- und Löschalgorithmen für B+Baum kurz skizziert. Beide Algorithmen wurden in XXL `VariableLengthBPlusTree` implementiert.

- *Range Insertion* Sei $(k_{\min}, k_{\max})$ der Schlüsselbereich, der eingefügt werden muss. Konstruiere den Baum über den Bereich gemäß dem Bulkloading-Algorithmus, halte die Wurzel, den linken und rechten Pfad im Speicher (oder Zeiger). O.B.d.A. wird angenommen, dass die Anzahl der Daten in dem einzufügenden Bereich kleiner als die Anzahl der Daten im Ziel-B+Baum ist (Falls es nicht der Fall ist, werden die Bäume links und rechts im konstruierten Baum angehängt). Zusätzlich wird angenommen, dass kein Schlüssel aus dem Bereich schon im Zielbaum vorhanden ist. Sei h_r die Höhe des Bereichsbaums.
  Berechne den Pfad bis zum Blattknoten, in den der Bereich eingefügt werden muss. Füge die Elemente aus dem Wurzelknoten des konstruierten Baums in den Knoten, der auf der gleichen Höhe ist. An dieser Stelle kann die Anzahl der Elemente betrachtet werden. Falls die Anzahl größer als b ist, kann ein neuer Schlüssel für die Wurzel erzeugt und direkt in den Knoten, der eine Stufe höher liegt, eingefügt werden (um den späteren Split zu vermeiden). Merke diesen Knoten.
  Für alle Knoten im berechneten Pfad, die unterhalb von h_r liegen, führe folgende Schritte aus: Teile die Knoten in zwei Knoten gemäß dem Schlüsselbereich. Füge die neu erzeugten Schlüssel zu den Elternknoten. Falls der Split die Knoten mit Anzahl der Elemente kleiner als b erzeugt, verschmelze die Knoten entweder mit dem rechten oder dem linken Rand des Bereichsbaums. Erstelle dabei die richtige Verkettung. Im An-

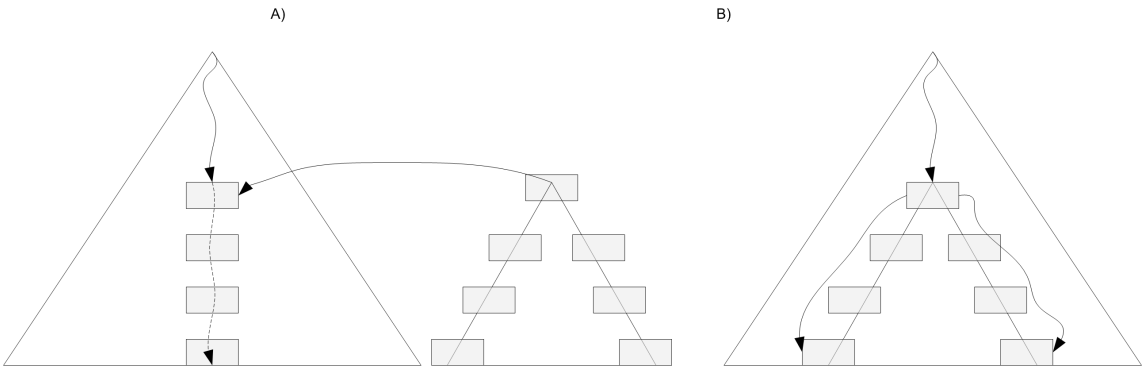

Abbildung 4.15.: A) Bereichseinfügealgortihmus B) Löschalgorithmus

schluss führe den normalen Splitalgortihmus aus, falls der Knoten auf der Höhe h_r überlauft . (vgl. Abbildung 4.15)

- *Range Deletion*: Sei $(k_{\min}, k_{\max})$ der zu löschende Schlüsselbereich. Sei D die Menge von Indexeinträgen, die Unterbäume des aufgespannten Bereichs repräsentieren. Initialisiere $D = \emptyset$. Im ersten Schritt berechne den Knoten, der die Wurzel von Unterbäumen enthält, die den Bereich aufspannen. Füge alle Indexeinträge, die innerhalb des Bereichs liegen, zu D hinzu und lösche sie aus den Knoten. Merke diesen Knoten. Ab diesem Knoten steige in den Baum links mit $k_{\min}$ und rechts mit $k_{\max}$ ab.
  Füge in den berechneten Knoten die Indexeinträge, die in dem Bereich liegen, D hinzu und lösche sie aus den Knoten. Korrigiere dabei die Verkettung. Gleichzeitig besteht die Möglichkeit die Knoten zu verschmelzen. Im Anschluss daran kann der Knoten mit gelöschter Wurzel auf die Unterlaufbedingung geprüft werden. Falls zu wenig Elemente vorhanden sind, kann entsprechendes Unterlaufbehandlung-Verfahren gestartet werden. Fall ein spezielles Modul für Freispeicherverwaltung exisitiert, übergebe die Menge D an diese Komponente. Falls jedoch nicht, dann führe für jeden Indexeintrag einen z.B. Tiefendurchlauf und lösche die Knoten in Unterbäumen (vgl. Abbildung 4.15).

Im Folgenden versuchen wir zu klären, welche Vorteile diese Verfahren bringen. Sei r Anzahl der Elementen aus dem Bereichsintervall $(k_{\min}, k_{\max})$. Falls die Elemente einzelnen aus dem Bereich gelöscht oder eingefügt werden, fallen im schlimmsten Fall etwa $O(r \cdot \log_b n)$ I/O-Kosten . Wird der Range Insertion betrachtet, und angenommen, dass $f = c \cdot B$ mit $0.5 \leq c \leq 1.0$ die Anzahl der Elemente sind, die der Bulk-Loading-Algortihmus konstant in den Knoten bei der Konstruktion einfügt. Dann werden die Kosten für das Aufstellen des Baums für r Elemente von etwa $O(\frac{r-1}{f-1})$ anfallen. Die Berechnung des Pfades, der in zwei Teilen geschnitten wird, erfordert im schlimmsten Fall den Aufwand von $O(\log_b n)$. Damit ergibt sich der Gesamtaufwand von etwa:

$$O(\log_b n + \frac{r-1}{f-1}) = O(r)$$

Für den Löschalgortihmus werden die Kosten auf ähnliche Weise aufgestellt. Die Berechnung des Pfades bis zur Wurzel des Unterbaums erfordert etwa $O(\log_b(n/r))$ Kosten, dazu kommt der Aufwand für die Berechnung des linken und rechten Randes $O(2 \cdot \log_b r)$. Falls jedoch der inneren Unterknoten von der inneren Baum von Freispeicherverwaltungkomponente verwaltet werden, dann setzten sich die Kosten nur aus der Berechnung des Pfades $O(\log_b(n/r)+2\cdot\log_b r)$. Ansonsten müssen die inneren Knoten durch den Algortihmus gelöscht werden. Dann fallen zusätzliche Kosten von etwa $O(\frac{r-1}{b-1})$ an, so dass der Gesamtaufwand zu

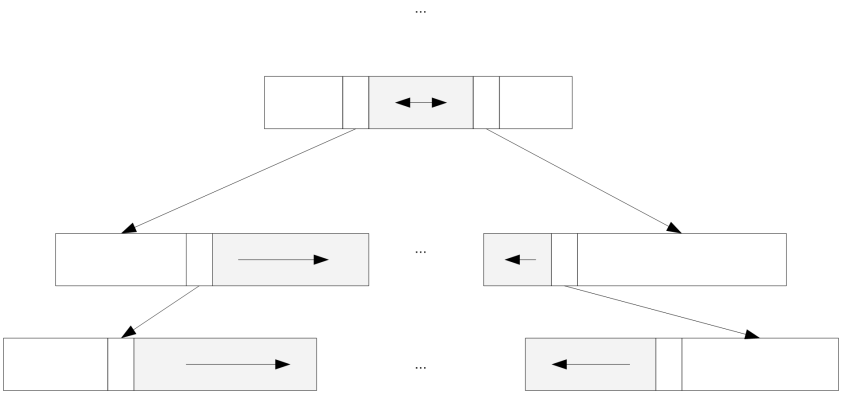

Abbildung 4.16.: Range-Deletion-Ablauf in MVBT

$O(\log_b(n) + \frac{r-1}{b-1})$ ergibt.
Auf ersten Blick scheint es, dass die Verfahren eine große Effizienzsteigerung erzielen. In der Praxis wird aber der große Aufwand von $O(r \cdot \log_b n)$ I/O-Zugriffen durch den Puffer stark gedämpft. Bei genügend größerem Puffer kann auch der Aufwand bis auf lineare Kosten gesenkt werden. Mit den Parameter b, B, f lässt sich auch die Schranke für r Elemente berechnen, ab der sich lohnt die Verfahren anzuwenden. Dabei muss auch natürlich die Puffergroße und die Pufferseitenersetzungsstrategie berücksichtigt werden. Andererseits wird der Puffer bei der Verwendung von oben beschriebenen Verfahren nicht unnötig belastet und kann für andere Zwecke verwendet werden. Im nächsten Abschnitt werden Algorithmen für Bereichsoperationen in MVBT besprochen.

Range-Operationen in MVBT

Die Grundidee des Löschalgortihmus lässt sich auch auf MVBT übertragen. In der Transaktionszeit-Umgebung werden die Knoten nur logisch gelöscht. Dabei werden nur „live“-Knoten des Baums betrachtet. Daher wird es nicht benötigt, die Knoten des inneren Unterbaums physisch zu Löschen. Ganz grob braucht der Algorithmus nur die Wurzel des Unterbäumen, die in dem Bereich liegen, zu löschen bzw. zu markieren mit Löschversion (vgl. Algorithmus 1, Abbildung 4.16).

Dabei wird der Pfad ähnlich dem B+Baum-Algorithmus bis zum Verzweigungsknoten und bis zum linken bzw. rechten Rand berechnet. Die Gesamtkomplexität würde dann nur logarithmisch sein. Etwas präzise: sei m_l Anzahl der „live“ Daten, mit d wird wie zuvor Mindestanzahl der „live“ Daten pro Knoten bezeichnet und r sie die Anzahl der Elemente in dem zu löschenden Bereich. Dann fallen folgende Kosten für Berechnung des Verzweigungsknotens im schlimmsten Fall $O(\log_d(m_l/r))$. Die Berechnung des linken bzw. rechten Randes erfordert $O(2 \cdot \log_d r)$ Kosten. Der Aufwand für die strukturelle Änderungen ist konstant pro Knoten. Somit ergibt sich der Gesamtaufwand von etwa $O(\log_d(m_l/r) + 2 \cdot \log_d r)$.
Im Folgenden klären wir, welche Auswirkungen auf Anfrageverarbeitung die direkte Umsetzung von Bereichlöschalgortihmus hat. Die Anfragen, die nach einem Schlüssel-Bereich, einem Zeitpunkt oder nach zweidimensionalen Punkten fragen, können unproblematisch ausgeführt werden. Probleme entstehen erst dann, wenn nach einer Schlüssel-Zeit-Region angefragt wird. Die Anfrageregionen, die vor dem Bereichslöschpunkt liegen, können ohne großen Problemen ausgeführt werden. Probleme bereiten die Anfragen, deren Endzeitpunkt nach dem Bereichslöschpunkt liegt. Der Grund dafür ist, dass nach dem Markieren des Wurzelsnotens die Blattknoten des Schlüsselbereichs nicht mehr über Vorgängerverlinkung erreicht werden, da auf den Blattknoten im Schlüsselbereich keine Versionssplits stattfinden

```
Input  : Schlüssel k_min; Schlüssel k_max; Version deleteVersion;
Output: Wurzel des Unterbaums, der den Schlüsselbereich aufspannt
Data: „live"-Wurzel und linker/rechter Rand:liveRoot, leftRoot, rightRoot;
Stacks für den gemeinsamen Pfad, linken/rechten Rand:commonPath, leftPath,
rightPath;
Gemeinsamer Knoten:commonSubRootNode;
begin
    berechne den gemeinsamen Pfad bis zum Knoten der die Wurzel Einträge enthält;
    commonSubRootNode ← GetNode (liveRoot);
    commonPath.Push (liveRoot, commonSubRootNode);
    while commonSubRootNode ist nicht Blattknoten do
        suche in den Knoten die mit k_min und k_max die Positionen von den
        Index-Einträgen;
        if sind die gefundene Positionen für k_min und k_max gleich then
            liveRoot ← die gefundene Position;
            commonSubRootNode ← GetNode (liveRoot);
            commonPath.Push (liveRoot, commonSubRootNode);
        else
            List ← Sublist zwischen den gefundenen Positionen; break;
        end
    end
    if Anzahl der Elemente in List ≥ 1 then
        leftRoot ← entnehme ersten Element;
        if Anzahl der Elemente in List ≥ 1 then
            rightRoot ← entnehme letzten Element;
            if Anzahl der Elemente in List > 0 then
                foreach Element e in List do MarkAsDeleted (e, deleteVersion );
            end
        end
    end
    berechne den Pfad für den linken und rechten Rand und markiere Elemente die
    größer gleich als k_min in linken Rand und kleiner gleich als k_max in den rechten
    Rand als gelöscht;
    if leftRoot ungleich null then
        ComputePathAndMarkDeleted (leftPath, leftRoot, k_min, deleteVersion, left =
        true);
        leftNewEntries ← HandleVersionOverUnderflow (leftPath, deleteVersion);
        füge die Elemente aus den Listen den gemeinsamen Knoten
        commonSubRootNode hinzu;
    end
    if rightRoot ungleich null then
        ComputePathAndMarkDeleted (rightPath, rightRoot, k_max, deleteVersion, left =
        false);
        gehe den berechneten Pfad nach oben und prüfe auf Unterlauf Bedingungen;
        rightNewEntries ← HandleVersionOverUnderflow (rightPath, deleteVersion);
        füge die Elemente aus den Listen den gemeinsamen Knoten
        commonSubRootNode hinzu;
    end
    prüfe Unterlauf Begingung für den Gemeinsamen Pfad ggf. Korrigiere;
     HandleVersionOverUnderflow (commonPath, deleteVersion);
    return commonSubRootNode;
end
```

Algorithm 1: Range-Deletion-Algorithmus in MVBT

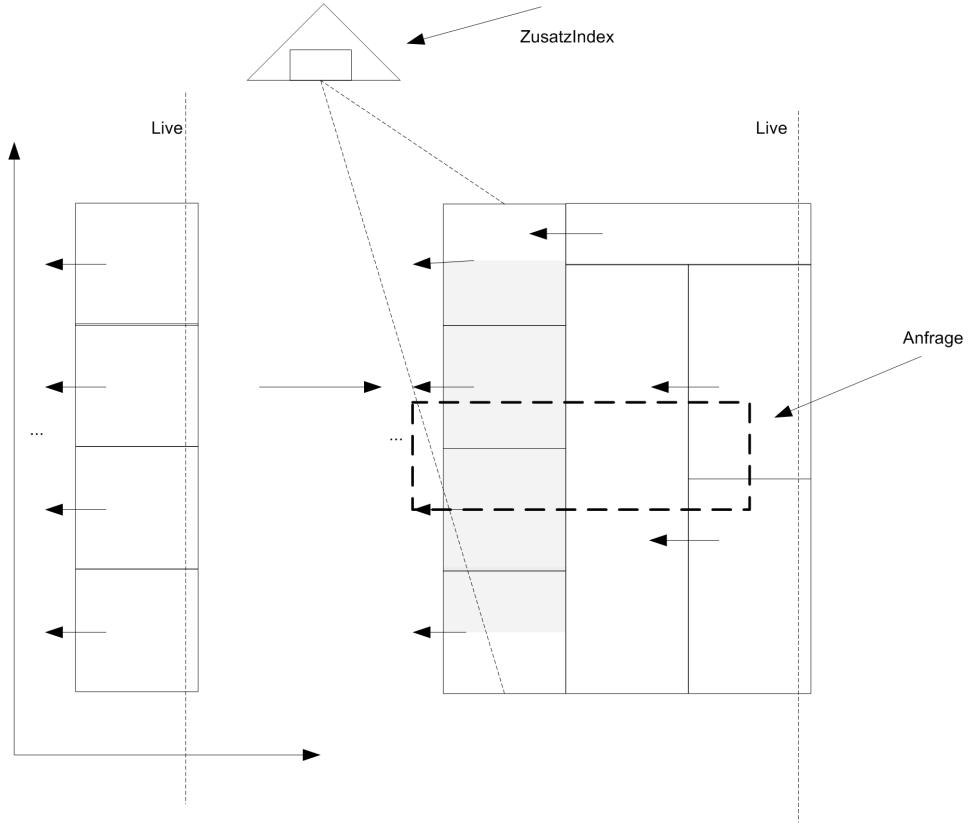

Abbildung 4.17.: Die Abbildung zeigt das Problem bei einer Schlüssel-Zeit-Anfrage, falls ein Bereich in einer vergangenen Version gelöscht wurde. Durch die fehlende Verlinkung in der Zeizachse, kommt der Anfragealgorithmus nicht in den gelöschten Bereich rein. Die Abhilfe schafft der Zusatzindex, der die Einträge für die gelöschten Wurzelknoten verwaltet.

(vgl. Abbildung 4.17). Auf ersten Blick muss teure Traversierung aller historischen Wurzelknoten, die in Anfrageregion liegen, durchgeführt werden. Die Ausgabe des Algortihmus 1 ist aber der IndexEintrag des Knoten, der die Wurzel der Unterbäumen , die den gelöschten Schlüssel Bereich aufspannen, enthält. Um die Effizienz der Schlüssel-Zeit Anfragen nicht zu verschlechtern, kann der gesamt MVBT um einen speziellen Index erweitert werden, der diese Wurzeleinträge Verwaltet. Diese Indexeinträge können dann nach Löschzeitpunkt und Schlüssel sortiert werden. Diese repräsentieren historische MVBT Unterbäume für einen bestimmten Schlüsselbereich.
Die Indexeinträge können dann wie folgt gestaltet werden *(Zeiger, Löschversion, Schlüssel Bereich)*. Der QueryCursor kann dann um einer Suche im Zusatzindex mit der Anfrageregion erweitert werden. Nachdem der Eintrag lokalisiert wurde, wird genau wie beim normalen Algorithmus fortgefahren. Im Allgemeinen kann die Schlüssel-Zeit-Anfrage wie folgt ausgeführt werden:

- Im ersten Schritt wird der normale MVBT Anfragealgorithmus gestartet. Dabei wird die bestehende Vorgängerverkettung ausgenutzt.
- Im zweiten Schritt wird der Zusatzindex nach der Überlappendeneinträgen abgefragt. Der native MVBT Algorithmus kann für die Wurzel des gelöschten Bereichs angewendet werden. Dabei werden die Blattregionen abgearbeitet, die komplett in dem gelöschten

Schlüsselbereich liegen. Da die Knoten, die Elemente enthalten, die zum Zeitpunkt des Löschens des Bereichs „live“ waren, und nur ein Teil von Elementen im Schlüsselbereich lag, über Vorgängerverkettung erreicht werden können, wurden schon im ersten Schritt abgearbeitet.

Pro gelöschten Bereich wird genau ein Zusatzeintrag in den Zusatzindex eingefügt. Der zusätzliche Index für gelöschte Bereiche kann als B+Baum aufgebaut werden. Damit kann die Suche im Index als B+Baum-Bereichssuche implementiert werden. Im Folgenden untersuchen wir die Komplexität der Anfrageverarbeitung in MVBT mit Bereichslösch-Funktionalität.
Sei l die Menge der Indexeinträgen von den gelöschten Bereichen, die den Anfragebereich überlappen. Sei $|l|$ Gesamtanzahl der gelöschten Bereiche. Sei r_{l_i} mit $0 \leq i \leq |l|$ die Anzahl der Elemente zum Zeitpunkt des Löschens im gelöschten Bereich. Sei r die Anzahl der Elemente, die im Anfragebereich liegen. Das Durchsuchen des Zusatzindexes erfordert im schlimmsten Fall $O(\log_b |l| + |l|/b)$ I/O-Kosten. Der Abstieg mit dem rechten Rand des Anfrageregions erfordert $O(\frac{r_k-1}{d-1})$ Kosten(mit r_k wird die Anzahl der Schlüssel, die Anfrageregion zum Zeitpunkt der rechten Grenze schneiden, bezeichnet). Die Abstiegskosten für die Menge l sind etwa $O(\sum_{i=0}^{|l|}(\frac{r_{l_i}-1}{d-1} + \log_d(m_i/r_{l_i})))$. Damit ergeben sich die Gesamtkosten von etwa:

$$O(\log_b j + \log_b |l| + |l|/b + r/d + \log_d(m_{max}/r_k) + \frac{r_k - 1}{d-1} + \sum_{i=0}^{|l|}(\frac{r_{l_i} - 1}{d-1} + \log_d(m_i/r_{l_i})))$$

Der Aufwand kann auch verbessert werden, falls die Doppeltseitigeverkettung der Blattknotenseiten implementiert wird, da nach dem Löschen eines zusammenhängenden Bereichs die Verkettung unberührt bleibt. Dann muss nur der Rand beim Abstieg in den gelöschten Bereichsunterbaum berechnet werden. Damit kann der Aufwand auf

$$O(\log_b j + \log_b |l| + |l|/b + r/d + \frac{r_k - 1}{d-1} + \sum_{i=0}^{|l|}\log_d r_{l_i})$$

reduziert werden. Die zusätzlichen Speicherkosten sind linear in der Anzhal der gelöschten Bereiche.
Bei dieser Architektur kann der QueryCursor bei der Ausgabe der „live“-Elementen aus den gelöschten Bereichen den tatsächlichen Löschzeitpunkt setzten, da beim Löschalgorithmus die Daten aus den Unterbäumen unverändert bleiben bzw die Intervalle der „live“-Daten zum Zeitpunkt des Löschens nicht angepasst wurden. Dies kann dann über Anfrageverarbeitung geschehen. Auf Zusatz-B+Baum kann auch verzichtet werden, indem die Bereichseinträge in dem Wurzelbaum mitverwaltet werden. Außerdem lässt sich unter anderem auch der ganze „live“ Wurzel mit diesem verfahren löschen. Im Endeffekt wird der Baum abgeschlossen und es kann ab diesem Punkt neuer Baum gestartet werden.
Ähnlich dem B+Baum-Range-Einfüge-Algorithmus kann auch Verfahren für MVBT implementiert werden. Dabei kann ähnlich zum Bulk-Loading von B+Baum die Knoten nur an Hand des Schlüssels verteilt werden, da die Daten zu einer Version gehören. Der Wurzel Knoten kann wiederum in „live“ Unterbaum des MVBT an der richtigen Position angehängt werden. Die Knoten auf dem Pfad bis zur „live“ Blattebene müssen dann in zwei Teile aufgeteilt werden, entsprechend müssen dann die MVBT Bedingungen an diesen Knoten überprüft, und unter Umständen strukturelle Änderungen vorgenommen werden. Anders als beim Löschen wird es nicht benötigt, die Indexeinträge der Wurzelknoten zusätzlich zu verwalten. Denn der Anfragealgortihmus würde dann bis zum zeitlichen Anfangspunkt von dem eingefügten Bereich gehen, da sonst keine zeitliche Verlinkung existiert. Falls jedoch die Daten aus dem Bereich zu dem früheren Versionen existieren (und später gelöscht wurden), werden die über

die Vorgänger-Verlinkung über den anderen Knoten erreicht.
Nachdem die Verfahren für Range-Operationen erläutert wurden. Bei der Indexarchitektur für RDF Daten in Transkationszeitumgebung sind noch mindestens zwei weitere „live“-Indexe vorhanden (vgl. Abbildung 4.13). In diesen muss auch der zusammenhängende Bereich an RDF Tripel gelöscht oder eingefügt werden. Leider lässt sich an diesen Bäumen keine Bereichsoperation anwenden, da die Attribute andere Reihenfolge besitzen (im Allgemeinen können wir auf einen der drei Indexe Bereichsoperationen anwenden, in Abhängigkeit der Sortierreihenfolge der Attribute). In diesen wird dann das Löschen oder Einfügen iterativ vorgenommen.
Zusammengefasst für effiziente Verwaltung von RDF Daten in Transaktionszeitsemanitk, werden mindestens zwei geclusterte B+Bäume für die Indexierung von „live“ Daten über (P, O, S) und (O, S, P) benötigt, der historischen Index wird als MVBT implementiert und verwendet als Schlüssel (S, P, O). Für die effiziente Verarbeitung von Anfragen an „live“-Daten muss MVBT in dem Blattebene neben den Vorgänger- auch die Schlüsselverlinkung implementieren. Die Verkettung wird am effizientesten als doppelseitige Verlinkung implementiert. Optional können die Range-Operationen von den Indexen angeboten werden.
Im nächsten Abschnitt wird kurz über die Möglichkeit der Entwicklung von Validzeitindexe für RDF Daten in XXL beschrieben.

4.1.5. Validzeit-Indexe für RDF in XXL

Für Validzeit RDF Daten gibt es in XXL verschiedene Möglichkeiten die bestehende Indexe anzupassen und zu verwenden. Die bestehenden R-Bäume können direkt für eindimensionale Intervalle und das zweidimensionale Mapping verwendet werden. Außerdem besteht die Option den B+Baum mit raumfüllenden Kurven zu benutzen. Eine Implementierung für *Z-Kurve* gibt es schon bereit, die dann den Einsatz von *UB-Baum* (B+Baum, mit Z-Kurve) möglich macht. Für diese Zwecke wurde spezielle QueryCursor entwickelt, der mehrdimensionale Fensteranfragen in B+Baum mit Z-Kurve durchführt. Dazu bietet XXL Bibliothek unter anderem auch die Methoden und Klassen für die Verwaltung von temporalen Objekten. Auch temporale Operatoren sind Bestandteil der Biblitohek.
Zusätzlich wurde bei der Arbeit der im Einführungskapitel über temporale Indexstrukuren angesprochene IR-Tree implementiert. Dabei wurde die Mapping-Funktion als statische Methode geschrieben, die den Knotenwert eines einzufügenden Intervalls in einem virtuellen Intervall-Baum berechnet. Dadurch, dass der virtuelle Baum über die Potenzen von Zwei aufgebaut ist, lässt sich die Funktion mit effizienten Bitoperationen implementieren. Entsprechend wurden auch zwei B+Bäume angelegt. Die Anfrageberechnung wurde dann mit Hilfe der `cursors` Paket entwickelt. Die Anfrageverarbeitung stellt dann die Folge von Breichsnanfrage-Cursors an jeweiligen B+Bäumen dar. Die Folge wird der Reihe nach verarbeitet. Im Anschluss werden die Ergebnisse von einzelnen Bereichsnafragen mit Hilfe des Merge-Cursors zusammengemischt.
Alternativ kann auch ein einziger B+Baum verwendet werden. Die Idee ist dabei den Eintrag *(Knotenwert, Anfangspunkt, Intervall)* bzw. *(Knotenwert, Endpunkt, Intervall)* durch die Ausnutzung der Schlüssel mit zusammengesetzten Attributen mit lexikographischer Sortierung auf einen Eintrag mit folgenden Struktur abzubilden *((Knotenwert, S , Anfangs/Endpunkt, Intervall))*. Wobei S ist ein boolsche oder numerische Wert, der Angibt, ob es um den Anfangs-oder Endpunkt handelt. Für Repräsentation von S kann ein einziges Bit verwendet werden (in Java wurde bei der Implementierung ein ganze Byte benutzt). Alternativ kann der Wert S als führende Attribut vorangestellt werden. Dann hätten wir eine ähnliche Architektur wie Partitions-BTrees. Im nächsten Abschnitt werden die Experimente mit den IR-Tree und R-Bäumen durchgeführt.

4.2. Experimente und Ergebnisse

Für die Durchführung von Experimenten wurden Klassen für RDF Knoten und Tripel für interne Darstellung von RDF Daten entwickelt. Die RDF Daten werden intern auf Java.String-Objekte abgebildet. Die Längen der Zeichenketten wurden auf 255 Zeichen limitiert. Zusätzlich wurden die entsprechenden Funktionen auf Abbildung auf drei komponentigen Zeichenkettenarray für die Indexierung bereitgestellt. Für die Tests wurden zwei reale Datenmengen verwendet (*Burton*[2] und *GovTrack*[3]). Die erste Datenmenge enthält etwa 80 Millionen RDF Tripel und wurde sowohl für die B+Baum als auch für die temporale Strukturen verwendet. Die zweite Datenmenge enthält auch temporale Daten über die legislativ Perioden von den Mitglieder des US-Senats.
Für das Parsern von RDF XML-Dateien wurde der Parser aus *Jena RDF Framework*[4] benutzt.
Aus der Burton-Datenmenge wurden alle Tripel mit der Zeichenketten, die länger als 255 Zeichen sind, entfernt. Somit wurde die Testmenge auf 78 Millionen Tripel reduziert. Im nächsten Schritt wurde mit Hilfe der entwickelten Klasse `VaribleLengthBPlusTree` der geclusterte Index auf (S, P, O) aufgebaut, um die Duplikate zu eliminieren.
Folgende Parameter wurden für den B+Baum benutzt: Schlüssel über S, P, O, Mindestfüllgrad $B * 0.33$ mit kleinstem Schlüssel-Splitstrategie, 16KB Seiten. Zusätzlich wurde ein Puffer mit 120 Seiten bereitgestellt. Die ganze Prozedur hat etwa 80 Stunden gedauert. Die Datenmenge wurde auf 51472909 RDF Tripel reduziert. Die Gesamtgröße des Indexes betrug etwa 14 GB. Danach wurden die Elemente aus der Blattebene für das *Bulk-Loading*-Algorithmus verwendet (da die Daten in der Blattebene sortiert vorliegen). Der Aufbau hat 88 Minuten für 51472909 RDF Tripel gedauert. Die Seiten wurde nah am 100% der Seiteninhalt gefüllt. Die Indexgröße hat sich gegenüber Tupel-bei-Tupel Methode auf 6 GB reduziert. Das Ergebnis entspricht den eingestellten Parameter für B+Baum .

4.2.1. Validzeit-Strukturen

Der Entwickelte IR-Tree wurde experimentell mit R-Bäumen verglichen. Dafür wurde ein R-Baum für eindimensionale Intervalle angelegt und gemäß der vorgestellten Mapping-Technik ein R-Baum für zweidimensionale Punkte erzeugt. Für das Experiment wurde eine Menge von 100000 Intervallen, deren Anfangspunkte in dem Bereich zwischen 1 bis 100000 liegen, erzeugt. 80% der Intervallen wurden mit der Länge, die gleichverteilt im Bereich 1 bis 100 ist, angelegt. Der Rest wurde mit der Länge zwischen 101 bis 5000 erstellt. Es wurden dann 10 Anfragetypen gestellt. Die Anfragen liefen als gleitendes Fenster über den ganzen Raum der Intervalle mit den längen (5, 10, 50 , 100, 500, 1000, 2000, 5000, 10000, 20000). Die Seitengröße wurde auf 2KB eingestellt. Die verwendete Variante des R-Baums war der R -Baum, mit füllungsfaktor von etwa 0.33. Im ersten Lauf wurden die Strukutren ohne den Einsatz vom Puffer verwendet. Im zweiten Lauf jedem Baum wurde ein Puffer von 12 Seiten zugewiesen (vgl. Abbildung 4.18) .
Ohne den Puffer hat der IR-Tree im Schnitt etwa drei mal soviel Seitenzugriffe im Vergleich zum eindimensionalen R-Baum gemacht. Durch einen großen Suchraum bei der Mapping-Technik hat der zweidimensionale R-Baum doppelt soviel I/O verursacht als eindimensionale R-Baum. Interessant war zu beobachten, dass durch das Backtracking von R-Bäumen der Puffer-Ausnutzung nah beim Null war. Dagegen wurde die Anzahl der I/O-Zugriffe mit dem

[2] http://simile.mit.edu/rdf-test-data/barton/
[3] http://www.govtrack.us/source.xpd
[4] http://jena.sourceforge.net/index.html

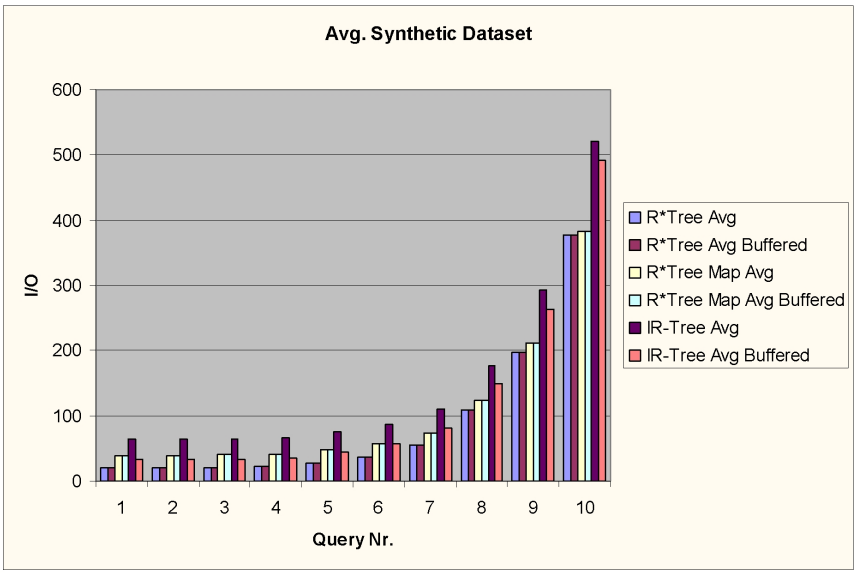

Abbildung 4.18.: Ergebnis des Tests von Validzeitindexe R-Baum, R-Baum zweidimensional, IR-Tree

Einsatz von Puffer an beiden B+Bäumen fast halbiert. Das wird erklärt durch sequentielle Verarbeitung einer Menge von Bereichsanfragen für die Mengen von „Aufspießwerten“ des virtuellen Intervall-Tree L, R und die Bereichsanfrage für Blattkonten $Inner$, die auf die B+Bäume gestellt werden. Dabei werden die Indexknoten in den höheren Ebenen der B+Bäume in den Puffer gehalten, so dass die nächste Bereichsanfrage auf den Teil der Knoten zugreifen kann. Insbesondere hilft der Puffereinsatz bei relativ kleinren Intervallanfragen. Zuvor wurden auch echte Intervalldaten aus dem GovTrack-Datenmenge herausgezogen, leider hat sich heraus gestellt, dass die Daten (legislativ Perioden von Mitarbeiter der US Senats) nach der Eliminierung der Duplikaten relativ kleinere Anzahl von Intervallen hatten (etwa 41000 Intervalle). Gleichzeitig hatten die Intervalllängen fast die gleiche Größe. Bei den Tests mit GovTrack-Datenmenge hatten wiederum die eindimensionale R-Bäume am besten abgeschnitten.
Im Großen und Ganzen im Systemen, bei den keine integriereten R-Bäume existieren, kann ohne großen Aufwand ein IR-Tree implementiert werden, da die B+Bäume sind feste Bestandteile der modernen RDBMS. Im Schnitt zeigen sie mit dem Puffereinsatz um Faktor zwei schlechtere Performanz. Eine andere Möglichkeit ist der Einsatz von den Raumfüllenden Kurven wie Z- und Hilbertkurve im Zusammenhang mit mehrdimensionalen Mapping im Systemen ohne implementierten R-Bäumen.

4.2.2. Transaktionszeit-Strukturen

Der MVBT für RDF Daten wurde getestet mit realen Daten aus dem Burton-Datenmenge. Dabei wurde die Speicherplatzausnutzung untersucht in Hinsicht auf Einfüge- und Löschoperationen. Für Testzwecken wurde ein zusammenhängender Bereich von RDF Burton-Daten ausgeschnitten. Die Testmenge wurde auf 500000 verschiedenen Tripel beschränkt. Die Gesamtspeichergröße der Testmenge beträgt 48.435 MB. Aus der Ursprungsmenge wurden fünf weitere Testmengen für Simulation von Transaktionszeitumgebung mit jeweils 10%, 20%, 30%, 40%, 50% der gelöschten Daten erstellt.
Dafür wurden mit jedem RDF Tripel ein Intervall assoziiert. Die Anfangspunkte der Intervalle wurden aus der Permutation von 1 bis 500000 entnommen. Der entsprechende prozentuale

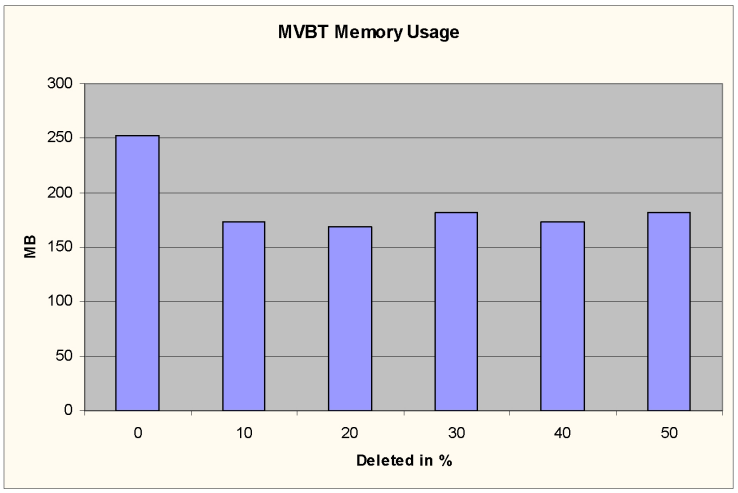

Abbildung 4.19.: MVBT Simulation mit 500000 RDF Tripel aus dem Burton Dataset

Anteil wurde mit dem Endpunkt des Intervalls gelöscht. Die Länge des Intervalls wurde mit Gleichverteilung aus dem Bereich von 10 bis 1000 erzeugt. Die erzeugten Mengen wurden nach dem Anfangspunkt bzw. dem Endpunkt der Intervallen sortiert. Die Daten wurden gemäß der Sortierung in MVBT eingefügt, um die monotone Zusammenspiel von Einfüge und Löschoperationen zu simulieren. Folgende Parameter wurden verwendet: die Blockgröße war auf 16 KB eingestellt, der Mindestfüllfaktor mit 0.25 vom verfügbaren Seiteninhalt initialisiert (Parameter d aus dem vorigen Kapitel), der Parameter e wurde mit 0.5 eingestellt (damit lagen die Grenzen für Versionbedingungen im Bereich zwischen 0.375 und 0.875 von dem Seiteninhalt). Die Schlüsselsplits wurden gemäß den Kleinsten-Schlüssel-Splitstrategie durchgeführt. Beim verwalten von ausschließlich „live“-Daten ist die Platzverbrauch des MVBT um den Faktor 5 größer als die Große der Ursprungsdatenmenge (vgl. Abbildung 4.19)(entspricht auch den Experimenten aus [31]).
Zusammengefasst lassen sich in XXL Bibliothek die Validzeit- und Transaktionszeit-Indexe entwickeln. Die eindimensionale R-Bäumen zeigen im Schnitt sehr gute Performance. Der MVBT und B+Baum mit Erweiterung auf variabel langen Schlüssel können zur Entwicklung eines Backend-Indexes für eine Versionsverwaltung und eine Archivierung von RDF Daten herangezogen werden. Offene Punkte sind noch die effiziente Kompression von Indexeinträgen in den Knoten der Indexstrukturen, Entwicklung eines Anfrageverarbeitungsmoduls mit einem Anfrageoptimierer und Übersetzung der SPARQL-Anfragen. Im Weiteren könnten die Tests zwischen den IR-Tree und B+Bäumen mit den raumfüllenden Kurven durchgeführt werden, da der Einsatz der beiden Techniken für Systeme ohne R-Bäumen von Interesse ist.

5. Schlusswort und Ausblick

Der einfache Aufbau des RDF Frameworks und seine semantischen Besonderheiten machen das Framework sehr attraktiv nicht nur für das Semantik Web, sondern für viele andere Anwendungen aus verschiedenen Bereichen von Datenverwaltung. Bei Applikationen mit einem stark variierenden Datenmodell lässt sich die Datenverwaltung mit RDF flexibler gestalten. Im manchen Bereichen stellt RDF eine gute Alternative für das relationale Modell, insbesondere in heterogenen Umgebungen. Zum Beispiel kann RDF in der Service orientierten Architektur (SOA) zur Beschreibung von Service-Anbietern und Service-Profilen eingesetzt werden. Dabei können die Service-Agenten mit semantischer Analyse selbst nach „passenden" Services suchen. RDF findet eine interessante Anwendung bei der Stammdatenverwaltung (eng. Master Data Management (MDM))[1] und Integration von Daten aus verschiedenen Quellen (unter anderem relationalen Datenbanken).
Der Schwerpunkt der Arbeit lag auf der Verwaltung von großen Datenmengen von RDF Daten. Es wurde der Frage nachgegangen, wie temporale RDF Daten effizient verwaltet werden können. Die Verwaltung von RDF Daten mit Hilfe von relationalen Datenbanksystemen in einer Tripletable hat sich am effizientesten herausgestellt. Dafür sprechen viele Gründe sowohl von allgemeiner Betrachtung von Datenverwaltung (die Verwaltung von großen Datenmengen, Datenmanipulation mit SQL, Nebenläufigkeit- und Wiederherstellungs- Komponenten) als auch aus der RDF Sicht. Funktionale Besonderheiten von RDF lassen sich auch mit Hilfe von relationalem Modell effizient abbilden. Die Graphanfragen werden anhand von relationalen Operatoren ausgeführt. Zudem wurde SPARQL „Lingua Franca"[9] für RDF Datenbanken und kann vollständig mit SQL ausgedrückt werden. Im Laufe der Entwicklung von Verwaltungsmöglichkeiten wurden auch verschiedene Tabellenstrukturen auf ihre Effizienz untersucht. Als Fazit aus ersten drei Kapiteln hat sich die Tripletable mit dem Einsatz von B+Bäumen sowohl in „normalen" als auch in spaltenorientierten RDBMS als beste Lösung herausgestellt.
Für die Indexierung von RDF Daten in nicht-temporalem Kontext sind am besten geclusterte B+Bäume geeignet. Es gibt aber auch offene Punkte im Bereich der konventionellen Indexstrukturen und Anfrageverarbeitung von RDF Daten, die aktuell von Forschern untersucht werden. Besondere Schwierigkeiten bereiten die Join-Anfragen bei der Berechnung von einfachen Pfadausdrücken. Dabei werden zurzeit Verfahren für Plangenerierung von Anfragen, mit minimalen Berechnungskosten in Zusammenhang mit RDF, untersucht. In [23] ist der aktuelle Stand zu diesem Thema zu finden. Ein anderer offener Punkt, der wenig in der Arbeit besprochen wurde, ist die Entwicklung und Anpassung von Indexstrukturen und Operatoren für allgemeine Pfadanfragen. Da dem SPARQL die Rekursion fehlt, müssen diese Anfragetypen gesondert behandelt werden. In Zusammenhang mit der Indexierung von RDF Daten wurden Themen wie Verwaltung von zusammengesetzten Attributen, Verwaltung von Zeichenketten, partielle Anfragen auf Indexstrukturen, effiziente Datenkompriemierung in Indexstrukturen neu ins Leben gerufen. Auch die Frage wie die Graphpartitionierung spielt in RDF eine große Rolle.
Im Laufe der Arbeit waren immer wieder die Querverbindungen zum relationalen Modell hergestellt. Auf den formalen Grundlagen der temporalen Datenbanken, die im Zuge der Entwicklung von relationalem Modell erforscht wurden, wurde auch Basis-Modell für tempo-

[1] http://domino.watson.ibm.com/comm/research.nsf/pages/r.web.innovation.semantic.html

rales RDF entwickelt. Die Ausarbeitung zu diesem Thema ist in Forschung sehr aktuell. Dafür spricht auch, dass die Unterstützung von temporalen relationalen Daten noch nicht von allen kommerziellen Datenbanken implementiert wurde. In Kapitel zwei wurde eine Forschungsarbeit besprochen, bei der das Timestamp-Modell als grundlegendes Modell für RDF untersucht und verwendet wurde. Nach diesem Modell lassen sich RDF Tripel um Zeitattribute erweitern. Damit wurde auch die theoretische Basis für den Einsatz von temporalen Strukturen aus der relationalen temporalen Welt geschaffen. Im Kapitel drei wurden verschiedene Indexstrukturen und allgemeine Techniken für temporale relationale (RDF) Daten präsentiert. Dabei wurden sie in Hinsicht auf Zeitsemantik (Transaktionszeit, Validzeit) und temporale Anfrageverarbeitung klassifiziert. Intuitiv ist die Validzeit ein fester Bestandteil der RDF Daten und ist explizit verfügbar. Der Großteil des Kapitels wurde aber auch den Transaktionszeit-Indexstrukturen gewidmet. Dabei wurde versucht die Frage nach ihren Einsatz-Möglichkeiten zu untersuchen. Der Bedarf nach komplexen temporalen Anfragen in Zusammenhang mit der Transaktionszeit-Semantik wurde in der Ontologie- und RDF/S-Verwaltung besprochen. In der Zukunft könnte der Einsatz von Strukturen wie MVBT, MVAS und TSB in diesem Bereich untersucht werden. In demselben Kapitel wurde kurz angesprochen, dass auch bei der XML Versionierung für die Unterstützung von komplexen Anfragen diese Art von Indexen eingesetzt wird.
In Zusammenhang mit RDF Versionierung wurden die Themen wie Bulk-und Range-Operationen auf Indexen zu einer interessanten Fragestellung. Die Motivation dafür waren der Einsatz von RDF für Datenintegration aus verschiedenen Quellen und das Graphmodell. Im vierten Kapitel wurde die Fragestellung tiefer betrachtet. Es wurden unter anderem die Algorithmen für Range-Operationen für MVBT und ihre Auswirkung auf Anfrageverarbeitung untersucht. Dabei wurde festgestellt, dass diese Operationen in MVBT ohne großen Einfluss auf die Anfragealgorithmen implementiert werden können. Es gibt auch offene Punkte im Bereich MVBT für RDF Daten wie z.B. allgemeine Bulkloading-Operationen, allgemeine Datenkompriemierung ähnlich wie in der neuen Variante von TSB. Auch die Entwicklung von effizienten Verfahren für Recovery und Nebenläufigkeit sind immer noch aktuell[32]. In der Zukunft könnte ein Versionierungssystem für Ontologie-Verwaltung auf Basis der XXL Bibliothek implementiert werden. Einer der besonderen Vorteile wäre ein an Anfrageoperatoren und Indexstrukturen reiches Paket, das eine komplexere temporale und konventionelle Anfrageverarbeitung ermöglicht.
Eine andere Richtung der Forschung in RDF, die kurz in Kapitel zwei in Zusammenhang mit temporalem RDF angesprochen, ist die allgemeine Erweiterung von RDF um andere Dimensionen. In [43] wird z.B. versucht sowohl temporale als auch Geo-Aspekte von RDF Daten zu traversieren und semantisch zu analysieren.
Zusammengefasst sind die folgenden Forschungsrichtungen in RDF sehr aktuell:

- Temporale Anfragesprachen für RDF
- Anfrageoptimierung von SPARQL Anfragen in relationalen Datenbanken
- Entwicklung von speziellen Operatoren und Indexstrukturen für allgemeine Pfadausdrücke
- Erweiterung von Geo-RDF Daten um Geo-Informationsverarbeitung (Indexstrukturen, Anfrageverarbeitung, Zusammenspiel mit semantischer Analyse)
- Verlagerung der semantischen Analyse auf relationale Datenbanken, z.B. Berechnung der Transitivität von Beziehungen
- Verbesserung der Effizienz von Indexstrukturen

A. Anhang

Im Anhang werden exemplarisch die UML Diagramme für die wichtigsten Klassen dargestellt. Diese wurden in Kapitel über die Indexstrukturen in Java Bibliothek XXL angesprochen.

UML

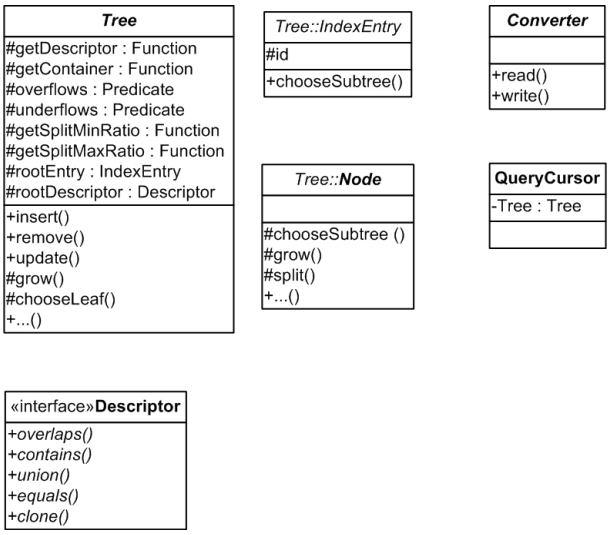

Abbildung A.1.: Exemplarisch Klasse Tree und die wichtigsten Klassen und Interface für Entwicklung von baumbasierten Indexstrukturen. Descriptor als Wegweiser Objekt. Abtsrakte Klasse Converter für Abbildung von Objekten und Knoten auf externen Speicher. QueryCursor stellt Basis Funktionalität für Entwicklung von Anfrageverarbeitung.

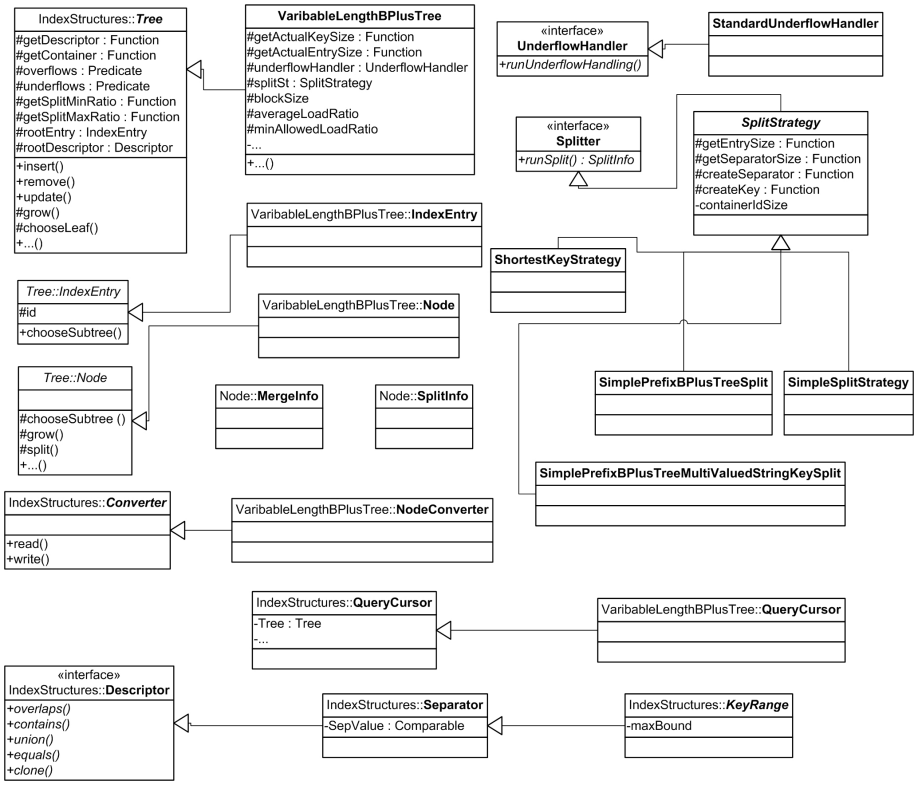

Abbildung A.2.: Exemplarisch die UML Struktur von VariableLengthBPlusTree mit Splitstrategie. IndexEntry erweitert den IndexEntry von Klasse Tree um die Verwaltung von Objekten Separator. Separator repräsentieren die Schlüssel von B+Baum. NodeConvereter zur Abbildung von Knoten von dem B+Baum auf externen Speicher, enthält auch die Seiten Layout Logik. QueryCursor verarbeitet Punkt- und Bereichanfragen. KeyRange ist Abstrakte KLasse und erweitert den Separator, die Klasse repräsentiert den Schlüssel Bereich. Merge und SplitInfo Objekte verwalten den Zwischenstatus Information nach der Reorganisationsoperationen.

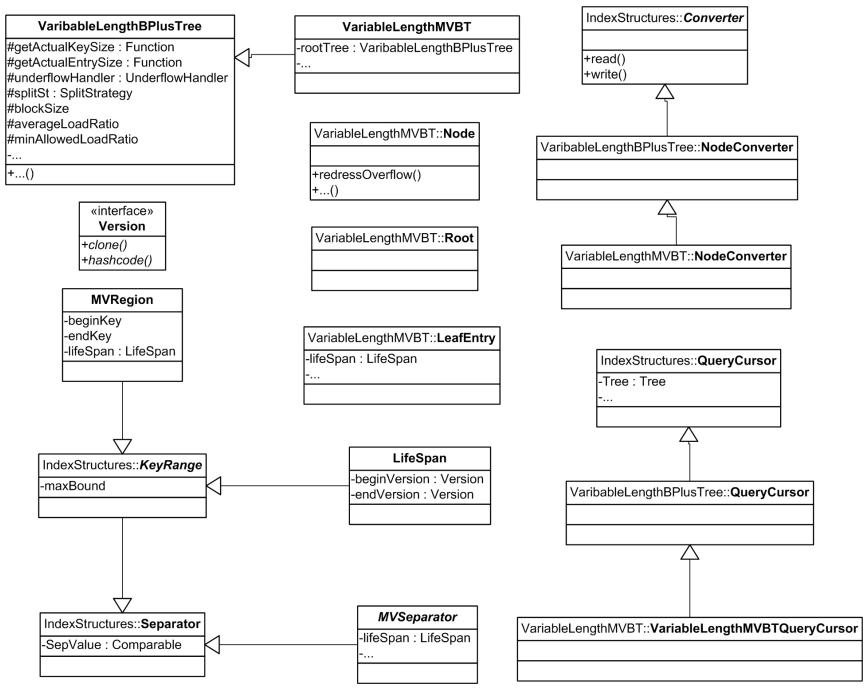

Abbildung A.3.: Exemplarisch UML Struktur von VariableLengthMVBT. Interface Version repräseniert den Zeitstempel, MVSeparator erweitert den Separator von B+Tree um den Zeitintervall der als Objekt der KLasse LifeSpan implementiert ist.

Literaturverzeichnis

[1] Graham Klein, Jeremy J. Carrol: „Resource Description Framework (RDF):Concepts and Abstract Syntax“ „http://www.w3.org/TR/rdf-concepts/“ W3C Recommendation 10 February 2004

[2] P.Hayes:„RDF Semantics“ W3C Recommendation, Feb 2004 „http://www.w3.org/TR/rdf-mt/“

[3] Shelly Powers: „Practical RDF“ O'Reilly, 2003

[4] Thomas B. Passin: „Explorer's Guide to The Semantik Web“ Manning, 2004

[5] Claudio Gutierrez, Carlos Hurtado, Alberto O. Mendelzon: „Foundations of Semantic Web Databases“ PODS 2004

[6] Peter Haase, Jeen Broekstra, Andreas Eberhart, Raphael Volz: „A Comparison of RDF Query Languages“, www.aifb.uni-karlsruhe.de/WBS/pha/rdf-query/

[7] Richard Cyganiak: „A relational algebra for SPARQL“ HP Labs, Bristol, UK

[8] Dave Beckett: „SPARQL RDF Query Language Reference v1.8“ http://www.dajobe.org/2005/04-sparql/

[9] Klara Weiand and François Bry and Tim Furche: „Reasoning and Querying – State of the Art.“ research report, PMS-FB-2008-11, Institute for Informatics, University of Munich, 2008

[10] Kevin Wilkinson, Craig Sayers, Harumi Kuno, Dave Reynolds: „ Efficient RDF Storage and Retrieval in Jena2“ HPL-2003-266 , Technical Report, HP Laboratories, 2003

[11] Eugene Inseok Chong, Souripriya Das, George Eadon, Jagannathan Srinivasan: „An Efficient SQL-Based RDF Querying Scheme“ VLDB Conference, Trondheim, Norway, 2005

[12] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, S. Manegold: „Column-Store Support for RDF Data Management not all swans are white“ Auckland, New Zealand, VLDB 2008

[13] D.J. Abadi, A. Marcus, S.R. Madden, K. Hollenbach: „Scalable Semantik Web Data Management Using Vertical Partitioning“, Endowment, VLDB 2007

[14] Sergei Melnik: „Storing RDF in a relational database“ „http://infolab.stanford.edu/ melnik/rdf/db.html“

[15] Richard Snodgrass, Ilsoo Ahn: „Temporal Databases“ Computer, 1986, IEEE

[16] Richard Snodgrass: „Developing Time-Oriented Database Applications in SQL“, Morgan Kaufmann 2000

[17] Jan Chomicki,David Toman, „A Mini-course on Temporal Databases“ Monmouth University

[18] Jan Chomicki, David Toman, „Temporal Logic in Information Systems“ BRICS Lecture Series LS-97-1, ISSN 1395-2048, November 1997

[19] M.H. Böhlen, R. Busatto and C.S. Jensen „Point-Versus Interval-based Temporal Data Models“, Proceedings ICDE 98

[20] Claudio Gutierrez, Carlos A. Hurtado, Alejandro Vaisman:“Introducing Time into RDF“ IEEE transactions on knowledge and data engineering, Vol 19, No.2, Feb 2007

[21] Octavian Udrea, Diego Reforgiato Recupero, V. S. Subrahmanian: „Annotated RDF“ ACM Transactions on Computational Logic, December 2007

[22] Manolis Gergatsoulis, and Pantelis Lilis: „Multidimensional RDF“ LNCS Springer, Volume 3761/2005, Pages 1188-1205

[23] Thomas Neumann, Gerhard Weikum: „RDF-3X: a RISC-style Engine for RDF“ VLDB 2008, Auckland, New Zealand

[24] Octavian Udrea,Andrea Pugliese, V.S. Subrahmanian:“GRIN: A Graph Based RDF Index“ AAAI, 2007

[25] Andrea Pugliese, Octavian Udrea, V.S. Subrahmanian:“Scaling RDF with Time“ WWW2008, April 21-25, 2008, Beijing, China

[26] Andreas Harth, Stefan Decker: „Optimized Index Structures for Querying RDF from the Web“, LA-WEB 2005

[27] Ying Yan, Chen Wang, Aoying Zhou, Weining Qian, Li Ma, Yue Pan: „Efficiently Queyring RDF Data in Triple Stores“ WWW 2008, Beijing, April, 2008

[28] Betty Salzberg, Vassilis J. Tsotras: “ Comparsion of Access Methods for Time-Evolving Data “ ACM Computing Surveys Vol.31, No. 2, June 1999

[29] Tansel, Clifford, Gadia, Jajodia, Segev, Snodgrass: „Temporal Databases Theory, Design, and Implementation“ The BenjaminCummings Publishing Company Inc., 1993

[30] David Lomet, Mingshen Hong, Rimma Nehme, Rui Zhang: „Transaktion Time Indexing with Version Compression“ VLDB 2008, Auckland, New Zealand

[31] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, Peter Widmayer: „An Assymptotically Optimal Multiversion B-Tree“ VLDB Journal 5(4): 264-275 (1996)

[32] Tuukka K. Haapasalo, Seppo S. Sippu, Ibrahim M. Jaluta, Eljas O. Soisalon-Soininen: „Concurrency Control and Recovery for Multiversion Database Structures“ PIKM 08, October 30, 2008, Napa Valley, California, USA.

[33] Peter J. Varman and Rakesh M. Verma: „An Efficient Multiversion Access Structure“ IEEE Transactions on Knoledge and Data Engineering, vol. 9, no. 3, may/june 1997

[34] Driscoll J.R., Sarnak N., Sleator D.D. and Tarjan R.E. „ Making data structures persistent“ Journal of Comp. and System Sci. 38:86 - 124, 1989

[35] Lars Arge, Jeffrey Scott Vitter: „ Optimal Dynamic Interval Management in External Memory “ In Proc. IEEE Symp. on Foundations of Comp. Sci 1996

[36] Jost Enderle, Matthias Hampel, Thomas Seidl: „Joining Interval Data in Relational Databases“ SIGMOD 2004, Paris

[37] Marion A. Nascimento, Margaret H. Dunham: „ Indexing Valid Time Databases Via B+-trees The MAP21 Approach“ Technical Report 1997, Southern Methodist University Dallas USA

[38] Max Völkel et.al. : „ SemVersion - Versioning RDF and Ontologies “ Knowledge Web Deliverable 2.3.3.v1, University of Karlsruhe. June 2005.

[39] Atanas Kiryakov, Damya Ognyanov: „Tracking Changes in RDF(s) Repositories“ Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management 2002, Pages 373 - 378

[40] Sören Auer, Heinrich Herre: „A Versioning and Evolution Framework for RDF Knowledge Bases “ In Proceedings of Ershov Memorial Conference 2006

[41] Yannis Tzitzikas, Yannis Theoharis, and Dimitris Andreou: „On Storage Policies for Semantic Web Repositories That Support Versioning“ The Semantic Web: Research and Applications, LNCS 2008, Springer, Pages 705-719

[42] Shu-Yao Chien, Vassilis J. Tsotras, Carlo Zanilo, Donghui Zhang: „Supporting Complex Queries on Multiversion XML Documents“ ACM Transaction on Internet Technology, Vol 6, No.1, February 2006, Pages 53-84

[43] Amit Shet, Matthew Perry:“Traveling the Semantic Web through Space, Time, and Theme“IEEE Internet Computing,Volume 12 , Issue 2 (March 2008) Pages 81-86

[44] Mohamed Ouksel, Peter Scheuermann: „Multidimensional B-Trees Analysis of Dynamic Behavior“ BIT Numerical Mathematics, Volume 21, Number 4 / December, 1981

[45] Harry Leslie, Rohit Jian, Dave Birdsall, Hedieh Yaghmai:“Efficient Search of Multidimensional B-Trees“ VLDB 1995, Zurich, Switzerland

[46] Goetz Graefe:“Sorting And Indexing With Partitioned B-Trees“ Proceedings of the 2003 CIDR Conference

[47] Donald E. Knuth „The Art of computer programming: Volume 3 / Sorting and Searching“ Addison-Wesley. 1998

[48] Hanan Samet: „Foundations of Multidimensional and Metric Data Structures“ Morgan Kaufmann, 2006

[49] T. Ottman, P Widmayer: „Algorithmen und Datenstrukturen“ Spektrum Akademische Verlag, 3 Auflage

[50] Ralf Hartmut Güting, Stefan Dieker:“Datenstrukturen und Algorithmen“ Teubner, Vieweg/Teubner; Auflage: 3. A. (10. Dezember 2004)

[51] Mike Stonebraker et.al.: „C-Store: A Column-oriented DBMS“ VLDB 2005, Trondheim Norway.

[52] S. Abiteboul,R Hull, V Viaunu: „Foundations of Databases“ Addison-Wesley 1995

Danksagung

An dieser Stelle möchte ich mich besonders bei meinem Professor Herrn Dr. Bernhard Seeger bedanken, der mich während meiner Diplomarbeit betreut und umfangreich unterstützt hat. Besonders möchte ich mich aber bei Herrn Sonny Vaupel bedanken, der mir aufgrund seiner langjährigen Erfahrungen eine große Hilfe war.
Mein ganz besonderer Dank geht meiner Kommilitonin Anastasiya Nonenmacher, die mit bei vielen Formulierungen und auch bei der Korrektur der Diplomarbeit sehr hilfreich zur Seite stand. Auch bei meinem Kommilitonen Khalid Balafkir möchte ich mich für die vielen Stunden Korrekturlesen bedanken. Vielen Danke auch an Herrn Philip Prange, der mir bei Problemen mit der Formulierung immer zur Seite stand.
Ich möchte diese Arbeit meinem Vater widmen. Meinen Mutter und Vater möchte ich meinen tiefen Dank aussprechen, denn ohne sie wäre dieses Studium niemals möglich gewesen. Ein herzliches Dankeschön geht auch an meine kleine Bruder und Schwester, denn ohne ihre moralische Unterstützung wäre ich niemals fertig geworden.

Erklärung

Hiermit versichere ich, daß ich diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Marburg, im Februar 2009