Apache Derby

Architektur und Implementierung

Jochen Woidich	Architektur
Stefan Bußweiler	Architektur
Michael Eckel	JDBC, *SVN/Trac Admin*
Eugen Labun	JDBC, *Editor*
Tobias Beimborn	Anfragebearbeitung
Steffen Rupp	Anfragebearbeitung
Mark Schlüter	B^+-Baum Implementierung
Dieter Kramer	B^+-Baum Implementierung
Thomas Kraebihl	Transaktionen und Isolationslevel
Christian Weber	Logging und Recovery
Widia Ahadi Putra	Logging und Recovery

26. März 2010

Referent: Prof. Dr. Burkhardt Renz
Veranstaltung: Architektur und Implementierung eines DBMS

Fachhochschule Gießen-Friedberg
Fachbereich Mathematik, Naturwissenschaften und Informatik

WS 2009/2010

Inhaltsverzeichnis

1 Architektur

Die Architektur von Apache Derby lässt sich auf zwei grundsätzlich verschiedene Weisen betrachten: Die logische Sicht, die sehr übersichtlich darstellt, wie Daten sequenziell durch das System fließen und verarbeitet werden und die Sicht auf die tatsächliche Softwarearchitektur, die eingesetzt wird um diese Funktionen umzusetzen.

Zuerst wird die logische Architektur von Derby in Form eines Schichtenmodells erläutert, um einen ersten Überblick über die Funktionsweise des Systems, die in späteren Kapiteln näher erläutert wird, zu geben. Des Weiteren werden die zwei Modi, in denen Apache betrieben werden kann, geschildert. Im Kapitel Softwarearchitektur werden die wichtigsten Bestandteile der in Derby verwendeten Architektur eingeführt und deren Zusammenspiel erläutert. Im Anschluss findet eine Darstellung des Derby-Engine-Systems im eimgebetteten Modus statt.

1.1 Logische Architektur

1.1.1 Schichtenmodell

Das Schichtenmodell eignet sich um einen ersten Überblick über die logische Architektur der Engine von Derby zu erlangen. Die vorhandenen Schichten ergeben sich aus den vier größten Java-Packages im Quellcode: JDBC, SQL, Store und Services.

Die JDBC-Schicht stellt die einzige Schnittstelle dar, über die Anwendungen mit Derby kommunizieren können und besteht aus Implementierungen der JDBC-Schnittstellen. Anwendungen können also ausschließlich über die von der jeweiligen JDBC-Version spezifizierten Schnittstellen auf die Funktionen von Derby zugreifen. Die JDBC-Schicht nimmt die Anfragen entgegen und leitet sie an die nächsttiefere Schicht, SQL, weiter.

Die Aufgaben der SQL-Schicht umfassen das Kompilieren und die anschließende Ausführung von Statements, was sich in den beiden Unterschichten Compilation und Execution widerspiegelt. Das Kompilieren lässt sich grob in die folgenden Schritte zerlegen: Zuerst wird das übergebene Statement mit Hilfe eines Parsers in eine Baum-Form gebracht. Anschließend werden alle Objekte im Baum den entsprechenden Objekten der Datenbank (z.B. Tabellen, Spalten) zugeordnet. Nach dem Optimieren des Baums, das zur Auffindung des schnellsten Zugriffspfades dient, wird aus dem Ergebnis eine Java-Klasse erstellt, die sofort geladen und instanziert wird. Diese Klasse repräsentiert den fertigen Zugriffsplan. Zum Ausführen des Statements wird die Methode `execute` der gerade erstellten Instanz aufgerufen, die ein Objekt der Derby-spezifischen Klasse `ResultSet` zurück gibt. Die Methoden dieser ResultSets verwenden die Store-Schicht, um auf die Funktionen der Datenbank zuzugreifen. Wie die SQL-Schicht besteht Store aus zwei Unterschichten. ResultSets nutzend die Access-Schicht von Store, die über ihre Schnittstelle zeilen- und spaltenbasierten Zugriff auf

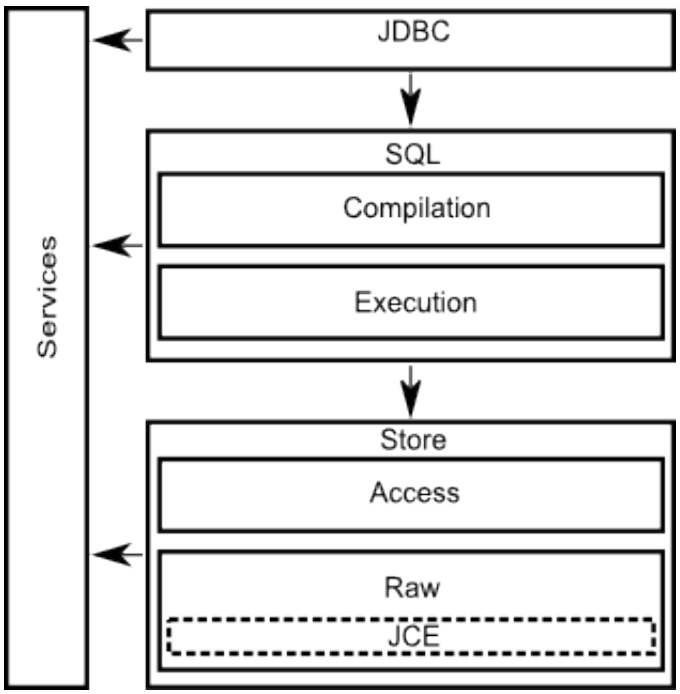

Abbildung 1.1: Apache Derby Schichtenmodell.

die Datenbestände verfügbar macht. Indexing- und Sortierfunktioen, sowie Transaktionen und Isolations-Levels sind ebenfalls in Access angesiedelt.

Die darunter liegende Raw-Schicht kapselt den physikalischen Zugriff auf die Datenbestände der Datenbank und ist ebenfalls für Transaktions-Logging und -Management zuständig. Die Schnittstelle zum Dateisystem ist so aufgebaut, dass sie sich durch beliebige Implementierungen austauschen lässt. So können Informationen je nach eingesetzter Implementierung beispielsweise in einzelnen Dateien, in JAR-Archiven oder nur im Arbeitsspeicher gehalten werden. Derby ermöglicht es, Daten verschlüsselt zu speichern. Zu diesem Zweck finden sich in der Raw-Schicht auch Schnittstellen für die Java Cryptographic Extension (JCE).

Alle oben genannten Schichten nutzen die Services-Komponente. Services kapselt häufig und von mehreren Schichten benötigte Funktionen, wie z.B. Caching-Mechanismen, Lock-Management oder Logging. Auf diese Art sind Funktionen, die nicht exklusiv zu einer einzigen Schicht gehören an einem zentralen Punkt verfügbar und können leicht gepflegt werden.

Der Datenfluss durch die genannten Schichten, der die Bearbeitung einer Anfrage an die Datenbank darstellt, kann wie folgt zusammengefasst werden: Anfragen an die Datenbank werden über die SQL-Schicht an das System gestellt, die die einzige öffentliche Schnittstelle darstellt. Die Anfragen werden anschließend in der SQL-Schicht optimiert und zu einer Java-Klasse kompiliert. Die so entstandene Klasse greift über die von der Store-Schicht zur Verfügung gestellten Schnittstellen schließlich auf die im Dateisystem gespeicherten Informationen zu.

1.1.2 Embedded- & Client/Server-Modus

Apache Derby bietet das Datenbanksystem in einem eingebetteten und in einem Client/Server-Modus an. Dabei stellt, wie oben bereits beschrieben, JDBC in beiden Modi die einzige Schnittstelle für die Anwendung dar.

Bei dem eingebetteten Modus ist die Datenbank Teil der Java-Applikation und nach außen nicht sichtbar. Datenbank und Applikation laufen im selben Prozess der Java Virtual Machine. Der Zugriff auf die Datenbank ist ausschließlich über eine JVM möglich. Die Datenbank ist einfach zu benutzen, sehr schnell und bringt einen geringen Konfigurationsaufwand mit sich.

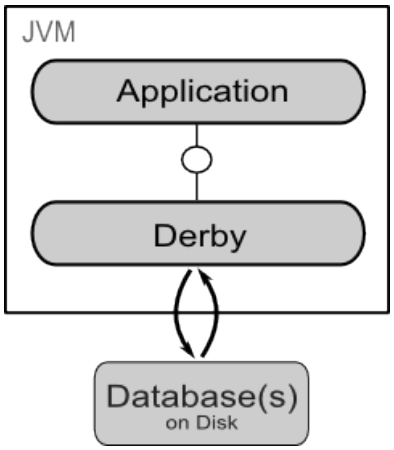

Abbildung 1.2: Apache Derby Embedded-Modus.

Des Weiteren ist es möglich Derby in einem Client/Server-Modus laufen zu lassen. Dazu stellt Derby das Framework *Derby Network Server* zur Verfügung. Das Framework und Derby laufen in derselben JVM. Die Funktionalität des Framework besteht darin, Anfragen von Client-Anwendungen entgegen zunehmen und an die Datenbank weiterzuleiten. Die Client-Anwendungen laufen in einer anderen JVM, entweder auf derselben oder auf einer entfernten Maschine.

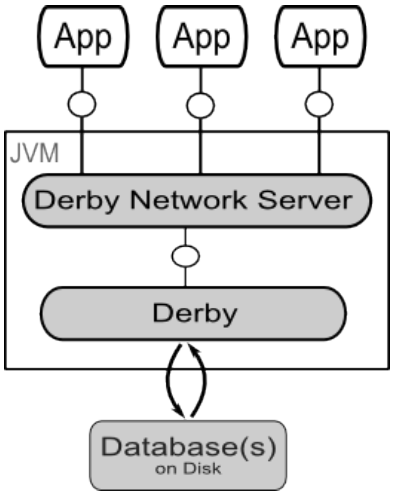

Abbildung 1.3: Apache Derby Client/Server-Modus.

1.2 Softwarearchitektur

1.2.1 Übersicht

Apache Derby implementiert eine Microkernel-Architektur. Systeme nach diesem Muster sind höchst modular aufgebaut. Kernfunktionalitäten werden durch sogeannte Module zur Verfügung gestellt, welche von einem zentralen Kernel verwaltet werden. Ein Modul ist vielseitig einsetzbar und kann ausgetauscht werden. Durch diesen Aufbau weisen die Systeme eine hohe Flexibilität und Erweiterbarkeit auf.

Das Derby-System besteht aus einem Monitor, Services und Modulen. Der Monitor agiert als Kernel und verwaltet die Services und die Module. Des Weiteren wird unterschieden zwischen Service-Modulen und System-Modulen. Ein Service-Modul lebt im Kontext eines Services. System-Module übernehmen hingegen systemweite Aufgaben. Abbildung 1.4 verdeutlicht diesen Aufbau.

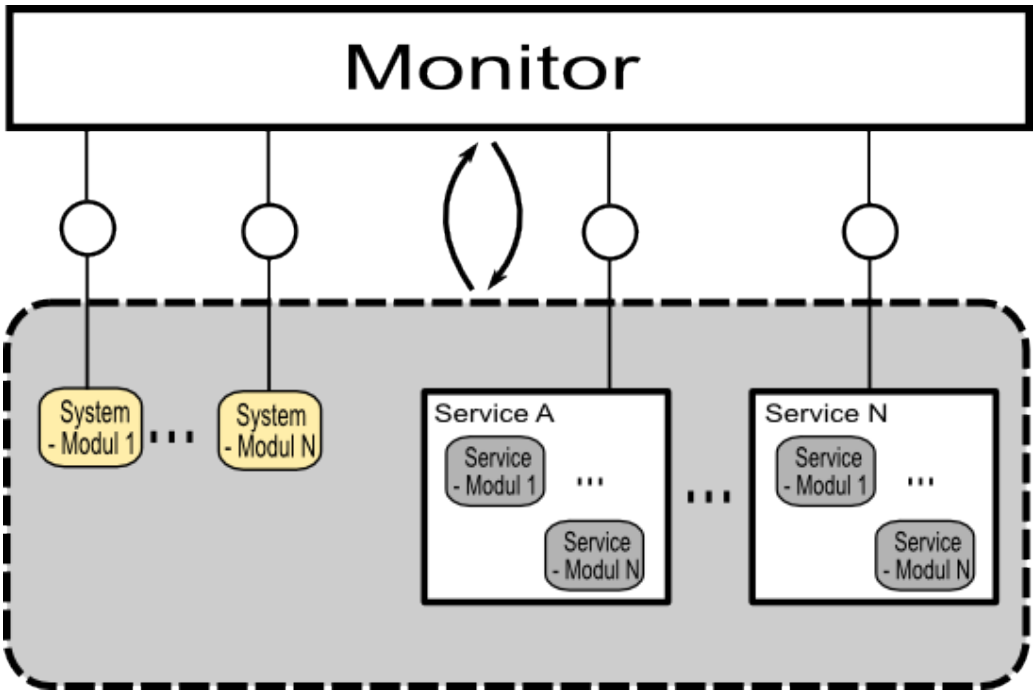

Abbildung 1.4: Apache Derby Monitoring.

Dieses Kapitel beschreibt die Softwarearchitektur von Apache Derby. Zunächst werden die Funktionsweise, die Struktur und die unterschiedlichen Typen eines Moduls dargestellt. Im Anschluss werden die aus den Modulen resultierenden Services erläutert. Nachdem der Aufbau dieser Elemente verdeutlicht wurde, werden der Monitor und das Modul- und Servicemanagement betrachtet.

1.2.2 Module

Was ist ein Modul ?

Ein Modul ist die kleinste Einheit in Derby und stellt eine grundlegende Funktionalität zur Verfügung. Apache Derby trennt die API strikt von der Implementierung. Jedes Modul wird durch eine Menge von Java-Interfaces definiert und es können beliebig viele Implementierungen existieren. Durch diesen Aufbau können die Implementierungen der einzelnen Module beliebig ausgetauscht werden.

Ein Modul wird im System durch die folgenden drei Punkte zugeordnet:

- Factory Interface / Protokoll
  Die API eines Moduls wird durch genau ein einzelnes Java Interface repräsentiert und besitzt in der Regel die Endung Factory. Derby bezeichnet dieses Factory Interface auch häufig als Protokoll.
- Identifier
  Ein Modul kann zusätzlich durch ein String identifiziert werden. Der Wert kann `null` oder fest einprogrammiert sein. Wenn der Wert null ist, wird das Modul als unidentified bezeichnet.
- Properties-Set
  Einem Modul können Status-Informationen zugeordnet werden. Diese bestimmen z.B. das Verhalten zur Laufzeit, die zu ladene Implementierung usw. Die Status-Informationen werden in Java durch die Klasse `java.util.Properties` dargestellt. Aus diesem Grund werden die Status-Informationen von Derby auch als Properties-Set bezeichnet.

  Beachte: Die Angabe eines Properties-Set für ein Modul ist optional.

Modul-Struktur

Ein Modul muss gestartet und beendet werden können. Weiterhin müssen für ein Modul manchmal bestimmte Kriterien erfüllt sein, damit es überhaupt gestatet werden kann. Um dies zu gewährleisten stehen zwei Interfaces zur Verfügung.

Das Interface `ModuleControl` stellt die Methoden `boot` und `stop` zur Verfügung. In der `boot`-Methode wird ein Modul korrekt initialisiert, erhält das Properties-Set, lädt abhängige Module und wird letztenendes gestartet. Die `stop`-Methode beendet ein Modul ordnungsgemäß.

Für ein Modul können verschiedene Implementierungen existieren welche je nach Umgebung geladen werden. Das Interface `ModuleSupportable` bietet einem Modul die Möglichkeit, diese vor dem Starten zu überprüfen. Anhand des Properties-Set kann die Methode `canSupport` überprüfen ob die übergebenen Properties die erwarteten Voraussetzungen erfüllen.

Bemerkung: Derby enthält Module die von diesem Konzept abweichen.

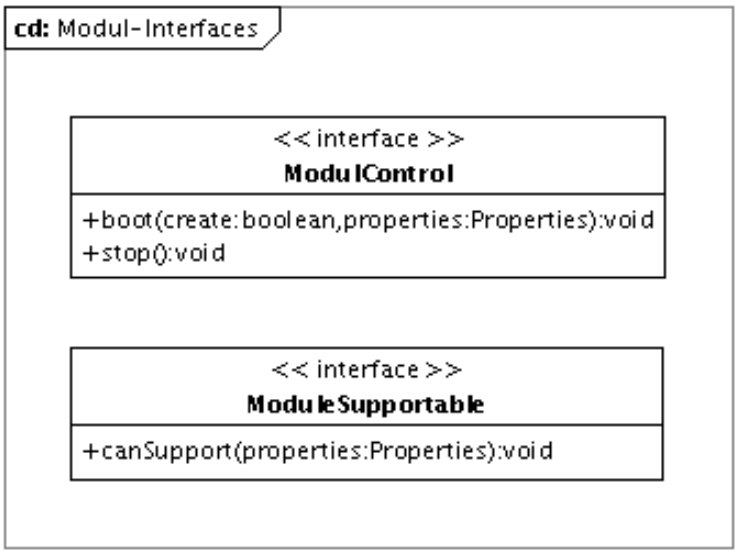

Abbildung 1.5: Modul-Interfaces

Modul-Typen

Module lassen sich in zwei Kategorien einteilen, dabei unterscheiden sie sich von ihrem Einsatzgebiet innerhalb eines Services oder im Gesamtsystem.

- Service-Module
  Service-Module sind einem bestimmten Service zugeordnet. Sie erhalten optional Status-Informationen in Form eines Properties-Set von ihrem Service. Die Lebensdauer eines Service-Moduls endet spätestens mit der des Services.
- System-Module
  System-Module sind keinem bestimmten Service zugeordnet, besitzen keinen Identifier und kein Properties-Set. Sie gehören zum laufenden System und nehmen grundlegende Aufgaben wahr. Die Lebensdauer endet spätestens mit der des Monitors. Bekannte System-Module sind z.B. InfoStreams, UUIDFactory, TimerFactory, CacheFactory, TypeCompilerFactory usw.

1.2.3 Services

Was ist ein Service ?

Ein Service ist ein logisches Gebilde, das mehrere Module bündelt und dadurch eine vollständige Funktionalität zur Verfügung stellt. Ein Service besteht aus einem Primär-Modul und mehreren Service-Modulen. Das Primär-Modul repräsentiert die API des Services nach außen. Aus diesem Grund erhält das Primär-Modul seine Bezeichnung. Diese existiert nur im Kontext eines Services und bezeichnet keinen neuen Modul-Typ.

Ein Service wird identifiziert durch das Factory Interface bzw. Protokoll des Primär-Moduls, einem Identifier und einem optionalem Properties-Set. Der Identifier des Primär-Moduls

und der des Services sind i.d.R. jedoch nicht dieselben. Die Identifizierung von den Service-Modulen findet anhand eines Paares, bestehend aus Factory Interface und Identifier, statt. Dieses Paar ist innerhalb des Services eindeutig. Die Lebensdauer eines Service endet spätestens mit der des Monitors.

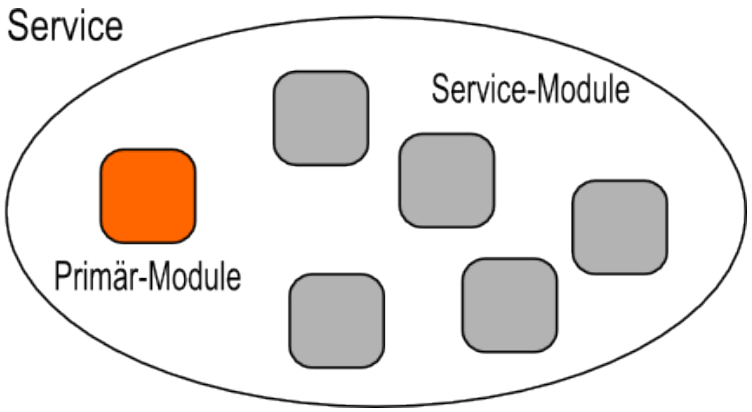

Abbildung 1.6: Apache Derby Service.

Persistent / Nicht-Persistent

Einem Service können beim Starten verschiedene Status-Informationen zur Verfügung gestellt werden. Bei einem nicht-persistenten Service werden diese Informationen zur Laufzeit generiert. Ein persistenter Service bezieht diese Informationen z.B. aus einer Datei auf dem Dateisystem. Diese Datei kann von dem Service erzeugt, verändert und neu eingelesen werden.

Nicht-persistente Services in Derby sind z.B. JDBC und Authentication. Ein persistenter Service ist z.B. Database, welcher für jede Datenbank im System erzeugt wird. Die Status-Informationen werden aus einer Datei namens `service.properties` bezogen. Diese Datei wird im Datenbankverzeichnis erstellt und wird nach jedem Booten des Services erneut eingelesen. Der Service stellt die enthaltenen Status-Informationen seinen Service-Modulen zur Verfügung.

Damit ein Service gestartet werden kann wird das Factory Inferface des Primär-Moduls benötigt. Dieses wird durch ein Property angegeben und unterscheidet sich in der Darstellung bei nicht-persistenten und persistenten Services:

Das Property eines nicht-persistenten Services wird zur Laufzeit generiert. Ein Servicename beginnt mit der Kennung `derby.service` gefolgt von dem eigentlichen Namen.

```
derby.service.jdbc=org.apache.derby.jdbc.InternalDriver
```

In diesem Fall handelt es sich um den Service `jdbc`, dem das Factory Interface des Primär-Moduls `org.apache.derby.jdbc.InternalDriver` zugeordnet wird.

Bei einem persistenten Service werden die Properties aus einer Datei bezogen. Dies wird durch den Term `serviceDirectory` angegeben, welcher einen Speichertyp auf Systemebene bezeichnet.

```
derby.service.derbyDB=serviceDirectory
```

Der Typ `serviceDirectory` besagt, das die Properties für diesen Service in einer Datei im Verzeichnis namens `derbyDB` zu finden sind. Dieses Verzeichnis enthält eine Datei mit dem Namen service.properties. Die Angabe des Factory Inferfaces findet in dieser Datei durch das Property `derby.protocol` statt. Dieses erhält als Wert den Namen des Factory Interfaces.

```
derby.protocol=org.apache.derby.database.Database
```

System-Service

In einem laufenden Derby-System existiert ein spezieller Service, der als System-Service bezeichnet wird. Im Gegensatz zu allen anderen Services hat er kein Primär-Modul, keinen Identifier und benötigt kein Properties-Set. Seine Aufgabe besteht einzig und allein darin, die bereits vorgestellten System-Module zu kapseln.

1.2.4 Monitor

Was ist der Monitor?

Der Monitor ist eine zentrale Komponente im System und agiert als Kernel. Er ist verantwortlich für das Starten und Beenden von Services und Modulen. Außerdem lädt er unter bestimmten Bedingungen für ein Modul unterschiedliche Implementierungen.

Beim Laden eines Moduls muss der Monitor die verschiedenen Modul-Typen berücksichtigen. Eine konkrete Beschreibung für den Start von Modulen und Services befindet sich im Kapitel 1.2.5.

Wie bereits erwähnt trennt Apache-Derby die API strikt von der Implementierung. Die Aufgabe des Monitors ist das Laden der korrekten Implementierung eines Moduls, da sich diese auf den verschiedenen Systemen unterscheiden kann. Dazu stellt der Monitor einen eigenen Klassenlader zur Verfügung. Ein Beispiel ist das Laden des JDBC-Treibers. Abhängig von der eingesetzten Version der JVM wird JDBC-3.0 oder JDBC-4.0 geladen. Ein weiteres Beispiel ist das Laden der Implementierung der RAW-Schicht. Jenachdem welche Implementierung geladen wird, werden die Daten als Datei, JAR-Archiv usw. gehalten. Die genaue Funktionsweise des Klassenladers wird in Kapitel 1.2.5 beschrieben.

1.2.5 Modul- & Service-Management

Einführung

Wie in den vorherigen Kapiteln erläutert besteht ein laufendes Derby-System mindestens aus einem Monitor und einer Menge von Services, die sich wiederum aus einzelnen Modulen zusammensetzen.

Um das System in einen lauffähigen Zustand zu bringen ist es nötig, die benötigten Klassen zu laden und anschließend alle Services samt ihrer Module zu starten. Zusätzlich muss es während dem Betrieb die Möglichkeit geben, bei Bedarf neue Services bzw. Module zu starten oder bereits laufende Instanzen im System aufzufinden.

Im Folgenden wird ein Überblick über die Komponenten gegeben, die direkt mit dem Service- und Modul-Management beschäftigt sind. In den anschließenden Unterkapiteln werden der Klassenlademechanismus sowie das Starten von Modulen und Services behandelt.

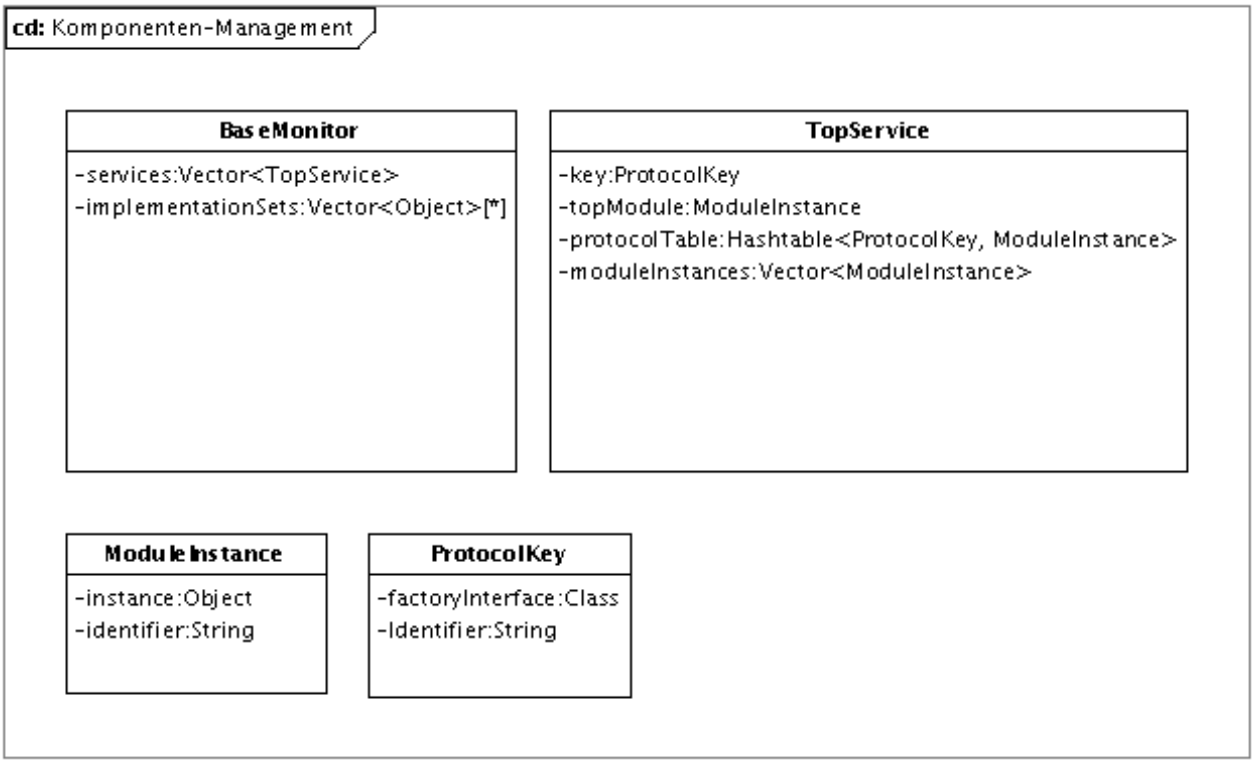

Abbildung 1.7: Übersicht über die wichtigsten Management-Klassen

Jeder im System laufende Service wird durch ein Objekt der Klasse `TopService` repräsentiert. Das Attribut `topModule` stellt jeweils eine, in ein Objekt vom Typ `ModuleInstance` gekapselte, Instanz des Primär-Moduls des Services dar.

Jeder TopService hält jeweils zwei Referenzen auf die zu ihm gehörenden Module. Der Vektor `moduleInstances` beinhaltet direkte Referenzen auf alle laufenden Instanzen in der Reihenfolge, in der die Module erstellt wurden und wird nur intern verwendet. Zusätzlich sind alle Modul-Instanzen über die Hash-Tabelle `protocolTable` zugänglich, die

beim Suchen von Modulen verwendet wird. Als Schlüssel werden Instanzen der Klasse `ProtocolKey` verwendet. Durch die Kombination von Factory Interface und Modul-Identifier (vgl. 1.2.2) ist garantiert, dass jeder Schlüssel innerhalb eines Services einzigartig ist.

Zu beachten ist, dass ausnahmslos alle laufenden Module in Services leben. Auch die System-Module, die zu keinen speziellen Service gehören, werden in einem TopService gehalten, der vor allen anderen beim Start von Derby erstellt wird. Dieser Service wird als System-Service bezeichnet. (vlg. Kapitel 1.2.3)

Der Monitor, bzw. dessen Hauptimplementierung `BaseMonitor`, hält im Vektor `services` Referenzen zu allen laufenden Services. Da das Vorhalten von verfügbaren Modul-Implementierungen ebenfalls eine Hauptaufgabe des Monitors ist werden vom Klassenlader instanzierte Objekte im Attribut `implementationSets` gespeichert.

Klassenlader

Wie in den vorhergehenden Kapiteln erwähnt ist die Engine von Derby extrem modular und flexibel aufgebaut. Die strikte Trennung von (Modul- bzw. Service-) Schnittstellen und den eigentlichen Implementierungen und die automatische Anpassung der Engine auf das laufende System bringen neben vielen Vorteilen allerdings auch Schwierigkeiten mit sich: Beim Starten von Modulen (und Services) muss eine zum jeweiligen Factory Interface passende Implementierung gefunden und, falls noch nicht geschehen, instanziert werden. Abhängigkeiten wie z.B. zusätzliche, für die einzelnen Module benötigte, Klassen sind dabei ebenfalls aufzulösen.

Der Monitor stellt zu diesem Zweck einen eigenen Klassenlademechanismus zur Verfügung, der das Instanzieren und vorhalten der benötigten Klassen übernimmt.

Wie auch im Rest des Systems wird hierbei in großem Umfang Gebrauch von Properties gemacht. Benötigte Klassen werden anhand von Property-Sätzen geladen, die entweder aus Dateien und der JRE bezogen oder dynamisch zur Laufzeit generiert werden können. Folgende Properties werden verwendet:

- `derby.module.<tag>=Klassenname`
  Dieses Property definiert die Implementierung eines Moduls im System. `<tag>` kann durch einen beliebigen String ersetzt werden und dient lediglich der Zuordnung zu den im Folgenden ebenfalls beschriebenen Properties. Meist wird der Name des Moduls verwendet, dass von Klassenname implementiert wird. Die mit Klassenname referenzierte Klasse muss das Interface `ModuleControl` implementieren, damit das Modul später korrekt gestartet (und gestoppt) werden kann. (vgl. 1.2.2)
- `derby.env.jdk.<tag>={1,2,4,5,6,7}`
  Definiert, dass das mit `<tag>` referenzierte Modul nur ab einer bestimmten Java-Version verwendet werden soll oder kann. Die möglichen Werte kodieren dabei die benötigte Version wie folgt:
  1 - jdk ab 1.1.x
  2 - jdk ab 1.2.x
  3 - jdk ab 1.3.x (Java 2)
  4 - jdk ab 1.4 x (Java 4)

5 - jdk ab 1.4.2
6 - J2SE ab 5.0
7 - J2SE ab 6.0

- `derby.env.classes.<tag>=Klassenname[,Klassenname]*`
  Mit diesem Property werden Klassen-Abhängigkeiten abgebildet. Die Implementierung des mit `<tag>` referenzierten Modul benötigt alle der durch Namen angegebenen Klassen um lauffähig zu sein.

Der Kern des Klassenladers, der eine Liste der genannten Properties auswertet und zum laufenden System passende Instanzen der in den Properties beschriebenen Modul-Implementierungen zurück gibt findet sich in der Methode `getImplementations` der Klasse `BaseMonitor`. Der Ablauf der Methode lässt sich auf diese Weise zusammenfassen:

Es wird über die Liste der übergebenen Properties iteriert. Wird ein Property der Form `derby.module.<tag>` gefunden, wird überprüft, ob für `<tag>` ein Eintrag der Form `derby.env.jdk.<tag>` existiert. Ist dies der Fall wird sichergestellt, dass die laufende Java-Version den Anforderungen der Modul-Implementierung entspricht. Anschließend werden die Abhängigkeiten der Implementierung überprüft. Zu diesem Zweck wird einfach versucht zu jeder im Property `derby.env.classes.<tag>` genannte Klassen eine Instanz zu erstellen, falls das Property überhaupt existiert.

Abschließend wird eine Instanz der eigentlichen Modul-Implementierung erstellt.

Schlägt einer der eben genannten Überprüfungen fehl oder treten beim Instanzieren der Klassen Fehler auf wird einfach zum nächsten Moduleintrag übergegangen, ansonsten wird die soeben erstellte Instanz in eine Liste mit möglichen Modulimplementierungen eingereiht, die nach Ablauf der Methode zurückgegeben wird.

Schon beim Start des Monitors wird auf diese Art eine Reihe von Property-Listen verarbeitet, insbesondere werden auch die von Derby gelieferten Modulimplementierungen auf diese Art instanziert. Die benötigten Properties werden aus der Datei `modules.properties` bezogen.

Der Monitor speichert alle in diesem Schritt gewonnenen Instanzen in einem zentralen Vektor, nämlich dem Attribut `implementationSets`. Sie werden später beim eigentlichen Laden und suchen von Modulen verwendet.

Starten von Modulen

Das Starten von einzelnen Modulen geschieht in einem engen Zusammenspiel zwischen dem Monitor und den im System laufenden Services.

Zum Starten eines Moduls, das über den Monitor veranlasst wird, werden im Wesentlichen vier Informationen benötigt:

- Ein Service-Modul (optinal)
- Das Factory Interface des zu ladenden Moduls

- Der Identifier des zu ladenden Moduls
- Ein Satz von Properties

Das Service-Modul dient der Auffindung des Services, in dem das zu ladende Modul später leben soll. Wir kein Service-Modul spezifiziert (`null`) wird automatisch davon ausgegangen, dass es sich bei dem Modul um ein System-Modul handelt, das im System-Service angesiedelt werden soll.
Andernfalls iteriert der Monitor über die Liste der laufenden Services und prüft jeweils, ob eine Instanz des Service-Moduls im Service existiert. Sobald der Service gefunden ist übergibt der Monitor ihm die Aufgabe, das Modul zu booten.
Existiert im Service bereits eine laufende Instanz, die das benötigte Factory Interface implementiert, wird sie einfach an den Monitor zurückgegeben. Ist dies nicht der Fall nutzt der Service den Monitor, um mit Hilfe des Factory Interfaces und der Properties eine neue Modulinstanz zu erzeugen. Der Monitor nutzt dazu den in Kapitel 1.2.5 beschriebenen Klassenlademechanismus und greift außerdem auf die bereits beim Starten von Derby geladenen Instanzen zurück.
Nachdem das neu instanzierte Modul in die Liste der Modul-Instanzen des Services aufgenommen wurde wird schließlich, erneut über eine Hilfsfunktion des Monitors, die `boot`-Methode des Moduls aufgerufen.

Hiermit ist der Start des Moduls abgeschlossen.

Starten von Services

Der Start von vollständigen Services wird ebenso wie der Start von Modulen über den Monitor veranlasst. Alle zum vollständigen Booten eines Services benötigten Informationen können aus den Properties des Services bezogen werden (vgl. 1.2.3). Von besonderem Interesse ist das Factory Interface, das das Primär-Modul des Services spezifiziert.

Nachdem der Monitor überprüft hat, dass der Service nicht bereits läuft wird ein neuer Service erzeugt, der sofort im System registriert und angewiesen wird das Primär-Modul zu booten. Der darauf folgende Ablauf entspricht im Groben dem normalen Starten von Modulen, wie er im vorherigen Abschnitt behandelt wurde. Der Monitor muss nur nicht mehr nach dem Service des Moduls suchen.

Innerhalb der `boot`-Methode des gerade instanzierten Primär-Moduls, die automatisch beim Laden aufgerufen wird, werden nun alle weiteren für den Betrieb des Services benötigten, Module gestartet.

Mit Ablauf der `boot`-Methode des Primär-Moduls hat sich der Service also Stück für Stück selbst aufgebaut und ist damit voll funktionsfähig und im System nutzbar.

1.3 Das Derby-Engine-System

Damit sämtliche Funktionen in einem Derby-System zur Verfügung stehen bedarf es der Aktivität bestimmter Services und Module. Dieses Kapitel liefert einen Überblick über ein laufendes Derby-Engine-System im eimgebetteten Modus. Eine Anwendung startet die Derby-Engine und läuft mit dieser in derselben Java Virtual Machine.

An dieser Stelle sei auf das Schichtenmodell im Kapitel 1.1.1 zu Beginn verwiesen. Die einzelnen Schichten finden sich im laufenden System in Form von Serivces und Modulen wieder.

Abbildung 1.8 liefert einen Überblick über die aktiven Services und Module im eingebetteten Derby-Engine-System:

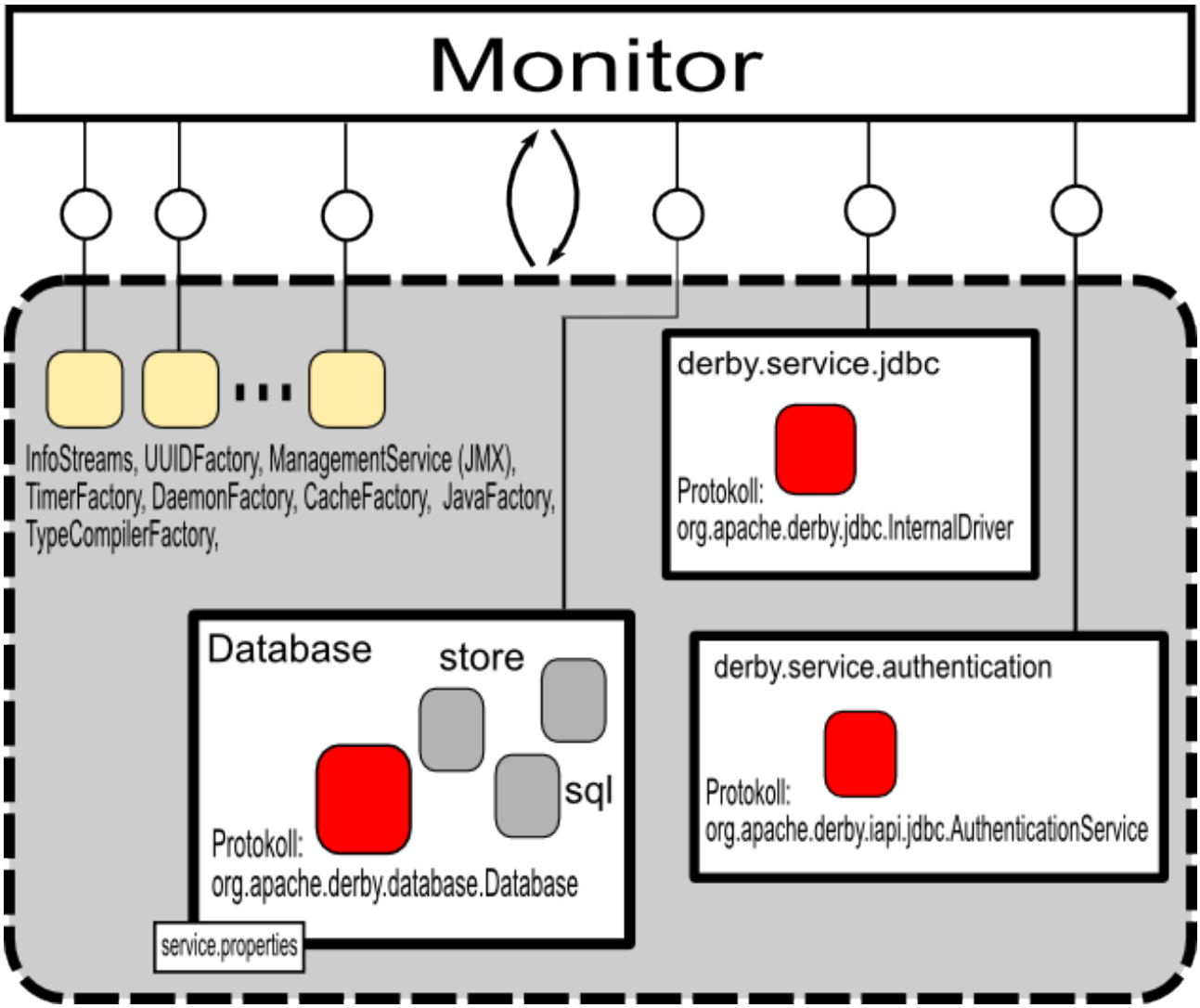

Abbildung 1.8: Apache Derby Engine.

Das laufende System im eingebetteten Modus besteht aus drei Services:

- derby.service.jdbc
  Der Service JDBC veranlasst das Laden des JDBC-Treibers und entspricht der JDBC-Schicht aus dem Schichtenmodell. Der Service ist nicht-persistent und wird durch das Protokoll *org.apache.derby.jdbc.InternalDriver* repräsentiert.

- derby.service.authentication
  Der nicht-persistente Service Authentication ermöglicht die Benutzer-Authentifizierung im DBMS. Durch eine konkrete Implementierung können verschiedene Mechanismen unterstützt werden.

- Database
  Für jede Datenbank im System wird ein persistenter Service erzeugt. Der Service repräsentiert die Schnittstelle zur Datenbank und besteht zur Laufzeit aus verschiedenen Service-Modulen der Schichten SQL und Store. Die Status-Informationen des Services werden aus der Datei service.properties bezogen. Diese Datei wird nach dem Erzeugen der Datenbank beim ersten Start des Services im Datenbankverzeichnis erstellt. Der Service liest die Status-Informationen nach jedem Neustart erneut ein und stellt sie bestimmten Service-Modulen zur Verfügung. Es werden Service-Module für die Unterschichten Compilation und Execution der SQL-Schicht als auch für Access und Raw der Store-Schicht geladen.

Des Weiteren bestehen neben den oben genannten Services mehrere System-Module welche systemweite Aufgaben übernehmen. Diese System-Module entsprechen in der logischen Sichtweise der Service-Komponente im Schichtenmodell. Architektonisch handelt es sich jedoch um System-Module welche dem System-Service zugeordnet sind. Beispielhaft seien zwei System-Module erläutert:

- TimerFactory
  Das System-Modul TimerFactory ermöglicht den Zugriff auf Timer-Ojekte für den unterschiedlichsten Gebrauch. Denkbar ist z.B. ein Timer der die Ausführung eines Statements nach einem bestimmten Intervall abbricht.
- CacheFactory
  Das Modul stellt einen Cache-Mechanismus für Objekte zur Verfügung. Die Verwendung ist vielseitig und wird z.B. zum Cachen von kompilierten SQL-Zugriffsplänen verwendet.

2 JDBC

JDBC ist eine API, um aus Java Programmen heraus auf Datenbanken zugreifen zu können. JDBC ist ein Markenname von Sun Microsystems, wird allerdings – in Anlehnung an ODBC – auch als Abkürzung für *Java Database Connectivity* verwendet. Dieses Kapitel beschreibt und analysiert diverse Aspekte der Realisierung und Implementierung von JDBC in Apache Derby. Dabei wird in Abschnitt 2.1 zunächst kurz auf die verschiedenen JDBC-Treibertypen eingegangen, bevor der Ladevorgang eines JDBC-Treibers genauer erläutert wird. Abschnitt 2.2 befasst sich daraufhin detaillierter mit den einzelnen Schritten, die nötig sind, um ein SQL-SELECT-Statement auszuführen.

In diesem Kapitel wird ausschließlich Java in Version 6 und JDBC in Version 4.0 verwendet. Deshalb werden in den folgenden Beispielen hauptsächlich Klassen geladen, die in ihrem Namen das Postfix „40“ tragen (z. B. `Driver40` und `EmbedStatement40`).

2.1 JDBC-Treiber

Um auf eine Datenbank mittels JDBC zugreifen zu können, wird ein JDBC-Treiber benötigt. Dieser Treiber ist für jede Datenbank speziell und wird üblicherweise vom Datenbankhersteller mitgeliefert. Apache Derby stellt diesen Treiber in Form eines nativen Protokoll-Treibers zur Verfügung. Insgesamt werden vier verschiedene JDBC-Treibertypen unterschieden, welche in Abschnitt 2.1.1 genauer betrachtet werden. Daraufhin wird in Abschnitt 2.1.2 das Laden, Instanziieren und die Verwendung eines JDBC-Treibers analysiert.

2.1.1 JDBC-Treibertypen

Dieser Abschnitt beschreibt die Funktionsweise der unterschiedlichen JDBC-Treibertypen und zeigt Vor- und Nachteile auf. Abbildung 2.1.1 enthält eine schematische Darstellung jedes Treibertyps.

Typ 1: JDBC-ODBC-Bridge

Wie der Name schon vermuten lässt, implementiert dieser Treibertyp keine eigene Funktionalität für Datenbankzugriffe, sondern delegiert alle JDBC-Aufrufe an den unterliegenden ODBC-Treiber. Einzig und allein der ODBC-Treiber kommuniziert mit der Datenbank. Er nutzt dazu allerdings eine Datenbankbibliothek, passend zum eingesetzten Datenbankmanagementsystem. Die Anfrageergebnisse, welche der ODBC-Treiber liefert, werden vom JDBC-Treiber entsprechend übersetzt und der Java-Anwendung zur Verfügung gestellt. Die JDBC-ODBC-Brigde fungiert hier also nur als Übersetzer und bildet somit eine zusätzliche

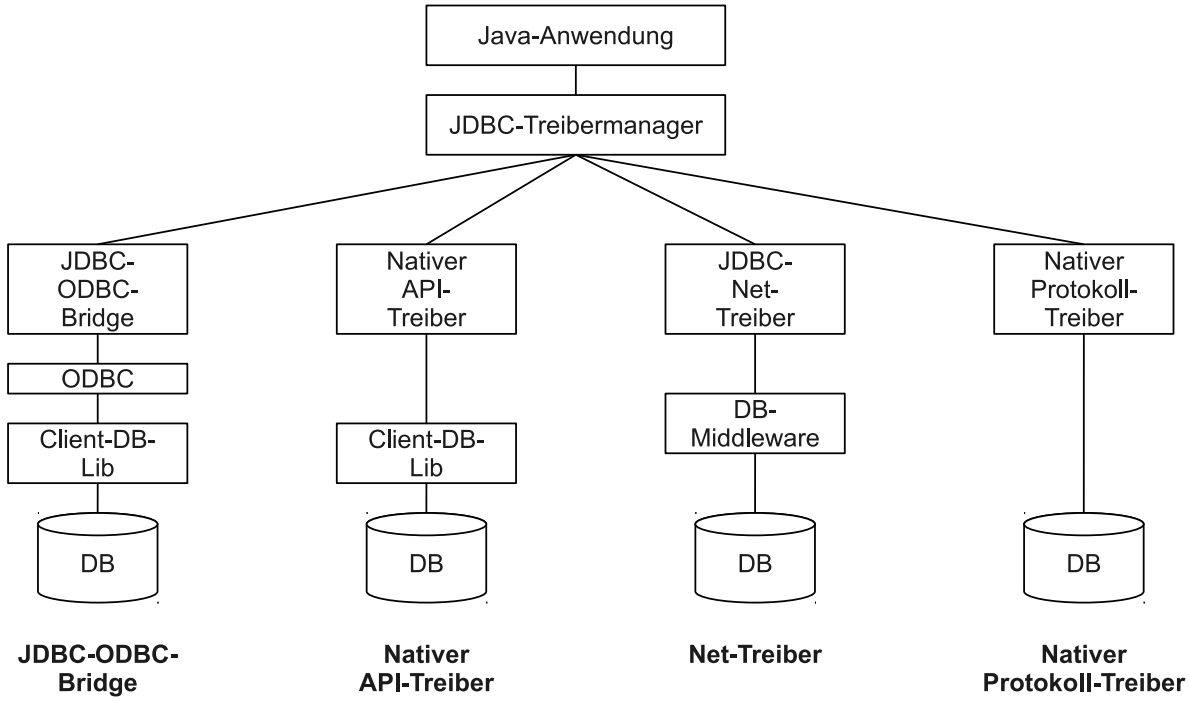

Abbildung 2.1: JDBC-Treibertypen

Abstraktionsebene. Durch die Hin- und Rückübersetzung der Befehle bzw. Ergebnisse, müssen Leistungseinbußen in Kauf genommen werden. Auf der anderen Seite, erhält man den Vorteil, jedes DBMS mit Java nutzen zu können, für das ein ODBC-Treiber vorhanden ist.

Typ 2: Nativer API-Treiber

Ein Treiber dieser Art interagiert direkt mit der Datenbankbibliothek des verwendeten DBMS. Durch den Wegfall der ODBC-Schicht, erhält man eine deutliche Leistungssteigerung, da Anfragen direkt an die Datenbankbibliothek gehen, ohne vorher eine „Übersetzungsschicht“ zu durchlaufen. Einen Nachteil stellt das Nichtvorhandensein von entsprechenden Treibern für manche DBMS dar, sodass in diesen Fällen auf einen JDBC-ODBC-Bridge-Treiber zurückgegriffen werden muss.

Typ 3: JDBC-Net-Treiber

Ein JDBC-Net-Treiber kommt zum Einsatz, wenn der Treiber nicht direkt mit dem Datenbanksystem, sondern mit einer Datenbank-unabhängigen Abstraktionsschicht („DB-Middleware“) kommuniziert. Die JDBC-Befehle müssen vom Treiber in entsprechende generische DBMS-Befehle übersetzt werden. Die Kommunikation erfolgt dabei über ein entsprechendes Netzwerkprotokoll. Seitens der DB-Middleware werden die generischen Befehle in Befehle des entsprechenden DBMS übersetzt. Die Vorteile einer solchen DB-Middleware liegen klar auf der Hand. Man ist deutlich flexibler, da die DB-Middleware den Zugriff auf verschiedene DBMS über eine einheitliche Schnittstelle anbietet. Das Anwendungsprogramm muss nichts über das verwendete DBMS wissen. Von der anderen Seite

betrachtet, ist bei dieser Lösung, ähnlich zum JDBC-ODBC-Treiber, wieder Übersetzungsarbeit seitens der DB-Middleware und evtl. des Treibers zu verrichten, was sich in Leistungseinbußen bemerkbar machen kann. Außerdem ist die Kommunikation vom Anwendungsprogramm zur Datenbank nun von einer weiteren Schicht abhängig. Fällt diese aus, kann keine Anwendung mehr auf Daten der von der DB-Middleware verwalteten Datenbanken zugreifen.

Typ 4: Nativer Protokoll-Treiber

Ein nativer Protokoll-Treiber ist komplett in Java geschrieben und ermöglicht den Zugriff auf ein DBMS, ohne eine DBMS-spezifische Bibliothek benutzen zu müssen. Es wird keinerlei Zwischenschicht zwischen dem DBMS und dem Treiber verwendet. Damit ist ein klarer Geschwindigkeitsvorteil gegenüber eines Typ 3-Treibers zu verzeichnen, da die Aufrufe direkt (ohne Übersetzung) an das verwendete DBMS weitergeleitet werden. Ein weiterer Vorteil, der durch einen in Java geschriebenen Treiber entsteht, ist die Plattformunabhängigkeit. Im Gegensatz zu einem Typ 3-Treiber wird bei einem Typ 4-Treiber allerdings kein generisches Kommunikationsprotokoll verwendet, weshalb für jedes DBMS ein angepasster Treiber vorhanden sein muss. Wie oben bereits erwähnt, nutzt Derby einen solchen nativen Protokoll-Treiber.

2.1.2 Laden eines JDBC-Treibers

Das Laden eines JDBC-Treibers erfolgt seit Java Version 6 und JDBC Version 4.0 völlig automatisch beim Laden der Treiberklasse. Diese besitzt dazu einen statischen Klasseninitialisierer („`static`-Block“), der beim Laden der Klasse automatisch ausgeführt wird. In Java Version 5 musste der Treiber noch wie folgt instanziiert werden:

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance();
```

Dieser Code ist in Java Version 6 optional, aber trotzdem zulässig. Wird der `static`-Block einer JDBC-Treiber-Klasse angestoßen, registriert dieser sich automatisch beim Treiber-Manager und steht damit im Programm zur Verfügung. Mit der Anweisung `DriverManager.getConnection()` kann man sich daraufhin eine Verbindung geben lassen und diese für Datenbankanfragen benutzen. Im Folgenden wird der Ladevorgang des Treibers genauer beschrieben. Abbildung 2.1.2 zeigt ein UML-Sequenzdiagramm, das den Ablauf veranschaulicht.

Ist der statische Klasseninitialisier der Klasse `EmbeddedDriver` erst einmal angestoßen, delegiert er seine komplette Arbeit an die Klassenmethode `boot`. Die Methode `boot` erstellt ein Objekt vom Typ `JDBCBoot()` und ruft auf diesem die Methode `boot` auf. `JDBCBoot` ist verantwortlich für das Starten eines Derby-Systems. Die Methode `boot` lässt sich von der Klasse `InternalDriver` zunächst den aktuellen Treiber geben, falls vorhanden. In unserem Fall ist dieser noch nicht vorhanden, sodass im nächsten Schritt über das Monitor-Konzept (vgl. Abschnitt 1.4) der Treiber veranlasst wird, zu starten. Dazu wird die Methode `startMonitor` der Klasse `Monitor` aufgerufen und die entsprechenden Parameter, vor allem der Klassenname des zu ladenden Treibers, übergeben.

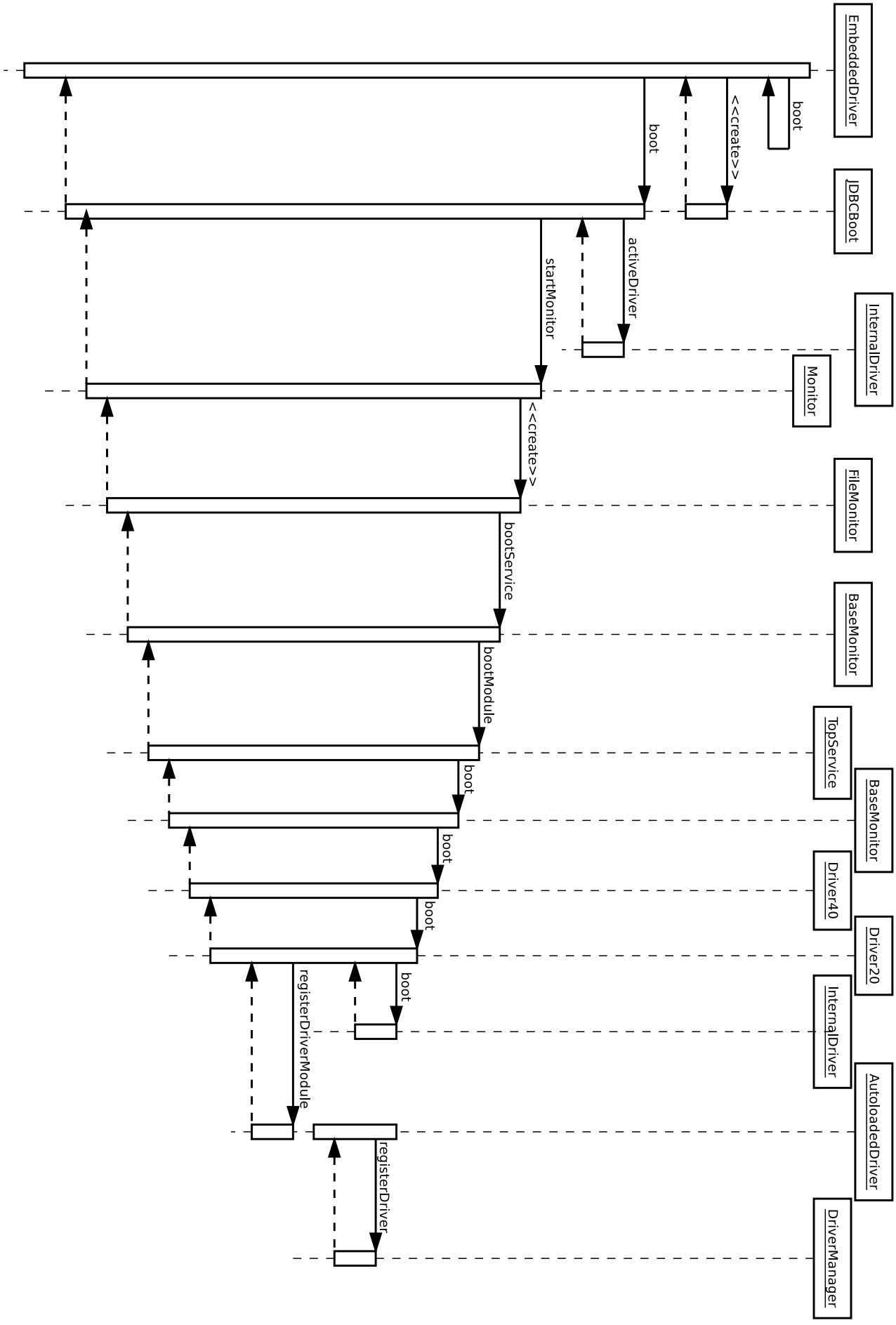

Abbildung 2.2: UML-Sequenzdiagramm: Ladevorgang eines JDBC-Treibers

Der Monitor erstellt eine Instanz von `FileMonitor`, welche wiederum die Methode `bootService` seiner Basisklasse `BaseMonitor` aufruft, um einen Service zu starten. `BaseMonitor` ruft die Methode `bootModule` der Klasse `TopService` auf, welche ihrerseits wieder die Methode `boot` der Klasse `BaseMonitor` aufruft. In dieser Methode wird nun der Treiber veranlasst zu booten. Dies geschieht über die Methode `boot` des `Driver40`, welche an die gleichnamige Methode der Super-Klasse `Driver20` delegiert wird. `Driver20` delegiert den Aufruf weiter an `InternalDriver`, welcher die Klasse `AutoloadedDriver` veranlasst, ihren statischen Klasseninitialisierer auszuführen. Dieser hinterlegt eine Instanz seiner Klasse mittels `DriverManager.registerDriver` beim Treiber-Manager.

Zurück in der Methode `boot` von `Driver20`, ruft diese die Methode `registerDriverModule` von `AutoloadedDriver` auf und übergibt eine Refrenz auf sich selbst. `AutoloadedDriver` merkt sich diese Referenz auf `Driver20` in seiner Klassenvariable `_driverModule`. Über die `AutoloadedDriver`-Instanz, die beim `DriverManager` registriert ist, kann natürlich auf die entsprechende Klassenvariable, und damit auf den Treiber, zugegriffen werden. Hiermit ist der Treiber erfolgreich geladen, beim Treiber-Manager registriert und kann verwendet werden.

2.2 Ausführung einer SQL-Anweisung

Wie in Kapitel 1 bereits erwähnt, unterscheidet Apache Derby zwei Betriebsmodi: *Embedded* und *Client/Server*. Ersterer wird benutzt, wenn Derby im gleichen Prozess läuft wie das Anwendungsprogramm. Zweiteres bedeutet, dass Derby und das Anwendungsprogramm in zwei unterschiedlichen Prozessen laufen und die Verbindung über ein Netzwerkprotokoll namens DRDA abgewickelt wird. DRDA steht für *Distributed Relational Database Architecture* und wurde von IBM entwickelt. Durch die konzeptionelle Nähe zu IBM DB2, verwendet auch Derby dieses Protokoll.

Die in diesem Kapitel behandelten Aspekte beziehen sich ausschließlich auf den Embedded-Modus, da alles Weitere den Rahmen dieses Dokuments sprengen würde. Das in Listing 2.1 dargestellte Programmstück wird dabei als Grundlage für die Analyse genommen. Im dargestellten Programmstück wird eine Verbindung zur Datenbank aufgebaut, ein SQL-SELECT-Statement ausgeführt und die Ergebnisse ausgegeben. Im hier beschriebenen Beispiel geht es nur darum, wie eine SQL-Anweisung an das DBMS gesendet wird und wie die Daten zurückkommen. Der genaue Prozess der Anfragebearbeitung und Optimierung wird in Kapitel 3 geschildert.

```
1  ...
2  java.sql.Connection conn;
3  java.sql.Statement s;
4  java.sql.ResultSet rs;
5
6  Properties props = new Properties(); // Properties für die Verbindung
7  props.put("user", "myuser1");
8  props.put("password", "mypassword1");
9
10 String dbname = "derbyDB";
11
```

```
12 /* 1. Verbindung erstellen */
13 conn = DriverManager.getConnection(
14     "jdbc:derby:" + dbname,
15     props);   // Benutzer, Passwort, ...
16
17 /* 2. Statement erstellen */
18 s = conn.createStatement();
19
20 /* 3. Query absetzen */
21 rs = s.execute("SELECT num, addr FROM location");
22
23 /* 4. Ergebnisse ausgeben */
24 System.out.println("Results:");
25 while (rs.next()) {
26   System.out.println(rs.getInt(1) + " " + rs.getString(2);
27 }
28
29 /* 5. Verbindung schließen */
30 conn.commit();   // vorher noch committen
31 conn.close();
32 ...
```

Listing 2.1: Programmstück für JDBC-Analyse

2.2.1 Erstellen einer Verbindung

In diesem Unterabschnitt wird beschrieben, wie eine Verbindung (`Connection`- Objekt) in Derby erstellt wird. In Abbildung 2.2.1 ist ein UML-Sequenzdiagramm zu sehen, welches den Ablauf verdeutlicht.

Das Erstellen einer Verbindung erfolgt über den Aufruf von:

```
DriverManager.getConnection();
```

Zurückgeliefert wird ein Objekt, dessen Klasse das Interface `java.sql.Connection` realisiert, in unserem Fall `EmbedConnection40`. Der Treiber-Manager durchläuft nun alle bei ihm registrierten Treiber und versucht eine Verbindung mit der übergebenen Verbindungs-URL – hier `jdbc:derby:derbyDB` – aufzubauen. Sobald ein Treiber eine gültige Verbindung, d. h. `!= null`, zurückliefert, wird diese an den Aufrufer zurückgegeben. Passt die Verbindungs-URL nicht zum Treiber, gibt dieser `null` zurück. Sind alle Treiber durchlaufen und es konnte keine gültige Verbindung aufgebaut werden, wirft der Treiber-Manager eine Exception. Im Folgenden wird der Ablauf beschrieben, der zutrifft, wenn der erste Treiber der passende ist.

Der Treiber-Manager ruft die Methode `connect` auf der ersten Treiber-Instanz auf (Typ `AutoloadedDriver`). Diese Methode wiederum fragt beim `InternalDriver` nach, ob die übergebene Verbindungs-URL überhaupt korrekt ist (Methode `embeddedDriverAcceptsURL`) und ob Derby damit umgehen kann. Dies dient

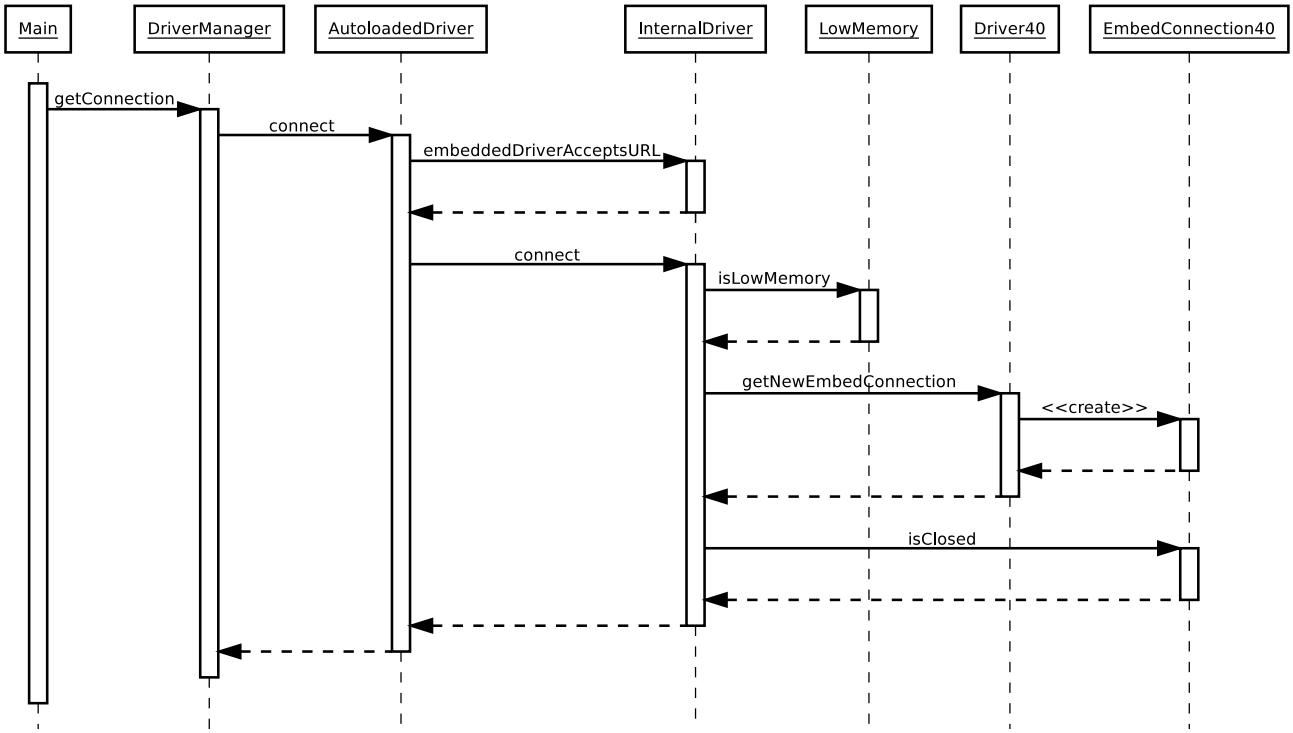

Abbildung 2.3: UML-Sequenzdiagramm: Verbindung erstellen

der Vorbeugung, dass Derby versehentlich gestartet wird, obwohl die URL syntaktisch garnicht korrekt ist. Ist die URL korrekt, wird der Aufruf an die Methode `connect` von `InternalDriver` weitergeleitet, welche prüft, ob überhaupt genügend Platz im Arbeitsspeicher vorhanden ist, um (noch) eine Verbindung zu erstellen: `EmbedConnection.memoryState.isLowMemory()`. Ist dies *nicht* der Fall, wird mit einer Exception abgebrochen.

Ist jedoch genügend Arbeitsspeicher vorhanden, wird die beim `AutoLoadedDriver` registrierte `Driver40`-Instanz aufgefordert, eine Verbindung zurückzugeben (Methode `getNewEmbedConnection`). In dieser Methode wird ein Objekt vom Typ `EmbedConnection40` mit der URL als Konstruktor-Argument erzeugt und zurückgegeben. Der `InternalDriver` prüft nun, ob die Verbindung erfolgreich erstellt werden konnte. Das ist daran zu erkennen, ob die Verbindung im Zustand „closed“ ist oder nicht. Programmtechnisch kann dies mit der Methode `isClosed` des `EmbedConnection40`-Objekts überprüft werden. Ist die Verbindung im Zustand „closed“, gibt der `InternalDriver` an den Treiber-Manager `null` zurück. Damit weiß dieser, dass der gerade geprüfte Treiber nicht der passende für die URL war und fährt, wie oben beschrieben, mit der Prüfung beim nächsten registrierten Treiber fort, sofern vorhanden. Ist die Verbindung *nicht* im Zustand „closed“, handelt es sich um eine gültige Verbindung und der Treiber-Manager kann diese an den Aufrufer zurückgeben.

2.2.2 Statement erstellen

Die Schritte, in denen ein Statement erstellt wird, sind in Abbildung 2.2.2 in Form eines UML-Sequenzdiagramms zu sehen. Im Folgenden wird der Ablauf jedoch detailliert beschrieben.

Ein Statement wird durch den Aufruf von `conn.createStatement()` erstellt (`conn` vom konkreten Typ `EmbedConnection40`). Als Rückgabe erhält man ein Objekt vom Typ `EmbedStatement40`, welches indirekt `java.sql.Statement` realisiert (vgl. Abbildung 2.2.2). Der Aufruf von `createStatement` auf der `EmbedConnection40`-Instanz wird an dessen Super-Klasse `EmbedConnection` weitergeleitet. In `EmbedConnection` existieren drei Überladungen der Methode `createStatement`. Die Parameter dieser betreffen allesamt die Art des Cursors für die resultierende Ergebnismenge (`ResultSet`). Der von uns getätigte, parameterlose Aufruf der Methode wird intern an die Parameter-behaftete Überladung

```
createStatement(
    int resultSetType,
    int resultSetConcurrency,
    int resultSetHoldability
)
```

mit folgenden aktuellen Parametern weitergeleitet:

`TYPE_FORWARD_ONLY`	Cursor kann auf der Ergebnismenge nur vorwärts bewegt werden.
`CONCUR_READ_ONLY`	Cursor kann keine Werte der Ergebnismenge verändern.
`HOLD_CURSORS_OVER_COMMIT`	Nach dem Commit der Transaktion bleibt der Cursor weiterhin bestehen.

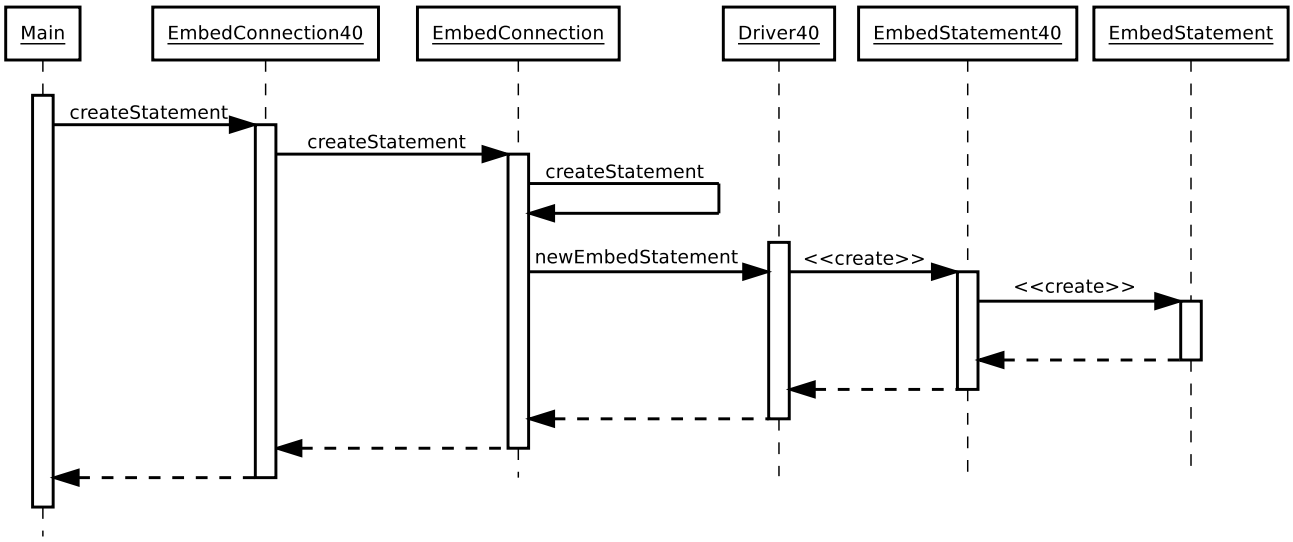

Abbildung 2.4: UML-Sequenzdiagramm: Statement erstellen

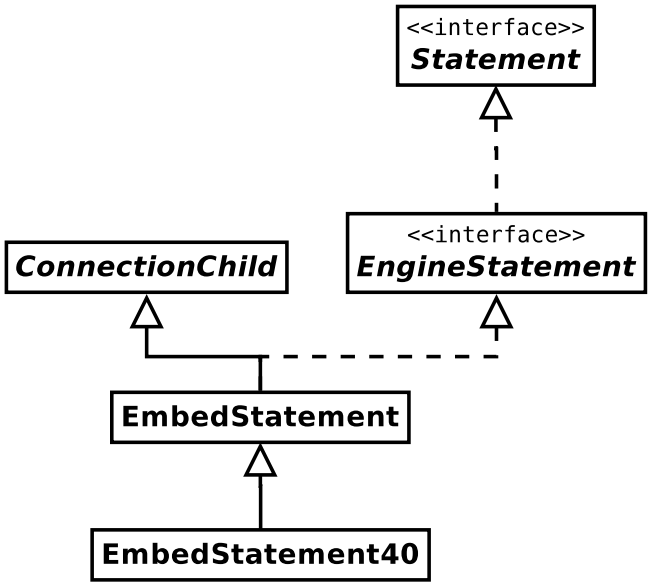

Abbildung 2.5: Typ-Hierarchie der Klasse `EmbedStatement40`

Das `EmbedConnection`-Objekt besitzt eine Instanzvariable namens `factory` vom abstrakten Klassen-Typ `InternalDriver`. `factory` hält eine Referenz auf die konkrete Implementierung `Driver40` (vgl. Abbildung 2.2.2). Die Methode `createStatement` der Klas-

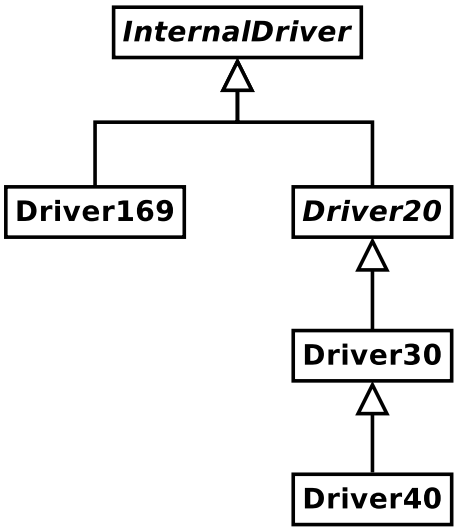

Abbildung 2.6: Typ-Hierarchie des Interface `InternalDriver`

se `EmbedConnection`, in der wir uns immernoch befinden, tätigt nun folgenden Aufruf: `factory.newEmbedStatement()`. Dabei werden Parameter betreffend den Cursor-Typ und eine Referenz auf die aktuelle Verbindung (Typ: `EmbedConnection`) übergeben. Die Methode `newEmbedStatement` erstellt ein Objekt vom Typ `EmbedStatement40`, welches den Konstruktoraufruf an seine Super-Klasse `EmbedStatement` delegiert. `EmbedStatement` realisiert, wie in Abbildung 2.2.2 zu sehen, indirekt das Interface `java.sql.Statement`. Schlussendlich gibt die Methode `newEmbedStatement` das soeben erstellte Objekt vom Typ `EmbedStatement40` zurück.

Anmerkung

Apache Derby unterstützt keine sensitiven Cursor! Vor dem Aufruf der Methode `factory.newEmbedStatement` wird deshalb geprüft, ob der Cursor-Typ auf „sensitiv“ (`ResultSet.TYPE_SCROLL_SENSITIVE`) gestellt ist. Ist dies der Fall, gibt Derby eine Warnung aus, dass dies nicht unterstützt wird und setzt den Cursor-Typ auf „in-sensitiv“ (`ResultSet.TYPE_SCROLL_INSENSITIVE`).

2.2.3 Ein SQL-Query ausführen

Abbildung 2.2.3 zeigt ein UML-Sequenzdiagramm, in dem der Ablauf beim Ausführen eines SQL-Querys in Derby dargestellt wird. Zum Ausführen wird die Methode `s.executeQuery` aufgerufen mit dem entsprechenden SQL-String als Parameter. Dabei ist `s` vom Typ `EmbedStatement`. In unserem Fall ist das SQL-Statement eine einfache Abfrage (vgl. Listing 2.1):

```
SELECT num, addr FROM location
```

Die Methode `executeQuery` wird intern auf die Methode `execute` weitergeleitet, die einige Parameter entgegennimmt, welche im Folgenden beschrieben sind:

Formaler Parameter	Typ	Aktueller Parameter
sql	String	SELECT num, addr FROM location
executeQuery	boolean	true
executeUpdate	boolean	false
autoGeneratedKeys	int	Statement.NO_GENERATED_KEYS
columnIndexes	int[]	null
columnNames	String[]	null

Beim Aufruf dieser Methode muss entweder der Parameter `executeQuery` auf `true` stehen und der Parameter `executeUpdate` auf `false` oder umgekehrt, je nachdem ob es sich um ein modifizierendes Statement (`UPDATE`, `DELETE`, `INSERT`, ...) oder ein rein lesendes (`SELECT`) handelt. In unserem Fall trifft letzteres zu. `sql` ist das auszuführende SQL-Statement. `autoGeneratedKeys` ist auf `Statement.NO_GENERATED_KEYS` gesetzt und gibt an, dass automatisch generierte Schlüssel nicht für die Abfrage bereit gestellt werden. Die letzten beiden Parameter spielen in unserem Fall keine Rolle und werden hier nicht weiter behandelt.

Die Methode `execute` stellt zunächst mit dem Schlüsselwort `synchronized` sicher, dass alle Zugriffe auf das `Connection`-Objekt (konkreter Typ `EmbedConnection40`) unter gegenseitigem Ausschluss stattfinden. Danach werden die Ergebnisse der letzten Abfrage gelöscht, falls welche vorhanden sind (Aufruf der Methode `clearResultSets`). Mit dem darauf folgenden Aufruf von `setupContextStack` wird sicher gestellt, dass ein Kontext-Manager (alias Kontext-Stack) vorhanden ist, woraufhin der aktuelle Thread mit dem Kontext-Manager verlinkt wird. Zum Kontext-Manager gehört unter anderem der aktuelle Transaktionskontext. Transaktionen und deren Verwaltung werden in Kapitel 5 genauer besprochen, sodass an dieser Stelle nicht näher darauf eingegangen wird.

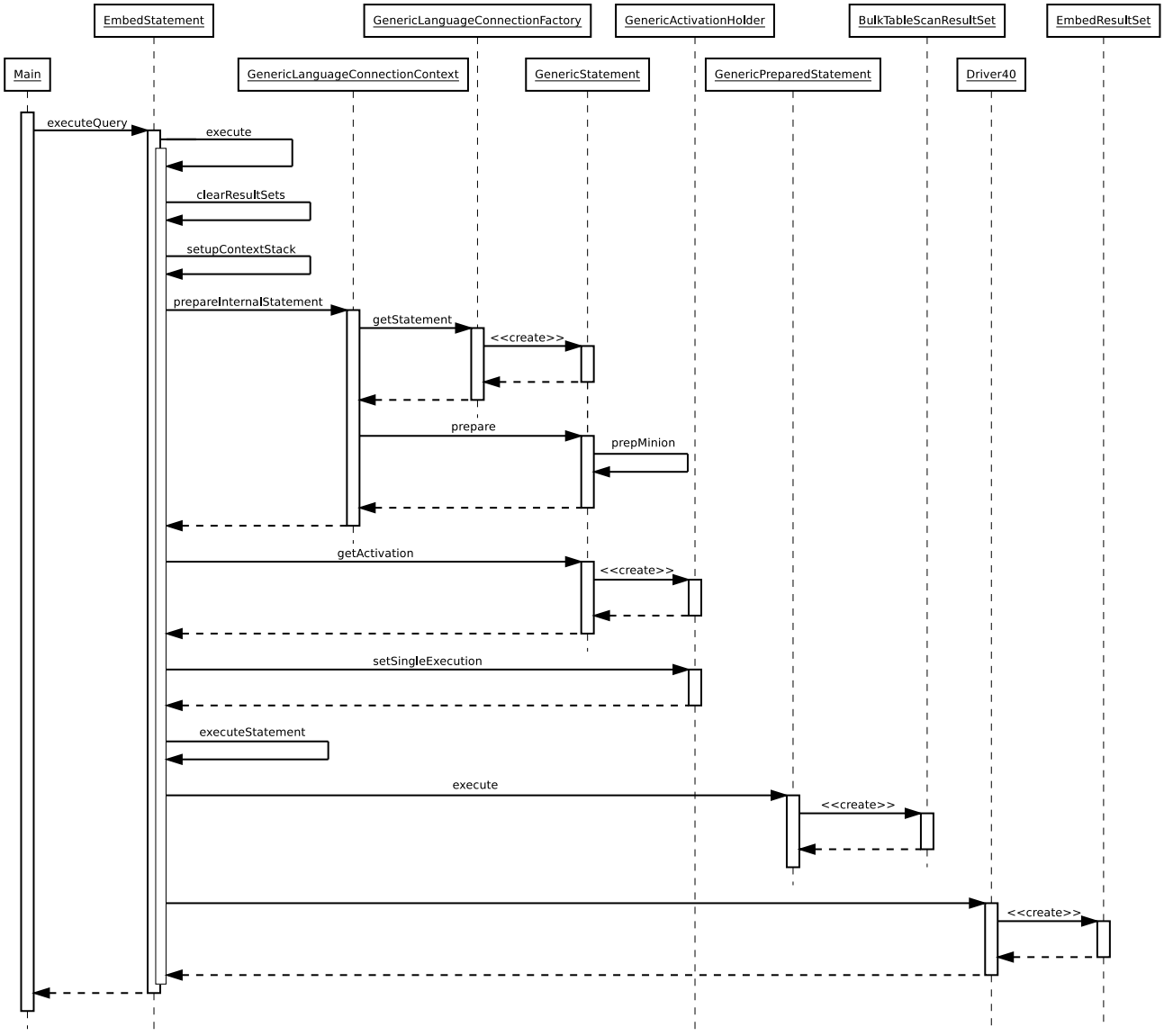

Abbildung 2.7: UML-Sequenzdiagramm: SQL-Query ausführen

Obwohl wir gerade im Begriff sind, ein „normales“ Statement und *kein* Prepared-Statement auszuführen, wird im nächsten Schritt ein Prepared-Statement erstellt. Dies macht insoweit Sinn, dass, egal für welche Art von Statement, sowieso ein binärer Zugriffsplan erzeugt werden muss. Bei einem Prepared-Statement wird dieser, im Gegensatz zu einem „normalen“ Statement, jedoch aufgehoben und nicht nach Ende der Ausführung bzw. Transaktion gelöscht.

Die Methode, die ein Prepared-Statement erzeugt, heißt `prepareInternalStatement` und wird auf einer Instanz von `GenericLanguageConnectionContext` aufgerufen. `GenericLanguageConnectionContext` ist verantwortlich für die Verwaltung des Pools von Prepared-Statements, Aktivierungen und gerade in Benutzung befindlichen Cursorn der Verbindung. Innerhalb von `prepareInternalStatement` wird mit Hilfe der Factory `GenericLanguageConnectionFactory` eine `GenericStatement`-Instanz erstellt (Methode `getStatement`) und auf dieser Instanz `prepare` aufgerufen. `prepare` delegiert seine komplette Arbeit an die Methode `prepMinion`, welche nun endlich den ersehnten binären Zugriffsplan erstellt und zurückgibt, jedoch *nicht* ausführt. Die Genrierung des Zugriffsplans wird hier nicht näher erläutert; Kapitel 3 widmet sich diesem Sachverhalt.

Zurück in der Methode `execute` von `EmbedStatement`, wird als nächstes eine „Aktivierung“ (Interface-Typ `Activation`) erzeugt. Diese bekommt man vom `GenericStatement`-Objekt mittels der Methode `getActivation`. Sie liefert als konkrete `Activation`-Implementierung `GenericActivationHolder` zurück. Eine Aktivierung hat den Sinn, einen Zugriffsplan bzw. ein Prepared-Statement auszuführen. Das bedeutet, dass mehr als eine Aktivierung desselben Prepared-Statements zur gleichen Zeit aktiv sein kann. Es existiert eine enge Kopplung zwischen Aktivierungen und einem Prepared-Statement. Eine Aktivierung hat immer eine Referenz auf das Prepared-Statement, von dem sie erzeugt wurde. Ein Prepared-Statement seinerseits führt Buch über all seine aktiven Aktivierungen.

Im nächsten Schritt wird die Aktivierung auf „single execution“ gesetzt (Methode `setSingleExecution`). Der Grund dafür ist, dass es sich hier um ein normales Statement handelt, welches nur ein einziges Mal ausgeführt wird, und nicht um ein Prepared-Statement, welches mehrmals ausgeführt werden kann. Daraufhin wird die Methode `executeStatement` auf der `EmbedStatement`-Instanz aufgerufen und die Aktivierung als Parameter übermittelt. Innerhalb der Methode `executeStatement` wird die Methode `execute` auf einer `GenericPreparedStatement`-Instanz aufgerufen, welche schlussendlich eine Ergebnismenge liefert. Die Ergebnismenge ist vom Typ `BulkTableScanResultSet`, welcher das Interface `java.sql.ResultSet` realisiert. Die Ergebnismenge wird mittels der Methode `newEmbedResultSet` einer `Driver40`-Instanz in ein `EmbedResultSet`-Objekt verpackt („ge-wrappt“) und zurückgegeben.

Das zurückgegebene `EmbedResultSet`-Objekt hat intern eine Variable namens `theResults` vom Typ `BulkTableScanResultSet`. Dies zeigt deutlich, dass die Klasse `newEmbedResultSet` nur ein Wrapper um `BulkTableScanResultSet` ist.

2.2.4 Die Ergebnismenge und deren Verarbeitung

Im `EmbedResultSet`-Objekt existiert eine Variable namens `resultDescription`, die ein Objekt vom Typ `GenericResultDescription` referenziert. Zum aktuellen Zeit-

punkt, das heißt vor einem Aufruf von `rs.next()`, beinhaltet die Variable schon Meta-Informationen über die Ergebnismenge. Abbildung 2.2.4 zeigt einen Screenshot mit den aktuellen Werten dieser Instanz.

resultDescription	GenericResultDescription (id=143)
columnNameMap	null
columns	ResultColumnDescriptor[2] (id=151)
[0]	GenericColumnDescriptor (id=157)
columnPos	1
hasGenerationClause	false
isAutoincrement	false
name	"NUM" (id=162)
schemaName	"MYUSER1" (id=163)
tableName	"LOCATION" (id=164)
type	DataTypeDescriptor (id=165)
updatableByCursor	false
[1]	GenericColumnDescriptor (id=159)
columnPos	2
hasGenerationClause	false
isAutoincrement	false
name	"ADDR" (id=173)
schemaName	"MYUSER1" (id=163)
tableName	"LOCATION" (id=164)
type	DataTypeDescriptor (id=175)
updatableByCursor	false
metaData	null
statementType	"SELECT" (id=152)

Abbildung 2.8: Screenshot: Inhalt der Variable `resultDescription`

Die Variable `theResults` des `EmbedResultSet`-Objekts referenziert, wie bereits erwähnt, eine Instanz vom Typ `BulkTableScanResultSet`. Zum aktuellen Zeitpunkt existieren darin keinerlei Datensätze aus der Datenbank. Allerdings ist eine Variable `rowArray` vorhanden, die ein leeres, zweidimensionales Array repräsentiert, das Platz für 16 Datensätze bietet: `DataValueDescriptor[16][]`.

Beim ersten Aufruf von `rs.next()`, d. h. beim Zugriff auf den ersten Datensatz durch den Benutzer, werden gleich 16 Datensätze, sofern vorhanden, aus der Datenbank in das `DataValueDescriptor`-Array gelesen. Damit sind die vorher freien Plätze nun komplett belegt. Für die nächsten 15 Aufrufe von `rs.next()` müssen nun keine Daten aus der Datenbank geholt werden, da diese entsprechend gepuffert sind. Erst der 16. Aufruf von `rs.next()` veranlasst das Nachladen weiterer 16 Datensätze, sofern vorhanden.

Das Puffern der Daten geschieht in Derby damit sehr statisch. Es werden immer genau 16 Datensätze (sofern vorhanden) aus der Datenbank geholt und zwischengespeichert. Man hätte an dieser Stelle erwarten können, dass das Puffern/Cachen der Datensätze durch einen intelligenten Algorithmus geschieht, der z. B. je nach Größe der Datensätze oder der Häufigkeit des Zugriffs (Aufruf der Methode `rs.next()`) mal mehr und mal weniger Daten von der Datenbank abfragt und zwischenspeichert. Auf der anderen Seite gibt man dem Benutzer des `ResultSet` mit dieser statischen, auf 16 Datensätze beschränkten Zwischenspeicherungsstrategie die Möglichkeit, eigene, mehr oder weniger intelligente, Caching-Lösungen zu entwickeln.

In diesem Kapitel haben wir gesehen, wie Derby bestimmte JDBC-Funktionalität implementiert. Vom Laden eines Treibers, über das Erstellen einer Verbindung und eines Statements, bis hin zum Ausführen einer SQL-Anweisung und der Navigation über die Ergebnismenge. Damit wurden große Teile der internen Verarbeitung durch Apache Derby bis zu einem gewissen Detailliertheitsgrad abgedeckt. Das Parsen und Optimieren einer SQL-Anweisung zum Beispiel, bis hin zum Erstellen eines Zugriffsplans, wurde jedoch nicht näher beschrieben. Diesem Thema widmet sich das nächste Kapitel.

3 Anfragebearbeitung

In diesem Kapitel wird behandelt, wie SQL-Anfragen von der Datenbank verarbeitet werden. Der Ablauf ist grundsätzlich in zwei Abschnitte aufgeteilt. Im ersten Abschnitt wird das abgesetzte SQL-Statement soweit verarbeitet, dass es ausgeführt werden kann. Dieser Schritt beinhaltet dann zunächst das Parsen. Dabei wird ein Baum erstellt, der das SQL-Statement repräsentiert. Anschließend wird dieses Statement validiert, d.h. auf korrekte Rechte des Nutzers und auf plausible Tabellen und Spaltennamen überprüft. Nach dem Validieren wird der valide Baum an einen Optimizer geschickt, welcher den Baum so umbaut, dass dieser möglichst optimal aufgebaut ist. Optimal heißt, dass das Statement möglichst schnell abgearbeitet werden kann. Mit dem optimierten Baum wird dann ein Zugriffsplan erstellt, der den Zugriff auf die Daten regelt. Der fertige Zugriffsplan wird in der Klasse preparedStatement abgelegt.

Dieses Statement wird dann im zweiten Schritt ausgeführt. Dazu wird der Zugriffsplan aufgerufen und dessen Befehle abgearbeitet.

Im Folgenden werden die einzelnen Schritte (Parsen, Validieren, Optimieren und Zugriffsplan erstellen) beschrieben.

3.1 Parsen

3.1.1 Aufbau des Parsers

Der Parser ist in Derby sehr erweiterbar aufgebaut. Grundsätzlich gibt es die Klasse Parser, welche wiederum auf die Klasse `SQLParser` zugreift. SQLParser ist jedoch nicht direkt in Derby als Klasse implementiert. Die Klasse `SQLParser` wird durch eine javacc-Datei definiert und zur Laufzeit generiert. Javacc ist der JavaCompiler-Compiler. Dieses Modul erzeugt aus einer Steuerdatei (hier: `SQLParser.jj`) meherere für den SQL-Parser nötige Klassen. Diese sind: `SQLParser`, `SQLParserTokenManager`, `SQLParserConstants`, `TokenMgrError`. Diese vier Klassen sind für das Parsen des Statements verantwortlich. Aufgrund der immensen Größe dieser Klassen wird auf ein UML-Diagramm verzichtet. Kurz gesagt beinhalten sie jegliches Syntaxwort der SQL-Sprache und können daraus dann Objekte, die Derby verarbeiten kann erstellen.

SQLParserConstants In der Klasse `SQLParserConstants` sind für sämtliche Tokens von SQL die Konstanten, die diese im Quelltext repräsentieren, abgelegt.

SQLParser Dies ist die Hauptklasse, die die Funktionalität des Parsers bereitstellt. Mit den anderen Hilfsklassen erzeugt sie den Parse-Baum des Statements.

SQLParserTokenManager Diese Klasse analysiert das Statement lexikalisch.

TokenMgrError Diese Klasse repräsentiert Konstanten für lexikalische Fehler und gibt Fehlermeldungen aus.

3.1.2 Ablauf des Parsens

Der Parser geht eine ankommende SQL-Anfrage Stück für Stück durch und erstellt einen Baum, der den Aufbau des Statements repräsentiert. Dieser Baum ist nicht optimiert oder validiert. Er ist nur auf korrekte Syntax geprüft. In Abbildung 3.1 wird der Aufbau eines solchen Baumes anhand des SELECT-Statements `SELECT num,addr FROM location ORDER BY num` dargestellt. Diese Abbildung wurde mit Hilfe von Debug Anweisungen erstellt, die Derby als Ausgabe in einer .log-Datei speichert. In Listing 3.1 ist der Teil dieser .log-Datei dargestellt, welcher den Optimierungsprozess betrifft.

Ein Baum besteht aus mehreren Knoten (Nodes),welche einen Teil eines Statements repräsentierten . Im dargestellten Beispiel gibt es einen `CursorNode`, welcher das Wurzelelement eines Select-, oder Update-Statements darstellt. Danach werden je nach SQL-Anfrage Kindknoten erzeugt, welche zum Beispiel die Spalten des Statements oder die Tabellen, von denen gelesen wird repräsentieren. Es gibt für jede Art von Befehl einen Knoten, der in Derby implementiert ist. Für variabel benennbare Elemente, wie z.B. Tabellennamen, gibt es ebenfalls entsprechenden Klassen (z.B. `TableName`). In Objekten dieser Klassen werden dann die Informationen, die das Statement beschreiben abgelegt. Wie die Abbildung zeigt wird auch für das `Order by` Element eine `OrderByList` erzeugt, welche die Liste der Spalten enthält nach denen sortiert wird. Diese wird dann in weiteren Schritten bearbeitet.

Nach dem der initiale Baum erstellt wurde wird dieser auf Korrektheit überprüft. Im nächsten Kapitel wird dieser Vorgang näher beschrieben.

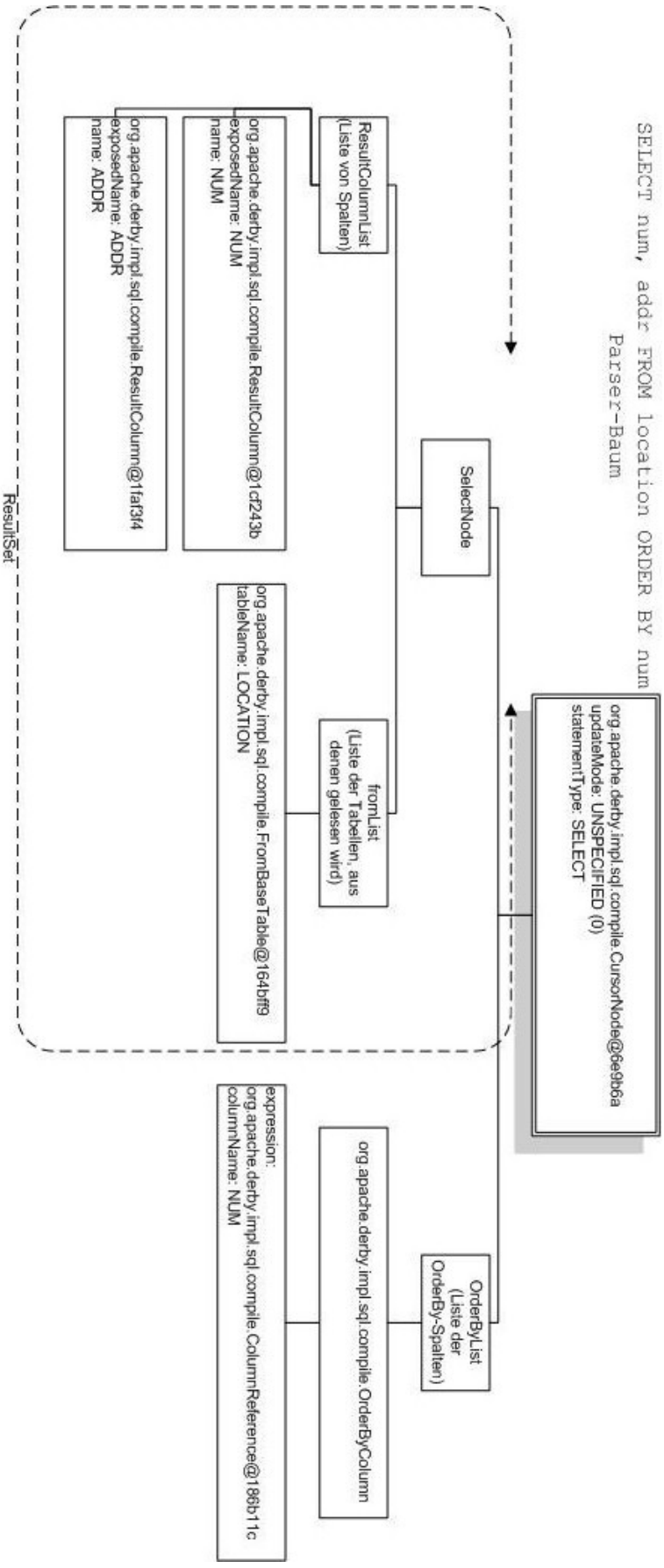

Abbildung 3.1: Darstellung des geparsten Baumes beim Select-Statement

3.2 Validieren

Nachdem das SQL-Statement zu einem Syntaxbaum zerlegt wurde, muss dieser nun einer semantischen Prüfung unterzogen werden. Beim validieren der SQL-Anfrage wird es unter anderem auf korrekte Rechte des Nutzers und auf korrekte Bezeichner (Spaltennamen usw. geprüft). Der Aufruf innerhalb des Programmcodes erfolgt in der Klasse `GenericStatement` mittels `qt.bindStatement()`. Jedes Statement wird als `StatementNode` repräsentiert und erst zur Laufzeit fest an ein von `StatementNode` abgeleitetes Objekt gebunden. Das Objekt qt steht demnach für den entsprechenden Node des Statements. Bei einer Select-Anweisung handelt es sich um einen CursorNode, dessen Methode `bindStatement()` aufgerufen wird. Danach erfolgen spezifische, für die Anfrage notwendige Überprüfungen. Damit der Baum validiert werden kann, wird ein Schema-Katalog verwendet, welcher alle wichtigen Informationen zum Datenbankschema, wie Tabellenbeschreibung, Beschreibung von Views oder Spalten, Indizierung oder Rechte enthält. Im Folgenden wird der Ablauf der Validierung unter Berücksichtigung der genannten Punkte beschrieben.

3.2.1 Ablauf der Validierung

Der valide Syntaxbaum zu dem beim Parsen genannten `Select`-Statement kann wie in Abbildung 3.2 beschrieben dargestellt werden. Alle Operationen zur Überprüfung der Korrektheit des Baumes sind abgeschlossen. Bevor es jedoch dazu kommt werden nacheinander folgende Schritte zum Validieren des Baumes vollzogen, wobei diese Aufzählung nicht der tatsächlichen Ausführreihenfolge entspricht:

Korrektheit der Tabellen und Spalten prüfen: Mit Hilfe des Schemakataloges in dem alle Informationen zu der Datenbank und den enthaltenen Tabellen gespeichert sind, wird überprüft, ob alle Spalten in der entsprechenden Tabelle auch wirklich existieren und die Datentypen übereinstimmen. Zudem werden die Namen der Tabellenspalten zu voll qualifizierten Namen der Tabellen zusammengefasst. Bei einem `Create`-Statement wird unter anderem noch geprüft, dass nur ein Primärschlüssel angegeben ist und die Anzahl der enthaltenen Tabellen in der Datenbank nicht überschritten wird.

Auflösen des * Operators: Wird zum Beispiel in einer `Select`-Anfrage der * Operator verwendet, wie zum Beispiel `Select * from my-db`, muss das * aufgelöst werden. Die entsprechenden Tabellenspalten werden dann an die Datenbank gebunden.

Abhängigkeiten, Indexe und Trigger prüfen: Tabellenspalten können bestimmten Bedingungen und Abhängigkeiten unterliegen, welche während der Validierung geprüft werden. Diese Abhängigkeiten sind im Datenbankschema definiert. Es handelt sich dabei zum Beispiel um die `not null` Eigenschaft einer Spalte. Diese Eigenschaft wird überprüft und mit an den Syntaxbaum gebunden. Ähnlich ist es bei Triggern und Indexen. Diese werden während der Validierung überprüft und dann an die entsprechende Stelle im Syntaxbaum des Statements angehängt.

Zugriffsrechte prüfen: Bei `SQL`-Anfragen können Daten verändert oder auch neue Tabellen angelegt werden. Beim Validieren wird sichergestellt, dass zum Beispiel bei einem

`Insert`-Statement auf die Datenbank auch entsprechende Rechte für die Datenmanipulation bestehen. Je nach Anfrage können unterschiedliche Privilegien notwendig sein.

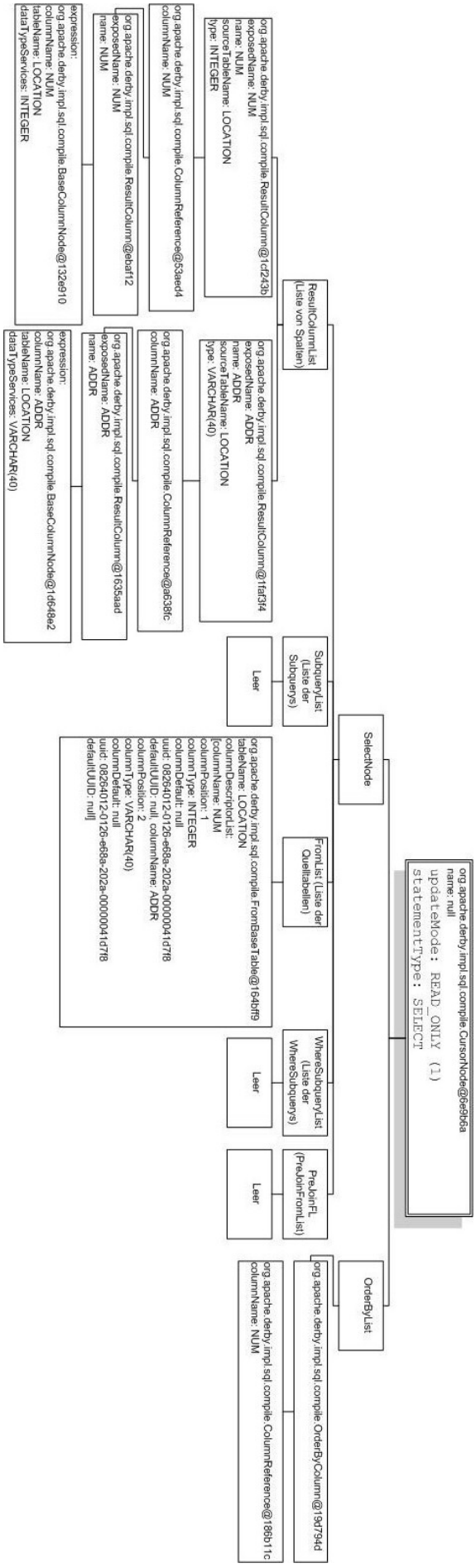

Abbildung 3.2: Darstellung des validen Baumes beim Select-Statement

3.3 Optimieren

3.3.1 Allgemeine Informationen zum Optimiervorgang

Die Optimierung legt einen validierten Syntaxbaum zu Grunde und sucht dann die optimale Ausführungsstrategie mittels kostenbasierter Optimierung. Bei diesem Prozess werden je nach `SQL`-Statement verschieden komplexe Strategien zur Verbesserung des Zugriffsplans verwendet. Bei einigen Statements ist keine Optimierung möglich. Hier wird die leere Methode `optimize()` implementiert. Ein Beispiel für eine solche Anfrage ist unter anderem die `Create Table` Anweisung. Andererseits können `Select`-Anweisungen JOINS über mehrere Tabellen mit einer `Where`-Bedingung verknüpfen, wodurch die Optimierung mittels Optimizerobjekt notwendig wird. Die Güte der optimierten Bäume wird mit Hilfe einer Kostenberechnung für den aktuellen Baum durchgeführt. Bei lang verzweigten Bäumen werden alle möglichen Permutationen bewertet und schließlich die kostengünstigste Anordnung des Baumes verwendet. Die best mögliche Anordnung von `Where`- oder `Having`-Klauseln im Syntaxbaum für die gewählte Permutation ist ebenso ein Teil der Optimierung.

Im Folgenden wird der Ablauf der Optimierung im Detail beschrieben. Dabei werden folgende Fragen beantwortet:

- Welche Klassen sind in Derby an der Optimierung einzelner Anfragen beteiligt?
- Welche Unterschiede bei der Optimierung gibt es abhängig von der SQL-Anfrage?
- Wie sucht der Optimierer den besten Ausführungsplan?
- Wie werden die Kosten bewertet?
- Wie sieht der optimierte Zugriffsplan aus?

3.3.2 Ablauf der Optimierung

Der Ablauf der Optimierung wird anhand des Sequenzdiagramms in Abbildung 3.4 beschrieben. Zu jedem `SQL`-Statement in der Klasse `GenericStatement` wird ausgewertet, um welche Art von Anfrage es sich handelt. Auf diesem speziellen `StatementNode` werden dann die Methoden aufgerufen, welche für eine Optimierung verwendet werden. Im Diagramm wurde dies anhand eines `Select`-Statement visualisiert.

Grundsätzlich sind Strategien zur Optimierung nur bei Anfragen zur Datenmanipulation notwendig. Bei der Datendefinition wird keine Optimierung vorgenommen. Das `Create Table`- oder `Alter Table`-Statement definiert Tabellen und deren Spalten, durchläuft demnach keine Optimierung. `Select` oder `Update Table` hingegen manipulieren die Daten in einer Datenbank und werden optimiert. Die Klassen, welche eine Optimierung durchführen, müssen die Methode `optimizeStatement()` der Klasse `StatementNode` überschreiben. Bei einem `Create Table`-Statement wird die genannte Methode nicht überschrieben und stattdessen die unimplementierte Methode des `StatementNode` aufgerufen. Auf diese Weise wird keine Optimierung durchgeführt.

Der Optimierungsvorgang wird mit der Methode optimize() gestartet. In Abbildung 3.4 wird deutlich, dass die Implementierung dessen für eine `Select`-Anfrage in `SelectNode`

geschieht. Es wird ein Objekt eines Optimizers erstellt und mit dessen Hilfe die kostengünstigste Permutation der Tabellen des JOIN für den Baum ermittelt. Wie zu Anfang schon erwähnt, handelt es sich um einen kostenbasierten Optimierungsprozess. Wenn also von kostengünstigster Permutation gesprochen wird, bedeutet dies einen gewissen Aufwand der Datenbank um diese Anordnung zu finden. In einem einzelnen `Select`-Statement können mehrere JOINS über Tabellen gemacht werden, welche mit `Where`-Bedingungen verknüpft werden. Im Optimizer werden die JOINS als Liste von Optimizables repräsentiert (Klasse `FromList`). Diese Liste stellt eine Permutation dar und wird dann durch vertauschen der einzelnen Optimizables innerhalb der Liste optimiert. Die Anordnung mit den geringsten Kosten repräsentiert den Optimierten Baum als ResultSetNode.

Mit den Methoden `getNextPermutation()` und `getNextDecoratedPermutation()` wird eine neue Anordnung der Optimizables ermittelt und die Kosten für diese Anordnung berechnet. Die zuerst genannte Methode ist vom Typ boolean, iteriert über die verschiedenen Permutationen und erstellt eine weitere Permutation, falls möglich. Sie liefert **`true`**, wenn es eine noch nicht getestete Anordnung gibt, sonst **`false`**. Ähnlich ist es bei der zweiten Methode. Diese iteriert über die möglichen Zugriffspfade im Baum für die derzeitige Anordnung der Optimizables. In einem mehrfach verzweigten Baum wäre es möglich auf mehreren Wegen zu einem Element des Baumes zu gelangen, wovon diese Methode den besten Weg sucht. Die Mehtode liefert **`true`**, wenn es einen Weg gibt, sonst **`false`**. Erst danach wird die so gefundene Permutation bewertet und geprüft ob es sich um eine günstigere Anordnung handelt, als die zuvor betrachteten. Beim Erstellen der Permutationen wird nach und nach geprüft, an welcher Position die Where-Bedingungen ausgewertet werden können. Das heißt, dass eine Bedingung "'früher"' ausgewertet werden kann, wenn die benötigten Tabellen in der Liste der Optimizables weiter vorne stehen. Ansonsten werden diese Bedingungen in der Hirarchie des Syntaxbaumes ganz nach unten verschoben. Ein Beispiel für einen schon optimierten Syntaxbaum für eine `Select`-Anfrage ist in Abbildung 3.3 veranschaulicht. Die Anfrage kann aus mehreren Klauseln und Subqueries bestehen, die im Laufe der Optimierung ebenfalls ausgewertet werden und ihren Platz im optimierten Syntaxbaum finden. So werden ähnlich wie `Where`-Bedingungen auch `Having` und `Order By` ausgewertet. Die Optimierung in Derby besteht folglich aus mehreren Teilen:

1. Suchen der besten Permutation
2. Anordnen der Klauseln (Where, Having, Order By)
3. Auswerten von Subqueries
4. Ablegen als Syntaxbaum

Die verwendete Traversierungsstrategie um den Baum zu durchlaufen ist die Tiefensuche. Da es sich um einen Left-Deep Tree handelt, der für die Optimierung verwendet wird, ist die Tiefensuche gut geeignet um schnell eine passende Anordnung zu haben, die gut in einer Liste umgesetzt werden kann.

Ein vergleichbarer Ablauf für den Optimierungsprozess ergibt sich zum Beispiel für das `Update`-Statement. Die Methodenaufrufe für eine Optimierung bleiben gleich, jedoch sind andere Klassen am Optimierungsprozess beteiligt. So wird beim Select-Statement zum Beispiel ein CursorNode verwendet, welches von `DMLStatementNode` abgeleitet ist. Beim Update hingegen wird die Klassen `DMLModStatementNode` verwendet, die von

`DMLStatementNode` abgeleitet ist. Danach wird in entsprechenden Klassen eine Optimierung unter Verwendung des Optimizers und Optimizables wie beschrieben durchgeführt. Die Klasse `DMLModStatement` wird von Statements verwendet, die eine Modifikation der Tabelle vornehmen (Insert, Update, Delete).

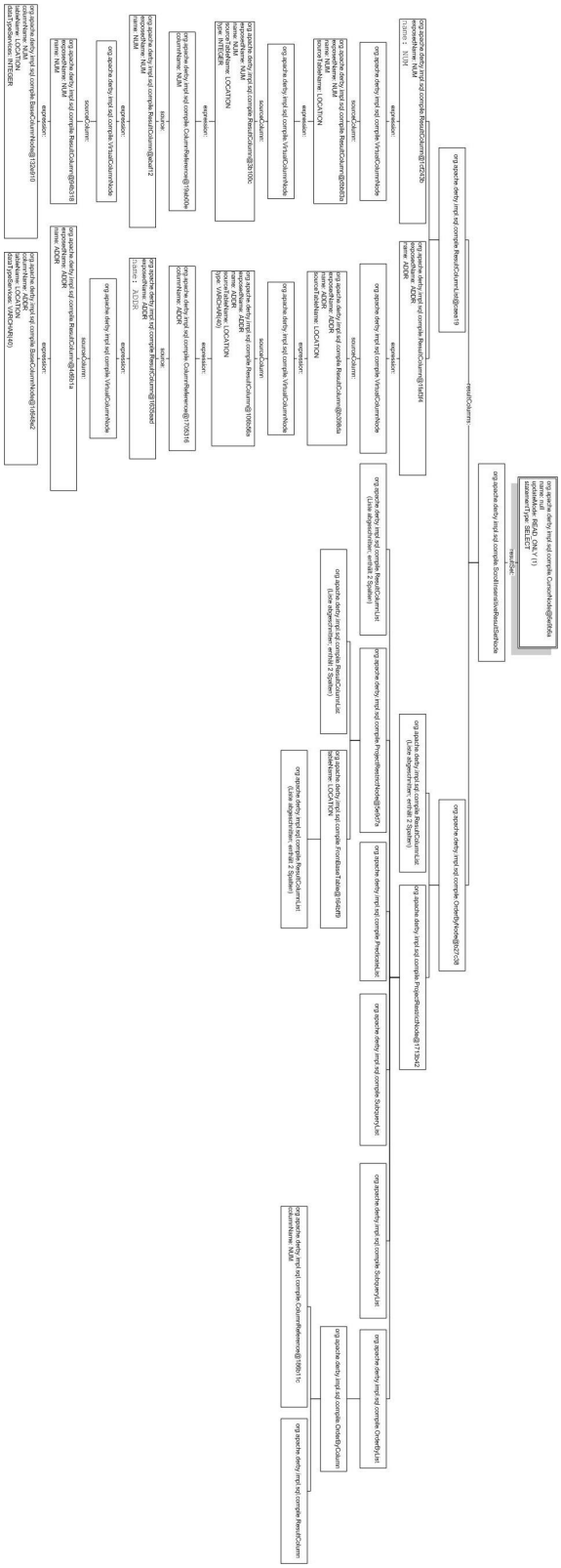

Abbildung 3.3: Darstellung des optimierten Baumes beim Select-Statement

Ausgabe des Optimizers in der Log-Datei

Während der Ausführung einer Anfrage ist kann in Derby in einer Log-Datei mitverfolgt werden, was im beim Optimierungsprozess vor sich geht. Im folgenden Listing ist die Ausgabe der Datei für die Optimierung angegeben. Diese Ausgabe ist unter anderem Grundlage des in Abbildung 3.3 dargestellten optimierten Syntaxbaum.

```
Anzahl der Öffnungsvorgänge = 1
Eingegebene Zeilen = 2
Zurückgegebene Zeilen = 2
Duplikate eliminieren = false
In sortierter Reihenfolge = false
Sortierungsinformationen:
  Anzahl der ausgegebenen Zeilen=2
  Anzahl der eingegebenen Zeilen=2
  Sortiertyp=innere
  Konstruktorzeit (Millisekunden) = 0
  Zeit für open (Millisekunden) = 0
  Zeit für next (Millisekunden) = 0
  Zeit für close (Millisekunden) = 0
  Vom Optimizer geschätzte Zeilenzahl:            7,00
  Vom Optimizer geschätzte Kosten:           59,56
Quellenergebnisliste:
  ResultSet der Tabellensuche für LOCATION bei Isolationsstufe
  READ COMMITED mit INSTANTANEOUS SHARE ROW-Sperrung
  wurde vom Optimizer ausgewählt
  Anzahl der Öffnungsvorgänge = 1
  Gesehene Zeilen = 2
  Gefilterte Zeilen = 0
  Abrufgröße = 16
    Konstruktorzeit (Millisekunden) = 0
    Zeit für open (Millisekunden) = 0
    Zeit für next (Millisekunden) = 0
    Zeit für close (Millisekunden) = 0
    Zeit für next in Millisekunden/Zeile = 0
  Suchinformationen:
    Anzahl der abgerufenen Spalten=2
    Anzahl der besuchten Seiten=1
    Anzahl der besuchten Zeilen=2
    Anzahl der qualifizierten Zeilen=2
    Bits der abgerufenen Spalten=Alle
    Suchtyp=Heap-Speicher
    Anfangspunkt:
      null
    Endpunkt:
      null
    Qualifikationsmerkmale:
      Keine
    Vom Optimizer geschätzte Zeilenzahl:            7,00
    Vom Optimizer geschätzte Kosten:           59,56
```

Listing 3.1: Inhalt der .log-Datei für den Optimizer

Es sind einige Informationen verfügbar. Zum einen werden die Kosten und die betroffenen Zeilen ausgegeben und zum anderen auch die Zugriffsmechanismen und Isolationsleveln unter denen diese Anfrage abläuft. Wie die Werte für die Kostenberechnung letztendlich zusammengesetzt sind wird im folgenden Kapitel näher beschrieben.

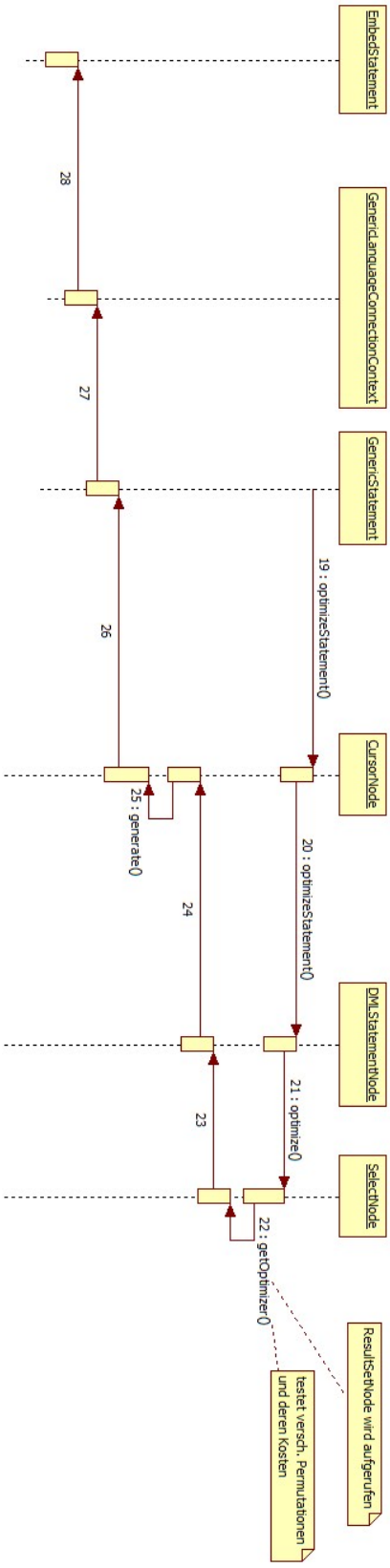

Abbildung 3.4: Darstellung des Optimierungsprozesses als Sequenzdiagramm

3.3.3 Kostenberechnung

Das Interface `CostEstimate` und dessen Implementierung `CostEstimateImpl` stellen Methoden zur Verfügung um die Kosten für ein `ResultSet` zu repräsentieren. Die Berechnung der Kosten findet dann in den Methoden der Optimizerimplementierung und des Optimizable statt. Bei der Initialisierung eines Optimizers werden auch Variablen für die Kostenschätzung angelegt, die später dann mit den besten Kosten je nach Permutation und Zugriffspfad belegt werden. Zunächst werden die Kosten für das Optimizable berechnet. In der Klasse `OptimizerImpl` werden unter anderem die Methoden `costBasedCostOptimizable()`, `estimateTotalCost()` und `getNewCostEstimate()` aufgerufen. Die erste Methode wird aufgerufen, wenn es sich wie in unserem Fall um eine kostenbasierte Optimierung handelt. Andernfalls können auch andere Strategien zum Einsatz kommen. Beim Aufruf der zweiten Methode werden schließlich die Kosten für das Optimizable Objekt berechnet. Dazu wird die Methode `estimateCost()` in der Klasse aufgerufen, welche die entsprechende Methode von Optimizable überschreibt. Im Falle des `Select`-Statements handelt es sich um die Klasse FormBaseTable, welche dann die Methode `getScanCost()` eines zuvor erzeugten `StoreCostController` Objektes aufruft und schließlich die Kosten für dieses Optimizable berechnet. `StoreCostController` ist ein Interface, das in der Klasse `HeapCostController` implementiert wird. Im folgenden Listing wird der Code gezeigt, welcher für die Berechnung der Kosten verwendet wird.

```
1        long estimated_row_count = ((row_count < 0) ?  num_rows :
            row_count);
2        double cost = (num_pages * BASE_UNCACHED_ROW_FETCH_COST);
3        cost += (estimated_row_count * row_size) * BASE_ROW_PER_BYTECOST;
4        long cached_row_count = estimated_row_count - num_pages;
5        if (cached_row_count < 0)
6            cached_row_count = 0;
7
8        if (scan_type == StoreCostController.STORECOST_SCAN_NORMAL)
9            cost += cached_row_count * BASE_GROUPSCAN_ROW_COST;
10       else
11           cost += cached_row_count * BASE_HASHSCAN_ROW_FETCH_COST;
12
13       ...
14
15       cost_result.setEstimatedCost(cost);
16       cost_result.setEstimatedRowCount(estimated_row_count);
```

Listing 3.2: Code für die Kostenberechnung beim Optimierungsprozess

In dieser Methode erfolgen mathematische Berechnungen der Kosten. Das Laden der Tabelle in den Zwischenspeicher verursacht Kosten, genau wie der Zugriff auf die Zeilen der Tabelle. Dabei wurden Konstanten definiert, die als Multiplikator zur Berechnung dienen. `BASE_UNCACHED_ROW_FETCH_COST` und `BASE_ROW_PER_BYTECOST` sind feste Variablen welche einen Teil der Kostengewichtung ausmachen. Die Variablen `num_pages`, `num_rows` und `row_size` werden beim Anlegen des Objektes initialisiert. In unserem Fall ist `num_pages = 1`, `num_rows = 7` und `row_size = 2048`. Die mit diesen Variablen berechneten Kosten werden in der Variable `cost` zwischengespeichert und dann an ein `CostEstimate` Objekt gebunden. Dieses Objekt kann dann später im Obtimizer zur Verfügung gestellt werden, wodurch dann die aktuellen Kosten vorliegen.

Für jede Permutation werden die Kosten berechnet und dann die geringsten Kosten als optimierter Zugriffsplan definiert.

3.4 Erstellen des Zugriffsplans

Nach dem Erstellen des optimierten Baumes, wird dieser für die Erstellung des Zugriffsplans zugrunde gelegt. Beim Erstellen des Zugriffsplans gibt es zwei Methoden, den Plan zu erstellen. DDL-Statements (`CreateTable`) benutzen eine vordefinierte Klasse um den Plan zu erstellen. DML-Statements hingegen erstellen zur Laufzeit neue Java-Klassen, die ausgeführt werden und nach der Ausführung wieder gelöscht werden.

Die Funktionalität wird in Derby in sogenannten `ActivationClasses` bereitgestellt, welche beim Generieren des Zugriffsplans erstellt werden. `ActivationClasses` sind Klassen, die Bytecode ausführen. Sie greifen auf im Speicher abgelegten Bytecode zu und führen diesen aus. Diese ActivatonClass wird in ein Objekt des `PreparedStatements` gespeichert. Die ActivationClass ist vom Typ `GeneratedClass`, eine Basisklasse, die generierte Klassen repräsentiert. Sie beinhaltet ein Attribut vom Typ `ClassInfo` welches wiederum ein Attribut `clazz` vom Typ `Class<T>` beinhaltet. Im diesem Attribut ist die generierte Klasse abgelegt. Bei DDL-Statements ist es eine vordefinierte Klasse vom Typ `ConstantActionActivation`. `Actions` sind Klassen, die Bytecode, der zum Ausführen bestimmter Aktionen benötigt wird, beinhalten und in den Speicher legen. Da die DDL-Statements immer gleich aufgebaut sind, wird hier keine dynamisch generierte Klasse benötigt. Für die verschiedenen DDL-Statementtypen gibt es verschiedene Ableitungen:

- Sämtliche Create-Anweisungen (z.B. `CreateTableConstantAction`)
- Sämtliche Drop-Anweisungen (z.B. `DropTableConstantAction`)
- sowie noch einige andere konstante Aktionen, die bei Statements anfallen

Anders sieht es bei DML-Statements aus. Diese Statements können sehr variabel aufgebaut sein. Um diese Statements im Zugriffsplan darstellen zu können, werden zur Laufzeit neue Klassen erstellt. Diese Klassen sind alle von der Basisklasse `BaseActivation` abgeleitet. Da diese Klassen dynamisch generiert werden, sind sie auch nicht im Quellcode von Derby enthalten. Zum Lesen des in den dynamisch generierten Klassen enthaltenen Codes wird ein spezieller Disassemblierer benötigt, weshalb hier im Folgenden nur auf die Implementierung in Derby eingegegangen wird.

Das Generieren läuft in folgenden Schritten in der Funktion `generate` der Klasse StatementNode ab:

- Zunächst wird geprüft, welche Art von `Activation` man benötigt. Bei DDL-Statements ist das beispielsweise eine Konstante `NEED\_DDL\_ACTIVATION`. Mit dieser Konstanten wird später geschaut, welchen Code man zu generieren hat (statisch oder dynamisch).
- Danach wird anhand der Konstante entschieden, welche Art von Generierenung man benötigt. Bei DDL-Statements wird nur eine vordefinierte Klasse `org.apache.derby.impl.sql.execute.ConstantActionActivation` geladen und zurückgegeben. Bei DML-Statements wird stattdessen ein String `SuperClass` angelegt und die Ausführung der Funktion fortgesetzt.

- Liegt ein DML-Statement vor, wird ein Objekt des Interfaces `ActivationClassBuilder` angelegt. Dieses enthält die gemeinsamen Funktionen, die zum Generieren von ausführbarem Code nötig sind. Implementiert werden die Funktionen von den einzelnen Nodes.
- Anschließend werden zwei Methoden generiert. Diese sind `execute()` zum Ausführes des Statements und `fillResultSet()`, um das ResultSet zu füllen, falls es nach der Ausführung leer ist. Die Generierung der Methoden erfolgt über den `MethodBuilder`.

```
1        MethodBuilder executeMethod = generatingClass.
            getExecuteMethod();
2
3        MethodBuilder mbWorker = generatingClass.getClassBuilder().
            newMethodBuilder(
4                Modifier.PRIVATE,
5                ClassName.ResultSet,
6                "fillResultSet");
```

Listing 3.3: Erstellen der Methoden execute() und fillResultSet()

- Diese beiden Methoden gehören nun zum Objekt `generatingClass` und können im Folgenden mit Aktionen gefüllt werden. Dabei kommt eine Push-Pop-Architektur auf einem virtuellen Stack zum Einsatz. Damit wird der ByteCode generiert. Die Bekannten Code-Elemente, wie z.B. IF, RETURN usw. haben eine Implementierung in Derby, welche den zugehörigen Bytecode erzeugt. Diese Implementierungen werden im folgenden Codeausschnitt aufgerufen. Man erkennt z.B. ein If-Konstrukt.

```
1   mbWorker.addThrownException(ClassName.StandardException);
2
3        // Generate the complete ResultSet tree for this statement.
4        // This step may add statements into the execute method
5        // for per-execution actions.
6        generate(generatingClass, mbWorker);
7        mbWorker.methodReturn();
8        mbWorker.complete();
9
10    executeMethod.pushThis();
11    executeMethod.getField(ClassName.BaseActivation, "resultSet",
12                ClassName.ResultSet);
13
14    executeMethod.conditionalIfNull();
15
16            // Generate the result set tree and store the
17            // resulting top-level result set into the resultSet
18            // field, as well as returning it from the execute method
                .
19
20      executeMethod.pushThis();
21      executeMethod.callMethod(VMOpcode.INVOKEVIRTUAL, (String) null,
22                  "fillResultSet", ClassName.ResultSet, 0);
23            executeMethod.pushThis();
24            executeMethod.swap();
```

```
25              executeMethod.putField(ClassName.BaseActivation, "
                   resultSet", ClassName.ResultSet);
26
27      executeMethod.startElseCode(); // this is here as the compiler
           only supports ? :
28        executeMethod.pushThis();
29        executeMethod.getField(ClassName.BaseActivation, "resultSet",
             ClassName.ResultSet);
30      executeMethod.completeConditional();
```

Listing 3.4: If-Konstrukt

- Die Methode wird dann beendet und auch der Konstruktor zur generierten Klasse wird generiert und beendet.

```
1       // wrap up the activation class definition
2     // generate on the tree gave us back the newExpr
3     // for getting a result set on the tree.
4     // we put it in a return statement and stuff
5     // it in the execute method of the activation.
6     // The generated statement is the expression:
7     // the activation class builder takes care of constructing it
8     // for us, given the resultSetExpr to use.
9     //   return (this.resultSet = #resultSetExpr);
10    generatingClass.finishExecuteMethod(this instanceof CursorNode);
11
12    // wrap up the constructor by putting a return at the end of it
13    generatingClass.finishConstructor();
```

Listing 3.5: Finalisieren der Methoden

- Zuletzt wird nur noch die generierte Klasse in eine ActivationClass gewandelt und an das Statement zurückggeben.

```
1       // cook the completed class into a real class
2     // and stuff it into activationClass
3     GeneratedClass activationClass = generatingClass.
         getGeneratedClass(byteCode);
4     return activationClass;
```

Listing 3.6: Ende der Funktion generate()

Nach diesen vier Schritten ist nun das `PreparedStatement` zur Ausführung bereit. Mit dem Aufruf `stmt.execute()` wird die Ausführung des Statements letztendlich gestartet und ein `ResultSet` zurückgegeben.

4 B$^+$-Baum Implementierung

Dieses Kapitel befasst sich mit der B$^+$-Baum Implementierung in Apache Derby und gibt einen Überblick über den Aufbau und die Funktionsweise dieser Implementierung.

Zu Beginn erfolgt zunächst ein kurzer Auszug über die wesentlichen Merkmale und Einsatzgebiete von B$^+$-Bäumen sowie den allgemeinen Aufbau der in Apache Derby verwendeten B$^+$-Bäume. Im Folgenden werden dann im Detail die Funktionsweise, die beteiligten Klassen und Methoden sowie die Kernfunktionen der B$^+$-Baum Implementierung in Apache Derby beschrieben und mit Hilfe entsprechender Funktionsablaufdiagramme erläutert. Abgerundet wird dieses Kapitel mit einem praktischen Beispiel, an Hand dessen die Funktionsweise bei der Erstellung und Erweiterung eines B$^+$-Baums mit den jeweils zugehörigen Debug-Ausgaben noch einmal veranschaulicht werden soll. Ein weiteres Augenmerk liegt dabei auf den Standard- Sperrmechanismen, die beim Lesen und Schreiben des B$^+$-Baums verwendet werden.

4.1 B$^+$-Baum Allgemein

Bei B-Bäumen handelt es sich um Datenstrukturen aus den Bereichen der Datenbanken und Dateisysteme wo diese für die optimierte Suche nach Datensätzen zum Einsatz kommen. Dabei werden in diesen Strukturen Schlüssel (Primär- und Sekundärschlüssel) mit den entsprechenden Verweisen auf die eigentlichen Datensätze hierachisch sortiert abgespeichert. Somit ist eine effiziente Suche und Verwaltung von großen Datenmengen möglich. B$^+$-Bäume sind eine spezielle Ausprägung der B-Bäume und findet in der Implementierung von Apache-Derby seinen Einsatz.

Eines der wesentlichen Merkmale von B$^+$-Bäumen gegenüber B-Bäumen ist, dass die eigentlichen Informationen, also die Tupel bestehend aus Schlüssel und Zeiger/ Verweise auf die Daten, nur in den Blättern vorliegen. Knoten, die keine Blätter sind enthalten lediglich die Informationen die notwendig sind, damit die Suchfunktionen die entsprechenden Blätter finden, also die Verweise auf die Blätter. Somit ist eine effiziente Suche sowie flache Strukturen möglich. Ein weiteres wichitges Merkmal von B$^+$-Bäumen ist die Verkettung der Blätter von links nach rechts, so dass auf dieser Ebene eine sequentielle Suche möglich ist, die die Kosten der Suchanfragen gegenüber herkömmlicher B-Bäume deutlich reduzieren, da ein Sprung von Blatt zu Blatt möglich ist, ohne jeweils wieder auf den Elternknoten zurückspringen zu müssen.

4.1.1 Aufbau eines B+-Baumes

Der Aufbau einer B+-Baum Struktur, wie sie in Derby genutzt wird ist identisch zu der in der Literatur beschriebenen Vorgehensweise. Wie schon im vorherigen Abschnitt beschrieben, werden B+-Bäume zur strukturierten Speicherung von Indizes eingesetzt.

Bei der Initialisierung eines Baums, also dem Anlegen eines ersten Elements, wird diesem zunächst ein entsprechender Speicherbereich in Form eines **Container** zugewiesen. Dabei handelt es sich um eine mitwachsende Speicherstruktur auf der Festplatte. Da in der Regel beim Anlegen einer Datenbank mehrere Indizies verwaltet werden, erhält jeder B+-Baum seinen eigenen Container zugewiesen, der dann auch beim Löchen des Baums ebenfalls komplett gelöscht wird. Innerhalb eines Containers ist die Baumstruktur in sogenannte **Pages** gegliedert, die ein Element des Baumes darstellen. Die Größe einer Page ist in diesem Fall auf **4096** Bytes festgelegt und hat damit wesentlichen Einfluss auf die Struktur eines B+-Baums. Ist der Speicherplatz einer Page erschöpft, so werden weitere Pages angelegt und somit neue Elemente des Baumes gebildet. Die folgende Abbildung 4.1 zeigt den elementaren Aufbau eines Baumelements.

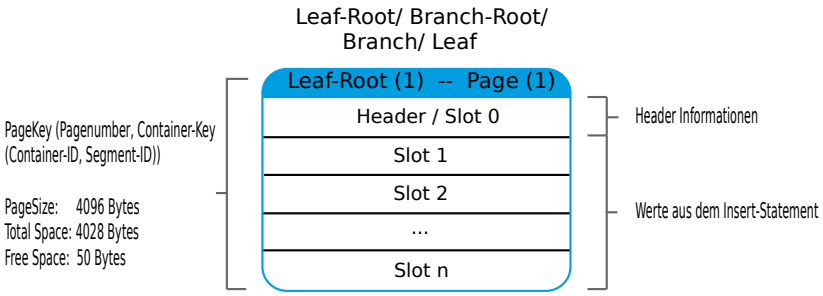

Abbildung 4.1: B+-Baum Grundelement

Ein B+-Baum besteht dabei im Wesentlichen aus drei unterschiedlichen Elementen, die sich jedoch in ihrer Struktur nicht unterscheiden. Das Wurzel-Element trägt die Bezeichnung **Leaf-Root** und drückt damit aus, dass keine weiteren Elemente folgen. Es ist somit Wurzel und Blatt zugleich. Wächst der Baum und ist eine Aufteilung des ersten Elements, also dem **Leaf-Root** notwendig, so wird daraus das Element **Branch-Root**, mit einer ersten Verzweigung auf die Blätter, die als **Leaf** bezeichnet werden. Knoten innerhalb eines Baums tragen die Bezeichnung **Branch**. Technisch gesehen wird ein Leaf / Branch durch eine Leaf-ControlRow bzw. BranchControlRow repräsentiert, in denen die nötige Anwendungslogik enthalten ist, um ein Blatt oder Knoten zu verwalten. Mit den zur Verfügung stehenden Methoden wie *search()*, *shrinkFor()* oder *splitFor()* lassen sich verschiedene Operationen wie das Verschmelzen oder Verteilen von Knoten oder Blätter durchführen.

Der weitere Aufbau der Elemente und damit die Strukturierung der Pages wird durch die im Folgenden aufgeführte Debug-Ausgabe ersichtlich. Die Debug-Ausgaben wurden über die, im B+-Baum-Paket enthaltenen Funktionen **debugConglomerate()** der Klasse **OpenBTree** sowie **printTree()** der Klassen **ControlRow**, **BranchControlRow** und **LeafControlRow** gewonnen.

```
1 Page(1,Container(0, 1041))is latched: true
2 Page 1 is latched: true
3
4 LEAF-ROOT(1)(lev=0): num recs = 4
5   left = -1;right = -1;parent = -1;isRoot = true;
6 PAGE:(1)----------------------------------------:
7  :row[1](id:7): col[0]=AF col[1]=(2,6) col[2]=x
8  ...
9  ...
```

Listing 4.1: Debugausgabe

In der ersten Zeile sind die Angaben zu der Page mit ihrer ID und dem dazugehörigen Container zu finden. Ebenfalls ist ersichtlich, dass die Page für den Zugriff gelockt wurde. Zeile 4 gibt den Elementtyp mit der jeweiligen ID, den Level, also dei Ebenen des Baums sowie die Anzahl der enthaltenen Datensätze/ Tupel an. Die Nummerierung der Elemente entspricht den ID's der zu den Baum-Elementen gehörigen Pages. Da es sich in diesem Fall um das Wurzel-Element handelt, ist der Level gleich 0. Das Feld "num recs" gibt die Anzahl der Datensätze an, was der Belegung der Slots innerhalb einer Page entspricht. Da ein Blatt mindenstens einen Slot besitzt, hat somit num recs den Anfangswert eins. Zeile 5 enthält Angaben über die Position des Elements im B+-Baum und die Folgeelemente, sofern vorhanden. Diese Informationen werden im Header der Page abgelegt. Ab Zeile 7 folgen dann die Informationen zur Belegung der Slots innerhalb der Page. Der Wert von Row entspricht dabei dem Slot und verfügt über eine eigene ID. Innerhalb des Slots werden die Einträge über die Spalten (col[i]) strukturiert. Die erste Spalte enthält den eigentliche Indexwert aus dem Insert-Statement, Spalte 2 den Verweis auf den eigentlichen Datensatz. Einträge in der 3. Spalte liegen nur vor, wenn Folgeelemente existieren und enthalten somit einen Verweis auf die zu diesem Tupel zugehörige Page-ID.

Die folgende Abbildung 4.2 zeigt noch einmal den allgemeinen Aufbau eines B+-Baums, der sich aus dem obigen Ablauf ergibt.

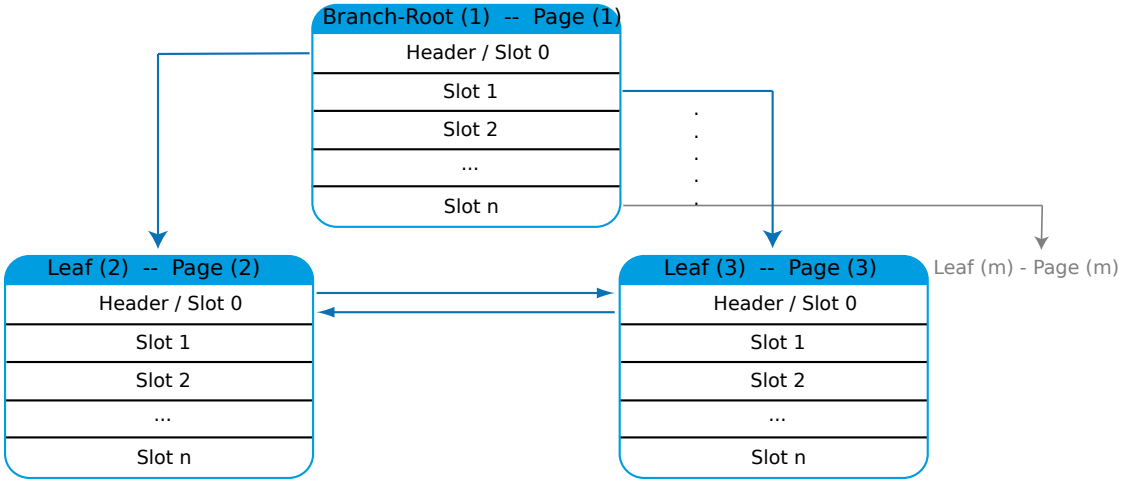

Abbildung 4.2: B+-Baum Grundaufbau

Am Ende dieses Kapitels über B+-Bäume wird die Vorgehensweise der systematischen Erzeugung eines B+-Baums bis auf Ebene/Level 3 an einem praktischen Beispiel grafisch und den passenden Debug-Ausgaben dargestellt.

4.2 Derbys B+-Baum Implementierung

Apache Derby's B+-Baum Implementierung liegt als einzelnes Paket vor und ist unter der folgenden Struktur zu finden:

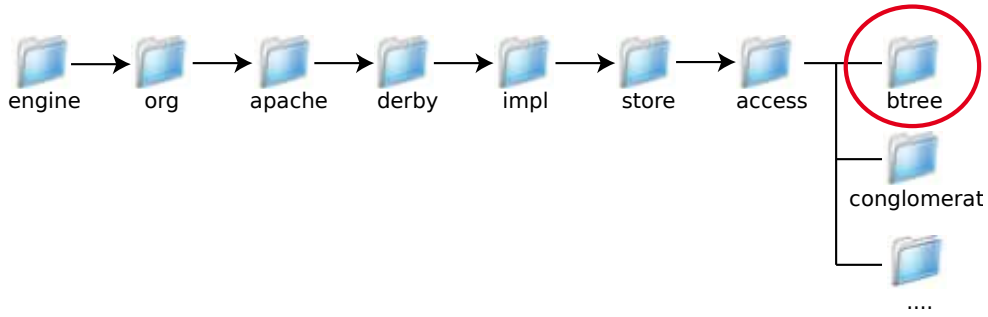

Abbildung 4.3: B+-Baum Paketstruktur

Nach der Paketübersicht (Abbildung 4.3) liegen alle Klassen, die für die Verwaltung eines B+-Baumes wichtig sind, in dem "btree"-Paket vor. In diesem Paket ist zudem das "index"-Paket enthalten, das verschiedene Klassen bereitstellt, die für die Verwaltung der Sekundärindizes verantwortlich sind. Im Folgenden werden die einzelnen B+-Baum Operationen wie das Suchen (Abschnitt 4.2.1), Einfügen (Abschnitt 4.2.2) oder Löschen (Abschnitt 4.2.3) eines Eintrages, mithilfe von Aktivitätsdiagrammen beschrieben.

4.2.1 Suchen im B$^+$-Baum

Wie aus der Abbildung 4.4 ersichtlich ist, ist die Klasse **BtreeForwardScan** für die Suche von den Indexeinträgen im B$^+$-Baum verantwortlich. Mit dem Aufruf der *positionAtStartForForwardScan()*-Methode beginnt die eigentliche Suche nach den Einträgen. Die Klasse **BtreeScan** bzw. **BtreeForwardScan** enthält verschiedene Informationen wie den Start- oder Stop-Wert nach dem im B$^+$-Baum gesucht werden soll.

Mit dem Aufruf der *get()*-Methode wird zunächst der Root-Knoten des B$^+$-Baumes geladen, der entweder vom Typ BranchControlRow oder LeafControlRow ist. Handelt es sich bei dem zurückgelieferten ControlRow um eine Instanz von BranchControlRow, dann wird die dazugehörige *search()*-Methode aufgerufen und nach dem ersten Blatt gesucht, die den gesuchten Start-Wert enthält.

Hierbei wird mit der *getChildPageAtSlot()*-Methode der linke Child-Knoten von dem BranchControlRow geladen und der aktuelle ControlRow freigegeben. Dieser Vorgang wird rekursiv solange wiederholt, bis der dazugehörige LeafControlRow erreicht wurde. Anschließend wird das gefundene Blatt an die aufrufende Funktion zurückgeliefert und mit der Suche nach weiteren Blättern fortgesetzt.

Im nächsten Schritt werden die Blätter durchlaufen und mittels der *compareIndexRowToKey()*-Methode mit dem Stop-Wert verglichen. Die Suche ist beendet, wenn entweder der zurückgelieferte Wert größer oder gleich null ist, oder das letzte Blatt im B$^+$-Baum erreicht wurde. Ansonsten wird mit der *positionAtNextPage()*-Methode das jeweils nächste (rechte) Blatt geholt und mit dem Stop-Wert verglichen.

Zum Schluß kümmert sich die *positionAtDoneScan()*-Methode um das Aufräumen und der Freigabe aller resevierten Ressourcen. Anschließend wird die Anzahl der gefundenen Indexeinträge an die aufrufende Funktion zurückgeliefert und die Suche damit erfolgreich beendet.

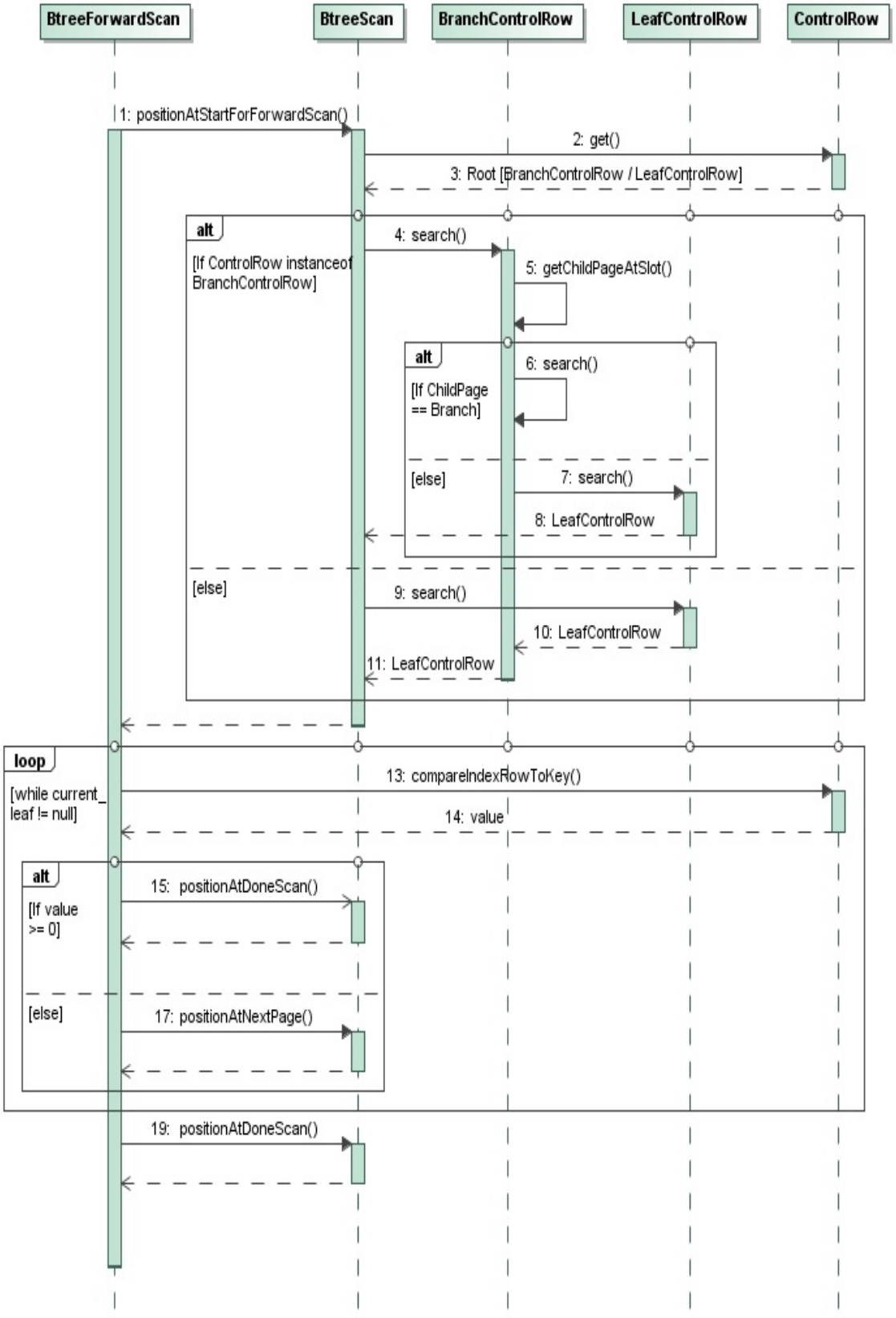

Abbildung 4.4: Suchen von Indexeinträgen

4.2.2 Einfügen im B+-Baum

An der Abbildung 4.5 erkennt man, dass die Klasse **RowChangerImpl** für das Einfügen eines neuen Eintrages zuständig ist. Zunächst werden mit der *insertAndFetchLocation()*-Methode die übergebenen Einträge auf dem Heap abgespeichert. Hierbei wird als erstes geprüft, ob schon eine Page auf dem Heap für die Einträge existiert. Dazu wird die Page entweder aus dem Heap geladen oder eine neue Page für die Einträge angelegt. Anschließend werden die Daten gesichert und mit der eigentlichen Verarbeitung des B+-Baumes fortgesetzt.

Im nächsten Schritt werden die Indexeinträge in den B+-Baum eingefügt. Dazu wird die *insert()*-Methode der Klasse **IndexSetChanger** aufgerufen, der für das Einfügen aller Indexe in den B+-Baum verantwortlich ist. In diesem Fall wird für jeden Indexeintrag die dazugehörige *insert()*-Methode aufgerufen.

Wie man an der Abbildung sieht, führt die Aufrufhierarchie über den **IndexChanger** bis zu dem **BTreeController**, der für die eigentliche Einfüge-Operation des Indexeintrags zuständig ist. Mit dem Aufruf der *doIns()*-Methode beginnt zunächst die Suche nach dem Blatt, in das der Eintrag eingefügt werden soll. Hierbei wird als erstes der Root-Knoten des B+-Baumes geholt und anschließend mittels der *search()*-Methode das passende LeafControlRow ausgewählt. Mit der *insertAtSlot()*-Methode wird versucht den Indexeintrag in das gefundene Blatt einzufügen. Hat das Blatt noch genug Platz, dann wird der Eintrag abgespeichert und der Status des Einfüge-Prozesses an die aufrufende Funktion zurückgeliefert.

Wohingegen bei Platzmangel eine "NoSpaceOnPage"-Ausnahme geworfen wird, so dass eine Teilung des aktuellen Blattes notwendig wird. Hierbei wird zunächst mithilfe der *createBranchRowFromOldLeafRow()*-Methode der Eintrag und die Pagenummer der alten Blattes zwischengespeichert, so dass der Zugriff auf das alte Blatt wieder freigegeben werden kann. Die Informationen aus dem BranchRow werden später bei dem Verteilungs-Prozess wiederverwendet.

Mit der *start_xact_and_dosplit()*-Methode startet anschließend die Verteilung der Einträge auf die Blätter oder innere Knoten. Dazu übernimmt der neue Knoten die alte Pagenummer des alten Blattes, so dass die neu erstellten Blätter eine neue Pagenummer zugewiesen bekommen. Mit der *get()*-Methode wird zunächst der Root-Knoten geladen, um mit dem "Top-Down"-Ansatz zu schauen, wo man durch eine Verteilung der Blätter bzw. Knoten noch Platz für den Indexeintrag schaffen kann. Daraufhin ist die *splitFor()*-Methode für die eigentliche Verteilung der Blätter und Knoten im B+-Baum verantwortlich. Hierbei werden neue Knoten und Blätter erstellt, die miteinander verlinkt werden und so Platz für den neuen Indexeintrag geschafft wird.

Letztendlich wird der Vorgang mit dem Einfügen des neuen Indexeintrags wiederholt, so dass dieser mittels der *insertAtSlot()*-Methode in das zuvor erstellte Blatt eingefügt wird und der Einfüge-Prozess damit erfolgreich abgeschlossen wird.

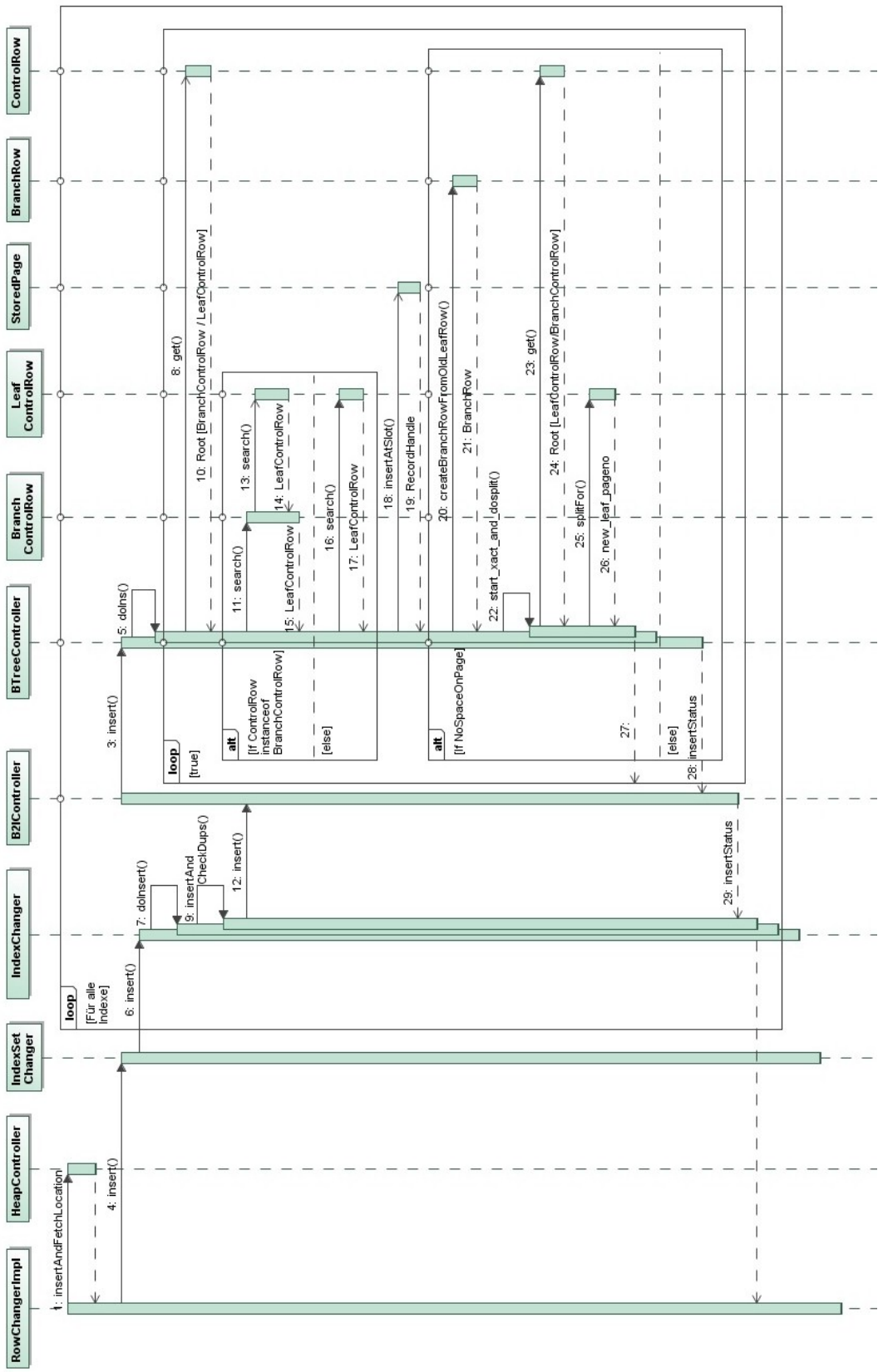

Abbildung 4.5: Einfügen von Indexeinträgen

4.2.3 Löschen im B+-Baum

Das Löschen eines Indexeintrags (siehe Abbildung 4.6) beginnt mit der **RowChangerImpl**-Klasse, die mit dem Aufruf der *delete()*-Methode die zu löschenden Einträge an den **IndexSetChanger** überträgt. Anschließend werden für alle Indexe die *delete()*-Methode aufgerufen. Mit der *doDelete()*- bzw. *delete()*-Methode beginnt der eigentliche Löschvorgang eines Eintrages aus dem B+-Baum. Hierbei wird mit dem Aufruf der *isDeletedAtSlot()*-Methode zunächst geprüft, ob der Indexeintrag schon als gelöscht markiert wurde. Trifft der Fall zu, dann ist der Eintrag schon gelöscht und es geht mit der weiteren Verarbeitung weiter. Im Gegensatz dazu, ist der Indexeintrag in dem B+-Baum noch verfügbar, so dass der Eintrag anschließend mittels der *deleteAtSlot()*-Methode als gelöscht markiert wird.

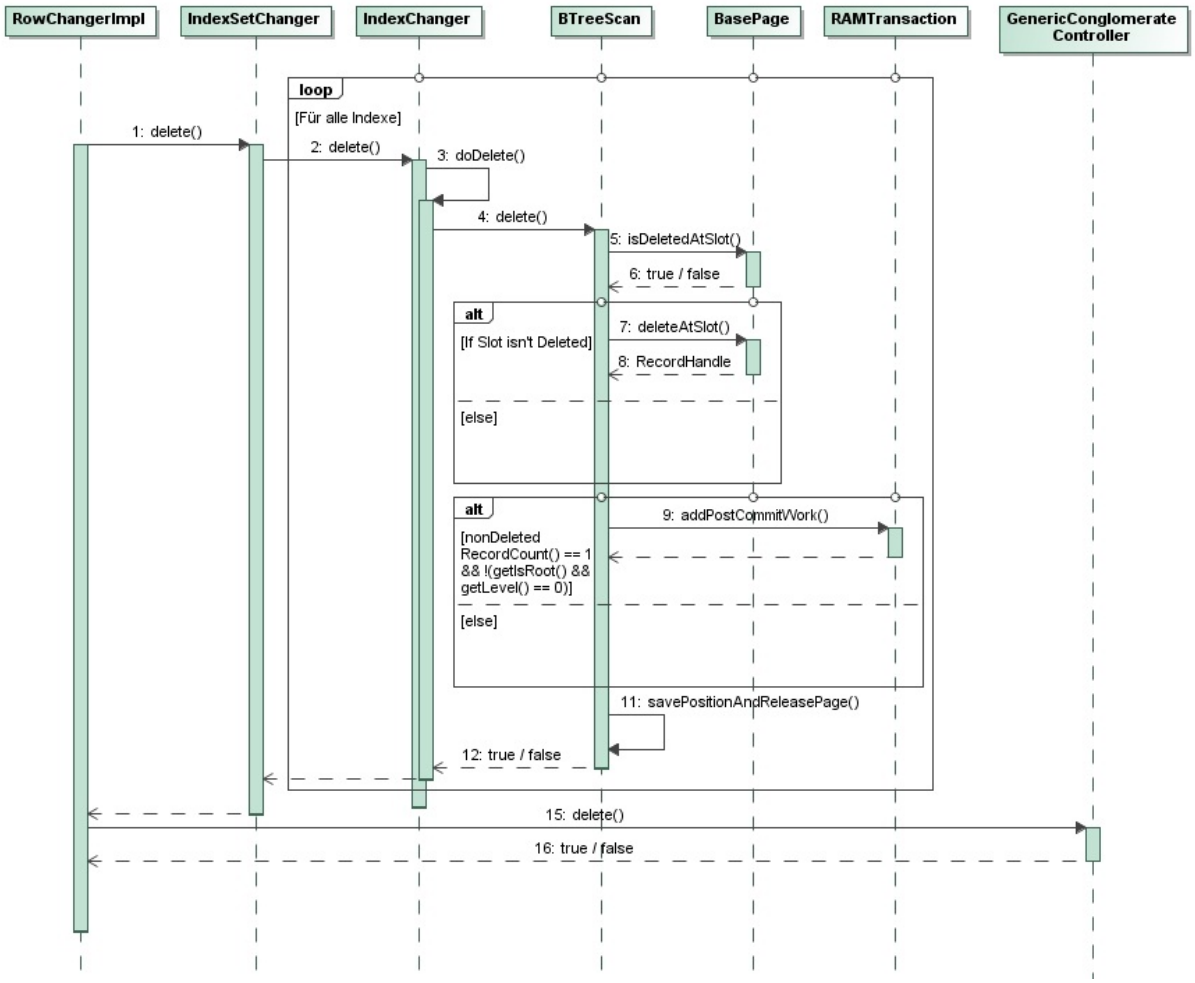

Abbildung 4.6: Löschen von Indexeinträgen

Im weiteren Verlauf des Löschvorgangs wird noch dafür gesorgt, dass beim Löschen von Einträgen die dazugehörigen Blättern und Knonten zusammengefügt werden können. Das Verschmelzen von Blättern und Knoten ist an bestimmte Bedingungen geknüpft. Hierbei wird zum einen geprüft, ob wir gerade den letzten Eintrag in dem Blatt gelöscht haben. Zum anderen wird geschaut, ob es sich bei dem Blatt nicht um den letzten Knoten bzw. dem Root von dem B+-Baum handelt. Wenn alle diese Bedingungen zu treffen, wird mittels der *addPostCommitWork()*-Methode ein "'PostCommitWork"'-Ereigniss an die aktuelle

Transaktion gehängt, das erst nach einem erfolgreichen Commit ausgelöst oder bei einem Transaktionsabbruch wieder verworfen wird.

Zum Schluß wird der exklusive Zugriff auf das Blatt mit der *savePositionAndReleasePage()*-Methode wieder freigegeben und die Indexeinträge mithilfe der *delete()*-Methode, die der **GenericConglomerateController** bereitstellt, in dem Heap bzw. Conglomerate als gelöscht markiert.

4.2.4 Verschmelzen von Knoten

Das "PostCommitWork"-Ereigniss wird wie bereits erwähnt, an die aktuelle Transaktion beim Löschen eines Eintrages gehängt. Diese Aktion ist dann für die eigentliche Verschmelzung der gelöschten Blätter und Knoten verantwortlich. Das Zusammenfügen von Blättern und Knoten wird erst durchgeführt, wenn z.B. alle Einträge in einem Blatt bzw. alle Blätter in einem Knoten als gelöscht markiert wurden und dies anschließend commitet wurde.

Wie man an der Abbildung 4.7 sieht, ist die Klasse **BTreePostCommit** für das Verschmelzen von Knoten und Blättern verantwortlich. Mit der *doShrink()*-Methode wird der Shrink-Vorgang gestartet. Zunächst wird mit der *get()*-Methode der Root-Knoten des B+-Baumes geladen, der dem BranchControlRow-Typ entspricht. Mit dem Aufruf der *shrinkFor()*-Methode wird als erstes der linke Child-Knoten von dem Root geladen. Anschließend wird die *shrinkFor()*-Methode für die Child-Knoten rekursiv solange aufgerufen, bis das gesuchte Blatt erreicht wurde. Die *unlink()*-Methode ist zum einen für das Löschen der Verweise zu und von dem Blatt verantwortlich. Zum anderen wird das Blatt nicht mehr nur als "gelöscht markiert", sondern endgültig aus dem Container entfernt und die benutzten Ressourcen wieder freigegeben.

Im nächsten Schritt müssen auch die Knoten aktualisiert werden, da diese immer noch einen Verweis auf das nun gelöschte Blatt enthalten. Hierbei müssen verschiedene Bedingungen geprüft werden, die für die weitere Verarbeitung des aktuellen Knotens wichtig sind. Handelt es sich bei dem gelöschten Blatt um einen Slot[1] in dem Knoten, dann wird der Slot mittels der *purgeAtSlot()*-Methode einfach entfernt. Wenn dies nicht der Fall ist, wurde stattdessen der linke Child-Knoten gelöscht, so dass eventuell ein neuer linker Child-Knoten aus den noch zur Verfügung stehenden Slots ausgewählt werden muss.

In diesem Fall wird zuvor geprüft, ob der Knoten mehr als einen Eintrag hat. Ist dies der Fall, dann hat der Knoten noch ein Blatt, das anschließend mittels der *getChildPageIdAtSlot()*-Methode geladen wird und mithilfe der *setLeftChildPageno()*-Methode als neuer linker Child-Knoten gesetzt wird. Danach wird der Eintrag in dem Knoten gelöscht und der Shrink-Vorgang damit abgeschlossen.

Hat der Knoten hingegen nur einen Eintrag, dann bedeutet dies, dass der Knoten keine weiteren Blätter oder Knoten hat. Diesbezüglich wird als nächstes geprüft, ob es sich bei dem Knoten um den Root-Knoten handelt. Wenn ja, dann wird ein neuer Root vom Typ LeafControlRow angelegt und mittels der *updateAtSlot()*-Methode mit den Informationen aus dem alten Knoten aktualisiert. Handelt es sich bei dem Knoten stattdessen nicht um einen Root-Knoten, so kann dieser ohne Probleme entfernt werden. Hierbei wird die *unlink()*-Methode aufgerufen und der Knoten entfernt. Wie zuvor erwähnt, werden auch seine Verweise zu

[1] Bei einem Slot (Eintrag) handelt es sich um den rechten Child-Knoten

und von anderen Knoten entfernt und die Verschmelzung der Blätter und Knoten damit abgeschlossen.

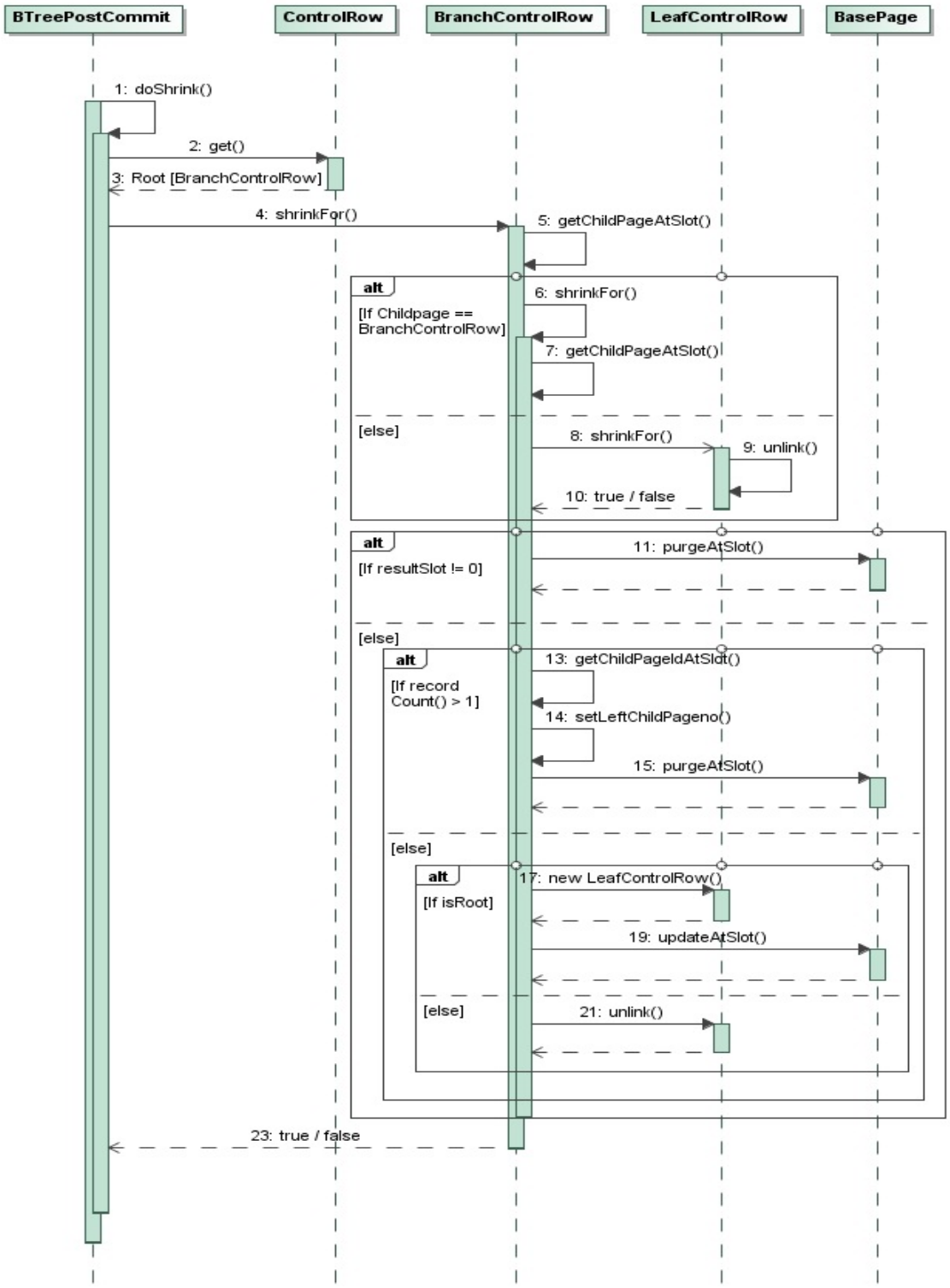

Abbildung 4.7: Verschmelzen von Blätter / Knoten

4.3 Exemplarischer Aufbau eines B+-Baums in Derby

In diesem Abschnitt wird an einem praktischen Beispiel der Aufbau eines B+-Baums und dessen Struktur grafisch und mittels der jeweiligen Debug-Logfile-Passagen beschrieben.

Für das Beispiel wurden Insert-Statements mit vereinfachten Datensätzen gewählt, die Informationen in Form von Ländernamen und den zugehörigen Ländercodes in die Datenbank "Countries" eintragen, wobei es sich bei den Ländernamen als auch den Länder-Isocodes um entsprechende Indizes handelt. Dadurch werden zur gleichen Zeit zwei B+-Bäume parallel angelegt und befüllt. Die Ländernamen wurden dabei um eine entsprechende Anzahl von Zeichen verlängert, so dass der Speicherplatz einer Page von 4096 Bytes maximal zwei Werte bzw. Ländernamen aufnehmen kann.

1. Datenbankeintrag:

```
1 insert into COUNTRIES values
2 ('AfghanistanAFaaaaaaaaaaaaaaaaaaaaa....','AF','Asia')
```

Dieses Statement führt, wie es die folgenden Debugausgaben zeigen, zur Initialisierung zweier B+-Bäume, die den Containern 1024 und 1057 zugeordnet wurden. Der Ländercode wird dabei in Container 1024 und der Ländername in Container 1057 eingetragen.

```
1 Page(1,Container(0, 1041))is latched:
2 true Page 1is latched: true
3
4 LEAF-ROOT(1)(lev=0): num recs = 2
5   left = -1;right = -1;parent = -1;isRoot = true;
6 PAGE:(1)----------------------------------------:
7  :row[1](id:7): col[0]=AFcol[1]=(2,6)
8
9 BasePage Page(1,Container(0, 1041))is latched: false
```

Listing 4.2: Insert AF-Container 1024

```
1 Page(1,Container(0, 1057))is latched: true
2 Page 1is latched: true
3
4 LEAF-ROOT(1)(lev=0): num recs = 2
5   left = -1;right = -1;parent = -1;isRoot = true;
6 PAGE:(1)----------------------------------------:
7  :row[1](id:7): col[0]=AfghanistanAFaaaaaaaaaaaaa...col[1]=(2,6)
8
9 BasePage Page(1,Container(0, 1057))is latched: false
```

Listing 4.3: Insert AF-Container 1057

Für die Einträge werden zunächst die jeweiligen Pages in den Zeilen 1 gelockt und am Ende der Aktion in den Zeilen 9 wieder freigegeben. Die Zeilen 4 bis 7 beinhalten die Informationen zu den Baumelementen und die eingetragenen Tupel. Dabei ist ersichtlich, dass die Verweise auf die eigentlichen Datensätze in beiden B+-Bäume identisch sind.

Nach der Analyse der Debug-Ausgaben ergibt sich daraus das in Abbildung 4.8 dargestellte Ergebnis der ersten Insert-Anweisung.

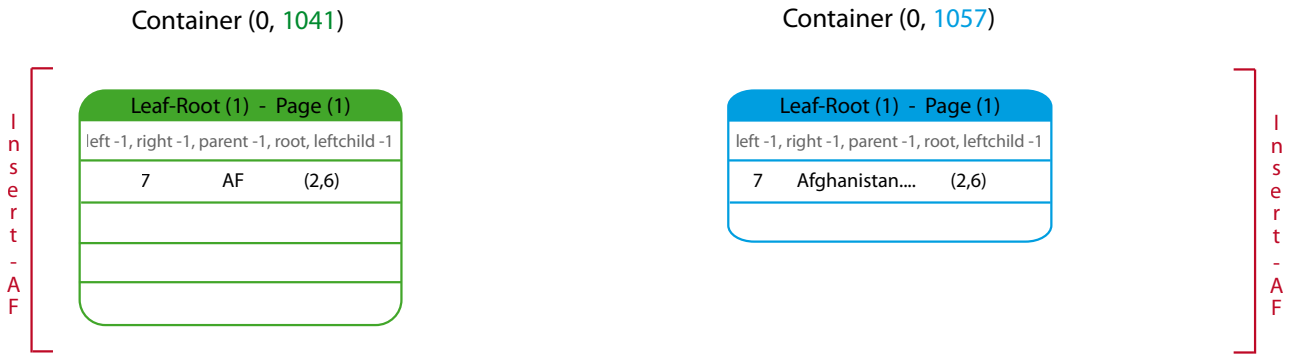

Abbildung 4.8: B+-Baum Bsp.-Insert AF

2. Datenbankeintrag:

```
1 insert into COUNTRIES values
2 ('Albaniaaaaaaaaaaaaaaaaaaaaaaaaaaaa....','AL','Europe')
```

Im Folgenden wird auf das Auflisten der Debug-Ausgaben verzichtet und direkt die Analyseergebnisse in grafischer Form dargestellt. Die zweite Insert-Anweisung führt zu dem Ergebnis in der Abbildung 4.9.

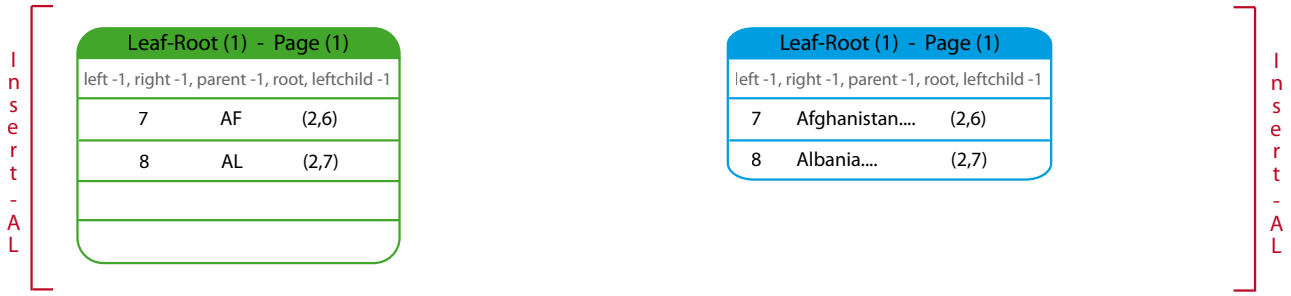

Abbildung 4.9: B+-Baum Bsp.-Insert AL

3. Datenbankeintrag:

```
insert into COUNTRIES values
('Algeriaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa....','DZ','North Africa')
```

Die dritte Insert-Anweisung liefert das in Abbildung 4.10 aufgeführte Ergebnis und zeigt erstmalig den Split-Vorgang in Container 1057. Aus dem Element **Leaf-Root** wurde das Element **Branch-Root** mit zwei angehängten Blättern, **Leaf(2)** und **Leaf(3)**. Die Besonderheit hier ist der Eintrag in col[2] mit dem Verweis auf Page(3). Des weiteren wird ein Verweis auf Page(2) im Header der Branch-Root eingetragen. Die Elemente Leaf(2) und Leaf(3) enthalten im Header die jeweiligen Verweise auf ihre Nachbarblätter und bilden somit einen verkettete Liste auf unterster Ebene.

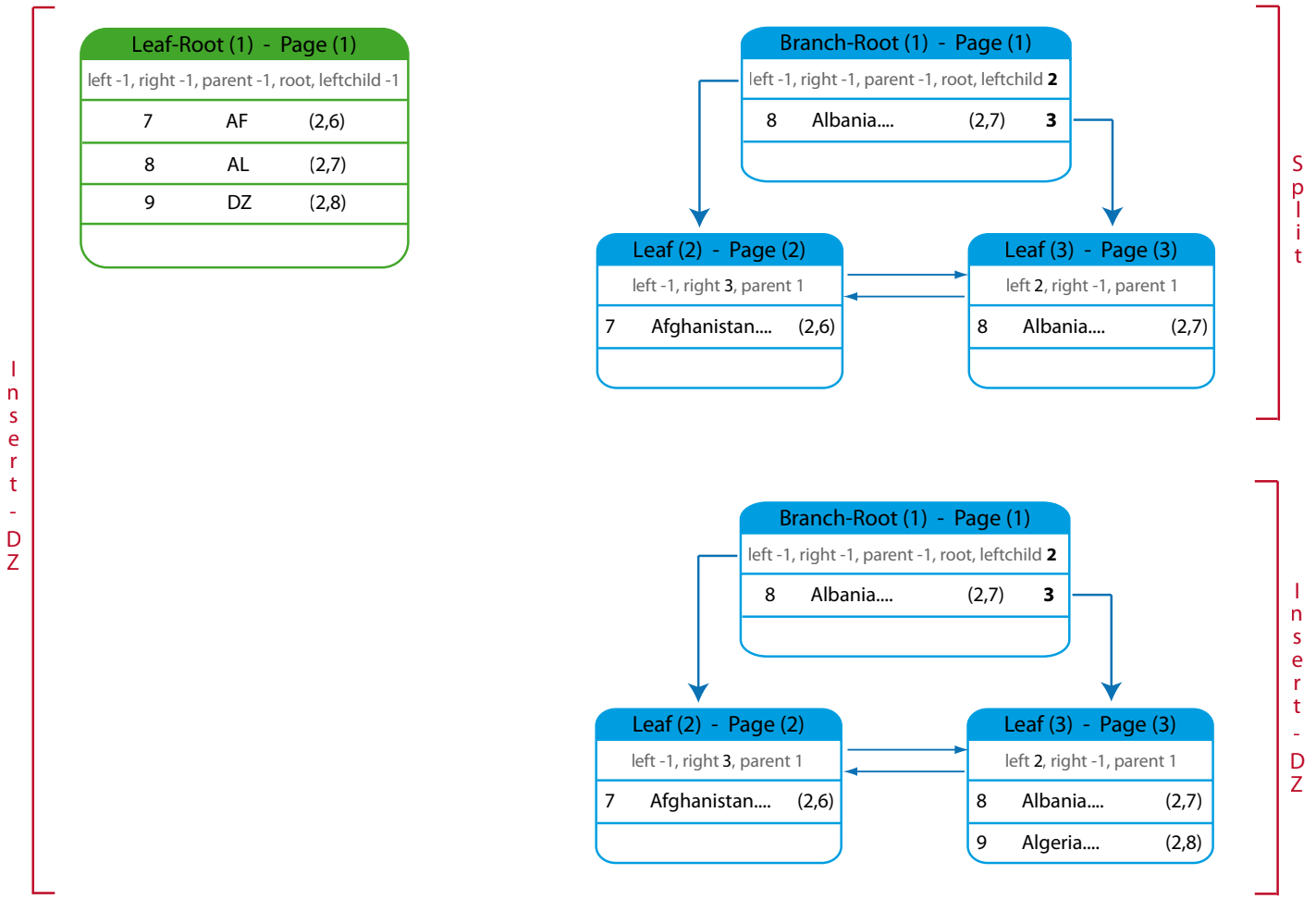

Abbildung 4.10: B+-Baum Bsp.-Insert DZ

4. Datenbankeintrag:

```
insert into COUNTRIES values
('American Samoaaaaaaaaaaaaaaaaaaaaaaa....','AS','Pacific Islands')
```

Abbildung 4.11 zeigt auf Grund der obigen vierten Insert-Anweisung einen erneuten Split, diesmal im Element **Leaf(3)**. Dadurch wird ein weiteres Blatt, **Leaf(4)**, dem Baum hinzugefügt und ein Verweis auf das neue Blatt im Element **Branch-Root** eingetragen.

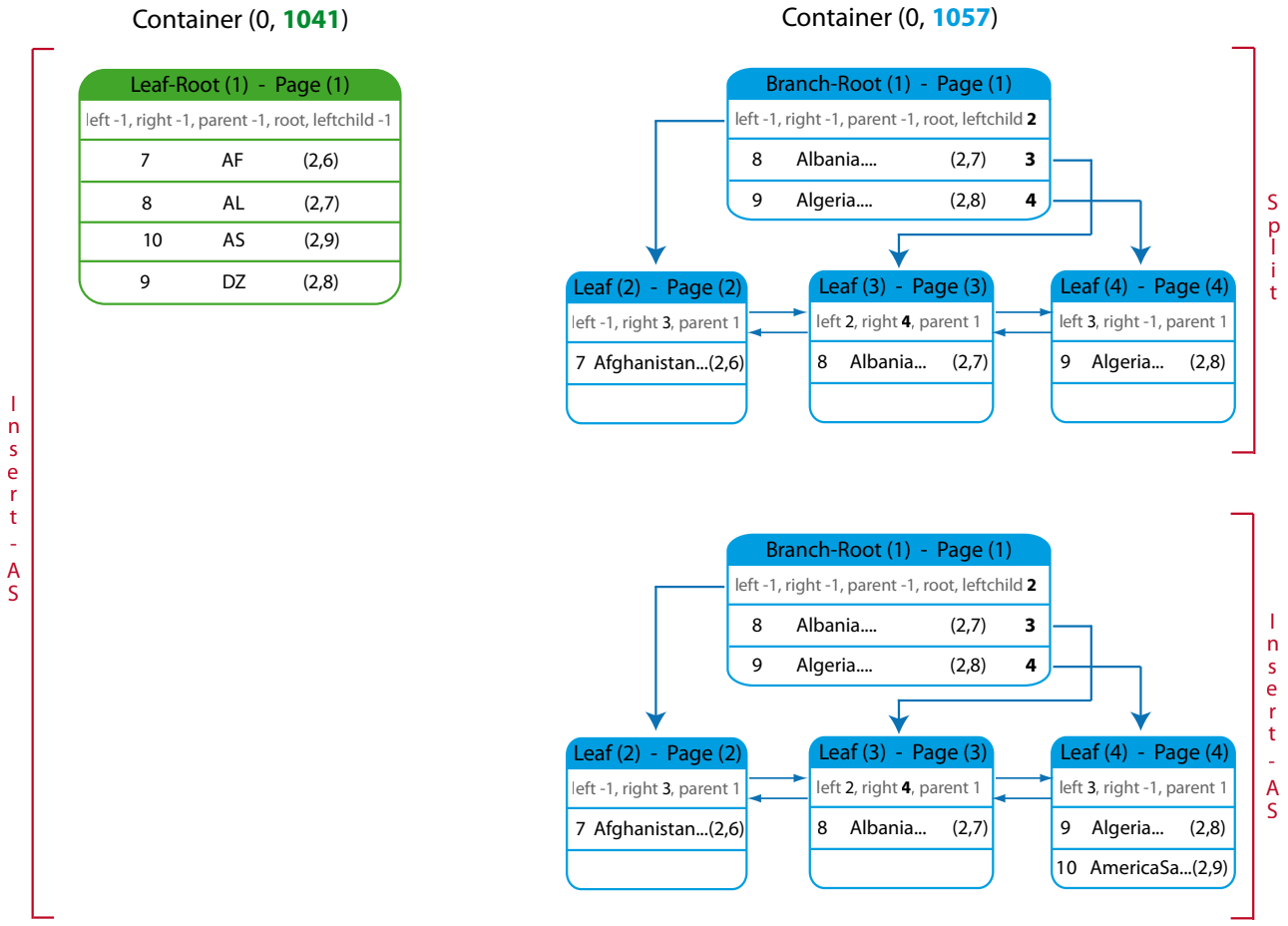

Abbildung 4.11: B+-Baum Bsp.-Insert AS

5. und letzter Datenbankeintrag:

```
insert into COUNTRIES values
('Angolaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa....','AO','Africa')
```

Die letzte Insert-Anweisung bewirkt einen erneuten Split-Vorgang, wie in Abbildung 4.12 zu sehen. Dieser Split führt diesmal zu einer größeren Veränderung des B+-Baums im Container 1057. Dabei wird der Baum um eine neue Ebene mit den Elementen **Branch(5)** und **Branch(6)** sowie um ein weiteres Blatt, **Leaf(7)** ergänzt.

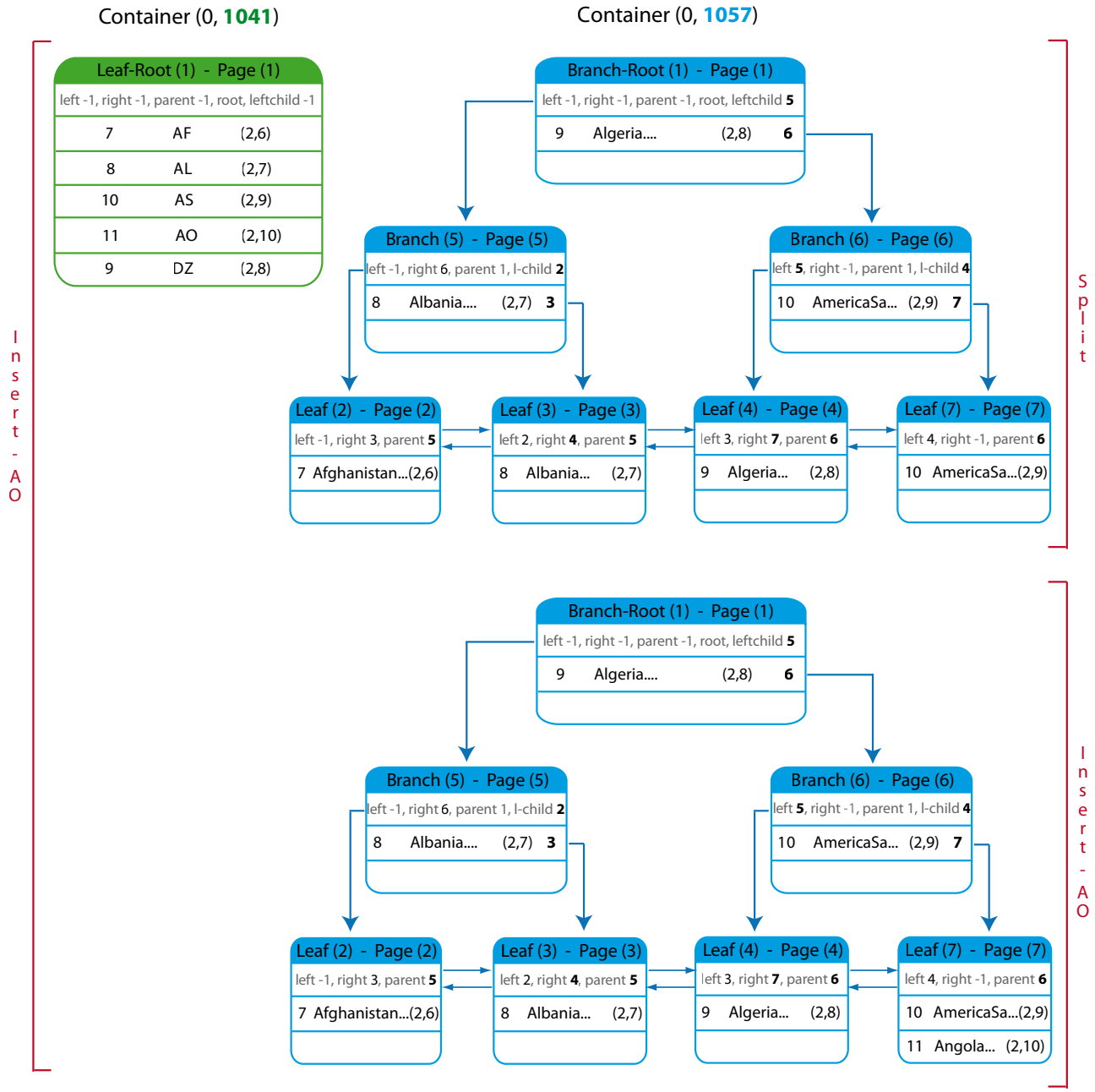

Abbildung 4.12: B+-Baum Bsp.-Insert AO

4.3.1 Sperrmechanismen beim Lesen und Schreiben in B+-Bäumen von Derby

Die in Apache Derby verwendeten Sperrmechanismen und Isolationslevel sind nicht Bestandteil dieses Abschnitts. Hier wird kurz auf das Locking bzw. Latching innerhalb des B+-Baums eingegangen. Weiterführende Informationen zu den in Derby verwendeten Isolationslevel sind im folgenden Kapitel "Transaktionen-Isolationslevel" zu finden.

Bei dem hier dokumentierten Locking bzw. Latching wird zwischen schreibenden und lesenden Aktionen im B+-Baum unterschieden. In beiden Fällen erfolgt das Latching immer über das Wurzelelements des B+-Baums und hangelt sich dann innerhalb der vorhandenen Struktur in Richtung der Blätter. Bei schreibenden Aktionen erfolgt das Latching über den gesamten Baum, während bei rein lesenden Aktionen die Elemente nacheinander wieder freigegeben werden (rollierende Sperren). Das bedeutet, dass nach einem erfolgreichen Latching des Kind-Elements das Elternelement wieder freigegenen wird.

Die folgende Abbildung 4.13 zeigt über den zeitlichen Verlauf das Latching, welches durch das Insert-Statement aus dem vorherigen Kapitel resultiert und zu dem, in Abbildung 4.12 dargestellten Ergebnis führt.

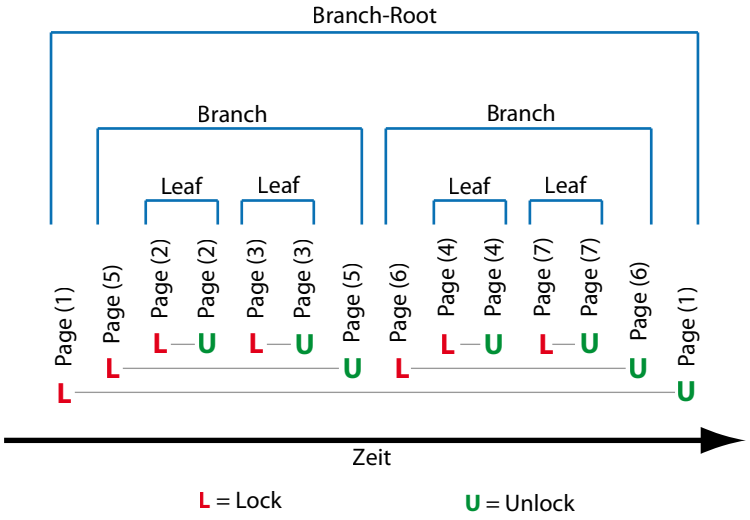

Abbildung 4.13: B+-Baum Bsp. für Latching

Das rollierende Sperren bei rein lesenden Zugriffen wird an dieser Stelle nicht extra grafisch dargestellt. Im Gegensatz zu dem in der obigen Abbildung 4.13 gezeigten Ablauf wird nach dem Sperren von Page(1), z.B. Page(6) gesperrt und im Anschluß Page(1) wieder freigegeben. Dieser Vorgang wird bis auf Blattebene fortgeführt.

5 Transaktionen und Isolationslevel

Dieser Kapitel beffast sich mit den Transakionen und den Isolationslevel. Es wird erklart wie Derby eine Transaktion in ein bestimmten Isolationslevel durchführt.

5.1 Isolation anlegen

Derby ist im Standard configuriert auf *TRANSACTION_READ_COMMITTED* wenn keine spezifische Isolation vom User eingegeben wurde. Es wird entweder durch das SQL statement *Set ISOLATION=* oder durch die funktion *SetTransactionIsolation(int)* der Klasse *Connection* geendert.

5.2 Locks

Die *LockFActory* Klasse, *org.apache.derby.iapi.-services.locks*, ist ein Interface das fur Clients die Anfrage und Hergabe von Locks an Objekten defieniert. Diese Locks sind an einem Compatibility Space geordnet. Dieser entscheidet zum Beispiel ob Zwei Locks von vershiedene Herkunft an einer selben Transaktion zugeteilt werden konnen.

5.2.1 Implementation

Die Implementation der *Lockfactory* Interface, *org.apache.derby.impl.- services.locks.SinglePool* unterstützt, wie der Name unterstreicht, einen Lock-pool mit einzelne Locksets. Die *Lockset* Klasse ist eine erweiterte *HashTable* indexiert durch Locktables. *SinglePool* erweitert die Hashtable um Lockspace-Objekten zu lagern. Aber nur ein Objekt in jedem SinglePool und LockSet ist erschafen. Das bedeutet das alle Locks-Anfragen seriel sind.

5.2.2 Typen von Locks

Die Daten-Locks sind weit verbreitet auf jede Tabelle und Row (multiple rows). Die Abstufung der mehrere Row Locks zu eine tabelle von Lock-level ist automatisch gemacht und ist ein veranderbarer Parameter. Derby gibt shared, update und exclusive Lock-Typen. Die vertraglichkeit zwischen de Locks zeigt ihnen das volgende Bild.

Der Locking ist unterschiedlich behandelt fur Containers (tabelle oder index) und Rows. Fur die Container ist es ein hierarchiches Locking. Die Lock-Typen definiert in

	Held		
Request	Shared	Update	Exclusive
Shared	ja	nein	nein
Update	ja	nein	nein
Exclusive	nein	nein	nein

Abbildung 5.1: Simple Lock compatibility

org.apache.derby.iapi.store.raw.ContainerLock sind CIS (Container Intent Shared), CIX (Container Intent Exclusive), CS (Container Shared), CU (Container Update) und CX (Container Exclusive). Die vertraglichkeit zwischen de Locks zeigt ihnen das volgende Bild.

	Held				
Request	CIS	CIX	CS	CU	CX
CIS	✓	✓	✓	-	-
CIX	✓	✓	-	-	-
CS	✓	-	✓	-	-
CU	-	-	✓	-	-
CX	-	-	-	-	-

Abbildung 5.2: Container Lock Compatibility

Fur Row-Locking sind es getrennte Lock-Typen: Level 3 Locks (serializable) und Level 2 Locks fur alle andere. Die Lock-Typen definiert in *org.apache.derby.iapi.store.raw.RowLock* sind RS2 (Row Shared lock fur level 2), RS3 (Row Shared lock fur level 3), RU2 (Row Update level 2), RU3 (Row Update level 3), RIP (Row Insert Previous key), RI (Row Insert), RX2 (Row Exclusive level 2) and RX3 (Row Exclusive level 3). Die vertraglichkeit zwischen de Locks zeigt ihnen das volgende Bild.

	Held							
Request	RS2	RS3	RU2	RU3	RIP	RI	RX2	RX3
RS2	✓	✓	✓	✓	✓	-	-	-
RS3	✓	✓	✓	✓	-	-	-	-
RU2	✓	✓	-	-	✓	-	-	-
RU3	✓	✓	-	-	-	-	-	-
RIP	✓	-	✓	-	✓	✓	✓	-
RI	-	-	-	-	✓	-	-	-
RX2	-	-	-	-	✓	-	-	-
RX3	-	-	-	-	-	-	-	-

Abbildung 5.3: Row lock compatibility

5.3 Der Gebrauch der Locks

Eine einfache Datenbank mit Primary-Keys. Der Isolation-Level ist Read Commited.

5.3.1 Lock Aktionen in eine single-row SELECT Transaktion

```
1 org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(
     RowLocking2.java:104): lockObject (no latch)(type 'IS')Container(0,
     1024)
```

Wahrend der suche im Index wird ein IS (Intention Shared) Lock im Container (file) der die B-Baum Struktur halt gesetzt. Die RowLocking2 Klasse stellt der Isolation-Level Read Commited dar.

Zur gleicher Zeit sieht es sich den LockSet Monitor und den SinglePool monitor an.

```
2 org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(
     RowLocking2.java:119): isLockHeld (type: 'X')Container(0, 1024)
```

Nachdem der IS Lock auf dem Container gesetzt wurde, wird uberpruft ob die Transaktion einen exclusiv Lock im Container schon besitzt. Wenn es so ist, kann der Lock gleich weggeschaffen werden.

Dies sieht sich den SinglePool Monitor an.

```
3 org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(
     RowLocking2.java:125): isLockHeld (type: 'S')Container(0, 1024)
```

Der dazu gehorige Check um zu sehen ob die Transaktion schon einen Shared Lock auf den Container halt.

Dies sieht sich den SinglePool Monitor an.

```
4 org.apache.derby.impl.store.raw.data.BasePage.setExclusive(BasePage.java
     :1781): latched. qual: (BaseContainerHandle:(Container(0, 1041)))ref:
     Page(1,Container(0, 1041))
```

Ein Latch wurde erhalten fur die Seite die den Root Knoten vom B-Baum enthaltet.

Dies sieht sich den LockSet Monitor an.

```
5 org.apache.derby.impl.store.raw.data.BasePage.setExclusive(BasePage.java
     :1781): latched. qual: (BaseContainerHandle:(Container(0, 1041)))ref:
     Page(5,Container(0, 1041))
```

Die Nachste Seite im B-Baum zu untersuchen wird ge-Latched.

Dies sieht sich den LockSet Monitor an.

```
6 org.apache.derby.impl.store.raw.data.BasePage.releaseExclusive(BasePage.
     java:1903): unlatched Page(1,Container(0, 1041))
```

Der Latch auf der Root-Seite des B-Baums wird aufgeloesst wenn der Latch an der nachste Seite zu untersuchen erhalten wurde.

Dies sieht sich den LockSet Monitor an.

```
7 org.apache.derby.impl.store.raw.xact.RowLocking2.lockRecordForRead(
     RowLocking2.java:166): lockObject (no latch)(type 'S')Record id=1 Page
     (5,Container(0, 1041))
```

Ein Read-Lock wird auf einen Record im B-Baum gesetzt wahrend der suche im B-Baum.

Dies sieht sich den LockSet Monitor an und dann den SinglePool Monitor.

```
8 org.apache.derby.impl.store.raw.xact.RowLocking2.lockRecordForRead(
     RowLocking2.java:166): lockObject (no latch)(type 'S')Record id=20
     Page(26,Container(0, 1024))
```

Ein Read-Lock wird auf ein anderer Record gesetzt der in einem anderen Container ist. Es handelt sich um den Record der sich unten den vorhige Record im B-Baum befindet.

Dies sieht sich den LockSet Monitor an und dann den SinglePool Monitor.

```
9 org.apache.derby.impl.store.raw.data.BasePage.releaseExclusive(BasePage.
     java:1903): unlatched Page(5,Container(0, 1041))
```

Der Latch auf der B-Baum Leaf Seite wird freigelassen.

Dies sieht sich den LockSet Monitor an.

```
10 org.apache.derby.impl.store.raw.xact.RowLocking2.unlockRecordAfterRead(
     RowLocking2.java:216): unlock (type 'S')Record id=1 Page(5,Container
     (0, 1041))
```

Der Record im B-Baum wird freigeschaltet.

Dies sieht sich den SinglePool Monitor an und dann den LockSet Monitor.

```
11 org.apache.derby.impl.store.raw.xact.RowLocking2.unlockContainer(
     RowLocking2.java:248): unlockGroup: BaseContainerHandle:(Container(0,
     1024))
```

Der von der Query zuruck gegebener ResultSet wird geschlossen und der Shared Lock auf den Container wird aufgemacht.

Dies sieht sich den LockSet Monitor an fur jeden Lock in der Groupe der aufgemacht werden soll.

```
12 org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857)
     : unlockGroup: 6548
```

Der Resultset wird geschlossen und weil der Auto-Commit lauft, wird dies den Commit auslosen. Dies sieht sich den LockSet Monitor an fur jeden Lock in der Groupe der aufgemacht werden soll.

```
13 org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857)
      : unlockGroup: 6548
```

Der Transaktion-Objekt ist noch da und releaseAllLocks is aufgeruffen wenn die Connection geschlossen wird.

Dies sieht sich den LockSet Monitor an fur jeden Lock in der Groupe der aufgemacht werden soll.

5.3.2 Lock Aktionen in eine single-row Update Transaktion

```
1 org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(
      RowLocking2.java:104): lockObject (no latch)(type 'IX')Container(0,
      1024)
```

Ein RowChanger resultset wurde geoffnet und die Suche im B-Baum is gestartet beim setzen eines IX (Intent Exclusive) Lock auf dem Data-Container.

```
2 org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(
      RowLocking2.java:119): isLockHeld (type: 'X')Container(0, 1024)
```

Ein Check wird durchgefuhrt um zu sehen ob es schon einen exclive Lock au dem Container gibt.

```
3 org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(
      RowLocking2.java:104): lockObject (no latch)(type 'IX')Container(0,
      1024)
4
5 org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(
      RowLocking2.java:119): isLockHeld (type: 'X')Container(0, 1024)
```

Die selben zwei locking Operationen werden noch einmal durchgefuhrt beim offnen eines *NormalizeResultSet*.

```
6 org.apache.derby.impl.store.raw.data.BasePage.setExclusive(BasePage.java
      :1781): latched. qual: (BaseContainerHandle:(Container(0, 1041)))ref:
      Page(1,Container(0, 1041))
```

Der Root-Block im B-Baum wird ge-latched.

```
7 org.apache.derby.impl.store.raw.data.BasePage.setExclusive(BasePage.java
      :1781): latched. qual: (BaseContainerHandle:(Container(0, 1041)))ref:
      Page(5,Container(0, 1041))
```

Der nachste Block im B-Baum wird ge-latched.

```
8 org.apache.derby.impl.store.raw.data.BasePage.releaseExclusive(BasePage.
      java:1903): unlatched Page(1,Container(0, 1041))
```

Der Root-Block im B-Baum wird freigeschaltet.

```
9 org.apache.derby.impl.store.raw.xact.RowLocking2.lockRecordForRead(
     RowLocking2.java:166): lockObject (no latch)(type 'S')Record id=1 Page
     (5,Container(0, 1041))
```

Ein shared Lock wird auf den Record vom B-Baum gesetzt.

```
10 org.apache.derby.impl.store.raw.xact.RowLocking3.lockRecordForWrite(
     RowLocking3.java:278): lockObject (no latch)(type 'X')Record id=16
     Page(27,Container(0, 1024))
```

Locking des Row in dem Data Container furs Schreiben. RowLocking3 weisst auf isolation level 3 (serializable) fur die Transaktion.

```
11 org.apache.derby.impl.store.raw.data.BasePage.releaseExclusive(BasePage.
     java:1903): unlatched Page(5,Container(0, 1041))
```

Der Leaf Knoten im B-Baum wird freigeschaltet.

```
12 org.apache.derby.impl.store.raw.data.BasePage.setExclusive(BasePage.java
     :1781): latched. qual: (BaseContainerHandle:(Container(0, 1024)))ref:
     Page(27,Container(0, 1024))
```

Und die Datenseite wird gelatched.

```
13 org.apache.derby.impl.store.raw.xact.RowLocking3.lockRecordForWrite(
     RowLocking3.java:278): lockObject (no latch)(type 'X')Record id=16
     Page(27,Container(0, 1024))
```

Ein anderer exclusive lock fur den Row zu updaten.

```
14 org.apache.derby.impl.store.raw.data.BasePage.releaseExclusive(BasePage.
     java:1903): unlatched Page(27,Container(0, 1024))
```

Und die Datenseite wird freigeschaltet.

```
15 org.apache.derby.impl.store.raw.data.BasePage.setExclusive(BasePage.java
     :1781): latched. qual: (BaseContainerHandle:(Container(0, 1024)))ref:
     Page(27,Container(0, 1024))
```

und noch ein Mal gelatched.

```
16 org.apache.derby.impl.store.raw.xact.RowLocking3.lockRecordForWrite(
     RowLocking3.java:278): lockObject (no latch)(type 'X')Record id=16
     Page(27,Container(0, 1024))
```

Und noch ein Mal wird ein exclusive Lock auf den Daten row gesetzt.

```
17 org.apache.derby.impl.store.raw.data.BasePage.releaseExclusive(BasePage.
     java:1903): unlatched Page(27,Container(0, 1024))
```

Die Datenseite wird freigeschaltet.

```
18 org.apache.derby.impl.store.raw.xact.RowLocking2.
      unlockRecordAfterReadRowLocking2.java:216): unlock (type 'S')Record id
      =1 Page(5,Container(0, 1041))
```

Der shared Lock auf dem B-Baum Row wird freigeschaltet.

```
19 org.apache.derby.impl.store.raw.xact.RowLocking2.unlockContainer(
      RowLocking2.java:248): unlockGroup: BaseContainerHandle:(Container(0,
      1024))
20
21 org.apache.derby.impl.store.raw.xact.RowLocking2.unlockContainer(
      RowLocking2.java:248): unlockGroup: BaseContainerHandle:(Container(0,
      1024))
22
23 org.apache.derby.impl.store.raw.xact.RowLocking2.unlockContainer(
      RowLocking2.java:248): unlockGroup: BaseContainerHandle:(Container(0,
      1024))
```

Alle Locks auf den B-Baum Datencontainer werden freigegeben.

```
24 org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857)
      : unlockGroup: 6751
25
26 org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857)
      :
27 unlockGroup: 6751
28
29 org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857)
      : unlockGroup: 6751
30
31 org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857)
      : unlockGroup: 6751
```

Alle Locks von der Transaktion werden entfernt wie beim Select.

5.3.3 Lock Aktionen in eine single-row Delete Transaktion

```
1 org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(
      RowLocking2.java:104): lockObject w/o latch ContainerKeyIX, Container
      (0, 960)
2
3 org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(
      RowLocking2.java:119): isLockHeld ContainerKeyX,Container(0, 960)
```

Ein IX Lock ist auf dem B-Baum des Data-Container gesetzt und es wird gecheckt ob die Transaktion bereits einen exclusiv Lock gesetzt hatte.

```
4 org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(
      RowLocking2.java:104): lockObject w/o latch ContainerKeyIX, Container
      (0, 960)
```

```
5
6 org.apache.derby.impl.store.raw.xact.RowLocking2.lockContainer(
     RowLocking2.java:119): isLockHeld ContainerKeyX, Container(0, 960)
```

Das selbe wird nochmal gemacht weil ein anderer result set geoffnet wurde.

```
7
8 org.apache.derby.impl.store.raw.data.BasePage.setExclusive(BasePage.java
     :1781): latched a StoredPage page id Page(1,Container(0, 977))
```

Der B-Baum Root knoten wird gelatched.

```
9 org.apache.derby.impl.store.raw.data.BasePage.setExclusive(BasePage.java
     :1781): latched a StoredPage page id Page(6,Container(0, 977))
```

Und dann den Leaf Knoten.

```
10 org.apache.derby.impl.store.raw.data.BasePage.releaseExclusive(BasePage .
     java:1903): unlatched StoredPage page id Page(1,Container(0, 977))
```

Und der Roor wird ungelatched.

```
11 org.apache.derby.impl.store.raw.xact.RowLocking2.lockRecordForRead(
     RowLocking2.java:166): lockObject w/o latch RecordIdS, Record id=1
     Page(6,Container(0, 977))
```

Ein shared Lock auf den record im B-Baum wurde erhalten.

```
12 org.apache.derby.impl.store.raw.xact.RowLocking3.lockRecordForWrite(
     RowLocking3.java:278): lockObject w/o latch RecordIdX, Record id=22
     Page(32,Container(0, 960))
```

Ein exclusive lock auf den Daten Record wurde erhalten.

```
13  org.apache.derby.impl.store.raw.data.BasePage.releaseExclusive(BasePage.
      java:1903): unlatched StoredPagepage id Page(6,Container(0, 977))
```

Die B-Baumseite wird ungelatched.

```
14 org.apache.derby.impl.store.raw.data.BasePage.setExclusive(BasePage.java
     :1781): latched a StoredPage page id Page(6,Container(0, 977))
```

Und nochmal gelatched.

```
15 org.apache.derby.impl.store.raw.data.BasePage.releaseExclusive(BasePage.
     java:1903): unlatched StoredPage page id Page(6,Container(0, 977))
```

Und wieder ungelatched.

```
16 org.apache.derby.impl.store.raw.data.BasePage.setExclusive(BasePage.java
     :1781): latched a StoredPage page id Page(32,Container(0, 960))
```

Die Datenseite wird gelatched.

```
17 org.apache.derby.impl.store.raw.xact.RowLocking3.lockRecordForWrite(
      RowLocking3.java:278): lockObject w/o latch RecordIdX, Record id=22
      Page(32,Container(0,960))
```

Und ein exclusiv Lock wird nochmal erhalten.

```
18 org.apache.derby.impl.store.raw.data.BasePage.releaseExclusive(BasePage.
      java:1903): unlatched StoredPage page id Page(32,Container(0, 960))
```

Die Datenseite wird ungelatched.

```
19 org.apache.derby.impl.store.raw.xact.RowLocking2.unlockRecordAfterRead(
      RowLocking2.java:216): unlock RecordIdRecord id=1 Page(6,Container
      (0,977))
```

Der B-Baum record wird vom shared Lock freigeschaltet.

```
20 org.apache.derby.impl.store.raw.xact.RowLocking2.unlockContainer(
      RowLocking2.java:248): unlockGroup: BaseContainerHandle:(Container(0,
      960))
21
22 org.apache.derby.impl.store.raw.xact.RowLocking2.unlockContainer(
      RowLocking2.java:248): unlockGroup: BaseContainerHandle:(Container(0,
      960))
23
24 org.apache.derby.impl.store.raw.xact.RowLocking2.unlockContainer(
      RowLocking2.java:248): unlockGroup: BaseContainerHandle:(Container(0,
      960))
```

Der Datacontainer wird dreimal unlocked wie beim Update.

```
25 org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857)
      : unlockGroup: 195
26
27 org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857)
      : unlockGroup: 195
28
29 org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857)
      : unlockGroup: 195
30
31 org.apache.derby.impl.store.raw.xact.Xact.releaseAllLocks(Xact.java:1857)
      : unlockGroup: 195
```

Und alle Locks werden viermal freigellasen.

6 Logging und Recovery

Im nachfolgenden Kapitel wird das Thema Logging und Recovery behandelt. Der Schwerpunkt dieses Kapitels liegt bei ARIES, Checkpoints, WAL und dem Logging-Verfahren von Derby. Das Kapitel ist folgendermaßen aufgebaut, zunächst werden die Konzepte allgemein erläutert und anschließend wird auf die Implementierung in Derby eingegangen. Beim Recovery sei zu erwähnen, dass wir lediglich das Thema Crash-Recovery näher betrachten werden.

6.1 ARIES - Algorithms for Recovery and Isolation Exploiting Semantics

Apache Derby verwendet für Logging und Recovery ARIES. ARIES ist eine Familie von Algorithmen zur Wiederherstellung von Datenbanksystemen nach einem Fehlerfall.

ARIES beruht auf drei wichtigen GrundsätzenWIKIPEDIA, Algorithms for Recovery and Isolation Exploiting Semantics. 01 Nov 2009a ⟨URL: http://de.wikipedia.org/wiki/Algorithms_for_Recovery_and_Isolation_Exploiting_Semantics⟩ – Letzter Zugriff am 23.03.2010, wikipedia:aries:

1. Write Ahead Logging (WAL)
   WAL stellt sicher, dass bevor eine physikalische Änderung innerhalb der DB stattfindet, diese zunächst in einem Log protokolliert wird.

2. Wiederholung der gesamten Historie während der Redo-Phase
   Das WAL-Protokoll gewährleistet, dass jede Änderungen von abgeschlossenen sowie unvollständigen Transaktionen protokolliert wurde. Während der Recovery-Phase, wiederholt ARIES sämtliche Log-Einträge, um das System wieder in den gleichen Zustand wie vor dem Absturz zu bringen. Anschließend identifiziert ARIES die zum Zeitpunkt des Crashs aktiven Transaktionen und führt auf ihnen einen Rollback aus.

3. Die Änderungen bringen das System immer voran.
   Sobald eine Transaktion eine Page modifiziert, wird ein Log-Eintrag geschrieben. Dies gilt auch für Rollback-Transaktionen. Während der Undo-Phase wird weiterhin in das Log geschrieben. Dafür gibt es spezielle Logeinträge die sogenannten Compensation Log Records(CLRs).
   Bsp.: Der ursprüngliche Log-Eintrag (nonCLR), beschreibt ein Delete von Zeile 2 auf Page 1. Während der Undo-Phase wird ein Log-Eintrag (CLR) geschrieben, welcher ein Insert von Zeile 2 auf Page 1 beschreibt C. MOHAN, Repeationg History Beyond ARIES - Procedings of 25th International Conference on Very Large Data Bases. 1999 ⟨URL: unbekannt, Pdfliegtvor⟩ – Letzter Zugriff am 25.03.2010, MOHAN S.3. Als

Ergebnis sieht man, dass jede Änderung immer das System voran bringt, obwohl die Änderung aus einer Rollback-Transaktion stammt.

Der ARIES-Recovery-Prozess unterteilt sich in drei Phasen APACHE SOFTWARE FOUNDATION, Derby Logging and Recovery. 2009 ⟨URL: http://db.apache.org/derby/papers/recovery.html⟩ – Letzter Zugriff am 25.03.2010, derbywiki:lar.

1. Analyse Phase
   Bei Analyse-Phase wird das Log vom Anfang bis zum Ende durchlaufen, um die Informationen zu sammeln. Hierbei wird nach einem Startpunkt S für den Redo-Prozess gesucht.
2. Redo Phase
   Die Redo-Phase bringt die Datenbank auf den gleichen Stand wie vor dem Crash. Hierbei wird das Log von S ab vorwärts durchlaufen und jede vorgefundene Aktion wird wiederholt. Jede Transaktion, die vor dem Crash beendet wurde steht nun nicht mehr in der Transaktions-Tabelle.
3. Undo-Phase
   Nach der Redo-Phase stehen lediglich die zum Zeitpunkt des Crashs, nicht abgeschlossenen Transaktionen in der Transaktions-Tabelle. Auf allen wird ein Rollback durchgeführt um das System in einen konsistenten Zustand zu bringen.

6.2 Derby-Recovery-Prozess

Abbildung 6.1 zeigt einen Ausschnitt aller Aktionen innerhalb des Recovery-Prozesses. Dargestellt sind lediglich die wichtigsten Aktionen.

Der Derby Recovery-Prozess wird beim Hochfahren des DBMS gestartet. Im Boot-Prozess wird die RawStore::boot() Methode aufgerufen. Bei diesem Vorgang wird die Log Control-Datei ausgelesen. Sie enthält Information über vorhandene Log-Dateien, sowie die Position (LogInstant) des letzten Checkpoints. Falls bislang noch keine Log-Datei existiert, wird eine neue Datei erstellt. Anhand der LogInstant des Checkpoints, wird mit Hilfe der LogCounter::getLogFileNumber() Methode, die Nummer der der Log-Datei ermittelt. Innerhalb dieser Log-Datei wird der letzte Checkpoint mit Hilfe der Methode LogToFile::findCheckPoint() gesucht. Falls einer gefunden wurde, wird zunächst die UndoLWM (siehe Checkpoints) des Checkpoints ausgelesen. Dies ist der Startpunkt für den Recovery-Prozess. Falls kein Checkpoint existiert, wird das Log von Beginn an durchlaufen.

Während des Recovery, wiederholt Derby die gesamte Log-History, um das System wieder in den gleichen Zustand wie vor dem Crash zu bringen. Derby wiederholt nicht nur die abgeschlossenen Transaktionen, sondern auch die Transaktionen, die zum Zeitpunkt des Absturtzes noch aktiv waren. Diese Phase nennt man Redo-Phase.

Transaktionen die zum Zeitpunkt des Crashs noch nicht beendet waren, werden zurückgesetzt. Das passiert in der Reihenfolge der zuletzt gestarteten Transaktion bis hin zur ältesten. Diese Phase bezeichnet man als Undo-Phase. Für den Recovery-Prozess wird eine eigene Transaktion gestartet. Dies bedeutet, dass alle Redo- und Undo-Operationen innerhalb dieser Transaktion ausgeführt werden. Allerdings ist diese Transaktion keine echte Transaktion und muss dies mitgeteilt bekommen (Xact::recoveryTransaction()). Dies bewirkt, dass sich die Transaktion selbst aus der Transaktionstabelle löscht. Nachfolgend wird diese Transaktion als RecoveryTransaction bezeichnet.

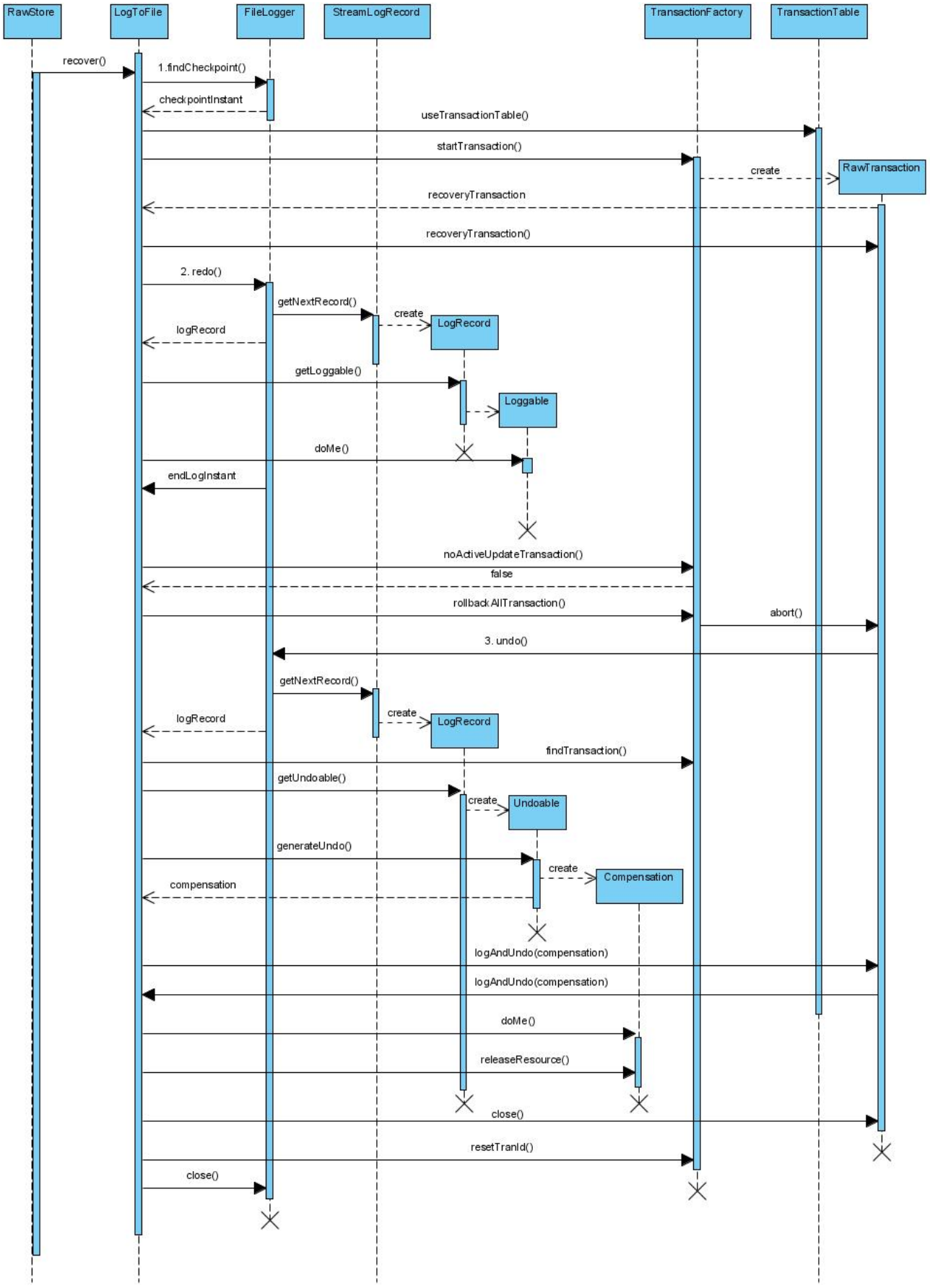

Abbildung 6.1: Apache Derby Recovery-Prozess.

6.2.1 Redo Phase

Während der Redo-Phase werden alle Log-Einträge einzeln mit Hilfe der Funktion StreamLogRecord::getNextRecord() vom Anfang (oder der UndoLWM des Checkpoints) bis zum Ende des Logs vorwärts gelesen. Anhand des LogRecords weiss das System die Transaktions-Id der ursprünglichen Transaktion (innerhalb derer die Operation ausgeführt wurde). Mit Hilfe der Funktion TransactionFactory::findTransaction() wird überpruft, ob diese Transaktions-Id bereits in der Transaktions-Tabelle vorliegt. Falls dort noch kein Eintrag existiert, wird einer angelegt.

Existiert jedoch bereits ein Eintrag, bleibt dieser zwar erhalten, allerdings nimmt die RecoveryTransaction die Identität der eingetragenen Transaktion an. D.h., dass ab sofort sämtliche Referenzen (der ursprünglichen Transaktion), in der Tranaktions-Tabelle auf die RecoveryTransaction verweisen.

Anschließend wird das Operations-Objekt aus dem Record mit Hilfe der Methode getLoggable() ausgelesen. Mit diesem Operations-Objekt wird überprüft, ob die Operation wiederholt werden muss. Wenn die Operation wiederholt werden muss, gibt es 2 Fälle für die Art der Wiederholung:

1. Der LogRecord ist NON-CLR
   Falls der LogRecord kein CLR ist, wird die doMe() Methode aufgerufen, und die Operation wird wiederholt.

2. Der LogRecord ist ein CLR
   Falls der LogRecord ein CLR ist, dann muss zuerst die zugehörige Undoable-Operation erzeugt werden. Anschließend wird deren doMe() Method aufgerufen und die Änderung wird rückgängig gemacht.

Am Ende wird der LogRecord darauf überprüft, ob es der letzte LogRecord der Transaktion ist (End-Transaction Log-Eintrag). Ist dies der Fall, wird die RawTransaction::commit() Methode der Transaktion aufgerufen, und die Transaktion wird bestätigt.

Am Ende des Redo-Prozesses wird die nächste LogInstant der Log-Datei zurückgegeben und in LogCounter gespeichert.

6.2.2 Undo Phase

Bei der Undo-Phase wird die Transaktion-Tabelle verwendet, welche in der Redo-Phase gefüllt wurde. Sie wird mit Hilfe der Funktion TransactionFactory::noActiveUpdateTransaktion() überprüft, dabei wird überprüft ob es eine aktive Transaktion in der Transaktions-Tabelle gibt. Gibt es eine aktive Transaktion, wird TransactionFactory::rollbackAllTransaction() aufgerufen und die aktiven Transaktionen werden mit Hilfe ihrer abort() Methode rückgängig gemacht.

Während dem abort() Aufruf wird die FileLogger::undo() Methode aufgerufen. Hier werden alle LogRecords mit Hilfe der Funktion StreamLogRecord::getNextRecord() vom letzten LogRecord bis zum BeginTransaction-LogRecord einer Transaktion rückwärts gelesen.

Anschließend wird aus dem LogRecord mit Hilfe der Methode Loggable::getUndoable() eine Undoable-Operation generiert. Eine Undoable-Operation ist eine Operation, die den Zustand eines RawStores im Kontext einer Transaktion geändert hat. Diese Änderung lässt sich zurückrollen.

Mit diesem Undoable-Operation-Objekt wird Undoable::generateUndo() aufgerufen. Wenn diese Methode ?null? zurückgibt, kann man nichts rückgängig machen. Wenn generateUndo eine Compensation-Operation zurückgibt, so loggt das System den CLR mit Hilfe FileLogger.logandUndo() und ruft danach die Compensation::doMe() Methode auf. Nachdem die CLR angewendet worden ist, ruft das System compensation.releaseResource() auf.

6.3 Checkpoints

Checkpoints dienen zur Minderung des Aufwands während des Recovery-Prozesses. Nach einem Crash müssen sämtliche Dirty-Pages wiederhergestellt werden, da die Änderungen zum Zeitpunkt des Crashs nur im DB-Puffer allerdings noch nicht in der physikalischen DB vorhanden waren. Dies geschieht innerhalb der Redo-Phase im Recovery-Prozess. Probleme bereiten hierbei die hochfrequentierten Pages, welche aufgrund ihrer hohen Frequenz, selten beim Flushen des Puffers freigegeben werden. D.h., es könnte Notwendig sein, sämtliche Änderungen seit Beginn der DB zu wiederholen was sehr lange dauern könnte. Ausserdem würde dadurch ein sehr großes Log entstehen.

Aus diesen Gründen verwendet man Checkpoints, um somit die zu wiederholenden Aktionen und gleichzeitig die Größe des Logs zu Begrenzen. Checkpoints lassen sich in direkte und indirekte Checkpoints aufteilen. Bei den direkten wird der komplette DB-Puffer zum Zeitpunkt des Checkpoints auf den physikalischen Speicher geschrieben. Solange finden keine Änderungsaktionen statt. Deshalb sind direkte Checkpoints sehr zeitaufwendig, bieten allerdings den Vorteil, dass beim Redo-Prozess direkt hinter dem Checkpoint gestartet werden kann. Die vorherigen Log-Einträge sind somit uninteressant. Man unterscheidet zwischen transaktionsorientierten-, transaktionskonsistenten- und aktionskonsistenten-Checkpoints.

Indirekte Checkpoints (Fuzzy Checkpoints) beinhalten nur wenige Statusinformationen wie z.B.: laufende Transaktionen und Dirty-Pages (Hinweis auf die Page). Das schreiben der Dirty-Pages erfolgt asynchron zum Schreiben des Checkpoints. D.h., das Ausschreiben der Pages kann zeitversetzt zum Ausschreiben des Checkpoints erfolgen. Bei diesem Verfahren kann das System nach dem Logeintrag mit der Verarbeitung von Transaktionen fortfahren und muss nicht auf das Schreiben der Pages warten [ELMASRI S. 744]. Dadurch dass kein kompletter Flush des Puffers erfolgt, befindet sich die physikalische DB weder in einem Aktions- oder Transaktionskonsistenten Zustand. Stattdessen befindet sie sich in einem unscharfen (fuzzy) Zustand [HAERDER S. 475].

6.4 Checkpoint-Implementierung

Checkpoints werden in Derby durch die Klasse CheckpointOperation repräsentiert. Derby verwendet das Konzept der Fuzzy-Checkpoints. Wenn ein Checkpoint geschrieben wird,

werden alle Dirty-Pages automatisch geflushed. D.h., nach einem Checkpoint existieren keine Dirty Pages im Puffer mehr.

Für die Erstellung der Checkpoints ist ein Daemon, welcher im Hintergrund läuft, verantwortlich. Dieser Daemon wird innerhalb der LogToFile::recover() Methode gestartet und erhält eine Referenz auf das LogToFile-Objekt. Bevor der Log-Puffer in die Log-Datei geschrieben wird (innerhalb von LogToFile::flush()), wird überprüft ob die Anzahl der Bytes die zwischen zwei Checkpoints liegen darf, überschritten wurde. Falls dies der Fall ist, wird beim Daemon ein Flag gesetzt, welches ihn dazu veranlasst, LogToFile::performWork() aufzurufen und darin dann RawStore::checkpoint().

Bevor ein Checkpoint geschrieben wird, wird mittels DataFactory::checkpoint() (innerhalb von LogToFile::checkpointWithTran()), sämtliche Dirty-Pages auf die Festplatte geschrieben. Nachdem der Checkpoint geschrieben wurde, wird dessen LogInstant in die Log-Control-Datei geschrieben, damit er beim nächsten Recovery-Prozess verwendet werden kann um nicht alle Log-Einträge analysieren zu müssen. Derby spart sich so, die Analyse-Phase im ARIES-Algorithmus. Jeder Checkpoint wird als eigene Transaktion in das Log geschrieben. D.h., zu jedem Checkpoint befinden sich jeweils ein umschließender Begin- und End-Transaction Log-Eintrag.

Ein CheckpointOperation-Objekt beinhaltet die folgenden Informationen:

- Transaktions-Id der umschließenden Transaktion
- RedoLWM
- UndoLWM
- Tranaktions-Tabelle

Die RedoLWM ist immer die LogInstant des Checkpoint-Log-Eintrags selbst. Die UndoLWM und die Tranaktions-Tabelle erhält der Checkpoint von der TransactionFactory (firstUpdateInstant() und getTransactionTable() Methode). Die UndoLWM ist die LogInstant (der Begin-Transaction Log-Eintrag) der ältesten aktiven Transaktion. Dieser Log-Eintrag ist der Startpunkt für den Redo-Prozess. Zur Transaktionstabelle sei gesagt, dass diese zwar mit dem Checkpoint in das Log geschrieben wird, allerdings während des Recovery-Prozesses nicht berücksichtigt wird.

Nach einem Checkpoint entsteht folgende Situation im Log (siehe Abb.). Die RedoLWM zeigt, dass an diesem Punkt der Checkpoint erstellt wurde. Die UndoLWM ist die BeginTransaction-LogInstant der am längsten andauernden Transaktion. Der Checkpoint selbst wird erst später in das Log geschrieben. Durch diesen Vorgang entstehen 3 interessante Abschnitte.

- Transaktion 1 startet in Abschnitt A und endet in B. Während des Redo werden nur Operations-Log-Einträge und ein EndTransaction-Log-Eintrag vorgefunden. Alle vorgefunden Operationen werden im Redo wiederholt. Für den Undo-Prozess ist T1 nicht relevant.
- T2 startet in B und endet in C. Beim Redo-Prozess werden BeginTransaction-, Operation- und EndTransaction-Log-Einträge vorgefunden. Die Transaktion wird komplett wiederholt. Von der Undo-Phase bleibt diese Transaktion unberührt.
- T3 startet und endet in B. Sie verhält sich genau wie T2 während des Recovery-Prozess.

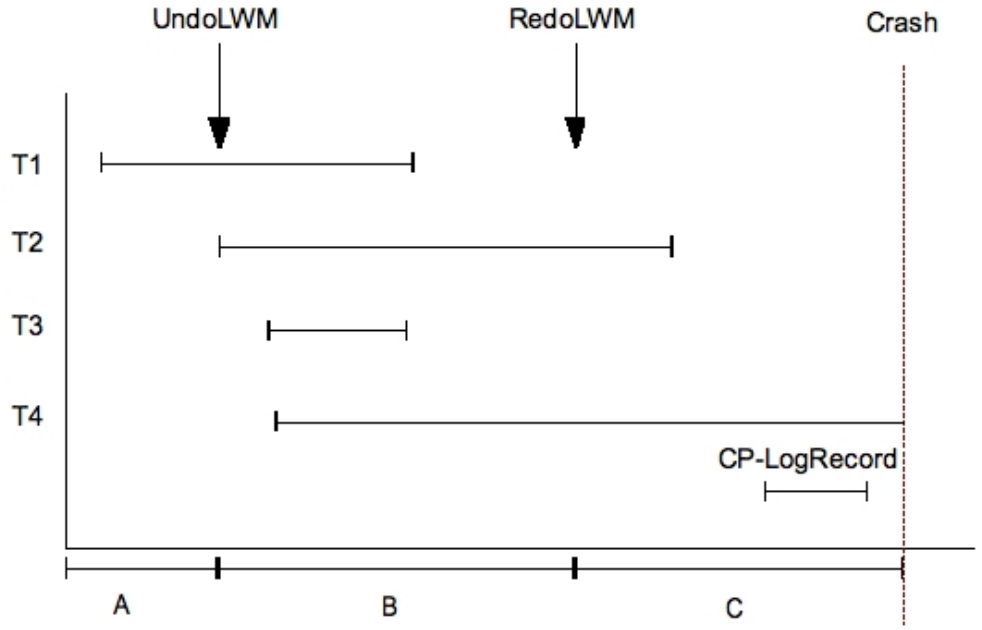

Abbildung 6.2: Ablauf eines Checkpoints.

- T4 startet in B und endet nie. Zum Zeitpunkt des Crashs war sie noch aktiv. Am Ende der Redo-Phase ist sie die einzige Transaktion die in der Transaktions-Tabelle übrig bleibt und sich einem Rollback unterziehen muss.

6.5 Write Ahead Logging und Commit-Regel

Write Ahead Logging ist ein Verfahren, welches zur Atomarität und Dauerhaftigkeit einer Transaktion beiträgt. WAL fordert, dass bevor eine physische Änderung an der Datenbank stattfindet, ein Eintrag in das Log gemacht werden muss. Dieser Log Eintrag muss Informationen über eine mögliche Undo-Operation enthalten. Diese Undo-Operation könnte z.B., ein Before-Image sein. Diese Informationen stellen sicher, dass eine bereits ausgeführte Transaktion während des Recovery Prozesses erneut ausgeführt werden kann.

Analog zu WAL gibt es die Force-Log-at-Commit-Regel (Commit-Regel), welche für die Redo-Recovery relevant ist. "Diese besagt, dass vor dem Commit einer Transaktion für ihre Änderung ausreichende Redo-Informationen zu sichern sind". Dadurch wird die Wiederholbarkeit und Dauerhaftigkeit sichergestellt[HAERDER S. 468]HAERDER.

6.6 Implementierung von WAL und der Commit-Regel

Nachfolgend wird anhand eines Szenarios die Implementierung von WAL und der Commit-Regel in Derby beschrieben. Szenario:

Nach dem Hochfahren der Datenbank wird der Recovery-Prozess gestartet. Nach Beendigung der Redo-Phase sollen alle noch aktiven Transaktionen rückgängig gemacht werden.

Hierbei wird innerhalb der LogToFile::recovery() Methode, die XactFactory::rollbackAllTransactions() Methode aufgerufen. Diese veranlasst jede noch aktive Transakton ihre abort() Methode auszuführen. Innerhalb dieser Methode wird FileLogger::undo() aufgerufen. Dieser Methode werden die LogInstant des ersten und des letzten Log-Eintrags der Transaktion übergeben, sowie eine Referenz auf die Tranaktion selbst.

FileLogger liest daraufhin das Log Rückwärts aus und überprüft ob der gerade ausgelesene Log-Eintrag zur Transaktion gehört. Falls dies der Fall ist, wird überprüft, ob der Log-Eintrag ein CLR ist. CLRs benötigen kein Undo und werden deshalb verworfen. Wenn der vorgefundene LogRecord kein CLR ist, wird dessen getUndoable() Methode aufgerufen, welche als Ergebnis eine Kompensations-Operation liefert. Diese wird dann an die logAndDo() Methode der Transaktion übergeben. Diese ruft wiederum die logAndUndo() Methode von FileLogger auf. Man beachte, bislang wurde die doMe() Methode der Kompensations-Operation noch nicht aufgerufen, d.h., bislang hat noch keine Änderung stattgefunden. Innerhalb der logAndUndo() Methode wird zunächst das Kompensations-Operation-Objekt an das Log angehängt, bevor die doMe() Methode der Operation aufgerufen wird.

6.7 Klassifizierung von Logging-Verfahren

Logging-Verfahren lassen sich nach drei Arten klassifizieren, physisches, logisches und physiologisches Logging.

6.7.1 Physisches Logging:

Hierbei beziehen sich die geloggten Informationen auf die physische Ebene. D.h., es werden Logeinträge wie z.B. eine Page vor der Änderung oder physikalische Datensätze gespeichert. Die Logeinträge finden z.B., auf der Ebene von Seiten statt. Hierbei bezieht sich das Logging auf eine bestimmte Byte-Position innerhalb einer Seite. Physikalisches Logging hat den Nachteil dass der Verwaltungsaufwand sehr hoch ist. Jede Byte Veränderung innerhalb der Seite würde einen Log-Eintrag mit sich bringen, sodass der Logging-Aufwand sehr hoch wäre.

Physisches Logging kann man in Zustands-Logging und Übergangs-Logging unterteilen. Beim Zustands-Logging werden die Zustände der Objekte (z.B. eine Page) vor und nach der Änderung als Before-Image und After-Image geloggt. Before-Images werden beim Undo-Prozess verwendet und After-Images beim Redo-Prozess.

Übergangs-Logging loggt eine Differenz der beiden Zustände Before- und After-Image. Anhand der Differenz kann man aus dem After-Image Zustand wieder zurück zum Before-Image Zustand gelangen[HAERDER S. 461]HAERDER.

6.7.2 Logisches Logging:

Logisches Logging ist eine Form des Übergangs-Logging. Hiebei werden logische Operationen in das Log gespeichert. Eine logische Operation kann z.B. eine Insert oder Update Operation sein. Jeder Log- Eintrag enthält die Operation mit all seinen Eigenschaftswerten.

Der Vorteil dieser Logging-Variante ist, dass der Aufwand im Vergleich zum Physikalischen-Logging wesentlich geringer ist. Probleme entstehen allerdings beim Undo-Crash-Recovery. Hierbei muss die Datenbank zunächst in einen aktionskonsistenten Zustand gebracht werden, um anschließend die geloggten Operationen ausführen zu können. Dies bedeutet, dass jede vorher ausgeführte logische Operation ganz oder gar nicht in der physikalischen DB abgebildet werden muss, bevor die im Log stehende Logische Operation rückgängig gemacht werden kann. Beim ARIES Algorithmus ist dies der Fall. ARIES bringt die physikalische DB zunächst in einen aktionskonsistenten Zustand (Redo-Phase), bevor die Undo-Phase beginnen kann[HAERDER S. 462]HAERDER.

6.7.3 Physiologisches Logging:

Physiologisches Logging ist ein Mix aus logischem und physischem Logging. Man sagt auch "physical to a page and logical within a page"THEO HÄRDER; ERHARD RAHM, Datenbanksysteme: Konzepte und Techniken der Implementierung. Springer Verlag, 1999, HAERDER S. 463. Die in das Log zu schreibenden Einträge sind logische Operationen, d.h., es wird nicht byteorientiert geloggt. Jedoch beziehen sich die logischen Einträge immer auf eine bestimmte Page.

Im Vergleich zum rein logischen Logging, muss die DB nach einem Crash keinen aktionskonsistenten Zustand aufweisen. Jedoch müssen die Pages in Bezug auf logische Operationen, konsistent sein. Dies bedeutet, dass jede logische Operation innerhalb einer Seite ganz oder gar nicht ausgeführt werden muss[HAERDER S. 463]HAERDER.

6.8 Logging-Verfahren in Derby

Derby verwendet physiologisches Logging. In der Regel beziehen sich sämtliche Operationen auf eine Page. D.h., es wird eine physiologische Operation innerhalb einer Page ausgeführt z.B., eine Update-Operation auf eine Zeile. Will man diese Update-Operation rückgängig machen, so ist lediglich eine physische Undo-Operation notwendig, da die Änderung lediglich eine Page betrifft. Im Gegensatz dazu stehen die Insert- und Delete-Operation. Beide sind rein logische Operationen, da eine mögliche Undo-Operation sich auf mehrere Pages auswirken kann. Dies könnte durch ein splitten der Page im B-Baum passieren [Derby-Doku: Logging u. Recovery]. Der Unterschied zwischen physiologischer und logischer Operation macht sich in Ableitungshierarchie der Operations-Klassen bemerkbar (siehe Klassendiagramm).

Für die Erzeugung der Operationen und deren Ausführung ist die Klasse LoggableActions verantwortlich. Sie besitzt für jede logbare Operation (damit sind nicht die Kompensations-Operationen gemeint) eine Methode z.B.: LoggableActions::actionUpdate() erzeugt ein UpdateOperation-Objekt und übergibt es an die logAndDo() Methode der Transaktion. Kompensations-Operationen können lediglich über die generateUndo() Methode des Undoable-Interface erhalten werden. Die beiden Klassen PhysicalUndoOperation und LogicalUndoOperation implementieren dieses Interface. Wie man im Diagramm erkennen kann, besitzen beide keine undoMe() Methode, da sie Kompensations-Operationen sind und somit kein Undo benötigen. Jede Kompensations-Operationen kann man sich wie einen Wrapper für die jeweilige Operation vorstellen.

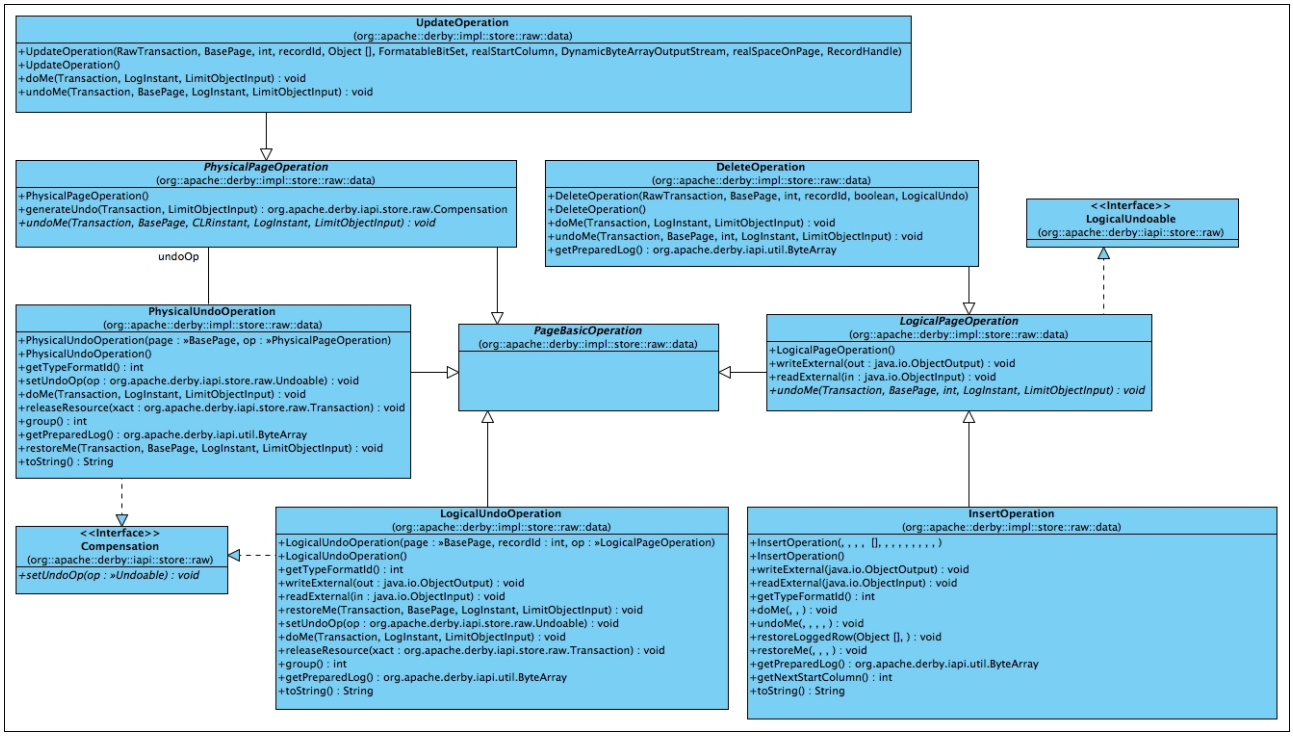

Abbildung 6.3: Ableitungshierarchie der Operationen

Bsp.: Eine UpdateOperation muss rückgängig gemacht werden. Hierzu wird die generateUndo() Methode der abstrakten Oberklasse (PhysicalPageOperation) aufgerufen. Diese erzeugt ein PhysicalUndoOperation-Objekt und übergibt dabei eine Referenz auf sich selbst. Anschließend wird die doMe() Methode des Undo-Objekts aufgerufen. Diese ruft wiederum die undoMe() Methode von PhysicalPageOperation auf. Diese ist allerdings abstakt und delegiert somit den Aufruf an die undoMe() Methode der UpdateOperation weiter.

Führt man eine SQL Update-Anweisung aus, so wird genau ein Log-Eintrag für diese Anweisung geschrieben. Nachfolgend kann man sich die String-Ausgabe eines InsertOperation-Log-Eintrags anschauen.

```
Transaction-ID :387 org.apache.derby.impl.store.raw.xact.XactId
Loggable :Page Operation: Page(20,Container(0, 1040)) pageVersion 73 :
   Insert :  Slot=36 recordId=42 org.apache.derby.impl.store.raw.data.
   InsertOperation
Loggables-Group : 384
```

Die InsertOperation gehört zur Transaktion mit der Id 387. Diese logische Operation wird innerhalb der Page mit Id 20 ausgeführt. Die neue Record-Id ist 42. Nach der Ausführung beträgt die neue Page-Version-Number 73

6.9 Aufbau des Logs und der Log-Einträge

6.9.1 Das Log

Im Log stehen sämtliche Informationen, die für den Recovery-Prozess benötigt werden um die Datenbank in einen konsistenten Zustand zurückzuführen. Das "Log" besteht aus den einzelnen Log-Dateien (.dat) und einer Verwaltungsdatei (.ctrl). Jeder RawStore besitzt ein eigenes Log. Das Log wird von der LogFactory verwaltet.

Log-Datei (.dat): Der Namen der Log-Datei besteht aus einem Prefix “log”, einer fortlaufenden Nummer und dem Suffix “dat”. Zum Beispiel: log1.dat, log2.dat, usw. Ein Wechsel einer Logdatei von einer Nummer auf die nächste wird anhand des LogToFile::LOG_SWITCH_INTERVAL Attributs bestimmt. Dieses Attribut gibt die Größe der Log-Datei an. Per Default ist jede 1MB groß. Der Benutzer kann aber auch den Wert von LOG_SWITCH_INTERVAL selbst einstellen. Jeder Log-Eintrag hat ein eindeutige Nummer. Diese Nummer bezeichnet man als Log Sequence Number (LSN). Wird ein neuer Log-Eintrag eingetragen, wird die LSN inkrementiert. In Derby Terminologie wird LSN als Log-Instant bezeichnet. LogCounter ist die Klasse, die LogInstant implementiert. Ein LogInstant besteht aus der Nummer der Logdatei und der Position innerhalb der Log Datei.

Log-Kontroll-Datei (.ctrl): Wenn die Datenbank nach einem Fehler neue gestartet muss, verwendet das Logging-System die gespeicherten Informationen aus Log-Dateien um die Datenbank in einen konsistenten Zustand zu bringen. Die Datenbank-Manager verwendet die sogenannte Log-Kontroll-Datei, welche die fogende informationen erhält :

1. Welche log-Datei vorhanden sind
2. Wo die letzte Checkpoint Log-Einträge sich befindet

Wenn die Log-Kontroll-Datei beschädigt ist, ist es nicht mehr möglich für die Datenbank-Manager, die Datenbank in einen konsistenten Zustand zu versetzen. Um diese zu vermeiden, ermittelt Derby andere Kopie von Log-Kontroll-Datei die sogenannte logmirror.ctrl.

6.9.2 Log-Einträge

Jeder Log-Eintrag beschreibt eine Änderungs-Operation oder Verwaltungsinformationen (z.B., ein Checkpoint-Log-Eintrag) innerhalb der DB. Ein Log-Eintrag in Derby unterteilt sich in einen Anteil fester Länge und einen variablen Anteil. Nachfolgend wird ein Eintrag für eine Update-Operation skizziert:

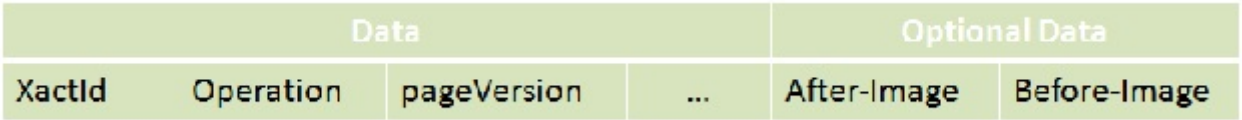

Abbildung 6.4: UpdateOperation-Log-Eintrag.

Der Data-Teil ist fest, d.h., jeder Update-Log-Eintrag schreibt die gleichen Informationen in das Log. Die Transaktions-Id der zugehörigen Transaktion, die neue PageVersion-Nr, die

Record-Id des Datensatzes und den Slot. Diese Daten werden innerhalb der writeExternal() Methode in einen Puffer geschrieben. Der optionale Anteil hat keine feste Länge. Bei der UpdateOperation stehen hier das Before- und After-Image.

Abbildungsverzeichnis

Listings

Literaturverzeichnis

Couvering, David van: Apache Derby Status Update. 2005 ⟨URL: http://db.apache.org/derby/binaries/DerbyBOF-OSCON-2005.pdf⟩ – Letzter Zugriff am 25.03.2010.

Debrunner, Daniel John: Internals of Derby, An Open Source Pure Java Relational Database System. 2004 ⟨URL: http://db.apache.org/derby/binaries/ApacheDerbyInternals_1_1.pdf⟩ – Letzter Zugriff am 25.03.2010.

Foundation, Apache Software: Derby Engine Architecture Overview. 2008 ⟨URL: http://db.apache.org/derby/papers/derby_arch.html⟩ – Letzter Zugriff am 25.03.2010.

Foundation, Apache Software: Derby Logging and Recovery. 2009 ⟨URL: http://db.apache.org/derby/papers/recovery.html⟩ – Letzter Zugriff am 25.03.2010.

Foundation, Apache Software: Derby Network Server. 2009 ⟨URL: http://db.apache.org/derby/papers/DerbyTut/ns_intro.html⟩ – Letzter Zugriff am 25.03.2010.

Foundation, Apache Software: Derby Write Ahead Log Format. 2009 ⟨URL: http://db.apache.org/derby/papers/logformats.html⟩ – Letzter Zugriff am 25.03.2010.

Mohan, C.: Repeationg History Beyond ARIES - Procedings of 25th International Conference on Very Large Data Bases. 1999 ⟨URL: unbekannt, Pdfliegtvor⟩ – Letzter Zugriff am 25.03.2010.

Navathe, Ramez A. Elmasri; Shamkant B.: Grundlagen von Datenbanksystemen. Pearson Studium, 2002.

Orsini, Francois: Apache Derby Case Study: Benefits of a Microkernel architecture. 2008 ⟨URL: http://blogs.sun.com/FrancoisOrsini/entry/apache_derby_and_the_benefits⟩ – Letzter Zugriff am 25.03.2010.

Rahm, Theo Härder; Erhard: Datenbanksysteme: Konzepte und Techniken der Implementierung. Springer Verlag, 1999.

Schnieder, Marc: JDBC. 1997 ⟨URL: http://www-vs.informatik.uni-ulm.de:81/Lehre/Seminar_Java/ausarbeitungen/JDBC/JDBC.html#1⟩ – Letzter Zugriff am 23.03.2010.

Wikipedia: Algorithms for Recovery and Isolation Exploiting Semantics. 01 Nov 2009a ⟨URL: http://de.wikipedia.org/wiki/Algorithms_for_Recovery_and_Isolation_Exploiting_Semantics⟩ – Letzter Zugriff am 23.03.2010.

Wikipedia: Distributed Relational Database Architecture — Wikipedia, Die freie Enzyklopädie. 2009 ⟨URL: http://de.wikipedia.org/w/index.php?title=Distributed_Relational_Database_Architecture&oldid=55145926⟩ – Letzter Zugriff am 23.03.2010.

Wikipedia: Java Database Connectivity — Wikipedia, Die freie Enzyklopädie. 01 Nov 2009c ⟨URL: http://de.wikipedia.org/w/index.php?title=Java_Database_Connectivity&oldid=66294918⟩ – Letzter Zugriff am 23.03.2010.