Studiengang:	**Informatik**
Betreuer:	**Dipl. Inf. Matthias Großmann**
Prüfer:	**Professor Dr. B. Mitschang**
Beginn am:	**01.12.03**
Beendet am:	**31.05.04**
CR-Klassifikation:	**H. 2.1, H. 2.8, H. 3.5**

Diplomarbeit Nr. 2163

Anfrageoptimierung im Spatial Model Server

Thomas Wahl

Fakultät Elektrotechnik, Informatik,
Informationstechnik
Universität Stuttgart
Institut für Parallele und Verteilte Systeme (IPVS)
Abteilung Anwendersoftware (AS)
Universitätsstr. 38
70569 Stuttgart

Kurzfassung

Gegenstand dieser Diplomarbeit ist die Optimierung der Anfrageverarbeitung im Spatial Model Server. Es soll ein neues Datenbankschema gefunden werden, das eine effizientere Anfrageverarbeitung erlaubt und die Mehrfachvererbung von Objekttypen unterstützt. Anfragen können in einer speziellen Anfragesprache namens Augmented World Query Language (AWQL) formuliert und über die SOAP-Schnittstelle an den Server gesendet werden. Die Resultate dieser Anfragen werden je nach Anfragetyp entweder als Dokument in der Sprache Augmented World Modeling Language (AWML) oder Change Report Language (CRL) formuliert und zurückgegeben. Als Backend wird eine objektrelationale Datenbank verwendet, welche als Speicher für die Nexusobjekte dient. Die verwendete DB-Sprache ist standardmäßig SQL. Die Hauptaufgabe des Spatial Model Servers liegt darin, Anfragen in AWQL nach SQL umzusetzen, diese auf der DB auszuführen und das Resultat anschließend nach AWML bzw. CRL umzuformen. Im Zuge der Neuimplementierung soll ein geeignetes Datenbankschema gefunden werden, das es erlaubt, Mehrfachvererbung auf Anwendungsebene zu simulieren.

Abstract

Subject of this diploma thesis is the query optimization in the spatial model server. A new database schema is to be found that allows more efficient query processing and that supports multiple inheritance of object types. Queries can be formulated in a special query language called Augmented World Query Language (AWQL) and be sent via the SOAP interface to the server. The results of these queries are rewritten either in a language called Augmented World Modeling Language (AWML) or Change Report Language (CRL) according to the type of query and sent back to the caller. For the backend system an object-relational database is used to store and retrieve so called static objects of the nexus world. The standard database query language is SQL. Main task of the spatial server is therefore to translate incoming queries from AWQL to SQL, to send them to the database and to rewrite the results to AWML or CRL. For the new implementation of the spatial model server an adequate database schema is to be found, that supports the simulation of multiple inheritance in the application layer for databases with only simple inheritance.

Inhaltsverzeichnis

1 Einleitung

1.1 Hintergrund

Nexus ist der Titel eines Sonderforschungsbereichs der Deutschen Forschungsgemeinschaft, der sich mit der Entwicklung einer Plattform für mobile, ortsbezogene Anwendungen beschäftigt [NGS+01]. Im Rahmen dieser Forschung werden Methoden und Verfahren zur Realisierung von Umgebungsmodellen für mobile kontextbezogene Systeme entwickelt. Dazu werden Objekte der realen Welt gemäß eines Schemas, dem Augmented World Schema (AWS), beschrieben und von den einzelnen Komponenten des Systems verwaltet. Alle Objekte zusammengenommen bilden das so genannte Augmented World Model (AWM).

Die statischen Objekte werden von so genannten Spatial Model Servern, die mobilen von Location Servern verwaltet. Eine Föderationskomponente vermittelt einzelne Anfragen nach statischen oder mobilen Objekten an die entsprechenden Dienste. Diese bieten u.a. die Möglichkeit, nach Objekten in einem bestimmten geografischen Gebiet zu suchen.

Diese Diplomarbeit befasst sich mit der Optimierung der Anfrageverarbeitung im Spatial Model Server, d.h. mit der Umsetzung von Anfragen nach SQL, und der Entwicklung und Umsetzung eines geeigneten Datenbankschemas für die Speicherung der Nexusobjekte in der Datenbank.

1.2 Aufgabenbeschreibung

Im Rahmen dieser Diplomarbeit soll die Komponente des Spatial Model Servers (SpaSe), welche für die Übersetzung von AWQL nach SQL zuständig ist, gegen eine neu zu implementierende ausgetauscht werden, die effizienter arbeitet und als zukünftige Erweitungen der Objektverwaltung Mehrfachvererbung unterstützt. Es sollen folgende Fragestellungen untersucht werden:

- Wie lassen sich auf effiziente Weise aus einer AWQL-Anfrage die betroffenen Datenbanktabellen ermitteln?
- Lassen sich auch bei Mehrfachvererbung objektrelationale Konzepte nutzen?
- Ist es sinnvoll, mehrere SQL-Anfragen zu kombinieren, so dass auf Kosten eines höheren Datenvolumens die Zahl der Anfragen gesenkt werden kann?
- Ist es sinnvoll, Tabellen mit Zusatzinformation anzulegen, um die Zahl der zur Beantwortung einer Anfrage zu berücksichtigenden Datenbanktabellen zu reduzieren?

Die Diplomarbeit umfasst die theoretische und experimentelle Bewertung dieser und gegebenenfalls weiterer Optimierungsmöglichkeiten sowie die Integration in den SpaSe. Als Programmiersprache wird Java™ eingesetzt.

1.3 Gliederung

Die Ausarbeitung gliedert sich wie folgt: Zunächst werden in Kapitel 2 einige verwandte Arbeiten vorgestellt, die sich allgemein mit der Abbildung eines komplexen Objektmodells auf ein objektrelationales Datenmodell beschäftigen. Da das Nexus-System als Grundlage dient, werden zunächst in Kapitel 3 die verschiedenen Dienste dieser Plattform vorgestellt. In Kapitel 4 werden das eigentliche Problem und die Motivation zu dieser Arbeit eingehend behandelt. In Kapitel 5 folgt die Vorstellung der gefundenen Lösungsansätze, die anschließend in Kapitel 6 theoretisch bewertet werden. In Kapitel 7 folgt die Beschreibung der Implementierungen einiger ausgewählter und viel versprechender Ansätze. Abgeschlossen wird der Bericht mit der praktischen d.h. experimentellen Bewertung in Kapitel 8 in Form von Performancetests, die auf den implementierten Spatial Model Servern durchgeführt wurden. Die dafür verwendeten Testfälle befinden sich im Anhang dieses Dokuments. In Kapitel 9 folgen eine Zusammenfassung sowie ein kurzer Ausblick auf mögliche Erweiterungen des gefundenen Ansatzes.

1.4 Danksagungen

Eine Diplomarbeit entsteht in den seltensten Fällen ohne die Unterstützung von anderen Personen. Mein Dank gilt daher besonders:

- Matthias Großmann für die kompetente Betreuung durch alle Phasen der Arbeit hinweg.
- Professor Bernhard Mitschang für die Aufgabenstellung und die Vorbereitung durch seine Vorlesungen.
- Den Mitarbeitern der Abteilung Anwendersoftware, insbesondere Thomas Schwarz, der mir bei kuriosen Datenbankproblemen eine wertvolle Hilfe war.
- Meinen Kommilitonen, die während der Entstehung dieser Arbeit mit viel Engagement, Witz und Humor für anregende Abwechslung sorgten.
- Serena und Michael, meine beiden Korrekturleser, ohne die einige kritische Schreibfehler unentdeckt geblieben wären.

2 Verwandte Arbeiten

Dieses Kapitel befasst sich mit einigen verwandten Arbeiten zu diesem Thema. Es konnte nur ein Projekt ausfindig gemacht werden, dass sich intensiv mit dem Problem der Simulation von Mehrfachvererbung auf objektrelationalen Datenbanken beschäftigt. Daher wurde dieser Arbeit ein eigenes Unterkapitel gewidmet (siehe 2.1).

In der Diplomarbeit von Malte Finsterwalder [Fin02] wird das Thema der Abbildung eines Objektmodells einer objektorientierten Software auf das Datenmodell der Datenbank behandelt. Dabei werden die Konzepte des relationalen und des objektrelationalen Modells gegenübergestellt und auf Tauglichkeit überprüft. Anhand eines Prototyps wird demonstriert, wie die objektrelationalen Erweiterungen der Datenbank für die Umsetzung genutzt werden können. Auf das Problem der Mehrfachvererbung wird, im Gegensatz zu der hier vorgestellten Arbeit, jedoch nicht weiter eingegangen. Es werden ausschließlich die von der Datenbank bereit gestellten Vererbungsmechanismen eingesetzt.

In [KJA93] wird selbiges für relationale Datenbanken umgesetzt. Es wurde ein Entwicklungswerkzeug realisiert, das aus einem sog. Database Interface Generator besteht, der ein Objektmodell einer Anwendung in C++ Klassen und relationale Tabellen umsetzen kann, und einem Runtime Object Management System, das für den schnellen und konsistenten Datenbankzugriff im Mehrbenutzerbetrieb sorgt.
Über ein sog. Persistence Interface kann der Benutzer das Objektmodell eingeben. Für jede Klasse können Vererbung und Primärschlüsselattribute der entsprechenden Tabelle festgelegt werden. Es ist nur die einfache Vererbung von Klassen erlaubt. Über Fremdschlüsselattribute werden Verbindungen zwischen den Klassen hergestellt. Aus diesen Eingaben werden automatisch die entsprechenden C++-Header und Klassendateien erzeugt (Getter- und Setter-Methoden) sowie die Datenbanktabellen. Das Runtime System greift transparent auf die Datenbank zu und erlaubt durch effizientes Prefetching, Pointer-Swizzling und Caching von Objekten schnelle In-Memory Navigation zwischen Objektinstanzen.

In [VV01] wird allgemein beschrieben, wie XML als Datenaustauschformat zur Generierung von Datenbankschemata und für die Anfrageformulierung verwendet werden kann. Es wurde eine relationale Datenbank namens X-Database entwickelt, die Anfragen im XML-Format entgegen nimmt und diese nach SQL umwandelt. Das Datenbankschema kann bei diesem System aus einem XML-Schema erzeugt werden. Dieses Schema dient, im Gegensatz zum SpaSe, anschließend auch zur Validierung der Anfragen. Im SpaSe wird hingegen für die Beschreibung der Objekte und für die Anfragen je ein eigenes Schema verwendet.

Im nachfolgenden Kapitel wird eine weitere Arbeit vorgestellt, die sich u.a. mit der Simulation von Mehrfachvererbungen auf objektrelationalen Datenbanken beschäftigt.

2.1 UMLRep

Im Rahmen des Forschungsprojektes SENSOR, Teil des Sonderforschungsbereichs 501, wurde ein UML-Repository entwickelt, in dem UML-Modelle gespeichert und verwaltet werden können. Die hier vorgestellte Arbeit entstand aus diesem Projekt.

Die Dissertation von Hans-Peter Steiert mit dem Titel „Aspekte der generativen Entwicklung von ORDBMS-basierten Datenverwaltungsdiensten“ [Ste02] behandelt u.a. den Einsatz von UML zur Spezifikation von Datenbankschemata. Es wird ein UML-Repository beschrieben, welches für die Speicherung von UML-Daten verwendet werden soll. Im Zuge dessen muss das UML-Metamodell auf ein objektrelationales Datenmodell abgebildet werden. Da UML intensiv Mehrfachvererbung verwendet, musste ein Weg gefunden werden, wie dieses auf effiziente Weise auf dem einfachen Datenmodell der Datenbank, das nur die Einfachvererbung unterstützt, simuliert werden kann. Für die Realisierung dieser Simulation wurden verschiedene Ansätze miteinander verglichen. Nachfolgend wird derjenige Ansatz vorgestellt, der letztlich in UMLRep umgesetzt wurde.

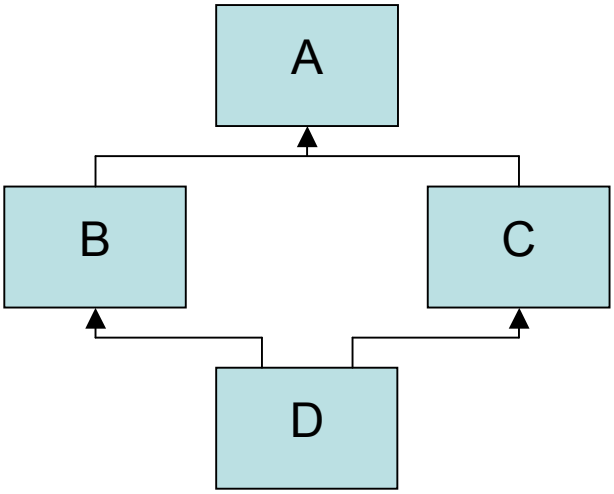

Abb. 2-1) Klassenhierarchie mit Mehrfachvererbung

Am Beispiel der einfachen Klassenhierarchie in Abb. 2-1 soll das verwendete Datenbankschema erläutert werden. Die hier verwendete Datenbank ist Informix™.
Zu den Klassen A, B, C und D existieren die Typen A_ty, B_ty, C_ty und D_ty mit zunächst einfacher Vererbung. In diesem Beispiel erbt B_ty von A_ty, D_ty von B_ty und C_ty von A_ty.

Zu jedem Typ wird eine Tabelle erzeugt, so z.B. für C_ty die Tabelle C_ta. Damit beim Zugriff auf C_ta auch alle Tupel der Tabelle D_ta gefunden werden, werden diese zusätzlich in C_ta dupliziert (die hier verwendete Datenbank unterstützte das Konzept der Referenzbeziehungen zwischen Klassen nicht). Durch ein besonderes Flag im Objektidentifikator werden diese Tupel als Duplikate markiert. Durch Sich-

ten auf den getypten Tabellen können diese Duplikate ausgeblendet werden. Trigger warten dabei die Abhängigkeiten zwischen Duplikaten und Originalen. Werden neue Tupel in D_ta eingetragen, so sorgt z.B. ein Insert-Trigger dafür, dass diese auch in C_ta eingetragen werden.

Insgesamt werden für jede Tabelle 7 Sichten definiert: Drei für den Zugriff auf die Hierarchie (lesen, ändern, löschen) und 4 für den Zugriff auf die Tabelle (lesen, ändern, löschen und einfügen). Diese Menge an Sichten ist erforderlich, da eine alleine nicht alle Anforderungen erfüllen kann. Die Gründe liegen zum einen an dem Wunsch, einige geerbte Attribute ggf. umbenennen zu können, zum anderen an der Weise, wie die Mehrfachvererbung realisiert wurde. Für Aufrufe von polymorphen Funktionen an der SQL-Schnittstelle muss der ROW TYPE eines Tupels zurückgegeben werden (mit „select table from table“). Daher wurde für jede Anweisungsart (insert, update, delete, query) eine spezielle Sicht definiert, die neben den Attributen auch den ROW TYPE zurückliefert. Eine SELECT-Sicht für Tabelle C sieht z.B. folgendermaßen aus [MS02]:

```
create view c_vi_select as
 select c_ta.*,
    (select a_ta from a_ta where a_ta.ID = c_ta.ID) AS as_row
 from c_ta
```

Dieses Konzept wurde Type-Table-View^n (TTV^n) genannt, da es pro Typ eine Tabelle und n Sichten gibt.

3 Die Nexus-Plattform

Der hier neu zu implementierende Spatial Model Server bildet einen Teil der Nexus-Plattform. Daher ist es zweckdienlich, zunächst die grundsätzliche Architektur und die einzelnen Komponenten von Nexus [NGS+01] kurz vorzustellen.

3.1 Die Dienste der Nexus-Plattform

Nexus stellt ähnlich dem World Wide Web ein verteiltes System dar und ist in drei Ebenen aufgebaut: Nexus-Anwendungen, Knoten und Dienste (siehe Abb. 3-1).

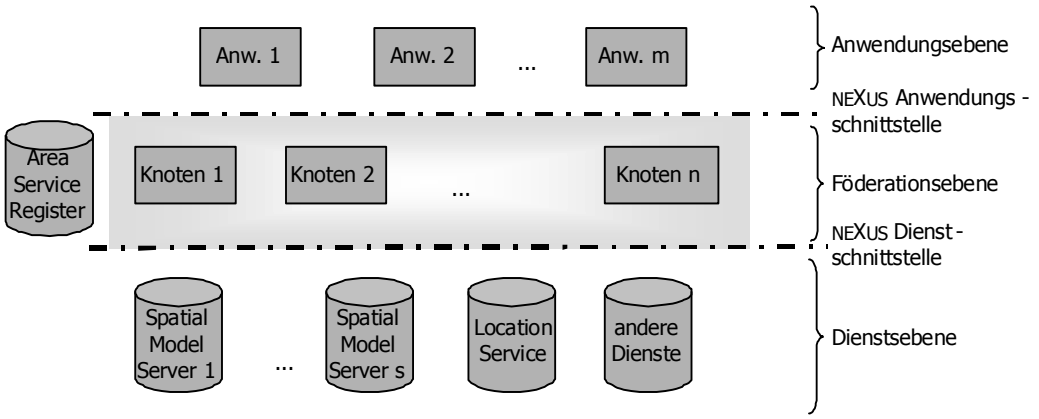

Abb. 3-1) Nexus Architektur (basierend auf [NGS+01])

Föderationsknoten vermitteln zwischen den Nexus-Diensten und den Anwendungen und bieten eine integrierte und konsistente Sicht auf das Augmented World Model (AWM), ein objektorientiertes Informationsmodell der realen Welt (siehe Abb. 3-2). Dieses Abbild beinhaltet u.a. Modelle von Gebäuden und Straßen, die mit Koordinaten versehen sind und real existierende Objekte repräsentieren.

Das Schema für die Beschreibung dieser Objekte heißt Augmented World Schema (AWS). Es enthält den Standardobjekttyp `NexusObject`, von dem alle weiteren Objekttypen abgeleitet werden. In dieses Umgebungsmodell lassen sich ebenso virtuelle Objekte einfügen, wie zum Beispiel virtuelle Litfaßsäulen oder Informationstafeln. Diese können auf externe Webseiten, passend zum Ort oder Gebäude, verweisen. Vielfältige Informationen bereichern somit das Weltmodell und werden zugleich an reale Objekte oder Orte gebunden.

Um ein Beispiel zu nennen: Geschichtliche Daten und Hintergründe zu Sehenswürdigkeiten einer Stadt sind für den Touristen am nützlichsten, wenn diese vor Ort, also mit direktem Bezug zum Objekt, zugänglich wären, z.B. über einen tragbaren Minicomputer mit drahtloser Anbindung an die Nexus-Plattform. In diesem Fall

würde es dann schon genügen, mit einem PDA und der entsprechenden Anwendung an das Denkmal oder das Gebäude heranzutreten, um alle Informationen zu erhalten.

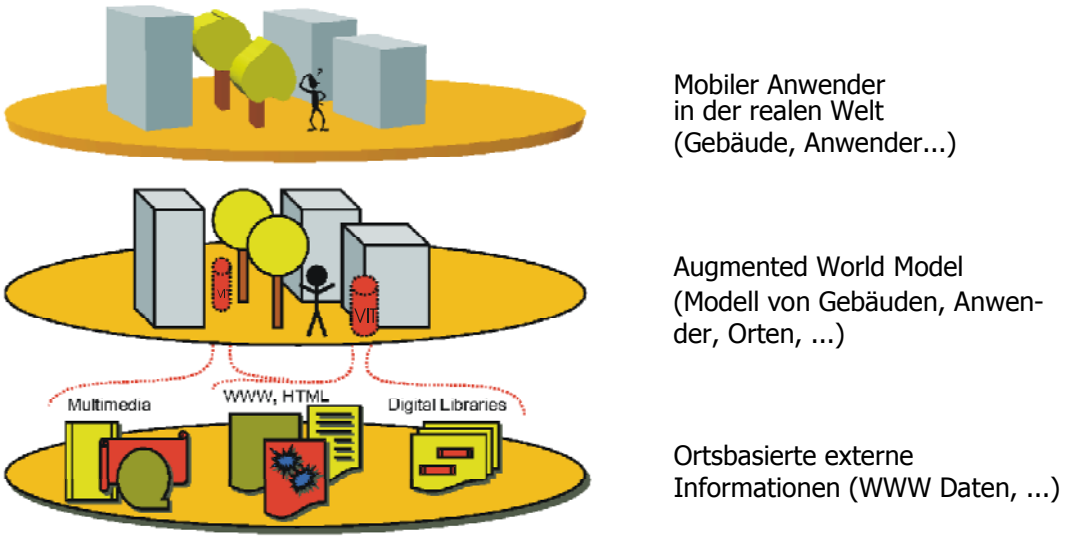

Abb. 3-2) Augmented World Model (AWM) (basierend auf [NGS+01])

Für das Speichern und Wiederauffinden der Daten werden verschiedene Server eingesetzt. Spatial Model Server werden zur Speicherung von statischen, d.h. mit fester Position, Location Server zur Verwaltung von mobilen Objekten verwendet. Statische Objekte sind zum Beispiel Straßen, Gebäude oder Räume, eventuell mit Links zu WWW-Seiten, mobile Objekte hingegen Menschen und Fahrzeuge. Jeder Spatial Model Server verwaltet ein bestimmtes geografisches Gebiet (Augmented Area) und registriert sich beim Area Service Register (ASR).

Die Spatial Model Server werden über die Föderation angesprochen. Der direkte Zugriff ist ebenfalls möglich. Zum Datenaustausch und für die Formulierung von Anfragen zwischen Anwendungen und Nexus-Diensten werden die dafür entwickelten XML-Sprachen Augmented World Modeling Language (AWML) und Augmented World Query Language (AWQL) verwendet.

Des weiteren existiert in Nexus ein so genannter Ereignisdienst (Event Service) [Til02, Fri02]. Mit dessen Hilfe können räumliche Ereignisse, wie zum Beispiel das Betreten oder Verlassen eines vordefinierten räumlichen Gebiets durch ein Mobiles Objekt, beobachtet und an interessierte Anwendungen kommuniziert werden. Als Ursprung von Ereignissen dienen z.B. die Location Server, die ein bestimmtes Geschehen beobachten und an den Event Service weiterleiten. Dieser sendet daraufhin eine Benachrichtigung an alle angemeldeten Klienten aus.

Als Ausführungsumgebung für mobile Anwendungen können so genannte Persönliche Digitale Assistenten (kurz PDAs) verwendet werden. Diese sind kleine tragbare Minicomputer mit Display und Kommunikationsschnittstellen, wie z.B. Infrarot, W-

LAN und Bluetooth, auf denen eine Nexus-Applikation installiert werden kann. Diese Kleinstcomputer verfügen über genug Leistung, um komplexe Clientanwendungen in akzeptabler Geschwindigkeit ausführen zu können.

4 Motivation

In diesem Kapitel werden die Aufgaben eines Spatial Model Servers vorgestellt sowie die Hauptgründe für eine Neuimplementierung aufgeführt.

Eine der Hauptkomponenten des Nexus-Systems bilden die Spatial Model Server (SpaSe). Die zum Zeitpunkt der Aufgabenstellung eingesetzte Implementierung verwendete ein objektrelationales Datenbankverwaltungssystem als Backend zur persistenten Speicherung der Nexusobjekte. Für die Formulierung von Anfragen an die Server wurde eine eigene Anfragesprache namens Augmented World Query Language (AWQL) entwickelt, welche auf XML basiert [SNG+01]. Anfragen müssen daher vom SpaSe in SQL übersetzt werden. Das verwendete Datenmodell sowie das Konzept der Anfragesprache weichen stark vom relationalen Modell ab, so dass pro AWQL-Anfrage in der Regel mehrere SQL-Anfragen auf verschiedene Tabellen entstehen. Deren Ergebnisse müssen in geeigneter Weise zu einem Gesamtresultat kombiniert werden.

4.1 AWQL und SQL

Eine häufig benötigte Anfrage in AWQL ist z.B. diese:

```
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
 <restriction>
  <and>
   <inside>
    <attr name="extent"/>
    <nexusdata>
     <WKT>MULTIPOLYGON ((( 9.179444 48.785555, 9.18555 48.783611, 9.1783333
48.771944, 9.17138888 48.7761111, 9.179444 48.785555)))</WKT>
    </nexusdata>
   </inside>
   <equal>
    <attr name="type"/>
    <nexusdata><Table>Building</Table></nexusdata>
   </equal>
  </and>
 </restriction>
</awql>
```

Obige Anfrage stellt eine gewöhnliche Suchanfrage nach Objekten vom Typ „Building“ dar, die sich innerhalb eines bestimmten räumlichen Gebiets befinden. Die einzelnen Suchbedingungen werden innerhalb eines „restriction“-Elements angegeben. Ein Suchgebiet kann, wie oben dargestellt, mittels eines „inside“-Elements beschrieben werden. Einzelne Attributvergleiche können z.B. mit „equal“-Elementen durchgeführt werden. Besonderes Augenmerk ist auf den Typvergleich zu legen. Anders als bei SQL steht „Building“ hier nicht für den Namen der Tabelle, in der ge-

sucht werden soll, sondern für den Anfangspunkt in der Vererbungshierarchie der Objekttypen. Konkret werden in dieser Anfrage Objekte sowohl vom Typ „Building“ als auch von sämtlichen Subtypen von „Building“ gesucht. Daraus resultieren dann bei dem bisher verwendeten Datenbankschema eine oder mehrere SQL-Anfragen an die Datenbank. Die Abarbeitung dieser Teilanfragen hat sich als sehr zeitaufwendig herausgestellt. Neben den bereits erwähnten Suchbedingungen gibt es noch die Standardoperatoren, wie z.B. „or“, „and“ und „not“, mit deren Hilfe sich beliebig komplexe Suchanfragen über mehrere Typen und Attribute formulieren lassen. Dies hat zur Folge, dass sich die Anzahl an Teilanfragen pro AWQL-Anfrage weiter erhöht. Es besteht daher der Wunsch, die bisherige Anfrageverarbeitung effizienter zu gestalten und ein neues Datenbankschema zu finden, das die Bearbeitung einer Anfrage insgesamt beschleunigt.

4.2 Mehrfachvererbung

Das bisher eingesetzte Datenbanksystem ist objektrelational und unterstützt ein einfaches Typensystem. Es können Objekttypen definiert und Tabellen daraus erzeugt werden (sog. getypte Tabellen). So existiert für jeden Objekttyp in Nexus eine eigene Tabelle in der Datenbank. Allerdings ist man hier auf einfache Vererbung beschränkt, da die Datenbank (DB2 V8.1) den SQL:1999-Standard nur teilweise unterstützt und auf die Implementierung der Mehrfachvererbung verzichtet wurde.

Der Wunsch ist daher, ein Datenbankschema zu finden, das es erlaubt, die Konzepte der Mehrfachvererbung auf Anwendungsebene zu nutzen. In Kapitel 5 werden dazu einige Datenbankschemata beschrieben, die dies bewerkstelligen sollen. Hauptaugenmerk wurde dabei vor allem auf die Realisierung der Mehrfachvererbung gelegt.

5 Gefundene Lösungsansätze

In den nachfolgenden Unterkapiteln folgt die Vorstellung der gefundenen Lösungsansätze. Diese umfassen zunächst nur die verschiedenen Datenbankschemata. Im Anschluss wird in Kapitel 6 eine theoretische Bewertung durchgeführt. Hauptaugenmerk wurde dabei auf die effiziente Realisierung der Mehrfachvererbung gelegt. Auf Basis dieser theoretischen Bewertung wurden dann einzelne Ansätze ausgewählt und implementiert (siehe Kapitel 7).

Beispielhierarchie

Für die plastische Veranschaulichung der Mehrfachvererbung wird in den nachfolgenden DB-Schemaansätzen folgende klassische Mehrfachvererbung umgesetzt (siehe Abb. 5-1). Der Objekttyp `InfoTable` erbt von `BuildingElement` und `Plate`, die wiederum selbst vom gemeinsamen Supertyp `SpatialObject` abgeleitet wurden. Es seien insgesamt drei 3 Objekte definiert (siehe Abb. 5-2).

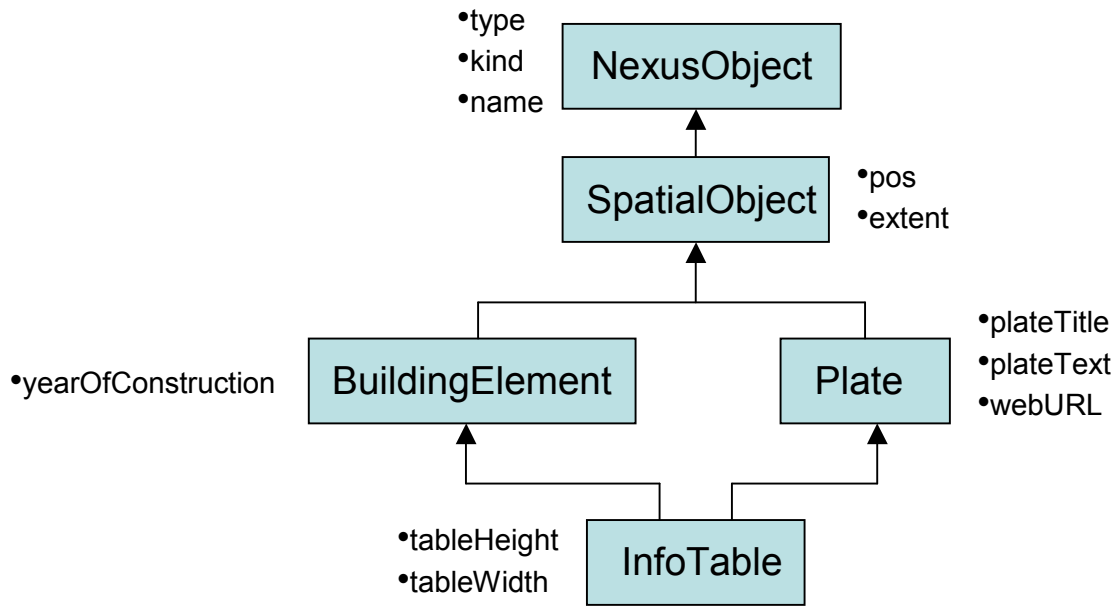

Abb. 5-1) Vererbungshierarchie

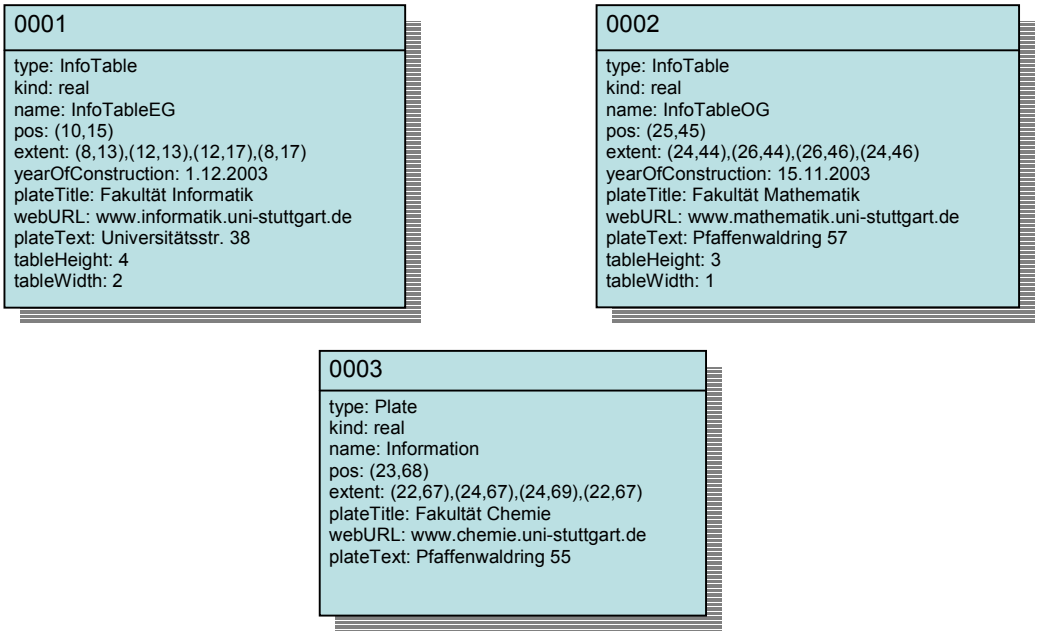

Abb. 5-2) Beispielobjekte in Nexus

5.1 Vertikale Partitionierung

Vertikale Partitionierung der Klassentabellen unter Verwendung einer rein relationalen Datenbank.

Dieser Ansatz wird in [CCN+99] und [RBP+93] beschrieben und stellt eine Möglichkeit dar, wie Objekte eines Objektmodells in mehreren Tabellen verwaltet werden können. Diese Art der Aufteilung wurde in die Betrachtung mit aufgenommen, da es sich um ein sehr intuitives und allgemein bekanntes Verfahren handelt.

Das Datenbankschema baut sich wie folgt auf:
Bei diesem Ansatz wird für jeden neuen Objekttyp eine eigene Tabelle angelegt. Diese enthält alle neu hinzukommenden Attribute. Über die schemaweit eindeutige Objekt-ID (OID) stehen die Tabelleneinträge in Relation. Um alle zu einem bestimmten Objekt gehörenden Attribute zu ermitteln, müssen die Objekttyptabellen gemäß ihrer Vererbungshierarchie miteinander verbunden werden, d.h. entlang des Vererbungspfades. Dies geschieht mittels Equal-Joins auf den OIDs. Der Vorteil hierbei ist, dass man keine Redundanz hat. Lediglich die mehrfache Speicherung der OID zum Wiederauffinden der einzelnen Segmente eines Objekts ist von Nöten. Verbunde können ebenfalls aufgrund der Primärindexeigenschaften der OIDs effizient durchgeführt werden. Nachteilig ist aber bei tiefen Vererbungshierarchien einzig die hohe Anzahl an Verbunden, die notwendig sind, um alle Attribute zu lesen.

Nachfolgend wird zur Veranschaulichung die Vererbungshierarchie aus Abbildung 5-1 auf diese Weise abgebildet. Zu jeder Tabelle steht der korrespondierende Objekttyp in Klammern. Zusätzlich erhält jeder Objekttyp eine eigene Tabellensicht, die einen zusammenfassenden Blick auf sämtliche Objektattribute ermöglicht. Diese

Sichten werden mit Hilfe der Vererbungshierarchie generiert und können für die weitere Anfrageverarbeitung genutzt werden.

5.1.1 Umsetzung der Beispielhierarchie

Es werden im nachfolgenden Beispiel insgesamt drei Nexusobjekte gespeichert (siehe Abb. 5-2), zwei vom Typ `InfoTable` und eines vom Typ `Plate`.

`NexusObject`: (vom Typ `NexusObject`)

oid	**type**	**kind**	**name**
0001	InfoTable	real	InfoTableEG
0002	InfoTable	real	InfoTabeOG
0003	Plate	real	Information

`SpatialObject`: (vom Typ `SpatialObject` unter `NexusObject`)

oid	**pos**	**extent**
0001	(10, 15)	(8,13),(12,13),(12,17),(8,17)
0002	(25, 45)	(24,44),(26,44),(26,46),(24,46)
0003	(23, 68)	(22,67),(24,67),(24,69),(22,67)

`BuildingElement`: (vom Typ `BuildingElement` unter `SpatialObject`)

oid	**yearOfConstruction**
0001	01.12.2003
0002	15.11.2003

`Plate`: (vom Typ `Plate` unter `SpatialObject`)

oid	**plateTitle**	**webURL**	**plateText**
0001	Fakultät Informatik	www.informatik...	Universitätsstr. 38
0002	Fakultät Mathematik	www.mathematik...	Pfaffenwaldring 57
0003	Fakultät Chemie	www.chemie...	Pfaffenwaldring 55

`InfoTable`: (vom Typ `InfoTable` unter `BuildingElement` und `Plate`)

oid	**tableHeight**	**tableWidth**
0001	4	2
0002	3	1

Bemerkung: Rein relationale Datenbanken kennen keine Objekttypen. Der Klassentyp einer Tabelle ist hier nur zur Verdeutlichung der Hierarchie in Klammern mit angegeben.

Tabellensichten am Beispiel von `InfoTable`: Es werden die Tabellen `NexusObject`, `SpatialObject`, `BuildingElement`, `Plate` und `InfoTable` über das Join-Attribut OID miteinander verbunden.

`InfoTableView` (Ausschnitt):

oid	**type**	**...**	**webURL**	**plateText**	**tableHeight**	**tableWidth**
0001	InfoTable	...	www.inf...	Universitätsstr. 38	4	2
0002	InfoTable	...	www.math...	Pfaffenwaldring 57	3	1

Die Objekte mit den OIDs 0001 und 0002 sind jeweils vom Typ `InfoTable`. Bei der vertikalen Partitionierung werden Objekte in Fragmente aufgeteilt. Entlang des Vererbungspfades müssen diese Fragmente beim Auslesen wieder zu einem vollständigen Objekt zusammengefügt werden. Dies geschieht z.B. in der oben beschriebenen Sicht, die alle beteiligten Tabellen über die OIDs verbindet.

5.1.2 Metadaten

Für die Verwaltung der Vererbungsbeziehungen der Objekttypen werden verschiedene Metadatentabellen angelegt. So existiert zum einen eine Tabelle `TypeHierarchy`, in der die Namen der direkten Subtypen zu jedem Objekttyp abgelegt werden und eine Tabelle `TableName`, die für die korrekte Abbildung von Objekttyp auf Objekttabelle sorgt. Diese Aufsplittung von Typname und Tabellenname ermöglicht die freie Benennung der Objekttypen, ohne auf die Beschränkungen für Tabellennamen durch die Datenbank limitiert zu sein.

`TypeHierarchy`:

Type	**SubType**
NexusObject	NexusObject
NexusObject	SpatialObject

SpatialObject	SpatialObject
SpatialObject	BuildingElement
SpatialObject	Plate
BuildingElement	BuildingElement
BuildingElement	InfoTable
Plate	Plate
Plate	InfoTable
InfoTable	InfoTable

`TableName`:

ObjectType	**TableName**
NexusObject	NexusObject
SpatialObject	SpatialObject
BuildingElement	BuildingElement
Plate	Plate
InfoTable	InfoTable

5.2 Horizontale Partitionierung

Horizontale Partitionierung der Klassentabellen unter Verwendung rein relationaler Datenbanken.

Dieser Ansatz stammt ebenfalls aus [CCN+99]. Er wurde in die Betrachtung mit eingeschlossen, da man hier auf Joins verzichten kann. Außerdem lässt sich diese Abbildung leicht erweitern und an die speziellen Bedürfnisse von Nexus anpassen (vgl. 5.3).

Das Datenbankschema baut sich wie folgt auf:
Bei diesem Ansatz wird für jeden Objekttyp eine eigene Tabelle angelegt, in der gegenüber dem vertikalen Partitionierungsansatz alle Attribute eines Objekttyps, d.h. sowohl die neuen als auch die ererbten, gespeichert werden. Über die schemaweit eindeutige Objekt-ID (OID) stehen die einzelnen Tabelleneinträge (Fragmente eines Objekts) in Relation. Zum Auslesen aller Attribute eines Objekts von einem bestimmten Typ muss nur eine einzige Objekttyptabelle herangezogen werden. Es werden keine Joins benötigt. Der Nachteil dieses Ansatzes ist die hohe Redundanz, da Attribute aus Supertabellen auch in allen Subtabellen repliziert gespeichert werden. Bei Änderungsoperationen müssen somit alle betroffenen Tabellen in der Vererbungshierarchie berücksichtigt und ggf. nachgeführt werden. Eine mögliche Modifikation,

um die Redundanz zu mindern, stellt der Ansatz in Kapitel 5.3 dar, bei dem die Objekte nur jeweils in ihren unmittelbaren Objekttabellen gespeichert werden.

Nachfolgend wird die Vererbungshierarchie aus Abbildung 5-1 mit Hilfe dieses Ansatzes umgesetzt.

5.2.1 Umsetzung der Beispielhierarchie

Es werden im nachfolgenden Beispiel insgesamt drei Nexusobjekte gespeichert, zwei vom Typ `InfoTable` und eines vom Typ `Plate`.

`NexusObject`: (vom Typ `NexusObject`)

oid	**type**	**kind**	**name**
0001	InfoTable	real	InfoTableEG
0002	InfoTable	real	InfoTableOG
0003	Plate	real	Information

`SpatialObject`: (vom Typ `SpatialObject` unter `NexusObject`)

oid	**type**	**kind**	**name**	**pos**	**extent**
0001	InfoTable	real	InfoTableEG	(10,15)	(8,13),(12,13),(12,17),(8,17)
0002	InfoTable	real	InfoTableOG	(25,45)	(24,44),(26,44),(26,46),(24,46)
0003	Plate	real	Information	(23,68)	(22,67),(24,67),(24,69),(22,67)

`BuildingElement`: (vom Typ `BuildingElement` unter `SpatialObject`)

oid	**type**	**kind**	**name**	**pos**	**extent**	**…**
0001	InfoTable	real	InfoTableEG	(10,15)	(8,13),(12,13),(12,17),(8,17)	
0002	InfoTable	real	InfoTableOG	(25,45)	(24,44),(26,44),(26,46),(24,46)	

…	**yearOfConstruction**
	01.12.2003
	15.11.2003

`Plate`: (vom Typ `Plate` unter `SpatialObject`)

oid	**type**	**kind**	**name**	**pos**	**extent**	**…**
0001	InfoTable	real	InfoTableEG	(10,15)	(8,13),(12,13),(12,17),(8,17)	
0002	InfoTable	real	InfoTableOG	(25,45)	(24,44),(26,44),(26,46),(24,46)	
0003	Plate	real	Information	(23,68)	(22,67),(24,67),(24,69),(22,67)	

…	**plateTitle**	**webURL**	**plateText**
	Fakultät Inf.	www.informatik...	Universitätsstr. 38
	Fakultät Math.	www.mathematik...	Pfaffenwaldring 57
	Fakultät Chemie	www.chemie...	Pfaffenwaldring 55

`InfoTable`: (vom Typ `InfoTable` unter `BuildingElement` und `Plate`)

oid	**type**	**kind**	**name**	**pos**	**extent**	**…**
0001	InfoTable	real	InfoTableEG	(10,15)	(8,13),(12,13),(12,17),(8,17)	
0002	InfoTable	real	InfoTableOG	(25,45)	(24,44),(26,44),(26,46),(24,46)	

…	**yearOfConstruction**	**plateTitle**	**webURL**	**plateText**	**…**
	01.12.2003	Fakultät Inf.	www.informatik...	Univer...	
	15.11.2003	Fakultät Math.	www.mathematik...	Pfaffenwald...	

…	**tableHeight**	**tableWidth**
	4	2
	3	1

5.2.2 Metadaten

Für die Verwaltung der Vererbungsbeziehungen der Objekttypen werden die bereits aus dem vertikalen Ansatz her bekannten Metadatentabellen verwendet (siehe Kapitel 5.1.2).

5.3 Modifizierte horizontale Partitionierung

Horizontale Partitionierung der Klassentabellen mit ausschließlicher Speicherung der Objekte in ihren unmittelbaren Objekttyptabellen.

Dieser Ansatz stellt eine eigene Variation des Ansatzes aus Kapitel 5.2 dar. Es wird der ursprüngliche horizontale Partitionierungsansatz verwendet, wobei die Objekte jedoch ausschließlich in ihren Objekttyptabellen gespeichert werden. Konkret bedeutet dies, dass es keine Fragmente in darüberliegenden Supertabellen mehr gibt. Da alle Einträge in Supertabellen redundant sind, können diese weg gelassen werden. Aufgrund des Umstandes, dass der Spatial Model Server stets alle verfügbaren Attribute eines Objekts zurückgeben soll und somit immer bis zu den Blattknoten der Vererbungshierarchie suchen muss, stellt diese Variation eine Reduzierung des auszulesenden Datenvolumens dar. Jedes Attribut eines Objekts wird höchstens ein Mal gelesen.

Nachfolgend wird die Vererbungshierarchie aus Abbildung 5-1 mit Hilfe dieses Ansatzes umgesetzt.

5.3.1 Umsetzung der Beispielhierarchie

Es werden im nachfolgenden Beispiel insgesamt drei Nexusobjekte gespeichert, zwei vom Typ `InfoTable` und eines vom Typ `Plate`.

`NexusObject`: (vom Typ `NexusObject`)

oid	**type**	**kind**	**name**

`SpatialObject`: (vom Typ `SpatialObject` unter `NexusObject`)

oid	**type**	**kind**	**name**	**pos**	**extent**

`BuildingElement`: (vom Typ `BuildingElement` unter `SpatialObject`)

oid	**type**	**kind**	**name**	**pos**	**extent**	**yearOfConstruction**

`Plate`: (vom Typ `Plate` unter `SpatialObject`)

oid	**type**	**kind**	**name**	**pos**	**extent**	**…**
0003	Plate	real	Information	(23,68)	(22,67),(24,67),(24,69),(22,67)	

…	plateTitle	webURL	plateText
	Fakultät Chemie	www.chemie....	Pfaffenwaldring 55

`InfoTable`: (vom Typ `InfoTable` unter `BuildingElement` und `Plate`)

oid	type	kind	name	pos	extent	…
0001	InfoTable	real	InfoTableEG	(10,15)	(8,13),(12,13),(12,17),(8,17)	
0002	InfoTable	real	InfoTableOG	(25,45)	(24,44),(26,44),(26,46),(24,46)	

…	yearOfConstruction	plateTitle	webURL	…
	01.12.2003	Fakultät Inf.	www.informatik...	
	15.11.2003	Fakultät Math.	www.mathematik...	

…	plateText	tableHeight	tableWidth
	Universitätsstr. 38	4	2
	Pfaffenwaldring 57	3	1

5.3.2 Metadaten

Für die Verwaltung der Vererbungsbeziehungen der Objekttypen werden die bereits aus dem vertikalen Ansatz her bekannten Metadatentabellen verwendet (siehe Kapitel 5.1.2).

5.4 Hierarchischer Ansatz

Hierarchische Partitionierung der Klassentabellen unter Verwendung von rein relationalen Datenbanken.

Dieser Ansatz wird in [CCN+99] beschrieben. Er wurde in die Betrachtung mit einbezogen, da er leicht umzusetzen ist und zudem eine hohe Performance verspricht.

Das Datenbankschema baut sich wie folgt auf:
Dieser Ansatz verwendet für die Speicherung der Objekte aus einer Objekttyphierarchie eine einzige große hierarchische Mastertabelle. In dieser werden sämtliche Attribute der verschiedenen Objekttypen abgelegt. Nicht verwendete Attribute eines Objekts werden auf NULL gesetzt. Die Zuteilung von Attributen zu Objekttypen geht hierbei jedoch verloren und muss daher separat in den Metadaten verwaltet werden. Der Vorteil liegt im schnelleren Auffinden sämtlicher zu einem Objekt gehörenden Attribute, da diese in einem einzigen Tupel in der Mastertabelle abgespeichert werden. Je nach gewünschtem Typ müssen lediglich die betreffenden Spalten selektiert werden.

Nachfolgend wird die Vererbungshierarchie aus Abbildung 5-1 mit Hilfe dieses Ansatzes umgesetzt.

5.4.1 Umsetzung der Beispielhierarchie

Es werden im nachfolgenden Beispiel insgesamt drei Nexusobjekte gespeichert (vgl. Abb. 5-2), zwei vom Typ `InfoTable` und eines vom Typ `Plate`.

`MasterTable`:

oid	**type**	**kind**	**name**	**pos**	**extent**	**…**
0001	InfoTable	real	InfoTableEG	(10,15)	(8,13),(12,13),(12,17),(8,17)	
0002	InfoTable	real	InfoTableOG	(25,45)	(24,44),(26,44),(26,46),(24,46)	
0003	Plate	real	Information	(23,68)	(22,67),(24,67),(24,69),(22,67)	

…	**yearOfConstruction**	**plateTitle**	**webURL**	**…**
	01.12.2003	Fakultät Inf.	www.informatik...	
	15.11.2003	Fakultät Math.	www.mathematik...	
	-	Fakultät Chemie	www.chemie...	

…	**plateText**	**tableHeight**	**tableWidth**
	Universitätsstr. 38	4	2
	Pfaffenwaldring 57	3	1
	Pfaffenwaldring 55	-	-

Bemerkung: Mit Hilfe von Sichten können die einzelnen Klassentabellen herausgefiltert werden.

5.4.2 Metadaten

Für die Verwaltung der Vererbungsbeziehungen der Objekttypen wird die bereits aus dem vertikalen Ansatz her bekannte Metadatentabelle `TypeHierarchy` verwendet (siehe Kapitel 5.1.2). Da es keine einzelnen Objekttyptabellen gibt, kann die Abbildung von Objekttyp auf Objekttyptabelle entfallen. Neu ist die Tabelle `ObjectTypeAttr`, die für jeden Objekttyp die Namen der neu hinzu gekommenen Attribute speichert.

`ObjectTypeAttr`:

type	**attribute**
NexusObject	type
NexusObject	name
NexusObject	kind
SpatialObject	pos
SpatialObject	extent
BuildingElement	yearOfConstruction
Plate	plateTitle
Plate	plateText
Plate	webURL
InfoTable	tableHeight
InfoTable	tableWidth

5.5 Modifizierter hierarchischer Ansatz

Dieser Ansatz ist in großen Teilen identisch mit dem vorherigen hierarchischen Ansatz (siehe Kapitel 5.4). Als Erweiterung wird hier jedoch die Spalte "type" in einem etwas anderen Kontext gebraucht. An die Stelle des Objekttyps tritt nun eine genaue Auflistung der einzelnen Objekttypen innerhalb des Vererbungspfades eines jeden Objekts. Die darin enthaltenen Objekttypen werden durch ein Sonderzeichen z.B. „#“ voneinander abgegrenzt. Da der Spatial Model Server nach Konvention nur vollständige Objekte zurückliefern soll, von etwaigen Filtern abgesehen, muss eine Suche immer bis zu den letzten Objekttypen, d.h. bis zu den Blättern des Vererbungsgraphen, verlaufen. Daraus resultiert zwangsläufig, dass bei einem gegebenen Objekttyp auch die direkten und indirekten Subtypen bestimmt und aufgefunden werden müssen. In diesem speziellen Ansatz kann jetzt die Bestimmung der Subtypen wegfallen, da jedes Objekt quasi von sich aus weiß, von welchen Objekttypen es abstammt.

5.5.1 Umsetzung der Beispielhierarchie

Es werden im nachfolgenden Beispiel insgesamt drei Nexusobjekte mit Hilfe dieses Datenbankansatzes gespeichert (vgl. Abb. 5-2), zwei vom Typ `InfoTable` und eines vom Typ `Plate`.

`MasterTable`:

oid	**type**	**...**
0001	#InfoTable#Plate#BuildingElement#SpatialObject#NexusObject#	
0002	#InfoTable#Plate#BuildingElement#SpatialObject#NexusObject#	
0003	#Plate#SpatialObject#NexusObject#	

...	**kind**	**name**	**pos**	**extent**	**...**
	real	InfoTableEG	(10,15)	(8,13),(12,13),(12,17),(8,17)	
	real	InfoTableOG	(25,45)	(24,44),(26,44),(26,46),(24,46)	
	real	Information	(23,68)	(22,67),(24,67),(24,69),(22,67)	

…	**yearOfConstruction**	**plateTitle**	**WebURL**	**…**
	01.12.2003	Fakultät Inf.	www.informatik...	
	15.11.2003	Fakultät Math.	www.mathematik...	
	-	Fakultät Chemie	www.chemie...	

…	**plateText**	**tableHeight**	**tableWidth**
	Universitätsstr. 38	4	2
	Pfaffenwaldring 57	3	1
	Pfaffenwaldring 55	-	-

5.5.2 Metadaten

Es werden dieselben Metadatentabellen verwendet, wie im vorherigen hierarchischen Ansatz (siehe Kapitel 5.4.2).

5.6 Attributtabellenansatz

Zerlegung der Objekte in ihre Attribute und Speicherung in einer gemeinsamen Tabelle.

Dieser Ansatz wird in [RBP+93] beschrieben und kommt in vielen Systemen zur Übersetzung von Wissensbasen in Datenbankdaten zum Einsatz. Diese Art der Abbildung des Objektmodells wurde in leicht veränderter Form in die Betrachtung mit aufgenommen, da dieser Ansatz die Umsetzung von boolesche Operatoren, wie z.B. AND, als Mengenoperatoren, wie z.B. INTERCEPT, erlaubt. Außerdem kann man auf diese Weise auf die aufwendige Umformung der Restriktion nach DNF verzichten (siehe 6.7).

Die Objekte werden in diesem Ansatz nicht mehr in verschiedenen Objekttyptabellen oder in einer gemeinsamen hierarchischen Tabelle verwaltet, sondern in ihre Be-

standteile, den Attributen, zerlegt und in einer speziellen Attributtabelle gespeichert. Jedes Attribut ist über eine schemaweit eindeutige Objekt-ID fest an ein Objekt gebunden und kann auf diese Weise wieder aufgefunden werden. Zum Auslesen eines kompletten Objekts müssen alle Einträge in der Attributtabelle mit dieser bestimmten Objekt-ID ausgelesen und im Spatial Model Server zu einem Objekt zusammengesetzt werden. Da dies die Regel ist, muss diese Operation möglichst effizient sein. Der Vorteil bei diesem Ansatz liegt in der leichten Erstellung eines Indexes für ausgesuchte Attribute. So kann z.B. die Suche von Objekten anhand ihrer Position über eine Indexstruktur erfolgen. Der Nachteil ist, dass einer der vielen Vorzüge von Datenbanken, die Selbstbeschreibungsfähigkeit des Inhalts, verloren geht. Die Daten können gewissermaßen nur noch von der Anwendung richtig interpretiert werden.

5.6.1 Umsetzung der Beispielhierarchie

Es werden im nachfolgenden Beispiel insgesamt drei Nexusobjekte (siehe Abb. 5-2) gespeichert, zwei vom Typ `InfoTable` und eines vom Typ `Plate`.

`Attributetable`

oid	**AttributeName**	**VarcharValue**	**GeometryValue**	**IntegerValue**	**DateValue**
0001	type	InfoTable	-	-	-
0001	kind	real	-	-	-
0001	name	InfoTableEG	-	-	-
0001	pos	-	(10,15)	-	-
0001	extent	-	(8,13),(12,13),…	-	-
0001	yearOfConstruction	-	-	-	01.12.2003
0001	plateTitle	Fakultät Informatik	-	-	-
0001	webURL	www.informatik...	-	-	-
0001	plateText	Universitätsstr. 38	-	-	-
0001	tableHeight	-	-	4	-
0001	tableWidth	-	-	2	-
0002	type	InfoTable	-	-	-
0002	kind	real	-	-	-
0002	name	InfoTableOG	-	-	-
0002	pos	-	(25,45)	-	-
0002	extent	-	(24,44),(26,44),…	-	-
0002	yearOfConstruction	-	-	-	15.11.2003
0002	plateTitle	Fakultät Mathematik	-	-	-
0002	webURL	www.mathematik...	-	-	-
0002	plateText	Pfaffenwaldring 57	-	-	-

0002	tableHeight	-	-	3	-
0002	tableWidth	-	-	1	-
0003	type	Plate	-	-	-
0003	kind	real	-	-	-
0003	name	Information	-	-	-
0003	pos	-	(23,68)	-	-
0003	extent	-	(22,67),(24,67),…	-	-
0003	plateTitle	Fakultät Chemie	-	-	-
0003	webURL	www.chemie...	-	-	-
0003	plateText	Pfaffenwaldring 55	-	-	-

5.6.2 Metadaten

Für die Verwaltung der Vererbungsbeziehungen der Objekttypen werden die aus dem hierarchischen Ansatz bekannten Metadatentabellen übernommen (siehe Kapitel 5.4.2). Für die Abbildung von Attributname auf Datentyp wird eine neue Metadatentabelle hinzugefügt, `AttributeType`. Da die Attributtabelle für jeden Datentyp eine separate Spalte verwendet, ist diese zusätzliche Speicherung der Zuordnung notwendig.

`AttributeType`

attribute	**type**
type	varchar
kind	varchar
name	varchar
pos	st_geometry
extent	st_geometry
yearOfConstruction	date
plateTitle	varchar
webURL	varchar
plateText	varchar
tableHeight	integer
tableWidth	integer

5.7 Erweiterter objektrelationaler Ansatz

Ausnutzung der Einfachvererbung von ORDBMS für die Simulation der Mehrfachvererbung mittels Referenzen (Mehrfachvererbung durch Vererbung und Delegation [RBP+93]).

Dieser Ansatz stellt eine Erweiterung des bisher verwendeten Datenbankschemas dar. Er wurde in die Betrachtung mit aufgenommen, da er die objektrelationalen Eigenschaften der Datenbank ausnutzt.

Bei diesem Ansatz werden getypte Tabellen nach dem Standart SQL:1999 eingesetzt. Für Einfachvererbungen ist dies eine gängige Praxis, so dass für die am häufigsten verwendete Vererbungsart keinerlei Änderungen an der bisherigen Vorgehensweise notwendig sind. Für den speziellen Fall der Mehrfachvererbung wird zunächst, wie bei der Einfachvererbung, ein neuer Objekttyp vom Supertyp abgeleitet. Da es hier mehrere Supertypen gibt, wird z.B. der mit den meisten Attributen ausgesucht. Zusätzlich zu den gewünschten eigenen Attributen dieses neuen Objekttyps wird für jeden weiteren Supertyp jeweils ein Referenzattribut hinzugefügt, das auf ein Objekt des jeweiligen Supertyps verweist.

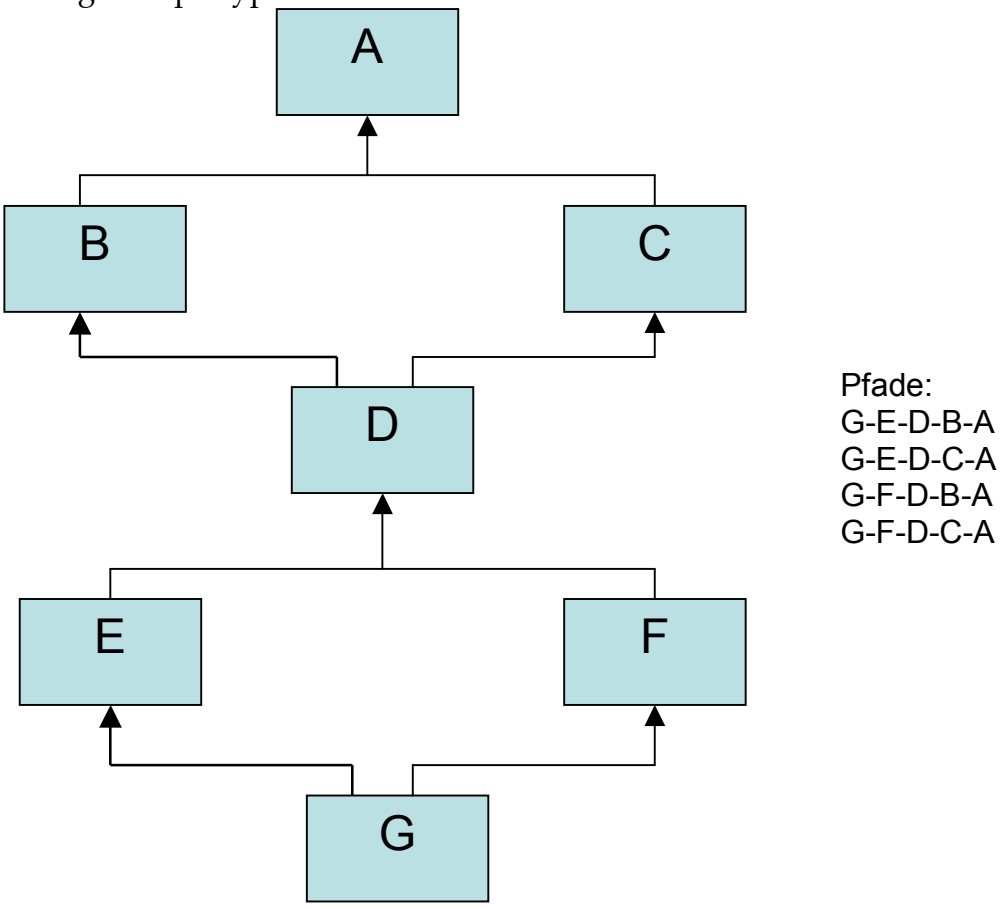

Abb. 5-3) komplexe Hierarchie

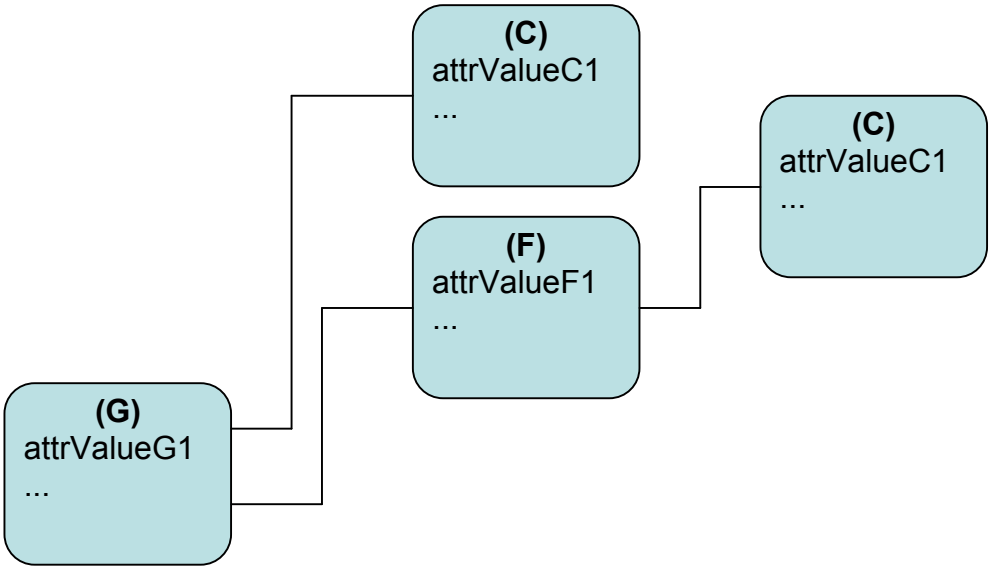

Abb. 5-4) Instanzdiagramm

In Abb. 5-3 ist eine komplexe Vererbungshierarchie mit zwei Vererbungszweigen (D und G) gegeben. Der linke Pfad ist jeweils der primäre Vererbungspfad. Insgesamt werden für ein Objekt vom Typ G vier Objekte erzeugt und miteinander verbunden (siehe Abb. 5-4). Die zusätzlich erzeugten Supertypobjekte sind jeweils über eine Referenz erreichbar (Kanten in Abb. 5-4). Die Gesamtanzahl an Objekten lässt sich allgemein aus der Anzahl an möglichen Pfaden vom Objekttyp bis zur Wurzel bestimmen.

Die entstandenen Supertypobjekte stellen für sich gesehen eigenständige Objekte dar, daher müssen sie von regulären Objekten unterschieden werden können. Dies geschieht mit Hilfe von weiteren Attributen. Das Attribut „duplicate“ gibt an, dass ein bestimmtes Objekt Teil eines mehrfachvererbten Objekts ist. Diese Bezeichnung wurde deshalb so gewählt, da in höheren Ebenen, d.h. bei den Objekttypen, bei denen die Vererbungspfade zusammenfallen, duplizierte Objekte auftreten, die sich lediglich in der Object-ID unterscheiden. In dem Beispiel aus Abb. 5-1 geschieht dies ab der getypten Tabelle `SpatialObject` in Richtung Wurzeltyp.

Das Attribut „exNoOnly“ gibt an, dass ein bestimmtes Objekt nicht in der Only-Menge dieser getypten Tabelle enthalten sein soll. Dieser explizite Ausschluss ist deshalb notwendig, da die Datenbank nicht zwischen regulären, d.h. einfachvererbten Objekten und Duplikatobjekten unterscheiden kann. Da diese Teilobjekte keine eigenständigen Objekte darstellen, dürfen sie nicht in der Only-Menge enthalten sein. Da der ONLY-Operator jedoch nicht modifiziert werden kann, muss dies in einer Tabellensicht realisiert werden. Mit Hilfe von Zeigerkonstrukten lässt sich zudem relativ einfach auf die geerbten Attribute weiterer Supertypen innerhalb einer solchen Sicht zugreifen, so dass sich insgesamt eine vergleichbare Ausgangsposition ergibt wie im horizontalen Ansatz.

Nachfolgend wird die Vererbungshierarchie aus Abbildung 5-1 mit Hilfe dieses Ansatzes umgesetzt.

5.7.1 Umsetzung der Beispielhierarchie

Es werden im nachfolgenden Beispiel insgesamt drei Nexusobjekte (siehe Abb. 5-2) gespeichert, zwei vom Typ `InfoTable` und eines vom Typ `Plate`.

`NexusObject` vom Typ `NexusObject`

oid	**type**	**kind**	**name**	**duplicate**	**exNoOnly**
0001	InfoTable	real	InfoTableEG	0	0
0002	InfoTable	real	InfoTableOG	0	0
0003	Plate	real	Information	0	0
0004	InfoTable	real	InfoTableEG	1	1
0005	InfoTable	real	InfoTableOG	1	1

`SpatialObject` vom Typ `SpatialObject` unter `NexusObject`

oid	**pos**	**extent**	**duplicate**	**exNoOnly**
0001	(10,15)	(8,13),(12,13),(12,17),(8,17)	0	0
0002	(25,45)	(24,44),(26,44),(26,46),(24,46)	0	0
0003	(23,68)	(22,67),(24,67),(24,69),(22,67)	0	0
0004	(10,15)	(8,13),(12,13),(12,17),(8,17)	1	1
0005	(25,45)	(24,44),(26,44),(26,46),(24,46)	1	1

`BuildingElement` vom Typ `BuildingElement` unter `SpatialObject`

oid	**yearOfConstruction**	**duplicate**	**exNoOnly**
0001	01.12.2003	0	1
0002	15.11.2003	0	1

`Plate` vom Typ `Plate` unter `SpatialObject`

oid	**plateTitle**	**webURL**	**plateText**	**duplicate**	**exNoOnly**
0004	Fakultät Informatik	www.informatik...	Universitätsstr. 38	0	1
0005	Fakultät Mathematik	www.mathematik...	Pfaffenwaldring 57	0	1
0003	Fakultät Chemie	www.chemie...	Pfaffenwaldring 55	0	0

`InfoTable` vom Typ `InfoTable` unter `BuildingElement` und indirekt `Plate`

oid	**tableHeight**	**tableWidth**	**superPlate**	**duplicate**	**exNoOnly**
0001	4	2	0004	0	0
0002	3	1	0005	0	0

5.7.2 Metadaten

Zu den bereits von der Datenbank erstellten Metadatentabellen kommt die bekannte `TypeHierarchy`-Tabelle aus den vorherigen Ansätzen hinzu, welche u.a. die Mehrfachvererbung katalogisiert (siehe 5.1.2).

6 Theoretische Bewertung

6.1 Vorbemerkungen

Es wird vorausgesetzt, dass der Spatial Model Server exklusiv auf die Datenbank zugreift, d.h. im Single-User-Betrieb. Auf diese Weise können zeitintensive und immer wiederkehrende Berechnungen auf den Daten bereits bei Programmstart vorgenommen werden und liegen so griffbereit im Hauptspeicher. Dazu zählt zum Beispiel die Berechnung aller Subtypen eines Objekttyps. Durch den Single-User-Betrieb der Datenbank ist sichergestellt, dass die ermittelten Metadaten konsistent sind. Eine Änderung dieser Daten außerhalb des SpaSes erfordert einen Neustart des Servers.

In den weiteren Unterkapiteln zu 6.1 folgt eine Auflistung der häufig benötigten Algorithmen bzw. Operationen mit ihren jeweiligen Aufwenden gemessen an der Anzahl der benötigten Seitenzugriffe (falls anwendbar).

Ab Kapitel 6.2 folgen die theoretischen Betrachtungen der in Kapitel 5 genannten Lösungsansätze. Zu jedem Datenbankschema werden drei häufig benötigte Anfragen untersucht und ihr geschätzter Aufwand bestimmt. Auf diese Weise soll ein Vergleich der Ansätze untereinander ermöglicht werden.

6.1.1 Index

Mit B*-Baum ergibt sich folgender Aufwand für das Auffinden und Auslesen eines Tabellentupels unter Verwendung eines Indexes gemessen an den benötigten Seitenzugriffen:
Die Höhe h des B*-Baumes mit N_T Tupeln ist begrenzt durch

$$1 + \log_{2k+1}(\frac{N_T}{2k^*}) \le h \le 2 + \log_{k+1}(\frac{N_T}{2k^*}) \quad \text{für } h \ge 2$$

typische Werte sind: h = 3 bis 4 (entspricht gleich der Anzahl an Seitenzugriffen) bei $N_T = 10^5 ... 10^7$ Tupeln. Hinzu kommt noch ein weiterer Seitenzugriff zum Lesen des eigentlichen Tupels [HR01].

6.1.2 Sort-Merge-Verbund (SMV)

Diese Verbundoperation setzt sich aus zwei Phasen zusammen. In der ersten Phase werden die beteiligten Relationen zunächst nach den Verbundattributen sortiert, falls nicht bereits geschehen, und frühzeitig alle nicht benötigten Tupel durch Überprüfung der Prädikate entfernt. In der zweiten Phase werden die Relationen jeweils paarweise und schritthaltend mit Durchführung des Verbundes gescannt.

Die Hauptkosten dieser Operation verursacht Phase 1 (O(N log N)). Wenn eine geeignete Sortierung bereits vorliegt, zum Beispiel mit Hilfe von Indexen, so kann Phase 1 entfallen. Die Kosten betragen dann lediglich O(N). Auf den Zugriffspfaden werden Scans durchgeführt und für jeweils zwei Schlüssel die Verbundattribute verglichen. Bei Gleichheit werden die zugehörigen Tupel geholt und, falls die Selektionsprädikate erfüllt sind, in die Ergebnismenge übernommen [HR01].

6.1.3 Disjunktive Normalform (DNF)

Die disjunktive Normalform (DNF) einer booleschen Funktion (BF) ist eine Disjunktion von einzelnen Elementarkonjunktionen. Eine BF kann mehrere DNFen haben.
Sei

$$x^0 := \bar{x}$$

$$x^1 := x$$

wobei x eine Variable (Literal) für eine beliebige Bedingung ist, z.B. name == „Doorplate_1“.
Eine Funktion der Form

$$C = x_{i_1}^{\alpha_1} \cdot x_{i_2}^{\alpha_2} \cdot \ldots \cdot x_{i_k}^{\alpha_k}$$

mit $1 \leq i_j \leq n$ (i_j paarweise verschieden), $\alpha_j \in \{0,1\}$ für alle j = 1 ... n heißt Elementarkonjunktion vom Rang k.

Ein Minterm ist eine Elementarkonjunktion, in der alle Variablen vorkommen. Ein Minterm einer BF F nimmt unter genau einer Belegung der Variablen den Wert 1 an. Jede BF F lässt sich als Disjunktion ihrer Minterme schreiben. Eine Elementarkonjunktion C nennt sich Implikant einer BF F, wenn stets gilt: „C wertet sich unter einer Belegung der Variablen zu 1 aus“ impliziert „F wertet sich unter dieser Belegung zu 1 aus“. Ein Implikant C heißt Primimplikant von F, wenn er unverkürzbar ist, d.h. wenn kein weiterer Implikant K für F mit $K \neq C$ existiert, so dass gilt: $\models (K \rightarrow F)$ und $\models (C \rightarrow K)$. Jeder Minterm von F ist in mindestens einem Primimplikanten von F enthalten, daher kann man F auch als Disjunktion aller seiner Primimplikanten schreiben (verkürzte DNF). Jede minimale (unverkürzbare) DNF ist eine Konjunktion aus einer minimalen Menge von Primimplikanten [Ass83].

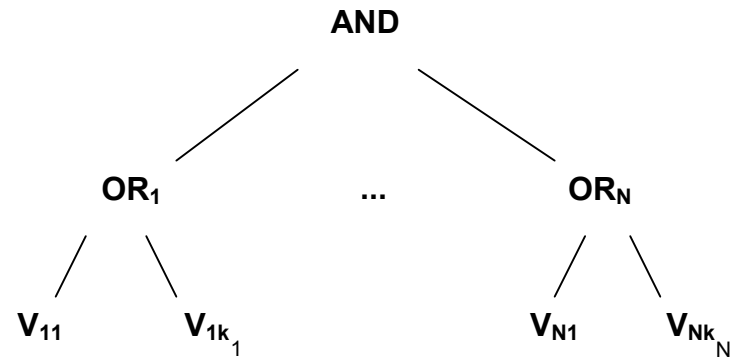

Abb. 6-1) Worst Case bei Anwendung des Distributivgesetzes

Komplexität des verwendeten Algorithmus zur Überführung der AWQL-Restriktion in die disjunktive Normalform (DNF):

Gegeben: AWQL-Restriktion als Baum.

Die Umformung geschieht in drei Schritten, die in der Implementierung jeweils durch eigene Methoden realisiert wurden. Jeder Schritt wendet eine oder mehrere Regeln des Ersetzbarkeitstheorems der Syntax (ET) an [Sch95].

Phase 1: *NOTs mit Hilfe von De Morgan bis nach unten zu den Variablen absinken lassen.* Bei N Knoten und einer durchschnittlichen Höhe des Baumes von log N ergibt dies eine Komplexität von O(N log N), da in jedem Durchlauf ein NOT-Operator höchstens eine Stufe nach unten sinken kann (Anwendung von De Morgan). Steht NOT zu Beginn auf der höchsten Stufe, ergibt das die Höhe des Baumes an Anzahl von Durchläufen. Die dabei auftretenden doppelten Negationen werden entfernt.

Phase 2: *Doppelte ANDs und ORs entfernen.* Bei jedem Durchlauf durch den Baum wird die Anzahl doppelter ANDs und ORs halbiert. Maximal können somit log N Durchläufe entstehen. Die Komplexität ist O(N log N).

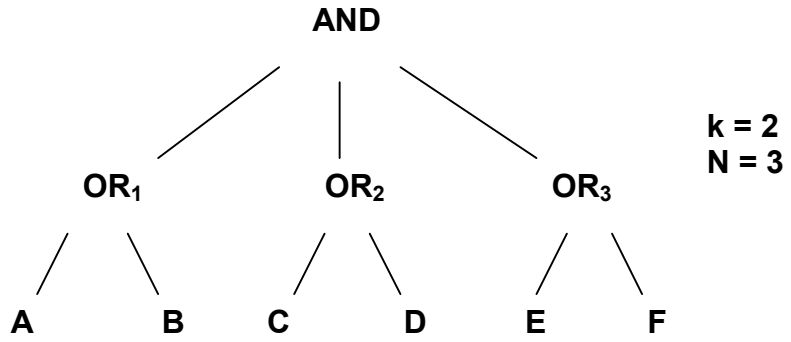

Abb. 6-2) Beispielbaum

Phase 3: *Anwendung des Distributivgesetzes.*

Ausgangsbaum siehe Abbildung 6-1. Die Anzahl der OR-Elemente betrage N.

Sei $k_1 = ... = k_N = k$ die Anzahl der Kinder des i-ten OR-Elements.

Der Algorithmus (Phase 3) wird N-mal durchlaufen, d.h. der Baum wird N-mal traversiert. Die vollständig ausmultiplizierte BF in DNF zum gegebenen Baum besitzt k^N Konjunktionen (AND-Elemente). Für jedes aufgefundene OR-Kindelement eines AND-Elements wird das Distributivgesetz angewandt, d.h. alle Kinder der OR-Elemente werden miteinander multipliziert. Dies geschieht folgendermaßen (siehe Beispiel in Abb. 6-2 und 6-3): In Jedem Durchlauf werden die k Kinder des i-ten OR-Elements ($1 \le i \le N$) k^{N-1} mal kopiert und der Reihe nach mit Wiederholungsintervall k^{N-i} fortlaufend unter den k^N AND-Elementen verteilt (siehe Abb. 6-3). Das ergibt eine Komplexität von $O(k^N)$. Anschließend werden diese k^N AND-Elemente unter ein gemeinsames OR-Element gehängt.

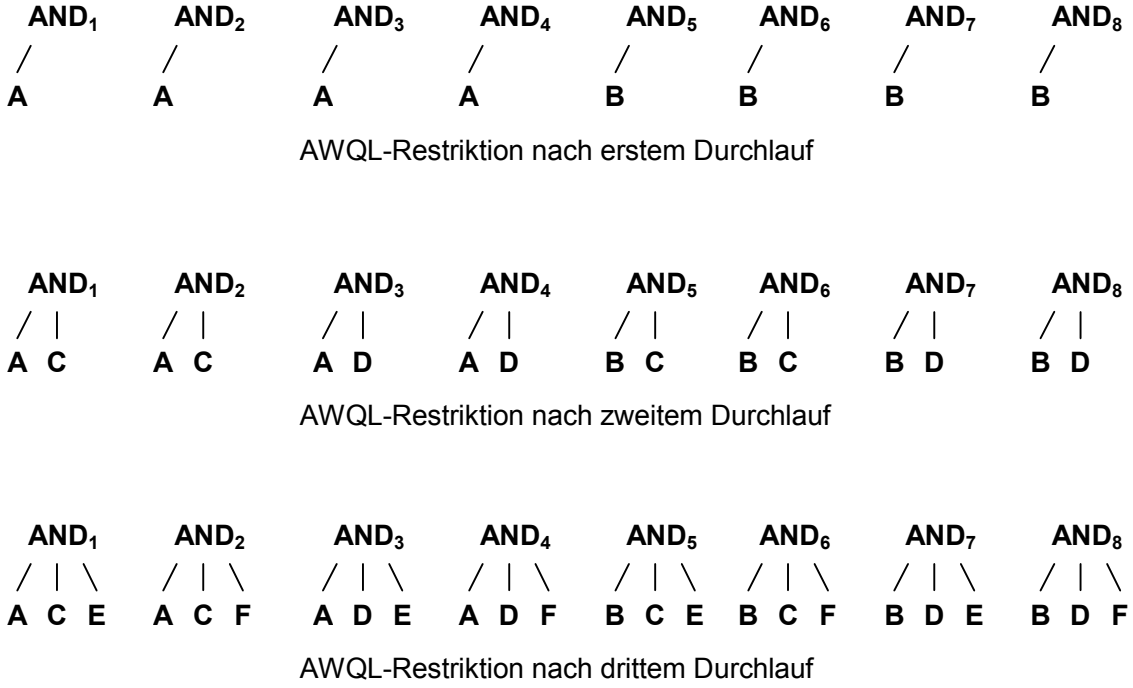

Abb. 6-3) Neue AWQL-Restriktion nach i-tem Durchlauf

Somit hat die Überführung einer AWQL-Restriktion in DNF exponentiellen Aufwand $O(k^N)$.

6.2 Vertikale Partitionierung

Beim Ansatz der vertikalen Partitionierung muss zum Auffinden der richtigen Tabellen die gegebene AWQL-Restriktion zunächst in DNF umgeformt werden. Dieser Schritt hat exponentiellen Aufwand (siehe Kapitel 6.1.4). Jede daraus entstandene Konjunktion stellt eine eigene Teilanfrage dar. Eine solche Anfrage bezieht sich immer auf einen einzelnen Objekttyp bzw. auf eine Hierarchie von Objekttypen. Die in der Teilanfrage spezifizierten Attributkriterien sind hierbei stets auf alle angegebe-

nen Objekttypen anwendbar. Die nachfolgenden theoretischen Untersuchungen beziehen sich immer auf eine einzelne Teilanfrage. Der Gesamtaufwand für eine vollständige AWQL-Anfrage setzt sich aus der Summe der Aufwendungen ihrer einzelnen Teilanfragen zusammen.

6.2.1 Suchaufwand für ein Objekt bei gegebener Objekt-ID

Die Suche nach einem Objekt mit einer bestimmten Objekt-ID gestaltet sich in diesem Ansatz wie folgt: Da nicht bekannt ist, in welcher der zahlreichen Objekttyptabellen sich einzelne Fragmente des gesuchten Objekts befinden, müssen zwangsläufig alle Tabellen durchforstet werden. Unter Verwendung des Primärindexes über den Objekt-IDs berechnet sich der geschätzte Aufwand bei insgesamt p Objekttyptabellen zu:

$$A_{oid} = \sum_{i=1}^{p} (\frac{3}{c} + \frac{1}{b}) = p(\frac{3}{c} + \frac{1}{b})$$

physische Seitenzugriffe bei angenommener Höhe des Indexbaumes von *h*=3.
p = Anzahl der Tabellen, die durchsucht werden müssen.
b = Anzahl Sätze pro Seite.
c = Anzahl Indexelemente pro Seite.
Die Komplexität beträgt $O(\log N_T)$, wobei N_T = Anzahl aller Tupel, da pro Tabelle nur ein Tupel gelesen werden muss und dieses über den Index gefunden werden kann. Der Indexzugriff hat logarithmischen Aufwand (siehe 6.1.1).

Falls man zur Suche der Objekte ausschließlich die Sichten zu jedem Objekttyp verwendet, ergibt sich folgender geschätzter Aufwand:

$$A_{oid} = \frac{p(p+1)}{2}(\frac{3}{c} + \frac{1}{b})$$

physische Seitenzugriffe bei angenommener Höhe des Indexbaumes von *h*=3.
p = Anzahl der Tabellen, die durchsucht werden müssen.
b = Anzahl Sätze pro Seite.
c = Anzahl Indexelemente pro Seite.
Die Komplexität beträgt $O(\log N_T)$, N_T = Anzahl der Tupel.
Es wird hierbei der Worst-Case angenommen, bei dem der Vererbungsgraph eine verkettete Liste darstellt. Auf die Wurzeltabelle wird insgesamt p mal zugegriffen, auf die Blatttabelle nur ein mal. Jedem Tabellenzugriff geht ein Indexzugriff voraus zur Ermittlung des Tupels mit der gesuchten Objekt-ID. Es werden insgesamt $N_T = \frac{p(p+1)}{2}$ Tupel miteinander verbunden. Die Komplexität wird durch den Indexzugriff bestimmt und beträgt $O(\log N_T)$.

6.2.2 Suche nach Objekten mit Angabe des Objekttyps

Ist der Objekttyp bekannt, so gestaltet sich die Suche relativ einfach. Es muss die Tabellensicht des angegebenen Objekttyps sowie die seiner direkten und indirekten

Subtypen betrachtet werden. Der Idealfall tritt ein, wenn zu dem Objekttyp keine Subtypen existieren. Dann bezieht sich die Suche lediglich auf eine einzige Tabellensicht.

Für die Komplexität kann folgendes angegeben werden:
Sei der Vererbungsgraph eine verkettete Liste von Objekttypen und alle Objekte seien vom Blatttyp. In diesem Fall enthält jede Tabelle genau N_T Tupel. Die Höhe betrage p, d.h. es müssen p Tabellen mit je N_T Tupeln verbunden werden. Die Suche auf der Tabellensicht des Blatttyps besitzt dann die Komplexität $O(N_T \log N_T)$, da insgesamt $p-1$ mal verbunden werden muss (z.B. mit SMV siehe 6.1.2). Wenn die Tabellen auf der Objekt-ID zudem einen Clustering-Index besitzen, so werden für Sort-Merge-Joins nur wenige E/A-Operationen benötigt [ME92].

Der Worst-Case hingegen tritt dann ein, wenn der angegebene Objekttyp die Wurzel ist. In diesem Fall müssen alle verfügbaren Tabellensichten durchsucht werden.
Für die Komplexität kann folgendes angegeben werden:
Sei wiederum der Vererbungsgraph eine verkettete Liste und alle Objekte seien vom Blatttyp. Jede Tabelle habe genau N_T Tupel. Die Höhe betrage p. Die gesuchten Objekte sollen mindestens vom Typ der Wurzel sein, somit kommen in diesem Fall alle Objekte in Frage. Die Suche erstreckt sich daher über alle p Tabellensichten. Es müssen insgesamt $P = 1 + 2 + ... + p = \frac{p(p+1)}{2}$ Tabellen mit je N_T Tupeln verbunden werden. Die Komplexität beträgt weiterhin $O(N_T \log N_T)$ bei Verwendung von Sort-Merge-Verbunden (siehe 6.1.2), jedoch ist der konstante Faktor weitaus höher als bei obigem Fall.

6.2.3 Suche ohne Angabe des Objekttyps

Bei diesem häufig vorkommenden Fall werden Objekte nur anhand bestimmter Attributwerte gesucht, ohne dass deren Objekttyp angegeben wird. Da in diesem Fall alle Objekte in Frage kommen können, müssen alle verfügbaren Sichten durchsucht werden. Als Komplexität lässt sich daher der Aufwand für den Worst-Case aus Kapitel 6.2.2 angeben. Die Komplexität ist $O(N_T \log N_T)$.

6.3 Horizontale Partitionierung

Zu Beginn jeder Anfragebearbeitung muss die AWQL-Restriktion in DNF umgeformt werden, da nur auf diese Weise eine komplexe Anfrage auf die richtigen Tabellen abgebildet werden kann. Dieser Schritt hat exponentielle Komplexität (siehe 6.1.3). Jede auf diese Weise entstandene Teilrestriktion bezieht sich auf einen einzelnen Objekttyp bzw. auf eine Menge von Objekttypen, bei denen die enthaltenen Selektionsprädikate anwendbar sind. Die nachfolgenden theoretischen Untersuchungen beziehen sich immer auf eine einzelne Teilanfrage. Der Gesamtaufwand für eine

vollständige AWQL-Anfrage setzt sich aus der Summe der Aufwendungen ihrer einzelnen Teilanfragen zusammen (vgl. 6.2).

6.3.1 Suchaufwand für ein Objekt bei gegebener Objekt-ID

Der Suchaufwand ist derselbe wie bei der vertikalen Partitionierung ohne die Verwendung von Sichten (siehe 6.2.1). Da hier die Verbunde praktisch schon vorab durchgeführt wurden, müssen insgesamt weniger Tabellen und damit Tupel eingelesen werden. Die Komplexität für die Suche mit angegebener Objekt-ID beträgt $O(\log N_T)$ für N_T = Anzahl der Tupel, da der Primärindex ausgenutzt werden kann (siehe 6.1.1).

6.3.2 Suche nach Objekten mit Angabe des Objekttyps

Die Suche nach Objekten bei angegebenem Objekttyp gestaltet sich wie folgt: Zunächst werden alle Subtypen des Objekttyps bestimmt. Im Gegensatz zur vertikalen Partitionierung müssen hier keine Tabellen miteinander verbunden werden. Die Suche beschränkt sich somit ausschließlich auf die gefundenen Objekttyptabellen. Der Aufwand berechnet sich wie folgt:
Sei der Vererbungsgraph eine verkettete Liste mit Länge p. Alle N Objekte seien vom Blatttyp und der angegebene Objekttyp stelle die Wurzel dar. Der geschätzte Aufwand beträgt dann:

$$A_{type} = p\frac{N}{b}$$

physische Seitenzugriffe, da alle p Tabellen ein Mal vollständig durchsucht werden müssen.
b = Anzahl Sätze pro Seite.
Die Komplexität ist somit $O(N_T)$ für N_T = Anzahl Tupel (hier $N_T = pN$).
Im besten Fall, wenn der gesuchte Objekttyp keine Subtypen besitzt, muss nur eine einzige Tabelle durchsucht werden. Die Komplexität beträgt weiterhin $O(N_T)$ jedoch mit geringerem konstanten Faktor.

6.3.3 Suche ohne Angabe des Objekttyps

Wie bei der vertikalen Partitionierung auch, muss bei Fehlen des Objekttyps eine Suche über alle vorhandenen Objekttypen erfolgen. Der maximale Aufwand begrenzt sich somit auf ca. $A = p\frac{N}{b}$ physische Seitenzugriffe (siehe 6.3.2). Die Komplexität beträgt $O(N_T)$.

6.4 Modifizierte horizontale Partitionierung

Beim modifizierten horizontalen Ansatz wird die Tatsache ausgenutzt, dass der Spatial Model Server nur vollständige Objekte zurückliefern darf. Das bedeutet, dass die Suche stets bis zu den Blättern des Vererbungsgraphen verlaufen muss.

Der Aufwand ist in der Regel geringer als beim ursprünglichen Ansatz (siehe 6.3), da hier keine redundanten Objektfragmente gefunden und ausgelesen werden. Die Vorgehensweise bei der Suche ist identisch, jedoch ist der konstante Faktor stets geringer, da weniger Tupel gelesen werden müssen.

6.4.1 Suchaufwand für ein Objekt bei gegebener Objekt-ID

Da nicht bekannt ist, von welchem Typ das gesuchte Objekt ist, müssen alle Objekttyptabellen durchsucht werden. Dies kann jedoch mit Hilfe des Primärindexes über der Objekt-ID geschehen. Der geschätzte Aufwand beträgt bei p Objekttypen somit:

$$A_{oid} = p\frac{3}{c} + 1$$

physische Seitenzugriffe bei angenommener Höhe des Indexbaumes von h=3.
p = Anzahl der Tabellen, die durchsucht werden müssen.
c = Anzahl Indexelemente pro Seite.
Die Komplexität beträgt $O(\log N)$ (Indexzugriff bei N Objekten).
Da ein Objekt nicht auf mehrere Tabellen verteilt wird, sondern vollständig in seiner Objekttyptabelle gespeichert wird, müssen keine unnötigen Tupel ausgelesen werden. Es reichen Indexzugriffe zum Finden des richtigen Tupels aus. Die Gesamtkomplexität dieser Suche wird demnach durch den Indexzugriff bestimmt.

6.4.2 Suche nach Objekten mit Angabe des Objekttyps

Die Suche nach Objekten bei angegebenem Objekttyp gestaltet sich auf identische Weise wie im unmodifizierten Ansatz (siehe 6.3.2). Zunächst werden alle Subtypen des Objekttyps bestimmt. Die Suche beschränkt sich ausschließlich auf die gefundenen Objekttyptabellen. Da in diesem Fall aber nur vollständige Objekte gespeichert werden, müssen schlimmstenfalls auch nur maximal N Objekte sprich Tupel verglichen werden. Der Aufwand berechnet sich somit wie folgt:
Sei der Vererbungsgraph eine verkettete Liste mit Länge p. Alle Objekte seien vom Blatttyp und der angegebene Objekttyp stelle die Wurzel dar. Der geschätzte Aufwand beträgt dann:

$$A_{type} = (p-1) + \frac{N}{b}$$

physische Seitenzugriffe, wenn man für jeden Tabellenzugriff als Minimum einen Seitenzugriff veranschlagt, auch wenn diese leer ist.
b = Anzahl Sätze pro Seite.

Die Komplexität ist somit $O(N)$. Der konstante Faktor ist hier jedoch weitaus geringer, als bei der ursprünglichen horizontalen Partitionierung, da effektiv weniger Tupel gelesen werden müssen.

6.4.3 Suche ohne Angabe des Objekttyps

Der Aufwand ist identisch mit dem Suchaufwand für Objekte vom Wurzeltyp (siehe 6.4.2). Hier müssen in jedem Fall alle Objekttyptabellen durchsucht werden. Die Komplexität beträgt also bei N Objekten (= N_T Tupeln) $O(N)$.

6.5 Hierarchischer Ansatz

Bei diesem Ansatz kann auf die Umformung der AWQL-Restriktion nach DNF verzichtet werden. Eine AWQL-Anfrage kann direkt in SQL umgeformt werden, da es nur eine einzige Tabelle (Mastertabelle) gibt und alle Attribute der Objekte darin zu finden sind. Es entstehen also nicht die u.U. sehr zahlreichen Teilanfragen auf verschiedene Objekttyptabellen, wie in den vorherigen Ansätzen. Diese Mastertabelle ist jedoch sehr umfangreich und es besteht leicht die Gefahr, dass sie die Seitengrenze von 32 KB (DB2) überschreitet. Auch muss darauf geachtet werden, dass die maximale Anzahl von Spalten (1012 pro Tabelle bei DB2 [IBM02]) nicht überschritten wird (siehe dazu Implementierung in Kapitel 7.3).

6.5.1 Suchaufwand für ein Objekt bei gegebener Objekt-ID

Da die Objekt-ID Primärschlüssel ist, kann die Suche nach dem passenden Objekt mit Hilfe des Primärindexes geschehen. Der Aufwand beträgt dann ungefähr

$$A_{oid} = \frac{3}{c} + 1$$

physische Seitenzugriffe bei angenommener Höhe des Indexbaumes von *h*=3.
c = Anzahl Indexelemente pro Seite.
Die Komplexität ist $O(\log N)$.

6.5.2 Suche nach Objekten mit Angabe des Objekttyps

Zunächst wird der Objekttyp expandiert, d.h. es werden sämtliche Subtypen bestimmt. Unter der Verwendung des IN-Operators kann der Typvergleich auch auf Mengen durchgeführt werden. Daher reicht hier nur eine einzelne Anfrage, um alle Objekte zu finden (`select * from mastertable where type in ('NexusObject', 'NexusDataObject',...)`).

Falls auf der Spalte „type“ kein Sekundärindex definiert ist, muss zum Auffinden der gesuchten Objekte ein Tablescan durchgeführt werden. Dabei ist zu berücksichtigen, dass ein Tupel aufgrund seiner Größe eine ganze Seite füllen kann. Der geschätzte Aufwand beträgt in diesem Fall dann

$A_{type} = N$
physische Seitenzugriffe.
N = Anzahl der Objekte der Tabelle (entspricht der Anzahl an Tupeln).
Die Komplexität ist $O(N)$.

Mit einem Sekundärindex auf „type“ kommt zum Indexzugriff von 3 bis 4 Seiten noch der Aufwand zum Auslesen der Objekte hinzu (1 Seitenzugriff pro Objekt).
Die Komplexität beträgt $O(\log N + N_{found})$, wobei N_{found} = Anzahl gefundener Objekte $\leq N$. Falls alle Objekte ausgewählt wurden beträgt die Komplexität $O(N)$.

6.5.3 Suche ohne Angabe des Objekttyps

Falls kein Objekttyp angegeben wurde, so wird ein vollständiger Tablescan durchgeführt. Der Aufwand beträgt
$A = N$
physische Seitenzugriffe.
N = Anzahl der Objekte/Tupel der Tabelle.
Die Komplexität ist $O(N)$.

6.6 Modifizierter hierarchischer Ansatz

Der modifizierte hierarchische Ansatz unterscheidet sich zum Vorgänger nur in der besonderen Behandlung des „type“-Attributs. Hier kann ebenfalls auf die Umformung der AWQL-Anfrage nach DNF verzichtet werden.

6.6.1 Suchaufwand für ein Objekt bei gegebener Objekt-ID

Der Suchaufwand ist identisch mit dem Aufwand der unmodifizierten Version dieses Ansatzes (siehe 6.5.1).

6.6.2 Suche nach Objekten mit Angabe des Objekttyps

In diesem Ansatz muss der Objekttyp nicht mehr expandiert werden (vgl. 6.5.2). Die Suche kann mit dem Operator LIKE durchgeführt werden.

Falls auf der Spalte „type“ kein Sekundärindex definiert werden kann – weil der Schlüssel[1] zu groß ist –, muss zum Auffinden der gesuchten Objekte ein Tablescan durchgeführt werden. Dabei ist zu berücksichtigen, dass ein Tupel aufgrund seiner Größe eine ganze Seite füllen kann. Der geschätzte Aufwand beträgt dann
$A_{type} = N$
physische Seitenzugriffe.

[1] Bei der hier verwendeten Datenbank (DB2 UDB V8.1) liegt die maximale Schlüssellänge inkl. Overhead bei 1024 Bytes [IBM02].

N = Anzahl der Objekte/Tupel der Tabelle.
Die Komplexität ist $O(N)$.

Mit Sekundärindex auf „type“ kommt zum Indexzugriff von 3 bis 4 Seiten noch der Aufwand zum Auslesen der Objekte hinzu (1 Seitenzugriff pro Objekt).
Die Komplexität beträgt $O(\log N)$.

6.6.3 Suche ohne Angabe des Objekttyps

Der Aufwand ist identisch mit 6.5.3.

6.7 Attributtabellenansatz

Bei diesem Ansatz kann auf die Umformung der AWQL-Restriktion nach DNF ebenfalls verzichtet werden. Eine AWQL-Anfrage kann somit direkt in SQL umgeformt werden. Es existiert eine einzelne Attributtabelle, in der alle Objektattribute zeilenweise gespeichert werden. Boolesche Operationen können auf Mengenoperationen auf dieser Tabelle abgebildet werden. Damit bei Attributvergleichen kein Tablescan durchgeführt werden muss, wird auf der Spalte „attribute“ ein Index erstellt. Auf diese Weise müssen nur die Tupel eingelesen werden, die tatsächlich gebraucht werden.
Eine Suche gliedert sich aufgrund der Trennung von Objekt und Attribut stets in zwei Phasen: In der ersten Phase werden die Objekt-IDs der in Frage kommenden Objekte gesucht. In der zweiten Phase werden mit Hilfe dieser gefundenen Objekt-IDs alle Objektattribute ausgelesen und in den Spatial Model Server übertragen. Diese beiden Phasen können in einer geschachtelten SQL-Anfrage miteinander verknüpft werden, so dass sie letztlich nur einen Datenbankaufruf benötigen.

6.7.1 Suchaufwand für ein Objekt bei gegebener Objekt-ID

Die Suche anhand der Objekt-ID gestaltet sich denkbar einfach. Da die Objekt-ID bereits bekannt ist, entfällt im Grunde die eigentliche Suche (Phase 1). Es wird gleich mit der 2. Phase begonnen und alle Attribute mit der angegebenen OID ausgelesen. Der geschätzte Aufwand beträgt dann:

$$A_{oid} = \frac{3}{c} + \frac{k}{b}$$

physische Seitenzugriffe bei angenommener Höhe des Indexbaumes von h=3. Es wird hierbei angenommen, dass ein Sekundärindex über den Objekt-IDs vorhanden ist, der das sequentielle Auslesen in Sortierreihenfolge erlaubt.
k = Anzahl der Objektattribute.
b = Anzahl Sätze pro Seite.
c = Anzahl Indexelemente pro Seite.
Die Komplexität ist $O(\log N)$.

6.7.2 Suche nach Objekten mit Angabe des Objekttyps

Die Suche nach Objekten eines bestimmten Objekttyps gestaltet sich wie folgt: Zunächst werden sämtliche Subtypen des angegebenen Objekttyps bestimmt. Eine separate Tabelle namens `ObjectType`, die zu jeder Objekt-ID den dazugehörigen Objekttyp speichert, kann für die Bestimmung der OIDs herangezogen werden. Ein Index über dessen Spalten verhindert einen sonst notwendigen Tablescan. Der Aufwand für diesen Schritt beträgt schätzungsweise:

$$A_{type} = p\frac{3}{c}$$

physische Seitenzugriffe bei angenommener Höhe des Indexbaumes von *h*=3 und einem Sekundärindex über den Objekttypnamen, der sequentiellen Zugriff in Sortierreihenfolge zulässt.
p = Anzahl der Subtypen inklusive angegebenem Objekttyp.
c = Anzahl Indexelemente pro Seite.
Die Komplexität ist $O(\log N)$.
Falls weitere Suchkriterien gegeben sind (der Normalfall), werden diese auf geschachtelte Unteranfragen abgebildet, deren Ergebnismengen anschließend gemäß ihrer booleschen Verknüpfungen miteinander verbunden werden. Die Suche von Attributen und damit von Objekten, die ein bestimmtes Prädikat erfüllen, hat einen geschätzten maximalen Aufwand pro einzelner Attributbedingung von

$$A_{attribute} = \frac{3}{c} + N$$

physische Seitenzugriffe bei angenommener Höhe des Indexbaumes von *h*=3.
N = Anzahl der Objekte.
c = Anzahl der Indexelemente pro Seite.
Es wird hierbei angenommen, dass ein Sekundärindex über dem Attributnamen vorhanden ist und alle Objekte das gesuchte Attribut besitzen (z.B. für Wurzelattribute der Fall).
Die Komplexität pro Attributbedingung beträgt also $O(\log N)$ für die Indexsuche und $O(N)$ für das Auslesen und Vergleichen der Attributwerte.
Die Verbundoperationen setzen in der Regel sortierte Mengen voraus (z.B. SMV siehe 6.1.2), daher ist die Komplexität durch den Sortieralgorithmus gegeben. Es wird somit eine Komplexität von $O(N \log N)$ für sämtliche anfallenden Verbundoperationen angenommen.

Die anschließende Auslesephase kann mit Hilfe eines Indexes über den Objekt-IDs in logarithmischer bis linearer Zeit durchgeführt werden (Indexzugriff + Lesen der Tupel). Die Komplexität hierfür beträgt $O(\log N + N_{found})$, für N_{found} = Anzahl Attribute der gefundenen Objekte.

Der Gesamtaufwand hängt allgemein stark von der Komplexität der Anfrage ab. Je mehr Attribute verglichen werden müssen, desto länger dauert die Suchphase und

desto häufiger müssen Verbundoperationen durchgeführt werden. Die Gesamtkomplexität einer Anfrage beträgt $O(N \log N)$.

6.7.3 Suche ohne Angabe des Objekttyps

Die Suchanfragen nach Objekten mit oder ohne Angabe des Objekttyps sind in ihrem Aufwand praktisch identisch. Bei letzterem fällt lediglich die Bestimmung der qualifizierenden Objekttypen weg und damit der zusätzliche Schnitt mit der Menge von gültigen Objekt-IDs aus der `ObjectType`-Tabelle. Die Komplexität beträgt $O(N \log N)$.

6.8 Erweiterter objektrelationaler Ansatz

Der erweiterte objektrelationale Ansatz stellt eine Weiterführung der aktuellen Spatial Model Server Implementierung dar. Es wird lediglich die Mehrfachvererbung hinzugefügt. Der Aufwand ist vergleichbar mit der horizontalen Partitionierung, wobei hier jedoch zusätzlich Sichten zum Einsatz kommen, die unerwünschte Objektfragmente und Duplikate ausfiltern.

Zunächst muss wie bisher eine gegebene AWQL-Restriktion in DNF umgeformt werden. Dieser Schritt hat exponentiellen Aufwand (siehe Kapitel 6.1.3). Jede daraus entstandene Konjunktion stellt eine eigene Teilanfrage dar, die sich auf einen einzelnen Objekttyp bzw. auf eine Hierarchie von Objekttypen bezieht. Der Gesamtaufwand für eine vollständige AWQL-Anfrage setzt sich aus der Summe der Aufwendungen ihrer einzelnen Teilanfragen zusammen.

6.8.1 Suchaufwand für ein Objekt bei gegebener Objekt-ID

Es müssen wie im horizontalen Partitionierungsansatz sämtliche Objekttyptabellen durchsucht werden. Hinzu kommt der Mehraufwand für die Verarbeitung der Sichten auf den jeweiligen Tabellen. Bei mehrfachvererbten Objekten wird mit Hilfe des Zeigerkonstruktes auf die Attribute der Supertypobjekte zugegriffen. Dies verursacht in der Regel weitere Seitenzugriffe pro Referenz. Der Aufwand ist somit höher als bei der jetzigen Implementierung. Da nach [CCN+99] getypte Tabellen intern in DB2 mit einem hierarchischen Ansatz gespeichert werden, ist dieser Ansatz eher vergleichbar mit der horizontalen Partitionierung (siehe 6.3). Die Objekttyptabellen können hier als spezielle Sichten auf der hierarchischen Tabelle angesehen werden. Durch die notwendige Umformung der AWQL-Restriktion nach DNF und die daraus resultierende hohe Anzahl an einzelnen Teilanfragen, geht der Vorteil der hierarchischen Speicherung jedoch verloren. Die Komplexität beträgt schätzungsweise $O(\log N_T)$, wobei N_T = Anzahl aller Tupel.

6.8.2 Suche nach Objekten mit Angabe des Objekttyps

Der Aufwand ist schätzungsweise vergleichbar mit 6.3.2, wobei hier jedoch der Mehraufwand für die Sichten und die Auswertung der Objektreferenzen hinzukommt.

6.8.3 Suche ohne Angabe des Objekttyps

Der Aufwand für diesen Anfragetyp ist vergleichbar mit 6.3.3. Es kommt jedoch der Mehraufwand für die Sichten und die Auswertung der Objektreferenzen hinzu.

6.9 Zusammenfassung der theoretischen Betrachtungen

In Tabelle 6-1 wurden die verschiedenen Komplexitäten der gefundenen Schemata gegenübergestellt.

Suche	Vert. A. (*)	Hor. A. (*)	Mod. hor. A. (*)	Hier. A.	Mod. hier. A.	Attr.-A.	Erw. OR. A. (*)
DNF-Umf.	$O(2^R)$	$O(2^R)$	$O(2^R)$	-	-	-	$O(2^R)$
Mit OID	$O(\log N_T)$	$O(\log N_T)$	$O(\log N_T)$	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log N_T)$
Mit Type	$O(N_T \log N_T)$	$O(N_T)$	$O(N_T)$	$O(N)$	$O(N)$	$O(N \log N)$ (***)	$O(N_T)$
Ohne Type	$O(N_T \log N_T)$ (**)	$O(N_T)$ (**)	$O(N_T)$ (**)	$O(N)$ (**)	$O(N)$ (**)	$O(N \log N)$ (***)	$O(N_T)$ (**)

N_T = Anzahl Tupel, N = Anzahl Objekte, $N_T \geq N$.

(*) hinzu kommt jeweils die Komplexität für die DNF-Umformung von
R = Anzahl Attribute in AWQL-Restriktion.

(**) konstanter Faktor höher als bei Suche mit Type

(***) konstanter Faktor abhängig von Anzahl der Attribute in AWQL-Restriktion

Tabelle 6-1) Gegenüberstellung der theoretischen Bewertungen

Allgemein kann gesagt werden, dass der Gesamtaufwand stets von der Anzahl der Attribute in der AWQL-Restriktion abhängt. Eine Abschätzung des Bearbeitungsaufwandes ist schwierig, da dies von der Implementierung der jeweiligen Datenbank abhängt. In den ersten drei Ansätzen spielen zudem die booleschen Verknüpfungen der Attribute untereinander eine Rolle. Davon hängt direkt die Laufzeit der DNF-Umwandlung ab. Je mehr AWQL-Dokumente aus einer Anfrage generiert werden müssen, desto länger dauert die Bearbeitung der Anfrage, da jedes entstandene Dokument eine Datenbankanfrage generiert. Im Attributtabellenansatz bestimmt die Anzahl der Attribute die Häufigkeit, mit der die gesamte Attributtabelle durchsucht wird und somit, wie häufig die Ergebnismengen miteinander verknüpft werden müssen. Hier fällt jedoch, wie auch in den beiden hierarchischen Ansätzen, die Umformung der AWQL-Restriktion weg. Als Folge dessen wird nur eine einzige Daten-

bankanfrage generiert (davon ausgenommen sind Updates, die in der Regel mehrere Anfragen benötigen). Der konstante Faktor ist somit bei den Ansätzen ohne DNF bei gleicher Komplexität stets viel geringer, als bei denen mit DNF.

6.10 Vorauswahl

Für die weitere praktische Implementierung wurde an dieser Stelle bereits eine Vorauswahl getroffen und die Ansätze anhand ihrer zu erwartenden Leistungen sortiert. Der erweiterte objektrelationale Ansatz wird nicht weiter verfolgt, da bei diesem keine Geschwindigkeitssteigerung zum bisherigen Spatial Model Server zu erwarten ist. Als sehr viel versprechend gelten die Ansätze „Hierarchischer Ansatz“ , „modifizierter hierarchischer Ansatz“ und der Attributtabellenansatz. Da diese Ansätze auf die Umformung von Anfragen nach DNF verzichten, wurden zum Vergleich ebenfalls die Ansätze „modifizierte horizontale Partitionierung“ und „vertikale Partitionierung“ implementiert, die beide auf die DNF-Umformung angewiesen sind.

7 Implementierung

Insgesamt wurden sieben Ansätze implementiert. Neben den ausgewählten fünf aus dem vorherigen Kapitel wurden zwei weitere Variationen der beiden hierarchischen Ansätze realisiert, die sich während der Implementierungsphase ergeben haben. Für die geplanten Performancetests gegen Ende wurden die einzelnen Spatial Model Server nicht als Entity Java Beans (EJB) implementiert, wie dies in der alten Version getan wurde, sondern vielmehr als eigenständige Kommandozeilenprogramme realisiert. Dadurch wurde der notwendige Kommunikationsaufwand zwischen Testprogramm und Spatial Model Server minimiert, da auf die Verwendung von SOAP gänzlich verzichtet werden konnte. Die Messungen zeigten so viel direkter die Auswirkungen des zugrunde liegenden Datenbankschemas. Als besondere Zugabe wurde der Siegeransatz dennoch zusätzlich als Web-Service implementiert, so dass er einen einsatzbereiten SpaSe darstellt. Auf seine Implementierung kann in dieser Arbeit jedoch nicht eingegangen werden, da dies nicht Bestandteil der Diplomarbeit war.

In den nachfolgenden Unterkapiteln werden alle Implementierungen der Reihe nach vorgestellt. Zu jeder vorgestellten Javaklasse wird, falls möglich, die zugrunde liegende EJB-Klasse als Basis aufgeführt, so dass eine einfachere Zuordnung zum bestehenden Spatial Model Server möglich ist. Die Klassen `ResultHashMap`, `QueryData`, `AttributesNameMapper`, `ResultColumn`, `ResultRow` und `SpaSeErrorHandler` wurden ohne Veränderung direkt aus der vorliegenden alten SpaSe-Implementierung übernommen. Diese sind im Verzeichnis `de.uni_stuttgart.nexus.spaSe.common` zu finden. Des Weiteren verwenden die einzelnen Ansätze die Klassen `Result2CRLApp` und `Results2AWMLApp` gemeinsam. Letztere sind umgeschriebene Klassen der EJBs `Result2CRLBean` und `Results2AWMLBean`. Zusätzlich verwenden alle Implementierungen die neue Klasse `AWMLDocument`, welche AWML-Dokumente einlesen und objektweise zurückgeben kann. Für die Verwaltung der Vererbungshierarchie verwenden alle Ansätze die Klasse `HierarchyContainer`. Diese wird am Ende dieses Kapitels näher erläutert.

Den Einstiegspunkt bildet jeweils die Klasse `QueryComponentApproach`, welche aus dem EJB `QueryComponentBean` konstruiert wurde. Darin sind die Methoden `query`, `insert` und `update_delete` definiert, welche die Grundfunktionen des Spatial Model Servers darstellen. Um einen Kommandozeilen-SpaSe zu einem jeweiligen Ansatz zu erzeugen, muss ein neues Objekt dieser Klasse mit Hilfe des parameterlosen Konstruktors angelegt werden.

7.1 Ansatz 1: Vertikale Partitionierung

Zu dem vertikalen Ansatz gehören im Wesentlichen die Klassen `AWQL2SQLApproach1`, `AWML2SQLApproach2`, `DBAccessApproach1` und `Query-`

ComponentApproach1. Die Klasse HierarchyContainer wird von allen Implementierungen gemeinsam genutzt.

7.1.1 QueryComponentApproach1

Basisklasse: QueryComponentBean
Package: de.uni_stuttgart.nexus.spaSe.approach1

Diese Klasse stellt den Einstiegspunkt zur vorliegenden SpaSe-Implementierung dar. Sie stellt die Grundmethoden query(), insert() und update_delete() bereit. Da diese Methoden in allen Implementierungen vorhanden sind und lediglich weitere Untermethoden aus anderen Klassen aufrufen, werden sie nur stellvertretend am ersten Ansatz vorgestellt. Die einzelnen Implementierungen weichen im Wesentlichen an dieser Stelle nur in der Realisierung der update_delete()-Methoden voneinander ab. Diese Methoden werden daher gesondert vorgestellt.

Es existieren folgende Methoden in dieser Klasse:

```
public String query(String awqlQuery)
      throws ServerException
```

Beschreibung:

Diese Methode wird zum Ausführen von AWQL-Query-Anfragen verwendet. Das eingehende AWQL-Dokument awqlQuery wird mit Hilfe der Methode awql2sql() der Klasse AWQL2SQLApproach1 in eine Menge von SQL-Statements umgewandelt. Anschließend wird diese Menge auf der DB ausgeführt (Aufruf query() der Klasse DBAccessApproach1). Das Ergebnis wird in ein Objekt vom Typ ResultHashMap eingetragen. Abschließend werden die darin enthaltenen Objekte zu einem AWML-Dokument umgewandelt und als Ergebnis dieses Methodenaufrufs zurückgeliefert (Aufruf results2AWML() der gemeinsamen Klasse Results2AWMLApp).

```
public String update_delete(String awql)
      throws ServerException
```

Beschreibung:

Diese Methode wird zum Ausführen von AWQL-Update/Delete-Anweisungen verwendet.

Vorgehensweise beim Ändern von Objekten:

Das Ändern von Nexusobjekten geschieht in mehreren Phasen: Im ersten Schritt werden alle in Frage kommenden Objekte gesucht und ihre NOLs und Typangaben in einer internen Ergebnismenge gespeichert. Dazu wird das eingehende AWQL-Dokument zunächst nach SQL umgewandelt. Dies erledigt die Methode awql2sql() der Klasse AWQL2SQLApproach1). Die SQL-Statements werden mit

Hilfe der `query()`-Methode der Klasse `DBAccessApproach1` ausgeführt. Das Ergebnis füllt eine `ResultHashMap`. Im zweiten Schritt werden die gefundenen Objekte sukzessive geändert. Dies geschieht mit Hilfe der Methode `updateFoundObjects()`. Die Menge der erzeugten SQL-Update-Anweisungen wird mit Hilfe der Methode `change()` der Klasse `DBAccessApproach1` innerhalb einer atomaren Transaktion ausgeführt. Das Ergebnis wird anschließend zunächst in einem Objekt vom Typ `ResultHashMap` gespeichert und schließlich in einem weiteren Schritt zu einem Changed-Result-Dokument umgeformt und als String zurückgegeben.

Vorgehensweise beim Löschen von Objekten:

Das Löschen von Nexusobjekten geschieht analog zum vorherigen Ändern ebenfalls in zwei Phasen, jedoch wird hier im zweiten Schritt statt der Methode `updateFoundObjects()`, `deleteFoundObjects()` aufgerufen.

```
public String insert(String awml)
      throws ServerException
```

Beschreibung:

Diese Methode wird zum Ausführen von AWML-Insert-Anfragen verwendet. Sie ist nicht Bestandteil der Diplomarbeit und wurde nur der Vollständigkeit halber implementiert. Die Beschreibung der Vorgehensweise entfällt an dieser Stelle.

7.1.2 AWQL2SQLApproach1

Basisklasse: `AWQL2SQLBean`
Package: `de.uni_stuttgart.nexus.spaSe.approach1`

Diese Klasse stellt im Wesentlichen die Methode zur Umformung einer eingehenden AWQL-Anfrage nach SQL bereit.

```
public QueryData awql2SQL(String awql, int mode)
      throws JDOMException, ServerException, SAXException
```

Beschreibung:

Diese Methode wandelt ein AWQL-Dokument in eine Menge von SQL-Statements um. Sie ist das eigentliche Herzstück des Spatial Model Servers.

Vorgehensweise:

Das AWQL-Dokument `awql` wird zunächst gegen das Schema validiert. Dazu wird der angegebene Webserver kontaktiert. Falls es sich um ein Update handelt, so werden die Änderungsanweisungen der einzelnen Attribute zwischengespeichert.

Falls es sich um ein Delete-Statement handelt, so wird aus dem AWQL-Dokument zunächst das „filter“-Elemente überschrieben bzw. neu gesetzt, so dass lediglich die Attribute „nol“ und „type“ selektiert werden, da nur diese für das Löschen von Ob-

jekten benötigt werden (mit der Angabe der NOL ist das Objekt eindeutig bestimmt, die Angabe des Typs erleichtert das Auffinden).

Im dritten Schritt wird das „excludes“-Filter ausgewertet und die darin enthaltenen Attribute in einer `ArrayList` abgelegt. Diese wird zusammen mit den generierten SQL-Statements in einem Objekt vom Typ `QueryData` als Ergebnis am Ende dieses Methodenaufrufs zurückgeliefert.

Im nächsten Schritt werden alle „in“-Elemente der AWQL-Restriktion zu Kindern eines neuen „or“-Elements umgewandelt. Anschließend folgt die Überführung der gesamten Restriktion in die DNF-Darstellung (siehe Algorithmus in 6.1.3).

Im Schritt 5 folgt die Aufsplittung des AWQL-Dokuments in mehrere kleinere mit je einer Konjunktion als Restriktion. Dazu wird das ganze Dokument kopiert und die Restriktion bis auf das gewünschte Kind-Element ausgehöhlt.

In Schritt 6 wird jedes AWQL-Dokument nochmals traversiert, die „Table“-Elemente in Subtables aufgesplittet und miteinander geschnitten. Zur Erinnerung: Ein AWQL-Dokument enthält hier nur eine Konjunktion als Restriktion, somit müssen nur die gemeinsamen Tabellen betrachtet werden. Falls kein „Table“-Element enthalten war, so wird die Wurzel „NexusDataObject“ angenommen und davon die Subtypenmenge bestimmt. Die entstandene „Table“-Schnittmenge wird nun als neues „Table“-Element eingesetzt.

Im nächsten Schritt werden die Dokumente weiter aufgesplittet und zu jedem einzelnen „Table“-Eintrag ein separates Dokument erstellt.

Im 8. Schritt werden all die Dokumente entfernt, die nach Objektattributen verlangen, welche nicht im jeweiligen Objekttyp (durch den „Table“-Eintrag bestimmt) enthalten sind. Anschließend werden diejenigen Dokumente miteinander vereint, die auf dieselben Tabellen zugreifen. Ihre enthaltenen Restriktionen werden mit „or“ verknüpft.

Im letzten Schritt werden schließlich aus den erzeugten AWQL-Dokumenten die SQL-Statements generiert und in die Ergebnismenge eingetragen. Dies geschieht mittels des Methodenaufrufs `transform()`. Als Gesamtergebnis dieses Methodenaufrufs wird ein Objekt vom Typ `QueryData` zurückgegeben, welches die Menge aller SQL-Statements und die Excludes-Menge enthält.

7.1.3 AWML2SQLApproach1

Basisklasse: `AWML2SQLBean`
Package: `de.uni_stuttgart.nexus.spaSe.approach1`

Die Klasse `AWML2SQLApproach1` stellt Methoden zur Generierung von SQL-Insert-Statements aus AWML-Dokumenten bereit. Da dies nicht Gegenstand dieser Diplomarbeit war, muss auf eine genauere Erläuterung der Vorgehensweise verzichtet werden. Der Vollständigkeit halber wurde diese Klasse dennoch implementiert.

7.1.4 DBAccessApproach1

Basisklasse: `DBAccessBean`
Package: `de.uni_stuttgart.nexus.spaSe.approach1`

Die Klasse `DBAccessApproach1` stellt den Zugriff auf die Datenbank bereit. Über die Methoden `query()` und `change()` können SQL-Statements an die Datenbank gesendet werden. Des Weiteren bietet diese Klasse Methoden zum Auslesen der Metadatentabellen an. Jeder Ansatz verwendet seine eigene DB-Klasse. Im Wesentlichen unterscheiden sie sich nur geringfügig, daher wird an dieser Stelle nur der erste Ansatz erläutert.

```
public ResultHashMap query(QueryData queryObject)
```

Beschreibung:

Diese Methode führt die in `queryObject` enthaltenen SQL-Statements auf der DB aus. Die einzelnen Ergebnismengen werden in einem Objekt vom Typ `ResultHashMap` gesammelt. Sie bedient sich dazu der Methode `addResultSet2HashMap()`, die unverändert aus der alten Implementierung des SpaSes übernommen wurde und daher nicht weiter beschrieben wird.

```
public ResultHashMap change(QueryData queryObject)
```

Beschreibung:

Diese Methode setzt die in `queryObject` enthaltenen Update/Delete-Anweisungen auf der DB ab. Die einzelnen Ergebnismengen werden in einem Objekt vom Typ `ResultHashMap` gesammelt.

```
public String mapToTableName(String objectType)
```

Beschreibung:

Diese Methode liefert zu einem Objekttyp die dazugehörige Objekttyptabelle zurück. Es wird dazu die Tabelle `TableName` ausgelesen. Auf diese Weise können Objekttypen beliebige Bezeichner annehmen und sind nicht an die Notation der Datenbank gebunden.

7.2 Ansatz 2: Modifizierte horizontale Partitionierung

Nachfolgend werden die Methoden der modifizierten horizontalen Partitionierung erläutert, die von den bereits bekannten Implementierungen abweichen. Um den Rahmen dieser Ausarbeitung nicht zu sprengen, wurde auf die Erläuterung der Hilfsmethoden weitestgehend verzichtet.

7.2.1 QueryComponentApproach2

Basisklasse: `QueryComponentBean`
Package: `de.uni_stuttgart.nexus.spaSe.approach2`

Diese Klasse stellt den Einstiegspunkt zur vorliegenden SpaSe-Implementierung dar. Sie stellt die Grundmethoden `query()`, `insert()` und `update_delete()` bereit (siehe 7.1.1).

```
public String update_delete(String awql)
      throws ServerException
```

Beschreibung:

Diese Methode wird zum Ausführen von AWQL-Update/Delete-Anweisungen verwendet.

Vorgehensweise beim Ändern oder Löschen von Objekten:

Das Ändern von Nexusobjekten geschieht in mehreren Phasen: Im ersten Schritt wird das AWQL-Dokument in eine Menge von SQL-Statements umgewandelt. Dies erledigt die Methode `awql2sql()` der Klasse `AWQL2SQLApproach2`. Im zweiten Schritt wird die Menge der erzeugten SQL-Anweisungen mit Hilfe der Methode `change()` der Klasse `DBAccessApproach2` innerhalb einer atomaren Transaktion ausgeführt. Das Ergebnis wird anschließend zunächst in einem Objekt vom Typ `ResultHashMap` gespeichert und schließlich in einem weiteren Schritt zu einem Changed-Result-Dokument umgeformt und als String zurückgegeben (Aufruf `result2AWML()`, Klasse `Results2AWMLApp`).

7.2.2 AWQL2SQLApproach2

Basisklasse: `AWQL2SQLBean`
Package: `de.uni_stuttgart.nexus.spaSe.approach2`

Diese Klasse stellt die Methode `awql2sql()` zur Umformung einer eingehenden AWQL-Anfrage nach SQL bereit.

```
public QueryData awql2SQL(String awql, int mode)
      throws JDOMException, ServerException, SAXException
```

Beschreibung:

Wandelt ein AWQL-Dokument in SQL-Statements um.

Vorgehensweise:

Die Vorgehensweise ist nahezu identisch mit dem vertikalen Ansatz 1 (siehe 7.1.2). Die spezielle Behandlung von Delete-Anweisungen entfällt hier jedoch. Die gefundenen Objekte können direkt geändert werden.

7.3 Ansatz 3.1: Hierarchischer Ansatz

Nachfolgend werden die wesentlichen Methoden des hierarchischen Ansatzes vorgestellt. Während der Implementierungsphase hat sich herausgestellt, dass es nicht ohne weiteres möglich ist, alle Attribute sämtlicher Nexusobjekte in einer einzigen Tabelle zu speichern. Die Seitengröße von 32 KB[2] setzt dem einen Riegel vor. Daher mussten die Attribute größtenteils als Large Objects (LOB) deklariert werden, da diese nicht wie primitive Datentypen inline gespeichert werden, sondern lediglich eine Referenz auf ein Datenobjekt diesen Typs erstellt wird, das sich wiederum auf einer anderen Seite befinden kann.

Auf LOBs kann jedoch prinzipiell auch verzichtet werden, wenn man die zu groß gewordene Tabelle vertikal aufteilt und stattdessen eine Sicht definiert, welche die einzelnen Fragmente wieder logisch zu einer vollständigen Tabelle verbindet. Da Sichten in DB2 nur in ganz bestimmten Fällen änderbar sind (Verbunde gehören nicht dazu), bedeutet diese Aufteilung eine Erhöhung des Aufwands bei Update- und Delete-Anweisungen. Auf jede einzelne Tabelle muss separat zugegriffen werden. Daher wurden zum hierarchischen Ansatz als auch zum modifizierten hierarchischen Ansatz jeweils zwei Versionen implementiert, die entweder auf eine einzige Tabelle zugreifen (single) oder auf den Tabellenverbund (multi).

Ganz frei von Limitierungen ist man jedoch auch bei dieser Lösung nicht. In einer Sicht dürfen nur maximal 5000 Attribute[3] angegeben werden. Die Operationen ORDER BY und GROUP BY erlauben zudem nur eine Zeilenlänge von 32677 Bytes [IBM02].

Die Wurzeln bilden jeweils die Klassen `QueryComponentApproach3` bzw. `QueryComponentApproach3b`, welche aus dem EJB `QueryComponentBean` konstruiert wurden. Darin sind die Methoden `query()`, `insert()` und `update_delete()` definiert, welche die bekannten Grundfunktionen des Spatial Model Servers darstellen. Die „single“- und „multi“-Versionen werden nachfolgend jeweils gemeinsam vorgestellt, da sie sich kaum voneinander unterscheiden. Überall dort, wo ein Unterschied besteht, wird explizit darauf hingewiesen.

7.3.1 QueryComponentApproach3

Basisklasse: `QueryComponentBean`
Package: `de.uni_stuttgart.nexus.spaSe.approach3.{single|multi}`

```
public String query(String awqlQuery)
      throws ServerException
```

[2] In DB2 V8.1 beträgt die maximale Länge einer Zeile bei 32 KB Seitengröße 32677 Bytes inkl. Overhead [IBM02].
[3] Bei „SELECT * “ sind sogar nur 1012 Attribute erlaubt, ebenso bei GROUP BY und ORDER BY.

Beschreibung:

Diese Methode wird zum Ausführen von AWQL-Query-Anfragen verwendet.

Vorgehensweise:

Das eingehende AWQL-Dokument `awqlQuery` wird mit Hilfe der Methode `awql2sql()` der Klasse `AWQL2SQLApproach3` in ein einzelnes SQL-Statement umgewandelt. In der Eintabellenversion bezieht sich dieses Statement auf die hierarchische Tabelle, in der Multitabellenversion auf die Sicht über den verschiedenen Fragmenten.

Anschließend wird das Statement auf der DB ausgeführt (Aufruf `query()` der Klasse `DBAccessApproach3`). Das Ergebnis wird in ein Objekt vom Typ `ResultHashMap` eingetragen. Abschließend wird dieses Objekt hergenommen und die darin enthaltenen Objekte zu einem AWML-Dokument umgewandelt (Aufruf `results2AWML()` der Klasse `Results2AWMLApp`). Das Ergebnis dieser Umwandlung wird als Rückgabewert zurückgeliefert.

```
public String update_delete(String awql)
      throws ServerException
```

Beschreibung:

Diese Methode wird zum Ausführen von AWQL-Update/Delete-Anweisungen verwendet.

Vorgehensweise beim Ändern/Löschen von Objekten:

Das Ändern von Nexusobjekten geschieht in mehreren Phasen: Im ersten Schritt wird aus dem eingehenden AWQL-Dokument ein einzelnes (single) bzw. mehrere (multi) SQL-Statements generiert. Dies erledigt die Methode `awql2sql()` der Klasse `AWQL2SQLApproach3`. Die erzeugten SQL-Statements werden mit Hilfe der Methode `change()` der Klasse `DBAccessApproach3` innerhalb einer atomaren Transaktion ausgeführt. Das Ergebnis wird anschließend zunächst in einem Objekt vom Typ `ResultHashMap` gespeichert und schließlich in einem weiteren Schritt zu einem Changed-Result-Dokument umgeformt. Das Ergebnis dieser Umwandlung wird als Resultat dieses Aufrufs zurückgegeben.

7.3.2 AWQL2SQLApproach3

Basisklasse: `AWQL2SQLBean`
Package: `de.uni_stuttgart.nexus.spaSe.approach3.{single|multi}`

Beschreibung:

Diese Klasse ist für die Umformung von AWQL-Dokumenten nach SQL-Statements zuständig. Es werden die Klassen `DBAccessApproach3` und `HierarchyContainer` verwendet.

```
public QueryData awql2SQL(String awql, int mode)
      throws JDOMException, ServerException, SAXException
```

Beschreibung:

Wandelt ein AWQL-Dokument in ein oder mehrere SQL-Statements um.

Vorgehensweise:

Das AWQL-Dokument wird zunächst gegen das Schema validiert. Dazu wird der angegebene Webserver kontaktiert.
Im zweiten Schritt wird das „excludes“-Filter ausgewertet und die darin enthaltenen Attribute in einer `ArrayList` abgelegt. Im nächsten Schritt werden alle „in“-Elemente der AWQL-Restriktion zu Kindern eines neuen „or“-Elements umgewandelt.
Im vierten Schritt werden alle doppelten Negationen entfernt und De Morgan auf der Restriktion angewendet. Dies geschieht mit Hilfe der Methode `traverseStep1()`. Auf dieses Weise sinken die Negationen bis auf die Blattelemente der Restriktion ein.
Im darauf folgenden Schritt werden doppelte ANDs und ORs entfernt. Dies geschieht mit Hilfe der Methode `traverseStep2()`. Diese eben erwähnten Umformungen sind hier nicht zwingend notwendig. Eine Vereinfachung der Restriktion bzw. der WHERE-Klausel würde in diesem Fall im Query-Optimizer der DB stattfinden. Durch einen Schalter kann diese optionale Vereinfachung der Anfrage im SpaSe ein- bzw. ausgeschaltet werden. Dazu dient die boolesche Konstante OPTIMIZEQUERY im Interface `ConstantsApproach3`.

Nur Multitabellen-Version:
Falls es sich um ein Update oder Delete handelt, so wird jetzt zunächst eine temporäre Tabelle namens `SESSION.TEMPTBL` erzeugt, in die später die gefundenen NOLs der Objekte abgelegt werden. Die entsprechende SQL-Anweisung dazu wird in die SQL-Liste eingefügt.
Ein Update/Delete geschieht prinzipiell in zwei Phasen: In Phase 1 werden alle NOLs der Objekte gesammelt, die in Frage kommen. Dazu wird das AWQL-Dokument in einen Queryteil mit AWQL-Restriktion und einen Updateteil ohne Restriktion aufgesplittet. Dies geschieht in der Methode `splitUpdates()`. Im Unterschied zu Ansatz 1 können die NOLs auf Datenbankseite bleiben und müssen nicht zum SpaSe übertragen werden. Mit Hilfe eines neuen Elements namens „temporary“ im AWQL-Dokument kann zwischen diesen beiden Query-Modi unterschieden werden.
In Phase 2 werden die NOLs aus der temporären Tabelle ausgelesen und damit die zu ändernden/löschenden Objekte selektiert. Dies geschieht in mehreren Update-/Delete-Anweisungen (je eine pro Fragment).

Im fünften und letzten Schritt werden schließlich aus den einzelnen AWQL-Dokumenten die entsprechenden SQL-Statements generiert. Dies geschieht mittels des Methodenaufrufs `transform()`. Diese werden zusammen mit den „excludes“-

Attributen vom Anfang der Verarbeitung in ein Objekt vom Typ `QueryData` geschrieben und am Ende als Resultat dieses Methodenaufrufs zurückgeliefert.

In diesem Ansatz wird ein gegebener Objekttyp in der AWQL-Anfrage zunächst expandiert, d.h. es werden sämtliche Subtypen ermittelt. Der EQUAL-Vergleich eines Typs wird zu einem IN-Vergleich mit der Subtypenmenge. Dies stellt den Hauptunterschied zum modifizierten hierarchischen Ansatz (siehe Kapitel 7.4) dar, da dort auf die Expansion verzichtet werden kann. Stattdessen wird ein EQUAL-Vergleich auf dem Attribut „type“ zu einem einfachen LIKE umgeformt.

7.4 Ansatz 3.2: Modifizierter hierarchischer Ansatz

Wie in 6.3 bereits erläutert, existieren zu diesem Ansatz zwei verschiedene Implementierungen. Diese werden nachfolgend mit „single“ bzw. „multi“ gekennzeichnet. Der Hauptunterschied zum vorherigen hierarchischen Ansatz ist die Verarbeitung des „type“-Attributs. Darin wird fortan der vollständige Vererbungspfad eines Objekts abgespeichert.

An den Methoden hat sich im Wesentlichen nichts zum vorherigen Ansatz (siehe 7.3) geändert. All jene Stellen, an denen das Attribut „type“ behandelt wird, wurden entsprechend angepasst. Das betrifft die Klassen `AWQL2SQLApproach3b`, `AWML2SQLApproach3b` und `DBAccessApproach3b`. Um Wiederholungen zu vermeiden, werden hier nur die Änderungen zu den Vorgängern beschrieben.

7.4.1 AWQL2SQLApproach3b

Basisklasse: `AWQL2SQLBean` (bzw. `AWQL2SQLApproach3`)
Package: `de.uni_stuttgart.nexus.spaSe.approach3b.{single|multi}`

Beschreibung:

Diese Klasse ist für die Umformung von AWQL-Dokumenten nach SQL-Statements zuständig. Es werden die Klassen `DBAccessApproach3b` und `HierarchyContainer` verwendet.

Änderungen:

Bei der Suche nach Objekten eines bestimmten Objekttyps wird nicht die Subtypenmenge bestimmt, sondern das Attribut wird daraufhin verglichen, ob der entsprechende Objekttyp-Eintrag darin enthalten ist (Bedingung: `type LIKE '%#`*`ObjectType`*`#%'`).

7.4.2 DBAccessApproach3b

Basisklasse: `DBAccessBean` (bzw. `DBAccessApproach3`)
Package: `de.uni_stuttgart.nexus.spaSe.approach3b.{single|multi}`

Beschreibung:

Diese Klasse ist für die Abarbeitung der SQL-Statements und die Erzeugung der Ergebnismenge als `ResultHashMap` zuständig. Außerdem bietet sie Zugriff auf die Metadatentabelle `SYSCAT.COLUMNS` der DB, zum Auslesen der Attributtypen der Objekttyptabellen.

Änderungen:

Die Methode `addResultSet2HashMap()` wurde dahingehend geändert, dass nunmehr nicht der komplette Inhalt des Attributs „type“ in das Ergebnis mit einfließt, sondern lediglich der erste Eintrag. Darin ist der eigentliche Objekttyp des jeweiligen Objekts abgespeichert.

7.5 Ansatz 4: Attributtabellenansatz

7.5.1 QueryComponentApproach4

Basisklasse: `QueryComponentBean`
Package: `de.uni_stuttgart.nexus.spaSe.approach4`

```
public String update_delete(String awql)
      throws ServerException
```

Beschreibung:

Diese Methode kann zum Ausführen von AWQL-Update/Delete-Anweisungen verwendet werden.

Vorgehensweise beim Ändern/Löschen von Objekten:

Das Ändern von Nexusobjekten geschieht in mehreren Phasen: Im ersten Schritt werden aus dem eingehenden AWQL-Dokument `awql` mehrere SQL-Statements generiert. Dies erledigt die Methode `awql2sql()` der Klasse `AWQL2SQLApproach4`. Die erzeugten SQL-Statements werden mit Hilfe der Methode `change()` der Klasse `DBAccessApproach4` innerhalb einer atomaren Transaktion ausgeführt. Das Ergebnis wird anschließend zunächst in einem Objekt vom Typ `ResultHashMap` gespeichert und schließlich in einem weiteren Schritt zu einem Changed-Result-Dokument umgeformt. Dazu wird die Methode `result2CRL()` der Klasse `Result2CRLApproach4` verwendet. Dieses CRL-Dokument wird als Resultat zurückgegeben.

7.5.2 AWQL2SQLApproach4

Basisklasse: `AWQL2SQLBean`
Package: `de.uni_stuttgart.nexus.spaSe.approach4`

Beschreibung:

Diese Klasse ist für die Umformung von AWQL-Dokumenten nach SQL-Statements zuständig. Es werden die Klassen `DBAccessApproach4` und `Hierarchy-Container` verwendet.

```
public QueryData awql2SQL(String awql, int mode)
      throws JDOMException, ServerException, SAXException
```

Beschreibung:

Wandelt ein AWQL-Dokument in SQL-Statements um.

Vorgehensweise:

Das AWQL-Dokument wird zunächst gegen das Schema validiert. Dazu wird der angegebene Webserver kontaktiert. Im zweiten Schritt wird das „excludes"-Filter ausgewertet und die darin enthaltenen Attribute in einer `ArrayList` abgelegt.

Im nächsten Schritt werden alle „in"-Elemente der AWQL-Restriktion zu Kindern eines neuen „or"-Elements umgewandelt.

Im vierten Schritt werden alle doppelten Negationen entfernt und De Morgan auf der Restriktion angewendet. Dies geschieht mittels des Methodenaufrufs `traverseStep1()`. Auf dieses Weise sinken die Negationen bis auf die Blattelemente der Restriktion ein. Im darauf folgenden Schritt werden doppelte ANDs und ORs entfernt. Dies geschieht mit Hilfe der Methode `traverseStep2()`.

Diese beiden Umformungsschritte sind zwar nicht zwingend notwendig jedoch empfehlenswert, da boolesche Ausdrücke zu verschachtelten Mengenoperationen umgeformt werden. Im Falle eines NOTs wird z.B. der Operator EXCEPT auf der Menge aller Nexusobjekte angewendet. Je weniger Mengen letztlich berechnet werden müssen, desto einfacher und schneller ist die Anfragebearbeitung in der DB. Über den Schalter OPTIMIZEQUERY im Interface `ConstantsApproach4` kann diese Option ein- bzw. abgeschaltet werden.

Falls es sich um ein Update handelt, so wird jetzt zunächst eine temporäre Tabelle namens `SESSION.TEMPTBL` erzeugt, in die später die gefundenen NOLs der Objekte abgelegt werden. Die entsprechende SQL-Anweisung, die dies für uns erledigen soll, wird direkt in die SQL-Liste eingefügt. Ein Update geschieht prinzipiell in zwei Phasen: In Phase 1 werden alle NOLs der Objekte gesammelt, die in Frage kommen. Dazu wird das AWQL-Dokument in einen Queryteil mit Restriktion und einen Updateteil aufgesplittet. Dies geschieht in der Methode `splitUpdates()`. Im Unterschied zu Ansatz 1 können die NOLs auf Datenbankseite bleiben und müssen nicht zum

SpaSe übertragen werden. Mit Hilfe eines neuen Elements namens „temporary“ im AWQL-Queryteil kann zwischen diesen beiden Query-Modi unterschieden werden.

In Phase 2 werden die NOLs aus der temporären Tabelle ausgelesen und damit die zu ändernden Objekte bestimmt. Für jedes zu ändernde Attribut wird ein eigenes AWQL-Dokument erzeugt, welches ausschließlich ein Update-Element enthält.

Im fünften und letzten Schritt werden schließlich aus den erzeugten AWQL-Dokumenten die entsprechenden SQL-Statements generiert. Dies geschieht mittels des Methodenaufrufs `transform()`. Diese werden zusammen mit den „excludes“-Attributen vom Anfang der Verarbeitung in ein Objekt vom Typ `QueryData` abgelegt und als Ergebnis dieses Methodenaufrufs zurückgeliefert.

```
private ArrayList splitUpdates(Document awqlDoc)
```

Beschreibung:

Diese Methode erzeugt aus einer AWQL-Update-Anfrage ein AWQL-Query-Dokument ohne Update-Element jedoch mit speziellem „temporary“-Element und je ein weiteres Dokument für die zu ändernden Attribute.
In letzteren sind außer einem Update-Element keine weiteren Elemente enthalten. Ein Update läuft in zwei Phasen ab: In der ersten Phase werden alle NOLs ermittelt, die für ein Update in Frage kommen und in eine temporäre Tabelle gespeichert. In der zweiten Phase werden die gefundenen Objekte geändert. Es wird dazu die temporäre Tabelle `SESSION.TEMPTBL` angelegt, beschrieben und wieder entfernt.

7.6 Gemeinsame Klassen

Die implementierten Ansätze teilen sich einige Klassen. Darunter fallen wie eingangs erwähnt die Klassen `ResultHashMap`, `QueryData`, `AttributesNameMapper`, `ResultColumn`, `ResultRow` und `SpaSeErrorHandler`. Diese wurden unverändert aus der bestehenden SpaSe-Implementierung übernommen und werden daher nicht weiter beschrieben.
Hinzu gekommen sind die Klassen `Result2CRLApp`, `Results2AWMLApp` und `HierarchyContainer`, die aus den EJBs `Result2CRLBean`, `Results2AWMLBean` und `HierarchyContainerBean` gebildet wurden. Eine vollständig neue Klasse stellt `AWMLDocument` dar. Da die beiden Klassen `Result2CRLApp` und `Results2AWMLApp` bis auf die fehlende Bean-Eigenschaft identisch mit ihren Vorgängern sind, werden sie nachfolgend ebenfalls nicht weiter erläutert.

7.6.1 HierarchyContainer

Basisklasse: `HierarchyContainerBean`
Package: `de.uni_stuttgart.nexus.spaSe.common`

Beschreibung:

Diese Klasse ermöglicht den einfachen Zugriff auf die Vererbungsinformationen der Nexusobjekte. Mit ihrer Hilfe lässt sich an zentraler Stelle die Super-/Subtypen-Hierarchie der Objekte aus der Datenbank einlesen, berechnen und für die weitere Verwendung zwischenspeichern. Da alle Ansätze dieselbe Vererbungshierarchie verwenden, kann diese Klasse gemeinsam genutzt werden. Es muss lediglich bei der Instanziierung angegeben werden, welches Datenbankschema verwendet werden soll.

```
public TreeMap getTableHierarchy()
```

Beschreibung:

Diese Methode liefert die Objekttyphierarchie zu den gespeicherten Objekten in der Datenbank zurück. Alle Angaben verwenden Groß-/Kleinschreibung. Dazu wird die `TypeHierarchy`-Tabelle ausgelesen und intern in einer `TreeMap` weiterverarbeitet. Nur der erste Aufruf dieser Methode führt zum tatsächlichen DB-Aufruf. Alle weiteren liefern die gespeicherte `TreeMap` zurück.

```
public TreeMap getSupertableHierarchy()
```

Beschreibung:

Diese Methode liefert die Supertyphierarchie zu den gespeicherten Objekten in der Datenbank zurück. Alle Angaben verwenden ebenfalls Groß-/Kleinschreibung (s.o.). Dazu wird die `TypeHierarchy`-Tabelle ausgelesen und intern in einer `TreeMap` weiterverarbeitet. Nur der erste Aufruf dieser führt zum tatsächlichen DB-Aufruf. Alle weiteren liefern die gespeicherte `TreeMap` zurück.

7.6.2 AWMLDocument

Package: `de.uni_stuttgart.nexus.spaSe.common`

Beschreibung:

Diese Klasse ermöglicht die einfache Verarbeitung von AWML-Dokumenten. Über spezielle Methoden können die darin enthaltenen Nexusobjekt-Beschreibungen objektweise (vom Typ `org.jdom.Element`) ausgelesen werden.

Methodenübersicht:

- `getLength()`
  Gibt die Anzahl der gefundenen Nexusobjekte im AWML-Dokument zurück.
- `getElementEntity(int)`
  Gibt das NexusObjekt an der angegebenen Stelle zurück.
- `hasNext()`
  Prüft, ob weitere Nexusobjekte vorhanden sind.
- `getNextElement()`
  Gibt das nächste NexusObjekt in der Liste zurück.
- `getPosition()`
  Gibt die aktuelle Position in der Liste zurück.
- `isEmpty()`
  Prüft, ob das übergebene AWML-Dokument leer ist.

8 Praktische Bewertung

Es folgt die praktische Bewertung der Implementierungen. Dazu wurde ein Testrahmenprogramm erstellt mit dessen Hilfe die sequentielle Ausführung von AWQL-Statements möglich ist. Für die Durchführung des Tests wurden die einzelnen Testfälle in einer gemeinsamen Textdatei gespeichert und nacheinander abgearbeitet. Die Ausführungszeiten sowie die generierten Ergebnisse wurden jeweils notiert.

Zunächst folgt in Kapitel 8.1 die Vorstellung des Testrahmens. Anschließend wird in Kapitel 8.2 der Versuchsaufbau beschrieben. In Kapitel 8.3 werden die erstellten Testfälle vorgestellt. Die Ergebnisse des praktischen Performancetests sind in Kapitel 8.4 zu finden. Ein abschließender Vergleich der Ergebnisse untereinander sowie eine Gegenüberstellung mit den theoretischen Bewertungen (vgl. Kapitel 6) folgt in Kapitel 8.5 bzw. 8.6.

8.1 Testrahmen

Package: `de.uni_stuttgart.nexus.spaSe.tools`

Der Testrahmen wird in mehreren Klassen implementiert. Die Wurzel mit der `Main`-Methode bildet die Klasse `TestFramework`. Des Weiteren existiert eine Helferklasse `TestQueries`, die das Einlesen und Bereitstellen der verschiedenen Testfälle in einem einzelnen Objekt kapselt. Dieses liefert AWML- bzw. AWQL-Statements in Form von Objekten des Typs `QueryEntity` an den Aufrufer zurück. Darin ist Platz für die Anfrage selber sowie für die Antwort des Servers. Neben der Ausführungszeit kann darin auch der jeweilige Anfragetyp (insert, update/delete oder query) angegeben werden. Auf diese Weise kann am Ende des Testlaufs jeder einzelne Testfall bequem ausgewertet werden.

8.1.1 Testframework

```
public static void main(String[] args)
```

Beschreibung:

Diese Methode ist der Einstiegspunkt für die Ausführen des Testrahmens. Es werden zwei Parameter erwartet, die Nummer des Ansatzes (1, 2, 3, 3b, 3m, 3bm, 4) und der Name der Testdatei mit den AWML-/AWQL-Statements.

Vorgehensweise:

Die im zweiten Parameter angegebene Textdatei wird mit Hilfe der Klasse `TestQueries` verarbeitet. Diese stellt Methoden zum sequentiellen Auslesen der einzelnen Statements bereit. Der ausgewählte SpaSe wird im ersten Schritt initiali-

siert. Die dazu benötigte Zeit wird gemessen. Je nach Typ des Statements wird eine der Methoden `query()`, `insert()` oder `update_delete()` des gewählten SpaSes aufgerufen. Die jeweilige Ausführungszeit wird ebenfalls gemessen und zusammen mit dem Ergebnis abgespeichert. Am Ende des Testlaufs folgt eine Zusammenfassung in der Form: Anfrage, Ergebnis, benötigte Zeit. Sämtliche Ausgaben werden sowohl auf dem Terminal als auch in eine Log-Datei geschrieben.

8.1.2 TestQueries

Package: `de.uni_stuttgart.nexus.spaSe.tools`

Beschreibung:

Diese Klasse kapselt die Testfälle in einem Java-Objekt. Über den Konstruktor wird der Name der Textdatei angegeben. Mit Hilfe von Getter- und Setter-Methoden kann nach erfolgreicher Instanziierung auf die einzelnen Testfälle zugegriffen werden.

8.1.3 QueryEntity

Package: `de.uni_stuttgart.nexus.spaSe.tools`

Beschreibung:

Die Klasse `QueryEntity` kapselt ein beliebiges AWML-/AWQL-Statement zusammen mit dem Resultat und der benötigten Ausführungszeit in einem Java-Objekt. Es werden verschiedene Getter- und Setter-Methoden zum Auslesen bzw. Beschreiben der einzelnen Daten zur Verfügung gestellt.

Methodenübersicht (public):

- `setTime(long)`
  Schreibt die Ausführungszeit (in ms) der Anfrage in das Objekt.

- `setQuery(String)`
  Schreibt die AWQL-/AWML-Anfrage in das Objekt.

- `setResult(String)`
  Schreibt das Resultat der Anfrage in das Objekt.

- `setQueryType(String)`
  Setzt den Anfragetyp fest.

- `getTime()`
  Gibt die gesetzte Ausführungszeit zurück.

- `getQuery()`
  Gibt die gesetzte Anfrage zurück.

- `getResult()`
  Gibt das ermittelte Ergebnis der Anfrage zurück.

- `getQueryType()`
  Gibt den gesetzten Anfragetyp zurück.

8.2 Versuchsaufbau

Als Testrechner wurde ein AMD© Athlon™ XP 2200+ mit 1.8 GHz Taktfrequenz, 1 GB Hauptspeicher und Windows XP Professional™ als Betriebssystem verwendet (nexuspc5). Auf diesem Rechner wurde die Testsoftware ausgeführt, welche über JDBC auf die entfernte Datenbank zugreifen konnte. Als Datenbankserver wurde ein 4-Prozessorsystem (Intel®) mit je 700 MHz, Windows 2000 und 4 GB Hauptspeicher sowie SCSI-Raid-Festplattensystem verwendet, auf dem IBM® DB2 UDB™ V8.1 Enterprise Server Edition mit dem Spatial Extender Zusatz installiert wurde (pcquaddmg). Beide Rechner sind über das hausinterne Netzwerk (100 Mbit) miteinander verbunden. Für die Durchführung des Tests wurde eine eigene Datenbank eingerichtet und mit Testdaten gefüllt. Diese stammten größtenteils aus der alten Spatial Model Server Datenbank und wurden durch einige weitere Nexusobjekte ergänzt.

8.3 Beschreibung der Testfälle

Es wurden insgesamt 21 Testfälle generiert, die verschiedene Implementierungsaspekte auswerten sollen. In nachfolgender Gegenüberstellung wurden die einzelnen Fälle klassifiziert. In der Spalte „Typ“ steht der jeweilige Anfragetyp, der mit den diskutierten Anfragen in Kapitel 6 korrespondiert. Zu jedem Testfall wurden zudem der Umfang des erwarteten Ergebnisses, die Tiefe der AWQL-Restriktion sowie einige Besonderheiten aufgelistet.

Testfall	Typ	Anz. Attribute in Restriktion	Restriktionstiefe	Anz. Objekte im Ergebnis	Besonderheit
0	Type	1	1	3	keine
1	o. Type	1	1	125	Spatial query mit INSIDE und Includes-Filter
2	Type	2	2	2	Spatial query mit INSIDE
3	OID	1	1	1	keine
4	o. Type	3	3	1	Spatial query mit OVERLAPS, KNF
5	o. Type	1	1	1	Spatial query mit EQUAL
6	Type	1	1	10	Closest 10, Includes-Filter, Objekttyp ist Wurzel
7	o. Type	1	1	2	Suche nach Name
8	o. Type	4	2	3	Suche mit LIKE
9	Type	4	4	3	Komplexe Typsuche, nutzt

					Mehrfachvererbung
10	Type	3	2	0	Spatial Query nach Straßen
11	Type	3	2	1	Spatial Query nach Emitter
12	Type	2	2	1	Spatial Query nach Räumen
13	OID (upd.)	1	1	1	Update mit drei Attributänderungen
14	Type (upd.)	2	2	150	Update von vielen Objekten
15	o. Type	6	7	2170	Komplexe Query mit doppeltem NOT-Element und AND-Element
16	o. Type	6	3	0	KNF-Form
17	Type	4	3	150	Spatial Query über extent-Attribut, Suchgebiet mit Loch
18	o. Type	8	2	5	Viele EQUAL-Elemente
19	o. Type	12	3	6	Sehr komplexe Query mit Closest-Element
20	Type	8	3	18	Komplexe Spatial Query mit “type not ...“-Bedingung

Tabelle 8-1) Merkmale der Testfälle

8.4 Testergebnisse

Der Test wurde folgendermaßen durchgeführt:
Für jeden Ansatz wurden die Testfälle bis zu fünf Mal hintereinander durchlaufen und die Ausführungszeiten für jede Anfrage notiert. Für jeden der insgesamt 21 Testfälle wurde der Mittelwert aus den Testläufen bestimmt und in eine Tabelle eingetragen. In den nachfolgenden Unterkapiteln folgt die Zusammenstellung der einzelnen Testergebnisse.

8.4.1 Ansatz 1: Vertikale Partitionierung

Bei diesem Ansatz wurde der Test wegen der schlechten Performance nur einmal durchlaufen. Das Ergebnis der Zeitmessung findet sich in nachfolgender Tabelle.

Testfall	Ausführungszeit
init	2766 ms
0	12031 ms
1	697328 ms
2	47703 ms
3	20578 ms
4	756579 ms
5	867312 ms
6	291953 ms
7	178125 ms
8	174406 ms
9	10250 ms

10	2672 ms
11	1750 ms
12	4657 ms
13	24546 ms
14	891 ms
15	701828 ms
16	215125 ms
17	543031 ms
18	184344 ms
19	616188 ms
20	669109 ms

Tabelle 8-2) Endresultat von Ansatz 1

8.4.2 Ansatz 2: Modifizierte horizontale Partitionierung

Es wurden insgesamt 3 Testläufe durchgeführt. Von den gemessenen Zeiten wurde das arithmetische Mittel berechnet und als Testergebnis für diesen Ansatz in die nachfolgende Tabelle eingetragen.

Testfall	Mittlere Ausführungszeit
init	2037 ms
0	1963 ms
1	17854 ms
2	792 ms
3	9849 ms
4	15313 ms
5	15214 ms
6	27427 ms
7	7182 ms
8	8219 ms
9	713 ms
10	365 ms
11	411 ms
12	380 ms
13	1287 ms
14	651 ms
15	23078 ms
16	78427 ms
17	8010 ms
18	11594 ms
19	114792 ms
20	127776 ms

Tabelle 8-3) Endresultat von Ansatz 2

8.4.3 Ansatz 3.1: Hierarchischer Ansatz (single)

Zu diesem Ansatz wurden insgesamt 5 Testläufe durchgeführt und jeweils die Mittelwerte in die nachfolgende Tabelle eingetragen.

Testfall	Mittlere Ausführungszeit
init	844 ms
0	1625 ms
1	2050 ms
2	2910 ms
3	312 ms
4	735 ms
5	1309 ms
6	10328 ms
7	410 ms
8	281 ms
9	228 ms
10	712 ms
11	897 ms
12	725 ms
13	306 ms
14	225 ms
15	11490 ms
16	2128 ms
17	10594 ms
18	331 ms
19	3466 ms
20	3087 ms

Tabelle 8-4) Endresultat von Ansatz 3.1 (single)

Tests mit ausgeschalteter Optimierung (Schalter OPTIMIZEQUERY siehe 7.3.2) ergaben im Mittel praktisch keine Veränderung an der Performance.

8.4.4 Ansatz 3.1: Hierarchischer Ansatz (multi)

Zu diesem Ansatz wurden ebenfalls 5 Testläufe durchgeführt. Bei der Durchführung des Tests kam es bei zwei Testfällen zu Datenbankfehlern, so dass kein vollständiges Resultat ermittelt werden konnte. Diese Fälle wurden speziell markiert und mit der entsprechenden Fehlermeldung dokumentiert.

Testfall	Mittlere Ausführungszeit
init	1003 ms
0	1862 ms
1	(*) 12006 ms
2	2116 ms
3	250 ms
4	1978 ms
5	2365 ms
6	10385 ms
7	556 ms
8	563 ms
9	475 ms

10	1894 ms
11	2213 ms
12	1994 ms
13	278 ms
14	400 ms
15	12959 ms
16	1935 ms
17	10669 ms
18	1068 ms
19	(**) 14028 ms
20	4841 ms

Tabelle 8-5) Endresultat von Ansatz 3.1 (multi)

(*)
Fehlermeldung: *SQL0901N The SQL statement failed because of a non-severe system error. Subsequent SQL statements can be processed. (Reason "sqlya_free_adt_MBs: MB must point back to ADT passed in".) SQLSTATE=58004*
Dieser Systemfehler ist reproduzierbar. Es handelt sich hierbei um einen bereits aus der Version 7 bekannten Fehler. Ein entsprechender Patch für Version 8 lag zum Zeitpunkt der Entwicklung nicht vor.
(**)
Fehlermeldung: *SQL1585N A system temporary table space with sufficient page size does not exist. SQLSTATE=54048*
Dieser Fehler ist reproduzierbar. Die CLOSEST-Funktion wird in SQL mittels ORDER BY realisiert. Die maximale Zeilenbreite ist hier auf die Seitengröße (32 KB) beschränkt. Da die verbundenen Tupel der hierarchischen Tabellensicht jedoch nicht auf eine Seite passen, führt dieser Aufruf zwangsläufig zu einem Fehler.

Tests mit ausgeschalteter Optimierung (Schalter OPTIMIZEQUERY siehe 7.3.2) ergaben eine geringfügig kürzere Gesamtausführungszeit (< 1 Prozent). Im Mittel ist jedoch keine Veränderung an der Performance bemerkbar.

8.4.5 Ansatz 3.2: Modifizierter hierarchischer Ansatz (single)

Zu diesem Ansatz wurden insgesamt 5 Testläufe durchgeführt. Das Ergebnis der Zeitmessung findet sich in nachfolgender Tabelle.

Testfall	Mittlere Ausführungszeit
init	535 ms
0	1450 ms
1	2013 ms
2	825 ms
3	244 ms
4	728 ms
5	1384 ms
6	10725 ms

7	337 ms
8	281 ms
9	253 ms
10	728 ms
11	866 ms
12	716 ms
13	312 ms
14	191 ms
15	13091 ms
16	2103 ms
17	4697 ms
18	381 ms
19	3478 ms
20	3106 ms

Tabelle 8-6) Endresultat von Ansatz 3.2 (single)

Tests mit ausgeschalteter Optimierung (Schalter OPTIMIZEQUERY siehe 7.3.2) ergaben im Mittel praktisch keine Veränderung an der Performance.

8.4.6 Ansatz 3.2: Modifizierter hierarchischer Ansatz (multi)

Bei diesem Ansatz wurden ebenfalls 5 Testläufe durchgeführt. Bei der Durchführung des Tests kam es wie zuvor bei Ansatz 3.1 bei zwei Testfällen zu Datenbankfehlern, so dass kein vollständiges Resultat ermittelt werden konnte. Diese Fälle wurden entsprechend markiert (vgl. 8.4.4).

Testfall	Mittlere Ausführungszeit
init	957 ms
0	1609 ms
1	(*) 12622 ms
2	2103 ms
3	253 ms
4	1966 ms
5	2525 ms
6	10369 ms
7	562 ms
8	547 ms
9	500 ms
10	1887 ms
11	2200 ms
12	1991 ms
13	228 ms
14	466 ms
15	13091 ms
16	1953 ms
17	8519 ms

18	859 ms
19	(**) 14082 ms
20	4850 ms

Tabelle 8-7) Endresultat von Ansatz 3.2 (multi)

Tests mit ausgeschalteter Optimierung (Schalter OPTIMIZEQUERY siehe 7.3.2) ergaben eine geringfügig kürzere Gesamtausführungszeit (< 1 Prozent). Im Mittel ist jedoch keine Veränderung an der Performance bemerkbar.

8.4.7 Ansatz 4: Attributtabellenansatz

Es wurden insgesamt 5 Testläufe durchgeführt. Die Mittelwerte wurden in die untenstehende Tabelle eingetragen.

Testfall	Mittlere Ausführungszeit
init	1034 ms
0	1781 ms
1	3378 ms
2	1075 ms
3	362 ms
4	669 ms
5	544 ms
6	4947 ms
7	812 ms
8	310 ms
9	606 ms
10	575 ms
11	694 ms
12	447 ms
13	216 ms
14	3219 ms
15	20150 ms
16	1256 ms
17	2469 ms
18	416 ms
19	2994 ms
20	3400 ms

Tabelle 8-8) Endresultat von Ansatz 4

Tests mit ausgeschalteter Optimierung (Schalter OPTIMIZEQUERY siehe 7.5.2) ergaben, praktisch keine Veränderung an der Performance. Im Mittel lag die Verbesserung bei eingeschalteter Optimierung bei unter 1 Prozent.

8.5 Vergleich der Ergebnisse

Es folgt der tabellarische Vergleich der durchschnittlichen Laufzeiten der einzelnen Ansätze. Die minimalen Zeiten pro Testfall wurden unterstrichen. Je mehr Erstplatzierungen ein Ansatz erzielte, desto besser war dieser in der Endauswertung.

Testfall	Ansatz 1	Ansatz 2	Ansatz 3.1 (single)	Ansatz 3.2 (single)	Ansatz 3.1 (multi)	Ansatz 3.2 (multi)	Ansatz 4
init	2766 ms	2037 ms	844 ms	535 ms	1003 ms	957 ms	1034 ms
0	12031 ms	1963 ms	1625 ms	1450 ms	1862 ms	1609 ms	1781 ms
1	697328 ms	17854 ms	2050 ms	2013 ms	(*) 12006 ms	(*) 12622 ms	3378 ms
2	47703 ms	792 ms	2910 ms	825 ms	2116 ms	2103 ms	1075 ms
3	20578 ms	9849 ms	312 ms	244 ms	250 ms	253 ms	362 ms
4	756579 ms	15313 ms	735 ms	728 ms	1978 ms	1966 ms	669 ms
5	867312 ms	15214 ms	1309 ms	1384 ms	2365 ms	2525 ms	544 ms
6	291953 ms	27427 ms	10328 ms	10725 ms	10385 ms	10369 ms	4947 ms
7	178125 ms	7182 ms	410 ms	337 ms	556 ms	562 ms	812 ms
8	174406 ms	8219 ms	281 ms	281 ms	563 ms	547 ms	310 ms
9	10250 ms	713 ms	228 ms	253 ms	475 ms	500 ms	606 ms
10	2672 ms	365 ms	712 ms	728 ms	1894 ms	1887 ms	575 ms
11	1750 ms	411 ms	897 ms	866 ms	2213 ms	2200 ms	694 ms
12	4657 ms	380 ms	725 ms	716 ms	1994 ms	1991 ms	447 ms
13	24546 ms	1287 ms	306 ms	312 ms	278 ms	228 ms	216 ms
14	891 ms	651 ms	225 ms	191 ms	400 ms	466 ms	3219 ms
15	701828 ms	23078 ms	11490 ms	13091 ms	12959 ms	13091 ms	20150 ms
16	215125 ms	78427 ms	2128 ms	2103 ms	1935 ms	1953 ms	1256 ms
17	543031 ms	8010 ms	10594 ms	4697 ms	10669 ms	8519 ms	2469 ms
18	184344 ms	11594 ms	331 ms	381 ms	1068 ms	859 ms	416 ms
19	616188 ms	114792 ms	3466 ms	3478 ms	(**) 14028 ms	(**) 14082 ms	2994 ms
20	669109 ms	127776 ms	3087 ms	3106 ms	4841 ms	4850 ms	3400 ms

Tabelle 8-9) Gegenüberstellung der Resultate

Um einen anschaulicheren Vergleich der einzelnen Ansätze zu bekommen, wurden obenstehende Resultate in Diagramme umgesetzt. Aufgrund der großen Variation der Laufzeiten wurde eine logarithmische Skala für die Ausführungszeiten gewählt.

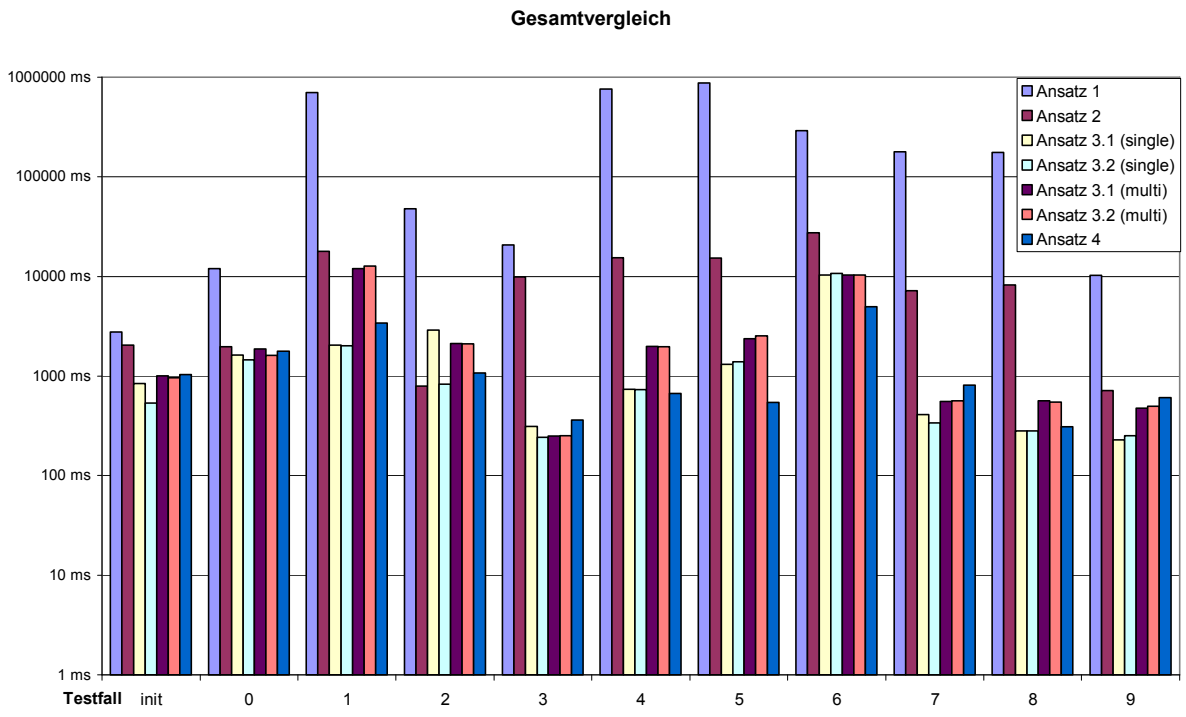

Diagramm 8-1) Ergebnisse der Testfälle 0 bis 9

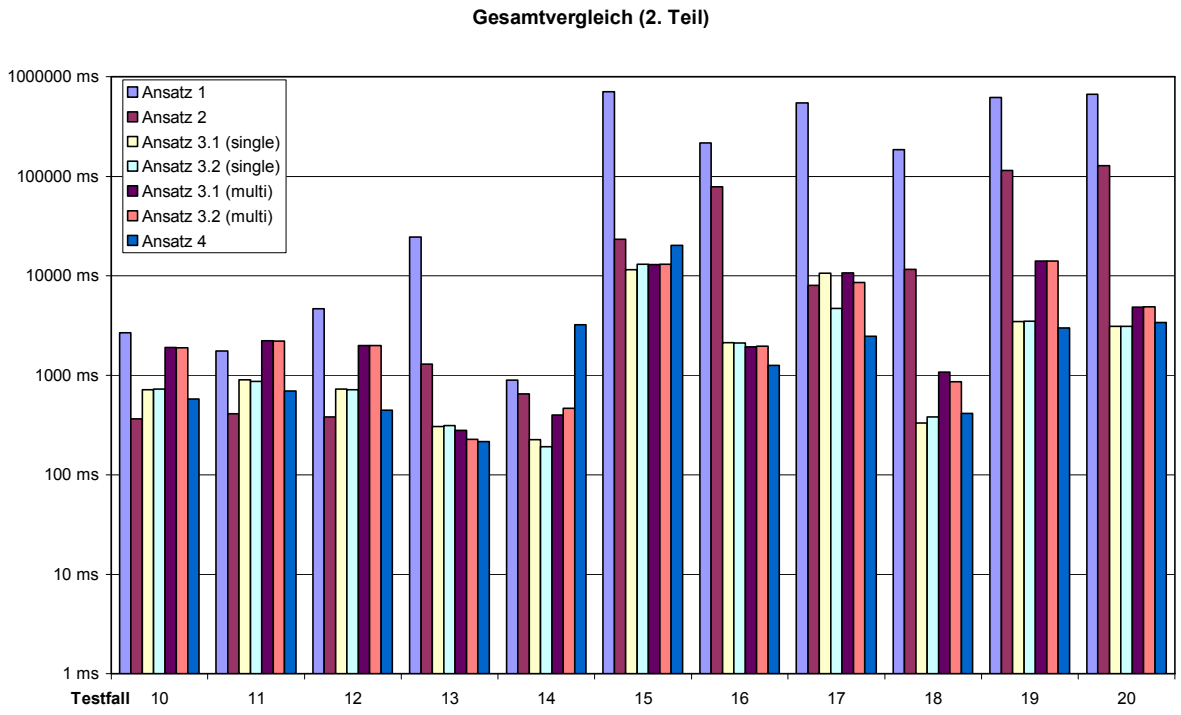

Diagramm 8-2) Ergebnisse der Testfälle 10 bis 20

Im nachfolgenden Diagramm werden alle Ansätze mit ihren prozentualen Anteilen an der Gesamtlaufzeit pro Testfall gegenübergestellt. Je kleiner der Balken für einen Ansatz ausfällt, desto besser ist die Performance.

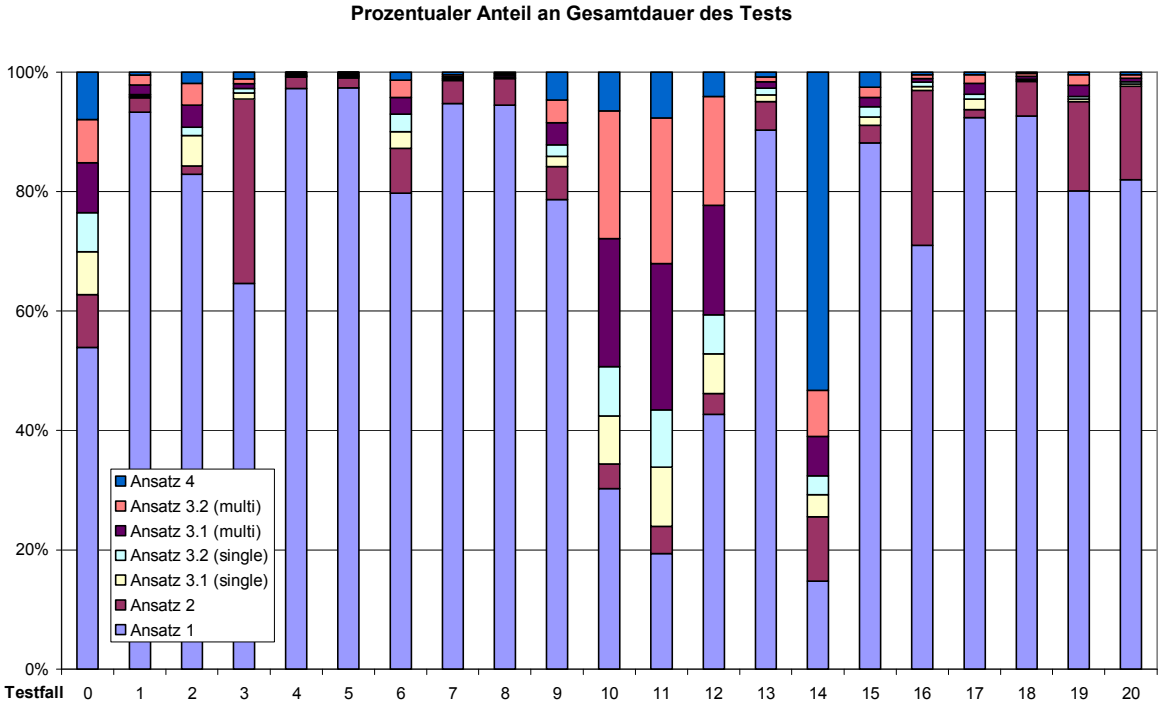

Diagramm 8-3) Vergleich der Ergebnisse

8.6 Vergleich der theoretischen und praktischen Bewertung

In diesem Unterkapitel wird ein Vergleich der theoretischen mit der praktischen Bewertung angestellt.
Sieht man sich die Laufzeiten für eine einfache Objekttyp-Anfrage (Testfall 0) an, so erkennt man, dass alle getesteten Ansätze außer dem vertikalen Ansatz sehr nahe beieinander liegen. Letzterer benötigte für diese Art Anfrage unerwartet viel Zeit, was auf einen beträchtlich hohen konstanten Aufwand für Tabellenverbunde hinweist. Die geschätzte Komplexität von O(N) bzw. O(N log N) für diese Anfrageart hat sich aber für die übrigen Ansätze bewahrheitet.

Die Suchanfragen nach Objekten anhand der Objekt-ID (z.B. Testfall 3) zeigen ähnliches Bild. Hier war die geschätzte Komplexität O(log N). Die gemessenen Werte unterstreichen dies eindrucksvoll. Lediglich der vertikale und der horizontale Ansatz bringen hier eine unerwartet hohe Laufzeit zu Tage. Hier spielt offensichtlich die große Zahl an Datenbankanfrage eine entscheidende Rolle. Wie auch in den übrigen Testfällen zu beobachten ist, ist der konstante Faktor pro Tabellenanfrage sehr hoch. Es gilt daher, möglichst wenige Teilanfragen zu erzeugen. Der Großteil der Verarbeitungszeit wird nicht für die Umformung der AWQL-Anfrage in DNF-Darstellung

benötigt, sondern für die daraus resultierenden Teilanfragen an die Datenbank. Da der vertikale Ansatz neben den Teilanfragen auch zahlreiche Tabellenverbunde durchführen muss, schnellt die Laufzeit hier noch viel drastischer nach Oben.

Anfragen ohne die Angabe eines Objekttyps (z.B. Testfälle 4, 5, 7 und 8) zeigen, dass die geschätzte lineare Komplexität der Ansätze durchaus zutrifft. Bei den Ansätzen mit der aufgeteilten hierarchischen Tabelle (Ansätze 3.1 (multi) und 3.2 (multi)) sieht man außerdem, dass die sich die Laufzeit bei den hier eingesetzten zwei Tabellen mehr als verdoppelt hat. Dies ist auf den Tablescan über den Verbund der beiden Teiltabellen zurückzuführen. Die höher geschätzte Komplexität für den Attributtabellenansatz von O(N log N) für diese Art von Anfragen bewahrheitet sich erst bei aufwendigen Anfragen (z.B. Testfall 15), die sehr viele Objekte zurückliefern. Bei Standardanfragen (z.B. Testfall 12) ist dieser Ansatz jedoch schneller, da Indexe (z.B. über den räumlichen Attributen) besser ausgenutzt werden können.

Der praktische Test deckte außerdem eine noch nicht diskutierte Schwäche des Attributtabellenansatzes auf: Updates von sehr vielen Objekten. Der Testfall 14 zeigt dies eindrucksvoll. Die gut 16-fach höhere Laufzeit gegenüber dem Besten in diesem Feld ist dadurch zu begründen, dass ein Update, bedingt durch die zeilenweise Speicherung der Attribute, in zwei Phasen durchgeführt werden muss (siehe 7.5.2). Die in der ersten Phase gefundenen Objekte werden in der zweiten Phase geändert. Die meiste Zeit benötigt dabei die erste Phase (Suche). Ein Blick auf den Ausführungsplan zeigte weiter, dass in der zweiten Phase die temporär erzeugte Tabelle mit den gefundenen Objekt-IDs mit der Attributtabelle verbunden wird (Nested-Loop-Join). Dieser Verbund benötigt weitere Verarbeitungszeit ehe tatsächlich ein Attribut eines Objekts geändert werden kann.

8.6.1 Fazit

Im Vergleich schneidet der Ansatz 3.2 (single) mit 6 Erstplatzierungen und der insgesamt kürzesten Bearbeitungszeit von 48,4 Sekunden am besten ab, gefolgt von Ansatz 4 mit 7 Erstplatzierungen und einer Zeit von 51,3 Sekunden. Die Ansätze 3.1 und 3.2 in der jeweiligen Multitabellenversion teilen sich den vierten Platz mit ähnlich guten Ergebnissen, wobei diese aber die Testfälle 1 und 19 wegen eines hervorgerufenen Systemfehlers nicht beenden konnten. Das nachfolgende Diagramm 8-4 gibt einen Überblick über die erzielten Bearbeitungszeiten.

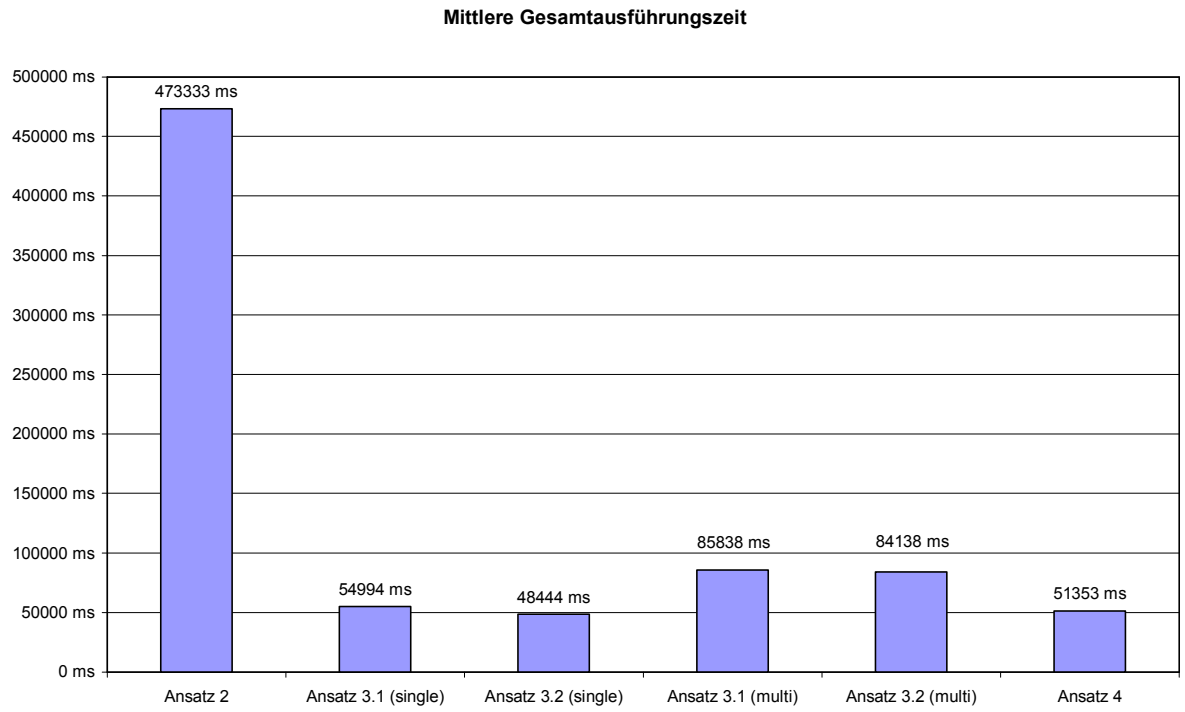

Diagramm 8-4) Vergleich der gemessenen Zeiten

Den letzten Platz belegt der Ansatz 2 mit rund 8 Minuten, wenn man den Ansatz 1 mit stolzen 100 Minuten außeracht lässt. Der modifizierte horizontale Ansatz ist von der Leistung her am ehesten vergleichbar mit der alten Implementierung des Spatial Model Servers, wobei er in den meisten Fällen sogar schneller ist.

Ansatz 1 enttäuscht auf ganzer Linie, was vor allem auf die zahlreichen Verbunde durch die Tabellensichten zurückzuführen ist. Diese werden nicht wie ursprünglich angenommen, durch Merge-Sort-Verbunde realisiert, sondern durch eine Armada von Nested-Loop-Verbunden, wie der Blick auf den Ausführungsplan offenbarte. Obwohl oder gerade weil für jedes Tupel erst auf den Index zugegriffen wird, bevor es schließlich aus dem Externspeicher geladen wird, dauert die Ausführung eines Verbunds mit vielen Tabellen sehr lange. Unerklärlich ist in diesem Fall jedoch, dass selbst bei einer leeren Tabelle die Query auf der dazugehörigen Sicht unerwartet viel Zeit in Anspruch nimmt.

Abschließend kann man sagen, dass der Ansatz 4 mit der Attributtabelle insgesamt der Testsieger ist, da dieser trotz der etwas längeren Gesamtbearbeitungszeit am einfachsten zu erweitern ist und keine Limitierungen in der Anzahl von Objekten bzw. Objekttattributen aufweist. Im Gegensatz zu den hierarchischen Ansätzen läuft man bei diesem nicht in Gefahr, die Seitengröße von 32 KB oder die maximale Anzahl von 1012 Spalten zu überschreiten und damit einen Fehler zu provozieren. Sämtliche Funktionen sind hier ohne Einschränkungen nutzbar.

9 Zusammenfassung

Gegenstand dieser Diplomarbeit war die Neuimplementierung der Komponente des Spatial Model Servers, welche die Übersetzung von AWQL nach SQL durchführt. Es sollte die Mehrfachvererbung eingeführt und dazu ein geeignetes Datenbankschema entwickelt werden.

Es haben sich insgesamt 7 Ansätze finden lassen, von denen letztlich 5 implementiert wurden. Dabei handelte es sich um die Ansätze „vertikale Partitionierung“, „modifizierte horizontale Partitionierung“, „hierarchischer Ansatz“, „modifizierter hierarchischer Ansatz“ und „Attributtabellenansatz“. Zu den beiden hierarchischen Ansätzen ließen sich noch zwei weitere Variationen finden, so dass sich die Anzahl der implementierten Lösungen auf insgesamt 7 summiert hat.

Der vertikale Ansatz teilt die Attribute der Objekttypen gemäß ihrer Vererbung vertikal auf. Für jeden Typ wird eine eigene Tabelle angelegt, in der nur die neu hinzugekommenen Attribute abgespeichert werden. Die Vererbungsbeziehungen werden hier, wie auch in allen übrigen Ansätzen, in gesonderten Metadatentabellen verwaltet.

Der horizontale Ansatz legt ebenfalls für jeden neuen Objekttyp eine eigene Tabelle an. Jedoch dupliziert dieser zusätzlich die ererbten Attribut der Supertypen. Da es für unsere Zwecke ausreicht, dass lediglich die Blatttabellen Objekte beherbergen, d.h. die Tabellen nur Tupel vom eigenen Objekttyp beinhalten, wurde dieser Ansatz in einer leicht veränderten Variante implementiert.

Der hierarchische Ansatz verwendet für die Speicherung der Objekte eine gemeinsame Tabelle. Darin sind Spalten für alle Objekttattribute enthalten. Aufgrund der Größe dieser hierarchisch aufgebauten Tabelle wurde dieser Ansatz auch mit einer vertikalen Partitionierung dieser Mastertabelle implementiert.

Zu jenem hierarchischen Ansatz wurde des weiteren eine Modifikation realisiert, die es auf einfache Art erlaubt, anhand der Angabe eines Objekttyps, Objekte aller direkten und indirekten Subtypen zu finden. Dazu wird in der Objekttypspalte für jedes Objekt der komplette Vererbungspfad in Form einer Liste abgelegt.

Der so genannte Attributtabellenansatz verwendet für die Speicherung der Objekte keine Objekttyptabellen sondern eine einzelne Attributtabelle. Darin werden Attribute zeilenweise abgelegt. Für jeden Datentyp existiert eine gesonderte Spalte. Über die Objekt-ID werden die Attribute später im Spatial Model Server wieder zu ganzen Objekten zusammengesetzt.

Der praktische Performancetest ergab, dass sich der Attributtabellenansatz für gängige Anfragen mit geringer bis mittlerer Komplexität am besten eignet, da dieser zum einen schnell ist und zum anderen keine Limitierungen in Bezug auf die Größe

des zu speichernden Datenbestandes aufweist. Die hierarchischen Ansätze, die allesamt sehr schnell waren, haben das Problem, dass die hierarchischen Tabellen nicht unbegrenzt viele Spalten aufnehmen dürfen. Die maximale Anzahl beträgt 1012 (DB2). Außerdem muss ein Tupel einer Tabelle immer vollständig in eine Seite passen. Daher sind diese Ansätze nur bedingt tauglich und eher für kleinere Datenbestände mit wenigen Objekttypen geeignet.

9.1 Ausblick

In diesem abschließenden Unterkapitel folgt ein kleiner Ausblick auf zum Teil bereits in Angriff genommene Erweiterungen des gefundenen Datenbankschemas (Attributtabellenansatz).

Das gefundene Datenbankschema (siehe 5.6) kann als Basis für weitere Entwicklungen dienen. Da die Attribute von den Objekten getrennt sind, könnte man sich vorstellen, einem Attribut mehrere Werte in Abhängigkeit der Zeit zuzuordnen. Dies bedarf lediglich einer weiteren Aufteilung der Attribute in Namen und Werte. Damit ließen sich Verläufe der Attributwerte erstellen, die zeitliche Anfragen erlauben würden.

Aufgrund der Trennung von Attribut und Objekt könnte man zudem häufig benötigte Attributwerte für immer wiederkehrende statische Anfrage z.B. repliziert auf mehrere Spatial Model Server speichern. Da man es nur mit einer einzigen Darstellung der Daten in Form einer Tabelle zu tun hat, macht es keine Probleme, einzelne Tupel oder ganze Objekte hin- und her zu kopieren.

10 Anhang

10.1 Datenbankschemadateien

Die vollständigen Datenbankskripte befinden sich im Projektordner DB2 bzw. auf der CD-ROM zu dieser Diplomarbeit.

10.2 Testfälle

Inhalt der Datei „testdata_2.awql“:

```
#0 einzelne type query nach Church
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
<restriction>
 <equal>
  <attr name="type"/>
  <nexusdata><Table>Church</Table></nexusdata>
 </equal>
</restriction>
</awql>

#1 Extent Query mit inside und includes-Filter
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
<restriction>
 <inside>
  <attr name="extent"/>
  <nexusdata><WKT>MULTIPOLYGON ((( 9.179444 48.785555, 9.18555 48.783611,
9.1783333 48.771944, 9.17138888 48.7761111, 9.179444
48.785555)))</WKT></nexusdata>
 </inside>
</restriction>
<filter>
 <includes>
  <attr name="name"/>
  <attr name="pos"/>
  <attr name="extent"/>
 </includes>
 <excludeallother />
</filter>
</awql>

#2 Query über 2 Attribute
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
<restriction>
 <and>
  <inside>
   <attr name="extent"/>
   <nexusdata><WKT>MULTIPOLYGON ((( 9.179444 48.785555, 9.18555 48.783611,
9.1783333 48.771944, 9.17138888 48.7761111, 9.179444
48.785555)))</WKT></nexusdata>
```

```
  </inside>
  <equal>
   <attr name="type"/>
   <nexusdata><Table>Church</Table></nexusdata>
  </equal>
 </and>
</restriction>
</awql>

#3 NOL Query mit equal
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
<restriction>
 <equal>
  <attr name="nol"/>
  <nexusdata><NOL>nexus:http://nexuspc5.informatik.uni-
stutt-
gart.de:8080/soap/servlet/rpcrouter|urn:QueryComponent|0x96e9fa12315b11d78b
f7080020a23633/0x79990559315d11d7af5a080020a23633</NOL></nexusdata>
 </equal>
</restriction>
</awql>

#4 query ueber 3 Attribute
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
<restriction>
 <and>
  <overlaps>
   <attr name="extent"/>
   <nexusdata><WKT>MULTIPOLYGON ((( 9.179444 48.785555, 9.18555 48.783611,
9.1783333 48.771944, 9.17138888 48.7761111, 9.179444
48.785555)))</WKT></nexusdata>
  </overlaps>
  <or>
   <greater>
    <attr name="altitude"/>
    <nexusdata><Integer>10</Integer></nexusdata>
   </greater>
   <like>
    <attr name="name"/>
    <nexusdata><String>Stuttgart</String></nexusdata>
   </like>
  </or>
 </and>
</restriction>
</awql>

#5 Extent Query mit equal
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
<restriction>
 <equal>
  <attr name="extent"/>
  <nexusdata><WKT>MULTIPOLYGON ((( 9.19044900 48.77894000, 9.19074800
48.77894600, 9.19074300 48.77905800, 9.19044400 48.77905100, 9.19044900
48.77894000)))</WKT></nexusdata>
 </equal>
</restriction>
</awql>
```

```
#6 type query nach NexusDataObject und closest 10 mit includes-filter
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
<restriction>
 <equal>
  <attr name="type"/>
  <nexusdata><Table>NexusDataObject</Table></nexusdata>
 </equal>
</restriction>
<closest num="10">
 <nexusdata><WKT>POINT ( 9.17895000 48.77115000)</WKT></nexusdata>
</closest>
<filter>
 <includes>
  <attr name="name"/>
  <attr name="pos"/>
  <attr name="extent"/>
</includes>
 <excludeallother />
</filter>
</awql>

#7 Name Query mit equal
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
<restriction>
 <equal>
  <attr name="name"/>
  <nexusdata><String>STAUFFENBERGSTRASSE</String></nexusdata>
 </equal>
</restriction>
</awql>

#8 Suche alle Objekte die museum oder MUSEUM im Namen oder der Beschreibung
tragen.
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
<restriction>
 <or>
 <like>
  <attr name="name"/>
  <nexusdata><String>museum</String></nexusdata>
 </like>
 <like>
  <attr name="name"/>
  <nexusdata><String>MUSEUM</String></nexusdata>
 </like>
 <like>
  <attr name="description"/>
  <nexusdata><String>Museum</String></nexusdata>
 </like>
 <like>
  <attr name="description"/>
  <nexusdata><String>MUSEUM</String></nexusdata>
 </like>
 </or>
</restriction>
</awql>
```

```
#9 Suche Objekte die entweder vom Typ Cinema und Bar sind ODER
# vom Typ Church aber nicht SightseeingBuilding
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
<restriction>
 <or>
  <and>
   <equal>
    <attr name="type"/>
    <nexusdata><Table>Cinema</Table></nexusdata>
   </equal>
   <equal>
    <attr name="type"/>
    <nexusdata><Table>Bar</Table></nexusdata>
   </equal>
  </and>
  <and>
   <not>
    <equal>
     <attr name="type"/>
     <nexusdata><Table>SightseeingBuilding</Table></nexusdata>
    </equal>
   </not>
   <equal>
    <attr name="type"/>
    <nexusdata><Table>Church</Table></nexusdata>
   </equal>
  </and>
 </or>
</restriction>
</awql>

#10 query nach bestimmten Strassen
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
<restriction>
 <and>
  <like>
   <attr name="name"/>
   <nexusdata><String>PLATZ</String></nexusdata>
  </like>
  <overlaps>
   <attr name="extent"/>
   <nexusdata><WKT>MULTIPOLYGON ((( 9.179444 48.785555, 9.18555 48.783611,
9.1783333 48.771944, 9.17138888 48.7761111, 9.179444
48.785555)))</WKT></nexusdata>
  </overlaps>
  <equal>
   <attr name="type"/>
   <nexusdata><Table>Road</Table></nexusdata>
  </equal>
 </and>
</restriction>
</awql>

#11 Query nach Emitter mit ID = 1302 auf dem Campus
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
<restriction>
 <and>
```

```
  <equal>
   <attr name="type"/>
   <nexusdata><Table>Emitter</Table></nexusdata>
  </equal>
  <equal>
   <attr name="emitterID"/>
   <nexusdata><String>1302</String></nexusdata>
  </equal>
  <overlaps>
   <attr name="pos"/>
   <nexusdata><WKT>MULTIPOLYGON ((( 9.10361111 48.74694444, 9.1088888
48.7466666, 9.1088888 48.7455555, 9.10472222 48.7438888, 9.10361111
48.74694444)))</WKT></nexusdata>
  </overlaps>
 </and>
</restriction>
</awql>

#12 Raumsuche anhand Position
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
<restriction>
 <and>
 <overlaps>
  <attr name="extent"/>
  <nexusdata><WKT>POINT ( 9.10660800 48.74496400)</WKT></nexusdata>
 </overlaps>
 <equal>
   <attr name="type"/>
   <nexusdata><Table>Room</Table></nexusdata>
  </equal>
 </and>
</restriction>
</awql>

#13 Update Doorplate, setze neue Position
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
<restriction>
 <equal>
  <attr name="nol"/>
  <nexusdata><NOL>nexus:http://nexuspc5.informatik.uni-
stutt-
gart.de:8080/soap/servlet/rpcrouter|urn:QueryComponent|0x70603058b5e111d7a6
64080020a23633/0xe998b42da17b11d7af59c95148accd40</NOL></nexusdata>
 </equal>
</restriction>
<update>
 <attr name="personPos"/>
 <nexusdata><WKT>POINT ( 9.10660800 48.74496400)</WKT></nexusdata>
 <attr name="status"/>
 <nexusdata><String>nicht anwesend</String></nexusdata>
 <attr name="place"/>
 <nexusdata><String>2.050 (Kommunikationsecke)</String></nexusdata>
</update>
</awql>

#14 update von vielen Objekten
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
```

```
<restriction>
 <and>
  <equal>
   <attr name="name"/>
   <nexusdata><String /></nexusdata>
  </equal>
  <equal>
   <attr name="type"/>
   <nexusdata><Table>Road</Table></nexusdata>
  </equal>
 </and>
</restriction>
<update>
  <attr name="name"/>
  <nexusdata><String /></nexusdata>
</update>
</awql>

#15 komplexe Query
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
<restriction>
<not>
 <and>
  <not>
   <equal>
    <attr name="nol"/>
    <nexusdata><NOL>nexus:http://nexuspc5.informatik.uni-
stutt-
gart.de:8080/soap/servlet/rpcrouter|urn:QueryComponent|0x70603058b5e111d7a6
64080020a23633/0xe998b42da17b11d7af59c95148accd40</NOL></nexusdata>
   </equal>
  </not>
  <and>
   <like>
    <attr name="name"/>
    <nexusdata><String>PLATZ</String></nexusdata>
   </like>
   <greater>
    <attr name="height"/>
    <nexusdata><Integer>0</Integer></nexusdata>
   </greater>
   <or>
    <not>
     <not>
      <less>
       <attr name="altitude"/>
       <nexusdata><Integer>10</Integer></nexusdata>
      </less>
     </not>
    </not>
    <equal>
     <attr name="pos"/>
     <nexusdata><WKT>POINT ( 9.10633300 48.74486400)</WKT></nexusdata>
    </equal>
    <overlaps>
     <attr name="extent"/>
     <nexusdata><WKT>MULTIPOLYGON ((( 9.10657700 48.74546100, 9.10664300
48.74545800, 9.10665000 48.74552300, 9.10658400 48.74552600, 9.10657700
48.74546100)))</WKT></nexusdata>
    </overlaps>
   </or>
```

```
  </and>
 </and>
</not>
</restriction>
</awql>

#16 komplexe Query 2
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
<restriction>
 <and>
  <or>
   <equal>
    <attr name="nol"/>
    <nexusdata><NOL>nexus:http://nexuspc5.informatik.uni-
stutt-
gart.de:8080/soap/servlet/rpcrouter|urn:QueryComponent|0x70603058b5e111d7a6
64080020a23633/0xe998b42da17b11d7af59c95148accd40</NOL></nexusdata>
   </equal>
   <equal>
    <attr name="nol"/>
    <nexusdata><NOL>nexus:http://nexuspc5.informatik.uni-
stutt-
gart.de:8080/soap/servlet/rpcrouter|urn:QueryComponent|0x96e9fa12315b11d78b
f7080020a23633/0xb3bac81231d011d7b27b080020a23633</NOL></nexusdata>
   </equal>
  </or>
  <or>
   <like>
    <attr name="name"/>
    <nexusdata><String>PLATZ</String></nexusdata>
   </like>
   <greater>
    <attr name="height"/>
    <nexusdata><Integer>0</Integer></nexusdata>
   </greater>
  </or>
  <or>
   <overlaps>
    <attr name="extent"/>
    <nexusdata><WKT>MULTIPOLYGON ((( 9.10657700 48.74546100, 9.10664300
48.74545800, 9.10665000 48.74552300, 9.10658400 48.74552600, 9.10657700
48.74546100)))</WKT></nexusdata>
   </overlaps>
   <equal>
    <attr name="pos"/>
    <nexusdata><WKT>POINT ( 9.10633300 48.74486400)</WKT></nexusdata>
   </equal>
  </or>
 </and>
</restriction>
</awql>

#17 Suchgebiet mit einem Loch in der Mitte
#   (Stadmitte ohne Domkirche St. Eberhard)
#   nur Buildings oder Roads
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
<restriction>
 <and>
  <overlaps>
```

```
   <attr name="extent"/>
   <nexusdata><WKT>MULTIPOLYGON ((( 9.179444 48.785555, 9.18555 48.783611,
9.1783333 48.771944, 9.17138888 48.7761111, 9.179444
48.785555)))</WKT></nexusdata>
  </overlaps>
  <not>
   <inside>
    <attr name="extent"/>
    <nexusdata><WKT>MULTIPOLYGON ((( 9.18111111 48.78166667, 9.18277778
48.7811111, 9.18166666 48.7788888, 9.17861111 48.7794444, 9.18111111
48.78166667)))</WKT></nexusdata>
   </inside>
  </not>
  <or>
   <equal>
    <attr name="type"/>
    <nexusdata><Table>Building</Table></nexusdata>
   </equal>
   <equal>
    <attr name="type"/>
    <nexusdata><Table>Road</Table></nexusdata>
   </equal>
  </or>
 </and>
</restriction>
</awql>

#18 Suche nach Objekten anhand von 8 Namen
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
<restriction>
 <or>
  <equal>
   <attr name="name"/>
   <nexusdata><String>Altes Schloss</String></nexusdata>
  </equal>
  <equal>
   <attr name="name"/>
   <nexusdata><String>Neues Schloss</String></nexusdata>
  </equal>
  <equal>
   <attr name="name"/>
   <nexusdata><String>Arbeitsamt</String></nexusdata>
  </equal>
  <equal>
   <attr name="name"/>
   <nexusdata><String>Postamt</String></nexusdata>
  </equal>
  <equal>
   <attr name="name"/>
   <nexusdata><String>WILHELMSPLATZ</String></nexusdata>
  </equal>
  <equal>
   <attr name="name"/>
   <nexusdata><String>Finde nichts</String></nexusdata>
  </equal>
  <equal>
   <attr name="name"/>
   <nexusdata><String>Suche zwecklos</String></nexusdata>
  </equal>
  <equal>
   <attr name="name"/>
```

```
   <nexusdata><String>Schau nochmal nach</String></nexusdata>
  </equal>
 </or>
</restriction>
</awql>

#19 Suche nach mehreren bestimmten Objekten
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
<restriction>
 <or>
  <and>
   <equal>
    <attr name="name"/>
    <nexusdata><String>Altes Schloss</String></nexusdata>
   </equal>
   <overlaps>
    <attr name="pos"/>
    <nexusdata><WKT>MULTIPOLYGON ((( 9.179444 48.785555, 9.18555 48.783611,
9.1783333 48.771944, 9.17138888 48.7761111, 9.179444
48.785555)))</WKT></nexusdata>
   </overlaps>
  </and>
  <and>
   <equal>
    <attr name="name"/>
    <nexusdata><String>Neues Schloss</String></nexusdata>
   </equal>
   <equal>
    <attr name="pos"/>
    <nexusdata><WKT>POINT ( 9.18139900 48.77810700)</WKT></nexusdata>
   </equal>
  </and>
  <and>
   <like>
    <attr name="name"/>
    <nexusdata><String>Sozialamt</String></nexusdata>
   </like>
   <overlaps>
    <attr name="extent"/>
    <nexusdata><WKT>MULTIPOLYGON ((( 9.179444 48.785555, 9.18555 48.783611,
9.1783333 48.771944, 9.17138888 48.7761111, 9.179444
48.785555)))</WKT></nexusdata>
   </overlaps>
  </and>
  <equal>
   <attr name="name"/>
   <nexusdata><String>Hauptpostamt</String></nexusdata>
  </equal>
  <and>
   <equal>
    <attr name="name"/>
    <nexusdata><String>WILHELMSPLATZ</String></nexusdata>
   </equal>
   <equal>
    <attr name="type"/>
    <nexusdata><Table>Road</Table></nexusdata>
   </equal>
  </and>
  <equal>
   <attr name="name"/>
   <nexusdata><String>Finde nichts</String></nexusdata>
```

```
  </equal>
  <equal>
   <attr name="name"/>
   <nexusdata><String>Suche zwecklos</String></nexusdata>
  </equal>
  <equal>
   <attr name="name"/>
   <nexusdata><String>Schau nochmal nach</String></nexusdata>
  </equal>
 </or>
</restriction>
<closest>
 <nexusdata><WKT>POINT ( 9.18083333 48.7794444)</WKT></nexusdata>
</closest>
</awql>

#20 Suche nach Objekten mit bestimmten Namen und Pos/Extent/Roadrun
<awql xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://nexuspc5.informatik.uni-
stuttgart.de:8080/spase/awql.xsd">
<restriction>
 <and>
  <or>
   <equal>
    <attr name="name"/>
    <nexusdata><String>Altes Schloss</String></nexusdata>
   </equal>
   <equal>
    <attr name="name"/>
    <nexusdata><String>Neues Schloss</String></nexusdata>
   </equal>
   <and>
    <equal>
     <attr name="type"/>
     <nexusdata><Table>Road</Table></nexusdata>
    </equal>
    <like>
     <attr name="name"/>
     <nexusdata><String>SCHLOSS</String></nexusdata>
    </like>
   </and>
  </or>
  <or>
   <overlaps>
    <attr name="pos"/>
    <nexusdata><WKT>MULTIPOLYGON ((( 9.179444 48.785555, 9.18555 48.783611,
9.1783333 48.771944, 9.17138888 48.7761111, 9.179444
48.785555)))</WKT></nexusdata>
   </overlaps>
   <inside>
    <attr name="extent"/>
    <nexusdata><WKT>MULTIPOLYGON ((( 9.179444 48.785555, 9.18555 48.783611,
9.1783333 48.771944, 9.17138888 48.7761111, 9.179444
48.785555)))</WKT></nexusdata>
   </inside>
   <overlaps>
    <attr name="roadRun"/>
    <nexusdata><WKT>MULTIPOLYGON ((( 9.179444 48.785555, 9.18555 48.783611,
9.1783333 48.771944, 9.17138888 48.7761111, 9.179444
48.785555)))</WKT></nexusdata>
   </overlaps>
  </or>
  <not>
```

```
   <equal>
    <attr name="type"/>
    <nexusdata><Table>EventBuilding</Table></nexusdata>
   </equal>
  </not>
 </and>
</restriction>
</awql>
```

11 Literaturverzeichnis

Ass83 G. Asser. *Einführung in die mathematische Logik – Teil 1: Aussagenkalkül.* Teubner Verlag, Leipzig, 1982.

CCN+99 M. Carey, D. Chamberlin, S. Narayanan, B. Vance, D. Doole, S. Rielau, R. Swagerman, N. Mattos. *O-O, What Have They Done to DB2?* Proceedings of the 25th VLDB Conference, Edingburgh, Scotland, 1999.

Fin02 M. Finsterwalder. *Anbindung objektorientierter Software an objektrelationale Datenbanken*. Diplomarbeit, Universität Hamburg, Oktober 2002.

Fri02 S. Fritsch. *Vorhandene Dienste in Nexus: Lokations- und Ereignisdienst*. Universität Stuttgart, Dezember 2002, Seminarausarbeitung. Download unter http://www.informatik.uni-stuttgart.de/ipvr/as/lehre/seminar/docws02/Lokationsdienst_Ereignisdienst_ Serena.pdf.

HR01 T. Härder, E. Rahm. *Datenbanksysteme – Konzepte und Techniken der Implementierung*. Springer Verlag, 2. Auflage, Berlin 2001.

IBM02 IBM® DB2 Universal Database™ SQL Reference, Version 8, 2002.

KJA93 A. M. Keller, R. Jensen, S. Agrawal. *Persistence Software: Bridging Object-Oriented Programming and Relational Databases*. Peter Buneman, Sushil Jajodia (Eds.): Proceeding of the 1993 ACM SIGMOD International Conference On Management of Data, Washington, D. C., May 1993, pp. 523-528.

ME92 P. Mishra, M. H. Eich. *Join processing in relational databases.* ACM CSUR, Volume 24, Issue 1, März 1992.

MS02 W. Mahnke, H.-P. Steiert. *Zum Einsatzpotential von ORDBMS in Entwurfsumgebungen*. Universität Kaiserslautern, FB Informatik, 2002.

NGS+01 D. Nicklas, M. Großmann, T. Schwarz, S. Volz, B. Mitschang. *A Model-Based, Open Architecture for Mobile, Spatially Aware Applications*. Advances in Spatial and Temporal Databases, SSTD 2001, LNCS 2121, Springer Verlag, 2001.

RBP+93 J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. *Objektorientiertes Modellieren und Entwerfen*. Prentice-Hall Internat., London, 1993.

Sch95 U. Schöning. *Logik für Informatiker*. 4. Auflage, Spektrum, Akad. Verlag, 1995.

SNG+01 T. Schwarz, D. Nicklas, M. Großmann, S. Volz. *Information Management and Exchange in Nexus.* Technical Report, Research Group Nexus, Universität Stuttgart 2001.

Ste02 H.-P. Steiert. *Aspekte der generativen Entwicklung von ORDBMS-basierten Datenverwaltungsdiensten*. Dissertation. Universität Kaiserslautern, 2002.

Til02 A. Till. *Erweiterter Notifikationsdienst für Nexus*. Diplomarbeit, Institut für parallele und verteilte Systeme, Fakultät Informatik, Universität Stuttgart, Diplomarbeit Nr. 2020 (2002).

VV01 I. Varlamis, M. Vazirgiannis. *Document Databases: Bridging XML-Schema and relational databases: A System for generating and manipulating relational databases using valid XML documents*. Proceedings of the 2001 ACM Symposium on Document engineering, November 2001.

Erklärung

Ich versichere, dass ich die Arbeit selbständig verfasst und nur die angegebenen Hilfsmittel verwendet habe.

(Thomas Wahl)