AG Datenbanken und Informationssysteme
Prof. Dr. Theo Härder

Entwurf und Implementierung der Anfrageverarbeitung von Meta-Akad

Projektarbeit
von
Björn Wagner

Betreuer:
Marcus Flehmig

Januar 2003

Inhaltsverzeichnis

Kapitel 1 Einleitung und Motivation

Über das Internet stehen heute eine Vielzahl von Lehr- und Lernmaterialien zur Verfügung. Größtenteils sind diese jedoch nur einem begrenzten Personenkreis bekannt und nur unzureichend erschlossen, was ein Auffinden geeigneter Dokumente erschwert.

Das Projekt Meta-Akad verfolgt das Ziel, jedem Nutzer einen möglichst umfassenden, einfachen und schnellen Zugriff auf das im Internet verfügbares Lehr- und Lernmaterial zu ermöglichen. Zu den Anwendern zählen dabei Lehrer und Dozenten die das zur Verfügung gestellte Material in ihren Veranstaltungen nutzen können, sowie Schüler, Studenten und andere, die im Internet verfügbares Material zum Selbststudium oder Vertiefung nutzen möchten. Das Projekt hat das Ziel einen Service, der durch entsprechende Organisationsstrukturen sowie entsprechende technische Mittel einen einfachen und bedarfsgerechten Zugang zu den gewünschten Lehrmitteln bietet, zur Verfügung zu stellen.

1.1 Gliederung der Arbeit

Diese Projektarbeit beschreibt die Anforderungen, den Aufbau und die Implementierung der Anfrageverarbeitung (*Query-Engine*).

Im diesem Kapitel werden die Zielsetzungen des Meta-Akad Projekts und die Realisierungsmöglichkeiten mit dem Java 2 Enterprise Edition Framework erörtert. Ferner wird die Einordnung der Anfrageverarbeitung in das Gesamtsystem gezeigt. Das zweite Kapitel erläutert grob Anforderungen sowie Ablauf der Anfrageverarbeitung und stellt das Implementierungskonzept dar. In den Nachfolgenden Kapitel wird dann näher auf die einzelnen Phasen der Verarbeitung und die auftretenden Probleme eingegangen.

Am Ende werden im Kapitel Ausblick Möglichkeiten für Erweiterungen und Verbesserungen der Anfrageverarbeitung des Meta-Akad Suchdienstes dargelegt.

1.2 Anforderungen an Meta-Akad

Der Zugang soll als *W*eb-basierte virtuelle Bibliothek realisiert werden, über die für die Lehrenden und Lernenden ein benutzerfreundlicher, schneller Zugriff auf die Dokumente ermöglicht wird.

Um dies zu realisieren, soll Online-Material auf kooperativer Basis gesammelt und eine möglichst umfangreiche, repräsentative Sammlung von qualitativ hochwertigen Materialien aufgebaut werden. Dabei muss die heterogene, fächerübergreifende Struktur des Materials durch die Erfassung von entsprechenden Metadaten berücksichtigt werden. Weiterhin muss auch ein einfacher Austausch von Daten mit anderen metadatenbasierten Diensten möglich sein. Die Qualität der Dokumente wird durch die Bewertung nach inhaltlichen und didaktischen Kriterien, sowie einer intellektuellen Nachbearbeitung erreicht.

Durch diese Qualitätssicherung wird ein wesentlicher Mehrwert des Dienstes gegenüber herkömmlichen Suchmaschinen erreicht. Der Nutzer erhält dadurch nicht nur Auskunft über die Verfügbarkeit eines Dokumentes sondern auch über dessen Qualität.

1.3 Java 2 Enterprise Edition

Die Anforderungen an den Dienst führen zu einem System mit einer sehr komplexen Struktur. Die Systemarchitektur muss diese Anforderungen berücksichtigen und zugleich für zukünftige Entwicklungen und Erweiterungen vorbereitet sein. Um dies zu gewährleisten wurde Java™2 Enterprise Edition (J2EE) von Sun Microsystems als Plattform gewählt.

Diese bietet einen Hersteller unabhängigen Standard für verteilte Java-basierte Anwendungen. Durch die Bereitstellung von grundlegenden Diensten (z.B. Namens- und Verzeichnisdiensten oder Transaktionsverwaltung) und die Unterstützung von allgemeinen Konzepten (wie beispielsweise Client/Server Architektur oder Persistenz von Objekten) ermöglicht sie die Entwicklung von mehrschichtigen, auf standardisierten Komponenten basierenden Systemen. Auf Techniken und Konzepte der Java™2-Standard-Edition (J2SE) aufbauend, unterstützt J2EE auch Datenbankzugriff mit JDBC, Enterprise-JavaBeans™ (EJB), Java-Server-Pages™(JSP), die Java-Servlet-Schnittstelle, XML und CORBA was die Entwicklung von unternehmensweiten Anwendungen ermöglicht.

Das J2EE Modell für unternehmensweite Anwendungen definiert eine Dreiteilung des Systems. Diese Aufsplitterung beruht auf der getrennten Betrachtung (Realisierung) von Präsentations-, Anwendungs- und Datenhaltungsaspekten in unterschiedlichen Schichten.

Die Präsentationslogik (Web-Tier) ist nochmals unterteilt in eine Client-seitige und eine Server-seitige Präsentationsschicht. Die Serverseite umfasst dabei den Webserver, der eine entsprechdene Infrastruktur (Web-Container) zur Ausführung der JSP-Seiten und Servlets zur Verfügung stellt. Auf der Clientseite sind verschiedene Komponenten, wie Web-Browser, Applets oder auch eigenständige Java Programme berücksichtigt, die auf Dienste der Serverschicht zugreifen können.

In der Anwendungsschicht (EJB-Tier) findet man die die entsprechende Umgebung (EJB-Container) für die Ausführung von EJB-Komponenten. EJBs sind wiederverwendbare Softwarekomponenten, die unabhängig erzeugt und gepflegt werden können. Ferner sind sie mit standardisierten Schnittstellen ausgestattet, um mit anderen Komponenten zusammenzuarbeiten. Die Infrastruktur dafür wird vom EJB-Container zur Verfügung gestellt. Es gibt zwei grundlegende Arten von EJBs. Entity-EJB und Session-EJB. Während letztere das Verhalten und die Geschäftslogik realisieren, repräsentieren Entity-Beans persistente Geschäftsobjekte, wie zum Beispiel ein einzelnes Tupel einer Datenbank. Der EJB-Container und der Web-Container bilden zusammen mit weiteren Diensten den Java-Application-Server.

In der dritten Ebene, der Datenhaltungsschicht (EIS-Tier) findet man Datenbanksysteme und so genannte Enterprise-Information-Systeme. Die einzelnen Schichten und Komponenten können größtenteils getrennt voneinander betrachtet und realisiert werden, was vor allem für Realisierung von großen Projekten von Vorteil ist.

1.4 Systemarchitektur

Der Architektur des Meta-Akad Dienstes liegt die, auch oben genannte Trennung in Präsentations-, Anwendungs- und Datenhaltungsgesichtspunkte zugrunde.

Die System besteht somit aus drei Schichten (), die die unterschiedlichen Aspekte realisieren :

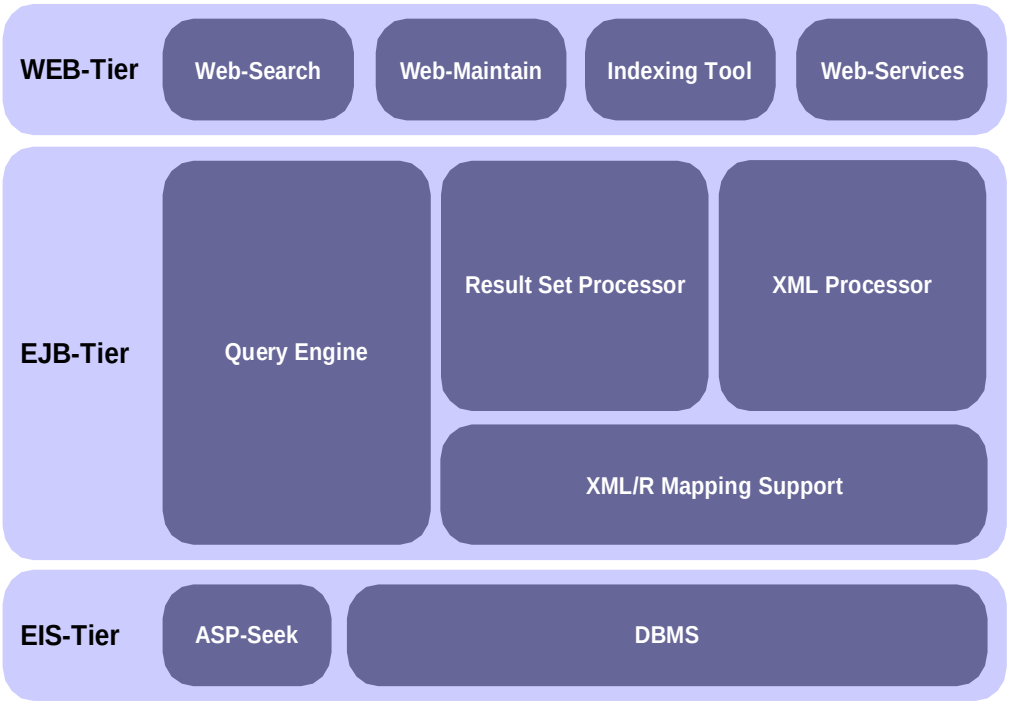
- Die *Präsentationslogik* (*Web-Tier)* verwendet auf der Clientseite einen Standard Web-Browser. Die Serverseite stellt verschiedene Dienste für Benutzung und Verwaltung des Systems zur Verfügung. Dies sind eine Web-basierte Suche (Web-Search), ein

Redaktionssystem (Indexing-Tool), eine Benutzerverwaltung (Web-Maintain) sowie unterstützende Dienste (Web-Services).

- Die *Anwendungslogik (EJB-Tier)* besteht aus den zentralen Komponenten des Systems: Der Anfrageverarbeitung (Query-Engine), der Ergebnisverarbeitung (Result Set Processor) und dem XML-Dokumenten-Management (XML Processor und XML/R Mapping Support).
- Die *Datenhaltungskomponente (EIS-Tier)* stellt eine Volltextsuche(ASP-Seek) und eine relationale Datenbank(DBMS) zur Verfügung.

stellt die verschiedenen Komponenten der Schichten grafisch dar und deutet ihre Beziehung zueinander an.

Abbildung 1 Meta-Akad Architektur Überblick



Aufbauend auf der Datenhaltungsschicht (*EIS-Tier*) realisieren drei Hauptkomponeten (*Query Engine*, *Result Set Processor* und *XML Processor*)der Anwendungsschicht (*EJB-Tier*) die eigentliche Anwendungslogik. Die Präsentationsschicht (*WEB-Tier*) nutzt diese, als EJBs implementierten, Module und realisiert die verschiedenen Webanwendungen.

Die Anfrageverarbeitung (*Query-Engine*) erhält von der jeweiligen Anwendung des *Web-Tier* eine Anfrage, die dann nach einer internen Weiterverarbeitung als temporäre Tabelle in einem relationalen Datenbanksystem materialisiert wird. Diese Ergebnistabellen werden in einer Verwaltungstabelle unter einem eindeutigem Schlüssel registriert. Mit Hilfe dieses Schlüssels, der in Form eines *ResultSetHandle*-Objekts zurückgeliefert wird, kann relativ einfach auf die Ergebnisse der Anfrage zugegriffen werden. Das Löschen von nicht mehr benötigten Daten obliegt dem aufrufenden oder ergebnisverarbeitenden Modul. Die *QueryEngine* löscht die angelegten Tabellen und Einträge nicht mehr, da die weitere Verwendungen der Daten nicht absehbar ist.

Kapitel 2 Anfrageverarbeitung von Meta-Akad

Die *Anfrageverarbeitung (AV)* ist eine der drei Hauptkomponenten der Anwendungslogik des Systems. Sowohl für die Web-Suche, als auch für das Redaktionssystem bildet sie eine zentrale Schnittstelle zum Datenhaltungsystem und ermöglicht es Dokumente auszuwählen und anschließend zu verarbeiten.

2.1 Anforderungen

Die AV hat die Aufgabe, beliebig formulierte Suchanfragen entgegenzunehmen und ein entsprechendes Ergebnis zurückzuliefern. Es soll möglich sein, aus den Meta-Akad Attributen oder den XML Elementen einfach Prädikate (wie beispielsweise **`Autor = 'Göb, Norbert'`**) zu formulieren und diese dann zu komplexeren Ausdrücken zusammenzusetzen (z.B. **`[ Autor = 'Göb, Norbert' ] AND [ Titel enthält 'Algebra']`**).

Weiterhin soll die Möglichkeit bestehen die Suche nach Meta-Daten mit einer Volltextsuche zu kombinieren. Ferner muss die Möglichkeit berücksichtigt werden für zunkünftige Erweiterungen Anfragen zwischenzuspeichern um kostenintensive Wiederholungen von Anfragen effizienter bearbeiten zu können.

Das von der AV gelieferte Ergebnis muss in einer Form vorliegen, die es erlaubt die AV sinnvoll von der Ergebnisverarbeitung zu trennen. Es sollte möglich sein das Ergebnis im *Result-Set-Processor* nach verschiedenen Kriterien zu sortieren und eine aktuelle Position Offset (Cursor) zu setzen. Falls eine Volltextsuche durchgeführt wurde soll ein zu den jeweiligen Dokument passender Volltextauszug mitgeliefert werden. Die Ergebnisse einer Anfrage werde in temporären Tabellen gespeichert, was zu einen einfacheren Zugriff führt. Im Wesentlichen besteht die Aufgabe der AV darin in der syntaktischen Analyse der Anfrage, ihrer Optimierung, der Übersetzung und der Ausführung. Bei Meta-Akad besteht jedoch die Besonderheit, dass eine Anfrage zwei verschiedene Datenhaltungssysteme betreffen kann. Zum einen das relationale DMBS mit den Metadaten und zum andern die Volltextdatenbank. Die AV muss also Anfragen an beide System stellen und die Ergebnisse entsprechend kombinieren.

2.2 Phaseneinteilung

Die Verarbeitung einer Anfrage kann mit einem einfachen 'top-down'-Ansatz beschrieben werden. An [MITCH95] angelehnt, kann die AV in fünf Phasen eingeteilt werden:

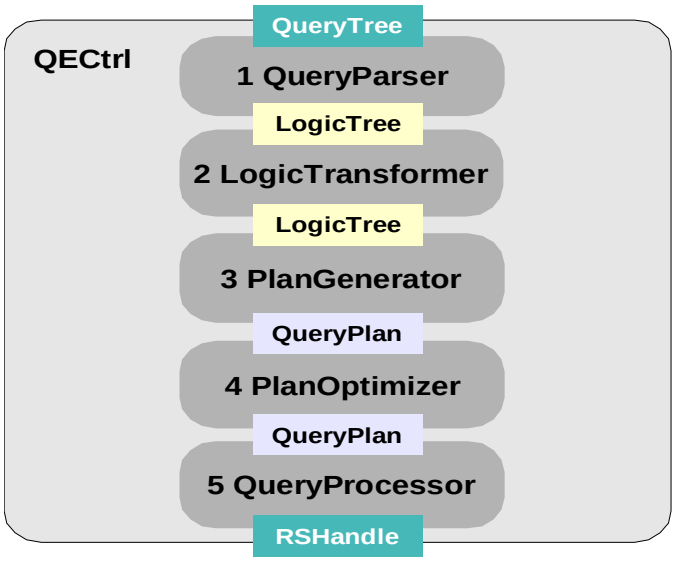
- Schritt 1: *Interndarstellung der Anfrage*. Um die Anfrage effizient verarbeiten zu können, wird sie vom *QueryParser* in ein geeignetes internes Format umgewandelt. Diese interne Darstellung soll einfach zu übersetzen sein und anschließend folgende Transformation der Anfrage unterstützen. Weiterhin wird eine Syntaxanalyse der Anfrage durchführt.
- Schritt 2: *Transformation der Anfrage*. Logische Umformungen sollen die Anfrage standardisieren und eventuell auch zusammenfassen. Geeignete Transformationen führen zu einem effizienteren Anfrageplan.
- Schritt 3: Erzeugung eines Anfrageplans. Der Plangenerator erzeugt aus der internen Darstellung einen Anfrageplan (*QueryPlan*), der die Grundlage für die spätere SQL-

Anfrage an die relationale Datenbank bildet. Ferner führt er Vorbereitungen für die Volltextsuche durch, indem er entsprechende temporäre Tabellen in der Datenbank erstellt.

- Schritt 4: *Optimieren des Anfrageplans.* Der im letzten Schritt erzeugte Anfrageplan führt nicht automatisch zu einer optimalen SQL-Anfrage. Der *PlanOptimizer* versucht nun den Anfrageplan entsprechend umzuformen um eine möglichst optimale SQL-Anfrage zu erhalten.
- Schritt 5: *Ausführung des Anfrageplans*. Im fünften und letzten Schritt wird aus dem Anfrageplan eine SQL-Anfrage generiert und ausgeführt. Das Resultat der Anfrage wird entsprechend aufbereitet, in eine temporäre Tabelle geschrieben und ein Handle für den Zugriff zurückgeliefert.

Diese fünf Schritte beschreiben im Grunde, die aufeinander folgenden Verarbeitungsphasen einer Anfrage. Der daraus folgende Ablauf ist in nochmals grafisch dargestellt ist.

Abbildung 2 Ablauf der Anfrageverarbeitung



Die Kontrolle des Ablaufs obliegt dem Kontrollmodul (*QECtrl*). Dieses EJB wird von den Komponenten des WEB-Tier benutzt und liefert ein Handle für das Resultat zurück.

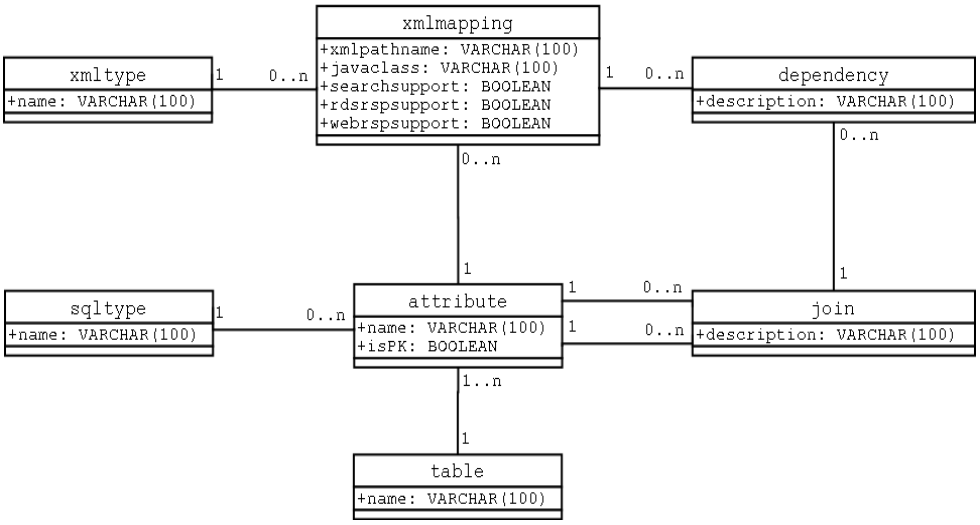
2.3 Implementierung

Da die Anfrageverarbeitung möglichst einfach erweiterbar und wartbar sein soll wurden die einzelnen Verarbeitungsphasen als möglichst unabhängige Komponenten implementiert. Das zentrale Kontrollmodul *QECtrl* ist als "*stateless Session-EJB*" realisiert. Es bietet entsprechende Schnittstellen an um eine Anfrage durchzuführen und realisiert die Transaktionsverwaltung. Dieses EJB führt nun die für die Abwicklung der Anfrage benötigten Schritte durch, indem es die entsprechenden Komponenten in entsprechender Reihenfolge aufruft und die Ergebnisse aufbereitet und verwaltet. Dies ist auch eine geeignete Stelle, um Erweiterungen wie Anfrageanalysen oder Caching in das System einzufügen.

Um eine entsprechende Anfrage generisch formulieren zu können, muss die Struktur der Datenbank in einer entsprechenden Form verfügbar sein. Die vorhandenen MetaAkad

Attribute werden durch das so genannte Metamodell beschrieben. Dabei handelt es sich um eine relationale Datenbank, welche die Zuordnung von XML-Attributen und Tabellenspalten, sowie die Beziehung zwischen den einzelnen Attributen beschreibt. Der Aufbau dieser Datenbank ist aus dem UML-Diagramm in ersichtlich.

Abbildung 3 UML-Diagramm Metamodell



Der Zugriff auf das Metamodell wurde mit einem Entity-EJB realisiert und in der Klasse *MetaModelDAO* gekapselt. Diese realisert das "*DataAccessObject*" Pattern um einfach auf die benötigten Daten zugreifen zu können. Ein *DataAccessObject* (*DAO*) kapselt den Datenzugriff auf externe Ressourcen um dem Anwender einen möglichst einfachen und komfortablen Zugriff zu ermöglichen. Durch die Realisierung als DAO wird der Zugriff auf das EJB Objekt wesentlich vereinfacht, da sich der Benutzer nicht mehr selbst um die Erzeugung des EJB-Objekts, den Datenzugriff und die die damit verbundenen Ausnahmefälle kümmern muss.

Das *MetaModelDAO* stellt drei Methoden zur Verfügung, die für einen gegebenen XML-Bezeichner, ein Attributobjekt, eine Liste der Abhängigkeiten und einen Typen Bezeichner liefern. Ferner muss das Ergebnis der Volltextsuche einer geeigneten Form verfügbar gemacht werden. Um die Volltextsuche möglichst unkompliziert und unabhängig in das System zu integrieren, wurde beschlossen, das Ergebnis des Suche als temporäre Tabelle in einer relationalen Datenbank zu materialisieren.

Diese Vorgehensweise ermöglicht es das Ergebnis der Volltextsuche direkt in die SQL-Anfrage einzubeziehen, ohne dass ein externes Zusammenführen der Ergebnisse erforderlich wäre. Ferner können so Suchergebnisse gespeichert und für andere Anfragen wiederverwendet werden.

Wie bei den Metadaten wurde auch hier der Zugriff über ein *DAO* implementiert. Diese bietet eine Methode die für einen übergebenen Suchtext den Namen einer Tabelle mit dem Ergebnis der Suche liefert. Diese Tabelle kann dann problemlos für die SQL-Anfrage verwendet werden. Für die Verwaltung dieser temporären Tabellen ist das FulltextCacheModul verantwortlich.

Um den Zugriff auf Dienste wie EJB-Objekte und Datenquellen zu erleichtern ist dieser mit Hilfe des *ServiceLocator* Moduls realisiert. Dieses Singelton bietet einen einfachen, optimierten Zugriff auf verteilte Objekte und Datenquellen.

Kapitel 3 Interndarstellung der Anfrage

Die AV erhält die Anfrage in einem speziellen Format. Um eine Anfrage zu formulieren, wurde hierfür ein binärer Baum aus Java Objekten gewählt. Dieser Suchbaum (*QueryTree*) bietet dem Anwender eine relativ einfache, Schema unabhängige Möglichkeit eine Anfrage an die Datenbank zu stellen. Ferner bietet sie die Möglichkeit die Volltextsuche unkompliziert zu integrieren. Der Nachteil dieser Art "Anfragesprache" liegt darin, dass nicht mehr alle Möglichkeiten einer direkten SQL-Abfrage gegeben sind.

3.1 Aufbau des Suchbaums (QueryTree)

Die Struktur des QueryTree erlaubt es einfach formulierte Suchanfragen zu stellen. Sie ermöglicht es Prädikate aus den Meta-Akad Attributen zu formulieren und diese dann mit Verknüpfungsoperatoren zu komplexeren Anfragen zusammenzufügen. Die Prädikate bilden dabei die Blätter des binären Baums.

Es gibt zwei Typen von Prädikaten. Der Anwender hat die Möglichkeit zu überprüfen, ob ein Attribut gesetzt ist oder Attribute einzuschränken. Dafür stehen bisher fünf Vergleichsoperatoren sowie die Möglichkeit nach dem Vorkommen eines Teilwortes, bzw. Präfix oder Postfix zu suchen zur Verfügung (siehe auch Abbildung 1). Die Objekte mit denen ein Attribut verglichen werden soll, können je nach verwendetem Operator einen der Datentypen "java.lang.String", "java.lang.Integer", "java.lang.Boolean" oder "java.util.Date" haben. Ferner besteht die Möglichkeit bei jedem Prädikat festzulegen wie das betreffende Attribut erfasst worden sein soll (automatisch, intellektuell, etc.). Dieser so genannte "Origin" wird im Zuge der Verarbeitung wie ein eigenständiges Attribut behandelt und entsprechend in der Anfrage berücksichtigt.

Tabelle 1 Zuordnung von Vergleichsoperatoren und Blattypen

Operator	Blattypen
=	LeafEqual
>	LeafGreater
>=	LeafEvenGreater
<	LeafLess
<=	LeafEvenLess
Enthält Teilwort	LeafContains
Enthält Präfix	LeafContainsLeft
Enthält Postfix	LeafContainsRight

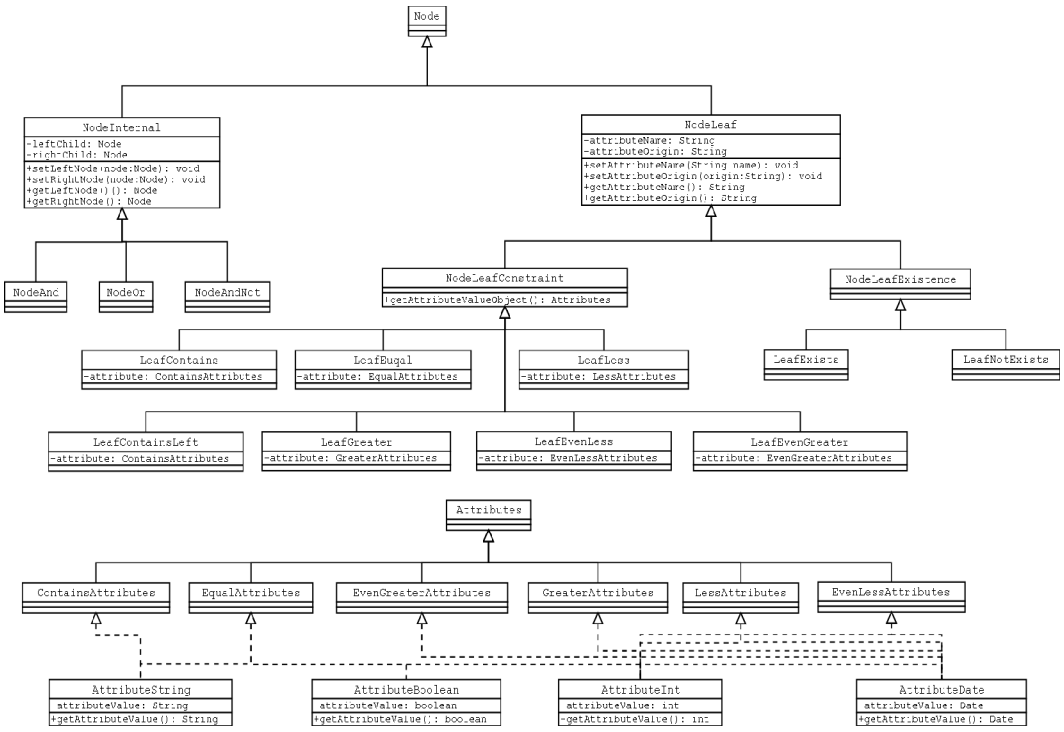
Um nun komplexere Anfragen formulieren zu können, besteht die Möglichkeit mehrere Prädikate mit den logischen Operatoren AND, OR und AND NOT zu kombinieren (siehe dazu Abbildung 2).

Tabelle 2 Zuordnung von logischen Operatoren und Knoten

Operator	Knoten
AND	NodeA
OR	NodeOr
AND NOT	NodeAndNot

Die Volltextsuche wird in einem QueryTree durch das spezielle Meta-Akad Attribut "fulltext" repräsentiert. Bei der Verarbeitung des QueryTree muss dies entsprechend beachtet werden. Die genaue Struktur der QueryTree Elemente ist aus dem UML-Diagramm in ersichtlich.

Abbildung 4 UML-Diagramm QueryTree Elemente



3.2 Aufbau der Interndarstellung (LogicTree)

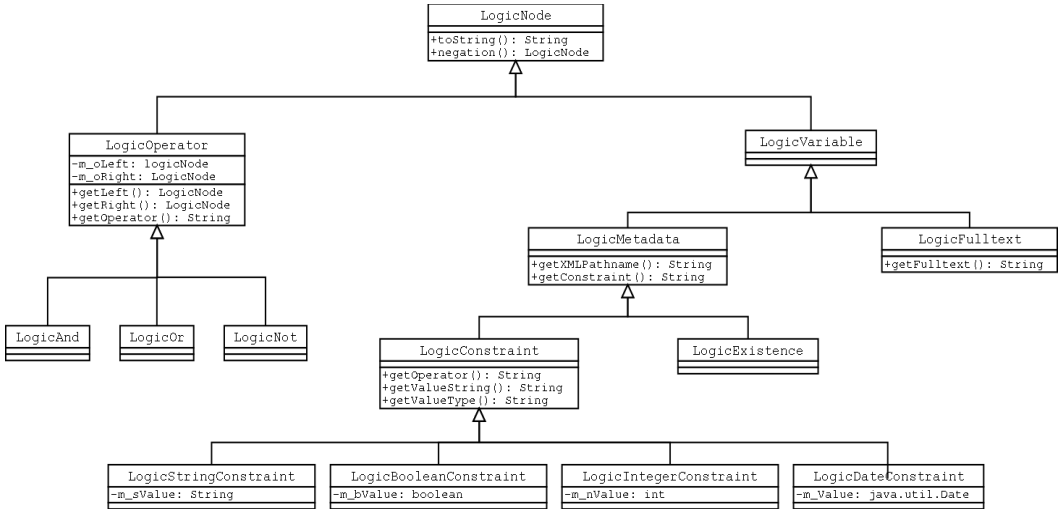
Um die Anfrage effizient und einfach verarbeiten zu können wird sie in eine entsprechende interne Repräsentation umgewandelt. Die interne Darstellung (LogicTree) einer Anfrage ist wie die Anfrage selbst mit Prädikaten als Blättern (LogicVariable) und Operatoren als Knoten (LogicOperator) ein binärer Baum. Im Gegensatz zum QueryTree berücksichtigt der LogicTree die drei vorkommenden Typen von Prädikaten mit einer hierarchischen Klassenstruktur. Die klare Trennung der verschiedenen Prädikattypen ist für das Erstellen des Anfrageplans und die Transformation eine große Erleichterung. Ferner bilden die zur Verfügung stehenden logischen Operatoren AND, OR und NOT eine Basis, das heißt jeder logische Term ist somit als ein LogicTree darstellbar. Diese Eigenschaft ist vor allem wichtig für spätere Transformationen der internen Darstellung.

Die Prädikate teilen sich in zwei Gruppen: Metadaten-Attribute (LogicMetaData) und Volltextsuche (LogicFulltext). Volltextprädikate enthalten nur den Suchtext. Metadaten-Prädikate hingegen enthalten den Attributbezeichner (XML-Pfad) und entsprechende Zusatzinformationen, je nachdem ob es sich um Existenzprüfungen (LogicExistence) oder Einschränkungen (LogicConstraint) handelt.

Existenzprüfungen benötigen außer dem Attributbezeichner keine Informationen. Einschränkungen hingegen haben einen Vergleichsoperator und einen Vergleichswert. Anders als beim Anfragebaum sind hier nicht die Operatoren als Ableitungen der Prädikatklasse realisiert, sondern die Datentypen. Dies resultiert vor allem aus der Notwendigkeit, die Werte in korrekte "SQL-Strings" umzuwandeln. Durch Überschreiben einer entsprechenden Funktion in LogicConstraint ist dies am effizientesten zu lösen. Die Operatoren werden einfach als Klassen-Attribute vom Typ String gespeichert.

Diese Vorgehensweise überprüft zwar nicht, ob die Operatoren und Datentypen korrekt zueinander passen, dies kann aber an dieser Stelle bedenkenlos außer Acht gelassen werden, da angenommen werden kann, dass der vom QueryParser erzeugte LogicTree korrekt ist. Das UML-Diagramm in verdeutlicht die Struktur der internen Darstellung noch einmal graphisch.

Abbildung 5 UML Diagramm Logic Tree



3.3 Erstellen der Interndarstellung (QueryParser)

Der *QueryParser* hat die Aufgabe für einen gegebenen *QueryTree* einen äquivalenten *LogicTree* zu erzeugen. Die binäre Baumstruktur erlaubt es einen recht einfachen rekursiven Algorithmus (*parseNode*) zu formulieren der diese Arbeit durchführt. Dieser Algorithmus erstellt für einen übergebenen Knoten des *QueryTree* einen entsprechenden Knoten des *LogicTree*. Erhält er einen Operatorknoten (*NodeInternal*), wird der Algorithmus zuerst auf die beiden Operanden angewendet, die dann einem entsprechenden neu erzeugten Operatorknoten (*LogicOperator*) als Operanden übergeben werden. Siehe dazu Codefragment 1.

Zum jetzigen Zeitpunkt unterstützt der *QueryParser* die Operatoren AND, OR und ANDNOT. Der Algorithmus kann jederzeit durch Hinzunahme einer weiteren Bedingung angepasst werden. Sollte ein unbekannter Operatortyp übergeben werden wird eine Fehlermeldung ausgelöst.

Codefragment 1 QueryParser, Operatorknoten erzeugen

```
NodeInternal operatorNode = (NodeInternal) node;
LogicNode tl = parseNode( operatorNode.getLeftNode() );
LogicNode tr = parseNode( operatorNode.getRightNode() );

if( operatorNode instanceof NodeAnd ) {
    /* and */
    return new LogicAnd(  tl, tr );
} else if( operatorNode instanceof NodeOr ) {
    /* or */
    return new LogicOr(  tl, tr );
} else if( operatorNode instanceof NodeAndNot ) {
    /* and not */
    return new LogicAnd(  tl, new LogicNot( tr ) );
} else {
    /* not implemented */
    throw new QueryParserException("Operator not implemented yet !");
}
```

Handelt es sich bei dem zu verarbeitenden Knoten hingegen um einen Blattknoten (*NodeLeaf*), also ein Prädikat, wird er vom zweiten Teil der Routine verarbeitet. Diese unterscheidet zuerst, ob es sich um eine Einschränkung (*NodeLeafConstraint*) oder eine Existenzprüfung (*NodeLeafExistence*) handelt. Bei letzterem wird einfach ein *LogicExistence* Knoten mit dem entsprechenden Attributnamen des Blattes erzeugt (siehe dazu Codefragment 2). Bei einer Einschränkung hingegen muss der *QueryParser* eine mehr oder weniger aufwendige Umwandlung durchführen.

Codefragment 2 QueryParser, Blattknoten für Existenzprüfung erzeugen

```
if( node instanceof NodeLeafExistence ) {
       /* check if attribute exists */
       NodeLeafExistence leaf = (NodeLeafExistence) node;
       if(  node instanceof LeafExists ) {
              return new LogicExistence( leaf.getAttributeName() );
       } else if( node instanceof LeafNotExists ) {
              return new LogicNot( new LogicExistence( leaf.getAttributeName() ) );
       } else {
                     /* not implemented */
                     throw new QueryParserException("Type of leaf not implemented yet !");
       }
}
```

Bei einem *NodeLeafConstraint* wird zuerst überprüft, ob eine Volltextsuche durchgeführt werden soll. Ist dies der Fall, wird ein *LogicFulltext* Knoten mit entsprechendem Volltext zurückgegeben. Zuvor wird jedoch noch der Datentyp überprüft und gegebenenfalls eine Fehlermeldung ausgelöst. Für Volltextsuche werden logischerweise nur Strings akzeptiert. Der Vergleichsoperator wird ignoriert. Wenn es sich hingegen um eine Suche auf den Metadaten handelt ist eine SQL konforme Umwandlung der Datentypen und Operatoren erforderlich.

Der *QueryParser* akzeptiert vier verschiedene Datentypen und acht Vergleichsoperatoren, wobei jedoch nicht jeder Operator mit jedem Datentyp verwendbar ist. Abbildung 3 gibt einen Überblick der möglichen Kombinationen.

Tabelle 3 Kombinationsmöglichkeiten von Datentypen und Operatoren

Datentyp	Zulässige Operatoren	Anmerkungen
AttributeString	LeafContains LeafContainsLeft LeafContainsRight LeafEqual	Wildcard am Textanfang und -ende Wildcard am Textende Wildcard am Textanfang
AttributeBoolean	LeafEqual	
AttributeInteger	LeafEqual LeafGreater LeafEvenGreater LeafLess LeafEvenLess	
AttributeDate	LeafEqual LeafGreater LeafEvenGreater LeafLess LeafEvenLess	

Je nach Kombination erzeugt der *QueryParser* ein entsprechendes Blatt für der *LogicTree*. Wobei im Fall einer unzulässigen Kombination eine Fehlermeldung ausgelöst wird.

Bei den Blättern *LeafContains*, *LeafContainsRight* und *LeafContainsLeft* ist zu beachten, daß der Text um entsprechende Wildcards ergänzt werden muss. Die Umwandlung der Werte in SQL konforme Formate wird von den Blättern des *LogicTree* durchgeführt. Weiterhin wird die Beachtung der Groß- und Kleinschreibung bei der Suche nach Strings deaktiviert wenn diese komplett klein geschrieben sind.

Die Suche auf den Metadaten ermöglicht es, die Erfassungsmethode zu berücksichtigen. Da dieses "*Origin*" genannte Zusatzmerkmal wie alle anderen Metadaten behandelt wird, genügt es dem *LogicTree* eine entsprechende Einschränkung hinzuzufügen (siehe dazu Codefragment 3).

Codefragment 3 QueryParser, Berücksichtigung der Erfassungsmethode

```
/* is an origin given ? */
if( !leaf.getAttributeOrigin().equalsIgnoreCase( "" ) ) {
            LogicNode origin = new LogicStringConstraint( leaf.getAttributeName()+".origin",
                                    "=", leaf.getAttributeOrigin() ) );
            result = new LogicAnd( result, origin );
}
```

Kapitel 4 Logische Transformation

Die logische Transformation stellt verschiedene Methoden zur Verfügung um die Interndarstellung in eine Normalform zubringen. Im Wesentlichen sind dies die konjunktive (KNF) und die disjunktive (DNF) Normalform. Je nach Struktur der Anfrage kann eine solche Transformation die spätere Optimierung begünstigen. Die im *LogicTranformer* verwendeten Algorithmen sind alle in [SCH95] zu finden.

4.1 Normalisierung

Die Normalisierung basiert auf drei Regeln:

1. Ersetze jedes Vorkommen von `LogicNot(LogicNot(a))` durch `a`.
2. Ersetze jedes Vorkommen von `LogicNot(LogicAnd(a,b))` durch `LogicOr(LogicNot(a),LogicNot(b))`.
3. Ersetze jedes Vorkommen von `LogicNot(LogicOr(a,b))` durch `LogicAnd(LogicNot(a),LogicNot(b))`.

Das führt zu folgendem Algorithmus:

Codefragment 4 LogicTransformer, Normalisierung

```
LogicNode normalize( LogicNode tree ) {
       if( tree instanceof LogicOperator ) {
              /* logic operator */
              if( tree instanceof LogicNot ) {
                     /* negation */
                     LogicNode neg = ((LogicNot)tree).getRight();
                     if( neg instanceof LogicOperator ) {
                            /* negatived logic operator */
                            LogicNode left = normalize(new
                                          LogicNot(((LogicOperator)neg).getLeft()));
                            LogicNode right = normalize(new
                                          LogicNot(((LogicOperator)neg).getRight()));
                            if( neg instanceof LogicNot ) {
                                   /* NOT NOT A => A */
                                   return right;
                            } else if( neg instanceof LogicAnd ) {
                                   /* NOT AND ( A AND B ) => NOT A OR NOT B */
                                   return new LogicOr(  left, right );
                            } else if( neg instanceof LogicOr ) {
                                   /* NOT OR ( A AND B ) => NOT A AND NOT B */
                                   return new LogicAnd( left, right );
                            }
                     } else if( neg instanceof LogicVariable ) {
                            /* negatived logic variable */
                            return ((LogicVariable)neg).negation();
                     }
              } else {
                     LogicNode left = normalize( ((LogicOperator)tree).getLeft() );
                     LogicNode right = normalize( ((LogicOperator)tree).getRight() );
                     if( tree instanceof LogicAnd ) {
                            /*  A AND B */
                            return new LogicAnd( left, right );
                     } else if( tree instanceof LogicOr) {
                            /*  A OR B */
                            return new LogicOr( left, right );
                     }
              }
       } else if( tree instanceof LogicVariable ) {
              /* logic variable */
              return tree;
       }
}
```

4.2 Transformation in KNF und DNF

Die Transformation in die konjunktive Normalform ist ebenfalls rekursiv formuliert.

1. Ersetze jedes Vorkommen von `LogicOr(a, LogicAnd(b,c))` durch
   `LogicAnd(LogicOr(a,b),LogicOr(a,c)).`
2. Ersetze jedes Vorkommen von `LogicOr(LogicAnd(a,b),c)` durch
   `LogicAnd(LogicOr(a,c),LogicOr(b,c)).`

Für die disjunktive Normalform gillt analog :

1. Ersetze jedes Vorkommen von `LogicAnd(a, LogicOr(b,c))` durch
   `LogicOr(LogicAnd(a,b),LogicAnd(a,c)).`
2. Ersetze jedes Vorkommen von `LogicAnd(LogicOr(a,b),c)` durch
   `LogicOr(LogicAnd(a,c),LogicAnd(b,c)).`

Kapitel 5 Erstellen eines Anfrageplans

Der Anfrageplan ist eine Repräsentation der späteren SQL-Anfrage, die eine einfache Manipulation und Analyse zulässt. Das Erzeugen des Anfrageplan ist einer der komplexesten Schritte der Anfrageverarbeitung. In dieser Phase findet die Zuordnung von Metadaten Attributen und Datenbanktabellen, sowie die Vorbereitung, Ausführung und Aufbereitung der Volltextsuche statt.

Die Attribute der erfassten Materialien werden in einem relationalen Datenbanksystem gespeichert, wobei die einzelnen Merkmale in verschiedenen Tabellen gespeichert und über Beziehungen den Dokumenten zugeordnet werden. Jedes erfasste Dokument wird von einem Datensatz der Tabelle *learningresource* repräsentiert. Die zu erzeugende SQL-Anfrage soll nun die Schlüssel der Datensätze liefern, deren Merkmale die vom *LogicTree* beschriebenen Eigenschaften erfüllen.

5.1 Struktur der SQL-Anfrage

Da die Merkmale der Dokumente in einer relationalen Datenbank mit mehreren Tabellen gespeichert werden, ist es in vielen Fällen nötig ein Kreuzprodukt (*Join*) der beteiligten Tabellen zu erstellen und die interessanten Zeilen mit einer entsprechenden Verknüpfungsvorschrift (*Join-Bedingung*) auszuwählen. Eine Liste der Tabellen und Bedingungen liefert das oben erwähnte Modul *MetaModelDAO*.

Bei der Formulierung der Anfrage ist jedoch zu beachten, dass eine einfach Verknüpfung aller beteiligten Tabellen in einem einzigen *Join* nur in Ausnahmefällen zum gewünschten Ergebnis führt. Eine solche Anfrage würde nur Datensätze liefern für die in jeder beteiligten Tabelle mindestens ein Datensatz vorhanden ist, der die Joinbedingung erfüllt. Da jedoch nicht für alle Dokumente jedes mögliche Merkmal erfasst wird können Datensätze aus dem Ergebnis herausfallen, obwohl sie den vom Benutzer festgelegten Kriterien genügen (siehe Beispiel 1).

Beispiel 1 Anfrage mit verschiedenen Attributen

Gegeben seien drei Tabellen:

dokument

id	beschreibung
1	Dokument 1
2	Dokument 2
3	Dokument 3

autor

id	idref	name
1	1	Schmitt
2	2	Maier

thema

id	idref	bezeichnung
1	2	Alpha
2	3	Beta

Gesucht sind alle Dokumente die vom Autor "Schmitt" verfasst wurden oder das Thema "Beta" haben. Diese Bedingungen treffen für Dokument 1 und Dokument 3 zu. Eine einfache SQL-Anfrage wie die folgende liefert jedoch als Ergebnis eine leere Menge zurück.

```
SELECT *
FROM   dokument, autor, thema
WHERE  dokument.id=autor.idref                                          ; Bedingung 1
       AND dokument.id=thema.idref                                      ; Bedingung 2
       AND ( autor.name = 'Schmitt' OR thema.bezeichnung = 'Beta' )     ; Bedingung 3
```

Betrachtet man das Kreuzprodukt der drei Tabellen sieht man auch warum.

dokument		*autor*			*thema*			
id	beschreibung	id	idref	name	id	idref	bezeichnung	
1	Dokument 1	1	1	Schmitt	1	2	Alpha	Bedingung 1 und Bedingung 3 erfüllt
1	Dokument 1	1	1	Schmitt	2	3	Beta	Bedingung 1 und Bedingung 3 erfüllt
1	Dokument 1	2	2	Maier	1	2	Alpha	
1	Dokument 1	2	2	Maier	2	3	Beta	
2	Dokument 2	1	1	Schmitt	1	2	Alpha	
2	Dokument 2	1	1	Schmitt	2	3	Beta	
2	Dokument 2	2	2	Maier	1	2	Alpha	
2	Dokument 2	2	2	Maier	2	3	Beta	
3	Dokument 3	1	1	Schmitt	1	2	Alpha	
3	Dokument 3	1	1	Schmitt	2	3	Beta	
3	Dokument 3	2	2	Maier	1	2	Alpha	
3	Dokument 3	2	2	Maier	2	3	Beta	Bedingung 2 und Bedingung 3 erfüllt

Es gibt keine Zeile, die alle drei Bedingungen gleichzeitig erfüllt. Um ein korrektes Ergebnis zu erhalten, muss für jedes geforderte Prädikat eine eigene Anfrage formulieren und diese dann kombinieren. Die entsprechende SQL-Anfrage lautet dann wie folgt:

```
SELECT *
FROM   dokument
WHERE  dokument.id IN ( SELECT dokument.id
                        FROM   dokument, autor
                        WHERE  dokument.id = autor.idref
       AND autor.name = 'Schmitt' )
OR     dokument.id IN ( SELECT dokument.id
                        FROM   dokument, thema
                        WHERE  dokument.id = thema.idref
                        AND    thema.bezeichnung = 'Beta' )
```

Diese Anfrage liefert dann das gewünschte Ergebnis.

Es wäre zwar ebenso möglich diese Anfrage mit Hilfe von so genannten "outer-joins" zu formulieren, doch diese Vorgehensweise wäre nur mit sehr großem Aufwand generisch zu realisieren.

Einen weiteren Stolperstein bilden die 1:n oder m:n Beziehungen in der Datenbank. Für bestimmte Merkmale kann ein Dokument Mehrfacheinträge besitzen. Somit ist es auch zulässig nach Dokumenten zu suchen die mehrere verschiedene Einträge für ein Merkmal haben. Eine einfach Verknüpfung mit *AND* würde aber zu einer leeren Menge führen, da in jeder Zeile der Merkmalstabelle nur ein Eintrag steht der eventuell nicht alle geforderten Bedingungen erfüllt und somit aus dem Ergebnis herausfällt. In Beispiel 2 ist dies nochmals verdeutlicht.

Beispiel 2 Anfrage von gleichen Attributen

Gegeben seien zwei Tabellen:

dokument

id	beschreibung
1	Dokument 1
2	Dokument 2
3	Dokument 3

autor

id	idref	name
1	1	Schmitt
2	2	Maier
3	2	Koch
4	3	Koch

Gesucht sind alle Dokument die von den Autoren Maier und Koch stammen. Diese Bedingungen treffen für Dokument 2 zu. Eine naive SQL-Anfrage wie die folgende liefert jedoch als Ergebnis eine leere Menge zurück.

```
SELECT *
FROM  dokument, autor
WHERE dokument.id=autor.idref     ; Bedingung 1
      AND autor.name = 'Maier'    ; Bedingung 2
      AND autor.name = 'Koch'     ; Bedingung 3
```

Betrachtet man das Kreuzprodukt, das dieser Anfrage zugrunde liegt, sieht man wieder sofort warum.

dokument		*autor*			
id	beschreibung	id	idref	name	
1	Dokument 1	1	1	Schmitt	
1	Dokument 1	2	2	Maier	
1	Dokument 1	3	2	Koch	
1	Dokument 1	4	3	Koch	
2	Dokument 2	1	1	Schmitt	
2	Dokument 2	2	2	Maier	Bedingung 1 und Bedingung 2 erfüllt
2	Dokument 2	3	2	Koch	Bedingung 1 und Bedingung 3 erfüllt
2	Dokument 2	4	3	Koch	
3	Dokument 3	1	1	Schmitt	
3	Dokument 3	2	2	Maier	
3	Dokument 3	3	2	Koch	
3	Dokument 3	4	3	Koch	

Es ist unmöglich, dass die Spalte autor.name gleichzeitig der Wert 'Maier' und 'Koch' hat.

Um das gewünschte Ergebnis zu erhalten gibt es nun zwei Möglichkeiten: Zum einen die Formulierung mit Unteranfragen und zum anderen ein Self-Join. Die entsprechenden SQL-Anfragen lauten dann wie folgt:

- mit Unteranfrage:

```
SELECT *
FROM  dokument
WHERE dokument.id IN ( SELECT dokument.id
                       FROM   dokument, autor
                       WHERE  dokument.id = autor.idref
                              AND autor.name = 'Maier' )
AND   dokument.id IN ( SELECT dokument.id
                              FROM dokument, autor
                              WHERE dokument.id = autor.idref
                              AND autor.name = 'Koch' )
```

- mit Self-Join:

```
SELECT *
FROM  dokument, autor a1, autor a2
WHERE dokument.id = a1.idref
      AND a1.name = 'Maier'
      AND dokument.id = a2.idref
      AND a1.name = 'Koch'
```

Auf ähnliche Weise müssen Negationen von Attributen behandelt werden. Durch die bei n-fach Beziehungen muss darauf geachtet werden, dass auszuschließenden Dokumente nicht durch eine weitere Beziehung hinzugenommen werden. Beispiel 3 veranschaulicht diese Problematik.

Beispiel 3 Negation von Attributen

Gegeben seien die zwei Tabellen aus Beispiel 2 Gesucht sind alle Dokument an denen der Autor Koch nicht mitgearbeitet hat. Diese Bedingungen treffen für Dokument 1 zu. Eine einfache SQL-Anfrage wie die folgende liefert jedoch als Ergebnis jedoch Dokument 1 und Dokument 2.

```
SELECT *
FROM  dokument, autor
WHERE dokument.id=autor.idref              ; Bedingung 1
      AND autor.name != 'Koch'             ; Bedingung 2
```

Betrachtet man das Kreuzprodukt der zwei beteiligten Tabellen, wird deutlich warum Dokument 2 auch zu der Ergebnismenge gehört.

dokument		*autor*			
id	**beschreibung**	**id**	**idref**	**name**	
1	**Dokument 1**	**1**	**1**	**Schmitt**	**Bedingung 1 und Bedingung 2 erfüllt**
1	**Dokument 1**	**2**	**2**	**Maier**	
1	**Dokument 1**	**3**	**2**	**Koch**	
1	**Dokument 1**	**4**	**3**	**Koch**	
2	**Dokument 2**	**1**	**1**	**Schmitt**	
2	**Dokument 2**	**2**	**2**	**Maier**	**Bedingung 1 und Bedingung 2 erfüllt**
2	**Dokument 2**	**3**	**2**	**Koch**	
2	**Dokument 2**	**4**	**3**	**Koch**	
3	**Dokument 3**	**1**	**1**	**Schmitt**	
3	**Dokument 3**	**2**	**2**	**Maier**	
3	**Dokument 3**	**3**	**2**	**Koch**	
3	**Dokument 3**	**4**	**3**	**Koch**	

Es existiert eine Zeile für Dokument 2 mit dem Autor Maier und erfüllter Joinbedingung. Das Ergebnis ist also für obige SQL-Anfrage vollkommen korrekt. Um nun das gewünschte Resultat zu erhalten muss man wieder mit einer Unteranfrage arbeiten und die Negation aus der Unteranfrage herausnehmen, um das korrekte Ergebnis zu erhalten.

```
SELECT *
FROM  dokument
WHERE dokument.id NOT IN ( SELECT dokument.id
                           FROM   dokument, autor
                           WHERE  dokument.id = autor.idref
                                  AND autor.name = 'Koch' )
```

Diese Anfrage liefert das gewünschte Ergebnis.

Die obigen Überlegungen führen zu dem Ergebnis, dass um ein der Suchanfrage entsprechendes Resultat zu erhalten, für jedes Prädikat eine eigene Unteranfrage gemacht und diese dann der Anfrage entsprechend kombiniert werden muss. Dies ist sicher nicht optimal, aber, da die Anfrage generisch sein soll und die Struktur der Daten nicht weiter bekannt ist, unumgänglich. Unter bestimmten Voraussetzungen ist es aber möglich die einzelnen Anfrage zu gruppieren und zusammenzufassen. Diese Möglichkeit wird in einen späteren Kapitel erörtert.

5.2 Aufbau des Anfrageplans

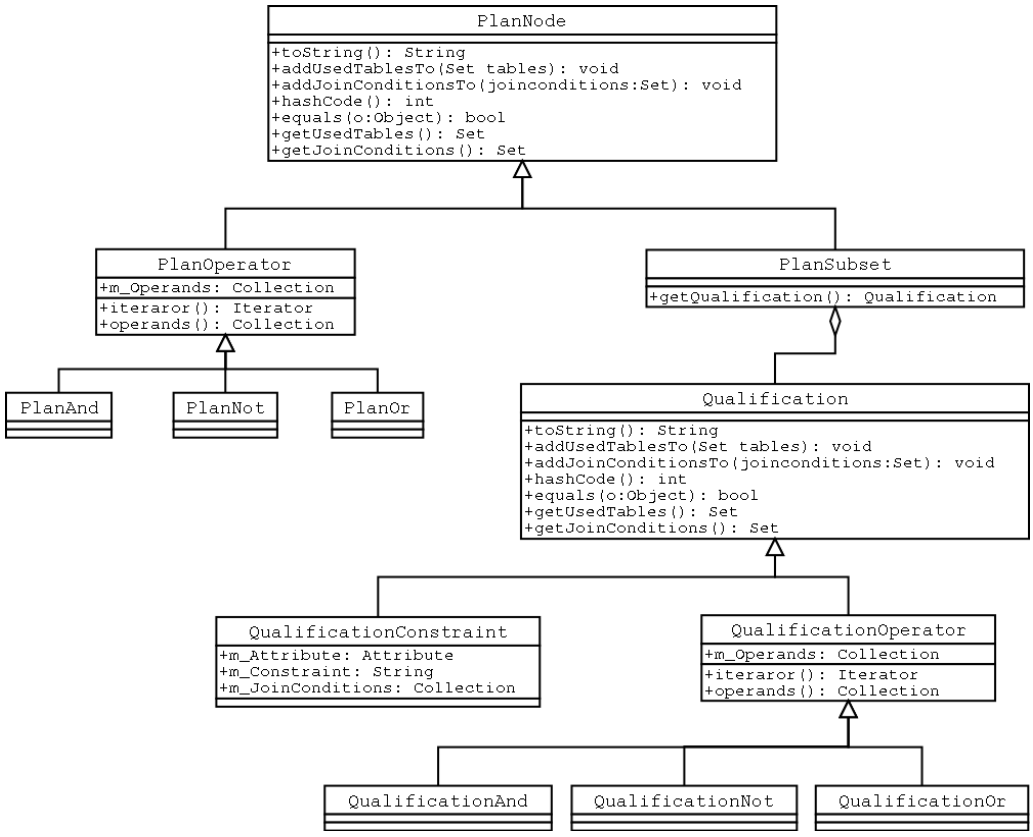
Der Anfrageplan (*QueryPlan*) enthält alle notwendigen Informationen um die Anfrage auszuführen. Dies sind im wesentlichen die Struktur der SQL-Anfrage und eine Übersicht der benötigten temporären Tabellen der Volltextsuche. Diese Übersicht ist einfach als *java.util.map* realisiert. Die Schlüssel dieser Zuordnungstabelle sind die jeweiligen Suchtexte deren Resultate in den betreffenden Tabellen zu finden sind. Die SQL-Anfrage wird als Zerlegungsbaum formuliert. Da die Projektion der SQL-Anfrage konstant ist, werden nur die Restriktionen als M-Baum, den so genannten *PlanTree,* gespeichert. Die Operatoren werden von den Knoten des Baumes repräsentiert und die einzelnen Prädikate von den Blättern. Für den *PlanTree* stehen wie für die Interndarstellung die logischen Operatoren AND, OR und NOT zur Verfügung. Allerdings können diese, bis auf den NOT Operator, nicht nur zwei Operanden haben wie beim *LogicTree,* sondern beliebig viele. Dies ermöglicht es auf recht einfache Weise identische Unteranfragen zu entfernen und die gesamte Struktur flacher zu gestalten.

Die Blätter des *PlanTree* entsprechen den im vorangegangenen Abschnitt erwähnten Unteranfragen, die für jedes Prädikat formuliert werden müssen. Die Qualifikationen dieser Anfragen werden wiederum als eigener Zerlegungsbaum (*QualificationTree*) gespeichert. Jedes Attribut wird als ein *QualificationConstraint*-Objekt in den Baum eingefügt.

Beide Baumstrukturen stellen Methoden zur Verfügung, die es sehr einfach erlauben eine Liste der verwendeten Tabellen und Joinbedingungen, sowie eine SQL-konforme Darstellung der Anfragen zu erhalten. Das UML-Diagramm in zeigt das Schema nochmals grafisch.

Die Aufteilung in *PlanTree* und *QualificationTree* soll im Quellcode die zwei Ebenen der Anfrage wiederspiegeln. Der *QualificationTree* formuliert eine Anfrage um eine Menge von Dokument Schlüsseln zu erhalten, wohingegen die vom *PlanTree* beschriebene Anfrage diese dann verknüpft.

Abbildung 6 UML-Diagramm PlanTree und QualificationTree



5.3 Erzeugung des Anfrageplans (PlanGenerator)

Der *PlanGenerator* hat, wie schon erwähnt, die Aufgabe aus der internen Darstellung eine SQL-Anfrage zu formulieren. Im wesentlichen geschieht dies durch den Aufbau eines entsprechenden Zerlegungsbaum, anhand dessen alle weiter benötigen Daten bestimmt werden können.

Der *PlanGenerator* arbeitet bei der Erstellung des Zerlegungsbaums ähnlich wie der *QueryParser* mit einem rekursiven Algorithmus. Da es sich hierbei jedoch nicht mehr um einen binären, sondern um einen M-Baum handelt, sind zusätzliche Maßnahmen erforderlich um möglichst ein Entarten des *PlanTree* zu vermeiden. Hier ist anzumerken, dass der *PlanGenerator* keinerlei logische Transformationen vornimmt. Dieses sollte in vorangegangen Arbeitsschritten durchgeführt worden sein.

Der *PlanGenerator* kategorisiert einen Knoten des *LogicTree* in zwei Gruppen: Operatoren und Prädikate. Jede dieser beiden Kategorien wird gesondert behandelt. Bei den Operatoren kommt es darauf an, ob diese einen (NOT) oder zwei (AND, OR) Operanden haben. Unäre Operatoren wie *LogicNot* werden unverändert übernommen. Es wird ein *PlanNot*-Operator Objekt erzeugt, das den aus dem Operanden des *LogicNot*-Knotens erzeugten Zerlegungsbaum als Operand erhält. Codefragment 5 zeigt das Vorgehen für *LogicNot*.

Codefragment 5 PlanGenerator, unäre Operatoren am Beispiel LogicNot

```
if( operator instanceof LogicNot ) {
       /* not operator */
   PlanNode rn = parseNode( operator.getRight() );
       return new PlanNot( rn );
}
```

Bei binären Operatoren, wie *LogicAnd* und *LogicOr*, gestaltet sich die Vorgehensweise etwas komplexer. Wie bei unären Operator wird ein entsprechender Knoten in den *PlanTree* eingefügt und die Operanden umgewandelt. Um jedoch ein Entarten des Zerlegungsbaums zu verhindern, muss überprüft werden, welchen Knotentyp die Operanden besitzen. Besitzt ein Operand den gleichen Typ wie der Knoten selbst, so werden dessen Operanden dem übergeordneten Operator Objekt hinzugefügt. Codefragment 6 verdeutlicht das Verfahren. Dadurch wird die Struktur des Zerlegungsbaums möglichst flach gehalten, was für eine spätere Optimierung von Vorteil ist. Quasi als Nebeneffekt können dabei auch einige, doppelt vorhandene Prädikate entfernt werden, die bei dieser Vorgehensweise durch den gleichen Knoten verknüpft werden. Dies kann ganz einfach durch Verwendung eines "java.util.set" Objekts als Operanden Container erreicht werden.

Codefragment 6 PlanGenerator, binäre Operatoren am Beispiel LogicNot

```
if( operator instanceof LogicAnd ) {
       /* and operator */

       Collection operands = new HashSet();

       PlanNode ln = parseNode( operator.getLeft() );
       PlanNode rn = parseNode( operator.getRight() );

       /* left operand */
       if( ln instanceof PlanAnd ) {
              for( Iterator i = ((PlanAnd)ln).iterator(); i.hasNext(); ) {
              operands.add( i.next() );
       }
   } else {
              operands.add( ln );
   }

       /* right operand */
       if( rn instanceof PlanAnd ) {
              for( Iterator i = ((PlanAnd)rn).iterator(); i.hasNext(); ) {
                     operands.add( i.next() );
       }
       } else {
              operands.add( rn );
       }

       return new PlanAnd( operands );
}
```

Die zweite Gruppe von Knoten, die Prädikate, enthält zur Zeit drei mögliche Typen:

- *LogicConstraint:* Einschränkung eines Attributs auf einen bestimmten Wert oder Bereich.
- *LogicExistence:* Prüfung auf Existenz eines bestimmten Attributs.
- *LogicFulltext:* Der Sonderfall Volltextsuche.

Für jeden dieser drei Typen ist nun eine entsprechenden Unteranfrage in Form eines *PlanSubset*-Objekts hinzuzufügen. Bevor ein solches Objekt erzeugt werden kann, wird zuerst eine Qualifikation für die Unteranfrage benötigt. Der *PlanGenerator* erzeugt nur einfach Qualifikationen, die aus einem einzelnen *QualificationConstraint*-Objekt bestehen. Erst bei der Optimierung werden diese zu komplexere Ausdrücken kombiniert.

Ein *QualificationConstraint*-Objekt besteht, wie in Abbildung 6 ersichtlich, aus einem Datenbankattribut, einer Einschränkung und einem Container mit eventuell benötigten Tabellenverknüpfungen. *LogicConstraint* und *LogicExistence*-Knoten beziehen sich beide auf Metadaten. In diesem Fall liefert das *MetaModelDAO* für den gegebenen XML-Bezeichner ein entsprechendes Datenbankattribut, sowie die Verknüpfungen.

Die Einschränkung wird bei *LogicConstraint* Knoten von einer Methode desselben generiert und als String zur Verfügung gestellt. Dieser String ist SQL-konform und kann ohne weitere Verarbeitung verwendet werden. Weiterhin wird noch eine Überprüfung der Datentypen vorgenommen, damit die in der Datenbank und bei der Anfrage verwendeten Typen zueinander passen. Wenn es sich um ein String-Attribut handelt, wird noch überprüft, inwiefern die Groß- und Kleinschreibung berücksichtigt werden soll, und gegebenenfalls eine entsprechende Aggregatfunktion eingefügt. Für *LogicExistence*-Knoten ist die Einschränkung immer einen Überprüfung auf "**NULL**".

Codefragment 7 PlanGenerator, LogicConstraint und LogicExistence

```
if( variable instanceof LogicConstraint ) {
	/* metadata */

	try {
		String sXMLPathname = ((LogicConstraint)variable).getXMLPathname();
		String constraint = ((LogicConstraint)variable).getConstraint();
		Attribute attribute = m_MetaModelDAO.getAttribute( sXMLPathname );

		/* type check */
		if( !m_MetaModelDAO.getJavaclassname( sXMLPathname ).equals(
			((LogicConstraint)variable).getValueType() ) ) {
			throw new PlanGeneratorException( "Type mismatch! Query \"" +
				((LogicConstraint)variable).getValueType() + "\" vs. MetaData \"" +
				m_MetaModelDAO.getJavaclassname( sXMLPathname ) + "\"." )
		}

		/* case sensitive ? */
		if( variable instanceof LogicStringConstraint
				&& !((LogicStringConstraint)variable).isCaseSensitive() ) {
			attribute = new AttributeFunction( "LOWER", attribute );
		}

		Qualification qualification = new QualificationConstraint( attribute,
					constraint,m_MetaModelDAO.getDependencies( sXMLPathname ) );
		return new PlanSubset( qualification );
	} catch ( MetaModelDAOException e ) {
		/* metadata access error */
		throw new PlanGeneratorException( "Metadata access error! Caught \""
				+ e.getClass().getName() + "\".", e );
	}

} else if( variable instanceof LogicExistence ) {
	/* metadata existence */

	try {
		String sXMLPathname = ((LogicExistence)variable).getXMLPathname();
		Attribute attribute = m_MetaModelDAO.getAttribute( sXMLPathname );

		Qualification qualification = new QualificationConstraint( attribute, "IS NOT
NULL",
					m_MetaModelDAO.getDependencies( sXMLPathname ) );
		return new PlanSubset( qualification );
	} catch ( MetaModelDAOException e ) {
		/* metadata access error */
		throw new PlanGeneratorException( "Metadata access error! Caught \""
				+ e.getClass().getName() + "\".", e );
	}
}
```

Handelt es sich bei dem Prädikat um eine Volltextsuche, so muss das *FulltextCacheDAO* benutzt werden um eine Tabelle mit dem Resultat der Volltextsuche zu erhalten. Die eigentliche Volltextsuche wird bei Bedarf für den *PlanGenerator* transparent ausgeführt. Die erhaltene Tabelle wird in der internen Zuordnungstabelle *m_FulltextMap* abgelegt und ein entsprechendes *QualificationConstraint*-Objekt für eine Unteranfrage erzeugt. Siehe dazu auch Codefragment 8.

Codefragment 8 PlanGenerator, LogicFulltext

```
if( variable instanceof LogicFulltext ) {
       /* fulltext */

       try {
              String sFulltextSearchString = ((LogicFulltext)variable).getFulltext();
              String sFulltextCacheTable;

              if( m_FulltextMap.containsKey(sFulltextSearchString) ) {
                     sFulltextCacheTable = (String)m_FulltextMap.get(sFulltextSearchString);
              } else {
                     sFulltextCacheTable = m_FulltextCacheDAO.getCacheTable(
                            sFulltextSearchString );
                     m_FulltextMap.put( sFulltextSearchString, sFulltextCacheTable );
              }

              Attribute attribute = new Attribute( sFulltextCacheTable, "fulltext" );
              Set dependencies = new HashSet();
              dependencies.add ( new Join( sFulltextCacheTable, "lridref",
                     "metabase.learningresource", "lrid" ) );
              Qualification qualification =
                            new QualificationConstraint( attribute, "IS NOT NULL",
                                   dependencies );
              return new PlanSubset( qualification );
              } catch ( FulltextCacheDAOException e ) {
                     /* metadata access error */
                     throw new PlanGeneratorException( "Fulltext error! Caught \""
                                   + e.getClass().getName() + "\".", e );
              }
}
```

Mit Hilfe des Zerlegungsbaums und seinen Funktionen lässt sich recht einfach eine SQL-Anfrage erzeugen, wie aus Codefragment 9 ersichtlich wird.

Codefragment 9 Erzeugen eine SQL-Anfrage aus dem PlanTree

```
SetToStringConverter sc = new SetToStringConverter();

String sqlQuery = "SELECT DISTINCT metabase.learningresource.lrid AS lridref FROM ";

Set tables = planTree.getUsedTables();
tables.add( "metabase.learningresource" );

sqlQuery += sc.convert( tables, ", " ) + " ";
sqlQuery += "WHERE ";

Set joins = planTree.getJoinConditions();
if( !joins.isEmpty() ) {
       sqlQuery += sc.convert( joins, " AND " ) + " AND ";
}
sqlQuery += m_PlanTree.toString();
```

Gesetzt dem Fall die Anfrage besteht nur aus einem einzige Prädikat, wird eine einfache Optimierung vorgenommen, indem das *PlanSubset*-Objekt durch ein *PlanConstraint* Objekt ersetzt wird, um die unnötige Anfrage einzusparen. Im nächsten Kapitel werden nun weit komplexere Methoden vorgestellt um die Anfrage weiter zu optimieren.

Kapitel 6 Optimierung des Anfrageplans

Das Ziel der Anfrageoptimierung besteht darin, den Anfrageplan in eine für die Ausführung möglichst günstige Form zu bringen. In diesem speziellen Fall, heißt dies Zusammenfassen von mehreren Unteranfragen zu einer. Dabei müssen jedoch wiederum die Besonderheiten des Schemas beachtet werden, die in vorangegangen Kapitel erläutert wurden.

Die Eigenschaften der Datenbank führen zu zwei Regeln für die Optimierung:

1. Fasse alle Disjunktionen (OR) zusammen welche die gleichen Tabellen betreffen.
2. Fasse alle Konjunktionen (AND) zusammen welche nicht die gleichen Attribute betreffen.

Da der *PlanOptimizer* keine logischen Umformungen vornimmt, kann es von Vorteil sein, die Anfrage schon im Hinblick auf die Optimierung zu formulieren oder vorher entsprechende logische Transformationen durchzuführen.

Negationen können nicht einfach zusammengefasst werden, da dies zu einem falschen Resultat führen kann, wie im vorangegangen Kapitel gezeigt wurde.

6.1 Optimierung von OR Verknüpfungen

Disjunktionen von Unteranfragen lassen sich zusammenführen, wenn die betreffenden Operanden die gleiche Menge von Tabellen verwenden. Würden sich die Mengen der benutzen Tabellen unterscheiden, so könnte es, wie in Beispiel 1 auf Seite 17, zum Wegfallen einiger potentieller Treffer führen, die eigentlich die Bedingungen der Anfrage erfüllen.

Der Algorithmus ist recht einfach gehalten und benutzt die in Java verfügbaren Standardcontainer um seine Aufgabe zu erfüllen. Zuerst werden die Operanden der Disjunktion selbst optimiert und anschließend ihrem Typ entsprechend sortiert. Die Qualifikationen von *PlanSubset* Objekten werden in eine Zuordnungstabelle von Typ *java.util.map* eingetragen, wobei die Menge der jeweils verwendeten Tabellen als Schlüssel dient. Alle anderen Operanden werden in einer Liste gesammelt. Nun wird für jeden Schlüssel der Zuordnungstabelle ein neues *PlanSubset*-Objekt erzeugt, indem die einzelnen Qualifikationen mit einem *QualificationOr*-Knoten verknüpft werden. Diese neuen *PlanSubset*-Objekt werden der Liste mit den anderen Operanden hinzugefügt. Aus dieser Liste wird dann am Ende, je nach Anzahl der Operanden, ein neuer *PlanOr* Knoten erzeugt oder der einzelne Operand zurückgeliefert.

Codefragment 10 Zusammenfassung von Disjunktionen

```
if( node instanceof PlanOr ) {
	/* or */

	PlanOr or = (PlanOr) node;
	Collection operands = new LinkedList();
	Map subsets = new HashMap();

	for( Iterator i = or.iterator(); i.hasNext(); ) {
		PlanNode operand = optimizeNode( (PlanNode)i.next() );
		if( operand instanceof PlanSubset ) {
			Qualification qualification = ((PlanSubset)operand).getQualification();
			Set tables = qualification.getUsedTables();
			if( subsets.containsKey( tables ) ) {
				((Collection)subsets.get( tables )).add( qualification );
			} else {
				Collection s = new HashSet();
```

```
                        s.add( qualification );
                        subsets.put( tables, s );
                    }
                } else {
                    operands.add( operand );
                }
            }

            for( Iterator i = subsets.values().iterator(); i.hasNext(); ) {
                Collection qualifications = (Collection)i.next();
                if( qualifications.size() == 1 ) {
                    operands.add( new PlanSubset(
(Qualification)qualifications.iterator().next() ) );
                } else {
                    operands.add( new PlanSubset( new QualificationOr( qualifications  ) ) );
                }
            }

            if( operands.size() == 1 ) {
                return (PlanNode) operands.iterator().next();
            } else {
                return new PlanOr( operands );
            }
        };
```

6.2 Optimierung von AND Verknüpfungen

Konjugierte Unteranfragen lassen sich nicht so einfach zusammenfassen wie disjunkte. Zwar gibt es hierbei vielmehr Möglichkeiten, doch der Algorithmus ist etwas komplexer als bei Vereinigungen. Genauso wie bei der Optimierung von OR Verknüpfungen müssen auch hier gewisse Bedingungen eingehalten werden, um das Ergebnis nicht zu verfälschen. In Abschnitt 5.1 wurden die zu beachtenden Einschränkungen ausführlich erläutert. Analog zum Algorithmus für Disjunktionen werden die Operanden der Disjunktion, ihrem Typ entsprechend sortiert. *PlanSubset*-Objekt werde in einer, alle sonstigen in einer anderen Liste gesammelt. Die Qualifikationen der *PlanSubset* Objekte werden nun in Gruppen angeordnet, in denen pro Gruppe keine Attribut mehr als einmal vertreten ist. Aus diesen Gruppen werden dann neue *PlanSubset*-Objekte erstellt und der Operandenliste hinzugefügt. Am besten wird dies durch Codefragment 11 deutlich. Aus dieser Liste wird dann am Ende, je nach Anzahl der Operanden, ein neuer *PlanAnd* Knoten erzeugt oder der einzelne Operand zurückgeliefert.

Codefragment 11 Zusammenfassung von Konjunktionen

```
if( node instanceof PlanAnd ) {
    /* and */

    PlanAnd and = (PlanAnd) node;
    Collection operands = new LinkedList();
    List subsets = new LinkedList();

    for( Iterator i = and.iterator(); i.hasNext(); ) {
        PlanNode operand = optimizeNode( (PlanNode)i.next() );
        if( operand instanceof PlanSubset ) {
            subsets.add( operand );
        } else {
            operands.add( operand );
        }
    }

    while( !subsets.isEmpty() ) {
        ListIterator i = subsets.listIterator();

        PlanSubset subset = (PlanSubset)i.next();
        List qualifications = new LinkedList();
        qualifications.add( subset.getQualification() );
        Set attributes = subset.getQualification().getAttributes();
        i.remove();

        while( i.hasNext() ) {
            subset = (PlanSubset)i.next();
```

```
                        Set intersection = new HashSet();
                        intersect( intersection, subset.getQualification().getAttributes(),
                                attributes );
                        if( intersection.isEmpty() ) {
                               qualifications.add( subset.getQualification() );
                               attributes.addAll( subset.getQualification().getAttributes() );
                               i.remove();
                        }
                 }

                 if( qualifications.size() == 1 ) {
                        operands.add( new PlanSubset(
                                       (Qualification)qualifications.iterator().next() ) );
                 } else {
                        operands.add( new PlanSubset( new QualificationAnd( qualifications  ) ) );
                 }
          }

          if( operands.size() == 1 ) {
                 return (PlanNode) operands.iterator().next();
          } else {
                 return new PlanAnd( operands );
          }
   }
```

Kapitel 7 Ausführung der Anfrage (QueryProcessor)

Die letzte Phase der Anfrageverarbeitung besteht aus der Ausführung der Anfrage. Dabei wird aus dem Anfrageplan eine SQL-Anfrage generiert, ausgeführt und das Ergebnis in einer temporären Tabelle gespeichert. Diese Tabelle wird registriert und ein eindeutiger Schlüssel zur Verfügung gestellt, der externen Modulen des Systems, wie der Ergebnisverwaltung, den Zugriff auf das Resultat der Suche ermöglicht. Überdies werden den Datensätzen, sofern vorhanden, Volltextauszüge und eine fortlaufende Nummer, abhängig von der Volltextsuche, zugeordnet um die Ergebnisse entsprechend gewichten zu können.

7.1 Finden einer gültigen Resultset Id

Die erzeugten Ergebnis-Tabellen wird in der Tabelle *resultset* erfasst um den Zugriff an einer zentralen Stelle zu koordinieren. Jedem Ergebnis wird dazu ein eindeutiger Schlüssel in Form eines Integer Wertes zugeordnet. Ferner werden zusätzliche Informationen, wie der Volltextsuchtext, ein Cursor und die aktuelle Sortierung, bereitgehalten.

Viele Datenbanken bieten zwar entsprechende Möglichkeiten einen Schlüssel automatisch zu erzeugen, doch darauf wurde bewusst verzichtet, um möglichst nicht von einer speziellen Datenbankplattform abhängig zu sein und ohne großen Aufwand das zugrunde liegende Datenbanksystem austauschen zu können. Vielmehr wird der Schlüssel durch ein einfaches iteratives Verfahren bestimmt.

Die Spalte *rsid* der Tabelle *resultset* erhält die Einschränkung *UNIQUE*, es sind also keine zwei identischen Werte erlaubt.

Der neue Schlüssel wird dann folgendermaßen bestimmt:

1. Bestimme den maximalen Schlüssel der *resultset* Tabelle und addiere 1 hinzu.
2. Füge einen Datensatz mit dem ermittelten Schlüssel in die Tabelle *resultset* ein.
3. Tritt ein Fehler auf, dann gehe wieder zu Schritt 1.

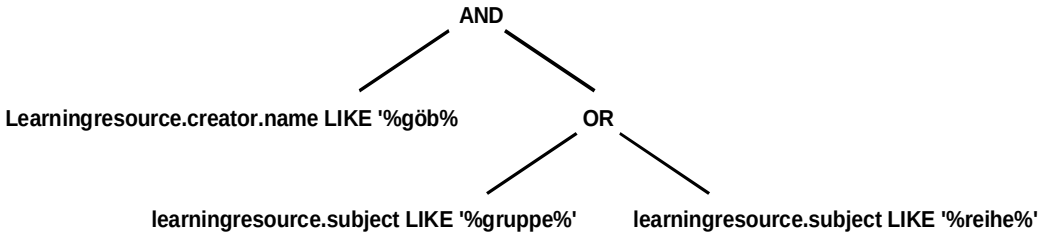
Der so gefundene Schlüssel ist durch die *UNIQUE* Bedingung auf jeden Fall eindeutig. Die verwendete Datenbank muss jedoch Constraints unterstützen, sonst ist dieses Verfahren nicht anwendbar. Der Programmcode ist jedoch unabhängig vom Datenbanksystem.

7.2 Query Ausführen

Nachdem nun ein eindeutiger Schlüssel gefunden wurde, kann eine temporäre Tabelle erstellt und die Anfrage an die Datenbank gestellt werden. Die Bezeichnung der temporären Tabelle setzt sich aus einen Präfix und dem Schlüssel zusammen (z.B. *resultset42*). Diese Kennung wird später in der Verwaltungstabelle *resultset* gespeichert. Die eigentliche Suchanfrage ist eine INSERT INTO SELECT Anweisung, wobei die SELECT Anweisung einfach mit Hilfe einer Methode der *PlanTree*-Klasse erstellt wird. In Beispiel 4 ist die SQL-Anfrage für einen einfachen *QueryTree* dargestellt.

Beispiel 4 Sql-Anfrage

Gegeben sei folgender *Query-Tree*:



Aus diesem wird dann mit Hilfe des *Plangenerator*-Moduls folgende SQL-Anfrage erstellt:

```
SELECT DISTINCT metabase.learningresource.lrid AS lridref
FROM metabase.learningresource
WHERE  (( metabase.learningresource.lrid IN ( SELECT metabase.learningresource.lrid
          FROM metabase.creatorofresource, metabase.learningresource, metabase.name
          WHERE ( LOWER(metabase.name.content) LIKE '%göb%' )
             AND metabase.creatorofresource.creatoridref=metabase.name.nameid
             ANDmetabase.creatorofresource.resourceidref=metabase.learningresource.lrid )
       )
       AND (( metabase.learningresource.lrid IN ( SELECT metabase.learningresource.lrid
          FROM metabase.subject_value, metabase.learningresource, metabase.subject
          WHERE ( LOWER(metabase.subject_value.value) LIKE '%gruppe%' )
             AND metabase.subject.resourceidref=metabase.learningresource.lrid
             AND metabase.subject.subjectidref=metabase.subject_value.vid )
       )
       OR ( metabase.learningresource.lrid IN ( SELECT metabase.learningresource.lrid
          FROM metabase.subject_value, metabase.learningresource, metabase.subject
          WHERE metabase.subject.resourceidref=metabase.learningresource.lrid
             AND ( LOWER(metabase.subject_value.value) LIKE '%reihe%' )
             AND metabase.subject.subjectidref=metabase.subject_value.vid )
       )
    )
)
```

Diese Anfrage einen recht einfachen *QueryTree* schon extrem komplex. Das kommt daher, das für jedes Merkmal eine eigene Unteranfrage generiert wurde. Mit dem *PlanOptimizer*-Modul kann diese Anfrage noch vereinfacht werden, was man am nachfolgenden SQL-Code sofort erkennen kann.

```
SELECT DISTINCT metabase.learningresource.lrid AS lridref
FROM metabase.creatorofresource, metabase.subject_value, metabase.subject,
   metabase.learningresource, metabase.name
WHERE metabase.creatorofresource.creatoridref=metabase.name.nameid
    AND metabase.subject.resourceidref=metabase.learningresource.lrid
    AND metabase.creatorofresource.resourceidref=metabase.learningresource.lrid
    AND metabase.subject.subjectidref=metabase.subject_value.vid
    AND ( ( LOWER(metabase.name.content) LIKE '%göb%' )
       AND ( ( LOWER(metabase.subject_value.value) LIKE '%gruppe%' )
          OR ( LOWER(metabase.subject_value.value) LIKE '%reihe%' )
       )
    )
```

Die beiden SQL-Anfragen liefern die gewünschten Einträge aus der Datenbank.

7.3 Ranking und Volltextauszug bestimmen

Um eine Gewichtung und entsprechende Sortierung der Ergebnisse zu ermöglichen, erhält jeder Datensatz eine eindeutige Nummer. Da das Resultat der Volltextsuche schon eine Rangfolge zur Verfügung stellt, wird diese für die entsprechenden Dokumente direkt übernommen. Den übrigen wird einfach eine fortlaufende Nummer zugeordnet. Bei dieser Zuweisung werden den Datensätzen auch eventuell vorhandenen Volltextauszüge beigefügt. Am Ende wird der im ersten Schritt erzeugte Datensatz der *resultset* Tabelle noch um die Anzahl der gefundenen Dokumente ergänzt.

Kapitel 8 Zusammenfassung

Das Ziel dieser Projektarbeit war zum einen das Aufzeigen der Problematik der Anfrageverarbeitung des Meta-Akad System und zum anderen die Entwicklung eines einfachen modularen Systems, das die geforderten Aufgaben erfüllt.

Hierzu wurde ein allgemeines Konzept entwickelt, das flexibel genug ist, um ohne größere Probleme an Erweiterungen der Datenbank und der Funktionalität angepasst werden kann. Die gezeigte Aufteilung der Anfrageverarbeitung in fünf Aufgabenbereiche bietet eine einfache Möglichkeit weite Verarbeitungsschritte einzufügen ohne aufwendige Änderungen an den anderen Komponenten vornehmen zu müssen. Ferner können einzelne Komponenten einfach ausgetauscht werden ohne die übrigen zu beeinflussen. Die Datenstrukturen wurden möglichst einfach und universell gehalten, um eine weitere Verarbeitung zu vereinfachen.

Die erzeugte Anfrage ist aufgrund der erörterten Problematik bezüglich des Datenbankschemas teilweise sehr komplex und unübersichtlich, da der Optimierer nur begrenzte Möglichkeiten hat die Anfrage zu vereinfachen. Es wäre daher sinnvoll den *Query-Tree* möglichst schon entsprechend aufzubauen um eine effiziente Verarbeitung zu ermöglichen.

Ein weiteres Problem ist die unterschiedliche Syntax der verschiedenen Datenbanksysteme. Um in diesem Punkt flexibel zu bleiben wurden keine Datenbank spezifischen Features, wie z.B. automatische ID-Erzeugung, verwendet, sondern entsprechende Alternativen konstruiert.

Da bisher noch keinerlei Informationen über das Verhalten des Systems in der Praxis vorliegen und mangels verfügbarer Daten keine Belastungstests durchgeführt werden konnten, bleibt abzuwarten wie sich das Konzept im Alltagseinsatz bewährt.

Kapitel 9 Ausblick

Die hier vorgestellte Anfrageverarbeitung ist noch recht einfach aufgebaut und hat quasi noch den Status eines Prototypen. Ihre Alltagstauglichkeit hat sie erst noch unter Beweis zu stellen.

Ferner gibt es noch einige Aspekte, wie zum Beispiel die Optimierung, die noch zu verbessern und auszubauen sind. Der *PlanOptimizer* führt nur eine einfache Zusammenfassung von geeigneten Unteranfragen durch. Durch eine entsprechende logisch äquivalente Umformung der Anfrage wäre es sicher möglich optimale SQL-Anfragen zu erzeugen als mit der aktuellen Version. Eventuell ist es auch möglich vor der eigentlichen Planerzeugung eine logische Optimierung durchzuführen, die auch komplizierte, verschachtelte Strukturen auflösen kann. Es wäre sicherlich möglich den Plan-Optimierer zu verbessern damit auch komplexer verschachtelte Strukturen aufgelöst werden können, dies würde jedoch den Rahmen dieser Arbeit sprengen.

Weiterhin würde es sich lohnen verschiedene Anfragepläne zu generieren, mit einer Analysefunktion zu bewerten und den kostengünstigsten auszuwählen. Eine sinnvolle Bewertungsfunktion wird sich aber erst durch eine statistische Analyse von realen Anfragen finden lassen.

Ferner sollten alle datenbankspezifischen SQL-Strings beliebig konfigurierbar sein, ohne das Projekt neu kompilieren zu müssen. Ein Ansatz auch die fest integrierten Spaltennamen und Tabellenbezeicher frei konfigurierbar zu implementieren ist zwar technisch machbar, würde jedoch die Ausführungsgeschwindigkeit stark beeinflussen, da für jedes Vorkommen ein teurer Zugriff auf die Konfigurationsdaten gemacht werden musste.

Ein weiterer Aspekt wäre der Einbau einen Anfrage Cache-Systems um wiederholte Anfragen schneller und effizienter bearbeiten zu können. Die Ausführung der einzelnen Anfragen sollte wahlweise auf möglichst schnelles Anzeigen der ersten n Treffen erzeugt werden können, bzw. es sollte möglich sein Anfragen iterativ auszuführen.

Anhang A Literaturverzeichnis

Mit95	Bernhard Mitschang, Anfrageverarbeitung in Datenbanksystemen, Vieweg, 1995
Sql94	Gregor Kuhlmann/Friedrich Müllmerstadt, SQL – Der Schlüssel zu realationalen Datenbanken, Rowohlt Taschenbuch Verlag, 1994
Sch95	Uwe Schöning, Logik für Informatiker, 4. Auflage, Spektrum Verlag, 1995
Fle02	Neugestaltete Architektur des META-AKAD-Systems mit Java 2 Enterprise Edition, Marcus Flehmig

Anhang B Abbildungsverzeichnis

Anhang C Tabellenverzeichnis

Anhang D Beispielverzeichnis

Anhang E Klassen Dokumentation

E.1 Package metabase.queryengine.ejb.qectrl

```
public interface metabase.queryengine.ejb.qectrl.QECtrlHome
```

The home of the Controller Bean to build a facade while processing a query

Methods **public metabase.queryengine.ejb.qectrl.QECtrl create()**

```
public class QECtrlException extends java.lang.Exception
```

Thrown when a user tries to execute a malformed query

Constructors **public QECtrlException(String name)**

Constructor

Parameters
name - error name

public QECtrlException(String name, Throwable cause)

Constructor

Parameters
name - error name
cause - error cause

```
public class QECtrlEJB
```

Controls the query processing

Constructors

public QECtrlEJB()

Methods

public void ejbCreate()

public RSHandle executeQuery(QueryTree querytree)

Query execution method

Parameters
querytree - the query

Throws
QECtrlException - if an error occurs

public RSHandle refineQuery(RSHandle oldqueryhandle,
QueryTree refinementquery)

Query refinement method

public void ejbActivate()

Called when the Session is activated.

public void ejbPassivate()

Called when the SessionBean is passivated due to any form of timeout.

public void ejbRemove()

Called when the SessionBean is invalidated.

public void setSessionContext(
SessionContext context)

Gives the bean access to it's context/environment.

public void unsetEntityContext()

Fields

protected context

```
public interface QECtrl
```

Remote Interface for QueryExecutionController Bean

Methods

public RSHandle executeQuery(QueryTree query)

Define and execute a query

Parameters

query - QueryTree-encoded query

Returns

RSHandle object that encapsulates the queryresults

public RSHandle refineQuery(RSHandle oldquery, QueryTree refinementquery)

Refine a previous query

Parameters

oldquery - result set handle of the old query
refinementquery - full QueryTree-encoded query (must be a subset/refinement/specialization of prev. query)

Returns

RSHandle new result set object that encapsulates the queryresults of the subquery

E.2 Package metabase.queryengine.beans.queryparser

```
public class QueryTree implements java.io.Serializable
```

Klasse zur baumartigen Darstellung der Suchanfrage

Constructors

public QueryTree()
Creates new QueryTree

public QueryTree(String origin)

Methods

public void setOrigin(String origin)

public java.lang.String getOrigin(String origin)

public void delTree()

public Node getTree()

public boolean addTree(QueryTree tree, String catenation)

public boolean addTree(Node tree, String catenation)

public boolean addNode(
String name,
String origin,
String value,
String valuetyp,
String operator,
String catenation)

Fields

private rootQueryTreeNode

private origin

```
public class QueryParserException extends java.lang.Exception
```

Thrown when an error occurs while parsing the query

Constructors

public QueryParserException(String name)

Constructor

Parameters
name - error name

```
public class QueryParser
```

QueryParser

Constructors **public QueryParser()**

Methods **public LogicNode parse(QueryTree tree)**

Parse a query tree an create a new logic tree

Parameters
tree - query tree

private LogicNode parseNode(Node node)

Parse a query tree node and create an equivalent logic node

Parameters
node - query tree node

Returns
logic tree node

E.3 Package metabase.queryengine.beans.queryparser.LogicTree

```
public MetadataAttribute extends LogicVariable
```

Constructors	**public MetadataAttribute(String sXMLPathname, String sConstraint)**
Methods	**public java.lang.String getXMLPathname()**
	public java.lang.String getConstraint()
	public java.lang.String toString()
Fields	**private m_sXMLPathname**
	private m_sConstraint

```
public abstract class LogicVariable extends LogicNode
```

logic tree leaf base class

Constructors	**public LogicVariable()**
Methods	**public LogicNode negation()**
	Returns
	negation

```
public class LogicStringConstraint extends LogicConstraint
```

logic tree string leaf

Constructors

public LogicStringConstraint(
String sXMLPathname,
String sOperator,
String sValue)

Constructor

Parameters

sXMLPathname - XML identifier
sOperator - constraint operator
sValue - value

public LogicStringConstraint(
String sXMLPathname,
String sOperator,
String sValue,
boolean casesensitive)

Constructor

Parameters

sXMLPathname - XML identifier
sOperator - constraint operator
bValue - value
casesensitive - true if constraint is casesensitive

Methods

public boolean isCaseSensitive()

Returns

true if constraint is case sensitive

public java.lang.String getValueString()

Returns

value string

public java.lang.String getValueType()

Returns

value type

Fields

public m_sValue

public m_bCaseSensitive

```
public class LogicOrOperator extends LogicOperator
```

Constructors **public LogicOrOperator(LogicNode oLeft, LogicNode oRight)**

Methods **public java.lang.String getOperator()**

```
public class LogicOr extends LogicOperator
```

logic tree or operator

Constructors **public LogicOr(LogicNode oLeft, LogicNode oRight)**

Constructor

Parameters
oLeft - left operand
oRight - right operand

Methods **public java.lang.String getOperator()**

Returns
operator string representation

```
public abstract class LogicOperator extends LogicNode
```

logic tree operator base class

Constructors **protected LogicOperator(LogicNode oLeft, LogicNode oRight)**

Constructor

Parameters
oLeft - left operand
oRight - right operand

Methods **public LogicTree.LogicNode getLeft()**

return left operand

public LogicNode getRight()

return left operand

public java.lang.String toString()

Returns
string representation

public abstract java.lang.String getOperator()

Returns
operator string representation

Fields **private m_oLeft**

private m_oRight

```
public class LogicNotOperator extends LogicOperator
```

Constructors **public LogicNotOperator(LogicNode oRight)**

Methods **public java.lang.String getOperator()**

```
public class LogicNot extends LogicOperator
```

logic tree nor operator

Constructors **public LogicNot(LogicNode oRight)**

Methods **public java.lang.String getOperator()**

Returns

operator string representation

```
public abstract class LogicNode
```

logic tree base class

Constructors **public LogicNode()**

Methods **public abstract java.lang.String toString()**

Returns

string representation

```
public abstract class LogicMetadata extends LogicVariable
```

logic tree metadata leaf base class

Constructors **public LogicMetadata()**

Methods **public abstract java.lang.String getXMLPathname()**

Returns

XML identifier

public abstract java.lang.String getConstraint()

Returns

constraint string

public java.lang.String toString()

Returns

string representation

```
public class LogicIntegerConstraint extends LogicConstraint
```

logic tree integer leaf

Constructors **public LogicIntegerConstraint(**
String sXMLPathname,
String sOperator,
int nValue)

Constructor

Parameters
sXMLPathname - XML identifier
sOperator - constraint operator
nValue - value

Methods **public java.lang.String getValueString()**

Returns
value string

public java.lang.String getValueType()

Returns
value type

Fields **public m_nValue**

```
public class LogicFulltext extends LogicVariable
```

logic tree fulltext leaf

Constructors **public LogicFulltext(String sFulltext)**

Constructor

Returns
sFulltext fulltext string

Methods **public java.lang.String getFulltext()**

Returns
fulltext string

public java.lang.String toString()

Returns
string representation

Fields **private m_sFulltext**

```
public class LogicExistence extends LogicMetadata
```

logic tree existence check leaf

Constructors **public LogicExistence(String sXMLPathname)**

Constructor

Parameters
sXMLPathname - XML identifier

Methods **public java.lang.String getXMLPathname()**

Returns
XML identifier

public java.lang.String getConstraint()

Returns
constraint string

Fields **private m_sXMLPathname**

```
public class LogicDateConstraint extends LogicConstraint
```

logic tree date leaf

Constructors **public LogicDateConstraint(**
String sXMLPathname,
String sOperator,
Date date)

Constructor

Parameters
sXMLPathname - XML identifier
sOperator - constraint operator
date - value

Methods **public java.lang.String getValueString()**

Returns
value string

public java.lang.String getValueType()

Returns
value type

Fields **public m_Value**

```
public abstract class LogicConstraint extends LogicMetadata
```

logic tree constraint leaf base class

Constructors

protected LogicConstraint(String sXMLPathname, String sOperator)

Constructor

Parameters

sXMLPathname - XML identifier
sOperator - constraint operator

Methods

public java.lang.String getXMLPathname()

Returns

XML identifier

public java.lang.String getOperator()

Returns

constraint operator

public java.lang.String getConstraint()

Returns

constraint string

public java.lang.String toString()

Returns

string representation

public abstract java.lang.String getValueString()

Returns

value string

public abstract java.lang.String getValueType()

Returns

value type

Fields

private m_sXMLPathname

private m_sOperator

```
public class LogicBooleanConstraint extends LogicConstraint
```

logic tree boolean leaf

Constructors **public LogicBooleanConstraint(**
String sXMLPathname,
String sOperator,
boolean bValue)

Constructor

Parameters
sXMLPathname - XML identifier
sOperator - constraint operator
bValue - value

Methods **public java.lang.String getValueString()**

Returns
value string

public java.lang.String getValueType()

Returns
value type

Fields **public m_bValue**

```
public class LogicAndOperator extends LogicOperator
```

Constructors **public LogicAndOperator(LogicNode oLeft, LogicNode oRight)**

Methods **public java.lang.String getOperator()**

```
public class LogicAnd extends LogicOperator
```

logic tree and operator

Constructors **public LogicAnd(LogicNode oLeft, LogicNode oRight)**

Constructor

Parameters
left - left operand
right - right operand

Methods **public java.lang.String getOperator()**

Returns
operator string representation

```
public class FulltextAttribute extends LogicVariable
```

Constructors	**public FulltextAttribute(String sFulltext)**
Methods	**public java.lang.String getFulltexte()**
	public java.lang.String toString()
Fields	**private m_sFulltext**

E.4 Package metabase.queryengine.beans.queryparser.QueryTreeElements

```
public class NodeOr extends NodeInternal
```

"Oder"-Knoten eines binären Suchbaumes. Konkrete Realisierung einer "oder"-Verknüpfung im binären Suchbaum Die beiden Kinder dieses Knotens werden bei der Auswertung der Sucheanfrage "oder"-verknüpft.

Constructors **public NodeOr()**

```
public abstract class NodeLeafExistence extends NodeLeaf
```

Constructors **public NodeLeafExistence()**

```
public abstract class NodeLeafConstraint extends NodeLeaf
```

Constructors **public NodeLeafConstraint()**

Methods **public abstract Attributes getAttributeValueObject()**

```
public abstract class NodeLeaf extends Node
```

Blattknoten eines Suchbaumes. In diesem Blatt wird gespeichert, in welchem Suchfeld einer Suchmenge nach welchem Wert gesucht werden soll.

Constructors **public NodeLeaf()**
Creates new NodeLeaf

Methods **public void setAttributeName(String name)**

public java.lang.String getAttributeName()

public void setAttributeOrigin(String origin)

public java.lang.String getAttributeOrigin()

Fields **private attributeName**

private attributeOrigin

```
public abstract class metabase.queryengine.beans.queryparser.QueryTreeElements.NodeInternal
extends Node
```

Abstrakter interner Knoten eines Suchbaumes. Ein Suchbaum baut sich aus vielen dieser internen Knoten auf, jeweils konkret als "und" oder "oder" realisiert. An den Blättern sind Objekte der Klasse QueryTreeNodeLeaf

Constructors	**public NodeInternal()**
Methods	**public void setLeftNode(Node node)**
	public void setRightNode(Node node)
	public Node getRightNode()
	public Node getLeftNode()
Fields	**private leftChild**
	private rightChild

```
public class NodeAndNot extends NodeInternal
```

"UndNicht"-Knoten eines binären Suchbaumes. Konkrete Realisierung einer "und-nicht"-Verknüpfung im binären Suchbaum Die beiden Kinder dieses Knotens werden bei der Auswertung der Sucheanfrage "und"-verknüpft und das Ergebnis negiert.

Constructors	**public NodeAndNot()**

```
public class NodeAnd extends NodeInternal
```

"Und"-Knoten eines binären Suchbaumes. Konkrete Realisierung einer "oder"-Verknüpfung im binären Suchbaum Die beiden Kinder dieses Knotens werden bei der Auswertung der Sucheanfrage "und"-verknüpft.

Constructors	**public NodeAnd()**

```
public abstract class metabase.queryengine.beans.queryparser.QueryTreeElements.Node implements
java.io.Serializable
```

Abstrakte Knoten eines Suchbaumes. Die Klasse kapselt alle möglichen Knoten (auch Blätter) eines Suchbaumes

Constructors **public Node()**

```
public interface LessAttributes implements Attributes
```

```
public class LeafNotExists extends NodeLeafExistence
```

Constructors **public LeafNotExists()**

Creates new LeafNotExists

```
public class LeafLess extends NodeLeafConstraint
```

Constructors **public LeafLess()**

Creates new LeafLess

Methods **public void setAttributeValueObject(LessAttributes attribute)**

public Attributes getAttributeValueObject()

Fields **private attribute**

```
public class LeafGreater extends NodeLeafConstraint
```

Constructors **public LeafGreater()**

Creates new LeafGreater

Methods **public void setAttributeValueObject(GreaterAttributes attribute)**

public Attributes getAttributeValueObject()

Fields **private attribute**

```
public class LeafExists extends NodeLeafExistence
```

Constructors	**public LeafExists()**
	Creates new LeafExists

```
public class LeafEvenLess extends NodeLeafConstraint
```

Constructors	**public LeafEvenLess()**
	Creates new LeafEvenLess
Methods	**public void setAttributeValueObject(EvenLessAttributes attribute)**
	public Attributes getAttributeValueObject()
Fields	**private attribute**

```
public class LeafEvenGreater extends NodeLeafConstraint
```

Constructors	**public LeafEvenGreater()**
	Creates new LeafEvenGreater
Methods	**public void setAttributeValueObject(EvenGreaterAttributes attribute)**
	public Attributes getAttributeValueObject()
Fields	**private attribute**

```
public class LeafEqual extends NodeLeafConstraint
```

Constructors	**public LeafEqual()**
	Creates new LeafEqual
Methods	**public void setAttributeValueObject(EqualAttributes attribute)**
	public Attributes getAttributeValueObject()
	public java.lang.String getOperator()
	public java.lang.Class getAttributeValueType()
Fields	**private attribute**

```
public class LeafContainsRight extends NodeLeafConstraint
```

Constructors	**public LeafContainsRight()**
	Creates new LeafContainsRight
Methods	**public void setAttributeValueObject(ContainsAttributes attribute)**
	public Attributes getAttributeValueObject()
Fields	**private attribute**

```
public class LeafContainsLeft extends NodeLeafConstraint
```

Constructors	**public LeafContainsLeft()**
	Creates new LeafContainsLeft
Methods	**public void setAttributeValueObject(ContainsAttributes attribute)**
	public Attributes getAttributeValueObject()
Fields	**private attribute**

```
public class LeafContains extends NodeLeafConstraint
```

Constructors	**public LeafContains()**
	Creates new LeafContains
Methods	**public void setAttributeValueObject(ContainsAttributes attribute)**
	public Attributes getAttributeValueObject()
Fields	**private attribute**

```
public interface metabase.GreaterAttributes implements Attributes

public interface EvenLessAttributes implements Attributes

public interface EvenGreaterAttributes implements Attributes

public interface EqualAttributes implements Attributes

public interface ContainsAttributes implements Attributes

public interface Attributes implements java.io.Serializable

public class AttributeString implements ContainsAttributes,                EqualAttributes
```

Constructors	**public AttributeString()**
	Creates new AttributeString
	public AttributeString(String value)
Methods	**public java.lang.String getAttributeValue()**
	public void setAttributeValue(String value)
Fields	**private attributeValue**

```
public class AttributeInt implements EqualAttributes,      LessAttributes, GreaterAttributes,
EvenLessAttributes, EvenGreaterAttributes
```

Constructors	**public AttributeInt()**
	Creates new AttributeInteger
	public AttributeInt(int value)
Methods	**public int getAttributeValue()**
	public void setAttributeValue(int value)
Fields	**private attributeValue**

```
public class AttributeDate implements EqualAttributes,      LessAttributes, GreaterAttributes,
EvenLessAttributes, EvenGreaterAttributes
```

Constructors	**public AttributeDate()**
	Creates new AttributeDate
	public AttributeDate(Date value)
Methods	**public java.util.Date getAttributeValue()**
	public void setAttributeValue(Date value)
Fields	**private attributeValue**

```
public class AttributeBoolean implements EqualAttributes
```

Constructors	**public AttributeBoolean()**
	Creates new AttributeBoolean
	public AttributeBoolean(Boolean value)
Methods	**public boolean getAttributeValue()**
	public void setAttributeValue(boolean value)
Fields	**private attributeValue**

E.5 Package metabase.queryengine.beans.logictransformer

```
public class LogicTransformerException
       extends java.lang.Exception
```

Thrown when an error occurs while performing a logical transformation

Constructors

public LogicTransformerException(String name)

Constructor

Parameters

name - error name

public LogicTransformerException(String name, Throwable cause)

Constructor

Parameters

name - error name
cause - error cause

```
public class LogicTransformer
```

Provides some logical transformations

Constructors

public LogicTransformer()

Methods

public LogicNode normalize(LogicNode tree)

Normalize a given logic tree

Parameters
tree - logic tree
Returns
normalized tree

public LogicNode transformToDNF(LogicNode tree)

generates a dnf from a given logic tree

Parameters
tree - logic tree
Returns
equivalent tree in dnf

public LogicNode transformToKNF(LogicNode tree)

generates a knf from a given logic tree

Parameters
tree - logic tree
Returns
equivalent tree in knf

protected LogicNode decomposeDNF(LogicNode tree)

generates a dnf from a given normalized logic tree

Parameters
tree - normalized logic tree
Returns
equivalent tree in dnf

protected LogicNode decomposeKNF(LogicNode tree)

generates a knf from a given normalized logic tree

Parameters
tree - normalized logic tree
Returns
equivalent tree in knf

E.6 Package metabase.queryengine.beans.plangenerator

```
public class QueryPlan
```

This class represents a query plan

Constructors

public QueryPlan(PlanNode plantree,Map fulltext)

Constructor

Parameters

plantree - query plan tree
fulltext - map of temporary fulltext tables

Methods

public PlanNode getPlanTree()

Returns

query plan Node

public void setPlanTree(PlanNode plantree)

Parameters

plantree - query plan Node

public java.util.Map getFulltextMap()

Returns

map of fulltext cache tables

public java.lang.String toString()

Returns

string representation

Fields

private m_PlanTree

private m_Fulltext

```
public class PlanGeneratorException extends java.lang.Exception
```

Thrown when an error occurs while generatian a query plan

Constructors

public PlanGeneratorException(
String name)

public PlanGeneratorException(
String name,
Throwable cause)

Constructor

Parameters

name - error name
cause - error cause

```
public class PlanGenerator
```

PlanGenerator

Constructors **public PlanGenerator()**

Constructor

Methods **public QueryPlan parse(LogicNode logicTree)**

create a new query plan

Parameters
logicTree - logic tree
Returns
query plan

private PlanNode parseNode(LogicNode node)

create a query plan node

Parameters
node - logic tree node
Returns
query plan node

Fields **private m_MetaModelDAO**

private m_FulltextCacheDAO

private m_FulltextMap

```
public class MetaModelDAOException extends java.lang.Exception
```

Thrown when a user tries to access invalid metadata

Constructors **public MetaModelDAOException(String name)**

Constructor

Parameters
name - error name

public MetaModelDAOException(String name, Throwable cause)

Constructor

Parameters
name - error name
cause - error cause

```
public class MetaModelDAO
```

DAO for metamodel access

Constructors

public MetaModelDAO()

Constructor

Methods

publicAttribute getAttribute(String sXMLPathname)

Returns the attributeobject which refers to a given XML name

Parameters
sXMLPathname - xml path
Returns
requestet attribute

public java.lang.String getJavaclassname(String sXMLPathname)

Returns the javaclass name of the attribute which refers to a given XML name

Parameters
sXMLPathname - xml path
Returns
java class name

public java.util.List getDependencies(String sXMLPathname)

Returns the dependecy list of the attribute which refers to a given XML name

Parameters
sXMLPathname - xml path
Returns
dependency list

Fields

private m_MetaModelObjectHome

private m_MetaModelObjectCache

private m_DependencyCache

```
public class metabase.queryengine.beans.plangenerator.Join
```

This class represents a join

Constructors

public Join(String sFromTable, String sFromAttribute, String sToTable, String sToAttribute)

Constructor

Parameters

sFromTable - source table
sFromAttribute - source column
sToTable - destination table
sToAttribute - destination column

public Join(Attribute from,Attribute to)

Constructor

Parameters

from - source attribute
to - destination attribute

Methods

public java.lang.String toString()

Returns

string representation

public java.lang.String getFromTable()

Returns

from table name

public java.lang.String getToTable()

Returns

to table name

public void addUsedTablesTo(Collection tables)

Adds all used table names to a given container

Parameters

tables - containter where the table names should be added

Fields

private m_From

private m_To

```
public class FulltextCacheDAOException extends Exception
```

Thrown when a user tries to execute a malformed fulltext query

Constructors **public FulltextCacheDAOException(String name)**

Constructor

Parameters
name - error name

public FulltextCacheDAOException(String name, Throwable cause)

Constructor

Parameters
name - error name
cause - error cause

```
public FulltextCacheDAO
```

DAO for fulltext access

Constructors **public FulltextCacheDAO()**

Constructor

Methods **public java.lang.String getCacheTable(String sSearchstring)**

Execute the fulltext query for a given searchstring

Parameters
sSerachstring - search string
Returns
temporary table name

Fields **private m_FulltextCache**

```
public class AttributeFunction extends Attribute
```

This class represents an aggregate function

Constructors **public AttributeFunction(String function,Attribute attribute)**

Constructor

Parameters
function - function identifier
attribute - function argument

Methods **public java.lang.String toString()**

Returns
a string representing the aggegat function

Fields **private m_sFunction**

```
public class CollectionLister
```

helper class for converting colletions to strings

Constructors **public CollectionLister()**

Methods **public java.lang.String list(Collection c, String seperator)**

Creates a string list of a given collection object

Parameters
c - collection
seperator - string list value seperator
Returns
string list

public java.lang.String list(Iterator iterator, String seperator)

Creates a string list of a given iterator

Parameters
i - iterator
seperator - string list value seperator
Returns
string list

```
public class metabase.queryengine.beans.plangenerator.Attribute
```

This class represents a column of a database table

Constructors

public Attribute(String sTable, String sIdentifier)

Constructor

Parameters

sTable - table name
sIdentifier - column name

Methods

public boolean equals(Object o)

Parameters

o - Object to compare

Returns

true if the give object is equal with this

public java.lang.String getTable()

Returns

name table

public java.lang.String getIdentifier()

Returns

name of column

public java.lang.String toString()

Returns

a database identifier for the attribute

Fields

private m_sTable

private m_sIdentifier

E.7 Package metabase.queryengine.beans.plangenerator.PlanTree

```
public class PlanSubset extends PlanNode
```

plan subset class

Constructors **public PlanSubset(Qualification qualification)**

Constructor

Parameters

qualification - qualification of the subquery

Methods **public java.lang.String toString()**

Returns

string representation

public void addUsedTablesTo(Set tables)

Adds all used table names to a given container

Parameters

tables - containter where the table names should be added

public void addJoinConditionsTo(Set joins)

Adds all used joins to a given container

Parameters

joins - containter where the join names should be added

public Qualification getQualification()

Returns

Qualification

Fields **private m_Qualification**

```
public class PlanOr extends PlanOperator
```

plan or operator

Constructors **public PlanOr(Collection operands)**

Constructor

Parameters
operands - collection of operands

public PlanOr(PlanNode left,PlanNode right)

Constructor

Parameters
left - left operand
right - right operand

Methods **public java.lang.String toString()**

Returns
string representation

```
public class PlanNot extends PlanOperator
```

plan not operator

Constructors **public PlanNot(PlanNode operand)**

Constructor

Parameters
operand - Operand

Methods **public java.lang.String toString()**

Returns
string representation

```
public abstract class PlanOperator extends PlanNode
```

base class for plan operators

Constructors **protected PlanOperator(Collection operands)**

Constructor

Parameters
operands - collection of operands

Methods **protected java.util.Collection operands()**

Returns
operand collection

public void addUsedTablesTo(Set tables)

Adds all used table names to a given container

Parameters
tables - containter where the table names should be added

public void addJoinConditionsTo(Set joins)

Adds all used joins to a given container

Parameters
joins - containter where the join names should be added

public java.util.Iterator iterator()

Returns
operand iterator

Fields **private m_Operands**

```
public abstract class PlanNode
```

base class plan nodes

Constructors **public PlanNode()**

Methods **public abstract java.lang.String toString()**

Returns
string representation

public abstract void addUsedTablesTo(Set tables)

Adds all used table names to a given container

Parameters
tables - containter where the table names should be added

public abstract void addJoinConditionsTo(Set tables)

Adds all used joins to a given container

Parameters
joins - containter where the join names should be added

public int hashCode()

Returns
hash code for this object

public boolean equals(Object o)

Parameters
o - Object to compare

Returns
true if the give object is equal with this

public java.util.Set getUsedTables()

Returns
set of used tables

public java.util.Set getJoinConditions()

Returns
set of join conditions

```
public class PlanConstraint extends PlanNode
```

plan constraint class

Constructors **public PlanConstraint(Qualification qualification)**

Constructor

Parameters
qualification - Qualification of this constraint object

Methods **public java.lang.String toString()**

Returns
string representation

public void addUsedTablesTo(Set tables)

Adds all used table names to a given container

Parameters
tables - containter where the table names should be added

public void addJoinConditionsTo(Set joins)

Adds all used joins to a given container

Parameters
joins - containter where the join names should be added

public Qualification getQualification()

Returns
Qualification

Fields **private m_Qualification**

```
public class PlanAnd extends PlanOperator
```

plan and operator

Constructors

public PlanAnd(Collection operands)

Constructor

Parameters

operands - collection of operands

public PlanAnd(PlanNode left, PlanNode right)

Constructor

Parameters

left - left operand
right - right operand

Methods

public java.lang.String toString()

Returns

string representation

E.8 Package metabase.queryengine.beans.plangenerator.QualificationTree

```
public class QualificationOr extends QualificationOperator
```

qualification tree or operator

Constructors **public QualificationOr(Collection operands)**

Constructor

Parameters
operands - operand collection

public QualificationOr(Qualification left, Qualification right)

Constructor

Parameters
left - left operand
right - right operand

Methods **public java.lang.String toString()**

Returns
string representation

```
public class QualificationNot extends QualificationOperator
```

qualification tree not operator

Constructors **public QualificationNot(Qualification operand)**

Constructor

Parameters
operand - operand

Methods **public java.lang.String toString()**

Returns
string representation

```
public abstract class QualificationOperator extends Qualification
```

qualification tree operator base class

Constructors

protected QualificationOperator(Collection operands)

Constructor

Parameters

operands - operand collection

Methods

public java.util.Collection operands()

Returns

operand collection

public void addUsedTablesTo(Set tables)

Adds all used table names to a given container

Parameters

tables - containter where the table names should be added

public void addAttributesTo(Set attributes)

Adds all used attibutes to a given container

Parameters

tables - containter where the attributes should be added

public void addJoinConditionsTo(Set joins)

Adds all used joins to a given container

Parameters

joins - containter where the join names should be added

public java.util.Iterator iterator()

Returns

iterator

public abstract java.lang.String toString()

Returns

string representation

Fields

private m_Operands

```
public class QualificationConstraint extends Qualification
```

qualification tree constraint leaf

Constructors

public QualificationConstraint(
Attribute attribute,
String sConstraint,
Collection joins)

Methods

public java.lang.String toString()
Returns
string representation

public void addUsedTablesTo(
Set tables)

public void addAttributesTo(
Set attributes)

public void addJoinConditionsTo(
Set joins)

Fields

private m_Attribute

private m_sConstraint

private m_JoinConditions

```
public class QualificationAnd extends QualificationOperator
```

qualification tree and operator

Constructors

public QualificationAnd(Collection operands)

Constructor

Parameters
operands - operand collection

public QualificationAnd(Qualification left, Qualification right)

Constructor

Parameters
left - left operand
right - right operand

Methods

public java.lang.String toString()

Returns
string representation

```
public abstract class Qualification
```

qualification tree base class

Constructors **public Qualification()**

Methods **public abstract java.lang.String toString()**

Returns
string representation

public int hashCode()

Returns
hash code for this object

public boolean equals(Object o)

Parameters
o - Object to compare
Returns
true if the give object is equal with this

public abstract void addUsedTablesTo(Set tables)

Adds all used table names to a given container

Parameters
tables - containter where the table names should be added

public abstract void addAttributesTo(Set attributes)

Adds all used attributes to a given container

Parameters
joins - containter where the attributes should be added

public abstract void addJoinConditionsTo(Set tables)

Adds all used joins to a given container

Parameters
joins - containter where the join names should be added

public java.util.SortedSet getUsedTables()

Returns
set of used tables

public java.util.SortedSet getJoinConditions()

Returns
set of join conditions

public java.util.SortedSet getAttributes()

Returns
set of attributes

E.9 Package metabase.queryengine.beans.planoptimizer

```
public class PlanOptimizerException extends java.lang.Exception
```

Thrown when a error occurs during plan optimization

Constructors **public PlanOptimizerException(String name)**

Constructor

Parameters
name - error name

public PlanOptimizerException(String name, Throwable cause)

Constructor

Parameters
name - error name
cause - error cause

```
public class PlanOptimizer
```

PlanOptimizer class

Constructors **public PlanOptimizer()**

Methods **public void optimize(QueryPlan plan)**

Optimizes a query plan

Parameters
plan - query plan

private PlanNode optimizeNode(PlanNode node)

create a optimzed node

Parameters
node - query plan node
Returns
a new opimized node

protected static void intersect(Set result, Set a, Set b)

creates an intersection of two sets

Parameters
a - set
b - set
result - resultset

E.10 Package metabase.queryengine.beans.queryprocessor

`public class QueryProcessorException extends java.lang.Exception`

Thrown when an error occurs during query execution

Constructors **public QueryProcessorException(String name)**

Constructor

Parameters
name - error name

public QueryProcessorException(String name, Throwable cause)

Constructor

Parameters
name - error name
cause - error cause

`public class QueryProcessor`

QueryProcessor class

Constructors **public QueryProcessor()**

Constructor

Methods **public int process(QueryPlan plan)**

execute a query plan and return a valid result set id

Parameters
plan - query paln
Returns
result set id

Fields **private m_ConfigDAO**

```
public class ConfigDAOException extends java.lang.Exception
```

Thrown when a user tries to access an invalid configuration entry

Constructors **public ConfigDAOException(String name)**

Constructor

Parameters
name - error name

public ConfigDAOException(String name, Throwable cause)

Constructor

Parameters
name - error name
cause - error cause

```
public class metabase.queryengine.beans.queryprocessor.ConfigDAO
```

DAO providing configuration access

Constructors **public ConfigDAO()**

Constructor

Methods **public java.lang.String getParameter(String sName)**

Returns the paramter value which refers to a given name

Parameters
sName - parameter name
Returns
parameter value

Fields **private m_ConfigurationStoreHome**

private m_ConfigurationStoreCache

E.11 Package metabase.queryengine.beans.servicelocator

```
public class ServiceLocatorException extends java.lang.Exception
```

Thrown when a service access failed

Constructors **public ServiceLocatorException(String name)**

Constructor

Parameters
name - error name

public ServiceLocatorException(String name, Throwable cause)

Constructor

Parameters
name - error name
cause - error cause

```
public class ServiceLocator
```

service access class

Constructors **private ServiceLocator()**
Constructor

Methods **Public static ServiceLocator getInstance()**

Returns the instance of the singelton

Returns
service locator instance

public java.lang.Object lookup(String jndiName, Class className)

Returns the Object value which refers to a given jndi name

Parameters
jndiName - object name
className - object type
Returns
requested object

public DataSource getDataSource()

Returns the DataSource object of the application

Returns
datasource object

Fields **private ic**

private cache

private static me

E.12 Package metabase.queryengine.beans.queryresult

```
public class RSHandle implements java.io.Serializable
```

Constructors	**public RSHandle(long queryID)**
Methods	**public long getRSId()**
	public java.lang.String getIdent()
Fields	**private ident**
	private rsid

```
public class InvalidResultSetHandleException extends java.lang.Exception
```

Thrown when a user tries to execute a malformed query

Constructors	**public InvalidResultSetHandleException(String name)**

E.13 Package metabase.queryengine.ejb.ftcache

```
public interface FTCacheHome
```

The home of the Controller Bean providing a full text query cache

Methods **public metabase.queryengine.ejb.ftcache.FTCache create()**

```
public class metabase.queryengine.ejb.ftcache.FTCacheEJB
```

Implementation of the full text query cache controller interface

See Also

metabase.queryengine.ejb.ftcache.FTCache

Constructors **public FTCacheEJB()**

Methods

public void ejbCreate()

public java.lang.String getCacheTableNameForFTQuery(
String searchstring)

public void pregenerateCacheTableForFTQuery(
String searchstring)

public void invalidateCacheForFTQuery(
String searchstring)

public void invalidateAll()

public int getMaxNumberOfCacheTable()

public void setMaxNumberOfCacheTable(int size)

public int getMaxNumberOfTuples()

public void setMaxNumberOfTuples(int size)

public int getTableUseTimeout()

public void setTableUseTimeout(int size)

public void ejbActivate()

public void ejbPassivate()

public void ejbRemove()

public void setSessionContext(SessionContext context)

public void unsetEntityContext()

Fields **protected context**

```
public interface metabase.queryengine.ejb.ftcache.FTCache
```

Remote Interface for QueryExecutionController Bean

Methods

public int getTableUseTimeout()

Retrieve MaxNumberOfTuples out of configuration store

Returns

MaxNumberOfTuples

public void setTableUseTimeout(int size)

Store TableUseTimeout persistently into configuration store, i.e. the specify the timeout after a query could be invalidated

Parameters

size - TableUseTimeout