Seminar Informationsintegration und Informationsqualität

Produkte und Prototypen

Matthias Käppler

18. Juni 2006

Betreuer:
Dipl.-Inf. Jürgen Göres,
AG Heterogene Informationssysteme,
TU Kaiserslautern

Inhaltsverzeichnis

Zusammenfassung Die Problematik, Anfragen auf verteilte und/oder heterogene Datenquellen durch eine gemeinsame Schnittstelle zu ermöglichen, ist beinahe so alt wie Datenbanksysteme selbst und stellt seit jeher eine Herausforderung für Unternehmen und Datenbankadministratoren im Besonderen dar. Strukturelle und semantische Unterschiede heterogener Datenquellen zu analysieren, sowie deren Inhalte in ein gemeinsames Datenschema zu integrieren, ist daher seit längerem Thema in Forschung und Entwicklung, wie auch Motivation hinter diversen industriellen Softwarelösungen. Neben einer Vorstellung der Problemfelder, die es im Rahmen der Informationsintegration zu bewältigen gibt, sowie einer kurzen Übersicht über Lösungstrategien und -techniken, soll im Laufe dieser Arbeit ein Überblick über Forschungsanstrengungen in der Entwicklung von Prototypen zur softwaregestützten Integration verteilter, heterogener Datenquellen gegeben, sowie darauf aufbauende Produkte vorgestellt werden.

1 Grundlagen der Informationsintegration

In den folgenden Abschnitten soll eine Einführung in die Problematik der Informationsintegration erfolgen. Da der Schwerpunkt dieser Arbeit auf den Werkzeugen zur Informationsintegration liegt, ist eine Klärung der zugrundeliegenden Problematik und Terminologie zwar erforderlich, erfolgt jedoch nur knapp.[1] Ein solides Grundverständnis im Bereich Internettechnologie und Datenbank- und Informationssysteme wird vorausgesetzt, da weiterführende Erklärungen in diesen Gebieten den Rahmen dieser Arbeit sprengen würden.

Neben einer Motivation zum Thema sollen die zentralen Problembereiche erfasst, sowie verbreitete und bewährte Verfahren und Techniken zur Integration heterogener Datenquellen vorgestellt werden. Desweiteren soll eine typische Referenzarchitektur sowie eine Anzahl abgeleiteter Kriterien vorgestellt werden, um Integrationssysteme zu klassifizieren.

1.1 Motivation

Häufig stehen Unternehmen vor dem Problem, eine Anzahl an unternehmensinternen oder -externen Datenquellen unter einer einzigen unternehmensglobalen Schnittstelle zu integrieren. Mögliche Gründe dafür sind die Zusammenlegung zweier Abteilungen oder die Fusion zweier Unternehmen. In beiden Fällen ist es wahrscheinlich, dass die Datenbestände, die es zu integrieren gilt, in vielerlei Hinsicht heterogen sind.

Datenquellen können zunächst *strukturell* unterschiedlichster Natur sein. Neben *strukturierten* Datenquellen wie (objekt-)relationalen und objektorientierten Datenbanksystemen finden sich *unstrukturierte* Quellen wie Webseiten (z.B.

[1] Der interessierte Leser findet in [1] und [2] eine gute Einführung in das Thema, sowie Hinweise auf weiterführende Literatur.

HTML und plain text) sowie *semi-strukturierte* Quellen (z.B. XML). Neben dieser Form der Heterogenität, die auf die Verwendung unterschiedlicher Datenmodelle zurückzuführen ist, kann auch eine rein deskriptive Heterogenität vorliegen. So können beispielsweise zwei autonome Datenbanksysteme unter Verwendung desselben Datenmodells eine identische Menge realweltlicher Objekte beschreiben, jedoch in Art und Struktur der Beschreibung voneinander abweichen. Es werden also Systeme benötigt, die eine Menge heterogener, möglicherweise verteilter Datenquellen unter einer einzigen Schnittstelle integrieren.

1.2 Schemaintegration

Der Vorgang, eine Menge heterogener *Quellschemas* in ein einziges *Zielschema* zu überführen, wird auch als *Schemaintegration* bezeichnet. Dabei lassen sich zwei Phasen oder Prozesse unterscheiden: Durch *Schema Matching* werden Korrespondenzen zwischen Elementen des Quellschemas und des Zielschemas identifiziert. Es gilt dabei also Zuordnungen zu finden und zu definieren, die den semantischen Bezug zwischen Quelle und Ziel herstellen. Anschließend werden durch den Vorgang des *Schema Mapping* Abbildungen definiert, die – basierend auf den zuvor definierten Zuordnungen – die Daten, die in den Strukturen der Datenquelle vorliegen, möglichst verlustfrei und korrekt in das Datenmodell und die Struktur des Zielschemas überführen.[2]

Für den Ablauf des Integrationsprozesses gilt es eine Reihe von Aspekten zu diskutieren, wie etwa Integrationskonflikte, die aufgrund von strukturellen oder semantischen Unterschieden in den Datenquellen auftreten können, als auch die Wahl eines geeigneten Integrationsverfahrens. Solche und weitere Punkte sollen in den folgenden Abschnitten erläutert werden.

1.3 Integrationsverfahren

Die Integration mehrerer heterogener Quellschemas in ein einziges Zielschema findet häufig durch die Definition von *Sichten* über die Quellschemas statt und wird daher auch als *sichtenbasierte Integration* bezeichnet [6]. Zwei weit verbreitete Verfahren, *Global-as-View (GaV)* und *Local-as-View (LaV)*, sollen hier kurz beschrieben werden. Eine ausführliche theoretische Einführung in sichtenbasierte Datenintegration und Anfrageverarbeitung findet sich in [5]. Falls im weiteren Verlauf der Arbeit andere Techniken Verwendung finden sollten, werden sie zusammen mit dem jeweiligen System vorgestellt.

Global-as-View. Bei Verwendung des GaV-Ansatzes ist das Zielschema als integrierte Sicht auf die Quellschemas zu verstehen, wobei jedem Element aus dem Zielschema eine Anfrage auf die Quellschemas zugeordnet wird. Bei Anfragen auf das Zielschema erfolgt also eine *Anfragetransformation* in Anfragen

[2] An dieser Stelle klingt bereits heraus, dass nicht immer eine optimale Zuordnung definiert werden kann; verschiedene Gründe warum dies fehlschlagen kann, werden in Abschnitt 1.4 diskutiert.

auf die Quellschemas. Bei GaV geschieht dies relativ trivial durch Ersetzen der relevanten Attribute durch ihre Definition in der Sicht, auch als *view unfolding* bezeichnet. Dieser Ansatz erscheint intuitiv, und findet in vielen Systemen Verwendung (vgl. Abschn. 3.1, 3.2, 3.6).

Local-as-View. Im Gegensatz zu GaV werden in LaV die Quellschemas als Sichten über das Zielschema modelliert, das integrierte Schema ist also *unabhängig* von den Quellschemas. Dies erscheint zunächst nicht intuitiv; in der Tat ist die Anfrageverarbeitung in LaV nicht trivial. Zwei in [5] vorgeschlagene Ansätze zur Anfrageverarbeitung in LaV-basierten Systemen sind das *view based query rewriting* (VBQR) sowie das *view based query answering* (VBQA). Bei VBQR findet ein Umschreiben der Anfrage an das Zielschema in Anfragen auf die Sichten in den Quellen statt, unter Verwendung einer fixen Anfragesprache (meist derjenigen die zur initialen Anfrage auf das integrierte Schema verwendet wurde). Diese Transformationen können jedoch offensichtlich verlustbehaftet sein. VBQA hingegen stellt keine Forderung, *wie* die Anfrage zu erfolgen hat, und ermöglicht somit das Ergebnis der Anfrage durch Ausnutzung von Zusatzinformationen (z.B. abgeleitet aus der Sichterweiterung) zu berechnen. LaV findet unter anderem Verwendung im Agora-System, wie beschrieben in [27].

Es soll noch Erwähnung finden, dass bei GaV das integrierte Schema offensichtlich in Abhängigkeit der Quellschemas erstellt werden muss; in der Literatur wird häufig hier ein Vorteil bei LaV gesehen, da etwa bei einer Schemaevolution oder dem Wegfall einer Quelldatenbank das Mapping im Zielschema angepasst werden muss. Da bei LaV das Mapping in den Quellen erfolgt, erlaube es – ein stabiles integriertes Schema vorausgesetzt – weit flexiblere Verbunde. In wie weit dies den Einsatz des weitaus komplexeren LaV-Ansatz rechtfertigt, ist jedoch sicherlich fraglich. GaV hat sich als praktischere Alternative in vielen Systemen durchgesetzt.

1.4 Konfliktklassen

Bei der Schemaintegration können aufgrund der Heterogenität der Datenquellen unterschiedliche Konflikte auftreten. Diese können basierend auf [1] in folgende Klassen gegliedert werden:

Semantische Konflikte. Im Kontext der unterschiedlichen Miniwelten, die in den Datenquellen abgebildet sind, können zwei Datenquellen identische Konzepte modellieren, ohne dass dabei die resultierenden Tupelmengen übereinstimmen müssen. Dies kann den Grund haben, dass für eine Quelle nur ein Ausschnitt der beschriebenen Objekte relevant ist. Durch solche rein *mengenmäßigen* Diskrepanzen kann es somit zu Überschneidungen, Inklusionen und Disjunktionen der beschriebenen Tupelmengen kommen, die es bei der Abbildung auf das integrierte Schema zu berücksichtigen gilt.

Beschreibungskonflikte. Selbst wenn zwei Datenbankschemas die gleiche Miniwelt beschreiben und sich äquivalente Objektklassen finden lassen, kann es zu Inkonsistenzen in der Beschreibung der Objekte der Miniwelt kommen. Homonyme (identische Bezeichnungen für verschiedene Sachverhalte) und Synonyme (verschiedene Bezeichnungen für denselben Sachverhalt), aber auch die Verwendung unterschiedlicher Datentypen oder Wertebereiche zur Modellierung derselben Information führen zu Konflikten dieser Klasse.

Heterogenitätskonflikte. Verwenden die zu integrierenden Quellen unterschiedliche Datenmodelle, so kann es bei einer Integration zu Informationsverlust kommen, falls die Ausdrucksmächtigkeit beider Modelle zu stark variiert und das Zielmodell bestimmte Aussagen nicht abbilden kann. Im schlimmsten Fall sind diese Konflikte nicht zu beheben.

Strukturelle Konflikte. Bei der Abbildung derselben realweltlichen Information kann es trotz einer äquivalenten Semantik zu Konflikten in der Art der Abbildung kommen. So könnte etwa das Geschlecht eines `Person` Objektes in Schema A durch ein Attribut `Geschlecht` beschrieben sein, in Schema B aber durch eine Unterklassenbildung in `Mann` und `Frau`.

Solche Konflikte müssen während des Integrationsprozesses von einem Experten manuell aufgelöst werden. Manche Werkzeuge bieten jedoch auch (semi-)automatische Mechanismen zur Auflösung von Konflikten, wie etwa durch Anwendung von Thesauri.

2 Charakteristika von Integrationssystemen

Nach den grundlegenden Erläuterungen zum Thema Informationsintegration sollen in diesem Kapitel die wesentlichen Merkmale unterschiedlicher Integrationssysteme diskutiert werden. Dazu werden zunächst zwei architekturelle Grundformen von Integrationssystemen vorgestellt und darauf aufbauend in Abschnitt 2.2 eine Referenzarchitektur vorgeschlagen, sowie deren wichtigsten Komponenten identifiziert. Es wird hierbei ein relativ weites, umfassendes Sichtfeld gewählt, einzelne Softwarelösungen können also durchaus disjunkte Bereiche abdecken (vgl. Garlic und Clio). Abschnitt 2.3 schlägt schließlich eine Reihe von Kriterien vor, anhand derer Integrationssysteme klassifiziert werden können.

2.1 Konsolidierung und Föderierung

Bei der Art der Integration mehrerer autonomer Datenquellen unterscheidet man in der Regel zwischen *föderierten* und *konsolidierten* Systemen [2]. Während bei letzteren die Daten der Quellen im Zielsystem aggregiert werden und Anfragen somit aus der lokalen Datenquelle sofort beantwortet werden können, werden Anfragen an föderierte Datenbanksysteme in Anfragen an die beteiligten Datenquellen umformuliert.

Konsolidierung. Die Materialisierung der zu integrierenden Sichten und die damit einhergehende Erzeugung von Redundanz kann in einigen Fällen erwünscht sein, etwa wenn ein Schnappschuss eines operationalen Datenbestandes erzeugt werden soll, wie es bei der Einrichtung und Pflege von *Data Warehouses* der Fall ist. Als Data Warehouse bezeichnet man eine zu Analysezwecken angelegte Datenbank, deren Datenbasis auf der Aggregation in der Regel mehrerer Quelldatenbanken, wie etwa der operationalen Datenbanken eines Unternehmens, beruht [3]. Da die Anfragen auf Kopien der operationalen Daten erfolgen, werden die Operativsysteme nicht belastet. Die extrahierten Daten können dann als Entscheidungsgrundlage dienen oder dazu benutzt werden, Geschäftsmetriken zu überprüfen und abzuleiten.

Föderierung. Im Gegensatz zur Konsolidierung findet bei föderierten Systemen keine Materialisierung des Zielschemas statt. Ein *Föderierungsdienst (Mediator)* sorgt stattdessen dafür, dass Anfragen an das Zielschema in Anfragen an die Quellschemas übersetzt werden, wobei der Zugriff auf die unterschiedlichen Datenquellen durch entsprechende *Wrapper* realisiert wird. Durch dieses Verfahren bleiben die integrierten Datenquellen autonom, Anfragen werden lediglich weitervermittelt.

Die Verfahren im Vergleich. Abb. 1 veranschaulicht die Prinzipien beider Verfahren im Vergleich. In *a)* werden die Quelldatenbanken zu einem föderierten System zusammengeschlossen. Durch den Föderierungsdienst findet eine Anfragetransformation zwischen Zielschema und den Quellschemas statt. In *b)* werden die Daten der Quelldatenbanken in einer Zieldatenbank materialisiert. Dies geschieht in der Regel periodisch und *unabhängig* von Anfragen auf das integrierte Schema in der Zieldatenbank, hier symbolisiert durch eine gestrichelte Linie.

Föderierte Systeme haben den Vorteil, dass Anfragen immer auf einen aktuellen Datenbestand erfolgen (dies erscheint intuitiv, da die fraglichen Quellen direkt angesprochen werden, und keine möglicherweise veralteten Kopien), und somit Inkonsistenzen vermieden werden können. Die Latenz ist jedoch wesentlich höher, da möglicherweise entfernte Datenquellen für die Beantwortung der Anfrage herangezogen werden müssen.

Bei schreibendem Zugriff auf Sichten, der sich generell als problematisch darstellt (View-Update-Problem, vgl. [4]), wird insbesondere bei Konsolidierung der Daten deutlich, dass ein Rückschreiben in die Quellen – falls überhaupt möglich – erheblichen Aufwand erzeugen würde, da eine Synchronisation mit den Quellen stattfinden müsste. Die Anfrageverarbeitung gestaltet sich jedoch wesentlich effizienter, da Anfragen durch Zugriff auf die lokale Datenbank beantwortet werden können. Gerade bei Data Warehouses werden die Daten häufig in nichtnormalisierten Relationen aggregiert, um teure Joins zwischen den Tabellen zu vermeiden, was sich insbesondere bei den großen Datenmengen einer solchen Datenbank in einer weiteren Effizienzsteigerung bei lesenden Anfragen niederschlägt.

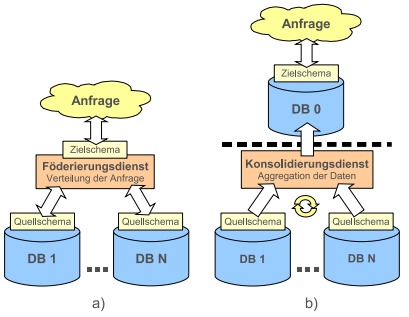

Abbildung 1. Föderierung (a) und Konsolidierung (b)

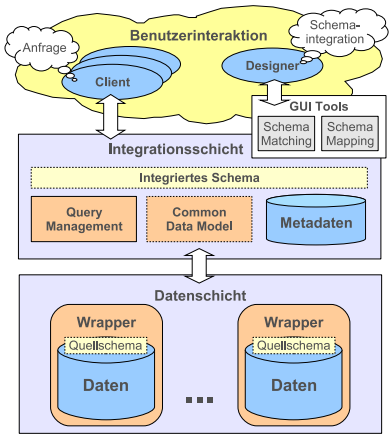

Abbildung 2. Referenzarchitektur

2.2 Referenzarchitektur

In diesem Abschnitt soll eine typische Architektur eines föderierten Integrationssystems vorgestellt werden, die für den Rest der Arbeit als Referenz dienen wird. Da es schwierig, wenn nicht gar unmöglich ist, ein Modell zu erstellen, welches allen vorgestellten Architekturen gleichermaßen gerecht wird, wird kein Anspruch auf Vollständigkeit erhoben. Die gezeigte Referenzarchitektur soll lediglich dazu dienen, die zentralen Komponenten eines typischen Integrationssystems zu identifizieren, sowie, ohne sich dabei auf ein konkretes System zu stützen, zum besseren Verständnis des Aufbaus eines solchen Systems beizutragen.

Abbildung 2 zeigt (stark vereinfacht) einen typischen Aufbau eines föderierten Integrationssystems. Es können zwei Schichten, sowie enthaltene Subsysteme identifiziert werden, die in den folgenden Abschnitten betrachtet werden.

Datenschicht. Die unterste Schicht bildet die *Datenschicht*; sie umfasst die Datenquellen, die zusammen den Datenbestand des föderierten Systems bilden. Wie eingangs schon erwähnt, können diese unterschiedliche Datenmodelle verwenden. Es werden also Softwarekomponenten benötigt, die diese adaptieren und in ein einheitliches Datenmodell überführen, so dass die Daten an der Schnittstelle mit der Integrationsschicht ausgetauscht werden können. Diese Komponenten werden auch als *Wrapper* bezeichnet. Es mag argumentiert werden, dass die Wrapper logisch ebenso der Integrationsschicht zugeordnet werden könnten. Dies ist durchaus nicht falsch, da sie eigentlich als Vermittler *zwischen* den Schichten sitzen. Im Endeffekt ist die Entscheidung der Zuordnung daher rein subjektiv. Aus Gründen der Anschaulichkeit wurde hier entschieden sie der Datenschicht zuzuordnen.

Integrationsschicht. Die Integrationsschicht bildet den Kern des Integrationssystems. Sie beinhaltet alle Komponenten und Konzepte mit denen die eigentliche Schemaintegration realisiert wird. Dazu zählt zunächst das gemeinsame Datenmodell (*CDM, Common Data Model*), in das die Quellschemas durch die entsprechenden Wrapper überführt werden und über das letztendlich die Korrespondenzen und das integrierte Schema definiert werden. Dies geschieht üblicherweise durch ein graphisches Schemaintegrations-Werkzeug, das vom Integrator bedient wird (siehe auch folgender Abschnitt). Die dazu benötigten Daten, wie etwa die exportierten Schemas aus den Wrappern, werden üblicherweise in einem Metadaten-Repository verwaltet. Für die Anfrageverarbeitung und Anfrageoptimierung ist schließlich ein Query Manager verantwortlich. Er verteilt insbesondere die ursprüngliche Anfrage auf die einzelnen Datenquellen.

Benutzerinteraktion. Obgleich keine Schicht im eigentlichen Sinne, bildet die Ebene der Benutzerinteraktion einen elementaren Bestandteil der Referenzarchitektur, da an ihr die Schnittstellen zu den Benutzern zu definieren sind. Wir unterscheiden hier zwischen zwei Benutzerrollen, und damit Arten der Interaktion: Clients können in einer vom System festgelegten *Anfragesprache* Anfragen an das System stellen (linke Seite). Die Anfragen erfolgen gegen das integrierte Schema und werden vom Query Manager weiterverarbeitet. Ein Benutzer kann auch die Rolle des Designers einnehmen, der u.a. für die Erzeugung des Zielschemas unter Verwendung eines oder mehrerer GUI-Werkzeuge verantwortlich ist (rechte Seite). Deren Funktionalität spielt eine nicht zu unterschätzende Rolle, da z.B. für das Schema Matching ein hohes Maß an Kenntnis des Benutzers über die zu integrierenden Quellen erforderlich ist, jedoch ein GUI-Werkzeug diesen Vorgang auch unterstützen muss.

2.3 Klassifizierung von Integrationssystemen

Das Hauptaugenmerk wird im weiteren Verlauf auf die Abläufe an den Schnittstellen zwischen GUI-Werkzeugen und Integrationsschicht, sowie zwischen Integrations- und Datenschicht (insbesondere den Wrappern) gelegt. Bei Betrachtungen der Integrationssysteme sollen dabei folgende Punkte besondere Beachtung finden:[3]

Common Data Model. Das gemeinsame Datenmodell, in welches die integrierten Quellen überführt werden, sowie die Sprache, um dieses zu formulieren, bilden den Kern der Integrationsschicht und sind somit von zentraler Bedeutung. Es ist also unerlässlich diese für die jeweiligen Systeme zu diskutieren, um den Vorgang der Schemaintegration nachvollziehen zu können.

Verwendetes Integrationsverfahren. Das verwendete Verfahren zur Integration trägt maßgeblich zur Charakterisierung eines Integrationssystems bei und ist

[3] Für weitere Möglichkeiten der Klassifizierung von Integrationssystemen siehe [6]

damit ein zentraler Punkt der Betrachtungen. Im weiteren Verlauf werden ausschließlich sichtenbasierte Systeme betrachtet, auch diese können jedoch weiter in ihren angewandten Verfahren unterschieden werden, von denen mit GaV und LaV in Abschnitt 1.3 schon die beiden bekanntesten Formen vorgestellt wurden. Ein weiteres Verfahren, *Both-as-View*, wird zusammen mit dem AutoMed-System (vgl. Kapitel 3.5) vorgestellt.

Grad der Automatisierung. In den meisten Systemen findet der Integrationsprozess nach wie vor weitgehend manuell statt. Die Qualität des Mappings und damit auch des resultierenden integrierten Schemas sind damit abhängig von der Expertise des Designers. Es hat daher Anstrengungen gegeben, diese Prozesse zu automatisieren. Obwohl vollautomatische Integrationssysteme laut [6] noch nicht existieren, ist das langfristige Ziel heutiger Systeme sicherlich, sich einer Vollautomatisierung so gut es geht anzunähern. Auch die hier vorgestellten Systeme unterstützen den Designer durch semi-automatisches Reasoning.

Verwendete Anfragesprache. Für den Endnutzer ist ein zentrales Kriterium letztendlich die Anfragesprache, mit der Anfragen an das föderierte System gestellt werden können. Offensichtlich ist diese eng mit dem gemeinsamen Datenmodell verknüpft, so dass sie ebenfalls mit in die Betrachtungen einbezogen werden soll.

Nachdem nun in die Grundlagen der Informationsintegration eingeführt wurde, sowie eine Referenzarchitektur und Kriterien zur Klassifizierung vorgestellt wurden, soll in den verbleibenden Kapiteln eine Auswahl an Integrationssystemen vorgestellt werden, die einen möglichst repräsentativen Schnappschuss vergangener und aktueller Forschungsanstrengungen, sowie industrieller Produkte bilden.

3 Produkte und Prototypen

Es werden nun eine Reihe von Prototypen und, falls zutreffend, darauf basierende Produkte, die am Markt erhältlich sind, vorgestellt. Die folgenden Abschnitte setzen sich dabei wie folgt zusammen:

Im umfassendsten Block werden mit Garlic und Clio zwei Forschungsprojekte für eine Informationsintegrations-Middleware bzw. ein Schema-Matching/Schema-Mapping-Werkzeug vorgestellt, deren Technologie maßgeblich in IBM-Produkte wie den DB2 Information Integrator[4] und den Rational Data Architect eingeflossen ist. Letztere werden dabei jeweils aufbauend auf den jeweiligen Prototyp vorgestellt, um zu demonstrieren, wie sich die Konzepte in einer Produktiv-Software manifestiert haben. Um das Bild abzurunden, werden in den verbleibenden Abschnitten kurz zwei weitere Prototypen vorgestellt, die durch neuartige und innovative Verfahren interessante Alternativen zu den klassischen Systemen vorschlagen.

[4] Mittlerweile WebSphere Information Integrator

3.1 Garlic

Garlic ist eine DB-Middleware zur Integration voneinander unabhängiger, möglicherweise verteilter und/oder heterogener Datenquellen unter einer einzigen logischen Sicht. Diese ermöglicht dem Benutzer, Anfragen an eine Vielzahl von verteilten und verschiedenartigen Datenquellen zu stellen, und dabei den Eindruck zu gewinnen, auf einer einzigen lokalen Datenbank zu arbeiten. Garlic erstellt aus der ursprünglichen Anfrage einzelne Anfragepläne für die registrierten Quellen und ist damit in der Lage, Anfragen auf das Zielschema durch Verknüpfung der Daten aus den Quellschemas zu beantworten. Datenquellen werden dabei über Wrapper in das Datenmodell von Garlic eingebunden, können also unterschiedlichster Natur sein. Insbesondere können über eine Wrapper-API eigene Wrapper entwickelt und eingebunden werden; als interessantes Beispiel sei hier die Einbindung des QBIC-Systems (Query by Content of Image Data) genannt, welches ermöglicht, Bilddaten anhand textueller Beschreibungen aufzufinden.[5]

Garlic verwendet ein objektorientiertes, sichtenbasiertes System, bei dem eine Benutzeranfrage auf „virtuelle“ Sichten und Objekte erfolgt, die eine Abstraktion der realen Objekte in den Quellen darstellen. Garlic folgt dem Global-as-View-Ansatz, da eine Transformation der Anfragen auf die integrierte Sicht auf Anfragen an die Datenquellen erfolgt. Benutzeranfragen erfolgen dabei über eine C++-Schnittstelle oder durch die *Garlic Query Language* (GQL), ein um objektorientierte Funktionalität erweiterter SQL-Dialekt.

Garlic wurde zu Forschungszwecken am IBM Almaden Research Center[6] im Silicon Valley in Kalifornien entwickelt, und bildet die technologische Grundlage für den DB2 Information Integrator (vgl. Abschn. 3.2).

Architektur. Abbildung 3 zeigt den konzeptionellen Aufbau des Garlic Systems. An den Wrappern findet eine Transformation der Daten in das Datenmodell von Garlic statt.[7] Dazu werden in einer auf der *Object Description Language* (ODMG-ODL) basierten Sprache, der *Garlic Definition Language* (GDL), Beschreibungen der lokalen Daten als Garlic-Objekte zur Verfügung gestellt. Die resultierenden Schemas, auch als *Repository Schemas* bezeichnet, werden anschließend durch Registrierung der Wrapper mit der Garlic-Middleware mit dem Zielschema *(Global Schema)* integriert und in einem Metadaten-Repository verwaltet. Im Herzen der Integrationsschicht sitzen ein Query Compiler, der Anfragen optimiert und Anfragepläne für die verteilten Datenquellen erstellt, sowie eine Query Execution Einheit, die für die Ausführung und Übermittlung der Anfrageergebnisse zuständig ist.

[5] Eine Beschreibung des QBIC-Systems und seine Einbindung in Garlic findet sich in [10], es soll hier jedoch nicht weiter betrachtet werden.

[6] http://www.almaden.ibm.com/

[7] Das Garlic-Objektmodell basiert auf dem ODMG-93-Standard [12], wurde jedoch um einige Konzepte erweitert, wie die Unterstützung mehrerer Interface-Implementierungen und objektzentrierte Sichten [9].

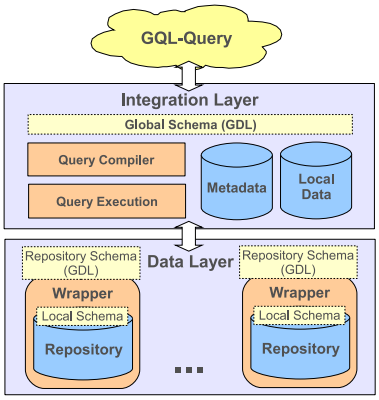

Abbildung 3. Garlic Architektur

```
create table Student (
    Matnr char(6) primary key,
    Name varchar(30) not null,
    Vorname varchar(30) not null,
    Semester integer,
    FB integer references
        Fachbereich(Fbnr)
        not null
);

create table Fachbereich (
    Fbnr integer primary key,
    Name varchar(30) not null
);
```

Abbildung 4. Ausschnitt einer fiktiven Universitätsdatenbank als Eingabe für einen relationalen Garlic-Wrapper

```
interface Student_Type {
    attribute string Matnr;
    attribute string Name;
    attribute string Vorname;
    attribute long Semester;
    attribute ref<FB_Type> FB;
};

interface FB_Type {
    attribute long Fbnr;
    attribute string Name;
};
```

Abbildung 5. Mögliche Definition der Relationen aus Abb. 4 in der GDL als Ausgabe eines relationalen Garlic-Wrappers

Abbildung an den Wrappern. Wie bereits erwähnt, müssen zunächst die Daten in den Quellen durch einen Wrapper auf Garlic-Objekte abgebildet werden. Die Definition der Garlic-Typen erfolgt dabei über *Interface-Definitionen* in der GDL. Dabei müssen bereits erste Abbildungsentscheidungen getroffen werden, wie etwa das Umbenennen von Attributen, die Umwandlung von Datentypen, sowie die generelle Entscheidung, wie Entitäten des lokalen Datenmodells auf Objekte und Attribute der GDL zu mappen sind. In einem relationalen System (ein unproblematischer Fall) liegt es beispielsweise nahe, Tupel Objekten und Spalten Attributen zuzuordnen. Die Beispiele in Abb. 4 und 5 zeigen, wie eine Umwandlung einer Tabelle in einen GDL-Typ durch einen relationalen Wrapper aussehen kann.

Für die neu definierten Interfaces müssen anschließend Implementierungen angelegt werden, die die Zuordnung der neu angelegten Typen zu den korrespon-

dierenden Typen im zugrundeliegenden Quelldatenmodell festlegen. Es können dabei durchaus mehrere Implementierungen für dasselbe Interface angelegt werden, die über einen *Implementation Identifier (IID)* in den Wrappern eindeutig identifizierbar sind. Es stellt sich jedoch offensichtlich das Problem, dass Garlic-Objekte eine globale Identität besitzen und damit eindeutig identifizierbar sein müssen. Daher ist es Aufgabe der Wrapper und der Middleware einen global eindeutigen *Garlic-Object-Identifier (OID)* für jedes Objekt zu erzeugen, der sich nach [8] aus zwei Teilen zusammensetzt:

Den ersten Teil bildet der oben erwähnte IID, der festlegt, welche Implementierung für das Garlic-Objekt zu verwenden ist. Diese wiederum legt fest, welches Interface das Objekt unterstützt und wo dieses im Wrapper zu finden ist. Der zweite Teil, der *Schlüssel*, identifiziert ein Objekt eindeutig in der jeweiligen Implementierung. Der OID ergibt sich dann durch Konkatenation des IID und des Schlüssels. Als Beispiel könnte für die Erzeugung eines OID für ein Objekt des Typs `Student_Type` aus Abb. 5 der IID der Implementierung `Student` aus Abb. 4 und das Attribut `Student.Matnr` als Schlüssel herangezogen werden.

Nachdem die Quelldaten als Garlic-Objekte modelliert wurden, können diese in Form von *Collections* exportiert werden, um später in GQL-Anfragen verwendet zu werden.

Abbildung in der Middleware. In der Middleware können anschließend durch Definition sog. *objektzentrierter Sichten (object-centered views)* „virtuelle“ Objekte angelegt werden, die eine Erweiterung, Umformung oder Vereinfachung einer Menge zugrundeliegender Garlic-Objekte darstellen. Dazu werden wie zuvor Interface-Definitionen und Implementierungen angelegt, die basierend auf den an den Wrappern exportierten Interfaces die Typen der virtuellen Objekte beschreiben. Virtuelle Objekte sind immateriell (daher ihr Name), da sie lediglich als Anfragen über die exportierten Objekte der Wrapper implementiert sind. Die OIDs der virtuellen Objekte werden dabei durch einen Vorgang, der als *Lifting* bezeichnet wird, vom OID des zugrundeliegenden Basisobjektes abgeleitet. Dazu werden der IID der objektzentrierten Sicht und der OID des Basisobjektes konkateniert.

Aufbauend auf dem vorherigen Beispiel könnten zusätzlich auf einer Webseite des Prüfungsamtes Studenten mit ihren Prüfungsnoten aufgelistet sein und durch einen HTML-Wrapper ein Interface `Zensuren` mit den Attributen `Matnr`, `Fach` und `Note` exportiert worden sein[8]. Abb. 6 zeigt eine Interface-Definition eines virtuellen Objektes in der Middleware, sowie die dazugehörige Sichtdefinition, die den Typ `Zensuren` mit dem Typ `Student` zusammenführt. Das Schlüsselwort `self` dient dabei zur Identifizierung und Referenzierung des Objektes bei der Definition von Methoden[9], ähnlich dem impliziten Parameter `this`, wie er aus objektorientierten Sprachen bekannt ist.

[8] Analog zu Abb. 5

[9] Methoden werden hier nicht weiter beschrieben; Beispiele finden sich in [7].

```
interface InfStatistik_Type {
    attribute string Matnr;
    attribute long Semester;
    attribute double Schnitt;
};

create view InfStatistik (Matnr, Semester, Schnitt, self)
as select S.Matnr, S.Semester, avg(Z.Note),
    LIFT('InfStatistik', S.OID)
from Student S, Zensuren Z
where S.fb->Name = 'Informatik' and S.Matnr = Z.Matnr
group by S.Matnr, S.Semester
```

Abbildung 6. Typdefinition und Implementierung eines virtuellen Objektes in der Middleware durch eine Sicht über zwei unabhängige, möglicherweise heterogene Datenquellen. `Student` und `Zensuren` sind Collections, die aus den Wrappern exportiert wurden.

Anfrageplanung. Das Ziel der Anfrageplanung ist es, mehrere alternative Pläne für eine Benutzeranfrage zu erzeugen und den (kosten)effizientesten auszuwählen. Garlic geht dabei bottom-up vor und erzeugt verschiedene Arten von Plänen. Zuerst werden Zugriffspläne für einzelne Collections erzeugt (*Single Collection Access Plans*), dann Pläne für 2-Wege-Joins, 3-Wege-Joins, usw. (*Join Plans* und *Bind Plans*). Der Ablauf der Planerstellung für die jeweiligen Pläne wird basierend auf [8] in den folgenden Abschnitten erläutert.

Erstellen von Single Collection Access Plans. Aus der ursprünglichen Anfrage werden für jedes beteiligte Repository die jeweiligen Teile der Anfrage extrahiert und an den verantwortlichen Wrapper geschickt (*Work Request*). Dieser kann daraufhin für jede involvierte Collection einen *Single Collection Access Plan* erstellen, der den Zugriff auf die Collection teilweise oder vollständig realisiert, und schickt diesen zurück an Garlic, zusammen mit den assoziierten Kosten. Es sei hierbei erwähnt, dass die unterschiedlichen Datenquellen offensichtlich in ihrer Ausdrucksmächtigkeit (und damit ihrer Fähigkeit eine Anfrage zu beantworten) zum Teil stark variieren. Es kann also durchaus vorkommen, dass eine Quelle nicht in der Lage ist einen bestimmten Teil der Anfrage umzusetzen; ein Wrapper für eine Webseite ist z.B. nicht in der Lage, eine Join-Operation durchzuführen. Fehlende Teile der Anfrage kompensiert Garlic daher durch *Plan-Operatoren (POPs)* wie `PROJECT`, `JOIN`, `FETCH`[10] und `FILTER`[11], wie beschrieben in [11]. Dazu fordert Garlic bei einem Work Request beim jeweiligen Wrapper immer *alle* in die Anfrage involvierten Attribute an, auch solche, die in `where`-Klauseln vorkommen, inklusive der OIDs der betroffenen Objekte. Kann nämlich ein Wrapper z.B. ein bestimmtes Prädikat einer `where`-Klausel nicht erfüllen (etwa weil er einen String-Vergleich mit `like` nicht unterstützt), so liefert er auf diesem Wege

[10] Anfragen von Attributwerten an der Quelle über Getter-Funktionen
[11] Anwenden von Prädikaten

zumindest die nötigen Attribute, damit Garlic den Vergleich durch Anwendung eines POP (in diesem Falle `FILTER`) ergänzen kann.

Erstellen von Join Plans. Stellt Garlic fest, dass zwei oder mehr Attribute der ursprünglichen Anfrage im selben Repository liegen, versucht es beim verantwortlichen Wrapper einen *Join Plan* anzufordern. Dazu schickt es die zuvor ermittelten Zugriffspläne zusammen mit den Join-Prädikaten zurück an den Wrapper, der daraufhin – wiederum nicht zwingend – einen Join Plan erstellen kann. Dazu mappt der Wrapper den Work Request auf einen Join über die adaptierte Datenquelle, kapselt die involvierten Attribute des Joins im Join Plan, und schickt diesen zusammen mit den Join-Prädikaten, die er unterstützt (im Falle eines relationalen Wrapper i.d.R. alle), sowie den Kosten des Joins, zurück an Garlic. Für n-Wege Joins wird dieser Prozess entsprechend oft wiederholt.

Erstellen von Bind Plans. Bei Joins über Collections, die aus unterschiedlichen Datenquellen stammen, muss der Join offensichtlich von Garlic selber durchgeführt werden, und somit die für den Join benötigten Daten aus den Wrappern zur Middleware transportiert werden. Dabei ist es wünschenswert, die Menge der benötigten Daten möglichst klein zu halten, um sowohl die Kosten des Datentransfers als auch des globalen (quellenübergreifenden) Joins zu minimieren. Der Garlic-Optimizer kann daher während der Join-Phase entscheiden, anstelle eines gewöhnlichen Joins einen *Bind Join* durchzuführen, um die Menge der sich für den globalen Join qualifizierenden Tupel zu verkleinern. Dazu wird der ursprüngliche Join Plan zusammen mit einem zusätzlichen *Bind-Join-Prädikat* zurück an den Wrapper geschickt, der daraus einen *Bind Plan* erstellt and an Garlic übergibt. Der globale Join arbeitet durch die zusätzliche Einschränkung durch das Bind-Join-Prädikat dann auf weniger Tupeln als zuvor, und kann somit effizienter ablaufen.

Erstellen des Anfrageplans. Ein vollständiger Anfrageplan wird schließlich als Baum von Operatoren erzeugt, mit den Plänen der Wrapper als Blätter. Abb. 7 illustriert einen solchen Baum für eine Anfrage über die Collections `Student`, `Fachbereich` und `Zensuren`. Dabei wird angenommen, dass der HTML-Wrapper für die Webseite des Prüfungsamtes Vergleiche mit SQL-`like`-Semantik nicht unterstützt; dieses Prädikat taucht daher im Zugriffsplan für `Zensuren` nicht auf. Es bleibt also der Middleware überlassen, einen entsprechenden `FILTER` einzufügen. Der Join zwischen `Student` und `Fachbereich` ist an der Quelle möglich, daher liefert der relationale Wrapper einen passenden Join Plan. Der Join über `Student` und `Zensuren` ist offensichtlich nur an der Middleware möglich, daher wird ein `JOIN`-POP eingefügt. Abschließend werden mit `PROJECT` die Attribute der Anfrage herausprojiziert.

Anfrageausführung. Nachdem ein Anfrageplan vorliegt und ausgewählt wurde, muss dieser ausgeführt werden. Die Anfrageausführung gliedert sich dabei in zwei Schritte:

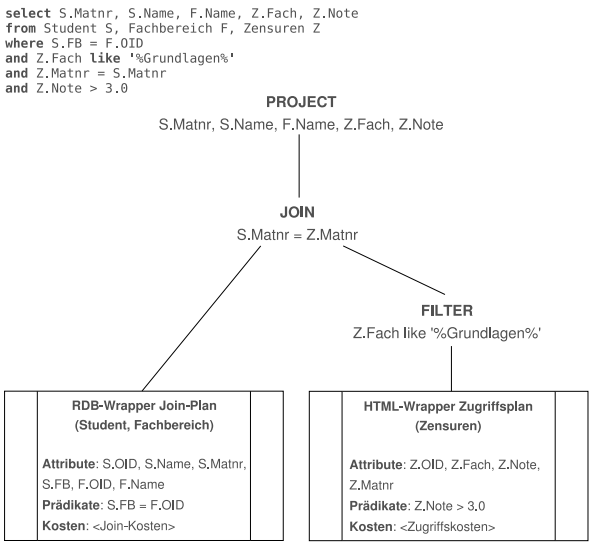

Abbildung 7. Ein vollständiger Anfrageplan als Operatorbaum mit den Wrapper-Plänen als Blättern. Die Anfrage liefert eine Menge von Tupeln für Studenten, die in Grundlagenfächern schlecht abgeschnitten haben.

Übersetzung des Anfrageplans. Für die Übersetzung eines Anfrageplans werden alle Operatoren in *Iteratoren* umgewandelt. Für die Iteratoren der Pläne an den Blättern sind die Wrapper zuständig; die Middleware ruft dann einfach an der Wrapper-Schnittstelle eine Methode auf, die diese Übersetzung implementiert. Das erzeugte Iterator-Objekt wird dann zurück an die Middleware geliefert. Der relationale Wrapper des Beispiels übersetzt den Plan dabei in `select-from-where`-Anfragen an die Quell-Relationen und speichert den enstandenen Query-String im Iterator (OIDs werden dabei in Primary-Key-Attribute konvertiert, wie weiter oben beschrieben). Der HTML-Wrapper könnte die Anfrage entsprechend in einen Query-URL übersetzen, den die Webseite versteht.

Ausführung der Anfrage. Die eigentliche Ausführung der Anfrage findet über Methoden der Iteratoren statt (`advance`, `reset`, u.a.), die vom Wrapper zu implementieren sind. So können schrittweise Werte ausgelesen oder der Iterator zurückgesetzt werden, um die Anfrage erneut auszuführen. Beim relationalen Wrapper des Beispiels wird durch den `reset`-Aufruf die Verbindung zur Datenbank vorbereitet und mittels `advance` werden die Ergebnisse tupelweise ausgelesen. Dabei werden Schlüsselwerte in OIDs konvertiert, da die Middleware diese erwartet. Entsprechend lädt der HTML-Wrapper bei `reset` die benötigte Webseite. Durch `advance` kann ein Parser dann die einzelnen Werte auslesen, wobei auch hier die Schlüssel in OIDs konvertiert werden.

3.2 IBM DB2 Information Integrator

Es folgt ein Überblick über den *IBM DB2 Information Integrator V8.1* (DB2II) wie beschrieben in [13]. Im Gegensatz zum vorherigen Kapitel soll diesmal der Fokus auf der Benutzerinteraktion mit dem System liegen, eine detaillierte „under-the-hood"-Betrachtung erfolgt somit nicht. Da jedoch das technologische Wissen aus dem Garlic-System bei der Entwicklung des DB2II zur Anwendung kam [14], sollten die Parallelen erkennbar sein.

Mit dem DB2 Information Integrator bietet IBM seit einiger Zeit eine komplette Lösung zur föderierten Integration heterogener und verteilter Datenquellen an. Basierend auf der DB2 Universal Database und Teilen der Garlic-Technologie wurden die Möglichkeiten zur Integration sowohl relationaler als auch nicht-relationaler Datenquellen stark ausgebaut, so dass Wrapper u.a. für folgende Systeme zur Verfügung stehen:

Relationale Wrapper: Informix, ORACLE, Sybase, Microsoft SQL Server, Teradata

Nicht-relationale Wrapper: Plain-Text, Excel Spreadsheets, XML, BLAST, Documentum

Schreibzugriff wird dabei für relationale Quellen voll unterstützt; auf nicht-relationale Quellen kann nur lesend zugegriffen werden. Analog zum Garlic-System erlaubt ein durch den DB2II föderiertes System Joins über Attribute der Quellen, als wären sie lokal gespeichert, Ausnutzung der an den Quellen lokal vorhandenen Funktionalität zur Verarbeitung von Anfragen *(„pushdown")*, sowie Maßnahmen zur Kompensation für fehlende oder zu teure Operationen. Zusätzlich bietet der DB2II Funktionen zum Replizieren von Daten, zur Materialisierung von Anfragen in Tabellen, sowie eine graphische Benutzerschnittstelle zur Konfiguration des föderierten Systems. Ein durch Einsatz des DB2II föderiertes System wird im Folgenden auch als *föderiertes DB2-System* bezeichnet.

Architektur. Die Architektur eines föderierten DB2-Systems ist grundsätzlich sehr ähnlich zu der des Garlic-Systems, wie gezeigt in Abb. 3. Die Begrifflichkeiten weichen jedoch an manchen Stellen ab, und werden daher hier kurz erläutert:

Ein föderiertes DB2-System besteht aus einer DB2-Instanz, einem globalen Katalog, Servern und deren Wrappern. Ein *Server* repräsentiert eine spezifische Datenquelle, auf die durch einen Wrapper zugegriffen werden kann, und entspricht dem was im Garlic-System als Repository bezeichnet wurde. Die Objekte eines Servers werden dem System dabei durch sog. *Nicknames* bekannt gemacht, die als Aliasnamen dienen und dem Benutzer zur Verwendung in Anfragen zur Verfügung stehen. Im *globalen Katalog* (entsprechend dem Metadaten-Repository in Garlic) werden schließlich alle Verwaltungsinformationen gespeichert. Dazu zählen u.a. die am System registrierten Server und Wrapper, die Mappings der Nicknames auf Elemente in den Servern sowie *User Mappings*, mit denen Authentifizierungsdaten für das föderierte System auf solche in den

einzelnen Servern abgebildet werden. Der globale Katalog wie auch lokale Tabellen werden in einer IBM DB2 Universal Database (DB2 UDB) verwaltet, hier auch als *föderierte Datenbank* bezeichnet. Alle Komponenten mit Ausnahme der Wrapper und der Server bilden zusammen einen sog. *föderierten Server*. Es können insbesondere auch existierende DB2-Instanzen durch Erweiterung mit dem DB2II als föderierte Server betrieben werden.

Integration mit dem DB2II. Um beim Vergleich mit dem Garlic-System zu bleiben, soll erwähnt sein, dass im verwendeten Datenmodell und der Definitionssprache der größte Unterschied in den Systemen liegt. Da einem föderierten DB2-System eine DB2 zugrundeliegt, müssen die Wrapper ihre Daten im Datenmodell der DB2 ausdrücken, also in Form von Relationen. Um eine Datenquelle mit dem DB2II in das föderierte DB2-System zu integrieren, müssen folgende Schritte durchgeführt werden:

1. Registrierung des Wrapper-Moduls der einzubindenden Quelle
2. Anmeldung der neuen Datenquelle als Server in der föderierten Datenbank
3. Anlegen von User Mappings für jeden DB2-Benutzer, der die Quelle benutzen darf
4. Testen der neuen Verbindung durch eine *Passthru-Session*, in der SQL-Statements direkt an die neue Quelle geschickt werden
5. Falls erforderlich oder gewünscht, Anlegen zusätzlicher Mappings zwischen den Datentypen der DB2 und der Quelle
6. Anlegen von Nicknames für jedes Objekt der Quelle

Die Schritte 1, 2, und 6 werden nun am Beispiel eines XML-Servers illustriert, der mit einem föderierten Server integriert werden soll. Dabei ist hervorzuheben, dass die Abbildungen von XML-Elementen auf relationale Elemente vollständig durch den Benutzer konfigurierbar sind, und so für die jeweilige Situation eine möglichst genaue und korrekte Adaptierung ermöglichen. Alle Schritte werden hier in Form von SQL-Befehlen an den föderierten Server beschrieben, können jedoch ebenfalls über die GUI durchgeführt werden.

Definieren eines Wrappers. Wrapper-Implementierungen werden dem föderierten Server über *Wrapper-Module* zur Verfügung gestellt. Ein Wrapper-Modul stellt dabei alle benötigten Routinen zur Verfügung, die der föderierte Server benötigt, um z.B. eine Verbindung zur gekapselten Datenquelle aufzubauen und Daten auszulesen. Die Daten in den Quellen werden dabei in Form von Tabellen exportiert; welche Operationen auf den Tabellen erlaubt sind, hängt vom Typ der Datenquelle ab. Die Definition eines XML-Wrappers sieht dabei wie folgt aus:

```
CONNECT TO <federated_db_name>;
CREATE WRAPPER XML LIBRARY 'libdb2lsxml.a'
```

Müssen Datenquellen integriert werden, für die keine vorgefertigten Wrapper-Module vorliegen, können entsprechende Bibliotheken unter Verwendung eines *Wrapper Development Kits* erzeugt werden.

Definieren eines Servers. Nachdem ein passender Wrapper für die zu integrierende Datenquelle angelegt ist, muss diese als Server dem System bekannt gemacht werden, um für Anfragen zur Verfügung zu stehen. Ist die Datenquelle ein relationales DBVS, das mehrere Datenbanken pro Instanz verwaltet, so kann jede einzelne Datenbank als Server registriert werden. Bei nicht-relationalen Quellen, um beim Beispiel von XML zu bleiben, erfolgt die Registrierung über ein einziges Server-Objekt:

```
CONNECT TO <federated_db_name>;
CREATE SERVER MY_XML_SERVER WRAPPER XML
```

Bei relationalen Datenquellen können zusätzlich Server-Optionen durch die `OPTIONS`-Klausel spezifiziert werden.

Erzeugen von Nicknames. Wie eingangs erwähnt, dienen Nicknames der Identifizierung und Referenzierung von Objekten in den Datenquellen. Sie ermöglichen somit *ortstransparenten Zugriff* auf Objekte der Datenquellen, da sie bei Anfragen an den föderierten Server wie lokale Tabellen verwendet werden können. Dies impliziert, dass eine Qualifizierung eines Quelldaten-Objektes in einer Anfrage an den föderierten Server mit der Datenquelle, auf dem es gespeichert ist, *nicht* stattfinden muss. Bei der Erzeugung von Nicknames für Objekte relationaler Datenquellen kann ein direktes Mapping auf Tabellen erfolgen; bei semi- oder unstrukturierten Quellen gestaltet sich dies etwas schwieriger. Bei XML-Quellen können über die `OPTIONS`-Klausel dazu XPath-Ausdrücke verwendet werden (es wird hierbei angenommen, dass `studenten.xml` eine Menge von `student`-Knoten mit den Attributen `matnr`, `name` und `vname` enthält):

```
CONNECT TO <federated_db_name>;
CREATE NICKNAME XMLSCHEMA.STUDENT (
    MATNR CHAR(6) NOT NULL OPTIONS(XPATH './matnr/text()'),
    NAME VARCHAR(30) NOT NULL OPTIONS(XPATH './name/text()'),
    VORNAME VARCHAR(30) NOT NULL OPTIONS(XPATH './vname/text()')
    FOR SERVER "MY_XML_SERVER"
    OPTIONS(XPATH '//student', FILE_PATH '/exchange/xml/studenten.xml')
```

Nachdem alle Datenquellen vollständig registriert sind, kann unter Verwendung eines Schema-Matching-/Schema-Mapping-Werkzeuges ein sog. *föderiertes Schema* (das Zielschema) angelegt werden, das die Datenquellen miteinander integriert. Ein Tool welches dabei direkt mit einem föderierten DB2-System zusammenarbeitet ist der Rational Data Architect, der in Abschnitt 3.4 beschrieben wird.

3.3 Clio

Bislang wurde nur aufgezeigt, wie existierende Datenquellen durch Schema- und Datentransformation an den Wrappern und in der Middleware kompatibel gemacht werden können, um sie miteinander zu verknüpfen. Wie in Kapitel 1 schon erläutert, kann es jedoch semantische Diskrepanzen zwischen den exportierten Schemas geben, die es aufzulösen gilt. So könnte es etwa sein, dass die Attribute `Matnr` der Objekte `Student` und `Zensuren` des Beispiels aus Abschnitt

3.1 dieselben realweltlichen Mengen beschreiben, jedoch unterschiedlich benannt sind[12].

Clio ist ein Werkzeug, mit dem *Matchings* zwischen den Quellschemas semi-automatisch ermittelt werden können, um solche Konfliktsituationen zu behandeln. Es erlaubt außerdem, unter Berücksichtigung gefundener Korrespondenzen, durch Definition von Anfragen über die Quellen *Mappings* zu erzeugen, um die Quellschemas in einem Zielschema zu integrieren. Die erforderlichen Abläufe unterstützt Clio durch eine grafische Oberfläche, in der Referenzarchitektur sitzt Clio also als GUI-Werkzeug auf der Integrationsschicht. Wie auch Garlic wurde Clio am IBM Almaden Research Center (in Zusammenarbeit mit der Universität von Toronto) entwickelt, und ist technologische Grundlage für den *Rational Data Architect* (vgl. Abschn. 3.4).

Architektur. Es folgt eine Beschreibung der Clio-Architektur basierend auf [17]. Die Herzstücke des Clio-Systems sind die Komponenten zur Mapping- und Anfragegenerierung. Erstere erzeugt auf Basis der zwischen Quell- und Zielschema gefundenen Korrespondenzen eine Menge von *logischen Mappings* in Form von deklarativen *Constraints*. Sie sind Grundlage für das *physische Mapping*, das anschließend durch die Anfragegenerierung in Form ausführbarer Scripts erzeugt wird. Zur Formulierung der Anfragen unterstützt Clio SQL, SQL/XML, XQuery und XSLT. Die Korrespondenzen zwischen den Quellschemas werden semi-automatisch ermittelt (s.u.), entweder durch Clios eigene Schema-Matching-Algorithmen, oder durch externe Matching-Module. Zu jedem Zeitpunkt gilt, dass der Benutzer über Clios GUI eingreifen und die vom System gemachten Vorschläge anpassen und verbessern kann.

Ein Überblick über die Architektur von Clio findet sich in Abb. 8.

Schemaintegration mit Clio. In den folgenden Abschnitten wird der Ablauf einer Schemaintegration mit Clio beschrieben, sowie erläutert, welche Mechanismen dabei zur Anwendung kommen.

Importieren von Schemas. Bevor eine Verarbeitung der Datenquellen vorgenommen werden kann, müssen die zu integrierenden Schemas importiert werden. Diese können dabei nach [16] aus einer objektrelationalen Quelle, einer durch einen objektrelationalen Garlic-Wrapper adaptierten Quelle, oder einer XML-Datei samt Schemadefinition gelesen werden. Da die Schemas zur Weiterverarbeitung homogen sein müssen, transformiert Clio sie in eine interne Repräsentation, die sowohl für relationale als auch XML-basierte Datenquellen geeignet ist. Die Ergebnisse werden dem Benutzer zur Kontrolle graphisch angezeigt, und können manuell angepasst werden.

[12] Nach Abschnitt 1.4 läge also ein Beschreibungskonflikt in den abgebildeten Miniwelten vor.

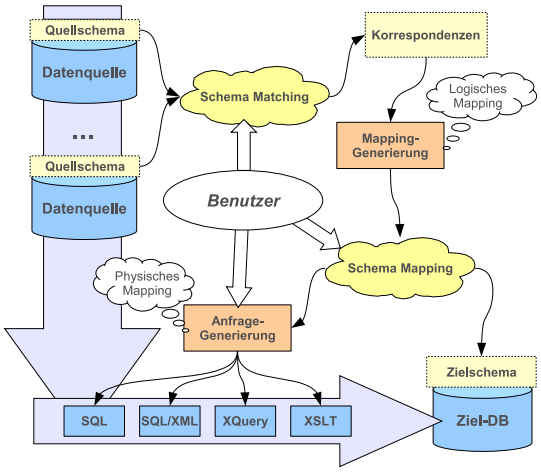

Abbildung 8. Clio Architektur

Wertbasierte Korrespondenz. Als Mapping-Vorschrift zwischen zwei Schemaelementen verwendet Clio sogenannte wertbasierte Korrespondenzen (engl.: *value correspondences*) [15]. Eine wertbasierte Korrespondenz ist dabei ein Paar aus einer *Funktion* f, die beschreibt, *wie* Werte aus den Datenquellen benutzt werden können, um Werte im Zielschema zu bilden, sowie einem *Filter* F, der festlegt, *welche* Quelldaten sich für die Abbildung qualifizieren[13].

Bei der Definition solcher Korrespondenzen versucht Clio den Benutzer zu unterstützen, indem es durch linguistische Verfahren aus Daten und Metadaten (z.B. Attributnamen von Relationen) mögliche Korrespondenzen ableitet. Hinweise zieht Clio dabei aus Fremdschlüsselbeziehungen, Synonymen bei Bezeichnern, oder den Daten selbst. Es können, wie oben bereits erwähnt, jedoch auch externe Komponenten zum erweiterten Schema Matching herangezogen werden. Der Designer hat dann die Möglichkeit auf Grundlage der von Clio vorgeschlagenen Mapping-Vorschrift Korrekturen oder Verfeinerungen vorzunehmen (häufig sind die Beziehungen zwischen Schemaelementen zu komplex, um von Clio eindeutig bestimmt werden zu können).

Ein Beispiel soll dies verdeutlichen: Es wird nun angenommen, dass die in Abb. 6 formulierte Sicht `InfStatistik` auf alle Fachbereiche verallgemeinert werden soll (im Folgenden: `Statistik`). Es müssen also weitere Datenbanken mit Zensuren aus anderen Fachbereichen herangezogen werden, sowie die Sichtdefinition erweitert werden. Eine dieser Datenbanken, etwa des Fachbereiches Biologie, speichert die Noten der Studenten dieses Fachbereiches, jedoch schon unterteilt in Notenschnitte aus dem Grundstudium und Notenschnitte aus dem Hauptstudium (im Folgenden: `Zensuren'`). Es liegt offensichtlich ein Konflikt

[13] Dies geschieht durch Formulierung eines Boole'schen Ausdrucks; falls nicht explizit angegeben, wird im Folgenden der Filter $F = true$ angenommen.

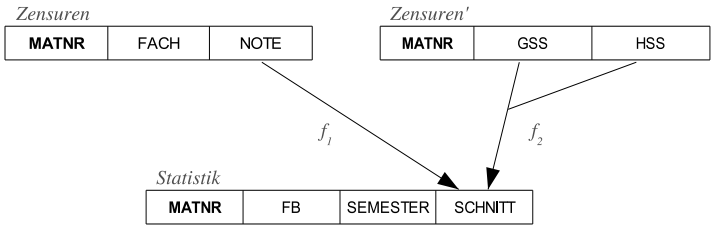

Abbildung 9. Zusammenführen zweier konfligierender Attribute in ein Zielschema über wertbasierte Korrespondenzen

vor: Noten für einzelne Fächer aus dem Fachbereich Informatik müssen in der gemeinsamen Sicht erst zu einem Schnitt umgerechnet werden, wogegen die Noten aus dem Fachbereich Biologie schon als Schnitte, aber getrennt für Grund- und Hauptstudium, vorliegen. Die neue Situation ist in Abb. 9 illustriert. Dieser Zusammenhang ist offensichtlich zu komplex, um von Clio erkannt zu werden; Clio könnte zwar durchaus erkannt haben, dass eine Beziehung besteht, jedoch nicht, wie die Abbildung zu formulieren ist.

Zur Lösung des Integrationskonfliktes könnten nun vom Designer folgende Korrespondenzen definiert werden:

$$f_1 : avg(Zensuren(Note)) \rightarrow Statistik(Schnitt)$$

Wie schon in `InfStatistik`, kann durch eine Abbildung der Noten aus `Zensuren` mit der Durchschnittsfunktion eine Korrespondenz zum Attribut `Schnitt` in der `Statistik`-Sicht hergestellt werden. Ähnlich lässt sich auch eine Korrespondenz zwischen den Noten aus `Zensuren'` und `Schnitt` definieren:

$$f_2 : (Zensuren'(GSS) + Zensuren'(HSS)) * 0.5 \rightarrow Statistik(Schnitt)$$

Die neu definierten Korrespondenzen dienen dann als Eingabe für Clios Schema-Mapping-Engine.

Schema Mapping durch Formulierung von Anfragen. Clio erzeugt Mappings zwischen zwei Schemas durch Formulierung von Anfragen, die die Quelldaten in die Form des Zielschemas übersetzen, unter Berücksichtigung der zuvor gesammelten Information über die Schemas und deren Korrespondenzen. Um dem Benutzer möglichst sinnvolle Vorschläge zu machen, folgt Clio zwei Grundprinzipien zur Ermittlung eines Mappings:

1. Alle Werte der Quelle sollten nach Möglichkeit auch in der Zielmenge erscheinen
2. Ein Wert der Quelle sollte nach Möglichkeit nur einmal in der Zielmenge erscheinen

Basierend auf diesen Prinzipien und den vom Benutzer definierten Korrespondenzen wird so ein initiales Mapping erzeugt, das der Integrator nach Wunsch modifizieren kann. Ein Beispiel für die Anwendung dieser Prinzipien ist die Entscheidung in einer Anfrage ein `union` zu verwenden anstatt eines Joins, um alle Kombinationen der Quelldaten zu erhalten.

Bei der Erzeugung der Mappings geht Clio nach [15] vier-phasig vor (es wird für diesen Abschnitt angenommen, dass das Zielschema relational ist): In der ersten Phase werden für jede Zielrelation Gruppen aus Korrespondenzen gebildet, und zwar so, dass jede Gruppe *eine* Möglichkeit repräsentiert die Attribute der Zielrelation aus Attributen der Quellrelationen zu erzeugen. Diese Gruppen müssen nicht zwingend vollständig sein, es muss also nicht für jedes Zielattribut eine Korrespondenz existieren. Auch Disjunktheit der Gruppen wird nicht gefordert, es darf aber für jedes Zielattribut höchstens eine Korrespondenz in jeder Gruppe enthalten sein. Jede solche Gruppe wird auch *potentielle Kandidatenmenge* genannt.

In der zweiten Phase des Algorithmus werden diejenigen potentiellen Kandidatenmengen eliminiert, die nicht (oder nur schlecht) geeignet sind, um daraus Anfragen zu erzeugen. So wird es wahrscheinlich vorkommen, dass bei den Korrespondenzen einer potentiellen Kandidatenmenge mehr als eine Quelle involviert ist, so dass eine vertikale Komposition vorgenommen werden muss (in diesem Falle durch Joins). Dabei versucht Clio möglichst effiziente Joins zu erzeugen, also solche über N:1-Beziehungen, wie es bei Fremdschlüsselbeziehungen der Fall ist. Gibt es mehrere Möglichkeiten, kann der Benutzer zwischen den verschiedenen Join-Pfaden auswählen. Lässt sich hingegen kein solcher finden, wird die potentielle Kandidatenmenge von weiteren Überlegungen ausgeschlossen. Als Ausgabe der zweiten Phase erhält man somit eine Menge von echten *Kandidatenmengen*.

In der dritten Phase versucht der Algorithmus Untermengen der Menge der Kandidatenmengen zu finden, die eine minimale und vollständige Überdeckung der Menge der gesamten Korrespondenzen (wie initial definiert) darstellen. Überdeckungen, die vollständig, aber nicht minimal sind, werden entfernt. Die verbleibenden Überdeckungen werden dann anhand der Anzahl der involvierten Kandidatenmengen bewertet, wobei offensichtlich ist, dass Überdeckungen mit wenigen Kandidatenmengen bevorzugt werden, da sie weniger Anfragen und damit ein einfacheres Mapping erzeugen. Gibt es davon ebenfalls mehrere, wird diejenige Überdeckung ausgewählt, die die meisten Zielattribute erreicht, und damit die wenigsten `null`-Werte erzeugt.

In der letzten Phase werden die Anfragen erzeugt. In Falle eines relationalen Mappings werden die Korrespondenz-Funktionen f_i für die `SELECT`-Klausel herangezogen, die Filter F_i für die `WHERE`-Klausel, sowie alle involvierten Relationen der Quellen für die `FROM`-Klausel, wobei die in Schritt 2 definierten Join-Pfade berücksichtigt werden. Wurden in den Korrespondenz-Funktionen oder den Filtern Aggregatfunktionen verwendet, werden diese durch `GROUP BY`- und `HAVING`-Klauseln realisiert. Die resultierenden Anfragen für jede Kandidatenmenge werden schließlich durch ein `UNION ALL` vereinigt.

3.4 IBM Rational Data Architect

Wie im Falle DB2II und Garlic existiert mit dem *Rational Data Architect (RDA)* von IBM eine Business-Lösung auf Basis von Clio. Der RDA ist ein Datenmodellierungs- und Integrationswerkzeug, das es ermöglicht, Datenbanken zu annotieren und dokumentieren, visuelle DB-Modelle zu erstellen, sowie Beziehungen zwischen den Daten zu erkennen und daraus Transformationen zu erzeugen, mit denen sich ein integriertes Schema anlegen lässt. Der RDA verwendet populäre Technologien wie die Eclipse-Plattform[14] als graphische Umgebung, sowie Java Database Connectivity (JDBC)[15], um sich zu den Datenquellen zu verbinden und mit diesen zu kommunizieren. Es sei hierbei noch kurz erwähnt, dass nicht notwendigerweise bereits existierende Datenbanken zur Modellierung herangezogen werden müssen. Der RDA erlaubt auch die Erstellung *logischer Modelle*, die nicht auf physischen Quellen beruhen und beim Entwurf neu anzulegender Datenbanken verwendet werden können.

Im Folgenden wird basierend auf [18] und [19] der Ablauf einer typischen Sitzung mit dem RDA beschrieben. Um mit dem RDA von einer Menge von Datenquellen mit heterogenen Schemas zu einem integrierten Zielschema zu gelangen, lassen sich dabei fünf Schritte identifizieren:

1. Annotieren der zu integrierenden Quellschemas
2. Definieren von Matching zwischen den Quellschemas
3. Modellierung des integrierten Schemas (Zielschema)
4. Definieren von Matchings zwischen Quellschemas und Zielschema
5. Erzeugen der Mappings als SQL-Statements

Die jeweiligen Schritte werden in den folgenden Abschnitten beschrieben.

Annotation der Quellschemas. Dieser Schritt erfordert umfassendes Wissen des Designers über die zu integrierenden Quellen und muss zum Großteil manuell durchgeführt werden. Zur Annotation einer Datenquelle muss eine Verbindung zu dieser aufgebaut werden, sowie die für das Zielschema relevanten Elemente der Quelle ausgewählt werden. Dazu lassen sich im *Database Explorer* der GUI Filter definieren, um alle nicht relevanten Elemente (wie etwa temporäre Tabellen) auszublenden. Anschließend wird dann ein physisches Modell erzeugt, das alle relevanten Elemente enthält und in den weiteren Schritten Verwendung findet.

Die Elemente des neuen Modells (Tabellen, Spalten, Constraints, Trigger, etc.) sollten nun dokumentiert werden. Dazu können Beschreibungen der Elemente, verständliche Namen, und sogar visuelle Kontextmodelle angelegt werden. Ergänzend dazu kann ein Glossar erstellt werden, der die Bedeutung von Namen und Abkürzungen im Modell erfasst. So könnte z.B. für den Namen

[14] http://www.eclipse.org
[15] http://java.sun.com/products/jdbc/

„Client“ das Kürzel „CL“ samt Beschreibung in den Glossar eingetragen werden. Falls dieses Kürzel in Attributnamen auftaucht, ist es damit sowohl für die Matching-Engine als auch für den Benutzer möglich, Rückschlüsse zu ziehen.

Definieren von Matchings zwischen Datenquellen. Nachdem jede Datenquelle annotiert und dokumentiert ist, sollte definiert werden, ob und wo Korrespondenzen und Überlappungen zwischen den Quellschemas existieren. Dieser Schritt ist nicht zwingend notwendig, hilft jedoch später dabei, das Zielschema möglichst kollisionsfrei zu erzeugen, da sowohl Werkzeug als auch Benutzer ein umfassenderes Wissen über die Beziehungen zwischen den Quellen erhalten. Unglücklicherweise geht hier die Terminologie auseinander: IBM bezeichnet den Vorgang der Korrespondenzbildung als *Mapping*, wohingegen hier bisher der Begriff *Matching* für denselben Sachverhalt verwendet wurde. Um die Schritte nachvollziehbar zu halten, wird für diesen Abschnitt die IBM-Terminologie übernommen, auch wenn sie in den Augen des Autors unsauber ist.[16]

Als *Mapping* zwischen zwei Datenstrukturen versteht der RDA eine Abhängigkeit, die nicht explizit in den zugrundeliegenden Datenquellen kodiert ist. Ein *Mapping Model* ist dann die Menge der Mappings zwischen zwei unabhängigen Schemas. Ein Mapping Model ist vom Benutzer anzulegen, dabei werden Korrespondenzen zwischen den Datenstrukturen entweder manuell oder semi-automatisch, z.B. durch Heranziehen des Glossars gebildet (in RDA-Terminologie wird dieser Vorgang als *Mapping Discovery* bezeichnet). Zur Verbesserung der Qualität der Mapping-Vorschläge können auch externe Thesauri zum Auffinden von Synonymen, aber auch die Daten selber herangezogen werden. Manche dieser Mappings können Transformationen beinhalten, wie z.B. die Konkatenation von Strings (analog zu den Value Correspondences in Clio). Nachdem aus den beiden Quellmodellen das Mapping Model gebildet wurde, können falls nötig zusätzliche Annotationen vorgenommen werden.

Modellierung des Zielschemas. In diesem Schritt wird das Zielschema entworfen, in das die Quellen zu integrieren sind. Dieses kann sowohl als *logisches Modell* als auch als *physisches Modell* angelegt werden. Logische Modelle werden durch Erstellen eines Entity-Relationship-Diagramms mit einem graphischen Editor erzeugt, physische Modelle können zusätzlich zur graphischen Erstellung durch DDL-Statements oder Reverse-Engineering einer vorhandenen Datenbank erzeugt werden. Wurde nur ein logisches Modell erstellt, muss dieses nach Abschluss der Modellierungsphase zunächst in ein physisches Modell exportiert werden. Dabei kann es vorkommen, dass nicht alle im Modell enthaltenen Beziehungen in der Zieldatenbank nativ ausdrückbar sind (z.B. Vererbungsbeziehungen), so dass in diesem Fall Transformationen vorgenommen werden müssen, die diese Modellelemente in der Zieldatenbank realisieren. Das Resultat kann der Benutzer betrachten und nach Wunsch abändern.

[16] In Abschnitt 1.2 wurde der Unterschied zwischen Schema Matching und Schema Mapping erläutert

Definieren von Matchings zwischen Quellschemas und Zielschema. Dieser Schritt beinhaltet die Integration der physischen Modelle der Datenquellen mit dem im vorangehenden Schritt erzeugten physischen Modell des Zielschemas. Es muss dazu ein neues Mapping Model zwischen jeder Datenquelle und dem Zielschema angelegt werden, das die Korrespondenzen beider Modelle vollständig beschreibt. Die Vollständigkeit der Abbildung ist hier von großer Wichtigkeit, da sonst kein DDL-Code im nachfolgenden Schritt erzeugt werden kann. Der eigentliche Vorgang zur Findung von Korrespondenzen erfolgt wie beschrieben in Abschnitt 3.4. Kompliziertere Fälle wie etwa Mappings, die mehr als eine Tabelle in der Datenquelle beinhalten, können durch Definition von *Mapping Groups* behandelt werden. Eventuell müssen auch weitere Transformationen zwischen den Datentypen definiert werden.

Erzeugen der Mappings als DDL-Statements. Bevor ausführbarer Code generiert werden kann, müssen alle Mapping Models der Quellen zusammengelegt werden, um ein einziges Mapping Model zwischen den Quellen und der Zieldatenbank erzeugen zu können. Dies wird inkrementell solange durchgeführt, bis nur noch ein einziges Mapping Model existiert. Es ist wahrscheinlich, dass durch das Zusammenlegen der Mapping Models konkurrierende Mappings entstehen, bei denen sich Elemente der Quellschemas für mehr als ein Mapping auf Elemente des Zielschemas qualifizieren. Diese gilt es aufzulösen, bevor Code generiert werden kann, da Elemente der Quellschemas nur für ein einziges Mapping verwendet werden dürfen, da es offensichtlich sonst zu Mehrdeutigkeiten käme.

Bei der Generierung der DDL-Statements lassen sich je nach Typ des Zielschemas entweder SQL oder SQL/XML als Sprache wählen; bei der Art der zu formulierenden Statements kann man zwischen `SELECT`, `INSERT` (für Replikation) und `VIEW` wählen. Die generierten DDL-Statements liegen anschließend als Skripte vor, und können im Zielsystem (z.B. einem föderierten DB2-System) ausgeführt werden.

3.5 AutoMed

Nachdem bislang bei den Integrationsverfahren der vorgestellten Systeme Global-as-View favorisiert wurde, soll mit AutoMed nun ein System vorgestellt werden, das ein alternatives Verfahren einsetzt. AutoMed ist laut [20] das erste System, das das *Both-as-View*-Verfahren (BaV) zur Integration heterogener Datenquellen implementiert. Die folgenden Abschnitte geben eine Übersicht über die AutoMed-Architektur, sowie den Prozess der Schemaintegration mit BaV.

Architektur. Abb. 10 veranschaulicht die AutoMed-Architektur. Im Herzen des AutoMed-Systems liegt das *AutoMed-Repository*, welches als Plattform für alle übrigen Komponenten des Systems dient. Zu den Komponenten zählen ein Schema-Matching-Tool, ein Tool zur Erzeugung von Transformations-Templates,

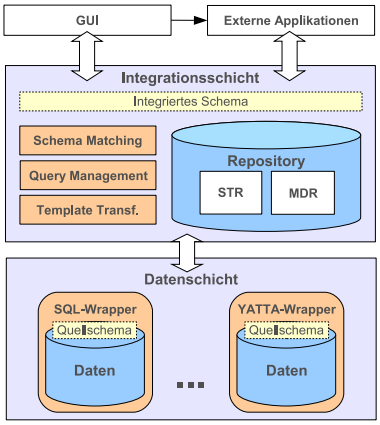

Abbildung 10. AutoMed Architektur

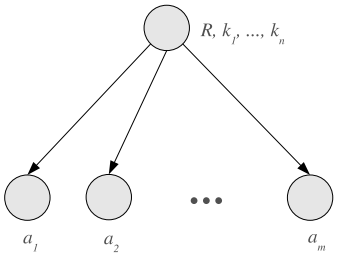

Abbildung 11. Relationales Schema im HDM

die Wrapper zur Einbindung relationaler und unstrukturierter Quellen (letztere basieren auf dem YATTA-Modell, welches hier nicht weiter behandelt wird), sowie eine GUI zur Steuerung der einzelnen Komponenten. Weiterhin können externe Applikationen in das System eingebunden und über die GUI gesteuert werden.

Dem Repository liegen zwei Kernkomponenten zugrunde: Das *Model Definition Repository (MDR)* speichert Information darüber, wie ein adaptiertes Datenmodell in das AutoMed-interne Format gemappt wird (durch Definition von sog. *Constructs*, s.u.). Es ist also in erster Linie für den verantwortlichen DBA relevant. Das *Schema Transformation Respository (STR)* speichert schließlich die adaptierten Schemas nach den im MDR definierten Regeln. Jedes Schema im STR besteht dabei aus einer Menge von *Schema Objects*, sowie *Transformationen* zwischen diesen. Für die Schema Objects fungieren die Constructs aus dem MDR als Konstruktionsvorlage. Durch diese Entkopplung wird ein hoher Grad an Flexibilität gewonnen: Sobald die Konstruktionsvorschriften für ein Datenmodell einmal im MDR vorliegen, können beliebige Quellen dieses Datenmodells miteinander durch entsprechende Schemadefinitionen und Transformationen im STR beschrieben werden.

AutoMed verwendet als internes Datenmodell das *Hypergraph Data Model (HDM)*, welches im Detail in [23] beschrieben wird. Ein Schema S wird im HDM als Tripel $\langle Nodes, Edges, Constraints\rangle$ modelliert, Knoten und Kanten bilden dabei einen benannten, gerichteten Hypergraphen[17], zusammen mit einer Menge von Nebenbedingungen $c \in Constraints$, formuliert als Boole'sche Ausdrücke.

[17] Ein Hypergraph ist in der Graphentheorie ein Graph, dessen Kanten mehr als nur zwei Knoten miteinander verbinden können

Im Falle des relationalen Modells lässt sich eine Relation R mit Schlüsselattributen $k_1, ..., k_n$ und Attributen $a_1, ..., a_m$ im HDM darstellen wie gezeigt in Abb. 11. Relationen und Attribute werden im HDM als Knoten repräsentiert, wobei Attributknoten durch Kanten mit dem zugehörigen Relationenknoten verbunden sind. Für jede Beziehung zwischen einer Relation und einem zugehörigen Attribut wird zusätzlich eine Nebenbedingung spezifiziert, die die korrekte Kardinalität sichert.

Schemaintegration mit Both-as-View. Beim BaV-Ansatz werden Mappings zwischen zwei Schemas durch eine Folge schrittweiser Transformationen beschrieben[18]. Jeder Transformationsschritt beinhaltet dabei lediglich eine einzige Operation, durch die ein Schemaelement hinzugefügt, gelöscht, oder umbenannt wird. Durch diese inkrementelle Vorgehensweise entsteht ein Pfad von Transformationen, der nach n Schritten ein Quellschema in das Zielschema überführt. Diese Transformationsfolgen sind bidirektional, können also in beide Richtungen zwischen Quell- und Zielschema laufen (ein Aspekt, der i.A. weder auf LaV noch auf BaV zutrifft).

Auch hier wird im Folgenden der (objekt)relationale Fall betrachtet. Zunächst werden Tabellen und Spalten als Tupel der Form $\langle\langle Tabelle\rangle\rangle$ respektive $\langle\langle Tabelle, Spalte, Kardinalität\rangle\rangle$ modelliert, die einer Transformationsfunktion als Eingabe dienen. Durch Angabe eines *Generators* können zusätzlich diejenigen Tupel ausgewählt werden, die sich für die Transformation qualifizieren. Als Beispiel dient basierend auf [21] folgendes Szenario: Es soll eine Zielrelation `person` mit den Attributen `id` und `name` angelegt werden, die sich vollständig in die Unterklassen `male` und `female` aufgliedert. Als Eingabe dient eine Quellrelation `staff` mit den Attributen `id`, `name` und `sex` (mit Werten `'m'` für male und `'f'` für female). Folgende Transformationsfolge überführt `staff` in die Zielrelation `person`:

1. $renameEntity(\langle\langle staff\rangle\rangle, \langle\langle person\rangle\rangle)$
2. $addEntity(\langle\langle male\rangle\rangle,\ [\{x\} \mid \{x, y\} \leftarrow \langle\langle person, sex\rangle\rangle;\ (=)\ y\ 'm'])$
3. $addEntity(\langle\langle female\rangle\rangle,\ [\{x\} \mid \{x, y\} \leftarrow \langle\langle person, sex\rangle\rangle;\ (=)\ y\ 'f'])$
4. $addGeneralisation(sex, total, person, male, female)$
5. $delAttribute(\langle\langle person, sex\rangle\rangle,$
   $([\{x, y\}|\{x\} \leftarrow \langle\langle male\rangle\rangle; (=)\ y\ 'm'] ++ [\{x, y\}|\{x\} \leftarrow \langle\langle female\rangle\rangle; (=)\ y\ 'f']),$
   $(['m','f'] = [\{y\}|\{x, y\} \leftarrow \langle\langle person, sex\rangle\rangle]))$

Transformation 1 führt eine Umbenennung in den Namen der Zielrelation durch. Die Transformationen 2 und 3 erzeugen die Relationen `male` und `female` mit denjenigen Tupeln aus `person`, deren Werte für das Attribut `sex` zu `'m'` respektive `'f'` evaluieren. Die Vererbungsbeziehung zwischen `person` und `male`

[18] Formuliert werden die Transformationen in der *Intermediate Query Language (IQL)*, die in [22] vorgestellt wird, hier jedoch keine weitere Beachtung finden soll. Anfragen auf das integrierte Schema in einer High-Level-Query-Language werden nach IQL übersetzt.

bzw. `person` und `female` realisiert Transformation 4. Die Argumente von links nach rechts spezifizieren den Namen der Beziehung, dass die Zerlegung total ist, sowie den Namen der Oberklasse und der beiden Unterklassen. Die letzte (etwas längliche) Transformation löscht schließlich das Attribut `sex`, und spezifiziert gleichzeitig, wie sich dieses wieder aus den neuen Entitäten rekonstruieren lässt, um die Bidirektionalität der Abbildung zu gewährleisten. Der Operator ++ dient zur Vereinigung von Mengen. Es wird außerdem ein Constraint angegeben, der besagt, dass die Transformation nur gültig ist, wenn das Attribut `sex` von `person` ausschließlich die Werte `'m'` oder `'f'` annimmt.

AutoMed Schema Matching. Schema Matching in AutoMed wird durch verschiedene Schema-Matching-Module realisiert. Die *Relationship-Identification-Module* versuchen untereinander kompatible Schemaobjekte zu finden (also Schemaobjekte, die für ein Matching in Frage kommen). Anschließend versuchen die *Relationship-Clarification-Module* für jedes kompatible Paar die Art der semantischen Beziehung zu ermitteln. Während dieses Vorgangs werden Ähnlichkeitsgrade zwischen Schemaobjekten gebildet, die die Wahrscheinlichkeit eines Matches angeben. Zum Auffinden von Matches werden die Instanzen der Schemaobjekte und ihre Metadaten herangezogen. Dabei werden Vergleiche z.B. von Schemaobjektnamen, Datentypen und Wertebereichen durchgeführt. Auch statistische Daten werden berücksichtigt.

3.6 MOMIS

Mit dem MOMIS-Projekt (Mediator Environment for Multiple Information Sources) arbeitet die Universität von Modena an einem ontologiebasierten Integrationssystem. Der Begriff der Ontologie ist in erster Linie im Kontext des Semantic Web zu finden, und wird in der Literatur auch als *formale, explizite Spezifikation einer gemeinsamen Konzeptualisierung* definiert (nach Gruber, vgl. [26]). Anschaulich ist eine Ontologie ein Vokabular, zusammen mit einem formalen Regelwerk, das die Elemente der Ontologie zueinander in Beziehung setzt und Aussagen über diese trifft. Anhand von Ontologien kann über entsprechend annotierte Quellen ein *Reasoning* durchgeführt werden, um neue Information abzuleiten. Die folgenden Abschnitte beschreiben die MOMIS-Architektur und den MOMIS-Integrationsprozess wie dargelegt in [24] und [25].

Architektur. MOMIS ist wie Garlic ein Mediator-basiertes System, und verwendet ebenfalls das Global-as-View-Verfahren zur Integration der Quellen. Die resultierende Sicht wird dabei in XML formuliert, und wird auch als *Global Virtual View (GVV)* bezeichnet. Architekturell ist das MOMIS-System sehr ähnlich zu Garlic, und damit auch typisch für ein föderiertes System. Interessant bei MOMIS sind in erster Linie die Komponenten, die zum Schema Matching bzw. Mapping herangezogen werden, da sie sich zum Teil grundlegend von anderen Architekturen unterscheiden.

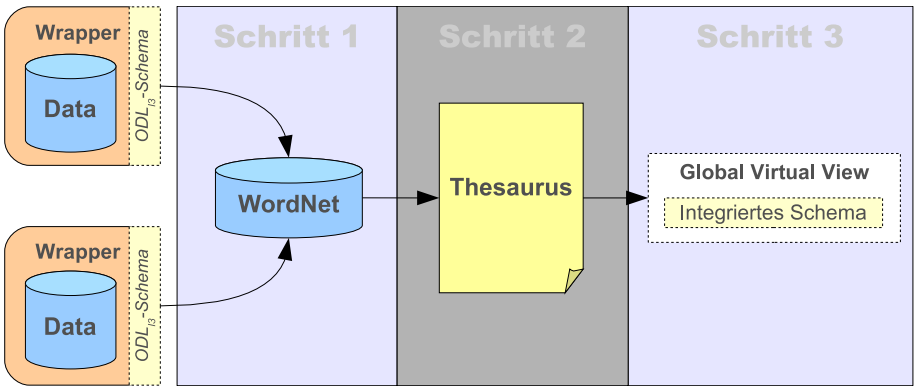

Abbildung 12. Der Integrationsprozess im MOMIS-System

MOMIS verwendet zur Beschreibung des internen Datenmodells ähnlich wie Garlic einen ODMG-ODL-Dialekt namens ODL_{I3}, der die ODL um Konzepte zum Ausdrücken von Beziehungen zwischen Schemaelementen erweitert. Folgende Beziehungen können formuliert werden, um Matches zwischen zwei Entitäten zu definieren: SYN (synonym of), BT (broader term), NT (narrower term) und RT (related term). An den Wrappern werden die Quellschemas, die entweder im relationalen Modell oder in XML vorliegen müssen, in ODL_{I3} adaptiert und exportiert.

Schemaintegration mit MOMIS. Der Integrationsprozess kann grob in drei zentrale Schritte gegliedert werden, wie illustriert in Abb. 12:

Manuelle Annotation der Quellen mit WordNet. Als Grundlage für ein Schema-Matching verwendet MOMIS *WordNet*[19] als Ontologie. WordNet ist ein elektronisches Lexikon, entwickelt an der Princeton University, das Verben, Substantive, Adjektive und Adverben zu sog. *synonym sets* zusammenschließt, falls eine semantische Verwandtschaft zwischen ihnen besteht. Eine solche Verwandtschaft kann sich in Form von Synonymen (bedeutungsgleichen Begriffen), Hyperonymen (Überbegriffen), Hyponymen (Unterbegriffen), Holonymen (Begriff ist das Ganze in einer Teil-Ganzes-Beziehung) und Meronymen (Begriff ist das Teil in einer Teil-Ganzes-Beziehung) manifestieren. Der Designer wählt in diesem Schritt für jedes Element eines Quellschemas eine, keine, oder mehrere von WordNet vorgeschlagene Begriffsdefinitionen. Eine Annotation kann dabei auch benannt werden.

Erzeugen des Thesaurus. In einem semi-automatischen Schritt wird nun ein Thesaurus generiert, bei dem Beziehungen innerhalb und zwischen Schemas durch die oben genannten SYN/BT/NT/RT-Beziehungen formuliert werden. Zunächst

[19] http://wordnet.princeton.edu/

werden dazu durch syntaktische Analyse der Quelle Beziehungen innerhalb der Schemas abgeleitet, z.B. RT-Beziehungen aus verschachtelten Knoten in einem XML-Dokument. Dieser Schritt geschieht voll-automatisch. Anschließend werden die Annotationen aus Schritt 1 herangezogen, von denen sich einige direkt abbilden lassen; so würde z.B. eine Hyperonymie auf eine BT-Beziehung abgebildet. Der Thesaurus kann schließlich durch domänenspezifisches Wissen des Designers vervollständigt bzw. korrigiert werden.

Affinitäts-Analyse, Clustering und GVV-Erzeugung. Für jedes Paar der ODL_{I3}-Klassen in den Quellschemas wird nun durch Heranziehen des Thesaurus ein *Affinitätskoeffizient* ermittelt, der festlegt, wie ausgeprägt die semantische Verwandtschaft zweier Elemente ist. Dabei wird sowohl die strukturelle (anhand der Attribute), als auch die lexikalische (anhand der Namen) Affinität berücksichtigt, um den Koeffizient zu ermitteln. Anhand der Affinität zwischen den Klassen wird dann ein *Clustering* durchgeführt, bei dem Klassen mit hoher Affinität zusammengruppiert werden. Für jedes dieser Cluster kann dann anhand einer Mapping-Tabelle eine *globale Klasse* erzeugt werden, wobei die Mapping-Tabelle bestimmt, wie die Attribute der globalen Klasse auf die Attribute der lokalen Klassen abzubilden sind. Die Spalten der Mapping-Tabelle bilden die lokalen Klassen des Clusters, die Zeilen die globalen Attribute. Für jede Zeile wird dann eine Funktion definiert, wie die Attribute der lokalen Klassen das Attribut der globalen Klasse bilden. Die Menge der globalen Klassen bildet schließlich das integrierte Schema.

4 Fazit

Neben einer Einführung in die Grundlagen der Informationsintegration im ersten Kapitel, und einer Klärung der einhergehenden Begriffe, Prozesse, Verfahren und Problemfelder, wurde im zweiten Kapitel eine typische Architektur eines Integrationssystems vorgestellt, sowie deren Kernkomponenten identifiziert und erläutert. Der Fokus lag dabei auf einer föderierten Architektur, um die Nähe zu den in Kapitel 3 vorgestellten Systemen zu unterstreichen. Dies bedeutet jedoch keinesfalls, dass föderierten Architekturen eine höhere Wichtigkeit unterstellt wird als Konsolidierungssystemen; ganz im Gegenteil sind Data Warehouses und ETL[20]-Werkzeuge zur Replizierung von Daten aus einer Quelldatenbank in eine Zieldatenbank weit verbreitet. Anfrageplanung und -ausführung in föderierten Systemen sind jedoch aufwändiger zu realisieren und bieten daher in den Augen des Autors eine interessantere Problematik.

Die Systeme in Kapitel 3 sollten einen möglichst repräsentativen Überblick im Bereich der Integrationssysteme geben, sowohl aus dem Forschungsbereich als auch aus der Industrie. Mit Garlic wurde im umfassendsten Abschnitt ein Prototyp einer Integrations-Middleware vorgestellt, dessen Komponenten sich leicht in der Referenzarchitektur wiederfinden lassen, und damit ein gutes Beispiel für den gesamten Ablauf von der Adaptierung der Daten an den Wrappern bis zur

[20] Extract - Transform - Load

verteilten Anfrageausführung liefert. Durch die darauffolgende Vorstellung des IBM DB2 Information Integrators konnte zusätzlich aufgezeigt werden, welche Teile der Garlic-Technologie sich im DB2II als Produktiv-Software wiederfinden, und wie sie aus dem Blickfeld des Datenbankadministrators verwendet werden, um heterogene Datenquellen zu einer Föderation zusammenzuschließen.

Um die in Kapitel 1 erläuterten Konflikte aufzulösen, die bei einer Integration heterogener Datenquellen auftreten können, wurde mit Clio ein Prototyp eines Werkzeugs vorgestellt, mit dem sich während einer Integration ein Schema Matching sowie Schema Mapping durchführen lässt, um den Zusammenschluss der Systeme durch eine Middleware überhaupt erst möglich zu machen. Auch hier wurde im Anschluss mit dem IBM Rational Data Architekt ein Produkt vorgestellt, welches auf der aus dem Clio-Projekt hervorgegangenen Technologie beruht. Der Blick erfolgte hier ebenfalls aus Sicht des Datenbankadministrators, um zu veranschaulichen, wie sich die Konzepte in der Praxis manifestieren.

Abschließend wurden mit AutoMed und MOMIS zwei weitere Systeme ausgewählt, die sich durch verschiedene Aspekte von vergleichbaren Systemen hervorheben. AutoMed stach im Gegensatz zu Garlic und MOMIS, die beide das Global-as-View-Verfahren einsetzen, durch die Verwendung von Both-as-View als alternativem Verfahren hervor, dessen Eigenschaft der Bidirektionalität es sowohl zu GaV als auch zu LaV kompatibel machen. MOMIS setzte sich von den anderen Systemen durch die Verwendung einer Ontologie zur Korrespondenzfindung zwischen Schemaelementen ab, und fügt sich damit nahtlos in das Konzept des Semantic Web ein.

Beim Common Data Model wurden unterschiedlichste Technologien aufgezeigt; dort stehen sich das graphbasierte HDM bei AutoMed, sowie die ODMG-ODL-basierten Ansätze von Garlic und MOMIS gegenüber. AutoMed versucht also durch ein sehr einfaches Datenmodell den kleinsten, gemeinsamen Nenner zu finden, der vielen Modellierungssprachen zugrunde liegt, wogegen Garlic und MOMIS eine Transformation in ein objektorientiertes Modell vornehmen. Auch bei der Anfragesprache versucht AutoMed mit der funktionalen IQL eine ähnlichen Weg wie beim HDM zu gehen, mit dem Argument, dass sich High-Level-Anfragesprachen wie SQL und XQuery auf einen funktionalen Kern reduzieren lassen. Insgesamt scheint jedoch immer noch SQL das Feld der Anfragesprachen zu dominieren.

Insgesamt konnte nur ein kleiner Ausschnitt der Produkte im Bereich Informationsintegration vorgestellt werden. Insbesondere die Gebiete Data Cleaning und Bestimmung von Informationsqualität stellen aktuelle Problemfelder dar. Die Vorstellung von Produkten zur Behandlung dieser Probleme hätte jedoch den Rahmen dieser Arbeit gesprengt. Einige Integrationslösungen befinden sich außerdem noch in der Entwicklung, insbesondere im Bereich Automatisierung sind also in den kommenden Jahren weitere Fortschritte zu erwarten.

Literatur

1. S. Conrad: Schemaintegration, Integrationskonflikte, Lösungsansätze, aktuelle Herausforderungen. Informatik Forsch. Entw., 17: 101-111, 2002.
2. S. Deßloch, A. Maier, N. Mattos, D. Wolfson: Information Integration - Goals and Challenges. Datenbank-Spektrum 06, 2003.
3. H. Gupta, I. Singh Mumick: Selection of Views to Materialize in a Data Warehouse. IEEE Transactions on Knowledge and Data Engineering, vol. 17, no. 1, pages 24-43, 2005.
4. T. Härder: Datenbankanwendung. Vorlesungsskriptum, Kap. 3, 16-22, TU Kaiserslautern, 2005.
5. M. Lenzerini: Data Integration: A Theoretical Perspective. In Proc. PODS'02, pages 233-246, ACM, 2002.
6. S. Fandrich: Schema Integration and Query Processing in Dynamic Grid Environments. Diploma Thesis, University of Hamburg, 2006.
7. L.M. Haas, R.J. Miller, B. Niswonger, M. Tork Roth, P.M. Schwarz, E.L. Wimmers: Transforming Heterogenous Data with Database Middleware: Beyond Integration. IEEE Data Engineering Bulletin, 22(1):31-36, 1999.
8. M. Tork Roth, P. Schwarz: Don't Scrap it, wrap it! A Wrapper Architecture for Legacy Data Sources. In Proc. of the VLDB Conference, pages 266-275, Athens, Greece, 1997.
9. M.J. Carey, L.M. Haas, P.M. Schwarz, M. Arya, W.F. Cody, R. Fagin, M. Flickner, A.W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J.H. Williams, E.L. Wimmers: Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. 5th International Workshop on Research Issues in Data Engineering-Distributed Object Management (RIDE-DOM'95), 1995.
10. W.F. Cody, L.M. Haas, W. Niblack, M. Arya, M.J. Carey, R. Fagin, M. Flickner, D. Lee, D. Petkovic, P.M. Schwarz, J. Thomas, M. Tork Roth, J.H. Williams, E.L. Wimmers: Querying Multimedia Data from Multiple Repositories by Content: The Garlic Project. Proceedings of the third IFIP WG2.6 working conference on Visual database systems 3 (VDB-3), pp. 17-35, June 1997.
11. L.M. Haas, D. Kossmann, E.L. Wimmers, J. Yang: An Optimizer for Heterogeneous Systems with NonStandard Data and Search Capabilities. IEEE Data Eng. Bull. 19(4): 37-44, 1996.
12. R. Cattell, ed.: The Object Database Standard: ODMG-93 (Release 1.1). Morgan Kaufmann, San Francisco, CA, 1994.
13. P. Bruni, F. Arnaudies, A. Bennett, S. Englert, G. Keplinger: Data Federation with IBM DB2 Information Integrator V8.1. IBM Redbook, 1st Edition, October 2003.
14. http://www-128.ibm.com/developerworks/db2/library/techarticle/dm-0506lin/
15. R.J. Miller, L.M. Haas, M.A. Hernández: Schema Mapping as Query Discovery. Proceedings of the 26th VLDB Conference, Cairo, Egypt, 2000.
16. R. Miller et. al.: The Clio Project: Managing Heterogeneity. ACM SIGMOD Record, March 2001.
17. L.M. Haas, M.A. Hernández, H. Ho, L. Popa, M. Roth: Clio grows up: From Research Prototype to Industrial Tool. In Proc. of the 2005 ACM SIGMOD international conference on Management of data, Baltimore, Maryland, June 14-16, 2005.
18. http://www-128.ibm.com/developerworks/library/ar-rdaint/
19. http://www.devx.com/IBMDB2/Article/31083

20. A. Persson, J. Stirna: AutoMed: A BAV Data Integration System for Heterogeneous Data Sources. In Proc. of the Advanced Information Systems Engineering 16th International Conference, CAiSE 2004, pp. 82-97, Riga, Latvia, June 7-11, 2004.
21. P. McBrien: A Simple Case Study for Schema Integration Tools. AutoMed Technical Report 1, Version 1, August 2003.
22. A. Poulovassilis: A Tutorial on the IQL Query Language. AutoMed Technical Report No. 28, Version 1.0, February 2004.
23. P.J. McBrien, A. Poulovassilis: A Uniform Approach to Inter-Model Transformations. In Proc. of CAiSE'99, volume 1626 of LNCS, pp. 333-348, Springer, 1999.
24. D. Beneventano, S. Bergamaschi, F. Guerra, M. Vincini: The MOMIS Approach to Information Integration. In AAAI International Conference on Enterprise Information Systems (ICEIS), 2001.
25. D. Beneventano, S. Bergamaschi: The MOMIS Methodology for Integrating Heterogeneous Data Sources. IFIP Congress Topical Sessions 2004: 19-24.
26. O. Wendt: Informationsstruktur der Unternehmung. Vorlesungsskriptum, Kap. 3, p. 110, TU Kaiserslautern, 2005.
27. I. Manolescu, D. Florescu, D. Kossmann: Answering XML Queries over Heterogeneous Data Sources. In Proc. of VLDB, 2001.