Schemamerging und -mapping

Stefan Hühner
Betreut von Dipl.-Inf. Philipp Dopichaj

Inhaltsverzeichnis

Schemamerging und -mapping 1
Stefan Hühner Betreut von Dipl.-Inf. Philipp Dopichaj
1 Einleitung 1
2 Schemaintegration 1
2.1 Ziele 2
2.2 Integrationskonflikte 3
2.3 Schemaintegrationstechniken 4
Zusicherungsbasierte Integration 4
Integrationsregeln 6
Upward Inheritance 6
Generic Integration Model - GIM 6
2.4 Multidatenbank-Anfragesprachen 6
SQL-Sichten 7
AJAX 8
SchemaSQL 10
MQL – Meta Query Language 12
FISQL – Federated Interoperable Structured Query Language . . 14
Weitere Sprachen 16
HiLog 16
SchemaLog 16
WOL 16
Vergleich 16
2.5 XML-Sichten 18
3 Anfragebearbeitung 19
3.1 AJAX-Framework 19
3.2 Schema-SQL 21
Ablauf einer SchemaSQL-Anfrage 21
Optimierung einer SchemaSQL-Anfrage 23
3.3 FISQL 24
3.4 Logische Sichten 25
4 Zusammenfassung 25

1 Einleitung

Im Rahmen der Informationsintegration ist es häufig erforderlich eine einheitliche Sicht auf verschiedene, unterschiedlich geartete Teilsysteme zu bilden, Anfragen auf dieser Sicht zuzulassen und diese auf Basis der Teilsysteme zu bearbeiten.

In Abbildung 1 ist schematisch die Struktur eines solchen integrierten Systems dargestellt. Verschiedene existierende Komponentensyteme, die jeweils lokal arbeitende Anwendungen, bedienen werden zu einem neuen integrierten System verbunden, auf dem dann neue Anwendungen den gemeinsamen Datenbestand nutzen können.

Ein Bereich innerhalb dieses Aufgabenfeldes ist die Abbildung und Integration der beteiligten Datenbankschemata. Im Kapitel 2 werden zuerst Ziele dieses Prozesses betrachtet. Darauf folgend werden die möglichen Konflikte, die bei dieser Integration auftreten können, klassifiziert und beschrieben. Danach werden im Rahmen von Schemaintegrationstechniken Verfahren für die Schema- und Datenabbildung betrachtet. Anschließend werden einige Sprachen vorgestellt, mit denen ein Zugriff auf mehrere Datenbanksysteme innerhalb einer Anfrage möglich ist. Im Kapitel 3 werden dann Verfahren zur Auswertung und Optimierung solcher Anfragen betrachtet.

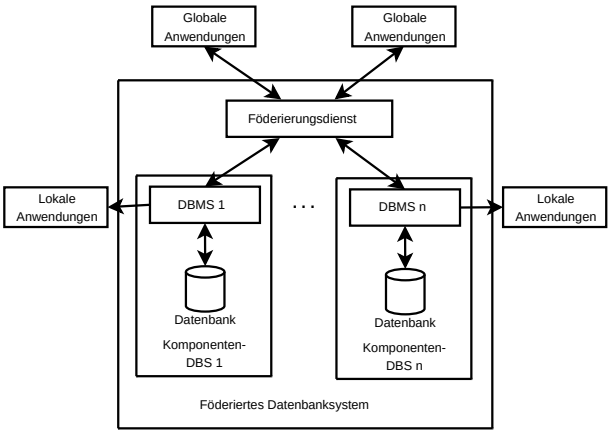

Abbildung 1. Föderiertes Datenbanksystem nach [Con01]

2 Schemaintegration

Der Prozess der Schemaintegration hat als Ziel die Schaffung eines föderierten (integrierten) Schemas, dass die möglichst alle Bestandteile der Schemata der beteiligten Komponenten-DBMS beinhaltet. In Abbildung 2 werden die am Prozess beteiligten Schema-Elemente dargestellt. Jede beteiligte Datenbank besitzt ihr lokales (konzeptionelles) Schema. Falls diese lokalen Schemata nicht

im gleichen Datenmodell vorliegen, kann über die Komponentenschemata ein Angleich durchgeführt werden. Im Rahmen dieser Arbeit wird nur die Integration von Schemata betrachtet, die im gleichen Datenmodell vorliegen. Daher fallen hier lokales Schema und Komponentenschema zusammen. Mit Hilfe des Export-Schemas kann, falls erforderlich, ein Teilbereich des lokalen Schemas zur Integration freigegeben werden. Ziel der Integration ist das schon erwähnte föderierte Schema. Für bestimmte darauf aufsetzende Anwendungen können, sofern notwendig, noch externe Schemata definiert werden, die einen Teilbereich des föderierten Schemas darstellen. Sie gleichen in Ihrer Funktion den Exportschemata der beteiligten lokalen DBMS.

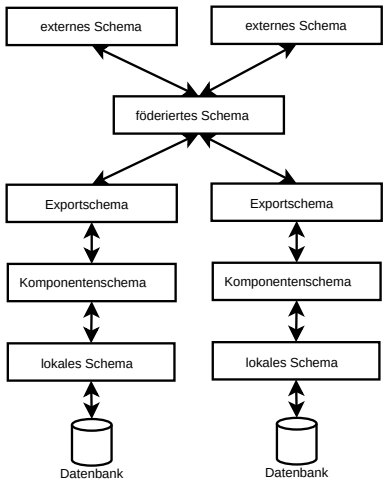

Abbildung 2. 5-Ebenen-Schema-Architektur nach [Con01]

2.1 Ziele

Im Rahmen von [BL86] werden folgende Kriterien als Ziele bei der Schemaintegration herausgearbeitet:

- Vollständigkeit und Korrektheit
  Das integrierte Schema soll möglichst eine Repräsentation aller Bereiche der in den Komponentenschemata abgebildeten Real-Welt-Domänen sein.
- Minimalität
  Falls ein Konzept[1] im den Quell-Schemata mehrfach vorkommt, so soll das Konzept im integrierten Schema nach Möglichkeit nur einmal vorkommen.

[1] Hier: Abgebildetes Real-Welt-Objekt

– Verständlichkeit
  Das integrierte Schema soll für dessen Nutzer möglichst einfach zu verstehen sein, was bei der Auswahl aus sonst gleichwertigen Möglichkeiten beachtet werden sollte.

2.2 Integrationskonflikte

Bei der Integration können verschiedene Probleme auftreten, da die beteiligten Komponentenschemata z.B. in unterschiedlichen Datenmodellen vorliegen, oder verschiedene Aspekte des abgebildeten Real-Welt-Ausschnitts auch innerhalb eines Datenmodells unterschiedlich dargestellt werden können. Nach [Con01] können diese Integrationskonflikte in verschiedene Klassen eingeordnet werden, die im nachfolgenden beschrieben werden:

– Heterogenitätskonflikte
  Sollen Schemata integriert werden, die in verschiedenen Datenmodellen vorliegen, ergibt sich das Problem, dass in diesen unterschiedliche Modellierungskonzepte verwendet werden und das Zielschema möglicherweise in einem weiteren, anderen Datenmodell vorliegen soll. Dann sind Transformationen anzuwenden, die diese heterogenen Datenmodelle in das globale Datenmodell des Zielschemas konvertieren.
  Da die Abbildung von Schemata unterschiedlicher Datenmodelle ist nicht primärer Bestandteil dieser Arbeit ist werden Ansätze zur Lösung dieser Problemklasse hier nicht weiter betrachtet.

– Strukturelle Konflikte
  Im Gegensatz zu Heterogenitätskonflikten, die zwischen unterschiedlichen Datenmodellen auftreten, kann eine ähnliche Klasse von Problemen auch innerhalb eines Datenmodells angetroffen werden. Die Struktur der modellierten Daten kann abweichen, obwohl die gleiche Abbildung von Real-Welt-Objekten betroffen ist. Im relationen Datenmodell kann ein Schema durch Vorliegen in verschiedenen Normalformen zwar den gleichen Inhalt modellieren aber die Struktur kann sich unterscheiden. In objektorientierten Modellen ist eine vielfältige Modellierung von Eigenschaften möglich: Einerseits als einfaches Attribut an einer Klasse, als eigenständige Klasse, die von der ersten referenziert wird, oder als Unterklasse pro Ausprägung dieser Eigenschaft.

– Extensionale Konflike
  Wenn während der Integration zwei Klassen von Objekten (z.b. Relationen) gefunden wurden, die die gleichen Mengen von Real-Welt-Objekten beschreiben, können die Mächtigkeiten der Mengen noch voneinander abweichen. Als Beispiel sei die Liste der Kontakte (Firmen) in einem Marketing-Informationssystem und die Liste der Rechnungsempfänger einer Buchhaltungslösung derselben Unternehmung genannt. Dabei wird sich die Menge der Rechnungsempfänger wahrscheinlich mit der Menge der Firmen des

Marketing-Systems überschneiden, diese aber nicht äquivalent sein. Eventuell stellen Rechnungsempfänger hierbei sogar eine Teilmenge der Menge der Kontakte dar.
Im Rahmen der Integration ist es notwendig diese beiden Mengen zu verschmelzen. Je nach Anforderung kann z.b. eine Obermenge der beteiligten Objekte gebildet und diese in das Zielschema übernommen werden.

- Beschreibungskonflikte
  Wenn in den Komponentenschemata Klassen von Objekten gefunden wurden, die in Beziehung zueinander stehen und aufeinander abgebildet werden sollen, können deren Eigenschaften (Attribute) trotzdem unterschiedlich modelliert worden sein. Ein Beispiel hierfür ist die Modellierung von Personennamen als Nachname, Vorname innerhalb eines Attributes oder als zwei getrennte Attribute. Eine weitere Abweichung zwischen zwei Modellierungen von gleichen Real-Welt-Objekten können unterschiedliche Namensgebungen auf Attribut-, Klassen- und Beziehungsnamen sein, die eine Umbenennung im Rahmen der Integration erforderlich machen.
  Weitere Arten von Problemen in dieser Klasse sind unterschiedliche Wertebereiche von Attributen, Skalierungskonflikte[2] oder Genauigkeitskonflikte.

- Datenkonflikte
  Ein weiterer Problembereich betrifft die eigentlichen Daten der beteiligten Komponentensysteme. Diese können auf Grund mehrerer Ursachen in anderen Ausprägungen vorliegen. Beispiele hierfür sind Eingabefehler, unterschiedliche Abkürzungen/Schreibweisen, oder die Verwendung unterschiedlicher Schlagworte beim Aufbau einer Klassifizierung wie z.B. Hiwi und Wissenschaftliche Hilfskraft.
  Konflikte in dieser Klasse lassen sich schwer automatisch lösen. In Kapitel 2.4 werden wir im Rahmen von AJAX auf diesen Bereich eingehen, u.a. wird das Finden von zusammengehörigen Datensätzen mittels Abstandsfunktionen betrachtet.

2.3 Schemaintegrationstechniken

In diesem Kapitel werden wir einige Verfahren betrachten, die Grundlagen für die Schemaintegration bilden. Damit lassen sich Korrespondenzen zwischen den beteiligeten Schemata beschreiben, sowie darauf aufbauend diese dann Integrieren.

Zusicherungsbasierte Integration Wie in [Con01] betrachtet liegen vielen Integrationstechniken Zusammenhänge zwischen Elementen verschiedener Schemata zugrunde. Mit diesen Zusicherungen oder Korrespondenzen lassen sich die relevanten semantischen Zusammenhänge zwischen Elementen beschreiben, um

[2] z.B. abweichende Maßeinheiten

auf deren Basis dann mit Hilfe von Regeln das integrierte Schema konstruiert werden kann.

Dabei sind dann spezifische Regeln für jede Art der Zusicherung notwendig, um die betroffenen Schemaelemente zu integrieren. Weitere Regeln werden zur Übernahme der Elemente benötigt, die von keiner Zusicherung erfasst werden.

Nach erfolgter Integration können diese Zusicherungen dann im integrierten Schema als Integritätsbedinungen verwendet werden, um die Konsistenz des Datenbestands zu sichern. Diese sind dann mit den aus SQL bekannten Assertions vergleichbar, wobei die aus Zusicherungen resultierenden Bedingungen mehrere Elemente verschiedener Schemata verbinden, was mit Hilfe von SQL nicht direkt darstellbar ist. Dies ist aber als Erweiterung v.a. im Rahmen von Multidatenbankanfragesprachen möglich.

- Element-Korrespondenzen
  Hierbei wird eine Beziehung zwischen genau zwei Objekten der beteiligten Schemata definiert. Dabei werden analog zu den verschiedenen Klassen bei extensionalen Konflikten folgende Arten von Korrespondenzen unterschieden:
  - $X_1 \equiv X_2$ (Äquivalenz)
    X_1 und X_2 sind semantisch äquivalent, d.h. sie repräsentieren immer dieselbe Menge von Objekten.
  - $X_1 \supseteq X_2$ (Einschluss)
    Die Objekte von X_2 sind immer auch in der Menge von X_1 enthalten.
  - $X_1 \cap X_2$ (Überlappung)
    Es gibt eine (leere oder nichtleere) Schnittmenge.
  - $X_1 \neq X_2$ (Disjunktheit)
    Die von X_1 und X_2 repräsentierten Mengen sind disjunkt.
- Element-Attribut-Korrespondenzen
  Die gleichen Arten von Beziehungen können nicht nur zwischen Objekten der beteiligten Schemata definiert werden, sondern auch zwischen deren Attributen. Dabei werden analog dazu folgende Arten von Beziehungen unterschieden:
  - $=$ Beide Attribute müssen immer denselben Wert haben.
  - $\supseteq$ Bei mengenwertigen Attributen wird hiermit eine Teilmengenbeziehung beschrieben. Andernfalls muß der Wert des einen Attributes immer dem Wert des anderen entsprechen, oder das Attribut muß den *null* Wert annehmen.
  - $\cap$ Mengenwertige Attribute deren Mengen sich überschneiden.
  - $\neq$ Die Werte der Attribute müssen immer verschieden sein.
- Pfad-Korrespondenzen
  Mit Hilfe von Pfad-Korrespondenzen können Beziehungen zwischen Objekttypen innerhalb eines Schemas beschrieben werden. Dies ist vergleichbar mit den Pfadausdrücken in objektorientierten Datenbanksystemen oder innerhalb von objektorientierten Programmiersprachen bei Zugriff auf Objekte und deren Attribute. Analog zu den zuvor beschriebenen Element- und Element-Attribut-Korresondenzen kann hier zwischen Äquivalenz, Einschluss, Überlappung und Ausschluß von zwei oder mehr Objekten unterschieden werden.

Integrationsregeln Auf Basis der im vorherigen Abschnitt beschriebenen Korrespondenzen, die für die zu integrierenden Schemata gefunden wurden, können dann im nächsten Schritt Integrationsregeln aufgestellt werden. Diese Regeln ordnen den einzelnen Korrespondenzen oder einer Gruppe von Korrespondenzen dann Aktionen zu, die festlegen wie die betroffenen Schemaelemente in das neue Schema überführt werden.

Wenn z.B. für zwei Schemaelemente X_1 und X_2 die Element-Äquivalenz als Korrespondenz gefunden wird, wird für diese beiden Schemaelemente im integrierten Schema ein einziges neues Schemaelement X eingeführt. Über die Menge der Attribute dieses neuen Schemaelements entscheiden dann die für die Attribute der Elemente X_1 und X_2 gefundenen Element-Attribut-Korrespondenzen. Eventuell weitere für X_1 und X_2 definierte Pfad-Korrespondenzen legen dann die zu definierenden Pfade von X fest.

Upward Inheritance Im Rahmen der Integration im objektorientierten Modell stellt die Methode der Upward Inheritance eine Erweiterung der Zusicherungsbasierten Integration dar. Ziel dieses Verfahrens ist die Integration von Klassenhierarchien. Analog zu den Element-Korrespondenzen werden hier vier mögliche Korrespondenzen definiert, die zwischen zwei Klassen gelten können: Äquivalenz, Einschluss, Schnitt und Disjunktheit. Je nach Art der Korrespondenz werden dann im integrierten Schema die beiden Klassen A und B wie folgt abgebildet: Wenn $A = B$ als Korrespondenz gilt, wird eine neue Klasse im integrierten Schema gebildet, die A (oder B) enthält. Wenn die Korrespondenz $A \subseteq B$ gilt, werden zunächst beide Klassen in das integrierte Schema übernommen. Die Klasse A wird im integrierten Schema von der Klasse B abgeleitet, und die gemeinsamen Attribute werden nur in der Klasse B weitergeführt. Analog lassen sich die anderen beiden Fälle der Korrespondenzen abbilden.

Generic Integration Model - GIM Beim Generic Integration Model werden auf Basis der extensionalen Beziehungen zwischen den beteiligen Objekttypen diese in sogenannte Basisextensionen zerlegt. Eine Basisextension beinhaltet die Menge von Objekten, die gleichzeitig Mitglieder derselben Objekttypen sind. Basierend auf diesen Basisextensionen können dann neue Klassen mit ihren Attributen abgeleitet werden, die ins integrierte Schema aufgenommen werden.

2.4 Multidatenbank-Anfragesprachen

Multidatenbank-Anfragesprachen stellen eine Erweiterung von SQL dar, die es zum einen ermöglicht, innerhalb einer einzelnen Anfrage mehrere Datenbanken als Quellen heranzuziehen. Andererseits soll es möglich sein, die Informationen während der Anfrage zu restrukturieren, so dass das Anfrage-Ergebnis eine deutlich andere Struktur haben kann als die, in der die Daten vorliegen. Dabei sollte die Möglichkeit bestehen, Meta-Daten gleichwertig zu den eigenlichen Daten zu betrachten, um eine hohe Flexibilität bei der Restrukturierung zu ermöglichen. Einige der bestehenden Multidatenbank-Anfragesprachen verfolgen das Ziel in

der Theorie möglichst alle Fälle abdecken zu können, so daß hier Analysen auf deren Basis möglich sind. Andere der Sprachen sind prototypisch in der Praxis umgesetzt, so daß hier empirisch Tests und Untersuchungen durchgeführt werden können.

Die Sprachen der letzteren Kategorie haben im Allgemeinen ähnliche Anforderungen als Basis:

- Abwärtskompatibilität mit SQL, damit bestehende Alt-Anwendungen weiter betrieben werden können
- Möglichst wenige Erweiterungen zu bestehenden DBMS notwendig, damit bestehende Systeme weiter (als Basis) verwendet werden können.
- Möglichkeit von dynamischen Ergebnis-Schemata bei Anfragen

Das folgende Szenario soll die Anforderung nach Restrukturierung und dynamischen Ausgabe-Schemata verdeutlichen und als laufendes Beispiel für die Darstellung der folgenden Multidatenbank-Anfragesprachen dienen. Dargestellt sind hier strukturell verschiedene Modellierungen des selben Real-Welt-Ausschnittes. Beispiel ist eine einfache Projekt-Zeiterfassung. Zu einzelnen Projekten sollen Zeiteinheiten je nach Mitarbeiter erfasst werden. Aufgabe bei der Restrukturierung ist hier die Abbildung einer Darstellung dieses Beispiels in eine andere, um die Notwendigkeiten / Möglichkeiten der verschiedenen Sprachen zu erläutern.

db2

zeit

projekt	heinz	hugo
P1	1,5	2,0
P2	heinz	null

db3

zeit_p1

mitarbeiter	arbeitsstunden
heinz	1,5
hugo	2,0

zeit_p2

mitarbeiter	arbeitsstunden
heinz	2,5

db1

zeit

projekt	mitarbeiter	arbeitsstunden
P1	heinz	1,5
P1	hugo	2,0
P2	heinz	2,5

db4_p1

zeit

mitarbeiter	arbeitsstunden
heinz	1,5
hugo	2,0

db4_p2

zeit

mitarbeiter	arbeitsstunden
heinz	2,5

Abbildung 3. Schema für die Projektzeiterfassung

SQL-Sichten Mittels SQL-Sichten ist es möglich, Sichten auf bestehende Relationen zu definieren, die die gleichen Datenbestände in unterschiedlicher Struktur darstellen. Sichten als Sprachmittel von SQL verfügen dabei fast über die

gleichen Möglichkeiten wie Anfragen in SQL. Vor allem ist es nicht möglich, innerhalb einer Anfrage Daten von mehren Datenbanken abzufragen. Desweiteren ist die Struktur der Sichten statisch und dynamische Sichten auf Basis der Dateninhalte sind nicht möglich.

Sofern aber diese Möglichkeiten zur Restrukturierung von Daten innerhalb einer einzelnen Datenbank ausreichend sind, so sind SQL-Sichten dennoch eine Möglichkeit, da sie von den bestehenden DBMS ohne Erweiterungen zur Verfügung gestellt werden.

Bezogen auf das in Abbildung 3 dargestellte Szenario ist mittels SQL-Sichten eine Transformation von/zu Struktur 4.) nicht möglich, da nur eine einzelne Datenbank angesprochen werden kann. Eine Abbildung der Strukturen 2.) und 3.) ist nur eingeschränkt möglich, da die Struktur der Ausgabeschemata und Zugriffe auf Tabellenattribute nicht dynamisch sind und daher eine transparente Transformation nicht möglich ist.

AJAX AJAX [GFSS00] ist eine deklarative, erweiterbare Sprache, die den Prozess von Datenintegration und -bereinigung als Kette von Basis-Transformationen beschreibt. Genauer wird ein solcher Prozess als gerichter azyklischer Graph von Transformationen beschreiben, bei dem Daten aus einer oder mehreren Eingangsdatenquellen schrittweise transformiert und zusammengefasst werden können. Dabei werden Lösungen für folgende Problembereiche dargestellt:

- Object Identity Problem
- Eingabefehler
- Freitext-Felder / Abkürzungen

Bei der Integration von Daten aus mehreren Quellen ist es erforderlich zu ermitteln, welche Datensätze sich auf das gleiche Real-Welt-Objekt beziehen. In der Regel findet sich dabei kein global eindeutiger Schlüssel, über den sich die Datensätze direkt abbilden lassen.

Eine Möglichkeit ist die Angabe einer Abbildung von Datensätzen, sofern sich eine solche Relation direkt finden lässt.

Falls sich z.B. Datensätze mit Personen-Informationen nur über den Namen der Person zuordnen lassen, besteht das Problem, verschiedene Schreibweisen des Namens trotzdem aufeinander abzubilden. Eine Möglichkeit hierfür ist die Verwendung von Ähnlichkeits- oder Abstandsfunktionen, die ein Maß angeben, wie ähnlich sich zwei Dateninhalte eines Attributes sind. Dann kann über die Angabe eines Schwellwertes eine Zuordnung erfolgen. Durch diese unscharfe Zuordnung können allerdings auch Datensätze fehlerhaft zusammengefasst werden, so dass eine Nachbearbeitung durch den Anwender notwendig wird.

Die Erweiterbarkeit ist konkret in drei Bereichen des Transformationsprozesses möglich: Zum einen lassen sich die Transformationen von AJAX mit regulären

SQL-Anweisungen mischen, zum anderen können domänenspezifisch Funktionsaufrufe durchgeführt werden. Dabei lassen sich die Funktionen noch klassifizieren in solche, die in der von der AJAX-Implementierung bereitgestellten Bibliothek mitgliefert werden und solche, die vom Nutzer des Systems hinzugefügt werden. Darüber lassen sich z.B. die zuvor erwähnten Ähnlichkeitsfunktionen abbilden. Die Menge der Algorithmen zum Clustern[3] lässt sich ebenfalls durch den Nutzer erweitern. Als weiteren Punkt wird bei AJAX die Integration des Nutzers in den Prozess explizit eingeschlossen. Mit den Möglichkeiten eigene, externe Funktionen und Algorithmen in diesen Prozess zu integrieren, also imperative Bestandteile mit aufzunehmen, muss die Klassifikation von AJAX als rein deklarative Sprache allerdings abgeschwächt werden.

Eine Fehlerbehandlung während der Verarbeitung wird durch einen Exception-Handling-Mechanismus ermöglicht. Beim Auftreten einer solchen Ausnahme wird die Verarbeitung allerdings nicht abgebrochen. Stattdessen werden die Datensätze markiert, bei deren Verarbeitung die Ausnahme aufgetreten ist, so dass mittels eines Reports, der nach dem Verarbeitungsvorgang erstellt wird, der Benutzer diese dargestellt bekommt und korrigierend eingreifen kann.

Folgende Basistransformationen finden im Rahmen von AJAX Anwendung:

- Mapping
  Die Mapping Operation entspricht einer Projektion im relationalen Modell, erweitert um die Möglichkeit von Funktionsaufrufen. Mit Hilfe von *LET*-Anweisungen können Funktionen komplexe Werte zurückliefern und deren Komponenten können dann wieder im Ausgabestrom verwendet werden.
- Matching
  Die Matching-Transformation erhält zwei oder mehr Eingabeströme und liefert einen Ausgabedatenstrom. Ihre Aufgabe ist die Zuordnung von zusammengehörigen Tupeln aus den Eingabedatenströmen. Durch Anwendung dieser Transformation auf den selben Eingabestrom mehrfach werden ähnliche Datensätze innerhalb dieses einen Eingabestroms zusammengefasst. Dadurch kann eine Eliminierung von Duplikaten durchgeführt werden. Optional ist es möglich einen zusätzlichen Ausgabestrom zu generieren, der diejenigen Elemente enthält, für die kein Gegenstück im Prozeß gefunden wurde. Die Ausgabe der Matching Operation ist ein Strom von n-Tupeln, wobei n die Anzahl der Eingabeströme ist. Diese Tupel enthalten die Schlüssel der zugeordneten Elemente der Eingabeströme.
- Clustering
  Die Aufgabe der Clustering-Transformation ist die Gruppierung von Datensätzen, die (potentiell) das gleiche Real-Welt-Objekt betreffen. Eingabe für die Clustering-Transformation ist dabei der Ausgabestrom der Matching-Transformation. Als Clustering-Algorithmen stehen z.B. die Bildung der transitiven Hülle zur Verfügung.

[3] Finden von zusammengehörigen Datensätzen

- Merging
  Die Merging-Transformation fasst die einzelnen Elemente, pro im Clustering-Schritt gefundener Gruppe, zu jeweils einem Element zusammenfassen.
- SQL-Sichten
  Mit der View-Transformation können SQL-Joins und SQL-Unions durchgeführt werden. Damit können z.B. Teilergebnisse früherer Transformationsschritte zusammengefasst werden.

SchemaSQL SchemaSQL [LSS96] wurde als Erweiterung der Sprache SQL konzipiert. Dabei wird diese mit der Möglichkeit ausgestattet, Anfragen gleichermaßen auf Daten und Metadaten durchzuführen und in einer Anfrage mehrere Datenbanken anzusprechen. Zweck ist die Restrukturierung der Daten während der Anfrage. Inhaltliche Ziele beim Entwurf der Sprache waren:

- Unterscheidung von verschiedenen Datenbankelementen in einer Anfrage
- Abbildung der Daten in eine erheblich andere Struktur
- Austausch von Daten und Metadaten
- Bilden von dynamischen Sichten auf Basis der Dateninhalte
- Deutliche Erweiterung der Aggregation auf Basis obiger Punkte

Desweiteren wird in [LSS96] eine Implementierung dargestellt, die unter Ausnutzung der Möglichkeiten bestehender DBMS die erweiterte Funktionalität bereitstellt. Dies wird als zusätzliches System realisiert, dass die Anfragen vorbearbeitet, dann konkrete SQL-Anfragen an die beteiligten Komponenten-DBMS stellt und deren Ergebnisse zusammensetzt um die Antwort für die eigentliche SchemaSQL Anfrage zu erzeugen.

Im Rahmen von SQL können Variablen (nur) Tupel von Relationen darstellen. Eine Variablen-Deklaration im Rahmen einer Select-Abfrage hat dabei die folgende Gestalt: <*range*> <*var*>. Dabei stellt <*range*> den Geltungsbereich der Variable <*var*> dar. Dieser Geltungsbereich wird in SchemaSQL um vier zusätzliche Typen erweitert:

- Datenbanknamen (in der Föderation)
- Relationsnamen (einer Datenbank)
- Attributnamen (einer Relation)
- Werte (eines Attributes einer Relation)

In der Syntax von SchemaSQL finden sich die insgesamt 5 Typen von Geltungsbereichen wie folgt wieder:

1. $\rightarrow$ entspricht der Menge der Datenbanknamen innerhalb der Föderation
2. $db \rightarrow$ entspricht der Menge der Relationsnamen innerhalb der Datenbank db
3. $db : rel \rightarrow$ entspricht den Attributenamen der Relation rel innerhalb der Datenbank db
4. $db : rel$ entspricht den Tupeln der Relation rel der Datenbank db

5. *db* : *rel.attr* entspricht der Menge der Werte, die in der Spalte *attr* der Relation *rel* der Datenbank *db* vorkommen.

Bei der Angabe des Geltungsbereiches in einer SchemaSQL-Anfrage ist es möglich diese Typen noch zu verschachteln. Als Beispiel dafür dient der Ausdruck *dba* $\rightarrow$ *X, dba:X T*, der eine Tupel-Variable *T* deklariert. Dabei ist der Werte-Bereich von *X* die Menge der Relationen in der Datenbank *dba* und *T* ist eine Tupel-Variable über die Tupel in jeder Relation *X*.

Als Beispiel für die konkrete Syntax von SchemaSQL soll eine Abbildung innerhalb des Szenarios von Abbildung 3 dienen. Folgende SchemaSQL-Anfrage bildet die Struktur 2.) auf 1.) ab. Dabei wird von der Möglichkeit Gebrauch gemacht, auf Metadaten zuzugreifen und diese als Daten in der Ausgabe zu verwenden. Desweiteren werden Attributnamen über Variablen referenziert, was in SQL ebenfalls nicht möglich ist. Das Ausgabeschema der Anfrage ist hier noch nicht dynamisch; dies ist für die Transformation zu Schema 1.) nicht nötig.

```
SELECT Z.projekt, S, Z.S
FROM db2:zeit Z, db2:zeit -> S
WHERE S <> 'arbeitsstunden';
```

Hier wird *Z* als Tupel-Variable über der Relation *zeit* der Datenbank *db2* deklariert und *S* als Variable über der Menge der Attributnamen von *zeit*. Die Instanzen von *S* werden einerseits direkt als Attributwerte in das Ergebnisschema eingebracht. Zusätzlich wird mit *Z.S* der Wert des über *S* referenzierten Attributes von *S* in das Ausgabeschema eingebracht. Dabei werden mit der *WHERE*-Bedingung diejenigen Attribute als Instanzen von *S* ausgeblendet, die nicht als Werte im Ergebnisschema auftauchen sollen.

Als Beispiel für dynamische Ausgabeschemata bei der Anfragebearbeitung soll die Abbildung von Struktur 1.) auf 2.) dienen. Hierbei ist es zum einen erforderlich, Daten aus db1 als Meta-Daten in der Ausgabe zu verwenden (Attributnamen). Desweiteren ist die Anzahl der Attribute des Anfrageergebnisses abhängig von der Anzahl der Zeilen.

Zur Erzeugung von dynamischen Anfrage-Schemata ändert SchemaSQL die Sichten-Definition gegenüber SQL ab. Dabei lassen sich in SchemaSQL Sichten definieren, deren Schema dynamisch abhängig von den Werten einer einzelnen Spalte der Abfrage ist.

```
CREATE VIEW
1zu2::zeit(projekt, S) as
SELECT Z1.projekt, Z1.zeit
FROM db1:zeit Z1, db1:zeit.mitarbeiter S
```

Hier wird *S* als Variable über der Menge der Werte des Attributes *mitarbeiter* der Relation *zeit* deklariert. *S* wird in der Sichten-Definition als Attribut verwendet. Da *S* eine Menge referenziert, ist dieses damit in der Schema-Definition der Sicht kein einzelnes Attribut, sondern vielmehr wird die Menge

der in *db1:zeit.mitarbeiter* referenzierten Attribut-Werte als Attributnamen im Schema der Sicht verwendet. Damit werden die Zeilen von db1 über eine Transposition zu Spalten des Ergebnis-Schemas.

Wie schon mit diesen beiden einfachen Beispielen klar wird, sind mit SchemaSQL sehr flexible Restrukturierungen möglich. Andererseits hat SchemaSQL hier noch folgende Einschränkungen / Schwächen: Es ist nur möglich eine einzelne Spalte mit Datenwerten als Meta-Daten / Attributwerten innerhalb dynamischer Sichten zu verwenden. Die Sichten-Definition von SchemaSQL definiert die Semantik der Sichten aus SQL um, was Probleme bei der Integration von Legacy-Code hervorruft. Eine weitere Einschränkung bezüglich der SchemaSQL Sichten ist die fehlende Möglichkeit, Sichten-Definitionen ineinander zu schachteln.

MQL – Meta Query Language In [WWG01] wird MQL als eine Erweiterung von SQL vorgestellt, die dieses um die Möglichkeit von dynamischen Ausgabe-Schemata von Anfragen erweitert. Ziele für diese Sprach-Erweiterung sind Abwärts-Kompatibilität mit SQL und die Möglichkeit Legacy-Anwendungen weiter zu verwenden. Desweiteren sollen die Eigenschaften von SQL, wie einfache Anfragegestaltung und die Möglichkeit der Aggregation übernommen werden. Der Sprachkern von MQL unterstützt noch nicht alle erwünschten Möglichkeiten zur Daten- / Metadaten-Integration. Daher werden in [WWG01] zusätzlich zwei Erweiterungen vorgestellt MQL+ON und MQL+JOINALL, mit denen diese Möglichkeiten gegeben sind.

An MQL und den beiden Erweiterungen wurden Komplexitätsanalysen durchgeführt, wobei Anfragen in MQL in LOGSPACE[4] möglich sind. Die Erweiterung MQL+ON erhält die Anfragebearbeitung in LOGSPACE, wohingegen bei der Erweiterung MQL+JOINALL Anfragen in PSPACE[5] möglich sind.

Als theoretische Basis für MQL ist eine äquivalente Algebra MA (Meta Algebra) als Erweiterung der Relationalen Algebra (RA) entwickelt worden. Durch diese Parallele zu SQL und RA können viele Prinzipien zur Anfragebearbeitung und -optimierung, die von SQL/RA bekannt sind, auch für MQL/MA verwendet werden.

Als erstes Beispiel zur Erläuterung der Syntax soll wieder das Beispiel auf Abbildung 3 dienen. Die folgende Anfrage soll die Struktur von 2.) in die Struktur von 1.) überführen. Dafür ist es notwendig, sowohl Metadaten abzufragen und diese im Ergebnis als Daten zu verwenden, als auch die Möglichkeit über die Metadaten Attribute zu referenzieren und deren Werte abzufragen.

```
SELECT M AS 'mitarbeiter', Z.projekt as 'projekt',
```

[4] Anfragebearbeitung ist mit logarithmischem Platzaufwand möglich (neben Platz für Input und Output)

[5] Anfragebearbeitung ist mit polynomiellem Platzaufwand möglich

```
      Z.M as 'Arbeitsstunden'
INTO 'zeit' WITHIN '2zu1'
FROM db1::zeit -> M, db1:zeit AS Z
WHERE M <> 'projekt'
```

An der Anfrage lässt sich erkennen, dass die Syntax von MQL ähnlich zu der von SchemaSQL ist. Vor allem im Bereich der Variablendeklaration mit Angabe des Gültigkeitsbereichs wird die gleiche Notation verwendet. Zur Benennung der Ausgaberelation und Datenbank werden hier die Schlüsselwörter *INTO* und *WITHIN* verwendet, mit denen explizit ein Relationsname und Datenbankname zur Aufname des Ergebnisses angegeben werden kann. Zur Benennung von Attributnamen wird das Schlüsselwort *AS* verwendet, die Notation ist hier ähnlich zu SQL gehalten. Auf der rechten Seite der *AS* Schlüsselworts können jedoch in MQL nur Konstanten verwendet werden.

Mit Hilfe der Spracherweiterung MQL+ON lässt sich diese Einschränkung umgehen. Notwendigkeit gibt sich in unserem laufendem Beispiel aus der Abbildung von Struktur 1.) in Struktur 2.), bei der es erforderlich ist, die Struktur und Attributnamen der Ausgabeparameter dynamisch zu bestimmen.

```
SELECT ((Z.projekt as 'projekt', Z.zeit ON Z.mitarbeiter)
INTO 'zeit) INTO '1zu2'
FROM db1::zeit as Z
```

Die hier verwendet Möglichkeit der Umstrukturierung ist vergleichbar mit der Transposition von Matrizen, bei der Zeilen zu Spalten werden. Die einzelnen Zeilen (pro Mitarbeiter) werden hierbei auf je ein Attribut (pro Mitarbeiter) in der Ausgaberelation abgebildet.

In [WWG01] wird MQL+JOINALL als weitere Spracherweiterung vorgestellt. Diese erweitert MQL um eine generalisierte JOIN-Operation, bei der Tupel-Variablen über outer joins von Relationen deklariert werden können.

Als Beispiel dient die Restrukturierung anhand unseres Beispiels von Struktur 3.) zu Struktur 1.) Als ersten Schritt werden mit MQL die Attributnamen der beiden Relation unter Verwendung des Relationsnamens eindeutig. Ergebnis dieser Abfrage ist eine Federation von Relationen, die im folgenden weiterverwendet wird. Die Relation wird im folgenden als A1 referenziert.

```
SELECT Z.projekt AS 'projekt', Z.zeit AS R INTO R WITHIN 'dbtmp'
FROM db3 -> R, db3::R AS Z
```

Die folgende Anfrage kapselt diese Föderation in den mit MQL+JOINALL eingeführen JOINALL Operator. Dieser wendet einen natural join[6] auf alle Relationen innerhalb von *dbtmp* an und liefert eine einzelne Relation als Ergebnis zurück.

[6] Das ist der Grund für die vorhergehende Umbenennung von Attributen, damit diese im natural join nicht verwendet werden

```
SELECT (( Z2.* ) INTO 'zeit') INTO '3zu1'
FROM JOINALL ( A1 ) AS Z2
```

Mit beiden Erweiterungen von MQL lassen sich die in den Beispielen angegeben Umstrukturierungen einfach abbilden. Je nach Struktur der Daten lassen sich bestimmte Strukturen einfacher mit MQL+ON bzw. MQL+JOINALL abbilden[7]. Die Sprache MQL+JOINALL schliesst dabei die Möglichkeiten von MQL+ON ein und übersteigt diese. Erkauft wird diese Flexibilität mit der erweiterten Komplexität bei der Formulierung und Auswertung von Anfragen (MQL $\subseteq$ PSPACE). Diese Auswertungskomplexität macht aber die praktische Anwendung im Rahmen eines DBMS problematisch.

FISQL – Federated Interoperable Structured Query Language In [WR05] wird die Federated Interoperable Relational Algebra (FIRA) als Erweiterung der Relation Algebra (RA) für die Metadaten Integration vorgestellt. Darauf aufbauend wird FISQL als SQL ähnliche mit FIRA äquivalente Anfragesprache vorgestellt. Vornehmliche Ziele bei der Entwicklung von FISQL/FIRA waren die Möglichkeit, dynamische Ergebnis-Schemata zu ermöglichen. Diese schon in SchemaSQL gegebene Möglichkeit wird von FISQL/FIRA in der Form erweitert, dass nicht nur ein Attribut zu Metadaten propagiert werden kann, sondern eine beliebige (feste) Anzahl von Attributen. Desweiteren bietet der theoretische Unterbau von FISQL mit FIRA als Erweiterung der RA die Möglichkeit bei der Anfrage-Optimierung die bereits bekannten Verfahren[8] von SQL/RA zu verwenden, bzw. diese für die neuen Operatoren von FIRA zu erweitern. Als konzeptioneller Vorgänger diente dabei SchemaSQL.

Ziele / Ergebnisse bei der Entwicklung von FISQL/FIRA sind nach [WR05]:

- Erstellung der Federate Relational Algebra als Erweiterung der RA. Dabei werden Metadaten wie Datenbank- und Relationsnamen als gleichgestellt mit Daten betrachtet.
- Komponierbarkeit – Eine FIRA-Anfrage bildet eine Menge von föderierten Datenbanken auf eine einzelne Datenbank ab. Parallel bildet RA-Anfrage eine feste Menge von Relationen auf eine einzelne Ergebnisrelation ab.
- Abwärtskompatibilität – FIRA als Erweiterung der RA enthält eine zu RA isomorphe Unteralgebra.
- Allgemeingültigkeit – Allgemeinere Unterstützung dynamischer Ergebnis-Schemata, als dies bisher verfügbare Sprachen bieten[9]
- Verschachtelte Anfragen – Unterstützung für geschachtelte Anfragen[10]

[7] z.B. die Abbildung von 3.) zu 1.) ist natürlicher mit MQL+JOINALL abbildbar

[8] wie z.B. Regel-basierte Optimierung

[9] Aufhebung der Beschränkung nur eine Spalte zu Attributnamen propagieren zu können, wie in SchemaSQL

[10] Im Gegensatz zum Sichten-Mechanismus von SchemaSQL, bei dem diese Möglichkeit nicht vorhanden ist.

Als augenscheinlichste Erweiterung von FISQL gegenüber SQL ist sicher die Möglichkeit der Deklaration von Metavariablen zu nennen. Als Wertemenge von Metavariablen können dabei Relations- und Attributnamen auftreten. Im Gegensatz zu z.B. SchemaSQL sind in FISQL explizite Schlüsselworte für die Verwendung von dynamischen Ergebnisschemata vorgesehen. Das *ON*-Schlüsselwort wird für die Vergabe von dynamischen Attributnamen verwendet und das *INTO*-Schlüsselwort dient zur Angabe der Zielrelation einer Anfrage. Durch diese Maßnahme wird der semantische Unterschied, der bei SchemaSQL in Bezug auf Sichten auftritt[11], vermieden. Desweiteren wird im Gegensatz zu SchemaSQL in FISQL klar, ob ein Bezeichner nur ein einzelnes Attribut oder eine Menge von Ergebnisattributen spezifiziert. Zusätzlich ist in FISQL die Verwendung von Unteranfragen möglich.

Als Beispiel dient wieder die Restrukurierung des Beispiels Projektzeiterfassung aus Abbildung 3. Die folgende Anfrage auf Daten in Struktur 1.) liefert diese im Format der Struktur 2.) zurück. Dabei wird vom Schlüsselwort *ON* Gebrauch gemacht, mit dem dynamische Attributnamen für die Struktur des Anfrage-Ergebnisses verwendet werden können.

```
SELECT Z.Projekt AS 'Projekt', Z.Arbeitszeit ON Z.Mitarbeiter
INTO "1zu2"
FROM db1.Zeit Z
```

Das folgende Beispiel veranschaulicht eine Anfrage auf Struktur 1.), die mit Hilfe des *INTO* Schlüsselwortes die Daten im Format von Struktur 3.) zur Verfügung stellt.

```
SELECT Z.Mitarbeiter AS 'Mitarbeiter', Z.Arbeitszeit AS 'Zeit'
INTO Z.Projekt
FROM db1.Zeit Z
```

Nachstehend ist die kanonische Form einer FISQL-Anfrage dargestellt. In dieser Form werden die Gültigkeitsbereiche von Metavariablen immer in der Form $D_1 : R_1 : A_1$ beschrieben. Die bisher verwendete Kurzschreibweise mit festen Relationsnamen kann einfach in die kanonische Form überführt werden. Dabei wird für jeden festen Relationsnamen eine Meta-Variable eingeführt. Zusätzlich wird im *WHERE* Teil der Anfrage eine Bedingung ergänzt, die den Wert der Meta-Variable auf den festen Relationsnamen einschränkt. Analog kann für feste Datenbank- und Attributnamen verfahren werden.

```
SELECT X1 AS "a1", ..., Xk AS "ak", Y1 ON Z1, ... Yp on Zp
INTO N
FROM D1 : R1 : A1 AS T1, ..., Dn : Rn : An AS Tn
WHERE C1 AND ... AND Cm
```

[11] In SchemaSQL wird das Sichten-Konstrukt als Kriterium für dynamische Ergebnis-Schemata verwendet. Daher ist die Ausführung einer Anfrage unterschiedlich je nachdem ob sie innerhalb oder ausserhalb einer Sichten-Definition auftritt.

Weitere Sprachen Im folgenden werden noch weitere Sprachen im Bereich der Datenintegration kurz vorgestellt. Auf diese wird in dieser Arbeit nur kurz verwiesen, als Basis für die Charakterisierung und deren Vergleich dient dabei der in [WR05] zu findene Vergleich. Bei den beiden Sprachen HiLog und SchemaLog ist deren Struktur als Logik-basierte Sprachen im Rahmen der Integration ein Hindernis. SQL und ähnliche deklarative Sprachen sind bei Entwicklern und Datenbankadministratoren in der Praxis bekannt. Logikbasierte Sprachen stellen hier ein ganz anderes Paradigma da.

Die deklarative Sprache WOL stellt eine reine Sprache zur Spezifikation von Datentransformationen dar. Damit fällt sie etwas aus dem Rahmen der anderen Anfragesprachen wie SchemaSQL, MQL und FISQL.

HiLog HiLog ist eine auf Prolog basierende Anfragesprache, mit der es möglich ist, Meta-Daten genauso wie Daten innerhalb einer Anfrage zu behandeln. HiLog hat aber klare Nachteile bei der Handhabung. Es ist nicht möglich dynamische Ausgabe-Schemata zu erzeugen. Desweiteren wurde HiLog im Rahmen von objekt-orientierten Datenbanken entworfen. Ein klarer Nachteil bei der Verwendung von HiLog ist die Basis von HiLog als logische Sprache im Gegensatz zu den deklarativen Sprachen wie SchemaSQL, MQL oder FiSQL.

SchemaLog SchemaLog ist eine weitere logik-basierte Sprache mit der Möglichkeit für Anfragen auf Meta-Daten. Als Vorteil von SchemaLog ist die Aufhebung der Beschränkung auf statische Ausgabe-Schemata zu sehen. Die Syntax an dieser Stelle ist ähnlich wie in SchemaSQL uneindeutig. Es ist nicht direkt erkennbar, ob eine Variable eine einzelnes Attribut oder eine Menge von Atrributen im Ausgabeschema darstellt. Der deklarativen Sprache SchemaSQL liegt eine Teilmenge von SchemaLog als Basis zugrunde.

WOL WOL [BDK98] ist eine deklarative Sprache zur Spezifikation von Datentransformationen. Der Begriff Datentransformation ist hier extra weiter gefasst als Datenbanktransformation um die Transformation der Dateninhalte explizit mit einzuschließen. Die Stärken von WOL liegen u.a. in der Möglichkeit auch rekursive Datenstrukturen abzubilden. Im Rahmen von [BDK98] wird die Beziehung zwischen *constraints* einer Datenbank und deren Verwendung im Rahmen der Integration herausgearbeitet. *Constraints* einer Datenbank können zum einen Rückschlüsse auf Zusammenhänge zwischen Daten geben, die dann im Rahmen der Integration verwendet werden können. Zum anderen werden durch den Integrationsprozess neue *constraints* geschaffen. Im Rahmen von [BDK98] ist ebenfalls ein Prototyp für eine Teilmenge von WOL entworfen worden. An diesem können dann in WOL spezifizierte Transformationen getestet werden.

Vergleich Im Kapitel wurden 2.4 haben wir verschiedene Multidatenbank-Anfragesprachen vorgestellt. Als erstes wurden hier SQL-Sichten als Mittel zur eingeschränkten Restrukturierung dargestellt. Mit diesen war es möglich Daten in einer anderen Struktur als in der vorgegebenen darzustellen. Dabei sind

aber bei SQL-Sichten einige Einschränkungen vorhanden: Eine uniforme Behandlung von Meta-Daten und Daten ist nicht möglich, die Möglichkeit von dynamischen Schemata ist nicht vorhanden. Als Vorteil ist aber zu verbuchen, daß SQL-Sichten als Sprachmittel von SQL in praktisch jedem DBMS standardmäßig vorhanden sind.

Als weiteres wurde AJAX als Sprache mit Fokus auf Datentransformation dargestellt. Die Möglichkeiten von AJAX im Bereich der Restrukturierung von Daten sind ähnlich beschränkt wie die von SQL-Sichten. Die Stärke von AJAX liegt mit den Basistranformationen des *matching*, *clustering* und *merging* klar im Bereich der Lösung von Datenkonflikten.

Die danach vorgestellten Sprachen SchemaSQL, MQL und FISQL haben Ihre Stärken bei der Restrukturierung von Daten. Mit Ihnen sind vor allem strukturelle Konflikte und Beschreibungskonflikte lösbar.

SchemaSQL ist dabei eine Erweiterung von SQL, die diese um die Möglichkeit erweitert, Meta-Daten flexibel mit Daten austauschen zu können. Dabei wird eine Transposition einer Relation möglich, eine Operation die mit den bisher dargestellen Sprachen nicht abbildbar war. Als weitere Erweiterung ermöglicht SchemaSQL die Verwendung von dynamischen Ausgabe-Schemata einer Anfrage, was vor allem im Zusammenhang mit der Möglichkeit der Transposition notwendig/sinnvoll ist. Als Kontrapunkte für SchemaSQL sind zum einen die Beschränkung zu sehen, dass nur eine einzelne Spalte einer Anfrage zu Meta-Daten gemacht werden kann, und dass für die Deklaration von dynamischen Ausgabe-Schemata die Sichten-Definition herangezogen wurde. Dadurch ändert sich die Semantik einer SchemaSQL-Anfrage je nachdem ob diese in einer Sichten-Definition eingebettet ist oder nicht. Desweiteren ist damit die Möglichkeit von verschachtelten Anfragen genommen. Im Rahmen der Implementierung eines SchemaSQL-Systems wird dieses *ontop* von bestehenden Komponenten-DBMS aufgesetzt. Dadurch wird ein radikales Modifizieren der schon bestehenden DBMS vermieden und diese Systeme können in der Föderation weiterverwendet werden. In der Praxis dürfte dieses Vorgehen den einzig praktischen Weg für Nutzer eines DBMS darstellen, da diese i.d.R. keine Änderungen am Kern des verwendeten DBMS durchführen können[12]

MQL mit den beiden Erweiterungen MQL+ON und MQL+JOINALL bietet als Erweiterung von SQL mindestens die gleichen Möglichkeiten zur Restrukturierung wie SchemaSQL, wobei der Sprachkern von MQL alleine betrachtet nur eine Teilmenge dieser Möglichkeiten bietet. Die Transpositionsoperation, die SchemaSQL bietet, ist im Sprachkern von MQL noch nicht abbildbar. Die Erweiterung MSQL+ON schließt genau diese Lücke. Die andere Erweiterung MQL+JOINALL fügt einen verallgemeinerten Join zum Sprachumfang hinzu, führt aber zu einem deutlichem Komplexitätsanstieg bei der Auswertung von An-

[12] Von der möglichen Anpassung eines *Open Source* DBMS sei hier abgesehen

fragen. MQL hat den Vorteil einer äquivalenten Algebra (MA), so dass hier, vor allem im Fokus von Optimierungen bei Anfragen, Analogien zum Paar SQL/RA gezogen werden können.

FISQL als weitere vorgestellte Sprache wurde mit SchemaSQL als konzeptionellem Vorgänger entwickelt. Die Einschränkungen von SchemaSQL nur eine Spalte für Meta-Daten verwenden zu können und die Umdefinition des Sichten-Konzepts sind in FISQL nicht vorhanden, sodass hier eine höhere Flexibilität bei der Restrukturierung vorliegt und die Syntax von Anfragen mit dynamischen Ausgabe-Schemata klarer wird. Dadurch sind geschachtelte Anfragen, wie sie schon in SQL möglich sind, auch in FISQL möglich. Analog zum Paar MQL/MA findet sich bei FISQL eine äquivalente Algebra FIRA, die eine Erweiterung der Relationalen Algebra (RA) darstellt. Mit diesem theoretischen Unterbau sind die bereits vom Paar SQL/RA bekannten regelbasierten Optimierungen auch für FISQL einsetzbar. Im Rahmen einer Implementierung wird FISQL nicht zusätzlich zu bereits vorhandenen DBMS aufgesetzt. Stattdessen wird FISQL mit seinen im Vergleich zu RA neuen Operatoren in ein DBMS integriert. Dadurch ist eine mögliche Implementierung auf die Parteien beschränkt, die Zugriff auf den Kern eines DBMS haben, oder solche die ein neues DBMS erstellen. Dies schränkt die Ersteller eines Systems das FISQL implementiert auf Hersteller von bestehenden DBMS, Forschungsgruppen die ein neues DBMS erstellen ein. Eine Erweiterung eines bestehenden Systems ist für andere Nutzer eines DBMS auf Grund des fehlenden Zugriff auf den DBMS-Kern nicht möglich[13]. Eine Alternative stellt die Erweiterung eines im Quellcode vorliegenden DBMS dar. Dabei wäre eine Implementierung prinzipiell möglich, wobei im Einzelfall zu klären ist, ob sich eine solche Änderung in die Systemstruktur integrieren lässt.

Eine weitere Gegenüberstellung von Sprachen für Metadatenintegration ist in [WR05] zu finden. Dort werden diese Sprachen unter Betrachtung Ihrer Eigenschaften wie die Verfügbarkeit einer äquivalenten Algebra, Ihrer Komplexität, der Möglichkeit verschachtelte Anfragen zu bearbeiten und weiterer Punkte gegenübergestellt.

2.5 XML-Sichten

Im Kontext der Restrukturierung von Daten während einer Anfrage auf eine relationale Datenbank haben wir SQL-Sichten betrachtet. Wenn wir als unterliegendes Datenmodell nun XML zugrundelegen besteht ebenfalls die Nachfrage, diese Daten in anderer Struktur darzustellen, als in der, in der diese vorliegen. In [Abi99] und [SR04] werden zwei Möglichkeiten betrachtet Sichten über XML-Daten auszudrücken.

In [Abi99] wird dabei die Extensible Stylesheet Language (XSL) als Mechanismus zur Restrukturierung von Daten zu Grunde gelegt. Mit Hilfe von XSL

[13] Basierend auf der Annahme, daß FISQL nicht mittels vorhandener Extension- oder Plugin-Mechanismen abgebildet werden kann

ist es möglich XML-Daten zu transformieren. Insbesondere ist es damit möglich, XML-Daten in XML-Daten anderer Struktur zu transformieren. Daher ist XSL eine Möglichkeit zur Implementierung eines Sichten-Konzepts über XML-Daten.

In [SR04] wird die XML Query Language (XQuery) als Basis für ein Sichten-Konzept über XML-Daten zugrundegelegt. Ein Sicht für XML ist dabei analog zu dem Sichten-Konzept von SQL eine gespeicherte, mit Namen versehende Anfrage.

3 Anfragebearbeitung

In diesem Kapitel soll die Bearbeitung von Anfragen im Umfeld der Schemaintegration betrachtet werden. Anhand von den in Kapitel 2.4 vorgestellten Anfragesprachen wird jeweils die Architektur und der Ablauf der Anfragebearbeitung dargestellt. Dabei wird im Rahmen der jeweiligen Sprachen auf die in dem Rahmen zusätzlich möglichen Optimierungsmöglichkeiten bei der Anfragebearbeitung eingegangen.

3.1 AJAX-Framework

Im Rahmen von [GFSS00] wird eine Systemarchitektur für AJAX vorgestellt. In Abbildung 4 ist diese vereinfacht dargestellt. Dabei sind hier nur die Teile dargestellt, die für die reine Anfragebearbeitung relevant sind. Der Ablauf einer AJAX-Anfrage durchläuft dabei drei Kern-Komponenten des Systems. Durch den *Analyzer* wird die Anfrage analysiert. Diese wird dann von der *Optimizer*-Komponente in einen möglichst effizienten Anfrage-Plan übersetzt. Die Aufgabe des *Scheduler* ist es dann, diesen optimierten Plan zur Ausführung zu bringen. Im Rahmen von [GFSS00] wird für den matching-Operator das kartesische Produkt als Basis-Operation verwendet. Daraus resultiert ein Performance-Problem, da ein kartesisches Produkt im Zusammenhang mit externen Funktionsaufrufen als nested-loop-Algorithmus abgebildet wird. Daher sind im besonderen für diesen Operator zusätzliche Optimierungen besonders wichtig.

Dabei können folgende Optimierungen angewendet werden:

1. Gemischte Auswertung
   Im Rahmen des matching-Operators können Teile des Vorgangs innerhalb des DBMS ausgeführt werden, wohingegen andere Teile komplett außerhalb des DBMS durchgeführt werden. Im Rahmen von [GFSS00] wurden Tests durchgeführt, die den Overhead von externen Funktionsaufrufen im Rahmen einer SQL-Anfrage bestimmen. Es hat sich gezeigt, dass ein signifikanter Overhead vorhanden ist. Als Folge kann z.B. die Verlagerung der Auswertung des Matching-Operators (mit externen Funktionsaufrufen) außerhalb der Datenbank eine deutliche Reduktion der Anfragekosten bewirken. Dabei werden dann Selektionen auf den Input-Datenströmen noch innerhalb der DBMS durchgeführt, dann aber die Datenströme exportiert und außerhalb der DBMS bearbeitet.

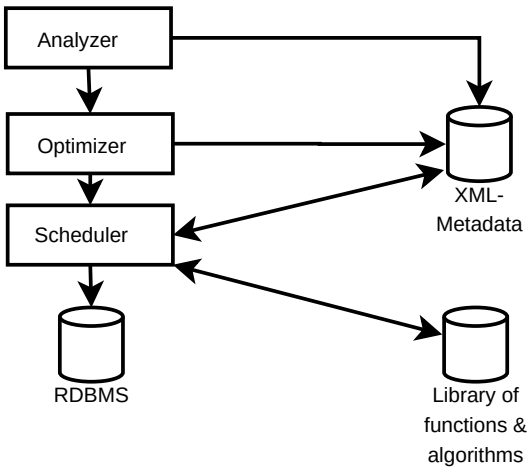

Abbildung 4. AJAX-Systemarchitektur nach[GFSS00] (vereinfacht)

2. Funktions-Umordnung
   Hierbei werden Selektionen ähnlich wie bei der klassischen SQL-Optimierung möglichst weit nach unten im Query-Plan verschoben, um die Anzahl der betroffenen Tupel zu reduzieren. Als Folge sinken die Kosten für die Ausführung eines nested-loop joins mit externen Funktionsaufrufen.
3. Neighborhood Hash Join
   Wenn bei der Anwendung des matching-Operators zusätzliches semantisches Wissen über die verwendete Ähnlichkeits- / Abstandsfunktion vorhanden ist, lässt sich der ineffiziente nested-loop-Join ersetzen. Zum Einsatz kommt stattdessen dabei ein Hash-basierter Algorithmus. Grundlage hierfür ist die Annahme, dass bei Verwendung einer Abstandsfunktion ein größter Abstand gefunden werden kann, der relevant ist. Als Folge brauchen Tupel deren Abstand größer ist nicht weiter betrachtet zu werden. Wenn nun eine einfache Funktion gefunden werden kann, die die Distanz zweier Tupel nach unten abschätzt, kann diese als Hash-Funktion für die Tupel verwendet werden, die die Tupel in Partitionen einteilt. Als Folge müssen nur noch Tupel benachbarter Partitionen betrachtet werden, wobei der Abstand der zu betrachtenden Partitionen kleiner oder gleich dem maximalen zu betrachtenden Abstand ist. Dadurch lässt sich vermeiden, jedes Tupel mit jedem anderen zu vergleichen, und die Kosten dieses Verfahren sind signifikant geringer als bei der Anwendung eines nested-loop-Joins mit externen Funktionsaufrufen.
4. Short-Circuited Computation
   Diese Optimierung hat Ihren Ursprung im Compilerbau. Dort wird bei der Auswertung von boolschen Ausdrücken die Auswertung abgebrochen, wenn der schon berechnete Teilausdruck das Ergebnis eindeutig bestimmt. Ziel im Kontext von AJAX ist die Minimierung der notwendigen externen Funktionsaufrufe. Dabei wird die Verwendung von Funktionsergebnissen in Prädi-

katen betrachtet, die das Ergebnis des Matchings bestimmen. Sobald das Ergebnis des Prädikats bekannt ist, müssen weitere eventuell vorhandene Funktionsaufrufe nicht ausgewertet werden, da sie das Ergebnis des Prädikats nicht mehr beeinflussen.

5. Cached Computation
   Ziel dieser Optimierung ist die Reduzierung der notwendigen Aufrufe externer Funktionen. Wenn über eine externe Funktion bekannt ist, dass sie ein deterministisches Ergebnis liefert[14], kann diese Optimierung angewendet werden. Dabei wird bei jedem Funktionsaufruf das Tupel von Parameter[15] und Funktionsergebnis gespeichert. Beim erneuten Funktionsaufruf kann, sofern der Parameter bereits im Cache abelegt wurde, das dort enthaltetene Funktionsergebnis zurückgeliefert werden, ohne erneut den externen Funktionsaufruf durchzuführen.
6. Parallel Computation
   Im Zusammenhang mit dem Neighborhood Hash Join ist es möglich, Teile der Anfrage parallel zu bearbeiten. Für beide beteiligten Mengen von Tupeln muss die Hash-Funktion angewendet werden. Dieser Vorgang lässt sich einfach parallelisieren wodurch eine Reduktion der Ausführungszeit möglich ist.

3.2 Schema-SQL

Ablauf einer SchemaSQL-Anfrage Im Rahmen von SchemaSQL wird die benötigte Funktionalität, wie schon in Kapitel 2.4 dargestellt, zusätzlich zu den beteiligten Komponentendatenbanken realisiert. Ziel ist dabei, das bestehende DBMS unverändert zu übernehmen und nur deren Funktionalität zu erweitern. Im Rahmen von [LSS96] wird eine Architektur zur Implementierung von SchemaSQL vorgestellt. In [LSS99] wurde prototypisch ein SchemaSQL-System implementiert.

Zusätzlich zu den beteiligten Komponeten-DBMS wird ein SchemaSQL-Server benötigt. An diesen werden die SchemaSQL-Anfragen gerichtet. Dieser richtet dann lokale Anfragen an die beteiligten Komponenten-DBMS. Zur Anfragebearbeitung werden im SchemaSQL-Server Meta-Daten benötigt, die die Schemata der lokalen DBMS beschreiben. Diese werden im SchemaSQL-Server in der *Federated System Table (FST)* vorgehalten und beinhalten die Namen der beteiligten Datenbanken, sowie die Namen aller Relationen und deren Attribute pro Datenbank[16]. Diese Informationen lassen sich i.d.R. aus den System-Tabellen der beteiligten DBMS auslesen.

[14] In diesem Kontext, dass Sie für die gleichen Eingabewerte immer das gleiche Ergebnis liefert.

[15] Der Parameter kann hier sowohl ein einzelner Wert sein oder aber wiederum ein Tupel der benötigten Parameter

[16] Zusätzlich können noch Statistiken über die Relationen der Komponenten-DBMS gespeichert sein, um diese im Rahmen der Optimierung zu verwenden

Wie in Abbildung 5 dargestellt durchläuft eine Anfrage im SchemaSQL-System mehrere Schritte, die in zwei Phasen eingeteilt werden können:

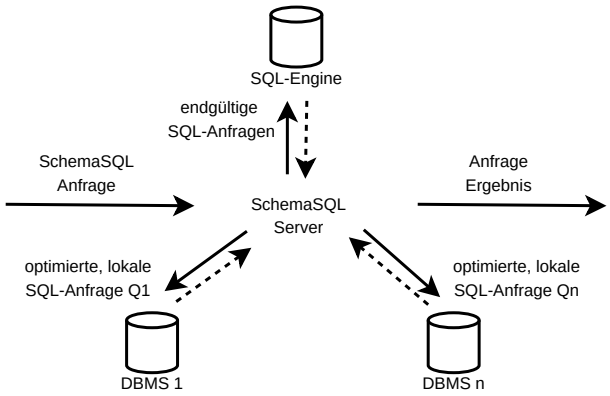

Abbildung 5. SchemaSQL - Implementierungs-Architektur nach [LSS96]

1. Aufbau von Variable Instantiation Tables (VIT)
   - Stellen von optimierten SQL-Anfragen an die beteiligen Komponenten-DBMS
   - Erzeugen der VIT aus den Ergebnissen
2. Umschreiben der SchemaSQL-Anfrage in eine äquivalente SQL-Anfrage auf den VIT
   - Bearbeitung dieser SQL-Anfrage mit der eigenen SQL-Engine

In der ersten Phase werden unter Benutzung von Anfragen auf den lokalen DBMS und/oder der FST die *Variable Instantiation Table* aufgebaut. Die VIT beinhalten dabei alle Variablen, die in der *from* Klausel der Anfrage vorkommen, sowie deren Belegungen. Mit den VIT als Basis lässt sich dann die Anfrage im zweiten Schritt auf Basis der VIT beantworten.

Als Basis dient dabei die SchemaSQL-Anfrage, die vorher in folgende Form umgeschrieben wird:

```
select S1, S2, ..., Sn
FROM <range1> V1, <range2> V2, ...,
where <cond1> and <cond2> and ...
group by groupList
having haveCondition
```

Dabei werden die *WHERE*-Bedingungen in die konjunktive Normalform umgeschrieben. Für jede Variablen-Deklaration V_i wird je nach Art der Variable

die VIT wie folgt gebildet: Wenn V_i eine Meta-Variable ist[17], dann kann die zugehörige VIT_i durch eine Abfrage auf der FST gebildet werden, da die FST alle Meta-Daten der beteiligten DBMS beinhaltet. Falls V_i eine Domain-Variable ist, wie z.B. V_i in der Deklaration $T.A\ V_i$, mit $db :: rel\ T$ oder $T.attr\ V_j$, dann werden zuerst die Deklarationen zusammengefasst, die die gleiche Tupel-Variable wie V_i betreffen. Seien nun V_i und V_j solche Variablen. Zuerst wird die Belegung T der Variablen A mit Hilfe der VIT für A bestimmt. Dann wird für jedes $a \in T$ eine SQL-Anfrage an die Datenbank *db* gestellt, um die Werte von V_i und V_j zu ermitteln. Die VIT von V_i ist dann der Verband (union) der Ergebnisse. Der allgemeinere Fall, dass *db* und *rel* ebenfalls Variablen sind, lässt sich analog behandeln, indem entsprechend deren VIT zuerst bestimmt werden. Falls V_i eine Tupel-Variable ist, werden zuerst die Belegungen der Meta-Variablen von $< range_i >$ bestimmt. Danach lassen sich die VIT für V_i bestimmen. Für Details sei hier auf [LSS96] verwiesen.

In der zweiten Phase wird dann die ursprüngliche SchemaSQL-Anfrage als SQL-Anfrage auf den in Phase 1 generierten VIT formuliert. Dazu werden die *select*, *group by* und *having* Teile der SchemaSQL-Anfrage übernommen, wobei Attribute, die Bestandteil von mehr als einer VIT sind, eindeutig gemacht werden. Der *from*-Bestandteil der SQL-Anfrage besteht aus der Menge der für das Ergebnis relevanten *VIT*. Der *where*-Teil der SQL-Anfrage wird aus der SchemaSQL-Anfrage übernommen und durch Bedingungen $VIT_i.X = VIT_j.X$ ergänzt, wenn VIT_i und VIT_j ein gemeinsames Attribut X besitzen.

Optimierung einer SchemaSQL-Anfrage Im Rahmen des beschriebenen Ablaufs können an mehreren Schritten Optimierungen vorgenommen werden:

1. Umordnung von Selektionen
   Bedingungen, die im *where*-Teil einer SchemaSQL-Anfrage vorkommen, sollten so weit wie möglich in die SQL-Anfragen geschoben werden, die an die lokalen DBMS gestellt werden.
2. Minimierung der VIT-Größe
   Durch Analyse der im *select*- und *where*-Teil verwendeten Variablen kann die Größe der VIT minimiert werden, indem nicht verwendete Attribute auf den lokalen SQL-Anfragen ausgelassen werden.
3. Zusammenfassung von VIT
   Wenn mehrere Tupel-Variablen dieselbe lokale Datenbank referenzieren und deren Bedingungen keine externen Daten von anderen Datenbanken beinhalten, können die VIT dieser Tupel-Variablen zusammengelegt werden. Dadurch wird die Gesamtgröße aller VIT reduziert und die Anzahl notwendiger Anfragen an die lokalen Datenbanken wird ebenfalls geringer, da die Anfragen für diese Tupel-Variablen ebenfalls in eine Anfrage zusammengelegt werden können.
4. Batch-Anfragen
   Ein Verbindungsaufbau zu einem lokalen DBMS ist eine Operation, die mit

[17] D.h. eine Variable über Meta-Daten

vergleichsweisen hohen Fixkosten versehen ist. Daher sollten Anfragen, die die gleiche Datenbank betreffen, zusammengelegt werden, um nur eine einzelne Datenbankverbindung pro lokaler Datenbank verwenden zu müssen.

5. Parallelisierung
   Da i.d.R. mehrere Komponentendatenbanken abgefragt werden, können die lokalen Anfragen parallel gestellt werden, um die Bearbeitungszeit weiter zu reduzieren.
6. Generierungsreihenfolge der VIT
   Die Reihenfolge der Generierung der VIT sollte so gewählt werden, dass die Belegungen der nachfolgend generierten VIT möglichst gering werden.
7. Sideway information passing (sip)
   Unter Verwendung der Heuristik, dass die Anzahl Belegungen von Meta-Variablen geringer als die Anzahl der Beleungen von anderen Variablen ist, können die vorkommenden Variablendeklarationen innerhalb der Anfrage umgeordnet werden. Daraus folgend kann die Anzahl der Belegungen von Variablen, die in Abhängigkeit von Meta-Variablen deklariert sind, reduziert werden.

3.3 FISQL

FISQL hat mit der darunterliegenden, äquivalenten Algebra FIRA klare Vorteile im Bereich der Anfragebearbeitung und Optimierung. Da FIRA eine RA-Erweiterung darstellt, sind die von SQL/RA bekannten Optimierungen zu FISQL/FIRA übertragbar. Wie in Abbildung 6 dargestellt ist der Ablauf bei der Bearbeitung einer FISQL-Anfrage gleichartig zur Bearbeitung einer SQL-Anfrage.

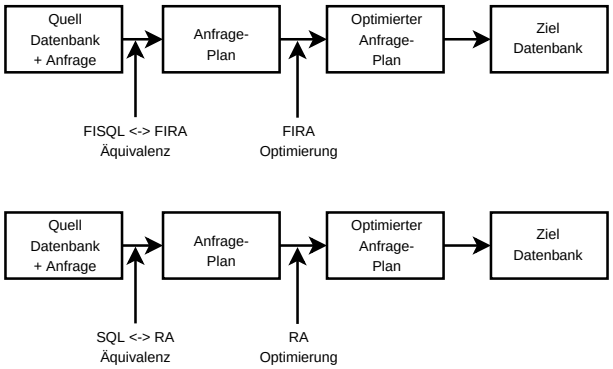

Abbildung 6. Paralleler Anfrageablauf von FISQL und SQL nach [WR05]

Durch die Einbettung einer zu RA isomorhpen Subalgebra in FIRA haben die Transformationen eines Query Plans, die von SQL bekannt sind, äquivalente Gegenstücke in FIRA. Damit können die beiden Haupt-Heuristiken, die bei

der Anfragebearbeitung in SQL wichtige Rollen spielen, auch für FISQL/FIRA verwendet werden:

- Selektionen und Projektionen werden soweit wie möglich im Syntaxbaum heruntergeschoben.
- Selektionen und kartesisches Produkt werden möglichst zum Join verbunden, da dieser i.d.R. effizienter ausgewertet werden kann.

Durch die Verwendung der von RA bekannten relationalen Operatoren in FIRA ist die Möglichkeit gegeben, die für diese Operatoren bekannten, effizienten Algorithmen und Implementierungen direkt zu nutzen. Die in FIRA neu hinzugekommenen Operatoren können auf bestehende Sortier- und Hashverfahren zurückgreifen die schon bei der Bearbeitung der relationalen Operatoren Verwendung finden.

Die Implementierung von FISQL/FIRA findet im Gegensatz zu SchemaSQL, wie auch schon in Abschnitt 2.4 beschrieben klar innerhalb eines DBMS statt, anstatt als Zusatzlösung zu bestehenden Systemen.

3.4 Logische Sichten

In [Ull97] wird ein weiterer Ansatz bei der Anfragebearbeitung vorgestellt, der im folgenden betrachtet wird. Basis ist ein vorhandenes integriertes Schema und die als Basis verwendeten Quell-Schemata. Diese Quell-Schemata werden nun als Sichten basierend auf dem integrierten Schema dargestellt. Wenn nun eine Anfrage auf dem integrierten Schema bearbeitet werden soll, ist dies durch Reformulierung dieser Anfrage auf Basis dieser Sichten möglich. Dabei dürfen in der reformulierten Anfrage nur noch die Sichten als Basis-Relationen verwendet werden. Wenn die Anfrage nach diesem Prinzip umgeschrieben worden ist, können die beteiligten Komponenten-DBMS abgefragt werden, wobei als Optimierung noch mit Hilfe einer Projektion nur die benötigten Attribute ausgewählt werden. Die im föderierten System vorhandene SQL-Engine kann dann ausgehend von diesen Basis-Sichten die umgeschriebene Anfrage auswerten.

Optimierungen sind dabei zum einen analog zu Optimierung in SQL möglich. Durch Selektion und Projekten direkt auf den Basis-Sichten können sowohl Anzahl der Tupel als auch die Anzahl der Attribute reduziert werden. Zum anderen ist im Bereich des Umschreibens der ursprünglichen Anfrage Spielraum diese in eine möglichst effiziente Form basierend auf den Sichten zu bringen.

4 Zusammenfassung

Im Rahmen dieser Arbeit wurde ein Überblick gegeben über die Aufgaben und Problemstellungen, die im Rahmen des Bereichs Schema-Merging und Mappings auftreten. Angefangen mit dem Hintergrund der Einführung eines föderierten

Datenbanksystem wurde die Notwendigkeit zur Schaffung von integrierten Datenbankschemata aufgezeigt. Eine Klassifizierung von möglichen Integrationskonflikten gab Anlass zur Betrachtung von verschiedenen Abbildungs- und Anfragesprachen mit denen sich zum einen Abbildungen zwischen Schemata beschreiben lassen bzw. mit Hilfe derer sich Anfragen im Multi-Datenbank-Umfeld durchführen lassen. Nach einem Vergleich der Vor- und Nachteile dieser Sprachen wurde der Bereich der Ausführung von Anfragen an föderierte Datenbanksystem betrachtet. Dabei wurden insbesondere die Möglichkeiten zur Optimierung, die im Rahmen dieser Sprachen möglich sind, aufgezeigt, um eine effiziente Bearbeitung von Anfragen zu ermöglichen.

Es zeigt sich das sich die technischen Aspekte des Schemamerging beherrschen lassen. Zum einen existieren Techniken zur Beschreibung der Zusammenhänge zwischen verschiedenen Schemaelementen. Auf deren Basis ist eine regelbasiertes Merging der Schemata möglich. Die Hauptarbeit bei diesem Prozess ist das Finden der Korrespondenzen. Ein Unterstützung bei diesem Vorgang ist hier in der Zukunft wünschenswert.

Andererseits ist mit den Multidatenbank-Anfragesprachen eine Möglichkeit der Anfrage-Formulierung im Rahmen eines föderierten Datenbanksystemem möglich. Zu dessen Nutzung in der Praxis ist allerdings eine Verbreitung in der Praxis notwendig. Dabei ist eine Adaption von SchemaSQL deutlich einfacher möglich, da diese ohne Eingriffe in ein bestehendes System möglich ist. FISQL bietet im Gegensatz dazu deutlich mehr Möglichkeiten. Eine Implementierung ist aber nur innerhalb eines DBMS möglich. Daher ist eine solche nur mit deutlich höherem Aufwand möglich. Daher wird hier vermutlich noch einige Zeit Vergehen, bevor eine FISQL-Implementierung in der Praxis eingesetzt werden kann. Daher ist für einen Einsatz in der Praxis im Moment eher SchemaSQL zu empfehlen, soweit nicht sogar die Möglichkeiten von SQL-Sichten für den konkreten Anwendungsfall ausreichend sind.

Literatur

[Abi99] Serge Abiteboul. On views and xml. In *PODS '99: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–9, New York, NY, USA, 1999. ACM Press.

[BDK98] Peter Buneman, Susan B. Davidson, and Anthony Kosky. Semantics of database transformations. In *Selected Papers from a Workshop on Semantics in Databases*, pages 55–91, London, UK, 1998. Springer-Verlag.

[BL86] C. Batini and M. Lenzerini. A comparative analysis of methologies for database schema integration. *ACM Computing Surveys*, 18(4), 1986.

[Con01] Stefan Conrad. Schemaintegration - Integrationskonflikte, Lösungsansätze, aktuelle Herausforderungen. *Informatik - Forschung und Entwicklung*, 2001.

[GFSS00] H. Galhardas, D. Florescu, D. Shasha, and E. Simon. Declaratively cleaning your data using ajax, 2000.

[LSS96] Laks V. S. Lakshmanan, Fereidoon Sadri, and Iyer N. Subramanian. SchemaSQL - A Language for Interoperability in Relational Multi-Database Systems. *Proceedings of the 22nd VLDB Conference Mumbai(Bombay), India, 1996*, 1996.

[LSS99] Laks V. S. Lakshmanan, Fereidoon Sadri, and Subbu N. Subramanian. On Efficiently Implementing SchemaSQL on an SQL Database System. In *VLDB*, pages 471–482, 1999.

[SR04] Deepak Srinivasa and Rajeshwari Rajendra. XViews - View Mechanism for Native XML Data. 2004.

[Ull97] Jeffrey D. Ullman. Information integration using logical views. In *ICDT '97: Proceedings of the 6th International Conference on Database Theory*, pages 19–40, London, UK, 1997. Springer-Verlag.

[WR05] Catharine M. Wyss and Edward L. Robertson. Relational languages for metadata integration. *ACM Trans. Database Syst.*, 30(2):624–660, 2005.

[WWG01] Catharine M. Wyss, Felix Wyss, and Dirk Van Gucht. Augmenting SQL with Dynamic Restructuring to Support Interoperability in a Relational Federation. In *EFIS*, pages 5–18, 2001.