Entdeckung von funktionalen Abhängigkeiten und unären Inklusionsabhängigkeiten in relationalen Datenbanken

Peter Brockhausen

betreut von
Prof. Dr. Katharina Morik
und
Dr. Joachim Hertzberg

August 1994

Diplomarbeit am Fachbereich Informatik an der
Universität Dortmund

Zusammenfassung

Die Kenntnis von unären Inklusionsabhängigkeiten und funktionalen Abhängigkeiten in relationalen Datenbanken ist zentral für die drei folgenden Problembereiche, das Datenbankdesign, die Entdeckung und Überwachung von Integritätsbedingungen und die semantische Anfrageoptiemierung. In dieser Arbeit wird zunächst ein Algorithmus vorgestellt, der unter Ausnutzung einer existierenden Axiomatisierung unäre Inklusionsabhängigkeiten entdeckt. Es wird die Korrektheit und Vollständigkeit dieses Algorithmus gezeigt, sowie eine Laufzeitabschätzung angegeben. Anschließend wird ein Algorithmus zur Bestimmung funktionaler Abhängigkeiten vorgestellt. Dabei werden neben den bereits entdeckten unären Inklusionsabhängigkeiten weitere Regeln ausgenutzt, die aus einer Axiomatisierung zur gemeinsamen Ableitung unärer Inklusionsabhängigkeiten und funktionaler Abhängigkeiten folgen. Die Arbeit endet mit einem experimentellen Vergleich zwischen den Algorithmen aus [Safnik und Flach, 1993] und [Schlimmer, 1993] und dem hier vorgestellten Algorithmus, nachdem zuvor die Korrektheit und Vollständigkeit des letzteren gezeigt wurde.

Danksagung

Mein Dank gilt Katharina Morik, daß sie es mir ermöglichte, diese Arbeit durchzuführen. Insbesondere danke ich Siegfried Bell und Joachim Hertzberg, die stets für anregende Diskussionen zur Verfügung standen.

Inhaltsverzeichnis

Kapitel 1

Motivation

Im Forschungsbereich „Datenbanken“ gibt es drei Probleme, zu deren Lösung die Ergebnisse dieser Arbeit verwendet werden können. Da ist erstens das „Datenbank–Design“ oder die Theorie von der Normalisierung von Datenbanken. Den zentralen Punkt bilden hierbei Datenabhängigkeiten. Die automatische Entdeckung dieser Abhängigkeiten ist allerdings ein bisher wenig beachteter Gegenstand der Forschung.

Eng damit zusammen hängt zweitens die Entdeckung von bestimmten Integritätsbedingungen und drittens die semantische Anfrageoptimierung. Die Verbindungen zwischen diesen drei Problemen bilden die funktionalen Abhängigkeiten und unären Inklusionsabhängigkeiten. In den nächsten Abschnitten wird dieses näher erläutert.

Danach folgt ein Abschnitt, in dem die Verbindungen dieser Arbeit zu den Forschungsgebieten „maschinelles Lernen“ und „Wissensentdeckung in Datenbanken“ vorgestellt werden. Beendet wird dieses erste Kapitel mit einer Übersicht über den weiteren Aufbau dieser Arbeit.

1.1 Datenbank–Design

Die Theorie des Datenbank–Designs bzw. der Normalformen beruht hauptsächlich auf funktionalen Abhängigkeiten. Diese werden im weiteren Verlauf kurz mit FA abgekürzt. Von diesen FAen wird verlangt, daß sie in jedem Zustand der Datenbank, d. h. bei jeder möglichen Belegung der Datenbank mit Daten, gültig sind. Die Normalisierung einer Datenbank setzt nun zwingend eine Menge von funktionalen Abhängigkeiten voraus. Dabei besteht die prinzipielle Schwierigkeit diese Abhängigkeiten zu entdecken. Bisher wird dies durch Intuition realisiert. In dieser Arbeit werden diese Abhängigkeiten durch die Anwendung wissenschaftlicher Methoden und Techniken entdeckt.

Weiterhin ist der Designer der Datenbank an einem möglichst „guten Design“ interessiert. Nun gilt jedoch, daß für ein besseres Design höhere Normalformen und damit „mehr verschiedene“ funktionale Abhängigkeiten benötigt werden. Er ist folglich mit dem Problem konfrontiert, das gewünschte

Design aufgrund ihm unbekannter funktionaler Abhängigkeiten nicht erreichen zu können. Bei hinreichend komplexen Datenbankanwendungen ist es äußerst schwierig, alle relevanten funktionalen Abhängigkeiten „durch Hinsehen“ zu entdecken.

Auch ist es möglich, daß der Designer bestimmte funktionale Abhängigkeiten ausschließt, da ihm Ausnahmen bekannt sind, bei denen sie nicht gelten. Nun kann es aber sein, daß — für ihn schwer erkennbar — diese Ausnahmen in der konkreten Datenbank nicht vorkommen und es sinnvoll ist, das Design erst dann anzupassen, wenn diese Ausnahmesituationen auftreten. Mit anderen Worten, er ist auch an temporär gültigen funktionalen Abhängigkeiten interessiert.

Der in dieser Arbeit vorzustellende Algorithmus bestimmt nun alle temporär gültigen funktionalen Abhängigkeiten. Diese umfassen natürlicherweise alle generell gültigen FAen. Der Benutzer muß dann entscheiden, welche FAen allgemeingültig sind und welche nur temporär gelten. Diese Entscheidung kann der Algorithmus ihm bei Kenntnis von nur einem Datenbankzustand nicht abnehmen. Wie darüberhinaus gezeigt wird, ist es ausreichend, nur eine Menge von minimalen funktionalen Abhängigkeiten zu bestimmen. Aus diesen ergeben sich alle anderen gültigen FAen.

Da der Algorithmus auf eine bestehende relationale Datenbank angewendet wird, ergeben sich vielfältige Möglichkeiten zu seiner Beschleunigung. So werden sämtliche von der Datenbank in Form des Datenlexikons zur Verfügung gestellte Informationen weitestgehend ausgenutzt.

1.2 Integritätsbedingungen

Zu den Integritätsbedingungen, die in dieser Arbeit betrachtet werden, gehören zum einen die funktionalen Abhängigkeiten. Weiterhin fallen hierunter Bedingungen, die an die Wertebereiche geknüpft werden, aus denen die Daten stammen. So erlauben fast alle Datenbanksysteme, obere und untere Schranken für die Daten festzulegen. Diese werden dann auch bei der Dateneingabe entsprechend überwacht. Hingegen kann es auch für den Benutzer wichtig sein, die zu einem bestimmten Zeitpunkt gültigen Schranken zu kennen. Daher werden alle bestehenden oberen und unteren Schranken bestimmt. Zusätzlich sind diese Minima und Maxima hilfreich sowohl zur Bestimmung weiterer Integritätsbedingungen, der Inklusionsabhängigkeiten, als auch zur im nächsten Abschnitt beschriebenen Anfrageoptimierung.

Eine weitere Integritätsbedingung ist die referentielle Integrität. Mittlerweile gehört es zum Standart, daß ein Datenbanksystem Möglichkeiten anbietet, diese entsprechend zu garantieren. Das unter anderem dahinterstehende Konzept von Teilmengenbeziehungen zwischen verschiedenen Datenfeldern läßt sich hingegen auf alle Datenfelder der Datenbank und insbesondere Kombinationen von diesen ausweiten. Der entsprechende Begriff dafür ist die Inklusionsabhängigkeit, im weiteren Verlauf mit IA abgekürzt. Diese

Arbeit beschränkt sich jedoch auf die einstelligen oder unären Inklusionsabhängigkeiten. Diese umfassen aber die erwähnte referentielle Integrität.

Wie schon bei den funktionalen Abhängigkeiten so existiert auch hier das Problem, daß zum einen die entsprechenden IAen nicht bekannt sind und deshalb vom Datenbanksystem über die referentielle Integrität nicht überwacht werden können. Oder der Benutzer ist wiederum nur an temporär gültigen IAen interessiert. Der entsprechende Algorithmus zur Bestimmung der IAen berechnet wieder alle temporär gültigen IAen. Dabei werden auch hier die Informationen ausgenutzt, die das Datenbanksystem zur Verfügung stellt.

1.3 Semantische Anfrageoptimierung

Unter der Optimierung von Anfragen an die Datenbank werden zwei verschiedene Dinge verstanden. Auf der einen Seite müssen Anfragen an die Datenbank übersetzt werden in die interne Repräsentation des Datenbanksystems. Dabei ergeben sich vielfältige Möglichkeiten zur Optimierung aufgrund der „Syntax der konkreten Anfrage“. Dieses ist weitestgehend bekannt und hier nicht gemeint.

Es geht hier vielmehr um die Optimierung von Anfragen aufgrund der „Semantik der konkreten Anfrage“. Wenn zum Beispiel bekannt ist, daß der größte vorkommende Wert für ein Datenfeld „Gehalt“ 7500 DM ist, dann braucht eine Anfrage an die Datenbank, die alle Mitarbeiter herausfinden soll, die mehr als 8000 DM verdienen, gar nicht mehr ausgeführt zu werden. Für diese und andere semantische Optimierungen werden sowohl FAen, IAen als auch numerische Integritätsbedingungen benötigt.

1.4 Maschinelles Lernen und Wissensentdeckung in Datenbanken

In den bisherigen Abschnitten wurden drei Problembereiche aus dem Gebiet der Datenbanken dargestellt, die diese Arbeit ganz wesentlich motivierten. Nun werden zwei Forschungsbereiche kurz vorgestellt, maschinelles Lernen und Wissensentdeckung in Datenbanken, in dessen Rahmen diese Arbeit auch angesiedelt werden kann. Sei es, daß Methoden verwandt werden, die aus ersterem Gebiet stammen, oder daß die Aufgabenstellung an sich in diesem Rahmen plaziert werden kann, wie im zweiten Fall.

1.4.1 Maschinelles Lernen

Im Forschungsgebiet des maschinellen Lernens beschäftigt man sich mit dem automatisierten Erwerb von Begriffen. Dieser Begriffserwerb wird als Lernen bezeichnet. Eine allgemeine Einführung in das Gebiet des maschinellen

Lernens gibt Morik in [Morik, 1993]. Auf eine Diskussion darüber, ob ein Erwerb von Begriffen als Lernen bezeichnet werden kann, sowie über die Unterschiede zwischen maschinellem Lernen und menschlichem Lernen, wird hier verzichtet. Dazu sei wieder auf [Morik, 1993] verwiesen.

Im folgenden soll kurz dargelegt werden, daß ein kleiner Ausschnitt aus dem Bereich des maschinellen Lernens, bei einer abstrakten Sichtweise, große Ähnlichkeiten mit der Vorgehensweise in dem Algorithmus zur Entdeckung der funktionalen Abhängigkeiten hat. Mitchell beschreibt in [Mitchell, 1982] „Lernen aus Beispielen“ als Suche. Dazu verwendet er zwei Operationen, die Generalisierung und die Spezialisierung, die in ähnlicher Form in dem genannten Algorithmus verwendet werden. Dieses wird im Kapitel 4.3.2 näher erläutert.

Die Beispiele werden im Rahmen dieser Arbeit durch die Datenbank realisiert. Der geforderte Erwerb von Begriffen — durch eine Suche — entspricht der Entdeckung von funktionalen Abhängigkeiten, die ebenfalls über eine Suche realisiert wird. Somit kann letzteres als „Lernen“ bezeichnet werden.

1.4.2 Wissensentdeckung in Datenbanken

Dem Gebiet der „Wissensentdeckung in Datenbanken“, engl. „knowledge discovery in databases“, ist in letzter Zeit vermehrt Aufmerksamkeit zuteil geworden. Dieses Forschungsgebiet hat einige Berührungspunkte zum maschinellen Lernen. Holsheimer und Siebes bezeichnen die Wissensentdeckung in Datenbanken als ein Teilgebiet des maschinellen Lernens, in dem der betrachtete Ausschnitt der Welt über eine Datenbank wahrgenommen wird, [Holsheimer und Siebes, 1994]. Die „Entdeckung von Wissen in Datenbanken“ wird als Prozeß des „nicht trivialen Extrahierens von implizit vorhandenen, vorher unbekannten und möglicherweise nützlichen Informationen aus Daten“ aufgefaßt, [Piatetsky-Shapiro et al., 1991].

Da die Entdeckung von Wissen als äquivalent zum Lernen betrachtet werden kann, ist es sicherlich gerechtfertigt — zumindest in bezug auf die Motivationen, die hinter beiden Gebieten stehen — zu sagen, daß die Wissensentdeckung in Datenbanken ein Teilgebiet des maschinellen Lernens sei. Inwieweit diese Aussage bei der Betrachtung konkreter Verfahren bestehen bleiben kann, hängt wesentlich davon ab, wie die persönliche Definition für Lernen aussieht. Somit ist es gerechtfertigt zu sagen — abgesehen von der dem Namen nach existierenden Ähnlichkeit zwischen dem Titel dieser Arbeit und der Wissensentdeckung in Datenbanken —, daß diese Arbeit sehr wohl in das Gebiet der Wissensentdeckung in Datenbanken einzuordnen ist.

Auf die spezifischen Unterschiede zwischen maschinellem Lernen und der Wissensentdeckung in Datenbanken, z. B. die Art und Anzahl der Beispiele betreffend, wird hier nicht weiter eingegangen. Einen Überblick hierzu geben z. B. [Piatetsky-Shapiro et al., 1991] als auch [Holsheimer und Siebes, 1994]. Letztere beschreiben dort auch einige erfolgreiche Adaptierungen maschineller Lernverfahren auf Datenbanken.

1.5 Überblick

Zuerst werden im nächsten Kapitel die theoretischen Grundlagen beschrieben, soweit sie für das Verständnis der vorgestellten Algorithmen benötigt werden. Anschließend folgt im dritten Kapitel eine Beschreibung der Gesamtarchitektur des Programmpaketes. Danach folgt im vierten Kapitel die Beschreibung der einzelnen Algorithmen. Diese werden unter Angabe ihrer jeweiligen Laufzeiten ausführlich erläutert. Weiterhin werden Angaben zur Korrektheit und Vollständigkeit gemacht. Im fünften Kapitel werden zwei verwandte Arbeiten kurz vorgestellt und die Ergebnisse dieser Arbeit mit jenen verglichen. Den Abschluß bildet das sechste Kapitel mit einer kurzen Zusammenfassung und Diskussion dieser Arbeit. Dort erfolgt auch ein Ausblick auf das, was im Hinblick auf eine verbesserte Version noch zu sagen oder zu erledigen übrig bleibt, jedoch noch nicht realisiert wurde.

Kapitel 2

Relationale Datenbanken

2.1 Aufbau einer relationalen Datenbank

In diesem Abschnitt wird eine Einführung in das relationale Datenmodell gegeben. Dabei wird bewußt auf eine Darstellung mit Hilfe des zugrunde liegenden mathematischen Modells der Relationen–Algebra verzichtet. Hierzu sei auf die Literatur, z. B. [Ullman, 1988, Kap. 3] verwiesen. Eine gängige Methode, dieses relationale Modell darzustellen, ist der Ansatz über Tabellen, der hier verwendet wird. In Abbildung 2.1 wird schematisch der Aufbau einer relationalen Datenbank gezeigt, die im weiteren Verlauf für entsprechende Beispiele benutzt wird.

Eine relationale Datenbank besteht aus Tabellen. Jede Tabelle und in jeder Tabelle jede Spalte haben ihren eigenen, eindeutigen Namen, z. B. AUFTRAG oder LName. Die Namen der Spalten werden als Attribute bezeichnet. Während Tabellennamen eindeutig bzgl. der gesamten Datenbank sein müssen, dürfen in verschiedenen Tabellen die gleichen Attributnamen verwendet werden. In den Zeilen der Tabelle stehen korrespondierend zu den Attributen die jeweiligen Daten oder Werte. Einzelne Zeilen werden als Tupel bezeichnet. Über den Tabellennamen und den Attributnamen wird eine Menge von Werten eindeutig bestimmt. Die Reihenfolge der Attribute generell und — eingeschränkt — die der Tupel ist beliebig, da beide jeweils als Mengen aufgefaßt werden. Jedoch können für Tupel Sortierungen erzwungen werden.

Im Gegensatz zur Auswahl eines Attributes über eindeutige Namen werden die Tupel, die die Ergebnisse von Anfragen bilden, über logische Ausdrücke bestimmt, die die Werte dieser Tupel erfüllen müssen. Es ist nicht vorgesehen direkt z. B. das vierte Tupel auszugeben. Die Ergebnisse von Anfragen an die Datenbank sind daher Mengen von Tupeln ohne feststehende Reihenfolge. Weiterhin werden die Tabellen als Relationen bezeichnet und die Menge der Attribute einer Relation als Relationenschema. Alle Relationenschemata zusammen bilden das Datenbankschema. Die Menge aller Attribute des Datenbankschemas wird als Universum U bezeichnet. Einen

KUNDE	KName	KAdr	Kto
	Meier	Zeisweg	1000
	Müller	Parkstraße	15000
	Schulze	Schloßallee	-3562
	⋮		⋮
	Weber	Schillergasse	-13

AUFTRAG	KName	Ware	Kto
	Meier	Brot	1
	Meier	Wurst	5
	Meier	Milch	1
	Müller	Saft	3
	Müller	Seife	1
	Schulze	Milch	5
	⋮		⋮
	Weber	Brot	1

LIEFERANT	LName	LAdr	Ware	Preis
	Meier	Zeisweg	Wurst	3.95
	Meier	Zeisweg	Milch	1.75
	Meier	Zeisweg	Käse	4.59
	Müller	Parkstraße	Seife	0.76
	Müller	Parkstraße	Brot	2.69
	Müller	Parkstraße	Saft	1.99
	Müller	Parkstraße	Milch	1.75
	⋮			⋮
	Müller	Parkstraße	Fisch	0.99

Abbildung 2.1: Eine mögliche Ausprägung der Datenbank „GENUG" der Genußmittelgesellschaft

umfassenden Einstieg in das relationale Datenmodell bietet zum Beispiel [Ullman, 1988, Kap. 2].

Ein relationales Datenbanksystem besteht aus der Datenbank und einem Datenbankmanagementsystem, kurz als DB und DBMS bezeichnet. Die Datenbank ist einheitlich gemäß dem oben skizzierten Modell strukturiert. Das DBMS erfüllt unter anderem die beiden folgenden Funktionen. Es bietet zum einen dem Benutzer eine komfortable Anfragesprache, um Daten aus der Datenbank zu extrahieren; dies ist in der Regel SQL, auf welche später noch eingegangen wird. Die andere Funktion besteht in der Verwaltung der Relationenschemata. Diese werden in einem Katalog abgelegt, der als Datenlexikon oder engl. data dictionary bezeichnet wird. Zu den weiteren Funktionen des DBMS und ihrer Realisierung sei auf [Lockemann und Schmidt, 1987] verwiesen.

Dieses Datenlexikon ist nun aus Benutzersicht genau wie jede Datentabelle strukturiert. Daher wird es auch synonym als Systemtabelle bezeichnet. In der Abbildung 2.2 ist der Aufbau einer Systemtabelle mit den für die folgenden Algorithmen wichtigen Informationen skizziert.

Die Bedeutung der Einträge unter den Attributen „Rel.Name" und „Attribut" ist offensichtlich. Für jedes Attribut einer Relation muß weiterhin ein Typ angegeben werden, welcher bei der Dateneingabe vom System überwacht wird. Für die hier gestellte Aufgabe sind nur zwei Grundtypen von

Rel. Name	Attribut	Typ	Schlüsseltyp	NULL-Werte	Index
KUNDE	KName	VARCHAR	Primärschlüssel	not NULL	Unique
AUFTRAG	KName	VARCHAR	Primärschlüssel	not NULL	Index
⋮					
LIEFERANT	LName	VARCHAR	Primärschlüssel	not NULL	Index
LIEFERANT	LName	VARCHAR	Fremdschlüssel	not NULL	Index
LIEFERANT	Ware	VARCHAR	—	NULL	—
LIEFERANT	Preis	NUMBER	—	not NULL	Index

Abbildung 2.2: Schematische Darstellung der Systemtabelle eines DBMS

Belang. Einmal ein numerischer Typ: da werden alle numerischen Typen, die das DBMS möglicherweise anbietet, z. B. INTEGER, FLOAT, in den Algorithmen zum Typ NUMBER zusammengefaßt. Dann bietet das DBMS für alphanumerische Zeichenketten in der Regel auch verschiedene Typen an, je nachdem ob die Zeichenketten feste oder variable Längen haben. Auch diese Differenzierungen werden in den Algorithmen ignoriert. Der entsprechende zweite Typ zu NUMBER ist ein beliebig langer „String“, der in Anlehnung an Oracle V7[1] mit VARCHAR bezeichnet wird.

Eine wichtige Rolle spielen die Einträge für das Attribut „Schlüsseltyp“. Handelt es sich um einen Primärschlüssel, so wird vom DBMS garantiert, daß jeder Wert für das entsprechende Attribut genau einmal vorkommt und daß es kein NULL–Wert ist, d. h. es gibt für das Attribut auch einen Wert. NULL–Werte oder der Eintrag NULL in einem Tupel für ein Attribut bedeutet, daß das Feld leer ist, also keinen Eintrag enthält. Dieses dient insbesondere zur Unterscheidung von z. B. einer Zeichenkette aus Leerzeichen oder der Zahl Null. Solche Felder enthalten sehr wohl Einträge. Kommen weiterhin mehrere NULL–Werte vor, so werden sämtliche NULL–Werte der Datenbank grundsätzlich als voneinander verschieden betrachtet.

Schlüssel können aus einem Attribut, wie z. B. KName, oder aus mehreren bestehen, wie z. B. LName Ware. In diesem Fall gilt, daß jede Kombination der Werte für den zusammengesetzten Schlüssel eindeutig ist und keine Null–Werte enthält.

Zusätzlich können Attribute einer Relation als Fremdschlüssel bezogen auf Attribute aus der selben oder anderen Relationen gekennzeichnet werden. Wird z. B. in dem Beispiel das Attribut LName als Fremdschlüssel bzgl. des Attributes KName aus der Relation KUNDE gekennzeichnet, so wird garantiert, daß jeder Wert in LName auch in KName vorkommt. Dies bedeutet, daß jeder Lieferant auch selbst ein Kunde ist. Hierbei ist aber zu beachten, daß KName ein Primärschlüssel sein muß. Auch Fremdschlüssel können mehrstellig sein.

Weiterhin kann zusätzlich für jedes Attribut — unabhängig davon, ob es zu einem Schlüssel gehört — eingetragen werden, ob NULL–Werte erlaubt sind

[1] Oracle V7 ist die Kurzform für das verwendete relationale Datenbankmanagementsystem „Oracle7 Server“ der Firma Oracle in der Version 7.

oder nicht. Auch dieses wird vom DBMS in jedem Zustand der Datenbank garantiert.

Als letztes sind noch die Indizes von Interesse. Hier wird zwischen normalen und eindeutigen Indizes unterschieden. Bei einem Index handelt es sich um eine Sortierung der Tupel einer Tabelle nach einem oder mehreren Attributen in auf– oder absteigender Reihenfolge. Ist der Index eindeutig, so wird wiederum garantiert, daß jeder Wert oder jede Wertekombination nur einmal vorkommt.

Im Unterschied zu Schlüsseln sind hier jedoch auch bei zusammengesetzten, eindeutigen Indizes Null–Werte erlaubt. Es wird nur verhindert, daß in allen beteiligten Attributen eines Tupels gleichzeitig Null–Werte stehen. In den Algorithmen, die im weiteren Verlauf vorgestellt werden, wird intensiv von der Systemtabelle Gebrauch gemacht, um Berechnungen abzukürzen bzw. überflüssige einzusparen.

Betrachtet man noch einmal die Beispieldatenbank, so fällt schnell das folgende Problem auf: Die Redundanz in der Datenhaltung. Es wird für jeden Lieferantennamen die Adresse wiederholt. Nimmt man an, daß Lieferantennamen eindeutig sind, so würde ein einmaliger Eintrag genügen. Ein weiteres Problem sind Änderungsanomalien, engl. update anomaly. Ändert sich die Adresse des Lieferanten Meier, dann muß sie in jedem Tupel geändert werden. Hierbei kann bei vielen Daten leicht ein Tupel übersehen werden, was als Konsequenz eine Inkonsistenz bedeuten würde. Diese Änderungen lassen sich auch nicht so automatisieren, daß dieses Problem in jeder denkbaren Situation vermieden wird.

Darüberhinaus lassen sich noch zwei weitere Anomalien beobachten. Zum einen die Einfügeanomalie, engl. insertion anomaly. Nimmt man an, daß LName und Ware den zusammengesetzten Schlüssel bilden für die Relation LIEFERANT, so ist es unmöglich, die Adresse eines Lieferanten einzutragen, wenn dieser im Moment keine Ware liefert, denn NULL–Werte für einen Teilschlüssel, d.h. für ein Attribut eines zusammengesetzten Schlüssels, sind modellgemäß verboten. Eng verbunden hiermit ist das inverse Problem der Löschanomalie, engl. deletion anomaly. Liefert ein Lieferant keine Ware mehr, muß auch seine Adresse gelöscht werden. Diese Probleme würden sich jedoch nicht stellen, wenn der Designer der Datenbank von vornherein das Schema LIEFERANT durch die zwei Schemata LIEFERANT(LName, LAdr) und ANGEBOT(LName, Ware, Preis) ersetzt hätte. Die Theorie, die einem hilft, diese Alternative direkt zu finden, wird im folgenden Abschnitt vorgestellt.

2.2 Funktionale Abhängigkeiten

In diesem Abschnitt werden nun die funktionalen Abhängigkeiten formal definiert. Ihre Einführung in die Theorie der Datenbanken geht auf Codd zurück, [Codd, 1970]. In den Definitionen und Sätzen bezeichnen Großbuchstaben A, B, C, ... vom Beginn des Alphabetes einzelne Attribute. Großbuchstaben vom Ende des Alphabetes hingegen stehen für Mengen von Attributen. Wenn nicht extra darauf hingewiesen wird, können diese einelementig sein. Die Buchstaben R und F bezeichnen Relationen bzw. Mengen von funktionalen Abhängigkeiten.

Definition 2.1 *Sei U eine Attributmenge. Eine* **funktionale Abhängigkeit, (FA)**, *ist ein Paar von Teilmengen $X, Y \subseteq U$; Notation: $X \to Y$. Sprechweise: X bestimmt (funktional) Y, oder Y wird von X (funktional) bestimmt.*

In den folgenden Beispielen wird, wenn nicht anders angegeben, immer auf das Beispiel in Abbildung 2.1 bezug genommen.

Beispiel 2.1 *In KUNDE gilt z. B. $KName \to KAdr\,Kto$ oder in LIEFERANT gilt $LName\,Ware \to Preis$* ◇

Definition 2.2 *Ein* **relationales Datenbankschema** $D = \langle (R_1(U_1), F_1), \ldots, (R_n(U_n), F_n) \rangle$ *besteht aus den Relationenschemata $(R_i(U_i), F_i)$ mit $1 \leq i \leq n$, wobei R_i Relationennamen, U_i Attributmengen und F_i endliche Mengen funktionaler Abhängigkeiten über U_i bezeichnet. D heißt einfach für $n = 1$.*

Beispiel 2.2

$$\begin{aligned} GENUG = \langle & KUNDE(KName, KAdr, Kto), \{KName \to KAdr\,Kto\}; \\ & AUFTRAG(KName, Ware, Kto), \{KName\,Ware \to Kto\}; \\ & LIEFERANT(LName, LAdr, Ware, Preis), \\ & \{LName\,Ware \to Preis, LName \to LAdr\} \rangle \end{aligned}$$

◇

Notation

1. Relationenschemata werden auch nur mit R oder $R(U)$ anstelle von $(R(U), F)$ bezeichnet.

2. Die Instanz einer Relation mit dem Schema $R(U)$ wird mit r bezeichnet. Analog wird die Instanz einer Datenbank mit dem Datenbankschema D mit d bezeichnet. Weiterhin bedeutet $r \in |R(U)|$, daß die Instanz r einer Relation das Schema $R(U)$ besitzt.

3. Wenn eine Attributmenge X eine Attributmenge Y nicht funktional bestimmt, wird dies mit $X \not\rightarrow Y$ bezeichnet.

4. Die Konkatenation der Bezeichner der Attributmengen bedeutet die Vereinigung der Mengen, z. B. $XY = X \cup Y$

In der folgenden Definition[2] bezeichne $\pi_X(t)$ die Projektion des Tupels t aus der Relation r mit dem Schema R auf die Attributmenge X aus R.

Definition 2.3 *Eine Relation $r \in |R(U)|$ erfüllt die FA $X \rightarrow Y$, Notation $r \models X \rightarrow Y$, genau dann, wenn:*

$$r \models X \rightarrow Y \iff \forall t_1, t_2 \in r : (\pi_X(t_1) = \pi_X(t_2) \Rightarrow \pi_Y(t_1) = \pi_Y(t_2))$$

Beispiel 2.3 *Wenn KName → KAdr in KUNDE gelten soll, dann dürfen folgende zwei Tupel in KUNDE nicht auftauchen: t_1(Müller, Parkstraße, 1000) und t_2(Müller, Schloßallee, 1000).* ◇

Alternativ läßt sich die letzte Definition auch in Prädikatenlogik erster Stufe formulieren, wobei „Gleichheit" benötigt wird, [Kanellakis, 1990, S. 1093]. Dazu wird das Relationenschema $R(A_1, \ldots, A_n)$ auf ein n–stelliges Prädikat R abgebildet, wobei jedem Attribut genau eine Stelle in dem Prädikat entspricht. Variable werden durch große Buchstaben bezeichnet. Dann gilt:

$$r \models X \rightarrow Y \iff R(X, Y_1, C) \wedge R(X, Y_2, D) \Rightarrow Y_1 = Y_2$$

Definition 2.4 *Eine FA $X \rightarrow Y$ heißt genau dann* **trivial***, wenn*

$$\forall r \in |R(U)| : \ r \models X \rightarrow Y$$

Beispiel 2.4 *Die funktionale Abhängigkeit $LNameWare \rightarrow Ware$ ist trivial.* ◇

Lemma 2.1 *$X \rightarrow Y$ ist trivial $\iff Y \subseteq X$*

Beweis Folgt aus der Definition 2.3. □

Definition 2.5 *$X \subseteq U$ heißt* **Schlüssel** *eines Schemas $(R(U), F) \iff$*

$$\begin{array}{ll} (i) & F \models X \rightarrow U \text{ und} \\ (ii) & (F \models Y \rightarrow U) \wedge (Y \subseteq X) \Rightarrow Y = X \end{array}$$

D. h. X ist eine **minimale Attributmenge** *mit $F \models X \rightarrow U$.*

[2] Diese Definition mit Hilfe einer Formel aus der Relationen–Algebra ist in der Literatur gebräuchlich. Die Relationen–Algebra wird jedoch nicht eingeführt, da sie an keiner weiteren Stelle verwendet wird.

Existieren in einem Schema $(R(U), F)$ mehrere Mengen X_i, die einen Schlüssel bilden, so wird jede dieser Mengen X_i auch als „Schlüsselkandidat" bezeichnet.[3]

Beispiel 2.5 *In* LIEFERANT *ist* LName LAdr Ware *kein Schlüssel, da er nicht minimal ist.* LName Ware *ist jedoch ein Schlüssel.* LName *alleine ist wiederum kein Schlüssel, da* LName $\rightarrow$ Ware *nicht gilt.* $\diamond$

Definition 2.6 *Seien $X_1, \ldots, X_k$ Schlüsselkandidaten. Ein Attribut $A \in U$ heißt* **Nichtschlüssel–Attribut**, *kurz NSA,* $\iff$ $A \notin (X_1 \cup \ldots \cup X_k)$

Definition 2.7 *Seien $X \subseteq U, A \in U, A \notin X$. X heißt* **Determinante** *von A bzgl. eines Schemas $(R(U), F)$* $\iff$

$$\begin{array}{l} (i)\ \ F \models X \rightarrow A \ und \\ (ii)\ (F \models Y \rightarrow A) \wedge (Y \subseteq X) \Rightarrow Y = X \end{array}$$

D. h. eine Determinante X von A ist eine minimale Attributmenge X, von der A abhängt.

Beispiel 2.6 *In LIEFERANT ist LName eine Determinante für LAdr.* $\diamond$

2.2.1 Armstrongs Axiome

In diesem Abschnitt werden Inferenzregeln angegeben, die die logische Implikation zwischen funktionalen Abhängigkeiten beschreiben. Sie gehen auf Armstrong zurück, [Armstrong, 1974], und werden als Armstrongs Axiome bezeichnet.[4]

Definition 2.8 *Gegeben sei das Relationenschema R(U) und eine Menge F von FAen über U. F impliziert $X \rightarrow Y$, genau dann, wenn:*

$$F \models X \rightarrow Y \quad \iff \quad \forall r \in |R(U)| : (r \models F \Rightarrow r \models X \rightarrow Y)$$

Die **Hülle** von F, $F^* = \{X \rightarrow Y | F \models X \rightarrow Y\}$, wird durch die Menge aller von F implizierten FAen gebildet.

Armstrongs Axiome

(R) Reflexivität: Falls Y $\subseteq$ X $\subseteq$ U, dann gilt $F \models X \rightarrow Y$. Aus dieser Regel folgen die trivialen Abhängigkeiten. Die Anwendung dieser Regel hängt nur von U ab.

[3] Diese Unterscheidung rührt nur daher, daß existierende relationale Datenbanken immer die Definition nur eines Schlüssels pro Relation erlauben.

[4] Armstrongs Axiome sind keine Axiome im mathematischen Sinne, sondern vielmehr Regeln, die auf funktionale Abhängigkeiten angewendet werden können. Sie werden jedoch in der einschlägigen Datenbankliteratur immer als „Axiome" bezeichnet.

(E) Erweiterung: Falls $X \to Y$ gilt und $Z \subseteq U$, dann gilt $XZ \to YZ$.

(T) Transitivität: Falls $X \to Y$ und $Y \to Z$ gilt, dann folgt $X \to Z$.

In der Art, wie Armstrongs Axiome hier angegeben sind, stammen sie aus [Ullman, 1988, S. 384]. Mit diesen Axiomen ist es nun möglich, für jede mögliche funktionale Abhängigkeit $A \to B$ zu entscheiden, ob diese aus einer Menge F folgt, d. h. ob $A \to B \in F^*$ ist.

Diese Axiomatisierung wird als abgeschlossen[5] und vollständig betrachtet, wenn zwei Bedigungen erfüllt sind, [Kanellakis, 1990]. Dazu bezeichne $\vdash$ die Anwendung der Axiome auf funktionale Abhängigkeiten.

Definition 2.9 *Bezeichne F die Menge der funktionalen Abhängigkeiten und f eine einzelne funktionale Abhängigkeit. Ein Axiomensystem ist abgeschlossen, wenn aus $F \vdash f$ folgt, daß $F \models f$ gilt. Und es ist vollständig, wenn aus $F \models f$ folgt, daß $F \vdash f$ gilt.*

Theorem 2.1 *Armstrongs Axiome sind abgeschlossen und vollständig.*

Beweis Siehe z. B. [Ullman, 1988, S. 387]. □

Weiterhin gelten folgende Regeln, für deren Beweis das Theorem 2.1 benötigt wird. Der Beweis für dieses Lemma findet sich ebenfalls in [Ullman, 1988].

Lemma 2.2 (V) Vereinigung: $\{X \to Y, X \to Z\} \models X \to YZ$.

(PT) Pseudotransitivität: $\{X \to Y, WY \to Z\} \models WX \to Z$.

(D) Dekomposition: *Falls $X \to Y$ und $Z \subseteq Y$, dann folgt $X \to Z$.*

Korollar 2.1 $X \to A_1 \ldots A_n \Leftrightarrow (X \to A_1) \wedge \ldots \wedge (X \to A_n)$

Das Korollar ist eine Konsequenz aus den Regeln (V) und (D). Es besagt, daß es ausreichend ist, nur funktionale Abhängigkeiten mit einstelligen rechten Seiten zu betrachten. Dies bedeutet, daß bei der Suche nach FAen mit einstelligen rechten Seiten begonnen wird. Alle anderen FAen mit mehrstelligen rechten Seiten ergeben sich durch alle möglichen Kombinationen bei der Anwendung der Regel (V) aus Lemma 2.2.

[5] Der Begriff „abgeschlossen“ ist äquivalent zu dem Begriff „korrekt“.

2.2.2 Minimale funktionale Abhängigkeiten

In dieser Arbeit werden minimale funktionale Abhängigkeiten wie folgt definiert.

Definition 2.10 (Minimale funktionale Abhängigkeit)
Eine funktionale Abhängigkeit $X \to Y$ ist genau dann minimal, wenn:

1. *Die rechte Seite Y der FA genau ein Attribut enthält.*
2. *Für jede Teilmenge $Z \subseteq X$ mit $Z \to Y$ gilt: $Z = X$.*

Ullman definiert in [Ullman, 1988, S. 390], ob eine Menge von funktionalen Abhängigkeiten minimal ist. Dazu muß zuvor noch erklärt werden, wann zwei Mengen funktionaler Abhängigkeiten äquivalent sind.

Definition 2.11 *Zwei Mengen F und G funktionaler Abhängigkeiten sind genau dann äquivalent, wenn $F^* = G^*$ ist.*

Ullman zeigte folgendes Lemma in [Ullman, 1988, S. 390].

Lemma 2.3 *Jede Menge F funktionaler Abhängigkeiten ist äquivalent zu einer Menge G funktionaler Abhängigkeiten, in der keine rechte Seite mehr als ein Attribut hat.*

Die folgende Definition ist äquivalent zu Ullmans Definition. Sie verwendet direkt die Definition 2.10, die bei Ullman fehlt.

Definition 2.12 (Minimale Menge funktionaler Abhängigkeiten)
Eine Menge F funktionaler Abhängigkeiten ist genau dann minimal, wenn:

1. *Die rechte Seite jeder FA aus F genau ein Attribut enthält.*
2. *Für keine minimale FA $X \to A$ aus F ist die Menge $F \setminus \{X \to A\}$ äquivalent zu F.*

Aus dieser Definition läßt sich nicht die Existenz einer eindeutigen minimalen Menge funktionaler Abhängigkeiten ableiten. Es ist sehr wohl möglich, daß aus einer gegebenen Menge F von funktionalen Abhängigkeiten mindestens zwei minimale Mengen konstruiert werden können. Zusätzlich kann die Anzahl der FAen in diesen beiden minimalen Mengen auch noch unterschiedlich sein. Ullman gibt dazu ein Beispiel an, [Ullman, 1988, S. 391]. Dort findet sich ein entsprechender Algorithmus zur Konstruktion dieser minimalen Mengen funktionaler Abhängigkeiten.

Für die Menge F der funktionalen Abhängigkeiten, die der Algorithmus aus Kapitel 4 berechnet, gilt, daß jede einzelne funktionale Abhängigkeit minimal im Sinne von Definition 2.10 ist. Die Menge F insgesamt ist jedoch nicht minimal im Sinne von Definition 2.12.

Beispiel 2.7 *Der Algorithmus* FUNCTIONAL DEPENDENCIES *aus Kapitel 4 findet die Menge F der folgenden minimalen funktionalen Abhängigkeiten: $F = \{A \to B, A \to C, B \to C\}$. Dann ist $F \setminus \{A \to C\}$ äquivalent zu F. Dies ist ein Widerspruch zu (2) aus Definition 2.12.* ◇

Die folgenden beiden Beispiele zeigen, daß die Anwendung des Axioms (T) bzw. der Regel (PT) auf minimale funktionale Abhängigkeiten nicht wiederum zu minimalen funktionalen Abhängigkeiten führen muß. Das heißt, daß das oben angegebene Axiomensystem in bezug auf minimale funktionale Abhängigkeiten nicht abgeschlossen ist.

Beispiel 2.8 *Es seien die folgende Relation R und die minimalen funktionalen Abhängigkeiten $AB \to C$ und $C \to D$ gegeben. Die Ausnutzung der Transitivität liefert $AB \to D$. Diese FA ist jedoch nicht minimal, da $A \to D$ und $B \to D$ gilt!*

R	A	B	C	D
	a	a	1	1
	a	a	1	1
	c	b	3	2
	e	b	2	2
	a	c	4	1

◇

Beispiel 2.9 *Es seien die folgende Relation R und die minimalen funktionalen Abhängigkeiten $X_1X_2 \to Y$ und $W_1W_2Y \to Z$ gegeben. Die Anwendung von (PT) liefert $X_1X_2W_1W_2 \to Z$. Diese FA ist jedoch nicht minimal, da z. B. $X_2 \to Z$ gilt!*

R	X_1	X_2	Y	W_1	W_2	Z
	a	a	1	d	a	1
	a	a	1	d	b	2
	b	d	2	a	c	3
	c	e	4	a	c	4
	a	a	3	e	b	2
	a	a	3	f	b	5

◇

2.3 Integritätsbedingungen

In der Literatur wird der Begriff „Integritätsbedingung“ in verschiedenen Zusammenhängen verwendet. In dieser Arbeit wird er immer im Sinne einer „semantischen Integrität“ oder auch „Konsistenz“ verwendet. Damit ist die „Erhaltung der logischen Korrektheit einer Datenbank bei Änderungen durch (berechtigte) Benutzer“ gemeint, [Fuhr, 1991, S. 5.1]. Diese Art der Integrität wird im englischen mit „integrity“ bezeichnet. Daneben kann noch zwischen einer „Zugriffsintegrität“, der „operationalen“ oder „Ablauf–Integrität“ und der „physischen Integrität“ unterschieden werden.

Weiterhin werden unter dem Begriff semantische Integrität zwei Sachverhalte subsumiert. Da sind zunächst einmal die zwei generellen Integritätsregeln des relationalen Modells, [Date, 1990, Kap. 12].

Entity Integrity Rule Keine Komponente eines Primärschlüssels einer Relation darf NULL–Werte enthalten.

Referential Integrity Rule Die Datenbank darf keine Fremdschlüsselwerte enthalten, für die kein Primärschlüssel existiert.

Die erste Regel wurde schon im Abschnitt 2.1 behandelt. Zwischen der zweiten Regel, der referentiellen Integrität, und Fremdschlüsseln besteht kein Unterschied. Der eine Begriff ist ohne den anderen nicht erklärbar. Wird ein Attribut A aus einer Relation R_i als Fremdschlüssel bezüglich eines Attributs B aus R_j gekennzeichnet, so bedeutet dies, daß erstens B ein Primärschlüssel ist und zweitens, daß jeder Wert für A auch in B vorkommt, (siehe Abschnitt 2.1). Genau dieses besagt auch die Regel. Mit den beiden Begriffen Fremdschlüssel und referentielle Integrität sind auch die Inklusionsabhängigkeiten eng verbunden. Der nächste Abschnitt wird dies noch einmal aufgreifen.

Sodann gibt es neben diesen beiden Regeln, die sich auf das relationale Modell beziehen, Integritätsregeln, die sich auf den Inhalt einer Datenbank beziehen. Die letzteren sorgen dafür, daß das durch die Daten repräsentierte Modell konsistent bleibt. Im übrigen lassen sie sich in zwei Klassen unterteilen. Da ist zum einen die dynamische Integrität. Hierzu gehört das korrekte Verhalten der Datenbank bei Änderungen des Datenbankinhaltes, das hier aber nicht weiter betrachtet wird.

Und zum anderen gibt es eine statische Integrität. Zu dieser Art Integrität gehören zunächst Regeln, die sich auf alle Paare von Tupeln aus einer Relation beziehen, z. B. alle Namen der Lieferanten in LIEFERANT.LName müssen eindeutig sein. Diese Regeln werden durch die Auszeichnung einzelner Attribute als Schlüssel garantiert und sind im relationalen Modell modellinhärent. Daher fallen sämtliche funktionale Abhängigkeiten in diese Gruppe von Integritätsbedingungen. Weiterhin lassen sich Regeln für Attribute, einzelne Tupel einer Relation, Tupelgruppen oder Tupel aus verschiedenen Relationen formulieren. Beispiele hierzu finden sich in [Fuhr, 1991, Kap. 5].

Regeln für einzelne Attribute aus beliebigen Relationen sind z. B. Einschränkungen des Wertebereiches, indem Minima und Maxima für ein Attribut vorgegeben werden. Diese Vorgabe geschieht in der Regel nur für numerische Attribute, z. B. keine Ware darf mehr als 100DM kosten. Dies sind dann die „numerischen Integritätsbedingungen“, die in Kapitel 4 bestimmt werden.

Weitere numerische Integritätsbedingungen, die in der Literatur beschrieben werden, wie zum Beispiel der Durchschnitt der Apfelpreise muß kleiner sein als der Durchschnitt der Apfelsinenpreise, werden nicht betrachtet. Eine Integritätsbedingung wird analog zu FA und IA mit IB abgekürzt.

2.4 Inklusionsabhängigkeiten

Eine der beiden Hauptaufgaben des in dieser Arbeit entwickelten Programmes ist die Bestimmung von Teilmengenbeziehungen zwischen den Wertemengen einzelner Attribute, z. B. $LName \subseteq KName$. Für eine bestimmte Klasse von Attributen, die Fremdschlüssel, sind aufgrund der referentiellen Integrität diese Teilmengenbeziehungen schon gegeben. Daher sind die Inklusionsabhängigkeiten, so der Name dieser Teilmengenbeziehungen, auch eine Erweiterung der referentiellen Integrität auf alle Attribute. Die nun folgende Theorie formalisiert diese Inklusonsabhängigkeiten. Casanova et al. definieren Inklusionsabhängigkeiten wie folgt, [Casanova et al., 1984].

Definition 2.13 *Seien $R_i(A_1,\ldots,A_n)$ und $R_j(B_1,\ldots,B_m)$ zwei (nicht notwendig verschiedene) Relationenschemata. Falls X eine Sequenz aus k verschiedenen Attributen aus $A_1,\ldots,A_n$ ist und Y eine Sequenz aus k verschiedenen Attributen aus $B_1,\ldots,B_m$, dann nennen wir $R_i[X] \subseteq R_j[Y]$ eine* **Inklusionsabhängigkeit** *oder englisch* **inclusion dependency***, abgekürzt IA. Falls $r_1,\ldots,r_s$ eine Datenbank d über $D = \{R_1(U_1),\ldots,R_n(U_n)\}$ ist, dann erfüllt d die IA $R_i[X] \subseteq R_j[Y]$, falls $r_i[X] \subseteq r_j[Y]$. Für $k = 1$ handelt es sich um einstellige oder* **unäre Inklusionsabhängigkeiten***.*

Genau wie für funktionale Abhängigkeiten läßt sich für Inklusionsabhängigkeiten ein Axiomensystem angeben. Das folgende Axiomensystems ist entnommen aus [Casanova et al., 1984].

(R) Reflexivität: R[X] $\subseteq$ R[X], falls X eine Sequenz aus verschiedenen Attributen aus R ist.

(P) Projektion und Permutation: Falls R[$A_1,\ldots,A_m$] $\subseteq$ S[$B_1,\ldots,B_m$] gilt, dann gilt auch R[$A_{i_1},\ldots,A_{i_k}$] $\subseteq$ S[$B_{i_1},\ldots,B_{i_k}$] für jede Sequenz i_1, ..., i_k von verschiedenen ganzen Zahlen aus $\{1,\ldots,m\}$.

(T) Transitivität: Falls R[X]$\subseteq$S[Y] und S[Y]$\subseteq$T[Z] gilt, dann gilt auch R[X]$\subseteq$T[Z].

Notation

1. Wenn aus dem Zusammenhang hervorgeht, aus welchen Relationen die Sequenzen der Attribute stammen, dann werden IAen auch kurz mit $X \subseteq Y$ bezeichnet.

2. Wenn die IA R[X]$\subseteq$S[Y] nicht gilt, wird dies mit R[X]$\not\subseteq$S[Y] bezeichnet.
3. Wenn R[X]$\subseteq$S[Y] und S[Y]$\not\subseteq$R[X] gilt, wird dies mit R[X]$\subset$S[Y] bezeichnet.

Beispiel 2.10 *Seien die Relationen ANGESTELLTER(PNr, Name, ..., ZimmerNr, Gehalt), ADRESSEN(..., HausNr, ...) und MANAGER(PNr, Name, Telefon) gegeben. Dann können in einer Datenbank z. B. die folgenden IAen gelten:*

1. *MANAGER[PNr]* $\subseteq$ *ANGESTELLTER[PNr]*
2. *MANAGER[Name]* $\subseteq$ *ANGESTELLTER[Name]*
3. *MANAGER[Telefon]* $\subseteq$ *ANGESTELLTER[Gehalt]*
4. *ADRESSEN[HausNr]* $\subseteq$ *ANGESTELLTER[ZimmerNr]*

$\diamond$

Der dritte Punkt soll verdeutlichen, daß IAen ein rein syntaktisches Konzept sind, das nur auf dem Vergleich von Werten beruht. Alle Telefonnummern der Manager sind zufällig identisch mit Gehältern einzelner Angestellter. Auch der spätere Algorithmus benutzt zur Entscheidung der Frage, ob gefundene IAen für die Anwendung, die durch die Datenbank modelliert wird, irgendeine Bedeutung haben, kein Wissen, das von seiten des Anwenders vorher eingegeben wurde. Das heißt insbesondere, daß „Zufallstreffer" wie MANAGER[Telefon] $\subseteq$ Angestellter[Gehalt] vom Benutzer des Systems erkannt und entsprechend behandelt werden müssen.

Weiterhin sind Inklusionsabhängigkeiten, zumindest wie sie in der Definition 2.13 eingeführt wurden, typenlos. Datenbanken hingegen verwenden eine Vielzahl von Typen für die verschiedenen Attribute, sei es, um die Daten dann effizienter speichern zu können, sei es, um bei der Dateneingabe Eingabefehler direkt erkennen zu können. Zu ersterem gehören in Oracle 7 z. B. die Typen VARCHAR2 und CHAR. Der Unterschied zwischen diesen beiden besteht darin, daß einmal für die Zeichenketten, die als Werte erlaubt sind, nur soviel Platz reserviert wird, wie die Zeichenkette lang ist. Im anderen Fall ist der Speicherplatz stets konstant. Aus Sicht des Inklusionsabhängigkeiten suchenden Benutzers, erscheint diese Unterscheidung jedoch irrelevant, da die Werte die gleichen bleiben.

Anders sieht die Sache jedoch aus, wenn die angebotenen Typen dazu verwendet werden, die Dateneingabe sicherer zu gestalten. Dazu gehört z. B. die Verwendung von Typen aus der „Klasse NUMBER" für Zahlen und der „Klasse VARCHAR" für Zeichenketten. Für diese Unterscheidung findet sich auch eine Entsprechung in der Anwendung, für die die Datenbank ein Modell ist. In dem Algorithmus Inclusion Dependencies werden alle Attribute in diese beiden Klassen eingeteilt und nur noch IAen innerhalb dieser Klassen gesucht. Dies führt dazu, daß Zufallstreffer wie ADRESSEN[HausNr] $\subseteq$ ANGESTELLTER[ZimmerNr] ausgeschlossen werden, vorausgesetzt HausNr ist vom Typ VARCHAR und ZimmerNr vom Typ NUMBER.

2.5 Ein Axiomensystem für unäre Inklusionsabhängigkeiten und funktionale Abhängigkeiten

In diesem Abschnitt wird ein Axiomensystem angegeben, mit dessen Hilfe Aussagen über funktionale Abhängigkeiten und Inklusionsabhängigkeiten gleichermaßen abgeleitet werden können. Dafür wird eine weitere Abhängigkeit benötigt, die Kardinalitätsabhängigkeit, [Kanellakis et al., 1983].

Definition 2.14 *Die Kardinalitätsabhängigkeit $|A| \leq |B|$ gilt genau dann, wenn die Kardinalität des Attributes A kleiner ist als die Kardinalität des Attributes B.*

Bei der Bestimmung der Kardinalität $|A|$ werden nur die verschiedenen Werte für das Attribut A gezählt. Ist die Anzahl für A echt kleiner als $|B|$, so wird dies mit $|A| < |B|$ bezeichnet. Weiterhin gilt, daß die Kardinalitätsabhängigkeiten reflexiv und transitiv sind.

Die folgende Definition ist aus [Bell, 1994] entnommen.

Definition 2.15 (Endliche Axiomatisierung der FAen und unären IAen) *Zusätzlich zu Armstrongs Axiomen, Seite 12, und den Axiomen für Inklusionsabhängigkeiten, Seite 17, gelten die folgenden Regeln:*

- $A \rightarrow B \Rightarrow |B| \leq |A|$
- $A \subset B \Rightarrow |A| \leq |B|$
- $A \rightarrow B, |A| \leq |B| \Rightarrow B \rightarrow A$
- $B \subseteq A, |A| \leq |B| \Rightarrow A \subseteq B$

Kanellakis et al. zeigen, daß diese Axiomatisierung für FAen und IAen abgeschlossen und vollständig ist und daß ein polynomieller Entscheidungsalgorithmus existiert, [Kanellakis et al., 1983]. In dem Algorithmus FUNCTIONAL DEPENDENCIES wird die folgende Regel verwendet, die eine Konsequenz aus der Definition 2.15 ist.

Lemma 2.4 *Seien die Attribute $A, B \in R(U)$ gegeben, und sei $R[A] \subset R[B]$. Dann gilt $A \not\rightarrow B$.*

Beweis Sei o. B. d. A. (A, B) das Relationenschema von R, und enthalte R insgesamt m Tupel. Da $B \not\subset A$ ist, gilt $|B| = k$ und $|A| = l$ mit $l < k \leq m$. Seien A_1, A_2, B_1 und B_2 Mengen von Tupeln wie in Abbildung 2.3 dargestellt[6]. O. b. d. A. gelte $B_1 = A_1 \cup A_2$ und $B_2 \cap (A_1 \cup A_2) = \emptyset$. Daraus folgt $|B_1| = l$ und $|B_2| \geq 1$, und somit ist $|A_2| \geq 1$ und $|A_1| = r < l$. Es ist zu zeigen, daß zwei Tupel (a, b_p) und (a, b_q) existieren mit $b_p \neq b_q$. Die beiden folgenden Fälle sind zu unterscheiden:

[6] Diese Darstellung kann durch eine Sortierung der Tupel immer erreicht werden.

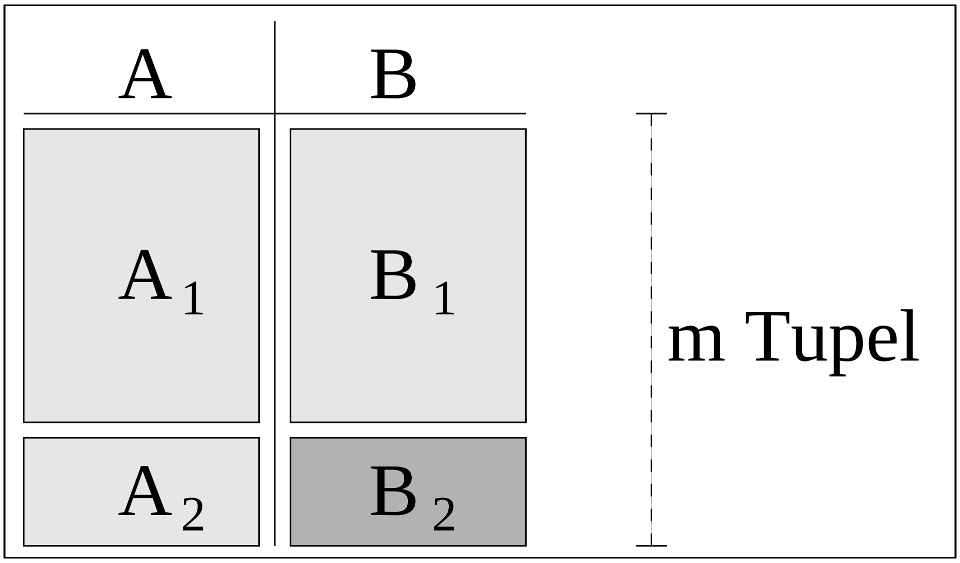

Abbildung 2.3: Skizze zum Beweis von Lemma 2.4

1. Sei $A_1 \cap A_2 = \emptyset$. Es gilt $|A_1| = r < l = |B_1|$. Daraus folgt, daß mindestens zwei Tupel $t_1(a_j, b_p)$ und $t_2(a_j, b_q)$ mit $a_j \in A_1$ und $b_p \neq b_q$ exitieren. Somit folgt die Behauptung $A \not\rightarrow B$.

2. Sei $A_1 \cap A_2 \neq \emptyset$. Dann existieren (mindestens) zwei Tupel $t_1(a_{j_1}, b_p)$ und $t_2(a_{j_2}, b_q)$ mit $a_{j_1} \in A_1$, $a_{j_2} \in A_2$ und $a_{j_1} = a_{j_2}$. Somit ist $b_p \in B_1$ und $b_q \in B_2$. Da $B_1 \cap B_2 = \emptyset$ gilt, folgt $b_p \neq b_q$ und damit die Behauptung $A \not\rightarrow B$.

□

2.6 SQL

In diesem Abschnitt werden zur Wiederholung drei Konstrukte der Datenbankabfragesprache SQL vorgestellt, die unter anderem im weiteren Verlauf der Arbeit verwendet werden. Für eine allgemeine Einführung in die Sprache SQL sei auf das Buch [Date, 1987] verwiesen. Die drei Konstrukte sind im einzelnen:

1. SELECT COUNT (DISTINCT A) vs. SELECT COUNT (A)

2. SELECT $R_k.A_i$, $R_l.A_j$ FROM R_k, R_l WHERE $R_k.A_i = R_l.A_j$

3. SELECT ... FROM ... [WHERE ...] GROUP BY $A_1 \ldots A_n$

Sowohl COUNT (A) als auch COUNT (DISTINCT A) zählen die Anzahl der Werte für das Attribut A in allen Tupeln, die durch die Anfrage insgesamt selektiert werden. Für beide Alternativen gilt, daß NULL–Werte nicht mitgezählt werden. Der Unterschied besteht darin, daß bei Verwendung von DISTINCT nur verschiedene Werte gezählt werden, während sonst alle Werte gezählt werden.

Beispiel 2.11
SELECT COUNT (A_1) *FROM* R_1 = *3*
SELECT COUNT (*DISTINCT* A_1) *FROM* R_1 = *3*,
SELECT COUNT (A_2) *FROM* R_1 = *4*,
SELECT COUNT (*DISTINCT* A_2) *FROM* R_1 = *2*

R_1	A_1	A_2
	1	*b*
	2	*a*
	3	*b*
	NULL	*b*

◇

Die Anfrage (2) beschreibt den „θ–Join“, mit θ gleich „=“, der dann auch „Equijoin“ genannt wird. Eine Realisierung besteht darin, daß zuerst das kartesische Produkt zwischen den Attributen $R_k.A_i$ und $R_l.A_j$ gebildet wird. Danach werden alle Tupel, auf die die WHERE–Bedingung nicht zutrifft, wieder eliminiert. Zuletzt wird die Projektion auf die Attribute in der SELECT–Anweisung gebildet. Wird hier das Schlüsselwort DISTINCT zusätzlich verwendet, so werden doppelte Tupel entfernt.

Beispiel 2.12
SELECT $R_1.A_2$, $R_2.A_3$ *FROM* R_1, R_2 *WHERE* $R_1.A_2 = R_2.A_3$
SELECT DISTINCT $R_1.A_2$, $R_2.A_1$ *FROM* R_1, R_2 *WHERE* $R_1.A_1 = R_2.A_1$
Tabelle [E1] zeigt das Ergebnis der ersten und [E2] das Ergebnis der zweiten SQL–Anfrage.

R_1	A_1	A_2
	1	*b*
	2	*a*
	3	*b*
	NULL	*b*

R_2	A_1	A_2	A_3
	1	*1*	*c*
	1	*2*	*e*
	2	*1*	*b*
	2	*2*	*a*
	3	*5*	*b*
	7	*4*	*f*

[E1]: $R_1 \bowtie R_2$	$R_1.A_2$	$R_2.A_3$
	b	*b*
	b	*b*
	a	*a*
	b	*b*
	b	*b*
	b	*b*
	b	*b*

[E2]: $R_1 \bowtie R_2$	$R_1.A_2$	$R_2.A_1$
	b	*1*
	a	*2*
	b	*3*

◇

Die Anweisung (3) sorgt für eine Einteilung der Attribute $A_1 \dots A_n$ in der GROUP BY Anweisung in Gruppen. Für jede Gruppe gilt, daß alle Werte für die Attribute $A_1 \dots A_n$ identisch sind. In der SELECT–Anweisung dürfen entweder nur Attribute vorkommen, die in der GROUP BY Liste stehen, oder Aggregationsfunktionen wie SUM, COUNT usw. Diese Funktionen dürfen sich auf alle Attribute beziehen.

Beispiel 2.13 *Die Tabelle [T1] zeigt das Resultat der GROUP BY Anweisung, auf dessen Basis dann das Ergebnis berechnet wird. Die Ausgabe dieser SQL–Anweisung wird in Tabelle [T2] gezeigt.*

SELECT COUNT (A,B,C,D,E) FROM R
WHERE A>1 GROUP BY A, B

R	A	B	C	D	E
	3	3	11	d	f
	2	1	7	d	f
	2	2	9	c	i
	3	3	11	d	g
	1	2	9	a	p
	2	2	9	c	h

[T1]: R	A	B	C	D	E
	2	1	7	d	f
	2	2	9	c	i
	2	2	9	c	h
	3	3	11	d	f
	3	3	11	d	g

[T2]: COUNT A	COUNT B	COUNT C	COUNT D	COUNT E
1	1	1	1	1
2	2	2	2	2
2	2	2	2	2

◇

Kapitel 3

Technische Realisierung

Die Eingabe für das Programm bzw. die drei aufeinander aufbauenden Algorithmen sind die Daten einer Datenbank mit dem dazugehörigen Datenbankschema. Für die Implementierung wird das DBMS Oracle 7 verwendet. Die Kommunikation zwischen dem in Prolog geschriebenen Programm und dem DBMS geschieht über ein Netzwerk mit dem TCP/IP Protokoll. Weiterhin ist es hierfür erforderlich, eine Umsetzung zwischen Prolog und der Datenbankabfragesprache SQL zu realisieren. Die in Prolog generierten SQL–Anfragen werden mit Hilfe einer in „C“ geschriebenen Schnittstelle an Oracle V7 weitergereicht. Die Antwort auf eine SQL–Anfrage ist eine Menge von Tupeln, wobei die Werte der Tupel in den Datenformaten von Oracle vorliegen. Diese Menge von Tupeln wird in der Schnittstelle in eine Liste von Prolog Atomen umgewandelt. Dabei werden alle numerischen Daten in den Prolog Zahlentyp und alle symbolischen Datentypen in Prolog Atome umgewandelt. Weiterhin erfahren Daten der Oracle Datentypen RAW und DATE eine Sonderbehandlung. RAW–Daten werden als reine binäre Daten herausgefiltert und nicht weitergereicht. Daten vom Typ DATE werden auch in Atome gewandelt.

Die drei Algorithmen bauen hierarchisch aufeinander auf. So wird das Ergebnis der Berechnung der numerischen Integritätsbedingungen zur Bestimmung der Inklusionsabhängigkeiten zwischen numerischen Attributen verwendet. Für die Attribute des Typs VARCHAR werden auch zuerst Minima und Maxima bzgl. der lexikographischen Ordnung bestimmt, ohne diese jedoch extra auszugeben. Anschließend werden die IAen berechnet. Diejenigen IAen, die innerhalb einer Relation gültig sind, werden zur Berechnung der funktionalen Abhängigkeiten weiterverwendet. Die Ausgabe des Programms bildet eine Liste sämtlicher numerischer Integritätsbedingungen sowie eine Liste der funktionalen Abhängigkeiten und der unären Inklusionsabhängigkeiten.

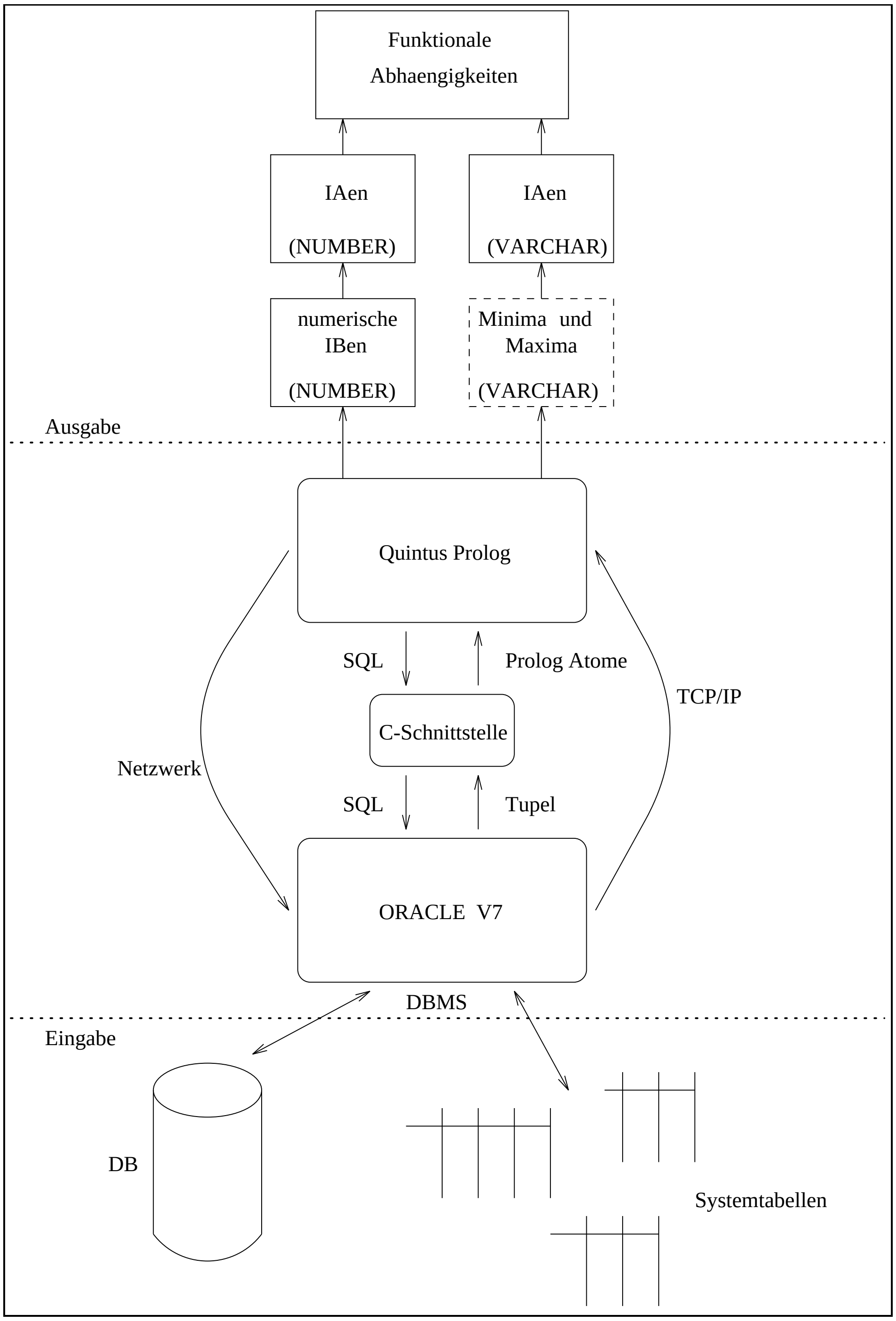

Abbildung 3.1: Die Architektur des Programms im Überblick

Kapitel 4

Bestimmung unärer Inklusionsabhängigkeiten und funktionaler Abhängigkeiten

In diesem Kapitel werden die drei Algorithmen zur Berechnung der numerischen Integritätsbedingungen, der unären Inklusionsabhängigkeiten und der funktionalen Abhängigkeiten vorgestellt. Jeder einzelne wird im Detail beschrieben und seine Vollständigkeit und Korrektheit gezeigt. Weiterhin werden auch die Laufzeiten der Algorithmen angegeben.

4.1 Numerische Integritätsbedingungen

Der folgende Algorithmus gibt in einem „pascalähnlichen" Pseudocode die Berechnung der Minima und Maxima numerischer Attribute an. Dafür wird vorausgesetzt, daß von allen Attributen der Relationen der Typ bekannt ist. Um diesen zu ermitteln wird die in Abbildung 2.2 schematisch gezeigte Systemtabelle des DBMS ausgelesen. Die Details hiervon können im Quelltext des Programms nachgelesen werden.

Da der Algorithmus ohne Änderung auch für Attribute des Typs VARCHAR funktioniert, bekommt er als Eingabe alle Attribute der Relationen ohne Berücksichtigung des Typs der Attribute. Die SQL–Anweisungen Min(A) bzw. Max(A) bestimmen bei numerischen Attributen erwartungsgemäß die Minima und Maxima. Handelt es sich jedoch bei dem Attribut A um den Typ VARCHAR, so werden die Minima und Maxima der Zeichenketten auf der Basis der lexikographischen Ordnung bestimmt. Diese Ordnung benutzt allein die ASCII–Zahlenwerte der Zeichen. Danach ist $0, \ldots, 9 < A, \ldots, Z < a, \ldots, z$. Erweitert auf Zeichenketten bedeutet dies, daß z. B. $e > ZZ$ und $aa < aaa$ ist. Zeichenketten werden nur bis zum jeweils letzten Zeichen der kürzeren Zeichenkette verglichen.

Bei dieser Ordnung wird die Länge der Zeichenketten nicht berücksichtigt. Alternativ wäre es möglich, diese Ordnung zweistufig zu definieren. Im ersten Schritt wird die Länge der Zeichenketten verglichen, so daß alle Zeichenketten der Länge k kleiner sind als alle Zeichenketten der Länge k+1. Und im zweiten Schritt wird dann die obige Ordnung benutzt. Dies geschieht hier jedoch nicht.

Die in dem Algorithmus MINMAX berechneten Minima und Maxima der Attribute des Typs VARCHAR werden nicht extra an den Benutzer ausgegeben. Sie dienen allein dazu, eine Vorauswahl für den Algorithmus INCLUSION DEPENDENCIES treffen zu können. Dies geschieht in dem Algorithmus INTERVALLE.

Für den Algorithmus MINMAX ist es im Prinzip gleichgültig, in welcher Reihenfolge die Attribute abgearbeitet, sprich die SQL–Anfragen gestellt werden, solange jedes Attribut einmal an die Reihe kommt. Es werden jedoch trotzdem erst alle Attribute einer Relation R_i durchlaufen, bevor mit den Attributen der Relation R_{i+1} fortgefahren wird. Dies geschieht allein aus der Vermutung, daß das DBMS ab dem zweiten Zugriff auf die gleiche Relation diese Anfragen optimieren kann. Zumindest bei der Datenbank Oracle 7 ist dieses Vorgehen begründet. Eine Vertiefung dieser Frage ginge jedoch über den Rahmen dieser Arbeit hinaus.

Algorithmus: MINMAX

Eingabe: Eine Liste L von Paaren mit folgendem Aufbau:
$[(R_1, A_{11}),\ldots,(R_1, A_{1r}),\ldots,(R_m, A_{m1}),\ldots,(R_m, A_{ms})]$,
wobei R_i die Relationen und A_{ij} die Attribute der Relation R_i bezeichne.

Ausgabe: Eine Liste ERG von Quadrupeln:
$[(R_1, A_{11}, min, max), \ldots, (R_m, A_{ms}, min, max)]$

```
ERG := {}
while L ≠ {} do
      (R,A) := Kopf(L)
      (min,max) := SELECT MIN(A),MAX(A)
                   FROM R
      ERG := [ERG | (R,A,min,max)]
      L := Rumpf(L)
od
return(ERG)
```

Ende

In dem Algorithmus gibt Kopf(L) das erste Element der Liste zurück und Rumpf(L) die Liste ohne das erste Element. Das Ergebnis der SQL–Anfrage ist ein zweielementiges Tupel, das dem Paar (min,max) zugewiesen wird.

4.1.1 Eigenschaften des Algorithmus

Die Liste L in dem Algorithmus MINMAX enthält n Elemente und damit n Attribute. Jede Relation R_i enthält m_i Tupel. Für die folgende wie auch alle weiteren Aussagen bezeichne n jeweils die Anzahl der Attribute. Die Zahl der Tupel einer Relation R_i, oder wenn die Attribute aus mehreren Relationen stammen, die Anzahl der Tupel der Relation R_i mit den meisten Tupeln, wird mit m bezeichnet.[1]

Lemma 4.1 *Die Laufzeit des Algorithmus beträgt* $\mathcal{O}(n * m)$.

Beweis Die Liste L enthält n Elemente, folglich wird der Algorithmus n–mal durchlaufen. Das Durchsuchen von m Tupeln eines Attributes nach dem Minimum und Maximum, welches in einem Durchlauf geschehen kann, benötigt $2m$ Vergleiche und damit $\mathcal{O}(m)$ Zeit. Beide Ergebnisse zusammen liefern $\mathcal{O}(n * m)$. □

Es ist leicht ersichtlich, daß der Algorithmus MINMAX unter der Voraussetzung, daß die Ergebniss der SQL–Anfragen korrekt sind, selbst vollständig und korrekt ist, d. h. es werden für alle Attribute die korrekten Minima und Maxima berechnet.

4.2 Unäre Inklusionsabhängigkeiten

Die Berechnung der unären Inklusionsabhängigkeiten in dem Algorithmus INCLUSION DEPENDENCIES geschieht grundsätzlich getrennt für Attribute der beiden Typen NUMBER und VARCHAR. Um diesen Algorithmus zur Berechnung aller existierenden IAen zu beschleunigen, wird zuerst eine Vorauswahl getroffen. Dies beschreibt der nächste Abschnitt.

4.2.1 Teilmengenbeziehungen zwischen Wertebereichen

Damit eine IA $R_i[A] \subseteq R_j[B]$ existiert, müssen alle Werte, die für das Attribute A vorkommen, auch als Werte für das Attribut B vorkommen. Nun wurden im vorherigen Abschnitt schon für jedes Attribut zwei besondere Werte bestimmt, nämlich die Minima und Maxima der Attribute. Die folgende Beobachtung verhilft dazu, die Menge der Attribute, die überhaupt nur für IAen in Frage kommen, durch eine Vorauswahl einzuschränken.

Gegeben seien zwei geordenete Mengen A und B. Damit A eine Teilmenge der Menge B sein kann, muß mindestens das minimale sowie das maximale Element von A auch Element von B sein. Ist diese notwendige Bedingung nicht erfüllt, so kann A keine Teilmenge von B sein und die hiermit korrespondierende unäre Inklusionsabhängigkeit $R_i[A] \subseteq R_j[B]$ nicht existieren.

[1] Dadurch lassen sich die Laufzeiten bei mehreren Relationen mit unterschiedlichen Anzahlen an Tupel einfacher abschätzen und angeben.

Diese Überlegung wird vom Algorithmus INTERVALLE zur Berechnung einer Vorauswahl von Attributen ausgenutzt. Dazu werden die Ergebnisse aus Abschnitt 4.1 verwendet. Bleibt anzumerken, daß dieser Algorithmus zweimal durchlaufen wird. Die lexikographische Ordnung wird nicht mehr zur gemeinsamen Berechnung der IAen ausgenutzt, weil es Fälle gibt, in denen symbolische Werte mit Zahlen beginnen, wie das nächste Beispiel zeigt.[2]

Beispiel 4.1 *Seien „31“ und „79“ bzw. „10, Rue du docteur Carrel“ und „Beethovenstr. 54“ die Minima und Maxima des numerischen Attributes A und des symbolischen Attributes B. Die Ausnutzung der lexikographischen Ordnung liefert, daß das von A aufgespannte Intervall in B enthalten ist.* $\diamond$

Algorithmus: INTERVALLE

Eingabe: Eine Liste der n Attribute aus allen Relationen, die vom gleichen Typ sind, zusammen mit ihren Minima und Maxima.

Ausgabe: Eine Liste ERG der Attribute, für die 2 gilt.

1. Berechne alle $n * (n-1)$ Möglichkeiten für $R_i[A] \subseteq R_j[B]$ bzw. $R_j[B] \subseteq R_i[A]$.
2. Fasse alle Attribute zu einer Liste ERG zusammen, für die gilt:
   (a) Das Intervall von A ist in B enthalten: $min(A) \geq min(B)$ und $max(A) \leq max(B)$.
   (b) Oder das Intervall von B ist in A enthalten: $min(B) \geq min(A)$ und $max(B) \leq max(A)$.
3. return(ERG)

Ende

Lemma 4.2 *Die Laufzeit des Algorithmus beträgt $\mathcal{O}(n^2)$.*

Beweis Es gibt n Attribute eines Typs und $n*(n-1)$ Möglichkeiten, zwei verschiedene Attribute zu kombinieren. Die für die Vergleiche der Minima und Maxima benötigte Zeit ist jedesmal konstant. $\square$

Während der Bestimmung dieser Vorauswahl könnten sicherlich die transitiven Beziehungen zwischen den Intervallen ausgenutzt werden, die durch die Maxima und Minima aufgespannt werden.

[2] Dieses würde zu ähnlichen Effekten führen wie in Beispiel 2.10; vgl. auch die sich dort anschließende Diskussion. Eine Erkennung dieser Fälle mit entsprechender Behandlung ist aufwendiger als der zweimalige Aufruf des Algorithmus INTERVALLE.

Beispiel 4.2

$$\Big([-1,23] \subseteq [-19,53]\Big) \wedge \Big([-19,53] \subseteq [-145,312]\Big) \Rightarrow [-1,23] \subseteq [-145,312]$$

$\diamond$

Der zusätzliche Aufwand zur Erkennung dieser Transitivität ist bei dem Algorithmus INTERVALLE nicht gerechtfertigt, weil er keine Anfragen an die Datenbank stellt, deren Einsparung immer oberstes Ziel sein muß. Beim nächsten Algorithmus ist das jedoch nicht der Fall. Dort wird gerade versucht, transitive Beziehungen auszunutzen.

4.2.2 Bestimmung der unären Inklusionsabhängigkeiten

Die folgenden grundsätzlichen Überlegungen gelten sowohl für numerische als auch für symbolische Attribute. Weiterhin sind die folgenden drei SQL–Anweisungen unabhängig davon, von welchem Typ die Attribute sind. Anhand der Tupel, die diese drei Anfragen als Ergebnis zurückliefern, läßt sich feststellen, ob eine Inklusionsabhängigkeit zwischen zwei Attributen A_1 und A_2 existiert, d. h. ob $A_1 \subseteq A_2$ oder $A_2 \subseteq A_1$ oder beides gilt.

1. SELECT COUNT(DISTINCT $R_i.A_1$)
   FROM R_i, R_j
   WHERE $R_i.A_1 = R_j.A_2$ $=: e$

2. SELECT COUNT(DISTINCT A_1)
   FROM R_i $=: e_1$

3. SELECT COUNT(DISTINCT A_2)
   FROM R_j $=: e_2$

Bei diesen drei Anfragen fällt auf, daß die zweite und dritte Anfrage, zumindest wenn $R_i = R_j$ ist, zu einer Anfrage zusammengefaßt werden können. Da es bei dem späteren Algorithmus aber Fälle gibt, bei denen eine derartige Möglichkeit von vornherein ausgeschlossen werden kann, geschieht dies nicht.

Das Ergebnis dieser drei SQL–Anfragen ist jeweils ein einelementiges Tupel, bestehend aus einer Zahl, die in e, e_1 und e_2 gespeichert wird. Die folgenden drei Gleichungen geben die Art der Inklusionsabhängigkeit an, falls eine existiert.

1. $e = e_1 \Rightarrow A_1 \subseteq A_2$
2. $e = e_2 \Rightarrow A_2 \subseteq A_1$

3. $e = e_1 = e_2 \Rightarrow A_1 \subseteq A_2 \wedge A_2 \subseteq A_1$, also $A_1 = A_2$

Ein naiver Algorithmus, der alle unären Inklusionsabhängigkeiten berechnet, sieht wie folgt aus: Berechne bei n Attributen eines Typs alle $\mathcal{O}(n^2)$ Kombinationen für $A \subseteq B$ oder $B \subseteq A$. Stelle für jedes Paar A, B die drei SQL–Anfragen und bestimme mit Hilfe der drei Gleichungen die Art der Inklusionsabhängigkeit.

Dieser naive Algorithmus hat eine Laufzeit von $\mathcal{O}(n^2 * m^2)$. Der Faktor n^2 ist klar, da es $n * (n-1)$ Möglichkeiten gibt, zwei verschiedene Attribute zu kombinieren. Bleiben die SQL–Anfragen. Die aufwendigste Anfrage ist die erste mit dem Join. Dieser benötigt bei maximal m Tupeln $\mathcal{O}(m^2)$. Dies zusammen ergibt die Gesamtlaufzeit.

Da in der Praxis verwendete Datenbanken um mehrere Zehnerpotenzen mehr Tupel beinhalten als die Relationen Attribute, muß das Ziel sein, möglichst viele Datenbankzugriffe einzusparen. Um dieses Ziel zu erreichen, werden zwei Dinge ausgenutzt. Zum einen wird die Systemtabelle auf Fremdschlüsseleinträge hin durchsucht. Für alle Fremdschlüsselattribute gilt, daß ihre Werte eine Teilmenge der Wertemenge des entsprechenden anderen, referenzierten Attributes sind. Für alle diese Attribute gelten die entsprechenden IAen trivialer Weise, da sie vom Datenbankmanagementsystem, zumindest bei Oracle 7, garantiert werden.

Zum anderen sind IAen transitiv. Diese Transitivität kann von einem entsprechenden Algorithmus ausgenutzt werden, um Datenbankanfragen einzusparen. Der obige naive Algorithmus durchläuft auch im günstigsten Fall $\mathcal{O}(n^2)$ mal die drei SQL–Anfragen. Dieses wird im Algorithmus INCLUSION DEPENDENCIES anders sein.

4.2.3 Ein Algorithmus für Inklusionsabhängigkeiten

Der nun folgende Algorithmus INCLUSION DEPENDENCIES nutzt die Transitivität der Inklusionsabhängigkeiten aus. Dazu berechnet er nach jeder neuen gefundenen IA alle diejenigen IAen, die sich aufgrund der Tansitivität ergeben und noch nicht an der Datenbank überprüft wurden. Damit dieses effizient durchgeführt werden kann, wird ein gerichteter Graph $G(V, E)$ verwendet. Die Menge der Knoten wird mit A_1 bis A_n bezeichnet, wobei $A_1 = 1$ und $A_n = n$ ist. Weiterhin wird eine injektive Funktion $label(R_i.A_j)$, die allen Attributen $R_i.A_j$ eine ganze Zahl A_k zuordnet, vorausgesetzt. Die Zahlen von A_1 bis A_n werden benötigt, da die natürliche Ordnung auf ihnen in dem Algorithmus ausgenutzt wird. In dem Graphen G existiert genau dann eine Kante von A_i nach A_j, wenn es eine unäre Inklusionsabhängigkeit der Art $R_p[A_k] \subseteq R_q[A_l]$ gibt. Außerdem wird im folgenden Algorithmus nicht extra zwischen Kanten von A_i nach A_j und den IAen, die sie repräsentieren, unterschieden. Bei $A_i \subseteq A_j$ ist jeweils sinngemäß entweder die Kante oder die IA gemeint.

Algorithmus: INCLUSION DEPENDENCIES

Eingabe: Eine Menge von Attributen eines Typs

Ausgabe: Eine Liste aller unären Inklusionsabhängigkeiten.

1. Numeriere alle Attribute $R_i.A_j$ von A_1 bis A_n durch.
2. Baue einen gerichteten Graphen auf mit allen Knoten A_i. Weiterhin enthalte dieser Graph Kanten $A_i \rightarrow A_j$ genau dann, wenn für A_i in der Systemtabelle der Fremdschlüssel A_j eingetragen ist.
3. Baue folgende Listenstruktur auf:
   $$\Big[[A_1 : [\underline{A_2}, \overline{A_2}], [\underline{A_3}, \overline{A_3}], \ldots, [\underline{A_n}, \overline{A_n}]] \\ [A_2 : [\underline{A_3}, \overline{A_3}], \ldots, [\underline{A_n}, \overline{A_n}]] \\ \vdots \\ [A_{n-1} : [\underline{A_n}, \overline{A_n}]] \Big]$$
   $\underline{A_j}$ bzw. $\overline{A_j}$ bedeutet, es muß noch getestet werden, ob $A_i \subseteq A_j$ bzw. $A_j \subseteq A_i$ gilt. Dabei enthalte die Liste von A_i genau dann $\underline{A_j}$ bzw. $\overline{A_j}$ mit $j > i$, wenn es keinen Weg von A_i nach A_j bzw. von A_j nach A_i gibt.
4. Führe folgende Schritte für A_i mit $1 \leq i < n$ aus:
   (a) Sei $\underline{A_{i+r}}$ mit $r \in \{1, \ldots, n-i\}$ der nächste Test. Gibt es eine Kante von A_{i+r} zu einem Knoten A_k mit $k < i$ und keine Kante von A_i nach A_k, dann mache mit dem nächsten Test in 4b weiter, sonst führe den Test aus. Gilt $A_i \subseteq A_{i+r}$, dann rufe erst UPDATE mit $A_i \subseteq A_{i+r}$ auf und mache dann bei 4b weiter, sonst gehe direkt zu 4b.
   (b) Sei $\overline{A_{i+r}}$ mit $r \in \{1, \ldots, n-i\}$ der nächste Test. Gibt es eine Kante von A_i zu einem Knoten A_k mit $k < i$ und keine Kante von A_{i+r} nach A_k, dann mache mit dem nächsten Test in 4a weiter, sonst führe den Test aus. Gilt $A_{i+r} \subseteq A_i$, dann rufe zuerst UPDATE mit $A_{i+r} \subseteq A_i$ auf und gehe dann zu 4a, sonst gehe direkt zu 4a.
   (c) Solange die Liste der tests für A_i nicht leer ist, gehe zu 4a und mache mit dem nächsten Test $\underline{A_{i+r+1}}$ weiter.
5. Fahre bei 4 mit A_{i+1} fort.

Ende

Bevor der Algorithmus UPDATE näher beschrieben wird, zuerst ein paar Bemerkungen. Die Eingabe des gerade vorgestellten Algorithmus besteht genau aus den Attributen, die die Ausgabe des Algorithmus INTERVALLE bilden. Weiterhin wird die in Schritt (3) beschriebene Listenstruktur, sobald die ersten IAen gefunden wurden, sehr schnell durch die Aufrufe von UPDATE unvollständig. Dies soll heißen, daß z. B. in manchen „Paaren“ von Tests der eine oder der andere fehlt, oder daß Paare ganz fehlen. Dies muß in den

Schritten 4a und 4b noch entsprechend berücksichtigt werden. Weiterhin fehlt bewußt die Angabe der Ausgabe des Algorithmus. In der implementierten Version werden alle Kanten des Graphen ausgegeben. Diese repräsentieren nach Beendigung des Algorithmus alle IAen. Durch eine kleine Änderung kann aber auch die transitive Hülle ausgegeben werden, da sie bei der Erkennung und Ausnutzung der Transitivität als Nebenprodukt anfällt.

Vergleicht man noch einmal die im vorherigen Abschnitt angegeben SQL–Anweisungen zur Bestimmung der unären Inklusionsabhängigkeiten mit den Schritten 4a und 4b, dann wird klar, daß es sinnvoll ist, zuerst nachzuschauen, ob beide Tests $\underline{A_i}$ und $\overline{A_i}$ oder nur noch einer vorkommt. Dies verhindert, daß die erste SQL–Anfrage zweimal gestellt wird. Auch diese Bedingung wurde der Übersichtlichkeit wegen aus dem Algorithmus entfernt.

Algorithmus: UPDATE

Eingabe: Eine gefundene IA $A_i \subseteq A_j$

1. Trage die Kante $A_i \to A_j$ in den Graph ein.
2. (a) $i < j$
   i. Finde alle Knoten A_k mit $k > i$, von denen es einen Weg zu A_i gibt.
   ii. Finde alle Knoten A_l mit $l > i$, die von A_j aus erreichbar sind.
   iii. Streiche alle Tests $\underline{A_l}, l > j$ in der Liste A_i.
   iv. Streiche alle Tests $\underline{A_l}, k < l$ in den Listen A_k.
   v. Streiche alle Tests $\overline{A_k}, k > l$ in den Listen A_l.

   (b) $i > j$
   i. Finde alle Knoten A_k mit $k > j$, von denen es einen Weg zu A_i gibt.
   ii. Finde alle Knoten A_l mit $l > j$, die von A_j aus erreichbar sind.
   iii. Streiche alle Tests $\overline{A_k}, k > i$ in der Liste A_j.
   iv. Streiche alle Tests $\underline{A_l}, k < l$ in den Listen A_k.
   v. Streiche alle Tests $\overline{A_k}, k > l$ in den Listen A_l.

Ende

Die Algorithmen INCLUSION DEPENDENCIES und UPDATE garantieren, daß jede IA, die sich aufgrund der Transitivität ergibt, entdeckt wird und nicht mehr an der Datenbank überprüft wird. Weiterhin werden auch die Fälle gefunden, in denen die IA $A_i \subseteq A_j$ ausgeschlossen werden kann, da sie wegen der Art des Durchlaufs durch alle Kombinationen nicht existieren kann. Jedoch spielt die Reihenfolge, in der die Attribute zum Testen der IAen durchlaufen werden, eine Rolle. Dies verdeutlicht das folgende Beispiel.

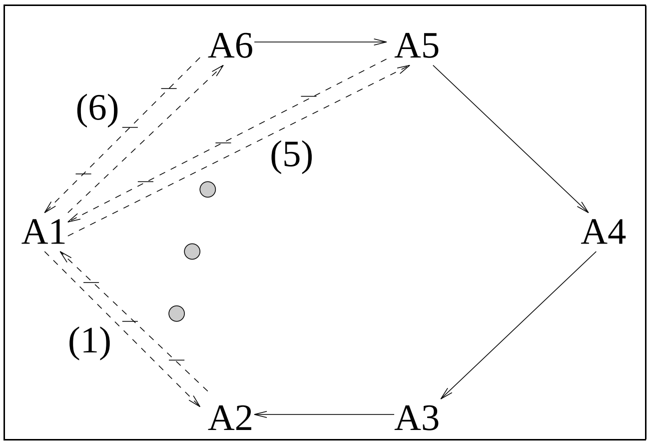

Abbildung 4.1: Skizze zum Beispiel 4.3

Beispiel 4.3 *Gesucht sind alle IAen zwischen A und $A_1, \ldots, A_n$. Angenommen, es existiert eine „transitive Kette" $A_n \subseteq A_{n-1} \subseteq \ldots A_1$ aufgrund der gefundenen Fremdschlüsseleinträge in der Systemtabelle. Damit sind diese Kanten nach dem Schritt 2 im Algorithmus* Inclusion Dependencies *im Graph eingetragen. Wenn nun jeweils gilt, daß $A \subseteq A_1, A \subseteq A_2, \ldots, A \subseteq A_n$ ist und gleichzeitig $A_1 \not\subseteq A, A_2 \not\subseteq A, \ldots A_n \not\subseteq A$ ist, dann werden hierfür $2 * n$ mal die SQL–Anfragen an die Datenbank gestellt. Wären die Attribute aber genau umgekehrt numeriert, so würden nur zweimal die SQL–Anfragen gestellt. Dies ergibt sich aufgrund der Transitivität und der Tatsache, daß $A_{n-1} \subseteq A, \ldots, A_1 \subseteq A$ nicht gelten kann, vgl. hierzu auch Lemma 4.3.* $\diamond$

Beispiel 4.4 *In Abbildung 4.1 wird das letzte Beispiel noch einmal aufgegriffen. Wird mit (6) und dem Test $\underline{A_6}$ angefangen, dann werden folgende Tests gestrichen: $\underline{A_5}, \ldots, \underline{A_2}$. Nach dem Test $\overline{A_6}$ können dann die restlichen Tests $\overline{A_5}, \ldots, \overline{A_2}$ entfernt werden.*

Um gegen solche Fälle gewappnet zu sein, könnte explizit nach Ketten mit einer bestimmten Mindestlänge gesucht werden. Der Algorithmus sähe dann so aus, daß zuerst der Graph aufgebaut wird. Dann wird nach diesen Ketten gesucht, und je nach Ergebnis werden dann die Attribute durchnumeriert. Der Rest bliebe gleich. Aus Gründen einer einfacheren Implementierung wird jedoch darauf verzichtet.

4.2.4 Analyse des Algorithmus

In diesem Abschnitt werden die beiden Algorithmen Inclusion Dependencies und Update näher analysiert. Dazu werden einige Eigenschaften gezeigt und jeweils ihre Laufzeiten angegeben. Die jeweils erste Anweisung in den Schritten 4a und 4b im Algorithmus INCLUSION DEPENDENCIES beschreibt eine Invariante, die während der Ausführung des Algorithmus aufrechterhalten wird. Das folgende Lemma zeigt die Korrektheit dieser Anweisungen.

Lemma 4.3 *1. Wenn eine gerichtete Kante vom Knoten A_{i+r} zu dem Knoten A_k existiert und keine solche Kante von A_i nach A_k mit $k < i$, dann kann es keine Kante von A_i nach A_{i+r} geben.*

2. Wenn eine gerichtete Kante vom Knoten A_i zu dem Knoten A_k exitiert und keine solche Kante von A_{i+r} nach A_k mit $k < i$, dann kann es keine Kante von A_{i+r} nach A_i geben.

Beweis

1. Angenommen die Kante von A_i nach A_{i+r} existiert. (Für den Algorithmus bedeutet dies, daß die entsprechende IA $A_i \subseteq A_{i+r}$ gilt.) Dann folgt aufgrund der Transitivität: $A_i \subseteq A_{i+r} \subseteq A_k \Rightarrow A_i \subseteq A_k$. Da jedoch $k < i$ ist, wurde die IA $A_i \subseteq A_k$ als Test $\overline{A_i}$ beim Durchlauf der Liste für A_k schon getestet. Und da es diese Kante A_i nach A_k nicht gibt, hat die IA $A_i \subseteq A_k$ nicht gegolten. Dies führt zum Widerspruch.
2. Analog zum ersten Fall.

□

Die Korrekheit des Algorithmus INCLUSION DEPENDENCIES beruht im wesentlichen auf diesem Lemma. Alle anderen Schritte sorgen nur für einen geordneten Durchlauf durch alle möglichen Tests und sind trivial. Der Algorithmus UPDATE ist für die Erkennung der Transitivität zuständig. Die einzelnen Fallunterscheidungen garantieren, daß nur versucht wird, in den Listen Tests zu streichen, in denen sie auch vorkommen können. Diese Schritte sind einfach nachzuvollziehen und werden nicht extra bewiesen.

Die Analyse der Laufzeit des Algorithmus INCLUSION DEPENDENCIES liefert folgende Ergebnisse: die Schritte (1) und (2) benötigen bei n Attributen jeweils eine lineare Laufzeit von $\mathcal{O}(n)$. In Schritt (3) müssen von jedem Knoten ausgehend alle Nachfolger gesucht werden. Diese werden benötigt, um die möglichen Tests zu streichen, die wegen der Transitivität der Inklusionsabhängigkeiten überflüssig sind. Da in der Systemtabelle jedes Attribut aber nur bezüglich eines anderen Attributs als Fremdschlüssel eingetragen werden kann, gibt es nur $\mathcal{O}(n)$ Kanten. Eine Suche per Breitensuche oder Tiefensuche benötigt daher $\mathcal{O}(n)$. Der Aufbau der Listen mit den auszuführenden Tests benötigt $\mathcal{O}(n^2)$. Dies führt zu einer Laufzeit für den dritten Schritt von $\mathcal{O}(n^2) + n * \mathcal{O}(n) = \mathcal{O}(n^2)$.

Die Schritte (4) und (5) arbeiten die Liste mit den Tests ab. Die erste Anweisung in 4a und 4b benötigt jeweils nur eine konstante Zeit. Jede der an die Datenbank gestellten Anfragen benötigt wiederum $\mathcal{O}(m^2)$ bei maximal m Tupeln. Dies führt zu einer Gesamtlaufzeit für den (4) und (5) Schritt von $\mathcal{O}(n^2 * (\text{UPDATE} + \mathcal{O}(m^2))$

Lemma 4.4 *Der Algorithmus* UPDATE *hat eine Laufzeit von* $\mathcal{O}(n^3)$.

Beweis Das Einfügen einer Kante in Schritt (1) benötigt jeweils konstante Zeit. Im zweiten Schritt wird immer nur (a) oder (b) ausgeführt. Die in i. und ii. ausgeführte Suche benötigt jeweils $\mathcal{O}(n+e)$. Bei n Attributen können maximal $\mathcal{O}(n)$ Knoten A_k bzw. A_l gefunden werden. Weiterhin gibt es $\mathcal{O}(n)$ Listen A_k bzw. A_l. Das Löschen eines Elementes in einer Liste mit $\mathcal{O}(n)$ Elementen benötigt wiederum $\mathcal{O}(n)$. Dies macht zusammen für die Punkte iii. bis v. $\mathcal{O}(n^3)$. Daraus ergibt sich folgende Gesamtlaufzeit: $2 * \mathcal{O}(n+e) + \mathcal{O}(n^3) = \mathcal{O}(n^3)$. □

Die Laufzeit läßt sich auf $\mathcal{O}(n^2)$ senken, wenn andere Datenstrukturen zum Speichern der notwendigen Tests verwendet werden. Dies können Arrays sein, die „indizierte Zugriffe“ in $\mathcal{O}(1)$ ermöglichen. Diese werden jedoch von Prolog nicht bereitgestellt.

Theorem 4.1 *Die Gesamtlaufzeit des Algorithmus* INCLUSION DEPENDENCIES *beträgt* $\mathcal{O}(n^5 + n^2 * m^2)$.

Vergleicht man das Ergebnis von $\mathcal{O}(n^5 + n^2 * m^2)$ mit $\mathcal{O}(n^2 * m^2)$ für den naiven Algorithmus, so läßt sich folgendes feststellen. Abgesehen von dem im Beispiel 4.3 gezeigten Problem, werden alle transitiven Beziehungen zwischen IAen entdeckt und die entsprechenden Datanbankzugriffe eingespart. Weiterhin werden auch die Fälle vorher entdeckt, in denen keine unären Inklusionsabhängigkeiten aufgrund des Lemmas 4.3 existieren können. Es werden also bei einer fest gewählten Reihenfolge des Durchlaufs durch alle Kombinationen nur soviel Datenbankanfragen gestellt, wie gerade nötig sind, um alle IAen zu finden.

Angenommen es existieren in einer Datenbank keine IAen. Dann ist der naive Algorithmus aufgrund seiner Einfachheit schneller, da beide Algorithmen alle n^2 Kombinationen an der Datenbank überprüfen müssen. Sobald nur einige wenige IAen existieren, ändert sich diese Situation. Ab wieviel gefundenen IAen der Algorithmus INCLUSION DEPENDENCIES nun wirklich schneller ist, hängt im wesentlichen von der Anzahl der Tupel in der Datenbank ab.

Bei einem sehr großen m im Verhältnis zu n, z. B. 10^6 zu 30, kann eine einzige Anfrage, gemessen in CPU–Zeit, länger dauern, als die Gesamtlaufzeit des Algorithmus INCLUSION DEPENDENCIES ohne die SQL–Anfragen. Daher kann sich der Aufwand von $\mathcal{O}(n^5)$ dann schon bei einer einzigen aufgrund der Transitivität eingesparten Anfrage, ob $A_i \subseteq A_j$ gilt, lohnen. Dies verdeutlicht auch nochmal, daß es in der Praxis uninteressant ist, welche genaue, polynomielle Laufzeit der Algorithmus zur Erkennung der Transitivität hat.

4.3 Funktionale Abhängigkeiten

Im Gegensatz zu den Inklusionsabhängigkeiten des letzten Abschnittes werden funktionale Abhängigkeiten für jedes Relationenschema des Datenbankschemas getrennt berechnet. Jedoch wird nicht mehr zwischen den Typen der Attribute unterschieden. Außer den Attributen der beiden Typen VARCHAR und NUMBER, werden hier noch die Attribute vom Typ DATE zur Menge der betrachteten Attribute hinzugenommen. Die Ausgabe des Algorithmus besteht in der Menge F der FAen der Form $A_1A_2 \ldots A_n \rightarrow A$, wobei die linke Seite minimal und die rechte Seite einstellig ist. Da, wie gezeigt werden wird, der Algorithmus vollständig und korrekt ist, wird sichergestellt, daß in F^* sämtliche in der Relation gültigen FAen enthalten sind.

4.3.1 Komplexität der Suchprobleme

In diesem Abschnitt wird der Suchraum analysiert, der von den minimalen funktionalen Abhängigkeiten aufgespannt wird. Dazu werden zwei verschiedene Fragestellungen erörtert. Zum einen „Wieviele minimale funktionale Abhängigkeiten gibt es?“, und zum anderen „Wie sieht die Struktur des Suchraumes aus?“.

Wie weiter oben gezeigt, vgl. Definition 2.10, gilt für die minimalen funktionalen Abhängigkeiten, die in dieser Arbeit betrachtet werden, daß die rechte Seite einstellig ist und daß das Attribut der rechten Seite auf der linken Seite nicht vorkommt. Betrachtet man nun eine Relation mit den vier Attributen A, B, C und D, so sieht der komplette Suchraum wie in Abbildung 4.2 aus. Dabei sind zur Vereinfachung der Darstellung die rechten Seiten der FAen entfernt worden. A bzw. AB usw. in dem linken Teilsuchraum steht für die funktionale Abhängigkeit $A \rightarrow D$ bzw. $AB \rightarrow D$. In den drei anderen Teilsuchräumen steht auch jeweils das „fehlende“ der vier Attribute auf der rechten Seite. Damit ist klar, daß gleiche Attributkombinationen wie z. B. AC für unterschiedliche FAen stehen und somit nicht zusammengefaßt werden. Bei dieser Art der Darstellung kann die Kombination $ABCD$ nicht vorkommen, da sie eine FA $ABCD \rightarrow X$ repräsentiert, die nicht minimal im Sinne der Definition 2.10 ist, weil sie trivialerweise gilt, vgl. Definition 2.4 und Lemma 2.1.

In der Abbildung sind die Verbindungen z. B. zwischen A, AB und AC wie folgt zu interpretieren. Wenn die FA $A \rightarrow D$, gilt, dann gelten auch die FAen $AB \rightarrow D$ und $AC \rightarrow D$. Oder wenn $AD \rightarrow C$ nicht gilt, dann können auch $A \rightarrow C$ und $D \rightarrow C$ nicht gelten. Da der in dieser Arbeit verwendete Minimalitätsbegriff sich nicht auf eine minimale Menge funktionaler Abhängigkeiten bezieht, vgl. Definition 2.12, wurden „Querverbindungen“ weggelassen, die aufgrund der Transitivität oder Pseudotransitivität bestehen.

Bemerkung 4.1 *1. Jeder einzelne Teilsuchraum aus der Abbildung 4.2 bildet einen Halbverband. Jeder dieser Verbände läßt sich zu einem*

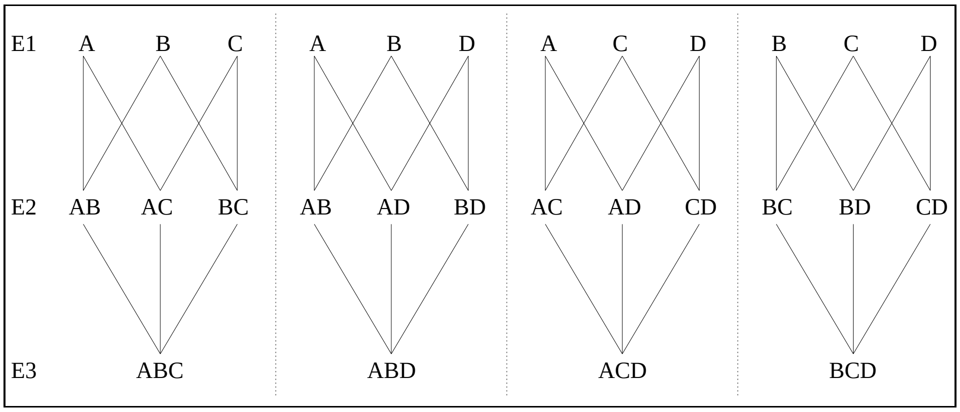

Abbildung 4.2: So sieht der vollständige Suchraum für 4 Attribute aus.

Booleschen Verband ergänzen, wenn auf der Ebene 0 sogenannte „nicht standard“ FAen eingeführt werden. Hierbei handelt es sich um FAen der Form $\emptyset \rightarrow A$. Diese sind erfüllt, wenn für das Attribut A nur identische Werte vorkommen, siehe hierzu [Kanellakis et al., 1983].

2. *Da bei der Verwendung des Begriffs „Teilsuchraum“ immer einer dieser Halbverbände in seiner Gesamtheit gemeint ist, wird im folgenden häufiger der Begriff „Halbverband“ verwendet.*

Lemma 4.5 *In einem vollständigen Suchraum für n Attribute existieren*

$$\sum_{i=1}^{n-1} \binom{n}{i} * (n-i) = n * (2^{n-1} - 1)$$

unterschiedliche nicht triviale funktionale Abhängigkeiten mit einer einstelligen rechten Seite.

Beweis Schichtet man die FAen nach der Anzahl der Attribute auf der linken Seite in Ebenen, siehe Abbildung 4.2, so gibt es bei n Attributen $n-1$ Ebenen. Die Ebenen werden dabei wie in der Abbildung mit E_1 bis E_{n-1} durchnumeriert. Pro Ebene gibt es $\binom{n}{i}$ Möglichkeiten, eine i–elementige Teilmenge zu bilden. Es verbleiben dann noch bei einstelligen Konklusionen $n-i$ Attribute für die rechte Seite der FA, wobei kein Attribut doppelt vorkommt, d. h. links und rechts steht. Nun muß noch über die Anzahl der Ebenen summiert werden. Damit ergibt sich die Summe auf der linken Seite der Gleichung. Der Beweis der Gleichung an sich befindet sich im Anhang A. □

Betrachtet man die Verteilung der FAen „über die Ebenen“, so erkennt man, daß auf den mittleren Ebenen die meisten Elemente vorkommen. Abbildung 4.3 zeigt dies für 20 Attribute. Diese Verteilung ist symmetrisch, (aufgrund des Symmetriesatzes für Binomialkoeffizienten: $\binom{n}{k} = \binom{n}{n-k}$), jedoch wegen

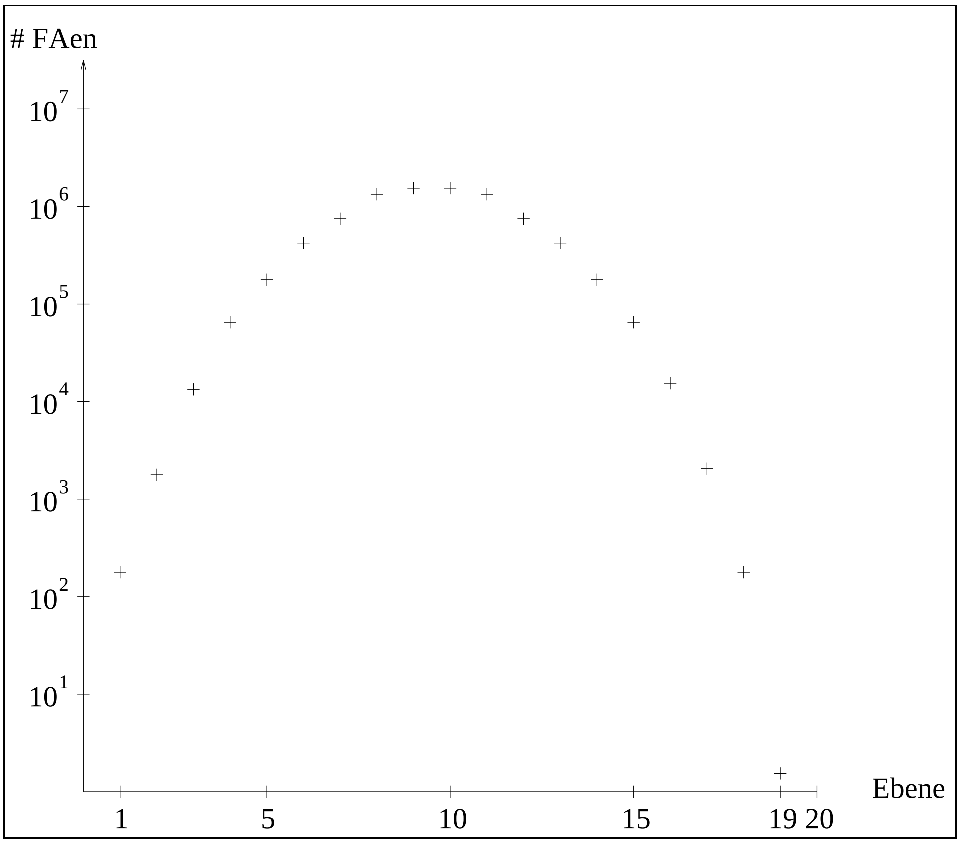

Abbildung 4.3: Verteilung der Anzahl der FAen über die Ebenen

des Faktors $(n-k)$ in der Summenformel leicht verschoben. Die Wachstumsrate zu der mittleren Ebene hin bzw. die Zerfallsrate nach dieser Ebene ist exponentiell[3].

Das nächste Lemma gibt eine obere Schranke für die Anzahl minimaler funktionaler Abhängigkeiten an. Auf der einen Seite wird der Binomialkoeffizient $\binom{n}{k} * (n-k)$ für $k = \left\lfloor \frac{n-1}{2} \right\rfloor$ maximal. Aber erst mit der Forderung nach minimalen FAen, und den damit verbundenen Eigenschaften, ergibt sich das Lemma. Der Beweis zeigt gleichzeitig, daß diese Schranke nur erreicht wird, wenn alle minimalen FAen auf der mittleren Ebene liegen, und daß es nicht mehr FAen geben kann, wenn diese anders auf die Ebenen aufgeteilt werden.

Lemma 4.6 *In einem vollständigen Suchraum für n Attribute existieren maximal*

$$\binom{n}{k} * (n-k) \quad mit \quad k = \left\lfloor \frac{n-1}{2} \right\rfloor$$

minimale funktionale Abhängigkeiten.

[3] Für Binomialkoeffizienten existiert eine äquivalente Schreibweise mit Fakultäten. Diese werden mit der Stirling'schen Formel approximiert, aus der sich die Behauptung ablesen läßt. Diese Behauptung folgt nicht aus der rechten Seite der Gleichung in Lemma 4.5.

Beweis Wir betrachten zunächst einen Halbverband bei einem Suchraum mit n Attributen.

1. Sei n ungerade. Dann gibt es auf der Ebene $i = \frac{n-1}{2}$ die meisten FAen, genau $\binom{n-1}{i}$. Seien diese alle minimal.
   (a) Angenommen es gibt auf einer Ebene k mit $k < i$ eine weitere minimale FA der Form $A_{j_1} \ldots A_{j_k} \to B$. Dann sind auf Ebene i mindestens zwei FAen nicht minimal, denn diese enthalten auf der linken Seite auch die Attribute $A_{j_1} \ldots A_{j_k}$. Folglich würde sich die Anzahl insgesamt vermindern.
   (b) Sei nun $k > i$, und es gebe eine weitere minimale FA der Form $A_{j_1} \ldots A_{j_k} \to B$. Dann können auf der Ebene i mindestens zwei FAen nicht minimal sein. Denn diese bestehen aus Attributen $A_{l_1} \ldots A_{l_i}$ auf der linken Seite, die alle Element aus der Menge $\{A_{j_1}, \ldots, A_{j_k}\}$ sind.

   Beide Fälle können entsprechend auf mehr als eine FA erweitert werden.

2. Sei n nun gerade. Dann gibt es auf den Ebenen $i = \frac{n}{2} - 1$ und $j = \frac{n}{2}$ sowohl gleichviel als auch die meisten FAen. Ohne Beschränkung der Allgemeinheit seien alle minimalen FAen auf Ebene $i = \frac{n}{2} - 1$. Dort gibt es $\binom{n-1}{i}$ FAen und es gilt die gleiche Argumentation wie in 1.

Da alle minimalen FAen in einem Halbverband unabhängig von den FAen in den anderen Halbverbänden sind, gilt die Argumentation entsprechend für den ganzen Suchraum mit n Halbverbänden. Dort können dann nicht mehr als $\binom{n}{k} * (n-k)$ FAen existieren. □

Zusätzlich zum Lemma 4.6, das eine obere Schranke für die Anzahl der minimalen funktionalen Abhängigkeiten angibt, läßt sich auch eine untere Schranke angeben. Diese untere Schranke zeigt, daß es Relationen gibt, bei denen jeder Algorithmus zur Berechnung aller minimalen funktionalen Abhängigkeiten eine exponentielle Laufzeit bzgl. der Anzahl der Attribute haben muß. Das Theorem stammt aus [Mannila und Räihä, 1987].

Theorem 4.2 (Mannila, Räihä) *Für jede natürliche Zahl n existiert eine Instanz r einer Relation R mit n Attributen und $\mathcal{O}(n)$ Tupeln[4], so daß mindestens $\Omega(2^{\frac{n}{2}})$ minimale funktionale Abhängigkeiten existieren.*

Beweis Der Beweis findet sich in [Mannila und Räihä, 1992, S. 275]. Dort wird eine Relation konstruiert, die die Aussage des Theorems erfüllt. □

[4] Die benötigte Anzahl der Tupel hängt nur linear von der Anzahl der Attribute ab.

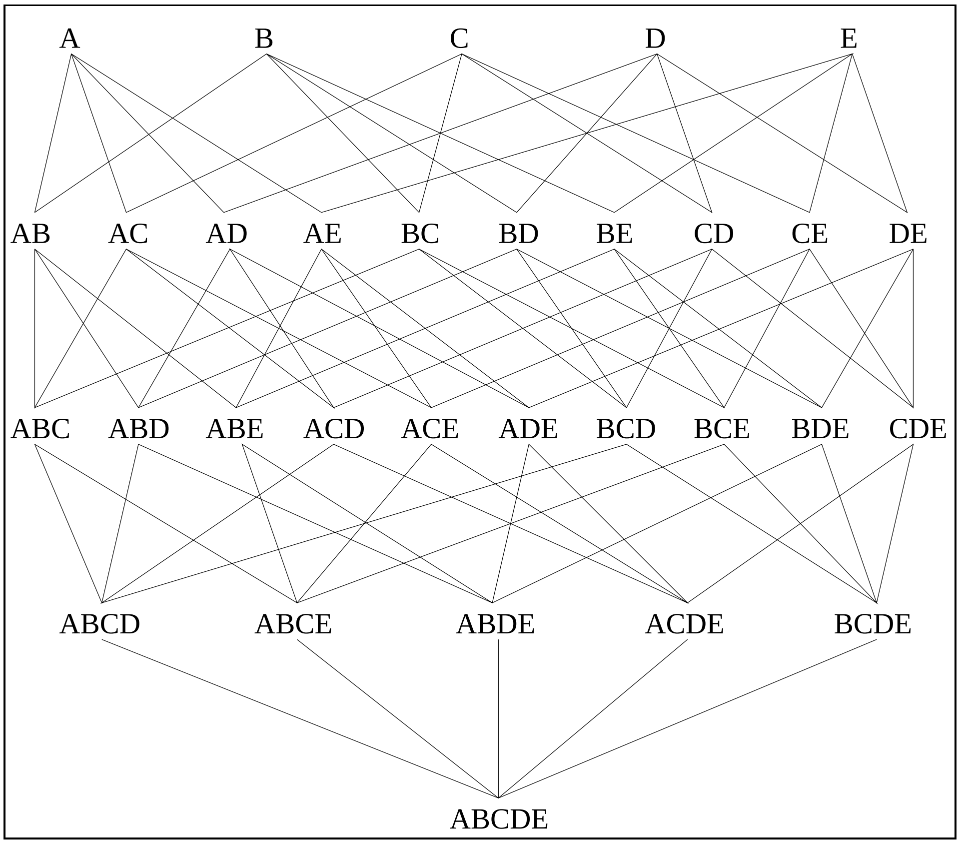

Abbildung 4.4: Ein Halbverband für 6 Attribute.

4.3.2 Suchstrategien

In Abbildung 4.4 wird ein Halbverband aus dem Suchraum für sechs Attribute gezeigt. A usw. stehen wieder für die funktionalen Abhängigkeiten $A \rightarrow F$ usw. Dabei ist jedes Element, z. B. ABC, nur eine Hypothese dafür, daß die funktionale Abhängigkeit $ABC \rightarrow F$ gilt. Auf der Ebene E_1 stehen die generellsten Hypothesen und die Ebene E_{n-1} wird in jedem Halbverband von der speziellsten Hypothese gebildet. Unter den Verfahren, die nun diesen Graphen oder „Hypothesenraum" vollständig nach den minimalen funktionalen Abhängigkeiten durchsuchen, kann zwischen zwei grundsätzlich verschiedenen Strategien unterschieden werden.

Da sind zum einen die „Top–Down–Suchverfahren" und zum anderen die „Bottom–Up–Suchverfahren". Da in dem vorliegenden Fall der Suchraum ein Halbverband ist, und somit eine Halbordnung existiert, sind nur diejenigen Suchverfahren von Interesse, die diese Ordnung ausnutzen können. Zuvor müssen jedoch zwei Operationen erklärt werden, die hierzu benötigt werden. Es handelt sich dabei um die Generalisierung und die Spezialisierung.

Eine Hypothese H_1 wird als „genereller" gegenüber einer Hypothese H_2 bezeichnet, wenn H_1 bezüglich der Ordnungsrelation in dem Hypothesenraum größer als H_2 ist. Somit ist H_2 kleiner als H_1 und wird daher als „spezieller" bezeichnet. Diese Begriffe werden wie folgt auf FAen übertragen. Eine FA $X \rightarrow A$ ist genereller als eine FA $Y \rightarrow B$, wenn $X \subseteq Y$ und $A = B$ gilt. Sie ist spezieller, wenn $Y \subseteq X$ und $A = B$ gilt. Da die Ordnungsrelation keine totale Ordnung erzeugt, gibt es unvergleichbare Hypothesen, hierzu gehören insbesondere alle Hypothesen mit $A \neq B$ auf der rechten Seite.

Mit dem Begriff Generalisierung wird die Operation bezeichnet, die ausgehend von einer Hypothese, generellere Hypothesen findet. Hypothesen $A_1 \ldots A_n \rightarrow B$ werden generalisiert, indem auf der linken Seite ein Attribut A_i entfernt wird. Die Spezialisierung ist die duale Operation, und somit werden bei der Spezialisierung Attribute auf der linken Seite hinzugefügt. Bei dieser Operationen wird vorausgesetzt, daß nur Attribute hinzugefügt werden, die nicht schon auf der linken Seite der FA vorkommen.

Beide Operationen erlauben damit eine Navigation im Hypothesenraum. Sie werden als „schrittweise Generalisierung“ bzw. „schrittweise Spezialisierung“ bezeichnet, wenn im Falle der Generalisierung folgendes gilt: ausgehend von einer Hypotheses H werden nur Hypothesen H_i gefunden, die genereller als H sind, und es existieren keine Hypothesen H_j, die genereller als H und spezieller als H_i sind. Für die Spezialisierung gilt analog der duale Fall. Für die Realisierung dieser Operationen sind zwei Alternativen denkbar. Ausgehend von einer Hypothese H werden entweder eine oder alle generelleren oder spezielleren Hypothesen gefunden.

Ein Top–Down–Suchverfahren ist nun dadurch gekennzeichnet, daß es mit den generellsten Hypothesen beginnt. Diese Hypothesen werden solange schrittweise spezialisiert, bis sie gelten. Dies sind genau die gesuchten minimalen funktionalen Abhängigkeiten. Dieses Verfahren endet spätestens bei den speziellsten Hypothesen.

In den Bottom–Up–Suchverfahren wird entsprechend die Dualität zwischen der Generalisierung und der Spezialisierung ausgenutzt, so daß sie genau umgekehrt arbeiten. Die Ausgabe bilden alle Hypothesen vor dem Generalisierungsschritt, der dazu führte, daß aus einer gültigen Hypothese eine ungültige wurde. Auch dies sind die gesuchten minimalen funktionalen Abhängigkeiten.

Top–Down– und Botom–Up–Suchverfahren, die auf die gerade beschriebene Art und Weise arbeiten, sind den Top–Down– und Bottom–Up–Lernverfahren sehr ähnlich, siehe zum Beispiel [Mitchell, 1982] oder [Morik, 1993]. Mitchell beschreibt noch eine weitere Möglichkeit, den Hypothesenraum zu durchsuchen, die Kombination der Generalisierung und der Spezialisierung im „Versionenraum“, oder Hypothesenraum, [Mitchell, 1982]. Hierauf wird in Abschnitt 6.3.2 als eine mögliche Erweiterung des im weiteren Verlauf vorgestellten Algorithmus noch einmal eingegangen.

Bei beiden Alternativen bezüglich der Suche im Hypothesenraum, der nun zweckmäßiger Weise wieder als Graph betrachtet wird, muß berücksichtigt werden, daß zwischen zwei Knoten, z. B. A und ABE, mehrere Wege existieren, die keinen Knoten gemeinsam haben. Für die beiden Alternativen folgt daraus, daß sie es vermeiden müssen, je nachdem von welcher der beiden

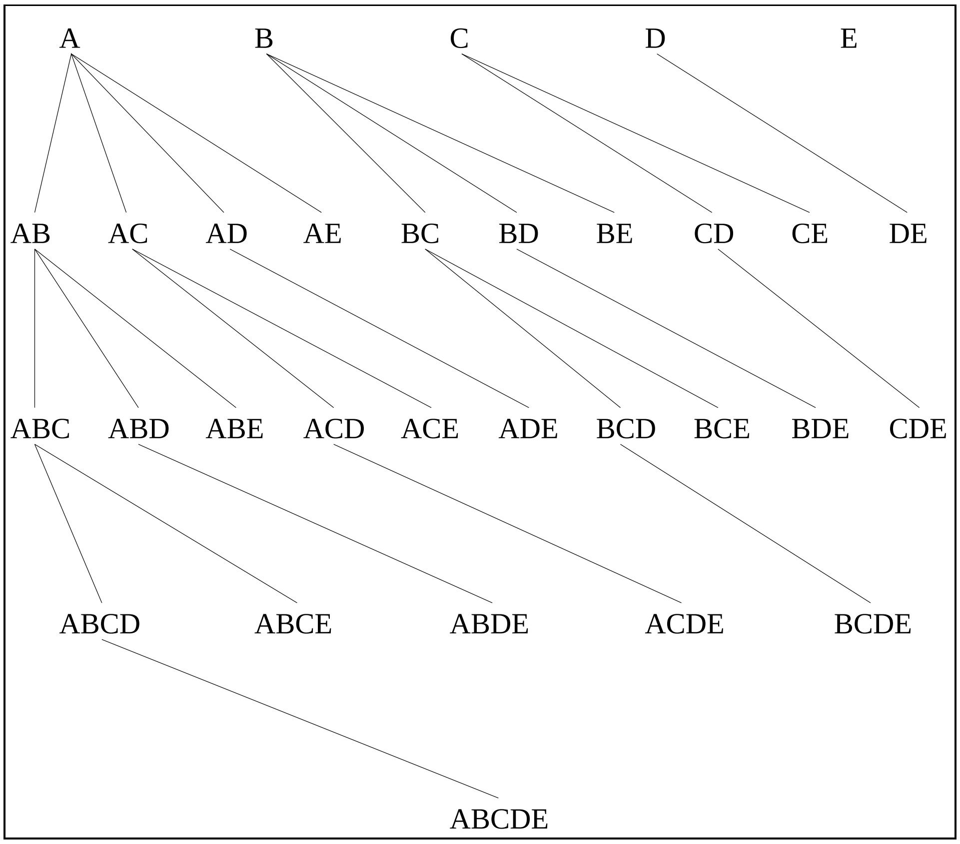

Abbildung 4.5: So sieht der auf Bäume reduzierte Halbverband aus Abbildung 4.4 aus.

korrespondierenden Hypothesen sie ausgehen, die jeweils andere mehrfach zu generieren und entsprechend zu testen.

Zur Lösung dieses Problems bietet sich folgende Vorgehensweise an. Zuerst wird eine der beiden Strategien ausgewählt. Dies sei die Top–Down–Suche. Anschließend wird der Hypothesenraum aus der Graphdarstellung in eine Baumdarstellung wie in Abbildung 4.5 transformiert.

Die Knoten in diesen Bäumen werden durch die Konkatenation der Attributnamen A_i zu „Ketten“ gebildet. In einer Kette $A_1 \dots A_n$ gilt, daß bzgl. der lexikographischen Ordnung jedes Attribut A_i größer als alle Attribute A_k mit $k < i$ und kleiner als alle Attribute A_l mit $l > i$ ist. Für jeden direkten Nachfolger eines Knoten gilt, daß genau ein Attributname an das Ende der Kette dieses Vorgängers angehängt wird. Dadurch ist das Problem der „doppelten Generierung von Hypothesen“ auf einfache Weise gelöst, und es gilt, (ohne Beweis):

Lemma 4.7 *Für alle Nachfolger $A_{i_1} \dots A_{i_k} A_{i_{k+1}} \dots A_{i_{k+l}}$, $l \leq n-k-1$, eines Knoten $A_{i_1} \dots A_{i_k}$ gilt bezüglich der lexikographischen Ordnung auf den Attributen $A_1, \dots, A_n$:*

1. *Sie sind größer als alle Knoten in den Bäumen mit den Wurzeln A_j und $j < i$ und kleiner als alle Knoten in den Bäumen mit den Wurzeln A_j und $j > i$,*

2. *Sie sind größer als alle Vorgänger auf dem Pfad von der Wurzel A_i dieses Baumes zu diesem Knoten $A_{i_1} \ldots A_{i_k}$,*

3. *Sie sind größer als alle Nachfolger der Knoten $A_{i_1} \ldots A_{i_{k-1}} A_j$ mit $j < k$ und kleiner als alle Nachfolger der Knoten $A_{i_1} \ldots A_{i_{k-1}} A_j$ mit $j > k$.*

Es existiert jedoch noch ein weiteres Problem, die „Generierung von überflüssigen Hypothesen". Angenommen, es liegt die folgende Situation vor: $CE \rightarrow F$ und $AC \not\rightarrow F$. Dann ist die Generierung der Hypothese $ACE \rightarrow F$ überflüssig und „falsch". Erstens ist ihre Gültigkeit bekannt, da sie eine Spezialisierung einer gültigen Hypothese ist. Und zweitens würde sie zu einer nicht minimalen funktionalen Abhängigkeit führen.

Bei der Suche in dem nun als Wald vorliegenden Hypothesenraum ist dieses weitere Problem zu berücksichtigen. Angenommen der Wald wird mit einer Tiefensuche durchlaufen, und es wird mit dem „links stehenden" Baum und in diesem wieder links begonnen. Dann wird die Zahl dieser überflüssigen Hypothesen am größten.

Darüberhinaus muß zusätzlich jede gefundene FA daraufhin überprüft werden, ob sie nicht eine Generalisierung einer bisher als minimal bekannten FA ist. Letztere muß daraufhin wieder verworfen werden, damit keine Inkonsistenzen entstehen. Dieses Problem tritt bei einer Breitensuche gar nicht auf. Das nächste Beispiel verdeutlicht nochmal diese beiden Aspekte.

Beispiel 4.5 *Sei der Suchraum wie in Abbildung 4.5 Es gelten folgende minimale FAen: $B \rightarrow F$ und $CD \rightarrow F$. Es wird mit dem Baum mit der Wurzel A begonnen. Wäre $CD \rightarrow F$ schon bekannt, dann könnten zwei Anfragen an die Datenbank eingespart werden, nämlich $ABCD \rightarrow F$ und $ACD \rightarrow F$.* [5] *Wird bei der Tiefensuche in dem Teilbaum für A links begonnen, so wird zuerst die „minimale" FA $ABCD \rightarrow F$ entdeckt. Später wird dann die FA $ACD \rightarrow F$ gefunden, die eine Generalisierung ist. Daraufhin muß $ABCD \rightarrow F$ als minimale FA verworfen werden. Dieses wiederholt sich für $CD \rightarrow F$. Ist zusätzlich schon $B \rightarrow F$ bekannt, dann ist weiterhin die Anfrage $AB \rightarrow F$ überflüssig.* ◇

Daher muß eine Tiefensuche mit dem am weitesten rechts stehenden Baum und in diesem Baum wiederum mit dem rechten Blatt beginnen. Wird nun jede generierte Hypothese zuerst mit allen gefundenen minimalen funktionalen Abhängigkeiten verglichen, so ist sichergestellt, daß keine überflüssige Hypothese an der Datenbank getestet wird. Außerdem werden nicht wie in

[5] Spezialisierungen von gültigen FAen werden weiterhin nicht meht betrachtet, z. B. $ABCDE \rightarrow F$.

dem Beispiel 4.5 je nach Durchlauf der Bäume auch noch Nachfolger solcher Hypothesen generiert und getestet.

Eine andere Alternative für den Durchlauf dieses Waldes besteht in einer Breitensuche, entweder Baum für Baum oder über alle Bäume gleichzeitig, also Ebenenweise. Bei der ersten Möglichkeit muß wiederum wie bei der Tiefensuche rechts begonnen werden, um die Anzahl der überflüssigen Hypothesen zu minimieren. Die zweite Möglichkeit wird in dem folgenden Algorithmus verwendet und dort ausführlich beschrieben.

Sowohl bei der Tiefensuche als auch bei der Breitensuche wurden bis jetzt nur die Bäume betrachtet, die aus einem Halbverband entstanden sind. Es wurde dabei davon ausgegangen, daß ein Teilsuchraum vollständig durchsucht wird bevor in dem nächsten weitergesucht wird. Dies muß so aber nicht sein. Es kann beliebig zwischen den einzelnen Teilsuchräumen „gesprungen“ werden. Das kann sowohl nach Beendigung der Suche in einem Baum oder Wald als auch während der Suche geschehen. Letzteres soll heißen, daß die Suche zu einem bestimmten Zeitpunkt „eingefroren“ wird und später an der „selben Stelle“ fortgesetzt wird. Es muß dann nur sichergestellt sein, daß die zwischendurch erzielten Ergebnisse entsprechend berücksichtigt werden.

Ein letztes Problem bleibt aber sowohl bei der Tiefen– als auch bei der Breitensuche bestehen. Für überflüssige Hypothesen gilt nach wie vor, daß sie stets generiert werden bevor entschieden werden kann, daß sie überflüssig sind, weil sie eine Spezialisierung einer mehrstelligen minimalen funktionalen Abhängigkeit sind. Damit werden sie dann nicht mehr an der Datenbank überprüft. Für entsprechende Spezialisierungen einstelliger funktionaler Abhängigkeiten hingegen wird dieses Problem in dem Algorithmus Functional Dependencies gelöst.

4.3.3 Vorarbeiten

Der Algorithmus Functional Dependencies setzt voraus, daß sämtliche Attribute einer Relation auf die vier folgenden disjunkten Klassen aufgeteilt worden sind.

1. **Sicherer Schlüssel (SS)**: Das Attribut A ist in der Systemtabelle als Primärschlüssel markiert.

2. **Schlüsselkandidat (SK)**: Für das Attribut A kommen nur verschiedene und keine NULL–Werte vor.

3. **Kein–Schlüssel–nicht–NULL (KSNN)**: Für das Attribut A kommen doppelte Werte, jedoch keine NULL–Werte vor.

4. **Kein–Schlüssel (KS)**: Es gibt NULL–Werte für A.

Die Klasse SS enthält nur dann ein Element, wenn ein Attribut als einstelliger Primärschlüssel ausgezeichnet ist. Sie ist daher einelementig oder leer, weil pro Relation nur ein Primärschlüssel existieren darf. Ist hingegen ein Primärschlüssel definiert, der aus den Attributen $A_1 \dots A_n$ besteht, so wird dieser als „möglicherweise minimal“ zusätzlich getrennt gespeichert.

Zur Einteilung der Attribute in die restlichen drei Klassen können die Informationen aus der Systemtabelle über einstellige eindeutige Indizes benutzt werden. Ist für A ein solcher Index definiert, so ist A ein Kandidat für SK. Jedoch ist es zumindest bei Oracle 7 erlaubt, daß A noch NULL–Werte enthält. Dieses muß noch überprüft werden und A kommt dann nach SK oder KS.

Wenn hingegen mehrstellige eindeutige Indizes definiert sind, so gibt es keinen entsprechenden Hinweis. Die beiden Eigenschaften „NULL–Werte“ und „doppelte Werte“ müssen für jedes beteiligte Attribut überprüft werden. Kommen aber alle beteiligten Attribute $A_1 \dots A_n$ eines mehrstelligen eindeutigen Index in die Klasse KSNN, so wird $A_1 \dots A_n$ wiederum als „möglicherweise minimal“ zusätzlich gespeichert.

Als nächstes wird der Begriff „möglicherweise minimal“ erlärt. Der Algorithmus zur Berechnung der funktionalen Abhängigkeiten wird nur auf eine und nicht auf alle Instanzen einer Datenbank angewendet. Daher ist es sehr wohl möglich, daß in dieser Instanz eine Auswahl $A_{i_1} \dots A_{i_r}$ von Attributen aus dem Primärschlüssel $A_1 \dots A_n$, $n \geq 2$, bereits ein Schlüssel oder zumindest eine Determinante für ein Attribut B ist. Dafür müssen im übrigen alle Attribute A_{i_1} bis A_{i_r} aus KSNN sein, denn sonst wäre $A_{i_1} \dots A_{i_r}$ nicht minimal. Daher wird $A_1 \dots A_n$ als nur „möglicherweise minimal“ bezeichnet, damit auch nach diesen FAen $A_{i_1} \dots A_{i_r} \rightarrow B$ gesucht wird.

Die Konsequenz hieraus ist, daß die Attribute $A_1 \dots A_n$ aus zusammengesetzten Primärschlüsseln $A_1 \dots A_n$ keine „Sonderbehandlung“ erfahren, sondern wie alle anderen Attribute behandelt werden. Der Algorithmus würde sonst minimale FAen wie z. B. $A_{i_1} \dots A_{i_r} \rightarrow B$ nicht finden. Auf der anderen Seite wird der Primärschlüssel $A_1 \dots A_n$ getrennt gespeichert, da sonst im Laufe des Algorithmus eine Spezialisierung von $A_{i_1} \dots A_{i_r}$, nämlich $A_1 \dots A_n$, getestet würde. Das heißt, es wird die Anfrage an die Datenbank gestellt, ob die FA $A_1 \dots A_n \rightarrow B$ gilt. Diese gilt natürlich trivialerweise. Bei zusammengesetzten eindeutigen Indizes ist die Situation ähnlich.

Es läßt sich also festhalten, daß die getrennte Speicherung von Primärschlüsseln und Indizes nur geschieht, um möglicherweise eine Anfrage an die DB zu sparen. Verglichen mit der Zeit, die eine überflüssige Anfrage an die Datenbank kostet, erscheint dieser zusätzliche Aufwand aber allemal gerechtfertigt. Eine Frage ist jedoch noch nicht beantwortet. Warum wird diese Einteilung der Attribute überhaupt gemacht? Die Gründe dafür werden im Abschnitt 4.3.4 nachgereicht.

SQL–Anfragen für die Klasseneinteilung

Die Attribute einer Relation werden mit den beiden folgenden Anfragen und den Informationen aus der Systemtabelle in die beschriebenen Klassen eingeteilt. Die erste Anfrage wird nur gestellt, wenn das Attribut nicht in der Systemtabelle als NOT NULL gekennzeichnet ist, siehe Abbildung 2.2.

1. SELECT DISTINCT (A_i)
   FROM R_i
   WHERE A_i IS NULL $=: e_1$

2. SELECT COUNT(DISTINCT A_i), COUNT (A_i)
   FROM R_i $=: e_2$

Das Ergebnis e_1 der ersten SQL–Anweisung ist gleich „NULL“, wenn NULL–Werte enthalten sind. Sonst ist e_1 gleich dem leeren Tupel, d. h. es wurden keine Tupel selektiert, und somit gibt es keine NULL–Werte.

Die zweite Anfrage liefert ein zweielementiges Tupel, bestehend aus zwei Zahlen. Sind diese beiden gleich, so gibt es keine doppelten Werte, im anderen Fall sind welche vorhanden. Dabei ist noch anzumerken, daß bei der zweiten Anfrage NULL–Werte weder bei COUNT DISTINCT noch bei COUNT mitgezählt werden.

4.3.4 Schnittkriterien

Folgende Gründe für die Einteilung der Attribute in die vier Klassen gibt es. Das eine Attribut, das in die Klasse SS kommt — pro Relation kann es höchstens einen einstelligen Primärschlüssel geben —, impliziert aufgrund der Schlüsseleigenschaft alle Attribute der Relation. Die Attribute der Klasse SK implizieren alle Attribute der Relation aufgrund der Definition dieser Klasse. Ein Attribut, das keine doppelten und keine NULL–Werte enthält, ist selbst ein Schlüssel für die Relation. Für die Attribute beider Klassen gilt, daß die Gültigkeit jeder möglichen FA, in der mindestens ein Attribut aus diesen beiden Klassen vorkommt, nicht mehr an der DB überprüft werden muß. Diese FAen gelten trivialerweise.

Attribute A_i der Klasse KS können weder alleine ein Attribut B bestimmen, noch können sie Bestandteil einer minimalen FA $A_1 \ldots A_n \rightarrow B$ sein. Folglich müssen sie als ein Attribut auf der linken Seite einer FA gar nicht berücksichtigt werden. Es bleiben daher als Attribute für linke Seiten zu suchender FAen alleine die Attribute aus der Klasse KSNN übrig. Als Attribute für die rechte Seite einer FA kommen natürlich alle Attribute aller vier Klassen in Frage.

Zur Vermeidung von „überflüssigen“ Anfragen an die Datenbank, ob $A_1 \ldots A_n \rightarrow B$ mit $n \geq 1$ gilt, oder nicht gelten kann, können die folgenden Kriterien[6] verwendet werden. Dabei bezeichne U die Menge aller Attribute einer Relation R. Die A_i bzw. B stehen entsprechend für einzelne Attribute dieser Relation.

[6] Diese werden Schnittkriterien genannt, da sie zur „Eingrenzung“ oder „Beschneidung“ des Hypothesenraumes dienen.

Theorem 4.3 (Schnittkriterien)

1. *Sei* $A \in SS \vee A \in SK$. *Dann gilt:* $A \rightarrow U$.
2. *Sei* $A \in KS$. *dann gilt:* $A \not\rightarrow U$.
3. *Sei* $A \in KSNN$ *und* $B \in SK \vee B \in SS$. *Dann gilt:* $A \not\rightarrow B$.
4. $A \subset B \Rightarrow A \not\rightarrow B$
5. *Aus* $A_1 \ldots A_n \rightarrow A$ *mit* $n \geq 1$ *und* $A \in SS \vee A \in SK$ *folgt:* $A_1 \ldots A_n \rightarrow U$.
6. *Sei* $A \in KS$ *und* $A_1 \ldots A_n \not\rightarrow B$. *Daraus folgt:* $A\,A_1 \ldots A_n \not\rightarrow B$

Beweis

1. Trivial.
2. $A \in KS$ enthält mindestens einen NULL–Wert. Damit kann A keine Attribute funktional bestimmen.
3. Da A und B aus der selben Relation stammen, haben sie gleichviel Tupel. A enthält mindestens zwei gleiche Werte w_1. B enthält nur verschiedene Werte v. Sei $R(\ldots, A, \ldots, B, \ldots)$ das entsprechende Datenbankschema. Damit gibt es zwei Tupel, die wie folgt aussehen: $(\ldots,\ w_1,\ \ldots,\ v_1,\ \ldots)$ und $(\ldots,\ w_1,\ \ldots,\ v_2,\ \ldots)$. Diese beiden Tupel sorgen dafür, daß $A \not\rightarrow B$ gilt.
4. Vergleiche Lemma 2.4.
5. Folgt aus der Transitivität der FAen und (1).
6. $A \in KS$ enthält mindestens einen NULL–Wert. Wegen $A_1 \ldots A_n \not\rightarrow B$ und (2) gilt $A\,A_1 \ldots A_n \not\rightarrow B$.

□

Zu dem obigem Theorem sind zwei Anmerkungen zu machen. Der Punkt 3 gilt natürlich nur für einstellige FAen dieses Typs. Und für Punkt 4 gilt, daß diese Implikation eine echte Teilmengenbeziehung, bzw. unäre Inklusionsabhängigkeit, zwischen A und B voraussetzt.

Die in dem Theorem 4.3 aufgeführten Schnittkriterien werden nun dazu genutzt den Hypothesenraum zu beschneiden. Es werden sowohl alle Hypothesen entfernt, die trivialerweise gelten, als auch alle Hypothesen, die aufgrund der Kriterien nicht gelten können. Erstere sind die Hypothesen, die wegen (1) gelten, letztere diejenigen, die wegen (2), (3) und (4) nicht gelten. Die beiden folgenden Theoreme beschreiben die Form, die alle in dem Suchraum verbliebenen Hypothesen nach der Anwendung von (1) bis (4) erfüllen.

Theorem 4.4 *Für die generellsten Hypothesen* $A \to B$ *gilt:*
$A \in KSNN$ *und* ($B \in KSNN$ *oder* $B \in KS$).

Beweis Anwendung der Kriterien (1) bis (4) aus Theorem 4.3. □

Theorem 4.5 *Für alle Hypothesen* $A_1 \ldots A_n \to B$ *mit* $n \geq 2$ *gilt:*
$A_i \in KSNN$ *und* ($B \in SS$ *oder* $B \in SK$ *oder* $B \in KSNN$ *oder* $B \in KS$).

Beweis Anwendung der Kriterien (1),(2) und (6) aus Theorem 4.3. □

Korollar 4.1 *Es gibt Teilverbände, in denen die Suche erst auf der zweiten Ebene beginnt.*

Das folgende Beispiel verdeutlicht das Theorem 4.5 und das Korollar.

Beispiel 4.6 *Für eine Datenbank mit fünf Attributen sind die Attribute wie folgt auf die Klassen verteilt.* $A \in SS, E \in SK, B, C, D \in KSNN$. *Dann besteht der Halbverband für* $A_1 \ldots A_n \to E$ *aus folgenden Hypothesen:* $BC \to E$,$BD \to E$,$CD \to E$ *und* $BCD \to E$. ◇

4.3.5 Eine SQL–Anweisung zur Bestimmung funktionaler Abhängigkeiten

In diesem Abschnitt werden die SQL–Anweisungen vorgestellt, die es ermöglichen, eine Entscheidung darüber zu treffen, ob eine Hypothese bei einer gegebenen Datenbank gilt und damit die funktionale Abhängigkeit $A_1 \ldots A_n \to B$ existiert. Um diese Entscheidung treffen zu können, bietet sich die folgende Idee an.

Projiziere die Relation R auf die Attribute $A_1, \ldots, A_n$ und B, die die Hypothese $A_1 \ldots A_n \to B$ bilden. Gruppiere diesen Ausschnitt der Relation nach den Attributen $A_1 \ldots A_n$. Zähle die Anzahl verschiedener Werte für B in den Tupeln jeder Gruppe und summiere diese Zahlen über alle Gruppen. Vergleiche das Endergebnis mit der Anzahl der Gruppen. Dabei ist es für die letzte Anweisung ausreichend, die verschiedenen Werte für ein einzelnes Attribut A_i in den Tupeln jeder Gruppe zu zählen und wiederum über alle Gruppen zu summieren.

Bei der Gruppierung nach A_1 bis A_n werden eventuelle doppelte Tupel eliminiert. Weiterhin darf jede Gruppe nur genau ein Tupel enthalten, damit die funktionale Abhängigkeit gilt. Enthält nun eine Gruppe noch mehr als ein Tupel, so gibt es immer für B verschiedene Werte. Denn alle Werte für alle Attribute A_i sind aufgrund der Gruppierung in dieser Gruppe gleich. Da nur die Attribute A_i und B in der Projektion vorkommen, fallen die Tupel, die in A_1 bis A_n und B übereinstimmen und sich erst in einem Attribut C unterscheiden, „in einem Tupel zusammen".

Dies ist insofern wichtig, als B das Attribut ist, das die Konklusion der zu testenden funktionalen Abhängigkeit bildet. In dem folgenden Beispiel würde die erste Hypothese nicht gelten, wenn diese Projektion auf $A_1 \dots A_n$ und B nicht durchgeführt würde.

Die folgende SQL–Anweisung realisiert diese Idee. Dabei wird die Projektion weiter eingeschränkt auf A_1 und B, da nur diese beiden Attribute für die Summierung benötigt werden. Das Ergebnis dieser SQL–Anfrage ist ein zweielementiges Tupel, das aus zwei Zahlen für die beiden Summen besteht.

1. SELECT SUM (COUNT (DISTINCT A_1)),
   SUM (COUNT (DISTINCT B))
   FROM R
   GROUP BY $A_1, \dots, A_n$ $=: a, b$

2. $a = b \Rightarrow A \rightarrow B$

Beispiel 4.7 *Eine Relation R enthalte die Attribute A_1 bis A_3, B und C. Es sind die Hypothesen $A_1A_2 \rightarrow B$ und $A_2A_3 \rightarrow B$ zu verifizieren. Die erste Tabelle zeigt den Inhalt der Relation. Die beiden anderen Tabellen verdeutlichen die Idee, die hinter der SQL–Anweisung steckt.*

A_1	A_2	A_3	B	C
3	3	11	d	f
2	1	7	d	f
2	2	9	c	i
3	3	11	d	g
1	2	9	a	p
2	2	9	c	h

A_1	A_2	B
1	2	a
2	1	d
2	2	c
3	3	d

A_2	A_3	B
1	7	d
2	9	a
2	9	c
3	11	d

Für die erste Hypothese ist das Ergebnis der SQL–Anfrage $a = b = 4$. Daraus folgt: $A_1A_2 \rightarrow B$. Für die zweite Hypothese ist hingegen $a = 3$ und $b = 4$. Damit ergibt sich: $A_2A_3 \not\rightarrow B$. ◇

Laufzeit der SQL–Anfrage

Die Sortierung der beiden Ergebnistabellen des Beispiels illustriert nur eine Möglichkeit, die Gruppierung nach den Attributen $A_1, \dots, A_n$ vorzunehmen. Denn die Semantik der GROUP BY Anweisung fordert diese Sortierung nicht. Je nach DBMS sind auch andere Implementierungen denkbar. Angenommen, die Gruppierung nach den Attributen $A_1, \dots, A_n$ wird durch eine Sortierung erreicht. Dann gilt:

Lemma 4.8 *Eine „gleichzeitige Sortierung" nach n Attributen mit m Tupeln benötigt $\mathcal{O}(m * \log m)$ Zeit.*

Beweis Ein Sortierverfahren wie z. B. Heapsort benötigt auch im schlechtesten Fall nicht mehr als $\mathcal{O}(m * \log m)$ für die Sortierung der m Tupel, siehe [Aho et al., 1983, Kap. 8]. Der Unterschied bei der gleichzeitigen Sortierung nach n Attributen besteht nur im aufwendigeren Vergleich, welches Tupel nun „kleiner oder größer" ist. Hier werden komponentenweise jeweils n Werte aus zwei Tupeln verglichen. Dies benötigt aber wie der Vergleich von einstelligen Tupeln jeweils nur $\mathcal{O}(1)$ Zeit. □

4.3.6 Ein Algorithmus für funktionale Abhängigkeiten

In diesem Abschnitt werden die Algorithmen FUNCTIONAL DEPENDENCIES und UPDATE sowie der Test „HAT–FA–ALS–TEILMENGE" beschrieben. Auf ihre Laufzeiten wird im nächsten Abschnitt eingegangen.

Algorithmus: FUNCTIONAL DEPENDENCIES

Eingabe: Eine Liste „Attributes" aller Attribute der Relation und für jedes Attribut A eine Liste Mls(A) der Attribute aus KSNN, die für Spezialisierungen in Frage kommen, und eine Liste EH(A) der einstelligen Hypothesen $A_i \rightarrow A$.

Ausgabe: Alle minimalen FAen, die nicht trivialerweise gelten.

1. Sei A das erste Element der Liste Attributes.

   Bottom–Up–Search–Start
   Teste die speziellste Hypothese $A_1 \dots A_n \rightarrow A$ mit $A_1, \dots, A_n \in Mls(A)$. Wenn diese FA nicht gilt, mache bei 1 mit dem nächsten Attribut weiter, sonst fahre fort.

   Top–Down–Search–Start
   Teste alle einstelligen Hypothesen $A_i \rightarrow A$ aus EH(A). Wenn $A_i \rightarrow A$ gilt, tue folgendes:

   (a) Streiche aus Mls(A) das Attribut A_i.

   (b) Rufe den Algorithmus UPDATE mit $A_i \rightarrow A$ auf.

   Gib alle erfolgreichen einstelligen Hypothesen als Teilergebnis aus. Ist die Liste MLS(A) leer, dann mache bei 1 mit dem nächsten Attribut weiter, sonst fahre fort.

   Top–Down–Search
   Sortiere die Bezeichner der Attribute aus der Liste Mls(A) nach ihrer lexikographischen Ordnung und in aufsteigender Reihenfolge. Seien $A_1 \dots A_n$ die so erhaltenen Attribute.

   (a) Bilde alle zweistelligen Kombinationen A_iA_j mit $i < j$, $i, j \in \{1, \dots, n\}$ und füge sie in eine Warteschlange ein.

(b) Führe den Test HAT–FA–ALS–TEILMENGE mit dem ersten Element $A_{r_1} \ldots A_{r_k}$ der Warteschlange aus. Wenn dieser negativ ist, überprüfe die Hypothese $A_{r_1} \ldots A_{r_k} \rightarrow A$ an der Datenbank. Gilt $A_{r_1} \ldots A_{r_k} \rightarrow A$, dann führe den folgenden ersten Punkt aus, sonst den zweiten und gehe dann zu (c). War der Test HAT–FA–ALS–TEILMENGE positiv, dann mache direkt mit (c) weiter.

   i. Gib die Hypothese $A_{r_1} \ldots A_{r_k} \rightarrow A$ als eine minimale FA aus. Rufe dann den Algorithmus UPDATE auf mit $A_{r_1} \ldots A_{r_k} \rightarrow A$.

   ii. Bilde alle $k+1$–stelligen Nachfolger $A_{r_1} \ldots A_{r_k} A_l$ mit $l \in \{k+1, \ldots, n\}$. Hänge diese an das Ende der Warteschlange.

(c) Wiederhole den Schritt (b) mit der nächsten Hypothese aus der Warteschlange und bilde bei k–stelligen Hypothesen k+1–stellige Nachfolger. Iteriere dies solange, bis die Warteschlange leer ist.

2. Mache bei (1) mit dem nächsten Attribut aus der Liste Attributes weiter, bis diese Liste leer ist.

Ende

Für die Eingabe des Algorithmus gilt, daß die Listen Mls(A) und EH(A) natürlich kein Attribut A_i enthalten mit $A_i = A$. Zu dem Punkt 1(b)ii in dem Algorithmus FUNCTIONAL DEPENDENCIES müssen zwei Dinge angemerkt werden. Zum einen kann die Menge der Nachfolger in 1(b)ii auch leer sein. Enthält die geordnete Liste Mls(A) die Elemente $(A_1, A_2, A_3, A_4, A_5)$, dann heißen die dreistelligen Nachfolger für A_1A_2: $A_1A_2A_3$, $A_1A_2A_4$ und $A_1A_2A_5$. Für A_1A_5 gibt es keine Nachfolger, d. h. A_1A_5 ist ein Blatt in diesem Baum, an dem die Suche endet.

Und zum anderen läßt sich leicht per Induktion zeigen, daß durch den Schritt 1(b)ii die Sortierung der Attribute innerhalb der linken Seiten der nach und nach generierten Hypothesen nicht zerstört wird. Dadurch wird auch sichergestellt, daß alle möglichen k–stelligen linken Seiten der Hypothesen genau einmal generiert werden. Dies gilt natürlich nur insoweit, als die Suche in einem Teilbaum nicht vorher beendet wird, weil eine FA gefunden wurde. Vgl. hierzu auch die Abbildung 4.5.

Algorithmus: UPDATE

Eingabe: Eine minimale FA $A_{r_1} \ldots A_{r_k} \rightarrow A$

1. Füge die Kante $A_{r_1} \ldots A_{r_k} \rightarrow A$ in den Graphen ein.

2. Unterscheide die folgenden beiden Fälle:

(a) $\underline{k = 1:}$ Suche nach allen FAen $A_{p_1} \dots A_{p_l} \rightarrow B$, die sich aufgrund der Transitivität ergeben und für die gilt: $B \geq A$, bezogen auf die lexikographische Ordnung auf den Attributnamen und $B \notin \{A_{p_1}, \dots, A_{p_l}\}$.
   i. $\underline{l = 1:}$ Lösche in Mls(B) das Attribut A_{p_l} und in EH(B) die Hypothese $A_{p_l} \rightarrow B$.
   ii. $\underline{l > 1:}$ Speicher die „möglicherweise minimale“ funktionale Abhängigkeit $A_{p_1} \dots A_{p_l} \rightarrow B$.

(b) $\underline{k > 1:}$ Suche nach allen FAen $A_{r_1} \dots A_{r_k} \rightarrow B$, die sich aufgrund der Transitivität ergeben und für die gilt: $B > A$ und $B \notin \{A_{r_1}, \dots, A_{r_k}\}$. Speicher alle diese FAen $A_{r_1} \dots A_{r_k} \rightarrow B$ als „möglicherweise minimal“.

Ende

Neben der Transitivität könnte in dem Algorithmus UPDATE auch noch die Pseudotransitivität der funktionalen Abhängigkeiten ausgenutzt werden, vgl. die Regel (PT), Lemma 2.2. Zum einen gilt jedoch bei der Pseudotransitivität genau wie bei der Transitivität, daß die resultierende FA nur „möglicherweise“ minimal ist, vgl. Beispiel 2.8 und 2.9. Es wird daher höchstens eine Datenbankabfrage eingespart. Und zum anderen erscheint der rechnerische Aufwand hier nicht angemessen. Angenommen es wird die FA $A_1 \dots A_n \rightarrow C$ gefunden. Dann müssen alle FAen $B_1 \dots B_m \rightarrow D$ daraufhin untersucht werden, ob $B_i = C$ für ein i gilt. Die resultierenden FAen $A_1 \dots A_n\ B_1 \dots B_{i-1} B_{i+1} \dots\ B_m \rightarrow D$ würden dann „recht lang“ in Bezug auf die Anzahl der Attribute auf der linken Seite.

Weiterhin fehlen noch die Fälle, in denen sowohl $B_1 \dots B_m$ sich z. B. erst durch Anwendung der Vereinigungsregel ergibt, als auch generell erst durch Anwendung der Axiome und Regeln die Pseudotransitivität oder Transitivität ausgenutzt werden kann. Aus diesen Gründen wird die Berücksichtigung der Pseudotransitivität bewußt außer Acht gelassen.

Bleibt als letztes in diesem Abschnitt die Beschreibung des Tests HAT–FA–ALS–TEILMENGE.

Eingabe: Die Attribute $A_{r_1}, \dots, A_{r_k}$ der linken Seite einer k–stelligen Hypothese $A_{r_1} \dots A_{r_k} \rightarrow A$ und alle l–stelligen minimalen FAen $A_{p_1} \dots A_{p_l} \rightarrow A$ mit $l < k$.

Ausgabe: „Wahr“ oder „falsch“.

1. Betrachte $A_{r_1}, \dots, A_{r_k}$ als eine geordnete Menge M und alle linken Seiten $A_{p_1}, \dots, A_{p_l}$ als geordnete Mengen M_1 bis M_n.

2. Existiert eine Menge M_i mit $M_i \subset M$ dann gib „wahr“ sonst „falsch“ als Ergebnis zurück.

Die Ordnung auf den Elementen der Mengen wird zur Beschleunigung der Überprüfung der Teilmengenbeziehung benötigt. Die generelle Notwendigkeit dieses Tests veranschaulicht die Abbildung 4.5. Angenommen es wird die FA $CE \rightarrow$ F gefunden. Dann darf nicht C und E aus $Mls(F)$ gestrichen werden. Sondern es muß nur ausgeschlossen werden, daß C und E zusammen auf der linken Seite einer später generierten Hypothese vorkommen. Z. B. müssen die Nachfolger ABC und ABE von AB noch generiert werden, $ABCE$ und $ABCDE$ jedoch nicht.

Korollar 4.2 *Der Test* HAT–FA–ALS–TEILMENGE *verhindert die Generierung zu spezieller Hypothesen und damit die Ausgabe nicht minimaler funktionaler Abhängigkeiten. Er sorgt weiterhin dafür, daß die Suche an der ersten zu speziellen Hypothese in einem Teilbaum endet.*

4.3.7 Analyse des Algorithmus

In diesem Abschnitt wird gezeigt, daß der Algorithmus FUNCTIONAL DEPENDENCIES vollständig und korrekt ist und daß er für jede Eingabe terminiert. Dazu werden die verschiedenen Phasen des Algorithmus einzeln analysiert. Zuvor wird jedoch auf den Test „Hat–FA–als–Teilmenge“ und den Algorithmus UPDATE näher eingegangen.

Laufzeit von Hat–FA–als–Teilmenge

Der Test Hat–FA–als–Teilmenge benötigt bei n bekannten FAen mit maximaler Länge k und einer „neuen“, erst noch zu testenden FA der Länge l eine Laufzeit von $\mathcal{O}(n*(k+l))$. Ein Beweis der linearen Laufzeit von $\mathcal{O}(k+l)$ für die Bestimmung von Teilmengen zwischen zwei geordneten Mengen findet sich in [Aho et al., 1983, Kap. 4].

Analyse von UPDATE

Bevor der Algorithmus FUNCTIONAL DEPENDENCIES aufgerufen wird, müssen alle trivialen FAen in einem gerichteten Graphen G(V,E) gespeichert sein. Dabei enthält die Menge V alle Knoten und die Menge E alle Kanten. Die Knoten v_i werden mit den linken bzw. rechten Seiten der FAen $A_1 \ldots A_n \rightarrow B$ markiert. Es existiert eine Kante von v_i nach v_j genau dann, wenn es eine FA gibt mit $(v_i = A_1 \ldots A_n) \rightarrow (B = v_j)$.

Nach dem Aufbau dieses Graphen gibt es zwischen je zwei Attributen A und B aus SS oder SK zumindest einen Weg. Es existiert jedoch noch kein Knoten v_i mit $v_i = A_1 \ldots A_n$ und $n \geq 2$. Bei n Attributen und damit n Knoten gibt es maximal $n*(n-1)$ Kanten. Der Aufwand zum Aufbau dieses Graphen ist folglich $\mathcal{O}(n^2)$.

Weiterhin kommt in dem Algorithmus UPDATE zweimal die bewußt „ungenaue“ Formulierung vor, „suche nach allen FAen, die sich aufgrund der

Transitivität ergeben". Es heißt nicht, „berechne mit dem Algorithmus von z. B. Warshall die transitive Hülle". Das Problem ist, daß es hier ausreichend ist, die transitive Hülle inkrementell zu berechnen. Bei jedem Aufruf von UPDATE kommt eine Kante hinzu, und es sind nur diejenigen Wege von Interesse, die aufgrund der neuen Kante entstanden sind. Hier wäre es wünschenswert einen Algorithmus zu haben, der dieses Problem „effizient" löst.

Dieses Problem wird hier auf einfache Weise wie folgt gelöst: es werden zwei Suchdurchläufe von der Eingabe $A_{r_1} \dots A_{r_k} \rightarrow A$ aus gestartet, einmal „in Richtung der Kanten" mit besuchten Knoten v_i und einmal in „umgekehrter Richtung" mit besuchten Knoten w_i. Dabei bilden alle Kombinationen von $w_i \rightarrow v_j$ „neue" FAen, die sich aufgrund der Transitivität ergeben. Dadurch ist das Problem, nur die wirklich neuen Wege zu finden, natürlich nicht gelöst. Für alle diese FAen werden dann die Anweisungen in (1.a) und (1.b) bzw. (2) ausgeführt. In der Implementierung wird zusätzlich eine Art Cache verwendet, um die Anweisungen in (1.a) nicht mehrfach für die selben Attribute auszuführen.

Ein weiteres und schwierigeres Problem sind die Knoten mit mehrstelligen Markierungen. Wie gezeigt wurde, kann es $\mathcal{O}(2^n)$ viele minimale FAen geben, vgl. Lemma 4.6. Folglich kann der Graph $\mathcal{O}(2^n)$ viele Knoten mit mehrstelligen Markierungen enthalten. Der Aufwand für die beiden Fälle in dem Algorithmus UPDATE läßt sich wie folgt abschätzen.

1. Es gibt nur $\mathcal{O}(n^2)$ viele einstellige FAen, folglich wird der Algorithmus maximal $\mathcal{O}(n^2)$ mal für den Fall (1) durchlaufen.
   (a) Das Löschen eines Elementes in einer geordneten Liste benötigt $\mathcal{O}(n)$
   (b) Dieser Schritt benötigt bei m Knoten und e Kanten auf der einen Seite „nur" $\mathcal{O}(m + e)$ Zeit und Platz. Gemessen an der Eingabe von n Attributen ist dies jedoch nur pseudopolynomiell, da $m = \mathcal{O}(2^n)$ werden kann. Für den Platzbedarf gilt das gleiche.
2. Da es $\mathcal{O}(2^n)$ viele FAen geben kann, wird dieser Fall dann $\mathcal{O}(2^n)$ mal durchlaufen. Jedoch gibt es bei n Attributen und einstelligen rechten Seiten nur n viele Möglichkeiten für das Attribut auf der rechten Seite. Der Aufwand für jeden Durchlauf beträgt $\mathcal{O}(n)$.

Damit ergibt sich:

Lemma 4.9 *Der Algorithmus* UPDATE *hat eine Laufzeit von:*

$$\mathcal{O}(n^2 * 2^n) = \mathcal{O}(2^n)$$

Lemma 4.10 *Der Algorithmus* UPDATE *ist korrekt, d. h. er sorgt dafür, daß alle minimalen FAen gefunden werden und keine Hypothesen generiert werden, die zu nicht minimalen FAen führen.*

Zum Beweis dieses Lemmas ist es ausreichend, vier Dinge zu zeigen:

1. Da jede gefundene minimale FA in den Graphen eingefügt wird, ist es immer möglich, jede FA zu finden, die sich aufgrund der Transitivität ergibt. Somit können alle minimalen FAen gefunden werden.

2. Da der Algorithmus FUNCTIONAL DEPENDENCIES erst einen Teilsuchraum vollständig durchsucht, bevor mit dem nächsten Attribut auf der rechten Seite weitergemacht wird, genügt es, in den Schritten 2(a) und 2(b) alle FAen mit $B < A$ außer acht zu lassen, da zu diesem Zeitpunkt alle Teilsuchräume mit $B < A$ auf der rechten Seite schon abgearbeitet sind. Der Fall $B = A$ kann bei der hier gewählten Art der Graphdarstellung nur in 2(a) auftreten. In 2(b) kann $B = A$ ausgeschlossen werden, weil dann das Ergebnis der Suche gleich der Eingabe für den Algorithmus wäre.

3. Das Löschen von Attributen in Mls(B) in dem Schritt 2(a)i ist notwendig, damit keine nicht minimalen FAen gefunden werden.

4. Das Löschen von Attributen in EH(B) in dem Schritt 2(a)i ist notwendig, damit keine überflüssigen Datenbankanfragen gestellt werden.

Beweis

1. Beweis durch vollständige Induktion. Trivial.

2. Beweis durch vollständige Induktion. Trivial.

3. Sei die FA $F \rightarrow C$ eine Kante im Graph und enthalte Mls(D) die Attribute A,F und G. Sei die FA $C \rightarrow D$ die Eingabe. Dann ergibt sich über die Transitivität $F \rightarrow D$. Angenommen F wird in Mls(D) nicht gestrichen, dann wird in der Phase Top–Down–Search im Algorithmus FUNCTIONAL DEPENDENCIES unter anderem die Hypothese $AF \rightarrow D$ generiert. Diese ist aber nicht minimal, da sie eine Spezialisierung von $F \rightarrow D$ ist. Dieses ist ein Widerspruch dazu, daß der Algorithmus FUNCTIONAL DEPENDENCIES nur minimale FAen berechnet. Folglich ist der Schritt 2(a)i in dem Algorithmus UPDATE notwendig.

4. Seien die Annahmen wie oben. Angenommen F wird in EH(D) nicht gestrichen. Dann wird in der Phase Top–Down–Search–Start unter anderem die Hypothese $F \rightarrow D$ generiert und getestet, obwohl sie abgeleitet werden konnte. Die damit verbundene Datenbankanfrage ist somit überflüssig.

□

Bemerkung 4.2 *Der einzige Schritt des Algorithmus* UPDATE, *der den Hypothesenraum wirksam beschränkt, ist der Schritt (1.a). Das Speichern von nur möglicherweise minimalen FAen spart — wie erwähnt — immer nur höchstens eine Anfrage an die Datenbank. Je nachdem aus wieviel Tupeln und Attributen eine Relation besteht, kann es sinnvoll sein, auf die Fälle (1.b) und (2) zu verzichten.*

Analyse des Algorithmus FUNCTIONAL DEPENDENCIES

Lemma 4.11 *Die Eingabe des Algorithmus* FUNCTIONAL DEPENDENCIES *ist hinreichend zur Berechnung aller nicht trivialen funktionalen Abhängigkeiten.*

Beweis Zur Diskussion, warum die Listen Mls(A_i) ausreichen, um alle minimalen FAen $A_1 \dots A_n \to A$ zu berechnen, vergleiche Theorem 4.3. Da weiterhin alle Attribute der Relation in der Liste Attributes enthalten sind, ist sichergestellt, daß jedes Attribut auf der rechten Seite einer funktionalen Abhängigkeit vorkommen kann. □

Die beiden folgenden Ergebnisse zeigen die Korrektheit der Abbruchkriterien in den beiden ersten Phasen des Algorithmus.

Bottom–up–Search–Start

Lemma 4.12 *Es gelte $A_1 \dots A_n \not\to B$. Dann existiert für kein r mit $r \in \{1, \dots, n-1\}$ eine funktionale Abhängigkeit $A_{i_1} \dots A_{i_r} \to B$ mit $\{A_{i_1}, \dots, A_{i_r}\} \subseteq \{A_1, \dots, A_n\}$.*

Beweis Angenommen die funktionale Abhängigkeit $A_{i_1} \dots A_{i_r} \to B$ gilt und $A_1 \dots A_n \not\to B$. Dann wird mit Hilfe des Erweiterungs–Axioms (E) die FA zu $A_{i_1} \dots A_{i_r} A_{i_r+1} \dots A_{i_n} \to A_{i_r+1} \dots A_{i_n} B$ erweitert, wobei alle Attribute A_i links genau einmal vorkommen. Auf diese FA wird dann die Dekompositionsregel (D) angewendet. Man erhält:

$$A_{i_1} \dots A_{i_r}\, A_{i_r+1} \dots A_{i_n} \to B = A_1 \dots A_n \to B$$

Dies ist ein Widerspruch zur Annahme. □

Top–Down–Search–Start

Lemma 4.13 *Der Schritt (a) in Top-Down-Search-Start im Algorithmus verhindert, daß in der Top-Down-Search Phase eine Spezialisierung einer minimalen FA generiert und getestet wird.*

Beweis Sei die funktionale Abhängigkeit $A_j \rightarrow B$ gültig. A_j kommt nach Konstruktion in Mls(B) vor und wird nicht gestrichen. Dann wird in (a) in Top–Down–Search entweder $A_i A_j$ oder $A_j A_k$ generiert mit $i < j < k$. Sowohl $A_i A_j \rightarrow B$ als auch $A_j A_k \rightarrow B$ gelten aufgrund des Axioms (E) und der Regel (D) und sind nicht minimal. □

Korollar 4.3 *Ist die Liste Mls(B) leer, dann gibt es keine minimalen funktionalen Abhängigkeiten mit* $A_1 \ldots A_n \rightarrow B$ *und* $n \geq 2$.

Top–Down–Search

Lemma 4.14 *In der Phase Top–Down–Search wird der Wald der möglichen funktionalen Abhängigkeiten mit Wurzelknoten* $A_i A_j$ *mit einer Breitensuche vollständig nach minimalen FAen durchsucht.*

Beweis Die Breitensuche in dem Wald ist unter anderem durch die Verwendung einer Warteschlange offensichtlich. Da zu Beginn alle Kombinationen $A_i A_j$ mit $i < j$ in die Warteschlange eingefügt werden, erstreckt sich die Suche über alle Teilbäume. Weiterhin genügen alle während der Suche generierten Nachfolger von Knoten Lemma 4.7. Dadurch ist sichergestellt, daß jede mögliche Hypothese generiert werden kann. Weiterhin verhindert der Test Hat–Fa–Als–Teilmenge, daß zu spezielle Hypothesen ausgegeben werden, vgl. Lemma 4.2. Dieser Test sorgt dann auch dafür, daß die Suche dann an dieser zu speziellen Hypothese in diesem Teilbaum endet. Da weiterhin die Suche an einem Knoten in einem Teilbaum beendet wird, wenn eine minimale FA gefunden wurde — vgl. Schritt (b) in der Phase Top–Down–Search —, ergibt sich insgesamt die Behauptung. □

Da der Algorithmus FUNCTIONAL DEPENDENCIES die drei Phasen mit jedem Attribut der Relation auf der rechten Seite durchläuft, siehe die Punkte (1) und (2) im Algorithmus, resultiert daraus das nächste Theorem.

Theorem 4.6 *Der Algorithmus* FUNCTIONAL DEPENDENCIES *ist vollständig und korrekt.*

Beweis Folgt aus den Lemmata 4.10 und 4.12 bis 4.14. □

Theorem 4.7 *Die Laufzeit des Algorithmus* FUNCTIONAL DEPENDENCIES *beträgt insgesamt* $\mathcal{O}(n * 2^n * n^2 * 2^n) = \mathcal{O}(n^3 * 2^{2n}) = \mathcal{O}(2^{2n})$.

Beweis Die Faktoren $n^2 * 2^n$ stammen aus dem Lemma 4.9. Da der Algorithmus vollständig ist, siehe Theorem 4.6, werden im schlechtesten Fall alle möglichen Hypothesen generiert. Diese lassen sich mit $\mathcal{O}(n * 2^n)$ abschätzen, siehe Lemma 4.5. Damit folgt die Behauptung. □

Dieser Algorithmus nimmt dann die Laufzeit aus dem Theorem an, wenn es sehr viele, $\mathcal{O}(2^n)$, FAen gibt und zusätzlich in dem Graphen viele Wege existieren. In diesem Fall benötigt jeder Algorithmus ohne Ausnutzung der Transitivität eine Laufzeit von $\Omega(n*2^n)$. Weiterhin ist es egal, daß in diesem Fall die Laufzeit des Algorithmus, wie im Theorem 4.7 angegeben, schlechter ist als unbedingt notwendig.

Angenommen eine Datenbank besteht aus 20 Attributen, und es existieren maximal viele FAen. Wenn es möglich ist, die Gültigkeit jeder FA in einer Sekunde an der Datenbank zu testen, würde der Algorithmus schon über drei Wochen rechnen. Hierbei ist die Annahme von einer Sekunde sehr wenig, bei hinreichend vielen Tupeln liegen die Antwortzeiten im Minutenbereich. Selbst die Ausnutzung der Transitivität spart hierbei im Verhältnis zur Gesamtzahl der zu testenden Hypothesen nur wenige Anfragen ein. Daher ist die Laufzeit für diesen Fall irrelevant.

Bleiben also die Fälle, in denen wenige funktionale Abhängigkeiten existieren. Auf das Verhalten dieses Algorithmus in diesen Fällen wird im nächsten Kapitel eingegangen.

Kapitel 5

Vergleich

In diesem Kapitel werden zuerst zwei andere Ansätze zur Entdeckung funktionaler Abhängigkeiten vorgestellt. Daran schließt sich ein experimenteller Vergleich zwischen diesen beiden Ansätzen und dem hier vorgestellten Algorithmus an. Beendet wird dieses Kapitel mit einer Diskussion der Gemeinsamkeiten und Unterschiede der besprochenen Realisierungen.

5.1 Verwandte Ansätze zur Bestimmung funktionaler Abhängigkeiten

Der in [Schlimmer, 1993] beschriebene Algorithmus ist von der Grundidee identisch mit dem hier beschriebenen Algorithmus FUNCTIONAL DEPENDENCIES. Der Suchraum wird Top–Down mit einer Breitensuche durchsucht. Im Gegensatz zu dem hier verwendeten Test, HAT–FA–ALS–TEILMENGE, wird die Überprüfung zu spezieller Hypothesen durch die Verwendung einer „perfekten Hashfunktion“ vermieden. Weiterhin wird der Suchraum generell durch eine Konstante k beschränkt, die die maximale Länge der linken Seiten der funktionalen Abhängigkeiten vorgibt. Die Gültigkeit einer Hypothese $X \rightarrow A$ wird in diesem Algorithmus mit einem Test überprüft, der linear in der Anzahl der Tupel ist. Verantwortlich hierfür ist die Verwendung des Hashing.

Der Platzbedarf dieses Algorithmus beträgt $\mathcal{O}(n^k)$ und die Gesamtlaufzeit $\mathcal{O}(m * k * n^{k+1})$, dabei wird die Anzahl der Attribute wieder mit n und die Anzahl der Tupel mit m bezeichnet. Asymptotisch ist dieser Algorithmus exponentiell in der Anzahl der Attribute, was er aufgrund von Theorem 4.2 auch sein muß.

Obwohl Schlimmer nicht explizit davon spricht, sind die entdeckten funktionalen Abhängigkeiten minimal im Sinne von Definition 2.10. Dies ist eine direkte Konsequenz aus der beschriebenen Art der Suche.

Einen anderen Ansatz verfolgen [Safnik und Flach, 1993]. Um die Anzahl der Hypothesen, die an den Daten überprüft werden müssen, zu minimie-

Datenbank	$\|r\|$	$\|R\|$	$\|X\|$	Zeit	FAen
Lymphography	150	19	19	18 Min.	1248
Lymphography	150	19	10	16 Min.	1226
Lymphography	150	19	7	9 Min.	641
Breast Cancer	699	11	4	1 Std. 14 Min.	31
Bridges	108	13	4	13 Min.	23

Tabelle 5.1: Zusammenfassung der experimentellen Ergebnisse aus [Safnik und Flach, 1993] und [Schlimmer, 1993].

ren, führen sie das Konzept der „negativen Hülle“ ein. Darunter verstehen sie die Menge der speziellsten, ungültigen funktionalen Abhängigkeiten. Jede einzelne dieser FAen enthält rechts wiederum nur ein Attribut. In der hier verwandten Notation sind das funktionale Abhängigkeiten der Form $AB \not\rightarrow C$. Die Laufzeit für die Konstruktion dieser negativen Hülle ist quadratisch gemessen in der Anzahl der Tupel. Ihr Ziel erreichen sie dadurch, daß der Test zur Überprüfung der Konsistenz einer Hypothese in eine Suche nach spezielleren Hypothesen in der negativen Hülle transformiert wird. So werden insbesondere die m Tupel der Relation nur einmal untersucht.

Der weitere Ablauf des Algorithmus ist wieder analog zu den bisher beschriebenen Algorithmen. Die Hypothesen werden Top–Down in einer Breitensuche generiert und getestet. Dabei werden die üblichen Abbruchkriterien benutzt, d. h. es werden wiederum keine Spezialisierungen gültiger Hypothesen mehr betrachtet. Auch für diesen Algorithmus gilt, daß die Gesamtlaufzeit exponentiell gemessen an der Anzahl der Attribute ist, genau $\mathcal{O}(n * 2^{2*n-2})$.

Genau wie der obige Algorithmus benutzt auch dieser Algorithmus eine optionale Tiefenbeschränkung, d. h. die Länge der linken Seiten der FAen kann durch eine Konstante beschränkt werden. Darüberhinaus erlaubt dieser Algorithmus die explizite Angabe, wieviel Tupel gegen eine FA verstoßen dürfen, damit diese dennoch als „gültig“ betrachtet wird.[1] Auch die mit diesem Algorithmus bestimmten funktionalen Abhängigkeiten sind im hier verwendeten Sinne minimal.

5.2 Experimentelle Ergebnisse

Die Tabelle 5.1 zeigt eine Zusammenfassung der experimentellen Ergebnisse aus [Safnik und Flach, 1993] und [Schlimmer, 1993]. Dabei bezeichnet $|r|$ die Anzahl der Tupel in den Datenbanken und $|R|$ die Anzahl der Attribute. Weiterhin steht $|X|$ für die maximale Anzahl der Attribute auf der linken Seite der funktionalen Abhängigkeiten. Die Laufzeiten der Algorithmen sind in Minuten CPU–Zeit angegeben. In der letzten Spalte steht jeweils die Anzahl entdeckter, minimaler funktionaler Abhängigkeiten.

[1] Auf eine Erweiterung des Algorithmus FUNCTIONAL DEPENDENCIES, die diese Idee aufgreift, wird in Abschnitt 6.3.3 eingegangen.

Datenbank	$\lvert r\rvert$	$\lvert R\rvert$	$\lvert X\rvert$	Zeit	FAen
Lymphography	150	19	7	> 33 Std.	262[2]
Breast Cancer	699	11	11	15 Min.	22
Breast Cancer	699	11	4	8 Min.	22
Breast Cancer	683	11	11	42 Min.	48
Breast Cancer	683	11	4	14 Min.	36
Bridges V1	108	13	13	7 Sek.	13
Bridges V1	108	13	4	6 Sek.	13
Bridges V1	70	13	13	10 Min.	126
Bridges V1	70	13	4	4 Min.	114
Bridges V2	108	13	13	10 Sek.	12
Bridges V2	108	13	4	9 Sek.	12
Bridges V2	70	13	13	11 Min.	71
Bridges V2	70	13	4	3 Min.	59
Bücher	9931	9	9	4 Std. 44 Min.	25
Bücher	9931	9	6	4 Std. 40 Min.	25
Bücher	9931	9	3	2 Std. 10 Min.	20

Tabelle 5.2: Zusammenfassung der vom Algorithmus FUNCTIONAL DEPENDENCIES entdeckten minimimalen funktionalen Abhängigkeiten bei verschiedenen Datenbanken.

Die Datenbank „Lymphography“ wird in [Safnik und Flach, 1993] verwendet und „Bridges“ und „Breast Cancer“ in [Schlimmer, 1993]. Der Algorithmus von Safnik und Flach erlaubt die Angabe, wieviel widersprüchliche Tupel für jede FA erlaubt sind. Da dieses Verhalten mit dem hier vorgestellten Algorithmus nicht simuliert werden kann, sind nur die Ergebnisse aufgeführt, in denen keine widersprüchlichen Tupel erlaubt waren.

Die entsprechenden Ergebnisse des Algorithmus FUNCTIONAL DEPENDENCIES sind in der Tabelle 5.2 aufgeführt. Die gemessene Zeit[3] schließt hier den Algorithmus INCLUSION DEPENDENCIES mit ein, da dessen Ergebnisse, wie erläutert, in dem Algorithmus FUNCTIONAL DEPENDENCIES zum Teil verwendet werden. In der bisher beschriebenen Realisierung existiert in dem Algorithmus FUNCTIONAL DEPENDENCIES keine Tiefenbeschränkung, da er alle minimalen FAen entdecken soll. Damit die Testergebnisse aber vergleichbar werden, wird eine solche eingeführt. Dies geschieht im Schritt 1(b)ii des Algorithmus, in dem nur noch Nachfolger gebildet werden, wenn diese Tiefenbeschränkung dadurch nicht verletzt wird.

Bis auf die Datenbank Bücher sind die verwendeten Datenbanken weitestgehend identisch mit denen aus den beiden angeführten Arbeiten. Die Da-

[2] Das Programm wurde mangels Speichers durch einen internen Fehler des Prologsystems abgebrochen.

[3] Das in Prolog realisierte Programm lief auf einer Sparcstation ELC und die Datenbank Oracle7 auf einer Sparcstation 10.

tenbank Lymphography zum Beispiel enthält in der erhaltenen Version zwei Tupel weniger als in [Safnik und Flach, 1993] angegeben. Oder es geht aus [Schlimmer, 1993] nicht hervor, welche der beiden Versionen der Datenbank „Bridges“ verwendet wurde.

Werden jedoch die Ergebnisse aus den Tabellen 5.1 und 5.2 miteinander verglichen, so fällt sofort auf, daß die Anzahlen der entdeckten minimalen FAen bei gleichen Testbedingungen differieren. Im weiteren Verlauf wird davon ausgegangen, daß die beteiligten Algorithmen korrekt sind, folglich müssen die Unterschiede von den Daten verursacht werden. Abgesehen von den oben angesprochenen Problemen, können Fehler bei der Datenübertragung oder der Dateneingabe in die Datenbanken zum größten Teil ausgeschlossen werden.

Daher bleibt als Ursache eine unterschiedliche Behandlung der vorkommenden NULL–Werte übrig. Zur Erinnerung sei gesagt, daß gemäß der Datenbanktheorie alle NULL–Werte für ein Attribut als voneinander verschieden betrachtet werden. Weiterhin sind NULL–Werte „nicht existent“. Der hier beschriebene Algorithmus FUNCTIONAL DEPENDENCIES behandelt alle Nullwerte gemäß dieser Theorie. Dies führt dazu, daß ein Attribut, das NULL–Werte enthält, nicht auf der linken Seite einer minimalen FA vorkommen kann. Für die Datenbank Bridges, in der ein Attribut ein Schlüsselkandidat ist, heißt dieses, daß nur noch 3 weitere Attribute für die Suche nach FAen in Frage kommen. Aus diesem Schlüsselkandidaten ergeben sich allein 12 FAen, und aufgrund der Suchstrategien des Algorithmus verbleibt die Laufzeit im Sekundenbereich.

Eine Analyse der erhaltenen Datensätze für die vier Datenbanken legt jedoch die Vermutung nahe, daß die beiden anderen Algorithmen NULL–Werte anders behandeln. Die beiden Tests mit den Datenbanken Bridges und Breast Cancer zeigen, daß, nachdem alle Tupel gelöscht worden sind, in denen NULL–Werte enthalten waren,[4] mehr FAen gefunden werden. Dies sind jedoch mehr, als bei Schlimmer angegeben. Somit verbleibt die Möglichkeit, daß alle NULL–Weret als identisch, jedoch verschieden von den „normalen“ Werten betrachtet werden. Diese Vermutung wird auch von der Analyse der Datensätze gestützt. Sie läßt sich aber mit dem Algorithmus FUNCTIONAL DEPENDENCIES nicht verifizieren, da die Interpretation der NULL–Werte durch das DBMS nicht veränderbar ist.

Bevor im nächsten Abschnitt die obigen Ergebnisse eingehend diskutiert werden, erfolgen noch einige Anmerkungen zu der Entdeckung der IAen. Die benötigten Zeiten zur Entdeckung aller IAen liegen außer bei der Datenbank Bücher jeweils unter einer Sekunde. Die Experimente bestätigten zweierlei. Erstens ist die Anzahl der Anfragen ganz entscheidend von der Reihenfolge abhängig, in der die Attribute betrachtet werden. Ist die Reihenfolge jedoch optimal, dann kommt der Algorithmus INCLUSION DEPENDENCIES mit der minimal notwendigen Anzahl von Anfragen aus, vergleiche hierzu das Beispiel 4.3. Und zweitens werden — wie nicht anders zu erwarten — genau

[4] Es handelt sich hier um die Tests in der Tabelle 5.2, bei denen $|r|$ kleiner geworden ist.

$\frac{n*(n-1)}{2}$ Anfragen benötigt, wenn keine IAen existieren. Dies gilt unter anderem für die Datenbank Bücher. Dort benötigen die notwendigen 36 Anfragen ungefähr drei Minuten.

5.3 Diskussion

Bei der Bewertung der mit dem Algorithmus FUNCTIONAL DEPENDENCIES erzielten Ergebnisse muß zwischen drei Aspekten unterschieden werden. Da ist zum ersten der Vergleich mit [Safnik und Flach, 1993]. Ihr Algorithmus verwendet keine Datenbank, und damit ist schon alles gesagt. Allein bei der Datenbank Lymphography sind bei einer Tiefenbeschränkung von 7 im schlechtesten Fall über 500.000 Datenbankanfragen nötig. Da mehrere hundert FAen exitieren, ist zu vermuten, daß diese Zahl auch annähernd erreicht wird. Bei der Annahme von einer Sekunde Rechenzeit pro Datenbankanfrage resultiert hieraus eine Gesamtlaufzeit von über sechs Tagen für den Algorithmus FUNCTIONAL DEPENDENCIES.

Ein Versuch, die Frage zu beantworten, in wieweit diese großen Unterschiede durch einen „überlegenen Algorithmus“ oder nur durch den Verzicht auf die Verwendung einer Datenbank verursacht werden, bleibt Spekulation. Hierzu wäre es erforderlich, daß einer von beiden Algorithmen so geändert wird, daß die Bedingungen für beide gleich sind, d. h. entweder benutzen beide oder keiner eine Datenbank.

Der zweite Aspekt wird bestimmt durch den Vergleich mit den Ergebnissen aus [Schlimmer, 1993]. Obwohl auch Schlimmer keine Datenbank verwendet, so zeigt der Algorithmus FUNCTIONAL DEPENDENCIES hier eine überzeugende Leistung. Auch die vollständige Durchsuchung des Hypothesenraumes ist jedesmal schneller als bei Schlimmer. Im Vergleich mit dem obigen Ergebnis zeigt sich hier wie wichtig die Anzahl der existierenden FAen für die Laufzeit des Algorithmus FUNCTIONAL DEPENDENCIES ist. Denn die Unterschiede zwischen den Datenbanken, die Anzahl der Tupel und der Attribute betreffend, sind nicht so groß, als daß sie die Ursache für die Laufzeitunterschiede sind.

Und als letztes zeigen die Ergebnisse für die Datenbank Buecher einen weiteren Aspekt. Diese Datenbank enthält nur neun Attribute und bei einer Tiefenbeschränkung von drei nur 20 FAen. Von diesen ergeben sich allein acht aufgrund eines Schlüssels. Weiterhin sind maximal auch nur ungefähr 800 Hypothesen zu testen. Trotzdem benötigt dieser Test wesentlich länger als die Tests mit den Datenbanken Bridges und Breast Cancer. Dabei ist das Suchproblem bei diesen auch noch wesentlich schwieriger, da mehr Attribute existieren und die Tiefenbeschränkung größer ist. Allein die Anzahl der Tupel in der Datenbank Bücher ist um ein vielfaches größer, als bei allen anderen verwendeten Datenbanken.

Die beiden weiteren Tests mit dieser Datenbank, einmal mit einer Tiefenbeschränkung von sechs und einmal ohne Tiefenbeschränkung, zeigen noch ein weiteres interessantes Ergebnis. Bei diesen beiden Tests ist die Anzahl der entdeckten FAen gleich. Und obwohl der Hypothesenraum vollständig nach FAen durchsucht wird, steigt die Rechenzeit nur minimal an. Dies ist ein Beleg sowohl für die Effizienz des Algorithmus, als auch für die verwendeten Schnittkriterien.

Als Fazit lassen sich zwei Punkte festhalten. Entweder die Datenbank ist klein genug, so daß kein DBMS zur Speicherung der Daten benötigt wird. Dann sollte das Verfahren zum Beispiel aus [Safnik und Flach, 1993] Verwendung finden. Hier besteht die Hoffnung, daß in hinreichend kurzer Zeit, auch bei der Existenz von vielen FAen, der gesamte Hypothesenraum durchsucht werden kann. Die Tabelle 5.1 zeigt dies für die Datenbank Lymphography.

Wenn aber ein DBMS benötigt wird, da die Datenmenge entsprechend umfangreich ist, dann sind lange Antwortzeiten unvermeidlich. Sollten darüberhinaus noch sehr viele FAen existieren, dann wird die Berechnung aller FAen praktisch undurchführbar. Auch die Verwendung einer Tiefenbeschränkung wird hier dann nicht viel weiterhelfen. Die Tabelle 5.2 illustriert diese zwei Punkte für die Datenbanken Bücher und Lymphography.

Die obige Diskussion läßt sich wie folgt auf den Punkt bringen. Mit dieser Arbeit wird ein Verfahren vorgestellt, daß ohne Benutzerbeteiligung alle minimalen funktionalen Abhängigkeiten in einer relationalen Datenbank entdeckt. Durch die Verwendung einer Datenbank sind dem Verfahren keinerlei Beschränkungen hinsichtlich der Anzahl der Tupel auferlegt, im Gegensatz zu den beiden anderen vorgestellten Verfahren. Der Vergleich mit dem Verfahren aus [Schlimmer, 1993] zeigt auch, daß hiermit dennoch ein Effizienzgewinn verbunden werden kann.

Weiterhin kann davon ausgegangen werden, daß in existierenden Datenbanken die offensichtlichen FAen bereits dahingehend ausgenutzt wurden, die Datenbank zu strukturieren und daß in den einzelnen Relationen sehr viele Tupel enthalten sind. Somit ist zu erwarten, daß in den einzelnen Relationen die Anzahl der FAen eher gering ist. Für diese Anwendungssituationen wird in dieser Arbeit ein Verfahren vorgestellt, das die Aufgabe minimale FAen zu entdecken, sehr effizient löst.

Kapitel 6

Ausblick

Dieses Schlußkapitel beginnt mit einer kurzen Zusammenfassung dieser Arbeit. Daran schließt sich eine Diskussion der erzielten Ergebnisse an. Beendet wird dieses Kapitel mit einem Ausblick auf mögliche Verbesserungen der vorgestellten Algorithmen.

6.1 Zusammenfassung

In dieser Arbeit werden drei Algorithmen zur Bestimmung numerischer Integritätsbedingungen, unärer Inklusionsabhängigkeiten und minimaler funktionaler Abhängigkeiten vorgestellt. Die zu ihnen korrespondierenden Programme arbeiten jeweils auf bestehenden relationalen Datenbanken. Alle drei verzichten dabei auf eine direkte Benutzerbeteiligung. Stattdessen werden die Informationen ausgewertet, die vorher zum Anlegen der Datenbanken benötigt wurden und in den Systemtabellen weiterhin vorliegen. Dadurch können die Progamme vollautomatisch ablaufen.

Sowohl für unäre Inklusionsabhängigkeiten als auch für funktionale Abhängigkeiten existieren Axiomatisierungen. Die wichtigste Eigenschaft, die für beide aus diesen resultiert, ist die Transitivität. Diese wird in den entsprechenden Algorithmen ausgenutzt, so daß sich, bezogen auf die Anzahl durchzuführender Datenbankanfragen, erhebliche Einsparungen ergeben. Danaben werden für die IAen und FAen Vorauswahlen getroffen, womit, vor dem eigentlichen Beginn der Algorithmen, die Anzahlen der Attribute, die überhaupt für IAen und FAen in Frage kommen, beträchtlich eingeschränkt werden.

Bei der Entdeckung der FAen werden insbesondere nur minimale FAen betrachtet. Es wird eine Menge F berechnet, in der alle gültigen minimalen funktionalen Abhängigkeiten enthalten sind. Auf der einen Seite wird durch diese Einschränkung der Suchraum aller möglichen FAen massiv beschränkt. Auf der anderen Seite läßt sich aber jede in einer Datenbank gültige FA mit Hilfe der Axiomatisierung aus diesen minimalen FAen ableiten, so daß der

Algorithmus nichts von seiner Vollständigkeit einbüßt. Vollständigkeit bedeutet in diesem Zusammenhang, daß jede mögliche gültige FA ein Element der Hülle F^* ist. Verfahren zur Durchführung dieser Ableitungen werden hier jedoch nicht vorgestellt.

6.2 Diskussion

Die in dieser Arbeit vorgestellten Algorithmen erfüllen die an sie gestellten Aufgaben, nämlich numerische Intergritätsbedingungen, unäre Inklusionsabhängigkeiten und funktionale Abhängigkeiten zu entdecken. Hierauf aufbauend können all jene Aufgaben und Probleme gelöst werden, die im Kapitel über die Motivation dieser Arbeit aufgeführt sind, zur Erinnerung sei nur das Datenbank–Design genannt.

Alle diese Aufgaben beschäftigen sich mit relationalen Datenbanken, insbesondere für das Datenbank–Design mit Normalformen werden relationale Datenbanken zwangsläufig vorausgesetzt. Weiterhin war es ein Ziel, Algorithmen zu entwickeln, die mit existierenden Datenbanksystemen arbeiten können, in diesem Fall Oracle7. Und es war nicht die Aufgabe, z. B. den Algorithmus aus [Safnik und Flach, 1993] an Effizienz zu „überbieten", der gerade keine Datenbank verwendet.

Diese Zielsetzung muß bei der Bewertung des experimentellen Vergleichs im letzten Kapitel berücksichtigt werden. Daher ist es um so höher zu bewerten, daß der Algorithmus Functional Dependencies nicht hinter dem Algorithmus aus [Schlimmer, 1993] zurücksteht, der ebenfalls keine Datenbank verwendet, sondern daß er schneller ist.[1]

Dieser Anforderung der Benutzung bestehender Datenbanksysteme wird auch durch die Verwendung von SQL als Datenbankanfragesprache Rechnung getragen. Dadurch ist eine Anpassung an beliebige relationale Datenbanksysteme, die SQL als Anfragesprache anbieten, sehr leicht möglich. Dazu ist nur eine Anpassung und Neuübersetzung des Schnittstellenmoduls zur Anbindung der Algorithmen an die Datenbank nötig, da jede Datenbank eigene Bibliotheken hierfür bereitstellt, nicht jedoch eine Änderung der Algorithmen.

Dadurch, daß das Datenbanksystem nicht nur als reiner Datenspeicher verwendet wird, sondern selbst die Gültigkeit von FAen und IAen berechnet, angestoßen durch SQL–Anfragen, entsteht keine Beschränkung für die Menge der Daten von seiten des Algorithmus. Allein das DBMS entscheidet über die maximale Größe der Datenbanken, und damit über die Anwendbarkeit des Algorithmus Functional Dependencies. Dieses ist ein wichtiger Unterschied zu den beiden oben zitierten Verfahren, die an ihre Grenzen stoßen, wenn die Datenmenge die Grenze des Primärspeichers übersteigt. In dieser Beziehung sind dem hier vorgestellten Algorithmus keine Grenzen gesetzt.

[1] Hier spielen natürlich noch andere Faktoren herein, die jedoch unberücksichtigt bleiben müssen, da auch für die beiden anderen Algorithmen nichts näheres über die Testsituationen bekannt ist.

Wird nur die Aufgabenstellung dieser Arbeit betrachtet, so sind keine Probleme offengeblieben. Das größte Problem bei der Entdeckung minimaler FAen bleibt jedoch der Zeitaufwand für jede einzelne Datenbankanfrage, ob eine FA gilt. Dieser ist um einige Zehnerpotenzen höher als der Aufwand, jeweils eine neue Hypothese zu generieren. Denn das Problem besteht nicht in einer exponentiellen Anzahl der Hypothesen gemessen an der Anzahl der Attribute — heutige Computer sind leistungsfähig genug, diese auch für große Attributanzahlen aufzuzählen — sondern in dem zur Überprüfung der Hypothesen benötigten Zeitaufwand, der den eindeutig bestimmenden Part einnimmt. Dies zeigte im letzten Kapitel z. B. der Test mit der Datenbank Lymphography eindrucksvoll.

Andererseits sind Verfahren, wie z. B. jenes aus [Safnik und Flach, 1993], nicht praktikabel, wenn die Datenmenge entsprechend groß ist. Und in der Praxis verwendete Datenbanken haben jetzt schon diese Größe überschritten, und sie tendieren dazu, immer schneller anzuwachsen. Unter der Annahme, daß weiterhin FAen in realen Datenbanken gesucht werden sollen, ergibt sich hieraus die Schlußfolgerung, daß versucht werden muß, den Algorithmus FUNCTIONAL DEPENDENCIES weiter zu verbessern. Hiermit wird dann erreicht, daß die Grenze bezüglich der Anzahl der zu testenden Hypothesen, ab der der Algorithmus nicht mehr in vernünftiger Zeit zu einem Ergebnis kommt, weiter hinausgeschoben werden kann. Im nächsten Abschnitt werden unter anderem zwei Möglichkeiten hierzu aufgezeigt.

6.3 Verbesserungen

In diesem Abschnitt werden drei mögliche Erweiterungen dieser Arbeit kurz skizziert. Dies ist zum einen die Einführung einer Heuristik zur Bestimmung der unären Inklusionsabhängigkeiten. Zum anderen sind dies mögliche Erweiterungen des Algorithmus FUNCTIONAL DEPENDENCIES. Und drittens wird eine mögliche Abschwächung der Gültigkeit der funktionalen Abhängigkeiten aufgezeigt.

6.3.1 Unäre Inklusionsabhängigkeiten

Wenn die Namen für die Attribute in einer Datenbank festgelegt werden, so geschieht dieses nicht willkürlich. Jeder Attributname wird eine mnemotechnische Bedeutung haben. Dieses äußert sich zum Beispiel darin, daß die Spalten mit Personalnummern in Tabellen mit Mitarbeiter– und Managerdaten nicht mit MPNR und MGPNR abgekürzt werden, sondern beidesmal mit PERSONALNR.[2]

Diese Tatsache führt zu der Idee, bei der Bestimmung der Inklusionsabhängigkeiten mit gleichnamigen oder ähnlichen Attributen zu beginnen. Damit

[2]Abkürzungen sind erforderlich, da in der Regel das DBMS die Länge der Bezeichner für Attribute beschränkt.

ist dann die Hoffnung verbunden, die IAen möglichst in einer solchen Reihenfolge zu finden, so daß das Problem aus dem Beispiel 4.3 vermieden wird. Diese Lösung zusammen mit der dort beschriebenen Idee der Suche nach Ketten über Fremdschlüssel wäre noch effektiver.

6.3.2 Funktionale Abhängigkeiten

Betrachtet man noch einmal die beiden Abbildungen 4.4 und 4.5 aus Abschnitt 4.3.2, so leuchtet es ein, daß bei einer Wahl von Bottom–Up als primärer Suchstrategie diese Reduktion in ähnlicher Weise geschehen kann. Die Kanten in den Bäumen gehen dann von den speziellsten Hypothesen aus. Die resultierenden Eigenschaften dieser Bäume lassen sich ähnlich formulieren wie in Lemma 4.7.

Eine alleinige Bottom–Up Suche anstelle der Top–Down Suche bringt jedoch noch keine Vorteile, da funktionale Abhängigkeiten in der Regel eher „kurz als lang sind", was die Anzahl der Attribute auf der linken Seite betrifft. Für die Suche bedeutet dies, daß bei Top–Down die Abbruchkriterien eher erreicht werden.

Jedoch ist es sicherlich vielversprechend, beide Suchstrategien zu kombinieren. Dies sähe dann so aus, daß abwechselnd Top–Down oder Bottom–Up ebenenweise gesucht wird, je nachdem auf welcher Ebene weniger Hypothesen für weitere Tests verblieben sind.[3] Die Anzahl der Hypothesen zu berechnen, die aufgrund der Anwendung eines Schnittkriteriums „herausgeschnitten" werden, ist ein einfaches kombinatorisches Problem. Für diese abwechselnde Suche spricht auch die symmetrische Verteilung der FAen über die Ebenen, vgl. Abbildung 4.3.

Der Grund dafür, warum dies bisher nicht realisiert wurde, ist sehr einfach. Es ist noch wesentlich aufwendiger, den dann notwendigen Abgleich der Kombinationen der Attribute durchzuführen. Hier ist dann nicht nur wie in dem Algorithmus FUNCTIONAL DEPENDENCIES der Vergleich HAT–FA–ALS–TEILMENGE nötig, sondern zusätzlich ist auch noch ein Vergleich nötig, der überprüft, ob die zu testende Hypothese die Generalisierung einer nicht gültigen Hypothese ist, die in der Bottom–Up Phase getestet wurde.

Beispiel 6.1 *Angenommen in dem Halbverband aus Abbildung 4.4 gilt:* $ABCD \not\rightarrow F$ *und* $ABCE \not\rightarrow F$. *Dann kommen darin alle Attribute vor, die auf der linken Seite stehen können. Aber die Hypothesen* $ADE \rightarrow F$, $BDE \rightarrow F$ *und* $CDE \rightarrow F$ *sind damit noch nicht ausgeschlossen.* ◇

Das Beispiel zeigt, daß bei n Attributen nur bei der Gültigkeit von einstelligen und der Ungültigkeit von $(n-1)$–stelligen Hypothesen Attribute aus der Menge aller möglichen Attribute für die linke Seite von FAen entfernt

[3] Diese abwechselnde Suche wäre dann auch analog zu der in [Mitchell, 1982] beschriebenen Methode, den Versionenraum zu durchsuchen.

werden dürfen. Dies wird in dem Algorithmus FUNCTIONAL DEPENDENCIES in den Phasen Bottom–Up–Search–Start und Top–Down–Search–Start entsprechend ausgenutzt. Für alle anderen Hypothesen gilt, daß jeweils nur bestimmte Kombinationen der Attribute ausgeschlossen werden können.

Weiterhin wurde schon für HAT–FA–ALS–TEILMENGE gezeigt, daß dieser Test nur pseudo–polynomiell ist. Dies gilt auch für den dann notwendigen anderen Fall. Durch die Verwendung von „Buchhaltungsfunktionen“ für diese Kombinationen von Attributen kann der Algorithmus noch beträchtlich optimiert werden. Dieses muß einhergehen mit der Verwendung von Datenstrukturen, die die Durchführung dieser Vergleiche für große Mengen minimaler funktionaler Abhängigkeiten wesentlich effizienter unterstützen.

6.3.3 Wahrscheinlich korrekte funktionale Abhängigkeiten

Bisher wurde immer davon ausgegangen, daß eine funktionale Abhängigkeit gilt, oder eben nicht. Für letzteres genügt ein einziges Tupel, das die Ungültigkeit einer FA verursacht. Weiterhin wurden bisher zwei Probleme völlig außer acht gelassen, die bei existierenden, großen Datenbanken in der Regel akut sind. Dies sind zum einen fehlende und zum anderen falsche Daten, die bei der Dateneingabe verursacht wurden. Zwei mögliche Folgerungen hieraus sind die Nichtexistenz von FAen, oder daß die linken Seiten der FAen länger werden, als bei korrekten Daten.

Da nun von einem gewissen Fehler innerhalb der Datensätze ausgegangen werden kann, ist es nur konsequent, auch bei den entdeckten FAen einen gewissen Fehler zuzulassen, der in einer Wahrscheinlichkeit für die Gültigkeit der FAen ausgedrückt wird. Dieses führt zu „wahrscheinlich korrekten funktionalen Abhängigkeiten“.

Besteht nun ein Interesse daran, diese Änderung durchzuführen, dann läßt sich dieses mit dem hier vorgestellten Algorithmus sehr leicht bewerkstelligen. Hierzu sind zwei Dinge erforderlich. Zum einen muß einmal pro Tabelle die Gesamtzahl der Tupel gezählt werden. Diese Information fällt bei der Einteilung der Attribute in die vier Klassen als Nebenprodukt an, wenn ein Attribut in die Klasse SK kommt. Ansonsten wird einmal pro Relation eine entsprechende Anfrage gestellt.

Und zum zweiten muß die Auswertung der SQL–Anfrage auf Seite 49 geändert werden. Bisher werden die Zahlen a und b nur auf Gleichheit überprüft. Ist a gleich b, dann gilt die entsprechende FA. Angenommen a ist gleich 80 und b gleich 100. Dann existieren 20 Tupel, die zur Ungültigkeit der FA führen. Mit diesen Informationen können dann die entsprechenden Wahrscheinlichkeiten berechnet werden. Angenommen, es sind insgesammt auch 100 Tupel vorhanden, dann gilt die entsprechende FA zu 80%.

Der Vorteil dieser wahrscheinlich korrekten FAen sind die erheblichen Beschneidungen des Suchraumes, die sich ergeben. Es lassen sich leicht Beispiele konstruieren, in denen schon bei einer minimalen Abweichung der Wahrscheinlichkeit von 1, die Anzahl der Attribute auf den linken Seiten der FAen erheblich sinkt. Damit geht dann eine wesentliche Beschleunigung der Suche einher.

Anhang A

Beweis der Gleichung aus Lemma 4.5

Die Gültigkeit der folgenden Gleichung ist zu zeigen:

$$\sum_{i=1}^{n-1} \binom{n}{i} * (n-i) = n * (2^{n-1} - 1) \tag{A.1}$$

Beweis Es wird die folgende Gleichung benötigt

$$\sum_{i=0}^{n} \binom{n}{i} = 2^n \tag{A.2}$$

Die Gültigkeit dieser Formel kann über die Summation aller Elemente einer Zeile im Pascal'schen Dreieck hergeleitet werden. Sie befindet sich aber auch in jeder Formelsammlung. Damit in den nachfolgenden Formeln Definitionsprobleme vermieden werden, wird ohne Beschränkung der Allgemeinheit $n > 1$ vorausgesetzt. Es gilt

$$\binom{n}{i} = \frac{n!}{i! * (n-i)!} \tag{A.3}$$

Da n in der Summe in A.1 fest ist, kann A.3 auf die Situation in A.1 angewendet werden, und es gilt

$$\begin{aligned} \binom{n}{i} * (n-i) &= \frac{n! * (n-i)}{i! * (n-i)!} \\ &= \frac{n * (n-1)! * (n-i)}{i! * (n-i) * (n-i-1)!} \\ &= \frac{n * (n-1)!}{i!(n-i-1)!} \\ &= n * \binom{n-1}{i} \end{aligned} \tag{A.4}$$

A.4 angewendet auf A.1 ergibt

$$\begin{aligned}
\sum_{i=1}^{n-1} \binom{n}{i} * (n-i) &= \sum_{i=1}^{n-1} n * \binom{n-1}{i} \\
&= n * \sum_{i=1}^{n-1} \binom{n-1}{i} \\
&= n * \left(\sum_{i=0}^{n-1} \binom{n-1}{i} - \binom{n-1}{0} \right) \qquad (A.5)
\end{aligned}$$

Per Definition gilt

$$\binom{n-1}{0} = 1 \qquad (A.6)$$

Mit A.2 ergibt sich

$$\sum_{i=0}^{n-1} \binom{n-1}{i} = 2^{n-1} \qquad (A.7)$$

A.7 und A.6 eingesetzt in A.5 liefert somit

$$n * \left(\sum_{i=0}^{n-1} \binom{n-1}{i} - \binom{n-1}{0} \right) = n * \left(2^{n-1} - 1 \right)$$

und damit die Behauptung. $\square$

Abbildungsverzeichnis

Literaturverzeichnis

[Aho et al., 1983] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. Data Structures and Algorithms. Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[Armstrong, 1974] W. W. Armstrong. Dependency structures of database relationships. In *Proc. 1974 IFIP Congress*, S. 580 - 583, North Holland, Amsterdam.

[Bell, 1994] Siegfried Bell. Inferring Independencies. Erscheint als *Forschungsbericht 14*, Universität Dortmund, Fachbereich Informatik, Lehrstuhl VIII, 1994.

[Casanova et al., 1984] Marco A. Casanova, Ronald Fagin und Christos Papadimitriou. Inclusion Dependencies and Their Interaction with Functional Dependencies. In *Journal of Computer and System Sciences*, 28, S. 29 - 59, 1984.

[Codd, 1970] E. F. Codd. A relational model for large shared data banks. In *Comm. of the ACM*, 13:6, S. 377 - 387, 1970.

[Date, 1987] Christopher J. Date. A guide to the SQL standard, a users's guide to the standard relational language SQL. Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.

[Date, 1990] Christopher J. Date. An introduction to Database Systems, Band 1. Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

[Fuhr, 1991] Norbert Fuhr. Notizen zur 4-stündigen Stammvorlesung Informationssysteme. Fachbereich Informatik, Universität Dortmund, Sommersemester 1991.

[Holsheimer und Siebes, 1994] Marcel Holsheimer und Arno Siebes. Data Mining, The Search for Knowledge in Databases. In *Report CS-R9406, ISSN 0169-118X*, CWI, Amsterdam, 1994.

[Kanellakis et al., 1983] P. C. Kanellakis, S. S. Cosmadakis und M. Y. Vardi. Unary inclusion dependencies have polynomiel time inference problems. In *Proc. 15th ACM Symposium on Theory of Computing*, S. 264 - 277, 1983.

[Kanellakis, 1990] Paris C. Kanellakis. Elements of Relational Database Theory. In J. van Leeuwen, Hrsg., *Handbook of theoretical Computer Science*, Kap. 12, S. 1074 - 1156, Elsevier Science Publishers B. V. , 1990.

[Lockemann und Schmidt, 1987] P. C. Lockemann und J. W. Schmidt, (Hrsg.). Datenbank-Handbuch. Springer Verlag, Berlin Heidelberg, 1987.

[Mannila und Räihä, 1987] Heikki Mannila und Kari-Jouko Räihä. Dependecy Inference. In *Proceedings of the 13ths VLDB Conference*, S. 155 - 158, Brighton, 1987.

[Mannila und Räihä, 1992] Heikki Mannila und Kari-Jouko Räihä. The design of relational databases. Addison-Wesley Publishing Company, Reading, Massachusetts, 1992.

[Mitchell, 1982] Tom M. Mitchell. Generalisation as Search. In *Artificial Intelligence*, 18, S. 203 - 226, 1982.

[Morik, 1993] Katharina Morik. Maschinelles Lernen. In Günther Görz, Hrsg., *Einführung in die künstliche Intelligenz*, Kap. 3, S. 247 - 301, Addison-Wesley (Deutschland) GmbH, 1993.

[Piatetsky-Shapiro et al., 1991] Gregory Piatetsky-Shapiro, William Frawley und Christopher Matheus. Knowledge discovery in databases - an overview. In Gregory Piatetsky-Shapiro und William Frawley, Hrsg., *Knowledge Discovery in Databases*, S. 1 - 27, AAAI, AAAI Press., Menlo Parc, CA, 1991.

[Safnik und Flach, 1993] I. Savnik und P. Flach. Bottom-Up Induction of Functional Dependencies from Relations. In Gregory Piatetsky-Shapiro, Hrsg., *KDD-93: AAAI Workshop on Knowledge Discovery in Databases*, S. 174 - 185, 11 - 12 Juli, 1993, Washington, D. C., Technical Report WS-93-02.

[Schlimmer, 1993] J. Schlimmer. Using Learned Dependencies to Automatically Construct Sufficient and Sensible Editing Views. In Gregory Piatetsky-Shapiro, Hrsg., *KDD-93: AAAI Workshop on Knowledge Discovery in Databases*, S. 186 - 196, 11 - 12 Juli, 1993, Washington, D. C., Technical Report WS-93-02.

[Ullman, 1988] Jeffrey D. Ullman. Principles of Database and Knowledge-Base Systems. Band 1. Computer Science Press, 1988.