
Studienarbeit

Lehrstuhl Datenbanken und Informationssysteme

Sebastian W.H. Czymaj

30. Januar - 17. August 2004



Zusammenfassung

Anfragen an eine Datenbank werden nicht immer von jemandem ausgeführt, dem der Aufbau der Datenbank bekannt ist. Ein typisches Beispiel für eine solche Situation ist eine Anfrage an eine Web-Datenbank. Um dem Benutzer einer Anfrageschnittstelle dies auch nicht abzuverlangen, ist es notwendig, die vorliegenden Daten in eine Universalrelation zu transformieren. Ein System, das dies realisiert, muss die Zusammenhänge in der Datenbank kennen, und aus der Anfrage des Nutzers die notwendigen Verbundbedingungen automatisch generieren können.

Die vorliegende Arbeit beschäftigt sich mit dem Zusammentragen aller notwendigen Informationen aus einer Datenbank, die es einem Anfrageprozessor ermöglichen sollen, eine Universalrelation aus einer Datenbank zu erzeugen. In der vorliegenden Implementation ist ein Prototyp umgesetzt, der Informationen aus einer Datenbank sammelt und sie in einer XML-Datei dem Anfrageprozessor zur Verfügung stellt. Die einzelnen dabei auftretenden Probleme, wie zum Beispiel die Darstellung des Datenbankausschnittes und der darin geltenden Beziehungen, und deren Lösungen sind hier zusammengetragen und werden ausführlich diskutiert.

Inhaltsverzeichnis

Inhaltsverzeichnis	1
Abbildungsverzeichnis	4
1 Einleitung	5
2 Vorbetrachtungen	7
2.1 Aufgabenstellung	7
2.2 Arbeitsumgebung	7
2.3 Grundlagen	8
2.3.1 Die Universalrelation	8
2.3.2 Zyklen und maximale Objekte	8
2.3.3 Anwendungsarchitektur	10
2.4 Anforderungen an die Datenbank	11
3 Eingabe der Daten	13
3.1 Darstellung der Daten im Programm	13
3.2 Erfassung der Attribute durch den Nutzer	13
3.3 Beziehungen in der Datenbank	15
3.3.1 Cluster	15
3.3.2 Formalia über Beziehungen	16
3.3.3 Algorithmen zur Klassifizierung von Beziehungen	19
3.3.3.1 Typbestimmung	19
3.3.3.2 Kardinalitätsbestimmung	20
3.3.3.3 Probleme mit Unteranfragen	21

3.3.4	„Erraten“ von Beziehungen	22
3.3.4.1	Paare bestimmen	23
3.3.4.2	Inkompatible Paare eliminieren	28
3.3.4.3	Paare ordnen	29
3.3.4.4	Ein Paar bewerten	31
3.3.4.5	Resultatliste sortieren	31
3.3.5	Löschen und Überschreiben von Beziehungen	31
3.3.6	Beziehungen zwischen Duplikaten	32
3.4	Weitere Informationen für das Interface	33
3.4.1	Primärtabelle	33
3.4.2	Maximale Objekte	33
4	Erzeugung des Interfaces	36
4.1	Voraussetzungen	36
4.2	Die Schritte zum Aufbau des Interfaces	37
4.3	DTD des Interfaces	40
4.4	Beispiel-Interface	41
5	Implementierungs-Spezifika	45
5.1	Projektstruktur	45
5.2	GUI	45
5.3	Hilfsklassen	47
5.4	Sammeln von Metadaten	48
5.5	Weitere datenbankspezifische Komponenten	49
5.6	Konfiguration	49
5.7	Sicherung des Arbeitszustandes	50
5.8	Interface	51
5.9	Logging	52
6	Fazit und Ausblick	54
7	Quellenangaben	55
7.1	Hardware	55

7.2 Software	55
Literaturverzeichnis	57

Abbildungsverzeichnis

2.1	Hypergraph zum „Banking Example“	9
2.2	Gesamtarchitektur	10
3.1	Darstellung der Datenbank und des Interfaces im Programm	14
3.2	Beziehungserfassung im Programm	16
3.3	Eingabe einer Beziehung	23
3.4	Erzeugung disjunkter Beziehungen, und deren Darstellung im Interface	32
3.5	Darstellung der Primärtabelle im Programm	33
3.6	Erfassung maximaler Objekte	34
5.1	Programmstruktur	46
5.2	Das Hauptfenster	47
5.3	Logging an der Konsole	52
5.4	Historie des Statusmeldungen	53

1. Einleitung

Der Benutzer eines Datenbanksystems hat beim Formulieren einer Anfrage im wesentlichen drei Dinge anzugeben, nämlich **welche** Daten er haben möchte, **woher** er diese Daten bekommt, und welchen **Bedingungen** die Daten unterliegen müssen. Speziell die Angabe woher die Daten kommen erfordern genaue Kenntnisse über den Aufbau der Datenbank, da in der Regel *Verbundbedingungen* angegeben werden müssen, die die Daten aus mehreren Tabellen in der Anfrage zusammenfügen und daraus das Ergebnis erstellen.

Datenbanksysteme eignen sich aber offensichtlich nicht für Anwendungen, in denen Daten den Nutzer in allgemeiner Form zur Verfügung gestellt werden sollen, also der Nutzer lediglich von der Existenz der Daten ausgehen kann, die Verteilung der Daten auf die jeweiligen Relationen jedoch dem Nutzer nicht zugänglich (und genauso wenig bekannt) ist. Ein Beispiel dafür wäre eine Anfrageschnittstelle für ein Bibliothekssystem, bei denen der Nutzer die Organisation der Bibliotheksdaten (Bestand, Ausleihdaten, etc.) nicht kennt und auch nicht kennen muss, aber dennoch Anfragen wie zum Beispiel `SELECT ISBN WHERE AUSLEIHER=' . . . '` möglich sind.

Für solche Szenarios benötigt man ein System, das in der Lage ist, die Verbundbedingungen automatisch aus der Anfrage des Nutzers zu generieren. Ein solcher Anfrageprozessor benötigt Informationen über die Tabellen in der Datenbank, und die Beziehungen zwischen den Tabellen. Er muss wissen, welche Tabellen im Verbund die gewünschten Daten liefern. Ein Problem, das dabei auftritt, ist die deterministische Anfrageumsetzung für den Fall, dass mehrere Tabellen im Verbund das gewünschte Ergebnis liefern. Dieses Problem wird in Abschnitt 2.3.2 aufgegriffen und genauer erläutert.

Dem Nutzer steht dann eine *Universalrelation* zur Verfügung, eine (nicht zwangsweise physisch erzeugte) Tabelle, die alle Daten der Datenbank enthält, und somit Verbundbedingungen überflüssig macht.

Diese Studienarbeit beschäftigt sich mit der Erfassung aller Daten, die für einen Anfrageprozessor nötig sind um eine Universalrelationensicht zu erzeugen. Ziel der Studienarbeit war es, eine prototypische Implementierung zu schreiben, mit der die Beschreibung einer Datenbank erzeugt werden kann. Diese Beschreibung wird in einer XML-Datei gespeichert, und wird in diesem Text auch abkürzend *Interface* genannt.

Der Text ist thematisch wie folgt gegliedert:

Das Kapitel 2 legt einige Grundlagen im Zusammenhang mit dem Projekt, wie beispielsweise das bereits oben erwähnte Problem mit verschiedenen Verbundmöglichkeiten, liefert aber auch

allgemeine Informationen über das Programm und die konkrete Aufgabenstellung.

In Kapitel 3 wird beschrieben wie die Informationen über die Datenbank erfasst und aufgearbeitet werden. Dies beinhaltet neben der Erfassung des *Datenbankfensters* (der in der Anfrage sichtbare Datenbankausschnitt) auch die Aufarbeitung von Beziehungen zwischen den Daten. Die dabei auftretenden Probleme werden in diesem Kapitel ausführlich analysiert.

Im Kapitel 4 wird die Umwandlung der erfassten Daten in eine für den Anfrageprozessor brauchbare Form behandelt. Dies umfasst Probleme wie beispielsweise das automatische Hinzufügen derjenigen Attribute, die später zum Verbund der Tabellen notwendig sind.

Im Kapitel 5 werden die wichtigsten Programmkomponenten vorgestellt, und deren Implementierung etwas näher betrachtet.

Das Kapitel 6 wird ein kurzes Fazit aus der Arbeit ziehen, und einen Ausblick auf eine mögliche Weiterentwicklung liefern. In Kapitel 7 sind abschließend alle verwendeten Quellen aufgeführt.

2. Vorbetrachtungen

2.1 Aufgabenstellung

Erstellung Universalrelationenschnittstelle für Web-Datenbanken - Gewinnung der erforderlichen Schemainformationen und Verbundbedingungen

Betreuer: Gunnar Weber

Charakter: Implementierung/Konzeption

1. Realisierung mit Java/JDBC
2. Auslesen der Relationen, Attribute und Schlüsseldefinitionen (Primärschlüssel, Fremdschlüssel)
3. Graphische Darstellung der gewonnenen Schemainformationen mit der Möglichkeit der manuellen Bearbeitung (Festlegung des benötigten Datenbankausschnitts, der notwendigen Projektionsattribute und gewünschten Selektionsattribute)
4. Ableitung der Beziehungen durch Fremdschlüsselbedingungen bzw. gleichbenannte Attribute (mit Domänenanalyse zur Sicherstellung der gleichen Rolle des Attributs in den beteiligten Relationen)
5. Gewinnung von Informationen aus Beispielanfragen
6. Ermittlung der benötigten Verbundoperationen für Universalrelationen-Anfragen mit OR-Operatoren: natürlicher Verbund (kein Auftreten von Dangling Tupeln) oder OUTER JOIN (left, right, full)
7. Abspeicherung der gewonnen Informationen in XML für Anfragebearbeitung

2.2 Arbeitsumgebung

Als Programmiersprache kommt *Java 2 (SE)* unter Verwendung von *SWING*-Klassen zum Einsatz. Java bietet außerdem Unterstützung für Datenbankzugriffe mittels *JDBC*, sowie Möglich-

keiten zur Erzeugung von XML-Dokumenten mittels *JDOM*.

Das Projekt wurde auf folgenden Datenbanksystemen (erfolgreich) getestet: IBM DB2 7.2.7 (Personal Edition), IBM DB2 8.1.6 (Enterprise Edition) und MySQL 4.0.17.

Zu den einzelnen Programmen, die an der Entwicklung beteiligt waren, ist in Kapitel 7 eine kurze Beschreibung angegeben.

Der Name des Projektes lautet URE, das steht für „Universalrelationenanfrageschnittstelle Teil Eins“.

2.3 Grundlagen

2.3.1 Die Universalrelation

Anfragen im relationalen Datenmodell, beispielsweise mit SQL, erfordern die Angabe mindestens einer Tabelle, aus der die Daten kommen. Weiterhin müssen vom Nutzer Daten, die auf mehrere Tabellen verteilt sind, mittels Verbund zusammengeführt werden.

Unter bestimmten Voraussetzungen können die Verbundbedingungen auch automatisch bestimmt werden. Der Benutzer arbeitet dann auf einer Universalrelation, eine Tabelle, die alle Datensätze der Datenbank in sich vereint. Natürlich enthält diese Tabelle die Daten in einer hochredundanten Form, andererseits muss die Tabelle aber nicht physisch erzeugt werden, es genügt schließlich die vom Nutzer angeforderten Daten zu liefern, und dies umfasst in der Regel nicht die gesamte Datenbank.

In der Universalrelation kann dann auf die Angabe der Herkunft der Daten verzichtet werden. SQL-Anfragen kommen so beispielsweise ohne die FROM-Klausel aus. Damit dies möglich ist, muss die Datenbank so entworfen sein, dass alle Daten verlustfrei wiederhergestellt werden können, das heißt eine JD (*join dependency*) über alle Relationenschemata einer Datenbank gilt.

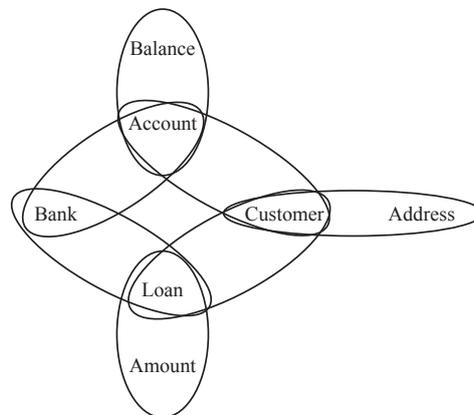
2.3.2 Zyklen und maximale Objekte

Wie bereits in der Einführung festgestellt wurde, kann es bei der Anfrage zu Problemen kommen, wenn es für die Ausführung eines Verbundes zwischen zwei Tabellen mehrere Möglichkeiten gibt. Dieses Problem geht auf *Zyklen* innerhalb der Datenbank zurück. Um den Begriff des Zyklus zu erläutern möchte ich mich des folgenden Beispiels aus [Ull82] bedienen:

Gegeben ist eine Datenbank mit den Attributen Bank, Account, Loan, Customer, Amount, Balance und Address. Die Beziehungen zwischen diesen Attributen wird im Hypergraphen in Bild 2.1 deutlich.

Wenn nun jede Kante dieses Hypergraphen eine Relation darstellt, welche mit Fremdschlüssel-Beziehungen untereinander logisch verknüpft sind, so ergibt sich ein Zyklus in der Datenbank. Ein Zyklus weist auf ein semantisch überladenes Attribut hin. Es ist nicht immer eindeutig wel-

Abbildung 2.1: Hypergraph zum „Banking Example“



ches Attribut überladen ist, da dies oft von der Betrachtungsweise abhängt. In unserem Beispiel sei das Attribut Customer als überladen angesehen, da es sich hierbei um Kunden (Customer) handelt, die sowohl ein Konto (Account), als auch ein Kredit (Loan) haben. Anfragen liefern hierbei unter Umständen falsche bzw. nicht erwartete Ergebnisse.

Betrachten wir beispielsweise die folgende Anfrage:

```
SELECT Bank WHERE Customer='Jones'
```

Wir erwarten alle Banken, bei denen Jones entweder ein Konto, oder einen Kredit hat, oder beides. Der Anfrageprozessor hat nun mehrere Möglichkeiten:

1. Verbund von Customer und Bank über Account
2. Verbund über Loan
3. Verbund über Account und Loan
4. Erst Verbund über Account, dann Verbund über Loan, anschließend Vereinigung (UNION) der beiden Ergebnisse

Es ist nicht vorhersagbar welches Ergebnis zustande kommt. Nutzt der Anfrageprozessor eine der Varianten 1 bis 3, so kommt in jedem Fall ein für uns unerwartetes Ergebnis zustande. Um das erwartete Ergebnis zu erzeugen, muss die Variante 4 ausgeführt werden.

Die Problematik zeigt recht deutlich, dass Zyklen die Anfrage negativ beeinflussen können. Um dem Anfrageprozessor die Entscheidung zu erleichtern eignen sich *maximale Objekte*. Ein maximales Objekt ist eine Menge von Objekten, über die wir höchstens bereit sind zu navigieren. Die Objekte in einem maximalen Objekt sind Ausschnitte von Tabellen, das heißt eine Teilmenge von Attributen aus einer Relation in der Datenbank. Ein maximales Objekt ist azyklisch, wodurch die Anfrage innerhalb eines maximalen Objektes eindeutig wird. Werden maximale

Objekte auf einer Datenbank definiert, so müssen alle Objekte in mindestens einem maximalen Objekt enthalten sein.

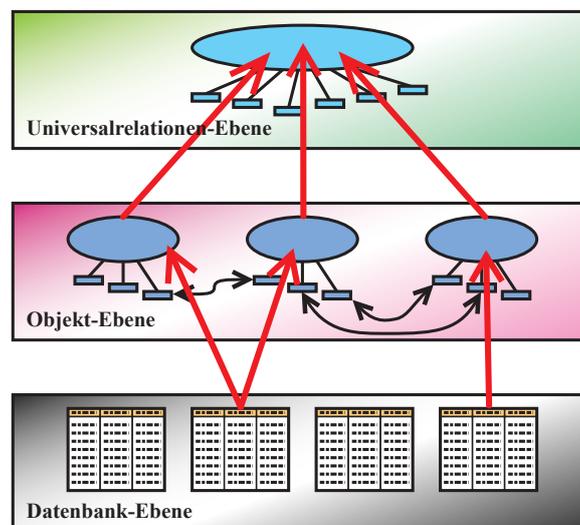
In unserem Beispiel wären zwei maximale Objekte zu definieren: $M_{oben} = (\{Bank\}, \{Account, Balance\}, \{Customer, Address\})$, $M_{unten} = (\{Bank\}, \{Customer, Address\}, \{Loan, Amount\})$. Wie wir sehen, können einige Objekte in mehreren maximalen Objekten enthalten sein. Der Anfrageprozessor kann diese maximalen Objekte nun benutzen um seine Anfrage eindeutig zu formulieren. Dazu ermittelt er alle an der Anfrage beteiligten maximalen Objekte, führt erst die Anfrage in jedem maximalen Objekt einzeln und anschließend die Vereinigung auf den Ergebnissen durch.

2.3.3 Anwendungsarchitektur

Das Projekt URE ist der erste Teil einer zweiteiligen Anwendung. URE bereitet die Daten vor, die später von einem Anfrageprozessor zur Realisierung von Anfragen an eine Universalrelation benötigt werden.

Die Gesamtarchitektur (URE+Anfrageprozessor) ist in Abbildung 2.2 dargestellt.

Abbildung 2.2: Gesamtarchitektur



Der Prozess der Datenverarbeitung ist dabei in verschiedene Ebenen untergliedert:

- Die unterste Ebene ist die Datenbankebene, auf der Tabellen, Attribute und Integritätsbedingungen wie Schlüssel und Fremdschlüssel bekannt sind.
- Daraus wird eine Objektebene erzeugt, auf der Objekte existieren, die wiederum Attribute haben und in Beziehung miteinander stehen. Objekte repräsentieren eine Relation in der

Datenbank, nur das Objekte nicht alle Attribute der zugehörigen Tabelle enthalten müssen. Die Objektschicht stellt den Datenbankausschnitt dar, der vom Nutzer festgelegt wird. Die Objektschicht wird außerdem um Informationen angereichert, die notwendig sind um die Universalrelationenanfrage zu ermöglichen.

- Aus den Objekten wird dann auf oberster Ebene eine Universalrelation erzeugt. Diese muss nicht wirklich manifestiert sein, aber die Applikation oder der Nutzer, der mit dieser Schicht arbeitet, soll genauso wie auf einer Universalrelation arbeiten. Es ist die Aufgabe des Anfrageprozessors dies effizient und korrekt umzusetzen.

Das Projekt URE beschäftigt sich mit der Transformation der Datenbankinformationen hin zur Objektschicht.

2.4 Anforderungen an die Datenbank

Für die Gewinnung der notwendigen Schemainformationen sind folgende Informationen aus der Datenbank notwendig:

- Schemata für die Tabellen (optional)
- Tabellen der Datenbank
- Attribute der Tabellen
- Primärschlüssel- und Fremdschlüsselbeziehungen
- Indizes (optional)

Alle Informationen können direkt mittels JDBC abgefragt werden. Voraussetzung ist, dass der Treiber folgende Methoden des `java.sql.DatabaseMetaData`-Objekts (sowie dessen Erzeugung) unterstützt:

- `getSchemas()` - gibt alle in der Datenbank enthaltenen Schemata zurück
- `getTables(...)` - gibt die in der Datenbank enthaltenen Tabellen zurück
- `getColumns(...)` - gibt zu einer Tabelle die Attribute zurück
- `getPrimaryKeys(...)` - gibt zu einer Tabelle alle Primärschlüssel zurück
- `getImportedKeys(...)` - gibt zu einer Tabelle alle Fremdschlüssel (importierte Schlüssel) zurück
- `getIndexInfo(...)` - liefert Informationen über Indizes zu einer Relation und somit über die UNIQUE-Eigenschaft einiger Spalten.

Alle verwendeten Methoden und Klassen sind in der Spezifikation von JDBC 1.0 enthalten. Sollte eine Datenbank zum Einsatz kommen, dessen JDBC-Schnittstelle nicht zur Version 1.0 des Standards konform ist, so müssen zumindest die hier angegebenen Funktionen unterstützt werden, da sonst mit Fehlern bei der Informationsgewinnung zu rechnen ist.

URE setzt ebenfalls voraus, dass alle Primärschlüssel in der Datenbank auch als solche definiert wurden. Sollte der JDBC-Treiber für eine Tabelle keine Primärschlüssel zurückgeben, so wird implizit angenommen, dass alle Spalten zusammen Primärschlüssel sind.

Weiterhin wird vorausgesetzt, dass die Datenbank so entworfen wurde, dass alle relevanten Daten in einer Universalrelation dargestellt werden können, d.h. alle relevanten Tabellen haben eine Beziehung zueinander. Ein Interface, bei dem Beziehungen fehlen, ist nicht gültig und kann mit diesem Programm nicht generiert werden.

Bei der Ermittlung der notwendigen Metadaten werden an einigen Stellen ineinander geschachtelte Anfragen benutzt. Sollte die verwendete Datenbank (z.B. MySQL in Versionen vor 4.1) solche Anfragen nicht unterstützen, ist dies dem Programm mitzuteilen, da dann alternative Anfragen genutzt werden müssen. Dadurch müssen mehr Daten von der Datenbank angefragt werden, wodurch sich die übertragene Datenmenge erhöht.

3. Eingabe der Daten

3.1 Darstellung der Daten im Programm

Bevor wir uns mit der Erfassung der Daten beschäftigen möchte ich zunächst die Darstellung der Daten vorstellen. Die Bearbeitung soll grafisch erfolgen, weshalb ich mich für SWING zur Darstellung entschieden habe. SWING ist ein Paket aus der Java-Entwicklungsumgebung, das Komponenten zur Herstellung einer grafischen Benutzeroberfläche bietet.

Der Nutzer sollte sowohl die Ursprungs- als auch die Zieldaten stets im Blick haben. Bei den Ursprungsdaten handelt es sich um die Datenbank mit ihren Tabellen und Attributen. Die Zieldaten sind die dem Interface hinzugefügten Attribute und die daraus resultierenden Objekte.

Die Datenbank-Informationen werden in einem Baum dargestellt, die Wurzel bildet die Datenbank selbst, darunter sind die Schemata angeordnet, auf der zweiten Ebene befinden sich die Tabellen, und die Attribute sind darunter als Knoten angehängt. Im Baum sind die Schlüsseigenschaften der Attribute grafisch dargestellt. In Abbildung 3.1 (links) ist ein entsprechender Screenshot aus dem Programm zu finden.

Die hinzugefügten Attribute werden in einer (alphabetisch sortierten) Liste dargestellt, auf Abruf wird die zugehörige Tabelle angezeigt.

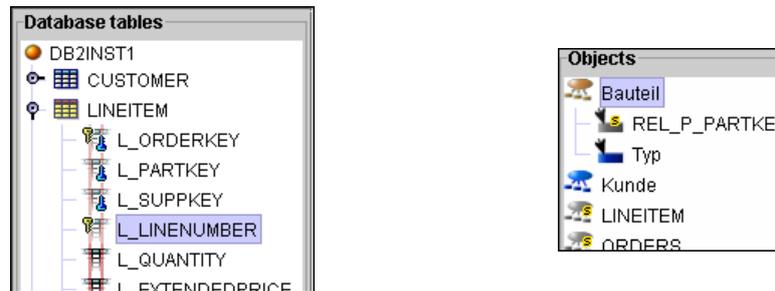
Die Objekte werden in einem Baum angezeigt, mit Interface als Wurzel, den Objekten auf der ersten Ebene, und den Attributen darunter auf der zweiten Ebene. In diesem Baum werden vom System hinzugefügte Attribute und Objekte gesondert gekennzeichnet. Dies wird im Programm wie in Abbildung 3.1 (rechts) dargestellt.

3.2 Erfassung der Attribute durch den Nutzer

Der Nutzer soll lediglich angeben, welche Attribute er dem Interface hinzufügen möchte. Anschließend soll das Programm daraus eigenständig Objekte aufbauen.

Beim Prozess der Datenerfassung ist der wohl zeitaufwändigste Teil die Auswahl der Attribute, die aus der Datenbank ausgewählt und einzeln hinzugefügt werden. Damit legt der Nutzer gleichzeitig das Datenbankfenster (den Ausschnitt der Datenbank) fest, das dann für Anfragen zur Verfügung steht.

Abbildung 3.1: Darstellung der Datenbank und des Interfaces im Programm



Dazu kommt die Angabe, ob es sich beim hinzuzufügenden Attribut um ein Projektions- oder Selektionsattribut handeln wird. Projektionsattribute sind diejenigen, auf die in der Anfrage projiziert wird. Selektionsattribute sind die Attribute, nach denen selektiert wird, das heißt die in der WHERE-Klausel vorkommen. Attribute können beides, aber auch keines von beiden sein. Diese Informationen werden in URE nicht verwendet, sondern nur gesammelt, um sie dann im Interface festzuhalten. Mit Hilfe dieser Angaben sind im Anfrageprozessor Optimierungen möglich, indem er beispielsweise mit einer Indizierung über den Selektionsattributen die Anfragezeit verbessert.

Ein besonderes Augenmerk ist dabei auf Attribute zu richten, die im Interface mehrmals hinzugefügt werden. Diese dienen dazu mittels Selbstverbund spezielle Anfragen zu ermöglichen. Folgendes Beispiel zeigt einen solchen Selbstverbund:

Beispiel 3.2.1: Beispiel einer Anfrage mit Selbstverbund

Das Relationenschema R sei $\{\text{Buch}, \text{Autor}\}$. Um Anfragen wie **SELECT** Buch **WHERE** Autor1='A' **AND** Autor2='B' zu ermöglichen, müssen zwei Objekte erstellt werden: $O_1 = \{\text{Buch}, \text{Autor1}\}$, $O_2 = \{\text{Buch}, \text{Autor2}\}$. Der Verbund beider Tabellen kann über das Attribut Buch erfolgen.

Beim Hinzufügen eines Attributs kann das Programm feststellen, ob das Tabellen-Attribut bereits im Interface referenziert wird, denn für jede Tabelle wird gespeichert wieviele Objekte aus dieser Tabelle erzeugt wurden. Sind es mehrere Objekte, so muss der Nutzer entscheiden zu welchem Objekt das aktuell hinzuzufügende Attribut gehört, falls dies nicht automatisch festgestellt werden kann.

Weiterhin hat der Nutzer die Möglichkeit, hinzugefügte Attribute wieder zu löschen. Falls dabei leere Objekte entstehen, so werden diese automatisch entfernt.

3.3 Beziehungen in der Datenbank

Dieser Abschnitt behandelt Probleme, die im Zusammenhang mit Beziehungen auftreten können. Dies umfasst die Klärung des Begriffs *Cluster*, der als Hilfsmittel dient um fehlende Beziehungen zu finden, sowie eine formale Definition von Beziehungen. Anschließend werde ich Algorithmen diskutieren, die Beziehungen bestimmen, und mit denen fehlende Beziehungen automatisch gefunden werden können. Abschließend werde ich noch einige Überlegungen zum Löschen von Beziehungen nennen, sowie die Probleme *Duplikaten* analysieren. Dabei handelt es sich um mehrere Objekte, die der selben Relation r in der Datenbank zugeordnet sind. Diese Objekte werden dann als *Duplikat bezüglich der Relation r* bezeichnet.

3.3.1 Cluster

Um herauszufinden wo in der Datenbank Beziehungen fehlen, ist es sinnvoll Tabellen zu clustern. In einem Cluster werden alle Tabellen zusammengefasst, die eine Beziehung zu einer anderen Tabelle in diesem Cluster haben. Existiert mehr als ein Cluster in einer Datenbank, so gibt es isolierte Bereiche von Tabellen, die mit dem Rest der Datenbank (über Fremdschlüssel) nicht in Beziehung stehen.

Um Cluster zu ermitteln wende ich folgenden Algorithmus an:

1. Gegeben ist eine Menge von Tabellen und eine Menge von Beziehungen.
2. Für jede Tabelle wird ein eigener Cluster erzeugt. Bei n Tabellen existieren (zunächst) n verschiedene Cluster.
3. Starte eine Schleife, die über alle nicht-disjunkten Beziehungen iteriert:
 - (a) Extrahiere die Tabellen der aktuellen Beziehung.
 - (b) Finde die zu den Tabellen zugehörigen Cluster.
 - (c) Verschmelze die beiden Cluster zu einem einzigen.
 - (d) Falls weitere Beziehungen existieren: mit nächster Beziehung zurück zum Schleifenanfang, sonst: die Schleife beenden
4. Übrig bleibt ein Clustering aller Tabellen. Die Liste der Cluster wird absteigend nach Größe des Clusters sortiert.

Die ermittelten Cluster werden nummeriert, der Cluster, der die Primärtabelle enthält, erhält die Nummer 1, alle anderen werden der Größe nach durchnummeriert.

Bei der Anzeige im Programm wird hinter der Nummer des Clusters noch eine der Tabellen angezeigt. Dies dient der besseren Übersicht. Die angezeigte Tabelle ist diejenige mit dem lexicographisch kleinsten Namen, die im Cluster enthalten ist. Dies ist wird in Abbildung 3.2 deutlich.

Der Nutzer muss nun dafür sorgen, dass alle Attribute, die im Interface enthalten sein sollen, im selben Cluster liegen, indem er fehlende Beziehungen ergänzt.

Abbildung 3.2: Beziehungserfassung im Programm

3.3.2 Formalia über Beziehungen

Neben den Beziehungen, die in der Datenbank als Fremdschlüssel definiert sind, gibt es noch Beziehungen, die zum Beispiel mehrwertig sind, oder nicht durch Fremdschlüsselbedingungen ausgedrückt wurden, sondern deren Integrität durch andere Mechanismen sichergestellt wird. Für den Anfrageprozessor ist nur wichtig, **wie** er Tabellen miteinander verbinden muss, und ob es dabei Tupel gibt, die aus dem Ergebnis herausfallen. Daher kann eine Beziehung formal folgendermaßen beschrieben werden:

- Eine Beziehung hat zwei **Seiten**, eine **linke** und eine **rechte**. Die zwei Seiten der Beziehung seien die Relationenschemata R_1 und R_2 .
- Da nicht alle Attribute der beiden Seiten an der Beziehung teilnehmen, fassen wir die relevanten Attribute in einer **Menge von 2-Tupeln** der Form (l, r) zusammen, wobei $l \in R_1$, und $r \in R_2$. Es sei angenommen, dass die Attribute eindeutig ihren Relationen zugeordnet werden können. Durch diese Annahme ist eine explizite Speicherung der beiden Seiten der Beziehung überflüssig. Die Menge dieser 2-Tupel sei *paare*.

- Eine Beziehung hat einen **Typ** *typ*, der die Qualität der Beziehung widerspiegeln soll, indem er angibt, ob und wo die Tupel einer Seite nicht im Verbund enthalten sind. Möglich sind die Werte „C-C“, „C-P“, „P-C“, „P-P“ und „disjunkt“. Die einzelnen Beziehungstypen werden nachfolgend in diesem Abschnitt genauer betrachtet.
- Eine Beziehung hat eine **Kardinalität** *kard*, wobei *kard* folgende Werte annehmen kann: „1:1“, „1:N“, „N:1“ oder „N:N“. Wie die Kardinalitäten zustandekommen wird ebenfalls weiter unten genauer erklärt.

Die Notation einer Beziehung erfolgt als 3-Tupel: (*paare, typ, kard*).

Die Typen einer Beziehungen haben folgende Bedeutungen, wobei C abkürzend für *Complete* und P für *Partial* steht:

- **C-C - Beziehung**

Diese Beziehung besteht zwischen 2 Tabellen, bei deren Verbund alle Tupel einen Verbundpartner bekommen. Das bedeutet es gibt in keiner Tabelle Tupel, die nicht verknüpft werden können.

- **C-P bzw. P-C - Beziehung**

Wenn beim Join in genau einer der beiden beteiligten Tabellen Tupel übrig bleiben, für die kein Verbundpartner gefunden werden kann (sogenannte *dangling tuples*), so wird diese Beziehung als einseitig vollständig klassifiziert. Hierauf kann der Anfrageprozessor mit einem LEFT beziehungsweise RIGHT OUTER JOIN reagieren.

- **P-P - Beziehung**

Falls es in beiden beim Verbund beteiligten Tabellen Tupel gibt, die nicht verbunden werden können, so wird die Beziehung zwischen den Tabellen als P-P klassifiziert. Um alle Tupel beider Seiten ins Ergebnis zu bekommen, muss der Anfrageprozessor einen FULL OUTER JOIN durchführen.

- **Disjunkte Beziehung**

Ist der Verbund leer, obwohl sich auf beiden Seiten Tupel befinden, so ist die Beziehung disjunkt. Solche Beziehungen haben für die Anfrage keine Bedeutung und sollten im Interface nicht vorkommen, sind allerdings auch nicht verboten.

Um Beziehungen vollständig zu charakterisieren, fehlt noch die Angaben der Kardinalität. Da Kardinalitäten in verschiedenen Modellen (zum Beispiel beim Entity Relation-, Erweiterten Entity Relation-Modell, UML, ...), oft unterschiedlich angegeben werden, möchte ich an dieser Stelle festhalten, wie die Kardinalitäten im Programm notiert werden.

Prinzipiell unterscheidet man bei Beziehungen zwischen „zu 1“- und „zu N“- Beziehungen. In einigen Modellen (zum Beispiel UML) werden auch Maximalwerte für die zu N-Beziehung angegeben. Oftmals kommt auch der Aspekt der *Optionalität* in geeigneter Form zum Ausdruck. Um all dies nun umzusetzen, habe ich mich für folgende Notation der Kardinalität entschieden:

In Kardinalitäten kommen nur 1 und N in Frage. Die Angabe eines Maximums für N ist nicht erforderlich, da sie für die Anfrage später keinen Nutzen darstellt.

Der Aspekt der Optionalität spiegelt sich bereits in den Beziehungstypen wider. So ist eine Seite mit Beziehungstyp C immer zwingend an einer Beziehung beteiligt und eine Seite mit Typ P stets optional.

Die Reihenfolge in der Notation (1:N oder N:1?) möchte ich anhand des folgenden Beispiels verdeutlichen:

Beispiel 3.3.1: über Kardinalitäten

Gegeben sind folgende Relationen:

Links:	A	B	Rechts:	C	D
	1	aaa		1	xxx
	2	bbb		2	yyy
	3	ccc		2	zzz

Die Beziehung zwischen den Relationen Links und Rechts ist dann folgende:
 ({ (A,C) }, P-C, N:1)

Die Kardinalität N:1 ergibt sich folgendermaßen: Es gibt Tupel in Links, die beim natürlichen Verbund mehrere Verbundpartner in Rechts finden, nämlich alle Tupel mit A=2. Somit gehen die Tupel aus Links mit N Tupeln aus Rechts Beziehungen ein. Umgekehrt stehen die Tupel aus Rechts stets nur mit jeweils einem Tupel aus Links in Beziehung, daher die Kardinalität 1 für die rechte Seite.

Fremdschlüssel können ebenfalls in dieses Schema von Beziehungen eingeordnet werden:

Ein Fremdschlüssel ist generell eine C-P-Beziehung. Eine P-P-Beziehung kann es nicht werden, da dies durch die Definition eines Fremdschlüssels nicht möglich ist.

Es kann also nur noch eine C-C-Beziehung sein, was sich durch statistische Auswertung des Datenbankinhaltes feststellen lässt.

Die Kardinalität eines Fremdschlüssels lässt sich ebenfalls voraussagen. Da ein Fremdschlüssel, der in einer Relation R_1 definiert ist, in einer anderen Relation R_2 Schlüssel ist, kann es sich entweder um eine 1:N-Beziehung handeln, oder im Spezialfall auch um eine 1:1-Beziehung, was sich ebenfalls durch statistische Auswertungen ermitteln lässt (oder durch den Nutzer angegeben werden kann).

Damit die Herkunft einer Beziehung aber nicht vollständig verloren geht, wird im Interface zusätzlich der Ursprung der Beziehung gespeichert. Möglich sind dabei folgende Werte:

- Statistik - Diese Beziehung wurde durch Auswertung des Datenbankinhalts bestimmt.
- Nutzer - Diese Beziehung wurde vom Nutzer eingegeben.
- Fremdschlüssel - Diese Beziehung ist eine Fremdschlüsselbeziehung in der Datenbank.

- Fremdschlüssel/Statistik - Diese Beziehung war ein Fremdschlüssel, der durch Auswertung des Datenbankinhalts verbessert wurde (zum Beispiel zu einer C-C und/oder 1:1-Beziehung).

3.3.3 Algorithmen zur Klassifizierung von Beziehungen

Stehen die beiden beteiligten Relationen und Attribute einer Beziehung fest, dann können Kardinalität und Typ der Beziehung automatisch mit Hilfe des Datenbankinhaltes bestimmt werden.

3.3.3.1 Typbestimmung

Um den Beziehungstyp zu bestimmen werden verschiedene Datenbankabfragen ausgewertet. Die Beziehung sei hierbei das Tupel $(paare, t, k)$, wobei alle Attribute l jedes Paares (l, r) in der Menge *Links*, und alle Attribute r in der Menge *Rechts* enthalten sein sollen: $Links = \{l | (l, r) \in paare\}$, *Rechts* analog. Aus den Anfragen ergeben sich die (ganzzahligen) Werte CNT_L, CNT_R und CNT_JOIN, deren Auswertung anschließend den Typ liefern. Um die nachfolgenden Ausdrücke etwas zu vereinfachen sei die linke Tabelle der Beziehung als *L*, und die rechte Tabelle als *R* hinter dem FROM notiert. Ein hinter der SELECT-Anweisung auftretendes *Links* und/oder *Rechts* bedeutet, das die Elemente aus *Links* (beziehungsweise *Rechts*) hier als Attributliste nach SQL-Syntax notiert werden. Das Auftreten von *verbund* in der WHERE-Klausel bedeutet eine AND-Verkettung der Verbundbedingungen aus der Menge *paare* der Beziehung in SQL-Syntax.

1. **CNT_L**: Das Ergebnis folgender Anfrage gibt an, wie viele verschiedene Werte es in der linken Attributmenge gibt:
`sql:= SELECT DISTINCT Links FROM L;`
`CNT_L:= SELECT COUNT(*) FROM sql AS COUNTER;`
2. **CNT_R**: Die Anfrage ist analog zu CNT_L, nur für die rechte/n Tabelle/Attribute statt der linken.
3. **CNT_JOIN**: Dieser Wert zählt die eindeutigen Tupel, die im Verbund enthalten sind.
`sql:= SELECT DISTINCT Links FROM L, R WHERE verbund;`
`CNT_JOIN:= SELECT COUNT(*) FROM sql AS COUNTER;`

Ist einer der beiden Werte CNT_L und/oder CNT_R gleich CNT_JOIN, dann bekommt diese Seite ein „C“ in ihrer Beziehung, andernfalls ein „P“. Sollte sich herausstellen, dass der Verbund leer ist (obwohl die CNT_x jeweils > 0 sind), so wird diese Beziehung als disjunkt gekennzeichnet.

Falls sich der Beziehungstyp nicht bestimmen lässt, zum Beispiel aufgrund eines Datenbankfehlers, oder weil eine der Datenbanktabellen leer ist, wird der Typ auf „unbekannt“ gesetzt. Unbekannte Beziehungen spielen nur im Programm eine Rolle, dürfen aber im Interface nicht enthalten sein.

3.3.3.2 Kardinalitätsbestimmung

Um die Kardinalität einer Seite der Beziehung zu ermitteln, bestimmen wir zusätzlich zu den Werten aus der Typbestimmung noch die Werte **CNT_ALL_L** und **CNT_ALL_R**. Zur Bestimmung der Kardinalität ist es stets notwendig die jeweils andere Seite zu untersuchen. Der Übersichtlichkeit halber ist die Vorgehensweise hier nur für die linke Seite dargestellt. Die rechte Seite bestimmt sich analog, nur das statt **CNT_ALL_R** der Wert **CNT_ALL_L** berechnet und ausgewertet wird.

Es gibt bei der Bestimmung zwei Fälle zu unterscheiden:

Fall 1: Die rechte Seite hat Beziehungstyp C

Fall 2: Die rechte Seite hat Beziehungstyp P

Zu Fall 1

Aus der Ermittlung des Beziehungstyps steht uns bereits der Wert **CNT_JOIN** zur Verfügung. Da es sich um eine Typ C-Beziehung handelt, wissen wir bereits, dass alle Tupel aus *L* einen Verbundpartner haben. Angenommen es existieren in *R* mehr Tupel als **CNT_JOIN**, so bedeutet das, dass einige Tupel aus *L* mehr als einen Verbundpartner haben. Der dafür notwendige Wert wird ähnlich wie **CNT_R** berechnet, allerdings diesmal nicht mit **SELECT DISTINCT**, sondern nur mit **SELECT**:

```
CNT_ALL_R: SELECT COUNT(*) FROM R;
```

Ist **CNT_ALL_R** = **CNT_JOIN**, so ist die Kardinalität der linken Seite 1, ansonsten ist sie N.

Zu Fall 2

Wie auch im ersten Fall bestimmt sich die Kardinalität durch den Vergleich einer **SELECT DISTINCT** mit einer **SELECT**-Anfrage, nur das wir im Falle des Beziehungstyps P erst noch eine Anfrage zur Bestimmung des Vergleichswertes durchführen müssen und nicht wie oben diesen bereits gegeben haben.

Wir wissen die rechte Seite hat Typ P, jetzt müssen wir herausfinden wie viele eindeutige Tupel davon in die Beziehung eingehen. Dies ist aber bereits in **CNT_JOIN** enthalten. Das bedeutet, dass von den **CNT_R** Tupeln aus *R* nur **CNT_JOIN** eindeutige Tupel in der Beziehung stecken. Um diese Tupel geht es uns, denn sollten dort Duplikate auftreten, so ist klar, dass diese dazu führen, dass mindestens ein Tupel aus *L* mehrere Verbundpartner bekommt. Der Wert **CNT_ALL_R** für diesen Fall bestimmt sich also wie folgt:

```
sql:= SELECT Rechts FROM L, R WHERE verbund;
```

```
CNT_ALL_R:= SELECT COUNT(*) FROM R WHERE (Rechts) IN sql.
```

CNT_ALL_R wird dann wie in Fall 1 mit **CNT_JOIN** verglichen und entsprechend ausgewertet.

Folgendes Beispiel soll die Beziehungsbestimmung verdeutlichen:

Beispiel 3.3.2: Bestimmung der Beziehung

Gegeben seien folgende zwei Relationen:

ALBEN = {ALBUMNO, TITEL} auf der linken Seite und **COVERS** = {ALBUM, PIC} auf der rechten mit

ALBEN:	ALBUMNO	TITEL
	1	Play
	2	The Unforgettable Fire
	3	Reptile

COVERS:	ALBUM	PIC
	1	PLAY.JPG
	2	UFIRE.JPG

CNT_L = 3, denn in ALBEN sind 3 verschiedene Werte von ALBUMNO.

CNT_R = 2, denn in COVERS sind 2 verschiedene Werte von ALBUM.

CNT_JOIN = 2, denn im natürlichen Verbund der beiden Relationen über die entsprechenden Attribute sind 2 verschiedene Tupel enthalten.

Daraus ergibt sich bereits der Beziehungstyp P-C.

CNT_ALL_L = 2, (Betrachtung nach Fall 1:) denn in COVERS sind insgesamt 2 Werte (inklusive Duplikate) von ALBUM.

CNT_ALL_R = 2, (Betrachtung nach Fall 2:) denn bei Projektion auf ALBUMNO im Verbund ergeben sich insgesamt 2 Werte (inklusive Duplikate).

Die Kardinalität ist somit 1:1.

Als Beziehung ergibt sich ((ALBEN.ALBUMNO,COVERS.ALBUM), P-C, 1:1).

3.3.3.3 Probleme mit Unteranfragen

Wie bereits erwähnt gab es beim Test mit MySQL Probleme mit diesen Anfragen, da MySQL in der mir vorliegenden Version noch keine Unteranfragen unterstützt. Da aber gerade im Web-Umfeld MySQL sich gewisser Beliebtheit erfreut, habe ich alternative Anfragen erstellt.

Allerdings sind nicht alle Anfragen äquivalent ohne Unteranfragen zu formulieren sind, daher ist es hier notwendig die äußere Anfrage im Programm, und nicht auf der Datenbank durchzuführen. Es wird nur die innere Anfrage an die Datenbank abgesendet, das COUNT wird stattdessen durch eine Schleife im Programm ersetzt. Auf eine Verwendung von temporären Tabellen habe ich verzichtet, da ich nicht voraussetzen kann, dass der Nutzer dafür immer die notwendigen Rechte hat. Das bedeutet allerdings auch, dass mehr Daten von der Datenbank geliefert werden müssen, was bei langsamen Verbindungen durchaus zu spürbaren Wartezeiten führt.

Dies sind die angepassten Anfragen:

- CNT_L: Für eine einelementige Menge *Links* ist folgende Anfrage möglich:
 CNT_L := **SELECT** COUNT(DISTINCT *Links*) **FROM** *L*;
 Da COUNT(DISTINCT . . .) mit Attributlisten nicht funktioniert, muss das Zählen manuell erfolgen falls in *Links* mehr als ein Element enthalten ist. Die Anwendung setzt dann nur die folgende Anfrage ab, und zählt die Tupel im Ergebnis selbst:
SELECT DISTINCT *Links* **FROM** *L*;
- CNT_R: analog zu CNT_L

- CNT_JOIN: Ein ähnliches Problem, das auch mit CNT_L bestand, tritt bei der Bestimmung von CNT_JOIN auf. Die folgende Anfrage ist nur mit einem Attribut in *Links* möglich:

```
CNT_JOIN := SELECT COUNT(DISTINCT Links)
           FROM L, R WHERE verbund;
```

Als Alternative für Attributlisten bleibt auch hier nur das manuelle Auszählen des Ergebnisses folgender Anfrage:

```
SELECT DISTINCT Links FROM L, R WHERE verbund;
```

- CNT_ALL_R (Fall 1): enthielt keine Unteranfrage;
- CNT_ALL_R (Fall 2): Zunächst wird die folgende Anfrage abgesetzt:

```
SELECT DISTINCT Rechts FROM L, R WHERE verbund;
```

Damit ist nun eine Liste aller Tupel in *R* verfügbar, die im Verbund mit *L* einen Partner haben. Dieses Ergebnis wird gespeichert. Da allerdings die Werte von *Links* und *Rechts* im Verbund gleich sind, entspricht dies der Anfrage, die für CNT_JOIN formuliert wurde. Anstatt also zur Berechnung von CNT_ALL_L die selbe Anfrage ein zweites mal zu stellen, speichern wir zwecks Optimierung das Ergebnis der Anfrage für CNT_JOIN bereits im Vorfeld und greifen hier darauf zurück.

Anschließend wird die folgende Anfrage abgesetzt:

```
SELECT Rechts FROM R;
```

Dies sind alle Tupel aus *R*. Schneidet man diese Ergebnismenge mit dem Ergebnis der vorhergehenden Anfrage, und zählt dann die Tupel im Durchschnitt, dann ergibt sich der gesuchte Wert.

- CNT_ALL_L: analog zu CNT_ALL_R

Für CNT_L und CNT_JOIN sind die Anfragen für einelementige Attributlisten kürzer als die ursprünglichen Anfragen. Das Programm verwendet deshalb für einelementige *Links* bzw. *Rechts* immer die oben angegebenen Anfragen und vermeidet Unteranfragen mittels manuellem Auszählen nur dann, wenn es wirklich notwendig ist.

3.3.4 „Erraten“ von Beziehungen

Nun kann es vorkommen, dass Beziehungen fehlen, und der Nutzer hat nicht die Möglichkeit diese nachzupflegen. Das Programm kann jedoch versuchen, die fehlenden Beziehungen zu ermitteln. Dazu bedient sich das Programm Informationen aus der Datenbank, aus denen dann die Beziehungen bestimmt werden. Ich bezeichne diese Domänenanalyse zur Beziehungsermittlung auch als *Statistik-Funktionen*, da die Auswertung statistischer Natur ist.

Guess

Das Problem der Beziehungsfindung kann in mehrere Einzelprobleme aufgeteilt werden, die auch in getrennten Methoden implementiert sind. Im Programm verbirgt sich die Statistik-Funktionalität hinter einer einzigen Schaltfläche. Diese wurde bereits in Abbildung 3.2 dargestellt. Sie trägt die Bezeichnung „Guess“. In Abbildung 3.2 dient sie dazu, alle Beziehungen zwischen den

selektierten Clustern anzuzeigen. In Abbildung 3.3 ist die Eingabe einer Beziehung dargestellt, die ebenfalls eine solche Schaltfläche enthält. Durch Betätigung versucht das Programm mittels Statistik-Funktionen die fehlenden Daten zu ergänzen. Hat der Nutzer beispielsweise keine Attribute zur Beziehung hinzugefügt, so wird eine Beziehung zwischen den gewählten Tabellen gesucht. Sind bereits Attribute enthalten, so wird versucht Typ und Kardinalität zu bestimmen.

Abbildung 3.3: Eingabe einer Beziehung

Cluster 1 (ACCOUNTS,...)		Cluster 1 (ACCOUNTS,...)	
ACCOUNTS		BANK	
ACCOUNTS.ACCOUNT (DECIMAL(12,0)) [PK]		BANK.NAME (CHAR(50)) [PK]	
ACCOUNTS.BALANCE (DECIMAL(12,2))			
ACCOUNTS.BANKNAME (CHAR(50)) [FK:BANK.N.]			

Buttons: Guess, Add Pair

Left: BANKNAME;
 Right: NAME;
 Type: Unknown
 Cardinality: Unknown

Buttons: Add, Abort

In den folgenden Abschnitten werden nun alle Einzelprobleme, die Teil der Beziehungsfindung sind, erläutert.

3.3.4.1 Paare bestimmen

Zunächst wird es nötig sein, eine Kandidatenliste zu erstellen, die später auf Beziehung überprüft wird. Die Methode hierfür soll zwei Attributmengen als Eingabe nutzen, um eine Menge möglicher Paare für Beziehungen zu erstellen. Es werden alle Attribute der ersten Menge mit allen Attributen der zweiten Menge in Beziehung gebracht, innerhalb der jeweiligen Attributmengen wird keine Beziehung aufgebaut.

Ein Algorithmus, der das Problem löst, hat als Eingabeparameter die zwei Attributmengen L und R (das steht für „Links“ und „Rechts“). Das können beispielsweise alle Attribute zweier Datenbankcluster sein. Die Eingabemengen sollten nach Möglichkeit disjunkt sein. Sind sie es nicht, so macht dies für den endgültigen Algorithmus allerdings keinen Unterschied, die Ergebnismenge wird nur etwas größer. Das liegt daran, dass nun innerhalb einer Menge Beziehungen gesucht werden müssen, und die Trennung in zwei Mengen nicht mehr ausgenutzt werden kann.

Der intuitivste Algorithmus, der das Problem nach einer „brute force“-Methodik löst, hat nun mehrere ineinander verschachtelte Schleifen:

Algorithmus 1:

1. Die äußerste Schleife iteriert i durch $1..min(|L|, |R|)$. In dieser Schleife wird jede i -elementige Teilmenge von L bestimmt.
2. Die aktuell betrachtete Teilmenge von L sei im folgenden als L^* bezeichnet. Innerhalb der äußersten Schleife iteriert eine nächste Schleife über alle ebenfalls i -elementigen Teilmengen von R .
3. Die aktuell betrachtete Teilmenge von R sei im folgenden als R^* bezeichnet. Innerhalb dieser Schleife iteriert eine weitere Schleife über alle Permutationen von L^* .
4. Innerhalb dieser Schleife iteriert eine letzte Schleife über alle Permutationen von R^* , nimmt die Permutation von L^* aus der übergeordneten Schleife und schreibt diese zusammen mit der Permutation von R^* in eine Ergebnismenge E .

Die Ergebnismenge E dieses Algorithmus wird extrem groß, da hier alle Permutationen der Potenzmenge, also alle Variationen von L und R betrachtet werden müssen. Die Formel für die Größe der Ergebnismenge E ergibt sich folgendermaßen:

Die Anzahl aller Permutationen einer n -elementigen Menge ist $n!$. Die Anzahl aller k -elementigen Teilmengen einer n -elementigen Menge ist $\binom{n}{k}$. Es sei $m = min(|L|, |R|)$. Zu bestimmen sind zunächst alle Teilmengen von L , dann von R , da aber nur maximal m Attribute in der Beziehung stecken können, vereinfacht sich die Formel. Da wir jede (nicht leere) bis zu m -elementige Teilmenge suchen, ergibt sich daraus:

$$\sum_{n=1}^m \binom{|L|}{n}$$

Weil wir aber auch alle Permutationen für die jeweilige n -elementige Teilmenge suchen, erhöht sich die Anzahl auf

$$\sum_{n=1}^m \left(\binom{|L|}{n} \cdot n! \right)$$

Da aus jeder Permutation einer n -elementigen Teilmenge aus L mit jeder n -elementigen aus R ein Paar gebildet werden muss, ergibt sich daraus folgende Formel für die Größe der Ergebnismenge $|E|$:

$$|E| = \sum_{n=1}^{min(|L|, |R|)} \left(\binom{|L|}{n} \cdot n! \cdot \binom{|R|}{n} \cdot n! \right)$$

Für $|L| = 10$ und $|R| = 12$ ergibt sich daraus beispielsweise $|E| = 1.197.530.884.927.200$.

Prinzipiell kann jedes Paar eine Beziehung sein, aber der oben genannte Algorithmus ist zu ineffizient. Es werden offenbar viel zu viele Paare unnötig und mehrfach geprüft. Das Ergebnis wird extrem groß, was daran liegt, dass die Menge aller Teilmengen sehr hoch ist, und zudem

auch noch in Kombination für zwei Attributmengen durchsucht wird. Wir benötigen also einen Algorithmus, der nur das nötigste an Mengen zur Paarberechnung benutzt:

Algorithmus 2:

1. Wir sammeln zunächst Informationen über kompatible Partner zu jedem Attribut:
 - Für jedes Attribut $X \in L$ wird eine Menge kompatibler Attribute erzeugt.
 - Eine Funktion $komp$ ordnet dafür jedem Attribut X aus L eine Menge von Attributen aus R zu: $komp : L \rightarrow 2^R$.
 - Ist $komp(X)$ leer, so kann X aus L entfernt werden, denn X kann nicht Teil einer Seite einer Beziehung sein, da es auf der anderen Seite keinen Verbundpartner gibt.
2. Für alle X aus der reduzierten Menge L wird mit jedem $Y \in komp(X)$ eine Beziehung R mit $R = (\{(X, Y)\}, t, k)$ erzeugt. Zur Erinnerung: Beziehungen haben die Form $(paare, typ, kard)$. Anschließend wird t bestimmt. Stellt sich heraus, dass die Beziehung disjunkt oder unbestimmbar ist, so wird Y der Menge D_X hinzugefügt (D_X steht für „Disjunkt in Beziehung mit X “). Nicht disjunkte Beziehungen bilden die Mengen $E_{\{X\}}$, die später ein Teil des Ergebnisses sind. Dies erspart uns die erneute Bestimmung der elementigen Paare.
3. Wir bilden für jedes Attribut $X \in L$ die Mengen $P_X = komp(X) - D_X$ (P steht hierbei für Partner). Attribute X , für die P_X leer ist werden aus L entfernt, denn wenn diese X in Beziehungen auftauchen, so finden diese keinen Verbundpartner und die gesamte Beziehung wird disjunkt. In P_X sind nun alle denkbaren Partner für eine Beziehung mit X enthalten.
4.
 - Wir betrachten nun alle möglichen Teilmengen $L^* \in 2^L - \emptyset$ (alle Elemente der Potenzmenge).
 - Es sei $|L^*| = k$ und $L^* = \{X_1, \dots, X_k\}$, wobei nur L^* mit $k \geq 2$ betrachtet werden, denn einelementige Teilmengen haben wir ja bereits bestimmt.
 - Ein Paar $(X, Y)_p$ sei nun definiert als 2-Tupel (X, Y) mit den Eigenschaften $X \in L$, $Y \in P_X$.
 - Die Ergebnismenge E_{L^*} für ein spezielles L^* ist nun folgende:

$$E_{L^*} = \{((X_1, p_1)_p, \dots, (X_k, p_k)_p)\}$$

5. Die Vereinigung aller E_{L^*} ist die gesuchte Gesamtergebnismenge $E = \{E_{L^*} | L^* \in 2^L - \emptyset\}$.

Dieser Algorithmus sollte nun – vorausgesetzt es steht nicht wirklich jedes Attribut aus L mit jedem in R in Beziehung, denn in diesem Fall wäre das Resultat identisch zu dem von Algorithmus 1 – in brauchbarer Zeit eine Liste aller Kandidaten erstellen, denn er eliminiert alle Attribute, die nicht für eine Beziehung in Frage kommen. Die Ergebnisgröße bei diesem Algorithmus lässt

sich allerdings nur schwer exakt abschätzen, da er stark von den vorliegenden Daten abhängig ist. Angenommen jedes Attribut aus L findet nur wenige potentielle Partner, im Durchschnitt k , so berechnet sich $|E|$ nach der folgenden Formel:

$$|E| = \sum_{n=1}^{\min(|L|, |R|)} \left(\binom{|L|}{n} \cdot (n \cdot k) \right)$$

Für obiges Beispiel mit $|L| = 12$, und $k = 5$ ist $|E| = 12800$. Angenommen, es fallen aus L zusätzlich noch die Hälfte aller Attribute heraus, da sie keinen Verbundpartner bekommen, so verringert sich das Ergebnis sogar auf $|E| = 400$ Paare, die es dann zu testen gibt, zuzüglich der 120 Beziehungen (von denen einige allerdings schon im Ergebnis enthalten sind) die getestet werden mussten um die Kompatibilität der Attribute festzustellen. Dies ist nicht nur eine wesentliche Verbesserung gegenüber dem obigen Beispiel, sondern auch eine Größe, die nicht den Hauptspeicher einer aktuellen Workstation zum Überlauf bringt. Angesichts der Überlauf-Problematik ist es sinnvoll, die maximale Anzahl von Attributen in einer Beziehung (nutzerdefiniert) zu limitieren.

Folgendes Beispiel soll die verschiedene Arbeitsweise der Algorithmen 1 und 2 demonstrieren:

Beispiel 3.3.3: Beispiel zu den Algorithmen 1 und 2

Die Mengen L und R seien folgende: $L = \{\text{Name, Vorname, GebDat}\}$, $R = \{\text{Name, TelNo}\}$. Name und Vorname sind jeweils vom Typ CHAR, GebDat (Geburtsdatum) vom Typ DATE, TelNo vom Typ DECIMAL. Alle Elemente aus L kommen aus der Relation R_1 , alle Elemente aus R aus einer anderen Relation R_2 .

Da nun die Elemente der jeweiligen Mengen aus einer Relation stammen geht es darum, eine Beziehung zwischen R_1 und R_2 zu finden. Bisher wurde noch nicht beachtet, dass bei Beziehungen alle Attribute aus der selben Tabelle stammen müssen. Wir werden dies in einem weiteren Ausbau unseres Algorithmus 2 berücksichtigen.

Der **Algorithmus 1** würde auf folgende Weise Kandidaten für eine Beziehung suchen:

Das Minimum $m = \min(|L|, |R|)$ ist 2. Es müssen also maximal 2-elementige Teilmengen gesucht werden. Eine 3-elementige Menge würde keinen Sinn machen, da dann aus L ein Element doppelt in der Beziehung vorkommen würde.

Die Teilmengen aus L sind $\{\text{Name}\}$, $\{\text{Vorname}\}$, $\{\text{GebDat}\}$, $\{\text{Name, Vorname}\}$, $\{\text{Name, GebDat}\}$, $\{\text{Vorname, GebDat}\}$. Durch Bildung der Permutationen entstehen dadurch folgende 9 Listen: (Name), (Vorname), (GebDat), (Name, Vorname), (Vorname, Name), (Name, GebDat), (GebDat, Name), (Vorname, GebDat), (GebDat, Vorname).

Die Teilmengen aus R sind $\{\text{Name}\}$, $\{\text{TelNo}\}$, $\{\text{Name, TelNo}\}$. Durch Bildung der Permutationen ergeben sich diese 4 Listen: (Name), (TelNo), (Name, TelNo), (TelNo, Name).

Aus den 1-elementigen Listen beider Seiten lassen sich insgesamt 9 Paare bilden, aus den 2-elementigen Listen lassen sich 12 Paare bilden, insgesamt sind es 18 mögliche Paare.

Algorithmus 2 arbeitet folgendermaßen:

Zunächst suchen wir für jedes Attribut aus L die kompatiblen Partner in R . Dazu wird eine Beziehung $(paare, t, k)$ mit jeder möglichen einelementigen Menge $paare$ der Form (l, r) mit $l \in L$ und $r \in R$ erstellt. Für jede Beziehung wird anschließend t bestimmt. Ist t weder disjunkt noch unbestimmbar, so wird r als potentieller Partner für l gespeichert. Findet ein l keinen kompatiblen Partner, so wird es aus L entfernt.

$l \in L$	kompatibel	nicht kompatibel
Name	Name	TelNo
Vorname		Name, TelNo
GebDat		Name, TelNo

Name in L und Name in R sind kompatibel, da der Verbund über dieses Attribut nicht leer ist. Vorname in L und Name in R sind nicht kompatibel, da der Verbund leer war. Der Verbund von GebDat in L mit Name in R ist ebenfalls leer, usw. Hier ist jedoch zu erkennen, dass der Verbund von Gebdat mit Name in R nicht hätte durchgeführt werden müssen, da durch die unterschiedlichen Typen bereits im Vorfeld klar gewesen wäre, dass hier kein Verbund möglich war. Dadurch hätte die Datenbank mit der Bestimmung dieser Beziehung nicht beauftragt werden müssen. Auch diese Überlegung wird in den nächsten Algorithmus mit einfließen.

Hieraus ergibt sich nun, dass Vorname und GebDat aus L entfernt werden können, so dass das neue L nun die folgende Menge ist: $L = \{R_1.Name\}$.

Die Menge der Partner ist $P_{R_1.Name} = \{R_2.Name\}$.

Im nächsten Schritt werden wieder alle Permutationen aller Teilmengen von L gebildet, in diesem Fall ist es nur eine: (Name).

Es werden nun alle Möglichkeiten gesucht, die Permutation der linken Seite mit kompatiblen Attributen der rechten Seite zu verbinden. In diesem Fall ist die einzige Beziehung, die getestet werden muss $(\{(R_1.Name, R_2.Name)\}, t, k)$. Die Ergebnismenge enthält einen möglichen Kandidaten für eine Beziehung, wobei jedoch im Vorfeld dafür $|L| \cdot |R|$ Beziehungen, in diesem Falle 6 Beziehungen, getestet wurden, um die Ergebnismenge zu reduzieren. Es ist jedoch schon bei diesem kleinen Beispiel deutlich, dass dadurch weniger Beziehungen getestet werden müssen. Bei größeren Eingabemengen macht sich dies noch wesentlich deutlicher bemerkbar.

Der Algorithmus 2 ist allerdings noch nicht vollständig, wie wir im Beispiel gesehen haben. Die Ursache für die Explosion der Ergebnisgröße ist in der hohen Anzahl der möglichen Teilmengen der Attributmeng L zu finden. Wir haben aber bisher nicht beachtet, dass alle Attributmengen einer Menge $L^* \subseteq L$ aus der selben Tabelle kommen müssen, da es schließlich darum geht Tabellen miteinander zu verknüpfen. Genauso müssen die rechten Attribute eines jeden Attributpaares einer Beziehung aus E_{L^*} aus der selben Tabelle stammen.

Berücksichtigen wir dies, so wird sich die Ergebnismenge E wiederum drastisch reduzieren. Voraussetzung ist, dass zu den Attributen auch die Tabellennamen an den Algorithmus übergeben werden. Formal ist dies aber durch die Definition der Beziehung abgesichert, da jedes Attribut seiner Relation zugeordnet werden kann (in der Praxis dadurch zugesichert, dass ein Attribut als $R.X$ notiert werden kann, wobei R die zu X gehörende Relation ist).

Der folgende Algorithmus korrigiert alle Nachlässigkeiten und berücksichtigt bereits eine möglicherweise angegebene Maximalanzahl von Attributen in einer Beziehung. Ansonsten sind die

wesentlichen Schritte identisch zu denen aus Algorithmus 2.

Algorithmus 3:

1. Wir sammeln zunächst Informationen über kompatible Partner zu jedem Attribut. Für jedes Attribut $X \in L$ wird eine Menge kompatibler Attribute (siehe dazu auch nächster Abschnitt) mittels der in Algorithmus 2 eingeführten Funktion $komp$ erstellt. Aus L werden alle Attribute X mit $komp(X) = \emptyset$ entfernt.
2. Für alle X im reduzierten L mit jedem $Y \in komp(X)$ eine Beziehung R mit $R = (\{(X, Y)\}, t, k)$ erzeugt und t wird bestimmt. Stellt sich bei der Bestimmung von t heraus, dass die Beziehung disjunkt oder unbestimmbar ist, so wird Y der Menge D_X hinzugefügt. Nicht disjunkte R werden als E_X in die Ergebnismenge E übernommen und müssen nicht erneut bestimmt werden.
3. Wir bilden für jedes Attribut die Mengen $P_X = komp(X) - D_X$.
4.
 - Wir betrachten nun alle möglichen maximal m -elementigen aus einer Tabelle stammenden Teilmengen L^* mit:
 $L^* \in \{M | M \in 2^L, 2 \leq |M| \leq m, \nexists X, Y \in M : tbl(X) \neq tbl(Y)\}$
 L^* enthält somit alle maximal m -elementigen Elemente der Potenzmenge, wobei m im Vorfeld vom Nutzer festgelegt wurde. Alle Elemente in L^* stammen aus der selben Tabelle.
 - Dabei liefert die Funktion $tbl(X)$ die Tabelle zum Attribut X .
 - Es sei $|L^*| = k$ und $L^* = \{X_1, \dots, X_k\}$. Wir betrachten nur L^* mit $k \geq 2$.
 - Ein Paar $(X, Y)_p$ sei definiert als 2-Tupel (X, Y) mit den Eigenschaften $X \in L$ und $Y \in P_X$.
 - Die Ergebnismenge E_{L^*} für ein spezielles L^* ist nun folgende:

$$E_{L^*} = \{((X_1, p_1)_p, \dots, (X_k, p_k)_p) | \forall i, j \in \{1..k\} : X_i \in L^* \wedge \nexists i \neq j : tbl(p_i) \neq tbl(p_j)\}$$
5. Die Vereinigung aller E_{L^*} ist die gesuchte Gesamtergebnismenge $E = \{E_{L^*} | L^* \in 2^L\}$.

3.3.4.2 Inkompatible Paare eliminieren

Es macht nur Sinn Paare zu behalten, deren Typen kompatibel sind. Da einige Typen als Schlüssel nur sehr bedingt oder nicht sinnvoll sind, werden die folgenden Datentypen generell bei der Prüfung ausgelassen, das heißt Attribute diesen Typs werden nicht als Kandidaten für Paare ins Ergebnis übernommen:

ARRAY, BINARY, BIT, BLOB, BOOLEAN, CLOB, DATALINK, DISTINCT, JAVAOBJECT, LONGVARBINARY, LONGVARCHAR, NULL, OTHER, REF, STRUCT, VARBINARY.

In der folgenden Tabelle sind nun alle relevanten Datentypen mit ihren kompatiblen Zieltypen aufgeführt. Als kompatibel wird ein Ursprungstyp angesehen, wenn er sich in einen Zieltyp

umwandeln lässt. Da die Datentypen mit sich selbst kompatibel sind, werden in der Tabelle 3.1 nur andere Typen als der ursprüngliche Typ betrachtet.

Einige Typen lassen sich nur mit Einschränkungen umwandeln, zum Beispiel INTEGER in SMALLINT, da der Wertebereich zu klein ist. Eine Beziehung zwischen zwei Tabellen ist aber dennoch denkbar, nur dass die zustandekommende Beziehung in der Regel nicht vom Typ C-x ist. Solche eingeschränkten Kompatibilitäten werden, soweit sie als sinnvoll erscheinen, daher ebenfalls berücksichtigt, und sind in Tabelle 3.1 mit einem * gekennzeichnet.

Tabelle 3.1: Übersicht über Kompatible Datentypen

SQL-Typ		kompatible Zieltypen
TINYINT	8 bit	SMALLINT*, INTEGER*, BIGINT*
SMALLINT	16 bit	TINYINT, INTEGER*, BIGINT*
INTEGER	32 bit	TINYINT, SMALLINT, BIGINT*
BIGINT	64 bit	TINYINT, SMALLINT, INTEGER
DECIMAL/NUMERIC	Fixed	REAL, DOUBLE, FLOAT
REAL	Single Prec.	DOUBLE*, FLOAT(* falls Precision zu hoch)
DOUBLE	Double Prec.	REAL, FLOAT
FLOAT	Single/Double	REAL, DOUBLE(* falls Single Precision)
CHAR	Zeichen	VARCHAR(* falls kürzer spezifiziert), TINYINT, SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, DOUBLE, FLOAT
VARCHAR	Zeichen	CHAR(* falls kürzer spezifiziert), TINYINT, SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, DOUBLE, FLOAT
DATE	Datum	TIMESTAMP
TIME	Zeit	TIMESTAMP
TIMESTAMP	Datum + Uhrzeit	keine

Die Kompatibilitätstabelle ist lediglich ein Mittel zur Vorsortierung, mit dem inkompatible Paare aussortiert werden sollen, und somit die Datenbank weniger Ergebnisse testen muss. Sollte die darunterliegende Datenbank eine implizite Typkonvertierung nach obiger Tabelle nicht unterstützen, so wird zur Laufzeit ein Fehler (Exception) auftreten. Tritt dieser Fehler auf, so werden die der Funktion übergebenen Typen ebenfalls als inkompatibel betrachtet. Aus Effizienzgründen wird diese Typprüfung bereits in den Algorithmus 3 in die Funktion *komp* bei der Paarbestimmung integriert.

3.3.4.3 Paare ordnen

Die bisher unsortierte Liste von Paaren muss sortiert werden, damit Kandidaten, die besonders wahrscheinlich sind, zuerst geprüft werden. Der Hintergrund dieser Sortierung ist folgender: die

Suche nach Beziehungen soll (optional) vorzeitig zu beenden sein, falls eine vom Nutzer festgelegte Anzahl geeigneter Beziehungen gefunden wurde. Nach dem die Liste der Paare erstellt wurde, muss sie abgearbeitet werden, wobei jedes Paar geprüft werden muss. Es ist daher von Vorteil, wenn die existierenden Bedingungen oben in der Liste stehen, da somit nicht unnötig disjunkte Beziehungen getestet werden.

Um die Sortierung zu vereinfachen, habe ich ein Punktesystem eingeführt. Jede Beziehung erhält zunächst die Punktzahl 0. Erfüllt die Beziehung ein bestimmtes Kriterium, so wird die Punktzahl entsprechend erhöht. Es ist so auch sehr einfach möglich die Punktevergabe zu erweitern, da lediglich nacheinander Punkte verteilt werden müssen, und somit weitere Kriterien einfach integriert werden können. Es ist auch möglich die Punktzahl jedes Kriteriums zu ändern, um besonders wichtige Kriterien hervorzuheben.

Zu den Kriterien für die Punktevergabe gehören folgende:

- Eine Seite oder Teile davon sind Primärschlüssel oder ein Teil davon. Primärschlüssel sind eindeutige Attribute, sie sich daher hervorragend als Teil einer Beziehung eignen und dementsprechend hoch qualifiziert werden müssen. Da das Vorhandensein von Primärschlüsseln vorausgesetzt wird kann dieses Kriterium stets angewendet werden.
- Attribute, die als eindeutig (UNIQUE) in der Datenbank definiert wurden, sind als potentielle Verbundattribute zu betrachten und werden dementsprechend bewertet. Leider ist es mit JDBC nicht möglich zu ermitteln, welche Spalten als eindeutig definiert sind. Somit ist der Umweg über die Indizes (unter der Annahme, dass das Datenbanksystem einen Index zur effizienten Prüfung der UNIQUE-Eigenschaft anlegt) der einzige sinnvolle Umweg um diese Information zu ermitteln. Die Alternative wäre, für jede Spalte ein `SELECT COUNT(*)` mit einem `SELECT DISTINCT COUNT(*)` zu vergleichen, aber dies ist gerade für große Datenbanken viel zu aufwändig um sinnvoll zu sein, und wurde daher nicht umgesetzt.
- Fängt ein Attributname einer Seite wie der entsprechende auf der anderen Seite an, oder endet er gleich, oder sind beide Attributnamen identisch, so wird dies positiv gewertet.
- Falls die Typen beider Attribute exakt übereinstimmen, wird auch dies höher gewertet, da es wahrscheinlicher ist, dass diese Attribute für einen Verbund angelegt wurden.
- Enthält eine Seite Attribute, die zu einer Tabelle gehören deren Attribute der Nutzer dem Interface hinzugefügt hat (User-Attribute) so werden diese Paare bevorzugt untersucht, da sehr wahrscheinlich ist, dass diese Attribute eine direkte Beziehung zueinander haben (das heißt ohne eine weitere Tabelle dazwischen).
- Da kürzere Beziehungen besser sind, aber durch dieses Punktesystem benachteiligt werden (da es ja nur wenige Attribute gibt, für die Punkte gesammelt werden können), erhalten Beziehungen mit drei oder weniger Attributen eine höhere Punktzahl.

Die Liste der Paare wird anschließend nach Punkten geordnet ausgegeben und kann in weiterverarbeitenden Schritten dann von oben nach unten abgearbeitet werden, da Beziehungen, die mit einer höheren Wahrscheinlichkeit nicht disjunkt sind, oben in der Liste stehen.

3.3.4.4 Ein Paar bewerten

Ein Kandidatenpaar wird zu einer Beziehung umgewandelt. Dabei wird der Beziehungstyp und die Kardinalität bestimmt. Sollte sich herausstellen, dass es keine disjunkte Beziehung ist, so wird die aus dem Paar erstellte Beziehung in eine Ergebnisliste aufgenommen.

Falls die Anzahl der Ergebnisse unter der vom Nutzer festgelegten Anzahl zu ermittelnder Beziehungen liegt, wird anschließend das nächste Paar bewertet.

3.3.4.5 Resultatliste sortieren

Sollten sich mehrere Paare als mögliche Beziehungen herausstellen, so ist es notwendig die Resultate der Qualität nach zu sortieren, da der Nutzer eventuell nicht immer intervenieren kann, beziehungsweise in einer großen Ergebnisliste auch unnütze Resultate enthalten sein können, und diese sollten selbstverständlich nicht ganz oben im Resultat stehen.

Die Qualität einer Beziehung kann an Beziehungstyp und Kardinalität festgemacht werden. „Gute“ Beziehungen werden demnach jene mit Typ C auf (mindestens) einer Seite sein. Die Kardinalität 1:x (oder umgekehrt) ist ebenfalls besser zu bewerten als N:x (oder umgekehrt), da der Grundgedanke bei einer Beziehung ja - siehe Fremdschlüsselbeziehung - eine möglichst redundanzfreie Speicherung in der Datenbank ist.

Hierbei wird ebenfalls ein Punktesystem zum Einsatz kommen, das die oben genannten Anforderungen umsetzt.

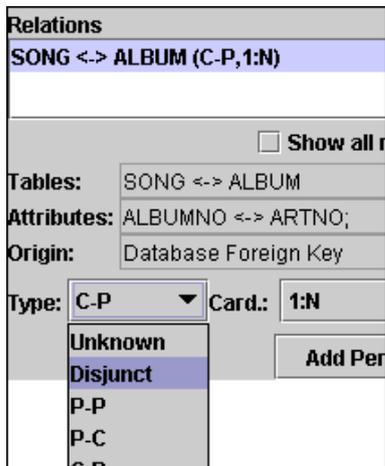
3.3.5 Löschen und Überschreiben von Beziehungen

Im Programm ist es selbstverständlich vorgesehen Beziehungen wieder zu löschen. Dies betrifft allerdings nur vom Nutzer hinzugefügte Beziehungen.

Beziehungen, die auf Fremdschlüssel in der Datenbank zurückgehen, können nicht gelöscht werden. Allerdings können diese Beziehungen überschrieben werden, indem eine Beziehung über die gleichen Attribute definiert wird. Dadurch lassen sich Zyklen in der Datenbank auflösen, denn disjunkte Beziehungen werden sowohl bei der Clusterbildung, als auch bei der Pfadbestimmung und Tests auf Azyklität nicht mit einbezogen. Disjunkte Beziehungen werden im Interface gespeichert, damit der Anfrageprozessor davon Kenntnis erlangt, das der Nutzer diese Beziehung nicht berücksichtigen will.

In Abbildung 3.4 ist dargestellt, wie die Erfassung disjunkter Beziehungen vonstatten geht, und wie das Resultat in der Interface-XML-Datei gespeichert wird.

Abbildung 3.4: Erzeugung disjunkter Beziehungen, und deren Darstellung im Interface



```
<relation to_obj="ALBUM"
origin="USER"
type="disjunct"
cardinality="1:N">
```

```
<reference from="ALBUMNO"
to="ARTNO" />
```

```
</relation>
```

3.3.6 Beziehungen zwischen Duplikaten

Objekte beziehen sich stets auf eine Tabelle in der Datenbank. Definiert man mehrere Objekte zu der selben Tabelle, so werden diese Objekte *Duplikate* genannt. Beziehungen unter Duplikaten müssen gesondert behandelt werden, denn der Zweck solcher Duplikate ist ein Selbstverbund, das heißt der Verbund einer Tabelle mit sich selbst.

Folgendes Beispiel zeigt Selbstverbund, bei dem der Verbund nicht über die selbe Spalte erfolgt:

Beispiel 3.3.4: Beispiel für einen Selbstverbund

Gegeben sei die folgende Relation:

ELTERN	Vater	Sohn
	Der Uropa	Der Opa
	Der Opa	Der Vater
	Der Vater	John

Um Informationen über die Großeltern von Personen zu ermitteln ist, es nötig einen Verbund mit

```
SELECT ELT1.VATER AS OPA,ELT2.SOHN AS ENKEL
FROM ELTERN AS ELT1, ELTERN AS ELT2
WHERE ELT1.Sohn=ELT2.Vater
```

durchzuführen. Das Ergebnis ist eine Tabelle, die Großväter und deren Enkel enthält. Um diesen Verbund im Anfrageprozessor zu ermöglichen ist es notwendig zwei Objekte ELT1 und ELT2 zu erstellen, und eine Beziehung zwischen beiden Objekten über die entsprechenden Attribute zu definieren.

Aus der Menge der Duplikate wird ein Objekt mit der „Außenwelt“, also mit (mindestens) einem der anderen Objekte im Interface in Beziehung gebracht. Alle anderen Duplikate werden mit

jeweils einem anderen Duplikat in Beziehung gebracht. Ein Zyklus kann dabei nicht entstehen. Bei der Erfassung der Daten werden die Objekte (ohne Beschränkung der Allgemeinheit) der Reihenfolge ihrer Eingabe nach miteinander verknüpft: das erste Objekt (das zu diesem Zeitpunkt noch kein Duplikat ist) wird mit den anderen Objekten im Interface in Beziehung stehen, das zweite Objekt mit dem ersten, usw.

Die Beziehungen werden bereits beim Hinzufügen der Attribute erzeugt. Zunächst wird eine Beziehung über die Primärschlüsselattribute erzeugt. Da dies aber nicht immer sinnvoll ist (je nach Anwendungsszenario) kann vom Nutzer selbst festgelegt werden, welche Attribute Verbundpartner werden.

Genauso wie bei normalen Beziehungen haben auch Duplikat-Beziehungen Typ und Kardinalität. Diese können automatisch bestimmt werden, aber auch vom Nutzer verändert bzw. (falls eine Bestimmung nicht möglich ist) angegeben werden. Ohne beide Angaben kann das Interface nicht geschrieben werden.

3.4 Weitere Informationen für das Interface

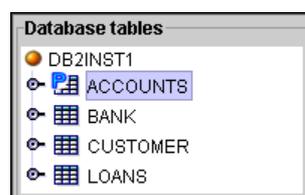
Neben den Attributen, die hinzugefügt werden müssen, gibt es noch eine Reihe weiterer Informationen, die für den Aufbau des Interfaces erforderlich sind.

3.4.1 Primärtabelle

Eine Datenbanktabelle kann als Primärtabelle ausgewählt werden.

Dies ermöglicht es der Anfrageverarbeitung einen Einstiegspunkt bei der Darstellung des Anfrageergebnisses anzubieten, zum Beispiel mittels Gruppierung von Konten, Krediten und Kunden nach einem Konto (siehe Beispiel in Abschnitt 2.3.2). Wie die Primärtabelle im Programm dargestellt wird ist in Abbildung 3.5 zu sehen.

Abbildung 3.5: Darstellung der Primärtabelle im Programm

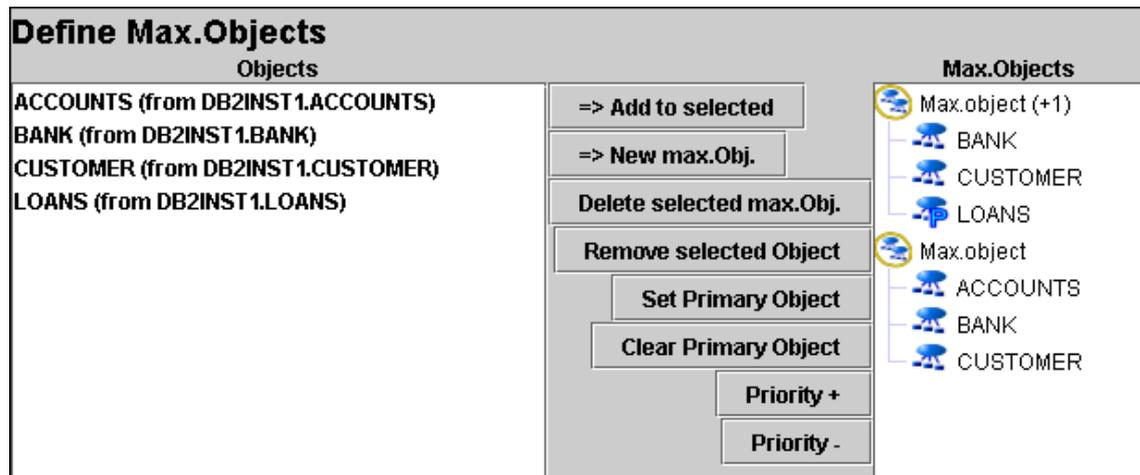


3.4.2 Maximale Objekte

Falls es einen Zyklus in den Beziehungen zwischen den Objekten gibt, so sind maximale Objekte erforderlich. Wird ein Zyklus gefunden, so kann das Interface nicht geschrieben werden ohne das

maximale Objekte definiert sind. Wird kein Zyklus festgestellt, so wird kein maximales Objekt benötigt und auch nicht ins Interface geschrieben, da es in diesem Fall nur ein (implizites) maximales Objekt gibt, nämlich das Interface selbst. Abbildung 3.6 zeigt, wie maximale Objekte im Programm erfasst werden. Auf dem Bild wurden zwei maximale Objekte erfasst.

Abbildung 3.6: Erfassung maximaler Objekte



Zum Test auf Azyklität wird ein einfacher Algorithmus zur Graphenreduktion verwendet:

1. Gegeben ist ein Graph G_{Start} , dessen Knoten die Objekte und dessen Kanten die Beziehungen zwischen den Objekten sind. Treten im Graphen Mehrfachkanten auf, d.h. existieren zwei Objekte mit mehr als einer Beziehung zueinander, so wird dies nur als eine Kante dargestellt, da dies keinen Zyklus darstellt. Dieser Graph sei der Anfangsgraph.
2. Schleife:
 - (a) Sichere den aktuellen Graphen G . Diese Sicherung sei G_s .
 - (b) Entferne vom aktuellen Graphen G alle Knoten, von denen nur eine Kante ausgeht. Es werden dabei auch implizit alle zu diesen Knoten gehörigen Kanten entfernt, wodurch nun andere Knoten nur noch eine Kante haben.
 - (c) Vergleiche den nun aktuellen Graphen G mit der Sicherung G_s . Falls sich Sicherung und aktueller Graph unterscheiden, so wird der Algorithmus bei Schleifenanfang fortgeführt. Sind Sicherung und aktueller Graph identisch, so kann am Graphen nichts weiter reduziert werden und die Schleife bricht ab.
3. Sind im aktuellen Graphen G noch Knoten enthalten, so sind diese Teil eines Zyklus. Ist der aktuelle Graph leer, so war der Anfangsgraph G_{Start} azyklisch.

Dieser Test wird auch verwendet um die Azyklität der einzelnen maximalen Objekte festzustellen.

In jedem maximalen Objekt kann ein Primärobjekt aus den enthaltenen Objekten gewählt werden. Wenn Anfragen auf einem maximalen Objekt durchgeführt werden, in dem die Primärtabelle nicht enthalten ist, so kann stattdessen das Primärobjekt verwendet werden.

Falls mehrere maximale Objekte für eine Anfrage benötigt werden, und in keinem die Primärtabelle enthalten ist, so kann die Auswahl des nun zu verwendenden Primärobjekts (und der zugehörigen Tabelle) über eine Priorisierung erfolgen. Jedem maximalen Objekt kann deshalb eine beliebige Priorität (zwischen 0 und unendlich) zugewiesen werden. In Abbildung 3.6 hatte das obere maximale Objekt das Primärobjekt „Loans“ und eine Priorität von +1.

Bei der Festlegung maximaler Objekte muss der Nutzer darauf achten, dass jedes Objekt auch in einem maximalen Objekt enthalten ist. Ist dies nicht der Fall, so bricht die Interface-Erstellung mit einer Fehlermeldung ab.

4. Erzeugung des Interfaces

4.1 Voraussetzungen

In Hinblick auf den Anfrageprozessor und die Anwendung des Interfaces können wir folgende Bedingungen an das Interface stellen:

- Aufbau:
 - das Interface besteht aus mindestens einem Objekt und beliebig vielen maximalen Objekten. Optional ist die Angabe der Primärtabelle.
 - Objekte bestehen aus (Schlüssel-)Attributen und mindestens einer Beziehung. Zu einem Objekt gehören die Angabe eines Namens und einer zugehörigen Tabelle.
 - Bei (Schlüssel-)Attributen müssen Name, Spaltenname in der Tabelle, SQL-Typ, sowie die Information, ob es sich um ein Projektions- und/oder Selektionsattribut handelt, angegeben sein.
 - Beziehungen beinhalten neben der Angabe eines Zielobjektes, Typ, Kardinalität und Herkunft noch eine Liste von mindestens einem Attributpaar.
 - Maximale Objekte enthalten mindestens ein Objekt, die Angabe von Priorität und einem Primärobjekt ist optional.
- Globale Eindeutigkeit der Objekte:

Objekte müssen im Interface eindeutig sein. Da Objekte über den Namen identifiziert werden, wird im Programm dafür gesorgt, dass die Namen eindeutig sind: falls nötig wird eine fortlaufende Zahl an den Tabellennamen angehängt, der Objektname kann aber auch vom Nutzer selbst eingegeben werden.
- Globale Eindeutigkeit der Attribute:

Objektattribute müssen global, d.h. im gesamten Interface, eindeutig sein. Im Zweifelsfall kann diese Eindeutigkeit durch Anfügen des Tabellennamen erreicht werden. Da dies aber bei Duplikaten problematisch wird, habe ich mich dazu entschlossen, im Programm eine fortlaufende Nummer an den Attributnamen anzuhängen (dies ist vom Nutzer änderbar).
- Alle Objekte stehen in Beziehung zueinander:

Es darf keine Objekte geben, die isoliert von anderen Objekten stehen, d.h. nicht mittels

Verbund erreichbar sind. Derartige Objekte sind am Aufbau der Universalrelation nicht beteiligt und deshalb im Interface unzulässig.

- **Maximale Objekte:**

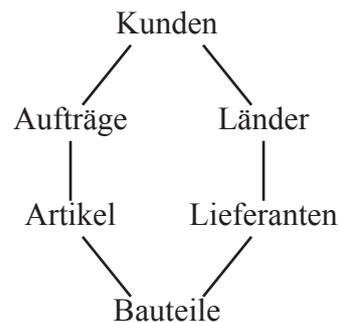
Falls maximale Objekte (aufgrund eines Zyklus in den Beziehungen) nötig sind, so müssen diese azyklisch sein. Außerdem müssen alle Objekte in maximalen Objekten enthalten sein. Falls keine maximalen Objekte notwendig sind, sollte auch kein maximales Objekt im Interface enthalten sein.

4.2 Die Schritte zum Aufbau des Interfaces

Gegeben sind bis hier lediglich Attribute, eventuell zusätzliche Beziehungen, und maximale Objekte. Da der Nutzer bei der Erfassung aber kein Augenmerk auf Attribute nehmen soll, die nur zum Verbund notwendig sind, müssen diese automatisch hinzugefügt werden, um die Voraussetzungen an das Interface zu erfüllen.

Zunächst müssen wir jedoch den Begriff Pfad klären, der hier verwendet wird. Ein Pfad ist wie folgt definiert: Er besteht aus Schritten, jeder Schritt ist eine Tabelle. Von einem Schritt des Pfades zum nächsten kommt man mittels einer Beziehung, welche ebenfalls (als Transition zwischen den Schritten) im Pfad gespeichert wird. Ein Pfad gibt an welche Beziehungen man ausnutzen muss, um die Tabelle am Start mit der Tabelle am Ende des Pfades mittels Verbund zu vereinigen. Der nachfolgende Algorithmus sucht solche Pfade. Das Problem, das dieser Algorithmus löst, ist NP-vollständig. Zunächst möchte ich den Begriff des Pfades jedoch an einem Beispiel erläutern:

Beispiel 4.2.1: Beispiel für Pfade



Angenommen im Interface sollen Attribute der Relationen *Kunden* und *Bauteile* enthalten sein. Es existieren zwei Pfade von der Relation *Kunden* zur Relation *Bauteile*: *Kunden-Aufträge-Artikel-Bauteile* und *Kunden-Länder-Lieferanten-Bauteile*. Gesucht sind Informationen über *Kunden*, und den ausgelieferten *Bauteile*. Der Pfad über die *Aufträge* ist daher der richtige, da nur so die gewünschte Information erzeugt werden kann. Der andere Pfad

würde alle Bauteile liefern, für die es Lieferanten gibt, die aus dem selben Land wie der Kunde kommen. Hieraus wird deutlich, dass das Programm nicht alleine entscheiden kann wie das Interface erzeugt wird.

Algorithmus zur Pfadbestimmung:

1. Gegeben ist eine Menge von Attributen $M_{UserAtt}$, die vom Nutzer hinzugefügt wurden, sowie eine Menge von Beziehungen M_{Bez} .
2. Aus der Menge von Attributen wird die Menge der vom Nutzer angesprochenen Tabellen erzeugt, indem alle zu den Attributen gehörenden Tabellen in einer Menge $M_{UserTab}$ gesammelt werden.
3. Nun wird für jede der Tabellen $t_1 \in M_{UserTab}$ jede Beziehung $b \in M_{Bez}$ gesucht, bei der auf der einen Seite t_1 , und auf der anderen eine Tabelle t_2 steht. Alle diese t_2 werden in einer Menge $M_{Partner}$ gesammelt.
4. Es wird eine Menge M_{Pfade} mit allen möglichen Pfaden der Länge 1 erzeugt, bei denen der Start $t_1 \in M_{UserTab}$ ist, und das Ende $t_2 \in M_{Partner}$.
5. Schleife: Iteriere über alle Pfade $p \in M_{Pfade}$
 - (a) Falls p mehr Schritte hat als es Tabellen in der Datenbank gibt, so entferne p aus M_{Pfade} und springe zum letzten Schritt dieser Schleife, da dann der Pfad nur noch zyklisch werden kann.
 - (b) Falls p eine Tabelle $t \in M_{UserTab}$ als letzten Schritt hat, so wird dieser Pfad in eine Ergebnismenge M_{Erg} hinzugefügt, und aus M_{Pfade} entfernt, und es wird beim letzten Schritt dieser Schleife (Schritt 5f) fortgesetzt.
 - (c) Sammle alle Beziehungen $b \in M_{Bez}$ in einer Menge $M_{PfadBez}$, bei denen das Ende von p auf einer Seite von b vorkommt.
 - (d) Entferne p aus M_{Pfade} .
 - (e) Iteriere über alle Beziehungen $b \in M_{PfadBez}$:
 - i. Erzeuge einen neuen Pfad q , indem zunächst eine Kopie von p erstellt wird.
 - ii. Das Ende von p ist eine Seite von b . Die Tabelle auf der anderen Seite wird als neuer Schritt an q angefügt. Das b wird ebenfalls im Pfad gespeichert.
 - iii. Prüfe, ob q einen Zyklus enthält (d.h. eine Tabelle im Pfad mehr als einmal vorkommt). Falls ja wird q verworfen. Ansonsten füge q der Menge M_{Pfade} hinzu, so dass dieses q beim nächsten Durchlauf der äußeren Schleife mit geprüft wird.
 - iv. Falls ein weiteres b existiert, gehe mit dem nächstem b zum Schleifenanfang (Schritt 5(e)i) zurück.
 - (f) Falls ein weiterer Pfad p zu testen ist: gehe mit dem nächsten p zurück zum Schleifenanfang (Schritt 5a). Es ist zu beachten, dass die in der inneren Schleife hinzugefügten Pfade werden erst im nächsten Durchlauf bearbeitet werden.

6. In der Menge M_{Erg} befinden sich diverse Pfade zwischen den Tabellen aus $M_{UserTab}$. Die Informationen aus dieser Menge werden nun in verschiedenen Listen gespeichert. Pfade mit gleichem Start/Ende werden in der selben Liste gespeichert. Diese Listen werden der Länge nach sortiert, da kürzere Pfade weniger Verbünde bedeuten, und somit zu bevorzugen sind. Existieren mehrere kürzeste Pfade, so wird derjenige mit den meisten Tabellen aus $M_{UserTab}$ an die Spitze der Liste gesetzt. Alle Pfade an den Spitzen der jeweiligen Listen werden zunächst *gesetzt*, das bedeutet diese Pfade werden zur Verwendung im Interface vorgesehen.
7. Dem Nutzer wird nun die Möglichkeit eingeräumt, die verwendeten Pfade zu ändern, da die semantische Bedeutung von Pfaden vom Programm nicht ermittelt werden kann (siehe Beispiel 4.2.1).
8. In der Ergebnismenge befinden sich nun alle zu verwendenden Pfade zwischen allen Paaren von Tabellen aus $M_{UserTab}$. Wie oben erwähnt gehören zu Pfaden definitionsgemäß auch die Beziehungen, aus denen sie entstanden sind. Wir erzeugen eine neue Menge M_{Atts} , die alle Attribute dieser Beziehungen enthalten.
9. Die (Objekt-)Attribute, die in M_{Atts} enthalten sind, könnten bereits vom Nutzer hinzugefügt worden sein. Wir bilden deshalb die Differenz $M_{Atts} - M_{UserAtt}$ und erhalten somit alle Attribute, die im Interface noch fehlen.

Der Algorithmus liefert also alle fehlenden Attribute, die benötigt werden, um einen Verbund zur Universalrelation zu ermöglichen. Diese Attribute werden dem Interface hinzugefügt. Damit der Nutzer diese von den „normalen“, also den von ihm selbst hinzugefügten, Attributen unterscheiden kann, werden sie im Programm speziell als System-Attribute gekennzeichnet.

Falls der Nutzer es wünscht, werden automatisch alle Schlüsselattribute der Tabellen zu den Objekten im Interface hinzugefügt. Die Schlüsselattribute werden ebenfalls als Systemattribute gekennzeichnet.

Das Interface ist an diesem Punkt vollständig spezifiziert, liegt aber noch in einer programminternen Repräsentation vor. Die Anforderungen an das Interface, die bei den Voraussetzungen (siehe 4.1) formuliert wurden, wurden bisher lediglich vom Programm zugesichert. Dies geschah auch nur in einem Maße, in dem es bei der Erfassung der Daten möglich war. Zum Beispiel kann ein eventuelles Fehlen von maximalen Objekten erst bei der Interface-Erstellung beim Nutzer reklamiert werden. Das Programm erzeugt deshalb nun aus der internen Repräsentation des Interfaces eine andere Repräsentation, die dem Ziel der XML-Datei schon recht nahe kommt. Diese Repräsentation beinhaltet alle Integritätsbedingungen des Interface, und sie kann nur erzeugt werden wenn eine syntaktisch gültige XML-Datei mit einem semantisch korrekten Inhalt erzeugt werden kann. Aus dieser Darstellung des Interfaces kann anschließend recht einfach eine XML-Datei erzeugt werden.

4.3 DTD des Interfaces

Viele der Anforderungen an das Interface können bereits in einer DTD formuliert werden:

Der Wurzelknoten:

```
<!ELEMENT interface (object+, maxobject*)>
<!ATTLIST interface primarytable CDATA #IMPLIED>
```

Objekte:

Laut DTD ist es möglich, dass keine Attribute im Objekt enthalten sind. Dieser Fall wird allerdings von der Anwendung verhindert, da leere Objekte sinnfrei sind. Um Referenzen auf Objekte bereits in der DTD zu verankern, ist das Attribut name eine eindeutige ID. Tabellen werden im Interface in der Form *Schema.Tabelle* gespeichert, sofern ein Schema in der Datenbank angegeben ist.

```
<!ELEMENT object (key*, col*, relation+)>
<!ATTLIST object name ID #REQUIRED
                table CDATA #REQUIRED>
```

(Schlüssel-)Attribute:

Attribute und Schlüsselattribute unterscheiden sich lediglich durch das XML-Tag, sind inhaltlich jedoch identisch.

```
<!ELEMENT key EMPTY>
<!ATTLIST key name CDATA #REQUIRED
              column CDATA #REQUIRED
              type CDATA #REQUIRED
              forselection (true|false) #REQUIRED
              forprojection (true|false) #REQUIRED>
```

```
<!ELEMENT col EMPTY>
<!ATTLIST col name CDATA #REQUIRED
              column CDATA #REQUIRED
              type CDATA #REQUIRED
              forselection (true|false) #REQUIRED
              forprojection (true|false) #REQUIRED>
```

Beziehungen:

Die Eigenschaften der Beziehung werden als XML-Attribut gespeichert, die Attribute aus der Menge *paare* der Beziehung werden als untergeordnete XML-Element-Menge gespeichert.

```

<!ELEMENT relation (reference)+>
<!ATTLIST relation to_obj IDREF #REQUIRED
                  origin (Statistic|User|FK_Stat|FK) #REQUIRED
                  type (C-C|C-P|P-C|P-P|Disjunct) #REQUIRED
                  cardinality (1:1|1:N|N:1|N:N) #REQUIRED>

<!ELEMENT reference EMPTY>
<!ATTLIST reference from CDATA #REQUIRED
                  to CDATA #REQUIRED>

```

Maximale Objekte:

```

<!ELEMENT maxobject (member)+>
<!ATTLIST maxobject primaryobject CDATA #IMPLIED
                  priority CDATA #IMPLIED>

<!ELEMENT member EMPTY>
<!ATTLIST member name IDREF #REQUIRED>

```

4.4 Beispiel-Interface

In diesem Abschnitt möchte ich an einem etwas komplexeren Beispiel für eine Interface-Datei den Weg der Interface-Erstellung zeigen.

In der Datenbank seien 3 Tabellen enthalten, ALBUM, COVERS und SONG, die folgende Inhalte haben:

ALBUM:

TITEL VARCHAR(50) NOT NULL	<u>ARTNO</u> DECIMAL(12,0) NOT NULL	GENRE CHAR(20)
Automatic for the people	093624505525	Independent
The unforgettable fire	007196101944	Pop
Reptile	093624796695	Blues
Play	016025611720	

COVERS:

<u>ALBUM</u> DECIMAL(12,0) NOT NULL	COVERFILE CHAR(50) NOT NULL
093624505525	REM_AFTP.pic
007196101944	U2_TUF.pic

SONG:

<u>ALBUMNO</u>	<u>TRACKNO</u>	TRACKNAME
DECIMAL(12,0) NOT NULL	SMALLINT NOT NULL	CHAR(50) NOT NULL
093624505525	1	Drive
093624505525	2	Try not to breathe
093624505525	3	The sidewinder sleeps tonite
007196101944	1	A sort of homecoming
007196101944	2	Pride
007196101944	3	Wire
007196101944	4	The unforgettable fire
016025611720	1	Honey
016025611720	2	Find my baby
016025611720	3	Porcelain

Die Schlüssel sind jeweils unterstrichen, und die Definition der Spalte ist im Tabellenkopf angegeben. Als Fremdschlüssel ist in der Tabelle SONG definiert:

```
CONSTRAINT ALBUM_SONG_FK FOREIGN KEY (ALBUMNO)
REFERENCES ALBUM (ARTNO) ON DELETE CASCADE
```

Das automatische Hinzufügen der Schlüsselattribute ist ausgeschaltet, genauso wie die DTD nicht intern gespeichert wird. Das Schema der Tabellen ist DB2INST1.

Nach Programmstart werden folgende Attribute dem Interface hinzugefügt: ALBUM.TITEL, SONG.TRACKNO, SONG.TRACKNAME, COVERS.COVERFILE. Alle Attribute sind Projektionsattribute, nur ALBUM.TITEL ist zusätzlich Selektionsattribut. Die Attributnamen im Interface sind identisch zu den Attributnamen in der Relation.

Als Primärtabelle wird anschließend ALBUM festgelegt.

Das Interface kann jetzt noch nicht erstellt werden, da Beziehungen fehlen. Das Programm wird folgende zwei Cluster finden: (ALBUM, SONG) und (COVERS). Das liegt daran, dass in der Datenbank keine Fremdschlüssel für die Tabelle COVERS definiert wurden. Die Fremdschlüsselbeziehung zwischen SONG und ALBUM wird im Programm folgendermaßen dargestellt:

({(SONG.ALBUMNO, ALBUM.ARTNO)}, C-P, 1:N), mit dem Ursprung „Fremdschlüssel“.

Folgende Werte werden vom Programm zur Beziehungsbestimmung ermittelt:

CNT_L=3, CNT_R = 4, CNT_JOIN=3, CNT_ALL_L=10, CNT_ALL_R=4.

Lassen wir das Programm die Beziehungen erraten, so findet es die folgenden zwei:

- ((ALBUM.ARTNO, COVERS.ALBUM)}, P-C, 1:1)
- ((SONG.ALBUMNO, COVERS.ALBUM)}, P-C, 1:N)

Beide Beziehungen werden dem Interface hinzugefügt. Dadurch entsteht jedoch ein Zyklus, und es wird notwendig sein maximale Objekte definieren.

Um maximale Objekte anzulegen müssen wir zunächst das Interface aufbauen. Damit will das Programm sicherstellen, dass keine Objekte fehlen. Tun wir dies, so werden alle direkten Beziehungen zwischen den Tabellen ausgenutzt um Pfade zu bilden, zum Beispiel wird zwischen ALBUM und SONG der Pfad direkt von ALBUM nach SONG genutzt, da es hier eine Beziehung

gibt. Ein Pfad über COVERS existiert zwar, ist aber sinnfrei und wird aufgrund seiner Länge vom Programm nicht verwendet. Wir müssen demnach die Pfade nicht ändern und bauen das Interface auf. Dadurch werden folgende Attribute hinzugefügt: COVERS.ALBUM (mit Namen REL_ALBUM im Interface), SONG.ALBUMNO (als REL_ALBUMNO) und ALBUM.ARTNO (als REL_ARTNO).

Nun definieren wir folgende maximale Objekte: (ALBUM, COVERS) und (ALBUM, SONG). Die Primärtabelle ist in beiden maximalen Objekten enthalten, somit werden hier keine Primärobjekte oder Prioritäten festgelegt. Damit sind alle Informationen für das Interface vollständig, und folgende XML-Datei wird erzeugt:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE interface SYSTEM "iface.dtd">

<interface primarytable="DB2INST1.ALBUM">
  <object name="ALBUM" table="DB2INST1.ALBUM">
    <key name="REL_ARTNO" column="ARTNO" type="DECIMAL(12,0)"
      forselection="false" forprojection="false" />
    <col name="TITEL" column="TITEL" type="VARCHAR(50)"
      forselection="true" forprojection="true" />
    <relation to_obj="COVERS" origin="Statistic"
      type="P-C" cardinality="1:1">
      <reference from="REL_ARTNO" to="REL_ALBUM" />
    </relation>
  </object>
  <object name="COVERS" table="DB2INST1.COVERS">
    <key name="COVERFILE" column="COVERFILE" type="CHAR(30)"
      forselection="false" forprojection="true" />
    <key name="REL_ALBUM" column="ALBUM" type="DECIMAL(12,0)"
      forselection="false" forprojection="false" />
  </object>
  <object name="SONG" table="DB2INST1.SONG">
    <key name="REL_ALBUMNO" column="ALBUMNO"
      type="DECIMAL(12,0)"
      forselection="false" forprojection="false" />
    <key name="TRACKNO" column="TRACKNO" type="SMALLINT"
      forselection="false" forprojection="true" />
    <col name="TRACKNAME" column="TRACKNAME" type="CHAR(50)"
      forselection="false" forprojection="true" />
    <relation to_obj="ALBUM" origin="FK"
      type="C-P" cardinality="1:N">
      <reference from="REL_ALBUMNO" to="REL_ARTNO" />
    </relation>
    <relation to_obj="COVERS" origin="Statistic">
```

```
        type="P-C" cardinality="1:N">
    <reference from="REL_ALBUMNO" to="REL_ALBUM" />
</relation>
</object>
<maxobject>
    <member name="ALBUM" />
    <member name="COVERS" />
</maxobject>
<maxobject>
    <member name="ALBUM" />
    <member name="SONG" />
</maxobject>
</interface>
```

5. Implementierungs-Spezifika

5.1 Projektstruktur

Die Implementierung ist modular aufgebaut, wodurch sich Änderungen lokal und sehr einfach durchführen lassen. Der generelle Aufbau ist in Darstellung 5.1 aufgeführt, wobei die wichtigen Zusammenhänge durch Linien gekennzeichnet sind. Nicht dargestellt ist die global verfügbare Komponente für das Logging.

5.2 GUI

Den Kern der grafischen Oberfläche bildet die Klasse `MainWindow`, welche das Hauptfenster darstellt und von dem aus der Benutzer alle Aktionen starten kann. Einige Komponenten sind in das Hauptfenster fest integriert, wie beispielsweise die Darstellung der Datenbank sowie der hinzugefügten Attribute und Objekte. In Abbildung 5.2 ist ein Screenshot aus dem Programm zu sehen, der das Hauptfenster darstellt, wobei dort gerade Beziehungen bearbeitet werden.

Die Eingabe- und Bearbeitungsmasken wurden jedoch in eigene Panels ausgelagert, die jeweils zur Laufzeit erzeugt und ins Hauptfenster eingebunden werden. Dazu zählen alle Panels, die im Mittelteil des Hauptfensters dargestellt werden. Dadurch ist ein einfaches Debuggen möglich gewesen, da geänderte Komponenten zur Laufzeit neu geladen werden konnten. Auch ist somit eine einfache Erweiterbarkeit gewährleistet, da das Hauptfenster bei Änderungen unberührt bleibt.

Die angepasste Darstellung von Objekt-, Datenbank- und Maximale Objekte-Baum, bei der Elemente mit bestimmten Eigenschaften mit anderen Symbolen angezeigt werden, ist durch die Verwendung eigener Klassen zum Zeichnen des Baumes realisiert, die die Klasse `DefaultTreeCellRenderer` erweitern. Um überhaupt erweiterte Informationen in den Bäumen ohne Einschränkung der Arbeitsweise zu integrieren, sind alle Knoten im Baum Instanzen einer speziellen Klasse, dem `ExpTreeNode`. Dieser besteht aus einem String, der im Baum angezeigt und von der `toString()`-Methode zurückgegeben wird, einem weiteren String welcher Flags speichern kann (zuzüglich passender Funktionen zum setzen, lesen und löschen von Flags), sowie einem Objekt, zu dem dieser Knoten gehört. Die Flags dienen den `TreeCellRenderern` dazu zu erkennen, ob es sich um einen Primärschlüssel, etc. handelt. Bei der Erstellung des Baumes werden diese Informationen mit angefügt. Der `ExpTreeNode` eignet sich auch her-

Abbildung 5.1: Programmstruktur

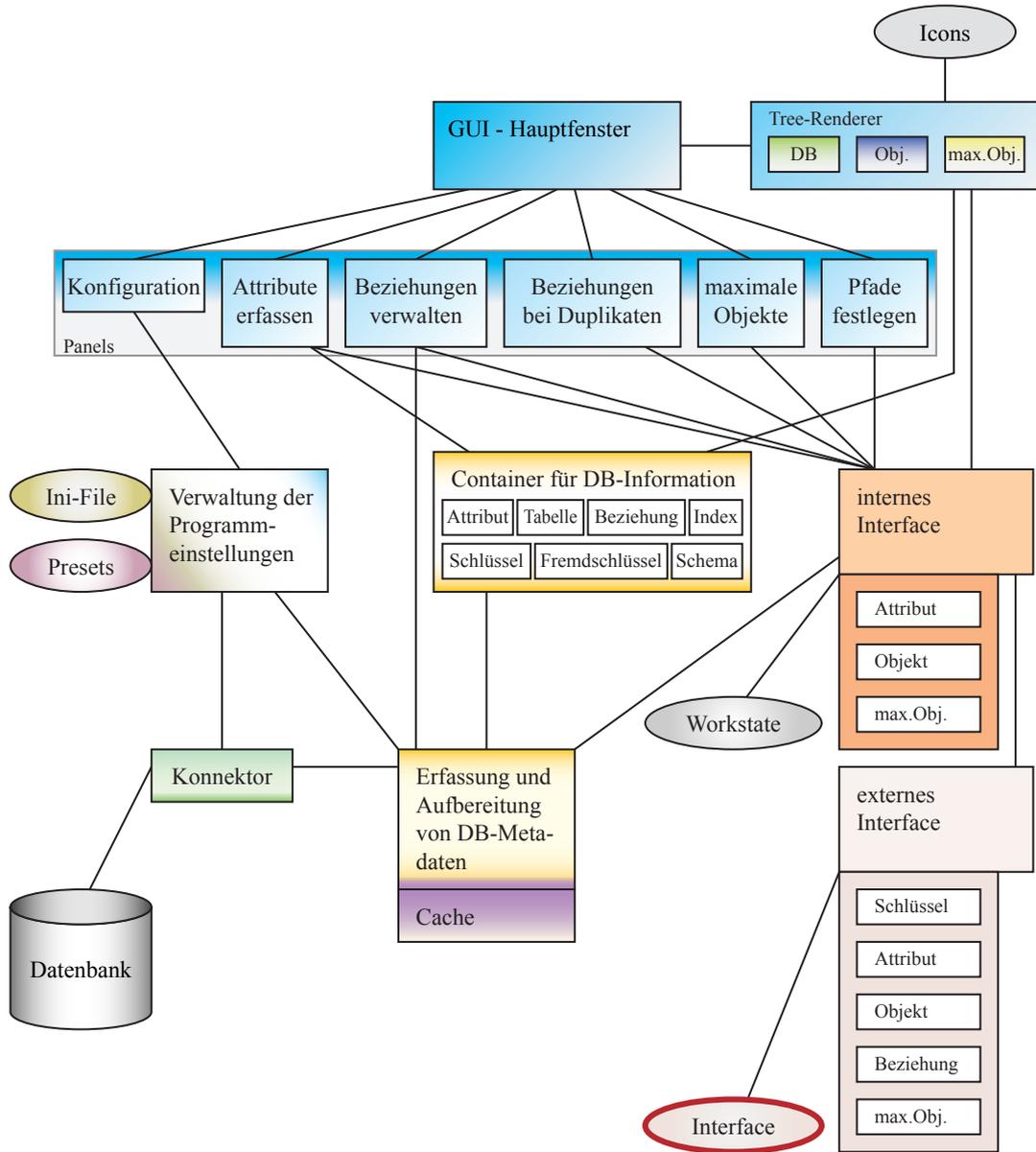
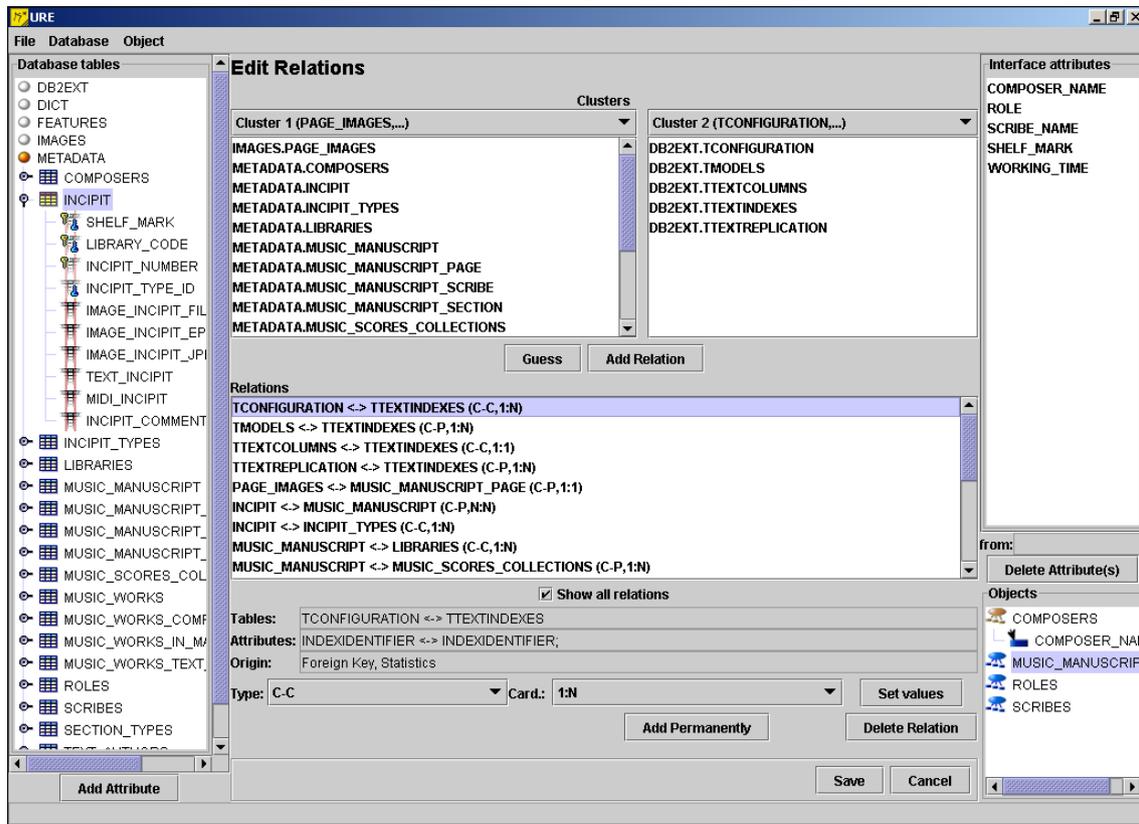


Abbildung 5.2: Das Hauptfenster



vorrangig für List- und ComboBox-Klassen, da es so möglich ist Objektrepräsentation und Objekt zusammenzuhalten, und somit eine komplizierte Zuordnung überflüssig wird.

5.3 Hilfsklassen

Folgende Klassen haben eine Hilfsfunktion: Bitfield, Cluster, Comparators, Mini-Interface und StringResult. Diese Klassen haben nichts mit der eigentlichen Interfacerestellung zu tun.

Die Klasse Bitfield ermöglicht die Arbeit mit Bitvektoren, die als kombinatorisches Hilfsmittel an der Teilmengenbestimmung bei den Statistikfunktionen beteiligt sind.

Die Klasse Comparators enthält einige statische Objekte, die das Interface Comparator implementieren. Diese werden bei der Arbeit mit Mengen und Zuordnungstabellen eingesetzt. Der wichtigste Comparator trägt den Namen Sicc, das bedeutet „String Ignore Case Comparator“, und sichert (unter anderem) die semantische Äquivalenz von Groß- und Kleinschreibung bei Objekten und Attributen zu. Zur Anwendung kommen diese Comparator-Objekte hauptsächlich im Zusammenhang mit TreeSet- und TreeMap-Objekten.

`Cluster` und `MiniIface` werden in anderen Kapiteln beschrieben, `StringResult` dient der Speicherung von Tupeln, die in einer Menge gesammelt werden um die alternativen Anfragen für die Bestimmung der Kardinalität zu unterstützen.

5.4 Sammeln von Metadaten

Die vermutlich umfangreichste Klasse verbirgt sich hinter dem Namen `Metacollector`. Diese Klasse war ursprünglich dazu gedacht, Metainformationen aus der Datenbank zu sammeln und bereitzustellen. Im Laufe der Entwicklung hat sich der Funktionsumfang stark erweitert. Hier eine Übersicht über die Aufgaben des `Metacollector`:

- Sammeln von Metainformationen aus der Datenbank, dazu gehören
 - Schemata
 - Tabellen
 - Attribute
 - Schlüssel
 - Fremdschlüssel
 - Indizes
- Cachen der gesammelten Informationen:
Zum Cachen steht dem `Metacollector` die Klasse `Tabulator` zur Verfügung. Diese speichert die Daten nach dem *Cache On Demand*-Prinzip: Daten werden erst gecacht, wenn das erste Mal auf sie zugegriffen wird. Der `Metacollector` stellt dabei die Anfragen nach Metadaten direkt an den Cache. Dieser prüft, ob die Daten im Speicher sind. Sind die Daten nicht gespeichert, dann ruft der Cache die entsprechende `retrieve`-Methode auf, um die Daten aus der Datenbank zu holen. Der Cache kann vom Benutzer manuell auch geleert werden.
- Beziehungsverwaltung:
Der `Metacollector` hat die Aufgabe Beziehungen zu erzeugen und zu verwalten. Beziehungen werden nicht im (internen) Interface verwaltet, sondern hier. Der Grund dafür ist, dass diverse Operationen auf Beziehungen mit Metadaten arbeiten, und eine Kapselung im Interface somit nicht sinnvoll gewesen wäre. Im `Metacollector` werden Beziehungsobjekte aus Fremdschlüsseln erzeugt, nutzerdefinierte Beziehungen verwaltet und Beziehungen bewertet.
- Clusterbildung:
Da zur Clusterbildung Informationen über Tabellen und Beziehungen notwendig sind befindet sich die Funktionalität ebenfalls in dieser Klasse. Der `Metacollector` nutzt diese Informationen, um die Cluster der Datenbank zu ermitteln.

- **Pfadbestimmung:**
Ebenfalls ein sehr datenbanknahes Problem, das an dieser Stelle am einfachsten zu lösen ist, ist die Pfadbestimmung, also das Finden eines Weges zwischen zwei Tabellen mittels Beziehungen. Der `Metacollector` bietet Funktionen zur Pfadbestimmung zwischen zwei gegebenen Tabellen, sowie zur Bestimmung des Typs eines Pfades (C-C, etc., siehe Abschnitt zu 3.3.2 Beziehungstypen) mittels einer Datenbankanfrage.
- **Statistik-Funktionen:**
Da bereits sämtliche Beziehungsprobleme hier verwaltet werden, liegt es natürlich nahe, dass die Statistik-Funktionalität zum Finden fehlender Beziehungen in dieser Klasse implementiert ist. Jedes der in 3.3.4 aufgeführten Einzelprobleme ist in einer eigenen Methode implementiert.

5.5 Weitere datenbankspezifische Komponenten

Die Speicherung der Datenbankinformationen erfolgt in speziellen Klassen, sogenannte *Container-Klassen*, die die Daten zusammenhalten und die Datentypen um grundlegende Funktionalität erweitern (zum Beispiel bei Beziehungen die Umwandlung von Typ und Kardinalität in Strings und zurück). Da sich die Verarbeitung dadurch wesentlich einfacher gestaltet, wurden viele dieser Klassen nicht als Kapseln/ADTs implementiert, sondern mit öffentlichen Feldern ausgestattet. Beispiele für Container sind die Klassen `DbRelation`, `DbAttrib`,

Eine weitere wichtige Klasse ist der `Connector`. Diese Klasse ist statisch, d.h. nicht instanzierbar. An diese Klasse werden JDBC-Treiber und URL sowie Nutzer und Passwort übergeben. Zurückgegeben wird eine `JDBC-Connection`. Es ist auch möglich eine Verbindung zu testen (unter Angabe von URL, Treiber, Nutzer und Passwort). Falls nötig kann diese Klasse einfach zu einem `Connection-Pool` erweitert werden, ohne das sich für die restlichen Klassen etwas ändert.

5.6 Konfiguration

Wie in jedem Programm gibt es auch bei URE einige Einstellungen, die notwendig sind. Neben den nicht problemrelevanten Einstellungen für den Programmablauf wären derer folgende zu nennen:

- Der Nutzer hat die Möglichkeit anzugeben, ob jedem Objekt die Primärschlüssel der assoziierten Tabelle hinzugefügt werden sollen. Dies würde dann automatisch beim Schritt der Interface-Erzeugung passieren.
- Falls es erwünscht ist, kann die DTD für das Interface in die Interface-XML-Datei mit hineingeschrieben werden, so dass sie nicht als externe Datei benötigt wird.

- In der Konfiguration wird auch die Verbindung zur Datenbank spezifiziert. Dies umfasst die Werte für JDBC-Treiber, JDBC-URL zur Datenbank, Nutzernamen und Passwort, sowie die Option zur Vermeidung von Unteranfragen. Da in der Regel mehrere Verbindungen genutzt werden, können Verbindungsdaten in so genannten *Presets* gespeichert werden. Diese sichern die aktuellen Verbindungsdaten, die danach mit einem nutzerdefinierten Namen versehen und gespeichert werden. Presets werden in einer eigenen Datei gespeichert.
- Es sind auch Einstellungen über das Verhalten bei der Statistik zur Beziehungsbestimmung möglich. Veränderbar sind folgende Werte:
 - Maximale Anzahl von Attributen in einer Beziehung: Dies beschränkt die Anzahl der Attribute in Kandidaten für Beziehungen, da es nicht immer sinnvoll ist, Beziehungen mit einer sehr hohen Anzahl von beteiligten Attributen zu testen, auch wenn dies theoretisch auf Grund der Datentypen möglich wäre.
 - Maximale Anzahl von ermittelten (gültigen) Beziehungen: Hierdurch bricht die Suche nach weiteren Beziehungen ab, sobald eine gewisse Anzahl an Beziehungen gefunden wurde, die nicht disjunkt oder unbestimmbar sind. Es ist allerdings möglich, dass dadurch die gesuchte Beziehung nicht angezeigt wird, wenn sie sich bei der Vorsortierung nicht ausreichend qualifiziert hat.
 - Präzision von P-P-Beziehungen: Falls P-P-Beziehungen gefunden werden, das heißt auf beiden Seiten nur jeweils ein Teil aller Tupel einen Verbundpartner finden, so können Beziehungen, bei denen nur wenige Tupel einen Partner finden, ein großer Teil jedoch nicht, aus dem Ergebnis ausgeschlossen werden, da in diesem Fall der Verbund eher zufällig als gewollt zustande gekommen sein könnte. Die Präzision ist der Prozentsatz an Tupeln mit Verbundpartnern gemessen an der Gesamtanzahl der Tupel. Ein zu hoch gewählter Prozentsatz kann das Finden von Beziehungen in sehr kleinen Relationen allerdings verhindern.

Für die Konfiguration steht die statische Klasse `Configurator` zur Verfügung. Die Klasse ist statisch, da die Konfiguration global zur Verfügung stehen soll, das heißt aus der GUI heraus ebenso wie aus dem `Metacollector`, oder dem `Connector`. Im `Configurator` befinden sich alle für das Programm wichtigen Einstellungen, inklusive der Verwaltung der Datenbank-Presets. Konfigurationsdaten und Presets werden in XML-Dateien gespeichert.

5.7 Sicherung des Arbeitszustandes

Es gibt Situationen, in denen es sinnvoll ist, den aktuellen Arbeitszustand zu sichern, um die Arbeit an einem Punkt zu unterbrechen. Um dies zu tun ist es notwendig, eine semantisch schwächere Version des Interfaces zu speichern. Wir haben bereits im vorigen Kapitel gesehen, dass zum Interface eine Reihe wichtiger Integritätsbedingungen gehören, die zum Teil erst

beim Schreiben zugesichert werden können. Dies ist hier jedoch nicht wichtig, da der Arbeitszustand soweit, wie er bis zum Zeitpunkt der Sicherung aufgebaut wurde, korrekt ist. Es ist daher lediglich notwendig sicherzustellen, dass der aktuelle Zustand wieder hergestellt wird.

Die Arbeit mit JDOM hat sich als sehr einfach herausgestellt, und XML ist für diese Aufgabe ein gut geeignetes Format, da es hierarchische Strukturen gut abbilden kann. Deshalb habe ich mich entschlossen, den Arbeitszustand als XML-Datei zu sichern. Diesen Prozess nenne ich – angelehnt an Konzepte der Programmiersprache – Serialisierung.

Das Interface im Programm ist intern ebenfalls hierarchisch gespeichert, Objekte, Attribute, etc. sind jeweils eigene Klassen, die miteinander in Beziehung stehen. Daher kann ein Großteil der Serialisierung in den entsprechenden Klassen des Interfaces verschoben werden. Dazu ist in jeder Klasse eine Methode zur Serialisierung und Deserialisierung implementiert.

Die koordinierende Klasse `MiniIface` sorgt für die Speicherung beziehungsweise Wiederherstellung von Referenzen im `AddedIface`, die vor bzw. nach der De-/Serialisierung eines Arbeitszustandes galten/gelten müssen. Diese Klasse ist zwingend erforderlich, da Referenzen von gleichen Objekten untereinander nicht im jeweiligen Objekt selbst, sondern nur in einer darüberliegenden Instanz aufgelöst werden können. Referenzen werden dadurch gesichert, dass zunächst jedem Objekt und Attribut eine eindeutige ID zugewiesen wird, und im nachfolgenden dann nur noch mit IDs statt den Objekten selbst gearbeitet wird.

5.8 Interface

Im Programm stehen drei verschiedene Interface-Repräsentationen zur Verfügung.

Die erste ist in der Klasse `AddedIface` zu finden. `AddedIface` speichert die vom Nutzer hinzugefügten Informationen (daher auch der Name). Mit Hilfe der Klassen `AddedObject`, `AddedAttrib` und `AddedMaxObj` wird hier bereits im Vorfeld eine Trennung der Daten nach Interface-relevanten Bausteinen vorgenommen. Die Klasse `AddedIface` bietet verschiedene Funktionalität an, mit der die Integrität des Interfaces im Vorfeld bereits gesichert werden kann, zum Beispiel die Sicherstellung der Vergabe eindeutiger Attribut- und Objektnamen. Weiterhin werden hier Duplikate weitestgehend automatisch verwaltet. Von dieser Klasse aus ist es auch möglich die anderen zwei (folgenden) Objektrepräsentationen zu erzeugen.

Die zweite Repräsentation der Daten ist das `MiniIface`. Die Arbeitsweise der Serialisierung wurde bereits in Abschnitt 5.7 beschrieben.

Die dritte Repräsentation der Daten ist in den Klassen `Iface*` enthalten. Die zugehörigen Klassen befinden sich im Paket `iface`. Im Gegensatz zu `AddedIface` sind diese Klassen aus den Anforderungen an das Interface entstanden, und nicht aus den Anforderungen und Gegebenheiten der Anwendung.

Jedes Element des Interfaces ist in einer eigenen Klasse gespeichert. Diese Klassen kapseln die enthaltenen Informationen und ermöglichen den Zugriff nur mittels Methoden. Jede der Klassen hat eine Methode `isValid()`. Diese Methode wird immer dann aufgerufen, wenn ein Ob-

jekt ins Interface eingefügt wird. Wird beispielsweise ein Attribut einem Objekt hinzugefügt, so wird das Attribut nur akzeptiert, falls die `isValid()`-Methode des Attributs `true` zurückgibt. Liefert sie `false`, so wird eine Exception ausgelöst. Diese Exception enthält detaillierte Informationen über den Fehler, so dass der Nutzer gezielt darauf reagieren kann. Das Interface ist somit stets konsistent. Die Klasse `Iface` besitzt eine Methode `toXML()`, die das Interface erzeugt. Die erzeugte XML-Datei wird nicht an der DTD geprüft. Änderungen an der DTD haben keinerlei Auswirkungen auf das Programm, diesen Änderungen muss daher eine Anpassung der Implementierung folgen.

5.9 Logging

Das Projekt URE nutzt Klassen aus dem Paket `java.util.logging` zum protokollieren von Informationen. Dazu wird ein zentrales Objekt zur Verfügung gestellt, das alle (größeren) Komponenten gemeinsam nutzen. Der Nutzer hat so die Möglichkeit über alle Aktionen des Programmes detailliert im Bilde zu bleiben, da bei einigen Datenbankoperationen (mit entsprechend großen Datenbanken) das Programm mitunter einige Sekunden beschäftigt ist (begleitet von der dann üblichen Nervosität beim Nutzer, die immer dann aufkommt wenn sich nichts auf dem Bildschirm bewegt). Die statische Klasse `Logbook` stellt anwendungsweit ein Objekt vom Typ `Logger` zur Verfügung, mit dem alle Komponenten ihre Log-Informationen ausschreiben können. Ein Beispiel für die Log-Ausgabe an der Konsole ist in Abbildung 5.3 zu sehen.

Abbildung 5.3: Logging an der Konsole

```

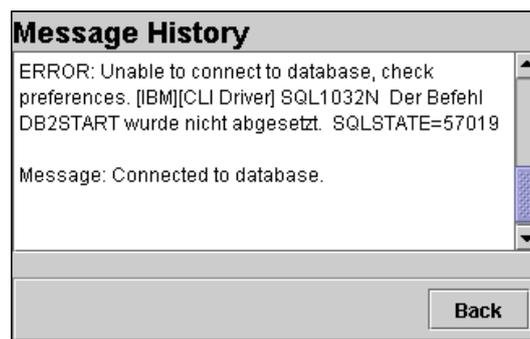
Console [F:\Programme\Java\jdk_1.4.2_04\jre\bin\javaw.exe (27.07.04 09:54)]
09:54:42 (CON) [mad.ure.helper.Configurator.setLogLevel()] Log-Level changed to FINE
09:54:42 (CON) [mad.ure.helper.Configurator.setJDBC_Driver()] JDBC-Driver set to COM.ibm.db2.jdbc.app.DB2Driver
09:54:42 (CON) [mad.ure.helper.Configurator.setJDBC_URL()] JDBC-URL set to jdbc:db2:enotel
09:54:42 (CON) [mad.ure.helper.Configurator.setDB_User()] DB-User set to db2admin
09:54:42 (CON) [mad.ure.helper.Configurator.setDB_Password()] DB-Password set.
09:54:46 (FIN) [mad.ure.helper.Metacollector.retrieveSchemas()] Connection is null
09:54:46 (FIN) [mad.ure.helper.Metacollector.retrieveTables()] Connection is null
09:54:46 (WAR) [mad.ure.gui.MainWindow.statusError()] Status Warning: Unable to connect to database, check preferences. [IBM]C

09:55:23 (FIN) [mad.ure.helper.AddedIface.<init>()] Creating new (empty) mad.ure.helper.AddedIface
09:55:25 (INF) [mad.ure.gui.MainWindow.status()] Status Message: Connected to database.
09:55:25 (FIN) [mad.ure.helper.Metacollector.retrieveSchemas()] Database Query: getSchemas
09:55:28 (FIN) [mad.ure.helper.Metacollector.retrieveTables()] Database query: getTables
09:55:28 (FIN) [mad.ure.helper.Metacollector.retrieveAttribs()] Database query: getColumns(DB2EXT.PROXYINFORMATION)
09:55:28 (FIN) [mad.ure.helper.Metacollector.retrievePrimaryKeys()] Database Query: getPrimaryKeys(DB2EXT.PROXYINFORMATION)
09:55:29 (FIN) [mad.ure.helper.Metacollector.retrieveForeignKeys()] Database query: getImportedKeys(DB2EXT.PROXYINFORMATION)
09:55:29 (FIN) [mad.ure.helper.Metacollector.retrieveUniqueIndices()] Database query: getIndexInfo(DB2EXT.PROXYINFORMATION)

```

Weiterhin speichert die Klasse `Logbook` die letzten 100 Statusmeldungen aus dem Main-Window, die der Nutzer in einer *Message History* im Programm abrufen kann. Abbildung 5.4 zeigt das entsprechende Fenster im Programm.

Abbildung 5.4: Historie des Statusmeldungen



6. Fazit und Ausblick

Die Implementierungsarbeit ist abgeschlossen, das bedeutet das Programm funktioniert entsprechend den geforderten Spezifika. Es ist möglich alle notwendigen Daten zu erfassen und eine XML-Datei zu erzeugen. Alle hier diskutierten Algorithmen wurden implementiert, und arbeiten ohne erkennbare Fehler. Das erzeugte Interface wird bereits in einem Anfrageprozessor benutzt, der im Zuge der Studienarbeit [SD04] entwickelt wird.

Folgende Ideen können in einer Weiterentwicklung aufgegriffen werden:

- Es sollte möglich sein maximale Objekte automatisch zu bestimmen. In [KKFGU84] ist ein solcher Algorithmus beschrieben. Dieser Algorithmus nutzt funktionale und mehrwertige Abhängigkeiten um maximale Objekte aufzubauen.
- Bei Datenbanken, die viele Attribute haben, kann es recht aufwändig werden alle Attribute einzeln zu bearbeiten, manchmal wäre es besser mehrere Attribute gleichzeitig zu erfassen. Technisch ist dies ohne weiteres möglich wenn keine Duplikate erzeugt werden, und der Nutzer mit den vom System vorgeschlagenen Namen für Objekte und Attribute leben kann, denn
- eine Möglichkeit zur nachträglichen Änderung der Attribut- und Objektnamen und deren Eigenschaften wäre als sinnvolle Erweiterung des Programmes denkbar. Genauso wie beim vorigen Punkt ist dafür eine Erweiterung der Nutzerschnittstelle erforderlich.
- Es bleibt zu untersuchen, inwiefern bei der Beziehungsbestimmung die UNIQUE-Eigenschaft eines Attributs implizit mitbestimmt wird. Diese Information könnte dann in die Statistik mit einfließen.
- Der Prototyp wurde nicht unter Usability-Aspekten entwickelt, eine Weiterentwicklung sollte sich daher auch der Optimierung der Nutzerschnittstelle widmen.

7. Quellenangaben

7.1 Hardware

Informationen über Universalrelationen, die Theorie und eine Anwendung (mit Namen System/U) ist, sowie weiterführende Informationen zu maximalen Objekten sind in [MaU183], [FAMU82], [Ull82] und [KKFGU84] zu finden.

Weitere Informationen zur Ermittlung von fehlenden Integritätsbedingungen sind in [Kle96] aufgeführt.

7.2 Software

Bei der Implementierung habe ich auf folgende (frei verfügbare) Software zurückgegriffen:

- Eclipse Workbench (www.eclipse.org):
Dies ist eine Open Source Entwicklungsumgebung für verschiedene Programmiersprachen, und wurde auch von IBM im WSAD (WebSphere Application Developer) ab Version 5 eingesetzt. Eclipse kann mittels eines Plugin-Mechanismus beliebig erweitert werden.
- Visual Editor Project for Eclipse (www.eclipse.org/vcp):
VEP ist ein Plugin für Eclipse, und ermöglicht es dem Programmierer visuelle Klassen mittels Drag- and Drop zusammenzustellen. Zwar ist VEP nicht so komfortabel und effizient wie beispielsweise Visual C++, J Builder oder Delphi, dennoch erleichtert es die Erstellung von SWING-Komponenten um ein Vielfaches gegenüber der „manuellen“ Erstellung.
- JDom (jdom.org):
JDom stellt Java-Klassen zur Verfügung mit denen XML-Dateien sowohl erstellt als auch geparkt werden können. JDom stellt die XML-Daten mit einer einfachen API dar, und ist außerdem speziell auf Java optimiert.
- MySQL (www.mysql.com):
MySQL ist ein frei verfügbares Datenbankmanagement-System. Es ist zwar in Punkten der Funktionalität noch ausbaufähig, ist aber im Web-Umfeld sehr weit verbreitet. Da das

Projekt für Web-Anfragen dienen soll, habe ich auch auf MySQL-spezifische Probleme in der Implementierung Rücksicht genommen.

- CVSNT (www.cvsnt.org):
CVSNT ist eine Windows-Portierung des CVS (Concurrent Versions System).

Literaturverzeichnis

- [MaU183] David Maier, Jeffrey D. Ullman: „Maximal Objects and the Semantics of Universal Relation Databases“, ACM Transactions on Database Systems, Vol.8, No.1, March 1983, pages 1-14.
~/Diss.new/maximal_objects.pdf
- [KKFGU84] H.F.Korth, G.M.Kuper, J.Feigenbaum, A.v.Gelder, J.D.Ullman: „System/U: A Database System Based on the Universal Relation Assumption“, ACM Transactions on Database Systems, Vol.9, No.3, September 1984, pages 331-347.
~/Diss.new/system-u.pdf
- [FAMU82] Ronald Fagin, Alberto O.Mendelzon, Jeffrey D.Ullman: „A Simplified Universal Relation Assumption and Its Properties“, ACM Transactions on Database Systems, Vol.7, No.3, September 1982, pages 343-360.
~/Diss.new/simplified_universal_relation_assumption.pdf
- [Ull82] Jeffrey D.Ullman: „The U.R. strikes back“, Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems, pages 10-22.
~/Diss.new/ur_strickes_back.pdf
- [Kle96] Meike Klettke: „Akquisition von Integritätsbedingungen in Datenbanken“, Dissertationen zu Datenbanken und Informationssystemen (DISDBIS) Band 51, Dissertation vom 12.12.1996, 1998 infix.
- [SD04] Steffen Dumman: „Universalrelationenschnittstelle für Web-Datenbanken - Anfragebearbeitung“, Masterthesis/Studienarbeit, Lehrstuhl Datenbanken und Informationssysteme, September 2004, Institut für Informatik an der Fakultät für Informatik und Elektrotechnik der Universität Rostock.