Masterarbeit

Vergleich von SQL-Anfragen Theorie und Implementierung in Java

Robert Hartmann

Betreuer: Prof. Dr. Stefan Brass

28. August 2013

Martin-Luther-Universität Halle-Wittenberg

Inhaltsverzeichnis

1 Einleitung / Motivation

SQL (structured query language) ist eine Datenbanksprache, die in relationalen Datenbanken zum Definieren, Ändern und Abfragen von Datenbeständen benutzt wird. Basierend auf relationaler Algebra und dem Tupelkalkül, ist sie einfach aufgebaut und ähnelt der englischen Sprache sehr, was Anfragen deutlich verständlicher gestaltet. SQL ist der Standard in der Industrie wenn es um Datenbankmanagementsysteme (DBMS) geht. Zu bekannten Vertretern gehören Oracle Database von der Oracle Corporation, DB2 von IBM, PostgreSQL von der PostgreSQL Global Development Group, MySQL von der Oracle Corporation und SQLServer von Microsoft.

Die Umsetzung von SQL als quasi-natürliche Sprache erlaubt es Anfragen so zu formulieren, dass sie allein mit dem Verständnis der natürlichen Sprache zu verstehen sind. Dieser Umstand hat auch dazu geführt, dass heutzutage relationale Datenbanksysteme mit SQL beliebt sind und häufig eingesetzt werden. Dies führt allerdings auch dazu, dass es mehrere, syntaktisch unterschiedliche, Anfragen geben kann, welche semantisch identisch sind. Einige Anfragen sehen sich dabei ähnlich, andere kann man nur nach Umformen ineinander überführen.

1.1 Motivation

Ein gängiges Mittel, um herauszufinden ob zwei SQL-Anfragen das gleiche Ergebnis liefern, ist es die Anfragen auf einer Datenbank mit vorhandenen Daten auszuführen. Das Ergebnis dieses Vorganges bildet jedoch lediglich Indizien für eine mögliche semantische Gleichheit. Da man die zwei, zu vergleichenden Anfragen nur auf einer endlichen Menge von Datenbankzuständen testen kann, ist nie ausgeschlossen, dass nicht doch ein Zustand existiert, der unterschiedliche Ergebnisse liefert. Weiterhin stehen solche Testdaten nur im begrenzten Umfang zur Verfügung oder Daten müssten händisch eingetragen werden bzw. von freien Internetdatenbanken beschafft werden. Dies kostet Zeit und Arbeitskraft, welche im universitärem Umfeld meist beschränkt ist. So haben Hochschulen immer weniger Geld für Tutoren oder Hilfskräfte, was die Zeit der wenigen Mitarbeiter umso wertvoller macht.

Durch diese Situation sind Professoren immer öfter dazu gezwungen mehr Lehre und weniger Forschung zu betreiben, was aber offensichtlich keine gute Lösung ist. Häufig werden dem Lernenden

Übungsaufgaben gestellt, die dieser dann innerhalb einer Frist bearbeitet und abgibt. Diese müssen dann kontrolliert und wieder ausgehändigt werden. Bei diesem Prozess kann nur schwer auf die einzelnen Fehler der Studenten eingegangen werden. Auch ist es ein zusätzlicher Zeitaufwand herauszufinden, welche Fehler besonders häufig auftreten. Des weiteren sind manche Lernende auch gewillt mehr zu üben, um sich gerüstet für eine Klausur zu fühlen. Andere möchten gezielt ein Thema üben, welches sie noch nicht gut beherrschen. All das ist mit Übungsaufgaben und Übungen innerhalb der Hochschule schwer zu erreichen.

Das Programm, welches im Rahmen dieser Arbeit entwickelt wird, soll helfen all diese Probleme zu lösen. Es soll mit wenig Aufwand für den Mitarbeiter der Universität möglich sein neue Aufgaben in das System einzupflegen. Durch Abspeicherung sämtlicher Lösungsversuche des Lernenden, können einzelne Aufgaben vom Dozenten durch das Programm auf häufig auftretende Fehler untersucht werden. Damit kann in der Übung gezielt besprochen werden, was noch oft falsch gemacht wird.

1.2 Aufgabenstellung

Nach der theoretischen Ausarbeitung soll ein Programm entwickelt werden, welches in der Lage ist zwei SQL-Anfragen zu vergleichen. Dieser Vergleich soll, im ersten Schritt, lediglich mit Musterlösung, Lösung des Lernenden und Datenbankschema möglich sein. Im zweiten Schritt werden die zwei Anfragen auch gegen reale Daten geprüft. Da die Fehlermeldungen des Standardparsers von SQL sehr allgemein gehalten sind, ist es auch wünschenswert, dass das Programm konkretere Hinweis- und Fehlermeldung ausgibt, als es der Standard SQL-Parser vermag. Weiterhin sollen die Fehlermeldung sich konkret auf den Unterschied von Musterlösung zur Lösung des Lernenden beziehen. Damit das Programm möglichst plattformunabhängig bedient werden kann, soll es als Webseite auf einem Server zur Verfügung gestellt werden. Da als Programmiersprache Java gewählt wurde, bieten sich die JSP (java server pages) an, welche auf den java-servlets basieren.

Wir bezeichnen zwei SQL-Anfragen als syntaktisch äquivalent, wenn die zwei Anfragen syntaktisch identisch sind. Dies ist der Fall wenn beide Anfragen als Zeichenketten in jedem Buchstaben übereinstimmen. Zwei SQL-Anfragen sind dagegen semantisch äquivalent, wenn beide Anfragen auf allen Datenbankzuständen einer Datenbank stets die selben Tupel zurückliefern. Zu Bemerken ist hierbei, dass zwei semantisch äquivalente Anfragen keinesfalls syntaktisch äquivalent sein müssen, wie das folgende Beispiel zeigt:

```
Anfrage 1: SELECT * FROM emp WHERE sal > 1000
Anfrage 2: SELECT * FROM emp e WHERE 1000 < e.sal
```

Abbildung 1.1: Beispiel: semantisch, aber nicht syntaktisch, äquivalente Anfragen

Wie bereits beschrieben erfolgt der Vergleich zweier SQL-Anfragen zweistufig. Können beide Anfragen durch Standardisierungen in syntaktisch äquivalente Anfragen umgeformt werden, so sind sie auf jeden Fall auch semantisch äquivalent. Das Umformen durch Standardisierung stellt daher eine hinreichende Bedingung für die Gleichheit der SQL-Anfragen dar. Es gibt aber auch Anfragen, die nicht syntaktisch, aber dennoch semantisch äquivalent sind. Solche Anfragen können in einigen Fällen strukturell sehr unterschiedlich und dennoch semantisch äquivalent sein. Mit dem Vorgehen im ersten Schritt würden wir solche Anfragen nicht aneinander anpassen. In diesem Fall fahren wir mit dem zweiten Schritt fort. In diesem werden beide Anfragen auf realen Datenbanken ausgeführt. Sind beide Anfragen semantisch äquivalent, so müssen beide die selben Ergebnistupel liefern. Ist dies nicht der Fall, so ein Gegenbeispiel für die semantische Äquivalenz gefunden. Das Liefern von gleichen Ergebnissen beider Anfragen auf dem selben Datenbankzustand ist also eine notwendige Bedingung für die semantische Äquivalenz. Im Kapitel 6 wird noch auf Anfragen eingegangen, die weder im Schritt 1 bestätigt noch im Schritt 2 abgelehnt werden konnten.

Ein Haupteinsatzgebiet ist die Lehre. Dort soll es möglich sein Untersuchungen des Lernfortschritts von Studenten oder anderen Interessierten, die den Einsatz von SQL erlernen möchten, durchzuführen. So kann das Programm dem Lernenden nicht nur sinnvolle Hinweise bei einer falschen Lösung geben, sondern auch erläutern, ob die gefundene Lösung eventuell zu kompliziert gedacht war. Des Weiteren ist es aufgrund der zentralisierten Serverstruktur möglich, Lösungsversuche des Lernenden zu speichern und eine persönliche Lernerfolgskurve anzeigen zu lassen. Damit hätten Studenten und Lehrkräfte die Möglichkeiten Lernfortschritte zu beobachten und Problemfelder (etwa JOINs) zu erkennen um diese dann gezielt zu bearbeiten. Dozenten könnten so im Zuge der Vorbereitung der Übung oder Vorlesung sich die am häufigsten aufgetretenen Fehler anzeigen lassen, um diese dann mit den Studenten direkt zu besprechen.

Damit ist es möglich eine Lernplattform aufzubauen, die dem Studenten mehrere Auswertungsinformationen über seinen Lernerfolg und seine Lösung deutlich macht. So kann die Lehrkraft eine Aufgabe mit samt Musterlösung und Datenbankschema hinterlegen und der Student kann daraufhin seine Lösungsversuche in das System eintragen. Durch sinnvolles Feedback ist es dem Lernenden möglich, beim Üben leichter und effizienter zu lernen. Weiterhin kann man eine solche Plattform auch für Tutorien oder Nachhilfe benutzen, sowie überall dort, wo SQL gelernt wird. Vorteile hier wären, dass man mehrere verschiedene Aufgaben stellen kann ohne viel Zeit beim Einpflegen von neuen Aufgaben verbringen zu müssen.

1.3 Aufbau der Arbeit

Im vorherigen Abschnitt haben wir geklärt warum es eine Notwendigkeit für das Thema SQL-Vergleich gibt. Zu dem wurde geklärt, was das Ziel der Arbeit ist.

Im Kapitel 2 betrachten wir den aktuellen Forschungsstand zur Thematik Lernplattformen und SQL. Das Ergebnis dieser Arbeit soll ein Produkt sein, was hauptsächlich in der Lehre eingesetzt wird. Daher ist es wichtig bereits vorhandene Lernplattformen zu untersuchen. Dabei interessieren uns insbesondere Gemeinsamkeiten und Unterschiede zu unserer Arbeit. Wir werden feststellen, dass jede Plattform auf einen, bestimmten Aspekt spezialisiert ist und andere Aspekte dann eine untergeordnete Rolle spielen. Weiterhin ist diese Bestandsaufnahme wichtig, da sie uns aufzeigen kann, wie man mögliche Ansätze miteinander verknüpft. Dieser Gedanke wird im Kapitel 6 genauer erläutert.

Im nachfolgenden Kapitel 3 beschreiben wir die verwendete Software. Zunächst wird der verwendete Parser ausführlich erläutert. Dabei gehen wir nicht nur auf seine genaue Funktionsweise, sondern auch auf seine Schwächen ein. Diese ziehen gewisse Einschränkungen mit sich, mit denen wir uns in nachfolgenden Teilen der Arbeit befassen werden. Des weiteren wird die Funktionsweise von Java Server Pages (JSP) kurz umrissen. Am Ende des Kapitels 3 listen wir die verwendete Software auf, um nachzuvollziehen wie unsere Anwendung entstanden ist.

Die Fragestellung “Sind zwei SQL-Anfragen äquivalent” ist nicht entscheidbar. Aus diesem Grund klären wir im Kapitel 4 wie wir den Entscheidungsprozess angehen wollen, sodass zumindest eine Teilmenge von SQL-Anfragen bearbeitet werden kann. Es wird zunächst eine Methode entwickelt, die zwei SQL-Anfragen auf semantische Äquivalenz prüfen soll. Danach werden die einzelnen Schritte dieser Methode ausführlich diskutiert. Dabei diskutieren wir auch alternative Ansätze und Einschränkungen, die durch verwendete Software oder durch erhöhten Komplexitätsgrad entstehen. Ansätze, die wir Aufgrund erhöhter Komplexität nicht in der Anwendung unterbringen können, werden dennoch ausführlich erläutert, so dass eine spätere Implementierung leicht umzusetzen ist.

In Kapitel 5 wird der konkrete Aufbau unserer Anwendung erläutert. Dabei klären wir die verwendete Datenbankstruktur, sowie den Programmablauf für die wichtigsten Hauptprozesse. Dieses Kapitel dient, vor allem, zum Verständnis der Umsetzung, der in Kapitel 4 erläuterten Methoden.

1.4 Produkt

Im Rahmen der Aufgabenstellung wird ein Programm entwickelt, das es erlaubt zwei SQL-Anfragen auf semantische Äquivalenz zu vergleichen. Dazu wird eine Lernplattform auf Basis von Java Server Pages geschaffen. Der Lernende meldet sich über ein Webformular bei der Plattform an. Auf der Startseite werden ihm seine letzten Aktionen auf der Plattform angezeigt. Weiterhin findet er eine Statistik über bereits behandelte Aufgaben. Der Lernende kann sich aus einer Liste von verschiedenen Aufgaben eine auswählen. Er erhält dann das konkrete Datenbankschema, so wie eine Textaufgabe, die ihm die Aufgabenstellung vermittelt. Nach Einreichen einer Lösung wird

ausgewertet, ob diese mit der Musterlösung gematcht werden konnte und ob eine semantische Äquivalenz sicher ausgeschlossen werden konnte. Das Ergebnis dieser Auswertung wird dem Lernenden, zusammen mit Hinweisen zu seiner Lösung, ausgegeben.

Der Dozent hat die Möglichkeit einfach und schnell weitere Aufgaben in das System einzupflegen. Dazu muss er eine Aufgabenstellung in Textform, sowie mindestens eine SQL-Anfrage als Musterlösung hinterlegen. Damit die Anfrage des Lernenden und die Musterlösung im zweiten Schritt unserer Anwendung auch gegen echte Daten geprüft werden können, bedarf es noch der Angabe von mindestens einer externen Datenbank. Diese externen Datenbanken können durch ein Webformular eingetragen werden.

2 Forschungsstand und Einordnung

2.1 Einleitung

Die Idee, SQL-Anfragen von Lernenden automatisch zu kontrollieren, ist nicht völlig neu. Weil eine Auswertung über den Standard SQL-Parser nicht sehr umfangreich ist und dieser semantischen Fehlern kein sinnvolles Feedback gibt, sind bereits einige Ansätze veröffentlicht worden, die es sich zum Ziel gemacht haben eine SQL-Anfrage näher zu analysieren. Verschiedene Projekte beschäftigen sich dabei z. B. mit dem Aufdecken von semantischen Fehlern. Andere Plattformen konzentrieren sich auf den Lernerfolg, den der Student erreichen soll, und analysieren die Art der Fehler des Lernenden. Damit erreicht man eine Zuteilung von passenderen Aufgaben, sodass der Lernende weder gelangweilt noch überfordert ist.

In diesem Abschnitt möchten wir die bereits existierenden Ansätze auf diesem Gebiet betrachten, um dann diese Arbeit einordnen zu können.

2.2 SQL-Tutor

In [3] beschreibt Antonija Mitrovic ein Lernsystem, das SQL-Tutor genannt wird. Nach Auswahl einer Schwierigkeitsstufe wird dem Studenten ein Datenbankschema und eine Textaufgabe vorgelegt. Der Student hat nun ein Webformular vor sich, in dem sich für jeden Teil der SQL-Anfrage ein Eingabefeld befindet. So werden die Anteile `SELECT`, `FROM`, `WHERE`, `ORDER BY`, `GROUP BY` sowie `HAVING` einzeln eingetragen.

Der SQL-Tutor analysiert nun die Anfrage des Studenten und gibt spezifisches Feedback. Dabei wird nicht nur geklärt, ob die Anfrage korrekt ist, sondern auch, was bei einer falschen Eingabe genau fehlerhaft ist. Das reicht von konkreten Hinweisen auf den spezifischen Teil der Anfrage bis hin zu eindeutigen Hinweisen wie »Musterlösung enthält einen numerischen Vergleich mit der Spalte a, ihre Lösung enthält aber keinen solchen Vergleich«.

Umgesetzt wird dieses Programm durch 199 fest einprogrammierte Constraints. Dadurch ist es potentiell möglich bis zu 199 spezifische Hinweismeldungen für den Studenten bereitzustellen. Das reicht von syntaktischen Analysen wie »Der SELECT-Teil von einer Lösung darf nicht leer

sein.« bis hin zu semantischen Analysen, gepaart mit Wissen über die Domain (Datenbankschema und Musterlösung), bei denen die Lösung des Studenten mit der Musterlösung und dem Datenbankschema verglichen wird. Insbesondere versucht der SQL-Tutor Konstrukte wie numerische Vergleiche mit gewissen Operatoren in der Lösung des Studenten zu finden, wenn diese in der Musterlösung auftauchen. Auch komplexere Constraints, die sicherstellen, dass bei einem numerischen Vergleich `a > 1` das gleiche ist wie `a >= 0` sind vorhanden.

Allerdings gibt es auch hier Schwächen. Da der verwendete Algorithmus die Constraints nach einander abarbeitet, kann es zu unnötigen Analysen der Anfrage kommen und damit auch zu einem unnötigen Zeitaufwand. Nach eigenen Tests werden manche äquivalente Bedingungen nicht erkannt. So wird `a < 0` für richtig, aber `0 > a` für falsch gehalten. Ähnlich verhält es sich, falls eine der Argumente des Vergleichs das Ergebnis einer Unterabfrage ist. Die Constraints sind fest einprogrammiert und nicht von der Anfrage abhängig und damit genügt es für eine neue Aufgabe Text und Musterlösung einzulesen.

Der SQL-Tutor lässt außerdem den eingesendeten Lösungsvorschlag auf einer SQL-Datenbank mit Testdaten laufen und vergleicht die Tupel mit den Antworttupeln, die man mit der gespeicherten Musterlösung erhält.

Abgrenzung zum SQL-Tutor

Der Grundgedanke des SQL-Tutors überschneidet sich durchaus mit dem Ansatz dieser Arbeit. Ein Grundpfeiler des SQL-Tutors ist es, dem Studenten detailliertes Feedback über seine semantischen und syntaktischen Fehler zu geben. Das Programm was im Zuge dieser Arbeit entsteht soll weniger semantische Fehler analysieren, als viel mehr versuchen zwei SQL-Anfragen zu vergleichen, unabhängig davon wie sie aufgeschrieben sind. Des Weiteren bedient sich der SQL-Tutor einer Testdatenbank mit realen Testdaten. Unser Programm soll nur das Datenbankschema kennen und ohne Daten bestimmen, ob zwei Anfragen das gleiche Ergebnis liefern. Erst in einem optionalen zweiten Schritt prüfen wir in unserem Programm, ob beide Anfragen gleiche Ergebnisse auf konkreten Daten liefern. Hat bereits der erster Schritt ein positives Ergebnis gemeldet, dann ist der zweite Schritt unnötig. Der SQL-Tutor operiert allerdings zur Ermittlung der Übereinstimmung nur auf Testdaten. Bei ungünstig gewählten Testdaten kann es passieren, dass der Eindruck entsteht zwei Anfragen wären gleich, obwohl sie es nicht sind. Entstehen kann dies, weil auf den vorhanden Testdaten zufällig beide Anfragen die gleichen Ergebnisse liefern könnten.

2.3 SQL-Exploratorium

Im Artikel [4] werden SQL-Lernplattformen in zwei Kategorien eingeteilt. Die erste Kategorie zeichnet sich durch existieren Plattformen aus, welche durch Multimedia versuchen dem Lernenden einzelne Bestandteile der Sprache SQL bildlich darzustellen. Hierfür werden meist Websites mit Multimediainhalten erstellt. Die zweite Kategorie beinhaltet Software, welche die Lösung eines Lernenden analysiert und konkrete Hinweismeldungen gibt. Dazu zählt auch der eben beschriebene SQL-Tutor.

Das SQL-Exploratorium macht es sich nun zur Aufgabe die beiden Ansätze zu verbinden und stellt sich dabei hauptsächlich verwaltungstechnische Fragen wie z. B.

- Wie ermögliche ich dem Studenten Zugriff auf verschiedene Lernsysteme, ohne sich mehrfach einloggen zu müssen?
- Wie können Lernerfolge in einem System einem anderen nutzbar gemacht werden?
- Wie kann man aus mehreren Logfiles der eingereichten Lösungen eines Studenten, von unterschiedlichen Systemen, einen Wissensstand des Studenten ableiten?

Da diese Fragen wenig Relevanz für diese Arbeit haben, betrachten wir im Folgenden welche einzelnen Plattformen für das SQL-Exploratorium genutzt werden.

2.3.1 Interactive Examples

Über eine Schnittstelle, die sich WebEX nennt, hat der Student Zugriff auf insgesamt 64 Beispielanfragen. Wählt man eine Anfrage aus können Teile davon in einer Detailansicht geöffnet werden. Dem Studenten wird dann ausführlich erklärt, was die einzelnen Teile der Anfrage genau bewirken. Sowohl die Beispielanfragen, als auch die Hinweise sind manuell erzeugt und abgespeichert. Hier wird nichts automatisch generiert, daher ist dieses Projekt nicht relevant für die Arbeit. Der Lernerfolg des Studenten wird hier über die ein »click-log« geführt, das bedeutet es wird aufgezeichnet, was der Student wann und in welcher Reihenfolge angeklickt hat. So ist es z. B. möglich herauszufinden welche Teile einer bestimmten Anfrage besonders interessant für den Lernenden sind.

2.3.2 SQL Knowledge Tester

Der SQL Knowledge Tester, im Nachfolgendem SQL-KnoT genannt, konzentriert sich darauf Anfragen eines Studenten zu analysieren. Dabei wird dem Studenten zur Laufzeit eine Frage generiert. Dabei werden vorhandene Datenbankschemata in einer bestimmten Art und Weise verknüpft

und Testdaten sowie eine Frage für den Studenten generiert. Dies geschieht mit fest einprogrammierten 50 Templates, die in der Lage sind über 400 Fragen zu erzeugen. Zu jeder Frage werden zur Laufzeit Testdaten für die relevanten Datenbanken erzeugt. Ausgewertet wird die Anfrage des Studenten dann, indem die zurückgelieferten Tupel der Studentenanfrage verglichen werden mit den Tupeln, welche die Musterlösung erzeugt.

Abgrenzung zur Arbeit

Erwähnenswert ist, dass initial keine Daten existieren. Wie beim Ansatz dieser Arbeit existieren zunächst nur Datenbankschemata. Die Daten und auch die Aufgabe an den Studenten werden aus Templates generiert. Die Auswertung erfolgt dann allerdings durch den Vergleich der zurückgelieferten Tupel der Muster- und Studentenanfrage. Hierbei kann wieder das Problem auftreten, dass für beide Anfragen die erzeugten Testdaten die gleichen Tupel zurückliefern, es aber bei einem anderen Zustand sein kann, dass sich die Tupelmengen unterscheiden.

Der Ansatz vom SQL-KnoT ist durchaus interessant, wird aber in dieser Arbeit nicht weiter ausgeführt, da wir in dieser Arbeit keine Testdaten erzeugen möchten. Wir benutzen vielmehr Beispieldaten als optionalen zweiten Schritt.

2.3.3 Weiteres

Adaptive Navigation for SQL Questions

Hierbei handelt es sich nur um ein Tool, was aufgrund früherer Antworten des Studenten, diesem möglichst passende neue Fragen vorlegen soll. Dieser Teil des SQL-Exploratoriums dient also dazu, den Wissensstand des Studenten festzustellen und ist für diese Arbeit daher unerheblich.

2.4 WIN-RBDI

Das Programm WINRBDI, welches in [5] beschrieben wird verfolgt einen weiteren, interessanten Ansatz. Anstelle von fest vorgegebenen Demoanfragen, wird die eingegebene Anfrage zunächst in esql eingebettet. Die Ausführung der Anfrage wird dann schrittweise durchgeführt. Der Student hat also die Möglichkeit die Anfrage im Schrittmodus, ähnlich eines Debugger, oder im Fortsetzen-Modus auszuführen. Im Schrittmodus wird jeder Teilschritt der Abarbeitung der Anfrage aufgezeigt. Es werden temporär erzeugte Tabellen angegeben sowie auch eine Erklärung welcher Teil der Anfrage für den aktuellen Abarbeitungsschritt verantwortlich ist. So soll es dem

Studenten möglich sein, die unmittelbaren Konsequenzen seiner SQL-Anfrage für die Abarbeitung zu begreifen.

Des Weiteren hilft dieser Ansatz dem Studenten die Abarbeitung einer Anfrage zu Visualisieren, indem die von der WHERE-Klausel betroffene Spalten markiert werden. Dies hilft gerade Lernanfängern bei der Visualisierung von Konzepten wie JOINs.

Abgrenzung zur Arbeit

Dieser Ansatz hebt sich von den bisherig betrachteten Ansätzen ab. Hier wird dem Studenten durch eine Visualisierung der Ausführung der Anfrage versucht deutlich zu machen, welche Teile der formulierten Anfrage was genau bewirken. Für den Lernerfolg des Studenten ist dies sicherlich hilfreich, zumal eine Visualisierung stets hilft, Zusammenhänge leichter zu begreifen. Diese Arbeit verfolgt allerdings ein anderes Ziel, da sie zwei SQL-Anfragen miteinander vergleicht und nicht versucht die Abarbeitung einer Anfrage verständlicher zu machen.

2.5 SQLLint

»SQLLint«, ein Semantik-Prüfer für SQL-Anfragen, beschäftigt sich mit semantischen Fehlern in SQL-Anweisungen, welche unabhängig vom Datenbankzustand auftreten. Dabei behandelt das Projekt Anfragen, von denen man ohne Kenntnis der Aufgabenstellung sagen kann, dass sie, in der vorliegenden Form, nicht beabsichtigt sind. Dies ist wahrscheinlich, wenn man z. B. Teile aus der Anfrage herausstreichen kann ohne die Funktion der Anfrage zu verändern. Das Problem besteht darin, dass aktuelle DBMS-Systeme solche Anweisungen ohne Fehler- oder Warnmeldung ausführen. Der Nutzer, also insbesondere der lernende Nutzer, ist somit kaum in der Lage überhaupt zu bemerken, dass es einen Fehler in seiner Anfrage gab. Die allgemeine Frage, ob eine Anfrage semantische Fehler enthält ist nicht entscheidbar. Dennoch macht es sich SQLLint zur Aufgabe eine große, typische Teilmenge von SQL-Anfragen zu bearbeiten. Ziel des Projektes ist es, mit semantischen Warn- und Fehlermeldungen, die Codeentwicklung zu beschleunigen und die Anzahl der Fehler darin zu verringern.

Eine wichtige Zielstellung des Projektes ist es, solche Fehlermeldungen in der Lehre einzusetzen. In [6] wird auch deutlich gemacht, das eine Motivation dieses Projektes aus typischen Fehlern von Studenten entspringt. So wurde im selben Artikel aufgeführt, dass semantische Fehler bei Lernenden am häufigsten auftreten. Unter den drei häufigsten semantischen Fehlern befinden sich: fehlende JOIN Bedingung, (zu) viele Duplikate, unnötiger JOIN. Diese Fehler machen bereits ca. 37 Prozent der semantischen Fehler aus.

Weiterhin fällt auf, dass die Anzahl syntaktischer Fehler mit fortschreitendem Schwierigkeitsgrad der SQL-Anfrage steigen, aber die Anzahl semantischer Fehler nahezu unabhängig von jenem Schwierigkeitsgrad ist. Einfache Anfragen haben sogar zwei mal mehr semantische Fehler als syntaktische Fehler. Siehe dazu Abbildung 4 in [6].

2.5.1 Algorithmus zum Finden von inkonsistenten Bedingungen

Die Algorithmen im SQLLint-Projekt sollen inkonsistente Bedingungen finden. Dieses Problem ist allerdings im Allgemeinen unentscheidbar. Dennoch ist es möglich Teilmengen von Anfragen anzugeben, für die man die Konsistenz algorithmisch Entscheiden kann. Folgende Ausführungen zum Algorithmus entstammen der Arbeit »Proving the Safety of SQL Queries« von Stefan Brass und Christian Goldberg [8].

Konsistenz im diesem Sinne soll bedeuten, dass es ein endliches Modell (relationaler Datenbankzustand, manchmal auch Datenbankinstanz genannt) existiert, sodass das Ergebnis der Anfrage nicht leer ist.

Wir nehmen im Folgenden an, dass die SQL-Anfragen keine Datentyp Operationen enthalten. Alle atomaren Formeln haben also die Form $t_1 \theta t_2$ mit $\theta \in \{=, <>, <, <=, >, >=\}$ und t_1, t_2 sind Attribute oder Konstanten. Aggregationsfunktionen sind noch Bestandteil der Forschung und werden daher nicht behandelt.

2.5.2 Bedingungen ohne Unteranfragen

WHERE-Bedingungen, die keine Unteranfrage enthalten, können mit bestimmten Methoden entschieden werden. Ein Beispiel dafür sind die Algorithmen von Guo, Sun und Weiss [7]. Als erster Schritt wird die Negation `NOT` so weit wie möglich an die atomaren Formeln weitergereicht, indem die `DE-MORGAN`'sche Regeln angewendet werden. Dadurch drehen sich die Vergleichsoperatoren um, wir sprechen hierbei von dem »gegenteiligem Operator«. Die Menge $O = \{\{\leq, >\}, \{\geq, <\}, \{=, =\}, \{\neq, \neq\}\}$ enthält jeweils 2er Mengen von einem Operator und seinem »gegenteiligem Operator«. Im nächsten Schritt wird die Bedingung in die disjunktive Normalform (DNF) umgeformt, so dass folgende Struktur entsteht: $\phi_1 \vee ... \vee \phi_n$. Diese ist genau dann konsistent, wenn mindestens ein ϕ_i konsistent ist. Nun können wir die Methoden aus [7] anwenden. Im wesentlichen handelt es sich dabei um einen gerichteten Graphen, in dem Knoten markiert sind mit »Tupelvariable.Attribut« und Kanten mit $<$ oder $\leq$. Dann werden Intervalle von möglichen Werten für jeden Knoten berechnet. Dabei ist zu beachten, dass die SQL-Datentypen, wie `NUMERIC(1)`, das Intervall zusätzlich einschränken. Wenn es endlich viele mögliche Werte für einen Knoten gibt, dann können Ungleich-Bedingungen ($t_1 <> t_2$) zwischen Knoten wichtig werden und ein Graphfärbungsproblem kodieren. Daher erwarten wir keinen effizienten Algorithmus, wenn es viele <>

Bedingungen gibt. In allen anderen Fällen ist die Methode in [7] schnell. Anzumerken ist allerdings noch, dass die Umwandlung in DNF zu exponentiellem Wachstum in der Größe der Anfrage führen kann.

2.5.3 Unteranfragen

Um unnötige Betrachtungen zu vermeiden, beschäftigt sich das SQLLint-Projekt nur mit `EXISTS`-Unteranfragen. Alle anderen Unteranfragen (`IN`,`>=ALL`, etc.) können auf die `EXISTS`-Unteranfrage reduziert werden. Oracle führt solche Umwandlungen durch, bevor der Optimierer beginnt an der Anfrage zu arbeiten.

Die Idee zur Behandlung von Unteranfragen stammt aus bekannten Methoden der automatischen Beweiser. Hierzu wird in der Arbeit [8] eine Variante der Skolemisierung vorgestellt. Das genaue Vorgehen wird in in jenem Artikel erklärt.

2.5.4 Unnötige logische Komplikationen

Es kann vorkommen, dass eine Teilbedingung inkonsistent ist, die gesamte Bedingung allerdings dennoch konsistent ist (Aufgrund der Disjunktion). Ebenso denkbar ist der umgekehrte Fall, dass also Unterbedingungen Tautologien sind. Beide Vorkommnisse sind vermutlich nicht gewollt und können zu einem unerwünschte Verhalten einer Anfrage führen. Wie in [9] festgestellt wurde, werden in Klausuren von Studenten auch öfter unnötige Bedingungen angegeben, welche bereits per Definition impliziert werden. Als Beispiel betrachten wir die Bedingung `A IS NOT NULL`. Diese wird unnötig, wenn wir wissen, dass `A` bereits als `NOT NULL` definiert ist.

Im Folgenden wird in [9] eine mögliche Formalisierung der Voraussetzung für »keine unnötigen logischen Komplikationen« erläutert. Immer wenn in der DNF der Anfragebedingung eine Unterbedingung mit »true« oder »false« ersetzt wird, ist das Ergebnis nicht zur Ausgangsbedingung äquivalent.

Realisiert wird dies durch eine Reihe von Konsistenzprüfungen. Es sei die DNF der Anfragebedingung $C_1 \vee ... \vee C_m$, mit $C_i = (A_{i,1} \wedge ... \wedge A_{i,n_i})$. Unser Kriterium ist genau dann erfüllt, wenn die folgenden Formeln alle konsistent sind:

1. $\neg(C_1 \vee ... \vee C_m)$ - Die Negation der gesamten Formel. Ansonsten könnte man diese durch »true« ersetzen.
2. $C_1 \wedge \neg(C_1 \vee ... \vee C_{i-1} \vee C_{i+1} \vee ... \vee C_m)$ mit $i \in [1,m] \cap \mathbb{N}$. Ansonsten könnte C_i mit »false« ersetzt werden.

3. $A_{i,1} \wedge ... \wedge A_{i,j-1} \wedge \neg A_{i,j} \wedge A_{i,j+1} \wedge ... \wedge A_{i,n_i} \wedge \neg(C_1 \vee ... \vee C_{i-1} \vee C_{i+1} \vee ... \vee C_m)$ mit $i \in [1,m] \cap \mathbb{N}$, $j \in [1,n_i] \cap \mathbb{N}$. Ansonsten könnte $A_{i,j}$ mit »true« ersetzt werden.

Zu weiteren unnötigen logischen Komplikationen zählen zu allgemeine Vergleichsoperatoren ($>=$ anstelle von $=$). Weiterhin gehören unnötige JOINs zu einem wichtigen Typ von unnötigen logischen Komplikationen.

2.5.5 Laufzeitfehler

Als Bemerkung ist festzuhalten, dass sich das SQLLint-Projekt auch mit Laufzeitfehlern beschäftigt. Als Beispiel stelle man sich folgende SQL-Bedingung vor: `A=(SELECT ...)` Es muss hier sichergestellt werden, dass die `SELECT`-Unteranfrage nur einen Rückgabewert hat. Solche Fehler sind schwierig zu finden, da sie nicht immer Auftreten müssen.

Wie das SQLLint-Projekt damit umgeht, soll hier nicht weiter besprochen werden. Details dazu sind zu finden in [9].

Zusammenhang zu dieser Arbeit

Obwohl SQLLint auf den ersten Blick eine andere Zielstellung als diese Arbeit verfolgt, so sind doch einige Ansätze deckungsgleich. Einige der Ansätze von SQLLint können Grundlagen für diese Arbeit sein. Der Ansatz der Standardisierung der SQL-Anfragen ist mit Umwandeln der Formeln in eine Normalform ein guter Ansatzpunkt und zeigt wie sich komplexe Bedingungen vereinheitlichen lassen. Auch die Erkenntnis, dass sich alle Unteranfragen auf `EXISTS` Unteranfragen reduzieren lassen, wird helfen die Unteranfragen zu standardisieren. Dadurch wird die Vielfalt der Unteranfragen eingeschränkt und ein Vergleich zweier SQL-Anfragen wird vereinfacht.

Weiterhin könnte man in späteren Ausbaustadien des Programmes, welches im Rahmen dieser Arbeit entsteht, die Funktionalitäten des SQLLint einbeziehen . Dies würde die Art des Feedbacks für den Lernenden deutlich verbessern, da wir uns in dieser Arbeit zunächst auf das Vergleichen von zwei SQL-Anfragen konzentrieren. Dabei stehen vor allem Hinweise im Vordergrund, die dem Lernenden zeigen sollen, warum seine Lösung mit der Musterlösung noch nicht übereinstimmen kann.

Ein davon unabhängiges Feedback für die Anfrage des Lernenden würde den Lernverlauf stark beschleunigen und mit hoher Wahrscheinlichkeit sogar die Fehler der Anfrage eliminieren, so dass die Anfrage dann auf die Musterlösung passt.

3 Verwendete Software

3.1 SQL-Parser

3.1.1 über den SQL-Parser: ZQL

Auf der Webseite vom [1] Projekt ist der Open-Source-Parser ZQL zu finden, welcher in der Lage ist SQL zu parsen und in Datenstrukturen zu überführen. Der Parser selbst ist mit [2] geschrieben, einem Java-Parsergenerator (zu vergleichen mit dem populärem Unix yacc Generator).

ZQL bietet Unterstützung für `SELECT`-, `INSERT`-, `DELETE`-, `COMMIT`-, `ROLLBACK`-, `UPDATE`-, `SET`- und `S TRANSACTION`-Ausdrücke. Wichtig für diese Arbeit sind dabei insbesondere `SELECT`- und `UPDATE`-Ausdrücke, sowie – die leider nicht enthaltenen – `CREATE TABLE`-Ausdrücke.

3.1.2 Funktionsweise des Parsers

ZQL kennt zwei grundlegende Interfaces `ZExp` und `ZStatement`. Wichtig für die weiterführende Erklärung ist, dass genau drei Klassen das Interface `ZExp` implementieren. Diese werden im Folgenden auch vorgestellt. Es handelt sich um `ZExpression, ZConstant, ZQuery`.

Das Interface `ZStatement` bildet eine abstrakte Oberklasse für alle möglichen Arten von SQL-Statements. Folgende Klassen implementieren dieses Interface in ZQL:

- `ZDelete` - repräsentiert ein `DELETE` Statement
- `ZInsert` - repräsentiert ein `INSERT` Statement
- `ZUpdate` - repräsentiert ein `UPDATE` Statement
- `ZLockTable` - repräsentiert ein `SQL LOCK TABLE` Statement
- `ZQuery` - repräsentiert ein `SELECT` Statement

Das Interface `ZExp` bildet eine abstrakte Oberklasse für drei verschiedene Arten von Ausdrücken:

- `ZConstant` - Konstanten vom Typ `NULL, NUMBER, STRING` oder `UNKNOWN`. Dazu gehören auch Spaltennamen. Dies sind keine Konstanten im eigentlichen Sinne, sondern vielmehr im

Sinne eines Parserbaumes. Im ZQL-Parserbaum sind Elemente entweder eine SQL-Anfrage (Query), ein Ausdruck, bestehend aus Operator und Operanden, oder eine Konstante. Spaltennamen fallen in die Kategorie Konstanten, da sie keine Ausdrücke oder SQL-Anfragen sind. Der Typ hierfür ist dann `COLUMNAME`.

- `ZExpression` - Ein SQL-Ausdruck bestehend aus einem Operator und einen oder mehreren Operanden
- `ZQuery` - Eine `SELECT` Anfrage ist auch ein Ausdruck

Wir klären in diesem Abschnitt nun genauer, wie der Parser SQL-Anfragen in interne Datenstrukturen überführt. Dazu müssen die Datentypen des Parserpaketes zunächst erläutert werden. Anschließend stellen wir eine Beispielanfrage und ihre Überführung in die Datenstrukturen des Parsers vor. Daraus leiten wir auch die Schwächen bzw. Grenzen des Parsers ab. Wir stellen im Folgenden dar, was für relevante Attribute und Methoden die jeweiligen Klassen besitzen. Da viele Klassen durch Vererbung entstehen, werden wir der Übersicht halber auch abgeleitete relevante Attribute und Funktionen mit einbeziehen.

Die Klasse **ZFromItem** bezeichnet genau eine Tupelvariable samt Relation. Gespeichert wird in dieser Klasse daher der *Name der Relation* und eine mögliche Tupelvariable als *Alias*.

Ähnlich ist die Klasse **ZSelectItem** aufgebaut. Sie speichert den Namen des Attributs und einen eventuell angegebene Tupelvariable. Weiterhin wird festgehalten, ob das SelectItem ein Ausdruck ist wie z. B. in `SELECT a+b`. In diesem Fall würde man über die Funktion `getExpression()` ein Objekt einer Klasse erhalten, das das Interface `ZExp` implementieren muss. Zu beachten ist dabei, dass ein Ausdruck in ZQL immer einen Operator und mindestens einen Operanden besitzen muss. Deswegen fallen Spaltennamen nicht in diese Kategorie, sondern sind unter Konstanten zu finden. Weiterhin wird gespeichert, ob es sich um eine Wildcard * handelt oder ob das Item eine Aggregatesfunktion ist. Bei `SELECT AVG(sal) FROM ...` würde der Name des Items `sal` sein und die Aggregatsfunktion ist dann `AVG`.

Der `WHERE`-Teil wird dargestellt mit Hilfe der Klasse **ZExpression**. Diese Klasse implementiert das Interface `ZExp`. Ein Objekt der Klasse `ZExpression` besteht immer aus einem Operator und mehreren Operanden. Ein Operator ist ein gültiger String. Ein Ausdruck speichert seine Operanden als *Vector* von Objekten, die das Interface `ZExp` implementieren. Demnach kann ein Operand vom Typ `ZExpression, ZConstant` oder `ZQuery` sein. Wichtig ist auch zu bemerken, dass ZQL nicht nur zwei Operanden pro Ausdruck kennt.

Ein üblicher Syntaxbaum ist binär, wobei die Wurzel den Operator mit der höchsten Priorität darstellt. Alle Teilbäume sind als Ausdrücke zu verstehen, jeweils mit Operator als Wurzelknoten und Operanden als Kindknoten. Dabei kann ein Operand auch ein weiterer Ausdruck sein. Generell wird dabei das Prinzip der Assoziativität benutzt um z. B. für gleichrangige Operatoren eine Auswertungsreihenfolge festzulegen.

So würde der `WHERE`-Teil von folgender SQL-Anfrage:

```
SELECT * FROM emp e WHERE e.sal > 1000 AND e.sal < 2000 AND e.id > 1234
```

zu folgendem, geklammerten Ausdruck werden:

```
(((e.sal > 1000) AND (e.sal < 2000)) AND (e.id > 1234))
```

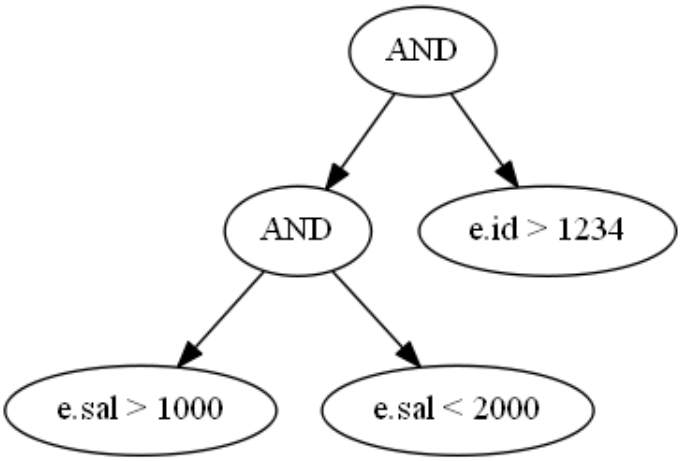


Abbildung 3.1: WHERE-Bedingung in üblichen Syntaxbäumen

Der ZQL-Parser funktioniert so allerdings nicht. Wird keine spezielle Klammerung benutzt, so werden gleichrangige Operatoren nicht assoziativ geklammert, sondern befinden sich auf einer Ebene des Baumes. Somit handelt es sich nicht um einen binären Baum.

Wir erhalten also aus obigen `WHERE`-Ausdruck:

```
((e.sal > 1000) AND (e.sal < 2000) AND (e.id > 1234))
```

So erklärt sich auch die Verwendung eines *Vector* zur Speicherung der Operanden.

Die Klasse **ZConstant** dient zur Darstellung von SQL-Konstanten. Sie speichert den Wert der Konstante sowie den Typ. Als Typen kommen in Frage: `NULL, NUMBER, STRING, UNKNOWN`. Da die Typen selbsterklärend sind, wird auf eine detaillierte Beschreibung verzichtet. Wie bereits erwähnt zählen auch Spaltennamen zu Konstanten, sie erhalten den Typ `COLUMNNAME`. Begründet ist dies dadurch, dass ein Spaltenname kein Operator mit Operanden ist.

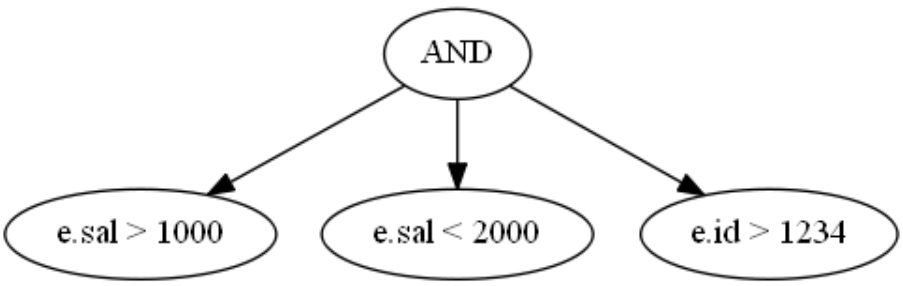


Abbildung 3.2: WHERE-Bedingung geparst mit ZQL

Ein Objekt der Klasse **ZGroupBy** speichert im Wesentlichen zwei Informationen. Die `GROUP BY` Ausdrücke werden gespeichert in einem *Vector* von Klassen, die das `ZExp` Interface implementiert haben. Der optionale `HAVING BY` Ausdruck wird als Objekt einer der Klassen gespeichert, welche `ZExp` implementieren. Typischerweise wird das ein Objekt der Klasse `ZExpression` sein.

Die Klasse **ZOrderBy** speichert einzelne Sortierkriterien. Gebündelt werden diese dann über einen *Vector*. Ein Objekt der Klasse `ZOrderBy` enthält einen Ausdruck vom Typ `ZExp` sowie die Information ob nach dem Suchkriterium aufsteigend oder absteigend sortiert werden soll.

Letztendlich vereinigt die Klasse **ZQuery** die eben vorgestellten Klassen. Die Funktion `getSelect()` liefert einen *Vector* von `ZSelectItem` zurück. Ähnlich dazu liefert die Methode `getFrom()` einen *Vector* von `ZFromItem` zurück. Hier erkennt man schon eine Beschränkung des Parsers. Da der `FROM`-Teil nur als Ansammlung von `FROM`-Items abstrahiert wird, werden Joins unter `FROM` nicht vom Parser erfasst. Die Methode `getWhere()` liefert ein Objekt zurück, was das Interface `ZExp` implementiert haben muss. Typischerweise ist dies ein Objekt der Klasse `ZExpression`. Analog dazu liefert die Methode `getGroupBy()` ein Objekt der Klasse `ZGroupBy` zurück. Die Methode `getOrderBy` liefert einen *Vector* von `ZOrderBy`-Objekten zurück. Schlussendlich existiert noch eine Methode `isDistinct()`, die klärt ob ein `DISTINCT` verwendet wurde.

Dies sind die am häufigsten gebrauchten Klassen des ZQL-Parser. Wir wollen nun die Arbeitsweise des Parsers anhand eines Beispieles verdeutlichen. Da die `SELECT`-Anfragen die wohl am häufigsten gebrauchte Form der Anfragen ist, wird sich die Erklärung der Funktionsweise des Parsers beispielhaft auf diese Art der Anfragen beziehen. Wie die anderen Statements geparst werden ist dann analog schnell zu verstehen.

Eine gewöhnliches Select-Statement wird wie folgt vom Parser zerlegt:

```
SELECT e.name, sal, dname
FROM emp e, dept d
WHERE e.sal > 1000 AND e.did = d.id
ORDER BY e.sal DESC
```

Die ganze Anfrage wird in einem Objekt vom Typ `ZQuery` eingebettet. Wir gehen nun nacheinander die Bestandteile dieses Objektes durch. Wir schreiben vereinfacht eine Art Pseudocode. Elemente in einem Array oder Vector werden einfach als Menge `{elem1, elem2, elem3}` dargestellt. Objekte werden vereinfacht dargestellt als Ansammlung von Attributen. Dies ist legitim, da die Methoden fast ausschließlich nur getter und setter sind. Ein Objekt o mit dem Attribut "name" und den Wert "Otto" schreiben wir daher als: `o = { [name=Otto] }`

Der **SELECT**-Teil wird durch einen *Vector* dargestellt. Alle Items sind vom Typ *ZSelectItem*. `SelectVector = { selItem1, selItem2, selItem3 }`.

`selItem1 = { [Table=e], [Column=name], [Expression=false], [Wildcard=false] }`

`selItem2 = { [Table=null], [Column=sal], [Expression=false], [Wildcard=false] }`

`selItem3 = { [Table=null], [Column=dname], [Expression=false], [Wildcard=false] }`

Auch im `FROM`-Teil finden wir einen *Vector*, der nun aus Objekten vom Typ `ZFromItem` besteht. `FromVector = { fromItem1, fromItem2 }`.

`fromItem1 = { [Table=emp], [Alias=e] }` `fromItem2 = { [Table=dept], [Alias=d] }`

Kommen wir nun zum komplexeren Teil, dem `WHERE`-Ausdruck. Da diese Struktur baumartig ist, lässt sich dies zunächst besser in einem Bild ausdrücken.

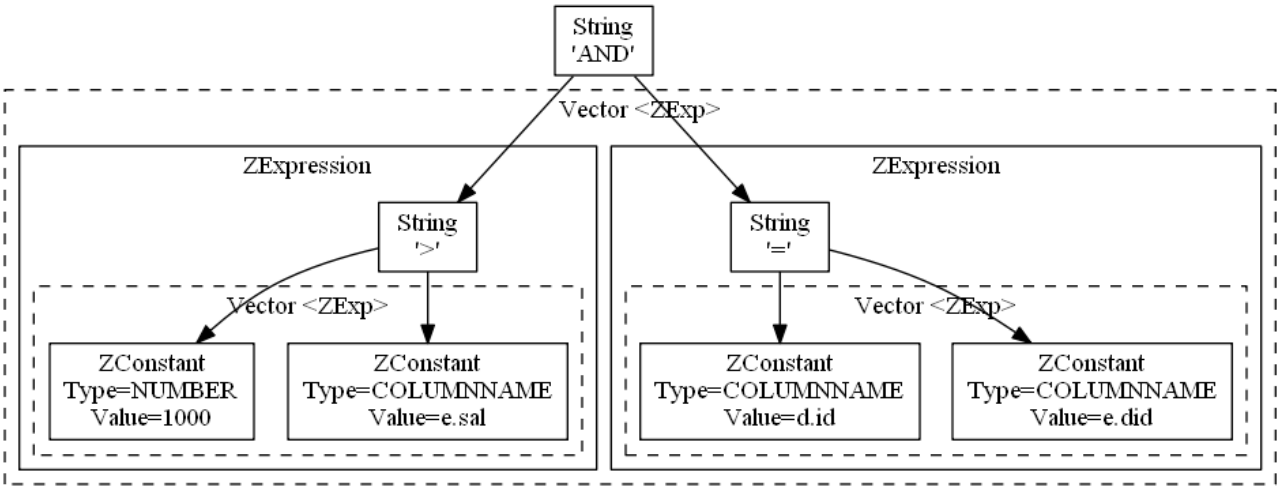


Abbildung 3.3: Geparster Baum der WHERE Bedingung des Beispiels

In Abbildung 3.3 sieht man nun den geparsten Baum der `WHERE`-Bedingung. Die gestrichelten Kästen sollen andeuten, dass es sich um den *Vector* von Klassen handelt, die `ZExp` implementieren, während durchgezogene Kasten konkrete Objekte sein sollen.

3.1.3 Grenzen des Parsers

Der Parser kann keine `CREATE TABLE`-Statements parsen. Somit ist es im Rahmen dieser Arbeit notwendig, den Parser zu erweitern, damit Tabellen in eigene Datenstrukturen geparst werden können. Für die Arbeit ist es zunächst nur notwendig Name, Datentyp, Fremdschlüsselbeziehungen und NULL-Informationen der Spalten in eine interne Datenstruktur zu überführen. Dabei wird beim Typ nur zwischen Zahlen und Sonstigem (Text) unterschieden. Wir speichern außerdem, wie viele Nachkommastellen ein Attribut haben kann. Unser Programm soll in der Lage sein, einfache arithmetische Operationen durchzuführen. Dazu ist das wissen um Datentypen der Variablen von Nöten.

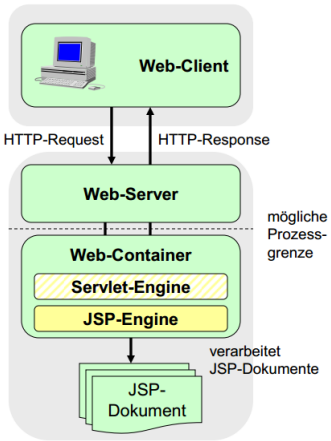
Der ZQL-Parser versteht das `FROM`-Statement als Liste von Relationen mit optionalen Tupelvariablen. Dadurch leiten sich mehrere Einschränkungen ab. ZQL ist dadurch nicht in der Lage `JOINS`

über die Schlüsselworte
`ON [LEFT OUTER|RIGHT OUTER|INNER] JOIN` zu erkennen. Der Parser erkennt nur innere `JOINS`, die im `WHERE`-Teil formuliert worden. Der ZQL-Parser unterstützt daher auch keine Unterabfragen unter `FROM`. Eine Möglichkeit diesem Problem zu begegnen ist es, solche Unterabfragen vor dem Parsen zu Entfernen und danach unverändert wieder einzufügen. Es wäre allerdings wünschenswerter wenn der ZQL-Parser erweitert würde, so dass auch ein `ZQuery` als `ZFromItem` benutzt werden kann.

Trotz dieser Einschränkungen sind alle Konzepte, die in dieser Arbeit vorgestellt werden einfach auf jedweden SQL-Parser übertragbar.

3.2 JavaServer Pages

JavaServer Pages (JSP), ist eine auf JHTML basierende Web-Programmiersprache zur dynamischen Erzeugung von HTML- oder XML-Dokumenten auf einem Webserver. Im Jahre 1999 wurden die JSP von Sun Microsystems veröffentlicht. Man kann sich die JSP als High-Level Abstraktion eines Java Servlets vorstellen. JSP werden zur Laufzeit in Servlets umgewandelt. Jedes dieser JSP-Servlets wird gecached und wiederverwendet, bis die Original-JSP verändert wurde. Ein skizzenhaftes Schema, finden wir in Abbildung 3.4a



(a) JSP-Konzep

Objekt	Typ
request	javax.servlet.http.HttpServletRequest
response	javax.servlet.http.HttpServletResponse
session	javax.servlet.http.HttpSession
pageContext	javax.servlet.jsp.PageContext
application	javax.servlet.ServletContext
config	javax.servlet.ServletConfig
out	javax.servlet.jsp.JspWriter
exception	java.lang.Throwable

(b) Implizte Objekte in JSP

Abbildung 3.4

Man könnte die JSP als Erweiterung der Servlets ansehen, da die JSP alle Funktionen dieser übernehmen. Zusätzlich können auch eigene Tag-Libraries erstellt werden. JSP funktioniert ähnlich wie ASP oder PHP. Sowohl Java-Servlets, als auch die JSP, haben die Aufgabe Javacode per HTML dem Nutzer verfügbar zu machen. Im Gegensatz zu den Java-Servlets, die komplett aus Javacode bestehen, bestehen JSP-Dateien aus HTML mit kurzen Einschüben von Javacode. Wäh-

rend man also bei den Servlets HTML-Ausgaben nur innerhalb einer Funktion bewerkstelligen kann, ist es möglich bei den JSP direkt HTML auszuschreiben.

3.2.1 Funktionsweise

Wie bereits erwähnt, handelt es sich bei einer JSP-Seite um ein HTML-Dokument. Daher gibt es hier keine expliziten Funktionen oder Klassen, die die Ausgabe eines Javaprogrammes steuern. Stattdessen stehen dem Programmierer unter JSP mehrere implizite Objekte zur Verfügung, um deren Konstruktion er sich nicht kümmern muss. Eine Übersicht der impliziten Objekte finden wir in Abbildung 3.4b. Da eine JSP-Datei im eigentlichem Sinne eine (J)HTML-Seite ist, muss der Javacode abgegrenzt werden. Wir schreiben sämtlichen Javacode daher innerhalb von `<% %>`. Diese Funktionieren ähnlich wie `<?php ?>` in PHP.

Wir gehen nun kurz auf wesentliche implizite Objekte ein. Das Objekt **request** beinhaltet alle Funktionen und Daten, die wir benötigen um die HTTP-Anfrage zu verarbeiten. Wir haben z. B. Zugriff auf: Cookies, HTTP-Header, übertragene POST- oder GET-Variablen (hier Parameter genannt).

Das **response**-Objekt ist das genaue Gegenstück zum request-Objekt. Es enthält alle Informationen für die HTTP-Response und wird vor allem genutzt, um den Content-Type festzulegen und Response-Header oder Cookies zu setzen.

Das **session**-Objekt regelt alle wichtigen Funktionen für die aktuelle Session. Wir können hier die Session-ID abfragen, sowie Objekte zur Session hinzufügen und entfernen.

Als letztes, möchten wir das **out**-Objekt erwähnen. Es repräsentiert einen BufferedWriter, genauer JSPWriter. Das Objekt kennt im Wesentlichen die Funktionen `print` und `println`. In PHP entspricht dies einem `echo` oder `print` Befehl. Der so gedruckte Text, wird zur Laufzeit in die entstehende HTML-Datei geschrieben.

Das Folgende Beispiel soll die Verwendung der impliziten Objekte verdeutlichen, ist aber, aus Platzgründen, stark vereinfacht.

```
<%@page import="java.util.Date"%>
<%@page language="java" contentType="text/html"
                pageEncoding="UTF-8" %>
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>Datum: <% out.print(new Date()); %></title></head>
<body>Zu dieser Seite sind
<% out.print(request.getCookies().length); %>
Cookies gespeichert.
</body></html>
```

3.3 Auflistung der Software

Um die Programmierung nachzuvollziehen wird im Folgenden die verwendete Software aufgelistet. Wir vermerken dabei Name, Version, Kurzbeschreibung sowie die Webpräsenz der Software.

Software: ZQL-Parser **Version**: 2011-08-26
Beschreibung: Ein Open-Source-Parser, der mit JavaCC, einem Java-Parser-Generator, realisiert wurde. Wir haben bereits einige Schwächen bemerkt, er ist aber leicht erweiterbar, da er quelloffen ist und unter der GNU GPLv3 steht.
URL: `http://zql.sourceforge.net/`

Software: Apache Tomcat **Version**:6.0
Beschreibung: Tomcat ist ein Webserver, der in einem Web-Container die Servlet- und die JSP-Engine bereithält. Mit ihm ist es möglich, JSP-Seiten zu betreiben.
URL: `http://tomcat.apache.org/`

Software: Oracle JDK **Version**:1.6.0
Beschreibung: Das Java-JDK ist notwendig um Programme zu übersetzen und auszuführen. Wir verwenden Version 1.6.0, da im universitärem Umfeld, in dessen Rahmen unser Programm entsteht, häufig noch diese Version verwendet wird. Unser Programm ist jedoch auch kompatibel zu JDK 1.7.0.
URL: `http://www.oracle.com/technetwork/java/javase/downloads/index.html`

Software: JDBC-Connector für MySQL **Version**:5.1.25
Beschreibung: Java Database Connectivity (JDBC) ist eine Datenbankschnittstelle der Java-Plattform, die eine einheitliche Schnittstelle zu Datenbanken verschiedener Hersteller bietet und speziell auf relationale Datenbanken ausgerichtet ist. JDBC ist in seiner Funktion als universelle Datenbankschnittstelle vergleichbar mit z. B. ODBC unter Windows oder DBI unter Perl. Zu den Aufgaben von JDBC gehört es, Datenbankverbindungen aufzubauen und zu verwalten, SQL-Anfragen an die Datenbank weiterzuleiten und die Ergebnisse in eine für Java nutzbare Form umzuwandeln und dem Programm zur Verfügung zu stellen. Für jede spezifische Datenbank sind eigene Treiber erforderlich, die die JDBC-Spezifikation implementieren. Diese Treiber werden meist vom Hersteller des Datenbank-Systems geliefert.
URL: `http://dev.mysql.com/downloads/connector/j/`

Software: JDBC-Connector für PostgreSQL **Version**:9.2-1003
Beschreibung: siehe JDBC-Connector für MySQL
URL: `http://jdbc.postgresql.org/download.html`

Software: JDBC-Connector für Oracle Database **Version**:-
Beschreibung: siehe JDBC-Connector für MySQL
URL: `http://www.oracle.com/technetwork/database/features/jdbc/index-091264.html`

4 Theoretische Betrachtungen

Um die Frage zu beantworten wie man zwei SQL-Anfragen miteinander vergleichen kann, muss man sich die Struktur einer solchen Anfrage betrachten. Exemplarisch betrachten wir im folgenden SELECT Anfragen. Es werden mehrere Ansätze in diesem Teil der Arbeit verfolgt, wie man die Gleichheit von zwei Anfragen zeigen kann. Offensichtlich sind zwei SQL-Anfragen semantisch äquivalent, wenn sie ebenfalls syntaktisch äquivalent sind. Interessanter sind daher Anfragen, die zunächst nicht syntaktisch deckungsgleich sind.

Ein Ansatz besteht darin beide SQL-Anfragen einer Standardisierung zu unterziehen. Wie genau so etwas durchgeführt werden kann, wird im Folgenden noch erläutert. Dann würden wir zwei standardisierte SQL-Anfragen erhalten. Sind diese syntaktisch äquivalent, so handelt es sich um identische Anfragen.

Mit der Standardisierung der Anfragen können wir jedoch nur ein hinreichendes Kriterium abdecken. Schaffen wir es nicht, mit Hilfe der Standardisierung, zwei Anfragen syntaktisch, und damit auch semantisch, gleich zu machen, so möchten wir sicher gehen, dass die Anfragen tatsächlich nicht semantisch äquivalent sind. Dazu testen wir die Anfrage gegen ein notwendiges Kriterium. Wir führen sowohl die Anfrage der Musterlösung, als auch die Anfrage des Lernenden auf einer echten Datenbank mit konkreten Daten aus. Liefern beide Anfrage unterschiedliche Ergebnistupel, so wissen wir sicher, dass unsere Anfragen nicht semantisch äquivalent sein können.

Einen ungeklärten Fall haben wir, wenn die beiden Anfragen durch Standardisierung nicht angepasst werden konnten und die Ergebnistupel beider Anfragen auf der konkreten Datenbank identisch sind. In diesem, dritten Fall, gelingt es uns keine Aussage über die semantische Äquivalenz zu treffen. Daher ist es wünschenswert, das Auftreten dieses Falles, so gut es geht, zu minimieren.

Zu Bemerken ist hierbei, dass im folgendem Kapitel die theoretischen Grundlagen behandelt werden, die notwendig sind, um zwei SQL-Anfragen auf semantische Äquivalenz zu prüfen. Dabei werden hier insbesondere auch Ideen vorgestellt, die nicht im praktischem Teil, dem Programm, auftauchen.

4.1 Hintergrund

Es gibt syntaktisch unterschiedliche Anfragen, die jedoch semantisch äquivalent sind. So liefern die folgenden Anfragen die gleichen Ergebnisse obwohl sie nicht syntaktisch äquivalent sind. Wir nehmen für das Beispiel an, dass `e.enr` vom ganzzahligem Typ Integer ist.

```
SELECT * FROM emp e WHERE e.enr > 5
```

```
SELECT * FROM emp e WHERE 5 < e.enr
```

```
SELECT * FROM emp e WHERE e.enr >= 6
```

Wie man leicht sieht, sind die Anfragen syntaktisch ähnlich und semantisch äquivalent.

Neben solchen syntaktischen Varianten, kann es auch sein, dass unnötige Bedingungen aufgeschrieben werden, die das Ergebnis nur unnötig kompliziert machen. Eine Möglichkeit ist folgende Anfrage, in der offensichtlich die letzte Bedingung überflüssig ist.

```
SELECT * FROM emp e WHERE e.enr > 5 AND e.enr <> 2
```

Unser Programm müsste nun erkennen, dass das Attribut `enr` bereits beschränkt ist, und der Wert 2 nicht mehr auftreten kann. Das Programm, was zu dieser Arbeit entwickelt wird, kann mit solchen redundanten Eigenschaften nicht umgehen. Es wäre auch eher ein Problem für einen "semantic checker", da es hier nicht auf zwei verschiedene Anfragen ankommt. Hier ist bereits diese eine Anfrage in sich selbst zu kompliziert. Mit solchen Problemen beschäftigt sich bereits andere Lernplattformen, die wir bereits im Kapitel 2 betrachtet haben.

Weitere Probleme sind Operatoren, die sich auf andere abbilden lassen. Man kann nie wissen, in welcher Art und Weise der Lernende die Aufgabe formulieren wird. Man betrachte dazu folgende zwei Anfragen:

```
SELECT * FROM emp e WHERE e.sal BETWEEN 10 AND 200
```

```
SELECT * FROM emp e WHERE e.sal >= 10 AND e.sal <=200
```

Offensichtlich sind die Anfragen äquivalent. Dies erreichen wir im Wesentlichen, indem wir bestimmte Operatoren wie `BETWEEN` abschaffen und durch äquivalente Ungleichungen mit >= und <= ersetzen. Ähnliches gibt es für `INNER JOIN` im `FROM`-Teil, mit Ersetzung durch Vergleiche im `WHERE`-Teil, was wir später im Detail betrachten werden.

4.2 Workflow

Im folgendem Abschnitt erläutern wir einzelne Schritte unseres Vorgehens. Dabei ist zu beachten, dass im entstehenden Programm einige Schritte ausgelassen werden oder in einer anderen Reihenfolge vorkommen. Auslassung werden entweder in diesem Kapitel oder im Kapitel 5 erläutert. Die tatsächliche Abfolge der Schritte im Programm, finden sich im ebenso im Kapitel 5.

4.2.1 Standardisierung

Im ersten Ansatz unterziehen wir jeder Anfrage einem Standardisierungsprozess. Ziel dabei ist es, die zwei Anfragen durch legale Umformungen so anzupassen, dass sie syntaktisch äquivalent werden. Dazu beginnen wir mit dem lexikographischem Sortieren alle vorkommenden Tabellen im FROM-Teil. Anschließend erhalten diese eine automatisch generierte Tupelvariable. Diese werden in allen anderen Teilen der SQL-Anfrage ersetzt durch bereits vorhandene, oder eingeführt, falls vorher keine Tupelvariablen benutzt worden sind. Die meiste Arbeit wird dann im WHERE-Teil erledigt.

Dort ersetzen wir syntaktische Varianten durch einheitliche Schreibweisen und entfernen syntaktische Details. Nachdem der Ausdruck des WHERE-Teils auf eine standardisierte Form (konjunktive Normalform) gebracht wurde sortieren wir den Ausdruck so, dass eine gewisse Ordnung bezüglich der Operatoren vorliegt. Weitere Anpassungen sind das Ersetzen von jedweden Unterabfragen durch äquivalente EXISTS-Unterabfragen. Im nächsten Schritt werden Verbunde (engl. Joins) kritisch untersucht und ggf. ersetzt oder vereinfacht.

Die Operatorenvielfalt ist ein Problem, was auf zwei unterschiedliche Arten angegangen werden kann. Wir diskutieren sowohl das Hinzufügen von implizierten Formeln, als auch das Ersetzen von Formeln durch eine repräsentante Formel des gleichen Typs. Zum Abschluss werden noch GROUP BY- und ORDER BY-Ausdrücke auf ähnliche Weise angepasst.

Sind alle diese Teilschritte ausgeführt, vergleichen wir die standardisierten Anfragen auf syntaktische Äquivalenz. Bei Erfolg ist klar, dass diese ebenso semantisch äquivalent sind.

4.2.2 Test mit konkreten Daten

Konnten wir, durch die Standardisierung beider Anfragen, keine semantische Äquivalenz feststellen, so muss dies nicht notwendig bedeuten, dass die zwei Anfragen nicht semantisch äquivalent sind. Wir prüfen diese Tatsache, indem wir die zwei Anfragen auf einer Datenbank mit einem konkreten Datenbankzustand ausführen. Wir vergleichen anschließend die Ergebnistupel beider Anfragen. Sind diese nicht identisch, so ist klar, dass wir keine äquivalenten Anfragen haben. Sind

beide jedoch identisch, wissen wir nicht mit Sicherheit, ob die Anfragen äquivalent sind, oder nicht. Wir erinnern uns, dass es eine Teilmenge von SQL-Anfragen gibt, bei denen eine Äquivalenz nicht entscheidbar ist. In einem solchen Fall ist es sinnvoll, dem Dozenten die Anfrage des Lernenden zu zeigen, damit dieser entscheiden kann, ob die Anfrage korrekt ist.

4.2.3 Preprocessing

Im Abschitt »Forschungsstand« haben wir bereits einige Lernplattformen/ -projekte zum Thema SQL kennen gelernt. Viele dieser Plattformen möchten dem Lernenden genügend Feedback beim Lernprozess geben. Dies ist nicht nur sinnvoll, damit der Student schneller auf korrekte Lösungen stößt, sonder auch, weil die Standardhinweise eines SQL-Systems meist nur auf syntaktische Fehler hinweisen.

Da in dieser Arbeit zwei SQL-Anfragen verglichen werden sollen, können wir nicht viele Ideen der »semantic checker« übernehmen. Dennoch möchten wir dem Lernenden auch ein Feedback über mögliche semantische Fehler geben. Egal ob das Matchen der Musterlösung und der Lösung des Lernenden gelingt oder nicht ein solches Feedback über semantische Fehler ist in jedem Fall hilfreich.

Dies geschieht in folgender Art und Weise. Direkt nach dem Parsen und noch bevor wir irgendwelche Anpassungen vornehmen, analysieren wir beide Anfragen und sammeln Metainformationen über diese. Nachdem wir beide Anfragen standardisiert haben, vergleichen wir die Metainformationen beider Anfragen systematisch miteinander. Fallen uns Unterschiede auf, so teilen wir dies dem Lernenden als Hinweis mit. Neben dem Bloßen Vergleich von Metainformationen sind noch andere Kriterien für den Vergleich interessant. So vergleichen wir weiterhin, ob beide Anfragen die gleichen Tabellen unter `FROM` ansprechen. Auch interessiert uns in diesem Zusammenhang ob Teile der SQL-Anfragen miteinander übereinstimmen. Wir prüfen also, ob `SELECT`, `WHERE`, `FROM`, `GROUP BY` und `ORDER BY` der beiden Anfragen jeweils zueinander identisch sind. So können wir dem Lernenden deutlich machen, dass z. B. der `WHERE`-Teil seiner Lösung korrekt ist, aber der `SELECT`-Teil noch nicht mit dem der Musterlösung übereinstimmt.

atomarer Formeln unter WHERE / Baumtiefe

Enthält die Anfrage des Lernenden, vor der Transformation durch unser Programm, mehr atomare Formeln als die Musterlösung, so wurden offensichtlich unnötige Formeln oder doppelte Formeln aufgeschrieben. Stellt unser Programm fest, dass beide Lösungen dennoch gleich sind, so muss dem Lernenden mitgeteilt werden, dass er redundante Formeln eingebaut hat, welche die Lösung unnötig verkomplizieren.

Gleichzeitig kann die unterschiedliche Tiefe des Parserbaumes der `WHERE`-Klausel auch auf redundante Formeln hinweisen, die unser Programm durch Konstanten ersetzt hat. Beispiele dafür sind arithmetisch/logische Ausdrücke, die nur Konstanten als Operanden haben und somit auswertbar und verkürzbar sind.

Existenz von Teilen der SQL-Anfrage

Geraden wenn der Lernende eine falsche Lösung einsendet, kann es sein, dass er sogar wichtige Teile einer Anfrage nicht benutzt, die aber von der Musterlösung verwendet werden. So ist es sinnvoll zu vergleichen, ob sowohl die Musterlösung, als auch die Lösung des Studenten einen nicht leeren `SELECT-`, `FROM-`, `WHERE-`, `GROUP BY-` und `ORDER BY`-Teil hat. Unterschiede müssen dem Lernenden mitgeteilt werden. Dabei ist sowohl die Information interessant, ob ein gewisser Teil existiert, als auch die Anzahl der Attribute oder Formeln im jeweiligen Teil der Anfrage.

Anzahl von JOINs / Unterabfragen

Sehr viele Fehler bei Anfängern finden sich im Bereich JOINs. Gerade deswegen ist es für den Lernenden interessant, ob er mehr oder weniger JOINs verwendet hat, als die Musterlösung. Im selben Atemzug sind Unterabfragen genannt, die entweder zu zaghaft oder zu häufig von Lernenden eingesetzt werden. Daher sollte auch diese Anzahl überwacht werden.

Anzahl der Operationen

Wie im vorherigen Abschnitt bereits erklärt ist der ZQL-Parserbaum nicht binär. Dadurch kann es durch zu vorsichtige Klammersetzung passieren, dass ein Teilbaum mit zwei Ebenen entsteht, obwohl nur ein Operator beteiligt ist. Erklärt ist dies im Abschnitt »Funktionsweise des Parsers«. Die dort vorgestellte Operatorkompression ist ein Verfahren um unnötige Klammerungen zu entfernen. Ist die Gleichheit der Lösung des Studenten mit der Musterlösung durch unser Programm gezeigt, aber die Studentenlösung musste mehr Operatorkompressionen durchführen, so hat der Student unnötige Klammern gesetzt, welche die Lösung wiederum unnötig verkomplizieren. Dies muss ihm durch unser Programm mitgeteilt werden.

Es folgt eine Zusammenfassung der Informationen, die wir im Zuge des Sammelns der Metainformationen speichern wollen.

- Anzahl der JOIN Bedingungen
  - Anzahl von OUTER/ INNER Joins
- Anzahl atomarer Formeln im `WHERE`-Teil

- Anzahl atomarer Formeln im `HAVING`-Teil
- Anzahl Tabellen in `FROM`-Teil
- Anzahl Attribute im `SELECT`-Teil *
- Existiert ein `DISTINCT`?
- Existiert ein `GROUP BY`? *
  - Wenn ja, tauchen Variablennamen und Aggregationsfunktionen auch im SELECT-Teil auf?
- Existiert ein `HAVING BY`?
- Tiefe des Parserbaums

Unterscheiden sich Musterlösung und Lösung des Studenten in den mit * markierten Punkten ist es extrem unwahrscheinlich, dass beide Lösungen die gleichen Tupel zurückliefern würden. In einem solchen Fall möchten wir dem Lernenden eine Warnung anzeigen, die beinhaltet, dass er höchstwahrscheinlich etwas vergessen hat. Im Allgemeinen vergleichen wir jede Metainformation der Musterlösung mit der jeweiligen Information der Lösung des Lernenden. So sind folgende Meldungen denkbar:

- "Die Musterlösung enthält zwei Joins, aber ihre Lösung enthält keinen Join."
- "Ihre Lösung ist korrekt, allerdings enthält die Musterlösung zwei atomare Formeln weniger (Formel1, Formel2)."
- "Ihre Lösung ist korrekt, aber die Musterlösung enthält kein DISTINCT. Überlegen Sie, ob dies wirklich notwendig ist.."

Folgende zusätzliche Vergleiche führen wir nach der Standardisierung beider Anfragen durch. Dabei bezeichnen wir die Musterlösung mit ML und die Lösung des Lernenden mit LL.

- Stimmen `SELECT`-Teil von ML und LL überein?
- Stimmen `FROM`-Teil von ML und LL überein?
- Stimmen `WHERE`-Teil von ML und LL überein?
- Stimmen `GROUP BY`-Teil von ML und LL überein?
- Stimmen `HAVING`-Teil von ML und LL überein?
- Stimmen `ORDER BY`-Teil von ML und LL überein?

Möglich wären auch deutlichere Hinweise, z. B. die Angaben welche Tabellen unter `FROM` in der ML vorkommen und in der LL nicht. Allerdings könnten solche Meldungen zu viel von der ML preis geben.

Zusammenfassend kann man Folgendes sagen: Das Preprocessing wird direkt nach dem Parsen einer SQL-Anfrage durchgeführt. Es sammelt Metainformationen über die Anfrage. Da wir zwei Anfragen vergleichen, werden diese Metainformationen einzeln für jede Anfrage gespeichert. Dann beginnen wir mit den weiteren Schritten, die im Folgenden detailliert erläutert werden.

Egal ob die Ergebnisse in diesen weiteren Schritt erfolgreich waren oder nicht, wir geben danach einen Vergleich der Metainformationen aus. Zusätzlich führen wir weitere Vergleiche mit den standardisierten Anfragen durch, welche eben bereits dokumentiert worden. Das soll dem Lernenden bei falschen Lösungen Anhaltspunkte geben, wie eine richtige Lösung aussehen könnte. Bei einer korrekten Lösung, können solche Hinweise trotzdem nützlich sein, denn die Anfrage des Lernenden kann trotz Korrektheit zu lang bzw. zu kompliziert sein. Dies würde bei einem Vergleich der gesammelten Metainformationen deutlich werden.

4.3 Standardisierung von SQL-Anfragen

Zunächst verfolgen wir den Ansatz zwei SQL-Anfragen zu vergleichen, indem wir sie standardisieren. Wir erhoffen uns durch die Standardisierung eine syntaktische Äquivalenz beider Anfragen zu erreichen. Dies würde automatisch eine semantische Äquivalenz bedeuten. Die Kriterien der Standardisierung werden im Detail behandelt. Standardisiert man die Musterlösung, als auch die Lösung des Lernenden nach den gleichen Kriterien, so kann man danach durch einen einfachen Stringvergleich auf die syntaktische, und damit auch auf die semantische, Äquivalenz schließen.

4.3.1 Entfernen von syntaktischen Details

Das Entfernen von syntaktischen Details übernimmt zu einem Teil bereits der Parser. Er entfernt unnötige Leerzeichen, Kommentare sowie unnötige Klammern. Aufgrund der Arbeitsweise des Parsers gibt es allerdings Situationen, in denen der Parser scheinbar nicht alle unnötigen Klammern entfernt. Wie im Abschnitt »Verwendeter Parser« erläutert wurde, sind die geparsten Bäume nicht binär. Unnötige Klammerungen müssen daher eventuell nachträglich entfernt werden. Dazu stellen wir noch eine Methode vor.

Der Parser hilft allerdings dabei die SQL-Anfrage in einer Datenstruktur zu überführen, die frei von allen syntaktischen Details ist. Dazu gehören Leerzeichen, Tabs, Zeilenumbrüche und Groß-/Kleinschreibung von Schlüsselwörtern.

4.3.2 SELECT-Teil

Im SELECT-Teil brauchen wir im Normalfall keine Änderungen vornehmen. Das einzige, was hier anpassbar wäre, ist die Reihenfolge der einzelnen Attribute bzw. Aggregationsfunktionen. Diese ist aber im Standardfall vorgegeben. Ist für eine Aufgabenstellung aber explizit vermerkt, dass die Reihenfolge der Spalten keine Rolle spielt, so sortieren wir die Spaltennamen lexikographisch. Dies muss der Dozent beim Einstellen der Aufgabe explizit vermerken. Eine weitere Besonderheit stellen die Spaltenüberschriften dar. Hier muss wieder vermerkt werden, ob Spaltenüberschriften von Bedeutung sind. Ist dies nicht der Fall, so entfernen wir aus dem SELECT-Teil alle Spaltenüberschriften.

4.3.3 FROM-Teil

Unter einer bestimmten Voraussetzung, können wir die Tabellen unter FROM lexikographisch sortieren. Besteht der SELECT-Teil nicht nur aus dem Wildcard *, dann ist die Reihenfolge der Spalten im Ergebnis nur von der Reihenfolge der Attribute und Aggregationsfunktionen im SELECT-Teil abhängig. Haben wir aber einen SELECT-Teil, der nur aus * besteht, dann hängt die Reihenfolge der Tabellen von ihrer Reihenfolge unter FROM ab. Betrachten wir dazu folgende Beispielanfragen:

```
1: SELECT * FROM emp e, dept d
2: SELECT * FROM dept d, emp e
```

In Anfrage 1 erhalten wir erst alle Spalten der Tabelle emp und danach alle Spalten der Tabelle dept. In Anfrage 2 ist das Verhalten genau umgekehrt. Daher ist klar, dass wir die Tabellen, die im FROM-Teil auftauchen genau dann sortieren können, wenn der SELECT-Teil nicht nur aus * besteht.

Danach werden automatische Tupelvariablen erzeugt. Sind bereits Tupelvariablen vom Nutzer vergeben wurden, so werden diese ebenfalls durch die automatischen ersetzt. Eine Hashtabelle speichert frühere Zuweisungen, damit im SELECT-, WHERE-, GROUP BY- und ORDER BY-Teil die Tupelvariablen ebenfalls korrekt ersetzt werden.

Hatten die vorkommenden Tabellen im FROM-Teil keine Tupelvariablen, so werden künstliche eingeführt.

Eingabe:
```
SELECT e.id, e.name, d.region FROM emp e, dep d WHERE e.depid = d.id
```

Anpassung:
```
SELECT a2.id, a2.name, a1.region FROM dep a1, emp a2 WHERE a2.depid = a1.id
```

Abbildung 4.1: Beispiel: Umwandlung des FROM-Teils einer SQL-Anfrage

4.3.4 WHERE-Teil

Es ist wünschenswert eine Normalform des `WHERE`-Teils zu erreichen, da dies die Übersichtlichkeit steigert und eine wichtige Voraussetzung zum Bearbeiten ist. Wir möchten den `WHERE`-Teil in eine konjunktive Normalform (KNF) überführen, da Anwender oft SQL-Anfragen formulieren, die konjunktiv einzelne Bedingungen verknüpfen.

Entfernung unnötiger Klammerungen

Ein Ausdruck `((a > 5) AND ((b > 5) AND (c > 5)))` enthält unnötige Klammern, da der Operator `AND` als Operand von einem weiteren `AND` vorkommt. Folgender Ausdruck ist äquivalent: `((a > 5) AND (b > 5) AND (c > 5))`. Diese spezielle Form der Klammerung entsteht aus der Tatsache, dass der ZQL-Parserbaum nicht binär ist und beide eben genannten Beispiele nicht den gleichen Baum beschreiben. Als ersten Schritt in Richtung KNF möchten wir solche unnötigen Klammern entfernen.

Es ist daher wünschenswert, wenn ein Operator $op1$ einen Operanden als Kindknoten besitzt, der wiederum $op1$ repräsentiert, den Kindknoten zu eliminieren. Alle Kinder vom eliminierten Kindknoten hängen wir nun an den verbleibenden Operatorknoten $op1$ an. Damit hätte man den Ausdruck vereinfacht, da die assoziative Klammerung wegfällt. Wir nennen dieses Vorgehen im Folgenden **Operatorkompression**. Verbildlicht wird dieser Vorgang in Abbildung 4.2.

BILD

Abbildung 4.2: Operatorkompression

Gegeben sei der ZQL-Parsebaum $B = (V, E)$. Es sei $children(v) = \{w : w \in V \wedge (v, w) \in E\}$, also die Menge aller Kindknoten von v. Gilt $v \in V$ und sei v ein innerer Knoten, dann muss v einen Operator repräsentieren, den wir mit $operator(v)$ erhalten. Gibt es einen Knoten $w \in child(v)$ mit $operator(v) = operator(w)$, so wird Knoten w eliminiert und alle Kindknoten von w werden zu Kindknoten von v, also $child(v) := child(v) \cup child(w)$ und $E = E \backslash \{(w, x) : x \in child(w)\} \cup \{(v, x) x \in child(w)\}$ und $V = V \backslash \{w\}$.

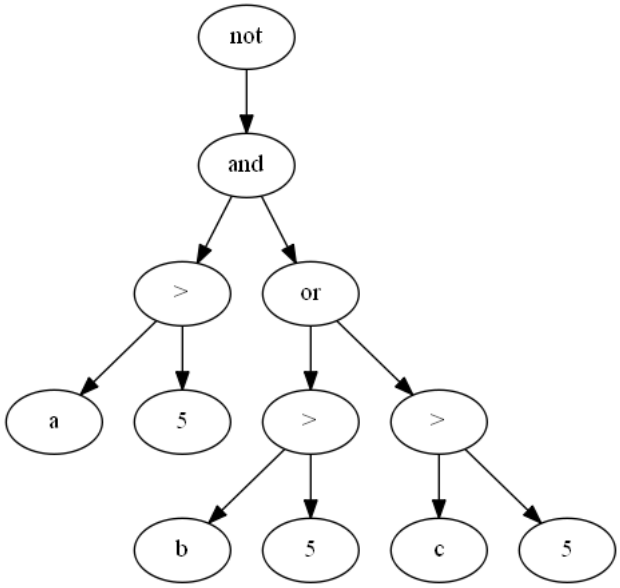
Im Sinne des Vergleiches der Komplexität der Musterlösung mit der Komplexität der Lösung des Lernenden ist es sinnvoll zu speichern, ob und wie oft eine solche Operatorkompression durchgeführt werden musste.

NOT auflösen

Im nächsten Schritt möchten wir auftretende `NOT`-Operatoren entfernen. Dies geschieht, indem der Operator `NOT` im Parserbaum nach unten geschoben wird. Dabei werden die *DE MORGAN*-Regeln angewendet.

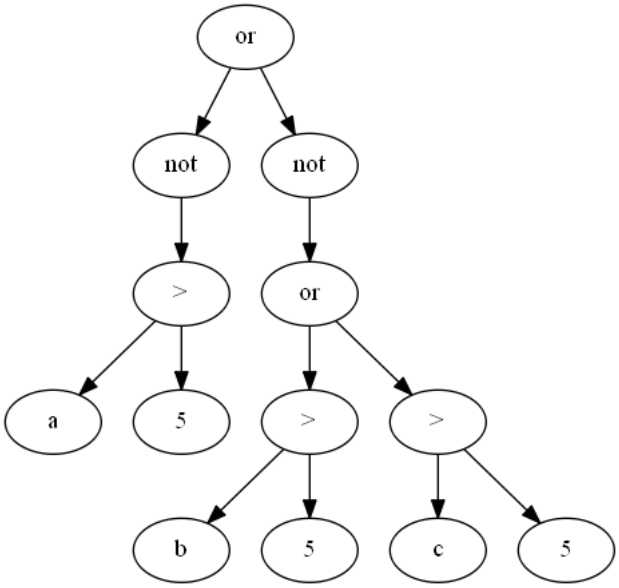
Eingabe:

```
not ((a > 5)  and ((b > 5) or (c > 5)))
```



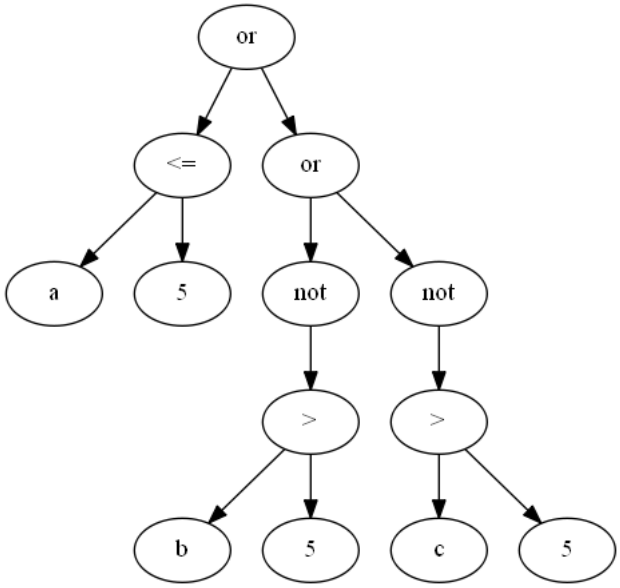
Umwandlung Teil 1:

```
(not(a > 5) or not((b > 5) or (c > 5)))
```



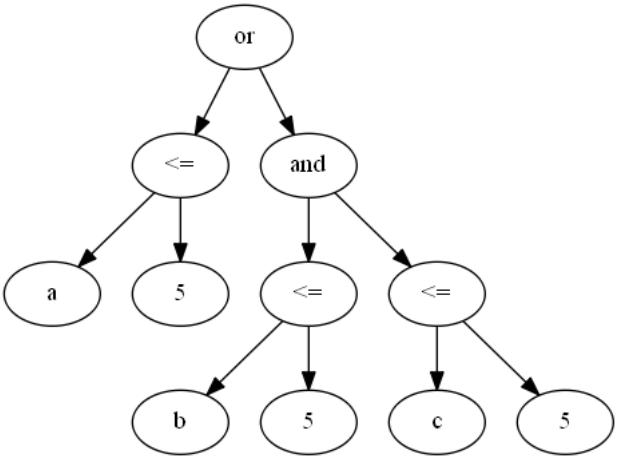
Umwandlung Teil 2:

```
((a <= 5) or (not(b > 5) and not(c > 5)))
```



Umwandlung Teil 3:

```
((a <= 5) or ((b <= 5) and (c <= 5)))
```



Anwenden des Distributivgesetzes

Im letzten Schritt haben wir die Formel `((a <= 5) or ((b <= 5) and (c <= 5)))` erhalten. Durch Anwenden des Distributivgesetzes können wir diese Formel im letzten Schritt umformen zu: `((a <= 5) or (b <= 5)) and ((a <= 5) or (c <= 5))`

Im Allgemeinen durchsuchen wir den Parserbaum auf Konstellationen, in denen ein Knoten mit dem Operator `OR`, Elternknoten von einem `AND`-Knoten ist. Dann müssen wir auf diesen Teilbaum das Distributivgesetz anwenden. Wir wiederholen diesen Vorgang, bis kein `OR`-Knoten mehr Vorfahr eines `AND`-Knotens ist.

4.3.5 Ersetzung von syntaktischen Varianten

Um eine Anfrage zu standardisieren müssen wir den syntaktischen Zucker entfernen. Dies geschieht, indem man nur eine syntaktische Schreibweise anerkennt und alle anderen Schreibweisen in die zulässige umgewandelt.

Mit dem Operator `BETWEEN` testen wir ob ein gegebener Ausdruck zwischen zwei Werten, lower und upper, befindet. Wir schreiben den Ausdruck `e BETWEEN lower AND upper` äquivalent um, in dem wir nur Vergleichsoperatoren benutzten.
Unser Ausdruck lautet dann: `e >= lower AND e <= upper`.

Der Operator `IN` prüft, ob ein gegebener Ausdruck einem bestimmten Wert aus einer Liste zugewiesen ist. Die Liste kann dabei als Unterabfrage oder als Liste von Konstanten beschrieben werden. Handelt es sich um eine Liste von Konstanten, so können wir
`e IN (const1, const2, ..., constn)` ersetzen, in dem wir prüfen ob `e` einen der konstanten Werte annimmt. Wir verknüpfen also n Vergleiche disjunktiv miteinander. Wir erhalten dadurch: `e = const1 OR e = const2 OR ... OR e = constn`.

Ähnlich dazu verhält sich der Operator `ALL`. Er tritt in folgender Form auf:
`operand comparison_operator ALL (subquery)`. Dabei kann `subquery` auch eine Liste von Konstanten sein. Es wird in jedem Fall geprüft ob der Operand und jedes der Listenelemente in der Vergleichsrelation enthalten ist. Haben wir als `subquery` eine Liste von Konstanten, so ist dies äquivalent mit dem paarweisen Vergleich von `operand` und jedem Listenelement. Diese n Vergleiche werden dann konjunktiv verknüpft, also:
`operand operator elem1 AND ... AND operand operator elemn`. Diese syntaktische Anpassung ist nicht im Programm enthalten, da der ZQL-Parser keine Listen unter `ALL` versteht.

Der Operator `ANY` ist beinahe äquivalent zu `ALL` mit dem Unterschied, dass die entstandene Liste der Vergleiche nicht konjunktiv, sondern disjunktiv verknüpft wird.

Befindet sich im WHERE-Teil eine EXISTS-Unterabfrage, so können wir den SELECT-Teil dieser Unterabfrage vereinfachen. Da wir nur wissen wollen ob es Ergebnistupel aus der Unterabfrage gibt, benötigen wir keine konkreten Spalten aus der Unterabfrage. Daher ersetzen wir den SELECT-Teil von EXISTS-Unterabfragen zu:SELECT 1. Ein Beispiel dieser Umwandlung ist in Abbildung 4.3 zu sehen.

```
SELECT e.sal FROM emp e WHERE EXISTS (
    SELECT expression FROM dept d
    WHERE e.deptno = d.deptno )

SELECT e.sal FROM emp e WHERE EXISTS (
    SELECT 1 FROM dept d
    WHERE e.deptno = d.deptno )
```

Abbildung 4.3: Umwandlung des SELECT-Teils einer EXISTS-Unterabfrage

Unteranfragen

Es ist bekannt, dass sämtliche Typen von Unteranfragen eliminiert oder durch EXISTS-Anfragen ersetzt werden können. Streng genommen handelt es sich hier zwar um mehr als nur eine syntaktische Variante, aber dennoch wollen wir das Ersetzen von Unteranfragen in diesem Abschnitt betrachten.

Wir behandeln in diesem Abschnitt nur echte Unterabfragen, das heißt die Unterabfrage darf keine Liste von Konstanten sein. Ist die Unterabfrage eine Liste, z. B. bei ALL, ANY oder IN gehen wir vor, wie im vorherigen Abschnitt beschrieben.

ALL-Unterabfragen

ALL-Unterabfragen haben immer das Muster: operand comparison_operator ALL (subquery). Dabei handelt es sich bei einem comparision_operator um einen Operator aus der Menge: {=,<>,>,<,>=,<=}. Die Unterabfrage subquery liefert eine Liste von Attributen. Dann wird so verfahren wie bereits im letzten Abschnitt erklärt. Unser Ziel ist es, diese Unterabfrage, wenn möglich, zu ersetzen durch eine EXISTS-Unterabfrage. Um diese Umwandlung nachzuvollziehen, wandeln wir zunächst die Unterabfrage mit ALL in eine Unterabfrage mit ANY um. Dazu ersetzen wir den comparison_operator mit seinem gegenteiligem Operator. Folgende Paare sind jeweils voneinander gegenteilige Operatoren: { {<,>=}, {>,<=}, {=,<>} }. Danach ersetzen wir ALL mit ANY und kapseln die gesamte Unterabfrage mit einem NOT.

```
WHERE t1 comparison_operator ALL (
        SELECT t2
        FROM R1 X1, ..., Rn Xn
```

```
        WHERE (φ)
)

Geschrieben als ANY-Unterabfrage:
WHERE NOT( t1 opposite(comparison_operator) ANY (
        SELECT t2
        FROM R1 X1, ..., Rn Xn
        WHERE (φ) )
)
```

Diese ANY-Unterabfrage können wir nun in eine EXISTS-Unterabfrage transformieren. Wie das genau durchgeführt wird, behandeln wir im nächsten Absatz.

ANY/ SOME-Unterabfragen

Da SOME und ANY äquivalente und synonyme Schlüsselworte sind, betrachten wir im Folgenden nur ANY-Unterabfragen. Eine solche Unterabfrage kann in eine EXISTS-Unterabfrage umgewandelt werden. Dazu verknüfen wir den Vergleichsoperator mitsamt linken Operanden konjunktiv mit dem WHERE-Teil der Unterabfrage. Als rechten Operanden wählen wir das Attribut, welches sich im SELECT-Teil der Unterabfrage befindet. Die Umwandlung findet also nach folgendem Muster statt:

```
WHERE t1 comparison_operator ANY (
        SELECT t2
        FROM R1 X1, ..., Rn Xn
        WHERE (φ)
)

Geschrieben als EXISTS-Unterabfrage:
WHERE EXISTS(
        SELECT t2
        FROM R1 X1, ..., Rn Xn
        WHERE (φ
        AND t1 comparison_operator t2 )
)
```

Wie wir am Ergebnis der Umwandlung erkennen, verstößt t_2 im SELECT-Teil der EXISTS-Unterabfrage bereits gegen unsere Konvention: “Im SELECT-Teil einer EXISTS-Unterabfrage steht nur die Konstante 1”. Es empfiehlt sich daher, die Normierung des SELECT-Teils von EXISTS-Unterabfragen erst nach dem Umwandeln von Unterabfragen durchzuführen.

IN-Unterabfragen

Das Schlüsselwort `IN` ist lediglich ein Synonym für `= ANY`. Diese Unterabfragen sind daher ein Spezialfall der, bereits betrachteten, `ANY`-Unterabfragen.

Voraussetzungen

Alle, eben genannten, Umwandlungen können nur durchgeführt werden, wenn bestimmte Bedingungen erfüllt sind.

1. Alle Tupelvariablen, die in t_1 vorkommen, müssen sich unterscheiden von allen X_i. Erreicht wird dies ggf. durch Umbenennung der X_i, da diese nicht für die eigentliche (Ober)anfrage wichtig sind.
2. Wenn t_1 Attributreferenzen A ohne Tupelvariable enthält, dann dürfen die R_i kein Attribut A haben. Erreicht wird dies, indem ggf. die Tupelvariable eingeführt wird.
3. Die Unteranfrage für t_2 darf keine Nullwerte liefern.

Wir haben im Abschnitt `FROM`-Teil bereits festgelegt, dass sämtliche Tupelvariablen automatisch erzeugt werden. Dies gilt in natürlicher Weise auch für Unterabfragen. Der Prozess erstreckt sich von Einführung von Tupelvariablen, falls vorher keine Vorhanden waren, bis hin zur Umbenennung bereits vorhandener, sodass niemals identische Tupelvariablen innerhalb einer SQL-Anfrage vorkommen können. Damit erfüllen wir bereits die Bedingung 1 und 2.

Da es nicht entscheidbar ist, ob eine Unterabfrage `NULL`-Werte liefern kann, vereinfachen wir das Entscheidungskriterium dafür. Wenn eine der zugehörigen Spalten in t_2 so definiert ist, dass `NULL`-Werte laut Definition des Datenbankschemas erlaubt sind, so gehen wir davon aus, dass die Unterabfrage `NULL`-Werte liefern wird. Wir wandeln, in einem solchen Fall, die Unterabfrage nicht um.

Andere Unteranfragen

Ungewöhnlichere Unterabfragen, wie z. B. Unterabfragen unter `FROM`, werden hier nicht betrachtet. Im Allgemeinen werden solche Unterabfragen kaum gebraucht und machen die Anfrage meist nur viel komplexer als notwendig. Der vorgestellte Parser unterstützt außerdem keine Unterabfragen unter `FROM`. Er versteht nur Listen von Relationen mit Tupelvariablen.

4.3.6 Operatorenvielfalt

Im folgenden Abschnitt soll geklärt werden wie mit verschiedenen Schreibweisen von ein- und demselben Ausdruck umgegangen werden soll. Betrachtet man sich z. B. `A > 5` ist dieser Ausdruck äquivalent mit `5 < A`. Wenn wir wissen, dass `A` ein ganzzahlige Variable ist, dann sind auch folgende Äquivalenzen wahr: `A >= 6` sowie `6 <= A`. Wir betrachten nun zwei verschiedene Ansätze, die diese Probleme lösen sollen. Der erste Ansatz beschäftigt sich damit, alle implizierten Schreibweisen mit in die Formel aufzunehmen. Damit stellt man sicher, dass sich alle korrekten Schreibweisen einer Formel in der Anfrage befinden. Der zweite Ansatz beschäftigt sich damit, nur bestimmte Schreibweisen zuzulassen und alle anderen durch die zulässigen zu ersetzen.

Analyse der unterschiedlichen Operatoren

Wir unterscheiden im wesentlichen zwei Arten von Operatoren. Zunächst haben wir Operatoren, bei denen wir die Operanden in beliebiger Reihenfolge anordnen können. Wir nennen diese Operatoren, kommutative Operatoren. Dies sind `{ =, AND, OR, +, *}`. Alle anderen Operatoren sind demnach nicht kommutativ, da wir die Reihenfolge der Operanden nicht ändern können, ohne den Ausdruck zu verfälschen. Zu dieser Gruppe von Operatoren zählen `{ >=, >, <, <=, -, /, }`. Eine dritte Gruppe bilden unäre Operatoren, die nur einen Operanden kennen. Diese Gruppe ist aber im Sinne der Anordnung uninteressant, da wir hier keine mehrdeutig aufgeschrieben Varianten haben.

Ziel ist es nach wie vor, eine gewisse Ordnung zu definieren, so dass nach der Standardisierung semantisch gleiche Anfragen auch syntaktisch gleich sind. Betrachten wir nun einen Ausdruck mit einem Operator op und n Operanden, die wir fortlaufen mit $child_1$, ..., $child_i$, ..., $child_n$ bezeichnen. Ist op ein kommutativer Operator, so können wir die Operanden $child_i$ beliebig anordnen ohne den Sinn des Ausdrucks zu verändern. Dies tun wir, indem wir eine bestimmte Sortierreihenfolge etablieren. Diese finden wir im späteren Abschnitt 4.3.7.

Diese Sortierung ist nicht möglich, wenn op nicht kommutativ ist. In einem solchen Fall müssen wir also die äquivalenten Schreibweisen des Ausdrucks mit in unsere Formel aufnehmen und konjunktiv mit dem Originalausdruck verknüpfen. Im folgenden Beispiel sei angenommen, dass a vom Typ `INT` ist. Zu beachten ist, dass im Beispiel der Operator > nicht kommutativ ist und daher all seine weiteren Schreibweisen hinzugefügt werden müssen.

```
Eingabe:
SELECT * FROM emp e WHERE e.sal > 1000

Vervollständigte Form:
SELECT * FROM emp e
```

```
WHERE e.sal > 1000 AND 1000 < e.sal
AND e.sal >= 1001 AND 1001 <= e.sal
```

Zu beachten ist weiterhin, dass wir nach der Umformung vier anstatt einer Formel haben. Diese Formeln müssen auch eine gewissen Ordnung unterzogen werden, da der übergeordnete Operator `AND` wiederum kommutativ ist. Die Art dieser Ordnung wird im Abschnitt 4.3.7 beschrieben.

Wir möchten im folgenden diskutieren, wie wir diese implizierten Formeln hinzufügen können. Im Anschluss daran diskutieren wir, wie eine Sortierordnung aussehen kann.

Hinzufügen implizierter Formeln

Wie bereits im vorherigen Beispiel ersichtlich, sind die hinzugefügten Formeln redundant und tragen nicht effizient zur Beschleunigung der Anfrage bei. Es soll hier lediglich sichergestellt werden, dass alle möglichen äquivalenten Formeln auftreten, da wir nicht wissen, was der Lernende für einen Repräsentanten der Formeln wählen wird. Weiterhin muss bemerkt werden, dass dadurch die gesamte SQL-Anfrage enorm aufgebläht wird. Es ist daher unbedingt notwendig, die Originalanfrage zu speichern. Weiterhin muss das Programm eine Verbindung zwischen den Formeln der Originalanfrage und den Formeln der veränderten, aufgeblähten Anfrage herstellen. Dem Lernenden soll in einem Feedback nur Fehler in der Originalanfrage aufgezeigt werden. Da intern aber mit der aufgeblähten Anfrage gearbeitet wird, muss beim Auftreten eines Fehlers oder Hinweises nachgeschlagen werden, von welchem Teil der Originalanfrage der Teil entstammt, der jetzt den Fehler auslöst.

Im Folgenden listen wir Mengen M_i von Ausdrücken. Finden wir in der zu bearbeitenden SQL-Anfrage eine Formel f, die auf einen Ausdruck $a \in M_i$ passt, dann verknüpfen wir alle Ausdrücke $\{b : b \in M_i \wedge b \neq a\}$ konjunktiv mit f. Dabei sind alle Variablennamen A, B, C keine (komplexen) Ausdrücke. Es handelt sich also jeweils um Blattknoten im Parserbaum. Ferner bezeichnen wir X, Y als numerische Konstanten.

M_1 $\{ A = B - C\ ,\ C = B - A\ ,\ B = A + C \}$
M_2 $\{ A = B \cdot C\ ,\ C = A / B\ ,\ B = A / C \}$
M_3 $\{ A > B - C\ ,\ C > B - A\ ,\ B < A + C \}$
M_4 $\{ A < B - C\ ,\ C < B - A\ ,\ B > A + C \}$
M_5 $\{ A > B, B < A\}$
M_6 $\{ A \geq B, B \leq A\}$
M_7 $\{ A > X, A \geq X + adjust(A)\}$
M_8 $\{ A < X, A \leq X - adjust(A)\}$

Beim Vergleich mit > und < ist es wichtig zu wissen, wie viele Nachkommastellen die numerischen Variablen A und B besitzen. Es sei $places(A)$ die Anzahl der Nachkommastellen der Zahl

A. Dann bezeichnen wir mit $adjust(A) = 1/(10^{places(A)})$, einen angepassten Wert, der sich nach der Stelligkeit der Variable A richtet.

Betrachten wir als Beispiel ein Attribut `salery`, welches als `NUMERIC(4,2)` definiert ist. Wir wissen also, dass `salery` zwei Nachkommastellen hat. Betrachten wir nun die Aussage `salery >= 5`. Wir haben auf einer Seite eine Variable (`salery`) und auf der anderen Seite eine numerische Konstante (`5`). Dieses Muster passt also auf M_7 und auf M_6. In M_7 heißt es $A \geq X + adjust(A)$. Bezogen auf unser Beispiel ist `A=salery` und `x+adjust(salery)=5`. Wir berechnen also:

$$adjust(salery) = 1/(10^2) = 1/100 = 0,01$$

Wir erhalten also `x=4,99`, weil `x+0,01=5`. Somit ergänzen wir unsere Ausgangsformel `salery >= 5` konjunktiv mit `salery > 4,99`. Weiterhin muss jetzt wegen M_6, `5 <= salery`, und wegen M_5, `4,99 < salery`, hinzugefügt werden.

Finden wir Ausdrücke mit $>, <, \leq, \geq$, welche als Argumente Variablen oder Konstanten haben, so unterscheiden wir also grundsätzlich drei Fälle.

Fall 1 (M_5, M_6): Beide Operanden sind Variablen. In diesem Fall ergänzen wir nur den jeweils symmetrischen Operator. Da beide Operanden Variablen sind, macht es weniger Sinn jeweils $\leq, \geq$ oder $<, >$ zu ersetzen, da normalerweise ein künstliches hinzufügen eines Summanden auf einer Seite, die Anfrage unnatürlich aussehen lässt. Es ist aber auch kein Problem diese Ersetzungen dennoch durchzuführen, wenn beide Operanden Variablen sind. Die Anfrage wird dann natürlich noch weiter künstlich aufgebläht.

Fall 2 (M_7, M_8): Einer der beiden Operanden ist eine numerische Konstante und der andere ist eine Variable. In diesem Fall fügen wir alle implizierten Gleichungen hinzu, also insbesondere die Operatoren $\leq, \geq, <, >$. Zu beachten ist hier, dass nicht nur Gleichungen der Form $A > X$ dazu führen, dass alle Ausdrücke von M_7 hinzugefügt werden. Auch wenn eine Gleichung der Form $Var1 \geq 5.2$ auftaucht, werden Ersetzungen durchgeführt. Diese Gleichung passt auf das Muster $A \geq X + adjust(A)$. Angenommen $Var1$ hat maximal eine Nachkommastelle, so würden dann folgende Gleichungen impliziert werden: $\{\ Var1 > 5.1\ ,\ 5.1 < Var1\ ,\ 5.2 \leq Var1\ \}$.

Fall 3: Beide Operanden sind numerische Konstanten. In dem Fall wird die logische Aussage ausgewertet und durch ihren Wahrheitswert ersetzt [0,1].

In einem anschließenden Schritt werden arithmetisch/logische Ausdrücke ausgewertet und durch ihre Ergebnis ersetzt. Dieser Schritt muss BOTTOM-UP geschehen, damit man auch mehrere Ersetzungen nach oben, im Parserbaum, weiterreicht. Haben wir am Ende einen SQL-Ausdruck dessen `WHERE`-Teil aus `false` besteht, dann haben wir eine Anfrage gefunden, die immer das leere Ergebnis liefern wird. In diesem Fall müssen wir natürlich eine Fehlermeldung ausgeben.

Sollten wir durch das Hinzufügen implizierter Formeln jetzt einige Formeln doppelt in unserer Anfrage haben, so werden diese bei der Sortierung entfernt.

Beschränkung der Operatorenvielfalt

Ein weiterer Ansatz das Problem der äquivalenten Formeln anzugehen, ist es bestimmte Operatoren zu »verbieten«. Das soll bedeuten, dass wir verbotene Operatoren definieren, welche am Ende der Umwandlungen nicht mehr in der SQL-Anfrage vorkommen dürfen. Dies wird erreicht, indem wir jeden verbotenen Operator umwandeln in einen zugehörigen, nicht-verbotenen Operator. Das Prinzip ähnelt dem eben Vorgestellten. Wir betrachten wieder die Mengen M_i. Des Weiteren hat jede Menge M_i einen Repräsentanten $r(M_i)$. Finden wir nun in der zu bearbeitenden Anfrage eine Formel f, die auf eine der Ausdrücke $a \in M_i$ passt, so ersetzen wir f mit $r(M_i)$. Folgende Tabelle soll die Mengen und deren Repräsentanten beschreiben.

Im Folgenden sind alle Variablennamen A, B, C keine (komplexe) Ausrücke. Es handelt sich also jeweils um Blattknoten im Parserbaum. Ferner bezeichnen wir X, Y als numerische Konstanten.

i	M_i	$r(M_i)$
1	$\{ A = B - C\ ,\ C = B - A\ ,\ B = A + C \}$	$B = A + C$
2	$\{ A = B \cdot C\ ,\ C = A / B\ ,\ B = A / C \}$	$A = B \cdot C$
4	$\{ A > B - C\ ,\ C > B - A\ ,\ B < A + C \}$	$A > B - C$
5	$\{ A < B - C\ ,\ C < B - A\ ,\ B > A + C \}$	$A < B - C$
6	$\{ A > B, B < A\}$	$A > B$
7	$\{ A \geq B, B \leq A\}$	$A \geq B$
8	$\{ A > X\ ,\ X < A, A \geq X + adjust(X)\ ,\ X \leq A - adjust(X) \}$	$A > X$

Ein Beispiel soll die Prozedur verdeutlichen.

Es sei unsere Ausgangsanfrage:

```
SELECT * FROM testtable WHERE X = 6 - Y
```

Die Formel $X = 6 - Y$ finden wir in M_1 in Form von $A = B - C$. Wir ersetzen nun also $X = 6 - Y$ mit dem Repräsentanten von M_1, und wir bekommen:

```
SELECT * FROM testtable WHERE 6 = X + Y
```

Ein Problem bei der Verwendung von Repräsentanten durch Einschränkung der Operatorenvielfalt ist, dass es zu überlappenden Mengen kommen kann. Damit ist gemeint, dass ein Ausdruck auf mindestens zwei Mengen M_i und M_j mit $i \neq j$ passt. Man muss sich in einem solchen Fall fragen, welchen Ausdruck man als Repräsentanten wählt. Es gibt mehrere Arten mit diesem Problem

umzugehen. Auf der einen Seite könnte man die Mengen so umgestalten, dass alle Mengen M_i jeweils, paarweise disjunkte Mengen darstellen, dann käme es zu keiner derartigen Überlappung. Zum anderen könnte man eine Ordnung vorschreiben. Würde ein Ausdruck *expr* auf die Mengen M_i und M_j mit $i \neq j$ passen wählen wir den Repräsentanten $r(M_i)$ genau dann, wenn $i < j$, ansonsten wählen wir $r(M_j)$.

In unserem Programm verwenden wir nicht diesen Ansatz, sondern wir fügen implizierte Formeln hinzu.

Einschränkungen bei arithmetischen Ausdrücken

Bevor wir zur Diskussion beider Ansätze kommen, müssen wir noch erklären, was die beiden Ansätze bisher nicht leisten können. Für beide Ansätze haben wir angenommen, dass arithmetische Ausdrücke maximal aus zwei Operanden auf der komplexen Seite bestehen. Natürlich können in der Praxis auch komplexere Ausdrücke auftauchen. Diese Arbeit aber, entsteht im Rahmen einer Lernplattform. In der Lehre kommt es selten vor, dass arithmetische Ausdrücke im Übermaß benutzt werden. Wenn sie auftauchen, sind sie auch meistens auf zwei Operanden beschränkt. Es ist kaum von Nöten komplexe arithmetische Ausdrücke in der SQL-Anfrage zu verwenden. Im Rahmen dieser Arbeit betrachten wir solche Ausdrücke also immer mit maximal zwei Operanden auf der komplexeren Seite. Dies begründet sich auch in der zunehmenden Schwierigkeit solche komplexen Ausdrücke zu behandeln, wie wir im Folgenden sehen werden.

Wir wollen uns dennoch mit der Frage beschäftigen: “Wie könnte man komplexere arithmetische Ausdrücke anpassen?”. Die folgenden Ansätze sind nur theoretische Überlegungen und stellen kein vollendetes Konzept dar. Wir gehen in unserem Programm trotz der folgenden Diskussion von der Beschränktheit der arithmetischen Ausdrücke aus.

arithmetische Ausdrücke – Standardisierung

Zunächst betrachten wir den Standardisierungsansatz. Hier möchten wir alle implizierten Gleichungen hinzufügen, sodass wir sicher gehen können, dass alle möglichen äquivalenten Gleichungen auftauchen. Dies gestaltet sich für komplexere arithmetische Ausdrücke schwierig. Es müssen alle Gleichungen, die durch äquivalente Umformungen entstehen können, errechnet werden, um sie dann der Lösung hinzuzufügen. Letztendlich führt uns diese Problematik zur Permutation aller Operanden. Für einen solchen Ansatz müssten alle Operatoren kommutativ sein. Wir behelfen uns in diesem Fall, indem wir die Operatoren $-$ und $/$ umschreiben in $+$ und $*$. Es folgt ein Beispiel:

$$\begin{aligned} A = B + C - D - E + F \quad &\rightarrow \quad A = B + C + (-D) + (-E) + F \\ A = B * C / D / E * F \quad &\rightarrow \quad A = B * C * (1/D) * (1/E) * F \end{aligned}$$

Nun können wir die Reihenfolge der Operanden permutieren. Im letzten Schritt müssen wir auch jeden Operanden auf jede Seite der Gleichung bringen und wieder die Reihenfolge der Operanden permutieren. So erhalten wir alle möglichen Schreibweisen eines komplexen arithmetischen Ausdrucks. Zu beachten ist, dass auf diese Weise schnell, sehr viele Gleichungen produziert werden. Dies macht die Lösung stark unübersichtlich. Zu bemerken ist weiterhin, dass gemischte Ausdrücke bezüglich $+$ und $*$ deutlich schwerer zu behandeln sind, als solche, die nur $+$ oder $*$ enthalten.

arithmetische Ausdrücke – Operatorenbeschränkung

Im zweiten Ansatz gestaltet sich das Problem etwas einfacherer. Wie bereits im ersten Ansatz eliminieren wir die Operatoren $-$ und $/$. Da wir am Ende nur eine Schreibweise als zugelassen betrachten, müssen wir nun festlegen, welche Schritte an jedem Ausdruck ausgeführt werden müssen. Wir entscheiden uns für folgende Konvention: Alle Variablen werden via äquivalenten Umformungen auf die linke Seite der Gleichung gebracht und alle numerischen Konstanten werden auf die rechte Seite der Gleichung gebracht. Im nächsten Schritt entfernen wir doppelte Minus- bzw. Divisionszeichen, aus $-(-2)$ wird also $+2$ und aus $1/1/E$ wird E. Nun rechnen wir den Wert des arithmetischen Ausdrucks auf der rechten Seite aus, da dieser nun nur noch aus Konstanten besteht. Die Linke Seite sortieren wir lexikographisch nach Namen der Variablen. Es muss daran erinnert werden, dass diese Namen automatisch generiert worden sind, sodass sichergestellt ist, dass wir Übereinstimmungen später finden können.

Zusammenfassend kann gesagt werden, dass eine Behandlung von komplexeren arithmetischen Ausdrücken möglich, aber nicht einfach umzusetzen ist. Wir haben nur Ansätze präsentiert, die in einem ausgearbeiteten Zustand das Problem der Standardisierung solcher arithmetischen Ausdrücke lösen könnten. Denkbar wären auch andere Ansätze, wie z. B. mehrfaches substituieren von zwei Operanden zu einer Variablen. Alles in allem bedarf es bei diesem Problem noch weiterer Forschung.

transitiv-implizierte Formeln

Formeln können auch transitiv-impliziert sein. Steht in der Musterlösung die Formel `A>B AND B>C` und der Student hat geschrieben `A>B AND A>C AND B>C`, so sind beide Aussagen logisch identisch. Leider erkennt dies unser bisheriger Ansatz nicht. Um solche Formeln zu erkennen, müssen wir auch alle transitiv-implizierten Formeln hinzufügen. Dabei gibt es verschiedene Fälle.

Existiert in der Formel nur eine Relation R, so wie im Beispiel eben $R = \ ' >'$, dann können wir im Ansatz des Hinzufügens implizierter Formeln die transitive Hülle von R berechnen und die entstandenen Paare der Ausgangsformel hinzufügen. Wollen wir keine implizierten Formeln

hinzufügen, sondern nur bestimmte Schreibweisen erlauben, so können wir zunächst die transitive Hülle R^+ berechnen und danach eine transitive Reduktion durchführen. Zu beachten ist bei diesem Ansatz, dass es nicht für jede Relation eine eindeutige transitive Reduktion gibt.

Komplexer wird das Problem, wenn eine Formel verschiedene Relationen enthält, wie z. B. `A>B AND B=9`. Diese Formel impliziert eine weitere Formel transitiv, weil die Relationen $>$ und $=$ zueinander kompatibel sind. Inkompatible Relationen sind jeweils $>, \geq$ zu $\leq, <$. Die Relation $=$ ist zu jedweder Relation kompatibel. Haben wir also eine Mischformel mit mehreren kompatiblen Relationen, dann ordnen wir das entstehende transitive Paar der allgemeineren Relation zu. In unserem Beispiel entsteht das Paar $(A, 9)$. Da $>$ allgemeiner ist als $=$ ordnen wir $(A, 9)$ der Relation '>' zu. Auch hier können wir die implizierten Formeln der Ausgangsformel hinzufügen (Ansatz Hinzufügen der Implikationen) oder danach die transitive Reduktion berechnen, wie oben bereits erwähnt. Ein Ansatz eines Algorithmus für solche Mischformeln wäre einen Graph zu erzeugen mit Operanden als Knoten und Relation als Kantenbeschriftung. Zwei Knoten sind genau dann miteinander verbunden, wenn sie durch eine Relation R_i miteinander in der Formel verknüpft sind. R_i ist dann auch die Kantenbeschriftung. Wenn auf einem Pfad alle Relationen zueinander kompatibel sind, wird mit üblichen Methoden die transitive Hülle bestimmt, indem neue Kanten eingezeichnet werden. Beschriftet werden diese Kanten mit dem allgemeineren der Relationen, die auf dem Pfad liegt. So erhalten wir alle transitiv-implizierten Paare.

implizierte Domänen

Ein weiterer Problempunkt sind implizierte Domänen von Attributen. Es geht darum zu erfassen welchen Wertebereich einzelne Attribute durch die Formeln in der SQL-Anfrage zugewiesen bekommen. Dabei können semantische Widersprüche entdeckt werden, wie z. B. `A>9 AND A<9`. Diese Widersprüche führen meist zu einer leeren Antwort des SQL-Systems. Eine Hinweis würde dem Lernenden klar machen, dass diese Bedingung wahrscheinlich nicht gewollt ist. Die Erfassung des Wertebereichs deckt aber auch weitere Einsatzgebiete ab. So können auch unnötige Bedingungen erkannt werden, wie z. B. `A>9 AND A>5`. Die letzte Bedingung wird ja bereits durch die erste Bedingung impliziert. Unser Matching-Ansatz würde in diesem Fall die Lösungen nicht unifizieren können, obwohl sie äquivalent sind. Man könnte hier aber auch argumentieren, dass die Lösung des Lernenden unnötige Formeln enthält, die keinen Nutzen für die Anfrage haben, und damit unser System, korrekterweise, keine Übereinstimmung erkennt. Andererseits wäre es hilfreich, wenn ein vorgeschalteter semantic checker solche Probleme erkennt, um entweder den Lernenden vorab ein Feedback zu geben oder, um solche unnötigen Formeln bei der Bearbeitung zu ignorieren.

Da dieses Problem eher in den Bereich semantic checking gehört wird dieses Feature nicht im Programm auftauchen. Will man dem Programm später aber semantische Prüfer vorschalten, wäre dies auf jeden Fall ein wichtiges und sinnvolles Feature.

Diskussion der beiden Ansätze

Ein wesentlicher Punkt beim Vergleich beider Ansätze ist der Aufwand bzw. die Laufzeit.

Betrachten wir zunächst den Ansatz des Hinzufügens von implizierten Formeln. Wir müssen in einer Tiefensuche jede Formel betrachten und mit allen Mengen M_i abarbeiten. Finden wir in einer Menge ein Muster wieder, so wird unsere Formel künstlich aufgebläht. Wir haben also für das Suchen eine maximale Laufzeit von $O(|Formeln| \cdot max\{i : M_i\})$. Das Einfügen der Formeln geschieht in konstanter Zeit $O(1)$, da wir immer eine konstante Anzahl an Formeln ergänzen.

Beim anderen Ansatz werden bestimmte Operatoren verboten. Wir realisieren dieses Verbot wieder über eine Suche. Es muss auch hier jede Formel auf ein Muster in M_i untersucht werden. Wir benötigen für das Suchen in diesem Ansatz also genau so viel Zeit, wie im ersten Ansatz. Auch das Ersetzen der Formeln hat keine Zeitersparnis gegenüber einem Hinzufügen von weiteren Formeln. Es muss bemerkt werden, dass in diesem Fall die Originalformel nicht weiter aufgebläht wird.

Da sich die Laufzeiten der beiden Varianten nicht unterscheiden, müssen andere Kriterien zum Vergleich herangezogen werden. Wichtig für Software ist nicht ausschließlich die Laufzeit, sondern auch die Wartbarkeit. Besonders bei Projekten, die im Rahmen einer Masterarbeit entstehen, ist es wahrscheinlich, dass der Autor sich später nicht mehr um das Projekt kümmern kann. Daher sollte man sich bei den hier vorliegenden Ansätzen fragen, welcher leichter wartbar und erweiterbar ist.

Muss das Programm erweitert werden und wir möchten den Ansatz des Hinzufügen implizierter Gleichungen verwenden, so muss lediglich eine weitere Menge M_k erstellt werden. Der Algorithmus sucht automatisch, dann auch in dieser neuen Menge nach Mustern und würde alle anderen Elemente dieser Menge konjunktiv-verknüpft zur Formel hinzufügen.

Bei der Verwendung von eingeschränkten Operatoren gestaltet sich dieser Ansatz bereits als schwierig. Hier muss man nicht nur die neue Menge M_k angeben, sondern sich auch Gedanken machen, was ein geeigneter Repräsentant dieser Menge ist. Unter Umständen kann das Auswählen eines ungünstigen Repräsentanten zu unerwarteten Problemen, wie dem Verkomplizieren der Anfrage, führen.

Aufgrund dieser Umstände werden in unserer Anwendung implizierte Formeln hinzugefügt.

4.3.7 Sortierung

Ein wesentlicher Aspekt bei der Standardisierung ist die Art der Sortierung. Wir betrachten dazu im Folgenden, ZQL-Parserbäume. Im wesentlichen ist dies ein gewurzelter Baum. Dabei stellen Knoten entweder Operatoren, Konstanten oder Variablen dar. Ein Operator o_1 kann natürlicherweise, als Kindknoten, wiederum einen Operator o_2 haben. Es sei $T(r)$ der gewurzelte Baum mit Wurzelknoten r. Wir bezeichnen mit der Menge $children(r)$, die Kindknoten von r in folgender Art und Weise: $child(r) = \{c_1, c_2, ..., c_n\}$. Dabei erscheint das Kind c_i direkt links vom Kind c_j genau dann, wenn $j = i + 1$. Anders ausgedrückt: Bei einer Breitensuche über $T(r)$ würden wir erst c_i und im Anschluss daran c_j auffinden, genau dann wenn $j = i + 1$.

Es sei $T(r)$ unser ZQL-Parserbaum. Ist der Operator, den r repräsentiert, kommutativ, so können wir alle Kinder c_i in beliebiger Reihenfolge permutieren und würden den Ausdruck, den $T(r)$ darstellt nicht semantisch verändern. Für unsere Standardisierung ist es aber wichtig, dass wir dennoch eine Ordnung auf solchen Operatoren festlegen, damit unsere Parserbäume am Ende vergleichbar sind. Wir verwenden die Ordnung, die in Abbildung 4.4 dargestellt ist. Ein niedrigerer Wert der *order*-Funktion bedeutet dabei eine höhere Priorität.

$r \in Relation$	OR	$\leq$	$\geq$	$>$	$<$	$=$	IS NULL	IS NOT NULL
$order(r)$	1	2	3	4	5	6	7	8

Abbildung 4.4: Reihenfolge der Sortierung eines WHERE-Ausdrucks

Die Kinder vom Baum $T(r)$ müssen so angeordnet werden, dass für alle Kinder paarweise das Folgende gilt. Dabei seien $c_i, c_j \in children(r)$ und r muss kommutativ sein.

$$\forall c_i, c_j : i \neq j \rightarrow (i < j \leftrightarrow order(i) < order(j))$$

In Worten ausgedrückt: Für alle, paarweise verschiedenen, Kinder von $T(r)$ gilt: c_i erscheint genau dann vor c_j im Baum (bezüglich eine Breitensuche), wenn die oben angegebene Ordnung eingehalten wird. Dies wird initial nicht der Fall sein, daher müssen wir die Kinder so umsortieren, dass diese Bedingung erfüllt ist.

Ist r nicht kommutativ, so können wir die Kinder von r nicht umsortieren. Wir verfahren dann rekursiv direkt mit den Kindern weiter, da diese ja wieder kommutative Operatoren sein können. Weiterhin kann es sein, dass eins der Kinder von r eine Konstante oder Variable ist, also ein Blattknoten, der keinen weiteren Baum aufspannt. Der *order*-Wert für eine Konstante ist immer 0 und für eine Variable immer -1. Damit ist sichergestellt, dass auf gleicher Baumebene immer zuerst Variablen, dann Konstanten und dann Teilbäume erscheinen, vorausgesetzt, der Elternknoten repräsentiert einen kommutativen Operator. Sind alle Kindknoten Variablen, so werden diese lexikographisch sortiert. Sind alle Kindknoten Konstanten, so wird der entsprechende Ausdruck

ausgewertet, da wir dann die Situation haben, dass ein Operator mit Operandenkonstanten gegeben ist und dieser Ausdruck ist auswertbar. Wir ersetzen dann den Baum $T(r)$ mit $eval(T(r))$, wobei *eval* den arithmetisch/logischen Ausdruck auswertet und sein Ergebnis zurückliefert.

Die Abbildung 4.5 soll den bisherigen Prozess darstellen. Dabei sortiert die Methode `sortiere_menge` die eingegebene Menge gemäß der Ordnung in Abbildung 4.4.

```
sortiere(rootnode r) {
   if( children(r) = {} )
      return;
   if( kommutativ(r) ) {
      sortiere_menge(children(r));
   }
   foreach( c in children(r)) {
      sortiere(c);
   }
}
```

Abbildung 4.5: Pseudocode: Erster Teil der Sortierung

Folgendes Beispiel soll diese Arbeitsweise verdeutlichen:

Wir betrachten den Ausdruck `a>5 AND (b=2 OR c IS NULL) AND (d=5 OR a<6)`. Dieser ist in Abbildung 4.6 als ZQL-Parserbaum zu sehen. Ziel ist es nun, diesen Baum nach unserem Verfahren zu sortieren.

Wir bemerken, dass wir einen kommutativen Operator (AND) als Wurzelknoten haben und sortieren daher die unmittelbaren Kinder von AND, also `{>,OR,OR}` zu `{OR,OR,>}`, wie in Abbildung 4.7 zu sehen ist.

Wir fahren dann fort und sortieren die Kindknoten c_1, c_2 und c_3 rekursiv. Am Ende erhalten wir den Ausdruck `(b=2 OR c IS NULL) AND (a<6 OR d=5) AND a<5`, welcher durch den Baum in Abbildung 4.8 dargestellt wird.

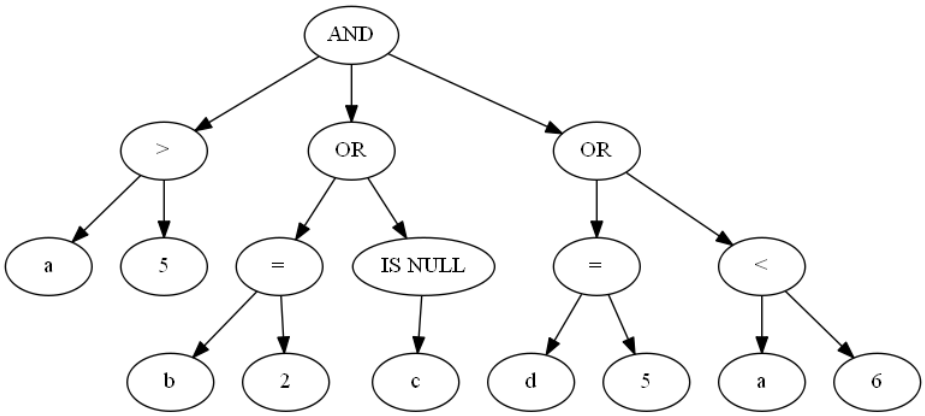


Abbildung 4.6: Sortierbeispiel: Ausgangsbaum

Abbildung 4.7: Sortierbeispiel: Sortierung der ersten Ebene

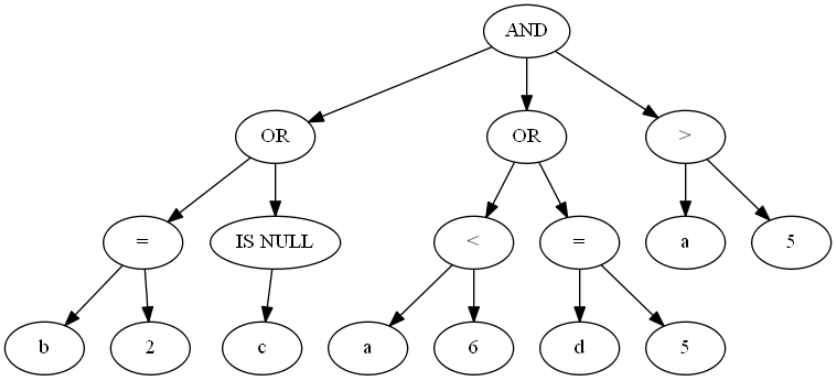


Abbildung 4.8: Sortierbeispiel: Vorläufiges Ende des Sortierens

Wenn wir uns Abbildung 4.8 genauer betrachten, so fällt auf, dass keine eindeutige Ordnung zugrunde liegt. Die beiden Bäume $T(c_1)$ und $T(c_2)$ repräsentieren beide den Operator `OR`. Es ist mit den bisherigen Regeln nicht klar, was passiert, wenn zwei Geschwisterknoten, den gleichen Operator repräsentieren.

Haben wir also den Fall, dass zwei Kindknoten c_i, c_j von $T(r)$ den gleichen Operator repräsentieren müssen wir rekursiv die beiden Teilbäume $T(c_i)$ und $T(c_j)$ untersuchen. Wir führen dabei eine gleichzeitge und schrittweise Tiefensuche in beiden Bäumen durch, indem wir sukzessive einen Schritt einer Tiefensuche in einem Baum, dann im anderen Baum durchführen und schließlich einen nächsten Schritt, in gleicher Weise durchführen. Wir bezeichen die Knoten, bei denen sich die Tiefensuche in einem aktuellen Moment befindet als v_i für $T(c_i)$ und v_j für $T(c_j)$. Sobald in einem Schritt $v_i \neq v_j$ ist, entscheiden wir anhand von $order(v_i)$ und $order(v_j)$, welcher Teilbaum vor dem anderen erscheinen soll. $T(c_i)$ erscheint damit vor $T(c_j)$, wenn $order(v_i) < order(v_j)$. Dabei schließen wir Blattknoten von der Betrachtung aus, es sind also nur innere Knoten von Bedeutung.

Ist die Tiefensuche beendet und gab es zu keinem Zeitpunkt zwei innere Knoten in beiden Bäumen mit unterschiedlichen Operatoren, so sind offensichtlich die beiden Teilbäume $T(c_i)$ und $T(c_j)$ strukturell gleich, sie haben also identische innere Knoten. In diesem Fall unterscheiden sich beide Bäume nur durch ihre Blattknoten. Wir bezeichnen die Blattknoten von $T(c_i)$ mit $leaf(c_i)$ und in entsprechender Weise $leaf(c_j)$ die Blattknoten von $T(c_j)$. Wir sortieren diese Blattknotenmengen jetzt lexikographisch. Es ist offensichtlich, dass beide Blattknotenmengen gleich mächtig sind, da ansonsten bereits ein Unterscheid beim Tiefensuchvergleich aufgefallen wäre. Wir gehen jetzt

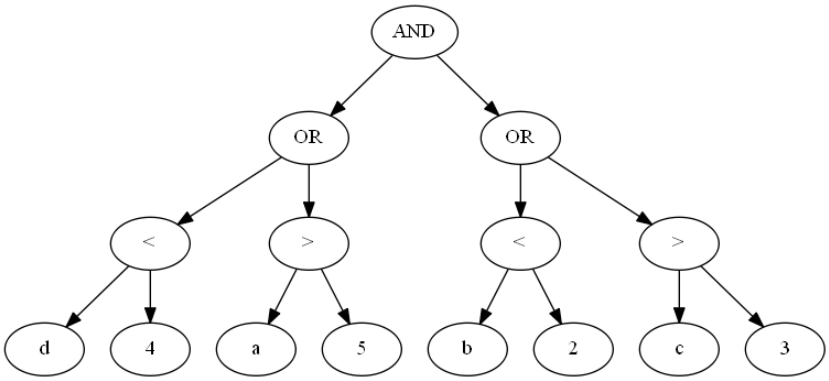


Abbildung 4.9: Sortierbeispiel: strukturell identische Teilbäume

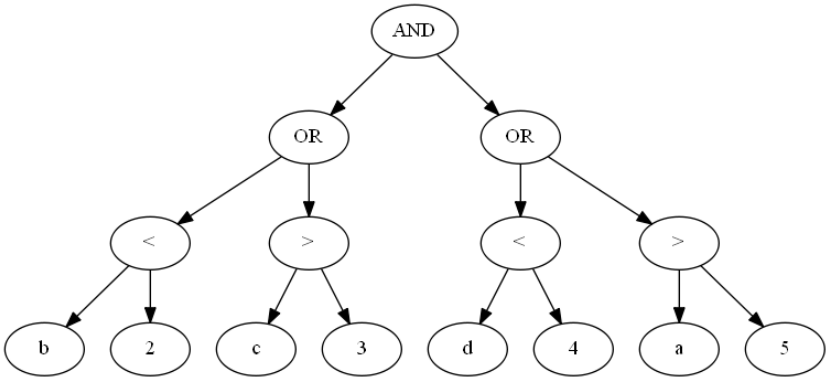


Abbildung 4.10: Sortierbeispiel: umsortierte Teilbäume

sukzessive beide, sortieren Blattknotenmengen durch und vergleichen jeweils das i. Element dieser Mengen, wobei $1 \leq i \leq n$, mit $n = |leaf(c_i)| = |leaf(c_j)|$. Unterscheiden sich die i. Elemente der zwei Mengen, dann erscheint $T(c_i)$ vor $T(c_j)$, wenn das i. Element in $leaf(c_i)$ lexikographisch kleiner ist als das i. Element in $leaf(c_j)$.

Zu beachten ist, dass wenn innerhalb eines Teilbaumes $T(c_i)$ wieder die Situation eintritt, dass es zwei Kinder $d_i, d_j \in children(c_i)$ gibt, mit $d_i = d_j$, aber $i \neq j$, dann wird dieser Fall zuerst behandelt. Wir behandeln also Kinder mit gleichen repräsentierten Operatoren von den Blättern aus, also Bottom-Up.

Als Beispiel betrachten wir den Baum in Abbildung 4.9.

Wie wir sehen, sind alle inneren Knoten gleich. Es unterscheiden sich nur die Blattknoten, weshalb eine simultane Tiefensuche über $T(c_1)$ und $T(c_2)$ keine Unterschiede aufgezeigt hat. Die Menge der Blattknoten sind $leaf(c_1) = \{d, 4, a, 5\}$ und $leaf(c_2) = \{b, 2, c, 3\}$. Wir sortieren diese Mengen nun und erhalten: $leaf'(c_1) = \{4, 5, a, d\}$ und $leaf'(c_2) = \{2, 3, b, c\}$. Wir gehen nun sukzessive beide Mengen durch und vergleichen zunächst das 1. Element beider Mengen. Diese beiden ersten Elemente unterscheiden sich bereits und weil $4 > 2$ müssen die beiden Bäume $T(c_1)$ und $T(c_2)$ vertauscht werden. Wir erhalten also den finalen, sortierten Baum, der in Abbildung 4.10 zu sehen ist.

Ist das Sortieren abgeschlossen, entfernen wir noch unnötige Duplikate. Hat der Operator `AND` zwei gleiche Operanden o_1, o_2, die äquivalent sind ($T(o_1) \equiv T(o_2)$), so können wir einen der beiden Operanden streichen. Gleiches gilt für den Operator `OR`. Durch die Sortierung stehen solche Duplikatskandidaten immer nebeneinander. Wir durchsuchen den Baum daher iterativ nach Duplikaten und entfernen diese.

4.3.8 GROUP BY / HAVING

Bisher haben wir den SELECT-, FROM- und WHERE-Teil einer SQL-Anfrage bearbeitet. Anfragen, die einen GROUP BY-Teil enthalten müssen auch nach festen Regeln angepasst werden. Da die Reihenfolge innerhalb dieses Teils unwichtig ist, sortieren wir die Spaltennamen lexikographisch. Die Tupelvariablen sind bereits automatisch benannt nach dem Schema `aN` mit $N \in \{1, 2, ...\}$ (siehe Unterabschnitt 4.3.3). Demzufolge erscheinen zuerst Tabellennamen und dann Aggregationsfunktionen.

Der HAVING-Teil wiederum ähnelt von seiner Struktur dem WHERE-Teil. Es handelt sich ebenso um einen komplexeren Ausdruck. Wir behandeln daher den HAVING-Teil so, wie den WHERE-Teil im Unterabschnitt 4.3.4.

Nicht wichtig für die einheitliche Anpassung ist, ob die Tabellen bzw. Aggregationsfunktionen im GROUP BY-Teil auch im SELECT-Teil vorkommen. Dies könnte aber auf einen ungewollten Fehler des Lernenden hinweisen. Diese Information wird daher im Preprocessing (siehe Abschnitt 4.2.3) mit aufgenommen.

4.3.9 ORDER BY

Der ORDER BY-Teil kann nicht im Sinne der Standardisierung angepasst werden. Die hier angegebene Reihenfolge kann nicht verändert werden ohne den Sinn des Ausdruckes zu ändern. Die Anpassung beschränkt sich also auf das Umbenennen der Tupelvariablen, wie es im Unterabschnitt 4.3.3 beschrieben ist.

Interessant für das Preprocessing sind allerdings andere Informationen über ORDER-BY. So merken wir uns, ob die Anfrage ORDER BY-Klauseln mit einem expliziten ASC enthält, da dies unnötig ist.

4.3.10 Selbstverbunde

Bevor wir alle Schritte zusammenfassen, möchten wir uns mit Selbstverbunden beschäftigen. Dazu betrachten wir das folgende Beispiel. Dies ist konstruiert und dient nur zur Veranschaulichung eines Problems.

```
Musterlösung:
SELECT * FROM emp e, emp f, dept d WHERE e.sal < 1000 AND e.id = f.id


Anfrage des Lernenden:
SELECT * FROM emp e, emp f, dept d WHERE f.sal < 1000 AND e.id = f.id
```

Wir bemerken, dass beide Anfragen semantisch äquivalent sind. Allerdings prüfen wir in der Musterlösung das Gehalt mit der ersten Tupelvariable und in der Anfrage des Lernenden mit der zweiten Tupelvariable. Unser Programm erkennt daher keine Äquivalenz zwischen beiden Anfragen. Problem ist hierbei der Selbstverbund. Wir können daher im `WHERE`-Teil die Tupelvariablen e und f beliebig tauschen und erhalten immer das gleiche Ergebnis.

Wir können Selbstverbunde erhalten, indem wir aus einer Anfrage mit Selbstverbund, mehrere Anfragen konstruieren. Dabei permutieren wir für jede erstellte Anfrage die Tabellen unter `FROM` und substituieren dann im `WHERE, GROUP BY` und `ORDER BY`-Teil die Tupelvariablen gemäß der jeweiligen Permutation. Um den Prozess zu verdeutlichen, betrachten noch einmal unser Beispiel.

Wir erstellen aus dem `FROM`-Teil eine Liste der Tupelvariablen: `[e, f, d]`. Dann gruppieren wir die Liste, so dass sich alle gleichen Tabellen in einer Gruppe befinden: `[ [e, f], [d] ]`. Nun erstellen wir alle Permutationen, indem alle inneren Gruppen permutiert werden. Danach entfernen wir die Gruppierungen wieder und erhalten die folgenden, permutierten Listen:

```
[ [e, f], [d] ]  =>  [ e, f, d ]
[ [f, e], [d] ]  =>  [ f, e, d ]
```

Auf Grundlage dieser Listen, erzeugen wir Substitutionen. Wir substituieren jeweils ein Element aus der Originalliste durch ein Element der erzeugten Permutationslisten. In unserem Fall entstehen dann folgende Substitutionen:

σ_1 = [e/e, f/f, d/d]
σ_2 = [e/f, f/e, d/d]

Diese Substitutionen wenden wir nun auf den `WHERE`-Teil unserer Musterlösung an und erhalten:

```
Musterloesung 1 (σ1):
SELECT * FROM emp e, emp f, dept d
WHERE e.sal < 1000 AND e.id = f.id
```

```
Musterloesung 2 (σ2):
SELECT * FROM emp e, emp f, dept d
WHERE f.sal < 1000 AND f.id = e.id

Musterloesung 2 (σ2, sortiert):
SELECT * FROM emp e, emp f, dept d
WHERE f.sal < 1000 AND e.id = f.id

Anfrage des Lernenden (unverändert):
SELECT * FROM emp e, emp f, dept d WHERE f.sal < 1000 AND e.id = f.id
```

Zu beachten ist, dass in Anfrage 2 auch das Verbundskriterium getauscht wird. Aufgrund unserer Sortierordnung, fällt diese Vertauschung dann aber nicht mehr auf, wie wir in Anfrage 3 sehen können. Weiterhin beobachten wir, dass die unveränderte Anfrage des Lernenden nun auf eine der entstandenen Musterlösungen passt.

4.3.11 Abschluss

Wir fassen die einzelnen Schritte noch einmal kurz zusammen: Zunächst haben wir den FROM-Teil der Anfrage vereinheitlicht, indem wir einheitliche Tupelvariablen erzeugt haben, nachdem alle Tabellen im FROM-Teil lexikographisch sortiert wurden. Alle neu erzeugten Tupelvariablen wurden im Rest der Anfrage korrekt eingesetzt bzw. ersetzt. Finden wir in diesem Schritt im FROM-Teil bereits zwei gleiche Tabellen mit unterschiedlichen Tupelvariablen, so könnte ein Selbstverbund dadurch entstehen. Um mehrdeutigen zu vermeiden, erzeugen wir eine Kopie der vorliegenden Anfrage und vertauschen die gleichen Tabellennamen im FROM-Teil. Wir erhalten somit für jeden Selbstverbund einen zusätzlich zu untersuchenden Baum. Am Ende prüfen wir ob einer dieser Bäume syntaktisch äquivalent ist mit einem der erzeugen Bäumde, der zweiten Anfrage. Danach haben wir den WHERE-Teil bearbeitet. Wir haben zunächst unnötige Klammern entfernt und die Formeln in die KNF überführt. Danach haben wir einfache syntaktische Varianten ersetzt um eine einheitlichere Darstellung zu erhalten. Dazu gehörte es auch Unteranfragen aufzulösen oder, wenn nicht möglich, in eine EXISTS-Unteranfrage zu überführen. Eine der letzten Schritte war das Behandeln der Vielfalt der Operatoren. Hier haben wir für alle nicht-kommutativen Operatoren ihre alternativen Schreibweisen konjunktiv-verknüpft mit in die Formel aufgenommen. Nach diesem Schritt müssen wir nun die KNF wiederherstellen, da diese durch hinzufügen von AND-Knoten zerstört sein kann. Schlussendlich haben wir den gesamten WHERE-Teil sortiert, um eine Vereinheitlichung zu erreichen. Anschließend sortierten wir die Attribute innerhalb des GROUP BY-Statements lexikographisch. Der Ausdruck im HAVING-Teil, wurde genauso behandelt wie der Ausdruck des WHERE-Teils.

4.4 Weitere Betrachtungen

4.4.1 JOINS

Wir betrachten in diesem Abschnitt die Behandlung von JOINs. Wir untersuchen wie wir die unterschiedlichen Verbundstypen vereinfachen oder in eine einheitliche Form umwandeln können. Weiterhin interessiert uns die Frage, wie wir unnötige JOINs erkennen können. Wir diskutieren in wie weit eine automatische Entfernung von unnötigen JOINs Sinn macht und wie das Feedback für den Lernenden aussehen kann.

Vorab möchten wir bemerken, dass die Standardisierung und Eliminierung von JOINs eine fortgeschrittene Strategie darstellt. Diese wird daher nicht im Prototypen auftauchen. Dennoch betrachten wir die theoretischen Grundlagen ausreichend und tiefgründig, so dass eine nachträgliche Implementation leicht möglich sein wird.

CROSS JOIN

Ein `CROSS JOIN` zweier Relationen liefert das kartesische Produkt beider zurück. Das Schlüsselwort `CROSS JOIN` unter `FROM` markiert diesen. Das Schlüsselwort kann allerdings auch weggelassen und durch ein Komma ersetzt werden. Um eine einheitliche Darstellung dieses Verbundtypes zu gewährleisten, werden wir das Schlüsselwort `CROSS JOIN` ersetzen, so dass wir eine Liste von Tabellen unter `FROM` erhalten, welche mit Komma getrennt sind.

NATURAL JOIN

Ein natürlicher Verbund (NV) ist ein Verbund im `FROM`-Teil. Er wird markiert durch die Schlüsselworte `NATURAL JOIN`. Der Verbund zweier Relationen erfolgt über die Attribute, die in beiden Relationen die gleiche Bezeichnung haben. Gibt es keine solchen Attribute, ist das Ergebnis das kartesische Produkt beider Relationen.

Es gibt zwei Möglichkeiten natürliche Verbunde zu standardisieren. Die erste Möglichkeit besteht darin, alle Verbundskriterien zu durchsuchen. Stellt man fest, dass die Attribute bei einem Verbundskriterium identisch sind, so handelt es sich um einen natürlichen Verbund. Man könnte diese Kriterien nun streichen und mit dem Schlüsselwort `NATURAL JOIN` explizit kennzeichnen, dass es sich um einen NV handelt. Bei dieser Möglichkeit können aber Probleme auftreten. Angenommen der Lernende L reicht eine Lösung ein, die einen natürliche Verbund zwischen `t1` und `t2` explizit enthält. Weiterhin gibt es kein namensgleiches Attribut in `t1` und `t2`, so dass der natürliche Verbund im Prinzip ein kartesisches Produkt ist. In der Musterlösung stehen `t1` und `t2` nur unter `FROM` um eben dieses Produkt zu erzeugen. Die Methode 1 würde nun garkeine Anpassung

```
Vorher:
SELECT t1.col1 FROM table1 t1 NATURAL JOIN table2 t2

Danach:
SELECT t1.col1 FROM table1 t1, table2 t2 WHERE t1.id=t2.id
```

Abbildung 4.11: Beispiel: Standardisierung eines natürlichen Verbundes

machen, da es kein Kriterium für einen potentiellen natürlichen Verbund in der Anfrage des L findet. Als Folge können wir die beiden Anfragen nicht matchen, obwohl beide das gleiche tun.

Die zweite Möglichkeit besteht darin explizite natürliche Verbunde umzuschreiben, in dem wir das Schlüsselwort `NATURAL JOIN` entfernen und beide Relationen durchsuchen nach gleichnamigen Attributen. Haben wir solche gefunden, so fügen wir diese explizit als Verbundskriterien ein. Haben wir keine solche Kriterien gefunden, handelt es sich um ein kartesisches Produkt. Wir fügen in diesem Fall die zweite Tabelle unter `FROM` hinzu und eliminieren das Schlüsselwort `NATURAL JOIN`.

Beide Methoden haben einen gewissen Aufwand zu bewältigen. Bei Methode 1 müssen wir alle Verbundskriterien durchsuchen, was im schlimmsten Fall alle atomaren Formeln im `WHERE`-Teil sein können. In Methode 2 müssen wir zwei Relationen paarweise nach gleichnamigen Attributen durchsuchen. Wir entscheiden uns für die 2. Methode, da Methode 1, wie oben erwähnt, Schwächen bei der Umwandlung hat. Weil wir den `NATURAL JOIN` unter `FROM` also auflösen, schreiben wir diesen gleich als `INNER JOIN` unter die `WHERE`-Bedingung, wie das Beispiel 4.11 zeigt (angenommen *table1* und *table2* haben beide eine gleichnamige Spalte namens *id*).

Die eben aufgeführten Transformationen sind nur möglich, wenn der `SELECT`-Teil nicht aus der Wildcard * besteht. Bei einem NV wird in einem solchen Fall die Spalte mit dem gleichnamigen Attribut in der zweiten Tabelle eliminiert. Bei einem Verbund mit expliziter Angabe des Verbundskriterium wird dies nicht realisiert. Deswegen sind natürliche Verbunde, die im `SELECT`-Teil nur das Wildcard * enthalten, nicht in explizite Verbunde unter `WHERE` überführbar.

OUTER JOIN

Bei einem `OUTER JOIN` unterscheiden wir zwischen einem `LEFT OUTER JOIN` und einem `RIGHT OUTER JOIN`. Ein `FULL OUTER JOIN` ist die Verbindung, also ein `UNION` vom linken und rechten `OUTER JOIN`. Wir betrachten daher exemplarisch zunächst den `LEFT OUTER JOIN`, da sich die Betrachtungen dann äquivalent auf den rechten Verbund und damit auf den `FULL OUTER JOIN` übertragen lassen.

Zunächst ist zu klären, wie der `OUTER JOIN` alternativ formuliert werden kann. Dazu gehen wir kurz darauf ein, was ein `LEFT OUTER JOIN` genau macht. Die "linke" Tabelle wird vollständig aufgeführt. Dann wird jeder Eintrag der Tabelle sukzessive betrachtet. Gibt es im aktuellen Eintrag

einen Verbundspartner in der “rechten” Tabelle (vorgegeben durch das Verbundskriterium), dann werden die Daten der “rechten” Tabelle an den Eintrag angehangen. Ist dies nicht der Fall, wird der Eintrag mit *NULL* Einträgen aufgefüllt.

Um die Umwandlung eines `LEFT OUTER JOIN` zu demonstrieren, betrachten wir folgendes Beispiel:

```
SELECT e.empno, e.job, d.dname
FROM dept d
LEFT JOIN emp e ON e.deptno = d.deptno
WHERE e.sal > 2000
ORDER BY e.empno
```

Abbildung 4.12: OUTER JOIN Beispiel

Wir haben also zwei verschiedene Mengen von Einträgen. Zum einen Einträge, die einen Verbundspartner besitzen und Einträge, die keinen solchen Partner haben. Anbieten würde sich an dieser Stelle die Verwendung einer Vereinigung (`UNION`) der beiden, eben genannten Mengen. Dabei ist die Umwandlung in ein solches Statement mit einem `UNION` keinesfalls trivial und unterliegt einigen Einschränkungen.

Ziel ist es also zwei SQL-Anfragen zu formulieren, die jeweils für die oben genannten Mengen stehen. Anschließend werden diese Mengen mit einem `UNION` verbunden.

Für die SQL-Anfrage, welche die Menge der Verbundspartner angibt, übernehmen wir zunächst den `SELECT`-Teil. Die zwei Tabellen, die am Verbund beteiligt sind werden im `FROM`-Teil nun als `CROSS JOIN` aufgeschrieben, also mit Komma getrennt. Der `WHERE`-Teil wird ebenfalls vom gegebenen `OUTER JOIN`-Statement übernommen und durch die Angabe des Verbundskriteriums ergänzt. Wir erhalten also die folgende Anfrage bezüglich unseres Beispiels:

```
SELECT e.empno, e.job, d.dname
FROM dept d, emp e
WHERE d.deptno = e.deptno
AND e.sal > 2000
ORDER BY e.empno
```

Abbildung 4.13: OUTER JOIN Umwandlung, 1. Teil

Nun muss noch die Teilmenge aller Tupel hinzugefügt werden, die keinen Verbundspartner haben. Wir übernehmen zunächst wieder den `SELECT`-Teil der Ursprungsanfrage, ersetzen aber alle Spaltennamen der “rechten” Tabelle mit `NULL`, weil diese Werte aufgrund fehlender Verbundspartner nicht belegt sind. Diese Ersetzungen müssen ebenso im `WHERE`-Teil stattfinden. Die rechte Tablle wird aus dem `FROM`-Teil gestrichen. Nun passen wir den `WHERE`-Teil an. Hier könnte man auf die Idee kommen mit einem `NOT IN` Semijoin zu prüfen, dass auch nur Tupel ohne Verbundspartner aufgenommen werden. Erlaubt das Verbundskriterium in der “linken” Tabelle `NULL`-Werte, so

würden diese Einträge nicht erscheinen. Bei einem LEFT OUTER JOIN sind diese Tupel aber mit erfasst. Es bietet sich daher eine NOT EXISTS-Unterabfrage an.

```
SELECT NULL, NULL, d.dname
FROM dept d
WHERE NOT EXISTS(
    SELECT 1 FROM emp e WHERE d.deptno = e.deptno )
AND NULL > 2000
ORDER BY e.empno
```

Abbildung 4.14: OUTER JOIN Umwandlung, 2. Teil

Nun müssen beide Anfragen mit einem UNION ALL verknüpft werden. Dabei können wir das ORDER BY-Statement ausklammern, da dieses erst auf dem Datensatz, der mit UNION ALL erzeugt wurde, ausgeführt wird. Unser Endergebnis bezogen auf unser Beispiel lautet also:

```
SELECT e.empno, e.job, d.dname FROM dept d, emp e
WHERE d.deptno = e.deptno AND e.sal > 2000
    UNION ALL
SELECT NULL,NULL,d.dname FROM dept d
WHERE NOT EXISTS(
  SELECT 1 from emp e where d.deptno = e.deptno)
AND NULL > 2000
ORDER BY empno
```

Abbildung 4.15: OUTER JOIN Umwandlung, finaler Schritt

Wie man bereits am Beispiel sieht, könnte man im WHERE-Teil den Ausdruck NULL > 2000 zu NULL vereinfachen oder gar die zweite Anfrage komplett streichen, da sie nie erfüllt ist.

Eine Besonderheit stellt das GROUP BY-Statement dar. Dies wurde nicht im Beispiel benutzt, daher betrachten wir nun im Folgenden, wie damit zu verfahren ist.

Wird in der OUTER JOIN Abfrage nach mindestens einem Attribut der "rechten" Tabelle gruppiert, so müssen wir die Gruppierung in eine weitere Anfrage schachteln. Dazu bauen wir die Anfrage, wie oben beschrieben, auf und streichen das GROUP BY-Statement zunächst. Aggregationsfunktion *agg(x)* werden ersetzt zu *x*. Um die dann entstandene, mit UNION ALL verknüpfte SQL-Anfrage A schachteln wir eine weitere Anfrage B. Der SELECT-Teil von B entspricht dem, der ursprünglichen OUTER JOIN-Anfrage. Der FROM-Teil von B entspricht der konstruierten Anfrage A. Einen WHERE-Teil von B gibt es nicht, da dieser bereits in der Anfrage A eingearbeitet wurde. Ergänzt wird die Anfrage B mit dem GROUP BY- und ORDER BY-Statement der Ursprungsanfrage.

Existiert im GROUP BY-Teil der Originalanfrage kein Attribut der "rechten" Tabelle, ist die eben genannte Schachtelung nicht notwendig. Es reicht hier aus den GROUP BY-Teil einfach in die Konstruierten Teilanfragen mit einzubeziehen. Grund dafür ist, dass wir disjunktive Gruppierungen in beiden Anfragen erhalten und diese ohne Probleme mit UNION ALL vereinigen können.

Dies waren alle notwendigen Schritte um einen LEFT OUTER JOIN äquivalent in eine komplexe Anfrage mit UNION ALL und Semijoins zu überführen. Zu Bemerken ist an dieser Stelle, dass ein RIGHT OUTER JOIN entsprechend behandelt wird, nur mit umgekehrten Seiten. Einen FULL OUTER JOIN erhalten wir durch ein UNION vom LEFT OUTER und RIGHT OUTER JOIN.

Nach dieser Betrachtung müssen wir jedoch feststellen, dass die Umwandlung eines OUTER JOIN recht vielschichtig und aufwendig ist. Es ist zwar möglich diesen äquivalent umzuformen, allerdings wird der Ausdruck dadurch enorm aufgebläht. Weiterhin funktioniert diese Umwandlung nur in eine Richtung problemlos, nämlich vom JOIN zum UNION-Ausdruck. Umgekehr ist es sehr aufwendig festzustellen, ob ein UNION ALL auch ein OUTER JOIN ist, da man zum Beispiel Vergleiche mit NULL-Werten (wie im Beispiel: NULL > 2000) gewöhnlich nicht ausschreibt oder gar entfernt. Unser Programm soll OUTER JOIN daher nicht in einen UNION ALL Ausdruck umwandeln. Wir beschränken uns in der Praxis also darauf, einen OUTER JOIN unter FROM rein syntaktisch zu erkennen und nicht umzuwandeln.

INNER JOIN

Unter einem INNER JOIN verstehen wir den Verbund zweier Tabellen unter FROM, mit den Schlüsselworten INNER JOIN. Ein solcher Verbund kann leicht als CROSS JOIN mit Auswahl in der WHERE-Bedingung formuliert werden. Dazu entfernen wir den INNER JOIN aus dem FROM-Teil und erzeugen einen CROSS JOIN. Die Verbundbedingungen werden unter WHERE eingefügt.

Ein explizit aufgeschriebener INNER JOIN kann auch ein NATURAL JOIN sein. Da wir solche Verbunde aber auch als CROSS JOIN aufschreiben wollen (siehe oben), müssen wir diesen Spezialfall nicht gesondert betrachten. Das folgende Beispiel soll die Umwandlung von INNER JOIN verdeutlichen:

Eingabe:
```
SELECT * FROM emp e INNER JOIN dept d ON e.deptno.id = d.deptno
```

Umwandlung:
```
SELECT * FROM emp e, dept d WHERE e.deptno = d.deptno
```

Abbildung 4.16: Umwandlung von INNER-JOIN

JOIN-Eliminierung

Im vorherigem Abschnitt haben wir bereits alle üblichen Verbunde betrachtet und gezeigt, wie wir diese äquivalent umformen können. Wir haben erreicht, dass alle Verbunde unter FROM ersetzt

wurden zu `CROSS JOIN` mit Auswahlkriterien unter `WHERE`. Wir möchten daher in diesem Abschnitt klären, welche Möglichkeiten es gibt, die so entstandenen Verbunde zu eliminieren, wenn sich herausstellt, dass sie überflüssig sind.

Wenn nur Schlüsselattribute einer Tupelvariable X benutzt werden und diese mit einer anderen Tupelvariable Y verglichen werden, dann ist X überflüssig.

Ziel dieses Abschnittes ist es, eine Idee vorzustellen, wie redundante Verbunde mit Hilfe von referentiellen Integritätsbedingungen (RI) gefunden werden können. Die mögliche Anwendung ist natürlicherweise die Entfernung einer Relation bei dem der Verbund aus Tabellen besteht, die durch eine RI-Bedingung verbunden ist (nachfolgend als RI-Joins bezeichnet) und bei der die Tabelle, die den Primärschlüssel enthält nur im Join referenziert wird. Ein RI-Join besteht natürlicherweise aus einem Primärschlüssel- und einem Fremdschlüsselattribut.

Das Verfahren wird in der Arbeit [11] näher vorgestellt und läuft im wesentlichen in 4 Schritten ab.

1. Wir bilden zunächst transitive Spaltenäquivalenzklassen von allen Joinprädikaten. Ist also `A=B` und `B=C`, dann können wir transitiv ableiten, dass ebenso `A=C` gilt. Alle drei Spalten, `A,B,C` befinden sich dann in einer Äquivalenzklasse.

2. Alle Tabellen unter `FROM`, die mit RI-Joins verknüpft sind, werden in zwei Gruppen eingeteilt. Tabellen in der R-Gruppe sind entfernbar (removable) und Tabellen in der N-Gruppe sind nicht entfernbar (non-removable). Die notwendige Bedingung, die bestimmt, wann eine Tabelle in die R-Gruppe gehört ist, dass die Tabelle eine parent-table für irgendeine RI ist. Dies ist jedoch nicht die einzige notwendige Bedingung. Andere, zu erfüllende Bedingung, sind weitaus komplexer und sind für die Präsentation des Ansatzes dieser Methode zu umfangreich. Nachzulesen ist das detaillierte Verfahren in der Arbeit [11].

3. Alle Tabellen in der R-Gruppe werden entfernt.

4. Da es Fremdschlüssel geben kann, die `NULL`-Werte erlauben, ist es notwendig, ein `IS NOT NULL`-Prädikat dort hinzuzufügen, wo die parents einer RI entfernt wurden und `NULL`-Werte erlaubt sind.

Als visuelles Hilfsmittel verwendet das Verfahren sogenannte Joingraphen $G = (V, E)$. Die Knotenmenge V besteht aus Joinattributen. Eine Kante zwischen zwei Knoten u, v existiert genau dann, wenn u und v Teil eines Joinprädikates sind. Handelt es sich bei dem Join um einen RI-Join, also ein Primär-/ Fremdschlüssel Join, dann sind u, v durch eine gerichtete Kante verbunden, wobei der Pfeil dann vom Fremdschlüsselattribut/Kind zum Primärschlüsselattribut/Eltern geht.

Nimmt man sich diese Joingraphen zur Hilfe gibt es drei auszuführende Schritte. Im ersten Schritt wird der Joingraph erzeugt, nach oben genannten Regeln. Dabei werden RI-Joins noch nicht gerichtet dargestellt. Wir erhalten G_1.

Im zweiten Schritt fügt man alle transitiv-implizierten Joins als nicht RI-Joins hinzu und markiert die echten RI-Joins entsprechend mit Pfeilen. Wir erhalten $G_2 = (V_2, E_2)$

Im dritten Schritt werden unnötige (redundante) Joins entfernt. Ein Knoten $v \in V_2$ ist entfernbar, wenn v Teil eines RI-Joins ist und v nur als Joinkriterium in der Anfrage vorkommt, also weder im `SELECT-`, `WHERE-` oder `GROUP BY`-Statement vorkommt. Wir erhalten G_3, aus dem wir auch ablesen können welche Joins noch notwendig sind.

Folgendes Beispiel soll das eben Erwähnte verdeutlichen.

```
SELECT ps.partkey, avg(ps.supplycost)
FROM   supplier s, partsupp ps, customer c, orders o
WHERE  s.suppkey = ps.suppkey
AND    s.suppkey = c.custkey
AND    c.custkey = o_custkey
AND    o.totalprice >= 100
GROUP BY ps.partkey
```

Wir erzeugen zunächst den ersten Joingraphen G_1, in dem wir jede Joinbedingug als verbundes Paar in den Graphen einfügen:

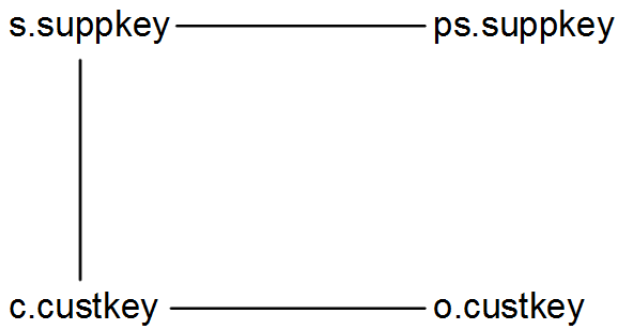


Abbildung 4.17: G_1

Im zweiten Schritt fügen wir nun alle transitiv-implizierten Joins als nicht RI-Joins hinzu, also als ungerichtete Kanten. Wir weisen dann die RI-Joins aus, in dem wir sie als gerichtete Kanten markieren und erhalten damit G_2:

Im letzten Schritt gehen wir alle Knoten sukzessive durch. Da jeder Knoten ein Attribut repräsentiert, prüfen wir ob das repräsentierte Attribut in einem Nicht-Join Prädikat unter `WHERE`, im `GROUP BY`- oder im `SELECT`-Teil vorkommt. Ist dies nicht der Fall, eliminieren wir den Knoten und alle zugehörigen Kanten. In unserem Beispiel kommt weder `s.suppkey` noch `c.custkey` in den genannten Teilen der Anfrage vor. Wir löschen diese Attribute aus G_2 und erhalten G_3 mit nur noch einem, notwendigen Join:

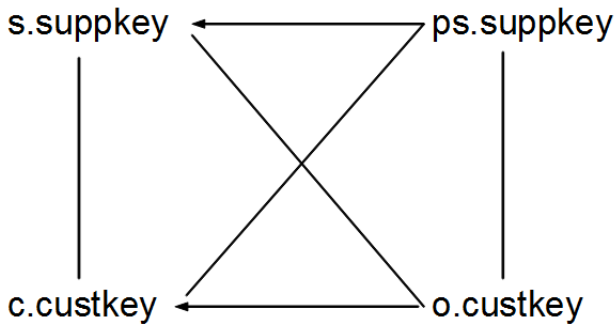


Abbildung 4.18: G_2

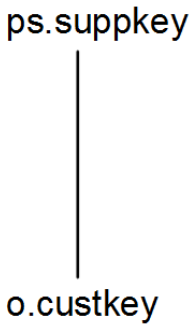


Abbildung 4.19: G_3

Durch die Entfernung von Joinprädikaten kann es nun sein, dass auf einige Tabellen, die unter `FROM` stehen, nicht mehr zugegriffen wird. Ist dies der Fall und sind weitere Bedingungen erfüllt, die detailliert in [11] aufgeführt sind, so müssen diese Tabellen entfernt werden, da sonst ein ungewollter `CROSS JOIN` entsteht, der u.a. zu vielen ungewollten Duplikaten führen kann.

Wir erhalten also folgende SQL-Anfrage:

```
SELECT ps.partkey, avg(ps.supplycost)
FROM   partsupp ps, orders o
WHERE  o.custkey = ps.suppkey
AND    o.totalprice >= 100
GROUP BY ps.partkey
```

Zu Bemerken ist, dass es noch mehr Fälle gibt, die der Algorithmus in [11] behandelt. Der vorgestellte Ansatz entfernt nur eine Art von redundanten Joins.

Redundante Joins zu finden ist keine triviale Aufgabe. Es gibt viele verschiedene Bedingungen die geprüft werden müssen und selbst dann gibt es keinen universellen Ansatz, der sämtliche redundante Joins entdeckt. Aufgrund dieser Komplexität, wird die Joineleminierung nicht in den Prototypen, der im Rahmen dieser Arbeit entsteht, Einzug halten. Für eine zukünftige Erweiterung des Programms um mehr semantic-checks, wäre dieser Ansatz aber gut geeignet.

OUTER JOIN-Eliminierung

Innere Joins sind einfacher zu handhaben als äußere Joins. In der Arbeit [13] wird ein Verfahren vorgestellt, welches `OUTER JOIN` äquivalent in `INNER JOIN` umwandelt. Das Verfahren ist komplex und mehrstufig, kann aber mit allen `OUTER JOIN` umgehen. Der entstandene `INNER JOIN` ist allerdings aufgebläht und nicht mehr lesbar, da er künstlich konstruiert ist. Im Rahmen unserer Arbeit interessiert dieser Fakt nur am Rande, im Zuge einer Erweiterung der semantischen Funktionen unseres Programmes, ist dieser Ansatz aber sicherlich erwähnenswert und interessant.

4.4.2 unnötiges DISTINCT

Eine interessante Frage ist, ob ein `DISTINCT` wirklich notwendig ist. Dies ist natürlich wichtig für den Vergleich zweier SQL-Anfragen. In [9] wurde im Rahmen des SQLLint-Projektes bereits in dem Prototypen ein Checker eingebaut, der prüft ob `DISTINCT` wirklich notwendig ist. Aber auch im Rahmen dieser Arbeit ist es notwendig zu wissen, ob das `DISTINCT` notwendig ist.

Auf den ersten Blick reicht es aus zu prüfen, ob die Musterlösung ein `DISTINCT` enthält. Ist dies der Fall, so muss die Lösung des Lernenden das offensichtlich auch enthalten. Allerdings setzt dieser Denkansatz voraus, dass die eingetragene Musterlösung stets perfekt ist. Um Fehler vorzubeugen ist es besser, alle Anfragen auf unnötige `DISTINCT` zu prüfen. So kann dem Korrektor beim Eintragen der Musterlösung bereits angezeigt werden, dass sein angegebenes `DISTINCT` unnötig ist oder ob ein `DISTINCT` notwendig wäre um Duplikate zu eliminieren. Auch wenn man sich weg bewegt vom Modell der Musterlösung und dem Vergleich mit dem Lernenden, ist dieser Check durchaus wichtig. Im Folgenden stellen wir daher einen Algorithmus vor, der erkennt ob die Lösung Duplikate enthalten kann oder nicht.

Dabei muss angemerkt werden, dass der nachfolgende Algorithmus die hinreichende Bedingung für ein unnötiges `DISTINCT` prüft. Das bedeutet, dass wenn der Algorithmus sagt »ja, `DISTINCT` ist unnötig«, können wir sicher sein, dass es tatsächlich unnötig ist. Sagt der Algorithmus allerdings »nein« bedeutet dies, nicht notwendigerweise, dass das `DISTINCT` wirklich unnötig ist. Gewisse Sachverhalte findet der Algorithmus nicht.

Der Algorithmus ist nicht in unserem Programm implementiert. Wir beschreibe ihn aber im Folgenden so, dass er in einer Überarbeitung ohne Probleme integriert werden könnte. Der Algorithmus müsste dann in jedem Fall nach der Umwandlung des `WHERE`-Teils in die KNF erfolgen, da dies eine Voraussetzung für den Algorithmus ist.

4.4.3 verbale Beschreibung

Es sei $\mathcal{K}$ die Menge aller Attribute, die als Ausgabespalten unter `SELECT` vorkommen. Wir fügen im nächsten Schritt nun alle Attribute A zu $\mathcal{K}$ hinzu, für die $A = c$ bzw. $c = A$ in der `WHERE`-Bedingung auftaucht. Hierbei wird, wie bereits erwähnt, vorausgesetzt, dass die Bedingung eine Konjunktion ist. Weiterhin ist zu bemerken, dass Unterabfragen vom Algorithmus ignoriert werden.

Solange eine der folgenden Aktionen zu einer Veränderung unseres Zustands führt, machen wir Folgendes:

- Wir fügen zur Menge $\mathcal{K}$ Attribute A hinzu, wenn $A = B$ Teil der `WHERE`-Bedingung ist und $B \in \mathcal{K}$ gilt.
- Enthält $\mathcal{K}$ einen Schlüssel einer Tupelvariable, so fügen wir alle Attribute dieser Variable hinzu.

Enthält $\mathcal{K}$ nun von jeder Tupelvariable unter `FROM` einen Schlüssel, so ist das `DISTINCT` sicher überflüssig. Haben wir ein `GROUP BY`-Statement so prüfen wir anstelle dessen, ob alle `GROUP BY`-Spalten in $\mathcal{K}$ enthalten sind.

4.4.4 Algorithmus aus [12]

Es sei unsere SQL-Anfrage der Form:

`SELECT` t_1`, ..., `t_k
`FROM` R_1 X_1`, ..., `R_n X_n
`WHERE` φ

Es sei $X = \{X_1, ..., X_n\}$ die Menge aller Tupelvariablen. Es sei $G = \{G_1, ..., G_m\}$ die Menge aller `GROUP BY` Spalten.

Die einzelnen Attribute t_i haben die Form $t = X.k$. Dabei ist X eine Tupelvariable und k ein Attribut. Wir bezeichnen die Menge aller t_i mit $\mathcal{K} = \{t_1, ..., t_k\}$.

$\mathcal{K}$ = $\mathcal{K} \cup$ {A}, wenn $A = c$ als Konjunktion in WHERE auftaucht
do
  $\mathcal{K}'$ = $\mathcal{K}$
  $\mathcal{K}'$ = $\mathcal{K}'$ $\cup$ A, wenn $A = B \in$ WHERE-Bedingung und $B \in \mathcal{K}$
  $\mathcal{K}'$ = $\mathcal{K}'$ $\cup$ S mit $S = \{b \in X\}$, wenn $t \in \mathcal{K}'$ ein Schluessel ist und t=X.k
while ($\mathcal{K} \neq \mathcal{K}'$)

if not Anfrage hat GROUP BY Statement:

```
  foreach x ∈ X do
    if not (∃k ∈ K' mit k ist Schluessel von x)
      break and return NO
    endif
  done

if Anfrage hat GROUP BY Statement:
  foreach g ∈ G do
    if g ∉ K'
      break and return NO
    endif
  done

return YES
```

4.5 SQL-Anfragen auf externen Datenbanken

Bis jetzt haben wir verschiedene Methoden und Ansätze diskutiert, die sich auf das Erfüllen der hinreichenden Bedingung, einer semantischen Äquivalenz, konzentriert haben. In den vorherigen Abschnitten dieses Kapitels haben wir festgestellt, dass eine, durch Überführung, erhaltene syntaktische Äquivalenz zweier SQL-Anfragen automatisch deren semantische Äquivalenz bedeutet. Wie Eingangs erwähnt, ist das Problem, zwei SQL-Anfragen auf semantische Äquivalenz zu prüfen, im Allgemeinen nicht entscheidbar. Aus diesem Grund kann es passieren, dass unsere, bisher erarbeiteten Methoden, bei einigen, semantisch äquivalenten, Anfragen keinen Erfolg bringen. Die Anfragen sind dann semantisch äquivalent, aber wir können sie nicht durch unsere Methoden aneinander anpassen.

Für diese Art von Anfragen prüfen wir eine notwendige Bedingung der semantischen Äquivalenz zweier Anfragen. Für diesen Schritt benötigen wir echte Daten für das, jeweils gegebene, Datenbankschema. Wir führen sowohl die Musterlösung, als auch die Lösung des Lernenden auf einer Datenbank mit realen Daten aus. Wir vergleichen dann die zurückgelieferten Tupel. Unterscheiden sich diese, dann wissen wir, dass die beiden Anfragen nicht semantisch äquivalent sein können. Dabei würde der Unterschied in beiden Antwortmengen das Gegenbeispiel zur Äquivalenz darstellen.

Sind die Antwortmengen identisch für alle vorhandenen Datenbankzustände, ist dies kein sicheres Zeichen dafür, dass die Anfragen semantisch äquivalent sind, da dies nur das notwendige Kriterium, nicht aber ein hinreichendes Kriterium ist. Haben wir also zwei SQL-Anfragen de-

ren semantische Äquivalenz weder durch Standardisierung nachgewiesen, noch durch Tests auf realen Daten widerlegt werden konnte, so wissen wir nicht, ob diese Anfragen sicher äquivalent sind oder nicht. Wir geben den Lernenden dann das Feedback, dass seine Lösung eventuell richtig oder falsch ist. Auch muss der Dozent, oder andere Berechtigte, über solche Lösungen informiert werden. Die berechtigte Person kann dann entscheiden ob die Lösung eine weitere Musterlösung darstellt, oder nicht korrekt ist.

Im Detail gibt es allerdings einige Feinheiten zu beachten, wenn man die Anfragen auf konkreten Datenbanken mit echten Daten ausführen will. Im Folgenden werden wir diese Feinheiten erläutern und mögliche Probleme disktuieren.

4.5.1 unterschiedliche DBMS

Obwohl es, in gewissen Abständen, zur Herausgabe eines Standards für SQL durch die ISO kommt, unterscheiden sich die existierenden DBMS bei der Handhabung und Implementation von SQL-Features. Die aktuellste Version des SQL-Standards ist zur Zeit: SQL:2008. Die verschiedenen Hersteller variieren diesen Standard allerdings, durch Entfernung oder Hinzufügen von Funktionalitäten. Diese Änderungen sind sowohl in der Data Definiton Langauge (DDL), der Data Manipulation Language (DML) als auch der Data Control Language (DCL) zu finden. Wir gehen im Folgenden auf einige Unterschiede ein. Da wir im Hinblick auf diese Arbeit gerne unabhängig von den Herstellern verschiedner DBMS sein wollen, ist für uns Interessant, welche Schnittmenge von Funktionalitäten alle DBMS gemeinsam haben. Nach heutigem Stand können wir sagen, dass DB2, MSSQL, MySQL, Oracle und Informix mindestens den SQL-92 Standard erfüllen. PostgreSQL hat zusätzlich noch das Konzept der Sichten, erlaubt aber nicht diese zu aktualisieren (update). Um diese Restriktion zu umgehen, führt PostgreSQL ein, nicht vom Standard vorgesehenes, Konzept von Regeln ein, auf das wir nicht näher eingehen wollen. Ist unsere Anfrage also SQL-92 konform, so können wir diese ohne Probleme auf allen gängigen DBMS ausführen.

Betrachten wir nun ein paar Unterschiede im Detail. Von den, oben genannten, DBMS verstehen nur PostgreSQL, MySQL und Oracle das Schlüsselwort `NATURAL JOIN` unter `FROM`. `FULL JOIN`, also die Vereinigung eines `LEFT OUTER` und `RIGHT OUTER` Join verstehen alle DBMS, bis auf MySQL. Alle DBMS verstehen aber das explizite kartesische Produkt, `CROSS JOIN`. Dieser Sachverhalt kommt uns entgegen, da wir versuchen im ersten Teil, der Standardisierung, alle Joins als kartesisches Produkt mit Auswahl zu formulieren, falls das möglich ist. Können wir bestimmte Joins nicht umschreiben, kann es zu Laufzeitfehlern kommen, welche später in diesem Kapitel behandelt werden.

Unterschiede in der DDL behandeln wir nicht, da wir uns im, Rahmen dieser Arbeit, nur mit dem `SELECT`-Statement beschäftigen. Ein wesentlicher Unterschied hierbei besteht in der Sortie-

rung von Werten. Der Standard trifft keine Aussage darüber, wie NULL-Werte im Verlgeich zu nicht-NULL-Werten sortiert werden sollen. Der Standard beschreibt lediglich die Schlüsselwörter NULLS FIRST sowie NULLS LAST, um das Verhalten dementsprechend zu erzwingen. In der Implementation haben die Hersteller sich allerdings entscheiden müssen, was im default-Fall zu tun ist. In den Systemen PostgreSQL, DB2 und Oracle werden NULL-Werte mit höherer Priorität als nicht-NULL-Werte behandelt. Dementsprechend behandeln MSSQL, Informix und MySQL die NULL-Werte niedriger, als die nicht-NULL-Werte. Bei Oracle tritt zusätzlich die Besonderheit auf, dass ein leerer String die gleiche Priorität hat, wie ein NULL-Wert.

Weitere Unterschiede zwischen den DBMS treten auf in Bezug auf das Schlüsselwort LIMIT n, welches bewirkt, dass nur die ersten *n* Datensätze ausgegeben werden. Lediglich MySQL und PostgreSQL kennen dieses Schlüsselwort. Bei allen anderen DBMS muss diese Anforderung durch Verwendung der *row-number* umgesetzt werden.

Weitere Unterschiede haben die DBMS bei Verwendung vom BOOLEAN-Datentyp, dem UNIQUE-Constraint, der Konkatenation von Strings, dem Schlüsselwort LOCALTIMESTAMP, der Funktion SUBSTRING und dem Datentyp TIMESTAMP. Es würde zu weit führen alle Unterschiede genau zu betrachten. Nachzulesen sind die genauen Unterschiede unter [14]

Wichtig für unsere Arbeit ist, dass wir keine Probleme zu erwarten haben, wenn eine Anfrage SQL-92 konform ist. Sollte dies nicht der Fall sein werden wir, mit großer Wahrscheinlichkeit, einen Laufzeitfehler oder einen Fehler beim Parsen der Anfrage erhalten.

4.5.2 Visualisierung von Ergebnistupeln

Wir wollen zwei SQL-Anfragen auf einer konkreten Datenbank ausführen. Nun müssen wir die entstandenen Ergebnismengen miteinander vergleichen. Dabei wollen wir in diesem Abschnitt diskutieren wie dies am sinnvollsten Geschehen soll.

Der erste Ansatz ist es, beide Ergebnismengen iterativ zu vergleichen. Wir gehen also beide Mengen Schritt für Schritt durch. Treffen wir dabei auf unterschiedliche Elemente, so geben wir an in welchem Tupel sich beide Mengen unterscheiden und beenden den Vergleich. Wir müssen sicher gehen, dass die Reihenfolge der Spalten und Zielen dabei keine Probleme verursacht.

Interessant sind dabei sowohl die Reihenfolge der Spalten, als auch die Sortierung der Tupel eine Rolle. Wir haben die Sortierung der Spalten bereits im Abschnitt SELECT-Teil behandelt. Spielt die Reihenfolge keine Rolle, so sind die Items im SELECT-Teil bereits lexikographisch sortiert. Andernfalls ist die Reihenfolge fix und wir müssen sie so nehmen, wie wir sie erhalten.

Für eine eindeutige Reihenfolge der Tupel haben wir allerdings bisher nicht gesorgt. Wir müssen hier mehrere Fälle betrachten.

Enthält die Musterlösung und die Lösung des Lernenden kein `ORDER BY`, müssen wir uns um die Reihenfolge nicht weiter kümmern, da jedes DBMS deterministisch eine Reihenfolge auswählen wird. Diese Reihenfolge ist dann insbesondere für beide Anfragen identisch, da deterministisch.

Der zweite Fall zeichnet sich dadurch aus, dass die Musterlösung kein `ORDER BY` enthält, die Lösung des Lernenden enthält aber ein solches. Wir können dann davon ausgehen, dass die Reihenfolge nicht von Bedeutung ist, denn sonst würde die Musterlösung ein `ORDER BY` enthalten. Wir könnten nun entweder das `ORDER BY` aus der Lösung des Lernenden entfernen, oder das `ORDER BY` des Lernenden in die Musterlösung einfügen. Im letzten Fall kann es Probleme geben, wenn die Lösung des Lernenden Spalten im `ORDER BY` enthält, die in der Musterlösung nicht vorkommen. Wir entscheiden uns daher zur Entfernung des `ORDER BY` aus der Lösung des Lernenden, bevor wir die Anfragen auf der realen Datenbank ausführen. Der Lernende erhält natürlich dennoch einen Hinweis, dass er ein `ORDER BY` verwendet, die Musterlösung aber nicht. Solche Informationen werden ja bereits im Preprocessing gesammelt und gehen dadurch nicht verloren.

Der dritte Fall wäre dann genau umgekehrt. Hier hat also die Musterlösung ein `ORDER BY`-Statement, die Lösung des Lernenden aber nicht. Hier ist offensichtlich eine Reihenfolge gewünscht. Wir werden daher keine Anpassung in einem solchen Fall durchführen. Unterscheiden sich die Musterlösung und die Lösung des Lernenden nur durch das `ORDER BY`-Statement, erhält der Lernende ohnehin eine Meldung, dass der Rest der Anfrage übereinstimmt, so wie im Bereich Preprocessing erläutert.

Der letzte Fall ist für uns nicht relevant. In beiden Anfragen befindet sich dann ein `ORDER BY` und daher können wir auch keine Anpassungen vornehmen.

Nachdem wir dann die zwei Anfragen auf der Datenbank ausgeführt haben erhalten wir zwei Ergebnistupel. Stellt sich durch einen schrittweise Vergleich heraus, dass beide Ergebnismengen identisch sind, müssen wir dem Studenten mitteilen, dass es nicht entscheidbar ist, ob seine Lösung korrekt ist oder nicht. In einem solchen Fall muss der Dozent entweder eine Mail erhalten, um die Lösung manuell zu prüfen, oder dem Studenten wird angezeigt, dass er diese Lösung doch in der nächsten Übung vorstellen soll, um deren Korrektheit zu prüfen.

Sind die Ergebnismengen aber nicht gleich, so zeigen wir beide Tupel in Tabellenform nebeneinander an. Der Lernenden kann dann schnell und übersichtlich erkennen, welche Tupel er mit seiner Anfrage zurückliefert und welche Tupel ihm fehlen bzw. zu viel sind.

4.5.3 Behandeln von Fehlern

Bei der Ausführung von SQL-Anfragen auf realen Datenbanken können verschieden Fehler auftreten. Wir möchten nun kurz untersuchen welche Arten von Fehlern wir erwarten können und wie

wir diese behandeln wollen. Der Begriff »Fehler« ist in diesem Sinne auch etwas weiträumiger zu verstehen, wie wir gleich merken werden.

Einer der typischsten Fehler ist der Syntaxfehler. Der Lernende hat also eine Lösung angegeben, die nicht in das Syntaxschema einer SQL-Anfrage passt. Die Hauptursachen für solche Fehler sind falsche Anordnung von Schlüsselwörtern oder gar »Rechtschreibfehler«. Jedoch kann es auch sein, dass der Lernende Syntax benutzt hat, welche nur ein bestimmtes SQL-System versteht, z. B. PostgreSQL. Wird die Anfrage aber auf einer Oracle Datenbank ausgeführt, versteht das DBMS diese Anfrage eventuell nicht und es kommt zu einem Syntaxfehler, der auf einem anderen DBMS keiner wäre. Wir könnten den Fehler verhindern, in dem wir in einem vorherigen Schritt prüfen, ob die Anfrage SQL-92 konform ist. Eine andere Lösung ist es, zu jeder Anfrage zu speichern, für welches konkretes DBMS sie gedacht ist. Dann könnte man sich sicher sein, dass der Syntaxfehler auch einen Fehler für alle SQL-Systeme darstellt. Wenn man keine gesonderte Behandlung in Betracht zieht, so kann man den Fehler, wie üblich, dem Nutzer anzeigen mit dem Vermerk, dass der Syntaxfehler eventuell zu DBMS spezifisch ist. In unserem Prototypen reichen wir derartige Syntaxfehler an den Nutzer weiter.

Ein weiterer Fehlertyp ist der semantische Fehler. Wie bereits, in der Einführung dieser Arbeit, erwähnt, ist dies ein Fehlertyp, der schwer zu entdecken ist. Typische Vertreter sind inkonsistente Bedingungen, also `WHERE`-Bedingungen, die entweder immer die leere Menge liefern, oder die immer die gesamten Tabellen als Ergebnistupel liefern, da die `WHERE`-Bedingung allgemeingültig ist (Tautologie). Die eben genannten Fälle erkennt unser Programm, da es mit der Methode `eval` die Ausdrücke so weit möglich auswertet. Andere semantische Fehler, wie unnötige JOINs und unnötiges DISTINCT wurden bereits im Kapitel 4 behandelt. Unser Programm wird davon allerdings nichts umsetzen, da es einen größeren Aufwand darstellt diese Methoden zu implementieren. Es gibt weitere semantische Fehler, die unser Programm nicht berücksichtigt. Hier sei verwiesen auf die Arbeit [9].

In Hinblick auf syntaktische und semantische Fehler sollten wir uns fragen, welche Anfrage wir, zur Ausführung auf der realen Datenbank, einsetzen wollen. Zur Wahl stehen dabei die Anfrage, die der Lernende eingibt und die Anfrage, die unser Programm im ersten Schritt durch Standardisierung erzeugt.

Da wir, bei der Ausführung der Anfragen auf realen Datenbanken, darauf bedacht sind, so wenig Fehler wie möglich zu erzeugen, ist es am sinnvollsten, die standardisierte Anfrage zu verwenden. Der Lernende bekommt in jedem Fall Hinweise, wenn bei der Standardisierung semantische oder syntaktische Besonderheiten gefunden wurden. hat der Student z. B. eine Tautologie als `WHERE`-Bedingung aufgeschrieben, so kann das unser Programm erkennen. Der Lernende erhält den Hinweis, dass seine Bedingung überflüssig ist und auf der realen Datenbank lassen wir dann, die schon verkürzte Fassung laufen.

Ein letzter Fehlertyp bilden die Laufzeitfehler. Da wir solche Fehler nur schwer oder gar nicht im Vorfeld erkennen können, sind wir gezwungen die Fehlermeldung an den Lernenden weiterzugeben. Diese Fehlermeldung werden vom konkreten DBMS erzeugt und können mitunter unverständlich sein. Jedes DBMS hat dabei auch eine eigene Formulierung für die gleichen Laufzeitfehler. Es bietet sich daher an, eine Art Wörterbuch für Fehlermeldungen zu verwenden. Tritt ein bestimmter Fehler in einem DBMS auf, so wird dieser »übersetzt« in eine, von uns standardisierte, Fehlermeldung. So können wir sicher sein, dass der Student beim gleichen Fehler, die gleiche Meldung erhält, unabhängig vom DBMS.

5 Praktische Umsetzung

Im folgendem Abschnitt beschreiben wir die Struktur des Programms und gehen, in übersichtlicher Form, auf die einzelnen Klassen und Funktionen des Programms ein. Eine noch detaillierte Fassung der Dokumentation findet sich im Quelltext als Javadocs-Kommentare.

5.1 Datenbankstruktur

Wir benutzen eine Datenbank um Aufgabenstellungen, Nutzer, Lösungsversuche und vieles mehr abzuspeichern. Im folgenden werden die einzelnen Tabellen erläutert. Wir verwenden dazu bewusst eine abstraktere Form der Beschreibung. Begründet ist dies, durch die Vielfalt der verschiedenen DBMS und ihrer verschiedenen Auslegung der Data Defintion Language. Im Prototypen verwenden wir das DBMS MySQL, man könnte aber alle DBMS verwenden, die einen JDBC Connector haben.

attempts (id, userid → user(id), taskid → tasks(id), timeat ,sqlstatement ,correct)

Die Tabelle `attempts` speichert je einen Lösungsversuch eines Benutzers. Dabei wird die SQL-Anfrage (`sqlstatement`) eines Benutzers (mit der ID `userid`) mit der Uhrzeit (`timeat`) abgespeichert. Weiterhin wird vermerkt, welche Aufgabe (`taskid`) der Student lösen wollte und, ob dieser Korrekt war.

dbschema (id, name, schema)

In der Tabelle `dbschema` speichern wir zu jeder Aufgabe `taskid`, wie die Struktur der Tabelle dafür aussieht. Wir speichern dazu, möglichst SQL-92 konforme, `CREATE TABLE`-Anweisungen im Attribut `schema`, mit Semikolon getrennt. Wir ordnen dem Schema außerdem einen Namen zu, damit es später einfach wiederzufinden ist.

external_database (id, uri, dbname, username, password, typ)

Die Tabelle `external_database` speichert die Zugriffsdaten (`username,password`) einer externen Datenbank, die verwendet wird, um die notwendigen Bedingung einer Äquivalenz zu prüfen. Weiterhin speichern wir den Zugriffspfad auf diese Datenbank (`uri`) und den Datenbanknamen (`dbname`). Schließlich speichern wir noch den Typ. Im Moment werden mysql, postgresql

und oracle als Typ unterstützt. Wir gehen später darauf ein, wie wir die Funktionalität auf noch mehr Datenbanken ausweiten können.

samplesolutions (id, taskid→tasks(id), sqlstatement)

Die Musterlösung für eine Aufgabe, wird in der Tabelle `samplesolutions` gespeichert. Wir erfassen dabei die Aufgabennummer (`taskid`), sowie die Musterlösung als SQL-Anfrage (`sqlstatement`). Wir verwenden einen künstlichen Schlüssel `id`, da es unter Umständen mehrere Musterlösungen zu einer Aufgabe geben kann.

tasks (id, description, respectColumnOrder, schemaid, createdAt^0)

Die Aufgaben werden in der Tabelle `tasks` gespeichert. Wir versehen jede Aufgabe mit einer `id` und einer Beschreibung, in Form eines Sachtextes (`description`). Wir speichern, ob die Reihenfolge der Spalten (im `SELECT`-Teil) eingehalten werden muss und, erfassen außerdem, welches Datenbankschema zu dieser Aufgabe zugeordnet werden soll. Optional können wir noch vermerken, wann die Aufgabe eingetragen wurde.

tasks_db (taskid → tasks(id), dbid → external_database(id))

Es kann unter Umständen möglich sein, pro Aufgabe mehrere externe Datenbanken anzugeben. Daher speichern wir die Zuordnung einer Datenbank zu einer Aufgabe in der Tabelle `tasks_db`. Hierzu verwenden wir den Fremdschlüssel zur Aufgabentabelle (`taskid`) und den zur externen Datenbank (`dbid`).

users (id, name, password, isDozent)

Einzelne Benutzer werden in der Tabelle `users` erfasst. Neben den üblichen Kontoinformationen, wie `name, password` speichern wir auch, ob der Nutzer ein Dozent ist. Diese Information ist wichtig, um zu entscheiden, ob ein Nutzer auch Aufgaben einpflegen darf.

5.2 Programmstruktur

In diesem Abschnitt werden wir die wichtigsten Klassen und Funktionen auflisten und erläutern. Eine vollständige Dokumentation aller Klassen, ist im Quellcode als Javadocs-Kommentare vorhanden.

Das Programm beinhaltet mehrere Prozesse, die sich aus der Abfolge mehrere Funktionen zusammensetzen. Im Folgenden werden alle wesentlichen Prozesse und deren Zusammensetzung erläutert. Wir gehen dabei auch auf Möglichkeiten zur Anpassung des Programms ein.

Prozess: Standardisieren einer SQL-Anfrage

Wir beschäftigen uns zu erst mit einer der Hauptfunktionen: das Standardisieren einer SQL-Anfrage. Die Klasse `QueryHandler` kümmert sich um das Standardisieren einer SQL-Anfrage. Für jede SQL-Anfrage wird ein separates Objekt dieser Klasse erzeugt. Anschließend importieren wir das, zu Grunde liegende, Datenbankschema in das Objekt. Dies geschieht mit der Methode:
`boolean createTable(String createTableStatement)`
Die Methode erwartet pro Aufruf ein, wenn möglich SQL-92 konformes, `CREATE TABLE`-Statement. Nachdem wir das Datenbankschema eingelesen haben, importieren wir noch die zu standardisierende Anfrage. Dies geschieht mit der Methode:
`setOriginalStatement(String s)`
Das eingegebene Statement wird dann mit Hilfe des ZQL-Parsers eingelesen. Etwaige Fehler beim Parsen werden per Exception an den Nutzer weitergereicht. Mehr dazu im Abschnitt »Webinterface«. Wir kopieren das Statement außerdem, so dass wir intern zwei ZQuery-Objekte speichern, zum einen die Originalanfrage im Objekt **original**, die niemals verändert wird, zum Anderen eine Arbeitskopie im Objekt **workingCopy**, welche wir dann standardisieren. Nun ist das Datenbankschema eingelesen und die, zu verarbeitende, Anfrage liegt vor. Wir können den Standardisierungsprozess nun starten, mit der Methode:
`ZQuery[] equalize(boolean respect)`
Als Eingabeparameter erhält sie die Information, ob die Reihenfolge im `SELECT` respektiert werden soll, oder nicht. Im Abschnitt »externe Datenbanken« haben wir bereits darüber diskutiert, dass diese Funktion durchaus interessant sein kann. Weiterhin beobachten wir, dass die Methode nicht eine Anfrage, sondern ein Array von Anfragen zurückliefert. Wir erinnern uns an die Problematik der Selbstverbunde. Dort werden mehrere Permutationen der Tabellen unter `FROM` auf die Anfrage angewandt. Es entstehen dann mehrere Anfragen, die alle semantisch äquivalent sind, aber Aufgrund ihrer unterschiedlichen Struktur alle, mit der Lösung des Lernenden abgeglichen werden müssen.

Die Funktion `equalize` startet die Funktion:
`handleQUERY(ZQuery q)`
Zu beachten ist der Parameter. Da die Abfrage während des Standardisierens verändert wird, ist die ursprüngliche Anfrage des Studenten unwiederbringlich verändert. Damit wir dennoch Zugriff auf die Originalanfrage haben , entscheiden wir nun welche Anfrage zum Standardisieren übergeben wird. Wir übergeben hier das Objekt *workingCopy*.

`handleQUERY` speichert zunächst alle Metainformationen der Anfrage in einem Objekt der Klasse `MetaQueryInfo`. Wie dies genau geschieht behandeln wir später, in diesem Kapitel. Anschließend verarbeitet `handleQUERY` mit Hilfe der Methode `handleFROM` den `FROM`-Teil unserer Anfrage.

Die Verarbeitung hängt vom SELECT-Teil ab. Wie bereits in Kapitel 4 besprochen, werden, abhängig vom SELECT-Teil, die Tabellen zunächst sortiert. Anschließend erhalten alle Tabellen künstliche Tupelvariablen, der fortlaufend sind. In der Standardeinstellung des Programmes ist dies `a1, a2, a3, ...` Die verwendeten Tupelvariablen werden nun in einer *HashMap* als Attribut **Zuordnung** in der Klasse QueryHandler gespeichert.

Anschließend bearbeiten wir den SELECT-Teil unserer Anfrage mit der Funktion:
`Vector<ZSelectItem> handleSELECTClause(Vector<ZSelectItem> sel)`
Im wesentlichen ändern wir hier vorhandene Tupelvariablen in unsere automatisch erzeugten. Finden wir dabei Attribute, die keine Tupelvariablen besitzen und es mehrere Tabellen unter FROM gibt, die dieses konkrete Attribut beinhalten, lösen wir eine Fehlermeldung aus, wegen Uneindeutigkeit.

Jetzt erstellen wir Permutationen der Tabellen in der FROM-Liste. Wir gehen dabei vor, wie im Kapitel 4 beschrieben. Sind in unter FROM keine zwei gleichen Tabelle, dann enthält die Menge der Permutationen nur die ursprüngliche Anordnung. Die nun folgenden Schritte werden für jede permutierte FROM-Liste durchgeführt.

`equalize` fährt fort mit der Methode: `ZExp handleWHERE(ZExp exp)`.
Innerhalb dieser Methode erledigen wir alle Arbeiten, die im Abschnitt »WHERE-Teil« in Kapitel 4 erläutert wurden. Zunächst führen wir die künstlichen Tupelvariablen ein mit der Funktion `makeAlias`. Dies geschieht in ähnlicher Weise, wie im SELECT-Teil. `makeAlias` sucht auch nach Unterabfragen. Findet es solche, werden diese ersetzt durch ihre standardisierte Form, indem der Standardisierungsprozess für diese rekursiv aufgerufen wird. Danach transformieren wir, mit der Funktion `transformToKNF`, die aktuelle Form des WHERE-Ausdrucks in eine konjunktive Normalform. Dabei bedient sich `transformToKNF` der Funktionen:
`operatorCompression, pushDownNegate` und `distribute`. Die Vorgehensweise entspricht der besprochenen im theoretischen Bereich dieser Arbeit. Wir ersetzen danach Unterabfragen durch EXISTS-Unterabfragen, wenn möglich. Anschließend werden implizite Formeln hinzugefügt sowie arithmetische und logische Ausdrücke ausgewertet, soweit dies möglich ist. Gerade durch die letzten, beiden Teilschritte, kann es passieren, dass die konjunktive Normalform wieder zerstört worden ist. Daher wenden wir an dieser Stelle erneut `transformToKNF` an. Als letzten Schritt sortieren wir den Ausdruck so, wie ausführlich im Bereich 4.3.7 beschrieben. Die Sortierung beinhaltet auch die Duplikateliminierung bei doppelten AND- oder OR-Operanden.

`handleQUERY` wendet sich jetzt dem GROUP BY-Teil zu. Dieser wird in zwei Teilen abgearbeitet. ZQL trennt GROUP BY- vom HAVING-Teil. Im GROUP BY-Teil sortieren wir die Elemente lexikographisch, wie schon im SELECT-Teil geschehen. Der HAVING-Teil wird mit der Funktion `handleWHERE` behandelt, da sich beide Ausdrücke strukturell gleichen.

Der entstandene SQL-Ausdruck, in Form eines `ZQuery`-Objektes, wird von `handleQUERY` direkt zu `equalize` zurückgeliefert. Der Standardisierungsprozess ist damit beendet. Abbildung 5.1 soll die Reihenfolge der abgearbeiteten Funktionen noch einmal verdeutlichen.

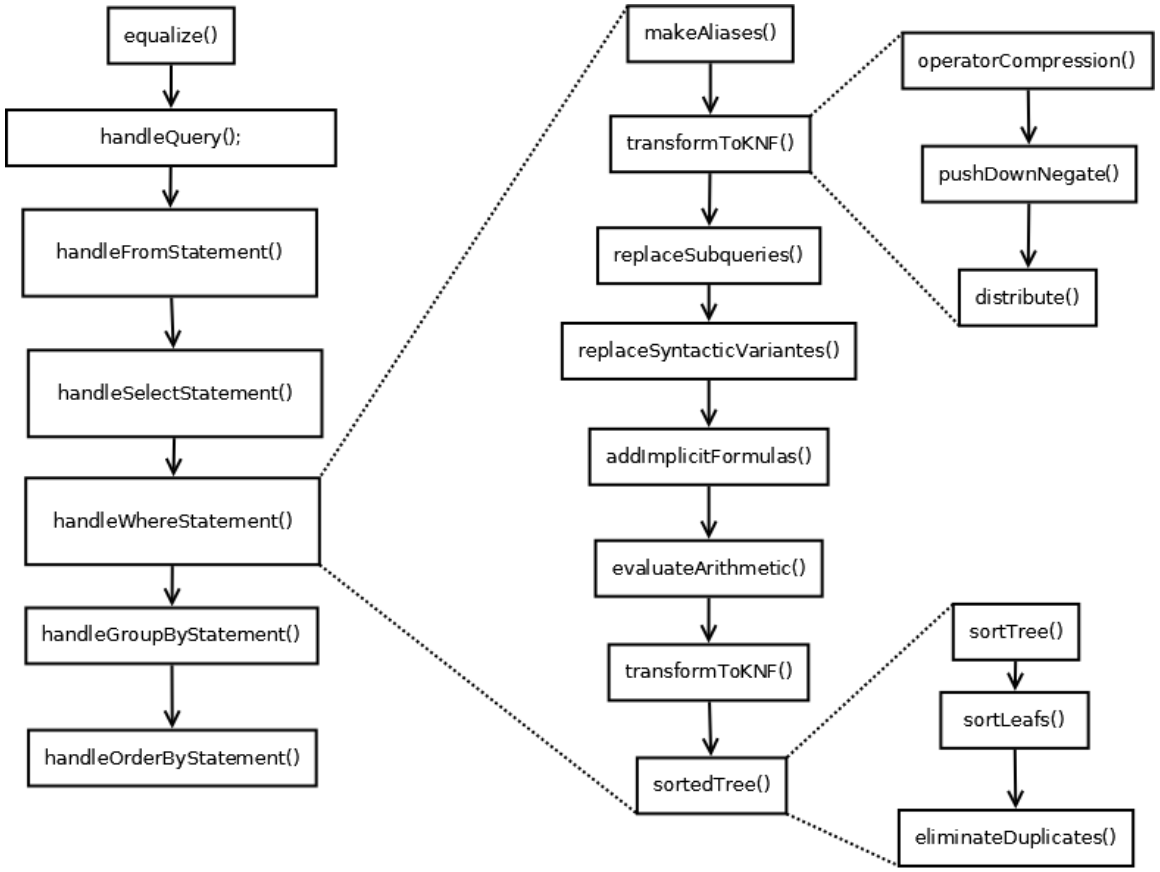


Abbildung 5.1: Einfacher Programmablaufplan

Prozess: Verwenden von externen Datenbanken

Nachdem wir den Prozess der Standardisierung ausführlich betrachtet haben, kommen wir nun zum zweiten, großen Prozess. Es handelt sich hierbei um das Ausführen der zwei Anfragen auf externen Datenbanken und dem Vergleich der Ergebnismengen. Hierfür verwenden wir hauptsächlich die Klasse `QueryUtils`, welche eine Sammlung von nützlichen Funktionen zum Behandeln der SQL-Anfragen beinhaltet. Doch zunächst beginnen wir mit dem Ausführen einer SQL-Anfrage auf einer externen Datenbank. Grundsätzlich können zu jeder Aufgabe beliebig viele externe Datenbanken angegeben werden. Die Lösungen würden dann auf allen diesen Datenbanken ausgeführt werden. Dies ist nützlich, um verschiedene Datenbankzustände zu testen. Weiterhin können so Eigenheiten unterschiedlicher DBMS aufgezeigt werden.

Alle, uns zur Verfügung stehende, externe Datenbanken sind wiederum in unserer internen Datenbank abgespeichert, in Form von ID, Hostname, Port, Datenbankname, Benutzername, Passwort

und Typ. Unterstützte Typen sind zur Zeit MySQL, PostgreSQL und Oracle. Allerdings kann die Funktionalität jederzeit erweitert werden. Notwendig ist dazu lediglich das Einpflegen des jeweiligen JDBC-Connectors. Die statische Klasse `externalDB` regelt nun das Ausführen einer eingegebene SQL-Anfrage `q` auf der externen Datenbank mit der ID `dbid`.
`static ResultSet executeQueryOn(String q, int dbid)`
Die Methode `executeQueryOn` liefert ein `ResultSet` zurück. Dabei extrahiert sie zunächst die Zugangsdaten für die externe Datenbank mit Hilfe der `dbid`. Im zweiten Schritt wird der korrekte JDBC-Connector geladen und eine Datenbankverbindung geöffnet. Nach dem Ausführen der Anfrage `q`, wird die Ergebnismenge in Form eines `ResultSet` zurückgegeben. Möchte man weitere DBMS unterstützen, so müssen in dieser Methode die JDBC-Connector eingepflegt werden.

Wie bereits im Kapitel 4 besprochen, entfernen wir ,vor dem Ausführen der Anfragen auf externen Datenbanken, das `ORDER BY`-Statement von der Lösung des Lernenden, wenn die Musterlösung kein solches Statement enthält.

Nach dem Ausführen der Anfragen erhalten wir zwei Objekte vom Typ `ResultSet`, nämlich `r1` und `r2`. Um zu prüfen, ob beide Ergebnismengen identisch sind, bedienen wir uns eine Funktion aus der Klasse `QueryUtils`.
`boolean isIdenticalResultSets(ResultSet r1, ResultSet r2)`
Die Methode geht recht intuitiv vor. Wir gehen iterativ alle Tupel durch und für jedes Tupel prüfen wir, Schritt für Schritt, ob alle Spalteneinträge identisch sind. Sind Spaltenlängen, Zeilenlängen oder Inhalte der Spalten nicht identisch, so sind beide Ergebnismengen ebenfalls nicht identisch. Die Reihenfolge der Tupel ist vorgegeben durch die `ORDER BY`-Statements beider Anfragen. Die Reihenfolge der Spalten ist vorgegeben durch das `SELECT`-Statement. Bereits im Schritt der Standardisierung wurden die `SELECT`-Einträge sortiert, wenn ausdrücklich in der Aufgabenstellung notiert war, dass die Spaltenreihenfolge keine Rolle spielt. Aus diesen Gründen ist es legitim, die Ergebnismengen in dieser Art zu vergleichen.

Prozess: Auswerten von Metainformationen

Im Prozess 'Standardisierung' haben wir bereits erfahren, dass vor dem Anpassen der SQL-Anfrage, Metainformationen über diese abgespeichert werden. Dazu wird ein Objekt der Klasse `MetaQueryInfo` erstellt. Dem Konstruktor wird die zu analysierende SQL-Anfrage in Form eines `ZQuery` übergeben. Mit Hilfe verschiedener Hilfsfunktionen werden dann Informationen über die SQL-Anfrage gesammelt und gespeichert. Beispiele für diese Informationen sind: Anzahl der Verbunde, Größe der `SELECT`-Liste, Anzahl der Tabellen unter `FROM`, Anzahl von Unterabfragen, e.t.c.

Die Informationen sind einzeln nicht interessant. Nur der Vergleich der Metainformationen von der Musterlösung mit der Lösung des Lernenden, geben Aufschluss über etwaige Fehler. Diesen

Vergleich übernimmt eine statische Funktion aus der Klasse `QueryUtils`.
`static String compareMetaInfos(MetaQueryInfo m1, MetaQueryInfo m2)`
Um die Art des Vergleiches zu verstehen, muss darauf hingewiesen werden, dass alle gespeicherten Attribute in einem Objekt vom Typ `MetaQueryInfo` entweder vom Typ **int** oder vom Typ **boolean** sind. Ganzzahlige Attribute verwenden wir um die Anzahl von etwas festzustellen, also z. B. die Anzahl von Formeln im `WHERE`-Teil. Attribute vom Typ **boolean** benutzen wir, um festzustellen, ob einzelne Teile einer Anfrage existieren oder nicht.

Der Vergleich erfolgt nun automatisch. Die Methode `compareMetaInfos`, geht iterativ alle Attribute der Objekte `m1` und `m2` durch und erzeugt dabei dynamisch eine Hinweismeldung in Form eines **String**. Ist das jeweilige Attribut ganzzahlig, so wird geprüft ob das von `m1` größer, kleiner oder gleich im Vergleich zu dem von `m2` ist. Ist es nicht gleich, dann wird eine Meldung erzeugt. Bei der erzeugten Meldung erscheint allerdings nicht einfach der Attributsname, da dies zu kryptisch und unverständlich gegenüber einem Nutzer wäre. Stattdessen kennt die Klasse `QueryUtils` ein Wörterbuch, in Form einer `HashMap`. In dieser ist für jedes Attribut ein Ausgabe-String abgespeichert, der für Nutzer verständlicher ist. Ähnlich wird dann auch mit den Attributen verfahren, die vom Typ **boolean** sind. Das angesprochene Wörterbuch kann später auch dazu verwendet werden, um Ausgaben in mehreren Sprachen umzusetzen. Ein Auszug aus dem Programmcode ist in Abbildung 5.2 zu sehen.

```
String res = "";
  for (Field f : m1.getClass().getDeclaredFields()) {
    if (f.getType().toString().equals("int")) {
      String word = null;
      if (f.getInt(m1) < f.getInt(m2)) {
        word = messages.get("less");
      } else if (f.getInt(m1) > f.getInt(m2)) {
        word = messages.get("more");
      }
      if (word != null) {
        String x = "There␣are␣" + word + "␣"
          + messages.get(f.getName()) + "␣in␣the␣"
          + messages.get("samplesolution") + "␣(" + f.getInt(m1)
          + ")␣than␣in␣yours␣(" + f.getInt(m2) + ").";
        res += x + "<br>";
      }
```

Abbildung 5.2: Quellcode zum Vergleich von ganzzahligen Metainformationen

5.3 Webinterface

Die JSP dienen dazu die erstellten Java-Klassen zu verbinden und zu steuern. Sie stellt außerdem das Benutzerinterface (UI), mit Hilfe von Webseiten, zur Verfügung. Dieser Teil ist nur prototypisch umgesetzt und durchaus beliebig erweiterbar. Die Grundfunktionalität ist allerdings implementiert. Nachdem sich der Nutzer über die **login.jsp** eingeloggt hat, sieht er auf der Hauptseite eine kleine Übersicht seiner bisherigen Lösungsversuche. Über das Navigationsmenü links, welches mit der **nav.jsp** realisiert wurden ist, kann der Nutzer nun die Aufgabensammlung einsehen. Nachdem er eine Aufgabe gewählt hat, wird mit der **task.jsp** die Aufgabenstellung dargestellt.

Zunächst sieht der Nutzer die Aufgabenstellung in Textform. Anschließend kann er erkennen, welche Tabellen für die Lösung der Aufgabe zur Verfügung stehen. Darunter befindet sich ein Eingabefeld (1), in dem der Nutzer nun seine Lösung in Form einer SQL-Anfrage stellen kann.

Nachdem der Nutzer seine Lösung eingetragen und durch betätigen des *submit*-Knopfes eingereicht hat, werden die Daten per POST-Methode wieder an die task.jsp gesendet. Die Seite wird wieder aufgebaut und nun um mehrere Ausgaben erweitert. Unter dem Textfeld zur Eingabe der Lösung findet der Nutzer nun seine geparste Eingabe (2). Weiterhin stößt die **task.jsp** den `equalize` Prozess an. Unter der geparsten Eingabe sieht der Nutzer jetzt auch die standardisierte Eingabe (3). Ein Beispiel ist in Abbildung 5.3 dargestellt.

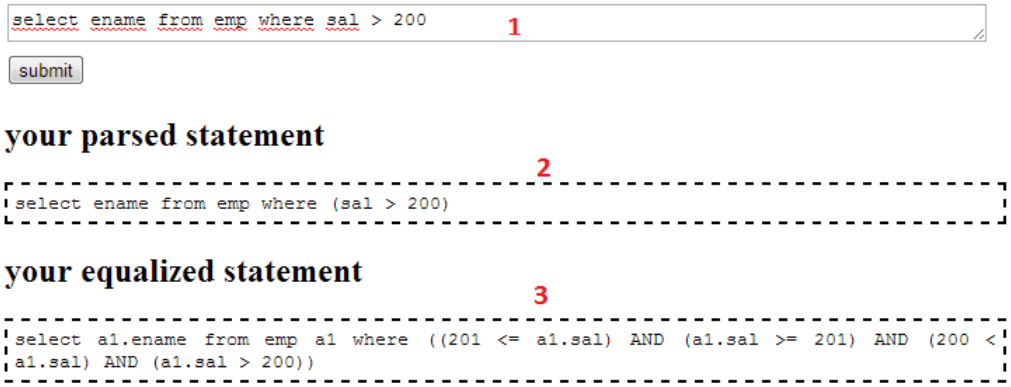


Abbildung 5.3: Eingabe (1), geparste Eingabe (2) und standardisierte Eingabe (3)

Nun erstellt die **task.jsp** drei Blöcke. Im ersten Block (4) sehen wir, ob die standardisierte Musterlösung mit der Lösung des Nutzers gematcht werden konnte. Dazu vergleicht die **task.jsp** die standardisierte Lösung des Nutzers mit allen standardisierten Lösungen der Musterlösung. Passt eine der standardisierten Musterlösung, so wird dies dem Nutzer mitgeteilt.

Im zweiten Block (5) lässt die **task.jsp** beide Anfragen auf der externen Datenbank ausführen, falls Schritt 1 fehlgeschlagen ist. Wir brauchen hier nicht jede der Musterlösungen ausführen, weil alle das gleiche zurückgeben. Danach werden die Ergebnisse verglichen. Gibt es keine Übereinstimmungen der Ergebnisse, so werden beide Ergebnismengen als Tabellen visualisiert, damit

der Nutzer schnell prüfen kann, welche Tupel fehlen oder überflüssig sind. Sind beide Ergebnismengen allerdings identisch, wird dem Nutzer mitgeteilt, das nicht entschieden werden konnte, ob seien Lösung korrekt ist.

Im letzten und dritten Block (6) wird der Vergleich der Metainformationen ausgegeben. Außerdem wird dem Nutzer mitgeteilt, welche Teile seiner Anfrage mit denen der Musterlösung übereinstimmen. Diese Informationen erhält der Nutzer auch wenn bereits Schritt 1 zum Erfolg geführt hat. Mit diesen Informationen sollte es dem Nutzer möglich sein, einen neuen – hoffentlich erfolgreichen – Versuch zu starten.

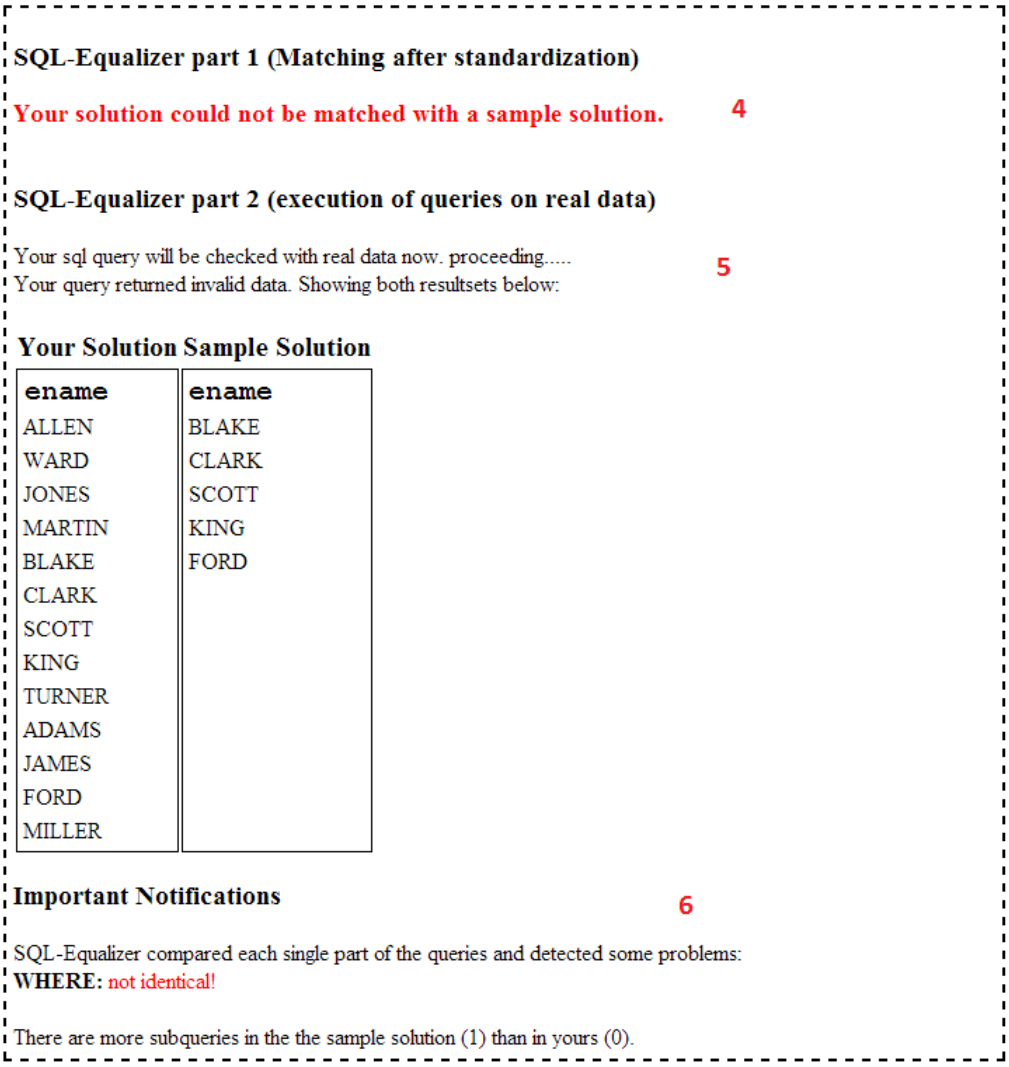


Abbildung 5.4: Matchingstatus (4), Ergebnistupelvergleich (5) und Hinweise (6)

Nach all diesen Schritten speichert die **task.jsp** den Versuch der Lösung in der internen Datenbank ab.

Für Dozenten steht noch die Seite acp.jsp bereit. Hier können neue Datenbankschemata, neue externe Datenbanken, neue Nutzer und neue Aufgabenstellungen eingepflegt werden.

6 Schlussteil

6.1 Zusammenfassung

Die Aufgabenstellung für diese Arbeit war es, Methoden zu entwickeln mit denen man in der Lage ist zwei SQL-Anfragen miteinander zu vergleichen. Dabei sollte herausgefunden werden, ob die beiden Anfragen semantisch äquivalent sind und damit immer die gleichen Ergebnistupel produzieren. Entwickelte Methoden sollten dann in einem Programm realisiert werden, welches potentiell in der Lehre einsetzbar wäre.

Zu diesem Zweck haben wir zunächst untersucht, was wir genau von unserem Programm erwarten. Wir haben festgestellt, dass semantisch-äquivalente Lösungen meist syntaktisch ähnlich sind. Daher haben wir uns überlegt, die SQL-Anfragen aneinander anzugleichen, indem wir legale Umformungen an beiden SQL-Anfragen durchführen. Motiviert von anderen Problemen in der Informatik, in denen eine Vorsortierung ein Problem oft vereinfachen kann, haben wir uns dazu entschlossen zunächst beide SQL-Anfragen nach festen Regeln zu standardisieren. Ziel dabei sollte sein, dass alle semantisch-äquivalenten SQL-Anfragen nach der Standardisierung auch syntaktisch gleich sind. Wenn zwei SQL-Anfragen nach der Standardisierung syntaktisch gleich sind, dann würden sie auch semantisch gleich sein. Dieser Schritt prüft also eine hinreichende Bedingung für semantische Äquivalenz.

Da es Anfragen gibt, die semantisch äquivalent sind, aber sich nicht aneinander angleichen lassen, haben wir einen zweiten Schritt entwickelt. Wir führen beide SQL-Anfragen auf Datenbanken mit echten Daten aus und vergleichen die Ergebnistupel. Sind diese nicht identisch, so verstößt dies gegen eine notwendige Bedingung: “Auf allen Datenbankzuständen müssen zwei semantisch äquivalente SQL-Anfragen stets das gleiche Ergebnis liefern.”

Nach eingehender Analyse der bereits existierenden Lernplattformen zum Thema SQL, konnten wir feststellen, dass dieser Ansatz noch nicht realisiert worden ist. Wir haben Lernplattformen gesehen, die sich mit unserer Idee überschneiden, sie aber nie in dieser Form umsetzen. Nach dem Sichten von existierenden Lernplattformen haben wir den Fokus dieser Arbeit auf das Angleichen von SQL-Anfragen gelegt und verzichteten dafür, zum großen Teil, auf semantische Fehlermeldungen.

Bevor wir mit der theoretischen Entwicklung unserer Methoden begonnen haben, untersuchten wir genau die zur Verfügung stehende Software. Dabei stand insbesondere der verwendete SQL-Parser 'ZQL' im Fokus der Betrachtungen. Nach einer eingehenden Analyse haben wir festgestellt, dass dieser einfach zu Verwenden und zu Erweitern ist, was ihn für unsere Zwecke attraktiv gemacht hat. Bei der Analyse haben wir auch festgestellt, dass der Parser Schwächen hat, die es vorerst verhindern einige unserer Konzepte in einem praktischen Programm umzusetzen.

In Kapitel 4 haben wir dann unsere Ideen aus der Vorbetrachtung in konkrete Methoden entwickelt. Wir haben uns überlegt, welche gleichen Konzepte in SQL in verschiedener Art und Weise formulierbar sind. Für Unterabfragen und Verbunde konnten wir so Methoden entwickeln, die die Vielfältigkeit der Formulierung dieser einschränken. Weiterhin haben wir syntaktische Variationen vereinheitlicht und uns mit der Frage beschäftigt wie man der Operatorenvielfalt in einer SQL-Anfrage begegnen kann. Schlussendlich entwickelten wir eine Sortiermethode, die es uns erlaubt eine feste Ordnung in einem Operatorbaum, der den `WHERE`-Ausdruck einer SQL-Anfrage darstellt, zu etablieren. Aufgrund der Beschränktheit durch den Parser war es uns nicht möglich alle entwickelten Methoden und Betrachtungen in unsere Anwendung umzusetzen. Weiterhin haben wir untersucht was für Probleme auftreten, wenn wir SQL-Anfragen auf externen Datenbanken ausführen, um die notwendige Bedingung der Äquivalenz zu prüfen.

Nachdem die entwickelten Methoden mit Java und JSP implementiert wurden, haben wir die Programmstruktur im Kapitel 5 eingehend erläutert. Die Anwendung wurde im Wesentlichen in drei Hauptprozesse aufgeteilt, die mit Hilfe der JSP gesteuert und dem Nutzer per Web-UI zur Verfügung gestellt werden.

Es wurde erfolgreich eine Anwendung entworfen, die theoretische Betrachtungen und, in dieser Arbeit, entwickelte Methoden umsetzt. Dabei werden, wie gefordert, zwei SQL-Anfragen miteinander verglichen. Im konkreten Fall handelt es sich dabei um eine Musterlösung und eine Lösung eines Lernenden. Es wird geprüft, ob die beiden semantisch Äquivalent sind. Die Anwendung kann als Grundstein für eine Lernplattform dienen, in der es leicht ist neue Aufgaben einzupflegen und der Lernende ein umfassenderes Feedback erhält, als es vom normalen SQL-Parser eines DBMS möglich ist.

6.2 Ausblick

Einige der entwickelten Methoden aus Kapitel 4 konnte nicht umgesetzt werden. Die Ursachen dafür liegen hauptsächlich bei den Schwächen des verwendeten Parsers. Dies sollte für zukünftige Arbeiten aber nur minimale Probleme mit sich bringen, da der Parser quelloffen ist und unter der GNU GPLv3 steht. Damit ist er leicht anpassbar und kann auch offiziell wiederverwendet werden. Die nicht implementierten Methoden sind dennoch ausführlich behandelt, sodass ein nachträgli-

ches Implementieren einfach möglich ist. Eine große Schwäche des Parsers ist es, dass die `FROM`-Klausel nur als Liste von Relationen geparst wird. Dadurch verhindert es der Parser Verbunde und Unterabfragen in `FROM` zu formulieren. Während Unterabfragen unter `FROM` recht selten sind, sind aber Verbunde innerhalb der `FROM`-Klausel durchaus üblich. Würde der Parser um diese Funktionen erweitert werden, so ist es leicht möglich eine Vielzahl unserer Verbundsüberlegungen aus Abschnitt **??** zu übernehmen.

Weiterhin gibt es noch Probleme bei der Standardisierung, die wir nicht ausführlich betrachtet haben. So ist weiterhin offen, wie genau man mit komplexeren arithmetischen Ausdrücken in einer Anfrage umgeht. Wir haben zwar einige Ideen aufgezeigt, diese sind aber nicht umfangreich genug, um sie in einem Algorithmus zu manifestieren. Weiterhin haben wir uns von Anfang an auf `SELECT`-Anfragen beschränkt. Um alle SQL-Anfragen abzudecken muss noch untersucht werden, wie mit den übrigen Arten von SQL-Anfragen umgegangen werden muss, damit diese vergleichbar sind.

Nützlich für unsere Anwendung wäre es außerdem, wenn wir Methoden von anderen Lernplattformen übernehmen und in unsere integrieren. Denkbar wäre es die Aufgaben mit einer Schwierigkeitsstufe zu versehen, damit man ein 'Matchmaking'-System so etablieren kann, dass der Student weder gelangweilt noch überfordert ist. Weiterhin ist denkbar viel mehr semantic-checks einzubauen, wie z. B. eine ausführliche Analyse des Wertebereichs von Attributen. Damit könnte man viele inkonsistente Anfragen bereits vor der Standardisierung erkennen. Dies würde dem Lernenden, unabhängig von der Musterlösung, mehr Feedback zu seiner SQL-Anfrage geben.

Unter Umständen haben wir den Fall, dass wir die Äquivalenz zweier Anfragen weder bestätigen, noch ablehnen können. Um die Leistung unserer Anwendung zu steigern gilt es, die Häufigkeit solcher Fälle zu verringern. Sind also Verbesserungen implementiert worden, so macht es Sinn Testreihen mit Studenten durchzuführen und zu messen, ob die Häufigkeit dieser ungünstigen Fälle zurückgegangen ist.

Literaturverzeichnis

[1] `http://zql.sourceforge.net`. 17

[2] `http://www.javacc.org`. 17

[3] A. Mitrovic: *Learing SQL with a computized tutor* in: ACM SIGCSE Bulletin, Volume 30 Issue 1, Mar. 1998, Pages 307-311. 9

[4] P. Brusilovsky, S. Sosnovsky, D. H. Lee et al: *An open integrated exploratorium for database courses* in: ACM SIGCSE Bulletin ITiCSE 08, Volume 40 Issue 3, September 2008, Pages 22–26. 11

[5] R. Kearns, S. Shead, A. Fekete: *A Teaching System for SQL*, March 7, 1997 12

[6] C. Goldberg: *Do you know SQL? About semantic errors in database queries*: TLAD 2009 (BNCOD 2009), Birmingham, United Kingdom 13, 14

[7] Sha Guo, Wei Sun, and Mark A. Weiss: *Solving satisfiability and implication problems in database systems* in: ACM Transactions on Database Systems, 21:270-293, 1996. 14, 15

[8] S. Brass, C. Goldberg: *Proving the Safety of SQL Queries* in: 5th Intern. Conf. On Quality of Software, Melbourne, Australia, 2005. 14, 15

[9] S. Brass, C. Goldberg: *Semantic Errors in SQL Queries: A Quite Complete List* in: 4th Intern. Conf. On Quality of Software (QSIC), Brunswick, Germany, 2004 15, 16, 63, 69

[10] U. S. Chakravarthy, J. Grant, J. Minker: *Logic-based approach to semantic query optimization* in: ACM Transactions on Database Systems (TODS), Volume 15 Issue 2, June 1990, Pages 162-207.

[11] Q. Cheng, J. Gryz, F. Koo et al: *Implementation of Two Semantic Query Optimization Techniques in DB2 niversal Database* in: VLDB '99 Proceedings of the 25th International Conference on Very Large Data Bases, Pages 687-698. 60, 62

[12] S. Brass: *Vorlesung: Datenbanken I* an der Martin-Luther-Universität Halle-Wittenberg, 2011, Page 5-131. 3, 64

[13] G, Hill, A. Ross: *Reducing outer joins* in: The VLDB Journal — The International Journal on Very Large Data Bases Volume 18 Issue 3, June 2009, Pages 599-610 63

[14] http://troels.arvin.dk/db/rdbms/, Stand: 12.6.2013 67

7 Anhang

Hiermit versichere ich, dass ich die Abschlussarbeit bzw. den entsprechend gekennzeichneten Anteil der Abschlussarbeit selbständig verfasst, einmalig eingereicht und keine anderen als die angegebenen Quellen und Hilfsmittel einschließlich der angegebenen oder beschriebenen Software benutzt habe. Die den benutzten Werken bzw. Quellen wörtlich oder sinngemäß entnommenen Stellen habe ich als solche kenntlich gemacht.

Halle (Saale), 27. August 2013