

MASTERARBEIT

Vergleich von SQL-Anfragen Theorie und Implementierung in Java

ROBERT HARTMANN

BETREUER: PROF. DR. STEFAN BRASS

28. AUGUST 2013



Inhaltsverzeichnis

1	Einleitung / Motivation	4
1.1	Motivation	4
1.2	Aufgabenstellung	5
1.3	Aufbau der Arbeit	7
1.4	Produkt	8
2	Forschungsstand und Einordnung	9
2.1	Einleitung	9
2.2	SQL-Tutor	9
2.3	SQL-Exploratorium	10
2.3.1	Interactive Examples	11
2.3.2	SQL Knowledge Tester	11
2.3.3	Weiteres	12
2.4	WIN-RBDI	12
2.5	SQLLint	13
2.5.1	Algorithmus zum Finden von inkonsistenten Bedingungen	14
2.5.2	Bedingungen ohne Unteranfragen	14
2.5.3	Unteranfragen	15
2.5.4	Unnötige logische Komplikationen	15
2.5.5	Laufzeitfehler	16
3	Theoretische Betrachtungen	17
3.1	Hintergrund	17
3.1.1	Standardisierung	18
3.1.2	Elementare Transformationen	19
3.1.3	Erwartungen	19
3.2	Workflow	20
3.3	Preprocessing	20
3.4	Standardisierung von SQL-Anfragen	22
3.4.1	Entfernen von syntaktischen Details	22
3.4.2	FROM-Teil	23
3.4.3	WHERE-Teil	23
3.4.4	Ersetzung von syntaktischen Varianten	26
3.4.5	JOIN-Eliminierung	28
3.4.6	Operatorenvielfalt	31
3.4.7	Sortierung	38
3.4.8	GROUP BY / HAVING	40
3.4.9	ORDER BY	41
3.4.10	Abschluss	41

3.5	Anpassung durch elementare Transformationen	41
3.5.1	Backtracking	42
3.6	Weitere Betrachtungen	43
3.6.1	Anzahl atomarer Formeln	43
3.6.2	Anzahl der Operatorkompressionen	44
3.6.3	unnötiges DISTINCT	44
3.6.4	Algorithmus aus [12]	44
4	Verwendete Software	46
4.1	SQL-Parser	46
4.1.1	über den SQL-Parser: ZQL	46
4.1.2	Funktionsweise des Parsers	46
4.1.3	Grenzen des Parsers	50
4.2	Java Server Pages	51
4.2.1	Überblick	51
4.2.2	Einbettung in JSP	51
4.2.3	Log	51
5	Praktische Umsetzung	52
5.1	Anforderungen	52
5.2	Funktionsumfang	52
6	Ergebnisse	53
7	Ausblick	54
8	Anhang	56
8.1	Algorithmus für Backtracking	57

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 Oracle, 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 verständlich 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. Manche sehen sich dabei ähnlich, andere Anfragen 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. Dies 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 universitären 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 auch keine gute Lösung ist. Häufig werden dem Lernenden Übungsaufgaben gestellt, die dieser dann innerhalb einer Frist bearbeitet und abgibt.

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

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 Fehlermeldung 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 Bemerkungen ist hierbei, dass zwei semantisch äquivalente Anfragen keinesfalls syntaktisch äquivalent sein müssen, wie das folgende Beispiel zeigt:

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 7 wird noch auf Anfragen eingegangen, die weder im Schritt 1 bestätigt noch im Schritt 2 abgelehnt werden konnten.

Ein mögliches 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 ihm möglich beim Üben direkt zu lernen. Weiterhin kann man eine solche Plattform auch für Tutorien oder Nachhilfe überall da benutzen, 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.

Weitere Einsatzgebiete könnten sich im Unternehmen befinden. So könnte man bei einer geplanten Umstrukturierung oder Erzeugung von Datenbanken bereits Anfragen prüfen und vergleichen, bevor man sich u.U. teure Testdaten kauft oder Daten migrieren muss.

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 eine Feinheit spezialisiert ist und andere Punkte 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 7 genauer erläutert.

Die Fragestellung “Sind zwei SQL-Anfragen äquivalent” ist nicht entscheidbar. Aus diesem Grund klären wir im Kapitel 3 wie wir den Entscheidungsprozess angehen wollen, sodass zumindest eine Teilmenge von SQL-Anfragen bearbeitet werden kann. Wir klären also zunächst, wie unser Programm vorgehen wird. Danach werden die einzelnen Schritte, die zum Vergleich notwendig sind, erklärt und besprochen. Dabei diskutieren wir für einzelne Teilschritte auch mehrere Herangehensweisen. Weiterhin werden wir alle möglichen Analyseschritte theoretisch untermauern, auch wenn nicht alle davon im Programm umgesetzt werden können. Mehr dazu im Abschnitt »Grenzen des Parsers«. Alle später umgesetzten Algorithmen werden in diesem Kapitel vorgestellt, erarbeitet und diskutiert.

Im nachfolgenden Kapitel 4 beschreiben wir die verwendete Software. Neben der üblichen Beschreibung der verwendeten Software, wird insbesondere auf den verwendeten Parser und die Java-Servelets eingegangen. Zu klären ist hier wie genau der Parser funktioniert und was er nicht kann. Daraus leitet sich eine gewisse Beschränkung in der praktischen Umsetzung ab. Da wir im vorherigen Kapitel allerdings alle theoretischen Betrachtungen ausführlich erläutert haben, stellt es kaum ein Problem dar, die vorgestellten Algorithmen auf einen anderen Parser zu übertragen. Ein weiterer Aspekt dieses Kapitels ist es, dem Leser klar zu machen wie Java-Servelets funktionieren und wie genau wir sie für unser Programm einsetzen.

In Kapitel 5 wird der Aufbau des Programms geklärt und erläutert. Dabei gehen wir den strukturellen Aufbau durch und klären, wie einzelne Aspekte aus Kapitel 3 umgesetzt werden konnten. Weiterhin klären wir, was aufgrund gewisser Beschränkungen nicht umsetzbar war. Wir diskutieren die Struktur des Programms hier eingehend auf Konzepte der Softwaretechnik, wie z. B. Wartbarkeit, Erweiterbarkeit usw.

1.4 Produkt

Es wurde im Rahmen der Aufgabenstellung ein Programm entwickelt, das es erlaubt zwei SQL-Anfragen miteinander zu vergleichen. Dazu wurde eine Lernplattform auf Basis von Java-Servlets geschaffen. Der Lernende meldet sich an der Plattform an und wählt eine Kategorie aus. Nun wird ihm eine Sachaufgabe gestellt und ein Datenbankschema angezeigt. Er soll nun daraus eine SQL-Anfrage formulieren, die die Aufgabenstellung löst. Dabei bekommt der Lernende Feedback von dem Programm. Dies schließt sowohl Hinweise als auch konkrete Fehlermeldungen ein. Hat der Lernende die Aufgabe bereits mehrfach bearbeitet, so kann er sich seine vorherigen, eingesandten Lösungen anschauen und seinen Lernerfolg leicht verfolgen. Das Programm zeigt auch an, in welcher Kategorie der Lernende noch große Defizite hat.

Der Dozent hat das Programm vorher einmalig mit einer Reihe von Aufgaben bestückt. Dazu gibt der Dozent eine textuelle Beschreibung der Aufgabe, eine oder mehrere SQL-Anfragen als Musterlösungen, ein Datenbankschema und optional eine Datenbank an, auf der Beispieldaten vorhanden sind.

Das Programm läuft im Wesentlichen in zwei Schritten ab. Im ersten Schritt versucht es die zwei Anfragen miteinander zu vergleichen, ohne den Einsatz von externen Daten. Gelingt dies, ist gezeigt, dass die Lösung des Studenten mit der Musterlösung übereinstimmt. Das Programm meldet Erfolg und zeigt eventuell abweichende Komplexitätsmaße an. Mehr dazu im Abschnitt [].

Schlägt der erste Schritt fehl, wird die Anfrage des Lernenden auf der angegebene Datenbank verarbeitet und mit den Ergebnistupeln verglichen, die die Musterlösung liefern. Sind beide in allen Beispieldaten gleich, so wird dem Dozenten gemeldet, dass eine eventuelle neue Musterlösung gefunden wurde. Diese ist strukturell so unterschiedlich, dass sie nicht auf die bisherige Musterlösung angepasst werden konnte. Wir können in einem solchen Fall nicht mit Sicherheit sagen, ob die Lösung falsch oder richtig ist, da dieses Problem im Allgemeinen nicht entscheidbar ist. Daher muss ein Prüfer solche Lösungen noch einmal manuell kontrollieren.

Schlägt aber auch der zweite Schritt fehl, so können wir sicher sein, dass die Lösung des Lernenden falsch ist. Das Programm meldet dann eine Fehlermeldung sowie mögliche Hinweise, was der Lernende falsch gemacht haben könnte.

2 Forschungsstand und Einordnung

2.1 Einleitung

Die Idee SQL-Anfragen von Lernenden auszuwerten ist nicht völlig neu. Weil eine Auswertung über den Standard SQL-Parser nicht sehr umfangreich ist und es bei 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 kurz betrachten, um dann diese Arbeit dann einordnen zu können.

2.2 SQL-Tutor

In [3] beschreibt Antonija Mitrovic ein Lernsystem, was SQL-Tutor genannt wird. Nach Auswahl einer Schwierigkeitsstufe wird dem Studenten ein Datenbankschema und eine Sachaufgabe vorgelegt. Der Student hat nun ein Webformular 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 »The SELECT Clauses of all solutions must not be

empty« 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 $a \geq 0$ sind vorhanden.

Allerdings gibt es auch hier Schwächen. Da der verwendete Algorithmus die Constraints nacheinander 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.

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 die Anfragen auf konkreten Daten. Hat aber bereits unser 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 vorhandenen Testdaten zufällig beide Anfragen die gleichen Ergebnisse liefern.

2.3 SQL-Exploratorium

Im Artikel [4] werden SQL-Lernplattformen in zwei Kategorien eingeteilt. Zum einen existieren Plattformen, 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 die Fragen als solche eher unwichtig für diese Arbeit sind, 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 uninteressant 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.

Abgrenzung zur Arbeit

Wie bereits erwähnt wird bei den Interactive Examples nichts automatisch erzeugt, was diesen Ansatz für diese Arbeit uninteressant macht.

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 so wie 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, in dem 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 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.

SQL-Lab

SQL-Lab ist lediglich ein Hilfsmittel um SQL-KnoT zu benutzen und daher für diese Arbeit auch nicht von Bedeutung.

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 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 zu begreifen. Diese Arbeit verfolgt allerdings ein anderes Ziel, da sie zwei SQL-Anfragen miteinander vergleicht und nicht versucht die Abarbeitung einer Anfrage zu visualisieren.

2.5 SQLLint

»SQLLint - Detecting Semantic Errors in SQL Queries« ist ein Projekt der Martin-Luther-Universität Halle-Wittenberg. Es beschäftigt sich mit semantischen Fehlern in SQL-Anweisungen, welche, unabhängig vom Datenbankzustand, nicht gewollt sind. 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. Eine generelle Frage der Gültigkeit solcher SQL-Anfragen 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.

Ein nicht unwesentlicher Ansatz des Projektes ist es, solche Fehlermeldungen in der Lehre einzusetzen. In [6] wird auch deutlich gemacht, dass 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 *figure 4* in [6].

2.5.1 Algorithmus zum Finden von inkonsistenten Bedingungen

Wie bereits erwähnt, soll der Algorithmus im SQLLint-Projekt 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 Datenbankstatus, 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 runter zu den atomaren Formeln »gedrückt«. Dadurch drehen sich die Vergleichsoperatoren um, wir sprechen hierbei von »opposite operator«. Die Menge $O = \{\leq, >, \geq, <, =, \neq\}$ enthält jeweils 2er Mengen von einem Operator und seinem »opposite operator«. Im nächsten Schritt wird die Bedingung in die disjunktive Normalform (DNF) umgeformt, so dass folgende Struktur entsteht: $\phi_1 \vee \dots \vee \phi_n$. 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 gelabelt 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 führen kann.

2.5.3 Unteranfragen

Um unnötige Betrachtungen zu vermeiden, beschäftigt sich das SQLLint-Projekt nur mit EXISTS-Unteranfragen. Alle anderen Unteranfragen (IN, \geq 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 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 $A \text{ IS NOT NULL}$. 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 $C_1 \vee \dots \vee C_m$, mit $C_i = (A_{i,1} \wedge \dots \wedge A_{i,n_i})$. Unser Kriterium ist genau dann erfüllt, wenn die folgenden Formeln alle konsistent sind:

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

3. $A_{i,1} \wedge \dots \wedge A_{i,j-1} \wedge \neg A_{i,j} \wedge A_{i,j+1} \wedge \dots \wedge A_{i,n_i} \wedge \neg(C_1 \vee \dots \vee C_{i-1} \vee C_{i+1} \vee \dots \vee C_m)$ 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 (\geq 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: $A = (\text{SELECT } \dots)$ 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 Thesis sein. Der Ansatz der Standardisierung der SQL-Anfragen ist mit umwandeln der Formeln in eine DNF ein guter Ansatzpunkt, 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 vereinfacht.

Weiterhin könnte man in späteren Ausbaustadien des Programmes, welches im Rahmen dieser Arbeit entsteht, die Funktionalitäten des SQLLint einbauen. 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 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. Dieser Ansatz wird uns mit einigen Problemen konfrontieren und daraus entwickeln wir einen zweiten Ansatz. Dieser versucht durch gleichartige Umformungen, die zwei Anfragen zu unifizieren (gleich zu machen). Bei diesem Ansatz würden wir also versuchen die geparsten Operatorbäume miteinander zu vergleichen. Auch diese Lösung bringt Vorteile aber auch Probleme mit sich, die im Folgenden besprochen werden.

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 praktischen Teil, dem Programm, auftauchen. Die Erklärung für diese Beschränkungen erfolgt in diesem Kapitel sowie im Abschnitt 4.1.3 und im Kapitel 5.

3.1 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.

```
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 ähnlich. Im folgenden werden zwei Strategien besprochen, welche beide zum Ziel haben, zwei SQL-Anfragen miteinander zu vergleichen.

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 derlei Problemen beschäftigt sich das SQLLint-Projekt der Martin-Luther-Universität Halle-Wittenberg, mehr dazu im Artikel [8] und [9].

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.

3.2 Workflow

Bereits in der Einleitung haben wir uns mit hinreichenden und notwendigen Bedingungen von semantischer Äquivalenz beschäftigt. Wir prüfen also zunächst die hinreichende Bedingung für die semantische Äquivalenz. Können beide Anfragen syntaktisch äquivalent gemacht werden, so ist klar, dass beide Anfragen auch semantisch äquivalent sind. Wir betrachten dazu zwei verschiedenen Ansätze.

3.2.1 Standardisierung

Im ersten Ansatz unterziehen wir jeder Anfrage einem Standardisierungsprozess. Alle vorkommenden Variablen im `FROM`-Teil werden nach ihrem Namen lexikographisch sortiert und erhal-

ten einen automatisch generierten Alias. Diese Aliase werden in allen anderen Teilen der SQL-Anfrage ersetzt durch bereits vorhandene oder eingeführt, falls vorher keine Aliase benutzt worden. Die meiste Arbeit wird im WHERE-Teil erledigt.

Zunächst 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 sind diese ebenso semantisch äquivalent.

3.2.2 Elementare Transformationen

Der zweite Ansatz ist das Anwenden von elementaren Transformationen auf einer Anfrage A um sie an eine zweite Anfrage B anzupassen. Dabei wird immer versucht Teile der Anfrage A durch Anwendung von elementaren Transformationen auf Teile von Anfrage B anzupassen.

Zunächst wird versucht den FROM-Teil von Anfrage A anzupassen, indem wir die Anordnung der Variablennamen permutieren. Ist dies gelungen werden Aliase, die Anfrage A jetzt verwendet, in den restlichen Teilen der Anfrage A übernommen, ähnlich wie im Ansatz »Standardisierung«. Im WHERE-Teil wird durch sukzessives Backtracking versucht beide WHERE-Ausdrücke aneinander anzupassen. Ähnlich wird dann mit GROUP BY und ORDER BY Ausdrücken vorgegangen. Konnte das Backtracking Erfolg vermelden, so wissen wir, dass beide Anfragen semantisch äquivalent sind.

3.2.3 Erwartungen

Wir erwarten, dass die Anpassung durch elementare Transformationen einfacher zu vollziehen ist, dafür aber eine hohe Laufzeit vorweisen wird. Dies ist zu begründen durch die exponentiell vielen Vergleiche, die beim Backtracking im worst-case auftreten können.

Dem entgegen erwarten wir eine schnellere Laufzeit bei der Standardisierung. Dafür müssen hier deutlich mehr Teilschritte eingebaut werden und muss genau klar sein, welche Anpassungen wirklich eindeutig sind.

```
INPUT: QUERY Q1,Q2;
P1 = preprocessing(Q1);
P2 = preprocessing(Q2);
compare(P1,P2); // possible warnings can be displayed now
ANSWER = match(Q1,Q2);
if ANSWER yes then
    /* If that worked, we know both solutions are the same */
    display success
else
    if do_real_db_compare(Q1,Q2) then
        /* now we don't know if they are the same because
        * they couldn't be matched but test on real data
        * showed the correct results
        display may be correct
    else
        /* if the real data test failed we have a proof
        * in form of a data set, that both queries can't be the same */
        display fail
    endif
endif
output result of compare(P1,P2)
/* The result may show the cause of a fail or a 'may be' solution.
* It can provide hints so that the student can improve.
* Even if the solution was correct i.e. it was matched with the sample solution,
* it may be that the students solution contained unnecassary joins, or formulas. */
```

3.3 Preprocessing

Im Abschnitt »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, sondern auch, weil die Standardhinweise eines SQL-Systems meist nur auf syntaktische Fehler hinweisen. Einen großen Beitrag zur Verbesserung von Fehlermeldungen hat das Projekt SQLLint vorzuweisen, da es Fehlermeldungen und Hinweise konkreterer Natur ausgibt. Hervor-

zuheben ist noch einmal, dass es sich hierbei um semantische Fehlermeldungen handelt. Schon nach kurzer Einlernzeit sinkt die Anzahl syntaktischer Fehler bei Lernenden. Dafür machen diese mehr semantische Fehler, was um so schlimmer ist, da bisher kaum oder keine Warnhinweise für solche Fehler existierten.

Dennoch sollen in dieser Arbeit zwei SQL-Anfragen verglichen werden. Wir können hier also nicht alle Ideen des SQLLint übernehmen. Egal ob das Matchen der Musterlösung und der Lösung des Lernenden gelingt oder nicht, wir möchten dem Lernenden ein Feedback geben, an dem er idealerweise sehen kann, warum das Matching nicht gelungen ist. Wir können dabei, wie bereits erwähnt, nicht an die Komplexität des SQLLint anknüpfen. Stattdessen werden wir uns eines einfachen Sammelns von Metainformationen der SQL-Anfrage bedienen. Diese sammeln wir bevor die zwei Anfragen durch Folgeschritte angepasst oder verändert werden. Am Ende des Matchingsversuchs sollen die Metainformationen der zwei Anfragen verglichen werden und dem Lernenden soll ein Feedback gegeben werden. Konnte keine Übereinstimmung der zwei Anfragen erreicht werden, so können die Metainformationen dem Lernenden Anhaltspunkte für eine richtige Lösung geben. Konnten die Anfragen unifiziert werden, so sind die Metainformationen dennoch von Interesse. Es könnte sein, dass der Lernende eine unnötig komplexe Lösung eingesandt hat, die sich durch Anpassungen vereinfachen ließe. So kann der Lernende potentiell auch aus einer korrekten Lösung noch etwas lernen.

Wir möchten für jede SQL-Anfrage ein Preprocessing vor der eigentlichen Bearbeitung vorschalten, was im Wesentlichen folgende Punkte beinhalten soll:

- 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
- existiert ein ORDER BY und ist es notwendig? (ORDER BY ... ASC) *

- Tiefe des Parserbaums kann Aufschluss über unnötige Klammerung geben. Siehe dazu Abschnitt »Wie funktioniert der Parser«

Unterscheiden sich Musterlösung und Lösung des Studenten in den mit * markierten Punkten ist es extrem unwahrscheinlich, dass beide Lösungen die gleichen Tupel zurückliefern würden. Hier möchten wir im Vorfeld dem Lernenden eine Warnung anzeigen, dass er höchstwahrscheinlich etwas vergessen hat. Alle anderen Punkte werden im Anschluss an die eigentliche Analyse der Anfragen abgeglichen. So sind etwa folgende Meldungen denkbar:

- “The sample solution contains two joins but your solution does not contain any join.”
- “Your solution is correct but the sample solution contains two less atomar formulas (formula1, formula2).”
- “Your solution is correct but the sample solution does not contain DISTINCT. Reconsider if it is really necessary.”

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 dem zweiten Schritt, dem Angleichen der SQL-Anfragen. Dazu verwenden wir Strategien, die in folgenden Kapiteln besprochen werden.

Egal ob die Ergebnisse im zweiten Schritt erfolgreich waren oder nicht, wir geben danach einen Vergleich der Metainformationen aus. Beispiele wurden eben bereits genannt. 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. kompliziert sein. Dies würde bei einem Vergleich der gesammelten Metainformationen deutlich werden.

3.4 Standardisierung von SQL-Anfragen

Zunächst verfolgen wir den Ansatz zwei SQL-Anfragen zu vergleichen, indem wir sie standardisieren. 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 Äquivalenz schließen.

3.4.1 Entfernen von syntaktischen Details

Das Entfernen von syntaktischen Details übernimmt zum großen Teil bereits der Parser. Er entfernt unnötige Leerzeichen, Kommentare sowie unnötige Klammern. Aufgrund der Arbeitsweise des

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 3.1: Beispiel: Umwandlung des FROM-Teils einer SQL-Anfrage

Parsers gibt es allerdings Situationen, in dem der Parser scheinbar nicht alle unnötigen Klammern entfernt. Wie im Abschnitt »Verwendeter Parser« erläutert wird, sind die geparsten Bäume nicht binär. Ein Baum wie in Abbildung 4.1.2 zu sehen, ist daher zu vermeiden.

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.

3.4.2 FROM-Teil

Wir beginnen mit der Betrachtung der FROM-Klausel. Da die Reihenfolge der Spaltennamen im SELECT-Teil oft von der Aufgabenstellung vorgeschrieben ist, wird diese auch nicht verändert.

Im FROM Teil werden zunächst alle auftretenden Tabellennamen lexikographisch sortiert. Danach werden automatische Aliase erzeugt. Sind bereits Aliase vergeben wurden, so werden diese ebenfalls durch die automatischen Aliase ersetzt. Eine Hashtabelle speichert frühere Zuweisungen, damit im SELECT-, WHERE-, GROUP BY- und ORDER BY-Teil die Aliase ebenfalls korrekt ersetzt werden.

Hatten die vorkommenden Tabellen im FROM-Teil keinen Alias wird nur der künstliche Alias eingeführt.

3.4.3 WHERE-Teil

Aufgrund der Eigenheiten des ZQL-Parsers ist es möglich, dass eine unnötige Klammerung nicht entfernt wird. Beispiele dafür sind im Abschnitt »ZQL-Parser« zu finden. Es ist daher wünschenswert eine Normalform des WHERE-Teils zu erreichen. In diesem Fall wurde die konjunktive Normalform (KNF) gewählt.

Entfernung unnötiger Klammerungen

Ein Ausdruck $((a > 5) \text{ and } ((b > 5) \text{ and } (c > 5)))$ enthält unnötige Klammern, da der Operator *and* als Operand von einem weiteren *and* vorkommt. Folgender Ausdruck ist äquivalent: $((a > 5) \text{ and } (b > 5) \text{ and } (c > 5))$. 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 *X* einen Ausdruck als Kindknoten besitzt, indem *X* ebenfalls der Operator ist, den Operator *X* im Kindknoten zu eliminieren. Alle Kinder vom eliminierten Kindknoten an hängen wir nun an den verbleibenden Operator *X*. Damit hätte man den Ausdruck vereinfacht, da die assoziative Klammerung wegfällt. Wir nennen dieses Vorgehen im Folgenden Operatorkompression. Verbildlicht wird dieser Vorgang im Abschnitt »Funktionsweise des Parsers«.

Gegeben sei der ZQL-Parserbaum $B = (V, E)$. Es sei $child(v) = \{w : w \in V \wedge (v, w) \in E\}$, also die Menge aller Kindknoten von *v*. Gibt es einen Knoten $w \in child(v)$ mit $v = w$, so wird Knoten *w* eliminiert und alle Kindknoten von *w* werden zu Kindknoten von *v*, also $child(v) = child(v) \cup child(w)$. $E = E \setminus \{(w, x) : x \in child(w)\} \cup \{(v, x) : x \in child(w)\}$ und $V = V \setminus \{w\}$.

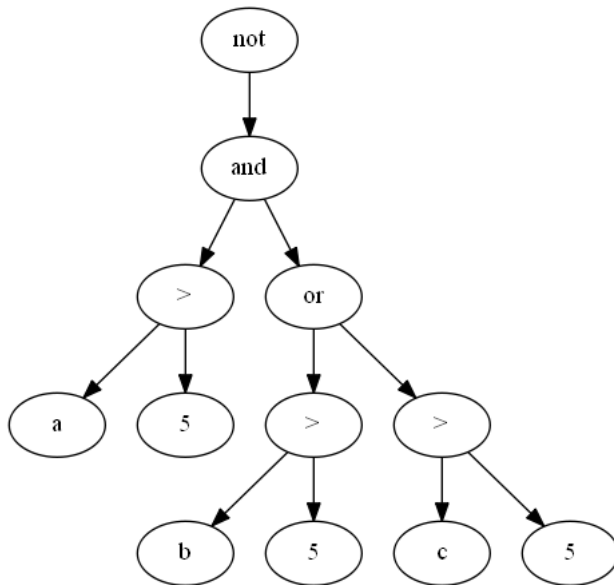
Im Sinne des Vergleiches der Komplexität der Musterlösung mit der Komplexität der Lösung des Lernenden ist es sinnvoll zu speichern, ob und wie oft eine solche Operatorkompression durchgeführt werden musste.

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.

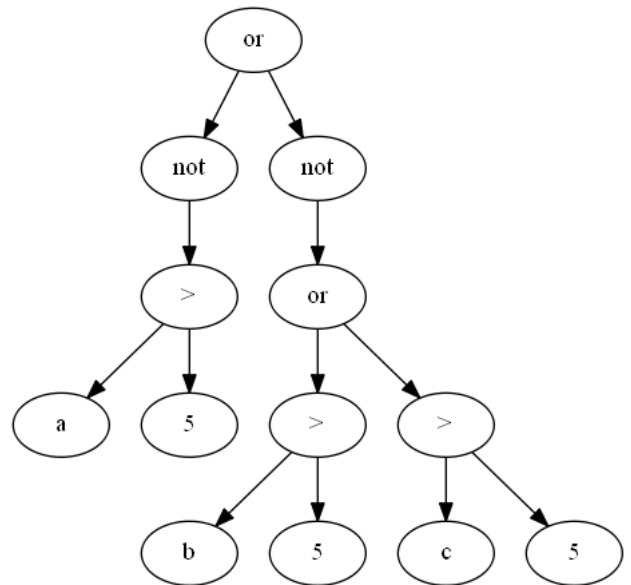
Eingabe:

$\text{not } ((a > 5) \text{ and } ((b > 5) \text{ or } (c > 5)))$



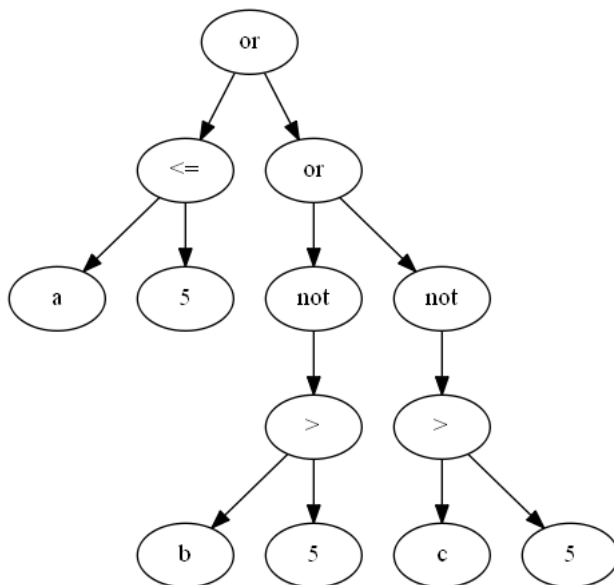
Umwandlung Teil 1:

$(\text{not}(a > 5) \text{ or } \text{not}((b > 5) \text{ or } (c > 5)))$



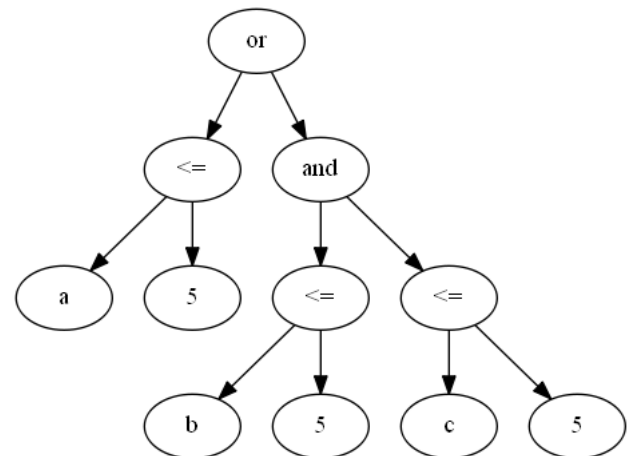
Umwandlung Teil 2:

$((a \leq 5) \text{ or } (\text{not}(b > 5) \text{ and } \text{not}(c > 5)))$



Umwandlung Teil 3:

$((a \leq 5) \text{ or } ((b \leq 5) \text{ and } (c \leq 5)))$



Anwenden des Distributivgesetzes

Im letzten Schritt haben wir die Formel $((a \leq 5) \text{ or } ((b \leq 5) \text{ and } (c \leq 5)))$ erhalten. Durch Anwenden des Distributivgesetzes können wir diese Formel im letzten Schritt umformen zu: $((a \leq 5) \text{ or } (b \leq 5)) \text{ and } ((a \leq 5) \text{ or } (c \leq 5))$

3.4.4 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. Zu erwähnen sind folgende Ersetzungen, die durchgeführt werden sollen um syntaktisch vielfältige, aber semantisch äquivalente Ausdrücke zu minimieren:

A BETWEEN B AND C	→	A >= B AND A <= C
SELECT ALL	→	SELECT
ORDER BY VAR ASC	→	ORDER BY VAR
A IN ('X', 'Y', 'Z')	→	A = 'X' OR A = 'Y' OR A='Z'
EXISTS (SELECT A,B,C ...)	→	EXISTS (SELECT 1 ...)

Abbildung 3.2: Entfernen von syntaktischen Varianten

Unterabfragen

Es ist bekannt, dass sämtliche Typen von Unterabfragen 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 Unterabfragen in diesem Abschnitt betrachten.

Ersetzen von ALL

... SAL >= ALL(1000, LOW_SAL) wird zu:
... SAL >= 1000 AND SAL >= LOW_SAL

Ersetzen von ANY/ SOME

X.SAL >= ANY(SELECT Y.SAL FROM EMP Y
WHERE Y.JOB = 'CLERK')

zu

EXISTS(SELECT Y.SAL FROM EMP Y
WHERE Y.JOB = 'CLERK'
AND X.SAL >= Y.SAL)

Befindet sich ein NOT vor der Unterabfrage, so wird dieses nicht zur Unterabfrage “durchgedrückt” sondern bleibt davor.

Ersetzen von IN

Unter bestimmten Voraussetzungen kann jede IN-Unterabfrage in eine äquivalente EXISTS-Unterabfrage umgewandelt werden.

Wir wandeln IN-Unterabfragen wie die Folgende:

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

unter Voraussetzungen, die noch erläutert werden, um in:

```
EXISTS (SELECT *
        FROM R1 X1, . . . , Rn Xn
        WHERE (φ) AND t1 = t2)
```

Folgende Voraussetzungen müssen erfüllt sein, damit diese Umwandlung angewendet werden kann:

- 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.
- 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.
- Die Unterabfrage für t_2 darf keine Nullwerte liefern.

Andere Unterabfragen

Ungewöhnliche 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.

JOINS

Ein INNER JOIN kann sowohl im FROM-, als auch im WHERE-Teil einer SQL-Anfrage formuliert werden. Damit Untersuchungen einheitlich geschehen können, formulieren wir solche JOINS im WHERE-Teil der SQL-Anfrage.

Eingabe:

```
SELECT * FROM foo f INNER JOIN bar b ON f.id=b.id
```

Umwandlung:

```
SELECT * FROM foo f, bar b WHERE f.id=b.id
```

Abbildung 3.3: Umwandlung von INNER-JOIN

Bei Anwendung dieser Ersetzungsregeln, soll dem Lernenden ein klares Feedback gegeben werden. Es soll verdeutlicht werden, dass eine korrekte Anfrage dennoch Mängel aufweist, da unnötige Formulierungen benutzt wurden.

Es ist hier bereits möglich Terme, die nur aus numerischen Konstanten bestehen, zu ersetzen durch das jeweilige Ergebnis. So könnten arithmetische Operationen bereits ausgeführt und Vergleiche, die nur aus numerischen Konstanten bestehen, durch entsprechende Wahrheitswerte ersetzt werden.

3.4.5 JOIN-Eliminierung

In einem weiteren Zwischenschritt möchten wir gern einige unnötige JOINS eliminieren.

OUTER JOIN

Betrachten wir eine SQL-Anfrage mit einem OUTER JOIN. Befinden sich im SELECT-Teil keine Attribute der JOIN-Tabelle, so ist der JOIN unnötig, weil Daten von der JOIN-Tabelle ohnehin nicht ausgegeben werden. Betrachten wir dazu folgendes Beispiel:

```
SELECT s.vorname, s.nachname  
FROM studenten s LEFT JOIN klausur k ON s.id = k.id  
WHERE k.wert <= 3
```

Abbildung 3.4: Unnötiger OUTER JOIN

Im Beispiel in Abbildung 3.4 sollen alle Studenten ausgegeben werden, die in einer Klausur die Note drei oder besser bekommen haben. Da es sich hier um einen OUTER JOIN handelt und wir nur Namen der Studenten ausgeben möchten, erhalten wir immer eine gesamte Liste der Studentennamen. Der Teil, der für den JOIN interessant wäre, also Daten aus der Tabelle klausur k, wird nicht ausgegeben. Damit wäre die folgende SQL-Anfrage mit der aus Abbildung 3.4 äquivalent.

```
SELECT s.vorname, s.nachname
FROM studenten s
```

Abbildung 3.5: Anfrage ohne JOIN

Bei einem RIGHT JOIN müsste man äquivalent überprüfen ob man Attribute im SELECT-Teil hat, die in der “linken” Tabelle sind und genauso verfahren.

Möchte man tatsächlich nur die Studenten ausgeben, die eine drei oder besser geschrieben haben, müsste man einen INNER JOIN verwenden.

Transitiv-implizierte INNER JOINS

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

In der Arbeit [11] wird dazu ein Algorithmus angegeben, der im Wesentliche oben genannte, überflüssige Tupelvariablen entfernt. Wir wandeln diesen Algorithmus leicht ab, erhalten aber die Grundidee.

Im ersten Schritt erstellen wir transitiv-abgeschlossene Äquivalenzklassen der Attribute, die im WHERE-Teil vorkommen. Haben wir also $X.A = Y.B$ und $Y.B = Z.C$, so befinden sich alle drei Attribute in einer Äquivalenzklasse: $\{X.A, Y.B, Z.C\}$. Eine Äquivalenzklasse enthält nur Tupelvariablen und ist transitiv abgeschlossen über dem Operator $=$. Dabei bemerken wir, dass Äquivalenzklassen der Mächtigkeit 1 einfache Vergleiche wie $X.A \text{ Operator Konstante}$ sind. Klassen der Mächtigkeit 2 sind einfache, nicht transitive JOINS und Klassen der Mächtigkeit größer als 2 sind offensichtlich mehrfache, transitive JOINS.

Beispiel:

```
SELECT t1.x
FROM test1 t1, test2 t2, test3 t3
WHERE t1.x = t2.y
AND t1.x = t3.z
AND t1.y = t2.z
AND t1.z > 3;
```

Bei diesem Beispiel erhalten wir die Äquivalenzklasse $\{t1.x, t2.y, t3.z\}$ sowie die Klassen $\{t1.y, t2.z\}$ und $\{t1.z\}$.

Im nächsten Schritt gehen wir jede Äquivalenzklasse durch, die mindestens drei Einträge haben. Für jeden Eintrag $e \in \text{Äquivalenzklasse}$ mit $e = T.A$ überprüfen wir, ob es andere Äquivalenzklassen gibt, die nicht aus einem transitiven JOIN entstanden sind (also weniger als drei Einträge

haben) und die ein Attribut der Tabelle *T* enthalten. Ist dies nicht der Fall und die Tabelle *T* kommt nicht im SELECT-Teil vor, dann ist das Attribut *A* in dem Vergleich, der durch die Äquivalenzklasse repräsentiert wird, unnötig und kann samt zugehöriger Bedingung im WHERE-Teil gestrichen werden. Hat die Äquivalenzklasse jetzt weniger als drei Einträge dann wird die Arbeit an dieser Klasse abgebrochen.

Im letzten Schritt müssen wir noch überprüfen, ob es Tupelvariablen im FROM-Teil gibt, die aber nun nicht mehr im WHERE-Teil auftauchen. Ist dies der Fall, dann streichen wir diese aus dem FROM-Teil.

NULL bearbeiten

Beispiel:

```
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;
```

Zunächst erzeugen wir alle Äquivalenzklassen:

```
Klassen =
{{s.suppkey, ps.suppkey, c.custkey, o.custkey }, {o.totalprice}}
```

Die zweite Menge interessiert uns nicht, da sie nicht mehr als zwei Elemente enthält. Wir betrachten daher nur die erste Menge. Wir untersuchen nun jedes einzelne Element auf seine Gültigkeit. *s.suppkey* kommt in keiner anderen Äquivalenzklasse vor und die Tabelle *supplier s* erscheint nicht im SELECT-Teil, daher streichen wir *s.suppkey*. Das nächste Attribut *ps.suppkey* kann nicht gestrichen werden, da die Tabelle *partsupp ps* im SELECT-Teil erscheint. *c.custkey* kann wiederum gestrichen werden, da es wieder nicht in einer anderen Äquivalenzklasse auftaucht und die Tabelle *customer c* nicht im SELECT-Teil auftaucht. Dahingegen ist *o.custkey* nicht zu streichen, da die Tabelle *orders o* in einer Äquivalenzklasse auftaucht, welche nicht zu einem JOIN gehört (weil sie weniger als drei Elemente beinhaltet). Nun können wir auch die Tabellen *supplier s* und *customer c* streichen, da keine Bedingung mehr Attribute aus diesen Tabellen enthält.

Daher erhalten wir nun die folgende, optimierte Anfrage:

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

PSEUDOCODE:

Alle Attribute im WHERE-Teil in Äquivalenzklassen E_i packen.

```
foreach  $e \in E_i$  mit  $|e| \geq 3$  do
  foreach  $t.a \in e$  do
    if  $|e| \geq 3$  and  $t \notin \text{SELECT}$  and
       $\nexists b \in E_i$  mit  $|b| < 3$  und  $b=t.*$ 
       $e = e - \{t.a\}$ 
    end
  done
done
```

Streiche unnötige Tabellen aus FROM-Teil

3.4.6 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: $A \geq 6$ sowie $6 \leq A$. Wir betrachten nun zwei verschiedene Ansätze um mit diesem Problem umzugehen. 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.

Hinweis: Diesem Schritt geht eine Teilsortierung vor. Diese wird ebenfalls im Abschnitt »Sortierung« erwähnt.

Teilsortierung

Wir betrachten Ausdrücke mit den Operatoren $\{>, <, \leq, \geq, =, +, \cdot\}$. Da es sich hier jeweils um binäre Operatoren handelt, sprechen wir, im Sinne der Anordnung, im Folgenden von einem linken und einem rechten Operanden. Ist einer der Operanden eine Variable, so wird diese links angeordnet. Sind beide Operanden Variablen, so werden sie lexikographisch-sortiert angeordnet. Operanden, die selbst wieder zusammengesetzte Ausdrücke sind, stehen rechts. Sind beide Operanden zusammengesetzte Ausdrücke, so steht der komplexere rechts und der weniger komplexe links. Ein Ausdruck A ist komplexer als ein Ausdruck B , wenn der zugehörige Operatorbaum von A tiefer ist als der Operatorbaum von B . Sind beide Ausdrücke gleich komplex, so wird die symmetrische Variante mit hinzugenommen. Wenn wir Operanden umsortieren bei denen der Operator

$\in \{>, <, \leq, \geq\}$ ist, dann muss der jeweilige Operator auch umgedreht werden. Bei den restlichen Operatoren ist dies nicht der Fall, da diese symmetrisch sind.

Hinzufügen implizierter Formeln

Trifft man im Parserbaum auf eine Formel, zu der es mehrere äquivalente Formeln gibt, so ist ein Ansatz alle diese äquivalenten Formeln konjunktiv zu verknüpfen und mit in den Parserbaum aufzunehmen. Treffen wir also zum Beispiel auf folgenden Ausdruck:

```
SELECT * FROM testtable WHERE A = B - C
```

, , so müssen wir auch alle äquivalenten Formeln mit aufnehmen. Daraus wird dann also der Ausdruck:

```
...WHERE A = B - C AND B = A + C AND C = B - A
```

Wie man bereits sieht, 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 Student für einen Repräsentanten der Formeln wählen wird. Weiterhin muss bemerkt werden, dass dadurch die gesamte SQL-Anfrage enorm aufgebläht wird. Es ist daher unbedingt wichtig, 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.

$M_1 \quad \{ A = B - C, C = B - A, B = A + C \}$

$M_2 \quad \{ A = B \cdot C, C = A / B, B = A / C \}$

$M_3 \quad \{ A > B - C, C > B - A, B < A + C \}$

$M_4 \quad \{ A < B - C, C < B - A, B > A + C \}$

$M_5 \quad \{ A > B, B < A \}$

$M_6 \quad \{ A \geq B, B \leq A \}$

$M_7 \quad \{ A > X, A \geq X + \text{adjust}(A) \}$

$M_8 \quad \{ A < X, A \leq X - \text{adjust}(A) \}$

Beim Vergleich mit $>$ und $<$ ist es wichtig zu wissen, wie viel 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 `salary`, welches als `NUMERIC(4,2)` definiert ist. Wir wissen also, dass `salary` zwei Nachkommastellen hat. Betrachten wir nun die Aussage `salary >= 5`. Wir haben auf einer Seite eine Variable (`salary`) 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 $A=salary$ und $x+adjust(salary)=5$. Wir berechnen also:

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

Wir erhalten also $x=4,99$, weil $x+0,01=5$. Somit ergänzen wir unsere Ausgangsformel `salary >= 5` konjunktiv mit `salary > 4,99`. Weiterhin muss jetzt wegen M_6 $5 \leq salary$ und wegen M_5 $4,99 < salary$ 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 oder Konstanten. In diesem Fall ergänzen wir nur den jeweils symmetrischen Operator. Da beide Operanden Variablen sind, macht es keinen Sinn jeweils \leq , \geq oder $<$, $>$ zu ersetzen.

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].

Sind durch die hinzugefügten Terme nun arithmetische Ausdrücke entstanden, die nur noch numerische Konstanten enthalten, so werden diese Ausdrücke ausgewertet.

Dieser Teilschritt erfordert weiterhin eine Sortierung der einzelnen Terme.

Beschränkung der Operatorenvielfalt

Ein weiterer Ansatz das Problem der äquivalenten Formeln anzugehen ist es, bestimmte Operatoren zu »verbieten«. Das soll bedeuten, wir definieren verbotene Operatoren, welche am Ende der Umwandlungen nicht mehr in der SQL-Anfrage vorkommen dürfen. Dies wird erreicht, indem wir jeden verbotenen Operator umwandeln in einen 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) Ausdrücke. 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	$\{ A > X, X < A, A \geq X + adjust(X), X \leq A - adjust(X) \}$	$A > X$

Im Folgenden soll ein Beispiel 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
```

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 passen wir komplexere arithmetische Ausdrücke an?”.

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:

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

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 einfacher. 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.

transitiv-implizierte Formeln

Formeln können auch transitiv-impliziert sein. Steht in der Musterlösung die Formel $A > B \text{ AND } B > C$ und der Student hat geschrieben $A > B \text{ AND } A > C$, 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. $A > B \text{ AND } B = 9$. Diese Formel impliziert eine weitere Formel transitiv, weil die Relationen $>$ und $=$ zueinander kompatibel sind. Inkompatible Relationen sind $>, \geq$ 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 $(A, 9) \in >$ 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 Tupelvariablen. Es geht darum zu erfassen welchen Wertebereich einzelne Tupelvariablen durch die Formeln in der SQL-Anfrage

zugewiesen bekommen. Dabei können semantische Widersprüche entdeckt werden, wie z. B. $A > 9 \text{ AND } A < 9$. 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. $A > 9 \text{ AND } A > 5$. 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.

Es bietet sich aus diesen Umständen eher an, das Hinzufügen von implizierten Gleichungen zu verwenden.

3.4.7 Sortierung

Im aktuell betrachteten Ansatz möchten wir zwei Anfragen dadurch vergleichen, dass wir sowohl die Musterlösung, als auch die Studentenlösung einer Standardisierung unterziehen. Ein ganz wesentlicher Aspekt dabei ist, die Art der Sortierung. Sind die ZQL-Parserbäume isomorph zueinander, dann lässt sich das leicht zeigen, in dem man beide nach gleichartigen Kriterien sortiert und dann einen direkten Abgleich vornimmt.

Dabei unterscheiden wir zwei Arten von Sortierung: Hat ein Operator als Operanden nur Ausdrücke und keine Konstanten oder Variablen dann sortieren wir die Kindknoten, welche jeweils wieder eigene Terme bilden.

Hat ein Operator als Operand mindestens eine Konstante oder Variable, so sortieren wir das innere dieses Terms.

Sortierung im Inneren der Terme

Hat ein Operator OPI als Kindknoten mindestens ein Blatt, dann werden die Kindknoten so sortiert, dass zunächst die Blattknoten (lexikographisch) und erst dann die Teilbäume erscheinen. Möglich wird dies, weil die Tabellen-Aliase in einem vorherigen Schritt bereits automatisch sortiert und benannt wurden. Bei symmetrischen Operatoren wie $=$, AND , OR können die Kindknoten einfach umgehangen/ umsortiert werden. Bei Operatoren wie \leq , \geq ist es notwendig den Operator OPI umzudrehen. Weil aber die Sortierung außerhalb von Termen auf den Operatoren basiert, ist es notwendig, die Sortierung im Inneren der Terme zuerst durchzuführen.

Dieser Schritt wurde bereits als Vorbereitung der Schritte »Hinzufügen von implizierten Formeln« und »Operatorbeschränkung« durchgeführt.

Sortierung von Termen

Hat ein Operator $OP1$ als Kindknoten nur weitere Operatoren $OP2, OP3$, dann muss anhand dieser Operatoren die Reihenfolge im Baum festgelegt werden. Dies geschieht, indem wir uns einfach eine Reihenfolge der Operatoren ausdenken. Wir überlegen uns folgende Ordnung $order : Relation \rightarrow \mathbb{N}$, in der eine Relation r vor einer Relation s im standardisierten Parserbaum erscheint, wenn $order(r) < order(s)$.

$order :$

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

Es sei $RT(OP)$ der Teilbaum des SQL-Ausdruckes mit der Wurzel OP . Wir bezeichnen mit $depth(RT(OP))$ die Tiefe des Baumes $RT(OP)$. Es seien $child(OP) = \{v_1, v_2, \dots, v_i, \dots, v_n\}$ die Kindknoten von OP . Die korrespondierenden Teilbäume $RT(v_1), RT(v_2), \dots, RT(v_i), \dots, RT(v_n)$ sollen nun wie folgt angeordnet werden: Der Teilbaum $RT(v_x)$ erscheint (bei einer fiktiven BFS) vor dem Teilbaum $RT(v_y)$ genau dann, wenn $depth(RT(v_x)) < depth(RT(v_y))$. Die Teilbäume werden also der Tiefe nach aufsteigend angeordnet.

Wie bereits im Abschnitt »Teilsortierung« angedeutet, werden Teilbäume mit gleicher Tiefe nicht sortiert. In diesem Fall erzeugen wir einen Alternativbaum, indem wir die zwei betreffenden Teilbäume vertauschen. Mit diesem Alternativbaum wird dann, parallel zum bisherigen Baum, weiter verarbeitet. Am Ende muss die Musterlösung auch gegen alle Alternativlösungen geprüft werden.

Eine weitere Alternative zur Behandlung von Teilbäumen mit gleicher Tiefe, ist die Sortierung Anhand der Blattknoten. Dazu sammeln wir in einer Tiefensuche die Werte der Blattknoten und hängen sie in einem String zusammen. Es seien also die Knoten $V_{DFS} = \{v_1, v_2, \dots, v_k\}$ die Knoten, die in einer Tiefensuche eines (Teil)baumes entstehen. Wir bezeichnen den Blattknotenstring eines gewurzelten Baumes, mit dem Operator O als Wurzel, mit:

$$leaf_string(RT(O)) = val(v_1) \oplus val(v_2) \oplus \dots \oplus val(v_k)$$

Dabei bezeichnen wir mit $val(v_k)$, den Wert eines Blattknotens.

$$val(v) = \begin{cases} \text{Variablenname,} & \text{wenn } v \text{ Variable,} \\ \text{Wert,} & \text{wenn } v \text{ eine Konstante.} \end{cases}$$

Aus den Teilbäumen mit gleicher Tiefe, werden solche Strings erzeugt und diese dann verglichen. Ist $leaf_string(RT(O_1)) < leaf_string(RT(O_2))$, so wird der Teilbaum $RT(O_1)$ als erstes Kind im Baum auftauchen.

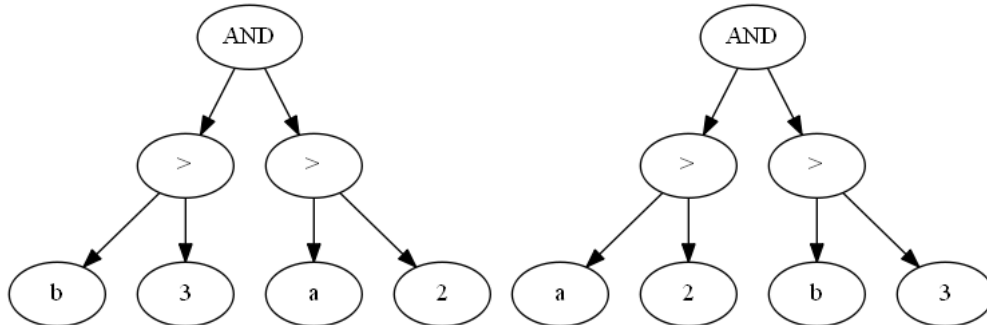


Abbildung 3.6: Beispiel von Bäumen mit gleicher Tiefe

Wir sehen in Abbildung 3.6, auf der linken Seite, zwei Teilbäume mit Wurzelknoten $>$. Wir bezeichnen den linken Teilbaum mit $RT(>_l)$ und den rechten mit $RT(>_r)$. Somit ergeben sich $leaf_string(RT(>_l)) = val(b) \oplus val(3) = 'b3'$ und $leaf_string(RT(>_r)) = val(a) \oplus val(2) = 'a2'$. Wegen $a2 < b3$ tauschen die Teilbäume ihre Position und ergeben das rechte Bild in Abbildung 3.6. Da wir diesem Schritt die Tupelvariablen bereits durch automatische Variablennamen vereinheitlicht haben, kann der gesamte Name des Attributs samt Präfix der Tupelvariable als $val()$ angesehen werden.

3.4.8 GROUP BY / HAVING

Bisher haben wir den SELECT-, FROM- und WHERE-Teil einer SQL-Anfrage bearbeitet und betrachtet. 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. Diese sind bereits automatisch benannt nach dem Schema aN mit $N \in \{1, 2, \dots\}$ (siehe Unterabschnitt 3.4.2). 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 ??.

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 ??) mit aufgenommen.

3.4.9 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 3.4.2 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.

3.4.10 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 Variablennamen wurden im Rest der Anfrage korrekt eingesetzt bzw. ersetzt. 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. Danach haben wir innere Verbunde (JOINS), die im FROM-Teil formuliert wurden, in den WHERE-Teil überführt, um anschließend unnötige äußere Verbunde und unnötige transitive, innere Verbunde zu eliminieren. Eine der letzten Schritte war das Behandeln der Vielfalt der Operatoren. Dem ging zunächst eine Teilsortierung des Parserbaumes voraus. Mit einer der beiden vorgestellten Methoden haben wir nun alle implizierten Formeln hinzugefügt, oder die Operatoren gemäß ihrer Repräsentanten beschränkt. Schlussendlich haben wir den gesamten WHERE-Teil sortiert, um eine Vereinheitlichung zu erreichen.

3.5 Anpassung durch elementare Transformationen

Der nun vorgestellte Ansatz stellt eine Alternative zur »Anpassung durch Standardisierung« dar. Beide Ansätze haben ähnliche Ideen, sind jedoch in ihrer Umsetzung verschieden. Beim Ansatz der Standardisierung haben wir zunächst eine SQL-Anfrage genommen und standardisiert. Dies geschah völlig unabhängig zu der zu vergleichenden, zweiten SQL-Anfrage. Es wurden gewisse allgemeine Regeln aufgestellt und nach diesen wurde jede SQL-Anfrage behandelt und angepasst. Der eigentliche Vergleich geschah dann aufgrund eines Vergleichs der standardisierten Anfragen und dem Vergleich von Metadaten (Anzahl Joins, Anzahl atomarer Formeln, etc.). Ein Nachteil von diesem Ansatz ist, dass wir für jedes mögliche Konstrukt in der SQL-Anfrage auch Regeln im

System haben, damit wir jene Konstrukte auch anpassen und standardisieren können. Außerdem findet der Vergleich erst im allerletzten Schritt statt.

Die »Anpassung durch elementare Transformationen« stellt einen intuitiven Ansatz vor. Wir versuchen hier per Backtracking konkret eine Lösung der anderen anzugleichen. Damit haben wir einen konkreten Unterschied zum bisherigen Ansatz. Wir vergleichen die Lösung direkt und umgehen die Vorverarbeitungen. Im Wesentlichen geben wir also ganz allgemeine Regeln an und in jedem Schritt versuchen wir durch Anwendung dieser Regeln die SQL-Anfrage so anzupassen, dass sie auf unsere zu vergleichende Anfrage passt.

3.5.1 Backtracking

[Dieser Abschnitt ist experimentell]

Für das Backtracking an sich bietet sich die Programmiersprache PROLOG an. Das liegt daran, dass unsere allgemeine Regeln einfach formuliert werden können und bereits den Großteil des Programms stellen. Außerdem benutzt PROLOG die SLD-Resolution, um logische Anfragen auswerten zu können und das ist bereits ein Backtracking. Da wir in unserer Arbeit allerdings die Programmiersprache JAVA gewählt haben, können wir nicht auf PROLOG zurückgreifen. Viel mehr müssen wir das Backtracking selbst implementieren.

Wir operieren weiterhin auf Parserbäumen, die so entstehen wie im Abschnitt »Funktionsweise des Parsers« erklärt wird. Es seien unsere zwei zu vergleichenden SQL-Anfragen mit *Query1* und *Query2* gegeben. Die dazugehörigen Parserbäume bezeichnen wir mit $RT(Query1)$ und $RT(Query2)$. Wir versuchen im Folgenden *Query1* an *Query2* anzupassen.

Um nicht zu viele Regeln definieren zu müssen, werden wir Teile der Vorverarbeitung des ersten Ansatzes wieder verwenden. Wir formen beide Anfragen zunächst in die konjunktive Normalform um. Danach gehen wir nach dem folgendem Prinzip vor.

Es sei $comm = \{+, *, AND, OR, =\}$, die Menge aller kommutativen Operatoren. Wir gehen den Vergleich zweier (Teil-)Bäume rekursiv an, das bedeutet unser Algorithmus erhält stets zwei (Teil-)Bäume in Form von zwei Wurzelknoten. Als Ergebnis gibt der Algorithmus dann an, ob diese zwei (Teil-)Bäume unifiziert werden konnten. Im Folgenden wird der Algorithmus textuell beschrieben. Ein detaillierter Pseudocode ist im Anhang unter Abschnitt 8.1 zu finden.

TODO Algorithmus in Worten beschreiben: Zum einen haben wir kommutative Operatoren wie $\{+, *, AND, OR, =\}$. Möchten wir zwei (Teil-)Bäume vergleichen, die einen solchen Operator als Wurzelknoten haben, so müssen wir uns im klaren sein, dass der gesuchte Operand an beliebiger Stelle auftauchen kann. Wir werden einen solchen Vergleich so angehen, dass wir uns den ersten Operanden aus Teilbaum 1 nehmen. Ist dieser Operand ein Ausdruck mit Operator o_1 , dann

suchen wir im zweiten Teilbaum nach einem Operanden, welcher auch ein Ausdruck ist mit Operator o_2 mit $o_1 = o_2$. Ist dies gelungen, so wird verglichen, ob die Teilbäume mit der Wurzel o_1 und o_2 gleich sind. Dies geschieht dann rekursiv. Ist dies erfolgreich, so streichen wir aus den Ursprungsbäumen die Teilbäume mit Wurzel o_1 und o_2 und fahren mit dem nächsten Operanden aus dem ersten Teilbaum fort.

Operatoren, die nicht kommutativ sind, sind meistens Vergleichsoperatoren wie \leq , $<$, $>$, \geq . Haben wir zwei Teilbäume mit einem solchen Operator, so ist der Vergleich sehr einfach, denn die jeweiligen Teilbäume unter den Operatoren müssen absolut identisch sein. Dies wird wieder mittels Rekursion geprüft.

Haben wir im ersten Baum einen Vergleichsoperator ausgewählt und können diesen im zweiten Baum nicht finden, so müssen wir auch nach dem umgedrehten Operator in Baum 2 suchen. Finden wir einen solchen, dann vergleichen wir die zugehörigen zwei Teilbäume weiter rekursiv. Zu beachten ist dabei, dass sich die Operandenreihenfolge vom ersten Teilbaum auch umdreht.

Weiterhin sei daran erinnert, dass arithmetische Ausdrücke in unsere Betrachtung nicht mehr als zwei Operanden im komplexen Teil beinhalten können. Siehe dazu Unterabschnitt 3.4.6.

3.6 Weitere Betrachtungen

Unabhängig von den bereits vorgestellten Ansätzen der »Standardisierung« und der »Anpassung durch elementare Transformationen« gibt es einige Umwandlungen, die entweder davor oder danach geschehen sollten. Diese Umwandlungen sollen dazu dienen dem Studenten ein Feedback zu geben. Das bedeutet, dass die Anfrage des Studenten richtig sein kann, allerdings unnötige oder unschöne Konstrukte enthält, welche die Anfrage unnötig kompliziert oder komplex machen.

Folgende verschiedene Komplexitätseinstufungen sollen eingeführt werden und auf jede Studentenanfrage angewendet werden.

3.6.1 Anzahl atomarer Formeln

Die Studentenanfrage enthält 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 Studenten mitgeteilt werden, dass er redundante Formeln eingebaut hat, welche die Lösung unnötig verkomplizieren.

3.6.2 Anzahl der Operatorkompressionen

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.

3.6.3 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.

3.6.4 Algorithmus aus [12]

Es sei unsere SQL-Anfrage der Form:

```
SELECT  $t_1, \dots, t_k$   
FROM  $R_1 X_1, \dots, R_n X_n$   
WHERE  $\varphi$ 
```

Es sei $X = \{X_1, \dots, X_n\}$ die Menge aller Tupelvariablen. Es sei $G = \{G_1, \dots, G_m\}$ 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 $\mathcal{K} = \{t_1, \dots, t_k\}$.

$\mathcal{K} = \mathcal{K} \cup A$, wenn $A = c \in \text{WHERE-Bedingung}$

do

$\mathcal{K}' = \mathcal{K}$

$\mathcal{K}' = \mathcal{K}' \cup A$, wenn $A = B \in \text{WHERE-Bedingung}$ 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 Anfrage hat GROUP BY Statement:

foreach $x \in X$ do

if not ($\exists k \in \mathcal{K}'$ mit k ist Schluessel von x)

break and answer NO

endif

done

if not Anfrage hat GROUP BY Statement:

foreach $g \in G$ do

if $g \notin \mathcal{K}'$

break and answer NO

endif

done

answer YES

4 Verwendete Software

4.1 SQL-Parser

4.1.1 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- und SET TRANSACTION-Ausdrücke. Wichtig für diese Arbeit sind dabei insbesondere SELECT- und UPDATE-Ausdrücke, sowie – die leider nicht enthaltenen – CREATE TABLE-Ausdrücke.

4.1.2 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 COLUMNNAME, NULL, NUMBER, STRING oder UNKNOWN

- **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 Tabelle. Gespeichert wird in dieser Klasse daher der *Name der Tabelle* und ein möglicher *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, die von **ZExp** erben muss. 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. Was ZQL für gültige Operatoren kennt wird im Anhang aufgeführt. 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))
```

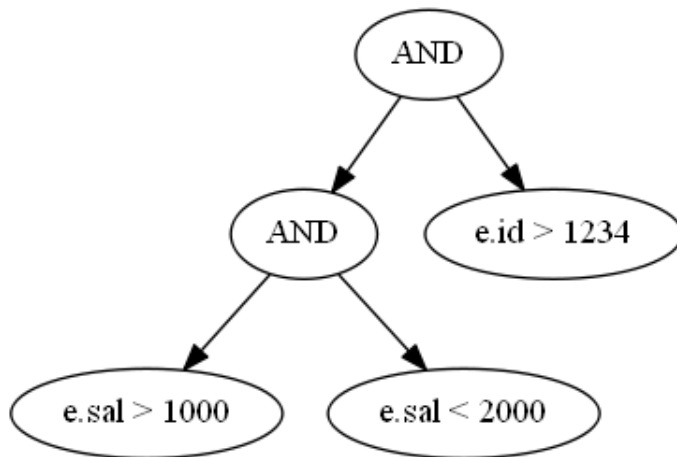


Abbildung 4.1: WHERE-Bedingung in üblichen Syntaxbäumen

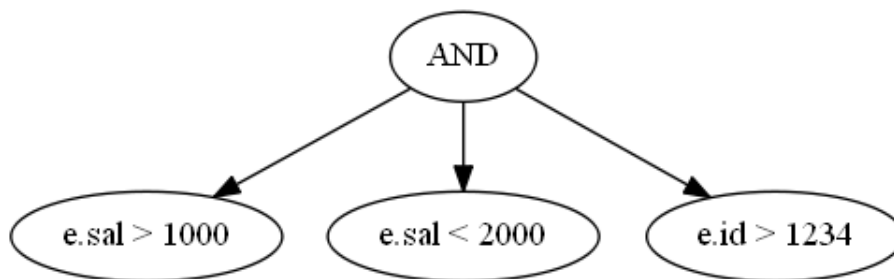


Abbildung 4.2: WHERE-Bedingung geparkt mit ZQL

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: COLUMNNAME, NULL, NUMBER, STRING, UNKNOWN. Da die Typen selbsterklärend sind, wird auf eine detaillierte Beschreibung verzichtet.

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 Beispiels 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 geparkt 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 *ZFormItem* 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.

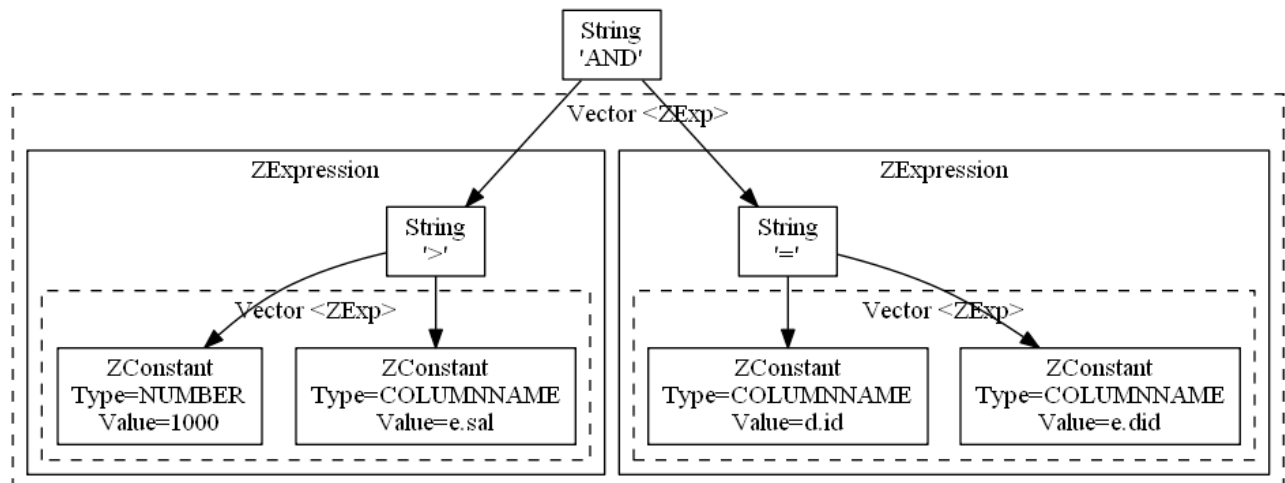


Abbildung 4.3: Geparster Baum der WHERE Bedingung des Beispiels

In Abbildung 4.3 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 Kästen konkrete Objekte sein sollen.

4.1.3 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 geparkt werden können. Für die Arbeit ist es zunächst nur notwendig Name und Datentyp der Spalten in eine interne Datenstruktur zu überführen. Dabei wird nur zwischen Zahlen und Sonstigem (Text) unterschieden. Unser Programm soll in der Lage sein, einfache arithmetische Operationen durchzuführen. Dazu ist das Wissen um Datentypen der Variablen von Nöten.

Weiterhin ist der *ZQL*-Parser nicht in der Lage *JOINS* über die Schlüsselworte *ON* [*LEFT OUTER*|*RIGHT OUTER*|*INNER*] *JOIN* zu realisieren. Der Parser erkennt nur innere *JOINS*, die im *WHERE*-Teil formuliert worden. Das soll für diese Arbeit ohne weitere Bedeutung sein, da dennoch Strategien entwickelt werden, wie man mit derartigen *JOINS* umgeht. Es muss an der Stelle nur erwähnt werden, dass das Programm, welches im Rahmen dieser Arbeit entsteht, mit derartigen *JOINS* nicht umgehen kann.

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

4.2 Java Server Pages

4.2.1 Überblick

4.2.2 Einbettung in JSP

4.2.3 Log

5 Praktische Umsetzung

5.1 Anforderungen

Das zu entwickelnde Programm wird online per Java Servlets zur Verfügung gestellt. Als Webserver wird der Tomcat eingesetzt. Folgende Funktionalitäten sollen umgesetzt werden:

- Loginsystem für Lernenden und Betreuer
- Grafische Oberfläche
- Eintragen von neuen Übungsaufgaben bestehend aus:
 - Sachtext /Aufgabenstellung
 - Musterlösung(en)
 - Angabe von realen Datenbanken mit Testdaten
 - Einteilung in Kategorie
- Löschen und Bearbeiten von Übungsaufgaben
- Erstellen, Bearbeiten und Löschen von Kategorien
- Lösen von Übungsaufgaben durch Angabe einer SQL-Anfrage
- Tracking von Lösungsversuchen (Versuch, AufgabenID, Timestamp, UserID)
- Tracking von Usern und letzten Aktivitäten
- Tracking von Fehlern/ Fehlversuchen pro Aufgabe/User

5.2 Funktionsumfang

6 Ergebnisse

7 Ausblick

Parser mit JOIN unter FROM Unterabfragen unter FROM generell FROM einschränkung

komplexe aritmetische Ausdrücke

mehr Semantic checks wertebereiche , siehe brass paper anfragen die durch rtschen

Literaturverzeichnis

- [1] <http://zql.sourceforge.net>. 46
- [2] <http://www.javacc.org>. 46
- [3] A. Mitrovic: *Learning 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. 10
- [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, 18
- [9] S. Brass, C. Goldberg: *Detecting Logical Errors in SQL Queries* in: 16th Workshop On Foundations Of Databases, Monheim, Germany, 2004. 15, 16, 18, 44
- [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. 29
- [12] S. Brass: *Vorlesung: Datenbanken I* an der Martin-Luther-Universität Halle-Wittenberg, 2011, Page 5-131. 3, 44

8 Anhang

8.1 Algorithmus für Backtracking

```
Node = { op, childs[] };
Konstanten habe eine leere childs Liste und stehen an der Stelle des op
vergleiche(node1:Node, node2:Node)

if node1.op == node2.op and isempty(node1.childs)
and isempty(node2.childs)
    //Wir haben uebereinstimmende Konstanten gefunden
    return true
endif

if node1.op != node2.op
    return false
/* Wenn beide Wurzelknoten bereits unterschiedliche Operatoren sind,
* dann koennen die entsprechenden Teilbaeume nicht uebereinstimmen */

if node1.op ∈ comm
/* der Operator in node1 ist kommutativ.
* Demzufolge koennen die Kinder von Baum1 und Baum 2
* in beliebiger Reihenfolge auftauchen. */

forall c1 in node1.childs do
    posList[] = getPosFromIn(c1,node2.childs)

    forall pos in posList do
        result = vergleiche(c1,node2.childs[pos]);
        if result = true
            streiche c1 aus Baum1 und node2.childs[pos] aus Baum2
            break
        endif
    done

if isEmpty(posList) and hasOpposite(c1)
/* wir haben c1 nicht in Baum2 gefunden und suchen jetzt
* nach seinem opposite */

    oppositePosList[] = getPosFromIn(opposite(c1),node2.childs);
```

```

        forall pos in oppositePosList do
            //reverse node2.childs[pos].childs
            result = vergleiche(c1, opposite(node2.childs[pos]));
            if result = true
                streiche c1 aus Baum1 und node2.childs[pos] aus Baum2
                break
            endif
        done
    endif
done

/* wenn wir an diese Stelle gelangen, wurden alle Teilbaeume von
 * node1 und node2 miteinander verglichen.
 * Wenn diese alle gematcht wurden, dann haben die Baeume1
 * und 2 keine Kindknoten mehr */

if isEmpty(node1.childs) and isEmpty(node2.childs)
    return true;
else
    return false;

else /*unsere Wurzelnoten node1 und node2 sind nicht kommutativ*/
    /* Dieser Fall ist einfacher, da die Operanden dieser Teilbaeume
     * nun in der gleichen Reihenfolge vorkommen muessen */

    for i = 0 to node1.childs.length - 1 do
        if not vergleiche(node1.childs[i], node2.childs[i]) then
            return false;
        endif
    done
    /* An dieser Stelle waren alle Kindknoten beider Baeume gleich,
     * und damit sind die Teilbaeume von node1 und node2 identisch*/
    return true;
endif

```

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