Masterarbeit

Vergleich von SQL-Anfragen Theorie und Implementierung in Java

Robert Hartmann

Betreuer: Prof. Dr. Stefan Brass

28. August 2013

Martin-Luther-Universität Halle-Wittenberg

Inhaltsverzeichnis

1 Einleitung / Motivation

SQL (structured query language) ist eine Datenbanksprache, die in relationalen Datenbanken zum Definieren, Ändern und Abfragen von Datenbeständen benutzt wird. Basierend auf relationaler Algebra und dem Tupelkalkül, ist sie einfach aufgebaut und ähnelt der englischen Sprache sehr, was Anfragen deutlich verständlicher gestaltet. SQL ist der Standard in der Industrie was DBMS angeht. 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 dennoch ähnlich andere gleiche Anfragen kann man nur nach Umformen oder umschreiben 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 oder von freien Internetdatenbanken beschafft werden. Dies kostet Zeit und Arbeitskraft, welche im universitärem Umfeld meist beschränkt ist. So haben Hochschulen immer weniger Geld für Tutoren oder Hilfskräfte, was die Zeit der wenigen Mitarbeiter umso wertvoller macht.

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

Lernenden Übungsaufgaben gestellt, die dieser dann innerhalb einer Frist bearbeitet und abgibt. Diese müssen dann kontrolliert und wieder ausgehändigt werden. Bei diesem Prozess kann nur schwer auf die einzelnen Fehler der Studenten eingegangen werden. Auch ist es ein zusätzlicher Zeitaufwand herauszufinden, welche Fehler besonders häufig auftreten. Des weiteren sind manche Lernende auch gewillt mehr zu üben um sich gerüstet für eine Klausur zu fühlen oder Lernende 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 möglich sein für den Mitarbeiter der Universität 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. Da die Fehlermeldung des Standardparser 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. 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), sowie java-servlets als Umsetzung dieser Anforderung an.

Ein mögliches Haupteinsatzgebiet ist die Lehre, so wie die Untersuchung des Lernfortschritts von Studenten oder anderen Interessierten, die den Einsatz von SQL erlernen möchten. 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 es ist 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 so 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 im sich im Unternehmen befinden. So könnte man bei einer geplanten Umstrukturierung oder Erzeugung von Datenbanken bereits Anfragen prüfen und vergleiche bevor man sich u.U. teure Testdaten kauft oder Daten migrieren muss.

1.3 Aufbau der Arbeit

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

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 **??** genauer erläutert.

Das Problem “Sind zwei SQL-Anfragen äquivalent” ist nicht entscheidbar. Aus diesem Grund klären wir im Kapitel 3 wie wir den Entscheidungsprozess angehen wollen, so dass 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 vorgstellt, 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.

TODO: Ergebnisse und Ausblick (zusammenfassen)

1.4 Produkt

Es wurde im Rahmen der Aufgabenstellung ein Programm entwickelt, welches 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 vom 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 Kapitel [Komplexitätsmaße].

Schlägt der erste Schritt fehl, wird die Anfrage des Lernenden auf der angegebene Datenbank verarbeitet und mit den Ergebnistupeln verglichen, die die Musterlösung liefert. Sind beide in allen Beispielzuständen gleich, so wird dem Dozenten gemeldet, dass eine eventuelle neue Musterlösung gefunden wurde, die strukturell so unterschiedlich ist, 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 Mensch – in Form des Dozenten – solche Lösungen noch einmal prüfen.

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 so wie mögliche Hinweise, was der Lernende falsch gemacht haben könnte.

2 Forschungsstand und Einordnung

2.1 Einleitung [in das Chapter]

Die Idee SQL-Anfragen von Schülern/Lernenden auszuwerten ist nicht völlig neu. Weil eine Auswertung über den Standard SQL-Parser nicht sehr umfangreich ist, und bei semantischen Fehlern gar 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 zum Beispiel 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 Studenten um ihn mit passenderen Aufgaben zu konfrontieren, damit er weder gelangweilt noch überfordert ist. [Anmerkung: Ähnlich einem Art Matchmaking System].

In diesem Abschnitt möchten wir die bereits existierenden Ansätze auf dem Gebiet kurz betrachten um dann diese Arbeit davon abzugrenzen bzw. diese 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 `SELECT`, `FROM`, `WHERE`, `ORDER BY`, `GROUP BY` sowie `HAVING` Anteile 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, bei einer falschen Eingabe, was genau falsch 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 versuch der SQL-Tutor Konstrukte wie numerische Vergleiche mit gewissen Operatoren in der Lösung des Studenten zu finden, wenn diese in der Musterlösung auftauchen. Auch komplexere Constraints, die sicherstellen, dass bei einem numerischen Vergleich `a > 1` das gleiche ist wie `a >= 0` sind vorhanden.

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

Der SQL-Tutor lässt außerdem auch 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 Grundgedanken dieser Arbeit. Ein Grundpfeiler des SQL-Tutors ist es, dem Studenten detailliertes Feedback zu geben über seine semantischen und syntaktischen Fehler. Das Programm was im Zuge dieser Arbeit entsteht soll weniger semantische Fehler analysieren, als viel mehr versuchen zwei SQL-Anfragen zu vergleichen und zwar egal 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. Damit entfällt für Lehrkräfte ein aufwendiges Ausdenken oder Besorgen von Testdaten. Des weiteren kann es bei ungünstig gewählten Testdaten passieren, dass der Eindruck ensteht zwei Anfragen wären gleich weil sie auf den Testdaten die gleichen Tupel zurück lieferten, auf anderen Testdaten würden aber Unterschiede aufgezeigt werden.

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 in 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 der Anfrage 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 zum Beispiel 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 mit 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 Vergleich der zurückgelieferten Tupel der Muster- und Studentenanfrage. Hierbei kann wieder das Problem auftreten, dass für beide Anfragen für die erzeugten Testdaten die gleichen Tupel zurückliefern, es bei einem anderen – nicht erzeugtem – 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 diese keine Testdaten erzeugen möchte, sondern gänzlich ohne Daten auskommen will.

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 möchte. 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 Stück für Stück 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, so wie 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, in dem 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, jedoch verfolgt diese Arbeit ein ganz 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 veringern.

Ein nicht unwesentlicher Ansatz des Projektes ist es, solche Fehlermeldungen in der Lehre einzusetzen. In [6] wird auch deutlich gemacht, das eine Motivation dieses Projektes aus typischen Fehlern von Studenten entspringt. So wurde im selben Artikel aufgeführt, dass semantische Fehler bei Lernenden am häufigsten auftreten. Unter den drei häufigsten semantischen Fehlern befinden

sich: fehlende JOIN Bedingung, (zu) viele Duplikate, unnötiger JOIN. Diese Fehler machen bereits ca. 37% 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. Wir bereits erwähnt, ist das Probleme 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, so dass 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 hier von »opposite operator«. Die Menge $O = \{\{\leq, >\}, \{\geq, <\}, \{=, =\}, \{\neq, \neq\}\}$ 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: $\phi_1 \vee ... \vee \phi_n$. 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 $\leq$ gelabelt sind. Dann werden Intervalle von möglichen Werten für jeden Knoten berechnet. Dabei ist zu beachten, dass die SQL-Datentypen, wie `NUMERIC(1)`, das Intervall zusätzliche 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`,`>=ALL`, etc.) können auf die `EXISTS` Unteranfrage reduziert werden. Oracle führt solche Umwandlungen durch, bevor der Optimierer beginnt an der Anfrage zu arbeiten.

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

2.5.4 Unnötige logische Komplikationen

Es kann vorkommen, dass eine Unterbedingung (?subcondition?) 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 wird. Als Beispiel können wir etwas angeben wie `A IS NOT NULL`. Diese Bedingung wird unnötig, wenn wir wissen, dass `A` bereits als `NOT NULL` definiert ist.

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

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

1. $\neg(C_1 \vee ... \vee C_m)$ - Die Negation der gesamten Formel. Ansonsten könnte man diese durch »true« ersetzen

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

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

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

2.5.5 Laufzeitfehler

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

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

Zusammenhang zu dieser Arbeit

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

Auch ist die gesamte Arbeit von SQLLint hilfreich für diese Arbeit. So könnte man in späteren Ausbaustadien des Programmes, welches im Rahmen dieser Arbeit entsteht, die Funktionalitäten des SQLLint einbauen. Dies würde die Art des Feedbacks an 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 zunächst 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 korrekt sind. Interessanter sind daher Anfragen, die zunächst nicht syntaktisch dekungsgleich sind.

Ein Ansatz besteht darin beide SQL-Anfragen einer Standardisierung zu unterziehen. Wie genau so etwas durchgeführt werden kann, wird im Folgenden noch erläutert. Wir würden dann 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 birgt Vorteile aber auch Probleme mit sich, die im Folgenden besprochen werden.

3.1 Hintergrund

Es gibt syntaktisch unterschiedliche Anfragen, die jedoch semantisch äquivalent sind. So liefern die folgenden Anfragen die gleichen Ergebnisse, sind aber nicht syntaktisch äquivalent.

```
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 gar nicht mehr auftreten kann. Das Programm, was zu dieser Arbeit entwickelt wird, kann mit solchen redundanten Eigenschaften nicht umgehen. Es wäre auch mehr ein Problem für einen "semantic checker", da es hier gar nicht auf zwei verschiedene Anfragen ankommt. Hier ist bereits diese eine Anfrage in sich selbst zu kompliziert. Mit derlei Problemen beschäftigt sich das Projekt "SQLLint" 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 dann nie wissen, in welcher Art und Weise der Lernende die Aufgabe formulieren wird. Man betrachte sich 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, in dem wir bestimmte Operatoren wie BETWEEN abschaffen und durch die äquivalenten Ungleichungen mit >= und <= ersetzen. Ähnliches gibt es für INNER JOIN im FROM Teil, mit Ersetzung durch Vergleiche im WHERE Teil.

3.2 Workflow

```
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 querys 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 soltion contained unnecassary joins, or formulas. */
```

3.3 Preprocessing

Im Abschitt »Forschungsstand« haben wir bereits einige Lernplattformen/projekte zum Thema SQL kennen gelernt. Viele dieser Plattformen möchten dem Lernenden genügend Feedback beim Lernprozess geben. Dies ist nicht nur sinnvoll, damit der Student schneller auf korrekte Lösungen stößt, sonder auch, weil die Standardhinweise eines SQL Systems meist nur auf syntaktische Fehler hinweisen. Einen großen Beitrag zur Verbesserung von Fehlermeldungen hat das Projekt SQLLint vorzuweisen, da es Fehlermeldungen und Hinweise konkreterer Natur ausgibt. Hervorzuheben ist, noch einmal, dass es sich hierbei um semantische Fehlermeldungen handelt. Schon nach kurzer Einlernzeit sinken die Anzahl an syntaktischen Fehlern 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 Feedback geben, an dem er möglicherweise 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 Metainformationen der zwei Anfragen verglichen werden und dem Lernenden soll 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 in `WHERE`-Teil
- Anzahl atomarer Formeln in `HAVING`-Teil
- Anzahl Tabellen in `FROM`-Teil
- Anzahl Attribute im `SELECT`-Teil *
- existiert ein `DISTINCT`
- existiert ein `GROUP BY` *
  - wenn ja, stimmen die Attribute überein?
- 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 Lerneden 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. Reconsidder 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 falscchen 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 ja trotz Korrektheit zu lang bzw. komnpliziert 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 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 Vereinheitlichen der FROM Klausel

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` und `WHERE` die Aliase ebenfalls korrekt ersetzt werden.

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

Hatten die vorkommenden Tabellen im `FROM` Teil keinen Alias wird nur der küsntliche Alias eingeführt.

3.4.3 Umwandlung der WHERE Bedingung in KNF

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

Es ist daher wünschenswert, wenn ein Operator X einen Ausdruck als Kindknoten besitzt, in dem X ebenfalls der Operator ist, den Operator X im Kindknoten zu eliminieren und alle Kinder vom eliminierten Kindknoten an den verbleibenden Operatorknoten X zu hängen. Damit hätte man den Ausdruck vereinfacht, da die assoziative Klammerung wegfällt. Wir nennen dieses Vorgehen im Folgenden Operatorkompression.

Gegeben sei der ZQL-Parsebaum $B = (V, E)$. 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 $child(v) = child(v) \cup child(w)$. $E = E \backslash \{(w, x) : x \in child(w)\} \cup \{(v, x) x \in child(w)\}$ und $V = V \backslash \{w\}$.

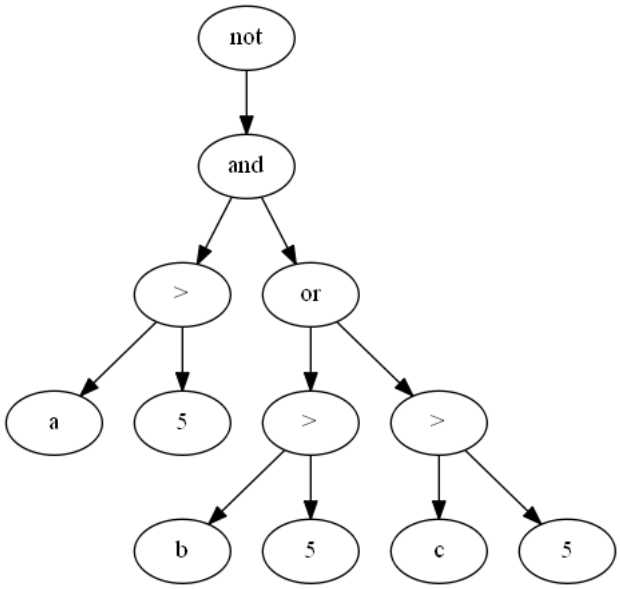
Im Sinne des Vergleiches der Komplexität der Musterlösung mit der Komplexität der Lösung des Lernenden ist es sinnvoll zu speichern, ob und wie oft eine solche Operatorkompression durchgeführt werden musste.

NOT auflösen

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

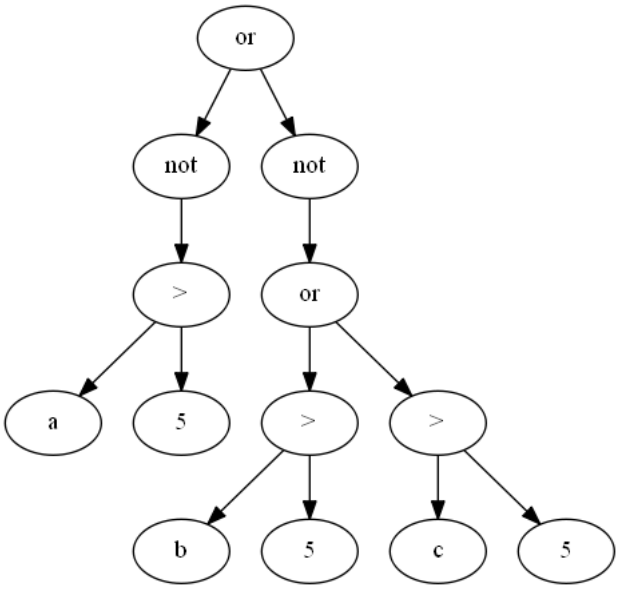
Eingabe:

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



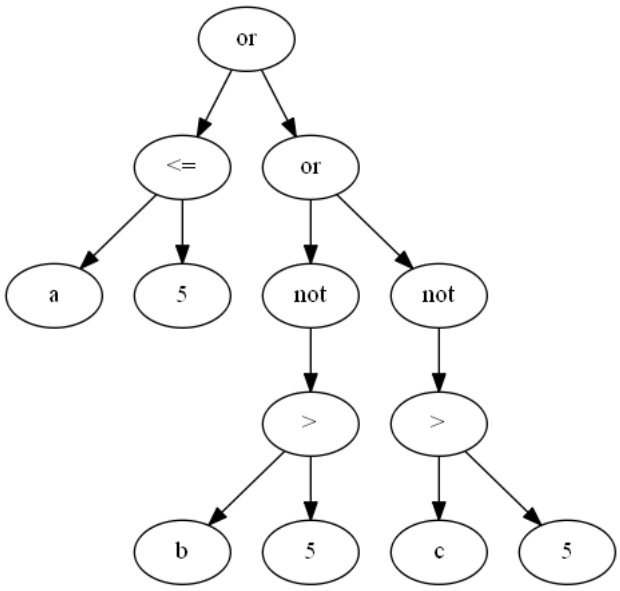
Umwandlung Teil 1:

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



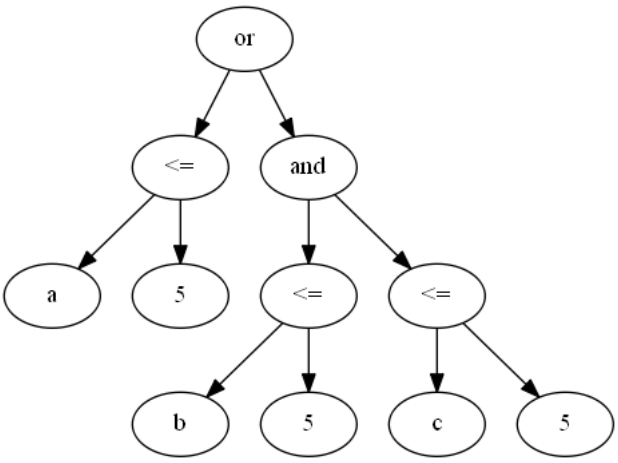
Umwandlung Teil 2:

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



Umwandlung Teil 3:

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



Anwenden des Distributivgesetzes

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

3.4.4 Ersetzung von syntaktischen Varianten

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

Unteranfragen

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

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 Unteranfrage, so wird dieses nicht zur Unterabfrage "durchgedrückt" sondern bleibt davor.

Ersetzen von IN

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

Wir wandeln IN-Unteranfragen wie die folgende:

t_1 `IN (SELECT` t_2
`FROM` R_1 X_1`, ...,` R_n X_n
`WHERE` φ`)`

unter – noch zu erläuternden – Vorrausetzungen um in:

`EXISTS (SELECT *`
`FROM` R_1 X_1`, ...,` R_n X_n
`WHERE (`φ`) AND` $t_1 = t_2$`)`

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 ja 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 einführt.
- Die Unteranfrage für t_2 darf keine Nullwerte liefern.

andere Unteranfragen

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 3.4.5 sollen alle Studenten ausgegeben werden, die in einer Klausur die Note 3 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 3.4.5 äquivalent.

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

Abbildung 3.5: unnötiger OUTER 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 3 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 Fremdschlüsselattributen 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 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 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 3 Einträge haben. Für jeden Eintrag $e \in Aequivalenzklasse$ mit $e = T.A$ überprüfen wir, ob es andere Äquivalenklassen gibt, die nicht aus einem transitiven `JOIN` entstanden sind (also weniger als 3 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 3 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 wiederm gestrichen werden, da es wieder nicht in einer anderen Äquivalenklasse 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 3 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 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 Aequivalenklassen E_i packen.
foreach e ∈ E_i mit |e| ≥ 3 do
  foreach t.a ∈ e do
    if |e| ≥ 3 and t ∉SELECT and
    ∄b ∈ E_i mit |b| < 3 und b=t.*
      e = e - {t.a}
    end
  done
done

Streiche unnoetige 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 zum Beispiel: `A > 5` ist dieser

Ausdruck äquivalent mit `5 < A`. Wenn wir wissen, dass `A` ein ganzzahlige Variable ist, dann sind auch folgende Äquivalenzen wahr: `A >= 6` so wie `6 <= A`. Wir betrachten nun zwei verschiedene Ansätze um mit diesem Problem umzugehen. Ein 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.

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

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

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

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

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

Finden wir Ausdrücke mit $>, <, \leq, \geq$, welche als Argumente Variablen oder Konstanten haben, so unterscheiden wir also grundsätzlich 3 Fälle.

Fall 1 (M_5, M_6): Beide Operanden sind Variablen oder Konstanten. In diesem Fall ergänzen wir nur den jeweils symetrischen 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 Vorgestelltem. Wir betrachten uns wieder die Mengen M_i. Des Weiteren hat jede Menge M_i einen Repräsentanten $r(M_i)$. Finden wir nun in der zu bearbeitenden Anfrage eine Formel f, die auf eine der Ausdrücke $a \in M_i$ passt, so ersetzen wir f mit $r(M_i)$. Folgende Tabelle soll die Mengen und deren Repräsentanten beschreiben.

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

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

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
```

Diskussion der beiden Ansätze

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

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 ja 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 *OP1* 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 *OP1* 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 festegelegt werden. Dies geschieht, in dem 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 $child(OP) = \{v_1, v_2, ..., v_i, ..., v_n\}$ die Kindknoten von OP. Die korrespondierenden Teilbäume $RT(v_1), RT(v_2), ..., RT(v_i), ..., RT(v_n)$ 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 $V_{DFS} = \{v_1, v_2, ..., v_k\}$ die Knoten, die in einer Tiefensuche eines (Teil)baumes entstehen. Wir bezeichnen den Blattknotenstring eines gewurzelten Baumes, mit dem Operator O als Wurzel, mit:

$$leaf\_string(RT(O)) = val(v_1) \oplus val(v_2) \oplus ...val(v_k)$$

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.

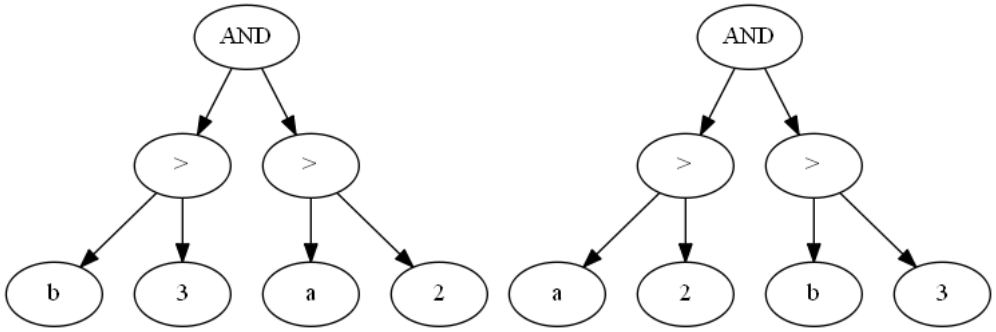


Abbildung 3.6: Beispiel von Bäumen mit gleicher Tiefe

Wir sehen in Abbildung 3.6, auf der linken Seite, einen zwei Teilbäume mit Wurzelknoten >. Wir bezeichen den linken Teilbaum mit $RT(>_l)$ und den rechten mit $RT(>_r)$. Somit ergeben sich $leaf\_string(RT(>_1)) = val(b) \oplus val(3) =' b3'$ und $leaf\_string(RT(>_2)) = val(b) \oplus val(3) =' a2'$. 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 Abschluss

Wir fassen die einzelnen Schritte noch einmal kurz zusammen. Zunächst haben wir den `FROM` Teil der Anfrage vereinheitlicht, in dem 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 syntatkische 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 (JOIN), 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

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, das 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 den 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 ist der Fall, so muss die Lösung des Lernenden dieses 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, ..., t_k
`FROM` R_1 X_1, ..., R_n X_n
`WHERE` φ

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

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

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

```
if Anfrage hat GROUP BY Statement:
  foreach x ∈ X do
    if not (∃k ∈ K' mit k ist Schluessel von x)
```

```
      break and answer NO
    endif
  done

if not Anfrage hat GROUP BY Statement:
  foreach g ∈ G do
    if g ∉ 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`.

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 `COLUMNAME, NULL, NUMBER, STRING` oder `UNKNOWN`
- `ZExpression` - Ein SQL-Ausdruck bestehend aus einem Operator und einen oder mehreren Operanden

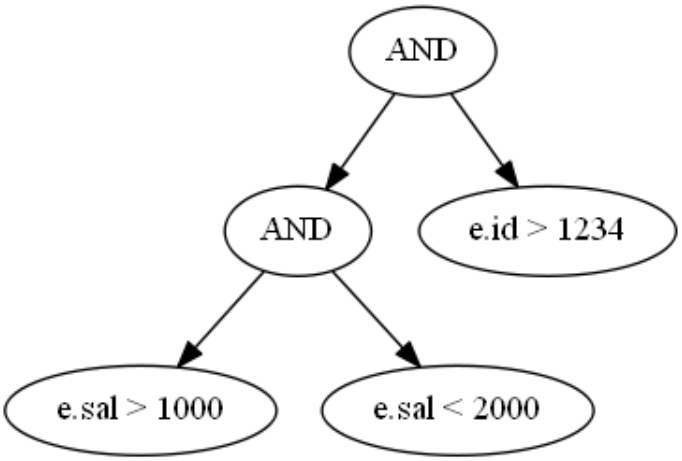


Abbildung 4.1: WHERE-Bedingung in üblichen Syntaxbäumen

- `ZQuery` - Eine `SELECT` Anfrage ist auch ein Ausdruck

Da die `SELECT` Anfragen die wohl am häufigsten gebrauchte Form der Anfragen ist, wird sich die Erklärung der Funktionsweise des Parsers beispielhaft auf diese Art der Anfragen beziehen. Wie die anderen Statements geparst werden ist dann analog schnell zu verstehen.

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

```
SELECT e.name FROM emp e WHERE e.sal > 1000 ORDER BY e.sal DESC
```

`SELECT e.name`	*Vector* von *ZSelectItem* enthält `e.name`
`FROM emp e`	*Vector* von *ZFromItem* enthält `emp` mit Alias `e`
`WHERE e.sal > 1000`	*ZExpression* mit Operator > und Operanden*Vector* der Form {`e.sal`, `1000`}
`ORDER BY e.sal DESC`	*ZOrderBy*-Objekt mit enthaltenem `ORDER BY` Sortierausdruck und Reihenfolge

Eine Besonderheit des ZQL-Parsers sind seine Parserbäume. Ein üblicher Syntaxbaum ist binär, wobei die Wurzel den Operator mit der höchsten Priorität darstellt. Alle Teilbäume sind wieder 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:

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

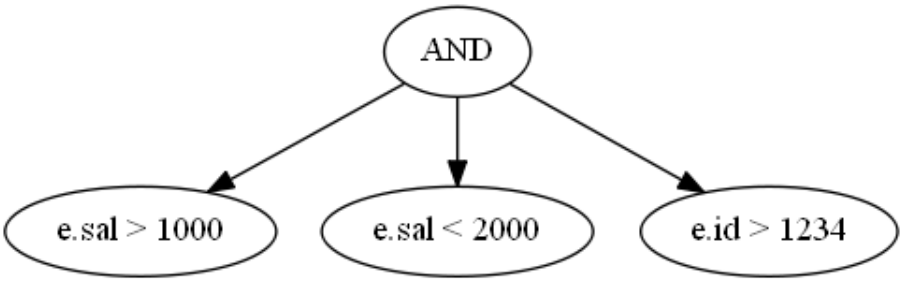


Abbildung 4.2: WHERE-Bedingung geparst mit ZQL

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))
```

Wie schon erwähnt werden Operanden daher in einer *Vector* Struktur gespeichert.

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 geparst 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

6 Ergebnisse

7 Ausblick

Literaturverzeichnis

[1] `http://zql.sourceforge.net`. 40

[2] `http://www.javacc.org`. 40

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

[4] P. Brusilovsky, S. Sosnovsky, D. H. Lee et al: *An open integrated exploratorium for database courses* in: ACM SIGCSE Bulletin ITiCSE 08, Volume 40 Issue 3, September 2008, Pages 22–26. 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, 37

[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. 39

[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. 27

[12] S. Brass: *Vorlesung: Datenbanken I* an der Martin-Luther-Universität Halle-Wittenberg, 2011, Page 5-131. 2, 38

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