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 **4**
- 1.1 Motivation . . . 4
- 1.2 Aufgabenstellung . . . 5
- 1.3 Produkt . . . 6

2 Forschungsstand und Einordnung **8**
- 2.1 Einleitung [in das Chapter] . . . 8
- 2.2 SQL-Tutor . . . 8
- 2.3 SQL-Exploratorium . . . 9
  - 2.3.1 Interactive Examples . . . 10
  - 2.3.2 SQL Knowledge Tester . . . 10
  - 2.3.3 Weiteres . . . 11
- 2.4 WIN-RBDI . . . 11
- 2.5 SQLLint . . . 12

3 Theoretische Betrachtungen **13**
- 3.1 Hintergrund . . . 13
- 3.2 Standardisierung von SQL-Anfragen . . . 14
  - 3.2.1 Entfernen von syntaktischen Details . . . 14
  - 3.2.2 Vereinheitlichen der FROM Klausel . . . 14
  - 3.2.3 Umwandlung der WHERE Bedingung in KNF . . . 15
  - 3.2.4 Ersetzung von syntaktischen Varianten . . . 18
  - 3.2.5 Hinzufügen einfacher Implikationen (transitiv) . . . 18
  - 3.2.6 Beschränkung der Operatorenvielfalt . . . 19
  - 3.2.7 Sortierung . . . 19
- 3.3 Anpassung durch elementare Transformationen . . . 20
- 3.4 weitere Betrachtungen . . . 20
  - 3.4.1 Anzahl atomarer Formeln . . . 20
  - 3.4.2 Anzahl der Operatorkompressionen . . . 21
  - 3.4.3 unnötiges DISTINCT . . . 21
  - 3.4.4 unnötiger JOIN . . . 21

4 Verwendete Software **22**
- 4.1 SQL Parser . . . 22
  - 4.1.1 über den SQL Parser: ZQL . . . 22
  - 4.1.2 Funktionsweise des Parsers . . . 22
  - 4.1.3 Grenzen des Parsers . . . 24
- 4.2 Java Server Pages . . . 24
  - 4.2.1 Überblick . . . 24
  - 4.2.2 Einbettung in JSP . . . 24

4.2.3 Log 24

5 Praktische Umsetzung **25**

6 Ergebnisse **26**

7 Ausblick **27**

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 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 **??** 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 **??** 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 **??** 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

TODO: Projekt von Prof. Brass

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.

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

3.2 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.

3.2.1 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.

3.2.2 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.

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

3.2.3 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.

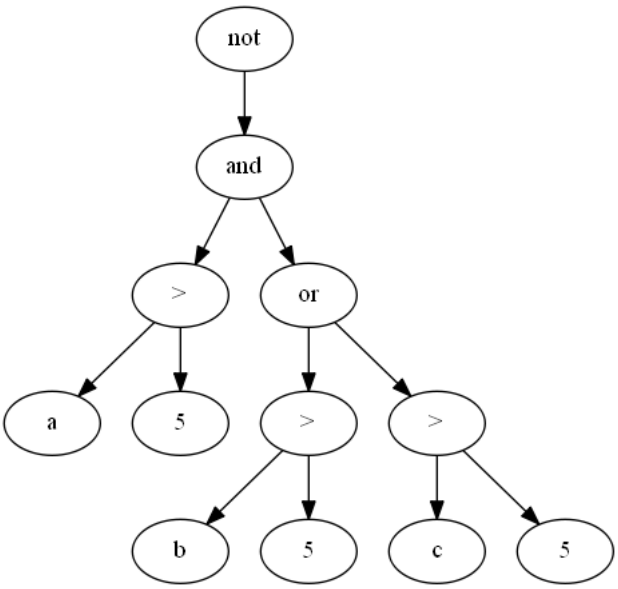
Gegeben sei der ZQL-Parsebaum $B = (V, E)$. Es sei $child(v) = \{w : w \in V \wedge (v, w) \in E\}$, also die Menge aller Kindknoten von v. Gibt es einen Knoten $w \in child(v)$ mit $v = w$, so wird Knoten w eliminiert und alle Kindknoten von w werden zu Kindknoten von v, also $child(v) = child(v) \cup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

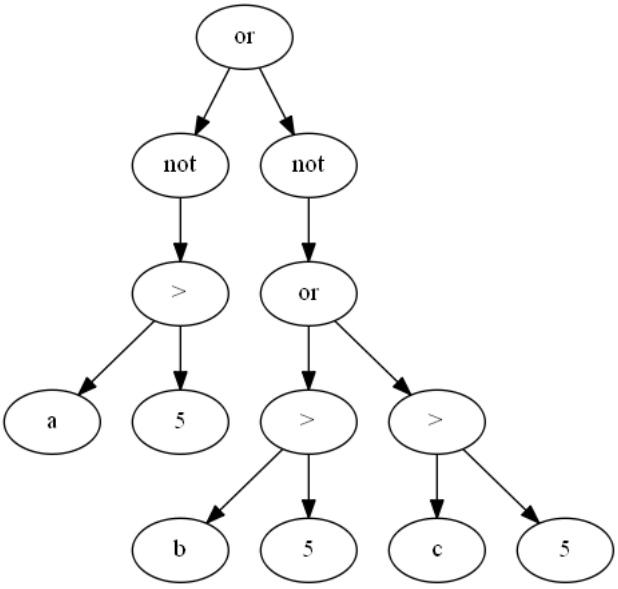
Im nächsten Schritt möchten wir auftretende `NOT` Operatoren entfernen. Dies geschieht indem der Operator `NOT` im Parserbaum nach unten geschoben wird. Dabei werden die *DE MORGAN* Regeln angewendet.

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



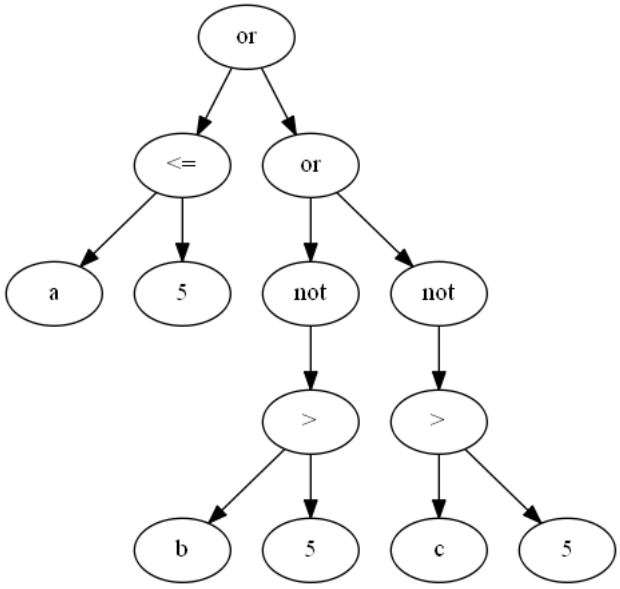
Umwandlung Teil 1:

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



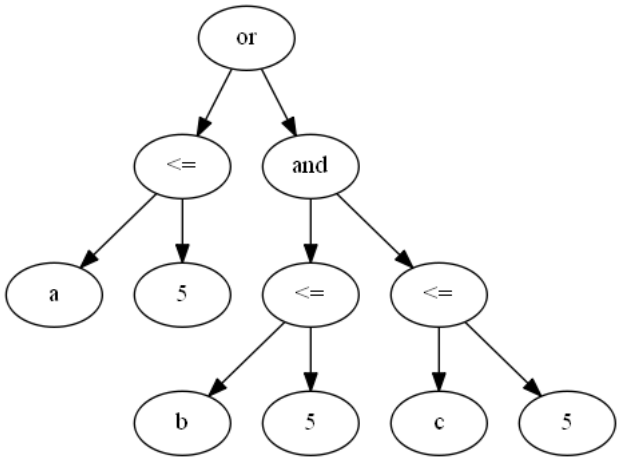
Umwandlung Teil 2:

```
((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))`

```
    A BETWEEN B AND C      →  A >= B AND A <= C
       SELECT ALL          →  SELECT
    ORDER BY VAR ASC       →  ORDER BY VAR
EXISTS (SELECT A,B,C ...)  →  EXISTS (SELECT 1 ...)
```

Abbildung 3.2: Entfernen von syntaktischen Varianten

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

3.2.4 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.

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.

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.

Eventuell ist es hier auch bereits möglich Terme, die nur aus numerischen Konstanten bestehen zu Ersetzen durch das jeweilige Ergebnisse. So könnten arithmetische Operationen bereits ausgeführt und Vergleiche, die nur aus numerischen Konstanten bestehen, durch entsprechende Wahrheitswerte ersetzt werden.

3.2.5 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.

Hinzufügen implizierter Formeln

Die Anfrage des Lernenden kann trotz zahlreicher Umformungen noch einige komplizierte Bedingungen enthalten, die wir mit bisherigen Methoden nicht entdecken konnten. Ein wichtiger Teil dabei sind transitiv-implizierte Bedingungen. Finden wir beispielsweise in der Musterlösung die Bedingung `A = C AND B = C` und der Student schreibt `A = B AND B = C`, so handelt es sich um semantisch äquivalente Aussagen. Damit unser Ansatz funktioniert, ist es also notwendig transitiv-implizierte Formeln immer hinzuzufügen. Wir bilden also im gewissen Sinne den transitiven Abschluss über den Operatoren $\{=, >, <\}$.

Beim Ansatz der Standardisierung mit Sortierung ist ein Betrachten von symetrischen Implikationen unnötig. Wie im Abschnitt Sortierung noch erläutert wird, werden zwei Bedingungen `A = B` und `B = A` nach Sortierungsregeln beide zu `A = B` standardisiert.

Beschränkung der Operatorenvielfalt

3.2.6 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.

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	OR	AND
$order(r)$	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 absteigend angeordnet.

3.3 Anpassung durch elementare Transformationen

3.4 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.

3.4.1 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.

3.4.2 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.

3.4.3 unnötiges DISTINCT

TODO: Algorithmus für unnötigen DISTINCT

3.4.4 unnötiger JOIN

TODO: Algorithmus für unnötigen Join

4 Verwendete Software

4.1 SQL Parser

4.1.1 über den SQL Parser: ZQL

Auf der Webseite vom [ZQL] 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 [JavaCC] 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

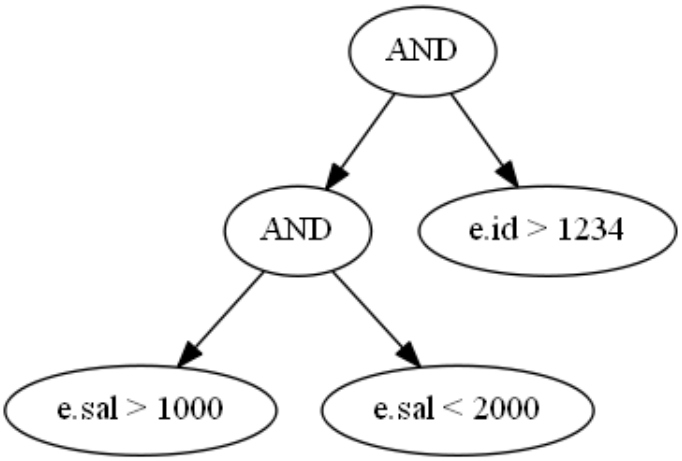


Abbildung 4.1: WHERE-Bedingung in üblichen Syntaxbäumen

- `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))
```

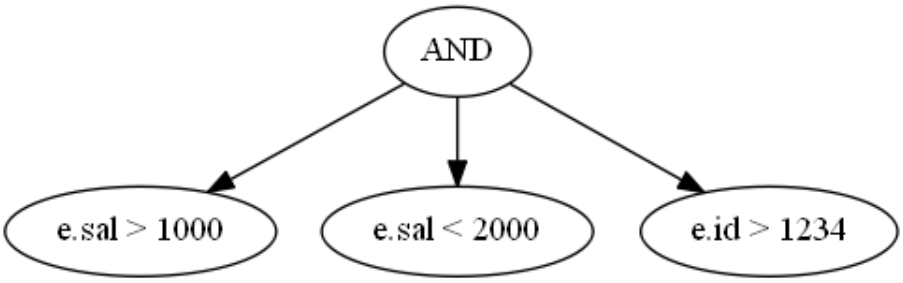


Abbildung 4.2: WHERE-Bedingung geparst mit ZQL

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.

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

[JavaCC] http://www.javacc.org

[ZQL] http://zql.sourceforge.net/

22

22

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